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 $N = 1.25{\cdot}10^8$ sized list and the weak scaling was tested on $1.25{\cdot}10^8, 2.5{\cdot}10^8, 5{\cdot}10^8, 1{\cdot}10^9, 2 \cdot 10^9$ respectively. Further, the three different pivot strategies were tested for each case.

3.2 Strong scaling

Table 1: Execution time $[s]$ for strong scaling with random order for $n = 1.25 \cdot 10^8$

	Pivot strategy		
PE (n)	1	2	3
1	25.80	25.93	25.59
2	13.84	13.74	13.73
4	6.81	7.76	7.78
8	3.66	5.05	5.06
16	3.86	3.86	3.88
32	2.62	2.12	3.20

Table 2: Execution time $[s]$ for strong scaling with backwards order for $n = 1.25 \cdot 10^8$

	Pivot strategy		
PE (n)	1	2	3
1	8.72	8.79	8.79
2	4.87	5.08	4.81
4	2.92	2.90	2.91
8	2.12	2.10	2.10
16	1.83	1.81	1.82
32	1.58	1.58	1.58

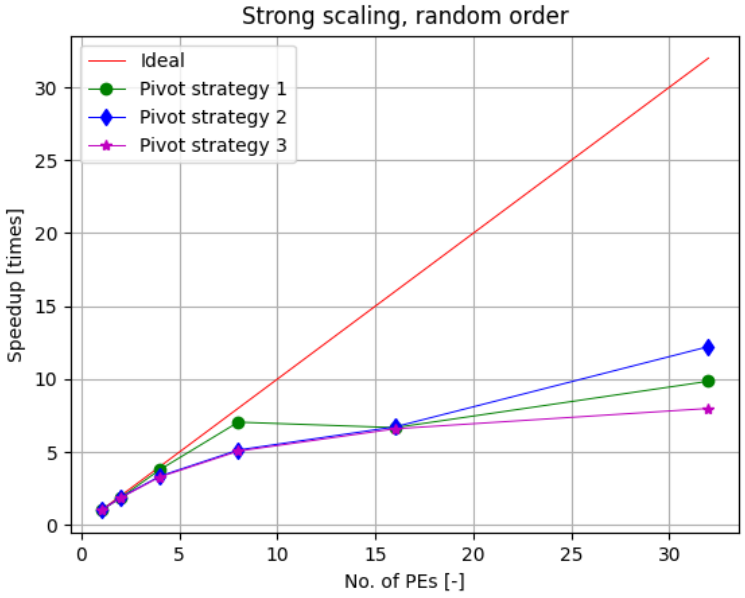

Figure 1: Strong scaling, random ordered input

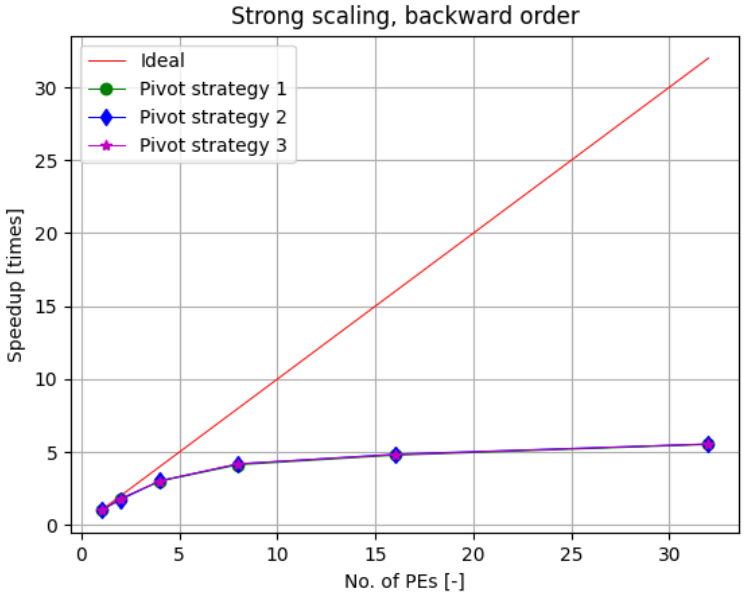

Figure 2: Strong scaling, backwards ordered input

3.3 Weak scaling

Table 3: Execution time $[s]$ for weak scaling

	Pivot strategy		
PE (n)	1	2	3
1 $(1.25 \cdot 10^8)$	25.93	25.91	25.77
2 $(2.50 \cdot 10^8)$	27.27	28.45	28.52
4 $(5.00 \cdot 10^8)$	32.82	32.79	32.73
8 $(1.00 \cdot 10^9)$	35.20	43.45	43.35
16 $(2.00 \cdot 10^9)$	39.90	66.47	62.01

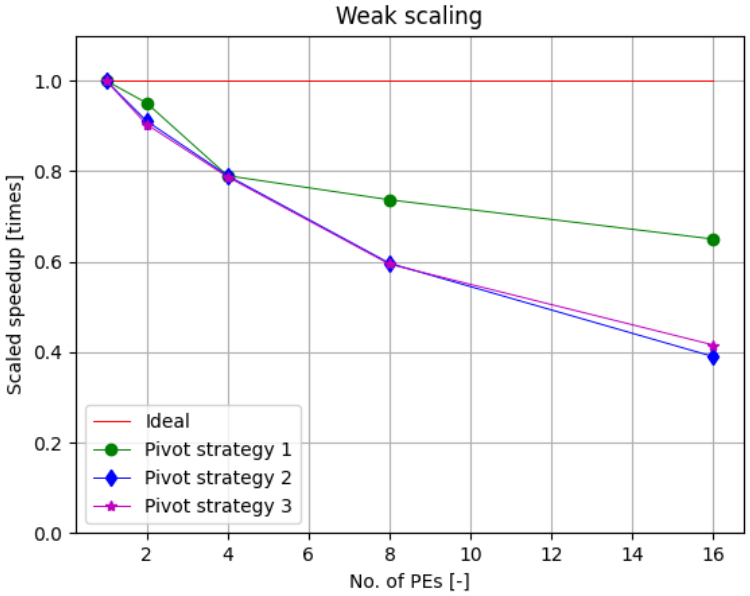

Figure 3: Weak scaling, random ordered input

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.

4.1 After revision

The difference between the first version and the revised version are quite significant in a few different ways. First of there was a misunderstanding in how the programme should be timed, in the previous version only the parallel communications were timed thus excluding `qsort()` and other parts of the program. Now the time is measured after the input has been distributed until before gathering the results from the processes. Changing this enables us to compare the results against the performance of a single process.

Another change that was made that the original program used values of type `double` it is now changed to `int` halving the memory usage and reducing time spent accessing memory. The

datatype was not explicitly told, but since the check script does not allow the output to have any decimals it was assumed that `int` would be sufficient.

To further improve performance the program was profiled using Allinea MAP. The largest entry to the overall execution time is reading the input. This was looked at to reduce the necessary time to perform tests, but no obvious approach to reading text files in parallel was found and was not entirely necessary to change. The second entry was calling `qsort()`, and since this now was included in the measurements this was an area of improvement. This was improved by reducing `compare()` from a collection of `if`-statements to a single `return`-statement. A profile of a recursive function as `Parallel_Qsort()` is, can be somewhat unclear but we were able to identify two areas of improvements, the merge function and memory allocation. The merge function was reduced in complexity as is was not longer having to solve a bug introduced in the data input. Further the arrays storing the incoming data were reused and not allocated and freed for every communication call, instead it was reallocated to fit the incoming data. However here there is still some improvement to be made as it not always necessary to reallocate the data, it should only be done if the incoming data is large than previously and then it should allocate some buffer to account for small changes.

To comment on the new results, they show a better scaling than seen in the previous version. The run time is however longer caused by the change in placement of the time measurements. In figure 1 there is a clear change in scaling at 16 processes. Using timings of the parallel section, it is concluding that it is the limiting factor when there are 16 or more processes as the serial code per process reduces execution time with smaller segments of the initial input, whilst the parallel segment necessarily has the same rate of decrease. In the case of the strong scaling with backwards order, a different behaviour is seen. Do note that the execution time initially is several times smaller than with random order which may play a part in this. Because of the lower overall time, the parallel segment will have more effect from the start, thus seemingly have a larger impact on the scaling tests.

The difference in computational time for random or backwards ordering can be explained by the fact that a backwards order initially needs to swap all values, but then get 2 evenly sized arrays to continue to sort, a random order gives you a more difficult time to get two evenly almost sorted arrays to continue to sort.

The difference in pivot-strategy is in some cases neglible, as in the case of reverse order. But when the input array is completely randomly ordered strategy 1 seems to have an edge on the other two with a few exceptions. Strategy 1 is the most intuitive approach since you focus on a single group of processors all with similar values. Pivot strategy 2 performs better at 32 PEs in the strong-scaling tests, this probably comes from the fact that the arrays will be split many times and the strategy takes more data into account.

Overall, the pivot strategy can have a large impact, but if knowledge of the array is limited beforehand, strategy 1 is probably the choice with the best result.