Advances in Aerodynamics

RESEARCH Open Access

A multi-colored Gauss-Seidel solver for aerodynamic simulations of a transport aircraft model on graphics processing units

Liu Yang[1*] and Jian Yang[2]

[1] Department of Research, MetaX Integrated Circuits (Shanghai) Co., Ltd., Shanghai 201210, China
[2] CTO Office, MetaX Integrated Circuits (Shanghai) Co., Ltd., Shanghai 201210, China

Abstract

For practical large-scale applications of computational fluid dynamics in the aerospace industry, implicit flow solvers are necessitated for efficient simulations. This paper presents the implementation of a solver that employs an unstructured finite volume approach and a Multi-Colored Gauss-Seidel (MCGS) method for steady-state compressible flow simulations on a server equipped with multiple Graphics Processing Units (GPUs). The mesh partition process is completed with PyMetis, and Message Passing Interface (MPI) is utilized for communications between mesh partitions. A parallel coloring algorithm is employed in the pre-processing module. The code is developed using a hybrid programming approach, with the main framework written in Python and the GPU kernel source codes written in C. The transonic turbulent flows over the CHN-T1 transport aircraft model are simulated on unstructured hybrid meshes. The numerical results are compared with experimental data, and the performance of the developed flow simulation framework is analysed.

Keywords: CFD, GPU, Finite volume method, Multi-colored Gauss-Seidel, High-performance computing

1 Introduction

Evaluations of aerodynamic performance are essential tools in the process of aircraft design. Nowadays, predictions that utilize Computational Fluid Dynamics (CFD) simulations are preferred because the required workloads and costs are often lower than conducting experiments, and traditional wind tunnel testing has become a supporting role. Aerospace engineers have been used to waiting hours to days for complex simulation tasks to be completed on high-performance clusters with a massive number of Central Processing Units (CPUs). Fortunately, owing to continuous advancements in compute capability of Graphics Processing Units (GPUs) and corresponding efficient algorithms in recent years, aerodynamic calculations can be accomplished in a shorter time with lower hardware costs and energy consumption [1–3]. Flow solvers that employ explicit time integration, such as Runge-Kutta methods, can be ported to GPUs straightforwardly. Nevertheless, explicit solvers typically suffer from low convergence rates when confronted with large-scale simulations on fine meshes. Consequently, the necessity arises for the development of implicit flow solvers on GPUs for the purpose of implementing practical large-scale industrial applications.

The Gauss-Seidel method has been one of the classical iterative methods for solving systems of linear equations. However, the underlying algorithm is inherently sequential, which prevents efficient computation on massively parallel machines. To address this limitation, researchers have proposed algorithms to resolve data race conditions and parallel the Gauss-Seidel methods on CPUs, which are known as Multi-Colored Gauss-Seidel (MCGS) methods [4]. The extensive experience gained in the implementation of multi-colored Gauss-Seidel methods on CPUs has led to their adoption as the preferred implicit scheme for GPU-accelerated computational fluid dynamics simulations. A multi-colored Gauss-Seidel GPU solver with mixed-precision algorithms has been developed for the unstructured finite volume code FUN3D, which is able to reduce memory traffic while maintaining double-precision accuracy [5]. Nguyen et al. [6] accelerated Bombardier's in-house CFD code FANSC on GPUs with a red-black Gauss-Seidel solver, which was a two-color version of the MCGS method that was often employed for structured meshes. They reported that the GPU-accelerated code was significantly faster than the original CPU code with Lower-Upper Symmetric-Gauss-Seidel (LU-SGS) algorithms. Zhang et al. [7] ported the LU-SGS solver of a production-level CFD software NNW-FlowStar to GPUs with multiple coloring strategies, achieving a speedup of more than 30 times compared with pure CPU computing. Watkins et al. [8] implemented a multi-colored Gauss-Seidel method for the direct flux reconstruction method to converge the steady-state Euler equations on two nodes of a GPU cluster, and the performance analysis showed that the MCGS scheme was able to maintain near-perfect weak scaling and achieve better scalability than the explicit Runge-Kutta scheme. These successful examples have been a demonstration of the promising potential of multi-colored Gauss-Seidel methods on GPUs.

In order to perform simulations in multi-GPU environments, the Message Passing Interface (MPI) is frequently utilized for the communication between GPUs. Typically, each MPI process is responsible for controlling a single GPU. There are two frequently adopted approaches for data exchange between GPUs. The first approach is more conventional and involves data transfers between the GPU and the CPU, with data exchange occurring on the CPU side. The second approach is to exchange data directly between two GPUs, although only some advanced series of GPUs support this feature. It is anticipated to produce better performance compared to the first approach. Nevertheless, several authors [9–11] have indicated that the second approach may not always be more efficient, as it is contingent upon the configuration of the hardware and necessitates greater attention to the implementation of the software. Accordingly, for the purposes of this paper, which represents an initial step towards the parallelisation of GPU-based computing on a large scale, the first approach is given priority consideration.

A prerequisite for the multi-colored Gauss-Seidel solver is that each element in the mesh must be painted with a color that is distinct from the colors of its neighbors. For problems of a relatively modest scale on multiple GPUs, it is possible to execute a serial coloring of the entire mesh on a single process, after which the colors are distributed to all mesh partitions. However, for large-scale jobs, this may result in excessive processing times. It is therefore more efficient and feasible to adopt the parallel graph coloring algorithms proposed by Bogle et al. [12]. The concept of parallel graph coloring algorithms is that each MPI process independently colors its mesh partition in parallel, subsequently identifying and resolving conflicts on the boundaries through iterative processes until no conflicts remain.

In this paper, a multi-colored Gauss-Seidel solver is developed and implemented for multi-GPU environments as the time-stepping scheme of an unstructured finite volume based flow simulation framework. It is an extension based on our previous single-GPU solver [13]. The remainder of the paper is organized as follows. An overview of the parallel flow simulation framework is given in Section 2. The numerical methods and algorithms are introduced in Section 3. Section 4 presents the results of flow simulations of a benchmark transport aircraft model on a multi-GPU server. Finally, the work is concluded in Section 5.

2 Overview of the flow simulation framework

The main framework of this work was developed using Python, because it excels at file operations and simple data processing, which makes the pre- and post-processing modules compact and readable. For the simulation module that is responsible for solving the flow and turbulence equations, GPU kernel source codes were programmed in order to make use of the computing power of the GPU. A distinctive feature of the framework was that the GPU kernel source codes were not hard-coded, but rather generated through the use of a Python-based template engine Mako [14]. The rationale behind this approach was to enable the framework to run on different GPU platforms by maintaining a single set of template files, thereby reducing the efforts associated with code development and maintenance. The generated GPU kernel source codes can be compiled and launched by Python wrappers for the GPU's compiler, driver, and runtime Application Programming Interfaces (APIs). The flowchart of the framework is depicted in Fig. 1. As illustrated, the modules in the purple boxes on the left are processed on the CPUs, while the modules in the green boxes on the right are performed on the GPUs.

The workflow starts by reading the input files. The framework employs the Python package Meshio [15] to read in unstructured meshes. Meshio is capable of handling unstructured meshes in a variety of formats with a few lines of code, thereby reducing the time and effort required for development. If the mesh has not been partitioned, the mesh partition module will partition it. Otherwise, the partitioning process will be skipped. The mesh partition module was developed by using the Python package PyMetis, a Python wrapper for the Metis graph partitioning software [16]. Subsequently, each MPI process reads the mesh of its own partition and creates mesh connectivity information. Thereafter, the framework employs Mako to generate GPU kernel source codes based on the configurations and parameters of the input files. The generated GPU kernel source codes are then compiled for later use in the flow simulation. The framework proceeds to compute the face normals and the volumes of the mesh elements.

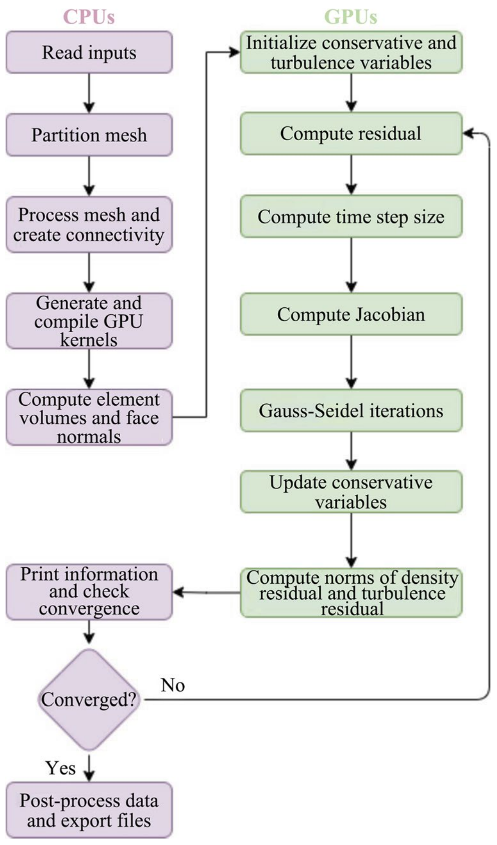
CPUs
GPUs
Read inputs
Partition mesh
Process mesh and create connectivity
Generate and compile GPU kernels
Compute element volumes and face normals
Initialize conservative and turbulence variables
Compute residual
Compute time step size
Compute Jacobian
Gauss-Seidel iterations
Update conservative variables
Compute norms of density residual and turbulence residual
Print information and check convergence
Converged?
No
Yes
Post-process data and export files

Fig. 1 Flowchart of the framework

Once the aforementioned preparation work has been completed, the geometrical quantities and connectivity data of mesh elements are transferred to the GPU. The computation work on the GPU side commences with the initialization of the flow and turbulence fields. This is then followed by the iterative solution of the flow and turbulence equations. The norms of density and turbulence residuals are computed and subsequently retrieved back to the CPU for the purpose of convergence checking. The force and moment coefficients are calculated on the CPU after the flow data is copied from the GPU to the CPU. Once the criterion for convergence has been achieved, the flow field file is exported in VTK format using Meshio for later use in analysis or visualization.

3 Methodology

3.1 Governing equations

The flow simulation framework in this work is developed targeting compressible steady-state turbulent flows around aircraft. Therefore, for the governing equations, consider the compressible Navier-Stokes equations using Einstein notation,

$$\frac{\partial \mathbf{W}}{\partial t} + \frac{\partial \mathbf{F}_i}{\partial x_i} - \frac{\partial \mathbf{F_v}_i}{\partial x_i} = 0, \tag{1}$$

where $\mathbf{W}$ is the conservative variables, and $\mathbf{F}_i$ and $\mathbf{F_v}_i$ are the inviscid and the viscous fluxes respectively. The non-dimensionalized compressible Reynolds-Averaged Navier-Stokes (RANS) equations are solved with an unstructured second-order cell-centered finite volume method. For turbulent simulations, Sutherland's Law and the Spalart-Allmaras (SA) turbulence model [17] are implemented. The Spalart-Allmaras turbulence model solves a single equation

$$\frac{D\tilde{\nu}}{Dt} = S_P - S_D + D, \tag{2}$$

where S_P, S_D and D are production, destruction and diffusion terms respectively.

3.2 Spatial discretization

To achieve second-order accuracy, the primitive variables $\mathbf{U}$ are converted from the conservative variables $\mathbf{W}$, and interpolated at the interface of element i and element j with the UMUSCL scheme [18], given by

$$\mathbf{U}_{ij}^{L} = \mathbf{U}_i + \phi_i \left[\frac{\chi}{2}(\mathbf{U}_j - \mathbf{U}_i) + (1-\chi)\nabla\mathbf{U}\cdot\mathbf{r}_{if} \right], \tag{3}$$

$$\mathbf{U}_{ij}^{R} = \mathbf{U}_j + \phi_j \left[\frac{\chi}{2}(\mathbf{U}_i - \mathbf{U}_j) + (1-\chi)\nabla\mathbf{U}\cdot\mathbf{r}_{jf} \right], \tag{4}$$

where ϕ is the Venkatakrishnan limiter [19] and $\mathbf{r}$ is the vector from the element center to the face center. χ is a parameter ranging between -1 to 1 to control the numerical property of the scheme. When the Venkatakrishnan limiter is employed in steady-state problems, to prevent the possible convergence stalling caused by the limiter reacting to machine-level noise, the limiter is often frozen after a certain number of iterations [18, 19]. The inviscid flux at the interface is computed with the Roe scheme [20] with an entropy fix [21].

3.3 Multi-colored Gauss-Seidel method

A first-order backward differentiation formula is applied to find the solution of element i at time step $m+1$,

$$\frac{\Delta\mathbf{W}_i}{\Delta t} + \mathbf{R}\left(\mathbf{W}_i^{m+1}, \mathbf{W}_{nbs}^{m+1}\right) = 0, \tag{5}$$

where $\Delta\mathbf{W}_i = \mathbf{W}_i^{m+1} - \mathbf{W}_i^{m}$ and nbs represents the nearest-neighbours of element i. The residual $\mathbf{R}$ at time step $m+1$ can be linearized as

$$\begin{aligned}\mathbf{R}\left(\mathbf{W}_i^{m+1}, \mathbf{W}_{nbs}^{m+1}\right) &= \mathbf{R}\left(\mathbf{W}_i^{m}, \mathbf{W}_{nbs}^{m}\right) \\ &\quad + \frac{\partial\mathbf{R}_i^m}{\partial\mathbf{W}_i}\Delta\mathbf{W}_i + \sum_{nb=1}^{nbs}\frac{\partial\mathbf{R}_i^m}{\partial\mathbf{W}_{nb}}\Delta\mathbf{W}_{nb}.\end{aligned} \tag{6}$$

Substitute Eq. (6) into Eq. (5), and rearrange the equation to yield

$$\left(\frac{\mathbf{I}}{\Delta t}+\frac{\partial \mathbf{R}_i^m}{\partial \mathbf{W}_i}\right)\Delta \mathbf{W}_i+\sum_{nb=1}^{nbs}\frac{\partial \mathbf{R}_i^m}{\partial \mathbf{W}_{nb}}\Delta \mathbf{W}_{nb}=-\mathbf{R}\left(\mathbf{W}_i^m,\mathbf{W}_{nbs}^m\right). \tag{7}$$

A sparse linear system can be formed with the above equations for all elements, written as

$$\mathbf{A}\Delta \mathbf{W}=\mathbf{b}. \tag{8}$$

For a mesh that has *nelems* elements, matrix **A** is a sparse matrix consisting of *nelems* × *nelems* blocks and each block has a dimension of *nvars* × *nvars*, where *nvars* is the number of conservative variables. Matrix **A** can be split into a block diagonal matrix **D** and a block off-diagonal matrix **O**,

$$\mathbf{A}=\mathbf{D}+\mathbf{O}. \tag{9}$$

The diagonal blocks of matrix **A** are

$$\mathbf{D}_i=\frac{\mathbf{I}}{\Delta t}+\frac{\partial \mathbf{R}_i^m}{\partial \mathbf{W}_i}, \tag{10}$$

and the non-zero off-diagonal blocks of **A** are

$$\mathbf{O}_{i,nb}=\frac{\partial \mathbf{R}_i^m}{\partial \mathbf{W}_{nb}}. \tag{11}$$

The entries of the right-hand-side vector **b** are given by

$$\mathbf{b}_i=-\mathbf{R}\left(\mathbf{W}_i^m,\mathbf{W}_{nbs}^m\right). \tag{12}$$

Once the left-hand-side matrix and the right-hand-side vector of Eq. (8) have been constructed, the Gauss-Seidel method is employed to solve the equation. In order to parallelize the Gauss-Seidel iteration on the GPU, each mesh element is painted with a different color from its neighbors. Calculations are performed in parallel for mesh elements with the same color, while their neighboring elements in other colors remain unchanged to avoid data race conditions. Consequently, Eq. (8) can be safely solved for mesh elements of one color after another. During the Gauss-Seidel iteration, the most recently updated solution variables will be used, which is denoted as $\Delta \mathbf{W}^*$. Then the solution for Eq. (8) is written as

$$\Delta \mathbf{W}=\mathbf{D}^{-1}(\mathbf{b}-\mathbf{O}\Delta \mathbf{W}^*). \tag{13}$$

The procedure to loop over all colors to compute $\Delta \mathbf{W}$ is called a sweep. There can be several sweeps based on empirical settings. When the sweeps are completed, the entire flow solution will be updated by

$$\mathbf{W}^{m+1}=\mathbf{W}^m+\Delta \mathbf{W}. \tag{14}$$

The algorithm is summarized in Algorithm 1.

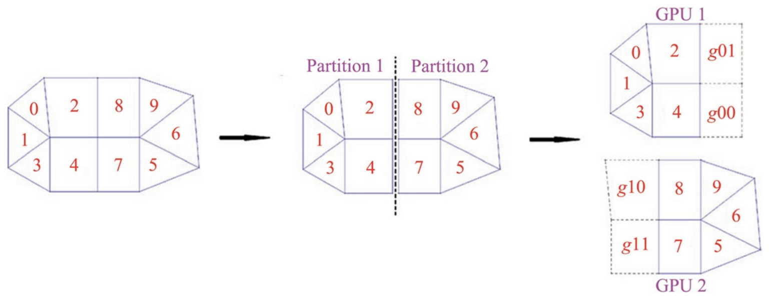

Fig. 2 Example of a mesh with 10 elements

Algorithm 1 Multi-colored Gauss-Seidel solver

```
1: Compute $\mathbf{b}_i = -\mathbf{R}(\mathbf{W}_i^m, \mathbf{W}_{nbs}^m)$ for all elements to construct the right-hand-side vector
2: Compute $\mathbf{D}_i = (\frac{\mathbf{I}}{\Delta t} + \frac{\partial \mathbf{R}_i^m}{\partial \mathbf{W}_i})$ for all elements to construct the diagonal blocks of the left-hand-side matrix
3: Compute $\mathbf{O}_{i,nb} = \frac{\partial \mathbf{R}_i^m}{\partial \mathbf{W}_{nb}}$,$nb$=1,...,$nbs$ for all elements to construct the off-diagonal blocks of the left-hand-side matrix
4: Invert $\mathbf{D}_i$ using Gauss-Jordan elimination for all elements to get $\mathbf{D}_i^{-1}$
5: Initialize $\Delta\mathbf{W} = 0$
6: for $ns$ in range($nsweeps$) do
7:     for $current\_color$ in $palette$ do
8:         $\Delta\mathbf{W} = \mathbf{D}^{-1}(\mathbf{b} - \mathbf{O}\Delta\mathbf{W}^*)$ for elements painted by $current\_color$
9:     end for
10: end for
11: $\mathbf{W}^{m+1} = \mathbf{W}^m + \Delta\mathbf{W}$
```

3.4 Distributed storage and computation

In this subsection, we demonstrate how the matrices and vectors are stored across GPUs using a simple example on a mesh with 10 elements, as illustrated in Fig. 2. The corresponding equation for the entire mesh is expanded in Eq. (15),

$$\begin{pmatrix} \mathbf{D}_0 & \mathbf{O}_{0,1} & \mathbf{O}_{0,2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \mathbf{O}_{1,0} & \mathbf{D}_1 & 0 & \mathbf{O}_{1,3} & 0 & 0 & 0 & 0 & 0 & 0 \\ \mathbf{O}_{2,0} & 0 & \mathbf{D}_2 & 0 & \mathbf{O}_{2,4} & 0 & 0 & 0 & \mathbf{O}_{2,8} & 0 \\ 0 & \mathbf{O}_{3,1} & 0 & \mathbf{D}_3 & \mathbf{O}_{3,4} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{O}_{4,2} & \mathbf{O}_{4,3} & \mathbf{D}_4 & 0 & 0 & \mathbf{O}_{4,7} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \mathbf{D}_5 & \mathbf{O}_{5,6} & \mathbf{O}_{5,7} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \mathbf{O}_{6,5} & \mathbf{D}_6 & 0 & 0 & \mathbf{O}_{6,9} \\ 0 & 0 & 0 & 0 & \mathbf{O}_{7,4} & \mathbf{O}_{7,5} & 0 & \mathbf{D}_7 & \mathbf{O}_{7,8} & 0 \\ 0 & 0 & \mathbf{O}_{8,2} & 0 & 0 & 0 & 0 & \mathbf{O}_{8,7} & \mathbf{D}_8 & \mathbf{O}_{8,9} \\ 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{O}_{9,6} & 0 & \mathbf{O}_{9,8} & \mathbf{D}_9 \end{pmatrix} \begin{pmatrix} \Delta\mathbf{W}_0 \\ \Delta\mathbf{W}_1 \\ \Delta\mathbf{W}_2 \\ \Delta\mathbf{W}_3 \\ \Delta\mathbf{W}_4 \\ \Delta\mathbf{W}_5 \\ \Delta\mathbf{W}_6 \\ \Delta\mathbf{W}_7 \\ \Delta\mathbf{W}_8 \\ \Delta\mathbf{W}_9 \end{pmatrix} = \begin{pmatrix} \mathbf{b}_0 \\ \mathbf{b}_1 \\ \mathbf{b}_2 \\ \mathbf{b}_3 \\ \mathbf{b}_4 \\ \mathbf{b}_5 \\ \mathbf{b}_6 \\ \mathbf{b}_7 \\ \mathbf{b}_8 \\ \mathbf{b}_9 \end{pmatrix}. \tag{15}$$

Assuming that the equation is about to be solved using 2 GPUs, then the mesh is partitioned into two partitions. Elements 0 to 4 are assigned to partition 1, while elements 5 to 9 are assigned to partition 2.

It is reasonably clear that each GPU should store the matrix entries from the overlapping region of the rows and columns corresponding to the indices of its mesh elements. However, in order to complete the computation, it is necessary to access the non-zero off-diagonal blocks in the columns whose indices belong to other partitions. For instance, GPU 1 needs to access $\mathbf{O}_{2,8}$ and $\mathbf{O}_{4,7}$. These two blocks are calculated using data from elements 7 and 8, respectively, which are located on GPU 2. This presents a challenge, and GPU 2 encounters a similar situation. It can be observed that non-zero blocks $\mathbf{O}_{2,8}$, $\mathbf{O}_{4,7}$, $\mathbf{O}_{7,4}$ and $\mathbf{O}_{8,2}$ arise on the columns corresponding to mesh elements adjacent to the boundary between two partitions. In this work, ghost elements are created to help handle this situation. To avoid confusion, the original mesh elements are referred to as real mesh elements when needed. We take GPU 1 as an example to explain how it works. Ghost elements $g00$ and $g01$ are created in partition 1 as shown in Fig. 2. The data exchange module using MPI communications will copy data of real mesh elements 7 and 8 from GPU 2 to ghost elements $g00$ and $g01$ on GPU 1 respectively. The data exchange procedure is depicted in Fig. 3. Subsequently, the off-diagonal blocks $\mathbf{O}_{2,8}$ and $\mathbf{O}_{4,7}$ in the left-hand-side matrix can be computed equivalently using the data from ghost elements $g00$ and $g01$, and written as

$$\mathbf{O}_{2,8} = \mathbf{O}_{2,g01}, \tag{16}$$

and

$$\mathbf{O}_{4,7} = \mathbf{O}_{4,g00}. \tag{17}$$

Therefore, for each partition, the data of both the real mesh elements and the ghost elements must be stored. For the left-hand-side matrix $\mathbf{A}$, we eliminate the columns with all zeros and collect only the columns for the ghost elements and place them to the right side of the columns for the real mesh elements. For the vector $\Delta\mathbf{W}$, the ghost element entries are concatenated to the end of the real element entries. The vector $\mathbf{b}$ remains unchanged. The actual data storage and computation for partition 1 on GPU 1 is shown in Eq. (18),

$$\begin{pmatrix} \mathbf{D}_0 & \mathbf{O}_{0,1} & \mathbf{O}_{0,2} & 0 & 0 & 0 & 0 \\ \mathbf{O}_{1,0} & \mathbf{D}_1 & 0 & \mathbf{O}_{1,3} & 0 & 0 & 0 \\ \mathbf{O}_{2,0} & 0 & \mathbf{D}_2 & 0 & \mathbf{O}_{2,4} & 0 & \mathbf{O}_{2,g01} \\ 0 & \mathbf{O}_{3,1} & 0 & \mathbf{D}_3 & \mathbf{O}_{3,4} & 0 & 0 \\ 0 & 0 & \mathbf{O}_{4,2} & \mathbf{O}_{4,3} & \mathbf{D}_4 & \mathbf{O}_{4,g00} & 0 \end{pmatrix} \begin{pmatrix} \Delta\mathbf{W}_0 \\ \Delta\mathbf{W}_1 \\ \Delta\mathbf{W}_2 \\ \Delta\mathbf{W}_3 \\ \Delta\mathbf{W}_4 \\ \Delta\mathbf{W}_{g00} \\ \Delta\mathbf{W}_{g01} \end{pmatrix} = \begin{pmatrix} \mathbf{b}_0 \\ \mathbf{b}_1 \\ \mathbf{b}_2 \\ \mathbf{b}_3 \\ \mathbf{b}_4 \end{pmatrix}, \tag{18}$$

and the actual data storage and computation for partition 2 on GPU 2 is shown in Eq. (19),

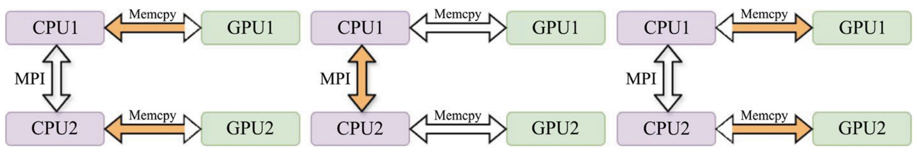

Fig. 3 Example of data exchange using MPI

$$\begin{pmatrix} \mathbf{D}_5 & \mathbf{O}_{5,6} & \mathbf{O}_{5,7} & 0 & 0 & 0 & 0 \\ \mathbf{O}_{6,5} & \mathbf{D}_6 & 0 & 0 & \mathbf{O}_{6,9} & 0 & 0 \\ \mathbf{O}_{7,5} & 0 & \mathbf{D}_7 & \mathbf{O}_{7,8} & 0 & 0 & \mathbf{O}_{7,g11} \\ 0 & 0 & \mathbf{O}_{8,7} & \mathbf{D}_8 & \mathbf{O}_{8,9} & \mathbf{O}_{8,g10} & 0 \\ 0 & \mathbf{O}_{9,6} & 0 & \mathbf{O}_{9,8} & \mathbf{D}_9 & 0 & 0 \end{pmatrix} \begin{pmatrix} \Delta\mathbf{W}_5 \\ \Delta\mathbf{W}_6 \\ \Delta\mathbf{W}_7 \\ \Delta\mathbf{W}_8 \\ \Delta\mathbf{W}_9 \\ \Delta\mathbf{W}_{g10} \\ \Delta\mathbf{W}_{g11} \end{pmatrix} = \begin{pmatrix} \mathbf{b}_5 \\ \mathbf{b}_6 \\ \mathbf{b}_7 \\ \mathbf{b}_8 \\ \mathbf{b}_9 \end{pmatrix}. \tag{19}$$

To sum up, for partition i, if $nelems_i$ is the number of mesh elements and $nghosts_i$ represents the number of ghost elements, then matrix $\mathbf{A}$ is a $nelems_i \times (nelems_i + nghosts_i)$ block matrix, where the block size is $nvars \times nvars$, and the dimensions of vector $\Delta\mathbf{W}$ and vector $\mathbf{b}$ are $nvars \times (nelems_i + nghosts_i)$ and $nvars \times nelems_i$ respectively. The diagonal blocks are stored in an array of the form $(nvars, nvars, nelems_i)$ and the off-diagonal matrix is stored in a block Compressed Sparse Row (CSR) format for saving memories.

It is worth noting that the number of ghost elements will usually be about 1% of the number of real mesh elements, according to our practical experience. A benefit of the current approach for data storage and computation is that it works for both multi-GPU and single-GPU environments. It can recover to the single-GPU version simply by setting the number of ghost elements to zero. The GPU kernels for the Gauss-Seidel iteration which has been developed for single-GPU can be used for multi-GPU straightforwardly. The Sparse Matrix-Vector (SpMV) multiplication involved in the Gauss-Seidel procedure is known to be memory bound on GPUs.

3.5 Parallel coloring algorithm

Since an approximate nearest-neighbor linearization of the residual is applied in the derivation, any distance-1 sequential coloring algorithm will satisfy the requirements for the MCGS methods in single-GPU settings. In this work, a parallel coloring algorithm for multi-GPU environments is built on top of a greedy distance-1 sequential coloring algorithm. This subsection presents the sequential coloring algorithm, followed by the parallel coloring algorithm.

The sequential coloring algorithm is presented in Algorithm 2. The colors utilized in the painting process are stored in a list, designated as the *palette*, with the color index starting from 1. The *palette* is initialized with two colors, and additional colors may be appended when needed during the painting process. The concept of using a color counter [8] is adopted to guarantee a balanced distribution of each color. A target set is used for the purpose of holding the unpainted mesh element candidates. The target set is initialized by putting the mesh element with index 0 into it. Subsequently, the painting process commences. If the target set is not empty, take an element out of it. Then, the neighbours of this element are traversed, and the colors of those that have already been painted are recorded. Meanwhile, the unpainted neighbours are added to the target set. Colors in the *palette* that have not been used by the neighbors are denoted as available colors. In the event that there exists at least one available color, the element should be painted with the color that has the minimal color count. Otherwise, a new color should be added to the *palette* and the element painted with the new color. Once the element has been painted, the color counter should be updated and we move to paint the next element in the target set. The process should be terminated when all mesh elements in the target set have been painted. Typically, to paint a three-dimensional unstructured hybrid mesh for cell-centered finite volume schemes will end up with a total of four to six colors.

Algorithm 2 Sequential coloring algorithm

```
palette=[1,2]
color_cts = {c:0 for c in palette}
target_set.add(elem_0)
end_flag=0
while end_flag==0 do
    elem=target_set.pop()
    nb_colors=set()
    for nb=1,nbs do
        if nb_color==0 then
            target_set.add(nb_elem)
        else
            nb_colors.add(nb_color)
        end if
    end for
    avail_colors=set(palette)-nb_colors
    if len(avail_colors)==0 then
        palette.append(palette[-1]+1)
        color_cts[palette[-1]]=0
        paint_color=palette[-1]
    else
        avail={c:color_cts[c] for c in avail_colors}
        paint_color = min(avail,key=avail.get)
    end if
    elem_color[elem]=paint_color
    color_cts[paint_color]+=1
    if len(target_set)==0 then
        end_flag=1
    end if
end while
```

The parallel coloring algorithm is shown in Algorithm 3, which is divided into several steps. First of all, each MPI process performs a greedy distance-1 sequential coloring on its mesh partition in parallel. This ensures that interior mesh elements in each partition are correctly colored, while color conflicts can only occur at the MPI boundaries. Then, data exchanges for element colors are performed between partitions such that the element colors at MPI boundaries are the most up-to-date. Next, each MPI process loops over the mesh elements at the MPI boundaries to detect if there exists any color conflicts. We set a rule that if a conflict occurs, the color of the mesh element belonging to the mesh partition with the smaller partition index is erased. When all conflict detection is completed, each MPI process recolors the mesh elements whose colors have been erased. The above detection and recoloring procedure can be repeated several times to completely resolve all conflicts.

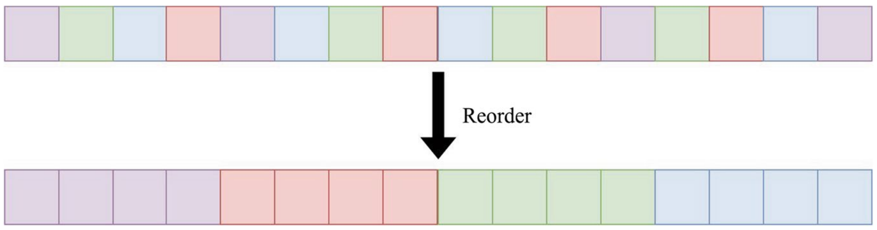

Fig. 4 Illustration of the color pattern for mesh elements before and after reordering

Algorithm 3 Parallel coloring algorithm

```
1:  Perform a greedy distance-1 sequential coloring algorithm for each MPI process
2:  Perform color exchanges for ghost elements for each MPI process
3:  Detect conflicts for each MPI process
4:  Sum up the total number of conflicts $N_c$
5:  while $N_c > 0$ do
6:      Recolor conflicted elements for each MPI process
7:      Perform color exchanges for ghost elements for each MPI process
8:      Detect conflicts for each MPI process
9:      Sum up the total number of conflicts $N_c$
10: end while
```

Once the mesh elements have been painted, the color pattern for the mesh elements will be found to resemble the squares displayed in the upper part of Fig. 4, which is completely random. This type of color pattern is not optimal for GPU to load and store data given that the MCGS solver operates on mesh elements of the same color simultaneously. For an efficient implementation, it thus becomes necessary to reorder the mesh elements in such a way that those of the same color are grouped together, as shown in the lower part of Fig. 4.

4 Numerical simulations of the CHN-T1 model

The benchmark model CHN-T1 was developed by CARDC for modern single-aisle commercial aircraft [22]. It has been studied in the wind tunnel [23] as well as in the first aeronautical computational fluid dynamics credibility workshop (AeCW-1) [24] for several different configurations. In this work, we focused on the baseline BWHV configuration, which consisted of a body, a wing, a horizontal tail, and a vertical tail. The meshes were generated based on a half-model of the configuration, and the computational domain resembled a hemisphere as shown in Fig. 5. The flow conditions and reference quantities are listed in Table 1.

Three unstructured meshes with mixed elements were available from the AeCW-1 workshop, named CHNT1-c, CHNT1-m, and CHNT1-f. The types of mixed elements were tetrahedron, prism, and pyramid. The total number of elements for the three meshes was 6518485, 17376357, and 49499123, respectively. A view of the CHNT1-c mesh around the aircraft is displayed in Fig. 6, where prism elements can be seen in the near-body region. The detailed information of the meshes is shown in Table 2.

All test cases were performed on a server consisting of an Intel CPU and 8 NVIDIA A100 40GiB PCI-e GPUs. The meshes were partitioned by PyMetis using the multilevel *k*-way partitioning algorithm. The MCGS solver was set with 10 sweeps per iteration and

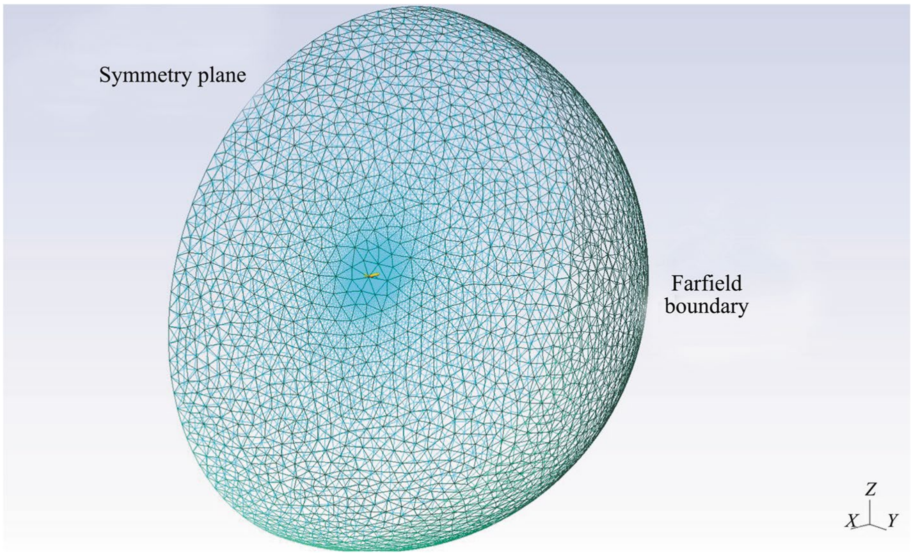

Fig. 5 Illustration of the CHN-T1 model and computational domain

Table 1 Flow conditions and reference quantities

Name	Value
Mach number	0.78
Chord Reynolds number	3.3 million
Reference temperature	290 K
Reference chord	0.324 m
Reference area (half model)	0.3605 m^2
Moment reference point	(1.10543 m, 0, 0)

a Courant-Friedrichs-Lewy (CFL) number of 100 was utilized for the flow solution. The parameter χ in the UMUSCL reconstruction was set to 1/3. The Venkatakrishnan limiter was frozen after the norm of the density residual decreased by 3.5 orders of magnitude. The Roe flux scheme with an entropy fix coefficient of 0.1 was adopted for the inviscid flux calculation. Convergence information was monitored and printed for every 500 time steps.

In this work, mesh elements were observed to be painted with six distinct colors in both cases, with either a single GPU or multiple GPUs in use. It was demonstrated that the number of color counts between any two colors differed by no more than two, thereby confirming that the developed algorithm is capable of effectively producing a balanced colored mesh. The MCGS solver will benefit from a smaller number of colors and larger, more balanced subproblem sizes for efficiency and convergence.

4.1 Verification for the parallelization

Before starting to use the flow simulation framework developed in this work, it is necessary to verify that the parallelization over multiple GPUs with MPI communication is properly implemented. Attention must be paid for that due to the parallel coloring algorithm used, the coloring patterns for the same mesh will be different when working with

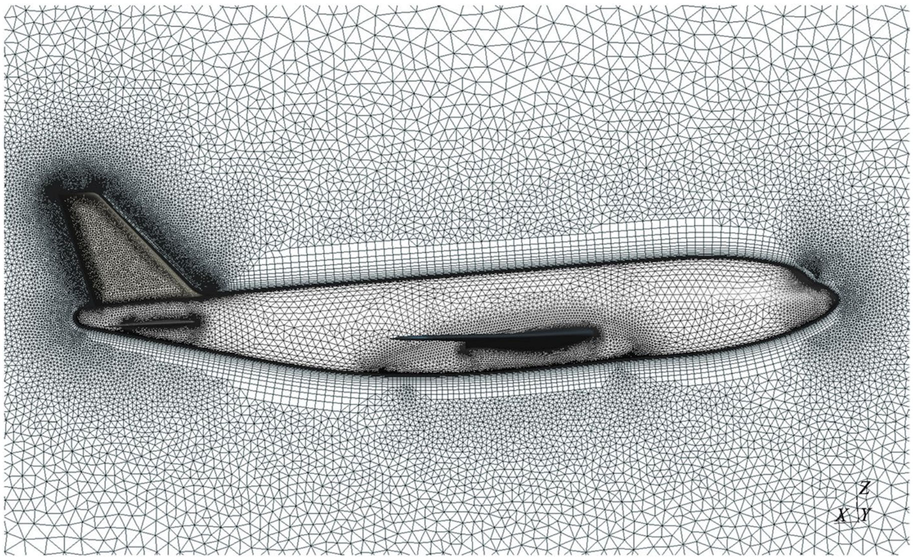

Fig. 6 A view of the CHNT1-c mesh

Table 2 Numbers of mesh elements (in millions)

Mesh	Tetrahedron	Prism	Pyramid	Total
CHNT1-c	2.94	3.55	0.03	6.52
CHNT1-m	8.44	8.88	0.05	17.38
CHNT1-f	22.89	26.48	0.12	49.50

different numbers of GPUs. This may result in slightly different convergence behavior for the MCGS solver. However, if the flow solution is sufficiently converged, the same test case should produce consistent results on any number of GPUs.

The verification was carried out on the CHNT1-m mesh at an Angle Of Attack (AOA) of 2.0°. The test cases were terminated when the norm of the density residual was reduced by 5 orders of magnitude. The computed lift coefficients, drag coefficients, and pitching moment coefficients are displayed in Table 3. The first few significant digits of the results on multiple GPUs are almost identical to those on a single GPU. The max relative deviation for the lift coefficient, the drag coefficient and the pitching moment coefficient is 0.02%, which indicates that the implementation of the MPI parallelization is correct.

4.2 Grid convergence study

In this subsection, a grid convergence study was performed at a fixed lift coefficient of 0.5000 (±0.001) using the meshes from Table 2. The calculated angles of attack, force coefficients, and pitching moment coefficients are displayed in Table 4 along with the Richardson extrapolation estimates [25]. The experimental data [26] is also given in Table 4 for reference. It should be noted that in the wind tunnel experiment, a support system was attached to the bottom of the model, which was not included in our numerical simulations. The AeCW-1 report summarized that the inclusion of the support

Table 3 Verification for the parallelization

GPUs	Lift Coefficient	Drag Coefficient	Pitching Moment Coefficient
1	0.4193315344321991	0.03027457734981084	0.05720560147290862
2	0.4193309161094509	0.03027515547206842	0.05721397356794956
4	0.4193352535221005	0.03027452757130499	0.05721671107808820
8	0.4193473414563671	0.03027297187572368	0.05721208643986125

Table 4 Grid convergence study for BWHV configuration at fixed $C_l = 0.5$

Mesh	AOA	Lift Coefficient	Drag Coefficient	Pitching Moment Coefficient
CHNT1-c	2.5870°	0.5000	0.03533	0.03938
CHNT1-m	2.5510°	0.5000	0.03376	0.04096
CHNT1-f	2.5270°	0.5000	0.03259	0.04308
R.E. estimates	2.5032°	0.5000	0.03143	0.04496
Experiment (FL-26)	2.6875°	0.5000	0.03197	0.0001

system resulted in a better agreement with the experimental data, and Li et al. [26] reported that the inclusion of the support system had less effect on the force coefficients but significantly improved the pitch moment coefficient.

The angle of attack for both our results and the median data from AeCW-1 is plotted in Fig. 7. The angle of attack decreased monotonically as the mesh became finer, which agrees well with the observations made in the AeCW-1 summary report for test cases on unstructured meshes. In addition, the AeCW-1 summary report stated that participants using the SA turbulence model obtained smaller angles of attack than those using the SST turbulence model, which explains why our results were generally lower than the median data from AeCW-1.

The grid-convergence behavior for the drag coefficient is illustrated in Fig. 8. The drag coefficient tended to be lower at better mesh resolutions. Our predictions were higher than the AeCW-1 median data, but the difference was narrowing on finer meshes. The estimate for the drag coefficient by Richardson extrapolation was 0.03143, which was in excellent agreement with the experimental drag coefficient.

The trend of the pitching moment coefficient is depicted in Fig. 9. The median data for the pitching moment coefficient reported in AeCW-1 was nearly constant at 0.044 on all mesh levels. Our calculation predicted a pitching moment coefficient of below 0.04 on the coarse mesh, but caught up on the fine mesh. Both our data and the AeCW-1 median data were significantly higher than the experimental data. The reason was explained by Li et. al [26] that the inclusion of the support system caused the tail to produce a nose-down pitching moment.

4.3 Performance analysis

In this subsection, we analyze the performance of the developed flow simulation framework. The memory usage for test cases with different sizes of meshes was evaluated and displayed in Table 5. The CHNT1-c and the CHNT1-m cases occupied approximately 14

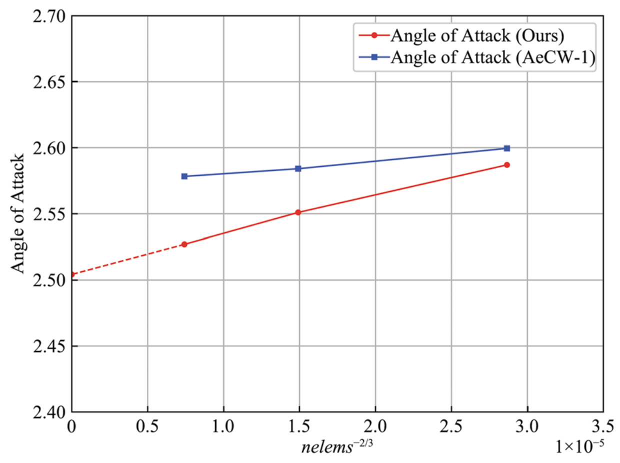

Fig. 7 Grid convergence of the angle of attack

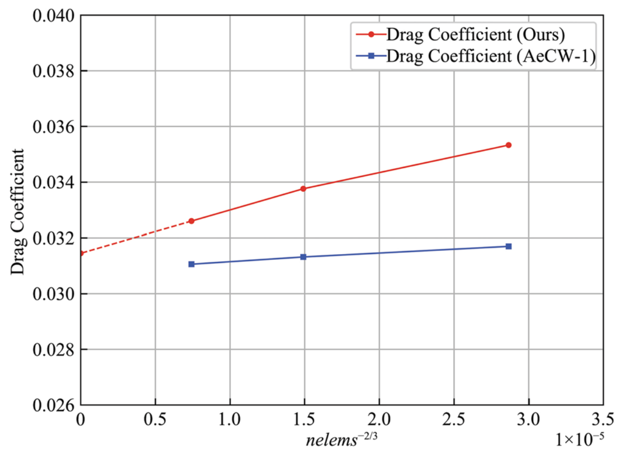

Fig. 8 Grid convergence of the drag coefficient

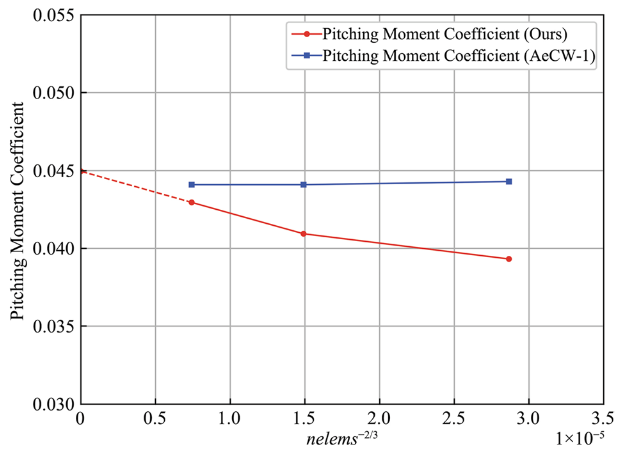

Fig. 9 Grid convergence of the pitching moment coefficient

Table 5 Total GPU memory usage on a single NVIDIA A100 40GiB PCI-e GPU

Case	CHNT1-c Case	CHNT1-m Case	CHNT1-f Case
Memory usage	14524 MiB	37806 MiB	OOM

GiB and 37 GiB respectively. The CHNT1-f case was too large to run on a single GPU, resulting in an Out of Memory (OOM) error, and we calculated that the CHNT1-f case requires approximately 105 GiB of GPU memory. It can be observed that the memory usage was almost linearly proportional to the number of mesh elements for the developed MCGS solver. This is due to the fact that the Jacobian matrix represents the predominant portion of the total storage, and in this work, it was approximately linearly proportional to the number of block rows. As a result, the memory requirement of unseen large test cases can be readily estimated in advance by this rule.

The kernel for the Gauss-Seidel iteration was identified as the most significant portion of the total computational time for the GPU-based computation. The reordering of mesh elements by color was a crucial factor in the performance of this kernel. Table 6 presents the statistics of the performance metrics for the Gauss-Seidel kernel, where the data were collected from a CHNT1-c test case. It can be seen that, with the implementation of reordering, significantly more coalesced memory accesses were achieved, and the number of instructions for both global load and global store was notably decreased. This resulted in a reduction in the memory traffic, thereby enhancing the performance of the memory-bounded kernel. Although there was a slight deterioration in the cache hit rate, the total execution time of the kernel was remarkably reduced.

Since our code can only run on GPUs and does not offer a CPU-based version, it is impossible to calculate an exact GPU-to-CPU speedup. However, by comparing its performance with the performance of several popular CFD software using similar numerical schemes on CPUs, valuable reference information can also be obtained. For the same RANS test cases and meshes, Xu et al. [27] provided the performance on CPUs of three unstructured CFD software, including their in-house code TNS, an open-source code SU2 [28], and the commercial software Fluent. The comparisons are shown in Tables 7, 8, and 9. All the runs utilized the SA turbulence model. For the solvers, the runs on CPUs used the LU-SGS, while we used the MCGS; both were variants of the implicit Gauss-Seidel methods. The Jameson-Schmidt-Turkel (JST) scheme [29] was selected for the runs by both TNS and SU2, while the Roe flux-difference splitting scheme [20] was adopted for the runs by both Fluent and our code.

Table 6 Statistics of the performance metrics for the Gauss-Seidel kernel

Metrics	Without reordering	With reordering
Global load instructions	35.29 M	5.98 M
Global store instructions	1017.40 K	167.99 K
Device memory read data size	10.64 GB	5.82 GB
Device memory write data size	166.95 MB	41.87 MB
L2 cache hit rate	46.5%	31.46%
Average execution time	8.044 milliseconds	4.288 milliseconds

Table 7 Comparison of computational costs on the CHNT1-c mesh

Software	Spatial	Temporal	Multigrid	Hardware	Time(h)	Total Cost
TNS [27]	JST	LU-SGS	Yes	32 CPU cores	9.20	294.4 CPU core hours
SU2 [27]	JST	LU-SGS	No	28 CPU cores	9.44	264.32 CPU core hours
Fluent [27]	Roe	LU-SGS	Yes	28 CPU cores	14.77	413.56 CPU core hours
Ours	Roe	MCGS	No	8 GPUs	0.09	0.72 GPU hours

Table 8 Comparison of computational costs on the CHNT1-m mesh

Software	Spatial	Temporal	Multigrid	Hardware	Time(h)	Total Cost
TNS [27]	JST	LU-SGS	Yes	32 CPU cores	15.22	487.04 CPU core hours
SU2 [27]	JST	LU-SGS	No	56 CPU cores	9.54	534.24 CPU core hours
Fluent [27]	Roe	LU-SGS	Yes	56 CPU cores	28.95	1621.20 CPU core hours
Ours	Roe	MCGS	No	8 GPUs	0.31	2.48 GPU hours

Table 9 Comparison of computational costs on the CHNT1-f mesh

Software	Spatial	Temporal	Multigrid	Hardware	Time(h)	Total Cost
TNS [27]	JST	LU-SGS	Yes	32 CPU cores	64.22	2055.04 CPU core hours
SU2 [27]	JST	LU-SGS	No	112 CPU cores	17.54	1964.48 CPU core hours
Fluent [27]	Roe	LU-SGS	Yes	112 CPU cores	47.85	5359.20 CPU core hours
Ours	Roe	MCGS	No	8 GPUs	1.19	9.52 GPU hours

Another difference that should be mentioned is that the runs by both TNS and Fluent were accelerated using multigrid methods, while the runs by both SU2 and our code were done without multigrid methods. For a fair comparison, the wall-clock times were recorded based on the same convergence criterion as that used in Ref. [27], which required the norm of the density residual to drop by 5 orders of magnitude. The pre-processing time for coloring and reordering mesh elements was also included in our code.

The spatial and temporal schemes used in the Fluent test cases were the most similar to ours, even though the multigrid was switched on for Fluent. Our code using the 8-GPU server can match up with Fluent using 4500 to 5200 CPU cores. This performance was comparable to the reported speedups for automotive and aerospace external aerodynamic benchmarks found in the official document of the recent Fluent multi-GPU solver [2]. Compared to both TNS and SU2, the performance of our code on the 8-GPU server was roughly equivalent to the performance of using about 1500 to 3200 CPU cores. This performance was acceptable considering that the Roe scheme was computationally more expensive than the JST scheme. Overall, the results met our expectations that each GPU would deliver the same performance as a few hundred CPU cores.

Parallel efficiency is an important metric that characterizes the performance of a parallel code. In this work, the strong scaling performance on multiple GPUs was evaluated. The parallel efficiency of the scaling test was calculated using

$$\mathrm{PE} = \frac{N/M}{N_\mathrm{b}/M_\mathrm{b}} \times \frac{T_\mathrm{b}}{T} \times 100\%, \tag{20}$$

where N and M represent the number of mesh elements and the number of GPUs respectively, and T is the wall-clock time used for the test. The subscript 'b' indicates that is from the baseline test.

The CHNT1-m mesh was selected for the strong scaling test. The test was conducted using different numbers of GPUs ranging from 1 to 8. The total wall-clock time to drive the norm of the density residual to be reduced by 5 orders of magnitude was recorded. The results in Table 10 show that as the number of GPUs doubled, the workload represented by the elements per GPU was reduced by half. As more GPUs were utilized, it took less wall-clock time to complete the work; meanwhile, communication time took up a larger portion of the total execution time, causing the parallel efficiency to decrease. For the 8-GPU configuration, the parallel efficiency of the developed code dropped to 92.94%. Nevertheless, this performance can be considered competitive when compared to that of Fluent's pressure-based GPU solver on an automotive aerodynamics case with 105 million mesh elements, for which the strong scaling efficiency using 8 GPUs (NVIDIA A100 80GiB) was 78.8% [2].

5 Conclusions

In this paper, a multi-colored Gauss-Seidel solver is developed for a multi-GPU environment for steady-state RANS simulations on unstructured hybrid meshes. A parallel coloring algorithm is implemented to efficiently color the mesh elements. Non-blocking MPI communication is used for data exchange between mesh partitions. A verification test case is performed to ensure that the parallelization of the code on multiple GPUs using MPI is correct. A grid convergence study is performed on the CHNT1 transport aircraft model with a fixed lift coefficient, which shows comparable predictions to the results reported in AeCW-1. The performance analysis shows that the developed GPU-based solver is significantly faster than the CPU-based solvers using similar numerical schemes. The equivalent number of CPU cores needed to match the performance of a server with 8 GPUs (NVIDIA A100 40GiB) could be in the thousands. The strong scaling performance across multiple GPUs is also competitive when compared to the performance of popular commercial software.

When each GPU processes about 17 million mesh elements, the current code has consumed nearly all of the 40 GiB of GPU memory. Mixed-precision algorithms [5] for the MCGS solver may be considered in the future, where the matrix $\mathbf{O}$ and the vector $\Delta\mathbf{W}$ are stored in single precision. This requires less GPU memory while also reducing memory traffic for the memory-bounded Gauss-Seidel iteration kernel. Moreover, given that the vector $\Delta\mathbf{W}$ is frequently exchanged between GPUs, storing it in single precision will result in a reduction in the amount of work for MPI communications.

Table 10 Strong scaling test

Mesh	GPUs	Elements/GPU	Normalized Time	Parallel Efficiency
CHNT1-m	1	17.38 million	1.0	-
CHNT1-m	2	8.68 million	0.5131	97.45%
CHNT1-m	4	4.34 million	0.2636	94.84%
CHNT1-m	8	2.17 million	0.1345	92.94%

Acknowledgements
The authors would like to acknowledge the support from Shanghai Pujiang Program.

Authors' contributions
LY worked on the code development, test case profiling, and manuscript writing. JY helped in the discussion. All authors read and approved the final manuscript.

Funding
This work was sponsored by Shanghai Pujiang Program (Grant No. 22PJ1420200).

Data availability
Data will be made available from the corresponding author upon reasonable request.

Declarations

Competing interests
The author declare that they have no competing interests.

Received: 23 July 2024 Revised: 4 November 2024 Accepted: 25 November 2024
Published online: 23 June 2025

References
1. Appa J, Turner M, Ashton N (2021) Performance of CPU and GPU HPC architectures for off-design aircraft simulations. Paper presented at the AIAA SciTech 2021 Forum, Virtual, 11–15 & 19–21 January 2021
2. Petrone G (2022) Unleashing the power of multiple GPUs for CFD simulations. https://www.ansys.com/blog/unleashing-the-power-of-multiple-gpus-for-cfd-simulations. Accessed 30 June 2024
3. Obiso D (2023) Fasten your seat belts: the coupled solver is taking off on GPU! https://blogs.sw.siemens.com/simcenter/cfd-coupled-solver-is-taking-off-on-gpu. Accessed 30 June 2024
4. Sato Y, Hino T, Ohashi K (2013) Parallelization of an unstructured Navier-Stokes solver using a multi-color ordering method for OpenMP. Comput Fluids 88:496–509
5. Walden A, Nielsen E, Diskin B et al (2019) A mixed precision multicolor point-implicit solver for unstructured grids on GPUs. In: 2019 IEEE/ACM 9th Workshop on Irregular Applications: Architectures and Algorithms (IA3), Denver, 18 November 2019.
6. Nguyen MT, Castonguay P, Laurendeau É (2018) GPUs accelerated three-dimensional RANS solver for aerodynamic simulations on multiblock grids. In: 10th International Conference on Computational Fluid Dynamics (ICCFDU 2018), Barcelona, 9-13 July 2018
7. Zhang J, Dai Z, Li R et al (2023) Acceleration of a production-level unstructured grid finite volume CFD code on GPU. Appl Sci 13(10):6193
8. Watkins J, Romero J, Jameson A (2016) Multi-GPU, implicit time stepping for high-order methods on unstructured grids. In: 46th AIAA Fluid Dynamics Conference, Washington DC, 13-17 June 2016
9. Walden A, Nielsen E, Nastac G (2020) Accelerating FUN3D solutions using GPU hardware. In: FUN3D GPU Training, Virtual, 20 October 2020.
10. Mishra S, Witherden F, Chakravorty D et al (2023) Scaling study of flow simulations on composable cyberinfrastructure. Practice and Experience in Advanced Research Computing 2023: Computing for the Common Good. Association for Computing Machinery, New York, pp 221–225
11. Zhang X, Guo X, Weng Y et al (2023) Hybrid MPI and CUDA paralleled finite volume unstructured CFD simulations on a multi-GPU system. Future Gener Comput Syst 139:1–16
12. Bogle I, Slota GM, Boman EG et al (2022) Parallel graph coloring algorithms for distributed GPU environments. Parallel Comput 110:102896
13. Yang L, Yang J (2023) GPU-accelerated flow simulations on unstructured grids using a multi-colored Gauss-Seidel method. In: Fu S (eds) 2023 Asia-Pacific International Symposium on Aerospace Technology (APISAT 2023) Proceedings. Lecture Notes in Electrical Engineering, vol 1050. Springer, Singapore, pp 657–671
14. Bayer M (2024) Mako: Hyperfast and lightweight templating for the Python platform. https://www.makotemplates.org. Accessed 30 June 2024
15. Schlömer N (n.a.) meshio: I/O for mesh files. https://pypi.org/project/meshio. Accessed 30 June 2024
16. Karypis G, Kumar V (1997) METIS: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. https://hdl.handle.net/11299/215346. Accessed 30 June 2024
17. Spalart PR, Allmaras SR (1992) A one-equation turbulence model for aerodynamic flows. In: 30th Aerospace Sciences Meeting and Exhibit, Reno, 6-9 January 1992
18. White JA, Nishikawa H, Baurle RA (2019) Weighted least-squares cell-average gradient construction methods for the VULCAN-CFD second-order accurate unstructured grid cell-centered finite-volume solver. In: AIAA SciTech 2019 Forum, San Diego, 7-11 January 2019
19. Venkatakrishnan V (1995) Convergence to steady state solutions of the Euler equations on unstructured grids with limiters. J Comput Phys 118(1):120–130
20. Roe PL (1981) Approximate Riemann solvers, parameter vectors, and difference schemes. J Comput Phys 43(2):357–372
21. Mavriplis D (2003) Revisiting the least-squares procedure for gradient reconstruction on unstructured meshes. In: 16th AIAA Computational Fluid Dynamics Conference, Orlando, 23-26 June 2003

22. Yu Y, Zhou Z, Huang J et al (2018) Aerodynamic design of a standard model CHN-T1 for single-aisle passenger aircraft. Acta Aerodyn Sin 36(3):505–513
23. Li Q, Liu D, Xu X et al (2019) Experimental study of aerodynamic characterictics of CHN-T1 standard model in 2.4 m transonic wind tunnel. Acta Aerodyn Sin 37(2):337–344
24. Wang Y, Liu G, Chen Z (2019) Summary of the first aeronautical computational fluid dynamics credibility workshop. Acta Aerodyn Sin 37(2):247–261
25. Vassberg J, Tinoco E, Mani M et al (2010) Summary of the fourth AIAA CFD drag prediction workshop. In: 28th AIAA Applied Aerodynamics Conference, Chicago, 28 June - 1 July 2010
26. Li W, Wang Y, Hong J et al (2019) Aerodynamic characteristics simulation of CHN-T1 model with TRIP3.0. Acta Aerodyn Sin 37(2):272–279
27. Xu C, Qiao J, Nie H et al (2019) Numerical investigation on aerodynamic performance of a standard model CHN-T1 using an unstructured flow solver. Acta Aerodyn Sin 37(2):291–300
28. Economon TD, Palacios F, Copeland SR et al (2016) SU2: An open-source suite for multiphysics simulation and design. AIAA J 54(3):828–846
29. Jameson A (2017) Origins and further development of the Jameson-Schmidt-Turkel scheme. AIAA J 55(5):1487–1510