UNIVERSITY OF PADOVA

Course of

"Calcolo Parallelo"

a.a. 2013/2014

A parallel algorithm to compute the Convex Hull of a cloud of planar points

Authors:

Fabrizio Greggio m. 1014065

Stefano Zanella m. 621796

Introduction

Computational geometry deals with problems that take place in n-dimensional space, that is, the space of points represented as n-tuples, $(x1, x2, ..., xn)$ where all x_i are real numbers. These problems concern sets of points, lines, and polygons.

In this project, we dealt with a particular problem of computational geometry: computing the convex hull of a set in 2D.

We are interested in this problem because it is central in practical applications in areas such as pattern recognition, image processing, and stock cutting and allocation. Also, the convex hull problem arises in the context of other computational geometric problems (which may not seem to be related), like the problem of the maximal element of a set or the diameter and width of a planar set of points.

The purpose of this project was to study and implement a parallel algorithm based on the ideas and strategies used in the Chan's, Jarvis March and Graham's scan algorithms.

Convex Hull

Formally, given a set of points P, the convex hull is the smallest convex polygon containing the points:

- polygon: A region of the plane bounded by a cycle of line segments, called edges, joined end-to-end in a cycle. Points where two successive edges meet are called vertices.
- convex: For any two points p and q inside the polygon, the entire line segment pq lies inside the polygon.
- smallest: Any convex proper subset of the convex hull excludes at least one point in P. This implies that every vertex of the convex hull is a point in P.

It is also possible to define the convex hull as the largest convex polygon whose vertices are all points in P, or the unique convex polygon that contains P and whose vertices are all points in P. Notice that P might have interior points that are not vertices of the convex hull.

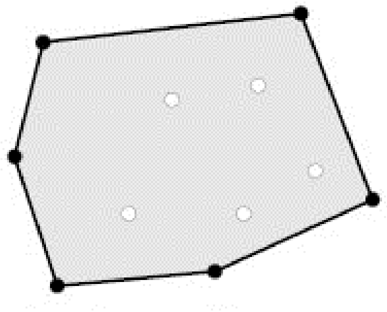


Figure1: *A set of points and its Convex Hull: The Convex hull vertex are colored in black, the interior points are white.*

The Convex Hull has two important properties:

- The first property is that all of the points in the final polygon must be indented outwards, or more formally, convex. If any points are concave, there exists a better path by excluding that point, and drawing a direct line between its two neighbor points, which reduces the overall perimeter point count by one. This property is exploited by Graham's Scan, the pioneering algorithm for convex hulls.
- Another important property is that the most extreme point on any axis is part of the convex hull. This fact is apparent if you consider an example in which an extreme point is not in the convex hull. This would mean the perimeter of the convex hull passes through a point less extreme than that point. However, it is then obvious that the extreme point would not be included in the enclosed set, thus voiding a fundamental characteristic of convex hulls. This property is used by a number of algorithms because they rely on information computed from an initial point in the convex hull

The problem of computing the convex hull of a 2D set of points has been studied extensively, and different algorithms to compute the convex hull have been proposed. They can be divided in two categories, on-line and off-line.

An algorithm that works off-line is one that requires that all inputs are read and stored before any processing can begin. Graham’s Scan is an example of an off-line algorithm to compute the convex hull of a set of n points.

An algorithm that works on-line is one that, without having the entire input from the start, processes the input piece by piece. A general feature of this type of algorithm is that a new item is input on request as soon as the update relative to the previously input item has been completed.

The Graham’s Scan algorithm

The Graham's scan algorithm starts by finding the leftmost point called l. Then it sorts the points in counterclockwise order around l. This can be done in $O(n\ log\ n)$ time with any comparison-based sorting algorithm (quicksort, mergesort, heapsort, etc.). To compare two points p and q, it must be checked whether the triple $l; p; q$ is oriented clockwise or counterclockwise. Once the points are sorted, it is needed to connect them in counterclockwise order, starting and ending at l. The result is a simple polygon with n vertices.

The second step is to change the polygon obtained into the convex hull, so it must be applied the following `three-penny algorithm', commonly used to explain how does it works:

There are three pennies, which will sit on three consecutive vertices $p; q; r$ of the polygon; initially, these are l and the two vertices after l. It is then applied the following two rules over and over until a penny is moved forward onto l:

- If $p; q; r$ are in counterclockwise order, move the back penny forward to the successor of r.

- If $p;q;r$ are in clockwise order, remove q from the polygon, add the edge pr, and move the middle penny backward to the predecessor of p.

Whenever a penny moves forward, it moves onto a vertex that hasn't seen a penny before (except the last time), so the first rule is applied $n-2$ times. Whenever a penny moves backwards, a vertex is removed from the polygon, so the second rule is applied exactly $n-h$ times, where h is the number of convex hull vertices. Since each counterclockwise test takes constant time, the scanning phase takes $O(n)$ time altogether.

The original Chan's algorithm

It is an output-sensitive algorithm, which was discovered by Timothy Chan in 1993, and the running time is $O(n\ log\ h)$. Chan's algorithm involves cleverly combining two slower algorithms, Graham's scan and Jarvis's March, to form an algorithm that is faster than each one.

First, suppose we know there are h points belonging to the convex hull, this algorithm starts by shattering the input points in $\frac{n}{h}$ arbitrary subsets, each of size h, and computing the convex hull of each subset using the Graham's scan technique.

This part of the algorithm requires $O(\frac{n}{h}h\ log\ h)\ =\ O(n\ log\ h)$ time.

Once we have the $\frac{n}{h}$ subhulls, we follow the general outline of Jarvis's march, wrapping a string around the $\frac{n}{h}$ subhulls. Start with the leftmost input point l, starting with $p = l$ we successively find the convex hull vertices in counter-clockwise order until we return back to the original leftmost point again. The successor of p must lie on a right tangent line between p and one of the subhulls, a line from p through a vertex of the sub hull, such that the subhull lies completely on the right side of the line from p's point of view.

We can find the right tangent line between p and any subhull in $O(log\ h)$ time using a variant of the binary search. Since there are $\frac{n}{h}$ subhulls, finding the successor of p takes $O(\frac{n}{h}log\ h)$ time together. Since there are h convex hull edges, and we find each edge in $O(\frac{n}{h}log\ h)$ time, the overall running time of the algorithm is $O(n\ log\ h)$. Unfortunately, this algorithm only takes $O(n\ log\ h)$ time if we know the value of h in advance.

So how do we know the value of h?

Chan's trick is to guess the correct value of h, let's denote the guess by h'. Then we shatter the points into $\frac{n}{h'}$ subsets of size h', compute their subhulls, and then find the first h' edges of the global hull. If $h\ <\ h'$, this algorithm computes the complete convex hull in $O(n\ log\ h')$ time. Otherwise, the hull doesn't wrap all the way back around to l, so we know our guess h' is too small.

Chan's algorithm starts with the optimistic guess $h' = 3$. If we finish an iteration of the algorithm and find that h' is too small, we square h' and try again. In the final iteration, $h' < 2h$, so the last iteration takes $O(n\ log\ h') = O(n\ log\ 2h) = O(n\ log\ h)$ time.

The total running time of Chan's algorithm is given by the sum
$O(n\ log\ 3 + n\ log\ 3^2 + n\ log\ 3^3 + ... + n\ log\ 3^k)$, for some integer k.

So Chan's algorithm runs in $O(n\ log\ h)$ time overall, even if the value of h isn't know a priori.

Parallel Chan's algorithm

As already stated, the problem of efficiently finding the convex hull for a given set has been extensively explored. As a result, many different algorithms exist, and some of them offer the same optimal asymptotic performance (as, for example, the Kirkpatrick-Siedel and Chan's algorithms). But despite that, not all of them seem particularly suited to be adapted for parallel execution. From this point of view, our research pointed out Chan's algorithm as the one most easily adaptable for being run in parallel.

The reason for this is made evident once we look at the structure of the algorithm; by doing so we'll also describe the idea behind the implementation analyzed in this report.

Looking at the original incarnation of Chan's algorithm [1], we see that once we fix the value of h', the algorithm can be splitted in two distinct parts:

- calculating the convex hull for every subset
- merging the convex sub-hulls into the final hull

That said, it is pretty clear how the algorithm could be parallelized:

- since calculating the convex hull for a given subset is a task that has no dependencies with the same calculation for any other subset (i.e. order of execution doesn't matter, no data interchange required), we can just partition the original set of points into P (where P is the number of processors) subsets of roughly the same size and perform the sub-hull calculation in parallel

This approach has a couple of desirable properties:

- in the first part we can take maximal advantage of the whole processor set: as this step requires no synchronization between the processes and the original set of points can be easily scattered into almost equally sized subsets that span all the available processors. This means we can complete this first step, given a set of n points, in $O(\frac{n}{P} log \frac{n}{P})$ time

- the communication between processors in the second parat this point each processor holds the convex hull for a given independent subset of points. From here we can just reduce the points, basically implementing a Jarvis March that takes advantage of the fact that some calculations can be made in parallel:

    - every processor can calculate independently and in parallel the next possible hull point among the ones held in its sub-hull, selecting the one that maximizes the angle made with the latest two points in the global hull
    - a coordinating routine can collect all the P selected points, then calculating again the one that maximizes the angle between the latest two points in the global hull
- the last step can then be iterated until we complete the "wrapping" process, that is, until we find again the first point in the global hull
- this kept at its minimum possible: data is exchanged only in the latest part of the iteration, and is reduced to just a single point per process. In total, given a hull with h points and P processors, the amount of data exchanged is in the order of $O(hP)$.

For what concerns the running time, we already stated that for the first part we incur in a global $O(\frac{n}{P}log\frac{n}{P})$ computational effort. Complexity for the second part, instead, can be dissected in the following way:

- calculating the tangential point for a single sub hull takes $O(log\frac{n}{p})$ time, as it can be performed with a binary search; this can be done in parallel, so this is also the total time required for this task
- the time required to select the right point among the subset of P is implementation dependent (see the next paragraph for a detailed explanation), but in general it doesn't take more than $O(P)$ time
- this last step is performed exactly $h+1$ times (we need to calculate again the first point to be sure we're done); that is, the algorithms performs $O(h)$ iterations

Putting all together, we obtain:

$$T(n) = O(\frac{n}{p}log\frac{n}{p}) + O(P) \cdot O(h) = O(\frac{n}{p}log\frac{n}{p}) + O(hP)$$

We see then that the number of processors contributes positively in the first term (more processor, less time), but it is an overhead factor in the second term (more processors, more time). This observation will be relevant when we will analyse the benchmarks; for the sake of analysing the asymptotic performance, we can safely assume that $h < \frac{n}{P}$ and $P < log\frac{n}{p}$, thus simplifying the above expression to:

$$T(n) = O(\frac{n}{p}log\frac{n}{p})$$

Implementation Details

We'll discuss here some details about the specific implementation choices that drove the development of this project:

- point representation: the basic structure that's used extensively throughout the whole project is the one that represents points. We've implemented this by means of a simple struct with two long long ints representing the point coordinates. We've chosen to use integer numbers so to avoid the precision problems that floating point arithmetic introduces. Note however that integer representation is still suitable for fixed point arithmetic, since given a fixed amount k of decimal places one can just consider a rational number n to be represented as $n \cdot 10^k$
- scattering the data: the first step, after loading the point cloud in memory on the master processor, is to divide the input among the whole processor set. This is done by calling once MPI_Scatterv with

a custom mpi_point datatype as argument, the definition of which can be found in init_mpi_runtime()

- merging the partial hulls: to merge the partial hulls held by the individual processors the algorithm needs in principle to collect the points selected by each CPU and then apply a function that selects the one that maximises the angle generated with the last two points in the final hull. A more efficient way that requires moving less data is to just keep track of the last point in the final hull l, then comparing pairs of points p, q and selecting the one that is either on the right side of the line passing between l and p or l and q, or the one further away in case of collinear points. We actually tried two different implementations that solve this part of the problem:
    - in the first implementation, we manually implemented communication over a simulated inverse-Omega network (that is, broadcasting the last point to all, then sending the selected tangential point to the right neighbor, which in turn selects the point that maximizes the angle, sending it to the right neighbor, and so on). For a single reduction step, the cost of this operation is on the order of $O(log\ P)$
    - a second implementation uses a custom collective operation, called MPI_MAX_ANGLE, that basically selects the point in a pair that maximizes the measurement we already discussed. The subtle point here is that we need to keep track of the last point in the final hull, but the allowed signature for a custom reduction operation only allows two parameters to be passed. To overcome this limitation, we broadcast the selected point at the end of each iteration, and store it in a global variable in every process. This way, we know that the value we need to perform the calculation is always present, and has always the same value among all processors. The advantage of this approach is that it requires less code and might take advantage of specific MPI implementation optimizations. We'll see, however, that in our specific case there is no significant performance gain (the reason of which will be explained in the following paragraph)

Benchmarking

In order to discuss over scalability and other performance properties of the proposed implementation, we started by selecting the values of the two input variables to use as arguments for the test runs, which are the size of the input and the number of processors to use. For the first parameter we selected the values 100k, 200k, 500k, 1M, 2M. For the second one, we've selected the powers of two from 1 up to 64.

For what concerns the measurements done, we've tracked for each combination of the values above the time spent doing each of the following subtasks:

- scattering the initial point cloud to all processors
- calculating the partial hull
- calculating tangential points
- reducing the tangential points to the one that maximizes the angle made with the final hull

In addition to this, we calculated the relative weight of communications with respect to total running time, to see how the two factors relate to each other and to global performance. Running time measurement started after loading the point cloud in the master process and ended after the last point in the final hull was computed. Communication time was calculated by wrapping each call to a MPI function with two calls that behave like a stopwatch: that is, their goal is to calculate the time spent in the communication routine and to increase a global counter that tracks overall communication time.

Finally, we wanted to see how the two proposed versions related to each other in terms of performance, and how much the compiler could help in speeding up execution.

In the first place, we analyzed the speed gain we could obtain by using compiler's optimization flags. Figure 2 shows the absolute running time progression for the three different levels available, for the algorithm version with manual merging ran on 8 CPUs. Clearly there is some consistent advantage in using the -O3 flag.

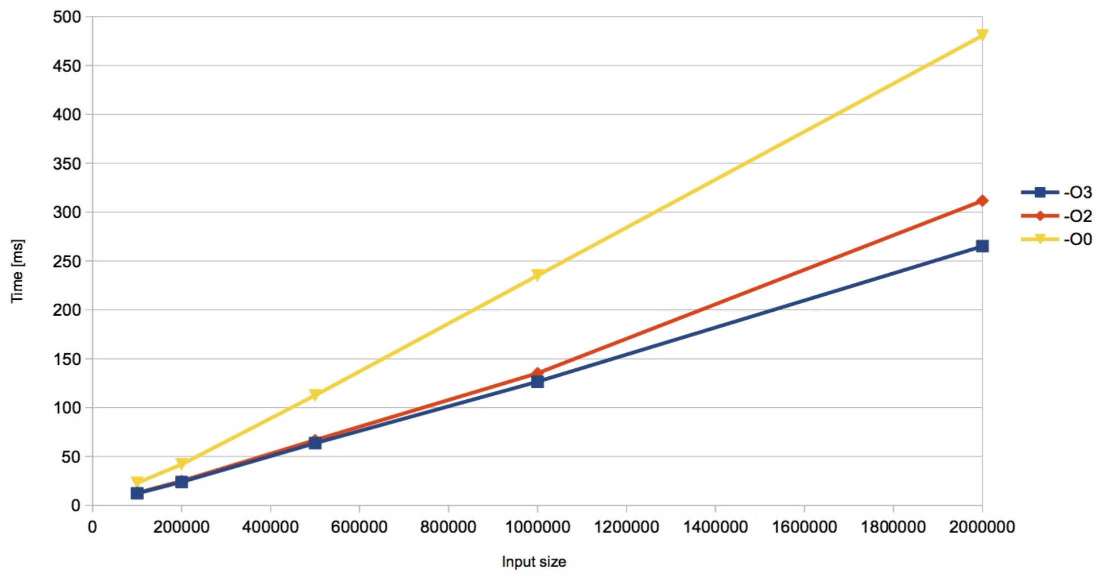


Figure 2: *Running times for the algorithm implemented with manual merge, P = 8*

Figure 3 shows the same graph for the algorithm version that uses the custom MPI reduction function. It can be seen that there's still a good advantage in using the optimization flags, but there is no more a large difference between -O2 and -O3. This is probably due the fact that the data exchange during the reduction phase is carried out directly by MPI, so the compiler cannot act on that part to enhance performance. (For the actual running times, see Table 1 and Table 2)

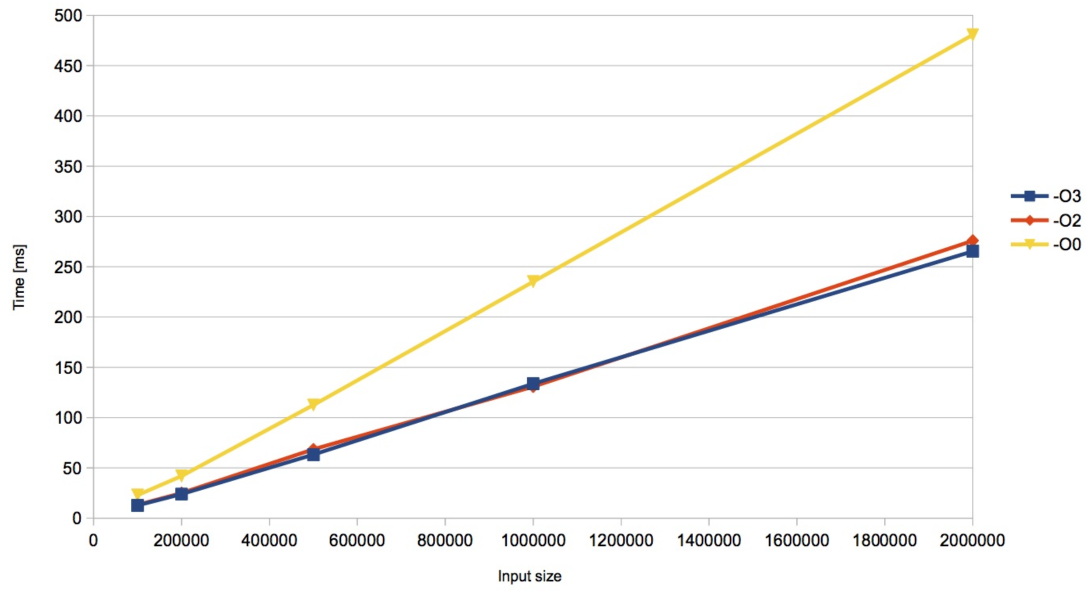


Figure 3: *Running times for the algorithm using a custom Allreduce function, P = 8*

Another interesting comparison is between the running times for the two versions on the same CPU set (figure 4). We can see that there is basically no difference between the two. This is due to the fact that the implementation of MPI_Allreduce follows the same binary-halving approach that we used in our first implementation (see [2]), and the heaviest part in terms of running time is contributed by data exchange, which is platform dependent and cannot be optimized by the compiler.

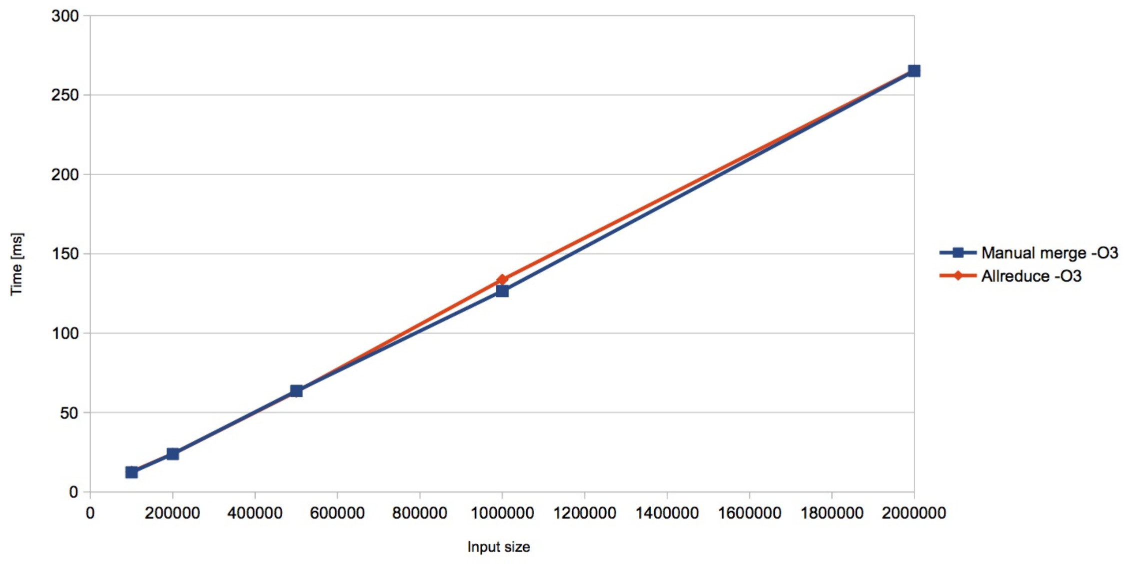


Figure 4: *Running times comparison between the two different implementations, P = 8*

The next step is to analyze the behavior of the two implementations when changing the number of processors, that is, their scalability. Since we already stated that we can obtain smaller running times when using the -O3 flag, we focus only on this particular case. The attached spreadsheet contains graphs also for the other, less-optimized, versions.

Figure 5 shows the relative speedup, i.e. the ratio $S = \frac{T(n,1)}{T(n,P)}$, when P varies among the selected set sizes for the algorithm using manual merging. Different input sizes correspond to distinct data series. The X axis is in logarithmic scale to better show the behavior when the CPU set doubles in size. A first important observation is that up to 4 processors, the speedup increases relatively constantly no matter the input size. When we reach $P = 8$ we start to see how parallelization helps when the input size increase. That is, we start to see that the speedup becomes more relevant as the size of the input point set increases. This phenomena is clearly evident when $P = 16$. We see in fact that for smaller size of the input the algorithm performs worse than the case with $P = 8$. As soon as we input 500k points, though, we start to see an improvement that increases as the input size does. As the input size reaches 2M, we obtain a speedup that is maximum, even though it's less than proportional to the CPU set size, indicating that the benefit of using more CPUs is starting to decrease. This is due to the tradeoff that exists in the algorithm between the computation in the first part and the data exchange in the second part. This observation stays relevant also when we see what happens when $P = 32$ and $P = 64$. In these cases we can argue that the input size in not big enough to make the speedup relevant, in the sense that the time spent exchanging data between processors increases more than how the time spent computing the partial hulls decreases. We'll come to this point later, when we'll analyze the computation vs communication ratio.

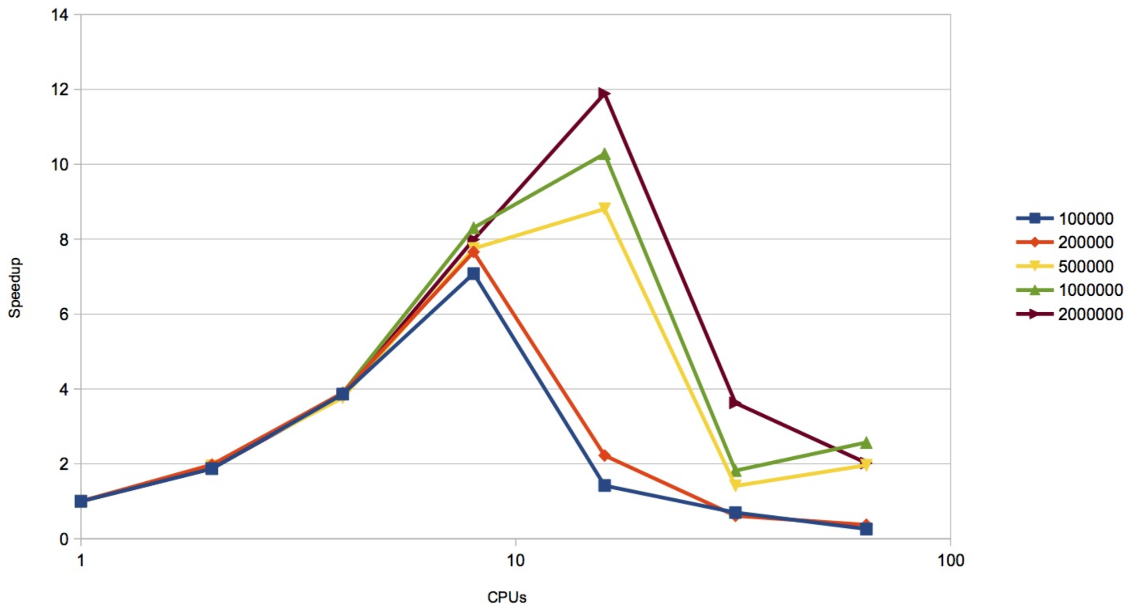


Figure 5: *Speedup obtained for the "manual merge" implementation.*

To conclude this point, we can look at figure 6 to see what kind of speedup we can reach with the implementation that uses a custom reduce function. Here we can see substantially the same behavior, with some unexpected behavior at $P = 8$ and $P = 16$. In the first case, we see little to no improvement for two very different input sizes, i.e. 100k and 1M. In the second case, we see instead that the relative speedup decreases as the input size increases, which goes against the observations we just made. The reason for this is probably due to some “noise” in the execution environment, which is not ideal (in the sense that it doesn’t provide 100% CPU availability and isolation from other tasks), noise that is also probably a bit amplified by the fact that we’re stressing out the compiler’s optimization capabilities (a fact that is known to bring some instability in some cases).

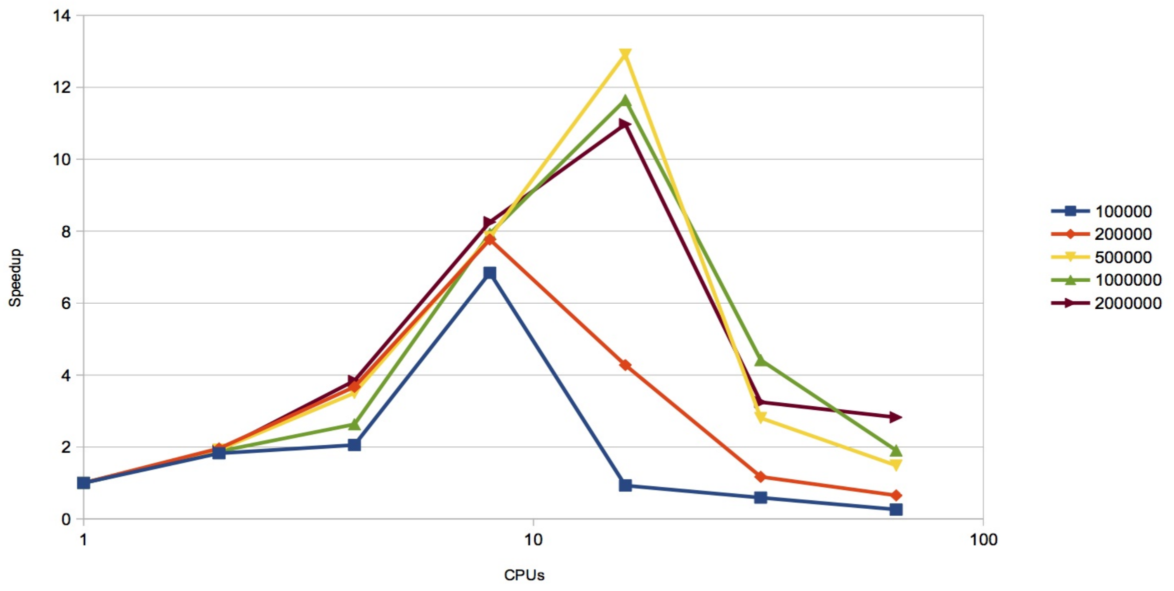


Figure 6: *Speedup obtained for the “custom allreduce” implementation.*

Moving forward, we now analyze in greater depth the communication footprint with respect to the overall computation time. Figures 7 and 8 shows the progression of time percentage spent doing communication for each algorithm execution.

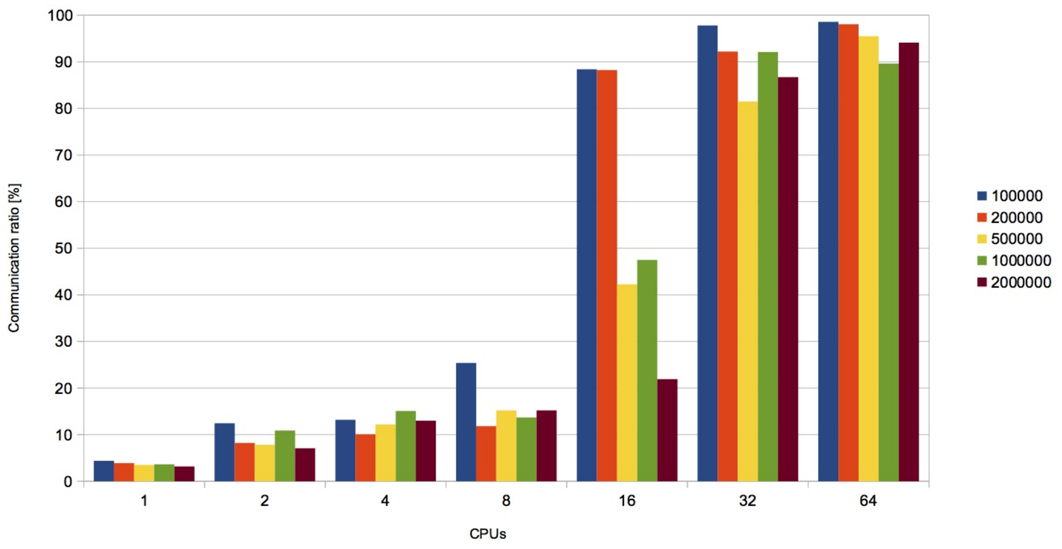


Figure 7: *Communication vs computation time ratio for the “manual merge” implementation.*

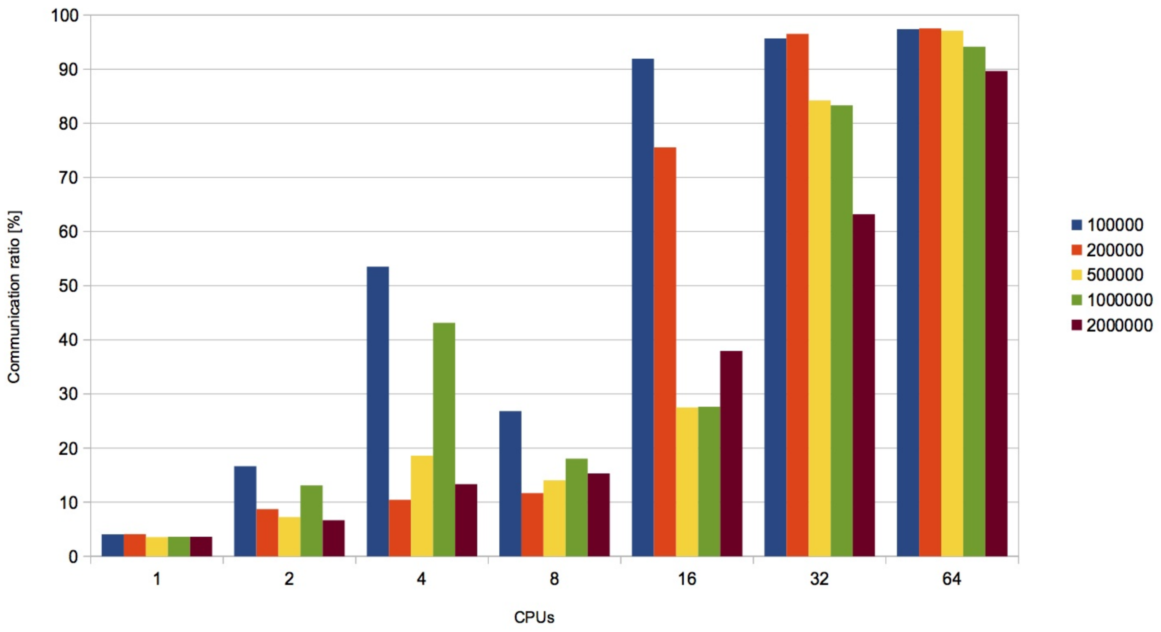
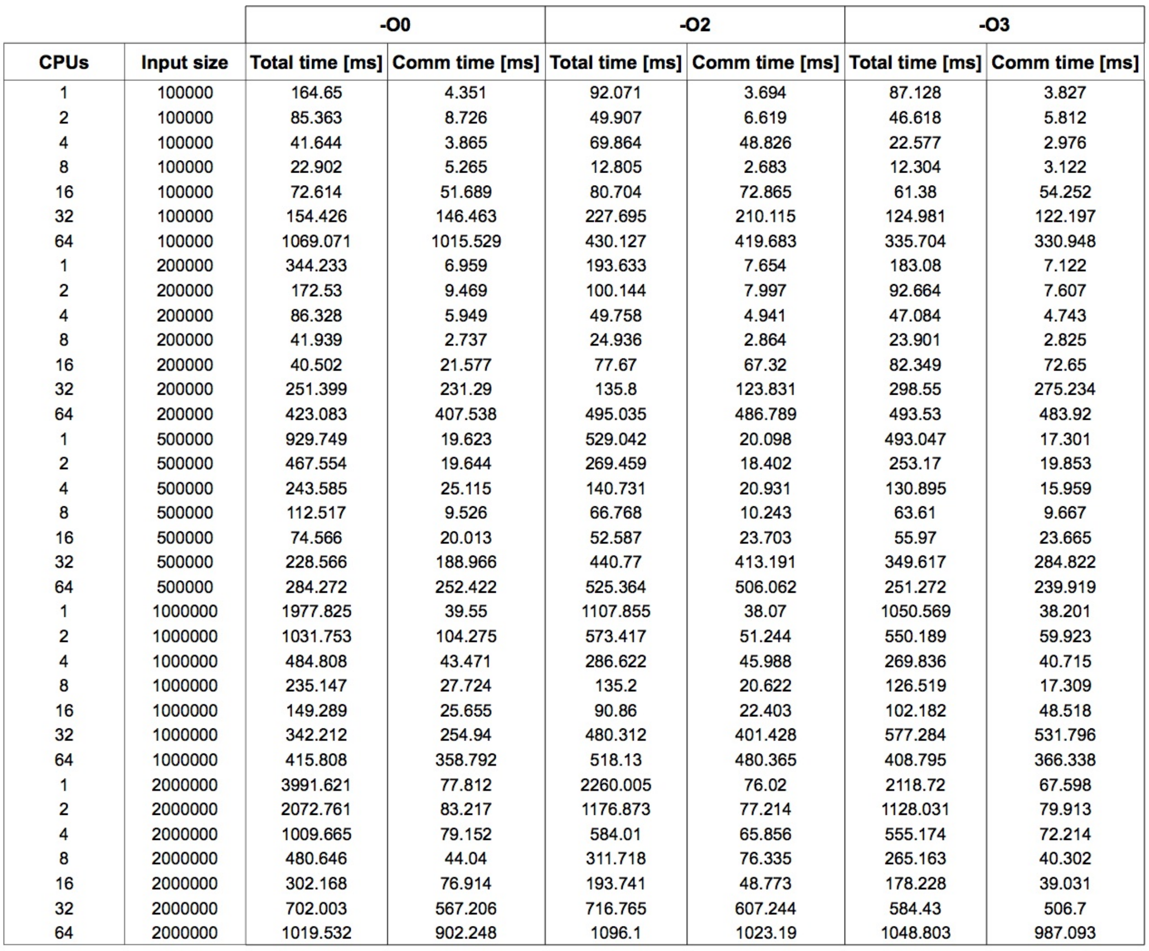


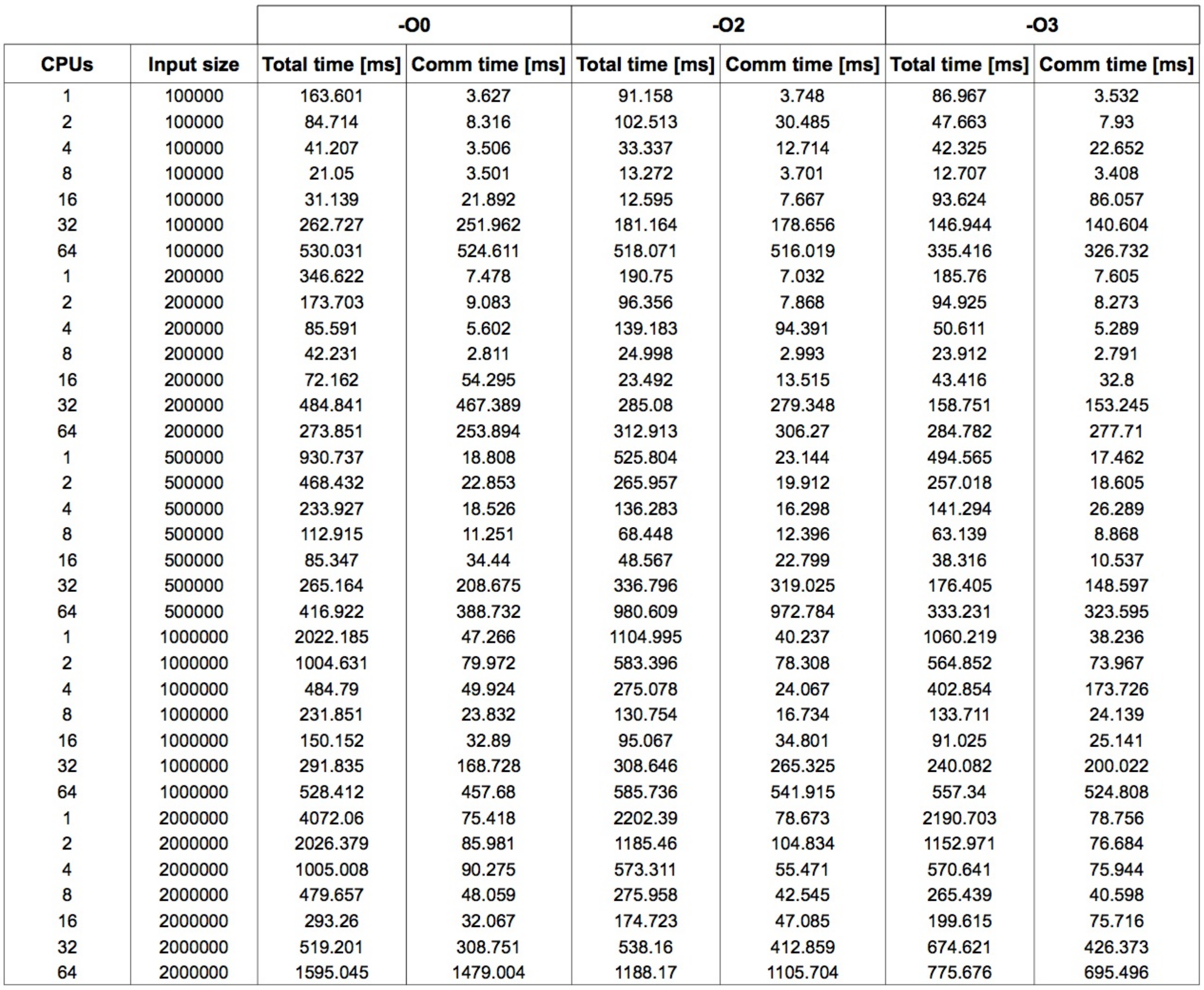
Figure 8: *Communication vs computation time ratio for the "custom allreduce" implementation.*

Here, we notice how communication becomes a bottleneck when the CPU set is larger than 16. In particular, we see how, for the "manual merge" version, the relative amount of time spent exchanging data rises from a rough 20% to almost 90% when we double the CPUs from 16 to 32. Of course, since in the "custom allreduce" implementation part of the computation is done while exchanging data, the ratio increases less abruptly; still, we reach 90% for P = 64. Keeping in mind that point comparison during the merging phase is comprised as "computation" in figure 7 and as "communication" in figure 8, the fact that apart from what already stated the ratios are comparable between the two implementations states the fact that the bottleneck is not due to computation problems, but simply due to the fact that there's not enough bandwidth to keep moving data at an efficient level as the CPU set increases beyond a certain level (with respect to the increase of input size).



		-O0		-O2		-O3	
CPUs	**Input size**	**Total time [ms]**	**Comm time [ms]**	**Total time [ms]**	**Comm time [ms]**	**Total time [ms]**	**Comm time [ms]**
1	100000	164.65	4.351	92.071	3.694	87.128	3.827
2	100000	85.363	8.726	49.907	6.619	46.618	5.812
4	100000	41.644	3.865	69.864	48.826	22.577	2.976
8	100000	22.902	5.265	12.805	2.683	12.304	3.122
16	100000	72.614	51.689	80.704	72.865	61.38	54.252
32	100000	154.426	146.463	227.695	210.115	124.981	122.197
64	100000	1069.071	1015.529	430.127	419.683	335.704	330.948
1	200000	344.233	6.959	193.633	7.654	183.08	7.122
2	200000	172.53	9.469	100.144	7.997	92.664	7.607
4	200000	86.328	5.949	49.758	4.941	47.084	4.743
8	200000	41.939	2.737	24.936	2.864	23.901	2.825
16	200000	40.502	21.577	77.67	67.32	82.349	72.65
32	200000	251.399	231.29	135.8	123.831	298.55	275.234
64	200000	423.083	407.538	495.035	486.789	493.53	483.92
1	500000	929.749	19.623	529.042	20.098	493.047	17.301
2	500000	467.554	19.644	269.459	18.402	253.17	19.853
4	500000	243.585	25.115	140.731	20.931	130.895	15.959
8	500000	112.517	9.526	66.768	10.243	63.61	9.667
16	500000	74.566	20.013	52.587	23.703	55.97	23.665
32	500000	228.566	188.966	440.77	413.191	349.617	284.822
64	500000	284.272	252.422	525.364	506.062	251.272	239.919
1	1000000	1977.825	39.55	1107.855	38.07	1050.569	38.201
2	1000000	1031.753	104.275	573.417	51.244	550.189	59.923
4	1000000	484.808	43.471	286.622	45.988	269.836	40.715
8	1000000	235.147	27.724	135.2	20.622	126.519	17.309
16	1000000	149.289	25.655	90.86	22.403	102.182	48.518
32	1000000	342.212	254.94	480.312	401.428	577.284	531.796
64	1000000	415.808	358.792	518.13	480.365	408.795	366.338
1	2000000	3991.621	77.812	2260.005	76.02	2118.72	67.598
2	2000000	2072.761	83.217	1176.873	77.214	1128.031	79.913
4	2000000	1009.665	79.152	584.01	65.856	555.174	72.214
8	2000000	480.646	44.04	311.718	76.335	265.163	40.302
16	2000000	302.168	76.914	193.741	48.773	178.228	39.031
32	2000000	702.003	567.206	716.765	607.244	584.43	506.7
64	2000000	1019.532	902.248	1096.1	1023.19	1048.803	987.093

Table 1: *Benchmark summary for the "manual merge" implementation.*



		-O0		-O2		-O3	
CPUs	**Input size**	**Total time [ms]**	**Comm time [ms]**	**Total time [ms]**	**Comm time [ms]**	**Total time [ms]**	**Comm time [ms]**
1	100000	163.601	3.627	91.158	3.748	86.967	3.532
2	100000	84.714	8.316	102.513	30.485	47.663	7.93
4	100000	41.207	3.506	33.337	12.714	42.325	22.652
8	100000	21.05	3.501	13.272	3.701	12.707	3.408
16	100000	31.139	21.892	12.595	7.667	93.624	86.057
32	100000	262.727	251.962	181.164	178.656	146.944	140.604
64	100000	530.031	524.611	518.071	516.019	335.416	326.732
1	200000	346.622	7.478	190.75	7.032	185.76	7.605
2	200000	173.703	9.083	96.356	7.868	94.925	8.273
4	200000	85.591	5.602	139.183	94.391	50.611	5.289
8	200000	42.231	2.811	24.998	2.993	23.912	2.791
16	200000	72.162	54.295	23.492	13.515	43.416	32.8
32	200000	484.841	467.389	285.08	279.348	158.751	153.245
64	200000	273.851	253.894	312.913	306.27	284.782	277.71
1	500000	930.737	18.808	525.804	23.144	494.565	17.462
2	500000	468.432	22.853	265.957	19.912	257.018	18.605
4	500000	233.927	18.526	136.283	16.298	141.294	26.289
8	500000	112.915	11.251	68.448	12.396	63.139	8.868
16	500000	85.347	34.44	48.567	22.799	38.316	10.537
32	500000	265.164	208.675	336.796	319.025	176.405	148.597
64	500000	416.922	388.732	980.609	972.784	333.231	323.595
1	1000000	2022.185	47.266	1104.995	40.237	1060.219	38.236
2	1000000	1004.631	79.972	583.396	78.308	564.852	73.967
4	1000000	484.79	49.924	275.078	24.067	402.854	173.726
8	1000000	231.851	23.832	130.754	16.734	133.711	24.139
16	1000000	150.152	32.89	95.067	34.801	91.025	25.141
32	1000000	291.835	168.728	308.646	265.325	240.082	200.022
64	1000000	528.412	457.68	585.736	541.915	557.34	524.808
1	2000000	4072.06	75.418	2202.39	78.673	2190.703	78.756
2	2000000	2026.379	85.981	1185.46	104.834	1152.971	76.684
4	2000000	1005.008	90.275	573.311	55.471	570.641	75.944
8	2000000	479.657	48.059	275.958	42.545	265.439	40.598
16	2000000	293.26	32.067	174.723	47.085	199.615	75.716
32	2000000	519.201	308.751	538.16	412.859	674.621	426.373
64	2000000	1595.045	1479.004	1188.17	1105.704	775.676	695.496

Table 2: *Benchmark summary for the “custom allreduce” implementation.*

Conclusions and further steps

To conclude this analysis, we should spend few words comparing the asymptotical complexity of the algorithm we designed (which, again, can be written as $O(\frac{n}{p} log \frac{n}{p})$) with the one of the original, non-parallel, incarnation of Chan's algorithm, which is $O(n\ log\ h)$. Clearly, the (asymptotical) advantage of the parallel version over the original one depends on the relative weight of each factor. We already stated that $h < \frac{n}{P}$; this means also that $log\ h < log \frac{n}{P}$. On the other side, it's clear that $\frac{n}{P} < n$. The actual implications of the opposite weight of these inequalities depend on the actual value of the parameters. In general, though, we can expect the logarithmic inequality to weigh more than the linear one, thus making the non-parallel implementation asymptotically faster. To improve this result, the parallel implementation could be made smarter so that within a single processor, instead of performing directly a run of Graham's Scan, a non-parallel version of the same Chan's algorithm is performed. In this case the asymptotic performance could be expressed as:

$$O(\frac{n}{Ph} h\ log\ h) = O(\frac{n}{P} log\ h)$$

Which, for the reasons already mentioned, would be faster than the implementation proposed in this paper. Note, however, that this evolution doesn't address the communication bottleneck already analyzed in the previous paragraph; thus, speed improvement won't be so relevant beyond a certain size of the processor set.

Possible further steps can involve the speedup of our algorithm by reducing the number of points that needs to be considered at the initial phase and then next in the localized Graham's scan step. One way to do that is interior elimination: throw away all the points that are known not to be in the convex hull or are for sure in the interior part of the polygon. This knowledge depends on the distribution of the points: if the distribution is random or even in both directions a marvelous interior eliminator would be a rectangle stretched between the points closest to the corners. All the points strictly inside the rectangle can be immediately eliminated.

Bibliography

[1] Timothy M. Chan. "Optimal output-sensitive convex hull algorithms in two and three dimensions". *Discrete and Computational Geometry*, Vol. 16, pp.361–368. 1996.

[2] Improving the Performance of Collective Operations in MPICH (http://www.mcs.anl.gov/~thakur/papers/mpi-coll.pdf)