MPI All-to-All optimization

Alessandro Oliviero
Computer Science departement
University of Salerno
Fisciano (SA), Italy
a.oliviero9@studenti.unisa.it
0522501083

Ciro Perfetto
Computer Science departement
University of Salerno
Fisciano (SA), Italy
c.perfetto1@studenti.unisa.it
0522501054

Roberto Veneruso
Computer Science departement
University of Salerno
Fisciano (SA), Italy
r.veneruso1@studenti.unisa.it
0522501060

***Abstract*—We present efficient algorithms for all-to-all communication operations in message-passing systems. We implemented some of Bruck's algorithm, such as index and radix-r and also the version with derived datatypes and zero copy mechanism. We furthermore show how the improved algorithm can be used to solve regular all-to-all communication problems. As a baseline to demonstrate the performance advantages of these algortihm, we used the basic function *MPI_Alltoall* of the OpenMPI library. We also discuss the time complexity of the algortihm and the performance in terms of weak scalability.**

***Index Terms*—Analysis of algorithms, All-to-all personalized communication, complete exchange, index operation, message-passing systems, parallel system.**

I. Introduction

Collective communication operations are communication operations that generally involve more than two processors, as opposed to the point-to-point communication between two processors. In this paper, we consider MPI collective communication routines, where multiple nodes participate in the communication operation. There are inherent limitations in the current implementations of MPI collective communication routines. For example, since the library routines are implemented before the topology information is known, it is impossible for the library to utilize topology specific algorithms. The performance of communication operations is a function of their latency and bandwidth costs. Latency is the fixed cost per communication step, which is independent of communication size, whereas bandwidth is the transfer time per byte. Typically, short-message communication is dominated by latency, while long-message communication is dominated by bandwidth. Short-message exchange, therefore, yields better performance with fewer underlying communication steps. The Bruck algorithm [3] is a well-known technique to reduce the total number of these internal communication steps in an All-to-all exchange from P to $\log_2(P)$. This is achieved by transmitting an overall larger amount of data, but over a smaller number of iterations. In this paper, we survey and implement some variants of the Bruck algorithm for uniform data loads and a topology specific algorithm. In particular, we implement the Bruck classic index and radix-r, the modified Bruck algorithm with derived MPI datatype and its version with zero copy.

II. Algorithms

A. Index

This algorithm achieves the AllToAll operation using the naive Bruck Algorithm without any datatype. Let P be the number of MPI processes, each bound to a processor or core. The processes are numbered (ranked) from 0 to $P-1$. Each process has an individual data element to each other process, including an element to itself. The element from process $i, 0 \leqslant i < P$, to process $j, 0 \leqslant j < P$, is denoted m_{ij}.

Basic Bruck has three steps, the second of which involves communication. Each process has a $p-element$ vector R where the elements received so far are stored, and from which the elements to send in the next communication round are also taken. Upon termination, $R[j]$ for process i shall store the element m_{ji} from process j to process i. The p processes carry out the same operations with process i, $0 \leqslant i < p$, doing the following:

- Local shift towards index 0 by i indices.
- Global communication step with $\log_2(P)$ rounds.
- Local reverse and shift towards index $P-1$ by $i+1$ indices.

The complexity of time for this algorithm [4] for P processes and elements of size n is:

- $\log_2(P)$ communication rounds
- $[\log_2(P)][P/2]n$ units of data sent and received per process
- $P + 2[\log_2(P)][P/2] + 2P$ local element copy/unpack/pack operations.

We will show some pseudocode.

Algorithm 1 Index

```
1:  procedure ALLTOALL_INDEX
2:      loop:
3:      //Local rotation
4:      index ← (i-rank+nprocs) mod nprocs
5:      recv_buf[index] ← send_buf[i]
6:      end-loop
7:      //copy blocks which need to be sent at this step
8:      temp_buffer ← send_buf + offset
9:      //send and receive data from other process
10:     MPI_SendRecv(send_proc, recv_proc)
11:     //Replace with received data
12:     send_buf + offset ← recv_buf
13:     //Second rotation
14:     recv_buf ← send_buf
15: end procedure
```

B. Radix r

This algorithm belongs to the class of algorithm for the index operation but has a parameter r to change the base rapresentation of number during the interprocessor communication phase. Each processor-memory configuration has n columns of n blocks each. Columns are labeled from 0 through $n-1$ and blocks are labeled from 0 through $n-1$. All the algorithms in the class consist of three phases:

- Each processor p independently rotates its n datablocks i steps upwards in a cyclical manner.
- Each processor p rotates its j_{th} data block j steps to the right in a cyclical manner. This rotation is implemented by interprocessor communication.
- Each processor p independently rotates its n data blocks i steps downwards in a cyclical manner.

The implementation of Phases 1 and 3 on each processor involves only local data movements and is straightforward. In the sequel, we

focus only on the implementation of Phase 2. Different algorithms are derived depending on how the communication pattern of Phase 2 is decomposed into a sequence of point-to-point communication rounds. We present the decomposition of Phase 2 into a sequence of point-to-point communication rounds, assuming the oneport model and using a parameter r (for radix) in the range $2 \leqslant r \leqslant n$. For convenience, we say that the block-id of the jth data block in each processor after Phase 1 is j. Consider the rotation required in Phase 2. Each block with a block-id j in processor i needs to be rotated to processor $(i+j)modn$. The block-id j, where $0 \leqslant j \leqslant n-1$, can be encoded using radix-r representation using $w = \log_r(n)$. For convenience, we refer to these w digits from zero through $w-1$ starting with the least significant digit. Phase 2 consists of w subphases corresponding to the w digits. Each subphase consists of at most $r-1$ steps, corresponding to the (up to) $r-1$ different non-zero values of a given digit. In subphase x, for $0 \leqslant x \leqslant w-1$, we iterate Step 1 through Step $r-1$, as follows:

- During Step z of subphase x, where $1 \leqslant z \leqslant r-1$ and $0 \leqslant x \leqslant w-1$, all data blocks, for which the xth digit of their block-id is z, are rotated $z \times r^x$ steps to the right. This is accomplished in a communication round by a direct point-to-point communications between processor i and processor $(i + z \times r^x)modn$, for each $0 \leqslant i \leqslant n-1$.

The following points are made regarding the performance and time complexity of this algorithm:

- Each step can be realized by a single communication round by packing all the outgoing blocks to the same destination into a temporary array and sending them together in one message. Hence, each subphase can be realized in at most $r1$ communication rounds.
- The size of each message involved in a communication round is at most $b \times \frac{n}{r}$ data.
- Hence, the class of the index algorithms has complexity $b \times (r-1) \times \frac{n}{r} \times \log_r(n)$

We will show some pseudocode.

Algorithm 2 Radix_r part 1

```
1: procedure ALLTOALL_RADIX_R
2:     //Calculate the number of digits when using r-representation
3:     w ←log(n_procs)/ log(r)
4:     loop:
5:     //Local rotation
6:     recv_buf ← send_buf[rank]
7:     recv_buf[n_procs − rank] ← send_buf[rank]
8:     end-loop
9:     //Convert rank to base r representation
10:    r_rep ← convert10tob(w, i, r)
11:    //Communication steps = (r - 1)w - d
12:    loop:
13:    temp_buf[index] ← recv_buf[i]
14:    end-loop
15:    //Exchange data using temp_buf
16:    MPI_SendRecv(send_proc, recv_proc)
17:    //Replace with received data
18:    recv_buf + offset ← send_buf+i
```

Algorithm 3 Radix_r part 2

```
19:    //Replace with received data
20:    index ← (rank-i+nprocs) mod nprocs
21:    send_buf[index] ← recv_buf[i]
22:    //Second rotation
23:    recv_buf ← send_buf
24: end procedure
```

C. Datatypes

The algorithm by Bruck [3] (originally implemented with explicit, hand-written packing and unpacking) is an exemplary candidate for the use of MPI derived datatypes. The advantage is a cleaner implementation that separates the algorithmic idea from data reorganization issues that are otherwise handled by customized packing and unpacking code. Depending on how well the MPI library implements the datatype mechanism and interacts with the communication system, a better performing implementation may be the added benefit. Let P be the number of MPI processes, each bound to a processor or core. The processes are numbered (ranked) from 0 to $P-1$. Each MPI process stores the elements m_{ij} to the other processes consecutively in sendbuf. Elements can be arbitrarily structured by the sendcount and sendtype arguments. The received elements will be stored in $recvbuf$ and can likewise be arbitrarily structured, independently of the structure of the elements to be sent; however, all sent and all received elements have the same structure and size. MPI requires data in $sendbuf$ to remain unchanged after the operation, but $recvbuf$ can be used for the R vector, as long as the structure of the data to be received is respected. In order to implement Modified Bruck we introduce a new datatype constructor that is the $MPI_Type_create_indexed_block$, it is an indexed block type where each element is indexed separately. The complexity of time for this algorithm [4] for P processes and elements of size n is:

- $\log_2(P)$ communication rounds
- $[\log_2(P)][P/2]n$ units of data sent and received per process
- $P+2[\log_2(P)][P/2]$ local element copy/unpack/pack operations.

Now we will show some pseudocode.

Algorithm 4 Datatype

```
1: procedure ALLTOALL_DATATYPE
2:     loop:
3:     //Local rotation
4:     index ← (2*rank-i+nprocs) mod nprocs
5:     recv_buf[index] ← send_buf[i]
6:     end-loop
7:     loop:
8:     //Create datatype
9:     MPI_Type_create_indexed_block
10:    //Exchange data
11:    MPI_SendRecv(send_proc, recv_proc)
12:    end-loop
13:    //Copy received data
14:    recv_buf ← send_buf
15: end procedure
```

D. Zero copy

The previous implementations receive elements into an in termediate buffer which is then unpacked into the recvbuf before the next communication round. It would be desirable to eliminate this overhead. For instance, a received element which will have to be sent further on in a later communication round could remain in and be sent directly out of the intermediate buffer with no need for unpacking into the recvbuf. In general, elements for which the number of set bits $k' > k$ in j is even will be received into recvbuf, and elements with an odd number of set bits $k' > k$ will be received into the intermediate buffer. We use the same communication and storage pattern as in Datatype algorithm. When a new element for $R[i+j]$ is received by process i in round k, the kth bit of j is set. In the same round,

element $R[i+j]$ is sent. We store the elements to be sent and received alternatingly in the $recvbuf$ and an intermediate buffer. If j has no further bits $k' > k$ equal to one, the element is at its destination process and should be received directly into position $i+j$ in $recvbuf$. In general, elements for which the number of set bits $k' > k$ in j is even will be received into $recvbuf$, and elements with an odd number of set bits $k' > k$ will be received into the intermediate buffer. Conversely, elements j with an even number of set bits $k' > k$, will be sent out of the intermediate buffer, elements j with an odd number of set bits after k out of the recvbuf. Finally, in each round k, the first element of each segment of $2k$ elements is a "new" element, and taken from position $(i-j)$ mod p of $sendbuf$. The time complexity is the same of Datatype algorithm. Now we will show some pseudocode.

Algorithm 5 ZeroCopy

```
1:  procedure ALLTOALL_ZERO_COPY
2:      loop:
3:      //Local rotation
4:      index ← (2*rank-i+nprocs) mod nprocs
5:      recv_buf[index] ← send_buf[i]
6:      end-loop
7:      //Initial data to recv_buffer and intermediate buffer
8:      temp_buf ← recv_buf
9:      //Copy received data
10:     send_buf ← recv_buf
11:     mask ← 0xFFFFFFFF
12:     loop:
13:     //Check if the number of bits k' > k is odd
14:     if (bits[jmask]0x1) == 0x1) then
15:         recvindex ← recv_buf + index
16:     end if
17:     //if k' > k is even, data received from send_buf, temp otherwise
18:     if (jmask) == j then
19:         sendindex ← send_buf + index
20:     else if (jmask)! = j then
21:         sendindex ← temp_buf + index
22:     end if
23:     if (bits[jmask]0x1)! = 0x1) then
24:         //Send to intermediate buffer with same step of lines 15-21
25:     end if
26:     end-loop
27:     //Create datatype to send and receive
28:     MPI_Type_create_struct(sendblocktype)
29:     MPI_Type_create_struct(recvblocktype)
30:     //Exchange data
31:     MPI_SendRecv(send_proc, recv_proc)
32:     end-loop
33:     //Exchange data with created datatype
34:     MPI_SendRecv(send_proc, recv_proc)
35:     end-loop
36: end procedure
```

III. Experimental Setup

We conduct a thorough evaluation of our algorithms using a pesonal computer with an Intel i7 8750h, a CPU with 6 core 12 thread, so we tested our All-to-all algorithms with a cap of 12 processes with 12 elements each. Each execution was repeated 1000 times to have a mean and a median more accurate.

IV. Performance results

For each algorithm implemented, and also baseline, was calculated mean and median of execution time, incrementing the number of processor and the number of data sent. We tested the running times maintaing same number of processor with different sizes and with a maximum number of 8 processor because we stick to a power of 2 number both for process and size. We used int type of data for each algorithm except for datatype and zerocopy, so data sent in terms of byte has to be multiplied by 4. We started, as a small size, with 2^4 elements to a maximum of 2^{20}.

A. Baseline

TABLE I
MEAN EXECUTION TIME (MSEC) OF BASELINE IMPLEMENTATION.

Processes/Size	2^4	2^8	2^{12}	2^{16}	2^{20}
2	0,00227	0,00276	0,00933	0,02364	0,77244
4	0,00485	0,05925	0,02033	0,05232	1,70120
8	0,02301	0,02432	0,02778	0,17713	3,59822

TABLE II
MEDIAN EXECUTION TIME (MSEC) OF BASELINE IMPLEMENTATION.

Processes/Size	2^4	2^8	2^{12}	2^{16}	2^{20}
2	0,00200	0,00240	0,00710	0,01880	0,71365
4	0,00440	0,00520	0,01950	0,04350	1,61465
8	0,01100	0,01370	0,02005	0,13310	3,37355

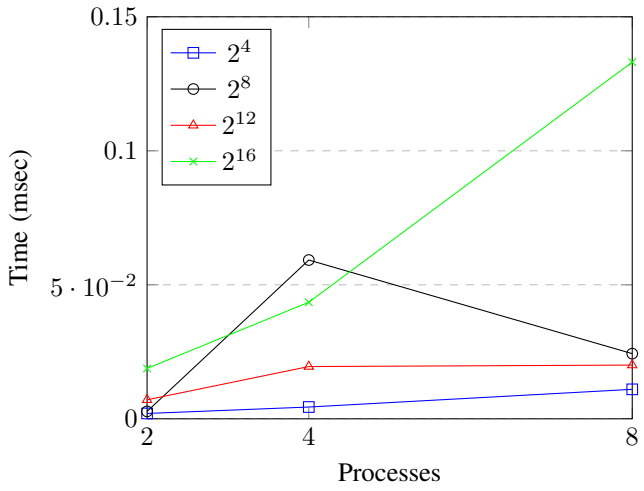

Fig. 1. Median time confrontation of baseline algorithm with different sizes.

B. Index

TABLE III
MEAN EXECUTION TIME (MSEC) OF INDEX IMPLEMENTATION.

Processes/Size	2^4	2^8	2^{12}	2^{16}	2^{20}
2	0,00215	0,0028769	0,009068	0,185213	4,82568
4	0,0039735	0,004964	0,018269	0,237754	11,1509
8	0,0098687	0,0220963	0,0523621	0,499971	26,6716

TABLE IV
MEDIAN EXECUTION TIME (MSEC) OF INDEX IMPLEMENTATION.

Processes/Size	2^4	2^8	2^{12}	2^{16}	2^{20}
2	0,0018	0,0025	0,0083	0,17425	4,35995
4	0,0033	0,0044	0,0156	0,21775	10,3298
8	0,0084	0,0148	0,0395	0,46095	25,8257

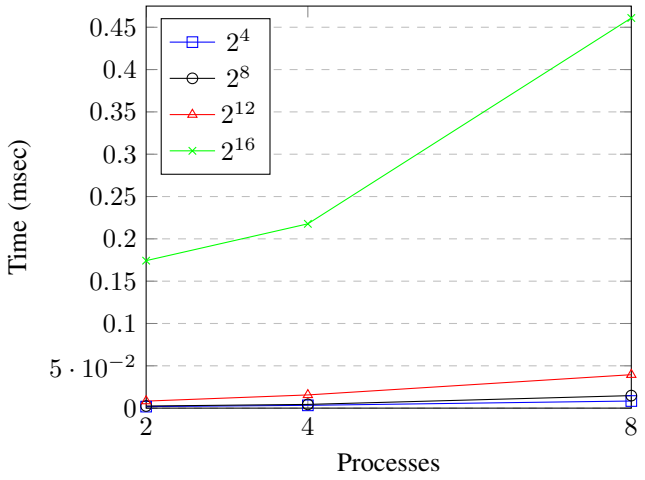

Fig. 2. Median time confrontation of index algorithm with different sizes.

C. Radix r

For this algorithm we used $r = 4$ as a factor of adjustment during execution.

TABLE V
MEAN EXECUTION TIME (MSEC) OF RADIX R IMPLEMENTATION.

Processes/Size	2^4	2^8	2^{12}	2^{16}	2^{20}
2	0,0033146	0,0039633	0,0121615	0,179386	4,49045
4	0,0063554	0,0065434	0,025497	0,217357	8,77907
8	0,0195996	0,021806	0,062701	0,517783	23,5523

TABLE VI
MEDIAN EXECUTION TIME (MSEC) OF RADIX R IMPLEMENTATION.

Processes/Size	2^4	2^8	2^{12}	2^{16}	2^{20}
2	0,0028	0,0036	0,0108	0,1672	4,3144
4	0,0056	0,0057	0,02265	0,1968	8,37325
8	0,0134	0,0158	0,0443	0,45775	22,6371

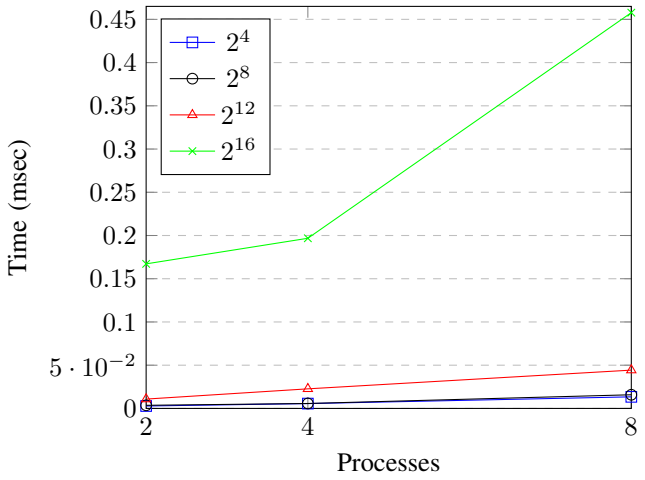

Fig. 3. Median time confrontation of radix r algorithm with different sizes.

D. Datatypes

TABLE VII
MEAN EXECUTION TIME (MSEC) OF DATATYPE IMPLEMENTATION.

Processes/Size	2^4	2^8	2^{12}	2^{16}	2^{20}
2	0,0038708	0,0044836	0,0114812	0,0481739	1,54891
4	0,0074947	0,0082108	0,0187981	0,120305	4,63056
8	0,0189832	0,0209	0,0528248	0,261129	13,7511

TABLE VIII
MEDIAN EXECUTION TIME (MSEC) OF DATATYPE IMPLEMENTATION.

Processes/Size	2^4	2^8	2^{12}	2^{16}	2^{20}
2	0,0033	0,0039	0,0093	0,03815	1,2992
4	0,0063	0,0069	0,0169	0,11215	4,3932
8	0,0148	0,0158	0,0464	0,21365	13,7419

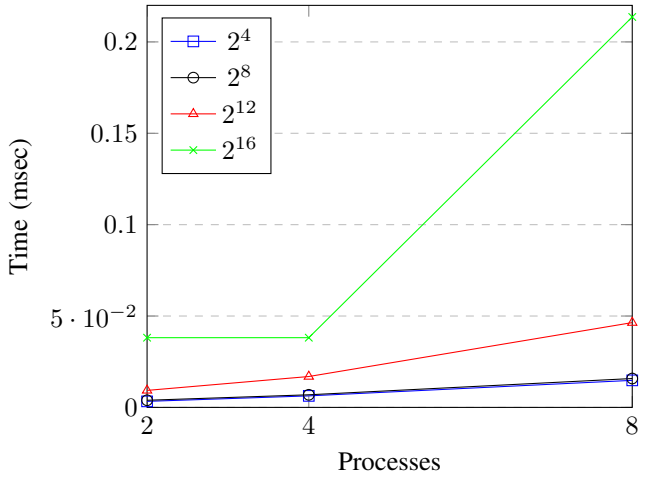

Fig. 4. Median time confrontation of datatype algorithm with different sizes.

E. Zero copy

TABLE IX
MEAN EXECUTION TIME (MSEC) OF ZERO COPY IMPLEMENTATION.

Processes/Size	2^4	2^8	2^{12}	2^{16}	2^{20}
2	0,0041155	0,004977	0,011716	0,069347	2,85352
4	0,0084503	0,0084196	0,017853	0,160593	6,62429
8	0,020777	0,037745	0,051019	0,411449	16,7783

TABLE X
MEDIAN EXECUTION TIME (MSEC) OF ZERO COPY IMPLEMENTATION.

Processes/Size	2^4	2^8	2^{12}	2^{16}	2^{20}
2	0,0035	0,0046	0,0115	0,05485	2,68345
4	0,0083	0,0075	0,0164	0,1213	6,3566
8	0,0167	0,0226	0,0434	0,34175	15,4588

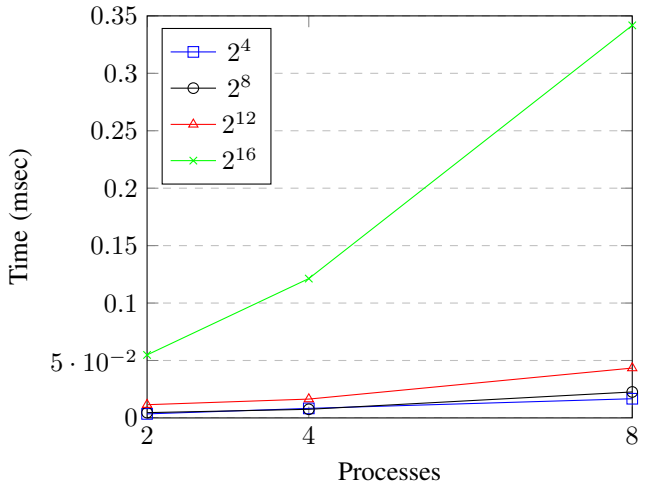

Fig. 5. Median time confrontation of zero copy algorithm with different sizes.

F. Large data

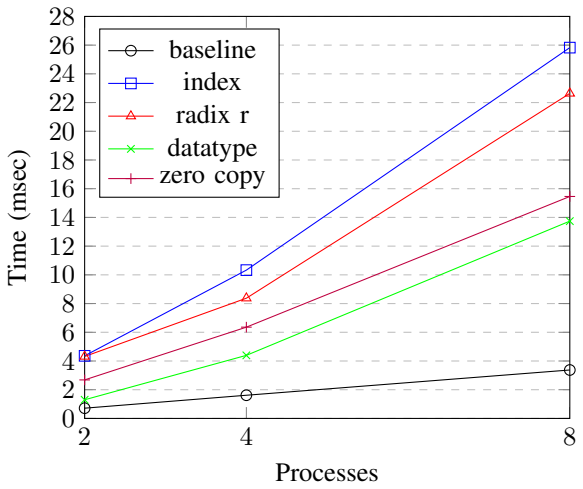

Fig. 6. Median time confrontation of all algorithm with 2^{20} elements for each processor.

V. Final Consideration

TABLE XI
MEDIAN EXECUTION TIME (MSEC) COMPARATION FOR ALL IMPLEMENTATIONS WITH 2 PROCESSES.

Processes/Size	2^4	2^8	2^{12}	2^{16}	2^{20}
Baseline	0,00200	0,00240	0,00710	0,01880	0,71365
Index	0,0033	0,0044	0,0156	0,21775	10,3298
Radix r	0,0028	0,0036	0,0108	0,1672	4,3144
Datatype	0,0033	0,0039	0,0093	0,03815	1,2992
Zero copy	0,0035	0,0046	0,0115	0,05485	2,68345

TABLE XII
MEDIAN EXECUTION TIME (MSEC) COMPARATION FOR ALL IMPLEMENTATIONS WITH 4 PROCESSES.

Processes/Size	2^4	2^8	2^{12}	2^{16}	2^{20}
Baseline	0,00440	0,00520	0,01950	0,04350	1,61465
Index	0,0033	0,0044	0,0156	0,21775	10,3298
Radix r	0,0056	0,0057	0,02265	0,1968	8,37325
Datatype	0,0063	0,0069	0,0169	0,11215	4,3932
Zero copy	0,0083	0,0075	0,0164	0,1213	6,3566

TABLE XIII
MEDIAN EXECUTION TIME (MSEC) COMPARATION FOR ALL IMPLEMENTATIONS WITH 8 PROCESSES.

Processes/Size	2^4	2^8	2^{12}	2^{16}	2^{20}
Baseline	0,01100	0,01370	0,02005	0,13310	3,37355
Index	0,0084	0,0148	0,0395	0,46095	25,8257
Radix r	0,0134	0,0158	0,0443	0,45775	22,6371
Datatype	0,0148	0,0158	0,0464	0,21365	13,7419
Zero copy	0,0167	0,0226	0,0434	0,34175	15,4588

We chose the median of execution times because on high number of iteration there were some outlayers and a long tail that augmented the mean values, creating a non normal distribution. In the tables $11, 12, 13$ are shown the perfomance of all algorithms with every size we used. We can notice that with 4 processor every algorithm used is approximately 1.5 times faster than the baseline except for the 2^{20} size. In almost every other case the baseline performs better both on little and big size, in fact we can see that with 8 process and 2^{20} elements, baseline is 4.8 times faster compared to the fastest algorithm performing All-to-all communication, that is the datatype one. As we expected, baseline would still perform better because test were executed in a local environment. So as a channel of communication it was used the local bus and processor have a shared memory (UMA); these algorithm were implemented for distributed architecture(NUMA) with network based communication channel. With larger amount of data bandwith is crucial, so the latency introduced with all the algorithm we presented led to poor communication times in some cases, but still, with a little number of processor and not exessively large number of data we can see some improvements.

References

[1] K. Fan, T. Gilray, V. Pascucci, X. Huang, K. Micinski and S. Kumar, “Optimizing the Bruck Algorithm for Non-uniform All-to-all Communication”.

[2] A. Faraj and Xin Yuan, “Automatic Generation and Tuning of MPI Collective Communication Routines”.

[3] J. Bruck, C. Ho, S. Kipnis, E. Upfal and D. Weathersby, “Efficient Algorithms for All-to-All Communications in Multiport Message-Passing Systems”.

[4] J. L. Träff, A.Rougier and S. Hunold, “Implementing a Classic: Zero-copy All-to-all Communication with MPI Datatypes”.