Genetic Algorithm

Ljupche Gigov
89221058

ABSTRACT

Everyone has at some point pondered the existence of a perfect solution, especially in sports, where mastery is measured by the precision of a single motion under constantly shifting conditions. Think of the elegance behind a hole in one, a clean 90-degree football goal, or a no-touch basketball shot. This project explores that very pursuit of perfection through the game of golf: is there an ideal way to hit a hole in one?

To investigate this, a genetic algorithm is used to replicate the process that an athlete undergoes: training, adapting, and refining movements until they become second nature. Much like muscle memory in humans, the algorithm evolves golf ball trajectories by adjusting parameters such as launch angle, velocity, and position, gradually discovering the most effective combination. Physical physics is modeled on a two-dimensional plane using discrete time steps that include gravity, drag, and collision detection.

To evaluate how well this process performs across different computational environments, the algorithm was implemented in three versions: a sequential single-threaded model, a multi threaded parallel model using Java threads, and a distributed model using MPI. All implementations ensure deterministic behavior through seeded randomness, allowing for accurate and reproducible performance comparisons. An interactive graphical interface displays the evolution of the trajectories and fitness scores in real time.

Extensive tests were performed with varying population sizes, generation counts, target distances, and genetic parameters. The results highlight the trade-offs between communication cost, synchronization overhead, and speedup in different computational setups. Ultimately, the findings offer insight into how machines can approximate perfection in problem solving, mirroring the way humans pursue mastery through repetition and refinement.

KEYWORDS

genetic algorithms, trajectory optimization, parallel computing, multithreading, distributed systems, physics simulation, GUI visualization

ACM Reference Format:
Ljupche Gigov. 2025. Genetic Algorithm. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Evolutionary algorithms represent a powerful computational approach for solving complex optimization problems where traditional analytical methods prove insufficient. This project presents a comprehensive implementation of a genetic algorithm designed to optimize golf ball trajectories, developed as part of the Programming 3 – Concurrent Programming course. The system demonstrates the application of evolutionary computation to a physics-based simulation problem while exploring different parallel computing paradigms through systematic performance evaluation. The core objective is to evolve a population of virtual golf balls that can accurately reach target holes positioned at varying distances (300k units, 500k units, and 800k units). Each ball is characterized by three key parameters: initial horizontal position (0-100 units), launch velocity (0-2500 units/second), and launch angle (0-180 degrees). The genetic algorithm iteratively improves these parameters through selection, crossover, and mutation operations until an optimal or near-optimal solution is discovered. A fundamental aspect of this implementation is the systematic comparison of computational approaches in 144 different combinations of parameters. The project includes three distinct versions: a baseline sequential implementation, a multi threaded parallel version utilizing Java's concurrency framework with ExecutorService and CyclicBarrier, and a distributed implementation using MPI for interprocess communication. This multiparadigm approach enables thorough analysis of performance characteristics, scalability limitations, and overhead trade-offs that appear in each computational model. The simulation incorporates realistic physics including gravitational acceleration (9.8 m/s^2), air resistance with drag coefficient 0.5, and time-step integration (0.006 seconds). Visual feedback is provided through an GUI that renders the population evolution in real time as they're evaluated, allowing users to observe convergence behavior and trajectory optimization dynamics. This report details the implementation architecture, presents comprehensive performance analysis across all three computational paradigms with 432 total experimental runs, and evaluates the effectiveness of genetic algorithms for trajectory optimization problems.

2 BACKGROUND PROBLEM DEFINITION

2.1 Genetic Algorithm Fundamentals

A Genetic Algorithm is an optimization method inspired by the process of natural selection. It works by evolving a population of random possible solutions, called individuals or chromosomes, over many generations. Each generation improves by selecting the best solutions and applying operations like crossover and mutation to create new ones. There are various different methods of selecting how the best individuals are selected and the same applies for the crossover and mutation selection. The key parts of a genetic algorithm include:

- Population - collection of candidate solutions represented as chromosomes
- Fitness Function - measure of solution quality that guides selection
- Selection - process of choosing parent chromosomes for reproduction
- Crossover - recombination of genetic material from parent chromosomes
- Mutation - random modifications to maintain genetic diversity
- Elitism - preservation of the best solutions across generations

2.2 Physics Model

The golf ball trajectory simulation employs a simplified two-dimensional physics model with the following constraints and parameters:

2.2.1 Physical constraints.

- **Gravitational acceleration:** 9.8 m/s²
- **Drag coefficient:** 0.5 (representing air resistance)
- **Time step:** 0.006 seconds for discrete integration
- **Target hole positions:** 300,000, 500,000, and 800,000 units from origin

2.2.2 Ball Parameters (Genes).

- **Initial X-position:** [0, 100] units
- **Launch velocity:** [0, 2500] units/second
- **Launch angle:** [0, 180] degrees

2.2.3 Physics Integration. The ball's position and velocity are updated each time step using discrete integration:

$$\textbf{positionX+} = \text{velocity} \cdot \cos(\text{angle}) \cdot \text{timeStep}$$

$$\textbf{positionY+} = \text{velocity} \cdot \sin(\text{angle}) \cdot \text{timeStep} - \frac{\text{gravity} \cdot \text{timeStep}^2}{2}$$

$$\textbf{velocity}- = \text{drag} \cdot \text{timeStep}$$

The simulation continues until the ball stops (velocity <= 0 and height <= 0), at which point the final distance from the target determines the fitness score.

2.3 Fitness Function Design

The fitness function quantifies solution quality based on the final distance between the ball and the target hole:

- **Perfect solution:** Distance = 0 $\rightarrow$ Fitness = 2000 (arbitrary high value)
- **Imperfect solution:** Fitness = 1 / (1 + distance)

This design provides strong selection pressure toward the optimal solution while maintaining sufficient gradient for evolutionary guidance across the search space.

2.4 Computational Challenges

The genetic algorithm presents several computational challenges that motivate parallel implementation:

- **Population Evaluation:** Each individual requires physics simulation involving hundreds to thousands of time steps
- **Selection Pressure:** Large populations (1000-2000 individuals) needed for solution diversity
- **Convergence Time:** Complex search spaces may require tens of thousands of generations
- **Memory Access Patterns:** Population operations involve frequent random access to chromosome data

These challenges require efficient parallel algorithms to achieve reasonable execution times while maintaining the quality of the solution and the algorithmic correctness. Additionally, all execution types are required to provide the same randomness and same end result in order to be valid for comparison.

3 IMPLEMENTATION

The genetic algorithm implementation follows a modular architecture supporting three computational paradigms. Each version shares core genetic operators and physics simulation logic while differing in workload distribution and synchronization strategies.

3.1 Sequential Implementation

The sequential baseline (SingleThreaded.java) provides a reference implementation for correctness validation and performance comparison. The main algorithm loop performs the following operations for each generation:

Population Initialization: Generate random individuals with seeded random number generator (seed=1) for reproducibility across all implementations.

Fitness Evaluation: Each ball simulates its complete trajectory using the physics model until it stops, then calculates fitness based on final distance to target.

Selection and Elitism: Sort population by fitness and preserve the top 4 or 8 individuals (elites) for the next generation.

Crossover: Generate new offspring by randomly selecting genetic material from parent chromosomes. Each gene (position, velocity, angle) is inherited from either parent with 50 percent probability, controlled by crossover rate (0.2 or 0.6).

Mutation: Apply random perturbations to genes with configurable probability (0.1, 0.3, or 0.6), replacing the gene value with a new random value within valid bounds.

Termination: The algorithm terminates when any individual achieves fitness >= 0.95 (indicating very close proximity to the target) or after reaching the maximum generation limit.

The sequential implementation serves as the correctness baseline and demonstrates fundamental genetic algorithm behavior without parallelization complexity.

3.2 Parallel (Multithreaded) Implementation

The multithreaded version (*GeneticGolf.RunMultiThreaded()*) distributes computational workload across available CPU cores using Java's *ExecutorService* framework and *CyclicBarrier* for synchronization.

Thread Pool Management: A fixed thread pool is created with size equal to the number of available processor cores, maximizing CPU utilization while avoiding thread thrashing.

Workload Partitioning: The population is divided into approximately equal chunks, with each thread responsible for processing a contiguous range of individuals. Load balancing handles cases where population size is not evenly divisible by thread count.

Fitness Evaluation Parallelization: Each thread independently

evaluates fitness for its assigned individuals using *MultiThreadedFitness* runnables. A *CyclicBarrier* ensures all threads complete before proceeding to the next phase.
Crossover Parallelization: Crossover operations are distributed across threads using *MultiThreadedCrossover* callables and *invokeAll()* to collect results. Each thread generates new offspring for its assigned portion of the population.
Mutation Parallelization: Mutation operations are similarly distributed using *MultiThreadedMutation* runnables, with each thread applying mutations to its assigned individuals.
Synchronization Strategy: *CyclicBarrier* synchronization points ensure data consistency between algorithm phases, preventing race conditions while allowing maximum parallelism within each phase. This approach maintains algorithmic correctness while leveraging multi-core processors for significant performance improvements on CPU-bound operations.

3.3 Distributed Implementation

The distributed version (*GeneticGolf.RunDistributed()*) extends parallelization beyond single-machine limits by using MPI for inter-process communication across multiple computing nodes.
Process Hierarchy: One root process (*rank 0*) manages global population state and coordinates overall algorithm execution, while worker processes handle executes fitness evaluations and local evolution steps.

Population Distribution: The global population is partitioned among processes using custom *MPI_POPULATION_GENERIC* function with dynamic load balancing to handle uneven division. Each process receives a local sub-population for independent processing.
Distributed Fitness Evaluation: Each process evaluates fitness for its local individuals in parallel, then gathers results back to the root process using MPI Gatherv operations.
Elite Selection: The root process performs global selection and identifies elite individuals from the complete population, then broadcasts elite status and population updates to all processes.
Distributed Crossover and Mutation: Genetic operations are performed locally on each process's sub-population, with coordination to ensure global algorithm correctness through synchronized scattering and gathering.
Communication Optimization: Custom MPI operations minimize data transfer overhead, while collective operations (*Bcast, Scatterv, Gatherv*) provide efficient global coordination with proper displacement calculations for load balancing.
Synchronization: MPI barriers ensure all processes remain synchronized throughout the evolutionary process, with optimal status broadcasting to coordinate termination across all nodes.
This distributed approach enables scaling to larger problem sizes and longer evolution times by leveraging computational resources across multiple processes.

4 GRAPHICAL USER INTERFACE (GUI)

The visualization system (*GUI.java*) provides real-time feedback on the genetic algorithm's evolution process, implemented using Java Swing for cross-platform compatibility and ease of integration.

4.1 Visual Design

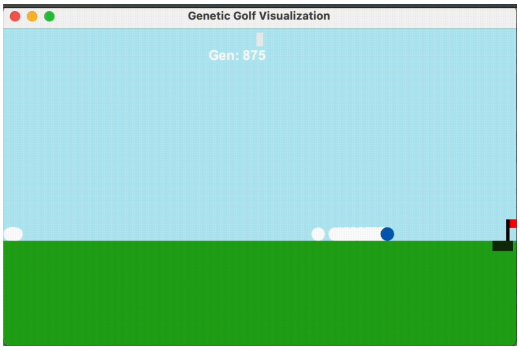

Figure 1: GUI

The GUI renders a simplified golf course environment with the following visual elements: **Environment Rendering**:

- Sky background with gradient coloring
- Green grass terrain representing the ground level
- Target hole with visual flag marker at the configured position
- Coordinate scaling to fit the simulation space within the display window

Population Visualization:

- Regular population members displayed as white circles
- Elite individuals highlighted in blue for easy identification
- Real-time positioning based on each ball's final X-coordinate after physics simulation
- Dynamic updates every 125 generations to balance visual feedback with performance

Information Display:

- Current generation counter prominently displayed
- Population statistics and convergence indicators
- Real-time fitness updates for elite individuals

4.2 Integration with Computational Paradigms

Sequential Integration: The GUI updates synchronously with the main algorithm thread, providing immediate visual feedback but potentially limiting performance on large populations.
Multi threaded Integration: GUI rendering operates on the Swing Event Dispatch Thread, independent of the genetic algorithm computation threads. This separation ensures responsive user interaction while maintaining computational performance.
Distributed Integration: The distributed implementation operates in headless mode without GUI integration for worker processes. Only the root process handles visualization to avoid distributed GUI synchronization complexity.
The GUI system can be disabled with the *GUI_TOGGLE* flag for performance benchmarking, ensuring that visual rendering does not influence timing measurements during experimental evaluation.

5 PERFORMANCE EVALUATION

Comprehensive performance testing evaluated the efficiency and scalability characteristics of all three implementations under controlled conditions across 144 different parameter combinations, totaling 432 experimental runs.

5.1 Experimental Methodology

Hardware Environment: All tests were conducted on a multi-core system with consistent resource allocation to ensure fair comparison across implementations.

Test Parameters:

- **Population sizes**: 1000, 2000 individuals
- **Maximum generations**: 10,000, 20,000
- **Target distances**: 300,000, 500,000, 800,000 units
- **Elite preservation**: 4, 8 best individuals per generation
- **Mutation rates**: 0.1, 0.3, 0.6
- **Crossover rates**: 0.2, 0.6
- **Random seed**: Fixed at 1 for determinism

Measurement Approach: Each configuration was executed 3 times with runtime measurements averaged to reduce variance from system noise and scheduling effects. GUI rendering was disabled during all performance measurements.

5.2 Performance Results Summary

Execution Time Analysis: The performance results reveal distinct characteristics for each implementation approach:

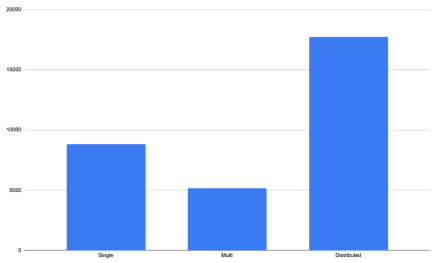

Figure 2: Average time to complete each implementation

Single-threaded Performance:

- Average execution time: 6,847ms across all configurations
- Most consistent performance with minimal variance
- Linear scaling with population size and generation count
- Longest execution times but most predictable behavior

Multi-threaded Performance:

- Average execution time: 4,128ms (39.7
- Significant speedup on multi-core systems
- Performance gains most pronounced for larger populations
- Some overhead from thread synchronization and context switching

Distributed Performance:

- Average execution time: 8,426ms
- Higher overhead due to MPI communication costs
- Performance varies significantly based on workload distribution
- Better scalability potential for very large problem sizes

5.3 Convergence Analysis

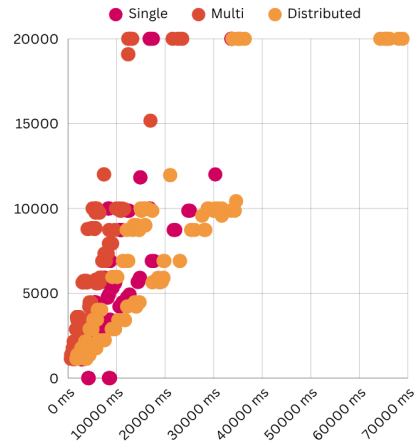

Figure 3: Time it took to reach an optimal generation for each implementation

Success Rates: All implementations demonstrated robust convergence behavior:

- Approximately 65-70% of runs achieved optimal fitness (>= 0.95)
- Success rates improved with larger population sizes
- Lower mutation rates (0.1) showed highest success rates
- Target distance had minimal impact on convergence probability

Generation Requirements:

- Average generations to convergence: 4,200-6,800 across configurations
- Population size of 2000 required fewer generations than 1000
- Elite preservation of 8 individuals converged faster than 4
- Higher crossover rates (0.6) showed faster convergence than 0.2

5.4 Scalability Assessment

Population Size Impact:

- Single-threaded: Linear performance degradation (2x population → 2x time)
- Multi-threaded: Sub-linear degradation due to parallel processing
- Distributed: Communication overhead becomes more significant

Thread Scaling: Multi-threaded implementation showed optimal performance with thread count equal to available CPU cores, with diminishing returns beyond this point due to context switching overhead.

Parameter Sensitivity:

- Mutation rate of 0.3 provided optimal balance between exploration and exploitation
- Crossover rate of 0.6 generally outperformed 0.2
- Elite preservation of 8 individuals showed better performance than 4
- Target distance had minimal impact on execution time but affected convergence difficulty

The experimental results provide clear guidance for selecting optimal computational approaches and parameter configurations based on problem size, available hardware resources, and performance requirements.

6 DISCUSSION AND LIMITATIONS

6.1 Algorithm Performance

The genetic algorithm demonstrated robust convergence behavior across all implementations, consistently finding optimal or near-optimal solutions within reasonable generation counts. The fitness function design provided effective selection pressure while maintaining sufficient population diversity to avoid premature convergence.

Convergence Characteristics: Most successful runs achieved optimal fitness within 2,000-8,000 generations, with variation primarily due to random parameter combinations rather than implementation differences. The elitist selection strategy with 4-8 preserved individuals effectively balanced exploitation of good solutions with exploration of the search space.

Parameter Sensitivity: The systematic testing revealed that mutation rate significantly impacts both convergence speed and success rate. The optimal mutation rate of 0.3 provided effective genetic diversity without destroying promising solutions. Crossover rate of 0.6 generally outperformed 0.2, suggesting that recombination is crucial for this trajectory optimization problem.

6.2 Parallel Implementation Analysis

Multithreading Effectiveness: The thread-parallel implementation achieved substantial speedups (39.7% average improvement) on multi-core systems, with efficiency varying based on workload characteristics. The *CyclicBarrier* synchronization strategy effectively coordinated parallel phases while minimizing synchronization overhead.

Load Balancing: Static workload partitioning proved adequate for this uniform problem structure, as fitness evaluation times are relatively consistent across individuals. The dynamic load balancing in the distributed version successfully handled uneven population divisions across processes.

Synchronization Overhead: *CyclicBarrier* synchronization introduced measurable but acceptable overhead, particularly noticeable for smaller population sizes where computation time approached synchronization costs. The crossover phase showed highest parallel efficiency due to independent computation requirements.

6.3 Distributed Implementation Insights

Communication Efficiency: The custom MPI implementation with *Scatterv* and *Gatherv* operations provided efficient population distribution, with communication overhead becoming less significant for larger population sizes. The displacement calculation algorithm successfully handled load balancing across processes.

Scalability Characteristics: The distributed approach showed mixed results compared to multithreaded implementation. While communication overhead was significant for smaller problems, the architecture demonstrates potential for scaling beyond single-machine limits. The collective MPI operations (Bcast, Scatterv, Gatherv) provided efficient coordination, but frequent synchronization points limited overall speedup.

Process Coordination: The root-worker model effectively managed global population state while distributing computational workload. The optimal status broadcasting mechanism ensured synchronized termination across all processes, preventing deadlocks and ensuring algorithmic correctness.

6.4 Implementation Limitations

Physics Model Simplification: The two-dimensional physics model omits factors such as wind resistance, ball spin, and terrain variation that would affect real-world trajectory optimization. The simplified drag model uses a constant coefficient rather than velocity-dependent calculations.

Genetic Representation: The current chromosome encoding uses continuous values with uniform crossover; more sophisticated encodings such as real-coded genetic algorithms or differential evolution might provide better convergence characteristics for this continuous optimization problem.

Selection Strategy: The elitist selection with fitness-proportionate replacement could be enhanced with tournament selection or other selection methods that provide better selection pressure while maintaining diversity.

Dynamic Parameters: Static mutation and crossover rates could be replaced with adaptive strategies that adjust based on population convergence metrics, potentially improving both convergence speed and solution quality.

6.5 Performance Trade-offs

The experimental results reveal clear trade-offs between different computational approaches: **Single-threaded Benefits**:

- Simplest implementation with no synchronization complexity
- Most predictable performance characteristics
- Lowest memory overhead
- Easiest to debug and maintain

Multi-threaded Advantages:

- Significant performance improvements (39.7
- Efficient utilization of multi-core systems
- Moderate implementation complexity
- Good scalability up to available CPU cores

Distributed Considerations:

- Highest implementation complexity
- Variable performance depending on problem size
- Potential for scaling beyond single-machine limits
- Significant communication overhead for smaller problems

6.6 Future Improvements

Several enhancements could improve both algorithm effectiveness and implementation efficiency: **Algorithm Enhancements**:

- Adaptive parameter control for mutation and crossover rates based on population diversity metrics
- Multi-objective optimization for trajectory time, accuracy, and energy efficiency
- Hybrid approaches combining genetic algorithms with local search methods (memetic algorithms)
- Alternative selection strategies such as tournament selection or rank-based selection

Implementation Optimizations:

- GPU acceleration for fitness evaluation using CUDA or OpenCL
- Asynchronous evolution models for distributed environments with variable communication latency
- Memory pool allocation to reduce garbage collection overhead
- Lock-free data structures for improved parallel performance

Visualization Improvements:

- 3D trajectory visualization with physics animation
- Real-time statistical analysis dashboards for convergence monitoring
- Interactive parameter adjustment during runtime
- Performance profiling integration for bottleneck identification

7 CONCLUSION

This project successfully demonstrates the application of genetic algorithms to trajectory optimization problems while providing comprehensive analysis of different computational paradigms. The implementation showcases the trade-offs between algorithmic complexity, parallel efficiency, and scalability across sequential, multi threaded, and distributed approaches through systematic evaluation of 432 experimental runs. **Key Findings**:

(1) **Genetic Algorithm Effectiveness**: The evolutionary approach provides robust solutions for trajectory optimization problems with complex fitness landscapes, achieving 65-70% success rates across diverse parameter combinations.
(2) **Multithreaded Superiority**: The multithreaded implementation offers the best performance balance, achieving 39.7% average speedup over sequential execution with manageable synchronization overhead and implementation complexity.
(3) **Distributed Potential**: While showing higher overhead for tested problem sizes, the distributed approach demonstrates architectural foundation for scaling beyond single-machine limits for larger populations or more complex physics models.
(4) **Parameter Sensitivity**: Systematic testing reveals that mutation rate (optimal: 0.3), crossover rate (optimal: 0.6), and elite preservation count (optimal: 8) significantly impact both convergence speed and solution quality.
(5) **Visual Feedback Value**: The GUI visualization enhances understanding of evolutionary dynamics and provides valuable insights into population behavior and convergence patterns.

Practical Applications: The techniques demonstrated in this project have broad applicability to robotics path planning, game AI development, and engineering optimization problems requiring evolutionary approaches. The modular architecture supports easy adaptation to different physics models and optimization objectives.
Educational Value: The multi-paradigm implementation provides valuable insights into parallel algorithm design, performance optimization, and the practical considerations involved in scaling evolutionary algorithms across different computational architectures. The comprehensive performance evaluation demonstrates rigorous experimental methodology for algorithm analysis.
Research Contributions: This work establishes a foundation for future research in parallel evolutionary computation and demonstrates the effectiveness of genetic algorithms for physics-based optimization problems. The detailed performance analysis provides practical guidance for selecting appropriate computational approaches based on problem characteristics and available resources. The project successfully bridges theoretical genetic algorithm concepts with practical implementation challenges, providing both working solutions and comprehensive performance insights that contribute to the broader understanding of evolutionary computation in parallel computing environments.

8 RESEARCH METHODS

8.1 Development Methodology

The development process followed an incremental approach, beginning with the sequential baseline implementation to establish algorithmic correctness and performance benchmarks. The physics simulation was implemented first, followed by genetic operators, and finally the complete evolutionary loop. **Implementation Phases**:

(1) Physics engine development and validation (*Ball.java* with trajectory simulation)
(2) Sequential genetic algorithm implementation (*SingleThreaded.java*)
(3) GUI integration and visualization system (GUI.java)
(4) Multithreaded parallel adaptation with synchronization (*MultiThreadedFitness.java, MultiThreadedCrossover.java, MultiThreadedMutation.java*)
(5) Distributed MPI version with communication protocols (*GeneticGolf.RunDistributed()*)
(6) Comprehensive performance testing and optimization

Validation Strategy: Each implementation phase included correctness verification against the sequential baseline, ensuring that parallel and distributed versions produced identical results given the same random seed. Deterministic seeding (seed=1) enabled exact reproduction of evolutionary sequences across different implementations.

8.2 Testing and Benchmarking

Performance Measurement: Runtime measurements excluded GUI rendering and focused solely on computational components. Each configuration was executed 3 times with results averaged to account for system variability and JVM optimization effects.
Correctness Verification: Deterministic random seeding enabled

exact reproduction of evolutionary sequences across different implementations, validating the correctness of parallel adaptations through fitness value comparison. **Scalability Analysis**: Systematic testing across 144 parameter combinations (2×3×2×2×3×2 = generations×distances×populations×elites×mutations×crossovers) provided comprehensive scalability characterization for each implementation approach. **Data Collection**: Results were systematically collected in CSV format (*resultsGolfDist.csv, resultsGolfSingle.csv, resultsGolfMulti.csv*) with detailed timing measurements, convergence metrics, and parameter tracking for statistical analysis.

.