PRL Project 2 Documentation – Merge-splitting sort

Author: Martin Krajňák
Login: xkrajn02
April 8th 2018

Algorithm analysis

Merge-splitting sort algorithm works with an array of processors where each processor is assigned an sub-array of the original array to sort. The number of elements per processor is determined by calculating *n/p* where *n* is number of the elements and *p* is number of processors. Each processor sorts its sub-array with optimal sequential sorting algorithm. The odd numbered processors are waiting until they receive the sorted sub-arrays from the even processors which are then merged with their own sub-arrays to one sorted array. Array is then equally divided, the first half is kept by the odd processors, the second half is send back to even processors. The second part of the iteration runs in similar manner as the odd processors swap their roles with the even processors, as described in Diagram(2). The sorting process is completed after p/2 iterations.

Time and cost analysis

Time complexity of the algorithm is represented in the following steps:

- Reading the array — $O(n)=O(n)$
- Sorting sub-arrays (sequential algorithm from std library) — $O(n)=(n/p)*\log_2(n/p)$
- Sending/Receiving sub-arrays — $O(n)=O(n/p)$
- Merging two sub-arrays to one sorted array — $O(n)=O(2*(n/p))$

From these steps we can calculate time complexity and cost:

$$t(n)=O((n*\log_2 n)/p) + O(n)$$
$$c(n)=p*t(n)=O(n*\log_2 n) + O(n*p)$$

Implementation details

After reading the values of elements, algorithm requires that every processor has the equal portion of values to sort. The important variable is `split_factor` which hold the number of elements per processor. In case that *n mod p* is not equal to zero *(n/p)+1* elements are assigned to each processors and missing values are compensated with `MAX_INT` value from `climits` library. The values are in the end of the array from the beginning of the sorting process and since they represent maximum possible values that can occur within used datatype they will always stay there even after sorting process ends. The variable that holds the number of values read from the file allows to print exact amount of elements and those internally added are just left out.
An algorithm used to sort sub-arrays in each of the processors is provided by sort function from standard c++ library *algorithm* with $O(n*\log_2 n)$ time complexity[1].

Communication

The parallelism is implemented via MPI library[2]. `MPI_Scatter` function is used at the very beginning to divide and distribute the sub-arrays to all processors. Function is prepared for this case by the design we just need to to allocate sub-array for every process and pass the `split_factor`. `MPI_Recv` and *`MPI_Send`* are functions used to receive and send sub-arrays during the sorting process. Both functions have the parameter that specify the processor which should receive the message. Final communication step is achieved with `MPI_Gather` function where we store sub-arrays from every processor one by one to a newly created array.

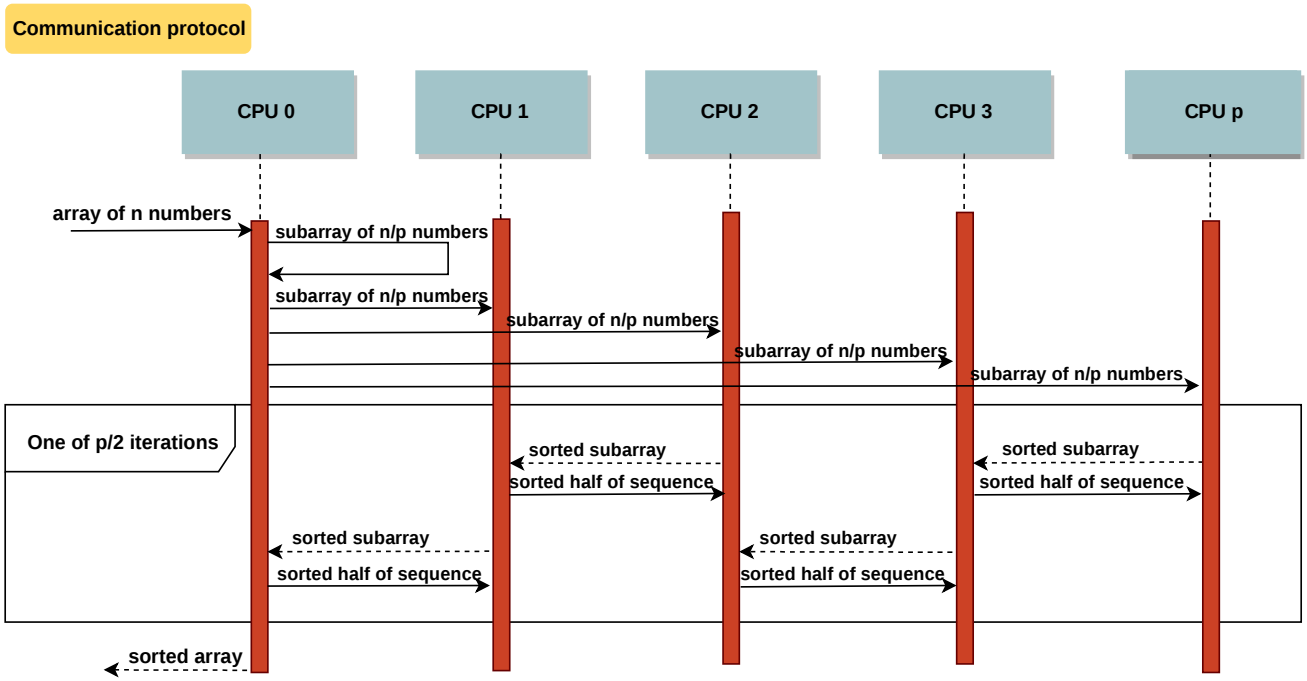

Diagram 1: Sequence diagram demonstrating the communication between processors

1 http://www.cplusplus.com/reference/algorithm/sort/
2 https://www.open-mpi.org/doc/current/

Testing and conclusion

To properly experiment and test the behavior and measure the time of execution was used *dd* utility with combination of values generated from */dev/random* or */dev/urandom* to provide required amount of number sequences. Also the Unix *time* utility was used to measure the exact amount of time taken to sort given array. Measure was only the execution of the sorting program since execution of dd and /dev/random was taking significant amount of time especially with bigger sequences.

The measured values of time in milliseconds are shown in Chart(3). Testing has been executed on two machines, the first is the school server Merlin, the second one is my personal laptop. The results for both machine are quite similar since we used four processors on both machines. The interesting fact that can be observed is that the amount if time start to grow almost in linear fashion just after amount of numbers $\log_2 n$ outgrows the number of processors. Before this point is reached algorithm takes almost the same amount of time regardless of length of sequence.

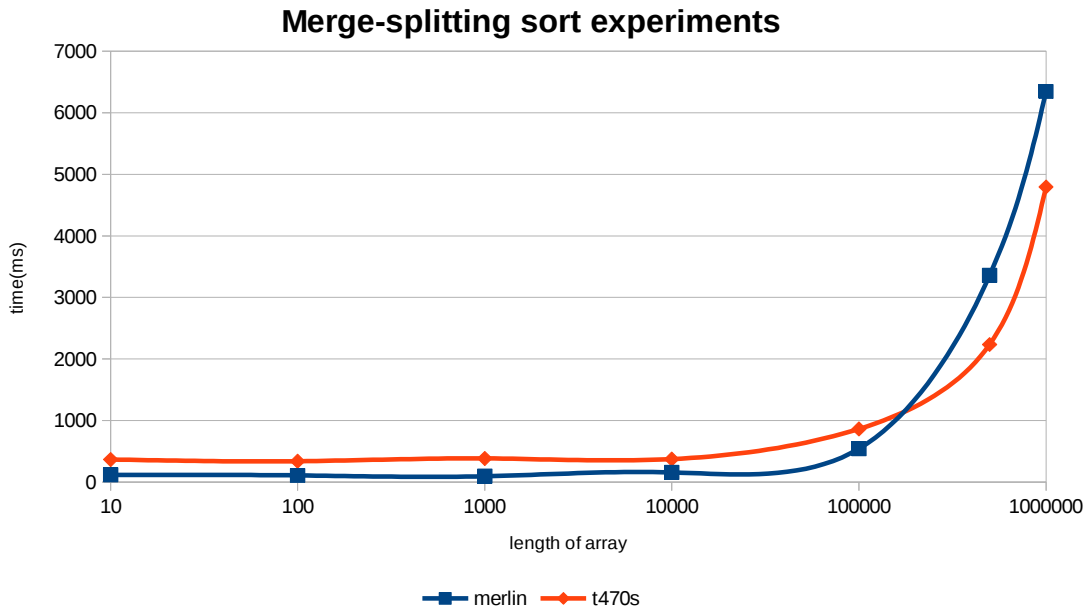

Chart 1: Time spend by sorting arrays on 4 processors