Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

High-Performance Computing Lab for CSE **2024**

Student: Danil Poluyanov Discussed with: Ielizaveta Polupanova

---

Solution for Project 4

Due date: Monday 29 April 2024, 23:59 (midnight).

HPC Lab for CSE 2024 — Submission Instructions
(Please, notice that following instructions are mandatory: submissions that don't comply with, won't be considered)

- Assignments must be submitted to Moodle (i.e. in electronic format).
- Provide both executable package and sources (e.g. C/C++ files, Matlab). If you are using libraries, please add them in the file. Sources must be organized in directories called:
  Project_number_lastname_firstname
  and the file must be called:
  project_number_lastname_firstname.zip
  project_number_lastname_firstname.pdf
- The TAs will grade your project by reviewing your project write-up, and looking at the implementation you attempted, and benchmarking your code's performance.
- You are allowed to discuss all questions with anyone you like; however: (i) your submission must list anyone you discussed problems with and (ii) you must write up your submission independently.

1. Ring sum using MPI [10 Points]

1.1. Implementation

The MPI ring sum algorithm was implemented to demonstrate basic MPI send/receive functionality within a ring topology. In this topology, each MPI process is connected in a circular chain, allowing each process to have two neighbors, with the first and last processes being neighbors as well.

The core of the implementation involves each process sending and receiving tokens in a ring, whereby each process initially sends its rank to the next process and receives a token from the previous process. This operation is repeated n times, where n is the total number of processes, ensuring each process accumulates the sum of all ranks.

The implementation avoids potential deadlock by using blocking send and receive calls, which are ordered to ensure that all sends are posted before waiting for the corresponding receives. This pattern ensures that the communication proceeds without any process waiting indefinitely for data that has not yet been sent.

Here is a snippet of the crucial part of the implementation:

```
for(int i = 0; i < size; i++){
    MPI_Send(&token, 1, MPI_INT, next, 0, MPI_COMM_WORLD);
    MPI_Recv(&token, 1, MPI_INT, prev, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    sum += token;
}
```

Each process updates its sum with the token received, which contains the rank of the preceding neighbor. This simple yet effective method ensures that the sum of all ranks is correctly calculated by each process without deadlocks.

1.2. Results

Upon execution, each process outputs its final accumulated sum, demonstrating the correct implementation of the ring sum algorithm. The output is verified against the theoretical sum of a series of ranks to ensure correctness. The algorithm's simplicity allows for easy verification and debugging.

The results indicate that the implementation successfully computes the ring sum without any synchronization issues or deadlocks across various tests with different numbers of processes.

1.3. Potential Improvements

While the current implementation uses blocking communications, which simplifies understanding and ensures correctness, it can be further optimized using non-blocking communications (`MPI_Isend` and `MPI_Irecv`). This change could potentially improve the performance by overlapping communication with computation, especially beneficial in larger ring sizes where the communication delay becomes significant.

1.4. Conclusion

The ring sum algorithm serves as an effective demonstration of MPI's capability to handle cyclic communication patterns robustly. The implementation showcases how to set up a basic MPI communication pattern, ensuring data integrity and deadlock avoidance in a distributed computing environment.

2. Cartesian domain decomposition and ghost cells exchange [20 Points]

2.1. Implementation

This task involved the implementation of a two-dimensional Cartesian domain decomposition and the exchange of ghost cells among neighboring MPI processes. The setup was configured for a grid of processes arranged in a 4x4 topology, with each process responsible for a sub-domain extended by ghost cells to facilitate data exchange with adjacent processes.

2.1.1. Cartesian Topology Setup

A Cartesian communicator was created with periodic boundaries to enable a wrapping connection among the edge processes. This was crucial for the cyclic nature of our data dependencies, especially since the problem statement defined the first and last ranks in each dimension as neighbors:

```
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, reorder, &comm_cart);
```

The dimensions were set to 4x4, and the periodicity was enabled for both dimensions to ensure seamless communication across the domain edges.

2.1.2. Neighbor Discovery

Using the MPI function `MPI_Cart_shift`, each process determined its four direct neighbors (top, bottom, left, and right), which facilitated targeted data sends and receives:

```
MPI_Cart_shift(comm_cart, 0, 1, &rank_bottom, &rank_top);
MPI_Cart_shift(comm_cart, 1, 1, &rank_left, &rank_right);
```

2.1.3. Data Type Definitions and Communication

Derived MPI data types were created for efficient communication of sub-array boundaries. `MPI_Type_vector` was initially used for column transfers, and `MPI_Type_contiguous` for row transfers:

```
MPI_Type_vector(SUBDOMAIN, 1, DOMAINSIZE, MPI_DOUBLE, &data_ghost);
MPI_Type_contiguous(SUBDOMAIN, MPI_DOUBLE, &data_rows);
```

Communication was structured to avoid deadlocks by first initiating all sends and then proceeding with the corresponding receives, using non-blocking calls (`MPI_Isend` and `MPI_Irecv`) followed by `MPI_Waitall` to ensure completion before progressing.

2.2. Challenges and Improvements

Implementing non-blocking communication was initially challenging due to my familiarity with blocking calls. However, by mirroring the placement of non-blocking sends and receives as I would with blocking functions, I was able to manage the data flow effectively without deadlocks. This experience highlighted the nuances of MPI communication patterns, especially in managing asynchronous operations.

2.2.1. Not Addressed: Bonus Task

The bonus task involving diagonal (ordinal) neighbor communication was not addressed in this implementation. For future work, extending the communication pattern to include diagonal neighbors could be achieved by adding additional `MPI_Cart_shift` calls to determine the northeast, southeast, southwest, and northwest neighbors, followed by similar send/receive logic.

2.3. Conclusion

The task effectively demonstrated the application of MPI for complex domain decomposition and inter-process communication within a Cartesian grid. The successful implementation underscored the capabilities of MPI in handling data dependencies across a distributed set of processes, paving the way for more sophisticated multi-dimensional parallel applications.

3. Parallelizing the Mandelbrot set using MPI [30 Points]

3.1. Implementation

The task involved implementing a parallel version of the Mandelbrot set computation using MPI, where the computation was divided among multiple MPI processes. Each process computed a specific portion of the Mandelbrot set, corresponding to its partition of the total domain.

3.1.1. Domain Partitioning

The computation domain was partitioned using a Cartesian topology defined in the `consts.h` file. The Cartesian communicator facilitated the identification of process coordinates and neighbors:

```
MPI_Cart_coords(comm, rank, 2, coords);
```

This setup ensured that each process was aware of its position within the process grid, thus enabling a tailored computation of the Mandelbrot set for its specific segment.

3.1.2. Local Domain Computation

Functions `createPartition` and `createDomain` were implemented to assign each process its segment of the domain:

```
typedef struct {
    long nx, ny, startx, starty, endx, endy;
} Domain;
```

The `createDomain` function calculated the boundaries of the local domain based on the process's coordinates within the grid, ensuring each process worked on a unique segment of the image.

3.1.3. Data Communication

Each non-root process sent its computed segment of the Mandelbrot set to the root process, which then assembled these segments into a final image. This was managed using `MPI_Isend` for non-blocking send operations and `MPI_Recv` at the root process:

```
if (rank == 0) {
    MPI_Recv(...);
} else {
    MPI_Isend(...);
}
```

3.2. Challenges

The primary challenge was ensuring the correct calculation of the local domain boundaries, particularly with potential size mismatches and orientation issues regarding the x and y axes. Debugging these issues required careful verification of the indices used in the `createDomain` function.

3.3. Performance Analysis

The performance on the Euler cluster was impressive, showing significant speedup with increasing numbers of processes. This scaling behavior underscored the benefits of parallel computation, particularly for computationally intensive tasks like the Mandelbrot set.

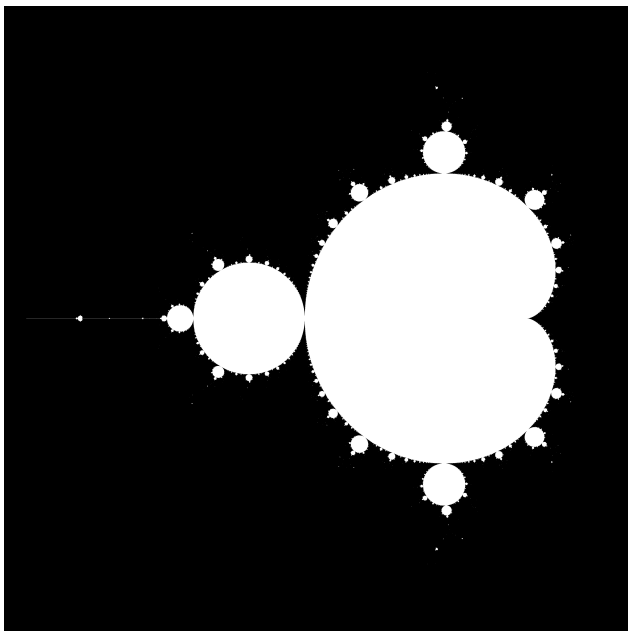

Figure 1: Mandelbrotset

3.4. Conclusion

The implementation demonstrated effective parallelization of the Mandelbrot set computation. The use of MPI to distribute the workload across multiple processors resulted in a substantial

reduction in computation time, confirming the potential of MPI for parallel processing tasks in high-performance computing environments.

4. Parallel matrix-vector multiplication and the power method [40 Points]

4.1. Introduction

This report describes the parallel implementation of the power method to compute the largest absolute eigenvalue of a matrix using MPI. The implementation involved partitioning the matrix row-wise among MPI processes and using collective communication operations to synchronize vector updates.

4.2. Implementation Details

4.2.1. Matrix Partitioning

The matrix was partitioned evenly among the available MPI processes to ensure load balancing. Each process handled a subset of rows, thus minimizing the communication overhead during the vector-matrix multiplication phase.

4.2.2. Vector Initialization and Distribution

A random initial guess vector was generated by the root process and broadcast to all processes using `MPI_Bcast`. This ensured that all processes started the computation with the same initial vector.

4.2.3. Local Computation

Each process computed the product of its matrix segment with the vector locally. This local result was then combined with the results of other processes through the use of `MPI_Allgatherv`, ensuring that all processes had the updated vector for the next iteration.

4.3. Convergence Checking

After each iteration, the root process checked for convergence by evaluating the norm of the difference between successive iterations. If the change was below a specified tolerance, the computation was terminated.

4.4. Challenges

The main challenge was ensuring that the data distributed across processes was managed correctly to prevent any data inconsistency, especially during the synchronization steps. Debugging issues related to MPI communication patterns required careful examination of the buffer sizes and offsets used in `MPI_Allgatherv`.

4.5. Performance Analysis

4.5.1. Scaling Studies

Two scaling studies were performed:

- **Strong Scaling:** The computation was run with increasing numbers of processes while keeping the matrix size constant. This tested the efficiency of the parallel implementation as the workload per process decreased.

- **Weak Scaling:** The matrix size was increased proportionally with the number of processes to keep the workload per process constant. This tested the scalability without changing the computational load per process.

4.5.2. Results

The scaling didn't work for some reason.