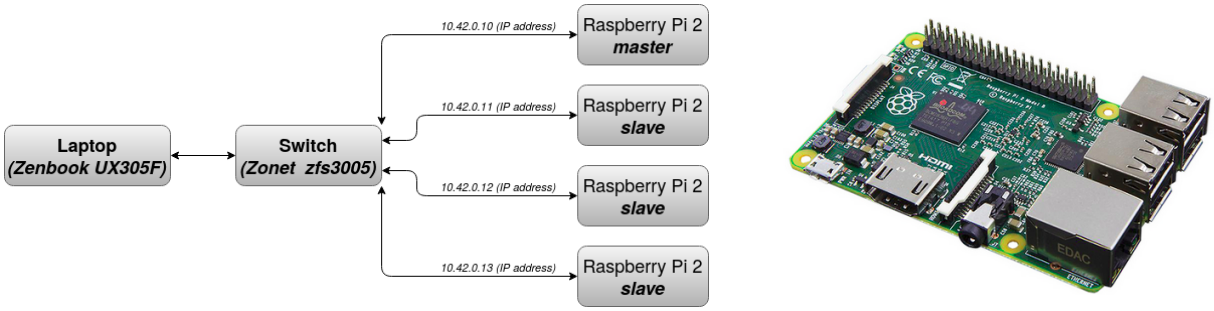
1 Introduction

When processing big amount of data or jobs for which a large amount of resources are required, newer computers often tend to be a pretty expensive solution. If the job does not have to be computed on a single unit, a distribution of the workload on multiple entities could be a possibility. If one got access to a supercomputer or another campus where multiple machines are accessible, this could effectively solve your problem. Hence, if parallelism is possible for the given task, a cluster of nodes could be used to decrease the run time of the calculations.

This project presents a way to build a Raspberry Pi cluster where parallel computing of big data will be distributed to multiple nodes using SLURM. The compute nodes will share data on the master node by building a network file system (NFS). The computation of data will then be distributed to the compute nodes using OpenMPI where the code is written in python. The time for which the data has been processed and completed will be compared to that of a regular computer to see how fast a Pi cluster is able to compute big amount of data.

Four Raspberry Pi 2 will be used in the cluster. This Raspberry Pi can be seen in Figure 1b. One of them will be a master and the rest slaves. A USB with a capacity of 128 GB will be used in the master for which file sharing will be utilized between the slaves. The laptop has an Intel Core M-5Y10c CPU 0.80GHz $\times$ 4 with 8 GB RAM.



(a) Raspberry Pi cluster *(b) Rapsberry Pi 2*

Figure 1: The Raspberry Pi 2 along with an illustration of the Raspberry Pi cluster with there IP addresses

For communication between the nodes and the laptop, a network switch with five I/O will be used. Furthermore, four MicroSD cards will be used to store the Raspbian Buster Lite image used to run the nodes.

The reason for choosing a Raspberry Pi 2 is the low cost of the unit as well as its good performance which will be put to a test in the following sections along with a covering of the setup of the Raspberry Pi cluster.

2 Raspberry Pi setup

The Raspbian Buster Lite operating system will be used on the nodes in order to minimize software as this is appropriate to solve the task. To flash the image to the Raspberry Pi, *balenaEtcher* is used. After this operation, an empty file called *ssh* is created in the boot drive of the MicroSD card in order to enable secure shell (SSH) as the communication between the Raspberry Pi and laptop. Now using **ssh pi@raspberrypi.local** from the laptop a communication is established. It is important that the default password will be changed which is *raspberry*. To do this, enter **raspi_confic** and follow the instructions. In the *raspi_config* menu, it is also possible to allocated all memory to the operating system in the *advanced options* and hence expand *files system*.

As a preparation to using SLURM, the hostname is changed to *node01* in the files /etc/hostname and /etc/hosts using **sudo hostname node01**, **sudo nano /etc/hostname** and **sudo nano /etc/hosts** respectively. This is done for the master (node01) and the slaves with numbers going from 2-4. Nano is the default command line text editor along with Vi in the Raspberry Pi.

2.1 Mount flash drive

The external storage which is going to be used as shared memory between nodes in the cluster, will be installed on the master (node01).

Using the command **lsblk**, a list of drives on the Raspberry Pi are listed in the terminal. In this case, the drive /dev/sda1 is the wanted USB. The drive is now formatted using **sudo mkfs.ext4 /dev/sda1**. The next step is to create a directory where the shared folder should be placed. This is done with **sudo mkdir /clusterfs**, **sudo chown nobody.nogroup -R /clusterfs** and **sudo chmod 777 -R /clusterfs** for making the directory, changing the ownership and read-write permissions respectively.

Each file system gets assigned a universally unique identifier (UUID) during formatting. Since it cannot be changed, it is an ideal way to select file systems for mounting [**fstab**]. To setup automatic mounting, this UUID for the flash drive must be known which can be found using **blkid**. Now the line *(UUID=”7e098cc9-23f9-4226-a73e-6fdeed70bd83” /clusterfs ext4 defaults 0 2)* can be added in the file /etc/fstab. Here the chosen directory is selected along with file type and fifth field with a 2 indicating that it is not a root file system [**fstab_pass**]. To mount the drive, **sudo mount -a** can be used or just reboot the Pi.

2.2 Networking

The Raspberry Pis must have access to the internet. This is accomplished by setting up a shared network where the Pis will use the network from the laptop with a connection named *Cluster* which is made using *nm-connection-editor* in the IPv4 settings where *shared to other users* are chosen.

First the laptop is setup for the shared network. A file called rc.nat is created in /etc/rc.d directory. Here the lines *(iptables=/sbin/iptables)*, which specifies the location of the iptables executables, *(iptables –flush -t nat)*, which clears iptables rules, *(iptables –table nat –append POSTROUTING –out-interface ppp0 -j MASQUERADE)* and *(iptables –append FORWARD –in-interface eth0 -j ACCEPT)*, for setting network address translation (NAT) and forwarding of packages and then *(echo 1 > /proc/sys/net/ipv4/ip_forward)* for enabling of packet forwarding, will be added to the file. Now make this file executable and it will be executed every time the computer is started [**Networking**].

To make the setup of each Raspberry Pi more systematic, the interface name for each Pi is changed to a constant value, namely *etc0*. This is done finding the MAC address of each Pi during setup. Then adding the lines *(ATTRaddress==”b8:27:eb:11:5e:b0”)* and *(NAME=”eth0”)* to the file /etc/udev/rules.d/70-persistent-net.rules and adding the lines *(DEVICE=”eth0”)* and *(HWADDR=”b8:27:eb:11:5e:b0”)* to the file */etc/sysconfig/network-scripts/ifcfg-eth0*. The master node is used as a description of the implementation for which its MAC address is used. This must of cause be changed accordingly to the other nodes.[**Interface_Name**]

In the Raspberry Pi, a static IP address is setup by adding the lines *(interface eth0)*, *(static ip_address=10.42.0.10/24)*, *(static routers=10.42.0.1)* and *(static domain_name_servers=10.24.0.1)* to the file /etc/dhcpcd.conf (there is no specific reason for choosing the mentioned IP address). Because this IP address it set for the master node, the slaves will have there last byte going from 11-13.

2.3 Network file system (NFS)

NFS is a program used to share files and folders between Linux/unix systems and hence an optimal chose for this project. The mounted drive (USB) on the master node must be exported as a network file system. This enables the other nodes on the network to get access to it. The NFS server is installed on the master node using **sudo apt install nfs-kernel-server**. By adding the line *(/home/pi/clusterfs 10.42.0.0/24(rw,sync no,root_squash no,no_subtree_check)* to the file /etc/export, the content of the given directory is exported to the remote hosts. The parameters gives read-write permission, forced changes to be written on each transaction, enables the root users of clients write files with root permissions and prevention of error if race condition occurs during read-write of files. The kernel server can now be updated using **sudo exportfs -a**.

Now the NFS client must be installed on the slave nodes which is done using the command **sudo apt install nfs-common**. Then a shared folder can be made using **sudo mkdir /clusterfs** with owner and permissions changed using **sudo chown nobody.nogroup /clusterfs** and **sudo chmod -R 777 /clusterfs**. To make the NFS share to mount automtically when nodes boot, the /etc/fstab file should be changed with the line *(10.42.0.10:/home/pi/clusterfs /home/pi/clusterfs nfs defaults 0 0)*. This sets the file system, mount point, type, options, dump and pass. [**NFS**]

3 Slurm

Simple Linux Utility for Resource Management (SLURM) is an open source highly scalable cluster management and job scheduling system for large and small Linux clusters. It can allocate access to resources for users for a duration of time so they can perform work. Moreover, it provides framework for starting , executing and monitoring work on allocated nodes along with an arbitration contention for resources by managing a queue of pending work. Hence this software is chosen for management of the Raspberry Pi cluster. Fore more information visit [**SLURM**].

First the master node is going to be configured. As already mentioned, the Raspberry Pi with IP address 10.42.0.10 (Node01) will be the master. To make access to the units in our cluster easier, the line *(10.42.0.X nodeY)* is added to the file /etc/hosts for every Pi connected to our cluster with the IP address going from 11-13 (X) and node from 2-4 (Y). Then the SLURM controller packages will be installed using **sudo apt install slurm-wlm**. In order to initiate SLURM with default configurations, the file slurm.conf.simple is fetched to /etc/slurm-llnl using **cp /usr/share/doc/slurm-client/examples/slurm.conf.simple.gz .** Now unzip the file **gzip -d slurm.conf.simple.gz** and renaming it to slurm.conf. We can make changes to the configuration which fits our needs. The control machine (master) has to be specified in this file with the line *SlurmctldHost=node01(10.42.0.10)*. Now in the bottom of the file, all the nodes in the cluster have to be specified with the line *(NodeName=node01 NodeAddr=10.42.0.10 CPUs=4 State=UNKNOWN)*. Again the nodes will go from 1-4 and the last byte of the IP address from 10-13. This is done so that SLURM knows all the units and the amount of resources in the cluster. SLURM runs of partitions for which nodes can be given. Two lines with a partition name along with the available nodes will be added as *(PartitionName=mycluster1 Nodes=node[02-04] Default=YES)* and *(MaxTime=INFINITE State=UP)*. The nodes given will be the compute nodes of the cluster. It is important that this file is consistent across all nodes in the cluster.

The way to restrict access to system resources is to create and modify the cgroup.conf file which will be utilized in the directory */etc/slurm-llnl/*. This will limit the amount of resources SLURM gives to each job. The configurations can be seen to the left in Figure 2.

```
CgroupMountpoint="/sys/fs/cgroup"
CgroupAutomount=yes
CgroupReleaseAgentDir="/etc/slurm-llnl/cgroup"
AllowedDevicesFile="/etc/slurm-llnl/
cgroup_allowed_devices_file.conf"
ConstrainCores=no
TaskAffinity=no
ConstrainRAMSpace=yes
ConstrainSwapSpace=no
ConstrainDevices=no
AllowedRamSpace=200
AllowedSwapSpace=0
MaxRAMPercent=100
MaxSwapPercent=100
MinRAMSpace=30
```

```
/dev/null
/dev/urandom
/dev/zero
/dev/sda*
/dev/cpu/*/*
/dev/pts/*
/clusterfs*
```

Figure 2: Configuration of cgroup.conf and cgroup_allowed_devices_file.conf respectively

The *CgroupAutomount* is set to yes so that the plugin will first try to mount the required subsystems. If not set, it will fail if they are not available. The *CgroupMountpoint* defines the path under which cgroups should be mounted. The *AllowedDevicesFile* specifies devices to use which is configured in the file *cgroup_allowed_devices_file.conf* that can be

seen to the right in Figure 2. The rest of the lines primarily specifies minimum, maximum and permission to use cores and memory of the system. [**cgroups**] [**piCluster**].

Munge is used as the default authentication mechanism in SLURM which is similar to key-based SSH. It will use a private key across all nodes, then requests are timestamp-encrypted and sent to the nodes that encrypts them using the identical key. For this to work, the time across all nodes must be synchronized. This is accomplished by installing *ntpdate* on all nodes using **sudo apt install ntpdate**.

Now the files slurm.conf, cgroup.conf and cgroup_allowed_devices_file.conf are copied to the /clusterfs directory using **sudo cp slurm.conf cgroup.conf cgroup_allowed_devices_file.conf /clusterfs**. Also the munge key will be copied to this directory as **sudo cp /etc/munge/munge.key /clusterfs**. This is done so that the configuration file is the same across all nodes so that they can be controlled by SLURM. Now we can start Munge, the SLURM daemon and control daemon using **sudo systemctl enable munge**, **sudo systemctl start munge**, **sudo systemctl enable slurmd**, **sudo systemctl start slurmd**, **sudo systemctl enable slurmctld** and **sudo systemctl start slurmctld** respectively.

Now on all the compute notes the slurmd and slurm-client must be installed using **sudo apt install slurmd slurm-client**. Like with the master node, the IP address and node name will be added to the file /etc/hosts of all the other nodes in the cluster. Moreover, the munge.key, slurm.conf and cgroup files from the created /clusterfs directory will be copied to each node using **sudo cp /clusterfs/munge.key /etc/munge/munge.key**, **sudo cp /clusterfs/slurm.conf /etc/slurm-llnl/slurm.conf** and **sudo cp /clusterfs/cgroup* /etc/slurm-llnl**. This way we can be sure that we have the same configuration files and Munge key across all the nodes in the cluster. Now we just have to enable and start Munge and the SLURM daemon using **sudo systemctl enable munge**, **sudo systemctl start munge**, **sudo systemctl enable slurmd** and **sudo systemctl start slurmd** respectively.

4 Big data processing

OpenMPI is an open source message passing interface which enables a single script to run a job spread across multiple nodes in the cluster. To use OpenMPI, the following must be installed on every node **sudo apt install openmpi-bin openmpi-common libopenmpi3 libopenmpi-dev**.

The approach to achieve parallelism is build upon a technique called SPMD (single program, multiple data), where tasks are split up and run simultaneously on multiple processors. This means that all available processors in the cluster will be given a clone of the program to execute. An example is the *Do loop*, where different processors will work on separate parts of the arrays involved in the loop. This is exactly the method which will be used in this section to achieve parallelism.[**SPMD**]

Because the most time consuming operation in the cluster will be the communication and sharing of data between nodes, two programs will be tested. A program of computing an integral running locally on each node will performed. Moreover, a program for counting pumpkins in a field where sharing of the data will happen from the USB of the master will also be utilized. The outcome of this will give an indication of the performance of the cluster compared to that of a laptop.

The code created for distributing the work of counting pumpkins to multiple processors in the cluster can be seen in Figure 3. Here the module *mpi4py* is used for python. In line 9-11, MPI is initialized by making a communication object for which data can be send between nodes. The *rank* will be the specific index of each process and *p* is the number of wanted processes. *p* does not have to be the number of available cores in the system. However, to achieve parallelism, this must be the case. In the line 14-27, the blocks are read into memory and split up into minor chunks in regard to the number of processes to run. This will effectively make arrays with a specific interval of blocks to analyze for pumpkins in the orthomosaic seen in Figure 6. When a process finishes, its result will be collected through the communication object with there results summed up to a total summation of all found pumpkins.

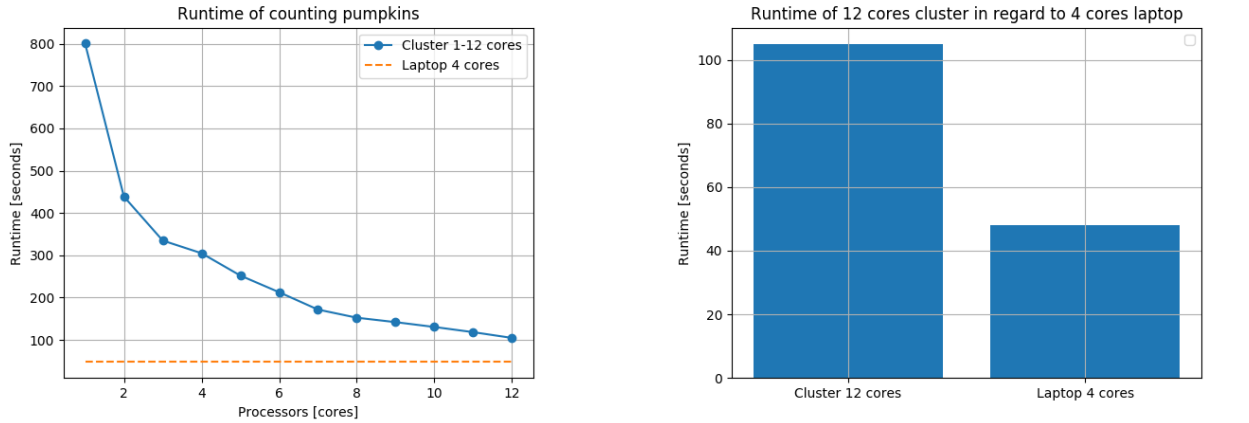
```
import sys
import os
from mpi4py import MPI
from big_data_processing  import*

instance = data_processing()

#Init OpencvMPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
p = comm.Get_size()

#Get number of blocks
instance.read_blocks_from_file(output/blocks.txt)
n = len(instance.list_of_blocks)

#Get number of local blocks to calculate
local_n = n/p

#Find interval for each process based on rank
interval = []
interval.append(rank*local_n)

if rank == (p-1):
    interval.append(rank*local_n + local_n + (n - rank*local_n - local_n))
else:
    interval.append(rank*local_n + local_n)

#Find pumpkins
test_images = ['../input/test_image_pumpkin3.png', '../input/test_image_pumpkin4.png']
instance.partition_blocks(interval)
pumpkins = instance.process_blocks(../input/orthomosaic1.tif,test_images,lab,1,4.5)

total_pumpkins = comm.reduce(pumpkins)

#Print final result
if rank == 0:
    print(Total number of pumpkins found: + str(total_pumpkins))

MPI.Finalize
```

Figure 3: The code in the script cal_pumpkins.py for parallel processing using OpenMPI

```
#!/bin/bash
#SBATCH --ntasks=12

cd $SLURM_SUBMIT_DIR

sudo touch runtime_pumpkins_history.txt
sudo chmod -R a+rw runtime_pumpkins_history.txt

for p in {1..12}
do
        start_=`date +%s%3N`
        mpiexec --allow-run-as-root -n $p python cal_pumpkins.py
        end_=`date +%s%3N`

        runtime=$((end_-start_))
        text=nodes $runtime p

        echo $text >> runtime_pumpkins_history.txt
        echo $runtime
done
```

Figure 4: The code in the script sub_mpi_pumpkins.sh for parallel processing using OpenMPI and SLURM

When using SLURM, a number of commands can be used to distribute jobs to nodes in the cluster. The *sbatch* command submits batch jobs to SLURM. This can either be accomplished by entering this command with specific parameters from the terminal or from a batch script. In Figure 4, the batch script used to test the execution time of the detection of pumpkins, which utilizes OpenMPI, can be seen. Here the number of tasks are set to the number of available cores in the cluster, which was given to the slurm.conf file in Section 3. Then the script are set to be launched from the current directory. However, it may be necessary to include the line *(sys.path.append(os.getcwd()))* from the modules *os* and *sys* in the top of the execution files in python, because SLURM is launched from a different directory than that of the placement of the files to import. Now the run time of counting pumpkins will be evaluated using one to twelve cores. This will be compared to that of a laptop having an Intel Core M-5Y10c CPU 0.80GHz $\times$ 4 with 8 GB RAM. The result of this can be seen in Figure 5.



(a) Run time of counting pumpkins

(b) Run time difference between laptop and 12 cores cluster

Figure 5: The performance of a compute cluster in regard to a laptop in counting pumpkins

Figure 6: Illustration of the field in which pumpkins must be found. This is an orthomosaic made by the program Metashape from Agisoft with a size of 23890×6009 *pixels*

It may be noticed that the biggest gain of using a multicore system is going from 1 to 2-4 cores. The reason for this being that when the number of cores increases, the inter-machine communication increases, while the amount of compute work per core decreases [**multicore_system**]. As mentioned earlier, the orthomosaic image is located on the master where

the nodes share data. This means that the throughput is reduced significantly as all the nodes must get the blocks of the orthomosaic when they each have found the pumpkins in a block of the image. One could have placed the othomosaic in the local memory of each node. This would have increased the performance as the nodes would only have to communicate in the beginning and end of delivering the final results. This is always a trade off of using extra memory to achieve higher speed.

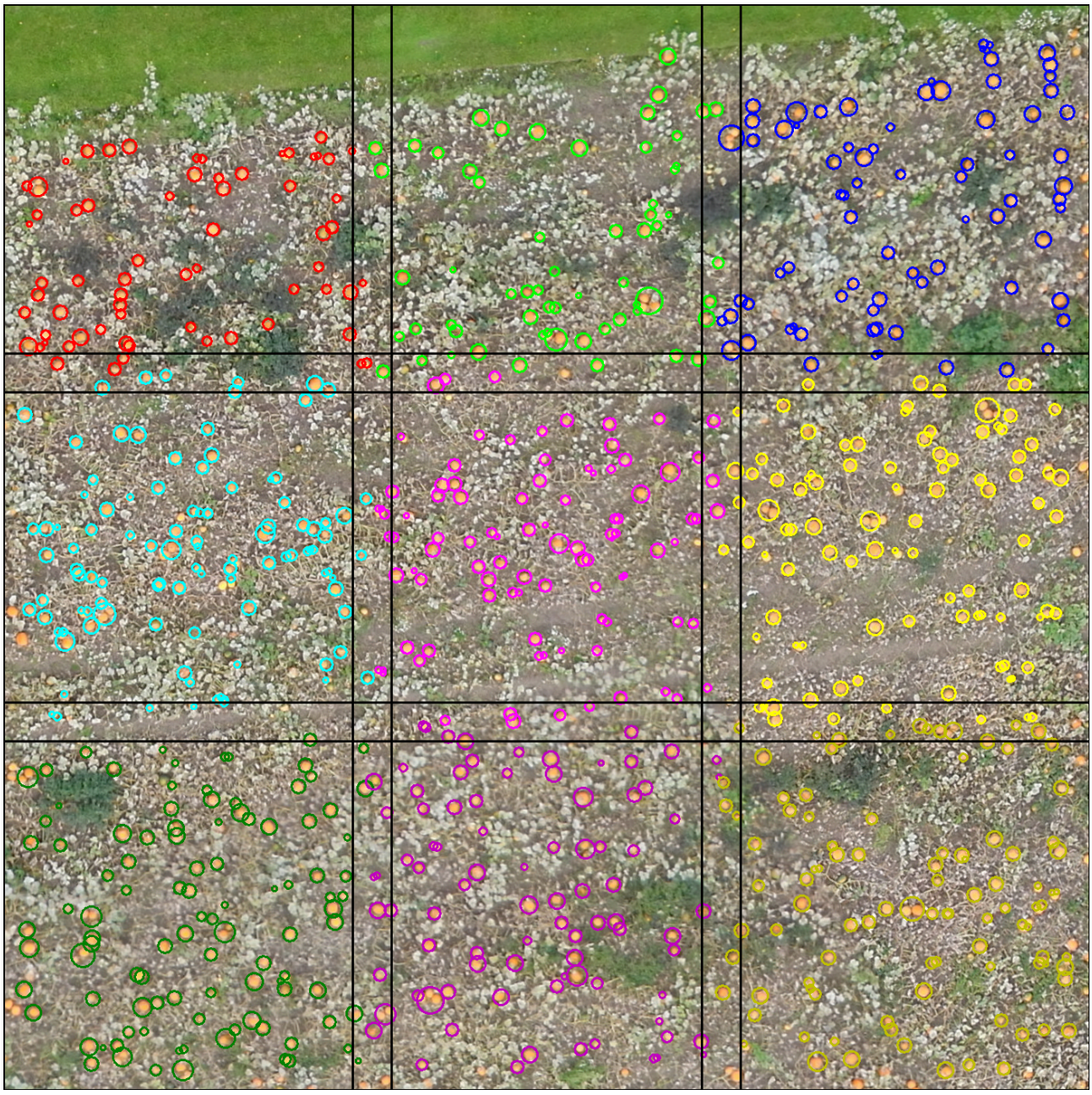
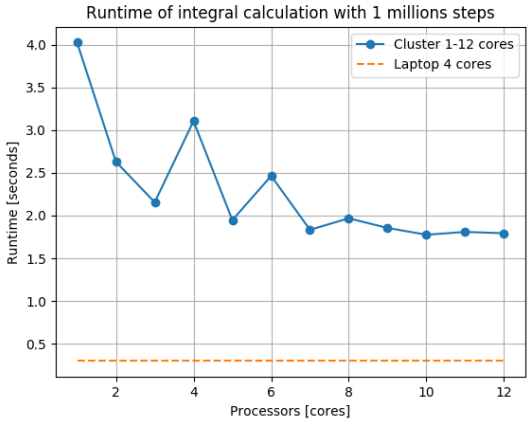


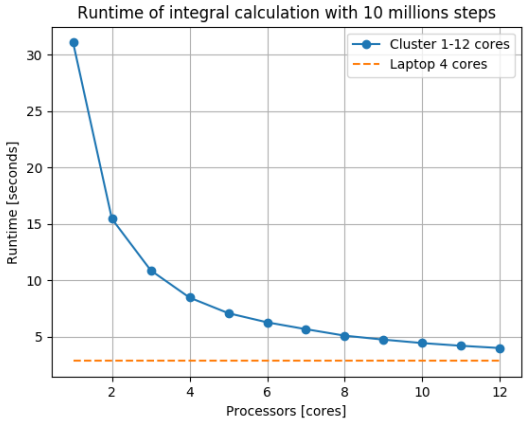
Figure 7: Illustration of the working of the pumpkin counting algorithm. The colors shows pumpkins detected in each block with an overlap to insure that pumpkins are only counted ones for each block. Each block has a size of 500×500 *pixels in this example*

To see the performance of the cluster when the nodes are not sharing big amount of data from the master, an integral calculation was performed for the function $f(x) = x^2$. Here the trapezoidal rule with end points $a = 0$ and $b = 1$ will be used. Giving the number of steps (n) to the function, an estimation of the integral can be found. As in the case of counting

pumpkins, a *cal_intergral.py* and *sub_mpi_integral.sh* have been created for the execution. Here each process will still get a clone of the program using OpenMPI and SLURM, but each process does not have to fetch data from the shared directory on the master besides the clone of the program. The performance of the Raspberry Pi cluster in regard to the laptop can be seen in Figure 8 and 9.

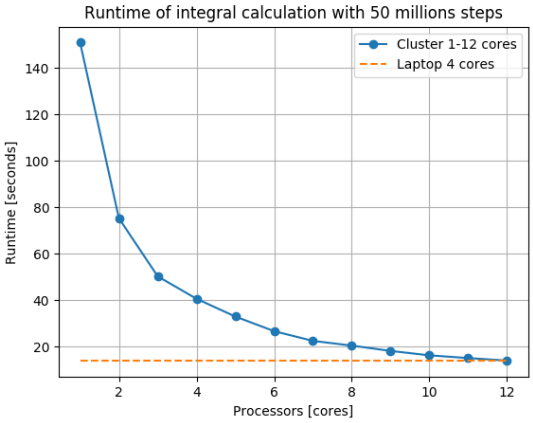


(a) Runtime of an integral with 1 million steps

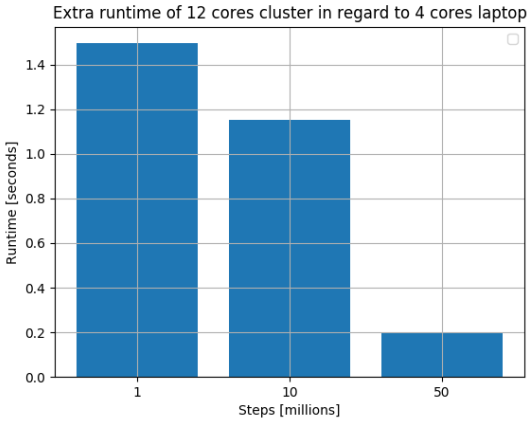


(b) Runtime of an integral with 10 million steps

Figure 8: The performance of a compute cluster in regard to a laptop



(a) Runtime of an integral with 50 million steps



(b) Runtime of an integral with 10 million steps

Figure 9: The performance of a compute cluster in regard to a laptop and the extra run time for the compute cluster summarized in a bar chart

As seen Figure 8a, when the amount of calculations for each node are low, a bad throughput will be achieved because of the bottleneck of communication between nodes in the cluster. However, when the amount of calculations for each node increases, the bottleneck will no longer be the communication and transportation of data between nodes, but the actual calculation of data. This will decrease the overall run time and exploit the multiple cores in the cluster more effectively. This can be seen in Figure 9a and highlighted in Figure 9b. Hence, when doing parallel programming, it is critical to avoid transportation of data as much as possible if this is the main reason for an increase in overall run time.

The results of the tests in the section are only an estimate, because background processes running on the laptop and Raspberry Pis could increase the run time by several milliseconds. However, multiple runs where made and they all yielded similar results. Thus, this gives an indication of the performance of the cluster which actually performs pretty well

compared to that of a laptop.

Only the scripts *cal_pumpkins.py* and *sub_mpi_pumpkins.sh* were briefly explained in this section. If one wants to have a closer look at the programming of the big data handling procedure and the code for counting pumpkins, it can be seen on GitHub [**kenni_github**].

5 Discussion

One of the main reasons for making a shared directory on the master node, was to enable the easy access of data between the nodes. That was quite beneficial when running large programs except for the increase in run time because of the large data transportation between nodes. Having a faster switch (Gigabit) would undoubtedly result in a decrease in run time (the switch used has 10/100 Mbps throughput).

Also the shared directory could be used to update the cluster with new nodes. Here a script could be created which would update the *slurm.conf* and *hosts* files at every startup. This bash script would have to be placed in the file *.bashrc* located in the home folder on the Raspberry Pi. This would effectively update the cluster with new nodes and avalible processing cores at startup. Besides of this, a script for installation of all files regarding SLURM, OpenMPI, Networking and NFS could be utilized with a setup of the change of hostname and password for a new Raspberry Pi could be done, so that the process adding a new node to the cluster could be fully automated. In this project, the installations have primarily been done using the SLURM command *srun* where all modules have been installed on each node in the cluster. Because only four nodes were used, this was not the big deal, but is does not scale to a cluster with many nodes. Therefore an automation of this would be desirable.

The automation could have been effectively accomplished using packer. Packer is a tool which lets you make changes to an OS. Packer makes use of *provisioners* with two types, namely *shell* and *inline*. Using *shell*, commands could have been written in a bash script, like adding packages or editing system configurations like network configurations, host name, hosts, password and more. These changes would then be incorporated in the image after build. Having knowledge about current number of nodes in the cluster, a specific image with an individual setup for each node could have been made, which would make maintenance of the cluster scalable [**packer**].

6 Conclusion

Tests showed that the run time of the calculations of counting pumpkins for a 23890×6009 pixel image split into 742 blocks (500×500 pixels each), which was run on a cluster with 12 cores, was approximately 110 seconds. A laptop with an Intel Core M-5Y10c CPU 0.80GHz $\times$ 4 with 8 GB RAM completed this in approximately 50 seconds. Because the nodes shared memory (the image) from the master, a bottleneck would be the transportation and communication over the network. However, the calculations of an integral, $f(x) = x^2$, using the trapezoidal rule with end points $a = 0$, $b = 1$ and steps of 50 millions split into blocks to be used in parallel processing in the cluster, yielded a run time of only 0.2 seconds more than that of the laptop. This concludes that the Raspberry Pi cluster with 12 compute cores performs pretty well, when transportation of large amounts of data between the nodes are not needed.

Hence, a Raspberry Pi cluster could be a good and cheap solution if parallel programming is possible for the given task.