MPI: les bases

Kitware Academy

Sommaire

1. Contexte
   - ○ Qu’est ce que MPI
   - ○ Quand l’utiliser
   - ○ Comment l’utiliser
2. Hello World
   - ○ En C / C++
   - ○ Le build system
3. Les communications
   - ○ Blocantes / Non blocantes
   - ○ Synchrones / Asynchrones
   - ○ Broadcast
4. Distribué

Contexte

MPI: Message Passing Interface

Un standard pour des routines permettant de passer des “messages”

↳données structurées

- Une norme
- Plusieurs implémentations

Utile dans un contexte distribué

Contexte

MPI: Message Passing Interface

Ce standard est définie en C

À l’utilisation: mpicc + mpirun (wrapper)

Hello World

- MPI_Init(int* argc, char **argv);
  - Avant le premier appel MPI
  - Par le thread maitre
- MPI_Finalize(void);
  - Après le dernier appel MPI
  - Par le thread maitre

MPI_Comm_Size(MPI_Comm com, int *nb_nodes);

MPI_Comm_Rank(MPI_Comm com, int *my_rank);

Communication

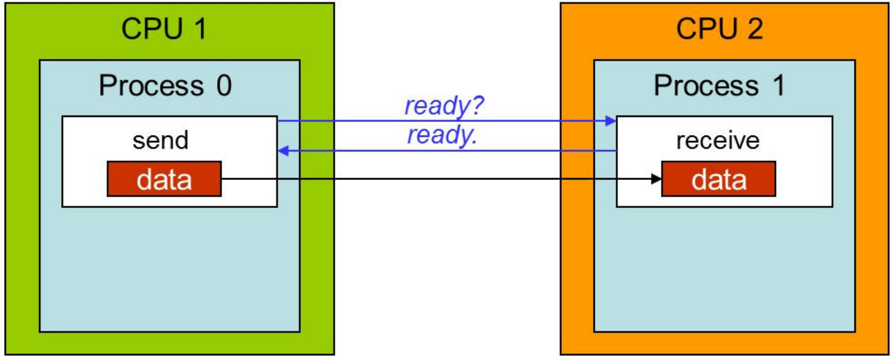
- MPI Data type:
    - MPI_INT, MPI_DOUBLE,
    - Tableau contigue de ⇧
    - Tableau indexé
    - Vecteur (tableau de blocs équidistant)
    - Structure

Communication

- Blocant: ...

MPI_Send(cont void* msg, int longueur, MPI_Datatype type, int dest, int tag, MPI_Comm com)

MPI_Recv(cont void* msg, int longueur, MPI_Datatype type, int dest, int tag, MPI_Comm com, MPI_Status* status)



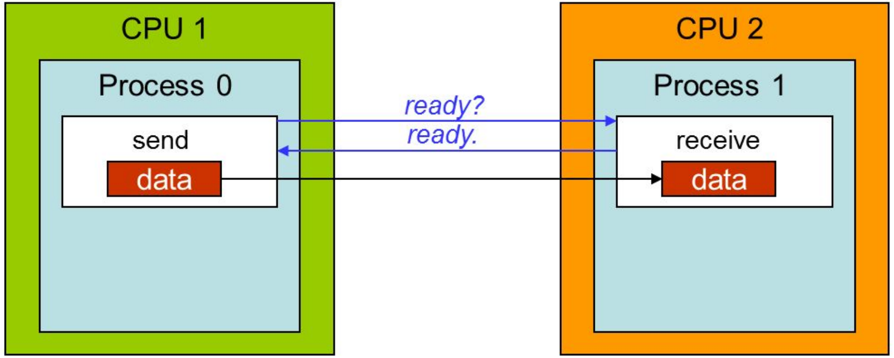
Communication

- Blocant: … la vrai réponse

MPI_Ssend(cont void* msg, int longueur, MPI_Datatype type, int dest, int tag, MPI_Comm com)

MPI_Recv(cont void* msg, int longueur, MPI_Datatype type, int dest, int tag, MPI_Comm com, MPI_Status* status)

Et oui, le **MPI_Send** est implementation dependant!



Communication

- Blocant:

MPI_Ssend(cont void* msg, int longueur, MPI_Datatype type, int dest, int tag, MPI_Comm com)

MPI_Recv(cont void* msg, int longueur, MPI_Datatype type, int dest, int tag, MPI_Comm com, MPI_Status* status)

- Non-Blocant: (Immediate)

MPI_Isend(cont void* msg, int longueur, MPI_Datatype type, int dest, int tag, MPI_Comm com, MPI_Request* req)

MPI_Irecv(cont void* msg, int longueur, MPI_Datatype type, int dest, int tag, MPI_Comm com, MPI_Request* req)

MPI_Test(MPI_Request *req, int *tag, MPI_Status *status) **et** MPI_Wait(MPI_Request *req, MPI_Status *status)

Communication

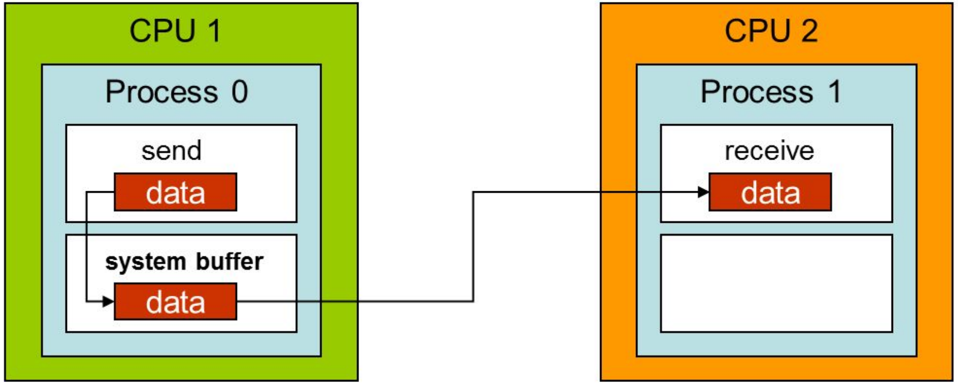
- Bufferisé:

MPI_Bsend(cont void* msg, int longueur, MPI_Datatype type, int dest, int tag, MPI_Comm com)

MPI_Recv(cont void* msg, int longueur, MPI_Datatype type, int dest, int tag, MPI_Comm com, MPI_Status* status)

- Il faut déclarer le buffer pour les rangs qui appellent Bsend:

MPI_Buffer_attach(void *buffer, int size)



Communication

The one-liner:

```
int MPI_Sendrecv(const void *sendmsg, int sendcount, MPI_Datatype sendtype,
                 int dest, int sendtag,
                 void *recvmsg, int recvcount, MPI_Datatype recvtype,
                 int source, int recvtag,
                 MPI_Comm comm, MPI_Status * status)
```

Resumé (coding game)

The **standard** mode (MPI_Send) is actually a "non-mode" that lets the MPI implementation choose which communication mode is preferable. This might be heavily dependent on the implementation. In OpenMPI, the observed behaviour is that for short messages, the send is automatically buffered while for long messages, the message will be sent using a mode somewhat close to the synchronous mode.

The **buffered** mode (MPI_Bsend) stores all the data to be sent in a temporary buffer and returns to the execution, just as a non-blocking send would do. The advantage here is that execution continues immediately even if the corresponding blocking Recv has not been called yet. On the other hand, buffered mode copies all the data of your buffer into another region of memory, duplicating the data. This might be dangerous memory wise if you are transferring large amounts of data.

The **ready** mode (MPI_Rsend) can start only if the corresponding receive has already been called. This allows your program to gain time from some additional overhead in the initialization of messages. If the corresponding Recv has not yet been called, the message that will be received //might// be ill-defined so you have to make sure that the receiving process has asked for a Recv **before** calling the ready mode.

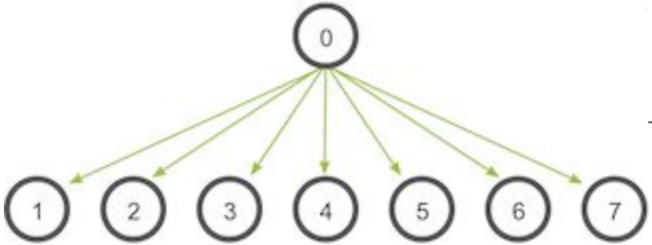
Finally, the **synchronous** mode (MPI_Ssend) will wait for the corresponding Recv to complete. The data transfer will occur at that exact moment, ensuring that both processes are ready for transfer.

Communication (groupe)

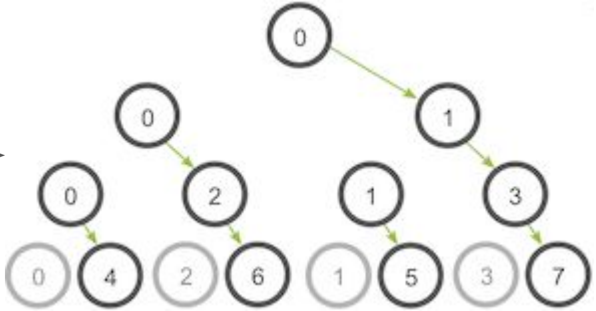
- Broadcast:

MPI_Bcast(void *msg, int count, MPI_Datatype datatype, int rang_maitre, MPI_Comm comm)

MPI_Barrier(MPI_Comm comm)



In practice

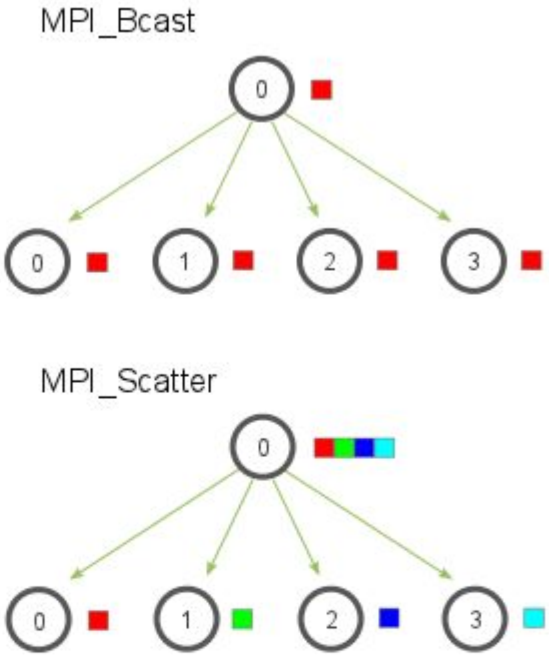


Communication (groupe)

- Scatter:

```
MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype,

   void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,

   MPI_Comm comm)
```

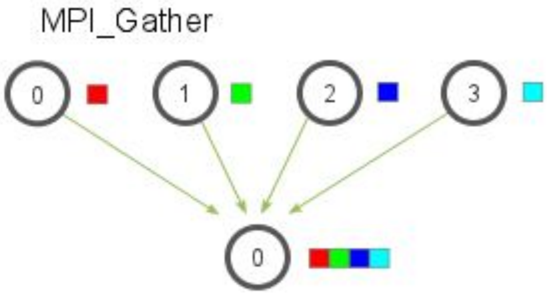
```
MPI_Iscatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype,

   void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,

   MPI_Comm comm, MPI_Request *request)
```



Communication (groupe)

- Gather

```
MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
                  void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
```

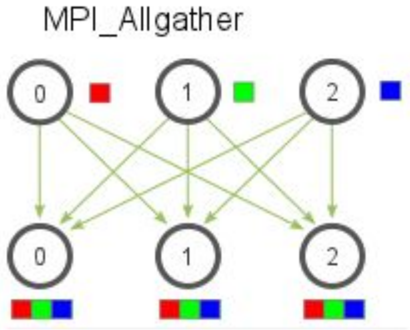


Communication (groupe)

- Allgather

MPI_Allgather(void* send_data, int send_count, MPI_Datatype send_datatype,

void* recv_data, int recv_count, MPI_Datatype recv_datatype, MPI_Comm communicator)



Distribué

- En LAN (Besoin du vpn)
- Fichier **hostfile**
    - 127.0.0.1 slots=1
    - 10.33.0…..
    - Une ligne par noeud
- Exécutable: même chemin absolue sur chaque noeud

Distribué

- Get hostname:

```
MPI_Get_processor_name(char* nom, int* taille);
```

- mpirun -np 2 --hostfile hostfile ./build/...exec
    - rank: 0 host: **Yokai**
    - rank: 1 host: **Dante**
    - rank: 2 host: **Dante**
    - process 0 recieved 100 elements.
    - recieve[0] = 0
    - recieve[1] = 1
    - ...

Questions ?

Useful resources:

- https://mpitutorial.com/
- https://www.codingame.com/playgrounds/349/introduction-to-mpi/