https://doi.org/10.1016/j.ultras.2023.107049

A multi-GPU and CUDA-aware MPI-based spectral element formulation for ultrasonic wave propagation in solid media

Feilong Li[a], Fangxin Zou[a,1], Jing Rao[b]

[a] Department of Aeronautical and Aviation Engineering, The Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong SAR, China

[b] School of Engineering and Information Technology, The University of New South Wales, Canberra, ACT 2600, Australia

Abstract

In this paper, we introduce a new multi-GPU-based spectral element (SE) formulation for simulating ultrasonic wave propagation in solids. To maximize communication efficiency, we purposely developed, based on CUDA-aware MPI, two novel message exchange strategies which allow the common nodal forces of different subdomains to be shared between different GPUs in a direct manner, as opposed to via CPU hosts, during central difference-based time integration steps. The new multi-GPU and CUDA-aware MPI-based formulation is benchmarked against a multi-CPU core and classical MPI-based counterpart, demonstrating a remarkable acceleration in each and every stage of the computation of ultrasonic wave propagation, namely matrix assembly, time integration and message exchange. More importantly, both the computational efficiency and the degree-of-freedom limit of the new formulation are actually scalable with the number of GPUs used, potentially allowing larger structures to be computed and higher computational speeds to be

[1] Corresponding author. Email: frank.zou@polyu.edu.hk.

realized. Finally, the new formulation was used to simulate the interaction between Lamb waves and randomly shaped thickness loss defects on plates, showing its potential to become an efficient, accurate and robust technique for addressing the propagation of ultrasonic waves in realistic engineering structures.

Keywords: ultrasonic wave; spectral element formulation; GPU; multi-GPU; CUDA; CUDA-aware MPI

1. Introduction

Owing to its versatility, the finite element method (FEM) has been extensively applied to solving ultrasonic wave propagation problems [1-3]. Like many other numerical methods, the FEM also faces the crucial limitation that the dispersion error and dissipation error of a model would increase with wave frequency and simulation time [4], leading to waveform distortions. Consequently, if the standard low-order FEM is used to address high-frequency ultrasonic wave propagation, very fine meshes would need to be employed and hence much computational resource would be required [5].

While the standard low-order FEM is indispensable to discretizing complex geometries, the spectral element method (SEM), which is essentially a high-order FEM, can achieve a higher accuracy, stability and mesh convergence rate [6-8], when dealing with rather uniform geometries. By using high-order polynomials (e.g., Lobatto polynomials, Chebyshev polynomials and Laguerre polynomials) to approximate the deformations of structures [9], the SEM would require fewer elements and, overall, less computational resource in modelling ultrasonic wave propagation. The SEM was first introduced by Patera in fluid mechanics [6]. It has since been applied to a variety of scenarios, including but not limited to linear [10] and nonlinear [11] elastic wave propagation in solids, the local wave propagation in a substructure of a concrete-filled steel tube member [12], compressible flows in 2D mixed grids [13], and parabolic initial boundary value problems [14].

The efficiency of a finite element (FE) or spectral element (SE) implementation would depend on both the hardware facility and the algorithm design. Nowadays, hardware facilities that embody multi-core CPUs are widely available. Multi-CPU core-based FE models [15-18] and SE models [19, 20] have long been utilized to solve various scientific and engineering problems. In CPU-based parallel implementations, the communications between different CPU cores are achieved through the classical MPI [21], which is a large library that contains numerous functions for data transfer.

Compared with CPUs, graphics cards possess a much greater capability in parallel computing. Over the past several years, graphics cards have been extensively associated with parallel FE and SE implementations [22-28]. A graphics card is essentially made up of two parts, i.e., a GPU and a series of cache memories. A GPU comprises of a large number of streaming multiprocessors (SMs), each of which further consists of numerous streaming processors (SPs). As such, a GPU is capable of executing an application, also known as a kernel, in a highly parallelized manner using an enormously large number of threads. The CUDA platform [29, 30] offers a convenient interface for programming kernels that are executed by NVIDIA GPUs and make use of the various accompanying cache memories. Under the CUDA platform, when a kernel is launched, a group of threads, collectively referred to as a grid, will be instructed to execute it. The different threads in the grid can share data through the global memory space, constant memory space and texture memory space that are assigned to the grid. Since these three types of memory spaces are also visible to CPUs, they make effective data communication channels between GPUs and CPU hosts. A grid can be further divided into a number of blocks. The different threads in a block can also share data through the shared memory space that is assigned to the block. What's more, every thread is assigned with its own register memory space and local memory space. All in all, GPUs, owing to their multi-thread computational hierarchy and multi-memory storage hierarchy, are intrinsically advantageous in computing, in parallel, problems that can be partitioned into many parts.

Various FE formulations have been implemented on GPUs with a view to achieve better computational performances. Huthwaite introduced a GPU-based FE formulation for simulating ultrasonic waves, mechanical vibrations and seismic waves [31]. To minimize the communication effort in explicit time integration steps, the author proposed a novel approach for arranging the storage of nodal information in GPU memories. In another work, a GPU was used to accelerate the domain decomposition-based solution processes of the spectral stochastic FE models of intrusive stochastic mechanics problems [32]. The GPU-based implementation achieved a significant improvement in both computational speed and energy efficiency over its CPU-based counterpart. Furthermore, Cao et al. implemented an edge-based smoothed FE formulation for simulating acoustic problems on a GPU [33]. It was shown that the acceleration that can be achieved by the use of a GPU would first increase with the number of elements and then plateau once the computational resource of the GPU used has been exhausted. The aforementioned works, as well as many others, demonstrate that in the implementation of FE models on GPUs, the distribution of computation among threads and the arrangement of data in GPU memories are two crucial steps.

GPU computing has also been exploited for speeding up high-order FE models and SE models. For instance, Kudela implemented on a single GPU a 3D SE formulation for Lamb wave propagation [34]. Also on a single GPU, Remacle et al. implemented an energy-stable, low-storage FE formulation that utilizes high-order hexagonal elements [35]. However, the use of a single GPU for computation would no doubt struggle in processing large-scale structures. Therefore, researchers have begun exploring the implementation of SE models on multiple GPUs. A SE formulation for simulating earthquake-induced seismic wave propagation was implemented on a large cluster of NVIDIA TESLA GPUs by Komatitsch [36]. However, the message exchange strategy adopted requires the data to be communicated between different GPUs to go through CPU hosts – certainly not the most ideal and efficient option in multi-GPU environments.

In order to implement a SE model on a multi-GPU cluster, the domain of the structure will need to be decomposed into several subdomains, each of which will be allocated to a single GPU to process. The subdomains should have similar numbers of degrees of freedom (DOFs) so that the computational loads of different GPUs will be comparable and hence different GPUs will not need to wait for each other during processing. To ensure that a multi-GPU-based implementation outputs the same result as its single-GPU-based counterpart, different GPUs will need to communicate with each other. However, message exchange is very time-consuming and therefore the amount of data which will need to be communicated between different GPUs should be minimized. A multi-GPU-based SE implementation is potentially a highly fine-grained parallel computing scenario. Thanks to the multi-thread computational hierarchy of GPUs, the mesh of a SE model can be partitioned into very small parts, each of which will be processed by a group of threads. Also, the accompanying multi-memory storage hierarchy allows the different types of data of a mesh to be stored in different cache memories. As such, the computational effort of an individual thread will be minimized and rapid memory access will be enabled.

In this paper, we introduce a novel multi-GPU-based SE formulation for addressing high-frequency ultrasonic wave propagation in solid media. For a given problem, the structure concerned will be decomposed into *N* subdomains, where *N* is the total number of GPUs that will be used for computation. To avoid GPU memory access conflicts during the assembly processes of global stiffness matrices and global mass matrices, elements will be organized into groups by mesh coloring such that the elements in the same group will not have any common nodes. Moreover, wave equations will be solved by the central difference method (CDM). The processing of different subdomains will take place simultaneously. Meanwhile, for each subdomain, the assembly process of its global matrices and the solution process of its nodal displacements will both be parallelized. In order to communicate the internal forces of the common nodes of different subdomains in an efficient manner, two original message exchange strategies, namely Point-to-Point strategy and All-reduce strategy, are developed based on CUDA-aware MPI [37], allowing information to be communicated between different GPUs without having to go through CPU hosts. The accuracy of the new GPU-based SE formulation is validated by the commercial FE software package Abaqus/Explicit [38]. The computational efficiency of the new formulation is extensively analyzed with respect to a multi-CPU core-based SE implementation which relies on the classical MPI for message exchange. Finally, the applicability of the new formulation is demonstrated through simulating the inspection of thickness loss defects in plates by Lamb waves.

2. Spectral element formulation for ultrasonic wave propagation in solids

The elastodynamics of a discretized solid structure is governed by

$$\mathbf{Ma}(t) + \mathbf{Ku}(t) = \mathbf{F}_{\text{ext}}(t) \tag{1}$$

where $\mathbf{u}$ denotes the displacements, $\mathbf{a}$: the accelerations, $\mathbf{M}$: the global mass matrix, $\mathbf{K}$: the global stiffness matrix, $\mathbf{F}_{\text{ext}}$: the external loads, and t: time. The initial conditions of the structure are defined by $\mathbf{u}^0 = \mathbf{u}(0)$ and $\dot{\mathbf{u}}^0 = \dot{\mathbf{u}}(0)$, where $\mathbf{u}(0)$ and $\dot{\mathbf{u}}(0)$ denote the initial displacements and the initial velocities, respectively.

In this work, 3D solid structures are considered and 125-node spectral elements (SEs) are used for spatial discretization, as shown in **Fig. 1**. The positions of the nodes of the elements are defined by Gauss-Lobatto-Legendre (GLL) quadrature points, and the shape functions of the elements adopt Lagrange polynomials. The mass matrix and stiffness matrix of a 125-node GLL element are given by

$$\mathbf{M}_e = \int_{\Omega_e} \mathbf{N}^T \rho \mathbf{N} d\Omega_e \tag{2a}$$

$$\mathbf{K}_e = \int_{\Omega_e} \mathbf{B}^T \mathbf{D} \mathbf{B} d\Omega_e \tag{2b}$$

where $\mathbf{N}$ contains the shape functions, $\mathbf{B}$ denotes the strain-displacement matrix, constructed based on the spatial derivatives of the shape functions, $\mathbf{D}$: the constitutive matrix of the structure concerned, ρ: the material density of the structure, and Ω_e: the volume of the element.

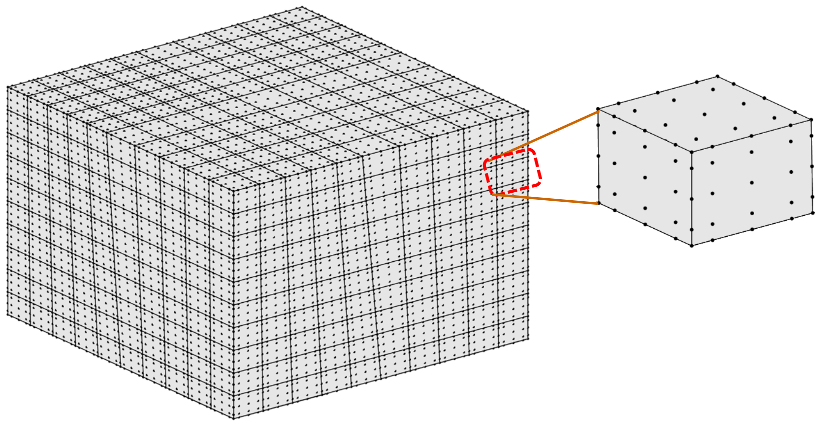


Fig. 1. The spatial discretization of a structure by 125-node GLL elements.

For large-scale elastodynamic problems, explicit time integration is often preferred over implicit time integration, because it demands less computational effort [39-41]. Under explicit time integration, it would be possible to carry out element-by-element computation [42, 43] which does not require global matrices to be assembled and stored, hence saving memory space. However, the time cost of element-by-element computation would be high, because in carrying out element-by-element computation, elemental matrices would need to be re-calculated at every time step. On modern workstations, the sizes of the cache memories are generally significant. Therefore, in solving an ultrasonic wave propagation problem which would very likely involve a large number of time steps, it would be more efficient to have the global matrices of the structure concerned assembled once for all. Global mass matrices and global stiffness matrices can be assembled according to

$$\mathbf{M} = \sum \mathbf{M}_e \tag{3a}$$

$$\mathbf{K} = \sum \mathbf{K}_e \tag{3b}$$

In this work, lumped mass matrices are adopted and only the non-zero diagonal of a matrix is stored. On the other hand, full numerical integrations that are based on the Gauss-Legendre quadrature are employed to construct sparse stiffness matrices. In this case, what are stored are all the non-zero values of a matrix.

Using the so-called regular explicit time integration scheme, the displacements of a structure at the time step $n+1$ can be effectively calculated by the following finite difference equation

$$\mathbf{u}^{n+1} = \Delta t^2 (\mathbf{F}_{ext} - \mathbf{F}_{\text{int}}) / \mathbf{M} + 2\mathbf{u}^n - \mathbf{u}^{n-1} \tag{4}$$

where Δt is the time increment and $\mathbf{F}_{\text{int}} = \mathbf{K}\mathbf{u}^n$ contains the internal forces of the structure.

The computational load of a SE simulation of ultrasonic wave propagation would be dominated by time integration, because the simulation would very likely comprise of a large number of time steps. Since the mesh of the structure concerned would not change over time, under explicit time integration, the computational effort that is associated with each of the time steps in the simulation would be consistent. Moreover, the assembly process of the global matrices of the structure would also incur a one-off cost which contributes to the total computational load of the simulation. Therefore, in optimizing the computational efficiency of the proposed SE formulation for ultrasonic wave propagation, the matrix assembly stage and the time integration stage should be concentrated on.

In this work, the SE formulation for ultrasonic wave propagation is implemented on a multi-GPU cluster. The implementation is designated to carry out parallel, domain decomposition-based computation.

3. Implementation on a multi-GPU cluster with CUDA-aware MPI

3.1. Domain decomposition and meshing

Consider a 3D solid structure. The structure is firstly discretized into 8-node brick elements using the commercial software package Abaqus, and then decomposed into several subdomains under the assistance of the open-source software package METIS [44]. METIS can simultaneously minimize the sizes of the common boundaries of the subdomains and maximize the fairness of the distribution of elements among the subdomains. As such, the communications between the different GPUs that will be used for computation will be minimized [45, 46], and the computational loads of the different GPUs will be maximally balanced. Finally, the 8-node brick elements of the structure are transformed into 125-node GLL elements by an inhouse code that is programmed using C. It worth mentioning that domain decompositions and meshing are executed by CPUs.

Fig. 2 shows the decomposition of a structure into four subdomains. In this work, a one-to-one match between GPUs and CPU cores is adopted, meaning that when multiple GPUs are used, each GPU is controlled by a separate CPU core. Therefore, in practice, the structure shown in **Fig. 2** would be processed by four CPU core/GPU pairs.

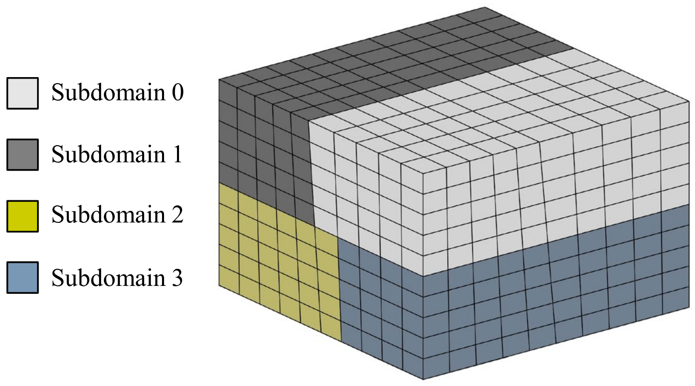


Fig. 2. The domain decomposition of a 3D solid structure.

3.2. Mesh coloring and assemblies of global matrices

Since a GPU houses a large number of processors, it can potentially calculate the stiffness matrices and mass matrices of a group of SEs simultaneously. However, if these elements have common nodes, memory access conflicts could occur when the stiffness matrices and mass matrices of these elements are used to update the corresponding global stiffness matrix and global mass matrix. In view of this, mesh coloring, which has been widely used in FE formulations to avoid conflicted updates of global matrices [47, 48], is adopted in this work. More specifically, the greedy coloring algorithm [49] is employed to accomplish mesh coloring. As shown in **Fig. 3**, the SEs in each of the subdomains of a structure are divided into subsets. The subsets are each marked by a unique color and the elements in the same subset do not have any common nodes. Since mesh coloring is a one-off process in a simulation and its computational cost is very small, it is programmed as a serial algorithm and executed by CPUs.

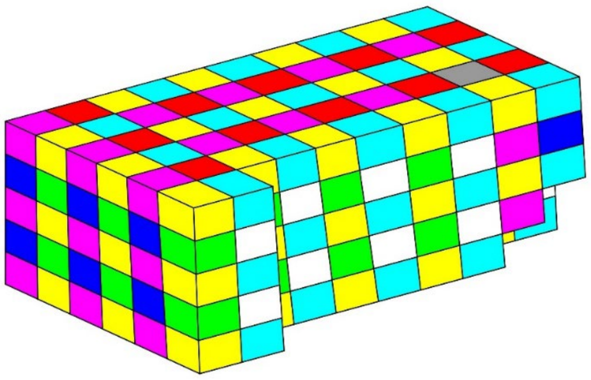


Fig. 3. The division of the SEs in a subdomain of a 3D solid structure by mesh coloring.

After the SEs in a subdomain have been divided into subsets based on mesh coloring, the details of the SEs and the element subsets are transferred to the global memory of the GPU that is dedicated to processing the subdomain. Two 1D arrays are created in the global memory, one to store the non-zero diagonal of the global mass matrix of the subdomain, and the other one to store all the non-zero values of the global stiffness matrix of the subdomain. By storing only the non-zero values of global matrices, memory usage will be minimized and computation around zero terms will be avoided, enhancing computational efficiency.

A kernel for assembling the global stiffness matrix and global mass matrix of a subdomain is programmed. The kernel is launched once for each of the element subsets of a subdomain. **Fig. 4** shows the computational arrangement that is adopted by this matrix assembly kernel. While each of the nodes of a SE is assigned with one thread to calculate its elemental stiffness matrix entries and elemental mass matrix entries, the threads that deal with the same SE are arranged into one block. In a GPU, the SPs in each SM are physically divided into groups of 32. Every group of 32 SPs is regarded as a warp and can only execute the same command. What's more, the SPs each compute one thread at a time, and the threads in the same block must be computed within the same SM. Since the SEs used in this work each have 125 nodes, the block_size (i.e., threads per block) for the matrix assembly kernel is fixed at 128, where 128 is the smallest multiple of 32 that is greater than 125. Although in every block three threads will be idle during computation, the remaining 125 threads, which will be active, can be computed in a synchronized manner and access the same shared memory space. On the other hand, the grid_size (i.e., blocks per grid) for the kernel is always set as the number of elements in the subset that it is designated to process.

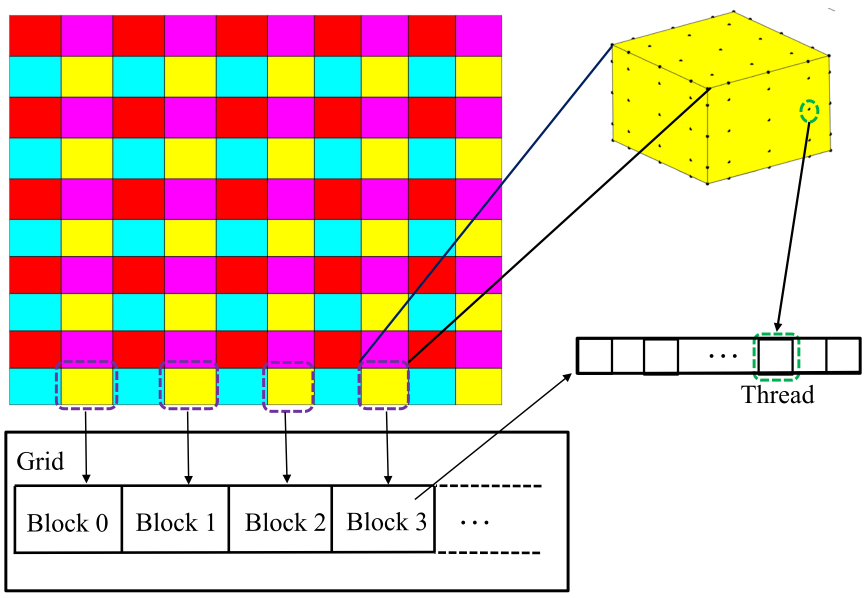
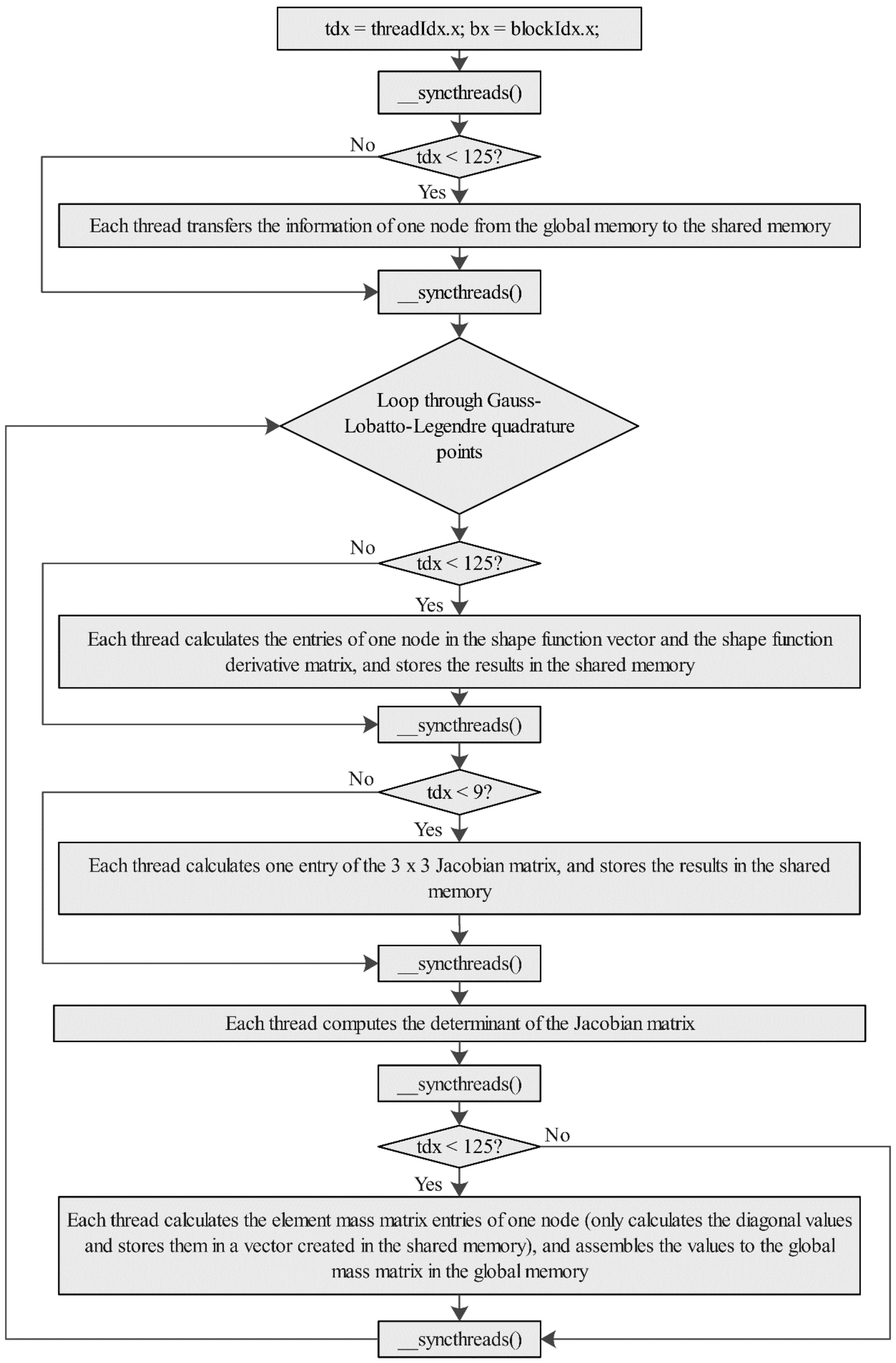


Fig. 4. The matching of threads and blocks to nodes and elements in the matrix assembly kernel.

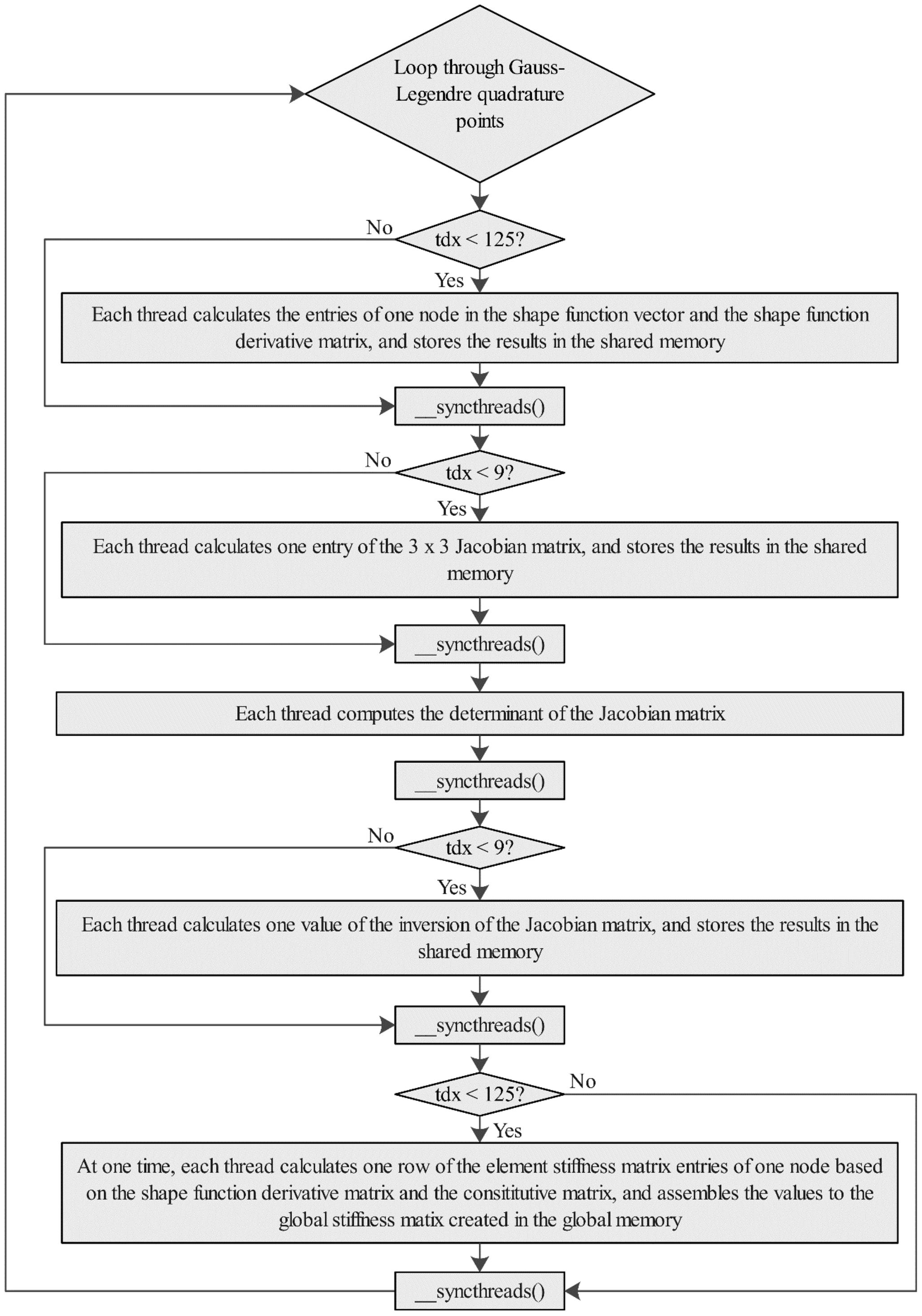
The assembly processes of the global stiffness matrix and global mass matrix of a subdomain are illustrated in **Fig. 5**. For each of the SEs in an element subset, its shape functions, the spatial derivatives of its shape functions and its Jacobian matrix are all stored in the shared memory space of the responsible block, so that they can be accessed at a high speed. Once the non-zero diagonal of the mass matrix of a SE has been calculated, it is also stored in the shared memory space of the responsible block, before it is transferred to the global memory of the GPU to update the global mass matrix of the subdomain. When calculating the stiffness matrix of a SE, the strain-displacement matrix of the SE is not explicitly obtained and, instead, the spatial derivatives of the shape functions of the SE are utilized. Since the stiffness matrices of the SEs used in this work are relatively large, i.e., 475 × 475, the size of the shared memory of a block might not be sufficient for storing the stiffness matrix of the element that the block is assigned to process. Therefore, for each of the nodes of a SE, the calculation of its 3 × 3 stiffness matrix entries is carried out over three loops, one row at a time. In each loop, the resultant stiffness matrix entries are stored in the register memory space or the local memory space of the responsible thread, and then transferred to the global memory of the GPU to update the global stiffness matrix of the subdomain. By doing so, the stiffness matrices of SEs do not need to be stored in the shared memory spaces of blocks. In fact, due to the scarcity of memory space on graphics cards, balances between calculations and data storage should always be sought.

3.3. Communication

Under the CUDA platform, both the *cudaMemcpy* function and the *Peer-to-Peer* function can be used to realize direct message exchange between different GPUs. However, these two CUDA functions were found to be inapplicable to this work, because in this work GPUs and CPU cores are matched on a one-to-one basis but the data that is transferred from one GPU to another using either one of these functions would not be visible to the CPU host of the destination GPU. One alternative strategy would be to transfer data between two GPUs via their CPU hosts [36]. During this process, the communication between a GPU and a CPU core would be achieved using the *cudaMemcpy* function, and that between the two CPU cores would make use of the classical MPI. However, an indirect message exchange strategy as such would no doubt be quite time-consuming.


tdx = threadIdx.x; bx = blockIdx.x;
__syncthreads()
tdx < 125?
No
Yes
Each thread transfers the information of one node from the global memory to the shared memory
__syncthreads()
Loop through Gauss-Lobatto-Legendre quadrature points
tdx < 125?
No
Yes
Each thread calculates the entries of one node in the shape function vector and the shape function derivative matrix, and stores the results in the shared memory
__syncthreads()
tdx < 9?
No
Yes
Each thread calculates one entry of the 3 x 3 Jacobian matrix, and stores the results in the shared memory
__syncthreads()
Each thread computes the determinant of the Jacobian matrix
__syncthreads()
tdx < 125?
No
Yes
Each thread calculates the element mass matrix entries of one node (only calculates the diagonal values and stores them in a vector created in the shared memory), and assembles the values to the global mass matrix in the global memory
__syncthreads()

(a)


Loop through Gauss-Legendre quadrature points
tdx < 125?
No
Yes
Each thread calculates the entries of one node in the shape function vector and the shape function derivative matrix, and stores the results in the shared memory
__syncthreads()
tdx < 9?
No
Yes
Each thread calculates one entry of the 3 x 3 Jacobian matrix, and stores the results in the shared memory
__syncthreads()
Each thread computes the determinant of the Jacobian matrix
__syncthreads()
tdx < 9?
No
Yes
Each thread calculates one value of the inversion of the Jacobian matrix, and stores the results in the shared memory
__syncthreads()
tdx < 125?
No
Yes
At one time, each thread calculates one row of the element stiffness matrix entries of one node based on the shape function derivative matrix and the consititutive matrix, and assembles the values to the global stiffness matix created in the global memory
__syncthreads()

(b)

Fig. 5. The processes for calculating (a) the global mass matrix entries and (b) the global stiffness matrix entries of each of the SEs in a subdomain.

In this work, the CUDA platform is integrated with classical MPI to enable CUDA-aware MPI, allowing data to be communicated between different GPUs directly without going through CPU hosts. Two message exchange strategies, namely Point-to-Point strategy and All-Reduce strategy, are developed based on CUDA-aware MPI to exchange, between different GPUs, the internal forces of the common nodes of a structure. Here, a node is defined as a common node if it is shared by two or more subdomains of a structure, but not necessarily by all the subdomains. While the Point-to-Point strategy makes use of the *MPI_Send* function and the *MPI_Recv* function which have been commonly employed by parallel FE and SE implementations, the All-Reduce strategy is an alternative that can be achieved with simplified programming. The kernels for message exchange are provided in detail in the Appendix. In order to carry out message exchange, the different subdomains of a structure are each assigned with a MPI rank, as shown in **Fig. 6**.

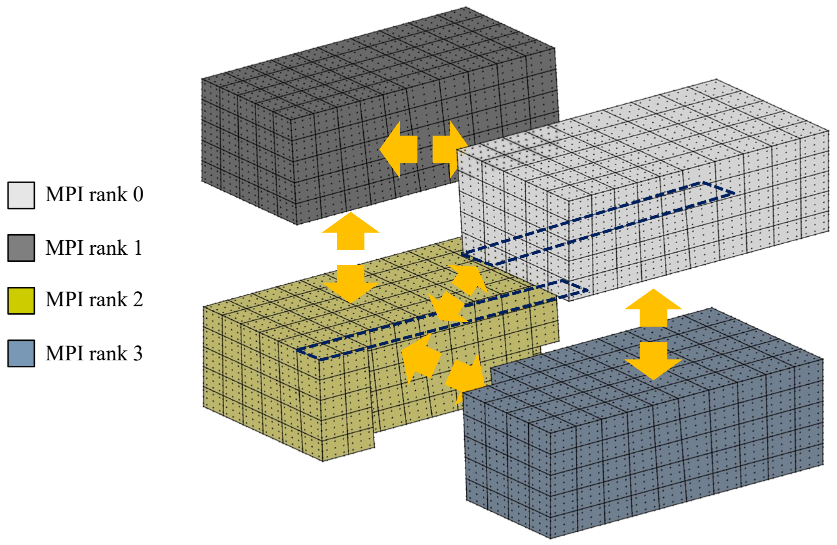


Fig. 6. The assignment of MPI ranks to the different subdomains of a structure.

3.3.1. Point-to-Point strategy

For each of the subdomains of a structure, its nodes are numbered locally with respect to their positions in the subdomain. As a result, the common nodes in the structure each possess a number of local indices. Communication paths are established between the different MPI ranks that are assigned to the subdomains, e.g., if MPI rank X and MPI rank Y share common nodes, two communication paths would be established between them, one from MPI rank X to MPI rank Y, and the other from MPI rank Y to MPI rank X. For each communication path, the common nodes that are shared by the two MPI ranks concerned are identified and the local indices of these nodes on both the native MPI rank and the target MPI rank are found. The establishment of communication paths and the identification of the local indices of common nodes are executed by CPUs. Table 1 shows the communication paths that would be established for the structure shown in **Fig. 6**.

Table 1. The communication paths between the different subdomains of the structure shown in **Fig. 6**.

Native MPI rank	rank 0	rank 0	rank 1	rank 1	rank 2	rank 1	rank 3	rank 2	rank 3	rank 3
Target MPI rank	rank 1	rank 3	rank 2	rank 3	rank 3	rank 0	rank 0	rank 1	rank 1	rank 2

For each MPI rank, three temporary buffers, **Send_buffer**, **Recv_buffer** and **Arr_con**, are created in its global memory. The sizes of the three temporary buffers are all set to $3 \times nc$, where *nc* denotes the number of common nodes in the whole structure. For each communication path, the contributions of the native MPI rank to the internal forces of the common nodes are extracted from the internal force array (**F_int**) of the native MPI rank, and stored in **Send_buffer**, using Kernel 1 (see the Appendix). The *MPI_Send* function is launched to send the data in **Send_buffer** to the target MPI rank, and the *MPI_Recv* function to receive the data at the target MPI rank and store it in **Recv_buffer**. Using Kernel 2, the nodal internal forces in **Recv_buffer** are re-arranged according to the local indices of the common nodes on the target MPI, and added to **Arr_con**. After the all the communication paths have been looped through, for each MPI rank, the nodal internal forces in **Arr_con** are added to its internal force array, using Kernel 3. The algorithm of the Point-to-Point strategy is illustrated in Algorithm 1. When Kernels 1, 2 and 3 are executed, the nodal values concerned are each processed by a single thread.

Algorithm 1. The Point-to-Point strategy for exchanging the internal forces of common nodes.

```
1. For each MPI rank, create three buffers, Send_buffer, Recv_buffer and Arr_con, in its
   global memory.

2. Loop through the communication paths $ip$ = 1, 2, …, $n$.

    3. For each MPI rank, if it is equal to the native MPI rank of the current communication
       path ($ip$)…

        4. Extract, from its internal force array, the internal forces of the common nodes of the
           current communication path, and store them in Send_buffer, using Kernel 1.
```

```
            5. Send the data in Send_buffer to the target MPI rank, using MPI_Send.
        6. For each MPI rank, if it is equal to the target MPI rank of the current communication
           path (ip)…
            7. Receive the data in Send_buffer, and store it in Recv_buffer, using MPI_Recv.
            8. Add the data in Recv_buffer to Arr_con, using Kernel 2.
    9. End the loop over the communication paths.
    10. For each MPI rank, add the data in Arr_con to the relevant entries of its internal force array,
        using Kernel 3.
```

3.3.2. All-Reduce strategy

For a structure that is decomposed into several subdomains, all of its common nodes are identified and assigned with a set of global indices. For each common node, its local indices on all the subdomains that it is shared by are also identified. The identifications of common nodes and their local indices are executed by CPUs.

For each MPI rank, two temporary buffers, **Arr_con1** and **Arr_con2**, are created in its global memory. The sizes of the two temporary buffers are all set to $3 \times nc$, where *nc* denotes the number of common nodes in the whole structure. Also, the two temporary buffers are all sequenced with respect to the global indices of the common nodes. For each MPI rank, the internal forces of the common nodes that it hosts are extracted from its internal force array (**F_int**), and stored in the entries of **Arr_con1** that correspond to these common nodes, using Kernel 4. The remaining entries of **Arr_con1**, which correspond to the other common nodes in the structure, are assigned with zero. Once the temporary buffers **Arr_con1** of all the MPI ranks have been updated, the *MPI_Allreduce* function is launched to sum all the temporary buffers **Arr_con1**, and store the result in the temporary buffer **Arr_con2** of each MPI rank. Finally, for each MPI rank, its internal force array entries that correspond to the common nodes that it hosts are replaced by the relevant nodal internal forces in **Arr_con2**, using Kernel 5. The details of the All-Reduce strategy are provided in Algorithm 2. In executing Kernels 4 and 5, every nodal value is processed by a single thread.

Algorithm 2. The All-Reduce strategy for exchanging the internal forces of common nodes.

1. For each MPI rank, create two buffers, **Arr_con1** and **Arr_con2**, in its global memory.

2. For each MPI rank, extract, from its internal force array, the internal forces of the common nodes that it hosts, and store them in **Arr_con1**, using Kernel 4.

3. Sum the data in the buffers **Arr_con1** of all the MPI ranks, and store the result in the buffers **Arr_con2** of all the MPI ranks, using *MPI_Allreduce*.

4. For each MPI rank, replace its internal force array entries that correspond to the common nodes that it hosts by the relevant entries in **Arr_con2**, using Kernel 5.

For a given communication task, the Point-to-Point strategy would accomplish it over multiple transfers of small buffers, and the All-Reduce strategy would accomplish it over a single transfer of a large buffer.

3.4. Explicit time integration

The CDM [50] is used to solve wave equations explicitly. The two kernels for time integration are detailed in the Appendix. At each of the time steps of a simulation, Kernel 6 is used to calculate the internal force array (**F_int**) of each of the subdomains of the structure concerned, according to $\mathbf{F}_{\text{int}} = \mathbf{K}\mathbf{u}^{n}$. The internal force arrays of the different subdomains are then exchanged and updated, using one of the two message exchange strategies that are introduced in Section 3.3. Finally, Kernel 7 is used to calculate the new displacement arrays of each of the subdomains of the structure at the next time step, according to Eq. (4). Throughout the time integration process, the nodes of the structure are each assigned to a single thread to process, meaning that at each time step, each thread is responsible for calculating three internal force values and three displacement values (because the structure is 3D). The procedure of the CDM is given in Algorithm 3. Since external force arrays (**F_ext**), internal force arrays (**F_int**) and displacement arrays (time step $n-1$: **u0**, time step n: **u1**, time step $n+1$: **u2**) are all relatively large, they are always stored in the global memories of GPUs.

Algorithm 3. The central different method for performing time integration.

```
1. Initially calculation

    i) For each MPI rank, assemble its global stiffness matrix $\mathbf{K}$ and global mass matrix $\mathbf{M}$.
    ii) For each MPI rank, define the initial conditions u0, u1.

2.  At every time step

    i) For each MPI rank, calculate its internal force array at the current time step via
       $\mathbf{F}_{\text{int}} = \mathbf{K} \times \mathbf{u1}$, using Kernel 6.

    ii) Exchange, between the different MPI ranks, the internal forces of the common nodes.

    iii) For each MPI rank, calculate its displacement array at the next time step via Eq. (4),
         using Kernel 7.

    iv) Move onto the next time step.
```

4. Numerical experiments

The SEM introduced in Section 3 was also implemented on multiple CPU cores, providing a benchmark for assessing the accuracy and the efficiency of the proposed multi-GPU and CUDA-aware MPI-based SE formulation. In order to realize message exchange in the CPU-based SE formulation, the Point-to-Point strategy and the All-Reduce Strategy were also developed using the classical MPI. The CPU workstation that was used in this work comprises of two Intel® Xeon® Silver 4122 Processors @ 2.6 GHz, each with 4 cores. On the other hand, the GPU workstation used consists of four NVIDIA® GeForce® 1080 Ti graphics cards.

4.1. Accuracy assessment

Fig. 7 depicts the model that was used for assessing the accuracy of the proposed GPU-based SE formulation. A dynamic pressure $P(t) = A_0 \sin(2\pi ft) \times \sin(\pi ft/5)^2$ (peak amplitude: $A_0 = 10\text{MPa}$, central frequency: $f = 1000\text{kHz}$) is applied to the center of the top surface of the structure over a 10 mm × 10 mm area. A sensor signal is acquired at the mid-point of the bottom surface of the structure. The SE simulations of the model, performed using either the GPU-based formulation or the CPU-based formulation, were all executed with four MPI ranks, i.e., four GPUs or four CPU cores. In these simulations, the structure was discretized into 125-node GLL elements, and the stable time increment was set to 1 ns to ensure both accuracy and efficiency.

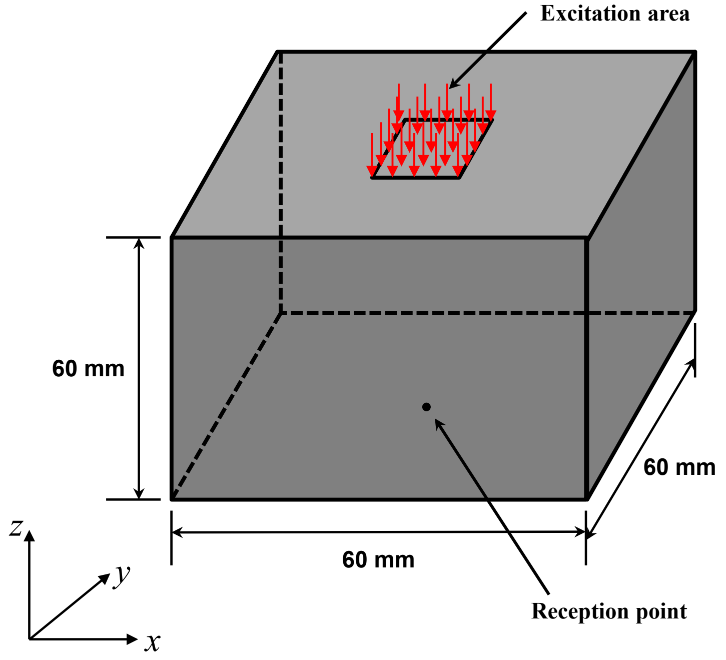
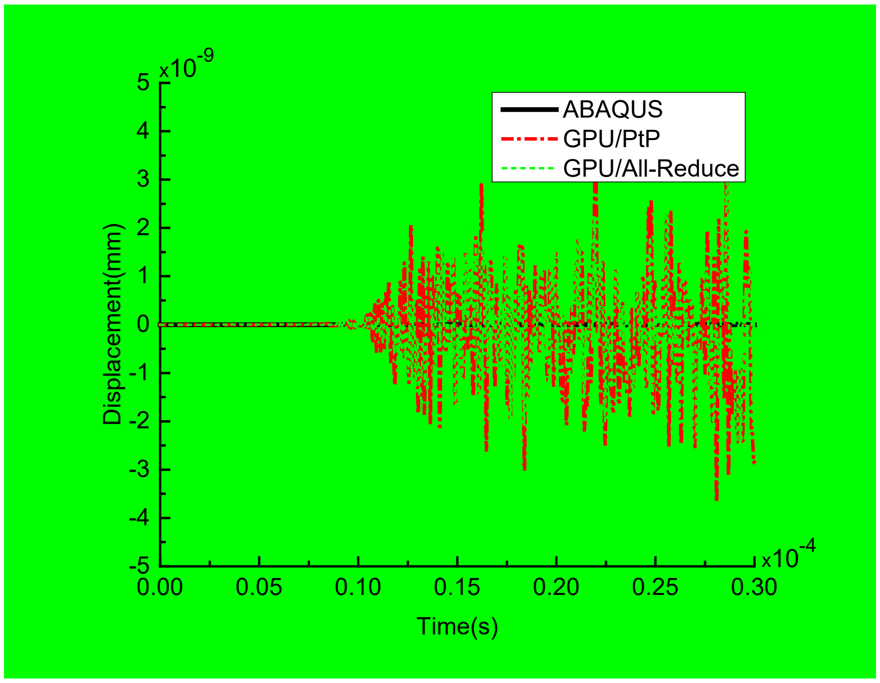
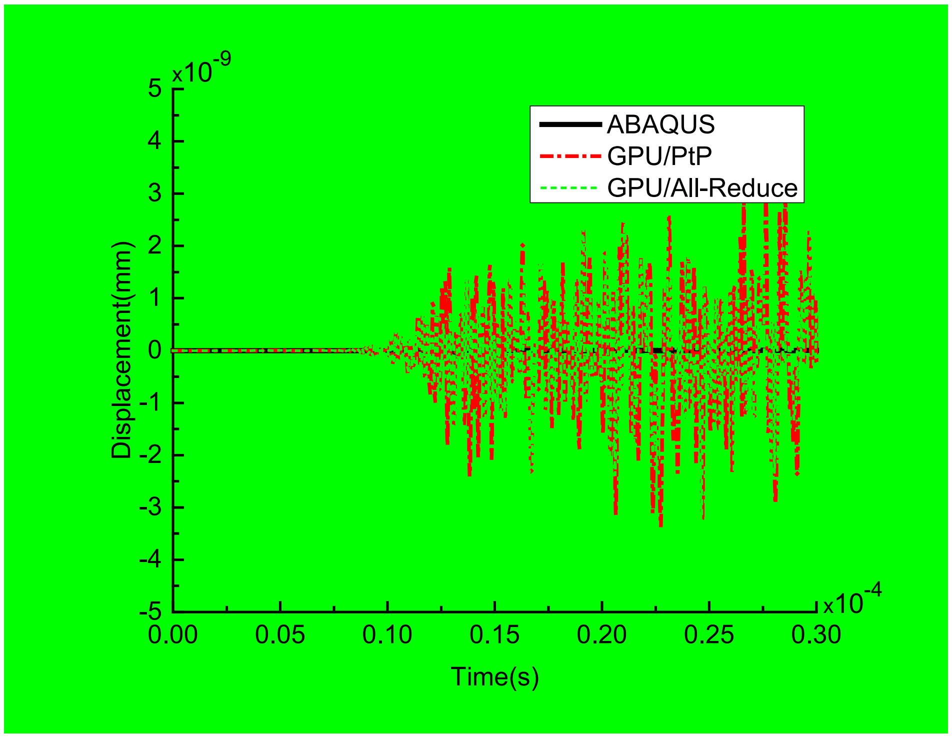


Fig. 7. The structure used for accuracy assessment purpose (Young's modulus: 70 GPa, Poisson's ratio: 0.33, density of 2700 kg/m^3).

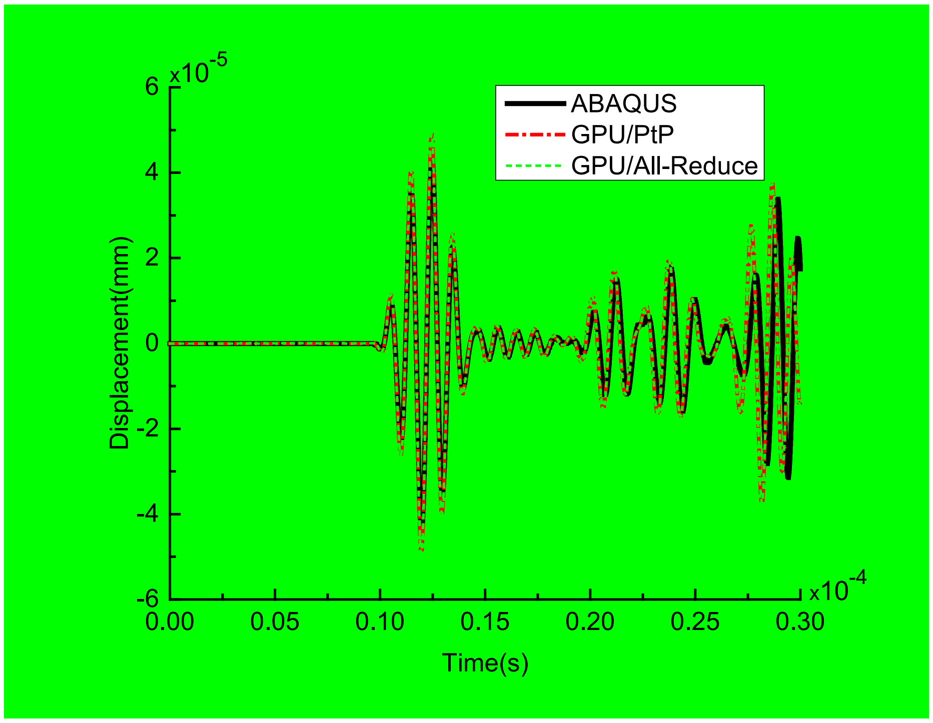
At first, the proposed GPU-based SE formulation was validated by the commercial FE software package Abaqus/Explicit. As shown in **Fig. 8**, both the GPU-based SE simulation results and the FE simulation results indicate that only longitudinal waves, which propagated in the z-direction, were captured. In the GPU-based SE simulation results, the x-direction and y-direction displacements, which are four orders of magnitude smaller than the z-direction displacements, are essentially numerical noise. The two sets of z-direction displacements that were obtained under the two different message exchange strategies do not exhibit any noticeable discrepancy. Also, they match very well with z-direction displacements in the FE simulation results.



(a)


x10^-9
ABAQUS
GPU/PtP
GPU/All-Reduce
Displacement(mm)
5
4
3
2
1
0
-1
-2
-3
-4
-5
x10^-4
0.00
0.05
0.10
0.15
0.20
0.25
0.30
Time(s)

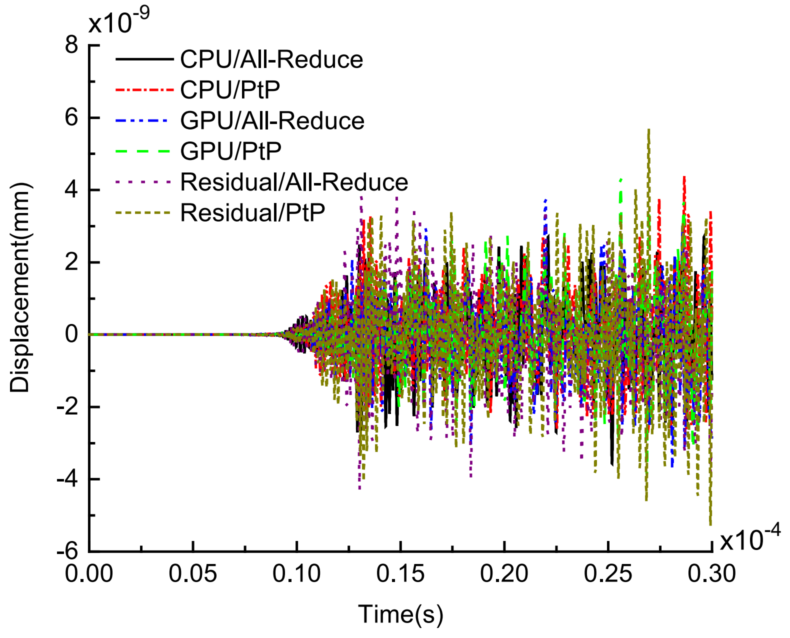
(b)


x10^-5
ABAQUS
GPU/PtP
GPU/All-Reduce
Displacement(mm)
6
4
2
0
-2
-4
-6
x10^-4
0.00
0.05
0.10
0.15
0.20
0.25
0.30
Time(s)

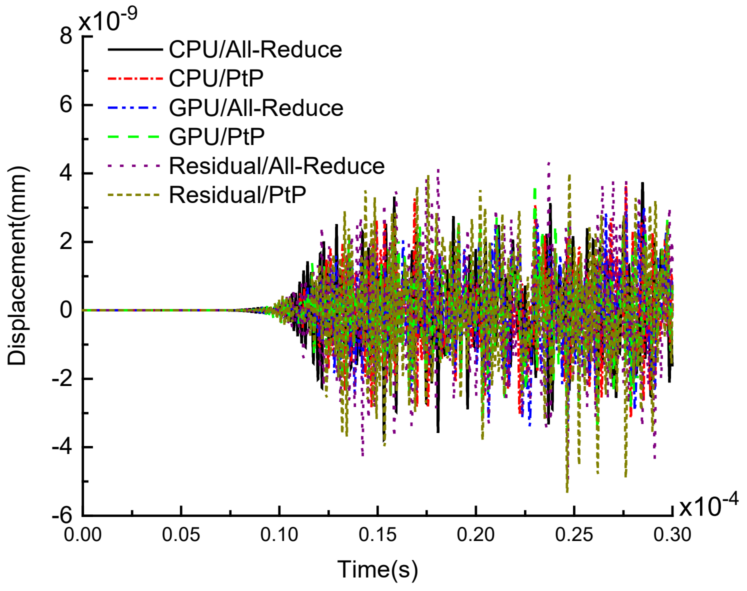
(c)

Fig. 8. The sensor signals acquired through multi-GPU-based SE simulations and Abaqus/Explicit FE simulations: (a) x-direction, (b) y-direction and (c) z-direction.

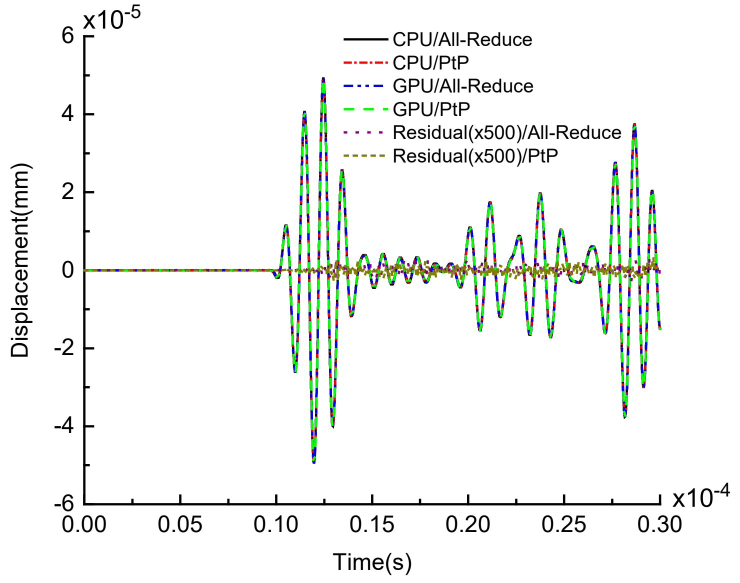
Then, the accuracy of the benchmark CPU-based SE formulation was assessed with respect to the proposed GPU-based SE formulation. As demonstrated in **Fig. 9**, the x-direction and y-direction displacements in the CPU-based SE simulation results are expectedly also within the noise level. The four sets of z-direction displacements that were obtained under the four different combinations of processing unit type (i.e., CPU or GPU) and message exchange strategy (i.e., Point-to-Point or All-Reduce) are highly identical. Under either message exchange strategy, the residual between the z-direction displacements in the CPU-based SE simulation results and those in the GPU-based SE simulation results was found to be negligible. Note that the residuals plotted have actually been scaled up by 500 times to enhance their visibility.



(a)



(b)



(c)

Fig. 9. The sensor signals acquired through multi-CPU core-based SE simulations and multi-GPU-based SE simulations: (a) x-direction, (b) y-direction and (c) z-direction.

4.2. Efficiency analysis

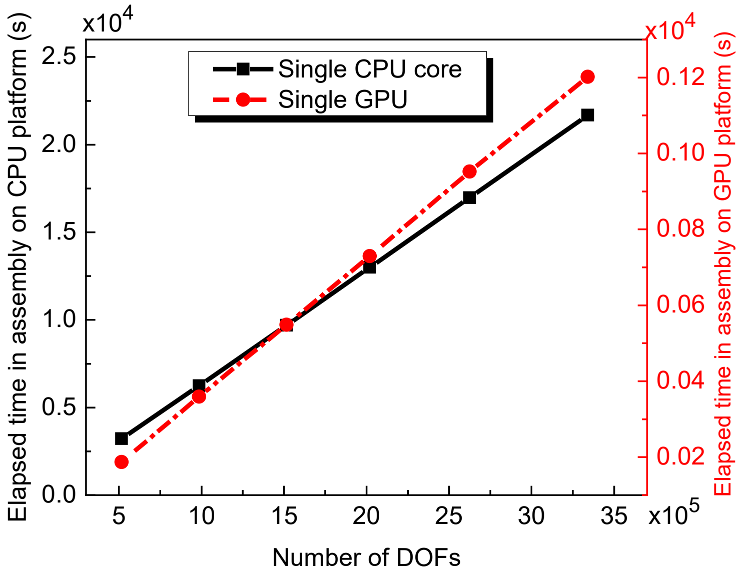
The model that is depicted in **Fig. 7** was further utilized for evaluating the computational speedups that can be achieved by the proposed multi-GPU and CUDA-aware MPI-based SE formulation in matrix assembly, wave equation computation and message exchange. The benchmark multi-CPU core and classical MPI-based SE formulation was used as the reference. To simulate structures with more DOFs and more common nodes, the model was gradually enlarged in all three directions. For a given scenario, the GPU speedup is calculated by

$$speedup = CPU_{time} / GPU_{time}$$

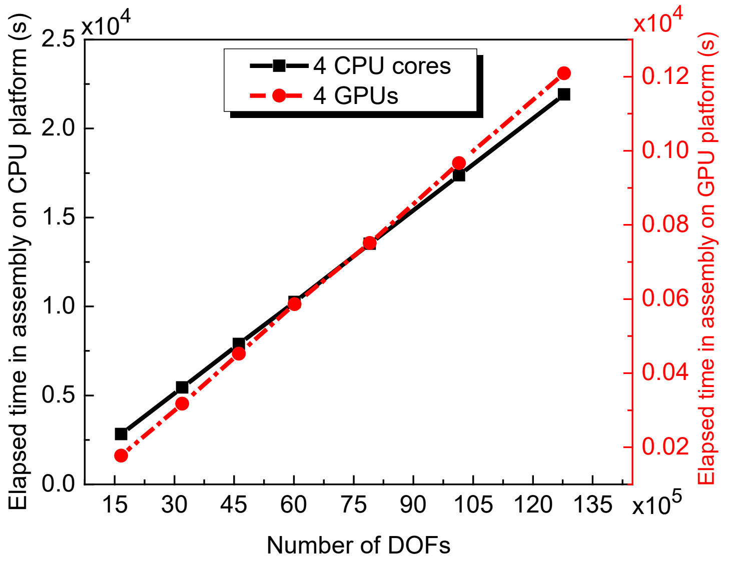
It is worth mentioning that the benchmark CPU-based SE formulation was structured in the same way as the proposed GPU-based SE formulation was, maximally ensuring the fairness of the comparison between the two formulations. While the proposed GPU-based SE formulation can be coupled with any number of GPUs, the largest number of GPUs that was considered in this work is four because the workstation used only has four graphics cards.

4.2.1. Global matrix assembly

Global matrix assembly processes often induce considerable time costs, especially if the models concerned are large. **Fig. 10** shows the times that were spent on assembling the global matrices of models with different numbers of DOFs, using different numbers of GPUs or CPU cores. It is worth mentioning that the largest number of DOFs that a certain number of GPUs could handle would be limited by the total size of the global memories of all the GPUs. It can be seen from **Fig. 10** that using either GPUs or CPUs, the elapsed time of the global matrix assembly process of a model increases with the number of DOFs in the model. This owes to the fact that the global matrix assembly process of a model mainly consists of repeated calculations of the elemental matrices of the model, rendering the number of elements or DOFs in a model the most dominant factor in determining the elapsed time of the global matrix assembly process of a model.



(a)



(b)

Fig. 10. The elapsed times of the assembly processes of global matrices of different dimensions: (a) one MPI rank and (b) four MPI ranks. Each assembly process embodies a global stiffness matrix and a global mass matrix, both of which have the same dimension.

As shown in **Fig. 11**, the GPU speedups under the different numbers of DOFs considered and under either one MPI rank or four MPI ranks all lie within the range of 16-18 times. The difference between the computational speed of the proposed GPU-based SE formulation and that of the benchmark CPU-based SE formulation is attributed to the fact that on a CPU core, elemental matrices are calculated one after another, whereas on a GPU, a large number of blocks can be formed to calculate multiple elemental matrices simultaneously, and within each block, the threads are executed in parallel to calculate the elemental matrix entries of all the nodes concerned. Also, the observation that the GPU speedup under either one MPI rank or four MPI ranks increases monotonically with the number of DOFs renders the proposed GPU-based SE formulation promising for large-scale applications.

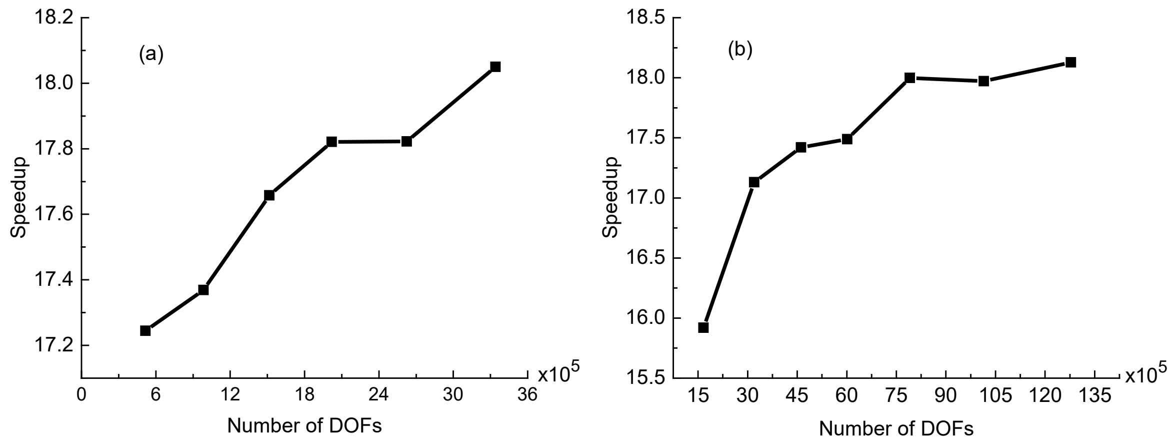
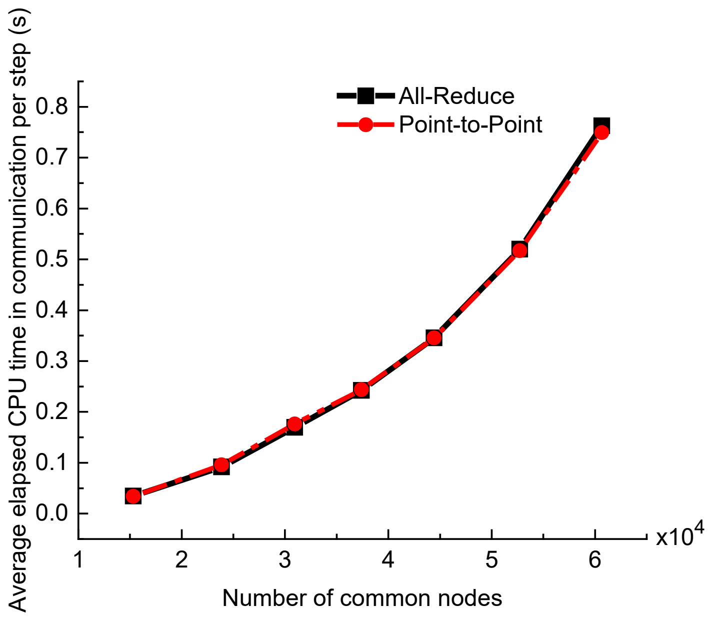


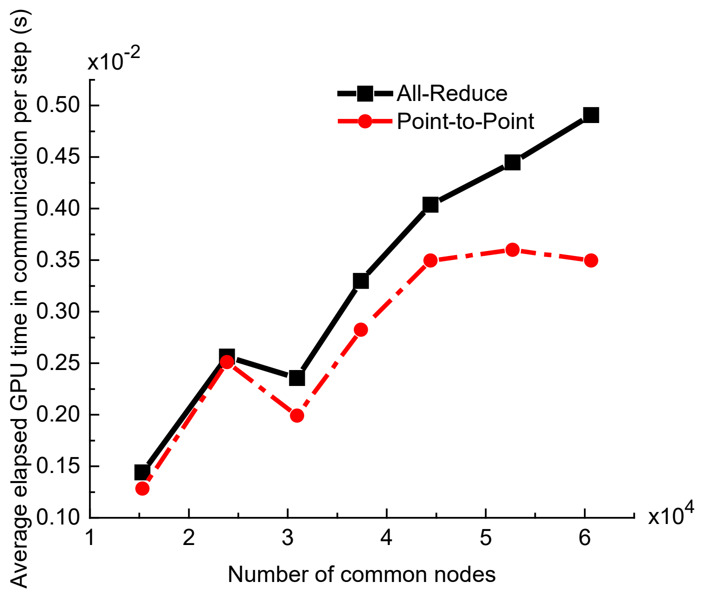
Fig. 11. The GPU speedups achieved in the assembly processes of global matrices of different dimensions: (a) one MPI rank and (b) four MPI ranks.

4.2.2. Communication

Both the classical MPI and the CUDA-aware MPI offer many functions for realizing message exchange in multi-processing unit environments. In this section, the times that were needed to exchange, between four GPUs or four CPU cores, the internal forces of different numbers of common nodes were examined. As shown in **Fig. 12**(a), in the multi-CPU core environment, the two message exchange strategies that are introduced in this work, i.e., the Point-to-Point strategy and the All-Reduced strategy, require the same amount of time to exchange the internal forces of the same number of common nodes. However, **Fig. 12**(b) shows that in the multi-GPU environment, the Point-to-Point strategy is more efficient, and its efficiency over the All-Reduce strategy increases with the number of common nodes. Moreover, it can be seen that in the multi-CPU core environment, the elapsed time of a message exchange step increases exponentially with the number of common nodes, whereas in the multi-GPU environment, the rate of the increase trend gradually slows down, and even plateaus, in the case of the Point-to-Point strategy. Furthermore, for the multi-GPU environment, the sudden drop in the relationship between the elapsed time of a message exchange step and the number of common nodes owes to the fact that if the computational load that is imposed upon a GPU is smaller than the computational capability of the GPU, additional memory alignment and thread alignment would need to take place, causing the computational performance of the GPU to be relatively unstable.



(a)



(b)

Fig. 12. The average elapsed times of the message exchange steps of the internal forces of different number of common nodes: (a) four CPU cores and (b) four GPUs. Each average elapsed time was calculated based on 6000 message exchange steps.

Fig. 13 shows that for the different numbers of common nodes considered, the maximum GPU speedups under the Point-to-Point strategy and the All-Reduced strategy are 220 times and 170 times, respectively. The GPU speedups would further increase if the number of common nodes becomes greater. It should be noted that in a message exchange step, apart from data transfer, processes like data extraction and distribution also induce certain time costs. Therefore, the use of GPUs would have also accelerated those processes. Moreover, the GPU speedups under the Point-to-Point strategy are generally higher than those under the All-Reduce strategy. This is because in a message exchange step, for each of the communication paths, the Point-to-Point strategy exchanges only the internal forces of the common nodes that are shared by the two subdomains concerned, whereas the All-Reduce strategy exchanges the internal forces of all the common nodes of the model, resulting in a longer elapse time. A lot of the data that is exchanged by the All-Reduce strategy may not be relevant to the two subdomains concerned. Nevertheless, the Point-to-Point strategy is more sophisticated to implement because for a given model, it requires prior knowledge of all the communication paths and the common nodes under each communication path. On the other hand, the All-Reduce strategy simply exchanges, through all the communication paths, the same dataset.

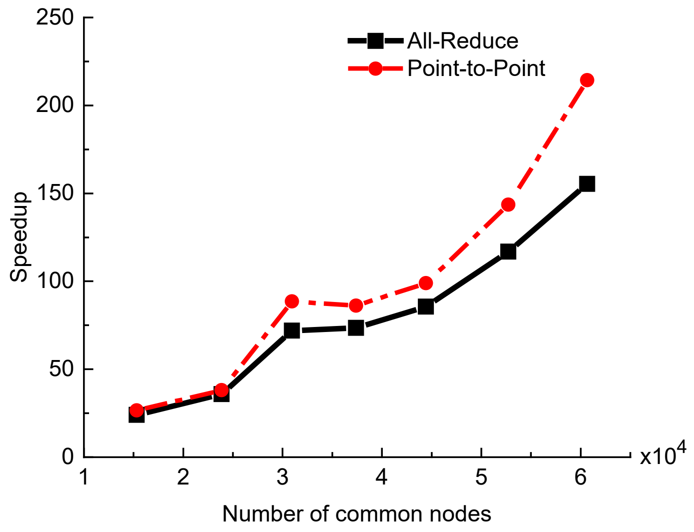
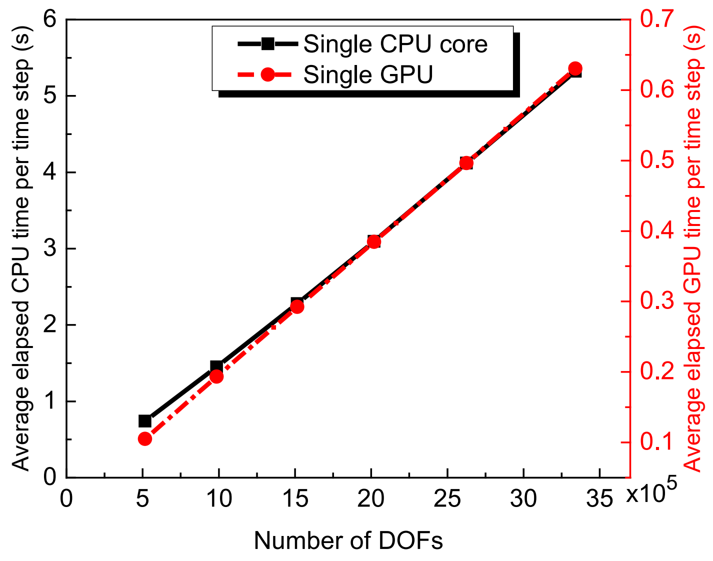


Fig. 13. The GPU speedups achieved in the message exchange steps of the internal forces of different number of common nodes.

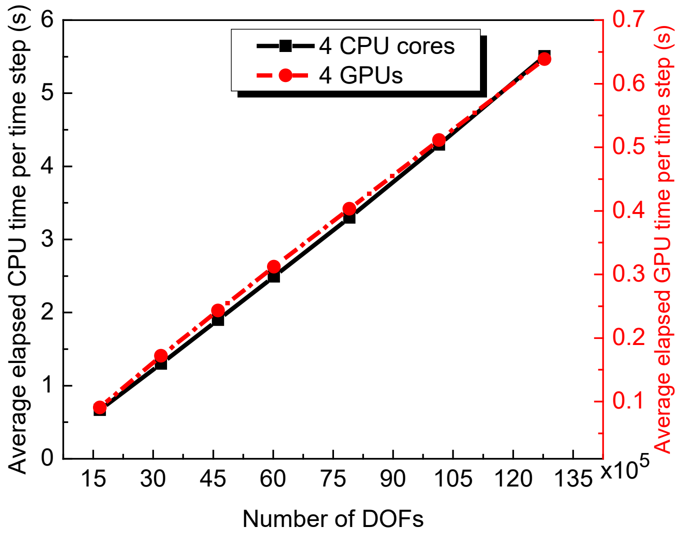
It is worth noting that fluctuations are observed amid both the increasing trend shown in **Fig. 11** and that shown in **Fig. 13**. This could be attributed to the fact that while the computational speed of a model would be influenced by the allocation of threads to processors and the assignment of data to memory spaces, each model could be computed under a different thread and memory arrangement, depending on the nature of the model.

4.2.3. Time integration

Generally speaking, for an elastodynamic simulation, its overall time cost is dominated by the time cost of the time integration, since the simulation would very likely consist of a large number of time steps. **Fig. 14** shows the times that were spent on executing the time integration steps for models with different numbers of DOFs, on different numbers of GPUs or CPU cores. Here, only the Point-to-Point strategy was adopted for message exchange, since it has already been proven to be more efficient than the All-Reduce strategy. As illustrated in **Fig. 14**, using either GPUs or CPUs, the elapsed time of a time integration step increases with the number of DOFs. Also, as shown in **Fig. 15,** under either one MPI rank or four MPI ranks, the GPU speedup increases monotonically with the number of DOFs, reaching approximately 8.5 times in the case of the largest model considered. The superiority of the proposed GPU-based SE formulation in time integration is attributed to the explicit matrix operations in the CDM which can be well executed in a parallelized manner. In this case, for given model, its nodes are matched with threads on a one-to-one basis, and the threads are executed simultaneously to calculate the internal forces and the displacements of the nodes.



(a)



(b)

Fig. 14. The average elapsed times of the time integration steps of models with different DOFs: (a) one MPI rank and (b) four MPI ranks. Each average elapsed time was calculated based on 6000 time integration steps.

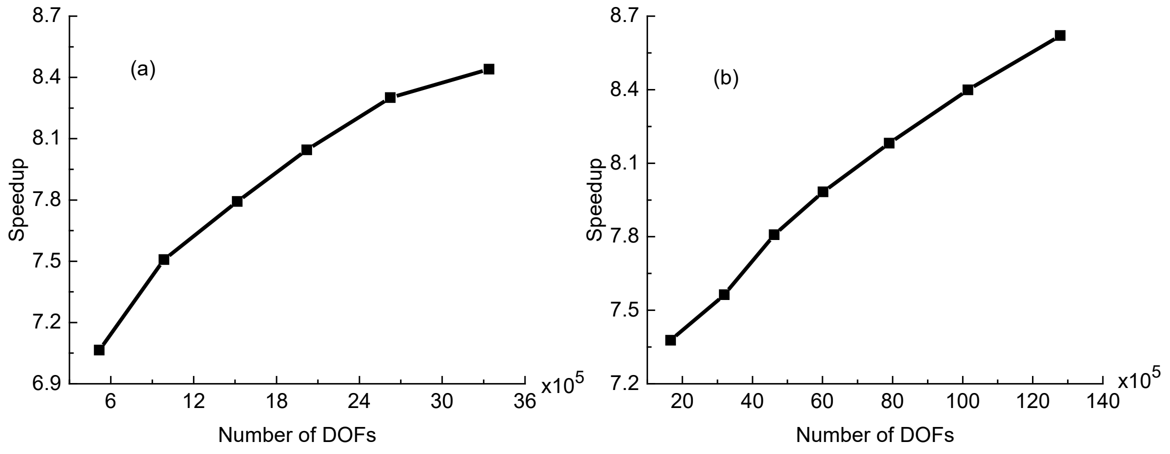


Fig. 15. The GPU speedups achieved in the time integration steps of models with different DOFs: (a) one MPI rank and (b) four MPI ranks.

Table 2 compares the proposed formulation to a couple of other noteworthy GPU-based SE formulations [36, 51] in terms of the maximum GPU speedup that can be achieved in time integration. It can be seen that the proposed formulation is well on par with the *start of the art*, proving itself to be a valid and feasible alternative. The discrepancy between the GPU speedups that can be achieved by the different formulations could be attributed to a couple of factors. First of all, the GPU speedup that a given formulation could achieve would be influenced by the difference between the computational capability of the GPUs used and that of the CPUs used. Since the three formulations under comparison each employ a different combination of GPUs and CPUs, the GPU speedups that they can achieve will no doubt be different. Secondly, Komatitsch et al. or Kudela's formulation does not carry out one-off global matrix assemblies at the beginning of a simulation like what the proposed formulation does, but rather calculates elemental matrices at each time integration step. For a given structure, using Komatitsch et al. or Kudela's formulation would potentially require a larger amount of computation than using proposed formulation. As demonstrated in this work (see e.g., **Fig. 15**), the larger the amount of computation that is to be carried out, the more advantageous the use of GPUs would be. Therefore, the observation that Komatitsch et al. and Kudela's formulations can, on certain occasions, achieve higher GPU speedups than the proposed formulation is somewhat expected. However, this does not mean that Komatitsch et al. and Kudela's formulations will always use less computational times, though this cannot be verified at this moment because the GPUs used are different.

Table 2. A comparison of the GPU speedups achieved by different works in time integration.

	D. Komatitsch et al. [36]	P. Kudela [51]	This work

GPU model	NVIDIA® Tesla™ S1070	NVIDIA® Tesla™ C1060	NVIDIA® Tesla™ K20X	NVIDIA® GeForce® 1080 Ti
CPU model	Intel® Xeon® Processor X5570 @ 2.93 GHz	Intel® Xeon® Processor X5660 @ 2.8 GHz		Intel® Xeon® Silver 4122 Processor @ 2.6 GHz
Maximum GPU speedup (GPUs vs. CPU cores = 1:1)	~20	~7	~14	8.5

5. Numerical application

Recently, guided wave tomography methods [52-54] have been used to inspect thickness loss defects in large-scale plate-like structures. The reason why such inspections are possible is that some of the features of a guided wave, such as its amplitude [55] and velocity [56-58], would vary when the guided wave encounters a thickness change in the propagation medium. In this section, the application prospect of the proposed multi-GPU and CUDA-aware MPI-based SE formulation is demonstrated through modelling the interaction between guided waves and thickness loss defects on plates. It is worth mentioning that in real-life scenarios, thickness loss defects are almost always irregularly shaped since service conditions are often highly complex.

A 600 mm × 600 mm × 2 mm aluminum plate was modelled as shown in **Fig. 16**. To embed a 3D randomly shaped thickness loss defect in the model, the following steps would be undertaken.

1) The aluminum plate would be meshed by 125-node GLL elements.
2) Between two circles whose radii are $R1 = 5$ mm and $R2 = 10$ mm, respectively, several element corner nodes would be randomly selected to establish the boundary of the defect in the xy-plane.
3) The z-coordinates of the all the element corner nodes on the top surface of the plate within the boundary would be randomly lowered. The maximum thickness loss of the defect would be set to 0.5 mm.

In order to demonstrate the generality of the proposed GPU-based SE formulation, the numerical simulations that were performed each embody a different thickness loss defect.

To generate an A_0 Lamb wave into the plate, a concentrated force would be applied in the z-direction to the top surface of the plate. On the other hand, to generate a S_0 Lamb wave, a pair of concentrated forces, whose signs are opposite to each other, would be applied at the same location, one to the top surface of the plate and the other to the bottom surface of the plate. The concentrated forces that were utilized in this section take the general form of $F(t) = A_0 \sin(2\pi ft) \times \sin(\pi ft/10)^2$, where $f = 200kHz$ is the central frequency and $A_0 = 10\text{N}$ is the amplitude. All of the simulation results that will be presented below were obtained using converged meshes and a stable time increment of 10 ns.

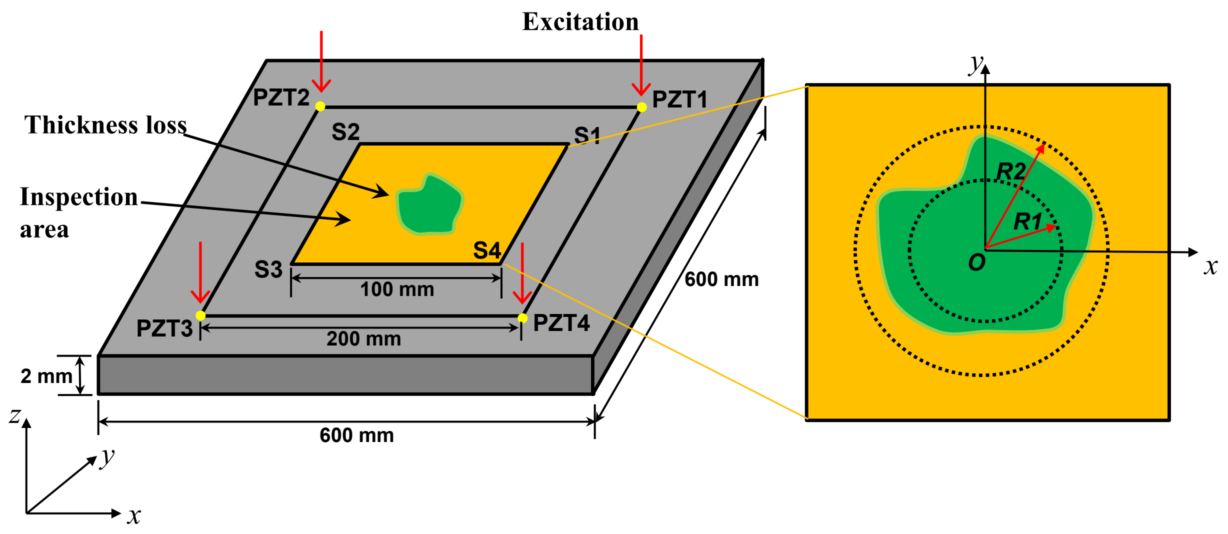
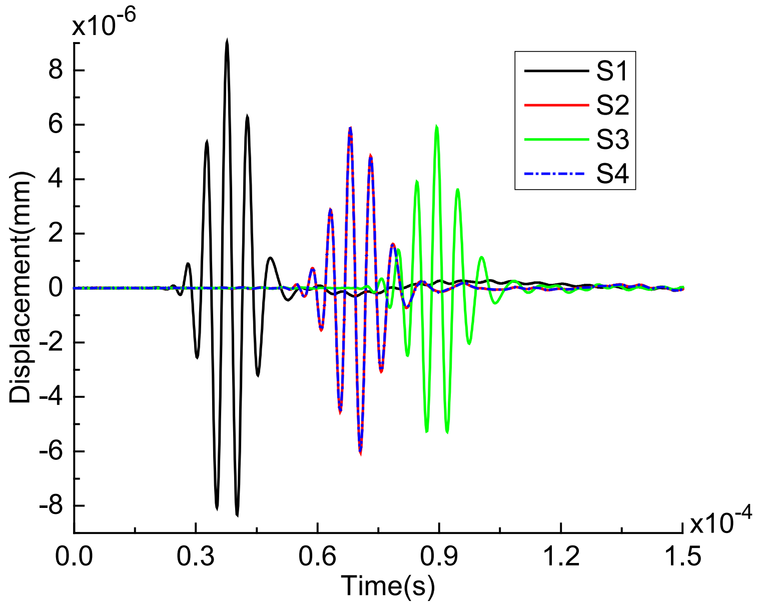


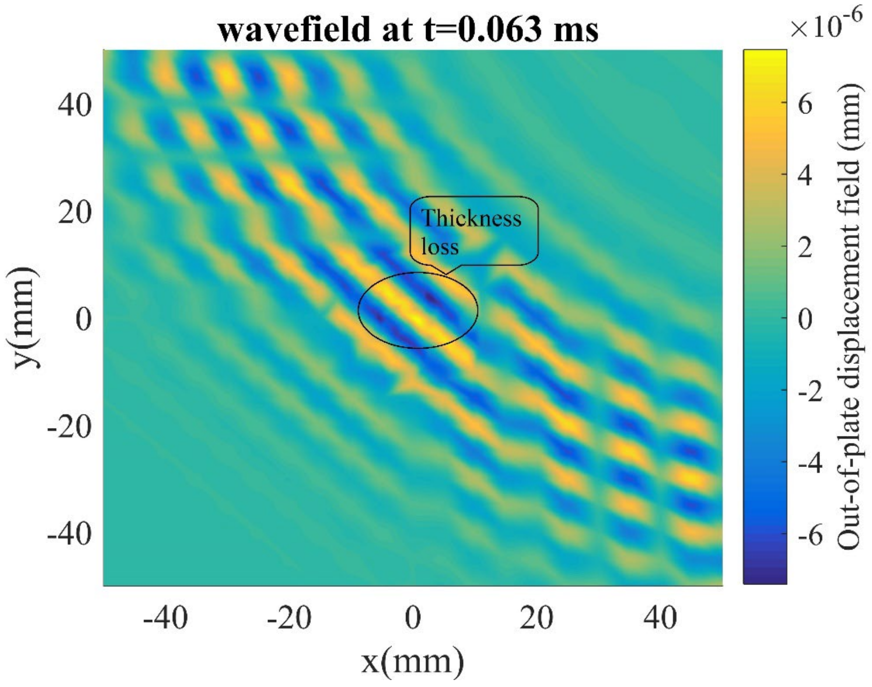
Fig. 16. The schematic diagram of an aluminiun plate (Young's modulus: 70 GPa, Poisson's ratio: 0.33, density of 2700 kg/m^3) with a 3D randomly shaped thickness loss defect.

5.1. A$_0$ Lamb wave mode

A concentrated force was applied at PZT1. **Fig. 17**(a) shows the out-of-plane displacements that were acquired at the four corners of the 100 mm × 100 mm inspection area. Among the four reception points, S1 is the closest to the excitation point. Therefore, the A$_0$ Lamb wave that was acquired at S1 has the largest peak-to-peak amplitude, meaning that it underwent the least attenuation. Moreover, since S2 and S4 are equidistant to the excitation point, the A$_0$ Lamb waves that were acquired at these two reception points are exactly the same. Furthermore, the A$_0$ Lamb wave that was acquired at S3, which is the furthest reception point from the excitation point, underwent the greatest attenuation, as evident through its lowest peak-to-peak amplitude. **Fig. 17**(b) illustrates the out-of-plane displacements of the entire inspection area at an elapsed time of 0.063 ms. It can be seen that the amplitude of the wave exhibited a sudden increase when it passed through the thickness loss defect that was simulated, agreeing with the existing understanding [59].


x10^-6
S1
S2
S3
S4
Displacement(mm)
x10^-4
Time(s)

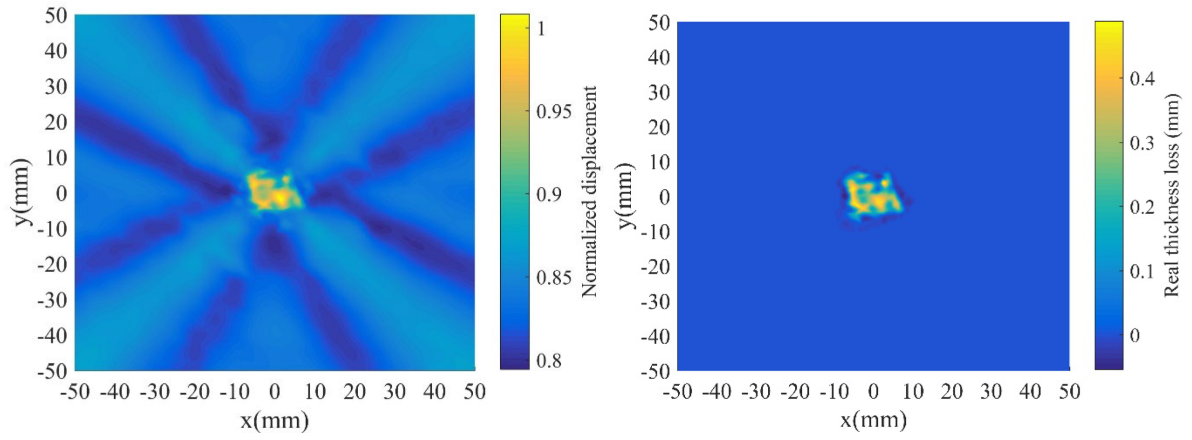
(a)


wavefield at t=0.063 ms
Thickness loss
y(mm)
x(mm)
×10^-6
Out-of-plate displacement field (mm)

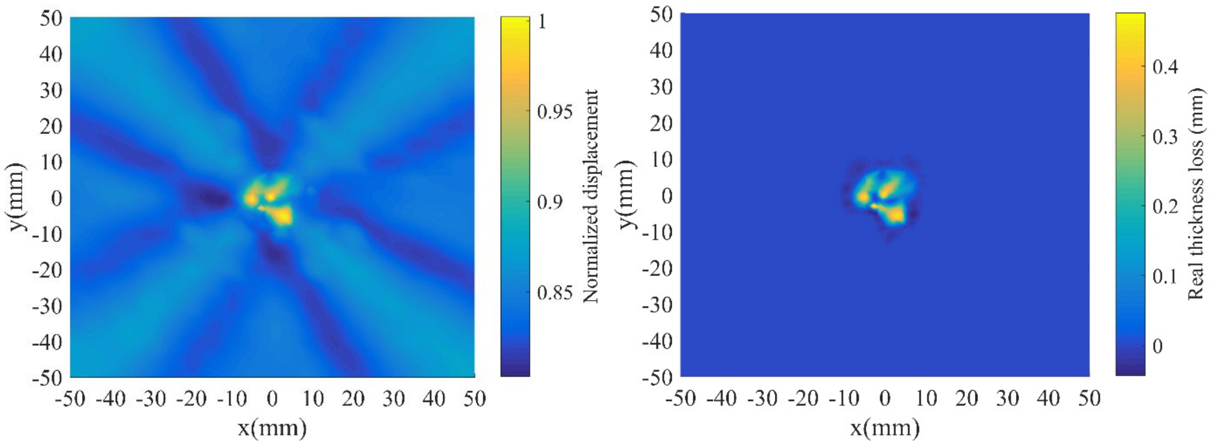
(b)

Fig. 17. (a) The A_0 Lamb waves that were acquired at the four corners of the inspection area when an A_0 Lamb wave mode was generated at the excitation point PZT1. (b) The wavefield of the A_0 Lamb wave mode in the inspection area at t = 0.063 ms.

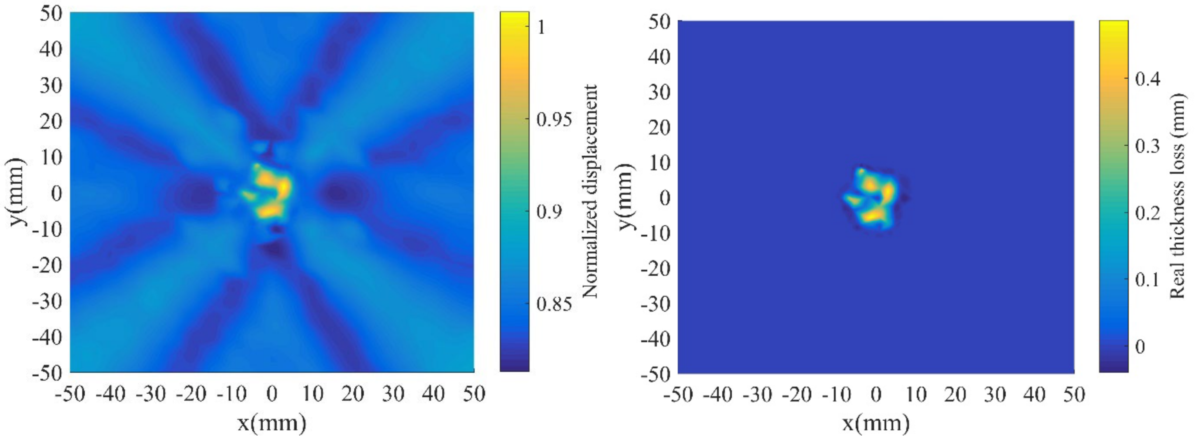
Three different randomly shaped thickness loss defects were further simulated. Inspired by the work of Liu et al. [60], for each case, all the excitation points, i.e., PZT1, PTZ2, PZT3 and PZT4, were simultaneously applied with a concentrated force, and the total displacement at each location of the inspection area is plot in **Fig. 18** in a normalized manner. Here, the total displacement at a given location refers to the maximum amplitude of the Hilbert transform of the out-of-plane displacements that was acquired at that location. From **Fig. 18**, it can be seen that for all three cases, the total displacements at a defect site are always the highest across the respective inspection area. The reconstructed shapes of the thickness loss defects match very well with the actual shapes of the defects, demonstrating the feasibility of the proposed GPU-based SE formulation in assisting the development of guided wave tomography methods.



(a)



(b)

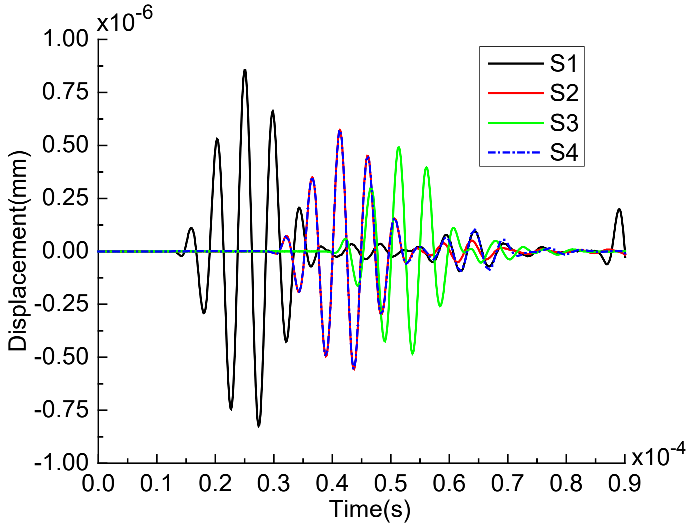


(c)

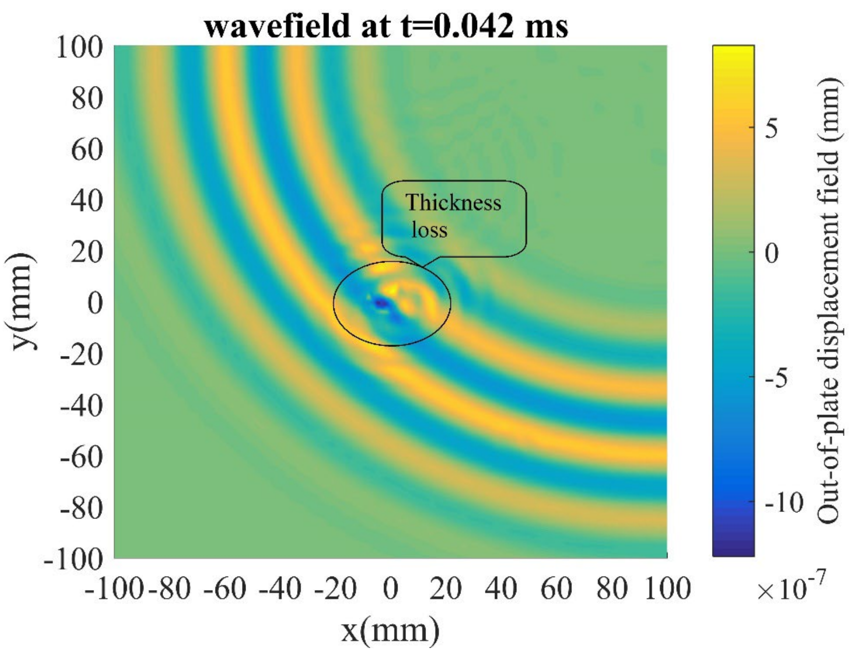
Fig. 18. Comparisons between the reconstructed shapes of three different randomly shaped thickness loss defects (left column), which were obtained based on the amplitudes of A_0 Lamb wave modes, and the actual shapes of the defects (right column).

5.2. S_0 mode

A pair of concentrated forces with opposite signs were applied at PZT1. To visualize S_0 Lamb waves, in-plane displacements were recorded. From **Fig. 19**(a), it is observed that that the attenuation of the S_0 Lamb wave with respect to propagation distance is also significant. Moreover, **Fig. 19**(b) shows that upon passing through the thickness loss defect, the waveform of the S_0 Lamb wave mode, which was generated at PZT1, also underwent an evident change.



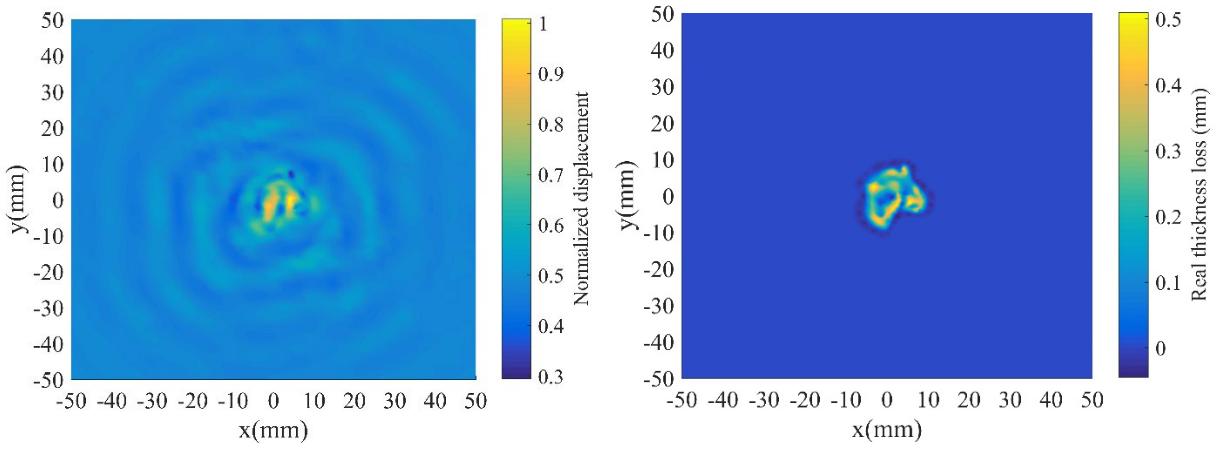
(a)



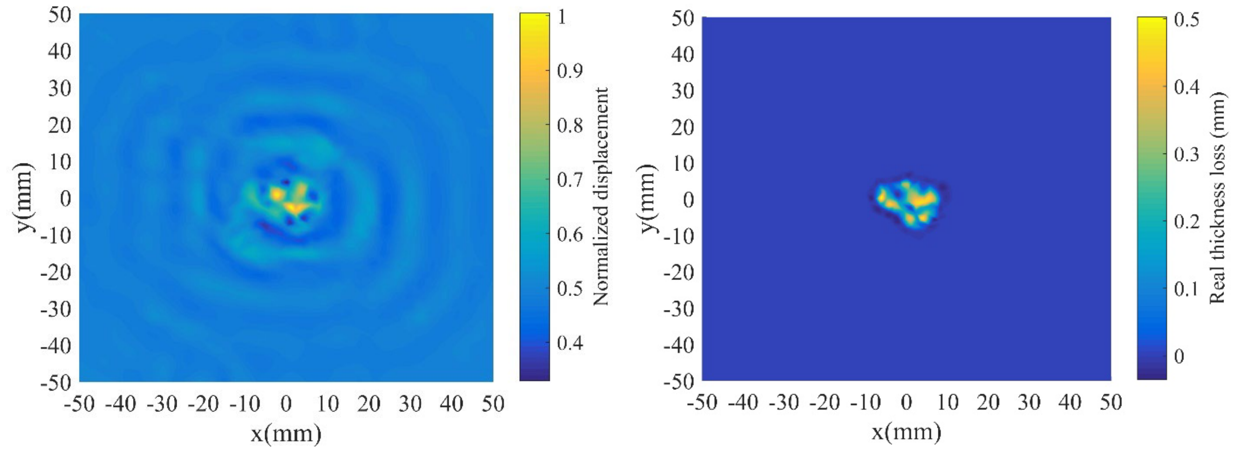
(b)

Fig. 19. (a) The S0 Lamb waves that were acquired at the four corners of the inspection area when an S0 Lamb wave mode was generated at the excitation point PZT1. (b) The wavefield of the S0 Lamb wave mode in the inspection area at t = 0.042 ms.

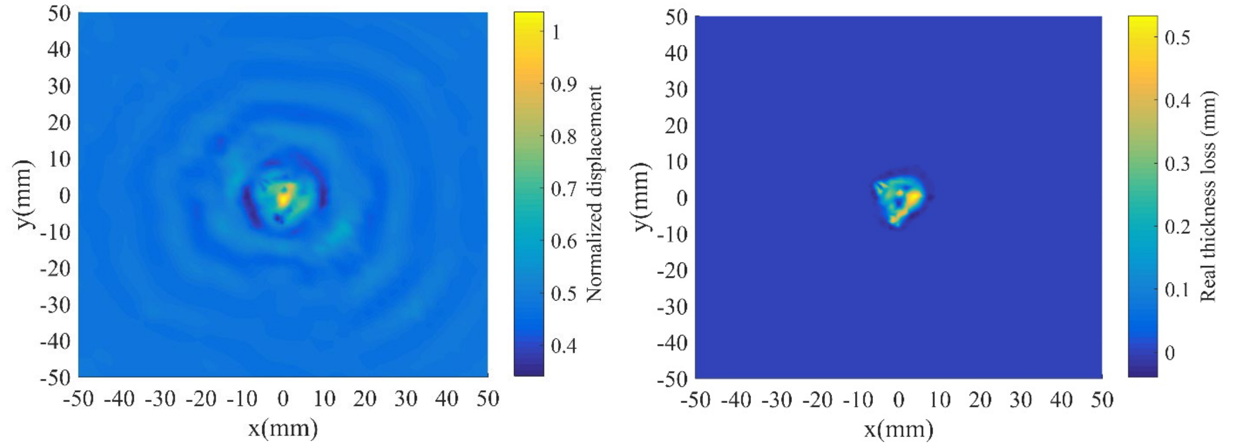
Three different randomly shaped thickness loss defects were simulated. For each case, all four excitation points were simultaneously applied with a pair of concentrated forces with opposite signs. As shown in **Fig. 20**, while changes in total displacements can be observed at the defect sites, the reconstructed shapes of the thickness loss defects are vague, compared with the reconstructions that were obtained based on the A0 Lamb wave mode. This is attributed to the fact that the S0 Lamb wave mode is less sensitive than the A0 Lamb wave mode to thickness losses [59], because the polarization of the S0 Lamb wave mode takes place in the in-plane direction, whereas that of the A0 Lamb wave mode happens in the thickness direction.



(a)



(b)



(c)

Fig. 20. Comparisons between the reconstructed shapes of three different randomly shaped thickness loss defects (left column), which were obtained based on the amplitudes of S_0 Lamb wave modes, and the actual shapes of the defects (right colume).

6. Conclusion

In this paper, we introduce a new multi-GPU and CUDA-aware MPI-based SE formulation for simulating ultrasonic wave propagation in 3D solids. For a given structure, the proposed formulation would decompose it into several subdomains and assign each subdomain to a single GPU to process. To avoid GPU memory access conflicts during the global matrix assembly process, the elements in each subdomain would be divided into groups by the greedy coloring algorithm, such that the elements in the same group would not share any common node. For each subdomain, the assembly process of its global stiffness matrix and global mass matrix would be executed by assigning one block to calculate the stiffness matrix and mass matrix of one element, and one thread to calculate the matrix entries of one node. The transient response of the structure would be solved by explicit time integration. At each time integration step, the internal forces of the common nodes of the different subdomains would be exchanged among the different GPUs used, using either of the two message exchange strategies that are introduced. Time integration and message exchange would both be executed by assigning one thread to deal with the all the data that is associated with one node.

The accuracy of the proposed GPU-based SE formulation has been well validated by a multi-CPU core-based counterpart that relies on the classical MPI for message exchange. As demonstrated by a series of numerical experiments, the time costs of the proposed GPU-based SE formulation are significantly lower than those of the benchmark CPU-based counterpart. For the largest model that could be accommodated by the GPU workstation used, the proposed GPU-based SE formulation sped up global matrix assembly, time integration and message exchange by up to 20 times, 10 times, and 220 times (Point-to-Point) or 170 times (All-Reduce), respectively. What's more, the GPU speedups for all three steps actually increase with the number of DOFs, meaning that the superiority of the new formulation would be more pronounced for larger models. Last but not least, the new formulation was applied to modelling the propagation of Lamb waves through 3D randomly shaped thickness loss defects in plates, exhibiting itself as a highly competitive alternative tool for studying wave propagation problems which are of significant importance to the development of ultrasonic inspection techniques.

Acknowledgment

This work is supported by the Research Grants Council of Hong Kong (grant no. 25211319). The authors would like to thank Dr Peter Huthwaite and Dr Mazdak Ghajari of Imperial College London for their constructive suggestions on this work.

Appendix

Kernel functions used are appended below:

Kernel 1 Point-to-Point strategy – Extract the internal forces of the common nodes of one communication path.

```
g_idx = threadIdx.x + blockDim.x*blockIdx.x;
if g_idx < the number of common nodes in the model
do
    On the native MPI rank, if the current common node is shared by the target MPI rank
    do
        m = the local index of the current common node on the native MPI;
        n = the index of the current common node in Send_buffer;
        (Note: both m and n can be extracted by the current thread g_idx independently)
        Send_buffer[3*n] = F_int[3*m];
        Send_buffer[3*n + 1] = F_int[3*m+ 1];
        Send_buffer[3*n + 2] = F_int[3*m + 2];
Return Send_buffer;
```

Kernel 2 Point-to-Point strategy – Add the data in **Recv_buffer** to **Arr_con**.

```
g_idx = threadIdx.x + blockDim.x*blockIdx.x;
if g_idx < the number of common nodes in the model
    do
        On the target MPI rank, if the current common node is shared by the native MPI rank
            do
                n = the index of the current common node in Recv_buffer;
                (Note: n can be extracted by the current thread g_idx independently.)
                Arr_con[3*g_idx] += Recv_buffer[3*n ];
                Arr_con[3*g_idx + 1] += Recv_buffer[3*n  +  1];
                Arr_con[3*g_idx + 2] += Recv_buffer[3*n  +  2];
Return Arr_con;
```

1

Kernel 3 Point-to-Point strategy – Add **Arr_con** to the internal force array of one MPI rank.

```
g_idx = threadIdx.x + blockDim.x*blockIdx.x;
if g_idx < the number of common nodes in the model
    do
        if the current common node is hosted by the native MPI rank
            do
                m = the local index of the current common node on the native MPI;
                (Note: m can be extracted by the current thread g_idx independently.)
                F_int[3*m] += Arr_con[3*g_idx - 1];
                F_int[3*m + 1] += Arr_con[3*g_idx + 1];
                F_int[3*m + 2] += Arr_con[3*g_idx + 2];
Return F_int;
```

2

Kernel 4 All-Reduce strategy – Extract the internal forces of all common nodes.

```
g_idx = threadIdx.x + blockDim.x*blockIdx.x;
if g_idx < the number of common nodes in the model
    do
        if the current common node is hosted by the current MPI rank
            do
                m = the local index of the current common node on the current MPI;
                (Note: m can be extracted by the current thread g_idx independently)
                Arr_con1[3* g_idx] = F_int[3*m];
                Arr_con1[3* g_idx + 1] = F_int[3*m + 1];
                Arr_con1[3* g_idx + 2] = F_int[3*m + 2];
Return Arr_con1;
```

1

Kernel 5 All-Reduce strategy – Update the internal force array of one MPI rank.

```
g_idx = threadIdx.x + blockDim.x*blockIdx.x;
if g_idx < the number of common nodes in the model
    do
        if the current common node is hosted by the current MPI rank
            do
                m = the local index of the current common node on the current MPI;
                (Note: m can be extracted by the current thread g_idx independently)
                F_int[3*m ] = Arr_con2[3* g_idx];
                F_int[3*m + 1] = Arr_con2[3* g_idx + 1];
                F_int[3*m + 2] = Arr_con2[3* g_idx + 2];
Return F_int;
```

2

Kernel 6 Time integration – Calculate the internal force array of one subdomain in terms of stiffness matrix **K** and **u1**.

```
g_idx = threadIdx.x + blockDim.x*blockIdx.x;
if g_idx < the number of nodes in the subdomain
do
    for node i < the total number of nodes in all adjacent elements
    do
        m = the index of node i in K in Compressed Row Storage;
        n = the local index of node i in the subdomain;
        (Note: m and n can be extracted by the current thread g_idx independently)
        F_int[3*g_idx )]  += K[m]*u1[3*n ];
        F_int[3*g_idx + 1]  += K[m +1]* u1[3*n + 1];
        F_int[3*g_idx + 2]  += K[m + 2]* u1[3*n + 2];
Return F_int;
```

Kernel 7 Time integration – Update the displacement vector of one subdomain.

```
g_idx = threadIdx.x + blockDim.x*blockIdx.x;
if g_idx < the number of nodes in the subdomain
do
    u2[3*g_idx ]  =(f_ext[3*g_idx ] - f_int[3*g_idx ])/M[3*g_idx] + 2* u1[3*g_idx ] –
            u0[3*g_idx] ;
    u2[3*g_idx + 1]  =(f_ext[3*g_idx + 1] - f_int[3*g_idx + 1])/M[3*g_idx + 1] + 2*
            u1[3*g_idx + 1] – u0[3*g_idx + 1] ;
    u2[3*g_idx + 2]  =(f_ext[3*g_idx + 2] - f_int[3*g_idx + 2])/M[3*g_idx + 2] + 2*
            u1[3*g_idx + 2] – u0[3*g_idx + 2] ;

    u0[3*g_idx] = u1[3*g_idx]
    u0[3*g_idx + 1] = u1[3*g_idx + 1]
    u0[3*g_idx + 2] = u1[3*g_idx + 2]
    u1[3*g_idx] = u2[3*g_idx]
    u1[3*g_idx + 1] = u2[3*g_idx + 1]
    u1[3*g_idx + 2] = u2[3*g_idx + 2]
Return;
```

References

1. Bathe, K.J.J.W.e.o.c.s. and engineering, *Finite element method.* 2007: p. 1-12.
2. Hughes, T.J., *The finite element method: linear static and dynamic finite element analysis*. 2012: Courier Corporation.
3. Li, F., Y. Zhao, P. Cao, and N. Hu, *Mixing of ultrasonic Lamb waves in thin plates with quadratic nonlinearity.* Ultrasonics, 2018. **87**: p. 33-43.
4. Sármány, D., M.A. Botchev, and J.J. van der Vegt, *Dispersion and dissipation error in high-order Runge-Kutta discontinuous Galerkin discretisations of the Maxwell equations.* Journal of Scientific Computing, 2007. **33**(1): p. 47-74.
5. Harari, I., *A survey of finite element methods for time-harmonic acoustics.* Computer Methods in Applied Mechanics and Engineering, 2006. **195**(13-16): p. 1594-1607.
6. Patera, A.T., *A Spectral Element Method for Fluid-Dynamics - Laminar-Flow in a Channel Expansion.* Journal of Computational Physics, 1984. **54**(3): p. 468-488.
7. Komatitsch, D. and J. Tromp, *Introduction to the spectral element method for three-dimensional seismic wave propagation.* Geophysical Journal International, 1999. **139**(3): p. 806-822.
8. Li, F.L. and F.X. Zou, *A hybrid spectral/finite element method for accurate and efficient modelling of crack-induced contact acoustic nonlinearity.* Journal of Sound and Vibration, 2021. **508**: p. 116198.
9. de Frutos, J. and J. Novo, *A Spectral Element Method for the Navier--Stokes Equations with Improved Accuracy.* SIAM Journal on Numerical Analysis, 2000. **38**(3): p. 799-819.
10. Komatitsch, D., J.P. Vilotte, R. Vai, J.M. Castillo-Covarrubias, and F.J. Sanchez-Sesma, *The spectral element method for elastic wave equations - Application to 2-D and 3-D seismic problems.* International Journal for Numerical Methods in Engineering, 1999. **45**(9): p. 1139-1164.
11. Dong, S. and Z. Yosibash, *A parallel spectral element method for dynamic three-dimensional nonlinear elasticity problems.* Computers & Structures, 2009. **87**(1-2): p. 59-72.
12. Xu, B., L.L. Luan, H.B. Chen, and H.D. Wang, *Local wave propagation analysis in concrete-filled steel tube with spectral element method using absorbing layers - Part I: Approach and validation.* Mechanical Systems and Signal Processing, 2020. **140**.
13. Li, W.N., J.H. Pan, and Y.X. Ren, *The discontinuous Galerkin spectral element methods for compressible flows on two-dimensional mixed grids.* Journal of Computational Physics, 2018. **364**: p. 314-346.
14. Biswas, P., P. Dutt, S. Ghorai, and N.K. Kumar, *Space-Time Coupled Least-Squares Spectral Element Methods for Parabolic Problems.* International Journal for Computational Methods in Engineering Science & Mechanics, 2019. **20**(5): p. 358-371.
15. Law, K.H., *A Parallel Finite-Element Solution Method.* Computers & Structures, 1986. **23**(6): p. 845-858.
16. King, R.B. and V. Sonnad, *Implementation of an Element-by-Element Solution Algorithm for the Finite-Element Method on a Coarse-Grained Parallel Computer.* Computer Methods in Applied Mechanics and Engineering, 1987. **65**(1): p. 47-59.

17. Yagawa, G., N. Soneda, and S. Yoshimura, *A Large-Scale Finite-Element Analysis Using Domain Decomposition Method on a Parallel Computer.* Computers & Structures, 1991. **38**(5-6): p. 615-625.

18. Oh, S.E. and J.W. Hong, *Parallelization of a finite element Fortran code using OpenMP library.* Advances in Engineering Software, 2017. **104**: p. 28-37.

19. Kwan, Y.Y. and J. Shen, *An efficient direct parallel spectral-element solver for separable elliptic problems.* Journal of Computational Physics, 2007. **225**(2): p. 1721-1735.

20. Liu, S.L., D.H. Yang, X.P. Dong, Q.C. Liu, and Y.C. Zheng, *Element-by-element parallel spectral-element methods for 3-D teleseismic wave modeling.* Solid Earth, 2017. **8**(5): p. 969-986.

21. Gropp, W., W.D. Gropp, E. Lusk, A. Skjellum, and A.D.F.E.E. Lusk, *Using MPI: portable parallel programming with the message-passing interface*. Vol. 1. 1999: MIT press.

22. Brandvik, T. and G. Pullan, *Acceleration of a two-dimensional Euler flow solver using commodity graphics hardware.* Proceedings of the Institution of Mechanical Engineers Part C-Journal of Mechanical Engineering Science, 2007. **221**(12): p. 1745-1748.

23. Anderson, J.A., C.D. Lorenz, and A. Travesset, *General purpose molecular dynamics simulations fully implemented on graphics processing units.* Journal of Computational Physics, 2008. **227**(10): p. 5342-5359.

24. Dziekonski, A., P. Sypek, A. Lamecki, and M. Mrozowski, *Finite Element Matrix Generation on a Gpu.* Progress in Electromagnetics Research-Pier, 2012. **128**: p. 249-265.

25. Cai, Y., G.Y. Li, H. Wang, G. Zheng, and S. Lin, *Development of parallel explicit finite element sheet forming simulation system based on GPU architecture.* Advances in Engineering Software, 2012. **45**(1): p. 370-379.

26. Fu, Z., T.J. Lewis, R.M. Kirby, and R.T. Whitaker, *Architecting the Finite Element Method Pipeline for the GPU.* J Comput Appl Math, 2014. **257**: p. 195-211.

27. Strbac, V., J.V. Sloten, and N. Famaey, *Analyzing the potential of GPGPUs for real-time explicit finite element analysis of soft tissue deformation using CUDA.* Finite Elements in Analysis and Design, 2015. **105**: p. 79-89.

28. Kudela, P., J. Moll, and P. Fiborek, *Parallel spectral element method for guided wave based structural health monitoring.* Smart Materials and Structures, 2020. **29**(9).

29. *Nvidia unleashes Cuda technology.* Computer Graphics World, 2006. **29**(12): p. 4-4.

30. Wilt, N., *The CUDA handbook : a comprehensive guide to GPU programming*. 2013, Upper Saddle River, NJ: Addison-Wesley. xxv, 494 pages.

31. Huthwaite, P., *Accelerated finite element elastodynamic simulations using the GPU.* Journal of Computational Physics, 2014. **257**: p. 687-707.

32. Stavroulakis, G., D.G. Giovanis, V. Papadopoulos, and M. Papadrakakis, *A GPU domain decomposition solution for spectral stochastic finite element method.* Computer Methods in Applied Mechanics and Engineering, 2017. **327**: p. 392-410.

33. Cao, X.G., Y. Cai, and X.Y. Cui, *A parallel numerical acoustic simulation on a GPU using an edge-based smoothed finite element method.* Advances in Engineering Software, 2020. **148**: p. 102835.

34. Kudela, P., *Parallel implementation of spectral element method for Lamb wave propagation modeling.* International Journal for Numerical Methods in Engineering, 2016. **106**(6): p. 413-429.

35. Remacle, J.F., R. Gandham, and T. Warburton, *GPU accelerated spectral finite elements on all-hex meshes.* Journal of Computational Physics, 2016. **324**: p. 246-257.

36. Komatitsch, D., G. Erlebacher, D. Goddeke, and D. Michea, *High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster.* Journal of Computational Physics, 2010. **229**(20): p. 7692-7714.

37. Kraus, J., *An introduction to CUDA-aware MPI.* NVIDIA Developer Blog, 2013.

38. Corporation, D.S.S. *Abaqus 6.14 Documentation*. 2014; Available from: http://130.149.89.49:2080/v6.14/.

39. Dahlquist, G.G., *A special stability problem for linear multistep methods.* BIT Numerical Mathematics, 1963. **3**(1): p. 27-43.

40. Noh, G. and K.J. Bathe, *An explicit time integration scheme for the analysis of wave propagations.* Computers & Structures, 2013. **129**: p. 178-193.

41. Soares Jr, D., *A novel family of explicit time marching techniques for structural dynamics and wave propagation models.* Computer Methods in Applied Mechanics and Engineering, 2016. **311**: p. 838-855.

42. Gravouil, A., T. Elguedj, and H. Maigre, *An explicit dynamics extended finite element method. Part 2: Element-by-element stable-explicit/explicit dynamic scheme.* Computer Methods in Applied Mechanics and Engineering, 2009. **198**(30-32): p. 2318-2328.

43. Smith, I.M., D.V. Griffiths, and L. Margetts, *Programming the finite element method*. 2013: John Wiley & Sons.

44. Karypis, G. and V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs.* Siam Journal on Scientific Computing, 1998. **20**(1): p. 359-392.

45. Mandel, J., *Two ‐level domain decomposition preconditioning for the p ‐version finite element method in three dimensions.* International journal for numerical methods in engineering, 1990. **29**(5): p. 1095-1108.

46. Yagawa, G. and R. Shioya, *Parallel finite elements on a massively parallel computer with domain decomposition.* Computing Systems in Engineering, 1993. **4**(4-6): p. 495-503.

47. Klöckner, A., T. Warburton, J. Bridge, and J.S. Hesthaven, *Nodal discontinuous Galerkin methods on graphics processors.* Journal of Computational Physics, 2009. **228**(21): p. 7863-7882.

48. Cecka, C., A.J. Lew, and E. Darve, *Assembly of finite element methods on graphics processors.* International Journal for Numerical Methods in Engineering, 2011. **85**(5): p. 640-669.

49. Mitchem, J., *On Various Algorithms for Estimating the Chromatic Number of a Graph.* The Computer Journal, 1976. **19**(2): p. 182-183.

50. Dokainish, M.A. and K. Subbaraj, *A Survey of Direct Time-Integration Methods in Computational Structural Dynamics .1. Explicit Methods.* Computers & Structures, 1989. **32**(6): p. 1371-1386.

51. Kudela, P.J.I.J.f.N.M.i.E., *Parallel implementation of spectral element method for Lamb wave propagation modeling.* 2016. **106**(6): p. 413-429.

52. Leonard, K.R., E.V. Malyarenko, and M.K. Hinders, *Ultrasonic Lamb wave tomography.* Inverse Problems, 2002. **18**(6): p. 1795-1808.

53. Huthwaite, P. and F. Simonetti, *High-resolution guided wave tomography.* Wave Motion, 2013. **50**(5): p. 979-993.

54. Rao, J., M. Ratassepp, and Z. Fan, *Investigation of the reconstruction accuracy of guided wave tomography using full waveform inversion.* Journal of Sound and Vibration, 2017. **400**: p. 317-328.

55. Huthwaite, P., *Guided wave tomography with an improved scattering model.* Proc Math Phys Eng Sci, 2016. **472**(2195): p. 20160643.

56. Ho, K.S., D.R. Billson, and D.A. Hutchins, *Ultrasonic Lamb wave tomography using scanned EMATs and wavelet processing.* Nondestructive Testing and Evaluation, 2007. **22**(1): p. 19-34.

57. Nagy, P.B., F. Simonetti, and G. Instanes, *Corrosion and erosion monitoring in plates and pipes using constant group velocity Lamb wave inspection.* Ultrasonics, 2014. **54**(7): p. 1832-1841.

58. Zimmermann, A.A.E., P. Huthwaite, and B. Pavlakovic, *High-resolution thickness maps of corrosion using SH1 guided wave tomography.* Proceedings of the Royal Society a-Mathematical Physical and Engineering Sciences, 2021. **477**(2245).

59. Rao, J., M. Ratassepp, and Z. Fan, *Quantification of thickness loss in a liquid-loaded plate using ultrasonic guided wave tomography.* Smart Materials and Structures, 2017. **26**(12): p. 125017.

60. Liu, Y., N. Hu, H. Xu, and W. Yuan, *Damage evaluation based on a wave energy flow map using multiple PZT sensors.* Sensors (Basel), 2014. **14**(2): p. 1902-17.