Day20

Table of Contents

1. Scripts

1.1. compile script

```
#!/bin/sh

#source /opt/ohpc/pub/apps/spack/share/spack/setup-env.sh
#spack load gcc/5i5y5cb
#spack load openmpi/c7kvqyq
source ~/git/spack/share/spack/setup-env.sh
spack load openmpi

inputFile=$1
outputFile="${1%.*}.out"      # extract the name of the file without extension and adding extension .out
#cmd=`mpicc $inputFile -o $outputFile`
cmd="mpicc $inputFile -o $outputFile -lm -fopenmp"     # running code using MPI
echo "------------------------------------------------------------------"
echo "Command executed: $cmd"
echo "------------------------------------------------------------------"
$cmd
```

```
echo "Compilation successful. Check at $outputFile"
echo "------------------------------------------------------------------"
```

1.2. run script

```
#!/bin/sh

#source /opt/ohpc/pub/apps/spack/share/spack/setup-env.sh
#spack load gcc/5i5y5cbc
source ~/git/spack/share/spack/setup-env.sh
spack load openmpi

cmd="mpirun -np $2 $1"
echo "------------------------------------------------------------------"
echo "Command executed: $cmd"
echo "------------------------------------------------------------------"
echo "##################################################################"
echo "##########                    OUTPUT                    ##########"
echo "##################################################################"
echo
mpirun -np $2 $1
echo
echo "##################################################################"
echo "##########                    DONE                      ##########"
echo "##################################################################"
```

2. Speedup, Amdahl's Law, and Performance Metrics in MPI

2.1. Performance Metrics

When measuring the performance of MPI programs, several metrics are commonly used:

1. Execution Time:
    - The total time taken for the program to run.
2. Speedup:
    - As described above, it measures how much faster the parallel program runs compared to the serial version.

3. Efficiency:
    - Efficiency measures how effectively the processors are being utilized. It's defined as the speedup divided by the number of processors:

$$\text{Efficiency}(E) = \frac{S(p)}{p} = \frac{T_s}{p \times T_p}$$

4. Scalability:
    - Scalability refers to how well a parallel algorithm performs as the number of processors increases. It is evaluated using strong scaling and weak scaling:
        - Strong Scaling: Fixing the problem size and increasing the number of processors.
        - Weak Scaling: Increasing both the problem size and the number of processors proportionally.
5. Latency and Bandwidth:
    - Latency: The time taken to send a message from one process to another.
    - Bandwidth: The rate at which data can be transmitted.

2.2. Speedup

Speedup measures the improvement in performance of a parallel algorithm over its serial counterpart. It's calculated using the formula:

$$\text{Speedup}(S) = \frac{T_s}{T_p}$$

where:

- T_s is the execution time of the serial algorithm.
- T_p is the execution time of the parallel algorithm using p processors.

2.3. Amdahl's Law

Amdahl's Law provides a theoretical limit on the speedup that can be achieved by parallelizing a portion of an algorithm. It states that if a fraction f of the algorithm is inherently serial, then the maximum speedup S using p processors is:

$$S(p) = \frac{1}{f + \frac{1-f}{p}}$$

where:

- f is the fraction of the algorithm that is serial (cannot be parallelized).
- $1 - f$ is the fraction that can be parallelized.
- p is the number of processors.

As p approaches infinity, the speedup approaches:

$$S_{\max} = \frac{1}{f}$$

This demonstrates that the speedup is limited by the serial portion of the algorithm.

2.4. Measuring Performance in MPI

To measure the performance of MPI programs, you can use the following methods:

1. Timing Functions:
    - Use `MPI_Wtime()` to measure the wall-clock time before and after the parallel section of the code.

    ```c double start_time, end_time; start_time = MPI_Wtime();

    // Parallel code

    end_time = MPI_Wtime(); printf("Execution time: %f seconds\n", end_time - start_time);
    ```

2. Profiling Tools Use profiling tools like gprof, TAU, and VTune to analyze the performance of MPI programs.
3. MPI Profiling Interface (PMPI) MPI provides a profiling interface (PMPI) that allows you to intercept and measure MPI calls.
4. Performance Analysis Tools: Tools like Paraver, Vampir, HPCToolkit, and Scalasca can be used to

visualize and analyze the performance of MPI programs.

3. Benchmarking in MPI

3.1. Benchmarking

Benchmarking is the process of measuring the performance of a system or application to evaluate its efficiency and effectiveness. In the context of MPI, benchmarking involves assessing the performance of MPI operations and parallel applications.

3.2. Microbenchmarking

Microbenchmarking focuses on measuring the performance of individual operations or small code segments. In MPI, microbenchmarking typically involves assessing the performance of basic MPI operations such as `MPI_Send`, `MPI_Recv`, `MPI_Bcast`, etc.

3.3. Macrobenchmarking

Macrobenchmarking evaluates the performance of entire applications or larger code segments. It considers the overall performance and scalability of parallel applications, including computation, communication, and I/O.

3.4. How Benchmarking is Done

1. **Timing Functions**: Use functions like `MPI_Wtime` to measure execution time.
2. **Profiling Tools**: Utilize profiling tools to analyze performance.
3. **Performance Analysis Tools**: Use tools for in-depth performance analysis and visualization.

3.5. Benchmarking Tools

1. **OSU Micro-Benchmarks (OMB)**: A suite of benchmarks for measuring MPI performance, focusing on latency, bandwidth, and collective operations.
2. **Intel MPI Benchmarks (IMB)**: A set of benchmarks for evaluating the performance of MPI operations.

3. **HPC Challenge (HPCC)**: A benchmark suite that measures the performance of HPC systems.
4. **SPEC MPI**: A benchmark suite designed to evaluate the performance of MPI-parallel, floating point, compute-intensive applications.

4. MPI Topology

MPI topologies provide a way to organize the processes in a communicator in a logical structure, which can enhance the performance of parallel applications by optimizing communication patterns.

4.1. Types of Topologies

1. **Cartesian Topologies**: Processes are arranged in a grid-like structure.
2. **Graph Topologies**: Processes are arranged in an arbitrary graph structure.

4.2. Cartesian Topologies

Cartesian topologies are useful for problems that have a natural grid structure.

4.2.1. Creating a Cartesian Topology

The function `MPI_Cart_create` is used to create a Cartesian topology.

1. Syntax

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims, int *periods, int reorder, MPI_Comm *comm_cart);
```

2. Parameters
    - `comm_old`: The original communicator.
    - `ndims`: Number of dimensions of the Cartesian grid.
    - `dims`: Array specifying the number of processes in each dimension.
    - `periods`: Array specifying whether the grid is periodic (wrap-around connections) in each dimension (logical array).
    - `reorder`: Ranking may be reordered (1) or not (0).
    - `comm_cart`: New communicator with Cartesian topology.

3. Example Code

```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int dims[2] = {0, 0};
    MPI_Dims_create(size, 2, dims); // Automatically create a balanced 2D grid
    int periods[2] = {0, 0}; // No wrap-around connections
    MPI_Comm cart_comm;
    MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 0, &cart_comm);

    int coords[2];
    MPI_Cart_coords(cart_comm, rank, 2, coords);
    printf("Rank %d has coordinates (%d, %d)\n", rank, coords[0], coords[1]);

    // Cleanup
    MPI_Comm_free(&cart_comm);

    MPI_Finalize();
    return 0;
}
```

```
bash compile.sh mpi_cart.c
```

```
----------------------------------------------------------------------
Command executed: mpicc mpi_cart.c -o mpi_cart.out -lm -fopenmp
----------------------------------------------------------------------
Compilation successful. Check at mpi_cart.out
----------------------------------------------------------------------
```

```
bash run.sh ./mpi_cart.out 2
```

```
----------------------------------------------------------------------
```

```
Command executed: mpirun -np 2 ./mpi_cart.out
--------------------------------------------------------------------
######################################################################
##########                    OUTPUT                    ##########
######################################################################

Rank 0 has coordinates (0, 0)
Rank 1 has coordinates (1, 0)

######################################################################
##########                     DONE                     ##########
######################################################################
```

4.3. Graph Topologies

Graph topologies are useful for problems with irregular communication patterns.

4.3.1. Creating a Graph Topology

The function `MPI_Graph_create` is used to create a graph topology.

1. Syntax

```c
int MPI_Graph_create(MPI_Comm comm_old, int nnodes, int *index, int *edges, int reorder, MPI_Comm *comm_graph);
```

2. Parameters
    - `comm_old`: The original communicator.
    - `nnodes`: Number of nodes in the graph.
    - `index`: Array of integers describing the graph structure.
    - `edges`: Array of integers describing the graph structure.
    - `reorder`: Ranking may be reordered (1) or not (0).
    - `comm_graph`: New communicator with graph topology.

4.4. Cartesian Topology Functions

- `MPI_Cart_create`: Creates a Cartesian topology.
- `MPI_Cart_coords`: Determines the coordinates of a process in the Cartesian topology.

- `MPI_Cart_rank`: Determines the rank of a process given its coordinates.
- `MPI_Cart_shift`: Determines the source and destination ranks for shifts in a Cartesian topology.

4.5. Graph Topology Functions

- `MPI_Graph_create`: Creates a graph topology.
- `MPI_Graph_neighbors_count`: Determines the number of neighbors of a process.
- `MPI_Graph_neighbors`: Determines the neighbors of a process.
- `MPI_Graphdims_get`: Retrieves the number of nodes and edges in the graph topology.
- `MPI_Graph_get`: Retrieves the graph structure.

4.6. Benefits of Using MPI Topologies

1. **Optimized Communication**: By organizing processes in a logical structure, communication can be optimized for better performance.
2. **Simplified Programming**: Topologies simplify the management of process coordinates and neighbor relationships.
3. **Improved Scalability**: Topologies can help applications scale more efficiently by reducing communication overhead.

5. MPI I/O:

5.1. Introduction to MPI I/O

- **What is MPI I/O?**
  - A set of routines in MPI designed for parallel file operations.
  - Supports both individual and collective I/O.
- **Why Use MPI I/O?**
  - Facilitates efficient data access in distributed systems.
  - Allows multiple processes to read/write to a single file concurrently.
  - Avoids the need for each process to manage separate files.

5.2. Key Features of MPI I/O

- **File Partitioning:** Files can be partitioned into non-overlapping chunks for different processes.
- **Collective I/O:** Multiple processes cooperate to perform I/O, reducing overhead.
- **Noncontiguous I/O:** Supports strided or indexed data access.
- **Data Representation:** Supports native, internal, and external32 data representations for portability.

5.3. MPI I/O Operations

5.3.1. File Open and Close

```
MPI_File_open(MPI_Comm comm, const char *filename, int amode, MPI_Info info, MPI_File *fh);
MPI_File_close(MPI_File *fh);
```

- **amode:** File access mode (e.g., `MPI_MODE_RDONLY`, `MPI_MODE_WRONLY`, `MPI_MODE_RDWR`).
- **info:** Hints for optimization (e.g., striping size, buffering).

5.3.2. Data Access

- **Read and Write Operations:**
  - Individual:

```
MPI_File_read(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status);
MPI_File_write(MPI_File fh, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status);
```

- Collective:

```
MPI_File_read_all(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status);
MPI_File_write_all(MPI_File fh, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status);
```

- **Position-Based Access:**

```
MPI_File_read_at(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status);
MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status);
```

5.4. Example Programs

5.4.1. Example 1: Writing and Reading a Simple File

```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char **argv) {
    MPI_File fh;
    int rank, size, buf[10];

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Initialize buffer with rank-specific data
    for (int i = 0; i < 10; i++) buf[i] = rank * 10 + i;

    // Open file for writing
    MPI_File_open(MPI_COMM_WORLD, "datafile.dat", MPI_MODE_CREATE | MPI_MODE_WRONLY, MPI_INFO_NULL, &fh);

    // Write data collectively
    MPI_File_write_at(fh, rank * 10 * sizeof(int), buf, 10, MPI_INT, MPI_STATUS_IGNORE);

    MPI_File_close(&fh);
    MPI_Finalize();

    return 0;
}
```

5.5. Best Practices

- Use collective I/O operations to optimize file access patterns.
- Minimize the number of file opens/closes to reduce overhead.
- Use file views to efficiently partition data among processes.
- Employ the `MPI_Info` object to pass hints for performance tuning.

5.6. Debugging MPI I/O

- Enable I/O profiling with tools such as:
  - **Darshan:** Captures I/O behavior of MPI applications.
  - **VampirTrace:** Visualizes I/O activity.
- Verify data consistency using checksum utilities.

5.7. Summary

- MPI I/O provides robust support for parallel file operations.
- Key concepts include collective I/O, file views, and data mappings.
- Efficient use of MPI I/O can significantly improve the scalability of parallel applications.

6. OpenMP Offloading

6.1. Introduction to OpenMP Offloading

- OpenMP offloading enables the execution of code on accelerators such as GPUs, FPGAs, or other specialized devices.
- Offloading improves performance by leveraging accelerators for computationally intensive tasks.
- Supported devices include:
  - GPUs (NVIDIA, AMD, Intel)
  - FPGAs
  - Many-core processors

6.2. Why Use Offloading?

- Accelerators provide high parallelism.
- Ideal for data-parallel and computationally heavy tasks.

6.3. Key Concepts

- **Target Regions:** Code to be executed on the device, specified using `#pragma omp target`.
- **Mapping Data:** Explicitly transfer data between host and device using `map` clauses.
  - `to`: Copy data from the host to the device.
  - `from`: Copy data from the device to the host.

    - `tofrom`: Bi-directional transfer.
    - `alloc`: Allocate memory on the device without data transfer.
- **Device Selection:** Specify the device using the `device` clause or environment variables.

6.4. Syntax

```
#pragma omp target [clauses]
{
    // Code to execute on the device
}
```

- **Clauses:**
    - `map`: Specifies data transfer.
    - `device`: Specifies the device to offload to.
    - `if`: Conditional offloading.

6.5. Data Mapping

- **Automatic Data Mapping:** Scalars are automatically shared between host and device.
- **Explicit Data Mapping:**

```
#pragma omp target map(to: a, b) map(from: c)
{
    c = a + b;
}
```

6.6. Example Programs

6.6.1. Example 1: Vector Addition

```
#include <stdio.h>
#include <omp.h>

#define N 1000
```

```
int main() {
    int a[N], b[N], c[N];

    // Initialize arrays
    for (int i = 0; i < N; i++) {
        a[i] = i;
        b[i] = N - i;
    }

    // Offload computation to the device
    #pragma omp target map(to: a, b) map(from: c)
    {
        for (int i = 0; i < N; i++) {
            c[i] = a[i] + b[i];
        }
    }

    // Verify results
    for (int i = 0; i < N; i++) {
        if (c[i] != N) {
            printf("Error at index %d: %d != %d\n", i, c[i], N);
            return -1;
        }
    }

    printf("Computation successful!\n");
    return 0;
}
```

6.6.2. Example 2: Matrix Multiplication

```
#include <stdio.h>
#include <omp.h>

#define N 512

int main() {
    int A[N][N], B[N][N], C[N][N];

    // Initialize matrices
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            A[i][j] = i + j;
            B[i][j] = i - j;
            C[i][j] = 0;
```

```
        }
    }

    // Offload computation to the device
    #pragma omp target map(to: A, B) map(from: C)
    {
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                for (int k = 0; k < N; k++) {
                    C[i][j] += A[i][k] * B[k][j];
                }
            }
        }
    }

    printf("Matrix multiplication completed!\n");
    return 0;
}
```

6.7. Advanced Features

- **Target Teams:** Create parallel regions on the device.

```
#pragma omp target teams distribute parallel for
for (int i = 0; i < N; i++) {
    // Workload
}
```

- **Target Update:** Explicitly update data between host and device.

```
#pragma omp target update to(data)
#pragma omp target update from(data)
```

- **Offloading Control with Environment Variables:**
    - `OMP_DEFAULT_DEVICE`: Specifies the default device.
    - `OMP_NUM_TEAMS`: Controls the number of teams.

6.8. Best Practices

- Minimize data transfer between host and device.
- Use `map` clauses effectively for efficient memory management.
- Use `target teams` for hierarchical parallelism on GPUs.
- Optimize kernel performance with thread and team configurations.

6.9. Summary

- OpenMP offloading bridges the gap between multi-core CPUs and accelerators.
- Key constructs include `target`, `teams`, and `map`.
- Use hierarchical parallelism (`teams` and `parallel`) for efficient utilization of accelerators.
- Optimize memory transfers and minimize host-device interactions.

7. MPI_Barrier

```c
#include <mpi.h>
#include <stdio.h>
#include <unistd.h>

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (rank == 0) {
        printf("Process %d: Preparing data...\n", rank);
        sleep(2); // Simulate data preparation
    }

    printf("Process %d: Waiting at the barrier.\n", rank);
    MPI_Barrier(MPI_COMM_WORLD);

    printf("Process %d: Proceeding to next phase.\n", rank);

    MPI_Finalize();
    return 0;
}
```

```
bash compile.sh barrier.c
```

```
---------------------------------------------------------------------
Command executed: mpicc barrier.c -o barrier.out -lm -fopenmp
---------------------------------------------------------------------
Compilation successful. Check at barrier.out
---------------------------------------------------------------------
```

```
bash run.sh ./barrier.out 5
```

```
---------------------------------------------------------------------
Command executed: mpirun -np 5 ./barrier.out
---------------------------------------------------------------------
#####################################################################
##########                    OUTPUT                       ##########
#####################################################################

Process 1: Waiting at the barrier.
Process 2: Waiting at the barrier.
Process 4: Waiting at the barrier.
Process 3: Waiting at the barrier.
Process 0: Preparing data...
Process 0: Waiting at the barrier.
Process 0: Proceeding to next phase.
Process 1: Proceeding to next phase.
Process 3: Proceeding to next phase.
Process 2: Proceeding to next phase.
Process 4: Proceeding to next phase.

#####################################################################
##########                     DONE                        ##########
#####################################################################
```

Author: Abhishek Raj
Created: 2025-01-13 Mon 17:44