Taichi Accelerated Island-Based Genetic Algorithm for device agnostic parallelization in solving Combinatorial NP-Hard Problems

Arsalan Hussain, Asadulllah Chaudhry, Burhanuddin Aliasghar Ezzi, Shayan Shoaib Patel

***Abstract*—Genetic algorithms (GAs) are widely used for global search in various fields. One promising variant is the Island Model GA (IMGA), which introduces migration between islands to explore a broader search space. However, IMGA can be slow for large-scale NP-hard problems. To speed up computation, GPUs are commonly used due to their highly parallel architecture. Previous studies have focused on GPU performance, but not solution quality. In our research, we investigate how to achieve better solutions within a reasonable time by parallelizing IMGA on GPUs. We use the TSP, knapsack problem and Graph Coloring problems as our case study. Our approach includes an efficient parallel selection operators and tunable key IMGA parameters (island count, migration strategies, migration counts etc). Balancing solution quality and execution time is a challenge we address.**

I. Introduction

Since their development, genetic algorithms (GAs) have gained popularity as optimization tools. Researchers have demonstrated the effectiveness of GAs in various real-world situations involving optimization, decomposition, design, and scheduling. Due to their population-based stochastic nature, GAs often require evaluating numerous potential solutions, leading to lengthy execution times [1]. However, despite being executed for a lengthy period, the issue of premature convergence remains paramount being one of the key factors in designing such an algorithm. It refers to a situation where the algorithms would settle on sub-optimal solutions very early during the optimization process hindering the process of the best sub-optimal solution.

Thus, we can utilize one of the key advantages that GAs offer is their ease of parallelization using different methodologies such as island-based or spatially structured approaches. By distributing the population into islands and allowing each island to conduct its own exploration and exploitation, premature convergence is mitigated. Each island preserves variety within sub-populations by concentrating on a particular region of the solution space. This method lowers the possibility of the algorithm deciding on less-than-ideal solutions too soon by ensuring that it investigates a wider variety of options. The algorithm is able to navigate the solution space and prevent premature convergence by using island models to balance exploration and exploitation. This results in higher-quality solutions for NP-hard problems. [2]

The aspect of parallelization would be tackled by the aspect of utilizing GPU which is mostly held at an idle state. Thus, we can leverage the highly parallel architecture of GPUs to accelerate the execution of the island model genetic algorithm, improving efficiency and reducing computation time [2].

However, the exploration of GPU parallelization for Genetic Algorithms (GAs) within the context of island models involves using the parallel processing power of GPUs to increase the performance of genetic algorithms. By distributing the population into islands and utilizing GPUs for parallel computation, the algorithm can explore the solution space more efficiently and effectively.

This would be achieved by distributing multiple islands across the GPU cores simultaneously. This would enable the algorithm to exploit the full potential of parallelization by processing multiple islands concurrently, leading to faster computation and improved scalability. By adopting a more distributed approach that assigns different GPU cores to handle various islands simultaneously, the algorithm can benefit from increased parallelization and computational efficiency.

Moreover, it even allows for faster convergence towards optimal solutions by conducting simultaneous computations across multiple islands. GPU parallelization optimizes the execution of island model genetic algorithms, improving scalability, speed, and solution quality in comparison to traditional CPU-based implementations.

II. Literature Review

Island Model Based Genetic Algorithm is on of the most popular and efficient ways of parallelizing evolutionary algorithms. IMGAs work by running multiple islands concurrently using threading. IMGAs are able to give better quality results than traditional EAs since they maintain high diversity using migration. [3] [4]

There are several migration strategies, however, not every strategy may give optimal results. For example, Whitely [4] underscore that using a global migration strategy can result in overall high level of genetic similarity between populations. Other migraion strategies include ring based migration, LCS and Hamming distances based migration strategies.

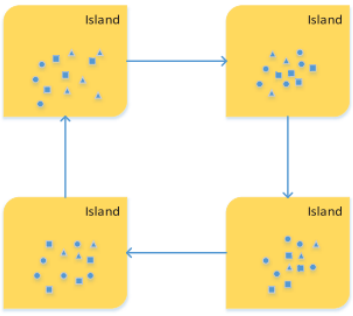


Fig. 1: Ring migration

In ring based migration, best individuals migrate to neighbouring islands and shown in the figure above. Whereas in LCS, we compare the best individuals of each island to other islands and migrate the best individual from an island to an island with which it has the Longest Common Subsequence (LCS) in its genome. Hamming distance approach works in a similar fashion, we migrate the best individuals from one island to the island with which they have the least hamming distance.

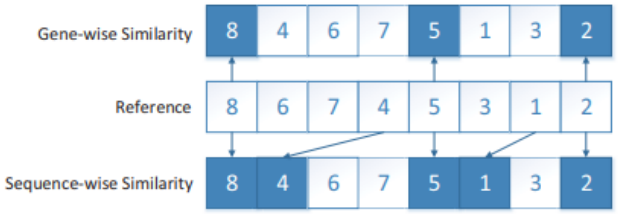


Fig. 2: Longest Common Subsequence and Hamming Distance

Ohira's [5] research uses GPU architechture with the Island Model Genetic Algorithm, introducing the advantage of shared memory to the IMGA. The research also demonstrates minimal computation time increases with more islands in the GPU implementation, showcasing the scalability and benefits of GPU parallelization for genetic algorithm optimization in ordered problems by leveraging CUDA, the study aims to harness the parallel processing capabilities of GPUs to accelerate the execution of IMGA. By incorporating sequence-based clustering to organize individuals based on genetic similarity, the study manages population diversity and guides the search process effectively within the solution space.

In similar researches [2], we find that using CUDA GPU architecture can accelerate IMGAs. However, slight variations of implementations exist between these researches. Some researches have also explored IMGA using a multi-GPU cluster [1]. The study presents an island-based genetic algorithm implementation where each GPU evolves a distinct island. By utilizing CUDA warps and the MPI interface for genetic material exchange, the research aims to eliminate thread divergence and enhance the scalability and performance of genetic algorithm computations on GPU clusters.

III. Methodology

Our approach utilizes each thread, each Island conducts local search on the initial randomly generated population. By using multiple Islands, we are able to search multiple solutions concurrently. We maintain a diverse global search by utilizing migration strategies that help each island to maintain its niche and prevents overlapping between different islands. The cost of migration can derail the overall performance, as given in other IMGA approaches which perform migrations for all Islands. Our approach on the other hand, employs migration on a single thread since memory is logically accessible to each thread (the implementation details section provides more details on TLS). Different crossover and mutation strategies are employed that are dependant on the problem we are trying to solve. In our implementation, we use IMGA to get optimal solutions of the **Travelling Salesman Problem (TSP)**, **Knapsack 0/1** and **Graph Coloring Problem**. We use these diverse problems to effectively test our strategies and provide a generic implementation.

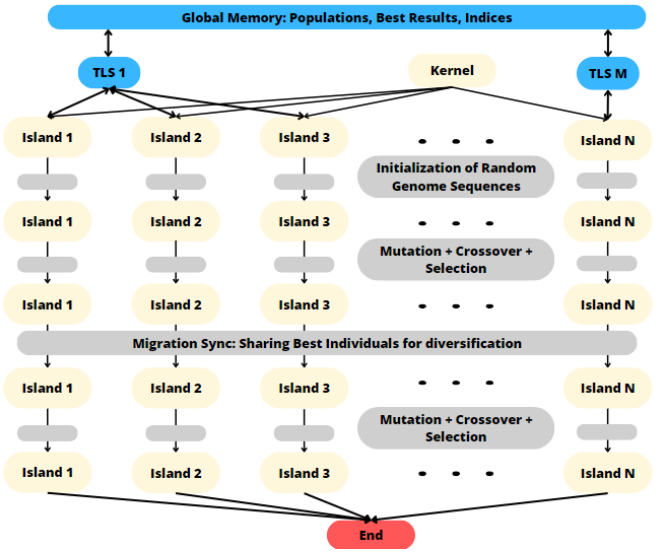


Fig. 3: Abstract Diagram of IMGA approach taken in this research

A. Knapsack Problem

1) Problem Introduction: The Knapsack Problem is a quintessential combinatorial optimization challenge where the objective is to maximize the value of items packed into a knapsack without exceeding its weight limit. It models real-life scenarios like cargo loading where one must optimize the selection of diverse resources under strict constraints.

2) Problem Representation: Initialization involves creating an initial population of chromosomes, where each chromosome represents a potential solution to the Knapsack Problem. Each gene in a chromosome indicates whether a specific item is included in the knapsack (1) or not (0). This process

typically starts by randomly generating a population of these chromosomes, ensuring a diverse range of solutions. This diversity is critical for exploring various parts of the solution space and avoiding premature convergence to local optima.

3) Fitness Function: The fitness solution for the Knapsack Problem is to maximize the total profit of items selected to be packed in the knapsack. The fitness of a chromosome (a potential solution) is calculated based on the total profit of the selected items, ensuring it does not exceed the weight capacity of the knapsack. If the weight limit is exceeded, the fitness is penalized, typically set to zero or a negative number to discourage selection.

The fitness function for the Knapsack Problem evaluates the total value of the selected items while ensuring the total weight does not surpass the knapsack's capacity. This function is crucial as it quantifies the quality of each solution, allowing the genetic algorithm to effectively compare and rank different solutions based on their fitness scores.

4) Genetic Algorithm Workflow: To implement the Knapsack problem, we will utilize a Genetic Algorithm framework that iteratively evolves a population of candidates, enabling us to reach an optimized solution.

Initialization Initialization involves generating an initial population of potential solutions, where each solution or chromosome represents a set of items selected to be included in the knapsack. This is typically achieved by initializing chromosomes randomly, ensuring a diverse pool of solutions. This diversity is critical for the robust exploration of the solution space, enhancing the genetic algorithm's ability to explore various combinations and avoid local optima early in the process.

Parent Selection Parent selection is a pivotal component in genetic algorithms, directly influencing the genetic makeup of subsequent generations. This process selects which chromosomes (solutions) from the current population will contribute to the next generation. Effective parent selection strategies maintain a balance between selecting high-fitness individuals to improve average solution quality (exploitation) and preserving genetic diversity within the population to explore new areas of the solution space (exploration).

Crossover Crossover is the genetic algorithm's mechanism to combine genetic information from two parent chromosomes to produce offspring. This process is crucial as it allows the algorithm to potentially combine beneficial traits from two good solutions to create even better ones. In the Knapsack Problem, common crossover techniques like one-point or two-point crossover slice the parent chromosomes at specific points and swap their segments to generate new offspring. This method not only perpetuates beneficial traits found in the parents but also introduces new combinations that may surpass any of the parent's fitness, driving the evolutionary process forward.

Mutation Mutation introduces random changes to an offspring, exploring new genetic configurations and preventing the algorithm from stagnating at local optima. For the Knapsack Problem, mutation might involve toggling the inclusion state of an item in the knapsack, thereby exploring alternative configurations that might yield a higher profit or fit better within the weight limit.

Survivor Selection Survivor selection determines which chromosomes remain in the population to form the next generation. This selection is crucial for retaining superior solutions and removing weaker ones, effectively shaping the future genetic structure of the population. The methods used for survivor selection often mirror those used in parent selection, focusing on maintaining a healthy diversity while enhancing the overall fitness of the population.

B. Traveling Salesman Problem

1) Problem Introduction: The Traveling Salesman Problem (TSP) seeks the shortest possible route that visits a set of cities and returns to the origin city, visiting each city exactly once. The problem is NP-hard, implicating no known polynomial-time solution. It's significant in logistics and scheduling, modeling real-world routing and sequencing issues.

2) Problem Representation: The Traveling Salesman Problem (TSP) in our code involves finding the shortest route that visits each city once and returns to the starting point, a classic example modeled as a graph where cities are nodes and paths between them are edges with associated distances. The implementation uses a genetic algorithm where each solution is a chromosome representing a tour, with cities encoded as genes in a sequence that determines the visitation order. This permutation encoding ensures each city is visited exactly once, making it a direct and efficient representation for TSP. This setup allows the genetic algorithm to evolve solutions over generations, using operations like crossover and mutation to explore and exploit the search space, refining tours to approach the shortest possible route effectively.

3) Fitness Function: The fitness function evaluates the total Euclidean distance of a tour. The distance for each pair of consecutive cities is calculated using the Euclidean formula, providing a quantitative measure of a tour's quality. By minimizing this distance, the GA directly targets TSP's primary objective, influencing all genetic operations by enabling comparisons between different chromosomes.

4) Genetic Algorithm Workflow: The following implementation adheres to the principles and foundation of the genetic algorithms, and allows the best solution to be reached and developed over multiple generations.

Initialization Initialization involves randomly generating a set of valid tours (solutions). Each "island" in a multi-island approach starts with a unique population, enhancing genetic diversity across the "archipelago." This diverse initial population prevents premature convergence on local optima and encourages broad exploration of the solution space.

Parent Selection Parent selection plays an important role in guiding the evolutionary process toward optimal or nearly optimal solutions in the setting of the Travelling Salesman Problem (TSP).

When it comes to TSP, parent selection effectiveness determines how well the algorithm can both explore various

route configurations and intensify its search around viable solutions. The algorithm can prevent local optima and increase the likelihood of locating the shortest path by choosing parents that contribute to a more varied or superior set of routes.
Because it affects the genetic diversity within the population, the parent selection mechanism is vital. This variety is essential for investigating various TSP routing options. Insufficient investigation could lead to the algorithm settling down on less-than-ideal pathways too soon. On the other hand, efficient exploitation guarantees that the best-found paths are honed, which may result in little steps towards the ideal answer.
Parent selection thus influences the success rate of finding the shortest path as well as the total computational cost in TSP, in addition to contributing to genetic diversity and convergence efficiency.

Crossover

Crossover recombines genetic information from two parents to produce offspring. The segment-based crossover ensures all cities are visited once by inheriting a segment from one parent and filling the rest with cities from the other parent in their order, avoiding duplicates and preserving good partial routes.

Mutation

Mutation introduces random changes to an offspring, exploring new genetic configurations and preventing the algorithm from stagnating at local optima. Swapping two cities can drastically alter the route's structure, potentially discovering shorter paths not previously considered.

Survivor Selection

Survivor selection determines which individuals continue to the next generation, impacting the future population's genetic makeup. Methods used reflect those in parent selection, maintaining diversity and focusing evolutionary pressure on promising areas of the solution space.

C. Graph Colouring Problem

1) Problem Introduction: Given an undirected graph G = (V,E), the goal is to colour each node in the graph in such a way that no two adjacent nodes (connected by an edge) have the same colour. Although there are many variations of this problem, we try to find the least number of colours needed to colour the graph keeping in mind the above rule. We know this problem to be NP-Complete [6] and although there exist known solutions for some standard graphs, generally finding such solutions is incredibly computationally taxing. Thus we use an Evolutionary Algorithm to try to best approximate the most optimal solution.

2) Problem Representation: We represent each candidate solution to the GCP using an adjacency matrix to capture the connectivity between vertices in the graph. Each vertex is assigned a unique index, and the presence of an edge between two vertices is denoted by a non-zero entry in the matrix. Additionally, we keep track of the color assigned to each vertex within the chromosome. Thus, each chromosome, representing a potential solution, keeps track of the structure of the graph as well as associating each vertex with a specific color.

3) Fitness Function: The fitness function evaluates candidate solutions (chromosomes) by penalizing violations of the colouring constraint, where adjacent vertices must be assigned distinct colours. Specifically, the function computes a penalty score based on the number of conflicts in the colouring, reflecting the extent to which neighbouring vertices share the same colour. Our objective is to minimize this penalty score, thereby achieving valid and efficient colouring of the graph.

D. Genetic Algorithm Workflow

The study employs a Genetic Algorithm framework to solve the Graph Colouring Problem, utilising the basic principles of natural selection and genetic operators (recombination, mutation) to iteratively evolve a population of candidate solutions towards optimal or near-optimal colorings.

Initialization: We initialize a population of chromosomes randomly, where each chromosome represents a potential coloring solution for the graph. The initialisation is constrained to use a specified maximum number of colours in colouring the graph. Each chromosome encodes both the coloring assignment for each vertex and the corresponding color index.

Parent Selection: Individuals are selected from the population to undergo genetic operations, such as crossover and mutation. Selection methods such as random selection, tournament selection, and fitness proportional selection are utilised to introduce stochasticity to the selection process and we fine-tune our variables governing these selection methods to find a good balance between exploitation and exploration of the solution space.

Crossover: Parents undergo crossover, generally an operation where segments of genetic material are exchanged between parents to generate offspring. This study utilises a one-point crossover mechanism, where a random crossover point is selected along the chromosome, dividing it into two segments. Offspring are then produced by swapping the segments between the parent chromosomes at the crossover point.

Mutation: The previously produced offspring chromosomes undergo mutation, introducing small random changes to their genetic makeup. In a typical GA, Mutation works to prevent premature convergence to suboptimal solutions, operating at the level of individual genes. A random mutation rate determines the probability of mutation for each gene. In our implementation, we check each pair of vertices and if they are adjacent and they have the same color, a mutation occurs where a random color is assigned to the corresponding vertex, thereby introducing novel genetic variation.

Survivor Selection: The mutated offspring are added to the main population, and from this new pool we reduce the population to get back to the original number of population members we had. We implement different survivor selection methods such as rank selection and truncation selection to retain high-quality individuals while allowing for the exploration of solution space.

The GA iterates a predefined number of times or until a termination criteria is met, where for us that is when the

algorithm finds a best fit individual which has zero penalty i.e is a perfect GCP solution but since we are trying to minimise the number of colours we use to solve the problem, we restart the algorithm with a different number of maximum colours allowed.

IV. Island Model Evolutionary Algorithm

Recent research has brought a lot of new information and the ability to parallise the traditional GA using a variant of the Genetic Algorithm, The Island Model Genetic Algorithm (LIMGA). It treats each island as a living entity. We divide a large global population to smaller 'islands', where each island's population undergoes the complete evolutionary process.[7]. Each island then contains it's own isolated population, with all islands being initiated randomly by elements from the solution phase. These islands evolve in parallel, allowing us to divide the computation power required among different processors. Every few generations, we share genetic material between the islands in a process called **Migration**. Although there exist multiple migration strategies, one of them could be selecting the a few random individuals in an island and then migrating them to the next and so on until each island has gone through migration once.[7]. This maintains diversity and prevents premature convergence for any one island, helping exploration. An effective IMGA should maintain a balance between the number of islands, the population of each island and the number of generations to control the computational power IMGA uses.

The way we migrate individuals can lead to changes in how soon the global population converges, how computationally expensive it is and in the amount of diversity within the system.

A. Ring-Based Migration:

In ring-based migration, islands are organized in a ring structure where each island is connected to its adjacent neighbors. During migration, a predefined number of individuals are selected from each island and sequentially migrated to the neighboring island in the ring. This process continues until each island has undergone migration once, ensuring a uniform distribution of genetic diversity among islands, facilitating exploration across the entire population

B. Best Individual Migration:

Best individual migration strategy involves each island selecting its best-performing individual based on fitness. These selected individuals are then transferred to the neighboring island, typically replacing the least fit individuals in that island's population. This strategy prioritizes the propagation of the most promising solutions across islands, potentially accelerating convergence towards optimal solutions.

C. Gene-Based Similarity Migration:

Gene-based similarity migration focuses on exchanging genetic material between islands based on sequence-wise similarity. This strategy identifies commonalities in genetic sequences across islands, such as shared genetic traits or patterns. Individuals with similar genetic sequences are selected for migration between islands, aiming to propagate beneficial genetic traits and facilitate exploration of diverse solution spaces. We establish similarity by calculating the sequence-wise similarity using the Least Common Subsequence algorithm between centeroids of the two islands.[7]

V. Implementation Details

The IMGA implementation provided here utilizes the Taichi programming language, a domain-specific language for high-performance computing. Taichi offers efficient data structures and parallel execution capabilities, making it well-suited for evolutionary algorithms like IMGA. While Taichi does not provide support for dynamic parallelism, but since our methodology did not employ dynamic parallelism by a reasonable factor, we chose to use Taichi for its powerful features. The implementation is done in a way so that it is easy to reuse code when applying the solution to different problems. Therefore the implementation is divided into two main parts, a generic evolutionary algorithm implementation and the implementation of specifics of the problem, like data input, defining an individual and crossover and mutation functions. This approach lets us use the same code on multiple problems.

The abstract level methodology shows Thread Level Storage or (TLS) which are an optimization feature in Taichi. Even though we logically divide each Island into a thread, Taichi on the other hand may or may not invoke a thread for each Island, since it does not give much control to the user, it will try to optimize global memory accesses using Grid Stride Loops. In our implementation therefore, we see that get good performance without configuring shared memory.

For this research, we used a workaround for random numbers in Taichi Kernel since Taichi's own random number generator was not good enough to compute Genetic Algorithms. We generated a few sequence of random numbers using Numpy and loaded them during runtime from a file into Taichi fields which gave us access to better random numbers in Taichi scope.

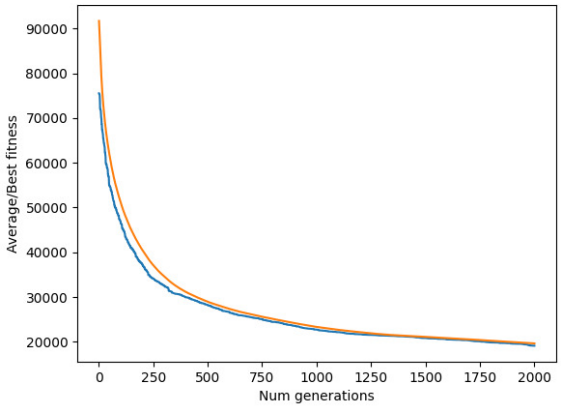
Populations, selection results, and best indices are defined (and implicitly initialized by Taichi) as Taichi fields which can be used in both Python and Taichi scope, and provide efficient memory management with parallel computation. This design decision was taken due to the limitation in taichi by which it is illegal to use array-like datastructure of structs inside a struct, also referred as **dataclass** in Taichi's jargon. Taichi kernel functions are used for parallel execution of key algorithmic components, such as running generations and migration. Kernel functions are decorated with **@ti.kernel** and operate on Taichi fields. Thread synchronization is ensured using Taichi's **ti.simt.block.sync()** function. This ensures that migration occurs collectively across all islands.

The user-defined initialization is done inside Taichi kernel before running the generations. The kernel is responsible for running the islands on each thread. The most efficient GPU

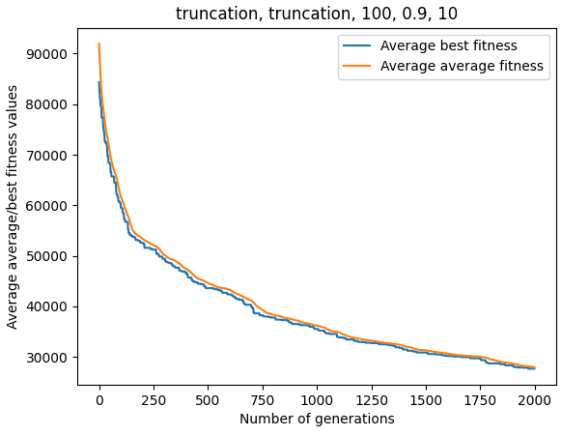
configuration is the one that Taichi figures out at runtime therefore it is suggested not to define configurations beforehand.

Various selection functions are implemented including truncation selection, fitness-proportional selection. Truncation selection is used for both parent selection and survivor selection, while fitness-proportional selection is employed for parent selection. These functions select individuals based on their fitness values. The crossover and mutation functions are defined as methods of the individuals class as they are usually problem specific.

Periodic migration of individuals between islands is the crux of the Island-based approach and facilitates information exchange and diversification. Three migration strategies are implemented: ring migration, LCS-based migration, and Hamming-based migration. These strategies determine how individuals are exchanged between islands. The key difference in our implementation compared to the traditional IMGA implementation is that instead of adding the individual to be migrated to its corresponding island, we instead randomly replace one of the already existing individuals of that island with the individual to be migrated. Another difference in our migration implementations is that instead of halting all threads to a stop and waiting until one threads migrates individuals for all islands and then synchronizing threads again, we instead synchronize threads once then let each thread migrate their specific individual to some other island. This causes some non determinism as islands will migrate individuals at their own pace rather than at the same generation. But we observed that this is one of the rare cases where non determinism turns out to be beneficial instead of destructive. Our algorithm performs faster and gives solutions with higher fitness when this method of migration is used instead of the normal approach. The main loop of the algorithm executes the evolutionary process for a specified number of generations.



(a) TSP using IMGA



(b) TSP using traditional GA

Fig. 4: Comparison of TSP solutions

VI. Experimental Results

In all our experiments, we have used ring migration as our migration strategy, truncation selection for both parent selection and survivor selection (For small number of generations, this configuration performs the best). We have set the mutation rate to 0.9 and population size per island to 100.

Ring migration, surprisingly outperformed all other methods of migration that we used. We hypothesize, this may be due to our usage of truncation selection for both parent and survivor selection functions because we know that in lower number of generations, exploitative strategies perform better.

We used Google Colab for all our experiments. The specifications for which are NVIDIA T4 GPU, Intel Xeon CPU @2.20 GHz, 13 GB RAM, and a Tesla K80 accelerator.

A. Non-Island based vs Island Based Solution Quality Comparison

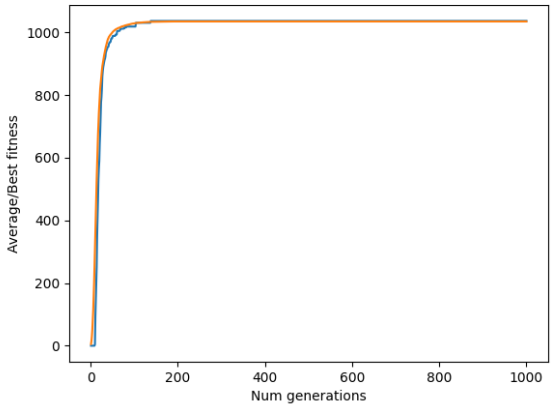
We use Qatar cities dataset [8] for TSP and a standard dataset [9] for Knapsack problem.

From Figure 5 and 6, it is evident that our IMGA outperforms the standard GA. This is indicated by the better fitness values achieved by the IMGA over fewer generations. Our IMGA also converges to a far better solution in the end; 19615 in TSP (optimal is 9352), 1024 in Knapsapck (optimal is also 1024). We see that it converges much faster in the case of Knapsack. From this we can establish that some problems benefit more from Island model's diversification of the solution space. Overall, these results suggests that our IMGA is more efficient at finding optimal or near-optimal solutions to combinatorial problems. This indicates that our island model is working correctly as our results are similar to those mentioned in the literature review.

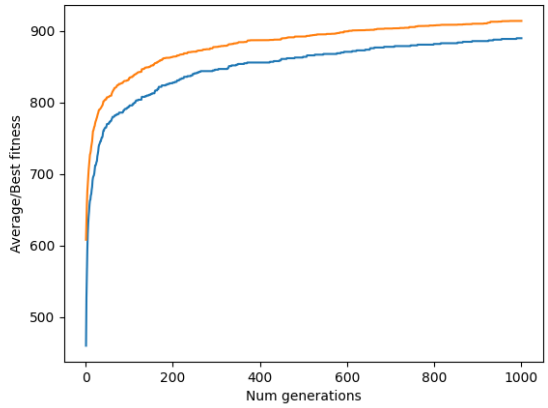
Note that we haven't attached plots for the Graph Coloring problem because we weren't able to implement it using Taichi due to time constraints. Though, its pure pythonic implementation is available in our repository.

B. Multithreaded CPU vs GPU time comparison

We have averaged out the execution times over 5 runs for each number of islands. The execution times have been skewed due to variability in kernel launch execution times of TaichiLang. Keeping the same configuration, according to



(a) Knapsack using IMGA



(b) Knapsack using traditional GA (In this graph, orange represents best whilst blue represents average mistakenly)

Fig. 5: Comparison of knapsack solutions

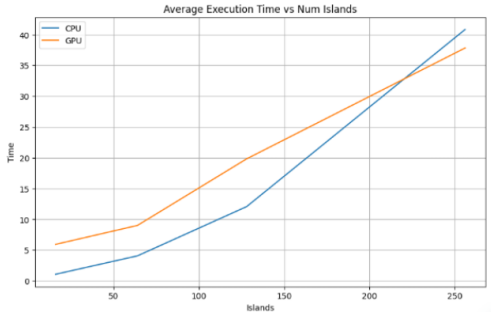


Fig. 6: Execution time(s) vs GPU and CPU based implementations for TSP

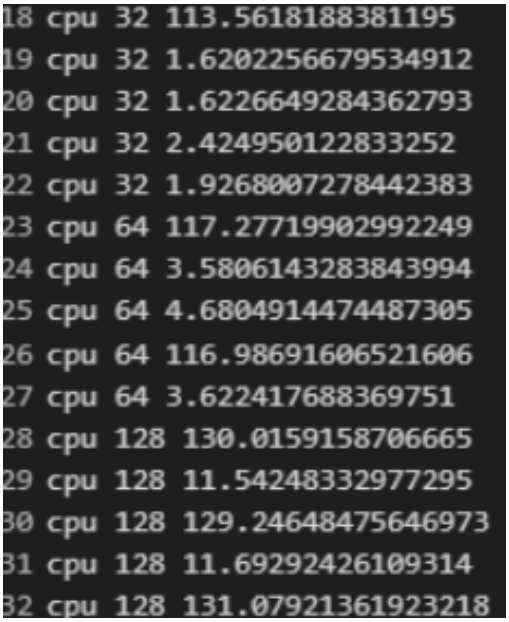


Fig. 7: Variability in execution times (s)

Figure 6 some launches only lasted a few seconds before completion while others ran for around two minutes. We suspect this is due to dynamically allocation of Taichi Vectors within device functions as it gives you a warning when you allocate one that has more than 32 elements. We have allocated numerous such vectors with sizes increasing up to 194. This is a limitation of TaichiLang.

Despite the time values not representing the actual time required for the computation, by averaging out over multiple runs we retain the trend between GPU and CPU implementations which is that the GPU implementation performs significantly better when number of islands are set to 250 or above. This is to be expected due to the large number of threads available present in GPUs given that each island maps to a thread.

VII. Conclusion

In this paper, we present an open source TaichiLang accelerated Island Model genetic algorithm. Harnessing the abstraction of TaichiLang, our implementation accelerates the IMGA using both multithreading on CPU and hyper parallelism via GPU. We demonstrate its effectiveness in solution quality by comparing its performance in solving problems with a traditional GA implementation. We also demonstrate the speedup of the GPU in comparison with the multi-threaded CPU using our unique architecture for parallelizing the IMGA. We have further explored TaichiLang's weakness with dynamically allocating Vectors and Matrices of large sizes. We leave it to readers to come up with efficient methods work around these weakness or for the maintainers of TaichiLang project to implement improvements in subsequent releases. There is a lot of work that can be done further in IMGAs; the first being optimizing this implementation further by removing dynamically allocated vectors, then generalizing the algorithm such that it can become a library that can be easily mapped to any problem. An open source implementation of the sequence based clustering as introduced in [5] is highly needed as the results presented in that paper are extremely impressive. Lastly, there is a big gap in research for using IMGA to solve non-combinatorial optimization problems.

VIII. Acknowledgements

- The maintainers of the Taichi-Lang project [10]
- Free GPU compute resources on Google Colab

IX. Appendix

A. Code

The code for this research paper is available on this github repository here. A guide is available on running the code on via colab notebook here

References

[1] J. Jaros, Multi-GPU island-based genetic algorithm for solving the knapsack problem, 2012 IEEE Congress on Evolutionary Computation (6 2012).

[2] X. Sun, P. Chou, C.-C. Wu, L.-R. Chen, Quality-Oriented Study on Mapping Island Model Genetic Algorithm onto CUDA GPU, Symmetry 11 (3) (2019) 318.

[3] Y.-J. Gong, W.-N. Chen, Z.-H. Zhan, J. Zhang, Y. Li, Q. Zhang, J.-J. Li, Distributed evolutionary algorithms and their models: A survey of the state-of-the-art, Applied Soft Computing 34 (2015) 286–300.

[4] D. Whitley, S. Rana, R. B. Heckendorn, The Island Model Genetic Algorithm: On Separability, Population Size and Convergence, https://hrcak.srce.hr/150198 (mar 30 1999).

[5] R. Ohira, M. S. Islam, Gpu Accelerated Genetic Algorithm with Sequence-based Clustering for Ordered Problems, 2020 IEEE Congress on Evolutionary Computation (CEC) (7 2020).

[6] M. R. Garey, D. S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, first edition Edition, W. H. Freeman, 1979.

[7] R. J. Ohira, M. S. Islam, Gpu accelerated genetic algorithm with sequence-based clustering for ordered problems, 2020 IEEE Congress on Evolutionary Computation (CEC) (2020) 1–8.
URL https://api.semanticscholar.org/CorpusID:221567976

[8] Solving TSPs, National Traveling Salesman Problems, Website, no date.
URL https://www.math.uwaterloo.ca/tsp/world/countries.html

[9] Lolik-Bolik, Lolik-Bolik/KNAPSACK_PROBLEM_0-1, GitHub, no date.
URL https://github.com/Lolik-Bolik/Knapsack_problem_0-1

[10] Y. Hu, T.-M. Li, L. Anderson, J. Ragan-Kelley, F. Durand, Taichi: a language for high-performance computation on spatially sparse data structures, ACM Transactions on Graphics (TOG) 38 (6) (2019) 201.