Parallel Sample Sort using MPI

Nicolas Barrios

Introduction to Sorting

Divide and conquer sorting algorithms are extremely prevalent throughout the field of Computer Science. Quicksort, Merge Sort, etc.

They lend themselves nicely towards parallelization, as each subproblem can be distributed across processors.

However, without guarantees about the data in each processor, the runtimes of these algorithms deteriorate when encountering a disproportionate amount of data in some processor(s).

Sample Sort is an improved version of divide and conquer algorithms that addresses this non-uniformity. Consider it a generalization of Quicksort.

As the name suggests, it **samples** the data to more accurately partition the data.

How does Samplesort work?

Where p = # processors and k = oversampling factor:

1. Sample *p* (* *k*) elements and sort them
2. Share these samples with every processor (MPI_Allgather)
3. Each *p* select *p-1* pivots aka splitters. These are the same across *p*'s.
    a. Each *b-th* pair of splitters denotes a "bucket" that will be sent to the *b-th* processor.
4. Re-arrange local data into the buckets described by the pivots.
5. Send the *b-th* bucket to the *b-th* processor (MPI_Alltoall[v])
6. Combine buckets and sort local data.

Oversampling?

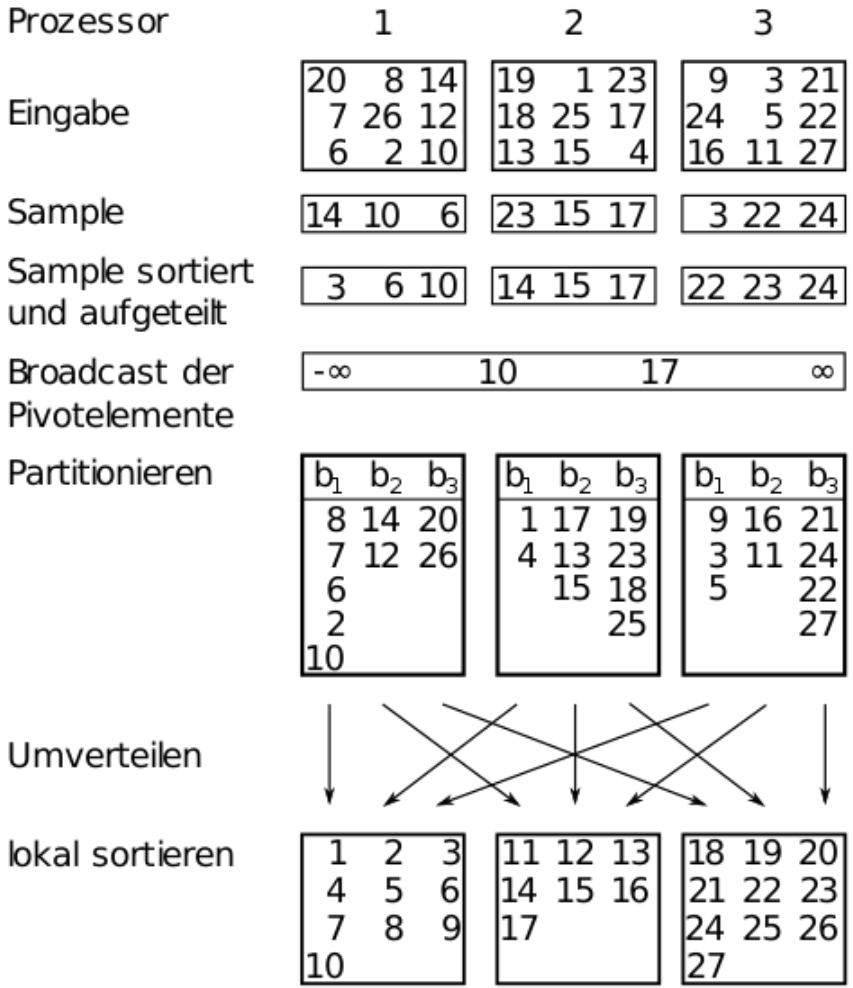
The oversampling factor comes in as a way to get a good representation of the full dataset. A large oversampling factor will result in many samples from which to choose from. This comes at the expense of an increased communication cost to get those samples to all other processors.

Therefore, there is a diminishing return when increasing the oversampling factor

An Example of Samplesort

From: [Wikipedia's Samplesort Page]

Sorry about the non-English image, but it's a great visual



My Implementation of Samplesort

Ran on CPU-Gold-6230 on CCR:

- Using [2,4,8,12,16,20,24,28,32] amount of cores for strong and weak scaling studies
- Using [2,4,8,12,16] amount of nodes for the oversampling factor study

Generate [node count *] 48 million floats total, ranging from [-1000,1000]

The number of splitters is exactly (node count - 1).

The sorting routine used at the processor level is a variation of Quicksort called Introsort implemented in the C++ STL. This algorithm has an O(n log n) runtime.

So What's New?

- Added higher granularity in timing the program
- Went from using a node's core to using multiple nodes (on CPU-Gold-6230)
- Added weak scaling data analysis
- Added dimensionality to the data collection suite by varying oversampling factor
- Reworked calculations to use the runtime of nodes=2 as T1 in speedup computations
- Added correctness checking at a local and global scale, with respect to the nodes used.
- Reduction of unnecessary computations for smaller runtimes
- Decided against OpenMP usage due to use of C++ vectors (they are not thread-safe)

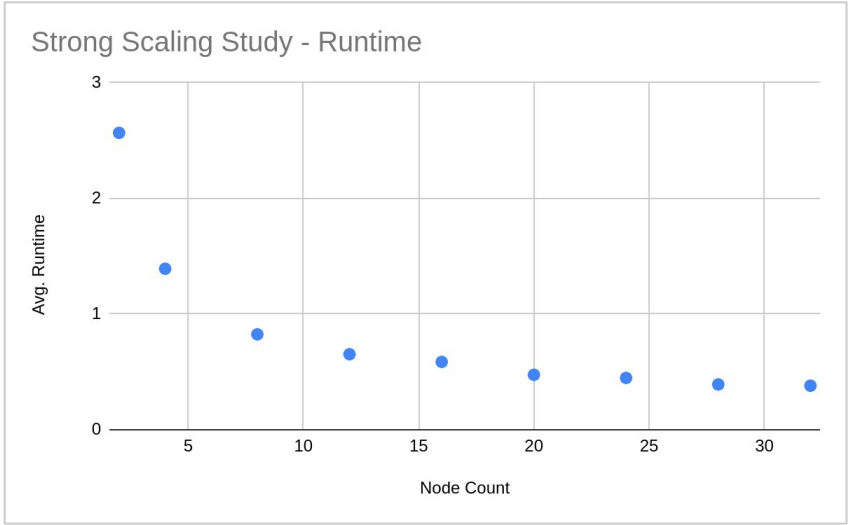
Strong Scaling Study

All runs done with:

- Problem Size of 48 million elements
- Range of elements: [-1000,1000]
- Oversampling Factor of 4

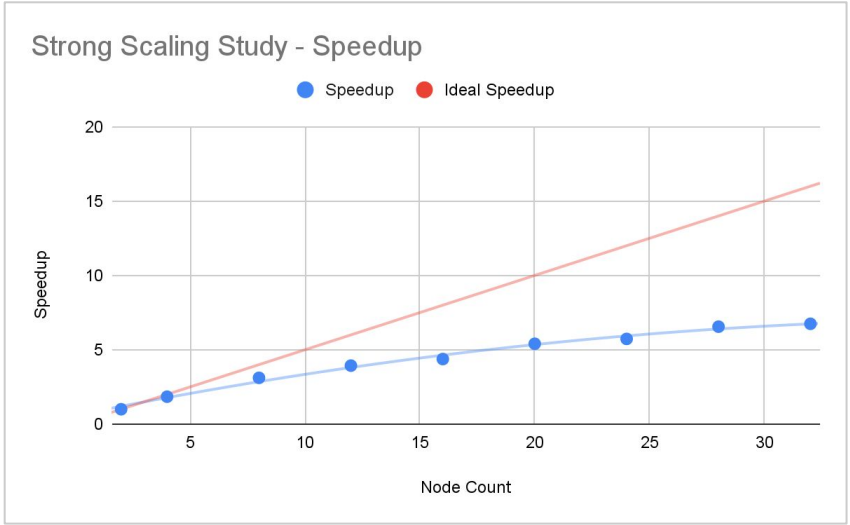
Runtimes

Node Count	Avg. Runtime	Stddev
2	2.566841311	1.768700598
4	1.392023328	0.7411341783
8	0.8247090096	0.3014132983
12	0.6525438199	0.1530763413
16	0.5865258697	0.1536956792
20	0.475012734	0.1209308714
24	0.4478911915	0.1177897571
28	0.3916956969	0.07176715147
32	0.3804872109	0.06831204493



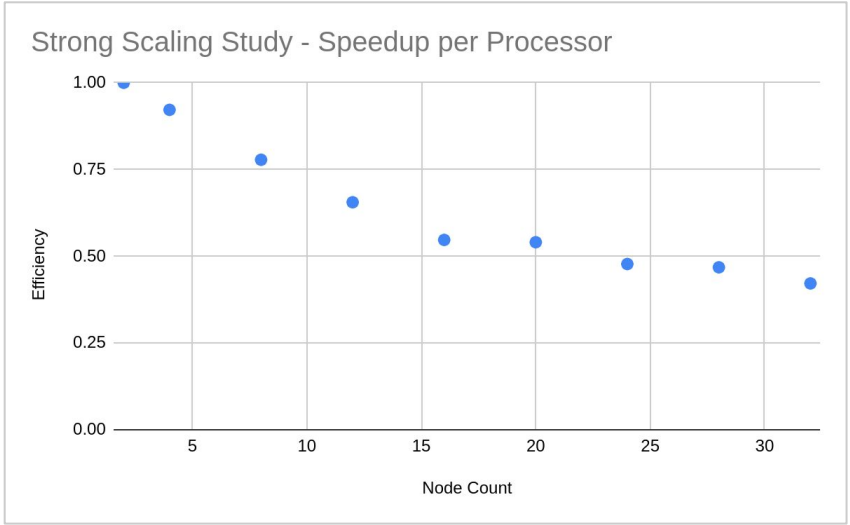
Speedup

Node Count	Speedup
2	1
4	1.843964292
8	3.112420601
12	3.933592247
16	4.37634799
20	5.403731579
24	5.730948408
28	6.553151672
32	6.746196028



Efficiency (aka Speedup Per Processor)

Node Count	Efficiency
2	1
4	0.9219821462
8	0.7781051501
12	0.6555987079
16	0.5470434988
20	0.5403731579
24	0.477579034
28	0.4680822623
32	0.4216372517



Strong Scaling Study - Takeaways

- The local sorting of the processor's respective data, after the program's communication of the elements that belongs to the processor's bucket, is the largest factor in runtime. Specifically, 41% to 85% of the runtime, depending on the number of nodes.
- Speedup and Efficiency quickly deviate from the ideal, showing that the implementation is not scalable with a larger number of nodes

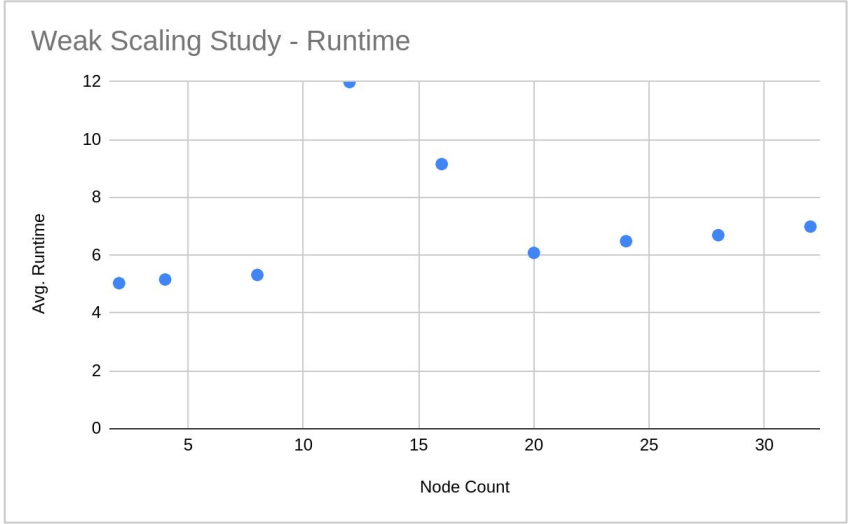
Weak Scaling Study

All runs done with:

- Problem Size of Node Count * 48 million elements
- Range of elements: [-1000,1000]
- Oversampling Factor of 4

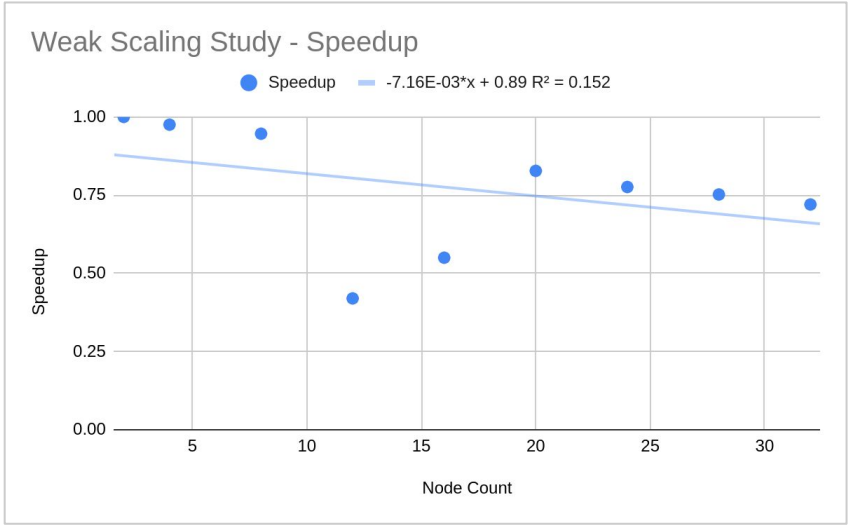
Runtime

Node Count	Avg. Runtime	Stddev
2	5.033522489	3.246492609
4	5.161099351	3.194947562
8	5.319048151	2.413134833
12	11.98935052	3.08237621
16	9.153063332	1.859961334
20	6.082119798	2.588565381
24	6.486668047	1.936077917
28	6.693445329	2.089441154
32	6.990220293	1.733020819



Speedup

Node Count	Speedup
2	1
4	0.9752810684
8	0.9463201584
12	0.4198327908
16	0.5499276369
20	0.827593447
24	0.7759796636
28	0.7520077092
32	0.7200806667



Weak Scaling Study – Takeaways

- The local sorting of the processor's respective data, after the program's communication of the elements that belongs to the processor's bucket, is the largest factor in runtime. Specifically, 52% to 87% of the runtime, depending on the number of nodes.
- The outliers when using 12 and 16 nodes were due to a jump in `MPI_Alltoallv` time when communicating the respective elements that each processor was assigned.

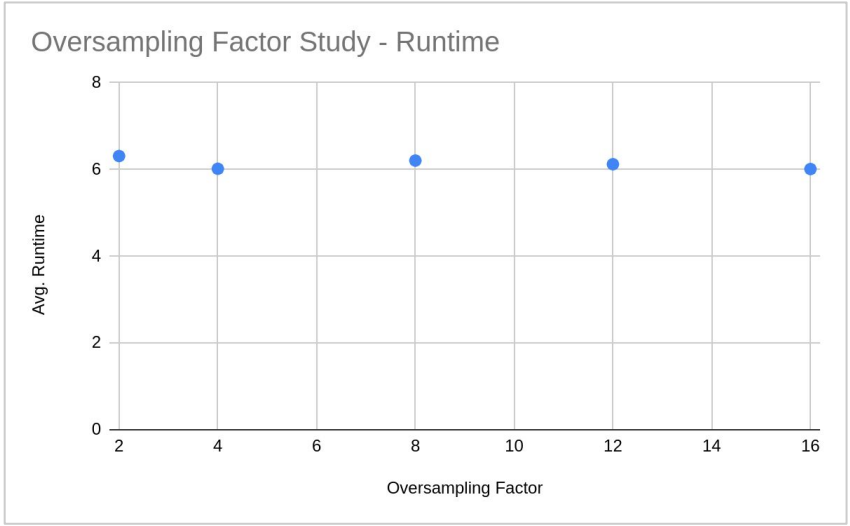
Oversampling Factor Study

All runs done with:

- Problem Size of Node Count * 48 million elements
- Range of elements: [-1000,1000]
- Node Count of 20

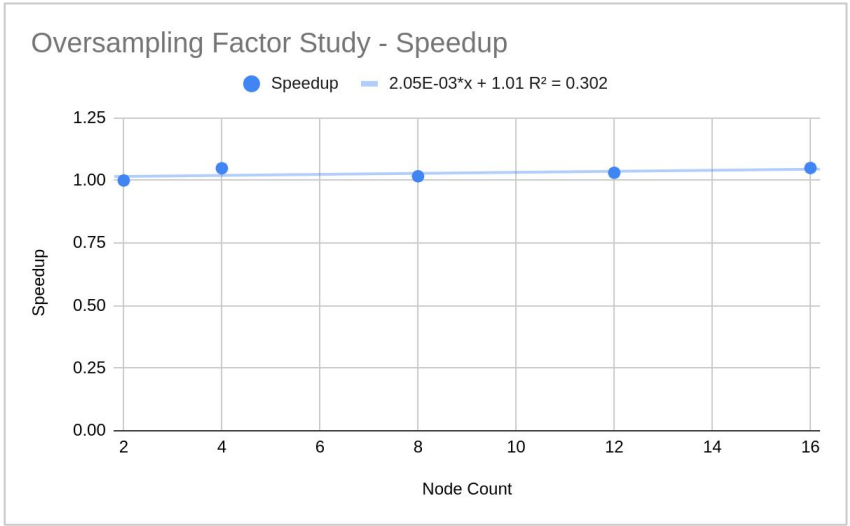
Runtime

Oversampling Factor	Avg. Runtime	Stddev
2	6.310588357	2.819590791
4	6.016740685	2.124496965
8	6.206278933	1.430231095
12	6.121792859	1.463750261
16	6.010414586	1.459056463



Speedup

Node Count	Speedup
2	1
4	1.048838347
8	1.01680708
12	1.030839903
16	1.049942274



Oversampling Scaling Study - Takeaways

- As with weak scaling, the final local sorting was the largest contributor to the total runtime, at about an average of 3.75 seconds across the runs.
- The data shows that varying the oversampling factor does not influence the pivot determination time, much less the total runtime of the program

Conclusions

- Sample Sort as a method of sorting across a cluster is powerful.
- However, it is heavily reliant the efficiency and robustness of the sorting mechanism that the processor uses at the local level.
- Using the C++ STL's vector data structure was a pitfall; an implementation that used OpenMP would have greatly reduced runtimes and addressed the local sorting problem with a custom sorting routine present.

Any questions? Ask away!