Term Paper – Parallelized FTCS Method

Rafay Nawaid Alvi–2132500 & Haider Hussain–2131610

August 25, 2024

Contents

1 Introduction

The project is to parallelize the FTCS (Forward Time Central Space) method to solve the given PDE (Partial Differential Equation) using `Python` and `mpi4py`. FTCS method is an explicit time-stepping method for numerically solving Parabolic PDEs [1]. The given equation is the incompressible Navier Stokes Equation, i.e. diffusion-convection equation. This particular PDE is further simplifed to a Pure-Diffusion Equation [2].

$$\frac{\partial v}{\partial t} = \frac{\partial}{\partial x}\left[D(x)\frac{\partial v}{\partial x}\right] + S(x,t) \tag{1}$$

Here,

$$\begin{aligned} \partial v/\partial t &: \text{Rate of change of the scalar quantity } v \\ \partial[D(x)\tfrac{\partial v}{\partial x}]/\partial x &: \text{Diffusion term} \\ S(x,t) &: \text{External source term} \end{aligned}$$

For our particular case, we will be solving the equation in (1+1)-Dimension i.e. 1 dimension in space and 1 dimension in time. For visualization, we can think of the equation to represent the rate of change of a scalar quantity v (e.g. Heat, Concentration, Potential, etc.), as it spreads along the length L of a 1 dimensional rod parallel to the x-axis as shown in Fig. 1

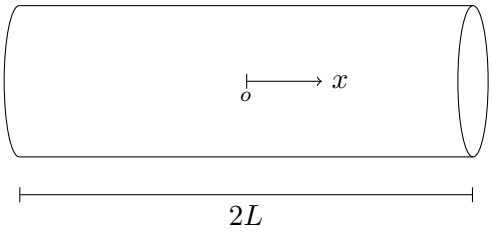

Figure 1: 1D rod model for visualization (the length is only relevant in the x-dimension, all other dimensions should be assumed as zero)

This project will discuss the numerical solution of the equation using the serial method and most importantly the parallelized method. In the first section, we'll discuss the algorithm of the FTCS method implemented serially (i.e. using a single process or `rank`), the discretization of the equation, stability condition, the corresponding code, results, and plots. The second section will primarily focus on the parallelization of the method, we'll use the same numerical method from the previous section, but also introduce communication between different processors. The relevant code to implement parallelization is also discussed. Finally, the results and plots of this method are compared and evaluated with the results of the first section. The last section will share the limitations and what can be done for further improvement of the current code.

2 Serial Method

In this section, the serial implementation of FTCS method is discussed. First the given PDE is decretized, and the initial and boundary conditions are specified. The stabilty condition is briefly touched to give reasoning on how to choose the step-sizes. Finally, the python program is revealed along with the results.

2.1 Domain Discretization

The numerical method that will be implemented to solve the given (1+1)-Dimensional Pure-Diffusion Equation is the FTCS (Forward Time Central Space) method. The given equation Eq. 1 is discretized into Eq. 2. Based on the centered space, symmetric finite differences for the space derivative was used (step-size in space $h/2$); and forward finite difference for the time derivative was used (step-size in time τ).

$$\begin{aligned} v(x, t+\tau) =& v(x,t) \\ &+ \frac{\tau}{h^2} D(x+h/2)[v(x+h,t) - v(x,t)] \\ &+ \frac{\tau}{h^2} D(x-h/2)[v(x-h,t) - v(x,t)] \\ &+ \tau S(x,t) \end{aligned} \tag{2}$$

The discretized domain is shown in Fig. 2. This is a discretization of the same rod shown in Fig. 1. Note that in both figures, the centre of our 1D domain is placed at the origin O. With each far end of the domain being L units from the origin. Each discretized cell is h units.

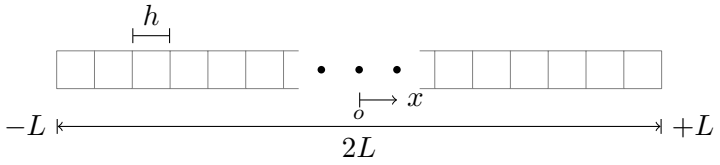

Figure 2: Discretized domain of the 1D geometry from Fig. 1.

2.2 Boundary and Initial Conditions

A key necessity to solving PDEs is the boundary and initial condition, here we use the Neumann boundaries: by setting $D(x) = 0$ on the boundaries, i.e. $D(-L - h/2) = D(L + h/2) = 0$ as shown in Fig. 3. The initial condition $v(x, t = 0) = v_0$, $D(x)$, $S(x, t)$, the simulation time T and domain length $2L$, can be given by the user in the program.

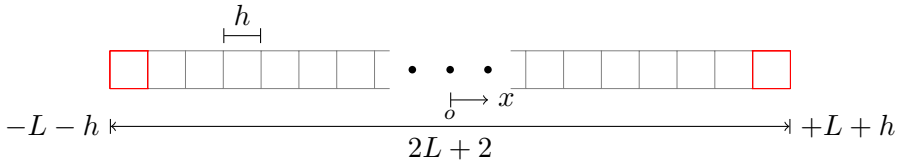

Figure 3: Discretized domain Fig. 2 with boundaries.

2.3 Solution Matrix

In order to plot the results, the value of each discretized cell $(x+h)$ at each time-step $(t+\tau)$ needs to be stored in solution matrix.
In our case, the solution matrix is a 2D matrix as shown in Fig. 4, with each element representing the value of the scalar quantity v at a particular time τ and space h. In the solution matrix, the rows represent the time-step τ, and the columns represent the space-step h. The size of the matrix is dependent on the coarseness of the discretization. However, the domain of our solution will always be $x = [-L, L]$ and $t = [0, T]$.

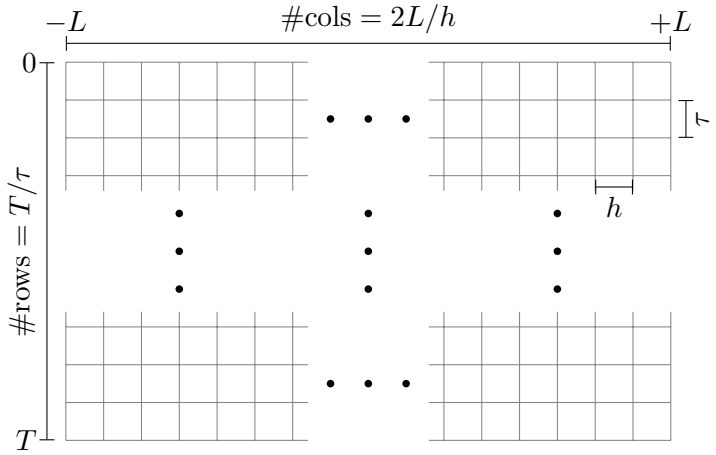

Figure 4: Solution Matrix. Each cell represents the value of v at a particular time τ and space h. The boundaries are not shown as they are not part of the final solution.

The implementation of discretizing the space and allocating the solution matrix in Python is shown below. Note that the boundaries, i.e. $-L-h$ and $L+h$ are included as well.

```
# decretized space
x = np.linspace(-L-h,L+h, np.ceil(2*L/h).astype('int')+2)
N = int(T/tau)
# solution matrix including L-h/2 and L+h/2
V = np.zeros([N, np.size(x)]);
```

2.4 Stability Condition

The step-sizes, h and τ are chosen such that accumulation of errors is prevented and the method is numerically stable. Thus we look into the stability condition. For the FTCS method of a 1D Parabolic PDE of the form $\frac{\partial u(x,t)}{\partial t} = \alpha \frac{\partial^2 u(x,t)}{\partial x^2}$, the following condition must be satisfied [3].

$$\frac{\tau}{h^2} \leq \frac{1}{2\alpha} \tag{3}$$

In our given differential equation, there's no scalar coefficient, hence, $\alpha = 1$. By fixing the value of τ, we can then determine the least step-size h, and hence the maximum number of discretized cells in our domain, i.e. columns of the solution matrix, is also determined.

Example: Let us fix $\tau = 0.001$, and determine the minimum value for h and the corresponding number of discretized cells. Continuing from Eq. 3.

$$\begin{aligned} h &\geq \sqrt{2\tau} \\ h &\geq \sqrt{2 \cdot 0.001} \\ h &\geq \frac{\sqrt{5}}{50} \quad (\text{where } \frac{\sqrt{5}}{50} \approx 0.0447) \end{aligned} \tag{4}$$

We can now determine the appropriate number of discretized cells of our domain (#cols).

Using the inequality obtained from Eq. 4 into the relationship $h = \frac{2L}{\#\text{cols}}$ from Fig. 4.

$$\begin{aligned} \frac{2L}{\#\text{cols}} &\geq \frac{\sqrt{5}}{50} \\ \#\text{cols} &\leq \frac{2 \cdot 5}{\sqrt{5}/50} \\ \#\text{cols} &\leq 100\sqrt{5} \quad (\text{where } 100\sqrt{5} \approx 223.6) \end{aligned} \tag{5}$$

The similar approach can be done for finer values as well, Tab. 1 below shows the values for h and #cols for various τ values.

For $\tau = 1 \times 10^{-1}$	$h \geq 0.1414$	$\#\text{cols} \leq 70.7$
For $\tau = 1 \times 10^{-2}$	$h \geq 0.0447$	$\#\text{cols} \leq 223.6$
For $\tau = 1 \times 10^{-3}$	$h \geq 0.0141$	$\#\text{cols} \leq 707.1$
For $\tau = 1 \times 10^{-4}$	$h \geq 0.0044$	$\#\text{cols} \leq 2236.1$

Table 1: τ and h that satisfy the stability condition Eq. 3, and the corresponding #cols for the solution matrix (i.e. the number of discretized cells in space)

It should be noted that the size of the entire solution matrix would increase pretty quickly with finer values of τ or h. For our demonstration, we will henceforth only work with $\tau = 0.001$ and $\tau = 0.0001$.

2.5 Python Program (Serial)

The following section will solve the FTCS method using a single processor. The parameters $2L$ for the domain length and T for the simulation time are specified. The desired coarseness h is chosen based on the value of τ and in accordance with the stability condition Eq. 3. The program is run for two cases $\tau = 0.001$ and $\tau = 0.0001$ respectively. The following parameters are fixed for each case:

$$\begin{aligned} 2L &= 10 \\ T &= 2 \\ h &= 2L/\text{cols} \end{aligned}$$

Thus specifiying our domain to $-5 \leq x \leq 5$ and $0 \leq t \leq 2$. Next, τ and h are chosen based on the information from Tab. 1.

case 1	**case 2**
$\tau = 0.001$	$\tau = 0.0001$
$\text{cols} = 223 \implies h = 0.0448$	$\text{cols} = 707 \implies h = 0.0141$

The initial condition $v_0 = \begin{cases} 1 & \text{if } |x| < 1.5 \\ 0 & \text{else} \end{cases}$. D and S are initialized as

```
v0 = lambda x   : float(abs(x)<1.5);
D  = lambda x   : 1
S  = lambda x,t : 0
```

The numerical method begins by assigning the initial values in the solution matrix `V`.

```
for l in range(np.size(x)):
    V[0,l] = v0(x[l])
```

The following code is the FTCS method. This is the direct translation of the discretized equation Eq. 2 to Python. The variables `Dp` and `Dm` represent the value of the coefficient $D(x)$ evaluated at $D(x+h/2)$ and $D(x-h/2)$, and it is $D(x) = 0$ at the borders as discussed in Section. 2.2.

```
for lt in range(1,N):
    for lx in range(1,np.size(x)-1):
        # evaluate D at lx
        Dp = D(x[lx]+h/2) * (np.abs(x[lx]+h/2)<L)
        Dm = D(x[lx]-h/2) * (np.abs(x[lx]-h/2)<L)

        V[lt,lx] = V[lt-1,lx] + (tau/h**2)*Dp*((V[lt-1,lx+1]) - V[lt-1,lx])+\
                                (tau/h**2)*Dm*((V[lt-1,lx-1]) - V[lt-1,lx])+\
                                tau*S(x[lx], -(lt-1)*tau)
```

From the for-loops, we can see that to evaluate `V` at every step in `lt` i.e. `V[lt,lx]`, the values of `V` at locations `lx-1, lx, lx+1` from the previous time-step `lt-1` must be known as shown in Fig. 5.

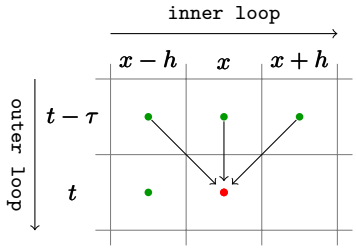

Figure 5: Evaluation of `V` at a single-step x (i.e. the inner for-loop) in the FTCS method. All green dots are known, and red is the unknown value being evaluated.

2.6 Program Files

The directory `serial_diffusion` contains files to run the program as shown in Fig. 6. To run, execute the command: `$ python main.py`.

Figure 6: Contents within serial directory before and after running `main.py`

The user inputs are intialized as:

```
L = 5
T = 2
h = (2*L)/223 # or (2*L)/707
tau = 0.001 # or 0.0001
```

2.7 Results

Currently, we are interested in knowing if our implementation is correct. Fortunately, for our 1D problem, Eq. 1, there exists an analytical solution [2]:

$$v(x,t) = \frac{1}{2}\left[\text{erf}\left(\frac{1.5-x}{2\sqrt{Dt}}\right) - \text{erf}\left(\frac{-1.5-x}{2\sqrt{Dt}}\right)\right] \tag{6}$$

Here, the error-function $\text{erf}(z) = \frac{2}{\pi}\int_0^z e^{-y^2}\text{d}y$. This error-function can be used in Python through the SciPy library: `scipy.special.erf()`
To verify our implementation, we can easily plot our solution obtained through the FTCS method against the analytical solution Eq. 6. Below are both solutions for $v(x,t)$ plotted at different times t.

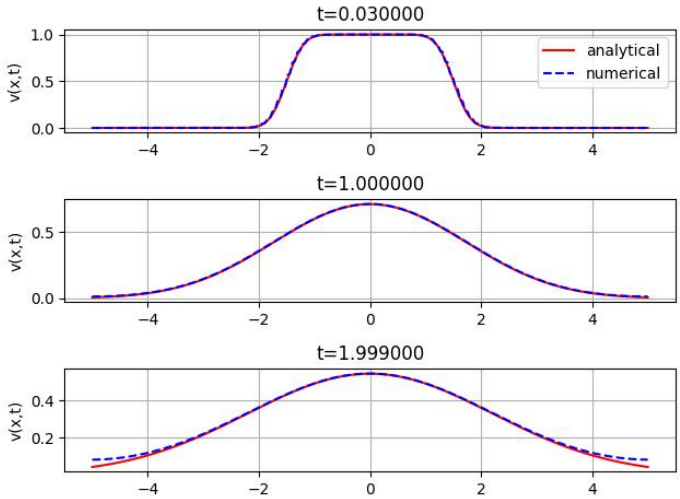

Figure 7: Comparison of the solutions obtained analytically (red) and FTCS method (dashed blue)

We can see in Fig. 7 that our numerical method performs really well, specially for times $t \ll T$.
The execution times for the numerical method for each case τ are:

	τ	h	solution matrix `V`	execution time $[s]$
case 1	1×10^{-2}	0.0448	2000×223	5.83
case 2	1×10^{-3}	0.0014	20000×707	187.27

Table 2: Execution times for each case. The size of `V` significantly increases with finer τ and h, thus increasing execution times. These programs were executed in the **stromboli** environment

3 Parallel Method

This section modifies the implementation done in the previous section such that numerical method is run by more than a single processor. The main idea is to divide the domain from Fig. 2 and distribute it to a certain number of processors. Since each processor has one chunk of the entire domain, they can all locally solve for their domains simultaneously and combine the results, resulting in a significant decrease in the total execution time. In Python, this approach makes use of the MPI (Message Passing Interface) module `mpi4py`.
In accordance with convention, the terminology `rank` will be used to refer to individual processes. And to keep things easier, the `size` (i.e. the total number of `ranks`) is chosen to always be a power of 2, i.e. `size` $= 2^r$, where $r \in \mathbb{N}_0$ (non-negative integer).

3.1 Domain Decomposition

Since the domain needs to be divided evenly among 2^r `ranks`, the #cols must be divisible by every `size` 2^r. Let us now estimate the appropriate h that generates #cols such that they can be evenly divided among 2^r `ranks`.
Example: Let us fix $\tau = 0.001$. From Tab. 1, we know $h \geq 0.0447$ and consequently we selected the maximum possible #cols $= 223$. In this case, 223 cannot be evenly divided into 2^r `ranks`, so we choose the nearest power of 2 to be our column size. Let us denote all such #cols: $2^r \leq 223$ in a set S_1.

$$S_1 = \{2^r \mid 2^r \leq \max(\#\text{cols}), r \in \mathbb{N}_0\} \tag{7}$$

For $\max(\#\text{cols}) = 223$, S_1 becomes:

$$\begin{aligned} S_1 &= \{2^r \mid 2^r \leq 223, r \in \mathbb{N}_0\} \\ &= \{1, 2, 4, 8, 16, 32, 64, 128\} \end{aligned} \tag{8}$$

S_1 contains all #cols that are divisible by every `size` $= 2^r \leq 223$, and the maximum #cols we can now use is $\max(S_1) = 128$, which is significantly less than 223 from our serial attempt. To improve on #cols, let us also fix the maximum `size` $= 2^k = 64$. With this we can now increase the column size to a number that is a multiple of 64 and also satisfies #cols ≤ 223. Let us denote all such #cols: $\max(S_1) \leq 64 \cdot p \leq 223$ in a set S_2.

$$S_2 = \{2^k \cdot p \mid \max(S_1) \leq 2^k \cdot p \leq \max(\#\text{cols}), p \in \mathbb{N}_0\} \tag{9}$$

For $\max(\#\text{cols}) = 223$ and $2^k = 64$, S_2 becomes:

$$\begin{aligned} S_2 &= \{64 \cdot p \mid 128 \leq 64 \cdot p \leq 223, p \in \mathbb{N}_0\} \\ &= \{128, 192\} \end{aligned} \tag{10}$$

The $\max(S_2)$ would be the maximum #cols for our solution matrix, given a τ and a specified max number of ranks, i.e `size` $= 2^k$.

$$\begin{aligned} \#\text{cols} &= \max(S_2) \\ &= 192 \implies h = 0.052 \qquad \because h = \frac{2L}{\#\text{cols}} \end{aligned} \tag{11}$$

To summarise, given $\tau = 0.001$ and `sizes` $2^k \leq 64$, the domain is discretized to 192 cells, i.e. $\max(\#\text{cols}) = 192$ which can be evenly distributed to all `ranks` as shown in Tab. 3.

`size` $= 2^k$	1	2	4	8	16	32	64
#cols per `rank`	192	96	48	24	12	6	3

Table 3: Domain decomposition for `sizes`: $2^k \leq 64$ and $\tau = 0.001$

Similarly, for $\tau = 0.0001$ and `sizes` $2^k \leq 64$, we will have the following:

$$\begin{aligned} S_1 &= \{1, 2, 4, 8, 16, 32, 64, 128, 256, 512\} \\ S_2 &= \{512, 578, 640, 704\} \\ \max(S_2) = 704 \implies h = 0.014 \qquad &\because h = \frac{2L}{\#\text{cols}} \end{aligned} \tag{12}$$

`size` $= 2^k$	1	2	4	8	16	32	64
#cols per `rank`	704	352	176	88	44	22	11

Table 4: Domain decomposition for `sizes`: $2^k \leq 64$ and $\tau = 0.0001$

The way the domain would be decomposed using MPI is as follows:

```
# allocate space of local_x in all ranks
x = np.zeros(global_array_size//size)
if rank == 0:
    h = (2*L/global_array_size)
    # discretization of the entire space 2L
    global_x = np.linspace(-L,L, np.ceil(2*L/h).astype('int'))
else:
    # initialize the variables in all other ranks to 0.
    global_x = np.zeros(1)
    h = 0

# distribution of global_x to all ranks
comm.Scatter(global_x, x, root = 0)
```

Here, `global_x` is the total discretization of the entire domain and it only exist in `rank=0`, it is distributed to each `rank` in their local variable `x`. This decomposition of the domain is done using `Scatter()`, the working of which is demonstrated in Fig. 8. Also, the value for #cols is defined in `global_array_size` which is used to initialize `h`.

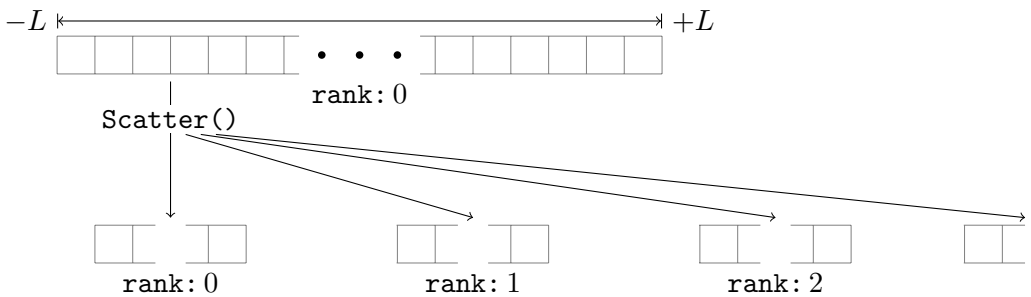

Figure 8: Domain decomposition of the discretized grid Fig. 2 to all `ranks`. The grid is evenly distributed using `Scatter()`

3.2 Communication

Since each `rank` has an equally divided part of the entire domain, it is required that they all implement the FTCS method locally and simultaneously. In order for this to work, each `rank` must be able to share its bordering values with its neighbouring `rank`. So we introduce a *halo* around each local discretization `local_mat` as shown in Fig. 9.

```
def create_halo(local_mat: np.ndarray, h: float):
    # ...
    local_mat = np.r_[local_mat[0]-h, local_mat, local_mat[-1] + h]
    return local_mat
```

Figure 9: `create_halo()` creates borders at each local discretization which is needed to implement the initial condition and sharing values.

At each rank, `create_halo()` concatenates one step after and one step before the local discretized `x`. This ensures that the bordering values of each rank are overlapped, which is useful for implementing the initial condition and storing the values as boundaries to perform FTCS (as shown in Fig. 5). Furthermore, the solution matrix is allocated at each rank.

```
x = create_halo(x,h)
V = np.zeros([N, np.size(x)])
```

After each time-step of the FTCS method, each `rank` should have a new row of calculated `V[lt,:]` values in its local solution matrix `V`. It will then *send* its the bordering values of the solution matrix to its neighbouring `ranks` on the left and right, and also *receive* the bordering values of the neighbouring `ranks`. This can be done using `Sendrecv()`

```
# send and receive from left rank
if rank > 0:
    comm.Sendrecv(sendleft, rank - 1, 0, recvleft, rank - 1)
# send and receive from right rank
if rank < size - 1:
    comm.Sendrecv(sendright, rank + 1, 0,recvright, rank + 1)
```

The above code is a snippet of a user-defined function `exchange_vals()`, that is used in the parallelized FTCS method shown below.

```
# ftcs method
for lt in range(1,N):
    V = exchange_vals(V,lt)
    for lx in range(1,np.size(x)-1):
        Dp = D(x[lx]+h/2) * (np.abs(x[lx]+h/2)<L)
        Dm = D(x[lx]-h/2) * (np.abs(x[lx]-h/2)<L)

        V[lt,lx] = V[lt-1,lx] + (tau/h**2)*Dp*((V[lt-1,lx+1]) - V[lt-1,lx])+\
                                (tau/h**2)*Dm*((V[lt-1,lx-1]) - V[lt-1,lx])+\
                                 tau*S(x[lx], -(lt-1)*tau)
```

This is the FTCS method that is implemented locally and simultaneously by every `rank` to solve for their own solution matrix. It can be seen from Fig. 10 that each rank `sends` a value from their borders, but `receives` a value into its solution matrix.

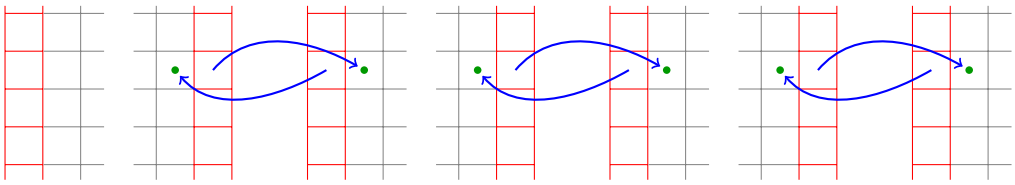

Figure 10: Sharing of bordering values at each time-step. As `rank:0` doesn't have a left neighbour, it only performs `Sendrecv()` on the right. The last `rank` would behave similarly

Finally, once the numerical method is completed, all of the solution matrices are combined back at `rank=0` into a complete solution matrix (like Fig. 4) using `gather()`.

```
global_V = comm.gather(V, root = 0)
```

3.3 Project Files and Submissions to SLURM

The intent is to generate strong scaling plots, i.e. we need to observe how the execution time for the FTCS method speeds up with increase in number of processors, i.e. `sizes` $\leq 2^r$ compared to `size` $= 1$. And we also need to observe the execution times for increasing h and τ.
To conduct these simulations, we have divided the project in two cases:

Case1: $\tau = 0.001$
Here, we will work with the domain decompositions shown in Tab. 3. Additionally, the discretization step h is also varied to show the execution times for each case. The other discretization steps used are from the set $S_1 \cup S_2 = S$ from Eq. 8 and Eq. 10. However we are only interested in values that keep $h \leq 1.5$ so as to keep a somewhat reasonable discretization. This amounts to values:

$$\begin{aligned} S_{\geq 8} &= \{x | x \in S, x \geq 8\} \\ &= \{8, 16, 32, 64, 128, 192\} \end{aligned} \tag{13}$$

`size` $= 2^k$	1	2	4	8	16	32	64	$h = \frac{2L}{\#\text{cols}}$
#cols per `rank`	192	96	48	24	12	6	3	0.052
#cols per `rank`	128	64	32	16	8	4	2	0.078
#cols per `rank`	64	32	16	8	4	2	1	0.156
#cols per `rank`	32	16	8	4	2	1	-	0.312
#cols per `rank`	16	8	4	2	1	-	-	0.625
#cols per `rank`	8	4	2	1	-	-	-	1.250

Table 5: All 36 subcases for Case1: $\tau = 0.001$

Similarly for **Case2:** $\tau = 0.0001$
The domain decompositions are taken from Tab. 4. The discretizations are from $S_1 \cup S_2 = S$ from Eq. 12.

$$S_{\geq 8} = \{8, 16, 32, 64, 128, 256, 512, 570, 578, 640, 704\} \tag{14}$$

size $= 2^k$	1	2	4	8	16	32	64	$h = \frac{2L}{\#\text{cols}}$
#cols per rank	704	352	176	88	44	22	11	0.014
#cols per rank	640	320	160	80	40	20	10	0.016
#cols per rank	578	289	144	72	36	18	9	0.017
#cols per rank	512	256	128	64	32	16	8	0.020
#cols per rank	256	128	64	32	16	8	4	0.039
#cols per rank	128	64	32	16	8	4	2	0.078
#cols per rank	64	32	16	8	4	2	1	0.156
#cols per rank	32	16	8	4	2	1	-	0.312
#cols per rank	16	8	4	2	1	-	-	0.625
#cols per rank	8	4	2	1	-	-	-	1.250

Table 6: All 64 subcases for Case1: $\tau = 0.0001$

The directory `parallel_diffusion` contains the files to run the program. The program is ran in the **stromboli** cluster environment provided by Bergische Universität Wuppertal (BUW). The following commands are executed within this folder to run the program:

```
$ module load tools/anaconda3/
$ python main.py
```

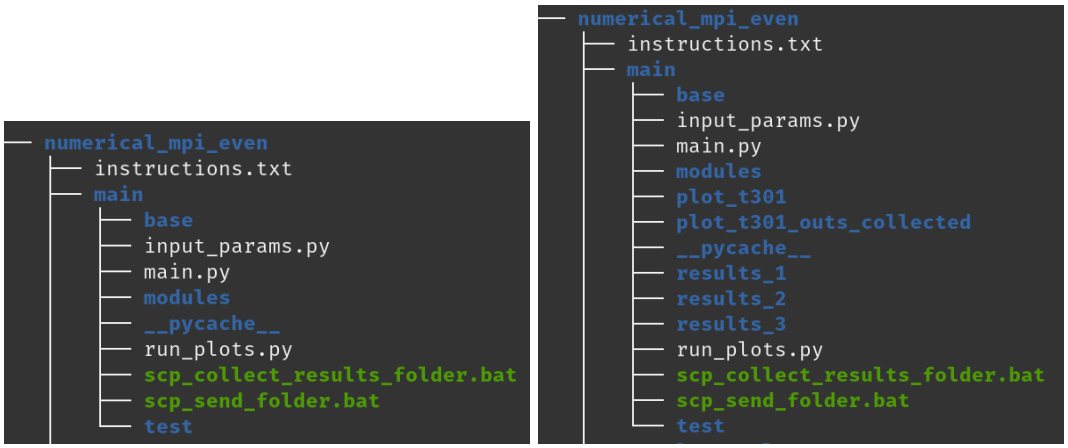

Figure 11: Contents within the directory before and after running `main.py`

The `base/` directory contains `ftcs.py` file with additional functionalities to enable parallelization. `main.py` copies the `base/` to the `results_*/` directories, with the parameters τ, h and `size` initialized for each particular subcase.

```
copy_files_base_to_case()

node = assign_node() # assign appropriate node

modify_case_params() # modification of parameters in each case

run_case() # run case inside each case directory
```

Figure 12: Functions inside `main.py` to illustrate the automation

This is `repeated` 3 times, as indicated in Fig. 13, resulting in the directories `results_1/`, `results_2/`, and `results_3/`. Each `results` folder contains all simulations for a given τ with similar plots as Fig. 7. The parameters for a different τ can be changed in `input_params.py`.

```
# USER INPUT for tau = 0.0001
tau = 0.0001
x_divs = [8, 16, 32, 64, 128, 256, 512, 576, 640, 704]
procs  = [1,2,4,8,16,32,64]
repeat = 3
delta_t = "plot_t401"
```

Figure 13: Snippet from `input_params.py` showing user input

The process of submitting each case through `SLURM` is automated in `script_main.py` found inside `modules/`, these functions are utitlized in `main.py` as shown in Fig. 12. This also makes sure that the appropriate `nodes` are selected based on the amount of processors used. To avoid overloading, a new case is submitted once the previous is completed or the program at least waits a certain amount of time before submitting.

```
os.system("sbatch submit_script.sh")
```

After execution, the program also prompts to run `run_plots.py` to generate the relevant plots. This program can be run separately as well, provided the `results_*` folders exist and the parameters provided in `input_params.py` are not changed.

```
$ python run_plots.py
```

3.4 Scalability Plot: Strong Scaling

Since we are already sure about the correctness of the program from Section. 2.7 because we're using the same implementation of the FTCS from the serial section. Here, we will be mostly focusing on the performance of our parallelization, i.e. the time taken to execute the numerical method in relation with the amount of processes used. The plots shown in this section are the strong scaling plots and the comparion of execution times for all cases from Tab. 3 and Tab. 4.
For the strong scaling, the size of the problem is kept constant, and the Speed-up($\frac{t_n}{t_1}$) is observed in relation with the increase in the number of processes (i.e `size`). Here, the speed-up is the ratio of t_n (the execution time for n processes) to t_1 (execution time for 1 process or the serial process).

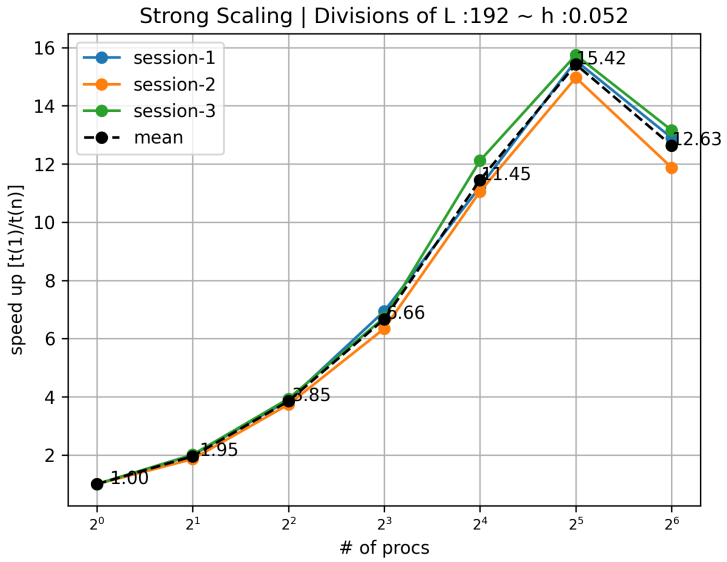

Figure 14: Strong Scaling Plot for $\tau = 0.001$, this shows the speed up in time with increase in `size`

The plots in Fig. 14 and Fig. 15 show the speed up in the execution time with increase in the number of processes. It can be observed that with each successive power of 2 in the number of processes, the speed-up nearly doubles from the previous step.
The sudden decrease in speed-up at the end can be explained by the fact that the overall communication between a large a number of processes is significantly more than the local problem size within each process. Thus, regardless of the speed of the numerical method, the high communication overhead between processes slows down the overall performance.

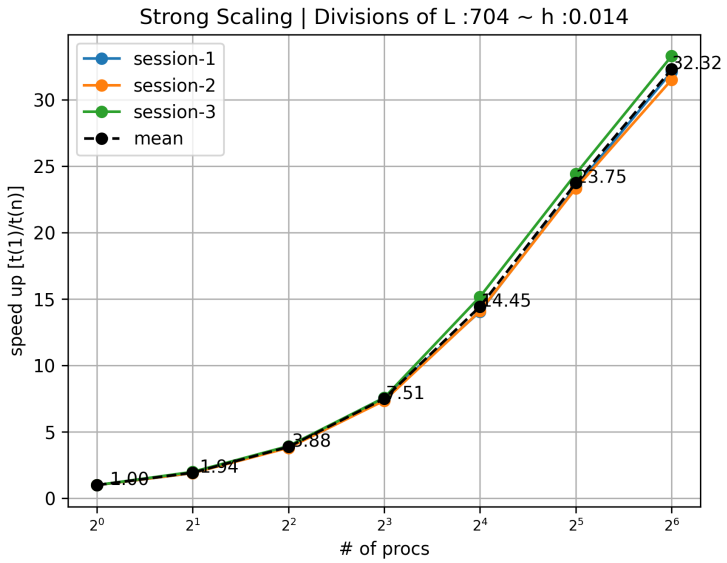

Figure 15: Strong Scaling Plot for $\tau = 0.0001$, this shows the speed up in time with increase in `size`

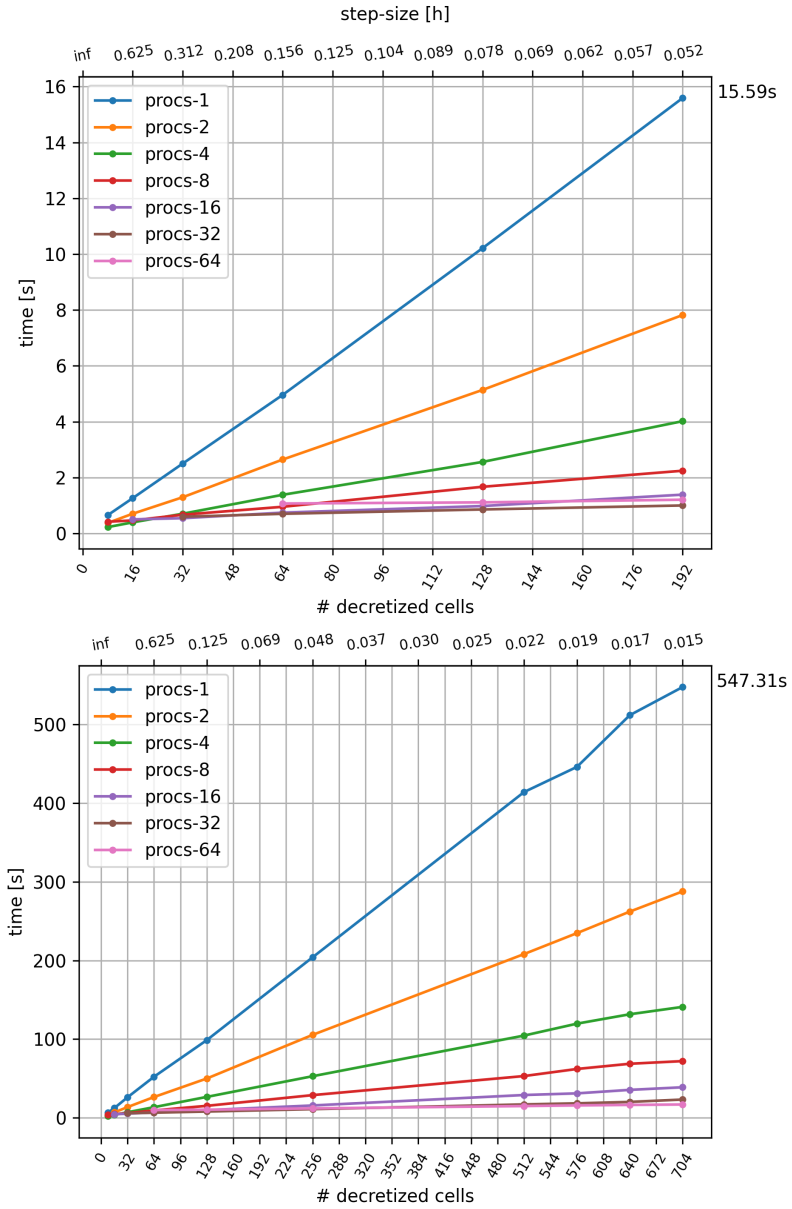

Figure 16: Execution times of all cases in Tab. 3 (top) and Tab. 4 (bottom)

It can be observed from Fig. 16 that the doubling of speed-up with increase in processors (or the halving of execution times) is true for other problem sizes as well. We can also recognize the effect of communication overhead for `procs-64` in the plot at the top.

4 Conclusion

This section gives a brief overall summary of the entire project as well as the assumptions made in the methods are also discussed alongwith ideas for improvement.

4.1 Summary

In this paper, we first introduce the FTCS method and briefly describe its usage in solving parabolic PDEs. We first discuss the serial implementation of the method, within which we discuss stability conditions to choose appropratie τ and h values. Next, the parallellization of the method is discussed, where we demonstrate how domain decomposition is done, and how communication within the processes is established. The results for both methodologies are shown and the improvement in execution times are discussed.

4.2 Uneven Distrbution: `Scatterv()`

The minimum step-size h used in the serial method is not used in the parallel method, reason being that we wanted even distribution of the decomposed domain and also to always keep the number of processes as a power of 2. This is a voluntary choice to easily distribute the decomposition through a simple method `Scatter()`. However, if further customization is needed, we can make the distribution uneven as well using `Scatterv()`, for this the first `rank` will the remainder of the decomposed domain. The uneven distribution allows us to use any number of processes, with no issues in using the same minimum step-size as in the serial method.

4.3 Final Thoughts

The overall project was exciting as well as very informative. We, however, find using MPI in C/C++ a bit more clear and explicit as opposed to `mpi4py` in Python which we found a bit less intuitive and abstract. But this hands-on approach to an MPI Project really made us appreciate the usefulness of parallelization in research projects as well as numerical methods.

List of Figures

List of Tables

References

[1] Richard H. Pletcher John C. Tannehill Dale A. Anderson. *Computational Fluid Mechanics and Heat Transfer (2nd ed.)* 1997. ISBN: 1-56032-046-X.

[2] Tomasz Korzec. *Nm2b Hydrodynamics python/MPI.*

[3] David F. Griffiths. *Essential Partial Differential Equations Analytical and Computational Aspects*. Springer Undergraduate Mathematics Series. Springer, 2015. Chap. Finite Difference Methods for Parabolic PDEs. ISBN: 978-3-319-22568-5.