V European Conference on Computational Fluid Dynamics
ECCOMAS CFD 2010
J. C. F. Pereira and A. Sequeira (Eds)
Lisbon, Portugal,14-17 June 2010

PARALLEL GRID GENERATION FOR LARGE EDDY SIMULATION

Gary J. Page

Aeronautical and Automotive Engineering,
Loughborough University,
Leicestershire, UK.
e-mail: G.J.Page@lboro.ac.uk

Key words: Grid Generation, Large Eddy Simulation, Parallel Processing, Immersed Boundary, Ray Tracing

Abstract. *As Large Eddy Simulation is increasingly applied to flows containing complex geometry, grid generation becomes difficult and time consuming when using software originally developed for RANS problems. The traditional 'pipeline' approach of grid generation → solve → visualise entails the time consuming transfer of large files and conversion of file formats. This work demonstrates a grid generation methodology developed specifically to be integrated with parallel LES. The current approach is to use a Cartesian grid with adaptive refinement based upon geometry intersection and surface curvature. User input is limited to definition of required large and small cell sizes, and resolution related to curvature. The grid is defined by an octree data structure with the geometry defined by triangular facets using the STL file format. The result is a set of 'cubical' subdomains, each with identical numbers of cells and uniform distributions within the cube. Some subdomains will be entirely fluid and can be solved using straightforward CFD techniques, whilst some cubes will be cut by the surfaces. Individual cells are then tagged as 'solid', 'fluid' or 'cut' with the solver expected to use an immersed boundary approach to model the surface. A key feature is the design of the algorithm to be parallelisable on both shared and distributed memory systems. For example to determine 'solid' or 'fluid' for each cell within a cube, ray-tracing is used in conjunction with a scanline fill to avoid dependencies between cubes whilst reducing the number of costly ray-tracing evaluations. Shared memory parallel and distributed memory parallel implementations are described. The distributed memory parallel dynamically partitions the grid as it is being generated. Grid generation testing has been carried out on a variety of input CAD files ranging from an abstract geometry with 888 facets to a realistic aircraft landing gear with 350,000 facets. The landing gear case shows how the grid generator correctly finds the fluid inside of the tire and other cavities within the hub. Running in scalar mode, a grid with 4,916 cubes and 468 million cells is generated in less than 100 seconds, whilst in parallel on 32 processor cores this can be achieved in 4.6 seconds. A typical parallel speed up on eight cores is a factor seven, whilst on 32 cores a speed up of 20 is achieved. As load balancing is optimised to achieve a good balance for the flow solver, the load balancing can be poor for the grid generator.*

1 INTRODUCTION

In industrial application of CFD using Reynolds Averaged Navier-Stokes (RANS) approaches, grid generation is often one of the most difficult and time consuming tasks; furthermore, the quality of the grid can have a major influence on the accuracy of the flow solution. Up until recently, Large Eddy Simulation (LES) has been restricted to calculations on simple geometries in order to develop and validate the core methodology. LES can be even more sensitive to grid quality and in particular high aspect ratio elements (typical in RANS) can lead to poor solutions. Existing RANS grid generators are not suitable for LES when creating large grids (e.g. 100 million cells or more) as they are usually interactive scalar processes that are slow and use large amounts of memory. For these large simulations, the 'pipeline' paradigm of grid generation $\rightarrow$ solve $\rightarrow$ visualise is becoming a hindrance as it entails the time consuming reading, writing and copying of large files and conversions between different formats. The aim of this work is to develop a grid generation technique for LES that is fast and efficient, exploits parallelism and is integrated into the flow solver.

Over the years of CFD development there have been many fundamental approaches to grid generation with the most popular being 'body-fitted': the grid is wrapped around the object of interest and the finite volume cell faces coincide with the boundary surface. However, for complex geometry this can be a difficult task and can lead to poor finite volume shapes. The alternative approach of using a Cartesian grid to fill the domain with special treatment where the grid intersects the solid boundaries was originally proposed for Euler flow calculations [1] [2]. De Zeeuw and Powell [1] used a hierarchical cell-based quadtree data structure to adaptively refine the mesh in regions of high gradient: both at a wall and where shocks were found. For Aftosmis et al. [2] the Cartesian grid is nested so that error estimates may be computed in order to drive the refinement in a rigorous manner. For viscous flow both Tseng and Ferziger [3] and Emblemsvag et al. [4] used Cartesian grids with an 'immersed boundary' and 'ghost cells' to handle the wall interface. However, both of these works use a simple background Cartesian grid with predefined clustering near the object of interest - there is no hierarchical adaption to help resolve features near the wall. Kang et al. [5] have studied the accuracy of the immersed boundary technique and demonstrated how this is suitable for LES.

Kamatsuchi [6] and Ishida et al. [7] present the 'Building Cube Method' which uses an octree data structure in three-dimensions to handle the subdivision of the cubes to resolve surfaces and flow features. Since each cube has an identical number of grid points the authors point out how this simplifies parallel load balancing. The method presented in the current work is similar, although it has been developed independently.

This paper continues with a description of the core grid generation methodology and the parallel implementations. The following section shows example grids for a variety of input geometries and timings to show the parallel speed-up and efficiency. Finally, conclusions are presented and issues that still need resolving are discussed.

2 METHODOLOGY

2.1 Solid Model CAD Representation

Commercial CAD packages use their own internal representation of three-dimensional solid models and proprietary file formats. Whilst the grid generation package could connect directly to the CAD package through interface libraries, this would involve multiple interfaces to support the most commonly used packages. More importantly, this typically requires licensing, and when generating a grid in parallel we may wish to access hundreds or thousands of licenses simultaneously. The alternative is to use a neutral standards defined format such as IGES or STEP, but in practice both of these formats have many pitfalls. An alternative is the stereo lithography (STL) file format used to transfer CAD data to rapid prototyping machines. Whilst this is a proprietary format, it is very simple, is available as an output option on all CAD systems and can be viewed by CFD visualisation tools. Essentially, the STL file defines a solid as a list of triangular facets with an optional facet normal. There is no connectivity between facets defined within the file and the coordinate of a vertex which is part of say four facets will be repeated four times in each facet definition. This does lead to the possibility that small gaps can appear between facets, but in practice this has found to be extremely rare and can be easily fixed by matching vertices with a given tolerance. The only drawback to the STL format is that smooth curved surfaces are always represented fundamentally in a faceted form. It is important that the user select a sufficiently fine facet resolution when generating the STL from CAD if a smooth representation of curved surfaces is important.

2.2 Octree Grid Generation Algorithm

The fundamental data structure used to represent the hierarchical refined grid is an octree. For an octree, each *parent* node has up to eight *children*. The nodes at the bottom of the tree are *leaf nodes* which contain data. Simple procedures are then available to grow nodes, prune the tree, traverse the complete tree, or just traverse the leaf nodes.

For grid generation the data stored in the leaf node is a *cube* which specifies the origin in space of the cube, its size and number of grid points in each Cartesian direction. This information is then sufficient to compute the (x, y, z) coordinates of any grid point for a given cube and (i, j, k) indices. To refine the grid in this cube, eight children are grown from the original node, which becomes a parent. These children inherit the number of grid points, but now the size of the cube has halved so grid resolution is doubled. The parent is now no longer a leaf node and does not contain a grid (i.e. the grids are contiguous and non-nested). Within the cubes, the grid lines define the edges of the finite volume cells for flow computation. (Note that cube is a convenient term, but it is not required that all three Cartesian directions are of equal size or of equal number of grid points).

Prior to the grid generation process we need one or more faceted representations of geometry, a size and position of an outer domain box, a target largest cell size, a target small size for resolving geometry, and an optional smaller target size for improved

refinement in regions of high geometry curvature.

To illustrate the process, refer to Figure 1 which shows the generation of a grid for a simplified car body. The problem is three-dimensional, but, for simplicity, the figures are showing a cut through the mesh and the size of the cells have been increased so that they are more visible.

To commence the process the root node is created and this is set to define the cube that represents the outer domain.

There is then an iterative process in which the leaf nodes of the octree are repeatedly traversed and cubes are refined depending upon surface intersection and curvature.

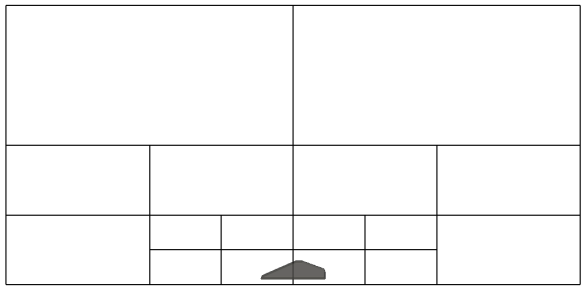
For each cube, a list of facets that intersect the cube is generated from the list of facets that intersected the parent cube using Moller's triangle-box overlap algorithm [8]. If the list is non-zero, then this cube intersects geometry and is tagged as 'cut' and refined. As the tree grows, this test for intersection becomes more efficient as this is effectively partitioning the geometry to the cubes and the list of facets to be tested for intersection become smaller in length.

The refinement is repeated until the target small size for surface resolution is reached (Figure 1(a)-(b)). To determine if a cube should be refined further due to high surface curvature then the Gauss-Bonnet scheme is used to compute the Gaussian curvature [9]. This utilises the list of facets that are already known to intersect the cube and so can proceed independently in each cube. This extra refinement can be seen at the corners of the model in Figure 1(e)-(f).

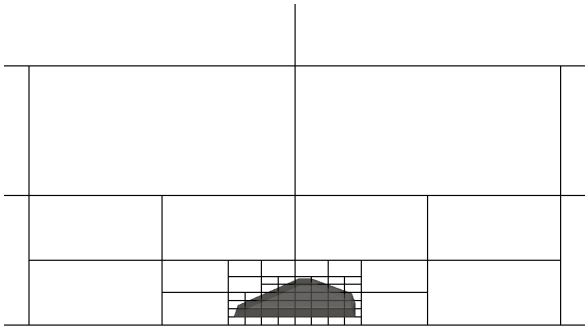
Generally, a large jump in grid size between cubes is detrimental to the numerical accuracy of the spatial discretisation of fluxes and it would be preferable if there is no more than a factor of two jump in linear cell size between cubes. The simple refinement process can produce large mismatches in size between cubes neighbouring in physical space (for example Figure 1(b)). To correct this, the cubes are traversed checking adjacent cubes for more than one level of mismatch - if this is found then the coarser cube is refined (Figure 1(c)). This may then lead to more mismatches in levels, so this is repeated until the traversal does not produce any refinements.

At this stage, some cubes will have been tagged as 'cut', the remaining 'unknown' cubes do not intersect geometry, but could be totally inside or outside of geometry (e.g. 'solid' or 'fluid'). There is then a traversal over all unknown cubes using a fast ray tracing algorithm [10] to set cube status as 'solid' or 'fluid'. A bi-directional ray is sent in an arbitrary direction from the centre of the cube and the number of intersections with surface facets is counted. If the origin of the ray is inside a solid then there will be an odd number of intersections, and if it is outside the solid there will be an even number of intersections. Both directions of the ray should agree as to whether there is an even or odd number of intersections: a discrepancy indicates a potential problem with gaps in the faceted geometry and the calculation is repeated with another arbitrary ray until agreement is found. Any cubes that are 'solid' are then removed (Figure 1(d)).

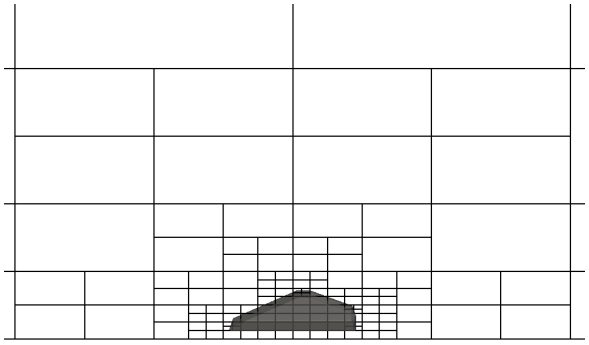
For the remaining intersecting cubes, we need to know which finite volume cells within



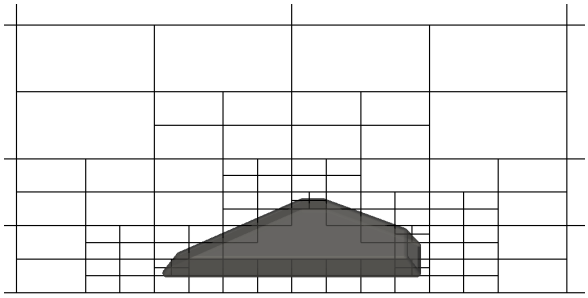
(a) 3 level refinement



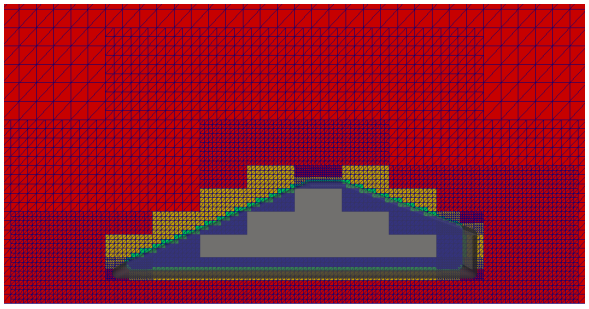
(b) 8 level refinement



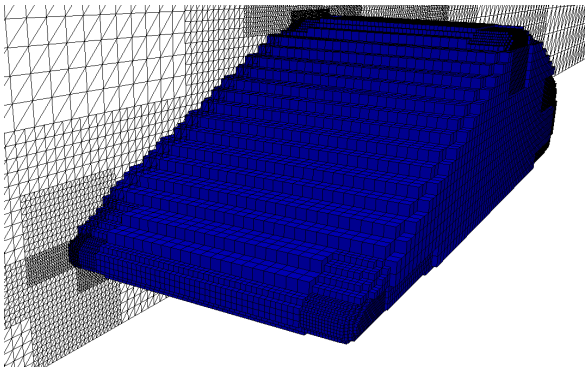
(c) fix level mismatch



(d) remove cubes within solid



(e) computational cells (blue: solid cell; green: surface intersection cell; yellow: fluid cell in cube that intersects surface; red: fluid cell in non-intersecting cube)



(f) solid/fluid boundary

Figure 1: Grid generation process

the cube intersect the surface geometry and this information will then need to be stored as a three-dimensional array. A naive use of ray-tracing for all cells within a cube will be computationally too expensive, this is particularly true for complex geometry with large numbers of facets. Ishida et al. [7] used a fill algorithm starting from a user defined point within the fluid. However, a fill algorithm inherently contains dependencies, the state of a cube is determined from the state of its neighbours, and within a cube each cell state will depend upon neighboroughing cell or cube states. This makes the pure fill algorithm a scalar process. In this work, a hybrid scanline fill is utilised within each cube. The cells of the cube are split into scanlines which are bounded by cube boundaries or geometry intersection (i.e. every cell in the scanline must be of the same type). The whole of this scanline can be set to 'solid' or 'fluid' by examining the status of cells adjacent to the scanline. If there are no known adjacent cells then we finally resort to a ray-tracing evaluation to determine the status of the scanline. A major advantage of this approach is that there are no dependencies between cubes and each can be computed simultaneously; however, it does mean that the computational work will be influenced by the number of facets describing the geometry.

2.3 Shared Memory Parallel

Until recently, each year CFD became faster to solve as computer processors dramatically increased in clock speed. But, processors have now reached a limit of clock speed, and the benefits of improved fabrication processes are being used to increase the *number* of processor cores within a single package. Whilst CFD solvers can utilise the multiple core processors by using the parallel techniques developed for running on compute clusters, the grid generation process also needs to exploit parallelism to keep pace.

Each core in the processor package has equal access to the system memory and so for these types of computer architecture it is a natural fit to used shared memory parallelism. In this work we have implemented shared memory parallelism using OpenMP. This uses compiler directives to indicate loops that can be computed in parallel; at run-time the loop is distributed across the processor threads and at the end of the loop returns to a single thread. This was straightforward to implement with most loops over the cubes being targeted to run in parallel. As the octree data structure is shared across threads, a critical region is used for locations where children are being added to the octree due to cube refinement.

As will be seen later, this produces good parallel speed-up and is totally transparent to the user - the grid generation automatically utilises all processor cores available on a given machine.

2.4 Distributed Memory Parallel

The ultimate aim of this work is to produce a system to solve LES on complex geometries with extremely large meshes. Since the flow solver will need to use distributed

memory parallelism and we wish to avoid the separation of grid generation and solver, then logically the grid generation needs to also use distributed memory parallelism.

One difficulty of parallelising grid generation is that it needs to be dynamically partitioned across the processors as the grid is created. In this work, as the octree is grown the leaf nodes and associated cubes are allocated to processes dynamically – the end result of the grid generation is not only a grid but a partitioning of the cubes across processors ready for the solver to carry out a parallel computation.

All processes have a complete copy of the geometry and generate the same octree data structure independently. However, leaf nodes of the octree are 'owned' by a process. The 'owning' process computes tasks such as intersection with geometry and then this status information is synchronised across all processes using a broadcast. Thus, if a cube is removed because it is within a solid, this will occur on all processes' copies of the octree data structure. Similarly, where significant data on the grid is stored, such as the three dimensional array of cells within a cube denoting solid, fluid or cut, this only exists in the owning process and is not duplicated. When iterating over the leaf nodes, the iterator for a process uses a list of nodes that are owned by that process.

One of the key issues is to be able to dynamically partition whilst the grid is growing, whilst achieving good load-balancing and a locality of intercommunication for the solver. Of significant benefit is that each fluid cube will have an identical number of cells and so will have equal work for the flow solver. For intersecting cubes then we will have a mixture of cells with fluid, solid and intersect status. For a given solver, there will be an extra computational cost of the special treatment of the cut cells, but this could be balanced by the solid cells that will not need any computation. The hope is that intersecting cubes will be less expensive for the flow solver to compute and thus will not increase overall computation time – this needs to be tested with real flow solvers.

It is assumed that we may be generating more cubes than the number of processes available so it is not just a procedure to allocate each new cube to its own process. Typically ratios of cubes to processes are likely to be around a factor of ten. As a cube is refined eight new child nodes are generated which need to be assigned to a process, and the parent node which is removed from a process and becomes globally owned. Since these cubes belong to the children are physically adjacent, for good flow solver partitioning it would be useful to give them to the same process and so we typically assign parcels of four or eight new leaf nodes and their cubes to a single process. The candidate process is found by determining the process with the lowest loading, and if there is more than one with the same low loading then using the lowest rank process. One small modification is that if the old owning process of the parent node is of equal low loading then this is used in preference. The aim is to improve the locality of partitioning in physical space, but this needs testing on real flow solutions to determine if it is of benefit in practice.

The distributed memory parallelism is implemented using the Message Passing Interface (MPI) standard and can run on a wide variety of systems.

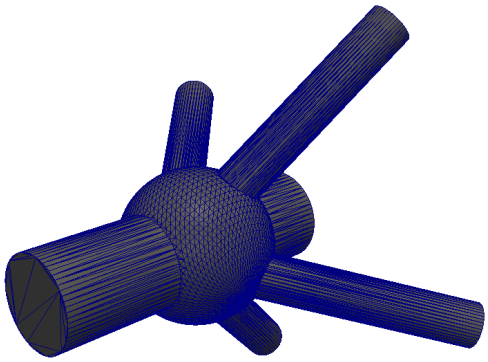
3 RESULTS

The algorithm has been tested on a range of input CAD geometries. Some of these are shown in Figures 2 and 4(a). These were chosen to test a variety of requirements. The simple geometry and Davis model both have well defined radii of curvature which are used to test the Gaussian curvature calculation and subsequent local curvature refinement. The Spitfire model has a small gap between the rudder and the fin; for a coarse grid this is not resolved and the CFD grid contains a single solid object, whereas as the grid is made finer, the gap becomes resolved and there are separate solid objects in the CFD domain (see Figure 3). In addition the Spitfire is placed in a nose up and yawed attitude so that the grid axes are not aligned with the predominant aircraft directions. The abstract geometry is chosen as this contains long thin triangular facets which will highlight flaws in the triangle-box overlap algorithm and a lattice like structure that will test the ray-tracing determination of solid or fluid. To test sensitivity to the number of triangular facets in the geometry definition, the abstract geometry only contains 888 facets whilst the landing gear geometry has 350,000 facets.

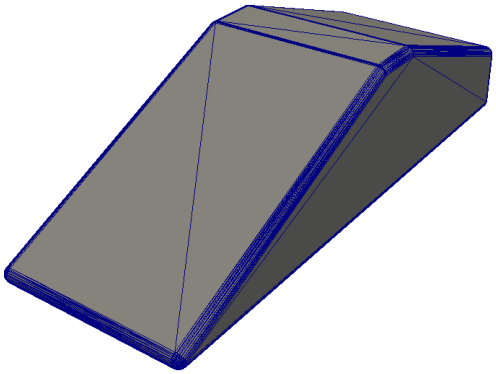
The landing gear case is shown in more detail as this is probably the most challenging of the test cases and is also an application area of current interest: LES can be used to predict unsteady shedding and hence noise production [11]. As can be seen in Figure 4(a), there is considerable detail resolved in the model, such as the grooves in the tyres, cavities and gaps in the wheel hub and multiple pivots due to the retraction mechanism. No modifications were made to the STL model, the grid generator was run with default parameters to produce results shown here in a matter of minutes. It would be inconceivable to use a multiblock structured grid generator for this case, and even state of the art unstructured grid generators would require considerable intervention and probably hours of calculation to produce grids of order 100 million elements.

An overall view of the domain is shown in Figure 4(b). The outer domain is set to be four characteristic lengths away from the maximum extents of the model geometry and there are seven levels of refinement - the cubes around the geometry become extremely small and on this diagram appear as a dark mass at the centre of the domain. This shows the ability of the method to give good resolution around the object without incurring the penalty of the refinement spreading to the far-field, as would be the case with simple structured grids. This is of particular importance to LES where we need to avoid high aspect ratio cells. The domain contains 2353 cubes each with $15 \times 16 \times 15$ cells giving a total of 8.47 million cells. This is considerably less than the parallel testing example shown later which has 63.5 million cells and was chosen to allow easier visualisation of the cubes and cells.

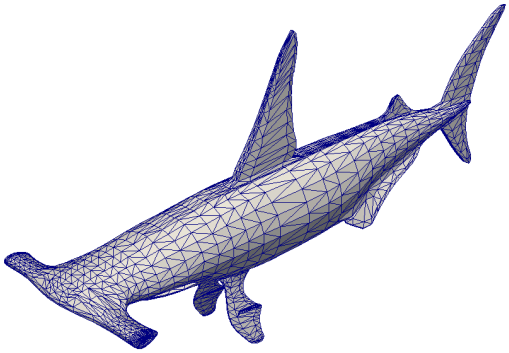
Figure 5(a) is a cut of the domain through the wheels and axle that illustrates the complexity of this case. The cells are coloured to distinguish those as intersected by the surface (green), solid (blue), fluid (red or yellow). The ray-tracing algorithm has found the inside of the tire and this is correctly marked as a fluid - even though it is completely



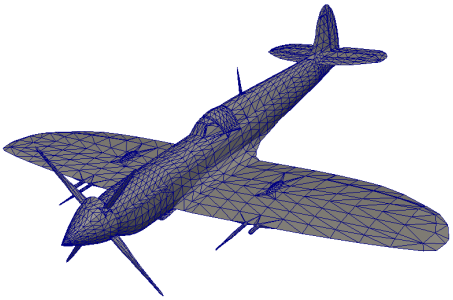
(a) simple geometry



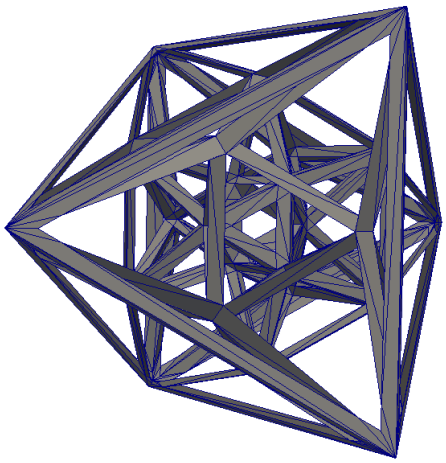
(b) Davis body



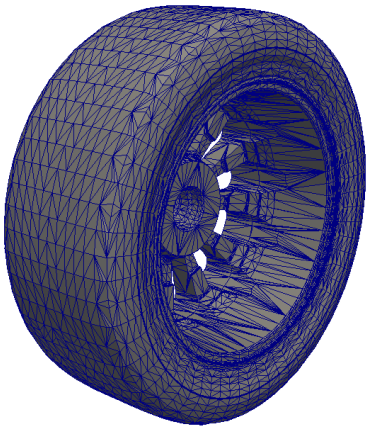
(c) shark



(d) Spitfire aircraft



(e) abstract geometry



(f) wheel

Figure 2: CAD geometries used for testing

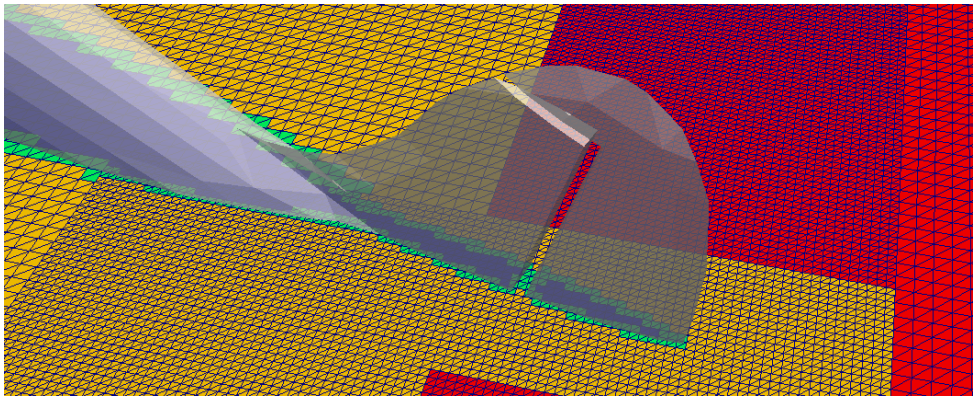


Figure 3: Spitfire fin and rudder gap (blue: solid cell; green: surface intersection cell; yellow: fluid cell in cube that intersects surface; red: fluid cell in non-intersecting cube)

number of:	facets	cubes	cells	fluid cells	cut cells
simple geometry	7,342	4,916	468,101,520	396,597,818	3,890,450
Davis body	1,852	3,954	164,407,320	135,432,458	2,172,250
landing gear	350,418	2,353	63,460,410	57,479,112	1,471,103

Table 1: Parallel testing cases

isolated from the fluid surrounding the wheel. Similar cavities are correctly marked within the wheel hub. This example shows both the strengths and weaknesses of the use of ray-tracing with a scanline fill within each cube: all regions are automatically marked, it is expensive to evaluate the ray-tracing as it scales with the number of facets, but this can be parallelised. An approach that takes a user defined seed point as being fluid, then flood filling all the cells would be cheaper to compute, but is more difficult to parallelise - and would need multiple user inputs to seed the fluid regions within the tyres.

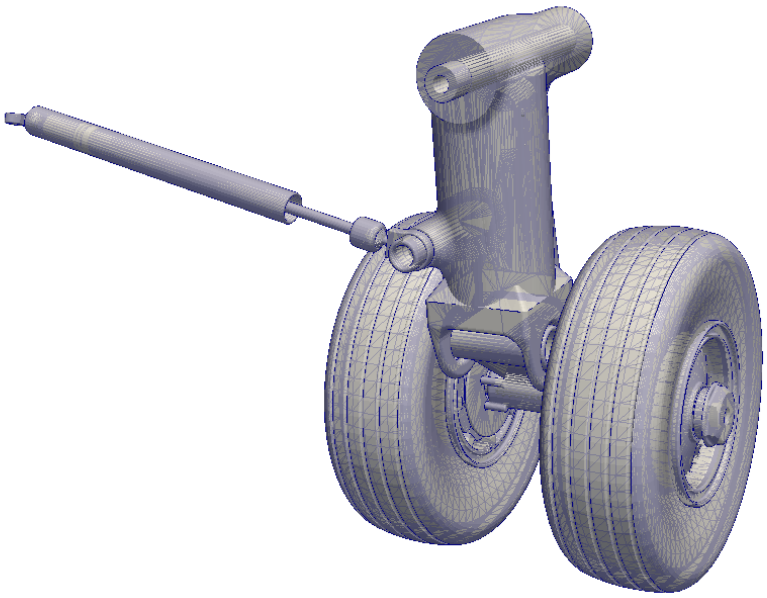
Finally, the surface created from the cells marked as intersecting the solid is shown in Figure 5(b).

3.1 Parallel Testing

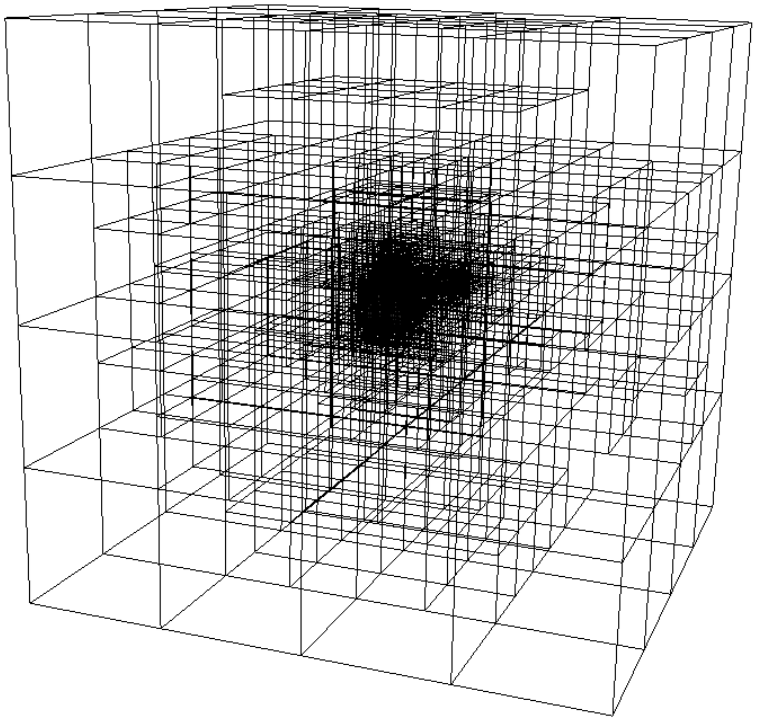
Three cases have been chosen to test the parallel speed up and efficiency. The cases are shown in Table 1 which summarises the number of facets in the CAD geometry and the resulting grid in terms of cubes and cells.

Testing was carried out on an Intel 'Nehalem' Xeon E5520 (2.26GHz) system. This was configured as two quad core processors per node, with nodes being connected by gigabit ethernet.

Plots of wall time for scalar, shared memory parallel (OpenMP) and distributed memory parallel (MPI) are shown in Figure 6 for up to 32 processor cores. Of note is the basic speed and efficiency of the algorithm: even running on a single processor core, a grid for the 'simple geometry' with 460 million cells can be generated in less than 100 seconds,

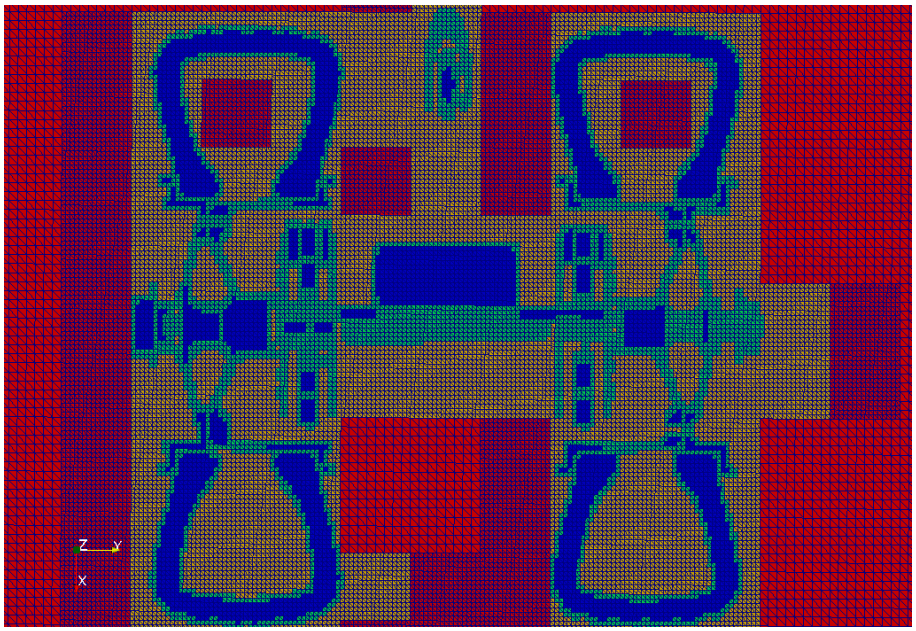


(a) STL CAD geometry

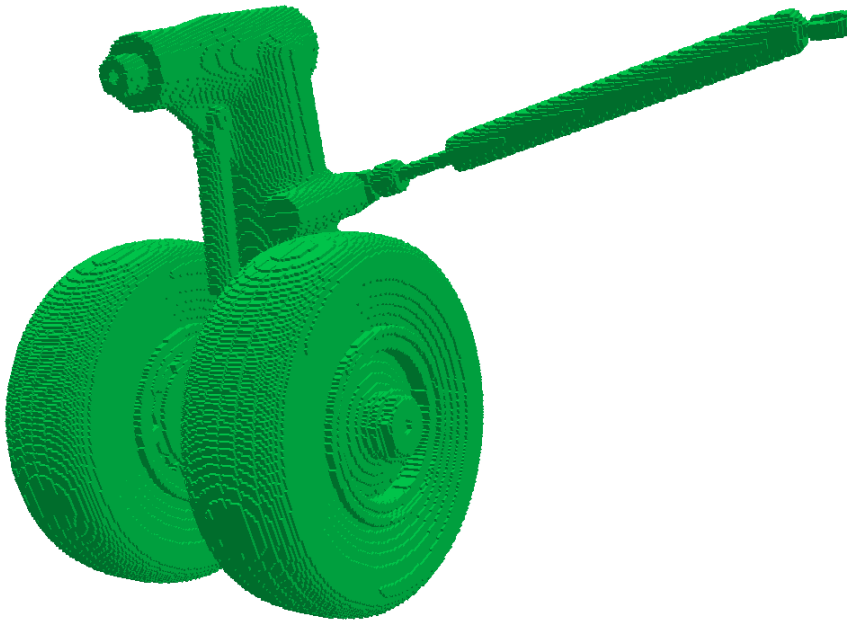


(b) cubes

Figure 4: Landing gear grid overall view

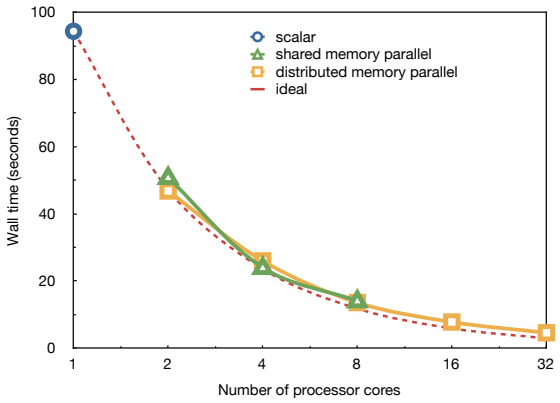


(a) cut showing computational cells (blue: solid cell; green: surface intersection cell; yellow: fluid cell in cube that intersects surface; red: fluid cell in non-intersecting cube)

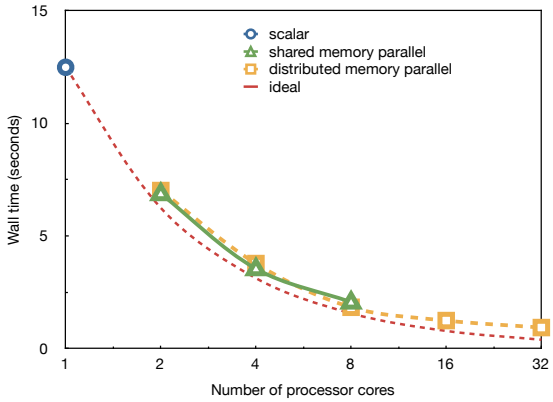


(b) solid/fluid surface

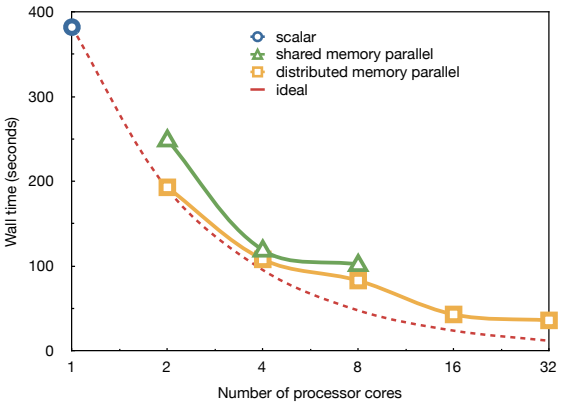
Figure 5: Landing gear grid detail view



(a) simple geometry



(b) Davis body



(c) landing gear

Figure 6: Grid generation wall time

simple geometry						
	number of cores	2	4	8	16	32
shared memory	speed up	1.86	3.93	6.69		
	efficiency	93%	98%	84%		
distributed memory	speed up	2.02	3.63	6.94	12.1	20.5
	efficiency	101%	91%	87%	76%	64%
Davis body						
	number of cores	2	4	8	16	32
shared memory	speed up	1.82	3.53	6.06		
	efficiency	91%	88%	76%		
distributed memory	speed up	1.78	3.31	6.82	10.1	13.4
	efficiency	89%	83%	85%	63%	42%
landing gear						
	number of cores	2	4	8	16	32
shared memory	speed up	1.53	3.21	3.75		
	efficiency	77%	80%	47%		
distributed memory	speed up	1.98	3.54	4.60	8.88	10.6
	efficiency	99%	88%	58%	56%	33%

Table 2: Parallel speed up and efficiency

and the maximum memory usage is 1.3GB. Running on 32 processor cores this reduces to 4.6 seconds. Of interest is the behaviour of the 'landing gear' case which is much slower than the other cases and exhibits anomalous behaviour on eight cores. This case differs from the first two in that the geometry definition has a large number of triangular facets. The ray-tracing algorithm cost scales with the number of facets and the complexity of the surface leads to more ray-tracing evaulations per cube. The net result is that compared to the 'simple geometry' it takes four times as long to generate a grid with 1/7 of the number of cells. It should be noted that this is perhaps better than may be expected considering that the geometry has 47 times more facets than the 'simple geometry'.

A more detailed analysis of the results are shown in Table 1. Parallel speed up and efficiency is generally good up to eight cores, but drops off at 16 and 32 cores for the distributed memory parallel. The main factor appears to be load balancing, the partitioning of the octree data structure as it is created is based upon even loading and reduced communication for the flow solver - it aims to achieve an equal load of cubes across processes with adjacent cubes being on the same processor core. However, for the grid generator, the computational load depends upon the number of intersecting cubes owned by a process. Examination of timings for individual processes showed a significant variation indicating poor load balancing. This may be an acceptable compromise as it is more important that the flow solver achieves good load balancing – this requires further testing, particularly on much larger number of processor cores. The load balancing problem does not explain

the anomalous result for eight cores on the 'landing gear' case as this occurs for both distributed and shared memory runs. For shared memory there is no explicit partitioning of the cubes as each parallel loop over the cubes in the domain is evenly distributed over the processes at run-time.

The implementations of both the shared and distributed parallelism are relatively simple and with further testing and instrumentation could be tuned for better parallel speed up. Nevertheless, both approaches achieve extremely fast grid generation, and in the case of the distributed memory also achieves a partitioning of the grid ready for the flow solver.

4 CONCLUSIONS

A fast and efficient parallel grid generation method using an octree data structure has been described. Testing on a variety of input CAD geometries has demonstrated the speed and robustness. Reasonable parallel speed ups have been achieved on both shared and distributed memory implementations. The computation time is strongly dependent upon the number of facets used to resolve the geometry.

The main area that needs addressing is the algorithm to determine solid or fluid within each cell of a cube that intersects the geometry. A deliberate choice of an algorithm that can be parallelised does make the computational cost increase with the number of facets due to the ray-tracing evaluation. The partitioning of the cubes across the processor cores is based upon good load balancing for the flow solver and can be poorly balanced for the grid generator. Testing also needs to be carried out on parallel systems with hundreds or thousands of processor cores.

The method shows considerable promise and now needs to be integrated with an LES flow solver to demonstrate its true capability.

REFERENCES

[1] D. De Zeeuw and K.G. Powell, An adaptively refined Cartesian mesh solver for the Euler equations, *Journal of Computational Physics* **104** 56-68 (1993) .

[2] M. J. Aftosmis, M. J. Berger, G. Adomavicius, A parallel multilevel method for adaptively refined Cartesian grids with embedded boundaries, In proceedings of the *38th Aerospace Sciences Meeting and Exhibit*, 10-13 January 2000, Reno, Nevada. AIAA-2000-0808 (2000).

[3] Y-H Tseng and J.H. Ferziger, A ghost-cell immersed boundary method for flow in complex geometry, *Journal of Computational Physics* **192** 593-623 (2003).

[4] J-E Emblemsvag, R. Suzuki, G.V. Candler, Cartesian grid method for moderate-Reynolds-number flows around complex moving objects, *AIAA Journal* **43** (1) 76–86 (2005).

[5] S. Kang, G. Iaccarino, P. Moin, Accurate immersed-boundary reconstructions for viscous flow simulations, *AIAA Journal* **47** (7) 1750–1760 (2009).

[6] T. Kamatsuchi, Flow simulation around complex geometries with solution adaptive Cartesian grid method, In proceedings of the *18th AIAA Computational Fluid Dynamics conference*, 25-28 June 2007, Miami, Florida. AIAA-2007-4189 (2007).

[7] T. Ishida, S. Takahashi, K. Nakahashi, Efficient Cartesian mesh approach for flow computations around moving and deforming bodies, In proceedings of the *19th AIAA Computational Fluid Dynamics conference*, 22-25 June 2009, San Antonio, Texas. AIAA-2009-3879 (2009).

[8] T. Akenine-Möller, Fast 3D triangle-box overlap testing, In proceedings of the *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, (Los Angeles, California, July 31 - August 04, 2005). J. Fujii,. doi http://doi.acm.org/10.1145/1198555.1198747 (2005)

[9] T. Surazhsky, E. Magid, O. Soldea, G. Elber, E. Rivlin, A comparison of Gaussian and mean curvatures estimation methods on triangular meshes, In proceedings of the *IEEE International Conference on Robotics and Automation* **1** 1021–1026 (2003).

[10] T. Möller and B. Trumbore, Fast, minimum storage ray/triangle intersection, In proceedings of the *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, (Los Angeles, California, July 31 - August 04, 2005). J. Fujii, Ed., doi http://doi.acm.org/10.1145/1198555.1198746 (2005).

[11] Y. Li, R. Satti, P-T Lew, R. Shock, S Noelting, Computational aeroacoustic analysis of flow around a complex nose landing gear conguration, In proceedings of the *14th AIAA/CEAS Aeroacoustics Conference*, 5–7 May 2008 June 2009, Vancouver, British Columbia. AIAA-2008-2916 (2008).