Distributed Systems
Autumn semester 2016
Prof. Peter Kropf, prof. Pierre Kuonen,
Assistants: Veronica Estrada, Andrei Lapin

Project 2

The Shortest Path (The Chandy-Misra algorithm)

Due: 29 November 2016, 13:00

Introduction

In this laboratory you will use a decentralized distributed algorithm. By decentralized we mean that no node in the distributed system has a global view of the system. Our goal is to calculate a global property of the system without having a master node, i.e. a node that has a global vision system.

A distributed system can be modeled using a graph. The vertices (nodes) correspond to processes and edges (links) correspond to a direct connection between two processes. Links can be in one direction (unidirectional) or in both directions (bidirectional). When the links are unidirectional, the graph is said directed and when the links are bidirectional the graph is said undirected. Finally the links can be weighted. These weights may have different meanings. In our case we consider that the weights correspond to the length of the link. In case the weights can be negative, the graph is said negative.

A path between two vertices (nodes) is an ordered list of edges (links) that must be crossed to go from the first node to the second. Note that if the graph is directed, we can only go across a link in one direction. The length of the path corresponds to the sum of the weights of all the links of the path. Of course, for two given nodes, there are, usually, several different paths, each path having its own length. The shortest path between two given nodes is (are) the path(s) with the shortest length.

In this laboratory we will only consider directed, non-negative graphs.

The global property we want to calculate is, for each node of the graph, its shortest path with a given node in the graph (the one who initiates the computation). To make this calculation we will use the Chandy-Misra algorithm [1].

Implementation

The Chandy-Misra algorithm has been implemented with POP-C++ in the *parclass ShortPathObject* (see files *shortpathobject.ph* and *shortpathobject.cc*). The main program creates the vertices of the graph (parallel objects of the *parclass ShortPathObject*) and sets the directed edges (links) between objects using the *setNeighbours* method. Then it launches the computation for one of the nodes of the graph (usually the one having the index 0) and waits until the result becomes available on this node.

Four main programs are provided. Each of these four main programs creates a different graph. They are described below.

- *torus2D.cc*: This program creates a torus (see Figure 1) of two dimensions. It is a square torus i.e. its length is equal to its width. This program creates a link in each direction for each connected node. Thus, even if the graph is formally directed, it is equivalent to a undirected graph. To launch the program type the following command:

  popcrun objmap ./torus2D nbWorkers Size

  where *nbWorkers* is the number of computers of the cluster you want to use and *Size* is the size of the torus (length and width).

- *kring.cc*: This program creates a K-Ring graph [2] (see Figure 1). Similarly to *torus2D.cc*, this program creates two links between each connected nodes. To launch the program type the following command:

  popcrun objmap ./kring nbWorkers Size k K1 K2 ...Kk-1

  where *nbWorkers* is the number of computers of the cluster you want to use, *Size* is the size of the K-Ring (number of nodes), *k* is the dimension (number of rings) and *Ki* the values for the steps (*K0* is always set to 1).

- *star.cc*: This program creates a graph having the shape of a star (see Figure 1). This graph has a central node which is linked to all other nodes by an unidirectional link. To launch the program type the following command:

  popcrun objmap ./star nbWorkers Size

  where *nbWorkers* is the number of computers of the cluster you want to use, *Size* is the total number of nodes of the graph.

- *randG.cc*: This program creates a random connected graph. To launch the program type the following command:

  parocrun objmap ./randG NbWorkers NbNodes nbMaxNeighbors

  where *nbWorkers* is the number of computers of the cluster you want to use, *NbNodes* is the size of the graph and *nbMaxNeighbors* is the maximum number of neighbours for a node i.e. each node has a random number of neighbours between 1 and *nbMaxNeighbors*.

You can notice that, for each of these programs, the number of created nodes and the number of computers you use are two independent values. The program allocates, in a round robin way, the created nodes to the different computers.

All these programs generate a result file called *Excel.txt* that contains the following information:

- the parameters which define the graph (size, ...);

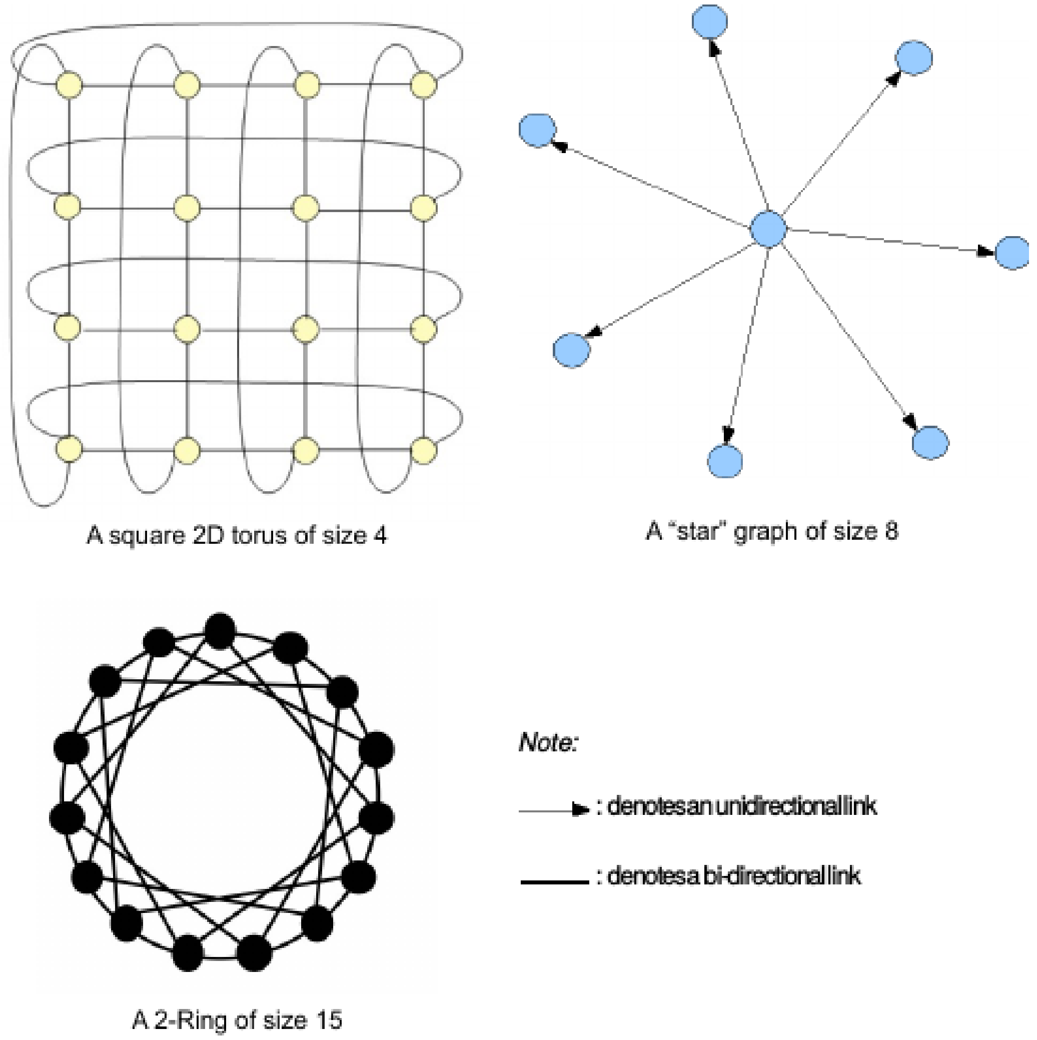
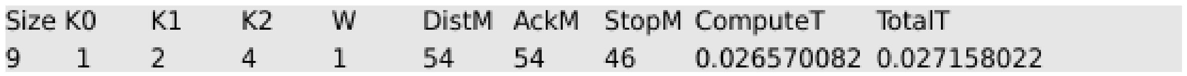

A square 2D torus of size 4
A "star" graph of size 8
A 2-Ring of size 15
Note:
: denotes an unidirectional link
: denotes a bi-directional link

Figure 1

- the number of computers the execution has used (W);
- the total numbers of messages of type *dist* which have been sent in the graph (*DistM*);
- the total numbers of messages of type *ack* which have been sent in the graph (*AckM*);
- the total numbers of messages of type *stop* which have been sent in the graph (*StopM*);
- the computing time in milliseconds (*ComputeT*);
- the total time in milliseconds (*TotalT*).

Figure 2 illustrates an example of the *Excel.txt* file for an execution with a K-Ring of size=9, dimension=3, K0=1, K1=2, K3=4 on one computer (W=1):



```
Size K0     K1     K2     W      DistM  AckM   StopM  ComputeT     TotalT
9    1      2      4      1      54     54     46     0.026570082  0.027158022
```

Figure 2

The computing time is the local time elapsed, on the node who initiated the computation (the initiator), from the beginning of the computation until the initiator have received all *ack* messages.

The total time is the local time elapsed, in the main, from the time it launches the computation until it gets the computing time of the node who initiated the computation.

Please notice that the following behaviors are normal:

- You can get several WARNING messages when you compile the files;
- When the computation is done (results are displayed on the screen) several seconds (up to several tenths of seconds) can elapse until you get the prompt back. This time is used by the POP-C++ run-time to detect that the computation is terminated and that it can destroy all the created nodes.

Work to do

Make sure you understand the Chandy-Misra algorithm and its implementation with POP-C++. To do so, analyse the *shortestpathobject.cc* source file. In addition, you may search on the internet for a description, for example on *http://en.wikipedia.org/wiki/Dijkstra's_algorithm* or you can read the original article [1], which is available on ILIAS.

1. Explain very shortly (max. one page, not in detail) how the algorithm works and especially the different phases of the implemented version in POP-C++.
2. Execute the four programs and test them with different parameters.
   - Explain the values of the the results that are provided by each experiment (were these results expected yes or no, and why);
   - Run each program several times with the same parameters and take the average values for the time
   - How the *nbWorkers* (W) parameter influences the results. Explain.
3. For *randG*: what is the influence of the parameter *nbMaxNeighbors* on the computing time? Explain.
4. For *kring*: use the following configuration:
   - *size* = 9, *dimension* = 2, *nbWorkers* = 9 and *K0* = 1 (always in a K-Ring). Vary the value of: *K1* = 2, 4; show graphically the number of dist and ack messages depending on the value of *K1*. Explain the obtained results.

5. In *torus2D.cc*, the computation is started on the *theObjects[0][0]*. What would be the impact if you would change this to *theObjects[torussize/2][torussize/2]*? Explain.

6. Same question for *star.cc*, if the computation is launched with *theStar[1]*.

7. Use and edit the provided resources (*torus2D.cc* or *kring.cc* or *star.cc* or *randG.cc*) to create your own graphs. Then, test if the algorithm succeeds with your graph.

    - For example, create a source file *"house.cc"*, which creates a "house" graph (Figure 3): Does the experiment work or is there an endless loop, or any other "strange" result when launched?
    - Do not only use this example, but define your own graphs, you must use at least three different graphs. Choose your graphs in order to check some specific results or proprieties of the algorithm. Analyze the expected results, check and explain the obtained results. Also include versions where nodes are connected by two or three edges and versions with negative edge costs etc. Does the algorithm still hold?

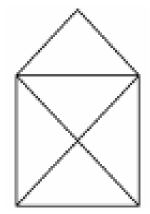


Figure 3

For tests running on only one machine, or tests you are not interested by time results, you can use your own machine.

Always indicate the hardware on which you run the tests.

Submission

The following documents must be uploaded on ILIAS:

1. A pdf version of your report (max. 12 pages!). Each student has to implement his/her own version of the report . This report should contain all your interesting results (in table and/or plot formats), as well as a pertinent analysis and evaluation.

2. A zip file containing all the result files (text format) produced by the program that you used to draw plots of your report.

Please respect the following point:

- Make sure that your compressed document is of reasonable size (no need for images of high-quality) and upload it on ILIAS by strictly following the naming conventions.

Note that the fact that you strictly followed all the instructions concerning the way to provide your report and results (formatting, type of file, deadline,...) will be taken into account to establish your mark.

References

[1] K. M. Chandy, J. Misra, *Distributed Computation on Graphs: Shortest Path Algorithms*, Communications of ACM, November 1982.

[2] P. Kuonen, *The K-Ring: a versatile model for the design of MIMD computer topology*, in proceedings of the High-Performance Computing Conference (HPC'99), pp. 381-385, SanDiego, 11-15 April 1999.