Recitation 2

Announcements

- Assignment 2 is huge. If you don’t start early, you will fail.
- Two parts:
  - Theory, pass/fail, deadline 21.09
  - Code, graded, deadline 28.09
- Late delivery: -10 precentage points per day (without doctors notice or similar).

Amdahl & Gustafson

Amdahl:

$$S = \frac{1}{s + \frac{1-s}{N}}$$

Gustafson:

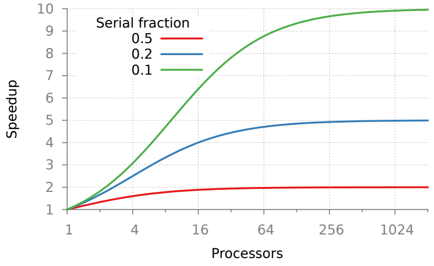
$$S = N + (1 - N)s'$$

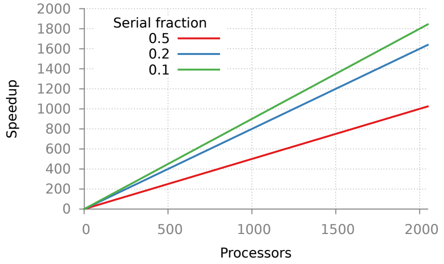
s is the serial fraction, on a serial machine. ***s'*** is the serial fraction, on a prallel machine. That is, if t_s and t_p are the execution times of the serial and parallel parts, then:

$$s = \frac{t_s}{t_s + t_p}$$

$$s' = \frac{t_s}{t_s + \frac{t_p}{N}}$$

Amdahl & Gustafson



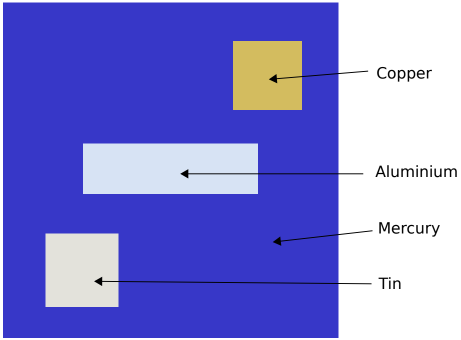


The 2D heat equation

$$\frac{\partial u}{\partial t} - \alpha\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) = 0$$

- Where $u(x, y, t)$ is the temerature at location x, y at time t.
- And α is the thermal diffusivity (a material constant).
- In this assignment we will solve this equation numerically.

The simulated system



- Initial conditions: Copper 60C, Tin 60C, Aluminium 100c, Mercury 20C
- Aluminium is kept at 100C for first half (with external heater)
- Boundary is constant at 10C

Discretized heat equation

Derivatives can be approximated like this:

$$\frac{df}{dx} \approx \frac{f(x+h) - f(x)}{h}$$

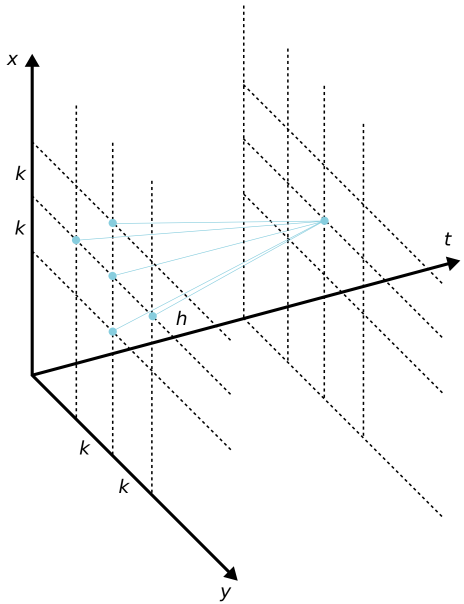
$$\frac{d^2f}{dx^2} \approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}$$

If we apply this to the heat equation, and rearrange, we get:

$$\begin{aligned} u(x, y, t+h) = u(x, y, t) + \frac{\alpha h}{k^2}(&u(x+k, y, t) + u(x-k, y, t) \\ &+ u(x, y+k, t) + u(x, y-k, t) - 4u(x, y, t)) \end{aligned}$$

Thus, if we know $u(x, y, 0)$, that is, the temperature at time 0, we can calculate how the temperature evolves directly.

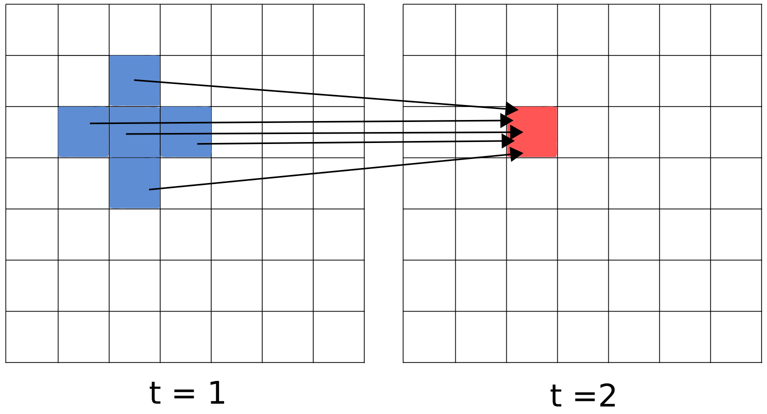
Figure



In pseudocode

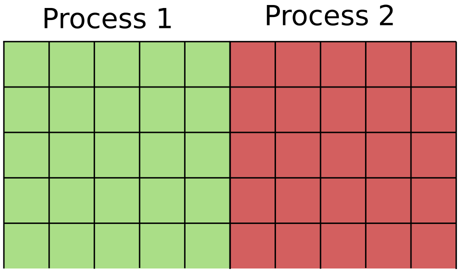
```
temp[x][y][t+1] = temp[x][y][t] +
                  material_const[x][y] *
                  (temp[x+1][y][t] +
                   temp[x-1][y][t] +
                   temp[x][y+1][t] +
                   temp[x][y-1][t] -
                   4 * temp[x][y][t])
```

Pictoraly



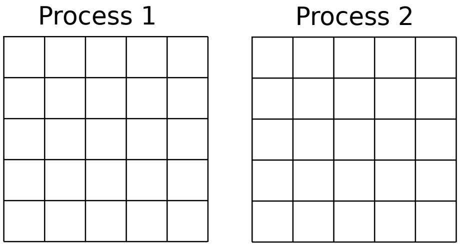
Parallelization

- 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.

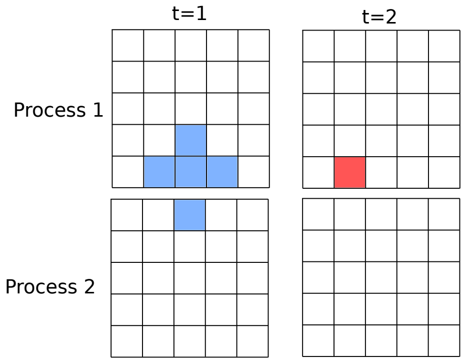


Parallelization, MPI

- 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.

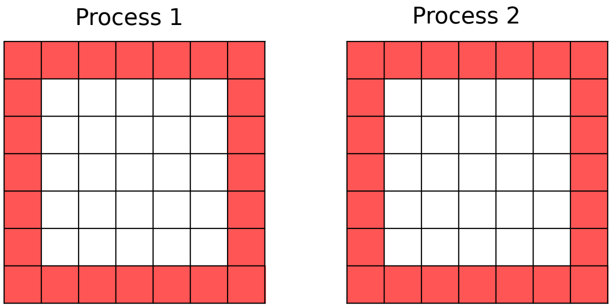


Internal boundaries



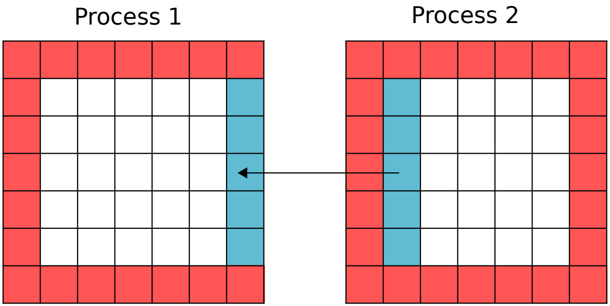
- When we update elements along the edge of the grid, we'll need data from the neighbouring process.
- In a shared memory model, we could just access this memory directly. With MPI, we must send/receive it explicitly.

The halo



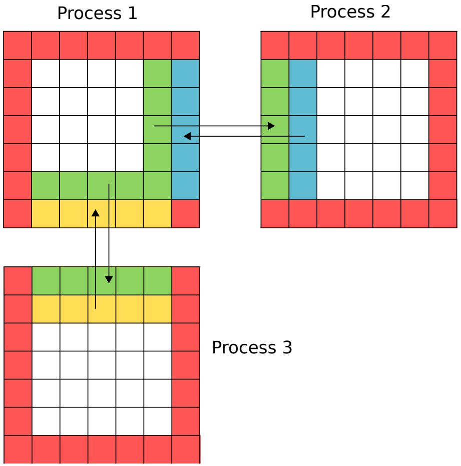
- To simplify this, we'll add an extra layer, a *halo*, around each process part of the grid.

Border exchange



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

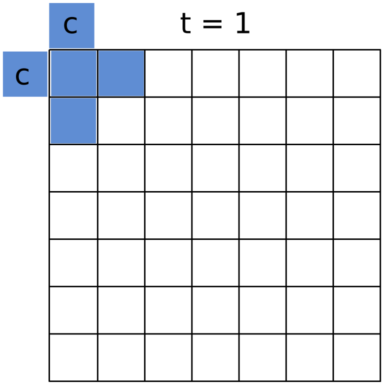
Border exchange



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

Global boundaries

- For accesses outside the global grid, we will just use a constant value ***c***.
- (Dirichelt boundary conditions, for the mathematically inclined).



Your task

- You should implement a MPI parallelized heat equation solver.
- You'll be given a framework, with some MPI setup. You'll need to complete a few functions to make it work.
- You'll also be given a separate serial version.

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_vector_types()`.

Distributing and gathering

- The variables `temperature` and `material` are for the entire simulation domain, and exists only on rank 0. They store the temperatures and material constants, respectively. These are filled with initial values on rank 0 in `init_temp_material()`
- The variables `local_temp` and `local_material` store each process part of the temperature fields and material constants. `local_temp` is initialized in `init_local_temp()` for the constant global boundary condition.
- The function `scatter_temp()` and `scatter_material` should send each rank its part of the global temperature and materail arrays.
- The function `gather_temp()` should gather the local temperature arrays to the global one.

Exchange

- In `border_exchange()` each process should send the pixels along its borders to its four neighbours in the grid.

Computation

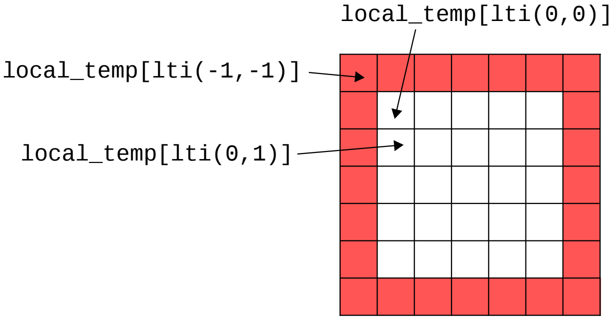
- You should just do the computation shown in the pseudocode above. You can also look at the serial version.
- We don't keep all the timesteps, just the two most recent. We then keep overwriting the oldest of these.
- That is why we have two `local_temp` arrays, we compute `local_temp[1]` from `local_temp[1]`, then `local_temp[0]` form `local_temp[1]` and so on.

Indexing

- All the 2D arrays are implemented with large 1D arrays.
- `GRID_SIZE` stores the size of the entire temperature and material array, `local_grid_size` stores the size of each process part of these grids (not including the halo)

Indexing

- The functions `ti()`, `mi()`, `lti()`, `lmi()` return the linear index given x and y coordinates, accounting for the halo.



Practicalities

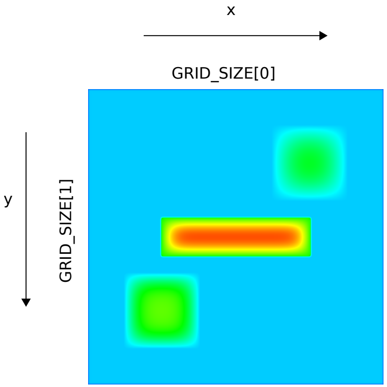
- `make` will compile the program.
- `make run` will run it, (using mpirun) (Currently with just 1 process, remember to change this)
- The program will run for 10000 timesteps, and dump the temperature field to a image file every 500 timesteps. (You can change these values for debugging).

Testing

- The serial implementation should be considered correct, and your parallel version should produce the same results.
- You may assume that the number of processors used will be a power of 2 (e.g. 2,4,8,16...)

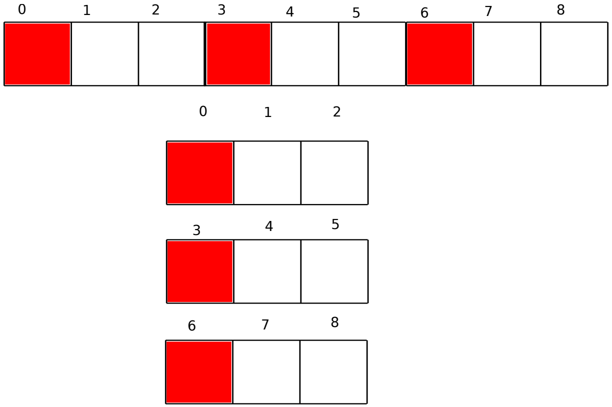
BMP

- .bmp images are stored "upside down"
- This has been corrected for in this assignment.



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:

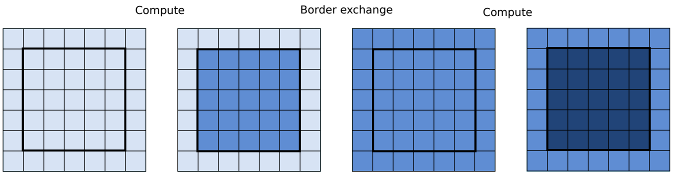
  ```
  MPI_Dims_Create(nnodes,ndims,&dims)
  ```

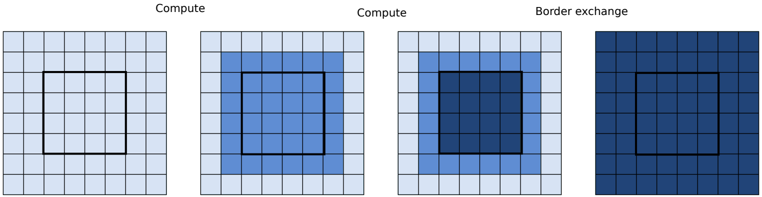
Thicker borders

- The following slides provides explanation for the optinal problem 2.
- By doing problem 2, you may get a score above 100%, this will be transfered to future assignments.

Avoiding communication

If (the startup cost of) communication is expensive, it may be beneficial to use thicker halos, and do multiple iterations between each border exchange:





Thicker borders

- The main changes you need to make are:
- Set `BORDER` to 2 (or more), change the loop in main to do 2 (or more) computations and one border exchange per iteration (and fewer iterations).
- Handle corners, when the border thickness is 1, we don’t need the values in the corners of the halos, with thicker borders, they are needed. By carefully ordering the border exchange, and using apropriately sized MPI datatypes, you can do this while still only communicating with 4 neighbours.
- The `local_material` arrays must be larger, since you need the material constants for parts of the halo region.
- Other changes may be needed.
- The output should be the same as for the serial version, and the version with a border of 1.