Assignment 3 - The Parallel Quicksort algorithm

Emy Engström
Jacob Kovaleff Malmenstedt
Viveka Olsson

May 2023

UPPSALA
UNIVERSITET

1 Problem description

The problem to be implemented is; given a sequence of n integers, use a parallel quicksort algorithm in C using MPI to sort the elements in the sequence. The performance is evaluated on UPPMAX compute cluster for different sizes of sequences and number of threads and cores.

The outline of the algorithm used is as follows:

1. Divide the data into p equal parts, one per process
2. Sort the data locally for each process
3. Perform global sort
   - 3.1 Select pivot element within each process set
   - 3.2 Locally in each process, divide the data into two sets according to the pivot (smaller or larger)
   - 3.3 Split the processes into two groups and exchange data pairwise between them so that all processes in one group get data less than the pivot and the others get data larger than the pivot.
   - 3.4 Merge the two sets of numbers in each process into one sorted list
4. Repeat 3.1 - 3.4 recursively for each half until each group consists of one single process.

The program takes three input arguments; the input file name, an output file name as well as a number specifying which pivot strategy to use. The different pivot strategies are to either select the median in one processor in each group of processors, to select the median of all medians in each processor group or to select the mean value of all medians in each processor group. An assumption can be made that the number of processes is equal to 2^k for an integer k. The time for the program is measured after the input is read and before the resulting sorting list is written into the output file.

2 Implementation

The code begins with initializing the necessary MPI variables. The input sequence and the number of values in the unsorted array is read. After the length of the data is broadcast to all the processes using *MPI_Bcast*, the array is now added to the *Quicksort* function.

The *Quicksort* function handles all distributing and collection of the data in the very first and very last moments of the sorting. It begins by allocating local lists of the same length as the original list for all processors. This might seem wasteful but in the worst case all elements might get placed in the same local list during the sorting so it is necessary to allocate this much memory. If the number of processes do not evenly divide the data the number of elements per process is increased by one on the first couple processes until the entire list has been divided. The function uses *MPI_Scatterv* to distribute the data across all the threads according to the calculated distributions. The local lists are now sent into the *qsort* function. This function sorts every local list before these then are sent to the inner part of the recursive quicksort algorithm, *QuicksortInner*.

This function checks first of all if the algorithm is at the last recursive step, the base case. If not, it finds the pivot element using the *pivot_selection* function, which finds the element using the specific strategy chosen as one of the input argument to the program. The *pivot_selection* function also broadcasts the pivot to all processors on the same communicator so that everyone agrees on the pivot. Afterwards the data is then partitioned so that in each processor the list is divided into elements larger than the pivot to the right of the list and elements smaller than the pivot to the left of the list.

Next, the program swaps the data between the processors depending on the pivot elements, i.e the left half of the processors sends data larger than the pivot element to the right half of the processors, and vice versa. To do this, first all the processors find their respective partners and sends the information about how many elements to be sent and received between each other, using a *MPI_Sendrecv*-command. Next, memory for the buffers used to send and receive the elements are allocated and the data is copied into these using *memcpy*. The elements can now be sent and received so that each processor has one piece of the original list and one piece of its partner processors list. These two lists are then merged into one continuous list and the length of this list is updated. Lastly, the *QuicksortInner* function is called again to continue with the recursive process until the base case is reached.

Once the base case is reached the local lists return to the outer *Quicksort* function. Here, all the data from each process is gathered into a final, now sorted list, and the local lists can be freed. The code then returns to the main function where the timings are stopped and process one prints the maximum time taken and writes the sorted list to the output file.

3 Description of numerical experiments

Numerical experiments were executed to determine and evaluate the weak and strong scaling. For weak scalability, all three methods of choosing the pivot element was run using 125M elements for 1 processor, 250M elements for 2 processors and 500M elements for 4 processors. For strong scalability the three methods has been tested for 125M elements and 250M elements respectively with 1, 2, 4 and 8 processes. The speedup is also presented in plots. All timings presented in this report has been executed on UPPMAX.

4 Results

4.1 Weak scaling

The resulting time measurements for the weak scalability performance experiments can be seen in table 1, 2 and 3, when comparing results for 125M elements for 1 processor, 250M elements using 2 processors and 500M with 4 processors.

Table 1: Running times for weak scalability, for pivot method 1

N.o processors	Number of elements	Running time[s]
1	125M	26.896454
2	250M	30.607479
4	500M	42.837852

Table 2: Running times for weak scalability, for pivot method 2

N.o processors	Number of elements	Running time[s]
1	125M	26.598747
2	250M	31.845017
4	500M	42.880150

Table 3: Running times for weak scalability, for pivot method 3

N.o processors	Number of elements	Running time[s]
1	125M	26.700452
2	250M	31.307399
4	500M	42.346078

4.2 Strong scaling

The resulting time measurements for the strong scalability performance experiments using lists of 125M elements can be seen in tables 4, 5 and 6. Respective plots showing the speed up compared to a theoretical speed up can be seen in figure 4.

Table 4: Running times for strong scalability, using 125M elements and using pivot method 1

N.o processors	Running time[s]
1	26.896454
2	15.225178
4	8.867180
8	7.077638
16	10.809385

Table 5: Running times for strong scalability, using 125M elements and using pivot method 2

N.o processors	Running time[s]
1	26.598747
2	15.260120
4	10.343076
8	8.982718
16	10.502750

Table 6: Running times for strong scalability, using 125M elements and using pivot method 3

N.o processors	Running time[s]
1	26.700452
2	15.153933
4	10.196858
8	8.960490
16	9.682772

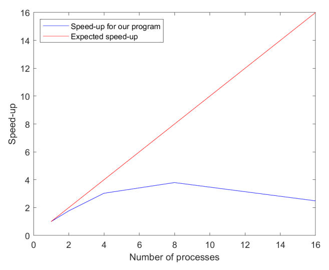

Figure 1: Pivot method 1

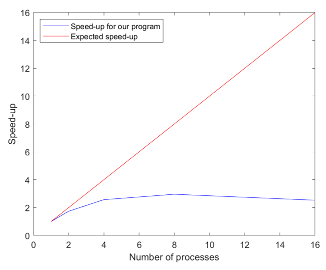

Figure 2: Pivot method 2

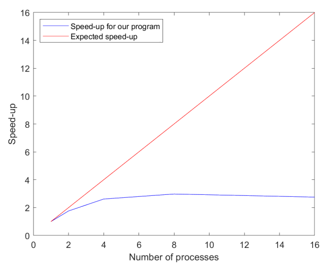

Figure 3: Pivot method 3

Figure 4: Strong scalability tests for 125M elements

The resulting time measurements for the strong scalability performance experiments using lists of 250M elements can be seen in tables 7, 8 and 9. Respective plots showing the speed up compared to a theoretical speed up can be seen in figure 8.

Table 7: Running times for strong scalability, using 250M elements and using pivot method 1

N.o processors	Running time[s]
1	55.925720
2	30.607479
4	18.729713
8	15.086099

Table 8: Running times for strong scalability, using 250M elements and using pivot method 2

N.o processors	Running time[s]
1	55.497260
2	31.845017
4	21.423265
8	18.106554

Table 9: Running times for strong scalability, using 250M elements and using pivot method 3

N.o processors	Running time[s]
1	55.677511
2	31.307399
4	21.117044
8	17.112509

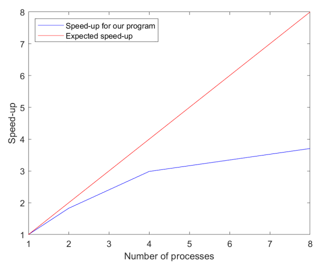

Figure 5: Pivot method 1

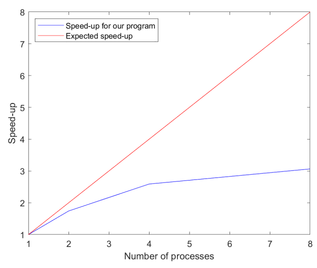

Figure 6: Pivot method 2

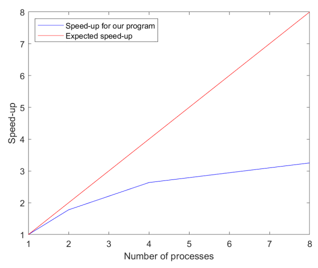

Figure 7: Pivot method 3

Figure 8: Strong scalability tests for 250M elements

5 Discussion

The resulting times for the weak scaling experiments behaves as expected. The few extra seconds added when increasing the input and processor size might be a result of the overhead created when calling on multiple processes. Over all, the weak scaling has given a satisfactory result. Also the result of the strong scaling tests is mainly expected. The strong scalability is decent, but still pretty far from optimal scaling and could be better. However, it is very normal for the actual result to differ a bit from the optimal which makes the speeup of this program fairly reasonable.

It can be noticed that the first pivot selection algorithm was generally fastest in our testing. This is probably because of the decreased need to compute multiple medians and communications that the other methods require. For a perfectly uniformly distributed random list the different pivot selection strategies will not improve performance much as each local list will have approximately the same distribution and therefore also approximately the same median. For lists with other distributions such as exponential or normal distribution this might not be the case and the increased overhead of pivot selection strategy 2 and 3 could be countered by the improved accuracy while partitioning the local lists. The communication between processes in the $pivot_selection$ function could probably be improved slightly to avoid having processes standing still while they wait for other processes to reach the same stage but in large part this is an unavoidable bottleneck.

We are satisfied with the performance of the program and how the parallel quicksort algorithm effectively and correctly sort all elements. The performance of the code is generally expecte