Report on Parallel Binary Tree Search Algorithm Using MPI

Introduction

The goal of this project was to implement a parallel binary tree search algorithm using MPI (Message Passing Interface). The implementation focuses on distributing the work among multiple processes to locate multiple target values in a large binary tree. The performance of the parallel search was evaluated based on speed-up and scalability metrics.

Program Overview

Binary Tree Creation

The binary tree created in the CreateTree function. This function initializes a binary tree with random values and ensures that the target values are inserted into random nodes. The tree is represented as an array where each node has four attributes: ID, Age, Left child index, and Right child index.

Work Distribution Algorithm

The work distribution algorithm assigns the nodes of the binary tree to different MPI processes based on the number of available processes. The algorithm follows these rules:

1. Single Process: If there is only one process, it starts from the root of the tree.
2. Multiple Processes: All processes start from the root of the tree to ensure they have access to all potential target values.

Parallel Search

Each process is assigned a subset of the target values to search for in the tree. The search function is modified so that each process starts from the root of the tree, ensuring that all potential target values are accessible.

Performance and Scalability

Speed-Up Observations

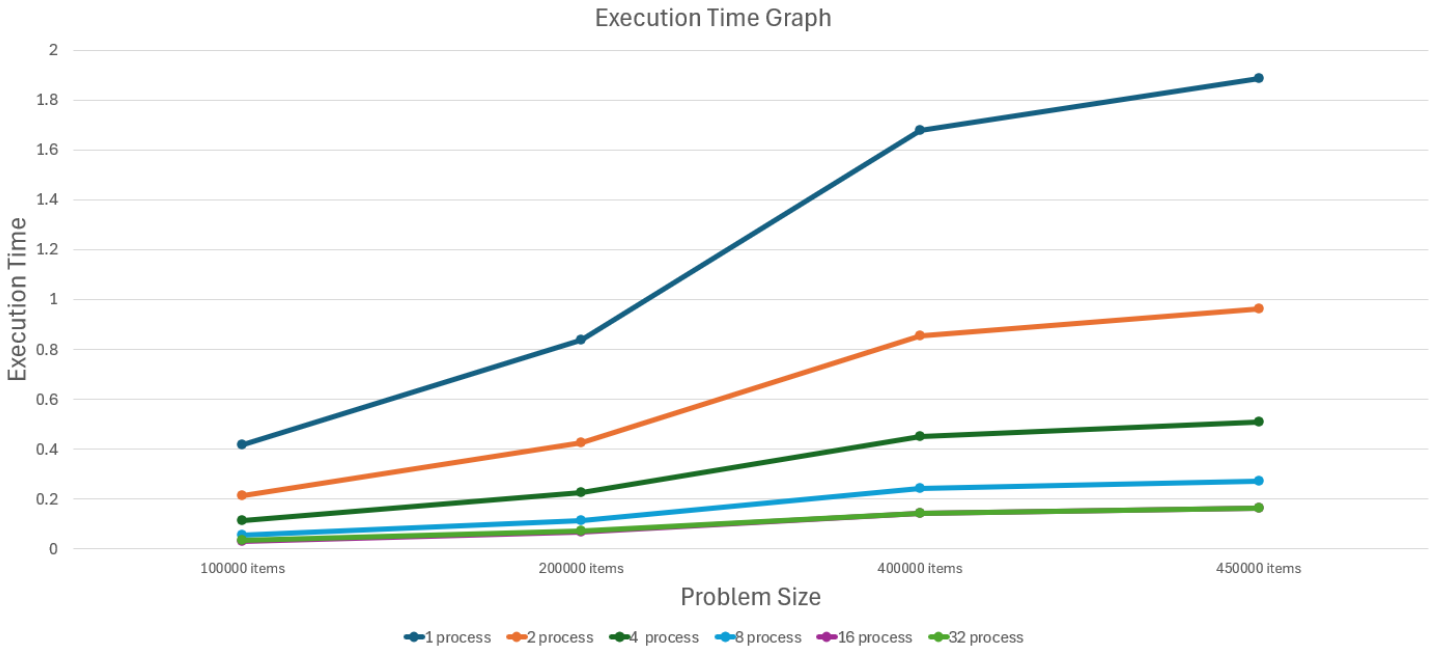
The parallel implementation showed notable speed-ups with an increasing number of processes. Each process independently searches for its assigned target values and synchronizes at an MPI_Barrier after completing its search. The recorded search times for each process are gathered and analyzed to determine performance.

Initially, the program used MPI_Abort to terminate when a process found a target value, but this resulted in multiple processes printing results before termination. The approach was revised so that each process saves its search time, synchronizes at a barrier, and sends the recorded times to the root process. The root process prints the shortest search time and the corresponding process ID.

Execution Time

The total execution time is recorded for all search operations. Each process's execution time is gathered using `MPI_Gather`, and the maximum execution time across all processes is determined using `MPI_Reduce`. This time is printed as the total execution time for the parallel search operation.

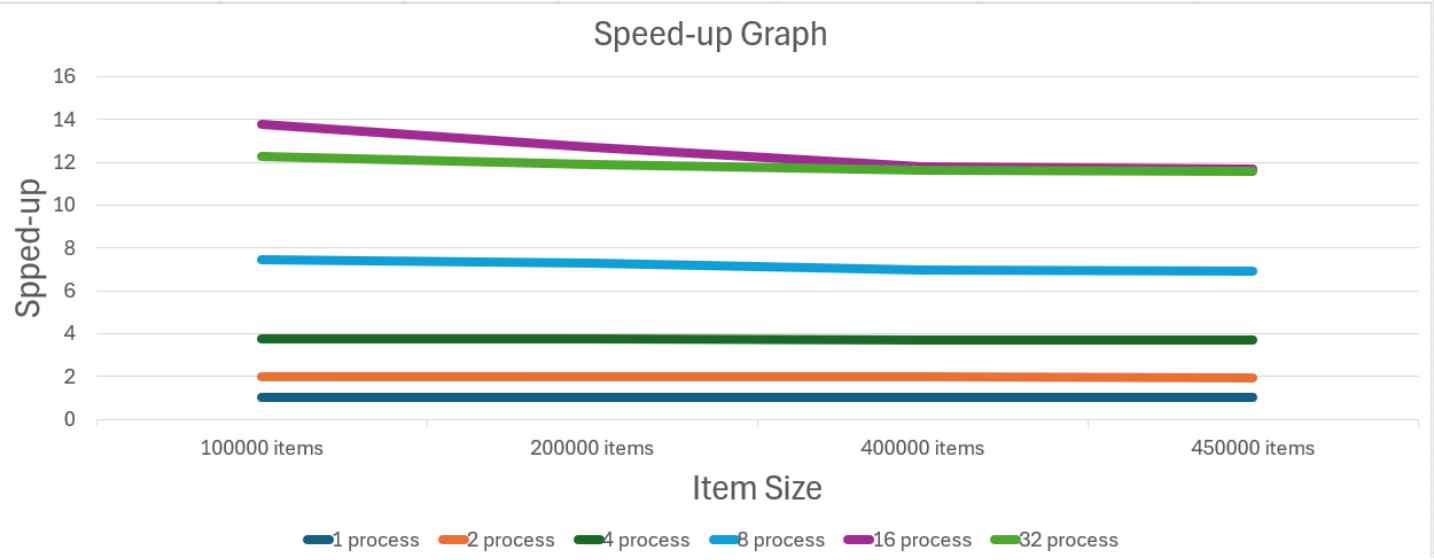
Size of the problem	1 process	2 process	4 process	8 process	16 process	32 process
100000 items	0.418704	0.211614	0.111438	0.056267	0.030459	0.034143
200000 items	0.839109	0.424773	0.224676	0.114817	0.066138	0.070570
400000 items	1.678039	0.855085	0.450592	0.240937	0.142106	0.144246
450000 items	1.888418	0.963047	0.508957	0.272691	0.161658	0.163226



Speed-Ups

The parallel search algorithm showed speed-ups as the number of processes increased. Each process efficiently handled its subset of target values, contributing to overall reduced search times.

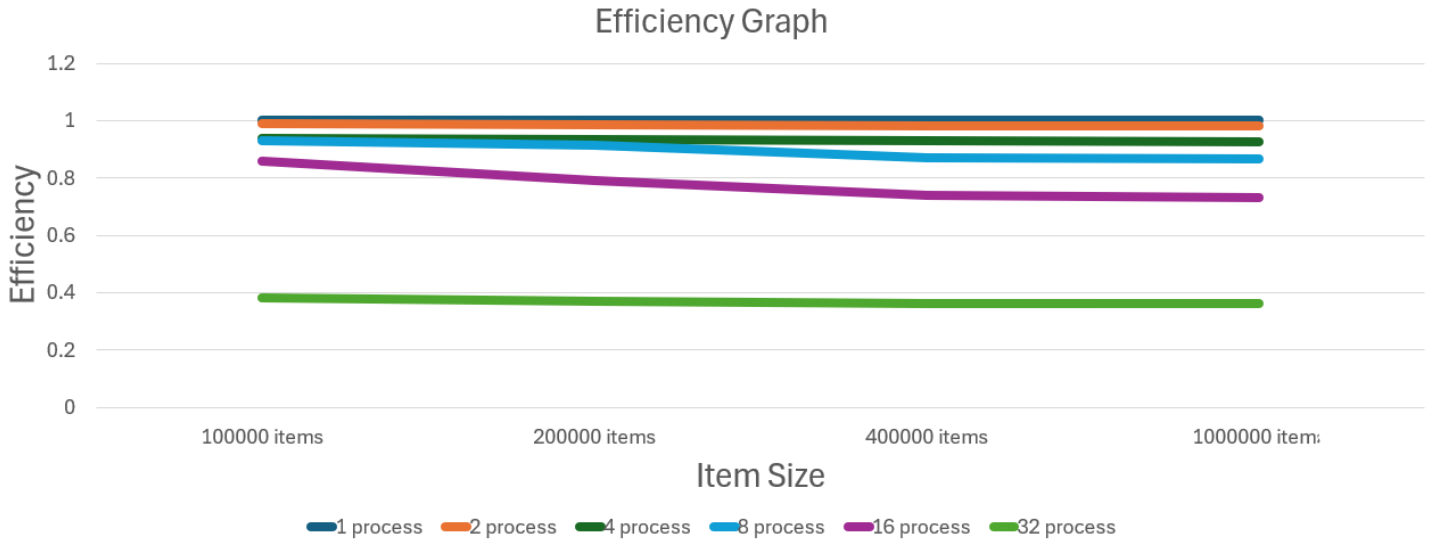
Size of the problem	1 process	2 process	4 process	8 process	16 process	32 process
100000 items	1	1.979	3.757	7.441	13.746	12.263
200000 items	1	1.975	3.735	7.308	12.687	11.890
400000 items	1	1.962	3.724	6.965	11.808	11.633
450000 items	1	1.961	3.710	6.925	11.682	11.569



Efficiencies

Efficiencies were calculated based on the speed-up observed for varying numbers of processes. The efficiency decreased as the number of processes increased, indicating some overhead in parallel execution.

Size of the problem	1 process	2 process	4 process	8 process	16 process	32 process
100000 items	1	0.989	0.939	0.930	0.859	0.383
200000 items	1	0.988	0.934	0.914	0.793	0.372
400000 items	1	0.981	0.931	0.871	0.738	0.364
450000 items	1	0.980	0.928	0.866	0.730	0.362



Strong Scalability

Strong scalability refers to the ability to increase the number of processes while keeping the problem size fixed and observing a fixed efficiency. Based on the results, as we increase the number of processes while keeping the problem size fixed, the efficiency does not remain fixed. Therefore, this problem is not strongly scalable.

Weak Scalability

Weak scalability refers to the ability to keep the efficiency fixed by increasing the problem size at the same rate as the number of processes. Based on the results, as we increase the problem size and the number of processes by a factor of two, the efficiency does not remain fixed. Therefore, this problem is not weakly scalable.

Conclusion

In conclusion, the parallel binary tree search algorithm using MPI was implemented successfully. The revised work distribution algorithm ensured that all processes started from the root, enabling access to all target values. The alternative approach to handling target value discovery ensured accurate and synchronized results. The implementation demonstrated notable speed-ups, though strong and weak scalability were not achieved.