Parallel Radix Sort with MPI

Yourii Martiak

Why sorting?

- One of the most common problems in computer science
- Applicable to different domains in the field
- Variety of serial sorting algorithms available

Sorting evolution

- Emergence of multi-core hardware prompted serial algorithm parallelization, although with varying success
- Some algorithms are easier to parallelize than others
- New parallel sorting algorithms were developed

Parallel Sorting Basics

- Split unsorted sequence into equal size partitions and distribute across multiple processors
- Run serial sorting algorithm on each partition in parallel
- When each processor done sorting its own data, communicate results with other processors
- Repeat multiple times until the whole sequence becomes sorted

Performance Factors

- The size of input data set
- Number of processors (number of partial sequences partitioned across processors)
- Time spent sorting each individual sequence
- Time spent on inter-processor communication
- Previous research shows that for large problem sets communication becomes major perfomance bottleneck

Radix Sort

- Non-comparative
- Sorts data by evaluating one of group of digits at a time
- Not limited to integers
- MSD and LSD variety
- Time complexity O(k*n) for n keys each having k or fewer digits
- In many cases an improvement over comparative sort for large data sets

Radix Sort Example

Unsorted sequence {170, 45, 75, 90, 802, 24, 2, 66}

LSD Pass 1

```
[0]  170  90   .
[1]
[2]  802  2    .    .
[3]
[4]  24   .    .    .
[5]  45   75   .    .
[6]  66   .    .
```

Continue until all digits sorted ...

Radix Sort Implementation

- P - number of processors
- n - problem size (total number of keys)
- g - group of bits examined during each pass
- b - number of bits for a number (32-bit int)
- r - number of passes (b / g)
- B - number of buckets, 2^g

Radix Sort Implementation

- 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

Parallel Radix Sort

- Similar to serial radix sort algorithm
- Big difference is that keys are stored across different processors
- Keys are moved across different processors
- Each processor can end up having varying key counts after each pass
- Given P processors and B buckets, each processor holds B / P buckets

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)

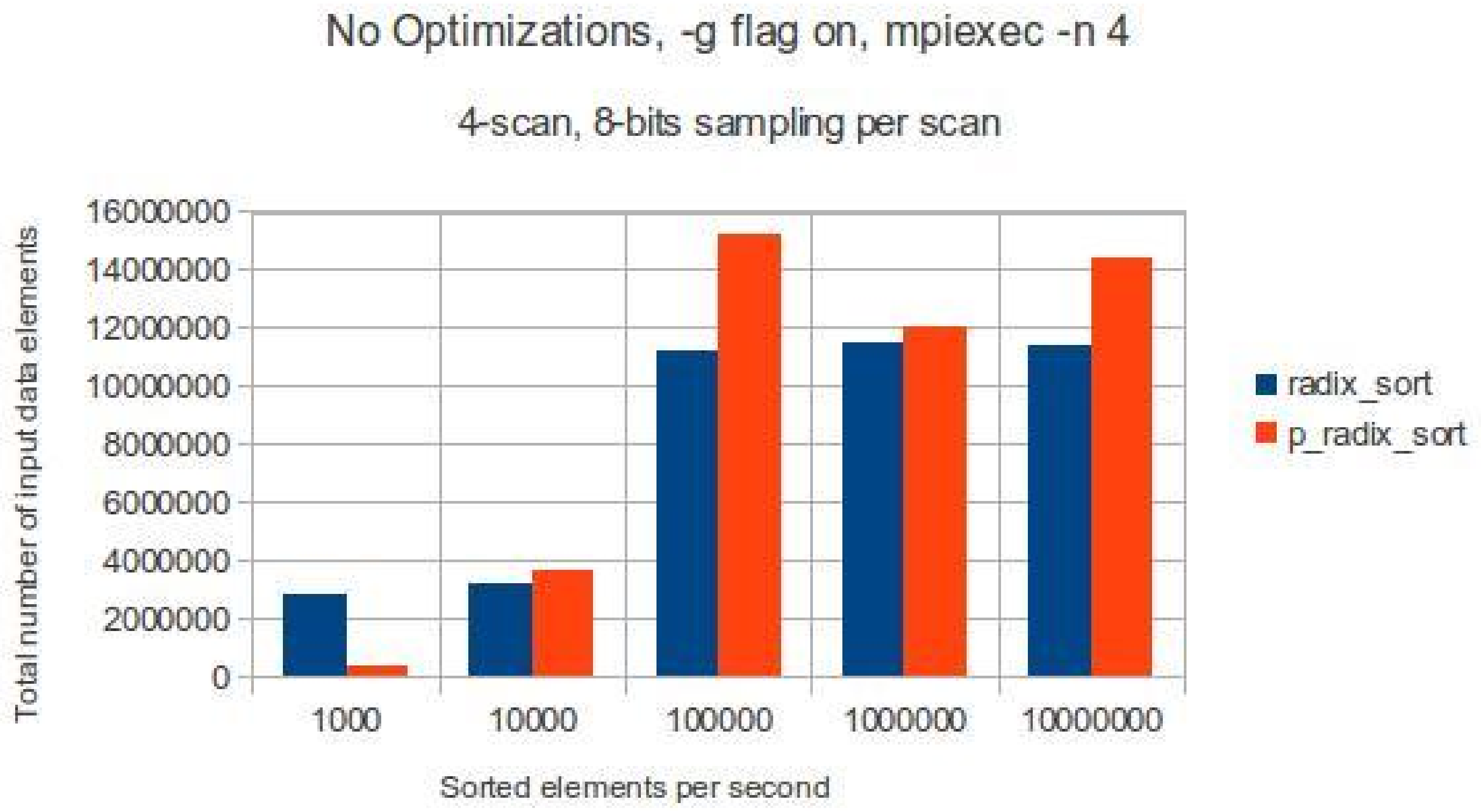
Parallel Radix Sort Bucket Counts Transpose

	B0	B1	B2	B3
P0	1	3	4	2
P1	3	6	1	0
P2	0	3	5	2
P3	1	2	2	5
P0	1	3	0	1
P1	3	6	3	2
P2	4	1	5	2
P3	2	0	2	5

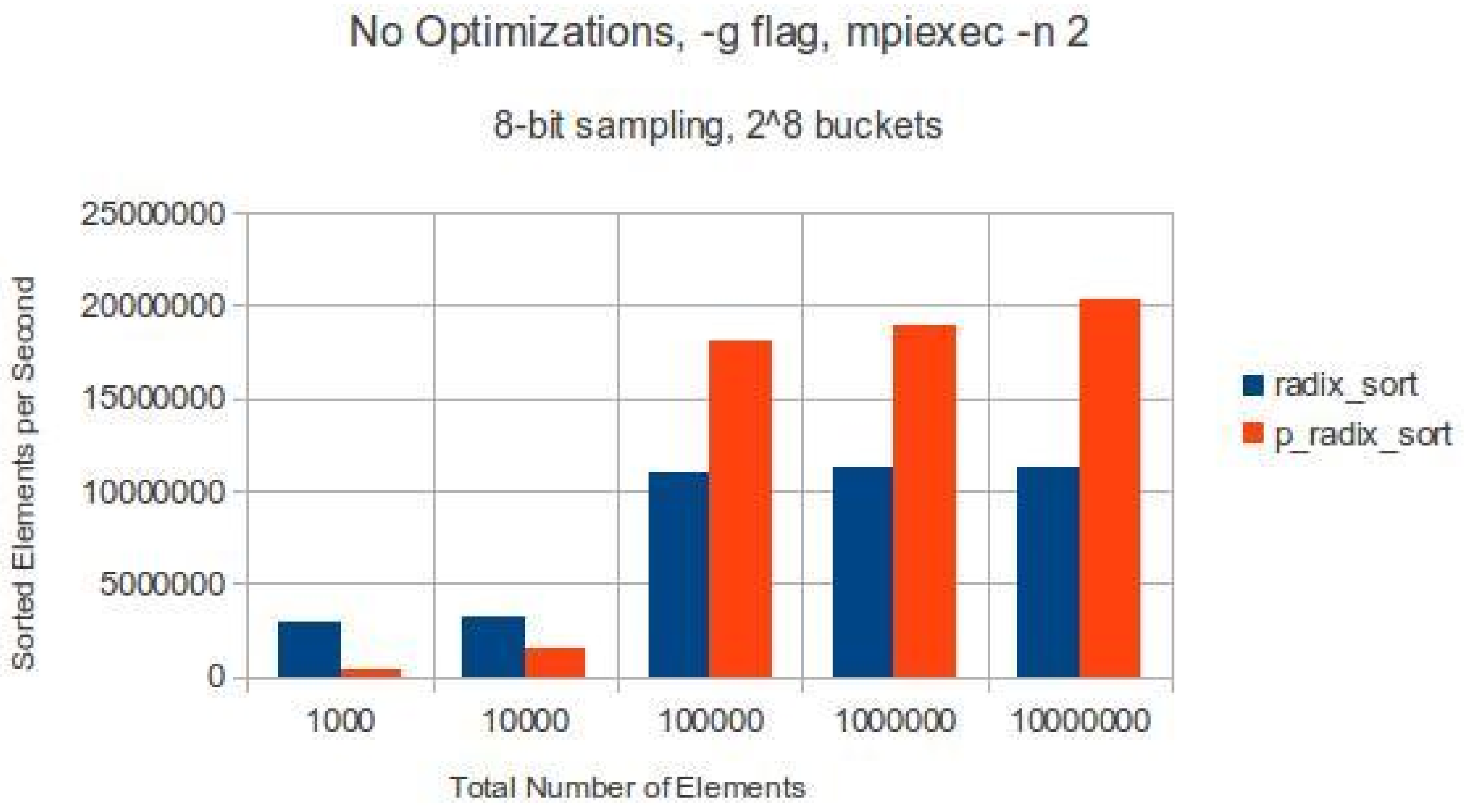
Parallel Radix Sort Sending Keys

- As the last step, processors communicate keys according to global map
- Sending keys done according to map before transpose
- Receiving done according to mapping after transpose
- Keys are stored according to new mapping
- Continue until all passes are done
- At the end, collect keys from all and print by master process

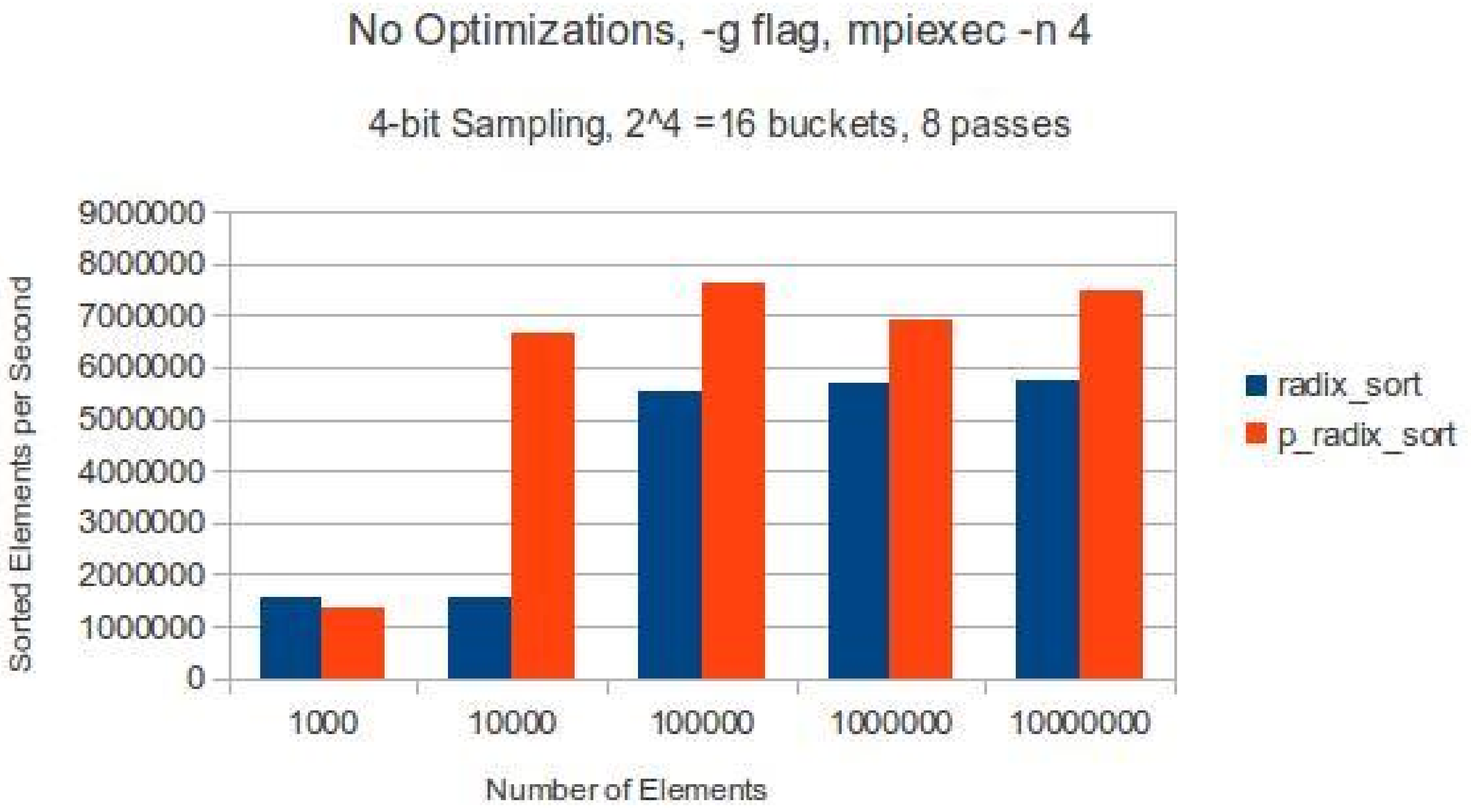
Test Results



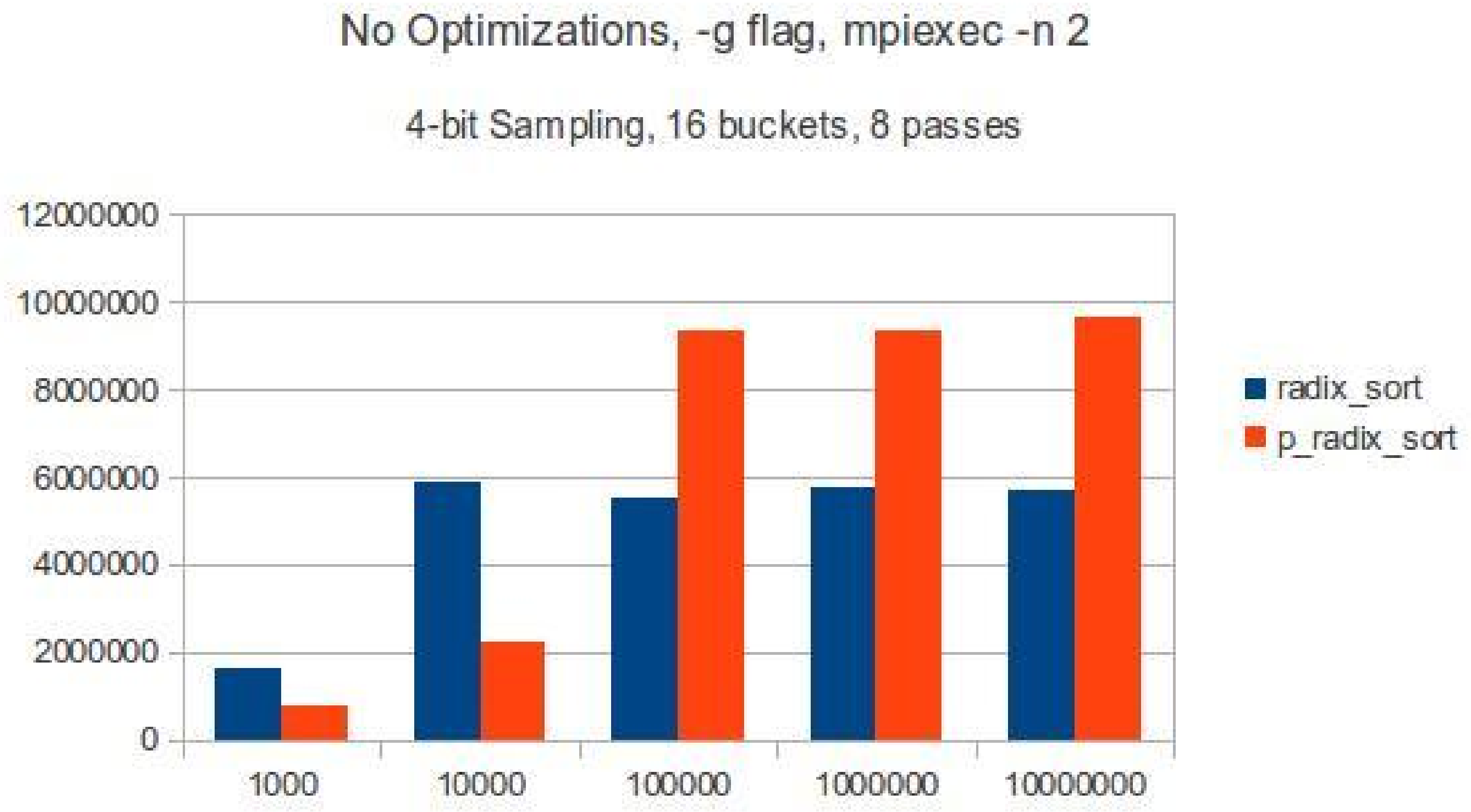
Test Results



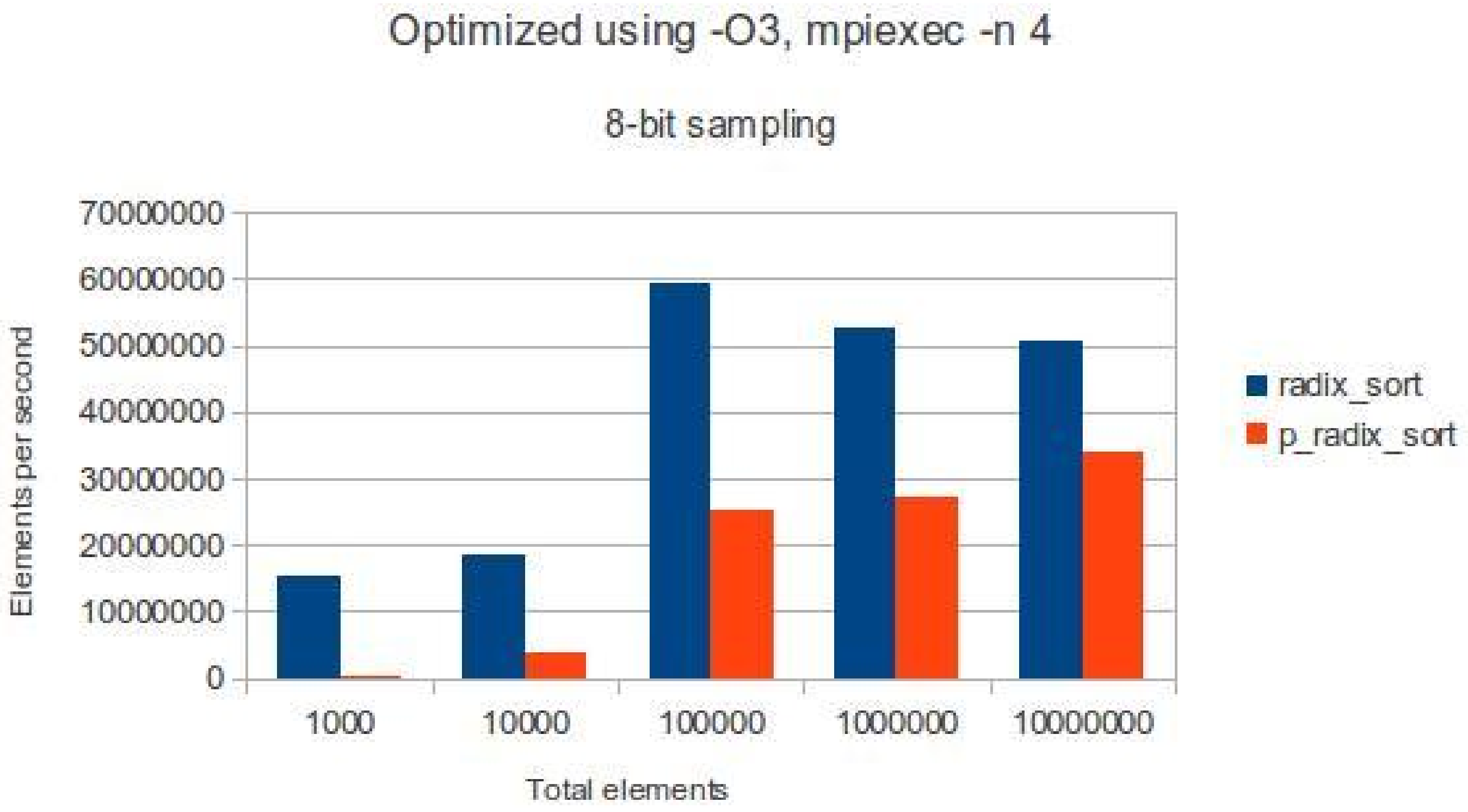
Test Results



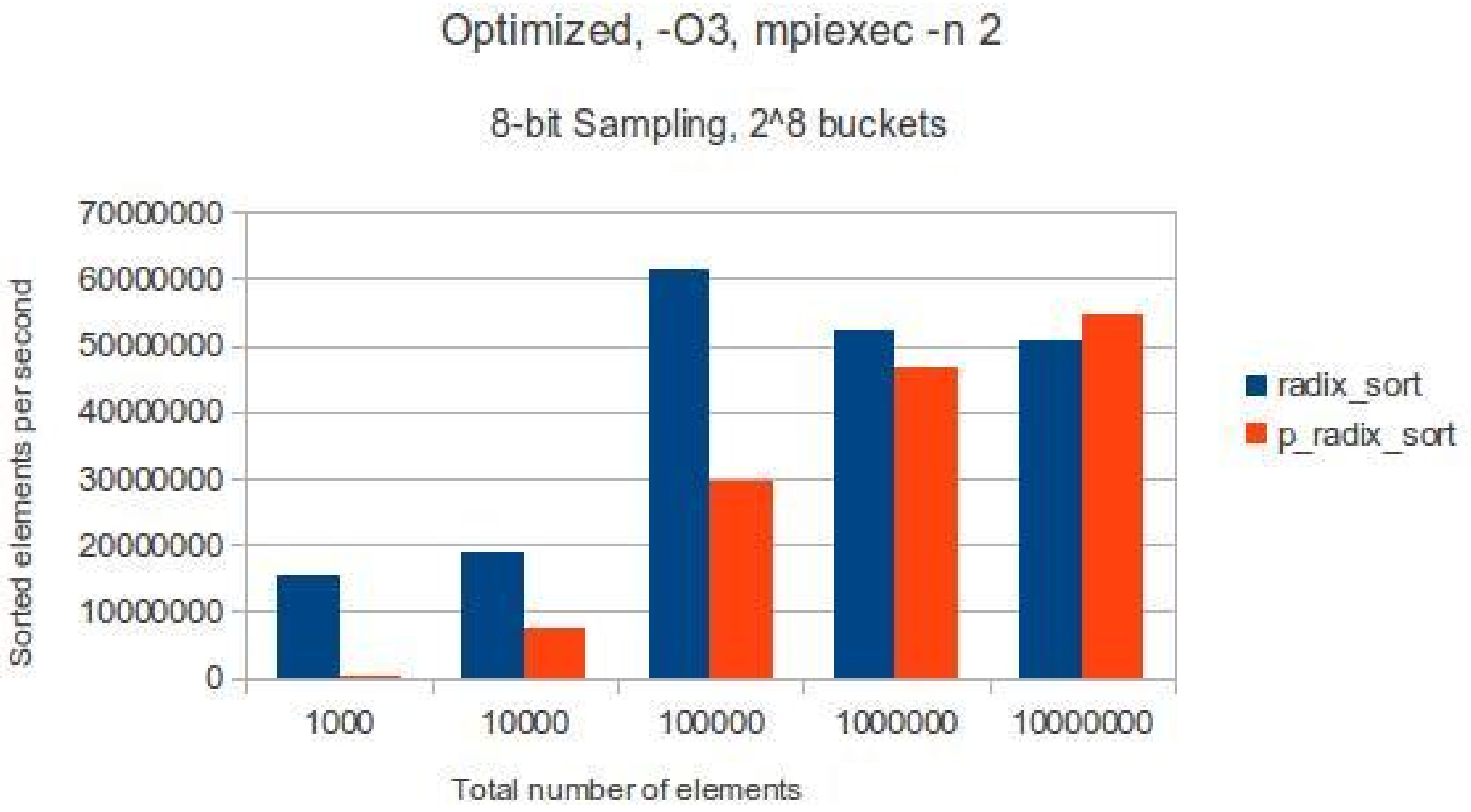
Test Results



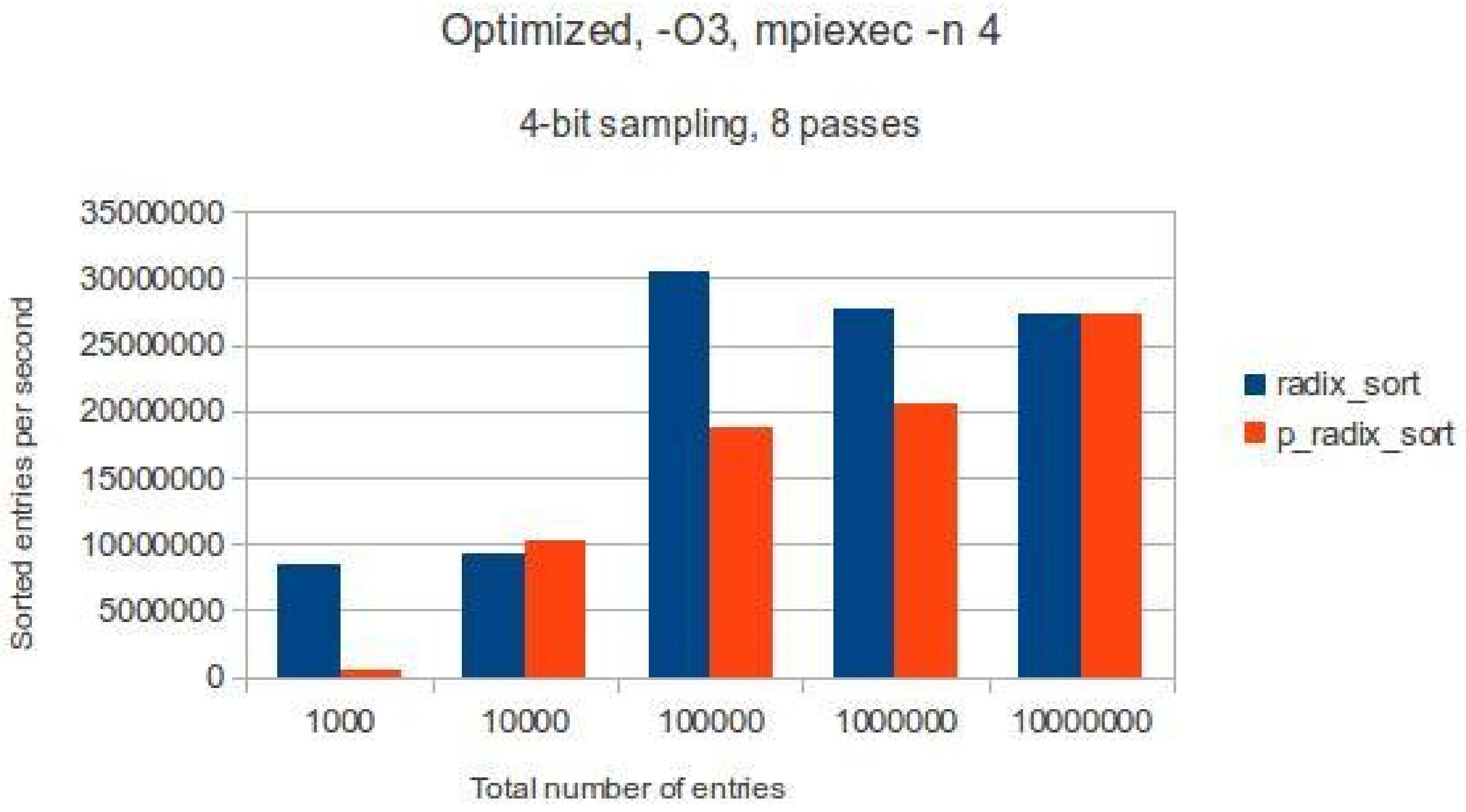
Test Results



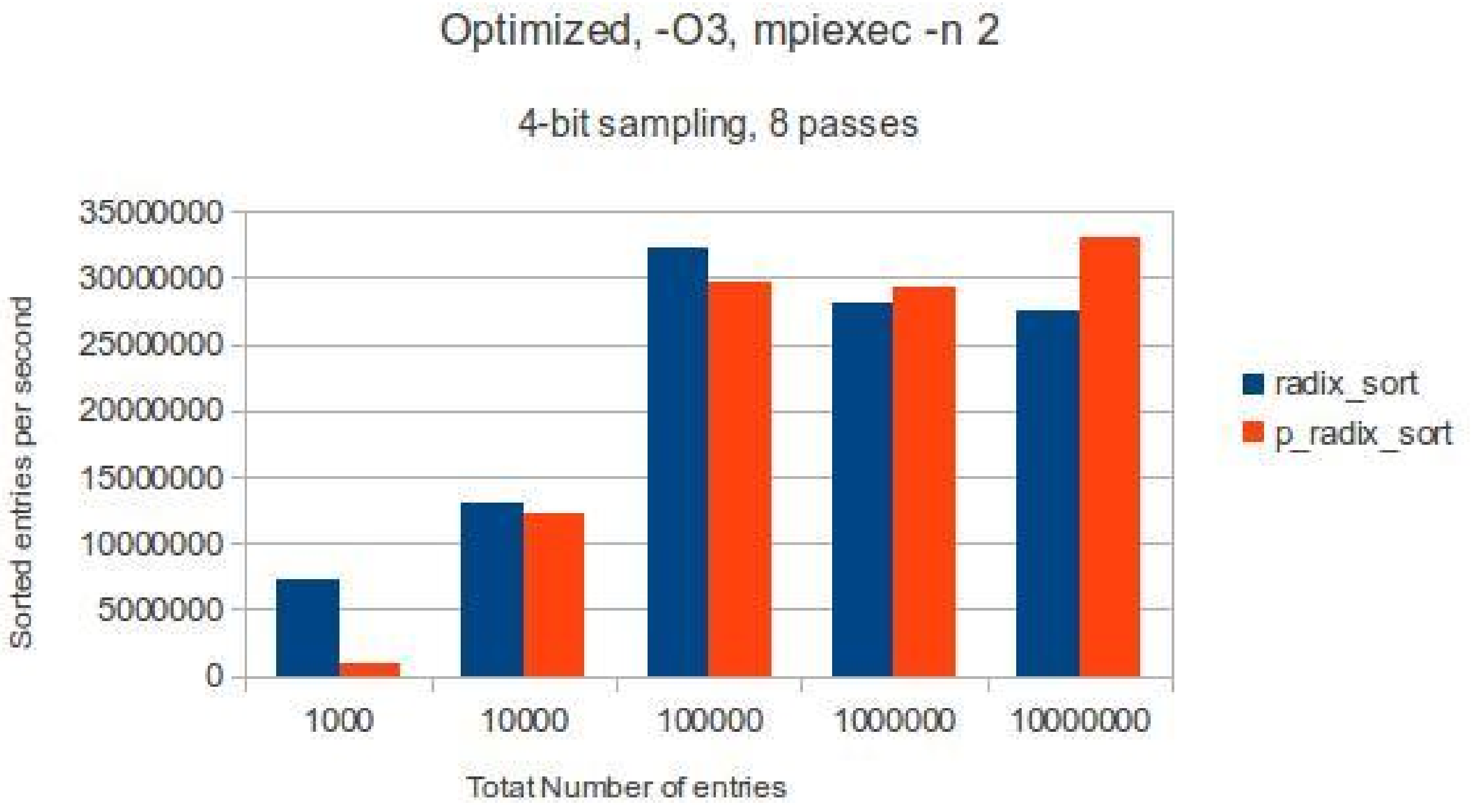
Test Results



Test Results



Test Results



Conclusions

- As per results of all benchmarks, it is apparent that parallel radix sort performance suffers for small problem sizes. However, it gets better as the problem size grows, while performance of serial algorithm goes down
- Best speedup of 1.8 over the serial version was achieved using 8-bit sampling, mpiexec -n equal number of processor and having large enough problem size

Conclusions

- Using 8-bit sampling per pass seems to work best to achieve balance between local processing and messaging overhead
- Minimizing number of buckets per processor appears to be counterproductive due to increase in payload size per message with keys that needs to be communicated across

Conclusions

- Optimizations do little for the MPI implementations, again due to overhead created by messaging whereas serial version benefits greatly from -O3 optimization
- Using mpiexec -n equal to number of processors provides best results (common sense)
- Performance only gets better with more processors and bigger problem sizes

The End

Questions?