GPU acceleration of wall distance calculation for computational fluid dynamics codes

Nathan A. Wukie*, Vasanth Ganapathy†, Chris Park‡
University of Cincinnati, Cincinnati, OH, 45221, USA

I. Introduction

Computational fluid dynamics(CFD) is a branch of simulation sciences that focuses on predicting fluid flows based on a set of governing physical equations. There are many different methods for accomplishing this that span both, different sets of governing equations, in addition to different numerical methods for computing a solution. Some different sets of equations that can be solved include the potential flow equation, streamfunction-vorticity equations, Euler equations, and Navier-Stokes equations. Analytical solutions to these sets of equations are often not available except in some extremely simple cases. Interesting geometries require a discretized approach. Some common discretization methods include Finite Difference, Finite Volume, and Finite Element methods.

This research effort is focused on a problem for a subset of those methods. In particular, we are interested in efficient wall distance calculation algorithms to support Reynolds-Averaged Navier-Stokes(RANS) solvers using a cell-centered, Finite Volume discretization with a moving mesh capability.

I.A. Background

I.A.1. Reynolds-Averaged Navier-Stokes Equations

The Reynolds-Averaged Navier-Stokes(RANS) equations govern viscous, laminar and turbulent flows. For turbulent flows, the turbulence is accounted for via a turbulence model, which usually takes the form of an additional partial differential equation or sometimes a set of partial differential equations that solve for turbulent working variables.

One general method for formulating these turbulence models is to include "production" and "destruction" terms for the turbulent working variables. In this way, phenomena that typically increase turbulence, such as vorticity, will "produce" a turbulence effect by increasing the turbulent working variable. One variable that governs such turbulence production/destruction terms is the distance of a grid cell to the nearest solid wall; simply known as the wall distance, d_{wall}.[1]

I.A.2. Cell-centered, Finite Volume discretization

The numerical method that this investigation directly supports is a cell-centered, finite volume discretization, although it could be readily extended to other methods with some modifications. The finite volume method relies on an integral form of the governing equations, and the method we are interested in stores all variables at the center of each computational cell. The values in each cell can then be updated, based on boundary fluxes computed at the interface of each computational cell.[2] The value of interest here is the d_{wall} variable required by the turbulence model.

*Graduate student researcher, School of Aerospace Systems, ML 70, Cincinnati, OH, AIAA Student Member.
†Student, University of Cincinnati
‡Student, University of Cincinnati

I.A.3. Moving mesh calculation

Some investigations that use fluid simulations are interested in moving body problems such as store separation for aircraft, and rotating machinery, among others. One way to accomplish this is to use body-local grids that overlap a background grid as shown in Figure 1.[3] In this way, the body grid can move on-top of the background mesh to facilitate body motion. This usually works by computing an iteration for the CFD code, computing the force on the body of interest, computing the distance moved, and updating coordinates of the body-local grid.

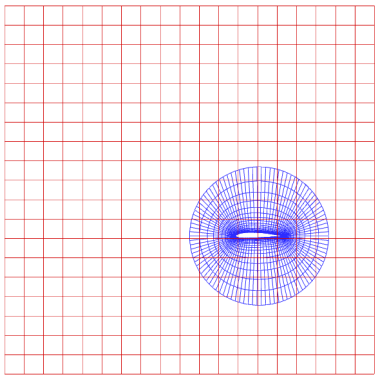

Figure 1: Chimera, overlapping grid for moving airfoil simulation

I.A.4. Motivation for efficient wall distance computation

Traditionally, the wall distance calculation would be of little importance from an efficiency perspective. In stationary grid simulations, the wall distance does not change, and so it would only be performed once as a pre-processing step, which would represent a trivial amount of time compared to the overall calculation. For moving body simulations however, the wall distance can change after each update of the body position and thus has to be recomputed for every iteration. It becomes extremely important then, to have an efficient method for recomputing the wall distance in order to maintain reasonable computation times. Using graphics processing units(GPU's) to accelerate the wall distance computation is seen as a promising method for improving computational efficiency for moving body problems.

II. Design and optimization approach

The application is laid out as a main function that consists of a grid preprocessor, and then function calls to the respective wall distance calculation routines. The preprocessor reads the computational grid and stores the cell centers as two arrays, x-coordinates and y-coordinates. It also reads in geometry faces and stores their respective x and y coordinates in two separate arrays. These are the input data to the wall distance calculation routines. There are two wall distance calculation methods that are being used for this investigation. They are a 'Brute Force' method and 'Advancing Boundary' method and will be described in the subsequent sections. A diagram of the main application can be seen in Figure 2.

II.A. Brute force method

II.A.1. Complexity

II.A.2. Parallelization

II.B. Advancing boundary method

The "Advancing Boundary" method used in this investigation is based on work done by Roget and Sitaraman, where they used voxelized marching spheres to reduce the wall distance calculation to a small subset of the original calculation.[4] The following process is largely the same as the description in the paper, with modifications to auxiliary cell structures and elimination of some of the more nuanced features.

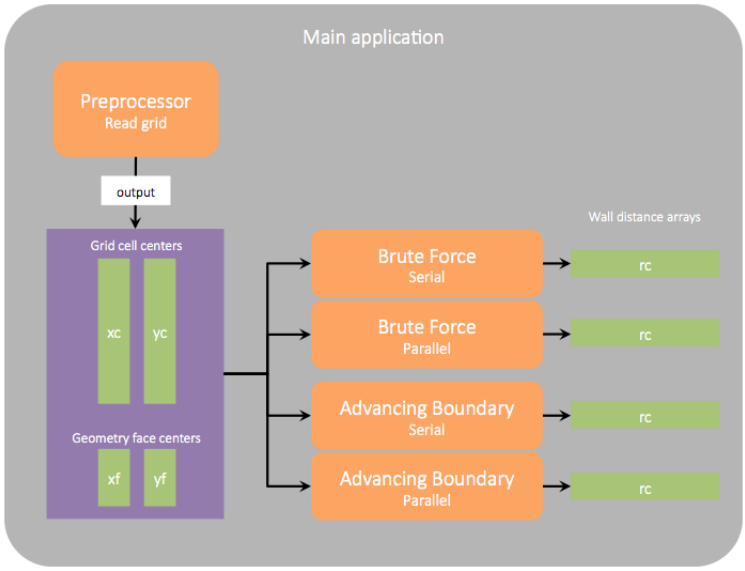

Figure 2: Main application diagram

The advancing boundary method begins with a preprocessing routine that computes a set of auxiliary cells, which are then provided to the wall distance calculation routine. This is shown as a diagram in Figure 3.

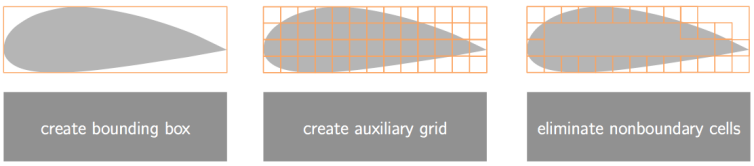

Figure 3: Advancing Boundary - Preprocessor

The goal of the auxiliary cells is that each would contain several solid geometry faces. In this way, the wall distance routine can first search through the smaller number of auxiliary cells to determine those eligible for containing the closest solid face. This allows the algorithm to reduce the number of total faces that need checked by eliminating multiple solid faces each time an auxiliary cell is eliminated. This wall distance calculation process is laid out in the diagram in Figure 4.

II.B.1. Complexity

The work complexity for this algorithm is related to the number of cells in the computational grid and the number of solid geometry faces that each cell has to search. In the worst case, each computational grid must search through each solid face to find the closest face. Therefore, for a case with n computational cells and m solid geometry faces, the work complexity of the algorithm is $O(n \times m)$. However, the computation for each cell executes independent of all the other cells and requires only one step, which is to search through all of the solid geometry faces. As a result, the step complexity of the algorithm is $O(1)$.

II.B.2. Parallelization

This algorithm, much in the same way as the brute force method, has a lot of parallelism that can be exploited for improved performance. Specifically, the wall distance calculation for each computational cell executes independently of all other computational cells. The parallel approach used for this study is a thread-per-cell technique, where one wall distance kernel thread is launched for each computational cell.

- Work: $O(n \times m)$
- Step: $O(1)$

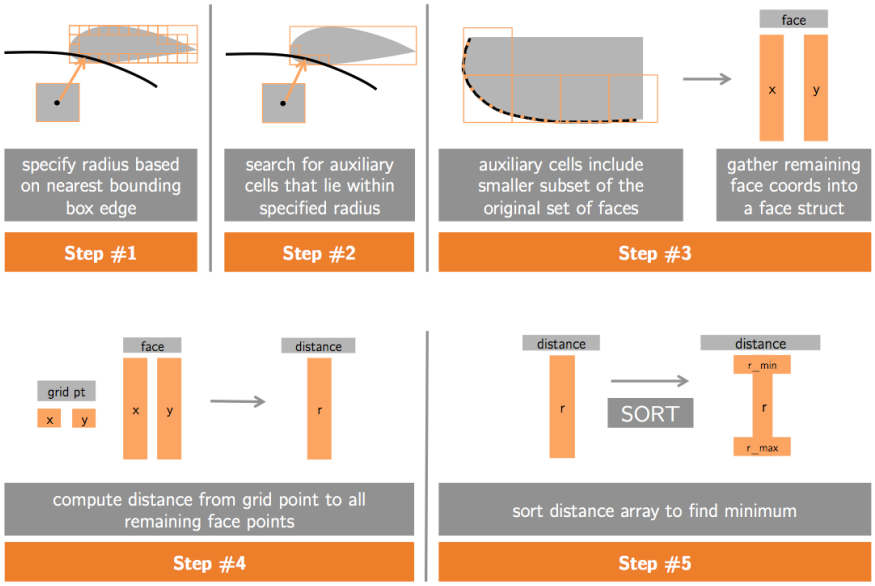

Figure 4: Advancing Boundary - Wall distance calculation

III. Application performance and results

The following sections present the methods used to benchmark each algorithm, along with the various modifications implemented to improve performance. Finally, timings for both serial and parallel implementations are presented for both methods to evaluate the overall speedup achieved via the parallel GPU implementation.

III..3. Problem set

The problem set chosen for this investigation was a solid circular body, surrounded by a circular discretized domain. This simple geometry allowed a computational grid to be easily refined in order to load algorithms more heavily and reduce the effect of computational overhead on the overall timings. The grid and wall distance field can be seen in Figure 5.

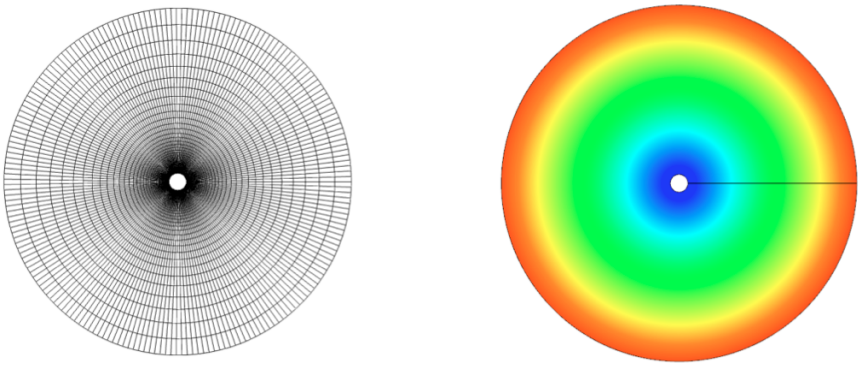

Figure 5: Computational grid and corresponding wall distance field

III..4. Brute-force algorithm

III..5. Advancing boundary algorithm

Several implementations of the advancing boundary algorithm were tested, both for serial and parallel algorithms. The different methods for improving performance included memory management and changing storage structures. The initial serial algorithm was implemented with an 'Auxiliary Cell' struct that would

include the geometric information for a given auxiliary cell, in addition to providing some method for accessing the faces that were contained in the face. The general layout of this struct was similar for all iterations of the serial and parallel codes. Only the face storage mechanism was altered for more efficient memory accesses. The first iteration of these stored the faces as a linked-list of face structs, each of which stored the x,y pair associated with a face center. This initial configuration is shown in Figure 6.

```
struct cell{

    // Cell Boundaries
    double xmin, xmax;
    double ymin, ymax;
    double xcenter, ycenter;

    // Included faces linked list
    struct face * root;

};
```

Figure 6: Initial auxiliary cell struct

Other methods for storing the faces included pointers to face arrays, explicitly allocated arrays in the cell struct, and also an array of face indices that would facilitate access to the correct face in an external array.

```
struct cell_t2{

    // Cell Boundaries
    double xmin, xmax;
    double ymin, ymax;
    double xcenter, ycenter;

    // Included faces linked list
    int faceNum;

    int storage;
    double * xface;
    double * yface;
```

(a) Struct A: Pointers to storage arrays

```
struct cell_pt3{

    // Cell Boundaries
    double xmin, xmax;
    double ymin, ymax;
    double xcenter, ycenter;

    // Included faces linked list
    int numFaces;

    double faces[200];

};
```

(b) Struct B: Explicitly allocated in-cell storage

```
struct cell_t3{

    // Cell Boundaries
    double xmin, xmax;
    double ymin, ymax;
    double xcenter, ycenter;

    // Included faces linked list
    int numFaces;

    double face_x[100];
    double face_y[100];
```

(c) Struct C: Separate in-cell storage

```
struct cell_pt1{

    // Cell Boundaries
    double xmin, xmax;
    double ymin, ymax;
    double xcenter, ycenter;

    // Included faces linked list
    int numFaces;

    int storage;
    int faceIndex[100];
```

(d) Struct D: Face index array

Figure 7: Auxiliary cell struct versions

The timings for the Serial algorithm are shown in Table 1. The linked-list method of storing faces was the most efficient access method of those tested. The best-performing serial algorithm was then compared to the parallel algorithms to gauge the parallel speedup factor. The parallel timings are shown in Table 2. The initial implementation used all global memory accesses and subsequent improvements focused on bringing in reusable data into shared memory or local registers. Some final performance improvements were achieved by restructuring the face storage structure.

IV. Schedule and division of work

Figure 8 shows the work split between the team contributors.

Iteration	Description	Average Time (ms)
T1	Baseline, linked-list of faces	8796
T2	New aux cell struct	11044
T3	New aux cell struct	10007

Table 1: Serial "Advancing Boundary" algorithm development and performance

Iteration	Description	Average Time (ms)	Speedup vs. Serial
T1	Baseline, no shared memory	134	65
T2	Memory management	114	77
T3	Memory management	114	77
T4	Memory management	112	78
T5	Memory management	113	78
T4 - new cell - 1	New aux cell struct	106	83
T4 - new cell - 2	New aux cell struct	102	86

Table 2: Parallel "Advancing Boundary" algorithm development and performance

IV.A. Self-assessment: Nathan Wukie

I provided the primary topic and guidance for the project, along with the application framework, preprocessor, and cmake usage. Additionally, I was responsible for the "Advancing Boundary" algorithm development, implementation, and parallelization.

IV.B. Self-assessment: Vasanth Ganapathy

References

[1]Allmaras, S. R., Johnson, F. T., and Spalart, P. R., "Modifications and Clarifications for the Implementation of the Spalart-Allmaras Turbulence Model," Iccfd7-1902, 2012.

[2]Hirsch, C., *Numerical Computation of Internal and External Flows: Volume 2*, John Wiley & Sons Ltd., 1994.

[3]Galbraith, M. C., *A discontinuous Galerkin Chimera overset solver*, Ph.D. thesis, The University of Cincinnati, 2013.

[4]Roget, B. and Sitaraman, J., "Wall distance search algorithm using voxelized marching spheres," Iccfd7-1204, 2012.

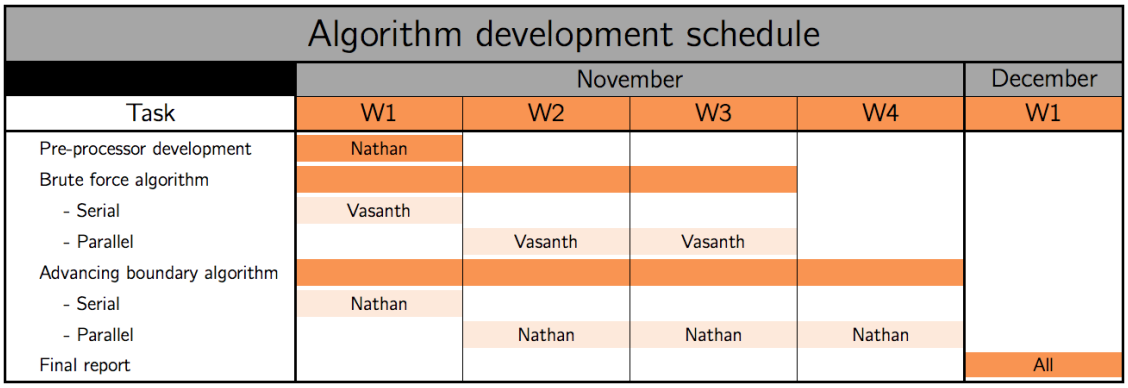

Figure 8: Development schedule and division of work