MPI_Allgather Optimization with P2P Implementations

Antonio Pappalardo
Master's Student in Computer Science
University of Salerno
Fisciano, Italy
a.pappalardo36@studenti.unisa.it

Domenico Rossi
Master's Student in Computer Science
University of Salerno
Fisciano, Italy
d.rossi8@studenti.unisa.it

Francesco Parisi
Master's Student in Computer Science
University of Salerno
Fisciano, Italy
f.parisi29@studenti.unisa.it

Biagio Cosenza
Professor of Computer Science
University of Salerno
Fisciano, Italy
bcosenza@unisa.it

Majid Salimi Beni
PhD Student in Computer Science
University of Salerno
Fisciano, Italy
msalimibeni@unisa.it

***Abstract* — This document deals with the study of the performance of various algorithms that implement the collective operation Allgather.**

I. Introduction

The collective communications provided by MPI are fundamental in the exchange of information/data between the various processes. In this work we focused on the operation "Allgather". This allows each process to exchange information with all other processes, including itself. The result of this operation is the scenario in which each process will contain the data set of each process. The goal we set ourselves is to optimize the Allgather operation, as a result we compared the algorithm provided by MPI with various algorithms implemented. To compare the efficiency of each algorithm we used two different configurations, namely local testing on a Linux operating system and a cluster in the cloud of Linux machines.
The rest of this paper is organized as follows:

• Background/ Related Works
• Methodology
• Implementation
• Discussion
• Conclusions
• References

II. Background/related works

For the development of our project we have researched papers concerning the optimization of MPI's collective communication operations. One of the papers taken into consideration is "Improving the Performance of Collective Operations in MPICH [1]", which turned out to be fundamental as it allowed us to take a view and have a basic knowledge concerning the methods of optimization of the algorithms that we will discuss later. Another paper considered is "a band-saving Optimization for MPI broadcast Collective Operation [2]", it was found to be important to understand the concepts of scalability of operations through the use of the broadcasting algorithm. We then considered " Scalable Algorithms for MPI Intergroup Allgather and Allgatherv [3]", which allowed us to have a general overview of Allgather and which guaranteed a correct implementation. In addition, for further study on the topics covered, we consulted "Optimization of Collective Reduction Operations [4]", "Optimization of Collective Communication Operations in MPICH [5]" and "Optimization Principles for Collective Neighborhood Communications [6]". Finally, the concept of mapping between threads and processes was studied through the paper "Inter-Cluster Thread-to-Core Mapping and DVFS on Heterogeneous Multi-Cores [7]", useful for understanding the execution of applications on clusters with multicore architecture.

III. Methodology

In this section it will be described the used algorithms. In particular, we will see from a theoretical point of view the same structure and will analyze the methodology with which each of these algorithms performs the operation of Allgather.

1. MPI_Send-MPI_Receive

This algorithm achieves the Allgather operation by using the MPI_Send and MPI_Receive operations provided by MPI. The operation is as follows: through a cycle each process sends, through a Send, its own portion of data to all the other processes; with a second cycle each process receives, thanks to a Receive, the data of each process (including itself).

2. MPI_Gather-MPI_Bcast

This algorithm achieves the Allgather operation by using the MPI_Gather and MPI_Bcast operations provided by MPI. The operation is as follows: through the collective operation of Gather, the master process receives the data from each process; then, through the collective operation of Broadcast, the master process sends the data previously received by Gather to all processes.

3. MPI_Allgather

This algorithm performs the Allgather operation by using the native operation MPI_Allgather provided by MPI. In conclusion, to have a general overview of the performance of what we are implementing, we have performed benchmarks directly on this algorithm.

4. Ring

This algorithm performs the Allgather operation by using the Send and Receive operations provided by MPI. The operation is as follows: by means of a loop you make the rounds of the ring, where in the first round each process sends its own portion of data to the next process and receives from the previous process. The master process receives from the last process and the last process sends to the master process. From the second round of the ring, each process sends the data received from the previous round. In this way, after n round, the Allgather operation is complete, where n is the number of processes.

5. Recursive Doubling

This algorithm performs the Allgather operation by using the Send and Receive operations provided by MPI. It works with a number of processes that is a power of two. The operation is as follows: we divide the exchange between the processes into several steps, that is, in the first step it will be created subtrees whose processes have a distance of 1. As a result, the processes within the same subtree will exchange their data. In the second step, it will be created subtrees whose processes have a distance of 2 which will exchange the data with processes that have distance 2. From the following steps will be created subtrees with the same methodology, or distances equal to powers of 2. This will happen for n step, where n is the logarithm in base 2 of the number of processors.

6. Recursive Halving

This algorithm performs the Allgather operation by using the Send and Receive operations provided by MPI. It works with a number of processes that is a power of two. The operation is as follows: we subdivide the exchange between the processes in various steps, that is, in the first step it will be created subtrees whose processes have a distance equal to $2^{(\log_2 np)} - 1$, where np is equal to the number of processes. As a result, processes within the same subtree will exchange their data. In the second step, subtrees will be created whose processes have a distance of $2^{(\log_2 np)} - 2$. They will exchange data with processes that have the same distance. From the following steps it will be created subtrees with the same methodology, that is distances equal to powers of 2. This will happen for n step, where n is the logarithm in base 2 of the number of processors.

IV. Implementation

In this section we will see in detail the code of each implemented algorithm. Before this it is necessary to give a quick explanation of how the variables entered in the command line are used,the variables are the following: number of processes, size of the message, number of iterations and delay time.

Below there is the usage of variables:

```
siz = atoi(argv[1]);       //Size of the data to broadcast
iterations = atoi(argv[2]); //Number of iterations to repeat the procedure
sleep_time = atoi(argv[3]); // Wait time between iterations (microseconds)
```

1. MPI_Send-MPI_Receive

This simple algorithm generates the Allgather operation by Send-Receive.

ALGORITHM 1: MPI_Send-Receive

```
for (int iteration = 0; iteration < iterations; iteration++)
{
    if (rank == 0)
    {
        tag = 10;
        for (size_t i = 0; i < n; i++)
        {
            msg[i] = rand() % 1000;
        }
        for (int i = 0; i < numtasks; i++)
        {
            rc = MPI_Send(&msg, n, MPI_INT, i, tag, MPI_COMM_WORLD);
        }
        for (int i = 0; i < numtasks; i++)
        {
            rc = MPI_Recv(&array[i * n], n, MPI_INT, i, tag, MPI_COMM_WORLD, &Stat);
        }
    }
    else
    {
        tag = 10;
        for (size_t i = 0; i < n; i++)
        {
            msg[i] = rand() % 1000;
        }
        for (int i = 0; i < numtasks; i++)
        {
            rc = MPI_Send(&msg, n, MPI_INT, i, tag, MPI_COMM_WORLD);
        }
        for (int i = 0; i < numtasks; i++)
        {
           rc = MPI_Recv(&array[i * n], n, MPI_INT, i, tag, MPI_COMM_WORLD, &Stat);
}
```

2. MPI_Gather-MPI_Bcast

This algorithm uses Gather and Broadcast operations to realize Allgather.

ALGORITHM 2: MPI_Gather-Bcast

```
for (int iteration = 0; iteration < iterations; iteration++)
{
    for (size_t i = 0; i < n; i++)
    {
        msg[i] = rand() % 1000;
    }
   // We use the collective operation of Gather, so the process 0 have all the elements
    MPI_Gather(&msg, n, MPI_INT, array, n, MPI_INT, 0, MPI_COMM_WORLD);
   // Now we use the collective operation of Broadcast to  implement allgather
    MPI_Bcast(array, n * numtasks, MPI_INT, 0, MPI_COMM_WORLD);
    usleep(sleep_time);
}
```

3. MPI_Allgather

This simple algorithm uses the Allgather operation provided by MPI.

ALGORITHM 3: MPI_Allgather

```
for (int iteration = 0; iteration < iterations; iteration++)
```

```
{
     for (size_t i = 0; i < n; i++)
     {
         msg[i] = rand() % 1000;
     }
     value = rand() % 100;
     MPI_Allgather(msg, n, MPI_INT, array, n, MPI_INT, MPI_COMM_WORLD);
     usleep(sleep_time);
}
```

4. Ring

This algorithm uses the ring method to perform the Allgather operation. Temporary variables are used in this code to improve the performance and to reduce the access time to the memory registers.

ALGORITHM 4: Ring

```
for (int iteration = 0; iteration < iterations; iteration++)
{
     if (numtasks < 2)
     {
        for (size_t i = 0; i < n; i++)
        {
            msg[i] = rand() % 1000;
            array[i] = msg[i];
        }
     }
     else
     {
     for (int ring = 0; ring < numtasks; ring++)
     {
         if (rank == 0)
         {
             if (ring == 0)
             {
                 for (size_t i = 0; i < n; i++)
                 {
                    msg[i] = rand() % 1000;
                    array[i] = msg[i];
                 }
             rc = MPI_Send(&msg, n, MPI_INT, rank + 1, 10, MPI_COMM_WORLD);
             }
         else
         {
         int tmp = ring * n;
         rc = MPI_Recv(&array[tmp], n, MPI_INT, numtasks - 1, 10,
MPI_COMM_WORLD, &Stat);
         rc = MPI_Send(&array[tmp], n, MPI_INT, rank + 1, 10,
MPI_COMM_WORLD);
         }
    }
         else
        {
            int tmp_recv = (ring + 1) * n;
            int tmp_send = (ring)*n;
            if (ring == 0)
            {
                for (size_t i = 0; i < n; i++)
            {
            msg[i] = rand() % 1000;
            array[i] = msg[i];
        }
        if (rank == numtasks - 1)
        {
            rc = MPI_Send(&msg, n, MPI_INT, 0, 10, MPI_COMM_WORLD);
        }
        else
       {
           rc = MPI_Send(&msg, n, MPI_INT, rank + 1, 10, MPI_COMM_WORLD);
       }
       rc = MPI_Recv(&array[tmp_recv], n, MPI_INT, rank - 1, 10,
MPI_COMM_WORLD, &Stat);
    }
    else
    {
    rc = MPI_Recv(&array[tmp_recv], n, MPI_INT, rank - 1, 10,
MPI_COMM_WORLD, &Stat);
    if (rank == numtasks - 1)
    {
        rc = MPI_Send(&array[tmp_send], n, MPI_INT, 0, 10, MPI_COMM_WORLD);
    }
    else
    {
    rc = MPI_Send(&array[tmp_send], n, MPI_INT, rank + 1, 10,
MPI_COMM_WORLD);
    }
   }
  }
 }
}
```

5. Recursive Doubling

This algorithm uses the recursive doubling method to implement Allgather.

ALGORITHM 5: Recursive Doubling

```
for (int i = 0; i < nstep; i++)
{
     int p = pow(2, i);
     int a = numtasks / pow(2, i + 1); // Number of subtrees
     int k = numtasks / a;             // Number of items in the subtree
     int st = (rank / k) + 1;          // Subtree considered
     int j = rank + p;                 // Leap forward
     int tmp = n * p;
     if (j >= st * k) // if the process j is not in subtree considered
     {
         j = rank - p; // Jump back
     }
     rc = MPI_Send(array, tmp, MPI_INT, j, 10, MPI_COMM_WORLD);
     rc = MPI_Recv(&array[tmp], tmp, MPI_INT, j, 10, MPI_COMM_WORLD, &Stat);
     }
     usleep(sleep_time);
}
```

6. Recursive Halving

This algorithm uses the recursive halving method to implement Allgather.

ALGORITHM 6: Recursive Halving

```
for (int iteration = 0; iteration < iterations; iteration++)
{
     for (size_t i = 0; i < n; i++)
     {
         msg[i] = rand() % 1000;
         array[i] = msg[i];
     }
     for (int i = 0; i < nstep; i++)
     {
         int p = pow(2, nstep - (i + 1));
         int a = numtasks / pow(2, nstep - i); // Number of subtrees
         int k = numtasks / a;                 // Number of items in the subtree
         int st = (rank / k) + 1;              // Subtree considered
         int j = rank + p;                     // Leap forward
         int tmp = n * pow(2, i);
         if (j >= st * k) // if the process j is not in subtree considered
         {
             j = rank - p; // Jump back
         }
         rc = MPI_Send(array, tmp, MPI_INT, j, 10, MPI_COMM_WORLD);
         rc = MPI_Recv(&array[tmp], tmp, MPI_INT, j, 10, MPI_COMM_WORLD, &Stat);
     }
         usleep(sleep_time);
}
```

V. DISCUSSION

In this section we will give a general overview of the obtained results by following a careful performance analysis. We tested the algorithms on two different hardware configurations:

1) Local: Asus Computer with Intel Processor(R) Core I7-10750H@2.60GHz, Architecture: x86_64, RAM 16 Gb, 6 cores, 12 logical processors, Cache L1 cache: 384 Kb, L2 cache: 1.5 Mb, L3 cache:12.0 Mb and Ubuntu Operating System 20.04 LTS.
2) Cluster: 8 virtual machines with Intel Processor(R) Xeon(R) CPU @2.20GHz, Architecture: x86_64, each with RAM 16 Gb, 4 vcpu, L1 cache: 64 Kb, L2 cache: 512 Kb, L3 cache: 55 Mb and Ubuntu Operating System 18.04 LTS.

An important aspect considered for the benchmarking phase concerns the mapping of threads. The built cluster is not oriented to obtain the best performance, as different types of processing cores or special optimizations are not offered for the mapping of threads to specific cores, but the parallel calculation is managed by the Linux kernel process scheduler. Therefore, strategies for mapping threads to different processing units were not considered, being aware of the possibility of balancing all available cores and allocating a specific amount of work, thus achieving a further improvement in terms of computing time. The benchmarking was carried out by testing the developed application five times, obtaining the estimated calculation time and the relative median was calculated on the basis of the total times acquired. During the various tests we realized that the Recursive Doubling and Recursive Halving algorithms had incalculable execution times for messages with a size of 512 Kb.

Consequently, it was not possible to test three configurations:

- Locally with 32 processes and data size equal to 2^6 integers;
- Locally with 32 processes and data size equal to 2^7 integers;
- Locally with 16 processes and data size equal to 2^7 integers;

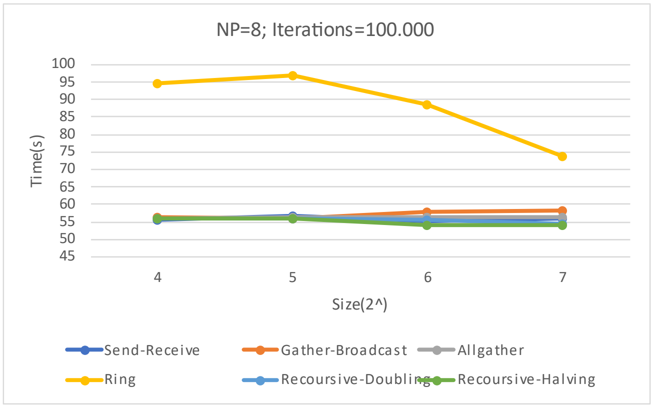

Figure 1: Local benchmarking with 8 processes and 100.000 interactions

In this graph (Figure 1), where 8 processes were used and 100.000 iterations were made, we can see the efficiency of the various algorithms used locally. In particular, as deducible from the diagram (Figure 1), the algorithm Ring is the one with most discordant performances from the other algorithms. We can also note that, by increasing the size, the execution time of all the other algorithms remains rather constant, albeit slightly increasing. In conclusion, we can affirm that the algorithm Ring, in the face of an increase in size, improves meaningfully (From size 5 to 7) its own execution times; vice versa, the other algorithms endure a decrease in performance against a size increase.

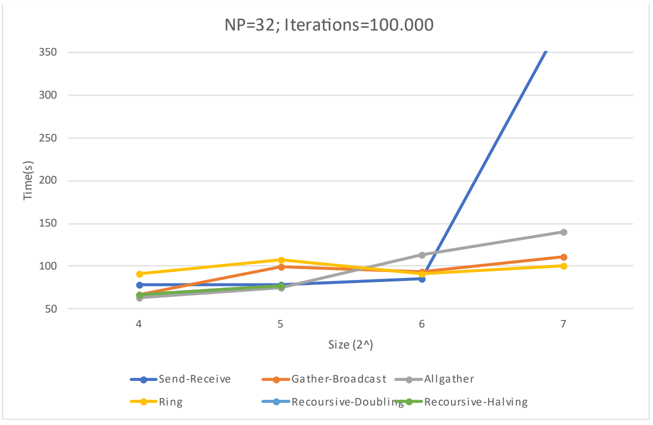

Figure 2: Local benchmarking with 32 processes and 100.000 interactions

In this graph (Figure 2), by using 32 processes and with 100.000 iterations, we can see the efficiency of the various algorithms used locally. The first observation that emerges from the graph (Figure 2) is that the Recursive Doubling and Recursive Halving algorithms stop at size 5: this is because in local execution times are incalculable. It's also noticeble that up to size 6, the algorithm Send-Receive remains more or less constant, while with an increase in size, the execution time increases dramatically.

In conclusion, we can say that, from size 6 on, the algorithms Gather-Broadcast and Ring are better in terms of efficiency than the others.

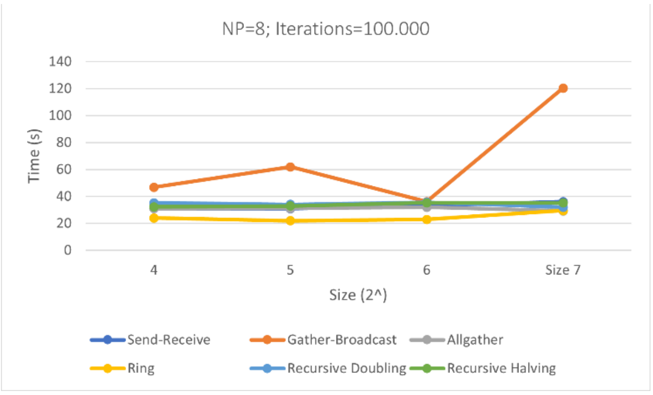

Figure 3: Benchmarking on a cloud cluster with 8 processes and 100.000 iterations

In this graph (Figure 3), by using 8 processes and with 100.000 iterations, we can see the efficiency of the various algorithms used by exploiting a cluster in the cloud. The first observation that emerges from the graph (Figure 3) is that apart from Gather-Broadcast, the other algorithms are quite similar, in terms of execution times. It can be seen that, while the size increases, Gather-Broadcast has a deterioration in performance, even quite dizzying.

In conclusion, we can therefore say that, by exploiting the cluster, all algorithms have a fairly balanced performance, also by virtue of the increase in size.

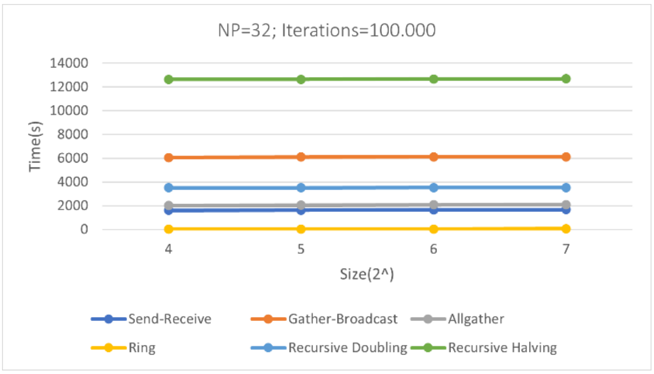

Figure 4: Benchmarking on a cloud cluster with 32 processes and 100.000 iterations

In this graph (Figure 4), by using 32 processes and with 100.000 iterations, we can see the efficiency of the various algorithms used by exploiting a cluster in the cloud. The first observation that emerges from the graph (Figure 4) is that there is a constancy in all the algorithms used, although their execution times differ significantly from each other. It can be noted that the Recursive Halving has a very high execution time, the Gather-Broadcast follows immediately, along with Recursive Doubling. Then there are Allgather and Send-Receive that have almost identical execution times.

VI. CONCLUSIONS

On the basis of what we have seen in the previous paragraph we can draw some conclusions. As for the local tests, there are some algorithms that behave better with the increase in size than the native Allgather algorithm and vice versa there are some that worsen. As for cluster tests, consistency prevails, both for algorithms with better times and vice versa.

A final overview is as follows:

The Ring algorithm both locally and on clusters, with the size and processes increase, is more powerful than the native Allgather algorithm.

VII. REFERENCES

[1] R. Thakur, W. Gropp, Improving the Performance of Collective Operations in MPICH, 2003.

[2] H. Zhou, V. Marjanovic, C. Niethammer, J. Gracia, A Bandwidth-saving Optimization for MPI Broadcast Collective Operation. International Symposium on Information Technology, 2010.

[3] Q. Kang , J. L. Träff, R. Al-Bahrani, A. Agrawal, A. Choudhary, W. Liao, Scalable Algorithms for MPI Intergroup Allgather and Allgatherv. Parallel Computing, Pages 220-230, 2019.

[4] R. Rabenseifner, Optimization of Collective Reduction Operations. International Conference on Computational Science, 2004.

[5] R. Thakur, W. Gropp, R. Rabenseifner, Optimization of Collective Communication Operations in MPICH. International Journal of High Performance Computing Applications,Volume 19, 2005.

[6] T. Hoefler, T. Schneider, Optimization Principles for Collective Neighborhood Communications, 2012.

[7] K. R. Basireddy, D. Biswas, A. Singh. Inter-Cluster Thread-to-Core Mapping and DVFS on Heterogeneous Multi-Cores. IEEE Transactions on Multi-Scale Computing Systems, September 2017.