# A Fast, Parallel Potential Flow Solver

John Moore
Advisor: Jaime Peraire

December 16, 2012

# Outline

# Why Potential Flow?

- It's easy and potentially fast
- Potential Flow: $\nabla^2 \phi = 0$ vs:
- Navier-Stokes: $\rho \left( \frac{\partial V}{\partial t} + V \cdot \nabla V \right) = -\nabla p + \nabla \cdot T + f$
- Linear system of equations
- Full-blown fluid simulation (Navier-Stokes) is expensive
- Many times, we are just interested in time-averaged forces, moments, and pressure distribution.

# Examples

Start movie [1]

[1] **P.O. Persson**

Discontinuous Galerkin CFD.
Runtime time: $>$ 1 week. 100s of CPUs

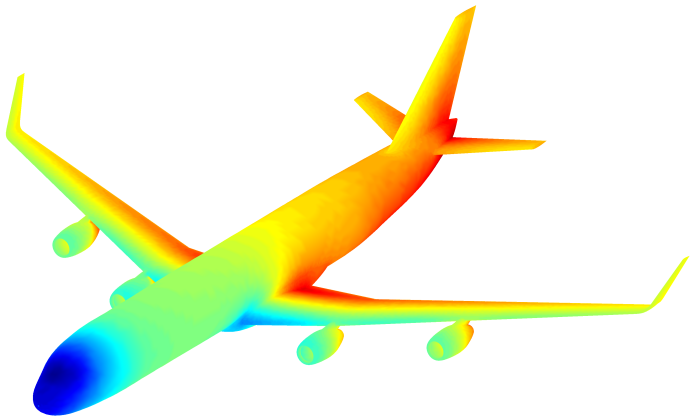


Figure 1: Potential Flow Solution.
Runtime time: 2 minutes on 4 CPUs

# Potential Flow: Can be Fast, But...

- Cannot model everything (highly turbulent flow, etc).
- Accuracy issues due linearisation assumptions...

## Potential Flow Assumptions

- Flow is incompressible
- Viscosity is neglected (can be a major cause of drag)
- Flow is irrotational ($\nabla \times \vec{V} = 0$)

- But, it turns out to predict aerodynamic flows pretty well for many cases (examples: Flows about ships and aircraft)

# Potential Flow: Governing Equation

- Governed by Laplace's equation $\nabla^2\phi = 0$
- Potential in domain written as: $\phi(\vec{r}) = \phi_s + \phi_d + \vec{V}_\infty \cdot \vec{r}$
- Enforce that there is no flow in surface-normal direction...
- Force *perturbation* potential to vanish just inside the body:
- $\phi(\vec{r}) = \phi_s + \phi_d = 0$
- Basically forces the aerodynamic body to be a streamsurface

# Potential Flow: Discretization

Can be discretized using the Boundary Element Method (BEM)

## BEM summary

1. Divide boundary into N elements
2. Analytically integrate Green's function over each of the N elements
3. Compute the potential due to singularity density at each element on all other elements
4. Solve for the surface singularity strengths

The BEM requires that either a Neumann or Dirichlet boundary condition be applied wherever we want a solution.

# Boundary Element Method: Green's Function

- There are several Green's functions that satisfy Laplace's equation:
- Singe-Layer potential: $G_s(\sigma_j, \vec{r}_i - \vec{r}_j) = \frac{1}{4\pi} \frac{\sigma_j}{||\vec{r}_i - \vec{r}_j||}$
- Double-Layer potential: $G_d(\mu_j, \vec{r}_i - \vec{r}_j) = \frac{1}{4\pi} \frac{\partial}{\partial \hat{n}_j} \frac{\mu_j}{||\vec{r}_i - \vec{r}_j||}$
- $\phi(\vec{r}_i) = \int_{S_j} (G_d(\mu_j, \vec{r}_i - \vec{r}_j) + G_s(\sigma_j, \vec{r}_i - \vec{r}_j)) = 0$
- These Green's functions can be analytically integrated to arbitrary precision over planar surfaces
- Analytic integral can be very expensive...

# Boundary Element Method: Collocation vs. Galerkin

- Collocation: Enforce boundary condition at N explicit points.

**Galerkin: Enforce Boundary condition in an integrated sense over the surface**

1. Write unknown singularity distribution $\mu$ as a linear combination of N basis functions $a_i$
2. Substitute into governing equations, and write a residual vector $R$
3. Multiply by test residual by test function.
4. Choose test function to be basis function $\rightarrow$ residual will be orthogonal to the set of basis functions.

$$R_i = \int_{S_i} a_i \phi_i(\vec{r}_i) dS_i = \int_{S_i} a_i \left[ \sum_{j=1}^{N_E} \left( \frac{1}{4\pi} \int_{S_j} \mu_j \frac{\partial}{\partial \hat{n}_{S_j}} \frac{1}{||\vec{r}_i - \vec{r}_j||} dS_j + \frac{1}{4\pi} \int_{S_j} \sigma_j \frac{1}{||\vec{r}_i - \vec{r}_j||} dS_j \right) \right] dSi = 0 \quad (1)$$

# Boundary Element Method: Computational Considerations

- Produces a system that is dense, and may be very large
- For example, the aircraft shown earlier would have resulted in a $180000x180000$ dense matrix
- Would require **259 GB of memory** just to store the system of equations!
- So parallelizing the matrix assembly routine won't help (yet)
- This would be a deal-breaker for large problems, but there is a solution...

# Hybrid Fast Multipole Method (FMM)/ Boundary Element Method (BEM)

What is FMM?

- A method to compute a fast matrix-vector product (MVP)
- Allows MVP to be done in $O(p^4 N)$ operations by sacrificing accuracy, where $p$ is the multipole expansion order.
- We would think that a MVP for a dense matrix scales as $O(N^2)$
- Theoretically highly parallelizable
- More on this later...

FMM can be applied to the BEM

- The FMM is easily applied to the Green's function of Laplace's Equation
- Can think of elements as being composed of many "source" particles
- Maintains same embarrassing parallelism as canonical FMM

# FMM Step 1: Octree decomposition

- Create "root" box encompassing entire surface
- Recursively divide box until there are no more than $N_{max}$ elements in a box.
- Easily Parallizable

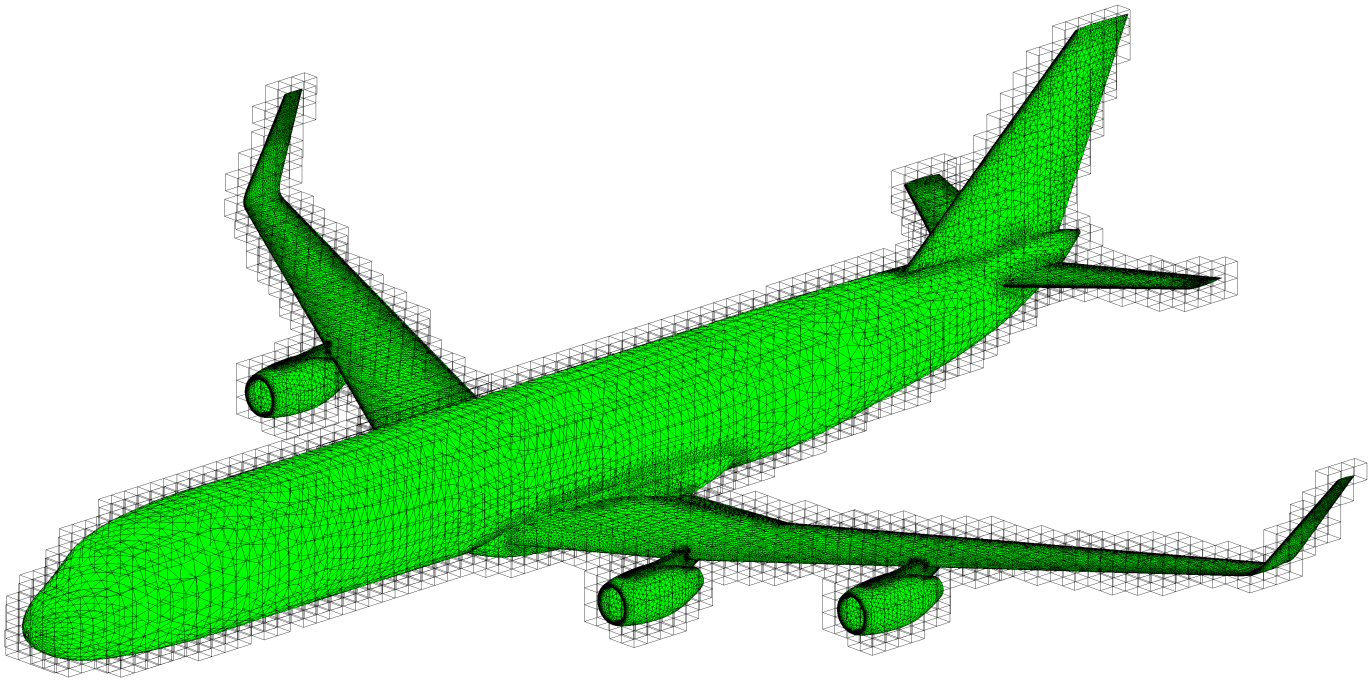
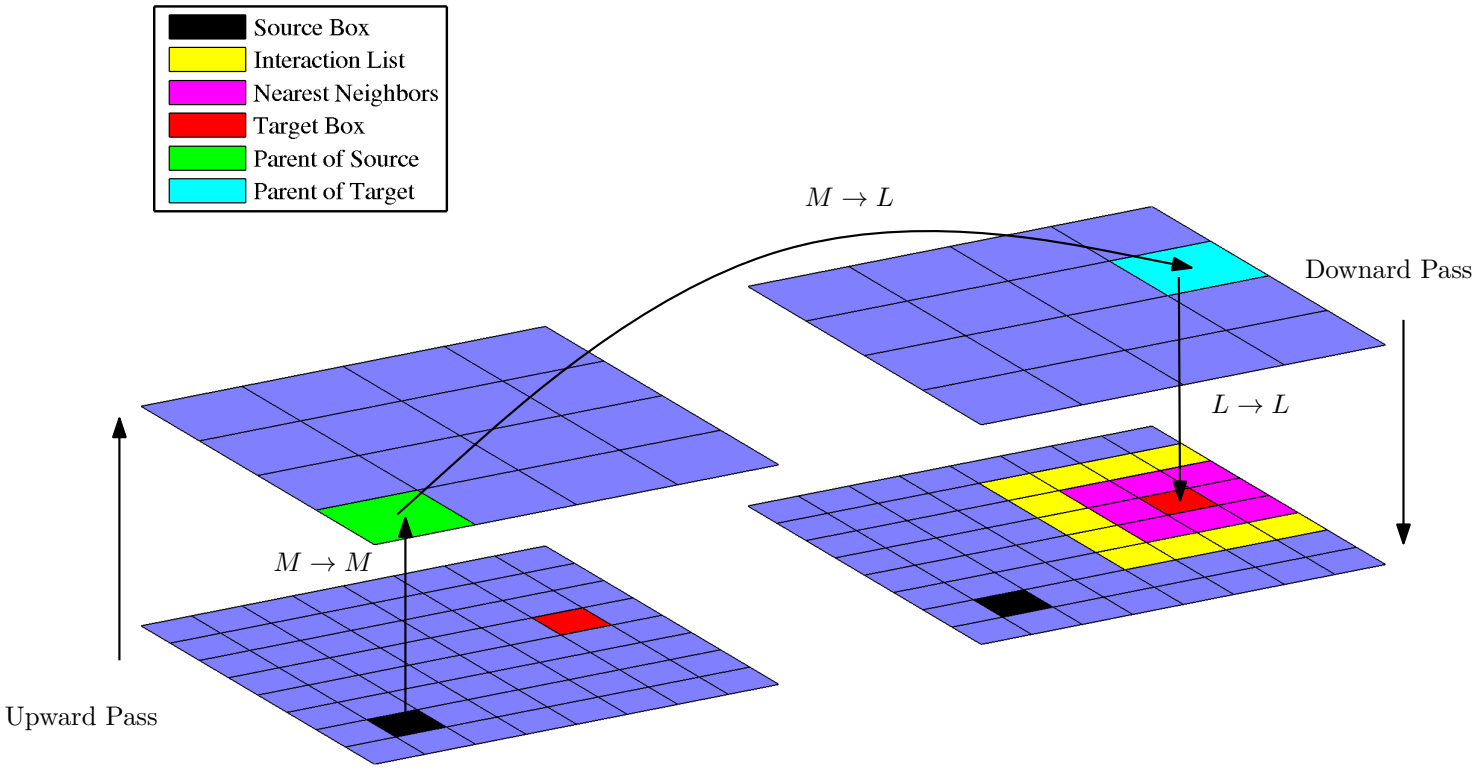


Figure 2: All level 8 boxes in octree about an aircraft

# FMM Steps 2 and 3: Upward and Downward Pass

Basic Idea: Separate near-field and far-field interactions



# Parallelization

## Four Options:

1. Distributed Memory (MPI)
2. Shared Memory (OpenMP)
3. GPU
4. Julia

- Originally, I wrote the FMM code in MATLAB, but was VERY slow
- Switched to C++, code sped up **4 orders of magnitude**
- Now, runtimes are at most several minutes
- Weary of scripting due to MATLAB implementation...
- MPI would be overkill
- Ended up computing matrix-vector product in C++ using OpenMP
- System solved in MATLAB using `gmres`

# Implementation

- First, had to get serial code to work (6,500 lines of code)
- Once serial code available, easy to parallize with OpenMP
- Simply add preprocessor directives and specify # of cores

Example:

```
1  // Multipole to local
2  for(int lev = 2; lev < Nlevel; lev++){
3 #pragma omp parallel for
4  for(int i=0; i<level_index[lev].idx.size(); i++){
5  int bx = level_index[lev].idx[i];
6       if(boxes[bx].isevalbox){
7       for(int j=0; j<boxes[bx].ilist.size(); j++){
8         if(boxes[boxes[bx].ilist[j]].issourcebox){
9           multipole_to_local(boxes[boxes[bx].ilist[j]],
                boxes[bx],Bnm,Anm,ilist_consts[bx][j],
                first_time, compute_single,compute_double);
10          }
11        }
12      }
13    }
14 }
```

# Solving the System

- Ax=b solved with GMRES
- Matrix is reasonably well-conditioned, but can we do better?
- But we never compute the A matrix, so how do we create a preconditioner?
- Assemble sparse matrix containing only near-field interactions.
- Then perform ILU on the near-field influence matrix to create preconditioner

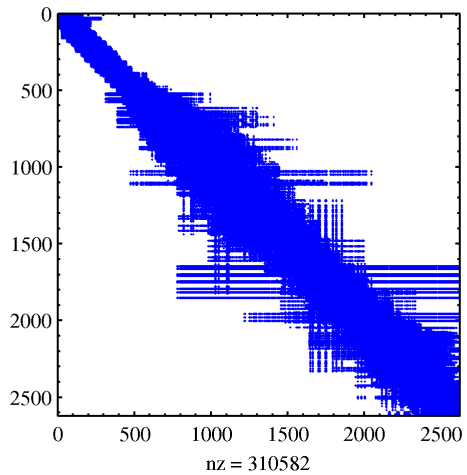



Figure 3: Near-field influence Matrix

# Test Case

## Falcon Buisness Jet

5234 Elements, 2619 Nodes

Linear Basis Functions

Requires $>$ 5 minutes to compute solution without FMM

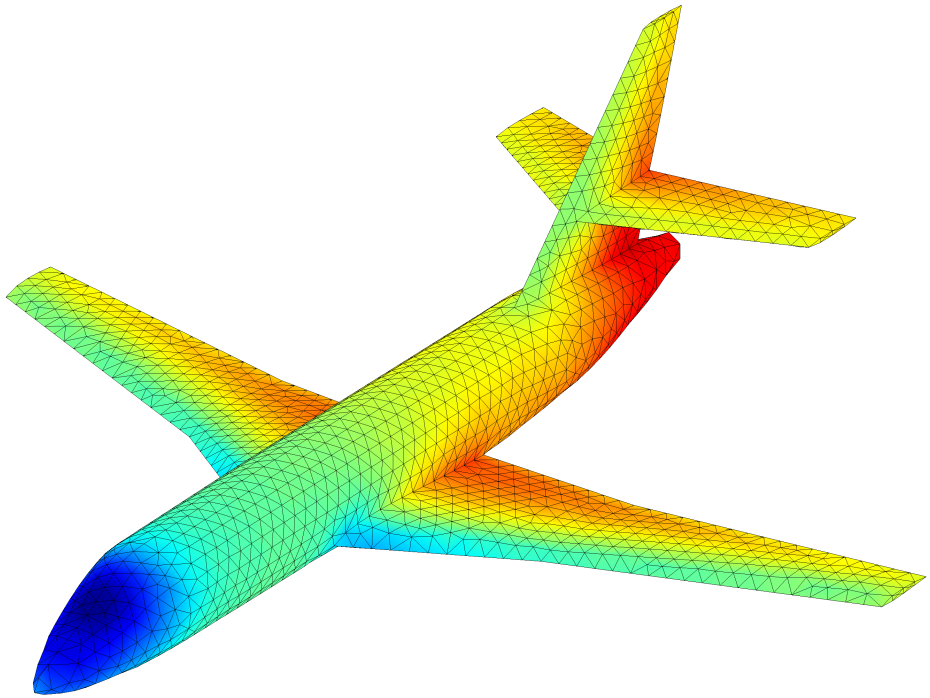


Figure 4: Falcon business jet

# Results

Table 1: Speedup compared to 1CPU

| p | 1 CPU (s) | 2 CPUs | 3 CPUs | 4 CPUs |
|---|---|---|---|---|
| 1 | 5.2414 | 1.24 | 1.36 | 1.37 |
| 2 | 8.6618 | 1.39 | 1.59 | 1.70 |
| 3 | 23.8976 | 1.65 | 2.04 | 2.34 |
| 4 | 52.4548 | 1.73 | 2.26 | 2.59 |
| 5 | 105.9322 | 1.76 | 2.38 | 2.79 |

# Conclusions

1. C++ is so much faster than MATLAB for BEMs
2. Only really makes sense to use shared memory parallelism (like OpenMP) for this application
3. Speedups of 2.8X possible on 4 CPUs in some cases
4. Implementation can be improved: this was my first attempt at parallel programming

# Thank you for your time!

Questions/Comments?

Table 2: Percent Speedup

| p | 1 CPU | 2 CPUs | 3 CPUs | 4 CPUs |
|---|---|---|---|---|
| 1 | 5.2414 | 4.2134 | 3.8450 | 3.8305 |
| 2 | 8.6618 | 6.2183 | 5.4455 | 5.1049 |
| 3 | 23.8976 | 14.5198 | 11.7185 | 10.2322 |
| 4 | 52.4548 | 30.3419 | 23.2016 | 20.2512 |
| 5 | 105.9322 | 60.3612 | 44.4813 | 37.9687 |