Parallel Semi–Meshless Stencil Selection for Moving Geometry Simulations

J. Angulo*, D. J. Kennett†, S. Timme‡ and K. J. Badcock§

University of Liverpool, Liverpool, England L69 3BX, United Kingdom

Computational fluid dynamics methods to simulate flows around geometries in relative motion are important for the aerospace industry. A meshless method to select stencils from overlapping and moving point distributions, and a corresponding flow solver capable of solving the Euler and Navier-Stokes equations on those stencils, have been developed previously. In order for these methods to be useful in real world applications, their computational efficiency needs to be addressed. This paper extends the meshless method from previous work in an effort to increase the computational efficiency by means of parallel programming. The flow solver and the original meshless stencil selection method are presented. Then, the developed parallel algorithm is described in detail and results are shown for aerofoils in two dimensions, and for a fighter aircraft configuration as well as a transient store-release case in three dimensions.

I. Introduction

A demanding task in computational fluid dynamics (CFD) applied to complex geometry is mesh generation. Conventional mesh generation techniques become difficult to apply, or are not applicable, when used to calculate flows over bodies in relative motion. Several techniques have been developed in the last few years to tackle the simulation of flow around bodies with parts in relative motion. One such technique is the overset or Chimera method.[1] Chimera is most often associated with finite volume/difference schemes, and its functionality is based on overlapping different grids belonging to each body or moving part. First, the method cuts holes in the background grids to accommodate the overlap, then inter-grid stencils are created, which provide the communication of flow information between grids.

An alternative technique is the meshless method,[2,3] which discretises the domain by using a set of points. Each point in the domain has a sub-domain of neighbouring points, called a stencil or cloud. These clouds are then used to calculate the spatial derivatives in the partial differential equations to be solved. The meshless method is attractive for moving-body problems, as points are allowed to move independent of one another during a time-dependent simulation.[4] The method described in this work is referred to as semi-meshless because it takes overlapping point distributions from underlying component meshes and uses their original connectivity as a guide to select the appropriate stencils. This method can be divided into two stages: a preprocessing stage that selects the required stencils, and a flow solver that uses these stencils to perform the CFD simulation. Serial versions of the flow solver and the stencil selection tool (called preprocessor from this point on) have been developed and described in Refs. 4 and 5, and a parallel version of the flow solver has been presented in Ref. 6. For the preprocessor to be competitive with established CFD techniques, the computational efficiency needs to be improved.

In this work, an automatic, scalable parallel preprocessing tool for the selection of stencils and a corresponding flow solver are described. The meshless flow solver used to solve the Euler equations is summarised in Section II and its parallel implementation in Section III. The preprocessor method is presented in Section IV, while its parallel implementation is discussed in Section V. Finally, results that demonstrate the capabilities of the method are shown in Section VI.

*PhD Candidate, School of Engineering, J.Angulo@liverpool.ac.uk.

†Research Associate, School of Engineering, D.Kennett@liverpool.ac.uk.

‡Lecturer, School of Engineering, Sebastian.Timme@liverpool.ac.uk, Member AIAA

§Professor, School of Engineering, K.J.Badcock@liverpool.ac.uk, Senior Member AIAA

II. Meshless Method

The Euler equations governing inviscid fluid flow can be written in conservative form as

$$\frac{\partial \mathbf{w}}{\partial t} + \frac{\partial \mathbf{f}}{\partial x} + \frac{\partial \mathbf{g}}{\partial y} + \frac{\partial \mathbf{h}}{\partial z} = 0 \tag{1}$$

where $\mathbf{w}$ is the vector of conservative variables to be determined, and $\mathbf{f}$, $\mathbf{g}$ and $\mathbf{h}$ are the inviscid flux vectors in the x, y and z directions respectively.

The meshless method used in this work evaluates the derivatives of a function ϕ at each point i by interpolating scattered data from the points contained in the associated stencil of the point i (called the star point from now on). This can be written as

$$\frac{\partial \phi_i}{\partial x} = \sum_{j=0}^{n_i} a_j \phi_j, \quad \frac{\partial \phi_i}{\partial y} = \sum_{j=0}^{n_i} b_j \phi_j, \quad \frac{\partial \phi_i}{\partial z} = \sum_{j=0}^{n_i} c_j \phi_j \tag{2}$$

where i denotes the star point, j represents each of the points in the stencil, n_i is the number of points in the stencil with $j = 0$ being the star point and a_j, b_j and c_j are coefficients independent of the function ϕ. These coefficients, called shape functions, are found using the least-squares method over the points in the stencil.[5]

Using this discretisation of the flux derivatives, the Euler equations described in Eq. (1) take the form

$$\frac{\mathrm{d}\mathbf{w}_i}{\mathrm{d}t} = -\sum_{j=0}^{n_i} \left(a_{j-\frac{1}{2}}\ \mathbf{f}_{j-\frac{1}{2}} + b_{j-\frac{1}{2}}\ \mathbf{g}_{j-\frac{1}{2}} + c_{j-\frac{1}{2}}\ \mathbf{h}_{j-\frac{1}{2}} \right) \tag{3}$$

where $a_{j-\frac{1}{2}}$, $b_{j-\frac{1}{2}}$ and $c_{j-\frac{1}{2}}$ are the shape functions calculated from the polynomial least-squares reconstruction. The shape functions and the inviscid fluxes are evaluated at the mid-point between the star point and each of its neighbours for stability of the hyperbolic equations. The fluxes are calculated using the approximate Riemann solver of Roe.[7] Second order accuracy is achieved by reconstructing the left (ϕ_L) and right (ϕ_R) states of the Riemann problem as

$$\phi_L = \phi_i + \psi_i\ \mathbf{l}_{ij} \cdot \nabla \phi_i \ , \qquad \phi_R = \phi_j - \psi_j\ \mathbf{l}_{ij} \cdot \nabla \phi_j$$

where $\mathbf{l}_{ij} = \frac{1}{2}\left(\mathbf{x}_j - \mathbf{x}_i\right)$ is the vector formed half way between the star and neighbouring point, ψ is the slope limiter of Barth and Jespersen,[8] $\mathbf{x}$ is the location of the points, and $\nabla\phi$ denotes the gradient of ϕ.

Once the right-hand side of Eq. (3) is calculated, the time integration is performed. This is done first by using an explicit scheme to smooth out the initial flow field

$$\mathbf{w}^{m+1} = \mathbf{w}^m - \Delta\tau \mathbf{R}^m \tag{4}$$

where $\mathbf{R}$ is the residual vector, consisting of the right-hand side of Eq. (3) and the superscript m denotes the time level in pseudo-time τ. Then, a fully implicit method is used to obtain a converged solution

$$\frac{\mathrm{d}\mathbf{w}}{\mathrm{d}\tau} = -\mathbf{R}^{m+1} \tag{5}$$

After linearising the flux residual $\mathbf{R}^{m+1}$ in pseudo-time, Eq. (5) becomes a system of linear equations to be solved for the primitive variables $\mathbf{p}$

$$\left(\frac{1}{\Delta\tau} \frac{\partial \mathbf{w}}{\partial \mathbf{p}} + \frac{\partial \mathbf{R}}{\partial \mathbf{p}} \right) \Delta\mathbf{p} = -\mathbf{R}^m \tag{6}$$

where $\Delta\mathbf{p} = \mathbf{p}^{m+1} - \mathbf{p}^m$ is the difference between these variables after each pseudo-time step, $\frac{\partial \mathbf{R}}{\partial \mathbf{p}}$ is the Jacobian matrix of the system and $\frac{\partial \mathbf{w}}{\partial \mathbf{p}}$ is the transformation matrix between conservative and primitive variables. For the solution of this system to steady-state an approximate form of the Jacobian matrix is formed, and the generalized conjugate residual method,[9] preconditioned with a block incomplete lower-upper (BILU) factorisation, is used.[10] The size of $\Delta\tau$ is determined by a local time-step estimate.[11]

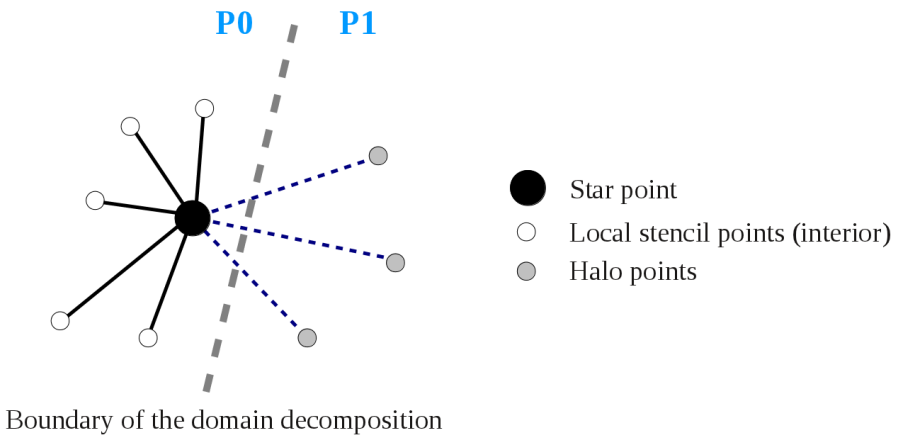


Figure 1. Classification of points across the domain boundary between two processes, P0 and P1.

For time-accurate, unsteady simulations, Eq. (5) must be solved in real-time t, such that

$$\frac{\mathrm{d}\mathbf{w}}{\mathrm{d}t} = -\mathbf{R}^{n+1} \tag{7}$$

where the superscript n denotes the time level in real-time t. The time integration is the done using a dual time-stepping method,[12] in which Eq. (7) becomes

$$\mathbf{R}^* = \frac{3\mathbf{w}^{n+1} - 4\mathbf{w}^n + \mathbf{w}^{n-1}}{2\Delta t} + \mathbf{R}^{n+1} = 0 \tag{8}$$

where $\mathbf{R}^*$ is defined as the unsteady residual. This is a non-linear system of equations that cannot be solved directly. Instead we can view Eq. (8) as a modified pseudo-time steady state problem, which can be solved iteratively for $\mathbf{w}^{n+1}$ by introducing a derivative with respect to the fictitious pseudo-time τ, as explained in Ref. 5. Finally, we obtain the following system for the updates

$$\left(\left(\frac{1}{\Delta\tau} + \frac{3}{2\Delta t}\right)\frac{\partial \mathbf{w}}{\partial \mathbf{p}} + \frac{\partial \mathbf{R}}{\partial \mathbf{p}}\right)\Delta\mathbf{p} = -\left(\frac{3\mathbf{w}^m - 4\mathbf{w}^n + \mathbf{w}^{n-1}}{2\Delta t} + \mathbf{R}^m\right) \tag{9}$$

Further details on the numerical implementation of this meshless method can be found in Ref. 5.

III. Parallel Flow Solver

The parallel implementation of the solver uses single-process-multiple-data (SPMD) programming techniques, in which different processes execute the same instructions on different sets of data. The parallel code is aimed at achieving a homogeneous domain decomposition, with asynchronous point-to-point communication between neighbouring processes. Asynchronous communication means that each process can continue its operations while the parallel communication is being performed. On the other hand, synchronous or blocking communications pause the program until the parallel communication is complete.

Partitioning of the domain is done using the METIS library.[13] Before performing the domain decomposition, connecting edges are formed between all of the points in the domain and their neighbouring points; the connectivities within each stencil are used to do this. The criteria for the domain decomposition are to balance the number of points among partitions and to have the least number of edges cut to decrease communication time.

We classify the points on each process as either interior (which include points on boundaries) or halo points, as shown in Fig. 1. Halo points are not local to the process but are included in local stencils. These points are needed to communicate between processes across communication boundaries. They are ordered so that their variables are updated first and the parallel exchange takes place while the rest of the domain is being updated.

Explicit message-passing-interface (MPI) instructions are used for communication between processes to provide portable parallel execution on distributed, shared, or shared/distributed machines. Non-blocking commands are used to avoid possible deadlocks, to reduce the overhead associated with buffering and to allow for asynchronous operation of the code.

When integrating in explicit mode, communication between processes occurs once per iteration. For implicit integration, using the parallel linear solver, the full Jacobian matrix is divided.[14] This way each process only stores Jacobian matrix data for its local points and the halo points. Due to this partitioning of the Jacobian matrix, the processes need to exchange data during the implicit integration to perform the matrix-vector operations required by the iterative linear solver. The preconditioner only uses local points to reduce memory usage and to avoid parallel communication when forming it. The use of this local BILU preconditioner has an influence on the convergence behaviour of the linear solver in parallel mode. Results in Section VI, however, show good convergence for the test cases with a good speed-up when using parallel processing.

IV. Meshless Preprocessor

The preprocessor described in this work selects stencils automatically from overlapping point distributions; associated with bodies, which may be moving relative to one another. The point distributions are obtained from structured or unstructured meshes; and the original connectivity of the input grids is used as an aid to select the best stencils from the points available. This way the tool works well for isotropic as well as anisotropic regions, which are usually found close to solid boundaries in CFD problems. The stencils selected by the preprocessor provide a direct input to the flow solver. The preprocessor and the solver are designed as two separate tools that can work in coupled mode for moving-body problems. For brevity, the method described here is for geometry in two dimensions; for details of the implementation in three dimensions please refer to Ref. 6. This work focuses on the efficiency of constructing these stencils, using the method presented in Ref. 4.

In broad terms, the preprocessor method can be divided into four stages:

1. Check if the surface elements belonging to the solid boundaries of the different input grids intersect in any way. If so, the boundaries need to be redefined accordingly.

2. Detect points falling inside solid surfaces as a result of the point distributions overlap, and exclude (blank) them from the calculation in the flow solver.

3. Select the final stencils for all active points.

4. Check that the selected stencils respect the boundaries i.e. to make sure that no stencil contains points that lie on the opposite side of a solid boundary.

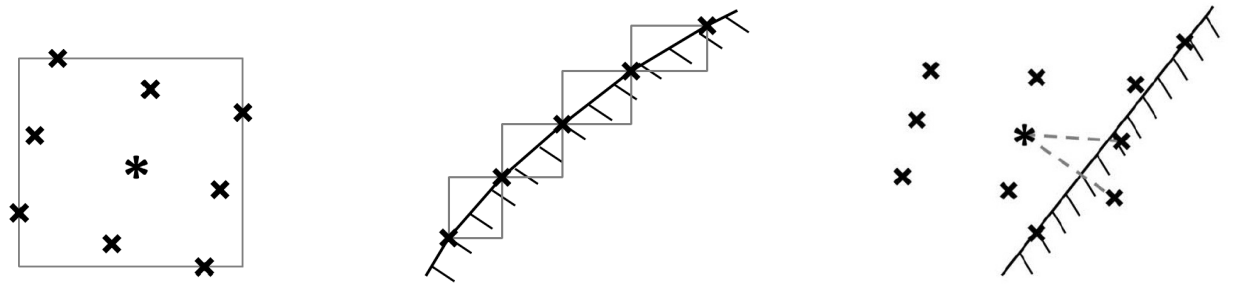
More details are given in the following.

A. Detecting Boundary Overlaps and Redefining Boundaries

A method of detecting solid boundary overlaps, and procedures to redefine the boundaries havae been developed and is described are Ref. 6, but so far this stage has not been implemented in the parallel version. In this work, Stage 1 from above is skipped as there is no boundary overlap in the test cases presented.

B. Blanking Points and Checking Final Stencils

Stages 2 and 4 above, which correspond to blanking the points internal to solid walls and to check that the selected stencils respect the boundaries, are similar in their implementation. They rely on looking for intersections between the boundary elements and the initial stencils of all the points in the domain. These operations are based on the use of higher-dimensional search trees as discussed in Ref. 15. These search trees allow for fast geometry searches by focusing only on regions that will be of interest. The procedure starts by forming bounding boxes around all initial stencils from the input domains, as in Fig. 2(a), and around all boundary elements, as in Fig. 2(b). Then, a search tree containing all stencil bounding boxes is formed. This tree is traversed with the bounding boxes of boundary elements used as a search region. If intersections



(a) Resolving vector for a grid stencil

(b) Bounding box for an internal point stencil.

(c) Blanking points inside boundary elements.

Figure 2. Examples of bounding boxes over stencils and boundary elements.

between bounding boxes are found, then some of the points in the stencil may lie inside solid walls, or in the case of the final checks, some points in the final stencils might not respect the boundaries. To test for these possibilities, intersection algorithms are used. Rays are formed between the star point and all neighbouring points in the stencil. The code then looks for intersections between the rays and the boundary elements as in Fig. 2(c), to define which of the points in the stencil, if any, are to be blanked. In the case of the final checks, these points are removed from the final stencils.

C. Selecting Stencils

Most CFD grids contain anisotropic regions to capture the rapidly changing flow behaviour without having to greatly increase the total number of grid points. The preprocessor method needs to take into account this topology when selecting the final stencils in Stage 3 above. The method defines a resolving vector $\mathbf{v}$ for each point in the domain to account for the topology of the original stencils. This vector is formed before the overlap occurs and points in the direction where the original stencil is the finest. An example can be seen in Fig. 3(a).

The algorithm then performs a search through all of the points in the domain, labelled i, looking for intersections with the stencils of other points. The stencil bounding boxes in Fig. 2(a) are used as before. All of the points labelled j, that intersect the stencil of i, are included in a list of possible candidates for the final meshless stencil of point i. The sum of the resolving vectors is made, including the resolving vector of point i to give the resultant resolving direction. This is illustrated in Figs. 3(b) and 3(c). Using this resultant direction, a new coordinate system is defined, as shown in Fig. 4(a). The basis η is chosen so that the basis vector η_1 lies collinearly to the resultant resolving direction, and η_2 lies orthogonal. Setting the origin of the coordinate system to be the star point, the algorithm calculates the quantities a and b as the projections of the stencils onto the newly created coordinate system, as shown in Fig. 4(b). It also defines the coefficients ξ_1 and ξ_2 as the coordinates of each of the candidate points in basis η. With these quantities, a merit function ψ is defined, which rates each candidate point in terms of the direction and refinement by balancing the orthogonality of the points chosen (for refinement) and distance. The merit function is given by

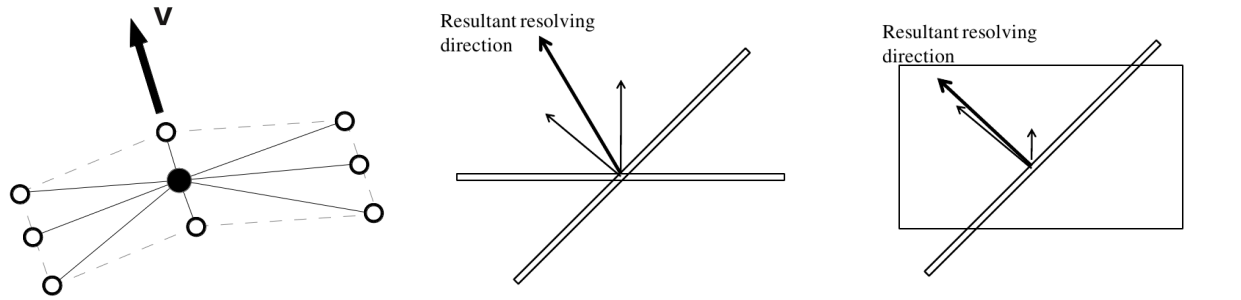
$$\psi = \frac{{\xi_1}^2}{a^2} + \frac{{\xi_2}^2}{b^2} \tag{10}$$

Finally, the method uses this merit function to rate the candidates, and select the most appropriate to form the final stencil by locating them across the quadrants shown in Fig. 4(c). In three dimensions the process is similar, with three-dimensional bounding boxes surrounding the stencils and boundary elements.

For more details on the stencil selection method in two dimensions please refer to Ref. 4, and for details of the implementation in three dimensions please refer to Ref. 6.

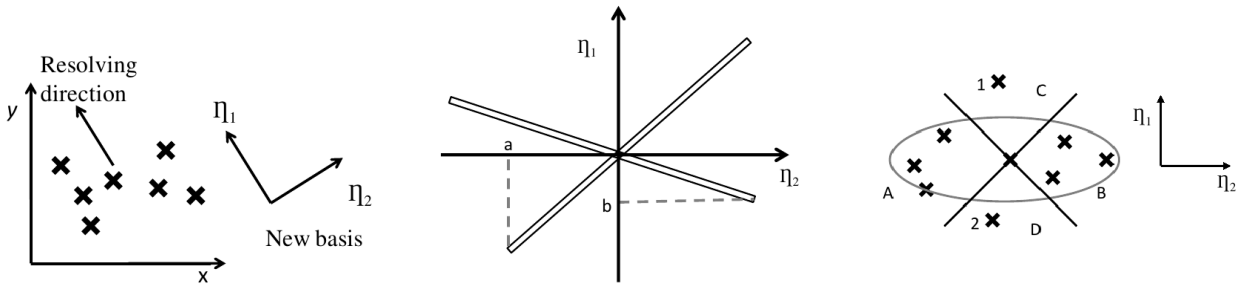
Sorting of Candidates

After the list of candidates is formed for each point in the domain, the stencils are selected according to the merit function ψ described before in Eq. (10). For each point in the domain, the method assigns a value of ψ



(a) Resolving vector for a grid stencil (b) Two anisotropic stencils overlap (c) An anisotropic and a regular stencil overlap

Figure 3. Definition of resolving vectors.



(a) Defining new coordinate system (b) Choosing a and b using stencil projections (c) Quadrants around resolving direction

Figure 4. Definition of new local coordinate system for merit function.

to each of its candidates. It is then required that the list of candidates is ordered according to these values, so that the two candidates for each quadrant with the lowest value of ψ are selected for the final stencil. The problem of sorting lists is well known in the field of computer science. Several algorithms have been developed to tackle the problem but a major issue with all of them is that their efficiency depends on the size of the lists to be sorted and how they are arranged initially. For all sorting problems, there is no a-priori information on how expensive the sorting operation will be for a particular sequence of numbers. In the case of the stencil selection problem this means two things: first, different sorting algorithms need to be tested to ensure efficient operation of the software; second, the parallel load balancing is difficult to achieve since we have no information about the cost of the sorting operation.

In an effort to improve the efficiency of the code, different sorting algorithms are tested. The algorithms to try are all found in Ref. 16:

- Selection Algorithm: The selection sort algorithm works by successively scanning the array to find the next smallest element and swapping it with the corresponding element in the correct position in the queue. The computational cost of this algorithm is fixed as it needs to scan the array the same number of times, regardless of how the array is originally ordered.

- Insertion Algorithm: While forming the array to be sorted, this algorithm finds the correct position for each element and then shifts all the candidates with bigger values of ψ one step to the right. The cost of this algorithm depends on how many elements need to be shifted to store each candidate.

- Bubble Sort Algorithm: This algorithm works by comparing pairs of successive elements, swapping them if necessary and repeating the operation until no more swaps are required. In the worst case scenario, this algorithm is as expensive as the selection algorithm, but depending on the initial ordering of the array it can prove more efficient.

- Shell Sort Algorithm: This algorithm in an extension of the insertion algorithm. It differs in the

fact that it allows the comparison and exchange of elements that are far apart before finishing with neighbouring elements.

- Quicksort Algorithm: This algorithm follows a "divide and conquer" strategy. It creates successive partitions of the array by selecting a pivot. All elements smaller than the pivot are moved before it and all greater elements are moved after it. The operation is repeated until the array is sorted.

The different sorting algorithms are discussed in more detail in Section VI.D, in terms of the efficiency of the preprocessor.

V. Parallel Implementation of Preprocessor

The parallel implementation of the preprocessor is designed to work as much as possible with the same operations of the serial preprocessor described in Section IV. All of these operations rely on the use of search trees as explained before. The main difficulties found when parallelising the preprocessor are distributing these search trees among processes and communicating the trees efficiently. Instead of forming one global search tree and distributing it among processes, the method uses the METIS library[13] to subdivide each of the individual input grids, assigning the same number of points from each grid to each process. From these assigned points each process then forms its own separate search trees that are communicated at run time.

Ideally, to increase the performance of the parallel stencil selection, each process should be allowed to work as independently as possible and the work load should be balanced between processes. An immediate drawback of performing tree searches for the stencil selection, is that it is not possible to have prior information about the computational cost of the operations for each individual point. The problem is aggravated as the candidate points need to be sorted according to the merit function, as explained in Section IV.C. These considerations make it difficult to correctly load balance the stencil selection problem, as it will be demonstrated in the results section.

Regardless of the domain decomposition, the parallel preprocessor follows the same four stages described in Section IV. Stage 3, which corresponds to selecting the final stencils, is modified and divided into the following operations:

(a) Form the local search trees and perform the local search for candidates, following the same method described for the serial preprocessor.

(b) Identify the points located on regions of inter-processor overlap, to make sure only information from relevant points is communicated across processes.

(c) Execute the parallel communication of search trees, coordinates and resolving vectors for the points identified as relevant.

(d) Perform the remote search for candidates using the information received from other processes.

(e) Select the final stencils from the candidate list.

The parallel versions of the four stages described in Section IV are discussed in more detail below, with emphasis on Stage 3 where most of the parallel operations are found.

A. Detecting Boundary Overlaps and Redefining Boundaries

The described parallel method currently only works for problems where no overlap of solid boundaries is found. For this reason, Stage 1 is skipped when running in parallel.

B. Blanking Points and Checking Final Stencils

When running in parallel, each process stores all of the global boundary elements. By doing this, all the operations related to boundaries can be performed by each process working independently from others. This is the case for stages 2 and 4. To detect and blank local points that lie inside solid boundaries, each process autonomously performs the same procedure described in Section IV.B. The final stage in the parallel preprocessor is checking that the selected stencils are valid and that they respect the boundary elements. Similar to Stage 2, all processes work independently to check the validity of the selected stencils for their local points.

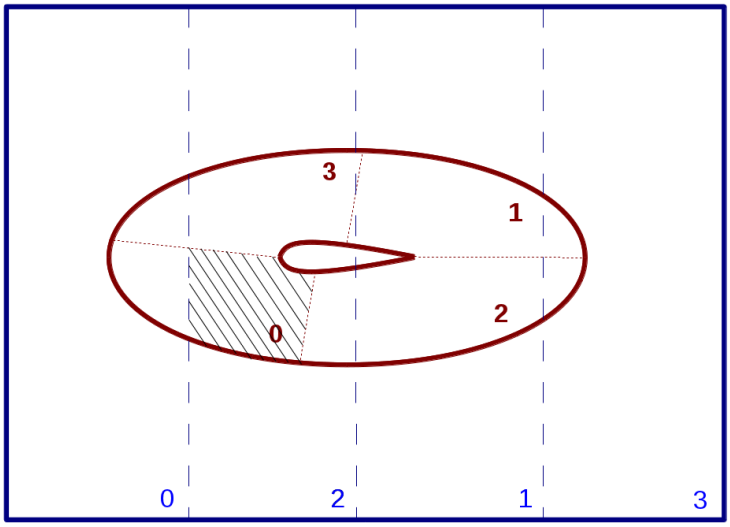


Figure 5. Example of pre-search for two overlapped domains divided among 4 processes.

C. Selecting Stencils

In parallel, the stencil selection procedure can be divided as follows:

Forming Local Search Trees and Performing Local Search

In the first operation above, all processes form bounding boxes for each of the local points in the domain. Then, search trees are created from these bounding boxes, as described in Section IV, with only local points taken into account. All processes then perform a search by traversing their local trees to find candidates for all their local points. This procedure runs independently on each process and is the same used in Section IV.C to find candidate points.

Identification of Relevant Points to be Communicated

When running in parallel, each process needs to find candidates for all its assigned points by searching through its local data, but it also needs to find potential candidates that are stored remotely in the other processes. In order for the parallel method to be efficient, each process needs to store and process as little remote data as possible. To increase efficiency, before communicating the trees each process identifies a list of relevant points to be sent to other processes. This means that all processes will perform a preliminary search for points located in regions of inter-process overlap, thus making sure that only relevant data is communicated.

Figure 5 shows an example where an aerofoil (grid A in red) overlaps with a rectangular background grid (grid B in blue) and the job is divided among four processes. The domain decomposition in the figure is arbitrary but it serves the purpose of showing how the preliminary search works. Each individual grid is divided into four partitions and the preliminary search will result, for instance, in process 0 identifying the area marked by the black diagonal lines, as the inter-process region between processes 0 and 2. Points from grid A, located inside this region are then identified as relevant to be sent to process 2. Process 0 on the other hand will not send any information about grid A to processes 1 or 3. After the preliminary search is complete, new search trees are created with the points that are found relevant for communication. These search trees will be exchanged next.

Parallel Communication of Trees

In the third step, information is exchanged among processes. The data to be communicated includes search trees, coordinates, bounding boxes and resolving vectors for all the points identified before. In the prepro-

cessor, the search trees are specialised data structures that store the needed information for each leaf (node) of the tree and directions on how these leaves connect to other leaves. The total memory size for these structures can vary depending on the actual arrangement of the tree.

The MPI interface requires a rigorous definition of the data types to be communicated. The basic predefined MPI data types allow for the communication of bytes, integers, floating point numbers and characters, as well as user-defined data structures, as long as the memory size of the user-defined structure is known (and fixed). Since there is no predefined size for the tree data structures used in this algorithm, a way of representing trees with the predefined MPI data types is needed.

A procedure to pack the skeleton of the trees using two integer arrays and one double array was devised. One of the integer arrays holds the map of how to traverse the tree, and the other one stores the index of the nodes. The double array holds the geometric data for the node. The procedure to pack the tree starts from the root and visits each of the nodes of the tree until it has been traversed entirely. This algorithm, besides allowing the use of standard MPI data types, reduces the size of the message to be sent when compared to storing all the nodes with all its children (existing or not). The process is analogous to compacting a full matrix into a sparse one. To unpack the trees after they are received, the code traverses the first integer array as a map, and creates nodes using the information stored in the other two arrays.

Remote Search

With the received data, step four calls for each process to perform a search for candidates by traversing the received remote trees. Any remotely located candidates are added to the candidate list from the previously performed local search. For the example of Fig. 5, this remote search means that star points located in process 2 will search through the trees received from the dashed region in process 0, trying to find suitable candidates.

Final Selection of Stencils

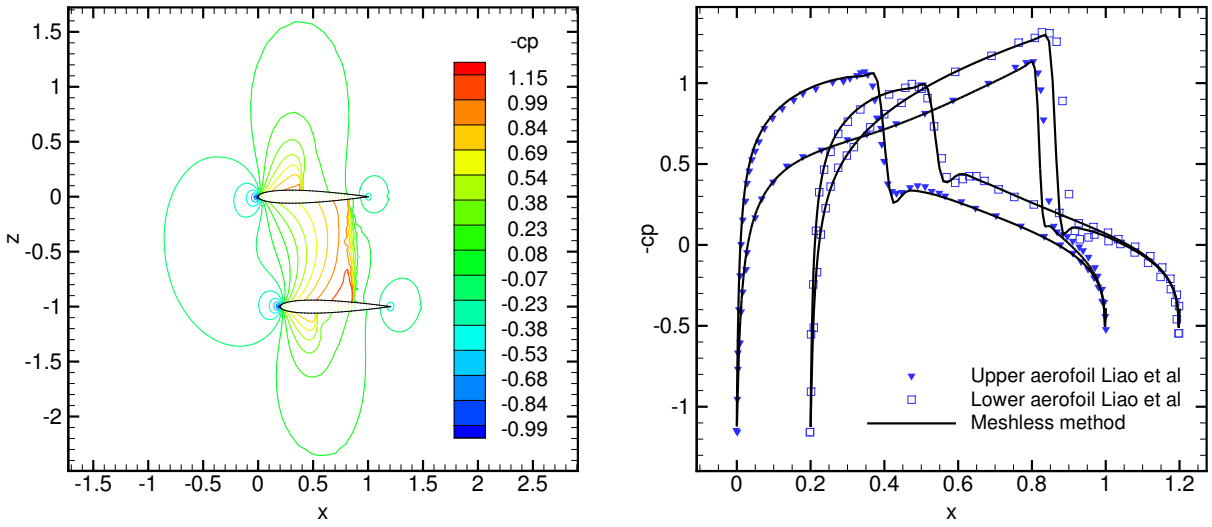
In the last step, each process performs the procedure described in Section IV.C to select the stencils. The received resolving vectors from the remote candidates are summed with the local vectors to form a final resultant resolving direction. With it, the new coordinate system is defined and candidate points are assigned a ψ value from Eq. (10). These candidates are then ordered by increasing value of ψ, and the final stencils are selected. It is in this stage that the biggest load imbalance is found. Even though all processes are assigned the same number of points, it is likely that the number of candidates per point is different as well as the computational cost of the sorting operation.

D. Parallel Transient Simulations

The process of running parallel transient simulations with movable geometry starts with a call to the preprocessor to initialise the simulation. The preprocessor reads the input grids and divides them using the METIS library in such way that each process stores the same number of points, as it was the case with steady-state simulations described before. The preprocessor will then perform all the stencil selection operations and write the stencils to an output file. This file is then read by the flow solver, which performs its own domain decomposition, as explained in Section III, before solving the governing equations.

After this first iteration, a closed loop starts with successive calls to the preprocessor and flow solver. At the beginning of each real time-step, the points are moved in space according to the prescribed motion and the preprocessor is called to re-calculate the new stencils with the method described. The flow solver then uses these stencils to calculate the flow solution. Both the preprocessor and flow solver perform their operations based on the initial domain decomposition from the parallel flow solver. This eliminates the need for input/output (I/O) files while running in coupled mode.

The decision to use the same domain decomposition in the preprocessor and solver was made to remove the cost of partitioning the domain in each call to the preprocessor and to avoid unnecessary I/O operations. It was demonstrated in ref. 6 that the flow solver is much costlier in terms of computing time than the preprocessor. For this reason it was decided to favour the flow solver domain decomposition as an acceptable compromise. This explains why the preprocessor is designed to work with any type of domain decomposition given by the solver.



(a) Pressure coefficient contours

(b) Surface pressure distribution

Figure 6. Flow solution of test case 1.

VI. Results

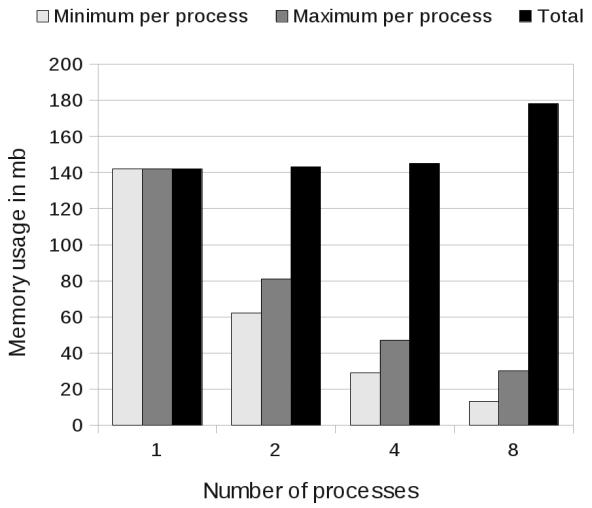
In this section, the performance of the parallel stencil selection and flow solver are evaluated using different test cases, both in two and three dimensions. The test cases are academic, but serve the purpose of demonstrating the capabilities of the method. The first test case is two-dimensional and consists of two NACA0012 aerofoils in a steady-state simulation. The second case is a bigger steady-state case in three dimensions, in which a store is overlapped with the grid of a generic fighter geometry. The final test is a transient three-dimensional case, in which a store is released from a wing geometry and shows the full capabilities of the parallel preprocessor and flow solver.

A. Two-Dimensional NACA0012 Biplane

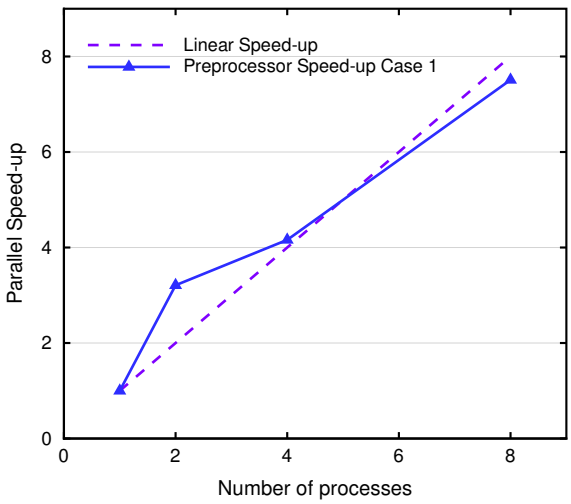
In this simple two-dimensional case, two NACA0012 aerofoils are overlapped. The background grid includes the first aerofoil and contains 18,500 points. The second grid, formed from 17,500 points, contains the other aerofoil and is positioned over the first grid. The location of the first aerofoil is such that the leading edge coincides with the origin. The leading edge of the second aerofoil is located at coordinates $(0.2c, -1.0c)$, where c is the chord length of the aerofoils. The preprocessor and flow solver were run in both serial and parallel modes with different numbers of processes. The flow conditions for the test case are a freestream Mach number of 0.755 and 0.016 degrees angle of attack. The test case is shown in Fig. 6(a). In the figure three shock waves are visible, one on the upper surface of the top aerofoil, one on the lower surface of the second aerofoil and a strong one in between the two. Figure 6(b) shows the surface pressure coefficient, along with numerical results from Liao *et al.*,[17] which uses the Chimera method on two overlapping structured aerofoil grids to solve the Euler equations. The results show good agreement, though there is a slight difference for the location of the main shock in between the aerofoils.

The performance of the parallel preprocessor is measured by two metrics, memory usage and speed-up. The cases were run on a system of desktop machines. Figure 7(a) shows the memory usage for each process when running the preprocessor on one to eight processes. As expected, for the higher number of processes the total memory consumption increases. This is due to the extra data stored in regions of inter-process overlap. The memory overhead is manageable however and shows good scalability.

The parallel speed-up for case 1 is shown in Fig. 7(b). Here we can see a super-linear reduction in calculation times for this particular test case when running on two and four processes. This is due to the pre-search, performed when generating the communication lists effectively ruling out points that might be included as candidates otherwise when running in serial mode. This means that the list of candidates per

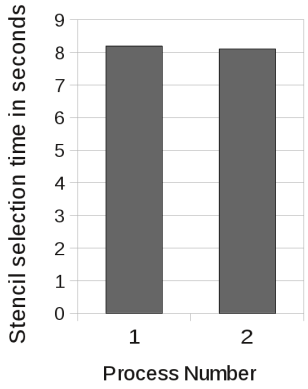


(a) Memory usage for the preprocessor for case 1

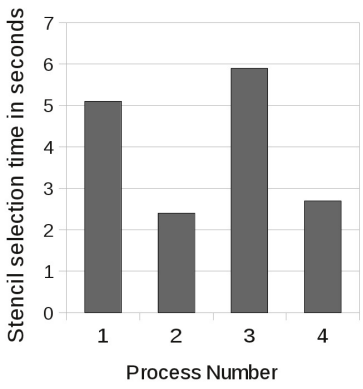


(b) Parallel speed-up for the preprocessor for case 1

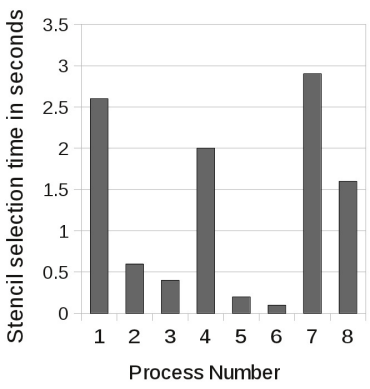
Figure 7. Parallel efficiency of preprocessor for case 1.



(a) Run times for 2 processes



(b) Run times for 4 processes



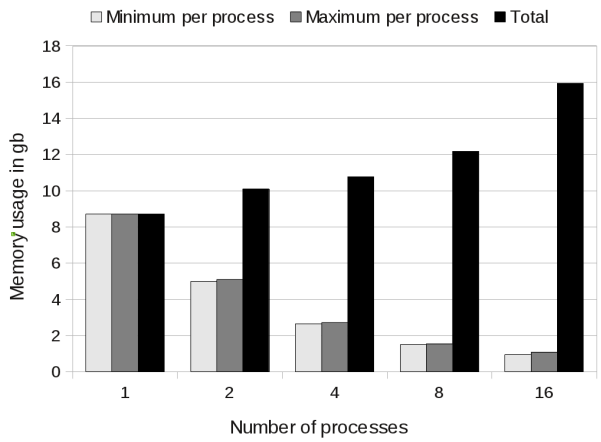
(c) Run times for 8 processes

Figure 8. Load balance for stencil selection for case 1.

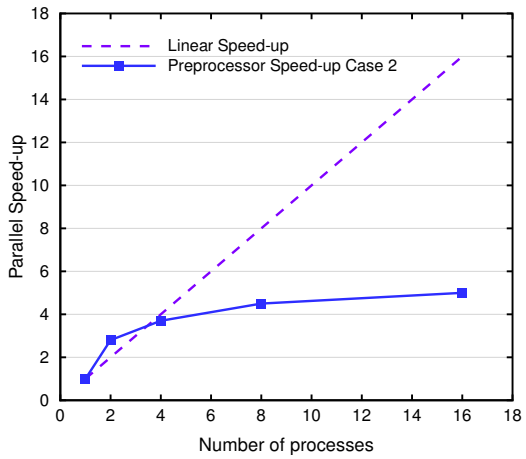
point will be smaller, making the sorting of the candidates faster. The highest speed-up shown is when running in two processes. This can be explained by Fig. 8, where the run times for the stencil selection part of the preprocessor are shown. This figure gives an indication of the load balance for the preprocessor. When using two processes for instance (Fig. 8(a)), the work load is well balanced and both processes finished their operations in about the same time. When running in eight processes on the other hand (Fig. 8(c)), the load balance is not as good, resulting in some processes finishing faster than the rest. As explained in Section V.C, this load balance issue is a result of making the preprocessor work with a domain decomposition based on the connectivity of the individual input grids. There is scope in the future to optimise the partitions to improve the load balance, thus improving the speed-up.

B. Open Source Fighter with Store

The evaluation of the method continues by analysing a bigger three-dimensional test case. It is a half-model of a generic fighter aircraft based on publically available data of an F-16 fighter. Details of the geometry are found in Ref. 18. The second body in the simulation is a store located under the wing of the aircraft. The aircraft domain is formed of 4.5 million points, while the store domain contains 390,000 points. The flow conditions for this case are a freestream Mach number of 0.6 and 1.2 degrees angle of attack.

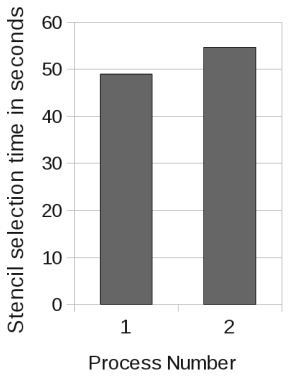


(a) Memory usage for the preprocessor for case 2

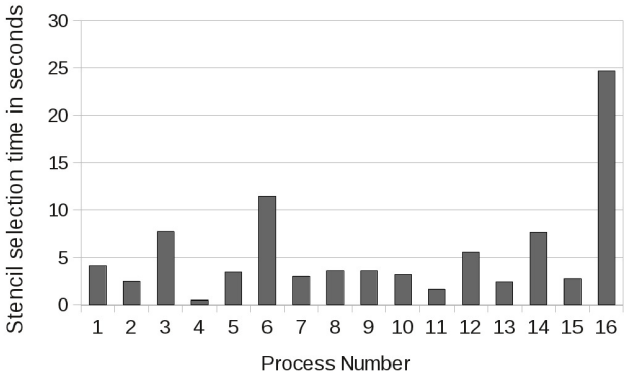


(b) Parallel speed-up for the preprocessor for case 2

Figure 9. Parallel efficiency of preprocessor for case 2.



(a) Run times for 2 processes



(b) Run times for 16 processes

Figure 10. Load balance for stencil selection for case 2.

The test case was run with the parallel preprocessor operating independently from the flow solver. As explained at the beginning of Section V, the preprocessor divides the domains based on the original connectivity of the input grids. After the selection of the final stencils, they are written to an output file. This file is then loaded into the flow solver which performs its own domain decomposition.

The preprocessor memory usage and speed performance for this case can be seen in Fig. 9. The results show that the speed-up for this test case is not as good as with test case 1. When running two processes, we see an increase in speed of 2.76 when compared with running one process. This improvement, however, decreases until reaching a speed-up of only 5 when using 16 processes. There are two reasons for this. The first one is the load balancing as with the aerofoil case. Figure 10 shows the run times for the stencil selection using 2 and 16 processes. The load balance for two processes is good, resulting in good speed-up as shown in Fig. 9(b). On the other hand, when using 16 processes it is obvious that the load balancing is not ideal, as some of the processes finished their tasks much faster, and were idle until the last one finished. When running on more processes, the time taken in the preliminary search to identify points to be communicated and in the actual communication increases. This is the second factor on why the parallel efficiency decreases for this test case. Nevertheless, the speed-up combined with the scalability in terms of memory usage means the parallel preprocessing method can be applied to bigger cases resembling real industrial applications.

After the preprocessor finishes selecting the stencils, they are written to a file used by the flow solver as input. The flow solver is the run on a different number of processes ranging from 1 to 16. Figure 11

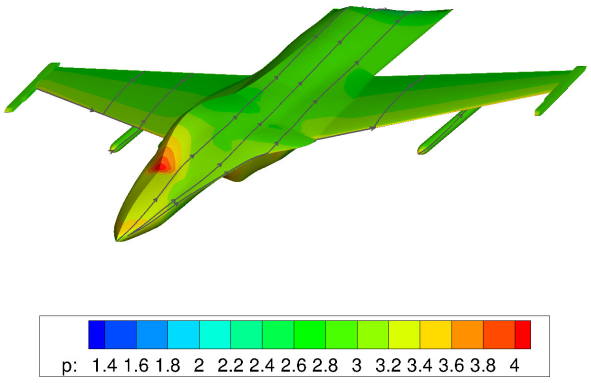


Figure 11. Flow results for fighter aircraft case.

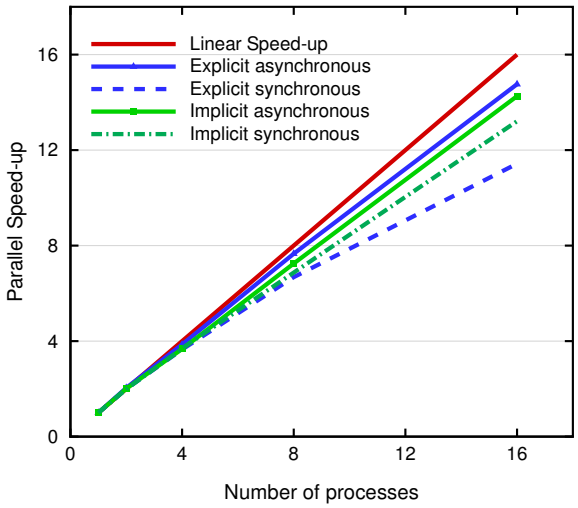
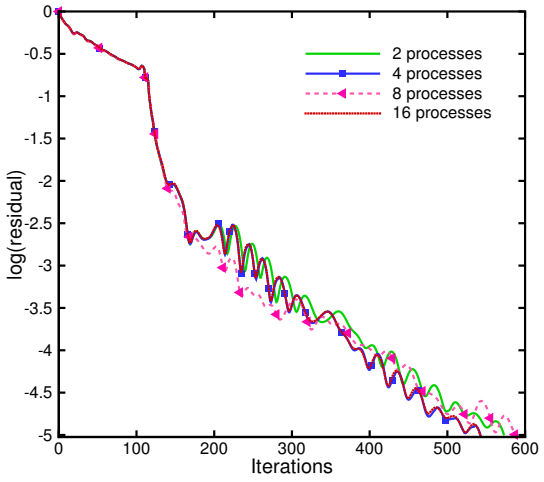
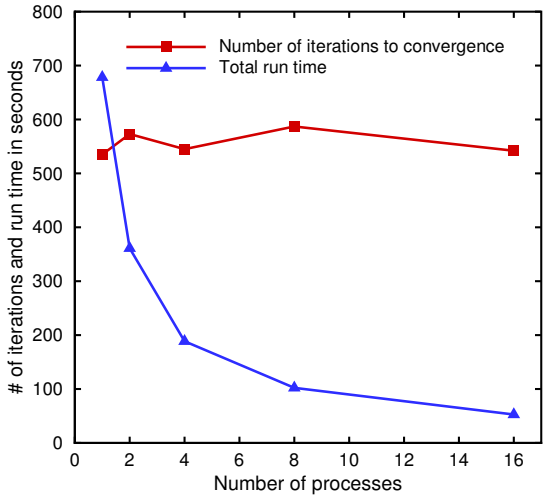


Figure 12. Parallel speed-up of flow solver for case 2.



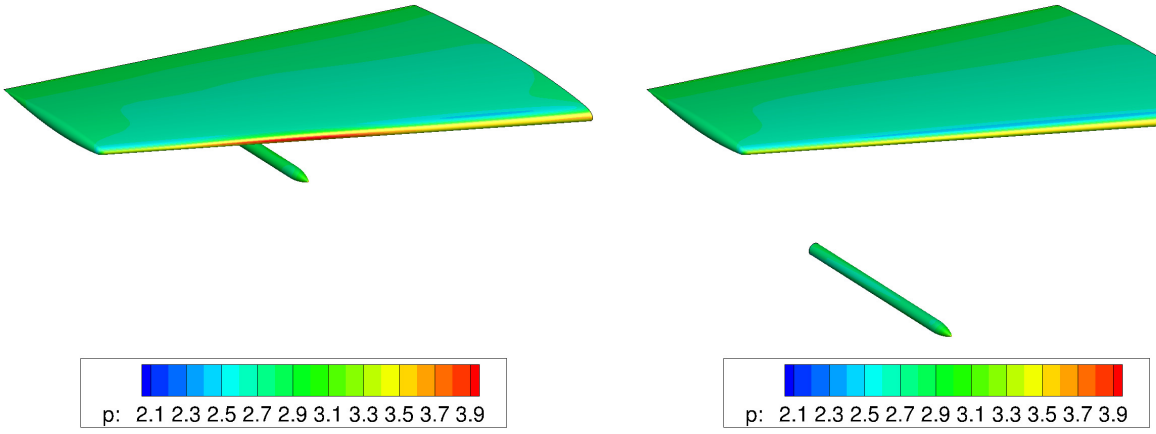
(a) Convergence histories for case 2



(b) Convergence of flow solver for case 2

Figure 13. Parallel flow solver performance.

shows the flow solution for the test case. Here the pressure contours are plotted and streamlines of velocity are drawn. Figure 12 shows the parallel speed-up per iteration of the flow solver. Here, the advantage of using asynchronous communication in the code is clear. On average for this test case, the asynchronous communication proves 10% faster per iteration than the synchronous one, with the difference increasing as we run in more processes. Figure 13(a) shows the normalized convergence histories for the case. This test case was run in explicit mode for 100 iterations before changing to implicit integration. In this graph we see the effects of using a local BILU preconditioner with approximate linear solves, as the convergence behaviour is different when running on a different number of processors. Figure 13(b) shows that, even though convergence is somewhat affected, an important reduction in the total calculation times is achieved.



(a) Pressure contours for case 3 at time 0

(b) Pressure contours for case 3 at time 5

Figure 14. Flow solution for test case 3.

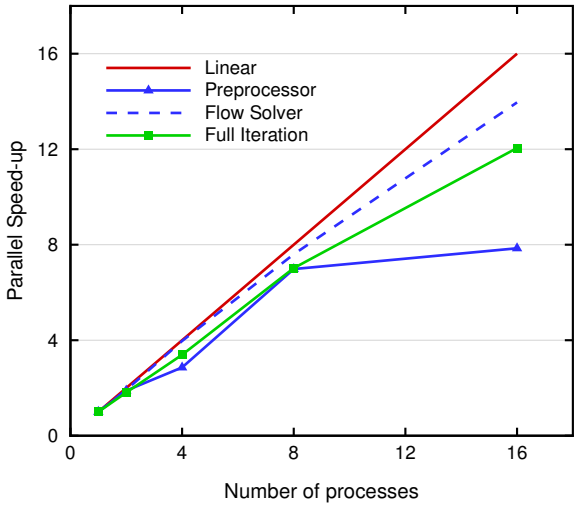


Figure 15. Average parallel speed-up for a full time-step for case 3.

C. Onera Wing with Store in Transient Mode

To assess the capabilities of the parallel preprocessor and flow solver of performing time-accurate simulations with bodies in relative motion, a simple store release case is investigated. The case consists of an ONERA M6 wing with a store located beneath it. The M6 wing is a semi-span wing, with a symmetric aerofoil section, leading edge sweep angle of 30 degrees, an aspect ratio of 3.8 and a taper ratio of 0.562. The point distribution for the wing contains 1.2 million points. The store used is the same as in case 2. The initial location of the store is at 15% mean chord length below the wing, and 62% wing span length from the root. The store is then moved down in a pre-defined motion, at a vertical speed of $\dot{z} = -0.1U_\infty$, where U_∞ is the freestream speed. There are 100 time steps performed with $\Delta t = 0.05$. The flow conditions for this test case are a Mach number of 0.5 and an angle of attack of 3 degrees.

The pressure contours at times $t = 0$ and $t = 5$ are shown in Fig. 14. The influence of the store on the pressure field around the wing is noticable at the beginning of the calculation and as expected, this influence reduces as the simulation progresses. Figure 15 shows the average parallel speed-up for a full time-step of the simulation. This is the total time needed to calculate one iteration of the preprocessor-solver loop as

explained in Section V.D. The computational cost per time-step of the flow solver is on average about 5 times higher than the cost of the preprocessor. This justifies the decision of making the preprocessor work with the flow solver domain decomposition. In the future, however, different domain decomposition methods for the preprocessor will be investigated, as well as methods for changing the load balance of the flow solver as the transient simulations progress.

Table 1. Profiling of the code.

Parallel Preprocessor Operation	Percentage of total CPU time for case 1	Percentage of total CPU time for case 2
Blanking points	0.7 %	2.5 %
Local candidate search	4.9 %	2.5 %
Preliminary search	3.9 %	3.2 %
Parallel communication	0.2 %	0.9 %
Remote candidate search	2.0 %	5.3 %
Sorting candidates and selecting stencils	85.3 %	79.4 %
Checking final stencils	3.0 %	6.2 %

Table 2. Speed-up of different sorting algorithms.

Test case	Selection sort	Insertion sort	Bubble sort	Shell sort	Quicksort
Case 1	1.0	2.8	2.8	7.1	9.6
Case 2	1.0	2.8	2.9	7.6	9.0

D. Profiling the Code and Sorting Algorithms

The performance of the different operations of the preprocessor was evaluated by profiling the code for the first two test cases above. The CPU time per operation was measured and the average of five runs is taken. Table 1 shows the relative computational cost of the preprocessing operations when running two processes. From the results it is clear that the most expensive operation of the preprocessor is the sorting of the candidates according to the merit function.

In an effort to improve the efficiency of the sorting operation, the different sorting algorithms described in Section IV.C were tested. Table 2 shows the speed-up compared to the most expensive algorithm (selection sort) for test cases 1 and 2. The bubble sort and the insertion algorithms proved to be of similar efficiency, with both of them being about 3 times faster than the selection algorithm. There is an increase in speed of more than 7 times using the shell sort algorithm in respect to the least efficient one, while the quicksort method is more than 9 times faster than the selection algorithm.

VII. Conclusions and Future Work

The development of a parallel automatic preprocessing tool has been presented. The preprocessor takes different overlapping grids that can be stationary, or moving relative to one another, and selects meshless stencils which are used by the flow solver to solve the governing equations. The method applies algorithms that allow the use of standard MPI routines to communicate the search trees needed by the preprocessor and employs asynchronous communication in the flow solver to improve the speed-up.

Three test cases were computed, showcasing the method as a powerful tool for applications where different component grids can move in relation to one another. The parallel preprocessor and flow solver prove to be efficient in terms of speed-up and memory usage, even though the preprocessor currently does not have an efficient load balancing method. Different algorithms were tested in an effort to increase the performance of the candidate sorting operation which turns out to be the most expensive operation of the stencil selection. The quicksort algorithm was found to be the most appropriate from the algorithms tested.

In future work, the parallel preprocessor needs to address the problem of boundary redefinition when solid walls from different input grids intersect. Also, better load balancing techniques for the parallel preprocessor are being investigated in an effort to improve efficiency further.

Acknowledgements

This study is supported by a studentship from the Engineering and Physical Sciences Research Council and BAE Systems.

References

[1]Steger, J. L., Dougherty, F. C., and Benek, J. A., "A chimera grid scheme," *Advances in Grid Generation ASME FED*, Vol. 5, 1983, pp. 59–69.

[2]Liu, G. R., *Meshfree Methods - Moving beyond the Finite Element Method*, CRC Press, 2002, ISBN: 0849312388.

[3]Belytschko, T., Krongauz, Y., Organ, D., Fleming, M., and Krysl, P., "Meshless methods: an overview and recent developments," *Computational Methods in Applied Mechanical Engineering*, Vol. 139, 1996, pp. 3–47.

[4]Kennett, D. J., Timme, S., Angulo, J. J., and Badcock, K. J., "Semi-Meshless Stencil Selection for Anisotropic Point Distributions," *International Journal of Computational Fluid Dynamics*, Vol. 26, 2012, pp. 463–487.

[5]Kennett, D. J., Timme, S., Angulo, J. J., and Badcock, K. J., "An Implicit Meshless Method for Application in Computational Fluid Dynamics," *International Journal for Numerical Methods in Fluids*, Vol. 71, 2012, pp. 1007–1028.

[6]Kennett, D. J., Angulo, J., Timme, S., and Badcock, K. J., "Semi-Meshless Stencil Selection on Three-Dimensional Anisotropic Point Distributions with Parallel Implementation," *AIAA Paper 2013–0867. Presented at the 51st AIAA Aerospace Sciences Meeting, Grapevine, Texas, Jan 2013.*.

[7]Roe, P. L., "Approximate Riemann solvers, parameter vectors and difference schemes," *Journal of Computational Physics*, Vol. 43, 1981, pp. 357–372.

[8]Barth, T. J. and Jespersen, D. C., "The Design and Application of Upwind Schemes on Unstructured Meshes," *AIAA paper*, Vol. 89, 1989.

[9]Eisenstat, S., Elman, H., and Schultz, M., "Variational Iterative Methods for Nonsymmetric Systems of Linear Equations," *SIAM Journal of Numerical Analysis*, Vol. 20, 1983, pp. 345–357.

[10]Axelsson, O., *Iterative Solution Methods*, Cambridge University Press, 2004.

[11]Katz, A., *Meshless Methods for Computational Fluid Dynamics*, Ph.D. thesis, Stanford University, 1999.

[12]Jameson, A., "Time Dependent Calculations Using Multigrid, with Applications to Unsteady Flows Past Airfoils and Wings," *SIAM Journal of Numerical Analysis AIAA Paper 91-1596, AIAA 10th Computational Fluid Dynamics Conference, Honolulu, HI*, 1991.

[13]Karypis, G. and Kumar, V., "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM. Journal on Scientic Computing*, Vol. 20, 1999, pp. 359–392.

[14]Woodgate, M., Badcock, K. J., and Richards, B. E., "A Parallel 3D Fully Implicit Unsteady Multiblock CFD Code Implemented on a Beowulf Cluster," *Parallel CFD*, Vol. 20, 1999, pp. 359–392.

[15]Bonet, J. and Peraire, J., "An alternating digital tree (ADT) algorithm for 3D geometric searching and intersection problems," *International Journal of Numerical Methods in Engineering*, Vol. 31, 1991, pp. 1–17.

[16]Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C., *Introduction to Algorithms*, The MIT Press, 2009, ISBN: 0262033844.

[17]Liao, W., Cai, J., and Tsai, H. M., "A multigrid overset grid flow solver with implicit hole cutting method," *Computer Methods in Applied Mechanics and Engineering*, Vol. 196, 2007, pp. 1701–1715.

[18]Marques, S., Badcock, K. J., Khodaparast, H. H., and Mottershead, J. E., "Transonic Aeroelastic Stability Predictions Under the Inuence of Structural Variability," *Journal of Aircraft*, Vol. 47, 2010, pp. 1229–1239.