Università
della
Svizzera
italiana

Institute of Computing CI

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

Student: Varese Lorenzo Discussed with: Kaan Sen

Solution for Project 4

1. Ring maximum using MPI [10 Points]

The purpose of this section is to describe the implementation of a parallel algorithm that determines the maximum value in a distributed system using MPI (Message Passing Interface). The algorithm utilizes a ring topology to pass messages between processes to achieve this goal.

The `custom_ring_algorithm` implements the functionality to compute the global maximum as per the provided function $(3 \times \text{rank} \mod (2 \times \text{size}))$. This is achieved by performing a series of send and receive operations with a ring strategy, updating the maximum value as the messages are passed around the ring.

The main steps in the `custom_ring_algorithm` function are as follows:

1. Each process calculates its initial value based on the rank and the provided function.
2. The process enters a loop of size equal to the number of processes, in which it sends and receives values to and from its right and left neighbors, respectively.
3. After each receive, the process updates the maximum value if the received value is greater.
4. This updated maximum value is then sent in the next iteration.

```c
int custom_ring_algorithm(int rank, int size, int left, int right, void *status)
{
    int snd_buf = (3 * rank) % (2 * size);
    int max = snd_buf;
    int rcv_buf = 0;

    for (int i = 0; i < size; i++)
    {
        MPI_Sendrecv(&snd_buf, 1, MPI_INT, right, 0,
                     &rcv_buf, 1, MPI_INT, left, 0,
                     MPI_COMM_WORLD, status);

        if (rcv_buf > max)
            max = rcv_buf;

        snd_buf = max;
    }

    return max;
}
```

The ring_sum_algorithm provided in the code is an additional implementation, this function demonstrates a basic ring summation algorithm, which could be used as a reference or a starting point for the custom implementation. The implemented program contains both the required custom_ring_algorithm for the global maximum computation and the ring_sum_algorithm for illustrative purposes.

The described MPI-based parallel algorithm successfully calculates the global maximum in a ring-structured network of processes, showing the utility of MPI in performing distributed computations.

2. Ghost cells exchange between neighboring processes [15 Points]

The goal of this exercise is to facilitate the exchange of ghost cells in a distributed memory system using MPI. This is crucial in parallel programs where computation on domain boundaries requires information from adjacent processes. By using a 2D Cartesian communicator with periodic boundaries, we can ensure that each process communicates effectively with its neighbors, even on the edges of the process grid.

2.1. Setting Up a 2D Cartesian Communicator with Periodic Boundaries

To model our domain, we use a 4x4 grid where each process is responsible for a 6x6 subdomain. The periodic boundary conditions are established using MPI's Cartesian communicator, which connects the grid's edges together. Here's a snippet of code creating the communicator:

```
int dims[2] = {4, 4}, periods[2] = {1, 1}, reorder = 0;
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, reorder, &comm_cart);
```

This communicator facilitates the identification of neighboring ranks, allowing for an efficient exchange of boundary data.

2.2. Defining a Derived Data Type for Column Border Communication

MPI's abstraction of data types allows us to define a columnar data structure that represents the ghost cells. We leverage MPI's datatype constructors to create a derived type that facilitates the vertical exchange of ghost cells. The following code defines such a datatype:

```
MPI_Type_vector(SUBDOMAIN, 1, DOMAINSIZE, MPI_DOUBLE, &ghost_col);
MPI_Type_commit(&ghost_col);
```

2.3. Conducting Ghost Cell Exchange and Verification

The exchange of ghost cells is a critical step in ensuring data consistency across the subdomains handled by each process. This is achieved using non-blocking receives and synchronous sends, which are orchestrated to minimize wait times and prevent deadlocks. The exchanges are conducted in each cardinal direction—top, bottom, left, and right—thereby updating the ghost cells with data from neighboring processes.

For example, the communication with the top neighbor is handled as follows:

```
MPI_Irecv(/* top ghost cells */, 1, ghost_row, rank_bottom, TO_TOP, comm_cart, &
    request);
MPI_Send(/* bottom data cells */, 1, ghost_row, rank_top, TO_TOP, comm_cart);
MPI_Wait(&request, &status);
```

The non-blocking receive function, MPI_Irecv, is posted first to listen for incoming data, followed by a synchronous send function, MPI_Send, to send the appropriate subdomain data. Once the communication is initiated, MPI_Wait is called to ensure that the non-blocking operation completes before proceeding further.

After the communication phase, it is essential to verify that the correct values have been placed in the ghost cells. This is done by inspecting the array representing our domain after the exchange has taken place. On a system with fewer cores than the required number of processes, such as a local machine with 8 cores, MPI allows for over-subscription to emulate a larger parallel environment. The following shell command demonstrates running the program with over-subscription to utilize 16 processes:

```
> mpirun -np 16 --oversubscribe ./ghost
```

The output printed by the rank 9 process illustrates the updated ghost cells, confirming that the boundaries have been successfully exchanged with its neighbors:

```
data of rank 9 after communication
9.0 5.0 5.0 5.0 5.0 5.0 5.0 9.0
8.0 9.0 9.0 9.0 9.0 9.0 9.0 10.0
...
9.0 13.0 13.0 13.0 13.0 13.0 13.0 9.0
```

This output reflects the correct data exchange across the ghost cells, validating the integrity of our communication scheme in a simulated distributed memory system. The ability to oversubscribe processes on a limited-core system is beneficial for development and testing purposes, allowing us to simulate and test parallel environments on a smaller scale.

3. Parallelizing the Mandelbrot set using MPI [20 Points]

The Mandelbrot set's computation is a classic example of a task that can benefit from parallel processing. By using MPI, we can distribute the task across multiple processors to reduce computation time significantly.

3.1. Image Partitioning with Partition Creation and Update Functions

The parallel computation of the Mandelbrot set requires dividing the image into partitions that can be processed independently. The `Partition` structure is central to this, as it defines the layout of the processor grid and the position of each process within it.

```
#define IMAGE_WIDTH   4096
#define IMAGE_HEIGHT  4096

typedef struct {
    int y;
    int x;
    int nx;
    int ny;
    MPI_Comm comm;
} Partition;
```

The `createPartition` function initializes this structure, creating a Cartesian communicator and determining the grid dimensions and the coordinates of each process. The `updatePartition` function is used to update this structure with the coordinates of a specific MPI process.

```
Partition createPartition(int mpi_rank, int mpi_size);
Partition updatePartition(Partition p_old, int mpi_rank);
```

3.2. Calculating Local Domain Dimensions from Partitions

Each process computes a portion of the Mandelbrot set defined by its local domain. The `Domain` structure holds the size of the local domain and its global indices.

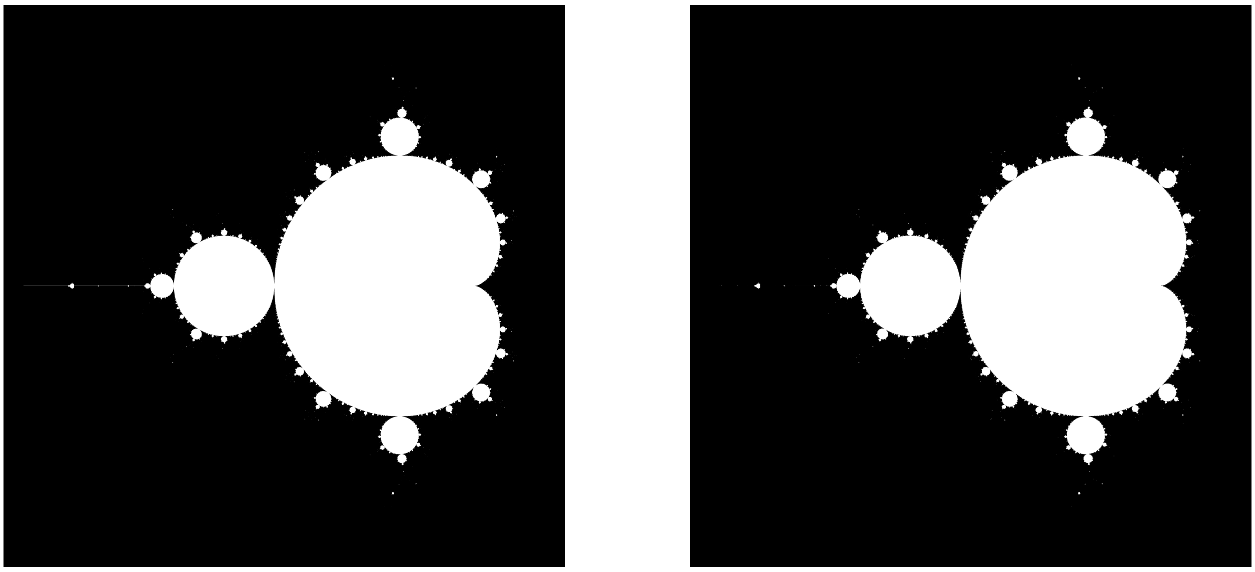
```
typedef struct {
    long nx, ny;
    long startx, starty;
    long endx, endy;
} Domain;
```

The `createDomain` function uses the `Partition` structure to calculate the size and boundaries of the local domain that each process will compute:

```
Domain createDomain(Partition p);
```

3.3. Communicating Local Domains Between Worker and Master Processes

The parallelized approach requires worker processes (with `mpi_rank` > 0) to send their computed local domains to the master process (`mpi_rank` $== 0$). The master process receives these domains and reconstructs the entire Mandelbrot set. The integrity of the parallel computation is verified by comparing the output with that obtained from a sequential program. Figure 1 shows the outputs of the parallelized and sequential programs side by side, illustrating their similarity.



(a) Parallelized program output

(b) Sequential program output

Figure 1: Comparison of the Mandelbrot set computed using parallelized (a) and sequential (b) approaches. Despite minor variations due to the nature of parallel computation and data assembly, the figures exhibit a high degree of similarity, validating the parallel algorithm's correctness.

3.4. Performance Analysis and MPI Parallelization Benefits

The efficiency of parallel computation in generating the Mandelbrot set is evaluated by varying the number of processors. Initial observations from the benchmark results, as seen in Figure 2, indicate a trend of decreasing computation time with an increasing number of processors, highlighting the potential for performance gains through MPI parallelization.

However, there are notable discrepancies in the scaling efficiency, particularly as the number of processes increases beyond the physical core count of the local machine, which is capped at eight cores. These discrepancies can be attributed to several factors:

- **Computational Load Imbalance:** The Mandelbrot set contains regions that converge rapidly (black areas) and others that require a higher number of iterations to determine divergence (white areas). Since processes are allocated equal-sized partitions of the set without considering the complexity of the region, some processes are burdened with more computationally intensive tasks than others. This results in an uneven distribution of computational load and increased wait times for synchronization points.
- **Physical Core Limitations:** When the number of processes exceeds the number of physical cores, the operating system must time-slice the available cores among the processes, leading to context-switching overhead. Furthermore, any additional processes are subjected to virtual core allocation, which does not provide the same performance benefits as physical cores.
- **Communication Overhead:** As the number of processes increases, so does the communication overhead for passing messages between processes, especially when assembling the final image from the local domains computed by individual processes.

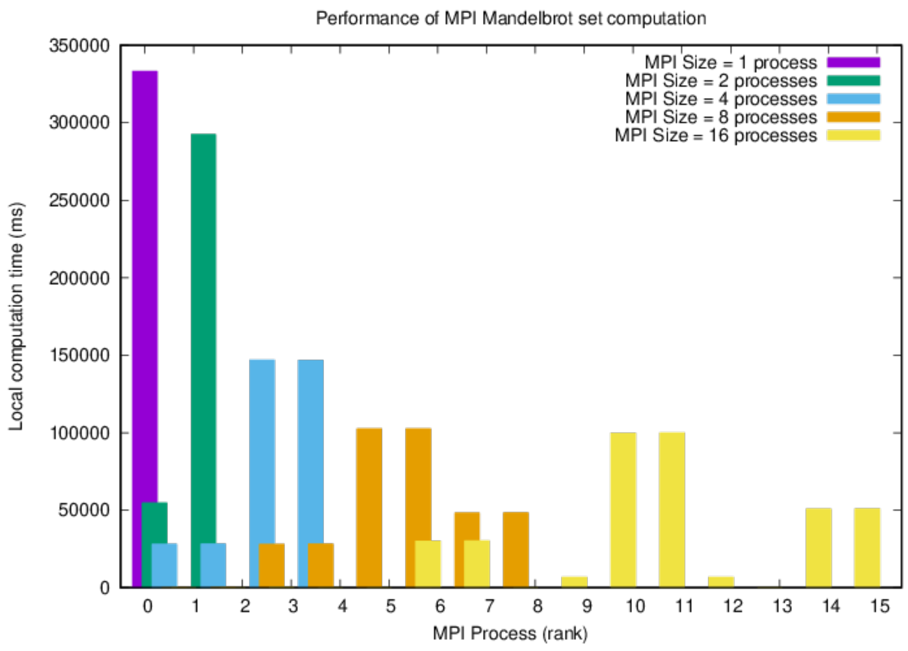


Figure 2: Benchmark results showing the computation time for varying numbers of processes.

These factors collectively contribute to the diminishing returns observed in the benchmark as the number of utilized processes exceeds the number of physical cores on the local machine. Consequently, while MPI parallelization is beneficial, it is crucial to consider load balancing and hardware constraints to optimize performance.

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

The power method, an iterative algorithm, is employed to approximate the largest eigenvalue of a given square matrix. This section outlines the development and implementation of the necessary MPI routines to perform matrix-vector multiplication and the subsequent application of these routines in the power method. The implementation aims at demonstrating the efficacy of parallel computation in reducing execution time while maintaining the accuracy of the results.

4.1. `hpc_generateMatrix` Implementation

The `hpc_generateMatrix` function is pivotal to our parallel power method implementation. It is responsible for generating a slice of the matrix A that will be used by each MPI process. This distributed approach is fundamental for parallel computations, as it allows us to handle matrices larger than what a single process could store in memory.

Each process calls this function to generate only the rows of the matrix it will operate on, defined by the parameters *startrow* and *numrows*. The function allocates memory for the required slice and sets the diagonal elements to the value of n, the size of the matrix, ensuring that the matrix is diagonally dominant.

Here is the code for the `hpc_generateMatrix` function:

```
double* hpc_generateMatrix(int n, int startrow, int numrows) {
    double* A = (double*)calloc(n * numrows, sizeof(double));
    for (int i = 0; i < numrows; i++) {
        int diag = startrow + i;
        A[i * n + diag] = n;
    }
    return A;
}
```

The function begins by allocating a zero-initialized block of memory large enough to hold the matrix slice for the local process. It then iterates over each row that the process is responsible for and sets the diagonal element. The choice of the calloc function for memory allocation ensures that all non-diagonal elements are initialized to zero by default, which simplifies the process of creating the matrix.

4.2. `power_method` Implementation

The `power_method` function is related to the eigenvalue computation in parallel computing using the MPI. It is designed to converge to the largest eigenvalue of a matrix through an iterative process that utilizes matrix-vector multiplication and vector normalization.

The signature of the function is as follows:

```
double power_method(double *partial_rows, double *vector,
                    int matrix_size, int rows_per_process,
                    int rank, MPI_Comm comm);
```

The algorithm operates as follows: starting with a random vector, it normalizes this vector, then multiplies it by the matrix, and repeats these steps for a fixed number of iterations (1000 in our implementation).

During each iteration of the power method, the `matrix_vector_multiply` function plays a critical role:

```
void matrix_vector_multiply(double *partial_rows, double *vector, int matrix_size
    , int rows_per_process, int rank, MPI_Comm comm);
```

This function begins by broadcasting the current vector to all processes using `MPI_Bcast`, ensuring every process works with the same vector data. It then computes a segment of the resulting vector by performing a dot product of the matrix's partial row slice and the vector. Each process allocates memory for its segment of the result, which is then gathered together using `MPI_Gather`. The root process (rank 0) collects the partial results to assemble the complete resultant vector.

It is important to note that the function uses dynamic memory allocation for the temporary result vector, which is then freed at the end of the function to prevent memory leaks.

The power method relies on the norm of the vector for both the normalization step and as the final result to approximate the eigenvalue after convergence. This norm is computed by the `vector_norm` function, which is called at the end of the power method iterations.

4.3. `main` Implementation

The `main` function acts as the orchestrator for the parallel computation of the power method. It initializes the MPI environment, distributes the work among processes, and manages the flow of the program from initialization to the output of results.

The function is structured as follows:

```
int main(int argc, char *argv[]) {
    ...
    MPI_Init(&argc, &argv);
    ...
    int matrix_size = strtol(argv[1], &endptr, 10);
    ...
    double *vector = create_random_vector(matrix_size);
    double *matrix_slice = hpc_generateMatrix(matrix_size, start_row,
                                               rows_per_process);
    ...
    double largest_eigenvalue = power_method(matrix_slice, vector, matrix_size,
                                  rows_per_process, my_rank, MPI_COMM_WORLD);
    ...
    MPI_Finalize();
    return 0;
}
```

Upon execution, each process determines its rank and the total number of processes involved. The matrix size is derived from the command line argument, and based on this size, each process generates its portion of the matrix and a random vector. The function `hpc_generateMatrix` provides each process with the appropriate slice of the matrix to work on.

The `main` function also handles the timing of the eigenvalue computation, starting the timer before and stopping it after the `power_method` function call. The largest eigenvalue of the matrix is then computed in parallel by calling `power_method`.

After the computation, process 0 prints the total time taken, the vector resulting from the power method, the verification status obtained by calling `hpc_verify`, and the largest eigenvalue computed. This process demonstrates a simple yet effective parallelization strategy, distributing the matrix-vector multiplication work and then aggregating the results to find the dominant eigenvalue.

The function concludes by finalizing the MPI environment, signifying the end of the parallel computation.

4.4. Strong Scaling Analysis

The strong scaling analysis evaluates the performance improvement of the parallelized power method as the number of processes increases while keeping the problem size constant. Efficiency in strong scaling is computed using the formula:

$$\text{Efficiency} = \frac{\text{Serial Execution Time}}{\text{Parallel Execution Time} \times \text{Number of Processes}} \tag{1}$$

The analysis, as depicted in the bar chart (Figure 3), utilized a fixed matrix size of $10,000 \times 10,000$ and $1,000$ iterations of the main loop. On a local machine equipped with 8 physical cores, the efficiency displays a downward trend as the number of processes exceeds the number of physical cores. This is indicated by the near-optimal efficiency when the number of processes is equal to or less than 8. As the number of processes increases to 16, 32 and 64, a notable drop in efficiency is observed, suggesting that the overhead from communication and potential CPU contention is impacting performance. This pattern underscores the diminishing returns of adding more processes than the available physical cores.

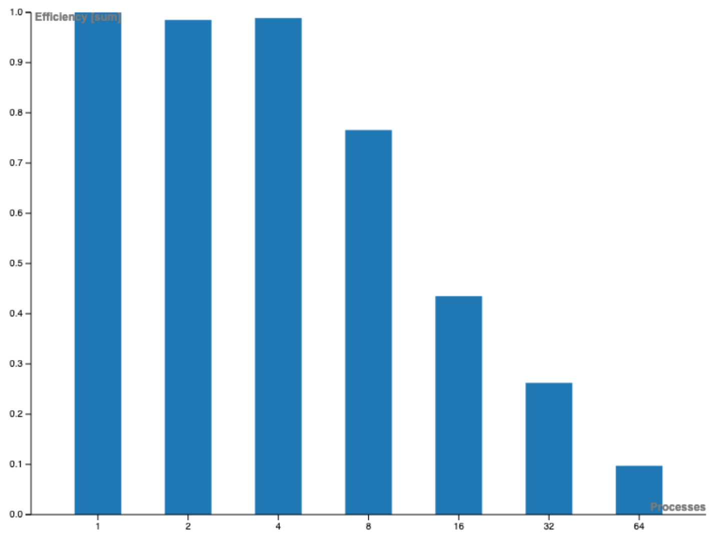


Figure 3: Strong scaling efficiency with a varying number of processes.

4.5. Weak Scaling Analysis

Weak scaling analysis aims to evaluate the efficiency of a parallel system when the workload per processor is kept constant while the number of processors is increased. In an ideal weak scaling scenario, as the problem size grows proportionally with the number of processors, the execution time should remain constant, reflecting perfect scalability. The efficiency in weak scaling is calculated with the formula:

$$\text{Efficiency} = \frac{\text{Base Execution Time}}{\text{Parallel Execution Time}} \tag{2}$$

In the conducted weak scaling experiment, the base matrix size was set to $4,096 \times 4,096$ elements, with a workload of $4,096 \times 4,096$ elements per process. The number of iterations was fixed at 100 for all runs. As depicted in Figure 4, the efficiency significantly decreases as the number of processes increases. Specifically, the plot shows an efficiency of 1 for a single process, as expected, since there is no parallel overhead. However, when the number of processes is equal to 16, the efficiency drops to just above 0.4. This trend continues, with efficiency decreasing further for 32 and 64 processes.

This observed degradation in efficiency could be attributed to several factors, including the communication overhead among an increasing number of processes and the limited memory bandwidth that becomes a bottleneck as more processes attempt to access and operate on larger matrices simultaneously. It's also noteworthy that the local machine has only 8 physical cores, and thus, going beyond this number with our parallel processes could lead to suboptimal CPU utilization due to context switching and other system-related overheads.

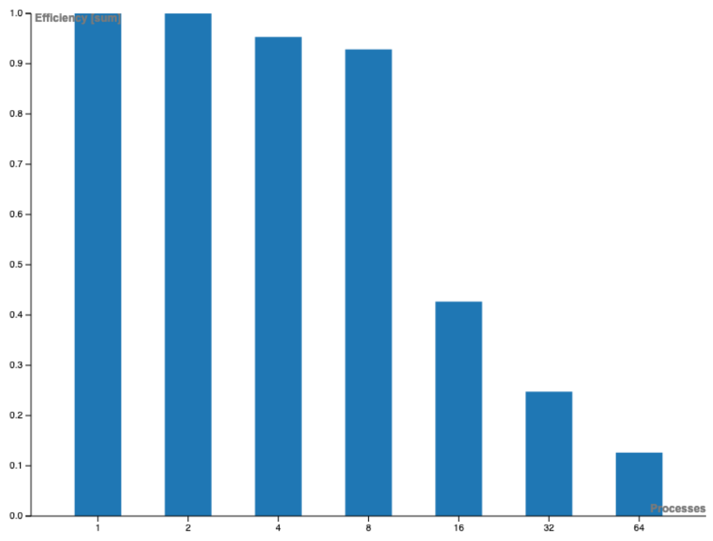


Figure 4: Weak scaling efficiency with a varying number of processes.

The results indicate that while weak scaling preserves the amount of work each processor performs, the increasing demand for resources and synchronization can lead to reduced efficiency.