Travelling Salesman Problem

Master M2 – Université Grenoble Alpes

Parallel Systems

Dec-2019

Overview

I will implement the TSP problem in three versions and I will try to show the difference between pure MPI version and hybrid (MPI and openMP) version and the sequential version. And also I will show the difference between using the collective communication mpi_Gather Vs the normal send and receive in terms of time. Performance evaluation will be in terms of execution time, and it is tested on Grid'5000 and on my personal laptop.

TSP

Traveling Salesman Problem (TSP) is one of the most common studied problems in combinatorial optimization. Given the list of cities and the distances between them, the Challenge is finding the shortest route visiting each member of a collection of locations and returning to your starting point. E.g. Traveling Santa Problem ☺.

Santa may try every different route he can until he maps out the shortest, but this is grossly inefficient. It will cost him O(n!).
The increase happens in a dramatic (exponential) way. The number of possible routes for an *n* amount of cities is *n*!.

this is a table summarizing the exploding number of routes that Santa needs to try, as the cities increase.

0	1
1	1
2	2
3	6
4	24
5	120
6	720
7	5040
8	40320

11	3.991680×10^7
12	4.790016×10^8
13	6.227021×10^9
14	8.717829×10^{10}
15	1.307674×10^{12}
16	2.092279×10^{13}
17	3.556874×10^{14}
18	6.402374×10^{15}

Genetic Algorithms (GA) for TSP

We saw that The exact solution would be to try all permutations and choosing the cheapest one using brute-force search

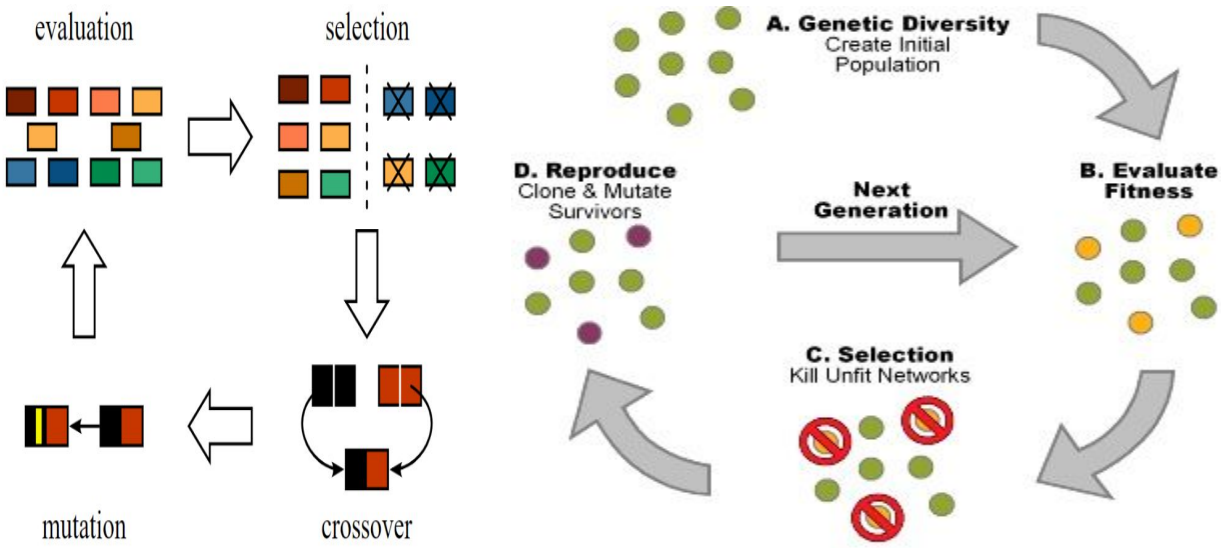
- NP-Complete, O(n!)
- Impractical, even for 20 cities

So using Genetic Algorithms (GA) we will try to solve the TSP. Genetic algorithms are commonly used to generate high-quality solutions to optimization and search problems by relying on bio-inspired operators such as mutation, crossover and selection.

- Candidate Solution => Individual or Chromosome, i.e. sequence of genes
- Population => (large) set of chromosomes
- Initial Generation => randomly generated population(s)
- Next generation => evolution of previous generation

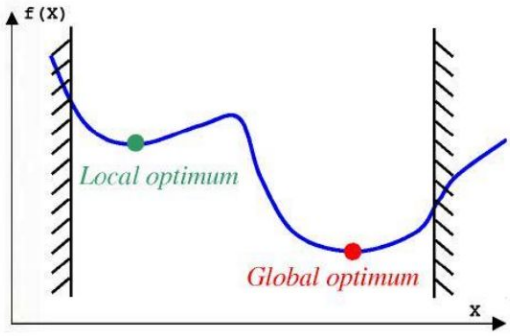
TSP using GA

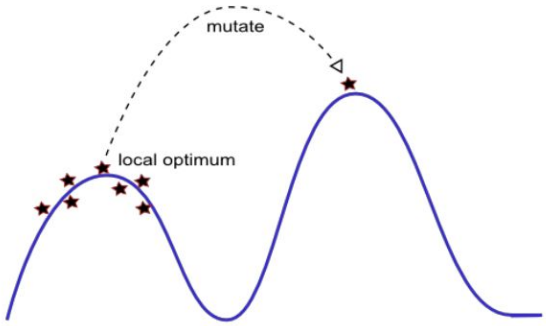
- Chromosomes: sequence of cities IDs (route)
- Fitness Function: path cost of a chromosome
- Selection: first 40% from the best Chromosomes and worst 10% of chromosomes
- Crossover: create new chromosome for 2 chromosomes that are selected according to some conditions (if the difference between the father and mother greater than 70% then select these 2 chromosomes to create the child)
- Mutation: just swap any two cities! In a chromosome



1. Next generation => evolution of previous generation by
    a. Reproduction of pairs of individuals i.e. crossover of 2 parent chromosomes to yield 2 offspring children
    b. Mutation of certain chromosomes change of certain gene(s) in a chromosome

Mutation to avoid Local Optima





Overview of Genetic Algorithm for the TSP

Each population consists of N chromosomes (routes), after calculation of the fitness (how much the path cost for each chromosome in the population) we start to create a new population (for the next generation) by making selection to choose which chromosome to keep for the next generation then crossover to create the new chromosomes then we do the mutation to avoid the local optimum and to keep the taste of randomness (2 non beautiful parents can birth a beautiful child). We stop by deciding how many generations we will do. The selection of the parents to create a child is when we find 2 parents from the selected part with 70% difference in path.

Parallelization

For parallelizing I used both technologies **MPI and OpenMp**. the main idea is to make workers work on subset of the main population (N/p where N is the number of chromosomes in a population and P is the number of workers processes) and then do the algorithm on a subset of the population locally in each process and finally send it back to master node and after the master receive all the subpopulations in one big population array master will sort and print the best path (note that also the master will work on a set of population).

Case study:

1. We will compare the performance for the parallel algorithms with the sequential algorithm(sequential vs pure MPI vs Hybrid) in terms of number of Generations.
2. There are 2 algorithms that I implemented for communication (for sending the sub population from each node to the master with and without collective communication), we will compare the two algorithms

three source codes are provided:

1. Sequential version
2. Pure MPI
3. Hybrid MPI with openMp version using collective communication
4. Hybrid without collective communication (MPI_Gather)

Note that the code is a bit big (~600 lines) but I commented it, and it is easy to read.

Implementation

The aim is to minimize total execution time by

- Maximize CPUs/Cores utilization
- Minimize communication time

First the input is a file wi29.txt which contains location of 29 cities, we read the file, we create the Distances Matrix (29 * 29) in each node (process).

Structures and Critical initializations

This is an example of initializing the number of chromosomes in a population and the number of generations. We will play with these values for the purpose of evaluation of the algorithm. Making them bigger for better view of the performance in each version(seq vs mpi vs mpi_openMP).

```
POPULATION_SIZE = 100 ;
NB_OF_GENERATION_PER_PROCESS = 500000;
SUB_POPULATION_SIZE_PER_PROCESS = POPULATION_SIZE / numtasks;
```

Chromosome Structure it contains an array of cities (genes) and the calculated fitness of this path.

```
typedef struct Chromosome{
    // genes represent a sequence of cities, i.e. a path
    int genes[29];
    float fitness;
} Chromosome;
```

Matrix of type Double: to store the distances between cities
population: Array of Chromosomes (size P * N routes)
Sub_population: Array of Chromosomes in each node (size = N)

Version of using collective communication MPI_Gather

```
MPI_Datatype newChroType;
MPI_Datatype type[2] = { MPI_INT, MPI_FLOAT};
int blocklen[2] = { CHROMOSOME_LENGTH, 1 };
MPI_Aint disp[2];

MPI_Init(&argc, &argv);
disp[0] = offsetof(struct Chromosome , genes);
disp[1] = offsetof(struct Chromosome , fitness);

MPI_Type_create_struct(2, blocklen, disp, type, &newChroType);
MPI_Type_commit(&newChroType);

MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &myID);

SUB_POPULATION_SIZE_PER_PROCESS = POPULATION_SIZE / numtasks;
```

In this version as we said before, we divide the population size by number of processes (SUB_POPULATION_SIZE_PER_PROCESS = N / P) and each process will start by initializing the sub population locally randomly(Chromosomes with random sequences of genes).

We create a new data type in mpi in order to be able to send an array of chromosomes (look at the chromosome structure) the new data type is defining a struct that contains array of int and a float, int type array we specify the types inside the structure and in blocklen we set the size of the corresponding type and disp array is used to represent addresses, which are unsigned quantities, after setting all these arguments we create structure in mpi and commit the datatype under name specified (newChroType).

After creating the datatype in all processes, process start to do the algorithm (selection - crossover - mutation - calculate fitness and sorting the population) it stops when it reaches the number of generations decided previously.

```
120       int i = 0 ;
121       while(i != NB_OF_GENERATION_PER_PROCESS)
122       {
123         i++;
124         selection(sub_population);
125         crossoverV2(sub_population);
126         mutation(sub_population);
127         calculate_population_fitness(sub_population);
128         sort_population(sub_population ,  SUB_POPULATION_SIZE_PER_PROCESS);
129       }
```

After this we need to collect the results in the master (node 0) and to collect we used the collective communication MPI_Gather in one version the normal mpi_send/recv in another version in order to compare the performance. As we can see in the figure below all nodes are sending there sub population to the master 0, and the master is collecting it into the population array with size (the full array of Chromosomes size = SUB_POPULATION_SIZE_PER_PROCESS * numtasks).

```
MPI_Gather( sub_population, SUB_POPULATION_SIZE_PER_PROCESS, newChroType, population, SUB_POPULATION_SIZE_PER_PROCESS, newChroType, 0, MPI_COMM_WORLD);
if(myID == 0)
{
  end_time = clock();
  double time_spent = (double)(end_time - start_time) / CLOCKS_PER_SEC;
  sort_population(population , POPULATION_SIZE);
  print_fitness(population , POPULATION_SIZE);
  printf("\nNumber of generations: %d\npopulation size per process: %d\npopulation size of master: (since 4 nodes we r runing 4*%d): %d" ,NB_OF_GENERATI
    printf("\nparallel Version using Both MPI and openMP: time measured on my personal pc: %f\n" , time_spent);
}
```

After collecting all the sub population arrays in the master, the master sort the full array of chromosomes (population) and print the fitness of the generation(in increasing order) the best path will be the first one.

```
----------------------------------------------Parallel Version using Both openMP and MPI of TSP----------------------------------------------


--------------------------------------------------------------------------------Fitness--------------------------------------------------------------------------------
24786.117 - 24786.117 - 24786.117 - 24786.117 - 24837.328 - 24837.328 - 24837.328 - 24837.328 - 24863.043 - 24863.043 - 24863.043 - 24863.043 - 24914.254 - 24914.254 -
24914.254 - 24914.254 - 24942.051 - 24942.051 - 24942.051 - 24942.051 - 24975.875 - 24975.875 - 24975.875 - 24975.875 - 24984.785 - 24984.785 - 24984.785 - 24984.785 -
24993.266 - 24993.266 - 24993.266 - 24993.266 - 24998.234 - 24998.234 - 24998.234 - 24998.234 - 34552.977 - 34552.977 - 34552.977 - 34552.977 - 36574.938 - 36574.938 -
36574.938 - 36574.938 - 43090.090 - 43090.090 - 43090.090 - 43090.090 - 44280.020 - 44280.020 - 44280.020 - 44280.020 - 44409.168 - 44409.168 - 44409.168 - 44409.168 -
46903.438 - 46903.438 - 46903.438 - 46903.438 - 50895.562 - 50895.562 - 50895.562 - 50895.562 - 53394.617 - 53394.617 - 53394.617 - 53394.617 - 53663.000 - 53663.000 -
53663.000 - 53663.000 - 54937.980 - 54937.980 - 54937.980 - 54937.980 - 55717.492 - 55717.492 - 55717.492 - 55717.492 - 56955.320 - 56955.320 - 56955.320 - 56955.320 -
59875.254 - 59875.254 - 59875.254 - 59875.254 - 63325.344 - 63325.344 - 63325.344 - 63325.344 - 116760.828 - 116760.828 - 116760.828 - 116760.828 - 137404.797 - 137404.
797 - 137404.797 - 137404.797 -
--------------------------------------------------------------------------------END---------------------------------------------------------------------------------

Number of generations: 5000
population size per process: 25
population size of master: (since 4 nodes we r runing 4*25): 100
parallel Version using Both MPI and openMP: time measured on my personal pc: 0.424284
```

example of a results using hybrid version, this is received in the master and the fittiest chromosome is of cost 24786, as we didn't do all the permutations and we used a Genetic algorithm, we can't say that this is the optimal one path cost. It is also dynamic maybe next run it

would give different values (which is rare if the number of generations is big) to be more accurate we need to increase the number of generations so it could give better results.

```
salman@salman-HP-Pavilion-Notebook:~/Desktop/Salman_farhat_project$ ./runComm.sh

-------------------------------------------------Generation Size 100----------------------------------------------------------

the best path founded is :
Fitness = 30503.117188  , Genes = 21_17_14_20_16_25_27_24_26_28_29_23_18_22_19_15_13_9_3_7_12_10_11_4_8_5_6_2_1_

parallel Version using Both MPI and openMP: time measured on my personal pc: 0.087359
the best path founded is :
Fitness = 33172.621094  , Genes = 11_10_12_15_19_29_28_26_23_22_17_16_25_27_24_20_14_21_18_13_8_9_7_4_3_5_6_2_1_

parallel Version using Both MPI and openMP: using normal mpi send receive 0.093779

-------------------------------------------------End --------------------------------------------------------------------------
```

Result for comparing 2 versions and printing the best path founded.

Use OpenMp to add more parallelism in each node

Some of the main parts that we used multiple threads is in calculating fitness and in crossover (creating new child). Here in the figure below I tried different scheduling but the most efficient one was the static (dynamic , static , guided) , also I tried different chunk size, but I didn't observe better results so i kept it to the machine to decide how many chunks it need to use partition.

```
void calculate_population_fitness(Chromosome *population){
  omp_set_dynamic(0);     // Explicitly disable dynamic teams
  omp_set_num_threads(4); // Use 4 threads for all consecutive parallel regions
  // printf( "number of threads runing is %d" ,  omp_get_num_threads() );
#pragma omp parallel for schedule(static)
  for(int i = 0 ; i < SUB_POPULATION_SIZE_PER_PROCESS ; i++){
    calculate_fitness(&population[i]);
  }

// #pragma omp parallel
// {
// #pragma omp parallel for schedule(static)
//   for(int i = omp_get_thread_num() ; i < SUB_POPULATION_SIZE_PER_PROCESS ; i =i + omp_get_num_threads() ){
//     calculate_fitness(&population[i]);
//   }
// }
}
```

The commented part, i was parallelizing it manually (without using #pragma omp parallel for) and I observe that it was not efficient as (#pragma omp parallel for schedule(static))

```
void crossoverV2(Chromosome *pop){
  int j=0 , nb=0;
  #pragma omp parallel for schedule(static)
  for(int i = 0 ; i <( SUB_POPULATION_SIZE_PER_PROCESS/2)  ; i++){
    do{
      nb= getRandomNumber()%(SUB_POPULATION_SIZE_PER_PROCESS/2);
    }while(nb == i && percentage_of_difference(pop[i] , pop[nb]) < 70);
    create_ChildV2(pop[i] , pop[nb] , &pop[(SUB_POPULATION_SIZE_PER_PROCESS/2) +i]);
  }
// }
  //create_ChildV2(pop[(POPULATION_SIZE/2)-1] , pop[0] , &pop[POPULATION_SIZE-1]);
}
```

Evaluation

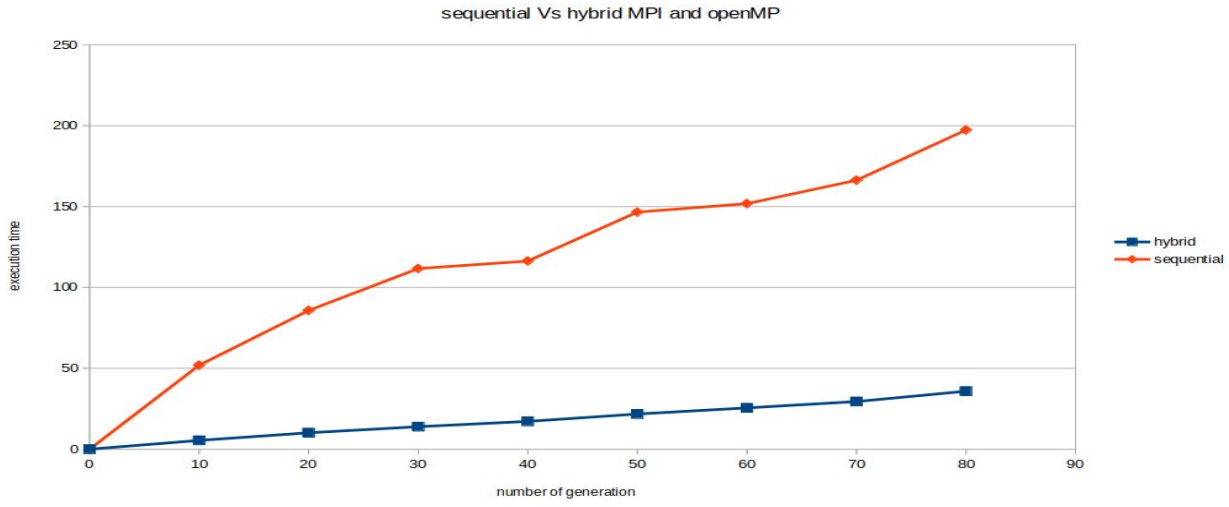
In this part, we will study the performance between the 3 versions (sequential , mpi alone , mpi_openMP) (mpi using 4 up to 80 machines), also the performance of using collective communication vs the using mpi_send/recv. The study will be conducted on the Grid'5000 and on my personal laptop.

Compare between the 3 versions on my personal laptop in terms of time execution

First I created a shell script to run the algorithm with different number of generations automatically (**run.sh**).

Sequential Vs Mpi_openMP

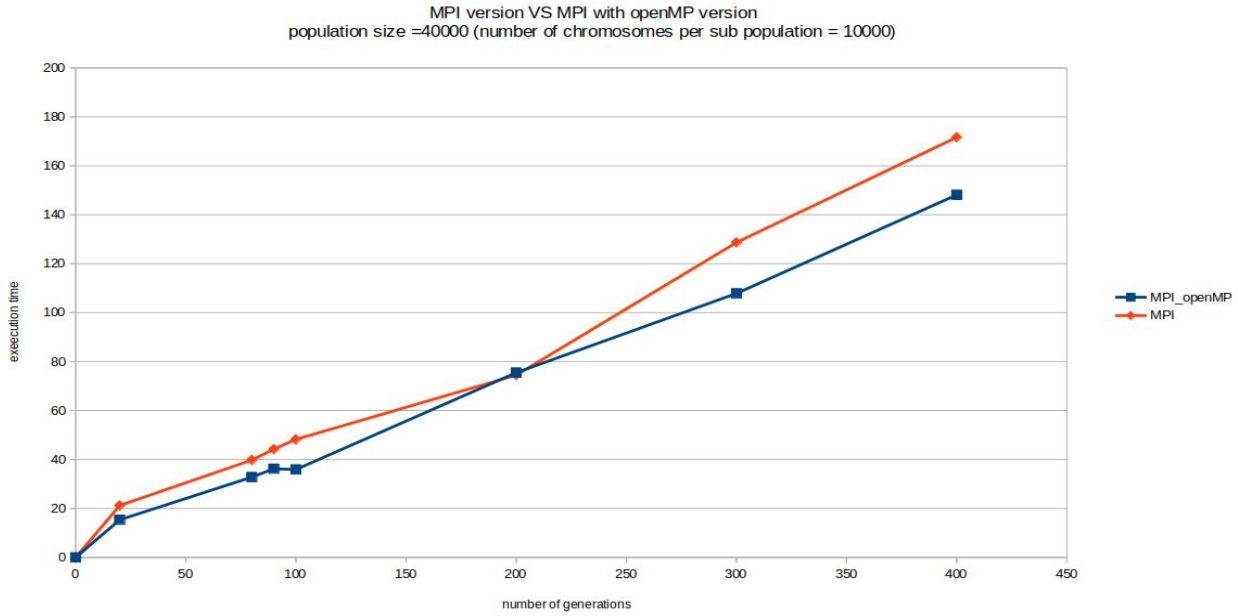
Population size 40,000, using 4 machines (sub population per process = 10,000)
First study is the sequential with hybrid version, as we can see in the figure below as number of generations the execution time of both versions increase but it is slightly increasing in mpi_openMP version while it is increasing faster in sequential version.



So we can see in Figure above that we actually have better performance while using hybrid or pure mpi version over the sequential version and the difference between time execution is deserving to go parallel.
Note That we used the same algorithm in sequential and parallel, the only difference is that once we have more nodes we split the population size on all nodes and each perform the same operations on the subset of the population and at the end master collect them.

Pure MPI Vs Hybrid (Mpi, openMP)

As we can see in Figure below as number of generations increase execution time increase. But the results are not satisfying, not as what I was expecting the hybrid is slightly better than the pure mpi version.




```
salman@salman-HP-Pavilion-Notebook:~/Desktop/Parallel_Project$ ./run.sh
-------------------------------------------------Generation Size 100----------------------------------------------------------------------------

parallel Version using Both MPI and openMP: time measured on my personal pc: 35.906919
parallel Version using MPI: time measured on my personal pc: 48.213737
-------------------------------------------------End --------------------------------------------------------------------------------------

-------------------------------------------------Generation Size 200----------------------------------------------------------------------------

parallel Version using Both MPI and openMP: time measured on my personal pc: 75.476445
parallel Version using MPI: time measured on my personal pc: 74.458431
-------------------------------------------------End --------------------------------------------------------------------------------------

-------------------------------------------------Generation Size 300----------------------------------------------------------------------------

parallel Version using Both MPI and openMP: time measured on my personal pc: 107.863780
parallel Version using MPI: time measured on my personal pc: 128.690396
-------------------------------------------------End --------------------------------------------------------------------------------------

-------------------------------------------------Generation Size 400----------------------------------------------------------------------------

parallel Version using Both MPI and openMP: time measured on my personal pc: 148.139517
parallel Version using MPI: time measured on my personal pc: 171.657808
-------------------------------------------------End --------------------------------------------------------------------------------------
```

The thing that was decreasing my performance in openMp is that i have dependencies between calculations that limits my ability to use openMP. The second thing is the value of population

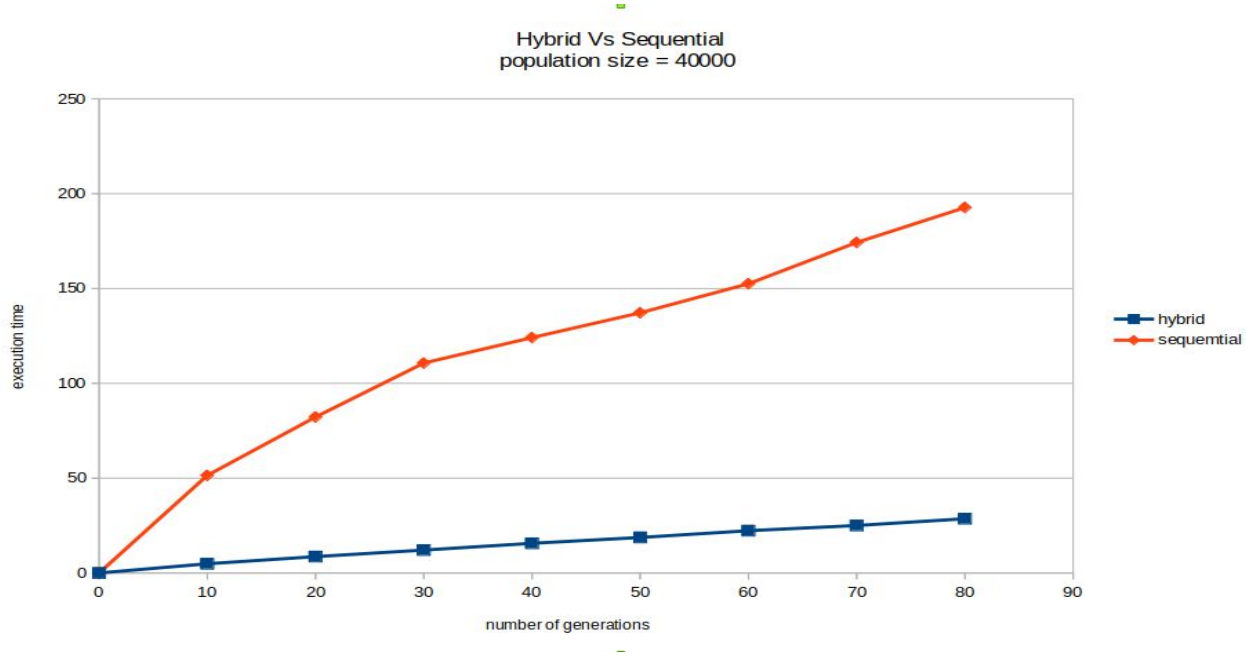
size is small to view the performance varying, openMP is applied and used in crossover and calculating population fitness so if we increase the number of chromosomes in population we will start to view better behavior.

Compare between the 3 versions on Grid'5000 in terms of time execution

In this part we have same programs tested on Grid'5000.

Sequential Vs Mpi_openMP

Here as we can see the results are satisfying

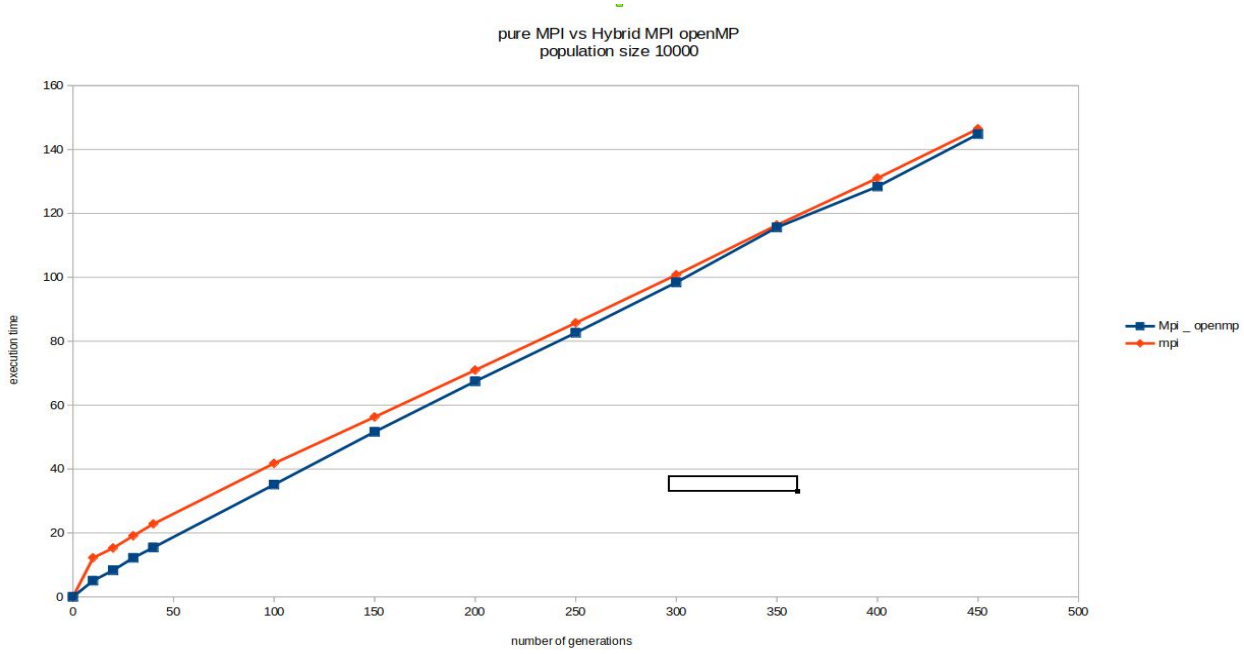


Pure MPI Vs Hybrid on Grid

Case 1

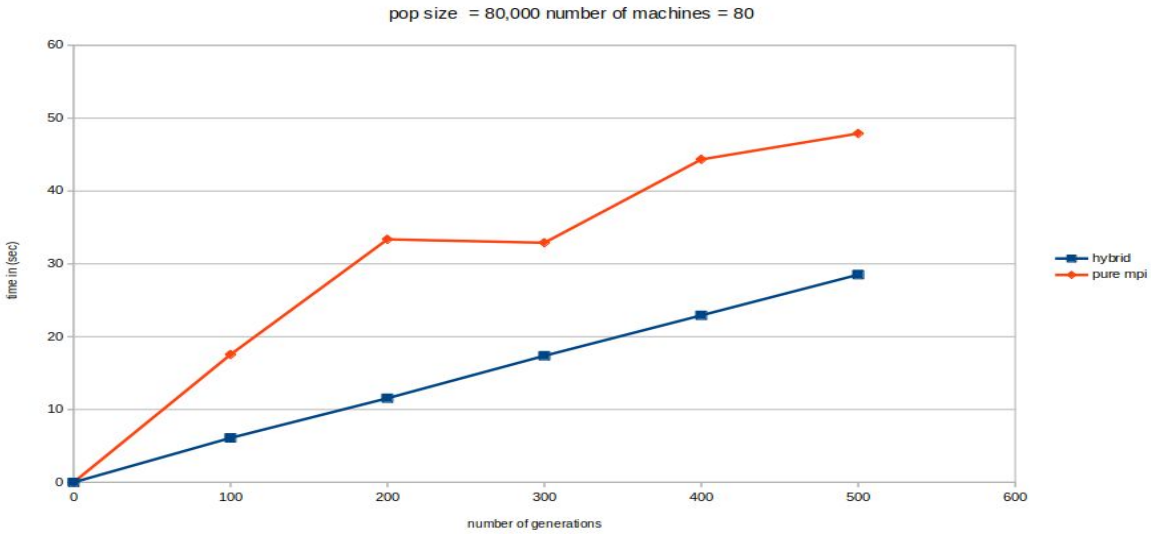
Total Population size = 40000 per process = 10000 and using 4 machines

Here we can observe that as the number of generations increase the mpi version is getting closer to the hybrid version.



Case 2

Population size per process = 80000 and using 80 processors



80 processors because i wanted the program to use 3 different machines dahu-6 , 7 and 8 each machine with 32 cores.

On personal laptop

```
salman@salman-HP-Pavilion-Notebook:~/Desktop/Parallel_Project$ ./run.sh
-------------------------------------------------Generation Size 100----------------------------------------------------------

parallel Version using Both MPI and openMP: time measured on my personal pc: 78.194725
parallel Version using MPI: time measured on my personal pc: 420.014280
-------------------------------------------------End ----------------------------------------------------------
```

It took a huge amount of time, most probably it is because of using 80 processors and also for the pure mpi it took 450 sec here we can see that it is not right because hybrid took 78 sec so maybe cores where busy while executing the pure mpi program, Thats why it is important to run it on grid'5000.

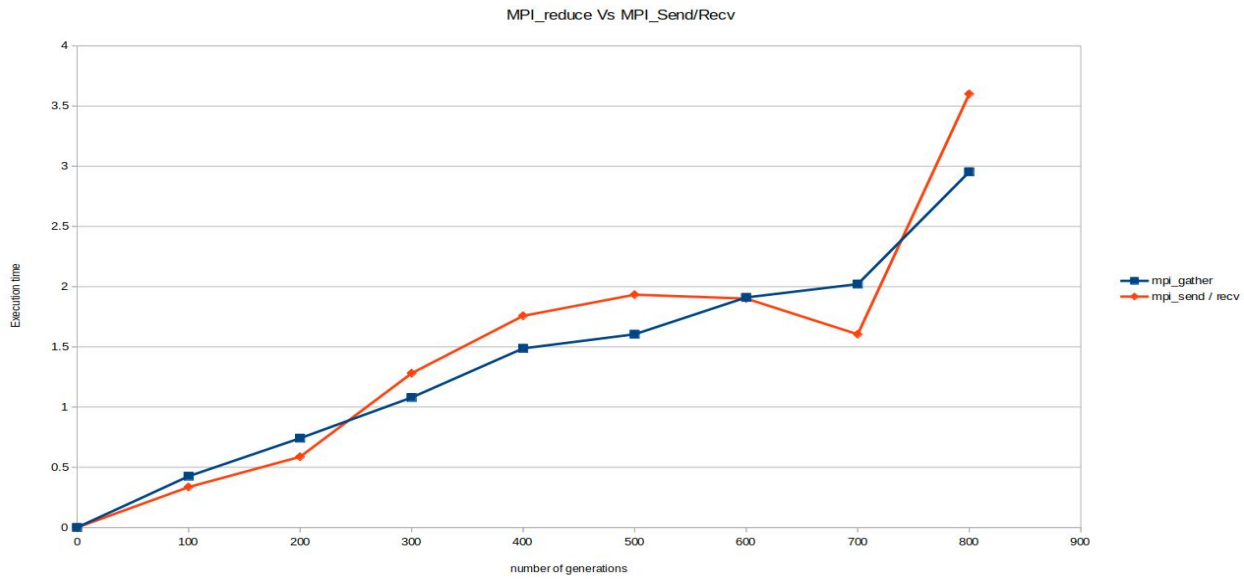
Evaluation Using collective communication Vs Send /Recv

To try to view the power of mpi_Gather vs mpi_send /recv i used every 5 generations to collect the sub population in the master as we can see in the figure below(same for gather every 5 generations).

```
while(i != NB_OF_GENERATION_PER_PROCESS)
{
  i++;
  selection(sub_population);
  crossoverV2(sub_population);
  mutation(sub_population);
  // fix_reduntancy(sub_population);
  calculate_population_fitness(sub_population);
  sort_population(sub_population ,  SUB_POPULATION_SIZE_PER_PROCESS);
  if(i%5 == 0){
    // printf("here");
    if(myID != 0 ){
      MPI_Send(sub_population, SUB_POPULATION_SIZE_PER_PROCESS, newChroType, 0, 1, MPI_COMM_WORLD);
    }
    else{
      for(int i  = 0 ; i < SUB_POPULATION_SIZE_PER_PROCESS ; i++){
        population[i] = sub_population[i];
      }
      for(int i = 1 ; i < numtasks ; i++){
        MPI_Recv(&population[i * SUB_POPULATION_SIZE_PER_PROCESS] , SUB_POPULATION_SIZE_PER_PROCESS , newChroType , i , 1,MPI_COMM_WORLD ,MPI_STATUS_IGNORE
      }
    }

  }
}
```

In the figure below we can see that the mpi_reduce is slightly better than using the normal send and receive in mpi. Also i checked and read a bit about mpi_Gather it is using mpi_send /recv is implemented on top of the mpi_send/recv.



Evaluation conclusion

So as we saw before the execution time of hybrid or pure mpi is way better than the sequential version, and it deserves to be done, while for hybrid vs pure mpi it wasn't as I was expecting, I expected the hybrid to be way more efficient than the pure mpi while the results shows that it is slightly better anyway it always depend on the size of the problem (bigger better).
We studied performance in terms of time execution. In terms of most fittest path as we increase the number of generations and number of chromosomes per population we will be more confident about the best path (the fittest one).
With small population size Hybrid is not good in terms of execution time, because we are creating threads while number of chromosomes in the population is small.

POPULATION_SIZE = 4 * 250 ; // 4 is for 4 machines to test it with highest number of machines increase 4. Example i tested it with 80 processors on grid. And in the scripts you can increase the number of generations.

How to run and Test

There are 3 versions sequential pure mpi hybrid versions. To make it easy to run i created scripts that runs the programs you just need to give the script permission using **chmod +x script.sh and then ./script.sh**
Programs:

1. Parallel_project_with_openMP.c (hybrid mpi and openMP using mpi_gather)
2. Parallel_project_without_openMP.c (pure mpi)
3. Serial_TSP.c (sequential)
4. Parallel_project_without_collectiveComm.c (without using collective communication gather use the normal mpi_send/rcv)

On local computer

For comparison between the 3 versions (pure MPI, Hybrid , Sequential)run the script (run.sh)
./run.sh
For comparison between the 2 versions of mpi (with and without collective communication mpi_gather) use runComm.sh

On Grid'5000

You need to reserve more than 4 machines because I'm using 80 processors (each machine is with 32 cores) using

- oarsub -I -t allow_classic_ssh -l nodes=4

You need to copy the files into grid (your site) you can use:

- scp * your_username@access.grid5000.fr:grenoble/project/
- And change the machinefile according to the machines you had reserved

After you can use runGrid.sh and runCommGrid.sh scripts.