Quick Select Algorithm through OpenMPI

Aimilia Palaska 10453, aimiliapm@ece.auth.gr

Introduction and Problem Description

The quick select algorithm is a variation of the quicksort algorithm designed to efficiently find the k-th smallest (or largest) element in an unordered list. It works by selecting a pivot element, partitioning the list into elements smaller and larger than the pivot, and recursively focusing on the partition containing the desired k-th element. In a parallel implementation using OpenMPI, the algorithm's efficiency is enhanced by distributing the workload across multiple computing nodes, thereby reducing the overall computation time for large datasets.

The code can be reviewed at either one of those links: GitHub and Google Drive

Input Handling & Data Distribution

As per instructions, the unsigned integers array was extracted from the html dumps of wikipedia, found here. The resulting assemblage of integers serves as a large enough data array to represent memory that cannot be handled by an ordinary computer, hence the remote execution on Aristotle University's HPC.

There were a number of approaches regarding the extraction of the data. The *getData()* function, which was deemed the best overall, was developed in order for the processes to acquire their corresponding portion of the data locally. It requires downloading the file in the same directory and including its name inside of the function. Additionally, a function provided by fellow students, encompassing the curl library for data retrieval in each iteration, was used but deteriorated notably the execution time, especially for the 16GB file. Lastly, an alternative input approach was explored in the file *input.c*, for the purposes of experimenting with the *MPI_Scatter()* command, although it was exclusively employed for local testing and debugging.

Algorithm

The nature of the original Quick Select Algorithm is recursive. However, due to challenges associated with synchronizing all processes in a coherent manner, an alternative strategy was employed. This method mimics the recursive essence through the utilization of a pseudo-infinite while loop and two pointers that delimit the accessible data range.

Specifically, each process initializes a two-integer array named "pointers," serving as dual pointers positioned at either end of the local data. As the name suggests, this array designates the "usable" segment of the array, denoting the region wherein the k-th element is located. After a pivot is selected, all processes perform the *partitionInPlace()* algorithm, as defined by the provided pdf.

Each process identifies a breakpoint, signifying a position beyond which all elements in the local array are greater than or equal to the pivot. The communicated breakpoints are transmitted to the master process through OpenMPI logic, which then sums them and determines if the pivot coincides with the k-th element. This results in three possible scenarios:

- the k-th element is identified, prompting the broadcast of a termination signal to all processes, forcing them to exit the while loop and display the result
- the position of the pivot is smaller than k, leading to the iteration of the while loop solely for the right segment (i.e. after the breakpoint) of each local array
- the position of the pivot is greater than k, leading to the iteration of the while loop solely for the left segment (i.e. before the breakpoint) of each local array.

It is important to note that following each parsing iteration, the last pivot is removed from the available portion of the array. This deliberate removal guarantees that at each step, the available data is reduced by at least one element, effectively preventing the occurrence of infinite loops. Additionally, this practice addresses potential challenges posed by repeated values, such as an array consisting exclusively of ones. While acknowledging that this approach may not be optimal for handling such exceptions, it nevertheless achieves successful termination, taking in consideration the improbability of such an occurrence.

Another noteworthy concern was the pivot selection at each iteration. Initially, the concept involved the master process assigning a new pivot; however, it became apparent that scenarios exist where the k-th element is not included within the master's data segment. Consequently, after a certain number of repetitions in the while loop, the master exhausted its elements for pivot selection without access to the data held by other processes.

To address this challenge, the solution involved transmitting the pointers array to the master process. The master then assigned a process with available data as the pivot sender, which assumed the responsibility of assigning and broadcasting the new pivot to every process.

Lastly, a counter was used to determine the number of steps performed by the algorithm. Beyond serving as a testing mechanism to evaluate performance for varying number of processes, the counter also played an important role in ensuring the termination of the while loop if it iterated a number of times equal to the length of the original array.

Use of Barrier

In OpenMPI, the integration of barriers, denoted by the *MPI_Barrier()* command, serves to enforce a synchronization point where each process awaits the others. This strategic use of a barrier enhances coordination and safeguards the integrity of the computational outcomes. However, it is imperative to acknowledge its inherent drawback, namely, its potential impact on program complexity and execution time. Consequently, it must be used when absolutely necessary.

Within the context of implementing the Quick Select algorithm through OpenMPI, Barriers were strategically employed at specific instances:

- after the data distribution, to ensure that all processes have their corresponding data segment in order to begin partitions
- at the start of every while iteration, to ensure synchronization
- prior to the broadcast of the termination signal, so that every process exits the loop
- before and after assigning a new pivot, to avoid false results from mismatching pivots across processes

Time Complexity

Concerning the assessment of time complexity, the algorithm's worst-case scenario is considered, specifically when confronted with an array comprising repeated values, and where the parameter 'k' aligns with the array's length. In this particular scenario, the

algorithm systematically restricts one value at each iteration until all values are exhausted, leading to the identification of the k-th smallest element.

It becomes evident, therefore, that Quick Select exhibits a linear time complexity in this worst-case scenario, as the number of steps required never exceeds the size of the given dataset. Moreover, in a more general case involving a varied values array, the average time complexity is marginally more favorable. This is especially notable when the pivot selected during each iteration eliminates on average half of the remaining elements.

Remote execution and Tests

The algorithm was tested remotely on Aristotle's HPC and the results regarding the execution times can be found here. Unfortunately, the large dataset of 15gb never terminated, either by a fault in the algorithm or unsuccessful data reception. Additionally, it became evident, when choosing random k values, that the algorithm is very sensitive to this random choice, meaning for certain ks the execution time is minimal.

Sources and Citations

- ➔ Documentation for OpenMPI
- ➔ The ChatGPT 3.5 language model was employed for the purposes of text formatting and the generation of certain portions of the code.

Please do not hesitate to contact me in the event of any nonfunctional links or if inquiries arise regarding the implementations. I will be happy to help.