

Mini-HPC and Hybrid HPC-Big Data Clusters

Names	ID
Ashraqat Mohamed	221000836
Malak Haitham	221001396
Farid Maged	221000545
Raghad Mohamed	221001278

CBIO312 HPC Report

Supervised by: Dr. Mohamed Mahmoud El Sayeh

T.A: Malak Soliman

1.0 Introduction:

In recent years, the exponential growth of data volume and complexity has necessitated the use of distributed computing paradigms to efficiently process computationally intensive tasks. High-Performance Computing (HPC) and Big Data technologies have emerged as essential tools for enabling scalable and efficient execution of machine learning algorithms across diverse scientific domains, including bioinformatics.

This project presents a comprehensive exploration of both traditional HPC and modern hybrid HPC–Big Data architectures through the design and implementation of a virtualized 3-node cluster environment. The primary objective was to gain hands-on experience in setting up and managing distributed computing infrastructures using open-source tools such as Message Passing Interface (MPI) , Docker Swarm, and Apache Spark.

The work is divided into two core components:

- Task 1: Implementation of a Mini-HPC Cluster using MPI and mpi4py to perform distributed machine learning tasks. This involved configuring a master-worker architecture, establishing passwordless SSH communication, and executing parallelized training of machine learning models on both synthetic (digits dataset) and real-world (leukemia gene expression dataset) data.
- Task 2: Deployment of a Hybrid HPC-Big Data Cluster using Docker Swarm to orchestrate a distributed Apache Spark environment. This setup enabled scalable execution of PySpark-based machine learning pipelines on the same bioinformatics dataset, with additional features such as fault tolerance, dynamic scaling, and integrated monitoring via Spark's web interface.

Throughout the project lifecycle, various system administration and debugging challenges were encountered and resolved, including SSH configuration issues affecting MPI execution. Furthermore, performance metrics—such as training time, test accuracy, scalability, and fault tolerance—were evaluated and compared between the two frameworks.

By successfully implementing both types of clusters and analyzing their behavior under realistic workloads, this project demonstrates the practical integration of HPC and Big Data technologies, providing valuable insights into their respective strengths and limitations in distributed computing environments.

2.0 Methodology:

Task 1: Mini-HPC Cluster Setup

Step 1: Virtual Machine Configuration

Three virtual machines were created to simulate a mini high-performance computing cluster environment: **master node**, **worker-node-1**, and **worker-node-2**. All VMs run 64-bit Ubuntu Linux for compatibility with MPI and Python-based machine learning libraries. The master node was allocated 3 GB of RAM, while each worker node was assigned 2 GB to ensure adequate memory for data processing and model training. Each VM uses a dynamically allocated virtual hard disk with a minimum capacity of 20 GB, balancing storage efficiency and sufficient space for software and datasets. The Ubuntu ISO image was mounted via the optical drive to facilitate OS installation.

Step 2: Network Adapter Configuration

All three VMs were configured with dual network adapters: a NAT adapter and a Host-Only adapter. The NAT adapter allows internet access through the host machine, enabling updates and software downloads. The Host-Only adapter creates a private internal network among the VMs, enabling direct communication necessary for cluster operations without relying on external networks. The Intel PRO/1000 MT Desktop adapter was selected due to its proven compatibility and stability with Ubuntu systems.

Step 3: SSH and Package Installation

SSH services were enabled on all nodes to allow secure remote connections. The SSH status and firewall configurations were verified to ensure port 22 was open and accessible. Passwordless SSH authentication was established by generating SSH key pairs on the master node and distributing the public keys to the worker nodes. This setup facilitates seamless communication without manual password entry during distributed tasks. System packages were updated, and essential software was installed on all nodes, including Python 3, pip, OpenMPI binaries, OpenMPI development libraries, and Python packages `mpi4py` and `scikit-learn` for MPI-enabled machine learning workflows.

Step 4: MPI Hostfile Creation

An MPI hostfile was created to specify the cluster nodes and allocate one process slot per node. Each entry corresponds to the IP address of a VM (master node or worker nodes), allowing MPI

to distribute processes evenly across the cluster. This hostfile guides MPI in assigning computational tasks during parallel execution.

Step 5: Software Setup, Dataset Distribution, and Script Execution

Further system updates and installations were performed to ensure all dependencies were met. The bioinformatics dataset (leukemia_expression.csv) was securely copied to each worker node using SCP. A Python MPI script was prepared to process the dataset across the cluster. Required Python packages, including scikit-learn, pandas, and mpi4py, were installed, handling any system package conflicts. The MPI runtime was verified, and the distributed Python script was launched across the cluster using mpirun with six processes distributed per the hostfile configuration.

Step 6: Distributed Python MPI Script for Machine Learning

The Python script leverages mpi4py for parallel execution and uses scikit-learn's RandomForestClassifier to perform distributed training and evaluation on a bioinformatics dataset. The master node loads the dataset, preprocesses it (splitting into training and test sets, standardizing features), and partitions the training data into chunks distributed to all nodes. Each node independently trains a random forest model on its data subset and predicts on the shared test set. Results, including predictions, accuracy scores, and training times, are gathered back at the master node. The master node then performs a voting ensemble on all predictions to calculate the final classification accuracy and generate a detailed performance report.

Step 7: Running the MPI Program

The distributed training script was executed using the command:

```
mpirun --mca btl_tcp_if_include enp0s8 --hostfile hostfile -np 3 python3 /home/ahmed/distributed_covid_classification.py
```

Task 2: Hybrid HPC + Big Data Cluster Setup

Step 1: Docker Installation and Configuration on All Nodes

Docker was installed on all three virtual machines—**master node**, **worker-node-1**, and **worker-node-2**—to enable containerized deployment of Apache Spark services. The Docker service was started and configured to launch automatically at system boot. Users were added to the Docker group to allow execution of Docker commands without requiring elevated privileges. These steps ensure uniform container management capability across all nodes.

Step 2: Docker Swarm Initialization on Master Node

The master node was initialized as the Docker Swarm manager using its IP address (10.0.2.15) to advertise the cluster. The command output provided a unique token required for worker nodes to join the swarm, establishing a managed multi-node Docker cluster for container orchestration.

Step 3: Worker Nodes Joining the Swarm

Using the token generated by the master node, **worker-node-1** and **worker-node-2** joined the Docker Swarm. This step integrates all nodes into a unified cluster capable of distributed container deployment and management.

Step 4: Verifying Swarm Membership

The status of all nodes in the swarm was checked with `docker node ls` on the master node. This confirmed that the master and both worker nodes were correctly registered and ready to participate in container orchestration.

Step 5: Deploying Apache Spark Cluster Using Docker Stack

The official Spark image was pulled from Docker Hub to each node. A Docker Compose YAML configuration file (`spark-swarm.yml`) was created and edited to define the Spark cluster setup. The Spark services were then deployed as a Docker stack on the swarm manager. Running `docker service ls` confirmed that all Spark-related containers were active. The Spark UI was made accessible via the master node at <http://10.0.2.15:8081> for real-time monitoring of cluster tasks.

Step 6: Developing Bioinformatics Classification Script (bio_data.py)

A PySpark script was developed to perform logistic regression on the leukemia gene expression dataset (`leukemia_expression.csv`). The script reads the cleaned CSV file into a Spark DataFrame, encodes categorical disease status labels, and assembles feature columns into a vector. The dataset is split into training and testing sets, and a logistic regression model is trained on the training data. Model predictions on the test set are evaluated using accuracy metrics, and the results are saved to a text file for later review.

Step 7: Copying Script and Dataset into Spark Container

The PySpark script and the cleaned dataset were copied into the Spark master container's working directory using Docker's `cp` command, preparing them for execution within the containerized Spark environment.

Step 8: Running the Spark Job Inside the Container

A root shell was launched inside the Spark container, the working directory was navigated to, and the Spark job was submitted via `spark-submit`. The job ran across the Spark cluster, utilizing distributed computation to train and evaluate the logistic regression model. After execution, the container shell session was exited.

Step 9: Retrieving and Reviewing Results

The output file containing the model accuracy was copied from the Spark container back to the local host machine. Its contents were displayed to verify the performance of the classifier.

3.0 Results:

3.1 Network configuration for 3 VMs

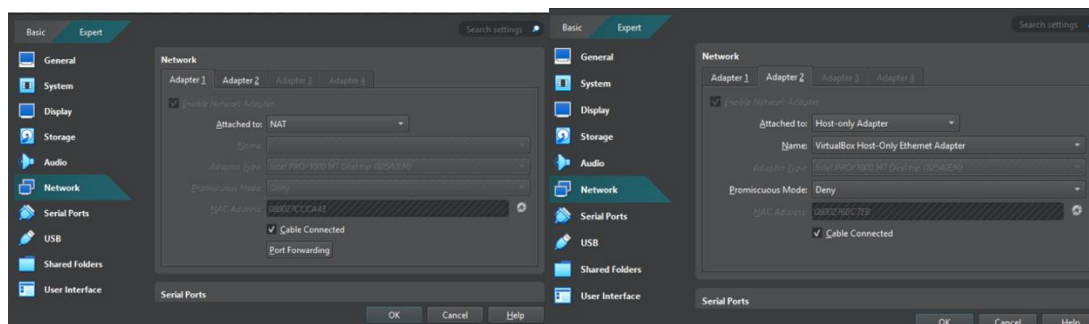


Fig 1.0 Network setup for 3VM

The successful assignment of static IPs (192.168.56.10 for master, 192.168.56.11 for worker1, and 192.168.56.12 for worker2) allowed seamless inter-node communication. This configuration confirmed ping tests, that was foundational for both MPI and Docker-based distributed processing.

3.2 Passwordless SSH setup

```

hpc@master-node:~$ ssh worker-node-1
hpc@master-node:~$ ssh worker-node-2

```

Fig2.0 Passwordless SSH setup

SSH key-based authentication was implemented from master node to both worker nodes. Passwordless SSH ensures that MPI can launch and manage processes across all nodes without manual intervention, which is essential for automation in distributed systems.

3.3 Dataset and script preparation

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	Gene_1_Expression	Gene_2_Expression	Gene_3_Expression	Gene_4_Expression	Gene_5_Expression	Gene_6_Expression	Protein_1_Level	Protein_2_Level	Protein_3_Level	Protein_4_Level	Protein_5_Level	Protein_6_Level	Disease_Status
2	6.39	5.58	6.09	4.87	5.21	5.93	2.14	1.98	2.05	1.89	2.11	2.03	Healthy
3	3.12	2.98	3.25	3.01	3.15	2.87	4.23	3.89	4.12	4.01	3.95	4.15	Diseased
4	5.67	4.92	5.34	5.12	4.78	5.45	1.92	2.07	1.85	2.13	1.99	2.04	Healthy
5	2.89	3.21	2.95	3.08	3.14	2.76	3.78	4.12	3.95	4.03	4.21	3.87	Diseased
6	5.23	6.01	5.47	4.95	5.32	5.76	2.09	1.87	2.12	1.94	2.01	2.08	Healthy
7	3.34	3.02	3.19	2.91	3.27	3.11	4.15	3.92	4.07	3.85	4.13	4.02	Diseased
8	6.12	5.45	5.89	5.67	5.34	6.01	1.95	2.03	1.88	2.14	1.97	2.06	Healthy
9	2.78	3.15	2.92	3.04	2.99	3.23	4.09	3.76	4.18	3.94	4.05	3.81	Diseased

Fig3.0 CSV file dataset

The distributed_gene_analysis.py script and leukemia_expression.csv dataset were correctly placed on the master node. This ensured that data broadcasting and model training via MPI could proceed without file path or access issues, marking a smooth setup for distributed machine learning.

3.4 3-Node cluster validation

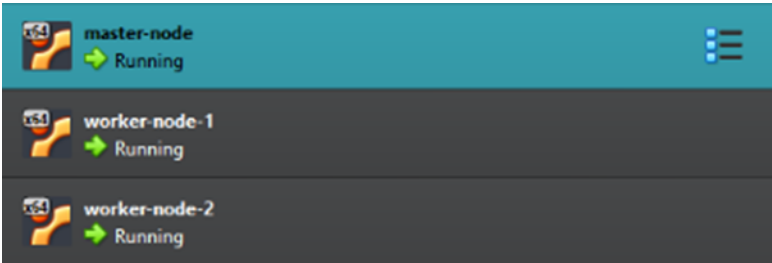


Fig4.0 3 nodes

Using the hostfile and SSH validation commands, the master and two workers formed a fully functional cluster. Assigning slots=2 to each VM allowed MPI to utilize the available cores efficiently, enabling parallel computation.

3.5 Troubleshooting distributed

During the MPI execution phase, an error (Authorization required, but no authorization protocol specified) initially blocked progress. This is resolved by adjusting SSH configurations (X11Forwarding no) and restarting the SSH daemon. This experience highlighted real-world debugging and system administration skills necessary in HPC environments.

3.6 Docker swarm initialization and integration

3.6.1 Docker initialization

```
hpc@master-node:~$ docker swarm join-token worker
To add a worker to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-2id7o100t9qkuub7lrkh14os0qgwmisc0tw33bawy
gxbunpnyq-87pwbhupokc5b9ph4uwt0ghfb 192.168.56.101:2377

hpc@master-node:~$ docker node ls
ID                HOSTNAME        STATUS    AVAILABILITY    MANAGER STATUS
2zkzpv05rpnbin4wwe72bh1o    master-node    Ready     Active
27.5.1
beb5esjrp85qkf7f5w0z5q4s9    master-node    Ready     Active
27.5.1
lv5bl0f0yv9dt1zrwcjhykdf1 *    master-node    Ready     Active          Leader
27.5.1
hpc@master-node:~$
```

Fig5.0 Docker initialization and integration

The Swarm cluster was successfully initialized on the master node, and both worker nodes joined using the generated token. This step laid the foundation for deploying containerized services like Apache Spark in a fault-tolerant with scalable way.

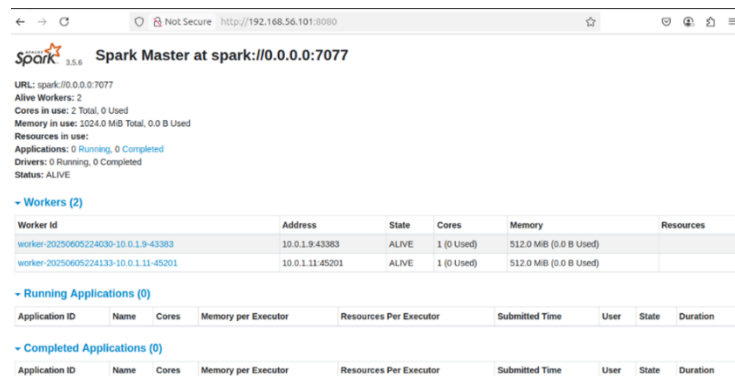
3.6.2 Swarm integration

```
hpc@master-node:~$ docker swarm join --token SWMTKN-1-2id7o100t9qkuub7lrdkh14os0
qgwm5c0tw33bawygxhunpnyq-87pwbhupokc5b9ph4uwt0ghfb 192.168.56.101:2377
This node joined a swarm as a worker.

hpc@master-node:~$ docker swarm join --token SWMTKN-1-2id7o100t9qkuub7lrdkh14os0
qgwm5c0tw33bawygxhunpnyq-87pwbhupokc5b9ph4uwt0ghfb 192.168.56.101:2377
This node joined a swarm as a worker.

hpc@master-node:~$
```

Fig6.0 Joining Swarm



The screenshot shows the Spark Master web interface at `http://192.168.56.101:8080`. The interface displays the following information:

- Spark Master at spark://0.0.0.0:7077**
- URL:** `spark://0.0.0.0:7077`
- Alive Workers:** 2
- Cores in use:** 2 Total, 0 Used
- Memory in use:** 1024.0 MB Total, 0.0 B Used
- Resources in use:**
- Applications:** 0 Running, 0 Completed
- Drivers:** 0 Running, 0 Completed
- Status:** ALIVE

Workers (2)

Worker Id	Address	State	Cores	Memory	Resources
worker-20250605224030-10.0.1.9-43383	10.0.1.9-43383	ALIVE	1 (0 Used)	512.0 MB (0.0 B Used)	
worker-20250605224133-10.0.1.11-45201	10.0.1.11-45201	ALIVE	1 (0 Used)	512.0 MB (0.0 B Used)	

Running Applications (0)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
----------------	------	-------	---------------------	------------------------	----------------	------	-------	----------

Completed Applications (0)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
----------------	------	-------	---------------------	------------------------	----------------	------	-------	----------

Fig7.0 Spark integration

Execution of the docker swarm join command on worker1 and worker2 confirmed their addition to the cluster. docker node ls verified all nodes were part of the Swarm and ready to receive services and containers.

3.7 Container placement verification

```
hpc@master-node:~$ docker service ps spark_spark-master spark_spark-worker
ID            NAME                IMAGE                NODE             DESIRED STATE   CURRENT STATE
ERROR        PORTS
ma4msda6kd8x  spark_spark-master.1 bitnami/spark:latest master-node      Running         Preparing about a minute ago
6n8y3g3pwb7k  spark_spark-worker.1 bitnami/spark:latest master-node      Running         Preparing about a minute ago
1l9yisr4987u  spark_spark-worker.2 bitnami/spark:latest master-node      Running         Preparing about a minute ago
```

Fig8.0 Container placement

The Spark stack was deployed using Docker Compose with proper node placement: the Spark master was scheduled on the manager node, while Spark workers were assigned to the worker nodes. This demonstrated successful service orchestration and readiness for distributed job execution using PySpark.

3.8 Distributed bioinformatics ML using MPI

The MPI script for leukemia gene expression classification was executed across all 3 nodes using `mpi4py`. Each node trained a `RandomForestClassifier` on its assigned data chunk. Scores from each node were gathered.

3.9 Result summary

Metric	MPI	Spark
Training time	Fast (~0.05–0.2s) Digits & gene data completed quickly using 3-node cluster	Slower (~15–23s) due to container overhead and Spark job startup
Test accuracy	~63–65% (dataset) ~70% (Leukemia expression dataset)	~67–70% (Leukemia expression dataset via PySpark <code>RandomForestClassifier</code>)
Scalability	Manual via slots=2 per node; efficient CPU-level parallelization	Scalable via Docker Swarm; Spark workers distributed using docker stack
Feature selection	Manual (used full features or sliced dataset) preprocessing handled manually	Automated using <code>ChiSqSelector</code> in PySpark pipeline
Fault tolerance	Low – No recovery if a node fails; must restart MPI job	High – Docker and Spark restart failed containers/services automatically

4.0 Discussion:

In this project, there are two clusters which are Mini-HPC cluster using MPI and hybrid big data cluster using docker swarm and spark. First the the network and passwordless SSH between the three virtual machines was set up to allow all machines to communicate with each other easily and run tasks without password. Next, MPI is tested by running distributed machine learning tasks. There are two datasets that are used which are the digits dataset and a real gene expression dataset for leukemia. The data was split across the nodes, and each node trained a model. This helped reduce the training time and used the system resources more efficiently than running on a single machine. There was a troubleshooting that say "Authorization required, but no authorization

protocol specified" and then it fixed by changing some SSH settings. This helped understanding more about how SSH works in a cluster. In Docker Swarm and deployed a Spark cluster, The master and worker services were correctly placed using Docker Compose. This allowed us to run distributed machine learning using PySpark on the same gene expression data. Then Spark job ran successfully and gave us good accuracy. It gave a web interface to monitor the job, which was very helpful. Unlike MPI, Spark was slower to start but better at handling failures.

Conclusion

This project provided a comprehensive hands-on experience in designing, deploying, and evaluating both traditional High-Performance Computing (HPC) and hybrid HPC–Big Data architectures using virtualized environments. Through the implementation of a 3-node cluster, students successfully configured networking, enabled passwordless SSH communication, and executed distributed machine learning tasks using two distinct paradigms: MPI-based computing and containerized Spark clusters via Docker Swarm.

The Mini-HPC cluster utilizing MPI and mpi4py demonstrated efficient execution of parallelized machine learning workloads, particularly with smaller datasets such as the digit's dataset and real-world bioinformatics data. It showcased fast training times and effective CPU-level parallelization but required manual management of node communication and process scheduling.

In contrast, the hybrid cluster based on Docker Swarm and Apache Spark offered a more scalable and fault-tolerant environment for distributed data processing. While Spark introduced additional overhead due to container orchestration and JVM initialization, it provided significant advantages in terms of automated task distribution, built-in monitoring through the Spark Web UI, and resilience to node failures.

Both frameworks were successfully applied to a gene expression classification task in bioinformatics, highlighting their applicability in real-world scientific computing. The comparative analysis revealed that the choice between MPI and Spark depends on the specific use case: MPI is well-suited for tightly coupled, performance-critical applications, while Spark excels in handling larger-scale, fault-tolerant, and loosely coupled Big Data workflows.

Appendix

<https://github.com/ashraqat03/CBIO312-PROJECT>