INTRODUCTION TO SCALABLE SYSTEM - ASSIGNMENT-3

Bhikshapathi Kaniki

Sr no :06-18-01-10-51-24-1-24489

Problem Statement

Write an OpenMP parallel program for parallelizing the prefix sum problem. Input array A and output array B will be shared by the OpenMP threads. Different threads will deal with different disjoint elements of B.

Tasks

1. Experiment with array sizes of 10,000, 20,000, and 30,000 integer elements.
2. For each array size, run your code with 2, 4, 8, 16, 32, and 64 threads.
3. For a given array size and a given number of threads, execute your code 5 times and compute the average execution time across these 5 runs.
4. Compute speedups with respect to sequential execution for all configurations.
5. Write a report describing:
    - **Methodology:** How the experiments were conducted.
    - **Experiments:** Details of the experiments performed.
    - **Results:** Include execution times and speedups as both numbers and graph plots.
    - **Observations:** Discuss trends, insights, and any anomalies observed in the results.

Methodology

Input Data Generation

- Input data is read from a file provided as the first command-line argument. The file contains the size of the array n followed by n integer elements.
- The size of the input array and its elements are dynamically allocated in memory for processing.

Sequential Prefix Sum Calculation (Baseline)

- The program includes logic to execute the prefix sum in parallel. A separate sequential version of the prefix sum calculation should be implemented to serve as the baseline for speedup calculations.
- The sequential execution time will be obtained by running the sequential version 5 times and recording the average execution time.

Parallel Prefix Sum Calculation Using OpenMP

The prefix sum computation is parallelized using OpenMP with the following steps:

1. **Local Prefix Sum:** Each thread calculates the prefix sum for its assigned chunk of the array. The chunk size is determined by dividing the input size n by the number of threads.
2. **Offset Calculation:** The last element of each thread's local prefix sum is stored in a shared `offsets` array. This value represents the cumulative sum for each thread's chunk.
3. **Offset Adjustment:** Using a critical section and a barrier, the offsets are computed sequentially by thread 0 to propagate cumulative sums across all threads.
4. **Adjust Local Sums:** After the offsets are computed, threads adjust their local prefix sums using the offset from previous threads to ensure correctness.

Experimental Setup

- The program will be executed for different array sizes of $10M$, $20M$, and $30M$ elements.
- For each array size, the program will be run with thread counts of $2, 4, 8, 16, 32$, and 64.
- Each configuration (array size and thread count) will be executed 5 times, and the average execution time will be recorded.

Performance Evaluation

- The following metrics will be computed:
  - **Execution Time:** Average time for parallel execution under different configurations.
  - **Speedup:** Computed as the ratio of sequential execution time to parallel execution time for each configuration.
- Results will be presented in tabular and graphical forms, showing execution times and speedups.

Experimental Setup

1. **Language:** C++
2. **Compute Node:** `bhikshapathi@10.24.36.80`
3. **Environment:** Experiments are performed on the cluster master node.
4. **Commands to Run OpenMP Files:**

```
g++ -fopenmp file_name.cpp -o file_name
./file_name input_file_txt no_of threads
```

Results

Threads	Execution Time (s)	Speedup
1	0.447331	1.000
2	0.361413	1.238
4	0.203156	2.202
8	0.093617	4.778
16	0.079522	5.625
32	0.088778	5.039
64	0.063693	7.023

Table 1: Execution Time and Speedup for 100M

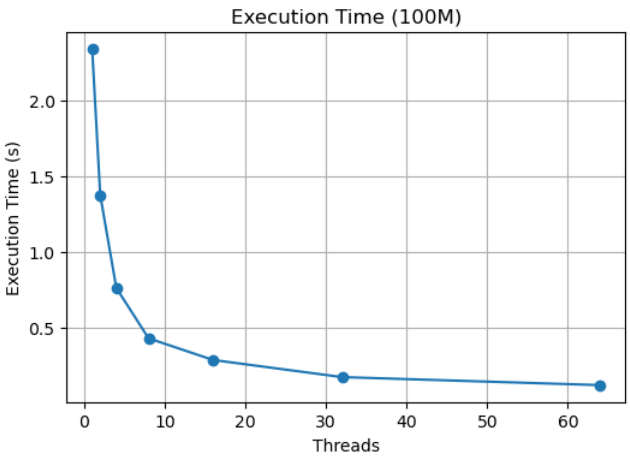

Figure 1: Execution Time for 100M

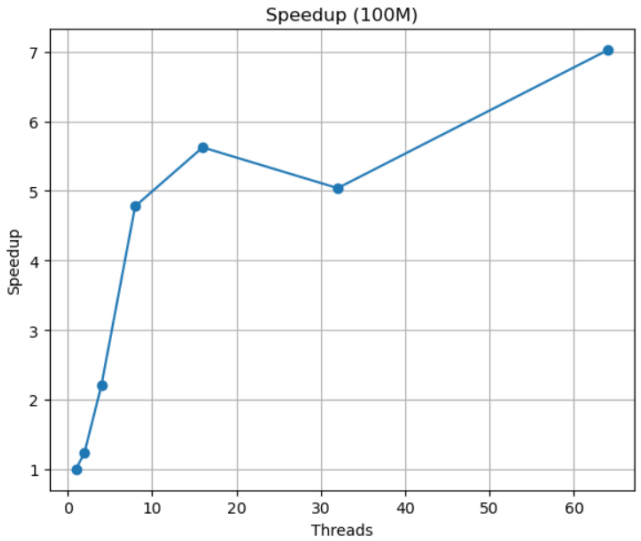

Figure 2: Speedup for 100M

Threads	Execution Time (s)	Speedup
1	1.181168	1.000
2	0.883847	1.337
4	0.550013	2.148
8	0.230277	5.128
16	0.188346	6.271
32	0.136879	8.630
64	0.128687	9.182

Table 2: Execution Time and Speedup for 200M

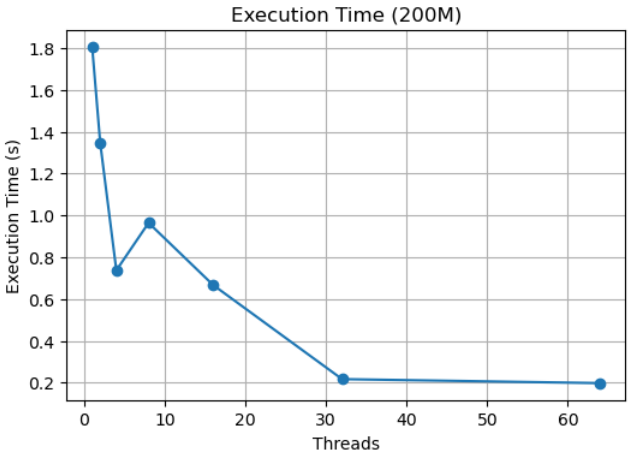

Figure 3: Execution Time for 200M

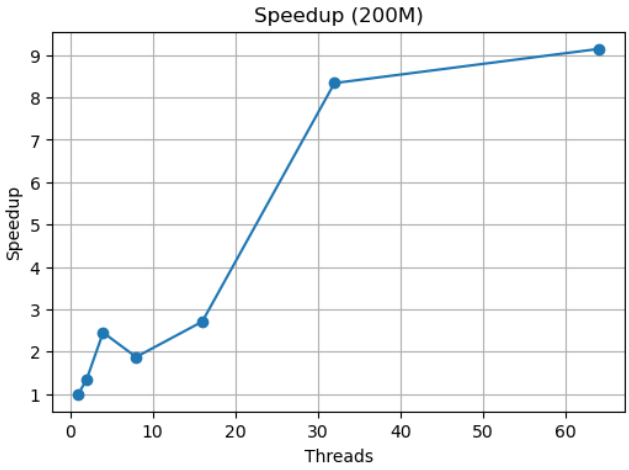

Figure 4: Speedup for 200M

Threads	Execution Time (s)	Speedup
1	1.806991	1.000
2	1.345480	1.343
4	0.736886	2.452
8	0.964041	1.874
16	0.666916	2.709
32	0.216739	8.337
64	0.197610	9.144

Table 3: Execution Time and Speedup for 300M

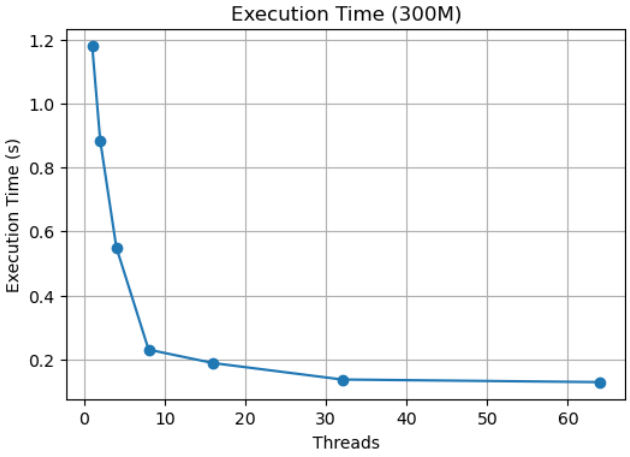

Figure 5: Execution Time for 300M

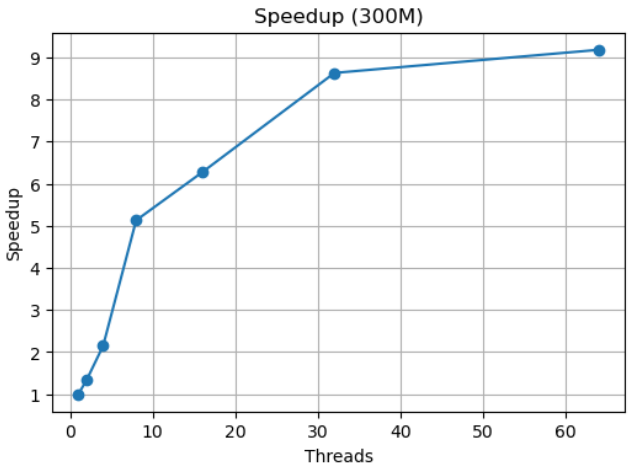

Figure 6: Speedup for 300M

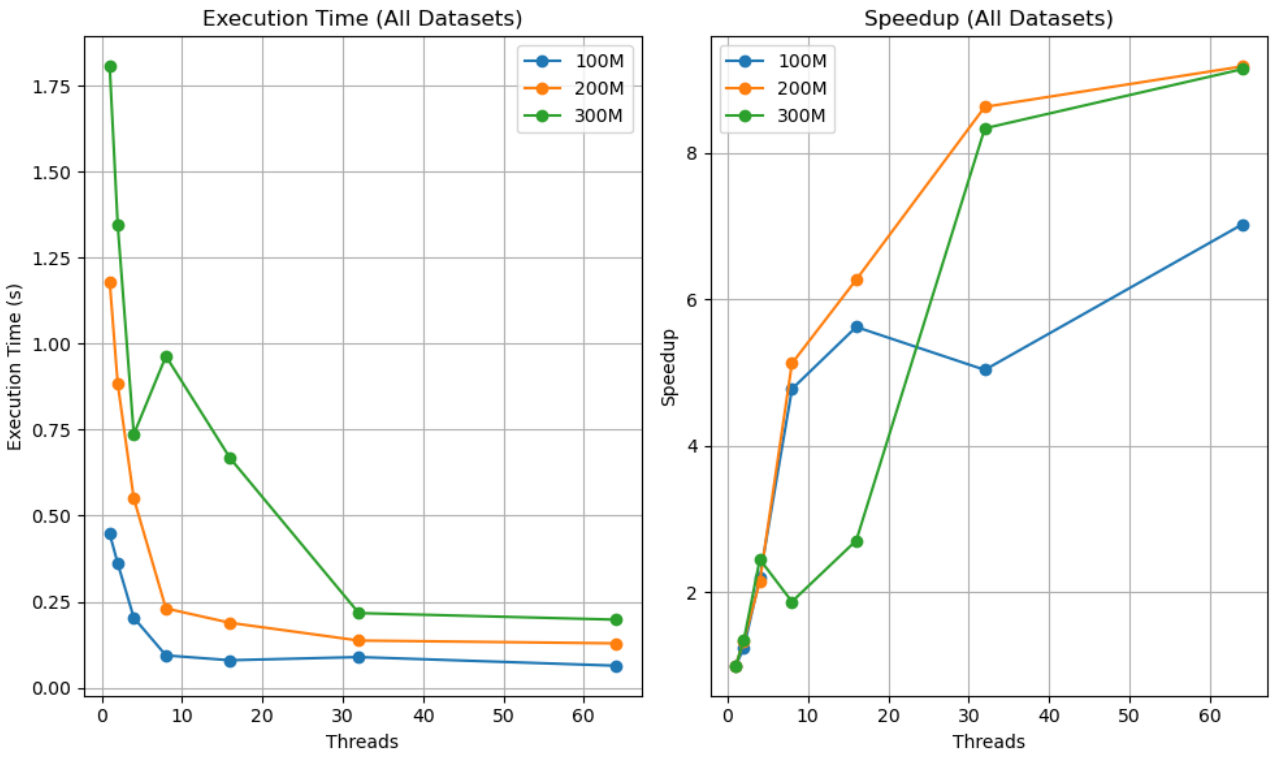

Figure 7: comparison of execution times and speed-ups

Observations

1. **Execution Time:**
   - Execution time decreases with increasing threads across all datasets, demonstrating the benefits of parallelization. The reduction slows after 16 threads due to diminishing returns.
2. **Speedup:**
   - Speedup increases with thread count, reaching its peak at 64 threads for larger datasets (200M and 300M), while 100M achieves a maximum speedup of 7.023 at 64 threads.
3. **Optimal Threads:**
   - For smaller datasets (100M), 8-16 threads offer the best performance. For larger datasets, 32-64 threads provide the best results.
4. **Bottlenecks:**
   - Some overhead is observed at higher thread counts, particularly with smaller datasets, where performance improvements decrease due to thread management and synchronization overhead.

In summary, larger datasets scale better with more threads, while smaller datasets experience diminishing returns after a certain point.

Problem Statement

Refer to the MPI class lecture slides that have an MPI program involving two processes that tries to find a particular element in an array distributed across the two processes. Now, write an MPI program that works with any number of processes.

For this program, generate a random integer array of size 1,000,000 (i.e., 1 million) with random integer elements between 1 and 5,000,000 (5 million). You can either generate this array in process 0 and distribute it equally across all the processes or generate this array to a file and make the processes read from different portions of the file (e.g., using `fseek`).

An integer element is given as input to the program and the processes start searching for this element in their local sub-arrays. As soon as a process finds the element, it informs the other processes and all processes will stop searching. Process 0 should print the global index of the overall array in which the element was found.

Tasks

1. Once a process finds the element, it should inform the other processes to stop the search. Process 0 should print the global index of the element.
2. Experiment with different numbers of processes, including 1 (sequential), 8, 16, 32, and 64 processes.
3. For each number of processes, execute the program with 20 different random input integers, measure the time taken for each search, and compute the average time.
4. Plot the execution times and speedups for different numbers of processes.
5. Ensure that different processes run on different cores.

- **Data Size**: The array size is set to 1,000,000 elements (`n = 1000000`).
- **Element Range**: Each element of the array is a random integer between 1 and 5,000,000.

Methodology

Data Generation

The program begins by creating a large random integer array, but only the root process (process with rank 0) is responsible for generating this data.

- **Data Size**: The array size is set to 1,000,000 elements (`n = 1000000`).
- **Element Range**: Each element of the array is a random integer between 1 and 5,000,000.

Steps:

1. In process 0, the array `data` is populated with random values.
2. The other processes will later receive chunks of this array.

Distribution of Data Among Processes

The array is distributed equally among all processes using the `MPI_Scatter` function. This allows each process to work on a portion of the array, effectively dividing the computational load.

- **Local Array**: Each process is assigned a portion of the data (i.e., `local_data`), where the size of each portion is `n / size`, with `size` being the total number of processes.
- `MPI_Scatter`: The function sends portions of the array from the root process to all other processes. Each process receives its corresponding part of the array.

Steps:

1. The root process splits the array into `size` equal-sized chunks.
2. Each process, including the root, receives a chunk of the array (`local_data`).

Search for the Target Element

The core functionality of the program is to search for a given element within each process's local data.

- **Command-Line Input**: The number to search for is passed as a command-line argument to the program.
- **Search Mechanism**: Each process independently searches through its assigned portion of the array for the target number.

Steps:

1. The root process (rank 0) checks if the correct number of arguments is provided.
2. Each process (including rank 0) checks if the target number is in its local array.
3. If the target number is found, the process records its index in the global array and sends a stop signal to all other processes to stop their searches.

Early Termination and Synchronization

To avoid redundant work, once a process finds the target element, it sends a stop signal to all other processes.

- **Stop Signal**: The stop signal ensures that no other process continues searching once the element is found. This is implemented using `MPI_Send` and `MPI_Irecv` (non-blocking receive).

Steps:

1. If a process finds the target number, it broadcasts the fact that it has found the element by sending a message (`found` = 1) to all other processes.
2. All processes check for the stop signal using `MPI_Test`, and stop their search if they receive it.

Gathering Results

Once the search is complete, the processes need to determine the **global index** of the target element (if found) across all the processes.

- `MPI_Reduce`: This operation gathers the global index from all processes and reduces it to the minimum index (i.e., the first occurrence of the target number).

Steps:

1. Each process computes its local index where the target is found (if found).
2. The global minimum index is determined using `MPI_Reduce`, ensuring that the first occurrence is reported.

Execution Time Measurement

The program measures the time taken for the search operation to complete across all processes.

- `MPI_Wtime`: This function provides high-resolution wall clock time, which is used to measure the execution time from the start of the data distribution to the end of the result gathering.

Steps:

1. The time is recorded before the search begins (right after data distribution).
2. The time is again recorded after all processes finish their searches and the result is gathered.
3. The execution time is calculated by subtracting the start time from the end time.

Final Output

After the global index is determined, process 0 (rank 0) prints the result:

- **Element Found**: If the target number is found, its global index is printed.
- **Element Not Found**: If the element is not found, a message is printed indicating the absence of the element in the array.
- **Execution Time**: The program also prints the total execution time for the parallel search.

Key Features of the Methodology:

1. **Parallel Search**: The array is split across processes, allowing each process to perform the search independently, speeding up the search operation for large arrays.
2. **Early Termination**: The stop signal ensures that only the process that finds the element needs to do further work, and all other processes stop searching early.
3. **Synchronization**: `MPI_Barrier` ensures that all processes finish their search before gathering the results. The `MPI_Reduce` operation ensures that the smallest index is collected.
4. **Performance Measurement**: The execution time is tracked to assess how well the parallelization performs with varying numbers of processes.

Experimental Setup

1. Language: C++
2. Compute node: `bhikshapathi@10.24.36.80` / Node 1
3. Experiments are done on the KIAC cluster master node.
4. Executed through queue jobs.
5. Array size: 1,000,000 elements.
6. Random data generation: Seeded random generator on the root process.
7. Data distribution: Performed using `MPI_Scatter`.
8. Search method: Each process searches for a target number passed as a command-line argument.
9. Stop signal handling: Implemented using `MPI_Irecv` and `MPI_Send`.
10. Result aggregation: Performed using `MPI_Reduce` at the root process.

Result

Processes	Execution Time (ms)	Speedup
1	78.036696	1.00
8	10.249516	7.61
16	7.1264912	10.95
32	13.192457	5.92
64	16.767392	4.66

Table 4: Execution Time and Speedup for Different Processes

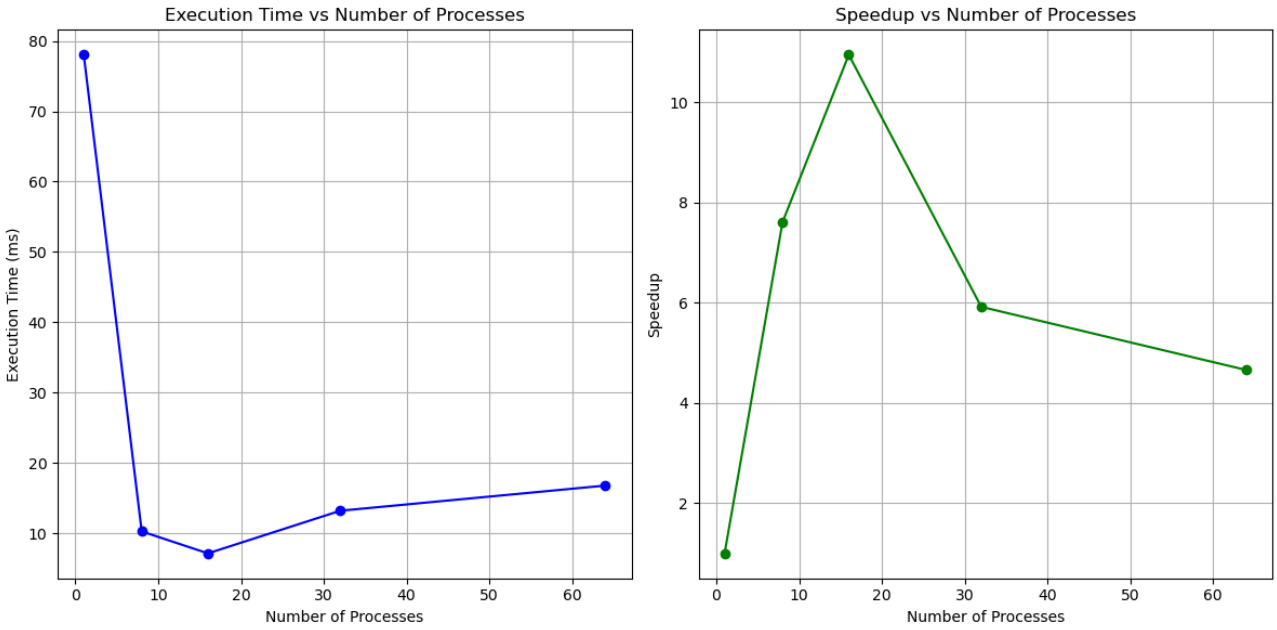

Figure 8: execution times and speed-ups

Observations

- **Execution Time:**
  - Execution time decreases significantly as the number of processes increases, demonstrating the benefits of parallelism.
  - The fastest execution time is observed with 8 processes, but it increases again at 64 processes, indicating diminishing returns with more processes.
- **Speedup:**
  - Speedup increases with the number of processes, peaking at 16 processes with a speedup of 10.95.
  - Beyond 16 processes, speedup starts to decrease, especially at 64 processes, due to communication overhead and thread synchronization.
- **Optimal Process Count:**
  - The optimal number of processes is 16, where the highest speedup is achieved. Further increases in the number of processes result in diminishing returns.