Volume: 15 Issue: 01 Pages: 5786 - 5790 (2023) ISSN: 0975-0290

Image Processing on Agricultural Dataset Using Parallel Processing Based on Python

Faisal Dharma Adhinata
Doctoral Student of Computer Science, Universitas Gadjah Mada, Yogyakarta, Indonesia
Email: faisaldharma@mail.ugm.ac.id
Ahmad Ashari*
Department of Computer Science and Electronics, Universitas Gadjah Mada, Yogyakarta, Indonesia
Email: ashari@ugm.ac.id
Muhammad Alfian Amrizal
Department of Computer Science and Electronics, Universitas Gadjah Mada, Yogyakarta, Indonesia
Email: muhammad.alfian.amrizal@ugm.ac.id

----------ABSTRACT----------

Parallel processing divides data to be processed in each core on the CPU. One programming language that is widely used for machine learning applications and can accommodate the use of parallel processing is Python. Multiprocessing, mpi4py and CuPy are examples of Python libraries that can process data in parallel. In this research, we will compare the use of these libraries to process huge amounts of data. This research takes agricultural data for data image enhancement through color conversion from RGB to grayscale. The results showed that multiprocessing, mpi4py, and CuPy can increase the image enhancement speed three times faster than its single-core execution. Then, using a combination of multiprocessing with CuPy, a 1.7 times performance improvement is achieved compared with multiprocessing-only. Also, using a combination of mpi4py with CuPy achieved 2.5 times performance improvement compared to mpi4py-only.

Keywords - **Core CPU, parallel processing, mpi4py, multiprocessing, CuPy**

Date of Submission: March 28, 2023 Date of Acceptance: April 20, 2023

I. INTRODUCTION

World population growth from year to year continues to increase. Based on world statistical data [1], in 2022, the world's population will continue to increase, even reaching more than 8 billion. This increasing population increases the need for daily food [2]. Therefore, the agricultural sector is vital to meet the population's daily needs [3]. In addition, the agricultural sector is also a source of income for farmers. Some of the challenges faced by farmers in cultivating their agriculture are agricultural land that continues to decrease due to the increasing number of residential buildings. In addition, farmers are also required to produce good crops and increase production. One of the factors that affect the quality and productivity of crops is the presence of weeds that compete with the surrounding crops [4][5][6].

Currently, efforts to clear weeds are still done manually using pesticide sprayers operated by farmers [7]. This manual method requires effort and time, making it inefficient [8]. In addition, since weeds and crops often grow next to each other, it is difficult to spray only on the weeds, causing a waste of pesticides. Therefore, an automation system is needed to detect areas of weeds and crops for higher precision in spraying. One of the processing stages of the automation system is image enhancement. Image enhancement prepares the dataset before being processed in the next processing stage [9]. In this research, many agricultural image datasets will be used. Due to the large number of datasets, the computation time for image enhancement will be huge. Hence, a strategy to speed up the computation time is required.

One technique to speed up the computation time is parallel processing [10]. Parallel processing is designed by dividing the data execution into several computers / or cores of the Central Processing Unit (CPU). The use of parallel processing is widely applied in various fields. In this research, we are going to explore parallel processing in Python because this language is widely used for image processing in general [11]. The common libraries for parallel processing in Python include mpi4py [12], multiprocessing [13], and CuPy [14]. For example, Abutaha et al. [15] use parallel processing based on the Message Passing Interface (MPI) for image encryption. This encryption aims to secure image transmission over the network. The results of this research show that MPI can speed up the encryption process 1.5 times faster compared to single-core execution. Parallel processing is also used for real-time face recognition [16]. In this research, a comparison was made between the use of multiprocessing library and without it. Experimental results show that multiprocessing can make the execution time two times faster. Parallel processing can also be done by using both CPU and the Graphics Processing Unit (GPU) using the CuPy library. CuPy allows the usage of GPU using a parallel computing platform and programming model named CUDA (Compute Unified Device Architecture) [14]. Chetlur et al. [17] used CuPy to process image dataset using CNN and show an increase of 36% in computation time. This research will implement parallel processing in

Volume: 15 Issue: 01 Pages: 5786 - 5790 (2023) ISSN: 0975-0290

the context of image enhancement, namely converting RGB colors to grayscale.

The result of a photograph from a digital camera is an image with RGB color (3 color channels). However, the image processing stage usually uses grayscale color (1 color channel) to speed up computation time. This process requires parallel processing because the datasets are processed in large quantities and are large. This research will compare the utilization of parallel processing using mpi4py, multiprocessing, and CuPy libraries. The expected result is to increase the computation time in performing image enhancement of large datasets. Evaluation is done by measuring how fast the execution time is. The following are the contributions of this research:

- Implementation of mpi4py, multiprocessing, and CuPy to speed up the process of computing large amounts of agriculture dataset containing the images of crops and weeds.
- Extensive performance comparisons with single core execution.

II. Materials and Methodology

The flow chart explaining the methodologies of this research is shown in Figure 1. Datasets in the form of RGB images will be converted to grayscale using parallel processing with the mpi4py library, multiprocessing, and CuPy in Python.

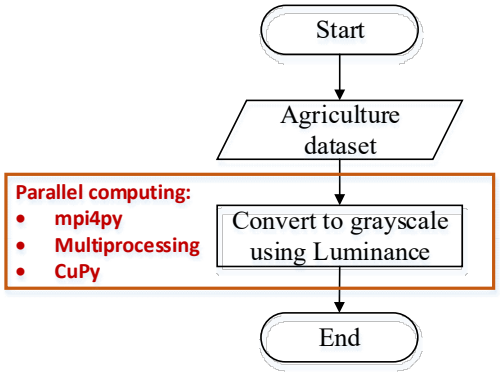

Fig 1. Procedure for comparing various Python's parallel processing libraries on agricultural dataset

A. Dataset and System Specification

The dataset used in this research contains 1931 images of corn and weeds photographed in the field in Ponorogo, Indonesia. The image resolution sizes used in this research are 1920 x 1080, 1280 x 720, and 640 x 360. The raw images use RGB as their color format. Figure 2 shows an example of the images used in this research.

Fig 2. An example of an image dataset in this research

B. Image enhancement data

Image enhancement data is a process that is carried out on RGB of raw data with grayscale operation. The equation used to change the color format from RGB to grayscale is shown in Equation (1).

$$\text{Gray} = 0.299 * \text{R} + 0.587 * \text{G} + 0.114 * \text{B} \quad (1)$$

R, G, and B are numbers between 0 and 255. Take note that the weights of R, G, B are not assigned equally. Green has a higher density than either red or blue, hence its weight is the largest. While blue is the darkest among the three, thus it carries the smallest weight.

Figure 3 shows the program to convert RGB to grayscale. This program has a function named RGB_to_Gray that accepts the RGB of each pixel stored as a NumPy array and returns the converted values in grayscale. This function calculates the dot product between the 3D matrix, representing the RGB values, and the weight vector [0.299, 0.587, 0.114], representing the color conversion's weight.

```
def RGB_to_Gray(rgb):
    return np.dot(rgb[...,:3], [0.299, 0.587, 0.114])
```

Fig 3. Code of converting RGB to grayscale

C. mpi4py

mpi4py (MPI for Python) is a library in Python that utilize the standard message exchange interface between multiple computers/cores named MPI (Message Passing Interface) [18]. Initially, it was a tool for high-performance computing in the early 1990s, with version 1.0 being launched in June 1994. In the early days of MPI, C/C++ and Fortran were utilized almost exclusively. Over time, versions of MPI or MPI bindings were made available for additional programming languages, including Java, C#, MATLAB, OCaml, R, and Python. Given the advent of machine learning with Python, mpi4py was built for Python using the MPI-2 standard's C++ bindings [12]. The schematic example of the MPI.COMM_WORLD communicator with six tasks and their corresponding levels is shown in Figure 4. The configuration of the MPI program in this research was carried out by splitting the image according to the number of cores in the system.

Fig 4. Example of MPI with six tasks corresponds to the task in core-0

The implementation of the mpi4py program is shown in Figure 5. The number of images distributed to each core is decided based on the number of images in the image_files folder divided by the number of cores. In this case, if the number of cores is four, the images in the folder will be divided by four and processed on the corresponding core according to the rank.

Volume: 15 Issue: 01 Pages: 5786 - 5790 (2023) ISSN: 0975-0290

```
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

num_images_per_core = len(image_files) // size
start_index = rank * num_images_per_core
end_index = start_index + num_images_per_core
```

Fig 5. mpi4py program

D. *Multiprocessing*

The Global Interpreter Lock (GIL), which permits only one thread to carry the Python interpreter at any given moment, causes single-CPU core usage in Python [19]. The GIL was implemented to address a problem with memory management, but as a result, Python is restricted to single-core execution. Bypassing the GIL when executing Python code enables the code to run quicker because multicore of the system may now be utilized. The built-in multiprocessing library of Python enables us to select specific parts of code to escape the GIL and execute concurrently on several cores. An illustration of the usage of multiprocessing in Python is shown in Figure 6. In this simplified example, if all three threads have identical execution durations, the total execution time will be reduced approximately by the factor of three.

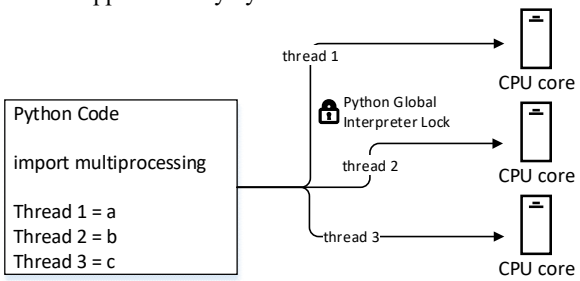

Fig 6. Illustration of multiprocessing in Python

The program code using the multiprocessing library is shown in Figure 7. The Pool function is responsible for preparing the worker threads. The number of worker threads is by default equals to the number of the system's logical CPU cores. In this code, list_image is the total number of images in the image_files folder.

```
def image_processing(image_files):
    img_ori = … # read image
    grayImage = RGB_to_Gray(img_ori)
    return

with Pool(workers) as p:
    p.map(image_processing, list_image)
```

Fig 7. Multiprocessing program

E. *CuPy*

CUDA (Compute Unified Device Architecture) is a platform for parallel processing as well as an application programming interface (API) that makes it possible for the software to use certain varieties of graphics processing units (GPUs) for general-purpose processing [20]. This method is referred to as general-purpose computing on GPUs (GPGPU). CUDA is a software layer that allows for direct access to the parallel processing units and the virtual instruction set of the GPU. This access is provided for the execution of computer kernels. Through parallel processing, CUDA speeds up the training process for machine learning and deep learning (i.e., instead of training on just a single compute instance, it divides the task and trains on different compute instances).

CUDA can be used in Python with the library CuPy [14]. It is implemented on top of CUDA and integrated with NumPy. The basic class for multi-dimensional arrays in CuPy is called cupy.ndarray, and many different functions are based on this class. It can speed up the already existing NumPy code by utilizing the GPU and CUDA libraries. The program code for using CuPy is shown in Figure 8. The code is almost the same as the original one, the main difference is, np.dot is replaced with cp.dot.

```
def RGB_to_Gray(rgb):
    gpu_weight = cp.asarray([0.299, 0.587, 0.114])
    return cp.dot(rgb[...,:3], gpu_weight)

for img in glob.glob('./dataHD/*.jpg'):
    image = cp.asarray(cv2.imread(img))
    gray = RGB_to_Gray(image)
```

Fig 8. CuPy program

F. *System Evaluation*

The computer system for the evaluation of the image enhancement code consists of an Intel Core i5-11400H (12 cores) CPU, an 8GB RAM, and an NVIDIA GTX 1650 GPU. Three different image resolutions, namely 1920 x 1080, 1280 x 720, and 640 x 360, were tested on six different implementations, i.e., simple (no parallel processing), mpi4py, multiprocessing, CuPy, mpi4py + CuPy, and multiprocessing + CuPy. For the mpi4py and multiprocessing implementations, the test is conducted by varying the number of CPU cores (2, 4, 6, 8, 10, and 12 cores). The evaluation results will be graphed to see the trends in execution time.

III. Result and Discussion

In this section, the discussion will start with a simple program for converting to grayscale color. Next, the program is modified using MPI. Then, simple programs are also modified by multiprocessing. Finally, the program is modified using GPU-based MPI or multiprocessing. The final section will compare the experimental results obtained with previous studies.

A. *Simple program*

A simple program uses a loop to call a dataset with the .jpg extension from a directory. The dataset is converted into a grayscale. The experimental results are shown in Table 1.

Table 1. Computation time using simple program

Resolution	Computation time (s)
1920 × 1080	126.0677
1280 x 720	55.6947
640 x 360	14.2379

Based on Table 1, the experimental results show that there is an effect of using image resolution on program computation time. The higher the image resolution, the

Volume: 15 Issue: 01 Pages: 5786 - 5790 (2023) ISSN: 0975-0290

longer the computation time. Even for a resolution of 1920 x 1080, it takes more than 2 minutes to process all datasets. Therefore, this simple program must be modified using mpi4py, multiprocessing, or CuPy libraries.

B. *mpi4py Program*

The MPI program in Python uses the mpi4py library. Figure 9 is an experimental graph of the program using mpi4py.

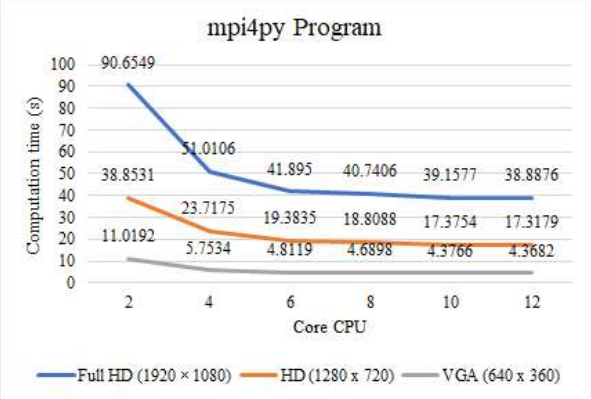

Fig 9. The result of experiment using mpi4py program

Based on Figure 9, the experimental results show that the more core values added to the CPU, the faster the computing time. For example, when compared with the simple program in Table 1, using mpi4py is three times faster than without using mpi4py. It is because there is an image division on each CPU core for RGB to grayscale conversion processing.

C. *Multiprocessing Program*

In this experiment, multiprocessing in Python uses the multiprocessing library. Then in Figure 10 is the result of an experiment using multiprocessing.

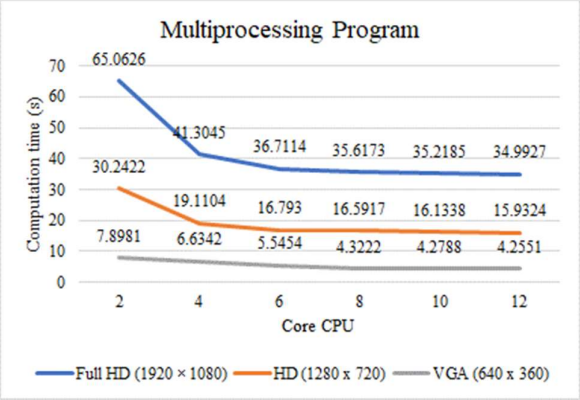

Fig 10. The result of experiment using multiprocessing program

Based on Figure 10, multiprocessing is three times faster than the simple program in Table 1. Compared to the use of mpi4py, as shown in Figure 7, multiprocessing is slightly faster than the use of mpi4py. This multiprocessing library uses several worker processes to process the program.

D. *CuPy Program*

The Python library used for GPU usage is CuPy. Then Figure 11 is the result of an experiment using a CuPy-based GPU.

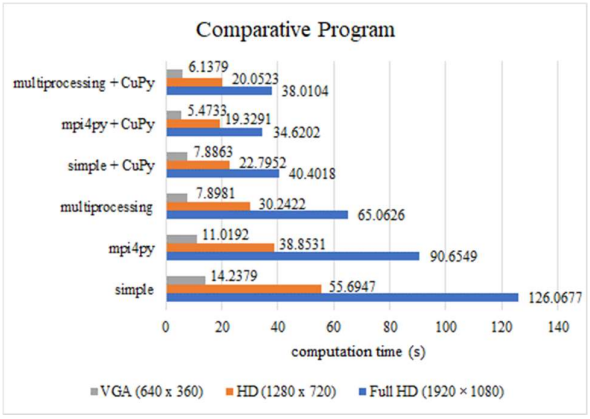

Fig 11. The result of experiment using simple, mpi4py, multiprocessing, and combined with CuPy

This CuPy usage experiment was carried out on a simple program, a combination of mpi4py and CuPy, and a combination of multiprocessing and CuPy. The number of cores used in this experiment is two cores. The experimental results are shown in Figure 11. First, The use of CuPy is three times faster than simple programs at Full HD resolution (1920 x 1080). Then, when using a combination of mpi4py and CuPy, the experimental results show that it is 2.5 times faster than regular mpi4py. Then, when using a combination of multiprocessing and CuPy, the experimental results show that it is 1.7 times faster than ordinary multiprocessing. These results are seen from the FullHD dataset.

E. *Discussion*

Big data processing requires a specific strategy to process it. In simple programming using Python, it only uses one thread at a time because it has a Global Interpreter Lock (GIL). In this research, the proposed strategy uses parallel processing. Using parallel processing in both mpi4py, multiprocessing, and CuPy can speed up the computation time of the programs created. Based on previous research, using mpi4py can increase performance by 1.5 times faster [15], then multiprocessing is two times faster [16], and using CuPy can speed up 0.36 [17]. In this research, using mpi4py and multiprocessing can speed up simple programs three times. Then the use of the CuPy combined with multiprocessing and mpi4py can speed up 1.7 and 2.5 compared to the use of ordinary multiprocessing and mpi4py. Suggestions for future research are to apply parallel processing for case studies that use real-time data. This real-time data requires the algorithm to run quickly to minimize delays.

IV. CONCLUSION

This paper compares parallel processing using MPI, multiprocessing, and GPU using Python programming. Python programming provides various libraries for parallel

Volume: 15 Issue: 01 Pages: 5786 - 5790 (2023) ISSN: 0975-0290

processing, such as mpi4py, multiprocessing, and CuPy. The use of parallel processing is an effort to speed up the program computation process. In addition, there are quite a lot of datasets to be processed. The results show that all the use of parallel processing libraries is proven to speed up computing time. The use of parallel processing is necessary to process large amounts of data to speed up program computation time.

REFERENCES

[1] P. D. Department of Economic and Social Affairs, World Population Prospects 2022, no. 9. 2022. [Online]. Available: www.un.org/development/desa/pd/.

[2] T. Talaviya, D. Shah, N. Patel, H. Yagnik, and M. Shah, "Implementation of artificial intelligence in agriculture for optimisation of irrigation and application of pesticides and herbicides," Artificial Intelligence in Agriculture, vol. 4, pp. 58–73, 2020, doi: 10.1016/j.aiia.2020.04.002.

[3] K. Prakash, P. Saravanamoorthi, R. Sathishkumar, and M. Parimala, "A Study of Image Processing in Agriculture," International Journal of Advanced Networking and Applications - IJANA, vol. 9, no. 1, pp. 3311–3315, 2017, [Online]. Available: https://www.ijana.in/v9-1.php#

[4] S. Sabzi, Y. Abbaspour-Gilandeh, and J. I. Arribas, "An automatic visible-range video weed detection, segmentation and classification prototype in potato field," Heliyon, vol. 6, no. 5, p. e03685, 2020, doi: 10.1016/j.heliyon.2020.e03685.

[5] A. Berquer, V. Bretagnolle, O. Martin, and S. Gaba, "Disentangling the effect of nitrogen input and weed control on crop–weed competition suggests a potential agronomic trap in conventional farming," Agriculture, Ecosystems and Environment, vol. 345, no. July 2021, p. 108232, 2023, doi: 10.1016/j.agee.2022.108232.

[6] C. Nathalie, N. Munier-Jolain, F. Dugué, A. Gardarin, F. Strbik, and D. Moreau, "The response of weed and crop species to shading. How to predict their morphology and plasticity from species traits and ecological indexes?," European Journal of Agronomy, vol. 121, no. August, 2020, doi: 10.1016/j.eja.2020.126158.

[7] P. Kanade, M. Akhtar, F. David, and S. Kanade, "Agricultural Mobile Robots In Weed Management And Control," International Journal of Advanced Networking and Applications, vol. 13, no. 03, pp. 5001–5006, 2021, doi: 10.35444/ijana.2021.13309.

[8] K. Xiao, Y. Ma, and G. Gao, "An intelligent precision orchard pesticide spray technique based on the depth-of-field extraction algorithm," Computers and Electronics in Agriculture, vol. 133, pp. 30–36, 2017, doi: 10.1016/j.compag.2016.12.002.

[9] A. Sharma and P. K. Mishra, "Image enhancement techniques on deep learning approaches for automated diagnosis of COVID-19 features using CXR images," Multimedia Tools and Applications, 2022, doi: 10.1007/s11042-022-13486-8.

[10] A. Al-shafei, H. Zareipour, and Y. Cao, "High-Performance and Parallel Computing Techniques Review: Applications, Challenges and Potentials to Support Net-Zero Transition of Future Grids," 2022.

[11] S. Farshidi, S. Jansen, and M. Deldar, "A decision model for programming language ecosystem selection: Seven industry case studies," Information and Software Technology, vol. 139, no. May 2020, p. 106640, 2021, doi: 10.1016/j.infsof.2021.106640.

[12] L. Dalcin and Y. L. L. Fang, "Mpi4py: Status Update after 12 Years of Development," Computing in Science and Engineering, vol. 23, no. 4, pp. 47–54, 2021, doi: 10.1109/MCSE.2021.3083216.

[13] D. Beazley, "Secrets of the Multiprocessing Module," Login, vol. 37, no. 5, pp. 61–70, 2012.

[14] R. Okuta, Y. Unno, D. Nishino, S. Hido, and C. Loomis, "CuPy: A NumPy-compatible library for NVIDIA GPU calculations," Workshop on Machine Learning Systems (LearningSys) at Neural Information Processing Systems (NIPS), no. Nips, pp. 1–7, 2017, [Online]. Available: http://learningsys.org/nips17/assets/papers/paper_16.pdf

[15] E. Based, P. Interface, M. Abutaha, I. Amar, and S. Alqahtani, "Parallel and Practical Approach of Efficient Image Chaotic Encryption Based on Message Passing Interface (MPI)," entropy Article, pp. 1–22, 2022.

[16] M. S. U. Yusuf and A. A. Fuad, "Real Time Implementation of Face Recognition based Smart Attendance System," Wseas Transactions on Signal Processing, vol. 17, pp. 46–56, 2021, doi: 10.37394/232014.2021.17.6.

[17] S. Chetlur et al., "cuDNN: Efficient Primitives for Deep Learning," pp. 1–9, 2014, [Online]. Available: http://arxiv.org/abs/1410.0759

[18] N. Sultana, M. Rüfenacht, A. Skjellum, P. Bangalore, I. Laguna, and K. Mohror, "Understanding the use of message passing interface in exascale proxy applications," Concurrency and Computation: Practice and Experience, vol. 33, no. 14, pp. 1–15, 2021, doi: 10.1002/cpe.5901.

[19] S. H. Lee, "Real-time edge computing on multi-processes and multi-threading architectures for deep learning applications," Microprocessors and Microsystems, vol. 92, no. December 2021, p. 104554, 2022, doi: 10.1016/j.micpro.2022.104554.

[20] M. J. Mišić, Đ. M. Đurđević, and M. V Tomašević, "Evolution and trends in GPU computing," in 2012 Proceedings of the 35th International Convention MIPRO, 2012, pp. 289–294.