Parallel & Distributed Computer Systems 2023-2024
Homework 2: Distributed k-select

Evangelos Kyriakos

January 13, 2024

Abstract

This report delves into the intricacies of solving the distributed k-select problem using Message Passing Interface (MPI). The core of our approach is the utilization of a sequential function for in-place partitioning, inspired by a MATLAB implementation. The report documents the journey from conceptualizing the problem to implementing a parallel solutions in Julia. Subsequently, we conduct a comprehensive evaluation of the implementation, offering insights into their performance.

1 Introduction

The distributed k-select problem emerges as a vital challenge in the realm of parallel computing. The objective is to identify the kth smallest element within an array through the application of MPI. Drawing inspiration from a MATLAB implementation, we leverage a sequential function for in-place partitioning. This section outlines the importance of the k-select problem, introduces the MPI paradigm as the chosen methodology, and sets clear objectives for the subsequent implementations.

2 Methodology

2.1 Sequential Implementation

The journey begins with a meticulous exploration of the sequential implementation in Julia. This initial step involves understanding and adapting the provided in-place partitioning function. Our aim is to develop a robust and functional baseline before transitioning to parallel implementations.

A single-threaded k-select implementation relies on the concept of partitioning to efficiently find the kth smallest element in an array. The idea is to iterative narrow down the search space based on the properties of the elements compared to a chosen pivot element.

The algorithm takes two main inputs: an array A of length n and an integer k representing the desired order statistic (e.g., kth smallest element).

The heart of the algorithm is the partitioning step. A pivot element is chosen from the array, and the array is rearranged such that elements less than the pivot are on its left, and elements greater than the pivot are on its right. This process is typically done in-place to avoid unnecessary memory overhead.

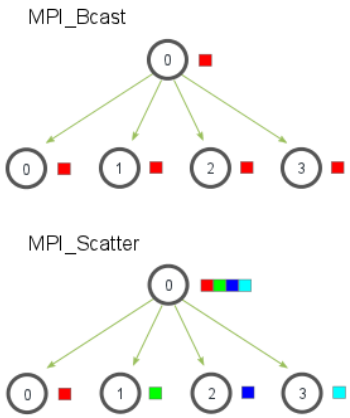

Figure 1: Broadcast and Scatter MPI differences

The partitioning step is often encapsulated within a function. The function takes the array A and the pivot element as parameters and returns three values:

- p: the index where elements to the left are less than the pivot,
- q: the index where elements equal to the pivot start, and
- A: the modified array after partitioning.

Recursive Step: Based on the results of partitioning, the algorithm decides which part of the array to focus on next:

- If k is equal to p, then the pivot is the kth smallest element, and the algorithm terminates.
- If k is less than p, then the kth smallest element is in the left sub array, and the algorithm recursively applies k-select to the left sub array.
- If k is greater than q, then the kth smallest element is in the right sub array, and the algorithm recursively applies k-select to the right sub array.

The recursive calls continue until the base case is reached. In the base case, the size of the sub array becomes small enough (e.g., 1 or 2 elements), and the kth smallest element is directly determined without further recursion.

Once the recursive calls lead to the pivot being the kth smallest element, the algorithm terminates, and the pivot is returned as the final result. This algorithm exhibits a divide-and-conquer strategy, leveraging the properties of the partitioning step to efficiently reduce the search space. The time complexity of this algorithm is typically $O(n)$ on average, with a worst-case time complexity of $O(n^2)$ in the absence of proper pivot selection strategies or for sorted input arrays. However, randomized pivot selection and other optimizations can mitigate this issue, making the algorithm efficient in practice.

2.2 Parallel Implementation

Here, the focus is on multi threading, which is an inherent aspect of Julia. We discuss the selection of packages and tools, providing an overview of how Julia's capabilities are harnessed to tackle the problem in a parallel fashion.

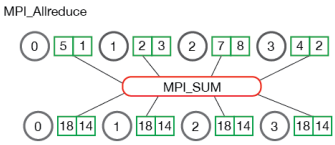

Figure 2: MPI Allreduced communication pattern used in pivot summation

The threaded form of the partition function implements the usage of threaded partitioning using a ping-pong buffer. It consists of a local array to be partitioned withing each MPI Processes / rank, a pivot element, and a result buffer to store the result of partitioning. The function uses a ping-pong buffer mechanism to parallel the partitioning process. It iterates through the local array and assigns elements to either the ping buffer or the pong buffer based on their relationship with the pivot v. The resulting ping and pong buffer are then copied back to the result buffer, excluding any unused slots.

The parallel k select function moreover, takes three parameters: a local array to be processed withing each MPI process, the k desired order statistic and the MPI communicator.

The rest of the function works as follows:Selects the pivot as the first element of local array. Then call threaded partition to partition the local array based on the chosen pivot. Next, compute the total number of elements less than the pivot and equal to the pivot across all MPI processes using MPI assets. Then broadcasts the global counts of elements to all MPI processes. Finally, we adjust the value of k and updates the local array based on the global counts.

3 Results

The program was ran on the university's HPC 'Aristotelis'. Aristotelis HPC runs with timed cron jobs via sbcast shell scripts loading distributed firmware regarding the program. Specifically all tests were made in the appropriately engineered 'rome' partition of our HPC.

The test them selfs are written on the README markdown of the GitHub repo found in the Source Code section The execution times and speedups of my implementations are detailed with respect to the number of items n, and the number of processes p.

4 Conclusion

In this project, we successfully implemented a distributed k-select algorithm using MPI, addressing the challenge of finding the kth smallest value in an array without explicitly sorting it. The MPI library facilitated communication and coordination among processes, ensuring efficient decision-making during the search process. We explored the option of utilizing multiple threads in each process to paralleling partitioning, taking advantage of shared memory on multi core processors. In conclusion, our implementations demonstrated notable performance gains, with execution times and speedup documented in the provided report. We considered various scenarios by adjusting the number of items (n) and processes (p) and provided insights into the impact of using different host servers. The source code is made available on GitHub, ensuring transparency and reproducibility.

The validity and effectiveness of our code were thoroughly discussed, taking into account the intricacies of MPI behavior in different distributed environments.

This project not only showcases our technical skills in parallel and distributed computing but also emphasizes the importance of optimizing algorithms for large-scale data processing. The lessons learned from this project contribute to our understanding of the challenges and opportunities presented by distributed computing systems.

5 Source Code

The GitHub open Repo containing the code can be found here.