Parallel and Distributed Computer Systems Second Project

Ιωάννης Παναγιώτης Μουτεβελίδης - AEM: 10442

INTRODUCTION

The present project implements the K-Select algorithm, a program which computes the the k_{th} smallest value of an unsorted array. The current program is being analyzed, both from a sequential and distributed aspect. The C++ programming language was utilized for the algorithm development, taking advantage of its support for task distribution techniques. The project was implemented in OpenMPI.

The project was developed using an Intel(R) Core(TM) i7-4765T CPU quad-core processor with a base clock speed of 2.00 GHz. This means that when tested on local machine, the CPU could use up to 4 processes maximum.

The algorithm has also been tested in HPC Cluster "Aristotle", which supports more processes per run. In this way, we conducted tests on larger files that exceed the capacity of a single computer, taking advantage of the nodes and the cores provided.

Each implementation, including the data of each diagram, is located in the Project_2 folder on Github at the link below:

Github:
https://github.com/jomout/Parallel_Distributed_Systems

THE PROBLEM

The K-Select algorithm's purpose is to find the k_{th} smallest element of an array without sorting it. It is similar to QuickSort algorithm, from the partition's aspect. The difference lies after the partition method.

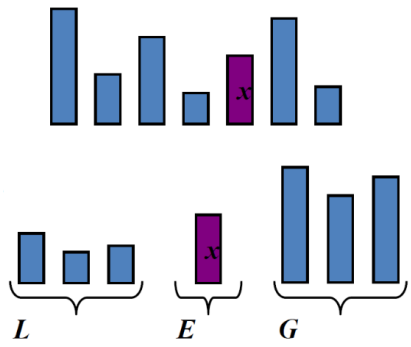

Figure 1. K-Select Partition

Although QuickSort tries recursively to sort the array by dividing it into sub-arrays, the K-Select recurs only for the sub-array that contains the k_{th} smallest element.

The program randomly selects a number from the given array, which serves as the pivot of the partitioning. It creates two sub-arrays with the one containing the elements smaller than the pivot and the other containing the larger ones.

In this semi-sorted array, the pivot is located between the former sub-arrays, meaning that this number is in its correct position, in the sorted array. If the pivot's position p is bigger than k, the algorithm recurs for the sub-array with the smaller elements, or for the sub-array with the larger ones otherwise. If k is equal to the corresponding p, we conclude that the k_{th} smallest element of the array is the current pivot.

I. ALGORITHM PRESENTATION

In this project, one C++ script is being utilized, $KSelect.cpp$. The algorithm is Zero-based, meaning that for the first smallest element, we declare k = 0, etc.

A. K-Select

The K-Select algorithm uses two functions:

1) `partitioninplace()`: As mentioned above, the process begins similarly, like the QuickSort algorithm. We call the `partitioninplace()` method, which takes the original array, the pointers $left$ and $right$ indicating the elements at the beginning and end of the current sub-array, as well as, the pivot that we are currently partitioning for. The pointers $left$ and $right$, actually, determine the sub-array, of the original array, we want to search in.

We are iterating through the elements at the beginning of the sub-array by using i as pointer, for elements smaller than the chosen pivot. If an element is equal to the pivot, the algorithm moves it to the start of the sub-array and continues along. This continues, until an element greater than the pivot appears.

We are iterating, consequently, through the elements at the end of the sub-array by using j as pointer, for elements larger than the chosen pivot. If an element is equal to the pivot, we move it to the end of the sub-array and continue our search, until we meet an element smaller than the pivot.

When the two processes above finish, we simply swap the two former elements, at which each iteration stopped. We continue the same process until the desired element is found.

The two processes above stop until they cross each other. When that happens, the two pointers i and j start swapping the elements, they are pointing, with the "equal" items that

were stored at the beginning and the end of the sub-array. In this way, when the method process terminates, the elements equal to the pivot are being stored in the middle of the sub-array, on the left-hand side there are unsorted elements smaller than the pivot, and on the right-hand side, elements greater than the pivot.

This function returns two pointers, p and q as a `Pointers struct`, where:

- p points at the first element equal to the chosen pivot.
- q points at the first element greater than the pivot.

2) `kselect():` This is the function of the K-Select algorithm. The function starts by partitioning the current array, calling the `partitioninplace()`. The pivot is chosen randomly from the elements of the sub-array we are currently working on.

The `partitioninplace()` function returns two pointers p and q, each of them pointing to the correct elements.

The `kselect()` function has to decide the sub-array that the recursion in going to happen.

- If $k \in [p, q)$, then function has found the k_{th} smallest value, which is equal to the pivot.
- If $k < p$, then the function recurs itself for the left-hand side sub-array.
- If $k \geq q$ then the function recurs itself for the right-hand side sub-array.

II. Data Arrays Information

The execution time of the program is displayed in the following diagrams.

Each one of the diagrams contains the average time values of three code executions in microseconds. The Data Arrays used in the diagrams:

1) https://dumps.wikimedia.org/other/static_html_dumps/current/el/wikipedia-el-html.tar.7z
   - Size: 107 Mb, n = 28,084,359 elements
   - Results:
     - k = 0: Answer = 0
     - k = 14,042,179 (n/2): Answer = 2147639940
     - k = 28,084,358 (n): Answer = 4294967175
2) https://dumps.wikimedia.org/other/static_html_dumps/current/en/wikipedia-en-html.tar.7z
   - Size 14.3 GB, n = 3,840,885,803 elements
   - Results:
     - k = 0: Answer = 0
     - k = 1,920,442,901 (n/2): Answer = 2147451815
     - k = 3,840,885,802 (n): Answer = 4294967294

III. Open-MPI

The Open-MPI version of the program is implemented in $MPI\_KSelect.cpp$ and conducted as followed:

A. Data Download

1) `getfile():` The algorithm starts by fetching a testing array from the local drive. By determining the file path, each process gathers a data chunk from the chosen data array. This distributed data gathering is triggered by the `MPI_File_set_view()` and `MPI_File_read_all()` commands. The array is being divided according to the initialized processes. In the end, the processes have gathered their partitions of the data array and are set to begin the search.

B. K-Select

After gathering the testing array, it is time for the algorithm to search though the data and return the k_{th} smallest element.

At first, we declare the `MPI_Init()` command, which initializes the processes.

After each process got its chunk of the data array, the `MPI_Barrier()` synchronizes the processes, right before the search begins.

Each process enters in a `while` loop. In this loop, the master process chooses a random number between the minimum and maximum `uint32_t` value possible and broadcasts it to the rest of the processes. This value is the pivot, according to which, each process will partition its chunk of data. The master process utilizes the `MPI_Reduce()` function to collect the pointers of each process, which indicate how many elements are equal to the pivot or less, as p points to the first element equal to the pivot, and q points to the first element greater than pivot, in the supposed sorted array.

The master process decides the actions of the program:

- If $k \in [p, q)$, then the master process broadcasts a termination signal `STOP`. The program has found the value at k_{th} position, which is equal to the current pivot.
- If $k < p$ then the master process broadcasts a signal `LEFT` which requests that the processes search to their left-hand side sub-arrays. The range of the possible values of the pivot is narrowed accordingly.
- If $k \geq q$ then the master process broadcasts a signal `RIGHT` which requests that the processes search to their right-hand side sub-arrays. The range of the possible values of the pivot is narrowed accordingly.

If a process determines that all its assigned elements are either greater or smaller than the current pivot, then it stops computing and gives the same pointers p and q, as outputs to the master process accordingly.

IV. Diagrams

The diagrams, below, are representing the execution time of the program, for the above data arrays, depending on the number of processes p used and the number of items m of the data array. For the number of processes p:

$$p = n \cdot c$$

where n: nodes, c: CPUs per node

Of course, there will be differences between execution times using the same number of processes, but different number of nodes and CPUs per node.

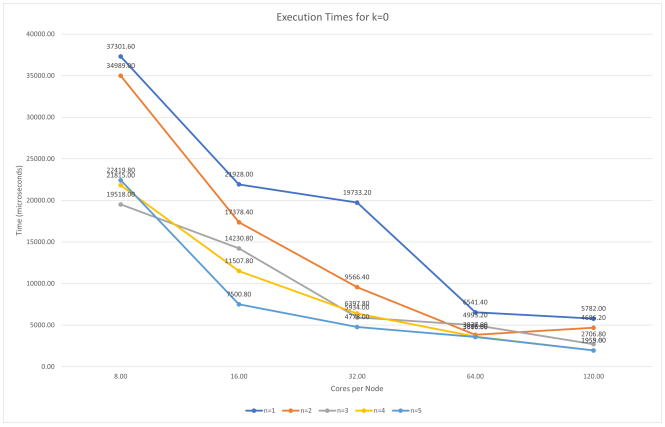

Figure 2. Time Executions for k = 0 - 107 Mb array

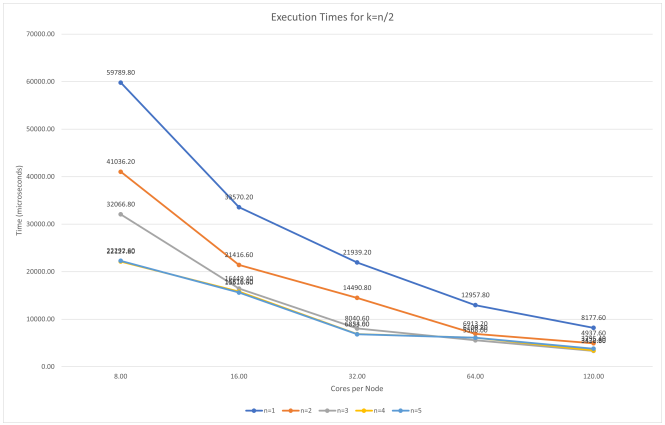

Figure 3. Time Executions for k = 14,042,179 (n/2) - 107 Mb array

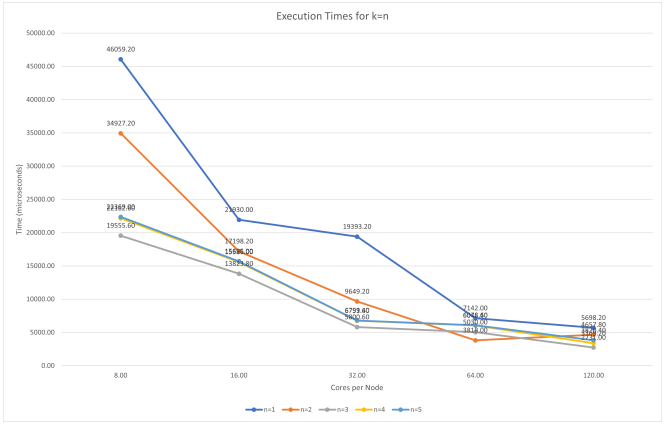

Figure 4. Time Executions for k = 28,084,358 (n) - 107 Mb array

The Figures 1, 2 and 3 depict the impact of the number of CPUs per Node used during the execution of the K-select algorithm for the 107 Mb data array for different positions of k inside the array. As we see, the distributed work load over the processes slightly reduces the execution time. Because the array is relatively small, there is not any such difference as the nodes increase. The position of the k that we search for slightly increases the computation time as it converges to the center of the array.

The best performance, as it can be seen from the diagrams, is displayed when 5 nodes and 120 CPUs per node are being used. The variations in time computations lack significance, given the increasing number of nodes and CPUs per Node. In other words, we allocate a considerable amount of resources without yielding any meaningful outcomes.

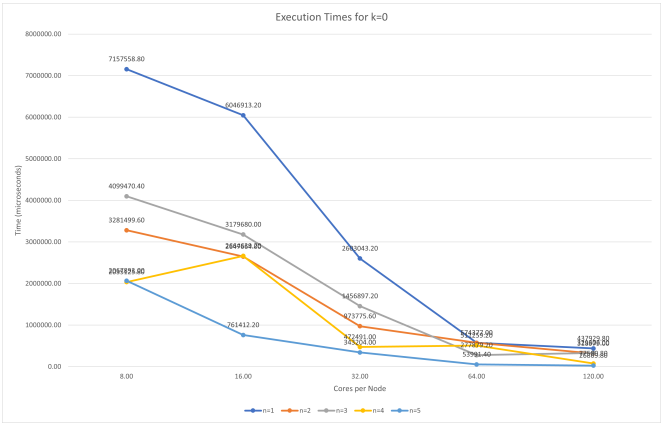

Figure 5. Time Executions for k = 0 - 14.3 GB array

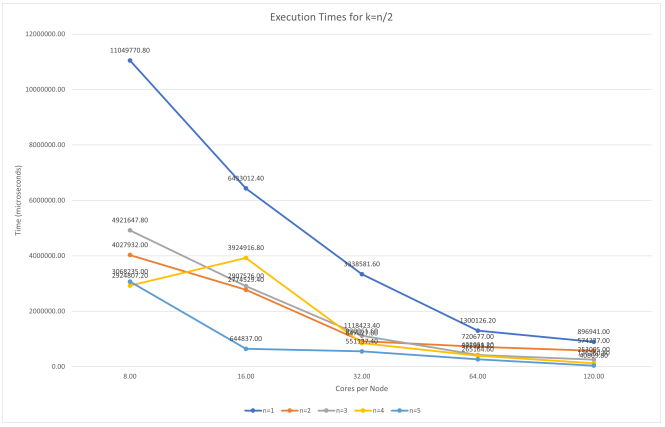

Figure 6. Time Executions for k = 1,920,442,901 (n/2) - 14.3 GB array

The Figures 4, 5 and 6 depict the impact of the number of CPUs per Node used during the execution of the K-select algorithm for the 14.3 GB data array for different positions of k inside the array. The distributed work over the processes, not only makes the execution of the program with really large arrays possible, but also reduces the computation time. Through the allocation of a substantial amount of resources, we achieve program execution in a matter of microseconds. In contrast, when working on only one node, the execution times extend to several seconds.

The position of the k that we search for, has a significant role over the execution time. As it converges to the center

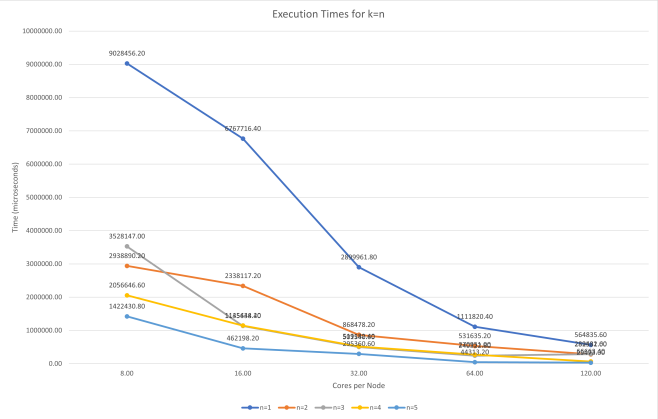

Figure 7. Time Executions for k = 3,840,885,802 (n) - 14.3 GB array

of the array, the computation time increases significantly and nearly doubles in some cases.

V. Results - Conclusion

In the diagrams above, we measure different execution times with respect to the number of nodes n, the number of CPUs per node used c and the number of total processes p, in order to visualize the effect of each factor to the final result.

The increase of the nodes utilized, actually enhanced the performance of the program. Although in many cases the processes used stay the same, when working on fewer nodes and more CPUs per node, the algorithm executes faster. This implies that OpenMPI has undergone optimization for scenarios where processes originate from fewer nodes but involve more CPUs per node.

For instance, in the Diagram on Figure 4, there are different execution times when using the same processes:

Table I
Execution Times in microseconds

1 node-64 CPUs/node	2 nodes-32 CPUs/node	4 nodes-16 CPUs/node
574377	973775.6	2664639.2

This observation appears for larger total number of processes (64), as for smaller ones,16 processes, for instance, the opposite happens.

The number of k, also, has a significant role in time execution. As the parameter k converges towards being positioned at the center of the array, there is a notable increase in execution time, often approaching a near doubling in certain instances.

Throughout the different implementations, we can safely assume that utilizing more processes, reduces the execution time to a certain level, depending on the scale of the problem. For the smaller data arrays, like 107 Mb, in cases where more nodes and processes are allocated, the execution times were roughly the same.

To sum up, when determining the number of processes, it is crucial to consider the problem's scale. Allocating substantial amount of resources is not the best solution, as comparable results can be achieved with less computational power.

References

[1] https://www.geeksforgeeks.org/quickselect-algorithm/
[2] https://www.freecodecamp.org/news/quickselect-algorithm-explained-with-examples/
[3] https://en.wikipedia.org/wiki/Quickselect
[4] https://github.com/xetqL/parmedian
[5] https://rookiehpc.org/
[6] https://www.open-mpi.org/