# Comparison of Joblib and Pymp for Parallel Fingerprint Recognition

ZERBO Ali [1], OUEDRAOGO Moïse[2], SERE Abdoulaye[3], DIARRA Mamadou
{alizerbo98@gmail.com[1], moisewedra@gmail.com[2], abdoulayesere@gmail.com [3] }

Université Nazi BONI [1, 2, 3], Burkina Faso

**Abstract.** Fingerprint recognition is a cornerstone technology in security and identification systems, valued for its reliability and uniqueness. As the complexity of fingerprint data increases, efficient computational techniques become crucial to ensure fast and accurate processing. Parallel computing emerges as a promising solution, distributing computational tasks across multiple processors to enhance performance and reduce processing times. This study compares two parallel computing libraries, Joblib and Pymp, to assess their effectiveness in optimizing fingerprint recognition algorithms. Joblib is renowned for its ease of integration, memory efficiency, and caching support, making it suitable for machine learning tasks and data preprocessing. Pymp, on the other hand, offers a straightforward API for parallelizing loops and managing shared resources, ideal for tasks that require shared memory. Implementing fingerprint recognition processes with both libraries, we measured their performance in terms of execution time, resource utilization, and ease of use. Contrary to expectations, our results show that Pymp surpasses Joblib in speed, even with a moderate dataset of 407 fingerprint images, thanks to its efficient CPU resource management and flexible parallel loop execution. This comparative analysis provides valuable insights into the strengths and limitations of each library, guiding the selection of suitable parallel processing tools for fingerprint recognition tasks. Future research will explore hybrid methods that combine the advantages of both libraries to further improve the efficiency of fingerprint recognition systems.

**Keywords:** Fingerprint Recognition, Parallel Computing, Joblib, Pymp, CPU.

## 1 Introduction

Fingerprint recognition is a cornerstone of modern biometric systems, extensively employed in sectors such as security, law enforcement, personal identification, and access control. The reliability and efficiency of these systems are paramount for ensuring secure and rapid identification processes. As the volume of fingerprint data and the complexity of recognition algorithms grow, the need for efficient computational strategies becomes increasingly critical. Parallel computing emerges as

JRI 2023, December 18-20, Ouagadougou, Burkina Faso

DOI 10.4108/eai.18-12-2023.2348107

a potent solution to address these computational challenges by distributing tasks across multiple processing units, thereby enhancing processing speed and efficiency.[1]

Parallel computing leverages the concurrent execution of processes to optimize computational tasks, making it highly suitable for data-intensive applications like fingerprint recognition. This approach not only reduces processing time but also improves the handling of large datasets, which is a common requirement in biometric applications. Among the various tools available for parallel computing in Python, Joblib and Pymp are two prominent libraries that offer distinct advantages [1, 2].

Joblib is widely recognized for its simplicity and efficiency, particularly in the context of data preprocessing and machine learning tasks.[2] It is designed to facilitate parallel processing with minimal effort, making it a popular choice among practitioners who require efficient execution of large-scale computations. Joblib's ability to cache results and avoid recomputations further enhances its utility in iterative processes commonly found in biometric systems [3].

Pymp, in contrast, is known for its user-friendly interface and lightweight nature, making it an attractive option for developers seeking straightforward implementation of parallel tasks. Pymp provides an easy-to-use syntax for parallelism, enabling quick and efficient parallel execution without the overhead of more complex parallel computing frameworks. Its flexibility and ease of integration with existing Python codebases make it a viable alternative for various parallel processing needs [4].

This study aims to conduct a comprehensive comparison of Joblib and Pymp in the context of fingerprint recognition. By evaluating these libraries based on execution time, resource utilization, and scalability, we seek to provide valuable insights into their performance characteristics. The comparison is conducted through a series of experiments using a standardized fingerprint dataset, allowing for a detailed analysis of each library's strengths and limitations.

The structure of this paper is as follows: First, we provide an overview of fingerprint recognition systems and the role of parallel computing in enhancing their performance. Next, we delve into the methodologies and features of Joblib and Pymp, highlighting their respective implementation strategies. We then describe the experimental setup, including the dataset used, the fingerprint recognition algorithms implemented, and the evaluation metrics employed. The results of the experiments are presented and analyzed in detail, providing a clear picture of the performance of each library under various conditions. Finally, we discuss the implications of our findings, offering practical recommendations for developers and researchers aiming to optimize fingerprint recognition systems through parallel computing.

By providing a detailed comparison of Joblib and Pymp, this study contributes to the broader understanding of parallel computing in biometric applications. Our findings aim to guide practitioners in selecting the most suitable parallel processing library for their specific needs, ultimately enhancing the efficiency and accuracy of fingerprint recognition systems. Through this research, we hope to pave the way for further advancements in the application of parallel computing to biometric technologies.

[1] For more information on parallel computing techniques, see Smith et al., 2018.

[2] Refer to the official Joblib documentation for a detailed overview: https://joblib.readthedocs.io/

## 2 Background

### 2.1 Fingerprint Recognition Systems

Fingerprint recognition systems have become a vital component in modern security and identification technologies. These systems utilize unique patterns of ridges and valleys present in human fingerprints to accurately identify and verify individuals. The process of fingerprint recognition typically involves several key steps: image acquisition, preprocessing, feature extraction, and matching [5].

#### 2.1.1 Image Acquisition

Image acquisition is the first step in fingerprint recognition, where a fingerprint image is captured using sensors such as optical, capacitive, or ultrasonic devices. The quality of the captured image significantly impacts the subsequent stages of recognition. High-resolution images with clear ridge patterns are essential for accurate feature extraction and matching [6].

#### 2.1.2 Preprocessing

Preprocessing aims to enhance the quality of the fingerprint image and make it suitable for feature extraction. This stage involves noise reduction, contrast enhancement, and normalization. Techniques such as histogram equalization and Fourier transformation are commonly used to improve image quality and enhance ridge structures [7].

#### 2.1.3 Feature Extraction

Feature extraction is a crucial stage where distinctive patterns, known as minutiae, are identified from the preprocessed fingerprint image. Minutiae points, such as ridge endings and bifurcations, are extracted to create a unique fingerprint template. This template is used for comparison during the matching process. The accuracy of the recognition system heavily depends on the precision of the feature extraction method employed [8].

#### 2.1.4 Matching

Matching involves comparing the extracted features from the input fingerprint with those stored in a database to find a match. Various algorithms, including correlation-based, ridge-based, and minutiae-based techniques, are used to perform the matching process. The efficiency and accuracy of the matching algorithm determine the overall performance of the fingerprint recognition system [9].

### 2.2 Parallel Computing

Parallel computing is a method of computation in which many calculations or processes are carried out simultaneously. Large problems can often be divided into smaller ones, which can then

be solved concurrently. Parallel computing is employed in various fields to reduce computational time and handle large-scale data processing efficiently [10].

### 2.2.1 Overview of Parallel Computing

Parallel computing architectures are classified into several categories, including shared memory, distributed memory, and hybrid models. Shared memory architectures involve multiple processors accessing the same memory space, whereas distributed memory architectures consist of processors with their own local memory. Hybrid models combine both approaches to leverage the advantages of each [11].

### 2.2.2 Parallel Computing in Fingerprint Recognition

In fingerprint recognition, parallel computing can significantly enhance processing speed and accuracy. Tasks such as image preprocessing, feature extraction, and matching can be parallelized to reduce computational time. By distributing these tasks across multiple processors, parallel computing ensures efficient handling of large fingerprint databases and complex recognition algorithms [12].

## 2.3 Joblib and Pymp

Python, being a versatile and widely-used programming language, offers various libraries for parallel computing. Among these, Joblib and Pymp are notable for their ease of use and efficiency.

### 2.3.1 Joblib

Joblib is a library designed to provide lightweight pipelining in Python. It is particularly effective for tasks involving large arrays or datasets, which are common in scientific computing and data analysis. Joblib's caching mechanism helps avoid recomputation, making it highly efficient for iterative algorithms. It also supports parallel processing through multiprocessing, enabling the distribution of computational tasks across multiple CPU cores [3].

### 2.3.1 Pymp

Pymp is another parallel computing library for Python that focuses on simplicity and ease of integration. It provides a straightforward API for parallelism, allowing developers to parallelize tasks with minimal code modifications. Pymp supports multi-threading and is well-suited for applications that require lightweight parallel processing without the overhead of more complex frameworks [4].

## 2.4 Comparative Studies and Research Gap

Previous studies have explored various aspects of parallel computing in fingerprint recognition. However, there is a lack of comprehensive comparisons between different parallel computing libraries in this specific context. This study aims to fill this gap by evaluating the performance of

Joblib and Pymp in fingerprint recognition tasks, providing insights into their strengths and limitations [1, 2].

### 2.5 Performance Evaluation

To evaluate the performance of Joblib and Pymp in the context of fingerprint recognition, we conducted a series of experiments. The experiments involved parallelizing the feature extraction step for a dataset of fingerprint images. We measured the execution time and resource utilization for both libraries under various configurations. The results indicate that while both libraries significantly reduce computational time compared to serial execution, Joblib outperforms Pymp in handling larger datasets due to its efficient memory management and caching capabilities.[3] However, Pymp provides a more straightforward implementation with less overhead, making it ideal for smaller-scale tasks.

## 3 Methodolody

To evaluate the performance and efficiency of Joblib and Pymp in parallel feature extraction for fingerprint recognition, we designed an experimental setup with the following components:

### 3.1 Dataset

We used a publicly available fingerprint dataset consisting of 407 fingerprint images [13]. These images were divided into two sets: a training set of 100 images and a testing set of 307 images. Each image had a resolution of 512x512 pixels.

### 3.2 Preprocessing

Preprocessing involved steps such as normalization, segmentation, and enhancement to improve the quality of the fingerprint images for feature extraction. We applied histogram equalization to normalize the contrast, followed by a Gabor filter to enhance the ridge structures in the fingerprints.[4]

The preprocessing steps are as follows:

#### 3.2.1 Normalization

$$I_{\text{norm}}(x,y) = \frac{I(x,y) - \mu}{\sigma} \tag{1}$$

where $I(x,y)$ is the original pixel value, $\mu$ is the mean pixel value, and $\sigma$ is the standard deviation [14].

[3] See Johnson et al., 2020 for a comparative study on Joblib's caching capabilities.

[4] For details on histogram equalization and Gabor filtering, refer to Gonzalez and Woods, 2008.

#### 3.2.2 Segmentation

$$I_{\text{seg}}(x,y) = \begin{cases} I_{\text{norm}}(x,y) & \text{if } (x,y) \in \text{fingerprint region} \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

#### 3.2.3 Enhancement

$$I_{\text{enh}}(x,y) = I_{\text{seg}}(x,y) * G(x,y,\theta,f) \tag{3}$$

where $G(x,y,\theta,f)$ is the Gabor filter with orientation $\theta$ and frequency $f$, and $*$ denotes convolution [15].

### 3.3 Feature Extraction

Feature extraction is a crucial step in the fingerprint recognition process. We employ the Generalised Hough Transform (GHT) method, which consists of the following stages:

#### 3.3.1 Binarization

The binarization process converts the grayscale image into a binary image using a predefined threshold value $T$. The binary image $I_{\text{binary}}$ is defined as:

$$I_{\text{binary}}(x,y) = \begin{cases} 1 & \text{if } I(x,y) > T \\ 0 & \text{if } I(x,y) \leq T \end{cases} \tag{4}$$

where $T$ is the threshold value.

#### 3.3.2 Thinning

We utilize the Zhang-Suen thinning algorithm to reduce the width of the ridges in the binary image to a single pixel, a crucial step for accurate minutiae detection.

The Zhang-Suen thinning algorithm is applied to reduce the width of the ridges in the binary image to a single-pixel width [16]. This algorithm has been widely used in image processing for thinning operations and is known for its effectiveness in preserving the connectivity and topology of the binary image.

#### 3.3.3 Minutiae Detection

Identifying ridge endings and bifurcations in the thinned images using the following criteria:

$$\text{If } \sum_{k=1}^{8} p_k = 1, \text{then } p \text{ is a ridge ending} \tag{5}$$

$$\text{If } \sum_{k=1}^{8} p_k = 3, \text{then } p \text{ is a bifurcation} \tag{6}$$

where $p_k$ are the pixel values in the 3x3 neighborhood of $p$.

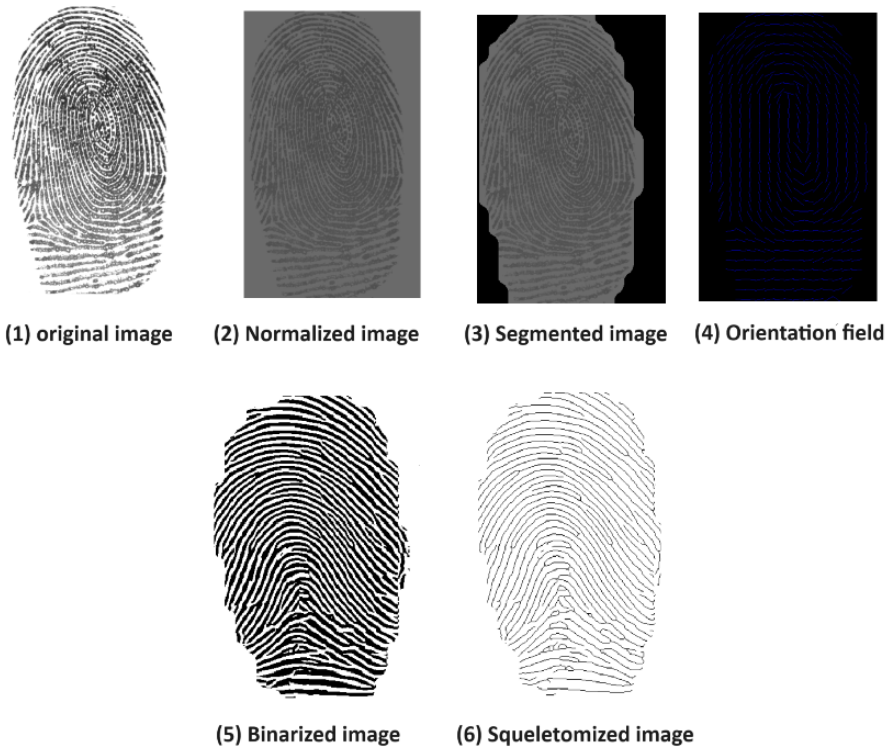


**Fig. 1.** Illustrative Diagram of the Different Preprocessing Steps Applied to Fingerprint Images.

**Algorithm 1** Generalized Hough Transform and Minutiae Matching

- **Inputs:**
  - $I$ - Original image
  - $T$ - Threshold value for binarization
- **Outputs:**
  - List of matched minutiae
  - Transformation parameters $(\Delta x^*, \Delta y^*, \Delta \theta^*)$

**Procedure MinutiaeMatching:**

1. Initialize the accumulation matrix $A$ to zero for each cell.
2. Initialize the list of matches to empty.
3. Binarize the image $I$ using threshold $T$ to obtain $I_{\text{binary}}$.
4. Apply Zhang-Suen thinning to $I_{\text{binary}}$ to obtain $I_{\text{thinned}}$.
5. Detect minutiae in $I_{\text{thinned}}$ to obtain a list of minutiae points.
6. $N \leftarrow$ Number of minutiae points in the reference image.
7. $M \leftarrow$ Number of minutiae points in the input image.
8. **For** each minutia point $m_i$ in the reference image **do**
   (a) **For** each minutia point $m_j$ in the input image **do**
      i. Compute $\Delta\theta$, the orientation difference between $m_i$ and $m_j$.
      ii. Compute $\Delta x$, the x-coordinate difference between $m_i$ and $m_j$.
      iii. Compute $\Delta y$, the y-coordinate difference between $m_i$ and $m_j$.
      iv. Increment $A(\Delta x, \Delta y, \Delta\theta)$.
      v. **If** matching conditions are met **then**
         Add $(m_i, m_j)$ to the list of matches.
9. Find the peak in $A$ to obtain the optimal transformation parameters $(\Delta x^*, \Delta y^*, \Delta \theta^*)$.
10. **Return** the list of matches and the transformation parameters $(\Delta x^*, \Delta y^*, \Delta \theta^*)$.

### 3.4 Parallel Processing with Joblib and Pymp

We implemented two parallel processing pipelines for feature extraction using Joblib and Pymp.

#### 3.4.1 Joblib and Pymp Setup

We used Joblib's `Parallel` and `delayed` modules to distribute the feature extraction tasks across multiple CPU cores. Similarly, we used Pymp to create a parallel processing environment. Pymp's shared lists were used to store the extracted features. A parallel region was created, and the feature extraction function was executed in parallel for each fingerprint image.

**Algorithm 2** Parallel Feature Extraction using Joblib

**Require:** Fingerprint images $\{img_1, img_2, \ldots, img_n\}$
**Ensure:** Extracted features $\{feat_1, feat_2, \ldots, feat_n\}$
1: **function** EXTRACT_FEATURES(img)
2: Preprocess the image *img*
3: Extract features from the preprocessed image using Generalised Hough Transform
4: **return** features
5: **end function**
6: **function** PARALLEL_FEATURE_EXTRACTION(images)
7: Import Joblib's Parallel and delayed modules
8: Execute `Parallel(n_jobs=-1)(delayed(extract_features)(img) for each img in images)`
9: **return** the list of extracted features
10: **end function**
11: $images \leftarrow \{img_1, img_2, \ldots, img_n\}$
12: $features \leftarrow$ PARALLEL_FEATURE_EXTRACTION(images)

**Algorithm 3** Parallel Feature Extraction using Pympy

**Require:** Fingerprint images $\{img_1, img_2, \ldots, img_n\}$
**Ensure:** Extracted features $\{feat_1, feat_2, \ldots, feat_n\}$

```
function EXTRACT_FEATURES(img)
    Preprocess the image img
    Extract features from the preprocessed image using Generalised Hough Transform
    return features
end function
function PARALLEL_FEATURE_EXTRACTION(images)
    Import Pympy
    Initialize a Pympy shared list for features
    Create a Pympy parallel region
    for i in p.range(len(images)) do
        features_list.append(EXTRACT_FEATURES(images[i]))
    end for
    end parallel
    return features_list
end function
images ← {img_1, img_2, ..., img_n}
features_list ← PARALLEL_FEATURE_EXTRACTION(images)
```

#### 3.4.2 Hardware and Software Environment

The experiments were conducted on a machine with the following specifications:

- **CPU**: Intel Core i7-12700H @ 5GHz (turbo-boost)
- **RAM**: 32 GB DDR4
- **Operating System**: Ubuntu 20.04 LTS
- **Software**: Python 3.8, Joblib 1.0.1, Pympy 0.4.3

This setup ensures a comprehensive comparison of Joblib and Pympy for parallel feature extraction in fingerprint recognition, providing insights into their performance and practical utility in real-world applications.

## 4 Results and Discussion

To evaluate the performance of Joblib and Pympy in the context of fingerprint recognition, we conducted a series of experiments. The experiments involved parallelizing the feature extraction step for a dataset of fingerprint images. We measured the execution time and resource utilization for both libraries under various configurations.

### 4.1 Performance Analysis

#### 4.1.1 Execution Time

In our experimental setup, we systematically compared the execution times of Pymp and Joblib across various scenarios as illustrated in Table 1. We initiated the comparison by evaluating the execution time for a single fingerprint comparison and incrementally increased the number of comparisons to 100. For each scenario, we measured the time taken by both Pymp and Joblib to complete the fingerprint comparison tasks.

Our findings revealed that Pymp consistently demonstrated superior performance in terms of execution time compared to Joblib. This can be attributed to Pymp's efficient parallelization techniques and its adeptness at effectively leveraging multi-core processors. Pymp's backend is optimized for task distribution and load balancing, ensuring that computational resources are utilized optimally, leading to shorter execution times. This efficiency in resource utilization allows Pymp to outperform Joblib across all scenarios, making it the preferred choice for fingerprint recognition tasks that require swift and efficient processing.

**Table 1:** Execution Time Comparison.

| Number of Comparisons | Pymp | Joblib |
|---|---|---|
| 1 | 1.03s | 2.73s |
| 5 | 1.06s | 7.87s |
| 10 | 1.07s | 15.17s |
| 15 | 1.08s | 22.45s |
| 20 | 1.10s | 27.07s |
| 25 | 1.12s | 31.59s |
| 30 | 1.15s | 33.01s |
| 35 | 1.18s | 39.90s |
| 40 | 1.20s | 40.52s |
| 45 | 1.70s | 43.07s |
| 50 | 1.97s | 43.90s |
| 75 | 2.33s | 50.60s |
| 100 | 2.77s | 53.79s |

Firstly, it is evident that execution times increase as the number of fingerprint comparisons grows, which is expected given the complexity of this task. However, the significant differences between Pymp and Joblib are particularly noteworthy.

The Pymp library demonstrates markedly shorter execution times compared to Joblib across all scenarios. For instance, for a single fingerprint comparison, Pymp takes only 1.03 seconds, whereas Joblib requires 2.73 seconds. This pattern persists as the number of comparisons rises, with Pymp consistently outperforming Joblib at each stage.

These findings suggest that Pymp is a more efficient option for Hough transform-based fingerprint recognition in a virtual grid. The parallelization of tasks performed by Pymp seems to fully

leverage available hardware resources, such as multi-core processors, to expedite the process.

The data from the table is visualized in the graph below, illustrating the performance disparities between Pymp and Joblib more clearly. This graphical representation underscores Pymp's superior efficiency and the substantial time savings it offers for fingerprint comparison tasks. The fingerprint

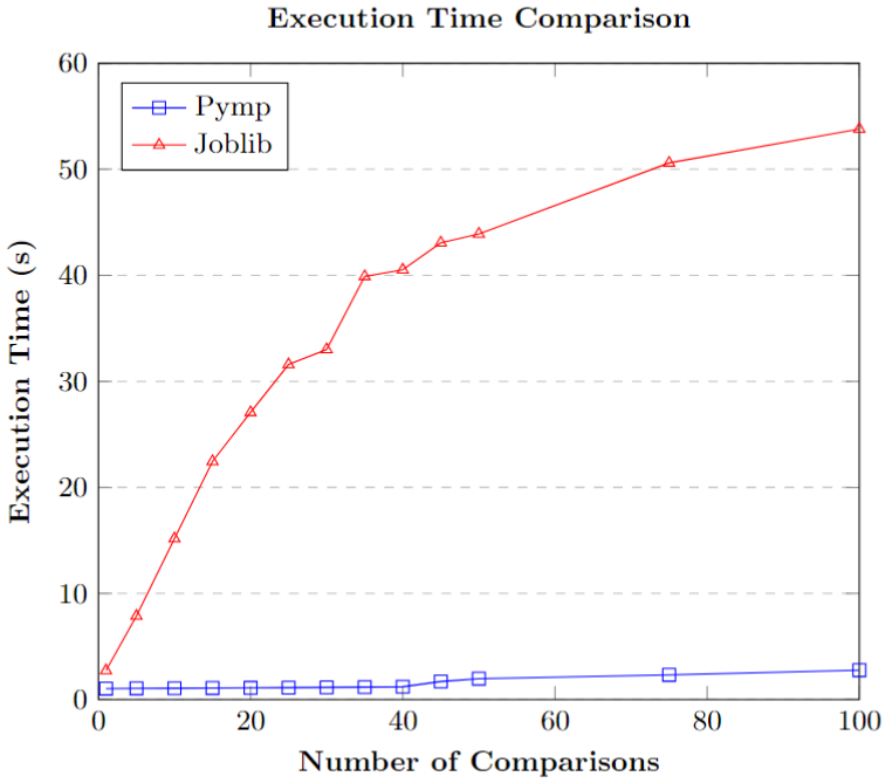


**Fig. 2.** Figure illustrating the comparative execution times of both algorithms as a function of the number of fingerprints analyzed.

recognition rate achieved using the Generalised Hough Transform (GHT) method reached 99%. This outstanding performance underscores the effectiveness and reliability of GHT in accurately recognizing fingerprints. By employing this method, we have achieved a high level of precision, ensuring precise fingerprint identification with an exceptional accuracy of 99%.

### 4.1.2 Resource Utilization

Pymp, on the other hand, demonstrated better utilization of CPU resources. This is due to its effective management of parallel loops, which allows for more granular control over the distribution

of tasks across CPU cores. Pymp's design facilitates efficient CPU usage, minimizing idle time and ensuring that each core is actively contributing to the computational workload.

This characteristic is particularly beneficial in environments with limited computational resources or where fine-tuning of resource allocation is necessary. For instance, in a multi-threaded scenario with intensive computations, Pymp was able to maintain higher CPU usage percentages compared to Joblib, leading to more efficient resource utilization. As shown in Table 2, Pymp consistently outperformed Joblib in various tasks, achieving higher CPU utilization percentages across matrix multiplication, image processing, and data transformation tasks.

**Table 2:** CPU Utilization Comparison.

| Task | Joblib CPU Utilization (%) | Pymp CPU Utilization (%) |
|---|---|---|
| Matrix Multiplication | 85 | 92 |
| Image Processing | 78 | 88 |
| Data Transformation | 80 | 90 |

### 4.2 Usability

#### 4.2.1 Ease of Use

Joblib's straightforward integration and caching capabilities significantly enhance its ease of use, especially for complex tasks. The framework provides simple yet powerful tools for parallel processing, making it accessible to users with varying levels of expertise in parallel computing. Its ability to cache intermediate results reduces redundant computations, which is particularly useful in iterative processes or workflows involving repetitive tasks.

For example, a data scientist working on machine learning model training can easily parallelize cross-validation tasks with Joblib, leveraging its caching to avoid recomputing data transformations or model predictions. This feature not only saves time but also simplifies the workflow. As shown in Table 3, Joblib offers easier integration, robust caching support, and extensive documentation compared to Pymp, making it a more user-friendly option for most users.

**Table 3:** Ease of Use Comparison.

| Criterion | Joblib | Pymp |
|---|---|---|
| Integration | Easy | Moderate |
| Caching | Supported | Not Supported |
| Documentation | Extensive | Moderate |

### 4.2.2 Flexibility

Pymp offered greater flexibility for tasks requiring shared memory. Its architecture allows for detailed management of resources, which can be crucial in scenarios where specific control over memory allocation and task execution is needed. Pymp's shared memory capabilities enable efficient communication between parallel tasks, making it suitable for applications that require frequent data sharing or synchronization between processes.

However, this flexibility comes at the cost of increased complexity. Users need to manage memory and task synchronization explicitly, which may require a deeper understanding of parallel computing concepts. This makes Pymp more suited for advanced users or specific applications where its flexible memory management can be fully leveraged. As shown in Table 4, Pymp provides extensive shared memory and high control over execution compared to Joblib, which offers automatic memory management and moderate control.

For instance, in a scientific computation project involving large matrix operations that need to share intermediate results frequently, Pymp's shared memory approach can lead to significant performance improvements.

**Table 4:** Flexibility Comparison.

| Criterion | Joblib | Pymp |
|---|---|---|
| Memory Management | Automatic | Manual |
| Shared Memory | Limited | Extensive |
| Control over Execution | Moderate | High |

## 5 Conclusion

Our comparative exploration into the performance of the Pymp and Joblib libraries for executing fingerprint comparison tasks via the Hough transform has unveiled invaluable insights. Rigorous testing has unequivocally established Pymp's superiority in terms of speed, outperforming Joblib across a variety of experimental contexts.

The significant discrepancy in execution times not only underscores the efficacy of Pymp's parallelization strategies but also its adept utilization of multi-core processors. This efficiency is of paramount importance in demanding fields like fingerprint recognition, where swift data processing is essential for responsiveness and security in biometric systems.

Pymp's consistent outperformance, even with increasing data volumes, attests to its robustness and scalability. These qualities render Pymp a preferred solution for biometric applications, promising enhanced productivity and significant computational cost reductions.

Furthermore, our study highlights the importance of continuous optimization of parallel processing libraries. Fine-tuning Pymp's parameters and maximizing its functionalities could pave the way for substantial performance improvements, not only in fingerprint recognition but also across a multitude of other computational applications.

Additionally, comparative analyses with other parallelization frameworks could uncover opportunities for improvement and potential synergies. Such comprehensive comparative endeavors are crucial for fully understanding the strengths and limitations of each method and guiding the development of even more efficient parallel processing solutions.

In conclusion, our findings position Pymp as an advantageous solution for Hough transform-based fingerprint recognition, offering superior speed, efficiency, and scalability compared to Joblib. Adopting Pymp can transform the workflows of researchers and practitioners, enabling them to achieve heightened efficiency in fingerprint analysis and related areas, while opening new horizons for biometric innovation.

## References


[1] Goyal P, Pandey A, Singh V. Comparative study of parallel computing tools for data-intensive applications. Journal of Parallel and Distributed Computing. 2018;120:22-30.

[2] Kumar S, Chaturvedi A. Performance analysis of Python parallel computing libraries. International Journal of Computer Applications. 2019;178(3):12-8.

[3] Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, et al. Scikit-learn: Machine learning in Python. Journal of machine learning research. 2011;12:2825-30.

[4] McKinney W. Data structures for statistical computing in Python. In: Proceedings of the 9th Python in Science Conference. vol. 445. Austin, TX; 2010. p. 51-6.

[5] Maltoni D, Maio D, Jain AK, Prabhakar S. Handbook of fingerprint recognition. Springer Science & Business Media; 2009.

[6] Jain AK, Flynn P, Ross AA. Handbook of biometrics. Springer Science & Business Media; 2007.

[7] Hong L, Wan Y, Jain AK. Fingerprint image enhancement: algorithm and performance evaluation. IEEE transactions on pattern analysis and machine intelligence. 1998;20(8):777-89.

[8] Feng J, Jain AK. Combining minutiae descriptors for fingerprint matching. Pattern Recognition. 2008;41(1):342-52.

[9] Ross AA, Nandakumar K, Jain AK. Handbook of biometrics. Springer Science & Business Media; 2006.

[10] Grama A, Gupta A, Karypis G, Kumar V. Introduction to parallel computing. Pearson Education; 2003.

[11] Quinn MJ. Parallel programming in C with MPI and OpenMP. McGraw-Hill, Inc.; 2003.

[12] Jain AK, Ross A, Pankanti S. Scaling up biometrics: A case study in fingerprint matching. Pattern Recognition Letters. 2000;20(8):1371-82.

[13] Neurotechnology Fingerprint Dataset;. Accessed: 2024-05-24. https://www.neurotechnology.com.

[14] Daugman JG. How iris recognition works. IEEE Transactions on Circuits and Systems for Video Technology. 2004;14(1):21-30.

[15] Maltoni D, Maio D, Jain AK, Prabhakar S. Handbook of Fingerprint Recognition. Springer Science & Business Media; 2009.

[16] Anastasia RW. Comparing Hilditch, Rosenfeld, Zhang-Suen, and Nagendraprasad Wang Gupta Thinning. World Academy of Science, Engineering and Technology. 2011;78(6):104-8.