Report Parallel Algorithms and Distributed Systems Project

Sorting Algorithms

1. How to run the program

Pre-requirements: Having allegro5 installed in the same directory of the project

```
git clone https://github.com/liballeg/allegro5.git

cd allegro5
mkdir build
cd build
cmake ..
make
sudo make install
```

To compile the program on Linux:
make

To run the program:
mpirun -np <num_processors> **./sorting.out** <size_sequence> <graphic>

example: **mpirun -np** 4 **./ sorting.out** 1024 1

2. Brief description

The following sorting algorithms have been implemented:

- Bitonic Sort
- Odd-even transposition
- Parallel Quicksort

A random sequence is generated with the size equivalent to the number supplied by the command line (sequence_size). This sequence is written on a .txt file preceded by its size. All the algorithms order the same sequence in order to get the correct execution times.

The graphics were created with the “Allegro5. Note that the speed changes if you decide to graphically display the execution of the sorting algorithms. If you decide to check the actual times of the algorithms, you have to run the program without the graphical display. For simplicity, the sequence displayed graphically is that of the *MASTER*.

The printouts, the time calculation, some checks, the generation of random numbers and the reading of the numbers are done only by the *MASTER*.

The sequence to be sorted is distributed equally on all processors (democratic system) via *MPI_SCATTER*. Each processor executes the sequential sorting algorithm on its portion of the array, and then joins the sequence into a single array via *MPI_GATHER*.

3. Bitonic Sort

The Bitonic Sort is a sorting algorithm which consists in the creation of a bitonic sequence, starting from an unordered sequence.
The bitonic sequence is a sequence whose value of the elements first increases and then decreases.
For the algorithm to work correctly, the number of elements must be a power of 2.

We first start by dividing the data sequence into *n* parts, where *n* is the number of processes, and then send a portion to each. process. Each process orders its own subsequence and then there will be *n* sequences of sorted data that must be merged to create a single contiguous result.

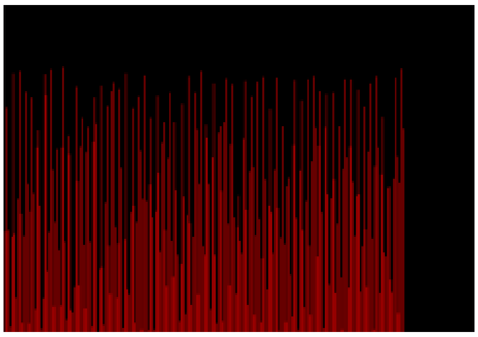
The data will be sorted globally in log2 n steps. These steps will follow the standard bitonic sorting model.
The first steps are to create a bitonic sequence and the rest to order it. Each process carries out a number of exchanges with various "partners".

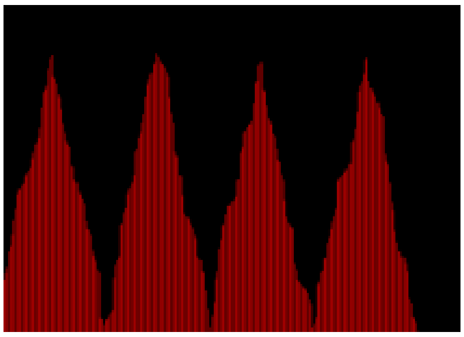
It uses bitonicSortCrescente or bitonicSortDecrescente calls. In these methods, partner data is sent to each other and then each calls a join function (joinSequence), which joins only half of the data.

joinSequenceHigh:
Transfers data to partner process
Receives data from partner process
Starts joining sequence data of type **HIGH** until it has joined dim elements, where di is the length of the original data array

joinSequenceLow:
the same except that starts merging from **LOW**

Once everything is complete, the data of all processes is sent back to the master and is in order. Due to the way the data is sorted, this is an unstable sorting algorithm.





Times:

Thread	1024	16384	131072	1048576
1	0,0004976	0,0140633	0,129507	0,746101
2	0,0018371	0,0065515	0,0421884	0,320929
4	0,0085829	0,0056829	0,0216644	0,220796
8	0,0015382	0,0200332	0,0203935	0,185531

SpeedUp $(\mathrm{SequentialTime}/\mathrm{ParallelTime})$ it is assumed that the sequential time is equal to the parallel time executed with 1 thread

Thread	1024	16384	131072	1048576
1	100%	100%	100%	100%
2	27,09%	214,66%	306,97%	232,48%
4	5,80%	247,47%	597,79%	337,91%
8	32,35%	70,20%	635,04%	402,14%

4. Odd-Even Transposition

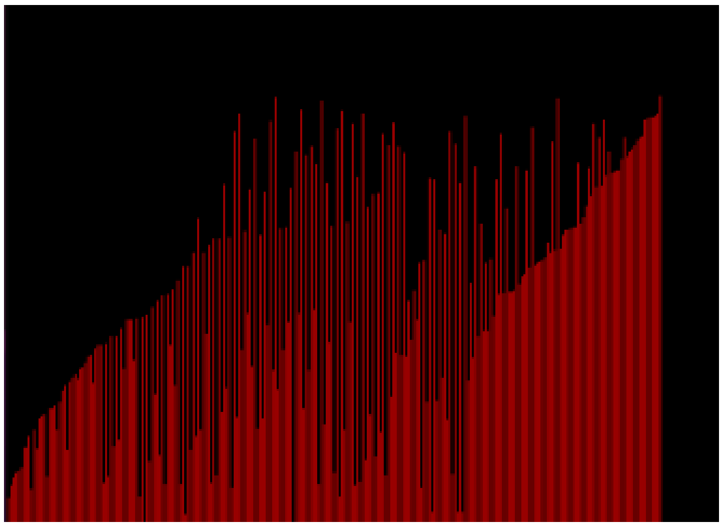
It works by comparing all the odd and even pairs of elements in a list and, if a pair is in the wrong order (the first element is greater than the second), swaps its elements. The check continues with the pairs of adjacent elements with even / odd positions. The algorithm continues sorting by alternating between odd / even and even / odd comparisons until the whole list is sorted.

Times:

Thread	1024	16384	131072	1048576
1	0,0016376	0,2690180	18,7366000	321,1225100
2	0,0007940	0,1070300	4,5741000	273,6657690
4	0,0004109	0,0226982	1,2081200	80,3679420
8	0,0014497	0,0213565	0,4750250	32,4912000

SpeedUp $(\text{SequentialTime}/\text{ParallelTime})$ it is assumed that the sequential time is equal to the parallel time executed with 1 thread

Thread	1024	16384	131072	1048576
1	100%	100%	100%	100%
2	206,25%	251,35%	409,62%	117,34%
4	398,54%	1185,20%	1550,89%	399,57%
8	112,96%	1259,65%	3944,34%	988,34%



5. Quick Sort

The quicksort is a divide-and-conquer algorithm by partitioning the elements into two sequences that are sorted recursively and then reuniting them. The divides part chooses an element of the sequence, called pivot, and partitions the sequence into two parts, one with the minor elements equal to the pivot number and the major ones in the "right" subsequence and then recursively ordering the two sub sequences obtained. Finally, the ordered subsequences are concatenated.

Times:

Thread	1024	16384	131072	1048576
1	0,0001264	0,0029607	0,0136256	0,1332570
2	0,0002897	0,0018367	0,0117243	0,0758822
4	0,0006664	0,0014889	0,0093129	0,0746069
8	0,0003114	0,0097465	0,0132725	0,0737833

SpeedUp $(\mathrm{SequentialTime}/\mathrm{ParallelTime})$ it is assumed that the sequential time is equal to the parallel time executed with 1 thread

Thread	1024	16384	131072	1048576
1	100%	100%	100%	100%
2	43,63%	161,20%	116,22%	175,61%
4	18,97%	198,85%	146,31%	178,61%
8	40,59%	30,38%	102,66%	180,61%