The Journal of Supercomputing manuscript No.
(will be inserted by the editor)

Multi-GPU Accelerated 3D Seismic Modeling Applications using NCCL and CUDAWARE

Murilo Boratto · Reynam Pestana ·

Received: date / Accepted: date

Abstract In this paper, we propose to organize the concurrent computational resources of a given multi-GPU environment in a parallel structure in order perform an algorithm for solving the finite-difference based 3D Laplacian operator, the sources and receivers on seismic modeling as fast as possible. The parallel method has been implemented using the application programming interface NCCL and CUDAWARE-MPI, which are based on collective operations on GPUs architectures, and compare it with other parallel applications. We tackle, not only the performance challenge but also programmability of our idea using parallel programming frameworks.

Keywords Seismic modeling · Performance evaluation · Multi-GPU · NCCL · CUDAWARE

1 Introduction

Nowardays, many supercomputers contain one or more GPUs. At first the idea of scaling an application from one to many GPUs should result in an increase of performance but, in practice, this benefit can be difficult to obtain. There are two common reasons behind poor multi-GPU scaling. The first one is that there is not enough parallelism to efficiently saturate the processors. The second reason is that processors exchange too much data and spend more time communicating than computing. To avoid such communication bottlenecks, it is important to make the most of the available inter-GPU bandwidth.

NVIDIA has created a friendly solutions to face this communication bottleneck issue by providing high bandwidth connection libraries. These libraries provide optimized collective communication to achieve high bandwidth over high-speed bus interconnection, that implements multi-GPU and multi-node collective communication primitives. Thus, these libraries of multi-GPU are topology-aware and can be easily integrated into an parallel application. Initially developed as open-source research projects, NCCL and CUDAWARE are designed to be light-weight, depending only on the usual C/C++ and CUDA libraries. A domain decomposition scheme is used to distribute the workload so that tasks communicate through collective operations through calls.

To address this issue, NVIDIA has developed a high-speed bus called NVLINK for GPU-TO-GPU acceleration. Such technology addresses this interconnection issue by providing higher bandwidth, more links, and improved scalability for multi-GPU systems. The connections take advantage of these technologies to give greater scalability for HPC applications. As the PCIe bandwidth is increasingly becoming the bottleneck of the multi-GPU system level, the need for a faster and more scalable multiprocessor. Environments with multiple GPUs are becoming common in a variety of industries as developers rely on more parallelism in HPC applications. These include 4 and 8 GPUs system configurations using NVLINK to solve very large, complex problems.

Murilo Boratto
Núcleo de Arquitetura de Computadores e Sistemas Operacionais, Universidade do Estado da Bahia, Salvador, Bahia, Brazil, E-mail: muriloboratto@uneb.br
Reynam Pestana
Depto. de Física da Terra e do Meio Ambiente, Universidade Federal da Bahia, Salvador, Bahia, Brazil, E-mail: reynam@ufba.br

Using these primitives we describe the implementation of an algorithm for solving the finite-difference based 3D Laplacian operator, together with the sources and receivers on seismic modeling. This modeling is an integral part of processing, as it provides us with the seismic response for a given model. In the recent past a great deal of attention has been focused on the use of the wave equation for modeling and imaging. Seismic wave modeling algorithms used for calculating the response for a given subsurface depth model require large computational resources in terms of speed and memory.

The seismic modeling application which has been implemented in this work is a good choice to show the benefits of a collective operation library on multi-GPU environment using NVLINK system. This particular application exhibits an algorithmic pattern which does demand collective interactions between the tasks when domain decomposition approach is applied. The interactions between tasks are local and can be implemented using several specific exchanges among pairs of tasks. The use of collective communication operations is effective to implement these algorithms on a multi-GPU platform and that choice is more efficient when the number of GPUs grows. One justification to use collective operations is the enhancement of the programmability by using a programming style based on collective operations without incurring in a high overhead. Another advantage of NVLINK interconnect is to implement the data communications among the GPUs, instead of using specific data movements.

Overall, this paper aims to try to benchmark the new communication primitives from NCCL and CUDAWARE. In this case, they are compared to the communication over MPI but the same underlying hardware. The propose is trying to pitch NVLINK versus PCIe, and CPU versus GPU, considering performance and scalability. This paper proceeds as follows: Section 2 brings information on some related work. Section 3 presents our mathematical model to 3D seismic modeling application. Section 5 details the implementations of our model on multi-GPU systems. Section 6 presents our case studies and experimental results. Some conclusions and future work ideas close the paper.

2 Related work

Scaling a seismic application out to multiple GPUs can be a difficult task due to the complexities of communications. The communication costs can be the performance bottleneck is a well-known problem that appears in multi-GPU environments. The literature has a vast set of solutions and tools providing to new approaches designed to deal with parallel applications. Despite this apparent diversity, few proposals exploring collective operations involving multi-GPU architectures on seismic application exist so far.

Message Passing Interface (MPI) is a message-passing aplication programmer interface, together with protocol and semantic specifications for how its features must behave in any implementation. Its goal are high performance, scalability and portability. Today, it remains the dominant model used in high-performance computing to manage multiple computing nodes, and both point-to-point and collective communication are supported. Although MPI has been proved to be very useful to manage the data communication and parallel task execution, it faces a critical bottleneck which may be due to external conditions but still a fundamental one: the speed of data communication throught Ethernet. Design and the evaluation of parallel seismic modeling algorithms using MPI are discussed in [8]. This approach implements a hybrid algorithm with MPI. The parallel version was tested with a problem sizes 64 million grid points, and presented a linear speedup varying from 10 to 15. These methods are better explained in [9] and [10].

Nowadays, most of the current systems contain a multicore processor. A multicore processor has 20 or more cores which are used to process multiple tasks in parallel. The Open Multi-Processing (OpenMP) library is an application programming interface that allows CPU cores to launch multiple threads and supports shared memory model. In [4] the authors present several strategies to optimize the modeling of the acoustic wavefield with the finite-difference method in the time domain based on an efficient vectorization of the computations achieved on the computing core, and evaluate sequential and parallel methods over 200 million grid points.

Recently, Graphics Processing Units (GPUs), which possess thousands of small but efficient cores, have become an important role to accelerate the computational applications in many scientific domains and achieve a better performance than original applications. CUDA (Compute Unified Device Architecture) is a programming model proposed by NVIDIA, which can be written in C/C++, and Fortran. CUDA adheres to the single instruction, multiple threads execution model, it exploits GPU to run many threads independently and simultaneously, which even allows divergent instruction streams.

To combine multiple GPUs to handle the rapid growth of data as GPU has such a powerful parallel computation ability and as it becomes cheaper than CPU. In 2016, one computer can only be equipped at most with 2 graphic cards by some techniques such as SLI (Scalable Link Interface) of NVIDIA and CrossFire of ATI to get an enhancement of 1.4 to 1.6 speedup of performance [5]. It was nevertheless unable to cope with intensive computation request with big data. Message Passing Interface has then become a solution which may connect multiple computers with 1 or 2 graphic cards installed, enabling them to send and receive messages to/from each other through Ethernet. In [2] Hung proposed an MPI version implementation of UPGMA on multiple computers equipped with GPUs. However, the bottleneck of the MPI version implementation is the slow response over Ethernet. Today, one computer is able to accommodate 2 to 8 GPUs, and these GPUs can communicate with each other through PCIe bus. Thus, the response time is much less than Ethernet. Furthermore, a new communication technique, called NVLINK, was proposed by NVIDIA to enable ultrafast communication between CPU and GPU or among GPUs. NVIDIA claimed that this technology can accelerate data transfer rate to 5 to 12 times faster than PCIe bus [6]. It means that GPUs can access data from each other at the speed of accessing local data from themselves. But in the programming level, it is usually complicated to write a program involving the communication among multiple GPUs. Fortunately, the NCCL library provides a communication model that is very similar to MPI.

Table 1 summarizes existing approaches to seismic modeling presenting some kind of parallelization on multiprocessor, multicore, and multi-GPU environments. We compared proposals that are closest to ours in terms of the following features:

1. Kind of processing unit (CPU or GPU) performing computation;
2. Use of performance evaluation techniques;
3. Use of communication strategies to increase performance comparison;
4. Scalability underlying communication library;
5. Efficiency on message size.

Table 1: Parallel approaches to seismic modeling.

	Features				
References	(1)	(2)	(3)	(4)	(5)
Multiprocessor (MPI) [8]	✓			✓	
Multicore (OpenMP) [4]	✓			✓	
GPUs (CUDA) [1] [5]	✓	✓		✓	
Hybrid [2] [9] [10]	✓	✓		✓	
Our Solution GPUs (CUDA + NCCL + NVLINK)	✓	✓	✓	✓	✓

Our approach contributes to the field in terms of the high performance and accuracy results given choosing the right GPUs combination can impose considerable impact on communication efficiency, as well as overall performance of the application.

3 Mathematical formulation to seismic modeling

The mathematical acoustic wave equation in a 3D model is a computational mathematical approach of a phenomenon that occurs during the wave propagation. This model can represent plenty of geophysical information from a site. One available technique to accomplish this representation for a variable density case, is given as follows.

$$\frac{1}{K}\frac{\partial^2 P}{\partial t^2} = \frac{\partial}{\partial x}\left[\frac{1}{\rho}\frac{\partial P}{\partial x}\right] + \frac{\partial}{\partial y}\left[\frac{1}{\rho}\frac{\partial P}{\partial y}\right] + \frac{\partial}{\partial z}\left[\frac{1}{\rho}\frac{\partial P}{\partial z}\right], \tag{1}$$

whereas P is the pressure wavefield, ρ is the density and K is the incompressibility. A particular form for the wave equation in Equation 1 can be exemplified for whereas ρ a constant case as follows,

$$\frac{1}{K}\frac{\partial^2 P}{\partial t^2} = \frac{1}{\rho}\left(\frac{\partial^2 P}{\partial x^2} + \frac{\partial^2 P}{\partial y^2} + \frac{\partial^2 P}{\partial z^2}\right). \tag{2}$$

Now, the Eq. 2 can be rewritten as:

$$\frac{\partial^2 P}{\partial t^2} = c^2 \nabla^2 P, \tag{3}$$

where $\nabla^2 = \dfrac{\partial^2}{\partial x^2} + \dfrac{\partial^2}{\partial y^2} + \dfrac{\partial^2}{\partial z^2}$ is the 3D Laplacian operator and $c^2 = \dfrac{\rho}{K}$.

We divide the 3D model into a grid of $I \times J \times K$ points, in order to obtain finite difference approximation to previously equation, let us introduce a set of indices i, j, k, such:

$$\begin{array}{ll} x_i = i * \Delta x & \rightarrow i = 0, 1, 2, \cdots I \\ y_j = j * \Delta y & \rightarrow j = 0, 1, 2, \cdots J \\ z_k = k * \Delta z & \rightarrow k = 0, 1, 2, \cdots K \end{array}$$

where Δx, Δy and Δz are the grid spacing and i, j and k are the number of grid points in x, y and z directions respectively, Δt is the time step and n is the total number of time steps. Physcial parameter, velocity is specified at each grid point. The spatial derivatives can be approximated by finite-difference schemes and are given by:

$$\begin{cases} \dfrac{\partial^2 P^n}{\partial x^2} = \dfrac{P^n_{i-1,j,k} - 2P^n_{i,j,k} + P^n_{i+1,j,k}}{\Delta x^2}, \\ \\ \dfrac{\partial^2 P^n}{\partial y^2} = \dfrac{P^n_{i-1,j,k} - 2P^n_{i,j,k} + P^n_{i+1,j,k}}{\Delta y^2}, \\ \\ \dfrac{\partial^2 P^n}{\partial z^2} = \dfrac{P^n_{i-1,j,k} - 2P^n_{i,j,k} + P^n_{i+1,j,k}}{\Delta z^2}. \end{cases} \tag{4}$$

Substituting the central finite-difference approximations of the derivatives in Eq. 3, and also using second order finite-difference to approximate the time derivative, we obtain the following scheme for calculating the wavefield $P^{n+1}_{i,j,k}$ from the knowledge of the wavefield at previous time levels:

$$\boxed{P^{n+1}_{i,j,k} = 2P^n_{i,j,k} - P^{n-1}_{i,j,k} = c(i,j,k) * \Delta t^2 * \nabla^2 P^n_{i,j,k}} \tag{5}$$

where $\nabla^2 P^n_{i,j,k}$ is compute using Eq. 4, that is a second order partial differential and the solution was approximated by finite difference approximation as a function of time and space domain by explaining the advance of the pressure wavefield from the measurements taken at a given position.

4 NVIDIA Collective Communications Library (NCCL)

NCCL is a stand-alone library of standard collective communication routines for multi-GPU, implementing collective communication primitives such as *All-Gather*, *All-Reduce*, *Broadcast*, *Reduce*, and *Reduce-Scatter*. It has been optimized to achieve high bandwidth on platforms. NCCL supports an arbitrary number of GPUs installed in a single node or across multiple nodes, and can be used in either single or multi-process applications.

Tight synchronization between communicating processors is a key aspect of collective communication. The CUDA API based collectives would traditionally be realized through a combination of memory copy operations and kernels for local reductions. The NCCL, on the other hand, implements each collective in a single kernel handling both communication and computation operations. This allows for fast synchronization and minimizes the resources needed to reach peak bandwidth.

It conveniently removes the need for developers to optimize their applications for specific machines. It provides fast collectives over multiple GPUs both within and across nodes. It supports a variety of interconnect technologies including PCIe, NVLINK and InfiniBand. NCCL also automatically patterns its communication strategy to match the underlying interconnect topology on multi-GPU environments.

Next to performance, ease of programming was the primary consideration in the design of NCCL. It uses a simple interface, which can be easily accessed from a variety of programming languages. NCCL closely follows the popular collectives API defined by MPI. Anyone familiar with MPI will thus find API of NCCL very natural to use. In a minor departure from MPI, NCCL collectives take a stream argument which provides direct integration with the CUDA programming model [7].

4.1 Using NCCL

Using NCCL is similar to using any other library in your code and your primitives have the function to perform data communication. Collective communication primitives are common patterns of data transfer among a group of CUDA devices. A communication algorithm involves many processors that are communicating together. Each CUDA device is identified within the communication group by a zero-based index or rank. Each rank uses a communicator object to refer to the collection of GPUs that are intended to work together. The creation of a communicator is the first step needed before launching any communication operation. Like MPI collective operations, NCCL collective operations have to be called for each rank to form a complete collective operation. Follow we explain some collective communication operations implemented in NCCL.

4.1.1 All-Reduce

The *AllReduce* operation is performing reductions on data across devices and writing the result in the receive buffers of every rank. This operation is rank-agnostic. Any reordering of the ranks will not affect the outcome of the operations. The operation starts with independent arrays V_k of N values on each of k ranks and ends with identical arrays S of N values, where $S[i] = V_0[i] + V_1[i] + \cdots + V_{k-1}[i]$, for each rank.

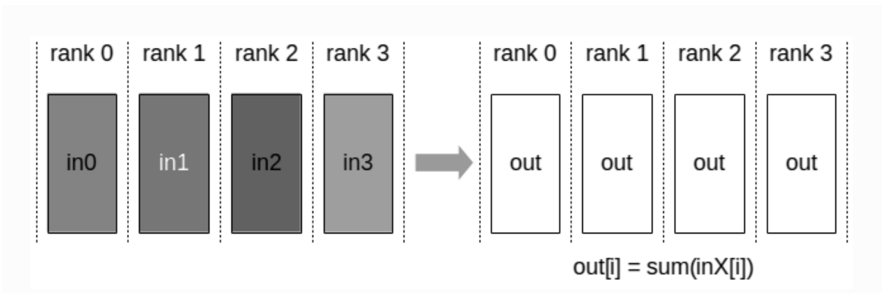


Fig. 1: The flowchart of *All-Reduce* operation: each rank receives the reduction of input values across ranks.

4.1.2 Broadcast

A *Broadcast* operation is one of the standard collective communication techniques. During a broadcast, there are copies N element buffer from the root rank to all ranks.

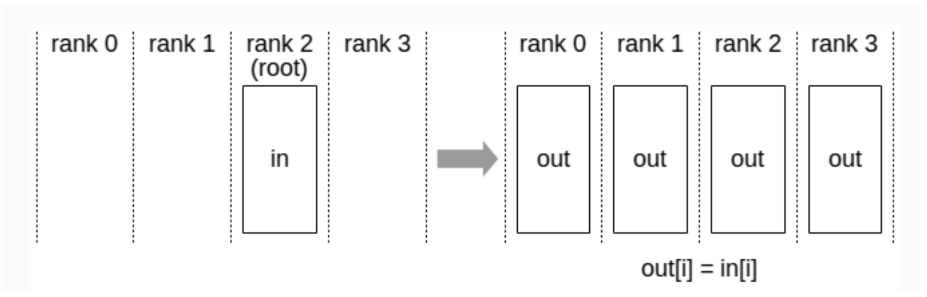


Fig. 2: The flowchart of *Broadcast* operation: all ranks receive data from a root rank.

4.1.3 Reduce

The *Reduce* operation is performing the same operation as *All-Reduce*, but writes the result only in the receive buffers of a specified root rank.

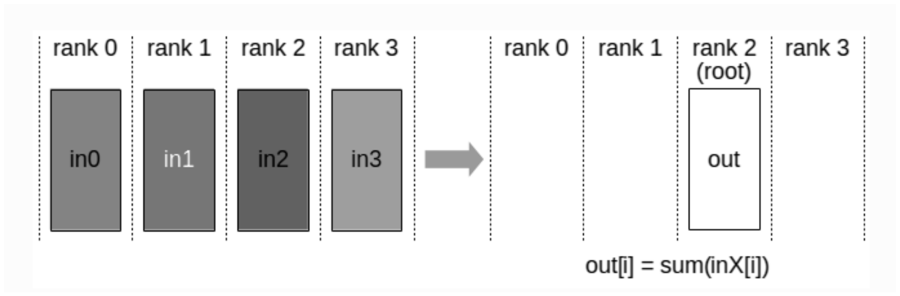


Fig. 3: The flowchart of *Reduce* operation: one rank receives the reduction of input values across ranks.

4.1.4 All-Gather

In the *All-Gather* operation, each of the K processors aggregates N values from every processor into an output of dimension $K * N$. The output is ordered by rank index.

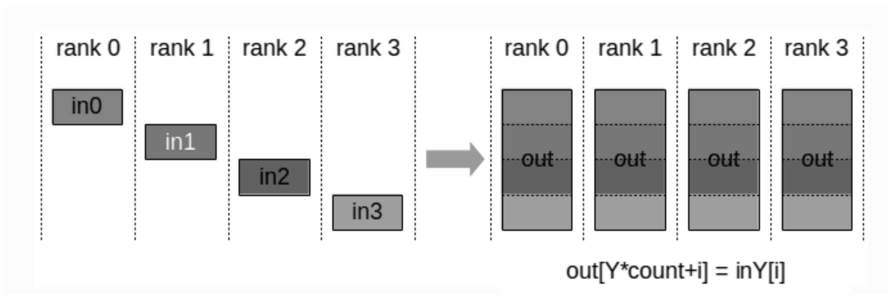


Fig. 4: The flowchart of *All-Gather* operation: each rank receives the aggregation of data from all ranks in the order of the ranks.

4.1.5 Reduce-Scatter

The *Reduce-Scatter* operation performs the same operation as the e *Reduce* operation, except the result is scattered in equal blocks among ranks, each rank getting a chunk of data based on its rank index.

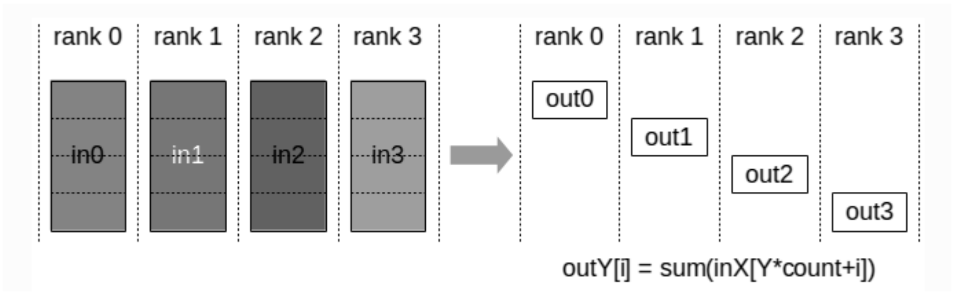


Fig. 5: The flowchart of *Reduce-Scatter* operation: input values are reduced across ranks, with each rank receiving a subpart of the result.

5 Implementation of the parallel algorithm

The most important part of GPU parallel programming is to map out a particular problem on a multi-GPU environment. The problem should be broken down into a set of tasks that can be solved concurrently. The choice of an approach to the problem decomposition depends on the computational scheme. For the finite-difference approximation to 3D acoustic wave equation scheme used here, one can observe that the calculation of the wavefields at a grid point at an advanced time and space level involves the knowledge of the wavefield at seven grid points of the current time level. Therefore, if we use a domain decomposition scheme for solving this problem, only the first order neighbors will be involved in communication for central difference scheme.

5.1 Domain decomposition and striped partitioning

The parallel implementation of the algorithm is based on domain decomposition, which involves assigning subdomains to different GPUs and solving the equations for each subdomain concurrently. The problem domain is a cuboid with grid point size $I \times J \times K$ in x, y and z directions, as shown in the Fig. 1. This domain can be partitioning in one way: partition in stripe. In the striped partitioning of the 3D domain, is divided into vertical planes, and each GPU is assigned one such plane. Striped partition can be done in one way as shown in Fig. 2. For load balancing we divide the domain in equal size of the pizza boxes, depending upon the number of available GPUs.

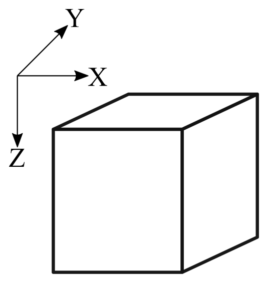


Fig. 6: Domain Decomposition.

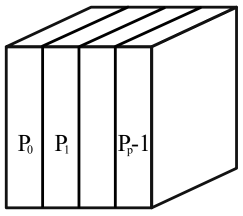


Fig. 7: Partition in y-direction.

5.2 Using multi-GPU with NCCL

Eventually, to construct a geophysic model from a larger amount of seismic synthetic data set using multi-GPU, a single GPU device is incapable to deal with them. Therefore, we proposed the parallel algorithm to build a tree by multiple GPU devices within on single machine. All of the GPU devices have to collaborate together based on this idea. Therefore, NCCL is chosen to achieve this goal. The flowchart is shown in Fig. 8 represents the steps of the proposed algorithm are listed as follows:

[Step 1**]** Inicialization: In the first step, all the available GPU devices have to been recognized by NCCL. The input sequences are thus split into a number of sequence groups corresponding to the number of recognized GPU devices.

[Step 2**]** Inter Devices Communication. In this step, we have used a velocity model for generating snapshots of 3D acoustic wave propagation on multi-GPU using collective communication routine to distribute all the local values to each other.

[Step 3**]** Updating the solving the finite-difference. When *All-Gather* is finished, each GPU has a full set of local values from itself and other GPU, and the global value will be found and the solution can be updated according to this value.

The most important part of the parallel programming is to map out a particular problem on a multi-GPU environment. The problem must be broken down into a set of tasks that can be solved

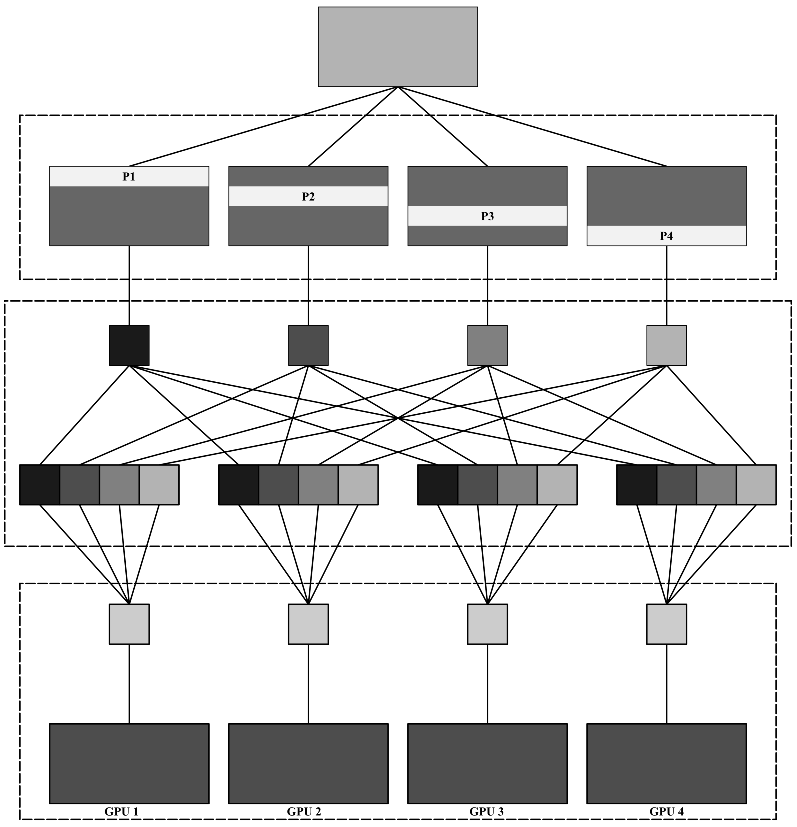


Fig. 8: The flowchart of the bottom-up approach of 3D seismic modeling on 4-GPUs using NCCL represents the sequence steps.

concurrently. The choice of an approach to the problem decomposition depends on the computational parallel scheme.

Computation performed by multi-GPU is implemented in the `kernel()` function, called in line 30 of Algorithm 1. This function firstly performs the typical operations of uploading data from the CPU to the GPU prior to call the CUDA kernel. Thus, in order to execute this kernel, it is supposed that `matrixA`, `nx`, `ny`, `nz`, `dx`, `dy` and `dz` have been previously uploaded into the GPU memory.

The problem to be solved through the kernel is highly parallelizable. All elements from `matrixA` can be computed concurrently. We need to imagine the shared data as a tree-dimensional cube where each position has a partial gather of each element of 3D Laplacian operator. In other words, elements of `matrixC`, for all `nx`, `ny`, `nz`, `dx`, `dy` and `dz` contain the partial computation corresponding to a given element, taking into account the parity between the array element and the thread coordinates. After this, it is necessary to add all partial computation. The collective communication routines used common patterns of data transfer among many GPUs. In our case, the *ncclAllGather* starts with each of the P_i on GPUs aggregates n values from every GPU into an output of dimension $P_i * n$.

However, the total amount of shared memory is what really determines the size of thread blocks. Regardless, the limitation in the number of threads per block is easily overcome by the number of blocks that can be run concurrently. More blocks means that data computed by each block in that dimension and stored in the shared memory should be also shared among the thread blocks. This can only be done through global memory yielding in a performance penalty.

Algorithm 1 The algorithm for parallel implementation of 3D Laplacian operator on multi-GPU.

```
INPUT
  matrixA           /* larger  matrix  with input data*/
  nx, ny, nz        /* parts of matrixA manipulated by GPU and CPU */
  dx, dy, dz        /* the number of grid points in x, y and z directions */
  ista, iend        /* stores the partition of matrixA assigned
                       to each processing unit */

OUTPUT
  matrixC           /* velocity model as a numerical example for generating
                       snapshots of 3D acoustic wave propagation */

kernel(double *matrixA, *c, int nx, ny, nz, ista, iend, dx, dy, dz) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  int k = blockIdx.z * blockDim.z + threadIdx.z;
  double sx, sy, sz;

if(k>=1 && k<(ny-1) && j>=1 && j<(nx-1) && i>=(ista-1) && i<iend){
  Py = matrixA[ i    +  j*ny  + (k-1)*(nx*ny)] +
       matrixA[ i    +  j*ny  + (k+1)*(nx*ny)] +
       2 * matrixA[i +  j*ny  +  k*(nx*ny)];
  Px = matrixA[ i    + (j-1)*ny  +  k *(nx*ny)] +
       matrixA[ i    + (j+1)*ny  +  k *(nx*ny)] +
       2 * matrixA[i + j*ny +  k *(nx*ny)];
  Pz = matrixA[(i-1) + j*ny +  k *(nx*ny)] +
       matrixA[(i+1) + j*ny +  k *(nx*ny)] +
       2 * matrixA[i + j*ny +  k *(nx*ny)];
  matrixC[i + j*ny + k*(nx*ny)] = (Px/(dx*dx))+(Py/(dy*dy))+(Pz/(dz*dz));
 }
}
 ...
  ncclGroupStart();
    for( myid = 0; myid < nGPUs; myid++ ) {

       domainDivided(...);

       stripedPartitioning(...);

       interDevicesCommunication(...);

       kernel<<<dimGrid(grid, grid, grid),
                dimBlock(sizeblock, sizeblock, sizeblock)>>>(...);

       ncclAllGather(...);
    }
  ncclGroupEnd();
 ...
```

6 Experimental results

This section presents some results obtained with the usage of our proposed scheme. Experimental results are shown and commented, with detailed explanation and useful insights.

6.1 Characterization of the execution environment

Our execution board comprises 4 Intel(R) Xeon(R) E5-2698 processors at 3.33 GHz and 100 GB DDR3 main memory. Each processor has 20 cores with 36 MB of cache memory. The board also contains single node with 4 NVIDIA Tesla P100-SXM2 GPUs, NVLINK-V1 interconnect, PASCAL arch, featuring each GPU with 10609 SP/5304 DP GFlops, 16 GB HBM2 at 732 GB/s. Floating point operations follow the IEEE 754-2008 standard. The installed CUDA toolkit is version 10.1.

6.2 3D Seismic modeling representation analysis

In order to validate the presented methodology and derived equations, it will be applied computing techiques for efficient solution of 3D seismic model to represent the wave propagation through the velocity model. Using all the points representing in the grid points, by Eq. 5. The computational time to estimate such a seismic model needs great computer power and a long time of processing. Finite-difference computation of the snapshots can help in our understanding of wave propagation in the medium. We have used a constant velocity model for generating snapshots of 3D acoustic wave propagation. Source is placed at the center of the cubic model. For simplicity sake there is no density variation within the model. Fig. 4 shows a common shot gather recorded along the surface of model.

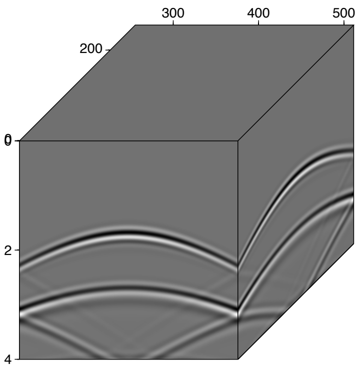


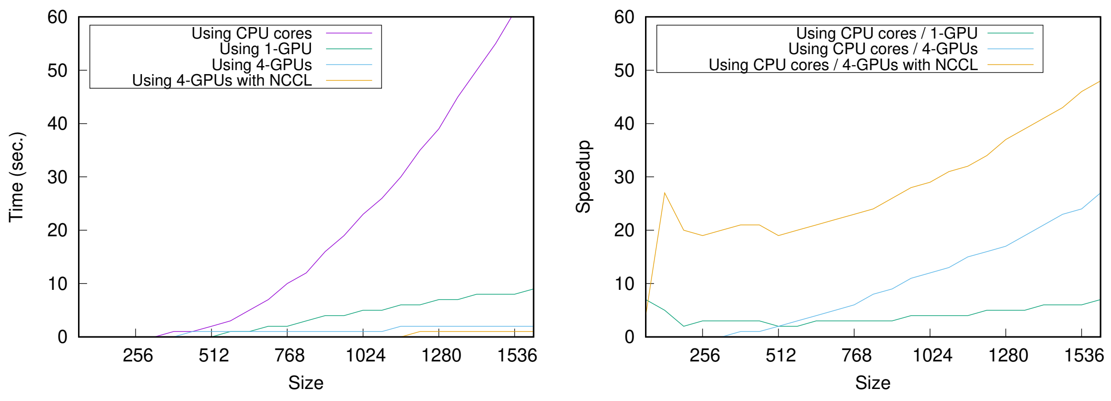
Fig. 9: Common shot gather of 3D acoustic wave propagation through a constant velocity model.

6.3 Performance analysis

In our experiments we use a parallel version of Algorithm 1 implemented using NCCL and CUDA for the 3D Laplacian operator for seismic modeling used in our multi-GPU environment. Many parameter values were used to estimate the best values for system. The available range for CPU cores (c) is $1, 2, \ldots, 32$, with Intel Hyper-Threading [3] set. Then, we checked for GPUs workloads (w) using 100% of the problem sizes to 1-GPU and 25% part of the problem sizes to 4-GPUs. The input sizes of the problem sizes, the larger matrix with input data and the number of grid points in x, y and z directions for the experiments. We show in Fig. 10a and Fig. 10b, the execution time and the speedup, respectively, for different sizes ranging from $256 \times 256 \times 256$ to $1,600 \times 1,600 \times 1,600$. The execution was carried out on each subsystem independently to have a measure for comparison purposes.

The execution time with multiple threads is denoted by “CPU cores”. Versions denoted by “1-GPU” and “4-GPUs” represent executions in one and four devices, respectively, through PCIe bus. The multi-GPU model (“4-GPUs-with-NCCL”) uses all cores available in the multi-GPU system on the high-speed NVLINK bus. The results show that the parallel CPU algorithm reduces the execution time significantly. We show in the figures and tables, the execution time and the speedup for the seismic wave modeling algorithms with different sizes ranging. The execution was carried out on each subsystem independently to have a measure for comparison purposes.

The results show that the parallel GPU algorithm using NVLINK bus reduces the execution time significantly, as can be observed in Fig. 10a, Fig. 10b, Tab. 2, and Tab. 3. Speedup has been obtained with regard to the use of the CPU cores subsystem. The maximum speedup is around 48 with the multi-GPU model, presenting a difference in performance that can be observed more clearly. Both plots show how the use of GPUs in our system clearly outperforms the computation on the CPU cores. We show different aspects of the behavior of the algorithm with different workloads. This experiment shows the reduction in time achieved by the use of 4-GPUs-with-NCCL, and how this improvement grows with the problem size due to the parallelization of the seismic wave modelling algorithms.



(a) Execution time (b) Speedup rate

Fig. 10: Comparison of execution time and speedup rate to striped partitioning for 3D acoustic wave modeling for available range.

Table 2: Comparative performance analysis of execution time (Best values marked in boldface).

Size	CPU cores	1-GPU	4-GPUs	4-GPUs-with-NCCL
$64 \times 64 \times 64$	0.0132	0.0018	0.9111	**0.0029**
$128 \times 128 \times 128$	0.0788	0.0134	0.9112	**0.0029**
$192 \times 192 \times 192$	0.1400	0.0492	0.9121	**0.0070**
$256 \times 256 \times 256$	0.3357	0.0899	0.9328	**0.0169**
$320 \times 320 \times 320$	0.6492	0.1730	0.9452	**0.0314**
$384 \times 384 \times 384$	1.1235	0.2977	0.9752	**0.0524**
$448 \times 448 \times 448$	1.7924	0.4897	1.1021	**0.0824**
$512 \times 512 \times 512$	2.6957	0.9271	1.1363	**0.1412**
$576 \times 576 \times 576$	3.9365	1.4023	1.2415	**0.1921**
$640 \times 640 \times 640$	5.5657	1.8188	1.2908	**0.2637**
$704 \times 704 \times 704$	7.6237	2.4232	1.3860	**0.3454**
$768 \times 768 \times 768$	10.0263	2.9505	1.4714	**0.4225**
$832 \times 832 \times 832$	12.8061	3.5038	1.5512	**0.5229**
$896 \times 896 \times 896$	16.0880	4.0949	1.6485	**0.6140**
$960 \times 960 \times 960$	19.8197	4.7310	1.7513	**0.7012**
$1,024 \times 1,024 \times 1,024$	23.0123	5.2272	1.8257	**0.7783**
$1,088 \times 1,088 \times 1,088$	26.9765	5.8059	1.9519	**0.8555**
$1,152 \times 1,152 \times 1,152$	30.9876	6.3916	2.0400	**0.9427**
$1,216 \times 1,216 \times 1,216$	35.0981	6.8239	2.1386	**1.0099**
$1,280 \times 1,280 \times 1,280$	39.9023	7.3056	2.2386	**1.0771**
$1,344 \times 1,344 \times 1,344$	45.0935	7.7561	2.2879	**1.1443**
$1,408 \times 1,408 \times 1,408$	50.0001	8.1883	2.3513	**1.2115**
$1,472 \times 1,472 \times 1,472$	55.9009	8.5206	2.4247	**1.2787**
$1,536 \times 1,536 \times 1,536$	61.9128	8.8528	2.4814	**1.3459**
$1,600 \times 1,600 \times 1,600$	68.9088	9.0850	2.5315	**1.4131**

7 Conclusions and Future Directions

In this work we have summarized the parallel implementation of a finite difference based 3D Laplacian operator. This algorithm is being used as forward modeling tool in seismic data processing for oil and gas exploration. The code for this implementation has been written using CUDA with NCCL library. Performance analysis shows that for the domain decomposition on Multi-GPU environment with NCCL on NVLINK bus gives the best performance. Experiments demonstrate that the proposed algorithm on multiple GPU devices is able to significantly improve the computational perfomance of GPU on a single GPU device when dealing with a large enough amount of data. And, in contrast to the CPU cores, 1-GPU and 4-GPUs solutions, the communication bottleneck is attenuated by the technique of NCCL and NVLINK, hence, the proposed algorithm is also much faster than all versions. We have also found that for the large size of the problems that can not fit into the memory of serial computers, parallel computing is the only solution.

Table 3: Comparative performance analysis of speedup rate (Best values marked in boldface).

Size	1-GPU	4-GPUs	4-GPUs-with-NCCL
$64 \times 64 \times 64$	7.3333	0.0145	**4.5517**
$128 \times 128 \times 128$	5.8806	0.0865	**27.1724**
$192 \times 192 \times 192$	2.8455	0.1535	**20.0000**
$256 \times 256 \times 256$	3.7341	0.3599	**19.8639**
$320 \times 320 \times 320$	3.7526	0.6868	**20.6752**
$384 \times 384 \times 384$	3.7739	1.1521	**21.4408**
$448 \times 448 \times 448$	3.6602	1.6263	**21.7524**
$512 \times 512 \times 512$	2.9077	2.3723	**19.0914**
$576 \times 576 \times 576$	2.8072	3.1708	**20.4919**
$640 \times 640 \times 640$	3.0601	4.3118	**21.1062**
$704 \times 704 \times 704$	3.1461	5.5005	**22.0721**
$768 \times 768 \times 768$	3.3982	6.8141	**23.7309**
$832 \times 832 \times 832$	3.6549	8.2556	**24.4905**
$896 \times 896 \times 896$	3.9288	9.7592	**26.2002**
$960 \times 960 \times 960$	4.1893	11.3171	**28.2670**
$1,024 \times 1,024 \times 1,024$	4.4024	12.6046	**29.5674**
$1,088 \times 1,088 \times 1,088$	4.6464	13.8205	**31.5330**
$1,152 \times 1,152 \times 1,152$	4.8482	15.1900	**32.8711**
$1,216 \times 1,216 \times 1,216$	5.1434	16.4117	**34.7540**
$1,280 \times 1,280 \times 1,280$	5.4619	17.8247	**37.0460**
$1,344 \times 1,344 \times 1,344$	5.8139	19.7093	**39.4071**
$1,408 \times 1,408 \times 1,408$	6.1063	21.2646	**41.2712**
$1,472 \times 1,472 \times 1,472$	6.5607	23.0544	**43.7170**
$1,536 \times 1,536 \times 1,536$	6.9936	24.9509	**46.0010**
$1,600 \times 1,600 \times 1,600$	7.5849	27.2201	**48.7643**

Acknowledgments

This work has been partially supported by CNPq, FAPESB, NVIDIA Hardware Grant Program, Generalitat Valenciana (grant PROMETEOII/2014/003), Spanish Government and European Commission through TEC2015-67387-C4-1-R, and network CAPAP-H. We have also worked in cooperation with the SENAI-CIMATEC Supercomputing Center.

References

1. Amado, J., Salamanca, W., Vivas, F., Ramirez, A.: A GPU implementation of the reverse time migration algorithm. pp. 1016–1021 (2015). DOI 10.1190/sbgf2015-202
2. Che-Lun, H., et al.: MGUPGMA: A fast UPGMA algorithm with multiple graphics processing units using NCCL. pp. 1016–1021 (2017). DOI 10.1190/sbgf2015-202
3. Étienne, E.Y.: Hyper-threading. TurbsPublishing (2012)
4. Etienne, V., et al.: Optimization of the seismic modeling with the time-domain finite-difference method. pp. 3536–3540 (2014). DOI 10.1190/segam2014-0176.1
5. Kim, Y., Jo, J.E., Jang, H., Rhu, M., Kim, H., Kim, J.: GPUpd: A fast and scalable multi-GPU architecture using cooperative projection and distribution. In: Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-50 '17, pp. 574–586. ACM, New York, NY, USA (2017). DOI 10.1145/3123939.3123968. URL `http://doi.acm.org/10.1145/3123939.3123968`
6. Li, A., et al.: Evaluating modern GPU interconnect: PCIe, NVLINK, NV-SLI, NVSwitch and GPUDirect (2019)
7. NVIDIA Corporation: NVIDIA Collective Communication Library (NCCL) Documentation (2019). Accessed: 2019-01-20
8. Phadke, S., Bhardwaj, D., Yerneni, S.: 3D Seismic modeling in a message passing environment (2000)
9. Tian, K., et al.: 3D forward modeling of viscoacoustic and viscoelastic media based on MPI-OpenMP combined parallelization. pp. 427–430 (2017). DOI 10.1190/IGC2017-110
10. Xiangbin, M., et al.: A universal seismic processing software development frame on heterogeneous parallel multi-core architecture. pp. 242–245 (2014). DOI 10.1190/IGCBeijing2014-062