Parallel Assignment 2

Hangbiao Meng, Zhiping Hu

April 21, 2025

1 Problem Description

In this assignment, we apply and **parallelize** a one-dimensional stencil to an array of function values $f(x)$ at N discrete points $x_0, x_1, \ldots, x_{N-1}$ over the interval $0 \leq x < 2\pi$. Each point is given by $x_i = i \cdot h$, where $h = \frac{2\pi}{N}$.

Applying the stencil to an element $v_0 = f(x_i)$ involves computing the following weighted sum:

$$\frac{1}{12h} \cdot v_{-2} - \frac{8}{12h} \cdot v_{-1} + 0 \cdot v_0 + \frac{8}{12h} \cdot v_{+1} - \frac{1}{12h} \cdot v_{+2}$$

Here, v_{-j} represents the element j steps to the left of v_0, i.e., $f(x_{i-j})$, and v_{+j} is the element j steps to the right of v_0, i.e., $f(x_{i+j})$.

2 General Parallelization

2.1 Array Decomposition

We distribute the array evenly across all processes. Considering that the number of processes might not divide the array size evenly, we followed the approach used in Assignment 1 — **distributing the remainder** evenly among the first few processes.

2.2 Period Neighborhood and Boundary Conditions

In original serial code, professor applied **modulo operation** using $(index+numvalues)\%num\_values$ to warp back into the valid range. This effectively connects the ends of a one-dimensional space, forming a circular boundary. We made a simple modification to the code by **applying periodic boundary conditions to the processes** in the one-dimensional space, so that the leftmost and rightmost processes are also considered 'neighbors' and exchange boundary data. See Figure 1

```
int left_rank  = (rank - 1 + size) % size;
int right_rank = (rank + 1) % size;
```

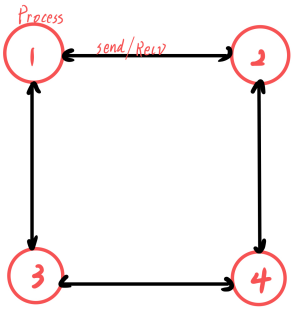

Figure 1: *Assuming that processes 1 and 3 in the diagram represent the first and last processes respectively, then process 1 can communicate with process 3, exchanging the head and tail data of the array in order to implement the same boundary handling method as in the professor's code.*

2.3 Extra Region(Padding)

When performing convolution operations on images, CNNs typically apply padding to preserve the shape of the tensor[1]. Inspired by this, each process now allocates additional space of length $EXTENT = STENCIL\_WIDTH/2 = 2$ on both sides of its local array to store the elements needed for computing the boundary values at the beginning and end (In the next section we will see how to do this using MPI functions). Figure 2 shows the padding method we use:

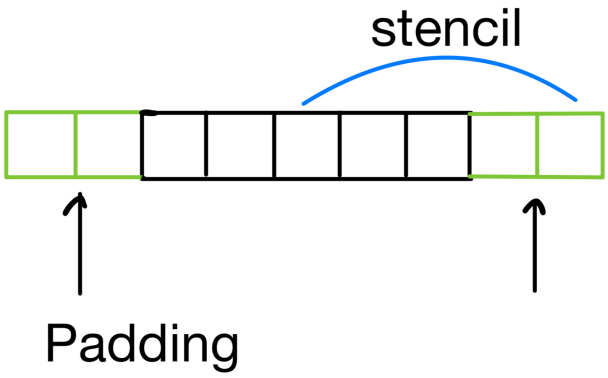

Figure 2: Extra Region(Padding), green color shows the padding area where we receive from nearby processes

2.4 Reducing memcpy: double buffering

Inspired by the pointer-swapping method in the professor's code, we also implemented **double buffering** for each process in our parallel assignment. That is, we use $buf\_current$ and buf_next, and after each computation, we swap the current and next buffers. This approach reduces the need for$memcpy$ during multiple iterations and helps speed up the program.

Listing 1: Double buffer allocation for each process

```
double *buf_current = (double*)malloc((local_N + 2 * EXTENT) * sizeof(double
    ));
double *buf_next    = (double*)malloc((local_N + 2 * EXTENT) * sizeof(double
    ));
```

3 MPI Communication Functions and Parallel Algorithm

3.1 Main MPI Functions:

In this assignment, apart from some basic functions for MPI, such as **MPI_Init / MPI_Finalize, MPI_Comm_rank / MPI_Comm_size, MPI_Reduce and MPI_Barrier** we use these functions to help us parallelize serial codes.

1. **MPI_Bcast**: Broadcast array length to all processes.
2. **MPI_Scatterv / MPI_Gatherv**: The reason for using Scatterv / Gatherv instead of Scatter / Gather is that the data may not be evenly distributed among processes (i.e., when num_values % size != 0).
3. **MPI_Isend / MPI_Irecv**: To **reduce the waiting time** for processes to receive padding data, we used this pair of functions so that the program can continue performing other computations while waiting for data transmission to complete.
4. **MPI_Request and MPI_Waitall**: These two functions are used together[2]. In this assgnment, we set $MPI\_Request \;\; requset[4]$ for two MPI_Isend operations and two MPI_Irecv

operations. Thus, our processes can first compute the portion of the data that is not affected by padding while waiting for the non-blocking communication. Once MPI_Waitall completes, it is safe to compute the values at the left and right boundaries.

3.2 Parallel Algorithm

The pseudocode 2 shows the main algorithm we use in this assignment. Algorithm 3 and 4 show the calculation method before and after MPI_Waitall.

Listing 2: Main MPI Function

```
main(argc, argv):
    MPI_Init
    parse (input_file, output_file, num_steps)
    if (rank == 0) read global_input
    MPI_Bcast(num_values)

    define STENCIL
    compute local_N and displacements
    allocate buf_current, buf_next

    MPI_Scatterv(global_input -> buf_current[EXTENT..])
    compute left_rank, right_rank
    MPI_Barrier;
    start_time = MPI_Wtime()

    for step in 0..num_steps-1:
        MPI_Irecv padding zones (left/right)
        MPI_Isend boundaries (left/right)

        // See listing 3 and 4 for detailed information
        compute interior
        MPI_Waitall
        compute boundary zones


        swap(buf_current, buf_next)

    MPI_Barrier; end_time = MPI_Wtime()
    // Here reduce the max time using MPI_MAX
    MPI_Reduce(local_elapsed = (end_time - start_time) -> max_elapsed)
    MPI_Gatherv(buf_current -> final_output)

    if (rank == 0)
        print(max_elapsed)
        optionally write final_output

    free resources
    MPI_Finalize
```

Listing 3: Compute Interior Region

```
compute_interior(buf_current, buf_next, STENCIL, EXTENT, eff_start, eff_end)
   :
    for i in (eff_start + EXTENT) .. (eff_end - EXTENT - 1):
        sum = 0.0
        for j in -EXTENT .. EXTENT:
            sum += STENCIL[j + EXTENT] * buf_current[i + j]
        buf_next[i] = sum
```

Listing 4: Compute Boundary Regions

```
compute_boundary(buf_current, buf_next, STENCIL, EXTENT, eff_start, eff_end)
    :
    // Left boundary: [eff_start, eff_start + EXTENT)
    for i in eff_start .. (eff_start + EXTENT - 1):
        sum = 0.0
        for j in -EXTENT .. EXTENT:
            sum += STENCIL[j + EXTENT] * buf_current[i + j]
        buf_next[i] = sum

    // Right boundary: [eff_end - EXTENT, eff_end)
    for i in (eff_end - EXTENT) .. (eff_end - 1):
        sum = 0.0
        for j in -EXTENT .. EXTENT:
            sum += STENCIL[j + EXTENT] * buf_current[i + j]
        buf_next[i] = sum
```

4 Performance Experiments and Discussion

4.1 Performance Experiments

The parallelization tool used in this experiment is **OpenMPI 5.0.5**, and the testing platform is **Rackham**.

For **strong scalability**, we tested problem sizes of 1,000,000; 2,000,000; 4,000,000; and 8,000,000 using 1, 2, 4, 8, and 16 MPI nodes(2500 iterations), and measured the execution time according to the requirements specified in Assignment 2. Every experiment repeats 5 times to get the average.

For **weak scalability**, we tested with problem size of 1,000,000 and 2,000,000 per MPI node.

The results are presented below.

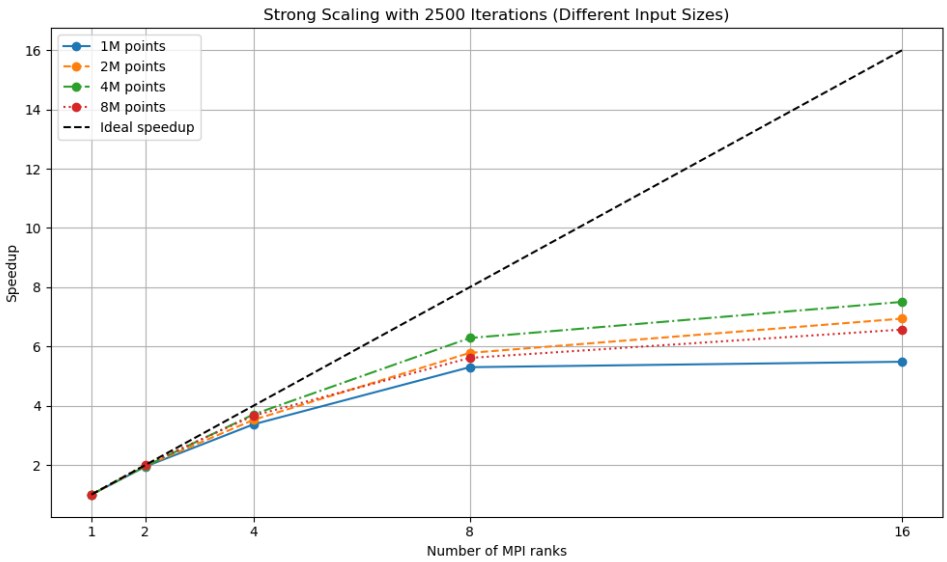

Figure 3: Strong scalability

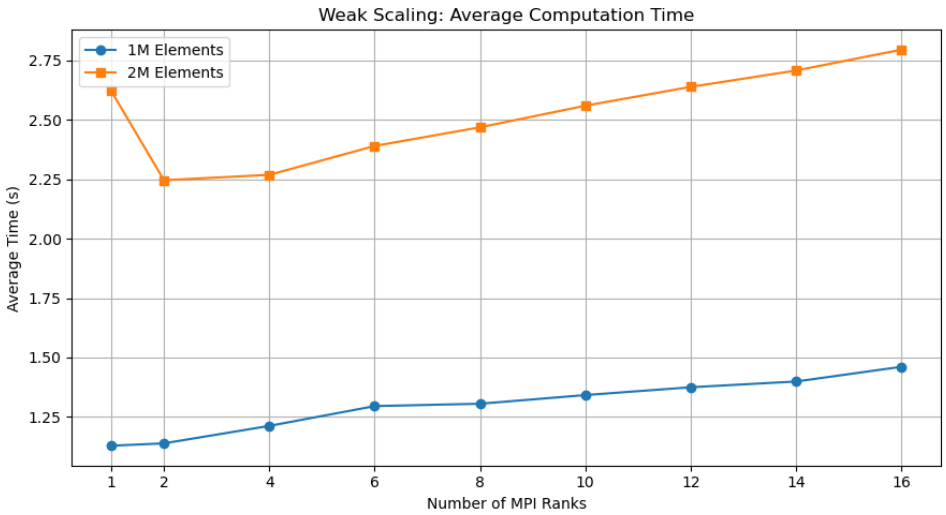

Figure 4: Weak scaling computation time

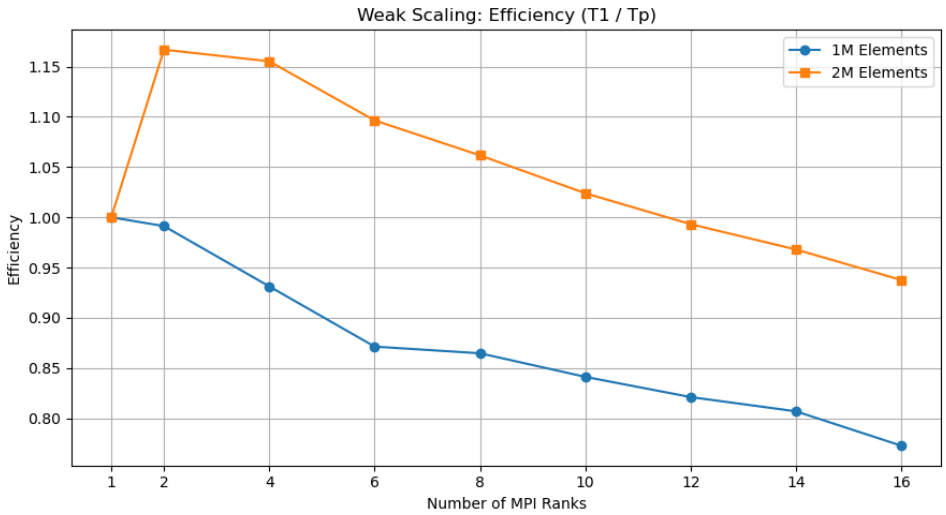

Figure 5: Weak scaling efficiency

4.2 Discussion

The results above generally align with our expectations and are consistent with the findings from the previous assignment. According to Amdahl's Law, the parallel speedup is limited by the portion of the code that cannot be parallelized. Even if only a small part of the code is serial, it can significantly restrict the maximum overall speedup[3].

In Figure 3, as the number of processes increases, the parallel efficiency decreases. This is likely related to potential parallel workload imbalance and data exchange bottlenecks. Notably, as the problem size increases, the performance does not improve consistently — in fact, the parallel efficiency drops for the 8,000,000 data size. We suspect this may be due to access limitations caused by the large volume of data.

For weak scalability, the conclusion is similar: although the workload per computing unit remains constant, for 2M elements, the efficiency was raised up to 1.15. The reason might be that, during scaling, we maintained a constant workload by increasing the number of iterations. However, the workload introduced by the number of iterations does not have a perfectly linear relationship with the problem size. Overall efficiency tends to decrease due to additional overhead from communication, synchronization, system resource contention, and scheduling.

References

[1] DigitalOcean. Padding in convolutional neural networks. `https://www.digitalocean.com/community/tutorials/padding-in-convolutional-neural-networks`, n.d. Accessed April 13, 2025.

[2] RookieHPC. Mpi_waitall, n.d. Accessed: April 13, 2025.

[3] Gene M. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485. ACM, 1967.