MPI

MESSAGE PASSING INTERFACE

David COLIGNON, ULiège

CÉCI - Consortium des Équipements de Calcul Intensif

http://www.ceci-hpc.be

Outline

- Introduction
- From serial source code to parallel execution
- MPI functions I
    - Global Environment
    - Point-to-Point Communication
- Exercice
- MPI functions II
    - Collective Communication
    - Global Reduction Operations
    - Communication Modes

Slides & Examples

- On every CÉCI cluster :

```
/CECI/proj/INFO0939/MPI/C/
```

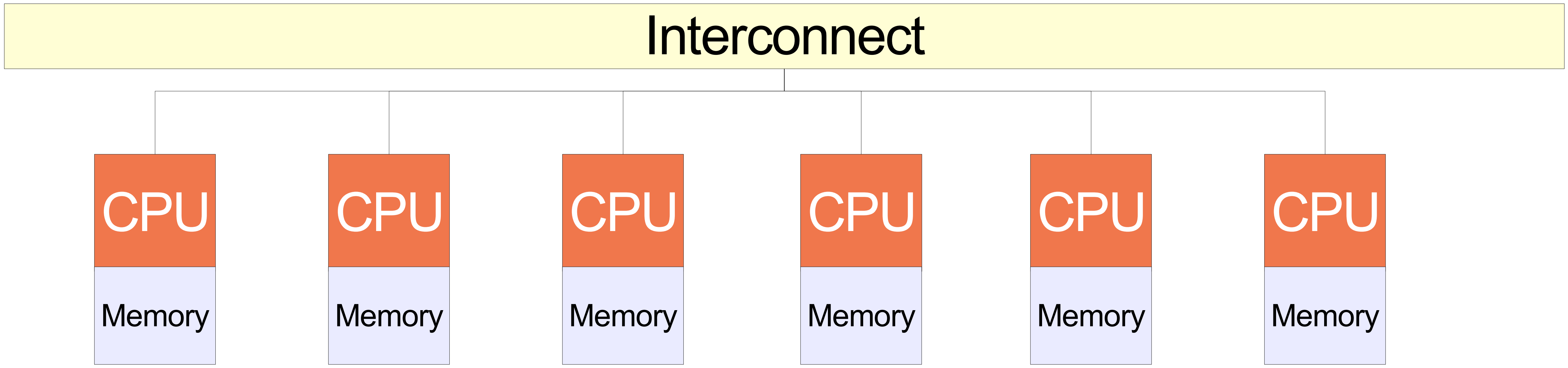
- And on NIC4:

```
module add openmpi/1.6.4/gcc-4.8.1
```

(more on this later)

Introduction: Target

Distributed Memory



Each server/node has its own memory
From one cpu per node to 2 or 4 multicore cpus...
→ Each core has its separate address space

Introduction: Goal

- What is parallel programming ?
- What are the (your) goals ?
    -
    -

Introduction: Goal

- What is parallel programming ?
- What are the (your) goals ?
    - Decrase the total execution time
    - Solve bigger problems
- Solution: Partition the work so that all nodes/cpus work together at the same time
- Partitioning a problem into workable subproblems: OK

Introduction: the MPI solution

- How to partition efficiently my problem to solve it in // ?
- How can we get nodes/cpus to work in // ?
- Solution: by **exchanging messages**
- To achieve a common parallel task, data are shared by sending/receiving "messages"
- **Message Passing Interface**: most widely used standard for parallel programming

Introduction: the MPI solution

- It's not a new programming language:
  it's a library of normalized functions for
  inter-process communication
  (can be called from Fortran, C, C++, ...)
  - Every cpu runs the same executable
  - Processes communicate with each other through the "infrastructure" provided by MPI.
- At first, no need to know the details of the implementation. You just need to know how to take advantage of it

Introduction: the MPI solution

- Carefully designed to permit maximum performance on a wide variety of systems
- Emphasis on **Portability** and **Efficiency**
- Hides many details but exposes many others to the programmer
- Sometimes called the "assembly language" of parallel computing

Introduction: the MPI solution

- each core (pure MPI, one process per core) runs the **same executable** and works on its **local data**
- Data are shared by sending/receiving "messages"
- MPI functions
  - Global management of the communications
  - Point-to-Point communication
  - Global communication

From serial source code to parallel execution

```c
// serial

#include <stdio.h>


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



printf("Hello, World !");



}

//   gcc hello_1.c

// ./a.out
```

```c
// parallel

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

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

MPI_Init(&argc,&argv);

printf("Hello, World !");

MPI_Finalize();
}

//   mpicc hello_mpi_1.c

//   mpirun -np 3 ./a.out
```

Tips & Tricks

- Questions: Which compilers are available ? Where ?
  Is MPI installed ? Where ?
  Which version should I use ?

- Answers: **R T F M !**

```
echo $PATH

mpi "+ TAB"

module available ; module add...

mpicc -show

gcc -v ; icc -V

which mpirun ; type mpirun
```

MPI : Overview

- All MPI function begins with **MPI_**
- All MPI function returns an error code
  (= MPI_SUCCES if OK)

```
int err;
err = MPI_*( * ) ;
```

- Each node/core runs exactly the same executable:

```
mpirun -np 4 prog.exe < input.txt

mpirun -np 4 /path_to/prog.exe < /path_to/input.txt
```

MPI: Global Environment

- A minimal MPI program (like hello_mpi.c) contains:
  - `#include <mpi.h>`
  - `MPI_Init(&argc, &argv);` before any call to a MPI function, in order to initialize the environment
  - `MPI_Finalize();` after the last call to a MPI function

MPI: Global Environment

- A **Communicator** is a pool of processes that can communicate together
- MPI_COMM_WORLD is the default communicator which contains all the active processes
  ```
  mpirun -np 8 [-machinefile mach.txt] ./a.out
  ```
- In a communicator, each process get identified by his **rank** , from 0 to (np - 1)

MPI: Global Environment

- A process can get back the total number of processes in a communicator with:

```
int nbproc ;
MPI_Comm_size( MPI_COMM_WORLD, &nbproc )
```

- A process can know his rank inside the communicator with:

```
int myrank ;
MPI_Comm_rank( MPI_COMM_WORLD, &myrank )
```

Hello, World ! 2.0

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char **argv) {
  int nbproc, myrank ;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank( MPI_COMM_WORLD, &myrank)
  MPI_Comm_size( MPI_COMM_WORLD, &nbproc)
  printf("Hello, World ! from proc %d of %d",
                                 myrank, nbproc );
  MPI_Finalize();
}
```

Point-to-Point Communication

- Bilateral communication between two processes (emitter and receiver), identified by their rank in their common communicator
- SEND and RECEIVE are mandatory
- Message = Envelope + Body
- Envelope:
    - rank of the source process
    - rank of the destination process
    - tag (integer) to classify the messages
    - communicator

Point-to-Point Communication

- Message = Envelope + Body
- Body:
    - buffer (start of the memory area where the data are located)
    - count (number of elements)
    - datatype

MPI Type	C Type
MPI_INT	int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_CHAR	signed char
MPI_PACKED	

MPI_Send & MPI_Recv

- `MPI_Send( BUF, COUNT, DTYPE,`
  `DEST, TAG, COMM )`

- `MPI_Recv( BUF, COUNT, DTYPE,`
  `SOURCE, TAG, COMM, STATUS )`

  - envelope (`SOURCE, TAG, COMM`) determines which message can be received
  - "wild cards" `MPI_ANY_SOURCE` & `MPI_ANY_TAG` can be used
  - `STATUS` contains `SOURCE` & `TAG` (if wild cards were used), and number of received data

MPI_Send & MPI_Recv

- `MPI_Recv( BUF, COUNT, DTYPE,`
  `SOURCE, TAG, COMM, STATUS )`
- STATUS is a structure (`MPI_Status mystatus`) that contains three fields named `MPI_SOURCE` , `MPI_TAG` , and `MPI_ERROR` (The structure may contain additional fields.) `mystatus.MPI_SOURCE` , `mystatus.MPI_TAG` and `mystatus.MPI_ERROR` contain the source, tag, and error code of the received message.
- The count argument specified to the receive routine is the number of elements for which there is space in the receive buffer. This will not always be the same as the number of elements actually received.
- `MPI_Get_count( STATUS , DTYPE , COUNT )`

MPI_Send & MPI_Recv

- `MPI_Send( BUF, COUNT, DTYPE,`
  `DEST, TAG, COMM )`
- `MPI_Recv( BUF, COUNT, DTYPE,`
  `SOURCE, TAG, COMM, STATUS )`

  - error if received message longer than expected (by `COUNT` and `DTYPE`)
  - `DTYPE` of `MPI_SEND` and `MPI_RECV` must match

MPI_Send & MPI_Recv

- `MPI_Send` and `MPI_Recv` are **blocking** ! operation must be completed (...) before jump to next instruction
- **Asynchronous** communication: possible delay between Send and Receive, sent data could be buffered. Even if Send is completed, it doesn't always mean that message has already been received
- Be cautious with **Deadlocks**: two processes waiting for a message that never come

MPI_Send & MPI_Recv : Deadlocks

```
// this code hangs !
if( myrank == 0 ) {
   MPI_Recv( &b, 100, MPI_DOUBLE,
                  1, 39, MPI_COMM_WORLD, status );
   MPI_Send( &a, 100, MPI_DOUBLE,
                  1, 17, MPI_COMM_WORLD);

else if ( myrank ==1 ) {
   MPI_Recv( &b, 100, MPI_DOUBLE,
                  0, 17, MPI_COMM_WORLD, status );
   MPI_Send( &a, 100, MPI_DOUBLE,
                  0, 39, MPI_COMM_WORLD );
}
```

MPI_Send & MPI_Recv : Deadlocks

```
~~// this code hangs !~~
if( myrank == 0 ) {
   MPI_Recv( &b, 100, MPI_DOUBLE, &
                  1, 39, MPI_COMM_WORLD, &status );
   MPI_Send( &a, 100, MPI_DOUBLE, &
                  1, 17, MPI_COMM_WORLD ); }

else if ( myrank == 1 ) {
   MPI_Send( &a, 100, MPI_DOUBLE,
                  0, 39, MPI_COMM_WORLD );
   MPI_Recv( &b, 100, MPI_DOUBLE,
                  0, 17, MPI_COMM_WORLD, &status ); }
```

Exercice: Sum of the first N Integers

```
// serial solution

int main() {

  int N = 1000, sum = 0, i ;

  for( i = 1 ; i<= N ; i++)

     sum = sum + i ;

  }

  printf(" The sum from 1 to %d is: %d", N, sum);

}
```

Sum of the first N Integers: Parallel

- How to Partition ?
- Magic Formula:

```
startval = N * myrank     / nbproc + 1
endval   = N * (myrank+1) / nbproc
```

- ! Caution ! Integer division
- Process of rank 0 receive partial sums and add (and also calculate its part !)

Exercice: Sum of the first N Integers

```
// SOME HINTS
#include <stdio.h>
#include <mpi.h>
int myrank, nbproc, mytag=23, N=1000 ;
MPI_Status  mystatus ;
MPI_Init(&argc, &argv) ;
MPI_Comm_rank( MPI_COMM_WORLD , &myrank ) ;
MPI_Comm_size( MPI_COMM_WORLD , &nbproc ) ;
MPI_Send( &aaaa, 1, MPI_INT,
                dest, mytag, MPI_COMM_WORLD ) ;
MPI_Recv( &bbbb,  1, MPI_INT,
                from, mytag, MPI_COMM_WORLD, mystatus ) ;
MPI_Finalize();
```

MPI References

- MPI standard :
  http://www.mpi-forum.org/
- MPICH :
  http://www.mpich.org/
- Open-MPI :
  http://www.open-mpi.org/
- Where can I learn about MPI ? Are there tutorials available ?
  http://www.open-mpi.org/faq/?category=all
- epcc "Writing Message Passing Parallel Programs with MPI"
  http://www.ia.pw.edu.pl/~ens/epnm/mpi_course.pdf
- "MPI par l'exemple" :
  http://algernon.cism.ucl.ac.be/mpi/mpi.html

MPI References (2)

- ME964: High-Performance Computing for Engineering Applications (Dan Negrut) http://sbel.wisc.edu/Courses/ME964/2008/LectureByLecture/me964Nov11.pdf
- 03-29-2011 - Running MPI on Newton. MPI Point-to-Point and Collective Communication. http://sbel.wisc.edu/Courses/ME964/2011/Lectures/lecture0329.pdf
- HLRS - Parallel Programming Workshop ONLINE https://fs.hlrs.de/projects/par/par_prog_ws/ https://fs.hlrs.de/projects/par/par_prog_ws/pdf/mpi_1_rab.pdf
- A Comprehensive MPI Tutorial Resource http://mpitutorial.com/
- ...