TDT4200 Parallel programming

PS2

Maren Wessel-Berg & Claudi Lleyda Moltó

September 2023

Practical information

Published: 12/09/23

Deadline: 19/09/23 at 22:00

Evaluation: Pass/Fail

- Completing the problem set is **mandatory**.
- The work must be done **individually** and without help from anyone but the TDT4200 staff.
- **Reference** all sources found on the internet or elsewhere.
- The **requirements**, and **how and what to deliver** is explained in the problem set description found on BlackBoard.
- **Start early!**

Where can you get help with the assignment?

- **Recitation lecture**: introduction to the problem set

  (Today)
  Slides will be made available online.
- **TA hours**: ask questions in person

  Friday, September 15, 10:00–12:00 in Cybele
  Monday, September 18, 13:00–15:00 in Cybele
- **Piazza**: question forum

  Ask questions any time (but give us time to answer).
  Select the ps2 folder for questions related to this problem set.
  Do not post full or partial solutions!

Topic

Finite difference approximation of the 2D heat equation using MPI

- In PS1 you worked with a **sequential** code that solved the **1D heat equation** using **implicit methods** (Jacobi, Gauss-Seidel, Red-Black Gauss-Seidel).

  The goal was for you to familiarise yourself with the exercise and code setup, and to program in C, which will be useful for all future exercises.
- In PS2 you will implement an **explicit method** for solving the **2D heat equation** using **MPI**.
- You will also **answer questions** about your implementation and the curriculum.

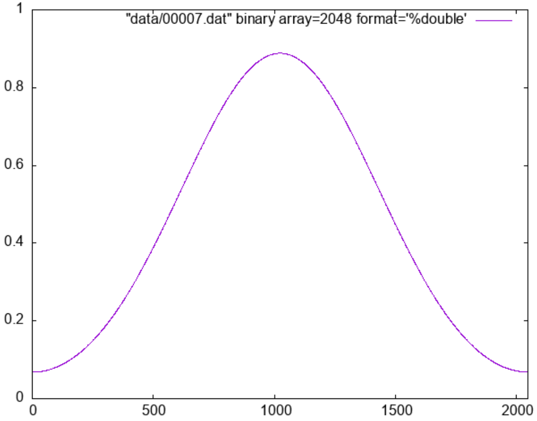
Today

- Introduce the problem set.
- Talk about potential challenges that can arise when parallelising the FDM for the 2D heat equation.
- Repeat some MPI concepts from the main lectures.

1D heat equation

Was explained in the recitation for PS1

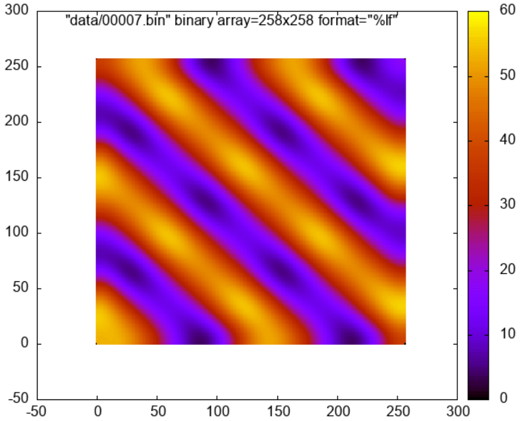
$$\frac{\partial u}{\partial t} = K\frac{\partial^2 u}{\partial x^2}$$



2D heat equation

For this exercise we are adding a dimension

$$\frac{\partial u}{\partial t} = K\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right)$$



Solving the 2D heat equation

Shockingly, finding a solution to the heat equation does not suddenly become easy when adding another dimension.

- Finding an analytical solution might be computationally expensive or infeasible.
- We can use some **numerical method** to find an **approximate solution**.
- In PS1 we used implicit methods– in this exercise we will use an **explicit method**.

Implicit: calculations involve both unknown and known system quantities.
Explicit: calculations only involve known system quantities.
What are the implications for the computational cost?

Finite Difference Method (FDM)

(The numerical method we will be using)

- The **main idea** is to approximate the derivatives with finite differences.

  Forward difference:

  $$\frac{\partial}{\partial x} f \approx \frac{f(x+\Delta x) - f(x)}{\Delta x}$$

  Backward difference:

  $$\frac{\partial}{\partial x} f \approx \frac{f(x) - f(x-\Delta x)}{\Delta x}$$

  Central difference:

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

- We will use the forward difference for the temporal derivatives and the central difference for the spatial derivatives.

Finite Difference Method (FDM)

We will use the forward difference for the temporal derivatives and the central difference for the spatial derivatives.

Our partial differential equation (PDE):

$$\frac{\partial u}{\partial t} = K\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right)$$

Finite Difference Method (FDM)

We will use the forward difference for the temporal derivatives and the central difference for the spatial derivatives.

Our partial differential equation (PDE):

$$\frac{\partial u}{\partial t} = K\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right)$$

Our discretized PDE:

$$\frac{u_{i,j}^{k} - u_{i,j}^{k+1}}{\Delta t} = K_{i,j} \cdot \left(\frac{u_{i-1,j}^{k} - 2u_{i,j}^{k} + u_{i+1,j}^{k}}{\Delta x^2} + \frac{u_{i,j-1}^{k} - 2u_{i,j}^{k} + u_{i,j+1}^{k}}{\Delta y^2}\right)$$

Finite Difference Method (FDM)

We will use the forward difference for the temporal derivatives and the central difference for the spatial derivatives.

Our reordered discretized PDE when we let $\Delta x = \Delta y = h$:

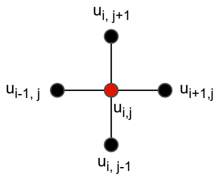
$$u_{i,j}^{k+1} = u_{i,j}^{k} + K_{i,j}\frac{\Delta t}{h^2} \cdot \\ \left(u_{i-1,j}^{k} + 2u_{i,j}^{k} - u_{i+1,j}^{k} - u_{i,j-1}^{k} + 2u_{i,j}^{k} - u_{i,j+1}^{k}\right)$$

Note that in the handout code $h = 1$.

You can try this yourself if you want.
We are also using **Neumann boundary conditions**.

Finite Difference Method (FDM)

We will use the forward difference for the temporal derivatives and the central difference for the spatial derivatives.

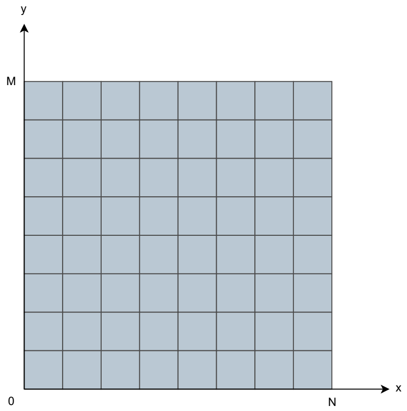


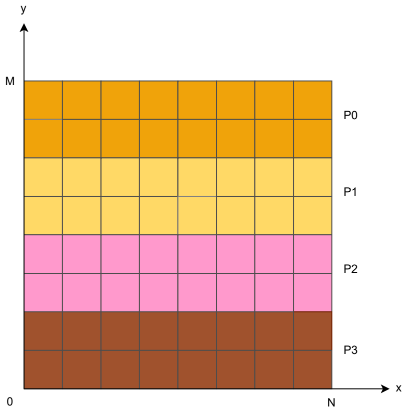
You can try this yourself if you want.

We are also using **Neumann boundary conditions**.

Parallelisation of FDM

Domain decomposition

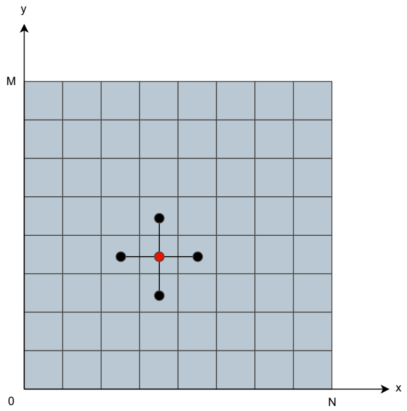


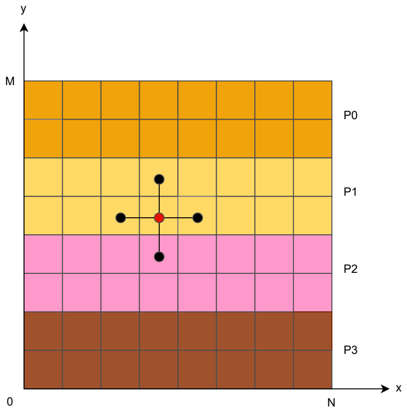


Each process will be responsible for calculations in a sub-grid.

Parallelisation of FDM

Border exchange





What do we do at the sub-grid boundaries?
We need communication between the processes!

The Message Passing Interface (MPI)

- You have gotten an introduction to **MPI** in the main lectures, which will be useful for completing this exercise.
- You were shown an example of **domain decomposition** and **border exchange** in the main lectures, which will also be useful for completing this exercise.
- You can use the functions covered in this weeks lectures, i.e.,

```
MPI_Init
MPI_Finialize
MPI_Comm_size
MPI_Comm_rank
MPI_Send
MPI_Recv
MPI_Bcast
```

Note on `MPI_Send` and `MPI_Recv`

`MPI_Send` and `MPI_Recv` are **blocking** functions and will not return until the buffer is ready to be reused.

The completion of `MPI_Send` indicate that the send buffer can be modified without affecting the data transmitted to the receiver, i.e., the send buffer has been emptied.

The completion of `MPI_Recv` indicate that the data in the receive buffer can be read, i.e., the receive buffer is filled.

They are easy to use, but...

Deadlock

… they are prone to *deadlocks*.

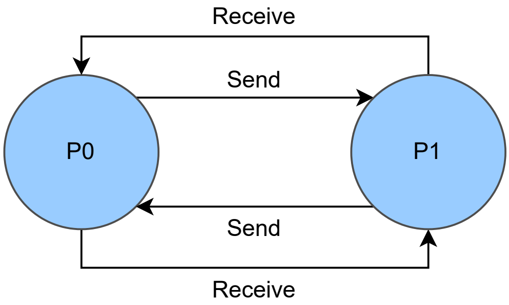
Consider the following sequence:

1. Stand in a circle
2. Extend your right hand to your right neighbor
3. Extend your left hand to your left neighbor when your right hand has been shaken

You could be standing for a while!

This is analogous of what could happen if you don’t pay attention when using blocking send and receive function calls.

Deadlock



Deadlock

```
int rank, sendData[N], receiveData[N];
MPI_Comm_rank( MPI_COMM_WORLD, &rank );

if ( rank == 0 ) {
    MPI_Send( sendData, N, MPI_INT, 1,
        0, MPI_COMM_WORLD );

    MPI_Recv( receiveData, N, MPI_INT, 0,
        0, MPI_COMM_WORLD, MPI_STATUS_IGNORE );
} else if ( rank == 1 ) {
    MPI_Send( sendData, N, MPI_INT, 0,
        0, MPI_COMM_WORLD );

    MPI_Recv( receiveData, N, MPI_INT, 1,
        0, MPI_COMM_WORLD, MPI_STATUS_IGNORE );
}
```

Avoiding deadlock

```
int rank, sendData[N], receiveData[N];
MPI_Comm_rank( MPI_COMM_WORLD, &rank );

if ( rank == 0 ) {
    MPI_Send( sendData, N, MPI_INT, 1,
        0, MPI_COMM_WORLD );

    MPI_Recv( receiveData, N, MPI_INT, 0,
        0, MPI_COMM_WORLD, MPI_STATUS_IGNORE );
} else if ( rank == 1 ) {
    MPI_Recv( receiveData, N, MPI_INT, 1,
        0, MPI_COMM_WORLD, MPI_STATUS_IGNORE );

    MPI_Send( sendData, N, MPI_INT, 0,
        0, MPI_COMM_WORLD );
}
```

Avoiding deadlock (alternative)

```
int rank, sendData[N], receiveData[N];
MPI_Comm_rank( MPI_COMM_WORLD, &rank );

MPI_Sendrecv ( sendData, N, MPI_INT, 1, 0, receiveData, N,
    MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE );

MPI_Sendrecv ( sendData, N, MPI_INT, 0, 0, receiveData, N,
    MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE );
```

MPI_Sendrecv

Combines a blocking send and a receive in a single call. The MPI implementation schedules the communication so that the program does not hang or crash.

```
MPI_Sendrecv(
    void *sendBuffer,
    int sendCount,
    MPI_Datatype sendType,
    int destinationRank,
    int sendTag,
    void *receiveBuffer,
    int receiveCount,
    MPI_Datatype receiveType,
    int sourceRank,
    int receiveTag,
    MPI_Comm communicator,
    MPI_Status *status
);
```

Writing to file

In the FDM code we are writing to file at regular time step intervals to save the state of the grid.

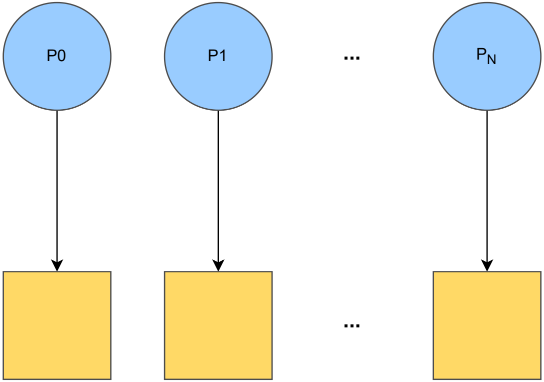
$\rightarrow$ We will write to file many times.

How do we write to file when we have more than one processes?

We have several options. . .

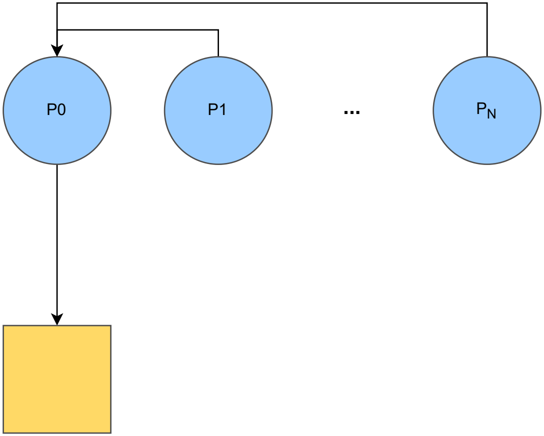
Writing to file

Processes write to multiple files independently.



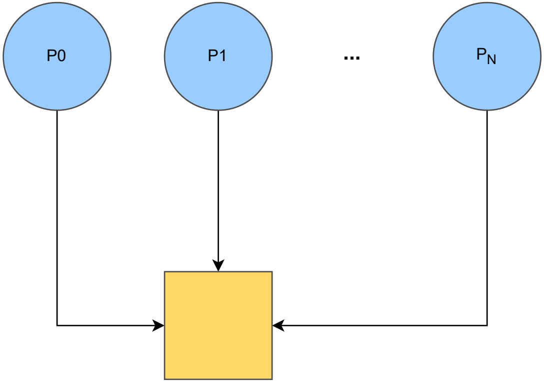
Writing to file

Data is gathered in the root process that writes to a single file.



Writing to file

Processes cooperate and write to a single file $\rightarrow$ MPI I/O!



MPI I/O

- MPI supports I/O!
- Why use MPI I/O?
    - Parallel I/O $\rightarrow$ Performance
    - A single file instead of one file per process
- Writing is like sending and reading is like receiving.
- You will need to
    - **Open** the file
    - **Write** to or **read** the file
    - **Close** the file

Writing to file

Open and close the file

```
void
write_to_file ( void )
{
    File *out = fopen ( 'results.bin', 'w' );
    fclose ( out );
}
```

Writing to file

Open and close the file

```
void
write_to_file ( void )
{
        MPI_File out;
        MPI_File_open ( MPI_COMM_WORLD,
                        'results.bin',
                        MPI_MODE_CREATE | MPI_MODE_WRONLY,
                        MPI_INFO_NULL,
                        &out
    );

    MPI_File_close ( &out );
}
```

Writing to file

`MPI_File_open` **and** `MPI_File_close`

```
MPI_File_open (
     MPI_Comm comm,
     char *filename ,
     int mode,
     MPI_Info info ,
     MPI_File *filehandle ,
);

MPI_File_close (
     MPI_File *filehandle ,
);
```

Each process in the communicator `comm` opens the file identified by `filename`

`info` provides extra information, but if you don't care put MPI_INFO_NULL

`mode` describes the access mode:

MPI_MODE_WRONLY $\rightarrow$ write

MPI_MODE_RDONLY $\rightarrow$ read

MPI_MODE_CREATE $\rightarrow$ create file if it does not exist

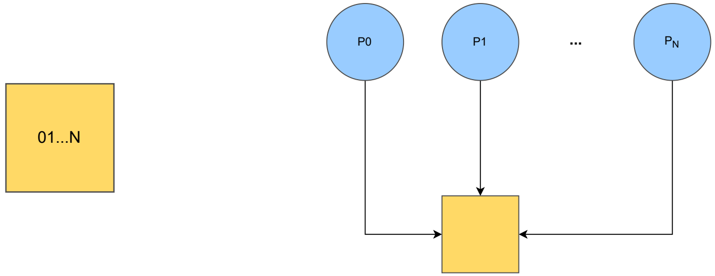
...

Closes the file associated with `filehandle`.

Writing to file

All processes write their ranks to the same file

We want it to look something like this:



Writing to file

`MPI_File_write_at_all`

```
MPI_File_write_at_all (
    MPI_File *filehandle,
    MPI_Offset offset,
    void *buffer,
    int elements,
    MPI_Datatype elementType,
    MPI_Status *status
);
```

Write to the file associated with the `filehandle` at an `offset`

`buffer` is the data we want to write

`elements` and `elementType` is the size of the data we are writing

`status` provides extra information, but if you don't care, just put MPI_STATUS_IGNORE

Writing to file

All processes write their ranks to the same file

```
void
write_to_file ( void )
{
    MPI_File out;
    MPI_File_open ( MPI_COMM_WORLD,
                    'results.bin',
                    MPI_MODE_CREATE | MPI_MODE_WRONLY,
                    MPI_INFO_NULL,
                    &out
    );

    MPI_Offset offset = rank * sizeof(int);

    MPI_File_write_at_all ( out, offset, &rank,
                            1, MPI_INT, MPI_STATUS_IGNORE );
}
```

Your tasks

- Initialize and finalize the MPI environment
- Broadcast program arguments
- Decide on how to divide the grid into sub-grids and take care of memory allocation and domain initialization for each process
- Perform calculations in a sub-grid
- Communicate border values
- Handle program output with MPI I/O

Extra resources

Documentation (and nice examples): RookieHPC

Debugging: tmpi