Distributed Computing for CFD in Python with MPI

Imad Kissami[1]

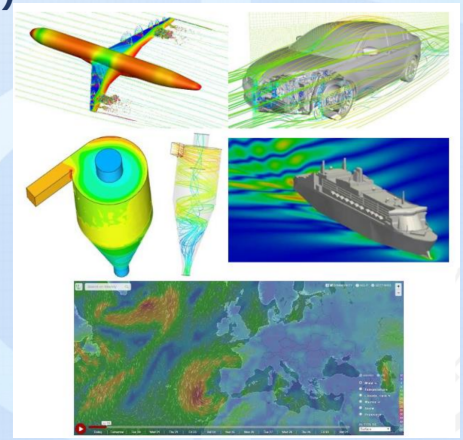
[1]Mohammed VI Polytechnic University, Benguerir, Morocco

September 16, 2025

What's CFD

Computational Fluid Dynamics (CFD)

- Branch of fluid mechanics using numerical analysis and data structures to solve problems involving fluid flows
- Aerospace, Automotive, Chemical Processing
- Hydraulics, Marine, Oil & Gas
- Power Generation, Weather forecasting

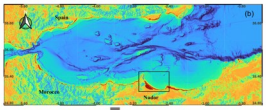


Why use HPC for CFD

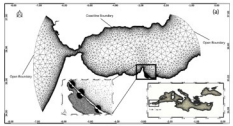
- Speed: Simulations executed quickly
- Memory: Large simulations can't fit in one machine
- Need for Supercomputers: Enables very large CFD studies

CFD Process

CFD Process

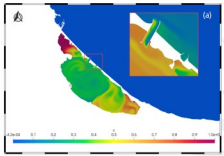


Preprocessing
=> From map to physical model



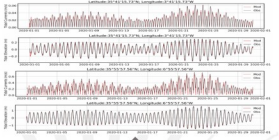
Solving:
=> Simulation using hundreds/thousands of cores

UM6P | College of Computing



Type of inlet	Initial tide	January March	April June	July September	October December
New inlet	high tides	15.58	14.67	16.02	17.36
	low tides	14.16	13.89	15.02	17.14
Old inlet	high tides	58.43	58.10	57.21	61.37
	low tides	57.22	56.73	56.61	59.06
two inlet	high tides	23.04	20.25	18.01	23.43
	low tides	22.85	20.03	19.79	22.74
three inlets	high tides	5.34	3.21	4.58	6.58
	low tides	3.02	3.36	2.67	6.23

Prediction => New inlets configuration

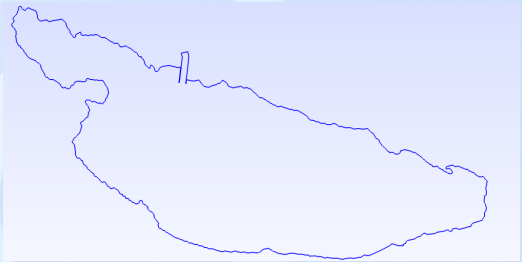


Analysis & Validation => Model data versus observed data comparison

CFD Example Cases

Preprocessing: Defining the geometry

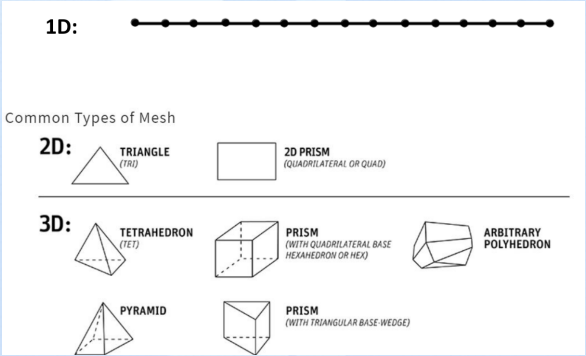
https://agencemarchica.gov.ma/le-projet/situation/



Domain defined using GMSH

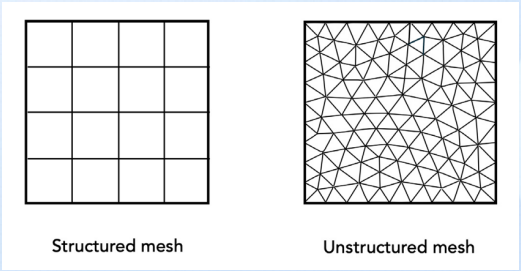
Preprocessing: Meshing

- The discretized domain is called the grid or the mesh.
- The governing equations are then discretized and solved inside each of these cells.
- Grids can either be:
  - Structured (quadrilateral, hexahedral)
  - Unstructured (cubic, tetrahedral)

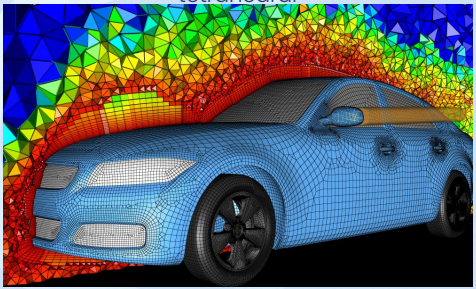


Preprocessing: Meshing

Structured for simple geometries ⇒ quad/hex

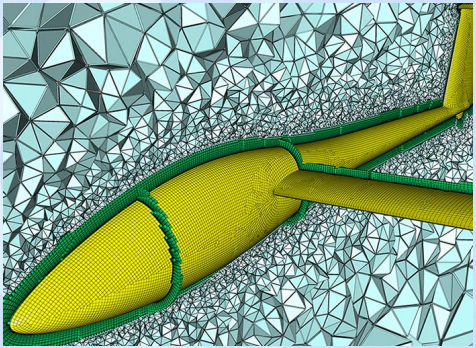


Unstructured for complex geometries ⇒ tetrahedral



Preprocessing: Meshing

- Hybrid for combined geometries



Preprocessing: Physics and Boundary Conditions

- Physics:
  - Turbulent vs. laminar
  - Incompressible vs. compressible
  - Single- vs. multi-phase
  - …
- Initial and Boundary Conditions:
  - Defining physical conditions on flow domain boundaries
  - Setting up initial conditions for the simulation

Discretization: Numerical methods in CFD

- Finite Volume Method (FVM):
  - Fluid flow problems
  - Accuracy and computational efficiency
- Finite Difference Method (FDM):
  - Structured grids but less adaptable to complex boundaries
- Finite Element Method (FEM):
  - High flexibility for complex geometrical domains
  - Minimize error functions over each element
  - Structural mechanics
- Physics Informed Neural Network (PINNs):
  - Mesh free

Available CFD packages

- Commercial:
  - ANSYS Fluent
  - COMSOL
  - STAR-CCM+
- Open-source/Free:
  - OpenFOAM
  - MFIX
  - SU2
  - FreeFem++
- Python packages:
  - Manapy (homemade)
  - SfePy
  - PyFR

FDM - DISCRETIZATION

Finite Difference Method (FDM)

Solving 1D advection equation:

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0 \qquad (1)$$

Discrete approximation:

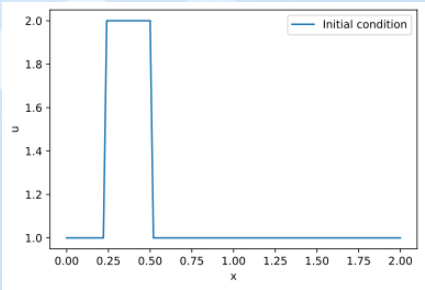
$$\frac{\partial u}{\partial x} \approx \frac{u(x+\Delta x) - u(x)}{\Delta x} \qquad (2)$$

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} + c\frac{u_i^n - u_{i-1}^n}{\Delta x} = 0 \qquad (3)$$

Solution in time:

$$u_i^{n+1} = u_i^n - c\frac{\Delta t}{\Delta x}(u_i^n - u_{i-1}^n) \qquad (4)$$

Initial condition



FDM - DISCRETIZATION

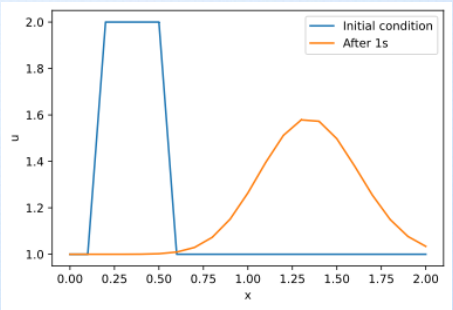
Finite Difference Method (FDM)

$$u_i^{n+1} = u_i^n - c\frac{\Delta t}{\Delta x}(u_i^n - u_{i-1}^n)$$

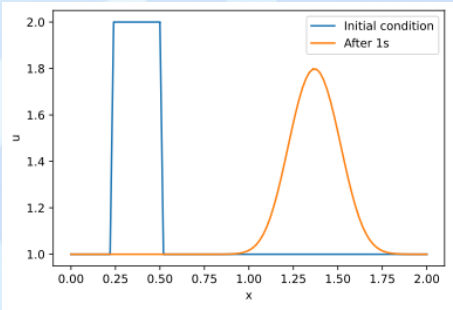
```
# CFL number
cfl =0.4
dt =cfl *dx / c

def solve_1d_linearconv(u, un, nx, dt, dx, c):
    t =0.
    while (t <tend):
        t +=dt
        un[:] =u[:]  # copy current values
        for i in range(1, nx):
            u[i] =un[i] -c *dt / dx *(un[i] -un[i -1])
        return u
```

FDM – DISCRETIZATION

Solution of 1D Linear Convection with Different Grid Resolutions



Grid with 21 points



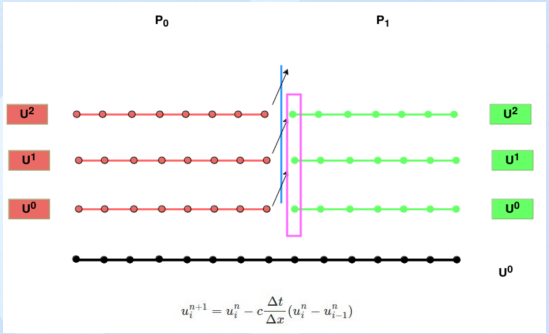
Grid with 101 points

Parallelization techniques

MPI parallelization

Grid decomposition

- MPI cart for structured grids
- Metis, Scotch for unstructured grids



MPI for communication

Initializing MPI in Python

Basic steps:

- Import the MPI module from `mpi4py`.
- Access the default communicator `COMM_WORLD`.
- Query:
  - `size`: total number of processes.
  - `rank`: ID of the current process.
- Each process executes the same code (SPMD model).

```
from mpi4py import MPI

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

print(f"Hello from rank {rank} "f"of {size}")
```

Output (4 processes):

```
Hello from rank 0 of 4
Hello from rank 1 of 4
Hello from rank 2 of 4
Hello from rank 3 of 4
```

FDM – P2P COMMUNICATION

Domain decomposition and ghost cell exchange

Steps:

- Split the 1D domain among processes.
- Each process stores its subdomain + 2 ghost cells.
- Exchange boundary values with left/right neighbors.
- Update interior points with FDM scheme.

```
# Voisins définis manuellement
left = rank -1 if rank >0 else MPI.PROC_NULL
right =rank +1 if rank <size -1 else MPI.PROC_NULL

if left !=MPI.PROC_NULL:
    comm.Send(u[1], dest=left)
    comm.Recv(u[0], source=left)

if right !=MPI.PROC_NULL:
    comm.Send(u[-2], dest=right)
    comm.Recv(u[-1], source=right)
```

FDM – 1D CARTESIAN TOPOLOGY

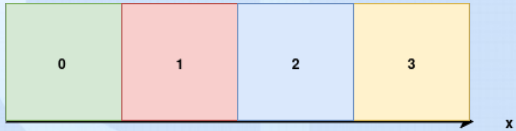
Python example with mpi4py:

```
# 1D Cartesian topology
dims =[size]
periods =[False]
cart_comm =comm.Create_cart(dims, periods=periods)

coords =cart_comm.Get_coords(rank)
left, right =cart_comm.Shift(0, 1)

# --- Exchange boundaries with neighbors ---
if left !=MPI.PROC_NULL:
    cart_comm.Sendrecv(sendbuf=un[1], dest=left, recvbuf=u[↩
        0], source=left)

if right !=MPI.PROC_NULL:
    cart_comm.Sendrecv(sendbuf=un[-2], dest=right, recvbuf=↩
        u[-1], source=right)
```

Output (for 4 processes):

- Rank 0 ⇒ coords [0], neighbors: left = -2, right = 1
- Rank 1 ⇒ coords [1], neighbors: left = 0, right = 2
- Rank 2 ⇒ coords [2], neighbors: left = 1, right = 3
- Rank 3 ⇒ coords [3], neighbors: left = 2, right = -2

(-2 means no neighbor if non-periodic).



FDM – SOLVING 1D ADVECTION EQ. WITH MPI4PY

Domain decomposition and ghost cell exchange

(1) Initialize MPI:

```
from mpi4py import MPI
comm =MPI.COMM_WORLD
rank =comm.Get_rank()
size =comm.Get_size()
```

(2) Set problem parameters:

```
nx_global =2001
dx =(2 -0) / (nx_global -1)
c =1
cfl =0.4
tend =1
dt =cfl *dx / c

if rank ==0:
    print("CFL =", cfl)
```

FDM – SOLVING 1D ADVECTION EQ. WITH MPI4PY

(3) Domain decomposition:

```
local_nx =nx_global // size
remainder =nx_global % size
if rank <remainder:
    local_nx +=1

u_local =np.ones(local_nx +2) # include ghost cells
un_local =np.ones(local_nx +2)
```

(4) Set initial condition:

```
grid =np.linspace(0, 2, nx_global)
u0 =np.ones(nx_global)
u0[int(0.25 / dx):int(0.5 / dx +1)] =2

counts =[(nx_global // size +(1 if r <remainder else 0)) for r in range(size)]
displs =[sum(counts[:r]) for r in range(size)]

local_data =np.zeros(local_nx)
comm.Scatterv([u0, counts, displs, MPI.DOUBLE], local_data, root=0)
u_local[1:-1] =local_data
```

FDM – SOLVING 1D ADVECTION EQ. WITH MPI4PY

(5) Ghost cell exchange routine:

```
def exchange_ghosts(u_local):
    if rank <size -1:
        comm.Send(u_local[-2].copy(), dest=rank +1, tag=0)
        comm.Recv(u_local[-1:], source=rank +1, tag=1)
    if rank >0:
        comm.Recv(u_local[0:1], source=rank -1, tag=0)
        comm.Send(u_local[1].copy(), dest=rank -1, tag=1)
```

(6) Time integration loop:

```
def solve_1d_linearconv(u_local, un_local, dx, dt, c):
    t =0.
    while (t <tend):
        t +=dt
        un_local[:] =u_local[:]
        exchange_ghosts(un_local)
        for i in range(1, len(u_local) -1):
            u_local[i] =un_local[i] -c *dt / dx *(un_local[i] -un_local[i -1])
    return u_local
```

FDM – SOLVING 1D ADVECTION EQ. WITH MPI4PY

(7) Gather results:

```
final =None
if rank ==0:
    final =np.empty(nx_global, dtype=float)

comm.Gatherv(u_local[1:-1], [final, counts, displs, MPI.DOUBLE], root=0)
```

(8) Post-processing:

```
if rank ==0:
    print("Timing is:", end -start)
    plt.plot(grid, u0, label="Initial condition")
    plt.plot(grid, final, label="After 1s")
    plt.xlabel("x")
    plt.ylabel("u")
    plt.legend()
    plt.savefig("mpi_solution.png")
```

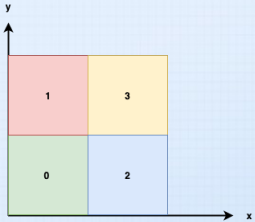
FDM - DISCRETIZATION (2D)

Finite Difference Method (FDM) in 2D

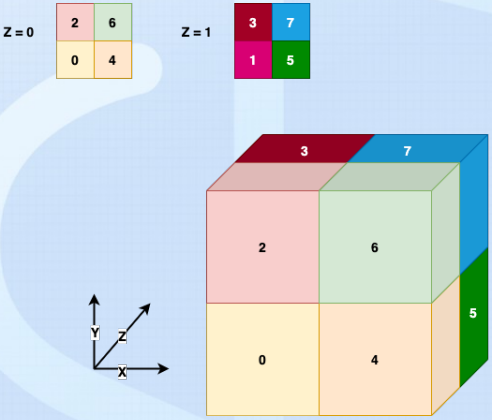
$$u_{i,j}^{n+1} = u_{i,j}^{n} - c_x \frac{\Delta t}{\Delta x}(u_{i,j}^{n} - u_{i-1,j}^{n}) - c_y \frac{\Delta t}{\Delta y}(u_{i,j}^{n} - u_{i,j-1}^{n})$$

```
# CFL condition
cfl =0.4
dt =cfl *min(dx/cx, dy/cy)

def solve_2d_linearconv(u, un, nx, ny, dt, dx, dy, cx, cy):
    t =0.
    while (t <tend):
        t +=dt
        un[:,:] =u[:,:] # copy current values
        for i in range(1, nx):
            for j in range(1, ny):
                u[i,j] =(un[i,j]
                    - cx*dt/dx*(un[i,j] -un[i-1,j])
                    - cy*dt/dy*(un[i,j] -un[i,j-1]))
    return u
```

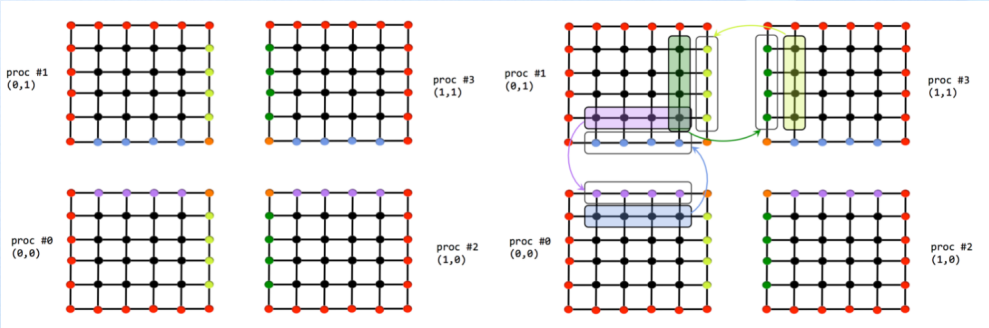
FDM – 2D/3D CARTESIAN TOPOLOGY




```
_periods =tuple([False,False])
_reorder =False
_dims =[2,2]

cart2d =comm.Create_cart(
              dims    = _dims,
              periods =_periods,
              reorder =_reorder
              )

coord2d =cart2d.Get_coords(rank)
```



FDM – 2D DISCRETIZATION



Cartesian decomposition between 4 processes

FDM – HALO EXCHANGE IN 2D CARTESIAN GRID

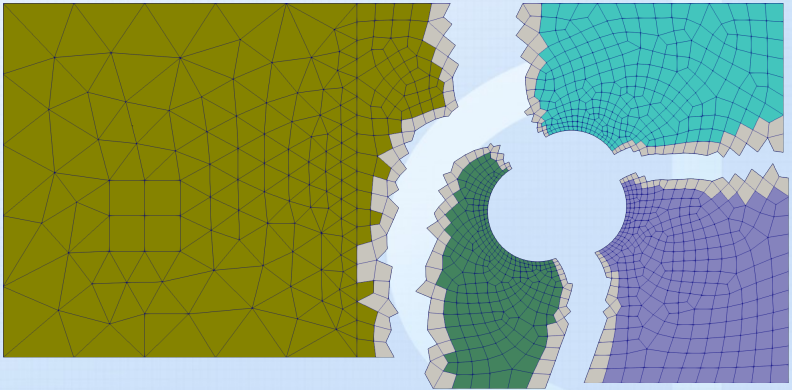
Two ways to communicate boundary (halo) data

- Option 1: Manual pack/unpack + **`Send/Recv`**
  - Extract boundary values into a contiguous buffer.
  - Use **`Send`** / **`Recv`** (or **`Sendrecv`**) with neighbors.
  - More flexible but requires explicit indexing and copies.
- Option 2: MPI Derived Datatypes
  - Define a custom type (e.g. column of a 2D array).
  - Allows direct communication of non-contiguous memory.
  - Cleaner and avoids manual packing.

```
# Row halo type
row_type =MPI.DOUBLE.Create_contiguous(nx)
row_type.Commit()

# Exchange top/bottom halos
comm.Sendrecv([u[1, :], 1, row_type], dest=up, [u[-1,↩
      :], 1, row_type], source=down)
```

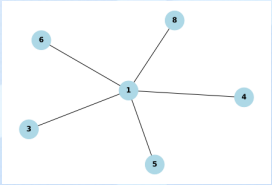
```
# Column halo type
col_type =MPI.DOUBLE.Create_vector(ny, 1, nx)
col_type.Commit()

# Exchange left/right halos
comm.Sendrecv([u[:,1], 1, col_type], dest=left, [u[:,↩
      -1],1, col_type], source=right)
```

Creating halo cells

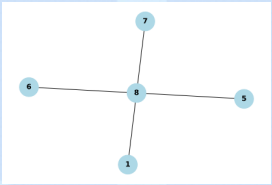


2D hybrid mesh splitted into 4 subdomain

FVM - 2D UNSTRUCTURED DISCRETIZATION



Neighbor procs for proc 1



Neighbor procs for proc 8

MPI Graph Communication

Creating a communicator based on a graph topology

- For unstructured meshes (partitioned with Metis/Scotch), the connectivity between subdomains is arbitrary.
- MPI provides graph communicators to handle this kind of neighborhood.
- Example in Python (`mpi4py`):

```
def create_mpi_graph(neighbors):
    topo =COMM.Create_dist_graph_adjacent(neighbors, neighbors, sourceweights=None, destweights=None)
    return topo
```

MPI Halo Exchange with Neighbor Collectives

Using `Neighbor_alltoallv` for halo communication

- Graph-based communicators allow exchanging halos with irregular neighbors.
- Two versions:
  - Blocking: `Neighbor_alltoallv`
  - Non-blocking: `Ineighbor_alltoallv`

```
def all_to_all(w_halosend, taille, scount, rcount, w_halorecv, ↩
      comm_ptr, mpi_precision):

   s_msg =[w_halosend, scount, mpi_precision]
   r_msg =[w_halorecv, rcount, mpi_precision]

   comm_ptr.Neighbor_alltoallv(s_msg, r_msg)
   w_halorecv =r_msg[0]
```

```
def Iall_to_all(w_halosend, taille, scount, rcount, w_halorecv, ↩
      comm_ptr):

   s_msg =[w_halosend, scount, MPI.DOUBLE_PRECISION]
   r_msg =[w_halorecv, rcount, MPI.DOUBLE_PRECISION]

   req =comm_ptr.Ineighbor_alltoallv(s_msg, r_msg)
   w_halorecv =r_msg[0]

   return req
```

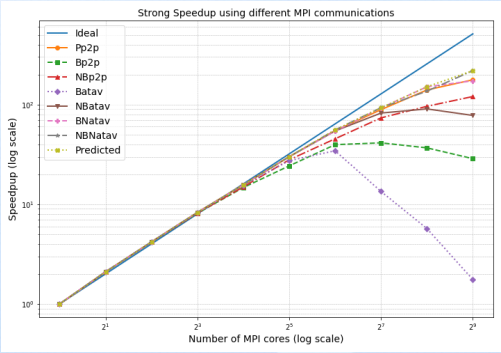
FVM - 2D UNSTRUCTURED DISCRETIZATION

Impact of MPI functions

#nb cells	compiler	#nb vars	#nb cores	#nb nodes	dim	var type	Pp2p	Bp2p	NBp2p	Batav	NBatav	BNatav	NBNatav
41322	gcc	4	1	1	2	float64	3.80e-04	5.04e-04	6.70e-04	**2.72e-04**	3.46e-04	4.94e-04	5.78e-04
9066872	intel	6	14	1	2	int64	9.53e-03	1.64e-02	1.03e-02	1.19e-02	**6.02e-03**	8.53e-03	7.50e-03
7046	gcc	9	56	2	2	int8	5.29e-03	2.25e-02	1.38e-02	2.22e-02	5.85e-03	**4.35e-03**	4.99e-03
10575418	gcc	7	112	3	2	int8	6.15e-03	3.82e-02	1.20e-02	1.00e-01	7.34e-03	5.68e-03	**5.52e-03**
102432	gcc	1	224	5	2	int16	**6.06e-04**	8.81e-03	2.04e-03	4.46e-02	1.50e-03	8.78e-04	7.80e-04
903630	intel	1	448	9	2	float32	**8.24e-04**	1.03e-02	2.08e-03	1.47e-01	3.31e-03	1.25e-03	1.18e-03
6082626	intel	3	1792	33	2	int16	4.30e-03	4.10e-02	5.96e-03	2.70e+00	2.63e-02	4.14e-03	**2.64e-03**
5576	gcc	3	1	1	3	float32	2.97e-04	3.79e-04	5.17e-04	**2.17e-04**	2.85e-04	3.98e-04	4.61e-04
508656	gcc	9	7	1	3	int16	1.15e-02	1.50e-02	1.27e-02	1.05e-02	**7.30e-03**	1.05e-02	9.28e-03
27556	intel	9	28	1	3	int16	9.78e-03	2.79e-02	2.49e-02	1.27e-02	9.18e-03	**9.01e-03**	1.07e-02
593783	gcc	9	56	2	3	int64	3.34e-02	1.35e-01	4.54e-02	5.50e-02	**2.34e-02**	3.05e-02	2.72e-02
180093	gcc	4	448	9	3	int16	9.00e-03	1.17e-01	1.73e-02	5.02e-01	1.27e-02	**6.51e-03**	6.65e-03
376499	gcc	7	896	17	3	float32	1.87e-02	4.13e-01	8.29e-02	2.37e+00	3.78e-02	1.67e-02	**1.29e-02**

FVM - 2D UNSTRUCTURED DISCRETIZATION

Impact of MPI functions



Strong Speedup using different MPI communication functions vs using the predicted one