Parallelizing the Genetic Algorithm on the Cloud

jamesn1197,pyrosnowman24,dvd9604
Department of Electrical and
Computer Engineering
University of Texas at San Antonio

***Abstract*—The Genetic Algorithm (GA) is a stochastic search algorithm that attempts to maximize an objective function. Depending on its construction, this problem can take a long time to converge, assuming it does. Because of this, any avenue by which algorithmic runtime can be reduced is necessary when employing this algorithm. One such avenue is through the use of the distributed compute power that the cloud offers. How exactly the distributed computing expediates this algorithm varies depending on the architecture. We employ this distributed compute power by calculating the fitness of all candidate solutions on separate cloud nodes, and we show that this approach scales monotonically with increased node counts.**

I. Introduction

As systems and their controllers become more complex, there arises a need for architectures that support hosting these algorithms for use in real-time. One available architecture is the cloud. The cloud offers distributed computation through the use of a multitude of networked machines that can work together to achieve a common goal. One way this is done is by taking programs and algorithms that were previously constructed for one computer to run from beginning to end, and revising them such that the programs or algorithms can be split up and worked on by multiple computers simultaneously in a process known as parallelization. After parallelization, users can send their newly revised algorithms to the cloud where it can be worked on by a cluster of virtual machines simultaneously, thereby dropping overall algorithm runtime in low-latency cases.

The algorithm my group worked on was the Genetic Algorithm (GA). The GA is a stochastic search algorithm based on naïve mimicry of Darwin's Theory of Natural Selection and biological hereditary mechanisms. The search is random, but the selection is not. Nevertheless, the random search can lead to long settling times, and, as such, any place where this search can be expedited is welcome. Because the random search branches this algorithm entails are structured to be independent of each other, GA offers itself very clearly to parallelization. The GA also lends itself to hierarchical structuring, to where if the population-level calculations become to cumbersome, then the populations can be split into sub-populations as necessary. For the intents and purposes of the search we implement, we do not explore the effectiveness of this hierarchical structuring for further parallelization, since our population size is not large enough to warrant it.

Within the group, I was more focused on implementing the GA so as to maximize the likelihood that our solution was globally optimal, and that it lends itself to effective demonstration of the power of the parallel computing offered by the cloud.

II. Implementation

All code was written in Python using NumPy, Matplotlib, MPI4Py, and SQLite3, and all final code was executed on an OpenStack cluster of varying node counts. Message passing interface (MPI) scatters and gathers the chromosome objects among the nodes in the cluster. The particular use case for GA employed is to automate the process of tuning a proportional-integral-derivative (PID) feedback controller driving a linear, time-invariant plant. The model for the plant is given by Equation 1. The PID controller operates on the error signal and has a continuous time model as given by Equation 2, where K_p, T_i, and T_d are the parameters being tuned. This system was optimized according to the integral time absolute error criterion as given by Equation 3. How the linear, time-invariant feedback control system was simulated in discrete time is discussed in the next section, then how the GA was implemented is shown in the following section.

$$G(s) = \frac{2500}{s^2 + 80s + 2500} \tag{1}$$

$$e(t) = K_p(e(t) + \frac{1}{T_i}\int_{t_0}^{t} e(\tau)d\tau + T_d\dot{e}(t)) \tag{2}$$

$$J(k) = \int_0^{t_f} \tau|e(k,\tau)|d\tau \tag{3}$$

A. Discrete-Time Simulation

For our implementation, we used a discrete-time PID controller that uses the difference equation model given by [1] in order to generate the control signal $u(m)$ at time instance m. For this controller, the sampling rate f_s is 100 Hz. For this simulation to work, the model needs to be converted into a discrete time model. First, the plant model is converted it to state space controllable canonical form to obtain its respective state matrix A, input matrix B, and output matrix C. Then, the discretization method given by Equation 4 is applied to these two matrices to obtain their discrete-time approximates, $\tilde{A}$ and $\tilde{B}$.

$$\tilde{A} = e^{A/f_s} \qquad \tilde{B} = \int_0^{1/f_s} e^{A\tau} \cdot B d\tau \tag{4}$$

$$\begin{gathered} x(m+1) = \tilde{A}x(m) + \tilde{B}u(m) \\ y(m) = Cx(m) \\ e(m) = e(m-1) + 0.01m|1 - y(m)| \end{gathered} \tag{5}$$

Now that the discrete time matrices $\tilde{A}$ and $\tilde{B}$ are known, the simulation can be run as a for loop iterating through an initially empty time series of state values x and u. This is done by saving all of these parameters as NumPy arrays and iteratively solving for the values of the states $x(m+1)$, output $y(m)$, and error $e(m)$ of the next time step as shown by Equation 5. This was simulated until a final time t_f of 10 seconds or 1000 time intervals, and the last value of $e(m)$ was returned as the final value of the cost function $J(k)$. If the controller causes the system to become unstable, the cost would exceed some threshold J_{max} at which point it would break from the loop and set the cost equal to this threshold.

Because of the iterative nature of this simulation, it unfortunately cannot be vectorized beyond how it is already. Accordingly, this is the most computationally taxing part of the implementation and as such this is the part that is distributed to the worker nodes so that they can tackle this problem in parallel. Instead of splitting this for loop like what was done in previous cases, this algorithm is parallelized by sending different groups of entire chromosome objects to different nodes.

B. Genetic Algorithm

Our implementation of the GA sets the chromosomes to be the candidate PID parameters expressed as a NumPy array. A block diagram of the implementation of this on the OpenStack cluster is shown in Figure 1. The algorithm starts with a random initialization of 30 uniform random vectors given by 6, all independently and identically distributed. The boundaries for this distribution were imposed to bias the initial search space so that many of the candidate chromosomes would be more likely to be stable PID parameters.

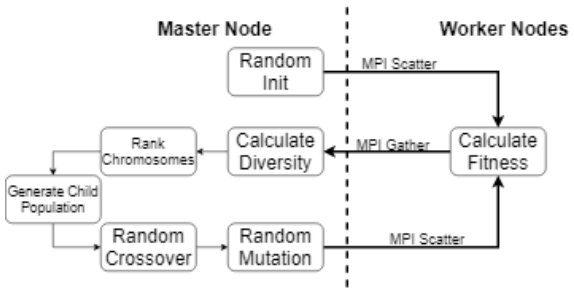

Fig. 1. Block diagram of GA implementation

$$\begin{bmatrix} K_p \\ T_i \\ T_d \end{bmatrix} \sim Uniform(0.001, 10) \tag{6}$$

1) Calculating Fitness: Next, each of these candidate solutions were simulated through discrete-time state space. Again, from the error signal, $e(t)$, the integral time absolute error cost function $J(k)$ is calculated by Equation 3. Once this has been calculated for all of the chromosomes, MPI gathers all of the cost functional values to the master node, where the mean PID parameters ($\bar{K}_p, \bar{T}_i$ and $\bar{T}_d$) are calculated. After this has been calculated, the diversity score $(d(k))$ of chromosome k is found by squaring its euclidean distance from the mean of the population, as given by Equation 7. Because GA is a non-convex optimization tool, adding this diversity term is important to add value to solutions that are different from the others, thereby promoting candidate solutions that explore different parts of the search space.

$$d(k) = \left\| \langle K_p(k), T_i(k), T_d(k) \rangle - \langle \bar{K}_p, \bar{T}_d, \bar{T}_i \rangle \right\|^2 \tag{7}$$

$$F(k,n) = d(k)e^{-0.05n} + \ln(J_{max}) - \ln(J(k)) \tag{8}$$

Seeing as I now have both my cost $J(k)$ and diversity score $d(k)$, I can now calculate the cardinal fitness score $F(k,n)$ of chromosome k in generation n shown in Equation 8. Since the cost is something that I want to minimize and the algorithm is meant to maximize a function, I map the cost to a monotonically decreasing function. I chose the function $-\ln(\cdot)$ not only because it is monotonically decreasing, but it also dilates the region in the neighborhood of zero so as the solutions approach zero their effect on the cardinal fitness score of chromosome k, $F(k,n)$, increases dramatically relative to the diversity term. Because $\ln(J_{max})$ is constant across generations, it does affect the optimization, though it is still included so that $F(k,n)$ is strictly positive, thereby making the fitness function slightly easier to debug, should the opportunity have presented itself. Also I forced the diversity score to be transient with respect to the generation and time constant 20 generations so that after a large number of trials, how spread out the solutions are is less important than the optimality of the solutions themselves.

Once the cardinal fitness is calculated, the chromosomes are then put into a list and sorted from highest to lowest, and then their index in that list becomes their ordinal fitness. Now the population is set to generate an population of equal size according to a finite geometric distribution of length N with the probability of the most fit individual to propagate its data p_k set as 20%. This is constant across generations. If the rank 1 probability is less than 50%, it is not a monotonic distribution, and as such I modified the distribution so that it pulls from a finite distribution of length N+1 where the N+1th event represents a re-roll. Because of the memoryless property of the geometric distribution, this is equivalent to mapping the event to a wrapped geometric distribution of base probability p_K set to 20% as given by 9, where chromosome k is the chromosome that gets selected to propagate. This is repeated N times to generate the N chromosomes for the next population.

$$k = \mod(K, N) \qquad K \sim Geom(p_k) \tag{9}$$

2) Crossover Events: Once the new population has been selected, the algorithm iterates through unique pairs of chromosomes and has a crossover event with a predetermined frequency. When a crossover event is set to occur, two chromosomes swap a respective data value. For example if chromosome k_0 with PID Parameters $\langle K_{p,0},\ T_{i,0},\ T_{d,0} \rangle$ swapped with chromosome k_1 with PID parameters $\langle K_{p,1},$

$T_{i,1}$, $T_{d,1}\rangle$, one possible resultant chromosome is $\langle K_{p,0}$, $T_{i,1}$, $T_{d,0}\rangle$. Insofar as this process explores the search space, there are only three unique crossover events: a swap of K_p, a swap of T_i, and a swap of T_d. I set each to occur with equal probability.

3) Mutation Events: After the crossover process is done, the algorithm performs random point mutations at a predetermined frequency. This mutation occurs as a standard normal disturbance added to the previous value as given by Equation 10.

$$X'_k = X_k + n \qquad X_k \in \{K_p, T_i, T_d\} \qquad n \sim \mathcal{N}(0,1) \quad (10)$$

Unlike with the crossover event, the different combinations of what is mutated cannot be reduced, and as such is treated as seven potential events. If a chromosome is selected for mutation, all seven possibilities can occur with equal probability and as such, each mutation event changes 1.7 values of the chromosome on average. Further, because of this mutation event, there lies a potential to generate positive feedback from the PID controller, which is mitigated by forcing all controller parameters that would otherwise go below 0.001 by this mutation to stay at 0.001.

III. Results

At the end of each generation, as characterized by one full cycle of the graph shown in Figure 1, all of the chromosomes had all of their properties written to an extensive database, and the most fit individual of each generation was recorded in a separate database, along with its respective K_p, T_i and T_d values. A plot of the GA in action is shown in Figure 2. The oscillation in the first few generations is expected, for in that time, the diversity outweighs the fitness gained from minimizing the cost function.

As stated previously, because of the different branches that constitute the independent search performed by this algorithm, it lends itself clearly to parallelization. This is shown by Figure 3 and Figure 4. By comparing the two graphs it is apparent that the way the database was implemented caused it to be a detriment to the performance of our system, making it run at least 2.5 times slower. Moreover, instead of the monotonic decrease, we have a spike in runtime every third node added.

IV. Conclusion and Future Work

This paper has shown how GA can be used in parallel on OpenStack clusters and that it generally scales with node count by decreasing the overall runtime. Though our instantiation of the GA could be tuned to better optimize this PID controller, demonstrating the performance of the algorithm itself on this arbitrary example is secondary to demonstrating that it stands to gain from the distributed computing offered by the cloud. Because of the poor results we got from using SQLite3, a clear direction to take this work is towards implementing a higher performing relational database management system such as MySQL or PostgreSQL.

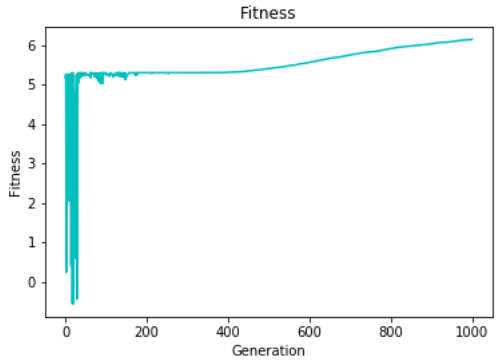

Fig. 2. Fitness score of most fit individual over time

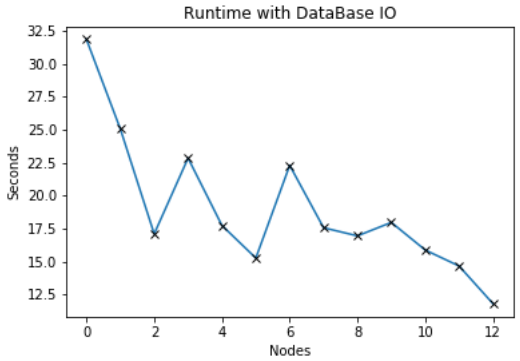

Fig. 3. Runtime with writing to SQLite3 Databases

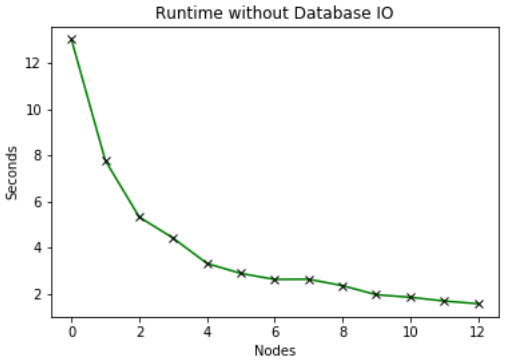

Fig. 4. Runtime without writing to SQLite3 Databases

References

[1] Astrom, K. J. and T. Hagglund (1995): PID Controllers: Theory, Design, and Tuning. Instrument Society of America, Research Triangle Park, North Carolina.