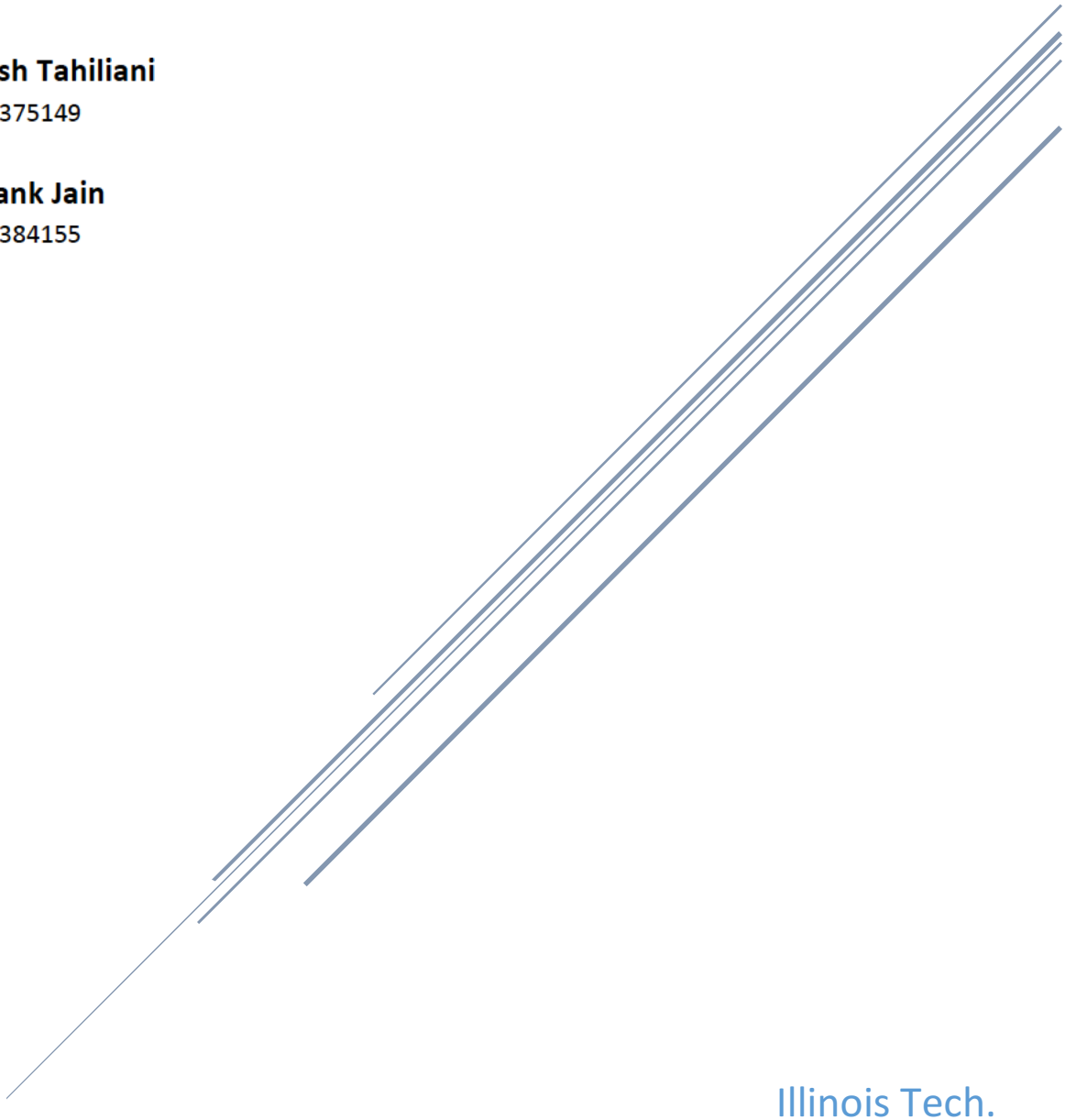
A PARALLEL DISTRIBUTED GENETIC ALGORITHM FOR CLASS SCHEDULING PROBLEM AND ITS PERFORMANCE ANALYSIS ON MPICH, CUDA & APACHE SPARK

Project Report

Ajay Ramesh
A20384062

Venkatesh Tahiliani
A20375149

Divyank Jain
A20384155



Illinois Tech.
CS 546

Table of Contents

Genetic algorithm(GA) is an idea introduced in 1967, its core roots are from the Darwin's theory of evolution. It is widely used in many fields relating to combinatorial optimization, machine learning, network routing, etc. We will design a parallel distributed genetic algorithm for Class scheduling problem , this problem is categorized as multi-constrained, NP-hard, combinatorial optimization problem . Our goal is to implement the designed algorithm for class scheduling in CUDA ,MPICH [and Apache Spark platforms to study on performance.

Motivation and background information

If you are scheduling class time table manually, then it will require a lot effort and might lead to inadvertent errors.

Effective solution is already generated using Genetic Algorithm , but not distributed parallel algorithm . Using Distributed GA will decrease the time complexity also it will make a win-win situation across students and lectures.

Platforms such as CUDA ,MPICH and Apache Spark are used in so many innovative applications which motivated us to learn those and study the performance of distributed genetic algorithm across those platforms .

Problem Definition

Generate the time-table which tells that teacher X is taking class Y on time Z. Consider a typical high-school, here class receives five hours of lessons, six days a week. Teachers may teach one or more subjects, usually in two or more classes. Each teacher has 18hr per week.

Typically, every school has these constraints. Some of them are hard constraints and some are soft constraints.

Some of the Hard constraints -

- Teacher can only be in one class at any given time
- Classrooms need to be big enough to host the class
- Classrooms can only host one class at any given time

Some of the Soft constraints -

- Room capacity should be suitable for the class size
- Preferred classroom of the professor.
- Preferred class time of the professor.

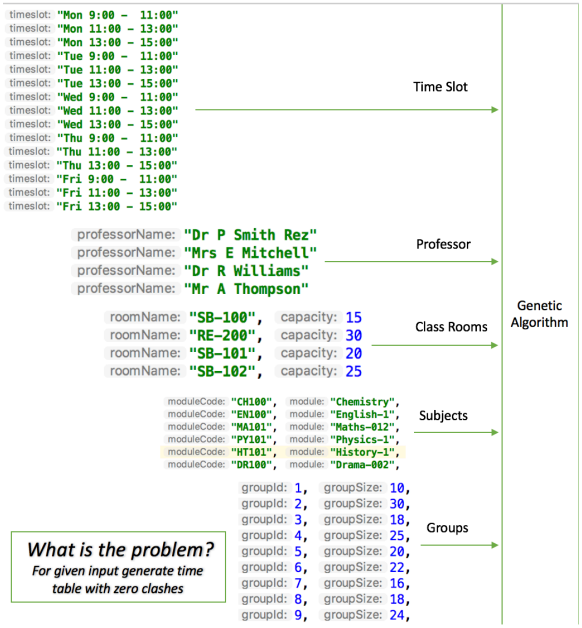


fig 1 – Mapping across different Objects to generate time table

Key term - Population

We identified the most critical part or heart of the genetic algorithm is in its population and evaluating its population. The randomness brings the results over repeated cross over and mutation. Let's talk about Population for some time in order to understand our solution better.

Population contains the chromosomes . Each chromosome represents a time table.
Here we have converted Chrome some as an ***linear array*** of N x 3 matrix. Where N is number of classes , 3 is its attributes describing timeslot , room Id and professor Id.

$$Chromosome\ [N * 3] = \begin{bmatrix} timeslotID_1, roomID_1, profID_1 \\ timeslotID_2, roomID_2, ProfID_2 \\ \dots\dots \\ timeslot_n, roomID_n, ProfIDn \end{bmatrix}$$

Max $_{\text{Population Size}}$ =(Total$Timeslots * TotalRooms * No\ Of\ Prof)^N$

Example $_{\text{population size}}$= (15 time slot * 4 rooms * 4 Prof) ^{26}

The size of chromosome depends on number of classes. For real complex application it will grow very big in size. We have used 7000 random population to generate the time table.

TTG – Spark

Parallel Genetic Algorithm Application for Time Table Generation Using Apache Spark .

Existing Solution :
I have implemented parallel version of the code from scratch over the well know serial version of the code in java. Analyzed its performance for different population size.

Solution Design & Algorithm

In this, I have used the Spark to implement the parallel code. Here I have used Java, Lambda expressions, and Spark functions to achieve it. I have devised more general version parallel algorithm of it so that any problem can apply this algorithm using in Apache Spark.

Parallel Genetic Algorithm for Apache Spark

Ajay Ramesh

December 2, 2016

1 Parallel Genetic Algorithm for Apache Spark

Parallel genetic algorithm is stated here ,the same is successfully implemented for Class scheduling problem.

Algorithm 1 Parallel Genetic Algorithm

```
1:  procedure ParallelGeneticAlgorithm(maxIteration = 1000)
2:      Population ← initalPopulation
3:
4:      PopulationRDD ← Population
5:      Cache ← PoplulationRDD
6:
7:      PopulationRDD ← evalutePopulation(PoplulationRDD)
8:      Cache ← PoplulationRDD
9:
10:     SortedPopulationRDD ← sortInParallel(PopulationRDD)
11:
12:     broadcast(SortedPopulationRDD)
13:     generation ← 1
14:
15:     while terminationConditionMet() ≠ 1 do
16:
17:         PopulationRDD ← crossOverPopulation()
18:         Cache ← PoplulationRDD
19:
20:         PopulationRDD ← mutatePopulation()
21:         Cache ← PoplulationRDD
22:         PopulationRDD ← evalutePopulation(PoplulationRDD)
23:
24:         SortedPopulationRDD ← sortInParallel(PopulationRDD)
25:         broadcast(SortedPopulationRDD)
26:
27:         generation ← generation + 1
28:     return populationRDD.getFittest()
```

figure(a)

Solution design in detail

Now we have the Hard Constraints to satisfy the problem, where we need to check each chromosome for satisfiability of Hard Constraints. If it does not meet it will be allocated with a proper fitness value based on its clashes. Fitness function will loop over existing chromosome to find the fitness. That is how many constraints are satisfying. Calculating fitness requires at least $O(n^2)$ computation cost for each constraints . In order to optimize I have followed below techniques.

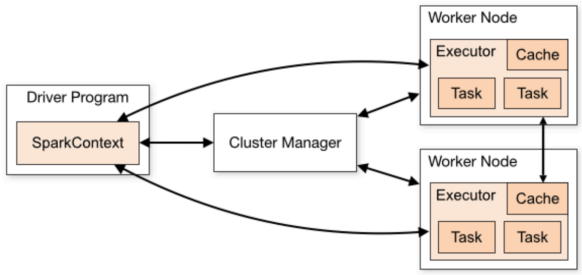
Partition of Data & Caching

In Apache Spark, I have parallelized the Entire Population into RDD. Using below code.

```
private static JavaRDD<Individual> populationRDD;
private static JavaRDD<Individual> sortedPopulationRDD;
private static Broadcast<List<Individual>> broadcastVar;

public static void setPopulationRDD(List<Individual> population) {
    populationRDD= SparkUtil.getSparkContext().parallelize(population);
    sortedPopulationRDD = SparkUtil.getSparkContext().parallelize(population);
    PopulationRDD.setSortedData(population.size());
    populationRDD.cache();
}
```

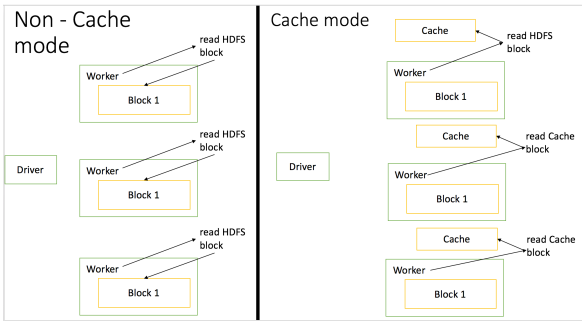
Now the RDD is partitioned depending on the number of Worker nodes. Worker nodes will contain the cached data of Population RDD, thus making it faster computation of fitness values. Across each node. Each node will calculate the fitness value in parallel. Once calculate fitness function evaluates all the chromosome, then it will be collected to form the new population.



Collection and mapping code snippet –

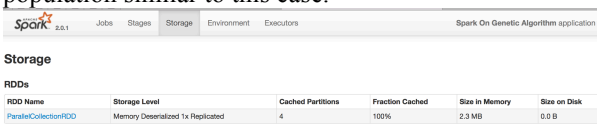
```
List<Individual> processedPopulation = PopulationRDD.getPopulationRDD().map(
        iChromesome -> {
    Timetable tt = new Timetable(timetable);
    tt.createClasses(iChromesome);
    double fitness = 1 / (double) (tt.calcClashes() + 1);
    iChromesome.setFitness(fitness);
    return iChromesome;
}).collect();
```

Caching is the flawless technique which is making spark faster. The figure (b) presents how caching in Spark work.



figure(b) Caching in Spark

Spark Web UI depicting Fraction of Cached Data. In this case, it is 100% of the data is cached; it applies to all the genetic algorithm problem design. Every genetic algorithm for most of the problem has the population similar to this case.



Sorting using Lambda expression

In the serial code below sequence is happening for the gigantic amount of time. The reason is it requires the sorting based on the fitness value.
1. Pick Random chromosome
2. Cross Over it with the fittest chromosome.
3.The resulting chromosome evaluated for its fitness.

Sorting is well known big data problem, in our case we need to sort it using the chromosome fitness value. Here I have written code for sorting using fitness value in spark. To keep simple, I will explain through sorting of Homes collection using its price example.

```
List<Home> homes; // initialize it  with some data
JavaRDD<Individual> homeRDD =  SparkUtil.getSparkContext().parallelize(homes);
   public class Home implements Serializable,Comparable<Home>{

        private double price = Math.Random() * 1000;

         @Override
        public int compareTo(Home o) {
            return Double.compare(this.getPrice(),o.getPrice());
        }
    }
```

Now use the apache spark inbuilt function to sort the array which will happen in parallel way. Here the first parameter below is a lambda expression for sorting it by price.

```
JavaRDD<House> houseRDD = houseRDD.sortBy(  i -> {return i.getPrice(); }, true, 1 );

houseRDD.top(4); // this will output top 4 houses
```

Broadcast sorted data to each stages

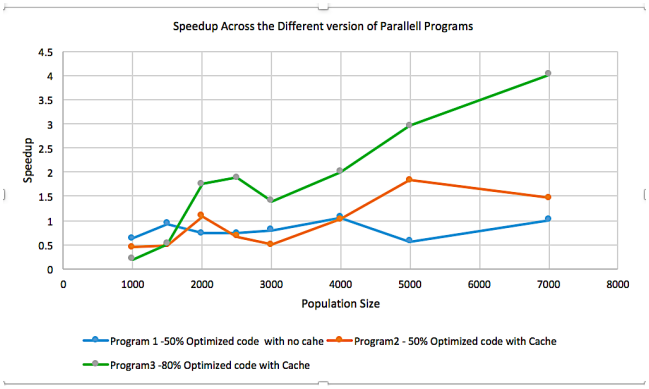
Broadcast variables allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks. They can be used, for example, to give every node a copy of a large input dataset in an efficient manner. Spark also attempts to distribute broadcast variables using efficient broadcast algorithms to reduce communication cost[].

Spark actions are executed through a set of stages, separated by distributed "shuffle" operations. Spark automatically broadcasts the common data needed by tasks within each stage. The data broadcasted this way is cached in serialized form and deserialized before running each task[].

This means that explicitly creating broadcast variables is only useful when tasks across multiple stages need the same data or when caching the data in deserialized form is important. In this application multiple stages using sorted population to pick the fittest chromosome. So broadcasting the data is highly useful in achieving performance.

Performance

I have tried 3 different versions. I have used above solution design to implement it.
Version 1 – *50% optimized with no cache.*
In this I have not used cache , broadcast, sorting techniques.
Version 2 - *50% optimized code with cache.*
Same as version 1, but with cache mode . I see a little increase in performance
Version 3 – *80% of optimized code with cache*
Implemented the algorithm(figure(a)) and all the technique described in solution design , the green line depicts the speedup. Its performance 4 folds increased compared to serial code.



Why Spark ?

Speed

Run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk.

Ease of Use

Write applications quickly in Java, Scala, Python, R.

Generality

Combine SQL, streaming, and complex analytics.

Runs Everywhere

Spark runs on Hadoop, Mesos, standalone, or in the cloud. It can access diverse data sources including HDFS, Cassandra, HBase, and S3.

Future Scope

- *These algorithms were present from very long time , now we have the computation power to leverage it. Spark gives easier and flexible approach.*
- *This Project will be an example of how spark solved the time table generation problem. I have hosted the source in GitHub - ubiquitous-eureka.git] for anyone to extend/learn from this. Imagine TSP problem using Genetic Algorithm in Parallel Way using Spark!*

TTG – Cuda

In aspects on parallelism that can be achieved using cuda, out of the four stages of this algorithm. We identify three stages which have looping constructs which can be parallelized, the first one is the initialization where we create the initial population and the following two stages are crossover and evaluation. The latter functions involve one calculate fitness computation which determines the fitness of a population. We proceed by converting these two functions to their parallel form.

Existing Solution:

A serial implementation can be found online[3] although an implementation involving improved performance with Cuda has not been done yet. Possibly because of the complexity of the lot of variable parameters and smaller functions in cuda which contribute to the overall performance. We have identified some smaller functionalities which can be parallelized and optimized and implemented them. These weakness refer to the overall genetic algorithm weakness, a reason why people don't scale up and achieve better performance is because of the local optima and fitness computation in large scale[2].

Approach:

We started by running smaller sample functions on Jarvis and proceeded to maximize the size of the problem. We also ran smaller code segments using JCuda which is a Java library for launching C kernels, here an independent Java program is written for managing all the serial function and also the cuda memory allocation and an independent C Kernel is written in C for the cuda threads. We started solving this problem with a Java development environment for all the three platforms however it was not easy to convert conventional serial Java code to parallel, It was difficult to proceed with the conversion of the sequential Java classes mainly because of the built in functions, string comparison, object creation and object level comparison and also the passing of objects between functions. Due to shared space on the cluster and cluster outage issues we moved our development in Cuda to a virtual image on Amazon AWS EC2 Cloud[1] which provided a training virtual image with the Tesla M2040 GPU Processor.

Implementation:

The serial version for the initialization function[3] is given as:

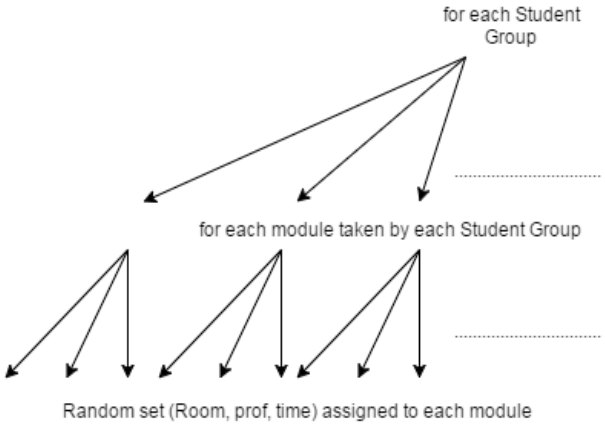
For simple understanding, a chromosome is a combination of [Timeslot, Room, Professor] that is assigned to each course module which is taken by each student group.

```
for (Each Student Group) {
// Loop through modules
    for (Each module taken by the student){
        // Add random time
        int timeslotId = timetable.getRandomTimeslot().getTimeslotId();
        newChromosome[chromosomeIndex] = timeslotId;
        chromosomeIndex++;

        // Add random room
        int roomId = timetable.getRandomRoom().getRoomId();
        newChromosome[chromosomeIndex] = roomId;
        chromosomeIndex++;

        // Add random professor
        Module module = timetable.getModule(moduleId);
        newChromosome[chromosomeIndex] = module.getRandomProfessorId();
        chromosomeIndex++;
    }
}
```

A pictorial representation of the serial version of initialization can be given as:



Here we see that each student group has ‘n’ modules and each module ‘i’ belonging to set ‘n’ is assigned a chromosome [Room, professor, Time]. Moving on to the parallel implementation we have the following change.

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
{
    int chromosomeIndex=i*3;

    *result = curand(&state) % ((timeSlotSize)-0)+0;
    deviceOutput[chromosomeIndex] = = timeslots[*result];;

    chromosomeIndex++;

    // Add random room
    *result = curand(&state) % ((roomsSize)-0)+0;
    deviceOutput[chromosomeIndex] = rooms[*result];
    chromosomeIndex++;

    // Add random professor
    *result = curand(&state) % ((maxProfSize)-0)+0;
    deviceOutput[chromosomeIndex] = ppm[*result];
    chromosomeIndex++;
}
```

In this case we see a that each thread generates a random chromosome for each module taken by each student and assigns it to a global array. A logical view is as follows:

Here, assume an example sample if we have 26 modules taken by various student groups then we have to generate 26*3 chromosomes and each thread generates 3 chromosomes in random for each module.

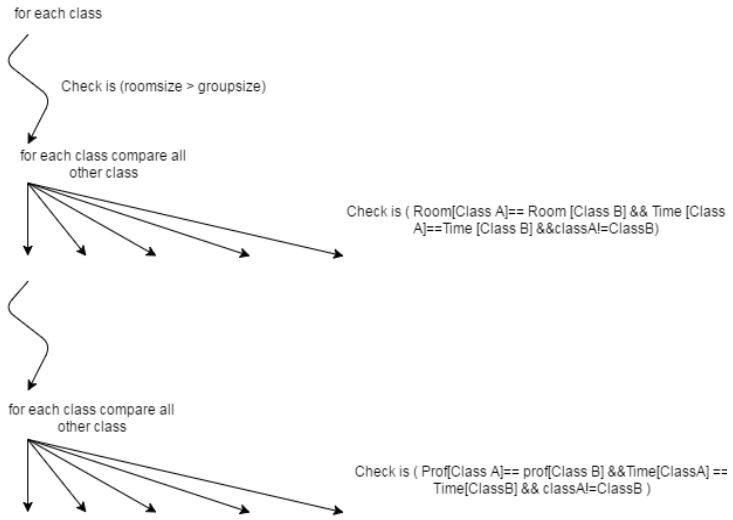
The next function that we parallelized using Cuda was the calculation of fitness. This function involved the comparison of a class with other similar classes and check if constraints are violated or not, in case if constraints are violated we do not add a penalty but if constraints are violated we add a penalty which results in an overall decrease in fitness.

The serial evaluation function snippet[] looks as follows:

```
for (Class classA : this.classes) {
// Check room capacity is greater than room size
    if (roomCapacity [Class A] < groupSize[Class A]) {
        clashes++;
    }
    // Check if room is taken or engaged in the same timespace and is not the same class
    for (Class classB : this.classes) {
        if (Room[Class A]== Room [Class B] && Time [Class A]== Time [Class B] && classA!=ClassB) {
        clashes++;
        break;
    }
}
// Check if professor is available in the same timespace and is not the same class
for (Class classB : this.classes) {
    if (Prof[Class A]== prof[Class B] && Time[Class A] == Time[Class B] && classA!=ClassB) {
    clashes++;
    break;
}
```

There are a series of 3 for loops and 2 of which are nested within the parent loop. Basically the function here checks if for every class A there exists no other class which has the same professor or using the same class room during the same time span and checks if the student size of current class is not less than the size of the classroom so that the students can be seated.

The above code can be logically represented as:

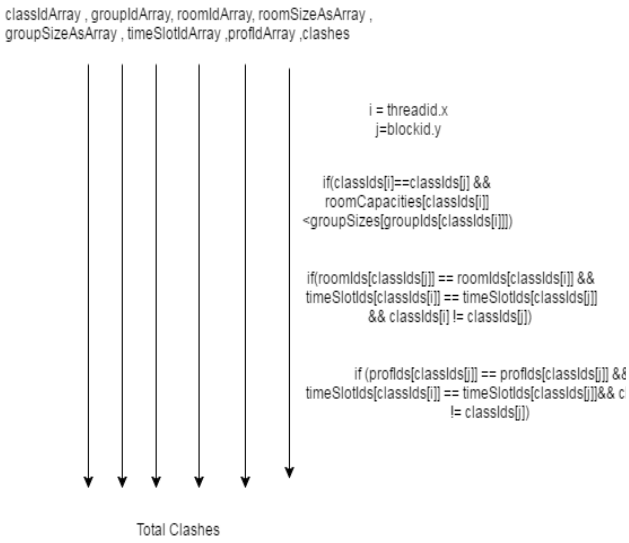


Here we have an opportunity for parallelism, we supply each block with a class and each thread with a class. Hence block and thread index can be used for comparison of properties of one class with another to identify any clashes.

The following code represents a parallel idea:

```
for (Class classA : this.classes) {
// Check room capacity is greater than room size
    if (roomCapacity [Class A] < groupSize[Class A]) {
        clashes++;
    }
    // Check if room is taken or engaged in the same timespace and is not the same class
    for (Class classB : this.classes) {
        if (Room[Class A]== Room [Class B] && Time [Class A]== Time [Class B] && classA!=Class
        clashes++;
        break;
    }
}
// Check if professor is available in the same timespace and is not the same class
for (Class classB : this.classes) {
    if (Prof[Class A]== prof[Class B] && Time[Class A] == Time[Class B] && classA!=ClassB) {
    clashes++;
    break;
}
```

A logical representation is as follows:



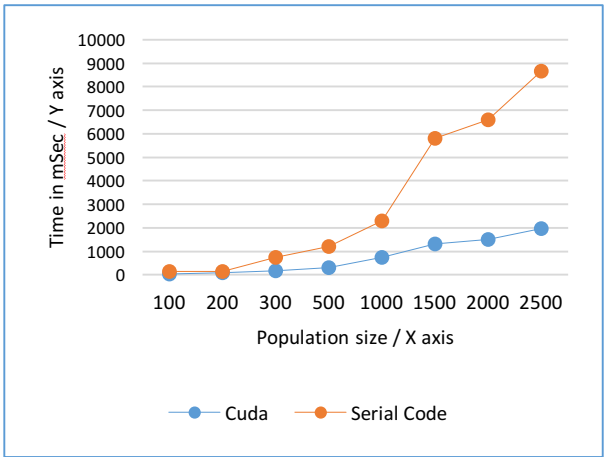
The threads and blocks have global access to the declared data arrays and the clashes output array. Once a parallel calculation is complete and each thread assigns its values to the corresponding index in the clashes array then we reduce the array to a sum which gives us the total clashes of an individual in the population. A possible future scope would be to extend to the potential of the GPU and calculate the fitness of all the individuals of a population simultaneously.

Some points of observation:

- Memory allocation consumed more than 50 percent of the execution time, this happened because the fitness and initialization function are simple and the cost of building the memory for the 7 arrays that re passed to the thread caused an overhead, hence when we separated out the allocation then we get a pure execution time which was significantly faster.
- The first phase initialization participated very less in optimization since it was executing only once, for larger problem set we can even ignore the parallelism of the initialize function.
- The fitness function was a major contributor this function is repeatedly called in the crossover and the evaluation stage, further called for every population generation for all individuals of a population. Hence our results showed significant improvement in performance.
- As an extension to this proposed solution one can even parallelize the mutation stage as well as the crossover stage to generate a completely parallel solution
- Implementation of java objects with Cuda would give us a better comparative analysis between Cuda and Hadoop or spark performance.
- However, an implementation in java will ease the testing process but may adulterate the result with its own slow nature of execution unlike C.

Results:

Following are some of the test results obtained. We compare the average of the results of a set of problem generation size v/s execution time results for the parallel segment of the code and clearly the execution has a positive speedup in parallel.



TTG – MPI

Problem Description:

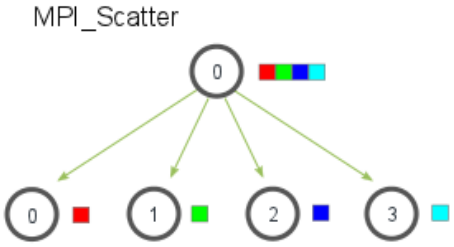
The genetic algorithm parallelization which is being performed using MPI focused on primary concept that was evaluate Population which was one of the stages in the whole implementation of genetic algorithm. There exist disparate functions inside evaluate Population that is calculate fitness. Let's elucidate the parallelization being performed.

Existing Solution:

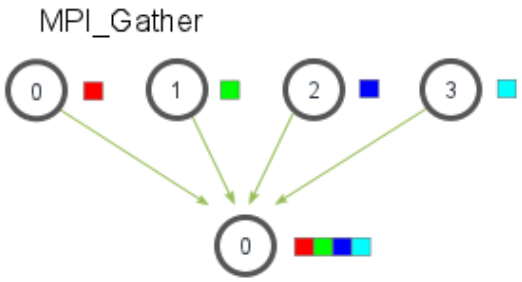
The implementation of this algorithm with MPI is not being performed. There are several complexity which exist, line in MPI parallel program execution characteristic is to perform the same procedure at the same time on each machine , therefore in order to realize the parallel processing and communications between the nodes, that is, dynamic logic of resources must be clear.[5]

Approach:

So the approach which was followed was to understand the key methods or functions working, first approach to convert the serial code which was in java to C, that had many roadblocks as C is not object oriented but Java uses object oriented concepts. Next was to implement MPJ i.e. MPI library for java. The backend library on which it was based on was MPICH. Starting phase was to test samples on Jarvis on various nodes having disparate number of processes to get hands on with MPJ. The Primary concept used was Scatter and Gather. In scatter the master node or the root node sends data to all processes. [7]



Then every node performs computation which is defined and gather is called in which every other node except master sends data back to master node.



Implementation:
Serial One:
The primary method which we are focusing in MPI is evaluate population function which is as follows:

```
public void evalPopulation(Population population, Timetable timetable) {
    double populationFitness = 0;

    // Loop over population evaluating individuals and summing populatic
    // fitness
    for (Individual individual : population.getIndividuals()) {
        populationFitness += this.calcFitness(individual, timetable);
    }

    population.setPopulationFitness(populationFitness);
}
```

Let's elucidate the evalPopulation method. In this method, the fitness calculation is being performed which is one of the vital steps in the algorithm. The calcFitness is called and every individuals fitness is being calculated and added in an array. This array is primarily used for out parallelization purpose because it holds all the fitness value. The class genetic algorithm is where the method resides, it is called when all the random population has been generated. There are series of loop inside calFitness function which takes individual and timetable as the parameters in which all the clashes are computed.
Parallel One (Implemented by team member):The parallel one version consists of optimized code which is done in MPJ. The difference between the scatter and gather which is there in C is the removal pointer concept, and introduction of buffer object and offset the difference can be seen in the images which are being shown.The MPI gather function in C is as follows:

```
int MPI_Gather(
  void *sendbuf,
  int sendcnt,
  MPI_Datatype sendtype,
  void *recvbuf,
  int recvcnt,
  MPI_Datatype recvtype,
  int root,
  MPI_Comm comm
);
```

The MPJ gather function which is of JAVA is as follows[6]:

```
void Intracomm.Gather(Object sendbuf, int sendoffset,
                      int sendcount, Datatype sendtype,
                      Object recvbuf, int recvoffset,
                      int recvcount, Datatype recvtype, int root)
```

Its reverse is of Scatter, it is very important to understand the Scatter and gather and distinguish between the two. In C the only thing which does the function of first two parameters mention in JAVA is pointers buffer. In Java we need to set the offset for sending as well as receiving.

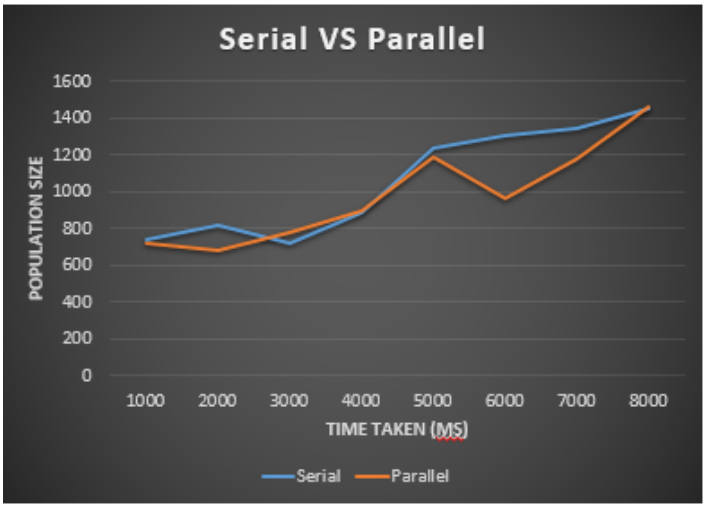
The parallel code snippet is provided. In that popFitnessCopy is the buffer which contains the whole chromosomes or fitness of individual, offset is set as start index which calculates from which point the start index is set. Similarly, the endIndex. The count calculates size of population which is provided to each process and then the datatype. The scatter primarily divides the chromosomes or fitness values of individual into how much the count values are provided. And the calcFUnction2 is called which does the whole calculation of computation by summing all the fitness value each process adds value and in gather each processor sends it to the master node. And master node contains the result of computation performed. Before scatter the Barrier function is used for synchronization and after gather also. The following code snippet is the parallel version:

```
MPI.COMM_WORLD.Scatter(popFitnessCopy,
        stIndex,
        chunkSize,
        MPI.DOUBLE,
        partialArray ,
        endIndex ,
        chunkSize,
        MPI.DOUBLE, 0);
double local_sum[] = new double[1];
local_sum[0]=calcFunction2(partialArray);

MPI.COMM_WORLD.Gather(local_sum,
        0,
        1,
        MPI.DOUBLE,
        global_sum,
        rank, 1,
        MPI.DOUBLE,0);
```

The testing was done on 4 processes in which 3 were the slave nodes and one master node was present. The optimization was not that much as expected.

Results:



The above graph explains about serial vs parallel code time taken varying the number of population size. We compare the average of the results of a set of problem generation size v/s execution time results for the parallel segment of the code and clearly the execution doesn't have speedup comparable in parallel which were supposed to be. The higher number of population will give the speedup. The Future scope will be to include various Advance MPI function to optimize.

Feedback from the oral presentation

1. The presentation includes the graph, which should take an average of test cases. The reason is some cases we have spikes in the chart where Serial algorithm got lucky and performed well but not always. To avoid unusual spikes in the graph, you must take average of those test cases.

2. It would be wise if you used billions size as population size in this. The population size is little low, though 7000 you have taken in the real situation, it would regard millions size.

3. Comments on CUDA optimization by utilization of complete GPU.

4.MPJ is not the way because it MPI is way faster than MPJ.

5.We received overall positive feedback for implementation However it would be great if we our analysis includes very big problem size. Taking 8 million as population size instead of 8000 as maximum population size.

Conclusion

So far we have seen the performance of Apache spark, Cuda and MPI. In an overall perspective, it is difficult to compare the mentioned three platforms as they are best fit for diverse application types. Spark is usually preferred when a big data process is handled, we can scale the timetable application to a larger dataset to be distributed among the nodes. But it does not work in the traditional fashion of reading a gigantic dataset primarily because the program dynamically generates this dataset itself and used by the program in the very next iteration, this phase as discussed is called the population generation. We consider this as a part of the algorithm because of its dynamic and random nature. This is one way to increase the size of the population but then we cannot increase size of each chromosome defining each class as there are pragmatically limited number of classes per week. Adding further the crossover and mutation comprises computing the fitness of a defined set of population which is distributed for computation among nodes. Future scope for this application in the spark platform can be to optimize the equal distribution of classes, eventually by keeping the data static we can span it to over a million or billion thereby categorizing its suitability towards to big data application, adding further [add what can be done as future scope of this project in spark that you have not implemented.
The problem of local optima and wide fitness calculation is a limitation of genetic algorithms. The Cuda platform is a exemplary fit as seen from the results obtained for the current problem population size range which is large but not in millions or billions which suits the big data architecture. We can compare the results between the Spark and Cuda implementation but various factors are varying in each test case, for instance we can consider the population size variable, a solution found in 50 generations with a recurring population of 100 will not necessarily produce the same best solution in 50 generations with a population size of 100. Hence, we can compare the fitness calculation in terms of speedup gained in both the platforms. Programming in Cuda is efficient because of its low-level characteristics i.e. C programming this is noticed in comparison with other languages like java the time spent for the parallel fitness function is less than is in C. This is a contributing factor for the delay in spark. As an improvement for Cuda we have 1024 blocks and each block with 1024 threads among which we use a defined number of blocks for parallelism which is very less than the available capacity, we can increase this distribution to more number of threads and obtain two times increase in performance at least. As discussed above MPICH libraries were used and the optimization of algorithm was done in MPJ i.e. java libraries of MPI. Primarily MPI optimizations are done in C language because it is much faster as compared to what is written in Java. The other problem faced was that in C to write the genetic algorithm is complex and hence due to that the debugging is tough in C. Due to that the switchover was made to Java. The comparison with apache Spark cannot be done because the Big Data application primarily focuses on static data, other than that with CUDA the comparison cannot be done because CUDA make use of GPU, threads and MPI make use of processors and CPU. The problems which occurred was about the speedup, for small population. The speedup can be compared, in which MPI is better than Apache Spark because the order of magnitude in terms of processing speed and provides more consistent performance. Now the primary concept which was used in this project was Scatter and Gather, the implementation can be extended by using Advanced MPI concepts.

References:

[1] https://www.udacity.com/course/intro-to-parallel-programming--cs344

[2]https://en.wikipedia.org/wiki/Genetic_algorithm

[3] **Jacobson**, Lee, **Kanber**, Burak Genetic Algorithms in Java Basics. 2015. Print.

[4] https://github.com/ajayramesh23/ubiquitous-eureka

[5]http://www.ipcsit.com/vol43/029-ICETC2012-T0268.pdf

[6]https://www.open-mpi.org/papers/mpi-java-spec/mpiJava-spec.pdf

[7]http://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/

[8] Yoshiki Sugawara, Nobukazu Takai , Masato Kato , Hiroaki Seki,Kento Suzuki ,Haruo Kobayashi - "Automatic design of doubly-terminated RC polyphase filters by using distributed genetic algorithm"- ASIC (ASICON), 2015 IEEE 11th International Conference on

[9] Alberto Colorni , Marco Dorigo , Vittorio Maniezzo - "A Genetic Algorithm To Solve The Timetable Problem (1993)"

[10] Mingjie Wang ,Jesse S. Jin, Gelin Wu,Wei Tong, Yu Peng - DISTRIBUTED GENETIC ALGORITHM BASED ON RESTFUL FRAMEWORK

[11] Mauro Castelli , Luca Manzoni , Leonardo Vanneschi - "The effect of selection from old populations in genetic algorithms"

[12] Francisco Rojas, Federico Meza- "A Parallel Distributed Genetic Algorithm for the Prize Collecting Steiner Tree Problem"

[13] List of genetic algorithm applications

[14] Genetic Algorithms in Java Basics By Lee Jacobson , Burak Kanber

[15] Genetic Algorithms tutorial

[16] Cuda - http://www.nvidia.com/object/cuda_home_new.html

[17] MPICH - https://www.mpich.org/

[18] Apache Spark - http://spark.apache.org/

[19] Apache spark use cases - https://www.qubole.com/blog/big-data/apache-spark-use-cases/

[20] gpu-application - http://www.nvidia.com/object/gpu-applications.html

[21] A Genetic Algorithm for Resource-Constrained Scheduling - MIT