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 CÉCl 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?
What is parallel programming?

What are the (your) goals?
- Decrease 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

<table>
<thead>
<tr>
<th>Serial Code</th>
<th>Parallel Code</th>
</tr>
</thead>
</table>
| ```c
#include <stdio.h>

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

    printf("Hello, World !");

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

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

    MPI_Init(&argc,&argv);
    printf("Hello, World !");
    MPI_Finalize();

} ``` |

<table>
<thead>
<tr>
<th>Serial Compile and Run</th>
<th>Parallel Compile and Run</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc hello_1.c</td>
<td>mpicc hello_mpi_1.c</td>
</tr>
<tr>
<td>./a.out</td>
<td>mpirun -np 3 ./a.out</td>
</tr>
</tbody>
</table>
Tips & Tricks

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

• Answers: RTFM!
  
  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_SUCCESS 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:

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

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

```c
int myrank;
MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
```
#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();
}

Hello, World ! 2.0
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

<table>
<thead>
<tr>
<th>MPI Type</th>
<th>C Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_INT</td>
<td>int</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>MPI_CHAR</td>
<td>signed char</td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td></td>
</tr>
</tbody>
</table>
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 );
}


// 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:

\[
\begin{align*}
\text{startval} &= N \times \text{myrank} \quad / \quad \text{nbproc} + 1 \\
\text{endval} &= N \times (\text{myrank} + 1) \quad / \quad \text{nbproc}
\end{align*}
\]

• ! 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();
Exercice: Sum of the first N Integers

// parallel solution
#include <stdio.h>
#include <mpi.h>

int main(int argc, char *argv[]){
    int myrank, np, i, j;
    int startval, endval, partial_sum, temp_sum, N=1000;
    MPI_Status mystatus1;
    MPI_Init( &argc , &argv );
    MPI_Comm_size( MPI_COMM_WORLD , &np );
    MPI_Comm_rank( MPI_COMM_WORLD , &myrank );
    startval = N * myrank / np + 1
    endval  =  N * (myrank+1) / np
    partial_sum = 0 ; tmp_sum = 0
    for( i=startval ; i<=endval ; i++ )
        partial_sum = partial_sum + i ;
    printf("Partial sum from %d to %d on proc %d equals %d",
        startval, endval, myrank,  partial_sum ) ;
}
Exercice: Sum of the first N Integers

```c
if( myrank != 0 )
    MPI_Send( &partial_sum , 1 , MPI_INT ,
              0 , 23 , MPI_COMM_WORLD );
else
    for( j=1 ; j<np ; j=j+1 ) {
        MPI_Recv( &temp_sum , 1 , MPI_INT ,
                   j , 23 , MPI_COMM_WORLD , &mystatus1);
        partial_sum = partial_sum + temp_sum ;
    }
if( myrank == 0 )
    printf(" The sum from 1 to %d is: %d ",
            N , partial_sum );
MPI_Finalize();
```
The standard send has the following form

```
MPI_SEND (buf, count, datatype, dest, tag, comm)
```

where

- `buf` is the address of the data to be sent.
- `count` is the number of elements of the MPI datatype which `buf` contains.
- `datatype` is the MPI datatype.
- `dest` is the destination process for the message. This is specified by the rank of the destination process within the group associated with the communicator `comm`.
- `tag` is a marker used by the sender to distinguish between different types of messages. Tags are used by the programmer to distinguish between different sorts of message.
- `comm` is the communicator shared by the sending and receiving processes. Only processes which have the same communicator can communicate.
The format of the standard blocking receive is:

```c
MPI_RECV (buf, count, datatype, source, tag, comm, status)
```

where

- `buf` is the address where the data should be placed once received (the receive buffer). For the communication to succeed, the receive buffer must be large enough to hold the message without truncation — if it is not, behaviour is undefined. The buffer may however be longer than the data received.
- `count` is the number of elements of a certain MPI datatype which `buf` can contain. The number of data elements actually received may be less than this.
- `datatype` is the MPI datatype for the message. This must match the MPI datatype specified in the send routine.
- `source` is the rank of the source of the message in the group associated with the communicator `comm`. Instead of prescribing the source, messages can be received from one of a number of sources by specifying a wildcard, `MPI_ANY_SOURCE`, for this argument.
- `tag` is used by the receiving process to prescribe that it should receive only a message with a certain tag. Instead of prescribing the tag, the wildcard `MPI_ANY_TAG` can be specified for this argument.
- `comm` is the communicator specified by both the sending and receiving process. There is no wildcard option for this argument.
- If the receiving process has specified wildcards for both or either of `source` or `tag`, then the corresponding information from the message that was actually received may be required. This information is returned in `status`, and can be queried using routines described later.
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
Collective Communications

• Global function called within all the processes of a specified communicator

• cannot interfere with p2p communications

• All the processes must call the same sequence of global functions in the same order

• `MPI_Barrier( MPI_COMM_WORLD) ;`
  blocks the calling process untill all others
We now present a parallel implementation of Gaussian elimination with back substitution. As a model problem, we solve for the interpolating polynomial of the Runge function (see section 3.1.4) by forming a Vandermonde matrix based on the Chebyshev points. Recall that the goal is to find the polynomial coefficients by solving the system $Ax = b$ where $A$ is the Vandermonde matrix and $b$ is the function of interest evaluated at the interpolation points.

To better explain the code, we have broken the entire program into six parts, labeled part one through part six. The six parts break down the code as follows:

1. Part 1 - MPI initialization/setup and initial memory allocations.
2. Part 2 - Generation of the matrix rows local to each process.
3. Part 3 - Gaussian elimination of the augmented matrix.
4. Part 4 - Preparation for back substitution.
5. Part 5 - Back substitution to find the solution.
6. Part 6 - Program finalization and clean-up.

For each part, we will first present the code and then present a collection of remarks elucidating the salient points within each part.

**Part 1 - MPI initialization**

```c
#include <iostream.h>
#include <iomanip.h>
#include "SCmathlib.h"
#include "SCchapter3.h"
#include<mpi.h>

void ChebyVandermonde(int npts, double *A, int row);
```

**Broadcast: MPI_Bcast**

Figure 9.7: *MPI_Bcast* schematic demonstrating a broadcast of two data objects from process zero to all other processes.
MPI_Bcast

Tree Implementation
• **MPI_Bcast( buffer, count, datatype, root, comm)**

• A broadcast has a specified root process and every process receives one copy of the message from the root.

• All processes must specify the same root (and communicator).

• The root argument is the rank of the root process.

• The buffer, count and datatype arguments are treated as in a point-to-point send on the root and as in a point-to-point receive elsewhere.
#include <stdio.h>
#include <mpi.h>

int main( int argc , char *argv[] ) {
    int myrank, np ;
    double param=0 ;

    MPI_Init( &argc , &argv );
    MPI_Comm_size( MPI_COMM_WORLD , &np );
    MPI_Comm_rank( MPI_COMM_WORLD , &myrank );

    if( myrank == 0 ) param = 23.7853 ;

    MPI_Bcast( &param , 1 , MPI_DOUBLE,
                 0 , MPI_COMM_WORLD ) ;

    printf("On proc %d , after broadcast, param = %g" ,
            myrank, param ) ;

    MPI_Finalize() ;
}
MPI_BARRIER (COMM) blocks the calling process until all other group members have called it.

In one phase of a computation, all processes participate in writing a file. The file is to be used as input data for the next phase of the computation. Therefore no process should proceed to the second phase until all processes have completed phase one.

7.2  Broadcast, scatter, gather, etc.

Figure 24: Schematic illustration of broadcast/scatter/gather operations. The circles represent processes with ranks as shown. The small boxes represent buffer space and the letters represent data items. Receive buffers are represented by the empty boxes on the “before” side, send buffers by the full boxes.

This set of routines distributes and re-distributes data without performing any operations on the data. The routines are shown schematically in Figure 24. The full set of routines is as follows, classified here according to the form of the routine call.

7.2.1 MPI_BCAST

A broadcast has a specified root process and every process receives one copy of the message from the root. All processes must specify the same root (and communicator).

MPI_BCAST (buffer, count, datatype, root, comm)

The root argument is the rank of the root process. The buffer, count and datatype arguments are treated as in a point-to-point send on the root and as in a point-to-point receive elsewhere.

7.2.2 MPI_SCATTER, MPI_GATHER

These routines also specify a root process and all processes must specify the same root (and communicator). The main difference from MPI_BCAST is that the send and receive details are in general different and so must both be specified in the argument lists. The argument lists are the same for both routines, so only MPI_SCATTER is shown here.
MPI_Scatter
MPI_Scatter

- **MPI_Scatter** (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

- Specify a root process and all processes must specify the same root (and communicator)

- The main difference from **MPI_Bcast** is that the send and receive details are in general different and so must both be specified in the argument lists

- Note that the sendcount (at the root) is the number of elements to send to each process, not to send in total. Therefore if sendtype = recvtype, sendcount = recvcount.

- The sendbuf, sendcount, sendtype arguments are significant only at the root
```
#include <stdio.h>
#include <mpi.h>

int main( int argc , char *argv[] ) {
    int i, myrank, np, sendcount, recvcount=1;
    double array[8], myparam;
    MPI_Init( &argc , &argv );
    MPI_Comm_size( MPI_COMM_WORLD , &np );
    MPI_Comm_rank( MPI_COMM_WORLD , &myrank );

    if( myrank == 0 ) {
        for( i=0 ; i<8 ; i++)
            array[i] = 10000. + i*i;
        sendcount = 1;
    }
    MPI_Scatter( array , sendcount , MPI_DOUBLE ,
                 &myparam , recvcount , MPI_DOUBLE ,
                 0 , MPI_COMM_WORLD );

    printf("On proc %d, after scatter, myparam = %g" ,
            myrank , myparam );
    MPI_Finalize();
}
```
MPI_Gather
MPI_Gather

- **MPI_Gather( sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm )**

- The argument list is the same as for MPI_Scatter
- Specify a root process and all processes must specify the same root (and communicator)
- Note that the recvcount (at the root) is the number of elements to be received from each process, not in total. Therefore if sendtype = recvtype, sendcount = recvcount
- The recvbuf, recvcount, recvtype arguments are significant only at the root
- datas in recvbuf are held by rank order
MPI_Gather Example

```c
#include <stdio.h>
#include <mpi.h>
int main( int argc , char *argv[] ) {
    int i, myrank, np, sendcount=1, recvcount=1;
    double array[8], myparam;
    MPI_Init( &argc , &argv );
    MPI_Comm_size( MPI_COMM_WORLD , &np );
    MPI_Comm_rank( MPI_COMM_WORLD , &myrank );
    myparam = 20000. + myrank*myrank ;

    MPI_Gather( &myparam, sendcount , MPI_DOUBLE ,
                array , recvcount , MPI_DOUBLE ,
                0 , MPI_COMM_WORLD ) ;

    if( myrank == 0 )
        for( i=0 ; i < 8 ; i++ )
            printf("On proc %d , after gather, array[%d] = %g ",
                    myrank, i, array[i] ) ;

    MPI_Finalize();
}
```
### MPI_Alltoall

The `MPI_Alltoall` function is used to distribute data from one process to all processes in a communicator. It is a collective operation, meaning it should be called by all processes within the communicator.

#### Remarks
- **MPI Alltoall** is a collective operation (i.e., it should be called by all processes within the communicator).
- Both the `sendbuf` and `recvbuf` arrays are relevant on all processes in the communicator.
- In most cases, the `sendtype` and `recvtype` are identical, and the value of `sendcount` and the value of `recvcount` are identical. MPI requires that the amount of data sent (`sendcount` times the size in bytes of the datatype `sendtype`) equals the amount of data received (`recvcount` times the size in bytes of the datatype `recvtype`) per process/root pair.
- The allocated size of both the `sendbuf` and `recvbuf` arrays should be at least equal to the value of `recvtype` times the number of processes (`totalnodes`).

#### Figure 10.1: MPI Alltoall schematic demonstrating data distribution to all processes of two data objects from each process.

```c
#include <iostream.h>
#include <iomanip.h>
#include "SCmathlib.h"
#include "SCchapter3.h"
#include<mpi.h>

void ChebyVandermonde(int npts, double *A, int row);

// Global variable to set size of the system
const int size = 10;

MPI_Alltoall( sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm )
```
// parallel solution
#include <stdio.h>
#include <mpi.h>
int main(int argc, char *argv[]){
    int myrank, nbproc, i, j;
    int startval, endval, partial_sum, tmp_sum, N=1000;
    MPI_Status status;
    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &nbproc );
    MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
    startval = N * myrank / nbproc + 1
    endval = N * (myrank+1) / nbproc
    partial_sum = 0; tmp_sum = 0
    for( i=startval; i<=endval; i++ )
        partial_sum = partial_sum + i;
    printf("Partial sum from %d to %d on proc %d equals %d",
            startval, endval, myrank, partial_sum );}
if( myrank != 0 )
    MPI_Send( &partial_sum , 1 , MPI_INT , 0 , 23 , MPI_COMM_WORLD ) ;
else
    for( j=1 ; j<np ; j=j+1 ) {
        MPI_Recv( &temp_sum , 1 , MPI_INT , j , 23 , MPI_COMM_WORLD , &status ) ;
        partial_sum = partial_sum + temp_sum ;
    }
if( myrank == 0 )
    printf(" The sum from 1 to %d is: %d ", N , partial_sum );
MPI_Finalize();
Sum of the first N Integers: **MPI_Reduce**

```c
MPI_Reduce( &partial_sum, &tmp_sum, 1, MPI_INT, 
            MPI_SUM, 0, MPI_COMM_WORLD ) ;

if( myrank == 0 )
    printf(" The sum from 1 to %d is: %d ",
            N, partial_sum );

MPI_Finalize();
```
MPI_Reduce
7.3.3 MPI_REDUCE

This is illustrated in Figure 25:

Figure 25:  Global reduction in MPI with MPI_REDUCE. o represents the reduction operator. The circles represent processes with ranks as shown. The small boxes represent buffer space and the letters represent data items. After the routine call, the light-shaded boxes represent buffer space with undefined contents, the dark-shaded boxes represent the result on the root. Only one of the four results is illustrated, namely AoEoloMoQ, but the other four are similar --- for example, the next element of the result is AoEoloMoQ.

MPI_REDUCE (sendbuf, recvbuf, count, datatype, op, root, comm)

All processes in the communicator must call with identical arguments other than sendbuf and recvbuf. See "Operators" on page 57 for a description of what to specify for the operator handle. Note that the root process ends up with an array of results — if, for example, a total sum is sought, the root must perform the final summation.

7.3.4 Operators

Reduction operators can be predefined or user-defined. Each operator is only valid for a particular datatype or set of datatypes.

7.3.4.1 Predefined operators

These operators are defined on all the obvious basic C and Fortran datatypes (see Table 7:). The routine MPI_MAXLOC (MPI_MINLOC) allows both the maximum (minimum) and the rank of the process with the maximum (minimum) to be found. See Table 7:
"Global reduction operations in MPI" on page 43. More details with examples can be found in the MPI document [1].

7.3.4.2 User-defined operators

To define his or her own reduction operator, in C the user must write the operator as a function of type `MPI_User_function` which is defined thus:

```c
typedef void MPI_User_function (void *invec, void *inoutvec, int *len, MPI_Datatype *datatype);
```

while in Fortran the user must write an `EXTERNAL` subroutine of the following type

```fortran
SUBROUTINE USER_FUNCTION (INVEC(*), INOUTVEC(*), LEN, TYPE)
    <type> INVEC(LEN), INOUTVEC(LEN)
    INTEGER LEN, TYPE
    ... subroutine body ...
END SUBROUTINE USER_FUNCTION
```

The operator must be written schematically like this:

```c
for(i = 1 to len)
    inoutvec(i) = inoutvec(i) o invec(i)
```

where `o` is the desired operator. When `MPI_REDUCE` (or another reduction routine is called), the operator function is called on each processor to compute the global result in a cumulative way. Having written a user-defined operator function, it has to be registered with MPI at run-time by calling the `MPI_OP_CREATE` routine.

```
MPI_OP_CREATE (function, commute, op)
```

Table 7: Predefined operators

<table>
<thead>
<tr>
<th>MPI Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_MAX</td>
<td>Maximum</td>
</tr>
<tr>
<td>MPI_MIN</td>
<td>Minimum</td>
</tr>
<tr>
<td>MPI_SUM</td>
<td>Sum</td>
</tr>
<tr>
<td>MPI_PROD</td>
<td>Product</td>
</tr>
<tr>
<td>MPI_LAND</td>
<td>Logical AND</td>
</tr>
<tr>
<td>MPI_BAND</td>
<td>Bitwise AND</td>
</tr>
<tr>
<td>MPI_LOR</td>
<td>Logical OR</td>
</tr>
<tr>
<td>MPI_BOR</td>
<td>Bitwise OR</td>
</tr>
<tr>
<td>MPI_LXOR</td>
<td>Logical exclusive OR</td>
</tr>
<tr>
<td>MPI_BXOR</td>
<td>Bitwise exclusive OR</td>
</tr>
<tr>
<td>MPI_MAXLOC</td>
<td>Maximum &amp; location</td>
</tr>
<tr>
<td>MPI_MINLOC</td>
<td>Minimum &amp; location</td>
</tr>
</tbody>
</table>
Exemple of Global Reduction Operation

- `MPI_Reduce( senbuf, recvbuf, count, datatype, operation, root, comm )`

- All processes must specify the same root (and communicator).
- Possibility to define your own reduction operation acting on your own datatype ...
- The operation must be associative (evaluation order doesn't account)
Collective Communication

Edinburgh Parallel Computing Centre

These return the operator handle `op`, suitable for use in global reduction calls. If the operator is commutative (`A o B = B o A`) — the value `commute` should be specified as `TRUE`, as it may allow MPI to perform the reduction faster.

7.3.5 MPI_ALLREDUCE, MPI_REDUCE_SCATTER, MPI_SCAN

These are variants of MPI_REDUCE. They are illustrated in Figure 26:

Figure 26: Global reduction in MPI with MPI_ALLREDUCE. The symbols are as in Figure 25:. The only difference from MPI_REDUCE is that there is no root — all processes receive the result.

RANK

0

1

2

3

4

MPI_ALLREDUCE (No Root)
MPI_Allreduce
Communication Modes

- The standard `MPI_Send` call is **blocking**: it does not return until the message data and envelope have been safely stored away so that the sender is free to modify the send buffer.

- “Completion” of a send means by definition that the send buffer can safely be re-used.

- The message might be copied directly into the matching receive buffer, or it might be copied into a temporary system buffer.

- To be continued, see good References next slide
Communication Modes

References:

- Standard Send and Recv:
  http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report/node41.htm

- Communication Modes:
  http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report/node53.htm

- "MPI par l'exemple" :
  http://algernon.cism.ucl.ac.be/mpi/mpi.html

- epcc "Writing Message Passing Parallel Programs with MPI"
  http://www.ia.pw.edu.pl/~ens/epnm/mpi_course.pdf

Section 4 page 25
MPI Tips

• Load Balancing: Distribute evenly the work among all the processes

• Minimize the communication: Because of the latency, even a zero-byte message takes an uncompressible minimum time.

• Superpose/Mix calculation and communication
## 4 Communication Modes

<table>
<thead>
<tr>
<th>Mode</th>
<th>Completion Condition</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronous Send</td>
<td>Only completes when the receive has completed.</td>
<td>MPI_Ssend</td>
</tr>
<tr>
<td>Buffered Send</td>
<td>Always completes (unless an error occurs), irrespective of whether the receive has completed.</td>
<td>MPI_Bsend</td>
</tr>
<tr>
<td>Standard Send</td>
<td>Either synchronous or buffered.</td>
<td>MPI_Send</td>
</tr>
<tr>
<td>Ready Send</td>
<td>Always completes (unless an error occurs), irrespective of whether the receive has completed.</td>
<td>MPI_Rsend</td>
</tr>
<tr>
<td>Receive</td>
<td>Completes when a message has arrived.</td>
<td>MPI_Recv</td>
</tr>
</tbody>
</table>
4 Communication Modes

• All four modes exist in both blocking and non-blocking forms.

• In the blocking forms, return from the routine implies completion.

• In the non-blocking forms, all modes are tested for completion with the usual routines (MPI_Test, MPI_Wait, etc.). See mpi_isend_irec.c
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/

• A Comprehensive MPI Tutorial Resource
  http://mpitutorial.com/

• ...