These slides were originaly created by Patrick Carribault from CEA as part of INF560 (Algorithmique Parallèle et Distribuée) at Ecole Polytechnique. They were (slightly) adapted to fit this class format.
Lecture Outline

- Parallel programming
- Introduction to MPI
  - Compilation & execution
  - Main organization
- Point-to-point communications
Two main paradigms
- Distributed-memory programming model
- Shared-memory programming model

Inspired from system organization

But: independent from system & hardware
- In theory, every model can be implemented on any system architecture
- In practice, mapping of some combinations can be difficult!
Shared-Memory Model

- **Requirement**
  - Parallel tasks should have the same view of memory
- **Consequence**
  - Concurrent accesses to memory should be handled

- **On distributed-memory system**
  - Difficult – How to share the memory view?
    - DSM (Distributed Shared Memory)
      - May generate a large overhead
      - Depend on the number of remote accesses

- **On shared-memory system**
  - Easy because of shared memory – Inside multithreaded process
    - Every thread have access to the same memory zone
    - Usually, whole node memory

- OpenMP, pthread, TBB, Cilk, ...
Distributed-Memory Model

- **Requirements**
  - Parallel tasks work on their own memory space
  - Data are split among parallel tasks to enable parallel execution

- **Consequence**
  - Need communications between tasks
  - Message-passing programming

- **On distributed-memory system**
  - Easy
  - Processes on such systems have to exchange messages which is included in distributed-memory model

- **On shared-memory system**
  - Easy as well! – Even if tasks may shared memory, the model can hide this feature
  - Implemented with processes, it is possible to use shared-memory segment to improve communication performance

- PVM, MPI
Parallel Programming

Message Exchange
Message Exchange

- Message characteristics
  - Sender
  - Destination task
  - Data to exchange

- High-level protocol
  - Pair of actions will resolve message exchange
  - Sender must send the message
    - Let’s consider a function called \textit{send}
  - Recipient must receive the message
    - Let’s consider a function called \textit{recv}
Main Principle

- Two parallel tasks $T_0$ et $T_1$
  - Distinct memory space
  - Each task has its own instructions to execute

**T0 Task**

```
instruction1;
instruction2;
```

**T1 Task**

```
instruction1;
instruction2;
```
**Main Principle**

- **T1** depends on **T0**
  - T0 must send data to **T1**
  - Data are located in `adr_send` with `nb_elt` elements

**T0 Task**

```
instruction1;
instruction2;
send(adr_send, nb_elt, T1);
```

**T1 Task**

```
instruction1;
instruction2;
```
Main Principle

- **T1** must receive data from **T0** (*recv*)
  - Size of message `nb_elt` should be known by recipient
  - Recipient may have to allocate a memory zone to get the received data (zone pointed by `adr_recv`)

**T0 Task**

```
instruction1;
instruction2;
send(adr_send, nb_elt, T1);
```

**T1 Task**

```
instruction1;
instruction2;
recv(adr_recv, nb_elt, T0);
```
Main Principle

- Communication
  - send blocks $T_0$ until data are sent
  - recv blocks $T_1$ until data are received

```
T0 Task
instruction1;
instruction2;
send(adr_send, nb_elt, T1);

T1 Task
instruction1;
instruction2;
recv(adr_recv, nb_elt, T0);
```

Data Transfer
Main Principle

- Communication
  - *send* blocks $T_0$ until data are sent
  - *recv* blocks $T_1$ until data are received

```
T0 Task

instruction1;
instruction2;
send(adr_send, nb_elt, T1);

T1 Task

instruction1;
instruction2;
recv(adr_recv, nb_elt, T0);
```

Data Transfer
Main Principle

- T1 owns a complete copy of data sent by T0

T0 Task

\[\text{instruction1; instruction2; send(adr\_send, nb\_elt, T1);}\]

T1 Task

\[\text{instruction1; instruction2; recv(adr\_recv, nb\_elt, T0);}\]
Main Principle

- Tasks $T_0$ and $T_1$ may continue their execution.
- Following instructions of $T_1$ may access to data stored at address $adr_{recv}$.
Example

- Parallel sum on each element of an array

- Hypothesis
  - Array \( t \) with \( N \) floats (\( N \) is even)
  - Array \( t \) is distributed across 2 tasks \( T_0 \) and \( T_1 \)
    - Parallelism type: data

- Goal
  - \( T_1 \) must print the sum of each element of \( t \)

- Code?
Example

T0 sends its partial sum to T1

double p = 0.0;
int i;

for ( i=0 ; i<N/2 ; i++ )
    p += tab[i];

send(&p, 1, T1);

T1 needs partial sum from T0

double p = 0.0;
double s;
int i;

for ( i=0 ; i<N/2 ; i++ )
    p += tab[i];

recv(&s, 1, T0);

printf("%g",s+p);
Send/Recv Matching

- Every *send* corresponds to one *recv* (and vice-versa)

- Model with an oriented graph
  - Vertices are tasks
  - Edges are communications

- A missing send or receive action lead to a deadlock situation
Introduction to MPI
Introduction

- MPI: Message-Passing Interface

- High-level API (Application Programming Interface)
  - Parallel programming
  - Distributed-memory paradigm

- Implementation as a library
  - Interface through functions

- Language compatibility
  - C
  - C++
  - FORTRAN
MPI Overview

- MPI includes (mainly MPI 1)
  - Execution environment
  - Point-to-point communication
  - Collective communications
  - Groups and topologies of tasks

- MPI 2.0 adds
  - One-sided communications
  - Multithreading
  - Parallel I/O

- MPI 3.0 adds
  - Non-blocking collective communications

- Lots of features!
  - 120 functions in MPI 1
  - More than 200 for MPI 2
#include <stdio.h>
/* MPI function signatures */
#include <mpi.h>

int main(int argc, char **argv){

    /* Initialization of MPI */
    MPI_Init(&argc, &argv);

    printf("Hello World!\n");

    /* Finalization of MPI */
    MPI_Finalize();

    return 0;
}
Compilation

» Basically
  ◦ *Compilation process like any other library*

» **But multiple ways to compiler an MPI program**
  ◦ Simple way: rely on `mpicc` script
  ◦ Complex way: launch regular compiler with options to specify paths to the library

» **Simple way**
  ◦ Script/program that hide the library configuration details
    `mpicc -o hello hello.c`
  ◦ Call the default underlying compiler
    ▶ Possible to change the compiler that will be invoked
  ◦ *This way for the Labs!*

» **Complex way**
  ◦ Use a standard compiler, and pass lots of options
    `gcc -I/dir/mpi/include -o hello hello.c -L/dir/mpi/lib -lmpi`
Execution

- mpirun can spawn MPI processes
  - Connect on machines
  - Create network connections

$ mpirun -n 4 -f machines ./hello
Hello World!
Hello World!
Hello World!
Hello World!

- Remarks
  - Creation of 4 processes
  - Every process has the same instructions
  - Processes are independent for execution
  - « machines » contains a list of machines
Communicator

mpirun -n 4 ./hello

- Group of processes form a communicator
  - Predefined: MPI_COMM_WORLD w/ all processes

- Communicator = set of processes + communication context
  - Type: MPI_Comm
Total Number of Processes

int MPI_Comm_size( MPI_Comm comm, int *size);

- Return size of communicator comm in *size
- If comm == MPI_COMM_WORLD, MPI_Comm_size returns the total number of MPI processes in the application

```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char **argv) {
  int N;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &N);
  printf("Number of processes = %d\n", N);
  MPI_Finalize();
  return 0;
}
```

$ mpirun -n 4 ./a.out
Number of processes = 4
Number of processes = 4
Number of processes = 4
Number of processes = 4
Process Rank

- Inside a communicator, MPI assigns rank from 0 to size-1
  - This is the rank of a process

- Function `MPI_Comm_rank` returns the rank in the communicator `comm` inside the address `*rank`:

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

```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char **argv) {
  int N, me;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &N);
  MPI_Comm_rank(MPI_COMM_WORLD, &me);
  printf("My rank is %d out of %d\n", me, N);
  MPI_Finalize();
  return 0;
}
```

```bash
$ mpirun -n 4 ./a.out
My rank is 1 out of 4
My rank is 0 out of 4
My rank is 3 out of 4
My rank is 2 out of 4
```
Process Rank

- Number of processes may different from number of available cores/processors!

- Execution of processes is not related to their rank
  - Parallel execution
  - At the beginning, no ordering between processes
  - Only communications can imply some partial ordering

- Rank is usually used to determine
  - Which part of data should I work on?
  - What is my role (master/slave)?
MPI Point-to-Point Communications

Send/Recv
MPI Communication

- MPI is a parallel distributed-memory model
  - Each process accesses its own memory space
  - Based on message passing

- What is the main interface for data exchange w/ MPI?

- To send a message
  - MPI_Send function
Sending Messages

- Function to send a message

```c
int MPI_Send (  
    void *buf\(^{(in)}\),  
    int count\(^{(in)}\),  
    MPI_Datatype datatype\(^{(in)}\),  
    int dest\(^{(in)}\),  
    int tag\(^{(in)}\),  
    MPI_Comm comm\(^{(in)}\)
);
```

Main characteristics of message to send
Function to send a message

```c
int MPI_Send(
    void *buf,  // Data address
    int count,  // Data to send inside an array pointed by buf whose elements are of type datatype.
    MPI_Datatype datatype,  // MPI predefined scalar types corresponding to existing C types.
    int dest,  //  
    int tag,  //  
    MPI_Comm comm);  //  
```
## Sending Messages

<table>
<thead>
<tr>
<th>MPI_Datatype</th>
<th>C Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>signed char</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>signed short int</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>signed int</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>signed long int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR</td>
<td>unsigned char</td>
</tr>
<tr>
<td>MPI_UNSIGNED_SHORT</td>
<td>unsigned short int</td>
</tr>
<tr>
<td>MPI_UNSIGNED</td>
<td>unsigned int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>long double</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td>One byte</td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td>Pack of non-contiguous data</td>
</tr>
</tbody>
</table>
int MPI_Send ( 
    void *buf\textsuperscript{(in)}, 
    int count\textsuperscript{(in)}, 
    MPI_Datatype datatype\textsuperscript{(in)}, 
    int dest\textsuperscript{(in)}, 
    int tag\textsuperscript{(in)}, 
    MPI_Comm comm\textsuperscript{(in)}
);
int MPI_Send ( 
  void *buf\text{(in)},
  int count\text{(in)},
  MPI_Datatype datatype\text{(in)},
  int dest\text{(in)},
  int tag\text{(in)},
  MPI_Comm comm\text{(in)}) ;

Communicator for message.

Communicator = (sub-)set of processes + communication context

MPI_COMM_WORLD contains all processes created during application launch
int MPI_Send (  
    void *buf,  
    int count,  
    MPI_Datatype datatype,  
    int dest,  
    int tag,  
    MPI_Comm comm);  

Recipient rank.  
This rank is valid inside communicator comm.  
For MPI_COMM_WORLD, dest should be between 0 and number of tasks (excl.).
Sending Messages

int MPI_Send (    
    void *buf(\textit{in}),    
    int count(\textit{in}),    
    MPI_Datatype datatype(\textit{in}),    
    int dest(\textit{in}),    
    int tag(\textit{in}),    
    MPI_Comm comm(\textit{in})    
);
Remarks on Sending Messages

- **MPI_Send** is a blocking function
  - Returning from **MPI_Send**, process can manipulate (e.g., write) the data buffer containing the message
  - It doesn’t mean that
    - Message has been sent
    - Message has been received

- How to determine the message tag
  - Can use any way you want
  - Not necessary for different send/recipient pair
  - Example:
    \[
    \text{tag} = \text{src} \times N + \text{dest}
    \]
    - \(N\) total number of MPI processes,
    - \(\text{src}\) sender rank,
    - \(\text{dest}\) recipient rank;

- Be careful: the number of tags is limited!
MPI Communication

- What is the main interface for data exchange w/ MPI?

- Message reception
  - MPI_Recv function
Receiving Messages

int MPI_Recv (
    void *buf\textit{(out)},
    int count\textit{(in)},
    MPI\_Datatype datatype\textit{(in)},
    int source\textit{(in)},
    int tag\textit{(in)},
    MPI\_Comm comm\textit{(in)},
    MPI\_Status *\textit{status(out)}
 );

\begin{quote}
Main characteristics of message to receive
\end{quote}
Receiving Messages

```c
int MPI_Recv (  
    void *buf(in),
    int count(in),
    MPI_Datatype datatype(in),
    int source(in),
    int tag(in),
    MPI_Comm comm(in),
    MPI_Status *status(out)
);  
```

- **Rank of sender.**
  - Rank should be valid in `communicator`.
  - Can specify the predefined value `MPI_ANY_SOURCE`.
  - May match a message from any sender in the target communicator.

- **Message tag.**
  - Should be the same of the one put in corresponding `MPI_Send` function call.

- **Information about received message.**
Information and Status

- MPI_Status is a C structure

```c
struct MPI_Status {
    int MPI_SOURCE; /* message sender (useful w/ MPI_ANY_SOURCE argument) */
    int MPI_TAG; /* message tag (useful w/ MPI_ANY_TAG argument) */
    int MPI_ERROR; /* error code */
};
```

- If message size is unknown to the recipient, it is possible to extract the actual size with MPI_Get_count

```c
int MPI_Get_count(  
    MPI_Status *status(in), /* status returned by MPI_Recv */
    MPI_Datatype datatype(in), /* Type of elements in the message */
    int *count(out) /* Size of the message (in number of elements of type datatype) */
);  
```
int main(int argc, char **argv) {  
  double p = 0., s0;  
  int i, r;  
  MPI_Status status;

  MPI_Init(&argc, &argv); /* Initialization of MPI library */  
  MPI_Comm_rank(MPI_COMM_WORLD, &r); /* Get the rank of current task */

  for( i = 0 ; i < N/2 ; i++ )  
    p += tab[i];

  tag = 1000; /* Message tag */
  if (r == 0) {  
    /* Rank 0 */
    MPI_Send(&p, 1, MPI_DOUBLE, 1, tag, MPI_COMM_WORLD);
  } else {  
    /* Rank 1 */
    MPI_Recv(&s0, 1, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD, &status);
    printf( "Sum = %d\n", s0+p );
  }

  MPI_Finalize();
  return 0;
}
sum = 0.;                        /* Each process has N/P elements of distributed */
for( i = 0 ; i < N/P ; i++ )      /* array and perform a partial sum */
    sum += tab[i];

  if (r == 0) {
    /* Process 0 receives P-1 messages in any order */
    for( t = 1 ; t < P ; t++ ) {

        MPI_Recv(&s, 1, MPI_DOUBLE,
                MPI_ANY_SOURCE, MPI_ANY_TAG, /* wildcards */
                MPI_COMM_WORLD, &sta);

        printf("Message from rank %d\n", sta.MPI_SOURCE);

        sum += s; /* Contribution of process sta.MPI_SOURCE to the global sum */
    }
  } else {

    /* Other processes send their partial sum to rank 0 */
    MPI_Send(&som, 1, MPI_DOUBLE, 0, rang, MPI_COMM_WORLD);
  }
Blocking Communications

- MPI_Send and MPI_Recv are blocking
  - MPI_Send returns when data buffer can be manipulate again by sender
  - MPI_Recv returns when the message arrived and has been processed

- Issue?
  - Be careful to deadlock situations!
Definition

- A send is **blocking** if after performing send it is possible to manipulate (read/write) the input data buffer without corrupting the communication.

Meaning

- A blocking send will not return while the communication library is not able to handle the message.
Blocking Communications

- After `send`, \( T_0 \) may modify the value of \( a \)
- \( T_1 \) will receive 100 (value of \( a \) as input of `send` by \( T_0 \))

```c
T0
a = 100;
send(&a, 1, T1);
a = 0;

T1
recv(&a, 1, T0);
printf("%d\n", a);
```

- Note
  - Resolving a blocking send does not mean that the receiver has the message
Blocking Communications

- **Definition**
  - A `recv` is **blocking** if after performing `recv` the output buffer contains the received message

- **Meaning**
  - A blocking `recv` will return only when the message has been received and processed
## Blocking Communications

- **After send,**
  - T0 may manipulate a and its content

- **After recv,**
  - Content of output buffer (a in T1) can be manipulated (read, write, print...) without concurrency issue

```c
a = 100;
send(&a, 1, T1);
a = 0;
recv(&a, 1, T0);
printf("%d\n", a);
```
MPI Point-to-Point Communications
Non-Blocking Communications
Non-Blocking Communication

- **Definition**
  - A *non-blocking* communication has not guarantee when send function returns!

- **Meaning**
  - No safe access to input message when function send returns
  - To be sure that message buffer can be reused, an additional function should be called and returned
Non-Blocking Send

MPI_Isend

```c
int MPI_Isend (  
    void *buf,  // input
    int count,  // input
    MPI_Datatype datatype,  // input
    int dest,  // input
    int tag,  // input
    MPI_Comm comm,  // input
    MPI_Request *req)  // output
```

One additional argument

`MPI_Request *req`.

Request id is returned in

`*req` (MPI_Request = MPI opaque type).
int MPI_Wait ( 
    MPI_Request *req\textsuperscript{(inout)}, 
    MPI_Status *sta\textsuperscript{(out)} 
); 

MPI_Wait blocks until communication represented by *req is done.

Detailed information about finished communication are store into *sta.

When MPI_Wait returns
- *req is assigned to MPI_REQUEST_NULL (invalid request)
- Input message buffer can be safely manipulated by sender

Remark:

\[
\text{MPI\_Send} \leftrightarrow \text{MPI\_Isend} + \text{MPI\_Wait}
\]
Non-Blocking Example

Advantages
- Recover communications and computation

```c
MPI_Request req;
MPI_Status sta;

MPI_Isend(buf, N, MPI_BYTE,
           dest, tag1, comm,
           &req);

instruction1;
instruction2;
...
instructionN;
MPI_Wait(&req, &sta);
```

Instructions between `MPI_Isend` and `MPI_Wait` should not write into `buf`.

In the meantime, message progresses.
Non-Blocking Reception

```
int MPI_Irecv(
    void *buf\(^{\text{out}}\),
    int count\(^{\text{in}}\),
    MPI_Datatype datatype\(^{\text{in}}\),
    int source\(^{\text{in}}\),
    int tag\(^{\text{in}}\),
    MPI_Comm comm\(^{\text{in}}\),
    MPI_Request *req\(^{\text{out}}\)
);
```

On additional argument

```
MPI_Request *req.
```

To finish the communication

```
MPI_Wait should be called.
```
Matching combinations

- Non-blocking send / Blocking recv
  - Sender: MPI_Isend, MPI_Wait
  - Recipient: MPI_Recv

- Blocking send / Non-blocking recv
  - Sender: MPI_Send
  - Recipient: MPI_Irecv, MPI_Wait

- Non-blocking send / Non-blocking recv
  - Sender: MPI_Isend, MPI_Wait
  - Recipient: MPI_Irecv, MPI_Wait
int MPI_Test ( MPI_Request *req, int *flag, MPI_Status *sta );

Write true (non-zero value) in *flag if request *req is over.

If *flag is true, *req is assigned to MPI_REQUEST_NULL and *sta is filled.

If *flag is false, values of *req and *sta are not guaranteed.
Non-Blocking Communication

Example:

```c
MPI_Irecv(msg, N, MPI_BYTE, dest, tag, comm, &req);
double {
    instruction1;
    ...
    instructionN;
    MPI_Test(&req, &flag, &sta);
} while( !flag );
```
Non-Blocking Communication

```c
int MPI_Waitall (  
    int nb_req(in),  
    MPI_Request *tab_req(inout),  
    MPI_Status *tab_sta(out)  
);
```

Return when `nb_req` requests located in array `tab_req` are completed.

Status of communications are available as output in array `tab_st`.a.

Remark:
Order of communication completion is not important
Non-Blocking Communication

Example: send/receive with left/right neighbors

```c
MPI_Request req[4];
MPI_Status sta[4];

left = (rang + P - 1) % P;
right = (rang + 1) % P;

MPI_Isend(&x[1], 1, MPI_DOUBLE, left, tag, comm, req);
MPI_Isend(&x[N], 1, MPI_DOUBLE, right, tag, comm, req+1);
MPI_Irecv(&x[0], 1, MPI_DOUBLE, left, tag, comm, req+2);
MPI_Irecv(&x[N+1], 1, MPI_DOUBLE, right, tag, comm, req+3);

MPI_Waitall(4, req, sta);
```
Other Available Functions

- MPI proposes multiple functions to complete non-blocking communications

- MPI_Testall
  - Test is all requests as input are completed

- MPI_Waitany / MPI_Testany
  - Wait/Test until at least one request is completed
  - Return index of completed request

- MPI_Waitsome / MPI_Testsome
  - Wait/Test until at least one request is completed
  - Return set of completed requests