DS221: Introduction to Scalable Systems
Assignment 2

November 21, 2024

Sandeep Kumar
24022

---

1. Given an integer array A, compute the prefix sum array B such that $B[j] = \sum_{i=0}^{j} A[i]$.

   OpenMP is used to split the computation into multiple threads, with each thread handling disjoint sections of the array B.

   Arrays of sizes 10,000, 20,000, and 30,000 were used. Thread counts of 2, 4, 8, 16, 32, and 64 were tested. Each configuration was run 5 times, and the average time was calculated. Sequential execution was used as the baseline.

   Performance was measured using execution time and speedup, calculated as the ratio of sequential execution time to parallel execution time.

   Solution:

Methodology

Two implementations of the prefix sum algorithm were evaluated:

- **Sequential Prefix Sum:** Computed iteratively in a single thread.
- **Parallel Prefix Sum:** Divided into chunks processed independently by multiple threads using OpenMP.

Sequential Prefix Sum

The sequential prefix sum computes each element in the output array $B[i]$ as:

$$B[i] = B[i-1] + A[i]$$

This implementation serves as the baseline for performance comparison and also helps check the OpenMP results.

Parallel Prefix Sum Using OpenMP

To parallelize the prefix sum, the operation is split into multiple phases:

1. **Local Computation:** Each thread computes the prefix sum for its assigned chunk of the array independently.
2. **Chunk Summation:** A separate step accumulates the sums of each thread's chunks to calculate the offsets required to adjust subsequent chunks. This step is performed by a single thread.
3. **Offset Adjustment:** Each thread updates its chunk by adding the appropriate offset from the preceding chunks.

Experimental Setup

- **Array Sizes:** 10^6, $2 \cdot 10^6$, $3 \cdot 10^6$, $4 \cdot 10^6$,$5 \cdot 10^6$,$6 \cdot 10^6$, $7 \cdot 10^6$, $8 \cdot 10^6$, $9 \cdot 10^6$ and $1 \cdot 10^7$.(These array sizes were taken as the once provided in the question were providing good speedups.)
- **Thread Counts:** $1, 2, 4, 8, 16, 32$, and 64.
- **Repetitions:** Each experiment was repeated 5 times, and the average execution time was recorded.

Speedup was computed as:

$$\text{Speedup} = \frac{\text{Sequential Execution Time}}{\text{Parallel Execution Time}}$$

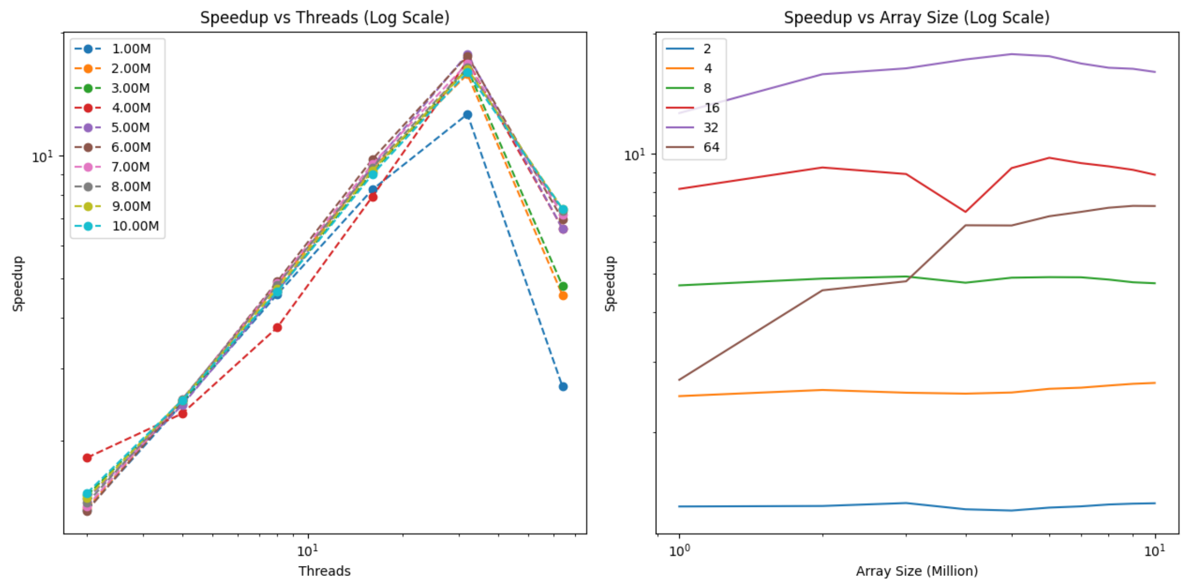

Figure 1: Speedup vs. number of threads for the prefix sum problem.

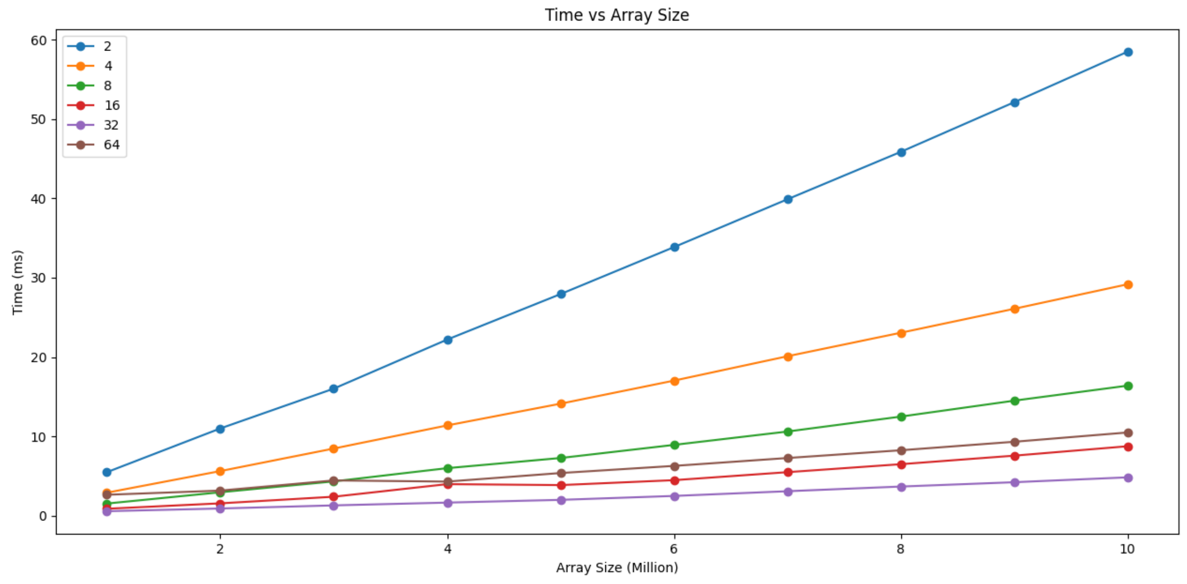

Figure 2: Execution time vs. array size for the prefix sum problem.

Conclusion for Problem 1:

From the plot of speedup versus threads, we see that as the number of threads increases, the speedup initially grows because the code is highly parallelizable. However, at a fixed array size, the speedup does not grow linearly, and the rate of improvement slows down as more threads are added. This happens because the program still has some sequential work that cannot be parallelized. Adding more threads reduces the time spent on the parallel part but does not affect the sequential part, which limits the overall speedup. This behavior follows Amdahl's Law, which states that the speedup is limited by the portion of the program that is sequential.

When the problem size increases, the parallel portion of the program also grows, making the sequential part less significant. As a result, the speedup improves with larger problem sizes because the additional threads have more work to do. This follows Gustafson's Law, which shows that increasing the problem size helps utilize more threads effectively.

At 64 threads, the speedup is significantly lower. This is because the system on which the program is running has 32 physical cores, with one thread per core and no hyper-threading. When running the program with more threads than the available cores (e.g., 64 threads), the operating system must manage multiple threads on the same core. This leads to context switching overhead, where the CPU repeatedly switches between threads, resulting in inefficiencies and reduced performance.

2. Write an MPI program to search for a specific integer in a large array, accommodating any number of processes. Generate a random integer array of size 1,000,000 with

values ranging from 1 to 5,000,000. You can either create this array in process 0 and distribute equal parts to all processes or write the array to a file and let each process read its assigned segment (e.g., using fseek). Given a target integer, each process will search for the number in its local segment. When a process finds the target, it will notify all other processes to stop searching. Process 0 will then output the global index of the target in the array.

Execute the program with various numbers of processes, including 1 (sequential execution), 8, 16, 32, and 64. For each configuration, test the program with 20 different random target integers, recording the time taken for each search. Calculate and report the average search time for each configuration. Additionally, generate a plot to visualize execution times and compute the speedups achieved with increasing process counts. Ensure that each process runs on a separate core to maximize efficiency.

Solution:

Methodology

Data Generation

The root process initializes an array of size 10^6 (1 million elements) filled with random integers in the range $[1, 5,000,000]$. A fixed seed (`srand(0)`) ensures reproducibility in array initialization across multiple runs. The array is evenly divided into chunks of equal size, and each chunk is distributed to a different process using `MPI_Scatter`.

Parallel Search Implementation

Broadcast Target Value

For each trial, the root process generates a random target value in the range $[1, 5,000,000]$. This target is broadcast to all processes using `MPI_Bcast`.

Local Search Execution

Each process performs a search for the target value within its local chunk of the array. If the target is found, the local index is converted to a global index using the formula:

$$\text{Global Index} = \text{Rank} \times \text{Local Chunk Size} + \text{Local Index}.$$

Result Aggregation

Each process sends its local result (global index or -1 if not found) to the root process using `MPI_Gather`. The root process determines the first occurrence of the target by examining the global indices.

Timing Measurement

Timing for the parallel execution starts just before broadcasting the target and stops immediately after the result aggregation. `MPI_Wtime` is used for precise timing measurements. Execution times for all trials are accumulated to compute the average parallel execution time.

Target Selection and Repetitions

The search is repeated for 20 trials, each with a new random target value, to ensure statistical reliability and variability in timing.

Speedup Calculation

To measure the efficiency of the parallel implementation, the speedup can be calculated as:

$$\text{Speedup} = \frac{\text{Execution Time for 1 Process}}{\text{Average Execution Time for } n \text{ Processes}}.$$

This involves running the same search program with $n = 1$ process and recording the execution time for comparison.

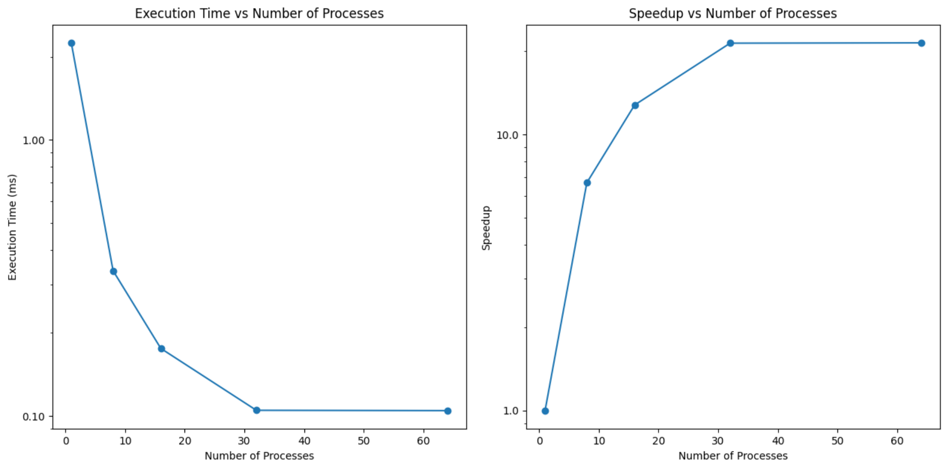

Figure 3: Speedup and execution time vs. number of processes for the MPI-based search problem.

Conclusion for Problem 2:

From the results of the MPI-based parallel search program, it is evident that increasing the number of processes reduces the search time, leading to significant performance improvement as more processes are utilized. The speedup demonstrates that distributing the workload across multiple processes effectively enhances efficiency.

However, the speedup is not linear with the number of processes. This is due to the increasing overhead associated with inter-process communication (`MPI_Bcast` and `MPI_Gather`) and synchronization. As the number of processes grows, especially beyond the number of available physical cores, resource contention and context switching start to degrade performance, leading to diminishing returns.

For lower process counts, the program achieves a good speedup because the workload is large enough to offset the communication overhead. However, as the number of processes exceeds the core count, the overhead from communication and synchronization outweighs the benefits of parallelization.

The results demonstrate that while parallel computing is effective for large-scale search tasks, the scalability is constrained by hardware resources and communication overhead. Balancing the number of processes with the available computational resources is critical to achieving optimal performance.

Appendix

A Performance Results for Question 1: Parallel Prefix Sum using OpenMP

Table 1: Performance Results for Parallel Prefix Sum using OpenMP

Array Size	Threads	Parallel Time (s)	Sequential Time (s)	Speedup
1000000	2	0.0054847	0.00713622	1.30111
	4	0.00289779	0.00713622	2.46264
	8	0.0015276	0.00713622	4.67151
	16	0.000874509	0.00713622	8.16025
	32	0.000564063	0.00713622	12.6515
	64	0.00263675	0.00713622	2.70644
2000000	2	0.0109952	0.0143472	1.30486
	4	0.00562197	0.0143472	2.55198
	8	0.002953	0.0143472	4.85851
	16	0.00155342	0.0143472	9.23585

Array Size	Threads	Parallel Time (s)	Sequential Time (s)	Speedup
	32	0.000906036	0.0143472	15.8351
	64	0.0031601	0.0143472	4.5401
3000000	2	0.0160107	0.0212507	1.32728
	4	0.00846134	0.0212507	2.5115
	8	0.00432099	0.0212507	4.91801
	16	0.00238926	0.0212507	8.89425
	32	0.00129694	0.0212507	16.3852
	64	0.00444247	0.0212507	4.78353
4000000	2	0.0221923	0.0284096	1.28015
	4	0.011373	0.0284096	2.49799
	8	0.00598818	0.0284096	4.74428
	16	0.00397682	0.0284096	7.1438
	32	0.00164722	0.0284096	17.247
	64	0.00429661	0.0284096	6.6121
5000000	2	0.0279294	0.0354938	1.27084
	4	0.0141154	0.0354938	2.51454
	8	0.00726785	0.0354938	4.88367
	16	0.00385624	0.0354938	9.20425
	32	0.00199527	0.0354938	17.789
	64	0.00537412	0.0354938	6.60458
6000000	2	0.0338457	0.0437151	1.29263
	4	0.017021	0.0437151	2.56894
	8	0.00892122	0.0437151	4.89997
	16	0.0044758	0.0437151	9.76856
	32	0.0024875	0.0437151	17.5732
	64	0.00627442	0.0437151	6.96724
7000000	2	0.0398742	0.0519323	1.30208
	4	0.020091	0.0519323	2.58584
	8	0.0106042	0.0519323	4.89555
	16	0.0054823	0.0519323	9.47163
	32	0.0030821	0.0519323	16.8477
	64	0.0072651	0.0519323	7.14692
8000000	2	0.0458523	0.0603475	1.3162
	4	0.0230547	0.0603475	2.61787
	8	0.0124878	0.0603475	4.83192
	16	0.0064895	0.0603475	9.29712
	32	0.003671	0.0603475	16.4413
	64	0.0082452	0.0603475	7.31874
9000000	2	0.0521094	0.0689118	1.32263
	4	0.0260732	0.0689118	2.64431
	8	0.0144951	0.0689118	4.75647

Array Size	Threads	Parallel Time (s)	Sequential Time (s)	Speedup
	16	0.0075632	0.0689118	9.11032
	32	0.0042168	0.0689118	16.3417
	64	0.0093142	0.0689118	7.39862
10000000	2	0.0584742	0.0775375	1.32543
	4	0.0291641	0.0775375	2.6584
	8	0.0163876	0.0775375	4.7297
	16	0.0087518	0.0775375	8.85918
	32	0.0048329	0.0775375	16.0503
	64	0.0104871	0.0775375	7.39226