Lossy all-to-all exchange for accelerating parallel 3-D FFTs on hybrid architectures with GPUs

Sébastien Cayrols [1], Jiali Li [1], George Bosilca [1], Stanimire Tomov [1], Alan Ayala [1], and Jack Dongarra [1,2]

[1]Innovative Computing Laboratory - University of Tennessee, Knoxville, TN, USA
[2]Oak Ridge National Laboratory, Oak Ridge, TN, USA

2022 IEEE International Conference on Cluster Computing (CLUSTER) | 978-1-6654-9856-2/22/$31.00 ©2022 IEEE | DOI: 10.1109/CLUSTER51413.2022.00029

***Abstract*—In the context of parallel applications, communication is a critical part of the infrastructure and a potential bottleneck. The traditional approach to tackle communication challenges consists of redesigning algorithms so that the complexity or the communication volume is reduced. However, there are algorithms like the Fast Fourier Transform (FFT) where reducing the volume of communication is very challenging yet can reap large benefit in terms of time-to-completion. In this paper, we revisit the implementation of the MPI all-to-all routine at the core of 3D FFTs by using advanced MPI features, such as One-Sided Communication, and integrate data compression during communication to reduce the volume of data exchanged. Since some compression techniques are 'lossy' in the sense that they involve a loss of accuracy, we study the impact of lossy compression in heFFTe, the state-of-the-art FFT library for large scale 3D FFTs on hybrid architectures with GPUs. Consequently, we design an approximate FFT algorithm that trades off user-controlled accuracy for speed. We show that we speedup the 3D FFTs proportionally to the compression rate. In terms of accuracy, comparing our approach with a reduced precision execution, where both the data and the computation are in reduced precision, we show that when the volume of communication is compressed to the size of the reduced precision data, the approximate FFT algorithm is as fast as the one in reduced precision while the accuracy is one order of magnitude better.**

I. Introduction

The Fast Fourier Transform (FFT) is a performance critical algorithm used in many applications such as PDE simulations and solvers, fast convolution, molecular dynamics, and many others. By essence, the FFT is a collection of orthogonal transformations that makes this operation numerically stable. With large 3D problems, the parallelization of the FFT is a necessity. The classical way for computing a 3D FFT is to perform a succession of 1D FFTs in each dimension, interleaved with transpose of the data across the dimension. In the general case, the original data may be placed in a domain decomposition fashion across the MPI processes, as illustrated on Figure 1, Left. Thus, in this case, prior to the computation in the first direction, a collective communication phase, called *reshape*, is required in order to have the data for complete 1D vectors redistributed to individual processes, as illustrated in the second (from Left) domain redistribution. This step is repeated for all the remaining directions. Finally, one last redistribution can place the data back into their original location. In the case where the original and final data are placed differently, the number of reshapes can be reduced. However, we consider in the following the general case of four reshapes, as depicted in Figure 1. Each reshape is a collective MPI communication operation where a subset of the processes communicates with another subset. This general case suits well the use of the MPI_Alltoallv routine, a generalized all-to-all.

The inherent sequentiality of the algorithm makes its performance challenging to improve. While most efforts on improving the FFT performance target the computation part of the algorithm, very few consider the communication aspect despite its critical importance. For example, in a prior study [1], authors ported the entire computation on the GPUs to obtain a $42\times$ speedup (vs. using multicore CPUs). The problem is that between each computation of the 1-D FFTs there is an all-to-all communication that cannot be accelerated in a similar way, and thus the communication is bound to become a growing bottleneck. Indeed, the same authors report that with a large number of nodes, more than 95% of the runtime is spent in communication, and as a result any effort to further optimize the computation part will have a minimal impact on the overall time-to-solution of the algorithm. Moreover, since the problem can hardly be balanced in all dimensions, the amount of data to transfer can vary from one destination to another, and as a result improving the overall performance is even more challenging [1]. This is a fundamental problem and can be observed in many FFT libraries, e.g., refer to performance report [2] for a compilation of the FFT libraries available in the literature, as well as their performance benchmarking and comparison.

A wide-spread use of the FFT is as a preconditioner, in which case the accuracy of the method could be reduced without impacting the overall accuracy of the application. One example of such an approach is the use of mixed-precision iterative refinement methods to accelerate dense linear systems solvers [3]. There the most expensive part of the solve is the factorization of a given matrix A. Therefore, computing the factorization of A in a lower precision, can give a significant speedup because modern hardware compute faster on lower number of bits, as highlighted in Table I. Then, a process of refinement is needed to obtain the originally desired accuracy. When applicable, the use of mixed-precision solvers leads to significant speedup [3]. In the context of FFT, using mixed-precision in the local 1D FFT computations would only give a limited overall benefit, as the local FFTs are already highly optimized using GPUs, and therefore already take only a

2168-9253/22/$31.00 ©2022 IEEE
DOI 10.1109/CLUSTER51413.2022.00029

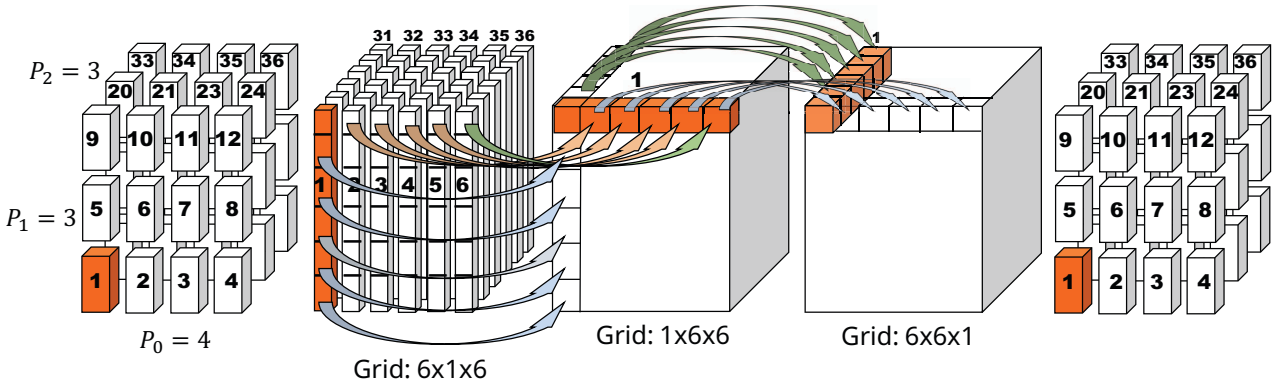

Fig. 1. Data movements in the x, y and z direction of the 3D FFT algorithm using MPI_Alltoallv, on a 3D grid of $4 \times 3 \times 3$ processors.

Arithmetic	Size (bits)	Range			Unit round-off	Peak Tflop/s	
		$x_{\min,s}$	$x_{\min}$	$x_{\max}$		V100	MI100
BFloat16	16	9.2×10^{-41}	1.2×10^{-38}	3.4×10^{38}	3.9×10^{-3}	N/A	92
`FP16`	16	6.0×10^{-8}	6.1×10^{-5}	6.6×10^{4}	4.9×10^{-4}	125	184
`FP32`	32	1.4×10^{-45}	1.2×10^{-38}	3.4×10^{38}	6.0×10^{-8}	15.7	23
`FP64`	64	4.9×10^{-324}	2.2×10^{-308}	1.8×10^{308}	1.1×10^{-16}	7.8	11.5

TABLE I
PARAMETERS FOR THE IEEE `FP16`, `FP32`, AND `FP64` ARITHMETIC PRECISION, AND THEIR RESPECTIVE PEAK PERFORMANCES ON NVIDIA V100 AND AMD MI100 GPUS.

few percents of the overall time. Thus, other approaches are needed, including the one pursued in this paper to develop fast approximate FFTs by reducing the FFT's communication costs through accelerated GPU-aware MPI integrated with on-the-fly data compression and decompression.

Our contribution in this paper is three-fold. First, we designed an approximate FFT algorithm (described in Section III) that reduces the volume of communication through lossy compression while controlling the accuracy. Second, we apply compression techniques on the data using GPUs before each communication, reducing the cost of communication, and thus the overall execution time in Section IV. Finally, we designed and implemented a very efficient all-to-all MPI algorithm for GPU-direct communications using one-sided communication scheme in Section V. All these developments were tested through the open source state-of-the-art heFFTe library for 3D FFTs [4], providing fast approximate 3D FFTs that can control the accuracy through user-specified error tolerance. Experimental setup and results are given in Section VI, and conclusion in Section VII.

II. RELATED WORK

There has been a great deal of work related to different components of the approach that we pursue in this paper. For example, efforts on development and optimizations of multidimensional FFTs have been put in heFFTe [1], leading to a very efficient implementation of the 3-D FFT. Data compression has also been intensively studied mainly driven by the power of GPUs. Libraries like ZFP [5] and SZ and their use in many applications [6] provide efficient implementations of lossy and/or lossless compression on both CPU and/or GPU. However, the use of data compression to minimize FFT communications has not yet been studied.

Similarly, mixed-precision methods have been intensively studied in various areas [7]. One relevant and interesting idea for the particular case of enabling FP16 acceleration of FFTs is to dynamically split a FP32 vector into two scaled FP16 vectors, apply the FFT transformations on the two vectors using GPU Tensor Cores acceleration, and combine back the results into an FP32 vector [8].

As a major user of large all-to-all communication, several studies investigate minimizing the communication impact on the FFT. In general they propose to change the collective algorithm to order the communications in a way to alleviate the network congestion, or to take advantage of specific network topologies, or specific network or NIC capabilities. One solution [9] leverages on off-loadable network interface, and designs a non-blocking all-to-all by enabling lists of operations over the interface. Leveraging on high-speed network resources like NVLink to overlap communication latency has been studied by [10]. Similarly, a recent study [11] has optimized all-to-all communication by offloading certain operations onto specialized NIC, such as SmartNIC. Besides relying on special hardware capabilities, study [12] proposed to allocate shared buffer for send and receive, and use Morton order to guide memory copies and thus maximize the memory bandwidth. More general studies investigate taking advantage of architecture awareness to achieve better communication

performance. Study [13] proposed kernel-assisted mechanisms for multi-core architectures, to improve collective operations. A parallel MPI software package [14] presents an implementation of the truncated Tucker decomposition, aiming to compressing distributed data. When considering architecture of dense GPU clusters, a GPU-based on-the-fly compression technique [15] integrated in MVAPICH2 library is introduced. To accelerate MPI communication, an approximate-communication scheme [16] has been proposed. [17] presents a study of several compression algorithms that can be used for run-time message compression, based on the datatype used by applications. Library [18] extends state-of-the-art MPI libraries with non-blocking (asynchronous) operations and low-precision data representations features. Additionally, improving MPI reduction with the combination of OpenMP and data compression is proposed in [19].

There are many multidimensional FFT Libraries for distributed-memory systems, including AccFFT [20], FFTE [21], fftMPI [22], heFFTe [1], 2Decomp&FFT [23], nb3DFFT [24], FFTW [25], SWFFT [26], and FFTADVMPI [27]. We choose to show the developments through heFFTe because heFFTe is open source state-of-the-art library when compared to the other FFT libraries, and is the only one providing support across the different GPUs from NVIDIA, AMD, and Intel [4].

III. An approximate FFT with lossy compression

We consider the approximate 3D FFT given in Algorithm 1. Note that if the compression is lossy, we propose to control the error within an error tolerance e_{tol}, resulting in an approximate FFT algorithm with controlled error.

Algorithm 1 Approximate 3D FFT with lossy compression.

Input : 3D data $D_{x,y,z}$ in FP64 precision and error tolerance e_{tol}

Output: Approximate 3D FFT of $D_{x,y,z}$ in FP64 precision

1: **for** $i := x, y, z$ **do**
2: Custom Alltoall (Algorithm 3) with compression of data $D_{x,y,z}$ in direction i
3: 1D FFTs for direction i in FP64
4: **end for**

Approximate FFTs have a wide use in applications that must guarantee a solution within a certain error. For example, FFTs are used in spectral methods to solve PDEs [28]. The general steps to solve a PDE with these methods, e.g., $-\nabla u + u = f$ in $\Omega = [0..L]$, where f is a smooth function and periodic on the boundary, is given in Algorithm 2. The main computational kernels that need acceleration are the forward FFT (step 2) and the inverse FFT (step 4), which can be done in $O(N \log N)$ time using FFTs, *vs.* for example $O(N^3)$ if a dense direct solver is applied. See also [29] for a comparison of FFT-based solvers to other best known methods like FMM and multigrid.

Algorithm 2 Solve $-\nabla u + u = f$ in $\Omega = [0..2\pi]$ using FFTs.

Input : function f, smooth and periodic on the boundary

Output: solution u

1. Sample $f[i] = f(x_i)$ at N grid points $x_i = i*h$, $h = 2\pi/N$ and error tolerance e_{tol}
2. Compute g = FFT (f, e_{tol})
3. Scale g point-wise, $g[i] = g(i)/(1 + (ih)^2)$
4. Compute u = IFFT(g, e_{tol})

To show the need for the approximate (including mixed-precision) FFTs that we propose, and to simplify their error analysis, we generalize the solver in Algorithm 2 as

$$Ax = b$$

. We recognize that there are various errors associated in this approach. Most notably, there are the discretization errors going from the continuous PDE problem to a discrete problem, $A_h x_h = b_h$, using FFTs of size N ($h = O(1/N)$), and round-off errors associated with solving the discrete problem in some finite precision arithmetic. These types of approximation errors can be subject of detailed study and evaluation, e.g., cf. [30]. Most notably, when multiple sources of errors are involved, the total approximation error e_a can be represented as:

$$e_a = x - \tilde{x}_h = (x - x_h) + (x_h - \tilde{x}_h) = e_d + e_r.$$

Thus, the error e_a (e.g., in certain computable quantities or directly in some norm $||\cdot||$ of interest) can be bounded by the maximum of the discretization error (e_d) and the round-off error (e_r):

$$||e_a|| \leq 2\ max(||e_d||, ||e_r||).$$

In other words, if a user requires a solver with a guaranteed error below e_{tol}, the $\|e_d\|$ and $\|e_r\|$ errors must be balanced, i.e., approximately the same, and made to be the largest possible that are still below the target e_{tol}, otherwise there will be inefficiencies and thus missed opportunities for acceleration. For example, if $e_{tol} = 10^{-5}$, $\|e_d\|$ must be made about that, e.g., through control of the number of discretization points N, and $\|e_r\|$ as well, e.g., through control of the accuracy in the FFT computations. Obviously, in this example, the use of FP32 arithmetic would have been sufficient in order to solve with enough accuracy, while being about $2\times$ faster than an FP64 solver achieving the same overall accuracy.

The e_d error for spectral methods can be shown to be of order up to h^N, leading to fastest possible so-called "exponential convergence", when the solution is smooth enough. In general, the FFT user must know some bounds for the error e_d and pass that value as the e_{tol} for the approximate FFT. If the user does not know it, we can propose error control based on a posteriori error analysis, similar to techniques used in FEM methods [31], using the approximate solutions on different grids to deduce an error estimate (or the value of P that makes the rate of computed convergence h^P). Further in the paper, we assume that the user knows e_d and passes it as e_{tol}, and therefore will not be concerned any longer with

the discretization error. This is a common assumption and is part of the API for many numerical libraries such as sparse iterative solvers.

Related to controlling the error due to compression and round-off, FFT is an orthogonal transformation, so truncating the input will result in roughly the same error in the output, e.g., casting to `FP32` removes about eight decimal digits of accuracy from the input, resulting in losing eight decimal digits from the output. There is also some accuracy lost due to round-off arithmetic, e.g.:

$$\frac{||e_r||}{||\tilde{x}_h||} \leq \kappa(A) \frac{||\tilde{b}_h - \tilde{A}_h \tilde{x}_h||}{\tilde{b}_h},$$

where $\kappa(A)$ is the condition number of A w.r.t. the norm $||\cdot||$. Since A is related to FFT here, we consider $\kappa(A)$ to be one, illustrating the above observation that the relative error in the output is bounded by the error in the input, since the typical magnification factor $\kappa(A)$ is one. Round-off errors are still present though from the simple operations in the FFTs (sin, cos, and dot-product computations). They can be bounded by $1.06(2N)^{2/3}\epsilon$ for DFT and $1.06\sum_j (2p_j)^{2/3}\epsilon$ for FFT, where p_j are the prime factors of N and ϵ is the working machine precision [32].

IV. Compressing the communication

In order to reduce the cost of communication in the FFT algorithm, we propose compressing the data that will be exchanged. The choice of compression technique is critical as it will determine the performance of the collective exchange as well as the accuracy of the resulting FFT algorithm.

A. Compression techniques

We consider different types of compression techniques, from lossy to lossless. On one extreme is *truncation*, a casting-like operation that is highly efficient due to the hardware support provided by modern architectures. The truncation corresponds to a change in the number of bits used for its representation. For example, let us consider a floating-point value with a 64-bits representation, i.e., double-precision, namely `FP64`. When its representation is truncated so that only 32-bits, i.e., single-precision, are used, commonly referred to as casting from `FP64` to `FP32`, we obtain a compression rate of two. The larger the number of trimmed bits, the greater the compression rate. However, this increases the potential loss because a larger truncation leads to a smaller range of floating-point representation, as presented in Table I. In the end, the choice depends on the data and algorithmic properties of the target application, which improves communication performance with stronger compression, and the desired accuracy.

At the other extreme are techniques that rely on more sophisticated algorithms to offer more flexibility and a variable accuracy that allows both lossy and lossless compression [5], [6], [33]–[35]. However, the absence of hardware support and a much higher computational complexity compared with truncation leads to lower efficiency. Fortunately, these techniques offer other advantages. For example, the library ZFP [5], which provides lossless and lossy compression, offers the possibility to control either the accuracy, or the compression rate, with both CPU and/or GPU support. But, a good compression rate requires special properties for the data to compress. By properties the authors refer to spatial correlation as to the relation between groups of values. Therefore, if this condition is satisfied, it is possible to compress at a fixed compression rate, say a compression rate of two, and then to decompress with a maximum error that is lower compared with a truncation of the same compression rate (such as from `FP64` to `FP32`). Otherwise, in the case of random data, sophisticated techniques like ZFP would behave similar to truncation operations.

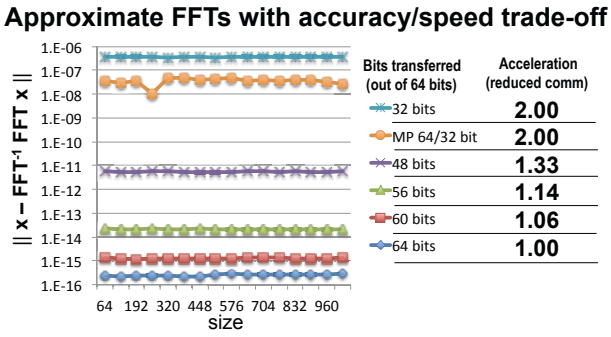

Fig. 2. Evolution of the accuracy of the FFT algorithm with respect to the number of bits in the mantissa.

B. Impact of truncating the mantissa on the FFT accuracy

In the following, we study the evolution of the accuracy of the FFT when the mantissa is trimmed. This is a compression where the error is controlled by the number of bits trimmed. The accuracy of the FFT is given by the norm of the difference between the input problem and the inverse of the FFT, i.e., $\|x - IFFT\,(FFT\,(x))\,\|$.

We consider as reference 64-bit FP numbers and trim them down to 32 bits, which is the `FP64` trimmed down to `FP32` representation. Figure 2 shows the impact of reducing the number of bits as well as the theoretical acceleration obtained by reducing the volume of communication. We first note that the accuracy for 64 bits is around the double-precision machine precision ($\approx 1e-16$), and, for 32 bits, around the single-precision machine precision ($\approx 1e-8$), as expected. We observe that the more the mantissa is trimmed the closer the accuracy is to the 32 bits accuracy.

Now, if we do the computation in double-precision but the communication in single-precision as in the proposed approximate FFT Algorithms 1, referred to as MP 64/32 in the figure, the accuracy is about an order of magnitude better than with 32 bits. This means that, compared with 64 bits, the overall execution will be accelerated twice while simultaneously having a better accuracy than executing everything at the lower 32-bits precision.

In the following, we focus on the truncation operations as this allows us to predict the gain (due to a constant compression ratio) as well as provides a lower bound on

accuracy. Unless explicitly mentioned, we consider in the following two truncation operations: double-precision (`FP64`) to single-precision (`FP32`), and double-precision (`FP64`) to half-precision (`FP16`), which gives us a compression rate of two and four, respectively. Therefore, applying it in the context of FFT, we expect a compression rate of two to give a speedup very close to two when communication represent most of the execution. Thus, our performance model for compression is that the overall performance increases at the rate of the data compression. We confirm in the experimental results section that performance indeed gets very close to these theoretically modeled results.

V. Compressed All-to-all using one-sided semantics

An all-to-all operation is a collective operation that exchanges the same amount of data between each pair of involved processes, and where each process has different data to send to every other processes. A classic implementation of this collective communication pattern is the pairwise algorithm, also known as the *ring algorithm*. Formally, for p processes involved, the completion of the all-to-all operation takes p steps, including the step for the local data transmission, i.e., a process sending data to itself. At step j, each process P_i, $i \in \{1, \ldots, p\}$, sends data to process $(P_i + j)\%p$. Consequently, at each step, each process sends and receives one message of same size to and from different processes. The main interest of this algorithm is to ensure a constant, bi-directional traffic for each process, by saturating network resources between processes.

As an extension to the classical algorithm for platforms with hierarchical resources where multiple processes are placed on the same node, such as multi-GPU nodes, or with specialized network topologies, such as fat tree or dragonfly, it is possible to create a permutation of ranks, such that the communications with ranks distance $permute[j]$ will minimize network congestion and potentially maximize the network utilization. This means that no two nodes (or the processes placed on them) will send or expect to receive data from the same remote node (or processes placed on it), such that all available network in each direction is, at any moment, only used between two nodes.

From an implementation perspective, the classical way to implementing this algorithm is to use two-sided communication. This means a handshake happens for each point-to-point communication, imposing an unnecessary overhead for such a synchronous algorithm. It is true that when the messages are large enough the cost of this handshake will be insignificant, but our goal in this paper is to create a pipeline between the compression and the data transmission, allowing us to, simultaneously, take advantage of the compute power of the processor to compress the next fragment while the previous fragment is moved through the network. This means we need to split each message in many smaller fragments, increasing the impact of this unnecessary handshake. To remove this overhead, we implement the architecture-aware ring-based collective by replacing all two-sided point-to-point communication with their one-sided equivalent as presented in a simplified form in Algorithm 3.

A. Revisit of the All-to-all algorithm using one-sided

In order to replace the two-sided point-to-point communication by the one-sided equivalent, each process P_i needs to expose its received buffer, named *recvbuf*, to all other processes (Line 3). By doing so, it gets a window that contains all information needed for managing RMA operations. It must be noted that the window creation is a collective operation and therefore has a high cost. However, when the all-to-all is performed multiple times on the same memory fragment, it is possible to cache this window, and thus reduce the startup cost of the all-to-all implementation. Then, after a synchronization phase to make sure all processes are ready, the ring algorithm starts. For each destination, the calls to MPI_Send (MPI_Isend) is replaced by the RMA operation MPI_Win_put which starts the communication (Line 8). Similarly to MPI_Isend, this operation is asynchronous and therefore requires to wait the completion of pending communication, i.e., the sending and reception of the data for P_i. Last, each process reaches the global synchronization (Line 11) needed to ensure all communication in the window are now completed at both the origin and the target, and thus the data is available in the user buffer everywhere.

Algorithm 3 Classical ring version of the OSC_Alltoall

Require: Same parameters as classical MPI_Alltoall
Ensure: *recvbuf* the buffer that contains the result of the data exchange

```
1:  Let n be the number of nodes
2:  Let k be the node id of the current rank
3:  Let win be the window that exposes recvbuf
4:  for j = 1 to n do
5:      n_j = (j + k)%n
6:      for i = 1 to #processes of node n_j do
7:          Let dest = permute[n_j][i] be the next target
            // Pipelining between compression and transfer
8:          Put the data into dest memory using win
9:      end for
10:     Wait the completion of all data movements
11: end for
```

It must be noted that because the put operation is asynchronous, Algorithm 3 behaves as if all communication have been posted upfront, using non-blocking communication, and they were all completed right before returning from the function. This might not be the best implementation on real platforms, as it will insert, almost in same time, a storm of messages in the network increasing the opportunity for collisions, and rerouting, and thus decreasing the achievable network bandwidth.

B. Integration of the compression

To accelerate our all-to-all, we propose to compress the data put in the network. The integration of the compression in

Algorithm 3 corresponds to add two steps. The first step is the compression of the data to be sent to *dest* just before the put into the destination memory (Line 8). As a consequence, the call to MPI_Win_put is made on the compressed data and so the target memory is filled with compressed data. This introduces the second step after the global synchronization (Line 11) which decompresses the received data.

However, to comply with the general requirements of MPI API, the compression of the data cannot be done inplace, because the send buffer of the all-to-all is constant. Thus, our algorithm needs two internal buffers: one to store the result of the compression for a destination, and the second to receive the compressed data, both buffers size depending on the compression technique used. Therefore, the memory exposed to the other processors is now the second internal buffer and not *recvbuf*.

We want to emphasize that the compression plays a similar role as packing and unpacking operation in MPI in the case where the data are not contiguous in memory. Indeed, the resulted compressed data residing in the internal buffer are contiguous and so MPI will not use another internal buffer. Note that communication relies on the GPU-direct for better performance.

In order to hide the cost of compression, we pipeline it with the communication, by carefully taking advantage of the sequential order of operations in a CUDA stream. To do so, the routine starts by splitting the data into chunks and submits a kernel for each chunk on the same stream. However, instead of using CUDA events to track the completed kernels, we simply call a second kernel on the same stream to update a memory location that indicates the current status of the compression. Thus the communication of the compressed chunks can be triggered by the CPU by watching the updates of the shared counter. The sequentiality of the execution of the kernels ensures that the value of the counter corresponds to the number of chunks compressed and that can be safely put into the target memory. On the target side, since we are using the one-sided communication scheme, no action is needed for the reception of the data. However, on the target side we could have also created a pipeline between the decompression operation and the next put, but the RMA API lacks efficient constructs for this. Thus instead of a pipeline on the target side, we will decompress the entire buffer later, once communications complete.

Our implementation leads to a total cost of the compressed transfer equals to the cost of the compression of the first chunk plus the communication of the compressed data. We observed in practice, with truncation operation, that this execution cost is very close to the upper bound given by the communication cost of uncompressed data divided by the compression rate.

VI. Experimental results

In this section, we evaluate our approach: i) by comparing the performance of our OSC_Alltoall with the classical MPI_Alltoall routine; ii) by comparing the accuracy and performance of our approximate FFT with two executions of heFFTe in `FP64` and `FP32`, respectively.

Given that in general, the compression technique used is application dependent [33], we consider random data which validates our approach as well as offers a good estimate of how our method performs. As stated in Section IV-A, with random data, sophisticated compression techniques are not relevant. We therefore use truncation operation in the remainder of the paper. It allows us to show the performance of our approach in the case where the compression technique is cheap (compared with sophisticated techniques) as well as offers the possibility to control the compression rate.

We did all our experiment on the Summit supercomputer, located at Oak Ridge National Laboratory. The machine consists of $4\,608$ dual-socket nodes, with each socket having three GPUs and 21 cores. Each node has two Infiniband lanes for a total theoretical bandwidth of 25GB/s. We evenly map one MPI process per GPU, which means six MPI processes per node. We use the following software stack: GCC 8.3.1, Open UCX 1.10, CUDA 10.1.234, and Open MPI 5.0 master[1].

A. Performance comparison of the OSC_Alltoall

In order to understand the quality of OSC_Alltoall we want to compare its performance with the classical all-to-all implementation. However, to the best of our knowledge confirmed by our experiments, the latest Spectrum-MPI 10.4 doesn't have support of one-sided communication to/from GPU memory. Therefore, we only test OSC_Alltoall using Open MPI, and use Open UCX as the communication engine. Hence, this section compares OSC_Alltoall with the default Open MPI MPI_Alltoall. Note that because the Summit supercomputer has two lanes per node, we select for each socket the closest network device to handle communication, which is mlx5_3:1 and mlx5_0:1 for socket 0 and 1, respectively.

Figure 3 compares the network bandwidth per node for the two implementations. In this experiment, each process sends to each other process $80KB$ of data. Thus, when there are 1536 GPUs involved, the total amount of data sent, and therefore received, by each process is $1536 * 80 = 122880$ KB. We note that for a small number of GPUs, both implementations achieve similar bandwidth. But when the number of GPUs increases, the performance of the default all-to-all decreases rapidly to reach around 5GB/s. Indeed, the bandwidth for the intra-node communication (50GB/s) being higher than the bandwidth of the inter-node communication (25GB/s), the overall bandwidth of the all-to-all decreases as the proportion of data sent outside of the node becomes dominant. For example, for 24 processes, the volume of intra-node communication is a quarter of the overall volume of communication, and thus artificially increases the perceived average bandwidth. However, when the number of GPUs increases, the fraction of the volume of intra-node communication decreases and so the average bandwidth decreases to its inter-node average bandwidth. On the other hand, OSC_Alltoall benefits as expected from the use of one-sided communication and offers twice the

[1]SHA:fafbb3702

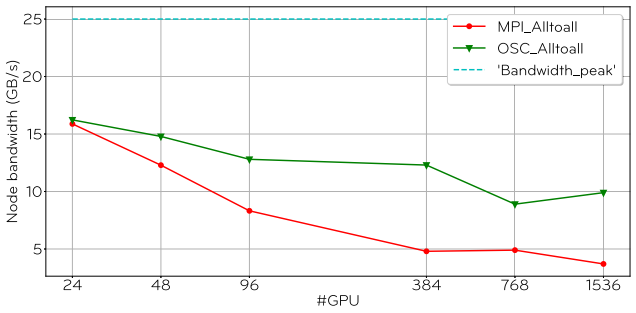

Fig. 3. Average node bandwidth usage for the different all-to-all implementations with an increasing number of GPUs, with the fixed message size per process of $80KB$.

bandwidth compared with the reference on the large number of GPUs.

B. Speedup and accuracy of heFFTe using compression

We compare the performance of heFFTe where the data is compressed during the communication with the original code. We perform a strong scaling experiment from two nodes, six GPUs per node, to 256 nodes, and a problem size of 1024^3. We consider as reference `FP64` and `FP32`, both doing the computation and the communication using their unique working-precision. For the compression, we truncate the data either from `FP64` to `FP32` or from `FP64` to `FP16`, respectively. Figure 4 shows the evolution of the performance (Gflops/s) when the number of GPUs increases. The solid lines correspond to the references while the dashed lines represent the performance with compression.

As expected, since `FP32` involves twice fewer bits, the volume of communication is divided by two, resulting in a performance around $2\times$ better. The `FP64` $\rightarrow$ `FP32` curve shows a greater speedup than the `FP32`, with the same volume of communication. This indicates that our implementation does not suffer from the overhead of compressing the data. Moreover, the use of the One-Sided Communication improves the overall performance, reaching up to $2.5\times$ speedup compared to `FP64`.

With a compression rate of four (`FP64` $\rightarrow$ `FP16`), heFFTe is able to reach 14 Tflops/s on 1 536 GPUs. When looking at the speedup with respect to the blue curve, we note that we exceed a $4\times$ speedup up to 384 GPUs. Then, when the number of GPUs continue to increase, the volume of communication, which is divided by 4, becomes too small and the latency starts becoming dominant.

Accuracy remains paramount for this application, thus the impact on the accuracy of doing the communication in lower precision and the computation in higher precision must be well understood. Table II shows the comparison between the reference and the casting operation from `FP64` to `FP32`. We observe that the mixed-precision gives one order of magnitude better accuracy compared with a unique working-precision of `FP32`. Furthermore, our approach allows us to consider lower precision without having the computational kernel usually needed with a unique working-precision.

#GPU	`FP64`	`FP32`	`FP64` $\rightarrow$ `FP32`
12	6.00e-15	4.96e-06	1.94e-07
24	6.17e-15	4.91e-06	2.20e-07
48	5.92e-15	4.49e-06	3.01e-07
96	6.00e-15	3.47e-06	3.90e-07
192	5.11e-15	3.54e-06	3.99e-07
384	5.25e-15	4.44e-06	5.09e-07
768	5.29e-15	3.13e-06	5.44e-07
1536	5.38e-15	3.06e-06	5.57e-07

TABLE II
COMPARISON OF THE FFT ACCURACY WHEN USING CASTING OPERATION FROM `FP64` TO `FP32` IN THE COMMUNICATION WITH THE TWO REFERENCES. EACH REFERENCE CORRESPONDS TO AN EXECUTION USING A UNIQUE PRECISION WHICH IS EITHER `FP64` OR `FP32`.

VII. CONCLUSION

The Fast Fourier Transform, a critical algorithm for many scientific applications, makes heavy use of the MPI_Alltoallv routine, up to the point where most time is spent in communication. Surprisingly, the efforts from the FFT community focusing on improving the computational aspects were not met with a similar effort to improve upon the communication aspects. We addressed this issue by taking advantage of the FFT capability to deal with accuracy loss. This allows us to compress the data pertaining to the reshape operation, using several lossy methods.

By redesigning the MPI_Alltoallv routine using the One-Sided Communication and integrating a compression technique, we showed that the performance of heFFTe increases with the compression rate, even exceeding the $4\times$ speedup expected for a compression rate of four. In addition, we demonstrated that the use of a lower precision in the communication and a higher precision in the computation improves the accuracy of the FFT by one order of magnitude, compared with the execution of heFFTe in the lower precision.

This hints that compression techniques more accurate than truncation, such as those provided by ZFP, which take advantage of the spatial distribution of the data, could simultaneously give us better compression rate or possibly a better accuracy.

This work has shown the potential of our approach and leads to future works. First, the approximate FFT implemented in heFFTe should be integrated into existing applications and the choice of the compression technique investigated thoroughly. Second, we mainly presented results using lossy compression like truncation operations. This work can be easily extended to lossless compression so that we fallback to the classical 3D FFT with a potential speedup. Third, the use of One-Sided Communication gives our all-to-all freedom and flexibility. However, implementation requires careful use of the network to avoid congestion for instance. Therefore, further investigations are needed to improve the performance.

ACKNOWLEDGEMENT

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S.

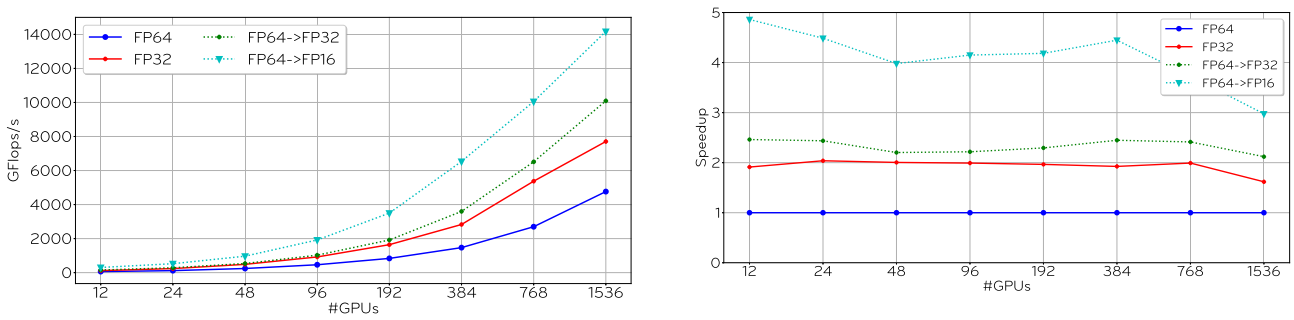

Fig. 4. Strong scaling of heFFTe for a problem of size 1024^3. The solid lines correspond to the use of the classical MPI_Alltoallv, while the dotted ones use our OSC_Alltoallv with compression. The `FP64` and `FP32` curves represent execution in double and single-precision, respectively. The `FP64` $\rightarrow$ `FP32` and the `FP64` $\rightarrow$ `FP16` curves correspond to the use of truncation operations with a compression rate of two and four, respectively. The left figure presents the performance while the figure on the right the speedup compared with the `FP64` version.

Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering and early tested platforms, in support of the nation's exascale computing imperative.

References

[1] A. Ayala, S. Tomov, A. Haidar, and J. Dongarra, "heFFTe: Highly Efficient FFT for Exascale," in *ICCS 2020. Lecture Notes in Computer Science*, 2020.

[2] A. Ayala, S. Tomov, P. Luszczek, G. Ragghianti, S. Cayrols, and J. Dongarra, "Interim report on benchmarking FFT libraries on high performance systems," University of Tennessee, ICL Tech Report ICL-UT-21-03, 2021-07 2021.

[3] A. Haidar, S. Tomov, J. Dongarra, and N. J. Higham, "Harnessing GPU Tensor Cores for Fast FP16 Arithmetic to Speed up Mixed-Precision Iterative Refinement Solvers," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '18. IEEE Press, 2018.

[4] A. Ayala, S. Tomov, P. Luszczek, S. Cayrols, G. Ragghianti, and J. Dongarra, "FFT Benchmark Performance Experiments on Systems Targeting Exascale," Tech. Rep. ICL-UT-22-02, 2022-03 2022.

[5] P. Lindstrom, "Fixed-Rate Compressed Floating-Point Arrays," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, 08 2014.

[6] S. Di and F. Cappello, "Fast Error-Bounded Lossy HPC Data Compression with SZ," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016, pp. 730–739.

[7] A. Abdelfattah, H. Anzt, E. G. Boman, E. Carson, T. Cojean, J. Dongarra, M. Gates, T. Grützmacher, N. J. Higham, S. Li, N. Lindquist, Y. Liu, J. Loe, P. Luszczek, P. Nayak, S. Pranesh, S. Rajamanickam, T. Ribizel, B. Smith, K. Swirydowicz, S. Thomas, S. Tomov, Y. M. Tsai, I. Yamazaki, and U. M. Yang, "A Survey of Numerical Methods Utilizing Mixed Precision Arithmetic," 2020.

[8] A. Sorna, X. Cheng, E. D'Azevedo, K. Won, and S. Tomov, "Optimizing the Fast Fourier Transform Using Mixed Precision on Tensor Core Hardware," in *2018 IEEE 25th International Conference on High Performance Computing Workshops (HiPCW)*, 2018, pp. 3–7.

[9] K. Kandalla, H. Subramoni, K. Tomko, D. Pekurovsky, S. Sur, and D. K. Panda, "High-performance and scalable non-blocking all-to-all with collective offload on InfiniBand clusters: a study with parallel 3D FFT." Comput Sci Res Dev 26, 237 (2011).

[10] K. S. Khorassani, C.-H. Chu, Q. G. Anthony, H. Subramoni, and D. K. Panda, "Adaptive and Hierarchical Large Message All-to-all Communication Algorithms for Large-scale Dense GPU Systems," in *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2021, pp. 113–122.

[11] M. Bayatpour, N. Sarkauskas, H. Subramoni, J. Maqbool Hashmi, and D. K. Panda, "BluesMPI: Efficient MPI Non-blocking Alltoall Offloading Designs on Modern BlueField Smart NICs," in *High Performance Computing*, B. L. Chamberlain, A.-L. Varbanescu, H. Ltaief, and P. Luszczek, Eds. Cham: Springer International Publishing, 2021, pp. 18–37.

[12] S. Li, Y. Zhang, and T. Hoefler, "Cache-Oblivious MPI All-to-All Communications Based on Morton Order," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 3, pp. 542–555, 2018.

[13] S. Chakraborty, H. Subramoni, and D. K. Panda, "Contention-Aware Kernel-Assisted MPI Collectives for Multi-/Many-Core Systems," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, 2017, pp. 13–24.

[14] G. Ballard, A. Klinvex, and T. G. Kolda, "TuckerMPI: A Parallel C++/MPI Software Package for Large-Scale Data Compression via the Tucker Tensor Decomposition," *ACM Trans. Math. Softw.*, vol. 46, no. 2, Jun. 2020.

[15] Q. Zhou, C. Chu, N. S. Kumar, P. Kousha, S. M. Ghazimirsaeed, H. Subramoni, and D. K. Panda, "Designing High-Performance MPI Libraries with On-the-fly Compression for Modern GPU Clusters," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021, pp. 444–453.

[16] Y. Hu and M. Koibuchi, "Accelerating MPI Communication Using Floating-point Compression on Lossy Interconnection Networks," in *2021 IEEE 46th Conference on Local Computer Networks (LCN)*, 2021, pp. 355–358.

[17] R. Filgueira, D. E. Singh, A. Calderón, and J. Carretero, "CoMPI: Enhancing MPI Based Applications Performance and Scalability Using Run-Time Compression," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, M. Ropo, J. Westerholm, and J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 207–218.

[18] C. Renggli, S. Ashkboos, M. Aghagolzadeh, D. Alistarh, and T. Hoefler, "SparCML: High-Performance Sparse Communication for Machine Learning," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019.

[19] H. Shan, S. Williams, and C. W. Johnson, "Improving MPI Reduction Performance for Manycore Architectures with OpenMP and Data Compression," in *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2018, pp. 1–11.

[20] A. Gholami, J. Hill, D. Malhotra, and G. Biros, "Accfft: A library for distributed-memory FFT on CPU and GPU architectures," *CoRR*, vol. abs/1506.07933, 2015.

[21] D. Takahashi, "FFTE: A fast Fourier transform package," *http://www.ffte.jp/*, 2005.

[22] S. Plimpton, A. Kohlmeyer, P. Coffman, and P. Blood, "fftMPI, a library for performing 2d and 3d FFTs in parallel," Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2018.

[23] N. Li and S. Laizet, "2DECOMP&FFT - A Highly Scalable 2D Decomposition Library and FFT Interface," 2010.

[24] J. H. Göbbert, H. Iliev, C. Ansorge, and H. Pitsch, "Overlapping of Communication and Computation in nb3dfft for 3D Fast Fourier Transformations," in *High-Performance Scientific Computing*, E. Di Napoli, M.-A. Hermanns, H. Iliev, A. Lintermann, and A. Peyser, Eds. Cham: Springer International Publishing, 2017, pp. 151–159.
[25] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005, special issue on "Program Generation, Optimization, and Platform Adaptation".
[26] D. Richards, O. Aziz, J. Cook, H. Finkel *et al.*, "Quantitative Performance Assessment of Proxy Apps and Parents," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2018.
[27] L. Dalcin, M. Mortensen, and D. E. Keyes, "Fast parallel multidimensional FFT using advanced MPI," *Journal of Parallel and Distributed Computing*, vol. 128, pp. 137–150, 2019.
[28] D. I. Gottlieb and S. A. Orszag, "Numerical analysis of spectral methods : theory and applications." Society for Industrial and Applied Mathematics, 1977.
[29] A. Gholami, D. Malhotra, H. Sundar, and G. Biros, "FFT, FMM, or Multigrid? A comparative Study of State-Of-the-Art Poisson Solvers for Uniform and Nonuniform Grids in the Unit Cube," *SIAM J. Sci. Comput.*, vol. 38, 2016.
[30] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2002.
[31] T. Grätsch and K.-J. Bathe, "A posteriori error estimation techniques in practical finite element analysis," *Computers and Structures*, vol. 83, no. 4-5, pp. 235 – 265, Jan. 2005. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01390203
[32] W. M. Gentleman and G. Sande, "Fast Fourier Transforms: For Fun and Profit," in *Proceedings of the November 7-10, 1966, Fall Joint Computer Conference*, ser. AFIPS '66 (Fall). New York, NY, USA: Association for Computing Machinery, 1966, p. 563–578.
[33] X. Liang, S. Di, D. Tao, S. Li, S. Li, H. Guo, Z. Chen, and F. Cappello, "Error-Controlled Lossy Compression Optimized for High Compression Ratios of Scientific Datasets," in *2018 IEEE International Conference on Big Data (Big Data)*, 2018, pp. 438–447.
[34] D. Tao, S. Di, Z. Chen, and F. Cappello, "Significantly Improving Lossy Compression for Scientific Data Sets Based on Multidimensional Prediction and Error-Controlled Quantization," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017, pp. 1129–1139.
[35] "NVCOMP," https://developer.nvidia.com/nvcomp, [Accessed: 16-Oct-2021].