High-Performance Computing Lab **Institute of Computing**

Student: Lorenzo Varese Discussed with: None

Solution for Project 5

This project will introduce you a parallel space solution of a nonlinear PDE using MPI.

1. Task 1 - Initialize and finalize MPI [5 Points]

In the file `main.cpp`, the MPI environment is initialized using the `MPI_Init_thread` function which sets up the necessary infrastructure for MPI processes to communicate. The program then queries for the rank of the MPI process (`MPI_Comm_rank`) and the total number of MPI processes (`MPI_Comm_size`) participating in the communicator `MPI_COMM_WORLD`.

Following the MPI initialization, the main computational domain is partitioned and distributed among the available MPI ranks. This is performed by the `domain.init` method, which takes into account the rank and size information to allocate the subdomains appropriately.

After executing the main computational work, the MPI environment is finalized with the `MPI_Finalize` call. This call signifies that the MPI implementation can release all resources associated with it; this is essential for clean shutdown procedures in any MPI application.

```
// Initialize MPI
int thread_level;
MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED, &thread_level);

// Get rank and the number of ranks
int mpi_rank, mpi_size;
MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);
MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);

// Finalize MPI
MPI_Comm_free(&domain.comm_cart);
MPI_Finalize();
```

This code snippet illustrates the typical lifecycle of an MPI application where initialization and finalization of the MPI environment form the outermost structure within the `main` function.

2. Task 2 - Create a Cartesian topology [10 Points]

For the mini-application utilizing a 2D grid, each MPI rank operates on a subset of the grid, requiring the creation of a 2D domain decomposition. This decomposition is contingent on the number of MPI ranks available. The file `data.cpp` encapsulates the logic for establishing the domain decomposition and the Cartesian topology.

Initially, the `MPI_Dims_create` function computes an optimal partitioning of the MPI processes into a 2D grid, taking into consideration the total number of MPI processes. Following this, a

2D Cartesian communicator is constructed using `MPI_Cart_create`, with non-periodic boundary conditions to facilitate communication between subdomains.

The coordinates of each rank within this Cartesian topology are determined by `MPI_Cart_coords`, which allows for the identification of each process's position within the grid. Furthermore, the `MPI_Cart_shift` function is utilized to identify the neighboring ranks in each cardinal direction, enabling the setup of communication channels for data exchange between neighboring subdomains.

Finally, the bounding box for each subdomain is calculated, ensuring that the entire domain is covered even if the global grid dimensions do not divide evenly among all subdomains.

2.1. Code Snippet

```
// Initialize subdomain
void SubDomain::init(int mpi_rank, int mpi_size, Discretization &discretization)
{
    // ... [code that sets up domain decomposition] ...

    // Create a 2D non-periodic Cartesian topology
    MPI_Comm comm_cart;
    // ... [code to create and use Cartesian topology] ...

    // Identify coordinates of the current rank in the domain grid
    // ... [code to identify coordinates] ...

    // Identify neighbours of the current rank
    // ... [code to identify neighbours] ...
}
```

This abstraction allows each MPI process to be aware of its unique position within the global computation and to communicate efficiently with its immediate neighbors, adhering to the domain decomposition strategy.

3. Task 3 - Change linear algebra functions [5 Points]

The implementation of the dot product and the norm computation functions within `linalg.cpp` file has been modified to perform collective operations over all ranks. The necessity of altering only these two functions, and not others, stems from their role in iterative solvers which require global communication. The dot product and norm are reduction operations that, in a distributed memory environment, necessitate the aggregation of values across all ranks to ensure consistency and correctness of the computations.

- The `ss_dot` function now includes an MPI collective operation, `MPI_Allreduce`, which performs a global reduction operation to find the sum of products across all processes in the MPI communicator. The result is then broadcasted to all processes.
- The `ss_norm2` function similarly employs `MPI_Allreduce` to compute the sum of squares of the vector elements across all ranks, followed by the square root of the reduced sum to obtain the norm.

Reduction operations are critical in parallel programs to ensure that all processes agree on certain values that are the result of operations involving data distributed across processes. The dot product and norm are fundamental operations in many iterative methods, including the Conjugate Gradient method used in this code, where a global view of these values is essential for the algorithm to function correctly.

```
// Computes the inner product of x and y
double ss_dot(Field const &x, Field const &y)
{
    // ... [code before MPI call] ...
    MPI_Allreduce(&result, &result_reduction, 1, MPI_DOUBLE, MPI_SUM, data::
   domain.comm_cart);
    return result_reduction;
}

// Computes the 2-norm of x
double ss_norm2(Field const &x)
{
    // ... [code before MPI call] ...
    MPI_Allreduce(&result, &result_reduction, 1, MPI_DOUBLE, MPI_SUM, data::
   domain.comm_cart);
    return sqrt(result_reduction);
}
```

The addition of collective operations in these functions is intended to ensure that the values computed reflect the collective state of the distributed system, which is crucial for the convergence and correctness of parallel iterative solvers.

4. Task 4 - Exchange ghost cells [45 Points]

In a distributed memory parallel computation, each process owns a part of the global computational grid. To perform stencil computations correctly, it is essential that each process exchanges boundary data, known as ghost cells, with its neighbors. This exchange must occur before each iteration to ensure data consistency across the computational domain.

The file `operators.cpp` implements the ghost cell exchange as follows:

- Point-to-point non-blocking communication is used to exchange ghost cells with neighboring processes in all directions (north, south, east, and west).
- Non-blocking sends (`MPI_Isend`) and receives (`MPI_Irecv`) are employed to facilitate overlap between computation and communication, aiming to minimize idle time and improve parallel efficiency.
- The exchange includes packing data into buffers (`buffN`, `buffS`, `buffE`, `buffW`) before sending and unpacking received data into ghost cell arrays (`bndN`, `bndS`, `bndE`, `bndW`).

Refer to Figure 1 in the task description for a visual representation of the ghost cell exchange process. The diagram illustrates how the northern and southern ghost cells are copied to buffers and then sent to the corresponding neighbors, with the reverse process for receiving the ghost cells.

The existence of neighbors is determined by ensuring the rank is non-negative. MPI ranks are non-negative integers, so a negative rank implies a non-existent neighbor, typically at the domain's boundary.

A counter, `num_requests`, tracks the quantity of non-blocking communications that have been initiated, facilitating the subsequent synchronization call `MPI_Waitall`.

While awaiting the completion of data transfers, interior grid points are computed in parallel, leveraging OpenMP directives.

`MPI_Waitall` is called to guarantee that all asynchronous communication operations conclude before utilizing the ghost cell data. This function synchronizes the program, ensuring that all processes have the updated data necessary for accurate computation.

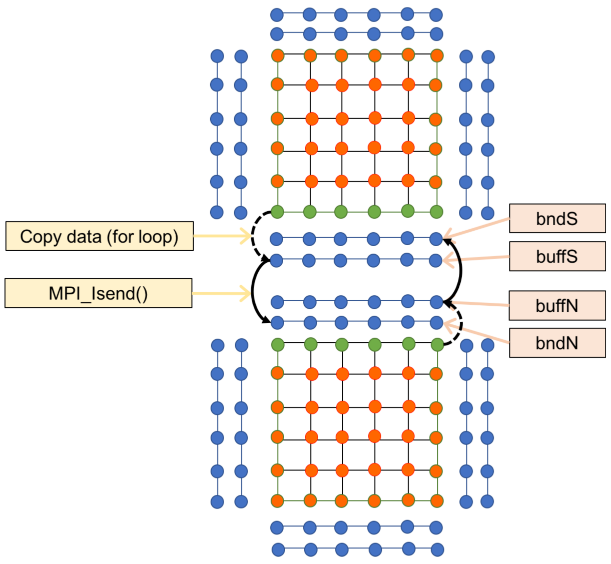

Figure 1: Ghost cell exchange, copy the North (South) row to buffN (buffS), send buffN (buffS) to the neighbor, receive to bndS (bndN)

5. Task 5 - Testing [20 Points]

The scalability of the application at different grid resolutions and parallel configurations is a crucial factor in evaluating the efficiency of the parallelization strategy. In this phase, the objective is to measure the time to solution for various grid sizes, ranging from 128×128 to 512×512, under different numbers of threads and MPI ranks. The performance data is gathered across a range of 1 to 10 threads and 1, 2, and 4 MPI ranks, providing insights into the application's behavior under scaling conditions.

To facilitate this testing, a Bash script is employed to automate the execution of the application across the specified configurations. The script iterates over predefined grid sizes and combinations of thread and rank counts, setting the number of OpenMP threads and MPI ranks accordingly. Each iteration invokes the application with the current parameters, captures the execution time, and records the results in a CSV file for subsequent analysis.

The collected data, structured in the CSV format with columns for grid size, MPI rank, thread count, and execution time, is ready for plotting.

5.1. Analysis of 128x128 Grid Size Benchmark

Figure 2 presents the execution times of our application with a grid size of 128×128, highlighting the impact of threading and MPI rank scaling on performance.

The results indicate a monotonic increase in execution time as the number of threads increases for each rank configuration. Notably, the application shows improved performance with a higher number of MPI ranks when executed without OpenMP parallelization. However, as both the ranks and the thread count increase, the data suggests a performance bottleneck. This is particularly evident with four MPI ranks, where execution time surges significantly as more threads are added.

The steepest decline in performance at the highest rank and thread combination implies a potential overhead, likely due to increased inter-process communication or resource contention.

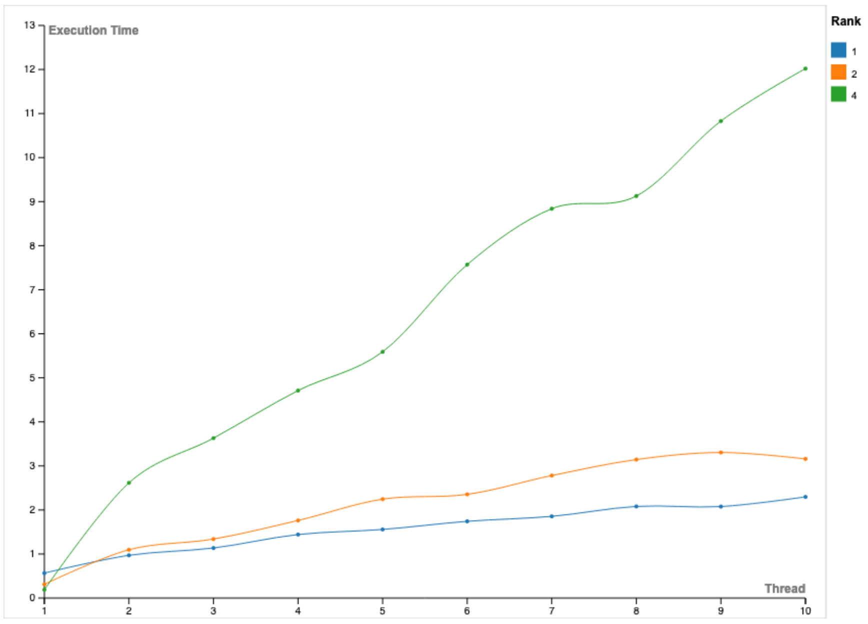

Figure 2: Execution time for a 128 × 128 grid size benchmark across different numbers of threads and MPI ranks.

5.2. Analysis of 256x256 Grid Size Benchmark

Figure 3 depicts the execution time for a 256 × 256 grid size under varying parallel configurations. This benchmark aims to shed light on the scalability and efficiency of the application as we increase the number of threads and MPI ranks.

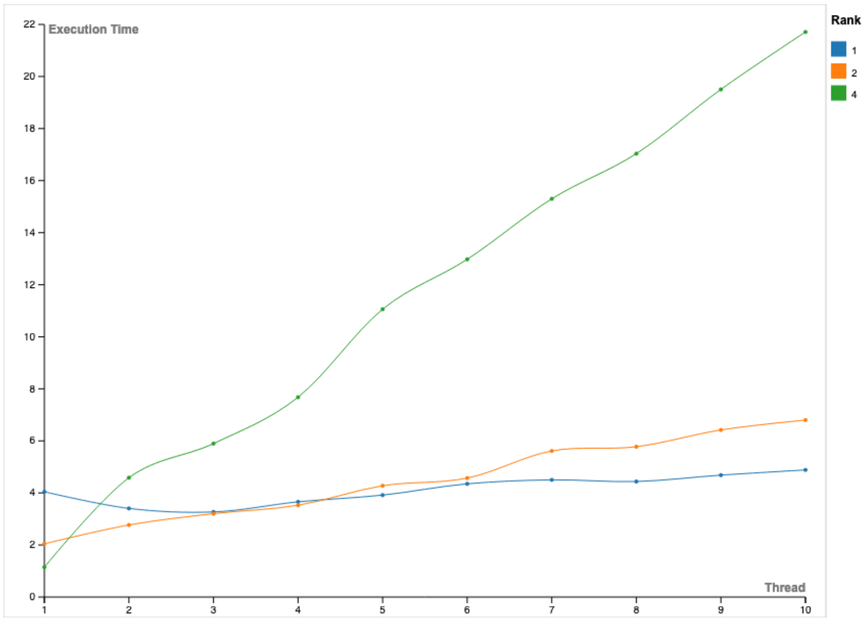

Figure 3: Execution time as a function of the number of OpenMP threads for different MPI ranks, assessed on a 256 × 256 grid.

In the analysis, we observe that with a single MPI rank, the application's execution time remains relatively stable with an increase in threads, up to a count of six. Beyond this point, there is no significant improvement, indicating that the application may not effectively utilize additional threads due to limitations such as memory bandwidth or computational overhead.

For two MPI ranks, the performance profile is similar to that of a single rank, with no substantial benefit observed from increased parallelism. This suggests that for this problem size, the overhead associated with inter-rank communication might counteract the performance gains from parallel processing.

The performance for four MPI ranks, however, diverges notably from the lower-rank cases. While the execution time for a single thread is optimal, it rapidly escalates as more threads are introduced. This trend highlights a critical performance bottleneck.

5.3. Analysis of 512x512 Grid Size Benchmark

Figure 4 presents the execution times for the application with a 512×512 grid, analyzed across a spectrum of threading models and MPI rank distributions.

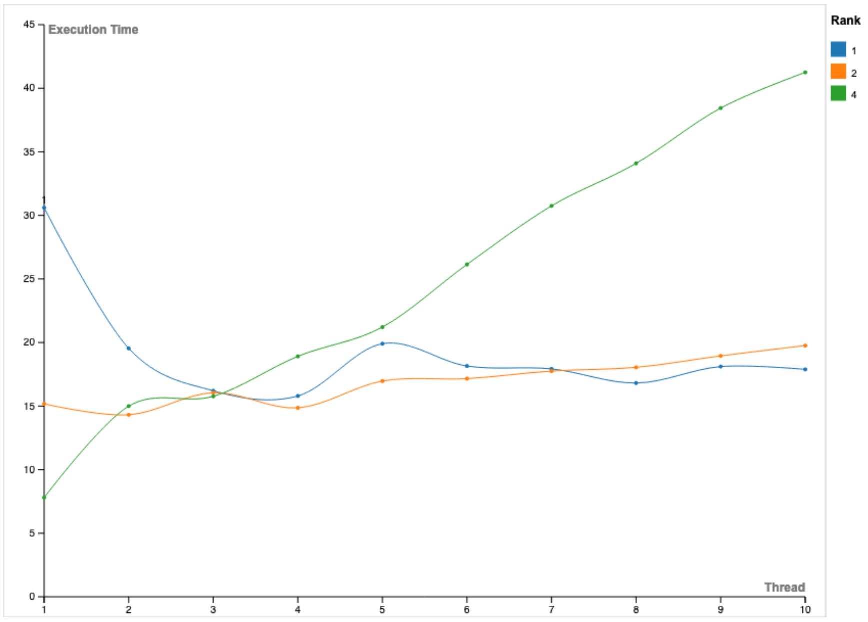

Figure 4: Execution time as a function of the number of OpenMP threads for different MPI ranks, tested on a 512×512 grid.

Consistent with the previous benchmarks, the configuration with four MPI ranks and a single thread yields the best performance. This outcome suggests that for this grid size, increasing the rank count while maintaining a lower thread count is beneficial up to a certain threshold.

A notable distinction in this benchmark is the convergence of execution times for a thread count ranging from two to five. This pattern indicates a plateau in performance gains when increasing threads within this range, irrespective of the rank count.

Similar to the earlier benchmarks, the results here also reflect the limitations and computational overhead encountered when running the application on the local machine. These constraints become increasingly apparent with larger grid sizes, which intensify the computational and memory demands.

5.4. Limitations in Benchmarking the 1024x1024 Grid Size

During the attempt to benchmark the application performance on a 1024×1024 grid, I encountered significant computational challenges. The local machine tasked with this benchmark recorded an average processing time of approximately 200 to 240 seconds per data point. Extrapolating from this figure, the total time estimated to acquire a complete set of data points for a comprehensive plot was around 90 minutes.

Regrettably, the benchmarking process was further complicated by recurrent system instabilities, which led to multiple system crashes. These technical interruptions rendered the completion of the full benchmarking suite unfeasible within the available time frame.