Article

Overset Grid Assembler and Flow Solver with Adaptive Spatial Load Balancing

Orxan Shibliyev * and Ibrahim Sezai

Department of Mechanical Engineering, Eastern Mediterranean University, Gazimagusa, Mersin 10, 99628 Famagusta, Turkey; ibrahim.sezai@emu.edu.tr

* Correspondence: orhan.shibliyev@emu.edu.tr

Abstract: An overset mesh approach is useful for unsteady flow problems which involve components moving relative to each other. Since the generation of a single mesh around all components is prone to mesh stretching due to the relative motion of bodies, using the overset grid methodology, an individual mesh can be generated for each component. In this study, a parallel overset grid assembler was developed to establish connectivity across component meshes. Connectivity information was transferred to the developed parallel flow solver. The assembler uses multiple methods such as alternating digital tree and stencil walking to reduce the time spent on domain connectivity. Both the assembler and solver were partitioned spatially so that overlapping mesh blocks reside in the same partitions. Spatial partitioning was performed using a 3D space partitioning structure, namely octree, to which mesh blocks are registered. The octree was refined adaptively until bins of octree could be evenly distributed to processors. The assembler and solver were tested on a generic helicopter configuration in terms of load balance, scalability, and memory usage.

Keywords: computational fluid dynamics; numerical algorithms; overset grid methodology; load balancing

Citation: Shibliyev, O.; Sezai, I. Overset Grid Assembler and Flow Solver with Adaptive Spatial Load Balancing. *Appl. Sci.* **2021**, *11*, 5132. <https://doi.org/10.3390/app11115132>

Academic Editor: Wojciech Bożejko

Received: 25 April 2021

Accepted: 27 May 2021

Published: 31 May 2021

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2021 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

An overset mesh approach is useful for problems involving multiple moving components. Rotor blades are examples of multiple moving components in helicopter configuration [1]. The generation of a single mesh covering all components is possible if the mesh topology is limited to unstructured mesh. However, in the vicinity of boundaries, structured mesh may also be desired to accurately capture flow characteristics such as drag in viscous flows and shock in inviscid flows. Even if satisfactory single mesh is generated, in subsequent time steps, due to relative motion of components, the mesh may stretch excessively lowering the accuracy of flow solution unless proper mesh deformation technique is implemented. Mesh stretching can be avoided by regenerating the single grid in each time step, which is time consuming. An overset mesh approach allows component meshes to be generated only once and independently with the desired mesh topology.

Since each mesh is independent from other meshes, it is necessary to assemble the meshes in order to identify different cell types. The cell types are mainly

- the field cells on which flow solution is computed;
- the receptor cells which interpolate flow variables across overlapping mesh cells;
- the hole cells which are excluded from flow solution due to overlapping invalid regions of the domain, such as holes.

The identification of cell types is the core task of the assembler and is called donor search. In each time step, the assembler identifies and transfers the cell types to flow solver.

It is desired that the workload be distributed among multiple computing resources in order to reduce the total assembly and flow solution duration. The initial input mesh system is usually partitioned with unweighted graph partitioning, which produces partitions

containing an approximately equal number of cells based on cell connectivity. With overset mesh methodology, in the absence of cell-to-cell connectivity, overset meshes require interpolation of flow variables across overlapping mesh cells. The mesh cells are partitioned spatially so that the ones which occupy the same region reside in the same partition and, therefore, the communication time for interpolation of variables is diminished. Although spatial partitioning reduces the total run time by localizing the interpolation of flow variables, it does not alleviate load imbalance. Spatial partitioning is performed adaptively by refining the most loaded bin of an octree until balanced partitions are obtained.

Several assemblers [2–10] have managed to reduce total run time. To the knowledge of present authors, load balancing in the context of overset grid methodology is ignored in all but in only one of the studies given in [11], where load balance was maintained dynamically and adaptively by examining the task duration of each processor and by transferring the workload across processors in subsequent time steps. However, the mesh system was partitioned based solely on cell connectivity and, therefore, overlapping mesh blocks had to be exchanged for donor search and processed information needed to be sent back to the host processor. In this study, time is saved by eliminating the need for sending processed information back to the host processor since overlapping mesh blocks reside in the same partitions. This is similar to spatial decomposition volume [6], where mesh blocks are registered spatially into predefined geometric volumes such as coaxial cylinders and Cartesian mesh depending on the geometry of problem. This study improves upon [6] by using adaptive spatial refinement without predefined shapes in order to achieve load balance independent of problem geometry.

A generic helicopter [12] configuration is simulated in order to evaluate performance of the developed parallel assembler and flow solver in terms of load distribution, scalability and memory usage.

In Section 2, details of adaptive spatial partitioning and load balancing are explained. In Section 3, donor search and auxiliary algorithms are presented. Details about governing equations solved by the developed solver are presented in Section 6. CAD models/meshes for the simulated test case are explained in Section 7. In Section 8, results of the test case are presented and discussed and finally, in the last section, conclusions are drawn based on the results.

2. Spatial Partitioning and Load Balancing

Unlike the single mesh approach, the overset mesh technique involves interpolation of flow variables across overlapping meshes. In a parallel environment, it is mandatory to decompose mesh system (all the meshes including component and background meshes) and store overlapping mesh blocks in the same processors. In this work, this process is termed as spatial partitioning akin to spatial decomposition volume (SDV) [6]. In [6], the mesh system is decomposed into predetermined volumes based on the expected path of the component meshes. As an example, rotor blade meshes are decomposed into cylindrical volumes, whereas wing-pylon-store configuration is decomposed into Cartesian volumes. One of the drawbacks of SDV is the fact that the decomposition volume is problem dependent while the other is that no load balancing is performed. In this work, an octree is used for both adaptive spatial partitioning and load balancing. Adaptive spatial partitioning is similar to decomposition with Cartesian volumes but, in addition, it has the merit of adaptive partitioning of spatial domain, eliminating problem-dependent decomposition.

First, the mesh system is partitioned by the graph-partitioner, METIS [13], with the default (unweighted vertices) setting so that each processor contains approximately the same number of cells. Even decomposition of mesh system is suitable for applications for which execution time of tasks are expected to be a function of number of cells. Typical applications are flow solvers involving iterative solution of the same discretized governing equations on all cells.

Each processor reads separate mesh files generated by METIS and stores local mesh blocks (which are a subset of a mesh). For clarity, local data stored in a processor can have different traits than their counterparts in other processors, whereas global data identical in every processor are obtained by combining local data.

In order to organize mesh blocks spatially, a global octree is constructed to divide domain into eight non-overlapping bins (subsets) recursively. The preference of octree over other types of trees such as binary or quad trees is natural for three-dimensional geometries. Each bin in the octree is axis-aligned as shown in Figure 1. Dimensions of the global octree are obtained by all-gathering and maximizing the dimensions of the axis-aligned bounding box (AABB) of local mesh blocks.

Figure 1. Octree space partitioning: (a) computational domain in the process of adaptive spatial partitioning; (b) slice view shows refinements in the most loaded regions; (c) close up view of the slice.

Processors register mesh cells in local mesh blocks to bins of octree based on the intersection of AABB of the mesh cell and the bin. Next, the load in each bin is calculated by summing up the number of cells. Bin loads are all-reduced with addition. The bins of octree are input to METIS as graph-vertices. The load of each bin corresponds to the weight of each graph-vertex. The connectivity of bins are also provided to METIS in order to obtain a connected graph, which is needed for contiguous partitions. The METIS_PartGraphKway function of METIS computes a k -way partitioning where k is the number of processors. If no partitioning with less than 10% imbalance is found, the most loaded bin of octree is refined. Once a satisfactory partitioning is obtained after all refinements, each partition (containing multiple bins) are distributed to respective processor. Finally, the mesh blocks corresponding to the same parent meshes in partitions are merged in order to reduce intra-processor communication. Note that ParMETIS, which is the parallel version of METIS, cannot be used at this stage since distribution of octree bins to processors is unknown.

With spatial partitioning, the spatial domain is decomposed into regions, and each region is assigned to a specific processor. As the mesh system is non-stationary, different mesh cells arrive to and leave the regions the processors are assigned with. On the other hand, the traditional approach is to assign certain mesh cells to processors regardless of the motion of the mesh system. One of the merits of the traditional approach is that cell-to-cell connectivity does not need to be maintained since there is no disconnection/reconnection of moving cells. In addition, no time is spent for re-organization of mesh system spatially. However, the overlapping mesh blocks are still required to be brought together regardless of the approach followed. The traditional approach solves this problem by transferring overlapping mesh blocks and sending processed information back to the host processor. The downside of this approach is that the communication has to be repeated twice in every time step. In this study, time is saved by eliminating the need for sending processed information back to the host processor since overlapping mesh blocks reside in the same partitions.

3. Donor Search

The donor search categorizes all mesh cells into certain types to be fed to the flow solver which invokes different algorithms based on the type of cells. The definitions of cell types are listed below.

- Field or computational cell on which governing equations are solved.
- Receptor cell on which governing equations are not solved but flow variables are interpolated from a (donor) field cell.
- Mandatory receptor cell which is located next to an intergrid boundary submerged inside another mesh. Since the boundary condition cannot be applied intrinsically, it is mandatory to interpolate flow variables.
- Hole cell on which neither governing equations are solved nor flow variables interpolated.
- Orphan cell is a mandatory receptor cell whose donor cell is also a receptor cell. It can be avoided by converting the type of donor cell from receptor to field cell.

Figure 2 shows several cell types of structured Cartesian and unstructured overset mesh after donor search.

Figure 2. A structured Cartesian mesh (left) and an unstructured overset mesh (right) after donor search. Gray: field cell; orange: hole cell; light blue: receptor; blue: mandatory receptor.

Since meshes in a mesh system have no face-to-face connectivity with each other, flow variables are interpolated across intergrid cells of different mesh blocks. Regardless of the order of interpolation, interface fluxes will not be conserved in the whole mesh system due to non-conforming faces of intergrid cells. Conservation of fluxes can be maintained by regenerating mesh in intergrid region so that intergrid faces of mesh blocks conforms perfectly [14,15] or by using auxiliary mesh in the intergrid region [16]. In this work, variables are interpolated non-conservatively.

4. Hole Cutting

The hole cutting task is for determining hole cells. One of the hole cutting methods involves usage of a hole map, which is a Cartesian mesh used to represent holes approximately. When hole maps are used in conjunction with the core donor search algorithm (explained in the next section), hole cells are determined exactly. There are exact methods which are not required to be used in conjunction with donor search. Direct cutter, for example, cuts a hole profile on overlapping mesh and determines interior hole cells with flood algorithm. However, direct cutting is found to have robustness issues with three-dimensional geometries.

Hole mapping requires all wall and outer boundaries to be present in each processor. Therefore, local wall and outer boundaries are all-gathered. A Cartesian mesh which initially has one sub-block is made to fit around wall boundaries. If any sub-block of the Cartesian mesh intersects both wall and outer boundaries, the Cartesian mesh is refined by doubling each dimension. That is, the initial $2 \times 2 \times 2$ Cartesian mesh becomes $4 \times 4 \times 4$ and so on. Once no sub-block is left to intersect both wall and outer boundaries, the sub-blocks which intersect wall boundaries serve as an approximate hole profile. Next, flood algorithm identifies sub-blocks representing the volume of hole. The procedure is identical to that in [11], where implementation details can be found.

If the initial AABB of wall boundary does not intersect the outer boundary, there is no refinement required, and the AABB of wall boundaries is sufficient to represent the hole inside wall boundaries.

5. Core Algorithm

In the first time step, the donor search is performed with alternating digital tree (ADT) [17], which is a space partitioning binary tree. ADT tree splits the spatial domain into halves with axis-aligned cutting planes, recursively. The axis of cutting plane alternates in cyclic order in accordance with the levels in ADT. In three-dimensional space, there are three cutting planes with cutting axes parallel to the x -, y -, and z -axis. The position of the cutting plane along the cutting axis depends on the number of dimensions required to define an object to be inserted. For example, if a point (0D) is to be inserted to an ADT, positions of cutting planes would be (x, y, z) coordinates of the point along the respective cutting axes. Moreover, if the inserted object is a rectangle, the positions of cutting planes would be the six $(x_{min}, y_{min}, z_{min}, x_{max}, y_{max}, z_{max})$ dimensions of the rectangle in the respective orientations of the cutting plane. In general, objects to be inserted to ADT are n -orthotopes or hyperrectangles. In the example above, point and rectangle are 1- and 2-orthotopes. In this work, the objects are AABB, which is 3-orthotope, of mesh cells. Therefore, both the domain and objects to be inserted are six-dimensional.

For visualization of the insertion operation into ADT, Figure 3 demonstrates the insertion of points into ADT in two-dimensional space. On the root level, point A is the median in the x -axis, and it is associated with the whole domain, Ω_A . Domain Ω_A is divided by a plane with cutting axis parallel to the x -axis at point A. Next, the cutting axis is changed to be the y -axis. Point B, which is one of the points in the left half of Ω_A and is the median in the y -axis, is assigned to Ω_B , which is the left half of Ω_A . Point C is inserted similar to Point B except that it belongs to the right half of Ω_A . The rest of the points are inserted into sub-branches of the tree in the same fashion.

Figure 3. Insertion of points into a 2D ADT. Black and white filled circles indicate partitioning of domain in the x - and y -axis.

The search operation on ADT is similar to insertion. Given a query point, AABBs which overlap the query point are returned. If the query point is outside the hypercube of the ADT, the search operation returns an empty array, meaning that none of AABBs in the tree overlaps the query point. Assuming that the query point is inside the hypercube of the tree, the search operates recursively by always starting from the root node. First, it is tested whether the query point is contained by the AABB in the current node. If so, the AABB is added to the array which is to be returned after the recursive search operation is completed. Regardless of containment of the query point by the AABB, the query point is redirected to one or both halves of the current node by testing the dimension of the query point against the cutting axis associated with the current level.

that the geometric shape of mesh cells are polyhedra which are not axis-aligned in an unstructured grid. However, ADT accepts only axis-aligned objects to be inserted. Therefore, given a mesh block, M_i , AABB and associated ID of each mesh cell are inserted into the respective ADT T_i . Once an ADT tree is constructed for every mesh block, the candidate donor cell for each mesh cell in M_i is searched out by providing the centroid (query point) of the cell as input to all trees T_j where $j \neq i$. As the output of search operation, each tree T_j returns an array of IDs corresponding to the mesh cells in M_j . The query point is tested for containment by each polyhedron corresponding to the returned ID. The mesh cell which contains the query point is added to the list of candidate donors. The procedure is shown in Algorithm 1.

Algorithm 1: Donor search using ADT

```

foreach Mesh-block  $M_i$  do
  foreach Tree  $T_j$  where  $j \neq i$  do
    foreach Mesh-cell  $c_i$  in  $M_i$  do
      Query-point  $qp$  = centroid of  $c_i$ ;
      array[ $ID_1, \dots, ID_n$ ] =  $T_j$ .search( $qp$ );
      foreach ID in array do
        if  $M_j$ .cell(ID).polyhedron contains  $qp$  then
          Add  $M_j$ .cell(ID) to the list of candidate donors of  $c_i$ ;

```

In the first time step, ADT is used for the donor search. In subsequent time steps, the donor search is performed with a stencil walk algorithm if the cell had a donor in the previous time step; otherwise, ADT is used as in the first time step. Referring to Figure 4, the stencil walk algorithm starts with choosing a starting or seed cell (shaded) and walking toward the query point (marked with a cross) by using cell-to-cell connectivity. First, a search line is drawn between the centroid of the seed cell and the query point. The search line is tested for intersection by the faces of the seed cell. The current cell (which was

initially the seed cell) is updated to be the neighbor cell which shares the intersected face. When the current cell is near a boundary and, therefore, has no neighbor which shares the intersected face, all other faces of that boundary are tested for intersection by the search line. If an intersected boundary face is found, the current cell is updated to be the interior cell of that boundary. If no other boundary face is found, then the algorithm stops indicating that no cell containing the query point is found. When none of the faces of the current cell intersects the search line, the current cell is added to the list of candidate donor cells.

Figure 4. The stencil walk algorithm starts from the seed cell (shaded cell) and, using cell-to-cell connectivity, walks toward the cell which contains the target (shown as a cross in a circle).

The reason for using the stencil walk algorithm in subsequent time steps is the absence of seed cells in the first time step. For receptors, the seeds are chosen to be the centroid of donor cells. For other cell types, seeds are not available and, therefore, as mentioned, ADT is used for the donor search. Since the stencil walk algorithm involves costly geometric intersections tests, a high number of walks is avoided by not picking random seed cells and using ADT whenever proper seed cells are not available.

After the donor search, if no candidate donor cell is found for a mesh cell, the query point is either outside of M_j or inside a hole (if one exists) in M_j . If M_j has a hole, and if the query point is inside any of the bins of hole map of M_j , then the mesh cell is labeled as hole cell. Once a mesh cell is labeled as a hole cell, its type cannot be changed regardless of interaction of the mesh cell with other mesh blocks. Otherwise, if the mesh cell is not a hole cell, it is labeled as field cell.

If a candidate donor cell is found and the volume of the query cell is larger than that of the candidate donor cell, then the mesh cell is labeled as receptor cell. A receptor cell may have multiple candidate donor cells. In this case, the candidate donor cell which has the smallest volume is assigned as the donor of the mesh cell. If the mesh cell has a smaller volume than a candidate donor cell, the mesh cell is labeled as a field cell. The reason for choosing the cell with smaller volume is to represent the domain with the highest resolution possible. Another option is to choose the cell closest to respective boundary. This kind of selection is beneficial in preserving boundary layer cells.

The cells which have a hole neighbor or are located next to an intergrid boundary are labeled as mandatory receptor cells. These receptor cells serve as intergrid boundary conditions.

The volume of intersection between mesh blocks can be reduced in order to save time by avoiding redundant interpolation of variables from donors to receptors. Minimization of the overlapped region is performed by excluding receptors having no field neighbor from the flow solution.

Once the donor search is completed, if the assembler and the solver have different partitioning layouts, the assembler maps cell types and donor IDs to the layout of the solver. In the case of sharing of the same partitioning layout, no further operation is required.

6. Governing Equations

The integral form of Euler equations is

$$\frac{\partial}{\partial t} \int_V \mathbf{U} dV + \int_A \mathbf{F} dA = 0 \quad (1)$$

where V and A are the volume and area of a cell and $\mathbf{U}$ and $\mathbf{F}$ are the vectors of conserved variables and interface fluxes:

$$\mathbf{U} = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ E \end{bmatrix}, \quad \mathbf{F} = \begin{bmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ \rho uw \\ u(E + p) \end{bmatrix} \quad (2)$$

u is the velocity component in face-normal direction, v and w are tangential velocity components, E is total energy, and p is static pressure. Euler equations are used in x-split form [18] and, therefore, velocity components are aligned in Cartesian coordinates such that u , v , and w are parallel to the x -, y -, and z -axis.

For non-Cartesian unstructured meshes, to calculate the interface flux, conserved variables are rotated to face-normal coordinate system with transformation matrix:

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & \cos \theta^{(y)} \cos \theta^{(z)} & \cos \theta^{(y)} \sin \theta^{(z)} & \sin \theta^{(y)} & 0 \\ 0 & -\sin \theta^{(z)} & \cos \theta^{(z)} & 0 & 0 \\ 0 & -\sin \theta^{(y)} \cos \theta^{(z)} & -\sin \theta^{(y)} \sin \theta^{(z)} & \cos \theta^{(y)} & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad (3)$$

where $\theta^{(y)}$ and $\theta^{(z)}$ are found from components of face-normal $[n_x, n_y, n_z]^T$:

$$\begin{aligned} n_x &= \cos \theta^{(y)} \cos \theta^{(z)} \\ n_y &= \cos \theta^{(y)} \sin \theta^{(z)} \\ n_z &= \sin \theta^{(y)} \end{aligned} \quad (4)$$

The Riemann problem is approximately solved at each cell interface with HLLC solver [18] in order to compute interface flux. When providing left and right states to the Riemann solver, the interface velocities are corrected by subtracting the rotated mesh velocity $[b_x, b_y, b_z]^T$ from the rotated velocity such that

$$\begin{aligned} u &= u - b_x \\ v &= v - b_y \\ w &= w - b_z \end{aligned} \quad (5)$$

Discretized equations are solved with dual time stepping approach with

$$\frac{V}{\Delta \tau} (\mathbf{U}^{s+1} - \mathbf{U}^s) = - \sum_f^{\text{nface}} \mathbf{F}(\mathbf{U}^s) A_f - \frac{V}{2\Delta t} (3\mathbf{U}^s - 4\mathbf{U}^n + \mathbf{U}^{n-1}) \quad (6)$$

where superscripts s and n denote states in pseudo and real times. The parameter Δt is real time step and $\Delta \tau$ is pseudo time step calculated from

$$\Delta \tau = \text{CFL} \frac{V}{\sum_f^{\text{inface}} \max(|\lambda|) A} \quad (7)$$

where $\max(|\lambda|)$ is the maximum absolute eigenvalue among all faces of a cell. Equation (6) is solved until steady state for maximum 100 pseudo time steps. When the left hand side of the equation vanishes, the original unsteady equation recovers. Interface flux $\mathbf{F}$ is evaluated explicitly as a function of conservative variable vector in previous pseudo time step, that is $\mathbf{U}^s$.

7. CAD Models and Meshes

The assembler and solver are tested on the generic helicopter configuration [12] consisting of a fuselage, pylon, and four NACA0012-profiled rotor blades.

In [12], several collective pitch angles were tested, whereas in this simulation only 4° of collective pitch is considered. Together with 8° of linear twist, the angle of attack ranges from 8° at the tip of a blade to 12° at the root cut. A sketch of a blade is shown in Figure 5.

Figure 5. Sketch of a rotor blade in generic helicopter configuration. aoa is abbreviation for wind angle of attack.

The shapes of the fuselage and pylon are defined with super-ellipse equations:

$$\begin{bmatrix} H(x/l) \\ W(x/l) \\ Z_0(x/l) \\ N(x/l) \end{bmatrix} = C_6 + C_7 \left(C_1 + C_2 \left| \frac{x/l + C_3}{C_4} \right|^{C_5} \right)^{1/C_8}. \quad (8)$$

x/l is the non-dimensional longitudinal coordinate with limits $[0, 1.997]$ for fuselage and $[0.400, 1.018]$ for pylon and l is the desired length of fuselage which is 78.57 inches. Dimensions given in imperial units in [12] were not converted to metric units to avoid ambiguity. Coefficients (C_1 to C_8) are shown in Table A1 in appendices. Cross-sectional coordinates y/l and z/l are obtained with

$$\begin{aligned} y/l &= r \sin \phi \\ z/l &= r \cos \phi + Z_0 \end{aligned}$$

where ϕ ranges from 0 to 2π and r is defined as

$$r = \left[\frac{\left(\frac{HW}{4} \right)^N}{\left| \frac{H}{2} \sin \phi \right|^N + \left| \frac{W}{2} \cos \phi \right|^N} \right]^{1/N}. \quad (9)$$

Equation (8) and the coefficients provided in [12] are modified since their original counterparts caused problems such as division by zero. Equations (8) and (9) are different

from the counterparts by modulus operator around $\frac{x/l+C_3}{C_4}$, and $\frac{H}{2} \sin \phi$ and $\frac{W}{2} \cos \phi$. The modified coefficients are encircled in Tables A1 and A2.

Figure 6 shows CAD models of fuselage, pylon, four rotor blades, and hub modeled with FreeCAD [19]. Leading and trailing faces of fuselage are blunted slightly in order to have satisfactory mesh quality. The hub is modeled as a cylinder with diameter 0.1 m and length 0.05 m, and extensions which connect the hub to the blades have the same profile and angle of attack with the roots of respective blades. The extensions span from the hub such that they overlap 5% of respective blades.

Figure 6. CAD model of generic helicopter configuration.

CAD models are transferred to the mesh generator, GMSH [20], to generate tetrahedral mesh for each component. Rotor blades have cylindrical meshes with diameter $10 \times \text{chord}$ and length $20 \times \text{chord}$. Fuselage has a spherical mesh with diameter of 20 inches enclosing all blade and hub meshes. The hub also has a spherical mesh with diameter scaling with root cutout length such that $1.5 \times 0.24R$. Figure 7 shows cross-sections of the fuselage, one of the blades, and the hub. Element size s on fuselage, blade, and hub surfaces are 3, 1, and 2 mm. Volumetric element size is calculated with $s + d/5$ for all components where d is the shortest distance from respective wall surface. Table 1 shows the number of cells in each mesh.

Figure 8 shows a slice of field cells after donor search. Each color corresponds to a different mesh. In the figure, there are cells that are undesirably larger than the surrounding cells such the one enclosed in a yellow circle. These cells were receptors having candidate donors which themselves were mandatory receptors, or by definition, they were orphans. Since they have to be the donors of mandatory receptors, they are converted to field cells.

Figure 9a shows receptors of a slice of fuselage mesh. It is clear that the receptors which only neighbor other receptors do not contribute to flow solution but increase simulation run time by interpolating flow variables across donors. Figure 9b shows the same mesh after exclusion of redundant receptors.

Table 1. Number of cells in each mesh.

Fuselage and Pylon	1,645,011
Blade 1	1,066,344
Blade 2	1,067,491
Blade 3	1,067,878
Blade 4	1,066,421
Hub	510,924
Total	5,356,578

Figure 7. Cross-sections of component meshes: (a) fuselage; (b) closer view of fuselage; (c) rotor blade; and (d) hub and extensions.

Figure 8. A slice of field cells after donor search. Each mesh is shown with a different color. The cell enclosed in a yellow circle is an orphan converted to a field cell.

Figure 9. Slice of fuselage mesh before and after overlap minimization: (a) before; (b) after.

8. Results

Test cases are run on TRUBA's [21] clusters. In order to reduce queue time, computations are run on different set of nodes with different specifications, as shown in Table 2. In this work, hyperthreading is not used and virtual processors are mapped to cores; therefore, processor is an alias for core for the rest of the paper. Since a node can have a maximum of 20 cores, for a job requesting more than 20 processors, cores cannot fit into a single node but they will belong to different nodes. Therefore, data transfer takes place between cores having both shared memory and distributed memories. Open MPI's byte transfer layer (BTL) framework uses the best component such as shared memory, TCP, Infiniband, and so on for data transfer between cores. Again, in order to reduce queue time, the number of nodes is left to be determined by the resource manager. The results in this section are obtained by using maximum allowable number of processors per user (128) usually distributed over 7–10 nodes.

Table 2. Hardware specifications.

Group	Processor Model	Core/Node	RAM (GB)
Barbun	Intel Xeon Scalable 6148	20	384
Sardalya	Intel Xeon E5-2690 V4	14	256
Levrek	Intel Xeon i5-2690	16	256

The most intensive memory usage is observed during data exchange between processors. Exchange of spatial partitions during load (re)balance and exchange of mesh cells after mesh motion are two major tasks involving intensive MPI communication. Although it is verified by Valgrind [22] that no memory leak occurs, memory usage increases after the mentioned tasks. It is expected that the reason for increasing memory usage is memory fragmentation caused by frequent allocation/deallocation of buffers leaving fragmented chunks of memory space. An obvious solution is using fixed buffers for data exchange. However, after load rebalance partitioning layout changes, therefore, the buffers used for data exchange have to be deallocated. Once memory usage hits the memory limit (8 GB per core), the simulation is halted by resource manager. In order not to lose progress, crucial data structures of assembler and solver are serialized with Boost's serialization library (which is also used by Boost MPI for data transfer), and written to binary archives for every quarter (90°) rotation. Processors read deserialized data from respective archives and restart the simulation with reset memory space.

The most time consuming tasks per time step are shown in Figure 10. These tasks remain as the most time consuming in all time steps. However, note that Rebalance may not be needed for all time steps. The execution time of Solve is based on the maximum number of pseudo time steps (100) to solve Euler equations. Since the assembler constitutes less time compared to the solver, it is unnecessary to optimize load balance for the assembler. As discussed in Section 2, it is possible to have separate partitions for the assembler and the flow solver in order to optimize load balance differently. However, having two separate partitions requires mapping of data across different partitioning layouts. The case with two different partitions is also tested, and it is found that mapping of data increases the total run time of simulation.

Figure 10. The most time consuming tasks in a time step. The task durations belong to the slowest processor.

The frequency of load rebalance depends on relative time-cost of load rebalancing and the solver. In the case of 100 pseudo time steps for the solver, rebalancing is more costly than the solver and, therefore, Rebalance is called when the ratio of the maximum to the minimum number of cells across processors exceeds 1000. The rebalance threshold is not the optimum and can be improved by an algorithm involving relative costs of tasks. Figure 11 shows the run time per time step with and without rebalance. Areas under each curve represent total run time over quarter rotation of rotor. The time saved with load rebalancing is 13%, which adds up periodically for every quarter rotation due to the fact that the initial spatial partitioning is restored for every quarter rotation.

The downside of the proposed load balancing method is that the whole load balancing procedure (octree construction and adaptive refinement) should be repeated from scratch. One of the main reasons of high cost of load rebalancing is the high number of bins resulting from the refinement of octree. As explained in Section 2, bins are merged after refinement of octree in order to reduce communication among bins. Time spent on merging of octree bins is proportional to the number of octree bins. In order to reduce load imbalance across processors below 10%, 344 refinement steps are executed, as shown in Figure 12 resulting in 1040 bins. Merging all 1040 bins into 128 (the number of processors) bins causes the load balancing to be the most time consuming task. Frequency of load rebalancing is also related to the number of merged octree bins. The higher the number of merged octree bins, the more is the number of mesh cells that move across octree bins after mesh motion in every time step.

Figure 11. Run time of simulation in each time step with and without load rebalance. Area under each curve represents total run time over 90° rotation of rotor.

Figure 12. Reduction of load imbalance with increasing number of octree refinements. Dashed line indicates the threshold under which refinement stops.

The tasks **Exchange** and **Reconnect** are for exchange of mesh cells after mesh motion and reconnection of arrival mesh cells to the local mesh blocks. **Exchange** has to be called in every time step in order to keep mesh cells in correct partitions/processors even though load rebalancing is to be executed.

Figure 13 shows the total run time and speed-up for different number of processors. Speed-up is calculated by dividing run time by the run time for 64 processors. One of the reasons for speed-up, as expected, is distribution of computational load among more computational resources. Another reason for speed-up, in the particular case of partially shared memory computing, may be the increase in shared memory. Resource manager allocates 8 GB memory per core. Although memory is provided per core basis, cores are free to read/write any portion of cumulative shared memory space assigned to a user.

The number of ‘walks’ in the stencil walk algorithm ranged from 3 to more than 1000. Whenever the number of walks exceeded 1000, the donor search algorithm switched from stencil walking to ADT. Usage of the stencil walk algorithm saves 5% percent of the time of donor search with only using ADT in every time step. However, in problems where the time step is sufficiently high and current donor cells are sufficiently far away from the seeds, the high number of cell-to-cell walks will cause the stencil walk algorithm to perform slower than ADT.

Figure 13. Run time and speed-up over 90° of rotor rotation.

Although some of the tasks (e.g., donor search) are common in research papers pertaining to overset mesh methodology [2,6,11], due to the lack of exact definitions and implementation details of tasks, comparisons of parallel performance parameters can be misleading even if better results are obtained. In terms of partitioning method, this paper is an improvement over Suggar++ [6] by usage of adaptive refinement instead of predetermined volumes for spatial partitioning and, therefore, it is useful to compare parallel performance results of Suggar++ with the results obtained in this paper. In [6], the wall clock time from the combination of tasks that are hole cut, donor search, and overlap minimization and respective speed-up is presented for the case of helicopter fuselage and blades for up to 8 processors. Since problem sizes (the number of cells) are different, it is appropriate to compare speed-up results only. Figure 14 shows speed-up for the mentioned tasks. The main inhibitor of speed-up is hole cutting task as it requires inter-processor communication for AABB of mesh walls. However, for up to 8 processors, the communication cost across processors is not substantial. Donor search and overlap minimization are completely local tasks, meaning that there is no inter-processor communication required. Therefore, speed-up for these three tasks remains nearly linear for up to 8 processors. Note that in this work, hole cutting is an integral part of donor search opposed to Suggar++'s direct hole cutting which is a standalone task. As implementation details of Suggar++ are unknown, it is unclear why Suggar++ shows such non-linear behavior. It should also be noted that spatial partitioning is a memory intensive operation. Since memory cannot explicitly be requested and is provided per core basis in this work,

some of the modules such as the flow solver have to be deactivated in order to obtain performance results for 8 and less number processors.

Figure 14. Speed-up for combination of tasks hole cutting, donor search, and overlap minimization.

9. Conclusions

This paper presents a parallel overset grid assembler and flow solver tested on a generic helicopter configuration. Both modules were made to share the same partitioning layout as the solver was found to be more time consuming than the assembler by a large margin, and therefore, load balancing based on the assembler was redundant. An adaptive octree was used to maintain load balance. The most loaded bin of octree was refined in each refinement step and optimal load distribution was computed with weighted graph partitioner. Even though load (re)balancing was found to be the most time consuming task, it was shown that frequent load balancing reduces simulation time considerably. The main reason for the high cost of load balancing was the high number of octree bins resulting from refinement of octree in order to reduce imbalance under a threshold. Usage of stencil walk algorithm saves 5% percent of time of donor search compared to alternating digital tree (ADT) in every time step.

Author Contributions: Conceptualization, methodology, software, validation, formal analysis, investigation, resources, data curation, writing—original draft, writing—review and editing, visualization, O.S.; Writing—review and editing, supervision, I.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Reported data can be generated with the code provided in <https://github.com/orxshi/tailor> (accessed on 29 May 2021).

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A. Coefficients in Super-Ellipse Equations

Table A1. Coefficients for the shape of fuselage.

$0.0 < x/l < 0.4$								
	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8
H	1.0	−1.0	−0.4	0.4	1.8	0.0	0.25	1.8
W	1.0	−1.0	−0.4	0.4	2.0	0.0	0.25	2.0
Z_0	1.0	−1.0	−0.4	0.4	1.8	−0.08	0.08	1.8
N	2.0	3.0	0.0	0.4	1.0	0.0	1.0	1.0
$0.4 \leq x/l < 0.8$								
	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8
H	0.0	0.0	0.0	0.0	0.0	0.25	0.0	0.0
W	0.0	0.0	0.0	0.0	0.0	0.25	0.0	0.0
Z_0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
N	0.0	0.0	0.0	0.0	0.0	5.0	0.0	0.0
$0.8 \leq x/l < 1.9$								
	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8
H	1.0	−1.0	−0.8	1.1	1.5	0.05	0.2	0.6
W	1.0	−1.0	−0.8	1.1	1.5	0.05	0.2	0.6
Z_0	1.0	−1.0	−0.8	1.1	1.5	0.04	−0.04	0.6
N	5.0	−3.0	−0.8	1.1	1.0	0.0	1.0	1.0
$1.9 \leq x/l < 2.0$								
	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8
H	1.0	−1.0	−1.9	0.1	2.0	0.0	0.05	2.0
W	1.0	−1.0	−1.9	0.1	2.0	0.0	0.05	2.0
Z_0	0.0	0.0	0.0	0.0	0.0	0.04	0.0	0.0
N	0.0	0.0	0.0	0.0	0.0	2.0	0.0	0.0

Table A2. Coefficients for the shape of pylon.

$0.4 < x/l < 0.8$								
	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8
H	1.0	−1.0	−0.8	0.4	3.0	0.0	0.145	3.0
W	1.0	−1.0	−0.8	0.4	3.0	0.0	0.166	3.0
Z_0	0.0	0.0	0.0	0.0	0.0	0.125	0.0	0.0
N	0.0	0.0	0.0	0.0	0.0	5.0	0.0	0.0
$0.8 \leq x/l < 1.018$								
	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8
H	1.0	−1.0	−0.8	0.218	2.0	0.0	0.145	2.0
W	1.0	−1.0	−0.8	0.218	2.0	0.0	0.166	2.0
Z_0	1.0	−1.0	−0.8	1.1	1.5	0.065	0.06	0.6
N	0.0	0.0	0.0	0.0	0.0	5.0	0.0	0.0

References

1. van der Wall, B.G.; Burley, C.L.; Yu, Y.; Richard, H.; Pengel, K.; Beaumier, P. The HART II test—Measurement of helicopter rotor wakes. *Aerosp. Sci. Technol.* **2004**, *8*, 273–284. [\[CrossRef\]](#)
2. Horne, W.J.; Mahesh, K. A massively-parallel, unstructured overset method for mesh connectivity. *J. Comput. Phys.* **2019**, *376*, 585–596. [\[CrossRef\]](#)
3. Landmann, B.; Montagnac, M. A highly automated parallel Chimera method for overset grids based on the implicit hole cutting technique. *Int. J. Numer. Methods Fluids* **2011**, *66*, 778–804. [\[CrossRef\]](#)
4. Buning, P.G.; Pulliam, T.H. Cartesian off-body grid adaption for viscous time-accurate flow simulations. In Proceedings of the 20th AIAA Computational Fluid Dynamics Conference, Honolulu, HI, USA, 27–30 June 2011; pp. 27–30. [\[CrossRef\]](#)
5. Sitaraman, J.; Floros, M.; Wissink, A.; Potsdam, M. Parallel domain connectivity algorithm for unsteady flow computations using overlapping and adaptive grids. *J. Comput. Phys.* **2010**, *229*, 4703–4723. [\[CrossRef\]](#)
6. Noack, R.W.; Boger, D.A.; Kunz, R.F.; Carrica, P.M. Suggar++: An improved general overset grid assembly capability. In Proceedings of the 47th AIAA Aerospace Science and Exhibit, Orlando, FL, USA, 5–9 January 2009; pp. 22–25. [\[CrossRef\]](#)
7. Plimpton, S.J.; Hendrickson, B.; Stewart, J.R. A parallel rendezvous algorithm for interpolation between multiple grids. *J. Parallel Distrib. Comput.* **2004**, *64*, 266–276. [\[CrossRef\]](#)
8. Hahn, S.; Iaccarino, G.; Ananthan, S.; Baeder, J.D. *Extension of CHIMPS for Unstructured Overset Simulation and Higher-Order Interpolation*; Center for Turbulence Research Annual Research Briefs: Stanford, CA, USA, 2009. [\[CrossRef\]](#)
9. Rogers, S.E.; Suhs, N.E.; Dietz, W.E. PEGASUS 5: An Automated Preprocessor for Overset-Grid Computational Fluid Dynamics. *AIAA J.* **2003**, *41*, 1037–1045. [\[CrossRef\]](#)
10. Jeanfaivre, C.; Juvigny, X.; Benoit, C. Parallel Chimera computations of helicopter flows. In *ONERA: Tire a Part*; Information Scientifique Et Technique: Yokohama, Japan, 2004; pp. 1–8.
11. Roget, B.; Sitaraman, J. Robust and efficient overset grid assembly for partitioned unstructured meshes. *J. Comput. Phys.* **2014**, *260*, 1–24. [\[CrossRef\]](#)
12. Mineck, R.E. *Steady and Periodic Pressure Measurements on a Generic Helicopter Fuselage Model in the Presence of a Rotor*; NASA Langley Research Center: Hampton, VA, USA, 2000.
13. Karypis, G.; Kumar, V. *METIS: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 5.0*; Citeseer: Princeton, NJ, USA, 1995.
14. Chand, K. Component-based hybrid mesh generation. *Int. J. Numer. Methods Eng.* **2005**, *62*, 747–773. [\[CrossRef\]](#)
15. Jung, M.S.; Kwon, O.J. A conservative overset mesh scheme via intergrid boundary reconnection on unstructured meshes. In Proceedings of the 19th AIAA Computational Fluid Dynamics, San Antonio, TX, USA, 22–25 June 2009; p. 3536.
16. Rinaldi, E.; Colonna, P.; Pecnik, R. Flux-conserving treatment of non-conformal interfaces for finite-volume discretization of conservation laws. *Comput. Fluids* **2015**, *120*, 126–139. [\[CrossRef\]](#)
17. Bonet, J.; Peraire, J. An alternating digital tree (ADT) algorithm for 3D geometric searching and intersection problems. *Int. J. Numer. Methods Eng.* **1991**, *31*, 1–17. [\[CrossRef\]](#)
18. Toro, E.F. *Riemann Solvers and Numerical Methods for Fluid Dynamics: A Practical Introduction*; Springer Science & Business Media: Berlin, Germany, 2013.
19. Riegel, J.; Mayer, W.; van Havre, Y. FreeCAD (Version 0.18.4). Available online: <http://www.freecadweb.org> (accessed on 29 May 2021).
20. Geuzaine, C.; Remacle, J.F. Gmsh: A 3-D finite element mesh generator with built-in pre-and post-processing facilities. *Int. J. Numer. Methods Eng.* **2009**, *79*, 1309–1331. [\[CrossRef\]](#)
21. TRUBA. Available online: <https://www.truba.gov.tr/index.php/en/main-page/> (accessed on 29 May 2021).
22. Nethercote, N.; Seward, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. *ACM Sigplan Not.* **2007**, *42*, 89–100. [\[CrossRef\]](#)