Introduction à MPI

Pierre LERMUSIAUX

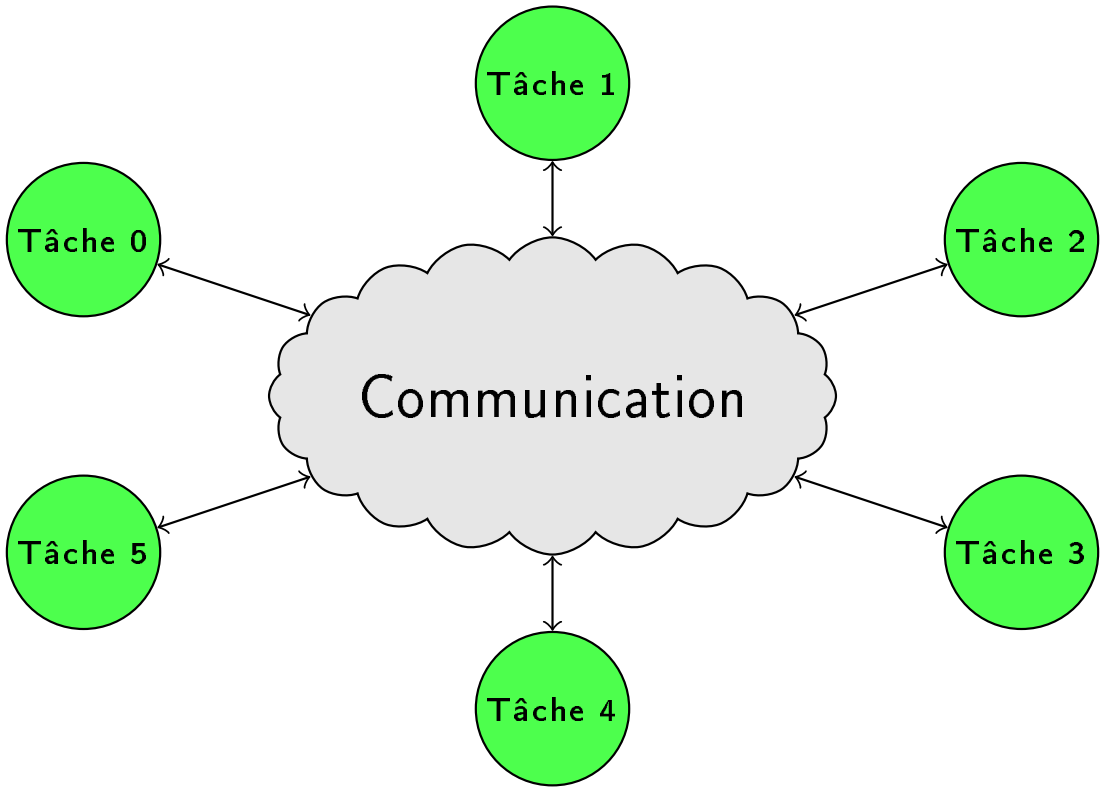
Polytech Nancy

Message Passing Interface

- Standard de développement pour la programmation parallèle
- Communication entre nœuds/process
- Adapté pour les systèmes distribuées:
  - Plusieurs tâches/process ayant leur propre mémoire
  - Coordination via des échanges de messages entre les tâches
- Modèle SPMD (Single Program Multiple Data)
- Peut s'adapter à un système à mémoire partagée utilisant des communications pour coordonner les tâches

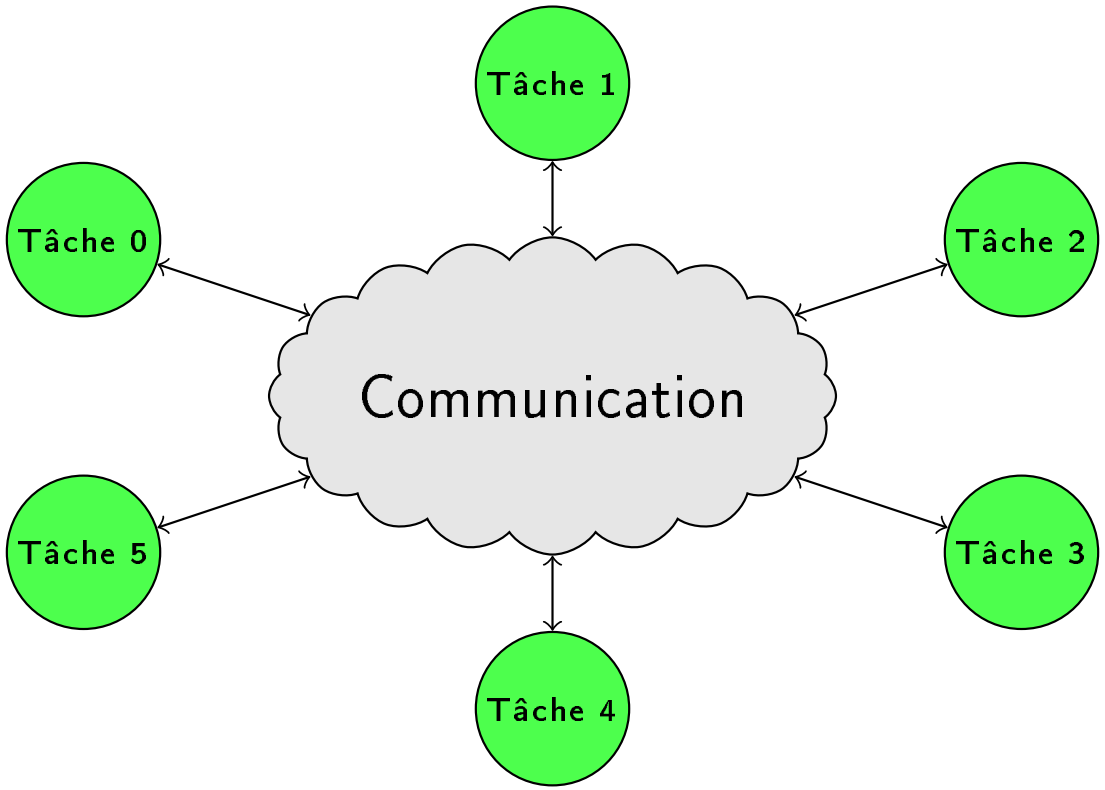
Modèle SPMD

- Toutes les tâches exécutent le même programme



Modèle SPMD

- Toutes les tâches exécutent le même programme



- Étendu au modèle MPMD (Multiple Program Multiple Data) dans les versions récentes

Le standard MPI et les librairies

- Multiples librairies proposant une implémentation du standard
- Majoritairement en C(++)/Fortran
  - OpenMPI
  - MPICH
  - MVAPICH
  - IntelMPI
  - IBM Spectrum MPI
  - ...

MPI en Java

- Java/JVM
- Moins adapté à l'écriture de programmes finement optimisés

MPI en Java

- Java/JVM
- Moins adapté à l'écriture de programmes finement optimisés
- Quelques implémentations de MPI
  - OpenMPI (non disponible pour Windows)
  - mpiJava (dernière release Janvier 2003)
  - MPJ Express (dernière release Avril 2015)

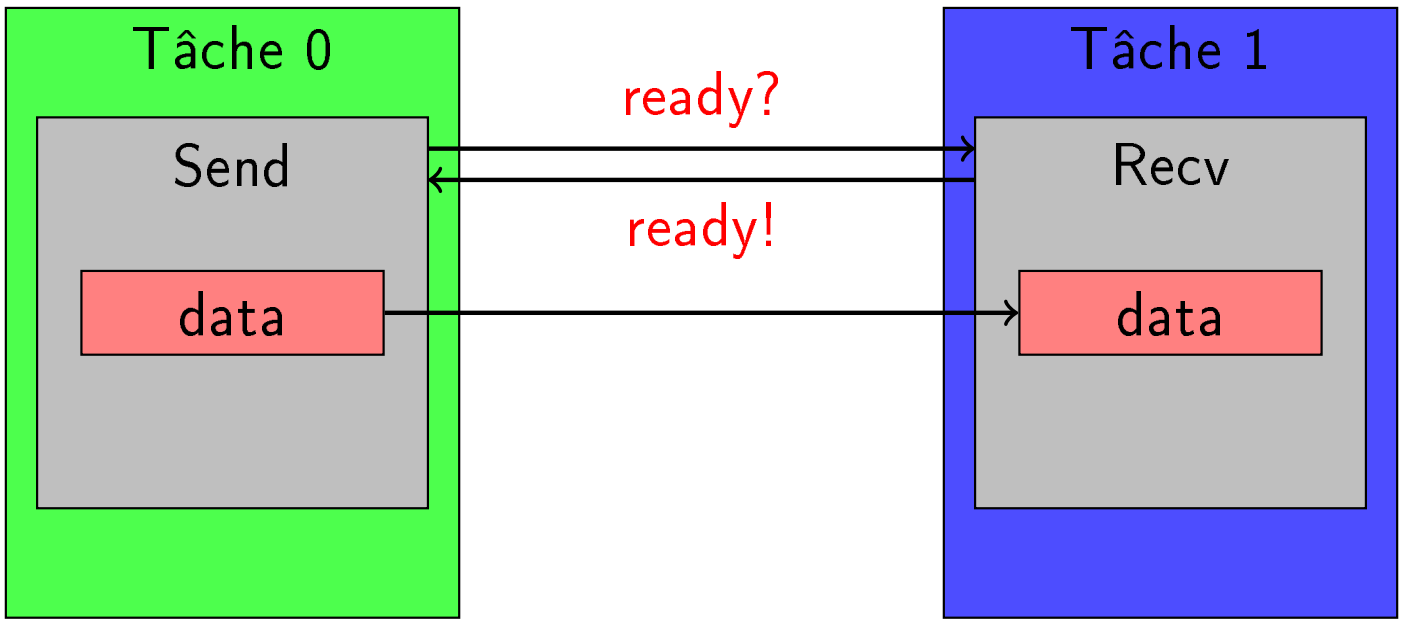
Tâches et exécution

- MPJ Express (Facilement intégrable à Eclipse, même sur Windows)
- Programme défini dans la fonction main:
  - MPI.Init(args);
  - MPI.Finalize();
- Les tâches sont numérotées par leur rang
  - rang: MPI.COMM_WORLD.Rank();
  - size: MPI.COMM_WORLD.Size();
- argument –jar ${MPJ_HOME}/lib/starter.jar
- argument –np pour définir le nombre de tâches

Exemple

```
import mpi.*;

class Main {

  static public void main(String[] args) {
    MPI.Init(args);
    int rank = MPI.COMM_WORLD.Rank();

    if(rank == 0) {
      char[] message = "Hello World".toCharArray();
      MPI.COMM_WORLD.Send(message, 0, message.length, MPI.CHAR, 1, 0);
    }
    else {
      char[] message = new char[11];
      MPI.COMM_WORLD.Recv(message, 0, 11, MPI.CHAR, 0, 0);
      System.out.println("Message received: " + String.valueOf(message)) ;
    }
    MPI.Finalize();
  }
}
```

Communication point-à-point



MPI Send et Receive

MPI.COMM_WORLD.Send(*message*, *offset*, *size*, *datatype*, *rank*, *tag*)
MPI.COMM_WORLD.Recv(*message*, *offset*, *size*, *datatype*, *rank*, *tag*)

message : Données échangées pendant la communication

offset : Décalage du message

size : Taille du message

datatype : Type de donnée du message

rank : Rang de la tâche destinataire/source

tag : Identification du message (doit correspondre entre l'envoi et la réception)

Retour sur l'exemple

```
import mpi.*;

class Main {

  static public void main(String[] args) {
    MPI.Init(args);
    int rank = MPI.COMM_WORLD.Rank();

    if(rank == 0) {
      char[] message = "Hello World".toCharArray();
      MPI.COMM_WORLD.Send(message, 0, message.length, MPI.CHAR, 1, 0);
    }
    else {
      char[] message = new char[11];
      MPI.COMM_WORLD.Recv(message, 0, 11, MPI.CHAR, 0, 0);
      System.out.println("Message received: " + String.valueOf(message)) ;
    }
    MPI.Finalize();
  }
}
```

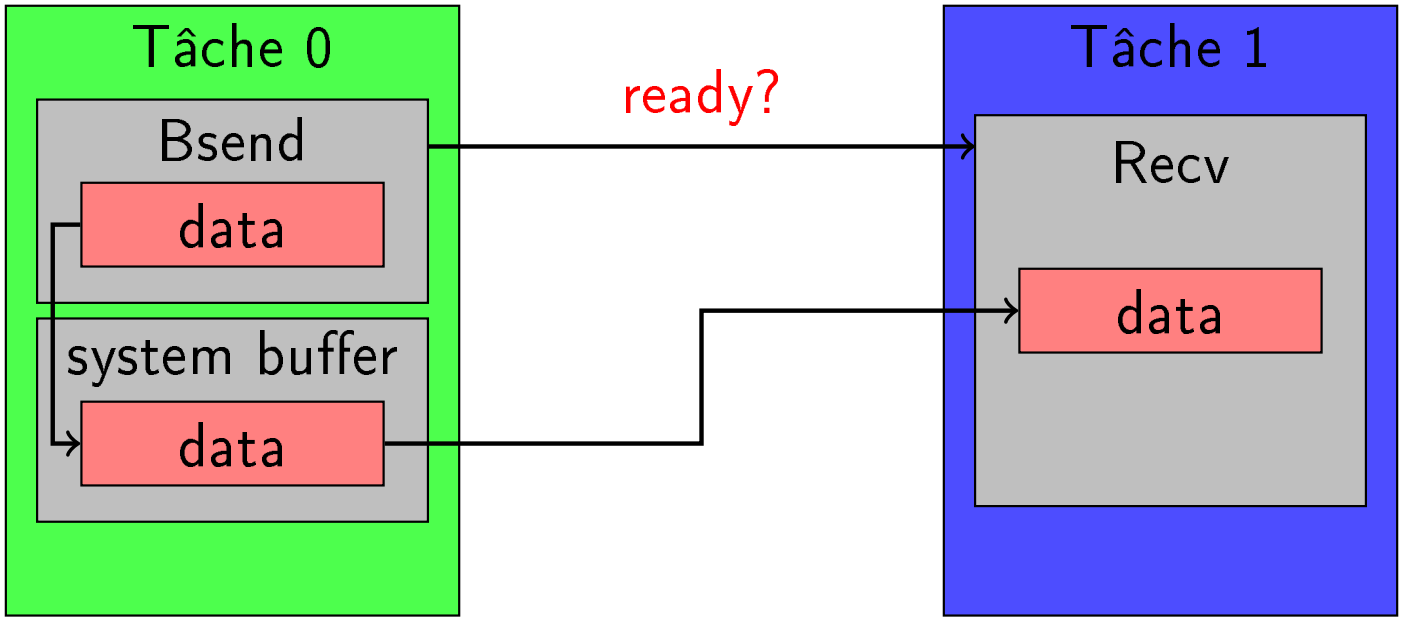
Modes de communication

Mode	Requêtes	
Standard	Send	Recv
Bufferisé	Bsend	
Synchrone	Ssend	
Ready	Rsend	

Choix du mode en fonction de l'implémentation et des coûts d'exécution:

- Synchronisation
  - Avantage : Garantie du comportement du programme
  - Coût : perte de temps à attendre la réception/l'envoi
- Gestion du buffer
- Éviter les deadlocks

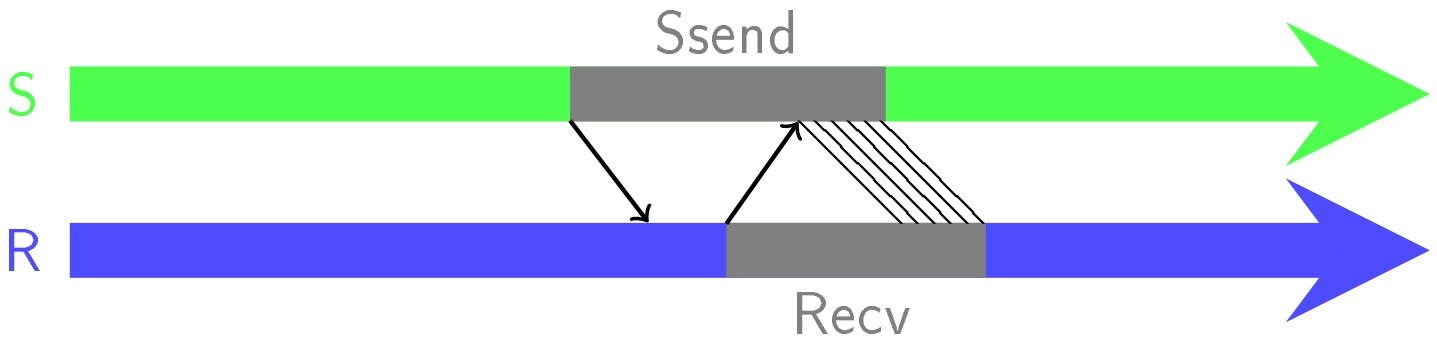
Mode bufferisé



- le message est envoyé directement si possible
- sinon, il est stocké dans un buffer
- Gestion du buffer
  - Allocation: MPI.Buffer_attach(ByteBuffer.allocate(size))
  - Coût mémoire/temps

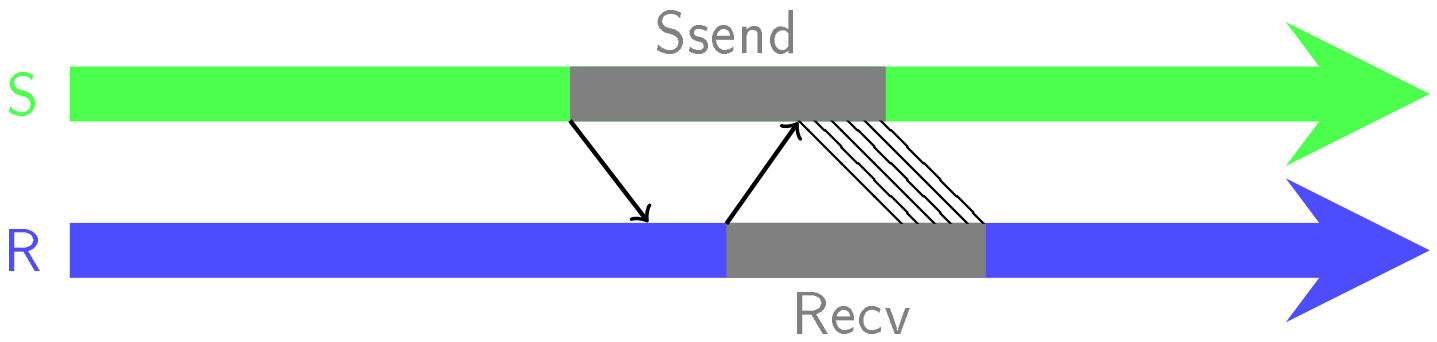
Modes synchrone et ready

- Mode synchrone

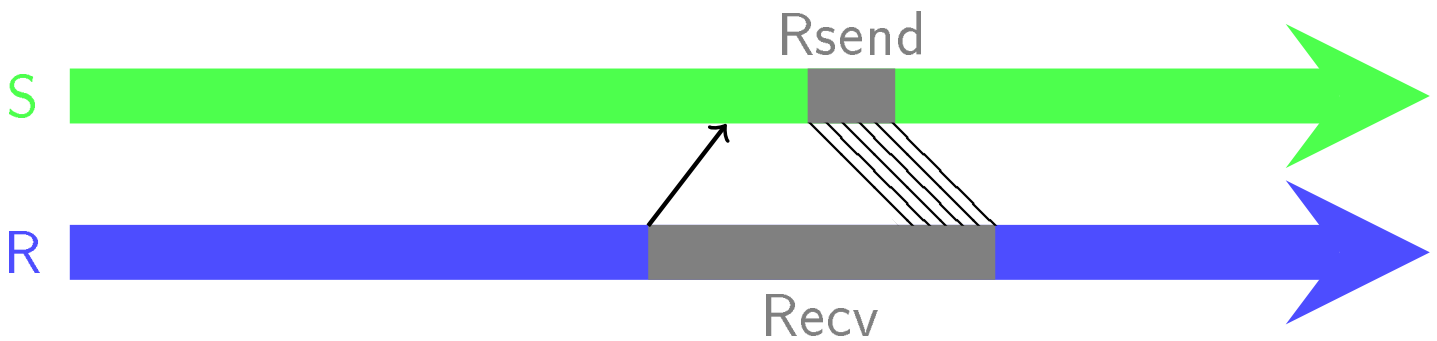


Modes synchrone et ready

- Mode synchrone



- Mode ready



Modes (résumé)

	Standard	Bufferisé	Synchrone	Ready
Def	Optimisation entre buffer et synchrone	• Envoi direct si possible • message stocké sinon	• Handshake	• Envoi direct • Erreur si receveur non-prêt
+	Optimisé	• Pas d'attente coté envoyeur • Envoyeur peut poursuivre	• Garantie de réception/synchronisation • Pas de coût de copie supplémentaire	• Aucun coût de synchronisation • Pas de coût de copie supplémentaire
−	Comportement variable	• Gestion du buffer • Coût de copie	• Contrainte forte de synchronisation • Perte de temps côté envoyeur	• Receveur doit être prêt • Peut échouer

Qu'est-ce qu'un deadlock ?

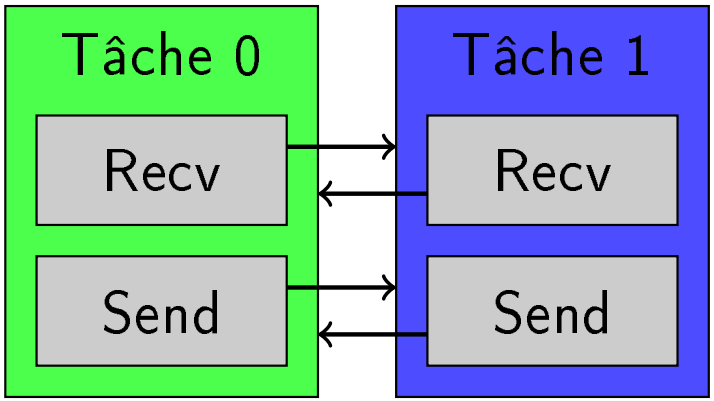
deadlock (fr: *interblocage*) phénomène se produisant en programmation parallèle quand 2 processus (ou plus) s'attendent mutuellement.

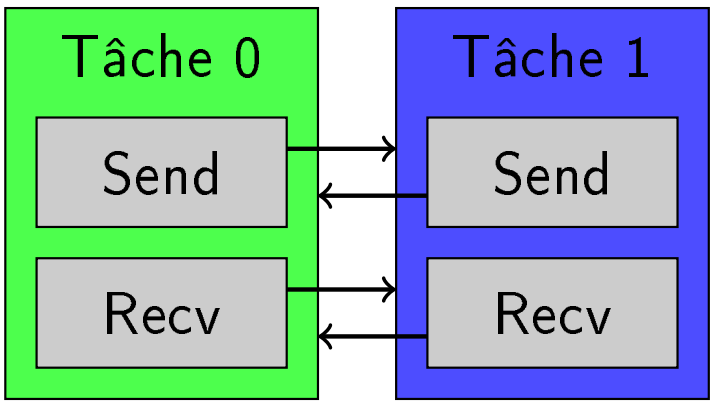
- Les processus en deadlock sont bloqués définitivement
- Situation catastrophique à éviter : le programme ne peut pas terminer.
- Plusieurs sources de deadlock en général:
  - les processus tente d'accéder à une même ressource
  - un processus détient de façon définitive une ressource nécessaire à l'avancée des autres processus
- en MPI:
  - les processus bloqués sont en attente d'un message ou du signal de réception d'un message envoyé

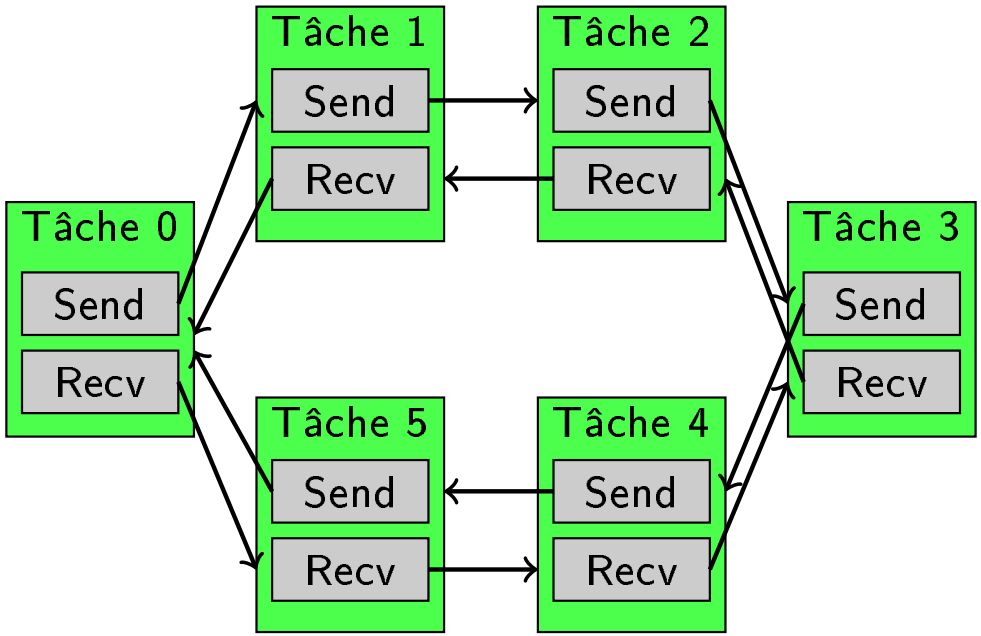
Exemple

```
import mpi.*;

class Main {

  static public void main(String[] args) {
    MPI.Init(args);
    int rank = MPI.COMM_WORLD.Rank();

    if(rank == 0) {
      char[] rec = new char[11];
      char[] message = "Hello World".toCharArray();
      MPI.COMM_WORLD.Recv(message, 0, 11, MPI.CHAR, 0, 0);
      MPI.COMM_WORLD.Send(message, 0, message.length, MPI.CHAR, 0, 0);
      System.out.println("Message received by 0: " + String.valueOf(message)) ;
    }
    else {
      char[] rec = new char[11];
      char[] message = "Hello World".toCharArray();
      MPI.COMM_WORLD.Recv(message, 0, 11, MPI.CHAR, 1, 0);
      MPI.COMM_WORLD.Send(message, 0, message.length, MPI.CHAR, 1, 0);
      System.out.println("Message received by 1: " + String.valueOf(message)) ;
    }
    MPI.Finalize();
  }
}
```

Quelques deadlocks classiques







Comment éviter des deadlocks ?

- Inverser l'ordre de séquentialité envoie/réception

Recv	Recv
Send	Send

$\Longrightarrow$

Recv	Send
Send	Recv

- Send/recv simultané: méthode MPI.COMM_WORLD.Sendrecv

Recv	Recv
Send	Send

$\Longrightarrow$

Sendrecv	Sendrecv

- Communication en mode bufferisé (déconseillé)

Send	Send
Recv	Recv

$\Longrightarrow$

Bsend	Bsend
Recv	Recv

Comment éviter des deadlocks ?

- Inverser l'ordre de séquentialité envoie/réception

Recv	Recv
Send	Send

$\Longrightarrow$

Recv	Send
Send	Recv

- Send/recv simultané: méthode MPI.COMM_WORLD.Sendrecv

Recv	Recv
Send	Send

$\Longrightarrow$

Sendrecv	Sendrecv

- Communication en mode bufferisé (déconseillé)

Send	Send
Recv	Recv

$\Longrightarrow$

Bsend	Bsend
Recv	Recv

- Communication non-bloquante

Exemple

```
import mpi.*;

class Main {

  static public void main(String[] args) {
    MPI.Init(args);
    int rank = MPI.COMM_WORLD.Rank();

    if(rank == 0) {
      char[] rec = new char[11];
      char[] message = "Hello World".toCharArray();
      MPI.COMM_WORLD.Recv(message, 0, 11, MPI.CHAR, 0, 0);
      MPI.COMM_WORLD.Send(message, 0, message.length, MPI.CHAR, 0, 0);
      System.out.println("Message received by 0: " + String.valueOf(message)) ;
    }
    else {
      char[] rec = new char[11];
      char[] message = "Hello World".toCharArray();
      MPI.COMM_WORLD.Send(message, 0, message.length, MPI.CHAR, 1, 0);
      MPI.COMM_WORLD.Recv(message, 0, 11, MPI.CHAR, 1, 0);
      System.out.println("Message received by 1: " + String.valueOf(message)) ;
    }
    MPI.Finalize();
  }
}
```

Communication non-bloquante

Mode	Requêtes	
Standard	Isend	Irecv
Bufferisé	Ibsend	
Synchrone	Issend	
Ready	Irsend	

- Amorce la communication sans bloquer le programme
- Le système est notifié du message mais l'échange de donnée peut avoir lieu plus tard
- Permet une plus grand flexibilité dans la séquentialité des appels MPI
- Évite les deadlocks

Contrôle

Request r = MPI.COMM_WORLD.Isend(message, offset, size, datatype, rank, tag)

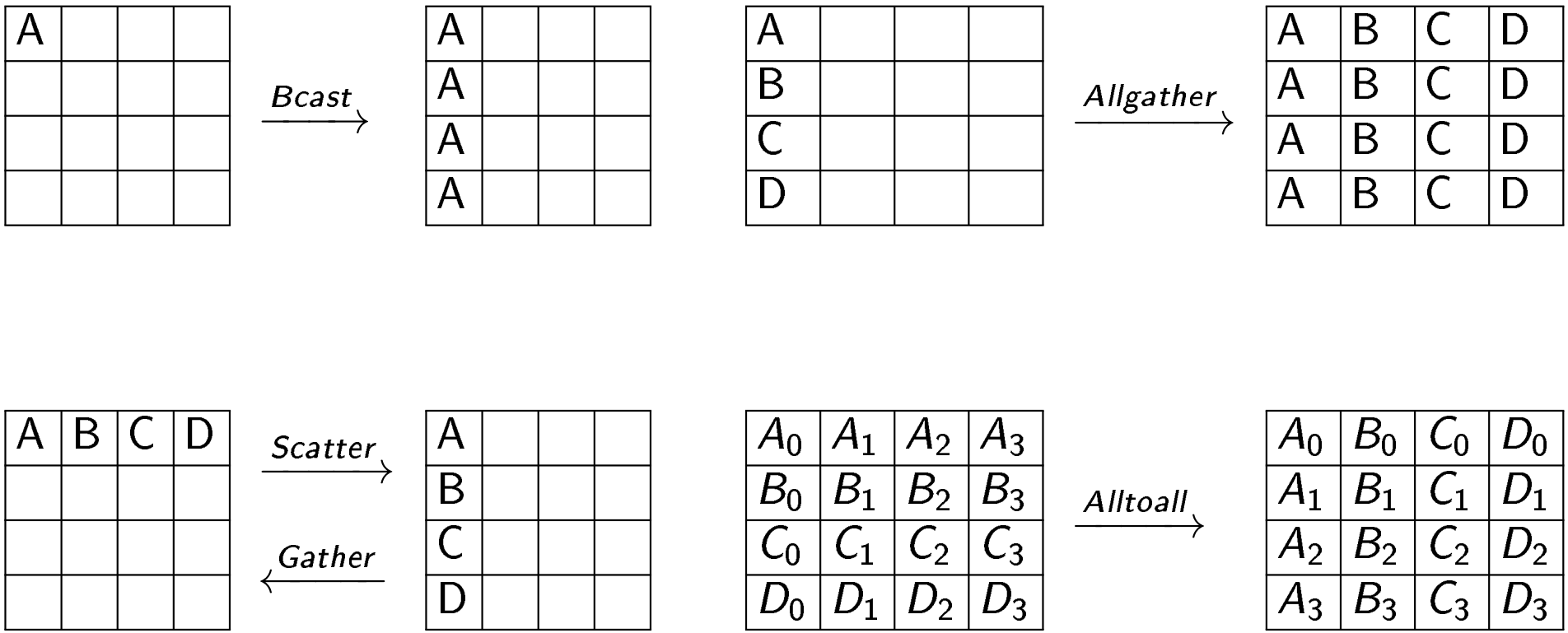
Request r = MPI.COMM_WORLD.Irecv(message, offset, size, datatype, rank, tag)

- L'utilisateur doit s'assurer que la requête a terminé avant de modifier/accéder les données;
- Permet de tester si la requête a réussi ou échoué.
- Attendre que la requête termine : r.Wait()
- Vérifier que la requête a terminé : r.Test()
- Annuler la requête : r.Cancel()

Communication collective (1/2)

- Synchronisation :
  - Barrière : MPI.COMM_WORLD.Barrier
- Propagation de données :
  - Broadcast (distribue) : MPI.COMM_WORLD.Bcast
  - Gather (récolte) : MPI.COMM_WORLD.Gather
  - Scatter (répartie) : MPI.COMM_WORLD.Scatter
  - Allgather (gather+broadcast) : MPI.COMM_WORLD.Allgather
  - Alltoall (recroisement) : MPI.COMM_WORLD.Alltoall
- Calcul collectif :
  - Reduce : MPI.COMM_WORLD.Reduce
  - Scan : MPI.COMM_WORLD.Scan

Communication collective (2/2)



Références & ressources supplémentaires

- https://cvw.cac.cornell.edu/mpip2p/default
- https://www.mpi-forum.org/docs/mpi-1.1/mpi-11-html/mpi-report.html
- https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf