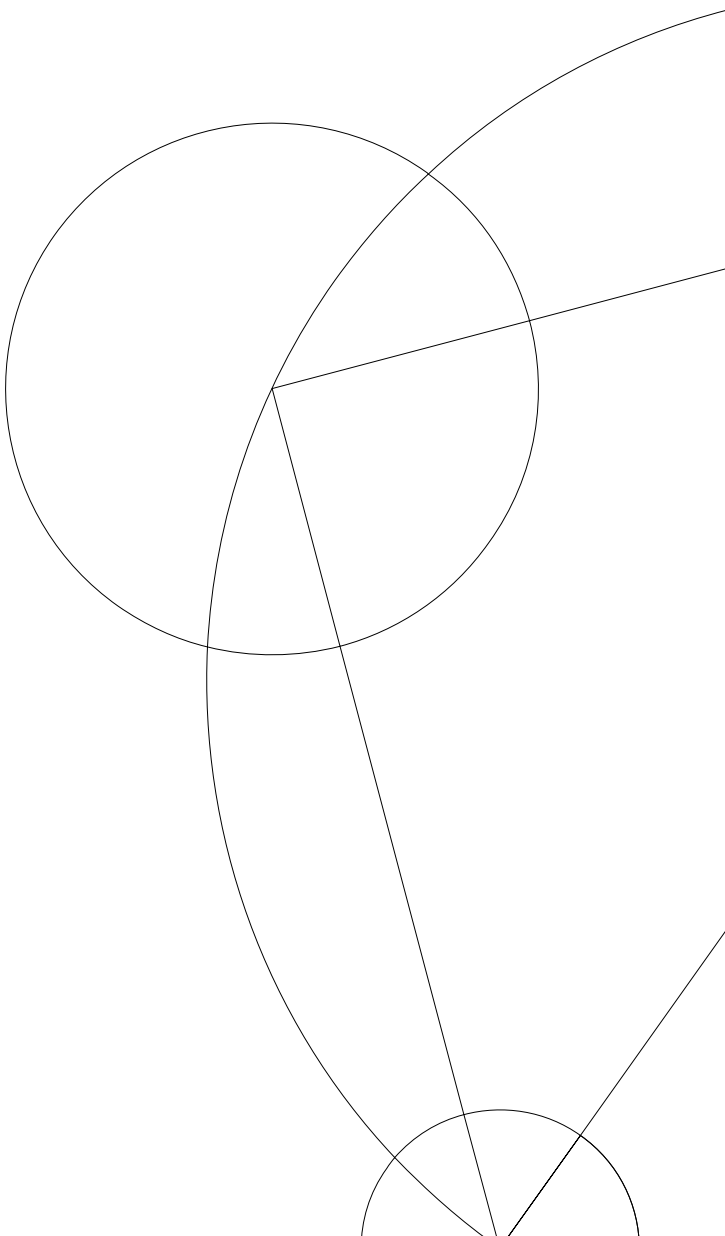
KØBENHAVNS UNIVERSITET

Computational Astrophysics

Laurent Lindpointner

Distributed Parallel Computing with Python

Exam project



Instructor: Troels Haugbølle

Faculty of Science, January 22, 2020

Contents

1 Introduction

In this project I took the code form assignment '2b Blast Waves' and parallelised it using OpenMPI and mpi4py. The advantage of a parallel implementation lies in its superior performance when compared to a sequential implementation. This report will give some background on distributed parallel computing, show the most important parts of my implementation and results as well as performance measures of the code.

2 Theory

2.1 Distributed Parallel Computing

Parallel computing is the use of multiple processors to do a task simultaneously. A single script is run by all participating processes and using a process' identification, the flow of the program can be tailored for each process. Parallel computing can be done either using CPUs, each of which might have multiple cores, on one compute node with a shared memory (like a personal computer), or using CPUs which are located in different nodes, having no shared memory. The latter case is called distributed parallel programming. In the case of shared memory, communication can be done e.g. through different processes saving and reading files from one another. In distributed programming this is not possible and another form of communication between processes, called Message Passing is used. [1]
In this project, the Message Passing Interface (MPI) is used which is a library of functions which allow passing of messages between processes without shared memory. All processes involved belong to communicator objects. These communicators can either hold all of the involved processes or a subset. The processes within a communicator have a rank by which they can identify each other. MPI supports both point-to-point and collective communication.

2.1.1 Point-to-point Communication

In point-to-point communication, one process sends data to another process. In the MPI.Send(data, destination) routine, data is sent to a specified destination. The MPI.Receive(data, source) routine has to be called by the targeted process to receive the sent information. Point-to-point functions can be either blocking or non-blocking. In the blocking version, both processes wait until the sending and receiving, respectively, is completed. In non-blocking functions, requests are sent for sending and receiving data which can be fulfilled at a later time while the process continues running the code that comes after the function call. [1]

2.1.2 Collective Communication

In collective communication, data is exchanged between all the communicator's processes. The MPI.Bcast(data, root) routine, for example, is used for sending data from a root process to all other processes. All processes in the communicator must call this function, only the specified root process, however, is sending data while all other processes are receiving it. Other important routines include MPI.Scatter(sendbuffer, recvbuffer, root), which is used to divide data evenly from the root, the sendbuffer, over all other processes and stores it in the specified recvbuffer, and MPI.Gather(sendbuffer, recvbuffer, root) which collects sendbuffers from all processes and stores them at the root process in the specified recvbuffer. [1]
Collective communication routines are always called by all the communicator's processes while point-to-point communication routines are only called by the processes involved.

2.2 Virtual Boundaries

When hydrodynamics is done in parallel computing, the whole mesh is divided into segments, each of which is calculated in one process. This means that there is a need for communication between the sub-meshes when stepping forward in time as fluid variables change across the mesh. A good way to do this is the use of virtual boundaries, or ghost zones. A scheme of virtual boundaries is shown in figure 1. The different processes know their location within the global mesh through their rank in the communicator. In figure 1, the blue zones show the processes' 'native' mesh, while the red zones around them show the processes' ghost zones. After one time-step, process 0 for example sends its right boundary values (blue area enclosed by white line along one side) to process 1 which is located

to the left in the global mesh. Process 1, in return, sends its left boundary values to process 0. The same is done for the other sides with boundaries which also are global boundaries, sending their global boundary to the process which is located at the opposite side of the global mesh. This is shown in the figure for process 1, sending its right boundary to process 0.
These boundary exchanges between processes are done using point-to-point communication.

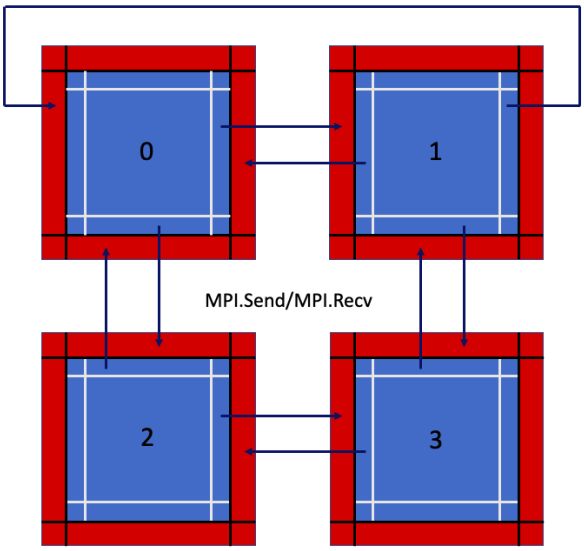


Figure 1: Ghost zones are shown in red, local meshes in blue. Boundary values are communicated using point-to-point communication.

3 Implementation

The general purpose of this program is to use hydrodynamics to evolve a blast-wave setup in time. Following, is a brief description of the most important sections of the code of the program. Especially the parts, where the parallelization comes into play. The code of each section can be found in the appendix.
It is important to note, that the code is built in such a way, that only numbers of processes can be used which have as a square root an even number (i.e. 4, 9, 16, 25,...). The code can then be run from the terminal using for example: ”mpirun -n 9 python parallel_hydrodynamics.py 300 300 2” where the last three numbers represent resolution, number of iterations and number of rows/columns in the ghost zone.

3.1 Localization

The first step is initiating a 'global' blast wave class with the same parameters in all processes. Each process gets a subsection of the whole grid of the class according to its rank. A class 'local' is initiated that takes a part of the whole grid, that corresponds to the process. In this class, all important variables of the experiment are defined such as density, total energy, momentum and coordinates. The ghost zones of the fluid variables are also initiated.

3.2 Globalizing time-step using MPI.Allreduce()

The next step in the code is the time evolution of the local variables. Here, it is essential to use the same time-step dt in all processes in order to ensure a consistent solution. This is achieved with the function MPI.Allreduce(local_dt, global_dt, operation). This function takes a floating point argument from each calling process, compares them and returns, in our case, the smallest of the arguments as a collective time-step to all processes.

3.3 Sending & Receiving of Virtual Boundaries

After stepping forward in time with the muscl_2d() function, all processes need their new virtual boundaries for the next time-step. This is achieved using point-to-point communication between the processes. The reason for using point-to-point rather than collective communication is, that each process knows where to send its boundary elements (as the new virtual boundaries of the receiving process) and therefore there is no need to broadcast the values to all processes. Hence, each process sends e.g. its left boundary elements to the process located to the left in the global grid for each variable individually, and so on for the other 3 sides.

3.4 Gathering Local Segments

The final step, after completing the time evolution, is putting the whole grid together again from the local processes. MPI.gather() is used for this. This function takes as arguments a send buffer from each process, where the local grid of each variable is sent, and a root. Each process calls the function and all processes act as senders of their individual send buffer while the specified root process receives all the data. It is up to the root process then, to put together the complete picture which is stored in the corresponding variables of density, total energy and momentum in the original global blast wave class.

4 Results

All displayed results were obtained using 2 rows/columns in the ghost zone unless stated otherwise.

Figure 2 shows the density and temperature distribution of a basic blast wave setup. It was obtained using 9 cores, a resolution of 700 and 2500 iterations.
Results of more advanced blast-wave setups as well as a link to GIF-animations of the setups can be found in the appendix section.

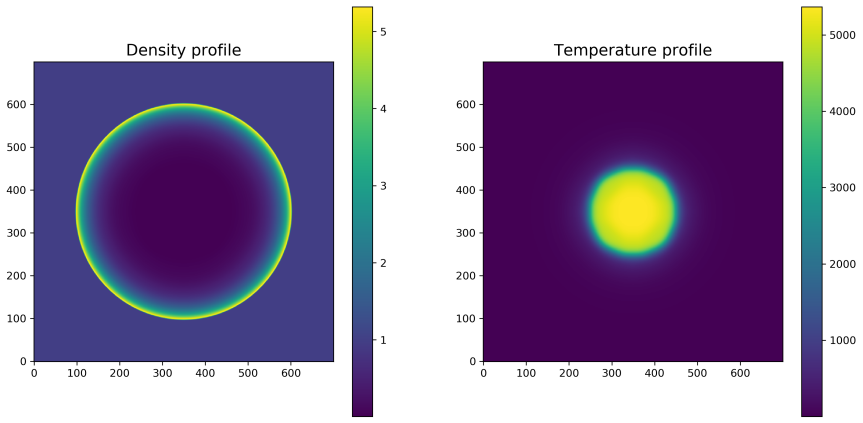


Figure 2: Density and temperature profile of a basic blast-wave setup using d0=1., e0=1e5, C=0.5, w=10., power=4. It was obtained using 9 processes, a resolution of 700 and 2500 iterations.

The result can be examined more closely by comparing it directly to a result obtained with the same parameters with a sequential algorithm. This comparison is shown in figure 3.

Figure 4 shows a performance comparison between the sequential code and the parallel code, run with different numbers of cores but always 1 row/column as ghost zone. The shown numbers are averages of a least 5 runs (more runs for lower resolution) of 300 iterations. These results have to be taken with a grain of salt as they depend strongly on the performance of the 'erda' platform at the time.

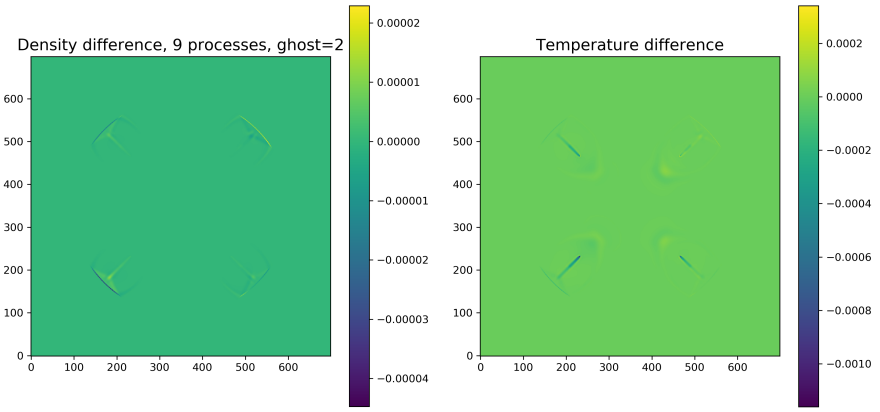


Figure 3: Comparison of basic blast-wave setup between run with sequential code and 9 cores. Using 2 rows/columns as ghost zone in the parallel code (resolution 700, 2500 time-steps, d0=1., e0=1e5, w=10., power=4, eps=0).

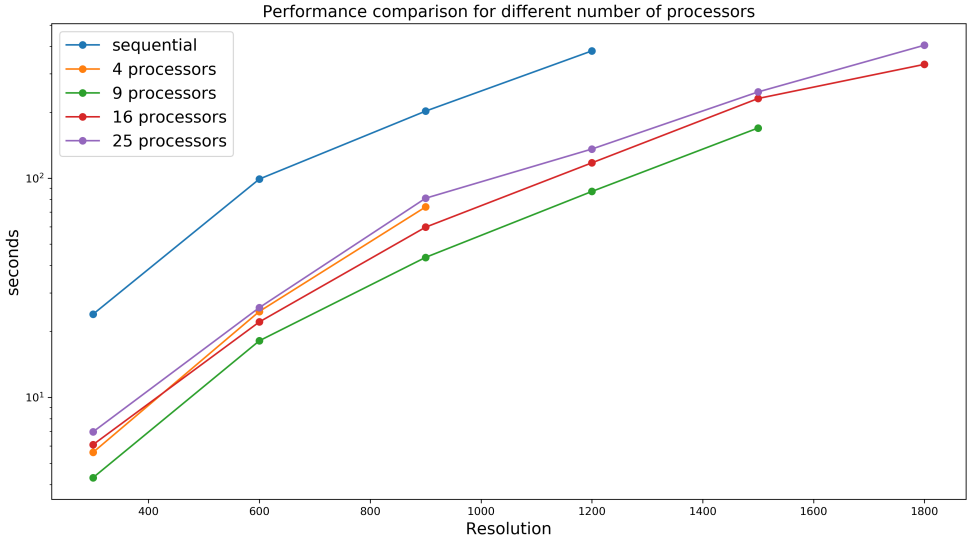


Figure 4: Comparison of performance between sequential and parallel algorithm, run with different numbers of cores, different resolutions and 300 time-steps.

5 Discussion

The result in figure 2 passes the eye test. A smooth time-evolution can be observed, also at the boundaries between the local grids.
The comparison with the sequential code reveals, that it is not enough to take only 1 row/column in the ghost zone as can be seen in figure 5. Especially in the temperature difference it can be seen, that there is a problem with transporting fluid variables across process boundaries to the next process when using only 1 row/column in the ghost zone. The maximum error in the density profile is almost as large as the peak values while the error in the temperature profile is around 10% of the maximum temperature. A much better result can be obtained when using 2 columns/rows in the ghost zone.

The error is reduced to 0.005% and 0.0002% of the maximum values respectively.

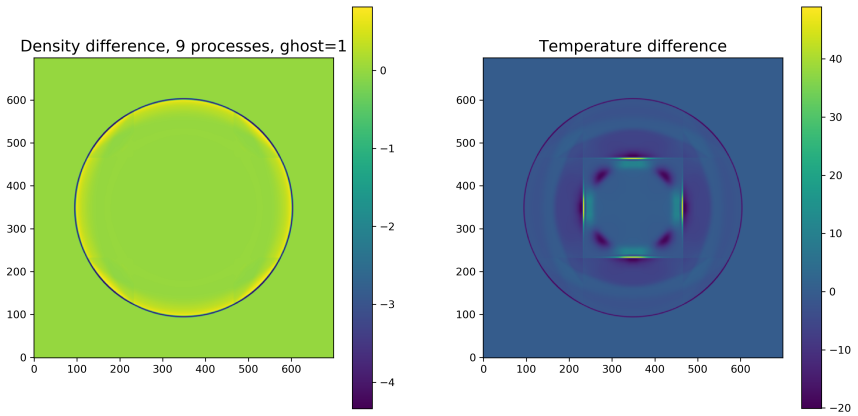


Figure 5: Comparison of basic blast-wave setup between run with sequential code and 9 cores. Using 1 row/column as ghost zone in the parallel code (resolution 700, 2500 time-steps, d0=1., e0=1e5, w=10., power=4, eps=0).

When plotting this for different numbers of processes, it can be observed, that the accuracy of the algorithm also depends on the geometry of the problem. Figures 6 and 7 show the same comparison using 2 rows/columns in the ghost zone for 16 and 24 processes. Dividing the axis into uneven numbers appears to yield much better results.

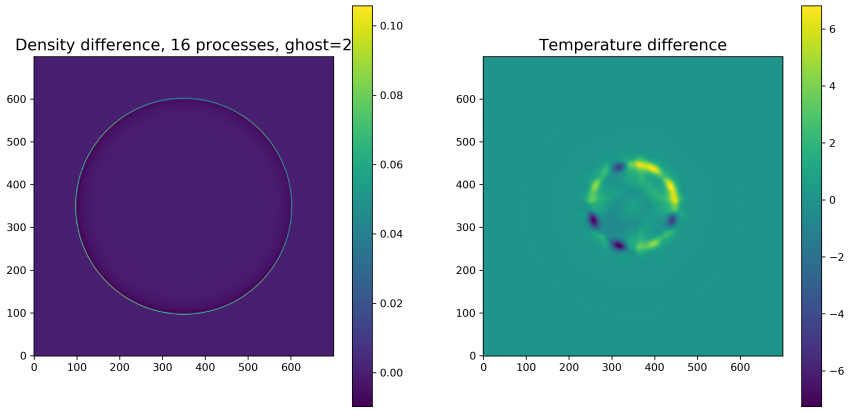


Figure 6: Comparison of basic blast-wave setup between run with sequential code and 16 cores. Using 2 rows/columns as ghost zone in the parallel code (resolution 700, 2500 time-steps, d0=1., e0=1e5, w=10., power=4, eps=0).

One important down side of using point-to-point communication it has a much lower limit of how much data can be sent than collective communication routines. The MPI.gather() command exchanges arrays of up to 500x500 between all processes. MPI.Send() appears to fail upwards of sending 1x500 arrays. sending 1x600 arrays was attempted with different numbers of processes but could not be handled. Seeing that there is a need to exchange 2 rows/columns in the ghost zone, this limit is reached at lower resolutions for each number of processes. Another option of handling the exchange of ghost zones would be to use MPI.scatter(). In a more in-depth project, this would be the next step to

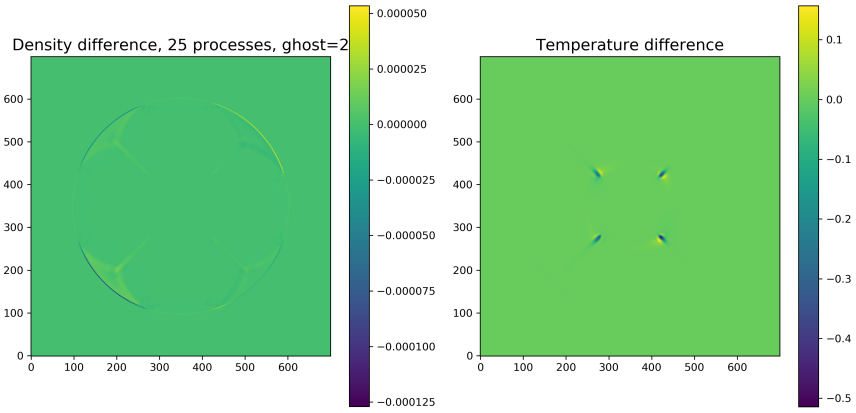


Figure 7: Comparison of basic blast-wave setup between run with sequential code and 25 cores. Using 2 rows/columns as ghost zone in the parallel code (resolution 700, 2500 time-steps, d0=1., e0=1e5, w=10., power=4, eps=0).

take.
Considering that fluid variables also get transported better or worse depending on how the sub-meshes are constructed, a try of different shapes for the sub-meshes might also yield better results.

Finally, 9 processes were found have the best performance across a wide range of resolutions. Time-wise, the muscl_2d() routine was the dominating operation for practically all resolutions which suggests that the blocking nature of the MPI.Send() and MPI.Recv() routines slows the program down when there are more processes and therefore more sending and receiving operations in play. All numbers of processes, however, are much faster than the sequential program. Higher numbers of processes become necessary when increasing the resolution further from the shown values as the MPI.Send() routine puts a relatively low limit on maximum resolution/number of processes used.

6 Conclusion

In conclusion, the code was successfully parallelized but has to be used correctly in order to limit errors and get the best possible performance. Distributed parallel programming is a great tool to speed up smaller programs and a necessary tool when running very large programs. It requires, however, much more caution than sequential programming when it comes to the geometry of the sub-meshes or the selection of an appropriate number of rows/columns in the ghost zones. Performance can likely be increased by using either non-blocking point-to-point routines or collective communication instead.

7 Appendix

7.1 Different Blast-wave setups

The following link contains a Dropbox folder with GIF animations of the four different blast-wave setups shown in this report: Dropbox/GIF animations

Blast-wave with mass and noise

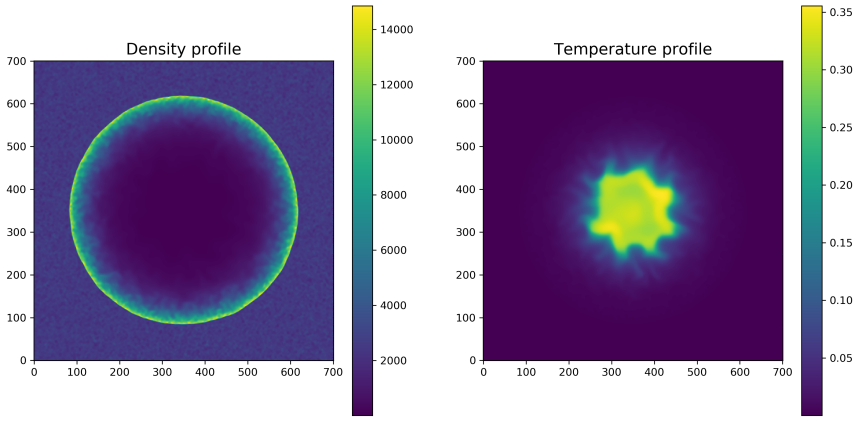


Figure 8: Blast-wave with noise, run with 9 processes. (resolution 700, 1500 time-steps, d0=1e4, e0=1e5, w=6., power=4, eps=0.5)

Blast-wave with Rayleigh instability

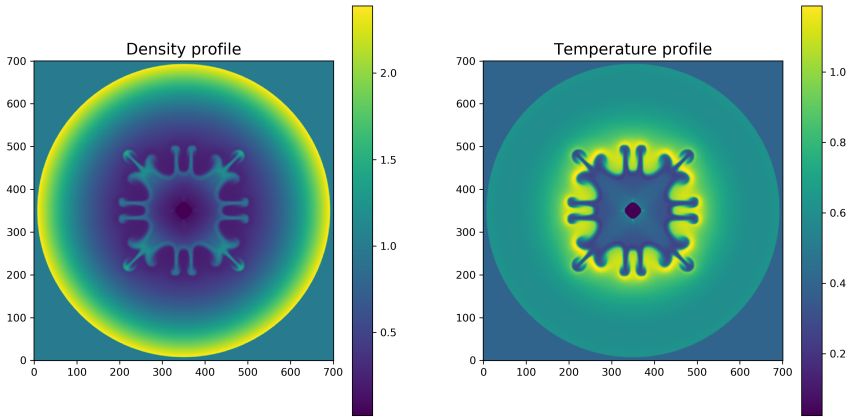


Figure 9: Blast-wave with Rayleigh instability, run with 9 processes. (resolution 700, 900 time-steps, d0=1e4, e0=1e5, w=6., power=4, eps=0.01)

Blast-wave collision

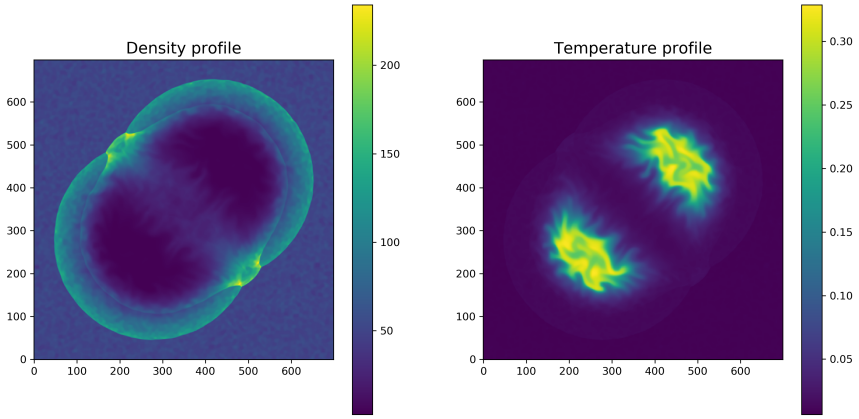


Figure 10: Blast-wave with Rayleigh instability, run with 9 processes. (resolution 700, 1300 time-steps, d0=1e4, e0=1e5, w=1., power=2, eps=0.01)

7.2 Code

```
import sys; sys.path.append("/home/jovyan/erda_mount/__dag_config__/python3")
import numpy as np
import matplotlib.pyplot as plt
from time import time
from mpi4py import MPI
sys.path.append("/home/jovyan/erda_mount/AstroProject/modules")
from HD_BLAST import hd, blast_wave, blast_wave_mass, blast_wave_mass_noise,
    blast_wave_collision
from HLL      import *
from MUSCL_2D import muscl_2d
from PLOTTING import imshow
from LOCAL    import local
from BCAST    import *

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

axis_divide = int(np.sqrt(size))   # dividing axis according to number of processes
setup       = np.reshape(np.arange(size, dtype=int), (axis_divide, axis_divide))
ghost       = int(sys.argv[3])     # taking the number of ghost zones from terminal

# Defining the blast wave to study:
n  = int(sys.argv[1])        # reading resolution from terminal
n  = n+n%axis_divide         # clause to get even number of resolution processes
nt = int(sys.argv[2])        # reading number of iterations from terminal
d0 = 1e4;e0 = 1e5; C = 0.5; solver = HLL
u  = blast_wave(n=n, e0=e0, d0=d0, gamma=1.4, w=1., power=4, eps=0.01)
# different blase waves such as blast_wave_collision can be run

global_n = n                       # resolution of global mesh
n_local  = global_n//axis_divide   # resolution of local sub-mesh

#calling local class in forder to localize global mesh for processor rank
loc = local(u, n_local, gamma=1.4, rank=rank, axis_divide=axis_divide, global_n=
    global_n, ghost=ghost, setup=setup)

start = time()
for it in range(nt):
    dt=np.array(loc.Courant(C))
    col_dt = np.zeros(1)
```

```
    comm.Allreduce(dt, col_dt, MPI.MIN)  # using Allreduce to find minimum dt of all
    processes
    loc = muscl_2d(loc, col_dt[0], Slope=MonCen, Riemann_Solver=solver)  # stepping
    forward in time
    sending(comm, loc, ghost)            # custom function for sending ghost zones
    loc = receiving(comm, loc, ghost)    # custom function for receiving ghost zones
    loc.t += col_dt[0]
used=time()-start
# printing time of time-evolution loop
if rank == 0:
    print('{:.1f} sec, {:.2f} microseconds/update'.format(used,1e6*used/(n**2*nt)))
# custom function for gathering all sub-meshes into local class for root=0
u = gathering(comm, rank, size, u, loc, 0, n_local, axis_divide, ghost)
if rank == 0:
    temp = u.temperature()   # calculating temperature profile
    # custom plotting function for plotting density and temperature profile
    imshow(u.rho, temp, 'result', e0, d0, C, n, headline=('Density profile', '
    Temperature profile'))
```

Listing 1: Main Program

```
import sys; sys.path.append("/home/jovyan/erda_mount/__dag_config__/python3")
import numpy as np
from HLL import *
from HD_BLAST import *

class local(hd):
    """
    class for localizing from global mesh according to process rank
    """
    t = 0.0
    def __init__(self, u, n_local, gamma, rank, axis_divide, global_n, ghost, setup):
        # calculating processor position within setup
        self.pos_i       = np.floor_divide(rank, axis_divide)
        self.pos_j       = rank%axis_divide
        # finding limits of sub-mesh within global mesh
        self.find_limits(n_local, global_n, ghost)
        # finding receivers/sources of boundary conditions
        self.find_receivers(setup)

        self.n     = n_local+2*ghost   # size of local mesh incl. ghost zones

        # initializing rho and Etot arrays
        self.rho   = np.ones((self.n, self.n))
        self.Etot  = np.ones((self.n, self.n))
        # copying relevant part from main mesh
        self.rho[ghost:-ghost, ghost:-ghost] = u.rho[self.limits['top']:self.limits['
    bottom'], self.limits['left']:self.limits['right']].copy()
        self.Etot[ghost:-ghost, ghost:-ghost]= u.Etot[self.limits['top']:self.limits['
    bottom'], self.limits['left']:self.limits['right']].copy()
        # initializing left boundary condition
        if self.edge_L:
            self.rho[ghost:-ghost, 0:ghost]       = u.rho[self.limits['top']:self.
    limits['bottom'], self.limits['left']-ghost:].copy()
            self.Etot[ghost:-ghost, 0:ghost]      = u.Etot[self.limits['top']:self.
    limits['bottom'], self.limits['left']-ghost:].copy()
        else:
            self.rho[ghost:-ghost, 0:ghost]       = u.rho[self.limits['top']:self.
    limits['bottom'], self.limits['left']-ghost:self.limits['left']].copy()
            self.Etot[ghost:-ghost, 0:ghost]      = u.Etot[self.limits['top']:self.
    limits['bottom'], self.limits['left']-ghost:self.limits['left']].copy()
        # initializing right boundary condition
        if self.edge_R:
            self.rho[ghost:-ghost, -ghost:]  = u.rho[self.limits['top']:self.limits['
    bottom'], 0:ghost].copy()
            self.Etot[ghost:-ghost, -ghost:]  = u.Etot[self.limits['top']:self.limits['
    bottom'], 0:ghost].copy()
        else:
            self.rho[ghost:-ghost, -ghost:]  = u.rho[self.limits['top']:self.limits['
    bottom'], self.limits['right']+1:self.limits['right']+1+ghost].copy()
```

```
            self.Etot[ghost:-ghost, -ghost:]  = u.Etot[self.limits['top']:self.limits['
bottom'], self.limits['right']+1:self.limits['right']+1+ghost].copy()
        # initializing top boundary condition
        if self.edge_T:
            self.rho[0:ghost, ghost:-ghost]       = u.rho[self.limits['top']-ghost:,
self.limits['left']:self.limits['right']].copy()
            self.Etot[0:ghost, ghost:-ghost]      = u.Etot[self.limits['top']-ghost:,
self.limits['left']:self.limits['right']].copy()
        else:
            self.rho[0:ghost, ghost:-ghost]       = u.rho[self.limits['top']-ghost:self
.limits['top'], self.limits['left']:self.limits['right']].copy()
            self.Etot[0:ghost, ghost:-ghost]      = u.Etot[self.limits['top']-ghost:
self.limits['top'], self.limits['left']:self.limits['right']].copy()
        # initializing bottom boundary condition
        if self.edge_B:
            self.rho[-ghost:, ghost:-ghost]  = u.rho[0:ghost, self.limits['left']:self.
limits['right']].copy()
            self.Etot[-ghost:, ghost:-ghost]  = u.Etot[0:ghost, self.limits['left']:
self.limits['right']].copy()
        else:
            self.rho[-ghost:, ghost:-ghost]  = u.rho[self.limits['bottom']+1:self.
limits['bottom']+1+ghost, self.limits['left']:self.limits['right']].copy()
            self.Etot[-ghost:, ghost:-ghost]  = u.Etot[self.limits['bottom']+1:self.
limits['bottom']+1+ghost, self.limits['left']:self.limits['right']].copy()
        #initializing remaining variables
        self.Px    = np.zeros((self.n, self.n))
        self.Py    = np.zeros((self.n, self.n))
        self.gamma = u.gamma
        self.x     = u.x[self.limits['left']:self.limits['right']]
        self.y     = u.y[self.limits['top']:self.limits['bottom']]
        self.ds    = self.x[1]-self.x[0]
        self.dx    = self.ds
        self.r = np.zeros((n_local, n_local))
        for i in range(n_local):
            self.r[i] = (self.y ** 2 + self.x[i] ** 2) ** 0.5

    def find_limits(self, n_local, global_n, ghost):
        """
        called by: each process
        returns:   the indices from which (L,T) to which (R,B) the local array
                   is cut out of the global array. Boolian indicating edge position
                   within global mesh.
        """
        self.limits = {}
        self.limits['left'] = int(self.pos_j*n_local)
        self.limits['right'] = int((self.pos_j+1)*n_local)
        self.limits['top'] = int(self.pos_i*n_local)
        self.limits['bottom'] = int((self.pos_i+1)*n_local)
        self.edge_L, self.edge_T, self.edge_R, self.edge_B = False, False, False, False
        if self.limits['right']+ghost >= global_n:
            self.edge_R = True
        if self.limits['bottom']+ghost >= global_n:
            self.edge_B = True
        if self.limits['left']-ghost <0:
            self.edge_L = True
        if self.limits['top']-ghost <0:
            self.edge_T = True

    def find_receivers(self, setup):
        """
        called by: each process
        returns:   receivers/sources of boundary conditions (cell on top, to the left,
...)
        """
        self.receivers = {}
        self.receivers['left'] = setup[self.pos_i, self.pos_j-1]
        self.receivers['top']  = setup[self.pos_i-1, self.pos_j]
        if self.edge_R:
            self.receivers['right'] = setup[self.pos_i, 0]
        else:
            self.receivers['right'] = setup[self.pos_i, self.pos_j+1]
        if self.edge_B:
            self.receivers['bottom'] = setup[0, self.pos_j]
```

```
        else:
            self.receivers['bottom'] = setup[self.pos_i+1, self.pos_j]
```

Listing 2: Localizing class

```
import sys; sys.path.append("/home/jovyan/erda_mount/__dag_config__/python3")
import numpy as np
from mpi4py import MPI


variables = ['rho', 'Etot', 'Px', 'Py']

# ---------------Boundaries and sending them--------------------
def boundary_init(rho, Etot, Px, Py, n, ghost):
    """
    called by: sending function
    returns:   dictionary containing all boundary condition arrays for one process
               (so for left, top, ... boundaries)
    """
    boundary = {'left':   {'rho':np.empty((n, ghost)), 'Etot':np.empty((n, ghost)),
                           'Px':np.empty((n, ghost)), 'Py':np.empty((n, ghost))},
                'top':    {'rho':np.empty((ghost, n)), 'Etot':np.empty((ghost, n)),
                           'Px':np.empty((ghost, n)), 'Py':np.empty((ghost, n))},
                'right':  {'rho':np.empty((n, ghost)), 'Etot':np.empty((n, ghost)),
                           'Px':np.empty((n, ghost)), 'Py':np.empty((n, ghost))},
                'bottom': {'rho':np.empty((ghost, n)), 'Etot':np.empty((ghost, n)),
                           'Px':np.empty((ghost, n)), 'Py':np.empty((ghost, n))}}
    i=0
    for var in [rho.copy(), Etot.copy(), Px.copy(), Py.copy()]:
        boundary['left'][variables[i]][:]   = var[:, ghost:2*ghost]
        boundary['top'][variables[i]][:]    = var[ghost:2*ghost, :]
        boundary['right'][variables[i]][:]  = var[:,-2*ghost:-ghost]
        boundary['bottom'][variables[i]][:] = var[-2*ghost:-ghost,:]
        i+=1
    return boundary

def sending(comm, l, ghost):
    """
    called by: each process, time evolution loop
    returns:   - (just completes sending operation of boundary conditions for each
               process)
    """
    boundary = boundary_init(l.rho,l.Etot,l.Px,l.Py, l.n, ghost)
    sides = ['left', 'top', 'right', 'bottom']
    tagg = 0  # defining tag as many similar packages are sent
    for side in sides:
        for variable in variables:
            comm.Send(boundary[side][variable][:], dest=l.receivers[side], tag=tagg)
            tagg += 1

def receiving(comm, l, ghost):
    """
    called by: each process, time evolution loop
    returns:   each of the variables with their new boundary conditions as received
               from the other processes
    """
    boundary = {'left':   {'rho':np.empty((l.n, ghost)), 'Etot':np.empty((l.n, ghost)),
                           'Px':np.empty((l.n, ghost)), 'Py':np.empty((l.n, ghost))},
                'top':    {'rho':np.empty((ghost, l.n)), 'Etot':np.empty((ghost, l.n)),
                           'Px':np.empty((ghost, l.n)), 'Py':np.empty((ghost, l.n))},
                'right':  {'rho':np.empty((l.n, ghost)), 'Etot':np.empty((l.n, ghost)),
                           'Px':np.empty((l.n, ghost)), 'Py':np.empty((l.n, ghost))},
                'bottom': {'rho':np.empty((ghost, l.n)), 'Etot':np.empty((ghost, l.n)),
                           'Px':np.empty((ghost, l.n)), 'Py':np.empty((ghost, l.n))}}
    sides = ['right', 'bottom', 'left', 'top']  # mirroring sending order
    tagg = 0
    for side in sides:
        for variable in variables:
            comm.Recv(boundary[side][variable][:], source=l.receivers[side], tag=tagg)
            tagg += 1
```

```
    i=0
    for var in [l.rho, l.Etot, l.Px, l.Py]:
        var[:, 0:ghost]  = boundary['left'][variables[i]][:]
        var[0:ghost, :]  = boundary['top'][variables[i]][:]
        var[:, -ghost:] = boundary['right'][variables[i]][:]
        var[-ghost:, :] = boundary['bottom'][variables[i]][:]
        i+=1
    return l

def gathering(comm, rank, size, u, l, root, n_local, axis_divide, ghost):
    """
    called by: each process, root process is sending & receiving, other processes
               just sending
    returns:   class u updated and put together after the local processes have done
               their jobs.
    """
    sendbuf_rho  = l.rho[ghost:-ghost,ghost:-ghost]
    sendbuf_Etot = l.Etot[ghost:-ghost,ghost:-ghost]
    sendbuf_Px   = l.Px[ghost:-ghost,ghost:-ghost]
    sendbuf_Py   = l.Py[ghost:-ghost,ghost:-ghost]
    master_rho  = comm.gather(sendbuf_rho, root=0)
    master_Etot = comm.gather(sendbuf_Etot, root=0)
    master_Px   = comm.gather(sendbuf_Px, root=0)
    master_Py   = comm.gather(sendbuf_Py, root=0)
    if rank == 0:
        for ran in range(size):   # looping over ranks
            pos_i = np.floor_divide(ran, axis_divide)
            pos_j = ran%axis_divide
            u.rho[pos_i*n_local:(pos_i+1)*n_local, pos_j*n_local:(pos_j+1)*n_local] =
    master_rho[ran].copy()
            u.Etot[pos_i*n_local:(pos_i+1)*n_local, pos_j*n_local:(pos_j+1)*n_local] =
    master_Etot[ran].copy()
            u.Px[pos_i*n_local:(pos_i+1)*n_local, pos_j*n_local:(pos_j+1)*n_local]   =
    master_Px[ran].copy()
            u.Py[pos_i*n_local:(pos_i+1)*n_local, pos_j*n_local:(pos_j+1)*n_local]   =
    master_Py[ran].copy()
    return u
```

Listing 3: Communication Functions

References

[1] Jeremy Bejarano. An introduction to parallel programming with mpi and python, 2013. https://materials.jeremybejarano.com/MPIwithPython/index.html, last checked: 22.01.2020, 11:00.