Using Genetic Algorithms to Teach an Agent to Walk

Omar Abdel Haq, Elie Eshoa, May Soshi, and Ricky Williams

Harvard University

May 4, 2023

Abstract

This paper discusses the use of genetic algorithms to teach an agent to walk efficiently. Walking is a complex task that requires coordination of multiple joints, making it difficult to find an optimal solution using traditional optimization methods. The research presented in this paper aims to solve this problem by utilizing genetic algorithms, a type of optimization algorithm inspired by natural selection, and parallelism. The parallel implementation of the algorithm shows significant improvement in the training time compared to the sequential baseline, providing a practical solution to the problem of teaching an agent to walk. The findings of this paper has significant implications for the fields of physics, computer science, and biology.

1 Background and Significance

Genetic algorithms (GA) are a type of optimization algorithm inspired by the natural selection and genetic processes observed in biological evolution. The concept of GA was first introduced by John Holland in the 1970s as a way to solve complex optimization problems by simulating the natural process of evolution. GA involves the use of a population of potential solutions to a problem that evolves over generations through processes such as selection, reproduction, and mutation. Each solution in the population is evaluated based on a fitness function, which measures how well it solves the problem at hand. The solutions with the highest fitness are selected to reproduce and create a new generation of solutions. The process continues until a satisfactory solution is found, or a predefined stopping criterion is met. GA has been widely applied to a variety of fields, including engineering, computer science, economics, and biology, and has shown to be an effective method for finding optimal solutions to complex problems.

In this research paper, we will explore the use of genetic algorithms to teach an agent to walk. Walking is a complex task that involves the coordination of multiple joints, and finding an optimal solution using traditional optimization methods can be challenging. We will use a GA to evolve a set of control parameters that govern the agent's movement and learn to walk by maximizing a fitness function that rewards forward motion and stability. To speed up the learning process, we will exploit parallelization by using a shared and distributed computing system to evaluate multiple solutions in parallel. This will allow us to explore a larger space of possible solutions in less time and increase the chances of finding an optimal solution. The results of this study could have implications for the development of autonomous robots and artificial intelligence systems that require complex locomotion capabilities.

Current research about walking agents focuses heavily on reinforcement learning (RL), a machine learning technique that involves an agent learning to make decisions based on trial and error, receiving rewards for actions that lead to positive outcomes and punishments for actions that lead to negative outcomes. Researchers have used RL to train walking agents in both virtual and physical environments,

enabling agents to learn a variety of gaits and locomotion strategies. In addition to RL, researchers have also explored imitation learning and genetic algorithms to optimize the performance of walking agents. H. Picado et. al. (2009) created a method based on partial Fourier series for joint trajectory planning and then used genetic algorithms to generate the parameters of the walk. Similarly, Y. Kambayashi, M. Takimoto and Y. Kodama (2005) utilized genetic algorithms to determine the optimal gait for a biped walking robot. The genetic algorithm proved more efficient in determining the parameters of the robot's gait than other traditional methods. Overall, the current research about walking agents is a developing and a unique topic of exploration for machine learning scientist, as they seek to develop agents that can navigate complex real-world environments.

2 Scientific Goals and Objectives

The scientific goal of this project is to investigate the effectiveness of using genetic algorithms to teach an agent to walk. This involves exploring the ability of the genetic algorithm to evolve a set of control parameters that govern the agent's movement and maximize its forward motion and stability. We aim to achieve significant speedup in the optimization process by employing parallelization through OpenMP and MPI. The objective of the project is to achieve the scientific goals through the following steps:

1. Design and implement a simulation environment that enables the evaluation of an agent's walking behavior
2. Use genetic algorithm to evolve a set of control parameters to identify the optimal solution
3. Employ a shared/distributed computing system to parallelize the genetic algorithm and optimization process
4. Analyze the results of the simulation to evaluate the effectiveness of the genetic algorithm in teaching the agent to walk as well as the impact of parallelization on the learning process

Since genetic algorithms are designed to emulate evolution, our genetic algorithm implementation requires epochs upon epochs of iterations to mimic different generations of agents. Even with a parallelized implementation of the algorithm, running it for eons would require dedicated machinery with large numbers of nodes and cores. Utilizing high performance computing structures will provide us with the necessary resources for parallelizing such a large optimization problem. HPC resources will significantly reduce the time required to simulate the genetic algorithm and allow us to arrive at a solution much faster. Our program, if ran successfully on a cluster, has the potential to be applicable to an enumerate number of problems, including but not limited to the development of autonomous robots and artificial intelligence systems that require complex locomotion capabilities. Having an optimized program and solution to an walking-agent problem will allow us to alter and update the program for other variations of physics-related problems.

3 Algorithms and Code Parallelization

Algorithm: OpenMP

For this computational study, we adopted a genetic algorithm to teach a walker how to walk in a 2D environment. The walker has 2 legs, a head, and 4 joints in total. The 2 upper joints fix the legs to the head, and the 2 lower joints connect the "upper" and "lower" halves of each leg, serving as knees. The walker is represented by a chromosome, which is a vector of 4 floating point numbers, each representing

the motor speed each joint attempts to reach. The main function of our program starts by initializing a population of 100 walkers, each with a random motor speed for each joint. We implemented the walker struct to include a `states` feature that stores the state of the walker at each iteration. This `states` feature is an array of structures that stores the position and velocity of each joint and the head at each iteration. Thus, for the nth iteration, the walker will store all previous $n-1$ positions and velocities in its `states` feature. This feature allows us to analyze and visualize the motion of the walker and identify any issues in its walking pattern.

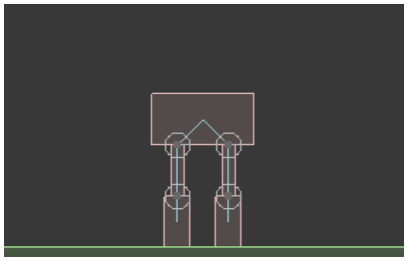

Figure 1: Walker Visualization

During each iteration, we set the motor speed values of the joints and get the new position of the head. We calculate the fitness of each chromosome based on the new position of the head. The fitness of a chromosome is defined as the x or horizontal displacement of the head from the starting position. After evaluating the fitness of each of the walkers, we sort the chromosomes by fitness and select the top 10% of chromosomes (i.e. the fittest 10%). We then create a new population of 100 chromosomes by randomly selecting chromosomes from the top 10% and applying crossover and mutation to them. We repeat the above steps for 10 iterations and print the chromosome with the highest fitness. This is the baseline implementation of our genetic algorithm. We are the main developer of all code utilized in this project.

We parallelized our algorithm by using OpenMP. Specifically, we parallelized the `create_new_population` and the `run_genetic_algorithm` functions. The `create_new_population` takes in a vector of the fittest walkers from the previous generation and their chromosomes and creates a new population of walkers by randomly selecting two walkers from the fittest walkers, performing crossover and mutation on their chromosomes, and using the resulting chromosome to create a new walker. The function returns a vector of pointers to the new walkers. The `run_genetic_algorithm`, on the other hand, takes in the number of walkers, the number of iterations to run the genetic algorithm, and a float `fit_ratio` that represents the ratio of fittest walkers to select from the previous generation for generating offspring in the new generation. It initializes the population of walkers, simulates them, and then runs the genetic algorithm for the specified number of iterations. In each iteration, it selects the fittest walkers, creates a new population of walkers using `create_new_population`, simulates them, and repeats the process. Using OpenMP, we are able to parallelize both of these functions by dividing the code into smaller units of work and assigning each unit to a different thread which are executed simultaneously. Parallelizing these two functions allows us to redution computation time by running multiple parts of the algorithm simultaneously.

Algorithm: MPI

For the MPI extension of our genetic algorithm, we develop an adjusted algorithm that does the following:

1. Initialize a Cartesian topology simulating different "islands" of walker populations of size $p \times p$. Each node/rank in our Cartesian topology is akin to an "island" or single walker population (this is what we were experimenting with in the earlier parts).

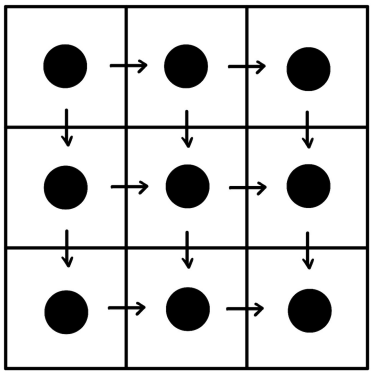

Figure 2: Cartesian Representation of The Islands

2. For every rank, we found its associated neighbours in the Cartesian topology, i.e. top, bottom, left, and right neighbours.

3. We send a predetermined amount of walkers from one island to its neighboring island. In our simulation, we send exactly 5% of our walkers to the neighbouring island to the right, and another 5% to the neighbouring island down.

4. We implemented non-blocking communication using MPI: while islands engage in walker-message sharing from send vectors `Depart1_, Depart2_` into receive buffers `Next1_, Next2_`, we incorporate "foreign" walkers from vectors `Arrive1` and `Arrive2` and run `STAY_GENERATIONS` iterations of the genetic algorithm. Figure 2 above shows our rank communication strategy.

5. Repeat steps (3)-(4) until the end of the experiment.

Unfortunately, the MPI implementation did not return all processes, but the complete implementation is in the branch mpi.

Validation, Verification

Given the random nature of the crossover involved in creating each generation, the verification of our algorithm will not be based on reference data from other experiments and results that might be available online.

For our case, we validate the correctness of our method by checking the following:

1. We validate the crossover function on its own by testing the resulting chromosomes and checking for valid outputs.

2. As for generation creation (i.e. creating a new population), we assert that the number of current walkers is equal to the intended size of the population that is given as an argument to the executable, as seen in the code.

3. Finally, we assert that the fittest walkers are selected at each generation before crossover, by testing the sorting function separately.

These steps were present in earlier commits in our main branch, and they ensure with high probability the correctness of the algorithm. Ultimately, since our implementation is a simplified version of the cited

sources (Picado et al., 2009), the comparison of a simple walker (what we have) is not appropriate with the referenced data. Furthermore, the cited sources implement a very different genetic algorithm compared to ours so it not appropriate to establish a comparison between the two. Thus, our validation will stick to the above mentioned steps in addition to the visualization provided later.

4 Performance Benchmarks and Scaling Analysis

We would like to preface this section by stating that a complete roofline analysis with operational intensity calculation is not appropriate for our algorithm because of the use of excessive third-party libraries. To account for that, we will provide a rich suite of results, and include both a strong scaling and weak scaling analysis. The aim is to evaluate the genetic algorithm's efficiency and effectiveness when solving complex problems. We will analyze the algorithm's performance using both weak scaling and strong scaling paradigms.

Motivation behind the scaling paradigms:

We will test the assumptions of Amdahl's Law:

Assumption 1: Fixed problem size. To evaluate the appropriateness of this assumption, we ran two sets of experiments: one with 100 walkers and the other with 2000 walkers. We ran the 100 walker experiment 15 times and the 2000 walker experiment 3 times (with each of the experiments having 1000 iterations). From these two sets of experiments, we selected the walker with the best performance. After running the experiments, we noticed that we achieved the same performance for both sets of experiment, however, achieving it faster for the experiment with larger problem size. This suggests that a bigger problem size gets us to the solution faster, while a smaller problem size takes many more simulations, but eventually reaching a similar solution. Hence, while the assumption of a fixed problem size is mostly appropriate, we cannot ignore the fact that the weak scaling paradigm is perfectly applicable.

Assumption 2: Negligible communication overhead. This assumption is mostly correct, as all threads or processors run on shared memory. If we introduce MPI, the performance measures will be mostly based on our weak scaling results. However, our proposed MPI is non-blocking, and thus this assumption is still—mostly—appropriate for our genetic algorithm.

Assumption 3: All-or-None parallelization. This assumption is not unreasonable to assume, as we can parallelize most of the code completely in most cases.

Strong Scaling

To evaluate the performance of the genetic algorithm, we utilized Amdahl's Law for strong scaling analysis. As mentioned in class, determining the serial fraction (f) is generally very difficult. To determine (f) for our genetic algorithm, we timed the non-parallelizable and parallelizable portions of the sequential code. We estimated f to be around 0.035-0.045 based on our experiments with 2000 walkers for about 100 generations. This was based on some assumptions discussed in our previous milestones. The algorithm is basically three main steps conceptually in each generation: creation, simulation, and selecting the fittest (and repeat). In short, f was approximated as the difference between the total runtime and the three parallelizable components in the code, divided by the total walltime of the algorithm.

Next, we measured the speedup of the algorithm as a function of the number of threads, while keeping the problem size fixed at 2000. The results of the strong scaling analysis are shown below, including the table of our job specifications (Test case A is our Strong Scaling Job). Note that the theoretical speedup is given by $\frac{1}{f+\frac{(1-f)}{p}}$ and our speedup is calculated by the formula T_s/T_p by running our experiments.

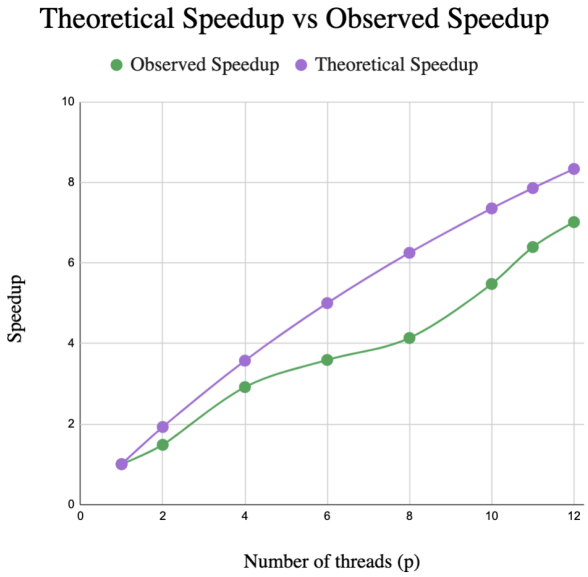

Figure 3: Strong Scaling Analysis on 2000 Walkers and 150 Generations, where $f = 0.04$

From the graph, we observe that the speedup increases with the number of threads, but it saturates at some point. This saturation is due to the limited parallelizable fraction of the algorithm. The maximum speedup achieved is limited by the serial fraction of the algorithm. In our case, the maximum speedup achieved was around 12 threads.

Weak Scaling:

The purpose of weak scaling analysis is to determine the efficiency of the parallelization scheme as we scale up the problem size. Firstly, we present some results that prove, experimentally, that using $n = p$ is an appropriate assumption when deciding on how many processors (or threads) p are needed as we scale our problem size n. As we see in the graphs below, the time needed on average (i.e. a timestep in our algorithm) increases linearly with the size of the problem (number of walkers, n).

We ran 150 iterations (generations) per simulation. We ran simulations with upto $p = 12$ threads and measured the weak efficiency E_p. Since the relationship is linear, we used $n = 200, 400, \ldots, 2400$ and $p = 1, 2, \ldots, 12$ respectively in our weak scaling analysis.

The result below shows the weak efficiency E_p:

The weak efficiency does not stay great because of the probable overhead introduced when scaling over multiple threads. This could be explained because the third-party libraries may be interfering with that, in addition to higher load imbalance throughout the runs.

Lastly, we performed local runs of our algorithm, setting the number of walkers to 2000 and the number of iterations to 150 as a standard job. This local run took approximately 13370ms using 12 threads (we stop at 12 because it is the number of threads for which E_p is around 50%, as requested). This is from the

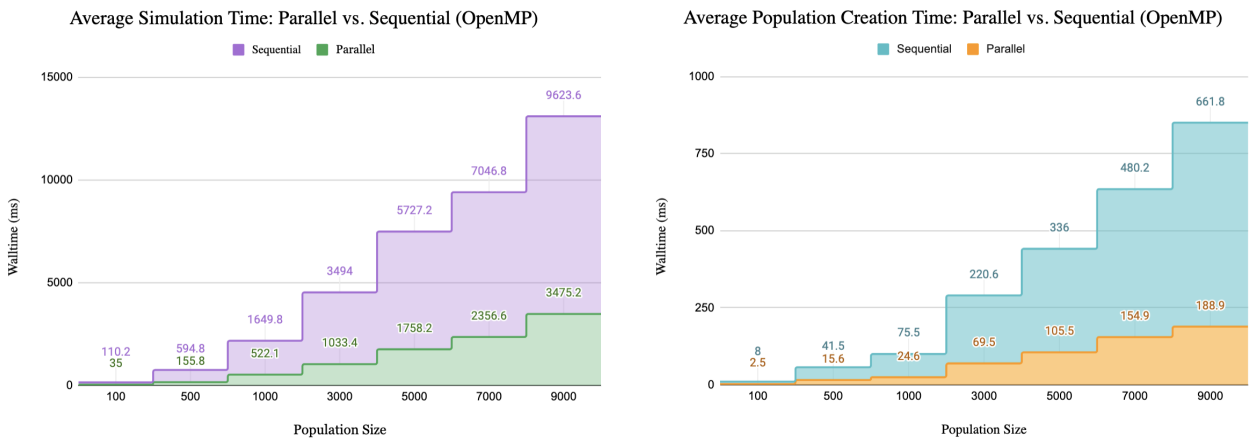

Figure 4: Linear Increase in Average Simulation and Population Creation Times

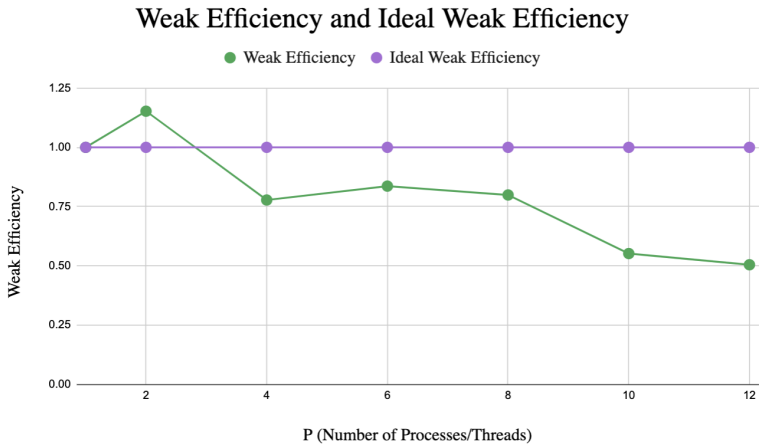

Figure 5: Weak Scaling Analysis over 150 Generations

OpenMP algorithm only, which was taken as a base case for our purposes. Lastly, because our algorithm does not use any libraries for I/O, neither does it use external files to run, we omit this information from the task's information Table 1 .

	OpenMP Test Case
Typical wall clock time (ms)	13370
Typical job size (nodes)	1
Memory per node (GB)	16
Maximum number of input files in a job	None (0)
Maximum number of output files in a job	None (0)
Library used for I/O	None

Table 1: Workflow parameters of the OpenMP test case, which we used during project development.

5 Resource Justification

More Populations >> More Generations

In Section 4, we reported our observation of plateauing behavior in the course of our experiments. This trend can be attributed to the fact that we generate populations at random and persist with them until they can no longer generate improved generations of walkers, limited by genetic mutations. Our investigations have indicated that this plateauing phenomenon is present across all population sizes that we examined. We provide graphical representations of this trend for populations sizes of 100 and 2000, as shown below. Note that in both cases, the performance of the best walker reaches a plateau.

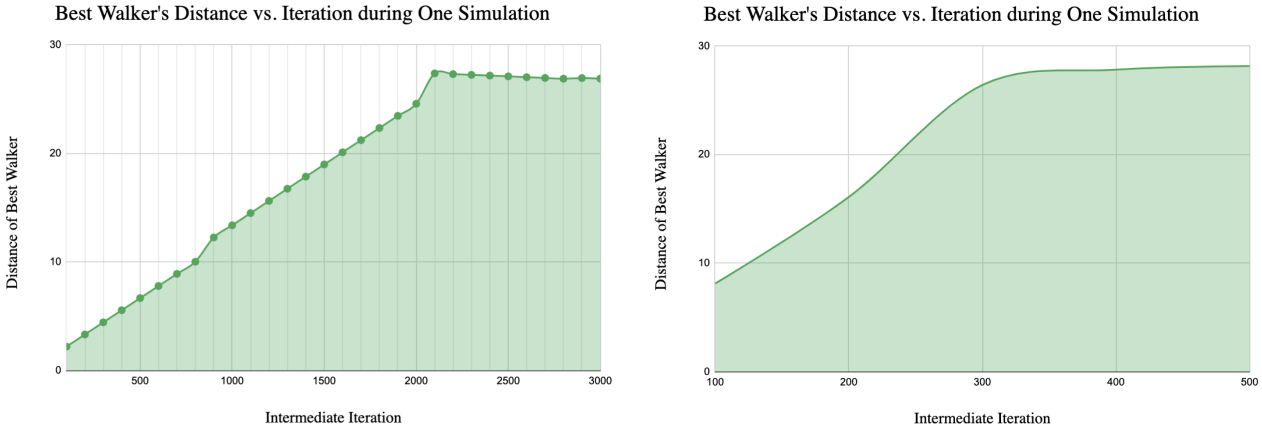

Figure 6: **100** Walkers vs **2000** Walkers

We have observed that continuously running the algorithm for more generations may not yield a better solution. Instead, initializing new populations with different parameters has proven to be more effective. Thus, to accommodate the increasing number of populations required for this approach, we would like to request additional nodes. The additional nodes will enable us to process the expanding number of walkers and ultimately improve our results.

To efficiently identify optimal solutions, it is imperative to broaden our search capabilities within the solution space. As previously noted, this can be accomplished by scaling up the size of the population. Therefore, we propose to enhance our capacity by increasing the population size, which will facilitate a more comprehensive exploration of the solution space and yield improved outcomes.

Utilizing our MPI implementation, we can execute a grid of $d \times d$ islands, or simulations, each comprising 5000 walkers (agents). Specifically, we can leverage 16 nodes to conduct 4×4 simulations of our genetic algorithm, running the algorithm on each island for a duration of 3000 generations. While the number of generations is flexible, our experimentation, as depicted in Figure 6, suggests that 3000 generations are adequate for algorithmic convergence. (Note that this argument is solely intuition-based)

As we have measured in our trials, per each node that holds a population size of 5000 walkers, an iteration (generation creation steps) takes about

$$0.00889 \text{ node hours} = 16 \text{ nodes} \times \frac{2s}{3600 \frac{s}{\text{hour}}}$$

Hence, the task can be seen in Table 2.

	MPI test case
Simulations per task	16
Iterations per simulation	5000
node hours per iteration	0.00889
Total node hours	711.11

Table 2: Justification of the resource request

This will request a total of 711.11 node hours.

References

[1] Picado, H., Gestal, M., Lau, N., Reis, L. P., & Tomé, A. M. (2009). Automatic generation of biped walk behavior using genetic algorithms. Lecture Notes in Computer Science, 805–812. https://doi.org/10.1007/978-3-642-02478-8_101

[2] Y. Kambayashi, M. Takimoto and Y. Kodama, "Controlling biped walking robots using genetic algorithms in mobile agent environment," ¡em¿IEEE 3rd International Conference on Computational Cybernetics, 2005. ICCC 2005.¡/em¿, Mauritius, 2005, pp. 29-34, doi: 10.1109/ICCCYB.2005.1511542.