Parallel implementation of a Convex Hull algorithm

Joy Battocchio, Leonardo Vicentini

github.com/joy-battocchio/parallel_convex_hull

12 December 2022

1 Introduction

The **convex hull** problem consists of finding the smallest convex polygon that encloses a given set of points in the plane, also referred as "cloud". This problem is commonly encountered in computational geometry, and it has a wide range of applications in areas such as image processing, robot navigation, pattern recognition and also towards economics, thermodynamics, quantum physics and beyond.

1.1 Challenge

The challenge we decided to tackle was the parallel implementation of an existing algorithm solving the convex hull problem. In particular, we focused on the 2D version of this issue (figure 1).

1.2 State of the Art

The topic has been extensively studied and algorithms have been proposed since the 1970s. The complexity of the proposed algorithms is usually estimated in terms of n, the number of input points, and sometimes also in terms of h, the number of points on the convex hull. The most famous algorithms that solve this problem are two: the Jarvis algorithm (or Gift wrapping) which has $O(nh)$ time complexity and the Graham scan — $O(n \log n)$.

We chose to use a **"Divide et impera"** algorithm, proposed for the first time by Preparata and Hong in 1977 [1], as the approach is the most suitable for parallelization. It works by recursively dividing the set of points into smaller subsets, and then computing the convex hull of each subset independently. The final convex hull is then obtained by combining the convex hulls of the subsets. The algorithm has a time complexity of $O(n \log n)$. It has been proven to be one of the best algorithms in terms of execution time, but it's seldom used due to its complex implementation and interpretability.

2 Problem analysis

2.1 Formal problem statement

The problem can be formalized as follows:

Given a set of points $S = \{p_1, p_2, \ldots, p_n\} \subseteq R^2$, find the smallest convex polygon $CH(S)$ that encloses all the points in S. $CH(S)$ is the output set of points.

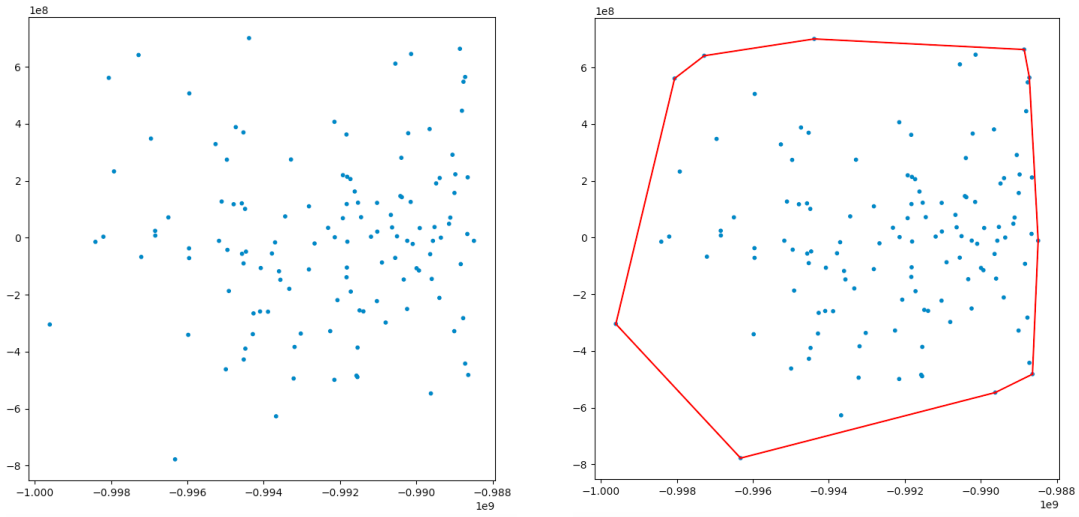
2.2 Assumptions

The input file (`cloud_to_load.txt`) is composed by 2^{22} lines, each one representing a point with x and y integer coordinates separated by a semicolon, like the following:

```
-999609121;-304579271
```

The points in the input file are already sorted by the x coordinate. The width and height of the area in which each point of the input set can lie is 10^9, starting from the origin $(0, 0)$ outwards. The maximum cloud size tested is 4194304 (2^{22}).

With the code provided it is also possible to generate a random input cloud with the function `cloud_generator()`. If needed, process with rank 0 is the one in charge of sorting the input set of



(a) Point cloud

(b) Convex hull

Figure 1: Convex hull problem representation

points by x coordinate, using the function `qsort(cloud, cloud_size, sizeof(point), compareX);` (which is currently disabled).

2.3 Main algorithm implementation

Our workflow started from a C++ implementation of the serial "divide et impera" algorithm [2]. After a first analysis, we did a porting into the C language. The file `convex_hull.c` contains the functions used to compute a convex hull, used by both the serial and parallel implementation. In particular, the most relevant function are `divide()`, `merger()` and `bruteHull()`. These functions are the implementation of the *divide et impera* algorithm used also by every process independently (in the parallel scenario) to compute the local convex hull:

- `divide()` simply divide the point cloud in two subsets recursively until it reaches a size < 6.
- `bruteHull()` represents the base case, in which it is computed the convex hull of a cloud of size < 6 through a brute force algorithm which has $O(n^3)$ time complexity.
- `merger()` first finds the upper and lower tangents of two adjacent hulls, then it finds the other points composing the merged convex hull from the two partial hulls.

In the parallel scenario all the local hulls are then merged to obtain the global convex hull using the `merger()` function.

3 Main steps

3.1 Design of the parallel solution

We implemented a solution leveraging a **hybrid parallelization** with **MPI** and **OpenMP** master-only style. Therefore MPI is used only outside of the OpenMP parallel regions. One advantage of this approach is that we avoid message passing inside of SMP (shared-memory multiprocessing) nodes. The trade-off is that all the other threads are sleeping while a master thread communicates. In particular, we decided to implement the part related to MPI first. Once we obtained a correct solution, we tried to optimize it by exploiting OpenMP.

3.2 Implementation

3.2.1 MPI

The first step was the creation of a derived datatype, `MPI_point`, representing a point. Defining an MPI datatype allowed us to optimize the performance of the MPI communication operations: by using an appropriate datatype, we ensured that the data was transmitted efficiently and with

minimal overhead. Additionally, it will be easier and faster to change the datatype of the point coordinates (like `float`) if needed; currently `long long` is used.

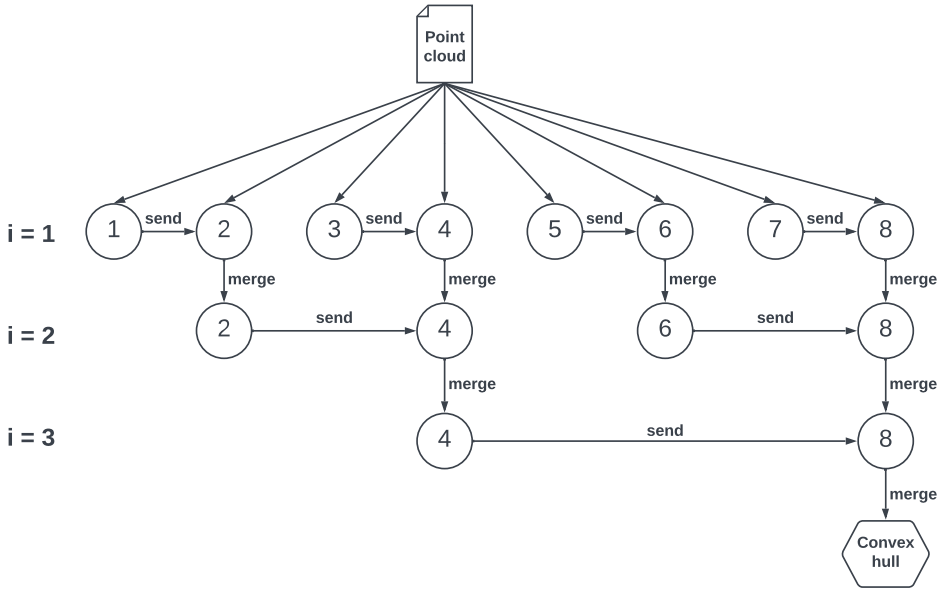


Figure 2: Communication topology with 8 processes

We designed a **Tree-structured** algorithm to compute the convex hull in a parallel way, leveraging MPI communications. The topology of the MPI processes involved, which can be seen in figure 2, can be only made up by a number of MPI processes that is equal to a power of 2.

After every process computes its **local convex hull**, half of the processes send their hull to the other half in order to be merged. The merging step is executed $log_2(n.processes)$ times.

At each step a process with rank *my_rank* has the role of receiver if the following condition holds: `if my_rank % 2^i = 0`.

The senders are all the other processes. After a process sends its convex hull, its execution is over. For instance:

- step 1: senders:{1,3,5,7}, receivers: {2,4,6,8}
- step 2: senders: {2,6}, receivers: {4,8}
- step 3: senders: {4}, receivers: {8}

The first implementation we designed exploited the `MPI_Scatter()` function in order to **split** and assign a subset of points of size N/p among the available processes. The aforementioned strategy could be adopted if there is the necessity or constraint of reading the input file by only 1 process. A brief time analysis pointed out that this call was a bottleneck and so in order to obtain a significant improvement with the respect of the serial implementation a new strategy was adopted.

With the current strategy, each process has the duty to read a specific section of the input file (using the function `cloud_load()`); this lead to a major performance improvement with the respect of the use of `MPI_Scatter()` to split the input set of points.

The last process is responsible for writing the final convex hull in the output file.

3.2.2 OpenMP

As mentioned before, OpenMP was added in a second time, to further optimize the implementation. The number of threads to be spawned is configurable, with the 4th argument when launching the program from command line. During the benchmarking phase, we set the number of threads to **4**, which is the configuration that gave the best results. In the shell script it is the current default.

In our solution there is only one parallel section, in the `divide()` function. Due to the highly recursive nature of that section of the algorithm, no improvements were visible at the beginning of our tests after introducing OpenMP; rather, performances were slightly worse. This behavior is plausibly due to the overhead introduced by OpenMP as well as the fact that recent compilers

optimizations make OpenMP increasingly less effective in certain situations. Taking this into consideration, we introduced an optimization exploiting a **threshold**: multi-threading is activated only when the number of points to manage by each thread is greater than 2000. [3] [4]

3.3 Testing and validation

In order to test the correctness of the implementation we took advantage of 2 web applications: Planetcalc tool for convex hull (using Jarvis algorithm) [5] and Desmos graphical calculator (for input sizes greater than 4096). [6]

3.4 PBS configuration

To submit the job on the cluster, with a shell script we requested a number of chunks (`select` directive) equal to the number of MPI processes, each one with a number of cores (`ncpus` directive) equal to the number of OpenMP threads.

For instance: `#PBS -l select=4:ncpus=4:mem=2gb -l place=pack:excl`

The `place=pack:excl` option is a scheduling directive that indicates that the job should be placed on a single node that have no other jobs running on it. This option can be used to ensure that a job has exclusive access to the resources of a node, which can be useful for certain types of workloads and for benchmarking. [7]

We also tried to tune the `mpiexec` (or `mpirun`) command with the `map-by` and `bind-to` options to specify how to map and bind the MPI processes and threads when running an MPI job. Unfortunately, with only a few tests conducted, we were unable to achieve improvements in terms of timing. in addition, we received warnings of various kinds. Further investigations in this direction could likely improve the performance of the implementation.

3.5 Benchmark on the HPC@UniTrento cluster

The configuration used for benchmarking always involved 4 threads, without the I/O steps for visualization (described in section 3.6).

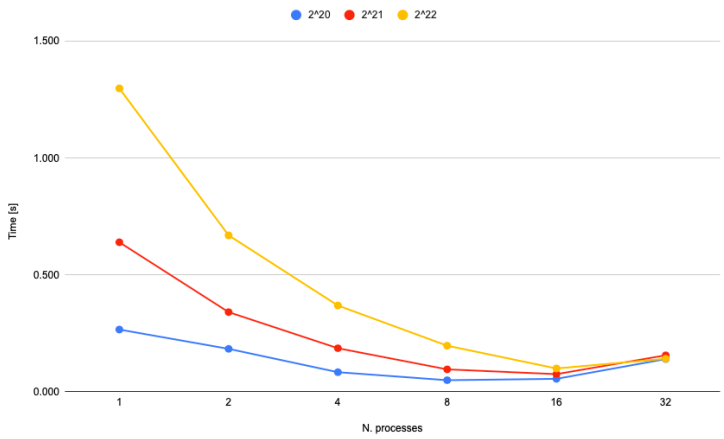


Figure 3: Execution time graph

As we can see from table 7 and graph 3, we observed a decreasing of the execution time until 16 processes, after that (32 processes) time started to increase again. In the case of the smallest input size we observed this behaviour starting even earlier, between 8 and 16 processes.

3.5.1 Speedup and Efficiency

In parallel computing, the **speedup** of an algorithm is defined as the ratio of the execution time of the serial algorithm to the execution time of the parallel algorithm (on multiple processors). This measure is used to evaluate the performance of parallel algorithms and to determine how well they **scale** to larger numbers of processors.

The formal definition of speedup can be written as follows:

$$Speedup = \frac{Tserial}{Tparallel}$$

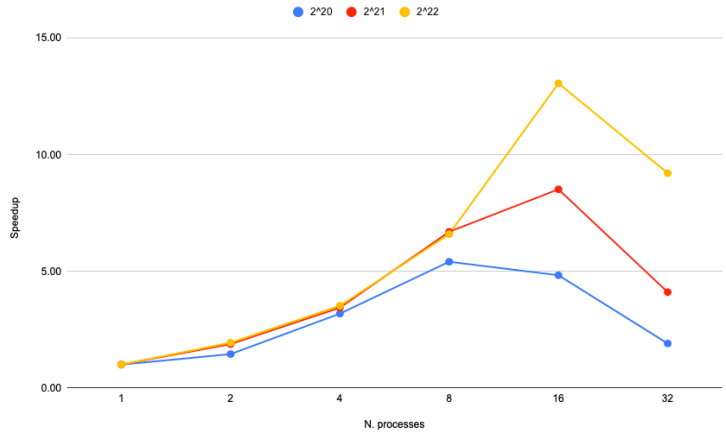


Figure 4: Speedup graph

The **efficiency** of an algorithm is defined as the ratio of the speedup of the algorithm to the number of processors used. It is used to evaluate the performance of parallel algorithms and to identify potential bottlenecks or limitations in their scalability. The formal definition of efficiency can be written as follows:

$$Efficiency = \frac{Speedup}{p} = \frac{Tserial}{p * Tparallel}$$

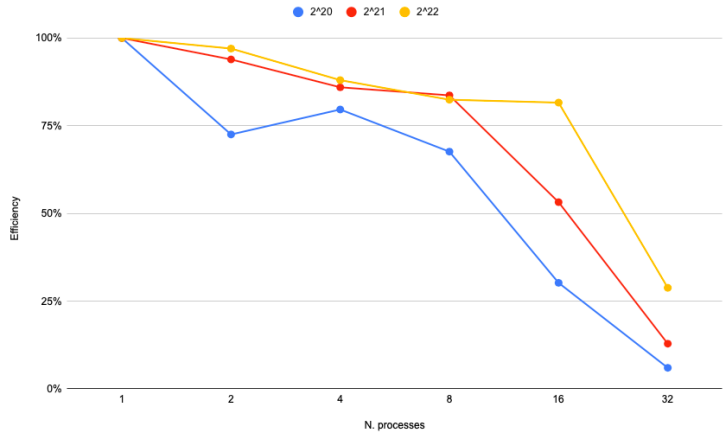
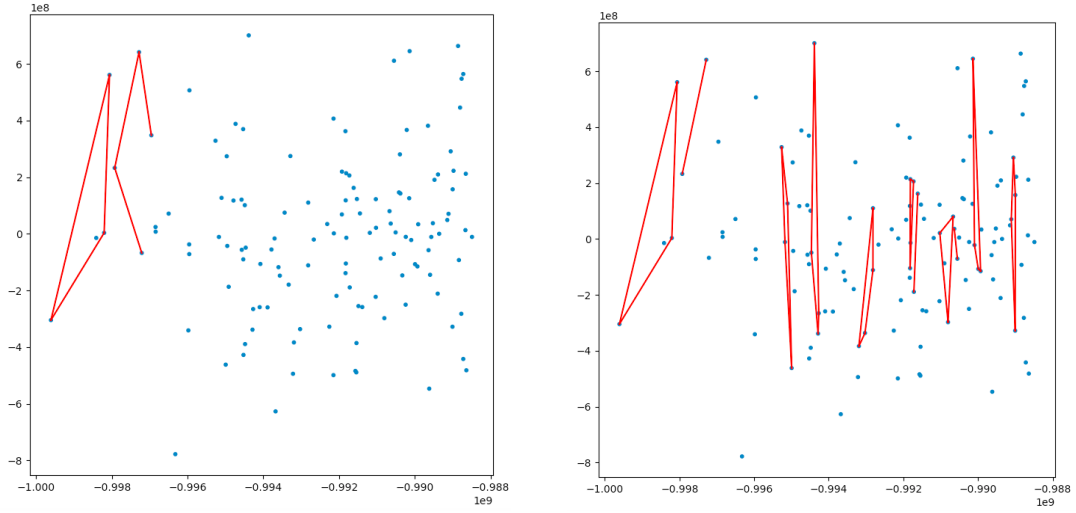


Figure 5: Efficiency graph

Our results confirmed what we observed in the execution time graph 3: we have been able to improve the performances up to 16 processes, in all the scenarios represented by the different input sizes. When we increased the number of processes from 8 to 16, we noticed the most interesting behavior: in case of input size = 2^{20} the speedup decreases, with input size = 2^{21} it increases slightly and when it's 2^{22} the speedup increases greatly (table 8). This can be seen also in the efficiency domain where we observe that in case of input size = 2^{22} it keeps the value over 80% until the 16 processes configuration (table 9). When we run our program with 32 processes we observe a dramatic drop in performance with all input sizes.

3.6 Visualization

We built a **Python** program to plot the algorithm steps to better show how the computation is actually split among the working processes. When launching the parallel program from command line, in this case in the shell script, it is possible to enable a boolean flag (`argv[3]`) which will enable the logs of all the relevant steps taken by each MPI process into several `.txt` files. The Python program, starting from those file, simulates the behaviour of the processes in a visual manner. A snapshot of the animation can be seen in figure 6.



(a) Serial plotting

(b) Parallel plotting

Figure 6: Snapshots of the execution of the serial and parallel "divide et impera" algorithm

3.7 Project organization

The following is how the main part of the project is organized in the repository:

- `convex_hull.c`: contains the functions used by both the serial and parallel implementation.
- `serial.c`: contains the serial implementation.
- `parallel.c`: contains the parallel implementation.
- `Makefile`: can be used to compile and run the programs.

In the README of the repository there is a section explaining how to compile and run the code taking advantage of the Makefile.

4 Final discussion

4.1 Outcomes and results accomplished

First of all, the various resulting convex hulls were verified to be equivalent to the ones obtained using the classical serial method, providing confidence in the **correctness** of the parallel implementation (3.3). With our parallel implementation, as seen in section 3.5 the convex hull was calculated in a **fraction of the time** it would have taken using the serial approach.

Furthermore, as is well known, the benchmark confirmed that, even in this specific scenario, using more processors does not always result in better running time performance, but it is always necessary to do a proper balancing of resources used and taking into account the overheads introduced.

The results confirmed that the choice of implementing a "divide et impera" algorithm was correct, since it is almost 100% parallelizable as evidenced by the efficiency table 9. It would be interesting to investigate the behaviour with much larger input sizes, because our experiments revealed an improvement limit of 16 processes, which could be pushed further.

4.2 Further improvements

A possible modification, to improve the flexibility of the implementation, could be to switch to a **Mesh**-structured parallel algorithm instead of the current Tree-structured. This will allow the exploitation of a number of MPI processes different from a power of 2.

Another possible improvement could leverage the `omp task` or `omp taskgroup` constructs, introduced in recent OpenMP versions, which could be used in order to tackle recursion according to the OpenMP community. [8] [9] [10] [11]

A Appendix

Input size / n. processes	2^20	2^21	2^22
1	0.266	0.639	1.298
2	0.183	0.340	0.669
4	0.083	0.186	0.369
8	0.049	0.096	0.197
16	0.055	0.075	0.099
32	0.140	0.156	0.141

Figure 7: Execution time averaged over 8 runs, after discarding minimum and maximum

Input size / n. processes	2^20	2^21	2^22
1	1.00	1.00	1.00
2	1.45	1.88	1.94
4	3.18	3.44	3.52
8	5.41	6.69	6.59
16	4.83	8.51	13.05
32	1.90	4.11	9.21

Figure 8: Speedup averaged over 8 runs, after discarding minimum and maximum

Input size / n. processes	2^20	2^21	2^22
1	100%	100%	100%
2	73%	94%	97%
4	80%	86%	88%
8	68%	84%	82%
16	30%	53%	82%
32	6%	13%	29%

Figure 9: Efficiency averaged over 8 runs, after deleting minimum and maximum

References

[1] F. P. Preparata and S. J. Hong. Convex hulls of finite sets of points in two and three dimensions. *Communications of the ACM*, 20(2):87–93, 1977.

[2] Convex hull divide and conquer serial implementation. https://www.geeksforgeeks.org/convex-hull-using-divide-and-conquer-algorithm/. Accessed: 2022-10-27.

[3] OpenMP clauses. https://learn.microsoft.com/en-us/cpp/parallel/openmp/reference/openmp-clauses.

[4] OpenMP examples. https://www.openmp.org/wp-content/uploads/OpenMP_Examples_4.0.1.pdf.

[5] Convex hull calculator tool. https://planetcalc.com/8576/.

[6] Graphical calculator. https://www.desmos.com/calculator.

[7] *Altair PBS Professional™ 19.2.3 Reference Guide*.

[8] How to get good performance by using OpenMP. http://akira.ruc.dk/~keld/teaching/IPDC_f10/Slides/pdf4x/4_Performance.4x.pdf.

[9] Mastering OpenMP performance. https://www.openmp.org/wp-content/uploads/openmp-webinar-vanderPas-20210318.pdf.

[10] Advanced OpenMP. https://www.archer.ac.uk/training/course-material/2019/06/AdvOpenMP-manch/L02-Tasks.pdf.

[11] Introduction to OpenMP. https://www3.nd.edu/~zxu2/acms60212-40212-S12/Lec-11-03.pdf.