EXERCISE 2B
HIGH PERFORMANCE COMPUTING

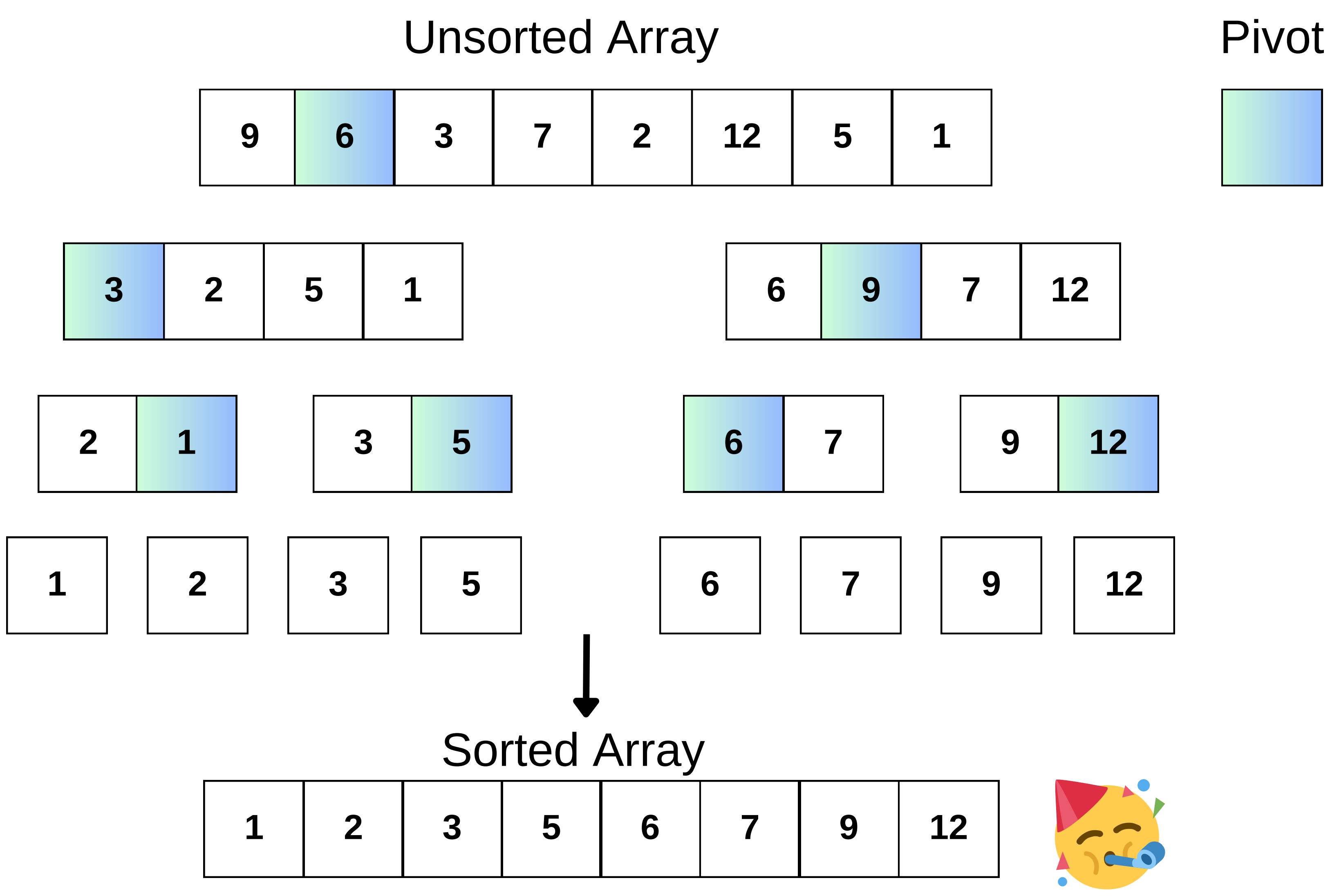
Giulio Fantuzzi

A QUICK LOOK AT QUICKSORT ALGORITHM

- Sorting algorithm that follows the divide-and-conquer paradigm;
- Consider an array X with n elements;
- Select an element (called pivot) x_{i^*}, with $i^* \in \{0, ..., n-1\}$;
- Divides the array into the following two partitions:

$$X_< = \{x_i \in X : x_i < x_{i^*}, i \in \{0, ..., n-1\}\}$$
$$X_\geq = \{x_i \in X : x_i \geq x_{i^*}, i \in \{0, ..., n-1\}\}$$

- Recursive call in each partition



PARALLELIZATION STRATEGY: OMP

Algorithm 1 Pseudocode of OMP Quicksort

```
procedure OMP_QUICKSORT(data, start, end, cmp_ge)
    size ← end − start
    if size > 2 then
        mid ← PARTITIONING(data, start, end, cmp_ge)
        #pragma omp task
        OMP_QUICKSORT(data, start, mid, cmp_ge)
        #pragma omp task
        OMP_QUICKSORT(data, mid + 1, end, cmp_ge)
    else
        if (size == 2) and (data[start] ≥ data[end − 1]) then
            SWAP(data[start], data[end − 1])
        end if
    end if
end procedure
```

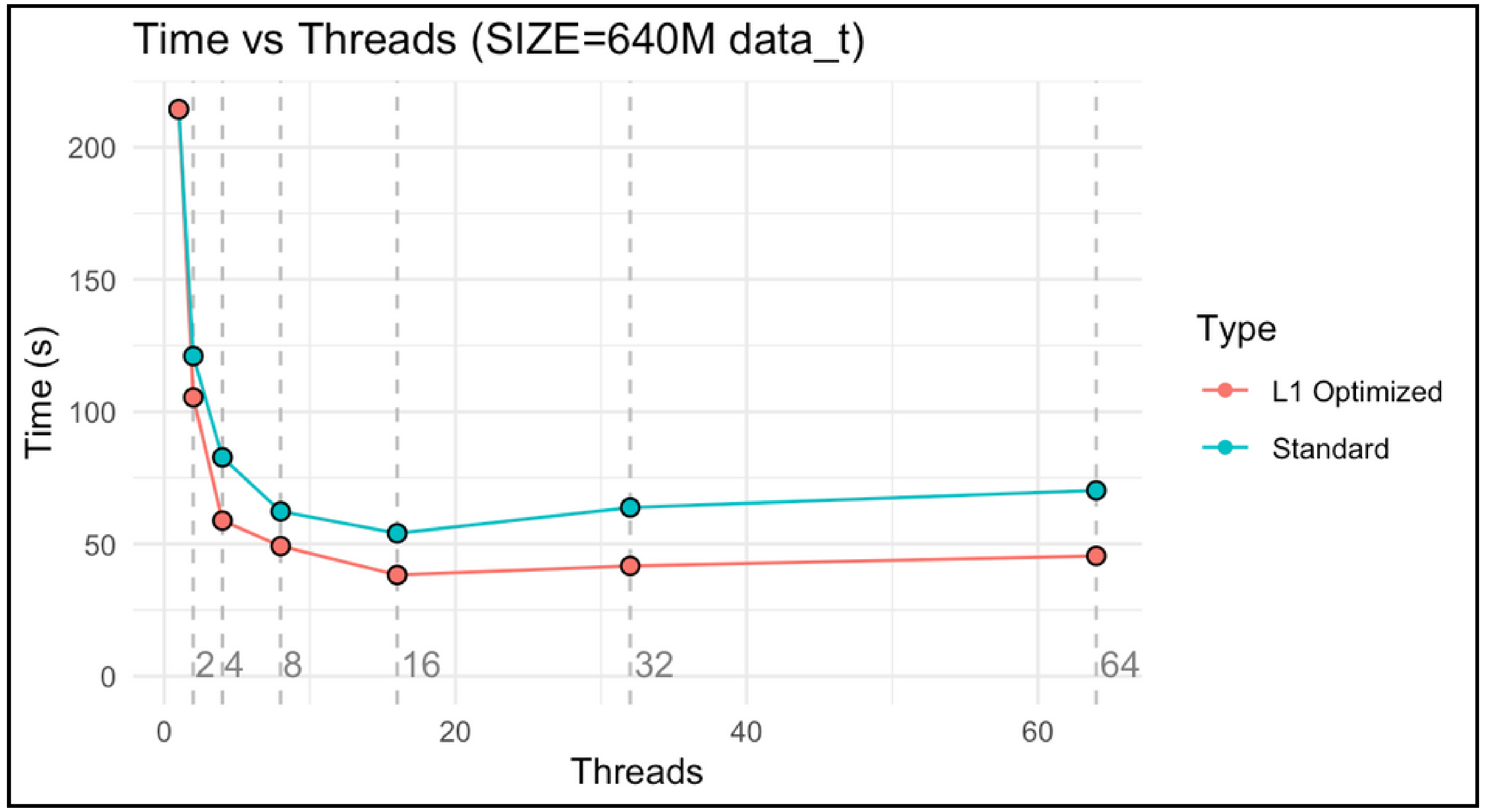
- Parallelism introduced primarily through the use of `omp tasks`
- Tasks facilitate the parallel execution of recursive calls, allowing the partitions $X_<$ and $X_\geq$ to be sorted concurrently;
- Tasks allow for a dynamic scheduling: runtime system intelligently distributes tasks to the available threads, as they become accessible
- Valuable flexibility in scenarios where the workload of each task is unpredictable at compile-time (*pivot-partition size* dependency)

OMP L1-CACHE OPTIMIZATION

Algorithm 2 Pseudocode of OMP Quicksort (L1 optimized)

procedure OMP_QUICKSORT_L1($data, start, end, cmp\_ge$)
  $size \leftarrow end - start$
  if $size > 2$ **then**
    $mid \leftarrow$ PARTITIONING($data, start, end, cmp\_ge$)
    if $size > \boxed{SIZE\_L1}$ **then**
      #pragma omp task
      OMP_QUICKSORT($data, start, mid, cmp\_ge$)
      #pragma omp task
      OMP_QUICKSORT($data, mid + 1, end, cmp\_ge$)
    else
      SERIAL_QUICKSORT($data, start, mid, cmp\_ge$)
      SERIAL_QUICKSORT($data, mid + 1, end, cmp\_ge$)
    end if
  else
    if ($size == 2$) **and** ($data[start] \geq data[end - 1]$) **then**
      SWAP($data[start], data[end - 1]$)
    end if
  end if
end procedure

- `data_t`: array of `DATA_SIZE` doubles
- Default `DATA_SIZE`=8 $\longrightarrow$ `data_t` requires 64 bytes;
- EPYC nodes equipped with an L1 cache of 4 MiB $\longrightarrow$ 4194304 bytes
- Max number of `data_t` that can fit within the L1 cache: `L1_SIZE` = 65536



MPI APPROACH

INITIAL SETUP

- $X_0, ..., X_{P-1}$ data chunks generated by processes $P_0, ..., P_{P-1}$
- Initially, all the segments $X_0, ..., X_{P-1}$ are unordered
- Processes are divided into 2 groups basing on their rank:
  1. upper processes (with rank $p > \frac{P-1}{2}$)
  2. lower processes (with rank $p \leq \frac{P-1}{2}$)

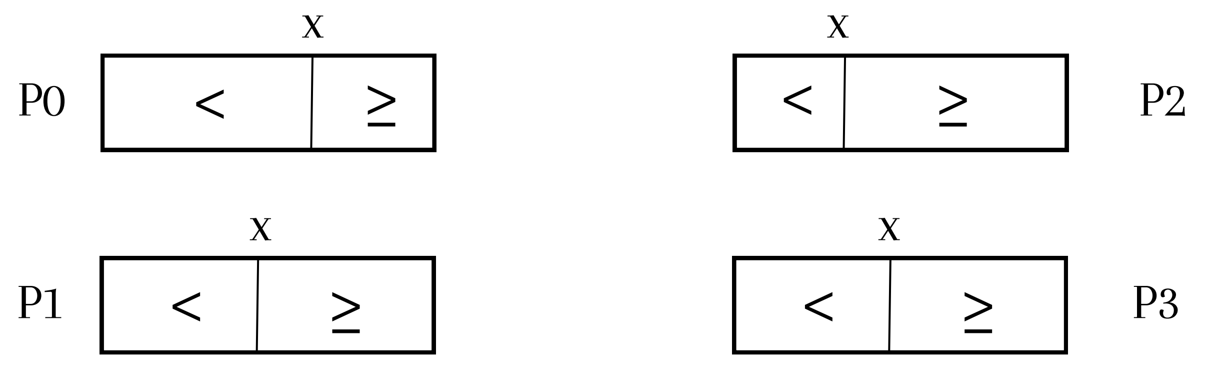
OBJECTIVE

$$X_0 \leq ... \leq X_{P-1}, \text{ with } X_i \leq X_j \iff \forall x_i \in X_i, \forall x_j \in X_j, x_i \leq x_j$$

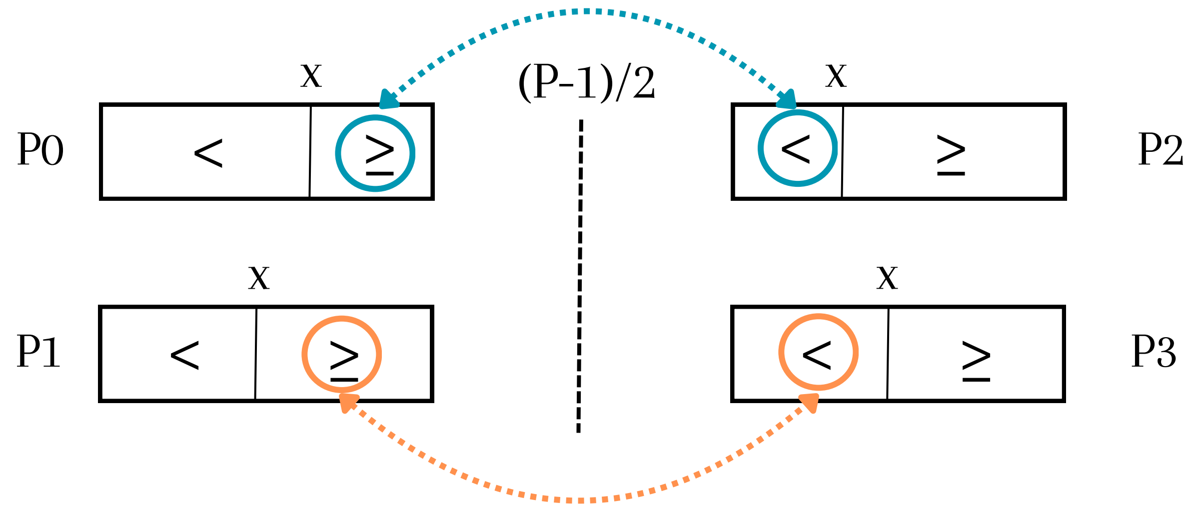
PROCEDURE

1. Select a global pivot and broadcast it to all the processes
2. Partition the chunk of each process p into 2 sub-arrays: $X_<^{(p)}$ and $X_\geq^{(p)}$
3. Exchange data portions between each process p of the upper group and process $p' = p - \left(\frac{P-1}{2} + 1\right)$ of the lower group. In particular:
   - p sends its lower partition $X_<^{(p)}$ to p'
   - p' sends its upper partition $X_\geq^{(p')}$ to p
4. Recursive call in each group of processes

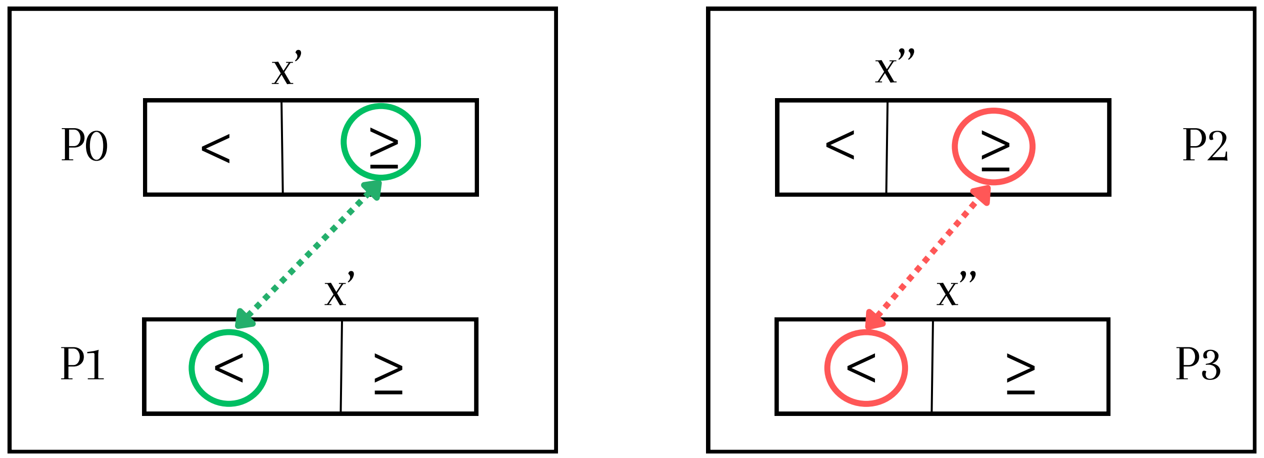
Step 1: select global pivot (x) and partition chunks



Step 2: group processes and exchange partitions



Step 3: recursive call on each group



PIVOT SELECTION AND LOAD IMBALANCE

- Quicksort's main drawback: processes might have very different workloads
- This aspect regards both the serial version both the hybrid configuration

SERIAL ALGORITHM ISSUES

- Pivot choice originally determined within the `partitioning()` function
- Lomuto partition scheme, which selects the rightmost element of the array
- 2 adjustments: *"Median of three"* rule + pivot positioning

HYBRID ALGORITHM ISSUES

How should we choose the global pivot?

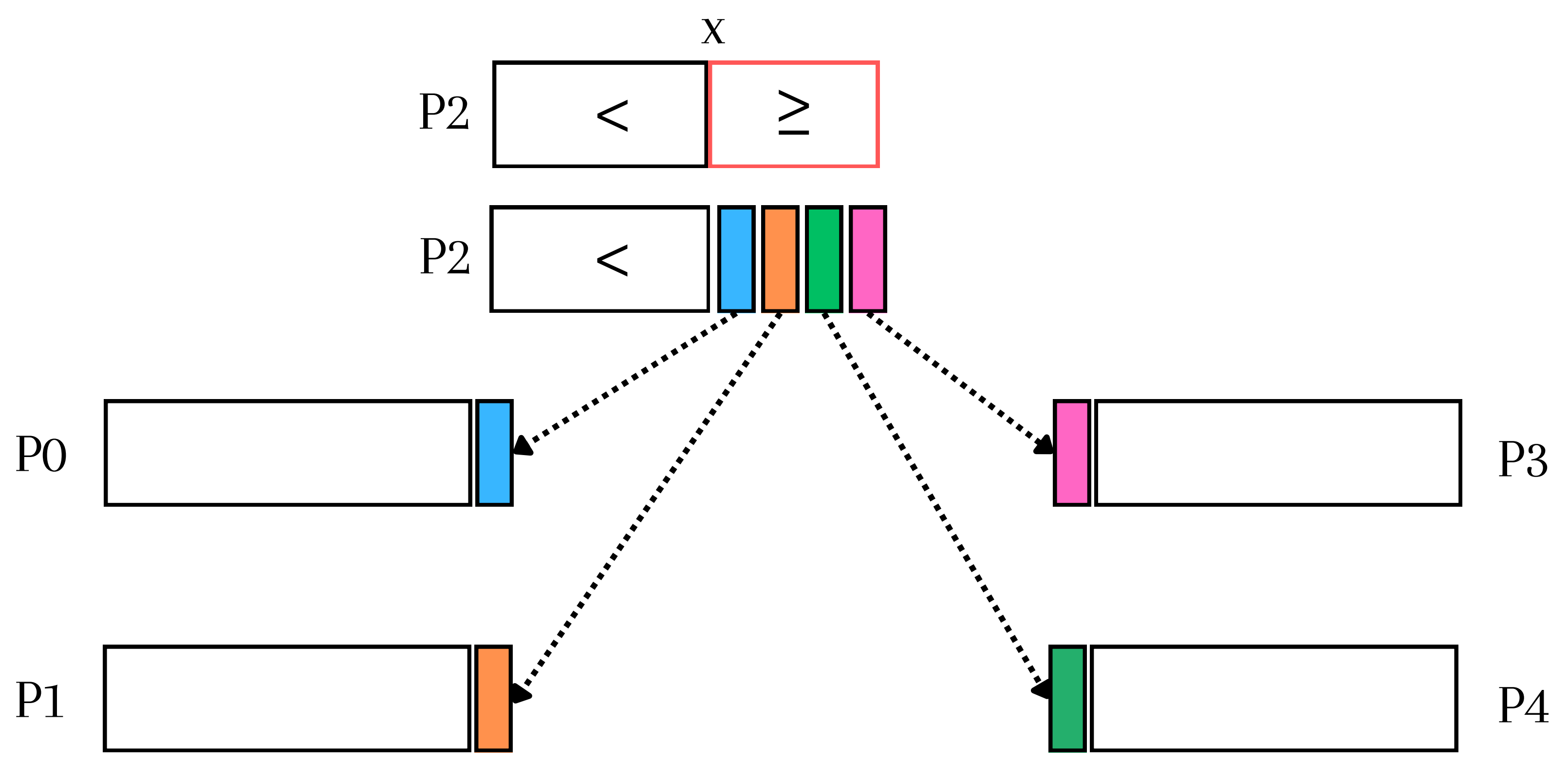
- Random choice not recommended
- *"Median of three"*-like strategy
- *Hyper-Quicksort* and *Parallel Sort Regular Sampling (PSRS)*
- My idea:
  1. Each process selects a random element from its chunk, with a seed based on its rank (just to enhance the degree of randomness)
  2. Such values are broadcasted to rank 0, populating an array of pivots
  3. The pivots array is then sorted, and its median is selected as the global pivot to start the partitioning phase

Algorithm 3 Pseudocode of "median-of-three"

$mid \leftarrow \frac{\text{start}+(\text{end}-1)}{2}$
if cmp_ge(data[start], data[mid]) **then**
    SWAP(data[start], data[mid])
end if
if cmp_ge(data[mid], data[end-1]) **then**
    SWAP(data[mid], data[end-1])
end if
if cmp_ge(data[mid], data[start]) **then**
    SWAP(data[start], data[mid])
end if
pivot $\leftarrow$ data[start]

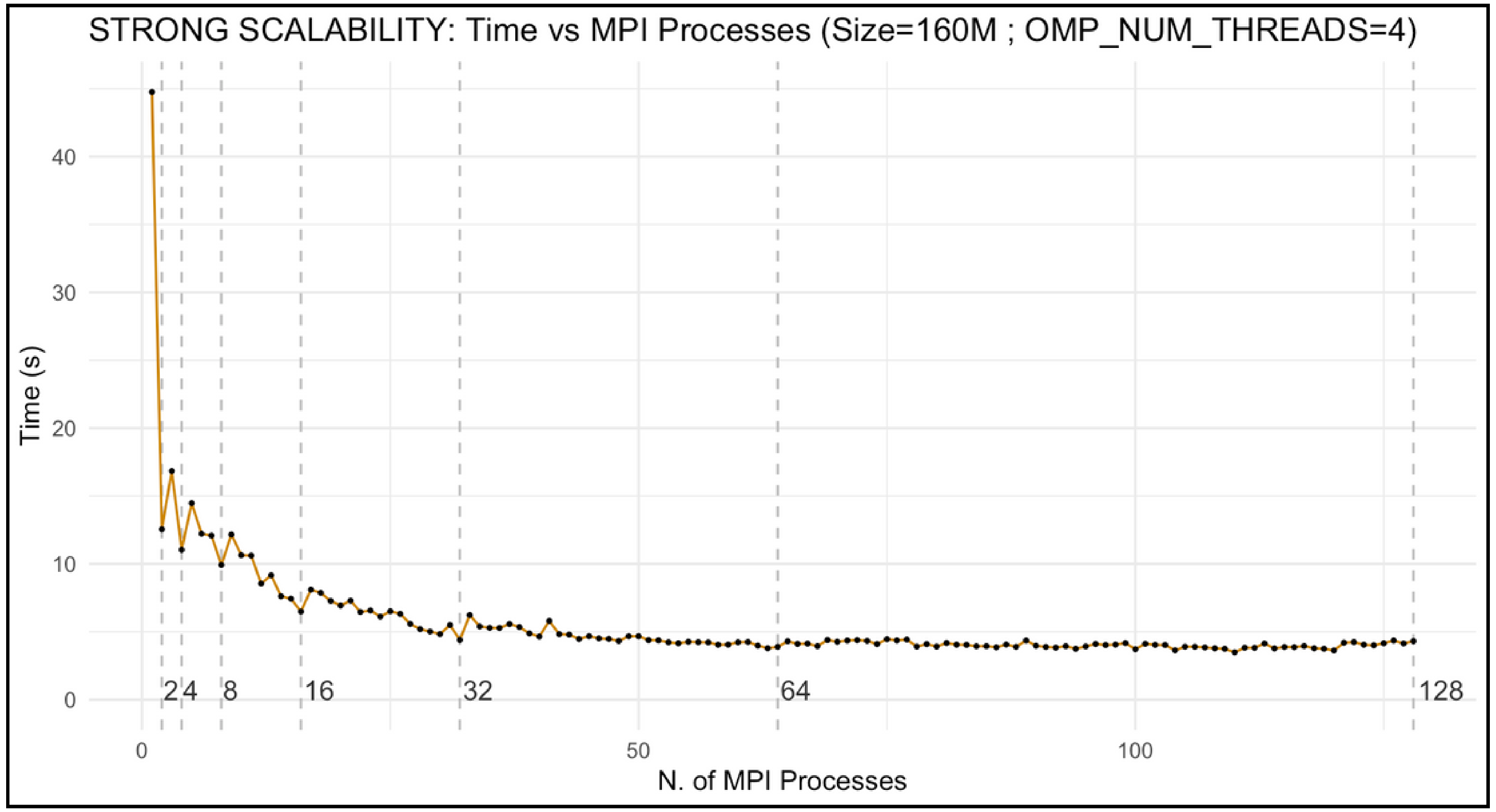
GENERALIZATION OF THE ALGORITHM

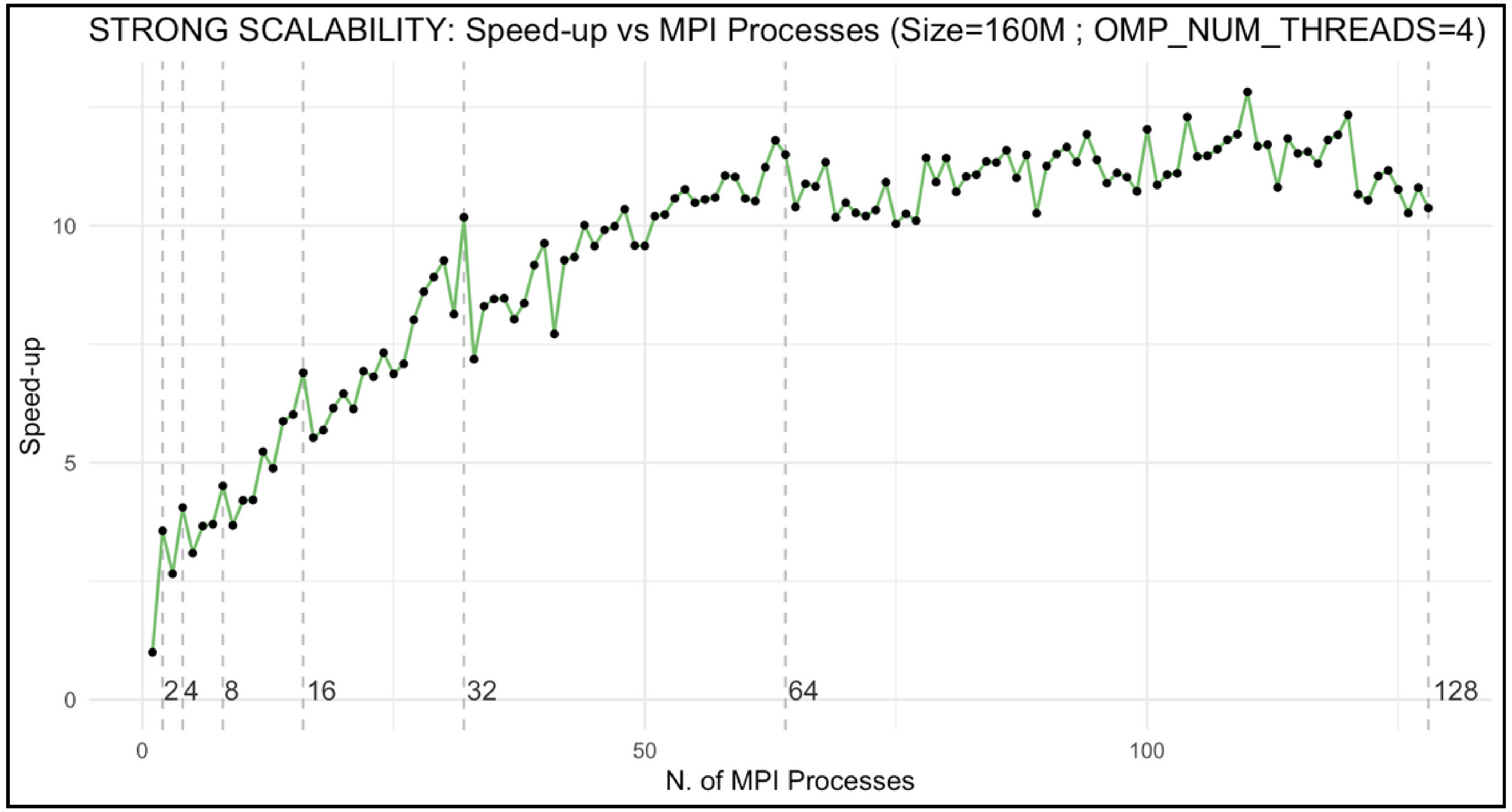
1. Select the global pivot and broadcast it to all the processes;
2. Partition the central process only (the one with rank $\frac{(P-1)}{2}$);
3. Compare the size of the two partitions $X_<^{\frac{(P-1)}{2}}$ and $X_\geq^{\frac{(P-1)}{2}}$. Scatter the elements of the minor partition to the other processes, while keeping the elements of the major partition;
4. Once concluded the scatter phase, proceed with the usual *partitioning and data exchange* procedure for all the other processes (with rank $p \neq \frac{(P-1)}{2}$);
5. Finally, assign the central process to the correct group for the recursive calls:
   - if the major partition was $X_<^{\frac{(P-1)}{2}}$, assign it to the group of processes with rank $p < \frac{(P-1)}{2}$
   - if the major partition was $X_\geq^{\frac{(P-1)}{2}}$, assign it to the group of processes with rank $p > \frac{(P-1)}{2}$



STRONG SCALABILITY

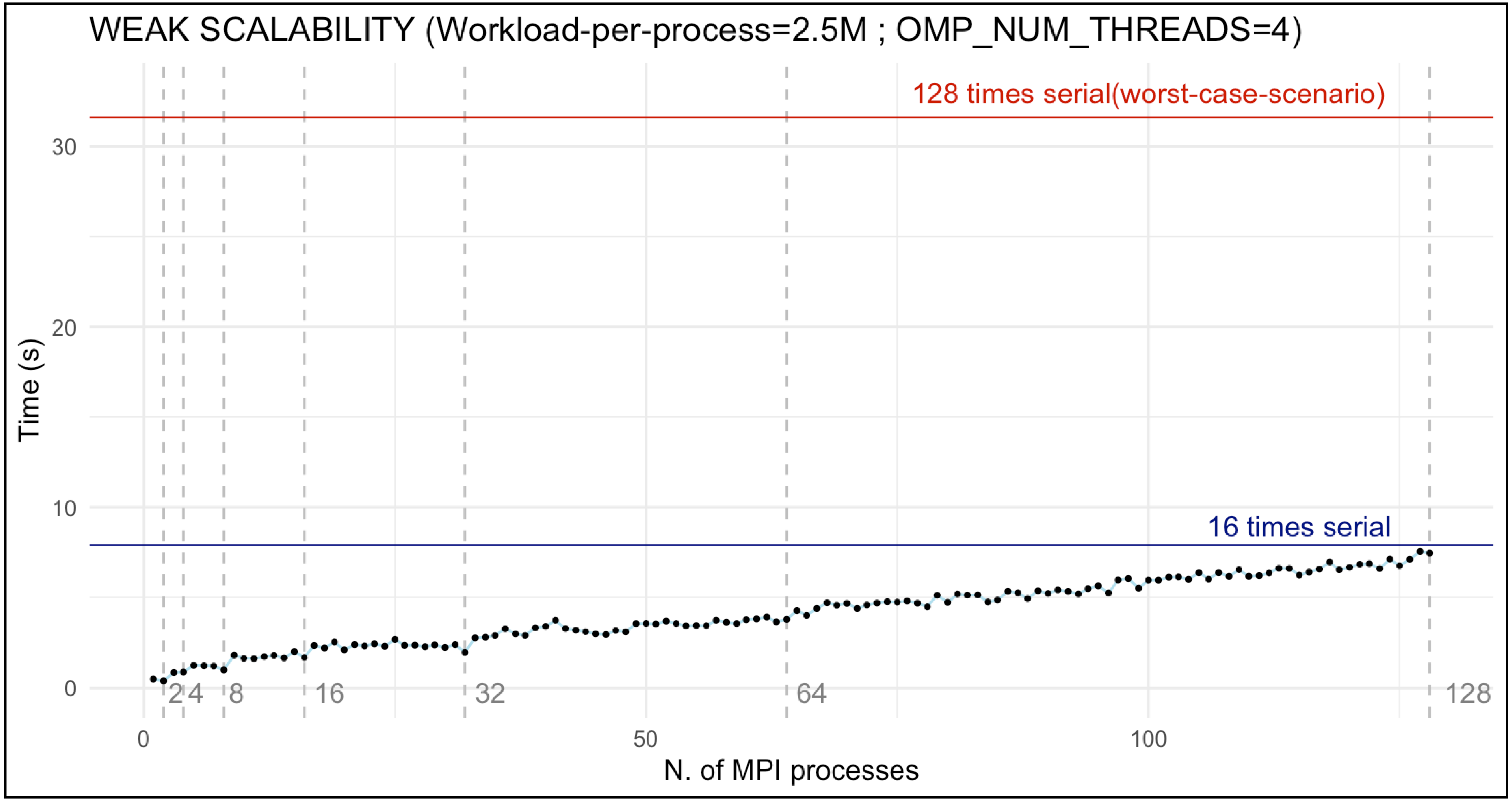
Strong scalability evaluation: size of the overall array fixed (to 160.000.000 `size_t`) and average sorting time (or, equivalently, the speed-up) measured by varying the number of MPI processes





WEAK SCALABILITY

Weak scalability evaluation: let the overall size n vary, keeping the workload-per-process as a constant (in my case, I opted for a workload of 2.500.000 `size_t`).



FINAL CONSIDERATIONS

- Both strong and weak scalability deviate from the ideal scenarios
- Strong scalability reveals a logarithmic trend
- Weak scalability upper-bounded by only 16 times the serial sorting time
- Clear limitations in scaling Quicksort
  - Recursive calls
  - Load imbalance issues
  - Exponential growth of MPI communications
- A worth highlighting achievement was the generalization of the algorithm