Full Title*

Subtitle†

Anonymous Author(s)

Abstract

Text of abstract ....

Keywords keyword1, keyword2, keyword3

1 Introduction

Collective operations are among the most important communication operations for parallel applications running on large scale high performance computing systems. In general, all of the processes in a parallel application are involved in a collective operation either to send and/or receive data from other processes. A few widely used collective operations are: distributing identical data to all processes (i.e., broadcast), receiving different/identical data from all processes at the root process (i.e., gather/reduce) and also, to distribute the collected data (at the root) to all the processes (i.e., Allgather, Allreduce), etc. Typically, acceleration of a parallel application involves optimizing collective operations.

Many virtual topologies have been used in the past for runtime optimization of collective operations [**?**]. Among them, pipelined tree algorithms have been observed to reduce the runtime of various collectives which involves medium to large message sizes. The message to be sent (or received) is divided into small chunks and is distributed in a pipeline along the edges of the virtual topology. Binary and linear trees are commonly used as virtual topologies for medium and large message sizes respectively.

Sanders et. al. in [2] observed that in binary tree, the leaf nodes utilize only half of their bandwidth. When these nodes are receiving (in broadcast), they never send any message and while sending (in gather, reduce) no receive operation is performed. To fully utilize the bandwidth of these nodes, they proposed a two-tree based approach (referred as TwoTreeS in the paper). To perform a collective, instead of one, two binary trees are used. The inner nodes of one tree becomes the outer nodes in the other tree and hence, bandwidth of all the nodes can be fully utilized.

The construction of TwoTreeS is rather complex and depends on perfect synchronization of send and receive operations in both the trees. In each round, a process is receiving in one tree and sending to some other process in the second tree. In a large communication network, perfect synchronization of send/receive operations in both the trees is not possible because of variable number of hops among processes and also due to traffic from other applications sharing the communication infrastructure. However, this synchronization becomes an overhead and does not allow to fully optimize the bandwidth with the two trees. In this paper, we propose simple construction of two trees that does not require any synchronization and implement three widely used collectives broadcast, reduce and allreduce using the proposed two tree construction.

We make the following major contributions:

- A runtime efficient implementation of broadcast, reduce and allreduce collectives with an easy to implement two tree topology.
- A close to lower bound and stable implementation of allreduce collective for power of two and non power of two cases.
- An exhaustive experimental evaluation of the proposed collectives on state-of-the-art high performance computing (HPC) system Piz Daint. {todo} other HPC systems
- An empirical and simulator based calculation of number of chunks based on LogGP [1] model.
- {todo} Speed-up of training deep learning models using the proposed allreduce implementation.

2 Motivation

The main idea behind TwoTreeS is to maximize the bandwidth utilization in a collective. They used a pair of binary trees such that leaf nodes in one tree become inner nodes in the other one, and the message to be communicated is halved between the two trees. They also describe a scheduling algorithm through colouring of edges. Our implementation of their algorithm, as expected, performed better than the binary tree. However, we recognized several issues:

- The major drawback of TwoTreeS is the overhead of synchronization due to colouring of edges. Colouring ensures that the communication goes round by round in a synchronized way. The synchronization might add delays due to underlying network traffic. We have implemented TwoTreeS both with synchronization and without, results clearly state that the overhead of synchronization is more than its benefit.
- TwoTreeS does not ensure proper balancing of tree, which could result in reduced bandwidth utilization in cases with non power of two number of processes. Effects were evident especially in reduce and all-reduce when one sub-tree becomes significantly shorter than

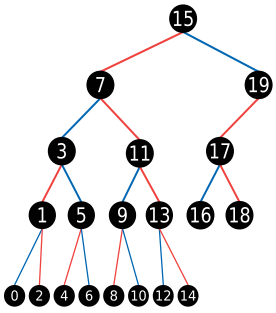
*Title note

†Subtitle note

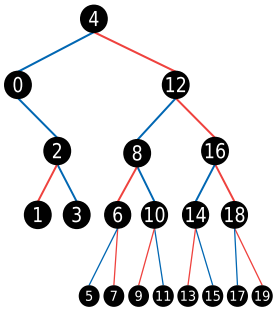
PL'18, January 01–03, 2018, New York, NY, USA
2018. ACM ISBN ...$15.00
https://doi.org/

the other. Figure 1 illustrates this imbalance for 21 processes.

- In a particular case of total $2^n + 1$ processes, the two trees are constructed such that they have a full binary tree of $2^n - 1$ processes, and one extra node at the top. This extra node thus only has one child. Hence, extra latency is added to the pipeline.
- The topology construction is quite complex. If we consider tree construction time in our analysis, then it is an overhead as well.



(a) Caption1



(b) Caption 2

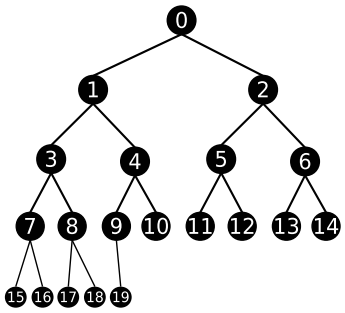
Figure 1. Caption for this figure with two images

3 Topology - Pipelined Two-Tree Complete

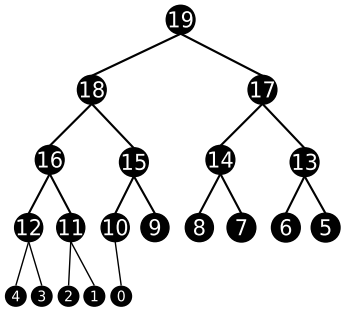
We construct two complete binary trees T_1 and T_2 each using $P - 1$ processes. These two trees are then assigned as left and right sub-tree of a root process. T_1 and T_2 are constructed as follows:

- These trees are designed in such a way that leaf nodes in one tree are the inner nodes in the other and vice-versa.
- T_1(left tree) is numbered in level wise, left to right, increasing order starting form 0.
- T_2(right tree) is numbered in level wise, left to right, decreasing order starting form $P - 2$.

Here P is the total number of processes. A TwoTreeC is shown in Figure 2 for 21 total processes. Tree Construction is shown in Algorithm 1. The trees are distributed. Each process only computes its parent and children in left tree and in right tree. Each process except root appears in both left and right trees. Therefore, $leftParent$ is used to denote parent of current process in left tree, and $rightParent$ is the parent of current process in right tree. $leftChild[]$ and $rightChild[]$ similarly denote the children of current node in left tree and in right tree respectively.



(a) Caption1



(b) Caption 2

Figure 2. Caption for this figure with two images

Algorithm 1: Two Tree Complete Construction

Require: Number of processes ← p, rank ← $processId$
1: **if** $rank = 0$ **then**
2: numberOfLeftChildren ← 1
3: numberOfRightChildren ← 1
4: leftChild[0] ← 1
5: rightChild[0] ← $p - 1$
6: **else** $\{rank \neq 0\}$
7: leftParent ← $rank/2$
8: rightParent ← $(p - \frac{p - rank}{2})\%p$
9: **if** $(2 \times rank) < p$ **then**
10: numberOfLeftChildren ← 1
11: leftChild[0]← $2 \times rank$
12: **end if**
13: **if** $((2 \times rank) + 1) < p$ **then**
14: numberOfLeftChildren ← 2
15: leftChild[1]← $((2 \times rank) + 1)$
16: **end if**
17: **if** $((2 \times rank) - p) > 0$ **then**
18: numberOfRightChildren ← 1
19: rightChild[0] ← $((2 \times rank) - p)$
20: **end if**
21: **if** $((2 \times rank) - p - 1) > 0$ **then**
22: numberOfRightChildren ← 2
23: rightChild[1] ← $((2 \times rank) - p - 1)$
24: **end if**
25: **end if**

- There is no overhead due to synchronization, because we perform no explicit synchronization. Throughout all of the implementation, all the send and receive operations used are non-blocking.
- Both the trees are balanced. This especially improves the performance of reduce and all-reduce because communication starts at the leaf nodes in both of these.
- TwoTreeC is simple to construct, it only takes O(1) time on each process.

4 Collective Operations

The TwoTreeC topology discussed in previous section 3 is used to implement Broadcast, Reduce and All-Reduce operations. This section describes how each of these is accomplished in detail.

{todo @Mohit LogGP analysis for each of these approaches.}

4.1 Broadcast

In Broadcast operation a message from root process is received at all other processes in the communicator. This root process is assumed to be the process with rank 0 and is the root of the two-tree constructed previously. The root node divides its data into a number of equal sized chunks, then one by one sends each odd-numbered chunk to the root of right tree and each even-numbered chunk to root of left tree.
Non-root processes expect odd numbered chunks from $leftParent$ and upon receiving, forward them to their $leftChild[0]$ and $leftChild[1]$. Same task is performed in right tree. In the end, each process has received all the chunks, odd ones in right tree and even ones in left tree. The algorithm to implement broadcast operation using two-tree topology is as shown in the algorithm 2

Algorithm 2: Two Tree Broadcast operation

```
Require: rank ← processId
if rank = root then
    for all Chunks do
        if Even Chunk then
            Non-blocking send this chunk to leftChild[0]
        else
            Non-blocking send this chunk to rightChild[0]
        end if
    end for
else {rank ≠ root}
    for all Chunks do
        if Even Chunk then
            Non-blocking receive this chunk from leftParent
        else {Odd Chunk}
            Non-blocking receive this chunk from rightParent
        end if
    end for
    while all chunks not received do
        Wait until any of the receives finishes
        if Even chunk received then
            Non-blocking send this chunk to leftChild[0] and leftChild[1]
        else {Odd chunk received}
            Non-blocking send this chunk to rightChild[0] and rightChild[1]
        end if
    end while
end if
Wait on all sends to finish
```

4.2 Reduce

Reduce operation starts with the leaf nodes by sending message chunks to the parent nodes. Each non-leaf node receives the chunks their child nodes and then themselves perform the reduce operation on matching chunks from all of their children and their own chunk. Then finally sends the reduced chunk to their parent node. This way the top root process receives a message that is reduced output of messages from all other processes. The algorithm to implement the same is shown in algorithm **??**.

4.3 All-Reduce

All-reduce operation can be implemented as combination of broadcast and reduce operation, in a way that first the message is reduced to root node and then the result is broadcast-ed to all other nodes. The previously stated design for the two tree was the first implementation used for all-reduce. Although good results were achieved but we realized that further improvements might be possible by making some modifications to the topology. When the exact same topology is used for both reduce and broadcast, leaf nodes are the first to finish the reduction part but are the last to receive the broadcast-ed reduced message. Hence we experimented with using different topologies for reduce phase and broadcast phase.

{todo @Sir lower bound}

4.3.1 TwoTreeC + Reordered TwoTreeC

The bandwidth of leaf nodes in reduce phase could be utilized further and so in this implementation of all-reduce operation, the reduction part uses the topology defined in algorithm 1 and for the broadcast part a new topology is defined as described in the algorithm 3. This topology is created in a way that nodes finishing the reduce part using previous topology will be closer to the root.

4.3.2 Reordered TwoTreeC

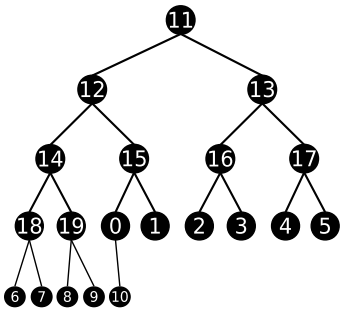
This tree is similar to the TwoTreeC(see 3) as in it consists of two complete binary trees T_1 and T_2 both constructed using $P-1$ processes, only difference is in the ordering of T_1 and T_2. The numbering is done as follows:

- T_1 is numbered in level wise, in increasing order(circular in range $1...P-1$) starting form $\left\lceil \frac{P}{2} \right\rceil$
- T_2 is numbered in level wise, in decreasing order(circular in range $1...P-1$) starting form $\left\lfloor \frac{P}{2} \right\rfloor$

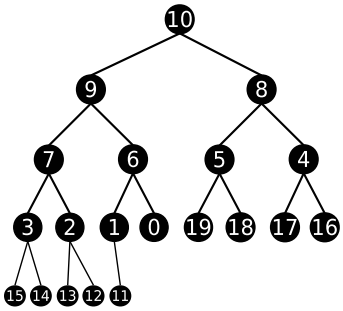
Figure 3 shows a reordered TwoTreeC for 20 processes.

4.3.3 TwoTreeC + TwoTreeS

As seen in the previous optimization, all-reduce operation can be improved using different algorithms for the sub parts



(a) T1



(b) T2

Figure 3. Reordered TwoTreeC: 20 nodes

reduce and broadcast. So similarly here this version of all-reduce operation uses the TwoTreeS for the first part *i.e.* reduce and uses the TwoTreeC for the second part *i.e.* broadcast.

5 Calculation of Chunks

Calculating the chunks value for different algorithms and for different parameters was also very challenging. We have used several methods to calculate the optimal values of the chunks for each algorithm and different number of processes.

5.1 Mathematical Approach

In the initial phase of the research when we were mostly working on the broadcast operation, we tried to calculate the number of chunks by using a mathematical formula for calculating the run time of the algorithm and by minimizing the run time. This worked well for simple enough implementations.

5.2 Simulation Tool

Then as we made different implementations and also started to analyze other operations it gets complicated and we switched our mode of calculating chunks to using the simulator. We started to calculate chunk values using the simulator tool LogGOPSim, a fast simulation framework for parallel algorithms at large-scale for calculating the optimal number of chunks. Using the simulator we got a good idea about the behaviour of our algorithms on changing the number of chunks which was very helpful in making many optimization.

Algorithm 3: Reordered Two Tree Complete Construction

Procedure FUNCRP (var)
1: **if** $var = 0$ **then**
2: **return** 0
3: **else**
4: **return** $(p - \frac{p - var}{2})\%p$
5: **end if**

Procedure FUNCONE (var)
1: **if** $var = 0$ **then**
2: **return** 0
3: **else**
4: **return** $\left(\left(((var - 1 - (\frac{p-1}{2})) + (p-1))\%(p-1)\right) + 1\right)$
5: **end if**

Procedure FUNCTWO (var)
1: **if** $var = 0$ **then**
2: **return** 0
3: **else**
4: **return** $\left(\left(((var - 1 + (\frac{p-1}{2})) + (p-1))\%(p-1)\right) + 1\right)$;
5: **end if**

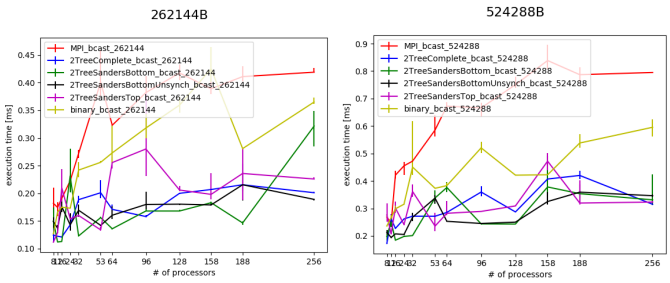
Algorithm algo()
Require: Number of Process $\leftarrow p$, rank $\leftarrow processId$
1: **if** $rank = 0$ **then**
2: number_leftChildren $\leftarrow$ 1
3: number_rightChildren $\leftarrow$ 1
4: leftChild[0] $\leftarrow$ FUNCTWO(1)
5: rightChild[0] $\leftarrow$ FUNCONE($p - 1$)
6: **else** $\{rank \neq 0\}$
7: leftParent $\leftarrow$ FUNCTWO(FUNCONE($rank$) /2)
8: rightParent $\leftarrow$ FUNCONE(FUNCRP(FUNCTWO($rank$)))
9: **if** 2× FUNCONE($rank$) $< p$ **then**
10: number_leftChildren $\leftarrow$ 1
11: leftChild[0]$\leftarrow$ FUNCTWO(2×FUNCONE($rank$))
12: **end if**
13: **if** (2× FUNCONE($rank$))+1 $< p$ **then**
14: number_leftChildren $\leftarrow$ 2
15: leftChild[1]$\leftarrow$ FUNCTWO((2× FUNCONE($rank$))+1)
16: **end if**
17: **if** (2× FUNCTWO($rank$))$-p > 0$ **then**
18: number_rightChildren $\leftarrow$ 1
19: rightChild[0] $\leftarrow$ FUNCONE((2× FUNCTWO($rank$))$-p$)
20: **end if**
21: **if** (2× FUNCTWO($rank$))$-p - 1 > 0$ **then**
22: number_rightChildren $\leftarrow$ 2
23: rightChild[1] $\leftarrow$ FUNCONE((2× FUNCTWO($rank$))$-p - 1$)
24: **end if**
25: **end if**

5.3 Experimentation

the results of the simulator was not always similar to the actual experimentation. Hence we needed a better solution. Finally we decided that instead of using one method we will use the values from both the methods and also add some more random values(+10, +20, -10, -20) so as to cover a big range and then choose the best out of them for different number of processes and message sizes. This way we insures that the values we used are much closer to the right values.

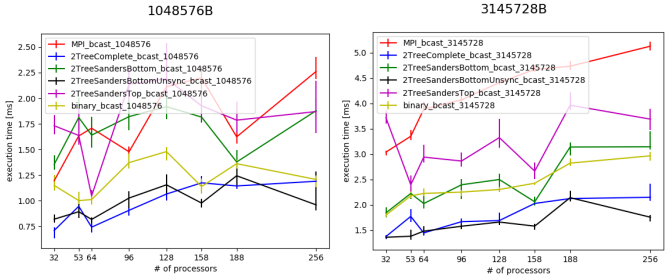
6 Experimental Configuration and Parameters

{todo @Mohit LogGP parameter values, other details like single process per node, etc.}



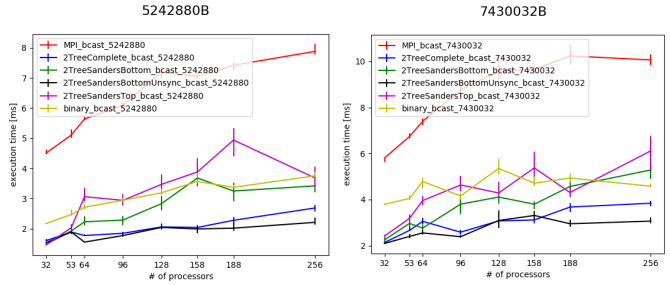
(a) Broadcast-256B **(b)** Broadcast-512B

Figure 4. Broadcast results for small sized messages



(a) Broadcast-1MB **(b)** Broadcast-3MB

Figure 5. Broadcast results for medium sized messages



(a) Broadcast-5MB **(b)** Broadcast-7MB

Figure 6. Broadcast results for medium sized messages

7 Results

Mention and briefly explain all the implementations compared: linear pipeline, pipelined binary tree, our pipelined two tree, mpi standard library, scatter-gather. Conclude how our approach outperformes the rest on larger data sizes.

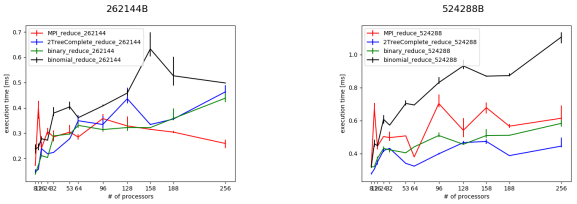
{todo @Mohit plots}

7.1 Broadcast

Message Sizes : 256B, 512B, 1MB, 3MB, 5MB, 7MB

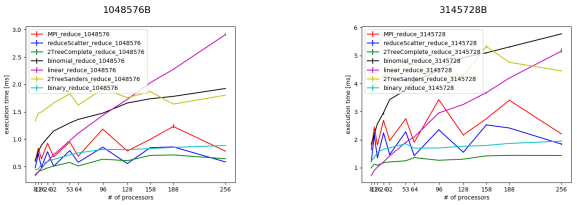
7.2 Reduce

Message Sizes : 256B, 512B, 1MB, 3MB, 5MB, 7MB



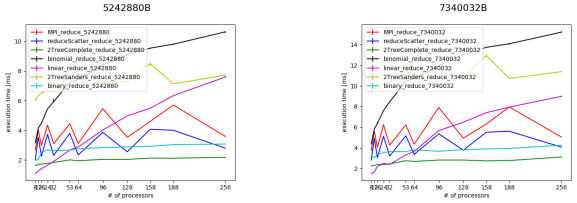
(a) Reduce - 256B **(b)** Reduce - 512B

Figure 7. Reduce results for small sized messages



(a) Reduce - 1MB **(b)** Reduce - 3MB

Figure 8. Reduce results for medium sized messages



(a) Reduce - 5MB **(b)** Reduce - 7MB

Figure 9. Reduce results for medium sized messages

7.3 All-Reduce

References

[1] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. 1995. LogGP: Incorporating Long Messages into the LogP ModelâĂŤOne Step Closer Towards a Realistic Model for Parallel Computation. In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '95)*. ACM, New York, NY, USA, 95–105. https://doi.org/10.1145/215399.215427 00547.

[2] Peter Sanders, Jochen Speck, and Jesper Larsson TrÃd'ff. 2009. Two-tree Algorithms for Full Bandwidth Broadcast, Reduction and Scan. *Parallel Comput.* 35, 12 (Dec. 2009), 581–594. https://doi.org/10.1016/j.parco.2009.09.001 00031.