FINAL PROJECT - HPC CLASS 18/19

Danilo Franco - s3809721

PROBLEM DEFINITION

The aim of this project is to provide a C++ implementation for the Traveling Salesman Problem (minimum cost for a strongly connected entire-graph traversing) that follows a genetic algorithm optimisation approach and, at the same time, makes use of both *OpenMP* (for parallel execution in a multicore architecture) and *MPI* (for message exchanging in a cluster of several nodes) libraries.

MAIN STRUCTURE

PREPROCESSOR DEFINES

- NUMTHREADS: how many processing elements are due to work on each parallel section;
- AVGELEMS: number of elements from which the average for early-stopping is computed;
- TRANSFERRATE: iterations rate on which the nodes are due to exchange their best nodes permutations (if expected);
- Several prints related, for evaluating the program correctness or values storing.

MAIN

The main method simply:

i. read the arguments given by terminal and check whether they are correct (in the code several comments on this regard are present);
ii. initialise the MPI connection (if needed);
iii. Redirect the standard output towards the appropriate directory according to the current execution approach;
iv. Launch the entry method for the genetic tsp.

ENTRY METHOD
[*genetic_tsp(...)*]

Type	*Datatype*	*Name*	*Description*
Input Arg	int	me	Index of the current executing node in the cluster - optional (only for MPI implementation)
"	int	numInstances	Nodes amount currently working on finding the solution - optional (only for MPI implementation)
"	Int	numThreads	Number of processing elements that are due to work on each parallel section
"	int*	cost_matrix	Pointer to memory that contains the symmetric node-traveling cost matrix
"	int	numNodes	Number of nodes in the strongly connected graph

Type	*Datatype*	*Name*	*Description*
"	int	population	Number of nodes permutations (possible solutions) found at each round
"	double	top	Percentage [0-1] of population elements that are going to generate new permutations
"	int	maxIt	Number of maximum iteration rounds
"	double	mutatProb	Probability [0-1] of mutation occurrence in the newly generated permutation
"	int	earlyStopRounds	Number of latest iterations from which the average of best AVGELEMS must be computed in order to establish convergence
"	double	earlyStopParam	Comparison parameter (standard deviation) for early stopping
Output Return	int*		Pointer to a vector that contains the found minimal-cost solution, its cost, a boolean values (integer) stating whether the process did converge or not and the number of effective iterations

This method is the entry point for finding and returning the solution of the TSP.
Before initiating the standard pipeline, it performs the arrays memory allocation, useful numeric conversion (e.g. conversion from probabilities and percentages to integers), a sequential initialisation of the permutation matrix and a random shuffle of it with, of course, the relative ranking, and then, at the end, the copy of the found solution and memory deallocation.

PIPELINE

NEW PERMUTATIONS GENERATION
[*generate(...)*]

Type	*Datatype*	*Name*	*Description*
Input Arg	int*	generation	Pointer to the permutation matrix (population*nodes) for the current iteration
"	int	population	Number of nodes permutation (possible solution) found at each round
"	int	bestNum	Number of best elements (parents) that will produce the next generation
"	int	numNodes	Number of nodes in the strongly connected graph
"	int	probPercentile	Probability [0-100] of mutation occurence in the newly generated population element
"	int	numThreads	Number of processing elements that are due to work on each parallel section
Output Return	void		

```
    int i,parent1,parent2,son;

    // fill from bestnum until all population is reached
#pragma omp parallel for num_threads(numThreads) private(parent1,parent2,son,i) schedule(static)
    for(i=0; i<population-bestNum; ++i){
        if (i<bestNum) // each best must generate at least one son
            parent1 = i;
        else
            parent1 = rand()%bestNum;

        do {    // two different parents
            parent2 = rand()%bestNum;
        } while(parent2==i);

        son = (bestNum+i)*numNodes;

        crossover_firstHalf_withMutation(generation, parent1, parent2, son, numNodes, probCentile);
    }
```

This method simply find the correct indexes that need to be passed to the inner function; the process is parallelised over the matrix filling (starting from the first row of no-survivor to the end), fixing the first parent (sequential iteration of all of the survivors first, then random between any of them) and, then, randomly choosing the second one (that of course must be different from the previous).

```
void crossover_firstHalf_withMutation(int *generation, int parent1, int parent2,
                                      int son, int numNodes, int probCentile){
    set<int> nodes;
    int j,k,half,elem,swap1,swap2;

    half = floor(numNodes/2);

    // take first half from parent1
    for(j=0; j<half; ++j){
        elem = generation[parent1*numNodes+j];
        generation[son+j] = elem;
        nodes.insert(elem);
    }
```

```
    // add the remaining elements from parent2
    for(k=0; k<numNodes; ++k){
        elem = generation[parent2*numNodes+k];
        if(nodes.find(elem)==nodes.end()){
            generation[son+j] = elem;
            ++j;
        }
    }
```

At this point, in order to generate a new valid permutation having two parent indices, the algorithm takes the first half from one parent and then the remaining ones from the second, maintaining the existent order (the procedure makes use of the *Set* C++ library in order to maintain node unicity within a single permutation - it adds all the nodes from the first parent into the set structure, thus if a node from the second parent is not already present in the set, then it is a valid one).

```
    // MUTATION
    if((rand()%100+1)<=probCentile){
        swap1=rand()%numNodes;
        do {
            swap2=rand()%numNodes;
        } while(swap2==swap1);

        elem = generation[son+swap1];
        generation[son+swap1] = generation[son+swap2];
        generation[son+swap2] = elem;
    }
    return;
}
```

Having a valid permutation, in order to get out from local minima the optimisation might end up in being locked into, the algorithm will try to swap randomly two different nodes in the current path (note that it will happen with probCentile frequency, in fact it represents an hyper-parameter that need its proper validation).

RANKING & REARRANGING
[*rank_generation(...)*]

Type	*Datatype*	*Name*	*Description*
Input Arg	int*	generation_cost	Pointer to the total permutation cost array
"	int* &	generation	Pointer to the permutation matrix (population*nodes) for the current iteration
"	int* &	generation_copy	Pointer to the copied permutation matrix
"	int*	cost_matrix	Pointer to memory that contains the symmetric node-travelling cost matrix
"	int	numNodes	Number of nodes in the strongly connected graph
"	int	population	Number of nodes permutation (possible solution) found at each round
"	int	bestNum	Number of best elements (parents) that will produce the next generation
"	int	numThreads	Number of processing elements that are due to work on each parallel section
Output Return	void		

Phase 1- Path cost computation & Rank initialisation

```
    int i,j,source,destination,*generation_rank;

    generation_rank = new int[population];

    // COST VECTOR COMPUTATION & RANK INITIALISATION
    fill(generation_cost, generation_cost+population, 0);
#pragma omp parallel for num_threads(numThreads) private(source,destination,i,j) schedule(static)
    for(i=0; i<population; ++i){
        // cost of last node linked to the first one
        source = generation[i*numNodes+numNodes-1];
        destination = generation[i*numNodes];
        generation_cost[i] += cost_matrix[source*numNodes+destination];
        // cost of adjacent cells
        for(j=0; j<numNodes-1; ++j){
            source = destination;
            destination = generation[i*numNodes+j+1];
            generation_cost[i] += cost_matrix[source*numNodes+destination];
        }

        generation_rank[i]=i;
    }
```

This section needs to compute the traversing cost for each element in the population matrix: this task apparently is easily parallelised over the different rows as they are independent from each others, but experiments showed that executing this piece of code in a sequential order might end up in being faster than the parallel version: intuitively this might be caused by the fact that, in the later iterations stages, the top permutations will look more and more similar to each other, thus a sequential execution will take big cache speed up: probably the portion of cost matrix relative to the node pair that we want to traverse is already loaded in cache; this clearly doesn't happen when parallelised, because different threads allocated to different executing cores evidently need to work on different L1/L2 cache as well, and eventually this procedure will soften the big speed up described before (we can think of a tradeoff between sequential cache hit and parallelisation over a matrix with very many long rows).
The method of computing the path cost is pretty straightforward: fill the array with zeros once and then start exploring each row summing for each pair (starting connecting the last node with the first one and then translating by one) the relative cost.

Phase 2 - Cost & Rank sorting

```
void sort_vector(int *generation_rank, int *generation_cost, int population, int numThreads){
    int low,high;
    low=0;
    high=population-1;

    #pragma omp parallel num_threads(numThreads)
    #pragma omp single
    mergesort(generation_cost, generation_rank, low, high, numThreads);
}
```

At this point, the procedure sorts the obtained cost array and, in order to keep track of the rows associated to a particular cost, performs the same operation to an auxiliary indexes array (that has been sequentially initialised in the previous phase).
For this very reason, a custom version of the parallel merge-sort has been implemented (the parallelisation has been carried out splitting the original array in as many pieces as the specified number of threads, obtaining different sorted pieces that can be merged in a tree fashion, in parallel as well).
The implemented merge-sort version is the standard one that costs *O(n)* in term of memory, so in order to deal with this complication a in-place quick-sort has been provided; moreover it can be expected that choosing the middle+1 element as pivot falls in the best case (pivot=middle, in fact it is indeed, at the start, the first newly generated permutation, having on its left the already ordered survivors of the previous iteration (when survivor rate is set to 50% of course) and on its right all the remaining generated ones; then the recursion tend to falls nearer and nearer to the best case).

Phase 3 - Matrix rearranging

```
void move_top(int *generation_rank, int *&generation, int *&generation_copy, int numNodes, int bestNum){
    int i,*start,*swap;
    for(i=0; i<bestNum; ++i){
        start = generation+generation_rank[i]*numNodes;
        copy(start, start+numNodes, generation_copy+i*numNodes);
    }
    swap = generation;
    generation = generation_copy;
    generation_copy = swap;
}
```

In order to keep the program logic as simple as possible, the procedure keep track of two population matrices, which will be swapped only after the copy from one to the other of the best survivor tracked by the ordered generation rank.

MESSAGE EXCHANGING
[*transferReceive_bests_allReduce(...)*]

Type	*Datatype*	*Name*	*Description*
Input Arg	int*	generation	Pointer to the permutation matrix (population*nodes) for the current iteration
"	int*	generation_cost	Pointer to the total permutation cost array
"	int	numNodes	Number of nodes in the strongly connected graph
"	int	bestNum	Number of best elements (parents) that will produce the next generation
Output Return	void		

```
int buff_size,*send_buff,*recv_buff;
MPI_Op op;

buff_size = numNodes+1;
send_buff = new int[buff_size];
recv_buff = new int[buff_size];

copy(generation, generation+numNodes, send_buff);
send_buff[numNodes] = generation_cost[0];

MPI_Op_create((MPI_User_function *)minimumCost, 1, &op);

MPI_Allreduce(send_buff, recv_buff, buff_size, MPI_INT, op, MPI_COMM_WORLD);

if (!equal_permutations(generation, recv_buff, numNodes)){
    copy(recv_buff, recv_buff+numNodes, generation+(bestNum-1)*numNodes);
    generation_cost[bestNum-1] = recv_buff[numNodes];
}

return;
```

As message passing, the program makes use of the custom Allreduce api of MPI, in fact it exactly performs what is needed: every node exchange its own buffer (filled preventively with its current best permutation and cost), then, every time it receives another path, it executes the specified comparison (custom MPI operation).

```
void minimumCost(int *in, int *out, int *len, MPI_Datatype *dtype){
    if(in[*len-1] < out[*len-1]){
        for (int i=0; i<*len; ++i){
            out[i]=in[i];
        }
    }
}
```

Simply, if the input (received) one has a smaller cost, overwrite the precedent (output). Two details should be noted: the first one is that we allow to keep track of two different minimum-cost permutations (the *equal_permutation* method serves exactly this purposes), while the second is that if the current iteration performs a message exchange, the pipeline will skip the next phase (convergence testing) in order to elaborate properly the obtained permutation.

CONVERGENCE TESTING

```
// TEST EARLY STOP (with short-circuit to ensure that lastRounds is filled
// before computing the stdDev over it)
if(i>=earlyStopRounds && stdDev(lastRounds, earlyStopRounds)<=earlyStopParam){
    // move to next exchange session (hoping that can help moving out from a fake convergence)
    // ... moreover other nodes might continue to expect messages
    if(i<maxIt-TRANSFERRATE){
        i += TRANSFERRATE-(i%TRANSFERRATE)-1;
    }
    solution[numNodes+1] = 1;
}
```

At each iteration, the program takes the first AVGELEMS best costs, computes the average of them and store it in the array responsible for early-stop testing (in a round-robin fashion).
Whenever the standard deviation of the array values falls below a fixed threshold, we can state that, at least for the last array-length iterations, the procedure has not seen a

significant global improvement, so it will jump straight to the next message exchange phase (the last assignment is just for stating a possible convergence of the procedure).

PROGRAM LAUNCH

PARAMETERS

As it has been stated, different hyper parameter play a role in the convergence and computational cost of the whole process (names refers to what is inside the two bash scripts, *phase1.sh* and *phase2_3.sh*):

- numCities: number of node in the strongly connected graph; the computational cost is surely proportional to this parameter; it can be used as threshold for deciding whether to parallelise or not;
- initialPop: number of permutation living at each iteration (rows in the generation matrix); besides being direct proportional to the program computational cost, it has implication in the convergence of the procedure too;
- top: percentage of initialPop that are going to act as parents at each iteration; clearly this parameters needs to be correctly tuned (in correlation with initialPop) in order to obtain a good ratio for convergence purposes;
- maxIt: number of possible iterations executed in the program; of course, this value must be sufficiently big (depending on numCities) for not cutting off the convergence procedure prematurely;
- mutP: probability of having a mutation in a newly generated permutation; depending on the chosen mutation procedure, a small value can result in being insufficient for escaping from local minima, while a big one can lead to frequent jumps in the cost function, possibly harming the learning process;
- earlyStRound: how many iterations are going to be considered in the early stop phase; surely it must be a value smaller than the message exchanging rate, but big enough to catch a descent in the cost function;
- earlyStParam: standard deviation threshold below the which the convergence of the process can be assumed; like the previous one, it needs to be tuned in order to avoid fake convergence.

PHASE 1

As a preliminary step it is useful to find an appropriate ratio between population size (*initialPop*) and survivor percentage (*top*), having the number of nodes in the graph (*numCities*) fixed.

Exploring the literature about genetic algorithms[1], it has been found out that an analytic relation between the initial population (and in this case, not only initial, since the amount does not change during the program execution) and the probability of having every possible edges between any two nodes (thus, having the one that participate in the optimal "tour" too) exists.

What is needed here, clearly, is a crossover algorithm that, generating a new solution, preserves the order of the past permutation (condition in part satisfied by the program, at least the first half of it; from this consideration we can think of improving the

crossover procedure alternating the maintaining of the first permutation half with the one that preserves the second part instead; from this point of view, the mutation process acts in part against it, but the observed benefits from the experiments suggest to keep it), in this way the algorithm is assured to converge in a certain amount of steps (not deterministic because of the random nature of the generation process; truth to be said, another important ingredient is required: bad permutation must not be completely discarded but just modified a litlle, this because they can contain, in some part, the optimal solution, that unfortunately is ruined by very costly other-nodes traversing).
The relation is

$$(1 - (\frac{n-3}{n-1})^S)^n \geq p \Rightarrow S \geq \frac{\log(1 - \sqrt[n]{p})}{\log((n-3)/(n-1))},$$

where S is the population size, p is the probability of having the whole optimal tour pieced in the starting population and n is the number of nodes in the graph.
So, this bash script exactly build a grid with all the possible combination of the chosen ranges for p (in the formula) and the survivor percentage (what it has been defined with *top*), thus finding convenient values that can both reach a decent minimum and don't explode on term of computational cost (all the other parameters remain fixed for now).
In order to have some reliable statistics of the measured values, every run is repeated a fixed number of times (variable *tries*); the script uses MPI (without communication) for parallelising the execution for what concern the sequential and parallel version, while a simple for cycle is performed for parallel+MPI.
For each row (setting) the output will consist of a triad of keys ([<*nodes number*> <*population size*> <*survivor amount*>]) followed by four other numbers representing the taken measures ([<*time taken*> <*minimum path cost*> <*convergence boolean*> <effectively executed iterations>]).

PHASE 2&3

Now, having a set of parameter that should provide a decent convergence rate, the algorithm can be tested on different graph sizes, thus it is easy to explore the computational cost of both the whole procedure (phase 2) and the singular building components (phase 3) as well.
So, for each value in the node range, the program is run in its three form (sequential, parallel and parallel+MPI) and output some standard prints that can be easily analysed (output of phase 2: same as for phase 1; output of phase 3: [<*nodes number*> <*population size*> <*survivor amount*> <single *stage time taken*>]).

RESULTS

DLTM CLUSTER

Experiments are carried out on a (available) 9 nodes cluster, each one providing a Dual-CPU Intel E52695 V3 with 28 core and 64gb of ram, the nodes are connected through a Infiniband FDR 56Gb/s; finally, each C++ instance is compiled with gcc version 4.8.5 20150623 (Red Hat 4.8.5-16), following the c++11 standard and the O3 optimiser flag turned on.

phase1.sh

Name	*Value*	*Description*
node_tries	3	How many times the simulation is run (for statistical purposes); also it represents the amount of cluster nodes used for MPI
numThreads	28	Number of processing elements that are due to work on each parallel section
numCities	1000	Number of nodes in the strongly connected graph
maxIt	10000	Number of max iteration rounds
mutP	0.5	Probability [0-1] of mutation occurrence in the newly generated permutation
earlyStopRound	100	Number of latest iterations from which the average of best AVGELEMS must be computed in order to establish convergence
earlyStopParam	100	Comparison parameter (standard deviation) for early stopping
i	[1, 2, 3]	Probability ($\frac{i}{10}$) of having the optimal tour with the random initialisation
j	[0.3, 0.4, 0.5]	Survivor percentage
k	[1…node_tries]	Simulation execution for-index for MPI approach

As stated before, at this point we are interested in finding a relation between population size and survivor amount that could lead to a good convergence rate (having fixed the graph size); the tried combination are {0.1, 0.2, 0.3}x{0.3, 0.4, 0.5} (optimal solution probability * survivor rate) for a total of 3 tries per approach (the first set of parameter leads to a population of respectively [3031, 3209, 3354] for a strongly-connected graph with 1000 nodes).

row index: population size, survivors amount;
col index: #trial;
value: minimum path cost

- *Sequential*

	try_0	try_1	try_2
3031,909	339.406566	345.221404	332.788625
3031,1212	345.294467	348.365348	299.753162
3031,1515	345.133937	336.027888	302.712236
3209,962	354.226609	343.637329	354.968825
3209,1283	363.904408	343.561741	367.816299
3209,1604	356.966093	355.128495	352.836406
3354,1006	398.040958	391.581365	398.465387
3354,1341	371.626766	407.827612	362.624501
3354,1677	345.003429	366.145586	372.691695

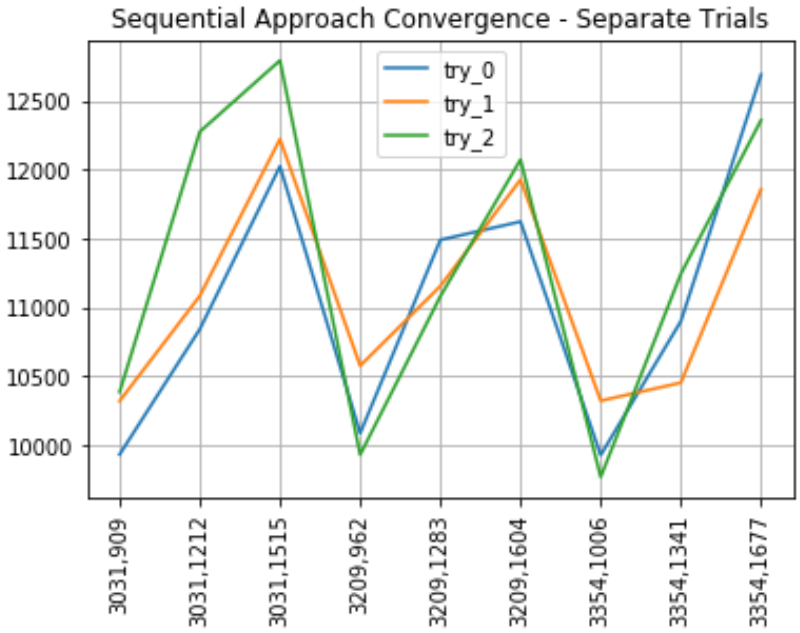
Execution Time

	try_0	try_1	try_2
3031,909	9931.0	10319.0	10385.0
3031,1212	10841.0	11083.0	12276.0
3031,1515	12025.0	12223.0	12794.0
3209,962	10085.0	10575.0	9929.0
3209,1283	11490.0	11154.0	11080.0
3209,1604	11623.0	11928.0	12073.0
3354,1006	9926.0	10321.0	9767.0
3354,1341	10899.0	10452.0	11243.0
3354,1677	12691.0	11858.0	12360.0

Minimum Permutation

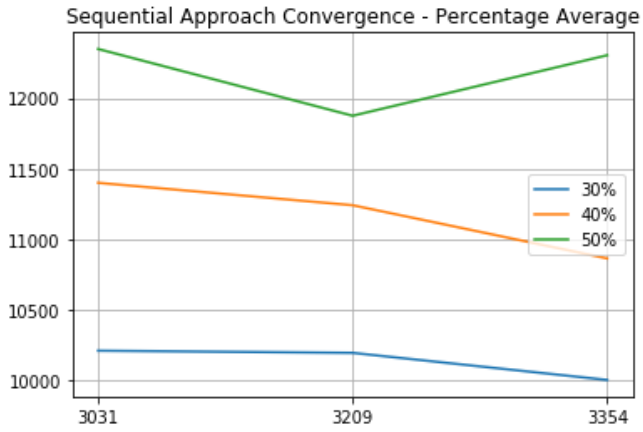
	try_0	try_1	try_2
3031,909	True	True	True
3031,1212	True	True	True
3031,1515	True	True	True
3209,962	True	True	True
3209,1283	True	True	True
3209,1604	True	True	True
3354,1006	True	True	True
3354,1341	True	True	True
3354,1677	True	True	True

Convergence Bool



Separate Trials

	30%	40%	50%
3031	10211.666667	11400.000000	12347.333333
3209	10196.333333	11241.333333	11874.666667
3354	10004.666667	10864.666667	12303.000000



Average Between Trials

- *Parallel*

	try_0	try_1	try_2
3031,909	148.758974	152.380505	152.939369
3031,1212	143.995698	146.483390	152.194323
3031,1515	131.132891	144.259436	134.627901
3209,962	159.521772	166.695077	162.088382
3209,1283	159.903185	171.817444	163.488503
3209,1604	154.090729	170.519979	145.639278
3354,1006	173.596418	177.165146	186.892266
3354,1341	160.979713	163.567252	165.052323
3354,1677	164.263544	163.924177	159.342698

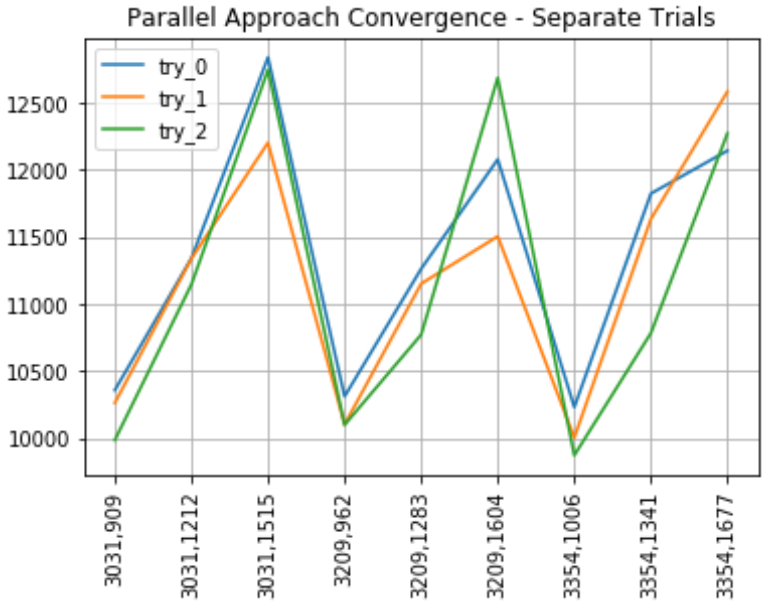
Execution Time

	try_0	try_1	try_2
3031,909	10359.0	10264.0	9987.0
3031,1212	11342.0	11340.0	11147.0
3031,1515	12839.0	12203.0	12746.0
3209,962	10312.0	10103.0	10100.0
3209,1283	11263.0	11152.0	10772.0
3209,1604	12079.0	11505.0	12687.0
3354,1006	10232.0	10005.0	9873.0
3354,1341	11824.0	11634.0	10784.0
3354,1677	12144.0	12585.0	12273.0

Minimum Permutation

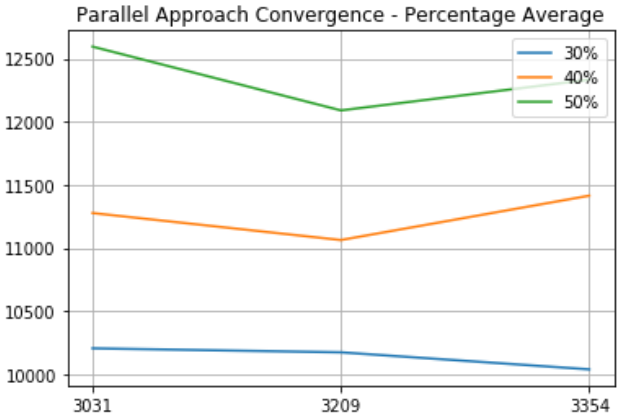
	try_0	try_1	try_2
3031,909	True	True	True
3031,1212	True	True	True
3031,1515	True	True	True
3209,962	True	True	True
3209,1283	True	True	True
3209,1604	True	True	True
3354,1006	True	True	True
3354,1341	True	True	True
3354,1677	True	True	True

Convergence Bool



Separate Trials

	30%	40%	50%
3031	10203.333333	11276.333333	12596.000000
3209	10171.666667	11062.333333	12090.333333
3354	10036.666667	11414.000000	12334.000000



Average Between Trials

- *Parallel + MPI*

	try_0	try_1	try_2
3031,909	286.279	286.118	285.949
3031,1212	265.851	266.002	265.715
3031,1515	254.607	254.747	254.549
3209,962	302.037	302.432	301.887
3209,1283	286.189	286.212	286.13
3209,1604	248.743	248.813	248.738
3354,1006	306.958	306.884	306.744
3354,1341	283.141	283.176	283.12
3354,1677	260.309	260.341	260.143

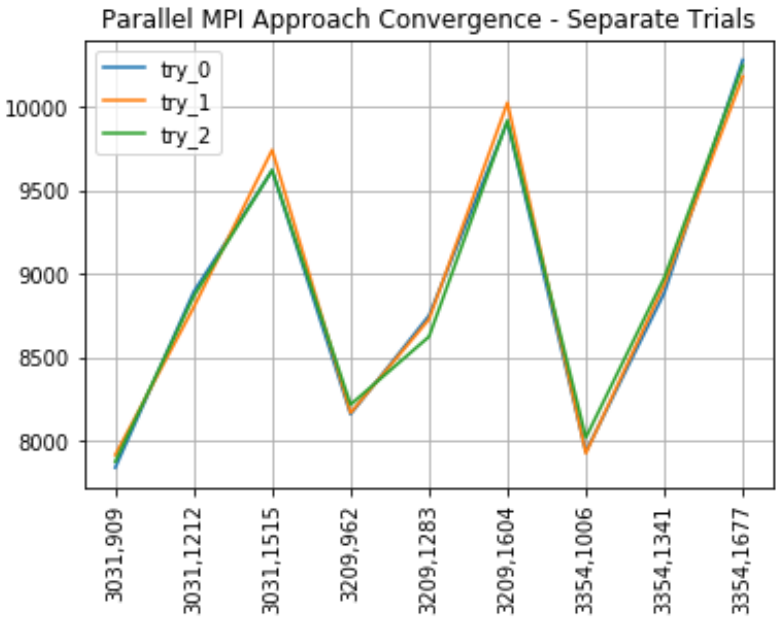
Execution Time

	try_0	try_1	try_2
3031,909	7842.67	7918.33	7878.67
3031,1212	8893.33	8801.67	8866.67
3031,1515	9619	9741.67	9620.67
3209,962	8162	8169.67	8219
3209,1283	8751	8729.33	8625.33
3209,1604	9910.67	10024	9917.67
3354,1006	7937	7927.67	8022
3354,1341	8887	8937.67	8978
3354,1677	10279.7	10181.3	10245.7

Minimum Permutation

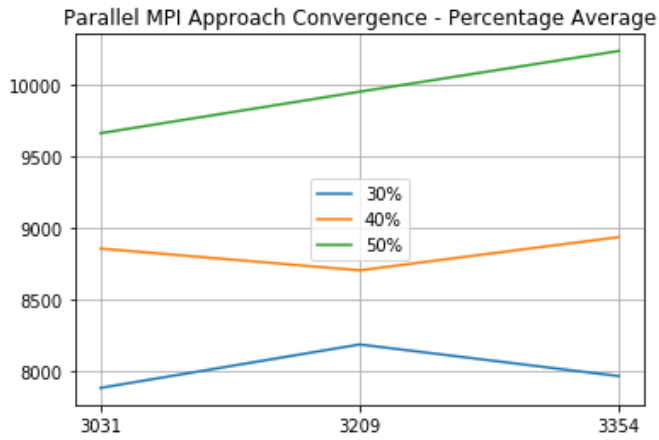
	try_0	try_1	try_2
3031,909	True	True	True
3031,1212	True	True	True
3031,1515	True	True	True
3209,962	True	True	True
3209,1283	True	True	True
3209,1604	True	True	True
3354,1006	True	True	True
3354,1341	True	True	True
3354,1677	True	True	True

Convergence Bool



Separate Trials

	30%	40%	50%
3031	7879.888889	8853.888889	9660.444444
3209	8183.555556	8701.888889	9950.777778
3354	7962.222222	8934.222222	10235.555556



Average Between Trials

At first glance, we can notice some nice expected behaviours:

1. The parallelisation over the pipeline phases offer a double speed-up against to sequential version (more details will be analysed during phase 2&3), while the third approach, clearly, add the overhead of realising the communication to the parallel costs (still less than the first one);
2. Different execution reach, more or less, the same convergence values, showing that, even with different initialisation, the procedure leans towards the absolute minimum; there are no observable big difference between the first two approaches, while the third one show a cost improvement of around 2000, proving the effective benefits of the message passing procedure (moreover, the

found minima are practically equal, thus, giving more iteration and a more strict convergence testing, the program can be expected to find a more efficient path):

3. Looking at the averages graphs, it can be easily noticed that the lowest survivor rate (30%) improves the convergence procedure towards the true minima in all the three approaches, on the other hand, no statistical stability can be inferred in the decision for the best solution probability, consequently, the parameter that lead to the lowest population amount will be chosen (in order to decrease the program computational cost).
4. From the convergence tables, we see that probably the chosen parameter is a bit too soft (since every procedure believes to have reached convergence), a justification can reside in the big similarity of the different permutation minima, but probably a second testing with a more strict parameter is needed .

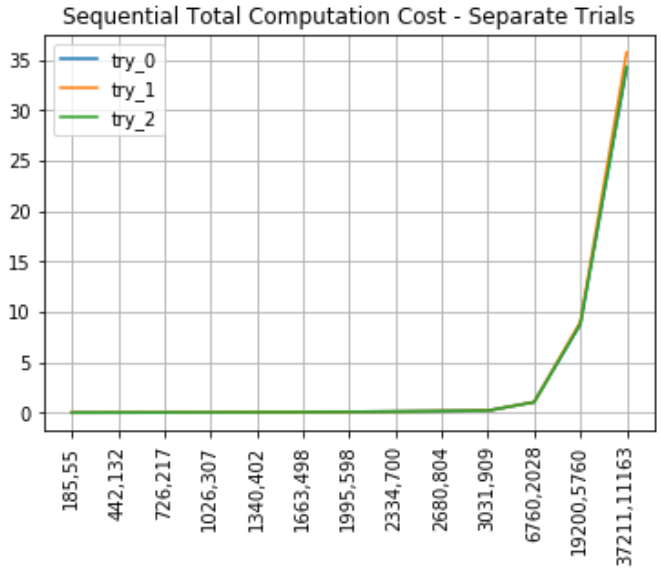
phase2_3.sh

Name	*Value*	*Description*
node_tries	3	How many times the simulation is run (for statistical purposes); also it represents the amount of cluster nodes used for MPI
numThreads	28	Number of processing elements that are due to work on each parallel section
top	0.3	Best survivor percentage found from phase 1
maxIt	100	Number of max iteration rounds
mutP	0.5	Probability [0-1] of mutation occurrence in the newly generated permutation
earlyStopRound	9	Number of latest iterations from which the average of best AVGELEMS must be computed in order to establish convergence
earlyStopParam	1	Comparison parameter (standard deviation) for early stopping
numCities	[100, 200, … , 1000, 2000, 5000, 9000]	Number of nodes in the strongly connected graph
k	[1…node_tries]	Simulation execution for-index for MPI approach

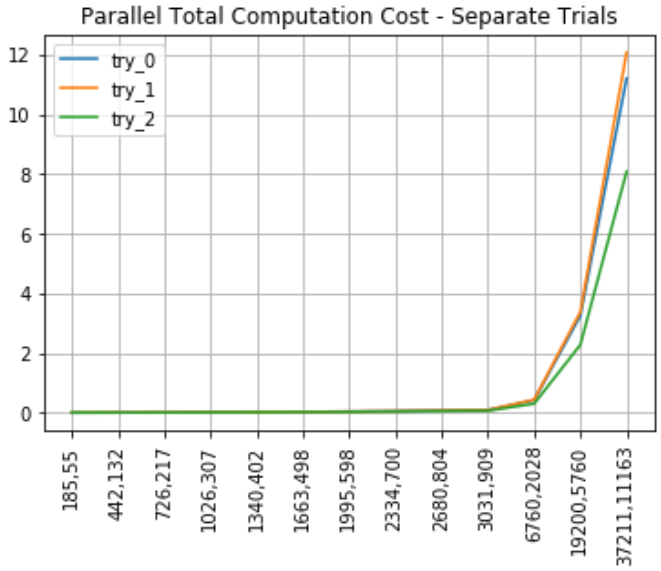
Having some already generated cost matrix (increasing in size) as input and having found a possible good convergence parameters for what concerns the population size and the survivor rate, we proceed computing the computational cost of the single iteration (phase 2) firstly, then the one of each pipeline stage (phase 3).

row index(phase 2): population size, survivor amount;
row index(phase 3): graph size, population size, survivors amount;
col index: #trial;
value: computational cost *or* iteration taken by that trials

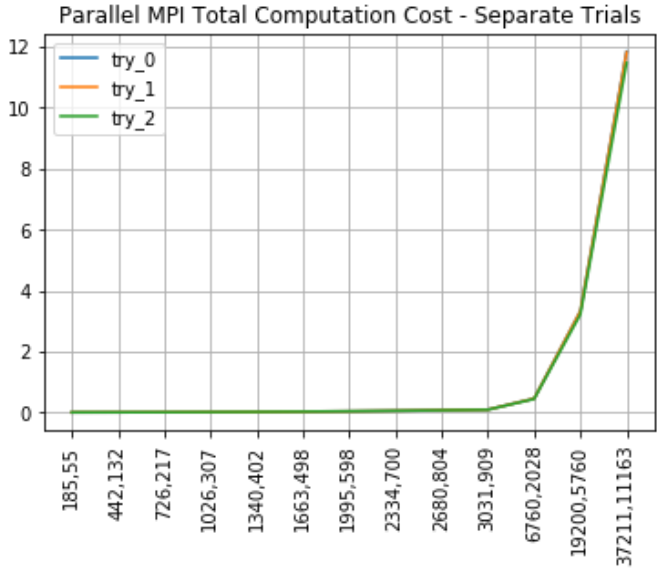
- *Total Procedure*



	try_0	iter_try_0	try_1	iter_try_1	try_2	iter_try_2
185,55	0.00054752	100	0.00053578	100	0.00056497	100
442,132	0.00369994	100	0.00370463	100	0.00347649	100
726,217	0.0110134	100	0.0113805	100	0.0110633	100
1026,307	0.023196	100	0.0227427	100	0.0226211	100
1340,402	0.0394509	100	0.0404405	100	0.0396069	100
1663,498	0.0632101	100	0.0613553	100	0.062667	100
1995,598	0.0883338	100	0.0897711	100	0.0900594	100
2334,700	0.123515	100	0.122202	100	0.122481	100
2680,804	0.160298	100	0.161826	100	0.160999	100
3031,909	0.204852	100	0.207648	100	0.205511	100
6760,2028	1.04458	100	1.06207	100	1.04617	100
19200,5760	8.75596	100	9.01148	100	8.79014	100
37211,11163	34.1228	100	35.7325	100	34.3174	100

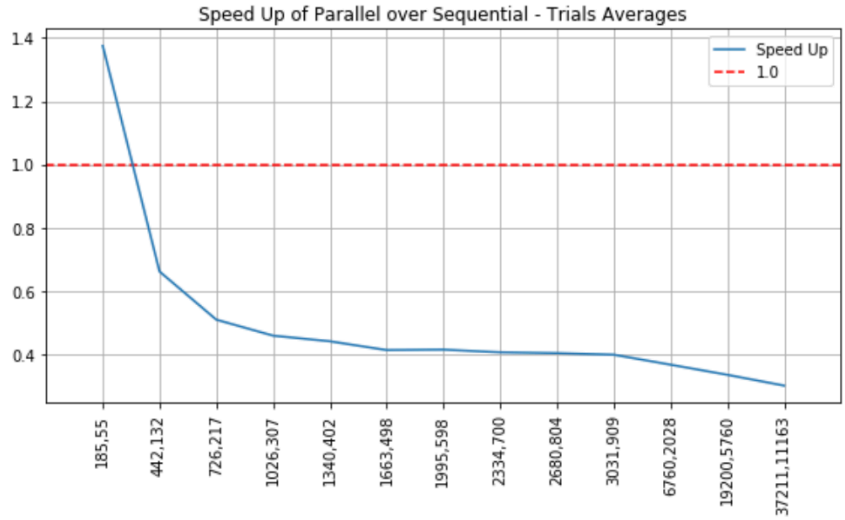


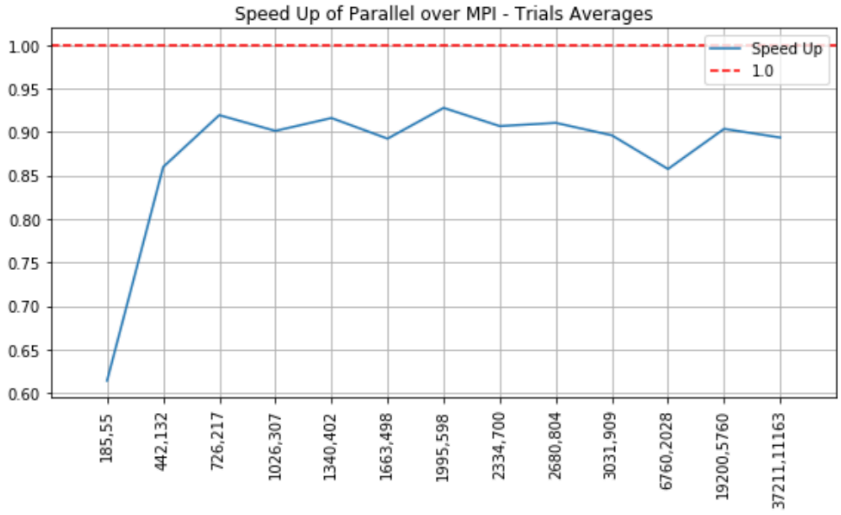
	try_0	iter_try_0	try_1	iter_try_1	try_2	iter_try_2
185,55	0.00070755	100	0.00070896	100	0.0008493	100
442,132	0.00249332	100	0.00241209	100	0.00229399	100
726,217	0.00575023	100	0.00613112	100	0.00516527	100
1026,307	0.0109695	100	0.0112024	100	0.00929868	100
1340,402	0.0183944	100	0.0191058	100	0.0152228	100
1663,498	0.0276241	100	0.0281201	100	0.0216727	100
1995,598	0.0400033	100	0.0400112	100	0.031239	100
2334,700	0.0542018	100	0.0536534	100	0.0417972	100
2680,804	0.0701772	100	0.0711812	100	0.0537071	100
3031,909	0.0896013	100	0.0900639	100	0.066827	100
6760,2028	0.42098	100	0.437428	100	0.299645	100
19200,5760	3.25192	100	3.37917	100	2.27572	100
37211,11163	11.2091	100	12.0838	100	8.08414	100



	try_0	iter_try_0	try_1	iter_try_1	try_2	iter_try_2
185,55	0.00122583	100 100 100	0.00122392	100 100 100	0.00123806	100 100 100
442,132	0.00279288	100 100 100	0.00279682	100 100 100	0.00278385	100 100 100
726,217	0.00620291	100 100 100	0.00620701	100 100 100	0.00613195	100 100 100
1026,307	0.0116623	100 100 100	0.0117378	100 100 100	0.0115184	100 100 100
1340,402	0.0192797	100 100 100	0.0193274	100 100 100	0.0189481	100 100 100
1663,498	0.0290635	100 100 100	0.0291683	100 100 100	0.0285226	100 100 100
1995,598	0.0402845	100 100 100	0.0402352	100 100 100	0.0394033	100 100 100
2334,700	0.0554078	100 100 100	0.0554085	100 100 100	0.0542268	100 100 100
2680,804	0.0718765	100 100 100	0.0720685	100 100 100	0.070305	100 100 100
3031,909	0.0925472	100 100 100	0.0925544	100 100 100	0.0899529	100 100 100
6760,2028	0.453645	100 100 100	0.455515	100 100 100	0.44139	100 100 100
19200,5760	3.32023	100 100 100	3.32259	100 100 100	3.21402	100 100 100
37211,11163	11.8378	100 100 100	11.8113	100 100 100	11.4629	100 100 100

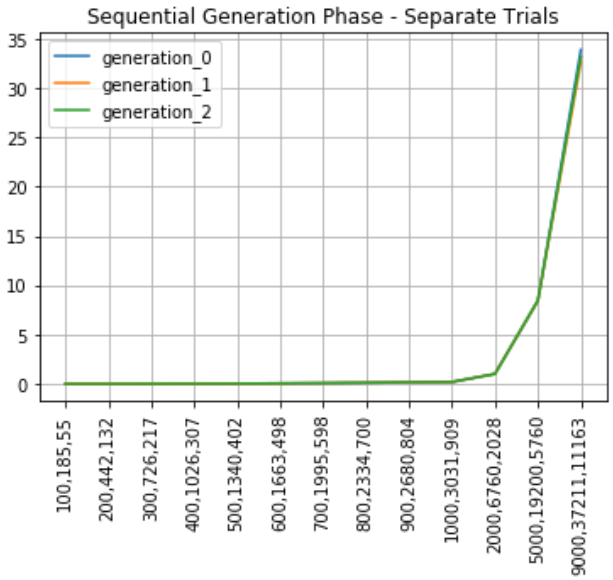
- *Total Speed-Ups*



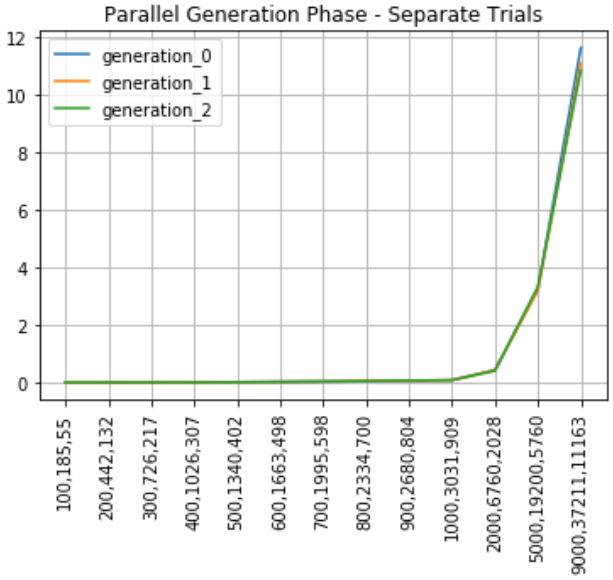


1. The phase 1 tables show that the procedure follow a linear increase in computational cost proportional to the graph size; measures are taken measuring the time taken for carrying out the entry point method and dividing it by the amount of executed iterations (aka pipelines, in this way we should have the gross idea of the expense of executing one round); where more than one measures for a particular setting is taken (e.g. parallel MPI, 3), an average is computed (mean of the several *time/iterations*) ;
2. The first speed-up graph (parallel over sequential) confirms the preferable choice of the sequential approach for very small matrices (as we could have expected), for the bigger ones it shows a very steep improvement that eventually converges at almost 0.35 ($\approx$ 3 times faster);
3. The second speed-up graph (parallel over MPI) also shows in details what was said for phase 1: the communication costs are expensive for smaller matrices (MPI if in fact 40% slower), while for the bigger ones the generation stage represents the bottleneck (the communication impacts only on 10% of the total time) .

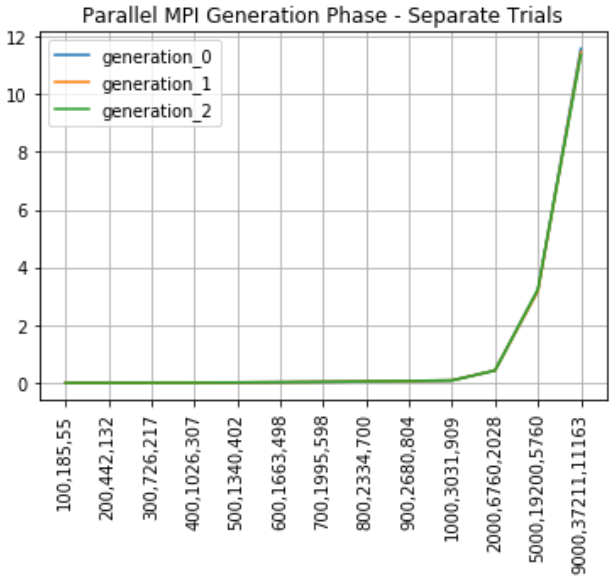
- *Generation*



	generation_0	generation_1	generation_2
100,185,55	0.00056523	0.00055211	0.00050178
200,442,132	0.00349537	0.00380623	0.00382193
300,726,217	0.0113997	0.0111109	0.0111691
400,1026,307	0.0230168	0.0221508	0.022481
500,1340,402	0.0388959	0.0391952	0.03927
600,1663,498	0.0605222	0.060792	0.0590094
700,1995,598	0.0866398	0.0861573	0.0864146
800,2334,700	0.118545	0.11894	0.118491
900,2680,804	0.156468	0.154801	0.156328
1000,3031,909	0.198957	0.198819	0.19803
2000,6760,2028	1.0091	1.00926	1.00934
5000,19200,5760	8.45639	8.43929	8.47188
9000,37211,11163	33.9199	32.6617	33.2086

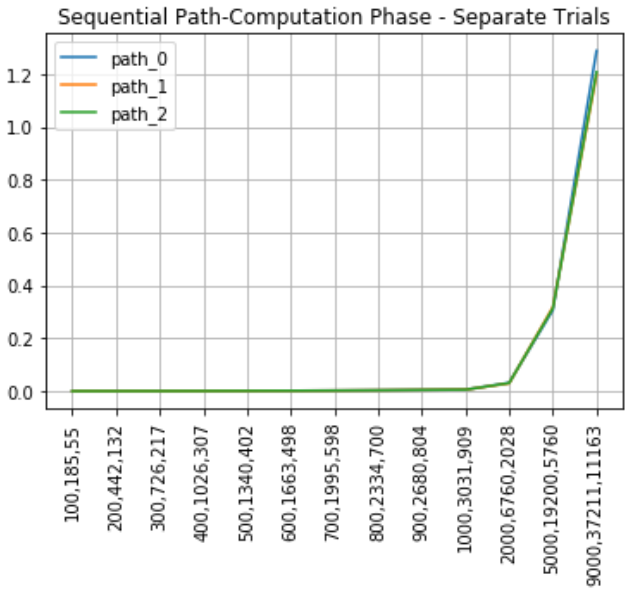


	generation_0	generation_1	generation_2
100,185,55	0.00049948	0.00050207	0.00050395
200,442,132	0.00219286	0.00217313	0.00218617
300,726,217	0.00541842	0.00547503	0.00545071
400,1026,307	0.0108199	0.010543	0.0108286
500,1340,402	0.0173653	0.0173695	0.0179983
600,1663,498	0.0262968	0.0269825	0.0264132
700,1995,598	0.0375376	0.0365777	0.0369734
800,2334,700	0.0505974	0.0504549	0.0509916
900,2680,804	0.0681611	0.0664941	0.0682702
1000,3031,909	0.083461	0.0876217	0.0847994
2000,6760,2028	0.438513	0.43233	0.407385
5000,19200,5760	3.26045	3.20832	3.3429
9000,37211,11163	11.6492	11.1167	10.8684

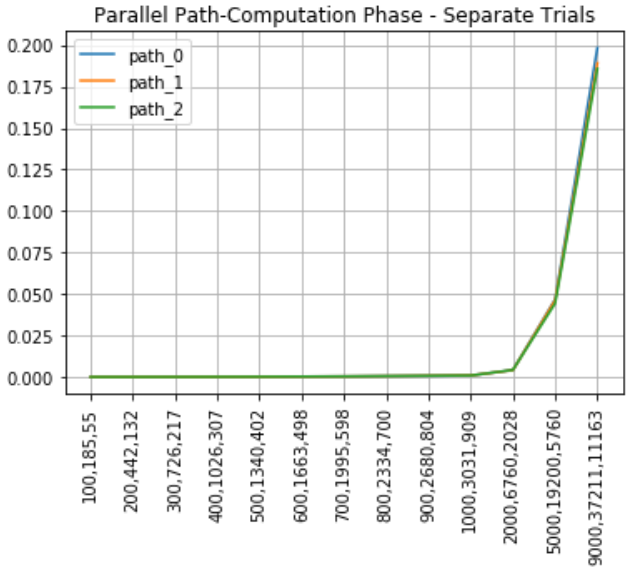


	generation_0	generation_1	generation_2
100,185,55	0.00049627	0.00049203	0.000500227
200,442,132	0.0021908	0.00221251	0.00218129
300,726,217	0.00547511	0.00547565	0.0055057
400,1026,307	0.010481	0.0107003	0.010492
500,1340,402	0.0176836	0.0175157	0.0176592
600,1663,498	0.0264755	0.0266044	0.0265024
700,1995,598	0.0372764	0.0375759	0.037351
800,2334,700	0.0510121	0.0515986	0.0507607
900,2680,804	0.067563	0.0681778	0.0674828
1000,3031,909	0.0852611	0.0881653	0.0875502
2000,6760,2028	0.431893	0.432044	0.427501
5000,19200,5760	3.20947	3.18452	3.24666
9000,37211,11163	11.5798	11.4596	11.3698

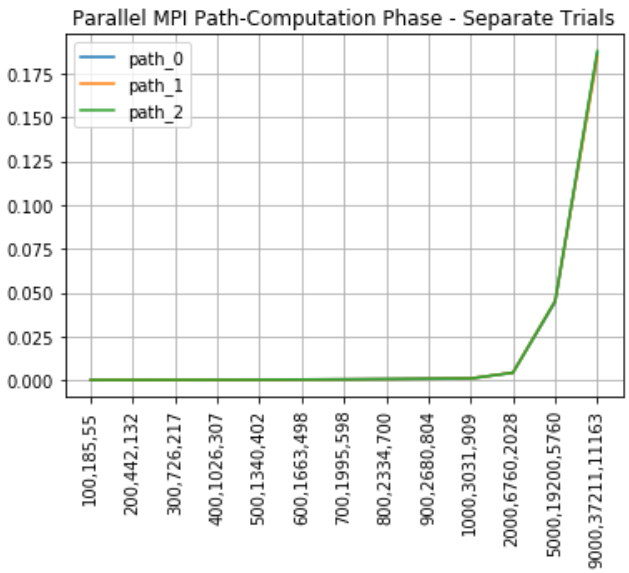
- *Path Cost Computation*



	path_0	path_1	path_2
100,185,55	2.78515e-05	2.78713e-05	2.8198e-05
200,442,132	0.000130842	0.000132495	0.000132515
300,726,217	0.000350842	0.000349772	0.00034795
400,1026,307	0.000691495	0.000687505	0.000685832
500,1340,402	0.00114934	0.00116366	0.00115734
600,1663,498	0.00172924	0.00173334	0.00173514
700,1995,598	0.00245904	0.00242976	0.00249174
800,2334,700	0.00341498	0.00341271	0.0033921
900,2680,804	0.00456679	0.00453939	0.00460423
1000,3031,909	0.00570965	0.00567551	0.00551907
2000,6760,2028	0.0316409	0.0294577	0.0306989
5000,19200,5760	0.306912	0.318367	0.313822
9000,37211,11163	1.28893	1.19778	1.20802

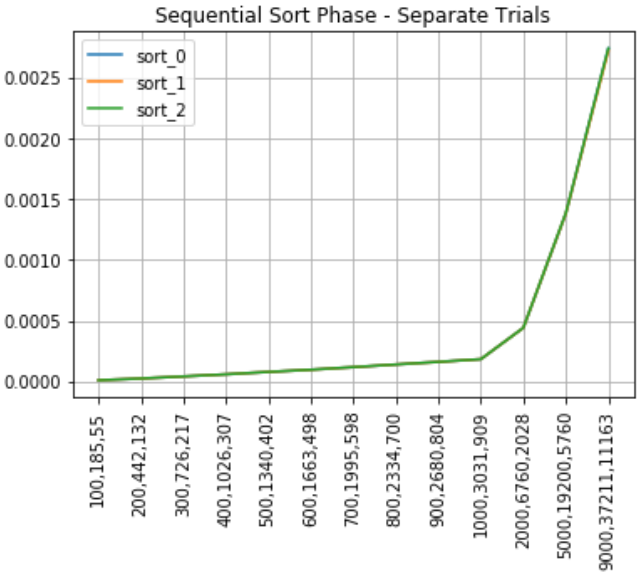


	path_0	path_1	path_2
100,185,55	4.88416e-05	4.76436e-05	4.83663e-05
200,442,132	6.31089e-05	6.1604e-05	6.08812e-05
300,726,217	9.27624e-05	9.13366e-05	9.4802e-05
400,1026,307	0.000143455	0.000146248	0.000148901
500,1340,402	0.000212752	0.000210416	0.000214861
600,1663,498	0.000326564	0.000320921	0.000326624
700,1995,598	0.000424475	0.000425426	0.000419842
800,2334,700	0.000585396	0.000579198	0.000578594
900,2680,804	0.000726733	0.000770911	0.000753921
1000,3031,909	0.000911347	0.000962564	0.000919832
2000,6760,2028	0.0042257	0.0041405	0.00428326
5000,19200,5760	0.0467991	0.0463932	0.0444054
9000,37211,11163	0.198305	0.189173	0.185747

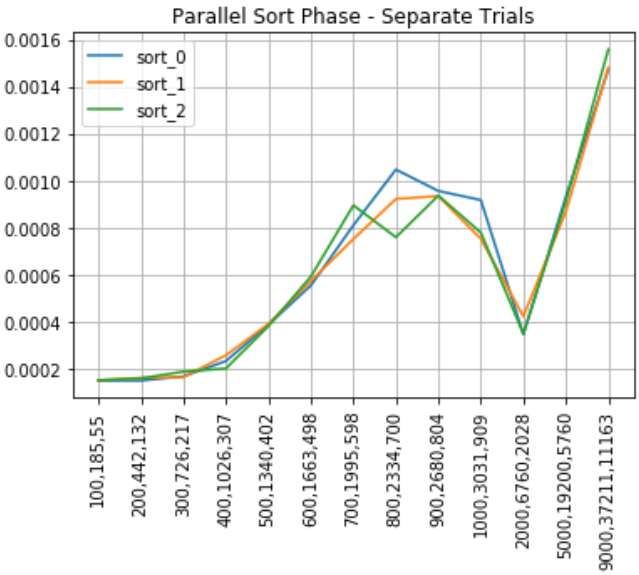


	path_0	path_1	path_2
100,185,55	4.79802e-05	4.7495e-05	4.72475e-05
200,442,132	6.15677e-05	6.18449e-05	6.11551e-05
300,726,217	9.42112e-05	9.33234e-05	9.30825e-05
400,1026,307	0.000145584	0.000148795	0.000145274
500,1340,402	0.000214149	0.000210502	0.000211762
600,1663,498	0.00031702	0.000310752	0.000324182
700,1995,598	0.000419822	0.000422875	0.000420191
800,2334,700	0.000593789	0.000550333	0.000568106
900,2680,804	0.000758937	0.000730416	0.000758977
1000,3031,909	0.000923515	0.000912888	0.000911861
2000,6760,2028	0.0041686	0.00425706	0.00420766
5000,19200,5760	0.0445505	0.045171	0.0452733
9000,37211,11163	0.186737	0.184905	0.188117

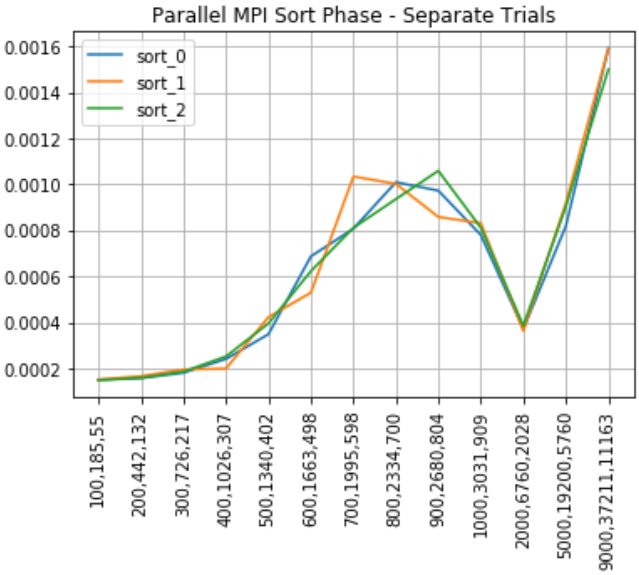
- *Sorting*



	sort_0	sort_1	sort_2
100,185,55	9.9703e-06	1.00495e-05	9.77228e-06
200,442,132	2.4e-05	2.40693e-05	2.43861e-05
300,726,217	4.14851e-05	4.14653e-05	4.14851e-05
400,1026,307	5.85347e-05	5.86337e-05	5.8604e-05
500,1340,402	7.84158e-05	7.85545e-05	7.85644e-05
600,1663,498	9.67228e-05	9.62673e-05	9.65149e-05
700,1995,598	0.000117257	0.000117287	0.000118168
800,2334,700	0.000139218	0.000139545	0.000139525
900,2680,804	0.000161564	0.000160594	0.000160871
1000,3031,909	0.000182871	0.000183149	0.000183376
2000,6760,2028	0.000441	0.000440465	0.000441218
5000,19200,5760	0.00138102	0.0013793	0.00138481
9000,37211,11163	0.00273829	0.00271751	0.00273771

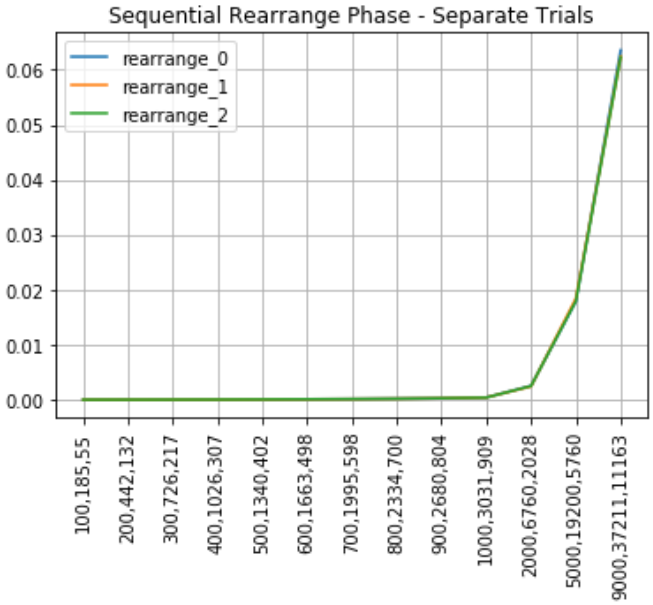


	sort_0	sort_1	sort_2
100,185,55	0.000151416	0.000152861	0.000152842
200,442,132	0.000151327	0.00016296	0.00016096
300,726,217	0.000169287	0.000166089	0.000189337
400,1026,307	0.000234366	0.000258832	0.000202842
500,1340,402	0.000388851	0.000391911	0.000382832
600,1663,498	0.000555485	0.000574129	0.000593317
700,1995,598	0.000811	0.000752446	0.000895683
800,2334,700	0.00104744	0.000922683	0.00076003
900,2680,804	0.000956525	0.000934861	0.00093695
1000,3031,909	0.000918089	0.000753257	0.000779733
2000,6760,2028	0.000347653	0.000423525	0.000350446
5000,19200,5760	0.000931564	0.000868535	0.00091203
9000,37211,11163	0.00147239	0.0014809	0.00155837

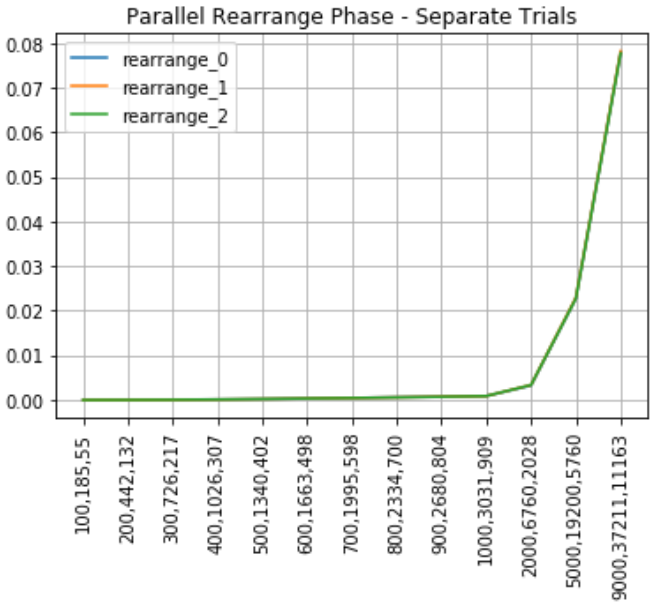


	sort_0	sort_1	sort_2
100,185,55	0.000150634	0.000151043	0.000148073
200,442,132	0.000157069	0.000166112	0.000158591
300,726,217	0.000180538	0.000194769	0.000185548
400,1026,307	0.000240518	0.000198696	0.000251686
500,1340,402	0.000348145	0.00042268	0.000394974
600,1663,498	0.000686802	0.000528845	0.00062271
700,1995,598	0.000807198	0.00103415	0.000811053
800,2334,700	0.00100933	0.00100207	0.000935779
900,2680,804	0.000972637	0.000858099	0.00105895
1000,3031,909	0.00077965	0.000831426	0.000811063
2000,6760,2028	0.000370908	0.000362492	0.000385386
5000,19200,5760	0.000818063	0.000918099	0.000902663
9000,37211,11163	0.0015914	0.00159022	0.00149961

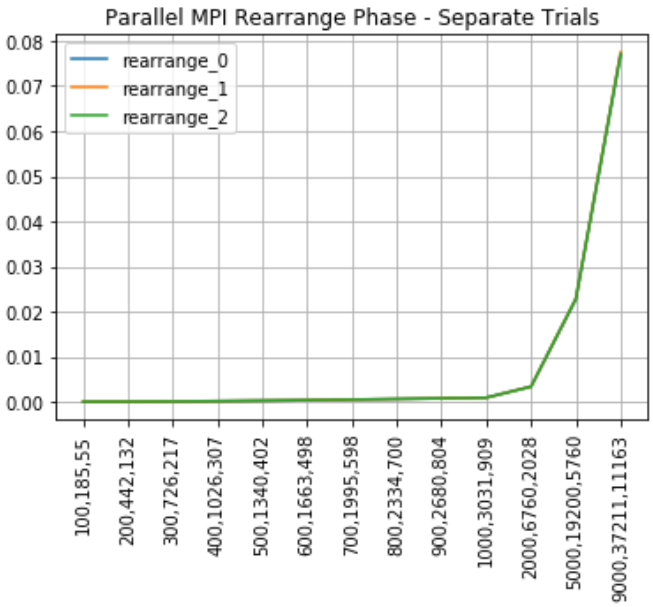
- *Rearrange*



	rearrange_0	rearrange_1	rearrange_2
100,185,55	1.08911e-06	1.07921e-06	1.07921e-06
200,442,132	7.12871e-06	7.13861e-06	7.24752e-06
300,726,217	1.91782e-05	1.91485e-05	1.91584e-05
400,1026,307	3.69109e-05	3.66436e-05	3.46535e-05
500,1340,402	6.00396e-05	6.03465e-05	5.94257e-05
600,1663,498	9.1099e-05	8.92475e-05	9.15743e-05
700,1995,598	0.000126366	0.000125396	0.000127713
800,2334,700	0.000193762	0.000177455	0.000173267
900,2680,804	0.000259	0.000257347	0.000265406
1000,3031,909	0.000405762	0.000405	0.00038901
2000,6760,2028	0.00253017	0.0024173	0.00248379
5000,19200,5760	0.0178825	0.0184292	0.0179519
9000,37211,11163	0.0635744	0.0623968	0.0622983

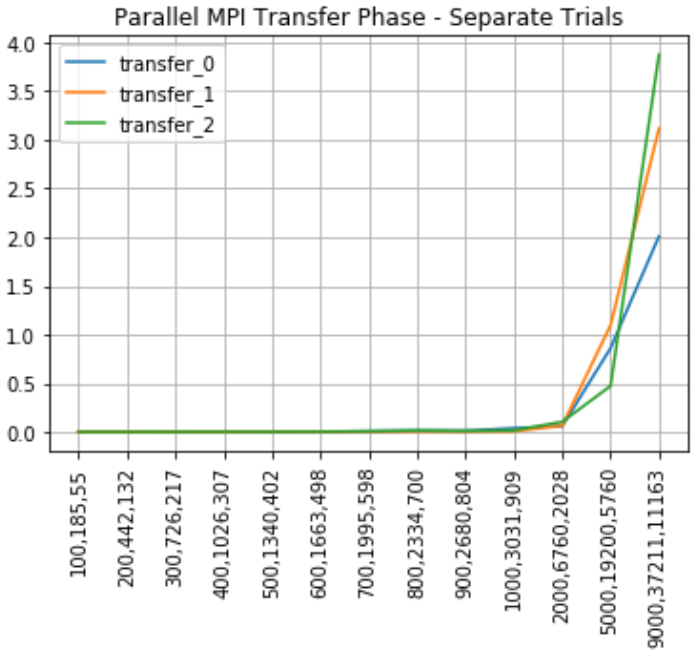


	rearrange_0	rearrange_1	rearrange_2
100,185,55	6.14851e-06	6.10891e-06	5.79208e-06
200,442,132	3.09307e-05	2.93663e-05	3.04554e-05
300,726,217	7.8604e-05	7.38614e-05	7.4297e-05
400,1026,307	0.000148228	0.000139802	0.000141644
500,1340,402	0.000230713	0.000229743	0.00023802
600,1663,498	0.000348792	0.000339525	0.000348178
700,1995,598	0.000477782	0.000470347	0.000470881
800,2334,700	0.000627099	0.000607851	0.000595455
900,2680,804	0.000734693	0.000748109	0.000733188
1000,3031,909	0.000897109	0.000897347	0.000898584
2000,6760,2028	0.00326286	0.00337926	0.00337705
5000,19200,5760	0.0227636	0.0229681	0.0227658
9000,37211,11163	0.0779312	0.078196	0.0777209



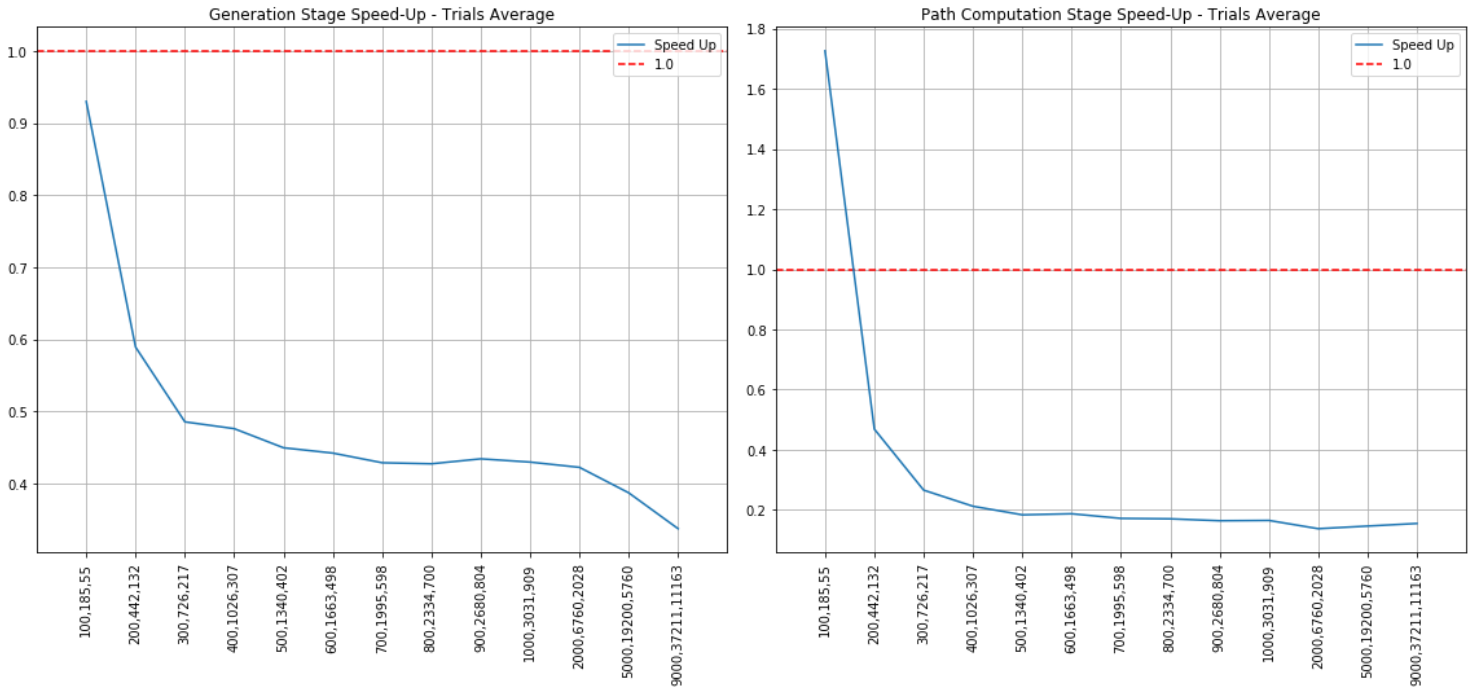
	rearrange_0	rearrange_1	rearrange_2
100,185,55	6.10561e-06	6.22442e-06	6.20462e-06
200,442,132	2.94191e-05	2.92079e-05	2.86799e-05
300,726,217	7.54653e-05	7.39934e-05	7.4297e-05
400,1026,307	0.000140261	0.000144106	0.000138162
500,1340,402	0.000240406	0.000235515	0.000231426
600,1663,498	0.000340462	0.000342287	0.000346637
700,1995,598	0.000464218	0.000469779	0.00047232
800,2334,700	0.000606779	0.000602795	0.000606851
900,2680,804	0.000738917	0.000746954	0.000733653
1000,3031,909	0.000899974	0.000898591	0.000903059
2000,6760,2028	0.00334197	0.00335532	0.00335609
5000,19200,5760	0.0226495	0.022864	0.0228655
9000,37211,11163	0.0770312	0.0774981	0.0771118

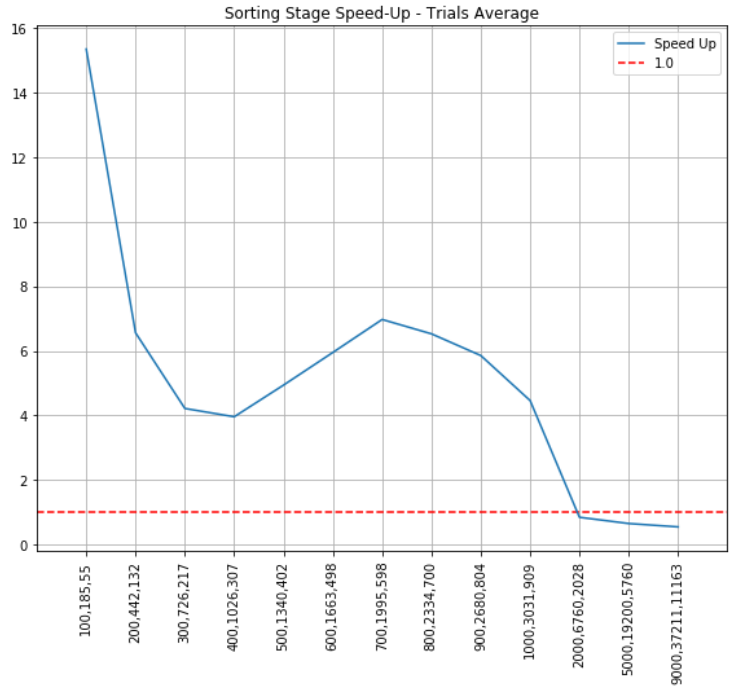
- *Message Passing*

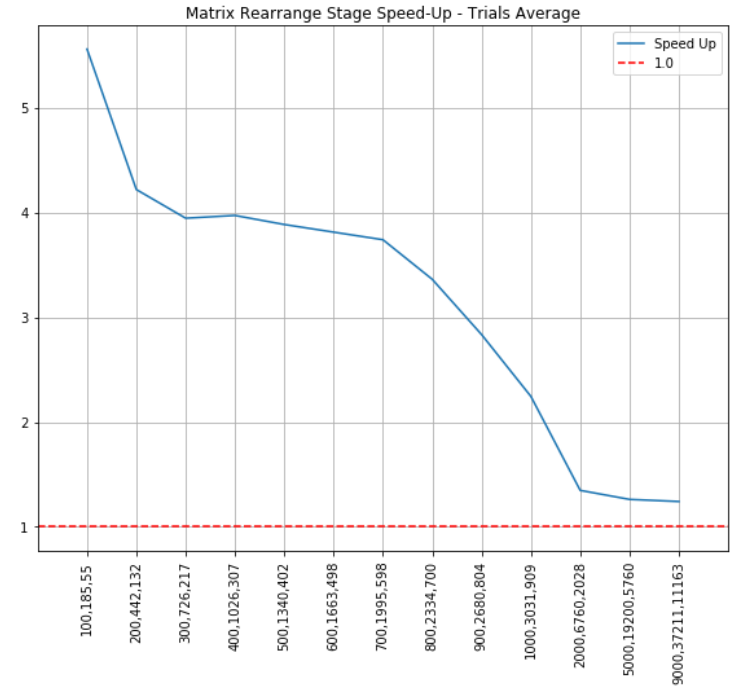


	transfer_0	transfer_1	transfer_2
100,185,55	0.00352493	0.00356759	0.00351381
200,442,132	0.00351289	0.00322044	0.00361741
300,726,217	0.00430785	0.00416989	0.004002
400,1026,307	0.00567733	0.00407574	0.00549622
500,1340,402	0.00453941	0.00621711	0.00502619
600,1663,498	0.00681341	0.00717704	0.0070463
700,1995,598	0.01182	0.00771093	0.0112291
800,2334,700	0.0155024	0.00963474	0.0190917
900,2680,804	0.0155851	0.0100037	0.0133413
1000,3031,909	0.0420961	0.0110855	0.0193101
2000,6760,2028	0.0630458	0.0665179	0.107512
5000,19200,5760	0.862791	1.09843	0.474904
9000,37211,11163	2.00989	3.11466	3.87288

- *Stages Speed-Ups (Parallel over Sequential)*







1. The generation phase clearly represents the bottleneck for each tested matrix size: the parallelisation eventually decently improve performances by almost a 3 times factor (with, recall, 28 working threads, probably too much for this kind of speedup); in second place, again about the most costly operation, the path computation benefits from parallel execution too by a factor of, this time, 6 times (from an average of 1.2'' to a decent 0.2''); the only procedure that does not take advantage of parallelisation is the rearrange one, where, evidently, the coordination overhead outweighs the possible benefits.

2. The sorting phase assumes a really interesting behaviour: experiments show that, for sizes that span from 100 to 1000 nodes, the sequential approach follows a slower computational cost increase, while the parallel versions (both simple and MPI) have, at first, a steeper trend (probably for the same reason of the rearrange phase in point *1*) that eventually ends up in being smoother with bigger matrices, consequently less expensive (no justification were found for the drastic decrease at 2000 nodes);
3. Different trials tend to confirm the same computational cost almost always, except for the message passing phase: this is possibly caused by different execution in time (for-loop) where different nodes (placed at different distance) are used; moreover if a node believes to have reached convergence, it will skip to the next permutation exchange phase, being locked in the message receiving waiting for all the other nodes, thus adding inactive seconds to the count;
4. The speed-up graphs tend to confirm the trend that has been outlined until now: the parallelisation overhead is worth paying for matrices that are bigger than a certain threshold (from these results happen to be ≈ 100-150 graph nodes), but we can also notice that the sorting (parallel merge sort) and the rearrange stages go against this statement: in fact, in order to obtain a positive parallelisation speed-up (< 1.0), they need even bigger matrices than the one that have been tested (from this point of view I suggest to review the code and choose algorithms that can take more benefits from multithreading rather than the ones that have been implemented).

REFERENCES

[1] "The Population Size Estimate", University of Vaasa