UPPSALA
UNIVERSITET

Assignment 3
Parallel Quick-sort

Written by:
Jakob Gölén
Simon Persson

May 15, 2022

1 Problem Description

Given a list of numbers, the goal of this assignment was to sort the numbers in ascending order and return a list with the sorted numbers. The algorithm for the parallel quicksort works in the following way:

First the data is separated as equally as possible between a chosen amount of processors which is assumed to be an exponent of two. Each local list is then sorted. Next, a pivot element is selected and the processes split into two groups. The elements in the local groups are then exchanged so that one group gets elements larger than the pivot element and the other group gets elements lower than the pivot element. The recieved elements are then merged with the elements already in the process that weren't sent and the process repeats recursively, with a pivot element being chosen per processing group. The process is repeated until every group only contains one processing element, which is $log_2(p)$ repetitions, where p is the number of processing element. When the process is complete, every processing element will contain a sorted list, and every element in process i is smaller than every element in process i+1. The lists can then be combined which leads to the final sorted list.

2 Implementation

2.1 Algorithm

The parallel quicksort algorithm was implemented with OpenMPI, using a program written in C. The program has three arguments: The first argument is the name of an input file containing N+1 elements, with the first element being how many elements should be sorted and the following N elements being the elements which should be sorted. The second argument is the name of an output file, where the N sorted elements should be written. The third argument is the pivoting strategy. Three different pivot strategies will be tested, numbered one to three.

1. The pivot point is chosen as the median of one process in the pairing groups.
2. The pivot is chosen as the median of all medians of each processor in the pairing groups.
3. The pivot is chosen as the mean values of all medians of each processor in the pairing groups.

The program then performs the parallel quicksort of the input file and writes the result into an output file.

2.2 Code implementation

First, the input name, the output name and the choice of pivot strategy is saved and MPI is initialized. The processes are laid out on a hypercube topology by using MPI_Cart_create and varying the number of dimensions based on the number of processing elements so that there are two processing elements per dimension, leading to a two dimensional Cartesian topology for four cores, a three dimensional Cartesian topology for eight cores and so on. The processing element that has rank 0 reads the input and

stores the unsorted values in an array. The number of values in the array are sent to every process with MPI_Bcast and the number of elements are divided equal between the processing elements. If the number of elements are not divisible by the number of processes, the remainder from an integer division is stored in the process with rank zero. The required memory is allocated and the elements are then scattered across the processing elements by using MPI_Scatterv and the initial input array is freed. A timer then starts.

Next, every processing element performs a local sort by calling the built in qsort function where after the number of iterations, i.e. the number of times the processing elements should split into smaller groups is calculated and a for-loop is initialized.

The processing elements are split into smaller groups with MPI_Comm_split as a function of the current iteration, so in the first iteration, the Cartesian topology is not split at all (i.e. split into one group), in the second iteration the topology is split into two groups, in the third iteration four groups and so one. The new sub ranks and group sizes are found.

Next the pivot element is calculated. If the first strategy is chosen, the median of the sublist in process 0 is found and sent to every other process in the group with MPI_Bcast. If one of the other strategies are found, every process in the group sends their median to process 0 which stores them in an array. If the second strategy is used, the array of medians is sorted and the median in the array is picked and sent to the other processes in the group. If the third strategy is used, the mean value of the medians in the array is found and sent to the other processes in the group.

With the pivoting element chosen, the pivot element in every local array is found. The processing groups are split into a lower and upper half. The processing elements picks a partner with its corresponding place in the other half of the group. This means that the lowest processing element in the lower half picks the lowest processing element in the upper half and so on, so if a group consists of eight processing elements, process 0 and 4 will be partners, process 1 and 5 will be partners and so on. A receive array is allocated in every processing element. Processes in the lower half sends the elements above and including the pivoting element in its local array to its partners receive array and processes in the upper half sends the elements below the pivoting element in its local array to its partners receive array. When the correct elements are sent, a merge array is allocated which will merge and sort the elements in the receive array with the elements that were not sent in the local array. To do this, every element in the receive array are compared with every element that was not sent in the local array, If the value in the local array is lower than the element in the receive array, the element from the local array is inserted into the merge array and the next element in the local array is compared to the current element in the receive array. If the opposite is true, the element from the receive array is inserted into the merge array and the current element in the local array is compared to the next element in the receive array. This is repeated until every unsent element in the local array and every element in the receive array has been inserted into the merge array, and this results in the elements being sorted in the merge array. Lastly, the size of the local array is changed with realloc in order to accommodate for the number of elements in the merge array and the elements are then copied from the merge array to the local array. Lastly the subgroup is freed so a new split can be performed in the beginning of the next iteration and then the next iteration begins.

When the iterations are done the timer stops and the merge and receiver arrays are freed. Every processor now has a sorted list. Every element in process 0 is smaller than

every element in process 1 and so on. The last step is to change the size of the local array in process 0 to accommodate the values from all the processes, and every process sends its local array to process 0 which stores it in its local array. The local array in process 0 then contains all the elements from the input, sorted. Every process sends its time to the processing element with rank 0, and the highest time is taken and printed to the console. Lastly process 0 writes an output file of the sorted array in text format with every element separated with white space. Lastly the local arrays and the Cartesian topology are freed, MPI is finalized and the program ends.

Since the sizes of the three arrays (local array, receive array and merge array) are not fixed, the program makes heavy use of realloc calls to resize the matrices as needed, with the hope of reducing the amount of total memory that is needed at the same time. This was needed since the largest input lists were upwards of 20 GB and the memory on Snowy where the program was tested was limited to 128 GB per node.

3 Performance Experiments

When the program had been written, the next step is to measure the parallel performance. All measurements were done on Uppmax on the Snowy clusters, using the timings printed by the program. Timing measurements were done on 1, 2, 4, 8 and 16 cores and arrays ranging from $1.25 * 10^8$ to $2 * 10^9$ elements were sorted. A sorted array with descending numbers of size $1.25 * 10^8$ was also sorted to an array with ascending number. Speedup was calculated by dividing the time on the current number of cores by the time gotten on one core.

For the ideal speedup, both Amdahl's law described in equation 1 and Gustafson's law described in equation 2 was considered. p is the number of processing elements and f_p is the part of the code that is parallelized.

$$S \leq \frac{1}{(1 - f_p) + \frac{f_p}{p}} \tag{1}$$

$$S \leq p + (p - 1) * (1 - f_p) \tag{2}$$

Since only the parallel part of the program is timed, f_p is one. This means that both equation 1 and 2 can be simplified to $S \leq p$. The ideal speedup is therefore just the number of processing elements used.

4 Results

Table 1: Weak form for the three different methods

Size	1.25e8	2.5e8	5e8	1e9	2e9
Cores	1	2	4	8	16
Method 1	25.994	27.836	32.301	34.314	43.0462
Method 2	26.230	28.045	29.949	33.360	43.148
Method 3	27.612	27.972	31.063	33.586	42.765

Table 2: Times for the different methods with different amount of processors, sorting 125,000,000 random values

cores	Method 1	Method 2	Method 3
1	25.994	26.230	27.612
2	13.512	13.965	13.976
4	7.159	7.128	7.4221
8	3.943	3.717	3.718
16	2.467	2.461	2.435

Table 3: Speedup for the different methods with different amount of processors, sorting 125,000,000 random values

cores	Method 1	Method 2	Method 3
1	1	1	1
2	1,924	1,878	1,976
4	3,631	3,680	3,720
8	6,593	7,057	7,427
16	10,537	10,659	11,340

Table 4: Times for the different methods with different amount of processors, sorting 125,000,000 descending values

cores	Method 1	Method 2	Method 3
1	9.218	9.495	9.197
2	5.261	5.179	5.015
4	3.067	2.940	2.788
8	1.782	1.698	1.611
16	1.423	1.452	1.504

Table 5: Speedup for the different methods with different amount of processors, sorting 125,000,000 descending values

cores	Method 1	Method 2	Method 3
1	1	1	1
2	1.752	1.833	1.834
4	3.006	3.230	3.298
8	5.173	5.592	5.709
16	6.478	6.539	6.115

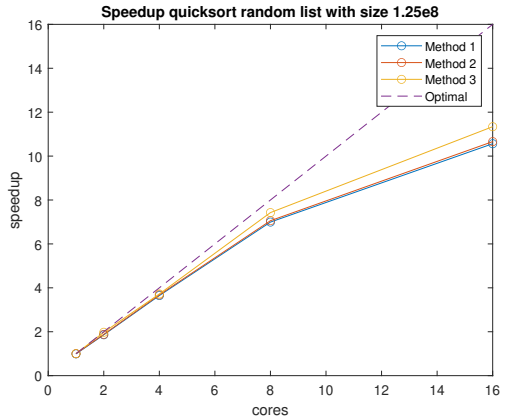

Figure 1: Plotting the times for the different methods with different amount of processors, sorting 125,000,000 random values. Segmented line represents optimal speedup.

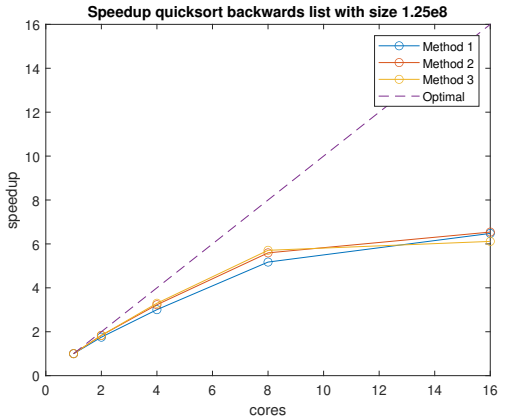

Figure 2: Plotting the times for the different methods with different amount of processors, sorting 125,000,000 descending values. Segmented line represents optimal speedup.

Table 6: Total time and relative time increase of method 3. n*log(n) and n are complexity models that can be compared to the relative time. The code was run on 4 cores.

list size	1.25e8	2.5e8	5e8	10e8
time method 3	3,106	6,310333333	13,42966667	29,074
relative time	1	2,031659154	4,323781927	9,360592402
n*log(n)	1	2,07435676	4,297427038	8,892281115
n	1	2	4	8

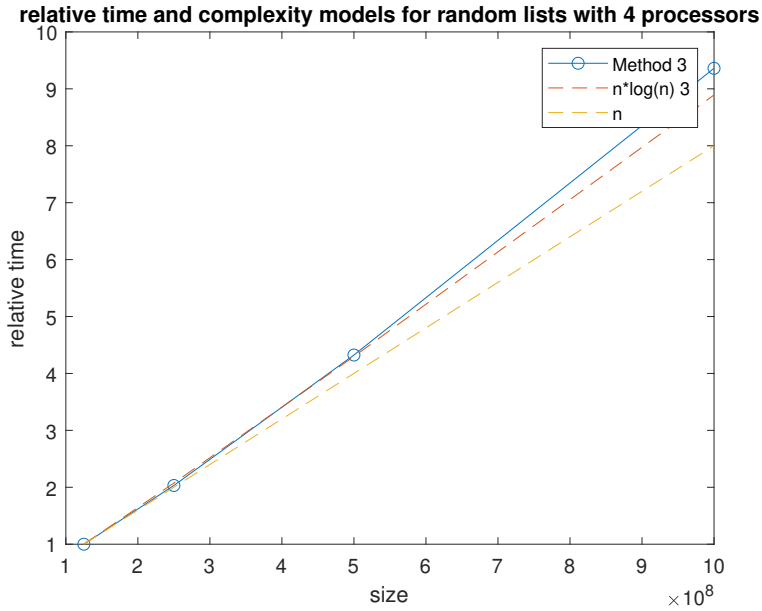

Figure 3: Plotting the total time and relative time increase of method 3. n*log(n) and n are complexity models that can be compared to the relative time. The code was ran on 4 cores.

5 Discussion

For the strong scalability, the speedup in tables 2 and 4 and figures 1 and 2 follows the optimal speedup very closely up to 8 cores when the random list is sorted with a 7-fold speedup, and up to 4 cores when the descending list is sorted to an ascending list with a 3-fold speedup. For the random list, even 16 cores gives a good 11-fold speedup even though it is lower than the 16-fold optimal speedup. With the descending list, there is barely any difference in the speedup of 8 cores compared to 16 cores. Both gives about a 6-fold speedup which means that it would be a waste to use more than 8 cores to sort that type of list.

For the weak scalability, by running the code on 4 cores and doubling the number of elements in the list, the code takes around n*log(n) times longer to run, as seen in figure 3. This means that the time should slightly increase if both the number of cores and the number of elements doubles. From table 1, this holds true for up to eight cores, which corresponds with the strong scalability when sorting a random list which followed the ideal speedup very well up to 8 cores.

The reason the speedup falls off after eight cores is probably due to load balance. Since the arrays will not be equally distributed due to the chosen pivoting element, situations can arise where only a few cores contain all the elements and the other cores has barely any elements at all. The few cores that have most of the elements then has to do most of the work merging and sorting the local array while the other cores wait, this will inevitable lead to a time slower than ideal. Another reason why the speedup tapers of could be that the number of realloc calls increases with the number of cores, thus increasing the time to change the allocated memory that is used. When it comes to OpenMPI, there will also be more messages in the system if more cores are used which could lead to a larger overhead and thus a slower speedup.

One thing of note is that the program is much faster at sorting descending lists

to ascending lists, as can be seen when tables 2 and 4 are compared. The number of elements to be sorted are the same, but the ascending list takes in general four times faster to sort than the random list, thus making this algorithm ideal to sort those types of lists.

Another thing of interest is that both from the weak form data in table 1 and the speedup data in tables 2 and 4 there seem to be little to no effect on changing the pivoting method.