The main goals of this assignment are

- To get acquainted with MPI
- To efficiently implement a simple parallel application

In this homework you will be required to implement the parallel version of the exact solution for Traveling Sales Person (TSP) problem.

**TSP**

The formal problem is defined as follows. One is given a fully connected graph (clique) where the nodes represent the cities and the edges represent the roads between them. Each edge has a weight greater than zero, which in our case will be assumed also to be finite. The weight represents the distance between each pair of the cities. We will assume that our graph is undirected, hence the distance from A to B is the same as the one from B to A. The problem is to find the shortest Hamiltonian cycle in the graph, namely a simple cycle (without self loops) which passes through all the nodes in the graph only once and has a minimal length. Note that in this assignment we limit our solution to metric TSP only, i.e. the triangular inequality holds. Hence, one should find the shortest traversal order for visiting all the cities only once, and returning to the city of departure (see http://en.wikipedia.org/wiki/Traveling_salesman_problem for more details).

**Branch-and-bound algorithm**

TSP is known to be NP-hard, and cannot be solved exactly without using some kind of branch-and-bound algorithm which traverses the search space of all possible solutions. Such a branch-and-bound algorithm contains the following two alternating stages: "branch" stage, where the new partial solution is expanded, and a "bound" stage, which attempts to predict the optimality of the partial solution without expanding it to the complete solution. If this partial solution can be ruled out upfront, one can prune some part of the search space without actually traversing it.

For example, the simplest algorithm is to systematically compare all possible routes which pass through all other nodes only once from some node A to itself, and pick the one with the smallest cost. Assume that you expanded a route consisting of \( k \)-1 cities. The branching step is to expand the route further by picking the next city \( k \). The bounding step is to always remember the best solution already found up to now, and to stop expanding the route if it is already longer than the best known solution.
The key to the efficiency of the branch-and-bound algorithm is the bounding heuristics which prunes as much of the search space as possible, or in other words is capable of generating tighter upper bound on the solution. However even the best heuristics may fail to prune the search space significantly, and then one has to go over the complete search space anyway.

**Branch-and-bound algorithm and master-worker parallelization**

The most intuitive way to implement the parallel algorithm is as follows. Split the search space into equally-sized chunks, one for each CPU in the system, and let each CPU compute its part independently of others. Then one of the CPUs aggregates all the partial solutions and chooses the best out of them. Let’s call such a solution static.

There are two problems with the static solution.

1. **Bad load balancing.** The chunks are formed with the underlying assumption of the worst case scenario, where each of the chunks must be traversed exhaustively. However this is not true in practice, since the search space is being pruned during the execution. However when splitting the search space before the execution, one cannot estimate the amount of the search space that will be pruned. So some chunks will be “hard”, and some will be “easy”, and consequently some CPUs will take much longer to complete while the others will stay idle. So the parallel efficiency of the implementation will be far from optimal.

2. **No bound propagation.** Since the CPUs do not communicate the best length each one has discovered in the process of execution, their bounding stage is based only on the local bound, which can be loose as compared to the minimum of all the local bounds on all the CPUs. Thus they could prune the search space more efficiently if they do exchange the local bounds.

The first problem can be solved as follows. We create special process (master) which splits the search space into a large number of sub-problems (much more than the number of the available CPUs), and maintains a queue of such sub-problems. All other CPUs (workers) request new work from the master, report back upon completion and request for more. Since the work is split into small chunks, the load imbalance is negligible.

The problem of bound propagation is solved by communicating the local bound to the other CPUs. This in turn involves

1. Having each worker sending the best local bound to all the nodes during the run.

2. Propagating the best bound available on the master to the worker when the new chunk is sent.

In the homework you will have to implement both versions, compare them, and investigate the influence of the chunk size on the performance.

**Technical specification**

You have to implement some version of the branch-and-bound algorithm (even the simplest one will do, but you are encouraged to read the Wikipedia article specified above for better bounds). You should parallelize the serial algorithm and implement it using MPI.

**Input format:**

You are provided with the coordinates of each node on the plane (x,y). For simplicity the weight of the edge between the cities $i$ and $j$ is computed as follows:

$$\text{abs}(x\text{Coord}[i] - x\text{Coord}[j]) + \text{abs}(y\text{Coord}[i] - y\text{Coord}[j])$$

For example, for a graph with three nodes, $A=(1,1)$, $B=(1,2)$, $C=(1,3)$, the distance between $A$ and $B$, $B$ and $C$ is 1, and the distance between $A$ and $C$ is 2.

You have to implement the function
int tsp_main(int nCities, int xCoord[], int yCoord[], int P[])

with the following parameters: nCities - number of cities to be visited, xCoord and yCoord - two
arrays with the X and Y coordinates of the cities, P - the indices of the cities. The function must return
the value of the shortest path. The function declaration is in tsp.h.

You should provide two different implementations of this function.

1. In file tsp_static.c: implement the simple static version where all nodes get equal chunks, and
communicate only in the beginning and in the end of the run

2. In file tsp.c: implement the dynamic version where all the nodes communicate the best bound and
the master maintains the queue of tasks.

The function must not call MPI_Init and MPI_Finalize. They are called before and after calling to
tsp_main.

The following files are provided on the course site:
main.c: Use it to check your version by changing the input arrays and the number of sites.
tsp_236370.o: Our MPI implementation of TSP. This implementation has the limitation of requiring
to be invoked with at least 2 processes, and accepts no less than 5 cities. Note that the order of the
traversal outputed by your program may differ from the one produced by ours. We supply this version
for your convenience. It has a bug, and may produce results worse than the optimal. However your
result must not be worse that the results of our version.

Makefile: Makefile to compile and run your version. Use the command “make tsp_236370” to compile our version, and “make run_236370” to compile and invoke it on 2 CPUs. Similar commands
can be used to compile and run your versions, just look at the make targets in the makefile.

Warning: while you can change the Makefile to increase the number of instances invoked by MPI
(-np argument), you MUST NOT omit -all-local argument for mpirun, since in the meantime we have
only available for the course. We hope to add more machines, and we will update the makefile if and
when that happens.

Warning 2: The problem is exponential in the input size. Thus it is easy to overload the computer.
Our implementation is capable of handling up to 18 cities within the reasonable time, and it is usually
enough to test your version. However in some cases 18 will be too much. Since the computer you will
be using is shared among half of the course

Part 1 (30%)

Implement a simple algorithm with static problem partitioning. In this version assume that the input is
available ONLY to the process with rank 0. In particular, nCities is NOT available to all processes, and
should also be distributed by rank.

1. Create two MPI_Datatypes: to send the data required for the computation by a given CPU and
another one for returning the result from CPU. Even if these can be achieved using the standard
MPI datatypes, you are required to create ones.

2. Use the most appropriate call from the collective communications to distribute the initial data to
all the CPUs

3. Use another the most appropriate call from the collective communications to get the results

4. All the results are to be returned to the process with rank 0, which selects the best one and returns
it to the caller of tsp_main.

5. You may not use Point-to-point communications, like MPI_Isend, only collective communica-
tions.
Part 2 (50%)

Implement the dynamic algorithm. In this version assume that the input is available to all the processes.

1. Process with rank 0 should serve as a master. It must split the problem into small sub-problems, we will call them jobs, and maintain the queue of the available sub-problems that are yet to be solved. If the program is called with a single process (-np 1), it must exit. It must also accumulate the results, update the local bound and make sure that any worker process asking for the job gets the new bound.

2. Processes with rank ≥ 1 should serve as workers. Workers fetch new jobs from the master, compute the result and report it back (in case it is better than the global minimum available to the slave), asking for more job. Also, when the workers find a solution which is better than the one they are aware of, they asynchronously update all others (including the master) with the new bound.

3. The master must signal the workers by assigning them “terminating” job. Workers must not exit before the master. You should use the appropriate call to collective communication to satisfy this condition.

4. You must use asynchronous communications where appropriate and avoid using collective communications if asynchronous design is more suitable.

5. The use of MPI_Send is prohibited.

Dry part (20%)

1. (10%) Draw the graph of speedup as a function of the number of processes you invoke (from 2 to 8). Try 3 different inputs for each number of processes and compute the average. For the static version - the comparison should be performed versus your implementation of the static case invoked with single CPU. For the dynamic version - compare versus your implementation of the dynamic case with single worker and one master. Do you expect to obtain super-linear speedup? If yes, is this the violation of Amdahl’s law? Does the speedup depend on the input? Explain.

2. (5%) Compare the performance of the static and dynamic versions and draw the graph of the ratio between the running times of the two versions as a function of the number of cities. Use at most 18 cities. Explain.

3. (5%) Dynamic version only. Run this version on 4 CPUs (-np 5) with as many cities as possible to complete the computations in 50 seconds and the size of each chunk roughly equivalent to the amount of work for ~1 second of single CPU. Now run it again for the half of size of each chunk, then decrease by half once again and run again. Draw the graph of the execution time as the function of the size of the chunk. Explain what you see.

Technical issues

- Your program is expected to run successfully on ds-fire0 in DSL. However you can develop on Windows (see the instructions on the course website).
- Read the manual on mpirun to understand how you invoke MPI programs. Test your program on your computer, but run it with many processes, as if it was invoked on many CPUs. We will test your program in a similar setup.
• In part 2. All the workers should finish TOGETHER with the master. Namely, if the worker decides that it has nothing to do, it should not EXIT, it should wait until ALL the workers and the master decide that there’s nothing to do and only then issue MPI_Finalize.

• You should fix the supplied Makefile to run with more processors (-np )

Grading policies
Below is a partial list of critical issues we will check and respective reduction of the grade

Correctness
• Incorrect computation result: -15
• Deadlock: -15

Implementation
• Use of MPI_Send: -20. (MPI_ISend is allowed)
• Part 1: Use of ANY point-to-point communication primitive is disallowed. Hense if used -20
• Part 2: Use of collective communications where it is clear that the asynchronous behavior is required -10

Documentation
• Internal documentation should be reasonably detailed to understand your code.
• External documentation with the explanation of your solution + the solution of the dry part.

Submission
Electronic submission
The submission will be allowed 2 days before the deadline and will close 5 hours after it. All later submissions should be sent to cdp.home.assignments@gmail.com.
   Electronic submission should be in the form of zip archive, which includes
   1. Text file IDS.txt with the names and IDs of the submitters, each on separate line. (submission is in pairs only)
   2. PDF/PS/DOC file of external documentation
   3. tsp.c and tsp_static.c files - everything is to be implemented in these files (no additional ones).

Printed version submission
You should submit:
   1. all the source code you wrote for this exercise
   2. external documentation

GOOD LUCK