ASSIGNMENT 3:
Parallel Quicksort algorithm

Parallel and Distributed programming, 5 hp

Erik Bjerned, Oskar Holmes & Mattias Persson

2023-04-17

UPPSALA
UNIVERSITET

Contents

1 Introduction

The task as presented is a parallel implementation of a quicksort algorithm using MPI in C.

2 Theory & Implementation

Quicksort is an efficient sorting algorithm witch utilizes pivot elements to quickly "divide and conquer". This description comes from the fact that the array that is to be sorted is divided over and over and over, based on pivot elements. The distribution between sub arrays is decided by a pivot element, where all elements larger than the pivot element are placed in one array and all elements smaller are placed in the other. Each smaller "subarray" is then sorted by itself. The pivot element can be chosen in a variety of ways and is usually the signifier in different implementations' performance. Worth noting is that quicksort is an *in-place* algorithm. This means that no auxiliary memory elements are used by it. All elements are moved in the memory element that is getting sorted. The algorithm has an average performance of $O(n\ log\ n)$.

The implementation of an ordinary quicksort algorithm is relatively simple. In this assignment however, the quicksort algorithm is implemented recursively over several groups of PE:s, which increases complexity. In practice, every call of the quicksort program should split the active array into two sub arrays and distribute them to two paired PE:s. The pivot element is in our case decided using a switch case, where one of three different strategies can be selected. The three pivot selection strategies that we were required to implement were as follows:

1. The pivot is the median of one processor in each group of processors
2. The pivot is the median of all medians in each group of processors
3. The pivot is the mean of the medians of all processors in each group of processors

3 Performance tests

3.1 Method

The measure the performance of this implementation a few tests were performed. The strong scaling was tested on $N = 10^9$ sized list and the weak scaling was tested on $1.25\cdot 10^8$, $2.5\cdot 10^8$, $5\cdot 10^8$, $1\cdot 10^9$ respectively. Further, the three different pivot strategies were tested for each case. Note that the performance of a single PE was not recorded since it will only be sorted by the built in `qsort()` which makes the results difficult to compare.

3.2 Strong scaling

Table 1: Pivot strategy 1

PEs	Execution time(s)
2	9.243501
4	9.110695
8	7.223739
16	5.933748
32	4.009466

Table 2: Pivot strategy 2

PEs	Execution time(s)
2	9.125843
4	8.954389
8	7.211948
16	5.603345
32	4.079232

Table 3: Pivot strategy 3

PEs	Execution time(s)
2	9.196635
4	9.005620
8	7.184509
16	5.533276
32	4.024808

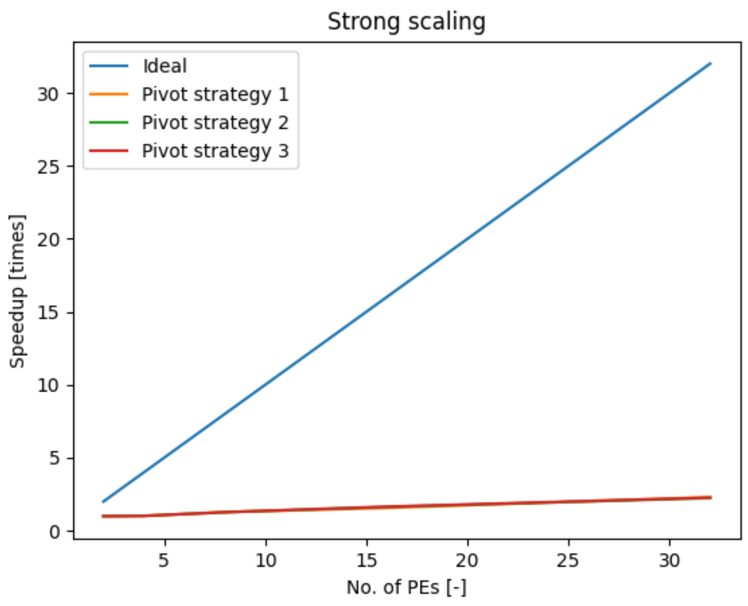

Figure 1: Strong scaling

3.3 Weak scaling

Table 4: Pivot strategy 1

PEs	Execution time(s)
2	1.179004
4	2.274950
8	3.605316
16	5.933748

Table 5: Pivot strategy 2

PEs	Execution time(s)
2	1.140971
4	2.227817
8	3.573983
16	5.603345

Table 6: Pivot strategy 3

PEs	Execution time(s)
2	1.135737
4	2.245100
8	3.556425
16	5.533276

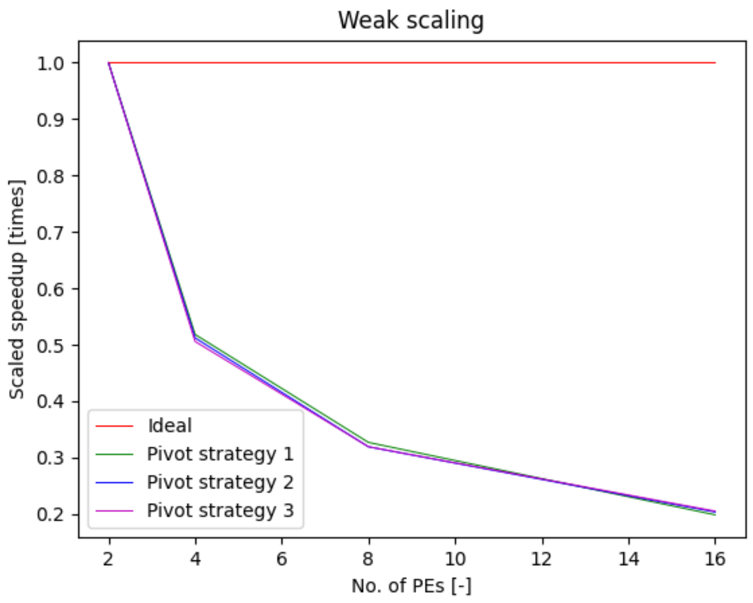

Figure 2: Weak scaling

4 Discussion

Regarding scaling, this implementation seems to not yield the results that we hoped for, both in the strong and weak sense, and all pivot strategies yield more or less the same result. This is surprising since the pivot strategies partition the data in ways that for some initial array setup can yield much different workload division. There might be something wrong in the setup of the testing script, which has happened before.