Luca Zanolo
luca.zanolo@studenti.unitn.it
University of Trento
Trento, Italy

I. Introduction

This project explores the parallel application of the Hough Transform (HT) for line detection on the high-performance computing (HPC) cluster at the University of Trento.

A. Overview

The project evaluates various implementations of the HT, beginning with the standard version [1] and extending to more complex variants such as the Probabilistic Hough Transform (PHT) [2] and the Progressive Probabilistic Hough Transform (PPHT) [3]. Each variant is implemented in both a sequential and multiple parallel versions, as summarized by Table I.

TABLE I
HT versions and parallelization strategy implemented.

HT Version	Sequential	MPI	OMP	Hybrid
HT	X	X	X	X
PHT	X	X	X	X
PPHT	X	X	X	

At the code level, the project employs both C++ and Python: C++ is used for developing the large part of the project, while Python handles dataset creation, image conversion and test generation. Key modules utilized include gcc91, OMPi-3.0.0–gcc-9.1.0, mpich-3.2.1–gcc-9.1.0, and python-3.7.2. The parallelization strategies for the HT and certain preprocessing steps leverage Message Passing Interface (MPI) or Open Multiprocessing (OMP) and a hybrid approach combining both MPI and OMP libraries. The execution of main program and the python scripts is controlled through the Portable Batch System (PBS) script, which allows for the configuration of different resource allocations to mi program on the HPC Cluster.

All the code and other materials related to the project are available on Github.

B. Report Structure

The report is organized as follow: next section introduces the HT and discusses its theoretical foundation and identifies opportunities for parallelization, avoiding implementation specifics. Subsequent sections present the project in detail: Section III makes an overview of the project organization, Section IV covers dataset organization, subsequent sections describe the various HT implementations and last Sections XII and XIII present the evaluation results and discuss them.

II. Hough Transformation

The HT is a technique in image analysis for detecting lines, curves and other parametric shapes. By transforming the detection problem from image space to parameter space, it simplifies the complex problem of shape detection into a peak detection task. This project focuses on line detection using various implementations of HT.

A. Hough Transform for Line Detection

The HT converts points in the image space (x, y) to a parameter space (ρ, θ). Here, ρ is the perpendicular distance from the origin $\circ$ to the line and θ is the angle between the x-axis and the line normal. The conversion is governed by:

$$\rho = x\cos(\theta) + y\sin(\theta) \tag{1}$$

This equation represents each image point as a sinusoidal curve in the parameter space. Lines in the image correspond to points of intersection of these curves in the parameter space. Figure 1 illustrates how a line can be represented by a pair (ρ, θ).

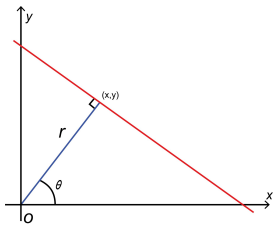


Fig. 1. A line represented with the related (ρ, θ) parameters highlighted. Original image from [4].

B. Hough Transform Algorithm

The standard HT involves several well-defined steps:

1) **Edge Detection:** Utilizing edge detection algorithms, such as the Sobel Edge Detection (SED) method, to preprocess the image.
2) **Parameter Space Definition:** Employing polar coordinates to define a comprehensive parameter space for line detection.
3) **Accumulator Array:** Creating an accumulator array to record votes for each parameter pair (ρ, θ).
4) **Voting:** Allowing edge points in the image to vote for all parameter pairs that could represent lines passing through them.
5) **Peak Detection:** Identifying high counts in the accumulator array which suggest the most likely lines in the image space.

This algorithm effectively maps edge points from input images into a parametric space where each point is characterized by the pair (ρ, θ) and an example of the resulting space is illustrated in Figure 2.

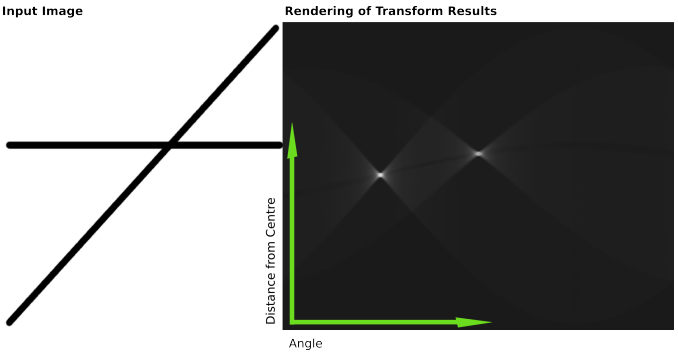


Fig. 2. Input image and related Hough space where points are represented in terms of (ρ, θ). Image from [4].

1) Probabilistic Hough Transform: The PHT modifies the standard HT by randomly sampling edge points rather than using it all. This approach, as detailed in [2], reduces the number of edge points considered speeding up the process while maintaining or improving the quality of the detections. PHT essentially changes the voting mechanism by limiting the number of participating points.

2) Progressive Probabilistic Hough Transform: The PPHT extends the PHT by dynamically adjusting the threshold for peak detection within the accumulator array. The algorithm proceeds by randomly selecting points for voting, continuously updating the accumulator and testing if the peak values exceed a dynamically computed threshold. When a line is detected, its supporting points reduce their votes and thus also their potential contribution to other segment detections.

C. Parallelization considerations

This section discusses potential parallelization strategies using both distributed memory programming with MPI and shared memory programming with OMP.

1) Distributed Memory Programming with MPI: Using MPI to parallelize different HT versions involves dividing the image or parameter space among multiple processes, each computing votes in a local accumulator. The primary challenge in this kind of parallelization is managing the communication overhead involved in sharing and combining data from all processes. First bottleneck is rapresented by how the image data are shared among processes and then the second one is rapresented by how the local accumulators are merged. Efficient communication, typically implemented via gathering or reduction operations to a master process will be crucial. Another significant issue is load balancing, which ensures that all processes have an approximately equal amount of work assigned preventing from scenarios where some nodes remain idle, waiting for others to complete their tasks.

2) Shared Memory Programming with OMP: In contrast, parallelizing HT versions using OMP involves parallelizing the loops that iterate through the image pixels and the angle calculations. Each thread can independently update a shared accumulator array with minimal setup, leveraging OMP capabilities to handle thread management and synchronization internally. However, this approach introduces potential data race conditions when multiple threads update the same cell of the accumulator array simultaneously. To address this, synchronization mechanisms such as critical or atomic operations may be employed. While these mechanisms ensure data integrity, they can also introduce significant performance overhead due to the costs associated.

3) Hybrid MPI and OMP Approach: An hybrid approach that combines MPI and OMP can also be employed. Here, MPI could be used to distribute parts of the image across different nodes, reducing the memory demands on any single node. Within each node, OMP can be utilized to further parallelize computations across available CPU cores, optimizing the use of local computational resources.

III. Project Organization

This section outlines the workflow and structural organization of the project. The high-level workflow is illustrated in Figure 3.

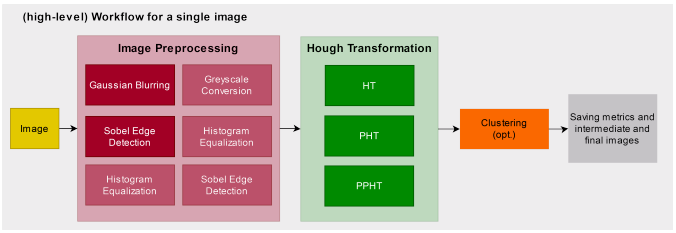


Fig. 3. Project pipeline.

The project pipeline of Figure 3 begins with reading necessary data and parameters from the file system, facilitated by a parameters file which path is provided as argument to the program. This file allows users to adjust image preprocessing steps, select the HT variant and related parameters, manage post-processing clustering and control file system interactions. The next phase is pre-processing the image, followed by applying the chosen HT version. Afterthat, the HT output may undergo a clustering operation to group lines together. The final step involves evaluating the detection effectiveness and saving the resulting images and metrics.

Figure 4 represents the project package diagram. The main file initiates the program, while the *HoughTransform* (.sh and .exe) objects represent the execution script and executable, respectively. The parameter file, as mentioned, contains specific parameters for different program phases and once built, the program can be executed multiple times with variations in these parameters.

The project folders are organized as follows:

1) **SRC folder:** Contains all C++ source files. Each component has a .cpp file with the implementation and a .h file with function signatures and descriptions of the operations.

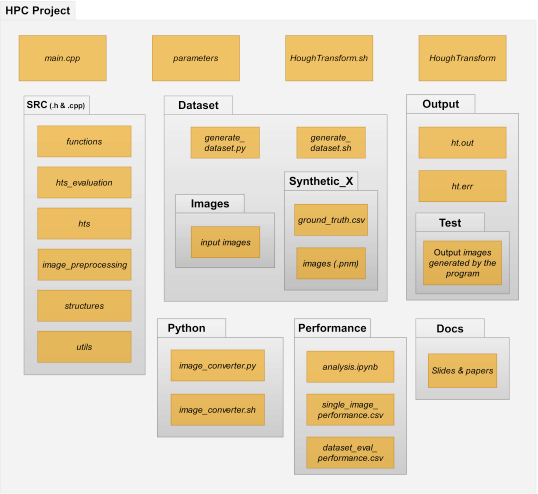


Fig. 4. Project package diagram.

2) **Python folder:** Contains Python code for image generation, conversion and test case generation. The image conversion program is also used by the main C++ program when requested in the parameters file.
3) **Output folder:** Stores all outputs produced by the project, including images, program messages and `.err` and `.out` files saved by PBS after script execution.
4) **Dataset folder:** Contains all images used to test the program. Further details on this folder are provided in Section IV.
5) **Performance folder:** Holds .csv files documenting all parameters and performance measurements of each program execution.
6) **Docs folder:** Contains various theoretical materials related to the project.

IV. Dataset

The dataset for this project initially consisted solely of natural environment images. However, to robustly test the program under controlled conditions a Python script was developed to generate synthetic image samples. This generator allows to specify the number of lines, the size of the images, the length of the lines and the number of samples to generate.

A. Synthetic Image Generation

All synthetic samples are stored in a designated folder within the `dataset` project directory. For each image or batch of images in a specific folder, the generator creates or updates a corresponding .csv file with ground truth data for images segments. Thus, the file contains a row for each line in every image within that specific folder. Each row details:

- The image to which the line belongs.
- The start and end points of each line segment.
- The coordinates of the projection of each segment to the image borders.
- The ρ and θ parameters for both the segment and its projected line.

These parameters are important for evaluating the algorithm's performance, as further discussed in Section **??**.

B. Synthetic Images for Testing

The synthetic images used to assess the program's performance represent increasing levels of complexity in terms of the number of lines to detect and the image size to process. Table II summarizes the characteristics of these synthetic images and Figure 5 shows sample `syn_img_5k`.

Sample	Resolution	Lines #	Lines Length
syn_img_5k	5000x5000	100	400
syn_img_10k	10000x10000	200	800
syn_img_20k	20000x20000	300	1200

TABLE II
PREDEFINED SYNTHETIC IMAGES USED FOR TESTING DIFFERENT VERSIONS OF THE HT. RESOLUTION AND LINES LENGTH ARE EXPRESSED IN PIXELS.

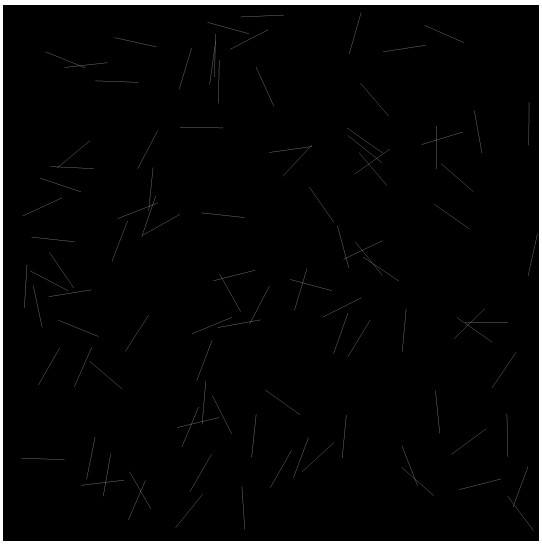


Fig. 5. Sample image `syn_img_5k`. The image was altered to make the segments more evident.

C. Random Samples

The dataset also contains random samples, created and downloaded from Google Earth, for which ground truth data are not available. In these cases, the results can be visually inspected in the `output` folder, where the original images are saved with detected lines superimposed on them. This method allows for a qualitative assessment of the HT implementations effectiveness in real-world scenarios, but is not suitable for structured testing of the program's performance.

V. Image Pre-processing

Starting with this section the report will describe in details each part of my implementation, following the program workflow presented in Section III.

The preprocessing steps available to users are selected through the parameters file to manipulate the image appropriately before line detection. The first and mandatory step

is converting the image to grayscale, which simplifies the image data to a single luminance channel, making subsequent processing steps computationally less intensive.

Additional optional image preprocessing steps include:

- **Gaussian Blurring:** this step blurs the image to reduce noise using a square kernel whose size and σ are controlled by parameters. This is particularly useful for real-world images to enhance edge detection efficacy.
- **Histogram Equalization:** this step adjusts the image's contrast, improving the visibility of features across varied lighting conditions.
- **Sobel Edge Detection:** This step, applied before executing the HT, especially the PPHT, uses the Sobel operator to highlight edges in the image, emphasizing areas with high spatial frequency which correspond to edges.
- **Binary Conversion:** The `toBinary` function is used post-edge detection to convert all pixel values greater than zero to 1, effectively segmenting the image into binary edge and non-edge regions.

A. Parallelization

The Gaussian blurring and Sobel edge detection steps, being computationally intensive, have parallel implementations using OMP. This choice avoids the overhead associated with managing data sharing constraints that would arise with MPI.

Details on Parallelizing Gaussian Blurring: The parallel version of Gaussian blurring uses OMP to divide the image into blocks, with each block being processed by a separate thread. This division is particularly beneficial when dealing with large images, as it allows the workload to be distributed across multiple cores. Here's how the parallel implementation works:

- **Kernel Sharing:** All threads share a single Gaussian kernel, reducing memory overhead.
- **Image Partitioning:** The image is divided into horizontal strips, with each thread processing one strip.
- **Boundary Handling:** Special care is taken at the boundaries of each strip to ensure that the kernel application does not read uninitialized memory. This is managed by extending the region each thread processes at the borders by the radius of the kernel, though only the central part of the processed region is written back to the output image.

B. Details on Parallelizing Sobel Edge Detection

Sobel edge detection, in its sequential form, involves iterating over each pixel (excluding the image border) and applying the Sobel operator to calculate the gradient magnitude. This operation is computational intensive due to the need to access multiple neighboring pixels for each calculation. When parallelized each thread computes the edge detection for a part of the image, similar to the Gaussian blur, After processing, threads synchronize to merge their results into the final output image.

Tests for these two steps were performed with an image downloaded from Google Earth, with a resolution of 8192x4576 pixel, shown in Figure 6. The timing of the execution are summarized in Table III. Both Gaussian blurring and Sobel edge detection benefit significantly from parallelization, particularly in terms of reduced processing time.

Cores	1 (Seq.)	2	4	8	16
Time taken	52.48s	28.07s	14.93s	9.10s	5.6s

TABLE III
PREPROCESSING TIMES AVERAGED ON TWO IDENTICAL RUN OF THE PROGRAM. TEST MADE ARE: SEQUENTIAL WITH 1 CORE AND THEN PARALLEL WITH 2,4 AND 8 CORES AVAILABLE FOR OMP. TIMES ARE TAKEN FOR THE PREPROCESSING OF IMAGE 6.

Fig. 6. Sample image used to evaluate the Gaussin Blurring and SED parallelization.

VI. IMPLEMENTATION OF THE HT AND PHT

Both the HT and the PHT are implemented through a unified function, `HT_PHT`. This function allows for flexible operation based on specified parameters and encapsulates the process of transforming image space into Hough space, where line detection is achieved by identifying peaks.

A. Function Overview

The function `HT_PHT` initializes by parsing the following parameters from the configuration file: the operational mode (HT or PHT), theta resolution, sampling rate for PHT and the vote threshold necessary for line detection.

B. Core Processing

Afterthat, the core processing involves the following key steps:

1) **Point Collection:** The function iterates over image pixels to collect edge points, highlighted using SED method. In PHT mode, points are probabilistically sampled based on the specified rate.
2) **Cosine and Sine Precomputation:** Cosine and sine values for each theta value are precomputed to enhance efficiency.
3) **Voting Mechanism:** Each selected point casts votes in Hough space for every possible theta angle, transforming its (x, y) coordinates into (ρ, θ) parameters, using the equation 1.
4) **Accumulator Array:** This array contains votes across discretized (ρ, θ) values, with each point potentially contributing to multiple line detections.

In all implementations of the HT, operations such as **Point Collection** and **Cosine and Sine Precomputation** are performed at the beginning. This sequence facilitates parallelization by isolating the edge points before the voting process. By exclusively distributing edge points, the approach facilitate data transfer and computational load distribution across processes.

C. Line Detection

Upon populating the accumulator, the next phase of the function identifies entries exceeding the vote threshold as potential lines:

1) Each valid entry's ρ and θ indices are translated into geometric parameters.
2) Line endpoints are calculated and mapped from parameter space back to image coordinates.
3) Detected lines are stored as segments, defined by their endpoints along the image borders, and also the original (ρ, θ) parameters are saved.

These line projections to image borders will be compared against ground truth data, detailed further in Section XII.

VII. Parallelization of the Hough Transform

This section outlines the parallelization strategies used in implementing the HT using MPI, OMP and an hybrid approach, highlighting key aspects of data handling and synchronization.

A. MPI Parallelization

1) Broadcasting Data: The MPI approach (and the hybrid approach) begins with the sharing of parameters. Since the parameter file is specified with a path, process 0 reads it from the file system and then broadcasts the content. This initial step is performed in `main.cpp`, while all subsequent communications are handled within the HT functions.

Following parameter sharing, all processors must align on the image dimensions to ensure that calculations related to pixel positions and transformations into Hough space are consistent. This is achieved using `MPI_Bcast` to share the image width and height among all processes.

2) Scattering Edge Points: The next step is the image's edge pixels distribution among processors. Initially, the image was split into equal stripes, shared among processes. But this approach presents the problem of the potential uneven distribution of workload, as some image parts might have significantly more edge points than others, potentially leaving some processors idle. Instead, the adopted solution involves sampling the edge points only on process 0, applying probabilistic sampling if necessary. Subsequently, from process 0, these edge points are distributed among all processes to guarantee a more balanced workload. `MPI_Scatterv` is used for this purpose, as it allows the distribution of unequal chunks of data, its usage is detailed in Algorithm 1.

Algorithm 1 Distribute edge points using MPI_Scatterv

Parameters:
    counts - array containing the number of elements to send to each process
    displs - array specifying the displacement relative to edgePoints from which to take the outgoing data
    MPI_INT - datatype of the elements in edgePoints
    rank - rank of the receiving process
    MPI_COMM_WORLD - communicator
Output: Local edge points.
procedure MPI_SCATTERV
    MPI_Scatterv(DATA(edgePoints), DATA(counts), DATA(displs), MPI_INT, DATA(localEdgePoints), counts[rank], MPI_INT, 0, MPI_COMM_WORLD)
end procedure

3) Reducing Results: Following the scattering of edge points, each processor continues its execution by computing a local accumulator based on its assigned points. The subsequent step is to integrate local accumulators into a global accumulator that represents the global result. This is accomplished using the `MPI_Reduce` function consolidating all the local accumulators into a single global accumulator at process 0. The details of this MPI operation are specified in Algorithm 2.

To streamline communication, accumulators are flattened into a single-dimensional array during the HT in all MPI implementations. Once the global accumulator is assembled on process 0, for clarity and ease of segment extraction, it is converted back to a 2D vector.

Algorithm 2 Reduce local accumulators to a global accumulator using MPI_Reduce

parameters:
    localAccumulator.size() – number of elements in the local accumulator
    MPI_SUM – operation to be applied (sum)
    0 – root process
Output: Flatten global accumulator
procedure MPI_REDUCE
    MPI_Reduce(DATA(localAccumulator), DATA(globalAccumulatorFlatten), SIZE(localAccumulator), MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD)
end procedure

4) Segment Detection on the Root Process: The final step of segment detection is performed only on process 0. Alternative method involving `MPI_AllReduce` to distribute global accumulator and `MPI_Gather` to have all segments on process 0 were tested to allow each process to participate in segment detection by distributing the rho ranges. However, due to the overhead associated with these additional synchronization steps, this method was deemed less efficient. Consequently,

segment detection is mantained centralized to minimize complexity and synchronization overhead.

B. OMP Parallelization

OMP is employed to leverage thread level parallelism with a shared memory, thus the bottleneck of sharing image data among processes doesn't exists. Instead the challenges in this kind of parallelization reside more on managing correctly the access to shared resources, such as the accumulator.

1) Parallel for loop for detecting edge point: The initial stage in the OMP implementation involves the parallel computation of sine and cosine values for each angle θ and this is achieved using a #pragma parallel for loop. Following this, the edge points detection and their probabilistic sampling (if required) are also parallelized. The details of this implementation are outlined in Algorithm 3, which shows how each thread independently processes the image to identify edge points in parallel.

Algorithm 3 Parallel point collection in an image

parameters:
    Image width and height.
    Total points in the image.
Output: Edge points.
procedure PARALLEL POINT COLLECTION
    #pragma omp parallel num_threads(numThreads)
    #pragma omp for collapse(2) reduction(+:totalPoints)
    for $y = 0$ to $height - 1$ **do**
        for $x = 0$ to $width - 1$ **do**
            // Point collection logic
            #pragma omp critical
            // Store point
        end for
    end for
end procedure

The first line of Algorithm 3 initialize a parallel region that will be executed by `numThreads` threads simultaneously. Then, the #pragma omp for directive tells OMP to distribute iterations of the following loops across the threads and the collapse(2) clause combines the two nested loops into a single loop. The reduction(+:totalPoints) clause is used for thread-safe accumulation of the totalPoints variable. The internal omp critical directive ensures that the code block that modifies the shared container of points is executed only by one thread at a time.

2) Accumulating Votes: The computation of the accumulator in the OMP parallelization approach involves managing concurrent access to the accumulator. To handle this efficiently, the algorithm ensures that multiple threads can update the accumulator without interfering with each other. This process is facilitated by using atomic operations, as detailed in Algorithm 4.

In Algorithm 4, each thread processes a subset of the points array independently. For each point, the perpendicular distance (rho) from the origin to the line, corresponding to the angle defined by $thetaIndex$, is calculated using equation 1. This rho value is adjusted to fit within the bounds of the accumulator array by adding $rhoMax$ and if it fits, the corresponding cell in the accumulator array is incremented. The use of the `#pragma omp atomic` directive ensures that this increment operation is atomic, preventing race conditions that could arise when multiple threads attempt to update the same cell simultaneously.

Algorithm 4 Incrementing an Accumulator Array in Parallel

parameters:
    Edge points.
    Theta resolution.
Output: Edge points.
procedure UPDATE ACCUMULATOR
    #pragma omp parallel for num_threads(numThreads)
    for $i = 0$ to totalEdgePoints $- 1$ **do**
        Calculate $rhoIndex$ and $thetaIndex$ based on point coordinates
        #pragma omp atomic
        accumulator$[rhoIndex][thetaIndex] += 1$
    end for
end procedure

3) Segment Extraction: The final phase of the HT involves identifying line segments using the accumulator. The parallelization strategy for this phase is outlined in Algorithm 5.

Algorithm 5 Parallel Segment Identification

parameters:
    Vote threshold as segment detection criteria.
    Size of rho and theta resolution.
Output: Detected segments.
procedure IDENTIFY SEGMENTS
    #pragma omp parallel for collapse(2) num_threads(numThreads)
    for $rhoIndex = 0$ to $rhoSize - 1$ **do**
        for $thetaIndex = 0$ to $thetaResolution - 1$ **do**
            Calculate segment details based on $rhoIndex$ and $thetaIndex$
            #pragma omp critical
            Store detected segment if it meets criteria
        end for
    end for
end procedure

This approach leverages the `#pragma omp parallel for collapse(2)` directive to flatten and distribute the nested loops over the ρ and θ dimensions of the accumulator array across the available threads. Each thread independently assesses potential lines at each accumulator cell ($rhoIndex$, $thetaIndex$). The use of the `#pragma omp critical` directive is crucial in this phase to ensure that the operation of adding detected segments to the shared list of segments is thread-safe.

C. Hybrid Parallelization

The hybrid approach for parallelizing the HT employs both MPI and OMP to leverage respective strengths in distributed and shared memory approaches.

1) Integration of MPI and OMP: MPI is utilized for distributing the workload across multiple processes, ensuring that each process handles a part of the edge points of the image. Each MPI process then executes multiple threads to perform tasks that in MPI approach described in Section VII-A were performed locally by each process. Key steps in this hybrid approach include:

- **Data Distribution:** The master process (rank 0) reads the entire image and distributes subset of the edge points of the image to other processes using `MPI_Scatterv`.
- **Parallel Precomputation:** Within each process, precomputation of sine and cosine values is done in parallel using `#pragma omp parallel for`.
- **Concurrent Accumulation:** The accumulation phase, where each point in the edge-detected subset votes in the Hough space, is parallelized using OMP to handle concurrent updates efficiently to the local accumulator arrays.
- **Global Reduction and Segment Detection:** After local processing, MPI is used to reduce data from all processes to a single global accumulator using `MPI_Reduce`. Segment detection, particularly intensive in computational terms, is then performed in parallel using OMP on process, enhancing the speed of this bottleneck phase .

This hybrid model combines the data distribution capabilities of MPI with the thread level parallelism of OMP, optimizing the computation within each process of the approach descibed in VII-A.

VIII. Implementation of the PPHT

The PPHT enhances the quality of line detection by focusing on the detection of finite segments. This method is detailed in the work by Matas et al. [3], which introduces modifications to the computational process and voting mechanism.

A. Core Processing

The core processing of the PPHT implementation follows the methods outlined for the HT and PHT in Section VI, with a key difference in the integration of accumulator updates and segment extraction. After edge points are sampled probabilistically, each point is processed and if an update in the accumulator surpasses the vote threshold, 3 additional steps are triggered:

- **Line Points Collection:** This step focuses on identifying accumulations in the Hough space that indicate potential lines, then collects the contributing image points. These points are analyzed for their connectivity; specifically the distance threshold is used to determine the allowable distance between the ρ values of consecutive points in a segment. This distance threshold helps in defining the closeness of points that can be considered part of the same line segment, ensuring that only points closely aligned in Hough space contribute to the same segment.
- **Segment Validation:** Validates the detected segment using the minimum line length and acceptable gap tolerance parameters. This ensures that only significant line segments are recognized as valid.
- **Unvoting Mechanism:** Post validation, the algorithm decrease votes from the contributing points in the accumulator. This refines the space for subsequent detections and reduce the influence of these points the detection of other segments.

An aspect of the PPHT involves dynamically adjusting the voting and distance thresholds, as proposed in [3]. Despite this, the original publication miss explicit instructions on how adjusting these thresholds dynamically. Attempts to adjust these thresholds based on environmental noise were explored but did not enhance performance beyond what was achievable with fixed thresholds. Consequently, the vote threshold remains static, determined by the parameters, and the distance threshold is set at 1. For transparency, these experimental modifications as the functions defined are noted within the code comments.

IX. Parallelization of the PPHT

This section outlines the strategies employed for parallelizing the PPHT using OMP or MPI.

A. Parallelization with MPI

In the MPI-based approach, tasks are distributed across multiple nodes, enabling simultaneous processing of different segments of the image. Key aspects of the MPI implementation include:

- **Image Data Broadcasting:** Initially, the master process broadcasts the image data, including dimensions and pixel values, to all nodes using `MPI_Bcast`. This ensures that each process works with a consistent view of the input image, necessary for line point collection phase described in VIII-A.
- **Edge Point Distribution:** Each process independently calculates potential edge points of an image part, accordingly to its rank. This approach reduces the computational load on each process by limiting it to a subset of the total edge pixels.
- **Local Accumulation and Global Reduction:** Each node computes its local accumulator based on the detected edge points and simultaneously detects segments. These local accumulators are then aggregated into a global accumulator using `MPI_Reduce`, and segments detected by each process are gathered at the master node.

A summary of the execution flow is presented in Algorithm 6.

Although the final reduction of accumulators at line 9 of Algorithm 6 could potentially be avoided since segments are constructed locally at each accumulator, this step is retained for output consistency and coherence with other HT implementations.

Algorithm 6 pseudocode for PPHT using MPI

parameters:
    Image and parameters.
Output: Accumulator and segments for the image.
Procedure: PPHT_MPI
MPI_Bcast for image dimensions and pixel data
Compute edge points based on $rank$.
for each edge point **do**
    Vote in local Hough space and record local segments
end for
MPI_Reduce to reduce local accumulators to a global one at root
MPI_Gatherv to collect all segments at the master node

B. Parallelization with OMP

The PPHT has been adapted to leverage OMP for shared memory parallel processing. edge points sampling is performed as detailed in Algorithm 3. The procedure thus can be summarized as presented in Algorithm 0.

Algorithm 7 Updating Accumulator and Detecting Segments

parameters:
    Image points and edge points.
    Parameters.
Output: Accumulator and segments for the image.
procedure UPDATEACCUMULATORANDDETECTSEGMENTS
    #pragma omp parallel
    vector[bool] localProcessed(points.size(), false)
    #pragma omp for
    for $i \leftarrow 0$ **to** edgePoints.size() **do**
        #pragma omp atomic
        Update accumulator
        Segment detection logic
        #pragma omp critical
        Add segment to shared container
    end for
end procedure

C. Accumulator Updates and Line Segment Detection

Each thread contributes to updating a shared accumulator and simultaneously checks for line segments exceeding the vote threshold. This involves atomic operations for updating the accumulator and critical sections for adding detected segments safely.

X. Clustering

Clustering operation is a post processing step to enhance the quality of the line detection results of the HT and PHT versions. This operation effectively reduce noise impacts by grouping similar lines based on predefined thresholds for ρ and θ values. The clustered segments are then merged into a single line segment, calculated as the weighted average based on votes of the constituent segments. This average accounts for the line's orientation and distance from the origin, encapsulated by the θ and ρ parameters, respectively.

Clustering uses two main thresholds: the maximum angular difference allowed between segments within the same cluster (theta threshold) and the maximum radial distance permitted between segments within a cluster (rho threshold).

While effective for standard HT and PHT outputs, for PPHT the direct application of this clustering method was not as beneficial. Consequently, the clustering was disabled.

XI. Evaluation Methodology

The quality of the detection for each HT versions is evaluated in terms of execution time, precision and recall. The methodology employed for obtaining precision and recall metrics is inspired by the PPHT evaluation presented by Matas et al. in [3]. The evaluation process involves calculating the overlap between detected and ground truth segments under the assumption that they are collinear. This yields the following statistics:

- **True Positives (TP):** Segments that are correctly identified as part of the ground truth.
- **False Positives (FP):** Segments identified that do not correspond to any ground truth segment.
- **False Negatives (FN):** Ground truth segments that were not detected.

Precision and recall are derived as follows:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}},$$
$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}.$$

A segment is classified as a true positive if the overlap with a ground truth segment exceeds a threshold, set to 80% of the segment's length. Furthermore, timing of the preprocessing phase, HT execution and overall program execution are recorded, along with the corresponding parameters for each program run.

For a more detailed and dynamic inspection of the results achived is available the python notebook `analysis.ipynb` in performance project folder.

XII. Performance and Scalability Analysis - Setup

To evaluate the performance of different HT implementations, three synthetic images, as detailed in Section IV-B, were used along with their corresponding ground truth data. The performance was assessed through 1188 runs of the program, resulting in 594 unique tests with each test configuration run twice. Each combination of image and HT version was examined across various PBS resource allocations, a summary of the tests performed is presented in Table IV.

The results of this analysis are presented through various charts in Section XIII. The PBS resources of each configuration were used as follows: For MPI configurations, the `-np` parameter of `mpiexec` instruction was equal to the number of PBS `ncpus` multiplied by `select`. For OMP configurations,

TABLE IV
COUNT AND TIME OF TESTS PERFORMED FOR EACH SYNTHETIC IMAGE SAMPLE.

Image	Total Tests	Total Time
syn_img_5k	198	54m
syn_img_10k	198	154m
syn_img_20k	198	494m

`-np` was set to 1, and the number of OMP threads was set to `ncpus` multiplied by `select`. In the hybrid approach, `-np` was equal to `select` and the number of OMP threads was set to `ncpus`.

The charts in Section XIII include:

- **Heatmaps of Efficiency:** Each heatmap corresponds to an HT version applied to a specific synthetic image. Rows represent combinations of parallelization strategies and PBS place parameter values, while columns represent different PBS resource configurations (`select` and `ncpus`).
- **Line Charts of Speedup (with Precision and Recall):** These charts display speedup, precision and recall metrics for each HT version executed on a synthetic sample under a specific parallelization strategy. The precision and recall metrics are shown in each chart's title and the line chart itself rapresents the speedup variations according to the number of cores utilized.

XIII. PERFORMANCE AND SCALABILITY ANALYSIS - DISCUSSION

The project was devoloped with the aim to encapsulate in a controlled and optimized parallel environment different HT versions. The modularization adopted enable to test different kind of HT, each rapresentend within a single function and the parallelization was kept as a core concept for the entire devolopment. This approach led to an inherently hybrid program, combining OMP for preprocessing tasks with any kind of parallelization of the HT algorithm after.

- **Sequential Baseline:** Development started with sequential implementations of HT versions as baselines in order to familiarize with data structures and the transformation itself. These initial implementations were tested in a non parallel environment with PBS settings of `select=1` and `ncpus=1`. The baseline speedup is not depicted in the line charts of Section XIII; however, the efficiencies for these tests are displayed in the first column of the matrices.
- **Parallel Implementations:** After establishing the baselines, specialized functions for MPI, OMP and hybrid strategies were developed and tested.

At the end the testing phase analyzing the performance obtained in relation to the devolopment of the project, the following considerations can be made.

- **Performance Hierarchy:** The OMP implementations consistently outperformed other strategies in terms of speedup and efficiency. This was followed by the hybrid implementations, and then the MPI versions. Notably, MPI often underperformed compared to the sequential baseline. This underperformance could be attributed to the overhead associated with distributed memory management in MPI, which can become significant when the data sharing and synchronization costs outweigh the computational benefits. In the case of my implementation of the HT, which involves a significant amount of data interaction and dependency, the distributed approach of MPI might not be the most efficient unless carefully optimized.
- **Configuration Impact:** Different configurations of the PBS system, particularly the `scatter:excl` and pack:excl place parameters, showed variable impacts on performance. Distributed memory implementations, in general, benefited more from the `scatter:excl` configuration, enhancing node level parallelism, while the pack:excl configuration favored shared memory approaches like OMP, as detailed in Section XIII.
- **Speedup Analysis:** The line charts depicting speedup illustrate distinct performance levels for each parallelization strategy. The OMP configuration achieved a maximum speedup of 6.65, with a PHT applied to the syn_img_20k sample. This test utilized 4 PBS `select` nodes and 4 `ncpus`, thus a total of 16 OMP threads and a single MPI process (`-np=1`). The hybrid approach, combining MPI and OMP, demonstrated low moderate speedup in the `scatter:excl` configuration, peaking at 2.17. This was observed with an HT on `syn_img_10k`, utilizing 4 `select`and 8 `ncpus`, thus configuring 4 MPI processes and 8 OMP threads per process. As anticipated in **configuration impact**, the `pack:excl` configurations within Hybrid approaches, which consolidates processes closer together, yielded lower speedups with identical resource allocations if compared to `scatter:excl`, maybe due to increased contention and communication overhead among the MPI processes. The MPI-based implementations, instead, showed the worst results, with a maximum speedup of only 0.25. This occurred during an HT application on `syn_img_20k`, configured with a single `select` node and two `ncpus`, confirming that the parallelization strategy adopted doesn't scale properly with the resources and this can be attributed to inefficient use of distributed resources and significant communication overhead.
- **MPI Challenges:** The MPI versions faced several bottlenecks, notably in the data distribution phase and during the reduction of local accumulators. The necessity for process 0 to manage the entire accumulator and segment detection enhance a lot these issues, leading to substantial overhead and inefficiency. This inefficacy was particularly pronounced in the PPHT implementation, where the sharing of all the image data across processes further increase the communication overhead. These factors collectively lead to MPI implmentations often underperforming compared to both the sequential baseline and more so against

the shared memory implementations.

- **Hybrid Advantages:** Incorporating OMP within each MPI process mitigated some of the inefficiencies by reducing the impact of bottlenecks in data handling and expecially in segment detection phase executed only on process 0. This approach slightly improved the results compared to pure MPI implementations.
- **Precision and Recall:** Across most implementations, precision and recall metrics were satisfactory, with the notable exception of PPHT. The lack of clustering in PPHT led to a proliferation of overlapping line segments, which, while achieving high precision, adversely affected recall. This issue underscores the need for parameter fine tuning (PPHT line gap and line length used for segment validation) and effective implementations of the dynamic threshold adjustments that are properties of the PPHT but are not available, as explained in Section VIII. An example of result with a PPHT applied on image presented in Section V Figure 6 is shown here in Figure 7 and in subsequent Figures 8 and 9 are shown a random sample with two figures and the detections with a PHT, respectively.

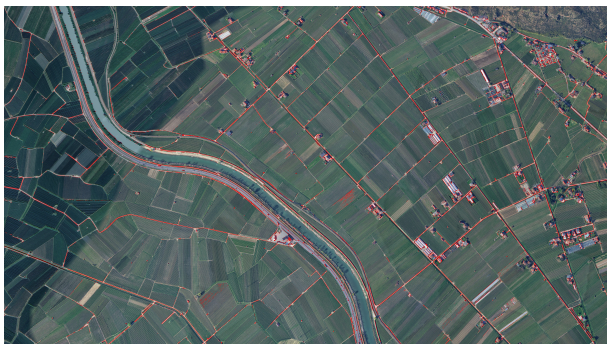


Fig. 7. Output of the PPHT applied on image shown in Figure 7.

To summarize, this analysis demonstrates that OMP implementations consistently delivered higher performance due to their ability to efficiently manage shared resources and minimize communication overhead, particularly compared to the MPI implementations. This underscores the critical importance of aligning and optimizing the chosen parallelization strategy with the specific computational characteristics of the algorithms to maximize efficiency.

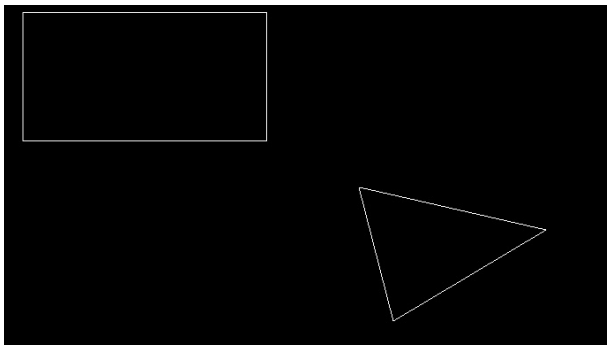


Fig. 8. Random input sample with shapes to the PHT.

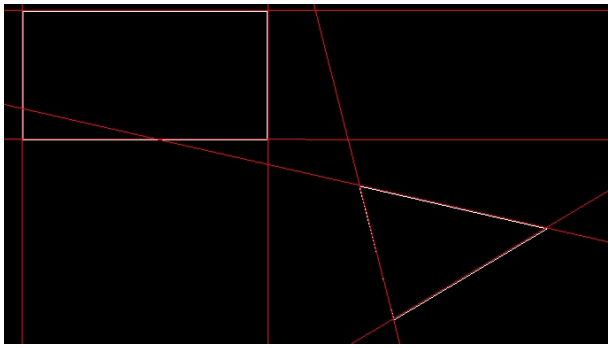


Fig. 9. Output of the PHT applied to Figure 8.

REFERENCES

[1] R. O. Duda and P. E. Hart, "Use of the hough transformation to detect lines and curves in pictures," *Commun. ACM*, vol. 15, no. 1, p. 11–15, jan 1972. [Online]. Available: https://doi.org/10.1145/361237.361242

[2] N. Kiryati, Y. Eldar, and A. Bruckstein, "A probabilistic hough transform," *Pattern Recognition*, vol. 24, no. 4, pp. 303–316, 1991. [Online]. Available: https://www.sciencedirect.com/science/article/pii/003132039190073E

[3] J. Matas and C. Galambos, "Progressive probabilistic hough transform," 08 1998.

[4] Wikipedia contributors, "Hough transform — Wikipedia, the free encyclopedia," 2024, [Online; accessed 8-July-2024]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Hough_transform&oldid=1226870376

APPENDIX: PERFORMANCE CHARTS

Efficiency matrices for each combination of image, HT version and parallelism and PBS properties.

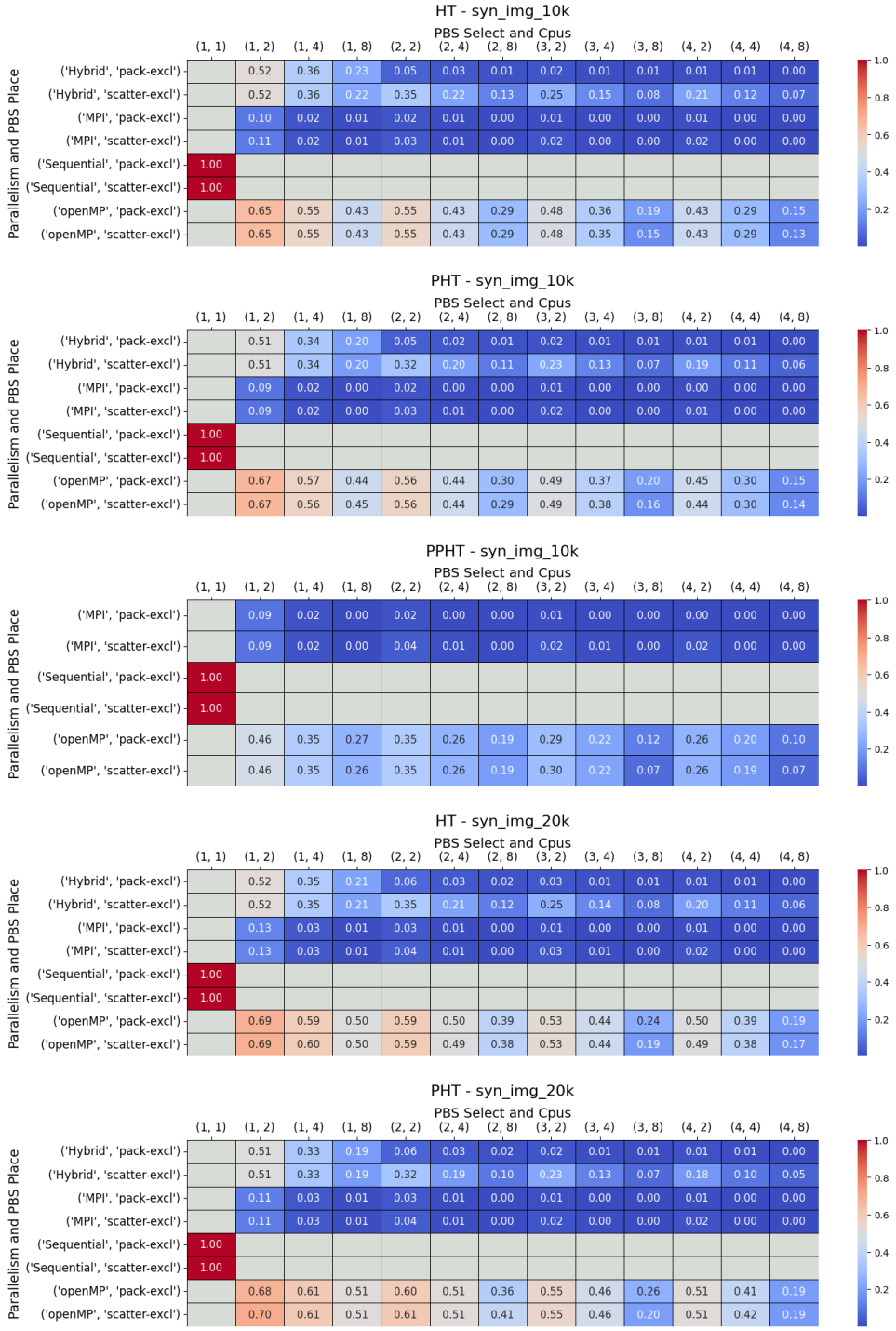
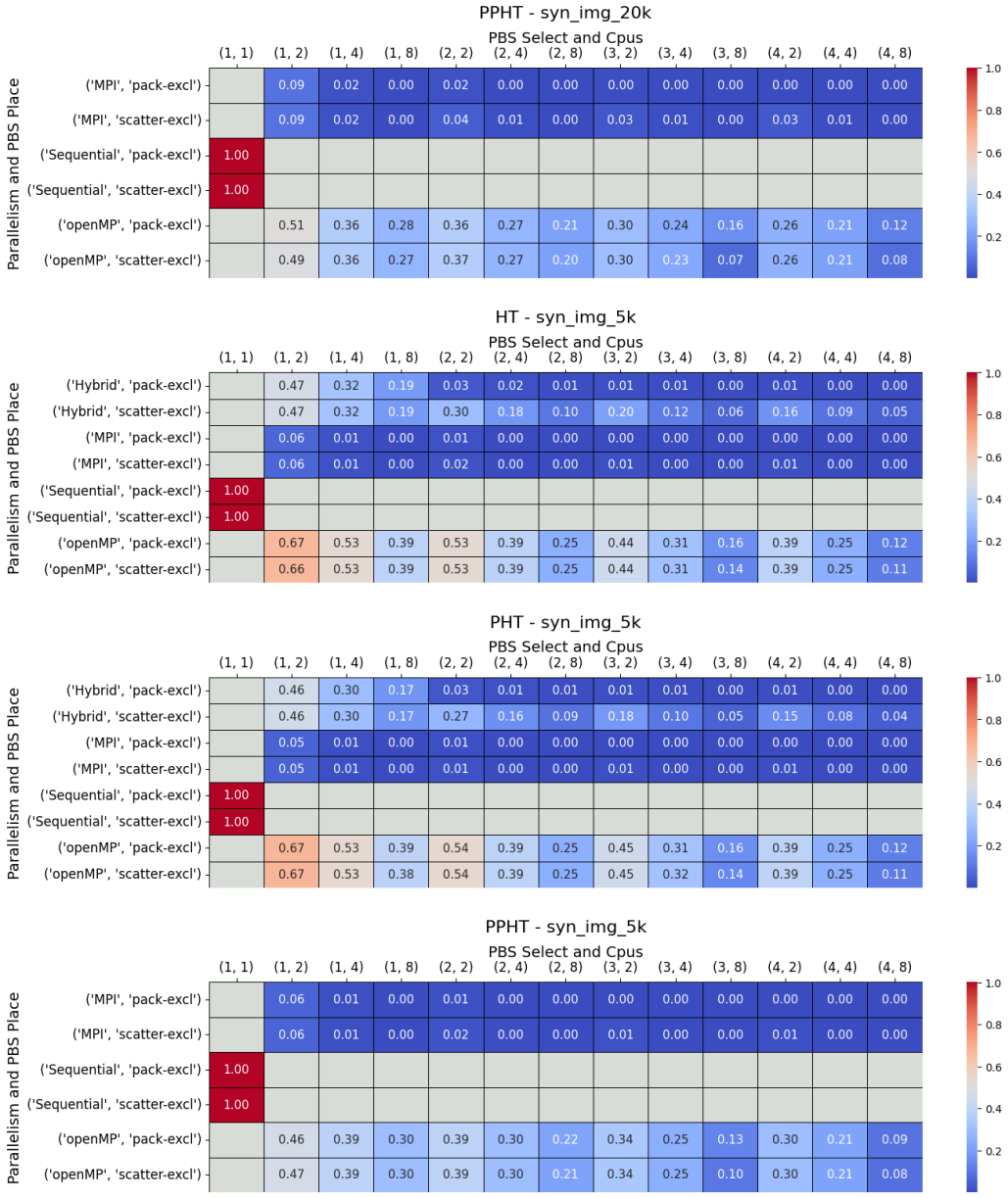


Fig. 10. Efficiency matrices for each combination of image, HT version and parallelism and PBS properties.



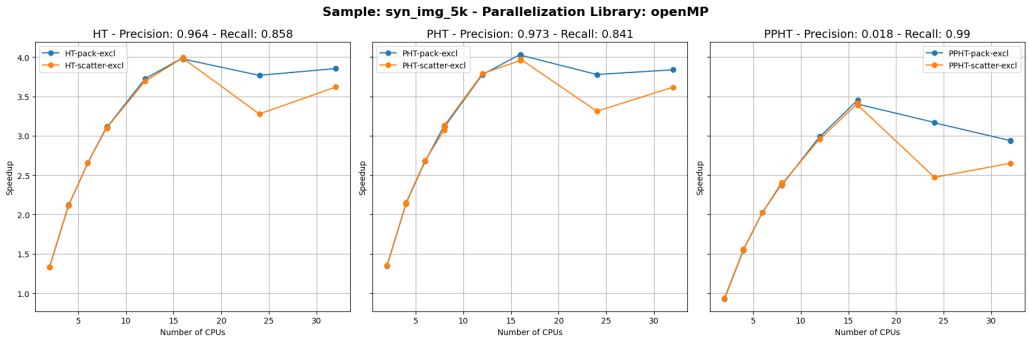


Fig. 11. Speedup line charts for OMP and MPI.

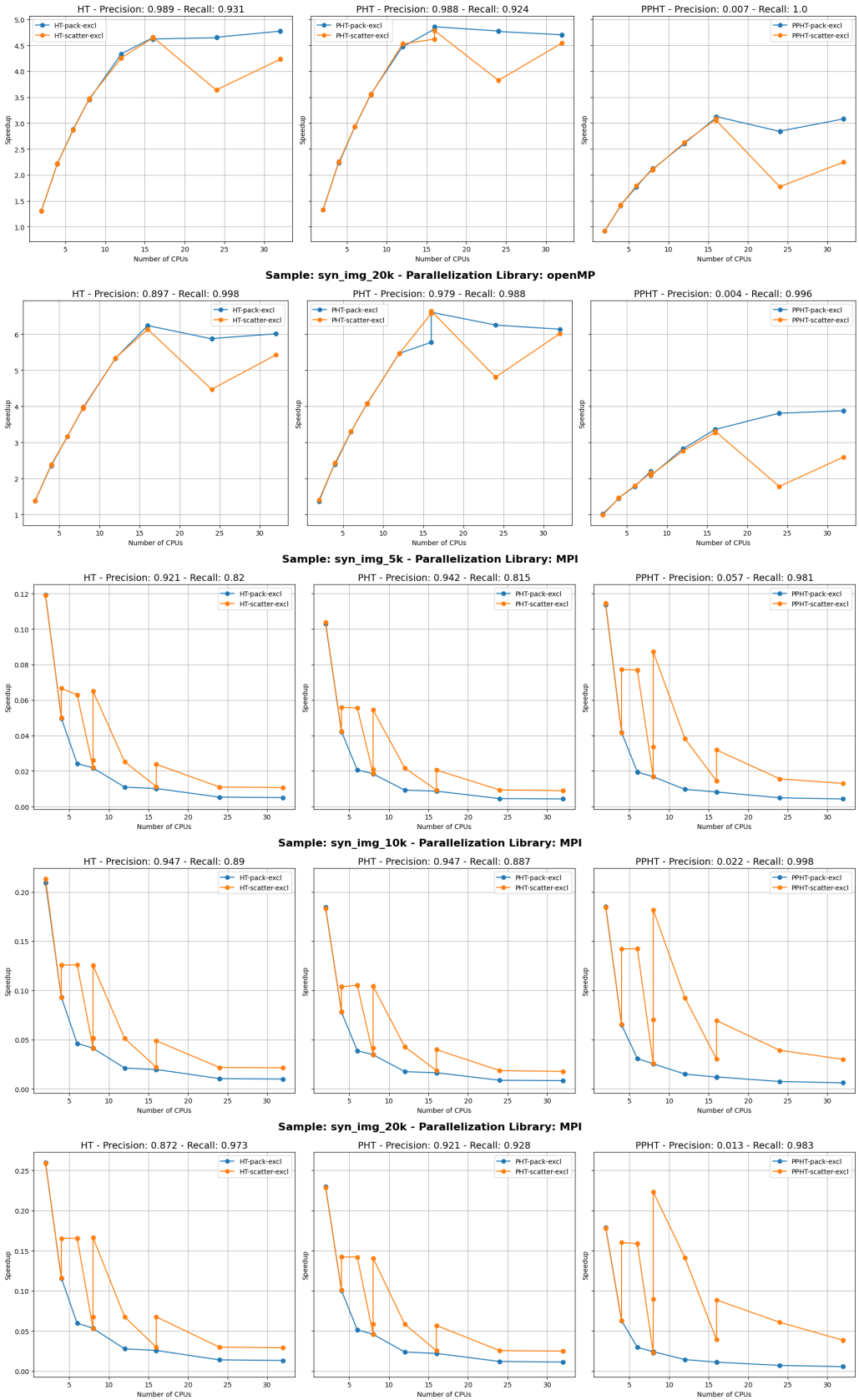


Fig. 12. Speedup line charts Hybrid parallelism.