Calcul Haute-Performance

High-Performance Computing (HPC)

Université Paris-Sud

Marc Baboulin (baboulin@lri.fr)

Contenu

- **I - Introduction**
- **II - Programmation des architectures à mémoire distribuée**
- **III - Présentation de MPI**
- **IV - Analyse de performance**
- **V - Programmation des architectures à mémoire partagée**
- **VI - Les bibliothèques numériques parallèles**

I – Introduction

Le parallélisme en quelques mots

Définitions:

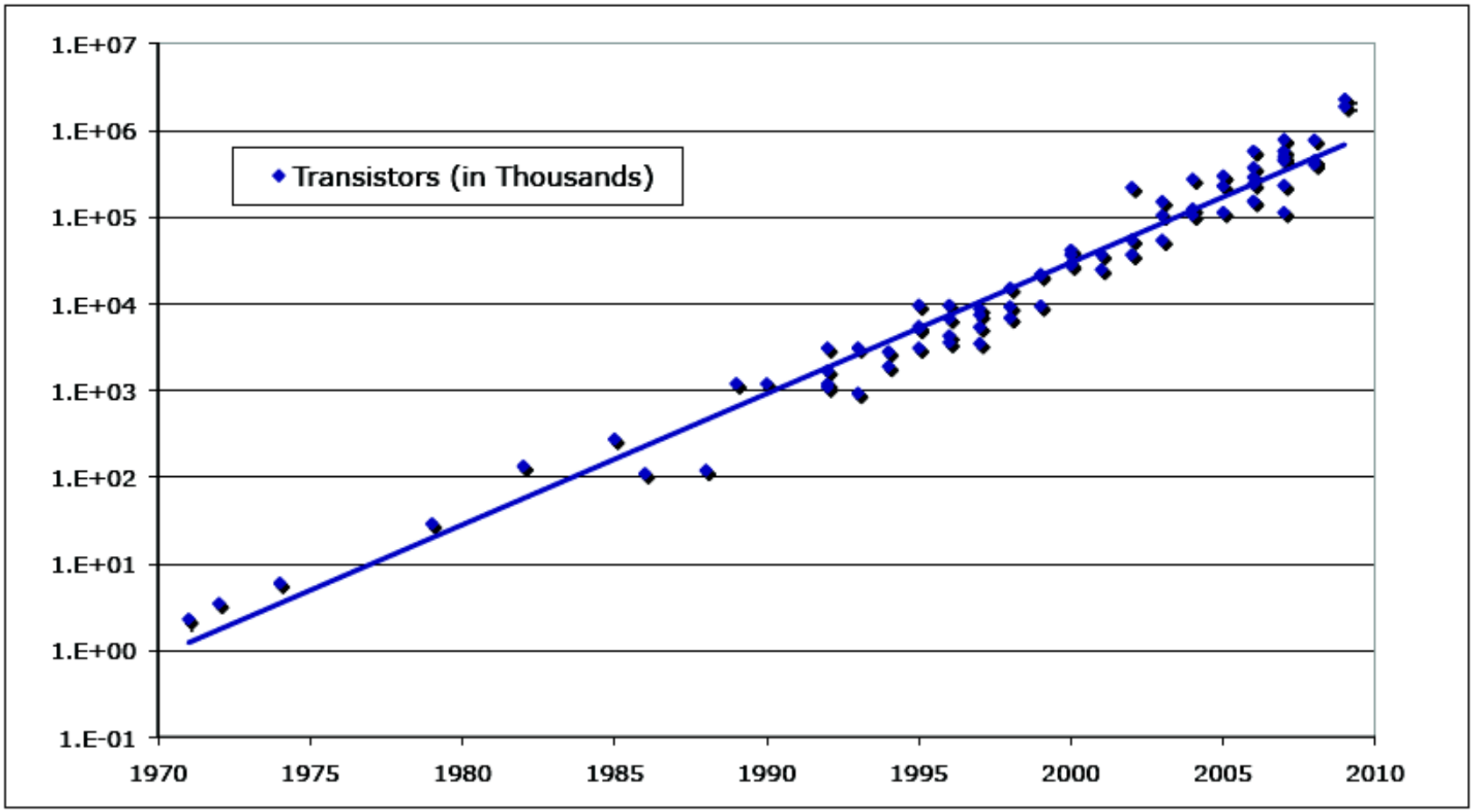
- Division d'un algorithme en tâches exécutables simultanément
- Exécution d'un algorithme en utilisant plusieurs processeurs
- Objectif: réduire le temps de résolution d'un problème et/ou traiter plus de données

Problématique:

- Architectures matérielles
- Modèle de programmation
- Notion de “parallélisabilité”

Loi de Moore: la course aux GHz

- Conjecture: le nombre de transistors double tous les 2 ans



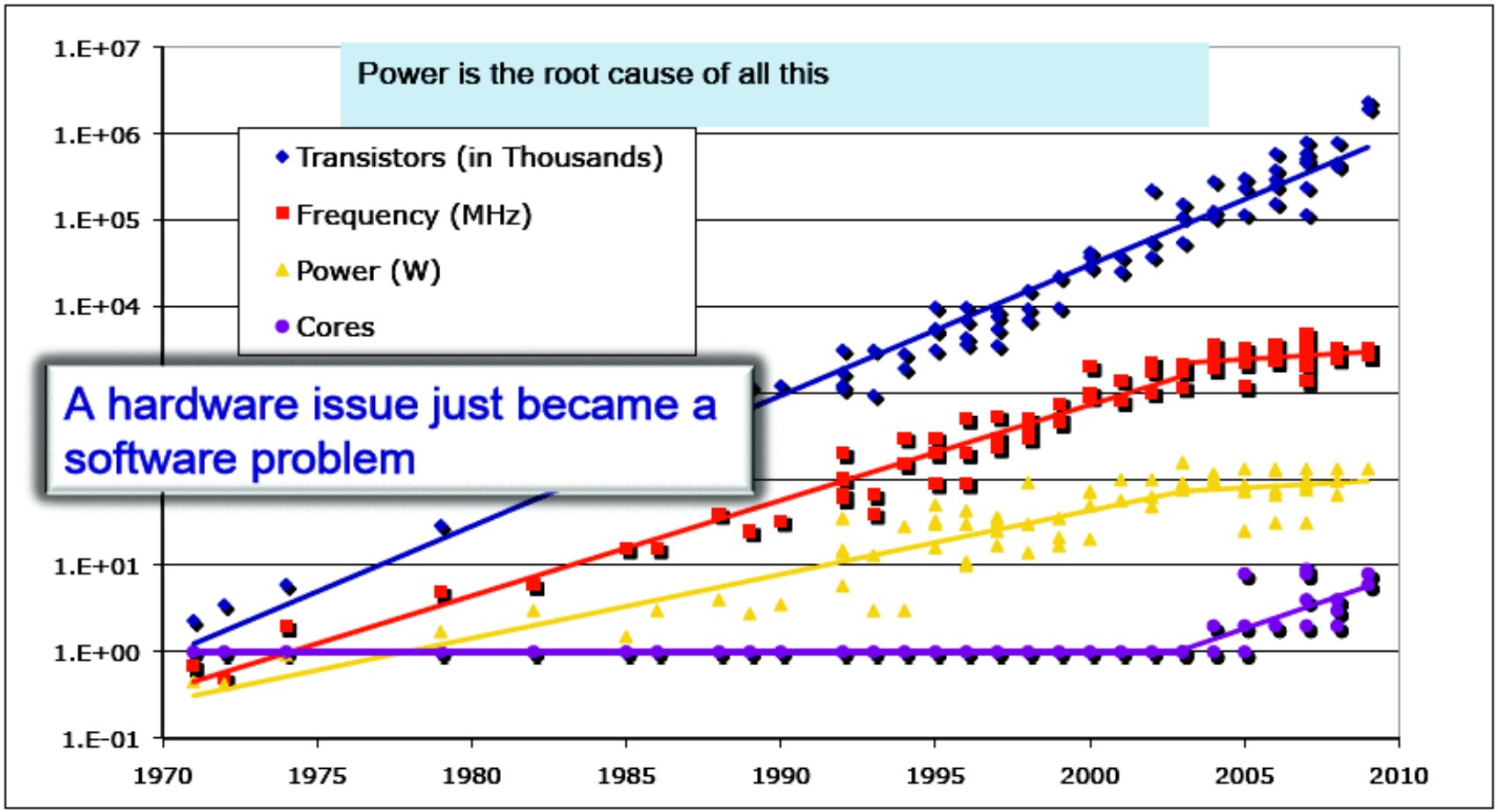
Data from Kunle Olukotun, Lance Hammond, Herb Sutter, Burton Smith, Chris Batten, and Krste Asanoviç
Slide from Kathy Yelick

Loi de Moore et containtes physiques

- Jusqu'en 2004, les gains de performances étaient obtenus par:
  - Augmentation des fréquences d'horloge
  - Amélioration du parallélisme via les jeux d'instructions
- Depuis 2004, difficultés dûes à la dissipation thermique et à la consommation énergétique:
  - Fin de la course à la fréquence d'horloge (énergie augmente avec MHz^3)
  - Augmenter le nombre de processeurs sur une puce: architectures multi-coeurs (énergie croît linéairement avec le nombre de transistors)
  - Parallélisme au niveau des threads
- Impact sur le logiciel qui doit être ré-écrit pour tirer parti de ces nouvelles architectures

Tendances hardware

- Ralentissement des gains en performance et en énergie consommée

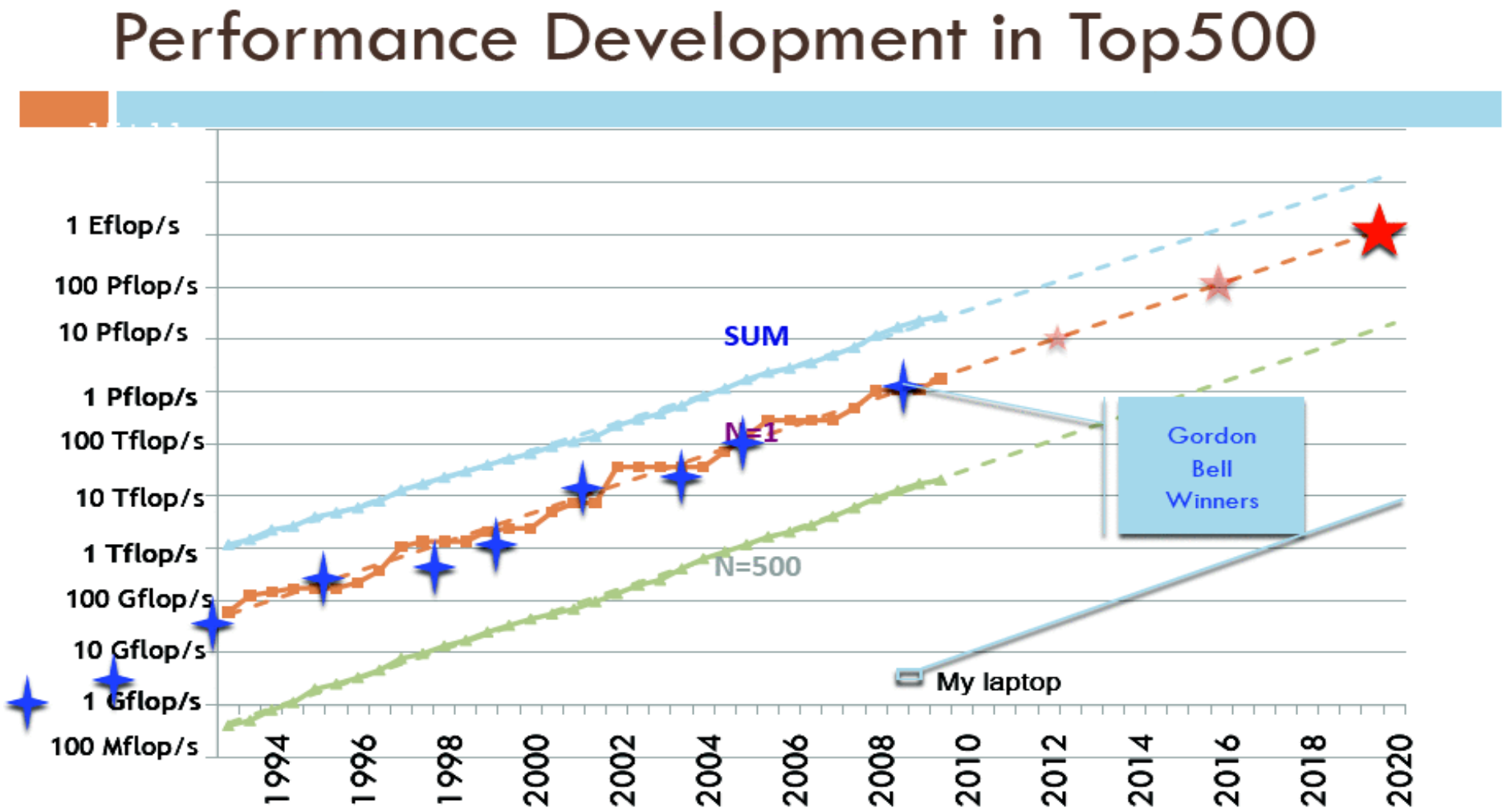


Data from Kunle Olukotun, Lance Hammond, Herb Sutter, Burton Smith, Chris Batten, and Krste Asanoviç
Slide from Kathy Yelick

TOP500: état de l'art des machines parallèles

- Liste des 500 calculateurs les plus puissants dans le monde
  http://www.top500.org
  publiée 2 fois par an (novembre et juin) depuis 1993
- Statistiques sur la performance, la localisation, les applications...
- Objectifs: photographie à un instant donné des possibilités des calculateurs parallèles, favoriser les collaborations au sein de la communauté HPC
- Méthode de classement: LINPACK Benchmark (résolution d'un système linéaire dense) “tuné” pour la machine testée.
  Algorithme utilisé: LU avec pivotage partiel $\simeq 2\,n^3/3$ flops
- Données fournies par le TOP500:
  - Rpeak (Gflop/s): performance pic théorique
  - Rmax (Gflop/s): performance pour système de taille Nmax
  - Nhalf: taille de problème pour lequel on obtient Rmax/2
- Benckmark récent: HPCG (sparse iterative solver)

Evolution des performances machines



Source: Jack Dongarra, UTK

Résumé

- Liste et tendances sur https://www.top500.org/list/2019/06/
- Janv. 2009: 1,1 Pflop/s – 2,5 MW
- Juin 2019: 149 Pflop/s – 10 MW
  x135 x4
- Objectif: 1Eflop/s vers 2020-2022 à 20 MW

Principaux enjeux:

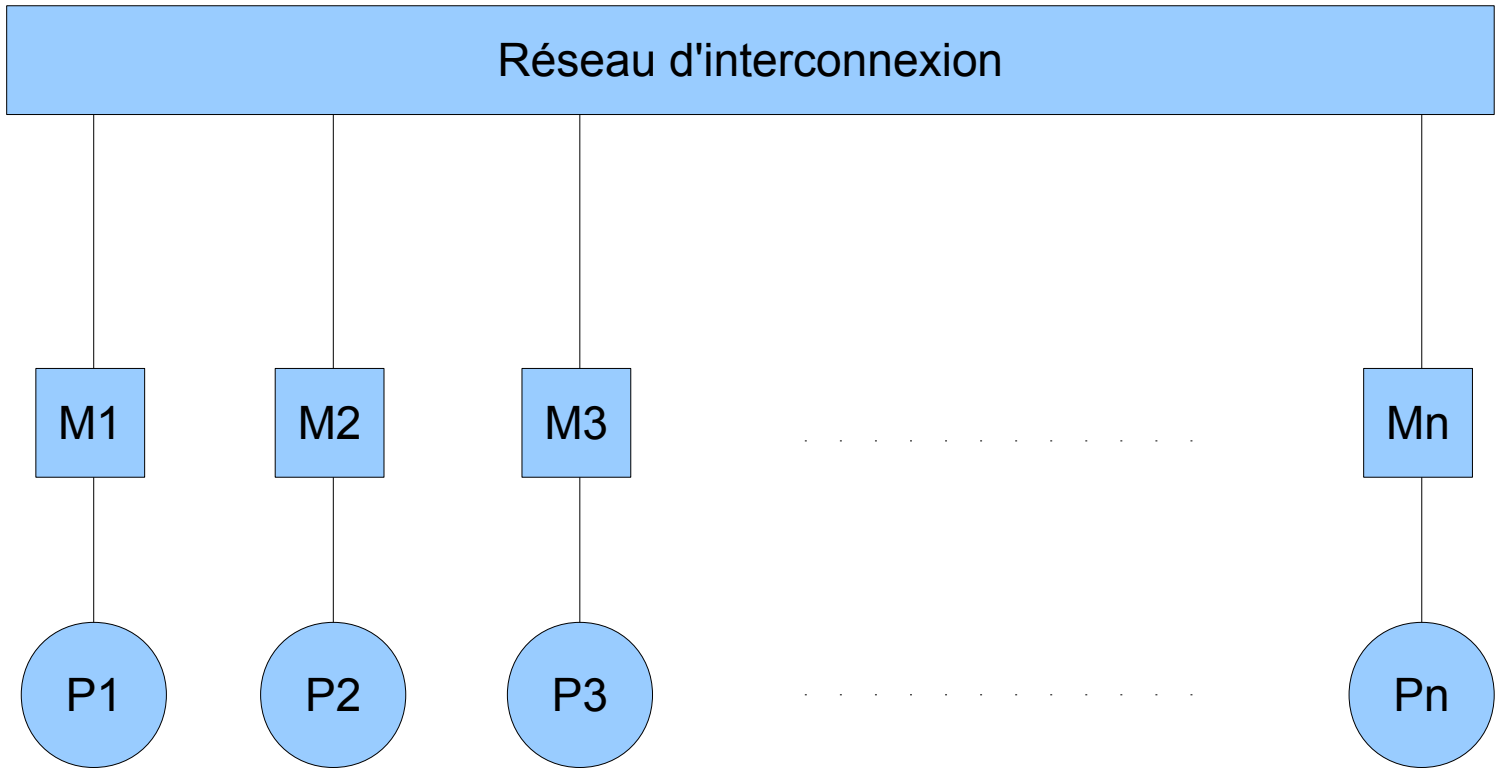
- Energie
- Communications (flops importent moins)
- Hétérogénéité (accélérateurs GPUs..)
- Tolérance aux pannes (Sequoia BG/Q: 1,25/noeud/jour)

II - Programmation parallèle des architectures à mémoire distribuée

Architectures à mémoire distribuée

- **Contexte hardware:** multiprocesseur à mémoire distribuée ou réseau de stations de travail (ou les deux, reliés par un réseau)
- **Contexte mémoire:** espace d'adressage disjoint où chaque processeur a son propre espace et la communication s'effectue à travers des copies explicites (transfert de messages)
- **Objectif:** répartir/gérer des calculs sur la machine cible
- **Outils nécessaires:**
  - sécurité et droits d'accès
  - création de processus distants
  - communication entre processus
  - synchronisation entre processus
  - cohérence des données et traitements
  - séquencement des tâches réparties
  - tolérance aux pannes, points de reprise

Architectures à mémoire distribuée



Modèle par transfert de messages

- Modèle le plus répandu en calcul réparti: permet de gérer la communication et la synchronisation entre processus
- Le parallélisme et la distribution des données sont à la charge du programmeur: chaque communication ou synchronisation nécessite l'appel à une routine
  - Communications: point-à-point, collectives
  - Synchronisations: barrières, pas de verrou (car pas de variable partagée à protéger)
  - Demandes (exemple: combien de processeurs? qui suis-je? y-a-t-il des messages en attente?)
- Echange de données explicite (pas de variable partagée)
- Prise en charge possible des réseaux hétérogènes avec gestion des pannes
- Différents niveaux (canal, processus, mémoire partagée virtuelle)

Modèles par transfert de messages: questions

- Comment décrire la donnée à transmettre ?
- Comment identifier les processus ?
- Comment le destinataire va reconnaitre les messages ?
- Quand peut-on dire que l'opération est terminée ?
- Y-a-t-il synchronisation entre l'envoi et la réception ?
- Quand peut-on réutiliser la donnée envoyée ?
- Peut-on bufferiser les communications ?

Bibliothèque pour la programmation d'applications parallèles distribuées

MPI : *Message Passing Interface* – MPI 3.1 (juin 2015)

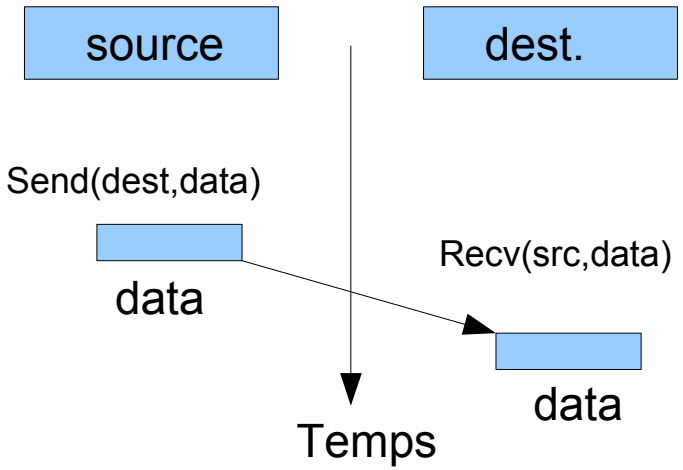
- *http://www.mcs.anl.gov/mpi/*

- standard pour le transfert de messages sur les machines multiprocesseurs. C'est une norme, pas un logiciel
- différentes implémentations : MPICH, CHIMP, LAM, constructeurs...gratuit ou payant

- Fonctions utilisables depuis C, Fortran, C++

Communications entre processus

- Point-à-point (one-to-one), entre 2 processus
- Collectives, impliquent un groupe de processus
  - one-to many (ex. broadcast)
  - many-to-one (ex. collect)
  - many-to many, entre plusieurs processus
- Envoi/réception



Envoi/réception de messages

- Environnement d'exécution des communications:
  Chaque processus est identifié par un numéro (rang dans un groupe ou communicateur)
- Le message: adresse, type, longueur
- L'enveloppe du message permet, en plus de la donnée, de distinguer les messages et de les recevoir sélectivement:
  - Source (numéro de l'émetteur), implicite pour un envoi
  - Destination (numéro du récepteur), implicite pour une réception
  - Label du message (tag), pour identifier, filtrer...
  - Contexte de communication (communicateur)

Types de communications

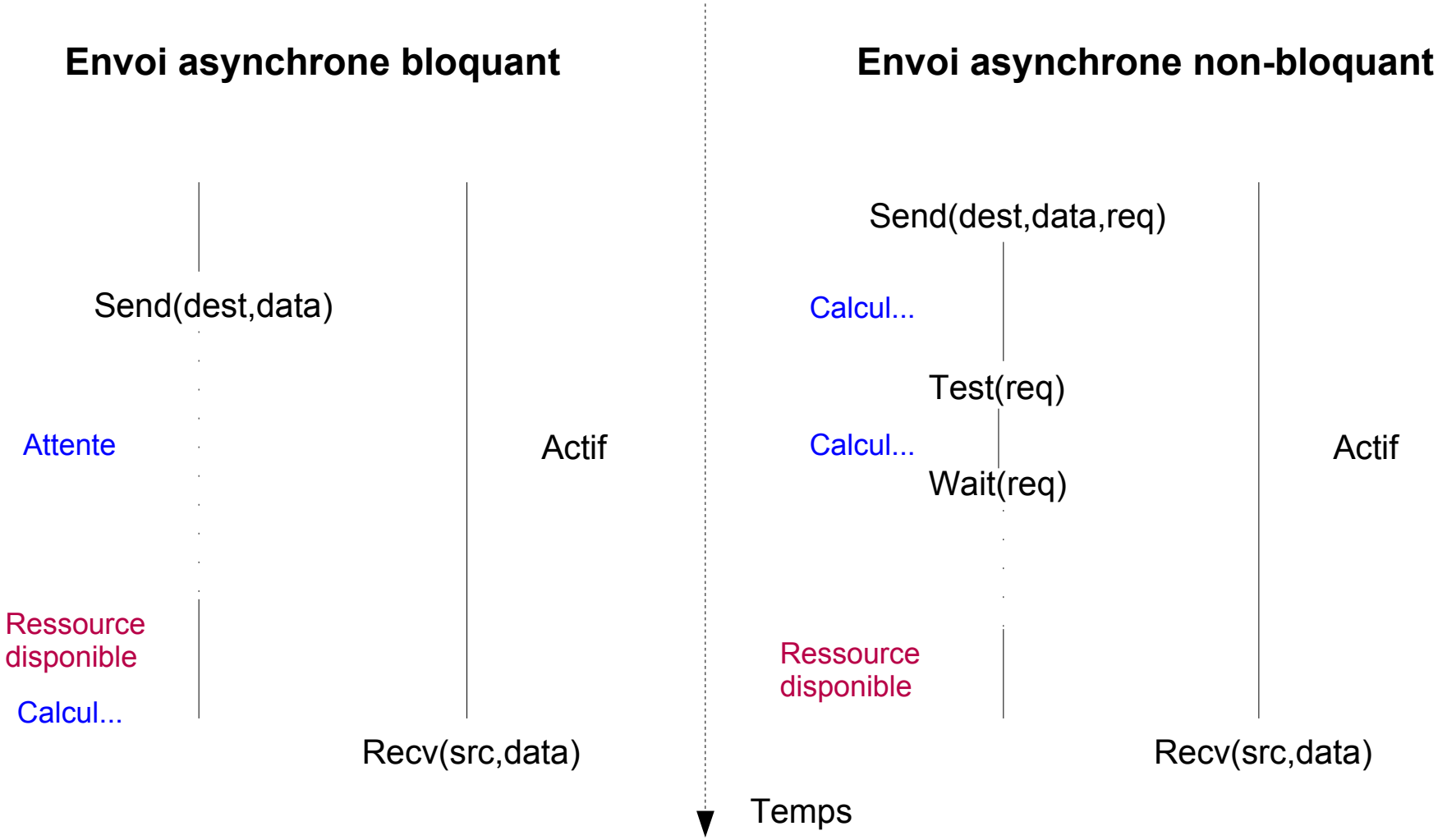
- Synchrone: le premier arrivé attend l'autre (rendez-vous)
- Asynchrone: l'émetteur et le récepteur ne s'attendent pas
- Un envoi asynchrone peut cependant être bloqué par la non consommation du message par le récepteur
- L'émetteur et le récepteur n'ont pas à être tous les deux synchrones/asynchrones
- Locale: si la bonne fin de la procédure dépend uniquement de processus executé localement
- Non-locale: si l'opération nécessite l'exécution d'une procédure par un autre processus (peut nécessiter une communication avec un autre processus)
- Collective: si tous les processus d'un groupe doivent appeler la procédure

Envoi/réception bloquant/non bloquant

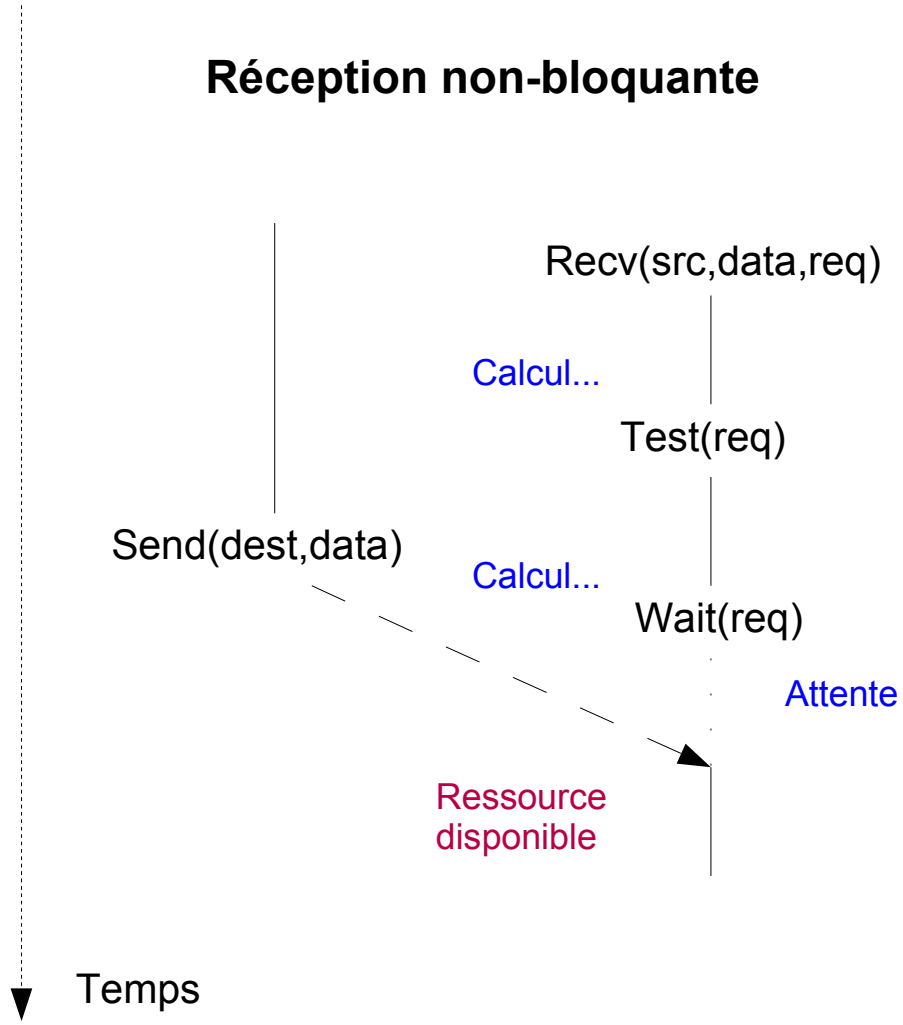
- **Envoi/Réception bloquant**: la ressource (message, enveloppe) est disponible en retour de la procédure.
- **Envoi/Réception non-bloquant**: On a un retour de la procédure sans que l'opération de transfert soit achevée et que l'utilisateur soit autorisé à utiliser la ressource.

  L'utilisateur ne peut pas réutiliser l'espace mémoire associé (au risque de changer ce qui sera envoyé)
  -Il faut tester ou attendre la libération (si envoi) ou la réception effective de la donnée grâce à un nº de requête
  `Send/Recv( dest/src,data,req )`
  `Test( req )` **et** `Wait( req )`

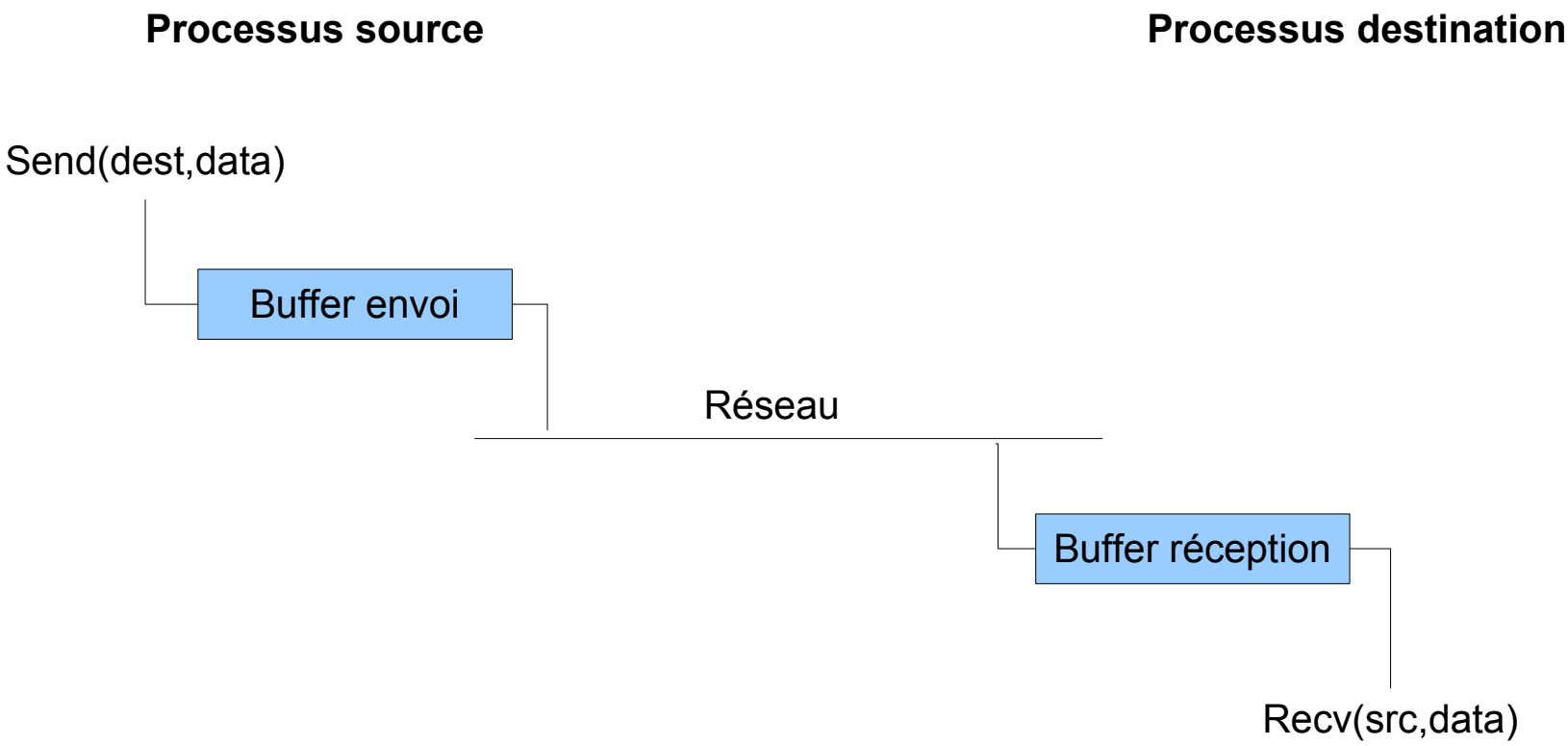
Exemples de communications
(Envois asynchrones)



Exemples de communications
(Réception non bloquante)



Transfert de l'information

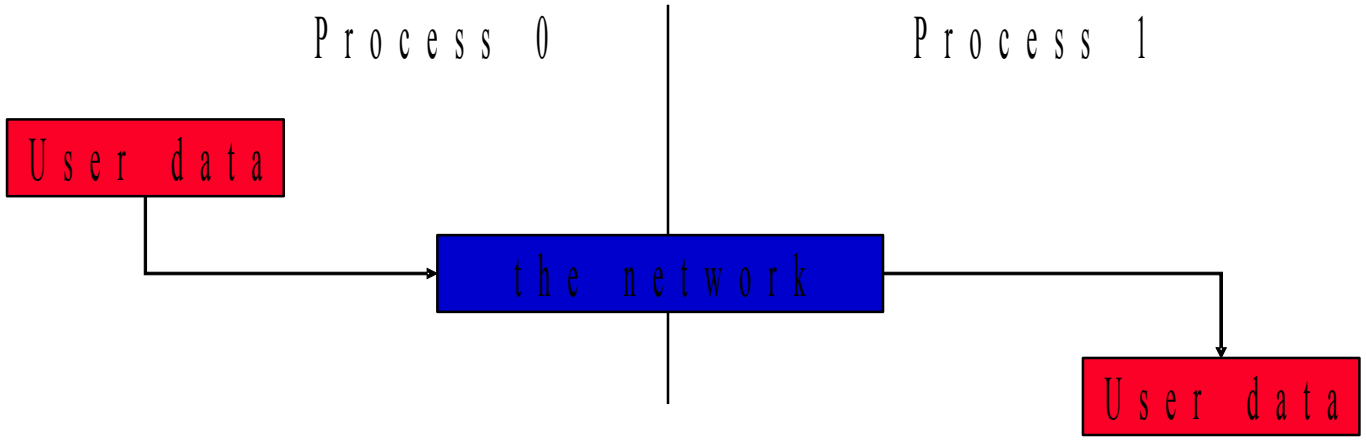


Communications “bufferisées”

- La bufferisation permet de découpler les opérations d'envoi et de réception
- Les buffers sont soit internes à la couche système, soit gérés par l'utilisateur (MPI propose plusieurs modes)
- Coût mémoire et temps lié aux copies multiples
- La bonne fin d'un transfert de message bufferisé dépend de la taille du message et du buffer disponible
- Permet en mode bloquant de libérer l'envoyeur rapidement (à condition que la taille du buffer soit suffisante)
- Attention à gérer les bloquages liés à la saturation des buffers, notamment en cas d'envoi asynchrone non-bloquant

Communications non bufferisées

- Eviter les copies permet d'utiliser moins de mémoire
- ...mais peut nécessiter plus ou moins de temps (car il faut attendre jusqu'à la réception ou accepter de continuer même si le transfert n'est pas terminé)

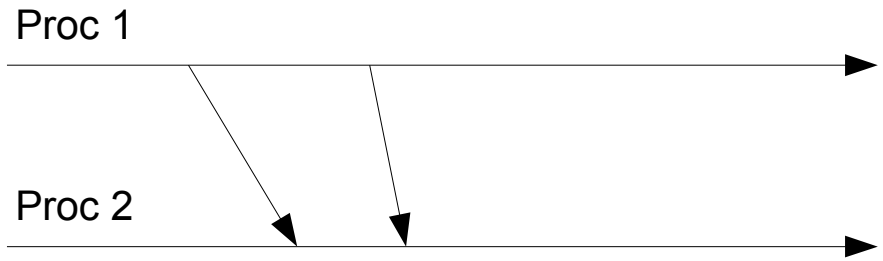


Exemples de communications point à point

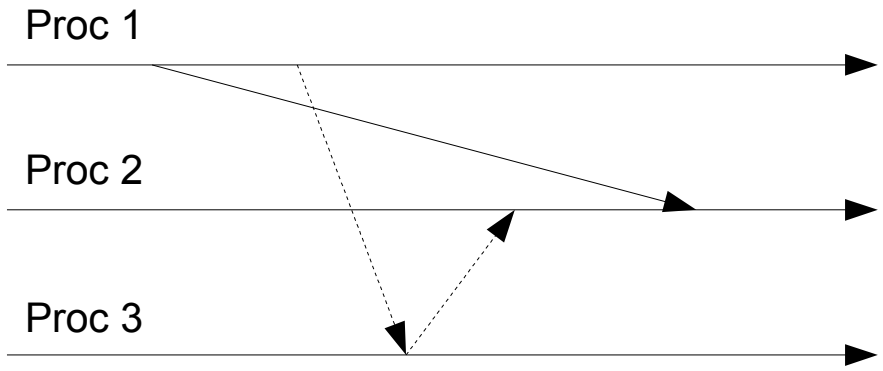
- Envoi/réception standard
  `MPI_SEND/MPI_RECV`: bloquant
  `MPI_ISEND/MPI_IRECV`: non-bloquant
- Envoi synchrone
  `MPI_SSEND`: bloquant
  `MPI_ISSEND`: non-bloquant
- Envoi bufferisé
  `MPI_BSEND`: bloquant
  `MPI_IBSEND`: non-bloquant
- Autres
  `MPI_SENDRECV`: émission et réception bloquantes
  `MPI_SENDRECV_REPLACE`: idem mais avec même buffer

Ordonnancement des communications

- Diffusion entre 2 processus (se fait dans l'ordre)



- Pas d'ordonnancement causal



Sources of Deadlocks

- Send a large message from process 0 to process 1
  - If there is insufficient storage at the destination, the send must wait for the user to provide the memory space (through a receive)
- What happens with this code?

Process 0	Process 1
Send(1)	**Send(0)**
Recv(1)	**Recv(0)**

- This is called "unsafe" because it depends on the availability of system buffers in which to store the data sent until it can be received

Some Solutions to the "unsafe" Problem

- Order the operations more carefully:

Process 0	Process 1
Send(1)	**Recv(0)**
Recv(1)	**Send(0)**

- Supply receive buffer at same time as send:

Process 0	Process 1
Sendrecv(1)	**Sendrecv(0)**

More Solutions to the "unsafe" Problem

- Supply own space as buffer for send

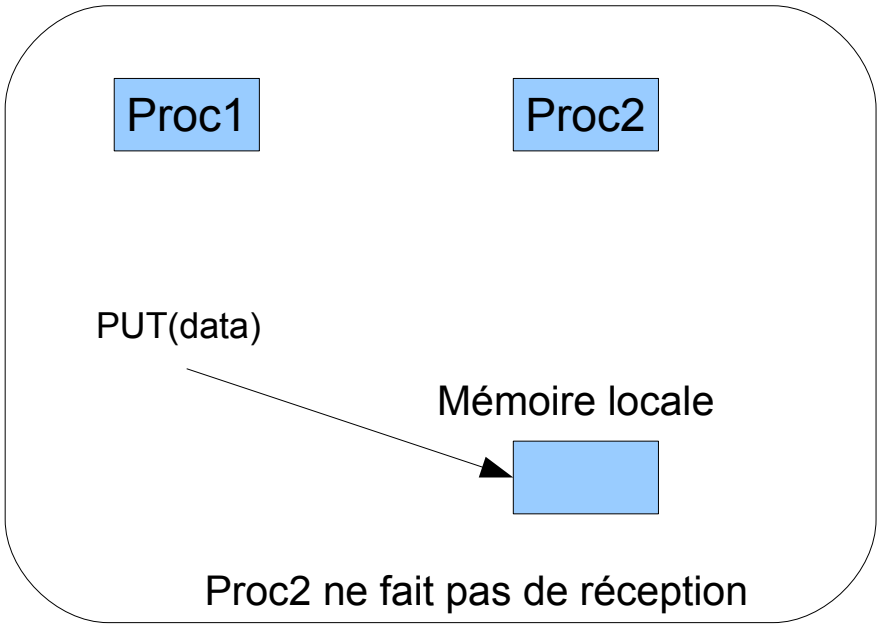
Process 0	Process 1
Bsend(1)	Bsend(0)
Recv(1)	Recv(0)

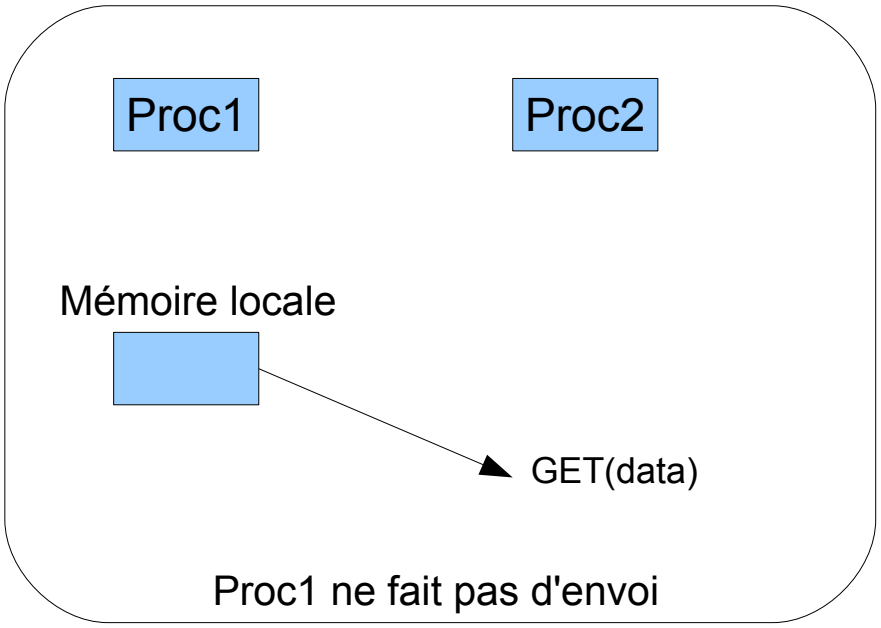
- Use non-blocking operations:

Process 0	Process 1
Isend(1)	Isend(0)
Irecv(1)	Irecv(0)
Waitall	Waitall

Communications non symétriques

- `PUT`: écriture directe dans la mémoire d'un autre processeur
- `GET`: lecture directe dans la mémoire d'un autre processeur
- Seulement dans MPI-2

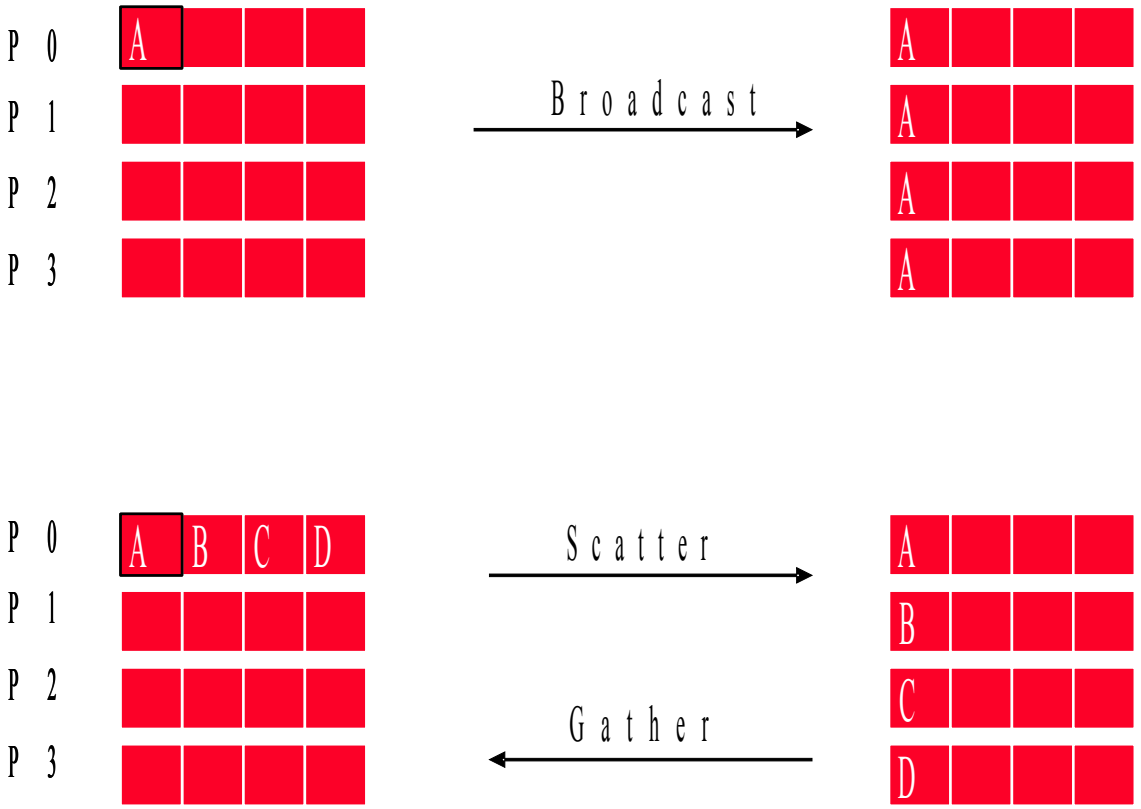




Communications collectives

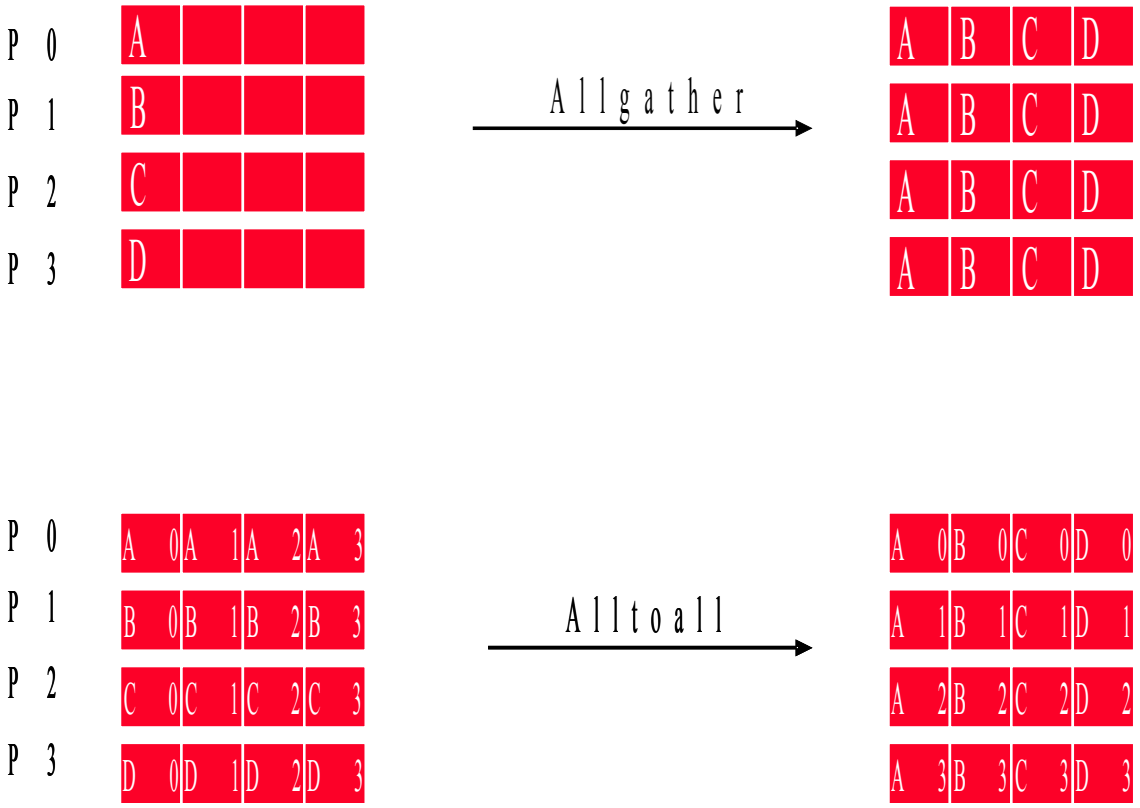
- A l'intérieur d'un groupe ou d'un communicateur
- 3 types d'opérations: synchronisation, mouvements de données, calcul collectif
- Barrière (Barrier): synchronisation entre les processus
- Diffusion (Broadcast) d'un processus à tous les membres
- Réduction (Reduce) par un processus après collecte de valeurs détenues par tous les processus
- Rassemblement (Gather) sur un processus par mise bout à bout de messages provenant de tous les processus
- Distribution (Scatter) sur tous les processus par ventilation d'un message provenant d'un processus (inverse du “Gather”)
- Rassemblement généralisé (AllGather): variation du “Gather” où le résultat est envoyé à tous les processus

Communications collectives



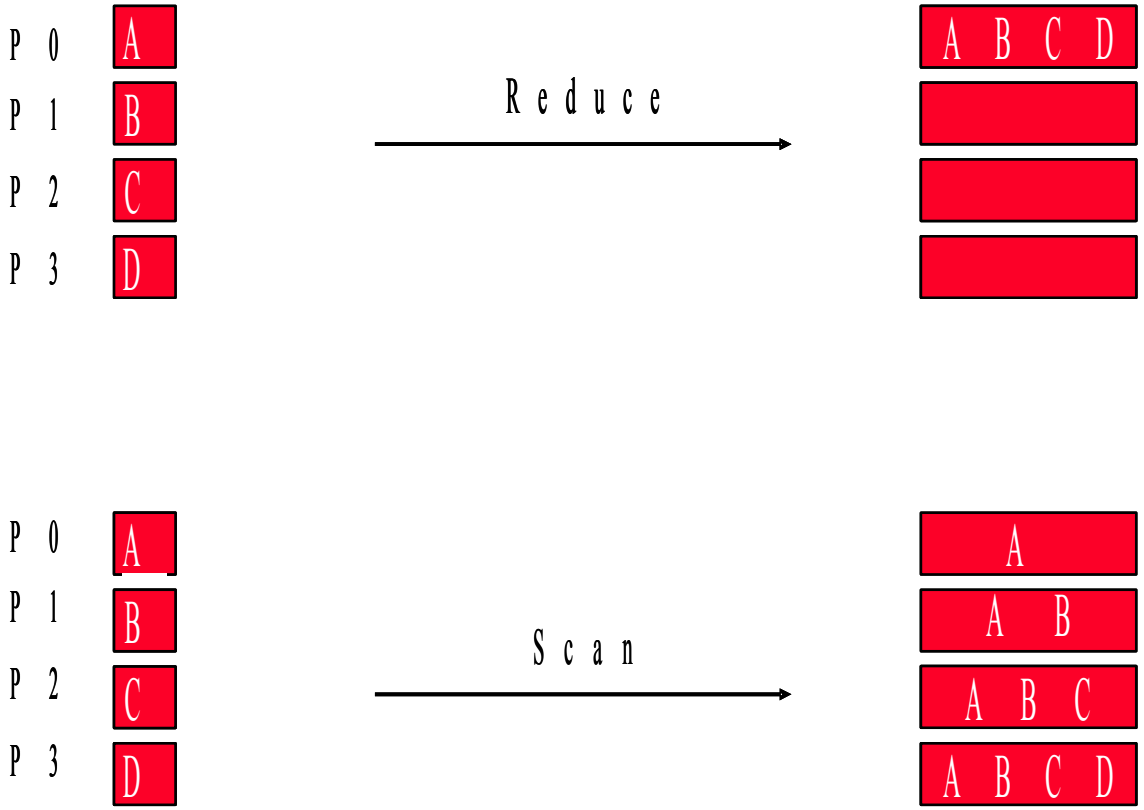
Source: Jim Demmel, UC Berkeley

Communications collectives



Source: Jim Demmel, UC Berkeley

Communications/calculs collectifs



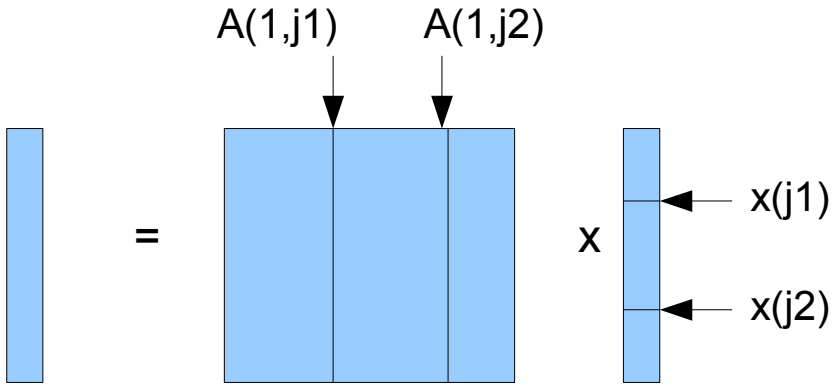
Source: Jim Demmel, UC Berkeley

Communications collectives (suite)

- Les appels collectifs sont bloquants
- Une opération Reduce permet d'exécuter une opération sur des données distribuées sur tous les membres d'un groupe (somme, max, “et” logique...)
- Le résultat d'une réduction peut être disponible sur tous les processus en utilisant AllReduce
- Exemple 1:
  `Reduce(sum,valeur_sum,valeur_loc,groupe,dest)`
  `valeur_sum` n'est disponible que sur le processus `dest`
- Exemple 2:
  `AllReduce(max,valeur_max,valeur_loc,groupe)`
  `valeur_max` est disponible sur tous les processus du groupe

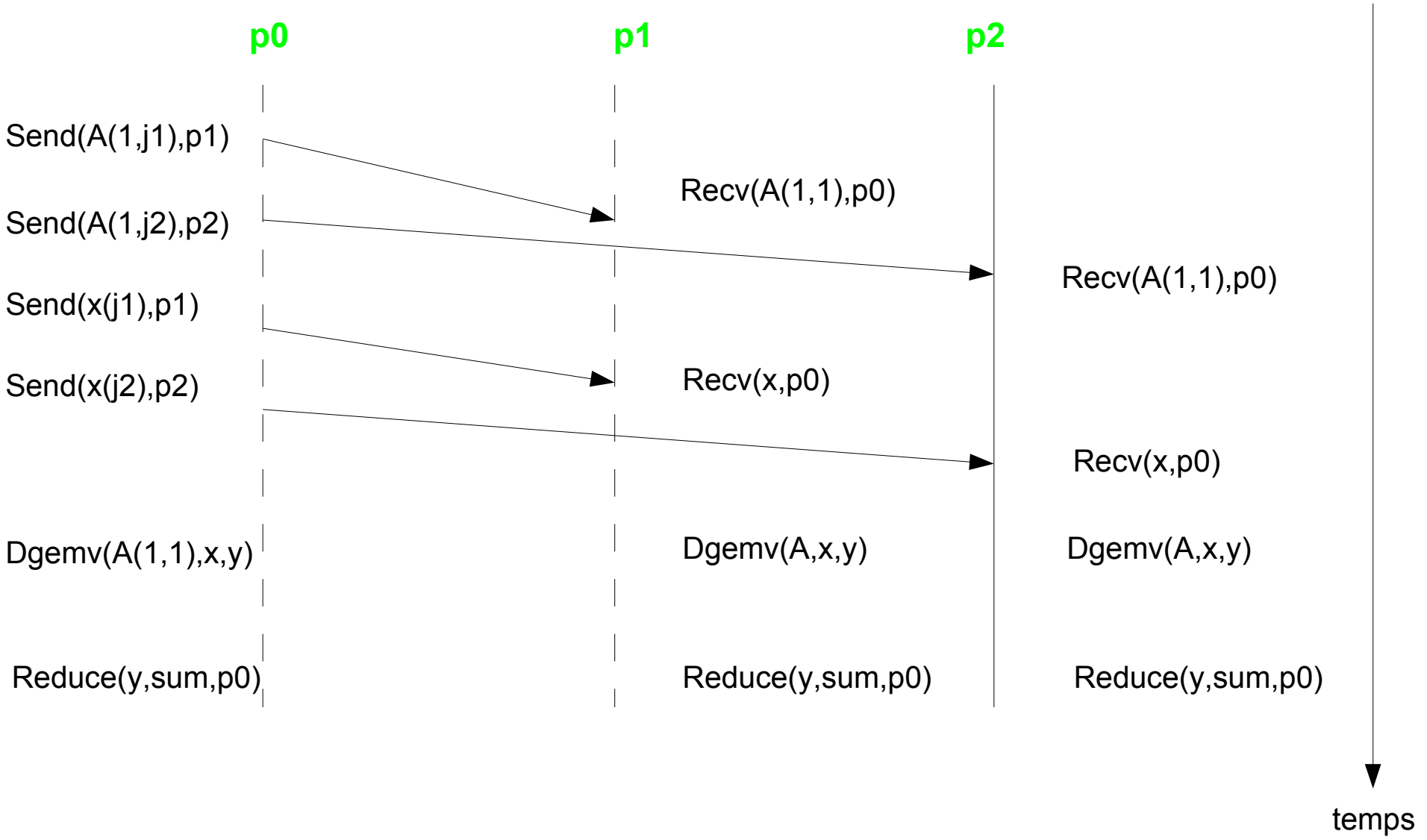
Exemple: produit matrice-vecteur

- 3 processus: p0, p1, p2
- Opération: y = A x



- Algorithme:
  - p0 distribue A par colonne aux autres processus
  - p0 distribue un sous-ensemble de x aux autres processus
  - Tous les processus effectuent un produit matrice-vecteur local
  - Réduction globale pour assembler la solution

Exemple: produit matrice-vecteur



Exemples de facteurs influant sur la performance

- Distribution des données, taille de bloc dans les algorithmes
- Recouvrement des communications par des calculs
- Equilibrage de charge (load-balancing)
- Bande passante du réseau
- Nombre de fois où un message est copié ou lu (ex. checksum)
- Taille du message (ex: puissances de 2 ou longueurs de ligne de cache peuvent donner de meilleures performances que des tailles plus petites
- Taille des buffers (alignement sur un mot, ligne de cache, page)
- Accès aux ressources partagées
- Protocole différent pour messages courts et longs
- Ecriture du code parallèle (ex: receive exécuté avant le send)

III – Présentation de MPI

Introduction

- Définition d'un standard de transfert de messages à destination des développeurs
- **Objectifs**: portabilité, simplicité, développement du calcul distribué, implantation par les constructeurs
- **Cible**: machines multiprocesseurs, clusters, réseaux
- Standard officiel: http://www.mpi-forum.org
  (dernière version: MPI 3.2, 06/2015)
- MPI-2: création et gestion de processus, “one-sided” communications, opérations collectives étendues, interfaces externes, I/O, langages additionnels (C++)
- Autres infos (présentations, didacticiels, FAQ...):
  http://www.mcs.anl.gov/mpi

Caractéristiques de MPI

- Parallélisme de tâches et communications par passage de messages
- Code source unique, exécuté par un ensemble de processus (SPMD, Single Program Multiple Data)
- Communicateurs: encapsulent les espaces de communications
- Types de données: élementaires, vecteurs, personnalisés
- Différents modes de communication: asynchrone bloquant, non bloquant, synchrone, bufferisé
- Opérations collectives: barrière, broadcast, scatter/gather, réduction (max, somme, produit...)
- Topologies virtuelles de processus: identification des voisins

Structure d'un programme MPI

```
#include "mpi.h"
...
main (int argc, char *argv [])
{
   ...
   MPI_Init (&argc, &argv) ;
   ...
   MPI_Finalize () ;
   ...
}
```

Exemple 1: qui suis-je, combien sommes-nous?

- MPI_Comm_size: donne le nombre de processeurs
- MPI_Comm_rank: donne le numéro (rang) de processeur identifiant le processus appelant dans le groupe (compris entre 0 et size-1)
- Groupe: ensemble de processus pouvant communiquer
- Contexte: environnement dans lequel un message est envoyé puis reçu
- Communicateur = Groupe + Contexte
- Communicateur prédéfini: MPI_COMM_WORLD (tous les processus disponibles pour le job MPI)
  Défini dans bibliothèque mpi.h (en C) ou mpif.h (en Fortran)

Exemple 1: code C

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

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

Exemple 1: code Fortran

```
program main
include 'mpif.h'
integer ierr, rank, size

call MPI_INIT( ierr )
call MPI_COMM_RANK( MPI_COMM_WORLD, rank, ierr )
call MPI_COMM_SIZE( MPI_COMM_WORLD, size, ierr )
print *, 'I am ', rank, ' of ', size
call MPI_FINALIZE( ierr )
end
```

Exécution d'un code MPI

- Chaque instruction est exécutée indépendamment par chaque processeur (y compris les impressions)
- Exécution: `mpirun -np nbprocs code` (pas dans MPI-1)
- Exemple 1: `mpirun -np 4 ex1`
- Résultat:
  “I am 0 of 4”
  “I am 1 of 4”
  …
  mais pas forcément dans l'ordre des processeurs
- MPI-2: `mpiexec <args>`
  (juste une recommandation)

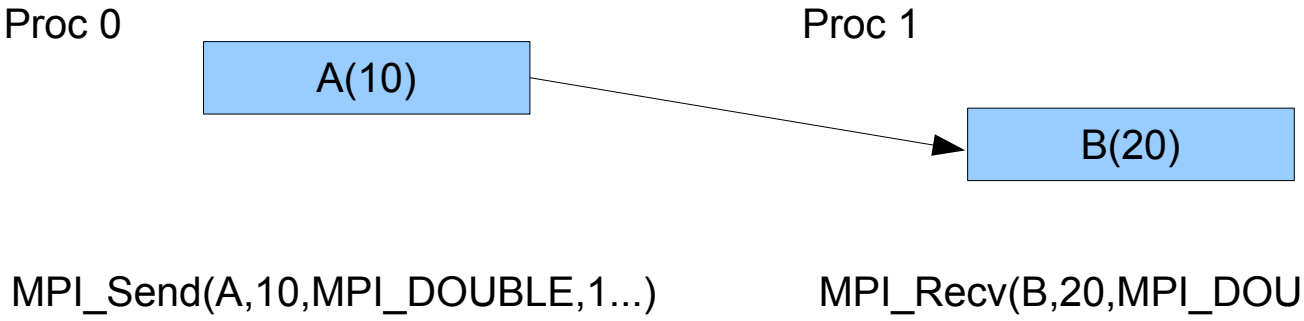
Description d'un message MPI

- Donnée:
  - Adresse
  - Nombre de valeurs
  - Type de données (exemple sur slide suivant)
- Enveloppe:
  - Rang du destinataire (si émission)
  - Rang de l'émetteur (si réception)
  - Tag (int): distingue les messages d'un couple émetteur/dest.
  - Communicateur

Types de données (MPI_Datatype)

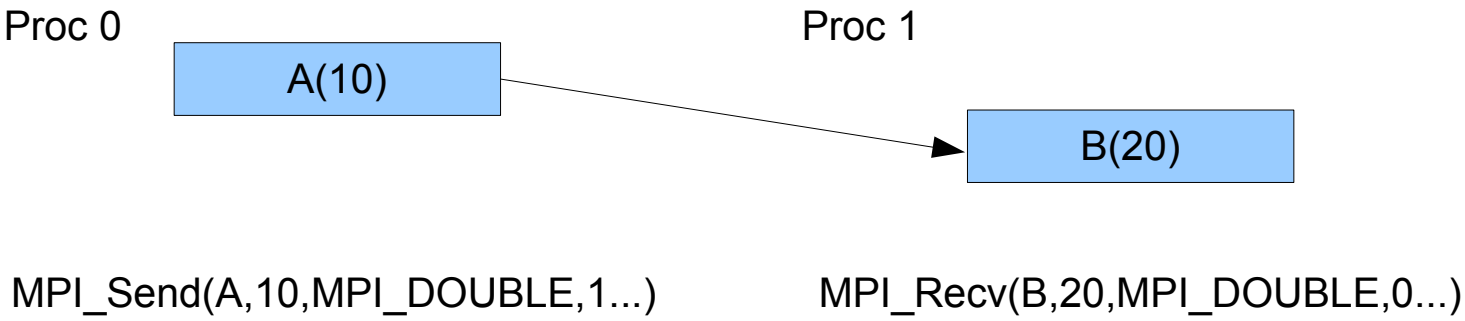
- Correspondances avec les types du C:
  - MPI_CHAR: signed char
  - MPI_INT: signed int
  - MPI_SHORT: signed short int
  - MPI_LONG: signed long int
  - MPI_FLOAT: float
  - MPI_DOUBLE: double
  - MPI_LONG_DOUBLE: long double
  - MPI_PACKED: <<struct>>
- Types particuliers:
  - MPI_BYTE: pas de conversion
  - MPI_PACKED: types construits

Emission simple (bloquante)



- **`MPI_Send(start, count, datatype, dest, tag, comm)`**
  - Buffer d'envoi spécifié par **`(start, count, datatype)`**
  - Destinataire identifié par son rang **`dest`** dans le communicateur **`comm`**)
  - En retour de fonction, la donnée à été envoyée au système et le buffer d'envoi peut être ré-utilisé
  - Le message peut ne pas avoir été reçu par le destinataire

Réception simple (bloquante)



- **`MPI_Recv(start, count, datatype, source, tag, commm, status)`**
  - Attend jusqu'à ce que le message soit reçu et que le buffer de réception soit libéré
  - Expéditeur identifié par son rang **`source`** dans le communicateur **`comm`** ou par **`MPI_ANY_SOURCE`**)
  - Possible de recevoir moins de **`count`** occurences de type **`datatype`** (mais en recevoir plus génère une erreur)

Exemple 2: envoi/réception (C)

```
#include "mpi.h"
#include <stdio.h>
int main( int argc, char *argv[])
{
  int rank, buf;
  MPI_Status status;
  MPI_Init(&argv, &argc);
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );

  /* Process 0 sends and Process 1 receives */
  if (rank == 0) {
    buf = 123456;
    MPI_Send( &buf, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
  }
  else if (rank == 1) {
    MPI_Recv( &buf, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
              &status );
    printf( "Received %d\n", buf );
  }

  MPI_Finalize();
  return 0;
}
```

Exemple 2: envoi/réception (Fortran)

```
      program main
      include 'mpif.h'
      integer rank, buf, ierr, status(MPI_STATUS_SIZE)

      call MPI_Init(ierr)
      call MPI_Comm_rank( MPI_COMM_WORLD, rank, ierr )
C  Process 0 sends and Process 1 receives
      if (rank .eq. 0) then
         buf = 123456
         call MPI_Send( buf, 1, MPI_INTEGER, 1, 0,
     *                              MPI_COMM_WORLD, ierr )
      else if (rank .eq. 1) then
         call MPI_Recv( buf, 1, MPI_INTEGER, 0, 0,
     *                                 MPI_COMM_WORLD, status, ierr )
         print *, "Received ", buf
      endif
      call MPI_Finalize(ierr)
      end
```

Communications non-bloquantes

Les opérations non bloquantes renvoient immédiatement un paramètre de requête pour tester ou attendre la disponibilité de la ressource

```
MPI_Request request;
MPI_Status status;
MPI_Isend(start, count, datatype, dest, tag,
comm, &request);
MPI_Irecv(start, count, datatype, src, tag,
comm, &request);
....
MPI_Wait(&request, &status);
```

ou

```
MPI_Test(&request, &flag, &status);
```

Communications collectives dans MPI

- Fonctions SPMD
- 3 types d'opérations: mouvements de données, calculs collectifs, synchronisation
- Les opérations collectives doivent être appelées par tous les processus d'un communicateur
- Pas d'appels collectifs non bloquants
- MPI-2: appels collectifs inter-communicateurs
- Dans de nombreux cas, possibilité de remplacer de multiples `Send/Recv` par un seul `Bcast/Reduce`

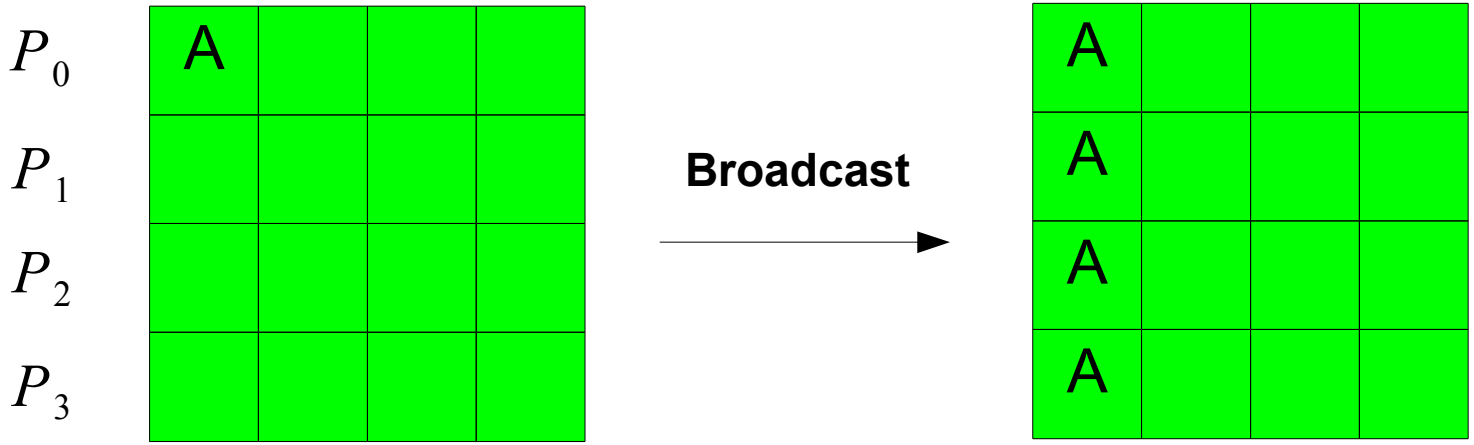
Opérations collectives

- Synchronisation des processus en un point de rendez-vous (pas d'échange d'information)

```
MPI_Barrier (comm)
```

- Diffusion: un même processus envoie une même valeur à tous les autres processus d'un communicateur

```
MPI_Bcast (message, count, datatype, root, comm)
```



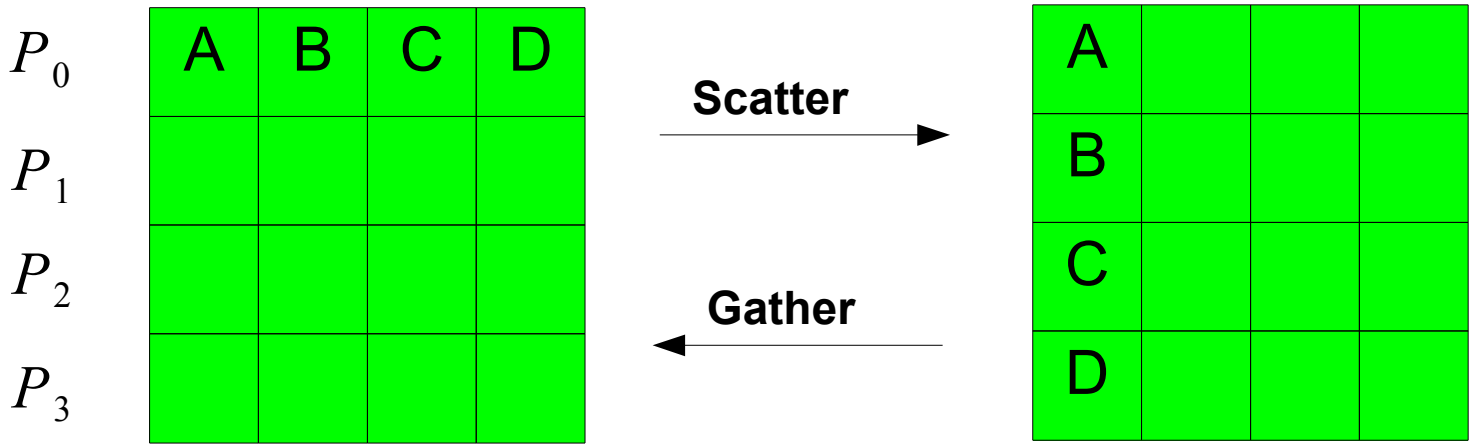
Opérations collectives

- **Scatter:** distribution d'un message personnalisé aux autres processus (one to all)

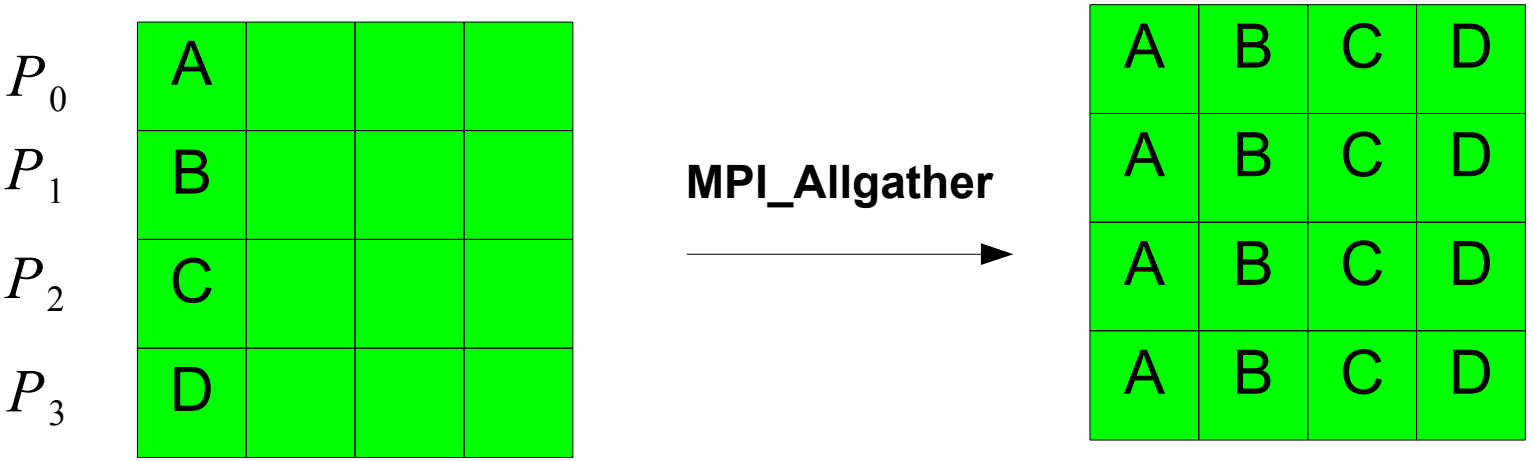
```
MPI_Scatter (send_buf, send_count, send_type,
recv_buf, recv_count, recv_type, root, comm)
```

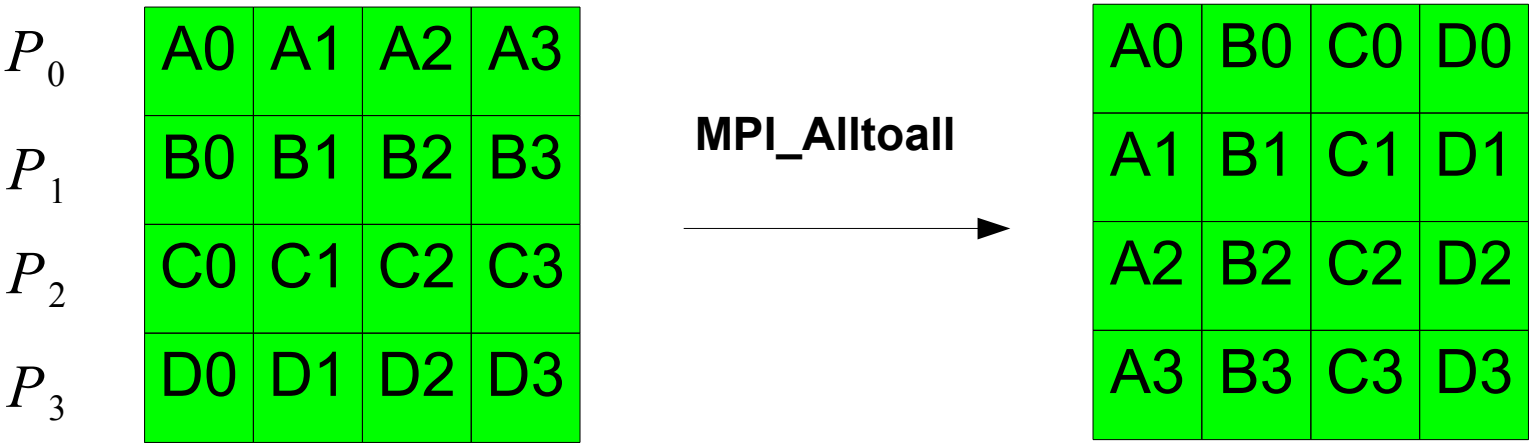
- **Gather:** mise à bout des messages de chacun des processus (all to one)

```
MPI_Gather (send_buf, send_count, send_type,
recv_buf, recv_count, recv_type, root, comm)
```



Autres opérations collectives

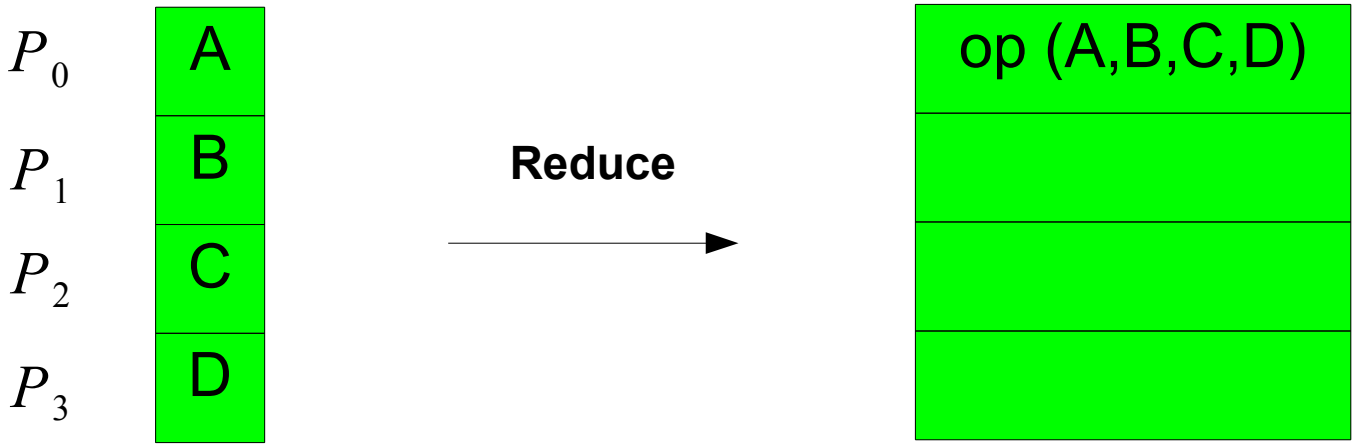




Calcul collectif

- Reduce: collecte par un processus d'un ensemble de valeurs détenues par tous les processus et réduction de cette valeur

```
MPI_Reduce (operand, result, count, datatype,
op, root, comm)
```



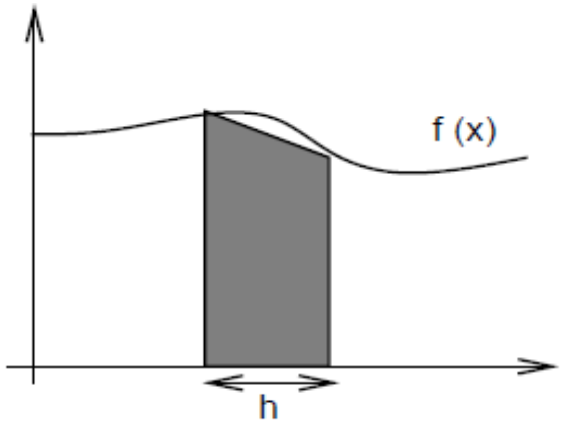
- Le résultat se trouve sur le processus `root`. Pour avoir le résultat sur chaque processus, utiliser `MPI_Allreduce`
- Opérations prédéfinies (`MPI_MAX`, `MPI_SUM`...), possibilité de définir de nouvelles opérations

Les quelques routines utilisées en pratique

- Communications point à point
  `MPI_SEND/MPI_RECV`
  `MPI_ISEND/MPI_IRECV`
  `MPI_WAIT`
- Initialisation/terminaison
  `MPI_INIT`
  `MPI_FINALIZE`
- Informations sur processus
  `MPI_COMM_RANK`
  `MPI_COMM_SIZE`
- Communications collectives
  `MPI_BCAST`
  `MPI_ALLREDUCE`
  `MPI_ALLGATHER`

Exercice: calcul de π par intégration

- On calcule π par évaluation de l'intégrale $\pi = \int_0^1 \frac{4}{1+x^2} dx = \int_0^1 f(x)\,dx$
- L'évaluation se fait par la méthode des trapèzes en divisant l'intervalle $[0,1]$ en n intervalles $[x_{0,} x_1], \ldots, [x_{n-1}, x_n]$ avec $x_0 = 0, \ldots, x_i = ih, \ldots, x_n = 1$ et $h = \frac{1}{n}$



On a alors: $\int_0^1 f(x)\,dx \simeq \sum_{i=0}^{n-1} h \frac{f(x_i) + f(x_{i+1})}{2}$

- Ecrire l'algorithme correspondant en répartissant le calcul sur p processeurs