Parallel Assignment 1

Hangbiao Meng, Zhiping Hu

April 4, 2025

1 Assignment 1 Tasks

1.1 Task 2.1: Formulas for block partitioning

In this exercise, we are going to compute the sum of an array of length n using p processing units, assuming 0-based indexing, i.e., array indices range from 0 to $n-1$.

Let

$$q = \left\lfloor \frac{n}{p} \right\rfloor, \quad r = n \bmod p.$$

Where q is the minimum number of elements each block receives when dividing n elements evenly into p blocks(that is quotient) and r is the number of extra elements, which should be distributed among the first r processes to ensure load balancing. $\lceil x \rceil$ is ceiling function and $\lfloor x \rfloor$ is floor function[1].

Let the process index be $k = 0, 1, \ldots, p-1$. Then, the **starting index** first(k) and the **ending index** last(k) of the block assigned to process k are defined as:

$$\text{first\_index}(k) = kq + \min(k, r),$$

$$\text{last\_index}(k) = \text{first\_index}(k) - 1 + q + \begin{cases} 1, & \text{if } k < r; \\ 0, & \text{if } k \geq r; \end{cases}$$

In this way, the first r processes (i.e., when $k < r$) receive one extra element each. For example, if we have an array with 11 elements and 3 units. Then the first unit and the second units will have 4 elements to calculate and the last one only have 3 elements. The indices are (0,3), (4,7) and (8,10), This ensures that the r extra elements are evenly distributed among the first few blocks, achieving a more balanced workload.

1.2 Task 2.2: Tree-structured global sum

1.2.1 Main algorithm

In this task, we are going to compute global sum using the tree-structured algorithm sketched in Figure 1.1 from our textbook for p processing units[2]. Since the number of processes is not always a power of two, the remaining elements need to be carefully handled during the tree-based reduction.

- **Expansion**:
  - Assume we expansion to 2^n processes, where $n = \lceil \log_2(p) \rceil$.
  - Example: For $p = 5$, expand to $2^3 = 8$. Apply boundary checks ignore non-existent processes.
- **Reduction Paths** (see Fig. 1 and Fig. 2. The blue line shows the boundary of subtrees):
  - *Left subtree* ($0 \sim 2^{n-1} - 1$): Standard half reduction (distance=2,4,...).
  - *Right subtree* ($2^{n-1} \sim p - 1$): Differently send data path (assume virtual nodes).

  - *Before the last reduction*: The left subtree will send the partial sum to node 0 and the right subtree will send the partial sum to node $2^{ceil(2^{log_2(p)})}/2$

- **Boundary Handling**:
  - Check `if (source < numprocs)` to avoid invalid communication.
  - Extra processes participate only in the final reduction step.

Example: $p = 5$, see Fig, 1

Distance	**Operations**
2	$0 \leftarrow 1$, $2 \leftarrow 3$, no action for 4
4	$0 \leftarrow (2 \leftarrow 3)$, no action for 4
8	$0 \leftarrow 4$ (direct send, here distance=8 $<=$ $2^{ceil(2^{log_2(5)})} = 8$)
Result	$S_0 = \sum_{i=0}^{4} \text{my\_sum}_i$

Algorithm 1 Tree Reduction Algorithm

```
Initialize distance = 2, max_distance = 1 << ceil(log2(numprocs))
while distance $\leq$ max_distance do
    if myid % distance == 0 then
        Receive from myid + distance/2 (if valid)
        Accumulate total_sum
    else if myid % distance == distance/2 then
        Send total_sum to myid - distance/2
        break
    end if
    distance *= 2
end while
```

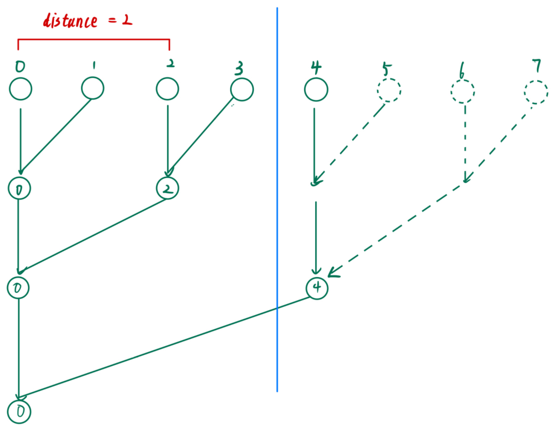

Figure 1: 5 process tree

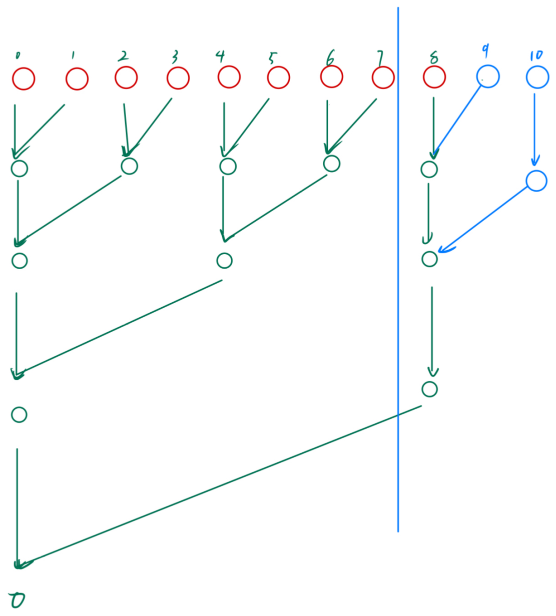

Figure 2: 11 process tree

1.2.2 Parallelization Description and Results

The parallelization tool used in the experiment is **OpenMPI 5.0.5**, and the testing platform is **Rackham**. Tests were conducted according to the assignment requirements for both **strong scalability** and **weak scalability**.

As specified in the assignment, each process generated random numbers, performed local summation, and applied reduction.

If we want to calculate the results from an array, we should first use *scatterv* function[3] to distribute the array using the algorithm in Task 2.1.

Remark: The correctness of the results was first verified through arithmetic series summation:

$$\sum_{i=1}^{n=671087864} i = 2251799847239680 \tag{1}$$

and

$$\sum_{i=1}^{n=10} i = 55 \tag{2}$$

Output from terminal:

Total sum: 2251799847239680.000000 Proc=8 Problem size=67108864 Time=0.069214 sec

and:

Total sum: 55.000000 Proc=11 Problem size=10 Time=0.000082 sec

Table 1: Strong Scalability Results

Processes	Execution Time (s)	Efficiency
1	1.783749	1.000000
2	0.906651	0.983702
4	0.492616	0.905243
8	0.253452	0.879727

Table 2: Weak Scalability Results

Processes	Execution Time (s)	Efficiency
1	0.013553	1.000000
2	0.018049	0.375450
4	0.020860	0.162428
8	0.021284	0.079596

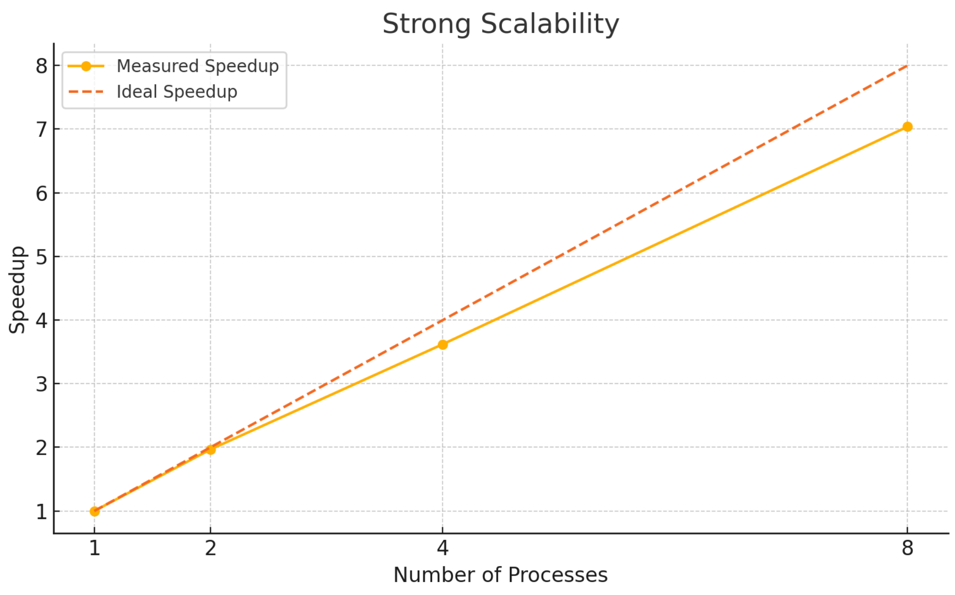

Figure 3: Strong Scalability Results

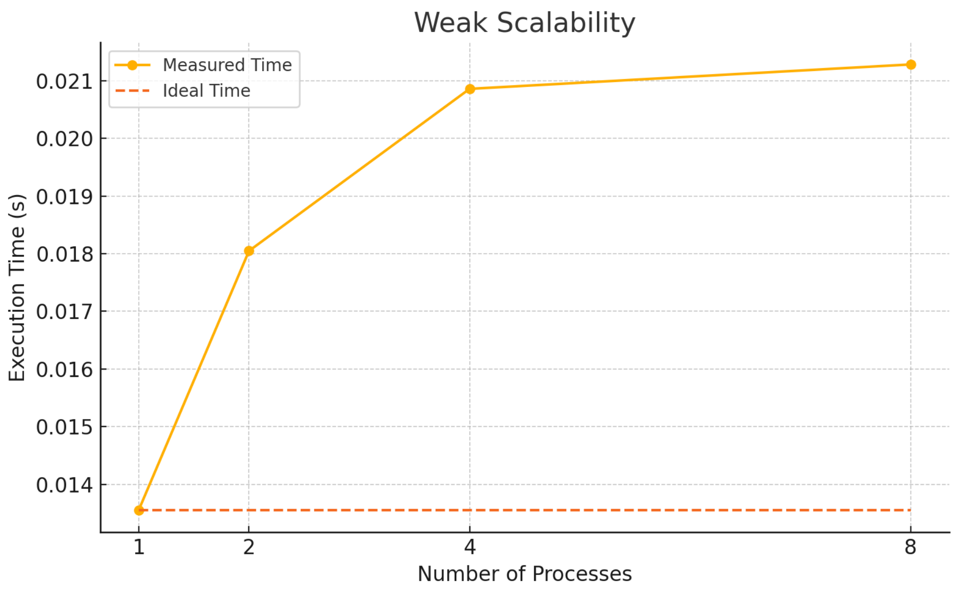

Figure 4: Weak Scalability Results

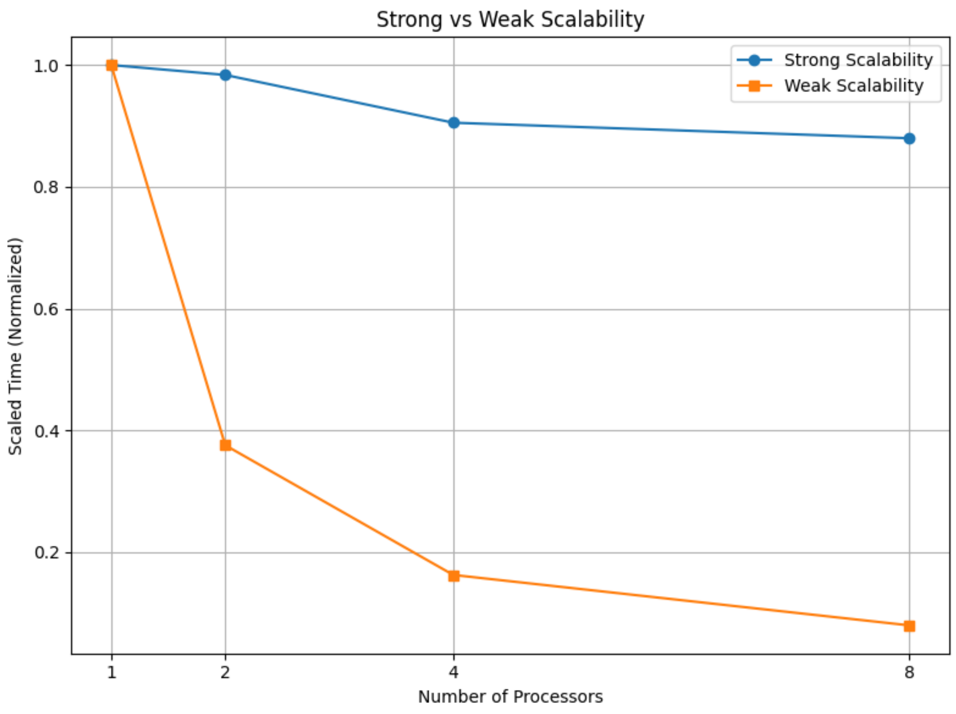

Figure 5: The efficiency of weak and strong scalability

1.2.3 Discussion

The strong scalability results closely follow expectations but are slightly below the ideal linear speedup due to inherent communication overhead.

However, as processors increase, incremental improvements in speedup diminish, indicating reduced parallel efficiency.

Communication overhead, synchronization, and data reduction (`MPI_Send`/`MPI_Recv`) contributed to the diminishing returns, especially noticeable in the weak scalability test. This can be found in Figure 4 that shows the decline of the efficiency for both methods.

1.3 Task 2.3: Cost analysis of tree-structured global sum algorithm

Based on the hints provided in the course material[4] and the results shown in the two figures: Fig.1 and 2, if we consider the concurrently executed additions and receive operations in each parallel step as a single operation, then the formula should be:

$$T(p) = \lceil \log_2(p) \rceil r \; + \; \lceil \log_2(p) \rceil a \tag{3}$$

This is also consistent with the conclusion presented in the textbook.

1.4 Task 2.4: Speedup and efficiency

Table 3: Speedup Table

	10	20	40	80	160	320
1	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
2	1.9608	1.9900	1.9975	1.9994	1.9998	2.0000
4	3.7037	3.9216	3.9801	3.9950	3.9988	3.9997
8	6.4516	7.5472	7.8818	7.9701	7.9925	7.9981
16	9.7561	13.7931	15.3846	15.8416	15.9601	15.9900
32	12.3077	22.8571	29.0909	31.2195	31.8012	31.9501
64	13.2231	32.6531	51.6129	60.3774	63.0542	63.7609
128	12.8514	39.5062	82.0513	112.2807	123.6715	126.8897

Table 4: Efficiency Table

	10	20	40	80	160	320
1	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
2	0.9804	0.9950	0.9988	0.9997	0.9999	1.0000
4	0.9259	0.9804	0.9950	0.9988	0.9997	0.9999
8	0.8065	0.9434	0.9852	0.9963	0.9991	0.9998
16	0.6098	0.8621	0.9615	0.9901	0.9975	0.9994
32	0.3846	0.7143	0.9091	0.9756	0.9938	0.9984
64	0.2066	0.5102	0.8065	0.9434	0.9852	0.9963
128	0.1004	0.3086	0.6410	0.8772	0.9662	0.9913

When the **problem size is fixed** and the number of **processing units increases**: The speedup increases with the number of processors, which is evident from the Speedup Table. And the efficiency (i.e., speedup divided by the number of processors) gradually decreases as more processors are used.

Looking across any particular row (**keeping the number of processors fixed**, for example: P = 128). As the problem size increases, the speedup increases because larger problem sizes allow for more effective utilization of the processing units, making the computation dominate over the overhead. The efficiency also improves as the problem size grows, since the parallelization overhead becomes less significant compared to the total computation.

Overall, these observations make sense: for a fixed small problem size, increasing the number of processors introduces diminishing returns due to overhead, while for a fixed number of processors, larger problem sizes lead to better speedup and efficiency.

1.5 Task 2.5: Scalability

Given:

$$T_s(n) = n, \quad T_p(n,p) = \frac{n}{p} + \log_2(p). \tag{4}$$

Efficiency can be written as

$$E(n,p) = \frac{n}{n + p\,\log_2(p)}. \tag{5}$$

If we increase processors with k,

$$p^* = k \cdot p \quad (k > 1). \tag{6}$$

Then we need to find a new n that $E(n^*, p^*) = E(n,p)$. Hence, we get the equation:

$$\frac{n}{n + p\,\log_2(p)} = \frac{n^*}{n^* + p^*\,\log_2\big(p^*\big)}. \tag{7}$$

Solve for n^*:

$$n^* = n\,\frac{k\,\log_2\big(k\,p\big)}{\log_2(p)}. \tag{8}$$

Hence, the factor ρ by which to increase n:

$$\rho = \frac{n^*}{n} = k\,\frac{\log_2\big(k\,p\big)}{\log_2(p)} = k(1 + \frac{\log_2(k)}{\log_2(p)}). \tag{9}$$

1.6 Conclusion:

In summary, based on the aforementioned code implementation and mathematical calculations in Task 2.4 and Task 2.5, the results generally align with the principles I previously imaged. Specifically, **increasing the computational scale can improve parallel efficiency**. Conversely, if the computational scale is small, **excessive parallelism may lead to performance waste** caused by parallel overhead, outweighing the benefits gained from parallelization.

References

[1] Wikipedia contributors. Floor and ceiling functions — Wikipedia, the free encyclopedia, 2024. Accessed: March 29, 2025.

[2] Peter S. Pacheco. Chapter 1 - why parallel computing? In Peter S. Pacheco, editor, *An Introduction to Parallel Programming*, pages 1–14. Morgan Kaufmann, 2011.

[3] RookieHPC. Mpi_scatterv - scattering a vector with varying counts in mpi. `https://rookiehpc.org/mpi/docs/mpi_scatterv/index.html`, 2024. Accessed: March 29, 2025.

[4] Peter S. Pacheco. Chapter 3 - distributed-memory programming with mpi. In Peter S. Pacheco, editor, *An Introduction to Parallel Programming*, pages 83–149. Morgan Kaufmann, 2011.