Parallel and Distributed Algorithms – Instruction 1

Written by: Karol Draszawka
Date: 10.03.2016

1 Aims of the laboratory

The first lab has three main aims:

- introducing our simple simulator of a distributed computing system, which will be used throughout the whole course in the laboratory;
- introducing algorithms performance measurement framework, which also will be used in most the labs;
- finally, familiarizing students with one of basic communication operations and possible algorithms besides that operation. Students will be given a naive version of that operation (implemented using algorithm of linear complexity), and will have to write a correct (of logarithmic complexity) version.

The code for this lab is in Lab01X.zip file. You should extract files and open the project from Netbeans IDE.

2 Simple Distributed Computation System Simulator

Package `distributedmodel` contains code of the simulator. The package implements a model of distributed computing nodes connected with each other using a network (see Illustration 1).

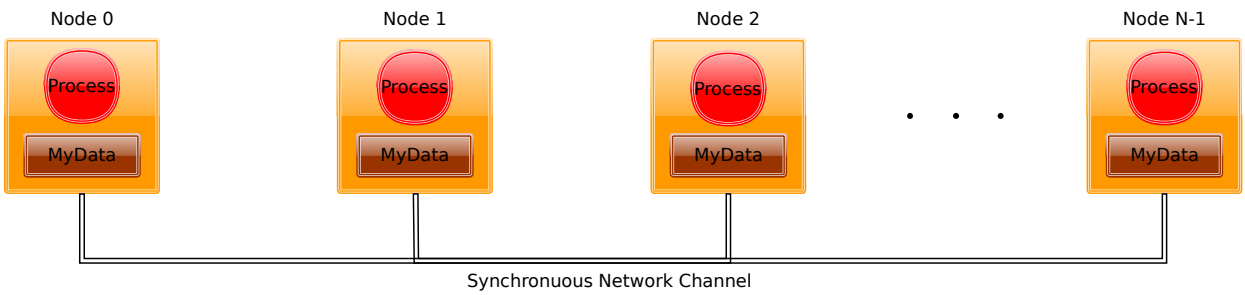

Illustration 1: Schematic view on a simulated distributed system of computing nodes.

In a system with N nodes, the nodes have indexes from 0 to N-1. The number of nodes in a distributed system is configurable during construction. The system is distributed, so there is no shared memory between the nodes. All the communication between them has to be done by messaging through the network.

Each node has its computation process as well as its memory, implemented as `MyData` field of `double[]` type. This field has its public getter and setter. A node can check what index it has in a current distributed system (`getMyIdx`() method) as well as how many nodes are in the system in total (`getNumberOfAllNodes`()).

The rest of public methods of class Node is related to point-to-point network communication. There are two families of methods: related to sending and receiving. Both sending and receiving methods are **synchronous**. This means that calling a `send` method (in all its overloads) blocks the execution of the instructions of the sending node until the message is received by a destination node (via one of `receive` methods). The same is true for receiving methods: they block a receiving node until some node has sent a message to the receiver.

Because sometimes the synchronous feature of communication methods makes troubles (for example, one of the nodes is waiting at send or receive and therefore prevents the whole distributed system from closing the simulation), the methods can be called with `verbose` flag set to true (this is the default behavior) enabling logging every communication detail to the console.

A node must specify which other node is the desired destination of the message by passing the destination parameter to a send method. On the other hand, receive methods do not have corresponding source arguments - they simply receive everything which comes to node's input port. If the knowledge of an origin of the message is required, this can be obtained by investigation of a received `DataPacket` structure.

Because it is often the case that nodes send data from MyData field as a whole, as well as nodes frequently set this field to just received data, there exist shortcut methods: `sendMyData` and `receiveAndSet`.

The knowledge of the details of the system not given above are not needed to do the exercises, but interested students are welcome to investigate the code in this package.

3 Performance measurement framework

Package `labs` contains starting code of the project as well as algorithms' performance measurement framework. When project is run it constructs a few times a new distributed system (with varying number of nodes) (see method `testAll` in Lab01X class). After each construction it invokes a tested distributed algorithm (the algorithm is run in parallel on all the nodes of a constructed distributed system) and waits for the algorithm to finish. The algorithm's running time is measured. This is repeated for systems with increasing number of nodes. This way, the complexity of algorithms can be empirically checked by simply looking at a chart generated at the end of `testAll` method. After opening the project and clicking run Button, such series of simulations should be executed and, after a while a chart should pop out, one like this on Illustration 2.

Besides measuring time, after each simulation the status of nodes in a distributed system can be validated (if `validateResults` flag is turned on). It is important to ensure that the validation is successful ('Results are CORRECT' text in the output console), otherwise the algorithm does not do what expected. Initially, the first four simulations are successfully validated, but the next four not. This is because the first four run a already implemented naive version of an algorithm under investigation and the second four run not yet implemented correct version (i.e. with better time complexity characteristics).

In addition, `verboseCommunication` flag controls whether all the details about point-to-point communication should be logged, and `printStatusBeforeAndAfter` flag determines whether to print contents of `MyData` fields of all nodes before and after the execution of an algorithm.

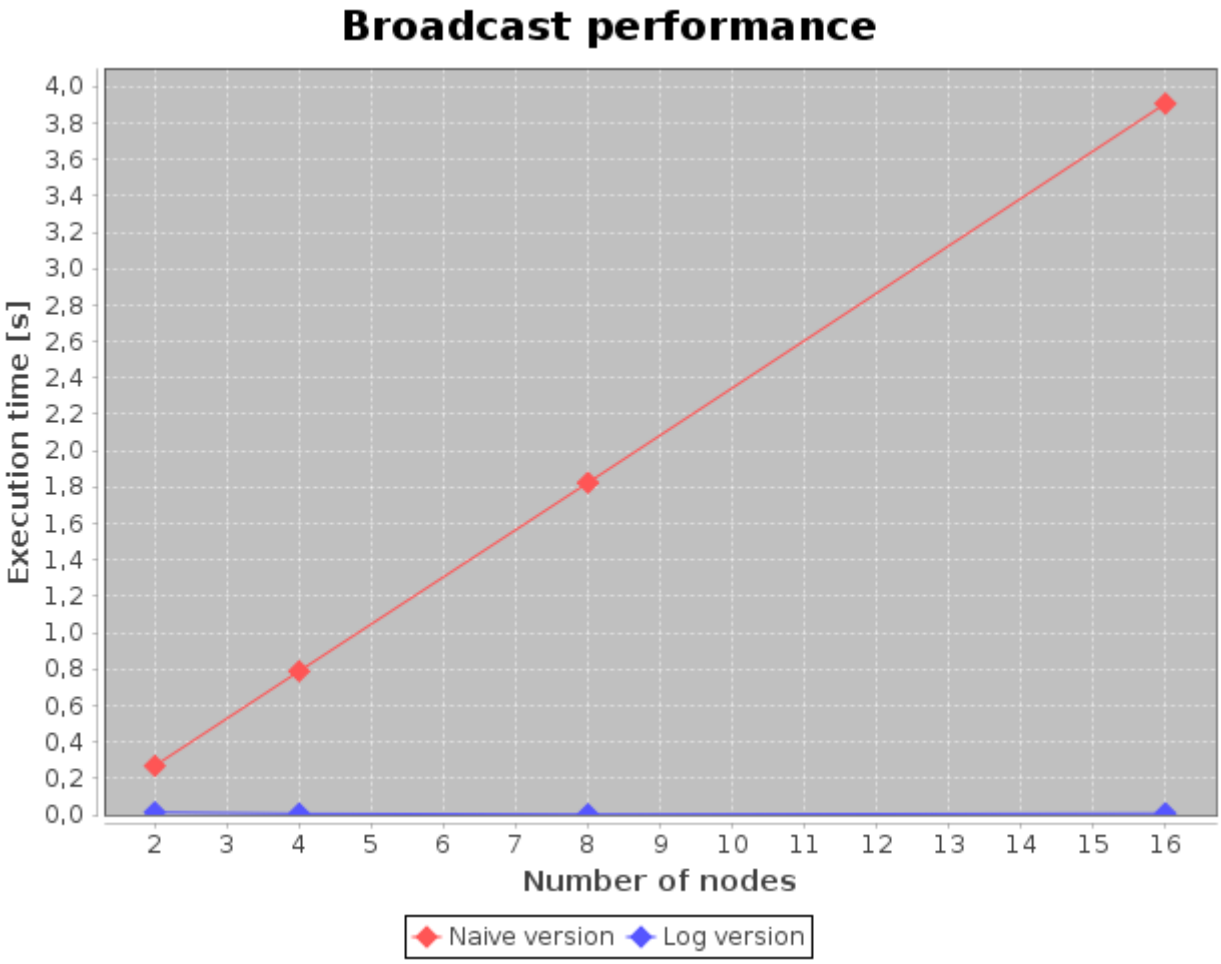

Illustration 2: Performance chart after running the project.

4 Basic communication algorithm – Broadcast example

Package `algoritms.distributed` is the place where classes implementing given algoritms are placed. In this case, algorithms are written as methods in BasicCommunication class. Here, the broadcast operation will be described, but in the lab there can be other algorithm for BasicCommunication (such as scatter, gather, reduce, all-to-all broadcast and so on).

The crucial thing to keep in mind that the methods implementing the algorithms (such as `public static void broadcastNaive(Node node)`) are executed in parallel locally on all the nodes of the distributed system. To differentiate the execution of the program between nodes, one must depend the computation on node's index. This is why almost every distributed program first checks what is the index the executing node has.

The aim of broadcast operation is to distribute data initially stored on one node to all the other nodes in the system. Here we assume that initially node 0 is the one which has data (in its MyData field) and all the other nodes have no data. The exepected state after broadcast operation is that all the nodes has their MyData fields equal to what initially was only in node 0.

Naively implemented broadcast do this task as follows. If a node does not have any data, it waits to receive it calling receiveAndSet method. If it gets data (or had data from the start, as in case of node 0), then it sends it further to the next node (unless it does not exist). After what was send is received, the node has done his work. Visually, the communication pattern is presented in Illustration 3.

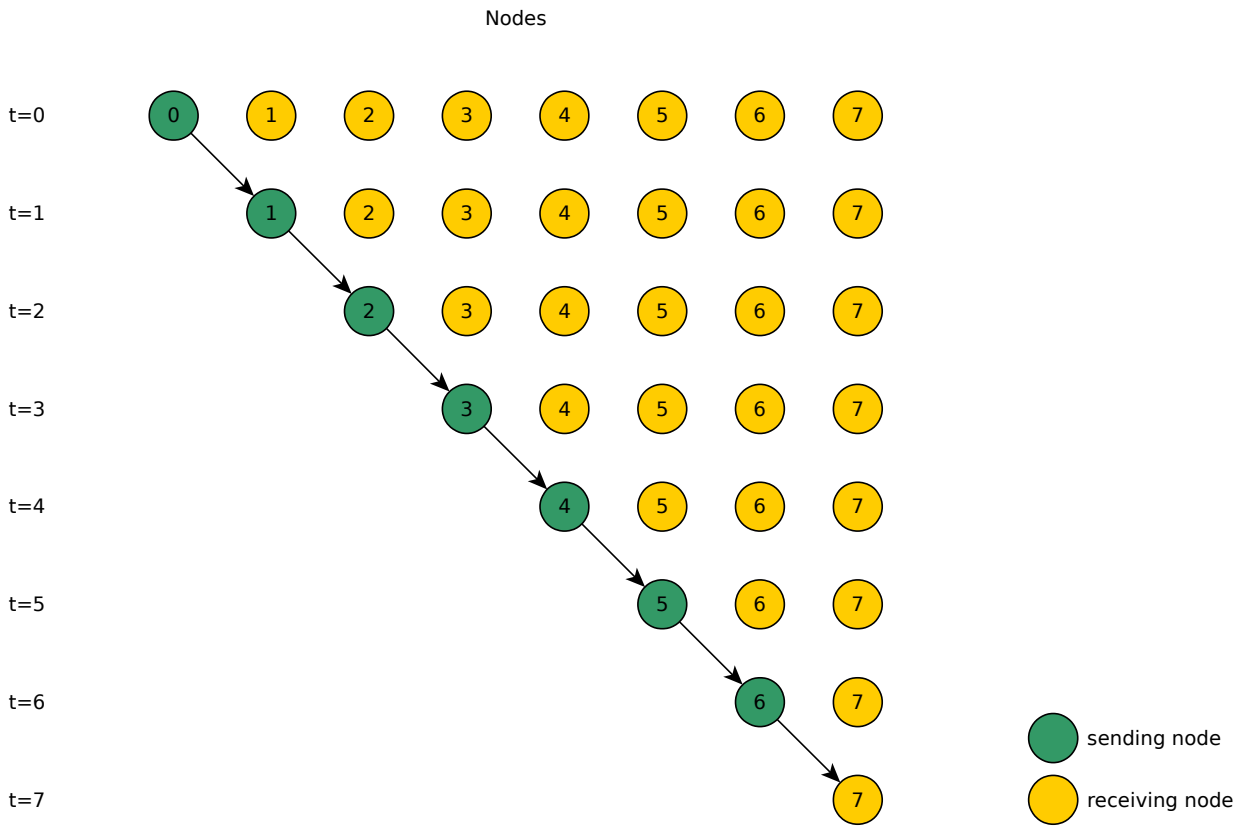

Illustration 3: Naive broadcast communication pattern.

5 Student's task

Student should fill in the code implementing the correct version of a basic communication algorithm. In case of broadcast, the communication pattern could be one of two presented in Illustration 4 and Illustration 5. Ensure that simulations are successfully validated and that the performance chart has the correct curve (should be logarithmic). In the coding, `binlog` helper methods, defined in `algorithms.Utils` class may be useful.

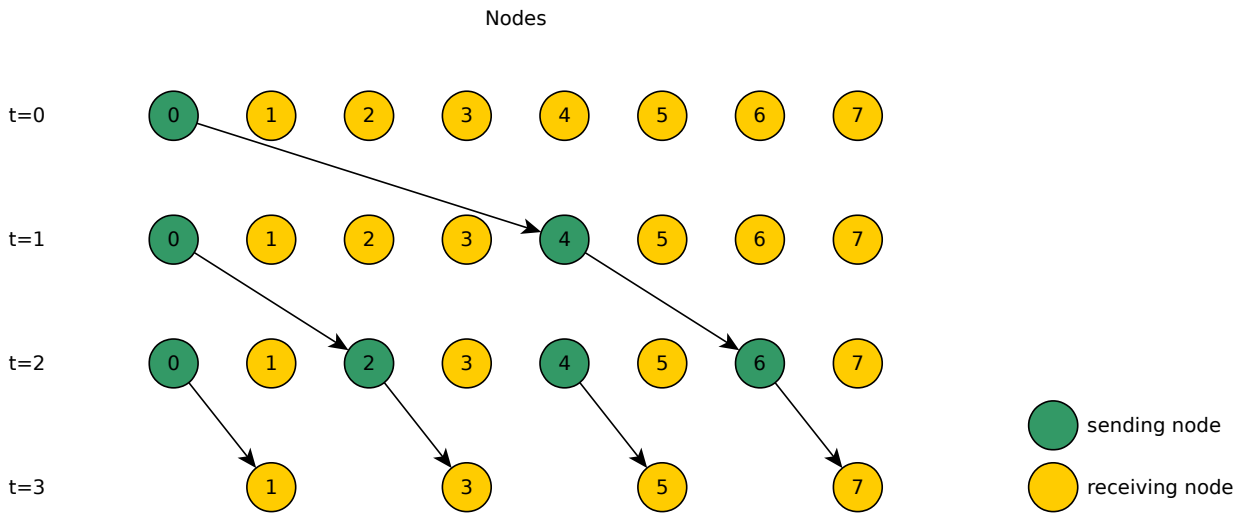

Illustration 4: Correct broadcast communication pattern - version 1.

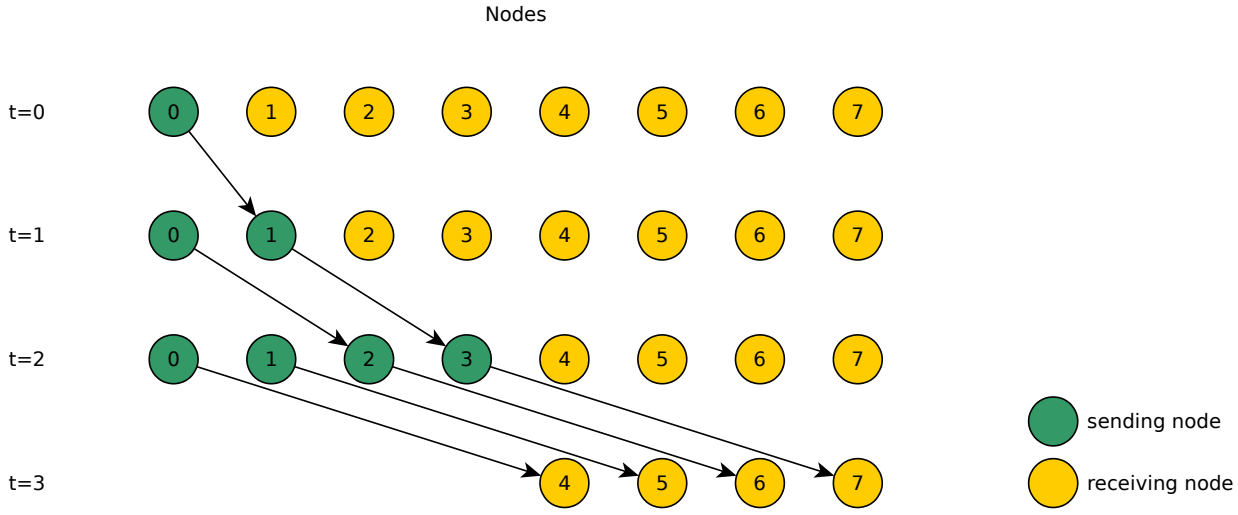

Illustration 5: Correct broadcast communication pattern - version 2.