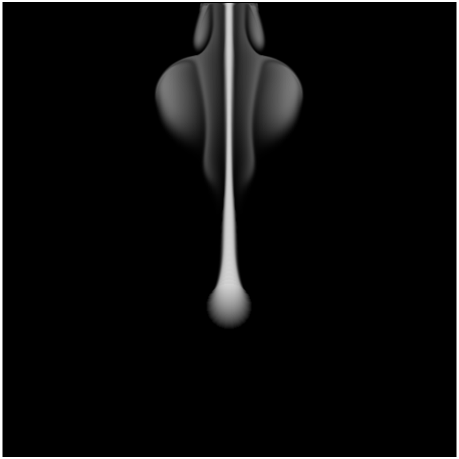
Recitation 2

Announcements

- Assignment 2 is huge. If you don’t start early, you will fail.
- It’s also graded, and mandatory. The grade will count towards your final grade, and you must get a score above a certain threshold to pass.
- Some/most recitations will be moved to the friday lecture slot, check It’s learning.

Computational fluid dynamics



Linear algebra

- We will only look at a small part of the problem, solving a set of linear equations, e.g.

$$A\mathbf{x} = \mathbf{b}$$

- Gaussian elimination can be used to solve such systems. However, if the matrix gets big enough, it is to slow.
- *Iterative* methods can be used instead, they don't allways give the correct answer, but they typically work well for the matrices that show up in many important applications, including this..

Iterative methods, example

Given the linear system $\boldsymbol{A}\mathbf{x} = \mathbf{b}$ with:

$$A = \begin{bmatrix} 8 & -2 \\ 5 & 10 \end{bmatrix} \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \mathbf{b} = \begin{bmatrix} 24 \\ 20 \end{bmatrix}$$

We can rewrite it, expressing each unknown only in terms of the others:

$$x_1 = 0.25x_2 + 3$$

$$x_2 = -0.5x_1 + 2$$

We can now make an initial guess, $\mathbf{x}^{(0)}$, and use that to calculate a new version $\mathbf{x}^{(1)}$, then we can use $\mathbf{x}^{(1)}$ to calculate $\mathbf{x}^{(2)}$ and so on.

Iterative methods, example

If we start with $\mathbf{x}^{(0)} = [1, 1]^T$ we get

Iteration 1:

$$x_1^{(1)} = 0.25 * 1 + 3 = 3.25$$

$$x_2^{(1)} = -0.5 * 1 + 2 = 1.5$$

Iteration 2:

$$x_1^{(2)} = 0.25 * 1.5 + 3 = 3.375$$

$$x_2^{(2)} = -0.5 * 3.25 + 2 = 0.375$$

Iteration 2:

$$x_1^{(3)} = 0.25 * 0.375 + 3 = 3.094$$

$$x_2^{(3)} = -0.5 * 3.375 + 2 = 0.313$$

Iteration 3:

$$x_1^{(4)} = 0.25 * 0.313 + 3 = 3.078$$

$$x_2^{(4)} = -0.5 * 3.094 + 2 = 0.453$$

Iterative methods, example

- And so on... after 10 iterations, you get

  $$\mathbf{x}^{(10)} = [3.11115, 0.4445]^T$$

  which is pretty close to the correct answer

  $$\mathbf{x} = [\frac{28}{9}, \frac{4}{9}]^T$$

  .
- This method is know as the Jacobi method.
- It does not allways converge! However, if A satisfies certain conditions, it will.

Formalization

To make the Jacobi method work, we need to express each variable in terms of all the others (not including itself). In general, we can do that by writing $A = D + R$, where:

$$D = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix} \text{ and } R = \begin{bmatrix} 0 & a_{12} & \cdots & a_{1n} \\ a_{21} & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & 0 \end{bmatrix}$$

Then the solution can be obtained iteratively via:

$$\mathbf{x}^{(k+1)} = D^{-1}(\mathbf{b} - R\mathbf{x}^{(k)})$$

Differential equations

In our case, the linear system we're trying to solve arises from the Poisson equation, so ***A*** will allways have a special pattern, like this:

$$A = \begin{bmatrix} 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 4 & 0 & 0 & -1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 4 & -1 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & -1 & 4 & -1 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & -1 & 4 & 0 & 0 & -1 \\ 0 & 0 & 0 & -1 & 0 & 0 & 4 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 \end{bmatrix}$$

Sparse matrices

We can exploit this pattern. Consider the equation for the new version of x_6 if we have a 16×16 matrix like this:

$$x_6 = \frac{1}{4} * (x_2 + x_5 + x_7 + x_{10} + b_6)$$

Now, if we organize the xs into a grid:

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \dots \\ x_{16} \end{bmatrix} \rightarrow \begin{bmatrix} x_1 & x_2 & x_3 & x_4 \\ x_5 & x_6 & x_7 & x_8 \\ x_9 & x_{10} & x_{11} & x_{12} \\ x_{13} & x_{14} & x_{15} & x_{16} \end{bmatrix}$$

we notice that x_6 depends upon its four neighbours.

Sparese matrices

This is the case for all the xs. If we rename the variables (as well as the bs) based on their coordinates, we get the equation:

$$x_{i,j} = \frac{1}{4} * (x_{i+1,j} + x_{i-i,j} + x_{i,j+1} + x_{i,j-1} + b_{i,j})$$

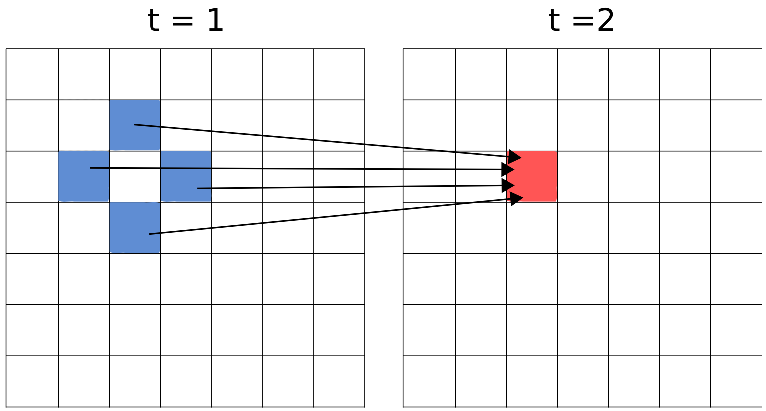
Along the edges and in the corners, we should only use the 3/2 neighbours we have.

Pseudo-code

Psuedo-code for the implementation of the Jacobi method for these kinds of matrices would therefore be:

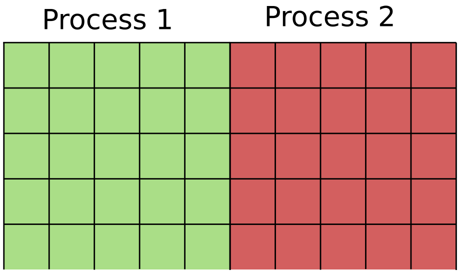
```
for(i = 0 to N)
  for(j = 0 to N)
    if(alongEdge(i,j)
      //Handle special case
    else
      Xnew[i][j] = 0.25(X[i+1][j]+X[i−1][j]+
                        X[i][j+1]+X[i][j−1]+
                        b[i][j]);
```

Pictoraly



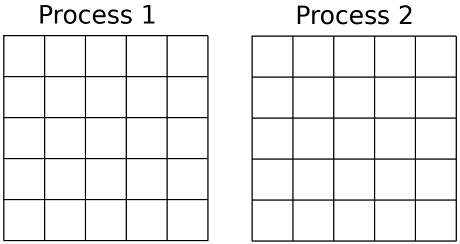
Parallelization

- Each element of the grid is independent of the others, so they can be computed in parallel, each thread/process can be assigned a part of the grid.
- We'll need some kind of synchronization at end of each timestep, though.

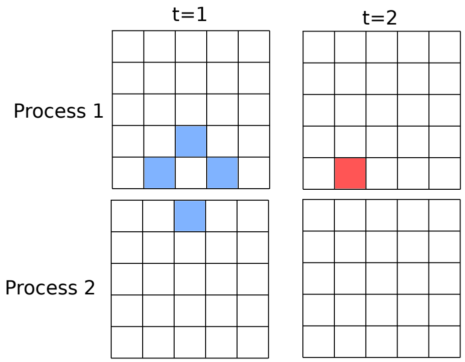


Parallelization, MPI

- In a shared memory model, all the threads could just use the same big array for the grid.
- With distributed memory models like in MPI, each process has it’s own array, to store its part of the grid.

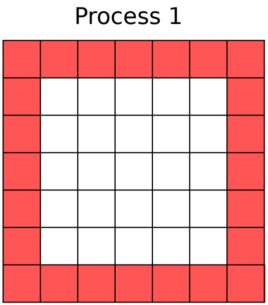


Internal boundaries

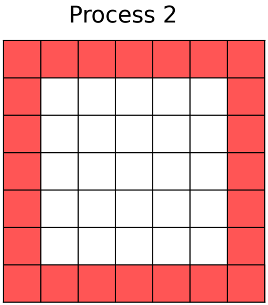


- When we update elements along the edge of the grid, we'll need data from the neighbouring process.
- In a shared memory model, we could just access this memory directly. With MPI, we must send/receive it explicitly.

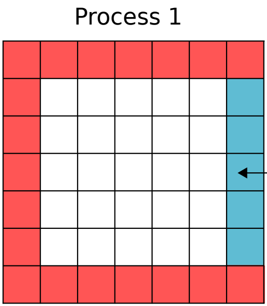
The halo

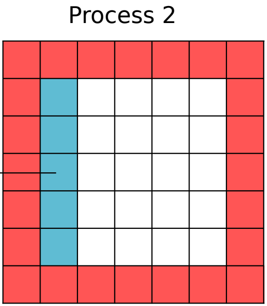


- To simplify this, we'll add an extra layer, a *halo*, around each process part of the grid.



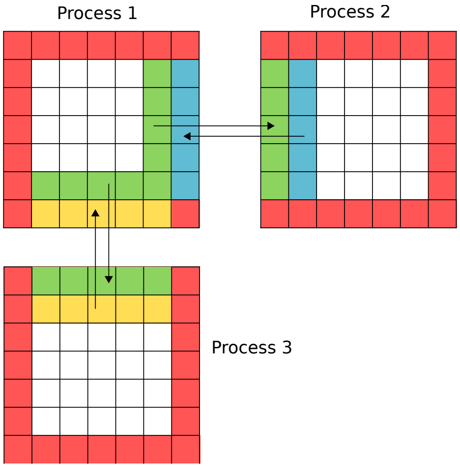
Border exchange





Before each update, we receive our neighbours borders, and store them in the halo.

Border exchange



Naturally, we'll have to send our own, possibly in multiple directions.

Your task

- You should implement a MPI parallelized Jacobi solver.
- You’ll be given code for the whole CDF application, you should only work on the jacobi part.
- To make it easier, I’ve allready included some code, you’ll basically have to complete a few functions.

Implementaion details

- The entire program consists of several files, you'll only need to look at/modify *main.c*, *global.h*, and *jacobi.c*.
- *main.c* contains the main method, where som MPI initialization is done, which you'll need to complete. It also contains some global variables which you might need to add/modify.
- *global.h* Just makes the global variables of *main.c* visible in other files.
- *jacobi.c* This is where the actuall work is done.

Initialization

- Most of the initialization is already done. However, you'll need to create and commit datatypes for the message passing.
- Two datatypes have already been declared, `border_row_t` and `border_col_t`, you might need more.
- You should create and commit the types in `commit_types()`.

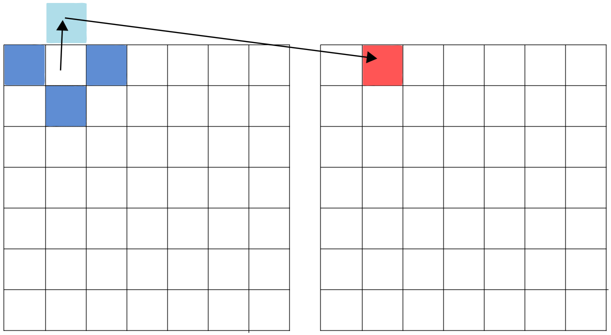
Solving the equation

- The `jabobi()` function solves the linear system, by calling the `jacobi_iteration()` function in a loop.
- In addition, before the iteration you'll need to distribute the ***b***s to the different processes, and at the end, gather the results at rank 0.

Solving the equations

- We’ll use two modifications to the plain Jacobi method described above.
- When we update along the borders, we’ll replace those variables missing with the current version of the variable we’re updating.
- We should subtract, not add, the bs.
- You should use 0 as the inital guess.

Edges

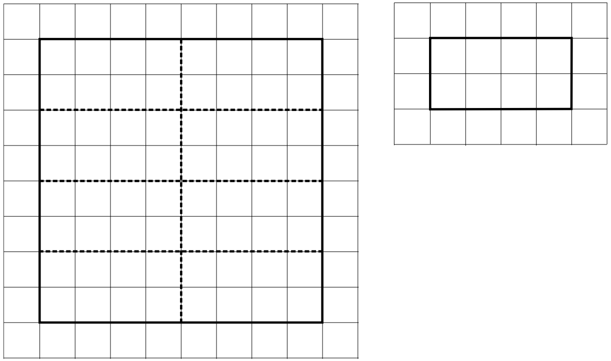


If we try to access a element outside the array, we use the central one instead.

Global variables

- The *x*s are stored in the array `pres` (for pressure)
- Each process' local copy of it's part is stored in `local_pres` and `local_pres0` (two arrays needed for the iteration).
- `diverg` and `local_diverg` (divergence) stores the ***b***s.
- All these arrays, except `local_diverg` have borders. For the global arrays, these are needed for other parts of the calculation, and you should not modify them. For the local arrays, they are needed fo the border exchange.

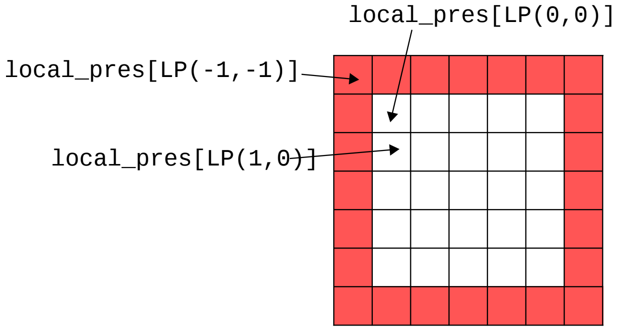
Arrays



`pres` and `local_pres` for an 8×8 system with 8 processors, each working on a 2×4 subdomain.

Implementation details

- Due to the halos, indexing in the `local_pres` can be tricky.
- The `LP` macro can make it simpler, note that its indexing is relative to the interior.



Practicalities

- `make fluid` will compile the program.
- `make run` will run it, (using mpirun and qrsh)
- The arguments to the program is the number of iterations (of the whole simulation, the jacobi part is just one step, and should allways run for 100 iterations), and the size.

Testing

- The program dumps the density of the fluid (which we don’t touch directly) to a .bmp image at the end.
- I’ll provide the correct output for some combinations of sizes and iterations.
- Identical solutions are not excpected, due to round-off errors, but they should be visualy indistinguishable.

Memory layout

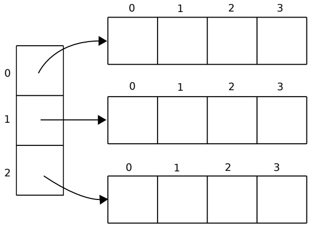
We can create two (or higher) dimensional arrays:

```c
int array[3][3] = { {1,2,3}, {4,5,6}, {8,8,9}};
array[1][0] += 5;
printf("%d\n", array[1][0]); //prints 9
```

2D-arrays with malloc(), method 1

There is no direct way to create 2D arrays with malloc, but we can make an array of pointers to arrays.

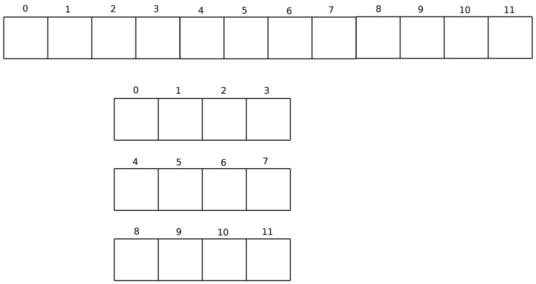
```
int** array = (int**)malloc(sizeof(int*)*N);
for(int i = 0; i < N; i++){
    array[i] = (int*)malloc(sizeof(int)*M);
}
array[1][2] = 5;
```



2D-arrays with malloc(), method 2

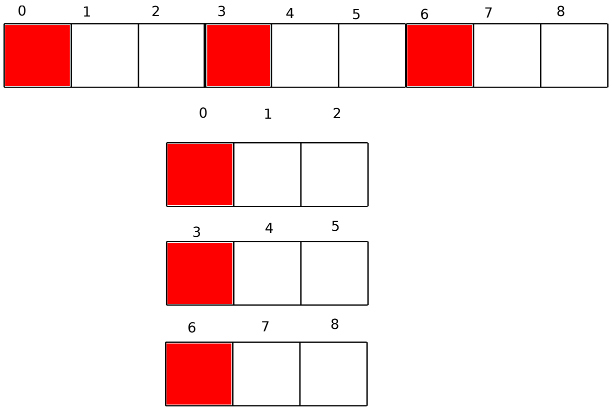
Or we can just create a 1D array, and take care of the colums and rows ourself. This way, we can be sure that the whole thing is in one contigous piece of memory.

```
int* array = (int*)malloc(sizeof(int)*N*M);
int row_width = 4;
array[1 * row_width + 2] = 5;
```



Derived Data Types

- If we want to send the first column of a 3x3 array, we can do it with 3 calls to MPI_Send.
- ...or create a derived datatype, and only use 1 MPI_Send.



Derived Data Types

```
MPI_Type_vector(count,blocklength,stride,oldtype,&newtype);
```

```
MPI_Datatype coltype;
MPI_Type_vector(3,1,3,MPI_INT,&coltype);
MPI_Type_commit(&columntype);
int a[3][3];
MPI_Send(a,1,coltype,dest,tag,MPI_COMM_WORLD);
```

Cartesian communicator

- In this problem, we clearly need to organize the processes in some kind of gird.
- If the number of processes and problem size is known beforehand, we can just hardcode everything.
- But if we have to figure out everything (i.e. which rank is my left neighbour) at runtime, it can be a lot of work.
- Luckily, we can use MPI’s Cartesian communicator.

Cartesian communicator functions

- To create the communicator:
  `MPI_Cart_create(comm_old, ndims, &dims, &periods,reorder,&comm_cart)`
- To find coordinates for a rank:
  `MPI_Cart_coords(comm,rank,maxdims,&coords)`
- To find the rank given the coordinates:
  `MPI_Cart_rank(comm,&coords,&rank)`

More functions

- To find our neighbour in the grid, we can use `MPI_Cart_coords()` followed by `MPI_Cart_rank()`.
- Since this is a common operation, MPI has a single function for it:

  `MPI_Cart_shift(comm,direction,displ,&source,&dest)`
- For `MPI_Cart_create()`, we need the dimensions of the grid (i.e 2x3 for 6 processes). If we don't want to do this ourself, we can use:

  `MPI_Dims_Create(nnodes,ndims,&dims)`

Example