Uppsala Universitet

Parallel and Distributed Programming 1TD070

Assignment 3

Agelii, Carl.

May 17, 2024

Parallelization

The quick-sort algorithm is a sorting scheme that uses a divide and conquer strategy. This naturally makes it susceptible parallelization. In this project, we use MPI and the C language to achieve the quick-sort algorithm and empirically gather results regarding execution times.
First off, a input file is read on the root process and the scattered to all other processes. Since the program should be independent of n (the amount of numbers to be sorted), I used the function MPI_Scatterv. This is because n is not necessarily divisible by the number of processes. Next, each process sorted their assigned part of the array locally. This was done using the function qsort. Then, the global sorting scheme started according to the quicksort algorithm: a pivot element was chosen for each communicator, and then two sub-arrays were allocated on each process which then were filled with numbers bigger and smaller than the pivot, respectively. After that, the current communicator was split in two and one group got all values smaller than the pivot and the other got all values bigger. After that, all processes merged the two arrays into one sorted array. This process went on recursively until each communicator only consisted of one process. Lastly, the result was gathered on the root process, were I reused the allocated input array. The timing was taken on the actual sorting only, not the reading of input and output.
Three different strategies for selecting pivot element was tested:

1. The median of one process for a whole communicator.
2. The median of all medians in a communicator.
3. The average of all medians in a communicator.

Experiments

When the program was working as expected, numerical experiments were conducted on Snowy. First off, the strong scalability of the program was tested. The problem size was fixed to $n = 125000000$, and the number of processes varied as $p = 1$, 2, 4 and 8. The pivot was selected to be 1.

Now, the weak scaling of the program was tested. This time, the problem size per process was fixed to 125000000 and the amount of processes was varied as for the strong case. The pivot was again set to 1.

Lastly, the impact of the pivot was experimented. Again, the input file was set to $n = 125000000$ and the pivot method was varied. Also, 8 processes were used in order to see the effect of the pivot method. After that, another input file was tested, but this time, it was sorted in descending order. This means that it is the opposite of the desired output.

Results

Table 1: Strong Scaling Times

Number of Processes	Execution Time (seconds)
1	26.04
2	13.51
4	7.09
8	3.92

Table 2: Weak Scaling Times

Number of Processes	Execution Time (seconds)
1	26.04
2	28.04
4	30.96
8	36.38

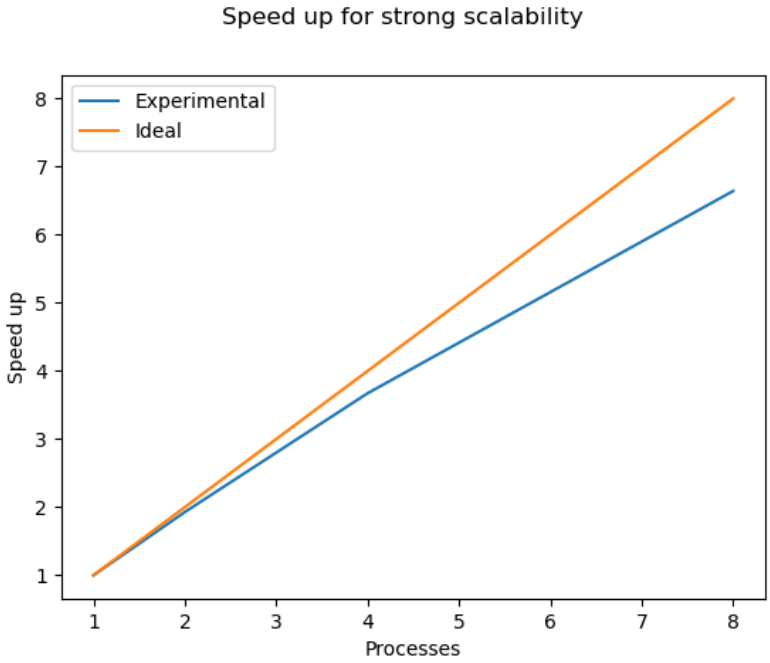

Figure 1: Speed up for strong scalability

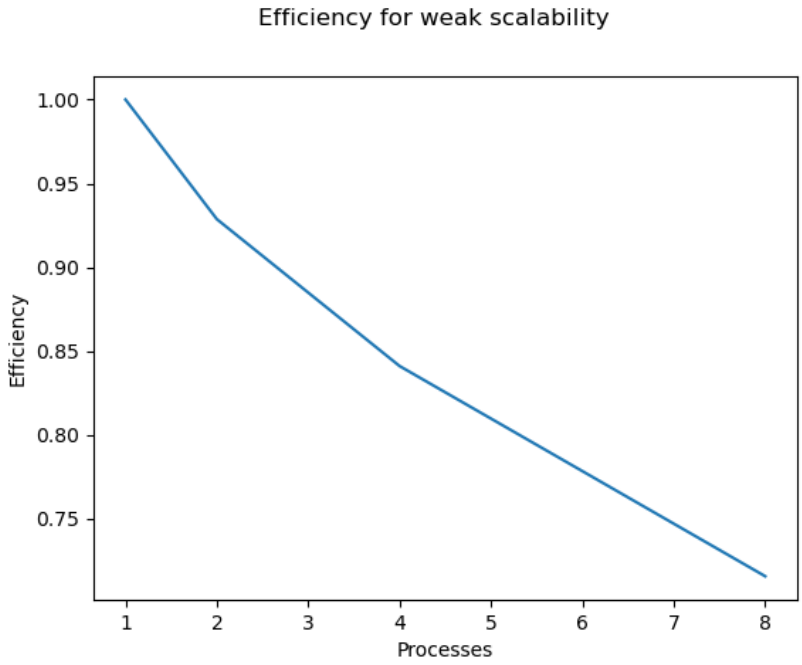

Figure 2: Efficiency for weak scalability

Table 3: Pivot selection times

Number of Processes	**Execution Time (seconds)**
1	3.93
2	3.94
3	3.53

Table 4: Pivot selection times (backward)

Number of Processes	**Execution Time (seconds)**
1	2.03
2	1.98
3	1.59

Discussion

The parallelization seems to have worked fairly well. By looking at figure 1 and 2, there is a slight discrepancy between the experimental and ideal values. This is most definitely because of the communication between processes that is necessary in the algorithm. The processes need to be synchronized when choosing pivot, which can potentially slow down performance.By looking at table 3 and 4, there is a clear benefit in choosing pivot number 3. This is probably because it creates less load imbalance for the processes.