Smart Grid Energy Allocation using Genetic Algorithm

Table of Contents

Table of Figures

1. Introduction

1.1 Project Background

Electricity is the foundation of modern life; with growing technological advancements, the use of electricity has grown **exponentially**. Due to this ever-growing demand and daily life dependence on electricity, it has become more essential than ever to manage and optimize energy distribution. As more electricity is being used, various new electricity sources are being developed. To manage energy more efficiently, modern cities are moving towards smart grid energy distribution solutions. These power grids must effectively manage and deliver electricity across various networks, changing demands, and multiple suppliers. Many factors such as line losses, storage limits, weather etc. (Zohuri & McDaniel, 2019)

This project aims to develop a parallelized Genetic Algorithm-based optimization system for energy distribution, taking various constraints into account in a smart grid system. The application of genetic algorithms allows natural evolution, which enables the system to produce various optimization solutions that traditional mathematical and computational systems might fail to.

1.2 Problem Statement

There are various challenges that smart grids must contend with to achieve efficient energy distribution, including the shift in demand, the presence of multiple power sources, energy loss in electrical lines, the limited capacity of power storage, and weather conditions. Market dynamics cannot be modelled by conventional optimization methods, which lack the speed to develop and respond to such changing phenomena. Thus, what is required is a more intelligent, more flexible approach to holding all these mobile variables in the air and mainstreaming energy into its most fruitful applications. The current project aims at developing a parallelized Genetic Algorithm-based system able to take energy management to even greater heights of performance, efficiency, and flexibility.

1.3 Project Objectives

The following are the primary objectives of this project:

- To implement a genetic algorithms-based solution for smart grid optimization.
- To design a parallel computing solution for the implemented solution to enhance performance and scalability.
- To implement greedy algorithms for performance comparison with the genetic algorithm.

1.4 Dataset Description

The dataset used for this project is a synthetically prepared dataset which aims to represent real life-like grid data. This dataset contains important parameters which are essential for efficient distribution of energy. In this dataset each row represents a node, energy source, time, demand, capacity, loss, cost etc. Features relevant to this project are:

Feature	Description
Hour	Hour represents the hour of the day
Source	Source represents the source of energy
Node	Node Represents an energy destruction node
Demand_kW	Represents demand per node in kilowatt
Cost_per_kwh	Represents cost per unit for a node given a source
Capacity_kW	Represents the amount of energy that can be received by a node.
LossRate	Represents energy lost during transmission

This dataset contains **721** entries with no missing value.

2. Mathematical Foundation

2.1 Smart grid distribution as a constrained optimization problem

The energy-allocation problem in a smart grid is a type of optimization problem whose aim is to find the optimal way to distribute electricity to various consumers with different energy sources over a specific twenty-four hours. The demand for power is not constant: it changes throughout the day, different working energy sources have a fixed capacity and cost, and the system should make sure that all the needs of consumers are served, avoiding oversupply and losses. At the same time, no single energy source can be overused. In this spirit, a thorough allocation plan must balance several constraints, such as capacity and limits of supply, demand, and cost considerations, to guarantee a secure, effective, and fair supply of energy. The ability to address this issue is an inseparable part of the proper management of energy systems in contemporary smart cities whose electricity demands keep growing, and sustainability and efficiency are top-priority issues. (Mary & R.Rajarajeswari, 2014)

2.2 Objective Functions and Constraints

2.2.1 Objective Functions

Objective functions defined in this project to ensure proper optimization of genetic algorithms are:

- **Total Energy Cost Minimization**: The main objective of this objective function is to minimize the total cost of energy supplied. The sum of the respective allocations is multiplied by the sum of respective costs.

$$\min \sum_{t=1}^{T} \sum_{n=1}^{N} \sum_{s=1}^{S} c_{t,s} \cdot x_{t,n,s}$$

Here, T = total time slot, N = number of nodes, S = number of energy sources, $c_{t,s}$ represents cost per unit and $x_{t,n,s}$ represents energy allocated from source s to node at time t.

- **Loss minimization:** The main objective of this function is to penalize over supply of energy. By penalizing energy wastage, the loss of energy can be minimized.

$$\min \lambda_2 \sum_{t=1}^{T} \sum_{n=1}^{N} \blacksquare \max\left(0, \sum_{s=1}^{S} x_{t,n,s} - d_{t,n}\right)$$

Here, λ_2 represents penalty for energy wastage, $x_{t,n,s}$ represents energy allocated from source s to node at time t and $d_{t,n}$ represents demand at node n at time t.

- **Demand Fulfillment**: The main objective of this function is to apply penalty to insufficient fulfillment of energy demand at certain node and time.

$$\min \lambda_1 \sum_{t=1}^{T} \sum_{n=1}^{N} \blacksquare \max\left(0, d_{t,n} - \sum_{s=1}^{S} x_{t,n,s}\right)$$

Here, λ_1 represents penalty for unmet demand, $x_{t,n,s}$ represents energy allocated from source s to node at time t and $d_{t,n}$ represents demand at node n at time t.

2.2.2 Constraints

For proper optimization some constraints need to be applied to limit and regulate the genetic algorithm. The constraints defined are:

- **Energy Capacity Constraint**: The objective of this constraint is to make sure that no source should exceed its generation limit.

$$\sum_{n=1}^{N} x_{t,n,s} \leq C_{t,s} \quad \forall\, t, s$$

Here, $x_{t,n,s}$ represents energy allocated from source s to node at time t, $c_{t,s}$ represents cost per unit. $\forall\, t, s$ means that this condition holds all at every time and for every source.

- **Non-Negativity Constraint**: This constraint ensures that under each condition the energy allocated is always positive.

$$x_{t,n,s} \geq 0 \quad \forall\, t, n, s$$

Here, this equation represents that allocated energy from sources s, node n and time t is always positive.

- **Demand Fulfillment Constraint**: This constraint ensures that energy given by all nodes at time t must be equal to or greater than the demand.

$$\sum_{s=1}^{S} x_{t,n,s} \geq d_{t,n} \quad \forall\, t, n$$

- **Line Loss Constraint:** This constraint ensures that the total line does not exceed a predefined limit.

$$[\alpha \times \sum_{t=1}^{T} \sum_{n=1}^{N} \sum_{s=1}^{S} x_{t,n,s} \leq L_{max}]$$

Here, α is the percentage of energy that might be lost during transmission and $\boldsymbol{L_{max}}$ is the maximum allowable loss.

2.3 Genetic Algorithm Feasibility

The Genetic Algorithms (GAs) are meant to resemble the processes of biological evolution to drive the quality of a set of candidate solutions in repetitive, cycles of natural selection, crossover and mutation.

- Initialization: The process starts with a randomly defined cohort of candidate solutions or hear each candidate solution is a configuration of an energy-allocation in the smart grid.
- Evaluation: Upon each parent a fitness function is applied, measuring the extent of its success relative to the minimization of cost, demand satisfaction, as well as constraints.
- Selection: Fittest individuals are kept carrying on to the next generation hence favoring reproduction of fit solutions.
- Crossover: In the process of creating children, parents provide parts of their own chromosomes bringing beneficial features that would have been lost through meiosis.
- Mutation: Random changes are made temporarily to offspring which allow them to keep diversity and explore new configuration spaces.
- Iteration: These steps repeat itself in different generations resulting in accumulative Improvement of general population.

GAs has a ability to overcome the challenges experienced by classical optimization methods and converge to optimal (or suboptimal) energy-allocation solutions in modern smart grids because they, like evolutionary algorithms, have a parallel search structure where they explore a large, multidimensional search space, as well as because they utilize evolutionary mechanics.

3. Chromosome Encoding and Fitness Function Design

3.1 Chromosome Encoding

In this Genetic Algorithm solution, each chromosome is represented in a one-dimensional array. After some preprocessing the data is converted to a three-dimensional array with dimensions being hours, nodes and sources. Each gene in the chromosome represents the amount of energy given from a particular source at a specific time.

Here the chromosome length is **NUM_HOURS x NUM_NODES x NUM_SOURCE**

When reshaped the matrix represents **allocation [hour,node,source]**

This encoding helps the genetic algorithm to optimally apportion the energy that derives resources of various sources to various nodes over all time periods and keep in consideration all the constraints and objectives given in the optimization task.

3.1 Fitness Function Design

The fitness function evaluates the quality of produced optimization solutions. In this context, combination of objective functions, constraints and penalties are considered while designing the fitness function. For this project the objective is to minimize the fitness value or the lower the better. The equation of fitness function is:

$$\text{Fitness} = \sum_{t=1}^{T}\sum_{n=1}^{N}\sum_{s=1}^{S} x_{t,n,s}\cdot C_{t,s} + 1000\cdot \max\left(0,\ \sum D_{t,n} - \sum x_{t,n,s}\right) + 1000$$
$$\cdot \max\left(0,\ \sum x_{t,n,s} - \sum D_{t,n}\right) + 1000$$
$$\cdot \sum_{t=1}^{T}\sum_{s=1}^{S} \max\left(0,\ \sum_{n=1}^{N} x_{t,n,s} - K_{t,s}\right) + 1000$$
$$\cdot \max\left(0,\ \alpha\cdot \sum x_{t,n,s} - \beta\cdot \sum x_{t,n,s}\right)$$

Here $K_{t,s}$ = **capacity of source s at time t, β = line loss factor**

Or in simple words, **fitness = total_cost + demand_satisfaction_penalty + energy_wastage + capacity_penalty + line_loss_penalty**

This is how the fitness function has been implemented in our project.

```python
def fitness_function(chromosome, demand, capacity, cost, line_loss_factor=0.03, max_line_loss_ratio=0.05):
    allocation = chromosome.reshape((NUM_HOURS, NUM_NODES, NUM_SOURCES))

    total_cost = np.sum(allocation * cost[:, np.newaxis, :])

    total_demand = np.sum(demand)
    total_supplied = np.sum(allocation)
    demand_satisfaction_penalty = max(0, total_demand - total_supplied) * 1000

    energy_wastage_penalty = max(0, total_supplied - total_demand) * 1000


    capacity_penalty = 0
    for t in range(NUM_HOURS):
        for s in range(NUM_SOURCES):
            supplied_from_source = np.sum(allocation[t, :, s])
            if supplied_from_source > capacity[t, s]:
                capacity_penalty += (supplied_from_source - capacity[t, s]) * 1000

    line_loss = total_supplied * line_loss_factor
    max_allowed_line_loss = total_supplied * max_line_loss_ratio
    line_loss_penalty = 0
    if line_loss > max_allowed_line_loss:
        line_loss_penalty = (line_loss - max_allowed_line_loss) * 1000

    #final fitness score (lower is better)
    fitness = total_cost + demand_satisfaction_penalty + energy_wastage_penalty + capacity_penalty + line_loss_penalty
    return fitness
```

✓ 0.0s

In this fitness function every violation of a constraint is multiplied by 1000 to make sure that violation of an important constraint has a major impact on total fitness.

3.3 Worked Out Example

For a worked example of the fitness score calculation, we have selected some data from our synthetically prepared dataset, hypothetical allocations have been added to make the calculation simpler:

Node	Source	Demand (D)	Capacity (K)	Cost (C)	Allocation (x)
1	Grid	67.39 kW	34.18 kW	0.34	30
2	Solar	92.75 kW	53.29 kW	0.15	50
3	Solar	90.98 kW	72.50 kW	0.07	70

By using the defined fitness function and based on the formula defined above:

Here, α 0.3 and $\beta = 0.05$

Cost Calculation:

Cost of Node 1 (Grid) = 30 x 0.34 = 10.2

Cost of Node 2 (Solar) = 50 x 0.15 = 7.5

Cost of Node 3 (Solar) = 70 x 0.07 = 4.9

Total cost = 10.2 + 7.5 + 4.9 = 22.6

Demand Penalty Calculation:

Node 1: 67.39

Node 2: 92.75

Node 3: 90.98

Total Demand = 251.12

Total allocation = 150

Unmet demand = 251.12 – 150 = 101.12

Penalty = 101.12 x 1000 = 1011120

Energy Wastage Penalty

As Total demand > Allocation no wastage

Capacity Penalty

Node 1 (Grid): 30 ≤ 34.18 = No penalty

Node 2 (Solar): 50 ≤ 53.29 = No penalty

Node 3 (Solar): 70 ≤ 72.5 = No penalty

Line loss Penalty

150 x 0.03 < 150 x 0.05

As no line is lost so no Penalty

Final Fitness score = Total Cost + Demand Penalty + Wastage Penalty + Capacity Penalty + Line Loss Penalty **= 22.6** + 101120 + 0 + 0 + 0 **= 101142.6**

4. Parallelization Strategy and Algorithm Implementation

4.1 Parallelization Strategy

The scale of data generated by a smart grid system is massive. For proper optimization of a smart grid system, the entire dataset needs to be fed to the Genetic algorithm system. In order to calculate fitness scores of each chromosome, a large amount of computational power is required. This can lead to high time and space complexities. To solve this issue, parallel computing solutions can be implemented. Application of parallel computing can make computation of genetic algorithms much more efficient.

To use parallel computing in this optimization problem, we have used **ThreadPoolExecutor** from the **concurrent.futures** module

The **ThreadPoolExecutor** creates a pool of worker threads as each thread can evaluate the **fitness score** of a different **chromosome** at the same time this **speeds up** the **process of fitness evaluation.**

ThreadPoolExecutor was chosen as the option to parallelize the fitness evaluation in the genetic algorithm because of its reliability and efficiency in the Jupyter notebook environment. **ThreadPoolExecutor** utilizes threads in comparison to process-based parallelism that can be inappropriate in notebooks since sharing data becomes hard, whereas kernel instability is possible, too. This strategy significantly speeds up the most time-consuming part of the algorithm, since it is allowing parallel fitness evaluations, but is overcoming the complexity and dangers of full-scale multiprocessing. For this reason, we have implemented **ThreadPoolExecutor** in this project.

Using the Python **ThreadPoolExecutor** method, the fitness of each of the chromosomes within the population will be calculated simultaneously. The fitness function is applied over the entire population of a given generation with the help of multiple threads and, thus, allows the algorithm to speed up the most computationally intensive part. Post evaluation, the remaining stages: **selection**, **crossover**, **mutation**, and **elitism**, are executed **serially**, as they impose only modest computational demands.

The produced results will be compared with serial implementation based on Computational usage, execution time and other metrices.

4.2 Algorithms Implementation

The following are the implementation details of genetic algorithm:

Fitness function: This function was defined to evaluate how good a optimization solution is.

- The chromosome was changed from a one-dimensional array to three-dimensional array
- Various objective functions, constraints and penalty were evaluated.
- After this a fitness score was produced by the sum of total cost and all the penalties
- In our implementation, the lower fitness score is better.

Crossover Function: This function combines two parents to create a new child

- Random crossover points were chosen using the random function.
- By combining genes of both parents, a child chromosome was created.

Mutation function: This function introduced random changes to a chromosome to introduce diversity.

Select Parents Function: This function was created to select the two best parents for the next generation of chromosomes.

- Tournament selection was used i.e. two random chromosomes were selected and the best was chosen. This process was further repeated until the required number of parents were selected.

For genetic algorithm implementation **parameters** passed were **population size = 100**, **generation = 20** and **mutation rate = 0.05**.

Serial Implementation: The fitness score of each chromosome was evaluated one by one. After computing the fitness score, parents were selected using tournament selection, then crossover and mutation were performed, and elitism was also used to select the best parents directly to the next generation. After all generations, the best chromosome, fitness history, execution time, and CPU utilization were returned.

Parallel Implementation: The fitness score of each chromosome was evaluated parallelly using **ThreadPoolExecutor**. This significantly reduced computational strain of the most computationally intensive part of the program. As a large amount of computation is done the rest of the process was implemented serially. Post fitness score computation, parents were selected using tournament selection, then crossover and mutation were performed, and elitism was also used to select the best parents directly to the next generation. After all generations, the best chromosome, fitness history, execution time, and CPU utilization were returned.

5. Performance Analysis and Result Interpretation

5.1 Parallel vs Serial Genetic Algorithm

Performance Comparison

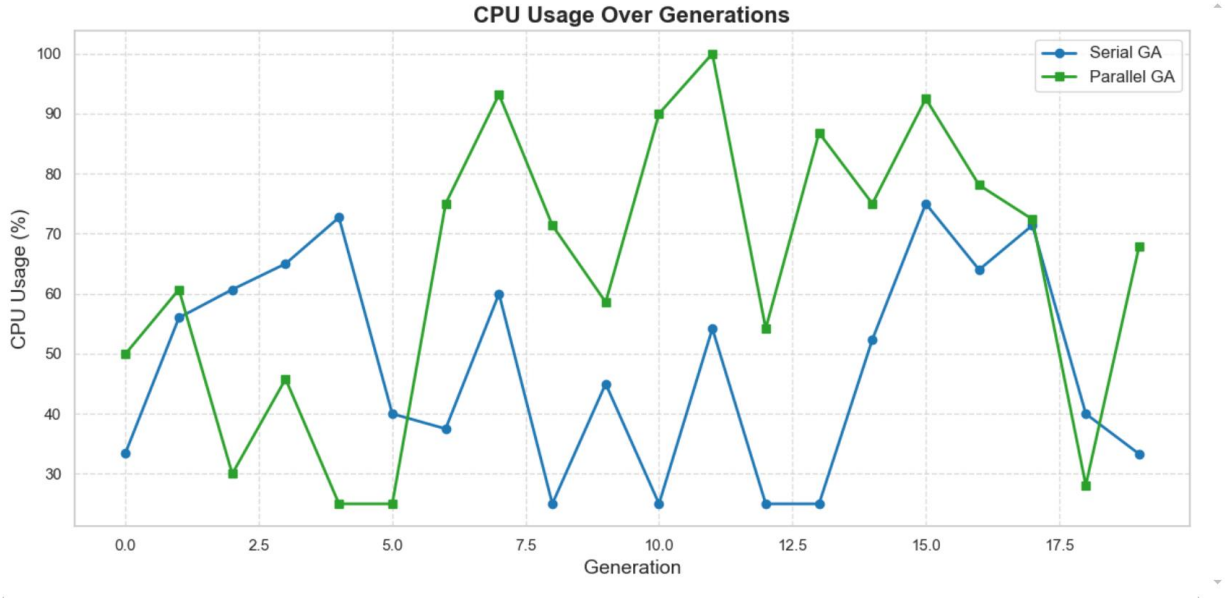

Figure 1: CPU Usage Over Generation

Serial implementation never reaches more than around 75% CPU utilization, but the parallel implementation constantly reaches more than 80% this indicates that parallel implementation utilizes more computational power than serial.

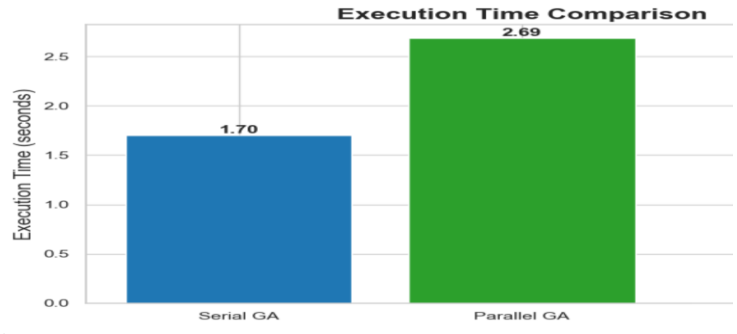

Figure 2: Execution Time Comparison

Among the applied serial and parallel genetic algorithm solution the serial was faster. This might be due to the small scale of the data set.

Serial vs Parallel GA based on total energy allocation per node

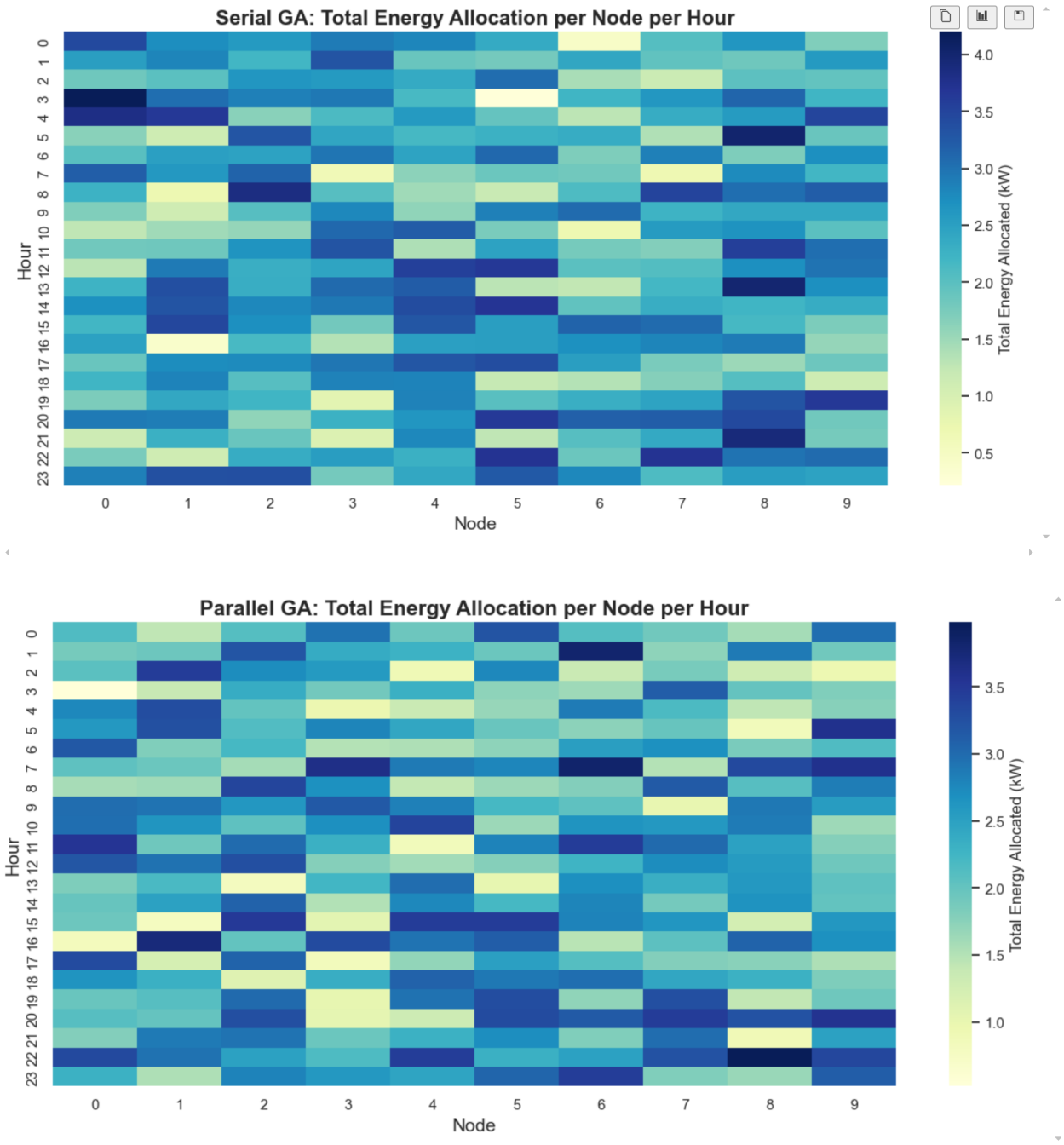

Figure 3: Serial vs Parallel Genetic based on total energy allocation

Total energy allocation per node per hour in the best serial and parallel solution is a bit different, this might be due to the randomness of genetic algorithm.

Serial vs Parallel: Source Utilization per Node

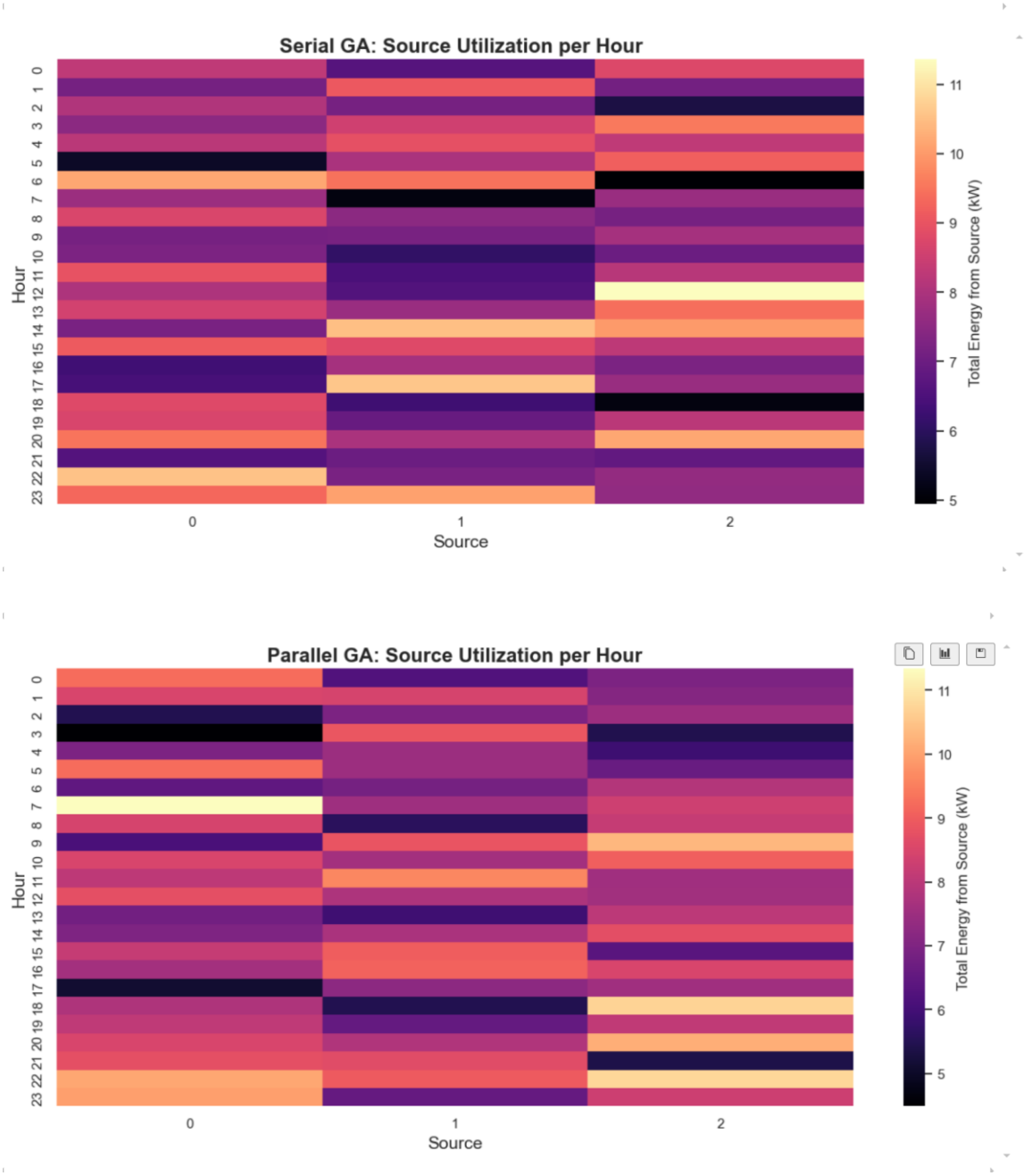

Figure 4: Serial vs Parallel: Source Utilization per Node

Again, we can see that the best source utilization proposed by serial and parallel implementation are a bit different which might be due to the randomness created by mutation and crossover functions.

5.2 Genetic Algorithm vs Greedy Method

To evaluate the performance of a complex algorithm like genetic algorithm we have compared them vs a greedy algorithm.

Figure 5: Execution Time Comparison

Greedy algorithm was much faster than both parallel and serial implantation of genetic algorithms. As greedy algorithms are less computationally intensive than genetic algorithms in this case, the execution time was very quick.

We can also visualize that parallel implementation is slower than serial implementation. This might be due to the moderate size of data as parallel processing shines while working with a large amount of data.

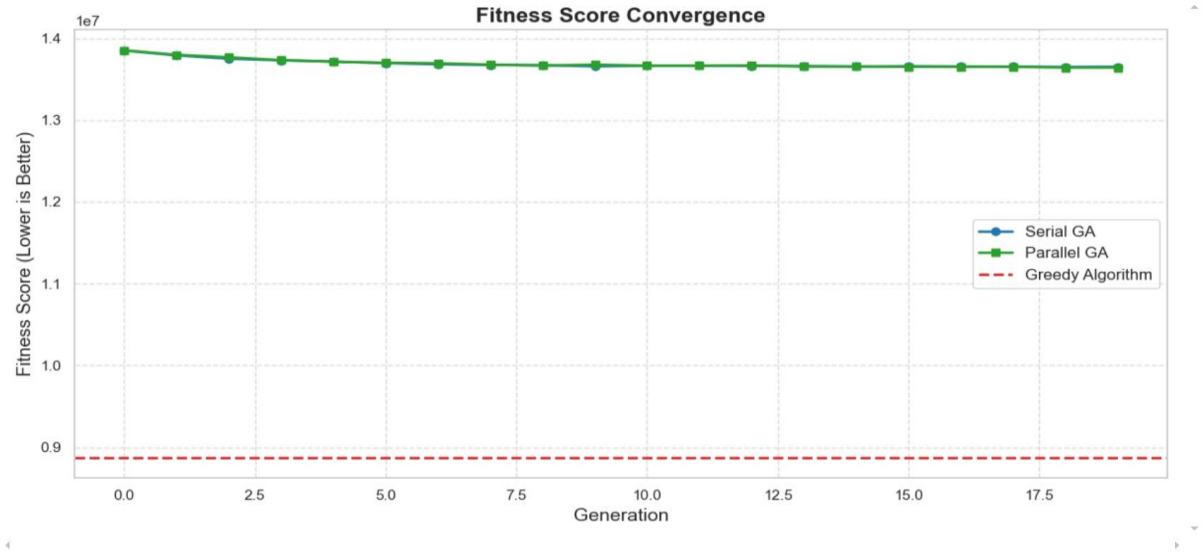

Figure 6: Fitness convergence over generations

We can see that the fitness scores of both serial and parallel implementation are nearly identical throughout 20 generations. With new generations the fitness score improved marginally. Greedy algorithms produce a consistent solution over generations.

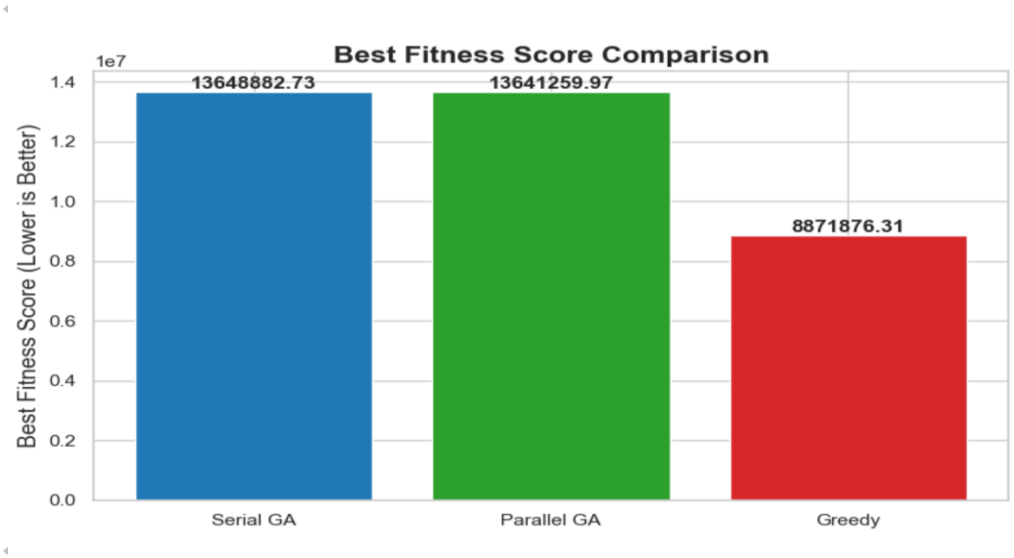

Figure 7: Best Fitness Score Comparison

The parallel and serial implementation of Greedy algorithms produced similar results with nearly identical fitness scores. Among these greedy algorithms performed the best and were able to produce the most optimal solution.

6. Conclusion

This project studies how one can implement a Genetic Algorithm (GA) optimization approach to the effective allocation of energy in a smart-grid setup. A joint optimization task that emerges upon the formulation of the energy-allocation problem as a constrained optimization task is that of demand fulfillment, cost minimization, line losses, and capacity constraints.

Experiments were executed with both serial and parallel implementations of the GA. Parallelization via **ThreadPoolExecutor** elevated CPU utilization and indicated potential for scalability, though, for the current dataset size, serial execution exhibited a narrower computational cost.

A Greedy algorithm was also used as a benchmark against which the performance of the GA could be compared. As expected, the Greedy algorithm performed better algorithmically and yielded favorable results with a deterministic and relatively simple computing nature. However, the GA offered additional freedom to search large, heterogeneous solution spaces and, thus, was more appropriate to large-scale, dynamic, or multi-objective energy-distribution problems.

The outcomes of this project indicates that Genetic Algorithms is a versatile optimization method that is ready to serve on solving complicated problems, even those where a lot of information is at hand and several constraints have to be tended to concomitantly, i.e. those cases when usage of a less advanced approach such as Greedy algorithms rarely promises the discovery of the globally optimal answe

References

Mary, G. A., & R.Rajarajeswari. (2014). SMART GRID COST OPTIMIZATION USING GENETIC ALGORITHM . *IJRET: International Journal of Research in Engineering and Technology*.

Zohuri, B., & McDaniel, P. (2019). The Electricity: An Essential Necessity in Our Life. *Advanced Smaller Modular Reactors*, 1-21.