Detecting Duplicate Images using MPI and Python

Sebastian Klemkosky

***Abstract*— The purpose of this project is to make a simple python program that can check if there is a duplicate image in a dataset. We will use parallel processing to search for all the duplicates in the dataset and output their directory.**

***Keywords*— Parallel Processing, Cloud Computing, Python, Mpi4py**

I. Introduction

Parallel Processing is a concept in cloud computing in which complex calculations, or tasks are broken up to be run simultaneously on multiple CPUs. Thus by doing so making the overall process time lower. We will demonstrate the significant benefits of parallel processing by showcasing it in action by doing a parallel search for duplicate images in a specific folder.

II. Chameleon Cloud, Python and Important Modules

Our program will take advantage of the Python programming language as well of MPI specifically mpi4py. In addition to other various Python modules.

A. Chameleon Cloud

Our project will be housed in the Chameleon Cloud servers. Chameleon is an NSF-funded testbed system for Computer Science experimentation. It is designed to be deeply reconfigurable, with a wide variety of capabilities for researching systems, networking, distributed and cluster computing and security[6]. For our project we will be using two virtual machines (VMs) which will both have the program and the same folder of images in them.

B. Python

Python is an interpreted, object oriented, high-level programing language with dynamic semantics[2]. It's very commonly used in Rapid Application Development as well as a simple scripting language to combine core components together. Two of Python's strengths are its simplicity and flexibility which would allow even novice programmers to learn quickly and efficiently. In addition Python supports packages which allows for programmers to create and import specific functions and libraries to better adapt the language and program to any need.

C. MPI

The Message Passing Interface or MPI is a standardized portable message passing system designed to function on a wide variety of parallel computers [1]. This standard defines the syntax and semantics of library routines which allow users to write programs in a variety of different languages[1]. These languages include Fortran, C, C++, and Python with the use of MPI4py.

D. NumPy

NumPy is a package that provides array manipulations and computational capabilities similar to those found in MATLAB[1]. NumPy allows for easy accessibility to its functions and is implemented quickly. Numpy provides more flexibility to the python arrays. In our project we will use Numpy arrays and populate them using the modules included functions.

E. OS

The OS module allows for interactions with the Operating System by providing functions in python to do so. With this module we will be able to get the current directory as well as the directory of specific files and folders.

F. PIL

PIL stands for Python Image Library and adds support for image processing. This includes opening, manipulating and saving images. This combined with the OS module will allow us to open and manipulate specific images given a file directory.

G. Hashlib

This module implements a common interface to many different secure hash and message digest algorithms[5]. This includes secure hash algorithms such as SHA1, SHA224, SHA256, SHA384, and SHA512. This module will allow us to hash our dataset into secure algorithms for later use.

H. Array

We will also import the array module in Python. Similar to Numpy this allows for Python array manipulation. The included function in this module will be useful to meet the goals of our project.

These modules will allow us to better implement our program and demonstrate the significant benefits of parallel processing.

III. Goals

The goals of our project's program are as follows:

1. Setup an MPI cluster of VMs, and populate it with several image files in the cluster.
2. Write an MPI program in Python to detect if a given image file already exists in the cluster. Similarity between images can be found by using a simple Hash-based technique.

3. When we want to check if a particular image file already exists in our dataset, the root process will first calculate the hash (e.g sha1) for that file, and perform a parallel look up using multiple worker processes to see if it's hash value matches with that of the various image files located on the VMs. For parallel lookup operation, each worker process will calculate the hash (e.g sha1) for an individual image file located on a VM and compare it with the hash value provided by the root process. The root process will display the result of this parallel look up.

IV. METHODOLOGY

This section will break down the steps of the creation of our project's program.

1) Example Images to Test

Before we start writing down our code it is important to create our dataset of images which would be used to test our program. For our program we went online and downloaded a collection of window 98 icons[7].

For this project's program we will consider a duplicate image as an image which is the same length, width and whose specific pixels are the same RGB and Alpha value. All images in this project will be of type .png for sake of simplicity. Fig 1 and Fig 2 below are examples of two different images which are similar in nature but are not duplicates.

Fig. 1 Example image of a windows 98 icon

Fig. 2 Different example image of a windows 98 icon

While both images are of a desktop personal computer, they are not considered duplicate images and will not be matched by our program.

An Image folder will be created to populate images of windows 98 icons. This folder is to be shared between both virtual machines.

2) Imports and Setup

As discussed before our program will use various different Python modules imported to easily achieve our project's goal. Fig 3 below shows our imports in use at the start of our program.

```
import os
from hashlib import sha1
from mpi4py import MPI
import numpy as np
from PIL import Image
from array import *
```

Fig. 3 Python imports used in our program

We will reference these imports later as we use specific functions contained within the module.

After we declare our imports, variables are created to be referenced by all processes within our program as seen in Fig 4 below. These shared variables contain important data about the rank of the current process, the total number of processes or size and allows us to send data between processes.

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

Fig. 4 mpi4py variables used in the program

In this next section of code we will find the directory of the image folder. With this directory the code will iterate through all files in it, and if that specific file is of type .png it will be added to a list of image files. In addition an empty list will be created to later be used to hold the matched directories.

```
image_folder = os.path.join(os.getcwd(),
'images')
image_files = [_ for _ in
os.listdir(image_folder) if _.endswith('png')]
image_path_dup_list = []
```

Each line of code is typed above to be referenced easier. The variable ***image_folder*** contains a string of the image folder directory. We achieve this by using the Python OS import and it's functions ***os.path.join*** which concatenates two components of a directory together and ***os.getcwd*** which gets the current working directory of the program. Together we are

able to concatenate our programs directory with an image folder named ***images*** and we can now reference the image folder directly.

The next line we create the list ***image_files*** and iterate through every file in the directory checking to see if that file is of type .png if so it will be added to the list. To be able to iterate through the file we make use of the function ***os.listdir*** which returns a list of entries in the directory.

The final line of code just creates an empty list called ***image_path_dup_list*** to hold the matched directories of any duplicate images, this will be used later.

3) Main Process Part I

Rank 0 will be referred to as the main process. This main process will handle many of the setup for the other processes. However since every process will also be running this program we will need a way to specify that only the main process will be able to access this portion of the code. To do so a simple if statement is created.

```
if rank == 0:
```

The variable rank was initialized before and every process will have a different rank. The other processes will skip this section of code. We will then ask the user to input the name of the image as below in Fig 5.

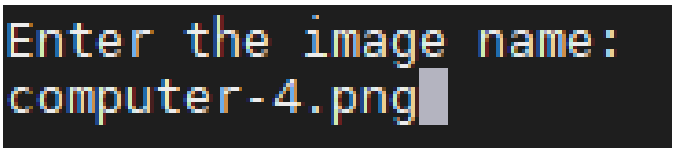

Fig. 5 User being asked to enter image name

Error handling is implemented in the case the user doesn't input a valid image name. When this happens all processes are stopped and the program exits.

After the main process receives a valid input a variable is created.

```
orig_image_path = os.path.join(image_folder,
user_input)
```

The variable ***orig_image_path*** will be assigned the concatenation of the image folder directory and the user's input using the ***os.path.join*** function as seen before.

```
orig_image = Image.open(orig_image_path)
orig_image_array = np.asarray(orig_image)
```

Two more variables are created: ***orig_image*** and ***orig_image_array***. The first uses the PIL module imported before to create an image object. This object will be processed and used in the next statement which creates a Numpy array from this object.

The Numpy array created will be a multidimensional array used to represent the image. Fig 6 will be an enlarged example image and Fig 7 is the original scaled image. Notice that in Fig 6 only the top left of the image is the color black. The gradient is due to the image being scaled up and can be ignored in Fig 6.

Fig. 6 A image of to showcase how Numpy converts images to arrays

.

Fig. 7 This small image contains only a single black pixel in the top left of the image

The variable ***orig_image_array*** will contain this multidimensional array:

```
[[[  0   0   0 255]
  [255 255 255 255]
  [255 255 255 255]
  [255 255 255 255]
  [255 255 255 255]
  [255 255 255 255]
  [255 255 255 255]
 [255 255 255 255]]

 [[255 255 255 255]
  [255 255 255 255]
  [255 255 255 255]
  [255 255 255 255]
  [255 255 255 255]
  [255 255 255 255]
  [255 255 255 255]
 [255 255 255 255]]

          ...

 [[255 255 255 255]
  [255 255 255 255]
  [255 255 255 255]
  [255 255 255 255]
  [255 255 255 255]
  [255 255 255 255]
  [255 255 255 255]
 [255 255 255 255]]]
```

The most inner array contains the RGB and Alpha of a pixel in the image. As you can see only the first inner array has an RGB and Alpha of [0,0,0,255] which translates to the color black. The other inner arrays have a RGB and Alpha of [255,255,255,255] which translates to the color white. In addition the array translates the image from top to bottom and left to right. Because the original image as seen in Fig 7 contained a single black pixel in the most top left, we know that Numpy accurately converted the image object into an array.

Next is to convert ***orig_image_array*** into sha1

```
match = sha1(orig_image_array).hexdigest()
```

Using the import Hashlib we make use of the functions ***sha1*** and ***.hexdigest*** to encode and return the data in hexadecimal format. We set the variable ***match*** to this hexadecimal string. Which will be used to later check the images if they are duplicates.

Now that we have a way to compare the original image to other images we need a way to communicate and send this information to the other processes. We do this by using MPI (specifically mpi4py for Python)

```
chunk = len(image_files) // size
image_files = [image_files[c:c + chunk] for c in
range(0, len(image_files), chunk)]
```

The two lines above take the length of the list ***image_files*** in order to find the total number of files in the folder which gets divided by the total number of processes as found before. This gets set to the integer variable ***chunk*** to be used to split the list ***image_files*** into equal chunks within the list.

```
    for x in range(size):
      comm.send(image_files[x],dest=x, tag=12)
      comm.send(match,dest=x, tag=11)
      comm.send(orig_image_path,dest=x, tag=13)
```

The code above takes all the necessary data and sends it to each other process including the main processes. Each process will be assigned a portion of the ***image_files*** list to search through, as well as the original image hash contained in the variable ***match*** to compare with. In addition the processes will also be sent the directory of the original image. This will be used to make sure the processes don't flag the original image as a duplicate.

4) Searching for Duplicates

All the processes can now begin their search for duplicate images in the image folder. Since we need the main process to gather all the necessary data for the search, the other process needed to wait.

```
comm.Barrier()
```

This function forces a process to wait until all other processes have called this function. This is a commonly used function as there are many cases in parallel programming where it is important to synchronize processes. In our case this is used to halt the other processes while the main process runs.

```
imagelist = comm.recv(source=0, tag = 12)
match = comm.recv(source=0, tag = 11)
orig_image_path = comm.recv(source=0, tag = 13)
```

The three lines of code above is how the other processes receive the information from the main process. We make use of another MPI function ***comm.recv*** to do so. Since the data we are receiving is all coming from rank 0, the main function, the source is set to 0. The tag specifies what data gets received where. The variable ***imagelist*** is the list that a specific process will search for duplicates which was split earlier in the main process. The hexadecimal string ***match*** is the set to the same as before and the ***orig_image_path***. It is important to notice that while these variables share the same name, this is the first time the other processes are using them as they haven't been initialized yet since that only occurred in the main process.

```
for i in imagelist:
  cur_image_path = os.path.join(image_folder, i)
  cur_image = Image.open(cur_image_path)
  image_array = np.asarray(cur_image)
  cur_image_hash = sha1(image_array).hexdigest()
```

The ***for*** loop above iterates through every element in the ***imagelist*** array. The rest of the code is very similar to what was already discussed in the main function. Remember that each element in the ***imagelist*** array is a string containing the name of the image and ***cur_image_path*** is the directory of that specific image which is concatenated using ***os.path.join*** like before. This works because the name of the image file is only used to find its location to be referenced. From there it converts the found image into an image object. Since the image objects are converted to a Numpy array as seen before, the file name of the image does not matter when searching for duplicates. Only ***image_array*** which is determined by the RGB and Alpha values per pixel of the image matters and that is what is then hashed using ***sha1*** and that hexadecimal string is contained within ***cur_image_hash.***

```
if cur_image_hash == match and cur_image_path !=
orig_image_path:
image_path_dup_list.append(cur_image_path)
```

This ***if*** statement is still within the ***for*** loop and it checks to see if the hexadecimal string ***cur_image_hash*** matches what was already found in the main processes contained in ***match*** which was received from the main processes before. It also checks to see if the current image directory is not the same as the original image's directory so that we don't count the original image as a duplicate. If the current image's hexadecimal string matches the one provided from the main process then the directory path to the current image is added to the list ***image_path_dup_list*** which was initialized as empty at the start of the program.

```
image_path_dup_list =
comm.gather(image_path_dup_list, root = 0)
```

After a process has searched through all of their images in their list, the function ***comm.gather*** will gather all the different ***image_path_dup_list*** lists from each process and combine them at the main process.

5) Printing the Results

In the main process, after all processes have searched through their respective list. The main processes will collect every ***image_path_dup_list*** from them and print out the results. If any duplicates are found the main processes will print their directory paths.

```
Enter the image name:
computer-4.png
Duplicate Images are found at
/home/cc/images/computer-4 - Copy.png
```

Fig. 8 Example of running the code

Fig 8 above shows the output of running the code. In this test both virtual machines contain the exact same image folder and we are searching for duplicates of the image shown in Fig 4. One duplicate image of Fig 4 was placed in the folder to have something to search for. The similar image shown in Fig 5 as well as various other images are included in the image folder as well. The program was able to correctly find the directory path of the only duplicate image in the folder.

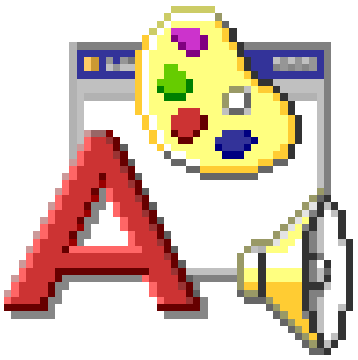

Fig. 9 a.png which is located at the beginning of a folder

It is common for files to be automatically sorted in a directory folder in many operating systems. In our program we split the files in the folder by the order they are already in so the name of the image is ***computer.png*** and the name of the duplicate image is ***computer - Copy.png*** it is likely that they will be both in the same list when the split occurs as well as the same process. To test this we will create a duplicate image of Fig 9 and rename it to ***z.png***. Since the duplicate image will be sorted to the very end of the folder and our original image will be at the beginning, both images will be in different lists and in different processes.

```
Enter the image name:
a.png
Duplicate Images are found at
/home/cc/images/z.png
```

Fig. 10 Found duplicate on a different process

Another test done was to check if images that were flipped would be flagged as a duplicate. Fig 11 and Fig 12 are the same images just flipped. flipped_mouse.png

Fig. 11 mouse-0.png

Fig. 12 flipped_mouse.png

Remember when we define what exactly a duplicate image means as stated before we will consider a duplicate image as an image which is the same length, width and whose specific pixels are the same RGB and Alpha value. Another thing to recall is how images were organized in the image array. Fig 7 only contained a single black pixel and was located at the topmost left part of the image. This translated to the first most inner array having the RGB and Alpha values [0,0,0,255]to represent color and location.

```
Enter the image name:
mouse-0.png
Duplicate Images are found at
```

Fig. 13 Example of duplicate not found for the flip test

Since Fig 11 and Fig 12 are not considered duplicate images the program will not match them as such.

V. Project Outcome

The goals of the project were met and overall it was a success. A MPI cluster was set up and populated with several image files as shown in part IV section 1. The second goal was met and a hash-based technique was used to compare similarity of images. The third goal was achieved as the root or main process was used to calculate the sha1 hash for a specific file which was sent to all processes which would also calculate the sha1 of a specific image file and compare. At the end of the search all results were sent to the root process which displayed the results.

VI. Contributions

The entire program and research done for the project was completed by Sebastian Klemkosky. The report and the demonstration video was created and edited by Sebastian Klemkosky as well. Access to the Chameleon Cloud and Virtual Machines used were provided by Dr. Palden Lama for the use of students enrolled in his cs4843 Cloud Computing course section 01T for the Summer 2021 semester.

References

[1] Mpi4py.readthedocs.io. 2021. Introduction — MPI for Python 3.0.3 documentation. [online] Available at: <https://mpi4py.readthedocs.io/en/stable/intro.html>

[2] Python.org. 2021. What is Python? Executive Summary. [online] Available at: <https://www.python.org/doc/essays/blurb/>

[3] Numpy.org. 2021. NumPy. [online] Available at: <https://numpy.org/>

[4] GeeksforGeeks. 2021. OS Module in Python with Examples - GeeksforGeeks. [online] Available at: <https://www.geeksforgeeks.org/os-module-python-examples/>

[5] Docs.python.org. 2021. hashlib — Secure hashes and message digests — Python 3.9.6 documentation. [online] Available at: <https://docs.python.org/3/library/hashlib.html>

[6] Chameleoncloud.readthedocs.io. 2021. Welcome to Chameleon — Chameleon Cloud Documentation. [online] Available at: <https://chameleoncloud.readthedocs.io/en/latest/index.html>

[7] Win98icons.alexmeub.com. 2021. Windows 98 Icon Viewer. [online] Available at: <https://win98icons.alexmeub.com>