CentraleSupélec

SG6: High Performance Computing

Message Passing principles and MPI programming

Stéphane Vialle

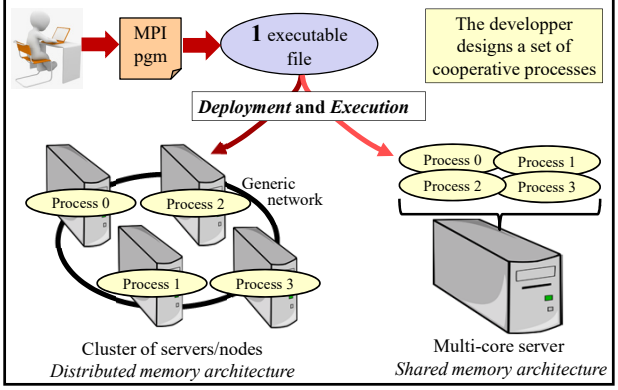
Stephane.Vialle@centralesupelec.fr
http://www.metz.supelec.fr/~vialle

CentraleSupélec

Message Passing with MPI: Programming

1 – Principles of message passing and MPI
2 – Point-to-Point communications
3 – Example: dense matrix product on a ring
4 – Collective communications

Set of processes for distributed architectures



Main difficulties of message passing

Processes have their own memory space (not shared)

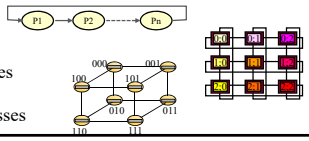
Message passing is mandatory to access data in remote process memory space

Message passing is used for genericity on shared memory machines

Design regular message passing schemes

Processes communicating according to a virtual topology are easier to manage, ex:

- virtual ring of processes
- virtual 2D torus of processes
- virtual hypercube of processes

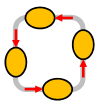


Main difficulties of message passing

Avoid *dead-locks*:

Ex: all processes waiting for a message, and no process available to send data… dead lock!

→ Schedule/plan Send and Recv operations



Minimize *latency* impact:

Time the first byte go from source to destination (*set up* of the comm)

$T_{comm}(Q) = t_s + Q/B_w = t_s + Q.t_w$ t_s : applicative latency time

1 message of 1000 data is faster than 1000 messages of 1 data

→ On each process: group communications to the same destination

Hide communication times:

Overlap communications and computations

$T = max(T_{comput}, T_{comm})$ *instead of:* $T = T_{comput} + T_{comm}$

→ Implement communication threads in parallel of computation threads

Main difficulties of message passing

Design distributed algorithms minimizing communication overheads:

Communication times are *overheads* of the parallelization

→ Design distributed algorithms:

- minimizing the amount of communications
- maximizing computation – communication overlap
- not requiring too many exchanges of small messages

Support any number of processes, or minimize the constraints:

→ Example on a virtual ring of processes:

- support to run with: **1**, 2, 3, 4, 5 … processes: perfect
- run only with: **1**, 2, 4 … processes: average
- run only with: 2, 4 … processes: uncomfortable

Main difficulties of message passing

Basic MPI instructions (C code):

Including MPI header file:

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

First MPI instruction of `main(int argc, char **argv)` function:

```
MPI_Init(&argc,&argv);
```

To know the number of run MPI processes (of the application):

```
MPI_Comm_size(MPI_COMM_WORLD,&NbP);
```

To know the process Id (from *0* up to *NbP-1*):

```
MPI_Comm_rank(MPI_COMM_WORLD,&Me);
```

Last MPI instruction of the **main** function:

```
MPI_Finalize();
```

MPI communication instructions:

Point-to-Point comms.

```
Ex : …
   MPI_Send(…);
   MPI_Recv(…);
```

Group comms.

```
Ex: …
   MPI_Bcast(…);
   …
```

MPI parallelism is very explicite!

MPI pgm example – without comms.

C code:

```
#include <stdio.h>
#include <mpi.h>
main(int argc, char **argv) {
  int Me, NbP;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&NbP);
  MPI_Comm_rank(MPI_COMM_WORLD,&Me);
  printf("Hello World from process %d/%d\n",Me,NbP);
  fflush(stdout);
  MPI_Barrier(MPI_COMM_WORLD);
  MPI_Finalize();
}
```

Group of all MPI processes of the program

To print all messages before program ending

Example of execution with 3 processes:

```
Hello World from process 0/3
Hello World from process 2/3
Hello World from process 1/3
```

No assumption about message printing order!

MPI compilation

MPI program compilation:

MPI is just a library:

```
cc -I…/include -L…/libs
   -O3 -o myAppli XXX.c YYY.c …
   -lmpi
```

or:

```
mpicc -O3 -o myAppli XXX.c YYY.c …
```

→ Generates only one executable file

MPI is compliant with multithreading:

- Compliant with OpenMP (`mpicc -O3 -fopenmp …`)
- IF MPI communication calls are achieved by only one thread at a time *(no parallelization of the communications)*

  THEN: standard MPI installation can be used

  ELSE: MPI *thread safe* installation/mode required.

MPI program deployment & run

MPI application deployment:

A virtual topology of *P* processes ←→ a cluster of *N* multicore nodes

→ Distributed application « **deployment** »

→ MPI deployment with « **`mpirun`** » command

```
mpirun
   -np <#P>                          Total nb of processes to create
   -machinefile <FileName>           List of available machines
   -mab-by … -rank-by … -bind-to …   Deployment control (see further)
   <Path/ExecName> [args]            Executable code and arguments
```

Examples:

`mpirun -np 3 ./HelloWorld` → *run 3 processes on the current PC*

```
mpirun -np 6 -machinefile mach.txt
   -map-by ppr:1:socket -rank-by socket
   -bind-to socket ./HelloWorld
```

→ *run 6 processes on 6 "sockets" of multi-processor PCs (see further)*

MPI application development & exec.

1. **« Parallel » algorithmics:**
   Distributed & Parallel & Vector algorithm design
2. **« Parallel » programming:**
   message passing + multithreading + vectorization
   → *MPI* + *OpenMP* + *vectorized kernels*
3. **Compilation**
   → production of ONE executable file (with ***`mpicc`***)
4. **Deployment strategy**
   → definition of the deployment control parameters
   (***`-map-by / -rank-by / -bind-to`***)
5. **Deployment & execution**
   → copy the binary file on each node, or mount a shared directory
   → deploy and run the MPI application (with ***`mpirun`***)

```
mpirun -np <#P> -machinefile machines.txt
       -map-by … -rank-by … -bind-to … ./MyProg ……
```

Message Passing with MPI: Programming

Available communications

Mode\Type	Not specified	Buffered	Synchronous	Ready
Blocking	`MPI_Send` `MPI_Recv`	`MPI_Bsend` `MPI_Recv`	`MPI_Ssend` `MPI_Recv`	`MPI_Rsend` `MPI_Recv`
Non-blocking	`MPI_Ibsend` `MPI_Irecv`	`MPI_Ibsend` `MPI_Irecv`	`MPI_Issend` `MPI_Irecv`	`MPI_Irsend` `MPI_Irecv`

+ `MPI_Sendrecv`
`MPI_Sendrecv_replace`

Combined and blocking point-to-point comms.

General MPI communication syntax

Sending & Receiving data:

```
MPI_Send(address, n, MPI_DOUBLE, dest, …);
```

→ read data from *address*, and send *n*×*sizeof(double)* bytes to process numbered *dest*

```
MPI_Recv(address, n, MPI_DOUBLE, src, …);
```

→ Receive (and accept) *n*×*sizeof(double)* bytes from process numbered *src* and write these data in memory at *address*

Predefined datatypes:

`MPI_CHAR`	`MPI_SHORT`	`MPI_FLOAT`	`MPI_UNSIGNED_CHAR`
`MPI_BYTE`	`MPI_INT`	`MPI_DOUBLE`	`MPI_UNSIGNED_SHORT`
	`MPI_LONG`	`MPI_LONG_DOUBLE`	`MPI_UNSIGNED`
			`MPI_UNSIGNED_LONG`

Rmk: developper can define new datatypes (arrays, vectors, structures)

Buffered & blocking comm.: Bsend/Recv

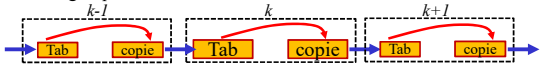
Bsend(…) :
- Make a local copy of data to send (while buffer is not full)
- Return as soon as the local copy is achieved

→ original data storage can be overwritten

Recv(…) :
- Requires data exchange & wait for (entire) data reception
- Return when all data received

Non-blocking & buffered *send,* and *blocking recv*

Ex. on a ring of processes:



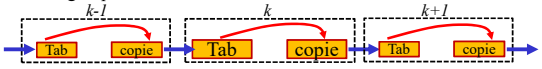
Buffered & blocking comm.: Bsend/Recv

Bsend(…) / Recv(…):
- A unique communication code for all processes

On each process:
1. Execute all Bsend(…) of the step (in any order)
2. Execute all Recv(…) of the step (in any order)

- *Rexaled synchronization* (but sufficient synchronization)
- But… local copy buffer has to be managed by the developer…

Ex. on a ring of processes:



Unique code :

```
……
Bsend(Tab,…,Me+1,…);
Recv(Tab,…,Me-1,…);
……
```

→ Simple communication schedule!

Buffered & blocking comm.: Bsend/Recv

Bsend(…) / Recv(…):

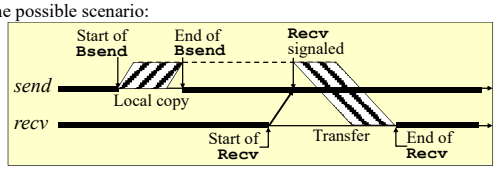
```
MPI_Bsend(data_adr,count,datatype,destproc,tag,comm)
MPI_Recv(data_adr,count,datatype,srcproc,tag,comm,stts_adr)
```

tag: Only *send* and *recv* with identical *tag* can match
(tag can remain at 0… or set to the step of the loop…)

comm: Id of the group of the processes including *destproc* and *srcproc*
→ MPI_COMM_WORLD *: group of all processes of the run*

stts_adr: address where MPI will store the balance sheet of the comm.

One possible scenario:



Buffered & blocking comm.: Bsend/Recv

Bsend(…) / Recv(…):

Developper has to *size*, *allocate*, *attach*, *detach*, and *free* the local copy buffer

```
                                          // Buffer size comput
MPI_Pack_size(n,MPI_DOUBLE,MPI_COMM_WORLD,&size1Msg);
sizeBuff = m*(size1Msg + MPI_BSEND_OVERHEAD);
ptBuff = (double *) malloc(sizeBuff);// Buffer allocation
MPI_Buffer_attach(ptBuff,sizeBuff);  // Buffer attachment
for (i=0; i<m; i++) {                // Message sending
  tab = …;                                 // - tab update
  MPI_Bsend(tab,n,MPI_DOUBLE,suivant,…);  // - msg passing
  MPI_Recv(tab,n,MPI_DOUBLE,precedent,…);
}
MPI_Buffer_detach(&ptBuff,&sizeBuff);// Buffer detachement
free(ptBuff);                        // Buffer dealloc.
```

Wait for all buffered message have been sent

Warning: `sizeBuff` can be huge!

One *overhead* per msg to store into the buffer

Point-to-Point communications

Buffered & blocking comm.: Bsend/Recv

Bsend(…) / Recv(…):

Developper has to *size*, *allocate*, *attach*, *detach*, and *free* the local copy buffer

```
                                          // Buffer size comput
MPI_Pack_size(n,MPI_DOUBLE,MPI_COMM_WORLD,&size1Msg);
sizeBuff = 1*(size1Msg + MPI_BSEND_OVERHEAD);
ptBuff = (double *) malloc(sizeBuff);// Buffer allocation
for (i=0; i<m; i++) {                     // Message sending
  MPI_Buffer_attach(ptBuff,sizeBuff);       // - attachment
  tab = …;                                  // - tab update
  MPI_Bsend(tab,n,MPI_DOUBLE,suivant,…);    // - msg passing
  MPI_Recv(tab,n,MPI_DOUBLE,precedent,…);
  MPI_Buffer_detach(&ptBuff,&sizeBuff);     // - detachment
}
free(ptBuff);                             // Buffer dealloc.
```

At each step: wait for all buffered message have been sent **(longer)**

One *overhead* per msg to store into the buffer

Point-to-Point communications

Synchronous & blocking comm.: Ssend/Recv

Ssend(…):

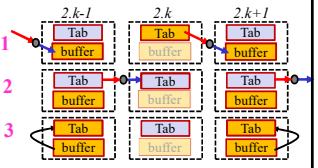
- Requires data exchange & wait for (entire) data emission

→ Original data storage must not be overwritten

Recv(…):

- Requires data exchange & wait for (entire) data reception
- Return when all data received

→ Communications are *appointments* (***Rendez-Vous***) between processes:



Some buffers are still required to avoid data losses

Point-to-Point communications

Synchronous & blocking comm.: Ssend/Recv

Ssend(…) / Recv(…):

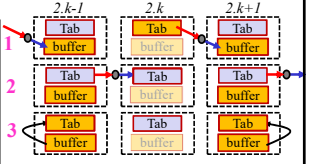
- At each step a *Ssend* has to match a *Recv* operation

→ **A communication schedule has to be entirely and finely designed:** to plan each *Ssend/Recv* appointment and to avoid dead-locks!

- **Execution of the communication schedule will be longer** than with Bsend/Recv: 1st half of the comms (1), then 2nd half of the comms (2)

Ex. on a ring of processes:

```
……
if (processId %2 == 0)
  1 : Ssend(Tab,…,Me+1,…);
  2 : Recv(Tab,…,Me-1,…);
else
  1 : Recv(buffer,…,Me-1,…);
  2 : Ssend(Tab,…,Me+1,…);
  3 : permut(buffer,Tab);
……
```



- Ssend/Recv: longer and higher dead-lock risk than Bsend/Recv…!

Point-to-Point communications

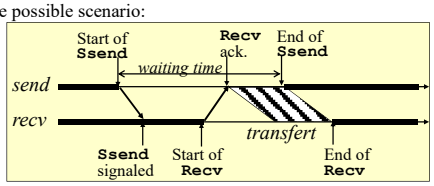
Synchronous & blocking comm.: Ssend/Recv

Ssend(…) / Recv(…):

```
MPI_Ssend(data_adr,count,datatype,destproc,tag,comm)
MPI_Recv(data_adr,count,datatype,srcproc,tag,comm,stts_adr)
```

→ Identical syntax to Bsend/Recv communications
But different behavior!

One possible scenario:



Point-to-Point communications

« Standard & Blocking » comm.: Send/Recv

Send(…): not entirely specified !

- **Allows constructors to implement optimizations** function of their architecture
- **Not a portable communication mechanism**

Example function of the message size:

- Under some threshold:
  → runs **like a Bsend** with automatic buffer management
- Above some threshold:
  → Runs **like a Ssend** with rendez-vous protocol

Recv(…): unchanged

- Requires data exchange & wait for (entire) data reception
- Return when all data received

Point-to-Point communications

« Standard & Blocking » comm.: Send/Recv

Send(…) / Recv(…) : not entirely specified !

- **Allows constructors to implement optimizations** function of their architecture
- **Not a portable communication mechanism**

2 opposed approaches:

- A MPI pgm should never used standard-blocking comms.
  → Portability is the main objective

PC cluster

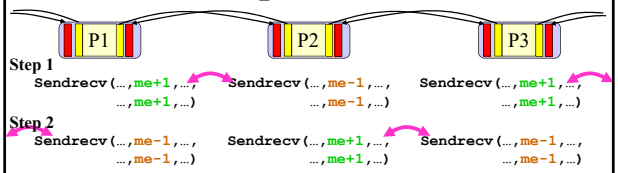
- A MPI pgm should use standard-blocking comms.
  → Efficiency of the communication is the main objective
  And a clear documentation on the standard protocol is available

Super-computers

Combined & Blocking comm.: Sendrecv

MPI_Sendrecv(…) : 1 send & 1 recv, 1 operation:

```
MPI_Sendrecv(send_adr,sendcount,sendtype,destproc,sendtag,
             recv_adr,recvcount,recvtype,srcproc,recvtag,
             comm,status_adr)
```

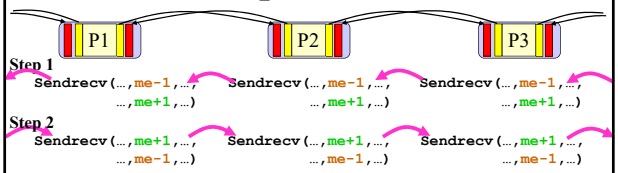


Ex: **frontier exchange** with `MPI_Sendrecv`

Combined & Blocking comm.: Sendrecv

MPI_Sendrecv(…) : 1 send & 1 recv, 1 operation:

```
MPI_Sendrecv(send_adr,sendcount,sendtype,destproc,sendtag,
             recv_adr,recvcount,recvtype,srcproc,recvtag,
             comm,status_adr)
```



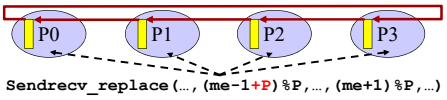
Ex: **frontier exchange** with `MPI_Sendrecv`

- Blocking comms.: returns when *Send part* and *Recv part* have completed
  → Sometimes a fine schedule of the communications is required
- **Very efficient** communications!

Combined & Blocking: Sendrecv_replace

MPI_Sendrecv_replace(…) : 1 send & 1 recv & buff management:

```
MPI_Sendrecv_replace(data_adr,count,datatype,
                     destproc,sendtag,srcproc,recvtag,
                     comm,status_adr)
```



Pb avec opérateur *modulo*

Ex: **data circulation** with `MPI_Sendrecv_replace`

- Blocking comms.: returns when *Send part* and *Recv part* have completed
- But no need for a fine schedule: just follow the circulation scheme
- Data storage must be allocated before usage
- But no buffer read/write conflicts to manage (done by the system)
- **Easy to use** & **very efficient** communications!

Available communications

Mode\Type	Not specified	Buffered	Synchronous	Ready
Blocking	*MPI_Send* *MPI_Recv*	MPI_Bsend MPI_Recv	MPI_Ssend MPI_Recv	MPI_Rsend MPI_Recv
Non - blocking	*MPI_Ibsend* *MPI_Irecv*	MPI_Ibsend MPI_Irecv	MPI_Issend MPI_Irecv	*MPI_Irsend* *MPI_Irecv*

`MPI_Sendrecv`
`MPI_Sendrecv_replace`
Communications pt-à-pt combinées et bloquantes

Portable communication routines

Asynchronous point-to-point comms.

Non-blocking Send and Recv operations:

- **Isend(…):** launch a sending data thread, and returns
- **Irecv(…):** launch a receiving data thread, and returns

→ Possible **overlap** of the communications and the next computations
But do not overwrite the data (`myTab`) before the end of the computation and the end of the send operation!
→ Use a second data buffer (`otherTab`) to receive new data

- **Wait(…):** resynchronize computations and communications: wait their end

→ It is now possible to overwrite the data (`myTab`)

```
..... // local computations
1 : Isend(myTab,…,dest,…,&Srq);  // launch a comm. thread
2 : Irecv(otherTab,…,src,…,&Rrq);// launch a comm. thread
3 : next_calcul(…)               // comput-comm overlap
4 : Wait(&Srq); Wait(&Rrq);      // comput & comm re-sync
..... // end of computations
```


Asynchronous point-to-point comms.

Non-blocking Send and Recv operations:

- With some MPI implementations: **Isend(…)** and **Irecv(…)** launch threads **remaining inactive** up to the **Wait(…)** operation !

→ **Computations and communications do not overlap!**

Solution :

- Create classic threads (Posix, OpenMP…) running blocking comms.
  → make non-blocking comms and achieve overlapping
- Implement a *barrier* / *join operation* on the death of the comm. Threads
  → resynchronize computations and communications

```
..... // local computations
1 : tidS = thread{Send(myTab,dest)};   // Comm. thread
2 : tidR = thread{Recv(otherTab,src)}; // Comm. thread
3 : next_calcul(…)                     // comput-comm overlap
4 : threadJoin(tidS, tidR);            // comput & comm re-sync
..... // end of computations
```

Asynchronous programming with overlapping is always complex!

Point-to-Point communications

Available communications

Mode\Type	Not specified	Buffered	Synchronous	Ready
Blocking	`MPI_Send` `MPI_Recv`	`MPI_Bsend` `MPI_Recv`	`MPI_Ssend` `MPI_Recv`	`MPI_Rsend` `MPI_Recv`
Non - blocking	`MPI_Ibsend` `MPI_Irecv`	`MPI_Ibsend` `MPI_Irecv`	`MPI_Issend` `MPI_Irecv`	`MPI_Irsend` `MPI_Irecv`

+ **Blocking communications run by explicit threads, to achieve non-blocking communications**

+ `MPI_Sendrecv`
  `MPI_Sendrecv_replace`
  Communications pt-à-pt combinées et bloquantes

Portable communication routines

Message Passing with MPI: Programming

1 – Principles of message passing and MPI
2 – Point-to-Point communications
3 – Example: dense matrix product on a ring
4 – Group communications

Example: dense matrix product on a ring of processes

Distributed algorithm

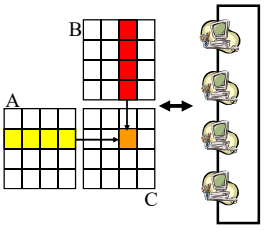
Problème à résoudre :

A, B, C : n × n = N éléments

C = A . B $\quad c_{ij} = \sum_{k=1}^{n} (a_{ik} b_{kj})$ $\quad$ O(Nbr d'op flotantes) = O($N^{3/2}$)

Comment répartir les données ?

- ~~Duplication des données~~
  → pas de *size up* possible !
- **Partitionnement** des données
  → size up possible
  → une **circulation** des données sera nécessaire



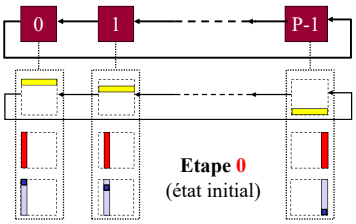
Example: dense matrix product on a ring of processes

Distributed algorithm

Partitionnement sur un anneau de processeurs :

- A partitionnée en blocs de lignes
- B et C partitionnées en blocs de colonnes

- **Circulation de A**
- B et C statiques



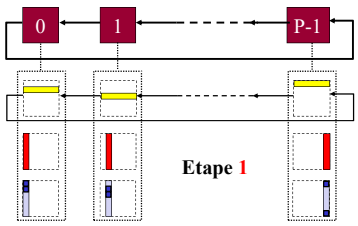
Example: dense matrix product on a ring of processes

Distributed algorithm

Partitionnement sur un anneau de processeurs :

- A partitionnée en blocs de lignes
- B et C partitionnées en blocs de colonnes

- **Circulation de A**
- B et C statiques



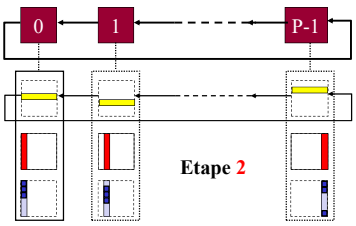
Example: dense matrix product on a ring of processes

Distributed algorithm

Partitionnement sur un anneau de processeurs :

- A partitionnée en blocs de lignes
- B et C partitionnées en blocs de colonnes

- **Circulation de A**
- B et C statiques



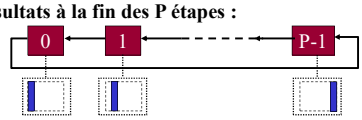
Example: dense matrix product on a ring of processes

Distributed algorithm

Partitionnement sur un anneau de processeurs :

• A partitionnée en blocs de lignes	• **Circulation de A**
• B et C partitionnées en blocs de colonnes	• B et C statiques

Résultats à la fin des P étapes :



Topologie des processus

Partitionnement statique de C

Bilan :

- Chaque PC a calculé un bloc de colonnes de C
- Les *P* PC ont travaillé en parallèle

→ **Calcul de tous les blocs de colonnes en parallèle**, en *P* étapes

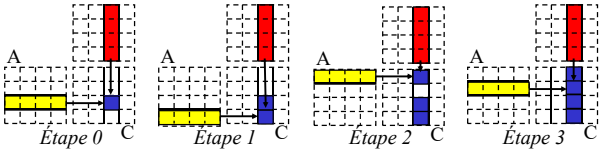
Example: dense matrix product on a ring of processes

Distributed algorithm

Partitionnement sur un anneau de processeurs :

• A partitionnée en blocs de lignes	• **Circulation de A**
• B et C partitionnées en blocs de colonnes	• B et C statiques

Déroulement de l'algorithme sur PE-2, avec *P* = *4* :



Example: dense matrix product on a ring of processes

Distributed algorithm

Stratégies d'implantation sur un anneau de P processeurs :

```
// Sans recouvrement
for (step=0; step<P; step++)
  calcul();
  barrier();    // si besoin
  circulation();
  barrier();    // si besoin
}
```

```
// Avec recouvrement
for (step=0; step<P; step++)
   thread{ calcul(); }
   thread{ circulation(); }
   joinThreads();
   permutBuff();
}
```

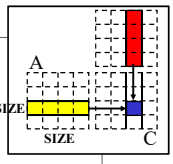
- Concevoir l'algorithme avec des barrières de (re)synchronisation potentielles
- Selon le mécanisme de communication utilisé :
  - Implanter des barrières explicites : synchronisation forte

  ou bien
  - Se contenter de la synchro des comms : synchronisation relaxée

Example: dense matrix product on a ring of processes

Distributed implementation

Blocking MPI_Sendrecv_replace version




```
int LOCAL_SIZE = SIZE/NbP;

//double A_slice[LOCAL_SIZE][SIZE];
double A_slice[LOCAL_SIZE*SIZE];

void ComputationAndCirculation()
{
 MPI_Status   status;

 // Loop: computation-circulation
 for (int step = 0; step < NbP; step++) {
   OneLocalProduct(step);
   MPI_Sendrecv_replace(
        A_slice, LOCAL_SIZE*SIZE, MPI_DOUBLE,
        (Me-1+NbPE)%NbPE, step, (Me+1)%NbPE, step,
        MPI_COMM_WORLD, &status);
 }
}
```

Example: dense matrix product on a ring of processes

Distributed algorithm

Stratégies d'implantation sur un anneau de P processeurs :

```
// Sans recouvrement
for (step=0; step<P; step++)
  calcul();
  barrier();    // si besoin
  circulation();
  barrier();    // si besoin
}
```

```
// Avec recouvrement
for (step=0; step<P; step++)
   thread{ calcul(); }
   thread{ circulation(); }
   joinThreads(); // barrier
   permutBuff();
}
```

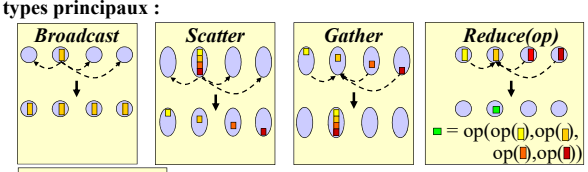
- Concevoir l'algorithme avec une barrière de (re)synchronisation et implanter la barrière de (re)synchronisation !
- Permuter les buffers de calcul et de communication (quasi-obligatoire)

Message Passing with MPI: Programming

Communications collectives

Principes des comm. collectives

5 types principaux :



+ les barrières !

Principes :

- Utilisent les *communicator* et les groupes de processus
- Opérations bloquantes
- Des variantes existent : *all-reduce*, *all-to-all*, *scatterv*, …

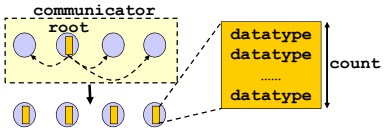
Intérêt dans un supercalculateur :

Le routage est optimisé selon le réseau sous-jacent
(arborescent – linéaire – sur bus – …)

Communications collectives

Broadcast

```
int MPI_Bcast(buffer,count,datatype,root,comm )
   void *buffer;  // Starting address of buffer
   int count;     // Number of elts in buffer (integer)
   MPI_Datatype datatype; // Data type of buffer
   int root;      //  Rank of broadcast root (integer)
   MPI_Comm comm; // Communicator
```



Chaque processus exécute `MPI_Bcast` (en émetteur ou récepteur)

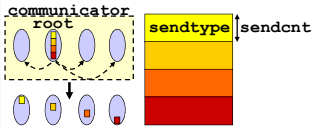
Généralisation :

`MPI_Alltoall` et `MPI_Alltoallv`

Communications collectives

Scatter

```
int MPI_Scatter(sendbuf,sendcnt,sendtype,
                recvbuf,recvcnt,recvtype, root,comm)
   void *sendbuf; // Address of send buffer
   int sendcnt;   // Nb of elements sent to each process
   MPI_Datatype sendtype; // Data type of elt to send
   void *recvbuf; // Address of receive buffer
   int recvcnt;   // Number of elements in receive buffer
   MPI_Datatype recvtype; // Data type of elt to receive
   int root;      // Rank of the sending process
   MPI_Comm comm; // Communicator
```



- Chaque processus exécute `MPI_Scatter` (en émetteur ou récepteur)
- Le buffer d'émission n'a de sens que sur le processus `root`

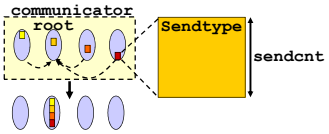
Généralisation :

`MPI_Scatterv` (avec partitionnement explicite des données)

Communications collectives

Gather

```
int MPI_Gather(sendbuf,sendcnt,sendtype,
               recvbuf,recvcount,recvtype, root,comm)
   void *sendbuf; // Starting address of send buffer
   int sendcnt;   // Number of elements in send buffer
   MPI_Datatype sendtype; // Data type of elts to send
   void *recvbuf; // Address of receive buffer
   int recvcount; // Nb of elts to receive from each proc
   MPI_Datatype recvtype; // Data type of elt to recv
   int root;      // Rank of the receiving process
   MPI_Comm comm; // Communicator
```



- Chaque processus exécute `MPI_Gather` (en émetteur ou récepteur)
- Le buffer de réception n'a de sens que sur le processus `root`

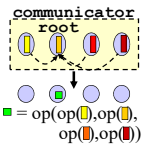
Généralisation :

`MPI_Gatherv`, `MPI_Allgather`, `MPI_Allgatherv`

Communications collectives

Reduce

```
int MPI_Reduce(sendbuf,recvbuf,count,datatype,op,
               root,comm)
   void *sendbuf; // Address of send buffer
   void *recvbuf; // Address of receive buffer
   int count;     // Number of elts in send buffer
   MPI_Datatype datatype; // Data type of elts to send
   MPI_Op op;     // Reduce operation
   int root;      // Rank of the process hosting result
   MPI_Comm comm; // Communicator
```



- Opérations de reduction disponibles :
  `MPI_MAX`, `MPI_MIN`, `MPI_SUM`, `MPI_PROD`
  `MPI_LAND`, `MPI_BAND`, `MPI_LOR`, `MPI_BOR`
  `MPI_LXOR`, `MPI_BXOR`, `MPI_MINLOC`
- Définition de nouvelles opérations avec `MPI_Op_create()`

Généralisation :

`MPI_Allreduce`, `MPI_Reduce_scatter` : les res sont redistribués

Message Passing principles
and MPI programming

Questions ?