Home Assignment 3
(CDP 2016-2017)

December 15, 2016

Deadline*: January 11 2017, 23:59*

The main goals of this assignment are

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

In this homework you will be required to implement the parallel version of the exact solution for Traveling Sales Person (TSP) problem.
NOTE: any question regarding the assignment should be sent to CDPhw236370@gmail.com

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 the exact solution may be too hard to find. Yet, some kind of branch-and-bound algorithm may improve the speed. Such an algorithm traverses the search space of all possible solutions and applies the following two alternating steps: "branch" stage, where the new partial solution is expanded, and "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 heuristic 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 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. Lets 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 chunk 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 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 having each worker sending the best local bound to all the nodes during the run.
Such solution is called dynamic.
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:

```
abs(xCoord[i]-xCoord[j])+abs(yCoord[i]-yCoord[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 citiesNum, int xCoord[], int yCoord[], int shortestPath[])
```

with the following parameters:
citiesNum - number of cities to be visited
xCoord and *yCoord* - two arrays with the X and Y coordinates of the cities
shortestPath - output: the indices of the cities in the shortest path.
The function must return the value of the shortest path **(If more than one shotrest path exists, you may**

return any one of them). The function is forward declared in *main.c*.
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 tsp_main.
The following files included in the provided archive:
main.c: Use it to check your version by changing the input arrays and the number of sites.
tsp_static.c: Where you should implement the static version.
tsp.c: Where you should implement the dynamic version.
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 produced by your program may differ from the one produced by ours. We supply this version for your convenience. It is slightly inaccurate**, and may produce results worse than the optimal.** However, your result must not be worse than 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: 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 a third of the course you should be considerate and avoid running large inputs.

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, citiesNum is **NOT** available to all processes, and should also be distributed **by rank 0**.

1. Use *the most appropriate* **point-to-point** set of calls to distribute the initial data to all the CPUs. **Do not continue** until all processes have received the list of cities, but make the data distribution as efficient as possible (using collective communications here might be more efficient, but you are required to achieve collective-like abilities using point-to-point calls. The use of MPI_Barrier is allowed as well).

2. Use *the most appropriate* **collective communication** call to gather the results.

3. 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.

4. Note: the way you distribute the work among the processes should not limit the number of cities your implementation supports. You will be limited if you encode complete paths as integers. You may encode path prefixes as integers, in which case you might get a slightly inaccurate division, but that should not be a problem here (think - why slightly unbalanced loads will not matter much?).

5. In order to prevent overloading on the servers, we will test the static implemention with up to 14 cities. Please make sure you do the same. Note that your program must finish within no more than 5 minutes, when using 8 processes.

Part 2 (70%)

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

1. Process with rank 0 serves 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. Master must also accumulate the results and update the local 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 that worker), and ask for a new job. If a worker finds a solution that is better than the one it is aware of, **it must asynchronously update all other workers with the new bound (the master should be updated only when asking for a new job)**
3. The jobs should be communicated using a dedicated MPI_Datatype (even if it could be done using regular types). It is up to you to define the content of that data type.
4. In order to terminate, the master must signal the workers by providing them a special termination job.
5. You must use asynchronous communications where appropriate and avoid using collective communications if asynchronous design is more suitable. It is important to make sure that the **memory requirements** of your implementation **are *O(1)* with respect to the number of MPI processes invoked.** Namely, the amount of memory each process uses should be linear to the number of cities, and should not depend on the number of processes (this requirement does not apply to buffers used for communication).
6. The use of **MPI_Send and MPI_ISend is prohibited - you should use all other variants of synchronous and asynchronous sends.** MPI_Send is optimized by the implementation, and in this assignment you should be able to justify the use of the send variant you've chosen according to the context of its use.
7. In order to prevent overloading on the servers, we will test the dynamic implemention with up to 18 cities. Please make sure you do the same. Note that your program must finish within no more than 5 minutes, when using 8 processes.

Technical issues

- Your program is expected to run successfully on **ds-bl20x** in DSL. However you can develop on your own machine, including Windows (see instructions on the course website). Note, the course staff will not assist you with Windows-based development. **You should develop in C** (C99 is supported).
- You may change the supplied Makefile to run with more processors (-np).
- If you have *printf*s in processes other than the master, their content might be buffered, and presented on the master's console later than you expect. Adding a call to *fflush(stdout)* could help better arrange the prints (though it might also slightly change the behavior of your program, as it might introduce unnecessary synchronization).

Grading policies

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

Correctness

- Incorrect computation result: -15 pts.
- Deadlock: -20 pts.

Implementation

- Use of MPI_Send or MPI_ISend: -20 pts.
- Part 1: Use of ANY collective communication primitive **for data distribution** and use of ANY point to point primitive **for results gathering** is disallowed. Hence if used: -20 pts.
- Part 1 + 2: Having the implementation limit the maximum number of cities, even if the limit is high: -10 pts.
- Part 2: Not using MPI_Datatype for job distribution: -20 pts.
- Part 2: Use of blocking calls where it is clear that the asynchronous behavior is required: -10 pts.

Documentation

- Internal documentation should be *reasonably* detailed to understand your code. Briefly comment MPI calls, and justify the selection of the call variant used.
- External documentation with the explanation of your solution.

Submission

Electronic submission

The submission will be electronic only, and will be closed 7 days after the deadline.
Electronic submission should be in the form of *zip* archive named *<ID1>_<ID2>.zip*, which includes:

1. A single PDF file named *<ID1>-<ID2>.pdf* with the following content:
   (a) Your names, IDs and email addresses.
   (b) The external documentation.
2. **tsp.c and tsp_static.c files - everything is to be implemented in these files (no additional ones).**

GOOD LUCK :)