DS221: Introduction to Scalable Systems

Parallel Programming Assignment

Aneesh Panchal
06-18-01-10-12-24-1-25223
Indian Institute of Science (IISc), Bangalore, IN

NOV 2024

1 OpenMP Prefix Sum

The prefix sum, also known as the cumulative sum, is a critical operation in parallel computing, designed to compute the running sum of an array of numbers. This operation is commonly used in fields like numerical simulations, image processing, and data analysis. Over time, various parallel algorithms have been developed to perform the prefix sum efficiently in parallel computing environments. Typically, parallel prefix sum algorithms are divided into two stages. In this works we have chosen **Work Efficient Parallel Scan Algorithm** which is a variant of well known Blelloch Scan Algorithm.

1.1 Methodology

1. **Hillis-Steele Scan Algorithm** [1] works in two phases. In the upward phase, each element, except for the first, adds the value of its immediate neighbour with an offset that doubles in each iteration. During the downward phase, each element (except the last) sends its current value to its right neighbour with a doubling offset until the final step, where all elements have the correct prefix sum. Hillis-Steele have **high complexity** compared to other parallel algorithms but it is easy to implement.

2. **Kogge-Stone Scan Algorithm** [2] is an efficient parallel algorithm that operates by recursively halving the input array and applying a prefix sum on smaller sections, then combining the results. It uses a tree-like structure for efficient parallelization, with each processor working on a small part of the input array. The algorithm involves a series of stages where the sum is propagated through the array, reducing the number of operations at each step. Kogge-Stone is one of the best algorithm but this algorithm suffers **high communication overhead and complexity** due to which it is not that much useful.

3. **Franklin's Algorithm** [3] is another well-known parallel prefix sum algorithm designed for distributed memory systems. In Franklin's approach, the prefix sum is computed in a series of steps, where each processor works on an independent section of the array, and the intermediate results are combined efficiently. The algorithm uses a hybrid approach, where local computation is combined with communication between processors to ensure the prefix sum is computed correctly across the entire array. Franklin's algorithm also **suffers communication overheads**.

4. **Blelloch Scan (Up-Down Scan) Algorithm** [4]. This method also operates in two stages: the **Upward Sweep** (**Reduction Phase**), where the array is partitioned into chunks, and each thread computes a local prefix sum for its chunk, and the **Downward Sweep** (**Propagation Phase**), where results from each thread are propagated to neighbouring threads, adjusting the prefix sums across the entire array.

In our **Work Efficient Parallel Scan Algorithm** [5] implementation, the input array is divided into chunks, with each chunk assigned to a separate thread. Initially, each thread calculates the prefix sum for its chunk. After these local computations, threads synchronize using barriers, allowing them to propagate the accumulated sums from previous chunks to update their own values (***similar to propagation in a tree from leaf to root where leaves calculate chunks***). This synchronization step ensures the proper computation of the prefix sum for the whole array. A final downward sweep phase is then used to adjust the values, ensuring consistency across the entire array. Compared to the Blelloch and Hillis-Steele scans, this method **reduces the communication overhead** by performing local computations independently and propagating the necessary offsets only once. This leads to **fewer synchronization points and greater memory locality**, making the implementation simpler for shared-memory systems and potentially faster.

1.2 Time Complexity Analysis

Worst case time complexity of **Sequential** algorithm for array of size n is given by,

$$T_{\text{seq}} = O(n) \tag{1}$$

For Work Efficient Parallel Scan Algorithm for array of size n and t available number of threads,

1. Local prefix sum computation by every thread have time complexity of $O\left(\frac{n}{t}\right)$
2. Propagation of the accumulated sums have time complexity of $O(t)$
3. Synchronization overheads have time complexity of $O(\log(t))$

Worst case time complexity of **Work Efficient Parallel Scan Algorithm** for array of size n and t number of available threads is given by,

$$T_{\text{par}} = O\left(\frac{n}{t} + t + \log(t)\right) \tag{2}$$

Hence, **Speedup** due to parallelization is given by,

$$\text{Speedup } (S) = \frac{T_{\text{seq}}}{T_{\text{par}}} = \frac{O(n)}{O\left(\frac{n}{t} + t + \log(t)\right)} \tag{3}$$

1.3 Results

Table 1: Average Time Over 5 Runs for Different Numbers of Threads

n (Array Size)	No. of Threads	Parallel Time (sec)	Sequential Time (sec)	Speedup
10000	2	3.40798e-05	2.55238e-05	0.748942189
	4	3.5352e-05	2.55238e-05	0.721990269
	8	5.57778e-05	2.55238e-05	0.457597826
	16	0.000101688	2.55238e-05	0.251001101
	32	0.00944594	2.55238e-05	0.0027020074
	64	0.00267183	2.55238e-05	0.009552928
20000	2	6.24278e-05	4.751e-05	0.761039152
	4	0.000319717	4.751e-05	0.148600168
	8	6.80998e-05	4.751e-05	0.697652563
	16	0.000107276	4.751e-05	0.442876319
	32	7.04978e-05	4.751e-05	0.6739217394
	64	0.00256082	4.751e-05	0.018552651
30000	2	7.45556e-05	7.09318e-05	0.951394664
	4	0.000297325	7.09318e-05	0.238566552
	8	6.28338e-05	7.09318e-05	1.128879679
	16	9.87616e-05	7.09318e-05	0.718212342
	32	6.604e-05	7.09318e-05	1.074073289
	64	0.00261033	7.09318e-05	0.027173499
50000	2	9.86938e-05	0.000118328	1.198940562
	4	0.000316021	0.000118328	0.374430813
	8	6.51418e-05	0.000118328	1.816468074
	16	0.000103182	0.000118328	1.146789169
	32	7.03858e-05	0.000118328	1.681134547
	64	0.00264063	0.000118328	0.044810519
100000	2	0.000161091	0.000235521	1.462036985
	4	9.71238e-05	0.000235521	2.424956602
	8	0.000142171	0.000235521	1.656603667
	16	0.0001119	0.000235521	2.104745308
	32	7.34218e-05	0.000235521	3.2077802506
	64	0.0025905	0.000235521	0.090917197
500000	2	0.000645398	0.00117466	1.820055222
	4	0.000325779	0.00117466	3.605695886
	8	0.000200899	0.00117466	5.847017656
	16	0.000168301	0.00117466	6.979518838
	32	0.00011912	0.00117466	9.8611484218
	64	0.00293929	0.00117466	0.39964073
1000000	2	0.0012252	0.00237893	1.941666667
	4	0.000612924	0.00237893	3.88128055
	8	0.000348695	0.00237893	6.82238059
	16	0.000249957	0.00237893	9.517356985
	32	0.000171923	0.00237893	13.8371829249
	64	0.00319223	0.00237893	0.745225125

n (Array Size)	No. of Threads	Parallel Time (sec)	Sequential Time (sec)	Speedup
5000000	2	0.00593194	0.0117046	1.973148751
	4	0.00295955	0.0117046	3.954858002
	8	0.00161036	0.0117046	7.268312675
	16	0.000914183	0.0117046	12.80334463
	32	0.000696336	0.0117046	16.8088394109
	64	0.00364378	0.0117046	3.21221369
10000000	2	0.0117798	0.0235231	1.996901475
	4	0.00588976	0.0235231	3.993897884
	8	0.00308026	0.0235231	7.636725471
	16	0.00168012	0.0235231	14.00084518
	32	0.000921947	0.0235231	25.5145903181
	64	0.0042361	0.0235231	5.553008664
50000000	2	0.0587335	0.117121	1.994108984
	4	0.0294203	0.117121	3.980958726
	8	0.0149504	0.117121	7.833970997
	16	0.00772356	0.117121	15.16412121
	32	0.00384764	0.117121	30.4394981007
	64	0.0108215	0.117121	10.82299127
100000000	2	0.119893	0.234457	1.955552034
	4	0.0594066	0.234457	3.946649026
	8	0.0296755	0.234457	7.90069249
	16	0.0152002	0.234457	15.42459968
	32	0.00763206	0.234457	30.7200153039
	64	0.0164757	0.234457	14.23047276
500000000	2	0.587964	1.17277	1.994628923
	4	0.306457	1.17277	3.826866412
	8	0.147269	1.17277	7.963454631
	16	0.07375	1.17277	15.9019661
	32	0.0376841	1.17277	31.121082897
	64	0.0533975	1.17277	21.96301325
1000000000	2	1.17237	2.34267	1.998234346
	4	0.587892	2.34267	3.984864567
	8	0.295155	2.34267	7.937083905
	16	0.155265	2.34267	15.08820404
	32	0.0796617	2.34267	29.4077329507
	64	0.105988	2.34267	22.10316262

1.4 Observations

From Table 1, Fig. 1,2, 3, 4 and 5, the following observations can be made,

1.4.1 Reduction in Runtime

- Worst case time complexity for sequential prefix sum algorithm is $O(N)$ which can be seen from Fig. 2 and 3. The running time increases proportional to the increase in size of the array.
- The sequential time complexity for parallel prefix sum algorithm (when the array is not of parallelizable size) is $O(N)$. But considering parallelization, the complexity is reduced to $O(\log(N)) < O(n)$ which is derived above in equations (1) and (2).
- For arrays of sizes $n \leq 10^5$, we observe that the parallel runtime exceeds sequential runtime. This is because even though the time complexity is lower for parallel execution, the synchronization overhead adds to the runtime. The overheads largely outweigh the gain due to parallelization.
- For arrays of sizes $n \geq 10^5$, we observe that the parallel execution time is lower than the sequential execution time. Hence we can conclude that parallelization is beneficial when we have large amounts of data but may have the opposite effect when dealing with smaller amounts of data.

1.4.2 Effect of Number of Threads

1. For array size, $n > 10^5$ from equation (3) we have Speedup, $S > 1$ for every number of thread. For large arrays, the advantages of parallelization surpass the costs associated with communication and synchronization, making parallel execution advantageous.

2. For array size $n \leq 10^5$ from equation (3) we have Speedup, $S \leq 1$. In the case of smaller arrays, the overheads from communication and synchronization negate the performance gains from parallelization, making parallel execution less beneficial.

1.4.3 Observations from Graphs

1. Fig. 1 shows that sequential time for small and large size array is saturated ($O(n)$) which is obvious. But parallel time decreases for larger size array and remain almost constant for smaller size arrays as number of threads increases and gets saturated after 32 threads due to limited availability of CPUs. However, speedup for larger size array in parallel computations increases upto 32 threads while it remains almost same for smaller size array.

2. Fig. 2 shows that the difference between sequential time and parallel time increases as number of number of threads increases showing the efficiency increase. Fig. 3 shows the same using log-log scaling, it is clear that when number of threads are 32, the prefix sum algorithm implemented performs best as it goes below sequential time very rapidly because maximum utilization of available resources happens in this case.

3. Fig. 4 shows the speedup vs number of threads using different array size. It can be seen that all the lines attain maxima at around 32 number of threads. This is due to maximum utilization of available resources.

4. Fig. 5 shows the speedup vs array size using different number of threads. It can be seen that 32 threads performs best, however 1 more observation can be made that 64 threads performs better than 16 and lower number of threads because 32 threads are utilizing here as well and some overheads are also there which makes it lower than 32 but higher than 16.

1.4.4 General Observations

1. Speedup for smaller arrays is almost unaffected by the number of threads which validates our earlier assumptions.

2. Speedup for larger arrays increases with an increasing number of threads due to increasing degree of concurrency (number of threads being executed concurrently).

3. Further we observe a decrease in speedup from 32 threads to 64 threads. This is because our node is limited by 32 available CPUs. At 32 threads, we have 32 available CPUs with each CPU working with each thread reaching optimal performance for the system with a degree of concurrency 32. Increasing the number of threads further from 32 does not increase of degree of concurrency as we do not have additional CPUs for concurrent processing. From 32 to 64 threads, there is no increase in the degree of concurrency, but there is an increase in synchronization overheads leading to lowering of speedup.

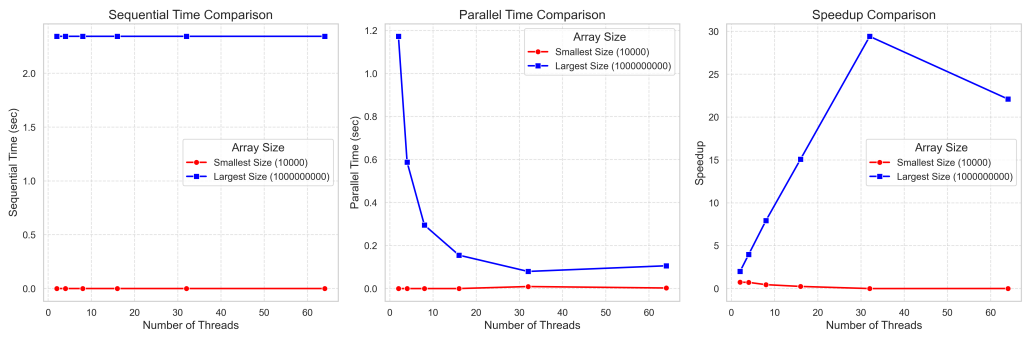

Figure 1: Comparison of Sequential Time, Parallel Time, and Speedup between the smallest and largest array sizes.

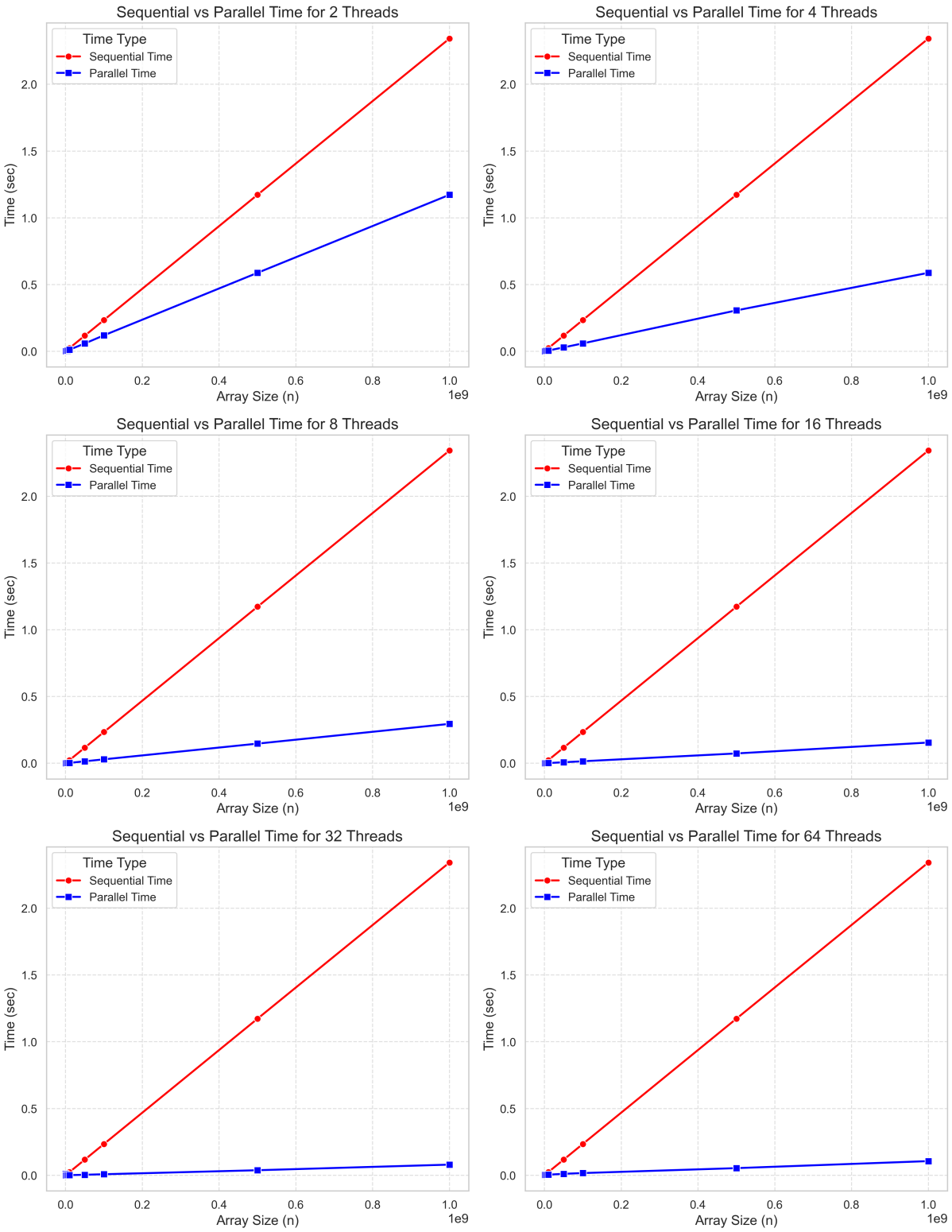

Figure 2: Comparison between the Sequential and Parallel Time for different Array Sizes (n).

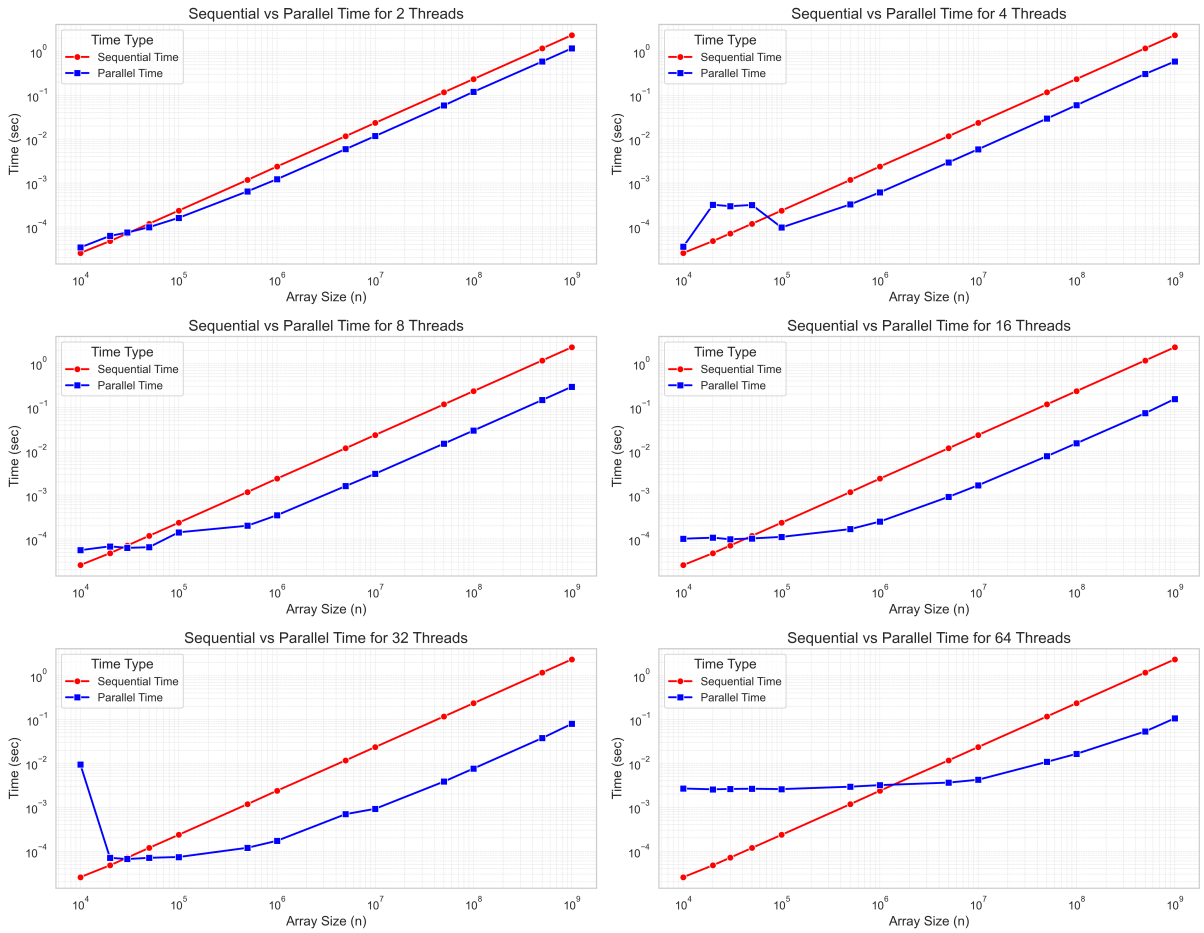

Figure 3: Comparison between the Sequential and Parallel Time for different Array Sizes (n) (**Logarithmic Scale**).

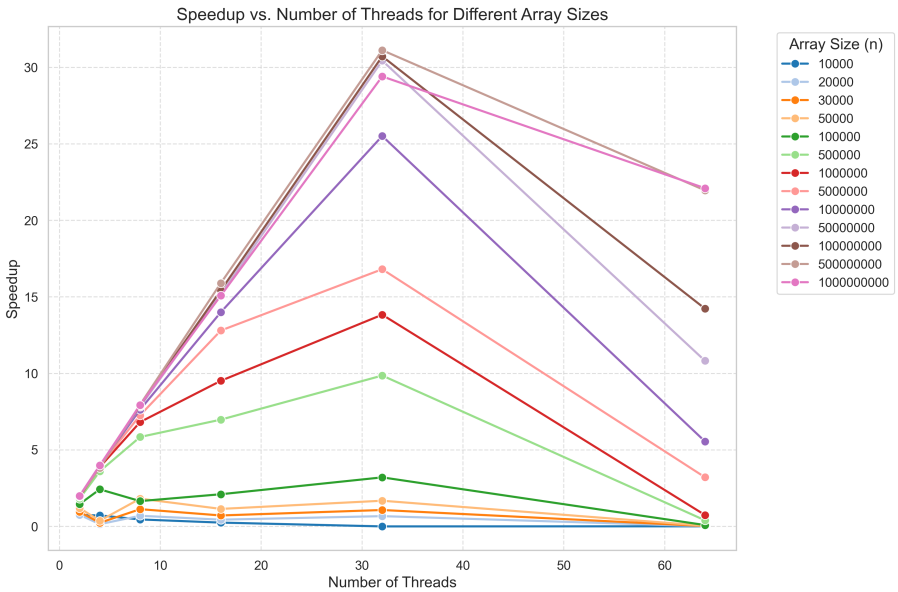

Figure 4: Speedup vs. Number of Threads for Different Array Sizes

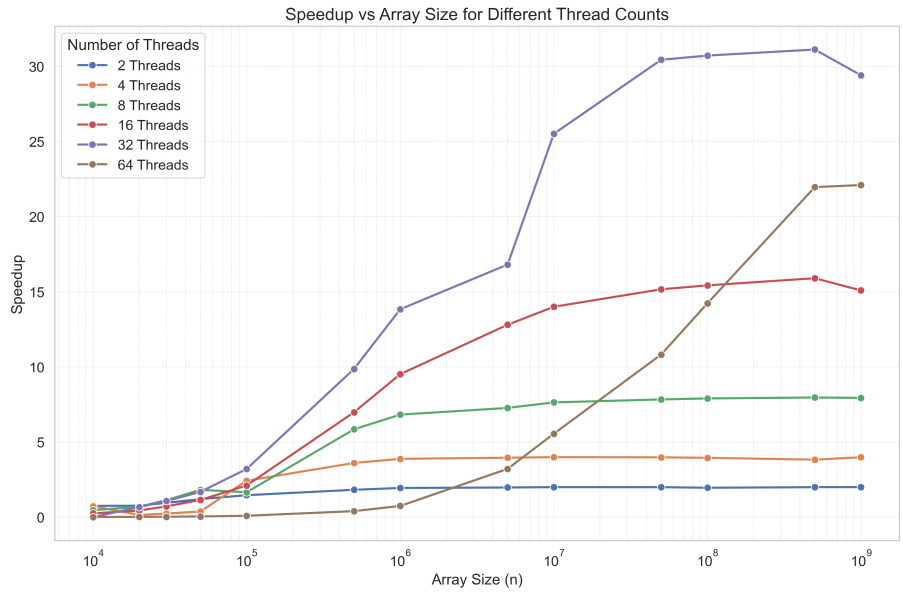

Figure 5: Speedup vs. Array Size (n) for Different Number of Threads.

2 MPI Finding an Element in a Large Array

Searching is the process of finding a specific item or element within a collection of data. This collection can be in different forms, like arrays, lists, or trees. The main goal of searching is to check if the item exists in the data and, if it does, find its exact location or retrieve it. Searching is important in many tasks, both in computers and everyday life, such as finding information, analyzing data, and making decisions.

2.1 Methodology

The algorithm we use is structured into the following key components,

1. **Main Function**
   This initializes MPI, allocates the necessary arrays, and distributes the data across multiple processes. Each process then carries out parallel search tasks for a specific target value, and the root process (rank 0) calculates and prints the average execution time.

2. **distributeArray**
   The root process creates a random array and splits it across all processes, with each process receiving a portion of the array to work on.

3. **localSearch**
   Each process searches for the target value within its assigned portion of the array. If the target is found, the process returns its index. Otherwise, it returns -1.

4. **parallelSearch**
   This function coordinates the overall search by managing the tasks of all processes. It uses `MPI_Bcast` to collect the results from each process and identifies the global index of the target value.

5. **performTrials**
   Multiple trials of the parallel search are executed in this step. The target values are broadcast to all processes, the time for each trial is recorded, and the root process computes the average search time.

This parallel approach[1] enhances search performance in large datasets by leveraging multiple processes.

[1] **Explanation of each step is provided in last subsection**

2.2 Complexity Analysis

Worst case time complexity of **Sequential Linear Search Algorithm** for array of size n is given by,

$$T_{seq} = O(n) \tag{4}$$

Worst case time complexity of **Parallel Linear Search Algorithm** for array of size n and p number of processes is given by,

$$T_{par} = O\left(\frac{n}{p}\right) + O(\log(p)) \tag{5}$$

where, $O(\log(p))$ is the added component caused by communication overheads in MPI.

Parallelization becomes increasingly efficient as the value of n grows and p increases proportionally to it. As n grows and p grows proportionally, $\log\left(\frac{n}{p}\right) << \log(n)$ and the benefit due to a higher degree of concurrency outweighs the communication overhead $O(\log(p))$.

2.3 Results

Number of Processes	Average Time (sec)	Speedup
1	0.000299223	1
8	4.32691e-05	6.9154
16	2.22956e-05	13.4207
32	1.1567e-05	25.8687
64	1.1975e-05	24.9873

Table 2: Average Time over 20 Trials for Different Numbers of Processes

2.4 Observations

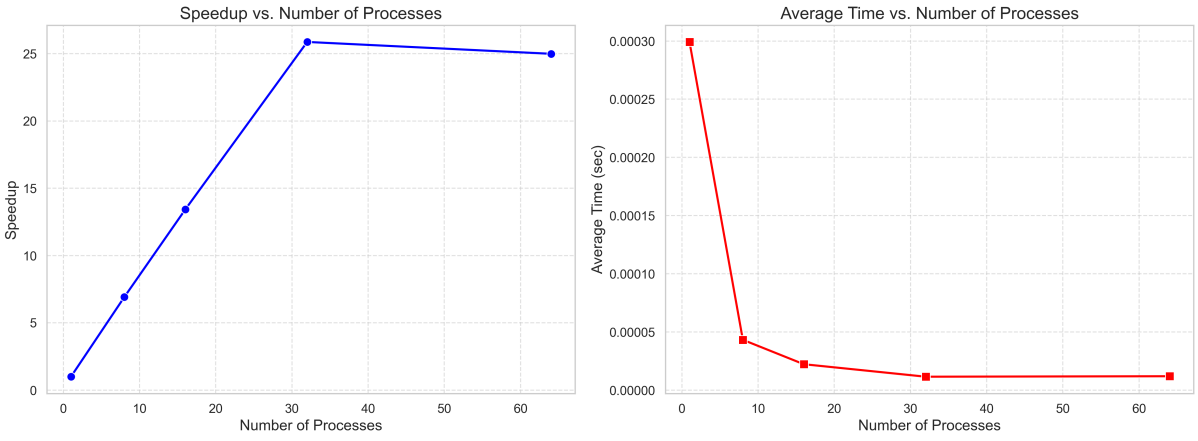

Figure 6: Speedup and Average Time for different numbers of processes in parallel program using MPI.

2.4.1 Effect of Increasing Number of Processes

Number of tasks per node $= 32$ (set by configuration in *job-submit.sh*)

From Table 2, we can see that as the number of processors increases, the average runtime decreases, which aligns with the expectations based on the complexity analysis of both the sequential and parallel algorithms. However, there is a slight reduction in speedup when increasing the number of processes from 32 to 64. This behavior can be explained by the system configuration, which allows a maximum of 32 processes to run in parallel per

node. Processes beyond this limit are queued and executed sequentially on the next node. As a result, the level of concurrency remains unchanged, and any additional processes must wait for previous tasks to complete, preventing further speedup beyond the 32 process threshold.

Fig. 6 illustrates this phenomenon, showing an increase in speedup and a decrease in average runtime as the number of processes grows, but with a noticeable plateau once the number of processes exceeds 32.

2.5 Explanation of each step used for MPI Execution

1. **Initialize MPI**: MPI environment is initialized, and processes are set up.
2. **Calculate chunk sizes and remainder**: The array is divided into chunks, and any remainder from the division is accounted for.
3. **Generate a random array (Rank 0)**: The root process (Rank 0) generates a random array of integers to be searched. The array is populated with values in the range $[1, \text{max\_val}]$.
4. **Distribute the array to all processes**: The array is split and distributed across the processes. Each process gets its own chunk of the array. For processes beyond Rank 0, data is sent using `MPI_Send`.
5. **Execution of Local Search**: Each process performs a local search within its assigned chunk to find the target value. This is done using the `localSearch` function, which returns the index if the value is found or -1 if not.
6. **Broadcast Target Value**: The target value to be searched is generated by Rank 0 and broadcast to all processes using `MPI_Bcast`.
7. **Measure Search Time**: The execution time for each trial is measured using `MPI_Wtime`, ensuring that time taken for each search is recorded.
8. **MPI Synchronization**:
   (a) If a process finds the target value, it returns its local index and sends this information to the root process.
   (b) The root process (Rank 0) uses `MPI_Bcast` to notify all processes of the search result.
   (c) The global index of the target value is calculated by each process based on its local findings. The global index is derived by considering the chunk size and rank.
9. **Complete the Trial**: After each search trial, the time taken for that trial is recorded, and once all trials are done, the average time for the search process is calculated.
10. **Final Output (Rank 0)**: The root process computes and outputs the average time over all trials.
11. **Finalize MPI**: After all trials and data gathering, the MPI environment is finalized using `MPI_Finalize`.

References

[1] W. D. Hillis and G. L. Steele. "Data Parallel Algorithms." In *Proceedings of the 1986 ACM Conference on Parallel Processing*, 1-12. ACM, 1986.

[2] Kogge, Peter M., and Harold S. Stone. "A parallel algorithm for the efficient solution of a general class of recurrence equations." *IEEE transactions on computers* 100, no. 8 (1973): 786-793.

[3] Franklin, M. "Parallel Prefix Computation: A Theoretical and Practical Analysis." 1991.

[4] Guy E. Blelloch. "Parallel Prefix Computation." In *Proceedings of the 1st Annual ACM Symposium on Parallel Algorithms and Architectures*, 1-10. ACM, 1989.

[5] S. Fraser, H. Xu and C. E. Leiserson, "Work-Efficient Parallel Algorithms for Accurate Floating-Point Prefix Sums," 2020 *IEEE High Performance Extreme Computing Conference (HPEC)*, Waltham, MA, USA, 2020, pp. 1-7, doi: 10.1109/HPEC43674.2020.9286240.