Parallelization of Simulated Annealing Algorithm

Hari Priya
School of Science, Technology Engineering & Mathematics
University of Washington
Bothell, United States of America
hdhanu@uw.edu

Rithi Afra
School of Science, Technology Engineering & Mathematics
University of Washington
Bothell, United States of America
rithij@uw.edu

Shengao Sun
School of Science, Technology Engineering & Mathematics
University of Washington
Bothell, United States of America
ckpaul@uw.edu

I. Application Overview

The project is to parallelize the simulated annealing algorithm. It is a metaheuristic algorithm that can be used to solve the Travelling Salesman Problem. For problems like TSP, addition of new cities will result in a combinatorial explosion in the number of routes to be traversed. This will result in a considerable increase in the amount of time it takes to find the optimal solution. This is when metaheuristic algorithms come into action. The Simulated Annealing algorithm resembles the annealing process that is used in metallurgy. In metallurgy, the atoms move fast when the system is in high temperature. When the temperature is reduced, the kinetic energy decreases and the system gets more easy to work with. Similarly in Simulated Annealing algorithm, the process starts with a high randomness and gradually lowers its temperature until it reaches an optimal solution. Simulated Annealing algorithm has the capacity to escape from local minima and converge to global maximum.

II. Sequential Algorithm Explanation

The algorithm starts by reading the cities from a file using the readCities() method. The cities are stored as a list of integer arrays, where each array represents the coordinates of a city.

Next, the algorithm initializes the temperature (temp) to a predefined initial temperature `(INITIAL_TEMP)`. It also initializes the current solution (currentSolution) by generating an initial solution using the generateInitialSolution() method. This method creates a list of integers from 0 to n-1, where n is the number of cities, and shuffles the list randomly. This represents an initial tour.

The best solution found so far is initially set to the current solution (bestSolution) along with its corresponding cost (bestCost).

The simulated annealing algorithm iterates until the temperature reaches a minimum value or a stopping criterion is met. Within each iteration, a neighboring solution (newSolution) is generated by swapping two random cities in the current solution using the generateNeighborSolution() method.

The cost of the current solution (currentCost) and the cost of the new solution (newCost) are calculated using the calculateCost() method. This method computes the total distance traveled in the tour by summing the Euclidean distances between consecutive cities.

The acceptance probability of the new solution is computed using the acceptanceProbability() method. If the new solution has a better cost, the acceptance probability is set to 1.0, indicating that the new solution should be accepted. Otherwise, the acceptance probability is calculated using the exponential function based on the difference between the current cost and the new cost, divided by the current temperature.

If the acceptance probability is greater than a random number between 0 and 1, the new solution is accepted as the current solution. Additionally, if the new solution has a lower cost than the best solution found so far, it becomes the new best solution.

After each iteration, the temperature is reduced by multiplying it by a cooling rate `(COOLING_RATE)`. This cooling rate determines the rate at which the temperature decreases. The algorithm continues iterating until the temperature reaches a minimum value, at which point the best solution and its cost are printed as the output.

In summary, the provided code implements the simulated annealing algorithm to solve the TSP. It uses an initial solution, generates neighboring solutions by swapping cities, calculates the cost of solutions, computes acceptance probabilities, and updates the current and best solutions based on the acceptance criteria. The algorithm gradually reduces the temperature while exploring the solution space to find an optimal or near-optimal solution to the TSP.

III. PARALLELIZATION TECHNIQUES

A. MPI JAVA

The MPI algorithm synchronizes the cost and optimal solution for every process. From the bestRoot, the optimal solution gets propagated to every other process. All the processes will therefore receive the best solution. After then, the existing

solution will be modified accordingly. A comprehensive description of the code can be found in [1]:

```
if (rank == 0) {
    for (int i = 1; i < numProcesses; i++) {
        MPI.COMM_WORLD.Recv(globalBestCostBuff, 0, 1, MPI.DOUBLE, i, 0);
        if (globalBestCostBuff[0] < globalBestCost) {
            bestRoot = i;
            globalBestCost = globalBestCostBuff[0];
        }
    }
    globalBestCostBuff[0] = bestRoot;
    for (int i = 1; i < numProcesses; i++) {
        MPI.COMM_WORLD.Send(globalBestCostBuff, 0, 1, MPI.DOUBLE, i, 0);
    }
} else {
    MPI.COMM_WORLD.Send(globalBestCostBuff, 0, 1, MPI.DOUBLE, 0, 0);
    MPI.COMM_WORLD.Recv(globalBestCostBuff, 0, 1, MPI.DOUBLE, 0, 0);
}
bestRoot = (int) globalBestCostBuff[0];
  System.out.println("<<< rank[" + rank + "] best root =" + bestRoot);
// update bestSolution from bestRoot
int[] tempBestSolution = new int[currentSolution.size()];
for (int i = 0; i < currentSolution.size(); i++) {
    tempBestSolution[i] = currentSolution.get(i);
}
MPI.COMM_WORLD.Bcast(tempBestSolution, 0, tempBestSolution.length, MPI.INT, bestRoot);
for (int i = 0; i < currentSolution.size(); i++) {
    currentSolution.set(i, tempBestSolution[i]);
}
if (rank == 0 && bestCost > calculateCost(currentSolution)) {
    bestSolution = currentSolution;
    System.out.println(">>> New best solution cost=" + bestCost + "\t tmp=" + temp);
}
```

Fig. 1. CODE SNIPPET OF MPI

The current process's unique identifier, is stored in the 'rank' variable, and the total number of processes in the MPI environment is stored in 'numProcesses'. The process with the lowest cost ('bestRoot') among all processes is identified in this section, which also synchronizes the best cost. Process 0 finds the process with the lowest cost by taking the best cost from all the other processes. The best cost is sent and received between processes via the `MPI.COMM_WORLD.Send()` and `MPI.COMM_WORLD.Recv()` functions.

B. MAP REDUCE JAVA

A Map Reduce job is configured in the main function along with input and output paths, mapper and reducer classes. The TSPMapper class generates a random initial tour, performs the Simulated Annealing Algorithm to find optimal tour and it outputs the best tour itinerary and respective distance. Here, we are using CalculateTourDistance to calculate total distance. [3]:

```
@Override
public void map(Object key, Text value, Context context) throws IOException, InterruptedException {
    long seed = Long.parseLong(value.toString());
    Random random = new Random(seed);
    System.out.println("Random seed=" + seed);
    // Generate a random initial tour
    int[] currentTour = generateInitialTour(random);
    // Perform Simulated Annealing
    double currentDistance = calculateTourDistance(currentTour);
    double temperature = INITIAL_TEMPERATURE;
    while (temperature > 1) {
        for (int i = 0; i < NUM_NEIGHBORS; i++) {
            int[] newTour = generateNeighborTour(currentTour, random);
            double newDistance = calculateTourDistance(newTour);
            if (acceptMove(currentDistance, newDistance, temperature, random)) {
                currentTour = newTour;
                currentDistance = newDistance;
            }
        }
        temperature *= 1 - COOLING_RATE;
    }
    // Emit the best tour and its distance
    valueOut.set(currentDistance + "#" + Arrays.toString(currentTour));
    context.write(keyOut, valueOut);
}
```

Fig. 2. CODE SNIPPET OF MAPPER

The TSPReducer class, receives the best tour itinerary and distance form multiple mappers, selects the best one using acceptMove function and emits final output.

```
public static class TSPReducer extends Reducer<Text, Text, Text, Text> {
    private Text valueOut = new Text();

    @Override
    public void reduce(Text key, Iterable<Text> values, Context context)
            throws IOException, InterruptedException {
        // Find the best tour and its distance
        double bestDistance = Double.MAX_VALUE;
        String bestTourTuple = "";
        for (Text value : values) {
            String tourTuple = value.toString();
            String[] tourArgs = tourTuple.split("#");
            double distance = Double.parseDouble(tourArgs[0]);
            if (distance < bestDistance) {
                bestDistance = distance;
                bestTourTuple = tourTuple;
            }
        }
        // Emit the best tour and its distance
        valueOut.set(bestTourTuple);
        context.write(key, valueOut);
    }
}
```

Fig. 3. CODE SNIPPET OF REDUCE

C. SPARK JAVA

In the case of Spark, a function called readCities is used to read city coordinates and stores them in a list of arrays. Spark configuration is setup and JavaSparkContext is initialized in main method. [4]

```
public static void main(String[] args) {
    // Create a Spark Context
    SparkConf sparkConf = new SparkConf().setAppName("SimulatedAnnealingSpark")
            .setMaster("local[2]").set("spark.executor.memory", "2g");
    JavaSparkContext sc = new JavaSparkContext(sparkConf);
    // log level
    sc.setLogLevel("WARN");
    SimulatedAnnealingSpark sa = new SimulatedAnnealingSpark();
    sa.run(sc);
}
```

Fig. 4. CODE SNIPPET OF SPARK

Two solution list named currentSolution and bestSolution is created and initial solution generated is also stored in it. The cost of bestSolution is calculated using calculateCost method. For every iteration, neighbouring solutions are generated and these neighbors are divided into partitions and parallelized in Spark cluster.

Spark map calculates the cost of each neighbor solution and its collected back using collect. Acceptance probability is calculated and if its better solution and is accepted, it becomes the best solution. Then after the temperature decreases below the threshold, the best cost and best solution are printed and Spark context is closed.[5]

D. MASS JAVA

In this work, the MASS (Migratable Agents for Simulation Systems) framewor is used where it allows for the execution

```
while (temp > 1) {
    List<List<Integer>> neighbors = generateNeighbors(currentSolution, rand);
    JavaRDD<List<Integer>> neighborsRDD = sc.parallelize(neighbors, NUM_NEIGHBORS);
    List<Tuple2<List<Integer>, Double>> solutionsAndCosts = neighborsRDD
            .map(solution -> new Tuple2<>(solution, calculateCost(solution, cities)))
            .collect();
    for (Tuple2<List<Integer>, Double> solutionAndCost : solutionsAndCosts) {
        List<Integer> newSolution = solutionAndCost._1;
        double newCost = solutionAndCost._2;
        if (acceptanceProbability(calculateCost(currentSolution, cities), newCost, temp) > Math.random()) {
            currentSolution = new ArrayList<>(newSolution);
        }
        if (newCost < bestCost) {
            bestSolution = new ArrayList<>(newSolution);
            bestCost = newCost;
            System.out.println(">>> New solution cost: " + newCost + " solution: " + newSolution);
        }
    }
    temp *= 1 - COOLING_RATE;
    if (temp < logTemp * 0.8) {
        logTemp = temp;
        System.out.println(">>> temperature = " + temp);
    }
```

Fig. 5. CODE SNIPPET OF SPARK run()

of simulations across multiple nodes. The code consists of two Java classes: SAMass and TSPPlace.

In the SAMass class, main methods initializes the MASS framework by creating Places object. The Simulated Annealing algorithm is executed in a loop till the temperature reached a threshold. callAll method invokes the `SET_TEMP_AND_GENERATE_SOLUTION` method on all places in parallel.Best cost and solution is continously updated and displayed at the end of iteration.[6]:

```
MASS.init();
// start time
long startTime = System.currentTimeMillis();
// Create places, init with cities and random seed in Object[]
Places places = new Places(1, TSPPlace.class.getName(), cities, NUM_PLACES_X, NUM_PLACES_Y);
int placeNum = places.getPlacesSize();
System.out.println("places=" + placeNum);
// Simulated annealing algorithm
double temp = INIT_TEMPERATURE;
double logTemp = temp;
double bestCost = Double.MAX_VALUE;
List<Integer> bestSolution = null;
while (temp > 1) {
    //get all final solution
    Object[] placeCallAllObjs = new Object[placeNum];
    Object placeArg = new Object[]{temp, bestSolution};
    for (int i = 0; i < placeNum; i++) {
        placeCallAllObjs[i] = placeArg;
    }
    Object[] solutions = (Object[]) places.callAll(TSPPlace.SET_TEMP_AND_GENERATE_SOLUTION, placeCallAllObjs);
    for (Object s : solutions) {
        System.out.println(s.getClass().getName());
        List<Integer> solution = (List<Integer>) s;
        double cost = TSPPlace.calculateCost(solution, cities);
        if (cost < bestCost) {
            bestSolution = solution;
            bestCost = cost;
            System.out.println(">>> New solution cost: " + bestCost + " solution: " + bestSolution);
        }
    }
```

Fig. 6. CODE SNIPPET OF CallAll

In the TSPPlace class, it contains an indivudual place needed for a part of simulation. initializeSolution method generates initial solution while generateNeighbors method genertaes set of neighbor solutions by swapping. Similarly, acceptance probability is used to decide whether to accept a solution or not. The callMethod takes care of the `SET_TEMP_AND_GENERATE_SOLUTION` method from SAMass class.The solution is updated continously and returned to SAMass class. Multiple places (TSPPlace instances) execute in parallel, generating and evaluating neighbor solutions.[7]:

```
public Object callMethod(int methodId, Object args) {
    System.out.println("callMethod id=" + methodId + ",args=" + args);
  switch (methodId) {
    case SET_TEMP_AND_GENERATE_SOLUTION:
      // temperature
      double temp = (double) ((Object[]) args)[0];
      List<Integer> solution = (List<Integer>) ((Object[]) args)[1];
      if (solution != null) {
        currentSolution = solution;
      }
      List<List<Integer>> newSolutions = generateNeighbors(currentSolution, rand);
      for (List<Integer> newSolution : newSolutions) {
        double newCost = calculateCost(newSolution, cities);
        double currentCost = calculateCost(currentSolution, cities);
        if (acceptanceProbability(currentCost, newCost, temp) > rand.nextDouble()) {
          currentSolution = newSolution;
        }
      }
      return currentSolution;
    default:
      return null;
  }
}
```

Fig. 7. CODE SNIPPET OF PLACES

IV. PERFORMANCE ANALYSIS

A. MPI

The following table shows the performance analysis of MPI code on different cities. The analysis was done on 5, 36 and 100 cities. The highest performance improvement was achieved when the program was executed under 4 nodes with 100 cities. The lowest performance improvement was under 3 nodes with 5 cities.

	5 Cities		36 Cities		100 Cities	
#nodes	Elapsed Time(ms)	Performance Improvement	Elapsed Time(ms)	Performance Improvement	Elapsed Time(ms)	Performance Improvement
1	915	NA	1238	NA	1991	NA
2	931	0.98	1151	1.07	1867	1.06
3	763	1.19	1160	1.06	2227	0.89
4	822	1.11	1209	1.02	2046	0.97

Fig. 8. PERFORMANCE OF MPI

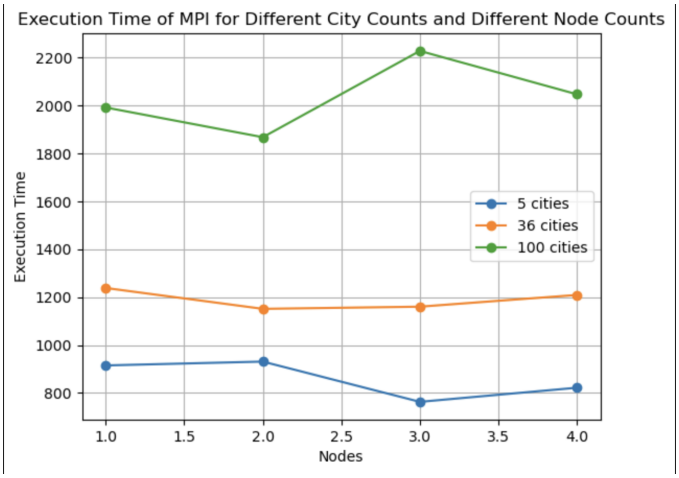

Fig. 9. OUTPUT OF MPI

B. MAP REDUCE

The following table shows the performance analysis of MapReduce code on different cities. The analysis was done on 5, 36 and 100 cities. The highest performance improvement was achieved when the program was executed under 2 nodes with 100 cities. The lowest performance improvement was under 3 nodes with 5 cities.

	5 Cities		36 Cities		100 Cities	
#nodes	Elapsed Time(ms)	Performance Improvement	Elapsed Time(ms)	Performance Improvement	Elapsed Time(ms)	Performance Improvement
1	1662	NA	2628	NA	3672	NA
2	1625	1.02	2709	0.97	3875	0.94
3	1526	1.08	2616	1.00	3624	1.01
4	1572	1.05	2652	0.99	3730	0.98

Fig. 10. PERFORMANCE OF MAPREDUCE

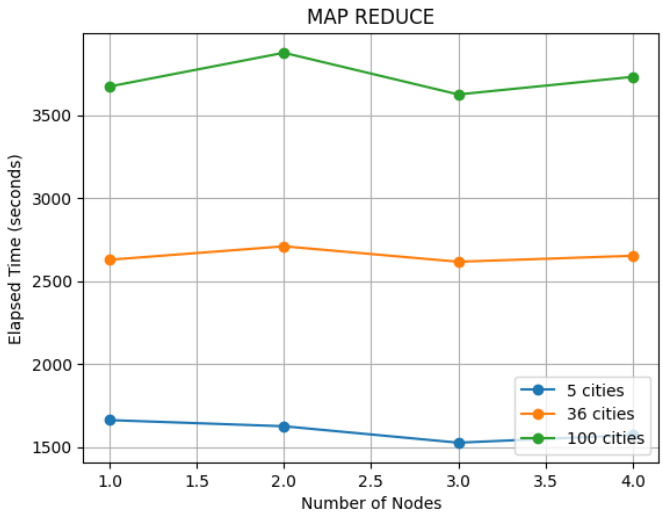

Fig. 11. OUTPUT OF MAP REDUCE

C. SPARK

The following table shows the performance analysis of Spark code on different cities. The analysis was done on 5, 36 and 100 cities. The highest performance improvement was achieved when the program was executed under 4 nodes with 100 cities. The lowest performance improvement was under 2 nodes with 5 cities and 4 nodes with 36 cities. For some reason, SPARK did not perform quite well. There are multiple reasons that might contribute to this fact. Data shuffling is a primary factor that increases the amount of execution time. Since Spark involves complex operations this could also contribute to the increased execution time. If the code can be optimized a bit more, the exeecutiontime can be improved a bit.

	5 Cities		36 Cities		100 Cities	
#nodes	Elapsed Time(ms)	Performance Improvement	Elapsed Time(ms)	Performance Improvement	Elapsed Time(ms)	Performance Improvement
1	93321	NA	99522	NA	103623	NA
2	89137	1.04	99102	1.00	107798	0.94
3	101785	0.91	95795	1.03	109690	0.96
4	94838	0.98	95548	1.04	115312	0.89

Fig. 12. PERFORMANCE OF SPARK

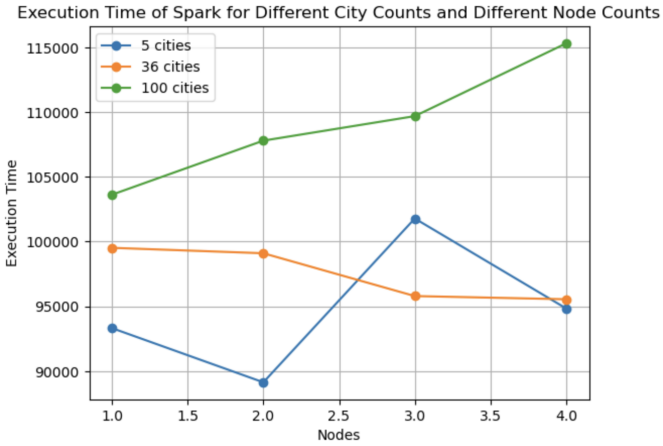

Fig. 13. OUTPUT OF SPARK

D. MASS

The following table shows the performance analysis of Spark code on different cities. The analysis was done on 5, 36 and 100 cities. The highest performance improvement was achieved when the program was executed under 4 nodes with 100 cities. The lowest performance improvement was under 2 nodes with 5 cities and 4 nodes with 36 cities. MASS did not perform as expected. We expected MASS to perform bit faster. But it could not beat the results of MPI and MapReduce. MASS code invloves complex algorithms. This could heavily impact the execution time. The overhead of managing agents and communication between agents increase the execution time to a greater extent.

	5 Cities		36 Cities		100 Cities	
#nodes	Elapsed Time(ms)	Performance Improvement	Elapsed Time(ms)	Performance Improvement	Elapsed Time(ms)	Performance Improvement
1	8563	NA	53267	NA	109238	NA
2	4701	1.82	45286	1.18	83427	1.31
3	3569	2.40	39235	1.36	54570	2.01
4	3153	2.72	20390	2.61	51839	2.12

Fig. 14. PERFORMANCE OF MASS

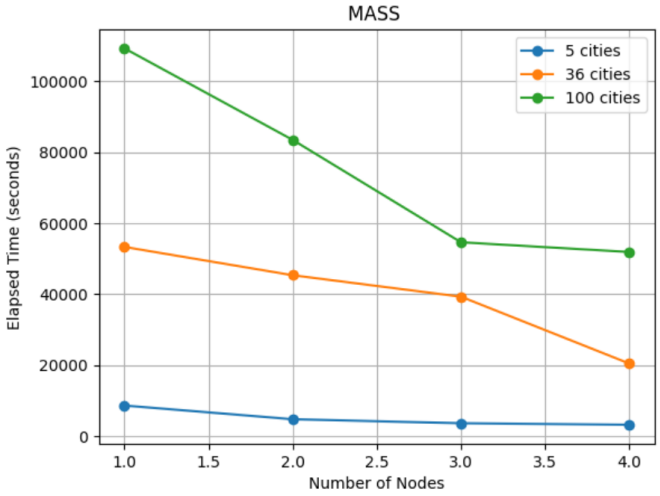

Fig. 15. OUTPUT OF MASS

The below plot shows the comparison between all the frameworks used with the best Elapsed time taken for 100 cities. This shows that MASS outperforms Spark while MPI and Map Reduce still performs better than MASS, with MPI being the best.

	Best execution (100 cities)s
MPI	1867
MAP REDUCE	3624
SPARK	109690
MASS	51839

Fig. 16. BEST ELAPSED TIME

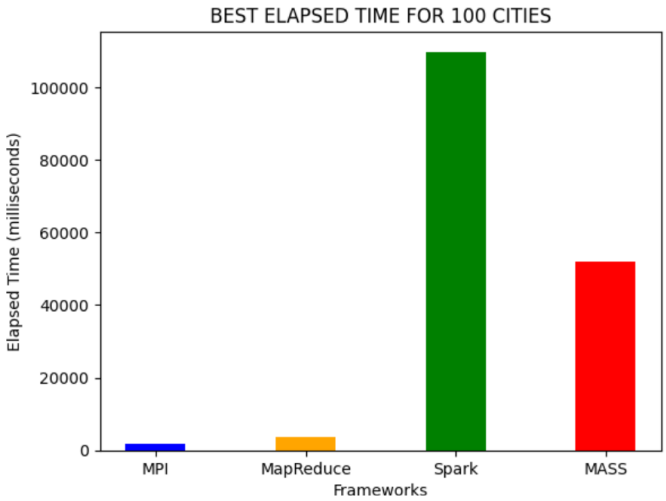

Fig. 17. COMPARISON OF BEST ELAPSED TIME

V. PROGRAMMABILITY

MPI: All computation happens on a single node. The performance is limited by the resources of that single machine. The workload can be split between two nodes, potentially nearly improving performance (assuming the problem and network overhead allow for efficient parallelization). However, managing communication overhead, load balancing, and ensuring proper synchronization between processes were the places we had to be careful.

MapReduce: Initially, it was executed limited to a single machine. MapReduce can split the work across the nodes, but network I/O and the cost of shuffling data between map and reduce stages may cause overhead. It was challenging to express the SA algorithm in a map and reduce paradigm. The mapper generates the initial solution , calculates cost and distance and decides if it needs to be kept or not, then the reducer takes all the solution and provides the best solution and cost. Any inefficiencies are likely due to overhead from the MapReduce model, which is not fully utilized on a single node. However, better performance was seen after increasing slaves.

	LOC	Cyclomatic Complexity
SEQUENTIAL	97	10
MPI	161	20
MAP REDUCE	202	9
SPARK	139	13
MASS	190	16

Fig. 18. PROGRAMMABILITY

SPARK: Spark can run in local mode on a single machine, but like MapReduce, there may be overhead due to the Spark model. As it provides a resilient distributed dataset (RDD) abstraction for parallel data processing, Spark may have a performance advantage over MapReduce due to its in-memory computation capability. However, shuffling costs still exist. However, few challenges were faced during efficient handling of iterations and minimizing data shuffling.

MASS: MASS would also be limited to the resources of the single machine, with potential overhead from the parallel computing model. However, MASS potentially doubled the performance by splitting the work across nodes, similar to MPI by giving a performance better than Spark and Map Reduce. Adapting the SA algorithm to the agent-based model and handling communication properly were few challenges we faced. Also, we were facing a delay in executing the MASS program as several nodes were busy and execution halted several times.