



UNIVERSITI MALAYA

**FACULTY OF COMPUTER SCIENCE AND INFORMATION
TECHNOLOGY
MASTER IN DATA SCIENCE**

**WQD7008 PARALLEL AND DISTRIBUTED COMPUTING
GROUP 2**

GROUP ASSIGNMENT

**Astronomical Image Mosaic Engine Using Distributed Computing
Architecture**

THE RETURNED CEOs

Name	Matric Number
Chan Tzyy Yan	17153472
Choo En Ming	22079929
Low Boon Kiat	17138399
Yeoh Joer	22077700
Zhou Yao	S2177633

Contents

Contents	2
1.0 Introduction.....	3
1.1 Background	3
1.2 Project Objectives	3
2.0 Architectural Design	4
2.1 Resource Management and Task Scheduler	4
2.2 Distributed Data Management	4
2.3 Workflow Management Engine	5
3.0 Setup and Configuration	6
3.1 Resource Management and Task Scheduler	6
3.1.1 EC2 Instances.....	6
3.1.2 HTCondor	6
3.2 Distributed Data Management – Network File System (NFS)	9
3.3 Workflow Management Engine – HTCondor DAGMan.....	13
4.0 Use Case	19
4.1 Overview of Montage	19
4.2 Sequential Workflow of Montage.....	19
4.3 Parallelisation.....	22
4.3.1 Multi-bands Parallelisation (DAG 1)	22
4.3.2 Single-band Parallelisation (DAG 2)	23
5.0 Finding and Challenges	26
5.1 Findings.....	26
5.2 Challenges	31
6.0 Conclusion	33
7.0 Appendices.....	34
8.0 References.....	34

1.0 Introduction

1.1 Background

Distributed computing is the backbone of modern data processing. It handles immense volumes of information across interconnected systems rather than relying on a single centralised unit. The advantages of distributed computing are multifaceted. It facilitates parallel processing, enabling multiple tasks to run simultaneously. It offers fault tolerance. If one node fails, the system continues operating without compromising the data integrity. Scalability is another key feature as more nodes can be added to the system to accommodate growing data needs. However, distributed computing presents challenges. Coordinating and managing resources across diverse nodes can be complicated, whereas balancing the workload and optimizing resource utilization requires careful orchestration (Van Steen & Tanenbaum, 2017).

Components of distributed computing include resource management and task scheduling, distributed data management, and workflow management engines. Resource management involves allocating tasks to available nodes and ensuring optimal resource utilisation. Distributed data management deals with storing and accessing data across multiple nodes. Workflow management engines orchestrate the sequence of tasks and manage dependencies within a workflow. In an effective distributed computing system, precise alignment among these components is crucial to ensure that tasks are executed efficiently within the allocated resources and data availability constraints (Kshemkalyani & Singhal, 2011).

Montage is a scientific workflow for generating astronomical images mosaic. By integrating multiple images into a single mosaic, Montage enables astronomers to study various celestial objects such as galaxies and nebulae comprehensively. Montage workflow involves complex tasks such as image re-projection and background rectification with strong demand for extensive computational resources and efficient data handling. The intricate nature of Montage's workflow, involving large volume of astronomical images and critical task dependencies, underscores the necessity for distributed computing to execute these tasks efficiently (Berriman et al., 2008).

1.2 Project Objectives

The objectives of this project are as follows:

- To design a distributed data processing platform using Amazon Web Services (AWS), leveraging HTCondor as the resource management and task scheduler, NFS as the distributed data management system, and HTCondor DAGMan as the workflow management engine within the AWS environment.
- To employ Montage, a scientific workflow known for generating astronomical image mosaic, as the use case to enact on the established platform.
- To evaluate the performance of the established platform by executing Montage workflow under varying conditions and identify limitations of the deployed distributed data processing platform.

2.0 Architectural Design

2.1 Resource Management and Task Scheduler

Amazon Web Services (AWS) is a comprehensive cloud platform provided by Amazon that offers a variety of services, including compute power, storage options, and networking capabilities. Amazon Elastic Compute Cloud (EC2) is a part of AWS that provides resizable compute power in the cloud, allowing users to run virtual servers.

HTCondor, which stands for High Throughput Computing Condor, is a specialised workload management system for distributed computing environments. It is designed to efficiently harness and manage computational resources across a cluster or grid of machines, allowing users to submit and manage jobs that require significant computing power. It consists of two main components: the Condor host, which is responsible for task tracking and scheduling, and the worker nodes, which are responsible for executing tasks. It allows to run on EC2 instances in AWS, joining multiple EC2 instances as worker nodes, which improves computing performance and speeds up task completion time. HTCondor is particularly useful for applications that can be parallelised as it helps to optimise the computing resources by intelligently scheduling these tasks across available resources in the environment and hence reducing the operational costs.

In this study, these components implement an adaptive algorithm that analyses the computational load and performance metrics of each node within the AWS EC2 instances' environment. It dynamically distributes image processing tasks to nodes with available capacity, prioritising tasks based on their criticality, dependencies and resource requirements. It seamlessly adjusts to fluctuations in demand, ensuring that the processing continues at optimal efficiency without overloading any single node.

2.2 Distributed Data Management

Network File System (NFS) is a distributed file system protocol that allows sharing of files between different computers over a computer network. It operates on client-server architecture that includes a server that hosts the file system and one or more clients that access the files over the network. NFS is implemented at the network layer, so it can work between different operating systems.

In this study, NFS has been carefully selected as a solution for distributed data management, which performs critical tasks required by the Astronomical Image Mosaic Engine (AIME) on the powerful cloud infrastructure of AWS. In an astronomical image processing environment, NFS not only facilitates efficient file sharing, but also accelerates data access and improves operational efficiency through its centralised management. In addition, NFS supports data backup and recovery, saving data from accidental loss.

The NFS system in the project leverages the capabilities of the AWS distributed file system and incorporates data redundancy and failover mechanisms to ensure the integrity and reliability of data during transmission and storage. The use of smart caching significantly improves the speed of data processing and reduces latency due to data migration. These features of NFS not only optimise the location of data storage and ensure that data is stored as close as possible to the compute nodes, thus reducing data transfer time, but also lay the foundation for the improvement of the performance of the entire system.

2.3 Workflow Management Engine

HTCondor's Directed Acyclic Graph Manager (HTCondor Dagman) is a workflow management engine designed to handle and execute complex job workflows in a distributed computing environment. HTCondor itself is a high-throughput computing system, and Dagman enhances its capabilities by allowing users to specify and manage dependencies among jobs within a workflow. The engine ensures that tasks such as image alignment, calibration, and mosaicking are performed in the precise sequence necessary for accurate results.

Basically, users submit the DAG file to HTCondor along with the necessary job descriptions, input files, and any specific requirements for each job. Then Dagman uses the DAG file to determine the order in which jobs should be executed based on their dependencies. A job cannot start until all its dependencies have successfully completed. Dagman is able to exploit the parallelism inherent in the workflow by starting independent branches of the DAG in parallel if their dependencies allow for it which helps maximising resources utilisation and reduce overall workflow completion time. The engine also provides robust error handling and recovery processes, enabling it to quickly adapt to any issues that arise during the computation, such as an instance failure or a data corruption incident, thus maintaining the integrity of the workflow.

3.0 Setup and Configuration

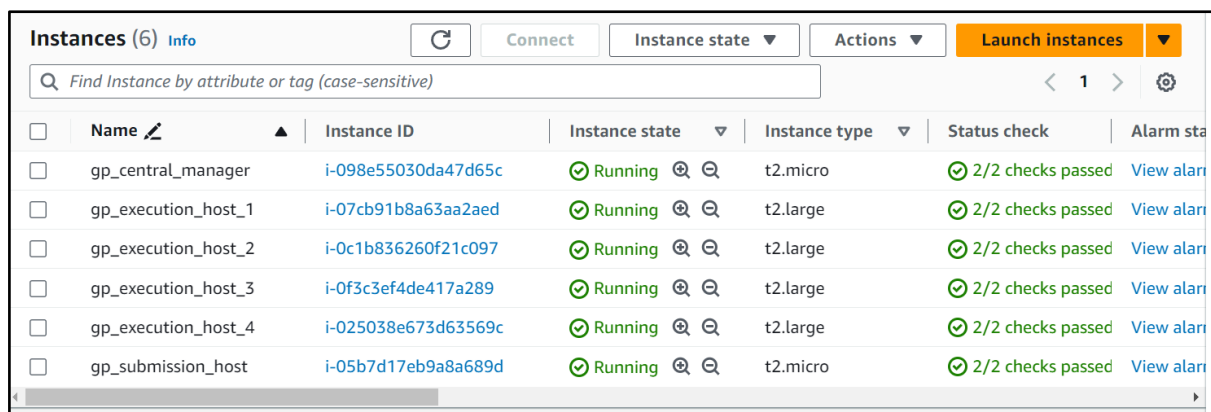
3.1 Resource Management and Task Scheduler

3.1.1 EC2 Instances

EC2 instances serve as the computing resources where tasks will be executed. Upon logging in to AWSConsole, navigate to the EC2 service and launch instance to create new EC2 instances for the HTCondor pool. In this project, “*Ubuntu Server 22.04 LTS (HVM), SSD Volume Type*” was chosen as the Amazon Machine Image (AMI) type, as it is free and open source as well as its large community support, which is valuable for troubleshooting, learning, and sharing knowledge. The instance configurations were set to default values except for the Security Group, where custom rules were defined. Table 1 provides a comprehensive summary of the security rules meticulously configured for different nodes within our distributed system.

Node	Inbound rules	Outbound rules
Central Manager	SSH (TCP) - 22 - 0.0.0.0/0	All - All - 0.0.0.0/0
Submission	SSH (TCP) - 22 - 0.0.0.0/0	All - All - 0.0.0.0/0
Execution	SSH (TCP) - 22 - 0.0.0.0/0	All - All - 0.0.0.0/0

Table 3.1: The Inbound and Outbound Security Rules for the Different Nodes



Name	Instance ID	Instance state	Instance type	Status check	Alarm status
gp_central_manager	i-098e55030da47d65c	Running	t2.micro	2/2 checks passed	View alarm
gp_execution_host_1	i-07cb91b8a63aa2aed	Running	t2.large	2/2 checks passed	View alarm
gp_execution_host_2	i-0c1b836260f21c097	Running	t2.large	2/2 checks passed	View alarm
gp_execution_host_3	i-0f3c3ef4de417a289	Running	t2.large	2/2 checks passed	View alarm
gp_execution_host_4	i-025038e673d63569c	Running	t2.large	2/2 checks passed	View alarm
gp_submission_host	i-05b7d17eb9a8a689d	Running	t2.micro	2/2 checks passed	View alarm

Figure 3.1: EC2 Instances Configuration

3.1.2 HTCondor

Upon accessing each EC2 instance using EC2 Instance Connect and update & upgrade on the package repositories, we then proceed to install and configure HTCondor.

```

- hosts: all
  become: yes
  tasks:
    - name: Add HTCondor's GPG key.
      get_url:
        url: https://research.cs.wisc.edu/htcondor/repo/keys/HTCondor-23.x-Key
        dest: /etc/apt/trusted.gpg.d/htcondor.asc
        force: true

    - name: Set up the HTCondor repository.
      apt_repository:
        repo: deb [arch=amd64] http://research.cs.wisc.edu/htcondor/repo/ubuntu/23.x jammy main
        state: present

    - name: Install HTCondor.
      apt:
        name: htcondor
        update_cache: yes
        state: present

    - name: Remove the default 9.0 security configuration.
      file:
        path: /etc/condor/config.d/00-htcondor-9.0.config
        state: absent

```

Figure 3.2: Ansible playbook for step 1 – 4

```

- name: Enable a shared file system.
  copy:
    src: ./config/02-domain.config
    dest: /etc/condor/config.d/02-domain.config
    owner: ubuntu
    group: ubuntu
    mode: 0755

- name: Create a credential.
  shell: condor_store_cred add -c -p 123

- name: Create a token.
  shell: condor_token_create -identity condor@172.31.18.184 > /etc/condor/tokens.d/condor@172.31.18.184

```

Figure 3.3: Ansible playbook for step 5 – 7

Figure 3.2 and Figure 3.3 provide a set of tasks for installing and configuring HTCondor on Ubuntu. Breakdown as below:

1. Add HTCondor's GNU Privacy Guard (GPG) key: Downloads the GPG key from HTCondor's repository and adds it to the trusted GPG keys.
2. Set up the HTCondor repository: Adds the HTCondor repository to the apt sources list.
3. Install HTCondor: Uses apt to install the HTCondor package.
4. Remove the default 9.0 security configuration: Deletes the default security configuration file for HTCondor 9.0.
5. Enable a shared file system: Copies a configuration file (02-domain.config) to enable a shared file system.
6. Create a credential: Uses the `condor_store_cred` command to add a credential with a password (-p 123).

7. Create a token: Uses the `condor_token_create` command to generate a token for a specific identity (`condor@172.31.18.184`) and saves it to a file.

While setting up for the HTCondor, it is essential for the team to ensure that security groups allow communication between EC2 instances on the HTCondor ports. If a firewall is active on the instances, team is to ensure that it allows traffic on the HTCondor ports. Last but not the least, team must confirm that EC2 instances can reach each other through their private IP addresses for efficient communication.

Below is configuration on the HTCondor Cluster. There are 6 nodes: 1 submission host, 1 central manager, and 4 execution host.

```
submission_host:
  hosts:
    server_1:
      ansible_host: 18.204.32.221
      ansible_user: ubuntu
      ansible_ssh_private_key_file: keys/gp.pem

central_manager:
  hosts:
    server_2:
      ansible_host: 18.211.126.128
      ansible_user: ubuntu
      ansible_ssh_private_key_file: keys/gp.pem

execution_host:
  hosts:
    server_3:
      ansible_host: 54.147.239.82
      ansible_user: ubuntu
      ansible_ssh_private_key_file: keys/gp.pem
    server_4:
      ansible_host: 54.161.255.100
      ansible_user: ubuntu
      ansible_ssh_private_key_file: keys/gp.pem
    server_5:
      ansible_host: 54.86.113.15
      ansible_user: ubuntu
      ansible_ssh_private_key_file: keys/gp.pem
    server_6:
      ansible_host: 3.84.41.153
      ansible_user: ubuntu
      ansible_ssh_private_key_file: keys/gp.pem

all:
  vars:
    nfs_server_ip: 172.31.28.6
```

Figure 3.4: An inventory.yml for HTCondor Cluster

```
ubuntu@ip-172-31-18-184:~$ condor_status
Name                               OpSys      Arch   State   Activity LoadAv Mem   ActvtyTime
slot1@ip-172-31-80-22.ec2.internal  LINUX      X86_64 Unclaimed Idle     0.000 7930 0+00:34:33
slot1@ip-172-31-91-183.ec2.internal  LINUX      X86_64 Unclaimed Idle     0.000 7930 0+00:34:33
slot1@ip-172-31-94-197.ec2.internal  LINUX      X86_64 Unclaimed Idle     0.000 7930 0+00:34:33
slot1@ip-172-31-95-69.ec2.internal  LINUX      X86_64 Unclaimed Idle     0.000 7930 0+00:34:37

Total Owner Claimed Unclaimed Matched Preempting Drain Backfill BkIdle
X86_64/LINUX      4      0      0      4      0      0      0      0      0
Total             4      0      0      4      0      0      0      0      0
ubuntu@ip-172-31-18-184:~$
```

Figure 3.5: Verification of the Presence of All Machines

3.2 Distributed Data Management – Network File System (NFS)

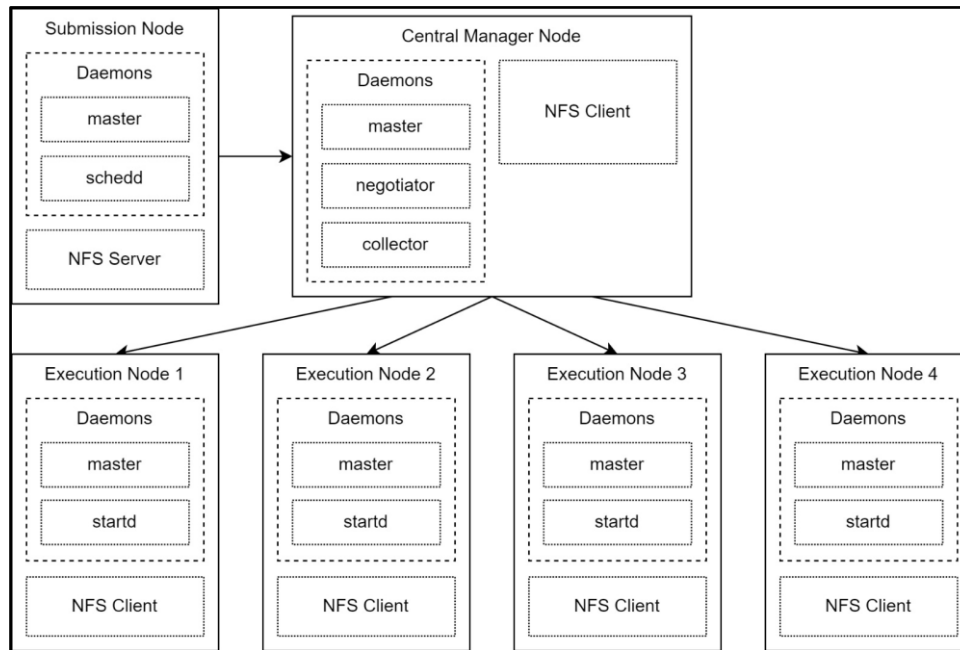


Figure 3.6: Architecture Diagram of NFS

The Network File System (NFS) architecture is centered around an NFS server that exports one or more directories from its local file system. This server runs the NFS daemon, which handles requests from NFS clients. The exported directories become accessible to NFS clients over the network. On the client side, machines run the NFS client daemon, interacting with the server's daemon to perform file operations. The communication between the server and clients occurs over the network, using the NFS protocol that defines the format of messages and requests for file operations. Clients mount the exported file system at specified mount points, presenting remote directories as if they were local.

Steps and configurations of the NFS are listed as below:

1. Install the NFS client package (nfs-common) on all hosts.
2. Downloads Montage binaries from a specific release on GitHub.
3. Creates a directory to store the Montage binaries.
4. Extracts Montage binaries from the downloaded tar.gz file.
5. Modifies the user's '.profile' file to include the Montage binaries in the system's PATH.

```

- name: Install NFS client.
  apt:
    name: nfs-common
    update_cache: yes
    state: present

- name: Download Montage binaries.
  get_url:
    url: https://github.com/yjoer/montage/releases/download/v6.0/montage_6.0_linux_amd64.tar.gz
    dest: /home/ubuntu/montage.tar.gz

- name: Create a directory if it does not exist.
  file:
    path: /home/ubuntu/montage
    owner: ubuntu
    group: ubuntu
    state: directory
    mode: 0755

- name: Extract Montage binaries.
  unarchive:
    src: /home/ubuntu/montage.tar.gz
    dest: /home/ubuntu/montage
    remote_src: yes

- name: Add Montage binaries to $PATH.
  lineinfile:
    path: /home/ubuntu/.profile
    line: "PATH=$PATH:/home/ubuntu/montage"

```

Figure 3.7: Common Tasking to Install NFS Client and Environment Set Up

Submission Host Tasks

1. Configure the Submission Host by copying the 01-submit.config file to the HTCondor configuration directory. Restarts HTCondor to load the new configuration.
2. Installs the NFS server package (nfs-kernel-server).
3. Configure the NFS Server by copying an exports configuration file to set up NFS exports. Creates a directory (/home/ubuntu/shared_data) to be shared via NFS. Applies the new NFS configuration.

```

# Submission host (incl. NFS server)
- hosts: submission_host
  become: yes
  tasks:
    - name: Configure the submission host.
      copy:
        src: ./config/01-submit.config
        dest: /etc/condor/config.d/01-submit.config
        owner: ubuntu
        group: ubuntu
        mode: 0755

    - name: Restart HTCondor to load the new configuration.
      service:
        name: condor
        state: restarted

    - name: Install NFS server.
      apt:
        name: nfs-kernel-server
        update_cache: yes
        state: present

    - name: Configure the NFS server.
      copy:
        src: ./config/exports
        dest: /etc/exports
        owner: ubuntu
        group: ubuntu
        mode: 0755

    - name: Create a directory if it does not exist.
      file:
        path: /home/ubuntu/shared_data
        owner: ubuntu
        group: ubuntu
        state: directory
        mode: 0755

```

Figure 3.8: Submission Host Tasks

```

CONDOR_HOST = 172.31.18.184

use role:get_htcondor_submit

```

Figure 3.9: Configuration of HTCondor Submission Host

Central Manager Tasks

1. Configure the Central Manager by copying the 01-central-manager.config file to the HTCondor configuration directory. Restarts HTCondor to load the new configuration.
2. Creates a directory (/home/ubuntu/shared_data) for mounting the NFS volume. Mounts the NFS volume from the specified server (nfs_server_ip).

```

# Central manager (incl. NFS client)
- hosts: central_manager
  become: yes
  tasks:
    - name: Configure the central manager.
      copy:
        src: ./config/01-central-manager.config
        dest: /etc/condor/config.d/01-central-manager.config
        owner: ubuntu
        group: ubuntu
        mode: 0755

    - name: Restart HTCondor to load the new configuration.
      service:
        name: condor
        state: restarted

    - name: Create a directory if it does not exist.
      file:
        path: /home/ubuntu/shared_data
        owner: ubuntu
        group: ubuntu
        state: directory
        mode: 0755

    - name: Mount an NFS volume.
      mount:
        src: "{{ nfs_server_ip }}/home/ubuntu/shared_data"
        path: /home/ubuntu/shared_data
        opts: rw, sync, hard
        state: mounted
        fstype: nfs

```

Figure 3.10: Central Manager Tasks

```

CONDOR_HOST = 172.31.18.184

use role: get_htcondor_central_manager

```

Figure 3.11: Configuration of HTCondor Central Manager

Execution Hosts Tasks

1. Configure the Execution Host by copying the 01-execute.config file to the HTCondor configuration directory. Restarts HTCondor to load the new configuration.
2. Creates a directory (/home/ubuntu/shared_data) for mounting the NFS volume. Mounts the NFS volume from the specified server (nfs_server_ip).

```

# Execution hosts (incl. NFS client)
- hosts: execution_host
  become: yes
  tasks:
    - name: Configure the execution host.
      copy:
        src: ./config/01-execute.config
        dest: /etc/condor/config.d/01-execute.config
        owner: ubuntu
        group: ubuntu
        mode: 0755

    - name: Restart HTCondor to load the new configuration.
      service:
        name: condor
        state: restarted

    - name: Create a directory if it does not exist.
      file:
        path: /home/ubuntu/shared_data
        owner: ubuntu
        group: ubuntu
        state: directory
        mode: 0755

    - name: Mount an NFS volume.
      mount:
        src: "{{ nfs_server_ip }}:/home/ubuntu/shared_data"
        path: /home/ubuntu/shared_data
        opts: rw, sync, hard
        state: mounted
        fstype: nfs

```

Figure 3.12: Execution Host Tasks

```

CONDOR_HOST = 172.31.18.184

use role: get_htcondor_execute

```

Figure 3.13: Configuration of HTCondor Execution Host

3.3 Workflow Management Engine – HTCondor DAGMan

HTCondor DAGMan is a powerful tool for managing and scheduling complex workflows on distributed computing resources. DAGMan is specifically designed for submitting and managing collections of dependent jobs, expressed as a Directed Acyclic Graph (DAG). Each node in the DAG represents a job, and the edges define the dependencies between jobs.

Below is an overview of the use of HTCondor DAGMan:

1. Define the Jobs and Dependencies:

```

executable = $ENV{HOME}/montage/mimgtbl
arguments = $(p)/images $(p)/images_parallel.tbl

initialdir = $(p)

output = 01_imgtbl.parallel.out
error = 01_imgtbl.parallel.err
log = 01_imgtbl.parallel.log

request_cpus = 1
request_memory = 3.5G

queue

```

Figure 3.14: Creation of Submission File (.sub file)

- **Job Definitions (*.sub files):** Create individual job submission files (.sub files) for each task in the workflow. These files typically contain information about the executable, arguments, input, output, and any specific requirements for the job.

```

JOB A_17k 01_imgtbl.sub
JOB B_17k 02_make_hdr.sub
JOB C_17k 03_proj_exec.sub
JOB D_17k 04_imgtbl.sub
JOB E_17k 05_overlaps.sub
JOB F_17k 06_diff_exec.sub
JOB G_17k 07_fit_exec.sub
JOB H_17k 08_bg_model.sub
JOB I_17k 09_bg_exec.sub
JOB J_17k 10_imgtbl.sub
JOB K_17k 11_add.sub
JOB L_17k 12_viewer.sub

VARS A_17k p="/home/ubuntu/shared_data/montage/M17/band_k"
VARS B_17k p="/home/ubuntu/shared_data/montage/M17/band_k"
VARS C_17k p="/home/ubuntu/shared_data/montage/M17/band_k"
VARS D_17k p="/home/ubuntu/shared_data/montage/M17/band_k"
VARS E_17k p="/home/ubuntu/shared_data/montage/M17/band_k"
VARS F_17k p="/home/ubuntu/shared_data/montage/M17/band_k"
VARS G_17k p="/home/ubuntu/shared_data/montage/M17/band_k"
VARS H_17k p="/home/ubuntu/shared_data/montage/M17/band_k"
VARS I_17k p="/home/ubuntu/shared_data/montage/M17/band_k"
VARS J_17k p="/home/ubuntu/shared_data/montage/M17/band_k"
VARS K_17k p="/home/ubuntu/shared_data/montage/M17/band_k"
VARS L_17k p="/home/ubuntu/shared_data/montage/M17/band_k"

```

Figure 3.15: Creation of DAG File

- **DAG File (dagfile.dag):** Create a DAG file that specifies the relationships between the jobs. The DAG file uses keywords like JOB, PARENT, and CHILD to define jobs and dependencies.
2. **Write the DAG File:** Use a text editor to create a DAG file. In this file, jobs and their dependencies are defined.
 3. **Submit the DAG to DAGMan:** Use the `condor_submit_dag` command to submit DAG to DAGMan. DAGMan will read the DAG file, parse dependencies, and submit jobs accordingly.

4. Monitor and Manage the Workflow: Use `condor_q` to check the status of DAG. Use `condor_rm` to remove the entire DAG and associated jobs if needed:
5. Logging and Debugging: HTCondor will generate log files for each job in the DAG, which can be helpful for debugging and monitoring the workflow.

Defined DAG for a Set of Jobs in HTCondor

```
JOB A_17k 01_imgtbl.sub
JOB B_17k 02_make_hdr.sub
JOB C_17k 03_proj_exec.sub
JOB D_17k 04_imgtbl.sub
JOB E_17k 05_overlaps.sub
JOB F_17k 06_diff_exec.sub
JOB G_17k 07_fit_exec.sub
JOB H_17k 08_bg_model.sub
JOB I_17k 09_bg_exec.sub
JOB J_17k 10_imgtbl.sub
JOB K_17k 11_add.sub
JOB L_17k 12_viewer.sub

VARS A_17k p="/home/ubuntu/shared_data/montage/M17/band_k"
VARS B_17k p="/home/ubuntu/shared_data/montage/M17/band_k"
VARS C_17k p="/home/ubuntu/shared_data/montage/M17/band_k"
VARS D_17k p="/home/ubuntu/shared_data/montage/M17/band_k"
VARS E_17k p="/home/ubuntu/shared_data/montage/M17/band_k"
VARS F_17k p="/home/ubuntu/shared_data/montage/M17/band_k"
VARS G_17k p="/home/ubuntu/shared_data/montage/M17/band_k"
VARS H_17k p="/home/ubuntu/shared_data/montage/M17/band_k"
VARS I_17k p="/home/ubuntu/shared_data/montage/M17/band_k"
VARS J_17k p="/home/ubuntu/shared_data/montage/M17/band_k"
VARS K_17k p="/home/ubuntu/shared_data/montage/M17/band_k"
VARS L_17k p="/home/ubuntu/shared_data/montage/M17/band_k"

PARENT A_17k CHILD B_17k
PARENT B_17k CHILD C_17k
PARENT C_17k CHILD D_17k
PARENT D_17k CHILD E_17k
PARENT E_17k CHILD F_17k
PARENT F_17k CHILD G_17k
PARENT G_17k CHILD H_17k
PARENT H_17k CHILD I_17k
PARENT I_17k CHILD J_17k
PARENT J_17k CHILD K_17k
PARENT K_17k CHILD L_17k
```

Figure 3.16: Sequential Workflow (Sequentially run of 12 tasks for a band)

DAG for Three Different Datasets (M8, M15, M17) with Three Bands Each (J, H, K)

The DAG below representing workflows for processing and merging images from three different astronomical datasets: M8, M15, and M17. Each dataset consists of three bands (J, H, K), and the workflows involve a sequence of jobs for image processing. Each dataset (M8, M15, M17) has a similar structure but operates independently. The DAGs are repetitive but customised for each dataset and band.

Each band within a dataset undergoes a parallel processing workflow. For instance, jobs A, B, C, D, etc., are processed concurrently for bands J, H, and K. There's a clear dependency structure among jobs, where certain jobs depend on the completion of others. For example, HDR creation (B jobs) depends on image table generation (A jobs), and so on.

```

# M8

## Band J

JOB_A_8j 01_imgtbl.sub
JOB_B_8j 02_make_hdr.sub
JOB_C_8j 03_proj_exec.sub
JOB_D_8j 04_imgtbl.sub
JOB_E_8j 05_overlaps.sub
JOB_F_8j 06_diff_exec.sub
JOB_G_8j 07_fit_exec.sub
JOB_H_8j 08_bg_model.sub
JOB_I_8j 09_bg_exec.sub
JOB_J_8j 10_imgtbl.sub
JOB_K_8j 11_add.sub
JOB_L_8j 12_viewer.sub

VARS_A_8j p="/home/ubuntu/shared_data/montage/M8/band_j"
VARS_B_8j p="/home/ubuntu/shared_data/montage/M8/band_j"
VARS_C_8j p="/home/ubuntu/shared_data/montage/M8/band_j"
VARS_D_8j p="/home/ubuntu/shared_data/montage/M8/band_j"
VARS_E_8j p="/home/ubuntu/shared_data/montage/M8/band_j"
VARS_F_8j p="/home/ubuntu/shared_data/montage/M8/band_j"
VARS_G_8j p="/home/ubuntu/shared_data/montage/M8/band_j"
VARS_H_8j p="/home/ubuntu/shared_data/montage/M8/band_j"
VARS_I_8j p="/home/ubuntu/shared_data/montage/M8/band_j"
VARS_J_8j p="/home/ubuntu/shared_data/montage/M8/band_j"
VARS_K_8j p="/home/ubuntu/shared_data/montage/M8/band_j"
VARS_L_8j p="/home/ubuntu/shared_data/montage/M8/band_j"

## Band H

JOB_A_8h 01_imgtbl.sub
JOB_B_8h 02_make_hdr.sub
JOB_C_8h 03_proj_exec.sub

```

Figure 3.17: DAG 1 (Parallel run of 111 tasks for 9 bands on 8 cores)

Defined DAG with Parallel Structure

The jobs are organised in a hierarchical manner with dependencies and variables for parallel execution. Parallel Jobs (C1_17k to J8_17k) are executed in parallel, and each has its own split and total_splits values to handle the parallelisation.


```

JOB A_17k parallel/01_imgtbl.sub
JOB B_17k parallel/02_make_hdr.sub

JOB C1_17k parallel/03p_proj_pp.sub
JOB C2_17k parallel/03p_proj_pp.sub
JOB C3_17k parallel/03p_proj_pp.sub
JOB C4_17k parallel/03p_proj_pp.sub
JOB C5_17k parallel/03p_proj_pp.sub
JOB C6_17k parallel/03p_proj_pp.sub
JOB C7_17k parallel/03p_proj_pp.sub
JOB C8_17k parallel/03p_proj_pp.sub

JOB D_17k parallel/04_imgtbl.sub
JOB E_17k parallel/05_overlaps.sub

JOB F1_17k parallel/06p_diff.sub
JOB F2_17k parallel/06p_diff.sub
JOB F3_17k parallel/06p_diff.sub
JOB F4_17k parallel/06p_diff.sub
JOB F5_17k parallel/06p_diff.sub
JOB F6_17k parallel/06p_diff.sub
JOB F7_17k parallel/06p_diff.sub
JOB F8_17k parallel/06p_diff.sub

JOB G1_17k parallel/07p_fitplane.sub
JOB G2_17k parallel/07p_fitplane.sub

```

Figure 3.18: DAG 2 (Parallel run of 42 tasks for 1 band on 8 cores)

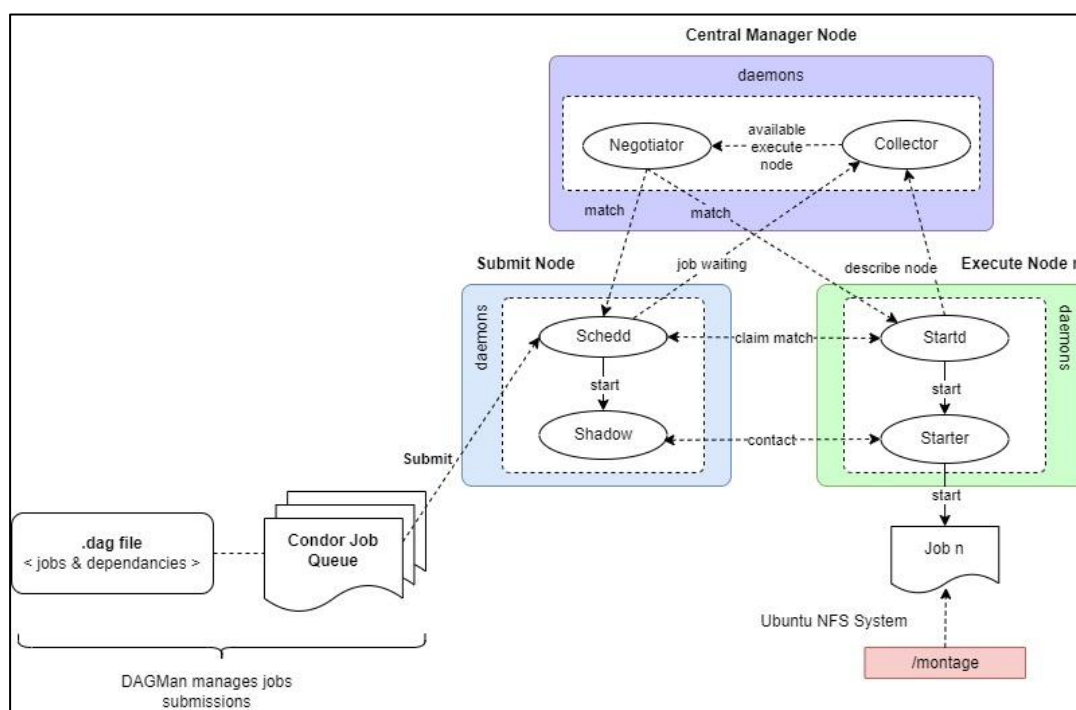


Figure 3.19: Workflow of Cluster Management System

Figure 3.19 represents a cluster management system, specifically the architecture of the HTCondor high-throughput computing system when job is submitted for execution. DAGMan, the workflow management engine that we selected will send a file *.dag file that describes jobs and their dependencies to the Condor Job Queue at the right time. Condor Job Queue holds the jobs that are waiting to be scheduled and run on the execution nodes, where the schedd daemon will initiate a message to the collector of the Central Manager that it has a job waiting. Shadow represents a user's job and handles all communication between the submitted job and the system until the job is completed. The negotiator asks the collector for a list of available execute nodes and matches available resources with submitted jobs based on a configured policy. The schedd contacts the startd to claim the match and schedd starts shadow daemon to monitor the job and startd starts starter daemon to initiate the execution of a job when the node has been matched with a job by the central manager. The shadow and starter contact each other.

4.0 Use Case

4.1 Overview of Montage

In astronomy, a montage refers to the process of creating a mosaic or composite image by combining multiple astronomical images of the same region of the sky taken with a telescope or other imaging instruments. Astronomers often capture images of celestial objects, such as galaxies, nebulae, star clusters, and other astronomical phenomena. The images might be taken using different filters, exposure times, or wavelengths to highlight various aspects of celestial objects (Amy, 2010).

A montage is created by precisely aligning and combining these individual images to form a seamless composite. This process involves precise alignment and registration of the images, considering factors like positions, orientations, and scales of each image, to ensure that the final mosaic accurately represents the specific region of the sky. During the creation of a mosaic, efforts are made to minimise artifacts or inconsistencies that can arise from combining multiple images. These artifacts could include seams between images, variations in background levels, or mismatches in colour or intensity (Berriman et al., 2008).

The primary goal of a montage is to provide a comprehensive view of a specific region of the sky, allowing astronomers to study and analyse celestial objects in greater detail, revealing structures, patterns, and characteristics that might not be readily apparent in single images. This technique helps in creating high-resolution and comprehensive views of astronomical phenomena, aiding astronomers in their research and exploration of the universe.

4.2 Sequential Workflow of Montage

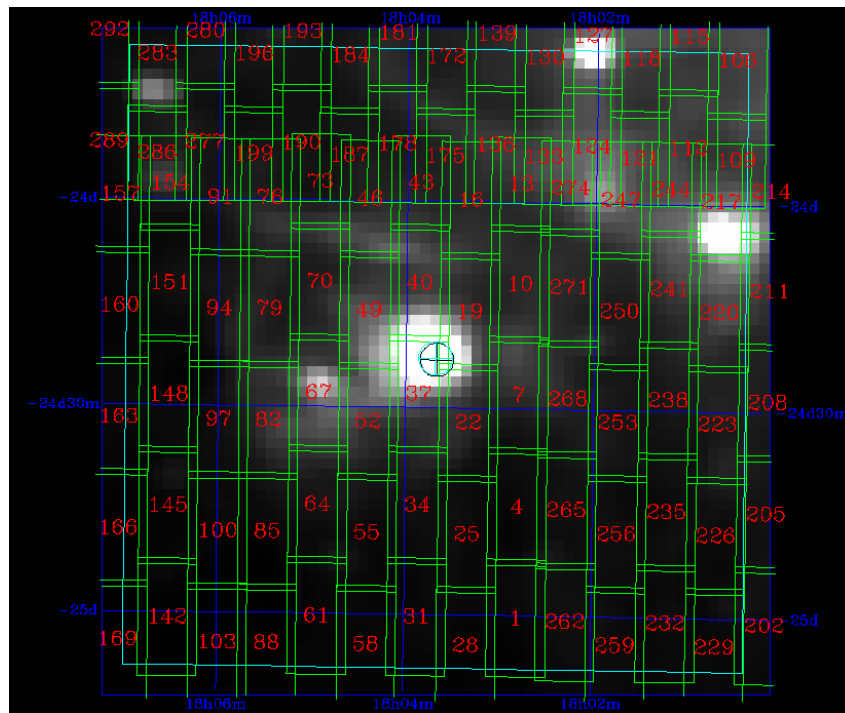


Figure 4.1: M8 images obtained from NASA/IPAC Infrared Science Archive.

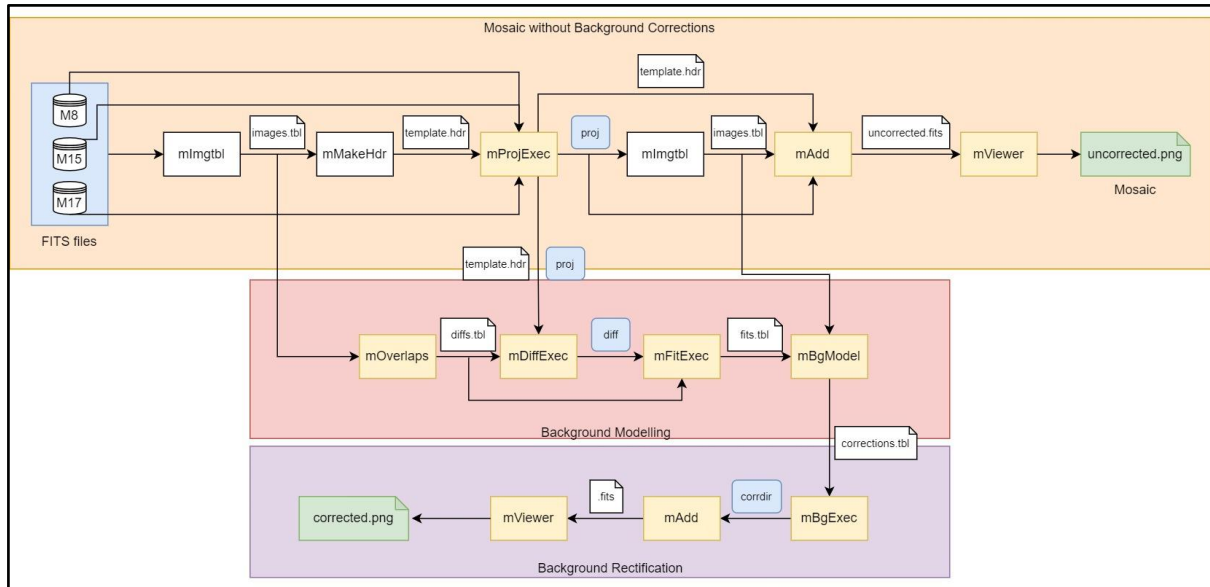


Figure 4.2: Sequential workflow of Montage

Building an astronomical image mosaic using the Montage software involves several steps. Figure 4.2 shows the workflow diagram of Montage in general, which consists of three main stages, namely mosaic without background corrections, followed by background modelling and rectification. Belows are the general procedures outlining the process (*Montage - Image Mosaic Software for Astronomers*, n.d.).

1. Installation of Montage

- Download the Montage source code tar file. Extract the contents of the downloaded file.

```
wget http://montage.ipac.caltech.edu/download/Montage_v6.0.tar.gz
```

```
tar -zxvf Montage_v6.0.tar.gz
```
- Move to the directory where the extracted files are located.

```
cd Montage_v6.0
```
- Compile the Montage software for installation.

```
./configure
```

```
make
```

2. Data Preparation

- Gather the individual astronomical images that will be assembled into a mosaic. In this project, we will use M8, M15 and M17 images obtained from NASA/IPAC Infrared Science Archive (*2MASS Batch Image Service Results*, n.d.). Download M8/M15/M17.tar files.

```
wget https://github.com/yjoer/montage/releases/download/v6.0/M8.tar
```

```
wget https://github.com/yjoer/montage/releases/download/v6.0/M15.tar
```

```
wget https://github.com/yjoer/montage/releases/download/v6.0/M17.tar
```
- Move them into M8/M15/M17 directories.

```
mkdir -p M8 M15 M17
```

```
mv M8.tar M8/
```

```
mv M15.tar M15/
```

```
mv M17.tar M17/
```
- Rename files to images.tar in each directory.

```
mv M8/M8.tar M8/images.tar
```

```
mv M15/M15.tar M15/images.tar
mv M17/M17.tar M17/images.tar
```

- Split images of J, H, and K bands into three directories. The process of splitting an image into three bands refers to dividing the image data into three distinct wavelength bands using specialised filters. The designations ‘J’, ‘H’ and ‘K’ refer to near-infrared astronomical filters used in astronomical observations. In infrared astronomy, filters are optical devices used to isolate specific wavelengths within the infrared portion of electromagnetic spectrum. These filters allow astronomers to selectively observe certain infrared wavelengths while blocking others, enabling them to study certain features of celestial objects (Alan T. Tokunaga, 2001).

```
python3 split.py
```

Here is a breakdown of how “split.py” script works.

- Extract the content of “images.tar” file into a directory called “images”.
 - Create three target directories (band_j, band_h, and band_k) to store the split images for J, H, and K bands respectively.
 - Iterate through files in the "images" directory. For those files with “.fits.gz” extension, decompress each file and determines its wavelength band based on filename prefixes ("aJ," "aH," "aK").
 - Copy the decompressed file into the respective target directory (band_j, band_h, and band_k) based on its determined wavelength band.
- Create directories for intermediate results under M8/M15/M17 (3 directories for each band and each astronomical images, with a total of 27 directories created).

```
cd M15
mkdir band_j/proj band_j/diff band_j/corr
mkdir band_h/proj band_h/diff band_h/corr
mkdir band_k/proj band_k/diff band_k/corr
```

Step 3 to step 7 uses M15, J-band images as example to illustrate the sequential workflow to create a mosaic. These steps will be repeated for each channel (J/H/K bands for M8 and M17 images as well).

3. Metadata Generation

- Move to desired directory.

```
cd M15/band_j
```
- Create a list of the image files and their metadata. This tool will examine the headers of the FITS files and generate a table of their information.

```
mImgtbl images images.tbl
```
- Process the image metadata table to generate the ‘template.hdr’ header file, conforming to the FITS/WCS convention. This file precisely delineates the mosaic's spatial footprint on the sky and will serve as the FITS header for the resulting mosaic.

```
mMakeHdr images.tbl template.hdr
```

4. Reprojection

- Reproject each individual image to a common spatial projection. This step transforms the images onto a common pixel grid.

```
mProjExec -p images images.tbl template.hdr proj stats.tbl
```

- Create a new metadata table that contains the updated header information in the reprojected files.

```
mImgtbl proj/ images_p.tbl
```

5. Overlap Correction

- Identify any overlaps between the reprojected images.
mOverlaps images_p.tbl diffs.tbl
- Subtract each pair of overlapping images and create a set of difference images in the diff subdirectory.

```
mDiffExec -p proj/ diffs.tbl template.hdr diff
```

- Calculate plane-fitting coefficients for each difference image.
mFitExec diffs.tbl fits.tbl diff
- Create a table of corrections that needs to be applied to each image.
mBgModel images_p.tbl fits.tbl corrections.tbl

6. Background Matching

- Apply the background removal to the original reprojected images. This step helps ensure uniformity in the background across the mosaic, reducing seams or variations in intensity.

```
mBgExec -p proj/ images_p.tbl corrections.tbl corr
```

7. Coadding and Mosaic Creation

- Combine the reprojected, corrected images into a single composite mosaic.

```
mAdd -p corr/ images_p.tbl template.hdr m15.fits
```

- Create a PNG file of the mosaic for visualization.

```
mViewer -ct 1 -gray m15.fits -ls max gaussian-log -out m15.png
```

8. Creation of Coloured Mosaic

- Combine three separate FITS files representing J/H/K bands of M15 to generate a coloured composite image (i.e. m15.png). Repeat this step for M8 and M17.

```
mViewer -t 1 \
  -red band_k/m15.fits 0s max gaussian-log \
  -green band_h/m15.fits 0s max gaussian-log \
  -blue band_j/m15.fits 0s max gaussian-log \
  -out m15.png
```

4.3 Parallelisation

4.3.1 Multi-bands Parallelisation (DAG 1)

Figure 4.3 below shows the directed acyclic graph (DAG 1) workflow that illustrates the multi-bands parallelisation in Montage. This parallelisation involves processing multiple sets of images, such as M8, M15, and M17, each consists of distinct near-infrared bands (J, H, K). These sets of images undergo a systematic workflow where individual bands are split, processed sequentially in parallel, and eventually combined to produce coloured composite mosaics. The process begins by dividing the multi-band images into their constituent J, H, and K bands. Subsequently, Montage orchestrates a sequential workflow independently for each band across the three image sets concurrently. These sequential tasks within each band include operations such as metadata preparation, re-projection, image differencing, fitting planes, background modelling, and co-addition, among others. Upon completion of the sequential

workflow for each band in the M8, M15, and M17 sets, Montage produces three separate FITS files for J, H, and K bands corresponding to each image set. The final step involves combining the three separate FITS files representing J, H, and K bands to generate a coloured composite mosaic for each image set. The final output comprises three mosaic images, each originating from the M8, M15, and M17 sets of images, respectively. The composite image provides a comprehensive and detailed view of the celestial objects M8, M15, and M17, highlighting different features and characteristics across various near-infrared wavelengths.

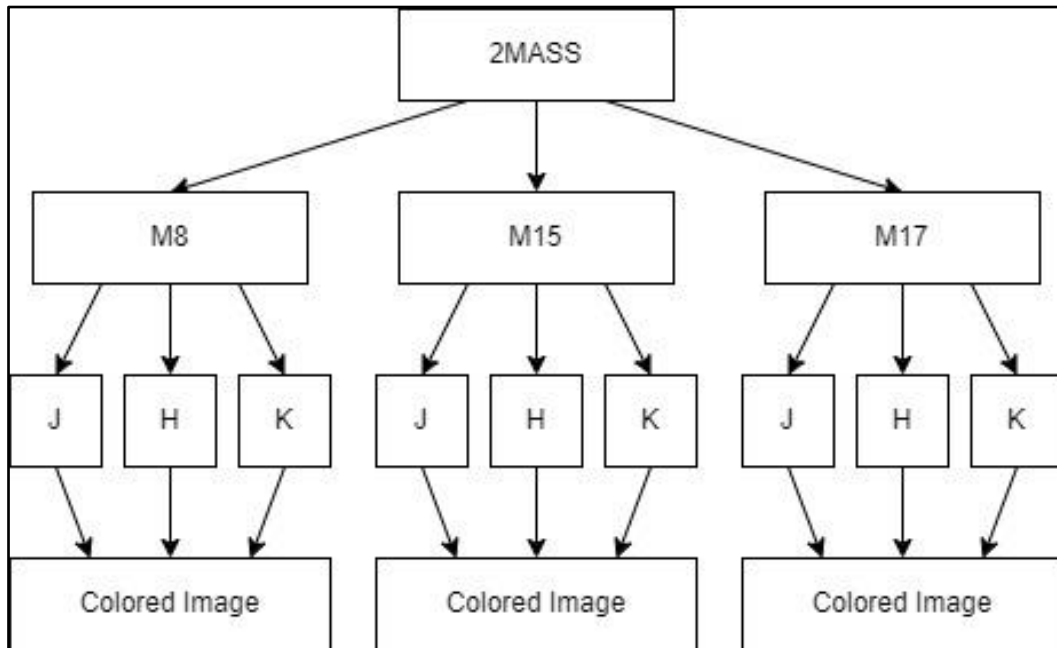


Figure 4.3: DAG1 workflow

4.3.2 Single-band Parallelisation (DAG 2)

Single-band parallelisation in Montage involves the processing a specific set of images, M17 k-band images in the context, by splitting the FITS files into multiple segments and executing multiple instances of the same program concurrently. The initial step involves splitting the FITS files of M17 k-band images into smaller segments. In this case, the files are divided into 8 splits to facilitate concurrent processing. A Python driver program is employed to manage the parallel execution. The driver program initiates the execution of the same program 8 times concurrently, each instance handling a separate split of the FITS files. The concurrent execution of the program across 8 segments allows for parallel processing of the M17 k-band images. Each instance of the program processes its designated split of the FITS files independently and concurrently. The final output comprises a single greyscale mosaic derived from the M17 k-band images. The individual segments processed concurrently are combined into a unified mosaic image, consolidating the information captured in the M17 k-band dataset. The single-band parallelisation approach in Montage optimises the processing of a specific set of images, M17 k-band in this case, by concurrently executing multiple instances of a program to handle segmented FITS files. This strategy enhances efficiency and allows for faster processing of large dataset.

Montage offers flexibility to enhance performance, particularly in speeding up the re-projection process, because the general re-projection algorithm ‘mProject’ tends to be slow in execution. To expedite the re-projection process, one approach involves utilising the optimised, faster re-projection algorithm ‘mProjectPP’ that sacrifice a small degree of accuracy for significantly improved computational efficiency. Another approach is to exploit parallelisation in the design of Montage workflow. The only step in the workflow that cannot be parallelised is the ‘mBgModel’ module, as it runs after the completion of image re-projections, differencing, and plane fitting (Berriman *et al.*, 2004; Katz *et al.*, 2005). Figure 4.4 below presents a Directed Acyclic Graph (DAG 2) workflow that illustrates the single-band parallelisation for generating a mosaic from M17 k-band images. DAG 2 outlines how the image processing tasks in Montage are interconnected through parallel processing to optimise the overall performance. The parallelised tasks involved in DAG 2 include ‘mProjectPP’, ‘mDiff’, ‘mFitplane’, and ‘mBackground’. Each step is dependent on the completion of prior tasks. In addition to the set of procedures listed in Section 4.2, DAG 2 involves an additional step ‘cat’ to concatenate the fitted planes after executing ‘mFitplane’ on segmented portions of the images in parallel.

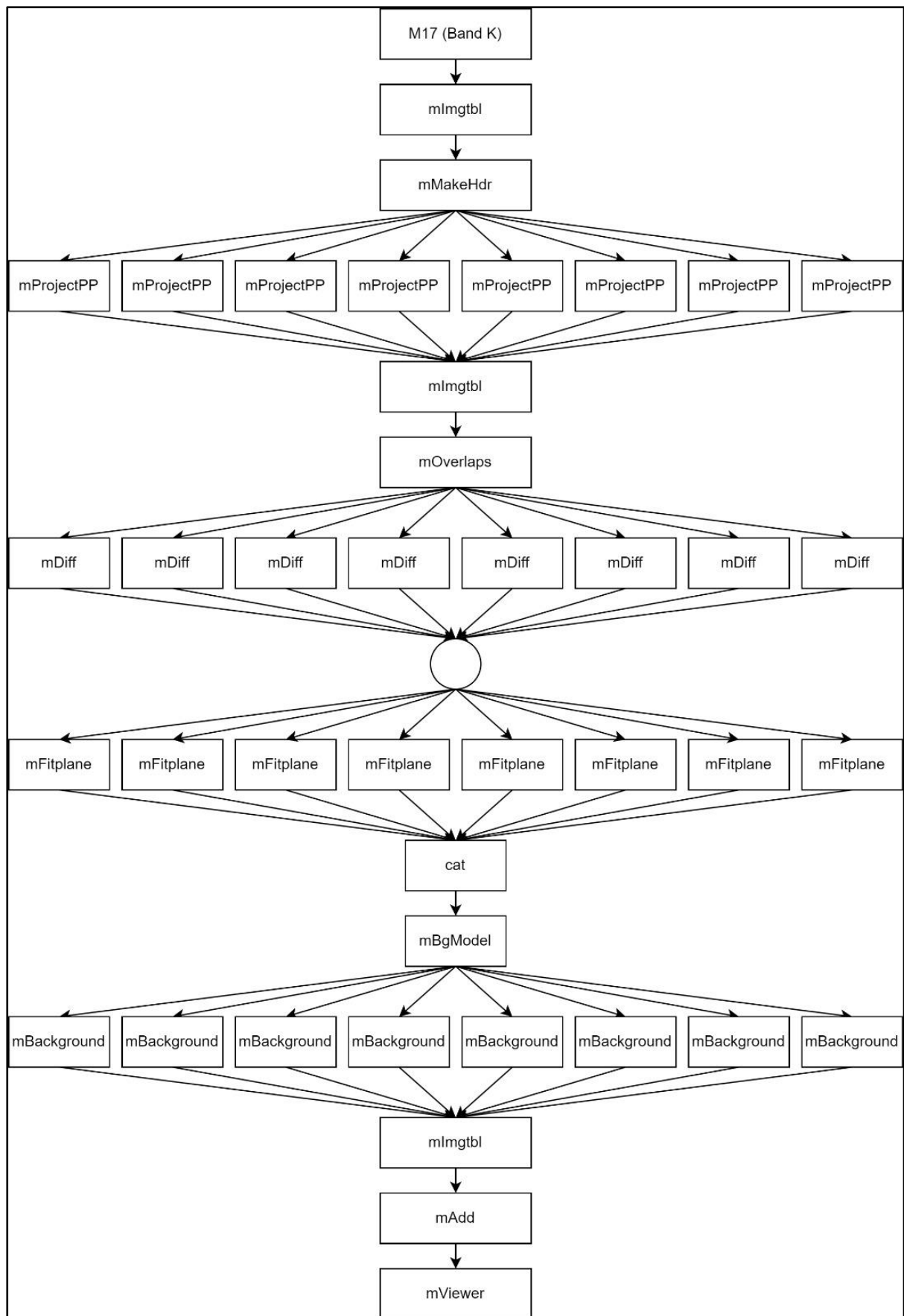


Figure 4.4: Directed Acyclic Graph (DAG 2) workflow for M17 k-band images.

5.0 Finding and Challenges

5.1 Findings

The incorporation of parallelisation strategies within Montage demonstrates a profound impact on the efficiency of creating astronomical mosaics. The multi-bands parallelisation approach (DAG 1) enables the simultaneous processing of distinct near-infrared bands (J, H, K) for multiple sets of images (M8, M15, M17). As noted in Step 3 until Step 7 abovementioned in Section 4.2, there are 12 jobs per band. With a total of 9 bands (3 images x 3 bands) available, the number of nodes totals up to 108 nodes, in addition to 3 tasks to combine 3 FITS files from each image set to generate a coloured composite mosaic for each image set. An analysis of the speed-up achieved in DAG 1 execution provides valuable insights. Running 111 tasks for 9 bands on 8 cores took 2099 seconds, showcasing a commendable speed-up of 4.61x compared to running 12 tasks sequentially for a band, which took 1077 seconds. The remarkable speed-up in DAG 1 showcases Montage's prowess in harnessing parallel computing resources to enhance the creation of intricate mosaics across distinct near-infrared bands for multiple celestial images.

The single-band parallelisation approach (DAG 2) focuses on optimising the processing of a specific set of images, exemplified by the M17 k-band images. This approach involves dividing the FITS files of M17 k-band images into smaller segments, and concurrently executing multiple instances of a program across segmented FITS files. An analysis of the speed-up achieved in DAG 2 execution provides valuable insights. Running 42 tasks for a single band on 8 cores took 546 seconds, showcasing a notable speed-up of 1.97x compared to running 12 tasks sequentially for a band, which took 1077 seconds. This is nearly two-fold speed-up in DAG 2 not only underscores its high efficiency in optimising the processing of specific image datasets, but also exemplifies Montage's flexibility to meet diverse processing demands, making it a versatile tool for astronomers with varying computational requirements.

Table 5.1: Performance comparison between sequential, DAG 1 and DAG 2 workflow.

Workflow	No. of Jobs in the Workflow	Duration of Workflow Execution (s)	Observed Speed-up
Sequential (Sequentially run 12 tasks for 1 band)	12 (1 band)	1077	-
DAG 1 (Parallel run 111 tasks for 9 bands on 8 cores)	111 (9 bands)	2099	$\frac{9 * 1077}{2099} = 4.61 \times$
DAG 2 (Parallel run 42 tasks for 1 band on 8 cores)	42 (1 band)	546	$\frac{1077}{546} = 1.97 \times$

Table 5.2: Performance of DAG 1 and DAG 2 workflow at varying number of cores.

Workflow	No. of Cores	Duration of Workflow Execution (s)	Observed Speed-up	Overhead
DAG 1 (Parallel run 111 tasks for 9 bands)	2	6158	$\frac{9 * 1077}{6158} = 1.57 \times$	$\frac{2 - 1.57}{2} = 21\%$
	3	4517	$\frac{9 * 1077}{4517} = 2.15 \times$	$\frac{3 - 2.15}{3} = 28\%$
	4	3696	$\frac{9 * 1077}{3696} = 2.62 \times$	$\frac{4 - 2.62}{4} = 34\%$
	6	2790	$\frac{9 * 1077}{2790} = 3.47 \times$	$\frac{6 - 3.47}{6} = 42\%$
	8	2099	$\frac{9 * 1077}{2099} = 4.62 \times$	$\frac{8 - 4.62}{8} = 42\%$
DAG 2 (Parallel run 42 tasks for 1 band)	2	957	$\frac{1077}{957} = 1.13 \times$	$\frac{2 - 1.13}{2} = 44\%$
	4	687	$\frac{1077}{687} = 1.57 \times$	$\frac{4 - 1.57}{4} = 61\%$
	6	672	$\frac{1077}{672} = 1.60 \times$	$\frac{6 - 1.60}{6} = 73\%$
	8	546	$\frac{1077}{546} = 1.97 \times$	$\frac{8 - 1.97}{8} = 75\%$

The performance evaluation of two DAGs reveals intriguing insights into the impact of varying number of cores on workflow execution. In DAG 1, the observed speed-up increases from 1.57x to 4.62x as the number of cores rises from 2 to 8. However, this acceleration is accompanied by a growing overhead from 21% to 42%. The diminishing returns is evident, indicating that the benefits of increased parallelisation are offset by the associated overhead. The increasing overhead suggests that there is a non-negligible portion of the workflow that remains unaffected by parallelisation. This is a clear manifestation of Amdahl's law, which states that the speed-up of a parallelized task is limited by the sequential portion of the workload.

Similarly, in DAG 2, the observed speed-up increases from 1.13x to 1.97x as the number of cores rises from 2 to 8. Meanwhile, the overhead steadily rises to 75% with 8 cores. The increasing overhead underscores the diminishing returns associated with parallelisation. This trend aligns with Amdahl's law, which states that as the parallelised portion of the workload increases, the overall speed-up becomes more constrained by the sequential components. The diminishing return in speed-up as the number of cores increases underscores the challenges of optimising workflows with complex task dependencies, emphasising the importance of carefully selecting the number of cores to balance efficiency and overhead.

In both DAGs, the findings collectively point towards the critical role of Amdahl's Law in parallel workflow optimisation. As the parallelisable portion of the workflow becomes a diminishing fraction of the total workload, the observed overhead and diminishing returns necessitate the need for fine-tuning and careful consideration of the optimal number of cores. These insights provide valuable guidance to maximise the performance of parallel workflows while navigating the intricate trade-offs between speed-up and associated overhead. Further

investigation into task dependencies and load balancing may offer additional avenues for optimisation in complex parallel computing environments (Sanjayshiradwade, 2023).

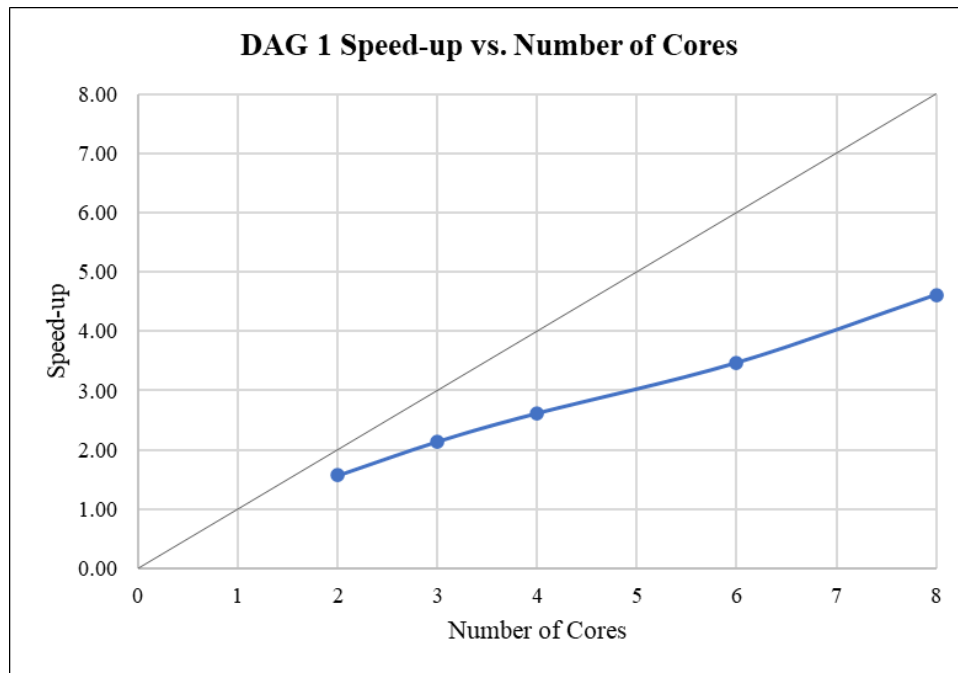


Figure 5.1: DAG 1 Performance at Varying Number of Cores

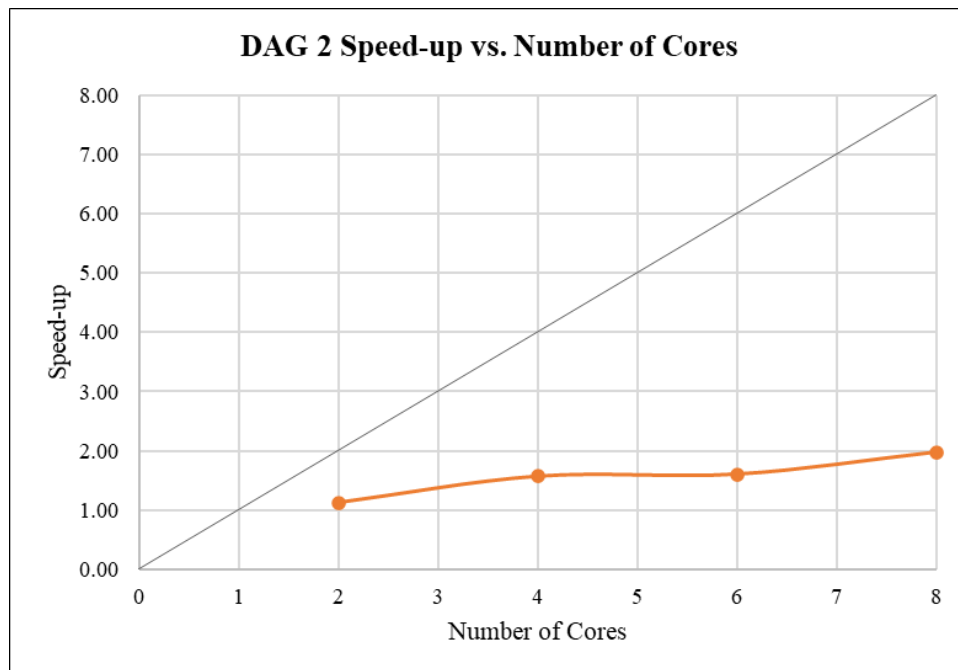


Figure 5.2: DAG 2 Performance at Varying Number of Cores

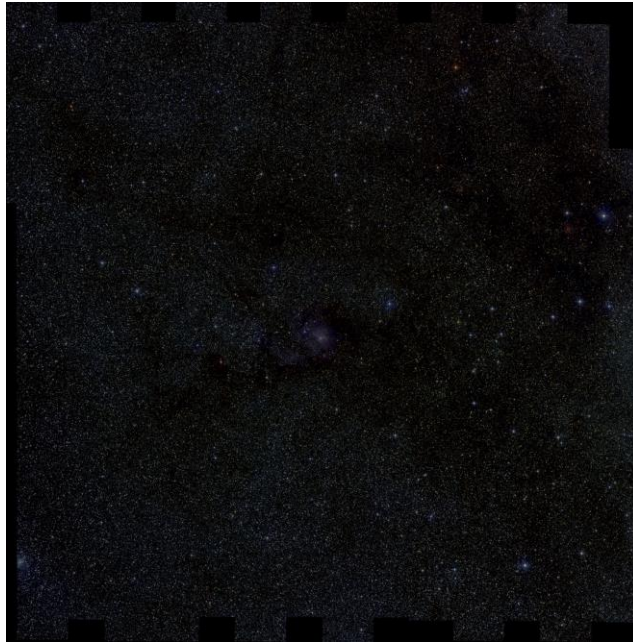


Figure 5.3: M8 Image Mosaic from DAG 1

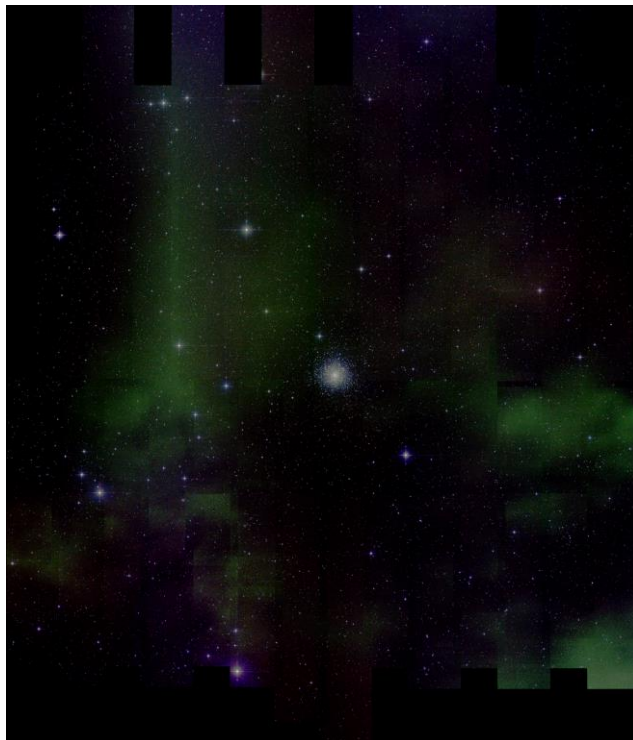


Figure 5.4: M15 Image Mosaic from DAG 1

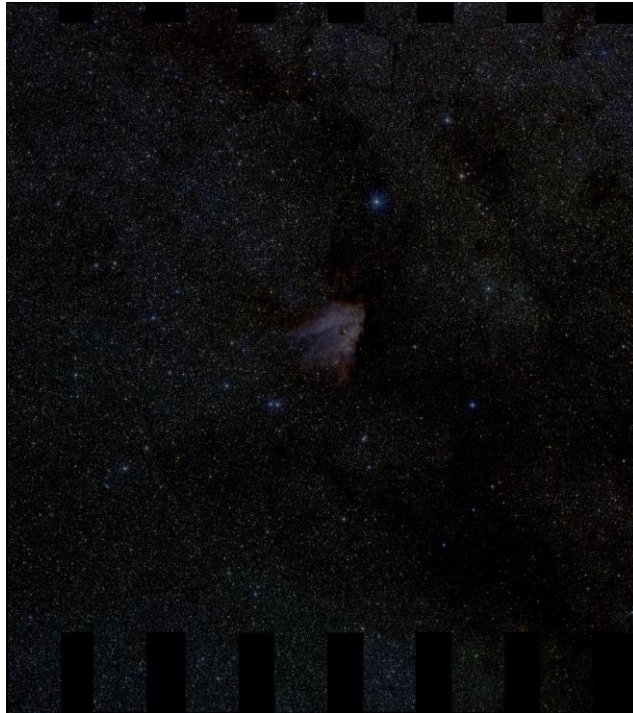


Figure 5.5: M17 Image Mosaic from DAG 1

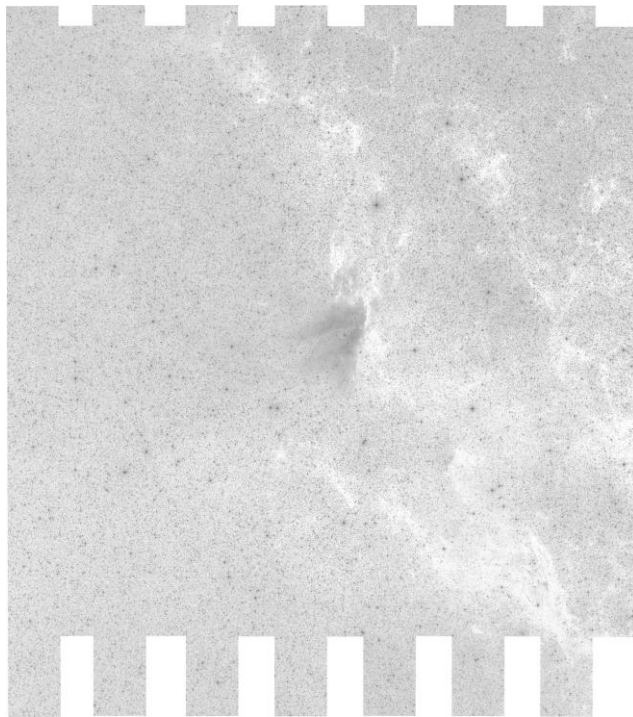


Figure 5.6 M17 k-band Image Mosaic from DAG 2

5.2 Challenges

In an ideal scenario where there is no communication delays or other inefficiencies associated with parallel processing, the speed-up would be linear. This implies that in an ideal parallelisation scenario, running tasks on 8 cores would ideally result in a speed-up of 8x. However, the observed speed-up in DAG 1 is only 4.61x. There is a notable 3.4x discrepancy between ideal speed-up and observed speed-up, which prompts an investigation into potential bottlenecks. First, the constrained throughput of the Network File System (NFS) hinders efficient data transfer between nodes, impeding seamless parallel execution. The throughput from one node to another is ~1 Gbps (125 MB/s) as shown in Figure 5.7. Four executor nodes theoretically have a combined throughput of 4 Gbps (500 MB/s). However, the NFS server is capped at 1 Gbps. Despite having a combined throughput of 4 Gbps across four executor nodes, the NFS server's capacity of 1 Gbps acts as a bottleneck, impacting the efficiency of parallelised tasks, especially when dealing with substantial datasets where rapid data transfer is crucial.

```
ubuntu@ip-172-31-91-141:~$ iperf3 -c 172.31.83.235
Connecting to host 172.31.83.235, port 5201
[ 5] local 172.31.91.141 port 52544 connected to 172.31.83.235 port 5201
[ ID] Interval           Transfer             Bitrate             Retr  Cwnd
[ 5]  0.00-1.00   sec    137 MBytes    1.15 Gbits/sec     10   1.02 MBytes
[ 5]  1.00-2.00   sec    121 MBytes    1.02 Gbits/sec      2    926 KBytes
[ 5]  2.00-3.00   sec    121 MBytes    1.02 Gbits/sec      3    804 KBytes
[ 5]  3.00-4.00   sec    121 MBytes    1.02 Gbits/sec      2    1.02 MBytes
[ 5]  4.00-5.00   sec    121 MBytes    1.02 Gbits/sec      4    935 KBytes
[ 5]  5.00-6.00   sec    120 MBytes    1.01 Gbits/sec      4    804 KBytes
[ 5]  6.00-7.00   sec    121 MBytes    1.02 Gbits/sec      1    1.02 MBytes
[ 5]  7.00-8.00   sec    121 MBytes    1.02 Gbits/sec      2    953 KBytes
[ 5]  8.00-9.00   sec    121 MBytes    1.02 Gbits/sec      4    804 KBytes
[ 5]  9.00-10.00  sec    121 MBytes    1.02 Gbits/sec      2    1.03 MBytes
- - - - -
[ ID] Interval           Transfer             Bitrate             Retr
[ 5]  0.00-10.00  sec    1.20 GBytes    1.03 Gbits/sec     34
[ 5]  0.00-10.05  sec    1.20 GBytes    1.02 Gbits/sec
iperf Done.
ubuntu@ip-172-31-91-141:~$
```

Figure 5.7: Network Performance between Nodes

Another contributing factor to the observed speed-up shortfall is the presence of uneven parallelism. With 9 parallel lines of tasks and only 8 cores available, one line consistently needs to find a gap for execution when a job from the other 8 is completed. In the end, the other 8 lines are completed, leaving that one line running sequentially. The consequence of one task line executing sequentially amidst other parallel lines significantly impacts overall efficiency. Despite having 9 parallel lines, the practical parallelism is constrained by the number of available cores. Eventually, this uneven distribution of tasks limits the achieved speed-up, emphasising the importance of aligning the number of parallel tasks with available computational resources.

In summary, the analysis highlights that achieving complete parallelism involves addressing both the constraints in data transfer capacities, particularly with NFS limitations, and the intricate balance between parallel task distribution and the available computational

resources. Resolving these bottlenecks is essential for maximising the potential speed-up in parallelised executions and ensuring an efficient workflow.

6.0 Conclusion

In conclusion, the project aimed to design and implement a distributed system using AWS for executing the Montage scientific workflow, which involves creating astronomical image mosaics. The architectural design incorporated AWS services such as EC2, HTCondor as the resource management and task scheduler, NFS for distributed data management, and HTCondor Dagman as the workflow management engine.

The project successfully demonstrated two parallelisation strategies within the Montage workflow. The first, DAG 1, focused on multi-bands parallelisation, allowing simultaneous processing of distinct near-infrared bands for multiple image sets. The second, DAG 2, demonstrated single-band parallelisation, optimising the processing of a specific set of images. Performance evaluations revealed significant speed-ups in both strategies, emphasising Montage's ability to harness parallel computing resources efficiently. However, challenges were identified, particularly in achieving ideal speed-ups. The limitations of NFS throughput and uneven parallelism contributed to a discrepancy between ideal and observed speed-ups.

In short, the project successfully designed, implemented, and evaluated a distributed data processing platform for Montage, showcasing the benefits of parallelisation in handling complex scientific workflows. The findings and challenges provide valuable insights for further optimisation and refinement of distributed computing solutions in astronomical data processing.

7.0 Appendices

Appendix A: HTCondor Configuration and Setup

<https://github.com/yjoer/htcondor-cluster>

Appendix B: Montage Workflow Scripts

<https://github.com/yjoer/montage>

8.0 References

- 2MASS Batch Image Service Results. (n.d.). Retrieved December 24, 2023, from <https://archive.ph/AWa3I#selection-135.0-135.33>
- Alan T. Tokunaga. (2001). *Specifications for Astronomical Infrared Filters*. <https://home.ifa.hawaii.edu/users/tokunaga/filterSpecs.html>
- Amy, B. (2010). *Making mosaics using MONTAGE* - CoolWiki. https://vmcoolwiki.ipac.caltech.edu/index.php/Making_Mosaics_Using_MONTAGE
- Berriman, G. B., Deelman, E., Good, J. C., Jacob, J., Katz, D. S., Kesselman, C., Laity, A. C., Prince, T. A., Singh, G., & Su, M. (2004). Montage: a grid-enabled engine for delivering custom science-grade mosaics on demand. *Proceedings of SPIE*. <https://doi.org/10.1117/12.550551>
- Berriman, G. B., Good, J. C., Laity, A. C., Jacob, J., Katz, D. S., Deelman, E., Singh, G., & Su, M. (2008). Chapter 19: Web-based tools-montage: An astronomical image mosaic engine. In *Astronomical Society of the Pacific eBooks* (Vol. 382, p. 179). <https://resolver.caltech.edu/CaltechAUTHORS:20100628-152638046>
- Katz, D. S., Jacob, J., Deelman, E., Kesselman, C., Singh, G., Su, M., Berriman, G. B., Good, J. C., Laity, A. C., & Prince, T. A. (2005). A Comparison of Two Methods for Building Astronomical Image Mosaics on a Grid. *Proceedings of the 2005 International Conference on Parallel Processing Workshops (ICPPW'05)*. <https://doi.org/10.1109/icppw.2005.6>
- Kshemkalyani, A. D., & Singhal, M. (2011). *Distributed computing: Principles, Algorithms, and Systems*. Cambridge University Press.
- Montage - image mosaic software for astronomers. (n.d.). <http://montage.ipac.caltech.edu/>
- Sanjayshiradwade. (2023). Understanding Amdahl's Law: a key to efficient parallel computing. *Medium*. <https://medium.com/@sanjayshiradwade/understanding-amdahls-law-a-key-to-efficient-parallel-computing-1cac756eba43>
- Van Steen, M., & Tanenbaum, A. S. (2017). *Distributed systems*. Createspace Independent Publishing Platform.