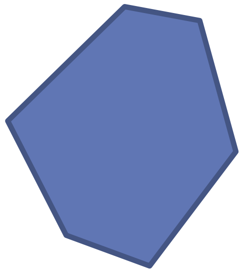
Parallel Convex Hull using MPI

CSE633 Fall 2012

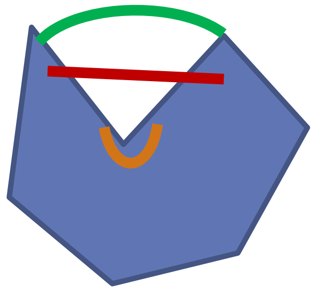
Alex Vertlieb

What is a Convex Polygon?

Convex



Not Convex



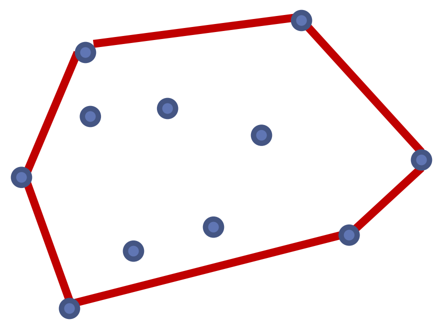
🟥 If you can draw a straight line from one point in the polygon to another and have the line exit the polygon, it is not convex.

🟩 If an external angel < 180 degrees, it is not convex.

🟧 If an internal angel > 180 degrees, it is not convex.

What is a Convex Hull?

The convex hull of a set of points S is the smallest convex polygon that contains all of the points in S.



Extreme Points: points on the polygon – can be used to describe the convex hull

Imagine the points are nails in a board. To get the convex hull, you would just need to take a string and wrap it around the outside of the set of points!

Importance of Convex Hull

- Used heavily in image processing, operations research, molecular biology, layout and design, and many others.

- Many points of data => lots of computation => lets parallelize the solution!

The Problem

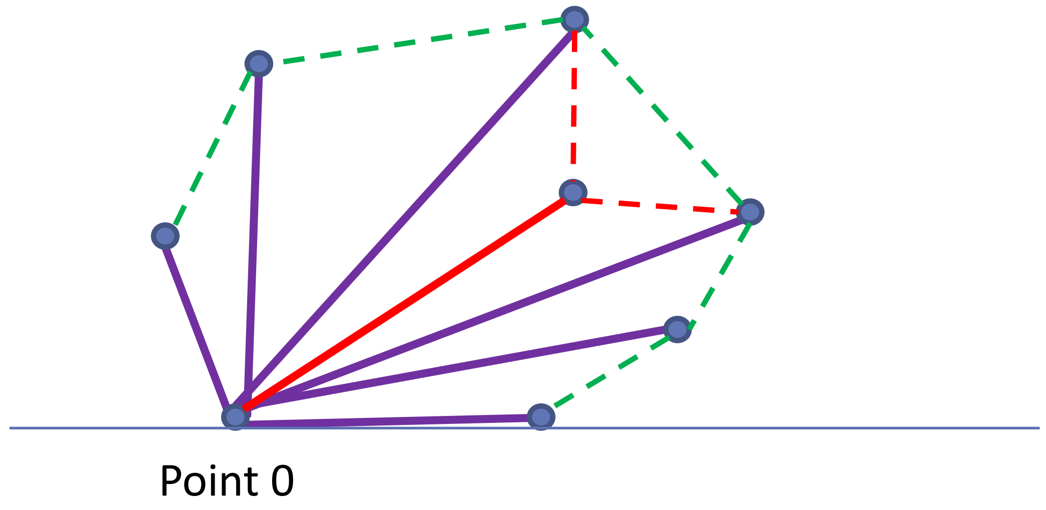
- Input: An set of N points ordered by x-coordinate.
  - (X,Y) coordinates – randomly generated a few sets of data; X & Y I used were double values
- Output: An ordered set of Extreme Points that represent the convex hull.
  - List of (X,Y) coordinates – if you draw straight lines between the points in order, you will have the convex hull

Graham's Scan

- 1. Select the lowest point in S as point 0, and as the origin (break ties with the leftmost point).
- 2. Sort the n-1 remaining points of S by angle in $[0,\pi)$ with respect to the origin point. If two points have the same angle, take the furthest from the origin point.
- 3. For points 1 to n-1, consider the current point
    - Add the point to the set of extreme points
    - If a right turn was introduced into the ordered set of extreme points, remove preceding items from the set of extreme points until there are no more right turns

- $\theta(n \log n)$ – dominated by sort

Graham’s Scan



Algorithm on a Mesh of size n

- Input: n points distributed 1 point per processor

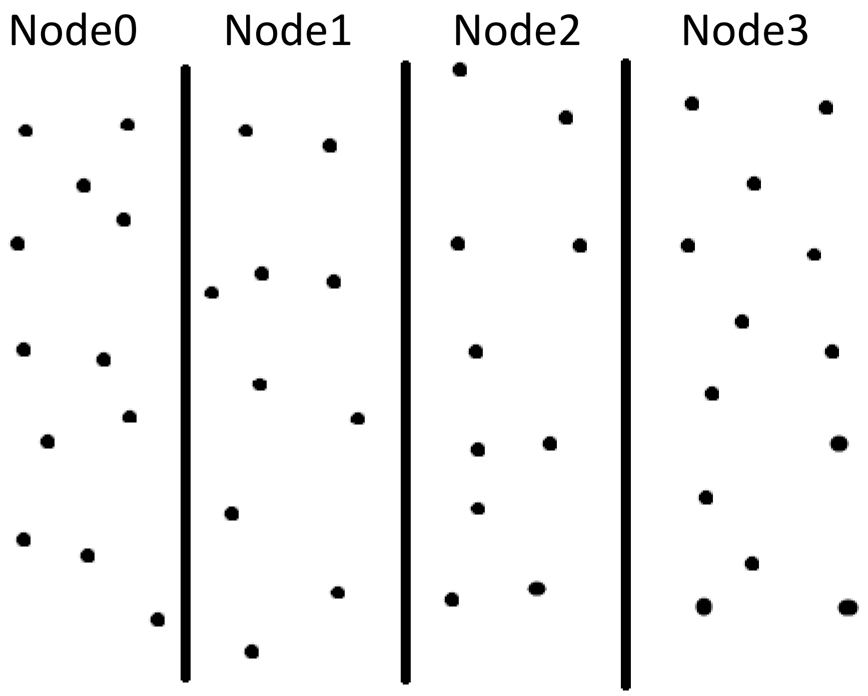
- 1. Sort the points by x-coordinate in shuffled row major ordering across the mesh
  - since every processor has 1 element, we cannot divide further and must begin stitching the 'hulls' together
- 2. Recursively stitch together the hulls
  - Perform a binary search on adjacent hulls to find the upper and lower common tangent lines
  - Eliminate the log(n) factor from the binary search by using a dual binary search with compression

Alteration Plan

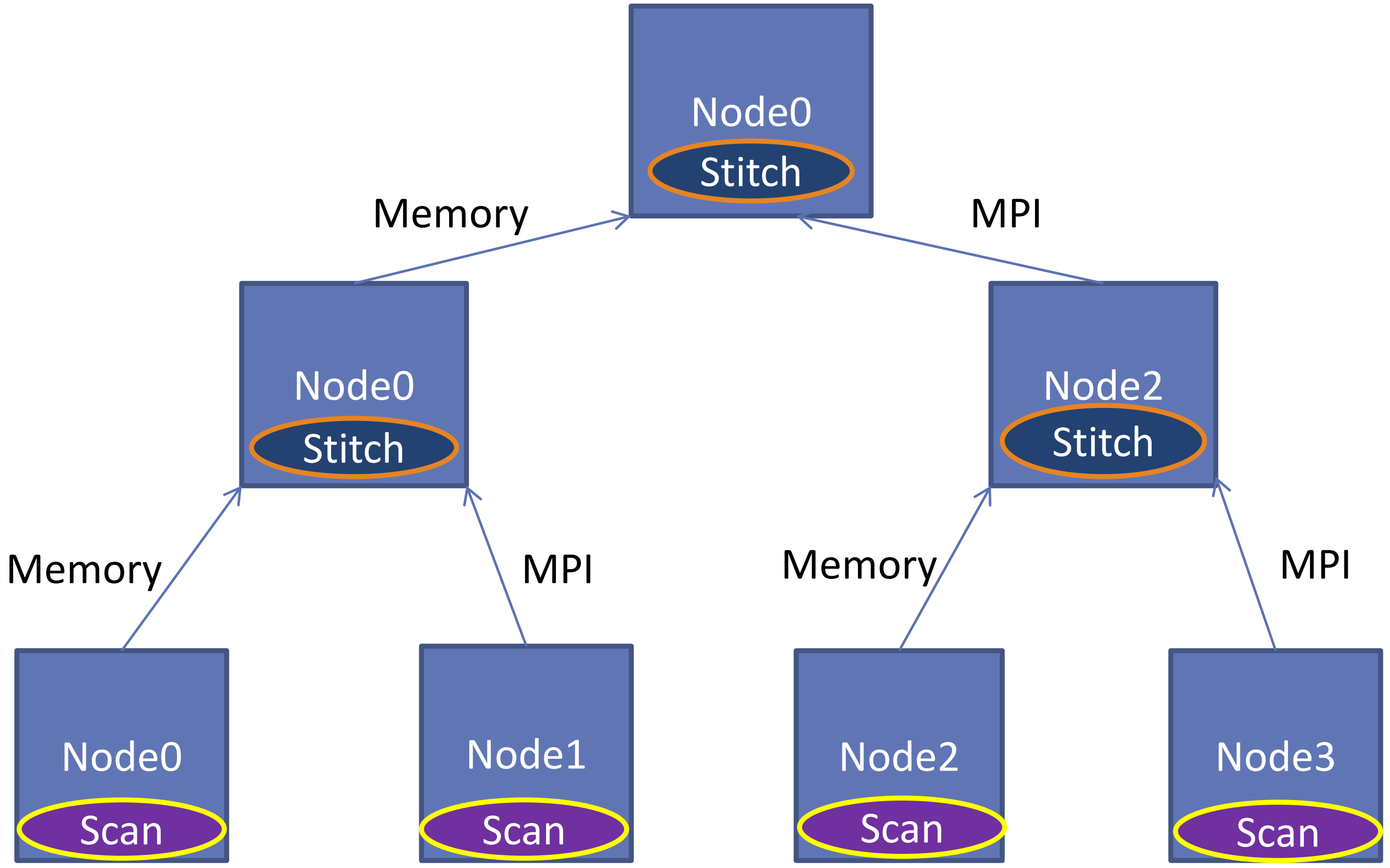
- Coarse Grain (More than 1 point per processor – less than n processors)
- Pre-sorted input to avoid unneeded complication
- Only include Dual Binary Search with Compression if time allows (P.S. time did not allow)
- Change from mesh to hypercube since mesh is more complicated and gives little benefit using the coarse grained cluster

My Algorithm (high level)

- Input: n/p points per processor in order of x coordinates
- 1. Sequentially find your local convex hull
- 2. Recursively Stitch together hulls

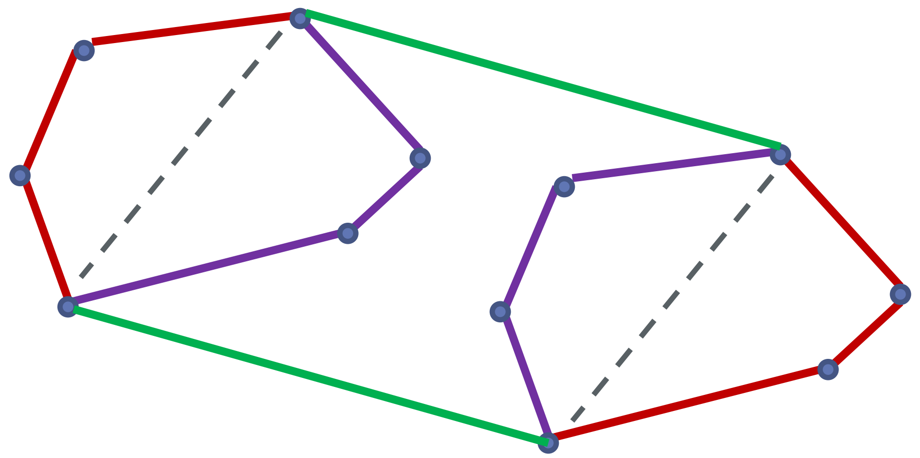


Visualization



Common Tangent Lines

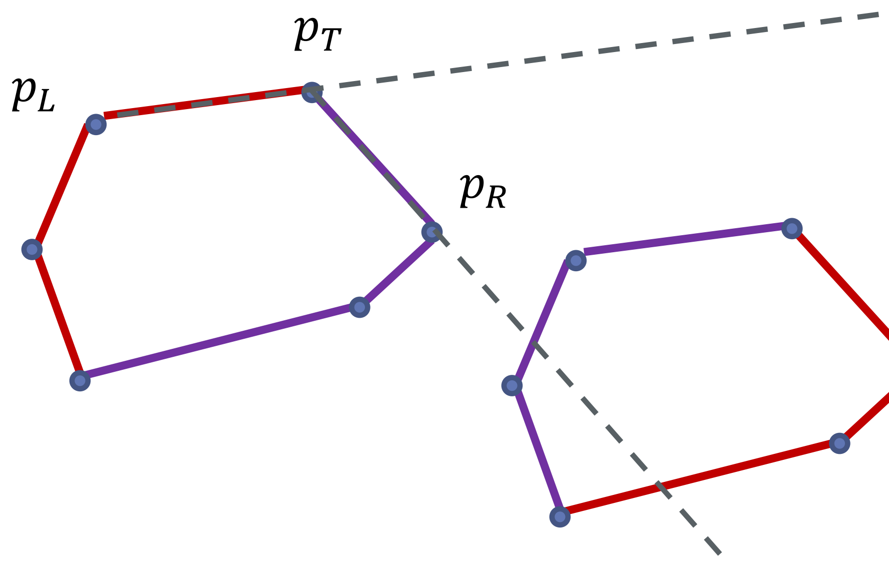
When stitching two hulls together, you must find the upper and lower common tangent lines, then eliminate any points in the quadrilateral formed by the two lines



Common Tangent Lines

You can find the points of the common tangent lines by finding a point where:

- The line between p_L and p_T touches or goes above each points in the opposing hull
- And the line between p_T and p_R touches or goes below a point in the opposing hull



Dual Binary Search with Compression

Preform a search for common tangent lines between the two convex hulls by doing simultaneous binary searches, and reducing the remaining possibilities on both sides at each step and continue searching through those.

- You are doing a binary search on one hull, but also halving the point set that needs to be checked by the other at every step
- Converges to a $\theta(n)$ time operation (proof using a geometric series)

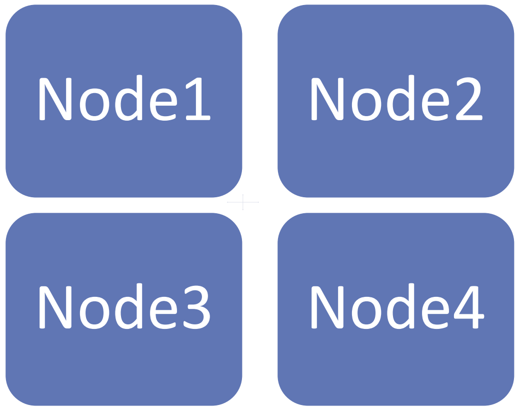
My Algorithm (low level)

- Input: n points sorted by x-coordinate distributed roughly n/p points per processor
- 1. Run Graham's Scan on your coordinates $\theta(n \log(n))$
- 2. For each bit in the rank (← recursive halving) $\theta(\log(p))$
  - Send your local hull to your neighbor $\theta(h = hull\ size)$
    - OR Recv your neighbor's hull from your neighbor
  - If you Recv'd a hull from your neighbor $\theta(h \log(h))$
    - Perform a binary search on both hulls to find the upper and lower common tangent lines
    - Eliminate the points in the quadrilateral formed by the common tangent lines' start and end points

Note: We can eliminate the log(h) factor from the binary search by using a dual binary search with compression

Old Plan & Runtime Analysis

I want to stop when each node to be left with approximately $\frac{\sqrt{n}}{\log(n)}$. Each node can then run the RAM algorithm ($\theta(n \log n)$) on this data to receive a local convex hull in $O(\sqrt{n})$ time. The whole solutions thus stays $\theta(\sqrt{n})$. If I do not implement the dual binary search with compression, there would be an extra log n factor in the solution and thus I could stop when each node is left with approximately $\sqrt{n}$ and not worry about eliminating the log n factor from the RAM algorithm.



Reality – Runtime Analysis

- Rough running time is $\theta(\frac{n}{p}\log\left(\frac{n}{p}\right) + \log(p)h\log(h)$)
- p = # of processors, h = max(size of local hulls), n = # of points
- Running time is $\theta(n)$ or $\theta(n$ log $n)$ for the worst case (when the convex hull itself is $\theta(n)$)
  - Doing a binary search for tangent lines between two $\theta(n)$ size hulls. Could be eliminated by doing a dual binary search w/ compression.
- Average case MUCH better.

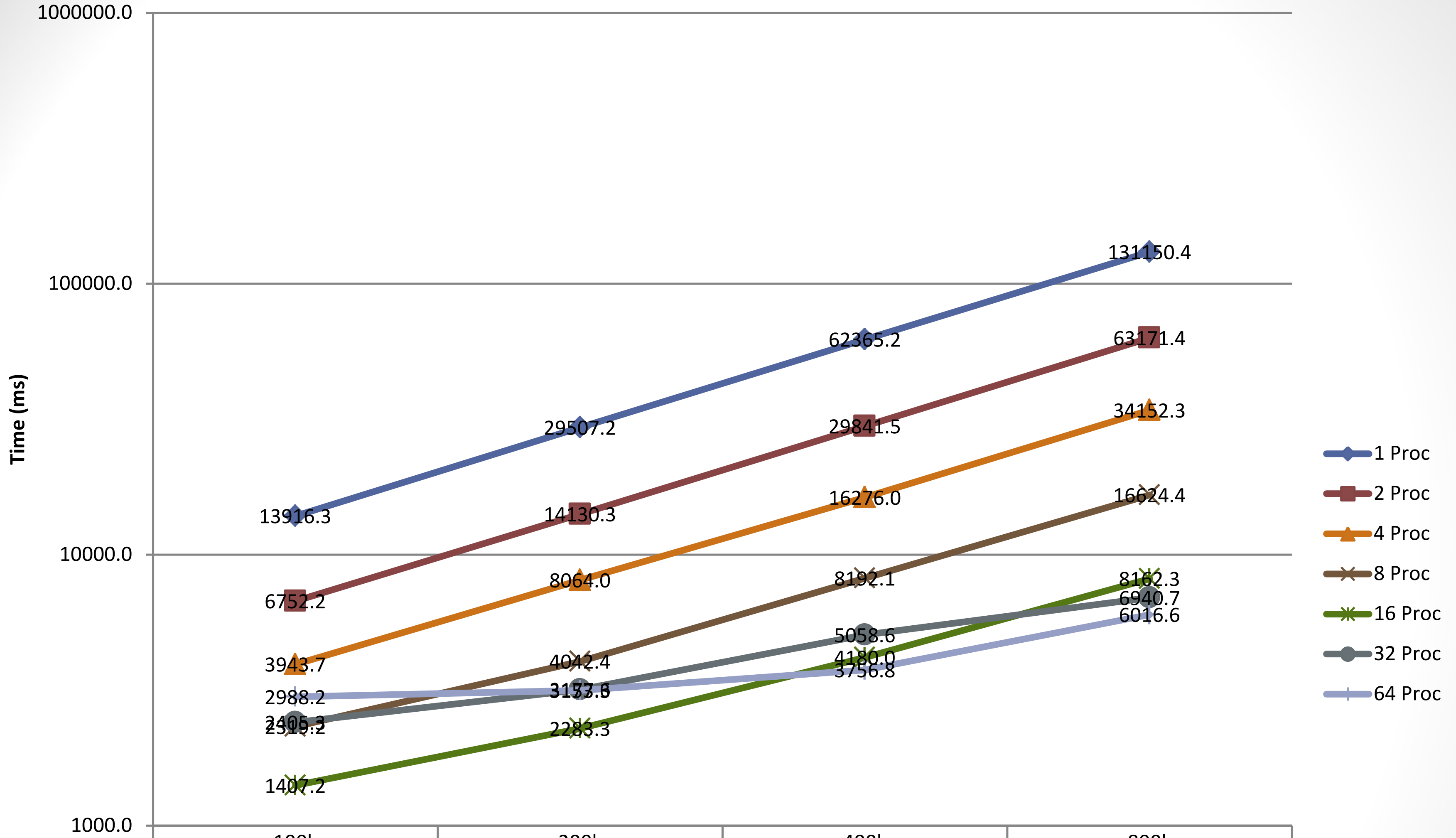
Testing Details

- 2 Core Machines – only using 1 core on each to enforce network traffic after 1 node
- Stored input data in parallel storage (/panasas/scratch)
- Ran on 1, 2, 4, 8, 16, 32, 64 processors

Testing Data

- Data generator for creating a given number of random points within a given rectangle or circle.
  - Can create multiple sets with the same bounds in one run
- Generated 10 sets each of 100k, 200k, 400k, and 800k random points bounded by a circle with a radius of 1 million
- Chopped the data up into 64 files so all processors could access the data independently
- Used the same sets of random data for all the different numbers of processes
- Hulls ended up being between 100-200 points

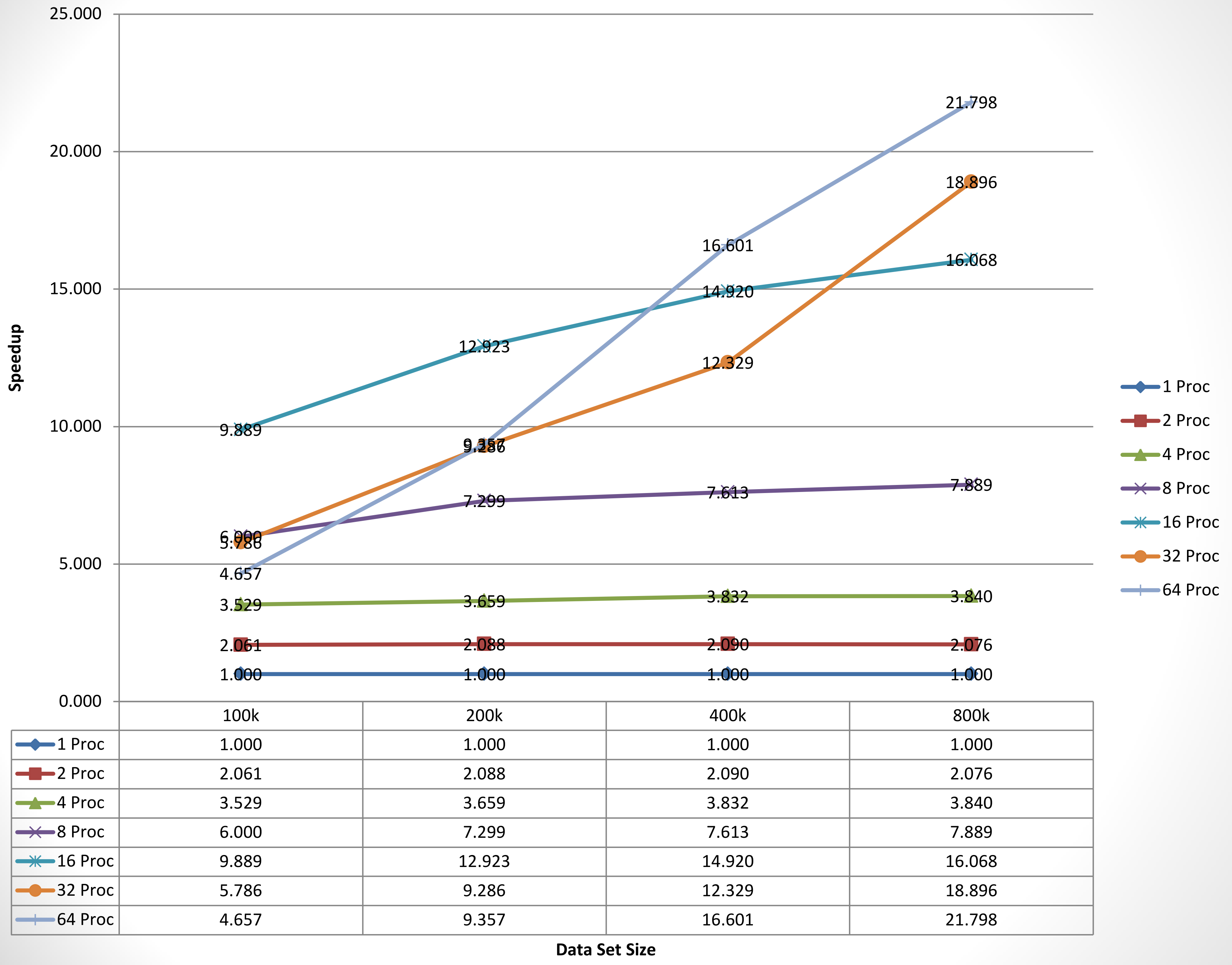
Running Time



	100k	200k	400k	800k
1 Proc	13916.3	29507.2	62365.2	131150.4
2 Proc	6752.2	14130.3	29841.5	63171.4
4 Proc	3943.7	8064.0	16276.0	34152.3
8 Proc	2319.2	4042.4	8192.1	16624.4
16 Proc	1407.2	2283.3	4180.0	8162.3
32 Proc	2405.3	3177.6	5058.6	6940.7
64 Proc	2988.2	3153.3	3756.8	6016.6

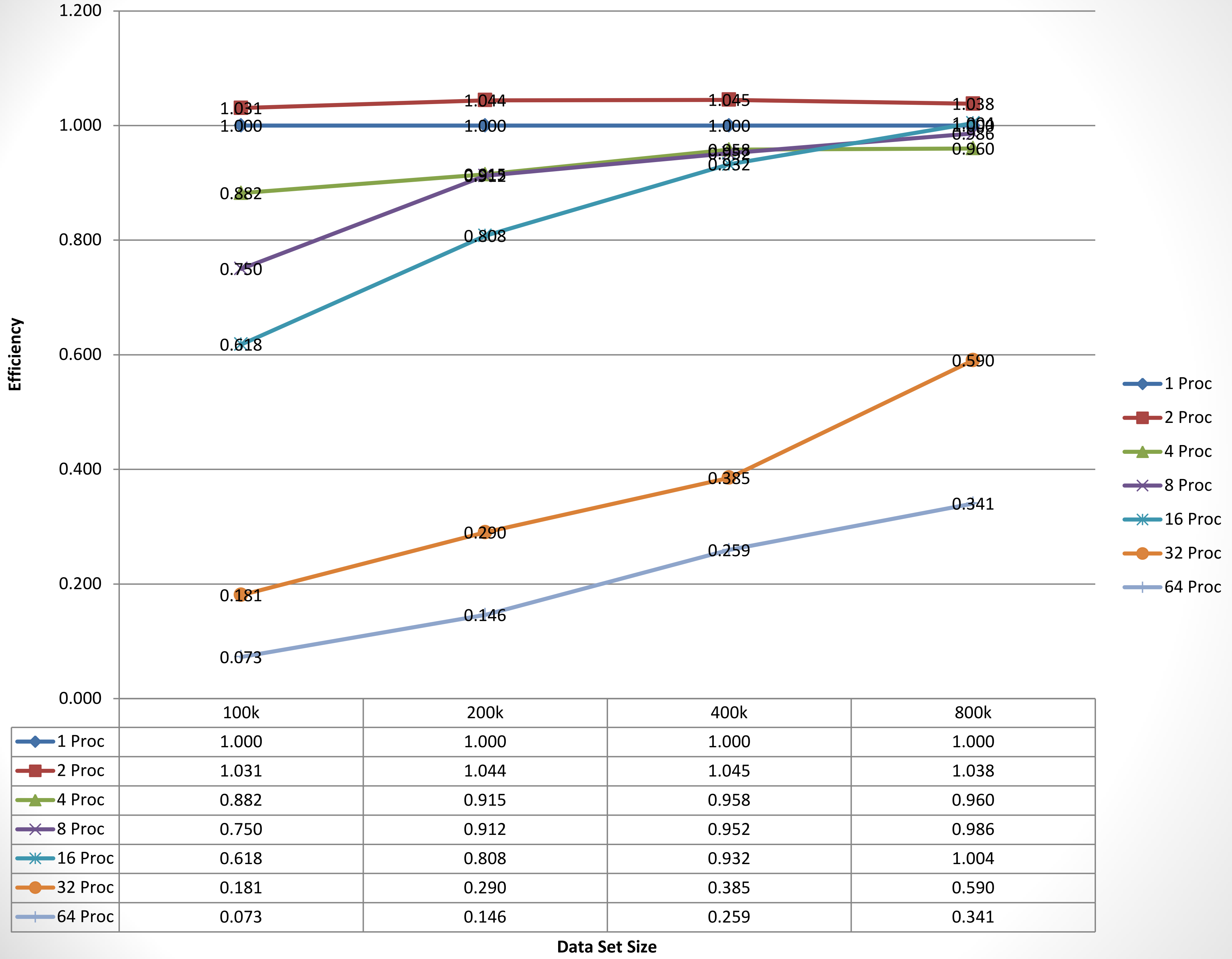
Data Set Size

Speedup



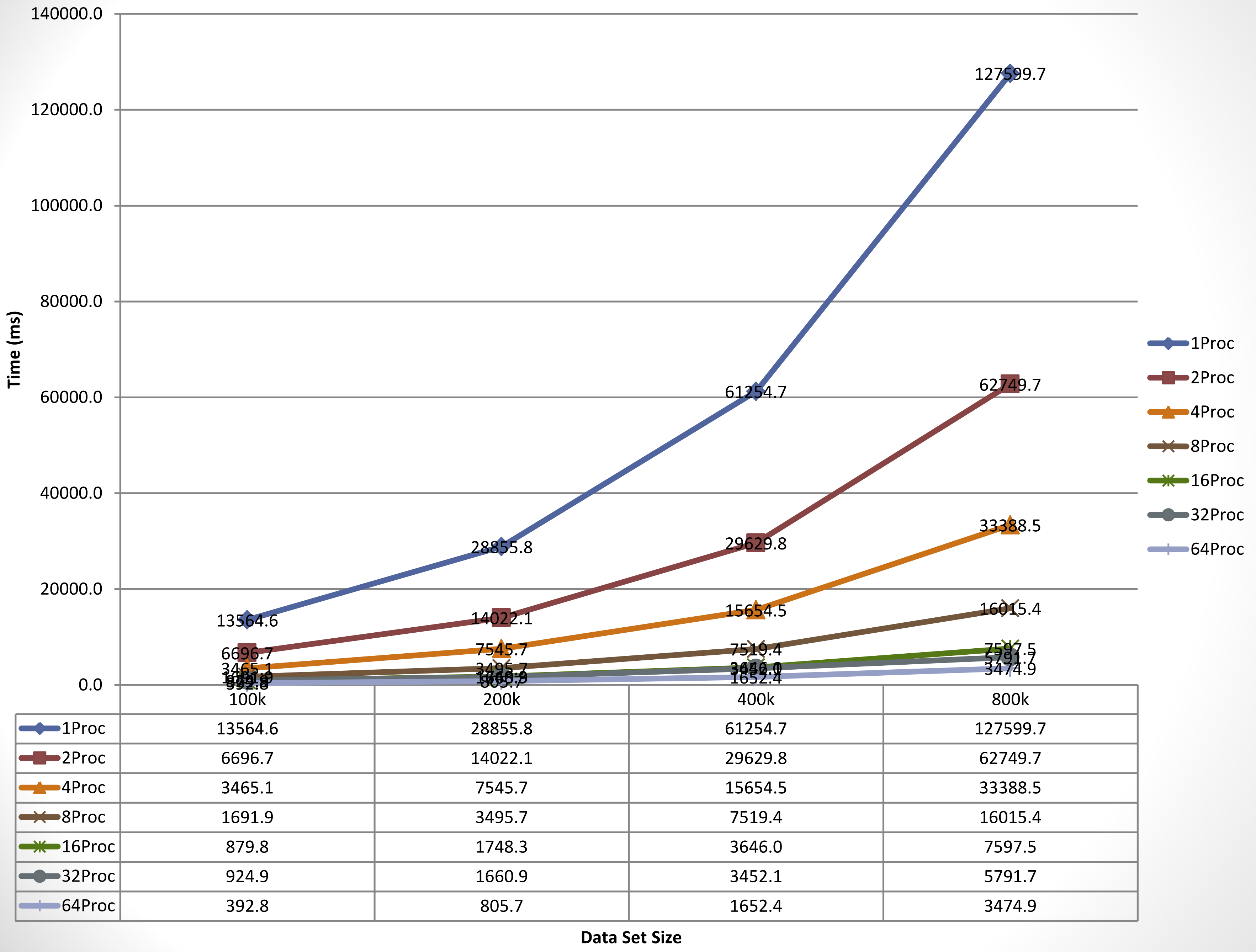
	100k	200k	400k	800k
1 Proc	1.000	1.000	1.000	1.000
2 Proc	2.061	2.088	2.090	2.076
4 Proc	3.529	3.659	3.832	3.840
8 Proc	6.000	7.299	7.613	7.889
16 Proc	9.889	12.923	14.920	16.068
32 Proc	5.786	9.286	12.329	18.896
64 Proc	4.657	9.357	16.601	21.798

Efficiency



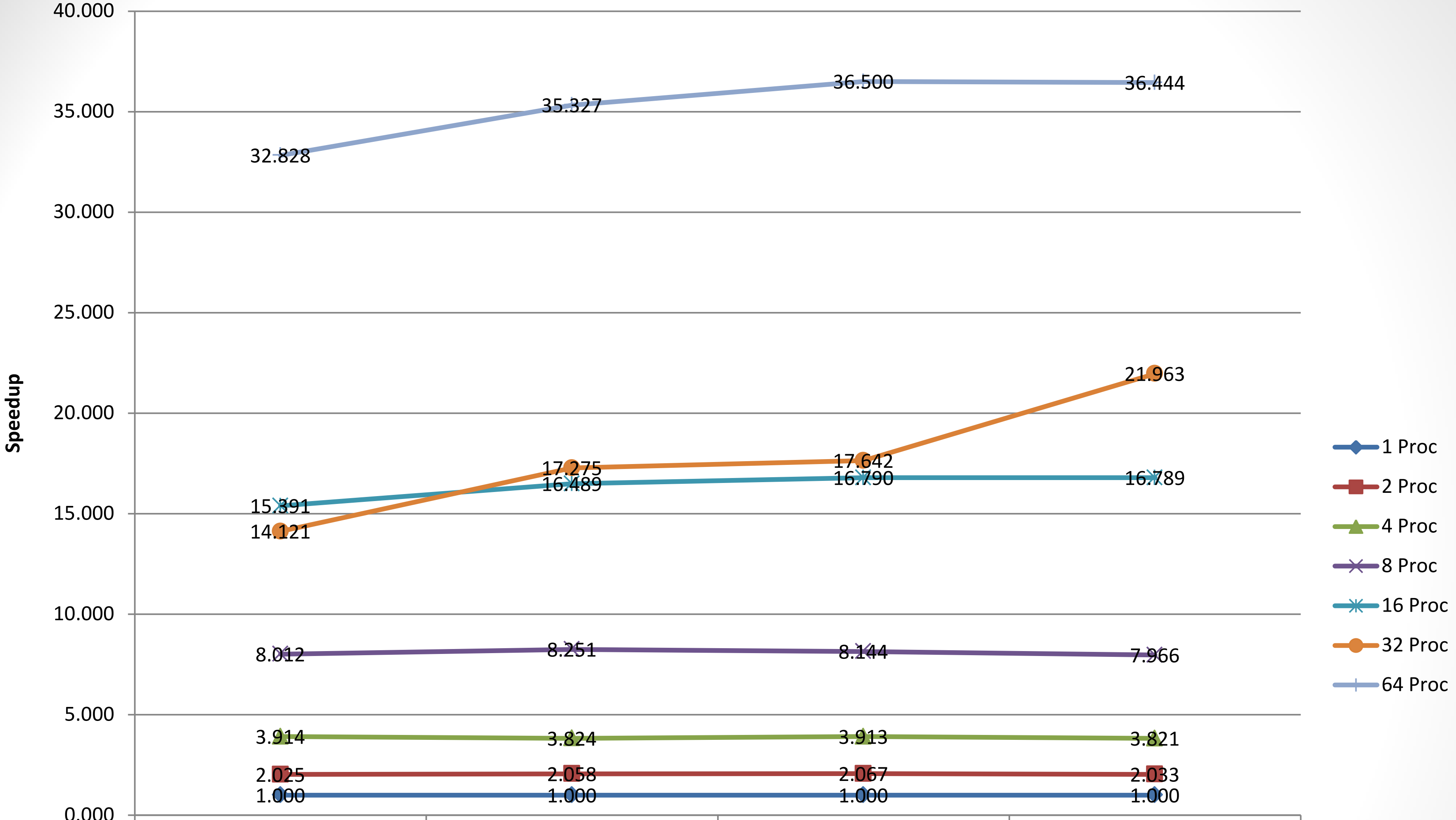
	100k	200k	400k	800k
1 Proc	1.000	1.000	1.000	1.000
2 Proc	1.031	1.044	1.045	1.038
4 Proc	0.882	0.915	0.958	0.960
8 Proc	0.750	0.912	0.952	0.986
16 Proc	0.618	0.808	0.932	1.004
32 Proc	0.181	0.290	0.385	0.590
64 Proc	0.073	0.146	0.259	0.341

Sequential Runtime



	100k	200k	400k	800k
1Proc	13564.6	28855.8	61254.7	127599.7
2Proc	6696.7	14022.1	29629.8	62749.7
4Proc	3465.1	7545.7	15654.5	33388.5
8Proc	1691.9	3495.7	7519.4	16015.4
16Proc	879.8	1748.3	3646.0	7597.5
32Proc	924.9	1660.9	3452.1	5791.7
64Proc	392.8	805.7	1652.4	3474.9

Speedup Excluding Reading Input Files



	100k	200k	400k	800k
1 Proc	1.000	1.000	1.000	1.000
2 Proc	2.025	2.058	2.067	2.033
4 Proc	3.914	3.824	3.913	3.821
8 Proc	8.012	8.251	8.144	7.966
16 Proc	15.391	16.489	16.790	16.789
32 Proc	14.121	17.275	17.642	21.963
64 Proc	32.828	35.327	36.500	36.444

Data Set Size

Super-Linear Speedup

- Could be due to $\frac{n}{p}\log\left(\frac{n}{p}\right)$ being the dominant factor in the runtime on my data sets since, $\frac{n}{2}\log\left(\frac{n}{2}\right) < \frac{1}{2} * n\log(n)$ for large enough n
- Assume log base 2 for the following:
- Ex. $\frac{8}{2}\log\left(\frac{8}{2}\right)$ = $4\log(4)$ = 4 * 2 = 8
    - $\frac{1}{2} * \frac{8}{1}\log\left(\frac{8}{1}\right)$ = $4\log(8)$ = 4 * 4 = 16
- So the running time for 2 processors can be less than half of the running time on 1 processor, which would give the super-linear speedup we see.
- Note: this is only for the data sets that I chose, since the parallel time component of them is negligible, i.e. their stitch steps involve little data

Thoughts

- Reading input took more time when there were more processors reading less data, also varied a lot
  - Because of network and parallel storage resources allocated to my nodes?
  - Because of my jobs fighting with each other to access the files?
  - Because of physical limitations of the hard drive(s) where the files were stored?
- Total parallel stitching runtime never took more than 50 ms
- Times are only recorded on the first processor since the first processor will hold the answer at the end
- Few anomalies could be due to lack of runs on the same datasets, lack of datasets, or how the datasets distributed their workload (since there is a barrier in the code)

Limitations

- Memory needs to be able to hold all points on one node
- Stitch only guaranteed to work on actual convex hulls (3 or more points)
- Integer indexing on the structures that hold the points (input/hull size must be less than MAX_INT)
- Function that says whether a point is above or below a line can’t take too big of values

Moving Forward

- Implement Dual Binary Search
- OpenMP – This algorithm or a PRAM algorithm which applies more to the shared memory model
- Optimize implementation
- Remove the limitations
- Try running the algorithm on a large circle (just the circumference, instead of points contained within a circle)

References

- Miller, Russ, and Laurence Boxer. Algorithms, sequential & parallel: A unified approach. 2nd ed. 2005.