DS221: Introduction to Scalable Systems

Naman Pesricha
namanp@iisc.ac.in
SR - 24115

Parallel Programming

Q1 OpenMP Prefix Sum (7.5 points)

Prefix sum of a set of elements contained in an integer array, A, produces an integer array, B, such that B[j] = sum (from i=0 to i=j) A[i] Example, if A[] = {5, 7, 3, 8, 2}, B will be B[] = {5, 12, 15, 23, 25}

Write a OpenMP parallel program for parallelizing the above prefix 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.

Experiment with array sizes of 10000, 20000 and 30000 integer elements. For each array, run your code with 2, 4, 8, 16, 32 and 64 threads. For a given array and given number of threads, run your code 5 times and obtain the average time across these 5 runs. Obtain speedups with respect to sequential execution. Write a report describing the methodology, experiments, results and observations. For results, give execution times, speedups as both numbers and grassph plots.

---

1 Methodology

The implementation of `prefixSumOMP` computes the prefix sum of an input vector A and stores the result in vector B using multiple threads with OpenMP. The approach divides the input array A into subarrays for each thread, calculates local prefix sums, and then adjusts these with offsets derived from the global prefix sums of previous threads. Below is the step-by-step methodology:

1. **Input Partitioning:**
   The input vector A of size n is divided into t equal parts (for t threads). Each thread t_i is responsible for calculating the prefix sum of its assigned subarray A_i. Let:
   - A_0, A_1, A_2, A_3 be the subarrays handled by threads t_0, t_1, t_2, t_3, respectively.
2. **Local Prefix Sum Calculation:**
   Each thread calculates the prefix sum of its assigned subarray:
   - Thread t_i starts at index start $= i \times \frac{n}{t}$ and ends at end $= (i+1) \times \frac{n}{t}$.
   - The first element of B_i is set as $B[\text{start}] = A[\text{start}]$.
   - For subsequent elements:
     $$B[j] = A[j] + B[j-1], \quad \forall j \in [\text{start}+1, \text{end}).$$

   Example:
   - t_0 computes the local prefix sum for A_0 and stores it in B_0.
   - Similarly, t_1, t_2, t_3 compute local prefix sums for A_1, A_2, A_3, respectively.
3. **Compute Offsets:**
   After completing local prefix sums:
   - Each thread calculates the total sum of its subarray:
     $$\text{offsets}[i] = B[\text{end}-1], \quad \text{for thread } t_i.$$
   - Thread t_0 sets its offset to 0:
     $$\text{offsets}[0] = 0.$$
4. **Global Offset Adjustment:**
   Using the offsets:
   - Thread t_0 requires no adjustment since offsets[0] = 0.
   - Thread t_1 adds the sum of A_0 to all elements of B_1.
   - Thread t_2 adds the cumulative sum of A_0 and A_1 to all elements of B_2, and so on.

Offsets are computed globally using a single-thread operation:

$$\text{offsets}[i] = \text{offsets}[i] + \text{offsets}[i-1], \quad \forall i > 0.$$

5. **Adjust Local Prefix Sums (SIMD):**
   Each thread adjusts its local prefix sum using its respective offset:
   - $B[j] = B[j] + \text{offsets}[i], \quad \forall j \in [\text{start}, \text{end}).$

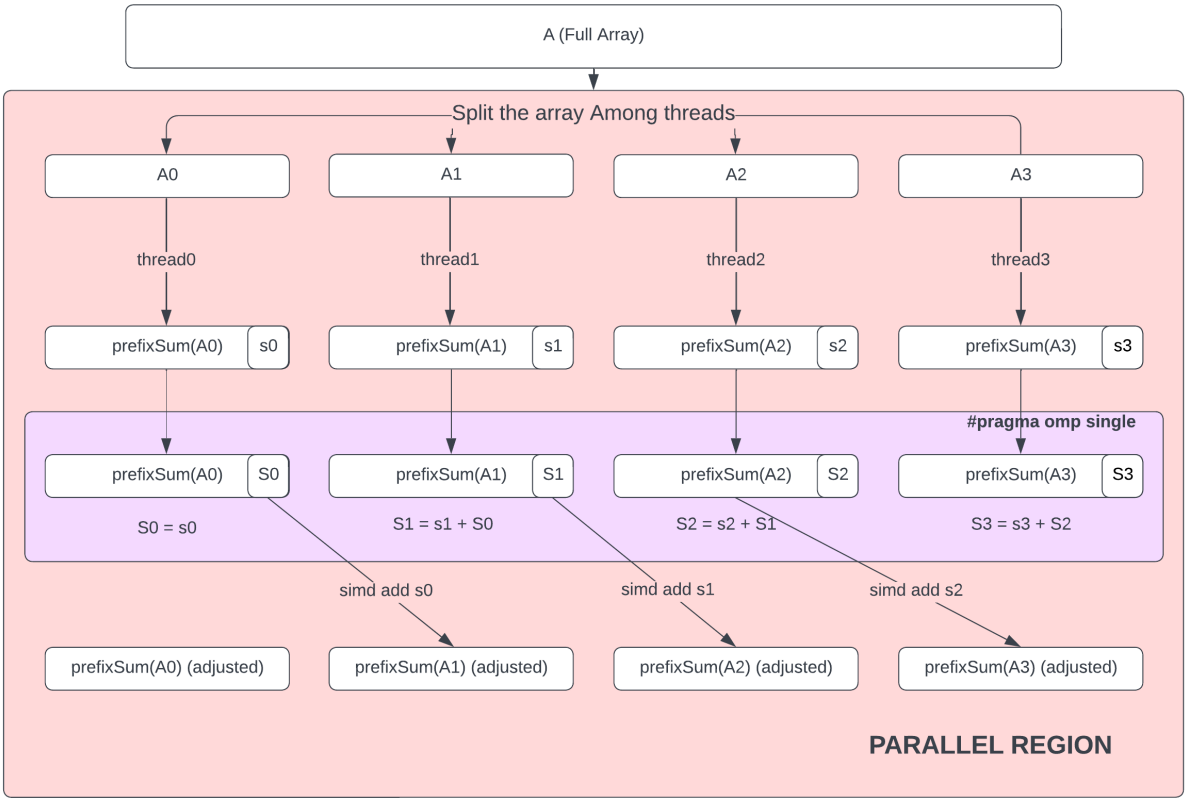

Figure 1: Methodology for parallel prefix sum.

1.1 Example with 4 thread:

Input Array A (size 16):

$$A = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]$$

Step 1: Partition A into Subarrays:

$$A_0 = [1, 2, 3, 4] \text{ for } t_0$$

$$A_1 = [5, 6, 7, 8] \text{ for } t_1$$

$$A_2 = [9, 10, 11, 12] \text{ for } t_2$$

$$A_3 = [13, 14, 15, 16] \text{ for } t_3$$

Step 2: Local Prefix Sums

$$B_0 = [1, 3, 6, 10]$$

$$B_1 = [5, 11, 18, 26]$$

$$B_2 = [9, 19, 30, 42]$$

$$B_3 = [13, 27, 42, 58]$$

Step 3: Calculate Offsets

$$\text{offsets} = [0, 10, 26, 42].$$

Step 4: Global Offset Adjustment

$$\text{offsets} = [0, 10, 36, 78].$$

Step 5: Adjust Local Prefix Sums

Adjust B_1: $[15, 21, 28, 36]$

Adjust B_2: $[45, 55, 66, 78]$

Adjust B_3: $[91, 105, 120, 136]$

Final Result B:

$$B = [1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78, 91, 105, 120, 136].$$

1.2 1.2 Time Complexity

The time complexity for the program is:

$$T(n, t) = O\left(\frac{2n}{t} + t\right) + parallelization\ overheads$$

Step 1 and 2 : O(n/t).

Step 3 and 4 : O(t).

Step 5 : O(n/t)

---

2 Experiments

1.3 Experimental Parameters

The code was ran for different values of N (#elements) and t (#threads). Below are the values of N and t.

$$T = [2, 4, 8, 16, 32, 64];$$

$$N = [1000000, 2000000, 3000000, 4000000, 5000000, 6000000, 7000000, 8000000, 9000000, 10000000]$$

1.4 Timing and Averaging

The time is calculated in milliseconds and the calculated time does not include creation of the array. To make sure the data is accurate, times were averaged over 30 runs. The results are discussed in the next section. We have used `std::chrono` for the same.
Note: The given values in the question (N = 10000, 20000 and 30000) were not used because :

1. They were too less in number very less information could be extracted from the plots.
2. The values were too small to notice significant speedup trends.

3 Results

The execution times and speed ups in numbers are mentioned in Appendix 1.

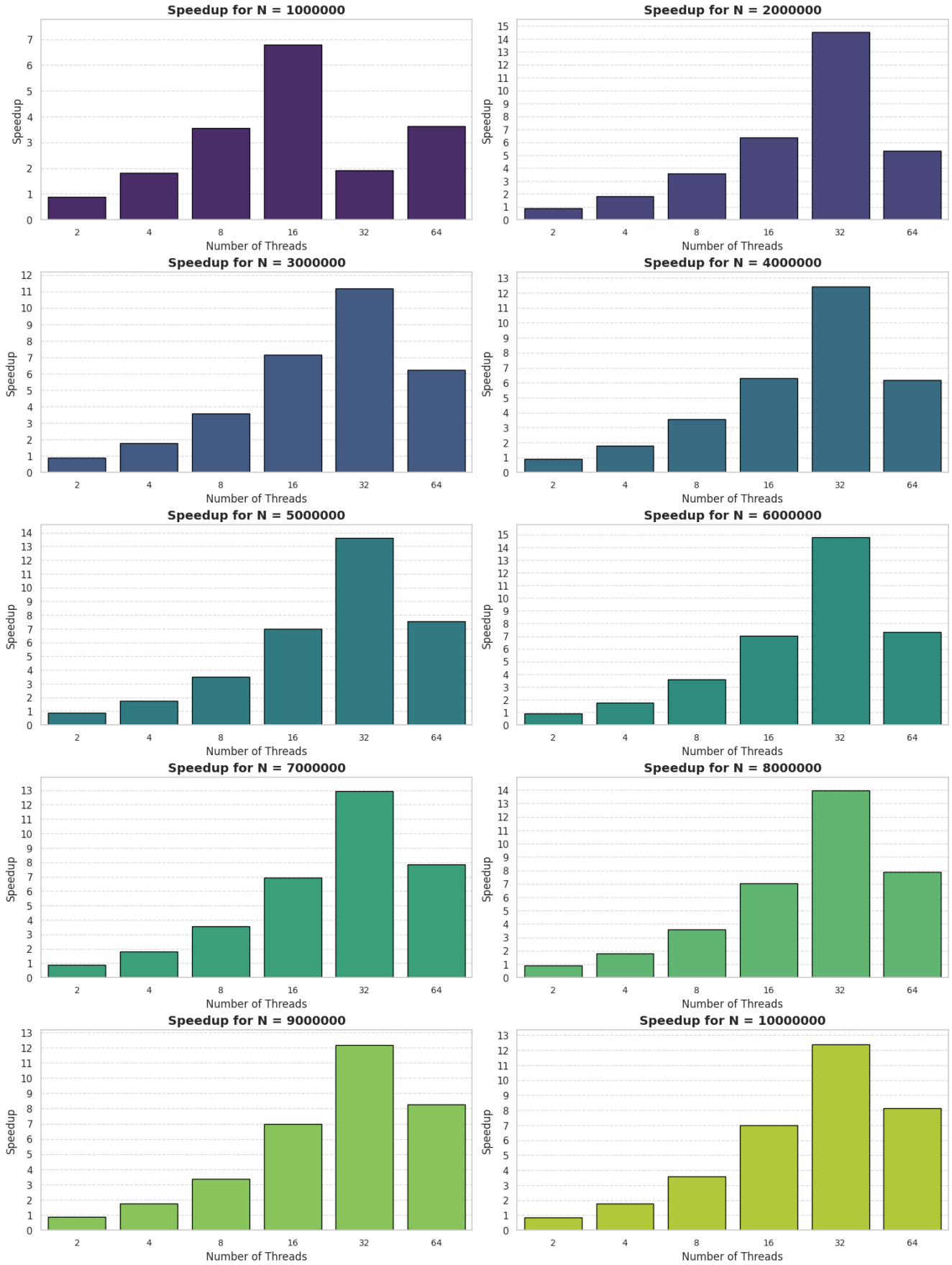

Figure 2: Speedups for different N values.

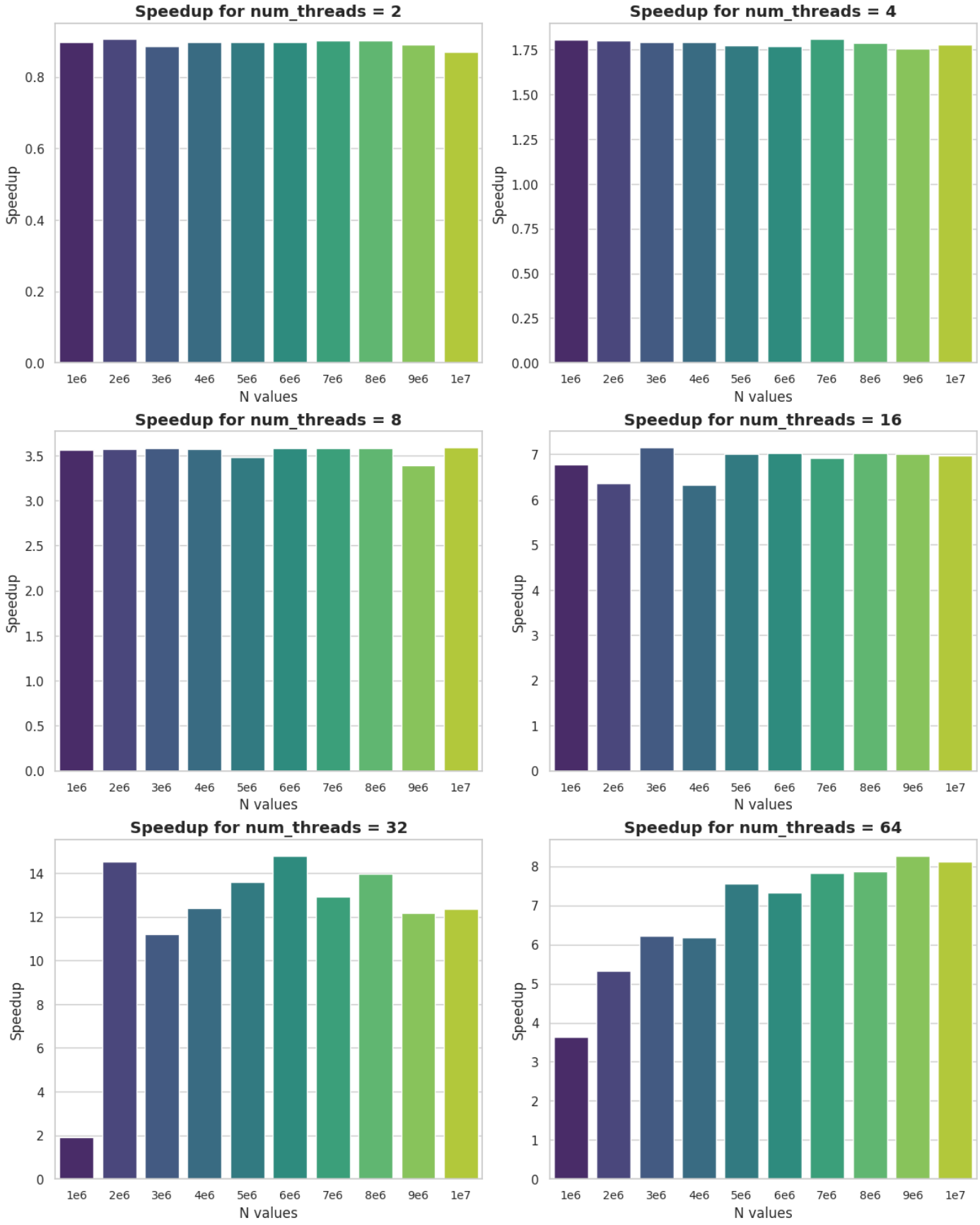

Figure 3: Speedups for different num_threads values.

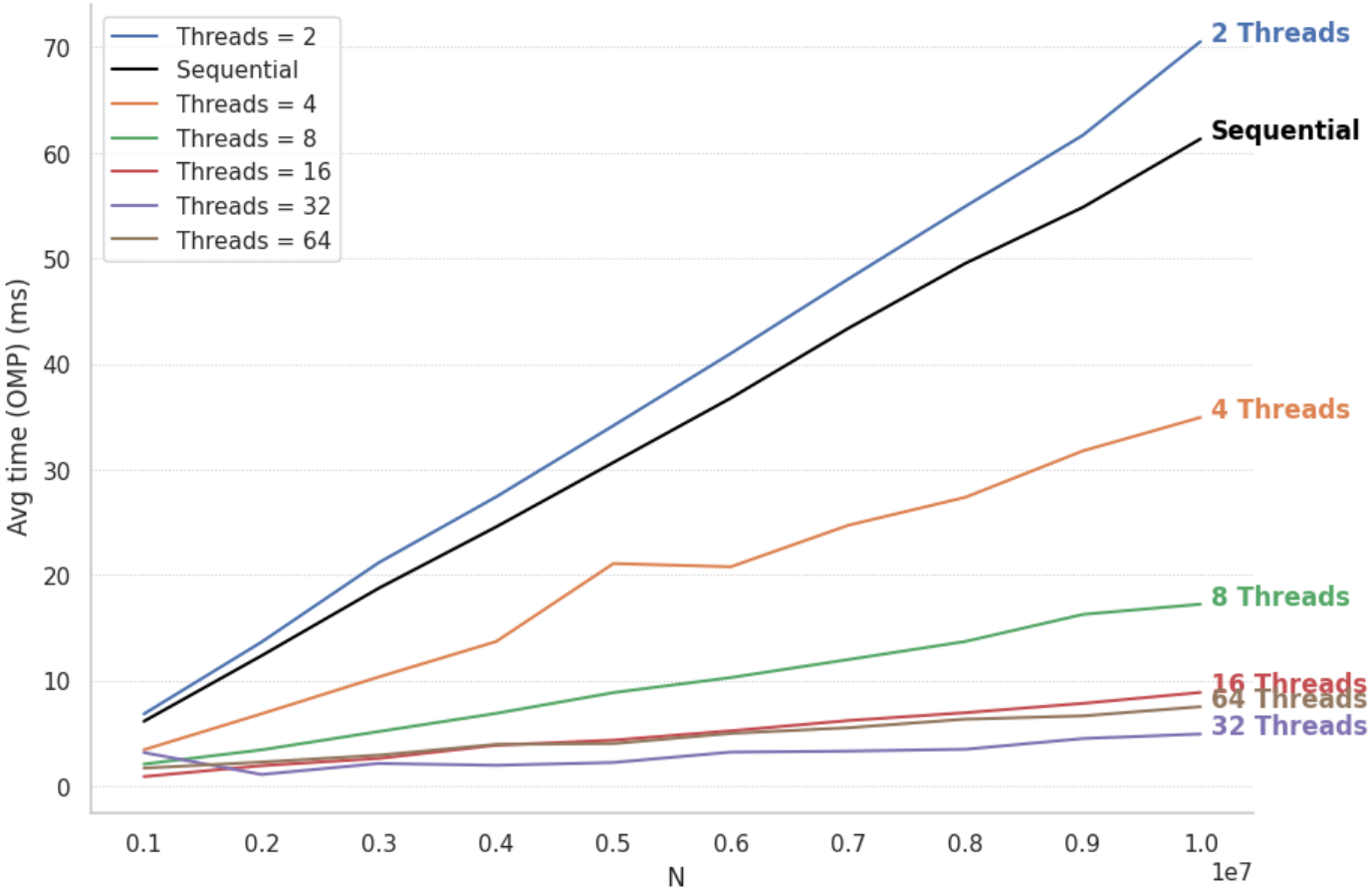

Figure 4: Time vs N plot for different values of num_thread values

4 Observations

1. ***In Figure 3 and Figure 4, for*** $num\_threads = 2$ ***we can see speeddowns across the board (around 0.9).*** :
   This can be explained by the fact that the time complexity of the serial code is $O(n)$ and that for parallel with t = 2 is :

$$O(\frac{2n}{2} + 2)) + parallelization\ overheads$$
$$= O(n) + parallelization\ overheads$$

   These parallelization overheads are the reason for slowdown. Therefore with our methodology, ***we need at least two cores to almost approximately match the performance of the serial code.***

2. ***In Figure 2, There is a slowdown for num_threads = 64 for every value of N (almost half of peak)***:

```
[namanp@ioe ~]$ lscpu
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              32
On-line CPU(s) list: 0-31
Thread(s) per core:  1
Core(s) per socket:  16
Socket(s):           2
NUMA node(s):        2
Vendor ID:           AuthenticAMD
CPU family:          23
Model:               49
Model name:          AMD EPYC 7302 16-Core Processor
```

   This is because the CPU provided to us for our profiling purposes has 32 cores and can execute 1 thread per core.

$$32 * 1 = 32 \text{ concurrent threads}$$

   Assigning 64 threads means multiple threads must share each core, leading to ***context switching overhead***. and hence a slowdown.

3. ***In Figure 2, for N = 1000000, we see a slowdown for num_threads = 32***:
   One explanation to this is because the parallelization overhead overtakes the time complexity for small N.

4. ***From Figure 3, except for very large t values (32 and 64), we see that the speedup is almost in a constant range.***

5. ***The average speed up values double for every num_thread doubling (till num_threads = 32 because of obs 2)***:

   We can see from the both Figures 2 and 3, that the speedup doubled for every doubling in threads. This is consistent with our time complexity analysis as $t << n$:

$$T(n,t) = O\left(\frac{2n}{t} + t\right)$$

$$T(n,t) \approx O\left(\frac{2n}{t}\right) \text{ as } t << n$$

$$T(n,2t) \approx O\left(\frac{2n}{2t}\right) = O\left(\frac{n}{t}\right) = \frac{1}{2}O\left(\frac{2n}{t}\right) = \frac{1}{2}T(n,t)$$

$$\frac{1}{T(n,2t)} = 2\frac{1}{T(n,t)}$$

$$\frac{T(n,1)}{T(n,2t)} = 2\frac{T(n,1)}{T(n,t)}$$

$$\boxed{S(n,2t) = 2S(n,t)}$$

Q2 MPI Finding an element in a large array(7.5 points)

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

For this program, generate a random integer array of size 1000000 (i.e., 1 million) with random integer elements between 1 to 5000000 (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). Now 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.

Execute the program with different number of processes including 1 (sequential), 8, 16, 32 and 64. For each number of processes, execute with 20 different random input numbers for searching and in each case, measure the time taken for the search. Report the average time (across the 20 instances) taken for each number of processes. Plot the execution times and speedups for different number of processes. Ensure that you execute different processes on different cores. Prepare a report with the methodology, execution times, and speedup graphs.

1 Methodology

1.1 Array Initialization and Distribution

The program begins by initializing the `MPI` environment and determining the rank and size of the MPI communicator. The total array size, `ARRAY_SIZE`, is divided equally among all processes, resulting in a local array of size `local_size` for each process.

- In the root process (rank 0), the global array is populated with random integers uniformly distributed between `MIN_ELEM` and `MAX_ELEM`.
- The `MPI_Scatter` function distributes chunks of the global array to all processes, assigning each process its corresponding local array.

1.2 Target Element Selection and Broadcasting

In each trial:

- The root process selects a random target element from the same range and broadcasts it to all processes using the `MPI_Bcast` function.

1.3 Local Search and Reduction

Each process performs the following steps:

- Search for the target element in its local array. If found, the local index is recorded; otherwise, a sentinel value (`ARRAY_SIZE + 5`) is used.
- Convert the local index to a global index based on the process's rank.
- Use the `MPI_Allreduce` function with the `MPI_MIN` operator to determine the smallest global index among all processes, representing the first occurrence of the target element.

2 Experiments

2.1 Experimental Parameters

The experiment uses the following parameters:

- `ARRAY_SIZE = 1,000,000`

- `MIN_ELEM = 1`
- `MAX_ELEM = 5,000,000`
- `NUM_TRIALS = 20`

2.2 Timing and Averaging

We are separately timing the time taken to distribute the array using `MPI_Wtime` and also separately timing the time taken to search for the element.
Note: in the question it is mentioned to use the time taken by `process=1` as sequential and we have done the same.

3 Results

The execution times and speed ups in numbers are mentioned in Appendix 2.

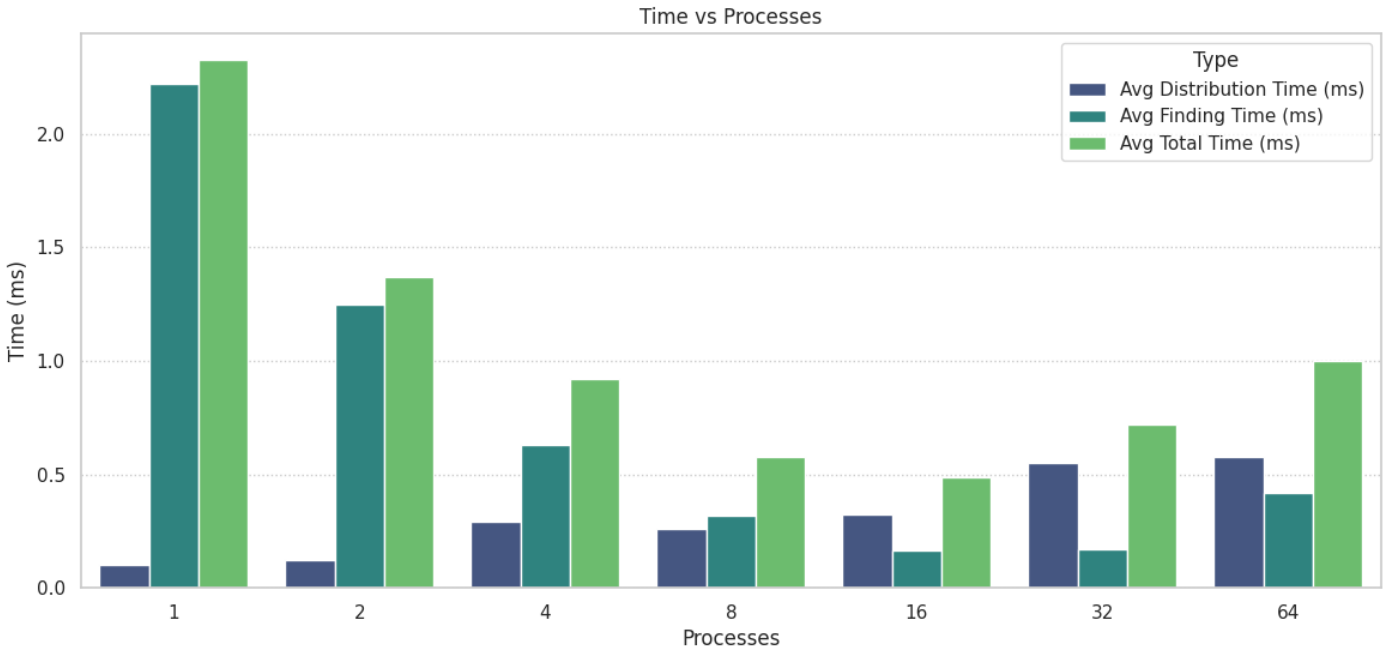

Figure 5: Time taken vs num_processes for N = 1,000,000 for distribution of data, finding element and total time.

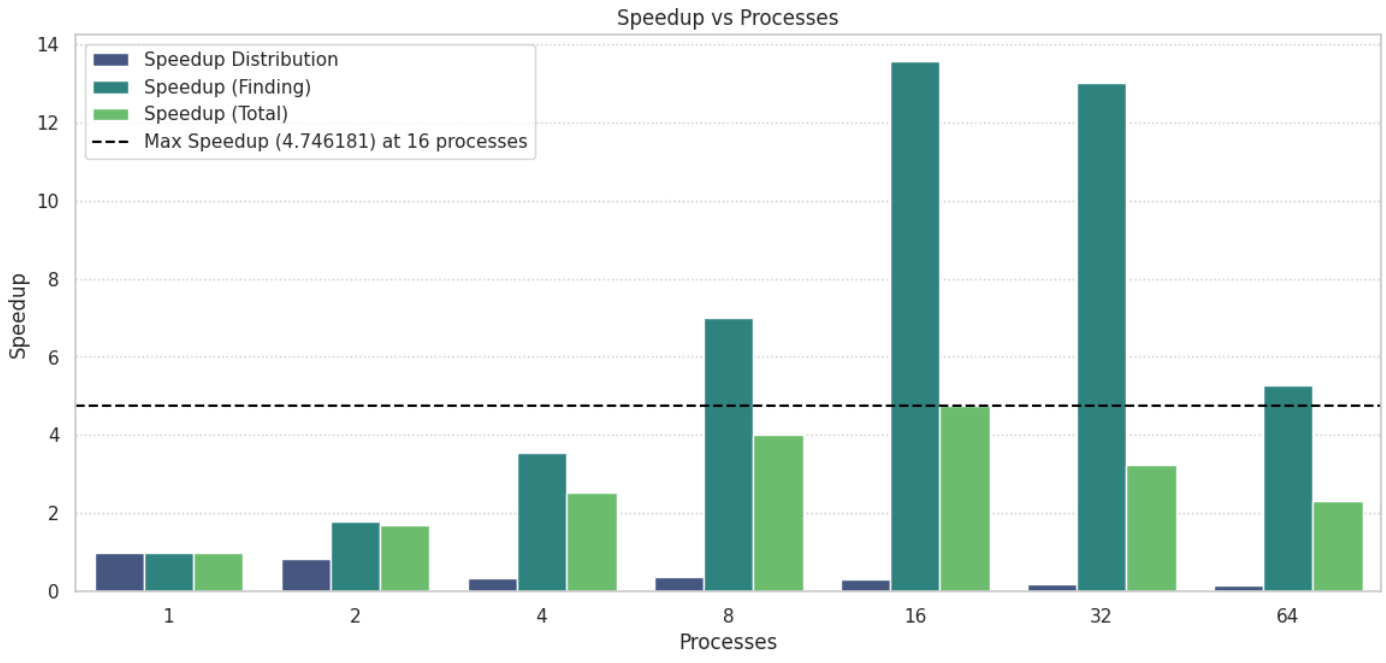

Figure 6: Speedup values vs num_processes for N = 1,000,000 for distribution of data, finding element and total time.

4 Observations

1. ***Avg Distribution time increases as processes increases***:

   This is expected because as processes increases, the communication time also must increase because now rank = 0 is sending the data to multiple other processes.

2. ***Avg Finding time decreases as processes increases. Moreover, it gets almost halved for each process doubling*** : This is also trivial as now the search region per process gets reduced to $\frac{ARRAY\_SIZE}{num\_processes}$ from $ARRAY\_SIZE$.

3. ***We get a maximum total speedup at processes = 16***:

   This is a consequence of 1 and 2. We have a trade-off here. Increasing num_processes increases the distribution time but reduces the finding time. We can see that there is a speedup in finding as processes increase but a slowdown in distribution (Form Figure 6). Hence we see even though element finding gets a speedup of around 14, the total speedup gets tanked to ≈ 4.746181 as there is a slowdown in array distribution.

4. ***The avg finding time gets increases for processes = 64*** :

   This seems counterintuitive at first, but this can be attributed to both parallelization communication overheads and similar reasons as `Q1::Observations::2` (**context switching**).

In the question it was mentioned:
.... 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.....

Which is similar to the approach employed in the class wherein a `MPI_Irecv` was used and the process once it found the element would send it to all the processes so that they could stop. A lower value of speedup was observed with that approach and hence I resorted on using `MPI_Allreduce`.

APPENDIX 1: Parallel openmp prefix sum results

Threads	N	Avg time (Seq) (ms)	Avg time (OMP) (ms)	Speedup
2	1000000	6.14711	6.84384	0.898197
2	2000000	12.349	13.642	0.905218
2	3000000	18.7492	21.1725	0.885543
2	4000000	24.5596	27.4071	0.896104
2	5000000	30.6714	34.161	0.897848
2	6000000	36.765	40.9983	0.896744
2	7000000	43.3712	48.0441	0.902737
2	8000000	49.5421	54.9397	0.901753
2	9000000	54.8437	61.673	0.889266
2	10000000	61.3151	70.5308	0.869338
4	1000000	6.23458	3.44651	1.80896
4	2000000	12.3555	6.84888	1.80402
4	3000000	18.529	10.3448	1.79115
4	4000000	24.5785	13.6958	1.79459
4	5000000	37.3856	21.0892	1.77273
4	6000000	36.8135	20.782	1.77141
4	7000000	44.7064	24.7138	1.80897
4	8000000	49.0091	27.3744	1.79033
4	9000000	55.8478	31.7654	1.75813
4	10000000	62.2262	34.9272	1.7816
8	1000000	7.43955	2.08745	3.56394
8	2000000	12.2628	3.4356	3.56934
8	3000000	18.5201	5.17478	3.57891
8	4000000	24.651	6.89381	3.57581
8	5000000	30.8679	8.86305	3.48276
8	6000000	36.8092	10.2873	3.57813
8	7000000	42.9332	11.9917	3.58023
8	8000000	49.04	13.708	3.57748
8	9000000	55.1261	16.2689	3.38842
8	10000000	61.8673	17.2359	3.58944
16	1000000	6.1392	0.905192	6.78221
16	2000000	12.3226	1.93853	6.35667
16	3000000	18.7812	2.6261	7.15174
16	4000000	24.5086	3.87811	6.31973
16	5000000	30.564	4.35826	7.01289
16	6000000	36.7813	5.24011	7.01919
16	7000000	43.0264	6.21369	6.92446
16	8000000	48.8786	6.95952	7.02328
16	9000000	54.926	7.84284	7.00334
16	10000000	61.9306	8.87595	6.97735
32	1000000	6.09912	3.1835	1.91585
32	2000000	16.0546	1.10533	14.5247
32	3000000	24.0895	2.15211	11.1934
32	4000000	24.4953	1.97336	12.413
32	5000000	30.4783	2.24099	13.6004
32	6000000	47.7387	3.22462	14.8045
32	7000000	42.864	3.31866	12.9161
32	8000000	48.8302	3.49501	13.9714
32	9000000	54.9348	4.51011	12.1804
32	10000000	61.0797	4.93918	12.3664
64	1000000	6.20154	1.70896	3.62883
64	2000000	12.1254	2.27641	5.32654
64	3000000	18.2182	2.93244	6.21264
64	4000000	24.4153	3.95159	6.17861
64	5000000	30.4608	4.03571	7.54781
64	6000000	36.6181	5.001	7.32215
64	7000000	43.3415	5.52984	7.83775
64	8000000	49.8982	6.34575	7.86325
64	9000000	55.0807	6.65583	8.27555
64	10000000	61.0623	7.52673	8.11273

APPENDIX 2: Parallel MPI find_max results

```
### **Table 1: Time vs. Processes**

| Processes | Avg Distribution Time (ms) | Avg Finding Time (ms) | Avg Total Time (ms) |
|-----------|----------------------------|-----------------------|---------------------|
| 1         | 0.101747                   | 2.221950              | 2.323697            |
| 2         | 0.123156                   | 1.247380              | 1.370536            |
| 4         | 0.290211                   | 0.628362              | 0.918573            |
| 8         | 0.260394                   | 0.317290              | 0.577684            |
| 16        | 0.325651                   | 0.163942              | 0.489593            |
| 32        | 0.549068                   | 0.170848              | 0.719916            |
| 64        | 0.577132                   | 0.420712              | 0.997844            |

### **Table 2: Speedups vs. Processes**

| Processes | Speedup (Distribution)  | Speedup (Finding) | Speedup (Total) |
|-----------|-------------------------|-------------------|-----------------|
| 1         | 1.000000                | 1.000000          | 1.000000        |
| 2         | 0.826164                | 1.781294          | 1.695466        |
| 4         | 0.350597                | 3.536099          | 2.529681        |
| 8         | 0.390742                | 7.002900          | 4.022436        |
| 16        | 0.312442                | 13.553269         | 4.746181        |
| 32        | 0.185309                | 13.005420         | 3.227734        |
| 64        | 0.176298                | 5.281404          | 2.328718        |
```