ADVANCES IN PARALLEL OVERSET DOMAIN ASSEMBLY

By

Cameron Thomas Druyor Jr.

Steve L. Karman, Jr.
Advance Research, Pointwise Inc.
(Chair)

James C. Newman III
Professor of Computational Engineering
(Committee Member)

William T. Jones
Computer Engineer, NASA Langley
Research Center
(Committee Member)

Craig R. Tanis
Assistant Professor of Computer Science
(Committee Member)

ADVANCES IN PARALLEL OVERSET DOMAIN ASSEMBLY

By

Cameron Thomas Druyor Jr.

A Dissertation Submitted to the Faculty of the University of
Tennessee at Chattanooga in Partial Fulfillment of
the Requirements of the Degree of Doctor of
Philosophy in Computational Engineering

The University of Tennessee at Chattanooga
Chattanooga, Tennessee

August 2016

Copyright © 2016

By Cameron Thomas Druyor Jr.

All Rights Reserved

ABSTRACT

The CFD (Computational Fluid Dynamics) community has long used the Overset Grid method to enable dynamic simulations with bodies in relative motion. In Overset simulations, information is transferred between overlapping grids via interpolation. Domain Assembly is the process that governs the location of intergrid boundaries and how the solution is interpolated across grids at those boundaries. Performing Domain Assembly in a distributed environment is computationally expensive and inherently poorly load balanced due to the solver partitioning. Dynamic load balancing is therefore required to alleviate the imbalance and make very large Overset problems feasible. In this work, a radically different parallel domain assembly method is introduced. The new method takes a fundamentally different approach to load balancing, concurrency, and communication patterns. A detailed discussion is provided that describes the method's implementation in YOGA (Yoga is an Overset Grid Assembler). Finally, several case studies are analyzed and preliminary performance and scaling results are provided.

DEDICATION

This work is dedicated to my family. Everything I have accomplished is built on the foundation of their love and support.

ACKNOWLEDGEMENTS

I would like to acknowledge my colleagues at NASA Langley for providing me a supportive and growth oriented environment. I would like to thank my committee for seeing my project through to the end, despite being spread across three states. I am particularly grateful for Dr. James Newman III for all of his hard work to make sure that so many students could graduate at the same time.

TABLE OF CONTENTS

ABSTRACT iv

DEDICATION v

ACKNOWLEDGEMENTS vi

LIST OF TABLES x

LIST OF FIGURES xi

CHAPTER

1. INTRODUCTION 1

1.1 Objective 3
1.1.1 Challenges 3

2. LITERATURE REVIEW 5

2.1 The Overset/Chimera Method 5
2.1.1 Grid Types 8
2.2 History of the Overset Grid Method 10
2.2.1 Overset Structured Grids 10
2.2.2 Overset Unstructured Meshes 21
2.3 Current State of the Overset Grid Method 28
2.3.1 Computational Cost 29
2.3.2 Geometric Searching Tools 30
2.3.2.1 Bounding Boxes 30
2.3.2.2 Auxiliary Meshes 32
2.3.2.3 X-rays 33
2.3.2.4 Spatial Trees 35
2.3.3 Load Balancing Strategies 36
2.3.3.1 Spatial Repartitioning 37
2.3.3.2 Dynamic Load Balancing 38

2.3.4 PUNDIT: Current State of the Art 39
2.3.4.1 Load Balancing Implementation 41

3. METHODOLOGIES AND ALGORITHMS 42

3.1 Monte Carlo Investigation of Load Balancing Strategies 42
3.1.1 The Simulator 43
3.1.1.1 Determining Standard Deviation 43
3.1.1.2 Generating work units 44
3.1.1.3 Running the simulator 45
3.2 Simulation Results 46
3.2.1 Predictive Load Balancing 46
3.2.2 Load Balancing via Over-Decomposition 49
3.2.3 Client-Server Load Balancing 50
3.3 Recasting Domain Assembly 52
3.3.1 Latency Hiding via Thread Oversubscription 53
3.3.2 Defining Appropriate Work Units 54
3.3.3 Processing Work Units 55

4. IMPLEMENTATION 56

4.1 Introduction to Stream and ZeroMQ 56
4.1.1 MessagePasser::Stream 56
4.1.2 ZeroMQ 57
4.1.2.1 Generic Server 58
4.1.2.2 Sending and Receiving Streams 62
4.1.2.3 Generic Client 63
4.2 PreProcessing 65
4.2.0.4 Global Mesh System Meta Data 65
4.2.0.5 Approximate Distance Field 65
4.2.1 Load Balancer 66
4.2.1.1 Estimating Mesh Density 70
4.2.2 Parallel Hole Map 71
4.2.2.1 Implicit Outer Boundary 71
4.3 Donor Searching with Dynamic Load Balancing 73
4.3.1 Voxel Server 73
4.3.2 Worker 75
4.3.2.1 Building A Work Voxel 76
4.3.2.2 Finding donors and identifying hole points 78
4.3.2.3 Distributing 80
4.3.3 Grid Server 81
4.3.4 Donor Collector Server 82

4.4 Post Processing 83
4.4.1 Interpolation in Yoga 83
4.4.1.1 Math 84
4.4.1.2 Implementation 84
4.4.1.3 Verification 85
4.4.2 Tracer 86

5. RESULTS 88

5.1 Store Separation Case 88
5.2 Hart II Rotor Case 92
5.3 Distributed Electric Propulsion Case 96

6. CONCLUSION 102

6.1 Technology Exploration 102
6.2 Future Work 103
6.2.1 Features 103
6.2.2 Performance 103

REFERENCES 104

VITA 107

LIST OF TABLES

5.1 Effect of dynamic work unit sizing 91

5.2 Timing results 95

5.3 Timing results (K cluster) 101

5.4 Timing results (Pleiades) 101

LIST OF FIGURES

1.1 Wing and store geometry 2

2.1 Explicit cut on a structured grid 7

2.2 Structured meshes 8

2.3 Unstructured Mesh Examples 9

2.4 Level curves in a structured mesh 11

2.5 Geometric cutters 12

2.6 Stencil-walk method 12

2.7 Inverse map 13

2.8 Quadtree 15

2.9 Hole map 15

2.10 Issue with bodies in close proximity 16

2.11 X-rays 19

2.12 Organizing query points 21

2.13 Unstructured neighbor walking 22

2.14 Implicit cut based on distance to the wall 23

2.15 Bounding boxes around a store geometry 27

2.16 Non-overlapping bounding boxes 31

2.17 Overlapping bounding boxes 32
2.18 Oriented bounding boxes 32
2.19 X-rays on store geometry 34
2.20 Adaptive x-rays 35
2.21 1-D ADT 36
2.22 Spatial Decomposition Volume 38
3.1 Normal distribution 45
3.2 Sample load 1 46
3.3 Sample load 2 47
3.4 Imbalance trend 48
3.5 Over decomposition trend 1 49
3.6 Over decomposition trend 2 50
3.7 Dynamic trend 1 51
3.8 Dynamic trend 2 52
4.1 Hole map memory reduction 73
4.2 Interpolation convergence 86
4.3 Tracing screenshot 87
5.1 Store separation surface meshes 88
5.2 Hole map for store separation 89
5.3 Wing mesh slice 90
5.4 Store mesh slice 90

5.5 Improved load balancing 92

5.6 Hart II geometry 93

5.7 Fuselage slice 1 94

5.8 Fuselage slice 2 94

5.9 Slice near hub 95

5.10 Joby Geometry 96

5.11 Joby multi slice 98

5.12 Fairing 1 slice 99

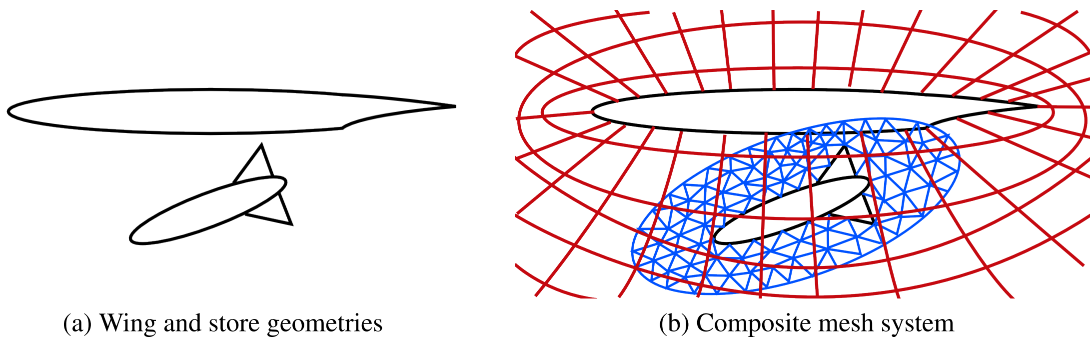
CHAPTER 1

INTRODUCTION

A typical finite-volume computational fluid dynamics (CFD) solver uses a single, contiguous mesh to discretize the computational domain (structured solvers typically have multiple blocks, but they are abutting and form a single contiguous domain). This requirement causes two problems. First, generating a single grid for a complex domain, particularly using a structured grid technique, can be challenging. Second, and perhaps more importantly, solving problems with multiple bodies in relative motion is difficult on a single grid unless the relative motion is small. However, both of these issues can be alleviated if the computational domain is spanned by multiple grids that are allowed to overlap forming a composite patchwork of grids covering the entire domain. The methodology of overset, or overlayed, grids is often referred to as the Chimera [1] technique due to its hybrid approach combining grids of separate components. With this approach, overall grid generation can be simplified and grid quality can be improved because individual bodies, or even components of complicated bodies, can be meshed separately. When considering moving bodies in relative motion, each body can be represented by its own grid. As such, the entire grid for a given body is allowed to move with the body independent of the grids of other bodies. This provides for robust support of the full range of motion within the domain.

The advantages of overset grids do not come for free, however. A single, continuous solution is still desired for the domain, but the domain is covered by a patchwork of unrelated grids.

These component grids must be associated with one another, in a process known as domain assembly, such that information can propagate across the domain. Numerous techniques and methodologies have been developed to perform domain assembly. However, the process is computationally expensive and can represent a large portion of the overall simulation cost for dynamic simulations with bodies in relative motion, because the domain must be reassembled between each time step. Additionally, the process has proven difficult to efficiently parallelize. Therefore, domain assembly is still an active area of research. the concept, consider a simple store separation problem as shown in Fig. 1.1a, where the wing and the store each has its own grid as shown in Fig. 1.1b. Because the bodies have their own individual meshes and the domain assembler facilitates information transfer between them, the bodies can move independently.



(a) Wing and store geometries

(b) Composite mesh system

Figure 1.1 Wing and store geometry

Each component grid contributes to the solution, so there must be some mechanism for information to pass between component grids. Because fluid dynamics problems are boundary value problems by construction, boundary conditions can be used to control how information enters and leaves the computational domain. A flow solver may have multiple boundary conditions to address

different physical constraints, such as inlet, outlet, wall boundary, or constant pressure conditions. Carefully chosen boundary conditions can also be used to transfer information between component grids in overset simulations, but this raises the following questions. How is the boundary information obtained, and where should the boundaries be defined? In essence, assembling overset domains is simply answering both of these questions. Unfortunately, neither of these questions can be answered without performing 3-Dimensional (3D) geometric searches. These searches can be quite expensive, particularly in a parallel context on partitioned domains.

1.1 Objective

The primary objective of this work is to develop a new dynamic load balancing strategy for performing domain assembly that scales to accommodate large mesh systems and large numbers of processors. Other domain assembly codes exist, but only one existing code scales beyond a small number of processors. The present work introduces a new load balancing strategy that is fundamentally different from any existing domain assembly code.

1.1.1 Challenges

In serial, donor searching —finding which cells contain a given point— drives the cost of domain assembly. In a distributed environment, potential donors for a given point will live on different processors, so the problem becomes much more difficult. In addition to performing the actual donor search, the domain assembler must somehow gather the necessary grid information from different processors. A particular partition may overlap with any number of partitions on other processes, so the domain assembler must determine what grid information needs to be transferred

between processors (and perform the communication) in order to set up donor searches. The amount of donor searches a particular processor will perform is unknown at the outset of the domain assembly. Furthermore, the work will be dependent on how the component grids overlap, so some processes will perform a large number of searches, and some processes will perform very few. Therefore some kind of load balancing must be introduced to insure that all processors have work to do. Even when load balancing is introduced, the actual cost of the work for each processor is difficult to predict, so significant imbalances can still occur. Roget and Sitaraman [2] introduced a dynamic load balancing scheme that measures the actual cost for each process and uses the measurement to improve load balancing for domain assembly in subsequent time steps (i.e., load balancing occurs one time per domain assembly based on "yesterday's weather"). In contrast, the method proposed in this work performs load balancing continuously *during* each domain assembly via a combination of over decomposition and a client server model. The proposed method is therefore fundamentally different from any existing domain assembly method.

CHAPTER 2

LITERATURE REVIEW

2.1 The Overset/Chimera Method

The Chimera technique simplifies the grid generation process by allowing component grids to be created independently. There is a range of restrictions associated with using overset grids, however, so care must still be taken when creating the individual grids. For example, grids should be of similar resolution in regions of overlap to minimize interpolation error. Additionally, regions of overlap must be sufficiently large. Specifically, enough points must exist in the region to build interpolation stencils of the appropriate size for the desired interpolation scheme. However, the flexibility afforded by allowing multiple component grids to cover a geometric body makes it much easier to create grid systems for large scale complex configurations. For example, the Overset method made it possible to perform a complex simulation of the integrated Space Shuttle vehicle (orbiter, solid rocket bosters, and external tank). [3]

The Chimera method affords a great deal of flexibility during the grid generation process, but it does impose some additional requirements on the flow solver. First, the flow solver must support a new interpolation boundary condition that allows information to propagate from one component grid to the next. Second, the flow solver must be able to exclude certain control volumes from computation that are designated to be outside the computational domain. An external domain assembly process is responsible for providing the solver with a classification for each

control volume and the identities and weights of *donors* that the solver needs to calculate the interpolated value. Because the driving concepts of the overset method are applicable to both cell centered and cell-vertex centered schemes, the author has chosen to use the term control volume to preserve generality and avoid confusion.

Control volumes are classified according to their update method. Control volumes that are internal to the computational domain are referred to as *solve* or *in* control volumes because the flow solver solves for their values just as it would for control volumes on a contiguous grid. Control volumes that are designated as external to the computational domain are referred to as *hole* or *out* control volumes. Control volumes on the boundary between *solve* and *hole* regions are designated as *receivers*, *receptors*, or *fringes* and are updated by interpolation. Control volumes that are the source of interpolated solution data for receivers are termed *donors*, and receivers for which an appropriate donor has not been found are termed *orphans*.

There are two broad categories of Overset techniques: explicit and implicit hole cutting. Explicit hole cutting uses the geometry to cut a hole in any mesh that overlaps with the geometry, and Implicit hole cutting uses some cell-wise criterion to determine where holes should be cut (e.g., cell volume). Figure 2.1 demonstrates an explicit hole cut in a structured curvilinear grid by a circular piece of geometry. The domain assembler has identified nodes that are internal to the geometry and marked them as hole points. The assembler also marked the nodes along the hole boundary as receivers and the remaining nodes as solve points. The solver will update the solution at the receivers with interpolated information from the component grid associated with the circular geometry (not shown). Note that the nodes classified in this illustration are appropriate for a cell-vertex centered scheme. If the grid was being prepared for a cell centered scheme, the

classification would be done on a cell-wise basis instead. Using the information provided by the domain assembler, the flow solver will ignore all of the points in the hole region, and solve for values at all of the points in the solve region. The hole boundary points, which are marked as receivers, will be updated at each iteration by using the interpolated solution values from the grid that is associated with the circle geometry. In this case, there is only one fringe layer, but there can be multiple fringe layers (e.g., second order or higher interpolation requires two or more layers for finite volume discretizations).

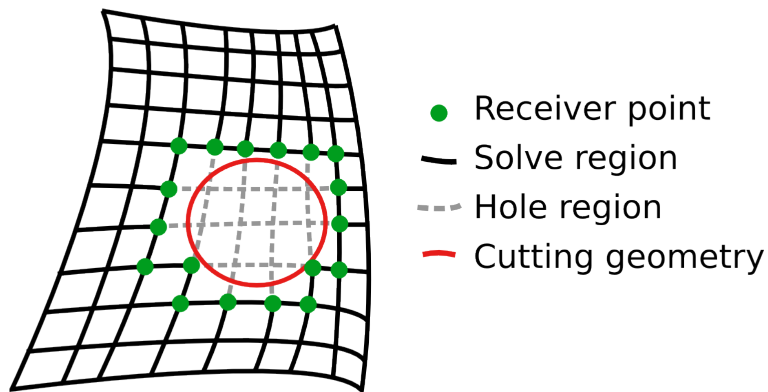
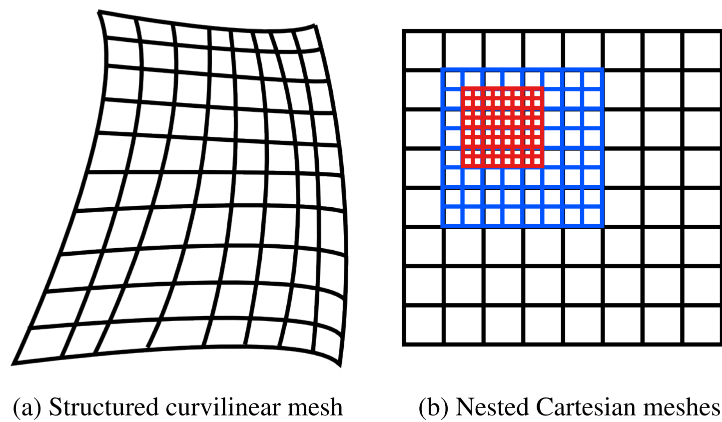


Figure 2.1 Explicit cut on a structured grid

The domain assembler is responsible for determining the status of each control volume, identifying donors, and calculating weights for each of the donors. These tasks require 3D geometric searching operations to find control volumes in different component grids that live in the same physical neighborhood. Geometric searching in 3D can be expensive, and is, in fact, the most computationally intensive aspect of the assembly process. [2], [4] Geometric searching algorithms typically use either a spatial data structure or connectivity information to accelerate the search.

2.1.1 Grid Types

Meshes can be lumped into two broad categories: structured and unstructured. Structured meshes, as shown in Fig. 2.2, have an implied cell ordering and typically come in two forms: curvilinear (Fig. 2.2a) and Cartesian (Fig. 2.2b). All structured meshes are defined by the number of nodes along each axis in computational coordinates (ξ, η, ζ) and a metric of transformation to physical coordinates (x, y, z). Uniform Cartesian meshes are special cases whose transformation metrics are unity i.e., the physical coordinates are the same as the computational coordinates. This property makes geometric searching trivial on Cartesian meshes, i.e., $O(1)$. Figure 2.2b specifically depicts Adaptive Mesh Refinement (AMR), [5] which involves multiple Cartesian meshes of different resolutions nested inside each other. Both curvilinear and Cartesian meshes are structured, and they often appear together; but the author makes the distinction for clarity.

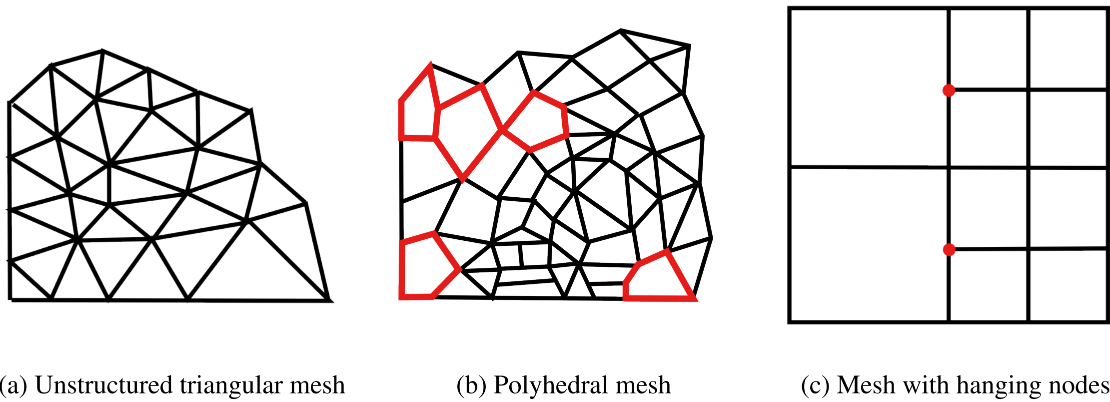


(a) Structured curvilinear mesh (b) Nested Cartesian meshes

Figure 2.2 Structured meshes

The term "unstructured mesh" is a vague term that simply means that the mesh does not have an implied cell ordering, so some auxiliary mapping must exist to associate cells with each

other. This can cause some confusion when the term is intended to mean something more specific, so again the author makes the distinction where necessary for clarity. Most of the time, researchers specifically mean either a mesh with only tetrahedral elements, or a mesh with tetrahedra, pyramids, prisms, and hexahedra, i.e., the "four basic" element types. Using this subset of element types restricts the number of faces, edges, and nodes that a given element in a mesh can contain. Figure 2.3a shows an unstructured mesh that contains only triangles. An arbitrary polyhedral mesh, i.e., a general unstructured mesh, does not place restrictions on the number of faces an element can contain. The mesh in Fig. 2.3b shows a general unstructured mesh, which has non-basic elements highlighted in red (in 2D, the two basic element types are triangles and quadrilaterals). All of the cells in the mesh in Fig. 2.3c are quadrilaterals, but hanging nodes are introduced where cell faces do not match one-to-one. Meshes generated by using hierarchical techniques have these features.



(a) Unstructured triangular mesh (b) Polyhedral mesh (c) Mesh with hanging nodes

Figure 2.3 Unstructured Mesh Examples

2.2 History of the Overset Grid Method

Benek et al. [1] first introduced the Chimera grid technique (now commonly known as Overset) in the early 1980s for steady state problems on multi-block structured grids. The method uses interpolation to transfer solution data between overlapping grids. The method was later extended to unstructured meshes and time-dependent problems with bodies in relative motion. The method can be expensive in terms of computational cost and user input (particularly for time dependent problems with bodies in relative motion). Since its inception, the overset method has been adapted to solve a wide variety of engineering problems and has evolved to meet the demands of drastic changes in computer hardware and flow solver methods. The remainder of this section is dedicated to describing many contributions that have helped increase automation, reduce computational cost, and evolve the overset method.

2.2.1 Overset Structured Grids

In the original overset implementation, Benek et al. [1] created hole boundaries in component grids explicitly. They first created level curves around pieces of geometry by marching along the computational axis that was normal to the surface as shown in Fig. 2.4. Once the boundary curve was constructed, a search was required to determine which points were inside the boundary curve, i.e., the points that will be cut. This method produced very accurate holes, but the process can become expensive for complex systems in three dimensions.

The initial goal of the overset method was to reduce the complexity of the grid generation process, but it was later extended by Meakin and Suhs [6] to unsteady simulations of multiple bodies in relative motion. Using overset grids allowed the individual bodies to move independently

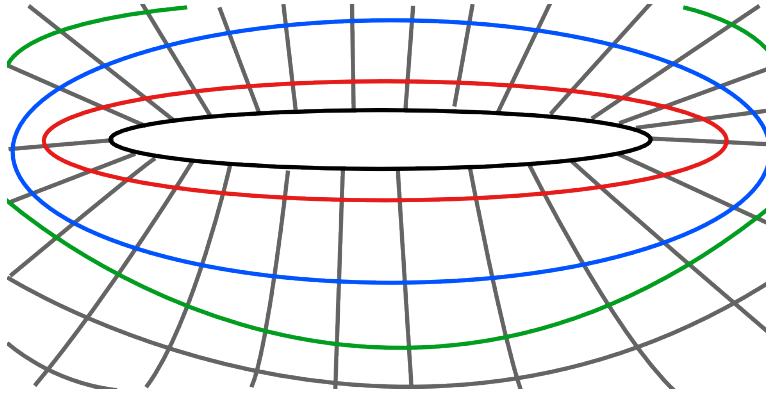
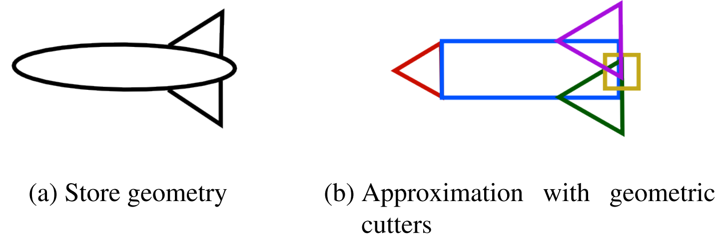


Figure 2.4 Level curves in a structured mesh

without affecting the quality or connectivity of any of the component meshes. However, the cost of searching for hole points at each time step was very high. To reduce the cost of searching for hole points in dynamic simulations, Meakin and Suhs [6] reduced the search space by only searching the subset of points on either side of the hole boundary from the previous time step.

Because holes do not need to be exact in many cases, low-cost approximations can be used to improve the efficiency of the method. Early overset implementations such as DCF3D [4] and PEGSUS [7] added this capability by allowing user-defined geometric "cutters." These cutters were simple analytical shapes, chosen by the user, to approximate pieces of the geometry. Because determining if a point lies inside an analytical shape is simple, their use resulted in dramatic performance increases. Figures 2.5a and 2.5b show a store geometry and its approximation using analytic shapes.

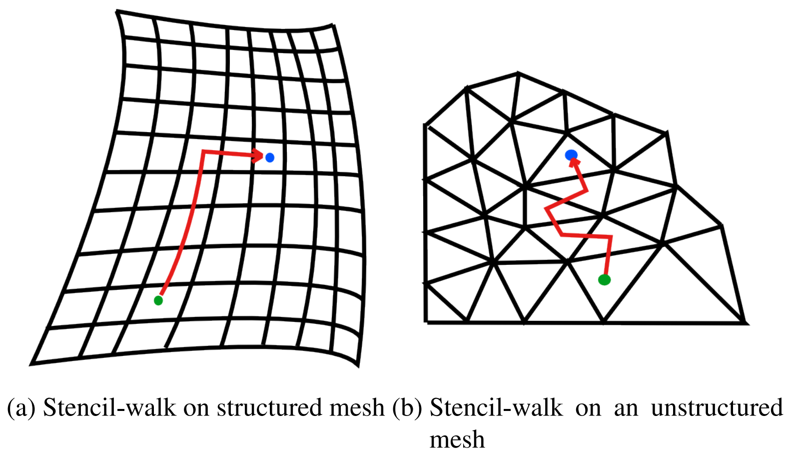
In an effort to reduce the cost of searching for donors, implementations such as PEGSUS [7] replaced exhaustive donor searches with cheaper, but still robust searches based on a "stencil-walk" method. Stencil-walk methods are a class of methods that use connectivity information to walk from one cell to the next. There are many ways to implement these methods for different classes



(a) Store geometry (b) Approximation with geometric cutters

Figure 2.5 Geometric cutters

of meshes, but they are conceptually equivalent. Figure 2.6 illustrates the path of a stencil-walk based search in both structured and unstructured-grid contexts. This approach greatly reduced the cost of finding appropriate donors, but the cost was still of the same order as the flow solver. [6]



(a) Stencil-walk on structured mesh (b) Stencil-walk on an unstructured mesh

Figure 2.6 Stencil-walk method

Meakin [4] introduced the concept of inverse maps to facilitate fast conversion from physical coordinates (x,y,z) to computational coordinates (ξ,η,ζ) of any component grid. This is accomplished by mapping the independent computational spaces of the component grids to a collection of auxiliary Cartesian grids. Because the maps are based on Cartesian grids, the donor search

becomes an $O(1)$ operation. To completely map each point in a component grid, each cell of the auxiliary grid must contain only a single point. To achieve this one to one mapping, the underlying Cartesian auxiliary grids may need to be extremely fine. This can lead to a high memory footprint. Meakin [4] introduced approximate inverse maps to reduce memory requirements. Approximate inverse maps do not require a one to one mapping, so they can be built upon coarse auxiliary Cartesian grids. Each cell in the map may contain multiple points from a component grid, but it is only associated with one of them. Because the approximate inverse map does not contain exact matches, a local search will be required, but this search should be short because the starting location will be near the target. A local walking search will be significantly shorter than a global walking search. In fact, using approximate inverse maps was shown to be several times faster than using global walking searches. [4] Figure 2.7 shows a coarse inverse map over a structured component grid.

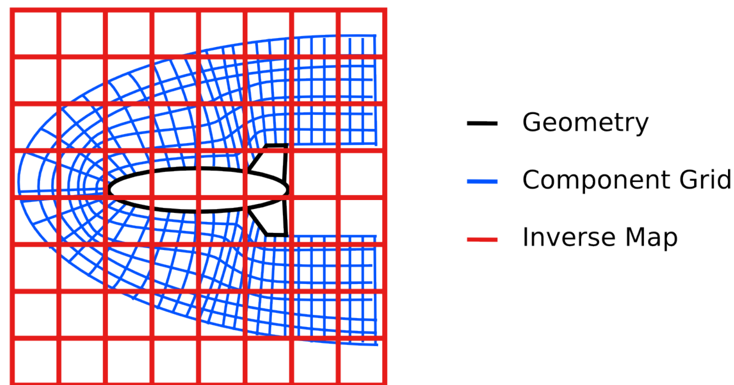


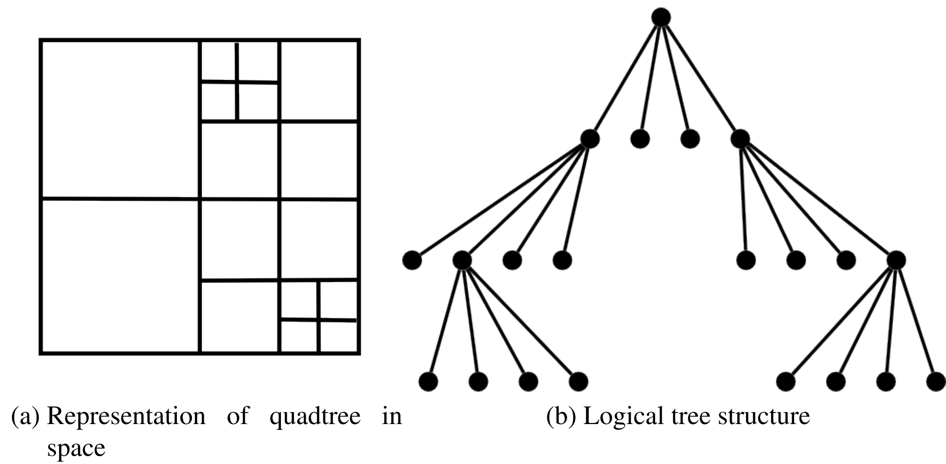
Figure 2.7 Inverse map

In the early 1990s, flow solvers were adapted to run on distributed memory machines, so researchers began exploring methods for parallelizing the domain assembly process to keep pace. In the first parallel implementation, inverse maps were used to determine which processes needed

to be queried to find donors for points that were marked as receptors. [8] This initial approach was improved by taking advantage of information from the previous time step. A lookup table containing potential donors was created by marching ± 1 in one or more of the indices of the donor grid. In this way, many donor searches could be completed by searching a small section of the lookup table instead of using the standard walking search, and the regular donor search could be used in case of failure.

In addition to the computational cost of domain assembly algorithms, the amount of user input required to set up a case was very high; user input for assembly of aircraft, pylon, and external store grids was often thousands of lines long. [9] Codes such as Beggar [9] were developed to attack this aspect of the problem. Beggar automatically used solid grid surfaces as cutters. This process was based on boundary conditions, which removed the requirement for the user to explicitly choose surface patches as cutters. For every component grid, Beggar would first identify cells that intersected surface patches, then perform a flood fill to mark the interior cells as holes. Beggar used an octree spatial data structure to store surface elements, which sped up the cutting process. An octree is a spatial data structure that is commonly used for geometric searching. It is based on an axis-aligned hexahedron. This hexahedron can be subdivided into 8 equal children, which can be recursively subdivided until the desired resolution is reached. Figure 2.8 shows a 2-Dimensional (2D) representation of an octree (a quadtree) and its logical structure.

While using the solid grid surfaces as cutters helped to automate the domain assembly process, it was still computationally expensive. In an effort to combine the computational benefits of geometric cutters with the automation of using boundary surfaces as cutters, Meakin introduced Cartesian “hole-maps”. [4] To generate hole-maps, first, Cartesian grids are created that span the

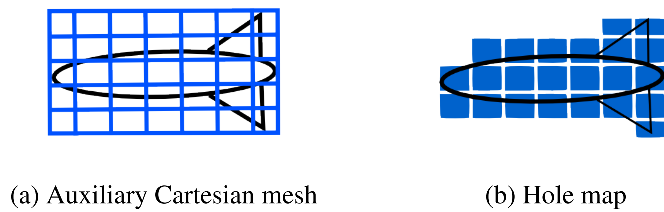


(a) Representation of quadtree in space

(b) Logical tree structure

Figure 2.8 Quadtree

space of each geometric entity and each component grid. Then, cells of the Cartesian maps that reside inside solid surfaces, or inside the component grids are flagged. These maps serve as approximations of the geometry and component grids. The maps can be used to determine in/out status of any x, y, z coordinate in constant time, i.e., $O(1)$. Figure 2.9a shows an auxiliary mesh that spans the space of a store geometry. This mesh can be used as an approximation for the geometry (i.e., a hole map) by flagging all the internal or intersecting cells (as shown in Fig. 2.9b).

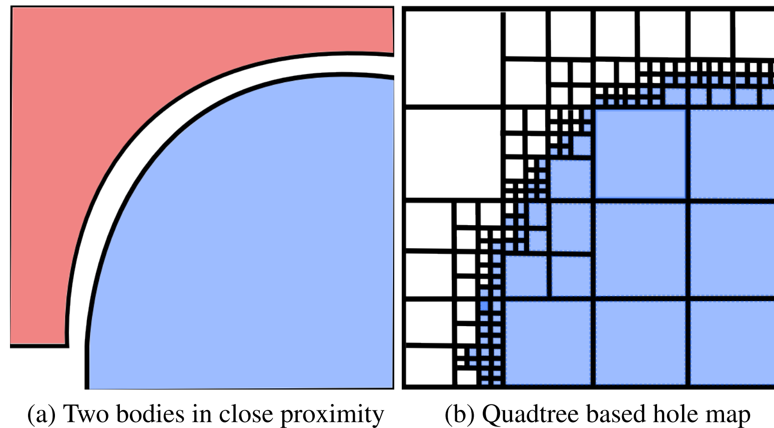


(a) Auxiliary Cartesian mesh

(b) Hole map

Figure 2.9 Hole map

While hole-maps facilitate quick in/out evalutations, they can become costly in terms of memory in cases where very high resolution is required to capture the geometry. Consider two bodies in close proximity, as in Fig. 2.10a. A hole-map would need very small cells in order to differentiate between the two bodies. For a Cartesian hole-map, the entire map would need to have that resolution, which could be costly. For hole-maps based on tree structures, such as the quadtree in Fig. 2.10b, the effect of the resolution requirements can be mitigated to an extent, but the cost can still be high if the required resolution is fine enough.



(a) Two bodies in close proximity (b) Quadtree based hole map

Figure 2.10 Issue with bodies in close proximity

In addition to hole-maps, Chui and Meakin [10] also introduced a method to automatically "optimize" hole boundaries to have minimum overlap. Starting with an initial hole boundary, the method locally expanded or contracted the boundary to reduce overlap and eliminate orphans (i.e., receivers with no valid donors). The method was shown to generate high quality hole cuts with significantly less user input, even for applications that involved bodies in relative motion.

Meakin [11] incorporated Adaptive Mesh Refinement (AMR [5]) into the overset method, which improved the flexibility and efficiency of the method. In overset simulations using AMR, many of the donor searches are performed on off-body Cartesian meshes. This reduces the overall cost of the donor search operation due to the convenient properties of Cartesian meshes discussed earlier.

Meanwhile, a new spatial data structure called an Alternating Digital Tree (ADT) [12] facilitated very efficient geometric searching, which codes such as CFD-FASTRAN [13] incorporated to speed up donor searches. This data structure is particularly useful for performing geometric searches over unstructured grids. The ADT was originally developed to improve advancing front grid generation, but it has proven to be quite valuable in the realm of Overset domain assembly as well.

Solvers with integrated domain assembly (e.g., Beggar [14]) could hide some of the hole cutting cost by overlapping the grid assembly with the flow solver at each iteration. Parallelization efforts for Beggar began with a hybrid approach in which the solver used coarse-grained parallelism at the component grid level, and the assembly was done on a single process, but overlapped with the solver to partially hide the cost. This concept served as a springboard for further developments, and was later extended by using multiple processes to complete the assembly during each flow step. [15] Because the entire grid was stored on these processes separate from the flow solver, dynamic load balancing was used to improve efficiency (without changing the partitioning of the solver).

The Overture framework, which was the result of 15 years of development upon its Fortran-based predecessor, CMPGRD, [16] was an object-oriented environment written in C++ for solving

PDEs on overset domains. [17] The primary purpose of Overture was to provide all of the functionality of the previous Fortran implementation, but provide the user with higher level tools that would simplify the process as much as possible. The user could manipulate objects, such as component grids, at a high level; and the framework would handle the low level hole cutting details by using existing methods. With this layer of abstraction, the underlying details of the assembly process could be changed without affecting code that relied on the assembly.

During the 1990s, most parallel overset methods employed coarse parallelism over a collection of component grids, where the number of component grids was on the order of the number of processors. This approach was extended by using AMR to generate large numbers of off body grids, which could be grouped by size (along with near-body grids) to improve load balancing. [18] Better load balancing was possible because the number of grids was 10–20 times higher than the number of processes. This approach to load balancing worked well for simulations where the number of processes was small, but the grid-to-process ratio is difficult to maintain on large scale simulations with many thousands of processes.

Drawing from the concepts of common ray casting [19] and hole-maps, Meakin introduced the Object X-Rays method. [20] This method combines the robustness of traditional ray casting methods with the efficiency of a 2D uniform Cartesian grid. When casting a ray from a given point, the parity of the number of intersections of the ray with any closed set of surfaces indicates if the point is inside or outside of the surface set. The cost of ray casting is proportional to the number of points in question and somewhat to the number of points defining the surfaces. The X-Rays method reduces this cost by casting rays from a 2D uniform Cartesian grid as opposed to each potential 3D target point. The Cartesian grid is placed below the hole-cutting objects and normal to a specific

coordinate direction. A normal ray is cast from each point in the plane with a magnitude sufficient to fully intersect the hole-cutting surfaces. Information pertaining to the intersections of each ray and the hole-cutting surfaces is stored for every point on the plane as shown in Fig. 2.11 and the result is analogus to common X-Ray imaging. When hole-cutting, the status of a 3D target point can be determined by projection of the target onto the 2D Cartesian grid followed by examination of the pre-computed ray casting results stored at the verticies of the encompassing 2D grid element. This method virtually eliminates the memory costs associated with 3D hole-maps as only planar data is stored. However, the robustness of the ray casting method is impacted by the resolution of the Cartesian grid.

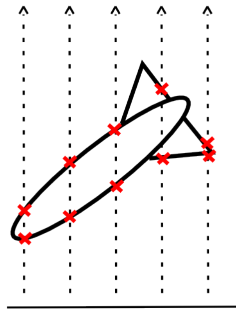


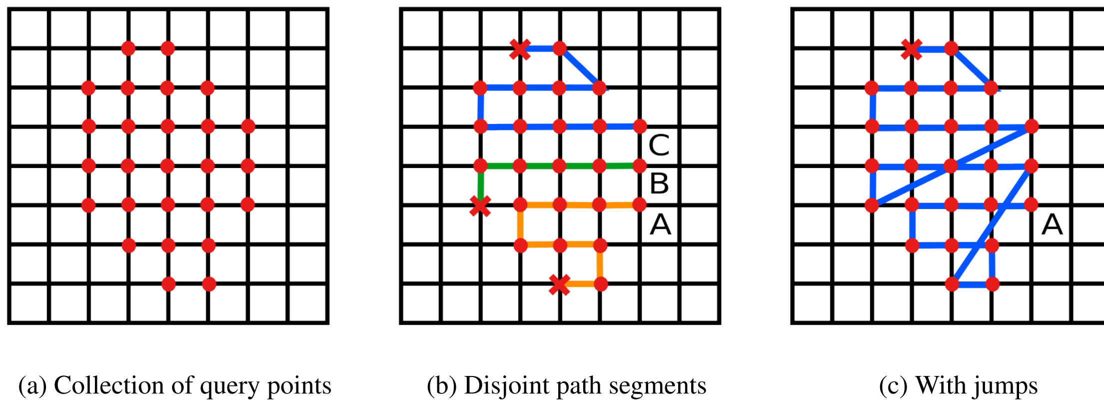
Figure 2.11 X-rays

PEGASUS 5, which was a new code to replace PEGSUS version 4 (the spelling was changed when the new code was adopted), incorporated a Cartesian hole map in conjunction with a line of sight algorithm. [21] These additions improved the automation and robustness of the hole cutting process. The efficient ADT data structure was also used to accelerate geometric searches. The new code employed coarse-grained parallelism based on component meshes as other codes

had done earlier. PEGASUS 5 obtained a speedup factor of 33 on 48 processors for a large test case (79 component grids). [21]

Researchers at NASA Ames compiled a collection of internally developed tools for assembling structured components called the Chimera Components Connectivity Library (C3LIB). [22], [23] This library provided an API for all of the common functions needed to perform domain assembly for structured grids (i.e., creation of Object X-Rays, multiple stencil-search methods, and functions to determine interpolation stencils for query points). In the context of explicit hole-cutting on structured grids, it is extremely cheap to find donors in Cartesian grids, but it is comparatively expensive (about a factor of 100) [22] to locate donors in curvilinear grids. In systems with curvilinear grids near the body and Cartesian grids in the off-body, the largest cost in the donor search process is attributed to the query points in the Cartesian grids (whose donors are in the curvilinear near-body grids). Chan [22] introduced a method of ordering query points along paths to reduce the number of global searches required during the donor search process. The method takes advantage of the fact that the donors for adjacent query points will be close, i.e., if the donor is known for a particular query point, that donor will be a good starting location for a stencil-walking search. Consider an unordered collection of query points as shown in Fig. 2.12a. Performing a global search for each of these points would be expensive, so it is desirable to reduce the number of global searches if possible. Those points can be ordered along path segments such that each point is adjacent to the previous point on the path as shown in Fig. 2.12b. The paths can be constructed by starting at a point, e.g., point A in Fig. 2.12b, and adding one of its neighbors to the path segment. This process continues until a query point is reached that has no unvisited neighbors, then the path segment terminates. The overall cost of the donor search is reduced because

global searches are only required for the first query point along each of the path segments. Chan recognized that the number of global searches could be reduced further by allowing jumps in the paths. Instead of terminating, e.g., segment *A* in Fig. 2.12b, Chan allowed the segment to jump to any point that was adjacent to any point on the path. In Fig. 2.12c, the path is allowed to jump to point *B* because it is adjacent to the path. This reduces the number of required global searches from three to one in this example.



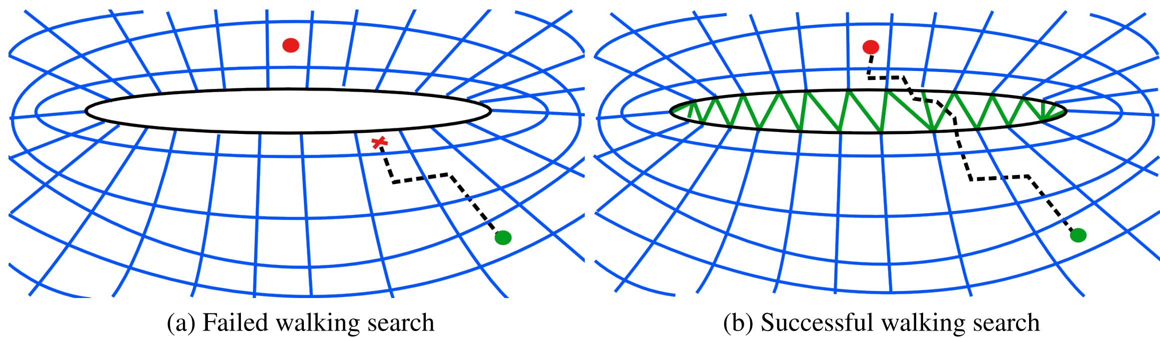
(a) Collection of query points (b) Disjoint path segments (c) With jumps

Figure 2.12 Organizing query points

2.2.2 Overset Unstructured Meshes

In the early 2000s, overset techniques were extended to unstructured meshes and quickly gained popularity. Nakahashi et al. [24] introduced implicit hole cutting (a term coined later by Lee and Baeder [25], [26]), in which they defined inter-grid boundaries based on a distance-to-the-wall criterion. Each node was assigned a value based on the distance from the node to the nearest solid surface in its containing component mesh. After assigning a distance to each node, the

domain assembly code searched for cells in other component meshes that contained the node via a stencil-walk algorithm. Figures 2.13a and 2.13b show stencil-walk paths for a donor search on an unstructured quad mesh. The red dot represents a query point in another component mesh, and the green dot represents the starting location of the stencil-walk. Stencil-walk algorithms terminate upon reaching a mesh boundary, so they can easily fail if the domain is not convex. Figure 2.13a shows an example of this failure mode. Additional logic can be added to the search to allow it to walk along boundaries, but the search cost could become very high for complicated geometries and robustness can still be an issue. Nakahashi et al. [24] augmented composite grid systems by adding subsidiary grids inside of bodies and outside the computational region. By adding these grids, they could guarantee that the augmented domain was a convex hull, therefore removing all failure modes of the stencil-walk algorithm.



(a) Failed walking search

(b) Successful walking search

Figure 2.13 Unstructured neighbor walking

After finding the containing cells, the assembler then compared the distance value of the point to the interpolated value of the containing cell to determine which piece of geometry was nearest. The search point was marked as either in or out if it was closer to its geometry or the

geometry associated with the containing cell respectively. In order to determine the distance to the geometry of the containing cell, the values of the cell's vertices were interpolated at the location of the search point. Therefore, a byproduct of implicit hole cutting is that the donor stencils are already calculated (i.e., a separate donor search phase is not required). Once all nodes were classified, the cells were marked as in, out, or fringe. If all nodes of a cell were marked as in, the cell would also be marked as in, likewise for out nodes. If one or more of the cell's vertices were marked as in and one or more of its vertices were marked as out, then the cell was marked as a fringe cell. This information was then used to determine which nodes would become receptors. Nodes that were marked as out and were also vertices of a fringe cell were marked as receptors. This guaranteed at least one node of overlap between component grids. Using a distance-to-the-wall as the hole cutting criterion, a domain assembler produces inter-grid boundaries that lie halfway between component meshes. Figure 2.14 shows a representation of two meshes that have cut each other based on distance-to-the-wall.

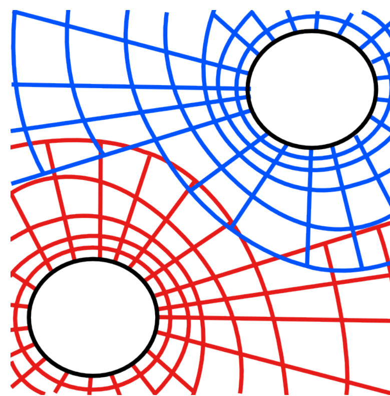


Figure 2.14 Implicit cut based on distance to the wall

Implicit hole cutting can also use criteria other than distance-to-the-wall to determine where the inter-grid boundaries should be defined. Lee and Baeder originally used cell volume. [25] Löhner et al. generalized the concept by introducing the term "dominant mesh criterion". [27] The dominant mesh criterion is some cell-wise criterion that can come from an arbitrary source and is used to define intergrid boundaries (i.e., a donor cell will have a lower value for the dominant mesh criterion than its receiver). For their work, they used the criterion:

$$s = d^p \times h^q \tag{2.1}$$

where d is the distance-to-the-wall, h is the cell size, and p and q are tunable parameters. When $p = 1$ and $q = 0$, the scheme reduces to distance to the wall. They also improved the efficiency of the method by using a fast distance to the wall calculation, and using incremental interpolation between time steps. If the relative motion of bodies is small from iteration to iteration, it is likely that the interpolation boundaries at a given iteration will be similar to those at the next iteration. Under this assumption, a domain assembly code can use donors from a given iteration as seeds for the walking search of the next, which could greatly reduce the number of steps needed to reach the correct donor.

To handle overlaps in near-body regions where the grid spacing is small, hole-maps, x-rays, and other approximation methods require high resolution. Because this level of detail may not be required for cutting holes in off-body meshes with larger spacings, Noack divided near-body and off-body assembly into two separate tasks that could use resolution appropriate for each task. [28]

He used an octree structure to speed up the hole cutting process, and an octree of different resolution to provide starting locations for a stencil-walk based donor search. He also incorporated an iterative process for reducing overlap based on a prescribed "donor suitability function," which gave control similar to implicit methods (though with a much different approach). Noack released an overset domain assembly code called SUGGAR [29] and an associated library called DiRTlib. [30] SUGGAR used binary trees to reduce the memory overhead associated with the octree structures that it originally used. [29] It was later enhanced to handle small gap regions via a direct cut approach. [31] Noack later wrote a new version (SUGGAR++), which was the first commercial product that targeted a wide variety of structured and unstructured solvers. [32] SUGGAR++ was capable of operating on distributed memory systems, and it used a Spatial Decomposition Volume repartitioning scheme [32] to improve load balancing. The SDV scheme is described in detail in 2.3.3.1.

Sitaraman et al. [33] introduced PUNDIT, a general parallel domain assembler, that operated on the same partitions as the solver. PUNDIT (Parallel UNsteady Domain Information Transfer) introduced the flexibility to transfer domain information between multiple solvers with potentially different mesh types (structured, unstructured, Cartesian). PUNDIT was designed to operate in parallel by using the native partitioning scheme of the solver(s) involved in the assembly. The module has a number of other distinguishing features. PUNDIT accesses grid and solution data directly through pointers, which are provided when each solver is "registered." This reduces memory overhead. PUNDIT also uses several techniques to reduce the search space for the implicit hole cutting problem. Oriented bounding boxes are created around grid partitions by using

inertial bisection, then those are divided into "vision space bins." Figure 2.15 shows the bounding box concept. Depending on the orientation of an object, an axis-aligned bounding box may contain much more empty space than necessary (Fig. 2.15a). This can be reduced by orienting the bounding box so as to minimize its volume (Fig. 2.15b). The cells in each partition are then reordered to make identifying each cell that lies in a particular bin cheaper. With these in place, oriented bounding boxes are shared among all processors, then each process can identify lists of potential receivers at the same time as determining where the incoming oriented bounding boxes overlap with its grid(s). Once potential receivers are identified, lists of those are exchanged, and donor searches are performed. The donor search algorithm uses the vision space bins to initialize a short stencil-walk. The processes of making near-body to near-body connections and near-body to off-body connections is separated to accommodate adaptive Cartesian off-body grid generation and take advantage of the inherent efficiencies if meshes of that type are used in the simulation. PUNDIT showed linear speedup on up to 12 processes, but the speedup quickly dropped off due to load imbalance imposed by the flow solver partitioning. PUNDIT originally used cell size, which they called "resolution capacity," as the dominant mesh criterion, but this was later updated to include the influence of wall distance. [33] Recently, Roget and Sitaraman [2] implemented a new donor search strategy based on EIM (Exact Inverse Maps) and explored dynamic load balancing. The EIM method was shown to be more efficient than the ADT method. Dynamic load balancing improved scalability (out to hundreds of cores) for the ADT method. Sitaraman presented results of applying dynamic load balancing to the EIM method. [34] The grid assembly process was shown to take 21% of the solver time per time step on 8192 cores for a large test case.

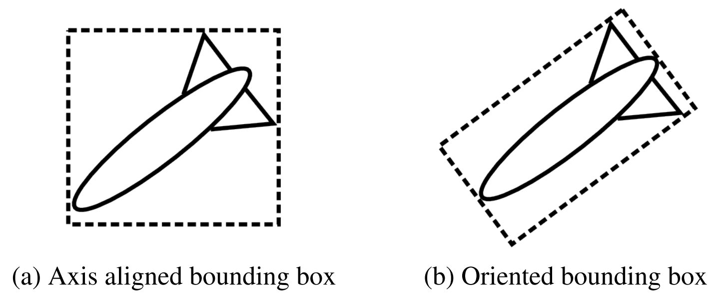


Figure 2.15 Bounding boxes around a store geometry

Zagaris et al. [35] developed another in-core parallel assembly to tackle the distributed assembly problem. This method performed explicit hole cutting and used axis-aligned bounding boxes to identify regions of potential overlap. This method used both “virtual grids” (i.e., auxiliary structured Cartesian grids) and octrees as tools to reduce search space during the hole cutting and donor searching processes. The method did not scale well in parallel, but the assembly process represented less than 10% of the overall solution for the problems presented in the original paper. [35] In a subsequent paper, Zagaris et al. [36] presented a caching strategy that drastically improved the performance of their donor search algorithm. Their strategy leveraged the fact that points that are close in physical space may be close in index space as well. When a donor cell was identified for a query point, its neighbors were added to the front of the cache, that was implemented as a doubly linked-list. The donor search for the next query point started by checking the cells in the cache and would revert to the standard search if there was a “cache miss.” For the cases examined, the cache miss rate was very low, and overall performance improvements of 2x or greater were observed.

2.3 Current State of the Overset Grid Method

As stated, there are two motivations for using an overset approach: (1) to simplify the process of creating structured grids of high quality on complex bodies and (2) to handle bodies in relative motion. Thanks to improvements to both structured and unstructured grid generation over the last 30 years, the primary benefit of the overset grid approach is to handle relative motion. In a dynamic, moving body simulation with a single grid spanning the computational domain, the grid must deform as the bodies move. This can cause elements to become highly skewed or even inverted, affecting accuracy, convergence rates, and stability. One option to mitigate these issues is to remesh the grid as the bodies move, however, this approach comes at a price (e.g., the size of the linear system changes if nodes are added or removed, refinement can cause load balancing issues, etc.). The overset approach requires less from the solver. The solver need only be able to ignore a list of control volumes, and apply Dirichlet boundary conditions for another list of control volumes by using interpolated data. Because linear interpolation is typically used, conservation is not strictly enforced, which can be a problem for certain applications. Conservative interpolation is possible, but is more computationally expensive and more difficult to implement. Inaccuracies caused by inappropriate donors are also possible. However, judicious application of overset best practices (e.g., grids have similar resolution in overlap regions, sufficient overlap exists between bodies in close proximity) minimizes the effect of these issues and leads to high quality solutions for problems that would otherwise be intractable. As modern engineering problems continue to increase in size and scope, the overset method will likely see even wider application. In the following subsections, we examine computational cost, geometric searching techniques, and load balancing strategies.

2.3.1 Computational Cost

At the most abstract level, the overset domain assembly problem has very few logical pieces. Prior to the assembly, there is a collection of separate, but overlapping, meshes that cover a domain on which we would like to obtain a flow solution. These meshes must be somehow associated with one another such that information can propagate across the domain. This is typically achieved through interpolation. For the flow solver to operate on the composite domain, it only needs four pieces of information: (1) a list of control volumes to be excluded from computation (i.e., blanked), (2) a list of receiver control volumes that should be updated with interpolated data, (3) IDs of donor control volumes for each receiver, and (4) normalized weights for each of the donors. The job of the domain assembler is to determine this information. To that end, the domain assembler needs to accomplish two results, both of which require expensive geometric searching. Every control volume must be assigned a status (e.g., in, out, receiver). For each receiver, donors must be identified (along with appropriate weights). Note that these are two *results* of any domain assembly process, but they need not be two separate *operations* (e.g., implicit hole cutting achieves both in one step).

The amount of geometric searching makes this process extremely intensive, and the complexity of the problem only increases in parallel (distributed memory). The domain assembler would ideally operate on the same partitions that are used by the flow solver. However, this partitioning is typically chosen to balance the workload for the solver, which is likely to be vastly different than that of the assembler. Even if the initial partitioning takes the assembly process into account, the load can still become imbalanced because the work load can change dramatically over the course of the simulation as the component meshes move.

2.3.2 Geometric Searching Tools

The main factor driving the cost of the overset domain assembly process is geometric searching. A detailed study of the 30 year history of the overset methodology reveals a rich set of tools to efficiently perform the assembly process. The following is a list of geometric searching tools encountered in the history section for convenience.

- Bounding boxes
- Auxiliary meshes
- X-rays
- Spatial trees
- Walking algorithms

There are three factors that influence the high cost of geometric searching: the number of searches performed, the size of the search domain, and the method of searching. Each of the listed tools targets one or more of these aspects to reduce the cost of searching and are described in the following subsections.

2.3.2.1 Bounding Boxes

Bounding boxes are both simple and extremely powerful. They can be used to remove entire grids from the search space of a given operation. For example, consider two component grids that do not overlap as shown in Fig. 2.16. With no a priori knowledge of the domain, an assembler performing implicit hole cutting would need to compare each control volume in each

component grid against each control volume in the other. An entire grid can be removed from the search space in one step if bounding boxes defined for the two components do not overlap. Explicit hole cutting also benefits from this concept. If a bounding box around a piece of geometry does not intersect the bounding box of a given mesh, then that mesh can be removed from the search space of the cutting geometry.

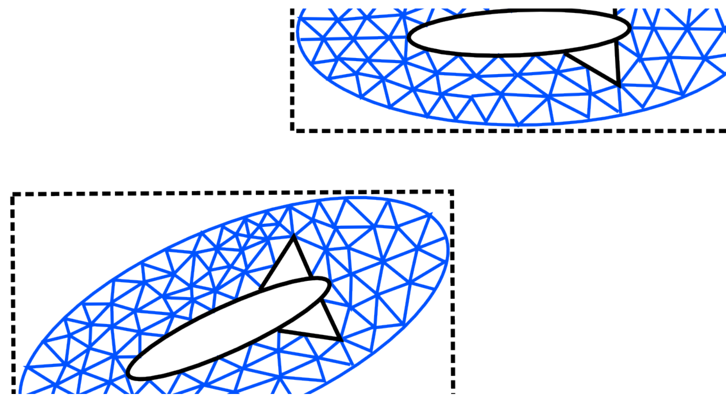


Figure 2.16 Non-overlapping bounding boxes

Because simple bounding boxes are aligned with the Cartesian axes, they excel when the major axes of the objects they represent mimic the Cartesian axes, but this is not always the case. Consider two meshes whose bounding boxes overlap, but the meshes themselves have no overlap as shown in Fig. 2.17. Bounding boxes can be oriented to eliminate the overlap as shown in Fig. 2.18.

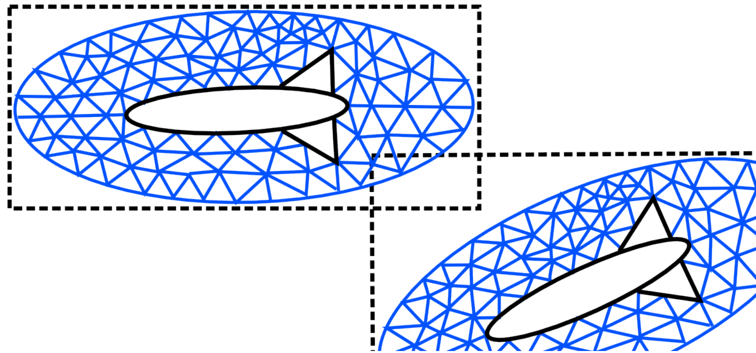


Figure 2.17 Overlapping bounding boxes

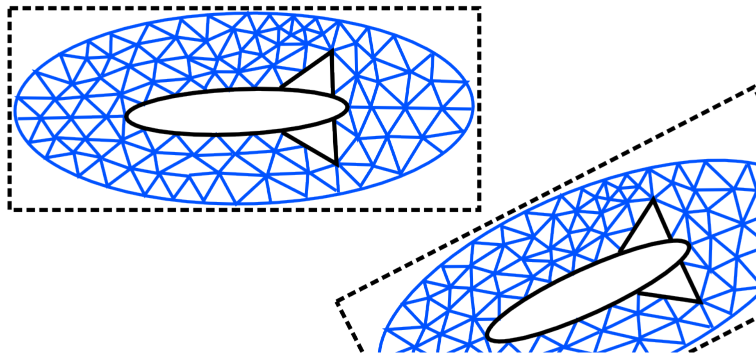


Figure 2.18 Oriented bounding boxes

2.3.2.2 Auxiliary Meshes

Auxiliary meshes are additional meshes that are created by the domain assembler that are not part of the composite domain, but are used to aid in the assembly. Structured Cartesian meshes, and their rotated counterparts, are typically used due to their convenient property that a geometric search is an $O(1)$ operation. Auxiliary meshes can be leveraged for several different aspects of the domain assembly process, from quickly determining in/out status to providing approximate starting locations for stencil walks. The hole map method is built upon auxiliary meshes that have each cell tagged as inside, outside, or crossing the geometry. Because point containment searches

are trivial on hole maps, they can be used to quickly determine if points in a domain are internal to the geometry.

Inverse maps also rely on auxiliary meshes that are constructed around each component mesh. Each cell of the auxiliary meshes that contains one or more control volumes from its component mesh is associated with one of those control volumes. The inverse maps can be used to quickly identify a control volume that is in the neighborhood of a query point. That control volume can then be used as a starting location for a stencil-walk, which can be significantly cheaper than a stencil-walk that starts from a less ideal location.

PUNDIT uses the EIM method (Exact Inverse Map), which is closely related to the original inverse map. Whereas each cell in an inverse map is associated with a single control volume, each cell in an EIM is associated with *every* control volume and boundary face that overlaps with it. This allows the stencil search to be completely localized to a single cell of the EIM, which improves the efficiency and robustness of the method. [2] PUNDIT also uses auxiliary meshes to reduce the number of query points and reduce the search space for donor searches.

2.3.2.3 X-rays

X-rays can be an efficient means of representing geometry for performing minimum hole cuts, but the cost can go up dramatically if the body has features that are aligned with the rays. Figure 2.19 shows an x-ray slice through a three-finned store geometry that has a vertical tail. The coarse x-ray representation captures the general shape of the geometry, but completely misses the vertical tail feature. To capture this feature for hole cutting, a much higher resolution set of x-rays is needed, which is expensive. For geometries that have multiple identical features, such as the

fins on the store, it is possible to solve this problem by manually creating multiple x-rays with different orientations. However, for complex geometries, this can require significant user expertise and effort. [20] Chan et al. [37] introduced adaptive x-rays to reduce the manual effort required to perform hole-cutting using x-rays. Figure 2.20a shows a top view of the same three-finned store geometry. The rays in Fig. 2.19 were fired from a uniform 2D Cartesian grid. If finer Cartesian grids are inserted into this coarse grid, a higher resolution can be achieved without incurring the cost of increasing the resolution across the entire region. Figure 2.20b shows the adapted Cartesian grid, in which a ray will be fired from the center of each cell.

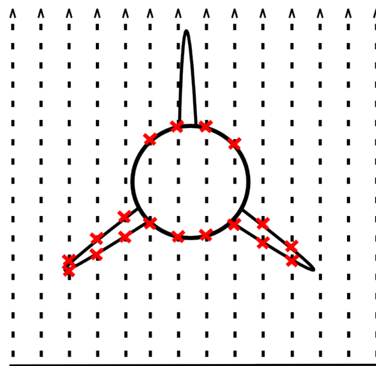


Figure 2.19 X-rays on store geometry

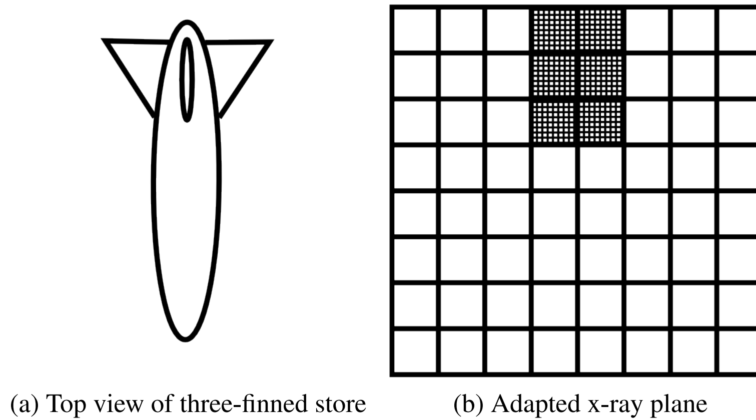

(a) Top view of three-finned store (b) Adapted x-ray plane

Figure 2.20 Adaptive x-rays

2.3.2.4 Spatial Trees

There are many varieties of spatial trees, but two are particularly noteworthy in the context of overset domain assembly: the Octree and the Alternating Digital Tree (ADT) [12]. The Octree, decribed in section 2.2.1, is simple to implement and offers a convenient way to store points. Storing objects other than points in an Octree, however, is complicated by the fact that those objects may lie in more than one octant. The ADT provides an elegant solution to this problem by treating an N-dimensional region (e.g., a bounding box) as a single point in 2N-dimensions. For example, Fig. 2.21 shows a 1D region $[x_{min}, x_{max}]$ represented as a point in 2D. The ADT is logically a binary tree (i.e., a digital tree), in which each child is half the size of its parent. The parent is split along the jth axis, where j is determined by

$$j = l\%N \tag{2.2}$$

where l is the level in the tree and N is the dimension of the space. A 2D ADT, for example, would be alternately split along the x and y dimensions (hence, *Alternating* Digital Tree).

Searching the tree is therefore straightforward, because only one floating point comparison (the current dimension) must be made to determine which child to visit at each level of the tree. Inserting new objects into the ADT is $O(logn)$. The ADT also provides an efficient method for removing nodes from the tree. Consider a node in the tree that needs to be removed as the root of a sub-tree. All elements in that sub-tree are contained by the root, therefore any leaf of the sub-tree can be promoted to replace the root. This flexibility can be useful in dynamic simulations in which it is cheaper to update the ADT than rebuild it.

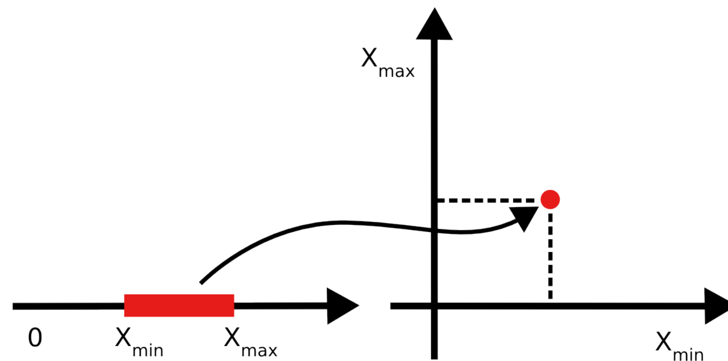


Figure 2.21 1-D ADT

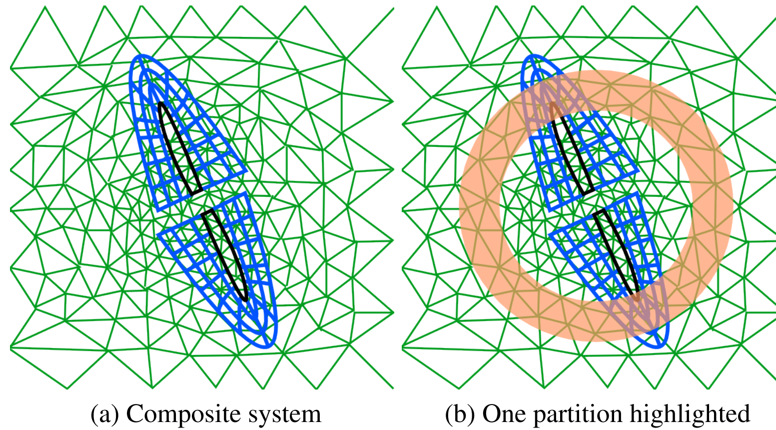
2.3.3 Load Balancing Strategies

For time dependent simulations with bodies in relative motion, the domain connectivity needs to be updated at every physical time step, therefore the parallel efficiency of the domain assembler is critical. The domain is typically partitioned such that each partition has a similar

number of control volumes, based on the assumption that the amount of work associated with each control volume in the domain is roughly the same. Unfortunately, this assumption does not hold for the domain assembly process. Control volumes in regions of component meshes that do not overlap with other meshes will require zero work, while control volumes in overlap regions will require expensive geometric searching. There are two ways to address this load balancing challenge.

2.3.3.1 Spatial Repartitioning

One approach to alleviating the load imbalance imposed by the partitioning of the flow solver is to repartition in a way that is amenable to the assembly process. SUGGAR++ repartitions with Noack's Spatial Decomposition Volume technique. [32] In this technique, the domain is partitioned based on prescribed regions of space based on the geometry and its potential motion rather than control volume connectivity. To illustrate this, consider a case with rotating blades as shown in Fig. 2.22. The motion of the blades is known a priori, therefore, it is possible to partition the mesh to improve the performance of the assembly process. Figure 2.22b shows a cylindrical partition that contains nodes and cells from the blade meshes and the background mesh. If the amount of work associated with each cylindrical shell is similar, then better load balancing can be achieved. Furthermore, as the blades rotate, donor searches within each cylindrical partition will likely be local, which can reduce communication costs. SUGGAR++ was shown to perform measurably better when using Spatial Decomposition Volume partitioning than when using a standard flow solver partitioning (ParMETIS in this case). [32] The improved performance was attributed to better load balancing across MPI ranks.



(a) Composite system (b) One partition highlighted

Figure 2.22 Spatial Decomposition Volume

2.3.3.2 Dynamic Load Balancing

In a large scale dynamic simulation, the assembler can not feasibly maintain its own partitioning or completely repartition at every time step. However, the assembler may distribute certain aspects of the assembly process, while leaving the solver partitioning intact. Roget and Sitaraman [2] implemented a dynamic load re-balance algorithm in PUNDIT that showed very promising results. In their method, there are two types of rebalancing: static and dynamic. For the first iteration, no work history exists, therefore a static balance must be performed using an estimate of the amount of work that each process would have. The number of query points in potential overlap regions is used to estimate the load. PUNDIT identifies these regions as part of its profiling step prior to load balancing. The estimated load for each process is computed, which is then used to calculate the average load among all processes. Then, iteratively, a fraction of the load from the most heavily loaded process is transferred to the least loaded process such that one of them reaches the average load. This is repeated until all processes are near the average load. In subsequent iterations, the load balancing process is the same, except that the load for each process is determined by

directly measuring the total time to perform the assembly process. Sitaraman recently combined dynamic load balancing with the EIM method in PUNDIT and showed further reductions in the domain assembly cost. Running a large case on over 8,000 cores, he demonstrated assembly costs of roughly 20% of the cost of the flow solver time step. [34]

2.3.4 PUNDIT: Current State of the Art

Roget and Sitaraman recently implemented an adaptive load balancing algorithm in PUNDIT [2]. Their method is the highest performing and most scalable domain assembler implemented to date, and is considered the current state of the art for the purposes of this work. This section will provide a high level view of the operations of PUNDIT, and some details about its load balancing method in order to set up an appropriate context for describing the differences in YOGA.

PUNDIT performs domain assembly in the following steps.

- hole profiling
- query point identification
- mesh-block profiling
- donor search
- point type assignment
- interpolation

Hole profiling: hole maps are created as an approximate representation of each geometric body. This step accounts for only a small fraction of the overall running time, so no more detail is required here.

Query point identification: determines which points have potential overlap with points in other component meshes. Once identified, query points, and their associated cell connectivity information are transferred between processors intelligently in order to load balance (described at the end of this subsection).

Mesh-block profiling: after each process gets the query points and cells for which it is responsible, it preprocesses the cells. There are two different donor search methods in PUNDIT, and they perform different preprocessing. The ADT method puts all of the cells into ADT's, and the EIM method builds Cartesian maps and neighbor connectivities. This step accounts for a large portion of the overall cost of an iteration, and is necessary to speed up the donor searching step.

Donor search: each process identifies candidate donor cells for each query point. This step accounts for the majority of the computation cost of an iteration and is the core operation of domain assembly.

Point-type assignment: classify points as hole points, solve points, or receptors. If there are conflicts, or undesirable interpolations, those are resolved during this stage. For example, any receptor that is also a donor and is not a mandatory receptor, will be changed to a solve point.

Interpolation: PUNDIT calculates donor interpolation weights via the Newton-Raphson technique [33].

2.3.4.1 Load Balancing Implementation

Performing donor searches is the core operation within domain assembly, therefore PUNDIT estimates processor loads initially based upon the number of query points (a donor search must be performed for every query point). Note that the load balancing assigns query points (and connectivities) to processors in such a way that donor searches will be performed locally.

The initial estimation does not account for the increased overhead induced by the load balancing process. To account for this, PUNDIT measures the actual load imbalance during each iteration, and uses that to adjust the load balancing of subsequent iterations. This adaptive rebalancing was shown to improve wall clock time significantly [34].

CHAPTER 3

METHODOLOGIES AND ALGORITHMS

This chapter introduces a new method for assembling large Overset grid systems in parallel. A new dynamic load balancing strategy lies at the heart of the method, so several load balancing concepts are first explored to provide context and motivation for the choices made for this work. Then a description of how the domain assembly process was recast to fit the load balancing strategy is given. Finally, implementation details are provided, and specific technologies that enabled this work are discussed.

3.1 Monte Carlo Investigation of Load Balancing Strategies

There are two load balancing concepts that are critical to this work: over-decomposition and dynamic load balancing. An MPI parallel load balancing simulator was developed for this work to explore the characteristics of both concepts. Specifically, the simulator demonstrates the effect of introducing uncertainty to work size estimation on computational cost in terms of total wall clock time. The simulator creates simulated “work units” as a series of integers generated according to a normal distribution. The work units are distributed amongst the MPI processes in the simulation. A worker performs a work unit by simply sleeping for the duration defined by the work unit. For non-dynamically load balanced simulations, all work units are assigned up front, and for dynamically load balanced simulations, each process requests a new work unit upon

completion of the previous work unit. Note that, in the case of dynamic load balancing, different processes will likely perform different numbers of work units.

3.1.1 The Simulator

The load balancing simulator is parameterized on the following quantities:

- Work units per process
- Static vs dynamic
- Confidence interval

The first two quantities are straightforward. By controlling the number of work units per process, the user can simulate a range of degrees of over-decomposition. The second quantity allows the user to switch between static and dynamic load balancing. The third quantity is used to control how accurately the work loads are predicted. The user supplies this interval as a percentage of the ideal load and a confidence level (e.g., 90% of work units will be within 10% of the ideal load). The confidence interval is then used to determine an appropriate standard deviation that the work server can use to generate work units according to a normal distribution.

3.1.1.1 Determining Standard Deviation

The user provides a confidence interval, but a standard deviation, σ, is needed to calculate the appropriate normal distribution. The normal distribution function $f(x,\sigma^2)$ gives the probability

that a normal variate assumes the value between x_0 and x_1.

$$f(x,\sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} \int_{x_0}^{x_1} e^{-\frac{x^2}{2\sigma^2}} \tag{3.1}$$

The simulator uses x_0, x_1, and the value of $f(x,\sigma^2)$ to solve an inverse problem to determine standard deviation σ that will match the confidence interval provided by the user. The simulator uses the bisection method to obtain an approximation for σ. At each step, the simulator evaluates the integral with the current guess for σ using the composite trapezoidal rule.

3.1.1.2 Generating work units

The amount of work for a given work unit is calculated by

$$W = \frac{P}{N}(1 + E) \tag{3.2}$$

where P is the predicted workload for each process, N is the number of work units per process, and E is the error in the prediction. The error is calculated based upon a normal distribution, centered about zero, with a variance of σ^2 (which was calculated from the user's confidence interval).

$$E = \mathcal{N}(0,\sigma^2) \tag{3.3}$$

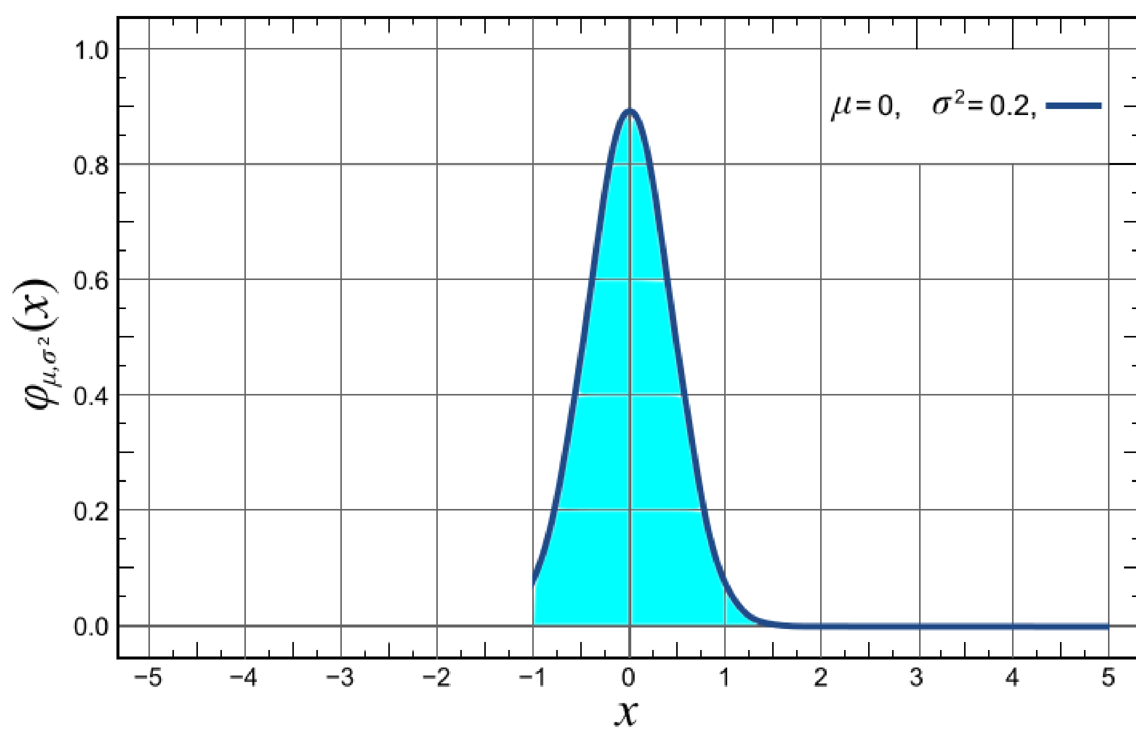


Figure 3.1 Normal distribution

3.1.1.3 Running the simulator

The load balancing simulator is written to be launched in an MPI environment to allow the user to perform simulations on different numbers of processes. The simulator, written in C++, performs a single representative simulation based on the chosen normal distribution. Monte Carlo simulations are then performed by running the simulator many times (driven by a Python script) with the same distribution to obtain statistical trends.

3.2 Simulation Results

3.2.1 Predictive Load Balancing

In predictive load balancing, each process is given a work unit that is predicted to have the same amount of work. Hence, if the prediction is very accurate then the actual load balancing will be close to ideal. However, the greater the uncertainty in the workload prediction, the more imbalanced the workload will be. Consider a simulation in which there is a 90% probability of a given work unit will be within 10% of the ideal load. Figure 3.2 shows the result of a single run of the simulator with 8 processes. The load imbalance for this run is relatively small.

Figure 3.2 Sample load 1

Consider a second scenario in which the confidence interval is wider (i.e., the uncertainty in the predicted work loads is higher). Figure 3.3 shows the result of a simulation with a 90% probability that a given work unit will be within 50% of the ideal load. For this case, the load imbalance is more pronounced due to the increased uncertainty.

Since the simulator generates work units based on a normal distribution, there is variance between each simulation. The results of a single simulation are not necessarily indicative of all

Figure 3.3 Sample load 2

other simulations. Therefore, it is important to analyze the results of multiple simulations. Figure 3.4 shows how the mean and maximum overhead costs are affected by the uncertainty in load predictions.

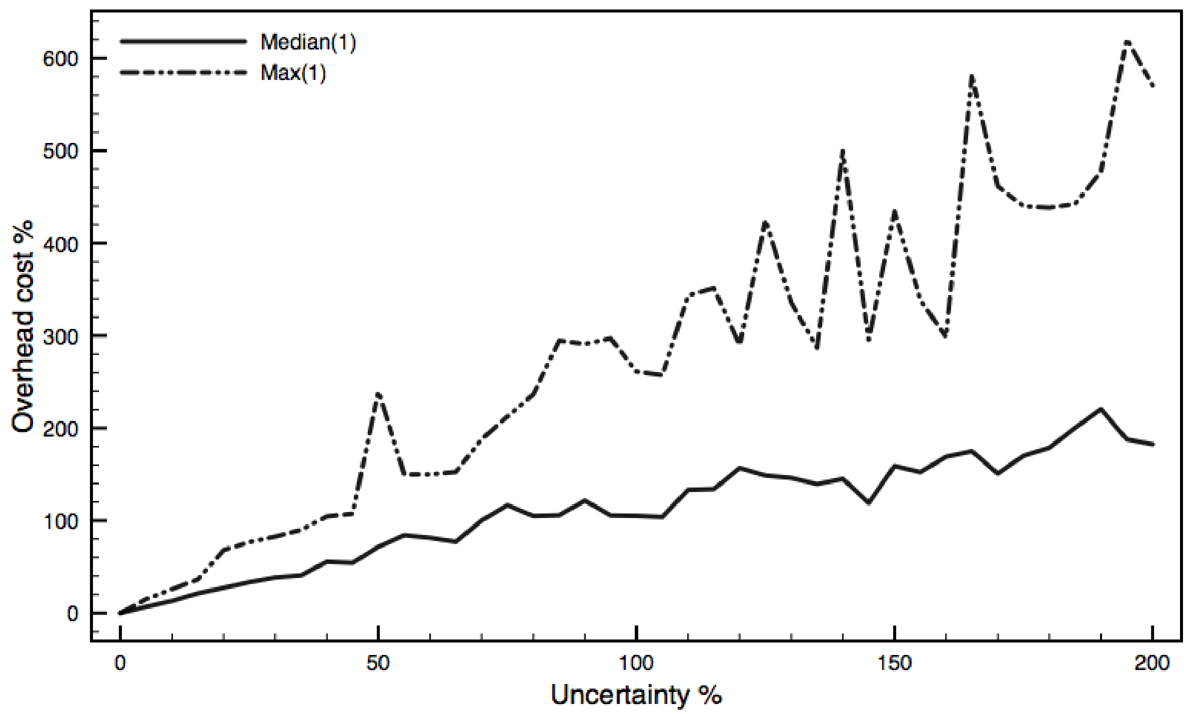


Figure 3.4 Imbalance trend

3.2.2 Load Balancing via Over-Decomposition

Dividing the work such that each worker has more than one work unit is called over-decomposition. Figure 3.5 shows how over-decomposition can reduce overhead of simulations with high uncertainty in load prediction.

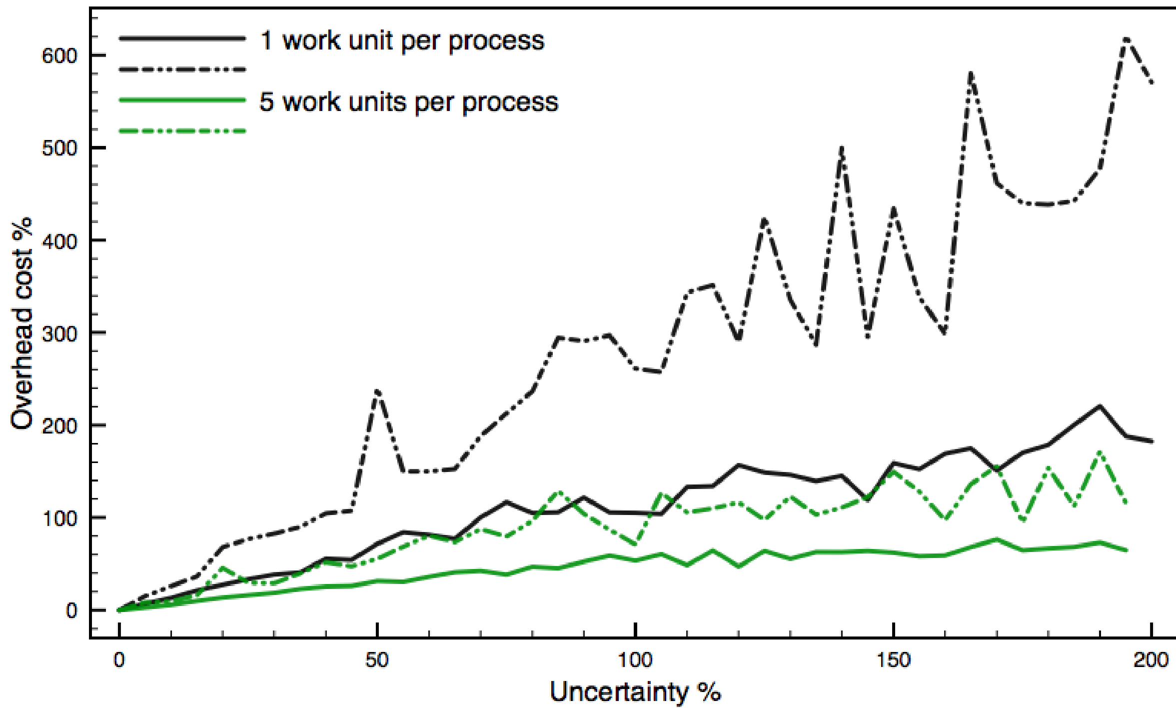


Figure 3.5 Over decomposition trend 1

The more fine grained the over-decomposition, the better load balancing can be achieved. Figure 3.6 shows several different levels of over-decomposition.

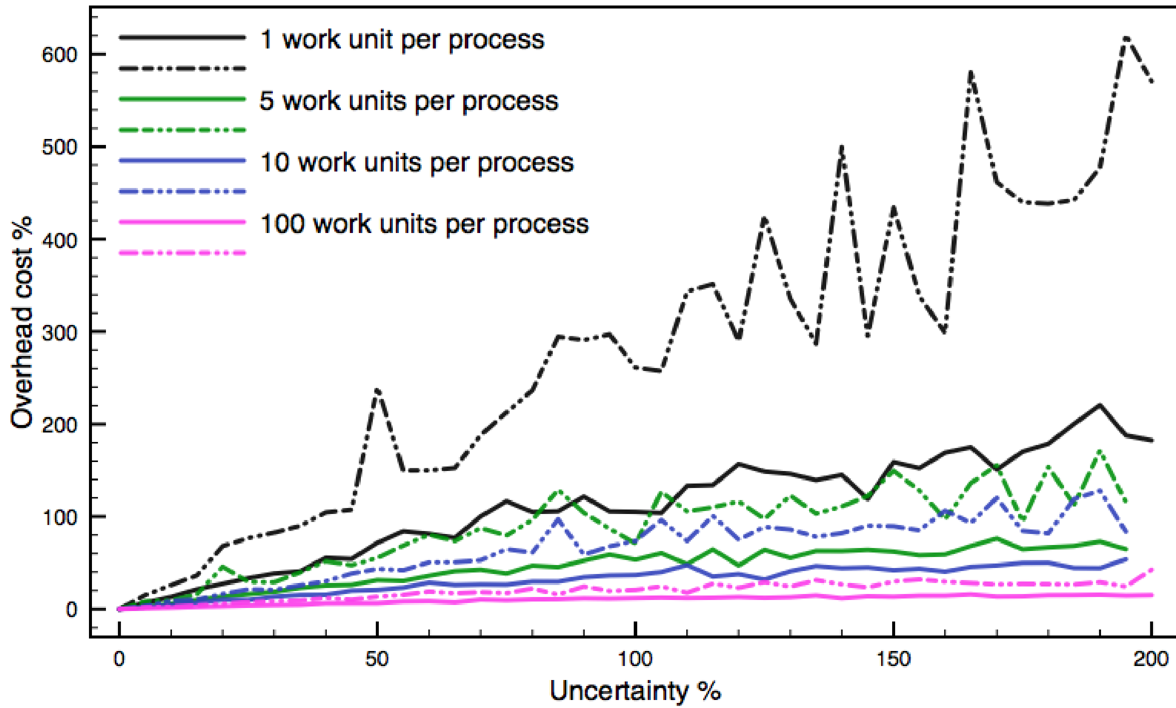


Figure 3.6 Over decomposition trend 2

3.2.3 Client-Server Load Balancing

Over-decomposition can drastically improve load balancing for work that can be divided into arbitrarily small units. However, it is often not feasible to divide real problems into small enough units to achieve sufficient load balancing. Dynamic load balancing offers a solution in this scenario. Instead of assigning each worker all of its work units up front, a load balancer can give out work units as workers become available. The load balancing simulator runs a server that generates all the work units for the simulation, and each worker queries the server when it finishes its previous work unit. Figure 3.7 shows how dynamic load balancing improves load balancing for simulations with five work units per worker. Furthermore, figure 3.8 shows that dynamically load

balanced simulations can achieve similar performance with an order of magnitude less work units.

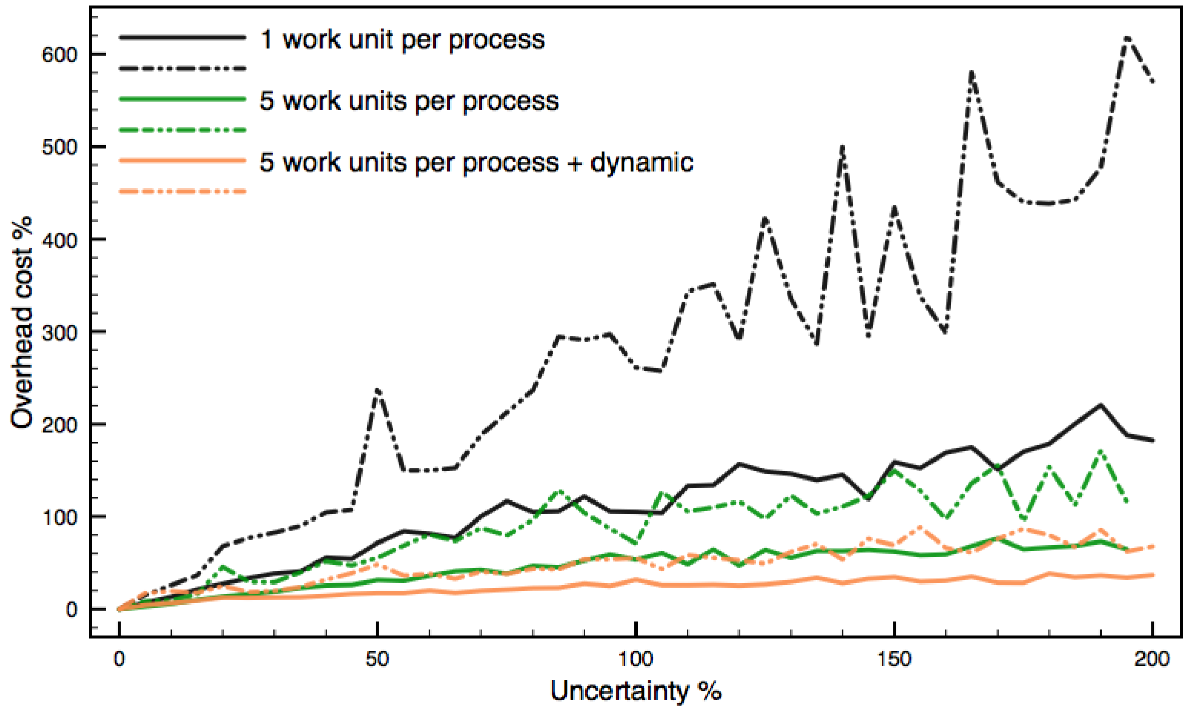


Figure 3.7 Dynamic trend 1

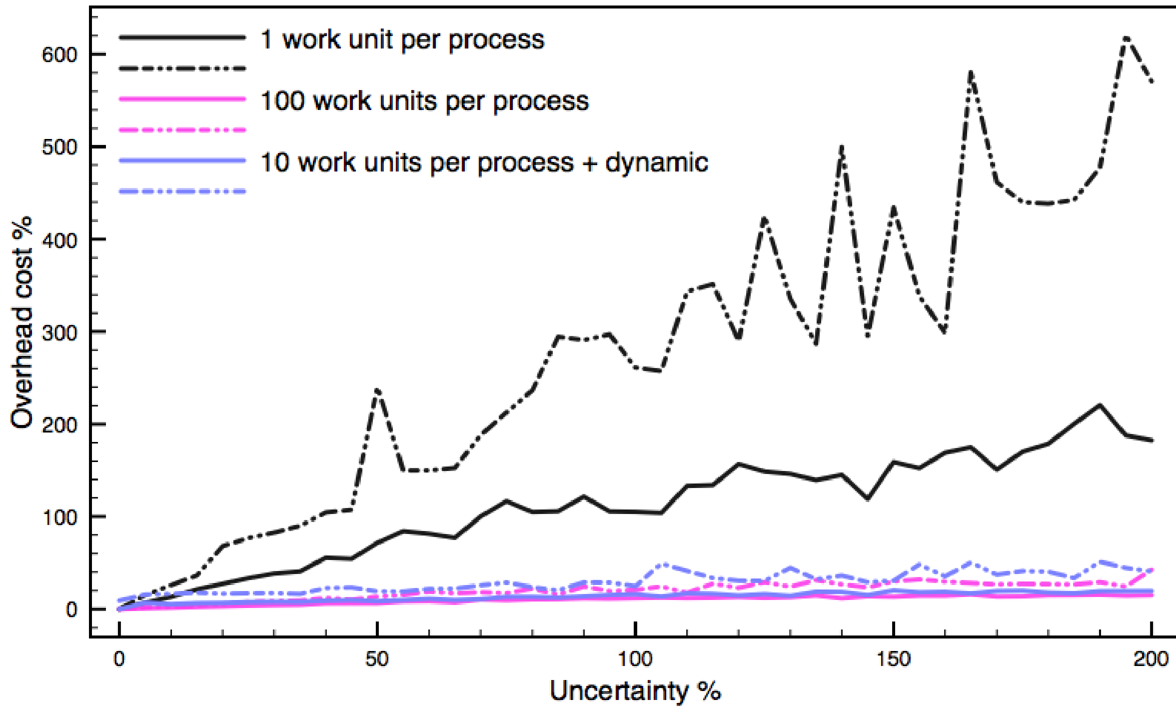


Figure 3.8 Dynamic trend 2

3.3 Recasting Domain Assembly

The previous section demonstrated the potential advantages of combining over-decomposition and dynamic load balancing in an ideal case. This section is dedicated to describing how the domain assembly process can be recast to realize these advantages, which is the primary distinguishing feature of this work.

Domain assembly requires intensive communication because a significant fraction of the global grid system may need to be transferred. Reducing the amount of communication overhead is therefore desirable. By its nature, dynamic load balancing already amortizes the grid communication cost because workers request small amounts of grid information when they begin each work

unit. If each worker only had a single work unit, all necessary grid information would have to be communicated at the same time.

3.3.1 Latency Hiding via Thread Oversubscription

If a single thread is running on each logical core, the core will be idle any time that thread is blocking for communication. If, however, multiple threads are running on each logical core (i.e., the core is oversubscribed), the operating system can hide some of the communication cost by switching out the blocked thread with another thread. This increases CPU utilization, thereby improving performance.

Domain assembly will only be amenable to oversubscription if it is first decomposed into several orthogonal operations. For this work, three such operations were identified that need to be performed on each process. First, each process owns a partition of the global mesh system, and is therefore responsible for providing grid information related to its partition upon request. Second, each process must participate in donor searching, because it represents the bulk of the computational cost. Third, each process is responsible for collecting candidate donor information for nodes that it owns. In this work, the root is a special node because it has one additional responsibility: running the load balancer that is responsible for distributing work units. Each of these operations can be performed completely independently of the others, and can therefore run on three separate threads for each process (four on the root). There is no shared mutable state between the threads, so the operating system can easily schedule the threads to increase CPU utilization.

3.3.2 Defining Appropriate Work Units

Identifying candidate donors for points in overlap regions is the most expensive operation during domain assembly and is therefore the most critical operation to efficiently parallelize. A domain assembly work unit can be defined in the following way. Given a set of query points and the appropriate grid information, a worker should yield a list of candidate donors for each point in the set. Here, the appropriate grid information is the set of all cells in the grid system that might contain any of the query points. Three challenges immediately present themselves. The first challenge is data locality. For a given set of query points, the cells that are required to perform the donor search are distributed across multiple processes, so communication of grid information is required. The second challenge is: how are query points and the necessary cells identified to begin with? The third challenge is related to the second and is: how does the load balancer send work units to workers?

Because every worker requests work units from the load balancer, the load balancer needs to be capable of fulfilling those requests quickly to avoid leaving workers idle. Work units should therefore be as lightweight as possible. An AABB (Axis Aligned Bounding Box) can be described by the coordinates of its minimum and maximum points, so its storage is compact (48 bytes), and sufficient to describe a domain assembly work unit: any point inside the AABB is a query point, and any cell that overlaps with the AABB is a potential candidate donor. Given a set of AABBs that describe the work units of the domain assembly problem, the load balancer can fulfill requests very efficiently.

3.3.3 Processing Work Units

When a worker receives an AABB that describes a work unit, the worker must first collect all of the points and cells that overlap with the AABB. Because the worker knows the AABB of each partition in the mesh system (defined during preprocessing), it can select which partitions possibly contain points and cells required for the current work unit. Then the worker requests a grid fragment from each process that owns a relevant partition. The request simply contains the AABB of the work unit, and the reply is the set of points and cells that overlap the AABB. After the worker receives all of the grid fragments it needs, it can then perform all of the donor searches necessary for the work unit. Finally, the worker sends candidate donor information for each point in the work unit to its respective owner.

CHAPTER 4

IMPLEMENTATION

The previous chapters described a new approach to dynamic load balancing for domain assembly at a conceptual level. This chapter details a specific implementation of that approach in a new parallel domain assembly code YOGA (Yoga is an Overset Grid Assembler). YOGA performs domain assembly in three phases which are, in order of execution: preprocessing, donor searching, and post processing. The phases are discussed in order, but first, a brief deviation is required. YOGA relies on, and cannot be sufficiently described in the absence of, two fundamental components: the data structure $MessagePasser :: Stream$, and the ZeroMQ messaging library. Therefore, these components must be introduced at the outset of the discussion.

4.1 Introduction to Stream and ZeroMQ

4.1.1 MessagePasser::Stream

Complex objects cannot be communicated across the network directly, and thus need to be deconstructed into communicable pieces on the sender and reconstructed by the receiver. MessagePasser::Stream is a class created at NASA Langley that facilitates the marshaling and unmarshaling of C++ objects via templated operators (<< and >> respectively). In the example below, an integer and a vector are copied into a MessagePasser::Stream with the << operator, and then back out with the >> operator.

```cpp
MessagePasser::Stream stream;
int a = 5;
std::vector<double> vec1 = {0.5, 0.6};

stream << a;
stream << vec1;

int b;
std::vector<double> vec2;
stream >> b;                    // now a == b
stream >> vec2;                 // now vec1 == vec2
```

Through template constraints, these operators will automatically work for POD (plain old data) and vectors of POD. We can use these operators to easily overload the << and >> operators for more complex data.

MessagePasser::Stream is a critical abstraction that makes it possible to preserve orthogonality between communication logic and other parts of the code. Specifically, there are multiple client-server relationships in YOGA, but there is only one generic implementation of a client and one generic implementation of a server. Because the client and server operate purely on Streams, they are completely agnostic to the type of data that they handle.

4.1.2 ZeroMQ

A side effect of performing load balancing inside of a single iteration is that a number of activities (which require communication) must be happening simultaneously and asynchronously. There are multiple messaging libraries that can facilitate asynchronous multithreaded communication, but the ZeroMQ message queueing library was chosen for this work (note that the MPI standard defines semantics for multithreaded communication, but support for this feature set is limited and implementation specific).

ZeroMQ is a low latency message queuing library that provides the user with a simple, but powerful socket programming interface. By removing complexities such as automatic reconnection, fair queueing, and asynchronous I/O, ZeroMQ allows the user application to focus on the task at hand rather than getting bogged down in communication details.

While there are several types of ZeroMQ sockets available, the REQ-REP socket pair is the simplest, and was sufficient for this work. A client creates a REQ socket and connects it to the REP socket of a server. Each communication between a client and server is synchronous and symmetric. The client calls $zmq_send()$ to make a request from the server, then calls $zmq_recv()$ to wait for the reply. The server calls $zmq_recv()$ to accept a request from a client and calls $zmq_send()$ to fulfill the request. The following subsections demonstrate how REQ and REP sockets are used to create the basic client-server relationship used throughout YOGA.

4.1.2.1 Generic Server

The first communication abstraction in YOGA is the server. There is only one server implementation in YOGA, and it is sufficiently general to support each type of server needed for this work. Below, is the declaration of the templated server class.

```cpp
template<typename Worker>
class Server {
public:
    Server(Worker& w,int channel);
    ~Server();
    void stop();
private:
    bool isRunning;
    Worker& worker;
    int serverChannel;
    int portNumber;
    zmq::context_t context;
    std::future<void> serverFuture;

    void run();
    bool isThereAMessage(zmq::socket_t& s);
};
```

The public interface for the Server class in YOGA is extremely simple, and consists only of a constructor, destructor, and stop function. In YOGA, different types of tasks are assigned to different "channels," which are associated with different port numbers for socket communication under the hood. Note that the Server launches a $std :: future$ in order to have the server running asynchronously from the calling thread. If the destructor or the stop method are called, the Server will shut down, and its thread will be joined.

One of the goals of this class is decouple the communication logic from the business logic of YOGA. The Server knows about ZeroMQ, and all of the required details of sending data between processes, but it is agnostic to the type of data it handles. The Server class is therefore templated on a a "Worker" who is responsible for everything in the problem domain. The server keeps a reference to its worker so that it may call upon the worker when necessary. The implementation of the Server's constructor is below:

```cpp
template<typename T>
ZMQMessager::Server<T>::Server(T &w, int channel)
    :isRunning(true),
     worker(w),
     serverChannel(channel),
     portNumber(getPortNumber(MessagePasser::Rank(),serverChannel)),
     context(1),
     serverFuture(std::async(std::launch::async,&Server::run,this))
{ }
```

The constructor stores a reference to the worker, sets the communication channel, determines the correct port number to use, creates a ZeroMQ context, and finally launches a separate thread via $std :: launch :: async$. The thread is bound to the server's $run()$ member function, which starts immediately upon being launched. The $run()$ function handles all of the communication of the server, and is shown below:

```cpp
template<typename T>
void ZMQMessager::Server<T>::run(){
    zmq::socket_t socket(context,ZMQ_REP);
    bindSocketToPort(socket,portNumber);
    while(isRunning){
        if(isThereAMessage(socket)){
            auto requestStream = receiveStream(socket);
            auto result = worker.doWork(requestStream);
            sendStream(socket,result);
        }
        else{
            std::this_thread::sleep_for(std::chrono::milliseconds(1));
        }
    }
    socket.close();
}
```

The $run()$ function creates a ZeroMQ socket, binds it to a port, and then starts running. The Server avoids busy waiting by sleeping until a message is received. When a message comes in, the Server receives the request as a Stream and passes that Stream off to the Worker. When the Worker completes its tasks, it returns a reply in the form of another Stream. The Server

sends the reply Stream back to the client. Because the server only deals with objects of type $MessagePasser :: Stream$, it is completely decoupled from the worker and the contents of the messages it handles. The $run()$ function also maintains a single level of abstraction (communicating streams over a socket) by wrapping the ZeroMQ details in functions. For example, the server must bind the socket to the specified port before it can start waiting for messages, which is done in the following function:

```
void ZMQMessager::bindSocketToPort(zmq::socket_t& s,int portNumber){
    std::string bindString = "tcp://*:" + std::to_string(portNumber);
    try {
        int zero = 0;
        s.setsockopt(ZMQ_LINGER,(void*)&zero,sizeof(int));
        s.bind(bindString.c_str());
    }
    catch(...){
        throw std::logic_error("Caught zmq error during bind");
    }
}
```

While the server is waiting for client requests, it alternates between sleeping briefly, and checking for new messages via $isThereANewMessage()$ which wraps the ZeroMQ mechanics for message polling:

```
template<typename T>
bool ZMQMessager::Server<T>::isThereAMessage(zmq::socket_t& s){
    zmq::pollitem_t items [] = { {(void*)s, 0, ZMQ_POLLIN, 0} } ;
    zmq::poll(items, 1, 0);
    return (bool) (items[0].revents & ZMQ_POLLIN);
}
```

4.1.2.2 Sending and Receiving Streams

The Server and Client classes handle requests in terms of MessagePasser::Stream variables via the methods *sendStream* and *receiveStream*. These are the functions that actually interact with ZeroMQ. The *sendStream* function is shown below.

```
void ZMQMessager::sendStream(zmq::socket_t& socket,
                             MessagePasser::Stream& s){
    do{
        auto frame = extractMessageFrameFromStream(s);
        socket.send(frame, s.empty()? 0 : ZMQ_SNDMORE);
    }while(not s.empty());
}
```

If the Stream contains multiple objects, each object is extracted and placed into a frame that is then "sent" over the $zmq :: socket_t$. Note that ZeroMQ does not actually send the message until the last frame is "sent" in order to preserve the semantics that messages are received in their entirety or not at all. The flag ZMQ_SNDMORE is used to tell ZeroMQ that more message frames are coming, and the last frame is marked with 0. The *extractMessageFrameFromStream*, as its name suggests, builds a $zmq :: message_t$ from the next item in a stream (shown below).

```
zmq::message_t ZMQMessager::extractMessageFrameFromStream(MessagePasser::
    Stream &s){
    std::vector<char> buffer;
    if(not s.empty())
        s >> buffer;
    zmq::message_t m(buffer.size());
    memcpy(m.data(),buffer.data(),buffer.size());
    return m;
}
```

The complementary function *receiveStream* receives an incoming ZeroMQ message, unpacks each frame of the message, adds the frame to a Stream, then returns the Stream. The

receiveStream functions is shown below.

```
MessagePasser::Stream ZMQMessager::receiveStream(zmq::socket_t& socket){
    MessagePasser::Stream s;
    do {
        zmq::message_t m;
        socket.recv(&m);
        s << unpackMessageIntoChars(m);
    }
    while(isThereMore(socket));
    return s;
}
```

The *unpackMessageIntoChars* function simply extracts the bytes from a ZeroMQ message frame into a vector of chars and returns the vector (shown below).

```
std::vector<char> ZMQMessager::unpackMessageIntoChars(zmq::message_t& m){
    return std::vector<char>((char*)m.data(),(char*)m.data()+m.size());
}
```

4.1.2.3 Generic Client

The second communication abstraction in YOGA is the client. The Client class in YOGA is similar in spirit and complementary to the Server class. The declaration for the Client is below.

```cpp
class Client{
public:
    Client(int channel);
    void connectToServer(int id);
    void disconnectFromServer(int id);
    void stop();
    MessagePasser::Stream makeRequest(int serverId,MessagePasser::
       Stream& request);
private:
    int serverChannel;
    zmq::context_t context;
    zmq::socket_t socket;
};
```

Similarly to the Server, the Client constructor takes a channel as input, which is used to determine communication port numbers internally. The Client also has a simple public interface. The user can connect it to a server (based on id, which is analogous to mpi rank), disconnect from a server, stop (close the socket), and make a request to a server to which it is connected. The request function is below.

```cpp
MessagePasser::Stream ZMQMessager::Client::makeRequest(
                        int serverId,
                        MessagePasser::Stream& request){
    sendStream(socket,request);
    return receiveStream(socket);
}
```

The $makeRequest()$ function simply takes a stream, sends it across the socket layer to the server, and returns the servers response stream back to the caller. $MessagePasser :: Stream$ allows the client to send messages of arbitrary size and type, hence this is the only client implementation in YOGA.

4.2 PreProcessing

Preprocessing is the first of the three phases of Overset domain assembly in YOGA. This phase is responsible for creating the load balancer, hole maps, and mesh metadata that are necessary for performing the donor searching phase in a load balanced setting.

4.2.0.4 Global Mesh System Meta Data

Each process in YOGA needs to know how many geometric bodies, component grids, and partitions are in the global mesh system. Each process also needs bounding boxes for each of those entities. This information is generated by a series of extent box exchanges between processes and then stored in an object with the following public interface:

```
int numberOfBodies()const;
int getComponentIdForBody(int i) const;
int numberOfComponents() const;
int numberOfPartitions() const;
Parfait::Extent<double> getBodyExtent(int id)const;
Parfait::Extent<double> getComponentExtent(int id)const;
Parfait::Extent<double> getPartitionExtent(int id) const;
```

Note that $Parfait$ is an in house library at NASA Langley that provides some grid related utilities (e.g., $Extent$, which is an implementation of an AABB).

4.2.0.5 Approximate Distance Field

YOGA uses distance to the wall to determine intergrid boundary locations. If the flow solver does not provide distance to the wall for each node, YOGA will generate an approximate distance field. In order to generate a distance field for each partition, each process needs to know the locations of the surface nodes of each body. The $ParellelSurface$ class gathers the surface

nodes for each component grid via the following static function.

```
template <typename MeshType>
static std::vector<Parfait::Point<double>> getSurfaceNodesForComponent(
    MeshType& m,int component){
     std::vector<Parfait::Point<double>> allPoints;
     auto myPoints = getLocalSurfacePointsInComponent(m,component);
     MessagePasser::AllGatherv(myPoints,allPoints);
     return allPoints;
}
```

For a particular component grid, each process creates a vector of the points in its partition that belong to the surface for that component. Then all the vectors are combined on every process by means of a wrapper for $MPI_Allgatherv$. The vector of local points is built by:

```
template<typename MeshType>
static std::vector<Parfait::Point<double>>
    getLocalSurfacePointsInComponent(MeshType& m,int component){
     auto isNodeInThisSurface = generateNodeMask(m,component);
     std::vector<Parfait::Point<double>> myPoints;
     for(int i=0;i<m.numberOfNodes();++i)
         if(isNodeInThisSurface[i])
             myPoints.push_back(m.getNode(i));
     return myPoints;
}
```

4.2.1 Load Balancer

In YOGA, a load balancer is responsible for generating work units for the donor searching phase. Multiple load balancing strategies were explored throughout the development of YOGA, and each implementation derives from the following abstract base class:

```cpp
class LoadBalancer{
public:
    virtual int getRemainingVoxelCount() = 0;
    virtual Parfait::Extent<double> getWorkVoxel() = 0;
};
```

The abstraction for a load balancer is simply to provide the next work unit upon request and to tell how many work units remain. YOGA's current load balancer employs auxiliary Cartesian grids to recursively subdivide the domain into appropriately sized work units. Work units are stored in a priority queue as an estimated cost paired with an AABB (Axis Aligned Bounding Box):

```cpp
    typedef Parfait::Extent<double> Extent;
    typedef std::pair<int,Extent> Pair;
    class Compare{
    public:
        bool operator() (Pair& A,Pair& B){return A.first < B.first;}
    };
    std::priority_queue<Pair,std::vector<Pair>,Compare> workVoxels;
```

The priority queue orders the work units based one the *Compare* functor, which simply compares the estimated size of two work units. By ordering the work units by estimated size, the load balancer can ensure that the most expensive work units are completed first, and the smaller work units can fill in the gaps at the end.

As shown in the constructor below, the load balancer begins by creating an N^3 Cartesian grid, and estimating the cost of each cell. Then it determines which cells have potential overlap, creates work units from them, and pushes the work units into the priority queue.

```cpp
template<typename MeshType>
CartesianLoadBalancer::CartesianLoadBalancer(MeshType& mesh,
                                             MeshSystemInfo& info,
                                                    int number_of_ranks,
                                                    int N)
{
    Parfait::CartBlock block(getExtentOfSystem(info),N,N,N);
    auto nodeCountPerCell = MeshDensityEstimator::
       tallyNodesContainedByCartCells(mesh,block);
    auto is_in_overlap_region = createCellMask(info,block);
    for(int i=0;i<block.numberOfCells();++i){
        if(1 == is_in_overlap_region[i])
            workVoxels.push(std::make_pair(nodeCountPerCell[i],block.
               createExtentFromCell(i)));
    }
    refine(mesh, info);
}
```

The last call in the constructor, $refine()$, initiates the recursive subdivision of the work units in the priority queue. The load balancer lives on the root, but must communicate with every process to obtain mesh density estimates each time it refines, so it is unfortunately long. The current version of refinement is listed below.

```
    template<typename MeshType>                                                          1
    void CartesianLoadBalancer::refine(MeshType& mesh, MeshSystemInfo& info) {           2
        int targetNodesPerVoxel = 50000;                                                 3
        bool needRefinement = true;                                                      4
        while(needRefinement) {                                                          5
            int refineFlag = 0;                                                          6
            std::vector<int> refineDimensions;                                           7
            Parfait::Extent<double> refineCell;                                          8
            if (MessagePasser::Rank() == 0) {                                            9
                int density = workVoxels.top().first;                                    10
                if(density > targetNodesPerVoxel) {                                      11
                    refineFlag = 1;                                                      12
                    int targetChunkCount = std::max(2,density/                           13
                        targetNodesPerVoxel);  // at least cut the voxel in
                        half
                    refineDimensions = calcRefineDimensions(targetChunkCount,            14
                        workVoxels.top().second);
                    refineCell = workVoxels.top().second;                                15
                    workVoxels.pop();                                                    16
                }                                                                        17
            }                                                                            18
            MessagePasser::Broadcast(refineFlag,0);                                      19
            if(1 == refineFlag) {                                                        20
                MessagePasser::Broadcast(refineDimensions,0);                            21
                MessagePasser::Broadcast(refineCell,0);                                  22
                Parfait::CartBlock block(refineCell,refineDimensions[0],                 23
                    refineDimensions[1],refineDimensions[2]);
                auto nodeCountPerCell = MeshDensityEstimator::                           24
                    tallyNodesContainedByCartCells(mesh,block);
                if(MessagePasser::Rank() == 0) {                                         25
                    for (int i = 0; i < block.numberOfCells(); ++i)                      26
                        workVoxels.push(std::make_pair(nodeCountPerCell[i],              27
                            block.createExtentFromCell(i)));
                }                                                                        28
            }                                                                            29
            else{                                                                        30
                needRefinement = false;                                                  31
            }                                                                            32
        }                                                                                33
}                                                                                        34
```

The cost of a work unit is approximated by the number of nodes contained in its AABB. Work units are subdivided until all of them have less than $50k$ nodes. This should be a tunable parameter in future work, but was found to be sufficient for the cases in the current study. Refinement is initiated in a while loop that terminates when the work unit at the front of the priority queue (i.e.,

the largest) is below the threshold. At each iteration, the root process inspects the work unit at the front of the queue to check its node count. If the node count is above the threshold, it determines how to subdivide the work unit (i.e., the dimensions of the Cartesian grid). The root process then pops the old work unit off the queue, and broadcasts the AABB and calculated dimensions to all processes. Each process then constructs a Cartesian grid based on that recipe. Finally, a node count for each cell of this new Cartesian grid is obtained from the $MeshDensityEstimator$, and new work units are pushed into the queue.

4.2.1.1 Estimating Mesh Density

The load balancer in YOGA uses estimates of mesh density in overlap regions to improve the quality of its work distribution. YOGA has a class, $MeshDensityEstimator$, that performs this function. It has a single public static function (listed below) that is to be called on every process. This function takes a mesh and a Cartesian grid, and returns a vector of integers on the root process with the total number of nodes contained in each cell of the Cartesian grid.

```
template<typename MeshType>
static std::vector<int> tallyNodesContainedByCartCells(MeshType& mesh,
    Parfait::CartBlock& block){
     auto localTally = tallyNodesLocally(mesh,block);
     return collectAndSumOnRoot(localTally);
}
```

Each process first tallies the number of nodes in each Cartesian cell (based on its local partition). Then each process sends its local tally to the root, who combines them to obtain the final estimate.

4.2.2 Parallel Hole Map

The final preprocessing step that YOGA performs is to create hole maps, which are used to approximate geometric bodies. The hole maps contain Cartesian grids whose cells are tagged as “in” “out” or “crossing” (this is how the geometry is approximated). They are constructed in parallel in the following steps:

- Tag crossing cells locally
- Communicate crossing cell ids
- Flood fill in/out statuses

When YOGA constructs hole maps, the extent box for each piece of geometry is already known, and the resolution for each hole map is already set. Each process creates bounding boxes for its surface elements, then tags any cells in the corresponding Cartesian grid that intersect each bounding box. Once all processes have marked crossing cells locally, a parallel max is performed on the tags so that all processes have the same cells tagged as crossing. Once each process has the same view of the hole map, the remaining unmarked cells of the hole maps can be marked as “in” or “out” with no further communication via a flood fill algorithm.

4.2.2.1 Implicit Outer Boundary

The cells of the hole map that intersect the geometry are marked first, then the remaining cells are classified as in or out. If the hole map is constructed such that the geometry does not cross any of the cells in the outermost layer of the hole map, those cells can be used as seeds to a flood

fill algorithm to mark all of the “in” cells (in the computational domain). All remaining cells must be in the hole region, and are therefore marked as outside the domain.

While it is convenient to have an “extra” layer of cells in the hole map to use as seeds to the flood fill algorithm, it is not strictly necessary. In fact, the total number of cells in the hole map can be reduced by about 40% with no loss in the resolution of the approximated geometry (thereby reducing the storage and computational cost to build the hole map). The key is to simply let the outer layer exist implicitly. Then the cells in the actual layer can be used as seeds, but just need to be first checked that they are not already marked as crossing cells.

If symmetry planes exist, any hole map cells that intersect symmetry planes are ineligible to be seeds, which prevents the flood fill from marking the cells inside symmetric geometries (e.g., wings or fuselages) improperly.

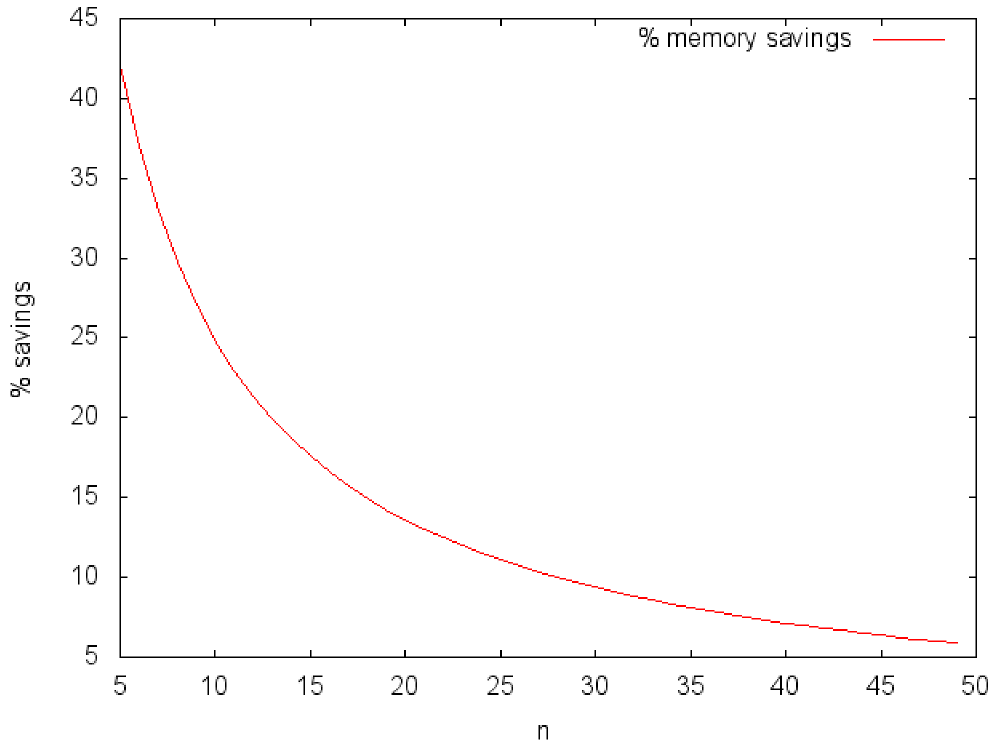


Figure 4.1 Hole map memory reduction

4.3 Donor Searching with Dynamic Load Balancing

In YOGA, dynamic load balancing is applied only to donor searching because the cost of Overset domain assembly is dominated by donor searching. There are four orthogonal actors during this phase: voxel server, grid server, donor finder, and donor collector. Each of which will be discussed in the following subsections.

4.3.1 Voxel Server

The work units for donor searching are defined by the load balancer, but they are actually distributed by the voxel server. The voxel server is implemented via the generic ZeroMQ server

discussed in section 4.1.2.1, and is instantiated on the root process by:

```
ZMQMessager::Server<decltype(voxelServer)>(voxelServer,
                                           MessageTypes::WorkRequest);
```

As discussed in section 4.1.2.1, the generic server only requires that the object (e.g., $voxelServer$) has a function $doWork()$ that takes as input and also returns a $MessagePasser :: Stream$. The implementation of the $VoxelServer$ class is shown below in its entirety.

```
class VoxelServer{
public:
    VoxelServer(LoadBalancer& L)
        :loadBalancer(L)
    { }
    MessagePasser::Stream doWork(MessagePasser::Stream& stream){
        MessagePasser::Stream reply;
        reply << loadBalancer.getRemainingVoxelCount();
        reply << getNextVoxel();
        return reply;
    }
private:
    LoadBalancer& loadBalancer;
    Parfait::Extent<double> getNextVoxel(){
        return 0 < loadBalancer.getRemainingVoxelCount() ?
                   loadBalancer.getWorkVoxel() :
                   Parfait::ExtentBuilder::createEmptyBuildableExtent(
                   Parfait::Extent<double>());
    }
};
```

When a work unit is requested, the $VoxelServer$ first asks the $LoadBalancer$ how many work units remain. If there are any remaining work units, it asks for the next work unit and puts it into the reply Stream. If there are no work units remaining, $VoxelServer$ puts an empty Extent into the reply Stream instead.

4.3.2 Worker

Each process creates a *Worker* object that is responsible for actually performing the work associated with each work unit. The *Worker* has a single function, which it launches in a separate thread, and it is listed below.

```
void work(){
    ZMQMessager::Client voxelClient(MessageTypes::WorkRequest);
    voxelClient.connectToServer(voxelServerId);
    while(true){
        MessagePasser::Stream emptyStream;
        auto reply = voxelClient.makeRequest(voxelServerId,emptyStream
            );
        Parfait::Extent<double> e;
        int nLeft;
        reply >> nLeft;
        if(0 >= nLeft) {
            voxelClient.stop();
            break;
        }
        reply >> e;
        auto workVoxel = WorkVoxelBuilder::build(meshSystemInfo,e);
        auto receptorsAndHoles = voxelDonorFinder.
            getCandidateDonorsAndHoleIds(workVoxel,e,holeMaps);
        auto candidateDonors = receptorsAndHoles.first;
        auto holes = receptorsAndHoles.second;
        DonorDistributor::distribute(candidateDonors);
        HolePointDistributor::distribute(workVoxel, holes);
    }
}
```

First, the worker creates a Client, and connects it to the VoxelServer. Then it enters a while loop, in which the work is actually done. Each iteration, the worker first makes a request to the work server. The reply contains an integer (the number of remaining work units), and a bounding box (the region of space for which the worker should perform domain assembly). If there are no remaining work voxels, the worker is finished, and it exits the while loop. Otherwise, it gets the bounding box out of the reply stream, and passes it to the $buildWorkVoxel()$

function, which is responsible for gathering all of the required grid information the correct processes. The $buildWorkVoxel()$ function returns a $WorkVoxel$, which contains all of the data required to identify candidate donors and hole points. The worker then passes the $WorkVoxel$ to the $VoxelDonorFinder :: getCandidateDonors()$ function, which identifies hole points and candidate donors. The $DonorDistributor$ is then responsible for sending donor information to the correct processes. Likewise, the $HolePointDistributor$ determines which processes own each of the hole points, then sends them.

4.3.2.1 Building A Work Voxel

The helper class $WorkVoxelBuilder$ has a single public static function:

```
static WorkVoxel build(MeshSystemInfo &info, Parfait::Extent<double> &e) {
    std::map<int, VoxelFragment> fragments;
    for (int id:getIdsOfServersToQuery(info,e))
        fragments[id] = requestFragmentFromGridServer(id,e);
    return createWorkVoxelFromFragments(fragments,e);
}
```

Given metadata for the mesh system and an AABB, the $WorkVoxelBuilder$ will identify grid servers to query, request a grid fragment from each, then combine the grid fragments into a $WorkVoxel$. Determining which grid servers to query is simply a matter of checking AABB intersections with each partition (partitions are tied to servers):

```
static std::vector<int> getIdsOfServersToQuery(MeshSystemInfo& info,
                                               Parfait::Extent<double>& e){
    std::vector<int> ids;
    for (int i = 0; i < info.numberOfPartitions(); ++i)
        if (e.contains(info.getPartitionExtent(i)))
            ids.push_back(i);
    return ids;
}
```

Querying a grid server is straightforward:

```
static VoxelFragment requestFragmentFromGridServer(int id, Parfait::
    Extent<double>& e){
    ZMQMessager::Client gridClient(MessageTypes::GridRequest);
    gridClient.connectToServer(id);
    MessagePasser::Stream request;
    request << e;
    auto reply = gridClient.makeRequest(id, request);
    VoxelFragment fragment;
    reply >> fragment;
    gridClient.disconnectFromServer(id);
    return fragment;
}
```

The $WorkVoxelBuilder$ first creates a client and connects it to the designated grid server. Then it packs the AABB into a request stream and sends the stream to the server. When it receives the reply stream from the grid server, the $WorkVoxelBuilder$ unpacks it into a grid fragment, and returns the fragment.

After the $WorkVoxelBuilder$ collects all relevant grid fragments, it combines them into a $WorkVoxel$ (i.e., the data structure that contains all the grid information necessary for donor searching and hole point identification).

```cpp
static WorkVoxel createWorkVoxelFromFragments(std::map<int,VoxelFragment
   >& fragments, Parfait::Extent<double>& e){
    WorkVoxel workVoxel(e);
    for (auto &pair:fragments) {
        auto &fragment = pair.second;
        workVoxel.addNodes(fragment.transferNodes);
    }
    for (auto &pair:fragments) {
        auto &fragment = pair.second;
        workVoxel.addTets(fragment.transferTets);
        workVoxel.addPyramids(fragment.transferPyramids);
        workVoxel.addPrisms(fragment.transferPrisms);
        workVoxel.addHexs(fragment.transferHexs);
        workVoxel.addCellIds(fragment.transferCellIds);
        workVoxel.addTriangles(fragment.transferTriangles);
    }
    return workVoxel;
}
```

4.3.2.2 Finding donors and identifying hole points

Once a work voxel is constructed, the real work can begin. For each node in the work voxel, a list of candidate donors must be created. Additionally, nodes which lie inside a piece of geometry should be identified. The following function performs both tasks.

```
template<typename HoleMapType>
std::pair<std::vector<Receptor>,std::vector<int>> VoxelDonorFinder::
   getCandidateDonorsAndHoleIds(WorkVoxel &workVoxel,

   Parfait::Extent<double> &extent,

   std::vector<HoleMapType> &holeMaps){

    auto outOfVoxelNodeIds = getIdsOfNodesOutsideVoxel(workVoxel);
    auto candidateDonors = buildCandidateDonorList(workVoxel);

    removeCandidatesWhoAreFartherFromTheSurface(workVoxel.nodes,
        candidateDonors);

   auto potentialHoleNodeIds = getIdsOfHoleNodes(workVoxel,holeMaps);
    auto actualHoleIds = getActualHoleIds(potentialHoleNodeIds,
        candidateDonors);
    auto candidateReceptors = buildCandidateReceptors(workVoxel,
        candidateDonors);

    return std::make_pair(candidateReceptors,actualHoleIds);
}
```

A work voxel must contain all cells that overlap with the voxel. Some of those cells may have nodes that are outside of the voxel and are therefore the responsibility of another work voxel. They must be therefore identified and ignored. Next, all candidate donors are found for each node in the work voxel. The candidate donor list for each node contains all cells that contain the node, but since YOGA uses distance to the wall to determine where interpolation boundaries should exist not all donors are relevant. Specifically, nodes should only interpolate from donors who are closer to a piece of geometry than the node itself.

Hole nodes—nodes that lie inside a solid surface—are identified by using a method proposed by Sitaraman [38]. First, approximate hole maps are used to identify potential hole nodes. Then, actual hole nodes are easily identified based on the following observation: any potential hole nodes that have donors from the grid associated with the solid surface must be outside the surface.

Finally, candidate donors and hole node ids are packaged up and returned.

4.3.2.3 Distributing

The *DonorDistributor* class is responsible for sending candidate donor lists to the owner of each node that has at least one donor candidate.

```
static void distribute(std::vector<Receptor>& receptorUpdates){
    auto ownerToReceptors = mapReceptorsToOwners(receptorUpdates);

    for(auto& stuffForOwner:ownerToReceptors){
        int messageType = DonorCollector::Receptors;
        int owningRank = stuffForOwner.first;
        auto& receptors = stuffForOwner.second;
        ZMQMessager::Client client(DciUpdate);
        client.connectToServer(owningRank);
        MessagePasser::Stream stream;
        stream << messageType;
        stream << receptors;
        client.makeRequest(owningRank,stream);
        client.disconnectFromServer(owningRank);
    }
}
```

First, the *DonorDistributor* maps receptors to their owners. Then, for each owner, the *DonorDistributor* creates a client, connects it to the appropriate server, packs the receptors into a request stream, and sends the request to the server.

The *HolePointDistributor* performs the same task for the hole ids.

```
static void distribute(WorkVoxel& workVoxel, const std::vector<int>&
    holeNodes){
    auto ownerToHoles = mapHolesToOwners(workVoxel,holeNodes);

    for(auto& x:ownerToHoles){
        int messageType = DonorCollector::Holes;
        int owningRank = x.first;
        auto& holes = x.second;
        ZMQMessager::Client client(DciUpdate);
        client.connectToServer(owningRank);
        MessagePasser::Stream stream;
        stream << messageType;
        stream << holes;
        client.makeRequest(owningRank,stream);
        client.disconnectFromServer(owningRank);
    }
}
```

4.3.3 Grid Server

Each process runs a server that can be queried by any process asynchronously during domain assembly. The *GridFetcher* class is responsible for fulfilling these queries, and its code is listed below:

```cpp
template<typename MeshType>
class GridFetcher{
public:
    GridFetcher(const MeshType& m,PartitionInfo& info,int rank) :
        mesh(m),
        partitionInfo(info),
        my_rank(rank)
    { }
    MessagePasser::Stream doWork(MessagePasser::Stream& stream){
        Tracer::beginEvent("GridServer: fetching fragment");
        Parfait::Extent<double> e;
        stream >> e;
        VoxelFragment fragment(mesh,partitionInfo,e,my_rank);
        MessagePasser::Stream result;
        result << fragment;
        Tracer::endEvent("GridServer: fetching fragment");
        return result;
    }
private:
    const MeshType& mesh;
    PartitionInfo& partitionInfo;
    int my_rank;

};
```

When the *GridFetcher* receives a request, it first unpacks it into a bounding box. Then it creates a *VoxelFragment*, which contains all of the nodes and cells that overlap with the bounding box (on this particular partition). Finally, the *GridFetcher* packs the fragment into a Stream and sends it back to the client.

4.3.4 Donor Collector Server

The *DonorCollector* class is responsible for gathering and storing candidate donors and hole ids. It fits the generic server template, and its *doWork*() function is listed below.

```cpp
MessagePasser::Stream doWork(MessagePasser::Stream& request){
        int messageType;
        request >> messageType;
        if(Receptors == messageType) {
            std::vector<Receptor> newReceptors;
            request >> newReceptors;
            storeDonorUpdate(newReceptors);
        }
        else if(Holes == messageType){
            std::vector<long> holePoints;
            request >> holePoints;
            holeNodes.insert(holeNodes.end(),
                             holePoints.begin(),
                             holePoints.end());
        }
        MessagePasser::Stream emptyReply;
        return emptyReply;
    }
```

This server is slightly different from the others. First, it handles more than one type of request based on an integer tag that is prepended to the request stream. Second, its reply is always empty. The clients are just pushing data to the server, so an empty reply is sufficient to indicate that the data was received.

4.4 Post Processing

4.4.1 Interpolation in Yoga

The initial version of YOGA uses barycentric coordinates to compute donor weights on tetrahedral meshes. The design and data structures in YOGA are flexible enough to handle other element types, but general interpolation is beyond the scope of this work and is therefore not covered here.

4.4.1.1 Math

Consider a tetrahedron with vertices $v_n = (x_n, y_n, z_n)$. The barycentric coordinates λ_n of a point p can be calculated by

$$\begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \end{bmatrix} = T^{-1} * (p - v_4), \qquad \lambda_4 = 1 - \lambda_1 - \lambda_2 - \lambda_3 \tag{4.1}$$

where T is the $3X3$ matrix:

$$T = \begin{bmatrix} x_1 - x_4 & x_2 - x_4 & x_3 - x_4 \\ y_1 - y_4 & y_2 - y_4 & y_3 - y_4 \\ z_1 - z_4 & z_2 - z_4 & z_3 - z_4 \end{bmatrix} \tag{4.2}$$

4.4.1.2 Implementation

The function that calculates barycentric coordinates in YOGA is based on the above description, and follows it nearly verbatim. Note that YOGA uses the Eigen library for all its linear algebra, [39].

```cpp
template<typename TetType, typename PointType>
std::array<double, 4> calculateBarycentricCoordinates(const TetType &tet,
    const PointType &point) {

    std::array<double, 4> x = {tet[0][0], tet[1][0], tet[2][0], tet[3][0]};
    std::array<double, 4> y = {tet[0][1], tet[1][1], tet[2][1], tet[3][1]};
    std::array<double, 4> z = {tet[0][2], tet[1][2], tet[2][2], tet[3][2]};

    Eigen::Vector3d p;
    p << point[0], point[1], point[2];

    Eigen::Vector3d v4;
    v4 << x[3], y[3], z[3];

    Eigen::Matrix3d T;
    T  <<   x[0] - x[3], x[1] - x[3], x[2] - x[3],
            y[0] - y[3], y[1] - y[3], y[2] - y[3],
            z[0] - z[3], z[1] - z[3], z[2] - z[3];

    auto lambdas = T.inverse() * (p - v4);

    return {lambdas[0], lambdas[1], lambdas[2], 1.0-lambdas[0]-lambdas[1]-
        lambdas[2]};
}
```

4.4.1.3 Verification

A correct implementation of interpolation based on barycentric coordinates must have second order accuracy. The order property of the interpolation accuracy in YOGA was demonstrated with a mesh refinement study. An Overset system was constructed, which consisted of a uniform receptor grid contained by a background uniform donor grid. The same receptor grid was used with a series of six donor grids of different resolutions. The nonlinear function

$$f(x,y,z) = 6\sin(x) + 12\cos(y) + \frac{1}{2}e^{z} - 0.3; \tag{4.3}$$

was calculated on the donor grid, then interpolated to the receptor grid. Then the largest interpolation error (compared to the exact solution), and the longest edge in the donor grid were measured. The following $log\ log$ plot shows that the interpolation in YOGA is second order.

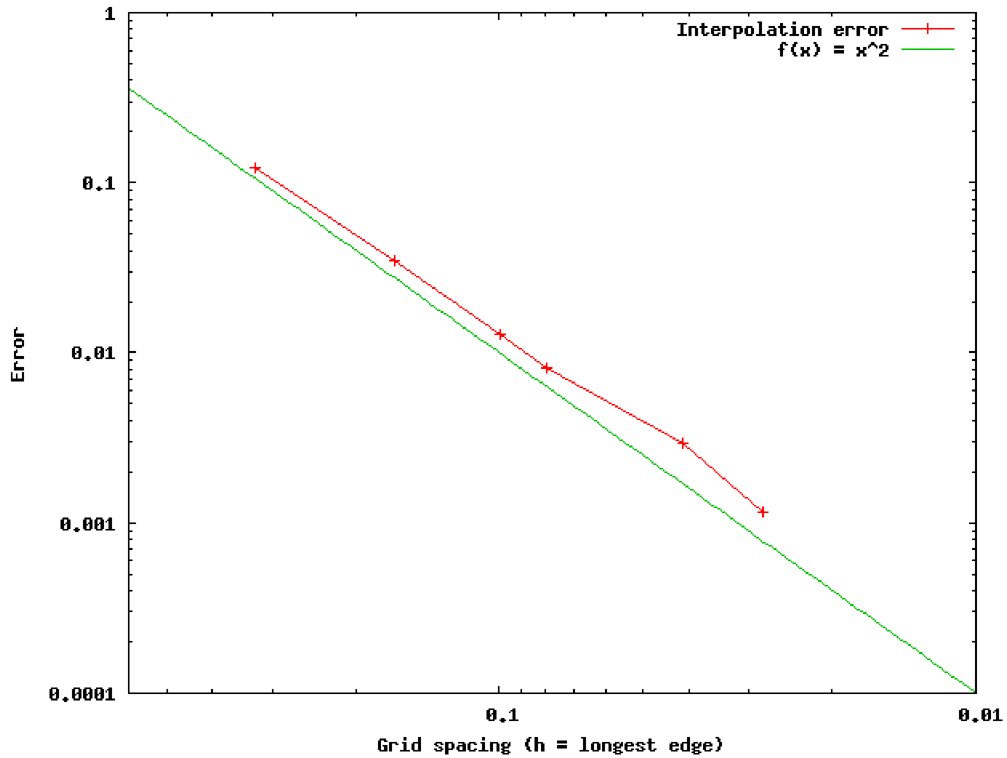


Figure 4.2 Interpolation convergence

4.4.2 Tracer

Obtaining and analyzing performance data for multithreaded and distributed codes can be cumbersome. There are a number of tools available to instrument code and visualize the data, but they often place requirements on compilation that interfere with third party libraries (etc.).

Google Chrome has a built in tool called Trace Viewer for analyzing website performance (at microsecond resolution). The Trace Viewer takes json data as input, so it can visualize performance data from any source that can generate fits a simple format. Matthew O'Connell at NASA Langley developed a simple tool to let users easily leverage Trace Viewer in their applications. Specifically, the following two functions can be used to record the beginning and end of events in an application. These functions are terse enough not to be a distraction, require only the inclusion of a single header, and are thread safe.

```
Tracer::beginEvent("Grid client: requesting grid");
// do work
Tracer::endEvent("Grid client: requesting grid");
```

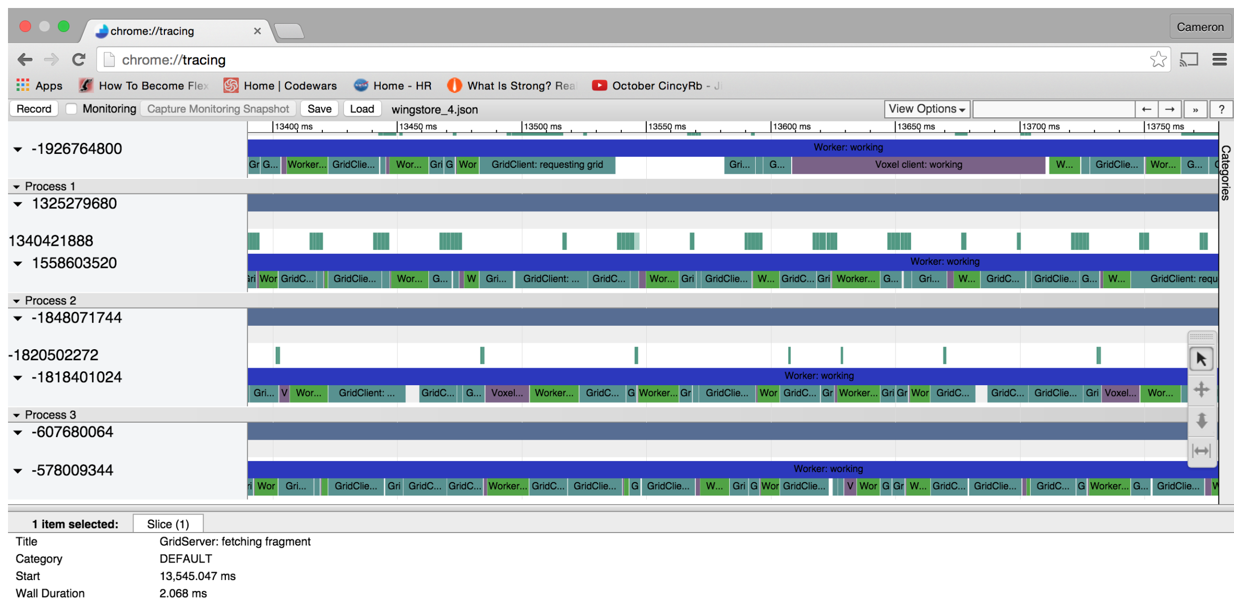


Figure 4.3 Tracing screenshot

CHAPTER 5

RESULTS

5.1 Store Separation Case

The first test case is a very coarse grid system for a dynamic store separation simulation (about 100 thousand nodes). Due to the small size of the grid system, this case could be easily run on a standard workstation in a matter of seconds, which was quite useful during the early stages of YOGA's development.

Figure 5.1 shows the surface meshes of the wing and store for this case.

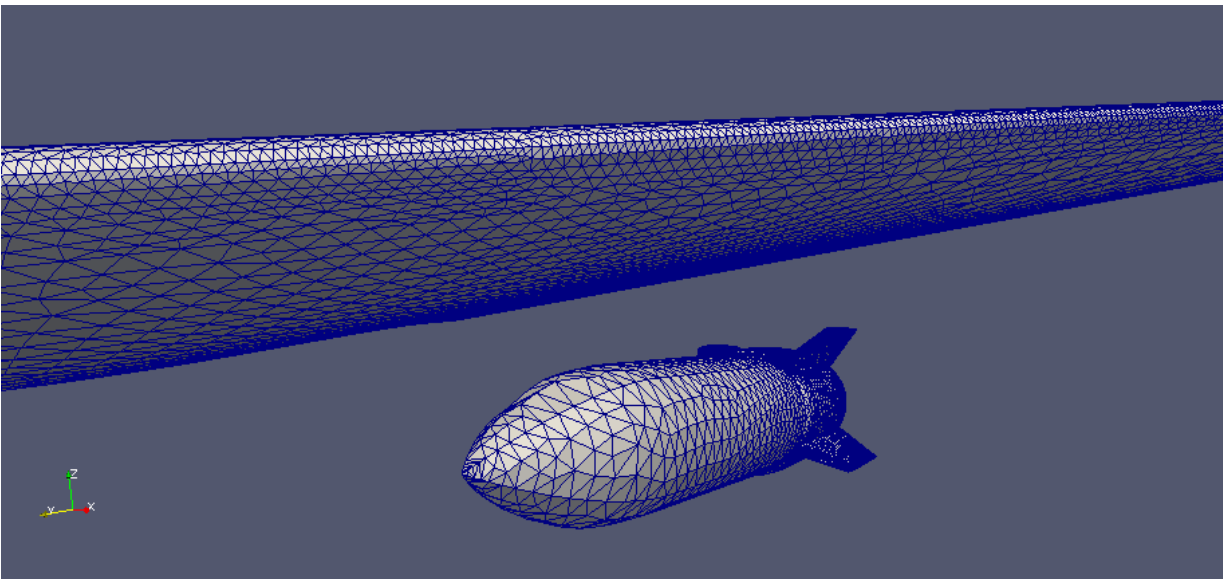


Figure 5.1 Store separation surface meshes

Figure 5.2 shows the hole maps that YOGA constructed to approximate both bodies. Recall that hole maps provide a very efficient way to check if a cell might intersect a solid body.

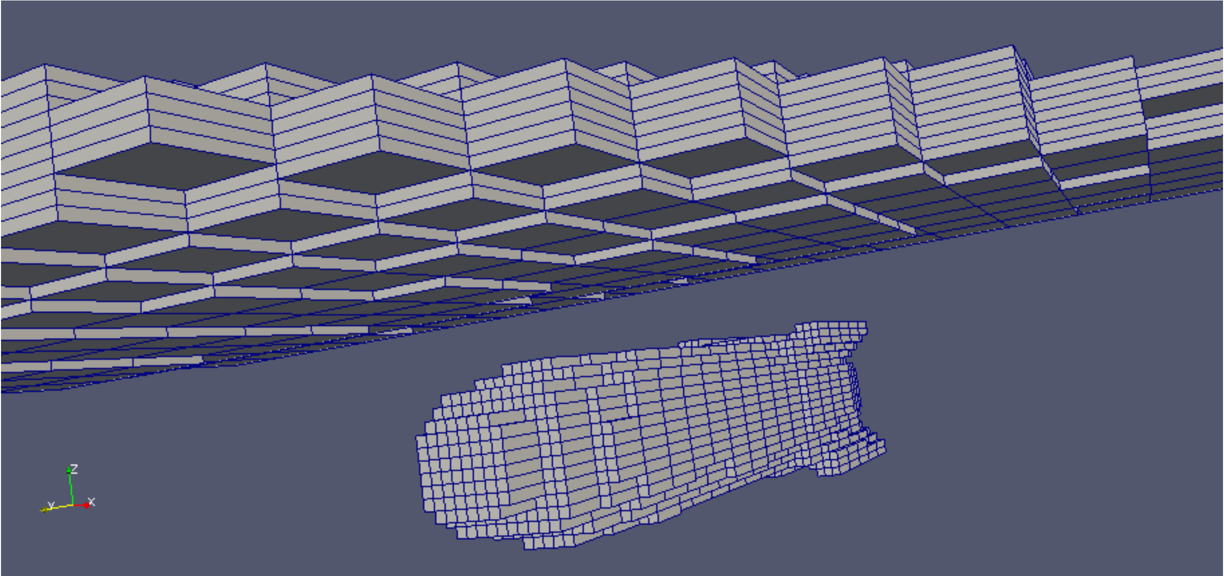


Figure 5.2   Hole map for store separation

Figure 5.3 shows a slice of the wing mesh with the cells colored by status. This figure showcases the fact that YOGA uses distance to the wall as its criterion for determining interpolation boundaries. The wing mesh is only responsible for computing the flow solution in the blue region, so it can ignore the nodes and cells in the other regions. The strip of red cells signifies the boundary between the region active and non-active nodes. If any vertices of a cell are receptors, the cell is marked red.

Figure 5.4 shows the statuses of the cells in the store mesh. Note that the shape and location of the interpolation boundary matches that in Figure 5.3, but the active and non-active regions are reversed. This is exactly the intended behavior of an Overset domain assembler that is based on

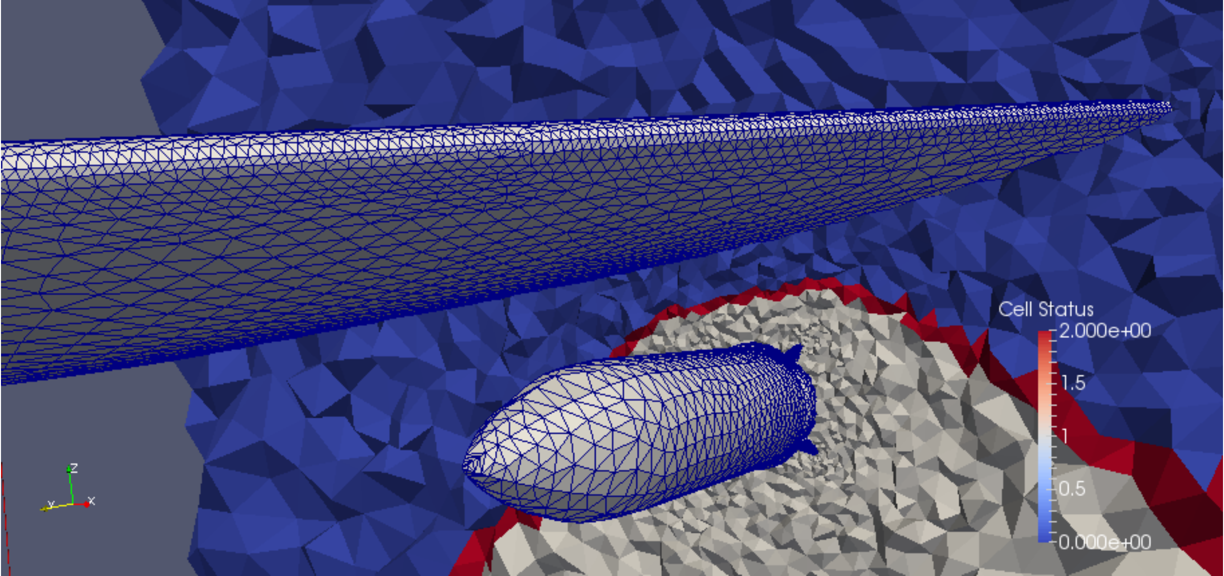


Figure 5.3 Wing mesh slice

distance to the wall. As noted by Nakahashi [24], using distance to the wall as a cutting criterion guarantees at least one node of overlap at intergrid boundaries.

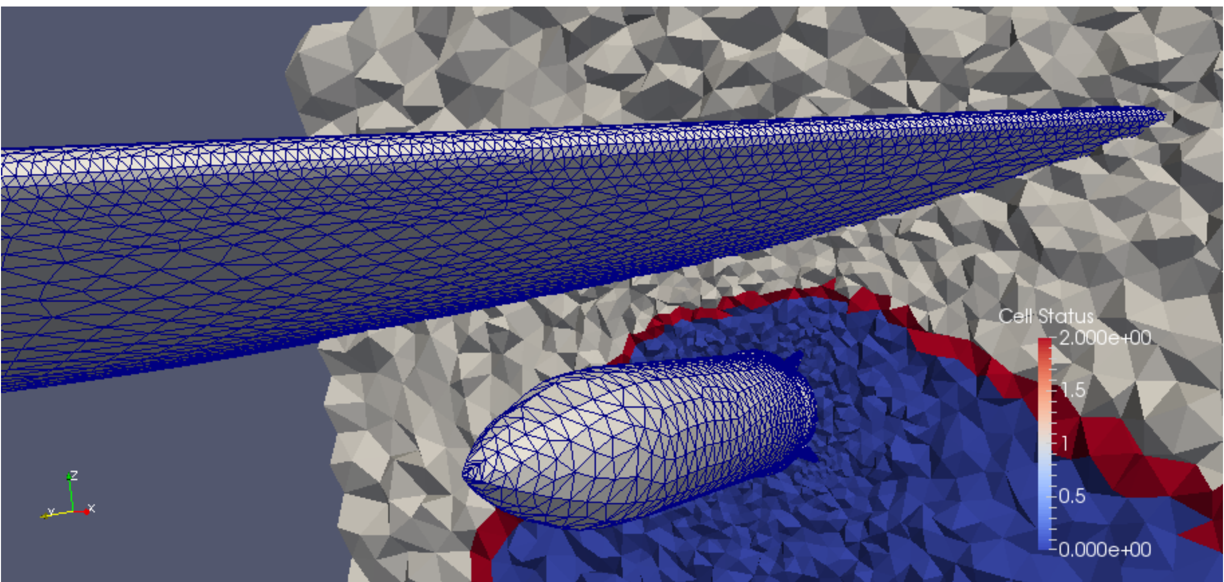


Figure 5.4 Store mesh slice

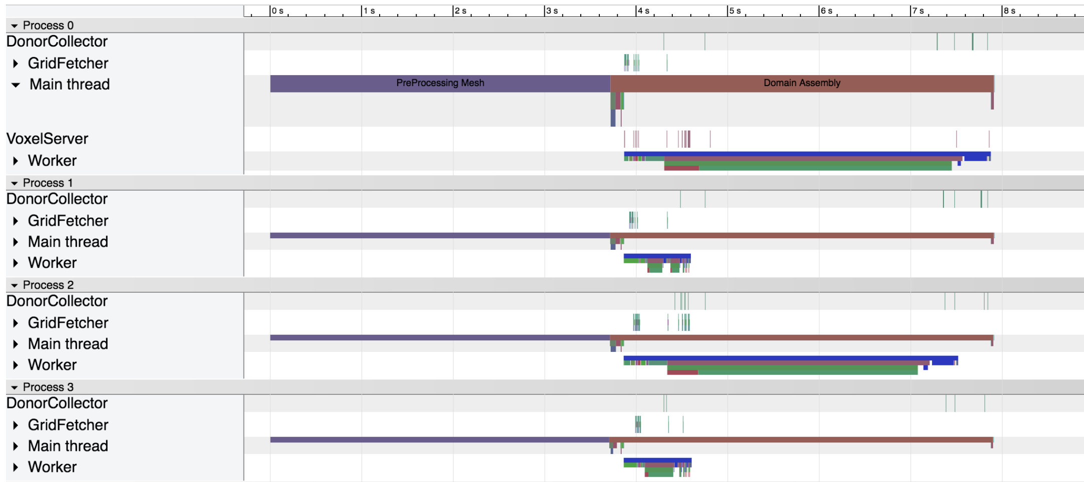
Figure 5.5 shows a visualization of timing data from two different runs of this case. YOGA initially used a static value of 50,000 as the target number of nodes to be contained by each voxel. While this strategy works well for large cases with plenty of work for each process, it can lead to large imbalances for small cases like this one. Figure 5.5a demonstrates imbalance caused by static target work voxel size. YOGA now calculates the target number of nodes per voxel T by:

$$T = \frac{N}{2P} \tag{5.1}$$

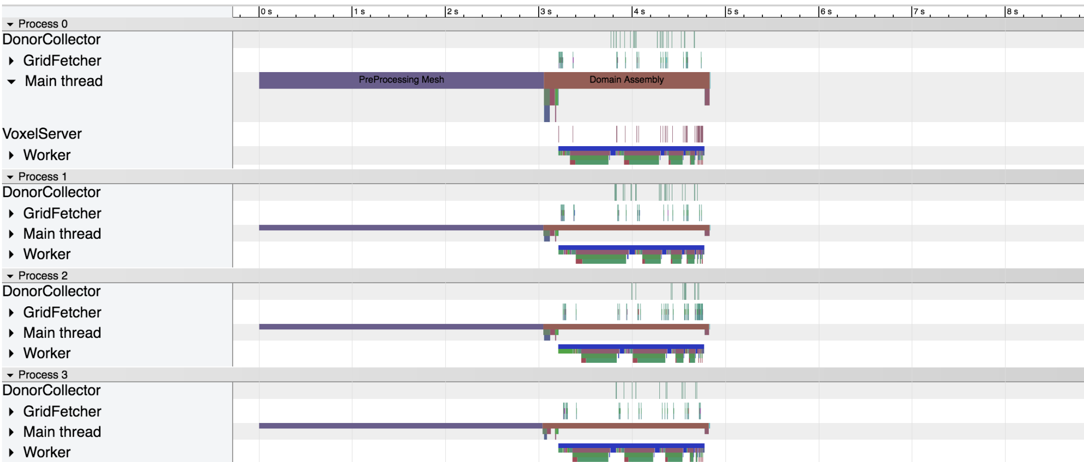
where N is the number of nodes in the global mesh system and P is the number of partitions (i.e., processes). Figure 5.5b shows the effect of calculating target voxel size dynamically. In this case, the total wall clock time of the Overset domain assembly was reduced by more than a factor of 2:

Table 5.1 Effect of dynamic work unit sizing

	time (s)
Static	4,2
Dynamic	1.8



(a) Static target voxel size



(b) Dynamic target voxel size

Figure 5.5 Improved load balancing

5.2 Hart II Rotor Case

Overset grid systems are often used in rotorcraft simulations, so it seems appropriate to include such a test case. This particular case features a small 4 million node grid system with 5 component meshes: one for each of the blades, and one for the fuselage which also acts as the background grid. Figure 5.6 shows the configuration of the fuselage and blades.

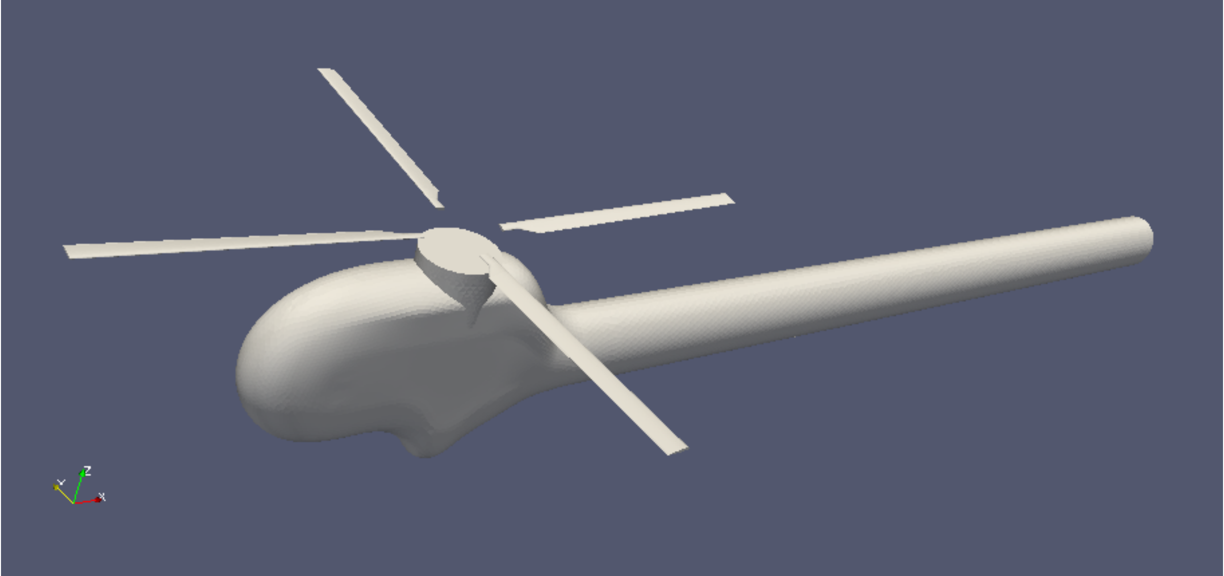


Figure 5.6 Hart II geometry

Figure 5.7 shows a slice of the fuselage mesh with the cells colored by region. It is clear that the fuselage mesh is dominated by the blade meshes in the vicinity of the blades. The interpolation boundaries caused by the other two blades are similar, as shown in Figure 5.8. Finally, Figure 5.9 contains a zoomed in view of the region near the hub. The interpolation boundary exists midway between the blade roots and the hub as expected.

Table 5.2 shows how the total wall clock time changes for Overset domain assembly for this case with respect to the number of cores. These cases were run on a machine with Intel Xeon X5660 processors. One of the design goals of YOGA is to perform well when mesh systems have approximately 50k nodes per partition (and one partition per processor core). Note that strong scaling is calculated as:

$$t_1/(N * t_N) * 100 \tag{5.2}$$

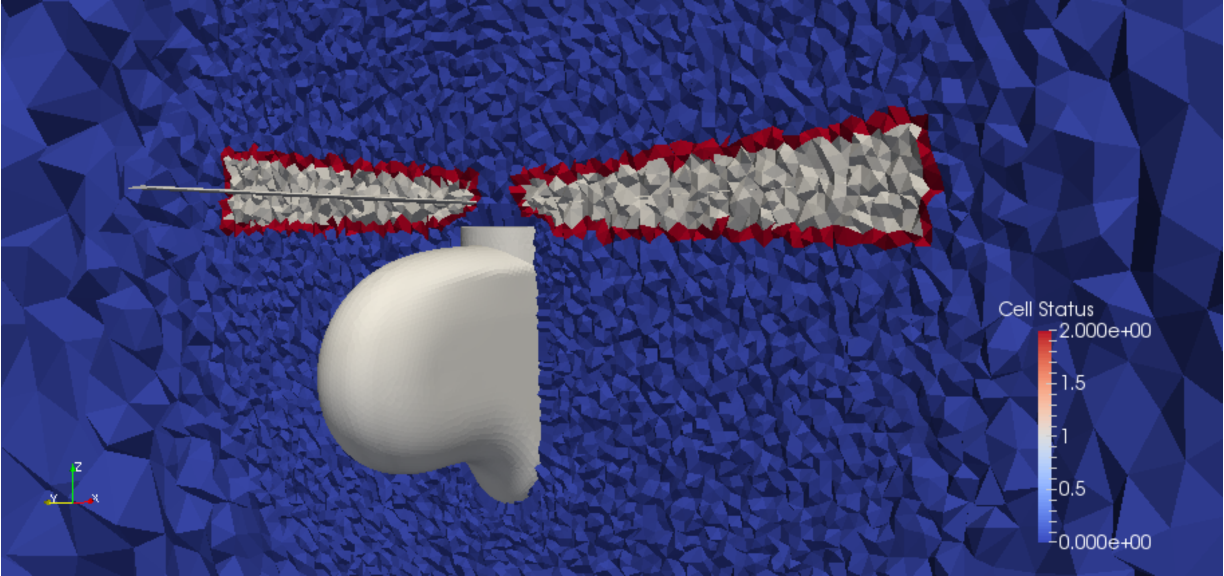


Figure 5.7   Fuselage slice 1

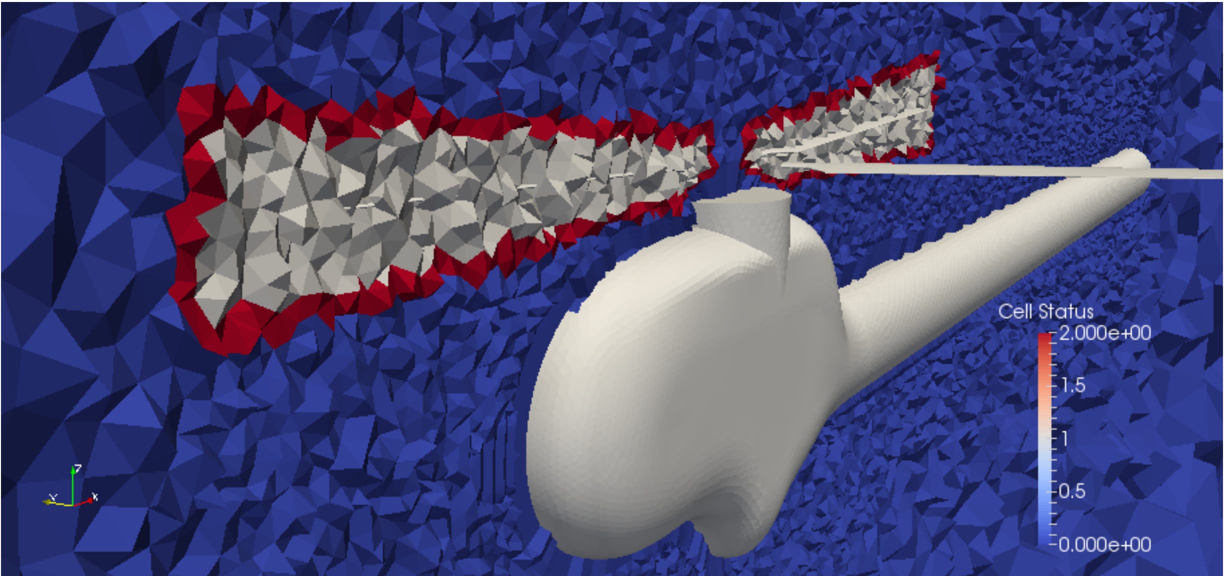


Figure 5.8   Fuselage slice 2

where t_1 is the wall clock time cost of performing the work on a single core (extrapolated from the 12 core case), N is the number of cores, and t_N is the cost of performing the same work on N cores.

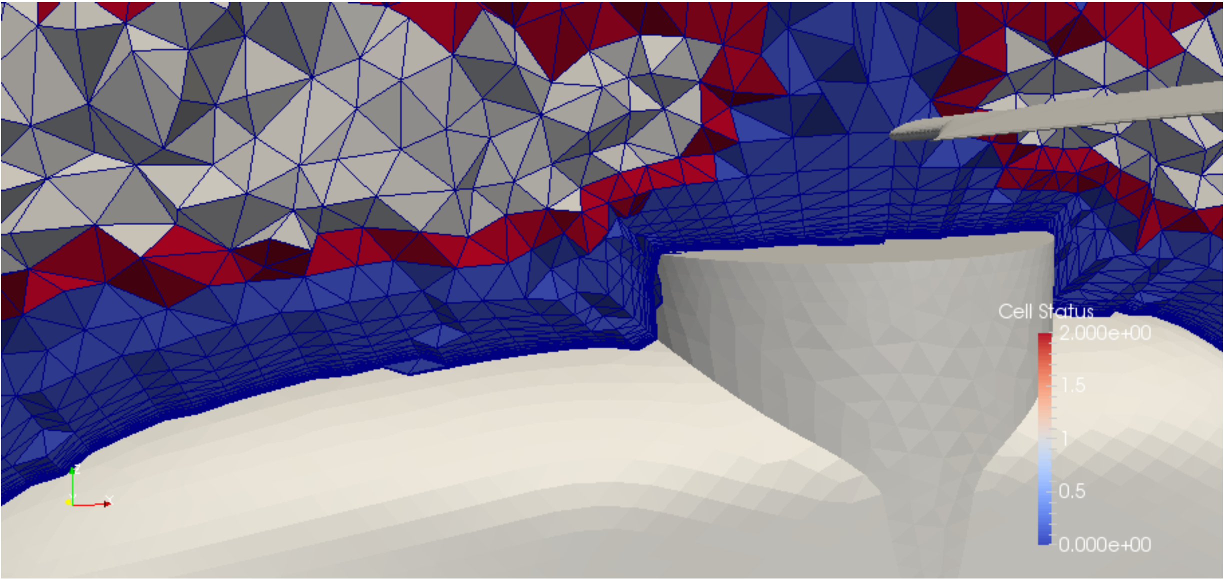


Figure 5.9 Slice near hub

Table 5.2 Timing results

cores	Total time (s)	Worker time (s)	Worker percentage	Nodes per core (thousands)	Strong scaling
12	39.3	35.3	90	312	100
24	19.7	17.6	89	156	100
36	13.5	11.7	86	104	97
48	10.4	9.0	86	781	94
60	8.6	7.4	86	62	91
72	6.8	5.6	82	51	96
84	7.1	6.0	85	45	79
96	5.6	4.5	80	39	88
108	5.0	4.1	82	35	87
120	4.5	3.4	76	31	87
132	4.9	3.7	76	28	73
144	4.3	3.4	79	26	76

5.3 Distributed Electric Propulsion Case

The first two test cases chosen for this work are both small grids with simplified geometry, so a third test case is necessary explore how YOGA behaves in a realistic production environment. The geometry for this case is a configuration of three propellers and their fairings, shown below in Figure 5.10. The five-blade propeller was designed specifically for distributed electric propulsion applications by Joby Aviation [40]. This case presents challenges in terms of both size and difficult geometry. The mesh system is much larger than the previous cases and contains approximately 115 million nodes and 680 million elements. Additionally, a tiny gap exists between each propeller and its fairing, which makes creating a quality interpolation boundary in that region challenging.

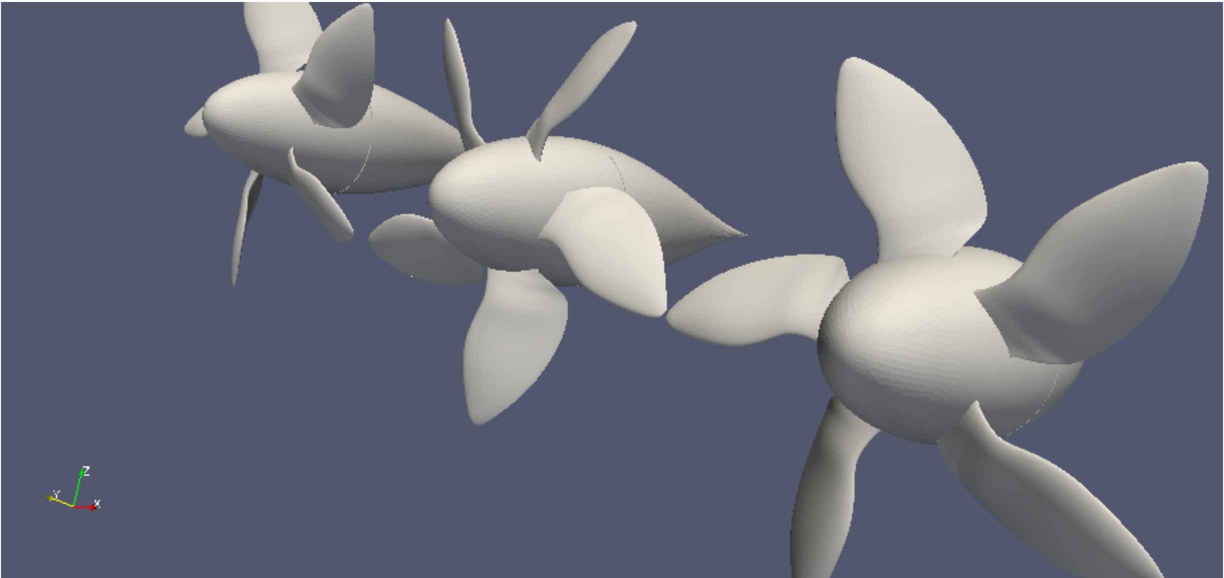


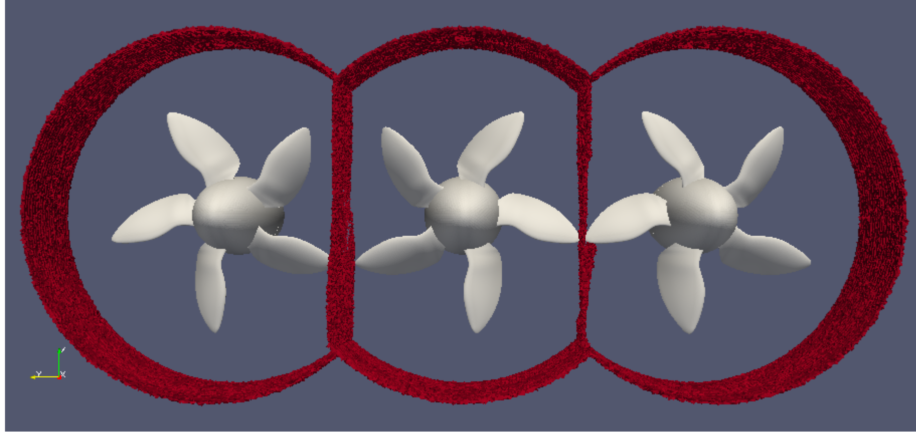
Figure 5.10 Joby Geometry

YOGA produces high quality hole cuts for this configuration. Because YOGA uses distance to the wall as its cutting criterion, the interpolation boundaries lie midway between bodies. Figure

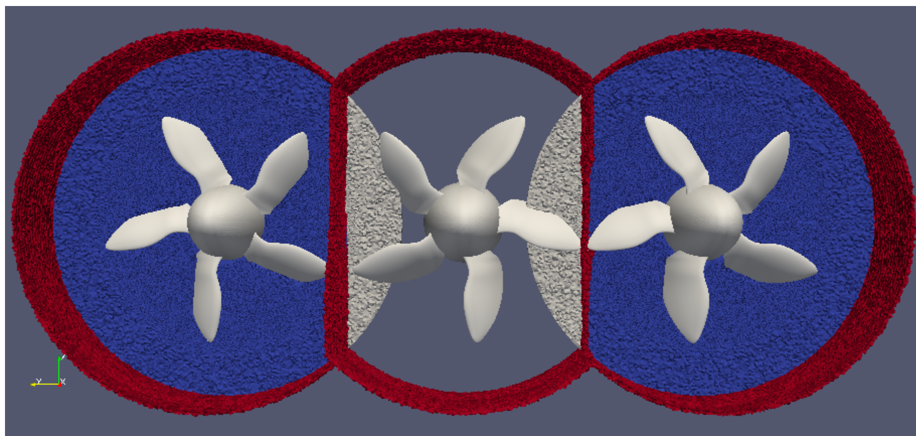
5.11a contains a series of slices of the mesh system with only the interpolation boundary cells selected. The boundaries are clean and located between bodies as expected. In Figure 5.12b a slice of the meshes for two of the propellers is added to show the solve regions (blue) for both propellers and the regions with inactive cells (white). Recall that the cells in inactive regions are inactive because the solution will be obtained on a different mesh which is closer to the geometry. Figure 5.11c then shows a zoomed in view of a region between two blades. Note that the interpolation boundary is slightly curved due to the shape of the distance field.

Figure 5.12 demonstrates that YOGA can handle small gap regions. First, Figure 5.12a shows a slice of one of the fairing grids. The fairing grid is responsible for the solution in the blue region, but is dominated by the propeller grid in the white region. The bright green square highlights a region at the bottom of the interface between the propeller and the fairing. The zoomed in view of that region in Figure 5.12b reveals that YOGA creates a high quality interpolation boundary in the small gap region between the propeller and the fairing.

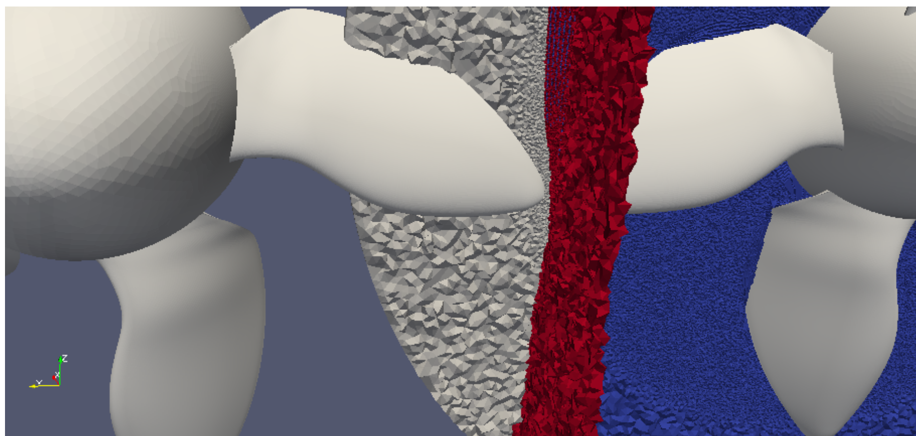
Table 5.3 demonstrates that YOGA's dynamic load balancing scheme is still effective for large scale mesh systems. The table starts at 96 cores because the size of the mesh system necessitates a large amount of memory. When the mesh system is divided into 96 partitions, each partition contains over 1 million nodes, which far exceeds the partition size of a typical CFD case. YOGA takes over ten minutes to perform Overset domain assembly on 96 cores largely because its grid servers are designed to operate efficiently on smaller more realistic partitions. While YOGA's raw performance is less than ideal, its dynamic load balancing —which is the primary focus of this work— does its job quite well. YOGA scales well on up to 1032 cores, which is the maximum job size available on NASA Langley's mid-range computer system.



(a) Interpolation boundaries

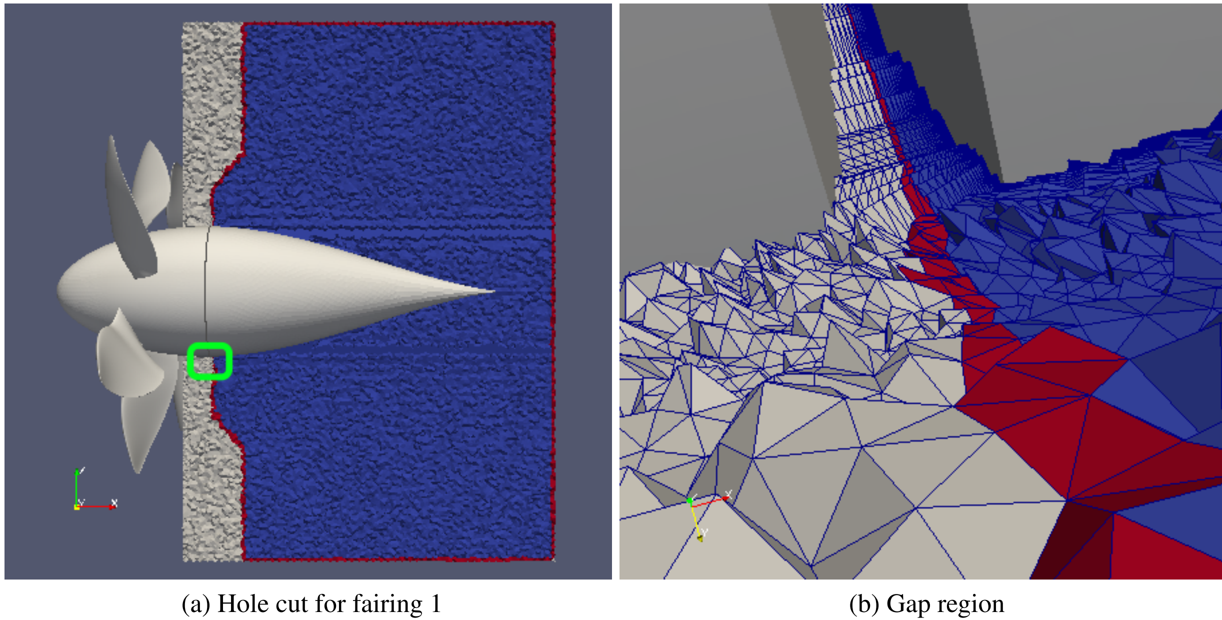


(b) Hole cuts for left and right propellers



(c) Gap between blades

Figure 5.11 Joby multi slice



(a) Hole cut for fairing 1 (b) Gap region

Figure 5.12 Fairing 1 slice

At the conclusion of this work, a limited opportunity arose to run a performance trial at NASA's Pleiades supercomputer. Table 5.4 shows the results of the trial. Performance analysis reveals several scalability concerns that arise at core counts beyond those tested during YOGA's development.

At small scales, the cost of constructing the load balancer gets cheaper as more cores are added, but the scaling eventually stalls out and the trend reverses. At 96 cores, constructing the load balancer constitutes only 7.3% of the total time, but that balloons to 21.7% at 4000 cores. The poor scaling of this phase is driven by the communication cost of estimating mesh density, which requires global communication. The initial implementation uses blocking collective communications because they were simple to implement and did not incur a large cost at smaller scales. Now that performance analysis has specifically identified mesh density estimation as a communication

bottleneck at larger scales, it can be an optimization target for future work. The load balancer is completely decoupled from the rest of YOGA, so it can be optimized in isolation.

The cost of processing work units also flatlines above 1000 cores, and starts to increase above 3000 cores. Performance data implicates two bottlenecks. First, the dynamic load balancer creates more work units than necessary based on the number of cores when the default of 50k would suffice. The adaptive ability was added to balance small cases (as discussed in section 5.1), but becomes prohibitive for large cases. This behavior can be changed with one or two lines of code. Second, Tracer output reveals that the majority of workers make requests to one particular grid server. The grid server becomes overloaded, and its response time goes up, causing many workers to block. A brief investigation of the grid server's partition immediately highlights the cause. The partition is part of the background grid, contains very large cells, and is toroidally shaped. Due to the partition's size and shape, its AABB overlaps with nearly every work unit, even though only a small number of work units actually intersect the partition itself. Now that this issue has been identified, it can be tackled in future work by reducing grid server response time for false positives.

Post processing —actually creating intergrid boundaries and selecting donors from candidates— is trivially inexpensive at small scales, so very little development effort was dedicated to making it efficient. YOGA is based on the concept that premature optimization is wasteful. Now that performance data shows a dramatic increase in cost on large cases, YOGA's post processing should be optimized. The cost is specifically driven by a single routine that resolves statuses for ghost nodes (i.e., nodes along partition boundaries). The communication uses blocking collective communication because the implementation was very simple, and performance was not at issue at the

tested scale. Txhe whole process is less than 100 lines of code and can be easily replaced with a less wasteful communication strategy in future work —now that performance data motivates the upgrade.

Table 5.3 Timing results (K cluster)

cores	Total time (s)	Worker time (s)	Worker percentage	Nodes per core (thousands)	Strong scaling
96	619.0	541.0	87	1197	100
144	425.1	371.6	87	798	97
384	134.7	103.2	77	299	115
768	70.6	47.9	68	150	110
1032	59.3	39.0	66	111	97

Table 5.4 Timing results (Pleiades)

cores	Total time (s)	Worker time (s)	Worker percentage	Build load balancer (s)	Post Processing	Nodes per core (thousands)	Strong scaling
96	471.6	405.7	86	34.2	3.6	1197	100
192	237.4	200.0	84	18.5	3.4	598	99
384	97.1	76.5	79	10.4	4.0	299	121
768	59.2	42.5	72	6.1	4.8	150	100
1536	47.3	22.3	47	10.1	7.0	75	62
1920	49.6	22.4	47	11.2	10.5	60	48
2496	47.3	22.3	47	10.1	7.0	46	38
3072	61.8	27.7	45	13.4	14.0	37	24
4000	65.0	26.4	41	14.1	19.9	29	17

CHAPTER 6

CONCLUSION

Dynamic load balancing is necessary to perform Overset domain assembly on large numbers of processors. A radically different approach to dynamic load balancing for Overset domain assembly has been presented in this work. The approach was implemented and tested in a new assembly code called YOGA (Yoga is an Overset Grid Assembler).

YOGA is capable of producing high quality interpolation boundaries both small and large grid systems, as demonstrated by the selected test cases. Additionally, YOGA can handle challenging grid systems in which bodies are in very close proximity, and very precise hole cuts are necessary.

YOGA's dynamic load balancing strategy effectively reduces wall clock time of Overset domain assembly for small and large cases. Scalability issues at large scales with thousands of cores have been identified for future work.

6.1 Technology Exploration

Additionally, this work serves as a technology demonstration. YOGA is designed to be called from a fluid solver that is running in an MPI environment, but it uses both MPI and ZeroMQ for communication. ZeroMQ is not widely used in the CFD community, but it offers several compelling features. First, ZeroMQ can easily handle communication from multiple asynchronous

threads simultaneously. This is the key feature that enabled the new dynamic load balancing approach in YOGA. Second, a ZeroMQ client and server do not have to be written in the same language. This could be an enabling technology for connecting multiple simulation codes (e.g., in multi-physics simulations) and also within CFD codes that have mixed language elements.

6.2 Future Work

6.2.1 Features

- Expose interface to C/Fortran
- Support mixed element grids
- Support cell centered codes
- Support structured grids
- Support Overset surface grids.

6.2.2 Performance

- Optimize communication in mesh density estimation
- Reduce grid server response times for false positives
- Optimize post processing communication pattern
- Augemented ADT's [41]
- Explore usage of accelerators (GPU/Phi)

REFERENCES

[1] Benek, J. A., Steger, J. L., and Dougherty, F. C., "A Flexible Grid Embedding Technique with Application to the Euler Equations," AIAA Paper 83–1944, 1983.

[2] Roget, B. and Sitaraman, J., "Robust and Efficient Overset Grid Assembly for Partitioned Unstructured Meshes," *Journal of Computational Physics*, Vol. 260, 2014, pp. 1–24.

[3] Buning, P., Chiu, I., Obayashi, S., Rizk, Y., and Steger, J., "Numerical simulation of the integrated space shuttle vehicle in ascent," *15th Atmospheric Flight Mechanics Conference*, Aug 1988.

[4] Meakin, R. L., "A New Method for Establishing Intergrid Communication Among Systems of Overset Grids," AIAA Paper 91–1586, 1991.

[5] Berger, M. and Colella, P., "Local Adaptive Mesh Refinement for Shock Hydrodynamics," *Jounral of Computational Physics*, Vol. 82, 1989, pp. 64–84.

[6] Meakin, R. L. and Suhs, N. E., "Unsteady Aerodynamic Simulation of Multiple Bodies in Relative Motion," AIAA Paper 89–1996, 1989.

[7] Suhs, N. E. and Tramel, R. W., "PEGSUS 4.0 User's Manual," Tech. Rep. AEDC-TR-91-8, Arnold Engineering Development Center, Arnold AFB, TN, Nov 1991.

[8] Barszcz, E., Weeratunga, S. K., and Meakin, R. L., "Dynamic Overset Grid Communication on Distributed Memory Parallel Processors," AIAA Paper 93–3311, 1993.

[9] Belk, D. M. and Maple, R. C., "Automated Assembly of Structured Grids for Moving Body Problems," AIAA Paper 95–1680, 1995.

[10] Chiu, I. T. and Meakin, R. L., "On Automating Domain Connectivity for Overset Grids," AIAA Paper 95–0854, 1995.

[11] Meakin, R. L., "On Adaptive Refinement and Overset Structured Grids," AIAA Paper 97–1858, 1997.

[12] Bonet, J. and Peraire, J., "An Alternating Digital Tree (ADT) Algorithm for 3D Geometric Searching and Intersection Problems," *International Journal for Numerical Methods in Engineering*, Vol. 31, No. 1, 1991, pp. 1–17.

[13] Hall, L. H. and Parthasarathy, V., "Validation of an Automated Chimera/6-DOF Methodology for Multiple Moving Body Problems," *AIAA Journal*, 1998.

[14] Prewitt, N., Belk, D., and Shyy, W., "Implementations of Parallel Grid Assembly for Moving Body Problems," AIAA Paper 98-4344, 1998.

[15] Prewitt, N. C., Belk, D. M., and Shyy, W., "Distribution of Work and Data for Parallel Grid Assembly," AIAA Paper 99-0913, 1999.

[16] Chesshire, G. and Henshaw, W. D., "Composite Overlapping Meshes for the Solution of Partial Differential Equations," *Journal of Computational Physics*, Vol. 90, No. 1, Sept. 1990, pp. 1–64.

[17] Brown, D. L., Henshaw, W. D., and Quinlan, D. J., "Overture: Object-Oriented Tools for Overset Grid Applications," AIAA Paper 99–3130, 1999.

[18] Meakin, R. L. and Wissink, A. M., "Unsteady Aerodynamic Simulation of Static and Moving Bodies Using Scalable Computers," AIAA Paper 99–3302, 1999.

[19] Sutherland, I. E., Sproull, R. F., and Schumaker, R. A., "A Characterization of Ten Hidden-Surface Algorithms," *Computing Surveys*, Vol. 6, No. 1, March 1974, pp. 1–55.

[20] Meakin, R. L., "Object X-Rays for Cutting Holes in Composite Overset Structured Grids," AIAA Paper 2001–2537, 2001.

[21] Rogers, S. E., Suhs, N. E., and Dietz, W. E., "PEGASUS 5: An Automated Preprocessor for Overset-Grid Computational Fluid Dynamics," *AIAA Journal*, Vol. 41, No. 6, June 2003, pp. 1037–1045.

[22] Chan, W. M., "Developments in Strategies and Software Tools for Overset Structured Grid Generation and Connectivity," AIAA Paper 2011–3051, 2011.

[23] Chan, W. M., Pandya, S. A., and Rogers, S. E., "Efficient Creation of Overset Grid Hole Boundaries and Effects of Their Locations on Aerodynamic Loads," AIAA Paper 2013-3074, 2013.

[24] Nakahashi, K., Togashi, F., and Sharov, D., "Intergrid-Boundary Definition Method for Overset Unstructured Grid Approach," *AIAA Journal*, Vol. 38, No. 11, Nov. 2000, pp. 2077–2084.

[25] Lee, Y. and Baeder, J. D., "High-Order Overset Method for Blade Vortex Interaction," AIAA Paper 2002–0559, 2002.

[26] Lee, Y. L. and Baeder, J. D., "Implicit Hole Cutting - A New Approach to Overset Grid Connectivity," AIAA Paper 2003–4128, 2003.

[27] Löhner, R., Sharov, D., Luo, H., and Ramamurti, R., "Overlapping Unstructured Grids," AIAA Paper 2001–0439, 2001.

[28] Noack, R. W., "Resolution Appropriate Overset Grid Assembly for Structured and Unstructured Grids," AIAA Paper 2003–4123, 2003.

[29] Noack, R. W., "DiRTlib: A Library to Add an Overset Capability to Your Flow Solver," AIAA Paper 2005-5116, 2005.

[30] Noack, R. W., "SUGGAR: A General Capability for Moving Body Overset Grid Assembly," AIAA Paper 2005–5117, 2005.

[31] Noack, R. W., "A Direct Cut Approach for Overset Hole Cutting," AIAA Paper 2007–3835, 2007.

[32] Noack, R. W., Boger, D. A., and Kunz, R. F., "SUGGAR++: An Improved General Overset Grid Assembly Capability," AIAA Paper 2009-3992, 2009.

[33] Sitaraman, J., Floros, M., Wissink, A., and Potsdam, M., "Parallel Domain Connectivity Algorithm for Unsteady Flow Computations Using Overlapping and Adaptive Grids," *Journal of Computational Physics*, Vol. 229, No. 12, June 2010, pp. 4703–4723.

[34] Sitaraman, J. and Roget, B., "Active Load Balancing for Overset Grid Assembly Procedures," 12th Symposium on Overset Composite Grids and Solution Technology.

[35] Zagaris, G. and Bodony, D., "A Collision Detection Approach to Chimera Grid Assembly for High Fidelity Simulations of Turbofan Noise," AIAA Paper 2010–836, Jan. 2010.

[36] Zagaris, G., Campbell, M., Bodony, D., Shaffer, E., and Brandyberry, M., "A Toolkit for Parallel Overset Grid Assembly Targeting Large-Scale Moving Body Aerodynamic Simulations," *Proceedings of the 19th International Meshing Roundtable*, edited by S. Shontz, Springer Berlin Heidelberg, 2010, pp. 385–401.

[37] Chan, W. M., Kim, N., and Pandya, S. A., "Advances in Domain Connectivity for Overset Grids Using the X-Rays Approach," *Proceedings of the Seventh International Conference on Computational Fluid Dynamics*, No. 1201, July 2012, pp. 1–20.

[38] Sitaraman, J. and Floros, M., "Parallel Unsteady Overset Mesh Methodology for a Multi-Solver Paradigm with Adaptive Cartesian Grids," AIAA Paper 2008–7177, 2008.

[39] Guennebaud, G., Jacob, B., et al., "Eigen v3," http://eigen.tuxfamily.org, 2010.

[40] Stoll, A. M., Bevirt, J., Moore, M. D., Fredericks, W. J., and Borer, N. K., *Drag Reduction Through Distributed Electric Propulsion*, American Institute of Aeronautics and Astronautics, 2016/06/29 2014.

[41] Feng, Y. T. and Owen, D. R. J., "An augmented spatial digital tree algorithm for contact detection in computational mechanics," *International Journal for Numerical Methods in Engineering*, Vol. 55, No. 2, 2002, pp. 159–176.

VITA

Cameron Thomas Druyor Jr completed his doctoral research in 2016 while working at NASA Langley Research Center. He graduated with a PhD in Computational Engineering from the University of Tennessee Chattanooga in August 2016.