Parallel and Distributed Algorithms – Instruction 2 – Scan operation

Written by: Karol Draszawka
Date: 18.03.2016

1 Aims of the laboratory

The second lab has two main aims:

- familiarizing students with scan (prefix op) algorithms, for distributed systems with blocking communication as well as for systems with shared memory
- introducing and familiarizing students with shared memory parallel computations using threads in Java programming language

The code for this lab is in Lab02.zip file. Extracted folder should be opened as a project from Netbeans IDE.

2 Scan operation and it's various algorithms

A **scan** operation for a data sequence returns another linearly ordered data such that an element at position *i* is a result of an binary associative operator applied to all elements of the source sequence up to that position *i*. For example, if a binary associative operator is simply addition of two elements, then for input sequence [1, 2, 3, 0, 5] the resulting sequence is [1, 3, 6, 6, 11] (the scan operation with addition as binary associative operator is often called **prefix-sum**).

Serial code for scan is therefore:

```
public static void scanSerial(double[] array, DoubleBinaryOperator op){
    for(int i = 1; i < array.length; ++i){
        array[i]  = op.applyAsDouble(array[i-1], array[i]);
    }
}
```

The serial code has time complexity *O(n)*, where *n* is the size of a sequence.

2.1. Two versions of scan for distributed systems with blocking communication

There is a number of communication patterns for distributed processes that want to perform a scan operation. Illustration 1 shows one possibility. The algorithm is composed of two phases.

The first one is simply a **reduce** operation, where the process with largest rank/ID collects the result of applying an operator through the whole sequence using binary tree structure. The second phase "fills" the gaps in resulting sequence to obtain the desired output. In this version, a value holded by a given process is updated every time it receives a message by applying a binary operator between this value and value received.

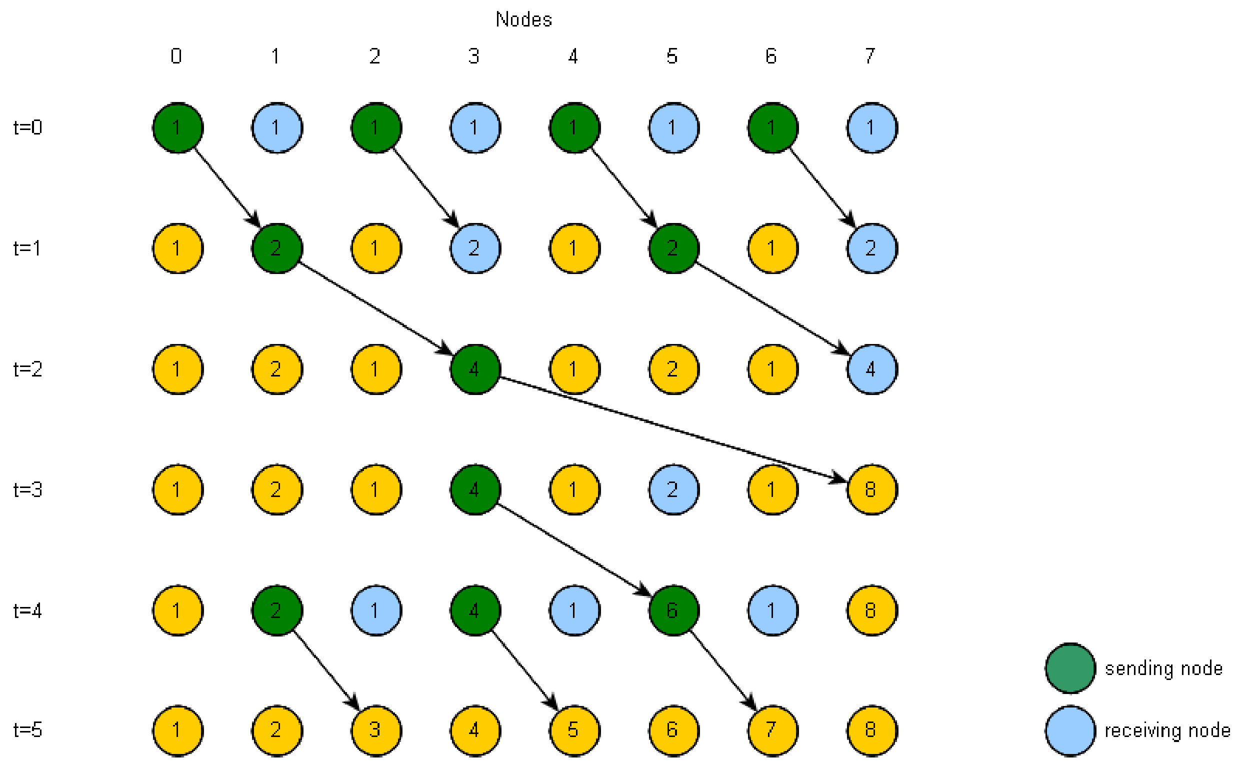

Illustration 1: Scan algorithm (prefix-sum) for a distributed system - version 1.

Illustration 2 shows another possible communication pattern. The pattern is composed of a number of mutual data exchanges between processes. With blocking communication, the exchange process requires two time steps. An already written method `exchangeWith` does this and can be used for this algorithm. Although the pattern is easier when compared to the above, to calculate correct scan results nodes must hold and calculate two separate values: it's *own* value (which at the end will compose resulting sequence of scan), and auxiliary *message* value (the value that will be send by this node in the next step). Message value is updated every time a node receives a message by applying binary operator between this value and received one. However, a node's own value is updated only if it received data from a node with smaller ID. Initially, both values are the same for a given node.

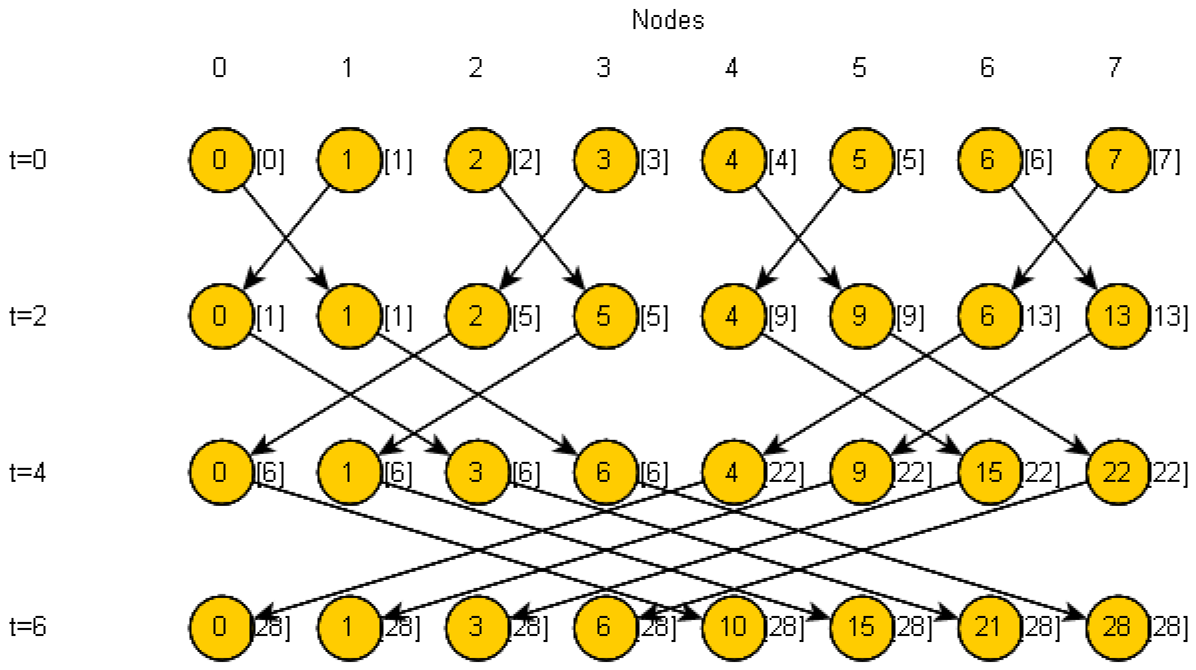

Illustration 2: Scan algorithm (prefix-sum) for a distributed system -version 2.The value inside a node represents the 'own' value of the node, while the number inside brackets besides a node describes a 'message' value of the node.

2.2. Scan algorithm for processes with shared memory (Hillis/Steele version)

For processes that share memory, another communication pattern guarantees valid scan results (see Illustration 3). Implementing this algorithm it is important to note, that between every write and read operations of a shared memory cell, there should be a synchronization of threads. Otherwise one thread can read not yet updated value by another cell, or a thread can update a cell before another even read its previous value.

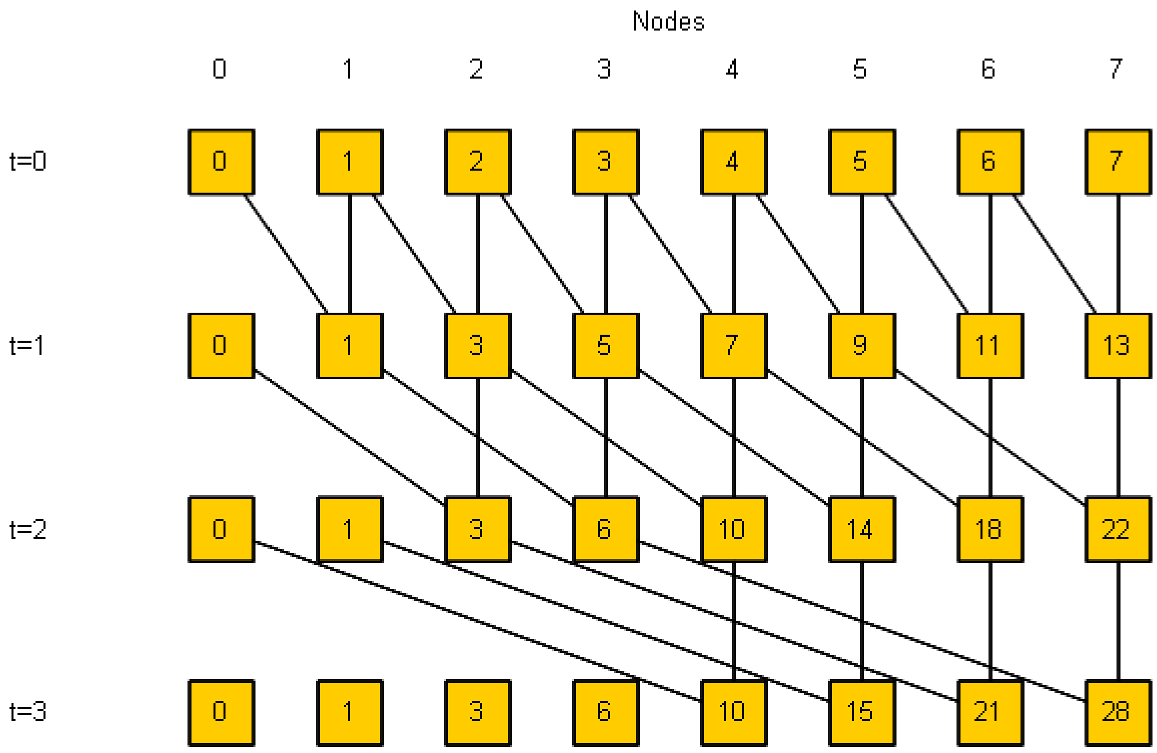

Illustration 3: Hillis-Steele scan algorithm for shared memory.

3 Shared memory parallel computation using threads in Java

Algorithms in package `algorithms.shared` will have a common structure. They will contain a definition of a **`Runnable`** class, that has to implement a public `run()` method, which will normally be a student's task. Then a number of **threads** will be created, each with a new instance of the Runnable class. Then all created threads are started to execute in parallel code defined in run() method. When all the threads finish their work, a given algorithm ends.

Because shared memory parallel algorithms often requires **synchronization** between all processes, each constructed thread has a handler to the same one instance of a `CyclicBarrier` class. When the synchronization is needed, they can call an `await()` method on that barrier instance. The barrier blocks threads in this method as long as all the threads will execute this call.

4 Student's task

Student should fill in the code implementing one of the presented versions of a scan operation for distributed system and Hillis/Steele algorithm for shared memory threads.
In `testAll` method of Lab02 class, student can select which tests to perform – initially single

tests should be chosen, so that the correctness is confirmed. After checking correctness on single runs, ensure that algorithms have correct time complexity by running series of simulations to show performance chart. As previously, in the coding, `binlog` helper methods, defined in `algorithms.Utils` class may be useful.