Parallel and Distributed Algorithms and Programs
TP n°3 - Parallel Breadth First Search

Oguz Kaya
oguz.kaya@ens-lyon.fr

Pierre Pradic
pierre.pradic@ens-lyon.fr

14/11/2016

Breadth first search (BFS) is one of the fundamental graph algorithms with many application domains. Computing the BFS of a given graph efficiently is vital in attacking and solving these real-world problems. In this TP, we will precisely be doing this, by practicing some fancy MPI routines that we have not used before. The sequential version of the BFS algorithm we will implement is given in Algorithm 1.

Algorithm 1 Breadth first search on a graph

Input: $G(V, E)$: A graph with the set V of vertices and E of edges
$root$: Root vertex from which the BFS should start
Output: $depth$: An array of size |V| indicating the BFS distance of each vertex from the root (-1 for unreachable vertices)

```
depth ← [−1, ..., −1]                          ▶ Initialize to -1.
depth[root] ← 0                                 ▶ Root has a depth of 0
while there are new visited vertices do
  for each vertex v_i ∈ V do
    if depth[i] ≠ −1 then                       ▶ The vertex v_i is already visited
      continue
    for each vertex v_j ∈ adj(v_i) do
      if depth[j] ≠ −1 then                     ▶ The vertex has the visited neighbor v_j
        depth[i] = depth[j] + 1
return depth
```

As usual, you are given a skeleton code `bfs-skeleton.c` and a solution file `bfs-solution.c` that you need to fill out with your implementation. Now, open up the solution file; you will see an empty function with the signature

```
void bfs(int N, int *adjBeg, int *adj, int *depth, int root)
```

whose parameters are explained as follows:

- **N**: The number of vertices in the graph.
- **adj**: The list including the the indices of adjacent vertices.
- **adjBeg**: The pointer indicating the beginning of the list of adjacent vertices for each vertex. These pointers are consecutive; the beginning pointer for the vertex v_{j+1} is the end pointer for the vertex v_j. For example, for the vertex v_j, its adjacent vertices are those with indices $adj[adjBeg[j]], adj[adjBeg[j]+1], \ldots, adj[adjBeg[j+1]-1]$. Similarly, the list of neighbors for the vertex v_{j+1} contains vertices with index $adj[adjBeg[j+1]], adj[adjBeg[j+1]+1], \ldots, adj[adjBeg[j+2]-1]$, and so on.
- **depth**: The array of size N to be filled with the depth of each vertex from the root. You may assume that the array is allocated, all its elements are initialized to -1, and each MPI process has a copy of the array.
- **root**: The index of the root vertex from which the BFS traversal begins.

You can assume that each MPI rank has the same initial copy of the graph data (`N, adjBeg, adj`) and the `depth` array.

To create a random graph, we provide a script named `create-graph.py` that takes the number of vertices $|V|$ in the graph as the first argument, the number of edges in the graph as the second argument, and the name of the output file for the graph as the third argument. Now, try to create a sample graph by typing

```
./create-graph.py 10 40 graph.txt.
```

This will create a graph with 10 vertices and 40 edges, and write it in the file `graph.txt`.

Compile the code, and run executable `bfs` by typing

```
smpicc bfs-skeleton.c -o bfs
smpirun -hostfile ... -platform ... -np [num-procs] ./bfs [graph-file-name] [bfs-root-node] .
```

by providing an appropriate host file, platform file, graph file name, and node index to start the BFS from.

Part 1

Parallel Breadth First Search

Question 1

a) Parallelize the BFS iterations given in Algorithm 1 with P MPI processes so that each process updates the `depth` of $\lceil N/P \rceil$ vertices with consecutive indices. For instance, the process 0 will update the depths of the vertices $v_0, \ldots, v_{\lceil N/P \rceil - 1}$, the process 1 will update the depths of $v_{\lceil N/P \rceil}, \ldots, v_{2\lceil N/P \rceil - 1}$, and so on (except the last process, who will update for the last $(N - (N-1)\lceil N/P \rceil)$ vertices).

Question 2

a) At the end of each BFS iteration, we need to make sure that each process has the new value of the entire `depth` array. Note that each process (except the last!) updates a contiguous set of $\lceil N/P \rceil$ elements of the `depth` array. Try to perform this communication using MPI_Gatherv, followed by MPI_Bcast.

b) In MPI, there is a dedicated function to exactly handle this type of communication, which is called MPI_Allgatherv. Look to the MPI documentation, understand how it works, and use this routine to communicate the entries of the `depth` array.

`HINT:` When searching, look for how to use it with MPI_IN_PLACE argument so that you do not have to specify the send buffers.

Question 3

a) Now that you have a working implementation, it is time to benchmark it! Create a large graph having 10000 vertices and 100000 edges. Try to use the scripts in the previous TP to create a Simgrid network, then run your algorithm using $1, 2, 4, 8, 16, 32$ MPI processes. What are the speedups your are getting? What happens when you increase/decrease the communication bandwidth?

Part 2

Project Discussion

Now that you are familiar with the graph traversal and all-to-all MPI routines, you have all it takes to start doing the project! Start reading the project assignment, and ask any questions you might have!

Make sure to backup all your implementations as they might be useful later on!