Genetic algorithm for scheduling parallel jobs in a class of financial systems

Milena Karova, Stanislav Stoyanov, Ivailo Penev
Technical University of Varna, Varna, Bulgaria, milena.karova@gmail.com

Abstract: The paper presents the problem about scheduling of parallel jobs with concurrent access to a common data source in a class of financial systems. A genetic algorithm for simulating various schedules, based on predicted execution times, is proposed. The algorithm aims to define a schedule, preventing the concurrent access to the data source, which would increase the performance of the parallel realization. Experimental results from the simulation of schedules, obtained by the algorithm are presented.

Keywords: parallel job scheduling, genetic algorithm

1. INTRODUCTION

The work is concerned with programming systems, typically used by financial institutions. These systems perform estimations of funds, containing sets of financial portfolios. The estimations are used for making investment decisions. They are obtained by executing various simulation analyses over the positions of each portfolio. Each position is described by historical data for a specific financial instrument (deposit, credit, fund, option, bond, etc.). Furthermore the simulated portfolios are independent each other. As data dependencies between the portfolios do not exist, the simulations could be performed in parallel. In this case the estimation of a portfolio could be realized as a separate job.

The historical data about the past periods of a financial object are stored in a data source (typically data base). The simulations of multiple objects, performing in parallel, cause concurrent access to the data source, which is a limiting condition about the efficiency of the parallel realization [4].

A key issue of the realization in a distributed environment is scheduling the parallel jobs’ execution. The problem is concerned with deciding in which moment and in which computing node a job is to be started. In the context of the financial systems described above the algorithm should construct schedules so that the parallel executing jobs do not access the data source concurrently.

As the whole data about a portfolio to be simulated is known in advance, the parallel jobs are assumed to be completely defined. Various schedules could be simulated in order to find the one with no or minimum concurrent access to the common data source. Because of the great number of jobs in a typical system, the problem for finding a suitable schedule could be described as a combinatorial optimization problem.

The class of evolution and especially the genetic algorithms is widely applied to solving scheduling problems, for example [2]. A genetic algorithm could be useful for obtaining high quality or even optimal schedules according to a chosen optimality criterion.
2. FORMAL DESCRIPTION OF THE PROBLEM

The estimation of a portfolio is defined as a job, consisting of three stages: reading data from a common data source, performing calculations according to simulation analyses, storing result data into the source. The time about each stage could be predicted. In [6] the evaluation of a regression mathematic model for obtaining times of jobs’ stages depending on the type and count of the consisting instruments is presented.

From viewpoint of the theory the problem for scheduling the jobs specified is classified as $P||\sum C_i$ [3]. In brief this means that a set of $n$ independent non-preemptive parallel jobs has to be executed by a set of $m$ identical computing nodes. At any moment a node is able to perform not more than one job and a job could be performed by not more than one node.

Lots of scheduling algorithms solving such problems are known. Detailed study of these algorithms could be found in [1, 3]. However, no scheduling algorithm avoiding the concurrent access of parallel jobs to a common data source is described.

In previous works the authors have proposed an approach for avoiding the concurrent access to a common data source, using the idea with delayed start of the jobs [5]. Multiple scenarios with shifted start of the jobs on the time axis are simulated. The final purpose is finding a scenario (schedule), in which the delay starting prevents completely or minimizes the concurrent access.

The great number of jobs in a typical system causes too many possible schedules. Therefore finding a suitable schedule by the use of traditional iterative algorithms is not effective. A genetic algorithm for faster producing of an optimal schedule could be used.

3. GENETIC ALGORITHM FOR PRODUCING A SOLUTION WITH MINIMUM CONCURRENT ACCESS

3.1. Algorithm description

3.1.1. Chromosome representation

The coding of the chromosomes is the following: each chromosome has a number of computers ($P$) - objects of class Computer, stored in a vector. Each of them executes a number of jobs, also stored in a vector.
The class `Computer` contains a field "order", determining the order in which the computers begin executing their first job. Each Nth computer in the order will execute after the reading stage of the first job in the N-1st computer in the order is done.

The class `Task` contains a field "delay", which indicates the amount of time the computer has to wait, before starting its job.

The jobs from the table are distributed evenly among the computers in the following way: initially, the jobs are sorted by total execution time (i.e. sum of all the stages’ times) in ascending order. After that, the first P jobs are assigned respectively to the first P computers. The other jobs are assigned in a reverse order, respectively to computers from P to 1. These two steps alternate until all jobs are assigned.

All populations are stored in a vector and the first ones are deleted if the number of populations exceeds the populations in memory limit. Each population after the first is created as a copy of the previous, and mutation is applied to each chromosome of the new population. Each population contains 10 chromosomes (solutions).
Also, each computer contains a dynamic bitset (as a field). This bitset is formed from the field "order" of the computer, the field "delay" of each of the jobs, which it executes, and the lengths of its three phases - reading, calculations and storing. The phases, in which there is a risk of intervals with concurrent access (reading and storing), are written in the bitsets as series of ones, and the phase "calculations", "delay" and "order" are written as series of zeros. The writing is done successively - "order", "delay", A, B, C, "delay", A, B, C...

![Fig. 5: Forming a bitset.](image)

The goal function of the chromosome is calculated in the following way: the bitsets of each two computers are ANDed and the resulting bitset's number of ones is added to the sum of collisions (time intervals with concurrent access). When this operation is done for all couples, this sum represents the goal function.

### 3.1.2. Genetic operators

The operation mutation is performed for a randomly selected job from a randomly selected computer from the chromosome. It's determined if the job's movement on the time axis can decrease the total number of collisions. This is done by calculating the goal function for the computer - in a similar way like the goal function of the chromosome is calculated, but only for this computer. If a new favorable position of the job (in time) exists, the job is moved.

The operator crossover is not applied in this algorithm, because when exchanging computers between two solutions, there are often more collisions than before the crossover, due to the solutions' different layouts. In one such exchange, the other computers' fields will have to be changed too, to avoid the new collisions, which is ineffective.

When the goal functions of all chromosomes are zeroes, the last 5 solutions are created again, like the solutions of the initial population. In this way, new solutions can be found even after many populations, after all collisions are avoided, which helps in minimizing the total time.

### 3.2. Programming environment for visualizing various scenarios

In order to test the work of the genetic algorithm presented above, a programming environment with suitable interface is developed. The interface visualizes the jobs to be scheduled with their times about each stage and the solutions, produced by the genetic algorithm.

An example schedule, produced by the algorithm is shown on the next figure.
4. EXPERIMENTAL RESULTS AND CONCLUSIONS

The tests are done with 5 computers, executing 10 jobs, which times are loaded from a text file:

<table>
<thead>
<tr>
<th></th>
<th>Read</th>
<th>Calc</th>
<th>Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>28</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>30</td>
<td>86</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td>90</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>44</td>
<td>284</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>26</td>
<td>131</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>43</td>
<td>187</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>62</td>
<td>407</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>47</td>
<td>297</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>43</td>
<td>434</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>28</td>
<td>81</td>
<td>1</td>
</tr>
</tbody>
</table>

In the solutions of the initial population there is an average of 150 collisions (time intervals/slots with concurrent access), and the time of executing all jobs is at least 603 seconds (time slots).

After pressing the button “Solve”, at the 51st population, the collisions in all solutions are reduced to 0, and the best solution is with total time 784 seconds. Reaching this population takes 0.16 seconds.

We press “Solve” again, but with a different jump and we reach the 10,051st population. The solutions are formed after 22.56 seconds. The total execution time of all jobs is minimized to 651 seconds.

After reaching the 20,051st population, the minimal total time is 631.
The future work will be referred to investigating the possibilities of the algorithm for realizing real case with scheduling great number of jobs. Further comparison of the algorithm to other known algorithms for finding suitable schedules will be done.

5. REFERENCES


