GWDG / Goettingen University
AGC / Mathematic and Informatics
Aasish Kr. Sharma

Exercise 1 / 2026-04-09
Practical Course on High Performance Computing / SoSe 2026
20 Minutes Total

Exercise Introduction

Before attempting the exercises in this document please ensure that you have read and understood the key topics covered in tutorial.

Contents

Task 1: IO Benchmarking (20 min)

As benchmarking could help to understand a system's performance, this task is designed to create a simple small-scale IO500 benchmarking test for estimating or learning how to run IO benchmarking using the IOR (throughput) and mdtest (metadata) tests from the IO500 suite. Understand how to interpret basic I/O performance results in an HPC context and gain hands-on experience with compiling and running a lightweight benchmark on a shared HPC resource. (Notes: When details contain typos mistake then understand it is not intensional and the reviewers has also failed to locate them.)

Task Structure:

1. **Introduction to Key IO500 Tests**
   - **Goal:** Understand the basic components of the **IOR** and **mdtest** benchmarks.
   - **IOR:** Measures the sequential read/write performance of the file system. Relevant for testing the bandwidth and throughput of HPC storage systems.
   - **mdtest:** Measures the metadata operations performance, such as file creation, file stat (status checks), and file deletion. Relevant for workloads that involve managing a large number of files or frequent file system operations.
2. **Part 1: Accessing the HPC Resources and collecting the test results.**
   - **Step 1:** Access the Cluster

     SSH into the SCC cluster:

     bash

     ```
     $ ssh username@cluster-address
     ```
     e.g., *ssh -i ssh_fileid username@login-mdc.hpc.gwdg.de*
   - **Step 2:** To run the IO benchmark test do the following:
     - Load the necessary modules for **MPI** and **GCC**:

       bash

```
$ module load mpi gcc
```

- Create a working directory for the benchmark tests:

  bash

  ```
  $ mkdir ./io500-small-exercise
  $ cd ./io500-small-exercise
  ```

- **Step 3:** Installing the Required Tools
  - Clone the IO500 repository from GitHub:

    bash

    ```
    $ git clone https://github.com/IO500/io500.git
    $ cd io500
    ```

  - Compile the IO500 benchmark:

    bash

    ```
    $ make
    ```

- **Step 4:** Running a Small-Scale IOR Test
  - Goal: Run a simple IOR test to measure read/write throughput on a small dataset.
  - Set up a scratch directory for the IOR test (can be on a shared parallel file system or local node storage):

    bash

    ```
    $ mkdir ./scratch/ior-small
    ```

  - Create an ior-small.ini configuration file to limit the dataset size:

    bash

    ```
    $ nano ior-small.ini
    ```

  - Inside ior-small.ini, configure a small IOR test (adjust the block size and transfer size for a quick run):

    ini

    ```
    $ [global]
    api = POSIX
    transferSize = 1m
    blockSize = 64m
    repetitions = 1
    ```

  - Run the IOR test using the following command:

    bash

    ```
    $ mpirun -np 4 ./ior -f ior-small.ini -o
    ./scratch/ior-small/ior-output
    ```

  - Analyze the IOR output, which will report the read/write bandwidth (in MB/s or GB/s).

- **Step 5:** Running a Small-Scale mdtest
  - Goal: Run a simple mdtest to evaluate metadata performance (file creation, stat, and deletion) on a small dataset.

- Create a directory for the mdtest run:

  bash

  ```
  $ mkdir ./scratch/mdtest-small
  ```

- Run mdtest with a reduced number of files (e.g., 1000 files) using this command:

  bash

  ```
  $ mpirun -np 4 ./mdtest -d ./scratch/mdtest-small -n 1000 -u -L
  ```

  * -n 1000 specifies the number of files to create.
  * -u ensures unique working directories for each process.
  * -L performs file stat operations.

- Once the test completes, the output will show the file creation rate, stat performance, and deletion performance.

3. **Part 2: Analyzing the Results**
   - **IOR Results:** Look at the reported write and read bandwidth. Discuss how these metrics reflect the file system's ability to handle sequential I/O operations.
   - **mdtest Results:** Examine the file creation rate (files/s), stat rate, and deletion rate. Discuss how file system metadata operations can impact performance in real HPC workloads (e.g., running simulations that generate many small files).
4. **Part 3: Class Discussion**
   - **Goal:** To think critically about the results and storage performance.
     - Why might the read performance differ from the write performance?
     - What are some factors that could affect metadata performance on an HPC system?
     - How would these results change if we increased the number of files or used a larger block size in IOR?
   - Explore potential optimizations (e.g., increasing the number of MPI processes, changing the block size, or utilizing a different file system).

Hints

- **Example solution:**
  - **IOR:** A small IOR test with a block size of 64MB and transfer size of 1MB should finish quickly with minimal load.
  - Example IOR output:
    * Write bandwidth: 200 MB/s
    * Read bandwidth: 180 MB/s
  - **mdtest:** A small mdtest with 1000 files can provide a quick evaluation of metadata performance.
  - Example mdtest output:
    * File creation rate: 30,000 files/s

        * File stat rate: 25,000 files/s
        * File deletion rate: 28,000 files/s

- **Discussion Points:**
  - You should notice that small-scale IOR runs are sufficient to demonstrate throughput bottlenecks, and mdtest offers insight into metadata handling efficiency.
  - You can also explore how increasing the number of files, adjusting block size, or running more processes would affect performance on the limited resources available.
- **Further Reading/References:**
  - **IO500** Documentation (https://github.com/VI4IO/io-500-dev/blob/master/doc/README.md)
  - **IOR** Benchmark User Guide (https://media.readthedocs.org/pdf/ior/latest/ior.pdf)
  - **mdtest** User Guide (https://www.illko.cz/images/dokumenty/mdtest_manual_en.pdf)

Optional Task 2: Optional: Compute Benchmarking (20 min)

This is a difficult **additional** task which will strengthen your understanding in the topic.

As benchmarking could help to understand a system's performance, this task is designed to create a simple benchmarking test program for estimating or measuring the compute performance of a system. For this, you could use your own choice of programming language and the guidelines for the tasks are as follows:

Exercise: MPI Matrix Multiplication Benchmark

Before attempting the exercises in this document please ensure that you have read and understood the key topics covered in tutorial.

Objective:

Benchmark the performance of matrix multiplication using MPI on an SCC cluster (your account). This exercise will involve writing a simple MPI program, varying the number of processes, and measuring the execution time to understand scalability.

To do this exercise you will need:

- Access to an SCC cluster with MPI installed.
- Basic knowledge of MPI programming (e.g., sending and receiving messages).
- Familiarity with compiling and running MPI programs.
- You will require python packages like MPI and numpy.

Contents

- Complete the given MPI program to perform matrix multiplication.

1. You could take help from ”Parallel Computing: Basic Principles” course by Oswald Haan on Friday (05.04.2024).

2. a) Skeleton code for function 1 (language python):

```
        def matrix_multiply(A, B):
        return np.dot(A, B)
```

   b) Skeleton Code for Function 2 (language python):

```
        /* Main function. */
        from mpi4py import MPI
        import numpy as np


        if __name__ == __main__:
        /* Declaring the required variables and the required MPI functions. */

        # Matrix dimensions
        N = 100
        M = 100
        K = 100


        # Create random matrices A and B
        np.random.seed(42)
        A = np.random.rand(N, M)
        B = np.random.rand(M, K)


        # Your code for MPI Functions




        # Your code for splitting matrices among processes




        # Your code for perform local matrix multiplication




        # Your code for gathering results




        if rank == 0:
        # Combine results
        result = np.concatenate(C, axis=0)
        print(Result Matrix (C):)
        print(result)
      }

      /* Finally, test and compile the code. Keep the file name mpi_matrix_multiply.py*/
```

3. Compile and run the program.

   – Compile the program using mpicc or mpifort, depending on your MPI compiler:

```
      mpicc -o mpi_matrix_multiply mpi_matrix_multiply.py
```

4. Run the program with varying numbers of MPI processes using the bash script:

```
    mpiexec -n <num_processes> ./mpi_matrix_multiply
```

5. Measure the execution time for each run using time or mpiexec's built-in timing options.

- Now, do the benchmarking using your program (known as strong and week scaling benchmark).
- Vary the number of MPI processes (-n) from 1 to a suitable maximum based on the available cores in your cluster.
- Record the execution time for each run.
- Plot a graph showing the scalability of the matrix multiplication with respect to the number of processes.
- X-axis: Number of MPI processes
- Y-axis: Execution time
- Analysis:
  - Analyze the benchmark results to understand the scalability of the matrix multiplication program.
  - Look for trends in execution time as the number of processes increases.
  - Identify any bottlenecks or inefficiencies in the program or the cluster configuration.
  - Summarize (short-document) and complete your benchmarking analysis.
  - Include all the details you argue relevant on your critical thinking.
  - For convenience you could also store your records in a CSV file, with the file name "hpctrainingNN.csv".
- Discussion:
  - How does the execution time change with an increasing number of processes?
  - Does the program exhibit strong or weak scaling?
  - What factors might contribute to the observed scalability?
  - How could the program or the cluster configuration be optimized for better performance?

Hints

- You could also create and include graphs and chart plots as felt necessary.
- Here "NN" is your user code. You could also share your findings during the session for the feedback.
- You can modify the matrix dimensions (N, M, K) to vary the size of the matrices and observe their impact on performance.
- We encourage you to experiment with different MPI functions, matrix sizes, and cluster configurations to deepen your understanding of HPC benchmarking.