MPI Virtual Topologies And User Defined Datatypes

Michael Hanke

Summer School on High Performance Computing

Outline

What You Already Know

- How to start and stop MPI
- How to exchange messages between pairs of processes
- How to avoid dealocks
- How to use communication routines efficiently
- Collective communication and performance issues

What Is Covered in This Module?

- Virtual topologies, that is structuring the communication pattern
- The role of communicators
- MPI derived data types

An Example

Problem

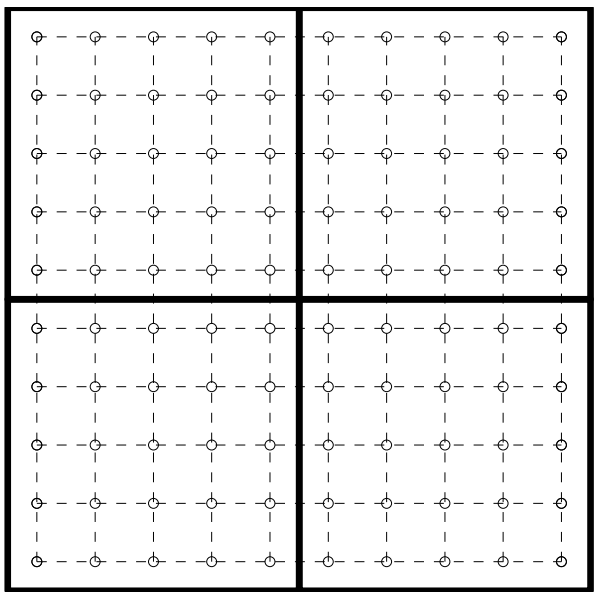
A grid function (or, a matrix) shall be distributed over a set of processes.

- Observation: A grid function inherits a natural topology:

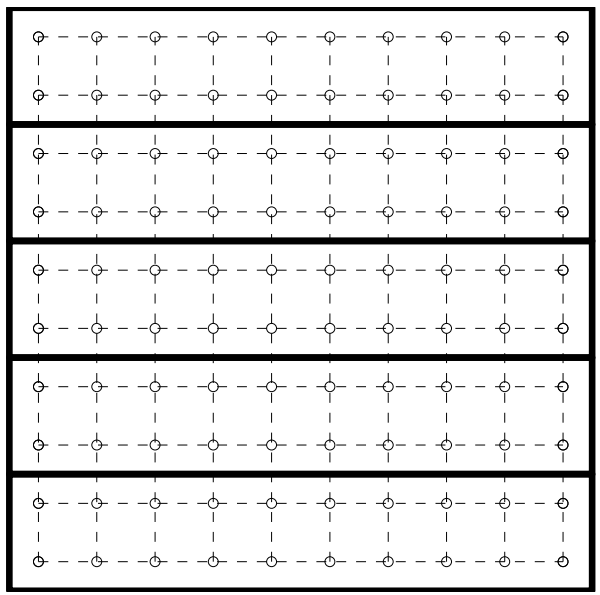
$$u(x, y) \longrightarrow u_{ij}$$

  (north/south/east/west, or left/right/above/below)
- Conclusion: Map it in a similar fashion to the processes, that is processes should be ordered grid-like

An Example (cont)



$P = Q = 2$



$P = 1,\ Q = 5$

MPI Tool

Virtual Topologies

Virtual Topologies

- Convenient process naming
- Naming scheme to fit communication pattern
- Simplifies writing code
- Can allow MPI to optimize communications

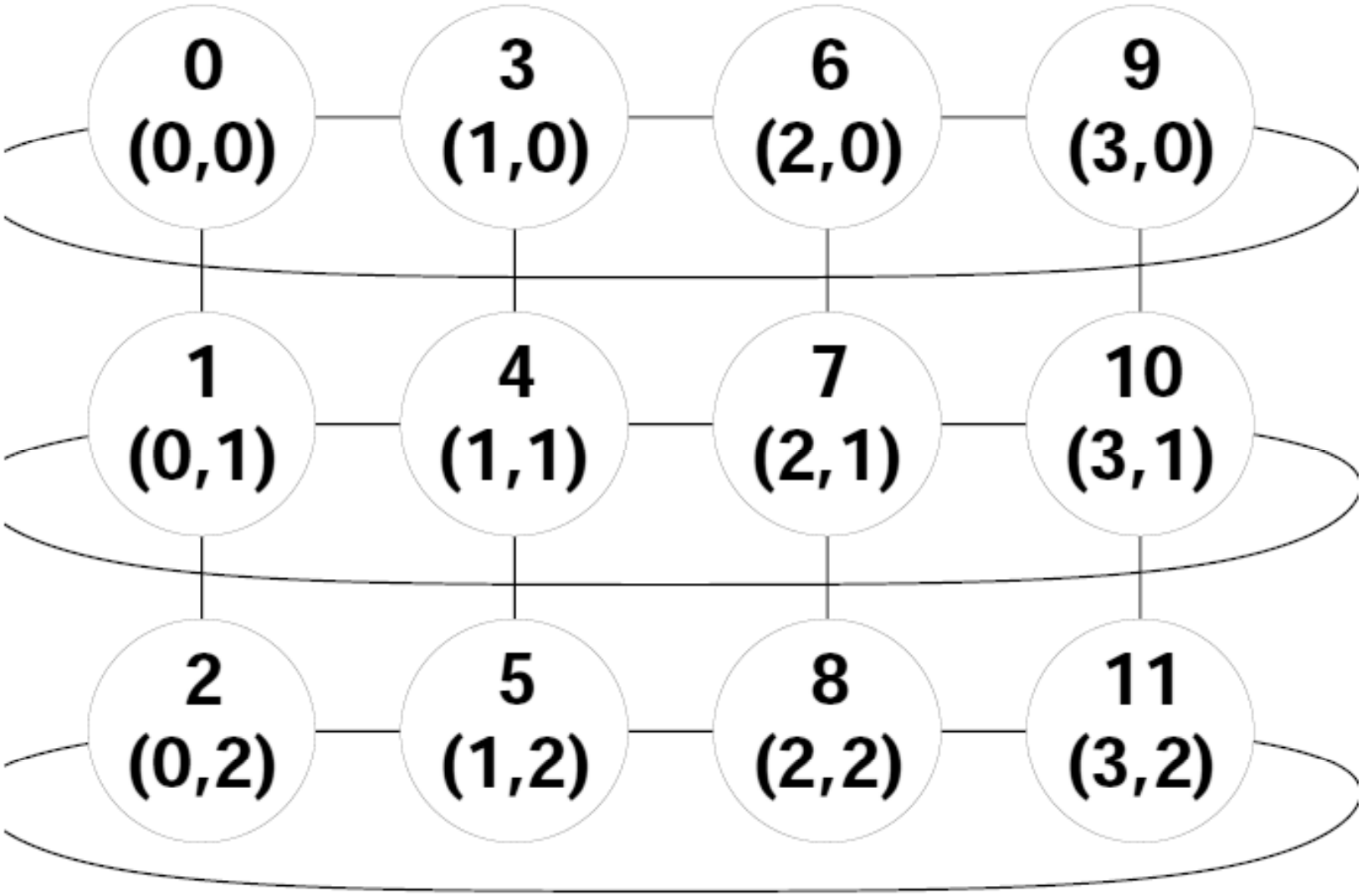
How To Use A Virtual Topology

- Creating a virtual topology produces a new communicator
- MPI provides "mapping functions" between the "serial" enumeration and the virtual topology
- Mapping functions compute processor ranks, based on the topology naming scheme

Topology Types

- Cartesian topology (generalization of a grid function)
  - each process is "connected" to its neighbors in a virtual grid
  - boundaries can be cyclic, or not
  - processes are identified by (discrete) Cartesian coordinates $i, j, k, ...$
- Graph topologies
  - graphs are used to describe communication patterns
  - the most general description of communication patterns
  - not covered here

Example: A 2-dimensional Cylinder



Creating a Cartesian Virtual Topology

C

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims,
            int *periods, int reorder, MPI_Comm comm_cart)
```

Fortran

```
MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder,
                comm_cart, ierror)
INTEGER comm_old, ndims, dims(ndims), comm_cart, ierror
LOGICAL periods(ndims), reorder
```

Creation (cont)

- `comm_old` - the old communicator (often `MPI_COMM_WORLD`)
- `ndims` - dimension of the grid
- `dims` - number of processes in every grid direction
- `periods` - periodic boundaries or not
- `reorder` - allow for a reorder of the ranks in `comm_old`
- `comm_cart` - the new communicator

Example (cont)

```
#define NDIMS 2

MPI_Comm grid_comm;
int dims[NDIMS];
int periods[NDIMS];
int reorder = 1;

dims[0] = 4;
dims[1] = 3;
periods[0] = 1;
periods[1] = 0;
MPI_Create_cart(MPI_COMM_WORLD, NDIMS, dims, periods,
                reorder, &grid_comm);
```

Cartesian Mapping Functions

Mapping process grid coordinates to ranks:

C

```
int MPI_Cart_rank(MPI_Comm comm, int *coords,
                  int *rank)
```

Fortran

```
MPI_CART_RANK(comm, coords, rank, ierror)
INTEGER comm, coords(*), rank, ierror
```

Cartesian Mapping Functions (cont)

Mapping ranks to process grid coordinates:

C

```
int MPI_Cart_coords(MPI_Comm comm, int rank,
                    int maxdims, int *coords)
```

Fortran

```
MPI_CART_RANK(comm, rank, maxdims, coords, ierror)
INTEGER comm, rank, maxdims, coords(*), ierror
```

Cartesian Mapping Functions (cont)

Compute rank of neighboring processes

C

```
int MPI_Cart_shift(MPI_Comm comm, int dir, int disp,
                   int *rank_source, int *rank_dest)
```

- dir gives the coordinate direction while disp is the (positive or negative) displacement
- both rank_source and rank_dest are returned
- If the corresponding process does not exists, MPI_PROC_NULL is returned (may save a test in communications)
- To find out diagonal neighbors, a process must ask a neighbor which neighbors it has

Example: Virtual Topologies in Action

Problem

Compute the sum of the ranks of all processes

Solutions:

1. $s = P(P-1)/2$
2. `MPI_Reduce`
3. Virtual Topologies (Do not use it in practice!)

Example (cont)

Idea:

- Create a ring topology.
- The sum is initialized to the actual rank.
- The individual ranks are sent cyclically and added to the local sum.

Example (cont)

```
#include <stdio.h>
#include <mpi.h>
#define tag 111

int main(int argc, char *argv[]) {
  int i, my_rank, size, left, right;
  int sum, send_buf, rec_buf;
  MPI_Comm ring_comm;
  int dims[1], periods[1], reorder;
  MPI_Status send_status, rec_status;
  MPI_Request request;
/* Initialize MPI */
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
```

Example (cont)

```
/* Set Cartesian topology */
  dims[0] = size;
  periods[0] = 1;
  reorder = 1;
  MPI_Cart_create(MPI_COMM_WORLD, 1, dims, periods,
                  reorder, &ring_comm);
  MPI_Cart_shift(ring_comm, 0, 1, &left, &right);
```

Example (cont)

```
/* Compute global sum */
  sum = 0;
  send_buf = my_rank;
  for (i = 0; i < size; i++) {
    MPI_Issend(&send_buf, 1, MPI_INT, right, tag,
               ring_comm, &request);
    MPI_Recv(&rec_buf, 1, MPI_INT, left, tag,
             ring_comm, &rec_status);
    sum += rec_buf;
    MPI_Wait(&request, &send_status);
    send_buf = rec_buf;
  }
  printf("Proc %d, sum %d, should be %d\n",
         my_rank, sum, size*(size-1)/2);
  MPI_Finalize();
}
```

Example: Do It In 2D

Problem

Create a two-dimensional Cartesian topology and compute the rank sums by rows or columns.

Example (cont)

```
#define DIM 1

int dims[2], periods[2], reorder;

periods[0] = periods[1] = 1;
reorder = 1;
dims[0] = dims[1] = 0;
MPI_Dims_create(size, 2, dims);

MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods,
                reorder, &torus_comm);
MPI_Cart_shift(torus_comm, DIM, 1, &left, &right);
```

Cartesian Partitioning

- The previous problem could be solved more efficiently if collective communications can be used (`MPI_Allreduce` row or column wise).
- Collective communications include *all* processes belonging to the same communicator
- Often, collective operations are only needed over *subsets* of processes belonging to a certain communicator
- Two possibilities:
  - Use point-to-point communications for emulating "partial" collective communications
  - Create communicators containing only those processes involved in collective communication
    `MPI_Comm_create, MPI_Comm_split`

Cartesian Partitioning (cont)

- Cut a grid into "slices"
- A new communicator is produced for each slice
- Each slice can then perform its own collective communications
- This procedure is accomplished by `MPI_Cart_sub` (a special case of `MPI_Comm_split`)

Cartesian Partitioning (cont)

C

```
int MPI_Cart_sub(MPI_Comm cart_comm,
                 int free_coords[], MPI_Comm *new_comm)
```

- Both `cart_comm` and `new_comm` have associated Cartesian topologies
- If `cart_comm` has dimensions $d_0 \times d_1 \times \cdots \times d_{n-1}$, then the dimension of `free_coords` is n
- If `free_coords[i]` is 0, then the `i`th coordinate is fixed for the constuction of the new communicators

Example: Cartesian Partitioning

Problem

Implement the previous example using Cartesian partitioning.

Example (cont)

```
#include <stdio.h>
#include <mpi.h>
#define DIM 1

int main(int argc, char *argv[]) {
  int sum, my_rank, size;
  MPI_Comm torus_comm, slice_comm;
  int dims[2], periods[2], reorder;
  int free_coords[2];
/* Initialize MPI */
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
```

Example (cont)

```
/* Create Cartesian topology */
  periods[0] = periods[1] = 1;
  reorder = 1;
  dims[0] = dims[1] = 0;
  MPI_Dims_create(size, 2, dims);
  MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods,
                  reorder, &torus_comm);
```

Example (cont)

```
/* Create Cartesian partitioning and sum up */
  free_coords[0] = free_coords[1] = 0;
  free_coords[DIM] = 1;
  MPI_Cart_sub(torus_comm, free_coords,
               &slice_comm);
  MPI_Allreduce(&my_rank, &sum, 1, MPI_INT, MPI_SUM,
                slice_comm);

  printf("Proc %d, sum %d\n", my_rank, sum);
  MPI_Finalize();
}
```

An Example Problem

Assume the following definition (C)

```
double A[M][N];
```

Problem

Send the columns of `A` individually!

- Recall: Arrays are stored row-major
- Note: In Fortran, arrays are stored column-major. The following discussion holds true for Fortran if rows and columns are exchanged.

Recall: Send/Receive in MPI

```
MPI_Send(void *data, int count, MPI_Datatype type, ...)
```

- Can be used for sending elements allocated contiguous in memory:

  ```
  MPI_Send(A[i], N, MPI_DOUBLE, ...);
  ```

  sends the `i`-th row of `A`.
- Columns are not contiguous in memory.

Alternatives

1. Send the items one by one:

   ```
   for (i = 0; i < M; i++)
     MPI_Send(A[i][j], 1, MPI_DOUBLE, ...);
   ```

   sends the j-th column. Very slow!
2. Copy the elements to a buffer:

   ```
   double buf[M];
   for (i = 0; i < M; i++) buf[i] = A[i][j];
   MPI_Send(buf, M, MPI_DOUBLE, ...);
   ```

   Note: This is implicitely done if you are using Fortran slices.
3. MPI derived datatypes.
   Note: Definition of your own datatypes during runtime, not at compile time!

Define a Vector Datatype

C

```
int MPI_Type_vector(int count, int blocklen, int stride,
                    MPI_Datatype, element_t, MPI_Datatype *new_t)
```

Fortran

```
MPI_TYPE_VECTOR(count, blocklen, stride, element_t,
                new_t, ierror)
integer count, blk_length, stride, element_t, new_t, ierror
```

- `count` - number of blocks
- `blocklen` - number of elements in each block
- `stride` - number of elements between starts of each block
- `element_t` - old datatype
- `new_t` - handle for new datatype

Application in Example

```
MPI_Datatype column_t;
/* define data type */
MPI_Type_vector(M, 1, N, MPI_DOUBLE,
                &column_t);
/* Make it accessible to communication */
MPI_Type_commit(&column_t);
/* Use it */
if (my_rank == 0)
  MPI_Send(&(A[0][j]), 1, column_t, ...);
else
  MPI_Recv(&(A[0][j]), 1, column_t, ...);
```

Other Constructors For Derived Datatypes

MPI_Type_contiguous An array of contibuous elements of an MPI datatype

MPI_Type_indexed Irregularly distributed elements of one MPI datatype

MPI_Type_struct Collection of elements of mixed MPI datatypes.

MPI_Pack/MPI_Unpack Not defining derived datatypes. Puts the burden of collecting data to the user (copy to buffer)

Example

Two floats and an integer shall be broadcast:

```
double a, b;
int n;
```

Problem

How can one send these values in *one* message?

Example (cont)

Observations:

- There are three elements to be transmitted.
-
  - The first element is a double.
  - The secont element is a double.
  - The third element is an int.
-
  - The first element has address &a.
  - The second element has address &b.
  - The third element has address &n.

This information describes the data completely.

Example (cont)

- The description is too specific such that it is impossible to reuse.
- Instead of (absolute) addresses, use the offsets with respect to the first element &a.
- Knowledge of &a is only necessary when the data is really sent.
- An MPI datatype consists of
  - an integer indicating the number of elements;
  - an ordered sequence of pairs (MPI_Datatype, MPI_Aint)

Example (cont)

```
#define ITEMS 3

int blocklen[ITEMS];
MPI_Aint displacements[ITEMS];
MPI_Datatype typelist[ITEMS];
MPI_Datatype mpi_new_t;

MPI_Aint start_add, address;

blocklen[0] = blocklen[1] = blocklen[2] = 1;
typelist[0] = MPI_DOUBLE;
typelist[1] = MPI_DOUBLE;
typelist[2] = MPI_INT;
```

Example (cont)

```
displacement[0] = 0;

MPI_Address(&a, &start_add);
MPI_Address(&b, &address);
displacement[1] = address-start_add;

MPI_Address(&n, &address);
displacement[2] = address-start_add;

MPI_Type_struct(ITEMS, blocklen, displacements,
                typelist, &mpi_new_t);
MPI_Type_commit(&mpi_new_t);
```

Summary

- Virtual topologies are a convenient way for handling standard communication patterns
- Communicators lie at the heart of virtual topology handling
- User defined datatypes are a convenient (but sometimes expensive) tool for handling non-contiguous data

Good Luck!!

Thank you for your patience!

The Mathematical Problem

- Ubiquitous equation
  - Fluid flow, electromagnetics, gravitational interaction, ...
- In two dimensions, Poisson's equation reads:
  - Solve $\Delta u = f(x, y)$ for $(x, y) \in \Omega$,
  - subject to the boundary condition $u(x, y) = g(x, y)$ for $(x, y) \in \partial\Omega$
- For simplicity, consider only $\Omega = (0, 1) \times (0, 1)$.
- Generalizations to other dimensions are obvious.

Discrete Approximation

- Define a *mesh* (or *grid*): For a given N, let

$$h = 1/(N-1), \quad x_m = mh, \quad y_n = nh$$

- Let $u_{mn} \approx u(x_m, y_n)$, $f_{mn} = f(x_m, y_n)$.
- Using the Laplace approximation from above, we obtain a system of equations

$$\frac{1}{h^2}(u_{m-1,n} + u_{m+1,n} + u_{m,n-1} + u_{m,n+1} - 4u_{mn}) = f_{mn},$$
$$0 < m, n < N-1$$

- In the context of pde's, the matrix W is usually called a *stencil*.

Jacobi Iteration

- *Basic idea*: Rewrite the equations as

$$u_{mn} = \frac{1}{4}(u_{m-1,n} + u_{m+1,n} + u_{m,n-1} + u_{m,n+1} - h^2 f_{mn})$$

- For some starting guess (e.g., $u_{mn} = 0$), iterate this equation,

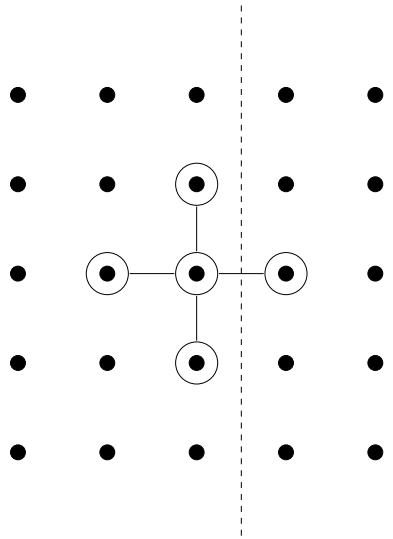
```
while (not_done)
  for (m,n) in 1:N-2 x 1:N-2
    ut(m,n)=(u(m-1,n)+u(m+1,n)+u(m,n-1)
             +u(m,n+1)-h^2*f(m,n))*0.25;
  end
  u = ut;
end
```

Accuracy

- How do we know that the answer is "good enough"?
  - When the computed solution has reached a reasonable approximation to the exact solution
  - When we can validate the computed solution in the field
- But often we do not know the exact solution, and must estimate the error, e.g.,
  - Stop when the residual is small enough, $r = Au - f$
  - Stop when the change $u - u'$ in u is small.

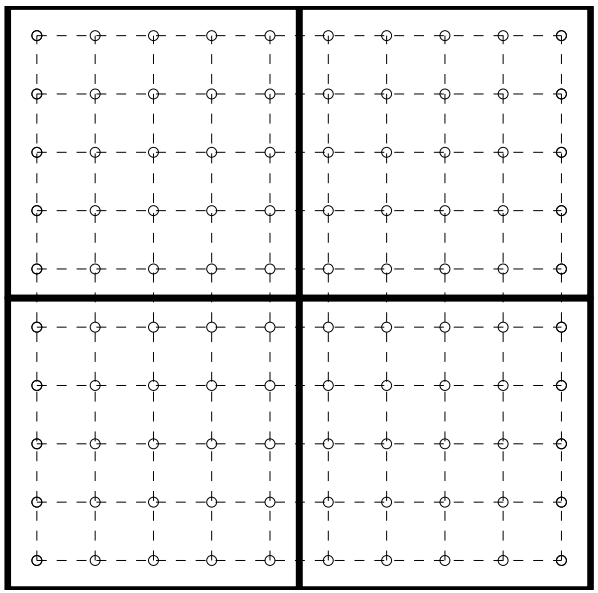
Generalizations To Two Dimensions

- Sample stencil (Poisson):

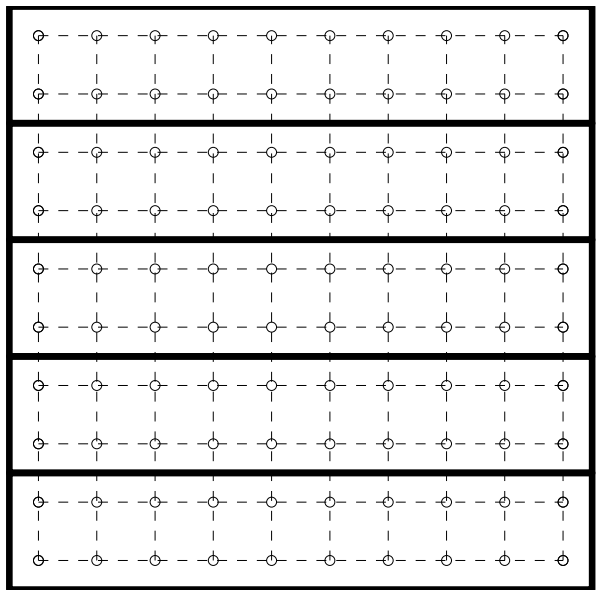


- Use an array of $R = P \times Q$ processors
- Distribute equal chunks of the pixmap/solution onto these processors
- Different partitions are called *processor geometry* or *processor topology*

Processor topology



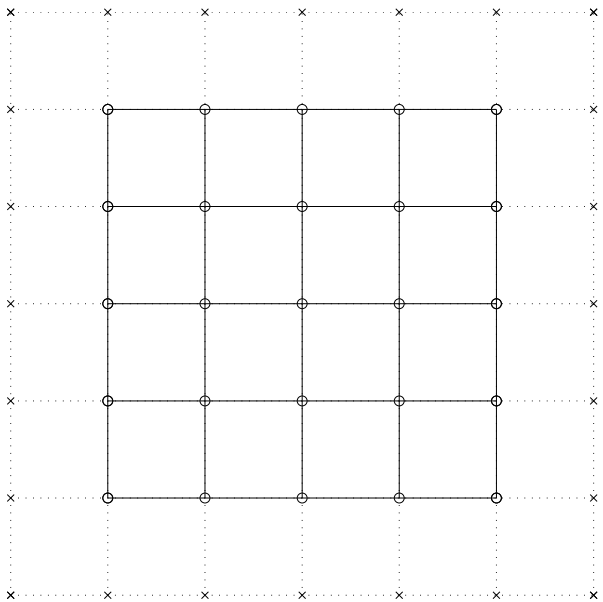
$P = Q = 2$ (to be written)



$P = 1$, $Q = 5$ (`oned.{fc}`, given)

Ghost Cells

- Each processor needs values found on neighboring processors
- Use *ghost cells*,



  - Circles: local grid points
  - Crosses: ghost points