Parallel Assignment 2

Hangbiao Meng, Zhiping Hu

May 9, 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 Before 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. This method could make our code more robust.

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;
```

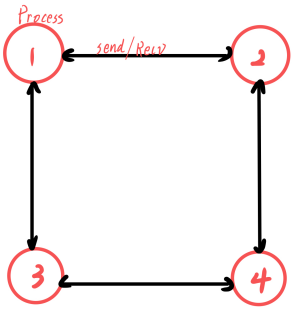

Figure 1: *Assuming that processes 1 and 3 in the diagram represent the first and last processes respectively, then process 1 can communicate with process 3, exchanging the head and tail data of the array in order to implement the same boundary handling method as in the professor's code.*

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:

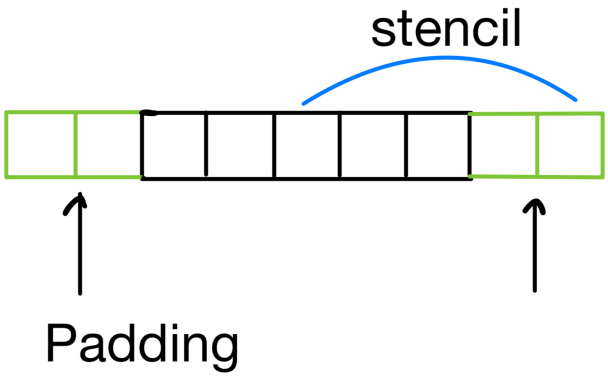

Figure 2: Extra Region(Padding), green color shows the padding area where we receive from nearby processes

2.4 Better memory usage

Inspired by the pointer-swapping method in the professor's code and open-source code which use double buffering for other parallelization method implemented in cuda[2][3](it's useful for our project), 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 memory per process during multiple iterations(The original code, however allocate a whole space with same size of input array). Additionally, in the serial code, that method applied a huge array for double buffering but in our case each process needs their own to store the results. We also should remember add 2*EXTENT for padding.

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(non-blocking communication) so that the program can continue performing other computations while waiting for data transmission to complete[4].

4. **MPI_Request and MPI_Waitall**: These two functions are used together[5]. 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)
    :
    offsets = [eff_start, eff_end - EXTENT]

    for region in 0 .. 1:
        for idx in 0 .. (EXTENT - 1):
            i = offsets[region] + idx
            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 processes (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 process.

The results are presented below.

Table 1: Strong Scaling Results: Runtime, Speedup, and Efficiency (2500 Iterations)

Problem Size	# of Ranks	Time (s)	Speedup	Efficiency (%)
1M	1	4.136	1.00	100.0
	2	2.127	1.94	97.2
	4	1.227	3.37	84.2
	8	0.780	5.30	66.3
	16	0.754	5.49	34.3
2M	1	8.250	1.00	100.0
	2	4.198	1.96	97.8
	4	2.342	3.52	88.0
	8	1.426	5.79	72.4
	16	1.189	6.94	43.4
4M	1	16.558	1.00	100.0
	2	8.451	1.96	97.9
	4	4.468	3.71	92.7
	8	2.634	6.29	78.7
	16	2.206	7.51	46.9
8M	1	29.721	1.00	100.0
	2	14.804	2.01	100.4
	4	8.112	3.66	91.5
	8	5.295	5.61	70.1
	16	4.523	6.57	41.1

Table 2: Weak Scaling Time (in seconds)

Ranks	1M Problem Time	2M Problem Time
1	1.12886	2.62112
2	1.13886	2.24634
4	1.21228	2.26888
6	1.29563	2.39058
8	1.30558	2.46892
10	1.34201	2.55990
12	1.37492	2.63926
14	1.39909	2.70796
16	1.46107	2.79568

Table 3: Weak Scaling Efficiency

Ranks	1M Efficiency	2M Efficiency
1	1.0000	1.0000
2	0.9912	1.1668
4	0.9312	1.1551
6	0.8715	1.0963
8	0.8647	1.0616
10	0.8410	1.0238
12	0.8211	0.9933
14	0.8067	0.9689
16	0.7723	0.9374

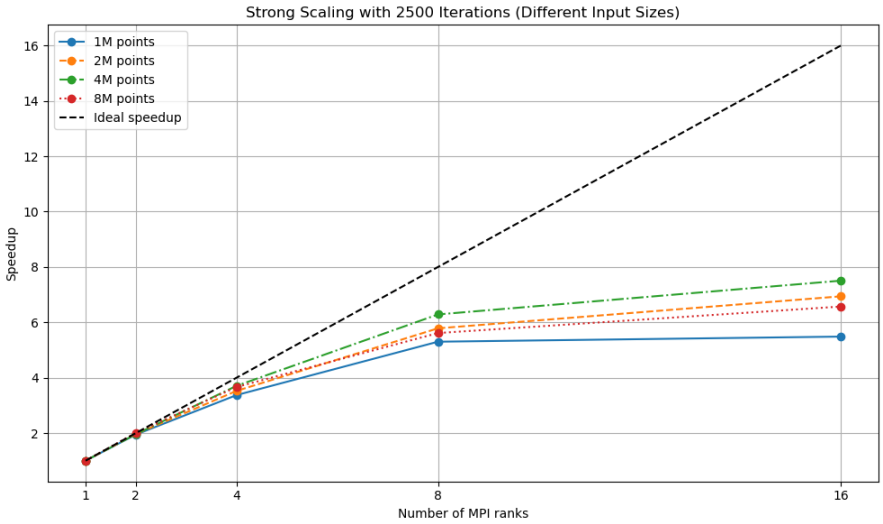

Figure 3: Strong scalability

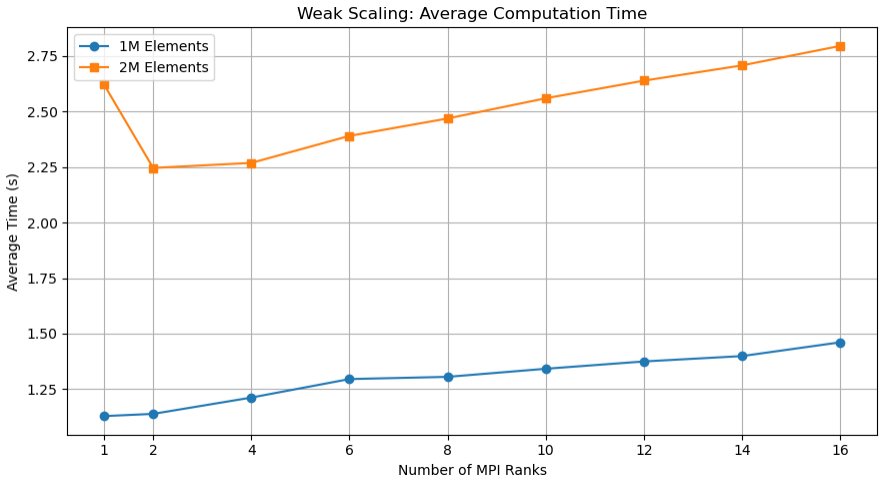

Figure 4: Weak scaling computation time

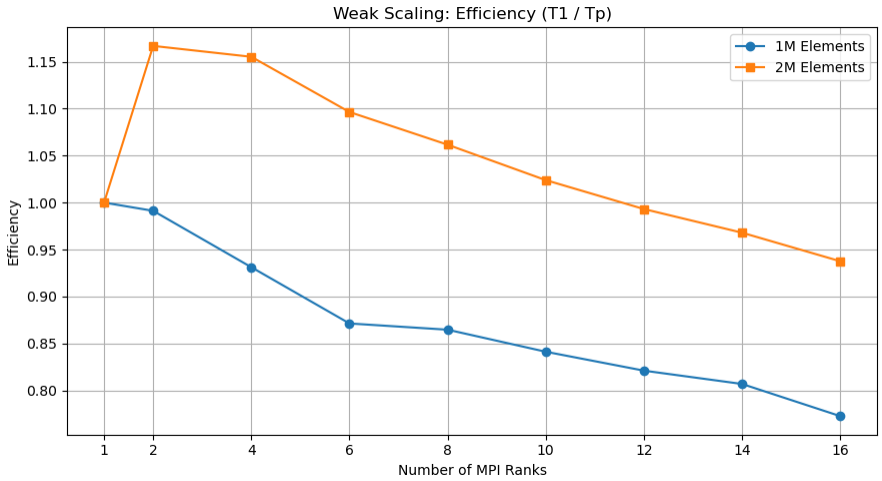

Figure 5: Weak scaling efficiency

5 Discussion

After the experiment, we find the results below:

1. **Fixed Problem Size – Scaling Limitations:** When the problem size is fixed and the number of processes increases, the efficiency gain diminishes.

2. **Fixed Process Count – Problem Size Increases:** Conversely, when the number of processes is fixed and the problem size increases, we observe that parallel efficiency does not consistently improve. In some cases, efficiency even drops for larger problems (e.g., 8M).

These results align with expectations and are consistent with previous findings. According to Amdahl's Law, parallel speedup is fundamentally limited by the serial portions of the program[6]. Even small non-parallelizable sections can significantly restrict scalability.

Apart from that, we suspect that **Cache Effects** is the other reason behind why we get these results: The CPUs in Rackham has a cache structure: L1 (32KB), L2 (256KB), and shared L3 (25MB per socket). With 1 rank, the entire dataset (e.g., 8M doubles = 61MB) exceeds cache capacity, resulting in frequent cache misses. As the number of ranks increases (e.g., 2–8), each rank handles a smaller portion of the data, improving cache residency and computation speed. For example, with

8 ranks, each processes about 8MB—more likely to fit within L3 or L2 cache—leading to significant speedup and even superlinear efficiency in some cases.

However, beyond 8 ranks, the computational workload per rank becomes so small (e.g., 500KB at 16 ranks for 1M problem size) that while cache efficiency peaks, communication latency shows. The time spent in MPI calls like `MPI_Waitall` or other necessary functions used for MPI becomes significant. Additionally, **false sharing** may also be a reason for the decrease in efficiency. As the number of processes increases, different processes accessing nearby cache lines could lead to false sharing. The combined effect of these factors has led to a decline in parallel efficiency.

Weak Scalability: For weak scalability, the conclusion is similar. 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. Additionally, the improvement in this aspect may be due to amortization of the initial overhead or incidental benefits from system scheduling biases. Although the workload of each process remains nearly constant, the repeated amplification of per-iteration overhead—such as subtle communication delays and synchronization issues—also leads to a decline in overall efficiency.

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] ifromeast. Cuda learning - sgemm v3. `https://github.com/ifromeast/cuda_learning/blob/main/03_gemm/sgemm_v3.cu`, 2023. Accessed: 2025-04-21.

[3] xlite dev. Cuda learn notes. `https://github.com/xlite-dev/CUDA-Learn-Notes`, 2024. Accessed: 2025-04-21.

[4] Dinei. Mpi: blocking vs non-blocking, 2017. Accessed: 2025-04-18.

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

[6] 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.