Project Documentation

Theme chosen: Generate a school timetable through genetic algorithms.

Making a timetable might be difficult to achieve when the input is complex, so it can be done by using genetic algorithms, as there are many requirements which must be taking into account, such as: professors, groups of students, classes, classrooms, laboratory equipment and so on. The search of the best solution can be parallelized, so the performance of the algorithm is improved by splitting the work to multiple threads/processes.

Entities needed for a school schedule:  
- **Professor**: has an id, a name and the list of classes the prof. teaches  
- **Classroom**: has an id, a name, a capacity and it contains the information if the room is a laboratory  
- **Course**: has an id and a name  
- **StudentsGroup**: has an id, the number of students in that group and contains a list of classes that the group attends  
- **CourseClass**: holds a reference to the course to which the class belongs, a reference to the professor who teaches, and a list of student groups that attend the class. It also stores how many seats (sum of student groups' sizes) are needed in the classroom, if the class requires computers in the classroom, and the duration of the class (in hours)  
  
For implementing the genetic algorithm, we need to represent the chromosome, the individual of a population, which is actually a possible solution to our problem. Its representation must be feasible for genetic operations such as crossover and mutation and we should be able to calculate the fitness value of our solution.  
  
The **Schedule** class is used to represent a chromosome, which is be described as:  
A time-space slot is needed for each hour, for every room, each day so that class placement in the timetable can be kept. We assume that classes begin at 8am, and should finish before or at 8pm (12 hours total), and working days are from Monday to Friday (5 days total). For this, we use a vector with a size 12\*5\*number\_of\_rooms. The slot is a list, because during the execution of our algorithm, we allow multiple classes during the same time-space slot. There is an additional hash map which is used to obtain the first time-space slot at which a class begins (its position in vector) from the address of the class' object. Each hour of a class has a separate entry in the vector, but there is only one entry per class in the hash map. The Schedule also stores as attributes the specific values used by genetic operations: fitness, criteria(a list of Booleans to check if a criterion is met, so the score of a solution can be increased), numberOfCrossoverPoints, mutationRate, mutationProbability, crossoverProbability.

**Fitness** - represents a value which is calculated by taking into the consideration the requirements below, being computed in the following way:  
Each class can have 0 to 5 points. If a class uses a spare classroom, we increment its score. If a class requires computers and it is located in the classroom with them, or it doesn't require them, we increment its score. If a class is located in a classroom with enough available seats, the score is incremented. If a professor has no other classes at the time, we increment the class's score once again. The last thing that we check is if any of the student groups that attend the class has any other class at the same time, and if they don't, the score increases. The total score of a schedule is the sum of points of all classes. The fitness value is calculated as score / number\_of\_classes\*5. The fitness value is represented as a double.

**Crossover -** a crossover operation combines data in the hash maps of two parents(chromosomes), and then it creates a vector of slots according to the content of the new hash map. A crossover 'splits' hash maps of both parents in parts of random size. The number of parts is defined by the number of crossover points in the chromosome's parameters. Then, it alternately copies parts from parents to the new chromosome and forms a new vector of slots.

**Mutation** - a mutation operation just takes a class randomly and moves it to another randomly chosen slot. The number of classes which are going to be moved is defined by the mutation rate in the chromosome's parameters.

1. Regular threads implementation

The genetic algorithm is performing the next four operations for each generation: crossover, mutation, evaluate and selection.  
The crossover operation randomly selects pairs of parents from the current population and produces new chromosomes by performing a crossover operation on the pair of parents, depending on the crossoverProbability. It is parallelized using a ForkJoinPool which executes on all available processors recursive actions in order to split in two halves the performing of the crossover operation. We set a threshold, so that if the range to be processed is smaller than the threshold, the crossover is performed sequentially. In a similar manner, the parallelization of the mutation and evaluation is implemented. Mutation is applied to an individual, depending on the mutationProbability and then the selection of individuals for replacement happens, but the best chromosomes can’t be replaced. The evaluation operation uses an array of the best chromosomes and an array of booleans in order to flag if an individual at a specified index is in the best array. The evaluation is also split in recursive tasks, which use the synchronized block to handle the updating of the bestSchedule array, ensuring that multiple threads won't interfere with each other during the update.

After generating possible solutions for a predefined number of generations, the best solution is chosen from the final generation.

1. MPI implementation

The message passing interface (MPI) is used in order to assign to multiple nodes to compute the fitness of each individual of the population. The master worker is the one that initializes the population, performs the other operations corresponding to the genetic algorithm and, for the last generation, sends the start and end indexes alongside the array of individuals in that range in order to one of the other processes to compute the fitness of the chromosomes in that range of population. After all workers compute the fitness for their assigned range of individuals, the master process evaluates the population and selects the best chromosome after a predefined number of generations.

Performance measurements:

Data that have been tested on contain the following parameters:  
- total classes: 27  
- mutationProbability: 50  
- crossoverProbability: 70  
- mutationRate: 4  
- populationSize: 16  
- numberOfGenerations: 20

For the above data, the execution times are:

- threads implementation (nr of threads = nr of available processors): 70 ms  
- mpi implementation (with 5 nodes): 200 ms  
- mpi implementation (with 7 nodes): 230 ms

Reference: <https://www.codeproject.com/Articles/23111/Making-a-Class-Schedule-Using-a-Genetic-Algorithm#Professor3>