Introduction

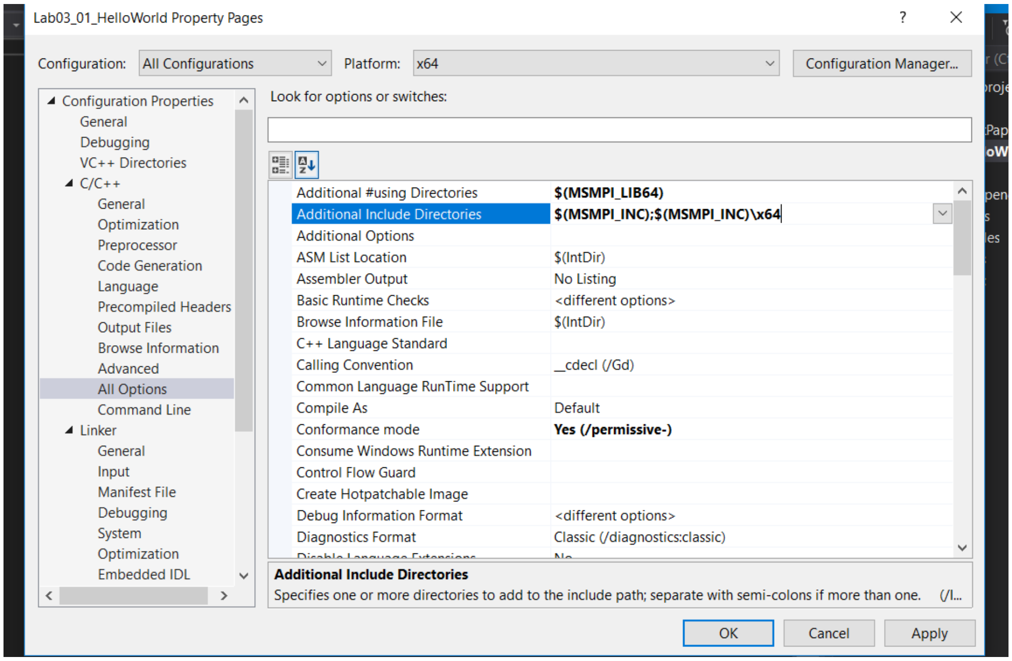
The tests were made on a Laptop PC using a dual-core Intel i7-7500U CPU with 4 threads clocked at 2.70Ghz.

The System has 8GB of RAM an SSD Hard-Drive and a dedicated GPU.

Note: other programs were running during the tests.

The executables were compiled using Visual Studio 2017 on Windows 10.

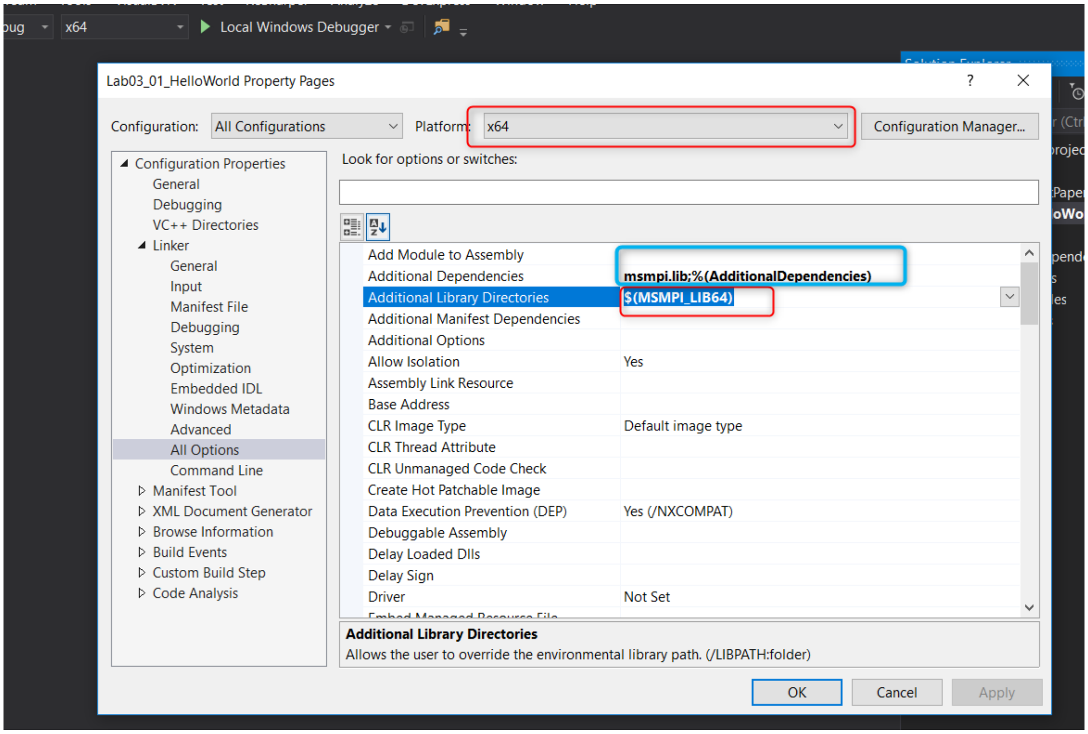
Small adjustments were made to the projects in order to be able to compile the MPI programs.



Configuration: All Configurations Platform: x64

Additional #using Directories	**$(MSMPI_LIB64)**
Additional Include Directories	**$(MSMPI_INC);$(MSMPI_INC)\x64**
Additional Options	
ASM List Location	$(IntDir)
Assembler Output	No Listing
Basic Runtime Checks	<different options>
Browse Information File	$(IntDir)
C++ Language Standard	
Calling Convention	__cdecl (/Gd)
Common Language RunTime Support	
Compile As	Default
Conformance mode	**Yes (/permissive-)**
Consume Windows Runtime Extension	
Control Flow Guard	
Create Hotpatchable Image	
Debug Information Format	<different options>
Diagnostics Format	Classic (/diagnostics:classic)

Additional Include Directories
Specifies one or more directories to add to the include path; separate with semi-colons if more than one. (/I...



Configuration: All Configurations Platform: x64

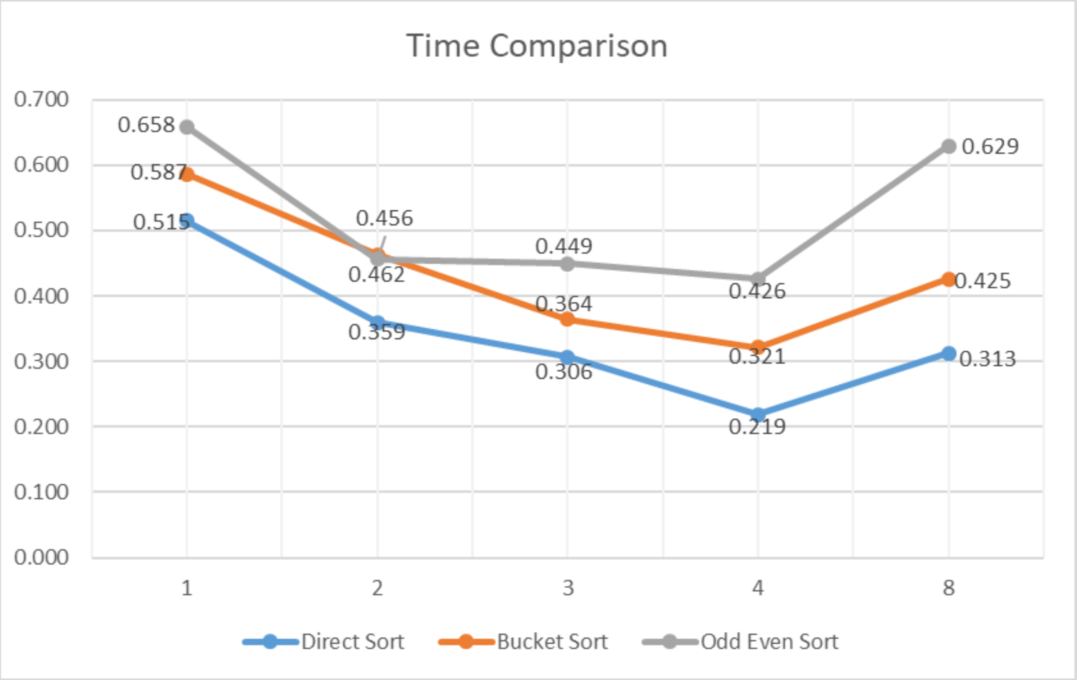
Add Module to Assembly	
Additional Dependencies	**msmpi.lib;%(AdditionalDependencies)**
Additional Library Directories	**$(MSMPI_LIB64)**
Additional Manifest Dependencies	
Additional Options	
Allow Isolation	Yes
Assembly Link Resource	
Base Address	
CLR Image Type	Default image type
CLR Thread Attribute	
CLR Unmanaged Code Check	
Create Hot Patchable Image	
Data Execution Prevention (DEP)	Yes (/NXCOMPAT)
Debuggable Assembly	
Delay Loaded Dlls	
Delay Sign	
Driver	Not Set

Additional Library Directories
Allows the user to override the environmental library path. (/LIBPATH:folder)

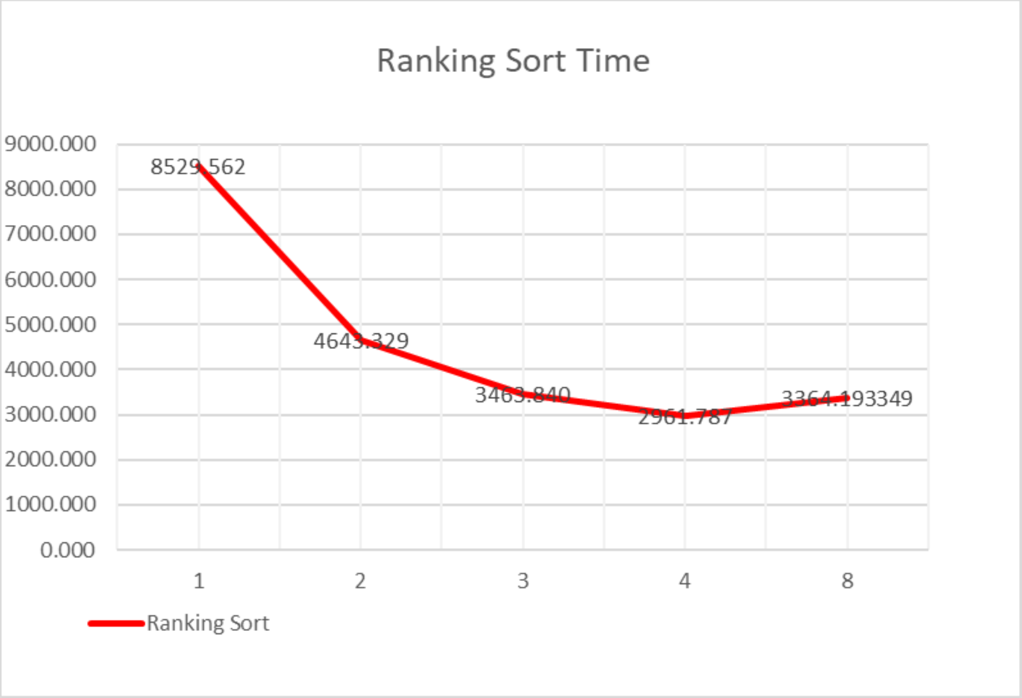
Execution Times

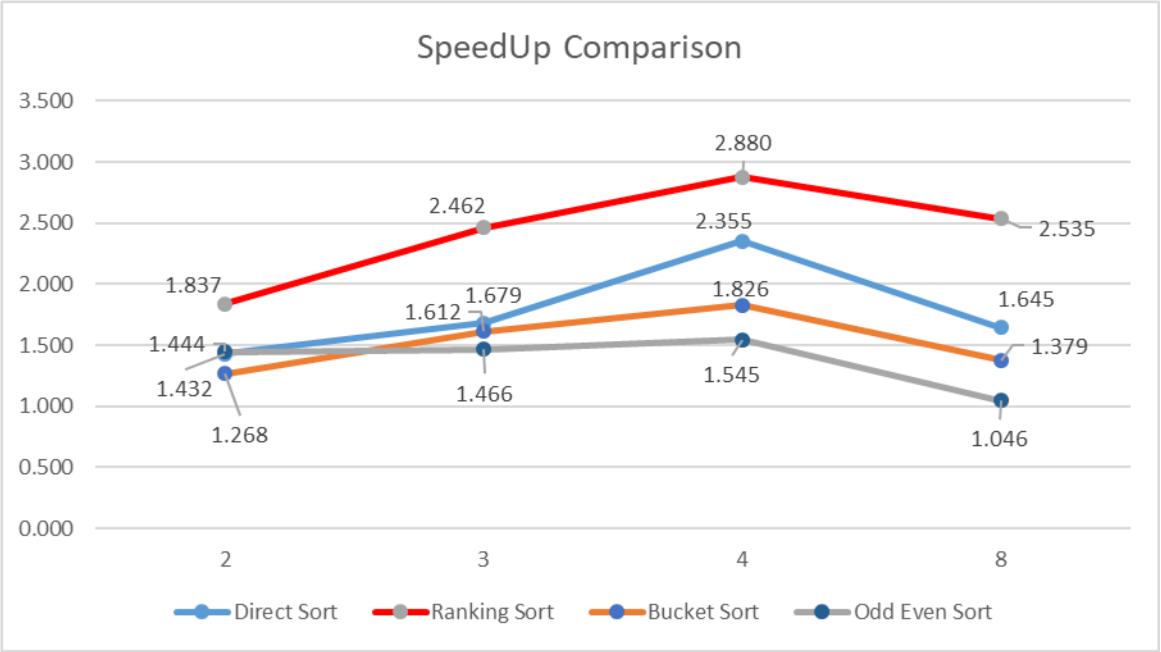
	No Procs	No Elems	Direct Sort		Ranking Sort		Bucket Sort		Odd Even Sort	
			Proc Idex	Time(s)	Proc Idex	Time(s)	Proc Idex	Time(s)	Proc Idex	Time(s)
Oversubscribe	8	1,000,000	Proc0	0.252	Proc0	3364.105	Proc0	0.296	Proc0	0.540
			Proc1	0.197	Proc1	3359.762	Proc1	0.374	Proc1	0.607
			Proc2	0.294	Proc2	3361.686	Proc2	0.414	Proc2	0.614
			Proc3	0.283	Proc3	3361.677	Proc3	0.405	Proc3	0.607
			Proc4	0.312	Proc4	3364.177	Proc4	0.400	Proc4	0.624
			Proc5	0.299	Proc5	3362.256	Proc5	0.425	Proc5	0.617
			Proc6	0.313	Proc6	3364.189	Proc6	0.399	Proc6	0.629
			Proc7	0.300	Proc7	3364.193	Proc7	0.412	Proc7	0.610
			MAX	**0.313**	**MAX**	**3364.193**	**MAX**	**0.425**	**MAX**	**0.629**
	4	1,000,000	Proc0	0.183	Proc0	2961.680	Proc0	0.233	Proc0	0.327
			Proc1	0.219	Proc1	2959.567	Proc1	0.316	Proc1	0.426
			Proc2	0.218	Proc2	2961.787	Proc2	0.315	Proc2	0.417
			Proc3	0.218	Proc3	2961.772	Proc3	0.321	Proc3	0.415
			MAX	**0.219**	**MAX**	**2961.787**	**MAX**	**0.321**	**MAX**	**0.426**
	3	1,000,000	Proc0	0.249	Proc0	3463.755	Proc0	0.287	Proc0	0.353
			Proc1	0.300	Proc1	3461.900	Proc1	0.359	Proc1	0.449
			Proc2	0.306	Proc2	3463.840	Proc2	0.364	Proc2	0.449
			MAX	**0.306**	**MAX**	**3463.840**	**MAX**	**0.364**	**MAX**	**0.449**
	2	1,000,000	Proc0	0.301	Proc0	4643.262	Proc0	0.387	Proc0	0.380
			Proc1	0.359	Proc1	4643.329	Proc1	0.462	Proc1	0.456
			MAX	**0.359**	**MAX**	**4643.329**	**MAX**	**0.462**	**MAX**	**0.456**
Serial Time	1	1,000,000	**Proc0**	**0.515**	**Proc0**	**8529.562**	**Proc0**	**0.587**	**Proc0**	**0.658**

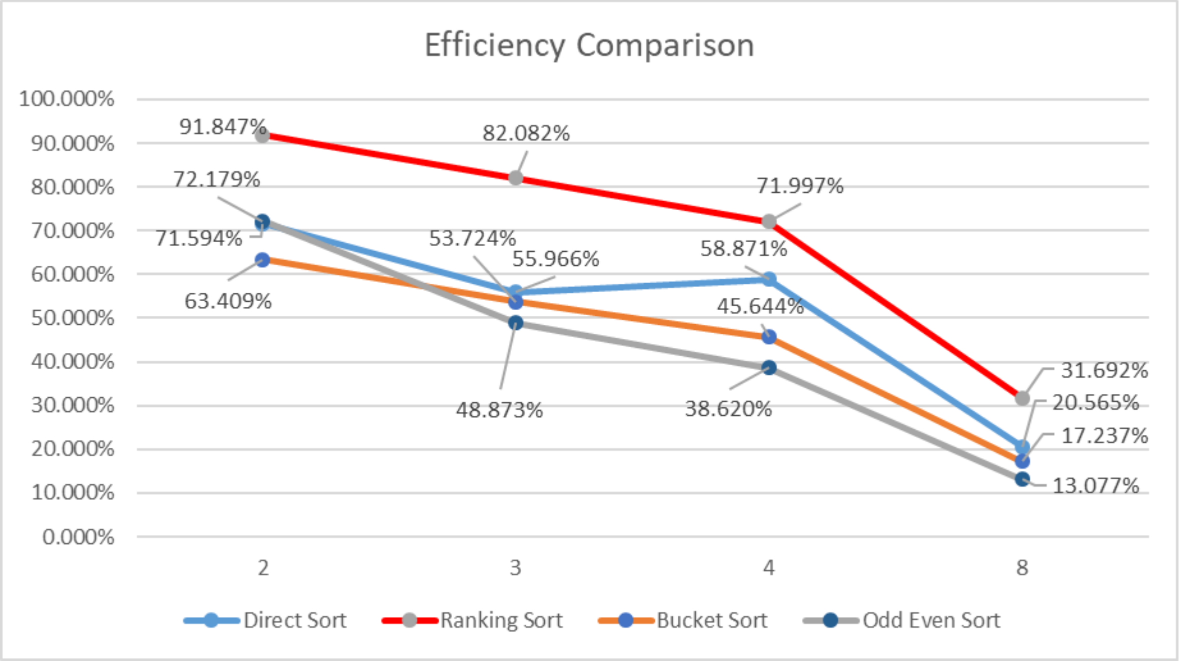
		Direct Sort	Ranking Sort	Bucket Sort	Odd Even Sort
SpeedUp	2 Procs	1.432	1.837	1.268	1.444
Ideal		2	2	2	2
Efficiency		71.594%	91.847%	63.409%	72.179%
SpeedUp	3 Procs	1.679	2.462	1.612	1.466
Ideal		3	3	3	3
Efficiency		55.966%	82.082%	53.724%	48.873%
SpeedUp	4 Procs	2.355	2.880	1.826	1.545
Ideal		4	4	4	4
Efficiency		58.871%	71.997%	45.644%	38.620%
SpeedUp	8 Procs	1.645	2.535	1.379	1.046
Ideal		8	8	8	8
Efficiency		20.565%	31.692%	17.237%	13.077%



**Ranking sort was not included in the same chart because the execution times are way too big compared to the others*



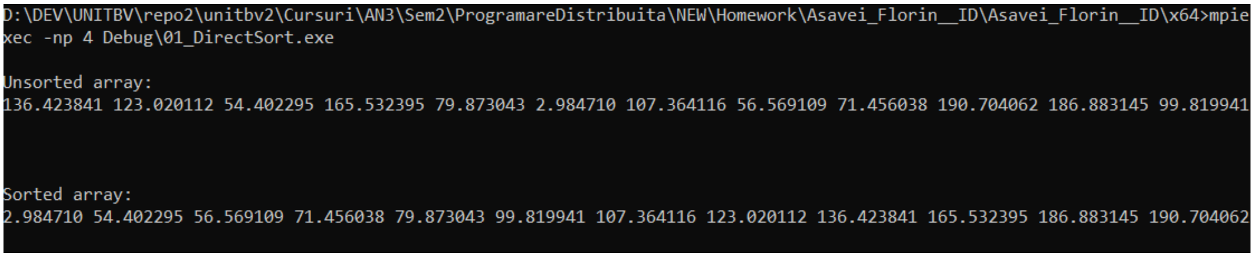

SpeedUp Comparison
3.500
3.000
2.500
2.000
1.500
1.000
0.500
0.000
2
3
4
8
1.837
2.462
2.880
2.535
2.355
1.645
1.612
1.679
1.826
1.444
1.432
1.268
1.466
1.545
1.379
1.046
Direct Sort
Ranking Sort
Bucket Sort
Odd Even Sort


Efficiency Comparison
100.000%
90.000%
80.000%
70.000%
60.000%
50.000%
40.000%
30.000%
20.000%
10.000%
0.000%
2
3
4
8
91.847%
82.082%
71.997%
72.179%
71.594%
63.409%
53.724%
55.966%
58.871%
45.644%
48.873%
38.620%
31.692%
20.565%
17.237%
13.077%
Direct Sort
Ranking Sort
Bucket Sort
Odd Even Sort

Observations:

1. Direct Sort

1.1 Checking that algorithm works (number of elements = 12) with 4 processors



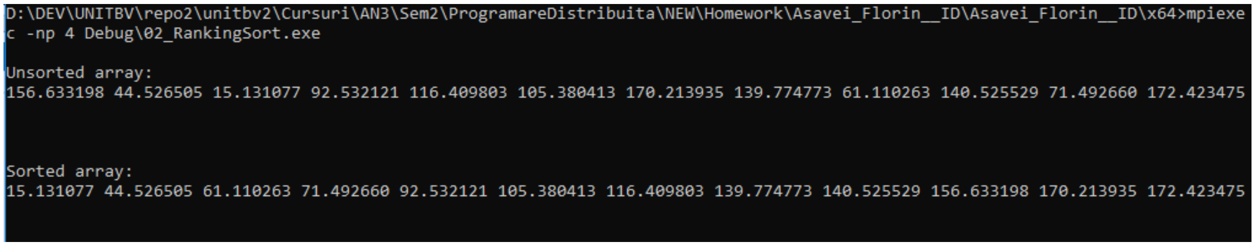
```
D:\DEV\UNITBV\repo2\unitbv2\Cursuri\AN3\Sem2\ProgramareDistribuita\NEW\Homework\Asavei_Florin__ID\Asavei_Florin__ID\x64>mpie
xec -np 4 Debug\01_DirectSort.exe

Unsorted array:
136.423841 123.020112 54.402295 165.532395 79.873043 2.984710 107.364116 56.569109 71.456038 190.704062 186.883145 99.819941



Sorted array:
2.984710 54.402295 56.569109 71.456038 79.873043 99.819941 107.364116 123.020112 136.423841 165.532395 186.883145 190.704062
```

1.2 Performance Observations:

- The execution time decrease by more than half when using 4 processors.
- Best efficiency is achieved when using 2 processors
- Compared to the other 3 algorithms, it is the fastest in all scenarios

2. Ranking Sort

2.1 Checking that algorithm works (number of elements = 12) with 4 processors



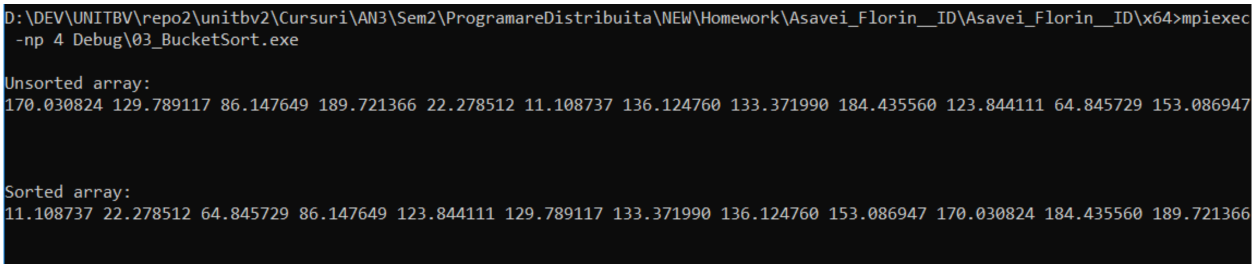
```
D:\DEV\UNITBV\repo2\unitbv2\Cursuri\AN3\Sem2\ProgramareDistribuita\NEW\Homework\Asavei_Florin__ID\Asavei_Florin__ID\x64>mpiexe
c -np 4 Debug\02_RankingSort.exe

Unsorted array:
156.633198 44.526505 15.131077 92.532121 116.409803 105.380413 170.213935 139.774773 61.110263 140.525529 71.492660 172.423475



Sorted array:
15.131077 44.526505 61.110263 71.492660 92.532121 105.380413 116.409803 139.774773 140.525529 156.633198 170.213935 172.423475
```

2.2 Performance Observations:

- This is the slowest algorithm ~ 15.000 times slower than the others
- Execution time is reduced in half just by using 3 processors and it becomes 3 times faster if we use 4 processors
- This algorithm benefits the most out of using Parallel Computing
- It is the only one that achieved more than 90% efficiency when using 2 processors.

3. Bucket Sort

3.1 Checking that algorithm works (number of elements = 12) with 4 processors



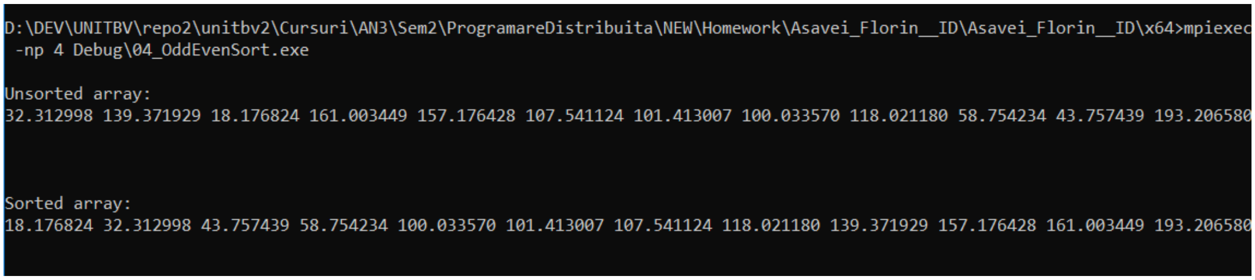
```
D:\DEV\UNITBV\repo2\unitbv2\Cursuri\AN3\Sem2\ProgramareDistribuita\NEW\Homework\Asavei_Florin__ID\Asavei_Florin__ID\x64>mpiexec
 -np 4 Debug\03_BucketSort.exe

Unsorted array:
170.030824 129.789117 86.147649 189.721366 22.278512 11.108737 136.124760 133.371990 184.435560 123.844111 64.845729 153.086947



Sorted array:
11.108737 22.278512 64.845729 86.147649 123.844111 129.789117 133.371990 136.124760 153.086947 170.030824 184.435560 189.721366
```

3.2 Performance Observations:

- Second fastest algorithm (after Direct Sort)
- The speedup is most linear out of all algorithms
- The load is not evenly distributed, there are significant differences in processor execution times, and since we have to consider the program finished when the last processor finishes sorting his bucket, the overall timing increases

4. Odd Event Sort

4.1 Checking that algorithm works (number of elements = 12) with 4 processors



```
D:\DEV\UNITBV\repo2\unitbv2\Cursuri\AN3\Sem2\ProgramareDistribuita\NEW\Homework\Asavei_Florin__ID\Asavei_Florin__ID\x64>mpiexec
 -np 4 Debug\04_OddEvenSort.exe

Unsorted array:
32.312998 139.371929 18.176824 161.003449 157.176428 107.541124 101.413007 100.033570 118.021180 58.754234 43.757439 193.206580



Sorted array:
18.176824 32.312998 43.757439 58.754234 100.033570 101.413007 107.541124 118.021180 139.371929 157.176428 161.003449 193.206580
```

4.2 Performance Observations:

- The second slowest algorithm out of the 4
- It the lowest overall efficiency
- Best efficiency achieved when using 2 processors
- Not much improvement is achieved if we use more than 2 processors
- Processor load is distributed evenly, there is not that much difference between the times

Overall Observations:

All algorithms resulted in better execution times when using multi-processing.

It seems that the best Efficiency is achieved when using 2 processors.

Speedup and Efficiency decreases when we oversubscribe (using 8 processors)