IMPROVEMENTS OF THE PARALLEL EVOLUTIONARY ALGORITHM FOR FINDING SOLUTION OF A SYSTEM OF ORDINARY DIFFERENTIAL EQUATIONS

Jane Jovanovski
Faculty of Computer Science and Engineering
Skopje, Macedonia

Boro Jakimovski
Faculty of Computer Science and Engineering
Skopje, Macedonia

Dragan Jakimovski
Faculty of Natural Sciences and Mathematics
Skopje, Macedonia

ABSTRACT

The goal of our research is to evaluate the general methods of finding solution of a system of differential equations. In this paper we present an improved parallelization approach of the two step parallel genetic algorithm approach that produces an analytical solution of the system. The evaluation of the algorithm reveals its capability to solve non-trivial systems in very small number of generations. In order to find the best solution, and due to the fact that the simulations are computational intensive, we use parallel grid genetic algorithms. Using the gLite based Grid, we propose a grid genetic solution that uses large number of computational nodes, that archives excellent performance. This research will be the basis on our goal of solving more complex research problems based around the Schrodingers equation.

I. INRODUCTION

Almost any problem in all fields of science are expressed in terms of ordinary differential equations (ODE's), partial differential equations (PDE's) or more generally as a system of ordinary or partial differential equations (SODE or SPDE). For that reason many methods have been proposed for solving ODE's and PDE's, such as Runge Kutta, Predictor - Corrector [1], radial basis functions [2] and feedforward neural networks [3]. Also several methods based on genetic programming have also been proposed [4], [5]. The technique of genetic algorithms (GA) is an optimization process based on the evolution of a large number of candidate solutions through genetic operations such as replication, crossover and mutation [6].

Our research focuses on developing an efficient method based on genetic algorithms for finding an approximate analytical solution of a SODE. In this paper we propose a novel method based on two step genetic algorithm for solving SODE. This feature of two step genetic algorithms, dramatically improves the optimization process. We further investigate its parallel implementation using Parallel GA.

The rest of this article is organized as follows: in section II we describe the chromosome function representation, in section III we describe in detail the new algorithm and its implementation, in section IV we provide details about the paralelization of the algorithm, V presents several experiments and in section VI we give our conclusions and ideas for further work.

II. FUNCTION REPRESENTATION AND EVALUATION

Search for the solution of system of ordinary differential equations using genetic algorithms requires an adequate representation of the candidate solution. Since the solution of a SODE is a set of functions, we need a chromosome that specifies a single and thus a set of functions. Computer representation of mathematical functions can be found im many different problems. The solution mainly lies in the fact that any set of correctly formed mathematical functions are based on a context-free grammar. This is the reason why the usual approach for representation or verification of a well formed mathematical function is either a push-down automate (or just a stack), a sequence of context-free rules, or a function tree.

In [5] they choose the approach from Grammatical Evolution for representation of the chromosome. Grammar Evolution is an evolutionary approach to automatic program generation, which evolves strings of binary values, and uses a BNF grammar to map those strings into programs. This mapping process involves transforming each binary individual into a string of integer values, and using those values to choose transformations from the given grammar. Even though this approach has good research background, in our opinion it is too complex and inefficient, due to the fact that the chromosomes can suffer from the "Ripple Effect" [7].

On the other hand in [4] the author opts for Revers Polish Notation for representation of the chromosomes. This approach gives the advantage of easy functional evaluation and function tree reconstruction. We find Polish Notation to be more adequate for function representation since it enables easier implementation of crossover function that we propose.

A. Candidate solution

For the construction of the functions, elements of the candidate solution of the SODE, we define a grammar $G = V, \Sigma, R, S$ for the function representation. The set of non-terminals is fixed to $V = S$. The set of terminals Σ contains the building blocks of the candidate solutions. In our current experiments we chose

$$\Sigma = Variable \cup Constants \cup Functions, \quad (1)$$

where the set $Variables = \{x\}$, the set $Constants = \{c_j | j \in \{1..J\}\}$ that contains the placeholders for the constants that will appear in the solution, and the set of all

accepted functions $Functions = \bigcup_k \left\{ f_{k,l} | l \in \{1..L_k\} \right\}$ where k represents the arity. The set of rules is shown in equation 2.

$$R = \left\{ S \to x, S \to c_j, S \to f_{1,l}S, S \to f_{2,l}SS, S \to f_{3,l}SSS, ... \right\}. \tag{2}$$

In our experiments we used the following sets and values as basis for solving the systems: number of constants J=10 and $Functions = \{sin, cos, exp, log, +, -, *, /\}$.

The chromosomes that represent the candidate solution for the SODE contain:

- one dynamic array per solution function representing a stack that holds the polish notation of the function
- one array for holding the values of the constants that appear in the system
- computed fitness value

Unlike the chromosomes that appear in Grammatical Evolution [7], [5], where chromosomes keep a lot of recess genes and their lenght is of fixed, we choose to keep the chromosomes length dynamic and containing only the functional elements.

B. Fitness function

The evaluation of the fitness of the candidate solution needs to estimate the amount of difference with the correct solution. Naturally, a good estimation of the fitness is to calculate the integral of the difference between the two functions (candidate and solution), or even better, an integral of the square difference between the two functions. The fitness values are always positive, with the best fitness value 0.

In our experiments we simplify the fitness by calculating an approximation of the integral of the square difference by calculating a sum of square differences on an equidistant points. Following is a detailed description of fitness function evaluation.

Lets assume that we want to solve system of differential equations 3, where x is the differentiating variable and we assume that functions are k_i differentiable on the interval $[a,b]$. In order to find a solution, we need to define initial conditions shown in equation 4.

$$\begin{cases} f_1(x,\ y_1,\ y_1^{(1)} \dots y_1^{(k_1)} y_2^{(1)} \dots y_2^{(k_2)} \dots y_n^{(1)} \dots y_n^{(k_n)}) = 0 \\ f_2(x,\ y_1,\ y_1^{(1)} \dots y_1^{(k_1)} y_2^{(1)} \dots y_2^{(k_2)} \dots y_n^{(1)} \dots y_n^{(k_n)}) = 0 \\ \cdot \\ \cdot \\ \cdot \\ f_n(x,\ y_1,\ y_1^{(1)} \dots y_1^{(k_1)} y_2^{(1)} \dots y_2^{(k_2)} \dots y_n^{(1)} \dots y_n^{(k_n)}) = 0 \end{cases} \tag{3}$$

$$\begin{cases} y_1(a) = y_{1a} \\ y_1^{(1)}(a) = y_{1a}^{(1)} \\ \cdot \\ y_1^{(k_1-1)}(a) = y_{1a}^{(k_1-1)} \\ \cdot \\ \cdot \\ y_n(a) = y_{na} \\ y_n^{(1)}(a) = y_{na}^{(1)} \\ \cdot \\ y_n^{(k_n-1)}(a) = y_{na}^{(k_n-1)} \end{cases} \tag{4}$$

We start the evaluation of fitness function by choosing N equidistant points (x_i) on the interval $[a,b]$, $(x_0 = a,\ x_1,\ \dots,\ x_{N-1} = b)$. Since the candidate solution contains n functions (stored as stacks) we will refer to these functions as $M_1, ..., M_n$. The evaluation of the candidate functions is by simulating the execution of $M_i(x_i)$ using an additional stack. Once we finish the evaluation of function points, we estimate the values of the derivatives $M_j^{(k)}(x_i)$ by using the Central Difference Approximation and Ridders interpolation [8]. Computed values are used to evaluate the functions square differenece E using the equations 5. In order to give more emphasis on the initial conditions, we compute the difference from the initial condition and the candidate solution and multiply it using a emphasis factor λ. The computation of this part and the complete fitness function is shown in equations 6 and 7.

$$\begin{aligned} E(M_j) = & \sum_{l=0}^{N-1} (f_j(x_l,\ M_1(x_l), \\ & M_1^{(1)}(x_l), \dots, M_1^{(k_1)}(x_l), \\ & M_2^{(1)}(x_l), \dots, M_2^{(k_2)}(x_l), \dots, \\ & M_n^{(1)}(x_l), \dots, M_n^{(k_n)}(x_l))^2, \\ E = & \sum_{j=1}^{n} E(M_j) \end{aligned} \tag{5}$$

$$\begin{aligned} P(M_j) = & \lambda (M_j(a) - y_{ja})^2,\ \forall j = 1, \dots n \\ P(M) = & \sum_{j=1}^{n} P(M_j) \end{aligned} \tag{6}$$

$$u = E + P \tag{7}$$

III. Genetic algorithm implementation

In this section we will present the two level genetic algorithm that we use to find the approximative analytical solution of the system of differential equations. The two levels are based on the construction of the chromosomes that represent the candidate solutions for the system. The top level optimization focuses on the function structure. This means that the focus on this level are the different candidate solutions, each representing a completely different function set. On the other hand, the bottom level optimization focuses on the optimization of the constants that are introduced in the chromosomes. On this level we focus on each candidate solution, and try to find optimal values of the constants for achieving as optimal as possible fitness values.

Having this, out approach is, for each generation of the genetic algorithm, first to run the bottom optimization on the entire population, and once we have optimal constants for all chromosomes, to start the top level optimization.

Following are the principles and implementation decisions of the genetic algorithm optimization. We will start from the top and proceed to the bottom level of optimization.

A. Initial population

The proper generation of an initial population of chromosomes is crucial for the success of the genetic algorithms. In our approach, we choose to build the initial population by randomly selecting rules from the chosen grammar. We use the George Marsaglia's KISS random generator built into the GNU Fortran and in each step, we first select which category of rule we will use on the first non-terminal character. By category we meen either $S \to x$, $S \to c_j$, $S \to f_{1,l}$ and $S \to f_{2,l}$, since we have only unary and binary functions. Once a category is chosen, we again randolmy choose a rule from that category, and apply the rule. The approach using categories is better than using a single random number to choose from uncategorzed rules, because if there are many constants, or many binary functions, the probability of choosing a terminating character (contant) might be much bigger than choosing a function, and the rule $S \to x$ will be chosen with very low probability. This will result with very nested or very simple functions with many constants.

B. Selection

The initial population is the basis of the optimization. Each generation of the algorithm starts with a selection of the population, in order to decide which chromosomes will survive to reproduce and mutate. The selection algorithm that we used is a variant of the tournament selection with two turns, quarterfinals and semifinals and at the end obtaining the finalists (reproduction set). In each turn we select a random subset of chromosomes and choose the winner that proceeds in the next turn.

C. Crossingover

Once the finalists are selected, we proceed with the reproduction steps using the following crossingover algorithm. The crossingoever between the chromozomes is on the level of separate functions, i.e. the first function of one chromosome exchanges material with all function of the other chromosome. this results with n^2 children from two chromosomes. For the crossing over between two functions, our goal is to select a subfunction of both functions (subtrees of the function trees), and switch the subfunctions. This process is depicted in Figure 1.

The subfunction selection process is implemented by first random selection of the top of the subfunction (random number represenets the index in the array that holds the prefix function tree). Then we itterate towards the end of the array and try to find the closing position of the subfunction. This approach is possible since each subree is a continous segment in the array using the postfix (Polish) notation.

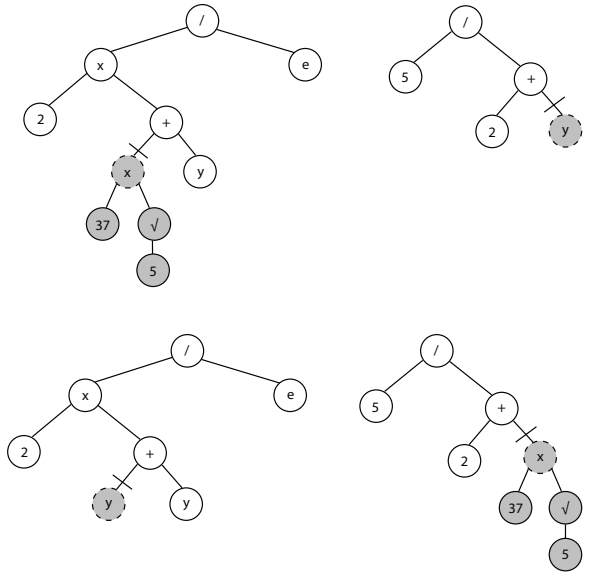


Fig. 1. Cutting and exchange of the subfunctions in the process of crossingover between two functions

This approach for crossingover is much more natural when crossing two functions. A similar approach can be found in [4] where the authors search for a binary operator, and exchange the operator together with one of the subtrees. We find our solution to be more general. On the other hand, when using a Grammar Evolution approach [5] one cannot simply select a closed form subfunction. This results into additional step that need to fix the broken parts of the function.

D. Mutation

After the crossingover generates the new population we proceed to mutation operation. we select a relatively small number of random chromosomes, where we randomly select an element and mutate it with another elemnet from the same set (constant with constant or variable, unary function with unary function, etc.).

E. Constant Optimization

Since we introduced constants into the building process of the candidate functions, we need to define their values prior to solution evaluation. In similar approaches [5] predefine the values of the constants to several integer numbers. On the other hand, in [4] they define constant on every element in the function tree (a more general approach). They continue by optimizing the values of the constants by using perturbation.

In our approach, we use another genetic algorithm optimization in order to select the best values of the constants. The chromosomes represent an array of double precision values, randomly initialized. We implemented the heuristic crossover [9]. We start by selecting two chromosomes from a sorted population (by fitness). The first parrent u is chosen from the upper part (better fitness), and the second v from lower

part of the sorted population. The offspring is generated by first selecting a random parameter $\alpha \in [0,1]$ and generate the offspring genes using the equations 8. All chromosomes enter the selection process, and the result is a new population where the upper part becomes lower part, and the newly generated chromosomes are located in the upper part of the population.

The mutation of the chromosomes is executed by selecting three chromosomes u, v, w and by selecting a single gene u_u, v_v and w_w. Then we compute the new value of $u_u = \frac{v_v + w_w}{2}$.

$$\begin{aligned} u_{inew} &= \alpha(u_i - v_i) + u_i, \quad \text{for } i = 1,\ 2,\ \ldots,\ n \\ v_{inew} &= u_i, \quad \text{for } \ i = 1,\ 2, \ldots,\ n \end{aligned} \tag{8}$$

The benefits of constant optimization are shown in section IV. There we analyze fitness value of best chromosome in population, when we exclude and include constant optimization. When we exclude constant optimization, the value of the best chromosome converges very slowly, needs large number of generations, with large number of chromosomes, for finding an approximate solution, the approximate solution is very long function etc. Those disadvantages are being solved when we include constant optimization.

IV. Parallelization

One of the main characteristics of genetic algorithms and genetic programming as techniques for implementation of evolutionary paradigms is their exceptional ability to be parallelized. This comes from the fact that the individuals can be evaluated in parallel as their performance rarely, if ever, affects that of other individuals. There are numerous ways for paralelization of genetic algorithms, but here we will consider the following two techniques:

- Island GA: The population is divided on several subpopulations - islands, each subpopulation is a population on its own and is developed on a separate processor. After a certain number of generations, all subpopulations are gathered together into a single population to get mixed, after which they are resent to the processors.
- Paralelization of the fitness function: The most used operation in the evolutionary algorithms is an evaluation of the fitness value of each of the chromosomes. The fitness value is evaluated during selection, after crossover, after mutation. Therefore, this operation takes most of the processor time. There are different techniques for paralelization of the fitness function depending on its shape.

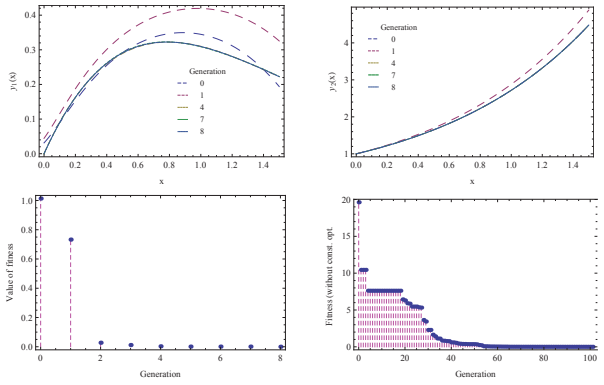
We used the Message Passing Interface (MPI) standard for building the parallel evolutionary algorithm. As it was explained above, the current population consists of chromosomes which contain as many stacks as there are equations in the system that is being solved, and each of the stacks that denotes a postfix representation of the function is represented by a stack with a variable size. As a result of the dynamic size of the arrays and due to the fact that MPI cannot deal with arrays with variable size, it is impossible to divide the population on smaller islands to be sent to the corresponding processors. This reason led us to the implementation of the second technique for paralelization of the fitness function.

From the definition of the fitness function given with the equations (5)-(7), it is clear that the interval $[a,b]$ is searched for the value of the candidate functions. This interval is divided into equidistant points with step ε, resulting with $N = \frac{b-a}{\varepsilon} + 1$ points at most. If there are n processors available, then this interval is divided on n disjoint and neighbouring intervals with at most $\frac{N}{n}$ points each. After the discretization of the initial interval into n subintervals, each of the subintervals is sent to the corresponding processor, i.e. the i-th subinterval is sent to the i-th processor. This implies a paralelization of (5). When the i-th subinterval is assigned to the i-th processor, the processor needs to calculate the sum defined in (5), which means that the value of the candidate functions which are part of the chromosome needs to be evaluated in each of the points. The values obtained by the evaluation of the functions in these points should be replaced in all equations of the system and afterwards the sum of the squares of the obtained results needs to be computed. This gives us an information about how close we are to the exact solution in the subinterval.

Hence, we first divide the interval into subintervals and then each processor computes in parallel the abovementioned value for each of the subintervals. However, there are serial calculations for each point of that interval, i.e. the stacks (expression trees) are traversed with the purpose to evaluate the value of the candidate functions in the current point. In order to avoid this anomaly, we introduce new structure representing a stack array which contains as many stacks as there are points in the subinterval, i.e. the i-th stack of this array stores the development of the i-th point in the evaluation of the candidate function. The number of stack arrays is equal to the number of candidate functions. This enables us to evaluate the values of the candidate functions in all the points of the subinterval with one traversal of the stacks. At the end of the evaluation, the stacks contain one element which stores the value of the candidate function in the corresponding point. Since each of the processors has as many stack arrays as there are candidate functions, at each processor the value of the candidate functions in each point of the corresponding interval is stored. We repeat the procedure described above and at the end we collect the results from all processors, eventuating in the parallel evaluation of the fitness value in the interval $[a,b]$.

V. Evaluaton

In order to evaluate the proposed two step genetic algorithm for SODE, we evaluated its performance, conversation speed and quality of solution. We tested the algorithm on several systems of differential equations with known analytical solution and show results for one example. Wolfram Mathematica's DSolve can not find results for this example. The results from the experiments are very promissing and we beleive that this approach can be used very successfully, especially its parallel implementation, therefore we measure speedup of the parallel algorithm. The process of optimization was run using a replication rate of 15% (i.e. cross over probability 85%) and mutation



$$
\begin{aligned}
y1: &\quad \cos\left(0.9092-0.417x+0.868\cos\left(\cos\left(x\right)\right)\right.\\
&\quad \left.-0.623\log\left(\cos\left(0.74687-x\right)\right)\right)\\
y2: &\quad e^{x}\\
fitness: &\quad 8.9\ 10^{-4}
\end{aligned}
$$

Fig. 2. Solution convergence of the SODE equation 9 by generations

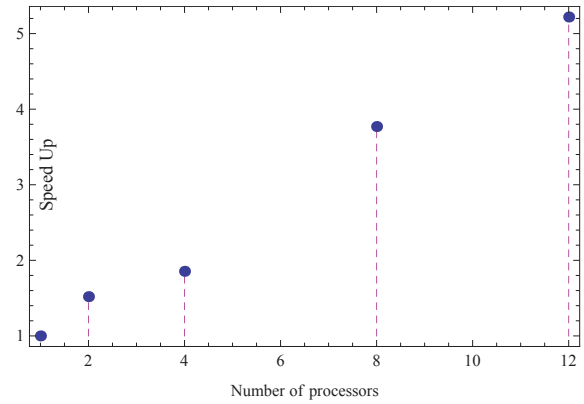


Fig. 3. Speed Up of parallelization with different number of processors

rate of 5%, $\lambda = 100$, over a population of 100 chromosomes. In the tournament selection algoritm we choose the number of quaterfinalists to be 25, semifinalists to 5 and the number of finalists that are generated is $(cross\ over\ probability) * (number\ of\ chromosomes\ in\ the\ generation)$. In genetic algorithm for optimization of the constants values we set population size to 200 and number of generation to 100. When we exclude constant optimization we use 10 constant with value $\{0,1,2,...,9\}$.

A. Example

The system we tested was the system show in equation 9 with initial conditions 10. The solution of this system is $y_1 = \frac{\sin x}{e^x}$ and $y_2 = e^x$. In our search for a solution, we chose the interval $x \in [0,\ 1.5]$ with step $0,005$ (301 discrete points in the interval). The correct solution was found in the 8th generation. The obtained solution of the SODE equation 9 is shown in Figure 2.

$$
\begin{cases}
y_1' = \frac{\cos x\ -\sin x}{y_2} \\
y_2' = y_1 y_2 + e^{x} - \sin x
\end{cases} \tag{9}
$$

$$
y_1(0) = 0, y_2(0) = 1 \tag{10}
$$

When we exclude constant optimization we find approximate solution in the 100th generation with fitness value of $3.012\ 10^{-3}$. After the 55th generation convergence is very slow.

In order to show the benefit of parallelization we measure the speed up in ratio to the time of running of serial algorithm. The result is shown in Figure 3. it can be concluded that for 12 processors the speed up is around 5 times faster.

VI. Conclusion and Future Work

In this paper we presented a very successfully implementation of a parallel genetic algorithm for solving SODE. Our two step optimization enables bigger flexibility of the solutions and makes the process of optimization very successful even with very small number of generations. The new optimization achieves substantial parallel speedup comparing with the original solution.

Even though we achieved great result, we will continue to optimize the algorithm and its implementation, mainly by using hybrid parallel implementation on HPC clusters. We want to further investigate different initial population generation approaches, where we won't use uniform distribution to select the subset of rules. Another interesting issue is the fact that in the currently generated candidate solutions we rarely see elements such as polynomials. We believe that this problem is related to the approach for building the initial set of chromosomes and that these elements will give much better candidate solutions.

Acknowledgement

This paper is based on the work done in the framework of the HP-SEE FP7 EC funded project.

References

[1] Lambert, J. D.: *Numerical methods for ordinary differential systems: the initial value problem*, John Wiley & Sons, Inc. (1991)

[2] Fasshauer, G.E.: *Solving differential equations with radial basis functions: multilevel methods and smoothing*, Advances in Computational Mathematics, vol 11, pp. 139–159, doi: 10.1023/A:1018919824891 (1999)

[3] Lagaris, I.E., Likas, A., Fotiadis, D.I.: *Artificial neural networks for solving ordinary and partial differential equations*, IEEE Transactions on Neural Networks, vol 9, pp. 987–1000 (1998)

[4] Burgess, G.: *Finding Approximate Analytic Solutions To Differential Equations Using Genetic Programming*, Technical Report DSTO-TR-0838, Surveillance Systems Division, Defence Science and Technology Organisation, Australia, Salisbury, SA, 5108, Austrlia (1999) http://www.dsto.defence.gov.au/corporate/reports/DSTO-TR-0838.pdf

[5] Tsoulos, I.G., Lagaris, I.E.: *Solving differential equations with genetic programming*, Genetic Programming and Evolvable Machines, vol 7, pp. 33–54 (2006)

[6] Goldberg, D.E.: *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley Longman Publishing Co., Inc. (1989)

[7] O'Neill, M., Ryan, C., Keijzer, M., Cattolico, M.: *Crossover in Grammatical Evolution*, Genetic Programming and Evolvable Machines, vol 4, pp. 67-93 (2003)

[8] Ridders, C.: Accurate computation of F'(x) and F'(x) F''(x). Advances in Engineering Software (1978), vol 4, pp. 75 – 76, doi: 10.1016/S0141-1195(82)80057-0 (1982)

[9] Sivanandam, S.N., Deepa, S.N.: *Introduction to Genetic Algorithms*, Springer Publishing Company, Inc. (2007)