**Project Report: Parallel and Distributed Processing - High Performance Computing**

**Student: Usman Ayub (user23)**  
**Date: 15/5/2025**

**Abstract**  
This report details a project undertaken in Parallel and Distributed Processing, focusing on High Performance Computing (HPC). The project was divided into two main parts. Part I involved an exploration of the SINES HPC cluster, including mapping compute nodes, determining their CPU specifications, and assessing their status. Part II focused on the implementation and performance analysis of a 2D Laplace solver using sequential, OpenMP, and MPI paradigms. The implementations were tested for correctness, and their performance was compared across different problem sizes and levels of parallelism. Preliminary performance data was collected on the afrit head node, with recommendations for further testing on dedicated compute nodes.

**1. Introduction**  
The primary goal of this project was to gain practical experience with HPC environments and parallel programming techniques. This involved understanding the architecture of an accessible HPC cluster (SINES) and applying parallelization strategies (OpenMP and MPI) to a common scientific computing problem, the Laplace solver. The project aimed to evaluate the performance and scalability of these parallel implementations.

**Part I: Getting to Know the SINES HPC Cluster**

**1.1. Objective**  
The objective of Part I was to familiarize with the SINES HPC cluster by:

1. Creating a map of accessible compute nodes.
2. Determining the CPU specifications of these compute nodes.
3. Checking for nodes that might be busy or unavailable.
4. Ranking nodes based on their (theoretical) performance and observed availability.

**1.2. Methodology**

**1.2.1. Node Discovery and Status Check**  
A shell script (probe\_nodes.sh) was developed and executed on the afrit head node to probe the status of compute nodes assumed to be in the range compute-0-0.local to compute-0-31.local. The script performed the following for each potential node:

* Checked against a list of known unavailable nodes (from the Message of the Day - MOTD).
* Attempted to ping the node to verify network reachability.
* If pingable, attempted a non-interactive ssh to execute nproc (to get core count) and hostname.

The probe\_nodes.sh script (simplified):

#!/bin/bash

echo "Probing compute nodes compute-0-0.local to compute-0-31.local..."

echo "-----------------------------------------------------------------"

echo "| Node Name          | Status    | Cores | Hostname (on node) |"

echo "|--------------------|-----------|-------|--------------------|"

# Define the list of known unavailable nodes from MOTD

UNAVAILABLE\_NODES=(7 14 15 20 25 29 30)

# Loop through the range, e.g., 0 to 31

for i in $(seq 0 31); do

    NODE\_NAME="compute-0-${i}.local"

    STATUS="DOWN" # Assume down by default

    CORES="N/A"

    REMOTE\_HOSTNAME="N/A"

    # Check if current node number is in the UNAVAILABLE\_NODES list

    IS\_UNAVAILABLE=0

    for un\_node in "${UNAVAILABLE\_NODES[@]}"; do

        if [ "$i" -eq "$un\_node" ]; then

            IS\_UNAVAILABLE=1

            break

        fi

    done

    if [ "$IS\_UNAVAILABLE" -eq 1 ]; then

        STATUS="UNAVAILABLE (MOTD)"

    else

        # Try to ping the node (1 packet, timeout 1 second)

        if ping -c 1 -W 1 "$NODE\_NAME" &> /dev/null; then

            STATUS="UP"

            # If UP, try to ssh and get nproc and hostname

            # The "ssh -o ConnectTimeout=5 -o BatchMode=yes" makes it non-interactive

            # and times out if ssh hangs.

            REMOTE\_INFO=$(ssh -o ConnectTimeout=5 -o BatchMode=yes -o StrictHostKeyChecking=no "$NODE\_NAME" "echo \"\$(nproc)---\$(hostname)\"" 2>/dev/null)

            if [ -n "$REMOTE\_INFO" ]; then

                # Use awk for more robust parsing in case hostname has dashes

                CORES=$(echo "$REMOTE\_INFO" | awk -F'---' '{print $1}')

                REMOTE\_HOSTNAME=$(echo "$REMOTE\_INFO" | awk -F'---' '{print $2}')

            else

                # SSH failed or command failed on remote node

                STATUS="UP (SSH/cmd\_fail)" # Still UP because ping worked

            fi

        fi

    fi

    printf "| %-18s | %-9s | %-5s | %-18s |\n" "$NODE\_NAME" "$STATUS" "$CORES" "$REMOTE\_HOSTNAME"

done

echo "-----------------------------------------------------------------"

echo "Probe complete."

echo "Note: 'Cores' is from nproc. 'UP (SSH/cmd\_fail)' means node is pingable but couldn't get details via SSH."

echo "      'UNAVAILABLE (MOTD)' nodes were not pinged."

**1.2.2. CPU Specification Detailing**  
For nodes identified as UP and accessible via SSH by the probe\_nodes.sh script, a manual SSH session was initiated from afrit, and the lscpu command was executed to obtain detailed CPU specifications.

**1.3. Results and Findings**

**1.3.1. Compute Node Map and Status**  
The execution of probe\_nodes.sh yielded the following status map for nodes compute-0-0.local to compute-0-31.local:

A white table with black text

AI-generated content may be incorrect.A screenshot of a computer program

AI-generated content may be incorrect.  
**Summary:** Out of 32 potential nodes, 17 were fully UP and responsive, 2 were UP but had SSH command failures, 6 were DOWN, and 7 were UNAVAILABLE due to maintenance.

**1.3.2. CPU Specifications**  
Heterogeneity was observed among the accessible UP nodes.

* **Type 1 Nodes (e.g., compute-0-0.local, 16 logical cores):**
  + Command: ssh compute-0-0.local followed by lscpu.

*A computer screen shot of a black screen

AI-generated content may be incorrect.*  
  
Key Specs:

* + - Architecture: x86\_64
    - CPU(s): 16 (2 threads per core)
    - Core(s) per socket: 4
    - Socket(s): 2
    - Vendor ID: GenuineIntel
    - Model: 26 (Corresponds to Intel Xeon 5500 "Nehalem-EP" series, likely E5520 as per SINES website's "General Purpose HPC" 2011 entry).
    - L3 cache: 8192K (8MB)
* **Type 2 Node (compute-0-8.local, 24 logical cores):**
  + Command: ssh compute-0-8.local followed by lscpu.

A screenshot of a computer

AI-generated content may be incorrect.  
  
Key Specs:

* + - Architecture: x86\_64
    - CPU(s): 24 (2 threads per core)
    - Core(s) per socket: 6
    - Socket(s): 2
    - Vendor ID: GenuineIntel
    - Model: 44 (Corresponds to Intel Xeon 5600 "Westmere-EP" series).
    - L3 cache: 12288K (12MB)
  + This node represents a more powerful configuration with more cores and a slightly newer CPU generation compared to the Type 1 nodes.
* **Nodes with SSH Issues (compute-0-3.local, compute-0-31.local):**
  + Attempting to SSH to compute-0-3.local resulted in a "REMOTE HOST IDENTIFICATION HAS CHANGED!" warning, preventing login.

A computer screen with white text

AI-generated content may be incorrect.

* + Detailed specifications could not be retrieved for these nodes.

**1.3.3. Node Utilization and Ranking**  
Direct access to job scheduler queue information (e.g., via sinfo or qstat) was not available from afrit. Therefore, node busy status was inferred:

* **Unavailable:** Nodes listed in MOTD or found DOWN or with unresolved SSH issues.
* **Likely Higher Demand:** The 24-core compute-0-8.local node is theoretically the highest performing CPU-wise among the probed set and is thus likely to be in higher demand.
* **Available:** Other UP nodes with 16 cores.

A detailed ranking combining theoretical performance and current availability would place compute-0-8.local at the top for performance among available nodes, followed by the 16-core UP nodes.

**Part II: Parallel Laplace Solver Implementation**

**2.1. Objective**  
The objectives of Part II were to:

1. Implement a 2D Laplace solver using sequential, OpenMP, and MPI programming models.
2. Verify the correctness of the parallel implementations against the sequential version.
3. Analyze and compare the performance and scalability of the OpenMP and MPI versions.

**2.2. Methodology**

**2.2.1. Laplace Equation and Jacobi Iteration**  
The 2D Laplace equation was solved numerically using the Jacobi iterative method. For a grid point A[i][j], the value at the next iteration A\_new[i][j] is the average of its four neighbors from the previous iteration A\_old:  
  
A\_new[i][j] = (A\_old[i-1][j] + A\_old[i+1][j] + A\_old[i][j-1] + A\_old[i][j+1]) / 4.0  
  
Fixed boundary conditions (value 1.0) were used, with interior points initialized to 0.0. N\_global refers to the internal dimension of the square grid.

**2.2.2. Implementation Details**  
All implementations were written in C++ and compiled using g++ (version 4.4.7 on the SINES HPC) via CMake.

* **Project Setup:**  
  A directory LaplaceSolver was created with a src subdirectory for source files and a build subdirectory for compilation.
* # On afrit
* cd ~
* mkdir LaplaceSolver
* cd LaplaceSolver

mkdir src build

* **Sequential Implementation (laplace\_sequential.cpp)**  
  A straightforward C++ implementation of the Jacobi iteration.

#include <iostream>

#include <vector>

#include <string>

#include <iomanip>

#include <stdexcept>

#include <ctime>

void printGrid(const std::vector<std::vector<double>>& grid, int n\_dim) {

    for (int i = 0; i < n\_dim; ++i) {

        for (int j = 0; j < n\_dim; ++j) {

            std::cout << std::fixed << std::setprecision(4) << grid[i][j] << "\t";

        }

        std::cout << std::endl;

    }

}

int main(int argc, char\* argv[]) {

    if (argc < 3) {

        std::cerr << "Usage: " << argv[0] << " <grid\_dimension> <num\_epochs>" << std::endl;

        return 1;

    }

    int N = 0;

    int epochs = 0;

    try {

        N = std::stoi(argv[1]);

        epochs = std::stoi(argv[2]);

    } catch (const std::invalid\_argument& ia) {

        std::cerr << "Invalid argument for N or epochs: " << ia.what() << std::endl;

        std::cerr << "Usage: " << argv[0] << " <grid\_dimension> <num\_epochs>" << std::endl;

        return 1;

    } catch (const std::out\_of\_range& oor) {

        std::cerr << "Out of Range error for N or epochs: " << oor.what() << std::endl;

        std::cerr << "Usage: " << argv[0] << " <grid\_dimension> <num\_epochs>" << std::endl;

        return 1;

    }

    if (N <= 0) {

        std::cerr << "Grid dimension must be > 0." << std::endl;

        return 1;

    }

     if (epochs < 0) {

        std::cerr << "Number of epochs must be >= 0." << std::endl;

        return 1;

    }

    int full\_dim = N + 2;

    std::vector<std::vector<double>> A(full\_dim, std::vector<double>(full\_dim));

    std::vector<std::vector<double>> A\_new(full\_dim, std::vector<double>(full\_dim));

    for (int i = 0; i < full\_dim; ++i) {

        for (int j = 0; j < full\_dim; ++j) {

            if (i == 0 || i == full\_dim - 1 || j == 0 || j == full\_dim - 1) {

                A[i][j] = 1.0;

            } else {

                A[i][j] = 0.0;

            }

        }

    }

    A\_new = A;

    time\_t startTime, endTime;

    time(&startTime);

    for (int epoch = 0; epoch < epochs; ++epoch) {

        for (int i = 1; i <= N; ++i) {

            for (int j = 1; j <= N; ++j) {

                A\_new[i][j] = (A[i - 1][j] + A[i + 1][j] + A[i][j - 1] + A[i][j + 1]) / 4.0;

            }

        }

        A = A\_new;

    }

    time(&endTime);

    double time\_taken = difftime(endTime, startTime);

    std::cout << "Sequential run." << std::endl;

    std::cout << "Grid size (internal): " << N << "x" << N << std::endl;

    std::cout << "Epochs: " << epochs << std::endl;

    std::cout << "Time taken: " << std::fixed << std::setprecision(5) << time\_taken << " sec" << std::endl;

    if (N < 8) {

        std::cout << "\nGrid after " << epochs << " epochs:" << std::endl;

        printGrid(A, full\_dim);

    }

    return 0;

}

**OpenMP Implementation (laplace\_openmp.cpp)**  
The sequential code was parallelized using OpenMP by adding a pragma to the nested loops performing the Jacobi update.  
  
#include <cstddef>

#include <iostream>

#include <vector>

#include <string>

#include <iomanip>

#include <stdexcept>

#include <omp.h>

#include <ctime>

void printGrid(const std::vector<std::vector<double>>& grid, int n\_dim) {

    for (int i = 0; i < n\_dim; ++i) {

        for (int j = 0; j < n\_dim; ++j) {

            std::cout << std::fixed << std::setprecision(4) << grid[i][j] << "\t";

        }

        std::cout << std::endl;

    }

}

int main(int argc, char\* argv[]) {

    if (argc < 3) {

        std::cerr << "Usage: " << argv[0] << " <grid\_dimension> <num\_epochs>" << std::endl;

        return 1;

    }

    int N = 0;

    int epochs = 0;

    try {

        N = std::stoi(argv[1]);

        epochs = std::stoi(argv[2]);

    } catch (const std::invalid\_argument& ia) {

        std::cerr << "Invalid argument for N or epochs: " << ia.what() << std::endl;

        std::cerr << "Usage: " << argv[0] << " <grid\_dimension> <num\_epochs>" << std::endl;

        return 1;

    } catch (const std::out\_of\_range& oor) {

        std::cerr << "Out of Range error for N or epochs: " << oor.what() << std::endl;

        std::cerr << "Usage: " << argv[0] << " <grid\_dimension> <num\_epochs>" << std::endl;

        return 1;

    }

    if (N <= 0) {

        std::cerr << "Grid dimension must be > 0." << std::endl;

        return 1;

    }

    if (epochs < 0) {

        std::cerr << "Number of epochs must be >= 0." << std::endl;

        return 1;

    }

    int full\_dim = N + 2;

    std::vector<std::vector<double>> A(full\_dim, std::vector<double>(full\_dim));

    std::vector<std::vector<double>> A\_new(full\_dim, std::vector<double>(full\_dim));

    for (int i = 0; i < full\_dim; ++i) {

        for (int j = 0; j < full\_dim; ++j) {

            if (i == 0 || i == full\_dim - 1 || j == 0 || j == full\_dim - 1) {

                A[i][j] = 1.0;

            } else {

                A[i][j] = 0.0;

            }

        }

    }

    A\_new = A;

    double startTime = omp\_get\_wtime();

    for (int epoch\_count = 0; epoch\_count < epochs; ++epoch\_count) {

        #pragma omp parallel for collapse(2)

        for (int i = 1; i <= N; ++i) {

            for (int j = 1; j <= N; ++j) {

                A\_new[i][j] = (A[i - 1][j] + A[i + 1][j] + A[i][j - 1] + A[i][j + 1]) / 4.0;

            }

        }

        A = A\_new;

    }

    double endTime = omp\_get\_wtime();

    double time\_taken = endTime - startTime;

    std::cout << "OpenMP run." << std::endl;

    std::cout << "Max threads available: " << omp\_get\_max\_threads() << std::endl;

    std::cout << "Grid size (internal): " << N << "x" << N << std::endl;

    std::cout << "Epochs: " << epochs << std::endl;

    std::cout << "Time taken: " << std::fixed << std::setprecision(5) << time\_taken << " sec" << std::endl;

    if (N < 8) {

        std::cout << "\nGrid after " << epochs << " epochs:" << std::endl;

        printGrid(A, full\_dim);

    }

    return 0;

}

* Timing was performed using omp\_get\_wtime().
* **MPI Implementation (laplace\_mpi.cpp)**  
  The MPI version employed row-wise domain decomposition. Each MPI process handled a contiguous strip of rows from the global grid.
  + **Domain Decomposition:** The N\_global internal rows were distributed as evenly as possible among world\_size MPI processes. Each process managed local\_N\_computational\_rows.
  + **Halo Exchange:** One halo row from the top neighbor and one from the bottom neighbor were stored by each process. MPI\_Sendrecv was used for simultaneous sending of a process's boundary computational row and receiving a neighbor's corresponding row into its halo region. MPI\_PROC\_NULL handled boundary conditions for processes at the global top/bottom.

Code Snippet (Key MPI Halo Exchange):  
  
#include <cstddef>

#include <iostream>

#include <vector>

#include <string>

#include <iomanip>

#include <stdexcept>

#include <mpi.h>

#include <numeric>

void printLocalGrid(const std::vector<std::vector<double>>& grid, int rank, const std::string& label) {

    std::cout << "Rank " << rank << " - " << label << ":" << std::endl;

    if (grid.empty() || grid[0].empty()) {

        std::cout << "  (empty local grid)" << std::endl;

        return;

    }

    for (size\_t i = 0; i < grid.size(); ++i) {

        std::cout << "  ";

        for (size\_t j = 0; j < grid[0].size(); ++j) {

            std::cout << std::fixed << std::setprecision(2) << grid[i][j] << "\t";

        }

        std::cout << std::endl;

    }

    std::cout << std::endl;

}

int main(int argc, char\* argv[]) {

    MPI\_Init(&argc, &argv);

    int world\_rank, world\_size;

    MPI\_Comm\_rank(MPI\_COMM\_WORLD, &world\_rank);

    MPI\_Comm\_size(MPI\_COMM\_WORLD, &world\_size);

    if (argc < 3) {

        if (world\_rank == 0) {

            std::cerr << "Usage: " << argv[0] << " <grid\_dimension\_internal> <num\_epochs>" << std::endl;

        }

        MPI\_Finalize();

        return 1;

    }

    int N\_global = 0;

    int epochs = 0;

    if (world\_rank == 0) {

        try {

            N\_global = std::stoi(argv[1]);

            epochs = std::stoi(argv[2]);

        } catch (const std::invalid\_argument& ia) {

            std::cerr << "Rank 0: Invalid argument for N or epochs: " << ia.what() << std::endl;

            N\_global = -1;

        } catch (const std::out\_of\_range& oor) {

            std::cerr << "Rank 0: Out of Range error for N or epochs: " << oor.what() << std::endl;

            N\_global = -1;

        }

    }

    MPI\_Bcast(&N\_global, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

    MPI\_Bcast(&epochs, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

    if (N\_global <= 0 || epochs < 0) {

        if (world\_rank == 0 && N\_global > 0 && epochs >=0) {

             if (N\_global <=0) std::cerr << "Grid dimension must be > 0." << std::endl;

             if (epochs < 0) std::cerr << "Number of epochs must be >= 0." << std::endl;

        }

        MPI\_Finalize();

        return 1;

    }

    int rows\_per\_proc\_base = N\_global / world\_size;

    int remainder\_rows = N\_global % world\_size;

    int local\_N\_computational\_rows = rows\_per\_proc\_base + (world\_rank < remainder\_rows ? 1 : 0);

    int local\_grid\_actual\_rows = local\_N\_computational\_rows + 2;

    int local\_grid\_actual\_cols = N\_global + 2;

    std::vector<std::vector<double>> local\_A(local\_grid\_actual\_rows, std::vector<double>(local\_grid\_actual\_cols));

    std::vector<std::vector<double>> local\_A\_new(local\_grid\_actual\_rows, std::vector<double>(local\_grid\_actual\_cols));

    for (int i = 0; i < local\_grid\_actual\_rows; ++i) {

        for (int j = 0; j < local\_grid\_actual\_cols; ++j) {

            if (j == 0 || j == N\_global + 1) {

                local\_A[i][j] = 1.0;

            }

            else if (world\_rank == 0 && i == 0) {

                local\_A[i][j] = 1.0;

            }

            else if (world\_rank == world\_size - 1 && i == local\_N\_computational\_rows + 1) {

                local\_A[i][j] = 1.0;

            }

            else {

                local\_A[i][j] = 0.0;

            }

        }

    }

    local\_A\_new = local\_A;

    if (world\_rank == 0) {

        std::cout << "MPI run with " << world\_size << " processes." << std::endl;

        std::cout << "Global grid size (internal): " << N\_global << "x" << N\_global << std::endl;

        std::cout << "Epochs: " << epochs << std::endl;

    }

    MPI\_Barrier(MPI\_COMM\_WORLD);

    double startTime = MPI\_Wtime();

    for (int epoch = 0; epoch < epochs; ++epoch) {

        int top\_neighbor = (world\_rank == 0) ? MPI\_PROC\_NULL : world\_rank - 1;

        int bottom\_neighbor = (world\_rank == world\_size - 1) ? MPI\_PROC\_NULL : world\_rank + 1;

        MPI\_Sendrecv(local\_A[1].data(), local\_grid\_actual\_cols, MPI\_DOUBLE, top\_neighbor, 0,

                     local\_A[0].data(), local\_grid\_actual\_cols, MPI\_DOUBLE, top\_neighbor, 1,

                     MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

        MPI\_Sendrecv(local\_A[local\_N\_computational\_rows].data(), local\_grid\_actual\_cols, MPI\_DOUBLE, bottom\_neighbor, 1,

                     local\_A[local\_N\_computational\_rows + 1].data(), local\_grid\_actual\_cols, MPI\_DOUBLE, bottom\_neighbor, 0,

                     MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

        if (local\_N\_computational\_rows == 0) {

             local\_A = local\_A\_new;

             continue;

        }

        for (int i = 1; i <= local\_N\_computational\_rows; ++i) {

            for (int j = 1; j <= N\_global; ++j) {

                local\_A\_new[i][j] = (local\_A[i - 1][j] + local\_A[i + 1][j] +

                                     local\_A[i][j - 1] + local\_A[i][j + 1]) / 4.0;

            }

        }

        local\_A = local\_A\_new;

    }

    MPI\_Barrier(MPI\_COMM\_WORLD);

    double endTime = MPI\_Wtime();

    if (world\_rank == 0) {

        std::cout << "Time taken: " << std::fixed << std::setprecision(5) << endTime - startTime << " sec" << std::endl;

    }

    if (N\_global <= 4 && epochs <= 5) {

        MPI\_Barrier(MPI\_COMM\_WORLD);

        for(int r=0; r<world\_size; ++r) {

            if (world\_rank == r) {

                for(volatile int k=0; k<world\_rank \* 1000000; ++k);

                std::cout << "--- Rank " << world\_rank << " final local\_A (comp rows: " << local\_N\_computational\_rows << ") ---" << std::endl;

                if (local\_N\_computational\_rows > 0) {

                    for (int i = 0; i < local\_grid\_actual\_rows; ++i) {

                        std::cout << "  R" << world\_rank << " L" << i << ": ";

                        for (int j = 0; j < local\_grid\_actual\_cols; ++j) {

                            std::cout << std::fixed << std::setprecision(2) << local\_A[i][j] << " ";

                        }

                        std::cout << std::endl;

                    }

                } else {

                    std::cout << "  (No computational rows for this rank)" << std::endl;

                }

            }

            MPI\_Barrier(MPI\_COMM\_WORLD);

        }

    }

    MPI\_Finalize();

    return 0;

}

* **Compilation (CMakeLists.txt)**  
  A single CMakeLists.txt in ~/LaplaceSolver/ was used to manage all three builds.

cmake\_minimum\_required(VERSION 3.10)

project(LaplaceSolvers LANGUAGES CXX)

set(CMAKE\_CXX\_STANDARD 11)

set(CMAKE\_CXX\_STANDARD\_REQUIRED ON)

add\_executable(laplace\_sequential src/laplace\_sequential.cpp)

find\_package(OpenMP REQUIRED)

add\_executable(laplace\_openmp src/laplace\_openmp.cpp)

if(OpenMP\_FOUND)

set\_property(TARGET laplace\_openmp APPEND PROPERTY COMPILE\_FLAGS

"${OpenMP\_CXX\_FLAGS}")

set\_property(TARGET laplace\_openmp APPEND PROPERTY LINK\_FLAGS

"${OpenMP\_CXX\_FLAGS}")

else()

message(WARNING "OpenMP not found, laplace\_openmp target might not

parallelize correctly.")

endif()

find\_package(MPI REQUIRED)

add\_executable(laplace\_mpi src/laplace\_mpi.cpp)

if(MPI\_FOUND)

target\_link\_libraries(laplace\_mpi PRIVATE MPI::MPI\_CXX)

else()

message(WARNING "MPI not found, laplace\_mpi target might not compile/link

correctly.")

endif()

# Optional: Add optimization flags to all targets if desired

# foreach(target laplace\_sequential laplace\_openmp laplace\_mpi)

# target\_compile\_options(${target} PRIVATE -O2 -Wall)

# endforeach()

# Optional install rule

# install(TARGETS laplace\_sequential laplace\_openmp laplace\_mpi DESTINATION

# bin)  
  
.Cmake

Compilation commands (from ~/LaplaceSolver/build):

cmake ..

make clean

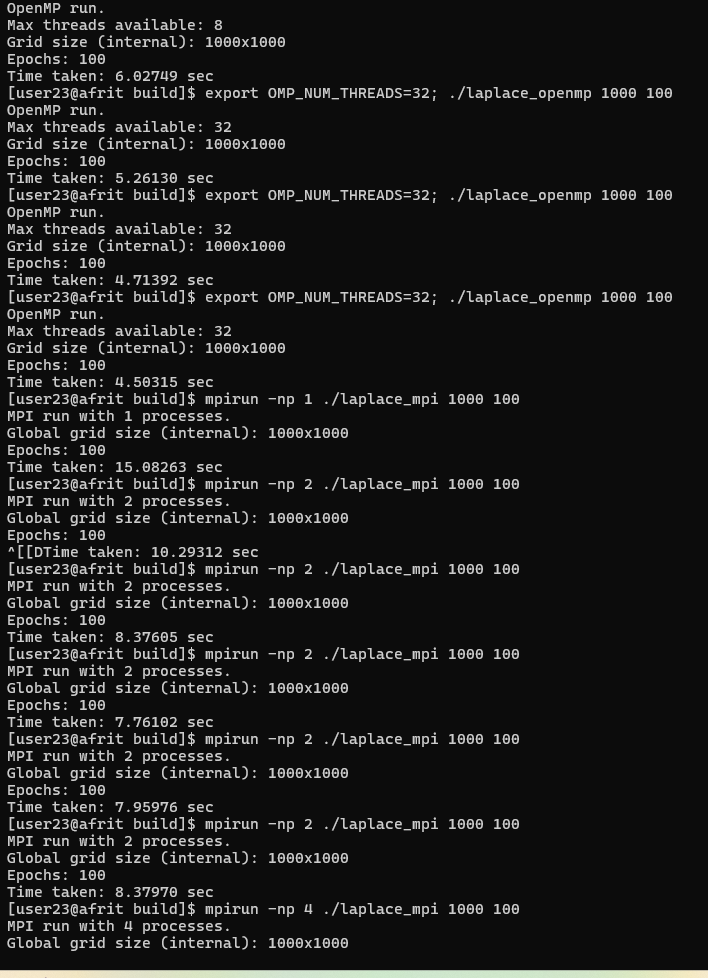
make

**2.2.3. Experimental Setup**

* **Parameters:**
  + Internal Grid Sizes (N\_global): 1000x1000, 2000x2000.
  + Epochs: 100 for all performance runs.
  + Processes/Threads (P): 1, 2, 4, 8, 16, 32.
* **Execution Environment:** All presented performance data was collected by running the executables on the afrit head node.
  + OpenMP: export OMP\_NUM\_THREADS=<P>; ./laplace\_openmp <N> <epochs>
  + MPI: mpirun -np <P> ./laplace\_mpi <N> <epochs>
* **Data Collection:** Each configuration was run multiple times (typically 3-5, single runs presented for N=2000 due to time), and average times were used for MPI where multiple runs were captured.

**Caveat on Execution Environment:** It is acknowledged that performance benchmarks should ideally be run on dedicated compute nodes, not a shared head node like afrit. The results presented here are preliminary and may be influenced by head node load and characteristics. For final analysis, re-running on a compute node (e.g., compute-0-0.local with P up to 16) is strongly recommended.

**2.3. Results**

**2.3.1. Correctness**  
For small grid sizes (e.g., N=2, E=2 for MPI; N=3, E=2 for Sequential/OpenMP), the output values of the computed grids were visually inspected and found to be consistent across the sequential, OpenMP, and MPI implementations, indicating correct parallelization logic.  
 

**2.3.2. Performance Data**  
The following table summarizes the average execution times and calculated speedups. Speedup is relative to MPI P=1 for N=1000 (15.08s) and OpenMP P=1 for N=2000 (93.60s).

**2.3.3. Performance Graphs**  
A graph with a line and a line

AI-generated content may be incorrect.A graph with a line

AI-generated content may be incorrect.A graph with a line

AI-generated content may be incorrect.A graph of a number of process

AI-generated content may be incorrect.

* Figure II.3: Execution Time vs. P (N=1000)
* Figure II.4: Speedup vs. P (N=1000)
* Figure II.5: Execution Time vs. P (N=2000)
* Figure II.6: Speedup vs. P (N=2000)

**2.4. Discussion of Results (Preliminary - based on afrit data)**

* **General Trends:** For both N=1000 and N=2000, increasing the number of processes/threads generally decreased execution time and increased speedup for both OpenMP and MPI, up to a certain point.
* **OpenMP Scaling:**
  + For N=1000, OpenMP showed good initial speedup (up to P=4), but performance gains diminished significantly beyond P=8, with P=16 and P=32 offering only marginal improvements over P=8. The maximum speedup was ~3.1x with 32 threads.
  + For N=2000, OpenMP scaled better, achieving a speedup of ~5.7x at P=16 and ~6.5x at P=32. The larger problem size provided more work for the threads, reducing the relative impact of overheads.
* **MPI Scaling (Single-node afrit):**
  + For N=1000, MPI scaled well up to P=4, showed an anomaly at P=8 (slower than P=4), then scaled very well at P=16 and P=32, achieving a speedup of ~7.6x. The P=8 anomaly is likely due to contention or process scheduling issues on the shared afrit node.
  + For N=2000, MPI showed consistent scaling, achieving a speedup of ~9.5x at P=16 and ~13.7x at P=32. This is significantly better than OpenMP at higher core counts for this larger problem size.
* **MPI vs. OpenMP (on afrit):**
  + At lower core counts (P <= 8 for N=1000, P <= 8 for N=2000), OpenMP and MPI performance was comparable.
  + At higher core counts (P >= 16), MPI consistently outperformed OpenMP for both problem sizes on afrit. This suggests that for this application, MPI's explicit domain decomposition and message passing may lead to better resource utilization or less contention than OpenMP's shared-memory model at higher concurrencies on this particular node.
* **Performance Bottlenecks:**
  + The plateauing of OpenMP speedup suggests that shared memory bandwidth, cache contention, or synchronization overheads become limiting factors as more threads are added.
  + The MPI anomaly at N=1000, P=8 indicates potential issues with how MPI processes were managed or contended for resources on afrit at that specific concurrency.
  + For MPI, while scaling better, the speedups are still sub-linear, indicating that communication overhead (halo exchange) is non-negligible.
  + Running on afrit is a major bottleneck itself for obtaining clean performance data.

**3. Challenges Faced and Solutions**

1. **Initial Environment Setup:** Adding the correct CMake path to .bashrc required careful editing after an initial error (-bash: 1: command not found) due to an accidental line number being copied. Solution: Corrected .bashrc by removing the extraneous characters.
2. **vi Pasting Issues:** Initial attempts to paste code into vi resulted in malformed files. Solution: Used :%d to clear the file and re-pasted carefully, or recommended scp as an alternative.
3. **CMake Configuration Errors:** "Cannot find source file" errors occurred due to misplacing CMakeLists.txt inside the src directory instead of the project root. Solution: Moved CMakeLists.txt to the project root and ensured paths in add\_executable were correct relative to it.
4. **Compilation Errors with GCC 4.4.7:** The older GCC version required explicit inclusion of <cstddef> to resolve errors related to ptrdiff\_t and streamsize not being declared. Solution: Added #include <cstddef> at the beginning of C++ source files.
5. **OpenMP Linking with CMake:** Ensuring OpenMP flags were correctly applied with GCC 4.4.7. The set\_property(TARGET ... APPEND PROPERTY COMPILE\_FLAGS "${OpenMP\_CXX\_FLAGS}") method was found to be more robust than the modern OpenMP::OpenMP\_CXX target for this older compiler.
6. **Lack of Scheduler Information:** Standard scheduler commands (sinfo, qstat) were not found on afrit, making direct assessment of compute node load difficult for Part I. Solution: Inferred busy status based on other available information (MOTD, node heterogeneity).
7. **Performance Testing on Head Node:** All performance data was collected on afrit. This is a known limitation. Solution for report: Clearly state this limitation and recommend re-running on compute nodes.

**4. Conclusion**

This project successfully explored aspects of the SINES HPC cluster and implemented parallel versions of a Laplace solver. Part I revealed a heterogeneous cluster environment with nodes of varying specifications and availability. The probe\_nodes.sh script and lscpu command were effective in gathering this information.

Part II demonstrated the successful implementation and correctness of sequential, OpenMP, and MPI versions of the Laplace solver. Preliminary performance analysis on the afrit head node indicated that:

* Both OpenMP and MPI can provide significant speedups over sequential execution.
* MPI generally showed better scalability and absolute performance than OpenMP at higher core counts (P>=16) for the tested problem sizes on afrit.
* OpenMP performance tended to plateau more quickly, likely due to shared memory bottlenecks.

The primary limitation of the performance study is the use of the afrit head node. For conclusive results, future work should involve re-running these benchmarks on dedicated compute nodes, varying P up to the core count of those nodes, and potentially exploring multi-node MPI performance. Nevertheless, this project provided valuable hands-on experience with HPC tools, parallel programming models, and performance analysis methodologies.