Tommi Mäklin

Optimization using a parallel genetic algorithm: traveling salesman problem

Tools for high performance computing 2020 final project report

Introduction to the problem

The travelling salesman problem (TSP) consists of finding a path that visits every node in some network exactly once and simultaneously minimizes the total length of the path. Under these constraints, the problem is a well-known example of an NP-hard problem — meaning, that there is no known polynomial time algorithm to solve the problem and an algorithm is not likely to be found. In practice, this means that an algorithm that finds the optimal path has a runtime that scales exponentially in the number of nodes in the network that need to be visited. For example, a greedy algorithm would need to calculate the path length for 2^N paths, where N is the number of nodes in the network. This solution becomes computationally intractable rather quickly when the number of nodes grows to more than a dozen or two.

Because finding the optimal solution to the TSP is obviously unfeasible, algorithms that attempt to find sufficiently good locally optimal solutions must be employed if the number of the nodes to visit is large. These types of algorithms are typically efficient in finding *some* solution to the problem that satisfies the constraints, and can be run multiple times in order to find many locally optimal solutions. The best solution among those found can then be chosen, which is often sufficiently close to the theoretical globally optimal solution. In this work, I will employ a genetic algorithm that has been adapted to the TSP and leverage massive parallelization to find the many candidate solutions.

Description of algorithms used

The genetic algorithm is based on imitating the evolution of biological organisms, which have evolved to solve the real-world problem of minimizing resource usage while maximizing reproductive success. This type of algorithm can be translated

into the optimization framework by considering some initial solutions to a problem which must evolve towards a better solution through some simple operations that mutate the solution and breed multiple solutions together to produce new solutions. If these operations are combined with selective pressure that filters out solutions which are, as defined by some fitness function, worse than others, then eventually a genetic algorithm incorporating these properties is expected to arrive at a solution that is significantly better than any of the initial solutions.

I've adapted the genetic algorithm to solve the TSP by initially seeding the population (set of potential solutions) with n randomly constructed paths through the network. These paths can then be mutated once with some mutation probability $p, p \in [0, 1)$ by swapping the positions of two distinct nodes in the path. In order to introduce more variation to populations that have stabilized to some set of solutions, I allow more subsequent mutations to occur in a single generation with the probability p^m, where m is the number of mutations. Mutation only happens in the children of two paths, which are obtained by breeding two solutions according to the heuristic described in the assignment. After breeding, the (Euclidean) length of all of the paths is calculated and a new population of size n selected from both the parents and the children by allowing the n most fit (shortest path lengths) individuals to survive and enter the next generation.

Since we wish to parallelize the algorithm, we have, in reality, multiple concurrent populations that are evolving simulations. In order to introduce more variability in the 'gene pool' of each population, I have introduced the possibility of a migration events happening in a generation with the migration probability, $q \in [0, 1]$. When a migration from one population happens, the migrating population randomly selects (weighted by the inverse lengths of the paths) e emigrants that are cloned and sent to another population, where they replace the e worst fit individuals. Migration events have the benefit of allowing populations to escape local optima if more fit individuals arrive, in addition to increasing the effectiveness of the breeding operations by adding more diversity in the set of potential solutions. In my solution, migration events are the first events to happen in a new generation. Note that the

population size in my implementation remains constant, meaning that clones of the emigrants are kept in the source population.

Principles of parallelization

I've parallelized the code to run on multiple computing nodes by using the message passing interface (MPI) for the C language. MPI divides the processing onto multiple tasks, where each task runs concurrently. In my implementation, each task has its own population of solutions to evolve in order to minimize the need to communicate between the tasks. Considering that the computation required in this implementation is rather minimal, there is no need to divide the population across multiple tasks, which would necessitate more communication between the tasks. Thus, the problem is 'embarrassingly parallel' in the sense that each thread runs what is essentially an independent realization of the algorithm, with the major exception of the migration events at the start of each generation, and pooling the results after each task has finished.

The migration events are implemented so that they occasionally happen at the beginning of a generation with the probability q. When a migration happens, the task containing the migrating population will first select the emigrants and then randomly select a neighboring task to receive them. In my implementation, the tasks are arranged in a circle where only neighbors (task id either one higher or lower) communicate. Periodic boundary conditions are present for tasks with the lowest and highest id, which communicate with each other. After selecting the receiving task, the task containing the emigrants calls MPI_Isend to perform a non-blocking send containing a 2D array consisting of the paths corresponding to the migrating solutions. Since the send is non-blocking, the task does not have to wait for the recipient to integrate the new solutions, and can instead continue to evolve its own population.

Receiving the immigrants is implemented by calling MPI_Iprobe to check for messages from other tasks that are waiting to be received. Checking for messages is performed in each generation after checking for whether a migration event happens

or not. Since MPI_Iprobe is non-blocking, the tasks do not need to wait for immigrants if none are coming, and can continue with their own populations. Note that the migration probability q only controls whether solutions emigrate to another task — immigrating solutions from other tasks are always received when they are ready. If an immigrating population is detected by MPI_Iprobe, the task will call the blocking receive method MPI_Recv and wait to receive the message before continuing to displace its worst-fit solutions with the immigrating solutions from the other task.

The migration procedure described above constitutes all communication between tasks during the runtime of the genetic algorithm. This means that the migration probability q and the size of the migrating population e directly control the amount of communication that is performed across tasks. With a lower migration probability the tasks will send fewer messages to each other and with a lower migration size the messages will be smaller in size. Considering that the communication between tasks is typically the bottleneck when running across multiple nodes, tuning these two parameters allows tailoring the algorithm to run on either a single or multiple computing nodes. I have implemented both parameters as command-line options.

At the end of the run when all tasks have finished, each task will extract the solution with the best fitness from its own population. These solutions are then gathered to the root process by calling the blocking method MPI_Gather, which gathers the fitness values from each task to the root process. The root process then checks which task had the best fitness overall, and announces the best task's identifier to all tasks using the blocking method MPI_Bcast. The task with the best solution is then instructed to print out its best solution to cout. After determining the winner in this fashion, the program finalizes the MPI part by calling MPI_Finalize, and then exits.

Since the vast majority (everything except receiving immigrants) of the communication during the runtime of the genetic algorithm is non-blocking and the number of times that the tasks need to communicate with each other is typically

low, the presented solution scales exceptionally well across multiple computing nodes. Should the migration probability or the migration size be increased, the scaling would become worse, but considering that migration is not the main mechanism driving the genetic algorithm, we can keep both parameters relatively small to maintain the performance. Scaling of the solution is examined in more detail later on.

Presentation of the code

The source code for the program is contained in its entirety in the 'tsp-mpi.c' file. The implementation consists of defining the necessary functions to read in the coordinates of the nodes in the network from an input file, calculating the fitness, defining the mutation, breed, selection, and immigration operations in the genetic algorithm, and the main routine which handles the MPI parallelization. Considering that the implementation is rather concise, I have opted not to break it down to multiple files or parts. A README file is also included with the code, which details basic compilation and run instructions.

The implementation uses only C features, the C standard library, and the MPI headers. I've implemented two custom MPI types to handle sending the coordinates of the nodes in the network and 'size_t' objects implemented in the header file 'mpi_size_t.h' included with the source code. The program can read multiple input parameters from the command line, which define the location of the file containing the 2D node coordinates control the genetic algorithm. The values of the input parameters to the genetic algorithm are validated before and the program will exit with a helpful error message if they are found wanting. The coordinates are supplied in a space-separated table, where each row contains the x coordinate in the first column and the y coordinate in the y column. I have included a Makefile with the project which handles compiling the source code.

Path lengths were calculated using 32-bit floating point numbers, and the paths themselves were stored using unsigned 16-bit integers. This limits the number of possible nodes to 65536, and also the accuracy of the operations somewhat, but

massively reduces the memory footprint of both the program and the messages that need to be passed between the tasks. Since there is some randomness in the source code and we do not want to arrive at identical solutions in each task, each task seeds their random generator differently by invoking srand from the C standard library with the sum of the current time and the id of the invoking task as the argument.

Instructions for using the code

The program is compiled by first loading the MPI environment using the commands appropriate for your computing system (e. g. 'module load iimpi' on the ukko2 cluster to load the Intel MPI implementation headers) , and then running the following commands from the root directory

```
cd src/
make
cd ..
```

The compiled executable can then be called by invoking mpirun with the appropriate number of tasks (16 in the example) from the root directory

```
mpirun -np 16 src/tsp-mpi run/input.dat 0.75 10000 10000 0.01 100
```

where the first parameter (`run/input.dat`) is the location of the node coordinates file, the second parameter (`0.75`) is the mutation probability, the third parameter (`10000`) the number of generations to run the algorithm for, the fourth parameter the population size (`10000`), the fifth parameter (`0.01`) the probability of a migration event, and the final sixth parameter (`100`) is the migration size or the number of emigrants.

When the executable is run, the tasks will silently perform their duties until finished. In the end, each task will print out the fitness of the best solution in its final population. When all tasks have finished, the program will determine the best performing solution across all tasks, and print both the fitness value and the path taken by the best solution to cout. It is possible to modify the source code by removing the comments around line 446 in 'tsp-mpi.c' to obtain periodical updates

printing out the best path and its fitness every 100 generations, but this behavior was disabled in the final source code after development.

Principles and results of benchmarking (scaling behaviour)

The program was benchmarked on an example input consisting of 20 nodes (cities) laid out in a $[0, 1]$ x $[0, 1]$ grid. The location of each node was determined by first sampling the x coordinate from a uniform distribution on the interval $(0, 1)$, and then sampling the y coordinate from the same distribution. Coordinates of the example input are included in the 'run/input.dat' file. The parameter values were set to 0.75 for the mutation probability and 0.01 for the migration probability with a population size of 10 000 and migration size of 100. The genetic algorithm was run for 10 000 iterations on each task. The layout of the cities, and the best path found across all benchmarking runs, is presented in Figure 1 in the results section.

Benchmarking was performed on the Department of Computer Science's ukko2 cluster, which is a high-performance computing cluster utilizing batch scheduling much like the computing clusters at the other departments. Computation was performed on the regular computing nodes, which contain two Intel Xeon E5-2680 v4 CPUs and 256 GB RAM each. The batch script was instructed to disable hyperthreading. Full technical specifications of the cluster are available online at https://wiki.helsinki.fi/display/it4sci/Technical+Specifications. The tsp-mpi program used in the benchmark was compiled by invoking 'mpicc' with the O3 optimization flag using the Intel MPI implementations, which was loaded with the 'module load iimpi' command. The specific command executed in the batch jobs was ($1 is the number of tasks)

```
mpirun -np $1 tsp-mpi-master/src/tsp-mpi tsp-mpi-master/run/input.dat \
                                         0.75 10000 10000 0.01 100
```

I benchmarked the scaling of the program in the number of tasks by running the program multiple times with 2, 4, 8, 16, 32, 64, and 128 tasks (cores). Running the program on the next power of two (256) was not possible due to constraints on the number of cores that can be reserved, and I was left satisfied with 128 tasks. Results of the scaling in terms of the number of tasks are presented in Table 1.

# Cores	# Nodes	Wall time taken (seconds)	CPU time utilized (seconds)	CPU efficiency (% utilized)	Max memory usage (MB)
2	1	338	671	99.26	18.47
4	1	342	1364	99.71	33.86
8	1	336	2664	99.11	70.99
16	1	341	5357	98.19	138.75
32	4	376	11978	99.55	986.59
64	4	389	24652	99.02	2038.54
128	5	365	45820	98.07	2279.42

Table 1 Results of the benchmarking tasks on the ukko2 cluster. The benchmark was run with the same input on 2, 4, 8, 16, 32, 64, and 128 cores (tasks). Number of cores is the number of physical cores reserved (hyperthreading was disabled with the option –hint=nomultithread), number of nodes the number of physical computing nodes, wall time taken the total runtime of the batch script in seconds, CPU time utilized the time the reserved cores were busy in seconds, CPU efficiency percentage of the time the cores were busy out of the total seconds reserved, and max memory usage the peak memory used concurrently across the nodes in MB.

The best route found among all runs was manually extracted from the outputs of the runs, and is displayed in Figure 1. The best route has the fitness value (total Euclidean length of the path) of 0.710519 and the same route, or the same route in inverse order, was consistently found across all runs except the run with only 2 cores reserved. An example output from running the program with 128 tasks is included in the ‘run/output.dat’ file, which contains the best path on its last line.

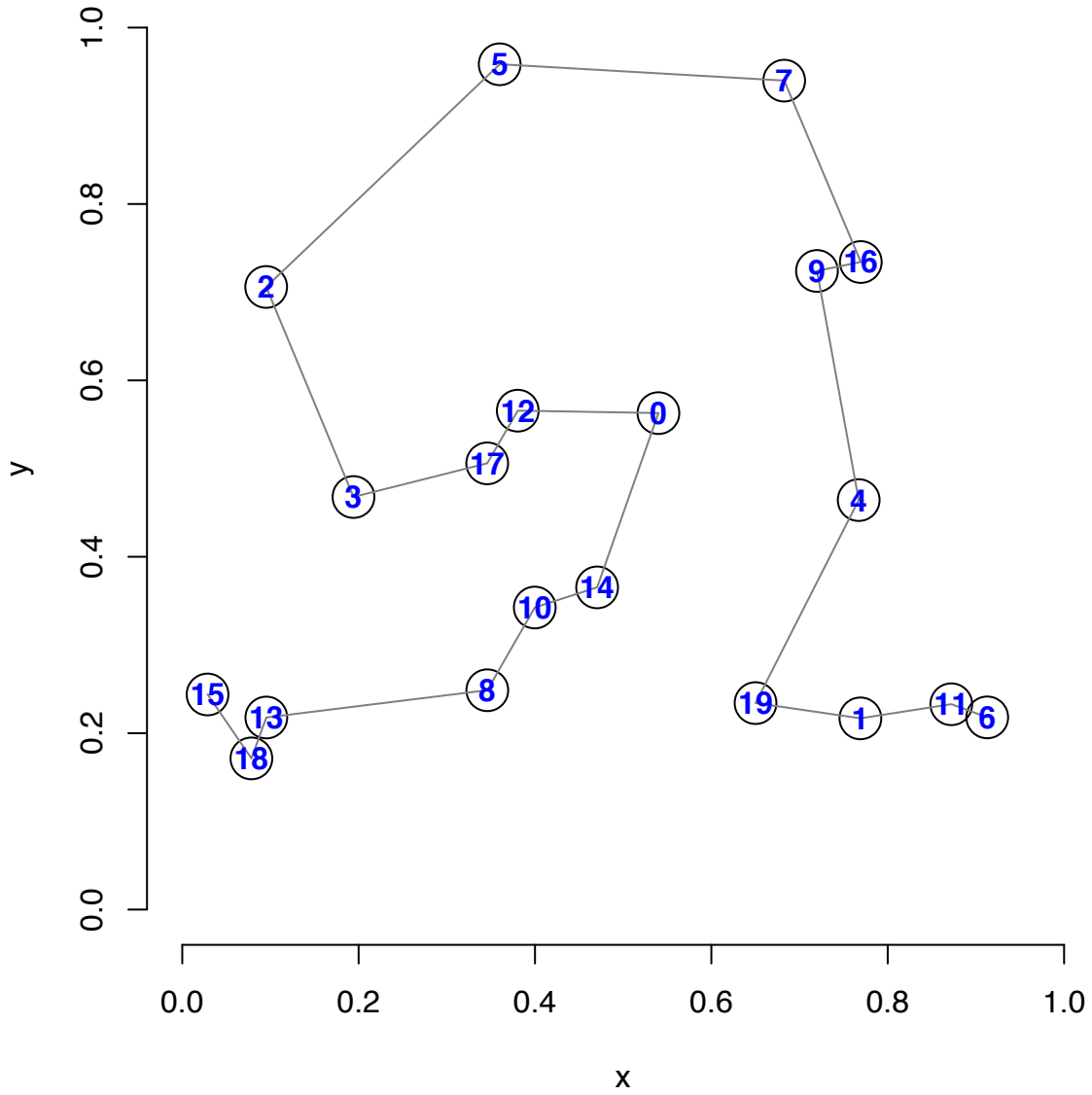

Figure 1 Best path found for traversing the example input. The coordinates of the nodes are available in the 'run/input.dat' file. The best path was extracted from the results of running the algorithm with 128 tasks, and the output from this run is included in the 'run/output.dat' file, where the last line contains the best path visualized in the figure as the gray line connecting the nodes. Nodes are labelled with their position in the input coordinates with the indexing starting from 0.

Conclusions

In this project report, I have presented a solution to the travelling salesman problem (TSP) using a genetic algorithm that has been parallelized with the message passing interface (MPI) to run on multiple computation nodes in a high-performance computing setting. The scaling of the solution was tested with up to 128 concurrent tasks running across multiple computation nodes on the

Department of Computer Science's ukko2 computing cluster. The program was implemented in pure C language using only the C standard library and the MPI headers. My solution has a nearly optimal linear scaling in the number of cores utilized, while keeping the wall time taken nearly the same for all runs up to the maximum test of 128 concurrent tasks. This fantastic scaling likely results from the minimal message passing that needs to be performed between the different tasks, as well as the use of non-blocking communication in the events where message passing is necessary.

Although the problem is embarrassingly parallel in the sense that the tasks rarely communicate with each other and the solution could easily be implemented with no communication at all, there some evidence in the benchmarking results that the migration events, which necessitate communication, are required for the best solution to be found. The benchmark that was run with only 2 cores, meaning only two initial starting populations and mixing only between these two, did not arrive at the better solution that was found by all other benchmarks. Since there was less mixing between the populations in the 2-core benchmark, this might indicate that having many different populations that exchange solutions is better than simply running the same number of iterations without the migration events. Thus, when the probability of a migration occurring and the number of individuals that emigrate is kept sufficiently small, it is worth including them despite the introduction of additional communication, even though we see some evidence in the impact of bandwidth starting to slow down the processing when running in the extreme setting with 128 cores (CPU utilization fell slightly to only 98%).

The presented algorithm could be further improved by adding genetic operators that affect more than just one node in the path. Introduction of these operators would increase the variability in the pool of possible solutions and possibly allow finding an even better solution. Additionally, the selection of the parents for breeding could be improved so that more fit solutions breed more often. Currently, the implementation pairs the solutions randomly by sampling the index of the mate for each solution, which could be changed to use the inverse of the fitness (lower

total path length is better, so the inverse is appropriate here for more weight to good solutions) as weights in the random sampling, which the selection of emigrating solutions already utilizes. While this may improve the breeding process, sampling with weights is much slower than uniform sampling and there is a tradeoff in weighted sampling significantly slowing down the calculations and the maximum population size that can be used. One way to get around this might be to further parallelize the individual *tasks* to use shared memory with a hybrid parallelization scheme using MPI + OpenMP or the threads library, so that the population of a single task is divided to multiple processors. Another option worth testing would be to use a better random generator than the C standard library's rand, which is notoriously bad at imitating randomness, to provide the different initial populations with more varied development paths when running with many cores.

In conclusion, the presented solution still performs adequately on the test input and scales exceedingly well to large numbers of cores distributed on multiple physical computing nodes. While I have presented some valid criticisms and improvements of my approach — and the code could, as always, be further improved — implementing the solutions is somewhat out of the scope for this assignment. Nevertheless, we can conclude from the results of this assignment and the benchmarks that MPI parallelization provides means to distribute workloads across a ridiculously large number of computing cores when the program is easily serializable and the amount of communication can be controlled. In the case of the genetic algorithm (applied to any problem), the presented approach of evolving multiple separate populations with migration between them could easily be adapted to efficiently solve other problems where the necessary genetic operators can be defined and finding the globally optimal solution is expensive or otherwise not straightforward.