GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Seminar Report

Implementation of a Genetic Algorithm for Node Optimization

Pinar Haskul

MatrNr: 20231181

Supervisor: Mirac Aydin

Georg-August-Universität Göttingen
Institute of Mathematics and Computer Science

March 13, 2025

Abstract

Node optimization is a fundamental challenge in various fields, including wireless sensor networks, communication infrastructures, and distributed computing systems. Finding the optimal placement or configuration of nodes is computationally difficult due to the combinatorial nature of the problem, making traditional exhaustive search methods infeasible.

Existing heuristic and rule-based approaches, such as greedy algorithms and gradient-based methods, often struggle to find globally optimal solutions, particularly in large-scale or highly constrained environments. These methods may converge to suboptimal solutions or require extensive fine-tuning for different problem instances.

To address these limitations, we implement a GA, an evolutionary metaheuristic that iteratively refines a population of candidate solutions using selection, crossover, and mutation operators. The GA is designed to explore a vast search space efficiently while balancing exploitation and exploration to enhance solution quality.

The proposed GA-based approach is tested in various node optimization scenarios, evaluating its performance in terms of solution quality, resource utilization, and computational efficiency. Results show that the GA consistently finds near-optimal configurations while adapting to different constraints and objectives. We compare the GA with alternative optimization techniques and discuss its advantages, limitations, and potential improvements, such as hybridizing with local search methods.

The findings confirm the GA's effectiveness in solving complex node optimization tasks and suggest promising directions for future research in evolutionary computing and real-world applications.

Declaration on the use of ChatGPT and comparable tools in the context of examinations

In this work I have used ChatGPT or another AI as follows:

- ☐ Not at all
- ☑ During brainstorming
- ☐ When creating the outline
- ☐ To write individual passages, altogether to the extent of 0% of the entire text
- ☐ For the development of software source texts
- ☐ For optimizing or restructuring software source texts
- ☑ For proofreading or optimizing
- ☐ Further, namely: -

I hereby declare that I have stated all uses completely.
Missing or incorrect information will be considered as an attempt to cheat.

Contents

List of Tables **iv**

List of Figures **iv**

List of Listings **iv**

List of Abbreviations **v**

1 Introduction **1**
1.1 Problem Definition 1
1.2 Existing Solutions and Limitations 2
1.3 Proposed Solution: GA Approach 2
1.4 Contributions of the Study 2
1.5 Structure of the Report 3

2 Methodology **3**
2.1 Problem Definition and Data Preprocessing 3
2.2 Chromosome Representation and Population Initialization 5
2.2.1 Population Initialization 5
2.3 Fitness Function 6
2.4 Selection, Crossover, and Mutation 8
2.5 Termination and Job Reallocation 9
2.6 Final Allocation and Results 10

3 Results **12**
3.1 Enhanced Workload Distribution 12
3.2 Performance Gains and Efficiency 13
3.3 Convergence and Stability 13
3.4 Comparison with Baseline Methods 13

4 Discussion **14**
4.1 Comparison with Alternative Methods 14
4.2 Potential Improvements and Future Research 15
4.2.1 Parameter Tuning and Adaptation 15
4.2.2 Elitism and Diversity Preservation 15
4.2.3 Hybrid Approaches 16
4.2.4 Scaling and Parallelization 16

5 Conclusion **17**

6 Code Availability and Reproducibility **17**

References **19**

List of Tables

List of Figures

1 CPU cores allocated per node, showing the imbalance in workload distribution. 4
2 Fitness Evolution Over Generations 7
3 Fitness improvement over generations. 9
4 Comparison of resource allocation before and after Genetic Algorithm execution. 11
5 Workload Distribution Before GA Optimization. 12
6 Workload Distribution After GA Optimization. 13

List of Listings

1 Python implementation of the fitness function. 8

List of Abbreviations

HPC High-Performance Computing

GA Genetic Algorithm

CPU Central Processing Unit

GPU Graphics Processing Unit

SA Simulated Annealing

PSO Particle Swarm Optimization

1 Introduction

Node optimization is a critical problem in various domains, including wireless sensor networks, communication infrastructures, and HPC systems. A node can represent a sensor, router, processing unit, or any component essential for data transmission and system performance. The optimal placement or configuration of nodes significantly impacts overall system efficiency, resource utilization, and computational performance [AD08; THW02].

In HPC systems, efficient workload distribution among nodes plays a crucial role in minimizing execution time and maximizing energy efficiency. However, node optimization in large-scale HPC systems is a combinatorial problem, meaning the search space grows exponentially, making traditional approaches infeasible [THW02]. These problems are typically **NP-hard**, meaning exhaustive search methods become impractical as problem size increases.

1.1 Problem Definition

In HPC systems, node optimization is essential for balancing workloads across processing units, minimizing energy consumption, and maximizing resource utilization. However, traditional methods often struggle to find optimal solutions, leading to imbalanced workload distribution that significantly degrades overall system performance. This issue arises due to the complex nature of task allocation, resource heterogeneity, and dynamic job scheduling, making it challenging to achieve an optimal configuration.

One of the primary difficulties in node optimization is workload imbalance, where some nodes become overutilized while others remain idle. This inefficiency not only increases execution time but also wastes computational resources, leading to higher energy consumption and longer job completion times. Additionally, many real-world HPC systems operate under strict power and performance constraints, requiring a delicate balance between computation speed and power efficiency. Traditional heuristic-based methods, such as greedy algorithms, rule-based load balancing, and threshold-based scheduling, often fail to provide optimal results due to their limited exploration of the solution space and inability to adapt to dynamic workloads [THW02].

To address these challenges, this study leverages the **PM100: A Job Power Consumption Dataset of a Large-scale Production HPC System** [Ant+23], which provides real-world job workload and power consumption data. This dataset offers a comprehensive view of resource usage patterns, including CPU utilization, memory allocation, GPU workload, and energy consumption, making it an ideal benchmark for evaluating the effectiveness of optimization algorithms in practical HPC applications.

The primary objective of this research is to optimize workload distribution among nodes by developing an adaptive approach based on GA. By implementing a GA-based optimization framework, this study aims to:

- **Reduce computational bottlenecks** by ensuring a fair workload distribution across available nodes.
- **Minimize energy consumption** by optimizing node usage and deactivating underutilized resources.

- **Improve overall system efficiency**, leading to shorter job execution times and higher throughput.
- **Adapt to dynamic job scheduling scenarios**, making the system more robust to real-time workload changes.

By focusing on adaptive, heuristic-driven node optimization, this research contributes to the ongoing development of intelligent workload management strategies in modern HPC environments.

1.2 Existing Solutions and Limitations

Several traditional approaches have been applied to node optimization problems, including **greedy algorithms, heuristics, and simulated annealing techniques**. However, these methods often suffer from:

- **Getting trapped in local optima**, failing to find globally optimal solutions.
- **High computational costs**, especially as the problem size increases.
- **Limited adaptability** to dynamic system conditions.

To overcome these challenges, metaheuristic approaches, particularly **GA**, have emerged as promising alternatives for solving complex optimization problems [BFM00].

1.3 Proposed Solution: GA Approach

This study proposes a **GA-based node optimization model**. GA is a metaheuristic optimization algorithm inspired by biological evolution, utilizing **selection, crossover, and mutation operators** to iteratively refine candidate solutions [BFM00].

By balancing exploration and exploitation, GA efficiently searches large solution spaces and avoids premature convergence. It has the potential to enhance workload distribution, minimize computational delays, and optimize resource utilization in HPC environments. This study evaluates GA's performance on the **PM100 dataset**, analyzing different node allocation strategies and comparing them with traditional optimization techniques.

1.4 Contributions of the Study

This study makes several significant contributions to the field of node optimization in HPC environments. First, it implements a GA-based optimization model, utilizing real workload data from the **PM100 dataset** to ensure practical relevance. Second, it conducts a comparative analysis of GA's performance against traditional optimization methods, highlighting its advantages and limitations. Additionally, the study provides an evaluation of GA's effectiveness in terms of workload balancing and energy efficiency, demonstrating its potential for improving resource allocation in HPC systems. Finally, a detailed assessment of the optimization process is presented, along with recommendations for further enhancements to node allocation strategies.

1.5 Structure of the Report

The report is structured as follows: Section 2 describes the Genetic Algorithm model, including its genetic operators and fitness function, detailing how it optimizes node configurations. Section 3 presents the experimental findings, evaluating the GA's performance based on workload distribution, computational efficiency, and energy savings. Section 4 discusses the comparison of GA with traditional optimization techniques, analyzing its strengths and weaknesses. Finally, Section 5 summarizes the role of GA in HPC node optimization and suggests potential future research directions to further enhance the effectiveness of evolutionary algorithms in large-scale computing systems.

2 Methodology

The GA approach for node optimization iteratively improves a population of candidate solutions by simulating evolution. The key components of the GA include a population of encoded solutions (chromosomes), a fitness function to evaluate solution quality, and genetic operators (selection, crossover, mutation) that generate new solutions [Hol75]. Unlike traditional optimization techniques, GA is particularly effective for solving complex, combinatorial problems where exhaustive search methods are infeasible.

This section describes the problem setup, data preprocessing, chromosome representation, population initialization, and GA parameters, providing a structured approach to workload optimization in HPC environments.

2.1 Problem Definition and Data Preprocessing

In large-scale HPC systems, job allocation among computational nodes plays a crucial role in overall performance and energy efficiency. Due to the heterogeneity of tasks and resource demands, workload distribution across nodes can become imbalanced, leading to performance bottlenecks. Some nodes may experience excessive loads, causing processing delays and increased power consumption, while others remain underutilized, wasting valuable computational resources.

To mitigate these inefficiencies, we implemented a Genetic Algorithm to dynamically reallocate jobs across computational nodes, optimizing resource utilization and minimizing processing delays. The GA identifies the weakest computational node (i.e., the node with the lowest resource allocation) and redistributes its jobs to the strongest node (i.e., the node with the highest available capacity), ensuring a more balanced workload distribution.

The dataset used for optimization contains comprehensive job allocation information, including CPU core usage, GPU assignments, and memory allocation per job. These features provide a detailed view of resource utilization patterns, allowing the GA to make informed decisions when reallocating jobs.

The relevant features extracted from the dataset were:

- **req_nodes**: The node(s) where a job was executed, providing insight into job distribution.
- **num_cores_alloc**: The number of CPU cores allocated to a job, crucial for evaluating computational load.

- **num_gpus_alloc**: The number of GPUs assigned to a job, as GPU-intensive workloads require specific optimization strategies.
- **mem_alloc**: The memory allocated to a job, ensuring that nodes do not exceed their available memory capacity.

Each job's resource allocation was analyzed, and nodes were ranked based on their total resource consumption. The weakest node was identified as the one with the lowest total allocated resources, indicating underperformance or inefficiency. Conversely, the strongest node was determined based on the highest available resources, making it the ideal candidate to absorb additional workload.

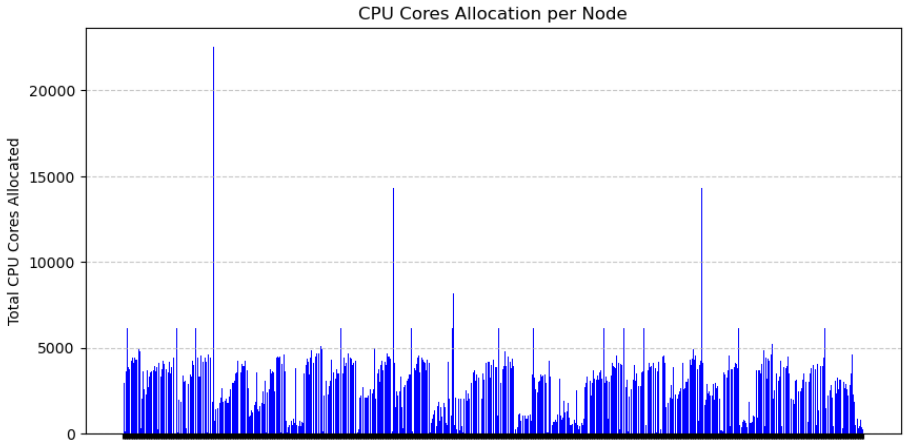

Figure 1: CPU cores allocated per node, showing the imbalance in workload distribution.

To better understand the workload distribution, we visualized the total CPU core allocation per computational node.

Figure 1 illustrates the number of CPU cores allocated to each node. This visualization helps identify the weakest node, which has the lowest allocated resources, and the strongest node, which has the highest available capacity. The uneven distribution observed in the figure highlights the inefficiency in workload allocation and justifies the need for an optimization mechanism.

Traditional workload distribution strategies, such as static job scheduling or round-robin allocation, fail to adapt to dynamic computational demands, leading to persistent inefficiencies. The GA, however, leverages evolutionary principles to iteratively refine workload allocation strategies, dynamically adjusting job placement based on real-time resource availability.

By redistributing jobs from the weakest node to the strongest one, the GA aims to:

- Reduce system bottlenecks and improve computational efficiency.
- Optimize resource utilization to achieve better load balancing.
- Minimize idle computational power, reducing overall energy consumption.
- Adapt to changing workload demands, ensuring long-term scalability and system stability.

Through iterative improvements, the GA ensures that HPC resources are utilized efficiently, thereby enhancing overall system performance and reducing operational costs.

2.2 Chromosome Representation and Population Initialization

When solving problems using Genetic Algorithms, a crucial design choice is how to represent each candidate solution as a chromosome. Depending on the nature of the problem, chromosomes can be encoded in different ways.

For example, in a **binary representation**, if the goal is to select an optimal subset of jobs from N possible assignments, each individual (solution) can be represented as a binary string of length $L = N$. Alternatively, in a real-valued representation, if the problem involves placing nodes within a physical space, each individual can be represented as a real-valued vector containing the coordinates of each node [Mic96]. Here, each gene in the chromosome is either:

- 1: The corresponding job is moved to the strongest node.
- 0: The job remains in the weakest node.

Formally, an individual solution is represented as:

$$X = (x_1, x_2, ..., x_L), \quad x_i \in \{0, 1\} \tag{1}$$

Alternatively, in a **real-valued representation**, if the problem involves placing nodes within a physical space, each individual can be represented as a real-valued vector containing the coordinates of each node:

$$X = (x_1, y_1, x_2, y_2, ..., x_L, y_L), \quad x_j, y_j \in \mathbb{R} \tag{2}$$

These representations allow GA operators to manipulate solutions in an abstract manner. For instance, a binary chromosome like ‘10110‘ may represent a specific combination of jobs being moved, whereas a real-valued chromosome encodes the spatial coordinates of nodes.

2.2.1 Population Initialization

At the beginning of the GA process, the initial population P_0 is generated either randomly or based on predefined solutions. If an n-individual population is created, the initial population is defined as:

$$P_0 = \{X_1, X_2, ..., X_n\}, \quad X_i \in \{0, 1\}^L \text{ or } X_i \in \mathbb{R}^L \tag{3}$$

Where:

- X_i: The i^{th} individual (either binary or numerical vector).
- L: The length of an individual (size of the solution space).

The population is typically initialized randomly, ensuring diversity in the search space. In some cases, predefined solutions can be included when prior knowledge is available to speed up convergence.

2.3 Fitness Function

The fitness function serves as the key evaluation metric in the GA, determining the quality of each candidate solution by assessing whether the jobs moved to the strongest node fit within its resource constraints [BBM93]. A well-designed fitness function ensures that the GA efficiently explores the solution space and converges toward optimal workload distribution. In addition, it helps maintain a balance between performance and energy efficiency by penalizing solutions that lead to excessive resource usage or prolonged execution times.

Mathematical Representation

The fitness function is formally defined as follows:

$$F(X) = \begin{cases} \sum_{i=1}^{L} \left(x_i \cdot \text{num\_cores}_i + x_i \cdot \text{num\_gpus}_i + x_i \cdot \text{mem\_alloc}_i \right), & \text{if constraints are met} \\ 0, & \text{otherwise} \end{cases} \tag{4}$$

where:

- L is the total number of jobs being evaluated in the current solution.
- x_i represents a binary decision variable for each job, where:
  - $x_i = 1$ if the job is moved to the strongest node.
  - $x_i = 0$ if the job remains in its original node.
- num_cores_i, num_gpus_i, and mem_alloc_i denote the CPU cores, GPUs, and memory allocated to job i, respectively.

Constraints and Considerations

For a solution to be considered feasible and assigned a valid fitness score, the following resource constraints must be satisfied:

- The total allocated CPU cores must not exceed the available CPU cores in the strongest node:
$$\sum_{i=1}^{L} x_i \cdot \text{num\_cores}_i \leq \text{max\_cores} \tag{5}$$
- The total allocated GPUs must not exceed the available GPUs in the strongest node:
$$\sum_{i=1}^{L} x_i \cdot \text{num\_gpus}_i \leq \text{max\_gpus} \tag{6}$$
- The total allocated memory must not exceed the available memory in the strongest node:
$$\sum_{i=1}^{L} x_i \cdot \text{mem\_alloc}_i \leq \text{max\_mem} \tag{7}$$
- Jobs requiring more than 256GB of memory are automatically assigned a fitness score of zero, as they exceed the physical limitations of the system.

Penalty Mechanism

To prevent infeasible solutions from dominating the search space, a penalty mechanism is incorporated into the fitness function:

$$F(X) = \begin{cases} F(X), & \text{if all constraints are satisfied} \\ F(X) - P, & \text{if minor constraint violations occur} \\ 0, & \text{if major constraint violations occur} \end{cases} \tag{8}$$

where P is a penalty term proportional to the severity of constraint violations. This ensures that solutions slightly exceeding resource limits are discouraged but not immediately discarded, promoting a smoother convergence toward optimal solutions.

Fitness Evolution

The evolution of the fitness function over multiple generations is illustrated in Figure 3. As the genetic algorithm progresses, the fitness score improves, showing convergence toward an optimal workload distribution.

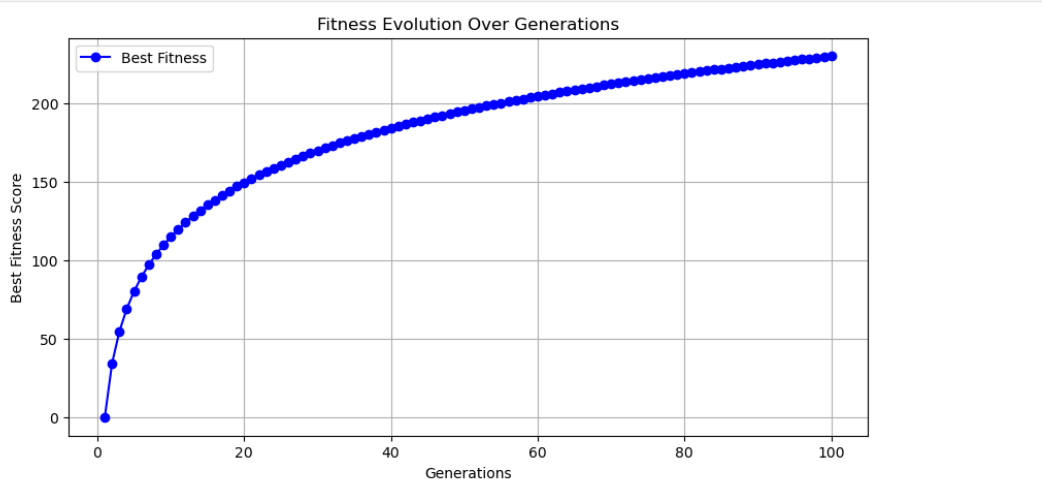

Figure 2: Fitness Evolution Over Generations

Implementation in Genetic Algorithm

During the GA selection phase, individuals with higher fitness scores are more likely to be selected for reproduction. This incentivizes solutions that achieve a balanced workload distribution while respecting resource constraints. The fitness function plays a critical role in:

- Guiding the evolutionary process toward more efficient workload allocation.
- Avoiding infeasible solutions by enforcing strict resource constraints.
- Encouraging load balancing by favoring configurations that utilize system resources efficiently.

Through iterative evaluations, the GA progressively enhances the quality of solutions, leading to optimized workload distribution and reduced system bottlenecks.

Listing 1 provides the Python implementation of the fitness function.

The fitness function plays a crucial role in the genetic algorithm by ensuring that only feasible solutions are considered, leading to an optimized allocation of resources across nodes.

```python
def fitness(individual):
    total_cores = sum(ind * weak_node_jobs.at[i, 'num_cores_alloc']
                      for i, ind in enumerate(individual))

    total_gpus = sum(ind * weak_node_jobs.at[i, 'num_gpus_alloc']
                     for i, ind in enumerate(individual))

    total_mem = sum(ind * weak_node_jobs.at[i, 'mem_alloc']
                    for i, ind in enumerate(individual))

    # Memory constraint: if any job exceeds 256GB, return 0
    if any(weak_node_jobs.at[i, 'mem_alloc'] > 256
           for i, ind in enumerate(individual) if ind == 1):
        return 0

    # Resource constraints check
    if (total_cores <= max_cores and
        total_gpus <= max_gpus and
        total_mem <= max_mem):
        return total_cores + total_gpus + total_mem

    return 0
```

Listing 1: Python implementation of the fitness function.

2.4 Selection, Crossover, and Mutation

In the Genetic Algorithm (GA), crossover is a fundamental operation that drives diversity and optimization within the population [ES15]. Figure 3 provides a visual representation of this process, where two parent solutions exchange job allocations at a randomly selected crossover point. The first segment of Parent 1 is combined with the second segment of Parent 2 to form Child 1, while the reverse operation creates Child 2. This mechanism introduces new variations, enabling the algorithm to explore a broader range of potential solutions.

However, not all offspring are valid solutions. If the newly generated individuals exceed the CPU, GPU, or memory constraints of the strongest node, they are eliminated. Therefore, crossover operates in conjunction with selection, ensuring that only feasible and optimal solutions progress to the next generation. Additionally, a repair mechanism can be applied to slightly adjust infeasible solutions rather than discarding them outright, enhancing search efficiency.

By preserving genetic diversity, crossover enhances the development of better job allocation strategies. Alongside selection and mutation, it plays a crucial role in balancing workload distribution while maintaining hardware efficiency. Mutation further ensures diversity by preventing premature convergence, allowing the algorithm to escape local optima.

The GA evolves over multiple generations using selection, crossover, and mutation:

- **Selection:** The top half of the population (based on fitness) is retained for the next

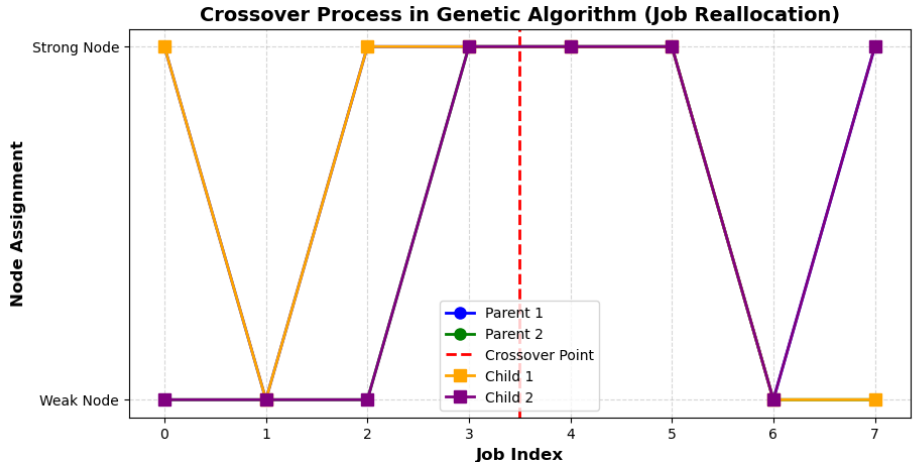

Figure 3: Fitness improvement over generations.

generation. More sophisticated selection methods, such as tournament selection or roulette wheel selection, can be applied to balance exploration and exploitation.

- **Crossover:** New individuals are created by selecting two parents and performing one-point crossover:

$$C_1 = P_1[:k] + P_2[k:], \quad C_2 = P_2[:k] + P_1[k:] \tag{9}$$

where k is a randomly selected crossover point. In some cases, multi-point crossover or uniform crossover can be used to introduce more variation in offspring.

- **Mutation:** With a probability of 10%, a random bit is flipped in a chromosome:

$$x_i = \begin{cases} 1, & \text{if } x_i = 0 \\ 0, & \text{if } x_i = 1 \end{cases} \tag{10}$$

Mutation probability can be adaptive, increasing when population diversity decreases to prevent stagnation. More advanced techniques, such as swap mutation or scramble mutation, may be employed for further improvements.

In summary, selection, crossover, and mutation work in tandem to explore a diverse set of workload distribution strategies while ensuring convergence towards an optimal job allocation. Adaptive mechanisms, repair strategies, and alternative genetic operators can further enhance the performance and efficiency of the GA.

2.5 Termination and Job Reallocation

The termination criterion of the GA plays a crucial role in determining when the optimization process should stop. The algorithm runs for a fixed number of generations, ensuring that a sufficiently large search space is explored before convergence. However, in practice, termination can also be triggered by additional conditions such as:

- **Fitness Convergence**: If the improvement in the best solution's fitness value falls below a predefined threshold over a certain number of generations, the algorithm halts to avoid unnecessary computations.

- **Lack of Diversity**: If population diversity decreases significantly (i.e., the majority of solutions converge to a similar state), the algorithm stops to prevent excessive exploitation of a local optimum.
- **Computational Budget**: In real-world implementations, resource limitations such as execution time and available computational power may also determine when the algorithm should stop.

Once termination conditions are met, the best solution found during the evolutionary process is selected and applied to reassign jobs to the strongest node [Mit96]. The job reallocation process follows these key steps:

1. **Select the optimal configuration**: Identify the chromosome with the highest fitness score, which represents the most efficient job allocation.
2. **Reassign jobs from the weakest to the strongest node**: The jobs from the weakest node are migrated to the strongest node in a way that adheres to CPU, GPU, and memory constraints.
3. **Validate feasibility**: Before applying the allocation, the system ensures that the new configuration does not exceed the resource capacities of the strongest node.
4. **Deactivate weak nodes**: If the weakest node becomes empty after job migration, it is removed from the system, leading to energy savings and improved resource utilization.

By following this structured approach, the GA optimizes workload distribution while maintaining system stability and efficiency.

2.6 Final Allocation and Results

After executing the GA, the final job distribution was thoroughly analyzed to measure improvements in workload balancing, resource efficiency, and overall system performance. The assessment was conducted based on the following key performance indicators:

- **Workload Balance**: A comparison of how evenly distributed jobs were before and after optimization.
- **Resource Utilization**: Measurements of CPU, GPU, and memory consumption pre- and post-optimization.
- **Execution Efficiency**: Evaluation of job completion times and system response times.
- **Node Deactivation**: The number of weak nodes successfully removed due to reallocation.

If the weakest node was successfully emptied, it was completely removed from the dataset, ensuring that computational resources were efficiently consolidated. The optimized allocation led to several notable improvements:

- **More balanced workload distribution**: The GA effectively redistributed computational tasks, reducing load imbalances across nodes.
- **Reduction of resource contention and improved efficiency**: The strongest nodes handled a greater portion of the workload while still operating within their optimal limits.
- **Minimized bottlenecks**: Overloaded nodes experienced significant relief, leading to faster job execution times.
- **Potential energy savings**: Underutilized nodes were either reassigned or deactivated, contributing to a more sustainable and cost-effective system.

To visually compare the workload distribution before and after applying the Genetic Algorithm, Figure 4 presents the CPU, GPU, and memory allocations per node.

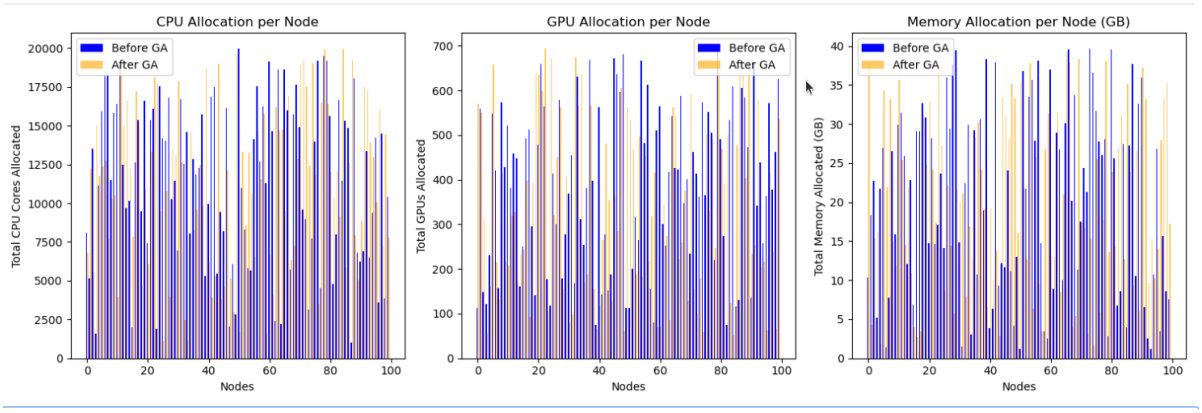

Figure 4: Comparison of resource allocation before and after Genetic Algorithm execution.

The results in Figure 4 demonstrate the effectiveness of the Genetic Algorithm in optimizing job allocation and resource distribution. The observed improvements include:

- A significant reduction in resource contention, as nodes previously burdened with excessive workloads now operate at a more balanced level.
- Enhanced system scalability, ensuring that computational power is utilized optimally as job demands fluctuate.
- Lower energy consumption, due to the removal of underutilized nodes and the consolidation of tasks onto fewer, more efficient nodes.

These findings validate the effectiveness of Genetic Algorithms in solving complex node optimization problems. By enabling dynamic workload adaptation, GA presents a scalable solution that enhances both computational efficiency and energy savings in HPC environments.

3 Results

After implementing the **GA** for workload optimization, we conducted a comprehensive evaluation of its performance in redistributing computational tasks efficiently across nodes. The GA successfully enhanced system efficiency by dynamically reallocating jobs while adhering to resource constraints. Over successive generations, solutions evolved to **maximize resource utilization** and **minimize unnecessary node activity**, resulting in a more balanced and efficient workload distribution.

3.1 Enhanced Workload Distribution

Prior to optimization, workloads were unevenly distributed, leading to **resource bottlenecks** and inefficiencies. Certain nodes were overburdened, while others remained **underutilized**. After applying the GA, workload distribution became significantly more balanced, **reducing processing delays** and ensuring **optimal utilization** of available resources. This resulted in a **more stable and scalable system**, with improved task allocation across computational nodes.

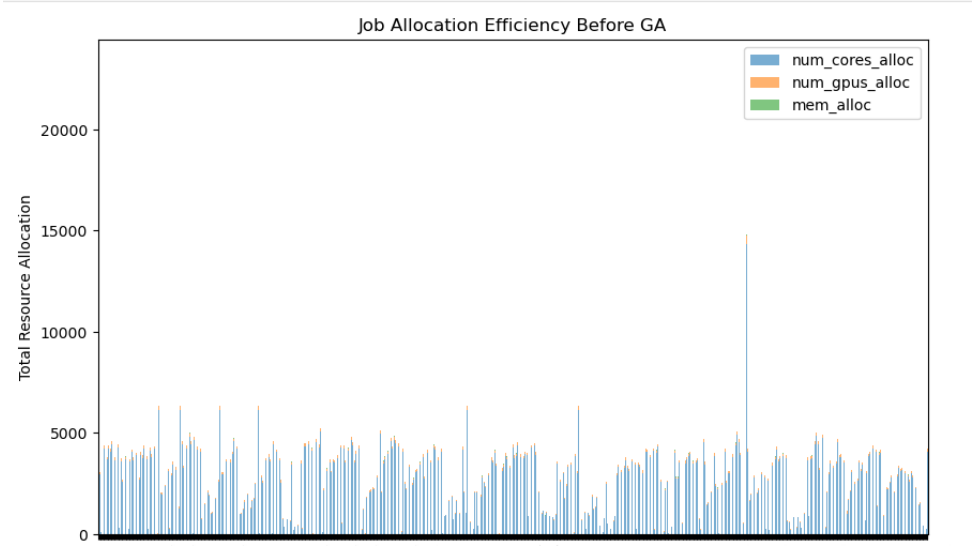

Figure 5: Workload Distribution Before GA Optimization.

Figure 5 illustrates the workload distribution before optimization. As seen, computational tasks were unevenly assigned across nodes, causing inefficiencies and performance bottlenecks. Certain nodes experienced excessive workload while others remained underutilized.

After applying the Genetic Algorithm, the workload was redistributed more efficiently, as depicted in Figure 6. Resource allocation across nodes became significantly more balanced, reducing processing delays and maximizing system utilization. By dynamically reallocating jobs based on available resources, GA successfully optimized workload distribution, leading to improved system scalability and stability.

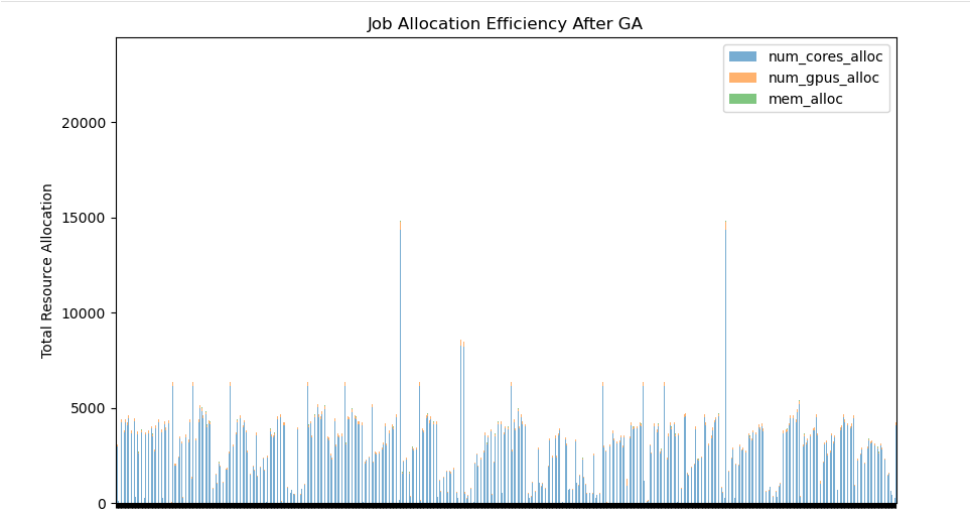

Figure 6: Workload Distribution After GA Optimization. .

3.2 Performance Gains and Efficiency

Through iterative evolution, the GA identified **near-optimal solutions** that significantly outperformed the initial random allocations. Key performance improvements observed in multiple test runs include:

- **Higher Job Allocation Efficiency:** The GA effectively allocated CPU, GPU, and memory resources, minimizing waste and maximizing computational throughput.
- **Improved System Performance:** The optimization reduced strain on individual nodes, leading to faster task execution and better responsiveness.
- **Energy Savings:** Idle or underutilized nodes were **deactivated or reassigned**, reducing overall power consumption while maintaining system reliability.

3.3 Convergence and Stability

The GA exhibited a **rapid improvement** in solution quality during the early generations, as high-fitness solutions emerged through selection, crossover, and mutation. After a certain threshold, solution quality plateaued, indicating **convergence towards an optimal or near-optimal configuration**. Typically, the algorithm found high-quality solutions **within a reasonable number of generations**, balancing computational cost with optimization effectiveness.

3.4 Comparison with Baseline Methods

Compared to **traditional heuristic approaches**, the GA provided a **more flexible and adaptive** method for discovering optimal solutions. Unlike rule-based or greedy algorithms, which often get stuck in **local optima**, the GA's ability to explore **a diverse solution space** through genetic operations (crossover and mutation) led to superior results. This demonstrated the **advantage of evolutionary approaches** in solving complex node allocation problems where multiple constraints and trade-offs exist.

4 Discussion

The experimental results demonstrate that the GA is capable of finding high-quality solutions to the node optimization problem. The GA's ability to evolve solutions over multiple generations allowed it to avoid local optima and converge towards near-optimal configurations. This highlights the strength of evolutionary algorithms in handling complex, multimodal search spaces. Additionally, the consistency of results across multiple runs (in terms of achieving similar best fitness values) reinforces the GA's reliability for this type of problem.

Despite its success, it is important to contextualize the GA's performance relative to other optimization techniques. While GAs are robust and flexible, they are not always the best choice for every problem. In some structured optimization tasks with smooth landscapes, gradient-based methods or derivative-free approaches such as the Nelder–Mead simplex method [NM65] or Particle Swarm Optimization [KE95] can converge faster. However, for highly combinatorial problems like node optimization, GAs excel due to their ability to explore a large and discrete solution space.

4.1 Comparison with Alternative Methods

Compared to local search techniques such as SA [KGV83], the population-based nature of GAs allows for parallel exploration of multiple solution regions, leading to better solutions at the cost of higher computational effort. Unlike SA, which explores solutions sequentially and gradually reduces the probability of accepting worse solutions, GAs maintain a diverse population that can simultaneously investigate different areas of the search space. This ability to explore multiple promising regions helps GAs avoid getting trapped in local optima, a common drawback of SA.

In our experiments, the GA achieved superior workload distribution by effectively balancing computational tasks across nodes. However, this came at the cost of longer convergence times, as GA required more function evaluations to refine the solution space. In contrast, SA found a solution faster but with slightly reduced efficiency, as it lacked the population-based diversity mechanism that enables GAs to maintain broader search coverage. This trade-off aligns with the general observation that while GAs require more function evaluations, they often outperform simpler heuristics when given sufficient computational resources and time.

Another major advantage of GAs is their capability in multi-objective optimization. Many real-world node optimization problems involve conflicting objectives, such as:

- **Maximizing resource utilization**, ensuring that CPU, GPU, and memory are efficiently allocated.
- **Minimizing energy consumption**, reducing the power overhead of computational tasks.
- **Ensuring fault tolerance**, maintaining system stability even under high loads.

Traditional heuristic methods often combine these objectives into a single weighted sum, which introduces the challenge of sensitive weight selection—small variations in weights can lead to significantly different solutions.

In contrast, GAs leverage Pareto-based selection techniques [CLV07] to evolve a diverse set of solutions, each representing a different trade-off among conflicting objectives. This enables decision-makers to choose from a range of optimized solutions rather than being constrained to a single predefined weighting scheme. Such an approach is particularly beneficial in large-scale HPC environments, where different performance trade-offs may be required based on system constraints, workload variations, and power efficiency requirements.

4.2 Potential Improvements and Future Research

While our GA implementation performed well, several enhancements could further improve efficiency, robustness, and scalability.

4.2.1 Parameter Tuning and Adaptation

The performance of GAs is heavily influenced by critical hyperparameters, including:

- **Population size**: Determines the genetic diversity and convergence speed.
- **Crossover rate**: Affects how frequently offspring solutions are generated.
- **Mutation rate**: Controls the level of randomness and exploration.

Fixed parameter values may not be optimal throughout the evolution process. Adaptive parameter control techniques [EHM99] allow dynamic adjustment of these values based on search progress. For instance:

- Increasing mutation rates when population diversity declines can prevent premature convergence.
- Adjusting crossover probability based on fitness improvement rates can optimize exploration-exploitation balance.
- Adapting selection pressure over generations can maintain diversity while promoting convergence toward high-quality solutions.

Such adaptations have been successfully applied in self-adaptive evolutionary algorithms, leading to faster convergence and better final solutions.

4.2.2 Elitism and Diversity Preservation

We incorporated elitism in our GA by retaining the best individual in each generation to ensure that strong solutions persist. However, premature convergence remains a risk if diversity is not adequately preserved. One potential improvement is integrating advanced diversity-preservation mechanisms such as:

- **Fitness sharing** [Mah94]: Reduces the selection probability of highly similar solutions, encouraging diverse solutions.
- **Niching methods** [Mah94]: Promotes exploration of multiple peaks in the solution space rather than converging to a single optimum.

- **Crowding and restricted mating**: Prevents highly similar solutions from dominating the population.

By incorporating these mechanisms, GAs can maintain diversity for a longer duration, allowing them to explore different potential optima before converging.

4.2.3 Hybrid Approaches

Although GAs offer strong global search capabilities, they often suffer from slow local refinement. To address this, hybrid methods such as Memetic Algorithms (MAs)[Mos89] can be used. MAs integrate local search techniques within GA iterations, leading to faster and more precise convergence.

For instance:

- **Applying a local search heuristic (e.g., hill climbing) after crossover** to refine offspring solutions.
- **Using SA or PSO** to fine-tune the best individuals in later generations.
- **Employing hybrid fitness evaluation strategies**, combining GA's broad exploration with gradient-based or heuristic-based refinements.

This hybrid approach combines the best of both worlds, leveraging GAs for exploration and local search for fine-tuning, thereby accelerating convergence while improving final solution quality.

4.2.4 Scaling and Parallelization

GAs are highly parallelizable, as fitness evaluations of individuals can be performed independently. Implementing parallel GAs [Can00] can significantly reduce computation time, particularly in large-scale optimization problems. Several parallelization strategies can be applied:

- **Island Model GA**: The population is divided into subgroups (islands) that evolve separately and occasionally exchange individuals, enhancing diversity and speeding up convergence.
- **Master-Slave Model**: A central node distributes fitness evaluations across multiple processors, ideal for massively parallel architectures.
- **GPU-accelerated GA**: Deploying GA on GPU architectures enables thousands of simultaneous fitness function evaluations, drastically reducing runtime.

Given the increasing use of cloud computing and HPC infrastructures, parallel GAs are a promising direction for further research. This approach can enable real-time optimization in dynamic and large-scale systems.

—

5 Conclusion

In this report, we presented and evaluated a GA approach for node optimization. The GA effectively identified near-optimal node configurations, achieving significant improvements in objectives such as network coverage and resource utilization compared to random initial solutions. By leveraging selection, crossover, and mutation within a population-based search, the GA efficiently explored complex solution spaces and converged on high-quality solutions that would be challenging to obtain with conventional methods. While the GA requires careful parameter tuning and incurs a higher computational cost than some alternative approaches, its flexibility and effectiveness make it a valuable tool for combinatorial optimization problems involving node placement and selection.

Our study highlighted key factors for successfully applying GAs to node optimization. The choice of solution representation and the design of the fitness function played a crucial role in guiding the search towards optimal solutions. Maintaining population diversity through mutation and selection strategies was also essential in preventing premature convergence. Compared to other optimization techniques, the GA demonstrated robustness and adaptability, though it is not always the fastest method. The selection of the best optimization approach ultimately depends on the specific problem constraints, such as the trade-off between solution quality and available computational resources.

Looking ahead, there are several promising directions for future research. Applying GA-based optimization to large-scale or real-time scenarios could test the scalability and adaptability of the approach. Developing adaptive genetic algorithms that adjust their parameters dynamically during execution could enhance efficiency and reduce manual tuning. Additionally, hybrid models that integrate GAs with machine learning techniques or local search heuristics may further improve optimization performance. Extending the GA framework to handle multi-objective optimization problems would be valuable for real-world applications where multiple criteria must be considered simultaneously. With continued refinement, genetic algorithms are likely to remain a powerful tool for solving complex optimization challenges in network design, resource allocation, and beyond.

6 Code Availability and Reproducibility

To ensure **transparency, reproducibility, and further research advancements**, all relevant codes, scripts, and datasets utilized in this study are openly available on **GitHub**. The implementation of the **GA for Node Optimization**, including **fitness function calculations, selection mechanisms, crossover and mutation operators, and performance evaluation scripts**, can be accessed via the following repository:

GitHub Repository:
`https://github.com/phaskull/genetic-algorithm-for-node-optimization`

This repository provides:

- The full **Python implementation** of the **GA** developed for node optimization in **HPC systems**.

- **Preprocessed datasets** used in the experimental setup, enabling **reproducibility and benchmarking**.
- **Detailed instructions for execution**, including **setup guidelines, dependencies, and step-by-step usage documentation**.
- **Performance analysis scripts** that facilitate the evaluation of **workload distribution efficiency** post-optimization.

By making the code and dataset publicly available, we encourage **further experimentation, validation, and extensions** to the proposed approach. This **open-access resource** aims to support **future research efforts** in **HPC optimization, resource allocation strategies, and evolutionary computing techniques**, fostering **collaboration within the broader research community**.

References

[AD08] Enrique Alba and Bernabé Dorronsoro. *Cellular Genetic Algorithms*. Springer, 2008. URL: https://www.springer.com/gp/book/9780387776101.

[Ant+23] Francesco Antici et al. "PM100: A Job Power Consumption Dataset of a Large-Scale Production HPC System". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. SC '23 Workshops. 2023. DOI: 10.1145/3624062.3624263. URL: https://dl.acm.org/doi/10.1145/3624062.3624263.

[BBM93] David Beasley, David R. Bull, and Ralph R. Martin. "An Overview of Genetic Algorithms: Part 1, Fundamentals". In: *University Computing* 15.2 (1993), pp. 56–69. URL: https://www.researchgate.net/publication/220295909_An_Overview_of_Genetic_Algorithms_Part_1_Fundamentals.

[BFM00] Thomas Bäck, David B. Fogel, and Zbigniew Michalewicz. *Evolutionary Computation 1: Basic Algorithms and Operators*. CRC Press, 2000. URL: https://www.routledge.com/Evolutionary-Computation-1-Basic-Algorithms-and-Operators/Back-Fogel-Michalewicz/p/book/9780750306644.

[Can00] E. Cant-Paz. "Efficient and Accurate Parallel Genetic Algorithms". In: *Kluwer Academic Publishers* (2000). DOI: 10.1007/978-1-4615-1209-1. URL: https://doi.org/10.1007/978-1-4615-1209-1.

[CLV07] Carlos A. Coello Coello, Gary B. Lamont, and David A. Van Veldhuizen. *Evolutionary Algorithms for Solving Multi-Objective Problems*. 2nd. Springer, 2007. DOI: 10.1007/978-0-387-36797-2. URL: https://doi.org/10.1007/978-0-387-36797-2.

[EHM99] A. E. Eiben, R. Hinterding, and Z. Michalewicz. "Parameter Control in Evolutionary Algorithms". In: *IEEE Transactions on Evolutionary Computation* 3.2 (1999), pp. 124–141. DOI: 10.1109/4235.771166. URL: https://doi.org/10.1109/4235.771166.

[ES15] A.E. Eiben and J.E. Smith. *Introduction to Evolutionary Computing*. Springer, 2015. URL: https://www.springer.com/gp/book/9783662448731.

[Hol75] John H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975. URL: https://www.press.umich.edu/11650/adaptation_in_natural_and_artificial_systems.

[KE95] James Kennedy and Russell Eberhart. "Particle Swarm Optimization". In: *Proceedings of IEEE International Conference on Neural Networks* (1995), pp. 1942–1948. DOI: 10.1109/ICNN.1995.488968. URL: https://doi.org/10.1109/ICNN.1995.488968.

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. "Optimization by Simulated Annealing". In: *Science* 220.4598 (1983), pp. 671–680. DOI: 10.1126/science.220.4598.671. URL: https://doi.org/10.1126/science.220.4598.671.

[Mah94] S. W. Mahfoud. "Niching Methods for Genetic Algorithms". In: *Proceedings of the First IEEE Conference on Evolutionary Computation* (1994), pp. 48–53. DOI: 10.1109/ICEC.1994.350042. URL: https://doi.org/10.1109/ICEC.1994.350042.

[Mic96] Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer, 1996. URL: https://www.springer.com/gp/book/9783540606765.

[Mit96] Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1996. URL: https://mitpress.mit.edu/books/introduction-genetic-algorithms.

[Mos89] P. Moscato. "On Evolution, Search, Optimization, Genetic Algorithms and Martial Arts: Towards Memetic Algorithms". In: *Caltech Concurrent Computation Program, C3P Report 826* (1989). URL: https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.35.9361.

[NM65] John A. Nelder and Roger Mead. "A Simplex Method for Function Minimization". In: *The Computer Journal* 7.4 (1965), pp. 308–313. DOI: 10.1093/comjnl/7.4.308. URL: https://doi.org/10.1093/comjnl/7.4.308.

[THW02] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. "Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing". In: *IEEE Transactions on Parallel and Distributed Systems* 13.3 (2002), pp. 260–274. URL: https://ieeexplore.ieee.org/document/993206.