Paradigmes et modèles de programmation parallèle distribuée

Camille Coti

Séminaire LCR

30 *octobre* 2014

Plan de l'exposé

1. Modèles théoriques de systèmes distribués
   - Modèle de système distribué
   - Communications par passage de messages
   - Communications par mémoire partagée
2. Mémoire distribuée
   - Communications bilatérales
   - Communications unilatérales
3. Espace d'adressage global
4. Sac de tâches
5. Conclusion

Modèle théorique

Système distribué : ensemble de processus $p_0, p_1, ..., p_{n-1}$ reliés par un **système de communications**

- Chaque processus exécute un programme
- Chaque processus a son propre système de contrôle, son propre flux d'instructions
  $\rightarrow$ SIMD = pas système distribué
- Les processus communiquent entre eux via le système de communications, pas forcément en point-à-point

Configuration du système

- Ensemble des états des processus à un instant donné
- Si e_k = état du processus k, une configuration C est $\bigcup_{k=0}^{n-1} e_k$

Modèle théorique (suite)

Évolution du système :

- Passage d'une configuration à une autre : **transition**
- **Système de transitions** : triplet $S = (C, \rightarrow, I)$ avec :
  - C : ensemble de configurations
  - $\rightarrow$: relation de transition binaire entre deux configurations de C
  - I : sous-ensemble des configurations initiales du système

Exécution d'un système de transitions $S = (C, \rightarrow, I)$:

- Séquence maximale $E = (\gamma_0, \gamma_1, ...)$ avec $\gamma_0 \in I$ où $\gamma_i \rightarrow \gamma_{i+1}$

Accessibilité :

- Une configuration δ est **accessible depuis une configuration** γ si il existe une séquence $\gamma_0, \gamma_1, ..., \gamma_k$ avec $\gamma = \gamma_0$ et $\gamma_k = \delta$ et $\gamma_i \rightarrow \gamma_{i+1}$
- δ est **accessible** si $\gamma_0 \in I$ (il existe un ensemble de transitions entre un état initial et δ)

Communications par passage de messages

Les processus sont eux-mêmes des **systèmes à états** , qui passent d'un état à un autre grâce à des **évènements** pouvant être :

- internes : définis par l'algorithme exécuté
- réceptions : messages qui arrivent du système de communications
- envois : messages envoyés sur le système de communications

Passage de messages :

- Les processus exécutent des **primitives d'envoi/réception** :
  - $send(buffer, destinataire)$
  - $receive(buffer, source)$
- Tout envoi **doit correspondre** à une réception (et inversement)
- **Asynchrone** : la communication se fait en un temps fini mais non borné

Communications par mémoire partagée

Modèle **à état** :

- Chaque processus a un ensemble de processus voisins
- Chaque processus peut lire **l'état** (dans son intégralité) de ses voisins
- NB : pour un processus p, chacun de ses voisins lira le même état de p.

Modèle **link-register** :

- Il existe des **registres de mémoire** entre deux processus (ou plus)
- Les processus qui accèdent à un registre r peuvent écrire (primitive $write(buffer, r)$) ou lire (primitive $read(buffer, r)$) **atomiquement** dans ce registre.

1 Modèles théoriques de systèmes distribués
- Modèle de système distribué
- Communications par passage de messages
- Communications par mémoire partagée

2 Mémoire distribuée
- Communications bilatérales
- Communications unilatérales

3 Espace d'adressage global

4 Sac de tâches

5 Conclusion

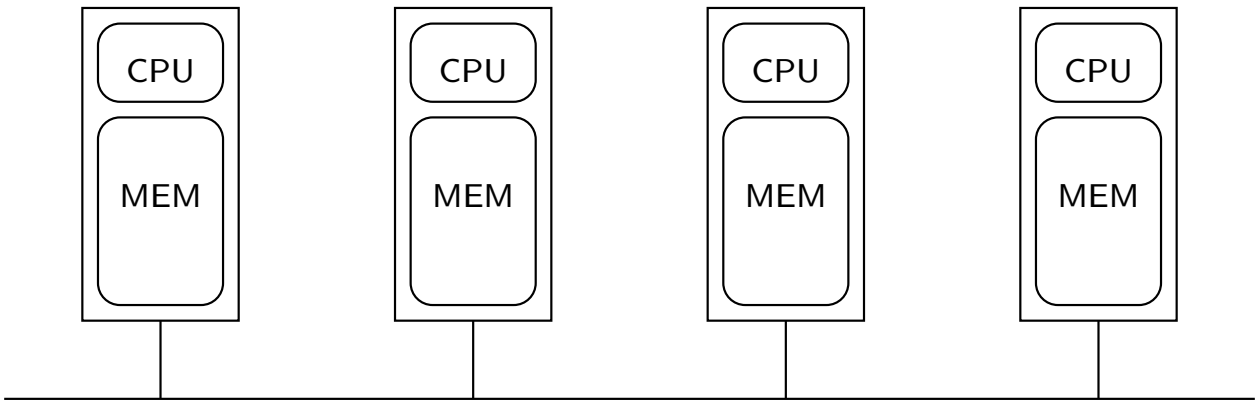
Mémoire distribuée

Concrètement :

- Un **ensemble de processus**
- Chaque processus a sa **mémoire propre**
- Un réseau pair-à-pair les relie : réseau (Ethernet, IB, Myrinet, Internet...) ou bus système

Le programmeur a à sa charge **la localité des données**

- Déplacement **explicite** des processus entre processus
- Si un processus P_i a besoin d'une donnée qui est dans la mémoire du processus P_j, le programmeur doit la déplacer explicitement de P_j vers P_i

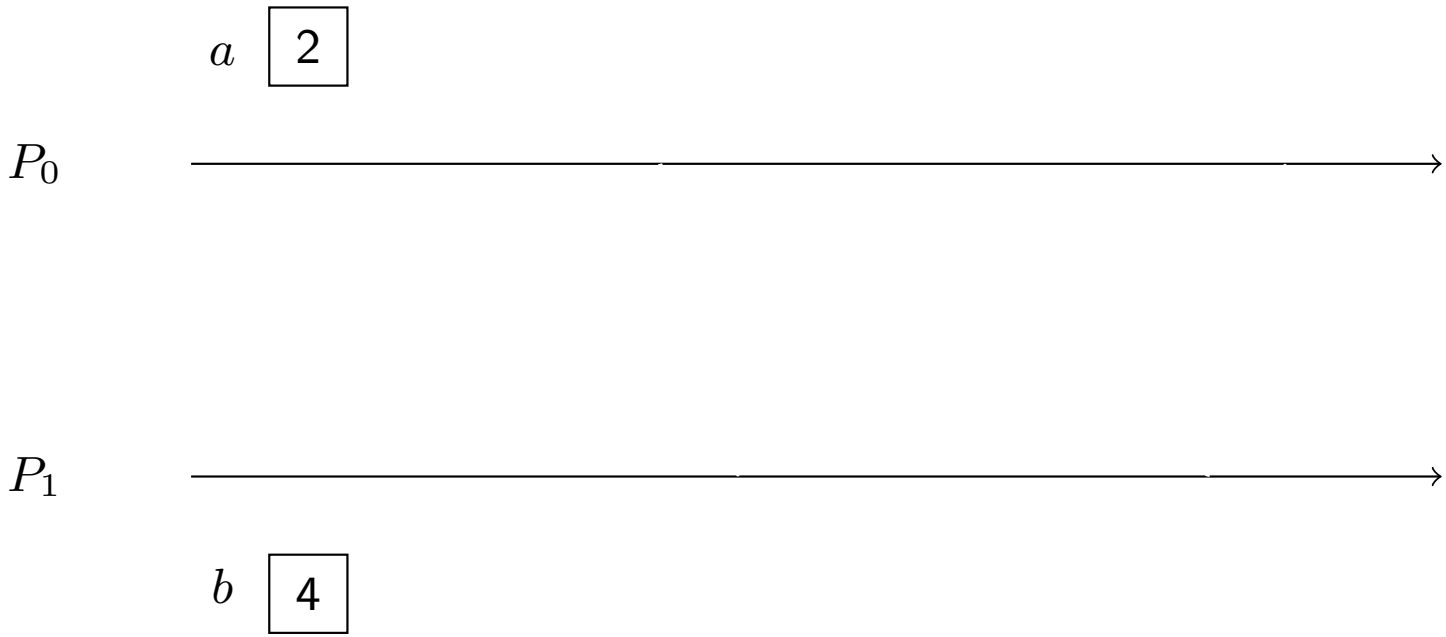


Communications bilatérales

Communications bilatérales

- Primitives $send/recv$
- Une primitive $send$ **doit** matcher une primitive $recv$ (et inversement)

Conséquence : quand on déplace des données entre deux processus, les deux processus participent *activement* au déplacement des données

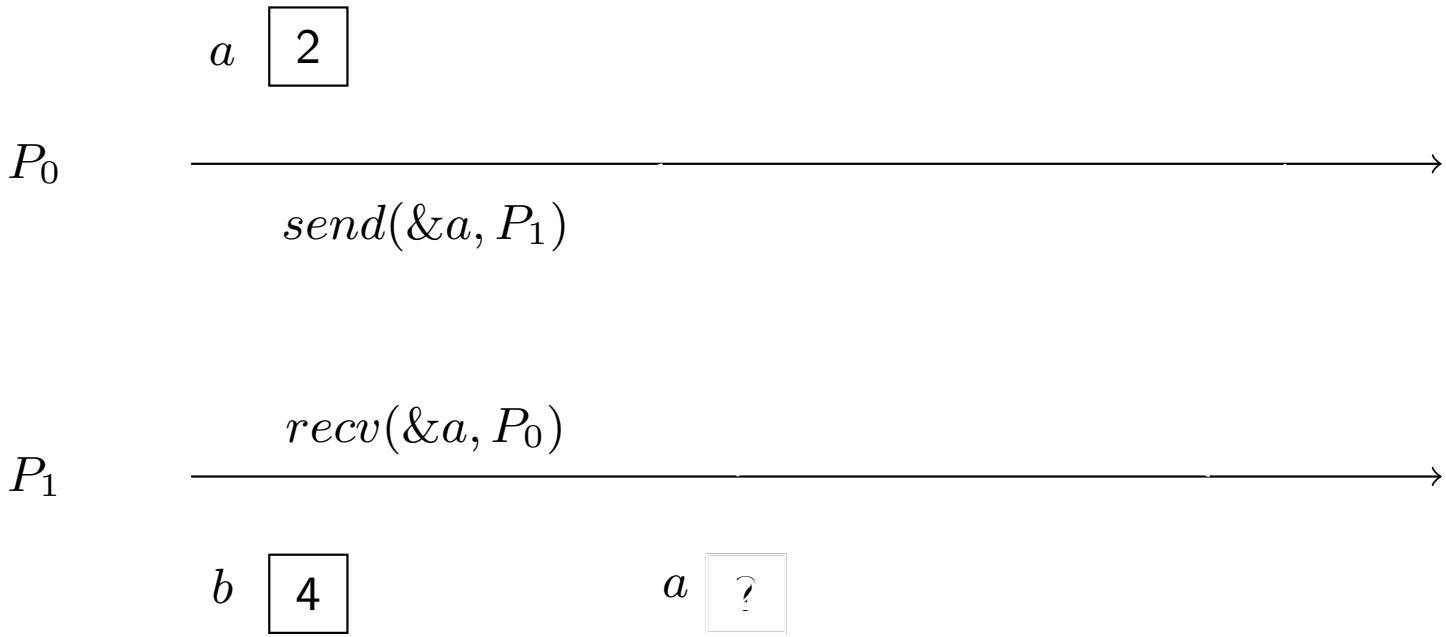


Communications bilatérales

Communications bilatérales

- Primitives $send/recv$
- Une primitive $send$ **doit** matcher une primitive $recv$ (et inversement)

Conséquence : quand on déplace des données entre deux processus, les deux processus participent *activement* au déplacement des données

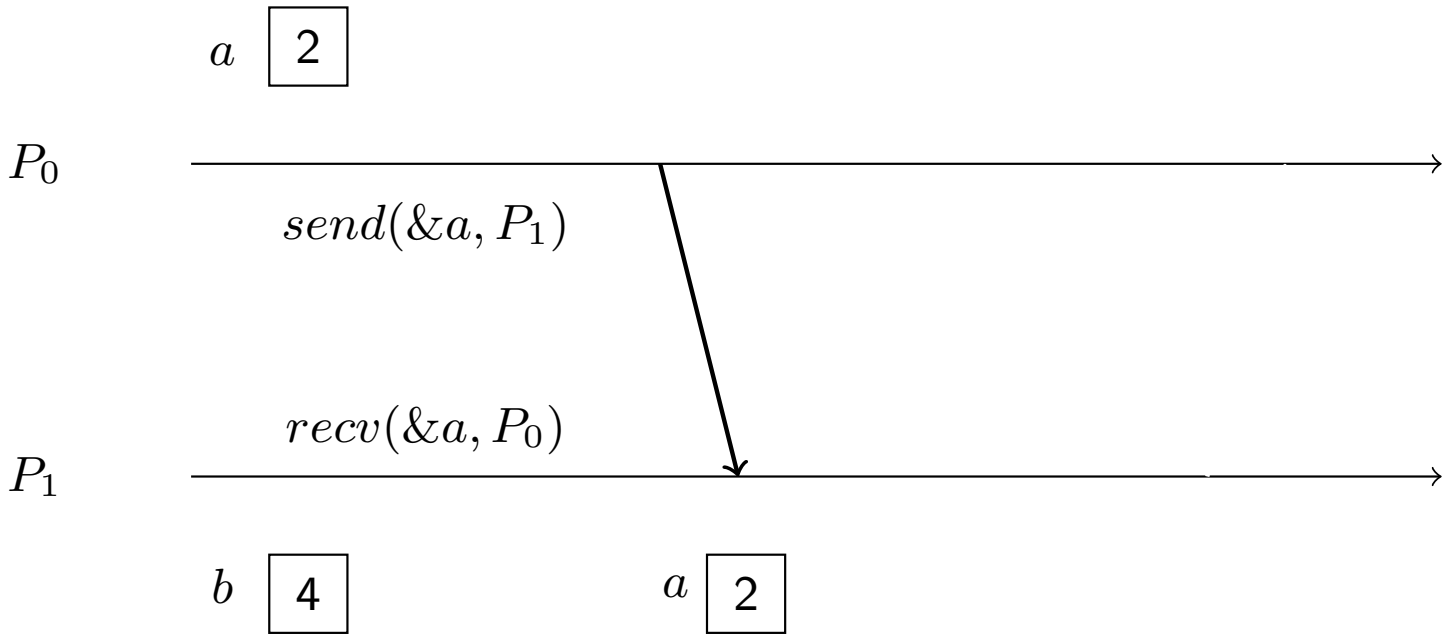


Communications bilatérales

Communications bilatérales

- Primitives $send/recv$
- Une primitive $send$ **doit** matcher une primitive $recv$ (et inversement)

Conséquence : quand on déplace des données entre deux processus, les deux processus participent *activement* au déplacement des données

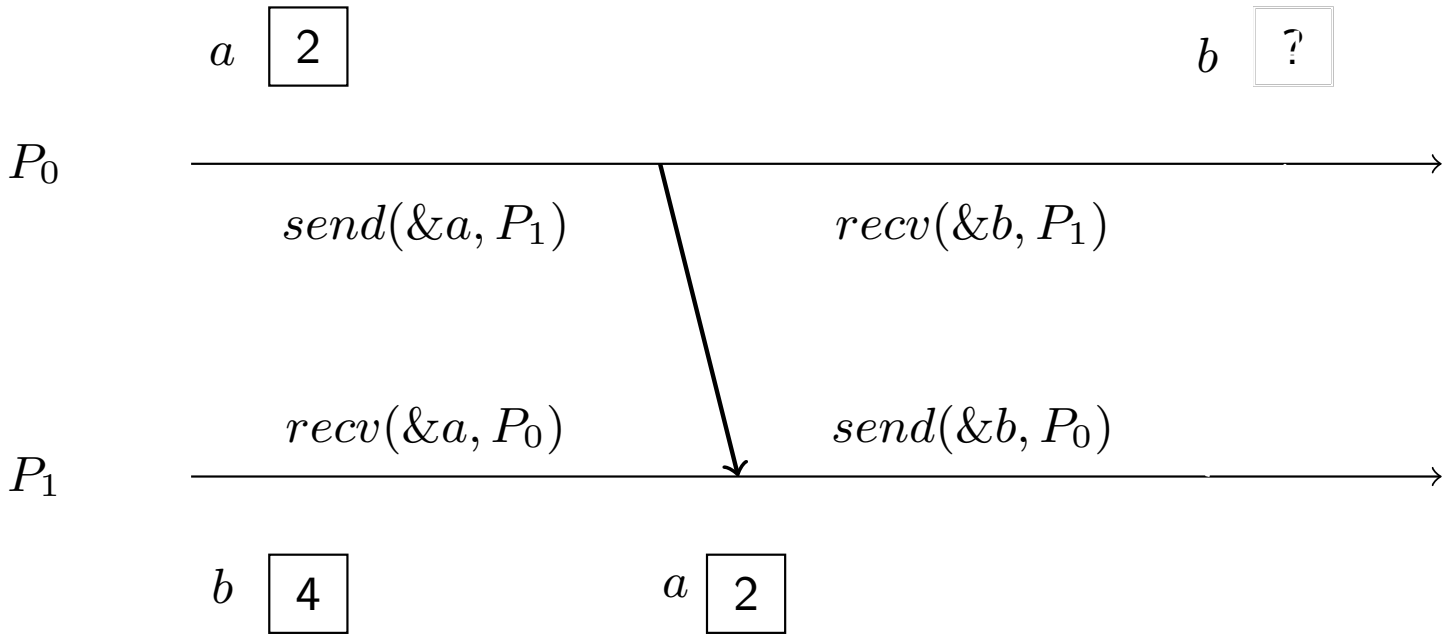


Communications bilatérales

Communications bilatérales

- Primitives $send/recv$
- Une primitive $send$ **doit** matcher une primitive $recv$ (et inversement)

Conséquence : quand on déplace des données entre deux processus, les deux processus participent *activement* au déplacement des données

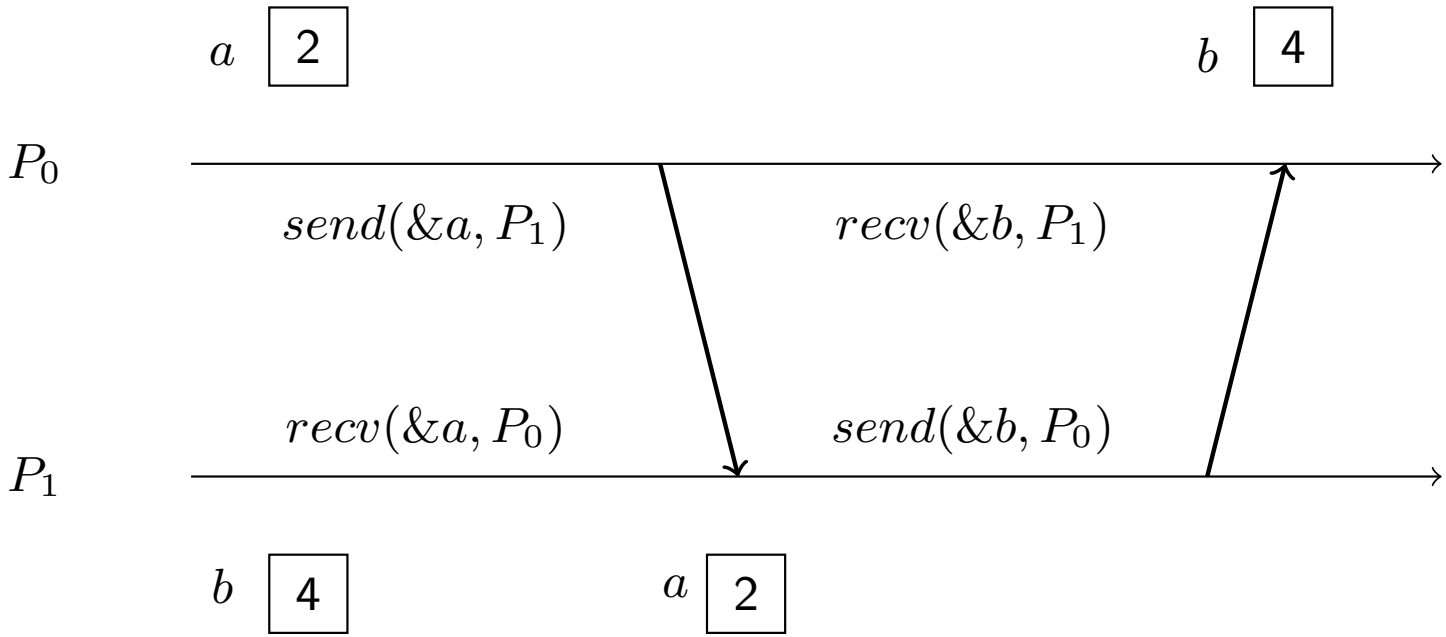


Communications bilatérales

Communications bilatérales

- Primitives $send/recv$
- Une primitive $send$ **doit** matcher une primitive $recv$ (et inversement)

Conséquence : quand on déplace des données entre deux processus, les deux processus participent *activement* au déplacement des données



Exemple

Exemple de bibliothèque de programmation parallèle distribuée par communications bilatérales : **MPI**

- Standard *de facto* en programmation parallèle
- Maîtrise totale de la localité des données (“*l'assembleur de la programmation parallèle*”)
- Portable
- Puissant : permet d'écrire des programmes dans d'autres modèles
- Communications point-à-point mais aussi collectives

Avantages :

- Totale maîtrise de la localité des données
- Très bonnes performances

Inconvénients :

- Besoin de la coopération des deux processus : source et destination
- Fort synchronisme

MPI : Exemple

Exemple : ping-pong

- Le rang 0 envoie un jeton
- Le rang 1 le reçoit et le renvoie au rang 0
- Le rang 0 le reçoit.

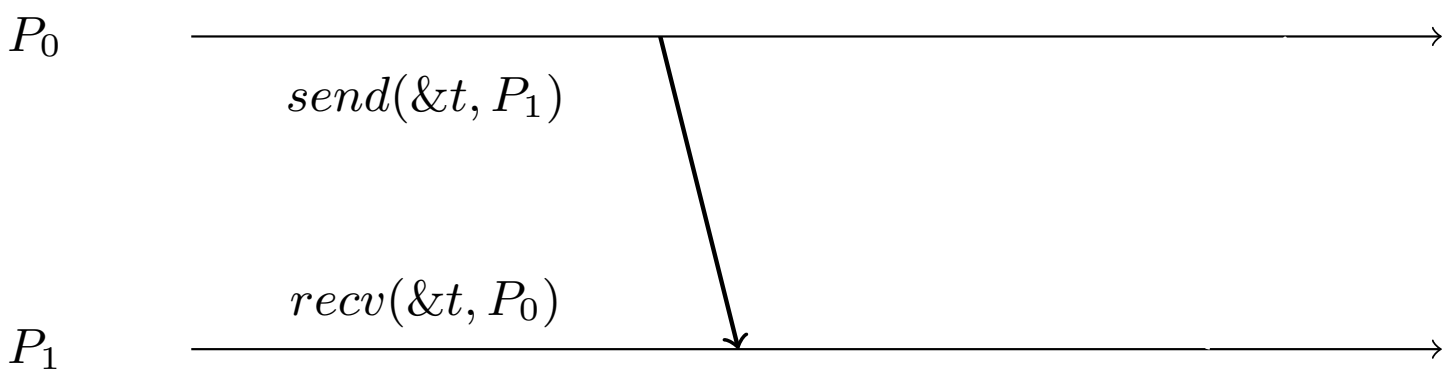
```
if( 0 == rank ) {
    MPI_Send( &token, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    MPI_Recv( &token, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &status );
} else if( 1 == rank ) {
    MPI_Recv( &token, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status );
    MPI_Send( &token, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
}
```

MPI : Exemple

Exemple : ping-pong

- Le rang 0 envoie un jeton
- Le rang 1 le reçoit et le renvoie au rang 0
- Le rang 0 le reçoit.

```
if( 0 == rank ) {
    MPI_Send( &token, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    MPI_Recv( &token, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &status );
} else if( 1 == rank ) {
    MPI_Recv( &token, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status );
    MPI_Send( &token, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
}
```

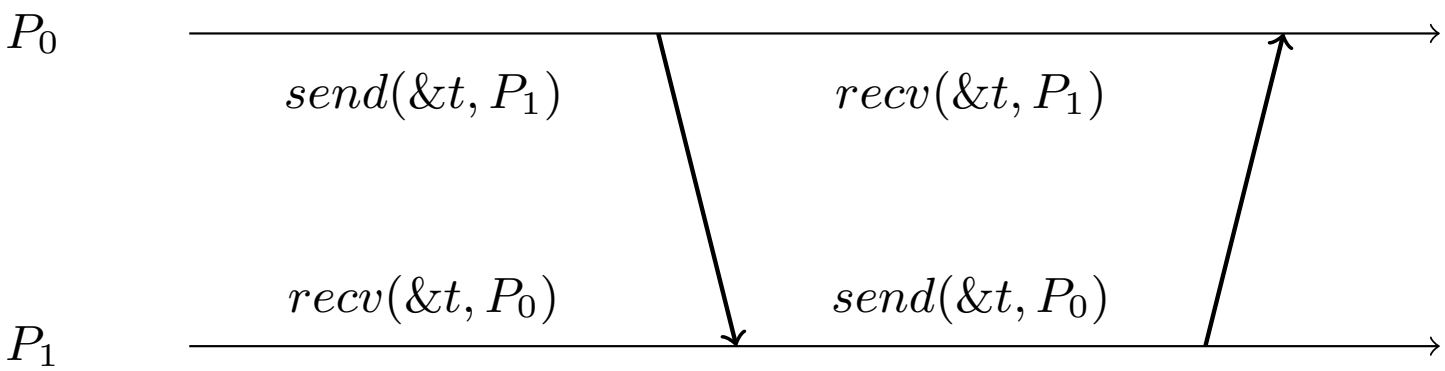


MPI : Exemple

Exemple : ping-pong

- Le rang 0 envoie un jeton
- Le rang 1 le reçoit et le renvoie au rang 0
- Le rang 0 le reçoit.

```
if( 0 == rank ) {
    MPI_Send( &token, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    MPI_Recv( &token, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &status );
} else if( 1 == rank ) {
    MPI_Recv( &token, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status );
    MPI_Send( &token, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
}
```

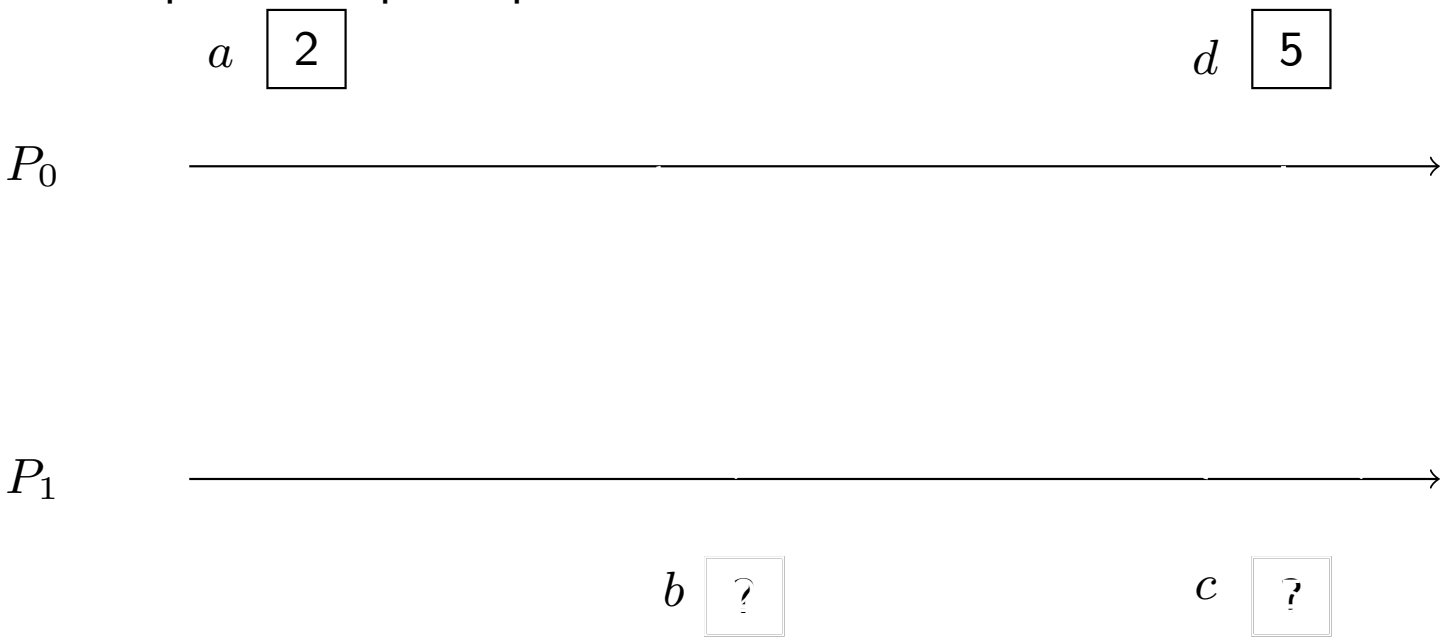


Communications unilatérales

Communications unilatérales

- Primitives put/get
- Modèle RDMA : Remote Direct Memory Access
- Un processus peut aller lire/écrire dans la mémoire d'un autre processus
- Concrètement : possible grâce à des cartes réseaux RDMA (InfiniBand, Myrinet...)

Un seul processus participe à la communication.

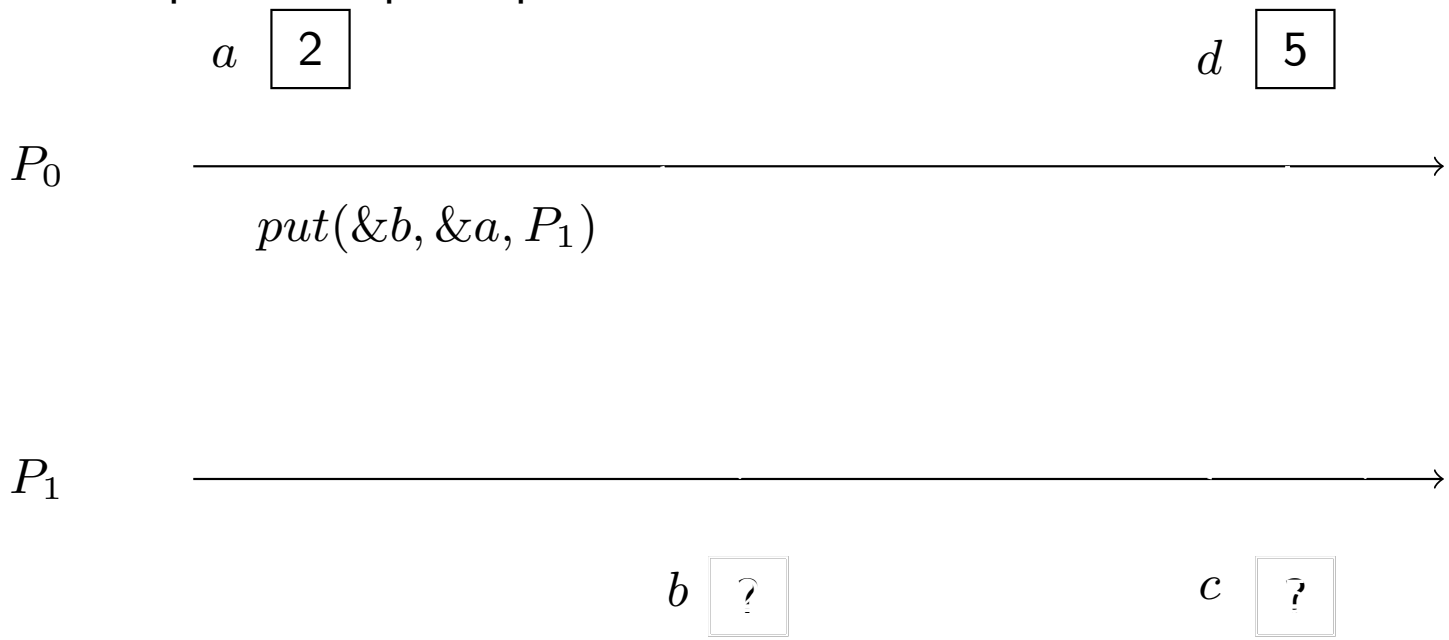


Communications unilatérales

Communications unilatérales

- Primitives put/get
- Modèle RDMA : Remote Direct Memory Access
- Un processus peut aller lire/écrire dans la mémoire d'un autre processus
- Concrètement : possible grâce à des cartes réseaux RDMA (InfiniBand, Myrinet...)

Un seul processus participe à la communication.

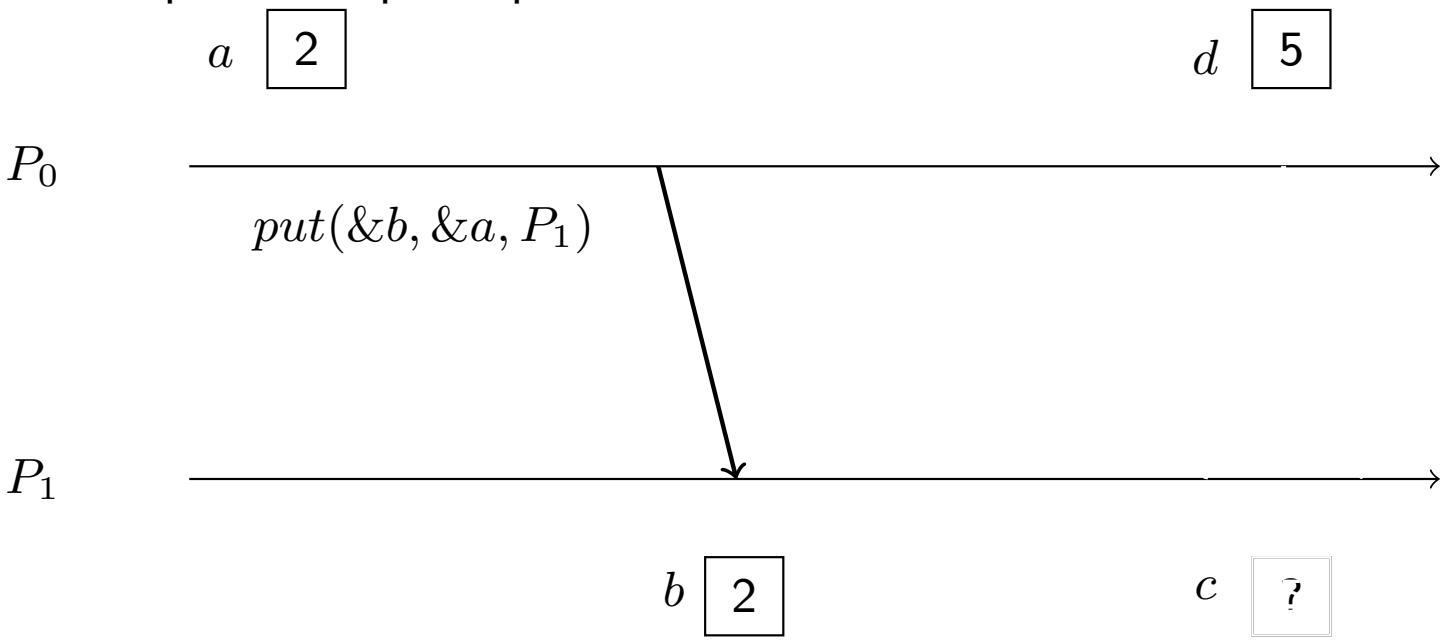


Communications unilatérales

Communications unilatérales

- Primitives put/get
- Modèle RDMA : Remote Direct Memory Access
- Un processus peut aller lire/écrire dans la mémoire d'un autre processus
- Concrètement : possible grâce à des cartes réseaux RDMA (InfiniBand, Myrinet...)

Un seul processus participe à la communication.

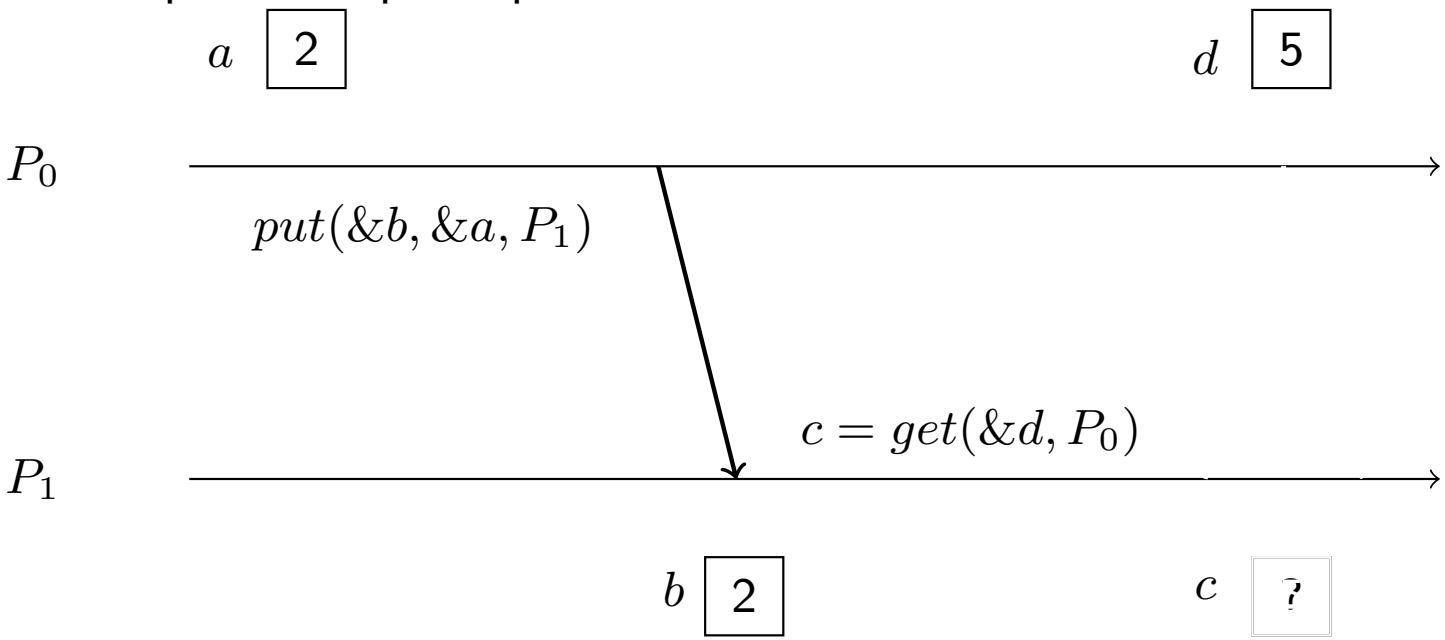


Communications unilatérales

Communications unilatérales

- Primitives put/get
- Modèle RDMA : Remote Direct Memory Access
- Un processus peut aller lire/écrire dans la mémoire d'un autre processus
- Concrètement : possible grâce à des cartes réseaux RDMA (InfiniBand, Myrinet...)

Un seul processus participe à la communication.

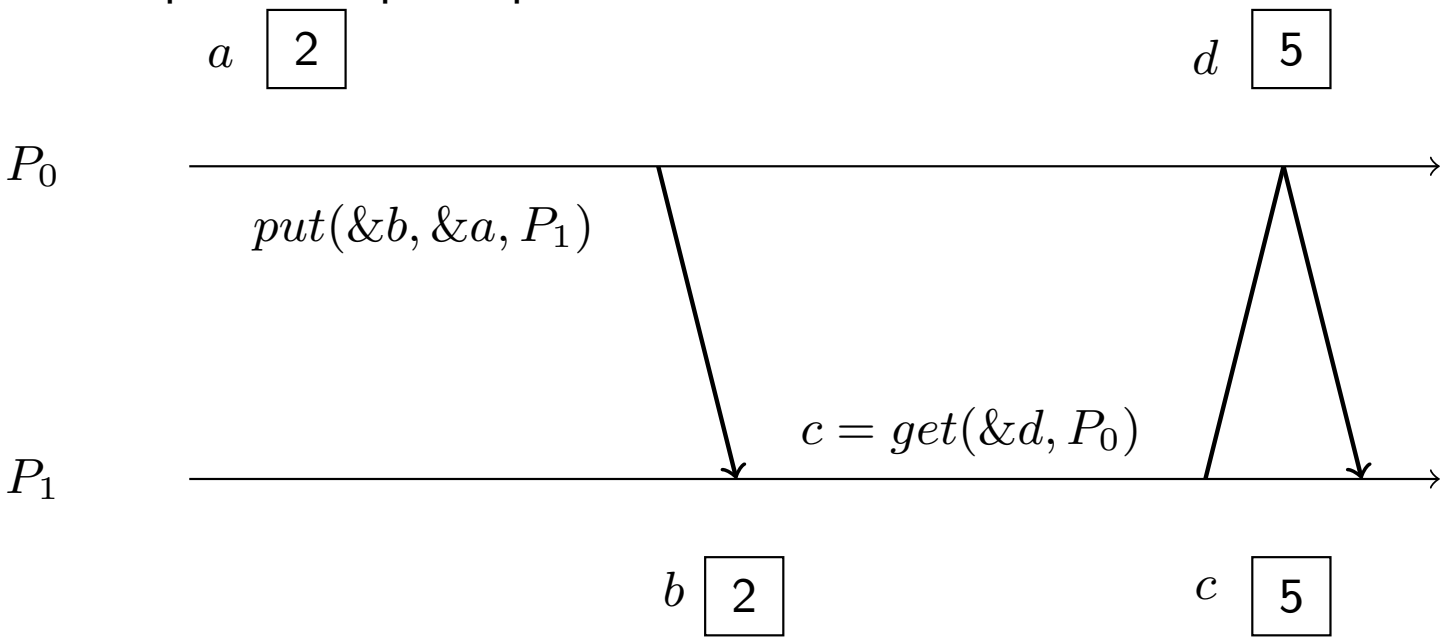


Communications unilatérales

Communications unilatérales

- Primitives put/get
- Modèle RDMA : Remote Direct Memory Access
- Un processus peut aller lire/écrire dans la mémoire d'un autre processus
- Concrètement : possible grâce à des cartes réseaux RDMA (InfiniBand, Myrinet...)

Un seul processus participe à la communication.



Exemples

Exemples :

- Communications unilatérales de **MPI**
- Fonctions put/get d' **UPC**
- **OpenSHMEM**

OpenSHMEM

- Héritier des SHMEM de Cray, SGI SHMEM... des années 90
- Standardisation récente poussée par les architectures actuelles

Avantages :

- Communications très rapides
- Très adapté aux architectures matérielles contemporaines
- Pas besoin que les deux processus soient prêts

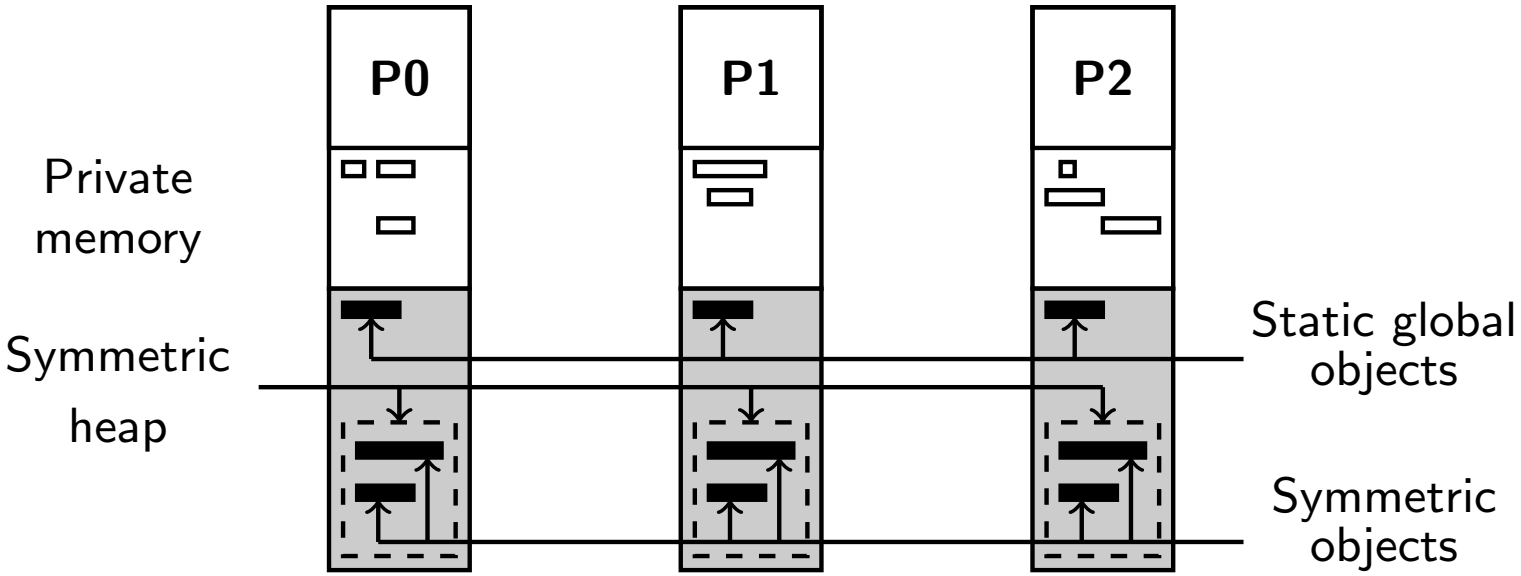
Inconvénients :

- Modèle délicat, risques de race conditions
- Impose une symétrie des mémoires des processus

OpenSHMEM

Modèle de mémoire : **tas symétrique**

- Mémoire privée vs mémoire partagée (tas)
- L'allocation de mémoire dans le tas partagé est une *communication collective*



OpenSHMEM : Exemple

Allocation dans le tas partagé :

- Fonction `shmalloc`
- Attention : collective

Déplacements de données :

- Fonctions `shmem_*_put`, `shmem_*_get`
- Une fonction par type de données

```
short* ptr = (short*)shmalloc( 10 * sizeof( short ) );
if (_my_pe() == 0) {
    shmem_long_put( ptr, source, 10, 1 );
}
```

1 Modèles théoriques de systèmes distribués
- Modèle de système distribué
- Communications par passage de messages
- Communications par mémoire partagée

2 Mémoire distribuée
- Communications bilatérales
- Communications unilatérales

3 Espace d'adressage global

4 Sac de tâches

5 Conclusion

Espace d'adressage global

Principe de l' **espace d'adressage global** :

- Programmer sur mémoire distribuée comme sur mémoire partagée
- Mise à contribution du **compilateur**
- L'union des mémoires distribuées est vue **comme une mémoire partagée**

Concrètement :

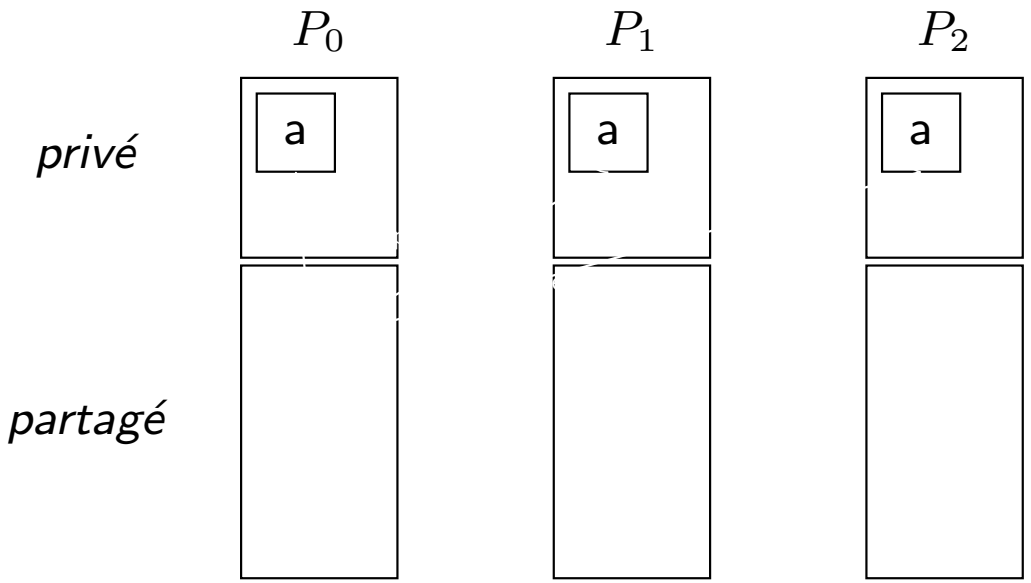
- Le programmeur déclare la **visibilité** de ses variables : privées (par défaut) ou **partagées**
- Pour les tableaux : le programmeur déclare la taille des blocs sur chaque processus
- Le compilateur se charge de
  - **répartir les données partagées** dans la mémoire des différents processus
  - **traduire les accès à des données distantes** $(a = b)$ en communications

Les questions relatives au caractère distribué **ne sont pas vues** par le programmeur.

Exemples

Langages **PGAS** :

- Unified Parallel C (UPC), Titanium, High-Performance Fortran (HPF)



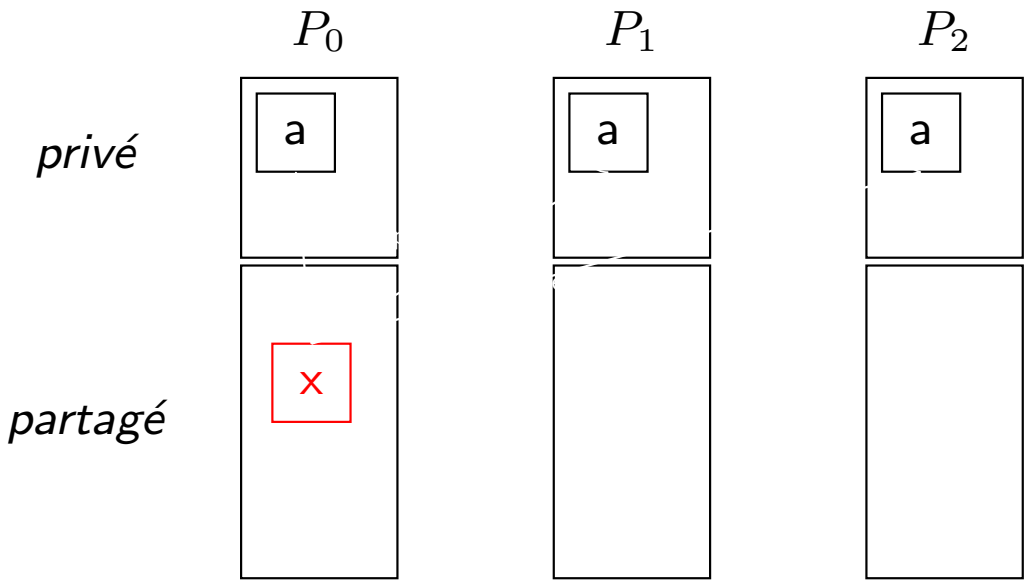
int a;

shared int x;

Exemples

Langages **PGAS** :

- Unified Parallel C (UPC), Titanium, High-Performance Fortran (HPF)



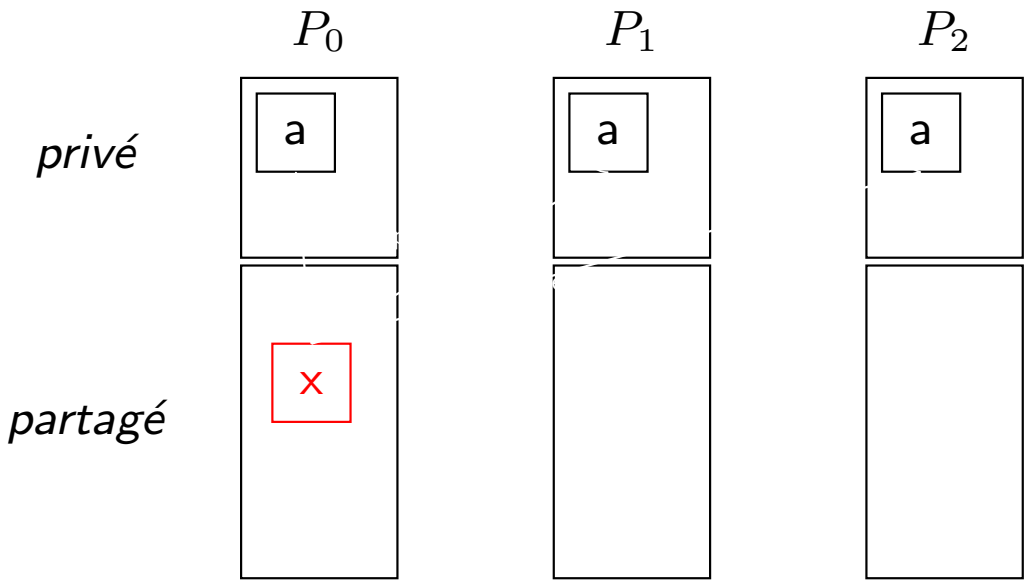
int a;

shared int x;

Exemples

Langages **PGAS** :

- Unified Parallel C (UPC), Titanium, High-Performance Fortran (HPF)

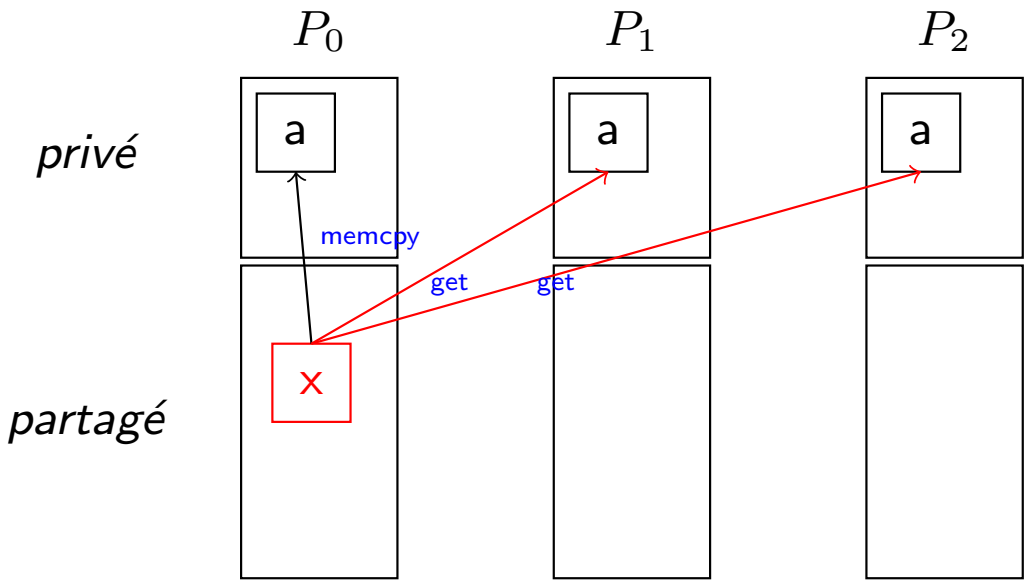



```
int a;
shared int x;
int a = x;
```

Exemples

Langages **PGAS** :

- Unified Parallel C (UPC), Titanium, High-Performance Fortran (HPF)




```
int a;
shared int x;
int a = x;
```

UPC : Exemple

Exemple :

- Une variable `x` est partagée, donc accessibles depuis tous les processus
  - Le compilateur la placera dans la mémoire d'un processus de son choix
- Le processus 0 (appelé `thread` dans la terminologie UPC) l'initialise à 42
- Une barrière globale assure que tous les processus ont atteint ce point du programme
- Tous les processus récupèrent la valeur de `x` dans une variable qui leur est privée
  - Le compilateur génère des communications réseau (vraisemblablement *get*) inter-processus

```
shared int x;
int a;
if( 0 == MYTHREAD ) {
    x = 42;
}
upc_barrier;
a = x;
```

1 Modèles théoriques de systèmes distribués
- Modèle de système distribué
- Communications par passage de messages
- Communications par mémoire partagée

2 Mémoire distribuée
- Communications bilatérales
- Communications unilatérales

3 Espace d'adressage global

4 Sac de tâches

5 Conclusion

Sac de tâches

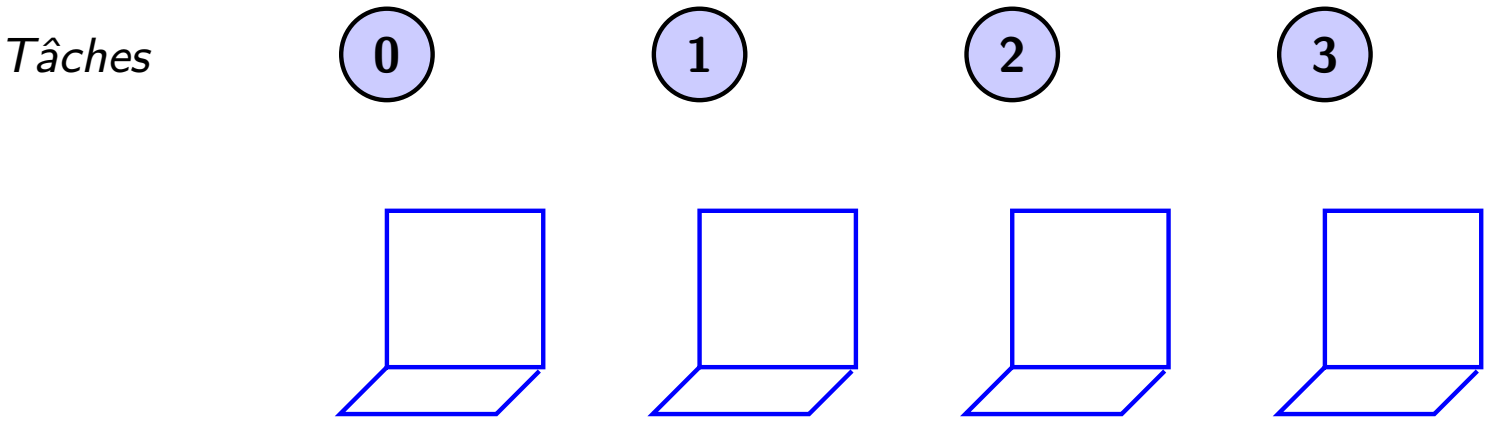
Qu'est-ce qu'un **sac de tâches** ?

- Un ensemble de calculs à faire
- **Indépendants** les uns des autres

Ces calculs peuvent être faits en **parallèle** les uns des autres

→ Un sac de tâches se parallélise *extrêmement* bien !

Pas de communications entre les processus exécutant les tâches.



Sac de tâches

Qu'est-ce qu'un **sac de tâches** ?

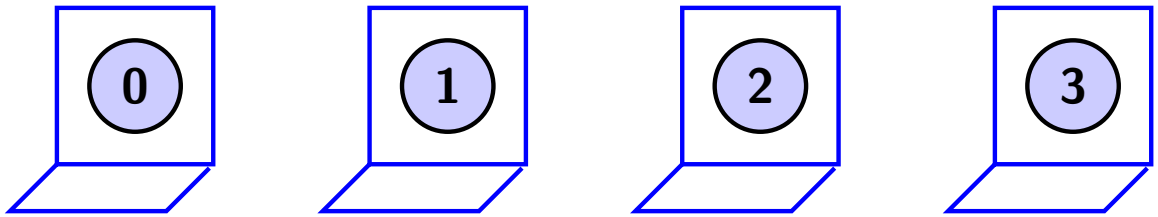
- Un ensemble de calculs à faire
- **Indépendants** les uns des autres

Ces calculs peuvent être faits en **parallèle** les uns des autres

→ Un sac de tâches se parallélise *extrêmement* bien !

Pas de communications entre les processus exécutant les tâches.

Tâches



Résultats

Sac de tâches

Qu'est-ce qu'un **sac de tâches** ?

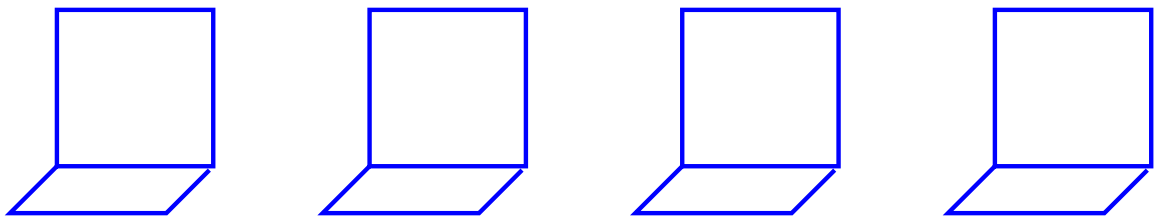
- Un ensemble de calculs à faire
- **Indépendants** les uns des autres

Ces calculs peuvent être faits en **parallèle** les uns des autres

→ Un sac de tâches se parallélise *extrêmement* bien !

Pas de communications entre les processus exécutant les tâches.

Tâches

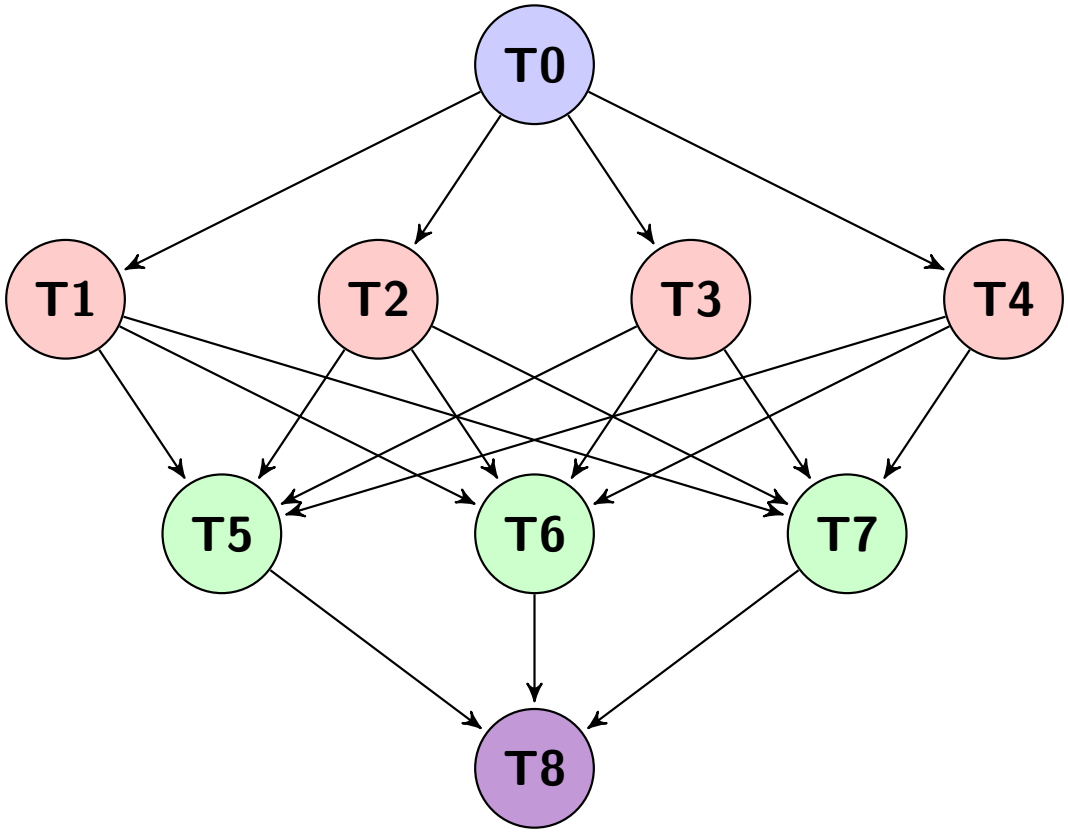


Résultats

Sac de tâches

Possibilité d'avoir un calcul en plusieurs phases :

- Définition de relations de dépendances entre des tâches
- Représentation sous forme d'un DAG



Exemples

Plein de façons d'implémenter un sac de tâches !

- **MPI** → un maître qui distribue le travail à des esclaves et récupère le résultat
- **HTCondor** → conçu spécifiquement pour ça, ordonnance des DAG sur un pool de nœuds
- **MapReduce** → un peu particulier : opération *map* pour traiter des tâches en parallèle, *reduce* pour récupérer le résultat

Simple car **pas de communications** entre les nœuds

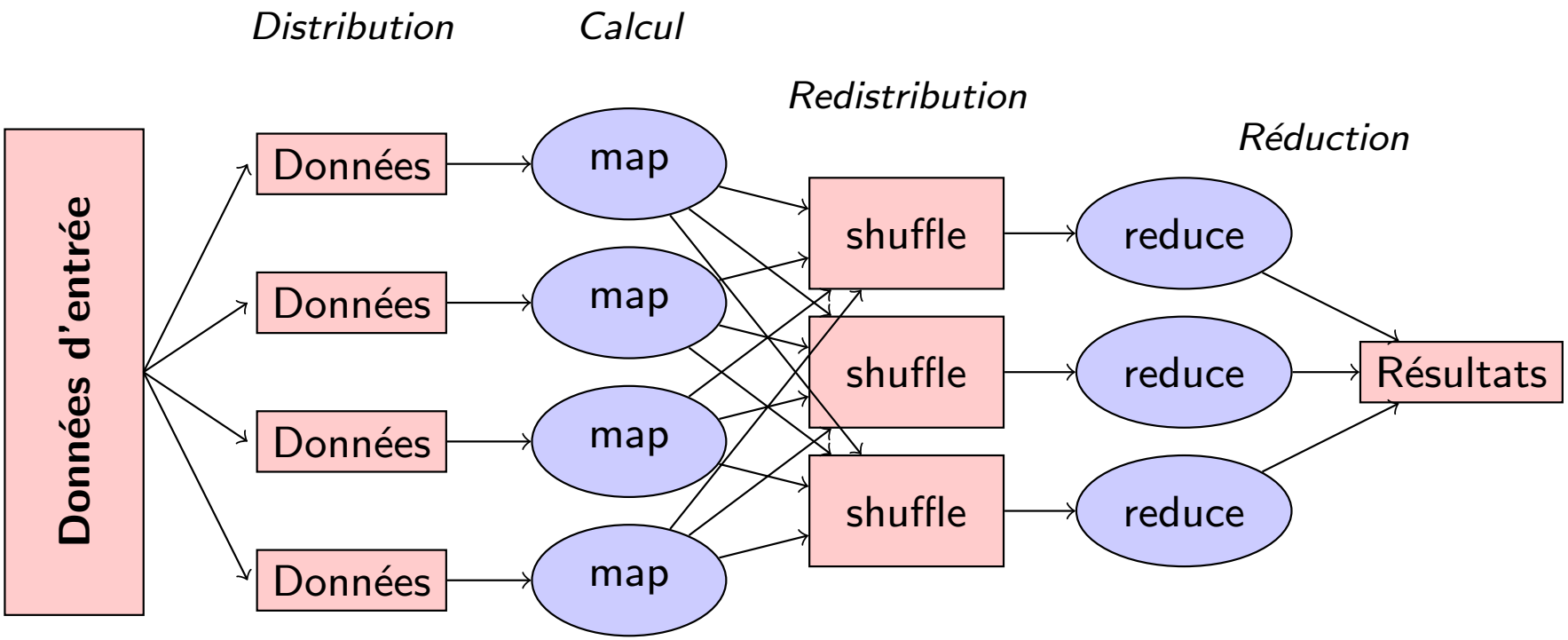
- Besoin d'un coordinateur qui distribue les tâches
- ... et qui récupère les résultats à la fin.

Seules communications : entre ce coordinateur et les nœuds de calcul, puis entre les nœuds de calcul et le coordinateur.

Le cas de MapReduce

But de **MapReduce** :

- Traiter des **gros volumes** de données
- Pas nécessairement faire du gros calcul parallèle !
- Orienté *big data*, *data mining*...
- Grosse phase de communication entre les nœuds entre le *map* et le *reduce*



Conclusion

Modèles de mémoire :

- Distribuée → communications explicites par passage de messages (MPI, OpenSHMEM)
- Partagée distribuée → espace d'adressage global, assistance du compilateur (langages PGAS)

Schémas de communications :

- Coopération des deux processus → communications bilatérales (MPI)
- Accès distant direct → communications unilatérales (OpenSHMEM, UPC)
- Aucune entre les processus → sac de tâches

Données du problème :

- Régulières → OpenSHMEM
- Irrégulières → MPI, UPC
- Très grosses → MapReduce