Parallel and Distributed Algorithms – Instruction 3 – Sorting algorithms

Written by: Karol Draszawka
Date: 7.04.2016

1 Aims of the laboratory

The second lab has two main aims:

- familiarizing students with two parallel sorting algorithms: bitonic sort for distributed systems with blocking communication and parallelized quicksort for systems with shared memory
- introducing and familiarizing students with shared memory parallel computations using ForkJoin mechanism in Java programming language

The code for this lab is in Lab03.zip file. Extracted folder should be opened as a project from Netbeans IDE.

2 Bitonic sort

Bitonic sort is one of many sorting algorithms based on sorting nets, i.e. simple computational nodes with connections forming special patterns. Illustration 1 depicts this connections using arrows: after exchanging a node from which an arrow starts saves a smaller value, while a node to which an arrow points to saves a greater value.

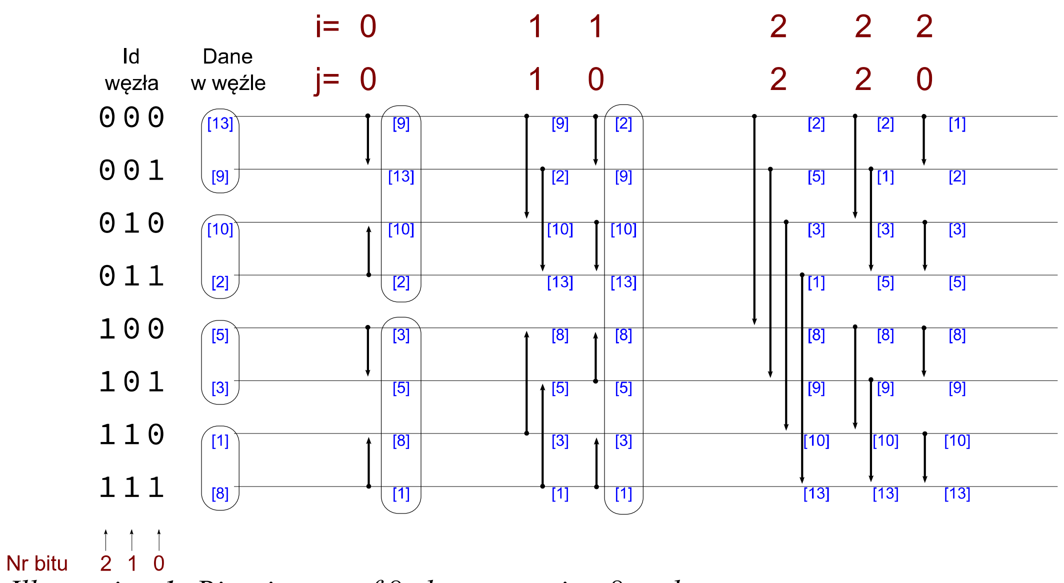

Illustration 1: Bitonic sort of 8 elements using 8 nodes.

Such a bitonic sort algorithm is already implemented in a `Sort` class in `algoritm.distributed` package:

```
/** Bitonic sort of the first elements of each node data.
 *
 * @param node
 */
public static void bitonicSortAsc(Node node){
    int myId = node.getMyId();
    double myData = node.getMyData()[0];
    int nNodes = node.getNumberOfAllNodes();
    int d = Utils.binlog(nNodes);

    for(int i = 0; i<d; ++i){
        for(int j = i; j >= 0; --j){
            int otherId = myId^(1<<j);
            double otherData = BasicCommunication.exchangeWith(node,
                                                               otherId, myData);

        //if in myId (i+1)-th bit is not equal to j-th bit
        if( ((myId&(1<<(i+1))) != 0) != ((myId&(1<<j)) != 0) ){
            myData = Math.max(myData, otherData);
        }else{
            myData = Math.min(myData, otherData);
            }
        }
    }
    node.setMyData(new double[]{myData});
}
```

Described algorithm needs *n* nodes to sort *n* elements. To sort *n* elements on *p* (*p* << *n*) nodes, it has to be modified in the following way:

- firstly, assuming that each node has exactly *n*/*p* elements, each node should sort them locally
- secondly, nodes should continue exchanging their arrays in accordance to bitonic sort net pattern applying an compareSplit operation in each step, as it is shown in Illustration 2.

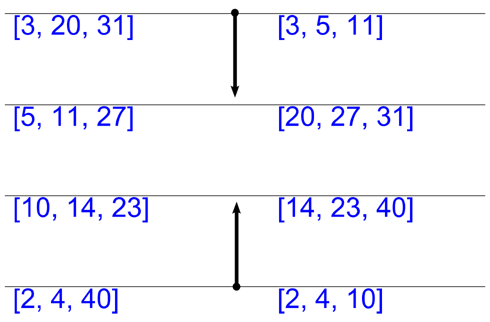

Illustration 2: compareSplit example.

3 QuickSort

QuickSort is a very popular sorting algorithm. It has Θ(n lg n) average time complexity with relatively small coefficients hidden under Θ notation. It sorts arrays *in place*. Pseudocode of quicksort algorithm is given below:

```
quicksort(A, lo, hi):
      if lo < hi:
            q := partition(A, lo, hi)
            quicksort(A, lo, q - 1)
            quicksort(A, q + 1, hi)
```

The algorithm recursively calls itself with the same array and different start and end point indexes. In the `partition` part one of array elements is chosen as so called *pivot* value. Then an array rearrangement is done, so that values smaller than pivot goes to positions before pivot and bigger values to positions after pivot. The final position of pivot value defines then ranges of two array parts that are then recursively sorted in the same manner.

The implementation of the algorithm can be found in `SortSM` class in `algorithms.shared` package.

There are a couple of ways in which quicksort can be parallelized. Here a 'divide & conquer' strategy is described. The parallelization can be achieved by simply starting each new recursive call to quicksort on a new thread, that potentially can run in parallel to others. Because these recursive calls work on different parts of an array there is no worry about hazards. However, naively creating `new Thread(...)` for each call to quicksort would cost significant overheads caused by threads creation, which would not be compensated by parallel computations, if the number of threads exceeds the number of available processors. This is where Java Fork/Join framework becomes handy.

4 Fork/Join Java framework

Fork/Join[1] framework, introduced in JAVA SE7 and added to `java.util.concurrent` package, is 'divide & conquer' parallelization technique. `ForkJoinPool` creates only as many threads as there are processors available. Recursive code has to be written in a class inheriting from `RecursiveAction` (or `RecursiveTask<T>`, if it returns a value). Tasks are scheduled between threads in a way that maximizes utilization of resources.

Class `SortSM` contains an example of using fork/join framework to find the maximum value of an array and place it at position 0 of that array (an example purposely formulated in *in place* form).

5 Student's tasks

- write a generalized version of bitonic sort filling the code in the method `bitonicSortGeneralizedAsc` in class `algorithms.distributed.Sort`.
- write a parallel quicksort algorithm using Fork/Join framework filling the code in class `algorithms.shared.SortSM`.

1 More at: https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html
and http://www.oracle.com/technetwork/articles/java/fork-join-422606.html