MPI

All communication occurs within a *communicator* (`MPI_Comm`), or group of processes, where each process has a unique identifier.

The predefined communicator `MPI_COMM_WORLD` contains all processes.

Each process can determine its *rank*, and the *size* of a group it belongs to:

```
MPI_Comm_rank(MPI_Comm comm, int *rank);
```

```
MPI_Comm_size(MPI_Comm comm, int *size);
```

Initialization is required before MPI can be used:

```
MPI_Init(int *argc, char ***argv);
```

where `argc` and `argv` are the arguments to the main program ("command line parameters").

Before termination a program should clean up MPI data structures:

```
MPI_Finalize();
```


MPI – a first example

```
#include <mpi.h>
#include <iostream.h>
int main(int argc, char **argv) {
  int rank;
  int size;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  cout << "Hello, I'm process " << rank
       << " of " << size << endl;
  MPI_Finalize();
  return 0;
}
```

Output (on 3 computers):

```
Hello, I'm process 0 of 3
Hello, I'm process 2 of 3
Hello, I'm process 1 of 3
```

Not necessarily sorted!

Point-to-point communication

The simplest communication involves only two processes.

Several different routines, e.g. `MPI_Send` and `MPI_Recv`.

Source process:

```
MPI_Send(buf, count, datatype, dest,
tag, comm);
```

`buf`	initial address of send buffer
`count`	number of elements to send
`datatype`	data type to send
`dest`	rank of destination process
`tag`	integer identifying message
`comm`	MPI communicator

The MPI data types correspond to basic C types, e.g. `MPI_INT` and `MPI_DOUBLE`.

It is possible to create derived data types (not to be confused with derived classes).

Point-to-point communication

Destination process:

```
MPI_Recv(buf, count, datatype, source,
tag, comm, status);
```

`buf`	initial address of receive buffer
`count`	number of elements to receive
`datatype`	data type to receive
`dest`	rank of source process
`tag`	integer identifying message
`comm`	MPI communicator
`status`	Pointer to `MPI_Status`.

The structure `MPI_Status` contains information about tag and sending process.

The tag must match the sending process.

"Wildcards": `MPI_ANY_TAG` and `MPI_ANY_SOURCE` accept all tags/sources.

A matched send/receive call will copy data from the send buffer to the receive buffer (between different processes).

Point-to-point communication

Send an array from process 0 to 1.

```
int main(int argc, char **argv) {
  int rank, size;
  double x[10];
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (rank == 0) {
    for (int i=0; i<10 ; i++) x[i] = 0.1*i;
    MPI_Send(x, 10, MPI_DOUBLE, 1, 666,
             MPI_COMM_WORLD);
  }
  else if (rank == 1) {
    MPI_Recv(x, 10, MPI_DOUBLE, 0, 666,
             MPI_COMM_WORLD, &status);
  }
  MPI_Finalize();
}
```


Point-to-point communication

The time needed to transfer n bytes is often modeled by

$$t = t_{startup} + \beta \cdot n,$$

where

$t_{startup}$	time to initiate communication
β	bandwidth, time to transfer one byte

$t_{startup} \gg \beta$. Send large blocks at a time!

```
MPI_Send(x, 10, MPI_DOUBLE, 1, 666,
         MPI_COMM_WORLD);
```

better than

```
for (i=0; i<10 ; i++)
  MPI_Send(&(x[i]), 1, MPI_DOUBLE, 1, 666,
           MPI_COMM_WORLD);
```


A distributed vector

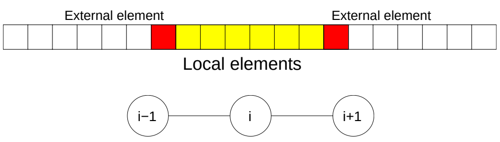
Assume that we have a one-dimensional grid distributed over n processes.

Let y be defined on the grid, and compute D_+D_-y with periodic boundary conditions,

$$D_+D_-y_i = \begin{cases} (y_m - 2y_0 + y_1)/\Delta x^2 & i = 0 \\ (y_{i-1} - 2y_i + y_{i+1})/\Delta x^2 & 0 < i < m \\ (y_{m-1} - 2y_m + y_0)/\Delta x^2 & i = m \end{cases} .$$

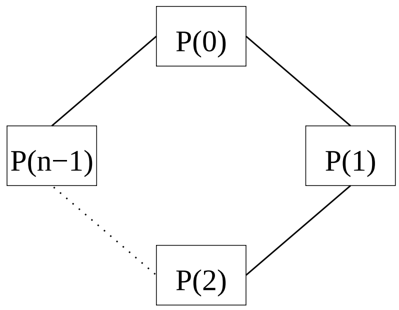
($0 \le i \le m$ "global" numbering.)

Note: Each process needs one element stored on the processes to the "left" and "right".



Stencil : D_+D_-

Difference operator



"Logical topology"

Can think of processors as connected in a circle.

Attempted (erroneous) solution
On process p:

1. Receive from "left" $((p + n - 1) \mod n)$
2. Send to "left" $((p + n - 1) \mod n)$
3. Receive from "right" $((p + n + 1) \mod n)$
4. Send to "right" $((p + n + 1) \mod n)$
5. Compute local part of D_+D_-y

Difference operator

Pseudo-code:

```
// Receive from (p+n-1) % n
// Send to (p+n-1) % n
// Receive from (p+n+1) % n
// Send to (p+n+1) % n
// Compute action of diff. op.
```

Deadlock!
Mismatch in communication. All processes waiting to receive.

No process will reach computation.

Possible solutions:

- Rewrite program so calls to `MPI_Send` and `MPI_Recv` are matched.
- Non-blocking communication. Perform communication in background while doing computation (*next week*).

Marco Kupiainen
marcok@nada.kth.se

Difference operator

Algorithm:

1. Send data clockwise starting from process 0
2. Send data counterclockwise starting from process 0

Pseudo-code:

```
if (rank != 0) {
  // Receive from (p+n-1) % n
  // Send to (p+n+1) % n
  // Receive from (p+n+1) % n
  // Send to (p+n-1) % n
}
else {
  // Send to (p+n+1) % n
  // Receive from (p+n-1) % n
  // Send to (p+n-1) % n
  // Receive from (p+n+1) % n
}
// Compute action of diff. op.
```

Marco Kupiainen
marcok@nada.kth.se

Difference operator

Algorithm works. All calls to `MPI_Send` are matched by calls to `MPI_Recv`.

Inefficient!

- Most processors are idle during communication.
- A parallel computer can handle many simultaneous local communications.

Possible solutions:

- Non-blocking communication (again)
- Red-black ordering allows processors to communicate simultaneously. See supplementary lecture notes.
- Rearrange communication & computation so that parts of D_+D_-y are evaluated while waiting (technical).

Marco Kupiainen
marcok@nada.kth.se