Tools for High Performance Computing final project: Optimization using a parallel genetic algorithm: traveling salesman problem

Ville Jantunen

December 21, 2015

Contents

1 Introduction

Travelling salesman problem (TSP) is classical problem in a sense of numerical problem solving. The problem is to f nd the shortest route through a set of cities so that each city is visited exactly once and then returning to the starting city so that a closed path is formed.

In travelling salesman problem, cities can be represented as a set of coordinates. Distance between each pair of cities can be calculated from their coordinates. In this report we only concider f at two dimensonal situation.

In this report we use a genetic algorithm with heuristic breeding and parallel approach. In this approach we take multiple processes, create a dif erent population to each process. We evolve each population using certain rules where we breed f ttest routes and create new population by mutating their children. After evolving populations for some generations we copy the best individuals of each population to be passed to replace the worst individuals of the other populations.

2 Algorithms used

2.1 General formula

General formula for each process (except process 0).
1. Read data
2. Generate population
3.1 Sort population by route length
3.2 Breed best routes
3.3 Mutate children of best routes to f ll the population
4.1 Sort population by route length
4.2 Send best routes to process 0
5.1 Receive new routes from process 0

5.2 Replace worst routes of current population with the new ones.
Repeat 3,4,5 n-times. Repeat 3 inside this m-times.

2.2 Generating routes

In random generation I used Park–Miller random number generator. Each route is generated by randomly choosing one city at a time until a full route is ready.

2.3 Validation of routes and sorting

In order to decide which routes are let to breed and which routes are removed from the population I simply calculated the length of each route and sorted the routes. It is noteworthy to say that I didn't use the formula given in the final project-instructions as it is mathematically false ($(a+b)^2 \neq a^2+b^2$). Instead I took a square root from each member of the sum so that I actually summed the lengths of the distances between cities and not the squares of these distances. This is of course more costly in terms of cpu-time.

In sorting I used Rosetta-code (http://rosettacode.org/wiki/Quicksort) implementation of quicksort-algorithm. Quicksort algorithm works by setting a so called pivot-value that divides the list to two sections. Bigger members are sorted below the pivot and smaller ones are sorted abowe it. This is done recursively to smaller and smaller subsections until the list completely sorted.
Note: I could have implemented the quicksort myself but I didn't feel like reinventing the wheel would make this any better so I just used working code from elsewhere (in this case rosetta code).

2.4 Breeding

In breeding I used the algorithm given in the final project-instructions ie.

1. Take as the first city of the child the first one from either of the parents.
2. Choose as the second city of the child the one of the corresponding cities in the parents that is closer to the first city in the child.
3. If the new city is already included in the child choose the second city of the other parent. If this is also included in the child then choose the next city from either of the parents randomly (and in such a way that it does not introduce a cycle).
4. Go in a similar fashion through all the cities until the child has all cities.

2.5 Mutating

Mutation of the children was done by swapping two randomly chosen cities of the child with each other. This was done from zero to two times for each child.

3 Principles of parallelization

Parallelization was implemented using MPI(Message Passing Interface). Parallelization in this case was done by creating a random population of routes to each process except process 0 that is taking care of the data distribution and IO. These populations evolve independenty. After n-generations (n is decided

by the user) the best routes of each process are passed to process 0 that again distributes them to processes so that each process receives good routes from other processes. The worst routes of each population are now replaced with the routes form other processes.

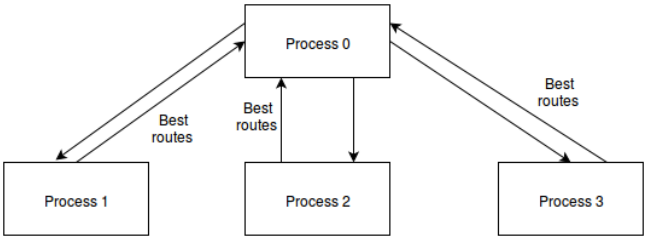

Figure 2: Parallelization scheme.

4 Code

4.1 length_measurement-module

Length measurement module contains subroutines and functions for calculating the length of a route and for loading data from file. Module also serves as a parameters list as floating point number type and character length parameters are declared. List of city coordinates and the number of cities are also (private) variables of this module. Number of cities can be accessed from other parts of the program using function get_city_count().

4.1.1 read_coordinates(filename)

Takes filename (character(len=clen)) as input. Read first line of input file as the number of cities (n) and then reads coordinates into a list (real(kind=rk), dimension(n,2)). Example of an input file:

```
4
12.3 15.6
13.6 45.7
44.1 67.2
43.4 44.4
```

Prints "Error in reading coordinates file" if there is an error in opening the file.

4.1.2 get_city_count()

Function return the number of cities that is read from the input file by the read_coordinates subroutine.

4.1.3 route_length(order,n)

Function route_length returns the length of the route (real(kind=rk)). The function takes the order (integer, dimension(n)) in which the city-coordinate list is walked. The other input parameter is the length of the order-list ie. the number of cities in the route.

4.1.4 distance2(city1,city2)

distance2 returns the squared distance two cities (integers corresponding to cities order number in coordinates list). Function is used in breeding the routes. Since in breeding section we are only interested in whether a distance is longer or shorter than another disntace and not about its actual value, we can use the squared value because $a^2 > b^2 \rightarrow a > b$ when a,b > 0. The distance between two points is always greater or equal to zero.

4.2 Breeding-module

Breeding-module contains most of the functionality of the program. It contains functions for breeding, mutating, passing good routes, and generating new routes. It also has the routes array (private integer array) routes(:,:) that contains all the routes in the population. It also contains an array [private integer array] routes_order(:) that contains the information about which route is shortest (for example integer 12 in first elements tells that route 12 is fastest. Integer 5 in second element tells that route 5 is the second fastest etc.).

4.2.1 init_routes

init_routes(number_of_cities, m) is a subroutine that needs to be run before generating routes as it allocates the array holding them. m is the number of routes in population and number_of_cities is the number of cities in one route. This also allocates the size of the array holding the information about which routes is fastest etc.

4.2.2 gen_route

gen_route(n) where n is the number of cities in a route, is a function that randomly generates and returns a route (integer array of dimension(n)).

4.2.3 gen_all_routes

gen_all_routes() subroutine fills the routes array with routes using the gen_route function.

4.2.4 sort_routes

sort_routes is a subroutine that uses Qsort subroutine to sort the routes_order array.

4.2.5 breed

breed(m,n) where m is the number of routes in the population and n is the number of cities. Sorts routes_order using sort_routes subroutine. After sorting, this chooses the best half of population and breeds them and then creates mutated copies of children using mutate function and so fills the population with new generation.

4.2.6 mutate

mutate(child, n) where child is a route (integer array) and n is the number of cities. Randomly chooses two cities from the route and swaps them with each other.

4.2.7 get_n_fastest

get_n_fastest(amount, n_of_cities, m_of_routes) returns the fastest routes (amount = how many). m_of_routes means the total number of routes in population and n_of_cities means the number of cities. Expects the routes_order to be in order (ie. sorted before).

4.2.8 replace_n_worst

replace_n_worst(better_ones, amount,n_of_cities,m_of_routes) m_of_routes means the total number of routes in population and n_of_cities means the number of cities. Replaces the worst routes (how many = amount) with better_ones (integer array, dimension(n_of_cities, amount)) Expects the routes_order to be in order (ie. sorted before).

4.2.9 get_next_city

get_next_city(route, n, this_city) return the next city of the route from this_city. This information is used for breeding. n is the number of cities in a route.

4.2.10 get_fastest

get_fastest(n) returns the length of the fastest route in the routes array.

4.3 randomgen-module

Randomgen-module is used for generating random floating point numbers [0,1] and random integers.

4.3.1 seed_pm

seed_pm(seed) requires integer (kind=8) seed to seed the random number generator.

4.3.2 rand_pm

rand_pm() returns a random floating point number in range [0,1].

4.3.3 rand_int

rand_int(min,max) returns a random integer in range [min,max] using rand_pm() function.

4.4 qsort_mod-module

qsort_mod.f95 consist of Rosetta code (http://rosettacode.org/wiki/Sorting_algorithms/Quicksort) implementation of quicksort sorting algorithm with very slight changes. It consist of a user derived datatype group, which consist of order number (integer) and value (floating point number) and recursive subroutine Qsort(a,na) where a is an array of type "group" and na is the length of the array. The subroutine sorts this list so that the smallest value is in the first element of the array.

5 Using the code

5.1 Compilation

There is a makefile included. Compilation is done simply by the command *make* in the src-folder. MPI-is used, so the compiler command used is mpif90 (basically gfortran using mpi).

5.2 Running the code

There are three optional command line arguments used by the code. The fisrt one is filename, second one is amount of rounds to run and the third one is the amoung of generations between mixing populations with other processes. Defaults are default_input.dat 100 10. Running is done by using *mpirun -np X ./salesman arg1 arg2 arg3*, where X is the number of processes (note that process 0 is taking care of the IO and data collection and distribution so X should be at least 2 and preferably more to make any sense.

5.2.1 Input-file

In input file in the first row there is the number of cities and next rows are two floating point numbers separated by whitespace. It is read in code by: read(file_unit,*), x, y. Example of the file could be:

```
4
12.3 15.6
13.6 45.7
44.1 67.2
43.4 44.4
```

I included a program that generates this kind of input-files. It's called gencoordinates.f95. It uses the module in randomgen.f95 and needs to be linked to this (gfortran gencoordinates.f95 randomgen.o).

6 Principles and results of benchmarking

For some reason I could not connect to Alcyone (network issues or something with the server) when I tried to run the benchmarking with higher amount of cores, so I was stuck with my laptops 4 cores. If I had been able to use Alcyone I would have done the same tests, just with more varying number of processes

In this case just measuring time of the process isn't sufficient as there is no ending condition except the amount of rounds run. I decided to plot the shortest route length as a function of time and investigate these. I used 2 different sets of cities and for the first city I used 2 different values for generations per cycle (ie. generations to calculate before mixing routes with other processes).

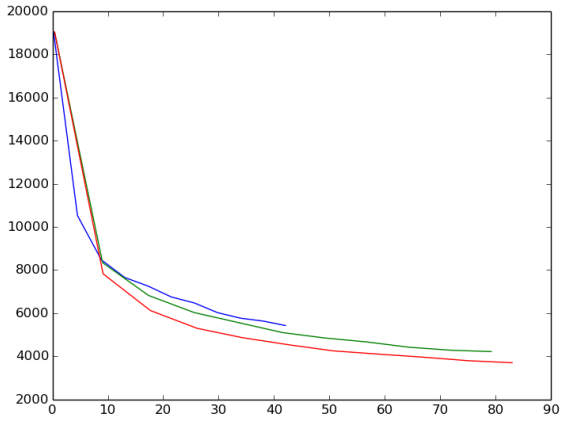

Figure 3: Y-axis is shortest route, X-axis is runtime in seconds. Red is 4-processes, green is 3-processes and blue is 2-processes. Running parameters were input1.dat, 300 and 20.

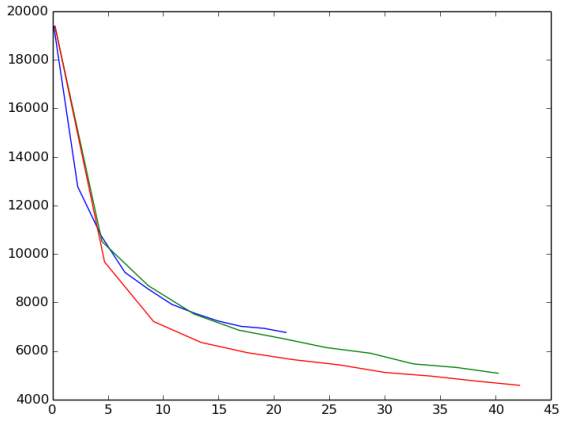

Figure 4: Y-axis is shortest route, X-axis is runtime in seconds. Red is 4-processes, green is 3-processes and blue is 2-processes. Running parameters were input1.dat, 300 and 10.

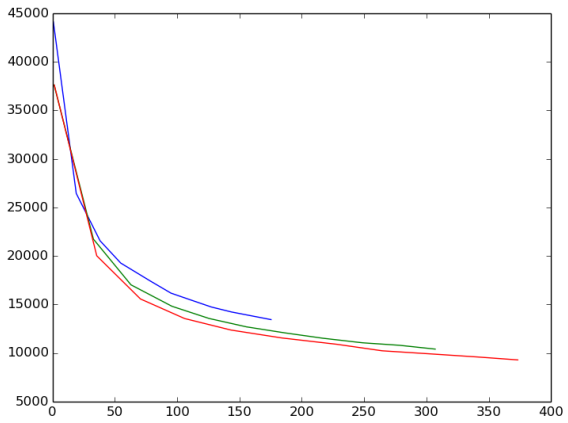

Figure 5: Y-axis is shortest route, X-axis is runtime in seconds. Red is 4-processes, green is 3-processes and blue is 2-processes. Running parameters were input2.dat, 300 and 20.

It seems that this scales pretty well at least with the limited amount of processors I was able to use (note that the number of population counting processes is one less than the number of total processes as the process 0 takes care of IO and data collection). As seen from the third data plot it took about 80 seconds for 4 processes to reach route length of 15000 compared to about 140 seconds for 2 processes. The problem wasn't embasasingly parallel as quite of lot of communication was happening so the scaling here is somewhat expected. More accurate results could be obtained with longer runs and numerical data-analysis.

7 Conclusions

From the instructions I wasn't really sure how the parallelization was meant to be done by in this particular problem so I decided to send everything to root processor and let the root do the distribution. I could have make the parallelization so that each processor sends data directly to its neighbour and not sending it to root inbetween.

As I was not able to do the benchmarking I wanted to It's hard to say about the scalability of this method. At least with parameters that I tested, this seemed to scale pretty well, but more tests with more cores and different sets of starting parameters are certainly needed.

Implementation could have been improved by thinking more about the breeding process and communication between processes. There was probably quite a lot of time wasted in communication between the processes as the processes waited the others to be ready before they started another cycle. To avoid this the first process that was ready with it cycle could have signaled the others to stop their cycles. This could have been implemented by creating a global variable checking its value after each cycle (generation). Amount of routes per population should also be varied as it might produce interesting results. There

could also have been a bit more randomization in which routes breed with which and what process gets which processes best individuals or for example take say four routes from each process and scramble them and pass randomly to processes.