Parallel Radix Sort Algorithm with MPI

Implementation and Analysis

Author: Yourii Martiak

New York University

December 20, 2012

Introduction

Among many problems in computer science, sorting is probably most common. Due to its applicability to different domains in the field, sorting algorithms are often used in the majority of business and scientific applications. Many different serial sorting algorithms have been developed over the years and many have become standards, fine-tuned for use in specific scenarios. Yet, because of this wide range of applications, the need for optimization and speed has always been in place. With the emergence of multi-core processors, many of the serial algorithms have been successfully parallelized and some new algorithms were developed. As a result, processing time have been reduced with varying success. The reality is that some algorithms are simply easier to parallelize than others.

The goals of this paper are to look at the details of implementation a parallel sorting algorithm and analyze the parts where the efficiency and bottlenecks come from, as they relate to overall performance common to most parallel algorithms.

The two algorithm implementations selected for this purpose:

1. Serial Radix Sort
2. Parallel Radix Sort

Implementation

Both algorithms are implemented in C programming language. Parallel Radix Sort is also using Message Passing Interface (MPI) to accomplish inter-process communications.

Serial radix sort was implemented as follows:

- For each pass scan g consecutive bits from LSD
- Store keys in 2^g buckets according to g bits
- Count how many keys each bucket has
- Compute exclusive prefix sum for each bucket
- Assign starting address according to prefix sums
- Examine g bits to determine bucket and move key to that bucket

As a rule, any parallel sort implementation involves running multiple sections of regular sort algorithm on multiple processors in parallel. When each processor is done, the results are communicated between the processors, and this pattern maybe repeated multiple times until the whole sequence is completely sorted.

Parallel radix sort implementation is similar to the above with a few exceptions. First, keys must be stored and moved across different processors. To accomplish this task, MPI is used for inter-process communication. As a result of the above, each processor can end up having varying number of keys to manage after each, which requires additional work to be done for keeping track of these moves. Here are the steps to parallel radix sort implementation:

- Split initial problem set into multiple subsets and assign to different processors
- Count number of keys per bucket by scanning g bits every pass (local operation)
- Move keys within each processor to appropriate buckets (local operation)

- 1-to-all transpose buckets across processors to find prefix sum (global operation)
- Send/receive keys between the processors (global operation)

Performance Implications

Parallel sort performance then depends on the following factors:

- total number of keys
- total number of processors (number of partial sequences to be sorted by each individual processor)
- time spent sorting local sequence by each processor
- time spent for processor communication

Considering the above points, we can see that to optimize parallel sorting algorithm, it is important to partition the individual sequences in a way to ensure balanced processor workload and to avoid any processor idling time while waiting for communications to be completed. At the same time, communication needs to be done as efficiently as possible. The latter can be affected by the number of nodes, frequency of communication, message payload and latency on the network. Previous research on this subject indicated that for a large number of keys, communication times take significant part of the total execution time.

Scalability of parallel radix sort is not without limits. The main limiting factor becomes the communication part, mainly the bandwidth and the latency of the network. This is simply because with adding more nodes for greater parallelization and ability to handle larger problem sizes, there will be a point of over saturation and message contention due to the above limitations.

Performance Results

Both algorithms were tested on the same hardware, having four processors (two hyper-threaded real cores) each running at 2.3GHz clock speed with 3MB cache per core. Machine has 4GB of random access memory.

Tests were done for different problem sizes, namely 1,000 to 10,000,000 numbers increasing each time by a factor of 10. A set of tests was done on code compiled without optimization flags and then the same test set was repeated for compilation with -O3 flag turned on. Also, varying number of MPI processes and different number of bit-sampling for each algorithm pass were used. The results are listed at the end of this document.

Parallel radix sort achieved a max speedup of 1.8 over the serial radix sort, close to our expectations. Some things came as surprise. Optimization flag had little effect on parallel algorithm performance, whereas it helped to greatly improve performance of the serial radix sort algorithm.

Conclusion

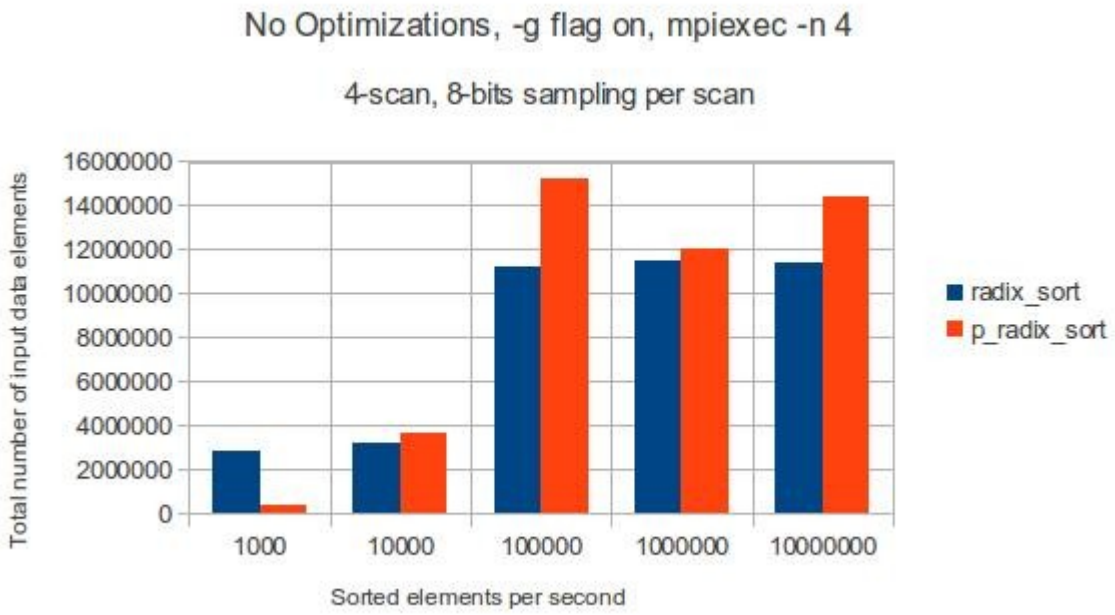
As per results of all benchmarks, it became apparent that parallel radix sort performance is sub-par for small problem sizes. However, it gets better and quickly surpasses the serial radix sort speed as the problem size grows, while performance of serial algorithm goes down.

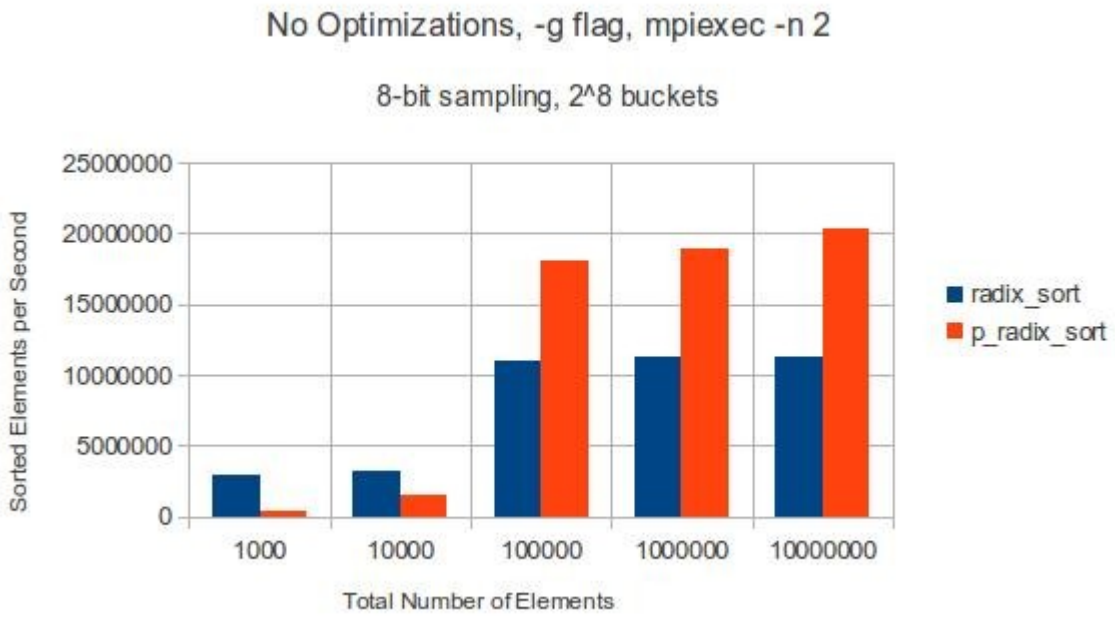
Maximum speedup of 1.8 over the serial version was achieved using 8-bit sampling having number of MPI processes equal number of processors on the machine and having large enough problem size.

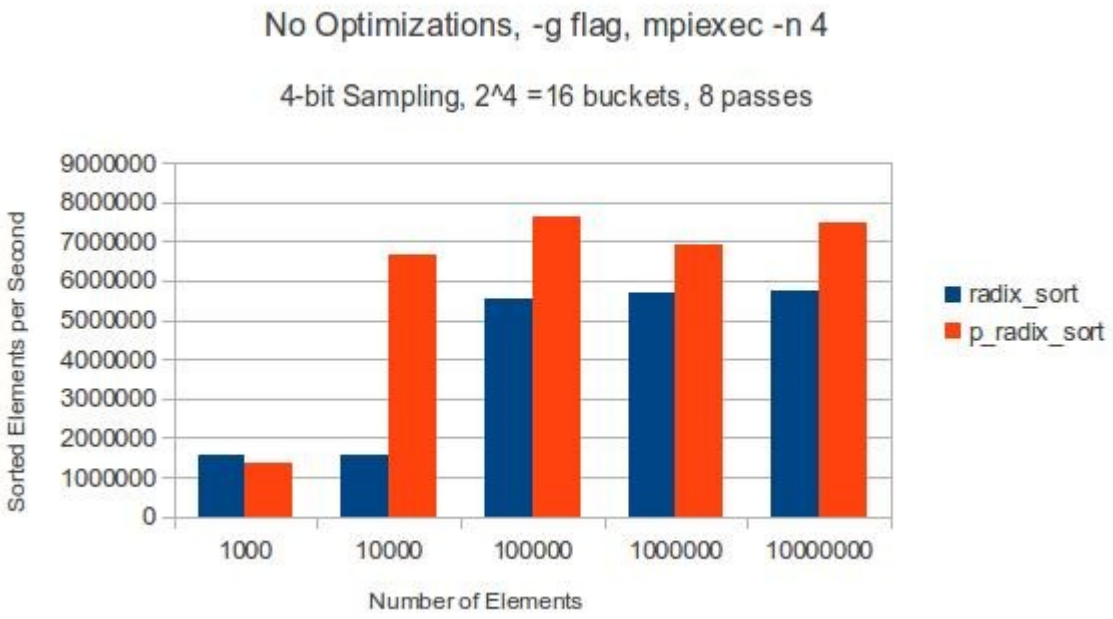
Using 8-bit sampling per pass seems to work best to achieve balance between local processor work and the message passing overhead. Minimizing the number of buckets per processor (by reducing sampling to 4-bit) appeared to be counter productive due to increase in payload size per message (more keys per bucket that need to be sent across).

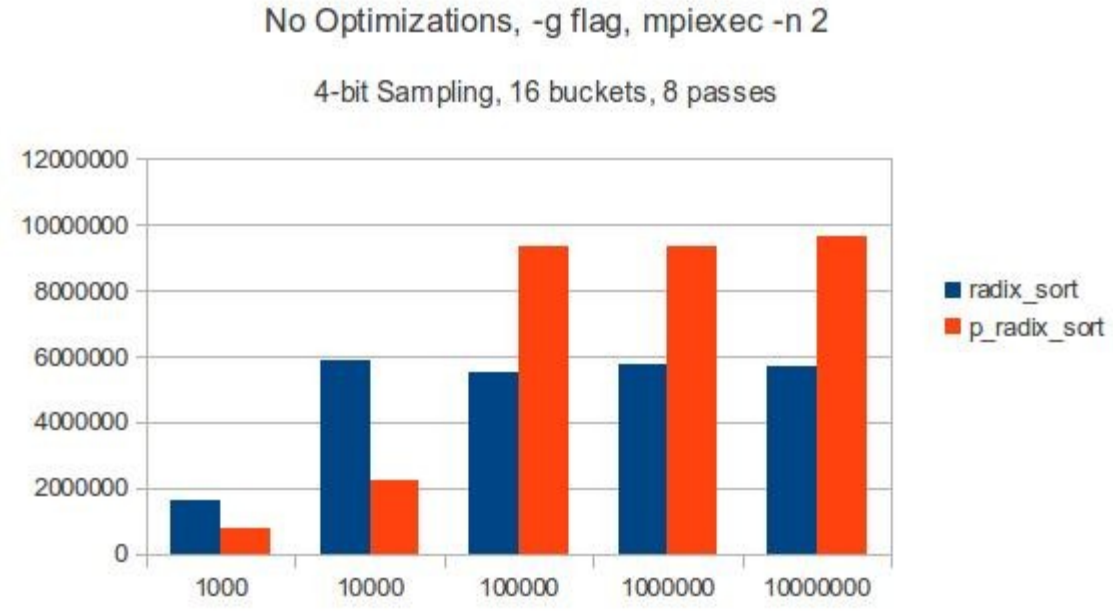
Increasing problem size while scaling the number of processors results in better performance up to the point of increased number of messages becoming a bottleneck and negating any performance gain that was achieved by parallelization.

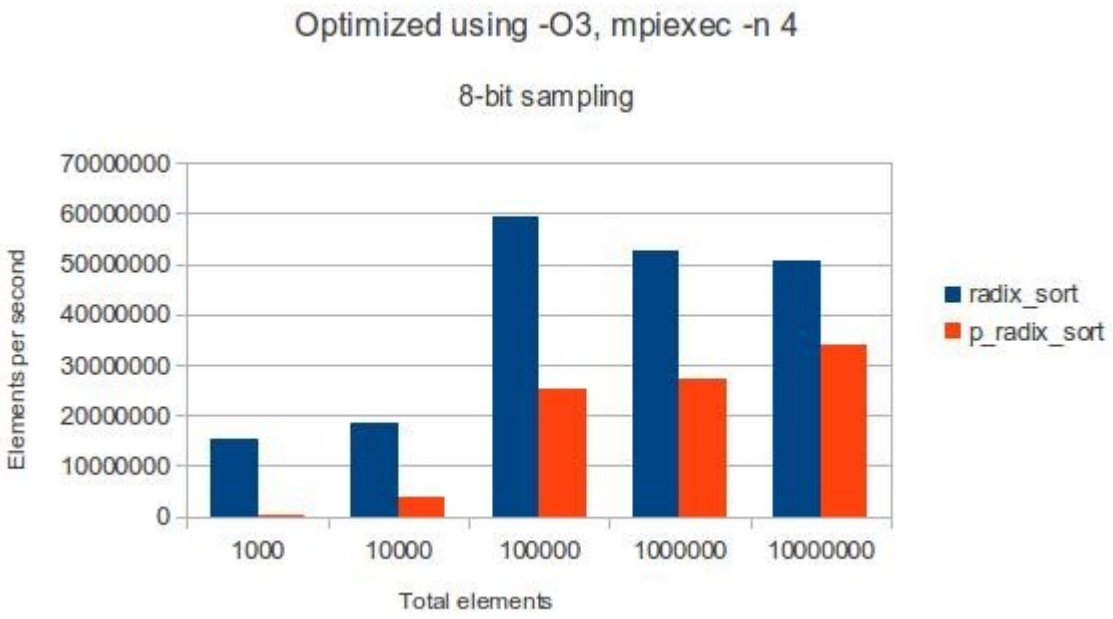
Performance Data

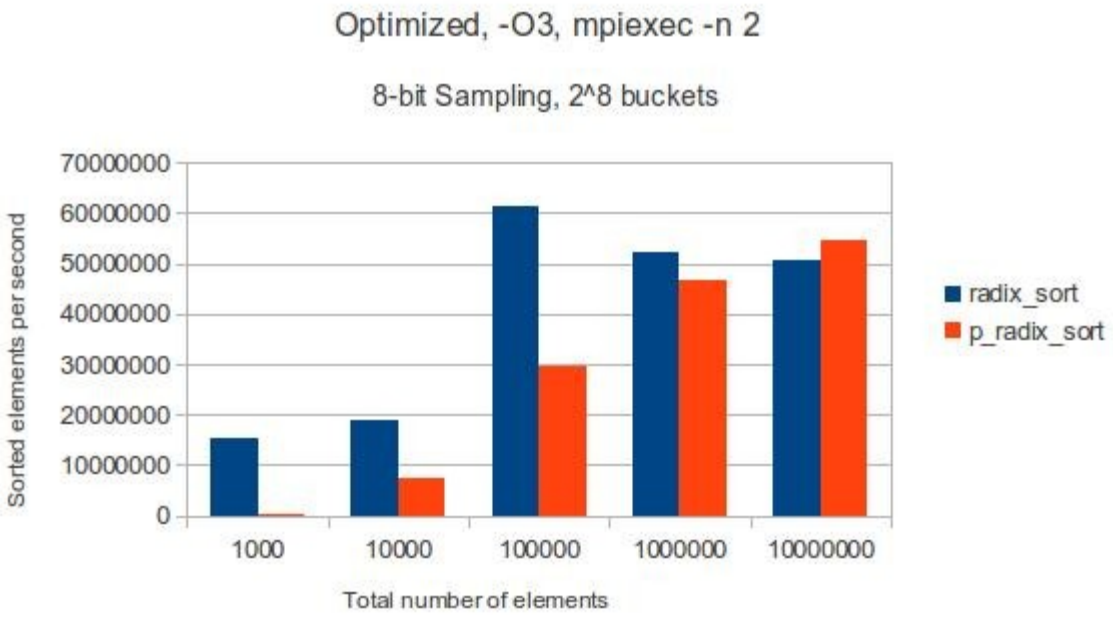


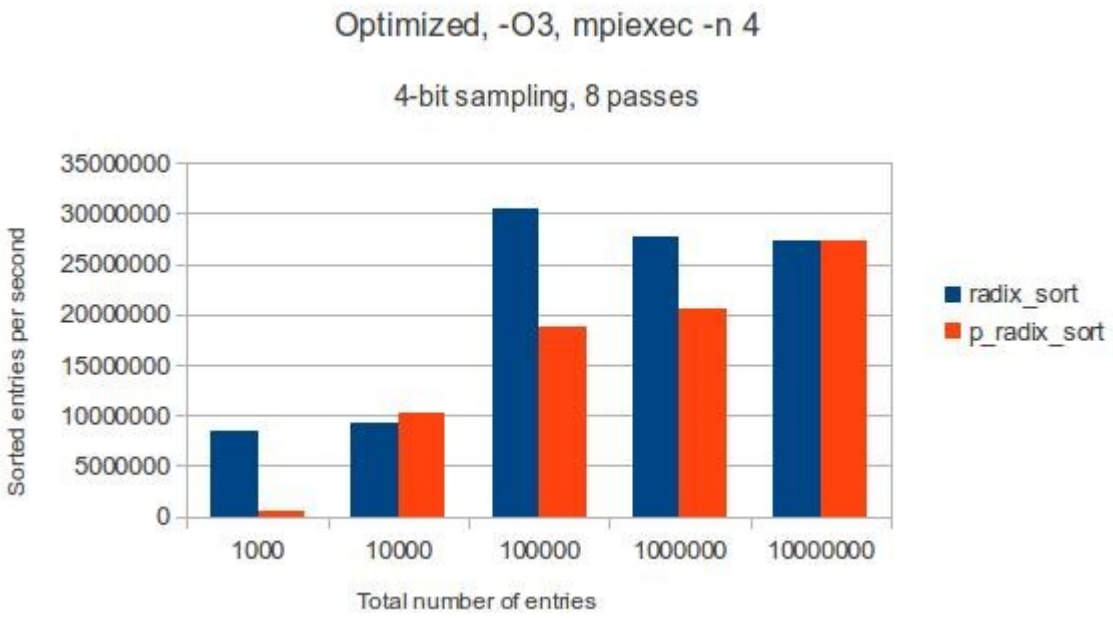


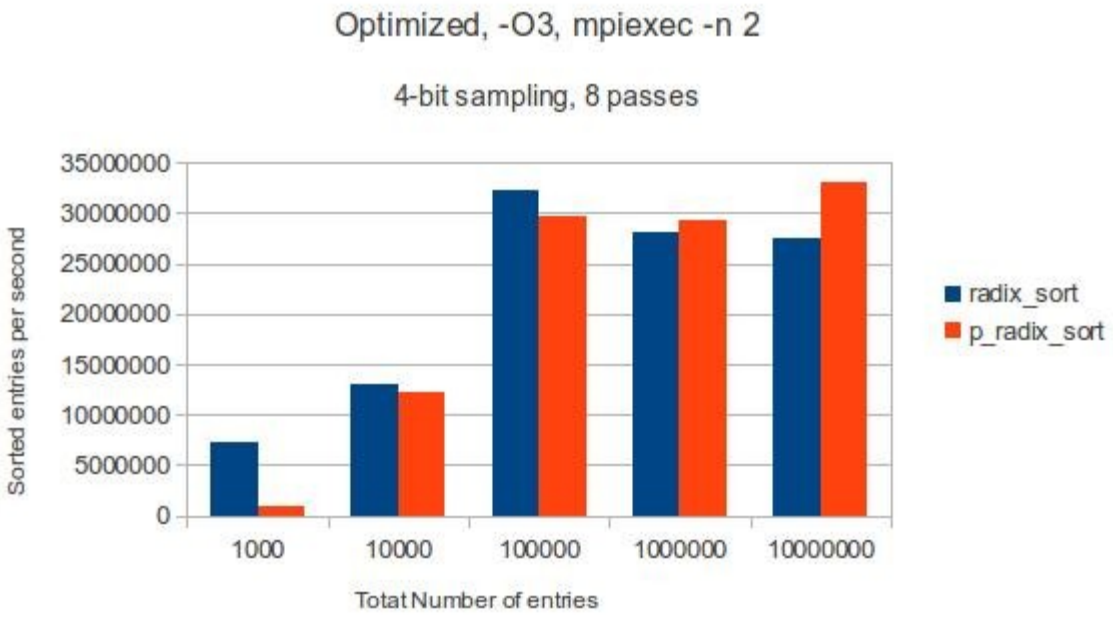

No Optimizations, -g flag, mpiexec -n 4
4-bit Sampling, 2^4 =16 buckets, 8 passes
Sorted Elements per Second
9000000
8000000
7000000
6000000
5000000
4000000
3000000
2000000
1000000
0
1000
10000
100000
1000000
10000000
Number of Elements
radix_sort
p_radix_sort


No Optimizations, -g flag, mpiexec -n 2
4-bit Sampling, 16 buckets, 8 passes
12000000
10000000
8000000
6000000
4000000
2000000
0
1000
10000
100000
1000000
10000000
radix_sort
p_radix_sort


Optimized using -O3, mpiexec -n 4
8-bit sampling
Elements per second
70000000
60000000
50000000
40000000
30000000
20000000
10000000
0
1000
10000
100000
1000000
10000000
Total elements
radix_sort
p_radix_sort


Optimized, -O3, mpiexec -n 2
8-bit Sampling, 2^8 buckets
Sorted elements per second
70000000
60000000
50000000
40000000
30000000
20000000
10000000
0
1000
10000
100000
1000000
10000000
Total number of elements
radix_sort
p_radix_sort


Optimized, -O3, mpiexec -n 4
4-bit sampling, 8 passes
Sorted entries per second
35000000
30000000
25000000
20000000
15000000
10000000
5000000
0
1000
10000
100000
1000000
10000000
Total number of entries
radix_sort
p_radix_sort


Optimized, -O3, mpiexec -n 2
4-bit sampling, 8 passes
Sorted entries per second
35000000
30000000
25000000
20000000
15000000
10000000
5000000
0
1000
10000
100000
1000000
10000000
Totat Number of entries
radix_sort
p_radix_sort

Source Code and Instructions

Source code for this project is freely available and can be downloaded from:

https://github.com/ym720/p_radix_sort_mpi/tree/master/p_radix_sort_mpi

After downloading the project, run make command in the project directory to build the files using Makefile provided with the project. The code for this project was built and tested on Ubuntu Linux 12.04 using the following tools and packages:

gcc (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3

mpiexec (OpenRTE) 1.5.4

Both - serial and parallel radix sort input should be a file with the list of numbers listed one number per line. To generate a sample test file with random numbers, dataset_gen program is provided with the project, run dataset_gen for usage instructions.

For the test, included run_test.sh bash script can be used. The script runs serial followed by a parallel version for a set of different problem sizes from 1,000 to 10,000,000.

References

1. Wikipedia – Radix Sort: http://en.wikipedia.org/wiki/Radix_sort
2. Robert Sedgewick, Algorithms in C, Addison-Wesley Publishing Company, 1990
3. Shin-Jae Lee, Minsoo Jeon, and Dongseung Kim, Partitioned Parallel Radix Sort, 2002: http://ebookbrowse.com/2002-09-partitioned-parallel-radix-sort-pdf-d54143025
4. Andrew Sohn, Yuetsu Kodama, Load Balanced Parallel Radix Sort, 1998: https://docs.google.com/open?id=13nlm1wEgYIrHJxdHTwa8tXptK_RjpzZP7ZnUx3pVjGeb5kp2qZS4GseSq1JG
5. Miguel Palhas, Parallel sorting algorithm implementation in OpenMP and MPI, 2012: https://github.com/naps62/parallel-sort