Documentation (PRL) - Assignment 2

Šimon Stupinský - xstupi00@stud.fit.vutbr.cz

The objective of this documentation is describing the implementation of the parallel sorting algorithm *Odd-Even Transposition sort* with using the library *Open MPI* in the language *C++*.

ALGORITHM DESCRIPTION

The Odd-Even Transposition algorithm sorts **n elements** from the sequence $\mathbf{S} = \{\mathbf{e_1}, \mathbf{e_2}, \ldots, \mathbf{e_n}\}$ with using the same count of processors $\mathbf{P_1}, \mathbf{P_2}, \ldots, \mathbf{P_n}$. Each processor $\mathbf{P_i}$ holds one element from the input sequence, we denote this element by $\mathbf{p_i}$ for all $\mathbf{i} \in \{\mathbf{1}, \mathbf{2}, \ldots, \mathbf{n}\}$, and let us initially $\mathbf{p_i} = \mathbf{e_i}$. It is a fact, that at any time during the execution of the sorting each processor holds one element, and upon termination, the $\mathbf{p_i}$ is the $\mathbf{i^{th}}$ element of the sorted sequence.

This algorithm alternates between two phases called the *odd* and *even*, that are performed repeatedly. In the first phase, all processors $\mathbf{P_i}$ with odd index ($\mathbf{i} = \mathbf{2k}, \mathbf{k} \in \{\mathbf{1}, \mathbf{2}, \ldots, \mathbf{n/2}\}$) receive the element $\mathbf{p_{i+1}}$ from processor $\mathbf{P_{i+1}}$. Subsequently, odd-numbered processors compare both numbers and when $\mathbf{p_i} > \mathbf{p_{i+1}}$, then processors $\mathbf{P_i}$ and $\mathbf{P_{i+1}}$ exchange their elements, which they held at the beginning of this step. Similarly, during the even phase, each processor with an even label ($\mathbf{i} = \mathbf{2k-1}$, $\mathbf{k} \in \{\mathbf{1}, \mathbf{2}, \ldots, \lceil \mathbf{n/2} \rceil\}$) perform the same operations as did the odd-numbered ones in the odd phase.

After $\lceil \mathbf{n/2} \rceil$ repetitions of these two phases in the given order, no further exchange of elements between processors is necessary. Hence the algorithm terminates in the state, where for all values of processors $\mathbf{P_i}$ applies $\mathbf{p_i} < \mathbf{p_{i+1}}$, where $\mathbf{i} \in \{\mathbf{1}, \mathbf{2}, \ldots, \mathbf{n-1}\}$.

ALGORITHM ANALYSIS

During each phase of the algorithm are executed the one comparison and two routing operations between processors. The complexity of each phase, odd and even, therefore requires constant time – $\mathcal{O}(\mathbf{1})$. As we indicated above, these two phases are executed $\lceil \mathbf{n/2} \rceil$ times, therefore a total of **n** such phases are performed in the case of the even count of numbers and $(\mathbf{n+1})$ in the case of the odd count of numbers. Thus, the running time of this algorithm is $\mathbf{t}(\mathbf{n}) = \mathcal{O}(\mathbf{n})$, where **n** represents the size of the input sequence **S**. Since at any time during the execution of the algorithm each processor holds one element, it is clear, that the required number of processors is $\mathbf{p}(\mathbf{n}) = \mathbf{n}$. The cost of the parallel implementation is in general define as $\mathbf{c}(\mathbf{n}) = \mathbf{t}(\mathbf{n}) \cdot \mathbf{p}(\mathbf{n})$, and therefore the cost of this algorithm is $\mathbf{c}(\mathbf{n}) = \mathcal{O}(\mathbf{n}) \cdot \mathbf{n} = \mathcal{O}(\mathbf{n^2})$. Since each processor with the left neighbour needs **one** temporary variable to receive the number and subsequently comparison, the space complexity of the algorithm is $\mathcal{O}(\mathbf{n})$.

The algorithm has the optimal cost when is applied that $\mathbf{c}(\mathbf{n})_{\mathbf{seq}} = \mathbf{t_{seq}}(\mathbf{n})$. Since the time complexity of the best sequential sorting algorithm for **n** elements is $\mathcal{O}(\mathbf{n} \cdot \log \mathbf{n})$, this implementation of Odd-Even Transposition sort is not cost-optimal. With respect to the sorting algorithm *QuickSort*, it achieves a speedup of $\mathcal{O}(\log \mathbf{n})$ only: $\mathbf{t_{seq}}(\mathbf{n})/\mathbf{t}(\mathbf{n}) = (\mathbf{n} \cdot \log \mathbf{n})/\mathbf{n} = \log \mathbf{n}$. This is due to the fact that in some circumstances a situation occurs when some processors contain the right ordered elements and therefore they do not have the element to exchange with other processors. Further, as we know, it uses a number of processors equal to the size of the input sequence, which is unreasonable. From this analysis of the properties, we can conclude, that the algorithm Odd-Even Transposition does not appear to be too. attractive [1]

IMPLEMENTATION

The algorithm is implemented in the language **C++** with using library **Open MPI** and without some other specific dependencies. The input of the algorithm is the binary file, which contains the numbers in the range $< \mathbf{0}, \mathbf{255} >$. Concerning this property of input numbers, was chosen data type `unsigned char` to represent numbers within the algorithm, respectively type `MPI_BYTE` within MPI communication.

In the beginning, the MPI execution environment is initialised and the overall number of processors is obtained, using appropriate functions (`MPI_Init`, `MPI_Comm_size` and `MPI_Comm_rank`). The master processor (rank 0) loads the numbers from the input file into an allocated array and obtained their count. Subsequently, the master processors distributes loaded numbers to all remaining processors using `MPI_Send` function. All processors, including the master, receive the sent numbers from the master (`MPI_Recv`) and store it to the own variable.

In the next steps follows the sorting algorithm itself, but since it has been described above, we introduce only some interesting implementation details. Firstly, before the sorting, it is needed to compute limits for both phases of the

sorting iteration. When runs the first phase with odd-numbered processors is necessary to ensure that, when was entered the even count of the numbers, then the last processor has not sent number to the right neighbour, because it does not exist. Further, it is also necessary to ensure that the number will be expected only by even-numbered processors, that have the left neighbour, so the master processor certainly not. Similar conditions must also be met in the second phase with even-numbered processors, where also the number can only be sent if the right neighbour exists and the number can be expected only by odd-numbered the processor, that have the left neighbour. These requirements are satisfied by the following formula $\mathbf{2} \cdot (\mathbf{n}/\mathbf{2}) - \mathbf{1}$ for the first phase, respectively by the $\mathbf{2} \cdot ((\mathbf{n} - \mathbf{1})/\mathbf{2})$ for the second, where the **n** represents the size of the input sequence. The exchange of the numbers between the processors is performed with use MPI functions `MPI_Send` and `MPI_Recv`, in the required order according to algorithm specification.

After sorting the input sequence, in the last step, all processors send the own number to the master processor, which these numbers receives and stores it to the allocated array. When the main processor receives all the numbers, it prints out them into standard output and then frees the allocated memory. Following this is terminated the MPI execution environment and program successfully exits.

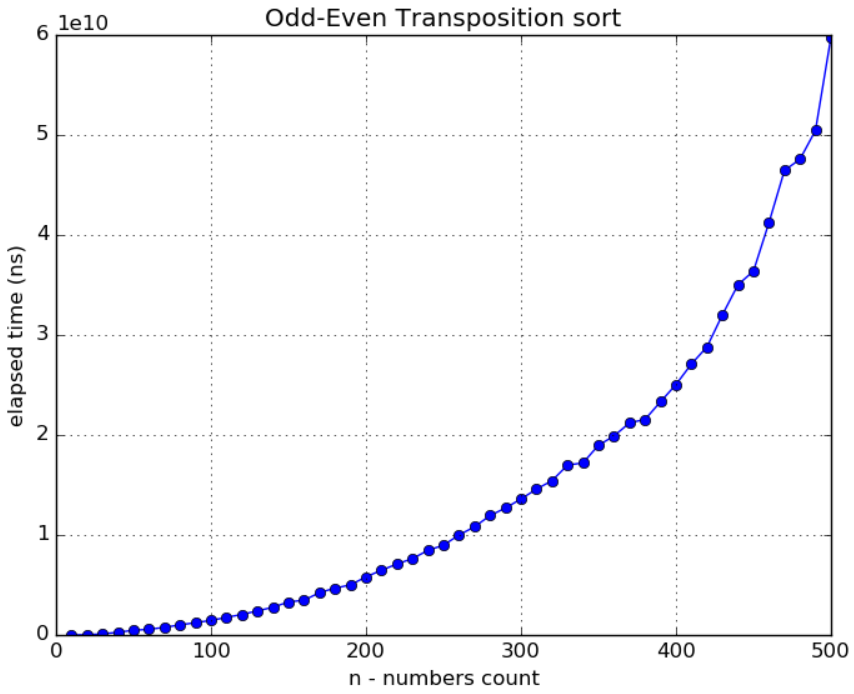

Figure 1. The graph of dependence the running time of the Odd-Even Transposition sorting algorithm on the size of the input sequence (i.e. the count of the numbers).

EXPERIMENTS

The goal of the experiments was to verify the time complexity of the parallel sorting algorithm Odd-Even transposition. For purposes to measure the time complexity as accurately as possible, was required to eliminate all intrusiveness, which has an indirect relation to the sorting algorithm. Into the final running time are not include the operations such as loading the numbers from the input file, distributing the numbers to all processors, respectively back distributing the ordered sequence to the master processor or the subsequently printing this sequence by the master. We decided to ignore these operations because of their role within sorting algorithm itself it is not significant. Therefore the measuring of the running time represents the running time of the main sorting loop, which was described above in the first section. This also corresponds to the performed analysis of the algorithm, and derived complexities and cost. The `MPI_Wtime` library function was used for measurement and it was inserted into the respective sections of the program.

The experiments were performed on the input sequence with length in the range $\{\mathbf{10}, \mathbf{20}, \ldots \mathbf{500}\}$, which was generated by the `dd utility` with using random generator `/dev/urandom`. For each selected length of the sequence were performed **14** iterations, where the first iterations was ignored due to possible noise. We also need to be

aware of the variability in individual measurements. When we run the parallel program several times, it is extremely likely that the elapsed time will be different for each run. Obviously under the assumption, when we use the same input of the program, which runs at the same system. It might seem, that the best way to deal with this would be report either a *mean* or *median* run-time. However, it is unlikely that some outside event could actually make our program run faster than its best possible run-time. Therefore it is more practical report the minimum run-time, instead of reporting the mean or median run-time. [2]

The experiments were performed on the *Anselm* supercomputer cluster[1], which belongs to the cluster of supercomputers managed by *IT4Innovations*. We use the recommended way to run an MPI application, using the required number of MPI processes per node (according to the number count) and **16** threads per socket, on **4** nodes. To allocate nodes via the express queue interactively, we use the following command, where the `mpiprocs` and `ompthreads` parameters allow for the selection of the number of running *MPI* processes per node as well as the number of OpenMP threads per *MPI* process: `qsub -q qexp -l select=4:ncpus=16:mpiprocs=100:ompthreads=1 -I`.

SUMMARY

The results of the performed experiments and the constructed Graph 1 show, that the run-time needed to the sorting of the sequence grows a little faster than linearly. This finding is not supported by the derived time complexity, which was linear with respect to the count of numbers in the input sequence. This is due to the fact that as the number of processors increases, the necessary overhead for communication between them is rising.

Note that in today's parallel computers it takes more time to send an element from one process to another than it takes to compare the elements. Consequently, any parallel sorting formulation that uses as many processes as elements to be sorted will deliver very poor performance because the overall parallel run time will be dominated by inter-process communication. [3]

[1] https://docs.it4i.cz/anselm/introduction/

COMMUNICATION PROTOCOL

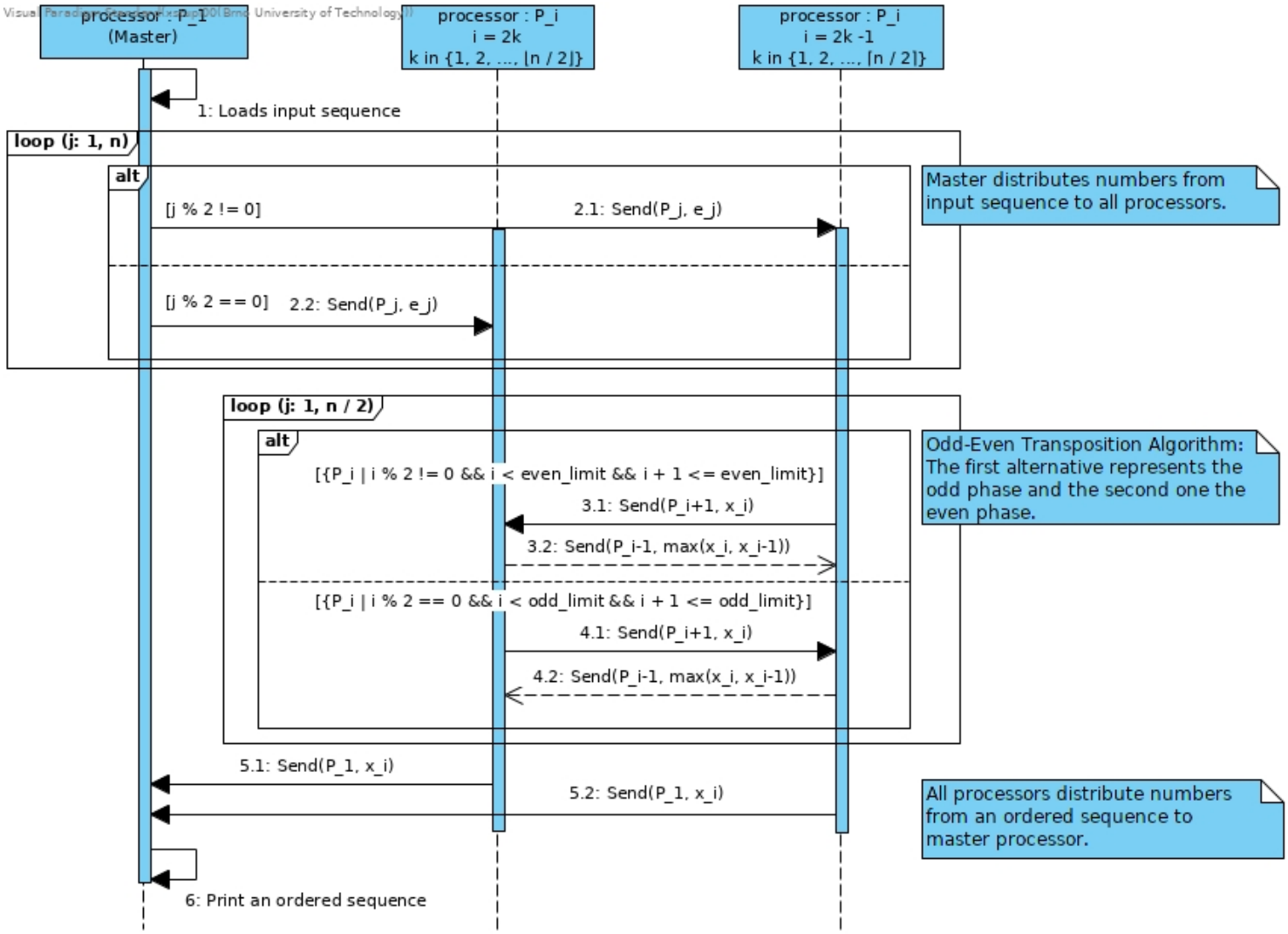

Figure 2. The sequence diagram describing the communication between the processors at sorting algorithm Odd-Even Transposition. The individual participating processors are represented by the two groups (odd or even) according to their numbering. The special role has a master processor in the initial and final operations, and therefore it is also represented individually. We do not depict the operations `MPI_Recv` because of it clear that two processes, such as sender and receiver, are involved in the operation **Send**

.

REFERENCES

[1] S.G. Akl. *The Design and Analysis of Parallel Algorithms*. Prentice-Hall international editions. Prentice Hall, 1989.

[2] Peter Pacheco. *An Introduction to Parallel Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011.

[3] Ananth Grama, George Karypis, Vipin Kumar, and Anshul Gupta. *Introduction to Parallel Computing*. Addison-Wesley, second edition, 2003.