Compulsory Assignment 2 in INF236
Spring semester 2024

Due date: Your solution should be bundled as zip file and handed in through mittuib no later than midnight Friday 22th of March.

Your solution should contain one nicely formated pdf report containing all necessary text. This should also list the names and purpose of your program files. Each program should be contained in one separate file. It is advisable to display results using both tables and graphs.

In this assignment you are going to implement and test a parallel algorithm for computing a breadth first search on a graph.

All algorithms should be implemented in C or C++ and parallelized using OpenMP. The final running of the programs should be done on brake.ii.uib.no. Do not use more than 80 threads for any run. For the final run perform each experiment at least three times and only use the best timing.

Individual work: You may talk and discuss the assignment with you fellow students, but you *must* do the programming yourself. If you copy code from someone both you and the person you copy from will automatically fail the course. The same is also true if you copy from the internet.

A *breadth first search (BFS)* in a graph $G(V, E)$ consists of traversing the edges in G from a given starting point $s \in V$ such that each vertex is visited according to its minimal distance to s. The edges used for this traversal makes up a BFS tree. Note that a BFS tree is not unique, but the minimum distance from each vertex to s is.

In the directory `/export/data/fredrikm/inf236/ob2` on brake.ii.uib.no you will find a program for reading graph files and for running the algorithms you are to solve in this exercise. Copy all files in this directory to your own directory. The main program is `driver.c`. This is compiled as follows.

```
gcc driver.c -fopenmp -lnuma -O3
```

To run the program you must give a file name as argument. This file should contain the following information:

Line 1: The number of graphs to run on. Line 2: The number of times each algorithm should be run. Line 3: The number of thread configurations, followed by the number of threads in each configuration. Line 4 and onwards: The path to each of the graphs.

As an example consider the file `data_files` which contains:

```
4
1
6 1 10 20 30 40 50
/export/data/fredrikm/inf236/graphs/road_usa.mtx
/export/data/fredrikm/inf236/graphs/delaunay_n24.mtx
/export/data/fredrikm/inf236/graphs/hugebubbles-00020.mtx
/export/data/fredrikm/inf236/graphs/rgg_n_2_22_s0.mtx
```

If you execute `./a.out data_files` then this will run the program on the specified files, each algorithm is executed once and any parallel algorithm will be run on 6 configurations, using 1, 10, 20, 30, 40 and 50 threads. Timings and result from any algorithms will be printed to the screen. The results are also printed to a file `results.m`. This file is in a format that can be read in to Matlab.

The current version of the program contains a method `sbfs()` (in the file `sbfs.c`) for performing a sequential BFS. Your main task will be to write parallel versions of this in the files `pbfs.c` and `abfs.c`. These files are available but do not contain any code.

Initially you should look through the program `driver.c` and `sbfs.c` to familiarize yourself with the setup and the sequential algorithm. As you can see you can turn algorithms on and off by setting an associated boolean variable in `driver.c`. Currently the sequential algorithm `sbfs()` is turned on, while the parallel algorithms `pbfs()` and `abfs` are turned off.

The format of graph files are as follows. First there is a line giving some internal information about the format. The second line contains three numbers.

These are the number of vertices given twice and then the number of unique edges. Finally, the endpoints of each edge is given on a separate line. The order in which the vertices for an edge is given is not significant. Note that the numbering of vertices starts from 1. This is also true in the program. The following is an example of a graph with 4 vertices and 3 edges.

```
%%MatrixMarket matrix coordinate pattern symmetric
4 4 3
1 2
2 3
3 4
```

If you write your own graph files then remember to keep the first line unchanged.

The assignment

Your main task is going to be to create two parallel BFS implementations along the following lines.

In both algorithms the exploration from a particular layer in the BFS should be done in parallel. Thus if the algorithm has discovered a set T of vertices at distance d from the starting point s, then the exploration from vertices in T with the objective to discover vertices at distance $d+1$, should be shared among the available threads.

In the first parallel version, `pbfs`, when the threads are exploring from vertices at distance d, each thread should first store locally all the new vertices it discovers at distance $d+1$.

Once all vertices at distance $d+1$ have been discovered, these should be placed in a common array S, before starting the search again. During this search it is possible that two or more threads discover the same vertex and places it in its own local list of discovered vertices. You can ignore checking for this as it should have no impact on the correctness of the algorithm and also little impact on the overall running time.

When performing the BFS, the program should create two lists, one containing the parent pointer for each vertex, (i.e. giving the edges used for the traversal) and one containing the minimum distance from each vertex to the starting vertex. This is similar to what is done in the sequential code `sbfs`. When `pbfs` exits, the distance values will be compared against the values computed from `sbfs` and if there are any discrepancies, the first one will be printed.

In the second parallel algorithm, `abfs`, you should also perform a parallel BFS. The main difference is that this algorithm should start out by sequentially executing the BFS algorithm. Then at some point it switches to a parallel scheme similar to `pbfs`. But now the discovered vertices are only copied back to the shared array S every kth iteration. That is, if the threads start searching for vertices at distance d from s, then only after finding vertices at distance $d+k$ should the vertices be copied back to S. In between this, each thread stores its discovered vertices locally. Thus during these rounds each thread continues searching from the vertices it has discovered by itself. This continues until no thread has any more vertices to explore. Part of the assignment is now to determine when it is best to switch from sequential to parallel, and also to determine the best value for k.

When you develop these algorithms you should experiment with different strategies and compare and document your results. The objective should always be to come up with the fastest possible algorithm. It is possible to allocate local memory in `pbfs` and `abfs` as needed, but do not change the signature of the methods.

Some hints.

- Note that `pbfs()` and `abfs()` are already executed within a parallel setting. Thus there is no need to use `#pragma omp parallel` inside these methods.
- All the parameters passed to `pbfs()` and `abfs` are shared. Thus these can be used for communication between the threads. They can also be used for storing values for each thread as long as one makes sure that each thread writes in a different area of each array.
- If you wish to alter the output in the file `solution.m` then this can be done in the routines `store_value()` and `store_p_value()` in `driver.c`.
- Remember that if you are using any type of collective operation such as for instance `#pragma omp barrier`, then all threads must execute this for the program to work.

There are four large graphs in the directory `/export/data/fredrikm/inf236/graphs` on brake.ii.uib.no that you should use for your final tests. Note that as these are large you should not copy these to your own directory, but instead specify the full path to graphs.

In addition there are a number of smaller graphs available in the directory `/export/data/fredrikm/inf236/smallGraphs` that you can use.

Divide your final report in the following way. First there should be an introduction where you define the problem you are solving. Next, you should outline the algorithms you have developed. This is then followed by an "Experiments" section, and finally a conclusion. The experiments should include timings and speedup results, preferably both as tables and plots.