Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

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

Student: CARLA JUDITH LOPEZ ZURITA Discussed with: FULL NAME

Solution for Project 5

Due date: Monday 13 May 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.

Project 5 is about continuing the work on the parallel solution of a nonlinear PDE using MPI and learning about Python for High-Performance Computing.

1. Parallel Space Solution of a nonlinear PDE using MPI [in total 60 points]

Following the steps of Project 3, we will parallelize the solution of the diffusion equation using MPI. We will also implement parallel I/O to write the solution to disk. We will then evaluate the strong and weak scaling of our parallel implementation.

1.1. Initialize/finalize MPI and welcome message [5 Points]

The code should initialize MPI, get the number of processes, and print a welcome message. After adding the header for the library, only a few lines of code were changed to initialize MPI and print the welcome message. The following code snippet shows the changes made to the code:

```
std::cout << "version :: C++ MPI" << std::endl;
std::cout << "processes :: " << size << std::endl;
```

As well as finalizing MPI at the end of the code and adding relevant print conditions for only the master thread.

1.2. Domain decomposition [10 Points]

On the `data.cpp` file, some changes were added to the SubDomain struct very similar to those in Project 3. The strategy implemented is based in determining automatically the number of subdomains in the x and y dimensions using the `MPI_Dims_create` function. The following code snippet shows the changes made to the code:

```
int dims[2] = {0, 0};
MPI_Dims_create(mpi_size, 2, dims);
ndomy = dims[0];
ndomx = dims[1];
```

We also added the `MPI_Cart_create` function to create the Cartesian as well as the `MPI_Cart_coords` function to determine the coordinates of each process and the `MPI_Cart_shift` function to determine the neighbors. We also added some conditions to adjust the cases for when a dimension is not divisible by the number of processes.

TODO: Discuss why this method was selected and analyze its implications on the performance of the application.

1.3. Linear algebra kernels [5 Points]

Next, we parallelized the relevant linear algebra kernels in `hpc_XXX` in `linalg.cpp` using MPI. Not all funcitons needed to be modified, only the ones that required communication between processes, such as `hpc_dot` and `hpc_norm2`. Both follow the same pattern of using the `MPI_Allreduce`. The code snippet below shows the changes made to the `hpc_dot` function:

```
// computes the inner product of x and y
// x and y are vectors on length N
double hpc_dot(Field const& x, Field const& y) {
    double global_result;
    double local_result = 0.0;
    int N = y.length();
    // compute local result
    for (int i = 0; i < N; i++) {
        local_result += x[i] * y[i];
    }
    MPI_Allreduce(&local_result, &global_result, 1, MPI_DOUBLE, MPI_SUM,
        MPI_COMM_WORLD);
    return global_result;
}
```

The rest of the functions were left unchanged.

1.4. The diffusion stencil: Ghost cells exchange [10 Points]

The next step was to implement the exchange of ghost cells between neighboring processes. This was done by adding the `MPI_Isend` and `MPI_Irecv` functions to the `diffusion` function in `operators.cpp`. This requeired following a series of steps. First, we needed to fill the send buffer with the values of the ghost cells. This was a bit tricky becase the indices needed to be inverted.

```
// Fill buffers
for (int k = 0; k < nx; k++)
{
    // The indices needed to be inverted
    buffN[k] = s_new(k,jend);
    buffS[k] = s_new(k,0);
}
```

```
for (int k = 0; k < ny; k++)
{
    buffE[k] = s_new(iend,k);
    buffW[k] = s_new(0,k);
}
```

Then, we needed to send the values and receive the values from the neighbours using non-blocking point-to-pointcommunication. A custom data type was not needed since we were sending contiguous data. The following code snippet shows the changes made to the code:

```
MPI_Request request[8];
int tag1 = 1, tag2 = 2, tag3 = 3, tag4 = 4;
int count = 0;

// SEND
// to top
MPI_Isend(&buffN[0], nx, MPI_DOUBLE, domain.neighbour_north, tag1, domain.
    comm_cart, &request[count++]);
// to bottom
MPI_Isend(&buffS[0], nx, MPI_DOUBLE, domain.neighbour_south, tag2, domain.
    comm_cart, &request[count++]);
// to right
MPI_Isend(&buffE[0], ny, MPI_DOUBLE, domain.neighbour_east, tag3, domain.
    comm_cart, &request[count++]);
// to left
MPI_Isend(&buffW[0], ny, MPI_DOUBLE, domain.neighbour_west, tag4, domain.
    comm_cart, &request[count++]);

// RECEIVE
// from top
MPI_Irecv(&bndN[0], nx, MPI_DOUBLE, domain.neighbour_north, tag2, domain.
    comm_cart, &request[count++]);
// from bottom
MPI_Irecv(&bndS[0], nx, MPI_DOUBLE, domain.neighbour_south, tag1, domain.
    comm_cart, &request[count++]);
// from right
MPI_Irecv(&bndE[0], ny, MPI_DOUBLE, domain.neighbour_east, tag4, domain.comm_cart
    , &request[count++]);
// from left
MPI_Irecv(&bndW[0], ny, MPI_DOUBLE, domain.neighbour_west, tag3, domain.comm_cart
    , &request[count++]);
```

Finally, we added a `MPI_Waitall` function to wait for all the communications to finish.

1.5. Implement parallel I/O [10 Points]

Finally, we implemented parallel I/O to write the solution to disk. We used as base the code provided in the assignment. The code in `main.cpp` was modified to use a new version of `write_binary` that uses MPI I/O. The function is based on the `MPI_File_write_at_all` function, calling on each subdomain to write its own data. One of the main changes was to change the order to `MPI_ORDER_FORTRAN` to match the order of the data in the file. Nevertheless, the final output was not as expected, some parts of the image were inverted on the y axis.

1.6. Strong scaling [10 Points]

I tried to compute the strong scaling of the code, but some configurations were not working. The error that I faced was that the code was not converging under the same number of iterations and timesteps when using more than 4 processes. I tried to change the number of iterations and timesteps, but the results were not consistent, converging on some cases and not others.

1.7. Weak scaling [10 Points]

For the same reason, I haven't done the weak scaling.

1.8. Bonus [20 Points]: Overlapping computation/computation details

2. Python for High-Performance Computing [in total 40 points]

In this section of the project, we will learn about MPI for Python and use it to parallelize the computation of the Mandelbrot set. We will use a manager-worker pattern to distribute the computation of the Mandelbrot set among the workers.

Every python code using mpi4py is initialized with the following lines:

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()
proc = MPI.Get_processor_name()
```

2.1. Sum of ranks: MPI collectives [5 Points]

Using MPI for Python's collective communication methods, we wrote a script that computes the sum of all ranks. This was done using the pickle-based communication of generic Python objects as well as the fast, near C-speed, direct array data communication of buffer-provider objects. The following code snippets show the changes made to the code using lower case functions:

```
    global_sum = comm.gather(rank)
    if rank==0:
    print(f"Sum: {sum(global_sum)}")
```

as well as using the uppercase functions:

```
    rank_array = np.array(rank, dtype=np.int32)
    total_sum = np.zeros(1, dtype=np.int32)
    comm.Allreduce(rank_array, total_sum, op=MPI.SUM)

    if rank==0:
    print(f"Sum: {sum(total_sum)}")
```

As the description of the assignment suggests, the uppercase functions are more efficient and also need to use arrays to store the data, otherwise the code will not work. Running the code with `salloc -n 8 --nodes=4 --ntasks-per-node=2`, the output for both was:

```
Sum: 28
```

Both methods worked as expected (7+6+5+4+2+3+1+0).

2.2. Ghost cell exchange between neighboring processes [5 Points]

Now, we implemented the exchange of ghost cells between neighboring processes using MPI for Python.

2.3. A self-scheduling example: Parallel Mandelbrot [30 Points]

The final task was the most fun for me. I implemented the manager-worker pattern to distribute the computation of the Mandelbrot set among the workers. The manager was responsible for distributing the work and collecting the results, while the workers were responsible for computing the Mandelbrot set for a specific region of the complex plane. The manager was also responsible for plotting the final image. The code was based on the code provided in the assignment. The following code snippet shows the implementation of the manager: