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.

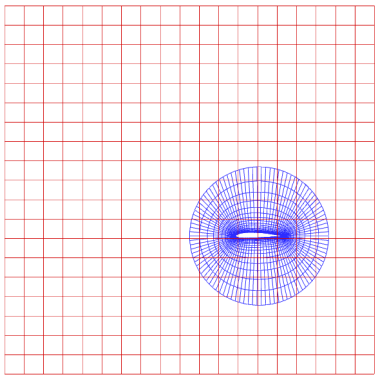


Figure 1: Chimera, overlapping grid for moving airfoil simulation

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

The brute force method for calculating the wall distance for a cell element is shown in Figure 3. For each cell element, its distance from each of the solid faces is calculated (2D distance is shown and used for this project). Wall distance for the cell is the minimum of these distances.

The serial implementation for the brute force method is shown in Figure 4. For each cell element, the distance from the cell to a large number of faces is calculated, and the wall distance is updated if this face has the least distance.

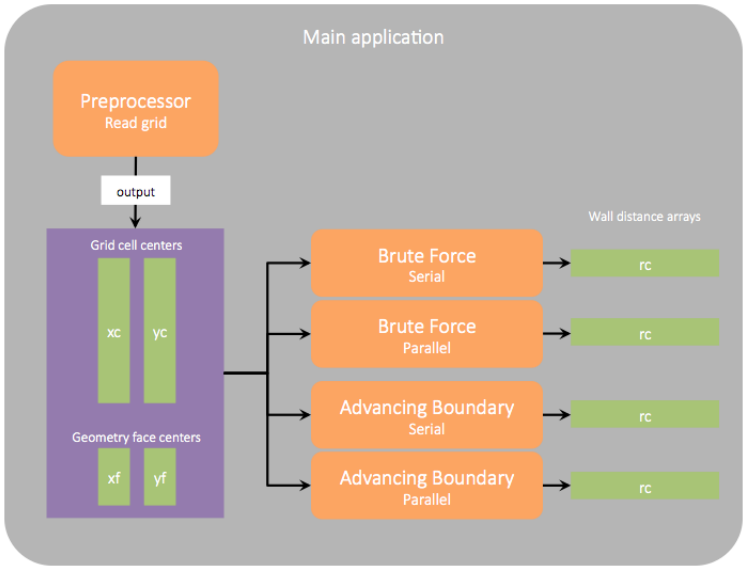


Figure 2: Main application diagram

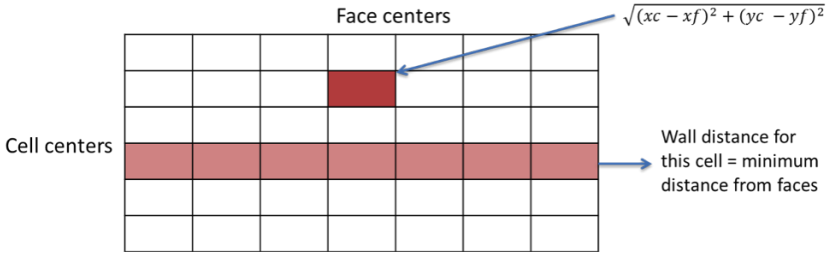


Figure 3: Brute force method

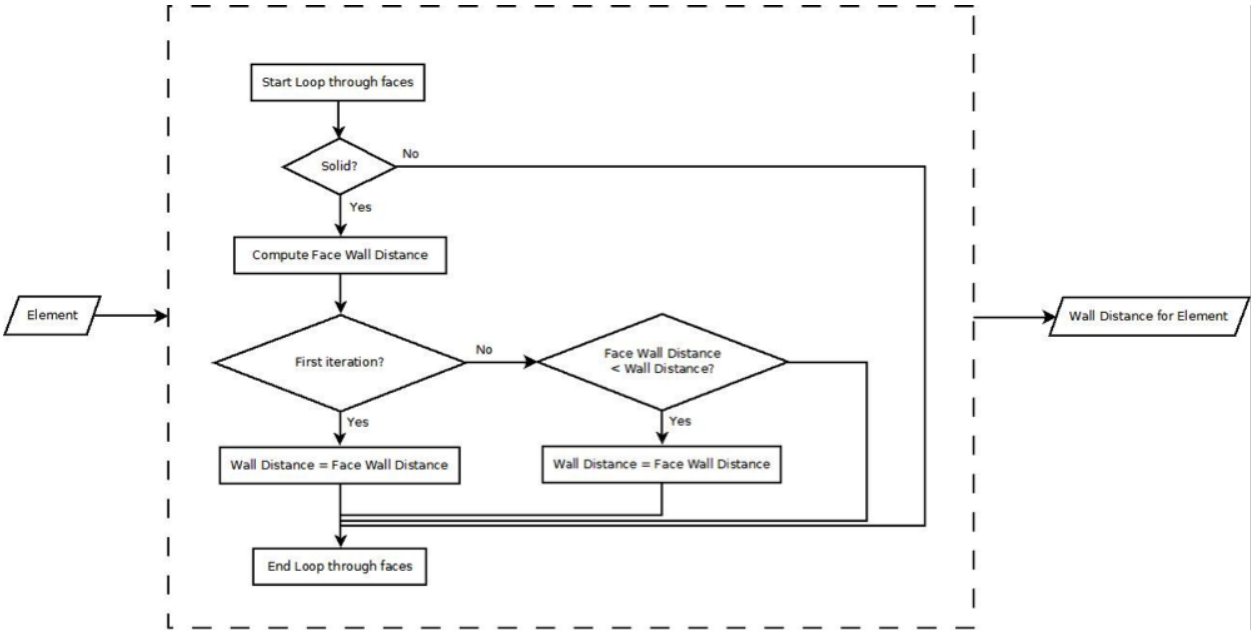


Figure 4: Brute force serial implementation

II.A.1. Complexity

The work and step complexities for the serial approach with n cell elements and m solid faces can be approximated as:

- Work: O($n \times m$) - for each cell element, each solid face will have to be evaluated. So, total number of operations is in the order of $n \times m$
- Step: O(m) - the cell elements can be evaluated in parallel, but the distance from each solid face to the element is evaluated in sequence. So, the longest chain of sequential dependencies is in the order of m.

II.A.2. Parallelization

The opportunity for parallelism arises in being able to perform the distance calculations and comparisons for each element and/or for each face separately. Two parallel approaches were implemented for the brute force method.

1. Block-per-cell (Figure 5): Each block (CUDA block) calculates the wall distance for a cell. Each thread within a block calculates distance from a face; after all the threads have calculated the distance, a reduce minimum operation is performed to get the cell wall distance.
2. Thread-per-cell (Figure 6): Each thread calculates wall distance for a cell. Each thread calculates distance from the cell to each of the faces, and keeps a running minimum distance value as each faces distance gets calculated.

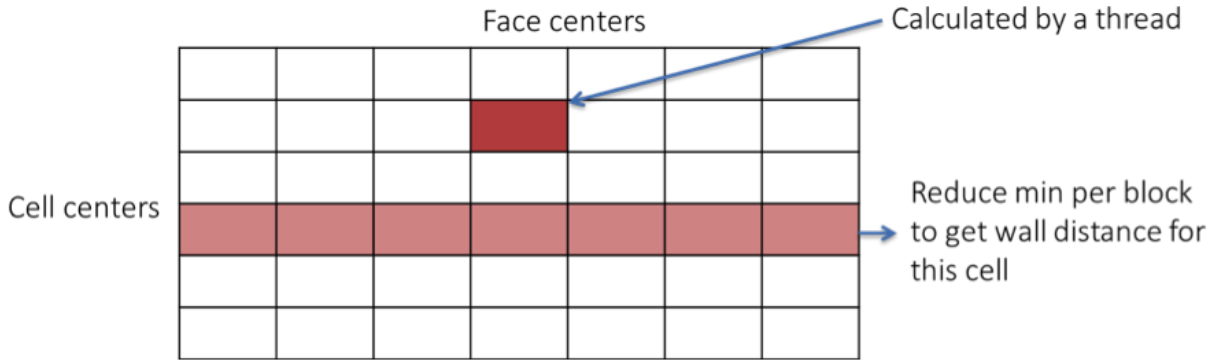


Figure 5: Brute force Block-per-cell implementation

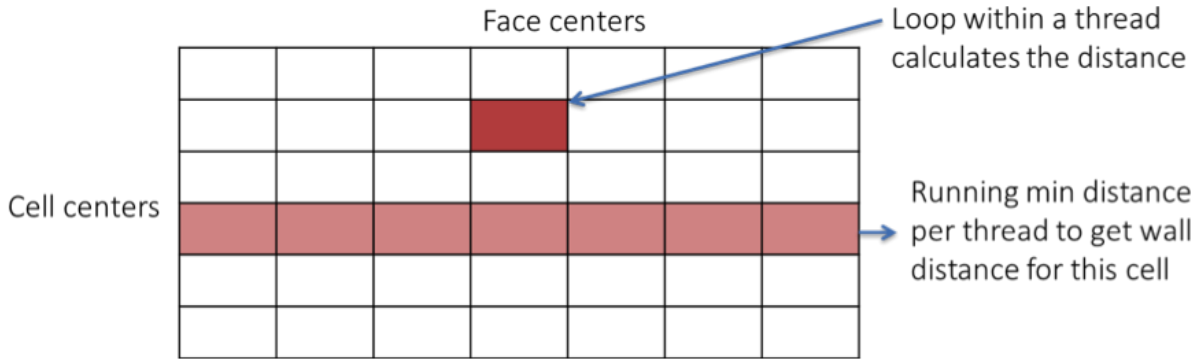


Figure 6: Brute force Thread-per-cell implementation

The work and step complexities for the parallel approaches with n cell elements and m solid faces can be approximated as:

- Work: $O(n \times m)$ work complexity for the two parallel implementations is similar to the serial implementation, as for each cell element, distance from each solid face will have to be evaluated. So, total number of operations is in the order of $n \times m$.
- Step: $O(log_2 m)$ for Block-per-cell, and $O(m)$ for Thread-per-cell In the Block-per-cell implementation, the wall distances for each face can be calculated in parallel in one step, and the reduce operation to compute the minimum wall distance has a step complexity of $log_2 m$. Therefore, step complexity for this parallel implementation is in order of $log_2 m$. In the Thread-per-cell implementation, each of the cell elements can be evaluated in parallel in a separate thread, but the distance from each solid face to the element is evaluated in sequence inside each thread. So, the longest chain of sequential dependencies is in the order of m.

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.

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 7.

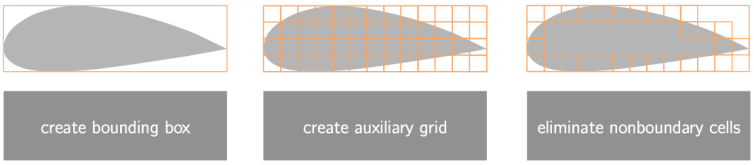


Figure 7: Advancing Boundary - Preprocessor

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 8.

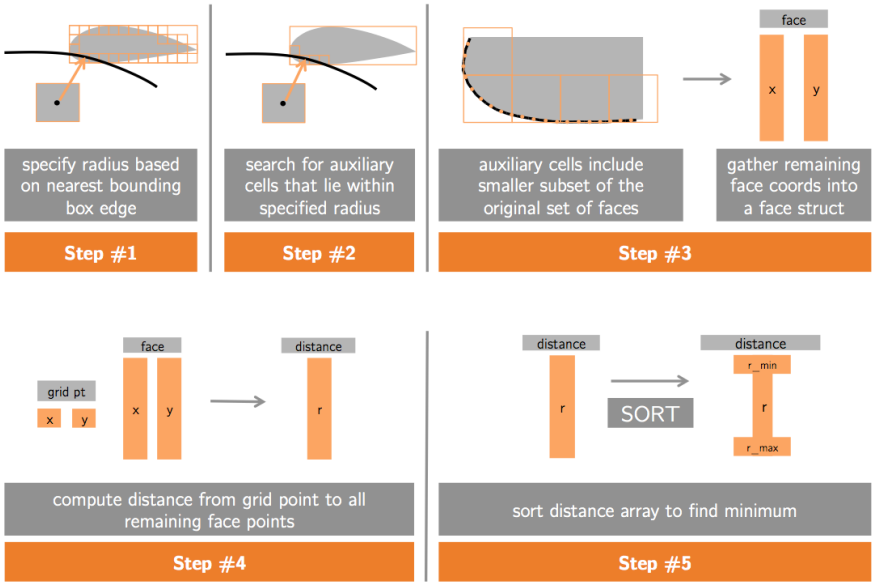


Figure 8: Advancing Boundary - Wall distance calculation

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, at worst, as many steps as there are faces, which is to search through all of the solid geometry faces. As a result, the step complexity of the algorithm is $O(m)$.

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: $O(m)$

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 9.

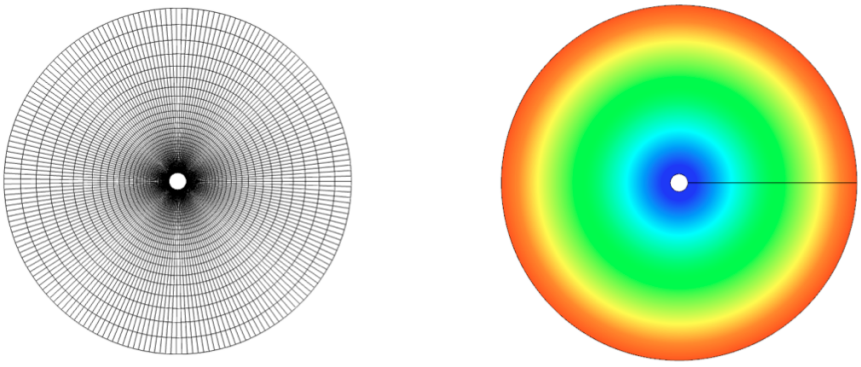


Figure 9: Computational grid and corresponding wall distance field

III..4. Brute-force algorithm

Several implementations of the two brute force parallel methods were tested in order to evaluate efficient memory usage such as coalesced memory access, use of available shared memory and thread local memory.

Three variations of the Block-per-cell method were implemented:

1. 1A: Shared memory writes for face distances since a reduce operation per block is performed to get the minimum distance from an array of distances calculated by the threads, the distance array is stored in shared memory.

2. 1B: (1A) + Shared memory reads to face (x, y) arrays storing the face (x, y) arrays in shared memory, so that every thread does not have to read all of the face (x, y) arrays from global memory.

3. 1C: Same as (1B) with (x, y) arrays converted to an interspersed array for coalesced shared memory reads, i.e. $x0, y0, x1, y1, ...$

Three variations of the Thread-per-cell method were implemented:

1. 2A: Each thread calculates wall distance for a cell.

2. 2B: Shared memory reads to face (x, y) arrays.

3. 2C: Same as (2A) with (x, y) arrays converted to an interspersed array for coalesced global memory reads, i.e. $x0, y0, x1, y1, ...$

The performance of the different brute force parallel methods is shown in Table 1. The times shown are based on several sample runs of the algorithm on the Ohio Supercomputer Center (OSC) Oakley cluster.

Table 1: Brute force algorithm performance

Algorithm	SM Grid(ms)	MD Grid(ms)	FN Grid(ms)	Speedup(FN vs Ser.)
Serial	33	595	13636	Not applicable
Block-per-cell(1A): Shared memory writes	1	28	-	-
Block-per-cell(1B): Shared memory reads	1	30	-	-
Block-per-cell(1C): Coalesced shared memory reads	1	31	-	-
Thread-per-cell(2A)	< 1	7	173	79
Thread-per-cell(2B): Shared memory reads	1	9	293	46
Thread-per-cell(2C): Coalesced global memory reads	< 1	7	166	82

The Block-per-cell methods do not support grid sizes with a large number of faces, as these methods require the number of faces to be less than the maximum number of threads available per block (1024 for the OSC cluster). Therefore, performance could not be measured for the extra fine grid. Also, the overhead of setting up shared memory and coalesced reads outweighs the performance gains for the small/fine grid sizes (1B and 1C). Therefore, performance for all three Block-per-cell methods was similar for the small/fine grid sizes.

The Thread-per-cell methods performed better than the Block-per-cell methods. Also, the Thread-per-cell methods do not have limitations on the grid size, and can accommodate any grid size. Thread-per-cell is not a good algorithm for utilizing shared memory, since each thread has to calculate the wall distance for a cell. However, this method (2B) was implemented in order to access if there is a performance gain from one thread in a block reading the face values and storing them in shared memory, instead of all threads reading them from global memory. As expected, this caused a negative impact on performance due to thread divergence, as one thread had to perform significantly large amount of work compared to all the other threads in the block. The coalesced global memory reads made a slight improvement in performance, since reading the blocks in a coalesced manner reduces the memory access time. Therefore, the Thread-per-cell method with coalesced global memory reads (2C) was the optimal brute force algorithm implemented in this project, with a speed-up of 82 compared to the serial method.

For future improvements, a combination of the Thread-per-cell and Block-per-cell methods can be used in order to achieve the best possible performance (as close to $O(log_2 m)$ as possible). This can be done by separating out the two main calculations into two steps: calculating the face distances from the cell centers

in one kernel, with each thread calculating a distance (i.e. thread-per-cell face distance); and then assigning blocks so that each block performs a reduce minimum on the face distances related to a cell (i.e. block-per-cell reduce). This will require a block/thread sizing algorithm that can handle the different grid sizes. Another future improvement that should be explored is to break up the grid calculations into tiles, as that helps with efficient memory reads.

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 10.

```
struct cell{

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

    // Included faces linked list
    struct face * root;

};
```

Figure 10: 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 11: Auxiliary cell struct versions

The timings for the Serial algorithm are shown in Table 2. 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 3. 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.

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 2: 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 3: Parallel "Advancing Boundary" algorithm development and performance

Future improvements to the Advancing Boundary wall distance calculation algorithm could include some alternate methods for memory management in order to improve memory access efficiency. It would be interesting to explore other storage structures for auxiliary cell data and other methods for organizing this information that may improve the overall efficiency of the algorithm.

IV. Schedule and division of work

Figure 12 shows the work split between the team contributors.

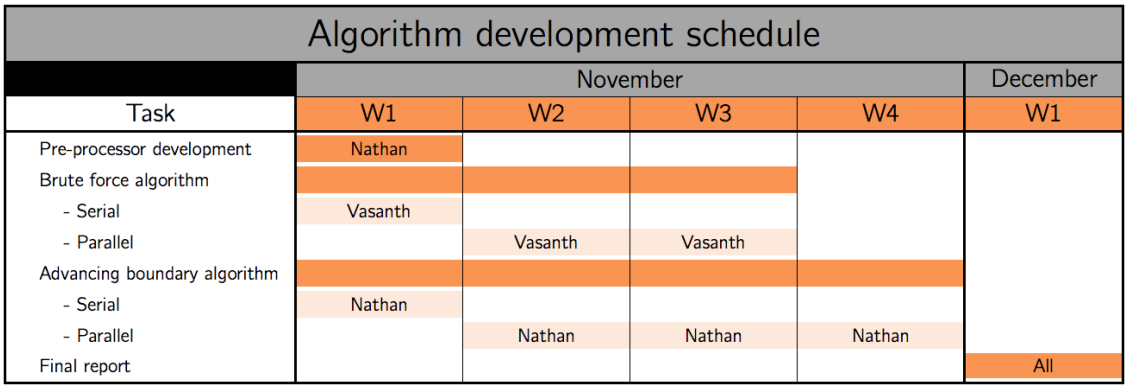


Figure 12: Development schedule and division of work

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

I was responsible for the Brute Force algorithm development, implementation, and testing. This includes the serial and parallel methods for the Brute Force algorithm.

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.

Appendices

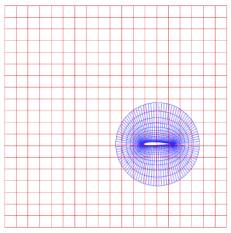
A. Source code

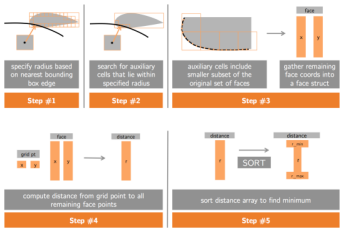
The source code and all supporting documents used for this project can be accessed through the following git repository:

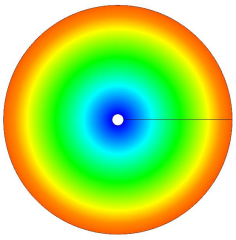
https://github.com/Basian/PC-WallDistanceProject.git

B. Presentation slides

GPU acceleration of wall distance calculation for computational fluid dynamics codes







Nathan Wukie Vasanth Ganapathy Chris Park

Outline

- Background
- Brute-force algorithm
- Advancing boundary algorithm

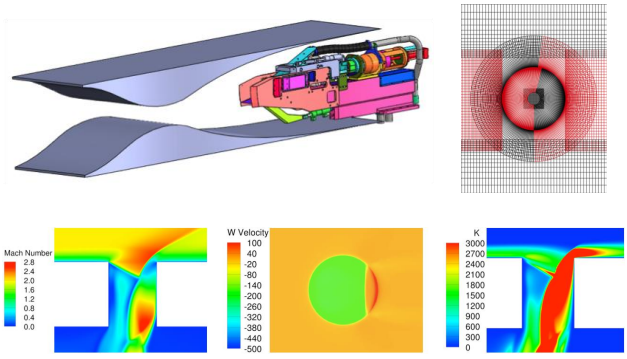
Background

- Reynolds-Averaged Navier-Stokes calculation

Conservation of mass

Conservation of momentum

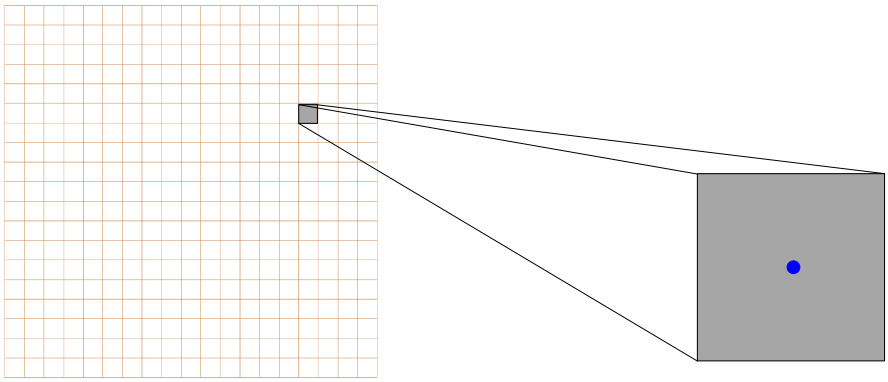
Conservation of energy



Wukie et al. 2012

Background

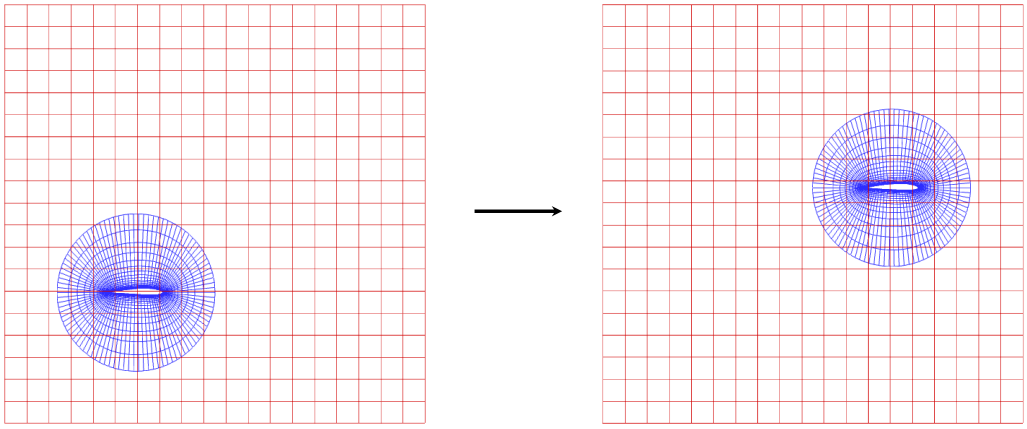
- Cell-centered, Finite Volume discretization



Solution variables stored at cell-center

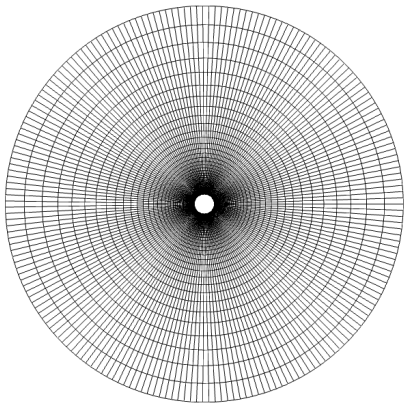
Background

- Moving mesh calculation

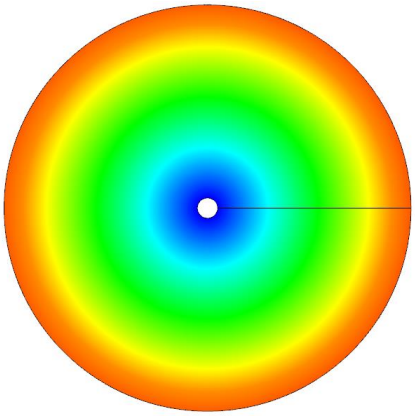


Background

- Test case: circle



Computational grid

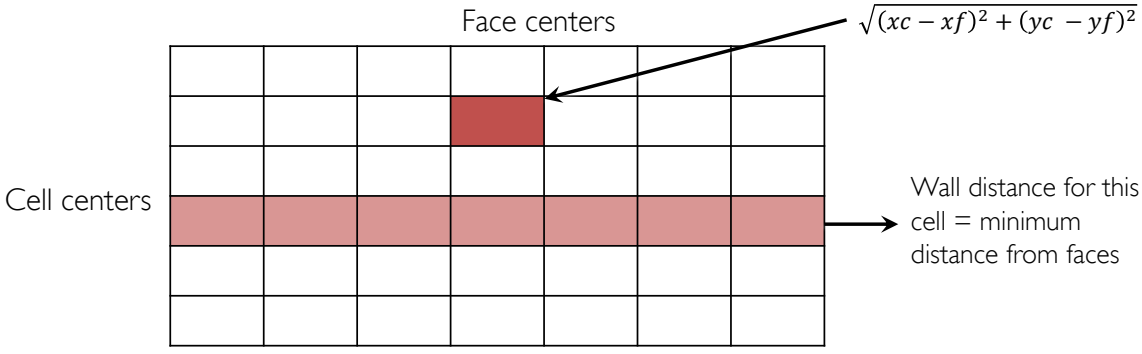


Wall distance field

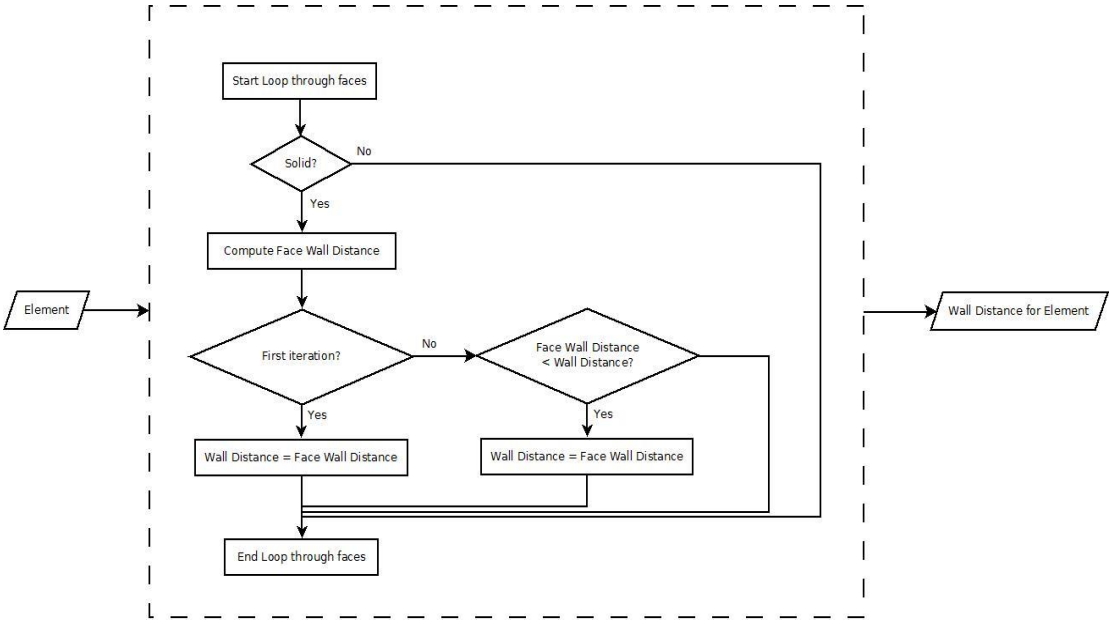
Brute-force algorithm

Brute-force Algorithm Outline

- For each cell element
  - Calculate its distance from each of the solid faces
  - Wall distance is the minimum of these distances

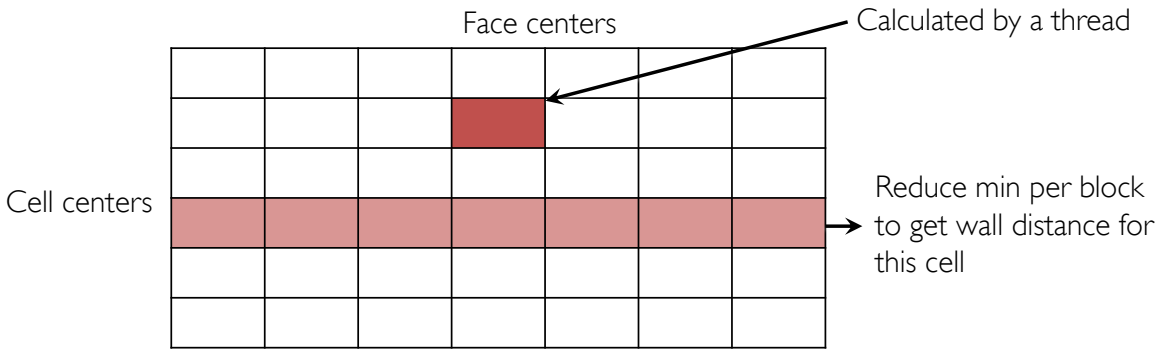


Brute-force – Serial



Brute-force – Parallel (Block Per Cell)

- Each block calculates wall distance for a cell
  - Each thread calculates distance from a face
  - Reduce minimum to get wall distance

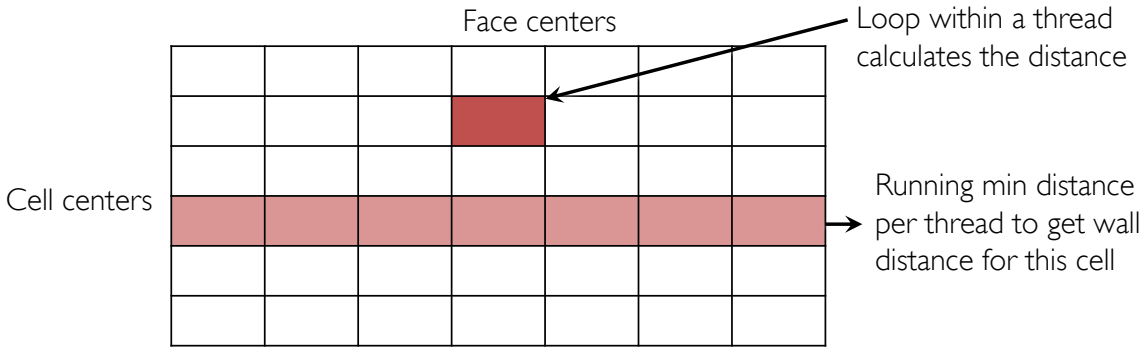


Brute-force – Parallel (Block Per Cell)

- Implemented 3 variations
  - (A): Shared memory writes for face distances
  - (B): (A) + Shared memory reads to face (x, y) arrays
  - (C): Same as (B) with (x, y) arrays converted to an interspersed array for coalesced shared mem reads
    i.e. x0, y0, x1, y1, ...

Brute-force – Parallel (Thread Per Cell)

- Each thread calculates wall distance for a cell
  - Calculates distance from the cell to each of the faces
  - Keeps "running" min distance value as each faces distance gets calculated

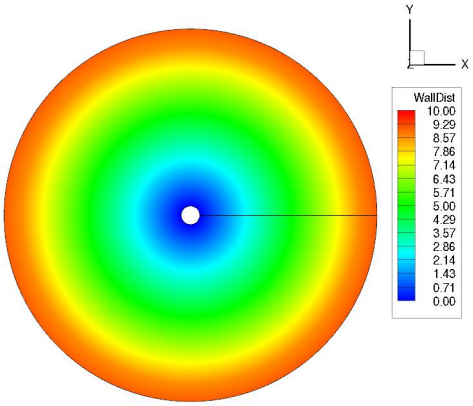


Brute-force – Parallel (Thread Per Cell)

- Implemented 3 variations
  - (A): Each thread calculates wall distance for a cell
  - (B): Shared memory reads to face (x, y) arrays
  - (C): Same as (A) with (x, y) arrays converted to an interspersed array for coalesced global mem reads
    i.e. x0, y0, x1, y1, ...

Brute-force – Verification

- Output from serial algorithm visually inspected for correctness
- Outputs from parallel algorithms verified by comparing against serial algorithm output



Brute-force – Performance

Algorithm	Small Grid (msec)	Fine Grid (msec)	Extra Fine Grid (msec)
Serial	33	595	13636
Block Per Cell (1A) - Shared mem writes	1	28	-
Block Per Cell (1B) - Shared mem reads	1	30	-
Block Per Cell (1C) - Coalesced shared mem reads	1	31	-
Thread Per Cell (2A)	<1	7	173
Thread Per Cell (2B) - Shared mem reads	1	9	293
Thread Per Cell (2C) - Coalesced global mem reads	<1	7	166

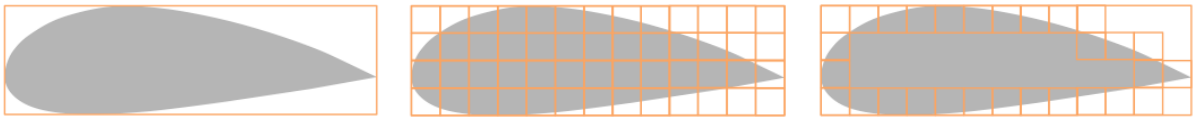
82x speedup vs. serial

Advancing boundary algorithm

Advancing boundary algorithm

- Background: Pre-processor

Goal: create a smaller subset of faces that need searched

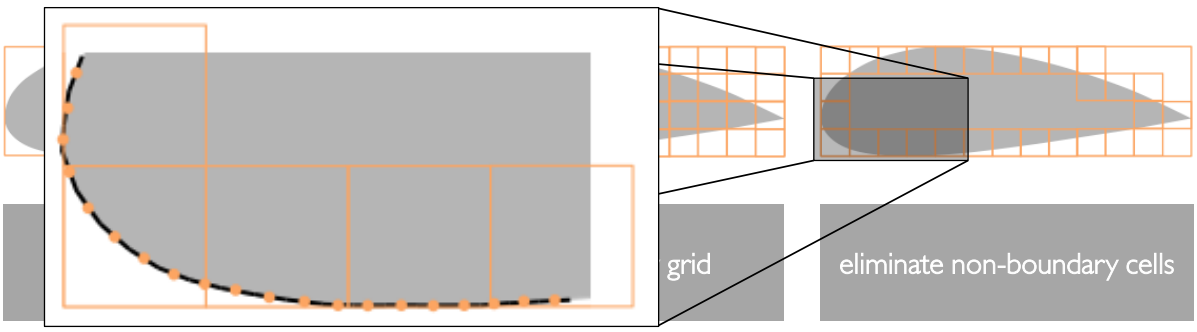


create bounding box | create auxiliary grid | eliminate non-boundary cells

Advancing boundary algorithm

- Background: Pre-processor

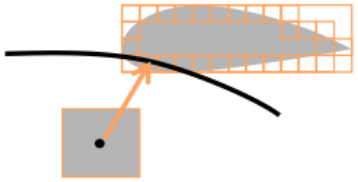
Goal: create a smaller subset of faces that need searched



grid | eliminate non-boundary cells

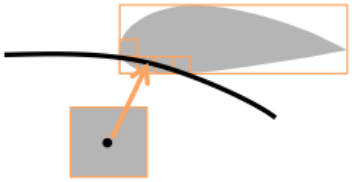
Advancing boundary algorithm

- Background: GPU-Kernel



specify radius based on nearest bounding box edge

Step #1

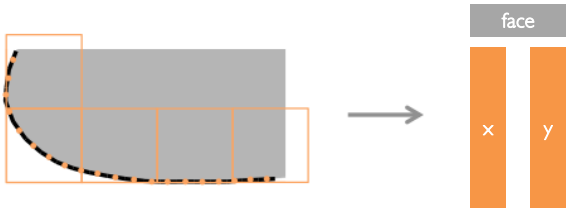


search for auxiliary cells that lie within specified radius

Step #2

Advancing boundary algorithm

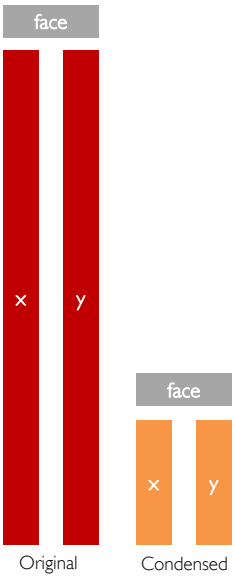
- Background: GPU-Kernel



auxiliary cells include smaller subset of the original set of faces

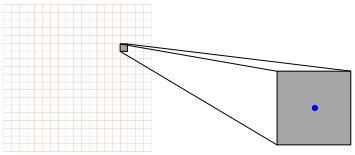
gather remaining face coords into a face struct

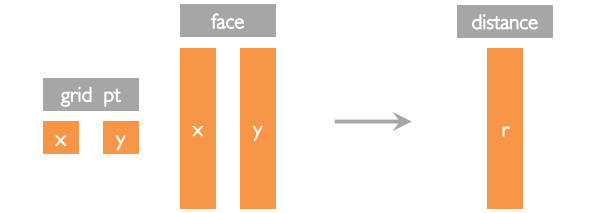
Step #3



Advancing boundary algorithm

- Background: GPU-Kernel



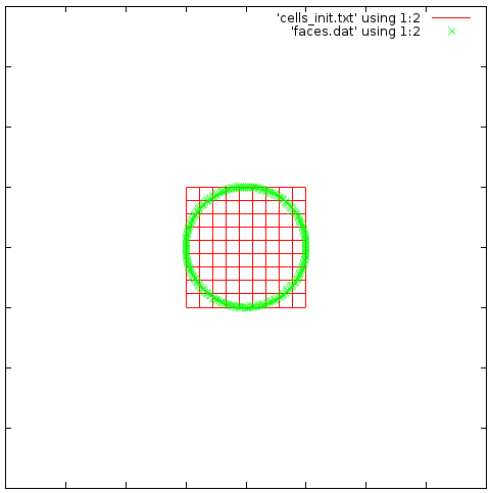


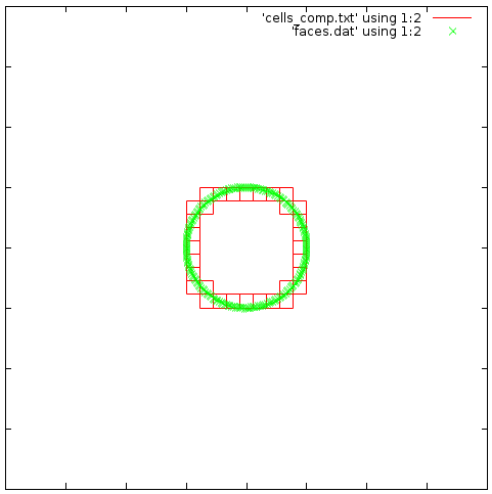
compute distance from grid point to all remaining face points

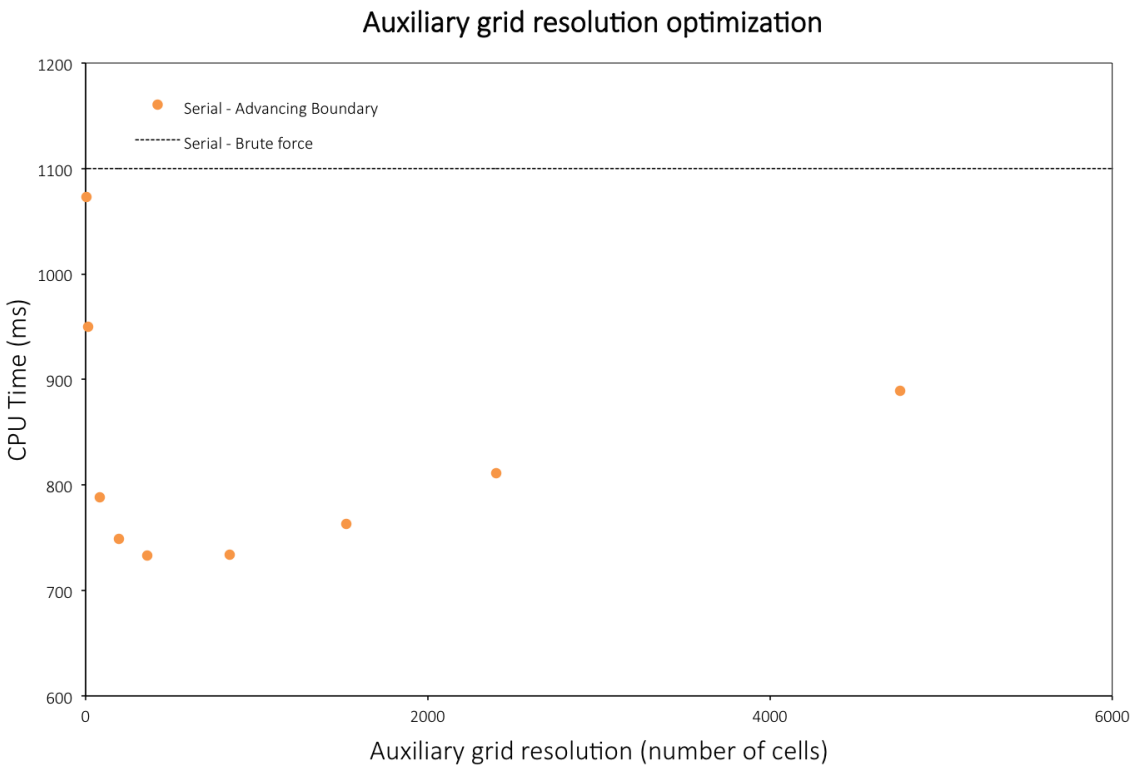
Step #4

Advancing boundary algorithm

- Implementation: Pre-processing







Advancing boundary algorithm

- Implementation: Aux Cell Structs

```
struct cell{

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

    // Included faces linked list
    struct face * root;

};
```

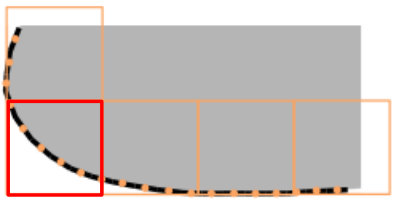
```
struct cell_t2{

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

    // Included faces linked list
    int faceNum;

    int storage;
    double * xface;
    double * yface;

};
```



```
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];

};
```

```
struct cell_pt3{

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

    // Included faces linked list
    int numFaces;

    double faces[200];

};
```

```
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];

};
```

Advancing boundary algorithm

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

Parallel	Description	Average Time (ms)	Speedup	
			Serial CPU vs. GPU	Parallel CPU vs. GPU
T1	Baseline, no shared memory	134	65	5.5
T2	Memory management	114	77	6.4
T4		114	77	6.4
T5		112	78	6.5
T6		113	78	6.5
pt2_T5	New aux cell struct	106	83	6.9
pt3_T5	New aux cell struct	102	86	7.2

Conclusions

- Successful GPU implementation of two wall distance algorithms
- Order-of-magnitude type speedups realized

Thank you…