



ZCHPC
Accelerating innovation through Supercomputing

ZIMBABWE CENTRE FOR HIGH PERFORMANCE COMPUTING (ZCHPC)

Accessing, Configuring, and Running Parallel Jobs on HPC

Introduction

- High-Performance Computing (HPC) clusters enable researchers to tackle complex, computationally demanding problems by combining the processing power of multiple nodes.
- This distributed computing approach allows tasks to be executed in parallel, significantly speeding up solutions to challenges that would overwhelm a single machine.
- By breaking down complex problems into smaller tasks and running them concurrently across multiple nodes, HPC clusters reduce computational bottlenecks, shorten solution times, and reveal insights beyond the reach of traditional computing methods.

Prerequisites

- A ZCHPC account is required to access the cloud platform.
- Proficiency with the Xen Orchestra interface is needed.
- Parallel processing programs, including scientific simulations, data analytics, and machine learning algorithms.
- Basic command-line knowledge enhances operational clarity and efficiency.

Logging In

- **Logging into Xen Orchestra**

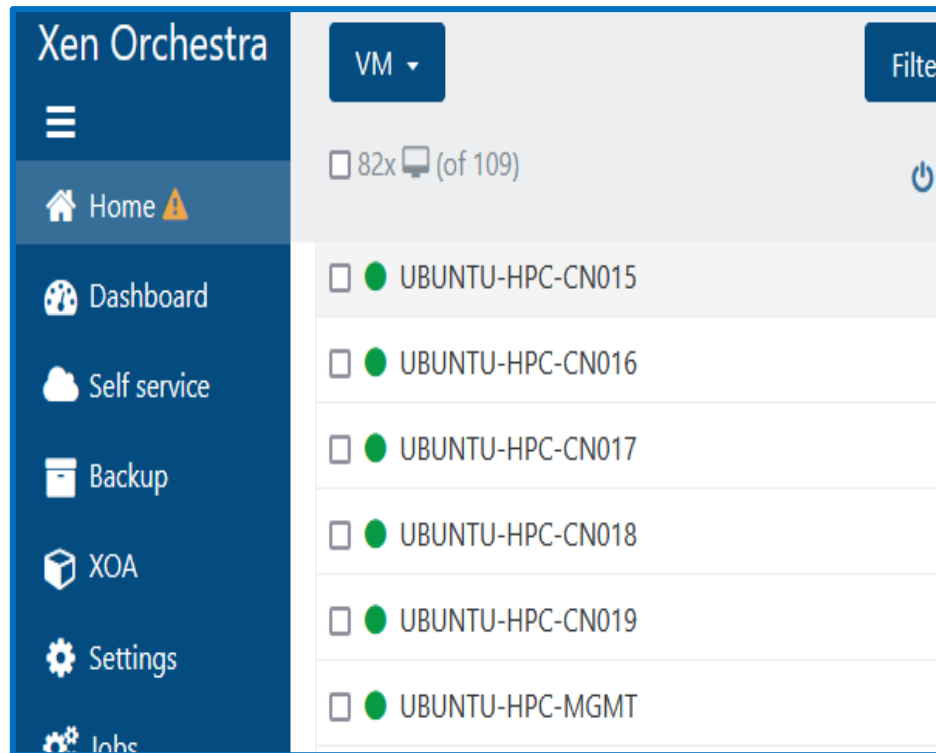
1. Go to <https://cloud.zchpc.ac.zw>.

2. Enter your **login credentials** to access the platform.



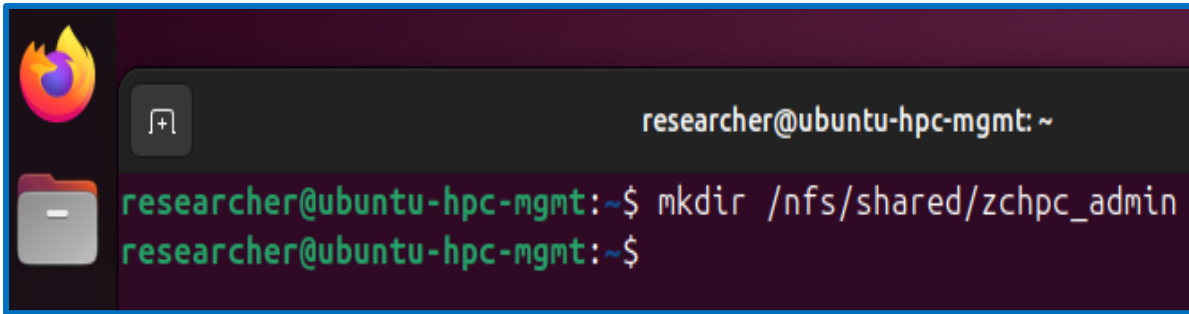
The image shows the Xen Orchestra login page. At the top is the Xen Orchestra logo, which consists of two red circles with a white 'X' shape overlaid. Below the logo is the text 'Xen Orchestra'. Underneath, there are two input fields: the first is for the username, containing the text 'administrator', and the second is for the password, which is masked with black dots. Below the password field is a checkbox labeled 'Remember me'. At the bottom is a blue button with the text 'Sign in with password'.

Accessing the Job Submission Virtual Machine



1. To submit HPC jobs, access the designated VM, **UBUNTU-HPC-MGMT**, displayed on the left.
2. Contact the HPC administrator for login credentials or further assistance.

Setting Up Your User Directory

A terminal window with a dark purple background. The top bar shows the Firefox logo on the left and a window icon with a plus sign on the right. The terminal text shows the user 'researcher@ubuntu-hpc-mgmt' at the '~' directory. The command 'mkdir /nfs/shared/zchpc_admin' is entered and executed, resulting in a new prompt.

```
researcher@ubuntu-hpc-mgmt: ~  
researcher@ubuntu-hpc-mgmt:~$ mkdir /nfs/shared/zchpc_admin  
researcher@ubuntu-hpc-mgmt:~$
```

1. Create your user directory using the command:

mkdir /path/to/directory

- Replace **zchpc_admin** with your username as shown in the example on the left.

2. Copy your data into the newly created directory using:

- **cp /source/path /destination/path**

Workflow for Submitting Jobs

❖ Create an application script

- Write or configure the application script for your task.
- For example, if using Python, the script might be `app.py`.

❖ Prepare your data

- Organise and verify your input data before submission.
- Ensure data is in the appropriate directory.

❖ Create a job submission script

- Write a job script that specifies resource requirements such as time, CPU, memory, and the executable.
- This is typically done with a script - `job.sh`.

❖ Submit the job

- Submit the job to the queue using sbatch.
 - Example: **sbatch job.sh**

❖ Monitor the job

- Check the status of the job in the queue with the following command:
 - **squeue -u your_username**

❖ Retrieve the output

- Once the job is complete, retrieve your output from the designated directory.
- Results are usually stored in the output file defined in the job script (**result.txt**).
 - To display the contents of the result.txt file, use the following command in the terminal:
 - **cat result.txt**

Example: Running a Python program

- In this section, we will explore snippets of a Python program designed to implement parallel processing.
- The complete code for this example can be found in the following directory:
 - **`/nfs/shared/examples/example02/`**
- This example aims to provide users with a foundational understanding of writing parallel programs that can be executed on the cluster.

Setting Up MPI (Message Passing Interface)

- **Purpose:** Initialises the MPI environment and retrieves each process's unique identifier (rank).
- **HPC Relevance:** MPI facilitates communication between processes running on different nodes in a distributed system, enabling parallel execution of tasks.

```
# MPI setup
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
```

Work Distribution Among MPI Ranks

- **Purpose:** Distributes the outer cross-validation folds among different MPI ranks (processes).
- **HPC Relevance:** This modulo operation ensures that each MPI process handles a subset of the folds, enabling parallel execution of cross-validation across multiple nodes or processors.

```
# Split the dataset using the outer CV
for i, (train_idx, test_idx) in enumerate(outer_cv.split(X, y)):
    if i % comm.Get_size() == rank:
        print(f"Rank {rank} working on fold {i}")
```

Multithreading with ThreadPoolExecutor

- **Purpose:** Utilises multiple threads within each MPI process to perform grid search concurrently.
- **HPC Relevance:** Combines distributed computing (MPI) with shared-memory parallelism (threads) to fully exploit the computational resources of multi-core CPUs within each node.

```
# Use threading to perform the grid search on all available CPUs
with ThreadPoolExecutor(max_workers=num_cpus) as executor:
    future = executor.submit(grid_search, X_train, y_train, X_test, y_test, param_grid, inner_cv)
    score, best_params = future.result()
    results.append((score, best_params))
```

Synchronisation and Gathering Results

- **Purpose:** Collects results from all MPI processes to a single root process for aggregation
- **HPC Relevance:** Ensures that all parallel computations are synchronised and that the final results are consolidated correctly, a critical aspect of distributed computing.

```
# Gather results from all ranks (MPI processes)
all_results = comm.gather(results, root=0)
```

Job Submission Script Breakdown

```
#!/bin/bash
#SBATCH --nodes=5
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=50
#SBATCH --output=output.log

# Load the Python environment module
module purge
module load python/fastpyenv

# Verify Python is coming from the correct environment
echo "Python executable: $(which python)"

# Allocate a folder specific to each node
NODE_FOLDER="/nfs/shared/zchpc_admin/node_${SLURM_NODEID}"
mkdir -p $NODE_FOLDER

# Run the Python script using mpirun
mpirun -np 5 python svm_hybrid_folder.py --node-folder $NODE_FOLDER --n-cpus 50
```

1. Job Configuration

- The first section configures the SLURM job submission.
 - #SBATCH --nodes=5: Allocates **5 nodes** for the job.
 - #SBATCH --ntasks-per-node=1: Runs **1 task per node**.
 - #SBATCH --cpus-per-task=50: Each task is assigned **50 CPU cores**.
 - #SBATCH --output=output.log: Specifies the output file, where job logs will be saved.

2. Loading Python Environment

- The job begins by purging any previously loaded modules using module purge.
- It then loads the Python environment (in this case, python/fastpyenv),.

3. Node-Specific Directory Creation

- The script dynamically creates a directory specific to each node where it runs.
- The variable NODE_FOLDER is constructed using SLURM_NODEID to ensure each node has its folder.

4. Running the Python Program with MPI

- The Python script svm_hybrid_folder.py is executed across the nodes using MPI (mpirun), ensuring parallel execution.
- The --node-folder argument passes the path to the node-specific folder created earlier, and --n-cpus 50 specifies the number of CPUs assigned for each task.