Closest Pair of Points

High Performance Computing for Data Science

Kevin Depedri, Davide Modolo

Problem Description

Finding the two points in a given set of points that are closest to each other in terms of their Euclidean distance

Implemented in ***k*** dimensions and extended returning multiple pairs

Assumptions

- the minimum distance between two points in the space is less than INT_MAX
- the maximum number of points in the space is less than (or equal to) INT_MAX
- there are no duplicate points
- coordinates are integers

Points Generator

Python script that, given a ***n*** number of points, a ***k*** number of dimensions and a range, it creates a text file containing such points

We also created a Python 2 version to generate an high amount of points using the cluster

Squential Implementations

Bruteforce and *Divide et Impera*

Bruteforce

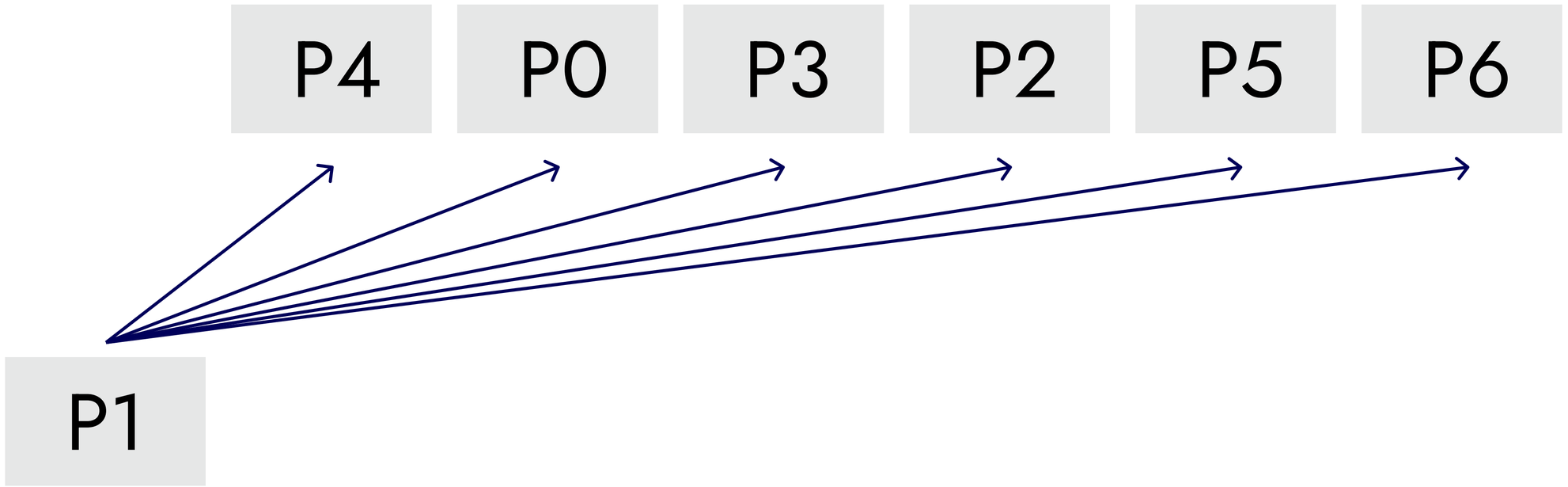
Implementation

Bruteforce

P1	P4	P0	P3	P2	P5	P6

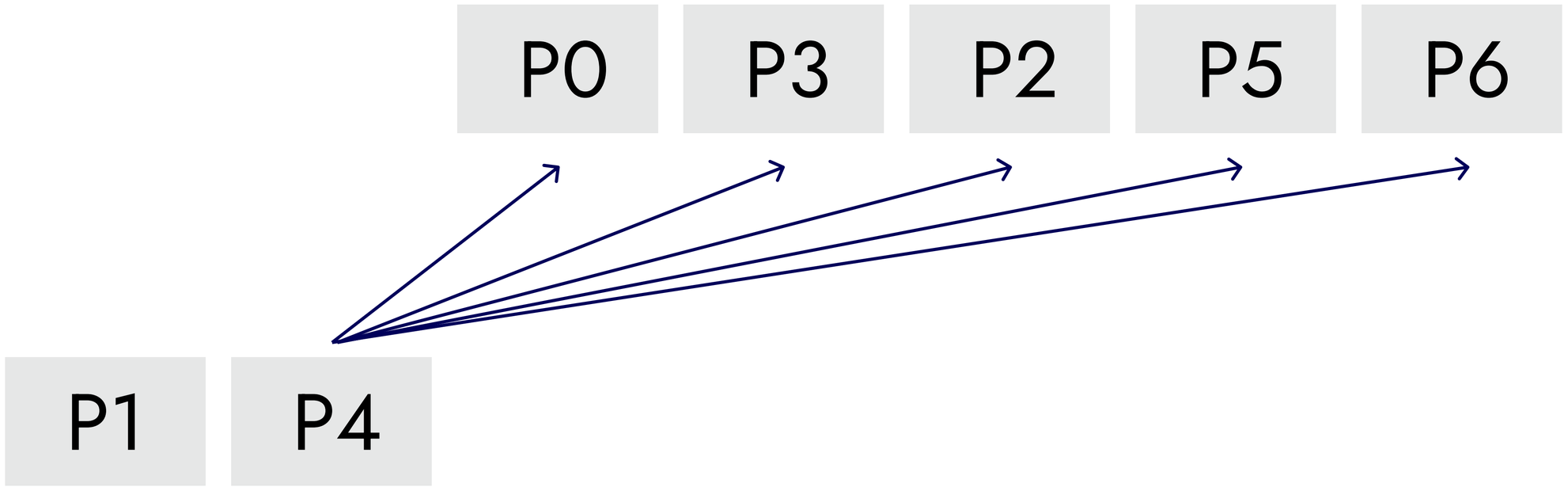
Start with set of points

Bruteforce



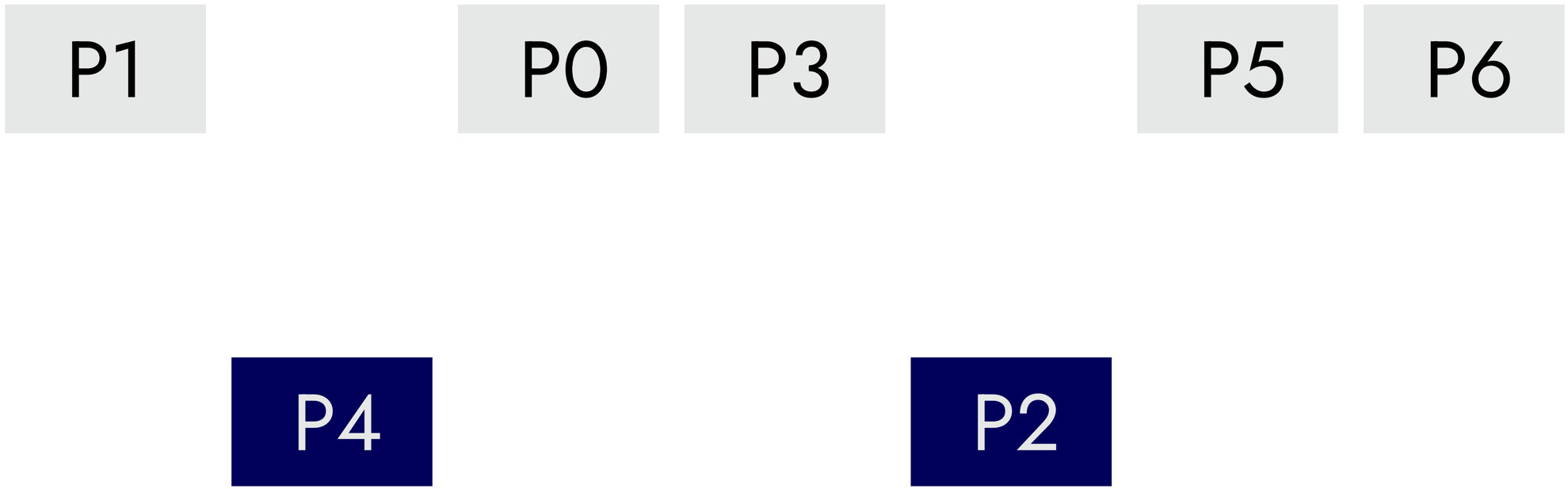
Compare each point with all the next points

Bruteforce



Compare each point with all the next points

Bruteforce



Return the pair(s) with minimum distance

Bruteforce

It worked, but with $O(n^2)$ time complexity it required too much time

Divide et Impera

Implementation

Sorting

This approach requires sorting multiple times, one time over ***x*** and a few times over ***y***. The sorting algorithm we chose is the Merge Sort

Divide et Impera

P0	P1	P2	P3	P4	P5	P6

Start with the sorted set of points

Divide et Impera

P0 P1 P2

P3 P4 P5 P6

Recursively split the space in half

Divide et Impera

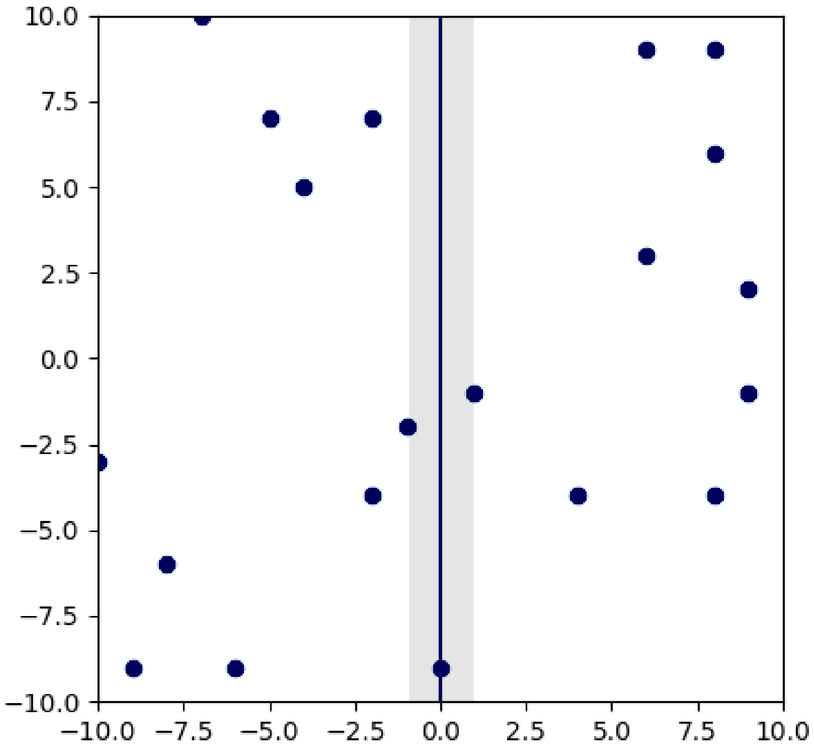
P0 P1 P2

P3 P4 P5 P6

Recursively split the space in half
until 2 or 3 points

Divide et Impera

- At each level of the recursion, the minimum distance is computed in both halves
- a strip of size $2d_{min}$ is built (gray area)
- points are sorted over *y* and compared to find a potentially new minimum distance



Divide et Impera

Given its *O(nlogn)* time complexity, it can handle this problem much better than Bruteforce, making it feasible even on our personal machine

Parallel Implementations
Bruteforce and *Divide et Impera*

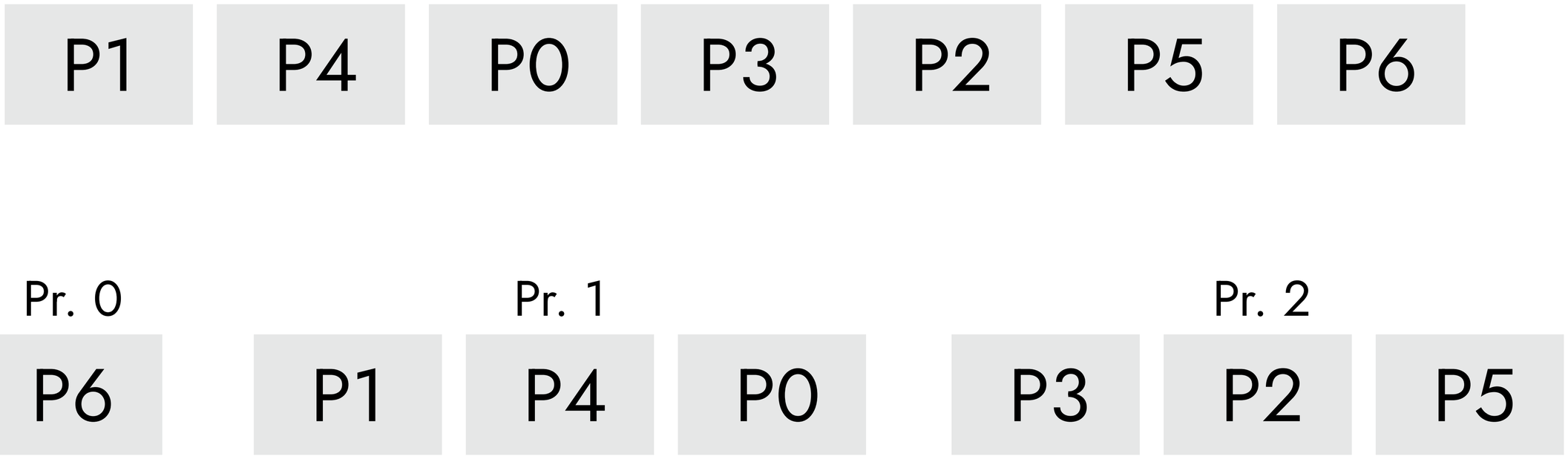
Parallel Bruteforce

Implementation

Parallel Bruteforce

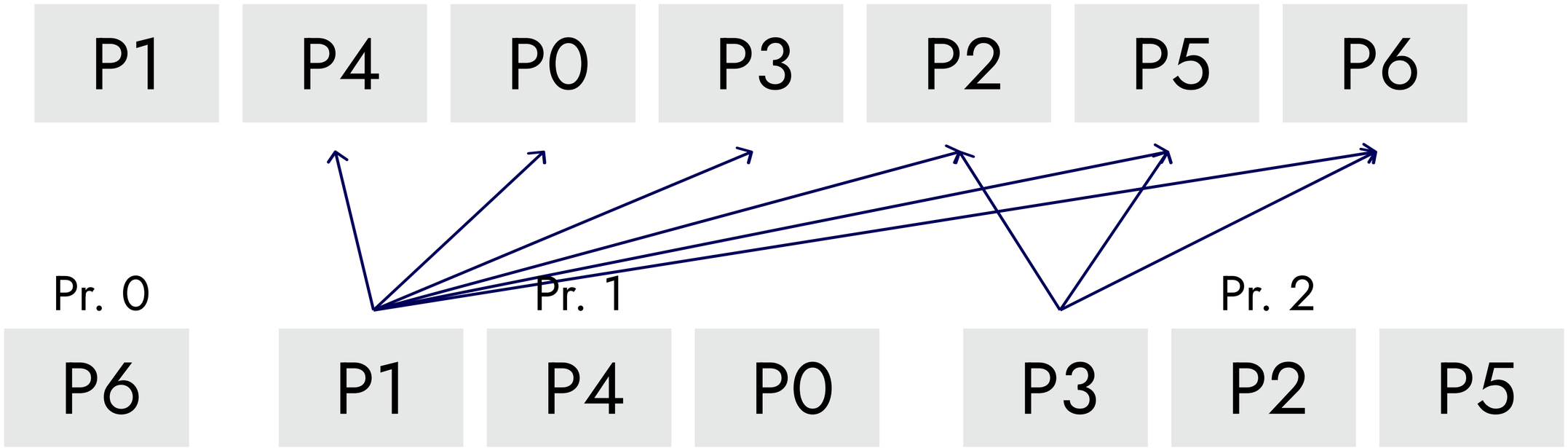
1. Process 0 reads the set of points
2. It sends to each other process all the points
3. Each process computes its slice to work with

Parallel Bruteforce



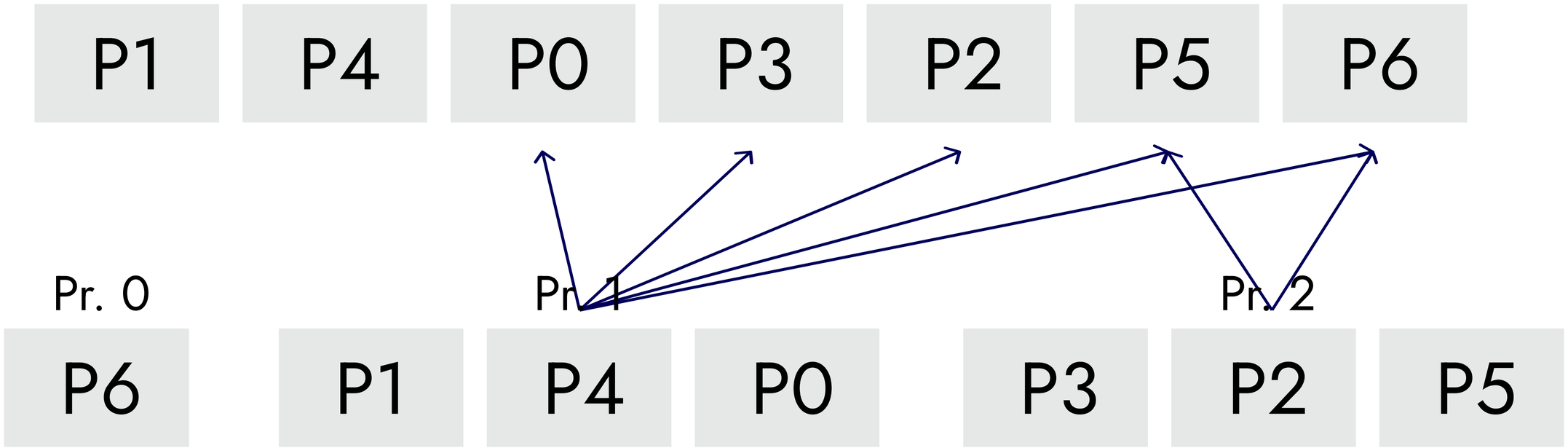
4. Compare each point with all the next points

Parallel Bruteforce



4. Compare each point with all the next points

Parallel Bruteforce



4. Compare each point with all the next points

Parallel Bruteforce

5. Allreduce call with MPI_MIN to share the minimum distance found

In the making, each process saved all points pairs that gave it minimum distance.
After the Allreduce call, if the global value equals the local one, they print their pairs

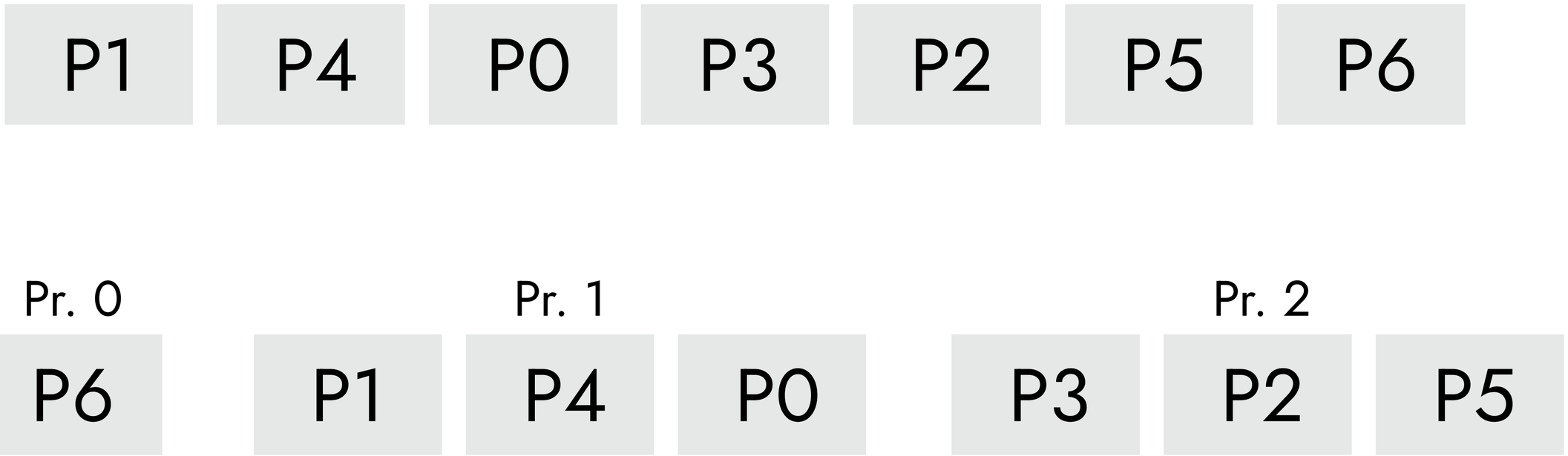
Parallel Mergesort

Implementation

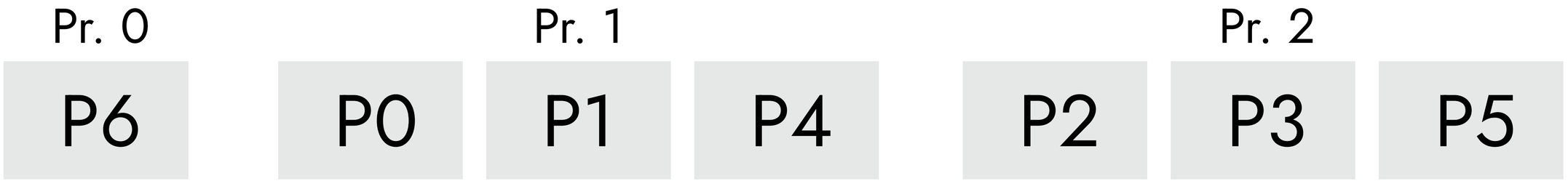
Parallel Mergesort

1. Process 0 reads the set of points
2. It sends to each other process an equal slice of points (keeping any leftovers for itself)

Parallel Mergesort

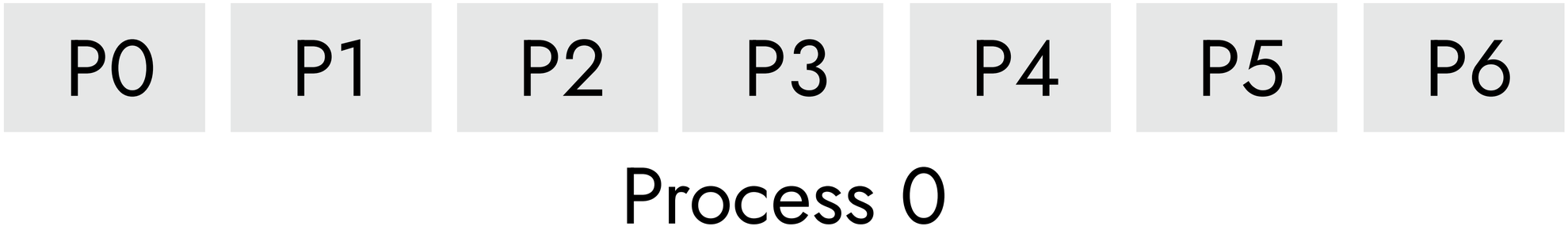


Parallel Mergesort



3. Each process sorts its subset of points

Parallel Mergesort



4. Process 0 receives and merges all the sorted subsets

Parallel Divide et Impera

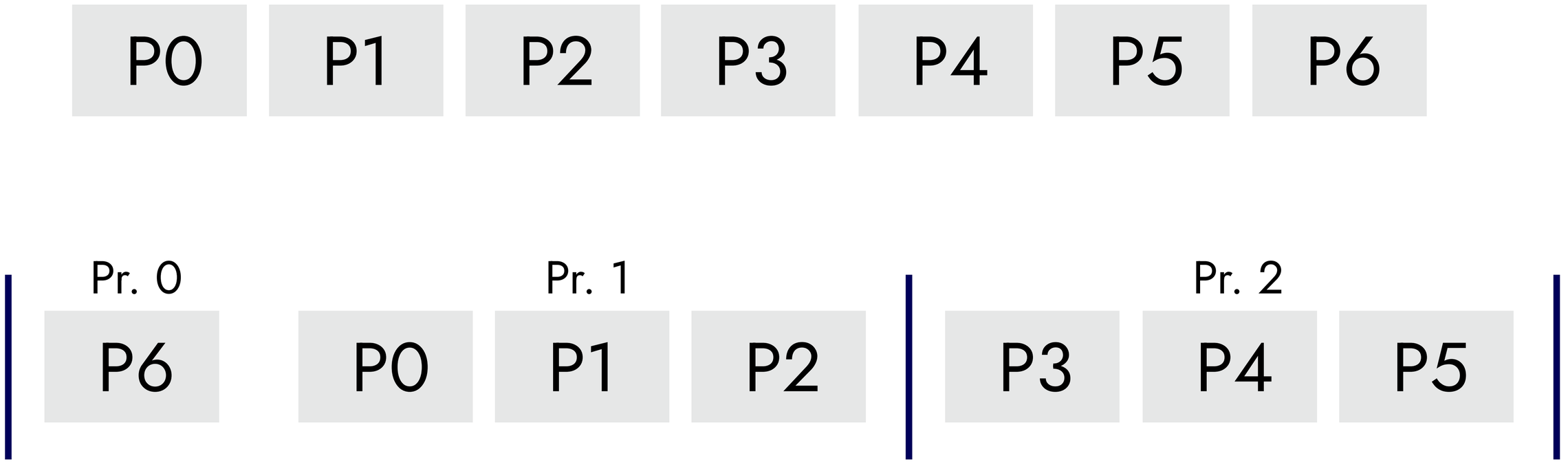
Implementation

Parallel Divide et Impera

Process 0 sends to the other processes:

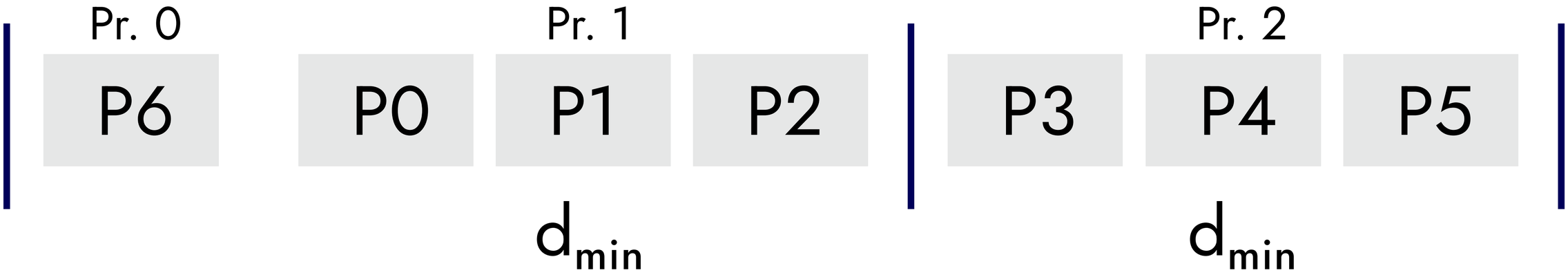
- the portion of points every process will work with
- the x value of the last element of the previous process
- the x value of the first element of the next process

Parallel Divide et Impera



Each process acknowledges the border that divides its space from the adjacent processes

Parallel Divide et Impera



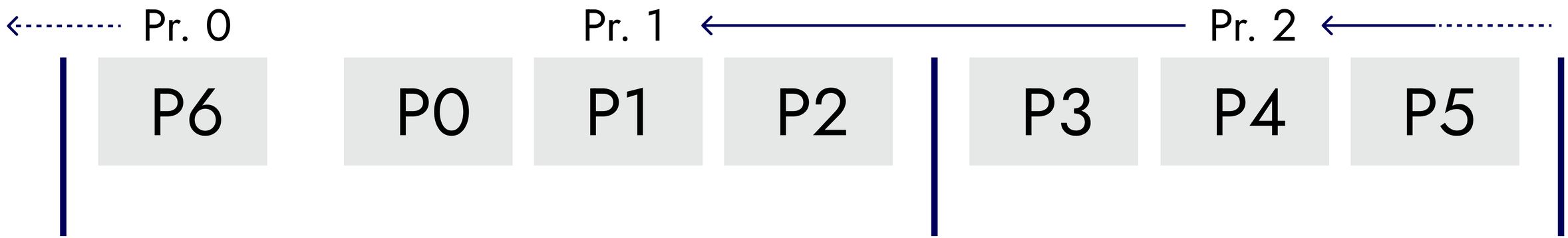
Each process computes the closest pair with sequential algorithm getting its local d_{min}

Parallel Divide et Impera

Then:

- Allreduce to get the global minimum distance found
- Processes 1 to N computes its right half of the strip
- Each process other than 1 computes its left half of the strip

Parallel Divide et Impera



Each process sends its left strip to the previous process that merges it with its right half

Parallel Divide et Impera

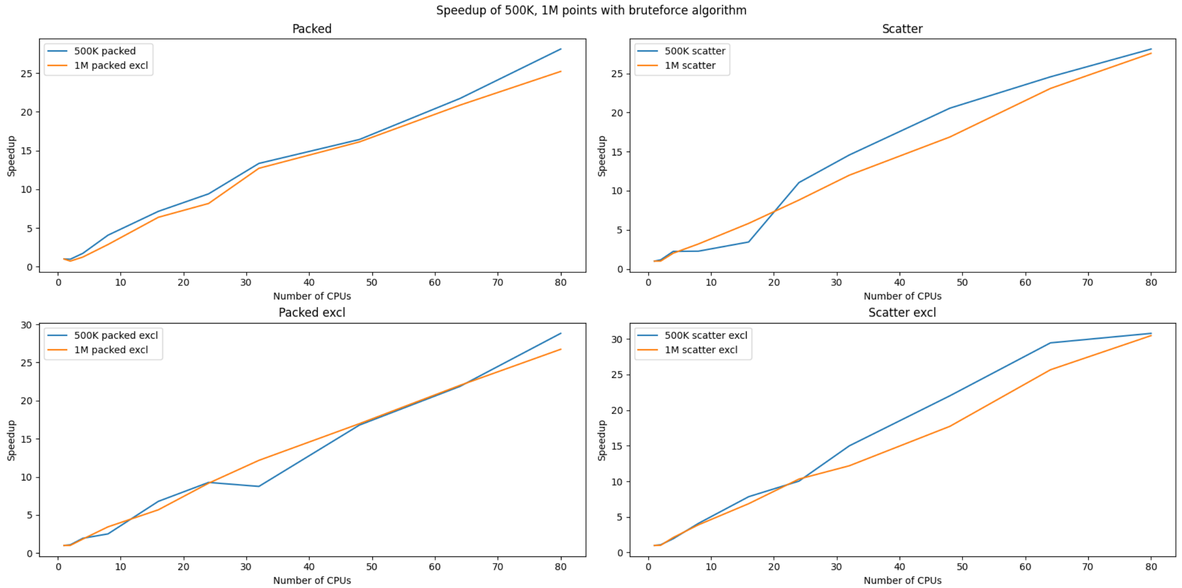
- Each process computes the second part of the Divide et Impera algorithm, using the strip computed before
- Another call of Allreduce to find eventually the new global distance
- As in Bruteforce, process print their pairs if the global distance equals theirs

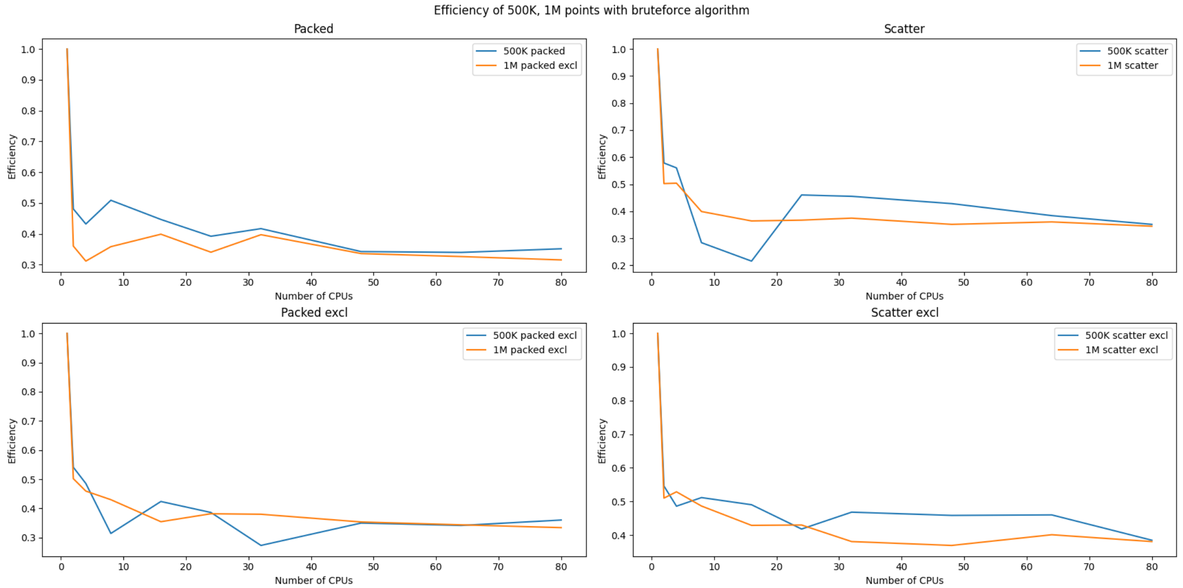
Results and considerations

Bruteforce and *Divide et Impera*

Results - Bruteforce

Closest Pair of Points

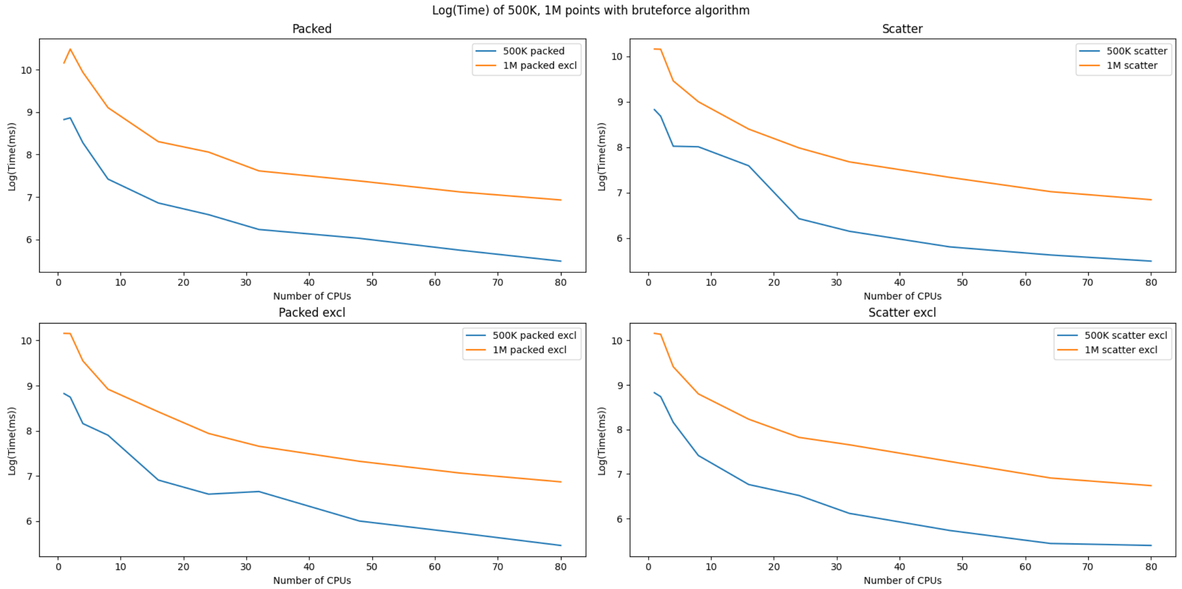


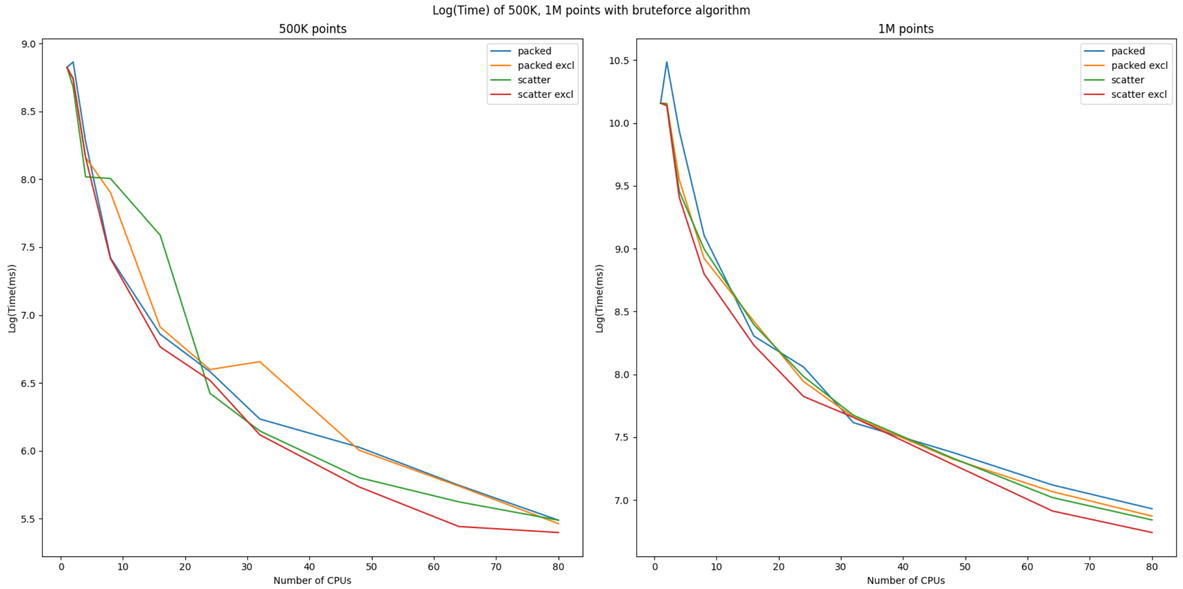


- Speedup almost linear
- Nice parallelization capability
- Efficiency easily drops

Results - Bruteforce

Closest Pair of Points

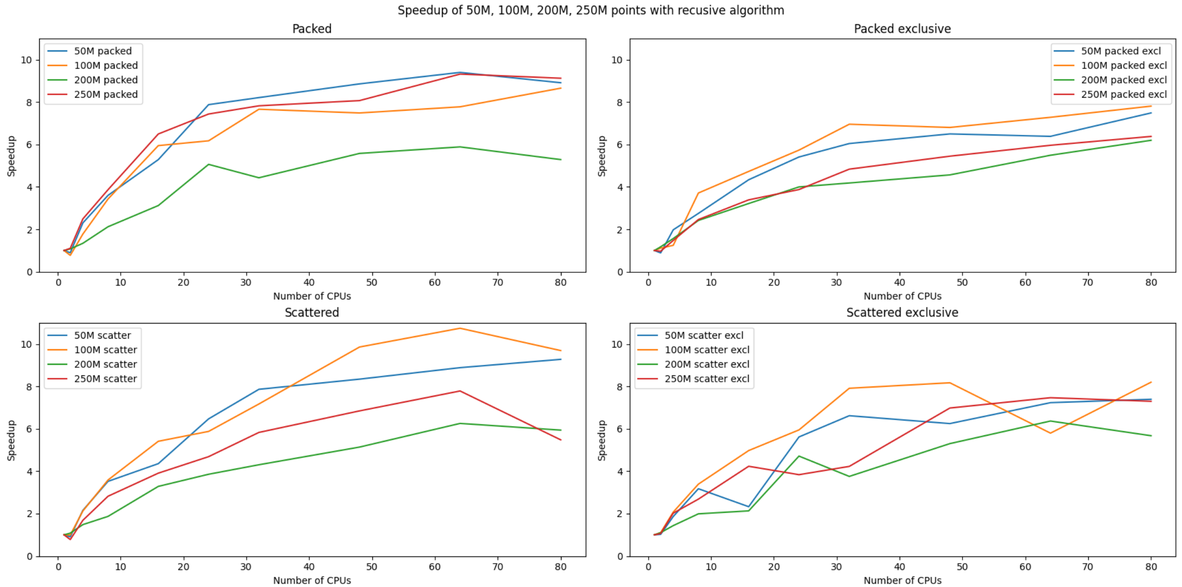


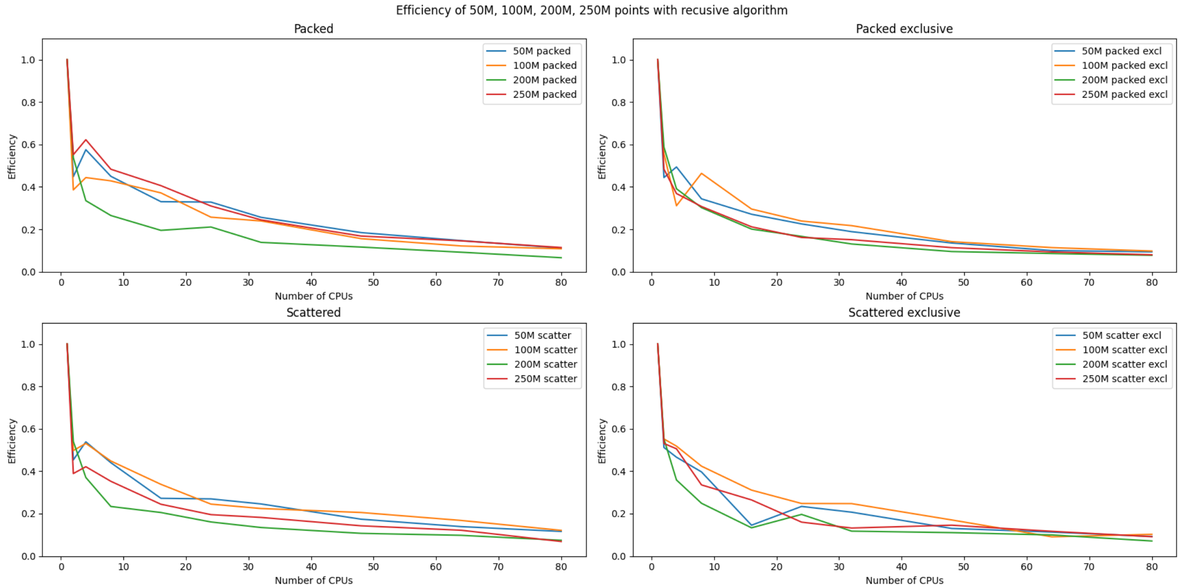


- Similar performances among different configurations
- Time depends mostly on #cores and input size
- Still too slow with respect to Divide et Impera

Results - Divide et Impera

Closest Pair of Points

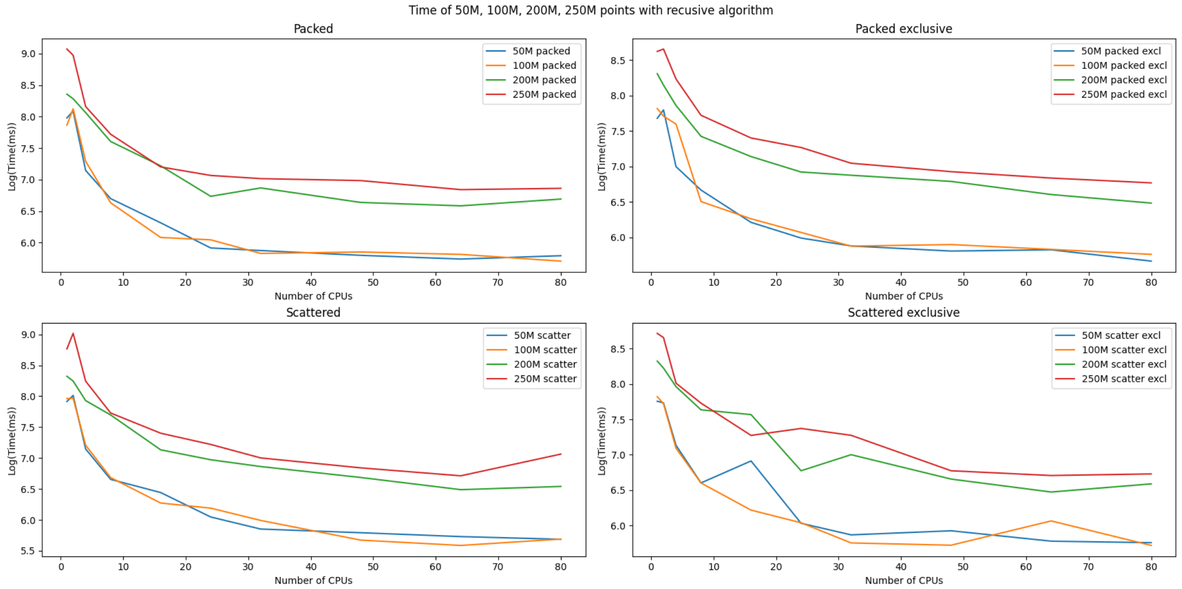


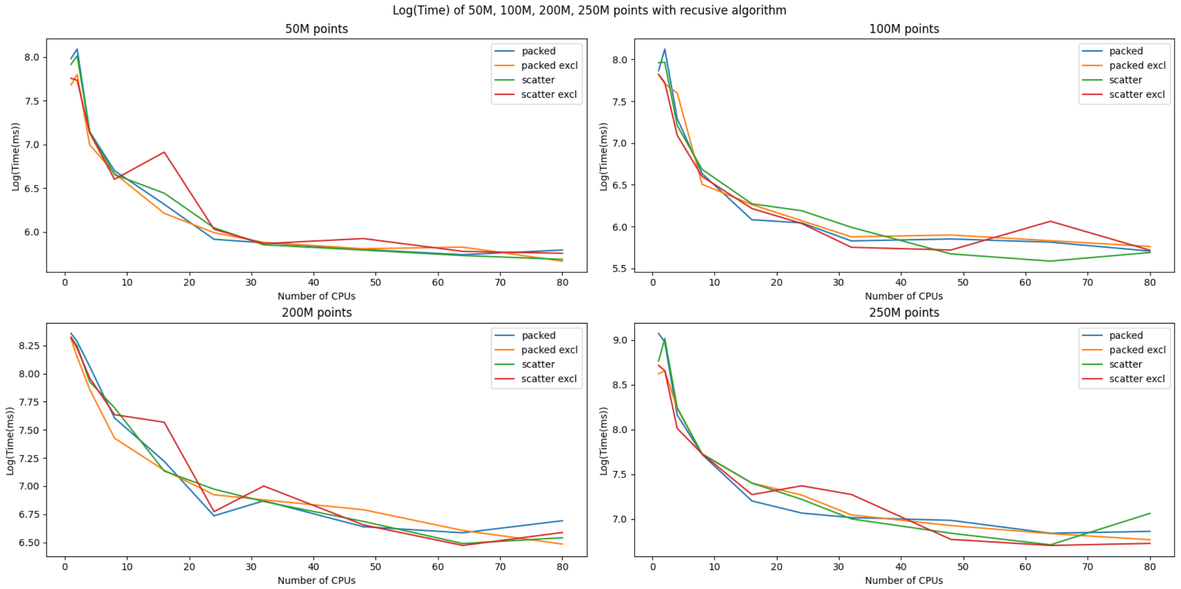


- Speedup with similar slope on different configurations
- Results in an efficiency that keeps decreasing
- Algorithm not fully paralellizable

Results - Divide et Impera

Closest Pair of Points





- Again similar curves among different configurations
- Time depends mostly on input size
- It saturates quickly

Closest Pair of Points

Thanks

Kevin Depedri, Davide Modolo