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 Graph Partitioning problem.

**The Graph Partitioning problem**

The Graph Partitioning problem describes situations, where it is necessary to minimize the weight (or number) of connections between two parts of a finite network.

The formal problem is defined as follows. Given weighted, undirected graph $G$ with vertex set $V$ and edge set $E$, and cost function $c : E \to \mathbb{N}$. The target is to partition $V$ into two disjoint subsets $V_1$ and $V_2$ of equal size such that the sum of costs of edges connecting vertices belonging to different subsets is as small as possible.

In Figure 1 you can see an instance of (undirected) Graph Partitioning problem. In this assignment all graphs will be directed and all edges will have a positive weight.

![Graph Partitioning Problem](image.png)

**Figure 1:** A graph partitioning problem and a feasible solution
Branch-and-bound algorithm

Graph Partitioning 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 partitions of the vertexes in to two equal sets and pick the one with the smallest cost. Assume that you have partitioned $k-1$ nodes. The branching step is to expand the partitioning further by choosing where to put node $k$. The bounding step is to always remember the best solution already found up to now, and to stop expanding the partitioning if it is already costs more 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

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.

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 a version of the branch-and-bound algorithm. You should parallelize the serial algorithm and implement it using MPI.

Input format: You are provided with the graph adjacency matrix, where the edges are specified with positive integers, and 0 specifies no edge.

For example this is adjacency matrix for graph given in Figure 2:

\[
\begin{pmatrix}
A & B & C \\
0 & 0 & 4 \\
5 & 0 & 0 \\
2 & 8 & 0 \\
\end{pmatrix}
\]

Note: You can assume the number of vertices in the graph is even, and the number of CPUs is less than \(\frac{\text{#vertices}}{2}\).

Figure 2: Directed graph with positive integers edge weights

Function to be implemented: \(\text{int graph\_partition(int verticesNum, int adjMatrix[[]], int partition[])}\)

\(\text{verticesNum}\) - number of vertices in the graph

\(\text{adjMatrix}\) - graph adjacency matrix, of size \(\text{verticesNum} \times \text{verticesNum}\)

\(\text{partition}\) - output: for vertex \(i\) in the graph \(\text{partition}[i]\) holds 0 if this vertex is in group 0, and 1 if it is in group 1.

The function must return the value of the minimal partition. The function is forward declared in \(\text{main.c}\).

You should provide two different implementations of this function.

1. In file \(\text{graph\_partition\_static.c}\): implement the simple static version where all CPUs get equal chunks, and communicate only in the beginning and in the end of the run

2. In file \(\text{graph\_partition.c}\): implement the dynamic version where all the CPUs communicate the best bound and the master maintains the queue of tasks.

The function must not call \(\text{MPI\_Init}\) and \(\text{MPI\_Finalize}\). They are called before and after calling \(\text{graph\_partition}\) function.
The following files included in the provided archive:

main.c : Use it to check your version by changing the adjacency matrix and the number of vertices.
graph_partition_static.c : Where you should implement the static version.
graph_partition.c : Where you should implement the dynamic version.
graph_partition_237370.o : Our MPI implementation of Graph Partitioning problem. Note that the partition produced by your program may differ from the one produced by ours, however, your result must not be worse than the results of our version. We supply this version for your convenience.

Makefile : Makefile to compile and run your version. Use the command make graph_partition_236370 to compile our version, and "make run_graph_partition_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 32 vertices within the reasonable time, and it is usually enough to test your version. However in some cases 32 will be too much. Since the computer you will be using is shared with other couples from the course, you should be considerate and avoid running large inputs. The static solution will be checked with up to 24 vertices input. The dynamic solution should finish up to 32 vertices. Both must finish within no more than 2 minutes, when using 8 processes.

Part 1 (30%)

Implement a simple algorithm with static approach. In this version assume that the input is available ONLY to the process with rank 0. In particular verticesNum 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 initial input.
2. Use the most appropriate collective communication call to gather the results.
3. All the results are to be returned to rank 0, which selects the best one and returns it to the caller of graph_partition.

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, vertices (including the master) with the new bound.
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. All processes should exit together. Namely, a worker must not terminate immediately after getting a termination job, if some other worker is still calculating a regular job. You should figure out the most efficient way to synchronize the processes until all workers have received the termination job.
6. The use of MPI_Send and MPI_ISend are 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.
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 *printfs* 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
- Deadlock: -20

Implementation

- Use of MPI_Send or MPI_Isend: -20.
- 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
- Part 2: Not using MPI_Datatype for job distribution: -20
- Part 2: Use of blocking calls where it is clear that the asynchronous behavior is required: -10

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, which includes:

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

**GOOD LUCK!**