Paralelné a distribuované algoritmy - PRL 2020/2021

Pipeline Merge Sort

Lukáš Dobiš
xdobis01@stud.fit.vutbr.cz

1 Algorithm description

Pipeline Merge sort (PMS) is parallel sorting algorithm which uses architecture of linearly connected processors. Number of processors k is fixed and derived from equation 1, which depends solely on size of input n. First processor gets on its input pipeline unsorted sequence of numbers to be sorted. These numbers are then sent to second processor alternating on first or second input pipeline. Second to last processors then have each two input pipelines, on which they have partially sorted subsequences of numbers. Last processor has only one output pipeline for final sorted sequence. Processors start sorting after meeting condition of receiving enough numbers on both pipelines. This condition is defined for i-th processor as having received 2^{i-1} numbers on first pipeline and having atleast 1 number on second pipeline. Smaller number is then send to input pipeline of next processor. Each i-th processor with two pipelines merges 2^{i-1} long subsequences from each pipeline into one 2^i long number sequence. Processors are indexed from 0 to $k-1$.

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

- First processor (P_0) – Takes input unsorted number sequence and alternating sends individual numbers to first or second input pipeline of second processor.
- Second to penultimate processor ($P_i \mid i \in 1, 2, .., k-2$) – Start sorting numbers at front of pipelines, after condition of receiving enough numbers on input pipelines is met. This condition is defined for i-th processor as having 2^{i-1} numbers on first pipeline and atleast 1 number on second pipeline. Processor then merges 2^{i-1} numbers from each pipeline into 2^i long sequence. Merging is done by comparing front numbers of each input pipeline and sending smaller number to output pipeline. Output pipeline is input pipeline of next processor. Sending processor after sending 2^i numbers switches output pipeline to second input pipeline of next processor. Having to send 2^{i-1} from each pipeline before output pipeline switch, also means that after sending 2^{i-1} from one pipeline, processor is required to send remainder of 2^{i-1} numbers to be send from other pipeline, while not taking into account if number is actually smaller then its counterpart on pipeline which has sent its 2^{i-1} sequence already.
- Last processor (P_{k-1}) – functions identically to P_i processor but has only single output pipeline onto which he merges his inputs into final sorted sequence of numbers.

2 Complexity analysis

First processor P_0 only reads and sends numbers to second processor, since he is just reading and immediately using number his time complexity $t(n)$ is linear $\mathcal{O}(n)$. Second and other P_i processors including last processor start their sorting after previous processor sends them on their first pipeline 2^{i-1} numbers and on the second pipeline 1 number. This means that they start $2^{i-1}+1$ cycles later than previous processor and that last processor in array will also be last one to end. Therefore P_i processor start cycle can be computed from next equation 2.

$$1 + \sum_{j=0}^{i-1} 2^j + 1 = 2^i + i \tag{2}$$

And after starting their sort, processors finish after sorting $n-1$ numbers so their last cycle will be 3.

$$2^i + i + (n-1) \tag{3}$$

Since last processor P_{k-1} differs from P_i processor only in printing his merged number to standard output, same equation can be used to compute his last cycle 4.

$$2^{k-1} + k - 1 + (n-1) = 2^{\log_2 n} + \log_2 n + (n-1) = 2n + \log_2 n - 1 \tag{4}$$

From this equation is then obvious that time complexity $t(n)$ for algorithm is also linear $\mathcal{O}(n)$. Finally combined with number of processors k from equation 1, can be derived the final cost of algorithm in cost equation 5. This cost is optimal for sorting algorithm.

$$c(n) = t(n) * p(n) = \mathcal{O}(n) * (\log_2(n) + 1) = \mathcal{O}(n \log_2 n) \tag{5}$$

3 Implementation

Algorithm is implemented in programming language C++, with openMPI library for message passing between processors. Code of algorithm is in single file `pms.cpp`. For sending and receiving messages are used open-MPI functions `MPI_Send` and `MPI_Recv`. Program starts with openMPI library initialization, after which each process gets his process rank, which is used for message passing and to distinguish each process behavior. Processes are numbered from 0 to $k-1$, where k is number of processes determined by 1 equation. Each process finds out how many processes are running, this information is later used to determine input size, which affects how many numbers will each process merge. After this part all processes diverge in behavior depending on their rank.

First process indexed with rank 0 reads unsorted numbers (integer 0-255) from file `numbers` and prints them to standard output. Each number is then sent as message to second process pipeline. Alternating between his input pipelines after each number. Pseudo code of first process is on figure 1, in project code this is all implemented in function `readNums`.

Algorithm 1 First processor implementation

```
1: Open FILE
2: for $i = 1$ to FILE size do
3:     Read number
4:     if $i \bmod 2 == 0$ then
5:         Send number on first input pipeline of second processor
6:     else
7:         Send number on second input pipeline of second processor
8:     end if
9: end for
```

Other processes start in while loop until they sort all numbers, whose size they get from number of used processes. Each process has two input pipelines implemented as `std::queue` queues from standard library, these pipelines have each their own tag either `PIPELINE_1_TAG` or `PIPELINE_2_TAG`. At start of the loop they wait to receive number as message. After receiving number they add it to the front of tagged pipeline queue, where tag is based on tag of the received message. When enough numbers are received on both pipeline queues to meet condition, sorting flag is set. Processes then start merging numbers as described in algorithm analysis 1. Chosen number is then printed on standard output or sent as message to one rank higher process, depending on process rank, if it is last process or not. Pseudo code of these processes is on figure 2, in project code this is implemented in function `pipe_line_mergesort`. Message communication is visualized in sequence diagram on figure 3. Experiments measured time duration averages, from first process start to last process end, for ascending, descending and unsorted sequence on Merlin and AMD Ryzen 5 2600 CPU. Measured averages are from 10 measurements, on table 1. Measuring was implemented by passing start time to last process to compute time duration as difference. (Experiment communication is excluded from communication protocol)

	Unsorted	Ascending	Descending
AMD CPU	0,000208s	0,000196s	0,000212s
Merlin	0,000927s	0,001014s	0,000864s

Table 1: Measured averages for different orderings of input number sequences

Algorithm 2 Second to last processor implementation

```
 1: while not all numbers are sorted do
 2:     Wait for message from previous processor
 3:     After receiving message add it into correct pipeline
 4:     if pipelines are filled to meet condition then
 5:         Set sort flag to start sorting
 6:     end if
 7:     if sorting flag is set then
 8:         Compare input pipelines front numbers and pick smaller number
 9:         if is last processor then
10:             Print smaller number
11:         else
12:             Send smaller number to next process
13:         end if
14:     end if
15: end while
```

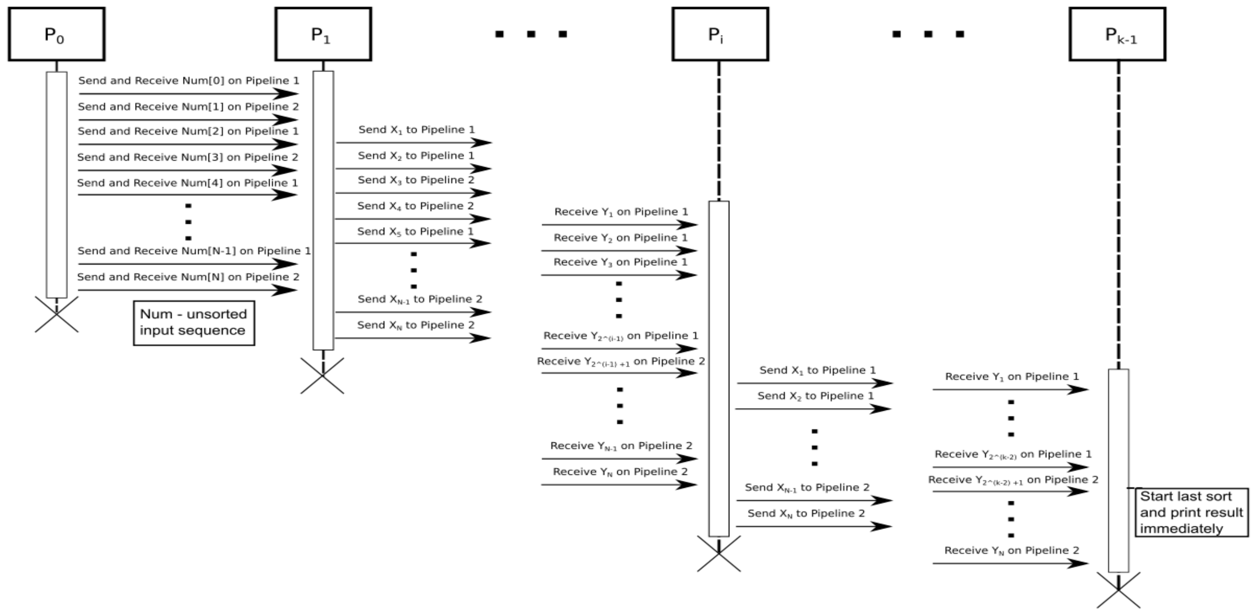
P_0
P_1
P_i
P_{k-1}
Send and Receive Num[0] on Pipeline 1
Send and Receive Num[1] on Pipeline 2
Send and Receive Num[2] on Pipeline 1
Send and Receive Num[3] on Pipeline 2
Send and Receive Num[4] on Pipeline 1
Send and Receive Num[N-1] on Pipeline 1
Send and Receive Num[N] on Pipeline 2
Num - unsorted input sequence
Send X_1 to Pipeline 1
Send X_2 to Pipeline 1
Send X_3 to Pipeline 2
Send X_4 to Pipeline 2
Send X_5 to Pipeline 1
Send X_{N-1} to Pipeline 2
Send X_N to Pipeline 2
Receive Y_1 on Pipeline 1
Receive Y_2 on Pipeline 1
Receive Y_3 on Pipeline 1
Receive $Y_{2^\wedge(i-1)}$ on Pipeline 1
Receive $Y_{2^\wedge(i-1)+1}$ on Pipeline 2
Receive Y_{N-1} on Pipeline 2
Receive Y_N on Pipeline 2
Send X_1 to Pipeline 1
Send X_2 to Pipeline 1
Send X_{N-1} to Pipeline 2
Send X_N to Pipeline 2
Receive Y_1 on Pipeline 1
Receive $Y_{2^\wedge(k-2)}$ on Pipeline 1
Receive $Y_{2^\wedge(k-2)+1}$ on Pipeline 2
Receive Y_N on Pipeline 2
Start last sort and print result immediately

Figure 3: Communication protocol for message passing between processes

4 Conclusion

This work explains and successfully implements Pipeline Merge Sort algorithm. Starts by describing each processor part in this algorithm, continues with analyzing algorithm complexity and ends with its implementation based on openMPI library. Experiments failed to show significant difference when sorting, this implies that since number of merge comparisons stays same, then time does not change for different orderings of sequences.