Bilkent University
Department of Computer Engineering
CS 525
Fall 2020

PROJECT 2: REPORT

Ricardo Román-Brenes, 22001125
ricardo@bilkent.edu.tr

Part A: Serial BFS

Implementation of the serial version follows the classic approach, using a queue to keep track of the navigated vertices, and two arrays to keep track of the level being processed and which vertex has been visited.

```
bfs(csr *g, int src)
{
    queue q;
    enqueue(q, src);
    seen[src] = TRUE;
    dist[src] = 0;

    while (!is_empty(q))
    {
        i = dequeue(q);

        neighbors_of_i;
        n = get_neighbors(g, i, neighbors_of_i);

        for (int j = 0; j < n; j++)
        {
            v = neighbors_of_i[j];
            if (!seen[v] && v != i)
            {
                enqueue(q, v);
                seen[v] = TRUE;
                dist[v] = dist[i] + 1;
            }
        }
    }
    return dist;
}
```

Another important function here is `get_neighbors(g, src, neighbors)`, which, given a graph `g` and a source vertex `src`, fills `neighbors` with the adjacent vertices of `i`, and returns the number of them.

```
get_neighbors(g, src, neighbors)
{
    n = 0;
    alpha = g->vertex[src];
    omega = g->vertex[src + 1];

    if (alpha == omega) // no neighbors
    {
        neighbors = null;
        n = 0;
    }
    else
    {
        n = omega - alpha;
        for (nli = 0, eli = g->vertex[src]; nli < n; nli++, eli++)
        {
            neighbors[nli] = g->edges[eli];
        }
    }
    return n;
}
```

Performance-wise, the program is quite fast, having no problem correctly computing the distances of a fully connected graph of 1000 vertices in less than `0.84s`, averaged over five trials). Using the examples provided by the assignment, the time taken was less than `0.002s`.

Part B: Parallel BFS

Unfortunately, the parallel version could not be completed on time. The program does a correct split on the data but gets hanged on a call to `MPI_AllReduce` for unknown reasons.

As shown in the assignment split, the vertex should be split as evenly as possible among all processes. The data split implemented divides the vertices trying to assign one vertex per process and the remaining ones are taken by process 0. Log files have been created for the most important operations, such as this split. These logs, shown below, emphasize how that is achieved.

0.log

p2 > 0.log

```
5
6   [0|3]:> numVertex > 4
7   [0] 0   [1] 2   [2] 4   [3] 5
8
9   [0|3]:> numEdges > 7
10  [0] 1   [1] 3   [2] 4   [3] 6   [4] 5   [5] 0   [6] 2
```

1.log

p2 > 1.log

```
3
4   [1|3]:> numVertex > 1
5   [0] 7
6
7   [1|3]:> numEdges > 0
8
```

2.log

p2 > 2.log

```
3
4   [2|3]:> numVertex > 1
5   [0] 7
6
7   [2|3]:> numEdges > 1
8   [0] 2
```

3.log

p2 > 3.log

```
3
4   [3|3]:> numVertex > 1
5   [0] 8
6
7   [3|3]:> numEdges > 4
8   [0] 2   [1] 3   [2] 4   [3] 5
```

In this case, there are seven vertices in the graph from the assignment, so, as best as could be done, each process takes one, then the remaining three are taken by process 0. This can also be visualized on the standard output, as shown below.

TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE

```
mpirun -np 4 ./parallel_bfs data.txt 0 1 out.txt
P0: local_number_of_vertices=4
P0:     vertices[0]=0
P0:     vertices[1]=2
P0:     vertices[2]=4
P0:     vertices[3]=5
P0: local_number_of_edges=7
P0:     edges[0]=1
P0:     edges[1]=3
P0:     edges[2]=4
P0:     edges[3]=6
P0:     edges[4]=5
P0:     edges[5]=0
P0:     edges[6]=2
P1: local_number_of_vertices=1
P1:     vertices[0]=7
P1: local_number_of_edges=0
P3: local_number_of_vertices=1
P3:     vertices[0]=8
P3: local_number_of_edges=4
P3:     edges[0]=2
P3:     edges[1]=3
P3:     edges[2]=4
P3:     edges[3]=5
P2: local_number_of_vertices=1
P2:     vertices[0]=7
P2: local_number_of_edges=1
P2:     edges[0]=2
```

After performing the partition, and as said before, a problem was encountered where the program hangs performing an `MPI_AllReduce`. An attached GDB debugger to the processes involved shows that all but one process are in idle wait. The remaining one is stuck doing a function named `mca_btl_vader_component_progress()`. This function is an inside functionality of OpenMPI and was untraceable. There is a bug report concerning it, which can be viewed at `https://github.com/open-mpi/ompi/issues/5842`.

Below are the traces of the 4 processes used. On the bottom right section, on the pink rectangle is the mentioned behaviour.

Despite the problems it is possible to hypothesize how the parallel implementation would have behaved. The amount of work performed by each process is little, check a few indexes on a couple of arrays, perform a small computation and done, yet the amount of communication needed to do that computation far outweighs it. An `MPI_Alltoall` is a costly bottleneck call, network-wise (or memory in this case), it must go through all the communication stack of the kernel and back. It is an educated guess that in a single machine with multiple processes, the parallel version would be much slower than the sequential one. On a computer cluster, it would probably be even slower since network access is even more costly than memory access.

One way that this version could be improved is by allowing each process to explore its own vertices independently before performing communications. This would improve the computation-to-communication ratio.