Uppsala University

Parallel and Distributed Programming

1TD070

Assignment 2:

Parallelizing Stencil Computations for Scientific Applications

---

Students:
Houwan Liu
Linjing Shen

Lecturer:
Prof. Roman Iakymchuk

May 4, 2024

Contents

1 Introduction

In applications such as computer simulations and digital image processing, a common operation known as "Stencil" is frequently employed. A "Stencil" can be regarded as an operator used to manipulate elements within vectors, matrices, or tensors. This operation involves computing a new value based on the current element and its neighboring elements. "Stencil" operations are characterized by inherent parallelism and regular computational patterns, enabling optimizations in memory access, instruction-level parallelism, and exploration of algorithmic-level time parallelism. Familiarity with optimization techniques for stencil computations facilitates more effective application of parallel optimizations in practical scenarios, such as solving heat conduction problems or finite difference problems.
The task at hand involves applying a one-dimensional stencil on an array representing function values $f(x)$, where the function values correspond to a finite set of N values $x_0, x_1, x_2, \ldots, x_{N-1}$ within the interval $0 \leq x < 2\pi$. Each value $x_i = i \cdot h$, where $h = \frac{2\pi}{N}$. Applying the stencil on an element v_0 entails 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} represents the element j steps to the right of v_0 (i.e., $f(x_{i+j})$). This sum approximates the first derivative $f'(x)$ at $x = x_i$.

2 Parallelisation Strategy

We decided to use Persistent communication, which can be used to reduce communications overhead in programs which repeatedly call the same point-to-point message passing routines with the same arguments. They minimize the software overhead associated with redundant message setup.
In this strategy, within the main loop, we initiate non-blocking communication first and then immediately apply the stencil computation. Communication and computation operations are tightly integrated in this approach. The advantage of this implementation is its direct and straightforward nature, but it may lead to coupling between communication and computation. Data exchange is achieved through copying operations, directly copying local data into extended data arrays and performing data exchange as needed. Because set up persistent communication is done in the `setup_persistent_communications` function, which defines fixed send and receive buffers. So instead of swapping Pointers, we chose to use `memcpy` to copy the contents of `output_MPI` back to `input_MPI`.
We implemented the parallel one-dimensional stencil computation using MPI. Initially, the program initializes MPI and retrieves the number of processes and the process ID through `MPI_Init`. The root process is responsible for reading function values from the input file and then uses `MPI_Bcast` to broadcast the number of function values to all processes, informing them about the amount of data to be processed. We calculated the number of values for each process that needs to be handled and used, `MPI_Scatter`

receive input data from the root process and distribute the data to local storage for each process.
To facilitate the exchange of data between adjacent processes required for stencil computation, each process creates an extended data array that includes local data and additional data for computing boundaries. We utilize non-blocking communication operations `MPI_Isend` and `MPI_Irecv` for inter-process boundary data exchange to ensure all necessary data for stencil computation is available. This approach allows processes to initiate communication and continue with other computations without waiting for communication to complete, enabling concurrent task execution and helping to hide communication latency. Moreover, compared to blocking communication, it dynamically manages communication and computational tasks, which can be more efficient in distributed computing environments.
Stencil computation is achieved by applying a specific weight array to local data. Each process executes stencil computation in a loop, generating local output results. At the end of each iteration, processes exchange data to prepare for the next stencil computation. Finally, after completing all computations, the program uses `MPI_Gather` to collect the computation results from all processes to the root process, and then writes the final results into an output file. The program also measures the execution time for each process and uses `MPI_Reduce` to find the maximum execution time among all processes, which helps evaluate the parallel performance.

3 Performance Experiment

Our performance experiments aim to evaluate the scalability of a parallel stencil computation program. Scalability involves adjusting resources and capabilities to match application demands by expanding or reducing them.
Firstly, we conduct a strong scalability experiment by maintaining a constant problem size and incrementally increasing the number of parallel processing processes. We utilize large input files stored in the specified directory as test data to comprehensively evaluate the program's performance. Each experiment involves recording the program's execution time with varying numbers of processes and calculating speedup to assess how the parallel program performs as the number of processors changes.
Second, our plan includes performing weak scalability experiments to ensure that the problem size handled by each process remains constant as the number of processes is increased. This means that as the number of processes increases, the total amount of data processed increases accordingly, thus maintaining the proportionality between the problem size and the number of processes. We record the execution time of the program for different problem sizes and number of processes and compute the acceleration to assess whether the execution time of the parallel program remains stable for different problem sizes and number of processes.
To ensure the accuracy and reliability of our experimental results, we conduct multiple experiments for each parameter combination and derive averages for the final results. The execution time and speedup for different parameter combinations are presented using tables and graphs, and we will analyze the differences between actual speedup and ideal

speedup.

3.1 Strong scalability experiment

Table.1 and Figure.1 shows the results of our strong scaling experiment where we maintained the constant problem size at 4,000,000 and stencil applications at 10 while incrementally increasing the number of parallel processing processes. Each sub-experiment was run five times, and the averages of these runs were computed.
The formula for the acceleration ratio is as follows, where T_s is the serial execution time and T_p is the parallel execution time.

$$\text{Speedup} = \frac{T_s}{T_p}$$

Number of Processes	Execution Time (seconds)	Speedup
1	0.1535	1.0
2	0.0818	1.9
4	0.0377	4.1
6	0.0266	5.8
8	0.0229	6.7
10	0.0209	7.4
12	0.0189	8.1
14	0.0193	8.0
16	0.0189	8.1

Table 1: Strong scalability experiment

Based on the table, it is evident that as the number of parallel processing processes increases, the execution time significantly decreases. From 1 process to 16 processes, the execution time decreases from 0.1535 seconds to 0.0189 seconds, indicating that the computational resources are being utilized more effectively with the increase in the number of processes. The calculated speedup increases from 1.0 to 8.1, demonstrating a linear acceleration trend as the number of parallel processing processes increases. Particularly from 1 process to 6 processes, the speedup almost equals the ideal speedup, showcasing excellent strong scalability.
However, with further increase in the number of processes, the disparity between actual speedup and ideal speedup becomes more pronounced. It can be observed that after 12 processes, the speedup reaches a stable state. This is due to communication-bound parallelism, where scalability is limited by communication overhead. As the number of processors increases, the overhead of data exchange and coordination between processors becomes a bottleneck. While the speedup may initially increase, the increasing communication costs can lead to reaching a stable state or even a decrease in speedup beyond a certain number of processors.

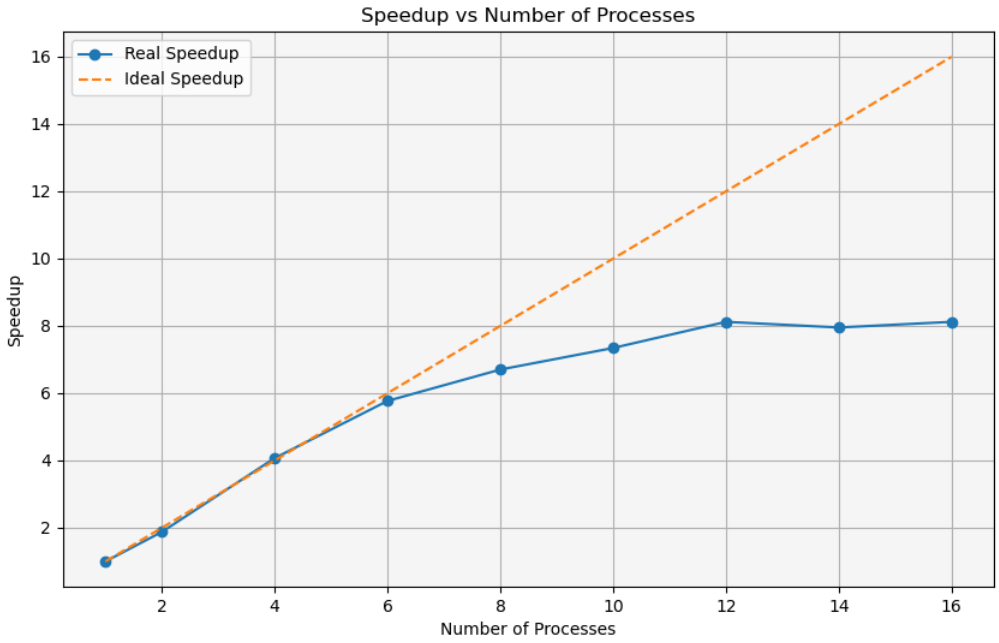

Figure 1: Strong scalability experiment

3.2 Weak scalability experiment

For the weak scalability experiment, we conducted two sets of experiments. The first group fixed the number of stencil applications to 10, with each process processing 10,000,000 samples, and computed the elapsed time and efficiency (E) for process numbers 1, 2, 4, and 8. The second group fixed the number of stencil applications to 10 and processed 5,000,000 samples per process and calculated the results when the number of processes was 2, 4, 8 and 16. The results are presented in Table.2 and Figure.2. Each configuration was run five times, and the averages were computed.
The efficiency calculation formula is as follows, where p denotes the number of processes, T_s is the serial execution time and T_p is the parallel execution time.

$$\mathrm{E} = \frac{T_s}{p * T_p}$$

From the results shown in the chart, we can observe fluctuations in the total execution time as the number of processors and the overall problem size increase. However, ideally, with an increase in the number of processors, the total execution time should not increase, as each processor handles a fixed amount of sample data. The observed fluctuations in execution time may be attributed to the increased consumption of parallel computing resources due to the higher number of processors.
Comparing the efficiency of two different problem sizes, typically, the efficiency of handling larger problem sizes (e.g., 8000000) tends to be lower than that of handling smaller problem sizes (e.g., 2000000). This is because larger problem sizes can introduce more

Number of Processes	Problem Size	Execution Time (seconds)	Efficiency
1	1000000	0.0534	1.00
2	2000000	0.0636	1.08
4	4000000	0.0611	1.04
8	8000000	0.0728	0.95
2	1000000	0.0323	0.83
4	2000000	0.0253	1.36
8	4000000	0.0345	0.92
16	8000000	0.0660	0.52

Table 2: Weak scalability experiment

communication overhead and competition among processors, thereby reducing parallel efficiency. Observing the efficiency at different numbers of processes and problem sizes, we find that the efficiency remains generally stable when each process handles 10,000,000 samples, whereas the efficiency is not consistently stable when each process handles 5,000,000 samples. Particularly, with 16 threads processing an input data size of 8,000,000, this may suggest that the system encounters bottlenecks or performance limitations with a higher number of processors.
Overall, the program exhibits good weak scalability because it maintains stable efficiency or only experiences slight efficiency degradation when increasing the number of processors and the overall problem size.

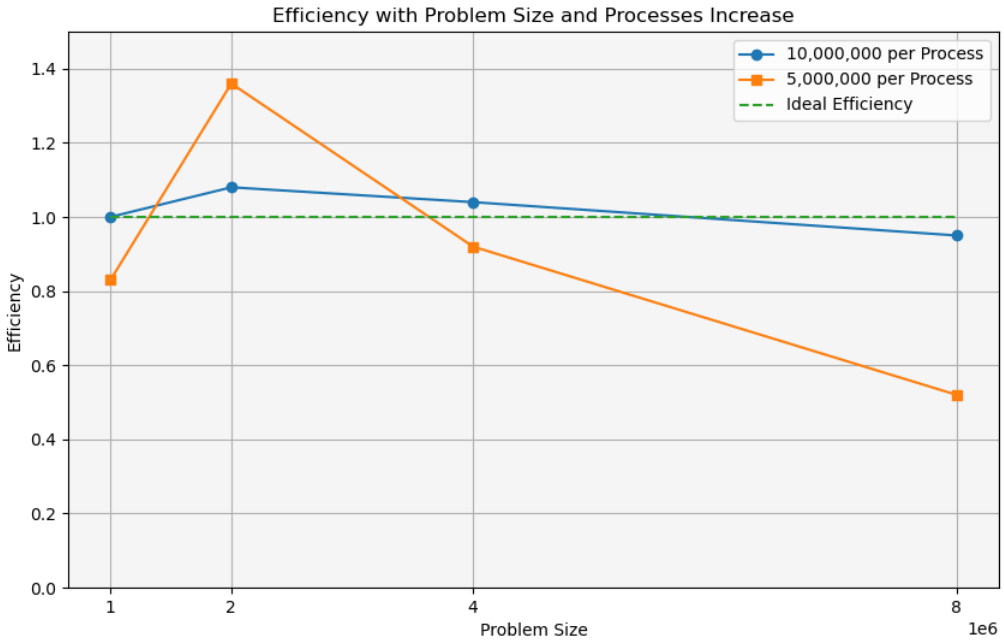

Figure 2: Weak scalability experiment

4 Discussion

In our experiments, we observe that strong scalability is maintained up to a certain point as the number of processing processes increases while the problem size remains constant. This behaviour is in line with expectations and demonstrates the potential of parallel computing to accelerate computation by exploiting additional processing resources. However, as the number of processors exceeds a certain number, the return on acceleration diminishes, highlighting the challenges posed by communication overheads in distributed computing environments. This observation highlights the importance of optimising communication patterns and workload allocation to sustain large-scale performance gains.

In addition to the scalability experiments, we verified the speedup ratios for different dataset sizes with the same number of cores, and found unexpected negative speedup ratios for smaller problem sizes, which emphasises the importance of the overheads associated with process synchronisation and data exchange. This finding highlights the complexity of balancing computation and communication in parallel programs, especially for smaller tasks where coordination overheads may outweigh computational efficiency.

Overall, our MPI parallelisation strategy is very effective in managing communication overheads and optimising performance. By leveraging non-blocking communication techniques and separating communication operations from computation, we alleviate latency issues and improve overall efficiency. However, our experiments also highlight the need to continuously refine the parallelisation strategy to achieve higher performance.