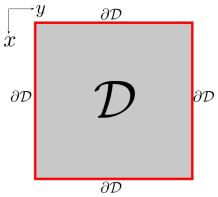
Logistics

- Assignment 1 released today! Available here
- Assignment 1 is due on Monday, 15th April 2024, 11:55PM (right after semester break)
- Read assignment specification carefully and ask questions on the forum if required
- Next week lab there will be a 1-hour session on Q&A for the assignment
- Start early!

Heat Equation in 2D (problem definition)

- given an squared metal sheet and known temperature distribution at the sheet edges, the 2D Heat Equation models the unknown temperatures in the middle
- let us denote by $\mathcal{D}$ the metal sheet (i.e., the domain of the equation) and by $\partial\mathcal{D}$ the edges of the metal sheet (i.e., the boundary of $\mathcal{D}$)
- the 2D Heat Equation is a PDE (Partial Differential Equation); its solution is an unknown bivariate function $u(x,y) : \mathcal{D} \rightarrow \mathbb{R}$ such that:

$$\begin{cases} \nabla^2 u = 0 & \text{in } \mathcal{D} \\ u = u_{\partial\mathcal{D}} & \text{on } \partial\mathcal{D} \end{cases}$$



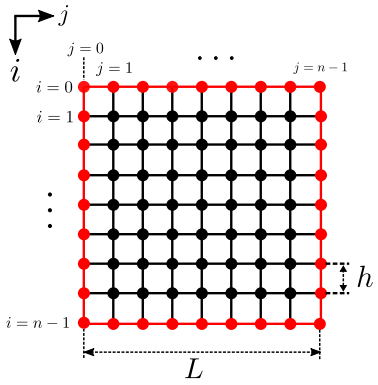
with $u_{\partial\mathcal{D}}(x,y) : \partial\mathcal{D} \rightarrow \mathbb{R}$ being the known temperature distribution at $\partial\mathcal{D}$

- $\nabla^2(\cdot)$ is the Laplacian operator, defined as (sum of 2nd partial derivatives):

$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}$$

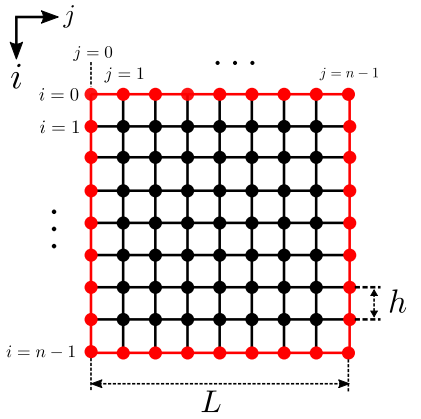
Heat Equation in 2D (finite difference discretization I)

- the 2D Heat Equation PDE is a continuous problem, we have to discretize it so that it can be solved in a computer
- we will use the most simple method to discretize it, the finite difference method
- transforms the PDE into a linear system by approximating the partial derivatives (covered later on) on a 2D grid of points
- for simplicity, we consider a square grid (i.e., we have the same # of points in each space dimension)
- we denote as h the grid size; distance among two consecutive points in either the vertical or horizontal space dimension



Heat Equation in 2D (finite difference discretization II)

- we use a 2D labeling of the grid points, with $0 \leq i \leq n-1$, $0 \leq j \leq n-1$
- we denote by $u_{i,j}$ the approximate value of u at the point labeled (i,j)
- mathematically, $u_{i,j} \approx u(ih, jh)$
- the value of $u_{i,j}$ at the boundary points (in red) is known, HOWEVER the value of $u_{i,j}$ at the interior nodes (in black) is unknown
- we thus have $(n-2)^2$ unknowns
- how can we formulate a discrete problem to determine the value of these unknowns? (next slide)



Heat Equation in 2D (finite difference discretization III)

- at each interior grid point, we approximate the (partial derivatives in the) PDE using finite difference formulas (these are derived from truncated Taylor series)
- we will use the central finite difference formula for the second partial derivatives
- for an univariate function, $f(x)$, the central difference formula is defined as

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

- applying this formula to $\nabla^2 u = 0$ *at each interior grid point* (x_i, y_j), we end up with

$$\nabla^2 u(x_i, y_j) = \frac{\partial^2 u}{\partial x^2}(x_i, y_j) + \frac{\partial^2 u}{\partial y^2}(x_i, y_j) = 0 \approx$$

$$\frac{u(x_i+h, y_j) - 2u(x_i, y_j) + u(x_i-h, y_j)}{h^2} + \frac{u(x_i, y_j+h) - 2u(x_i, y_j) + u(x_i, y_j-h)}{h^2} = 0 \rightarrow$$

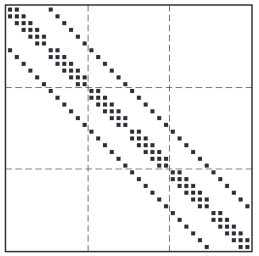
$$\frac{-4u_{i,j} + u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}}{h^2} = 0 \quad \text{with } 1 < i < n-1 \text{ and } 1 < j < n-1$$

Heat Equation in 2D (linear system after discretization)

- relabeling $u_{i,j}$ as x_k, with $k = (i-2)n + j - 2$, and $1 < i < n-1$, $1 < j < n-1$, then the previous expression can be re-written as (to-think: why?):

$$-4x_k + x_{k+1} + x_{k-1} + x_{k+n} + x_{k-n} = 0 \quad \text{with} \quad k = 0, 1, \ldots, (n-2)^2 - 1$$

- this is a linear system $Ax = b$, where $A \in \mathbb{R}^{(n-2)^2 \times (n-2)^2}$ is a sparse matrix (it has non-zeros only in 5 diagonals) and $b \in \mathbb{R}^{(n-2)^2}$ is zero for all interior points BUT those (interior points) which are adjacent to the boundary points (to-think: why?)



A with $(n-2)^2 = 36$

- let us now *cleverly* implement Jacobi iteration in order to solve this linear system by efficiently exploiting its particular structure (next slide)

Sequential Jacobi iteration for FD-discretized 2D Heat Equation

```
... // set boundary of u_new/u to b
... // init interior points of u_new/u
for (iter = 0; iter < max_iter; iter++)
{
 for (i = 2; i < n-1; i++)
   for (j = 2; j < n-1; j++)
     u_new[i][j] =
       0.25*(u[i-1][j]+u[i+1][j]+
             u[i][j-1]+u[i][j+1]);
 for (i = 2; i < n-1; i++)
   for (j = 2; j < n-1; j++)
     u[i][j] = u_new[i][j];
}
```

Questions:

- are we explicitly storing entries of A?
- are we explicitly storing the zeros of b?

- we do NOT store A and b into arrays `a[][]` and `b[]` as before (why not?)
- instead, we use two 2D arrays of the same size as the grid, i.e., of size $n \times n$, namely `u[][]` and `u_new[][]`
- on the interior points, `u[][]` and `u_new[][]` hold respectively the values of $x^{(k)}$ and $x^{(k+1)}$ (i.e. Jacobi iterates)
- on the boundary nodes, `u[][]` and `u_new[][]` are both initialized to the known boundary values

Parallel Jacobi iteration for FD-discretized 2D Heat Equation

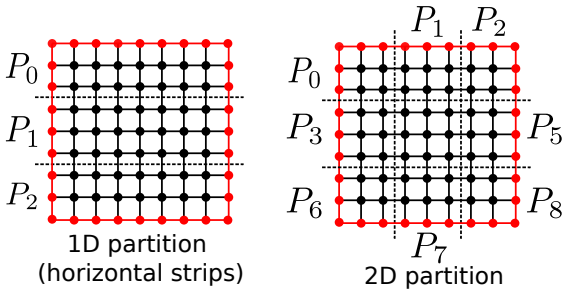
- consider a (naive!) static mapping of a single interior grid point per process (i.e., we have as many interior grid points as processes)
- both `u[][]` and `u_new[][]` are now partitioned/mapped to the processes (i.e., *not replicated* as before with dense matrices)
- the message-passing code for a process not in contact with the boundary looks like: (exercise: *how would it look like for processes in contact with the boundary?*)

```
... // Init u_{i+1,j}, u_{i-1,j}, u_{i,j+1}, u_{i,j-1}
for (iter = 0; iter < max_iter; iter++)
{
  u_new_{i,j} = 0.25*(u_{i+1,j}+u_{i-1,j}+u_{i,j+1}+u_{i,j-1});
  send(&u_new_{i,j}, 1, P_{i-1,j});
  send(&u_new_{i,j}, 1, P_{i+1,j});
  send(&u_new_{i,j}, 1, P_{i,j-1});
  send(&u_new_{i,j}, 1, P_{i,j+1});
  recv(&u_{i-1,j}, 1, P_{i-1,j});
  recv(&u_{i+1,j}, 1, P_{i+1,j});
  recv(&u_{i,j-1}, 1, P_{i,j-1});
  recv(&u_{i,j+1}, 1, P_{i,j+1});
}
```

- communication/synchronization is "local" (each process only synchronizes with nearest neighbours)
- is this algorithm dead-lock free?

Partitioning

- better to feed each processor with larger workload
- regular 2D data (grid) can be either partitioned one-dimensionally (into horizontal or vertical strips) or two-dimensionally (into blocks)



- if p is the # of processors, and $n \times n$ is the grid size, the work per process (assuming equal sized partitions) is proportional to $\frac{n^2}{p}$ for both strategies (why?)
- BUT ... *communication differs among the two approaches*! (next slide)

Modelling communication: strip versus block partition

- strip communication time:
  (2 stages: bottom-top, top-bottom)

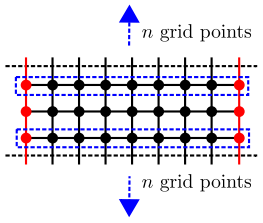
$$t_{\text{comm}}^{\text{strip}} = 2(t_s + nt_w)$$

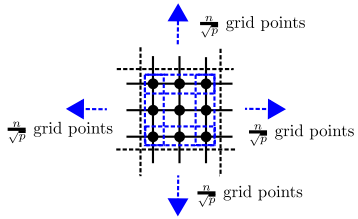
- block communication time:
  (4 stages)

$$t_{\text{comm}}^{\text{block}} = 4(t_s + \frac{n}{\sqrt{p}}t_w)$$

Assumptions:

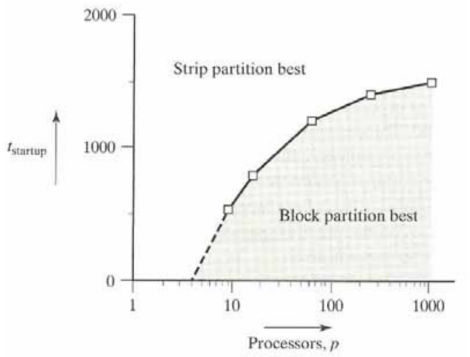
- neglect number of links and t_h
- the nodes can only send/recv single message at a time
- the messages of each stage can be delivered in parallel (e.g., mesh network topology)





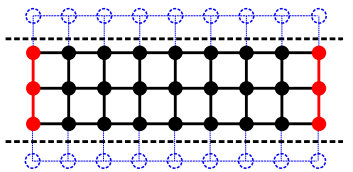
Strip vs Block Partition Cross-Over

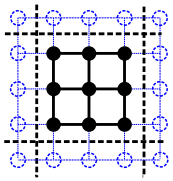
- according to our model, which partition is better (i.e., leads to less overhead)?
- $t_{\text{comm}}^{\text{block}} > t_{\text{comm}}^{\text{strip}}$ if and only if $t_s > n\left(1 - \frac{2}{\sqrt{p}}\right)t_w$
- let us instantiate the model with $n^2 = 1024$, $t_w = 50t_s$
- for different values of p (x-axis), the curve below provides the cross-over t_s (y-axis)



Ghost Layer of Points (aka Halo)

- most grid-based parallel codes store on each process extra layer(s) of adjacent grid points owned by neighbouring processes
- the grid points in these layers are referred to ghost points, and the set of all of these layers as halo region of the local portion of the grid
- the halo region is used to hold data values received as a result of the communication with nearest neighbours
- it is not actually needed, but significantly eases code implementation
- for the central finite difference formula a single layer of ghost points suffices (figure)


1D partition


2D partition

Avoiding Deadlocks

- the algorithm that we saw before to perform nearest neighbour exchanges was NOT deadlock-free
- two dead-lock free algorithms for 1D partitioning in horizontal strips are provided below (for processes not in contact with the top nor bottom boundary edges)
- note halo usage

```
me=process_rank_id()
if ((me%2) == 0) { // even process
 send(&u_new[1][1], n-2, me-1));
 recv(&u[0][1], n-2, me-1));
 send(&u_new[n/p][1], n-2, me+1));
 recv(&u[n/p+1][1], n-2, me+1));}
else {             // odd process
 recv(&u[n/p+1][1], n-2, me+1);
 send(&u_new[n/p][1], n-2, me+1);
 recv(&u[0][1], n-2, me-1);
 send(&u_new[1][1], n-2, me-1);}
```

reorder sends/recvs

```
me=process_rank_id()
isend(&u_new[1][1], n-2, me-1));
isend(&u_new[n/p][1], n-2, me+1));
irecv(&u[0][1], n-2, me-1));
irecv(&u[n/p+1][1], n-2, me+1));
waitall();
```

non-blocking sends/recvs

- other solutions include: (1) buffered sends (`MPI_BSend`); (2) combined send/recvs: `MPI_Sendrecv`, which are guaranteed to be deadlock free

Early termination

- in a parallel setting, we have to ensure that all processes finish the iterative solver loop at once, i.e., at the same iteration (otherwise deadlock may occur)
- so far we have guaranteed that by always performing a fixed # of Jacobi iterations
- iterative solvers typically may terminate early if, e.g., the distance among two consecutive iterates is "small enough"
- the distance among two vectors x and y can be measured, e.g., using the Euclidean norm $||x-y||_2 = \sqrt{\sum_{i=0}^{n-1}(x_i-y_i)^2}$ or the infinity norm (as in Lab #3)
- in our parallelization of Jacobi solver for 2D Heat Equation, x (i.e., `u[]`) and y (i.e., `u_new[]`) are distributed among processes
- in order to compute the norm in parallel, each processor computes a partial sum locally; then all processes execute an `MPI_Allreduce` (sum) collective communication to reduce the partial sums into a single sum on all processes
- the collective communication guarantees that all processes have the same value for $||x-y||_2$ (up to rounding errors), and thus that early termination happens at once on all processes