Effect of Partially-Ionized Fuel on Mixing and Combustion of Transverse Jet in Supersonic Cross-flow

Abhinab Paul Choudhury
Muhammad Suleman Mahmood
Yu Jia

1. Introduction

Research on hypersonic vehicle technology has seen a surge, in recent times, due to its myriad applications in aerospace and defence industries. One of the key enabling technologies for the realization of hypersonic vehicle is supersonic combustion. However, the high velocity of free-stream and small residence time of the flow inside the combustor makes supersonic combustion very challenging. Furthermore, combustion at supersonic speed results in the formation of shockwaves, which in turn leads to total pressure loss.

A variety of fuel injector designs, including ramp and strut injectors, and flame holding devices, primarily cavity flame holders, have been studied in literature for mixing and flame holding in supersonic combustors. Recently, plasma assisted combustion have also been studied, with electrodes being placed ahead or/and behind the fuel injector. A recent study by Firsov et al[2] studied the mixing and flame holding in supersonic flow using dielectric discharge, which is partially located inside the injector orifice.

In this study, a parallel Computational Fluid Dynamics (CFD) code is developed using Charm++ to study the effect of electrons on the combustion chemistry inside a scramjet.

2. Program Design

In order to simulate fluid flow, the fluid domain is divided into a number of 3D chares called Cell chares. Each chare is again divided into a number of control volumes or cells. The Cell chares are responsible for computing the source terms, i.e., changes in the concentration and energy due to chemical reactions. The cell chares also assimilate the fluxes from flux chares to calculate the flow variables and species concentration for each time step.

The basic components in each volume section are the flow objects. The flow struct is used to store flow variables. It stores information like the velocity, density, and energy for each cell. To simplify the model, the program focuses on implementing only inviscid flow, and therefore quantities like viscous stresses and turbulence, are neglected.

The Cell chares can do their computations assuming isolation for only a short duration. After that, the Cells needs to communicate with their neighbors so that they can update information according to fluxes coming in and leaving the volume represented by those Cells. The information about flow travelling from one Cell to all neighboring Cells also needs to be computed. This calculation is delegated to another chare called Flux chare. There is one Flux chare for every cell chare. To calculate the fluxes at the boundary of each cell/flux chare, a new
4D chare called Interface chare is created. This chare receives flow information from the neighboring flux chares and calculate the flux at the interface. The first index of this chare represents the direction of normal vector for the surface of the Interface chare. The remaining three identify the location of the interface. A more clear and concise summary of the design can be found in the following table.

Table 1: Program Structure

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Intuitive Meaning</th>
<th>Members and Methods</th>
<th>Interactions with others</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flow (C++ struct)</td>
<td>Fluids in Motion</td>
<td>Density; Velocity (in three directions); Energy and species concentration</td>
<td>Part of every chare</td>
</tr>
<tr>
<td>Cell (3D Chare Array)</td>
<td>Finite Volume Section</td>
<td>3D flows; in-Cell calculations including temperature and species reaction rates; Entry method to communicate with flux</td>
<td>Ask Flux for calculation, and then receive from Flux.</td>
</tr>
<tr>
<td>Flux (3D Chare Array)</td>
<td>Boundary of each cell in a Cell Object</td>
<td>Flux and cell flow variables; Do inviscid flux calculation; Entry method to communicate with interfaces</td>
<td>Ask interface for boundary condition calculation, send communication information back to Cell</td>
</tr>
<tr>
<td>Interface (4D Chare Array)</td>
<td>Interface between two adjacent Cells objects</td>
<td>2D flow since it is a surface; Calculation and communication for both sides of the interface.</td>
<td>Communication with adjacent Flux chares.</td>
</tr>
</tbody>
</table>
3. Communication Pattern

At the very beginning of each iteration, each Cell asks the associated Flux chare to calculate the flux passing through each face of the cells. While the Flux is calculating the fluxes, the cell chare calculates the species production and destruction rate due to chemical reaction inside each cell. The flux chare calculates fluxes for each face inside the Cell chare object, except the boundaries. The flux for faces located at the interface of two chares are calculated by another chare called Interface chare. This chare receives flow data from the neighboring flux chares, calculates the fluxes and sends them back to the adjacent flux chares. The flux chare calculates the cumulative flux of flow variables for a cell and sends it to Cell chare. The Cell chare calculates the flow variables from the fluxes and the source terms (species production/destruction rate) and continues to next iteration.

This communication pattern allows parallelization of all the independent calculations. The fluxes and source terms are calculated separately. Also, the most computationally intensive task is carried out by the Cell chare, so that the communication of data with the Flux chare can be overlapped with computation of the source term inside the cell chare. The Flux chare calculates fluxes of flow variables, which is calculated once for each time. However, to compute the source terms, multiple sub-iterations have to conducted as the chemical rate equations are very stiff.

The Interface chare provides mechanism to compute the fluxes at the interface of two adjacent chares. It is also responsible for imposing boundary conditions at the boundaries of the flow domain. The interface chare determines the type of boundary from its location and imposes the appropriate boundary conditions.
4. Serial Computations

The design presented above recognizes three different serial computations.

1. Calculation for source terms in the Cell chares
2. Calculation of fluxes in the Flux chares.
3. Calculation of fluxes and imposition of boundary conditions

For calculation of the chemical reactions, fourth-order Runge-Kutta (RK4) method is used. It calculates the value of the variable at the next time step by calculating four-intermediate values and use weighting to combine them. A typical RK4 method is shown below:

\[
\begin{align*}
\frac{dy}{dt} &= f(t,y), \quad y(t_0) = y_0 \\
\frac{dy}{dt} &= f(t, y(t)) \\
\frac{dy}{dt} &= f(t, y(t) + \frac{h}{2}, y(t) + \frac{h}{2}) \\
\frac{dy}{dt} &= f(t, y(t) + h, y(t) + h) \\
\end{align*}
\]

\[
k_1 = f(t, y_n) \\
\frac{y_n + \frac{h}{2}}{2} \\
\frac{y_n + h}{2} \\
\frac{y_n + h}{2} \\
\]

In the above equation y at n+1 timestep is calculated using data from y at timestep n. First the change in y with respect to x is calculated for half a timestep. The updated value of y is used to calculate another value of y for the same timestep. This updated value is used to calculate the value of y at n+1 timestep. These three values and value of y at timestep n is summed with higher weighting for the mid-timesteps to find the value of y at timestep n+1.
To calculate the reaction rates for plasma-assisted methane-air combustion, 84 reversible reactions of DRM19, which is a reduced mechanism using GRI-Mech-1.2, consisting of 19 species in addition to N\textsubscript{2} and Ar. Another 51 irreversible reactions are used from Bak et al\cite{3} for plasma chemistry. The reaction rate equation is shown below.

\[ \alpha A + \beta B \rightarrow \gamma C + \delta D \]

\[ \text{Rate} = -\frac{1}{a} \times \frac{\Delta[A]}{\Delta t} - \frac{1}{b} \times \frac{\Delta[B]}{\Delta t} + \frac{1}{c} \times \frac{\Delta[C]}{\Delta t} + \frac{1}{d} \times \frac{\Delta[D]}{\Delta t} \]

For calculation of inviscid flux, Simple Low Diffusion Advection Upstream Splitting Method (SLAU2) is used. In SLAU2, the flux-splitting is determined by the local Mach Number and direction of flow in the adjacent cells of the face for which flux is to be determined. If the local Mach number is less than 1, the fluxes are calculated by using values of both the cells. If the local Mach number is more than 1, values from the upwind values are used. The equations are described below.

\[ \Gamma \frac{\partial Q}{\partial t} + \frac{\partial F_k}{\partial x_k} = \frac{\partial F_{V_k}}{\partial x_k} \]

\[ Q = \begin{bmatrix} \rho \\ \rho u_k \\ \rho v_k \\ \rho E \end{bmatrix}, \quad F_k = \begin{bmatrix} \rho u_k \\ \rho u_k u_k + \rho \delta_{ik} \\ \rho u_k v_k \\ \rho u_k H \end{bmatrix} \]

The above equation describes the inviscid Navier-Stokes equation in matrix form. The following equation describes the same in format used in SLAU2.

\[ F_{SLAU} = \frac{\vec{m} + \vec{h}}{2} \Psi + \frac{\vec{m} - \vec{h}}{2} \Psi^\top + \vec{p} \vec{N} \]

\[ \Psi = (1, u, v, w, H)^T, \quad \vec{N} = (0, n_x, n_y, n_z, 0)^T \]

The mass flux is calculated in the following way.

\[ (\vec{m})_{SLAU} = \frac{1}{2} \left[ \rho_L |V_{nl}| + |V_n| + \rho_R |V_{nr}| - |V_n| - \frac{V}{c} \delta p \right] \]

\[ |V_n|^+ = (1 - g) |V_n| + g |V_{nl}|, \quad |V_n|^+ = (1 - g) |V_n| + g |V_{nr}| \]

\[ |\vec{v}_n| = \frac{\rho_L |V_{nl}| + \rho_R |V_{nr}|}{\rho_L + \rho_R} \]

\[ g = -\max[\min(M_L, 0), -1] \cdot \min[\max(M_R, 0), 1] \quad \in [0, 1] \]
The pressure flux is:

\[ \chi = (1 - \tilde{M})^2 \]
\[ \tilde{M} = \min \left\{ 1.0, \frac{1}{c} \sqrt{\frac{u_x^2 + u_y^2}{2}} \right\} \]
\[ M = \frac{V_n}{c} = \frac{u \cdot n}{c} \]
\[ \frac{c}{c} = \frac{c_L + c_R}{2} \]

The pressure flux is:

\[ (p)_{SLAT} = \frac{p_L + p_R}{2} + \frac{p^+|_{a=0} - p^-|_{a=0}}{2} (p_z - p_x) + \sqrt{\frac{u_L^2 + u_R^2}{2}} \left( p^+|_{a=0} + p^-|_{a=0} - 1 \right) \]

\[ p^+|_a = \begin{cases} \frac{1}{2} (1 \pm \text{sign}(M)), & \text{if } |M| \geq 1 \\ \frac{1}{4} (M \pm 1)^2 (2 \mp M) \pm \alpha M (M^2 - 1)^2, & \text{otherwise} \end{cases} \]

To impose boundary conditions, a ghost cell-like approach is implemented, without actually creating ghost cell charges. The ghost cell information is stored at the boundary interface charges. In this study, three boundary conditions have been implemented, namely wall, inlet and outlet. For the wall boundary condition, all the variables from the adjacent cell are reused, except the normal velocity, which is reflected. This is done to impose zero flux through the wall and also avoid numerical instabilities. The inlet condition is imposed in two ways depending on the flow Mach number. If Mach number is greater than 1, pre-determined values of all the variables are imposed. However, if Mach number is less than 1, pressure is reused from the adjacent cell and everything else is akin to greater than Mach number 1 situation. For outlet condition, if Mach number is greater than 1, the ghost cell has all the values equal to the values at the adjacent cell but if Mach number is less than 1, predetermined pressure value is imposed on the ghost cells.

5. Experiments and Test Cases

Test Cases:
To test the validity of the numerical algorithm of the code, two tests are performed. To check the communication and computation pattern works perfectly, Sod Shock Tube test is performed, in which two fluids are separated by a diaphragm. At t=0, the diaphragm breaks, this results in a flow from high pressure to the low pressure region. This also creates a shock wave traveling towards the low pressure region and an expansion wave traveling towards the high pressure region. The results at t = 0.2 s are compared with analytical solution below.
From the figure, it is seen that the numerical solution is a good match with the analytical solution. A slight discrepancy comes from the numerical dissipation provided by the numerical algorithm for stability.

Also, the plasma chemistry is tested separately. Methane-air combustion at 1000K with and without plasma is tested and is shown below.

It can be clearly seen that the addition of electrons change the chemical pathway of the reactions. Without the plasma, the concentration of H radical is more than O radical. However, when plasma is turned on, more O radicals are produced than H radical. This is expected as ionization
energy of $\text{O}_2$ is less than $\text{CH}_4$ and only one H radical is created for every $\text{CH}_4$ atom and two O radicals are created for every $\text{O}_2$.

**Experiments:**
For this project we have two versions of the code. The first version does not have all the boundary conditions for fuel injection, air inlet and outlet for simplicity. This version of the code is used for basic analysis of the parallel structure of the code and doesn’t perform a full simulation of Plasma Combustion inside a scramjet. However, it does perform Plasma combustion in a shock tube, which is a common experimental method to study combustion chemistry. Since the parallel structures and communication patterns of both versions are same, all the performance analysis results presented in this report come from this version of the code. The second version which simulates the plasma-assisted combustion in scramjets has been run for small number of time steps as the program is too computationally intensive, due to stiff equations for plasma combustion chemistry and the program remains in queue forever in Campus Cluster if a high number of processors or time are requested. We had one job with a request of 8 nodes and time request of 1h 40min in queue for more than 24 hours before we cancelled it.

**Experiment 1: Runtime Analysis**
For this project we have experimented with different values for a number of iterations in the serial computation part as well as the number of cells. The relationship between the number of iterations in serial communication and the total runtime is roughly linear where other parameters like number of processors and cells are kept constant. The results are shown in Figure 5. The number of processor cores was 12 and the number of chares was 80 in all readings.

Similarly we investigated the relation between the number of chares and the runtime. The runtime increases as the number of chares increases. But in our case, the total number of iterations required also depends on the number of Cells. In this study, the Courant number is fixed, so the time step is directly proportional to the cell size. This means that the total time required for the simulation increases because of 2 reasons.

1. Total number of iterations increase linearly with cell size (length of cell)
2. Total number of chares increase linearly with cell size (length of cell)

Therefore, we expect a quadratic increase in time with x-dimension of the tube. It is to be noted that in y and z-dimension, the number of chares remain constant as the flow is invariant in those directions. Experimental results confirm this code analysis. We varied the x-dimension while keeping all other factor constant. The runtime is observed to increase quadratically. The results are shown in Figure 6.

Based on the results of these experiments we decided the parameters for further testing. We also used these results to estimate the time required for the jobs submitted for additional experiments.
Figure 5: Change in runtime with increase in serial computation

Figure 6: Changes in runtime against x-dimensions of the Cell array
Experiment 2: Scaling with number of processors
In order to check the scaling of the application with respect to number of processors we tested the application with different number of processors while keeping the application parameters constant. Overdecomposition is the key to achieving good scaling so we have used 500 shares for our simulation. In order to keep the runtime at a reasonable level we set the serial iteration to 100. The results of the scaling experiments are shown in Figure 7.

![Graph showing runtime vs number of processors](image)

**Figure 7: Runtime against Number of Processors**

Experiment 3: Load Analysis without Load Balancing
We obtained projections data for running the application in all 8 case presented in Experiment 2. Figures 8, 9, 10 and 11 show data for 12, 24, 84 and 96 processors respectively. We omit the data for 36, 48, 60 and 72 nodes because they show patterns which are simply interpolation of the presented cases. Usage profiles are shown averaged over a 10s period close to the end of the execution. These show the usage patterns when small number of processors are used and pattern when large number of processors are used. We observe that all processors are generally less loaded when we have a larger number of processors.

In the case of 96 processors we notice that there is no processor which is 100% loaded so we looked at the communication pattern to find the cause of the bottleneck. Figures 12 -15 show the number of messages received for the 4 cases for the same timeslots. The pattern for number of messages sent and data sent and data received is similar to the patterns shown by the number of messages sent.
Figure 8: CPU usage for 12 processors (1 node)

Figure 9: CPU usage for 24 processors (2 nodes)
Figure 10: CPU usage for 84 processors (7 nodes)

Figure 11: CPU usage for 96 processors (8 nodes)
Figure 12: Communication Pattern for 12 processors (1 node)

Figure 13: Communication Pattern for 24 processors (2 nodes)
We observe that the same communication pattern is present in all 4 cases. There are two chunks of processors which have sent and received a much larger number of messages as compared to the other processors. This indicates that other processors may be waiting on messages from these processors to progress. This may have be the cause of the processors being idle.

Based on these results we decided to include load balancers in our system.
Experiment 4: Load Balancing Comparison

We experimented with 4 load balancing strategies in total. Two of them were load balancer which took communication into account as well. We decided to experiment with communication based load balancers because we observed communication imbalance in our earlier experiments. Our experiments with load balancing were all done with 96 processors, 500 cell chares and 100 iterations in the serial code. We estimated that each cell will takes approximately 2 to 3 seconds to execute 10 iterations so we placed AtSync() call in all three cell arrays at 10 iterations. This gives allows the load balancers to run for around 20 times. The number of times the load balancer gets to balance load is important for RefineLB and RefineCommLB because they rely on moving small number of chares. We believe that its execution for 20 times should be good enough to show its effect. The run with 96 processors was selected because it has the most opportunity to gain from load balancing. The load balancing runtime results are summarized in Table 2.

<table>
<thead>
<tr>
<th>Load Balancing Strategy</th>
<th>Runtime (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>No load balancing</td>
<td>51.8</td>
</tr>
<tr>
<td>GreedyLB</td>
<td>54.5</td>
</tr>
<tr>
<td>RefineLB</td>
<td>46.3</td>
</tr>
<tr>
<td>GreedyCommLB</td>
<td>48.5</td>
</tr>
<tr>
<td>RefineCommLB</td>
<td>51.0</td>
</tr>
</tbody>
</table>

The results above indicate that RefineLB may be the best choice because of the speedup achieved. We investigated the usage and communication patterns for these for more detailed info. For usage patterns we looked at approximately the last 5s of execution in each case and we observed the communication pattern for the same duration. Figures 16-23 show these results.
With GreedyLB the total number of messages has decreased for all processors but the peaks are still there and follow the same pattern as the version without load balancing.
At first the results for GreedyCommLB look similar to GreedyLB but on closer inspection, we see that processor utilization for GreedyCommLB is slightly better as well. This can also be inferred from the runtime which has decreased by about 5s.
RefineLB has the shortest runtime. Its profiles are still similar to greedy. One thing to note in these profiles is that unlike Greedy and GreedyComm, there is a processor which is hitting 100%. Therefore we think that RefineLB may improve even more if it gets more than 20 runs if it can resolve the 100% usage in further runs.
RefineCommLB has worse performance than RefineLB. On closer inspection we see that around 3 processors are nearing 100 with RefineCommLB.
6. Conclusions
In this project, the effect of using plasma to assist combustion in a supersonic flow is simulated. The basic framework for computation is established and its scalability is tested thoroughly. It is observed that the amount of work increases quadratically as the computation becomes fine grained. It is observed that the amount of serial work in each iteration increases the run time linearly. Based on this, we calculated the resources that needs to be executed for all other experiments.

We also studied the scaling of the system as the number of processors are increased and found that the program structure that has been established earlier scales well. It is also found that load is balanced well when overdecomposition is high. For example for the case where nearly 50 chares/processor (12 processor case), load balance is nearly perfect. When there are about 5 chares per processor(96 processor case), load balance is not good.

We also experimented with load balancers to improve the performance for a large number of processor and found that RefineLB showed about 10% improvement in performance. GreedyCommLB is also a good option. Even with these load balancers, there is a lot of potential for improvement as shown by the usage profiles with the load balancers. Based on these details we conclude that overdecomposition with 50 chares/processors is probably the best way to reduce load imbalance for this application.

References:
[7] lxcat.net
[8] GRI-Mech-3.0