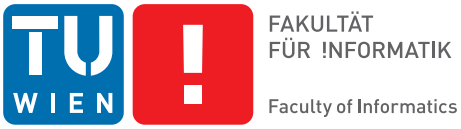


Project Report

for

Project 2

as part of VU on

High Performance Computing

SS 2017

Philipp Paris

e1325664@student.tuwien.ac.at

1325664

Emmanuel Pescosta

e1326934@student.tuwien.ac.at

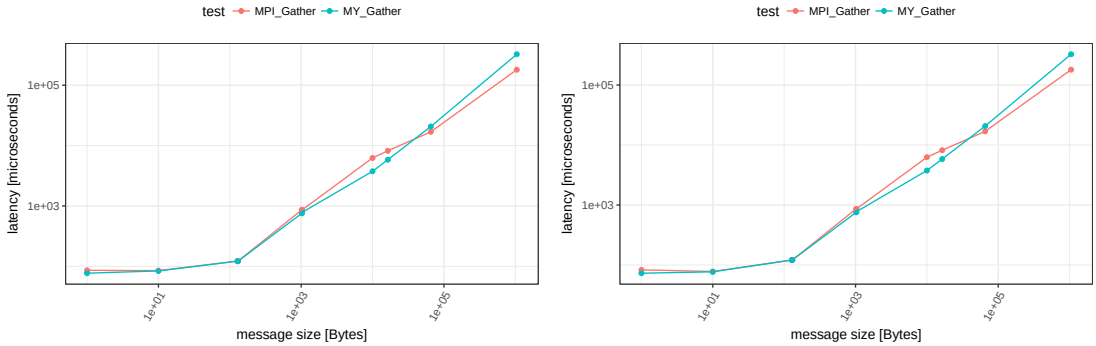
1326934

12 June 2017

1 Algorithm `MY_Gather`

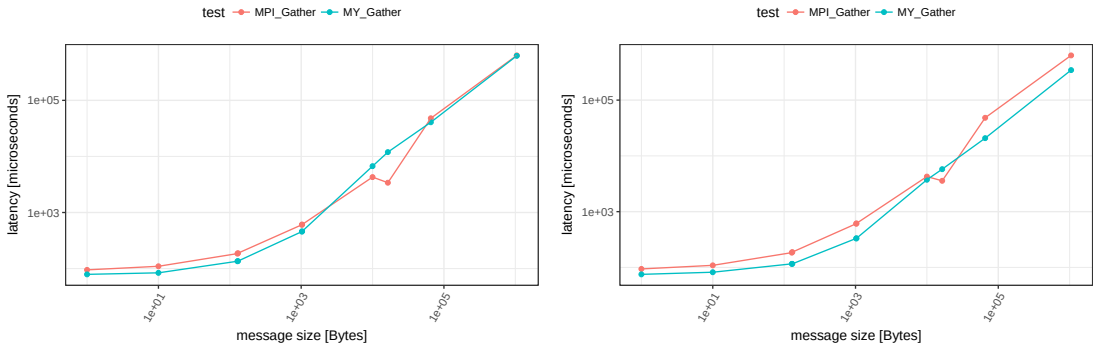
1.1 Strategy Comparison

Two different `Gather` algorithms, one based on Binomial-Trees and one based on a Divide-And-Conquer approach were implemented and tested. Both algorithms fulfill the specification of `MPI_Gather` and use the minimum amount of memory whenever possible. Figure 1 shows the performance of each implementation compared to the `Open MPI Gather` implementation with $root = 0$. Figure 2 shows the same comparison with $root = 2$.



(a) Binomial-Tree Implementation (b) Divide-And-Conquer Implementation

Figure 1: Binomial Tree and Divide-And-Conquer `Gather` with $root = 0$



(a) Binomial-Tree Implementation (b) Divide-And-Conquer Implementation

Figure 2: Binomial Tree and Divide-And-Conquer `Gather` with $root = 2$

As seen in Figure 1, both algorithms perform similarly when the root node has rank 0. In this case the binomial-tree based algorithm directly receives into the *recvbuf* without the need of expensive reordering. But when using rank 2 as the root, the reordering of data in the Binomial-Tree implementation cannot be avoided and thus leads to an overhead.

Therefore the Divide-And-Conquer algorithm performs better when rank $\neq 0$, as seen in Figure 2.
Because the following benchmarks are all done with $root = 0$, the binomial-tree implementation was chosen as the main strategy.

1.2 Description / Strategy

All processing nodes are organized in a binomial-tree with the root of the `Gather` algorithm as the root of the tree. Each leaf node sends its data to its parent node and all non-leaf nodes forward all received data as well as the local data of the node itself to the parent. Due to the use of a binomial tree, the calculation of the parent and child ranks can be done via fast bit-operations.

For cases where the root node has $rank \neq 0$, the algorithm uses virtual ranks: $vrank = (rank - root - N) \mod N$. In this case the data is received in virtual rank order at the root and has to be reordered to regain the real rank order.

To reduce the memory footprint of the algorithm, each node determines the expected amount of data it needs to store in the temporary buffer. Leaf nodes don't require any temporary buffer. For the root node the number of expected blocks, where a block is the amount of data of a single node, is equal to the the number of the nodes in the communicator. Each forwarding node (non-leaf and non-root nodes) can simply calculate the amount of expected blocks by taking the minimum of $2^{count\_zeros(vrank)}$ and $size - vrank$.

1.3 Round- and Bandwidth Optimality

1.3.1 Round Optimality

Due to the tree structure nodes on the same level send data to their parent nodes in parallel, so each layer of the tree represents one communication round. Therefore the number of communications rounds is limited by $\mathcal{O}(log(p))$.

1.3.2 Bandwidth Optimality

When m is the total amount of data to be gathered, then every process sends and receives $\frac{m}{p}$ units of data at every step. The algorithm takes $p - 1$ steps to complete, and therefore the bandwidth term is $\frac{p-1}{p} m\beta$.
This cannot be reduced because the root node has to receive $\frac{m}{p}$ units of data from $p - 1$ processes.

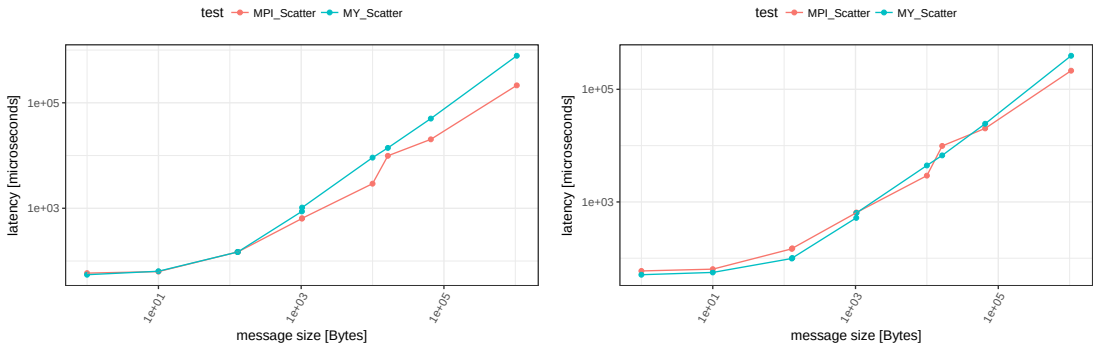
1.4 Harmful Algorithmic Latency

No Algorithmic Latency: Due to the calculation of child and parent nodes via bit operations, each processor can start to send/receive after $\mathcal{O}(1)$ steps.

2 Algorithm `MY_Scatter`

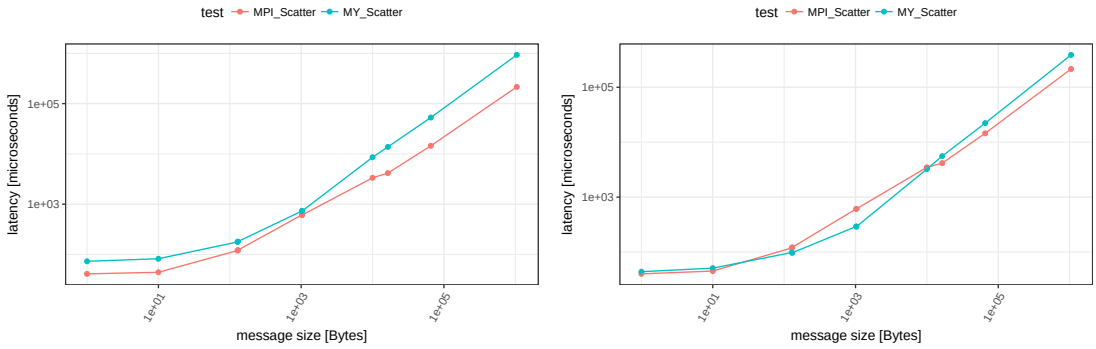
2.1 Strategy Comparison

Again two `Scatter` algorithms one based on Binomial-Trees and one based on Divide-And-Conquer were implemented and compared. Both algorithms fulfill the specification of `MPI_Gather` and use the minimum amount of memory whenever possible. Figure 3 shows the performance of each implementation compared to the `Open MPI Scatter` implementation with $root = 0$. Figure 4 shows the same comparison with $root = 2$.



(a) Binomial-Tree Implementation (b) Divide-And-Conquer Implementation

Figure 3: Binomial Tree and Divide-And-Conquer `Scatter` with $root = 0$



(a) Binomial-Tree Implementation (b) Divide-And-Conquer Implementation

Figure 4: Binomial Tree and Divide-And-Conquer `Scatter` with $root = 2$

As before, the performance of the Binomial-Tree algorithm performs worse than the Divide-And-Conquer approach when using $rank \neq 0$, due to the reordering overhead on the `Scatter` root node. Also the Divide-And-Conquer algorithm performs better for medium and big message sizes.
Due to the better overall performance of the Divide-And-Conquer based algorithm, this

implementation was chosen as baseline implementation for the following discussions.

2.2 Description / Strategy

Divide-And-Conquer works by recursively breaking down a given problem into a number of sub-problems, in this case two sub-problems. The processes are split into two sets and the root nodes selects one process, denoted as *subroot*, from the other set and sends half of the data to it. This is repeated for both sets until every node has received its data. On each communication round the number of senders is increased by a factor of two, while the amount of data to be send on each node is decreased by a factor of two.

Cases where the `Scatter` root node has $rank \neq 0$ are already covered by this algorithm and thus don't require additional efforts.

2.3 Round- and Bandwidth Optimality

2.3.1 Round Optimality

Using the Divide-And-Conquer approach, the data is distributed to the nodes along a binary tree, where all communication on the same tree-layer is executed at the same time. Therefore the number of communication rounds is limited by $\mathcal{O}(log(p))$.

2.3.2 Bandwidth Optimality

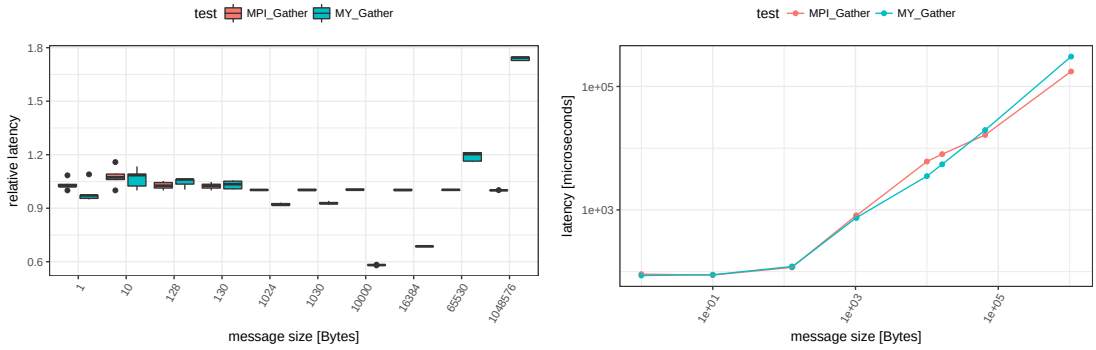
The root node has to send $\frac{m}{p}$ units of data to $p-1$ processes and therefore the bandwidth term for the Divide-And-Conquer Scatter is $\frac{p-1}{p}m\beta$.

2.4 Harmful Algorithmic Latency

No Algorithmic Latency: Each processor can start to send/receive after $\mathcal{O}(1)$ steps.

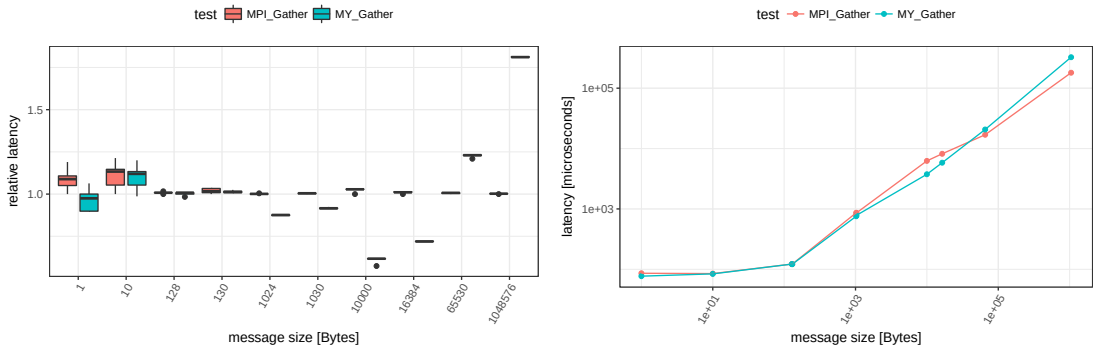
3 Experimental Results

3.1 Experiments and Discussion – `MY_Gather`



(a) Relative Runtime with $p = 31 \times 16$

(b) Absolute Runtime with $p = 31 \times 16$

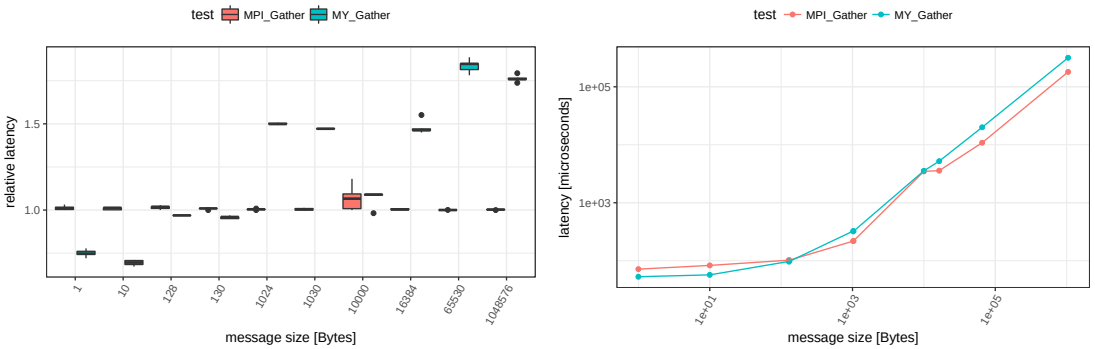


(c) Relative Runtime with $p = 32 \times 16$

(d) Absolute Runtime with $p = 32 \times 16$

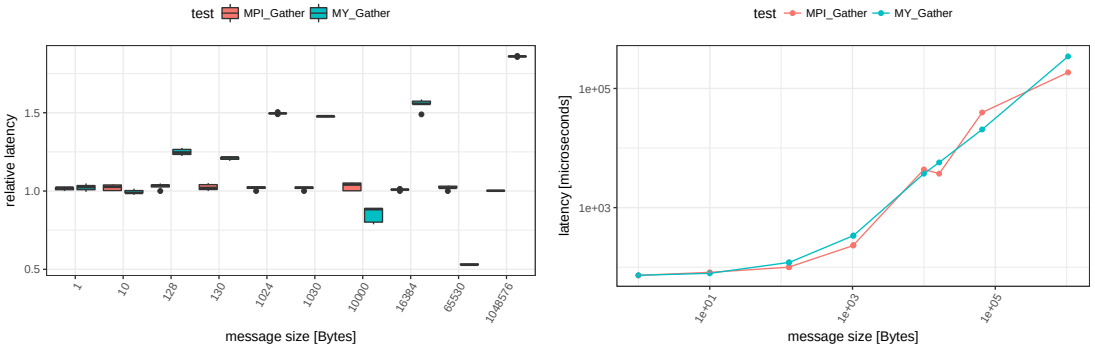
Figure 5: `MY_Gather` compared to `MPI_Gather` of `Open MPI 1.10.3`

In both test cases, either with 512 or with 496 processes the implementations of `MY_Gather` and `MPI_Gather` (`Open MPI 1.10.3`) perform similarly in most of the cases. But with message sizes > 100 KByte the `MPI_Gather` implementation outperforms `MY_Gather`. We assume `MPI_Gather` implements some optimizations when dealing with big message sizes.



(a) Relative Runtime with $p = 31 \times 16$

(b) Absolute Runtime with $p = 31 \times 16$



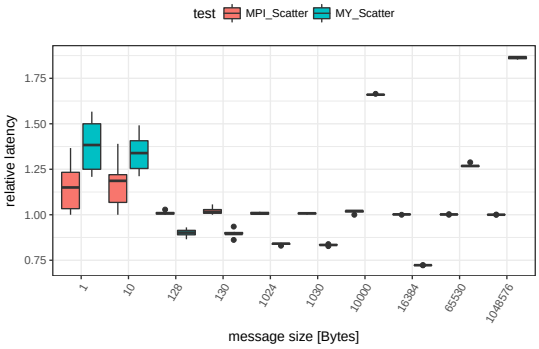
(c) Relative Runtime with $p = 32 \times 16$

(d) Absolute Runtime with $p = 32 \times 16$

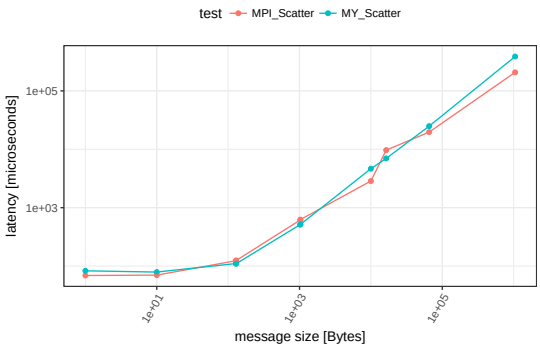
Figure 6: `MY_Gather` compared to `MPI_Gather` of `MVAPICH2 2.2`

Compared to the benchmark results before, the `MPI_Gather` implementation of `MVAPICH2 2.2` also slightly outperforms the `MY_Gather` implementations with smaller message sizes. The rather big latency changes when the message size exceeds 10 KByte is also quite interesting, we assume that `MVAPICH2 2.2` switches to another algorithm when reaching a certain threshold.

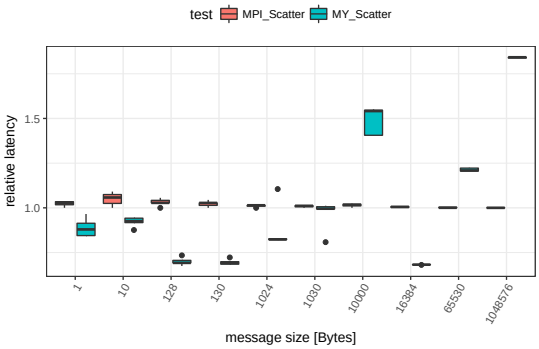
3.2 Experiments and Discussion – `MY_Scatter`



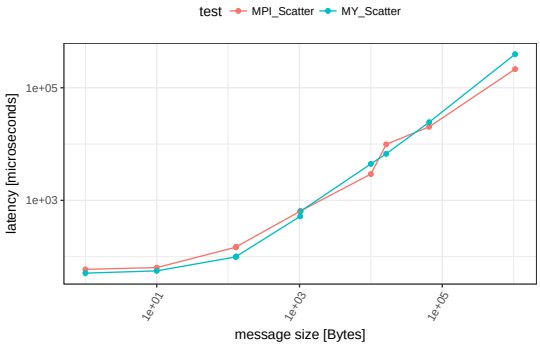
(a) Relative Runtime with $p = 31 \times 16$



(b) Absolute Runtime with $p = 31 \times 16$



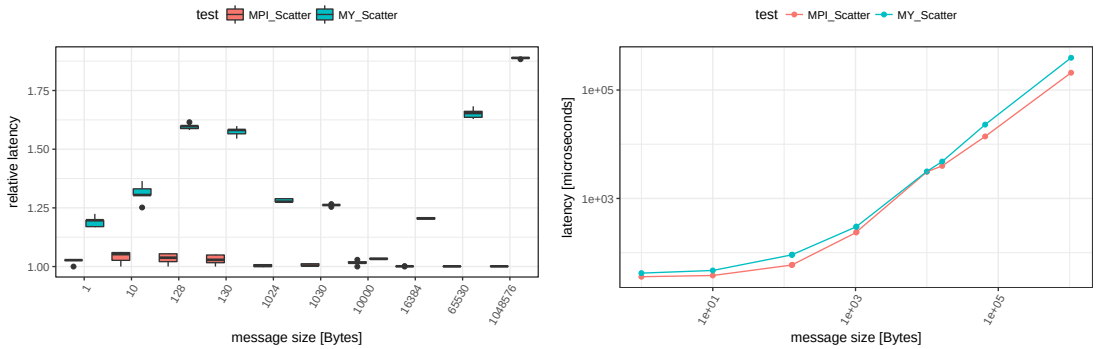
(c) Relative Runtime with $p = 32 \times 16$



(d) Absolute Runtime with $p = 32 \times 16$

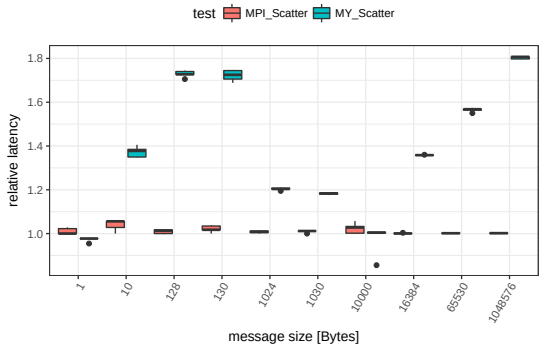
Figure 7: `MY_Scatter` compared to `MPI_Scatter` of `Open MPI 1.10.3`

Compared to the `Open MPI 1.10.3` implementation of `MPI_Scatter`, `MY_Scatter` achieves similar performance for smaller message sizes. For bigger message sizes `MPI_Scatter` outperforms `MY_Scatter`, which is again assumed to be because of optimizations for bigger messages. We haven't found a reason for the particular widespread latency distribution of `MY_Scatter` when the message size is 10 kByte, as seen in Figure 7c.

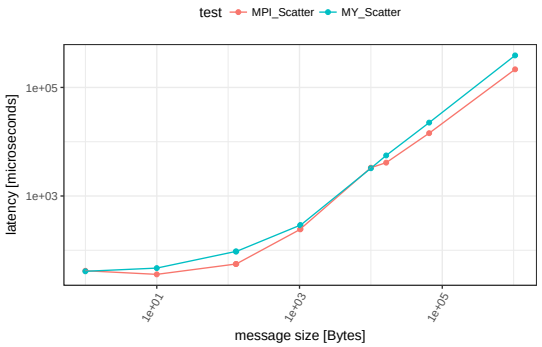


(a) Relative Runtime with $p = 31 \times 16$

(b) Absolute Runtime with $p = 31 \times 16$



(c) Relative Runtime with $p = 32 \times 16$



(d) Absolute Runtime with $p = 32 \times 16$

Figure 8: `MY_Scatter` compared to `MPI_Scatter` of `MVAPICH2 2.2`

The `MVAPICH2 2.2` implementation of `MPI_Gather` outperforms `MY_Gather` in nearly all cases. The rather big latency changes when the message size exceeds 10 KByte is also quite interesting, we assume that `MVAPICH2 2.2` switches to another algorithm when reaching a certain threshold.