Tools for High Performance Computing 2013

Final project

Juho Eskelinen

1. Travelling salesman problem

Travelling salesman problem (TSP) is notorious for being hard to solve exactly. In the standard version of the problem, the shortest route must be devised that goes through all cities in a region once and only once. Brute force search for the route has time complexity of O(N!), where N is the number of cities. Other algorithms also exist which improve the time complexity, but they still scale very badly with number of cities. Therefore approximation techniques are required especially for a larger problem set. For this project, a simple TSP is solved using a parallelized genetic algorithm for cities in a 2d cartesian space.

2. Algorithms

First cities' positions are generated in a 1.0 x 1.0 box. This makes route lengths correlate with number of cities for easier inspection of progress. Then a number of different populations of routes are generated. Each population includes a certain number of random routes visiting all cities.

After initialization the populations go through a number of generations. For each generation, every route in a population mates with a random partner from the same population. The child is generated as follows:

- First city is the first city of a random parent.
- After that the next city is chosen from the parent where the distance to it from the previous city is shorter.
- If one parent's city is included, the other parent's candidate is used. If both are already in the child, both parents are scanned until a city is found, which is not included. However if both candidates are valid at this point, one is chosen at random.
- After a child is created, there is a certain possibility that a mutation occurs. If so, two cities are swapped in the route at random.

Every few generations a migration occurs. When migrating, a few best routes are selected from each population and they are send to the two neighboring populations in a cyclical manner (so the first and the last communicate).

3. Parallelization

The program was parallelized using OpenMP. This makes it really easy to develop the parallel version in tandem with serial code. Conditional compilation is used when there are differences between the two versions. Timing of the program always uses fortran intrinsics, but the values are replaced when the OpenMP version is ran. Setting and printing number of threads used is also hidden in serial code with conditional compilation. Only region to be parallelized is creating a new generation. Each of the populations can be calculated independently and therefore that part is parallelized using parallel do. Each thread takes a population and makes a local copy of it. Child creation is then done on the local copy. When all children have been generated, the new population is copied over the original in shared memory.

4. Implementation

First a note on version control. As it was a part of the curriculum, a repository is included in the folder. Git was used and the repository can be accessed also via the URL https://github.com/bugi-/GA-TSP. The 'README' file is used by GitHub and gives basic information and instructions about the program.

A makefile is also included for easier compilation. Instructions are given in the next chapter.

Other supporting programs include a Python program for visualizing routes written to a file by the main program. There are also shell scripts for profiling the main program and counting the number of occurrences of the string ':)' in the source code. These additional programs should not be be strictly considered in grading but are included for completeness. They may also prove helpful and / or hilarious.

In the folder there is a file called 'preferences' which is used for input. Parameters for the run are read from the file.

All that remains are the Fortran files. Sizes.f90 is the module heavily used in the examples during the course. There are also 3 other modules: TSP_functions.f90, ga_functions.f90 and helper_functions.f90. In order, they include functions related to TSP, the genetic algorithm and general function usually used by other functions. The executable programs are main.f90 and test.f90. First one has the main program for solving TSP and the other has a suite of simple unit tests. The tests were used mainly during early development to make sure the underlying components work as expected. Later on tests for the complex cases were hard to generate, so testing relied on output from main.

Some things worth mentioning are:

- TSP_functions.f90 declares two custom types (see source for details).
- Both TSP_functions.f90 and ga_functions.f90 use module level variables which are set by the main program according to preferences.
- Main program outputs a file called main.out. This file includes coordinates of the cities in order for the shortest route in all populations during a certain point. This file is used as input for Python plotting as such.
- Some convenience functions are implemented in a very unoptimal way. They should not increase the runtime dramatically as linear searches dominate according to profiling.
- Output includes the length of the shortest route and its location in the populations. Also standard deviation of all routes is displayed as it can be used as a marker of convergence.

Also some things from README:

- Current implementation relies on a gnu extension so gfortran should be used. It is used to set the seed for intrinsic random_number with a single integer seeded generator.
- random_number used by this program is thread safe at least when compiled with GNU compilers

- Due to the indeterministic way the treads are scheduled by the CPU, the sequence of random numbers used by a certain population may change between runs and thus produce differing results.

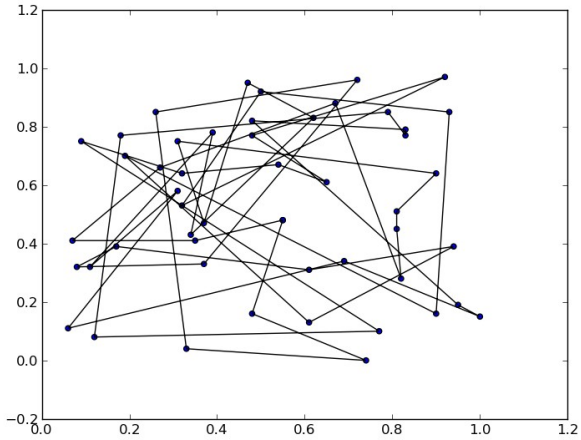
5. Instructions

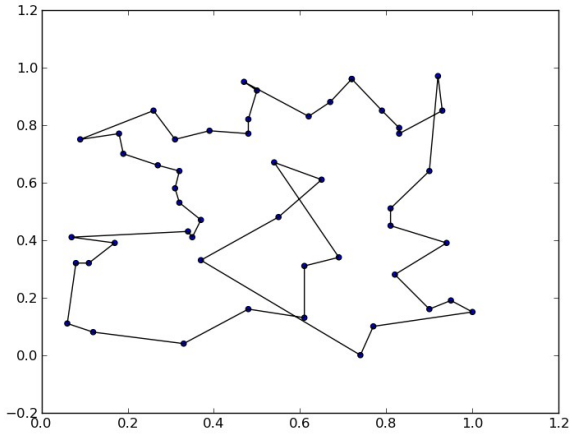
First compile the program. Use 'make' for serial version and 'make openmp' for the parallel one. The n set preferences in the according file. The types and meanings of each are explained in the file. The rows must not be swapped as they are read by position and common sense is required for setting the parameters as no checking is done. Run the program with './main'. If printing and writing are enabled, the program will print out e.g.:

```
Read preferences from file preferences
Printing every        100  generations.
Using          2 threads.

Starting...

Generation 0
Min length:   21.160999386454243      in population          1 index          12
Std dev:      1.3801201286088745

Generation         100
Min length:   6.4426043720023589      in population          2 index          6
Std dev:      0.89179480344535711

Time taken by program:  0.53015407799830427
```

Also the file 'main.out' will appear. The best routes at printing intervals can be

visualized with 'python plot_routes.py' if 'python' maps to a later Python 2.x and necessary packages are installed. Example of best route for 50 cities at the start and after 100 generations in below. Their lengths are 21.2 and 6.4 units respectively.





6. Scaling

Development was done on my local computer and after many tries the alcyone compiler is not working at all even on the login node. The problem was segfaults even with bounds checking enabled. This maybe at least in part due to older version of gfortran as it requires options that are now deprecated. Therefore no scaling test could be made on the cluster. Also tried mutteri but yet again the compiler is the barrier. Please update the compilers people! Tried out all the linux boxes I got into, but there was always a problem with segfaults or compiler incompatability. Maybe I am using something too fancy, but I'm not sure.

Finally got some results with a little help from a friend with 2 cores + hyperthreading (Intel on OSX). Sorry, the best I can do right now. Preferences are as in the same state as returned i.e. 50 cities with 4 populations of 50. Results are:

- 1 thread: 0.47633600234985352
- 2 threads: 0.29665589332580566
- 4 threads: 0.26627683639526367

So in this case HT gives negligible improvement, if any.

7. Conclusions

The genetic algorithm seems to provide good quality results relatively fast. No actual comparisons to exact methods are made but for the computing power they seem to do pretty well. A disadvantage is of course the increased memory usage for all the populations used. The question of optimizing the input parameters for a given number of cities remains open. There is room for improvement; heuristic for partner selection could be improved and maybe implement mutation only if it improves the route.