Università della Svizzera italiana

Institute of Computing CI

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

Student: Zitian Wang Discussed with: None

Solution for Project 5

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

- Assignments must be submitted to iCorsi (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.

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

1. Task 1 - Initialize and finalize MPI

```
[wangzi@icsnode33 mini_app]$ mpirun ./main 128 100 0.005
============================================================
                        Welcome to mini-stencil!
version   :: C++ MPI
processes :: 4
mesh      :: 128 * 128 dx = 0.00787402
time      :: 100 time steps from 0 .. 0.005
iteration :: CG 300, Newton 50, tolerance 1e-06
============================================================
------------------------------------------------------------
1513 conjugate gradient iterations, at rate of 6599.61 iters/
   second
300 newton iterations
------------------------------------------------------------
Goodbye!
```

Listing 1: Execution of mini-stencil Simulation

Limited condition `rank == 0` to make sure only process 0 report in terminal.

2. Task 2 - Create a Cartesian topology

```
    // TODO: determine the number of sub-domains in the x and y dimensions
    //       using MPI_Dims_create
    int dims[2] = {1, 1};
    MPI_Dims_create(mpi_size, 2, dims);
    ndomy = dims[0];
    ndomx = dims[1];

    // TODO: create a 2D non-periodic Cartesian topology using
       MPI_Cart_create
    int periods[2] = {0, 0};
    MPI_Comm comm_cart;
    MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 0, &comm_cart);

    // TODO: retrieve coordinates of the rank in the topology using
    // MPI_Cart_coords
    int coords[2] = {0, 0};
    MPI_Cart_coords(comm_cart, mpi_rank, 2, coords);
    domy = coords[0] + 1;
    domx = coords[1] + 1;

    // TODO: set neighbours for all directions using MPI_Cart_shift
    MPI_Cart_shift(comm_cart, 0, 1, &neighbour_south, &neighbour_north);
    MPI_Cart_shift(comm_cart, 1, 1, &neighbour_west, &neighbour_east);
```

Listing 2: Setting Up a 2D Cartesian Topology with MPI

I used a **2D Cartesian domain decomposition** to divide the grid across MPI processes. `MPI_Dims_create` determined a balanced decomposition, and `MPI_Cart_create` established a non-periodic Cartesian topology for identifying neighboring subdomains.

Each process handles a specific subdomain, ensuring consistent workload distribution and efficient computation.

2.1. Method selected reason

1. **Load Balancing**: Dividing the domain into equal-sized subdomains ensures each process has similar work, preventing load imbalance.
2. **Communication Efficiency**: The 2D Cartesian topology allows efficient communication with only immediate neighbors, minimizing overhead.
3. **Scalability**: This strategy scales well with increasing grid size or number of processes, maintaining balance and efficiency.

2.2. Implications on Performance

1. **Load Balancing**: `MPI_Dims_create` ensures even domain division, minimizing idle time and maximizing resource use.
2. **Communication Overhead**: Communication is limited to neighboring processes, reducing data transfer and overall cost.
3. **Computational Efficiency**: Each process independently handles a subdomain, allowing effective parallelization and overlapping of communication and computation.

3. Task 3 - Change linear algebra functions

```
double hpc_dot(Field const& x, Field const& y) {
    double local_result = 0;
    int N = y.length();

    // Compute local result
    for (int i = 0; i < N; i++) {
        local_result += x[i] * y[i];
    }

    double global_result = 0;
    // Perform an MPI_Allreduce to sum up results across all processes
    MPI_Allreduce(&local_result, &global_result, 1, MPI_DOUBLE,
        MPI_SUM, MPI_COMM_WORLD);

    return global_result;
}
```

Listing 3: Implementation of `hpc_dot` for Dot Product with MPI

```
double hpc_norm2(Field const& x) {
    double local_result = 0;
    int N = x.length();

    // Compute local result
    for (int i = 0; i < N; i++) {
        local_result += x[i] * x[i];
    }

    double global_result = 0;
    // Perform an MPI_Allreduce to sum up the squared values across
        all processes
    MPI_Allreduce(&local_result, &global_result, 1, MPI_DOUBLE,
        MPI_SUM, MPI_COMM_WORLD);

    // Return the square root of the global result
    return sqrt(global_result);
}
```

Listing 4: Implementation of `hpc_norm2` for 2-Norm Calculation with MPI

```
void hpc_cg(Field& deltay, Field const& y_old, Field const& y_new,
            Field const& f, const int maxiters, const double tol,
            bool& success) {
    // This is the dimension of the linear system to solve
    int nx = data::domain.nx;
    int ny = data::domain.ny;

    if (!cg_initialized) cg_init(nx, ny);

    // Epsilon value for matrix-vector approximation
    double eps     = 1.e-4;
    double eps_inv = 1. / eps;

    // Initialize to zero
    hpc_fill(fv, 0.0);

    // Jacobian times deltay matrix-vector multiplication
        approximation
```

```
18          // J*deltay = 1/epsilon * ( f(y_new + epsilon*deltay) - f(y_new) )
19
20          // v = y_new + epsilon*deltay
21          hpc_lcomb(v, 1.0, y_new, eps, deltay);
22
23          // fv = f(v)
24          diffusion(y_old, v, fv);
25
26          // r = b - A*x = f(y_new) - J*deltay
27          // where A*x = (f(v) - f) / eps
28          hpc_add_scaled_diff(r, f, -eps_inv, fv, f);
29
30          // p = r
31          hpc_copy(p, r);
32
33          // r_old_inner = <r, r>
34          double r_old_inner = hpc_dot(r, r), r_new_inner = r_old_inner;
35          MPI_Allreduce(MPI_IN_PLACE, &r_old_inner, 1, MPI_DOUBLE, MPI_SUM,
                MPI_COMM_WORLD);
36
37          // Check for convergence
38          success = false;
39          if (sqrt(r_old_inner) < tol) {
40              success = true;
41              return;
42          }
43
44          int iter;
45          for (iter = 0; iter < maxiters; iter++) {
46              // Ap = A*p
47              hpc_lcomb(v, 1.0, y_new, eps, p);
48              diffusion(y_old, v, fv);
49              hpc_scaled_diff(Ap, eps_inv, fv, f);
50
51              // alpha = r_old_inner / p' * Ap
52              double pAp = hpc_dot(p, Ap);
53              MPI_Allreduce(MPI_IN_PLACE, &pAp, 1, MPI_DOUBLE, MPI_SUM,
                    MPI_COMM_WORLD);
54              double alpha = r_old_inner / pAp;
55
56              // deltay += alpha * p
57              hpc_axpy(deltay, alpha, p);
58
59              // r -= alpha * Ap
60              hpc_axpy(r, -alpha, Ap);
61
62              // Find new norm
63              r_new_inner = hpc_dot(r, r);
64              MPI_Allreduce(MPI_IN_PLACE, &r_new_inner, 1, MPI_DOUBLE,
                    MPI_SUM, MPI_COMM_WORLD);
65
66              // Test for convergence
67              if (sqrt(r_new_inner) < tol) {
68                  success = true;
69                  break;
70              }
71
72              // p = r + (r_new_inner / r_old_inner) * p
73              double beta = r_new_inner / r_old_inner;
74              hpc_lcomb(p, 1.0, r, beta, p);
```

```
75
76              r_old_inner = r_new_inner;
77          }
78          stats::iters_cg += iter + 1;
79
80          if (!success) std::cerr << "ERROR: CG failed to converge" << std::
                endl;
81      }
```

Listing 5: Implementation of `hpc_cg` for Conjugate Gradient Method

3.1. Functions Modified and Not Modified

I modified several functions to enable parallel computation. The **hpc_dot** function, responsible for inner product calculation, was updated to compute the local inner product within each process and then use `MPI_Allreduce` to aggregate results globally. Similarly, the **hpc_norm2** function for 2-norm calculation was modified to calculate the local sum of squares, aggregate using `MPI_Allreduce`, and compute the square root of the global result. The **hpc_cg** function is a Conjugate Gradient solver for solving large, sparse, symmetric positive-definite systems of linear equations from PDE discretization.For Jacobian matrix-vector multiplication, I implemented halo cell exchanges using non-blocking communication (`MPI_Isend` and `MPI_Irecv`) to ensure efficient handling of subdomain boundaries. Additionally, I updated the convergence check to use `MPI_Allreduce` for global residual norm computation.

I did not modify the vector operations (**hpc_axpy**, **hpc_add_scaled_diff**, **hpc_scaled_diff**, **hpc_scale**, **hpc_lcomb** and **hpc_copy**). These functions only involve local calculations within each process and do not require cross-process communication, so the original implementation was sufficient.

3.2. Modification Method and Analysis

For the inner product calculation (**hpc_dot**), I ensured that each process computes the inner product of its local vector segment and then aggregates the results using `MPI_Allreduce` to obtain the global value. This approach efficiently parallelizes the computation while ensuring consistency.

For the 2-norm calculation (**hpc_norm2**), I applied a similar strategy. Each process calculates the sum of squares of its local vector elements, and `MPI_Allreduce` aggregates these sums. The square root of the global result is then computed to finalize the norm calculation.

For the **hpc_cg**, replaced standard dot product with an MPI-based version using `MPI_Allreduce` for global aggregation. Added halo cell exchanges using non-blocking communication (`MPI_Isend` and `MPI_Irecv`) to handle subdomain boundaries efficiently. Convergence Check (hpc_norm2), Used `MPI_Allreduce` to compute the global residual norm after each iteration.

The modifications were designed to balance load and minimize communication overhead. Each process performs its share of the computation, ensuring that the workload is evenly distributed. Communication overhead is introduced by `MPI_Allreduce` and halo cell exchanges, but these operations are essential for maintaining accuracy and consistency in parallel computations. Using non-blocking communication for boundary exchanges further optimizes performance by overlapping communication with computation.

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

4.1. Ghost cell exchange function

```
1       MPI_Request requests[8];
2       int req_count = 0;
3
4       // Send to and receive from the northern neighbor
```

```
    MPI_Isend(&s_new(0, jend-1), nx, MPI_DOUBLE, domain.neighbour_north,
        0, MPI_COMM_WORLD, &requests[req_count++]);
    MPI_Irecv(&buffN[0], nx, MPI_DOUBLE, domain.neighbour_north, 1,
        MPI_COMM_WORLD, &requests[req_count++]);

    // Send to and receive from the southern neighbor
    MPI_Isend(&s_new(0, 0), nx, MPI_DOUBLE, domain.neighbour_south, 2,
        MPI_COMM_WORLD, &requests[req_count++]);
    MPI_Irecv(&buffS[0], nx, MPI_DOUBLE, domain.neighbour_south, 3,
        MPI_COMM_WORLD, &requests[req_count++]);

    // Send to and receive from the eastern neighbor
    MPI_Isend(&s_new(iend-1, 0), ny, MPI_DOUBLE, domain.neighbour_east, 4,
         MPI_COMM_WORLD, &requests[req_count++]);
    MPI_Irecv(&buffE[0], ny, MPI_DOUBLE, domain.neighbour_east, 5,
        MPI_COMM_WORLD, &requests[req_count++]);

    // Send to and receive from the western neighbor
    MPI_Isend(&s_new(0, 0), ny, MPI_DOUBLE, domain.neighbour_west, 6,
        MPI_COMM_WORLD, &requests[req_count++]);
    MPI_Irecv(&buffW[0], ny, MPI_DOUBLE, domain.neighbour_west, 7,
        MPI_COMM_WORLD, &requests[req_count++]);

    // Wait for all communication requests to complete
    MPI_Waitall(req_count, requests, MPI_STATUSES_IGNORE);
```

Listing 6: Non-Blocking Communication for Neighbor Exchange

To exchange ghost cells between neighboring processes, I used `MPI_Isend` and `MPI_Irecv`, which allow overlapping communication with computation, improving efficiency. Each subdomain sends its boundary values to neighboring subdomains and receives boundary values from them. Communication is required for the north, south, east, and west boundaries.

For each direction:

- Initiate an `MPI_Isend` to send the boundary values.
- Initiate an `MPI_Irecv` to receive the ghost cell values.

After initiating all communications, I used `MPI_Waitall` to ensure all communication had completed before proceeding with further calculations.

4.2. Implement parallel I/O

```
    [wangzi@icsnode30 mpi_io]$ mpirun -np 4 ./mpi_io 100 100
    [Process  3 ( 1,  1)] global dims: 100 x 100; local dims: 50 x
       50 (50:99 50:99);
    [Process  1 ( 0,  1)] global dims: 100 x 100; local dims: 50 x
       50 (0:49, 50:99);
    [Process  0 ( 0,  0)] global dims: 100 x 100; local dims: 50 x
       50 (0:49, 0:49);
    [Process  2 ( 1,  0)] global dims: 100 x 100; local dims: 50 x
       50 (50:99, 0:49);
```

Listing 7: Output of MPI Program for Distributed IO

The demonstration code showcases the use of **MPI-IO** to write a distributed 2D grid into a single binary output file efficiently. The grid is decomposed across MPI processes, with each process handling a subdomain. The grid values are initialized to the MPI rank, and the data is written collectively to a binary file. Below are the main components:

Domain Decomposition: The global grid is divided into subdomains using a `2D Cartesian decomposition`. `MPI_Dims_create` determines the decomposition dimensions, while `MPI_Cart_create` creates a Cartesian topology to simplify communication and indexing. The `decomp1d` function ensures balanced workload distribution across processes.

Memory Allocation and Initialization: Each process allocates memory for its subdomain and initializes values to its MPI rank, enabling easy identification of ownership in the output.

MPI-IO for Data Output:

- **Subarray Types:** `MPI_Type_create_subarray` defines the layout of each process's subdomain within the global grid.
- **File View:** `MPI_File_set_view` sets the global file view for each process, ensuring correct placement of data.
- **Collective Writing:** `MPI_File_write_all` performs a collective I/O operation, ensuring synchronization and optimized performance.

BOV File Creation: After data writing, the root process generates a **BOV (Brick of Values)** header file, enabling visualization tools like `VisIt` to interpret the binary data correctly.

Performance Considerations: The domain decomposition ensures load balancing, while the use of MPI-IO and Cartesian topology supports scalability. Collective I/O minimizes file system contention, and derived datatypes ensure efficient data placement.

4.3. Strong scaling

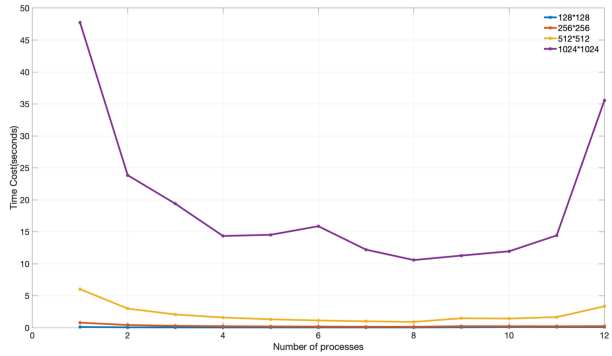

Figure 1: Strong scaling

Smaller resolutions (128 × 128, 256 × 256): When utilizing the parallel version of the PDE solver at these resolutions, performance does not improve and may even degrade. The reason for this condition might be that numerical procedures at this resolutions are computationally inexpensive. The parallel approach may even reduce performance owing to the expense of generating and managing threads.

Higher resolutions (e.g., 512 × 512 and 1024 × 1024) improve speed by employing several threads. The improvement is most noticeable at 1024 × 1024 resolution. Larger grid sizes are computationally more costly than smaller grids. In certain cases, the parallel version can greatly improve performance. The non-linear time cost can be ascribed to the trade-off between the expense of managing processes and the benefits derived from employing numerous processes.

4.4. Weak scaling

The result is that as the number of processes increases, the time cost will rise proportionally. The reason for this is that, while increasing the number of processes, the communication overhead associated with the greater problem size is computationally expensive, resulting in an increase in

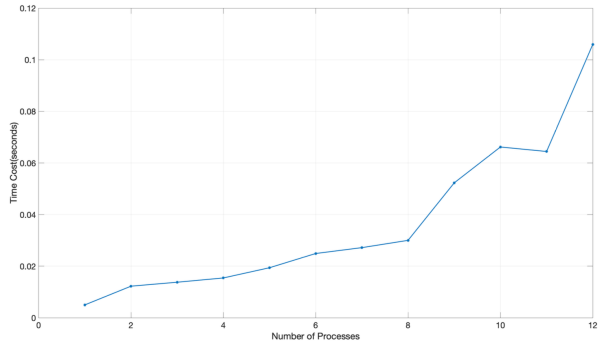

Figure 2: Weak scaling

time consumed. Furthermore, increasing the number of nonlinear iterations can degrade the PDE solver's performance.

5. Task 5 - Testing

5.1. Sum of ranks: MPI collectives

```
    sum_of_ranks = comm.allreduce(rank, op=MPI.SUM)
    if rank == 0:
        print(f"Sum of all ranks (pickle-based): {sum_of_ranks}")
    comm.Barrier()

    rank_array = np.array(rank, dtype='i')
    sum_array = np.zeros(1, dtype='i')
    comm.Allreduce([rank_array, MPI.INT], [sum_array, MPI.INT], op=MPI.SUM
        )
```

Listing 8: MPI-based Sum of Ranks Calculation in Python

5.2. Ghost cell exchange between neighboring processes

```
    dims = MPI.Compute_dims(size, [0, 0])
    rows, cols = dims
    periodic = [True, True]
    reorder = False
    cart_comm = comm.Create_cart(dims, periods=periodic, reorder=reorder)
    coords = cart_comm.Get_coords(rank)
    north, south = cart_comm.Shift(0, 1)
    east, west = cart_comm.Shift(1, 1)
```

Listing 9: Creating a 2D Cartesian Topology in MPI

5.3. A self-scheduling example: Parallel Mandelbrot

```
    def manager(comm, tasks):
        completed = []
        num_workers = comm.Get_size() - 1
        for worker_id in range(1, num_workers + 1):
            comm.send(tasks[worker_id - 1], dest=worker_id, tag=TAG_TASK)
        for task_index in range(num_workers, len(tasks)):
            finished = comm.recv(source=MPI.ANY_SOURCE, tag=MPI.ANY_TAG)
            completed.append(finished)
```

```
            comm.send(tasks[task_index], dest=finished.Get_source(), tag=
                TAG_TASK)
        for _ in range(1, num_workers + 1):
            final_task = comm.recv(source=MPI.ANY_SOURCE, tag=MPI.ANY_TAG)
            completed.append(final_task)
            comm.send(None, dest=final_task.Get_source(), tag=TAG_DONE)
        return completed

    def worker(comm):
        task_flag = -1
        tasks_processed = 0
        while task_flag != TAG_DONE:
            task = comm.recv(source=MANAGER, tag=MPI.ANY_TAG)
            task_flag = task.Get_tag()
            if task_flag != TAG_DONE:
                task.process()
                comm.send(task, dest=MANAGER, tag=TAG_TASK_DONE)
            tasks_processed += 1
        return tasks_processed
```

Listing 10: Manager-Worker Model in MPI