Max Le
ID: 901223283
Introduction To Parallel Process
Final Project: Parallelization Of 1D Shocktube

Contents

1 INTRODUCTION

The aim of this project is to parallelize a 1-D shocktube. In compressible aerodynamics, shock waves are detrimental to the performance of aircraft; especially when we are going much faster than the speed of sound (supersonic). In order to simplify things, we are going to assume the following:

- problem is 1D
- affects of heat transfer or external forces are negligible
- ideal gas and single phase
- boundary conditions are zero gradient, so that we are only interested in how the shock propagates.

This problem is governed by a set of hyberbolic PDEs, which representw the conservation of mass, momentum and energy:

$$\frac{\partial \rho}{\partial t} + \frac{\partial \rho u}{\partial x} = 0$$

$$\frac{\partial \rho u}{\partial t} + \frac{\partial (p + \rho u^2))}{\partial x} = 0 \tag{1}$$

$$\frac{\partial E}{\partial t} + \frac{\partial (E + p)u}{\partial x} = 0$$

where ρ, u, p, and E are density, velocity, pressure and total energy respectively. In a compact form, these equations can then be written as:

$$\frac{\partial U}{\partial t} + \frac{\partial F(U)}{\partial x} = 0, \text{ where}$$
$$U = \begin{bmatrix} \rho \\ \rho u \\ E \end{bmatrix} \tag{2}$$
$$F = \begin{bmatrix} \rho u \\ p + \rho u^2 \\ (E + p)u \end{bmatrix}$$

The shocktube looks something like this at initial time:

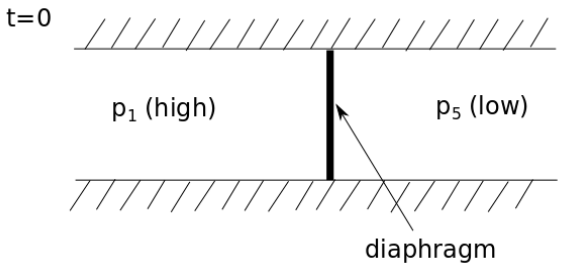

Figure 1: The shocktube at t = 0

The basic idea is that we are going to set the left and right chamber in the shocktube at different pressure and densities. Then, once the diaphragm is removed, we will have a shockwave propagating from the higher pressure region into the lower pressure region. At the same time, we would also see an expansion fan going in the opposite direction. In order to make things simpler, our shocktube's length is from -10 m to 10 m and the location of the diaphgram is at 0 m. The initial data for both chambers are, assuming non dimensional units:

$$
\begin{aligned}
P_L &= 1.00 \\
P_R &= 0.125 \\
\rho_L &= \frac{1.00}{\gamma - 1} \\
\rho_R &= \frac{0.125}{\gamma - 1} \\
U_L &= U_R = 0.0
\end{aligned}
$$

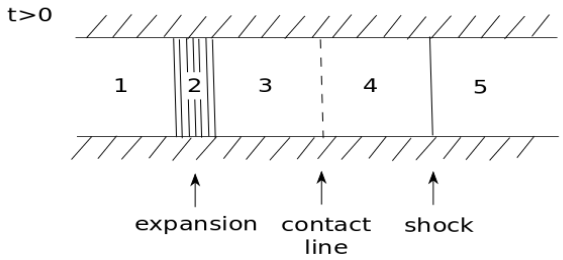

Figure 2: The shocktube as t > 0

This is how the solution looks at initial time:

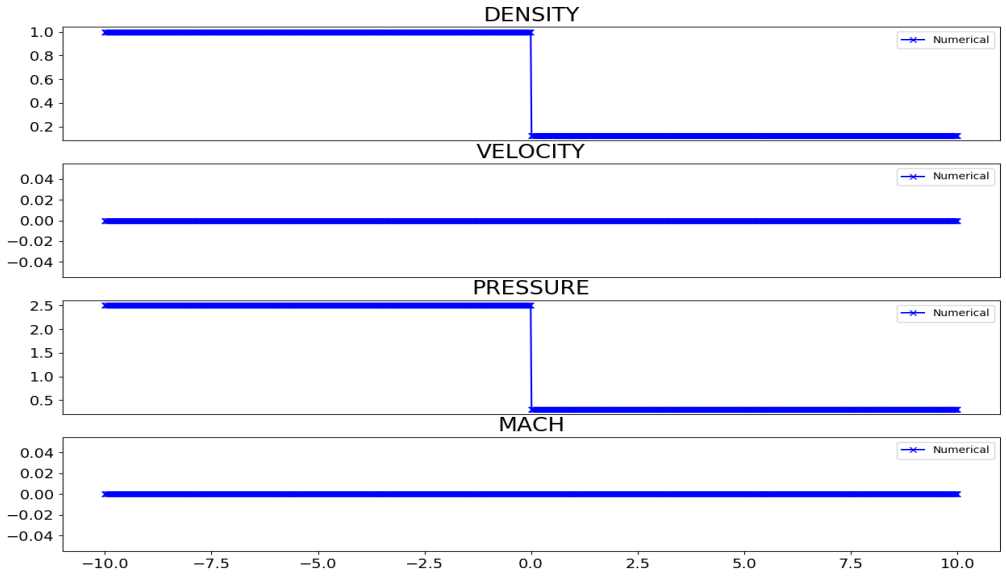

Figure 3: Initial conditions

2 NUMERICAL METHODS

2.1 Discretizations

We are going to employ a first order scheme to solve this problem. Specifically, the temporal derivative is discretized as a first order forward in time, while the spatial derivatives are discretized as first order in space. For the spatial terms, the sign of the speed of sound will dictate whether we should use first order forward or first order backward (upwind and downwind). Generally, the discretization looks like this:

$$\begin{aligned} \frac{\partial U}{\partial t} &\approx \frac{U_i^{n+1} - U_i^n}{\Delta t} + \mathcal{O}(\Delta t) \\ \frac{\partial F}{\partial x} &\approx \frac{F_{i+1}^n - F_i^n}{\Delta x} + \mathcal{O}(\Delta x) \end{aligned} \tag{3}$$

Together, we can write our finite difference equation for this scheme as follow:

$$U_i^{n+1} = U_i^n - \frac{\Delta t}{\Delta x}\left[(F^+ + F^-)_{i+1/2} - (F^+ + F^-)_{i-1/2}\right] \tag{4}$$

2.2 Computational grid

It should be noted that F^+ and F^- stand for positive and negative fluxes respectively. For a given edge, say at $i - \frac{1}{2}$, there will be both positive and negative fluxes.
Our grid for this finite volume method will be a cell centered grid, where U is defined at the center and the fluxes are defined on the edges.

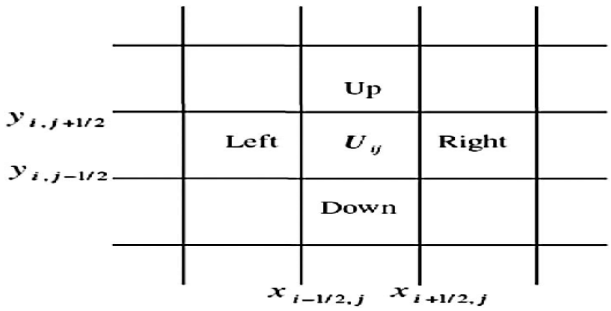

Figure 4: Finite volume grid

In order to retain numerical stability, we would calculate our timestep as follow and the minimum timestep will be chosen:

$$dt = \frac{CFL * dx}{|u| + a} \tag{5}$$

2.3 Fluxes

The fluxes are computed as Van Leer fluxes for its simpliicty and ability to handle both subsonic as well as supersonic case. Denote the local mach number as "M" and the speed of sound as "a", the Van Leer fluxes are as follow for both positive and negative fluxes:

$$\vec{F}^{\pm} = \begin{bmatrix} f_a^{\pm} \\ f_a^{\pm} f_b^{\pm} \frac{1}{\gamma} \\ f_a^{\pm} f_b^{\pm} f_b^{\pm} \frac{1}{2(\gamma^2 - 1)} \end{bmatrix} \tag{6}$$

where f_a and f_b are as follow:

$$f_a^{\pm} = \pm \frac{1}{4} \rho a (M \pm 1)^2$$
$$f_b^{\pm} = \pm (\gamma - 1) M a \pm 2a$$

3 PARALLELIZATION STRATEGY

In order to parallelize this problem, we are going to use MPI to split up the arrays into chunks. Each chunk will be processed by each processor. Again, to keep this simple, we are going to assume that the number of processors is divisible by the array's length. This is done so that we only get whole number when calculate the local indices.

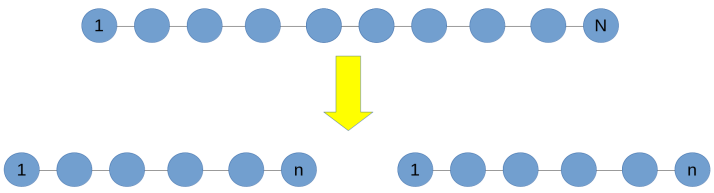

Figure 5: Splitting of the main arrays

The local indices are calculated as follow. Because Fortran starts at 1, so if we have 101 data points; then the last processor will be 1 data point shorter. Therefore, we added 1 point if it is the last processor.

```
nblocks = FLOOR(REAL(IMAX/nprocs))
rem = IMAX - nblocks*nprocs
!! Real start index on each chunk
iStart = (nblocks*rank) + ghostPoints

localLOW = 1+ghostPoints
localHIGH = nblocks+ghostPoints
maxLocal = localHIGH + ghostPoints
minLocal = localLOW-ghostPoints

!! adding the remaining point
if(rem /= 0) then
    if(rank == nprocs-1) then
        localHIGH = localHIGH + 1
        maxlocal  = maxlocal + 1
    end if
end if
```

The current numerical scheme that we are using is a first order scheme; therefore, we would need data from left or right neighbor to update the boundaries. As a result, ghost points are added and then they are exchanged across the boundaries using MPI Isend and MPI Irecv to avoid blocking communications.

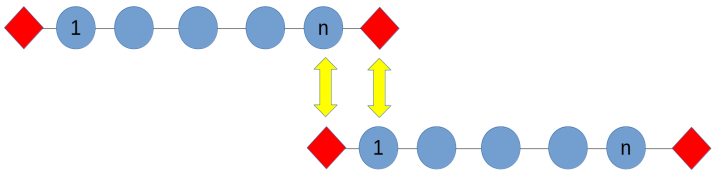

Figure 6: Ghost points send/recv

Furthermore, we can also optimize this by sending more ghost points. The idea is that if more ghost points are sent; then the main solver should have enough data to keep updating the solutions without asking for more communications. In other words, if 4 ghost points are sent; then for the next 4 timesteps, the code only needs to update the solution because it already has enough information (from 4 ghost points) to carry out the calculations.

The way to implement this optimization is by using a flag in the update function. This flag will be set to the number of ghost points and send/recv are also done initially. Then everytime the code runs, this flag will decrease by 1 until it reaches zero and then it would be set to the number of ghost points again. In code, this looks something like this:

```
subroutine update(temp)
    include 'mpif.h'
    use globalvar
    integer, intent(inout) :: temp

    !! FLUX CALCULATIONS
    do i = minLocal, maxLocal

        call vanleer(fiph, fimh, state_vector, i, gamma)
        fGLOBALm(:,i) = fimh(:)
        fGLOBALp(:,i) = fiph(:)

    end do

    !! FINITE DIFFERENCE
    do i = localLOW,localHIGH
        do j=1,3
            U(j,i,2) = U(j,i,1)- (global_dt/dx)*&
            (fGLOBALp(j,i)-fGLOBALp(j,i-1))&
            - (global_dt/dx)*(fGLOBALm(j,i+1)&
            -fGLOBALm(j,i))
        end do
```

```
    end do
    if (temp == 0) then
        print*, "DOING SEND/RECV"
        call bc_send
        call bc_recv
        temp = ghostPoints
    end if
    temp = temp -1
    print*, "This is flag ", temp
    !! Update boundary
    call boundary
end subroutine update
```

We are going to run this code until t = 2 seconds. The main code is written in Fortran 90; while the plotting/analytical results are in Python. The numerical results (my own work) are then compared to the analytical results. These analytical results are available online and information about these are referenced in details in the later chapters.

4 RESULTS

4.1 Numerical vs Analytical

Below is the comparison between the numerical and the analytical results. Excellent agreement is observed for a first order scheme, although this is only valid for low pressure ratio. For higher pressure ratio, a higher order scheme is needed. Please note that this analytical solution is not part of the project. Its only purpose is to show that the results obtained are correct. The analytical solver was taken as a Python version of Dr. Timmes's Fortran code at Arizona State University. This is reference in details later on.

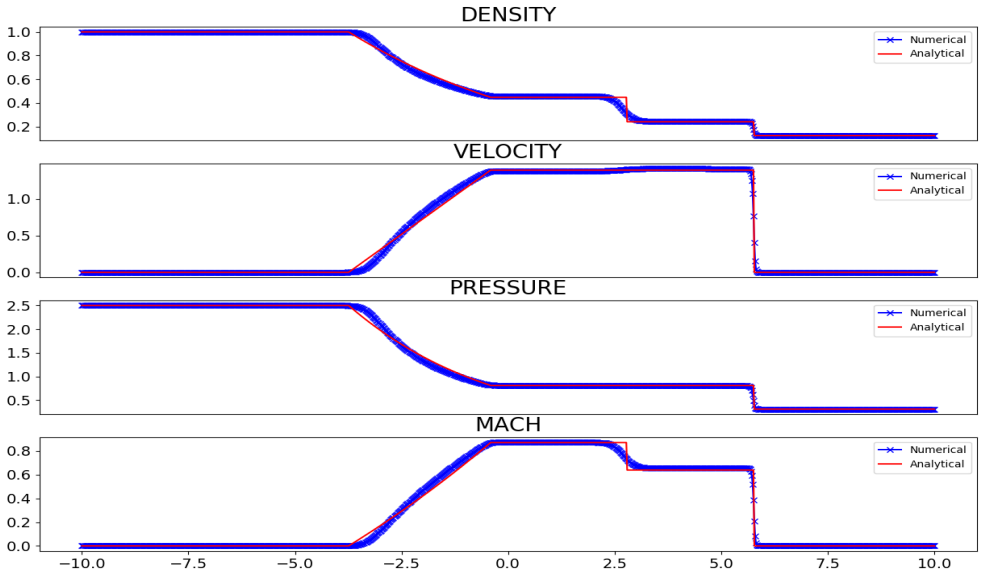

Figure 7: Analytical vs Numerical

4.2 Speedup

The code is timed using the "time" function in Linux. Blueshark runs a version of Linux distro so this method works. This is to prevent the hassle of putting everything back together in 1 giant array at the end. The output is done by having each processor appends out its own version of the array to a file. Using Blueshark, the following speedup results are obtained for a problem size of **10,000** and **2 ghost points**.

CORE	TIME	EXP	THEORETICAL
1	10.888	1	1
2	9.953	1.093941525	2
4	5.425	2.007004608	4
8	3.207	3.395073277	8
10	2.78	3.916546763	10

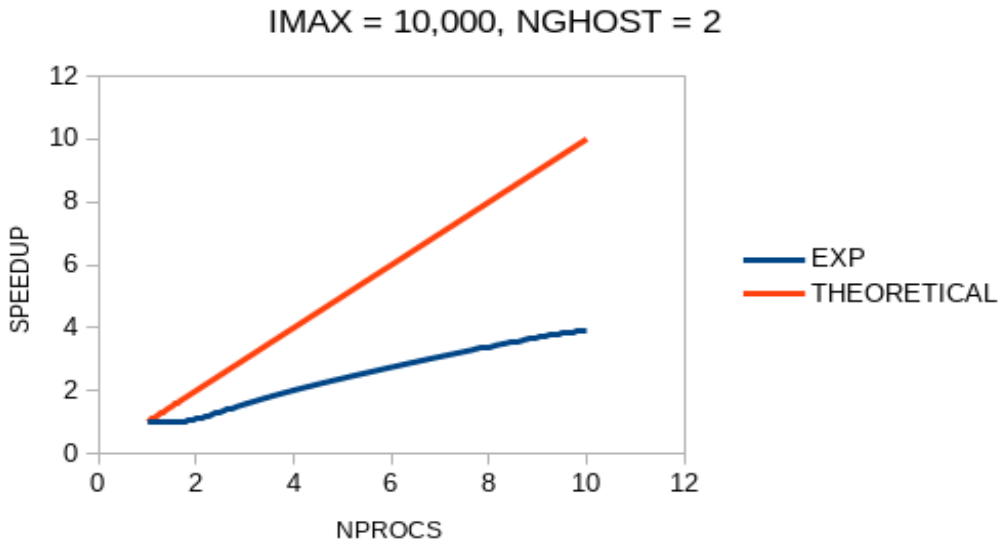

Figure 8: Speedup for 2 ghost points

Also using Blueshark, the results are also obtained for a problem size of **10,000** and **10 ghost points**.

CORE	TIME	EXP	THEORETICAL
1	11.237	1	1
2	10.198	1.101882722	2
4	5.962	1.884770211	4
8	3.941	2.851306775	8
10	3.526	3.186897334	10

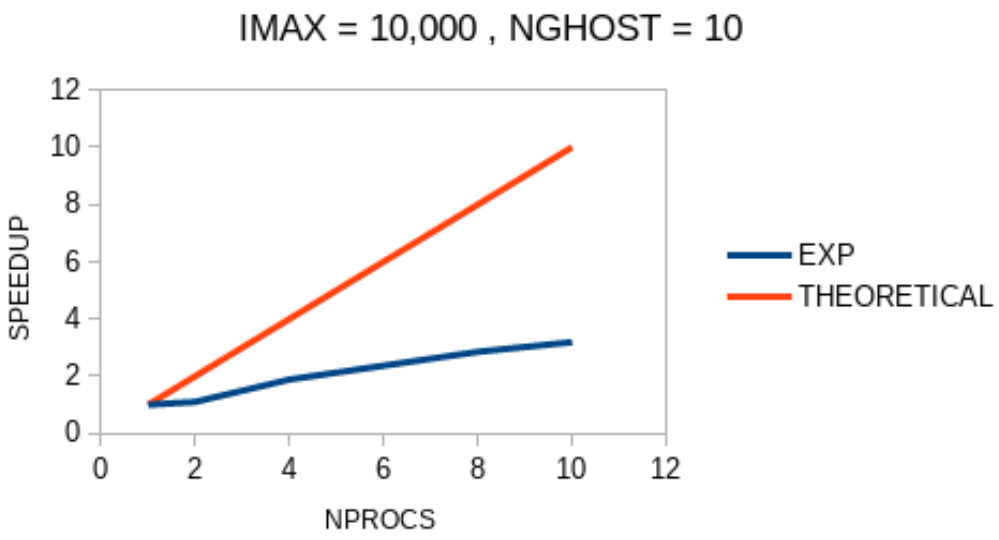

Figure 9: Speedup for 10 ghost points

Effects of ghost points are investigated on smaller problem size

NG	TIME (1 core)	TIME (5 cores)
1	0.568	0.582
2	0.548	0.624
3	0.59	0.568
4	0.612	0.56
5	0.562	0.525

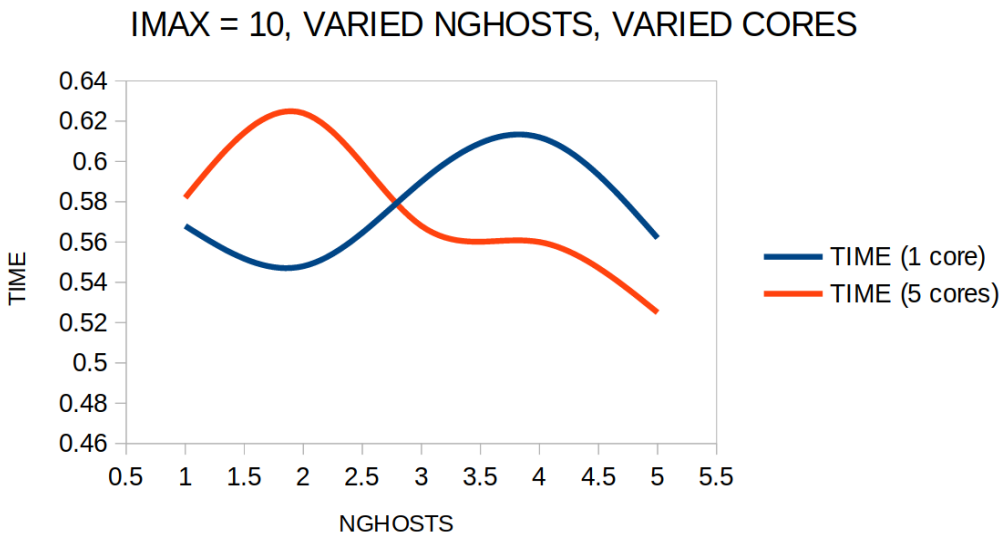

Figure 10: IMAX = 10, varied ghost points and cores

4.3 Scalable

Regarding the scalability of the code, below are results for using Blueshark's "mpif.h" and the official MPI's "use mpi" for Fortran. The problem size is **10,000** and the number of ghost points is kept at **2**. To do the scalability test, we assume problem size and number of core are related by a 1000:1 ratio.

Using Blueshark's mpif.h

N	CORE	TIME	EXP	THEORETICAL
1000	1	0.778	1	1
2000	2	1.053	0.738841405508	1
4000	4	1.664	0.467548076923	1
8000	8	3.113	0.249919691616	1
10000	10	3.695	0.210554803789	1

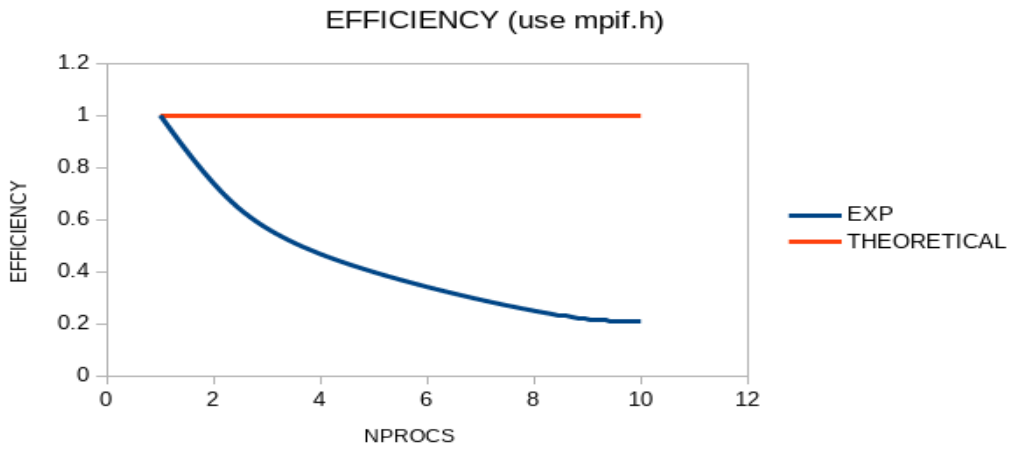

Figure 11: Scaling for mpif.h

Using the official MPI header

N	CORE	TIME	EXP	THEORETICAL
1000	1	0.405	1	1
2000	2	0.481	0.841995841996	1
4000	4	0.655	0.618320610687	1

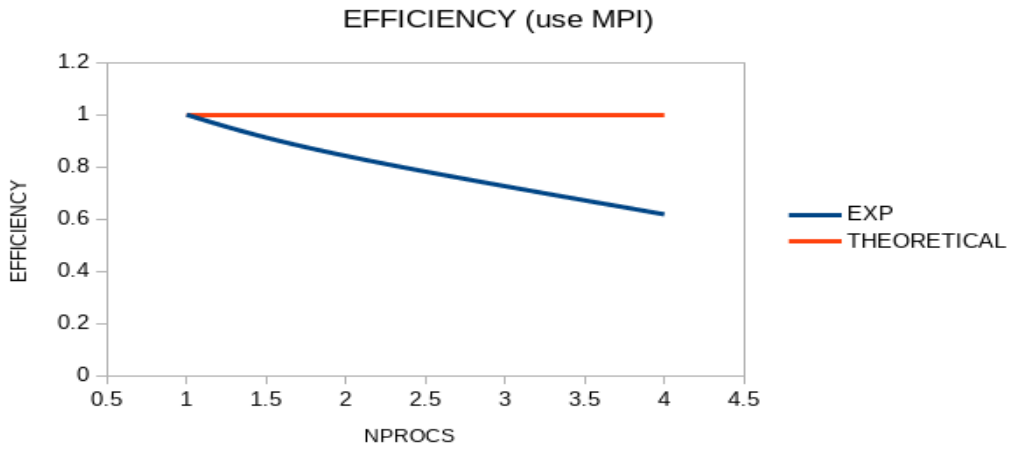

Figure 12: Scaling for mpi

5 CONCLUSION

5.1 Speed up and ghost points

We can see from Figure 7 and 8 that although speedups are achieved, they are not that great. In fact, the experimental speedups are much lower than what we would expect. In terms of ghost points optimization, using 2 ghost points or 10 ghost points do not make a large difference. When using 10 ghost points, on some occasions such as using 8 and 10 cores, the results are worse than using 2 ghost points. In order to investigate this thoroughly, I conduct ran the test again with smaller problem size (IMAX = 10) and then study the effects of ghost points and number of cores. Figure 9 shows that for 1 core, we experience a very small window (between 1 and 2 ghost points) where we do have improvements. However, after this, we do not see any improvements; in fact, it actually takes longer to run the code. For 5 cores unde the same experiment, we can see that there are no improvements initially; however, as we increase the number of ghost points, we actually see speedup. I can then conclude that there is a sweet spot between problem size and number of cores which can help to improve the speedup. In other words, for large problem size in Figure 7 and 8, the number of ghost points are not that great compare to the big problem size. For 10,000 nodes, if I use 10 ghost points, then that only saves 10 iterations where the code does not have to perform any send/recv, this is insignificant. For smaller problem size, 10 nodes, using 5 ghost points can save you 5 iterations which is basically half of the problem size. Then, I tried to see if using 10,000 nodes and a significant number of ghost points can make any difference. The results are as follow:

NG	TIME
500	4.306
600	4.632
700	4.929
800	5.112
900	5.593
1000	5.761

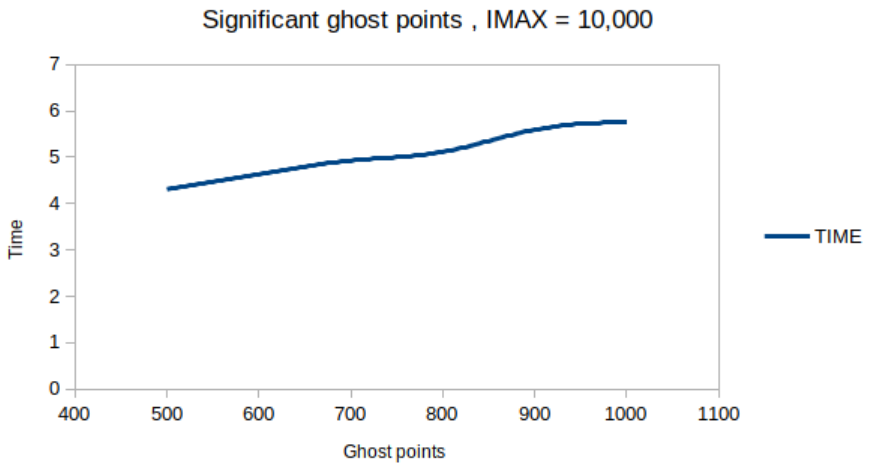

Figure 13: Significant ghost points for 10,000 nodes

We do not see any improvements. The rationale here is that if a significant number of ghost points is used, then the code does not have to send/recv as much. Although this is true, the problem is in how this code is constructed. This prevents any speedup and will be discussed later on.

5.2 MPI headers

It is interesting to note that the MPI header file to be used in Fortran, "mpif.h", is in fact said to be obsolette. When running with the officially supported MPI header, we have better results in terms of scalability. These are shown in Figure 10 and 11. Unfortunately, my laptop does not have more than 4 cores so we really can't make better conclusions regarding the MPI headers.

5.3 Remarks

- Firstly, the code runs in parallel and we do see speedup; although these are not significant
- The ghost points optimization works but we need to do some more experiment to choose that sweet spot between problem size and number of ghost points. As of right now, there are no significant improvements.

- Using more significant ghost points with bigger problem size seems to tbe the correct thinking; however, the current code advects the conservation of mass,momentum and energy together as 1 global vector U. This is costly in terms of send/recv because we are actually sending and receiving a matrix of dimension 3 by number of ghost points. A better way to implement would be to advect the different conservation laws individually. This would help when doing send/recv but we would need to figure out how compute the fluxes for mass, momentum and energy.

6 REFERENCES

1. MTH/CSE 4082 Intro to Parallel Process lecture notes. Dr. Jones Spring 2019. Florida Institute of Technology

2. Chimrac CFD. Web April 2019. `http://chimeracfd.com/programming/gryphon/fluxvanleer.html`

3. Analytical Shocktube Python code. Web April 2019. `https://github.com/ibackus/sod-shocktube`

4. Analytical Shocktube original source (referenced by the Github above). Web April 2019. Dr. Timmes. University of Arizona. `http://cococubed.asu.edu/code_pages/exact_riemann.shtml`

7 CODES

7.1 Analytical Python Solver

Note: The code below was taken to show the accuracy of the implementation. The author's work was cited in the the previous section. These are not my own codes.

```
import numpy as np
import scipy
import scipy.optimize


def sound_speed(gamma, pressure, density, dustFrac=0.):
    """
    Calculate sound speed, scaled by the dust fraction according to:

        .. math::
            \widetilde{c}_s = c_s \sqrt{1 - \epsilon}

    Where :math:`\epsilon` is the dustFrac
    """
    scale = np.sqrt(1 - dustFrac)
    return np.sqrt(gamma * pressure/ density) * scale

def shock_tube_function(p4, p1, p5, rho1, rho5, gamma, dustFrac=0.):
    """
    Shock tube equation
    """
    z = (p4 / p5 - 1.)
    c1 = sound_speed(gamma, p1, rho1, dustFrac)
    c5 = sound_speed(gamma, p5, rho5, dustFrac)

    gm1 = gamma - 1.
    gp1 = gamma + 1.
    g2 = 2. * gamma

    fact = gm1 / g2 * (c5 / c1) * z / np.sqrt(1. + gp1 / g2 * z)
    fact = (1. - fact) ** (g2 / gm1)

    return p1 * fact - p4


def calculate_regions(pl, ul, rhol, pr, ur, rhor, gamma=1.4, dustFrac=0.):
    """
    Compute regions
    :rtype : tuple
    :return: returns p, rho and u for regions 1,3,4,5 as well as the shock speed
    """
    # if pl > pr...
    rho1 = rhol
    p1 = pl
    u1 = ul
    rho5 = rhor
    p5 = pr
    u5 = ur

    # unless...
```

```
    if pl < pr:
        rho1 = rhor
        p1 = pr
        u1 = ur
        rho5 = rhol
        p5 = pl
        u5 = ul

    # solve for post-shock pressure
    p4 = scipy.optimize.fsolve(shock_tube_function, p1, (p1, p5, rho1, rho5, gamma))[0]

    # compute post-shock density and velocity
    z = (p4 / p5 - 1.)
    c5 = sound_speed(gamma, p5, rho5, dustFrac)

    gm1 = gamma - 1.
    gp1 = gamma + 1.
    gmfac1 = 0.5 * gm1 / gamma
    gmfac2 = 0.5 * gp1 / gamma

    fact = np.sqrt(1. + gmfac2 * z)

    u4 = c5 * z / (gamma * fact)
    rho4 = rho5 * (1. + gmfac2 * z) / (1. + gmfac1 * z)

    # shock speed
    w = c5 * fact

    # compute values at foot of rarefaction
    p3 = p4
    u3 = u4
    rho3 = rho1 * (p3 / p1)**(1. / gamma)
    return (p1, rho1, u1), (p3, rho3, u3), (p4, rho4, u4), (p5, rho5, u5), w


def calc_positions(pl, pr, region1, region3, w, xi, t, gamma, dustFrac=0.):
    """
    :return: tuple of positions in the following order ->
            Head of Rarefaction: xhd,  Foot of Rarefaction: xft,
            Contact Discontinuity: xcd, Shock: xsh
    """
    p1, rho1 = region1[:2]  # don't need velocity
    p3, rho3, u3 = region3
    c1 = sound_speed(gamma, p1, rho1, dustFrac)
    c3 = sound_speed(gamma, p3, rho3, dustFrac)

    if pl > pr:
        xsh = xi + w * t
        xcd = xi + u3 * t
        xft = xi + (u3 - c3) * t
        xhd = xi - c1 * t
    else:
        # pr > pl
        xsh = xi - w * t
        xcd = xi - u3 * t
        xft = xi - (u3 - c3) * t
        xhd = xi + c1 * t

    return xhd, xft, xcd, xsh
```

```
def region_states(pl, pr, region1, region3, region4, region5):
    """
    :return: dictionary (region no.: p, rho, u), except for rarefaction region
    where the value is a string, obviously
    """
    if pl > pr:
        return {'Region 1': region1,
                'Region 2': 'RAREFACTION',
                'Region 3': region3,
                'Region 4': region4,
                'Region 5': region5}
    else:
        return {'Region 1': region5,
                'Region 2': region4,
                'Region 3': region3,
                'Region 4': 'RAREFACTION',
                'Region 5': region1}


def create_arrays(pl, pr, xl, xr, positions, state1, state3, state4, state5,
                  npts, gamma, t, xi, dustFrac=0.):
    """
    :return: tuple of x, p, rho and u values across the domain of interest
    """
    xhd, xft, xcd, xsh = positions
    p1, rho1, u1 = state1
    p3, rho3, u3 = state3
    p4, rho4, u4 = state4
    p5, rho5, u5 = state5
    gm1 = gamma - 1.
    gp1 = gamma + 1.

    x_arr = np.linspace(xl, xr, npts)
    rho = np.zeros(npts, dtype=float)
    p = np.zeros(npts, dtype=float)
    u = np.zeros(npts, dtype=float)
    c1 = sound_speed(gamma, p1, rho1, dustFrac)
    if pl > pr:
        for i, x in enumerate(x_arr):
            if x < xhd:
                rho[i] = rho1
                p[i] = p1
                u[i] = u1
            elif x < xft:
                u[i] = 2. / gp1 * (c1 + (x - xi) / t)
                fact = 1. - 0.5 * gm1 * u[i] / c1
                rho[i] = rho1 * fact ** (2. / gm1)
                p[i] = p1 * fact ** (2. * gamma / gm1)
            elif x < xcd:
                rho[i] = rho3
                p[i] = p3
                u[i] = u3
            elif x < xsh:
                rho[i] = rho4
                p[i] = p4
                u[i] = u4
            else:
```

```
                rho[i] = rho5
                p[i] = p5
                u[i] = u5
    else:
        for i, x in enumerate(x_arr):
            if x < xsh:
                rho[i] = rho5
                p[i] = p5
                u[i] = -u1
            elif x < xcd:
                rho[i] = rho4
                p[i] = p4
                u[i] = -u4
            elif x < xft:
                rho[i] = rho3
                p[i] = p3
                u[i] = -u3
            elif x < xhd:
                u[i] = -2. / gp1 * (c1 + (xi - x) / t)
                fact = 1. + 0.5 * gm1 * u[i] / c1
                rho[i] = rho1 * fact ** (2. / gm1)
                p[i] = p1 * fact ** (2. * gamma / gm1)
            else:
                rho[i] = rho1
                p[i] = p1
                u[i] = -u1

    return x_arr, p, rho, u


def solve(left_state, right_state, geometry, t, gamma=1.4, npts=500,
          dustFrac=0.):
    """
    Solves the Sod shock tube problem (i.e. riemann problem) of discontinuity
    across an interface.

    Parameters
    ----------
    left_state, right_state: tuple
        A tuple of the state (pressure, density, velocity) on each side of the
        shocktube barrier for the ICs.  In the case of a dusty-gas, the density
        should be the gas density.
    geometry: tuple
        A tuple of positions for (left boundary, right boundary, barrier)
    t: float
        Time to calculate the solution at
    gamma: float
        Adiabatic index for the gas.
    npts: int
        number of points for array of pressure, density and velocity
    dustFrac: float
        Uniform fraction for the gas, between 0 and 1.

    Returns
    -------
    positions: dict
        Locations of the important places (rarefaction wave, shock, etc...)
    regions: dict
        constant pressure, density and velocity states in distinct regions
```

```
    values: dict
        Arrays of pressure, density, and velocity as a function of position.
        The density ('rho') is the gas density, which may differ from the
        total density in a dusty-gas.
        Also calculates the specific internal energy
    """

    pl, rhol, ul = left_state
    pr, rhor, ur = right_state
    xl, xr, xi = geometry

    # basic checking
    if xl >= xr:
        print('xl has to be less than xr!')
        exit()
    if xi >= xr or xi <= xl:
        print('xi has in between xl and xr!')
        exit()

    # calculate regions
    region1, region3, region4, region5, w = \
        calculate_regions(pl, ul, rhol, pr, ur, rhor, gamma, dustFrac)

    regions = region_states(pl, pr, region1, region3, region4, region5)

    # calculate positions
    x_positions = calc_positions(pl, pr, region1, region3, w, xi, t, gamma,
                                 dustFrac)

    pos_description = ('Head of Rarefaction', 'Foot of Rarefaction',
                       'Contact Discontinuity', 'Shock')
    positions = dict(zip(pos_description, x_positions))

    # create arrays
    x, p, rho, u = create_arrays(pl, pr, xl, xr, x_positions,
                                 region1, region3, region4, region5,
                                 npts, gamma, t, xi, dustFrac)

    energy = p/(rho * (gamma - 1.0))
    rho_total = rho/(1.0 - dustFrac)
    val_dict = {'x':x, 'p':p, 'rho':rho, 'u':u, 'energy':energy,
                'rho_total':rho_total}

    return positions, regions, val_dict
```

7.2 Python Plotting

I wrote this code on my own to use the analytical solver in previous section and the numerical results from my own implementation

```
import numpy as np
import matplotlib.pyplot as plt
import sod

#LOAD NUMERICAL DATA
data = np.loadtxt('result.txt')
x  = data[:,0]
rho= data[:,1]
u= data[:,2]
p= data[:,3]
mach = data[:,4]
figwidth = 15
figheight = 10

#LOAD ANALYTICAL DATA
gamma = 1.4
npts = 1000
pleft = 1./(gamma-1)
pright = 0.125/(gamma-1.)
positions, regions, values = sod.solve(left_state=(pleft, 1, 0),
        right_state=(pright, 0.125, 0.),
        geometry=(-10., 10., 0.0), t=2.00, gamma=gamma, npts=npts)
p_ana = values['p']
rho_ana = values['rho']
u_ana = values['u']
speedsound_ana = np.sqrt((gamma*p_ana)/(rho_ana))
mach_ana = u_ana/speedsound_ana

#PLOTTING
plt.subplots(figsize=(figwidth,figheight))
plt.subplot(4,1,1)
plt.title("DENSITY",fontsize=20)
plt.plot(x,rho,'-xb',label='Numerical')
plt.plot(x,rho_ana,'-r',label='Analytical')
plt.xticks([], [])
plt.yticks(fontsize=14)
plt.legend()
plt.subplot(4,1,2)
plt.title("VELOCITY",fontsize=20)
plt.plot(x,u,'-xb',label='Numerical')
plt.plot(x,u_ana,'-r',label='Analytical')
plt.xticks([], [])
plt.yticks(fontsize=14)
plt.legend()
plt.subplot(4,1,3)
plt.title("PRESSURE",fontsize=20)
plt.plot(x,p,'-xb',label='Numerical')
plt.plot(x,p_ana,'-r',label='Analytical')
plt.xticks([],[])
```

```
plt.yticks(fontsize=14)
plt.legend()
plt.subplot(4,1,4)
plt.title("MACH",fontsize=20)
plt.plot(x,mach,'-xb',label='Numerical')
plt.plot(x,mach_ana,'-r',label='Analytical')
plt.yticks(fontsize=14)
plt.xticks(fontsize=14)
plt.legend()
plt.savefig("shock.png")
plt.show()
```

7.3 1D Shocktube MPI Implementation

Below is the main code for this project. This is my own work. Results are written to a file "result.txt". The python codes in previous sections are used to read and plotted along the analytical solutions

```
!! Name: Max Le
!! Parallel Process Final Project Spring 2019

!! SOLVING 1D EULER
!! GRID AS FOLLOW
!!  |---------- |--------|------------|
!!  |   i-1     |    i   |     i + 1  |
!!  |---------- |--------|------------|
!!             i-1/2     i+1/2

!! ASSUME POSITIVE FLUX TO THE RIGHT, NEGATIVE FLUX TO THE LEFT

!! EACH EDGE (HALF) EXPERIENCES 2 FLUXES: POSITIVE AND NEGATIVE

!! GRID NODE 1 and NMAX are BC.  CALCULATIONS OCCUR 2 to NMAX-2



!!**********************  MODULE FOR ALL VARIABLES **************************** !!

module globalvar

    integer :: IMAX = 10000
    integer :: I,J,N
    real*8  :: dx,dt,cfl = 0.67, gamma = 1.4
    real*8,dimension(:),allocatable :: a

    !! FLUX VECTORS
    real*8,dimension(3):: fiph, fimh


    real*8,dimension(:,:),allocatable :: fGLOBALp, fGLOBALm

    !! STATE VECTOR
    real*8,dimension(:,:),allocatable :: state_vector


    !! Mach vector
    real*8,dimension(:),allocatable   :: mach_array

    !! OTHERS
    real*8 :: t, tmax , ai
    real*8, dimension(:,:,:),allocatable :: U
    real*8,dimension(:),allocatable :: x_arr_local
    real*8 :: xstart, xend, contactpoint

    integer :: iTrue, iStart!! real index
```

```
    !! MPI VARIABLES

    integer :: rank, nprocs, ierr
    integer :: nblocks, rem
    real*8 :: min_dt, global_dt
    integer :: leftREQUEST, rightREQUEST, tempREQUEST

    integer :: sendLEFT,sendRIGHT, recvLEFT, recvRIGHT

    !! LOCAL VAR
    integer :: localLOW, localHIGH
    integer :: ghostPoints = 2
    integer :: maxLocal, minLocal



    !! SEND/RECV MATRIX
    DOUBLE PRECISION,DIMENSION(:,:),ALLOCATABLE :: s1mat, r1mat, s2mat, r2mat

    !! FLAG

    integer :: flagdum,npointsdum

    !*********************  INTERFACE   ************************!
    interface
        subroutine vanleer(fp, fm, w_arr, Nsize, g)
            implicit none
            !! INTENT VARIABLES
            real*8,intent(in),DIMENSION(:,:), ALLOCATABLE :: w_arr
            real*8,intent(in) :: g
            integer,intent(in) :: Nsize
            real*8,intent(out),DIMENSION(3):: fp, fm
        end subroutine
    end interface


end module globalvar



!!***********************  END MODULE  ************************************** !!
```

```
!!*************************************************************************** !!
!!**********************  MAIN PROGRAM ************************************** !!
!!*************************************************************************** !!

program SHOCK
    use globalvar
    include 'mpif.h'


    !!****************** MPI START  *************************** !
    call MPI_INIT(ierr)
    call MPI_COMM_SIZE(MPI_COMM_WORLD,nprocs,ierr)
    call MPI_COMM_RANK(MPI_COMM_WORLD,rank,ierr)

    !************* INITIALIZING *************************** !
    PRINT*, "INITIALIZING"
    tmax = 2.00
    xstart = -10.00
    xend = 10.00
    contactpoint = 0.00
    dx = (xend-xstart)/DBLE((IMAX-1))



    print*, "SETTING NBLOCKS, ILOW, IHIGH"

    nblocks = FLOOR(REAL(IMAX/nprocs))

    rem = IMAX - nblocks*nprocs


    !! starting REAL index on each chunk
    iStart = (nblocks*rank) + ghostPoints

    localLOW = 1+ghostPoints
    localHIGH = nblocks+ghostPoints
    maxLocal = localHIGH + ghostPoints
    minLocal = localLOW-ghostPoints

    ! adding the remaining point
    if(rem /= 0) then
        if(rank == nprocs-1) then
            localHIGH = localHIGH + 1
            maxlocal  = maxlocal + 1
        end if
    end if


123 FORMAT("I am rank: ",i2," and ILO=",i4," and IHI=",i4,&
    " minlocal=", &i4, " maxlocal=",i4)
    WRITE(*,123) rank,localLOW,localHIGH,minlocal, maxlocal


    print*, "FINISHED SETTING NBLOCKS"
```

```
print*, "ALLOCATING LOCAL ARRAYS"
ALLOCATE(a(maxLocal))

ALLOCATE(fGLOBALm(3,maxLocal))

ALLOCATE(fGLOBALp(3,maxLocal))

ALLOCATE(U(3,maxLocal,2))

ALLOCATE(state_vector(3,maxLocal))

ALLOCATE(x_arr_local(IMAX+2*ghostPoints))

ALLOCATE(mach_array(maxLocal))

ALLOCATE(s1mat(3,ghostPoints))
ALLOCATE(s2mat(3,ghostPoints))
ALLOCATE(r1mat(3,ghostPoints))
ALLOCATE(r2mat(3,ghostPoints))

print*, "SETTING INITIAL CONDITIONS"



do i = 1,IMAX+2*ghostPoints
    x_arr_local(i) = xstart + (i-1-ghostPoints)*dx
end do



!! SETTING INITIAL RHO, U, P
do i = minLocal , maxLocal
    iTrue = i+iStart- ghostPoints

    if (x_arr_local(iTrue) .LE. contactpoint) then
        state_vector(1,i) = 1.00                    !! RHO LEFT
        state_vector(2,i) = 0.0                     !! U LEFT
        state_vector(3,i) = 1.00/(gamma-1)          !! P LEFT
    else
        state_vector(1,i) = 0.125                   !! RHO RIGHT
        state_vector(2,i) = 0.0                     !! U RIGHT
        state_vector(3,i) = 0.125/(gamma-1)         !! P RIGHT
    end if
end do


!! AT TIME = 0
do i = minLocal,maxLocal
    U(1,i,1) = state_vector(1,i)
    U(2,i,1) = state_vector(1,i)*state_vector(2,i)
    U(3,i,1) = state_vector(3,i)/(gamma-1)  &
        + 0.5*state_vector(1,i)*state_vector(2,i)*state_vector(2,i)
```

```
        a(i) = sqrt((gamma*state_vector(3,i))/state_vector(1,i))
        mach_array(i) = state_vector(2,i)/(a(i))
    end do


    flagdum = 0
    print*, "FINISHED INITIALIZING"



    PRINT*, "SOLVING"
    t = 0
    do  while (t .LE. tmax)

            !!! GET GLOBAL DT
            min_dt = 100

            do i = localLOW, localHIGH

                ai = sqrt(gamma*state_vector(3,i)/state_vector(1,i))
                dt = cfl*dx/(abs(state_vector(2,i)) + ai)

                if (dt < min_dt) then
                    min_dt = dt
                end if

            end do

            call MPI_ALLREDUCE(min_dt, global_dt, 1, MPI_DOUBLE_PRECISION, &
                MPI_MIN, MPI_COMM_WORLD,ierr)



            !!! CALCULATE FLUX AND UPDATE U
            call update(flagdum)

            !! UPDATE U
            do i = localLOW-ghostPoints,localHIGH+ghostPoints
                do j = 1,3
                    U(j,i,1) = U(j,i,2)
                end do
            end do


            !!PULL VARIABLES
            do i = minLocal , maxLocal
                 ! PULL RHO
                state_vector(1,i)   = U(1,i,2)
                ! PULL U = RHOU / RHO
                state_vector(2,i)   = U(2,i,2)/U(1,i,2)
                ! PULL PRESSURE
                state_vector(3,i)   = (U(3,i,2)-0.5*U(1,i,2)*&
                    (state_vector(2,i)**2))*(gamma-1)
                ! UPDATE SOUND SPEED
```

```
                a(i)     = sqrt((gamma*state_vector(3,i))/(state_vector(1,i)))
            end do

            !! UPDATE TIME STEP
            t = t + global_dt
            print*, "t = ", t

    end do

    print*, "FINISHED SOLVING"

    call output

    call MPI_FINALIZE(ierr)
    print*, "DEALLOCATE ARRAYS "


    !!!! DEALLOCATE ARRAY
    DEALLOCATE(a)
    DEALLOCATE(fGLOBALm)
    DEALLOCATE(fGLOBALp)
    DEALLOCATE(U)
    DEALLOCATE(state_vector)
    DEALLOCATE(x_arr_local)
    DEALLOCATE(mach_array)
    DEALLOCATE(s1mat)
    DEALLOCATE(s2mat)
    DEALLOCATE(r1mat)
    DEALLOCATE(r2mat)



end program SHOCK



!!************************************************************************** !!
!!**********************  FINISHED MAIN PROGRAM ***************************** !!
!!************************************************************************** !!
```

```
!!**********************  SUBROUTINE FOR FLUX  ****************************** !!


subroutine vanleer(fp, fm, w_arr,Nsize, g)

    implicit none


    !! INTENT VARIABLES
    real*8,intent(in),DIMENSION(:,:), ALLOCATABLE :: w_arr
    real*8,intent(in) :: g
    integer,intent(in) :: Nsize
    real*8,intent(out),DIMENSION(3) :: fp, fm



    !! LOCAL VARIABLES
    integer :: i
    real*8 :: machNumber
    real*8 :: faplus = 0.0, fbplus = 0.0, faminus = 0.0, fbminus = 0.0

    real*8 :: ai, Ei

    i = Nsize
    !! Calculate ai, pi

    ai = sqrt((g*w_arr(3,i))/(w_arr(1,i)))

    !! Build variables
    machNumber = w_arr(2,i)/ai                          !! mach = u/ai

    if (abs(machNumber) .LE. 1) then
        faplus = 0.25*w_arr(1,i)*ai*(machNumber+1)**2
        fbplus = (g-1.0)*machNumber*ai + 2*ai

        faminus = -0.25*w_arr(1,i)*ai*(machNumber-1)**2
        fbminus = -(g-1.0)*machNumber*ai - 2*ai

        !! Building up Van Leer flux vector
        fp(1) = faplus
        fp(2) = faplus*fbplus*(1.0/g)
        fp(3) = faplus*fbplus**2*(1.0/(2.0*(g**2-1)))


        fm(1) = faminus
        fm(2) = faminus*fbminus*(1.0/g)
        fm(3) = faminus*fbminus**2*(1.0/(2.0*(g**2-1)))

    else
        Ei = w_arr(3,i)/(g-1.0) + 0.5*w_arr(1,i)*w_arr(2,i)**2
        if (w_arr(2,i) > 0) then
            fp(1) = w_arr(1,i)*w_arr(2,i)
            fp(2) = w_arr(1,i)*w_arr(2,i)**2 + w_arr(3,i)
            fp(3) = (Ei+w_arr(3,i))*w_arr(2,i)
```

```
            fm(1) = 0.0
            fm(2) = 0.0
            fm(3) = 0.0


        else

            fm(1) = w_arr(1,i)*w_arr(2,i)
            fm(2) = w_arr(1,i)*w_arr(2,i)**2 + w_arr(3,i)
            fm(3) = (Ei+w_arr(3,i))*w_arr(2,i)

            fp(1) = 0.0
            fp(2) = 0.0
            fp(3) = 0.0

        end if
         !! Reset to zero for next run
        machNumber = 0.0
        faplus = 0.0
        fbplus = 0.0
        faminus = 0.0
        fbminus = 0.0
    end if
end subroutine



!!**********************  END SUBROUTINE FOR FLUX  ************************** !!
```

```
!!*********************  SUBROUTINE FOR SEND/RECV ************************** !!


subroutine bc_recv

    use globalvar
    include 'mpif.h'

    if (rank > 0) then
        call MPI_Wait(sendLEFT,MPI_STATUS_IGNORE,ierr)
        call MPI_WAIT(recvLEFT,MPI_STATUS_IGNORE,ierr)
    end if


    if (rank < nprocs-1) then
        call MPI_Wait(sendRIGHT,MPI_STATUS_IGNORE,ierr)
        call MPI_Wait(recvRIGHT,MPI_STATUS_IGNORE,ierr)

    end if

    !! SETUP RECV MATRIX
    do j = 1,ghostPoints
        U(:,localLOW-(ghostPoints-j+1),2) = r1mat(:,j)
        U(:,localHIGH+j,2) = r2mat(:,j)

    end do

end subroutine bc_recv

subroutine bc_send
    use globalvar
    include 'mpif.h'


    !! SEND/RECV
    if (rank > 0 ) then
        call MPI_Irecv(r1mat,ghostPoints*3,MPI_DOUBLE_PRECISION,rank-1,0,&
            MPI_COMM_WORLD,recvLEFT,ierr)

    end if


    if (rank <nprocs-1) then
        call MPI_Irecv(r2mat,ghostPoints*3,MPI_DOUBLE_PRECISION,rank+1,1,&
            MPI_COMM_WORLD,recvRIGHT,ierr)

    end if


    !! SETUP SEND MATRIX
    do j = 1,ghostPoints
        s1mat(:,j) = U(:,localHIGH-(ghostPoints-j),2)
        s2mat(:,j) = U(:,localLOW+(j-1),2)
```

```
    end do



    if (rank > 0 ) then
        call MPI_Isend(s2mat,ghostPoints*3,MPI_DOUBLE_PRECISION,rank-1,1,&
            MPI_COMM_WORLD,sendLEFT,ierr)
    end if


    if (rank <nprocs-1) then
        call MPI_Isend(s1mat,ghostPoints*3,MPI_DOUBLE_PRECISION,rank+1,0,&
            MPI_COMM_WORLD,sendRIGHT,ierr)
    end if



end subroutine bc_send




!!**********************  END SUBROUTINE FOR SEND/RECV ********************** !!
```

```
!!*********************  SUBROUTINE FOR OUTPUT  ***************************** !!


subroutine output
    use globalvar
    include 'mpif.h'


    print*, "WRITE TO FILE"

    if (nprocs > 1) then
        if (rank == 0) then
            open(unit=  42, file = 'result.txt', action = 'write')
            do i = localLOW,localHIGH
                iTrue = i+iStart- ghostPoints
                write(42,*) x_arr_local(iTrue), state_vector(1,i), &
                state_vector(2,i), state_vector(3,i),state_vector(2,i)/a(i)
            end do
            close(42)
        end if

        call MPI_BARRIER(MPI_COMM_WORLD,ierr)

        if (nprocs > 2) then
            do N = 1,nprocs-2
                if (rank == N) then
                    open(unit=  42, file = 'result.txt', status='old',&
                        position='append',action = 'write')
                    do i = localLOW,localHIGH
                        iTrue = i+iStart- ghostPoints
                        write(42,*) x_arr_local(iTrue), state_vector(1,i), &
                            state_vector(2,i), state_vector(3,i),&
                            state_vector(2,i)/a(i)
                    end do
                    close(42)
                end if
                call MPI_BARRIER(MPI_COMM_WORLD,ierr)
            end do
        end if

        if (rank == nprocs-1) then
            open(unit=  42, file = 'result.txt', status='old',&
                position='append',action = 'write')
            do i = localLOW,localHIGH
                iTrue = i+iStart- ghostPoints
                write(42,*) x_arr_local(iTrue), state_vector(1,i), &
                    state_vector(2,i), state_vector(3,i),state_vector(2,i)/a(i)
            end do
            close(42)
        end if
        call MPI_BARRIER(MPI_COMM_WORLD,ierr)


    else  !! FOR 1 core case
```

```
        open(unit=  42, file = 'result.txt', action = 'write')
        do i = localLOW,localHIGH
            write(42,*) x_arr_local(i), state_vector(1,i), state_vector(2,i), &
                state_vector(3,i),state_vector(2,i)/a(i)
        end do
        close(42)
    end if

end subroutine output



!!**********************  END SUBROUTINE FOR OUTPUT  ************************** !!
```

```
!!**********************  SUBROUTINE FOR UPDATE  **************************** !!


subroutine update(temp)
    use globalvar
    include 'mpif.h'

    integer, intent(inout) :: temp


    !! FLUX CALCULATIONS
    do i = minLocal, maxLocal
        call vanleer(fiph, fimh, state_vector, i, gamma)
        fGLOBALm(:,i) = fimh(:)
        fGLOBALp(:,i) = fiph(:)
    end do


    ! FINITE DIFFERENCE
    do i = localLOW,localHIGH
        do j=1,3
            U(j,i,2) = U(j,i,1)- (global_dt/dx)*(fGLOBALp(j,i)-fGLOBALp(j,i-1))- &
                (global_dt/dx)*(fGLOBALm(j,i+1)-fGLOBALm(j,i))
        end do
    end do


    if (temp == 0) then
        print*, "DOING SEND/RECV"
        call bc_send
        call bc_recv
        temp = ghostPoints
    end if

    temp = temp -1
    call boundary

end subroutine update


!!**********************  END SUBROUTINE FOR UPDATE  ************************** !!
```

```
!!**********************  SUBROUTINE FOR BOUNDARY  ************************** !!

subroutine boundary
    use globalvar
    include 'mpif.h'

    !!! BOUNDARY CONDITIONS
    if (rank == 0) then
        do j = 1,ghostPoints
            U(:,localLOW-j,2) = U(:,localLOW-j+1,2)
        end do
    end if

    if (rank == nprocs-1) then
        do j = 1,ghostPoints
            U(:,localHIGH+j,2) = U(:,localHIGH+j-1,2)
        end do
    end if
end subroutine boundary

!!**********************  END SUBROUTINE FOR BOUNDARY  ********************** !!
```