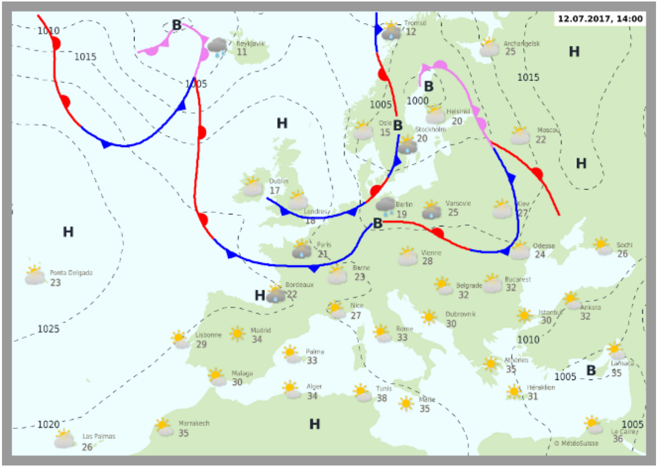
A little introduction to MPI

Jean-Luc Falcone

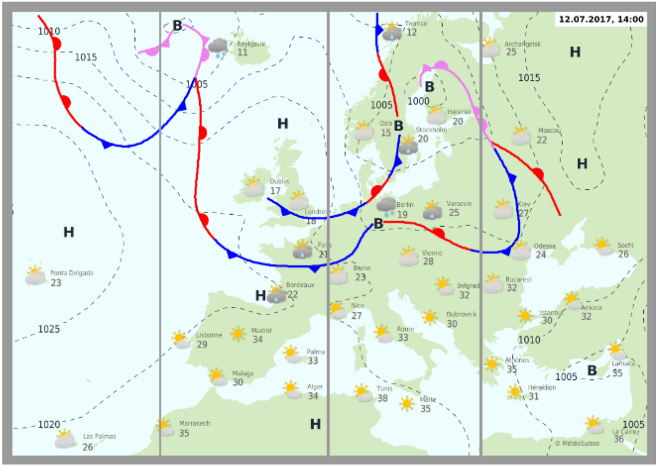
July 2017

Outline

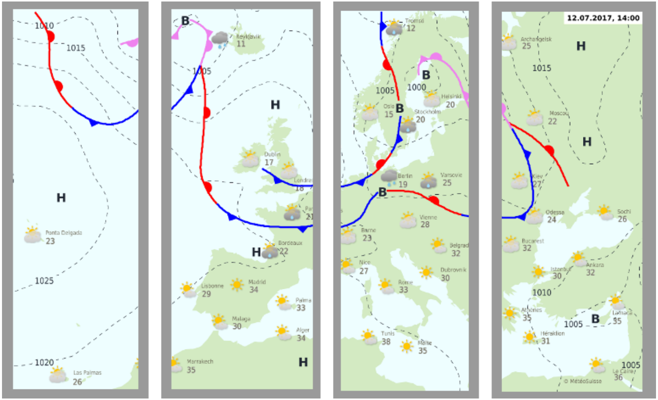
Sequential


12.07.2017, 14:00
Reykjavik 11
Tromsø 12
Archangelsk 25
Helsinki 20
Oslo 15
Stockholm 20
Moscou 22
Dublin 17
Londres 18
Berlin 19
Varsovie 25
Kiev 27
Paris 21
Vienne 28
Odessa 24
Sochi 26
Ponta Delgada 23
Berne 23
Bordeaux 22
Belgrade 32
Bucarest 32
Nice 27
Dubrovnik 30
Istanbul 30
Ankara 32
Lisbonne 29
Madrid 34
Rome 33
Palma 33
Athènes 35
Larnaca 35
Malaga 30
Alger 34
Tunis 38
Malte 35
Héraklion 31
Marrakech 35
Las Palmas 26
Le Caire 36
© MétéoSuisse
1010
1015
1005
1000
1025
1020
H
B

Parallel: Shared Memory



Parallel: Distributed Memory



Main idea

- Independent processes with data isolation
- Pass messages to communicate and synchronize
- May run on the same machine or on a network of machines.

Advantages

- Not limited to a single machine
- No shared stated: less bugs

Disadvantages

- Data must be divided explicitly
- Sending a message is slower than memory access

MPI (Message Passing Interface)

- Proposed in the early 90s
- Maintained and extended since (MPI-3.1 in 2015)
- Widely used for HPC
- Deployed in all current supercomputers
- Main programming languages: FORTRAN, C & C++
- But bindings for others: R, Python, Java, Rust, etc.

SPMD

- Single Program, Multiple Data
- ie. Several instance of the same program will be executed in parallel.

Let's try

Exercice 0.0

- Dowload `helloworld.c`
- Compile it: `mpicc -o helloworld helloworld.c`
- Run it:
  - `./helloworld`
  - `mpirun -np 2 helloworld`
  - `mpirun -np 8 helloworld`
  - What does `np` mean ?
- Run it again:
  - `mpirun -np 8`
  - `mpirun -np 8`
  - What can we observe ?

Skeleton

Most MPI programs have the following structure:

```
MPI_Init(NULL,NULL);

/* Perform actual stuff */

MPI_Finalize();
```

World communicator and ranks

- MPI process use communicators to communicate
- By default, they are all in the `MPI_COMM_WORLD`
- The size of a communicator can be retrieved with `MPI_Comm_size`
- Instead of an adress, each MPI process of a single execution has a `Rank`.
- The rank of a single process can be retrievd with `MPI_Comm_rank`

Who's the Boss ?

Exercice 0.1

- Copy the preceding example into `boss.c`
- Modify the program such as to:
  - only the process with the highest rank greets the world.
  - all other process must stay calm and silent

Point to point communications

- The simplest way of passing a message is using point to point communications
- Paired process can send/receive data

Send

To send data use the `MPI_Send` function:

```
MPI_Send(
    void* data,
    int count,
    MPI_Datatype datatype,
    int destination,
    int tag,
    MPI_Comm communicator)
```

Send: What ?

Data to send are described by three arguments:

- `void* data`: The address of the beginning of the data
- `int count`: How many data
- `MPI_Datatype datatype`: The type of data

Warning

If you pass arguments with incorrect value, everything will still compile fine. If you are lucky it will crash at runtime. It may also fail silently. . .

Send: Where and how ?

The last three arguments of send are:

- `int destination`: the rank of the destination process in the communicator
- `int tag`: the message tag (user defined)
- `MPI_Comm communicator`: the communicator to be used

Send: examples

```
int x = 12;
MPI_Send( &x, 1, MPI_INT, 3, 2, MPI_COM_WORLD );

int[] y = {3,5,7,9};
MPI_Send( y, 4, MPI_INT, 0, 0, MPI_COM_WORLD );
MPI_Send( &y[1], 2, MPI_INT, 1, 0, MPI_COM_WORLD );
```

MPI Datatypes

MPI Datatype	C type
MPI_CHAR	char
MPI_SHORT	short int
MPI_LONG	long int
MPI_UNSIGNED	unsigned int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_BYTE	8 bits
...	...

- You could add your own types...

Tags

Tags are user defined. They may be useful for:

- Debugging your code
- Sending and receiving message out of order, etc.

Receive

To receive data use the MPI_Recv function:

```
MPI_Recv(
    void* data,
    int count,
    MPI_Datatype datatype,
    int source,
    int tag,
    MPI_Comm communicator,
    MPI_Status* status)
```

Receive: What ?

Data to send are described by three arguments:

- `void* data`: The address where the data will be received.
- `int count`: How many data
- `MPI_Datatype datatype`: The type of data

Warning

You must allocate the reception buffer (here `data`), before receiving data...

Receive: Where and how ?

The last three arguments of receive are:

- `int source`: the rank of the sender process in the communicator
- `int tag`: the expected message tag (user defined)
- `MPI_Comm communicator`: the communicator to be used
- `MPI_Status* status`: a pointer to a status struct (info about the reception)

Wildcards

- If you wish to receive data from any sender, you can use the constant `MPI_ANY_SOURCE` instead of source.
- If you don't care about the the tag: `MPI_ANY_TAG`
- If you don't need the status: `MPI_STATUS_IGNORE`

Receive: examples

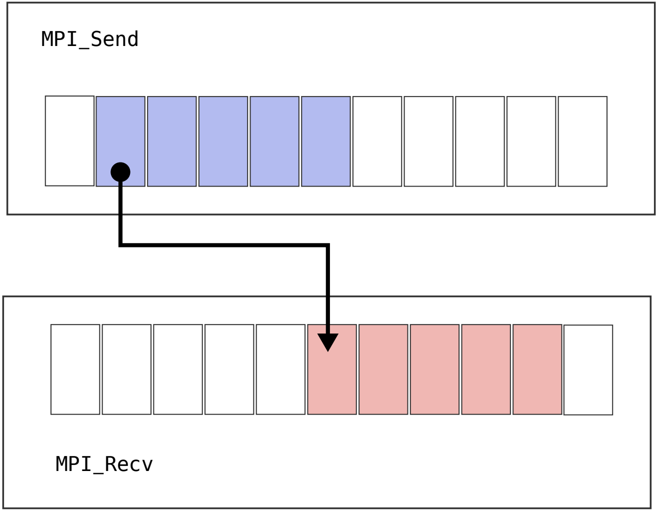
```
int x = -1;
MPI_Recv( &x, 1, MPI_INT, MPI_ANY_SOURCE,
  MPI_ANY_TAG, MPI_COM_WORLD, MPI_STATUS_IGNORE );

MPI_Status status;
int *machin = (int*) malloc( sizeof(int) * 4 );
MPI_Recv( machin, 4, MPI_INT, 1, 0, MPI_COM_WORLD, &status )
```

Circularize

Exercice 1.0

- Compile the file `secretChain.c` and run it with:
  - `mpirun -np 2 secretChain`
  - `mpirun -np 4 secretChain`
  - `mpirun -np 50 secretChain`
  - `mpirun -np 1 secretChain` (it should crash)
- Read the code source and try to make some sense of the output.
- Copy `secretChain.c` into `secretCircle.c`.
- Edit `secretCircle` to close the circle:
  - The process with rank 0 will receive from the last rank.
  - The process with the last rank will send to the rank 0.

Concurrency issues



Solution #1: Buffering

- `MPI_Send` may use a hidden and opaque buffer.
- If data to send fit inside this buffer, it is copied and send return fast.
- The size of this buffer depends on the implementation never rely on that.

Solution #2: Wait for reception (blocking)

- `MPI_Ssend` is similar to `MPI_Send`, but it is synchronized
- It will block until the destination process reaches the reception.
- When `MPI_Ssend` returns:
  - the data buffer can be reused.
  - the destination did receive the message.

- When data to be sent is *large*, `MPI_Send` behaves like `MPI_Ssend`.

Solution #3: Non-blocking transmission

- Calls to `MPI_Isend` returns almost immediately.
- Data will be sent in background (possibly in another thread).
- Your program may perform some work in the mean time.
- But the data buffer shall not be reused until everything is sent.

- `MPI_Isend` takes an additional parameter which allows to query or wait for transfer completion.

```
int MPI_Isend(const void *buf, int count,
   MPI_Datatype datatype, int dest, int tag,
   MPI_Comm comm, MPI_Request *request)
```

Non-blocking send: MPI_Request

- MPI_Request can be used through several functions:
  - MPI_Test allows to check if transfer is completed.
  - MPI_Wait allows to wait for transfer completion
  - MPI_Waitall allows to wait for several transfer completions.
  - MPI_Testany allows to check for the completed transfer among many.
  - ... and many other combinations

Non-blocking send: Example

```
MPI_Request req;

MPI_Isend( data, 40000, MPI_DOUBLE, 0, 0,
 MPI_COM_WORLD, &req );

//Here we compute some stuff
//(without touching 'data')

MPI_Wait( &req, MPI_STATUS_IGNORE );
```

Non-blocking receive

Similarly, there is a non-blocking receive:

```
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
     int source, int tag, MPI_Comm comm,
     MPI_Request *request)
```

Nicer circler

Exercice 1.1

- Copy `secretCircle.c` into `secretCircleNB.c`.
- Edit `secretCircleNB.c`:
  - Use `MPI_Isend` to send the messages
  - Try to simplify the code and remove all the `if` and `else`.

Comparison game

```
MPI_Send( x, 2, MPI_INT, 0, 0, MPI_COMM_WORLD );
```

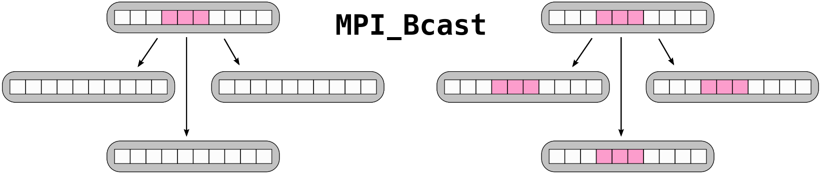
vs.

```
MPI_Send( &x[0], 1, MPI_INT, 0, 0, MPI_COMM_WORLD );
MPI_Send( &x[1], 1, MPI_INT, 0, 0, MPI_COMM_WORLD );
```

More boring code: what does it do ?

```
double x = 0.0;

if( myRank == 0 ) {
  x = getValue();
  for( int i=1; i<comSize; i++ )
    MPI_Send( &x, 1, MPI_DOUBLE, i, 1, MPI_COMM_WORLD );
} else {
   MPI_Recv( &x, 1, MPI_DOUBLE, 0, 1,  MPI_COMM_WORLD,
     MPI_STATUS_IGNORE );
}
```

Broadcast: the big picture



Broadcast

To broadcast values to all process in a communicator:

```
int MPI_Bcast( void *buffer, int count,
   MPI_Datatype datatype,
   int root, MPI_Comm comm
)
```

- `int root` is the rank of the process sending the value.

Warning

All process must call `MPI_Bcast` (instead of send/recv pairs).

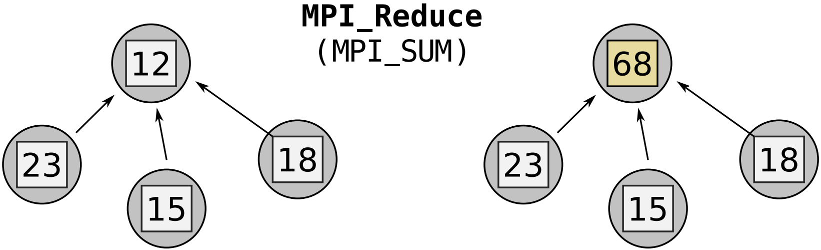
Broadcast: Example

```
double x = 0.0;

if( myRank == 0 ) {
  x = getValue();
}

MPI_Bcast( &x, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD );
```

Do you spot the problem ?

```
int x = computeLocalSum(...);
int sum = 0;

if( myRank != 2 ) {
  MPI_Send( &x, 1, MPI_INT, 2, 0, MPI_COMM_WORLD );
} else {
  sum += x;
  for( int i = 1; i<comSize; i++ ) {
    MPI_Recv( &x, 1, MPI_INT, i, 0, MPI_COMM_WORLD, &stat );
    sum += x
  }
}

if( myRank == 0 ) {
  printf( "The global sum is: %d.\n", y );
}
```

MPI Reduce



MPI Reduce: signature

```
MPI_Reduce(
    void* send_data, //Data to reduce
    void* recv_data, //Result of reduction
    int count,
    MPI_Datatype datatype,
    MPI_Op op,       //Reduction operation
    int root,        //Process getting the result
    MPI_Comm communicator)
```

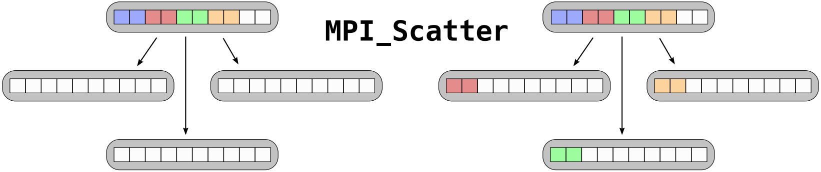
MPI Operations

- The following operations are predefined (`MPI_Op`):
  - `MPI_MAX`: largest element
  - `MPI_MIN`: smallest element
  - `MPI_SUM`: sum of elements
  - `MPI_PROD`: product of elements
  - `MPI_LAND`: logical *and*
  - `MPI_LOR`: logical *or*
  - ...
- If arrays of man elements are reduced, operations are element-wise.
- You could define your own reduction operation.

MPI Reduce: Example

```
int x, y = 0;

x = computeLocalSum(...);

MPI_Reduce( &x, &y, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD )

if( myRank == 0 ) {
  printf( "The global sum is: %d.\n", y );
}
```

MPI Scatter



MPI Scatter

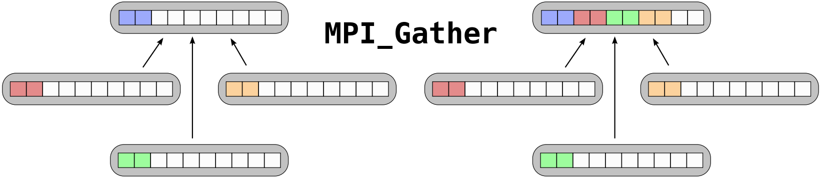
```
MPI_Scatter(
    void* send_data,   //Data source
    int send_count,    //Total
    MPI_Datatype send_datatype,
    void* recv_data,   //Data destination
    int recv_count,    //Per process
    MPI_Datatype recv_datatype,
    int root,          //Scattering process
    MPI_Comm communicator)
```

Don't forget: `recv_count` must be `send_count / comSize`.

MPI Scatter: Example

```
double *x, *y = NULL;

if( myRank == 0 ) {
  x = malloc( sizeof(double) * 64 );
  //Fill x with relevant data
}
y = malloc( sizeof(double) * 4 );

MPI_Scatter( x, 64, MPI_DOUBLE, y, 4, MPI_DOUBLE,
0, MPI_COMM_WORLD );
```

MPI Gather



MPI Gather

```
MPI_Gather(
    void* send_data,
    int send_count,
    MPI_Datatype send_datatype,
    void* recv_data,
    int recv_count,
    MPI_Datatype recv_datatype,
    int root,
    MPI_Comm communicator)
```

Same as `MPI_Scatter` but in that case `send_count * comSize` must be `recv_count`.

MPI Gather: Example

```
double *x, *y = NULL;

if( myRank == 0 ) {
  x = malloc( sizeof(double) * 64 );
}
y = malloc( sizeof(double) * 4 );
//Compute y values

MPI_Scatter( y, 64, MPI_DOUBLE, x, 4, MPI_DOUBLE,
0, MPI_COMM_WORLD );
```