Pipeline merge sort in Open MPI

PRL of 2020/2021

Author: **Patrik Németh**
Login: **xnemet04**

1 The algorithm and its implemetation

The pipeline merge sort is a variation of the merge sorting algorithm suited for multiprocessing. It is diferrent in that it divides the merging of different length arrays between multiple processes, which are set up as a pipeline. Each channel between two processes consists of two queues - a top and a bottom one, which hold the values to be merged by the next process. The algorithm requires $p(n) = \log_2(n)+1$ processes for n input values. The processes will be indexed from 0 to k, where k is the last process in the pipeline. The merge sorting pipeline is visualized in figure 1.

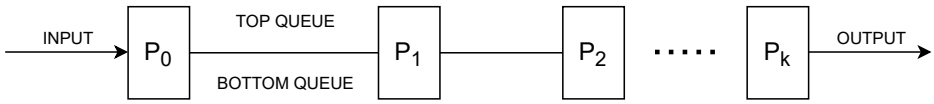

Figure 1: Diagram of the merge sorting pipeline.

The first process (P_0) serves as the input parser that loads the input values and sends them one by one to P_1. After parsing every input value, P_0 may exit. The rest of the processes ($P_i | 1 \leq i \leq k$) work very similarly to each other. They all carry out two operations during each sorting cycle (*receive* and *merge*, explained in sections 1.1 and 1.2 respectively).

Figure 2 shows the lifetimes of the processes. The first process receives no messages, it only sends the input values to the second process. The rest of the processes must first wait until they receive the appropriate number of values and only then they may begin sorting. This is signified by the slight delay between receiving the first value and sending the first sorted value to the next process. Every process terminates after sorting n values. All inter-process communication is carried out by the `MPI_Send()` and `MPI_Recv()` library functions within the `MPI_COMM_WORLD` communicator.

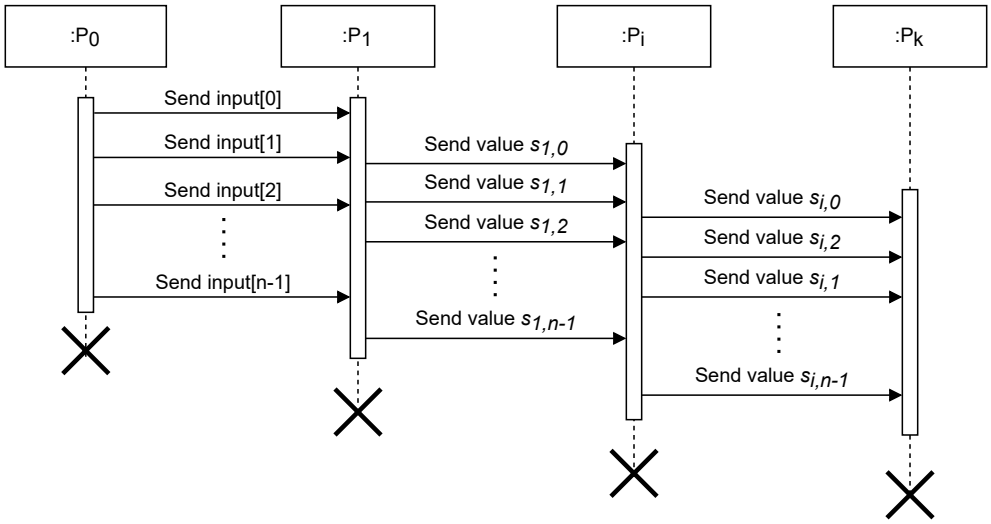

Figure 2: The sequence diagram of the pipeline merge sort implementation. The symbol $s_{i,j}$ represents the j-th sorted value sent by process i.

1.1 Receive

Every Pi manages its input queues, so that they are assigned the correct number of incoming values sent from the previous process. This number, q, is governed by the rank of the current process, such that $q = 2^{i-1}$, where i is the rank. The incoming values are first assigned to the top queue up until q number of values is reached, after which the queue is switched and the incoming values are assigned to the bottom queue. After every q-th value, the queue is switched again.

This operation is implemented as the `receive()` function. The static `q_picker` variable is used to keep track of the correct placement of the incoming values. It is incremented after every received value. If `q_picker` is less than q, then the value is placed to the top queue, otherwise it is placed to the bottom. Upon `q_picker` reaching $2q$, it is reset to 0 and the next value is again placed to the top queue.

1.2 Merge

This operation sorts the values contained in the input queues by merge sorting. The process merges the first value only after its top queue contains q values and the bottom contains at least one. One full merge cycle is completed after taking q number of values from both input queues (i.e. $2q$ values in total). The process compares the first values of the input queues and the lower value is sent to the next process P_{i+1}. Each process must take at most q number of values from each of its input queues via comparison. Otherwise the remaining values, until $2q$, are taken from the other queue regardless of those values' relation to any other values in the input queue. After a full merge cycle, the process restarts the merge cycle and repeats it until all input values have been merged. The last process in the pipeline is only different in that its output value is not sent to the next process, but rather to the output data structure (or, as in the case of this implementation, to the standard output stream).

This operation is implemented as the `merge()` function. Each of the input queues has its own q counter (`from_top` and `from_bot`) that are implemented as static variables. These are incremented depending on from which input queue the last sorted value was taken. Upon one of the variables reaching q, the behaviour is changed according to the specification above, and upon both counters reaching q, they are reset to 0 to initialize the next merge cycle.

1.3 Implementation overview

The program starts by loading the input values from the standard input stream. This is done by the first process. As the value loading is quite quick, no measures were implemented for the other processes to passively wait until loading is completed. After this the sorting begins. Each process counts the number of sorted values and after reaching n, they exit. The sorting itself is driven by a loop, where the processes are divided into three distinct groups based on their behaviour. These groups are for the first process, the intermediary processes and the last process. The first process just sends the loaded values to the succeeding process. The intermediary processes call `receive()`, where they receive the incoming value, and `merge()`, where they actually sort the values. The last process does the same as the intermediaries, with the difference of calling `merge()` with a flag modifying its behaviour, so that it doesn't send the sorted value to the succeeding process, but rather to the standard output stream.

2 Complexity and cost

The required number of processes is $p(n) = \log_2(n) + 1$, where n is the number of input values. The time complexity can be derived from the number of values needed by each process to begin sorting and the total number of inputs. Process P_0 requires 1 input to start sorting and all remaining inputs will be processed in parallel with the rest of the processes. Every remaining process requires the top queue to be full and the bottom queue to contain at least one value. This means $2^{i-1} + 1$ steps before the i-th process may start

sorting. Generalized to the k-th process in the pipeline, the equation

$$1 + \sum_{i=1}^{k} (2^{i-1} + 1) = 2^k + k \tag{1}$$

can be derived. However the last process needs an additional $n - 1$ steps until all values are merged. This yields

$$2^k + k + (n-1) = 2^{\log_2(n)} + \log_2(n) + (n-1) \tag{2}$$

steps for completely sorting n input values through the k-th process. The index k was substituted for its formula on the right hand side of equation 2. After simplifying the previous, we are left with

$$2n + \log_2(n) - 1\,. \tag{3}$$

The cost of a parallel algorithm is defined as $c(n) = p(n) * t(n)$, where $p(n)$ is the number of required processes for n inputs, and $t(n)$ is the number of steps required for completing the algorithm with n inputs. The process requirement is $\log_2(n) + 1$, which may be notated as $O(\log_2(n))$. The time complexity was shown in equation 3, which may be notated as $O(n)$. Therefore the cost of the algorithm is

$$c(n) = O(\log_2(n)) * O(n) = O(nlog_2(n))\,. \tag{4}$$

As shown in equation 4, the algorithm is optimal.

3 Conclusion

The implemented pipeline merge sorting algorithm works well on 16 values. It however could easily be modified to accept an arbitrary number of inputs. The Open MPI library provided easily introducable message passing capabilities into the implementation.