Grilles de Calcul et Cloud

– M2 PLS –

Camille Coti
camille.coti@lipn.univ-paris13.fr

Institut Galilée, Université Paris XIII

Plan du cours

Plan du cours

Organisation du module

Cours-TP

- En général : 1 cours / 1 TP
- TP complémentaire du cours

Les TP

- Beaucoup de lecture dans les TP, bouts de code fournis
- Langage au choix
  - 1ere partie : Python de préférence
  - 2ème partie : C, C++ ou Python possible
- À terminer en autonomie

Plan du cours

Plan du cours

Qu'est-ce qu'un système distribué ?

Définition

Système exécuté par un ensemble de **ressources autonomes**

- reliées par un réseau
- qui fonctionnent ensemble, de manière collaborative

Andrew Tanenbaum :

- *ensemble d'ordinateurs indépendants qui apparaîssent à l'utilisateur comme un seul ordinateur.*

Leslie Lamport :

- *système dans lequel un ordinateur dont vous n'avez jamais entendu parler et qui tombe en panne rend le votre inutilisable.*

Mise en place

Ressources autonomes

- Potentiellement hétérogènes
- Chacun a son système, pourrait fonctionner de façon indépendante

Couplage :

- **Fortement couplé :** interactions fréquentes entre les ressources
- **Faiblement couplé :** peu d'interactions entre les ressources

Utilisation d'un **middleware**

- Logiciel
- Orchestre l'utilisation de ces ressources
- Donne l'impression d'utiliser une seule ressource

Nécessités des systèmes distribués

Possibilité d' **extension** facile

- Ajout de ressources dans le système
- **Passage à l'échelle**

Continuité de service

- Même si des parties du système sont hors service (panne, déconnexion...)
- Résilience
- Idéalement : l'utilisateur ne s'en rend même pas compte

Ouverture , normalisation

- Permet l'interopérabilité de ressources différentes
- Utilisation d'une interface bien définie

Nécessités des systèmes distribués

D'après A. Tanenbaum :

1. Aucune machine n'a l'information complète sur l'état du système
2. Les machines prennent des décisions en e basant uniquement sur l'algorithme local
3. La défaillance d'une machine ne cause pas l'échec de l'algorithme
4. Pas d'hypothèse implicite sur l'existence d'une horloge globale

Exemples de systèmes distribués

Le **Web**

- Au début : document distribués
- Maintenant : applications distribuées
- Serveur Web / navigateurs ou clients
- Clients légers d'applications, applications dans le navigateur
- YouTube YOUPORN Pornhub xHamster

Système de fichiers en réseau (NFS et consorts)

- Client : sur le poste client
- Serveur : effectue le stockage "pour de vrai"

Routage réseau

- Algorithmes de routage : distribué
- Tables de routage distribuées

Évolution historique (1)

Années 60-70 : les **mainframes**

- un ordinateur central
- terminaux légers
- réseau pour relier les terminaux au mainframe
- temps partagé sur le mainframe

Évolution historique (2)

Années 2010 : le **cloud**

- Externalisation des ressources
- Stockage “dans le nuage”
- Services fournis par le cloud (cloud applicatif)
- Clients légers : tablettes, netbooks...

Évolution historique (2)

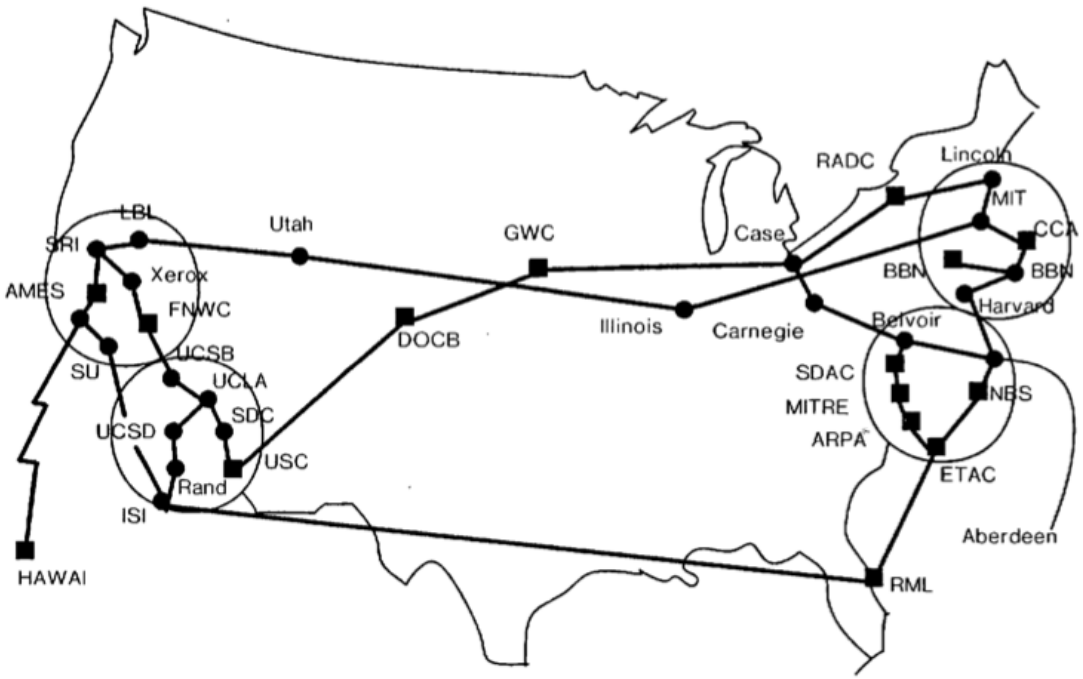
Années 2010 : le **cloud**

- Externalisation des ressources
- Stockage “dans le nuage”
- Services fournis par le cloud (cloud applicatif)
- Clients légers : tablettes, netbooks...

Évolution historique (3)

Entre les deux :

- Création d'Internet : but de décentraliser
- Systèmes de fichiers en réseau, annuaires
- Invention d'Internet, du WWW
- ...



ARPA NETWORK
(may 1973)

Caractéristiques des systèmes distribués

Connaissances des processus

- *Propriétés locales*
  - identité des voisins...
- Sa propre identité *si système identifié* (pas anonyme)
- Variables locales

Pas de temps global

- Heure = horloge locale, pas de vision sur l'horloge des autres
- Utilisation d' **horloges logiques**

Vision du système partielle : état local

- Tâche réalisée : propriété **globale** en utilisant la connaissance **locale** et la vision de chaque processus.
- Communications entre processus via des canaux de communications

État d'un système distribué = ensemble des états des processus + états des canaux de communications

Communications entre les processus pour coopérer

- Asynchrones
- Sauf systèmes spécifiques (temps réel...)
- Défaillances possibles
- Temps de communications non-nuls

Complexité : complexité locale (en mémoire, étapes, opérations...) MAIS AUSSI communications (nombre de messages, taille)

Intérêt des systèmes distribués

Fournir un service

- Web !
- Candy Crush
- email, MMORPG

Partager des ressources

- cloud
- NFS
- annuaire

Répartir la charge

- calcul parallèle distribué
- serveurs distribués avec load balancing

Répartition géographique

- Serveurs OVH à Roubaix

Non-centralisaton

- création d'Internet
- fiabilité

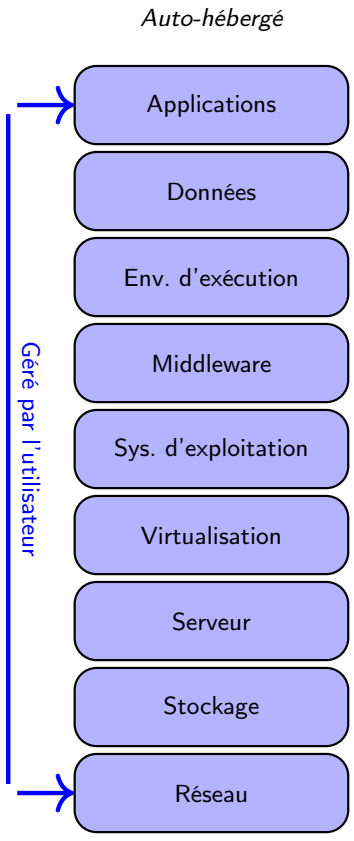
Plan du cours

Types de cloud

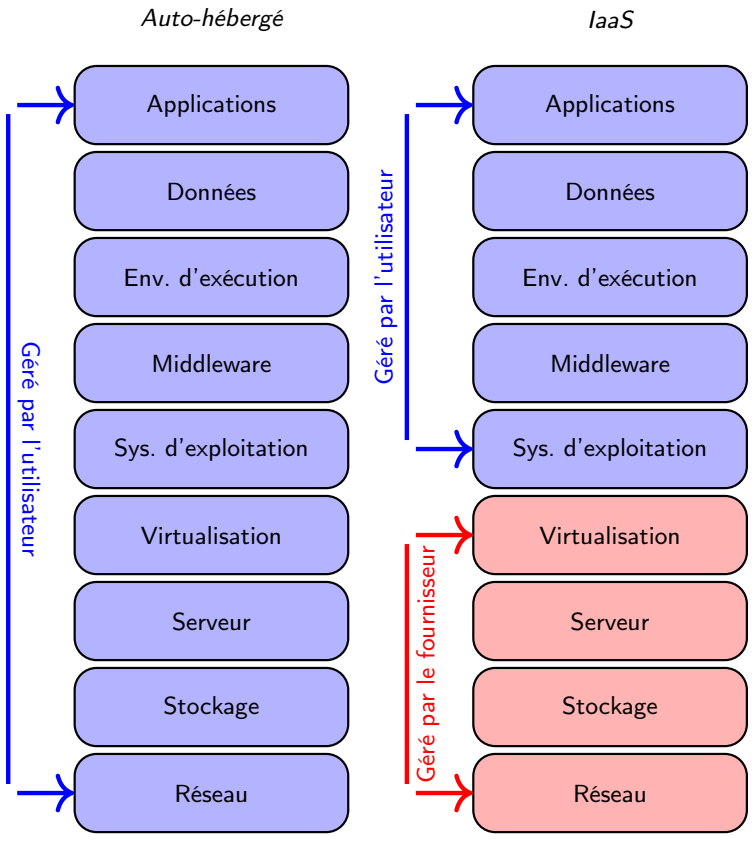
Le niveau de service dépend du type de cloud.

- Cas extrême (non cloud) : **machine auto-hébergée** . L'utilisateur s'occupe de tout.
- **Infrastructure as a Service** : le prestataire fournit l'*infrastructure matérielle*, le client exécute son propre système sur un système de virtualisation. Exemple : Amazon EC2
- **Platform as a Service** : le prestataire fournit l'*environnement d'exécution*, le client exécute son application et stocke ses données sur le cloud. Exemple : hébergement Web.
- **Service as a Service** : le prestataire fournit un *service* préinstallé : blog, application... Le client ne fait que l'exécuter. Exemple : Google Docs, interface Gmail...

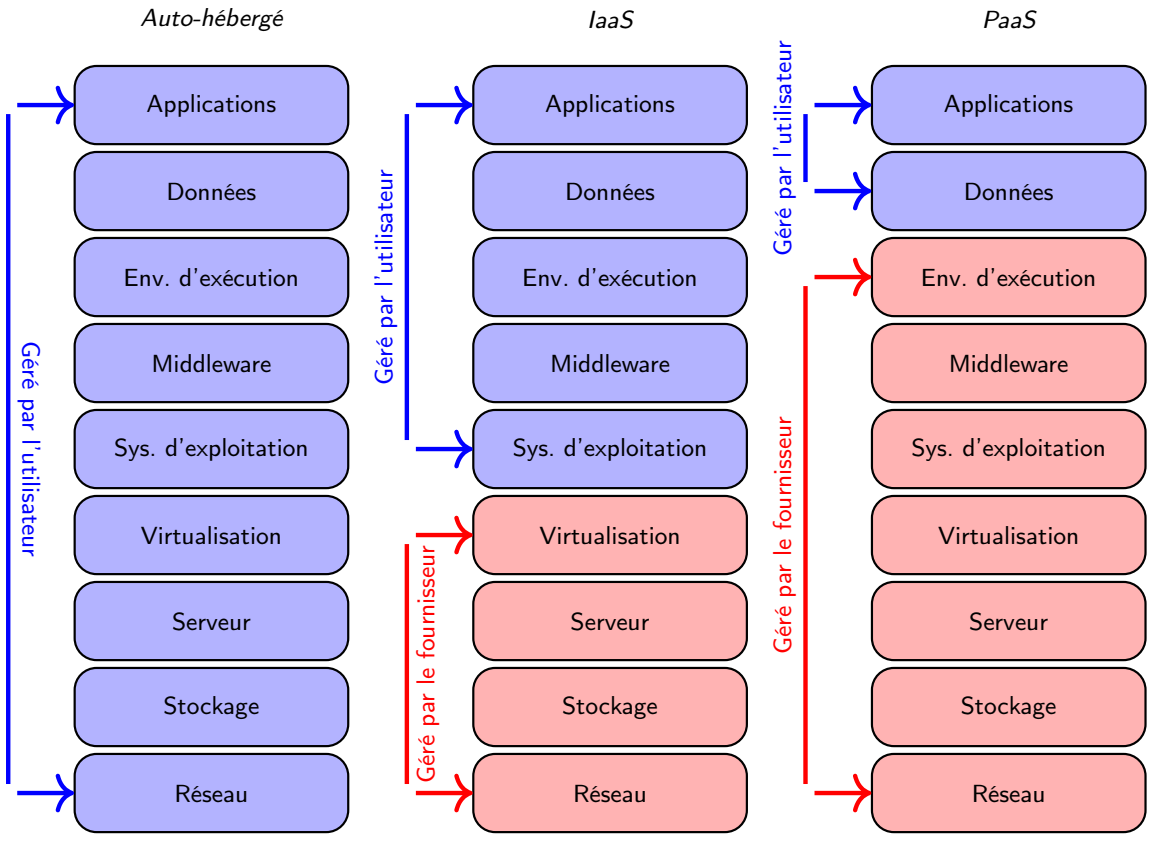
Types de cloud



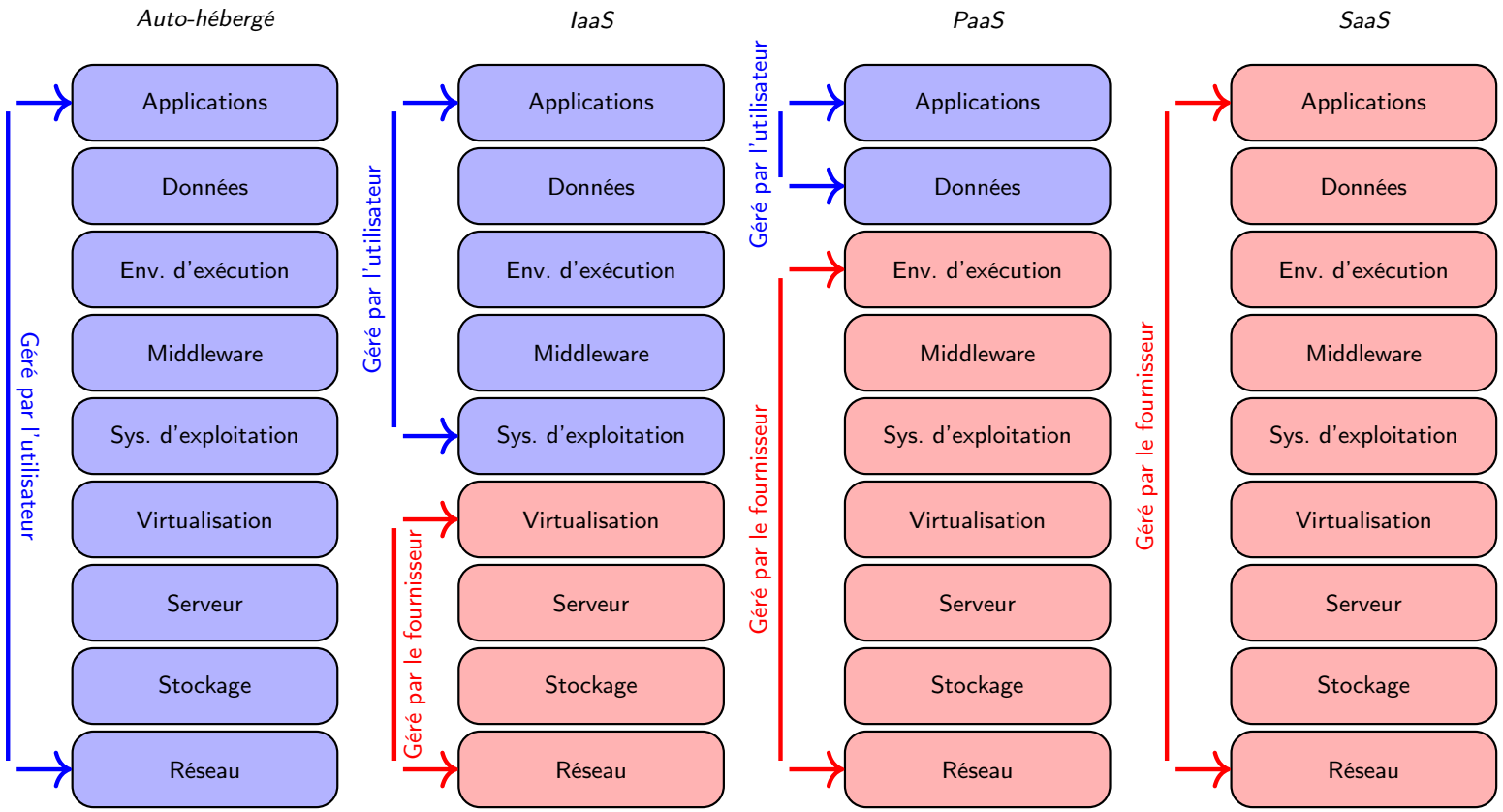
Types de cloud



Types de cloud



Types de cloud



Plan du cours

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
- Choix d'une transition parmi celles possibles : faite par un **ordonnanceur**
  - L'ordonnanceur est **juste** (*fair*) si pas de raison particulière de choisir une transition plutôt qu'une autre
  - Sinon : modèle probabiliste, etc.

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.

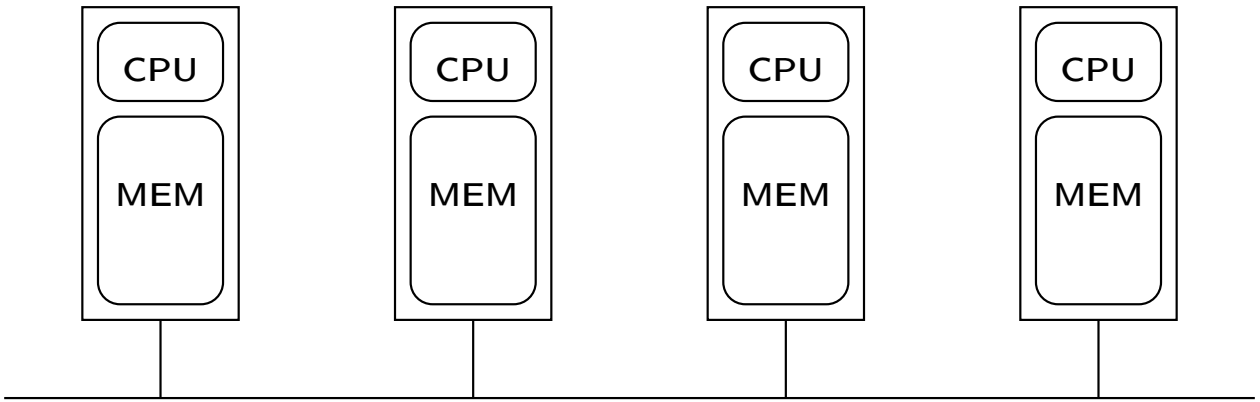
Mémoire distribuée

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

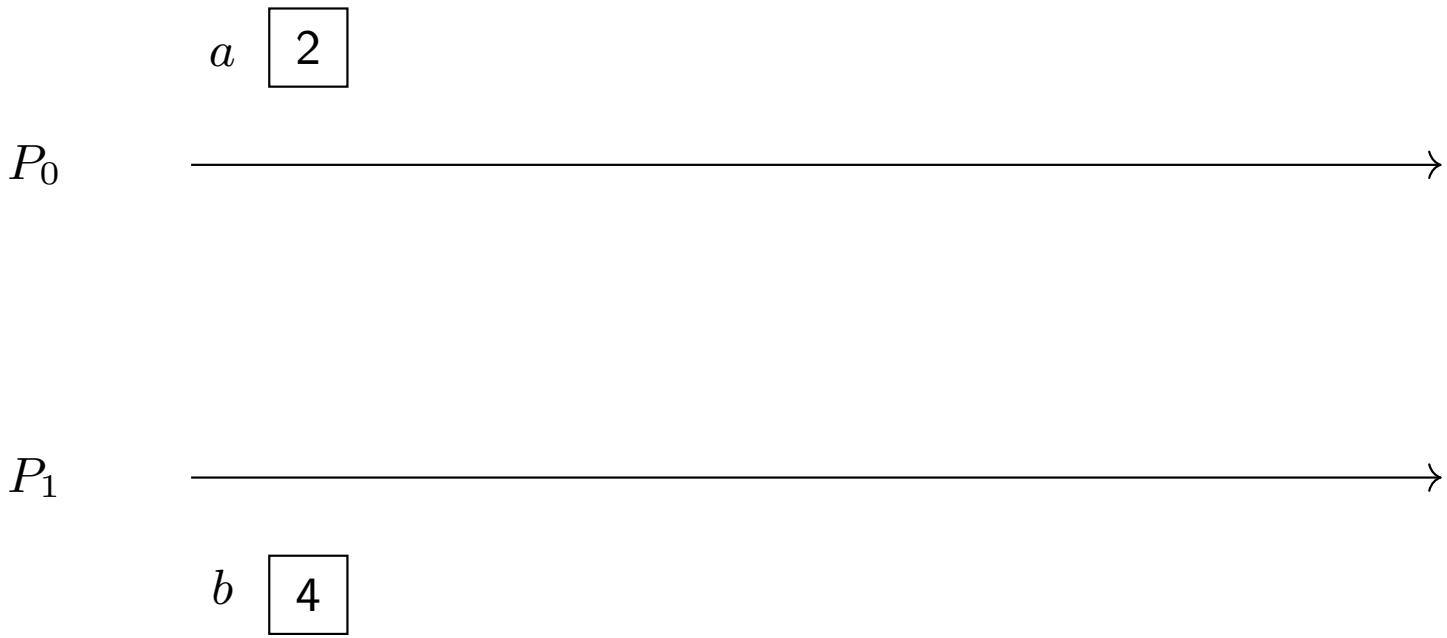


Communications bilatérales

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

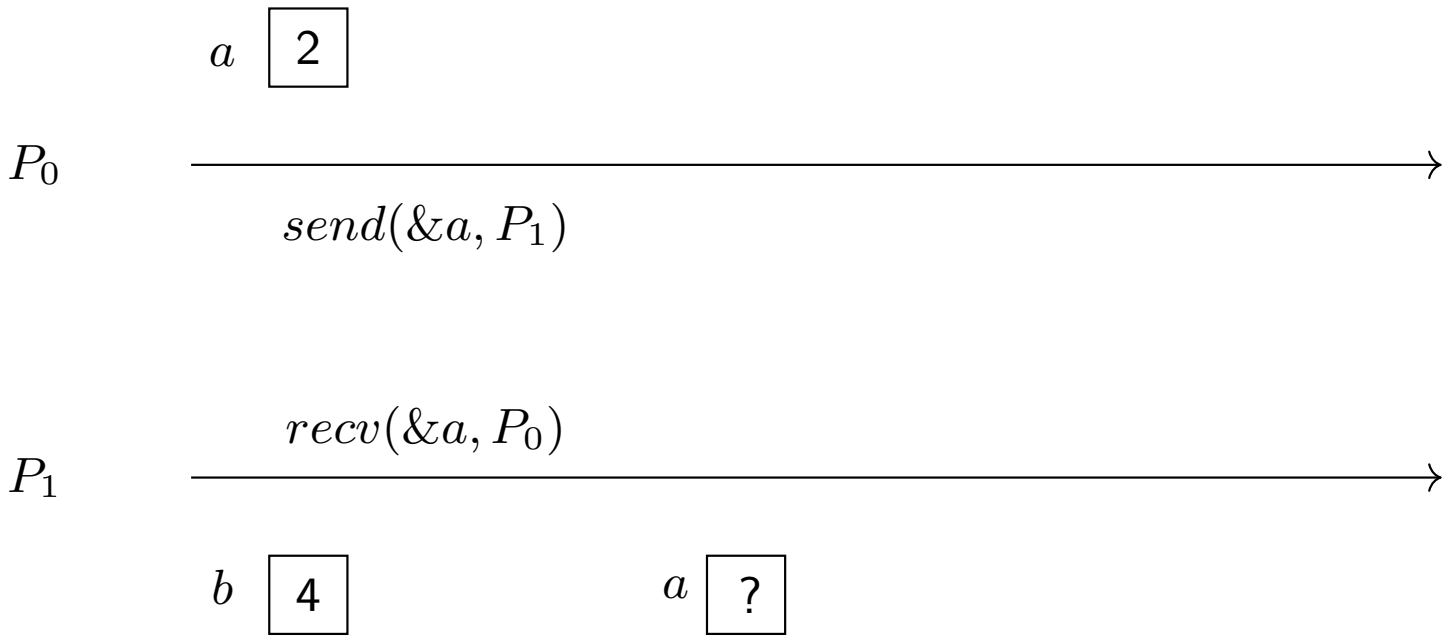


Communications bilatérales

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

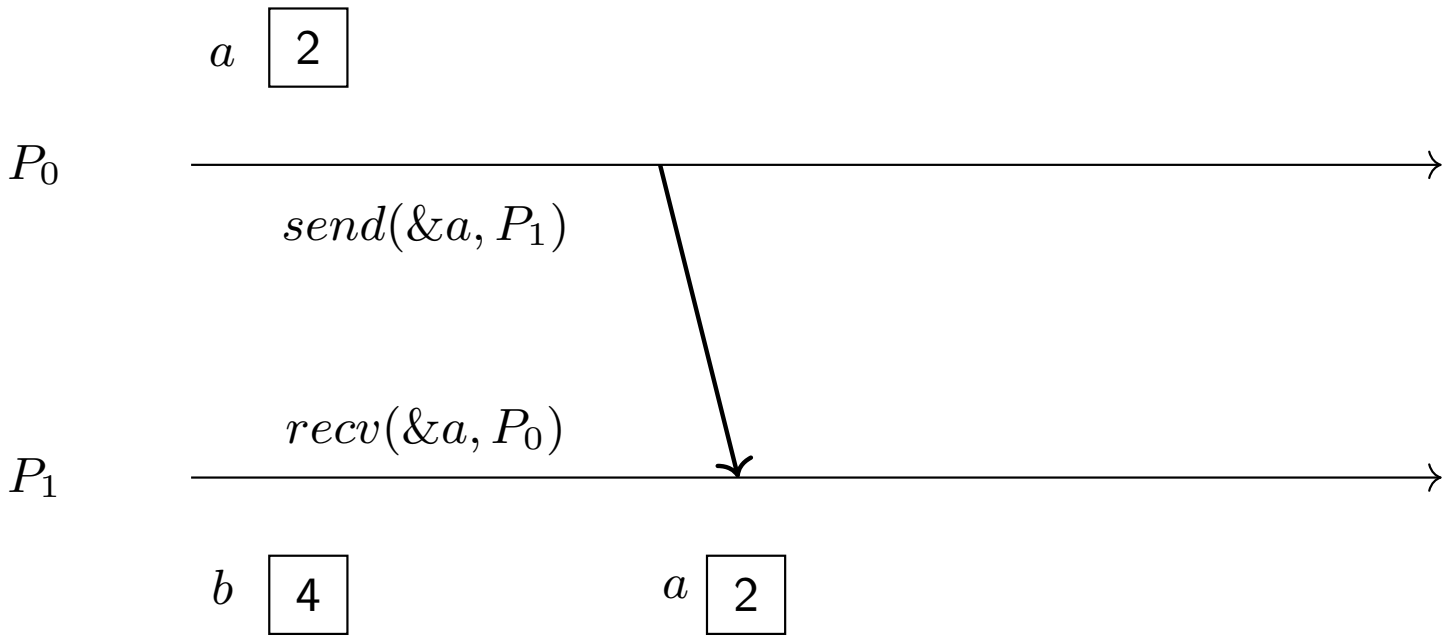


Communications bilatérales

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

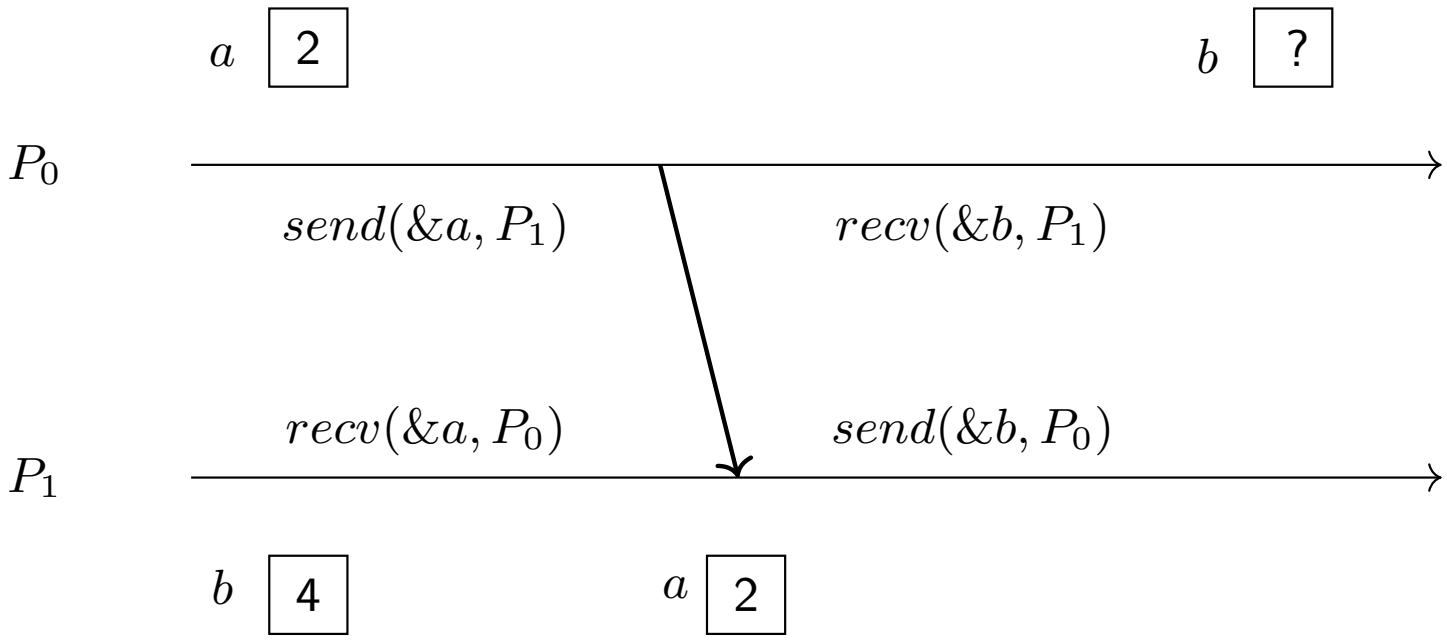


Communications bilatérales

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

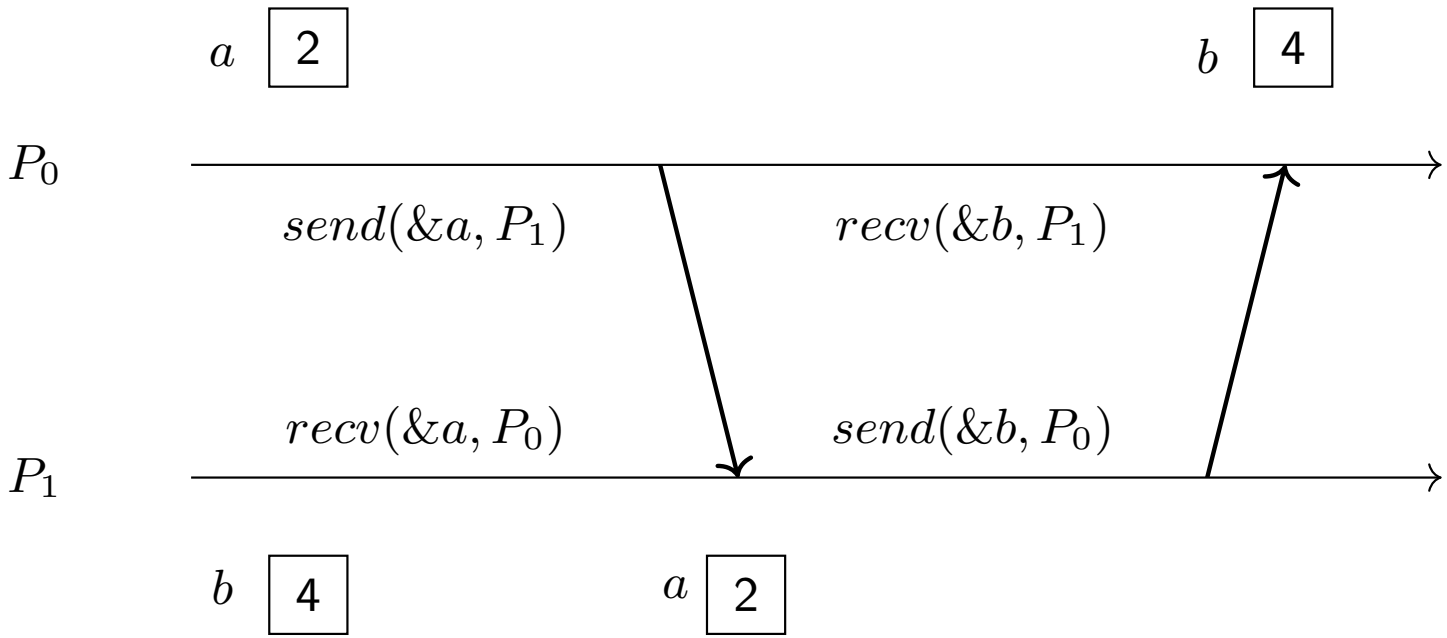


Communications bilatérales

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 :

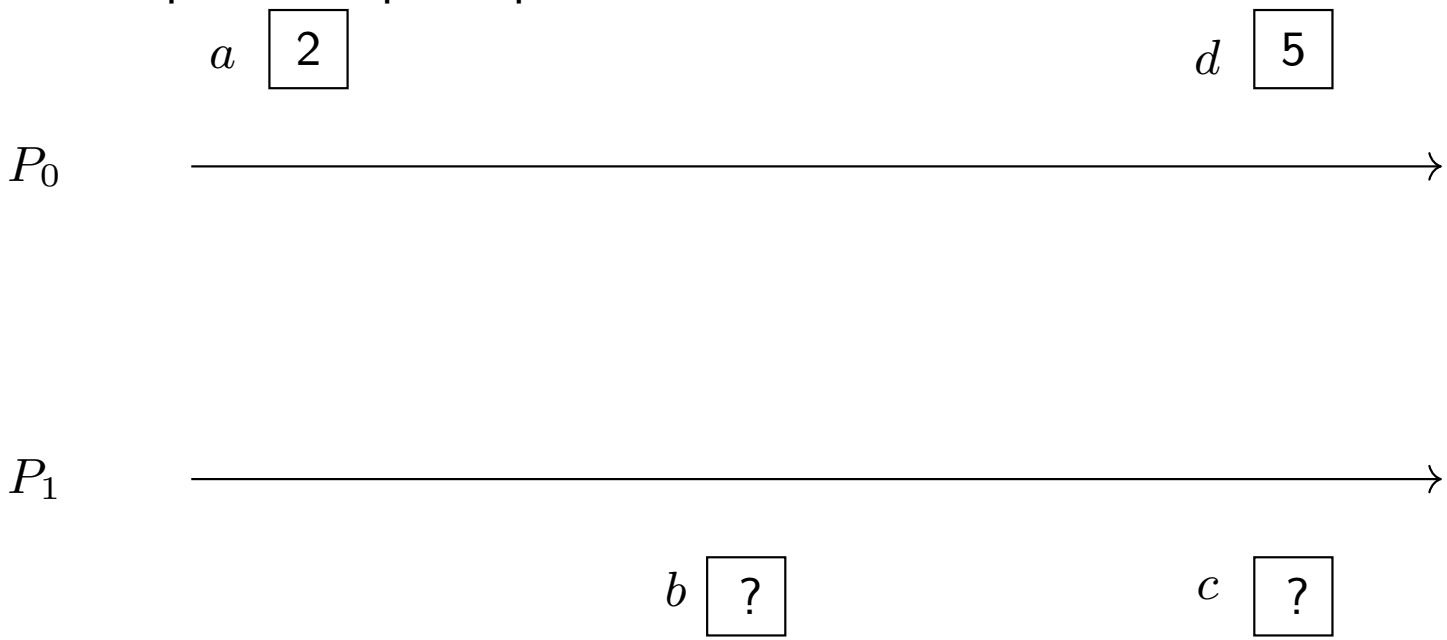
- Besoin de la coopération des deux processus : source et destination
- Fort synchronisme

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.

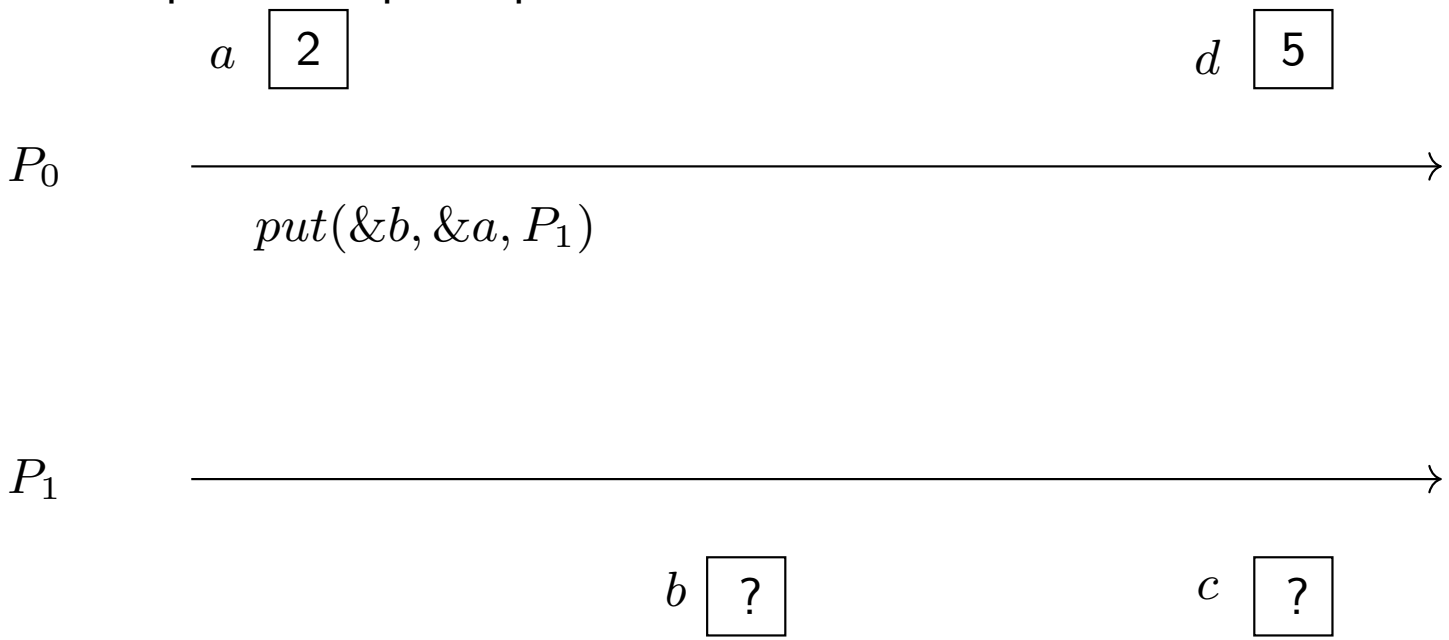


Communications unilatérales

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.

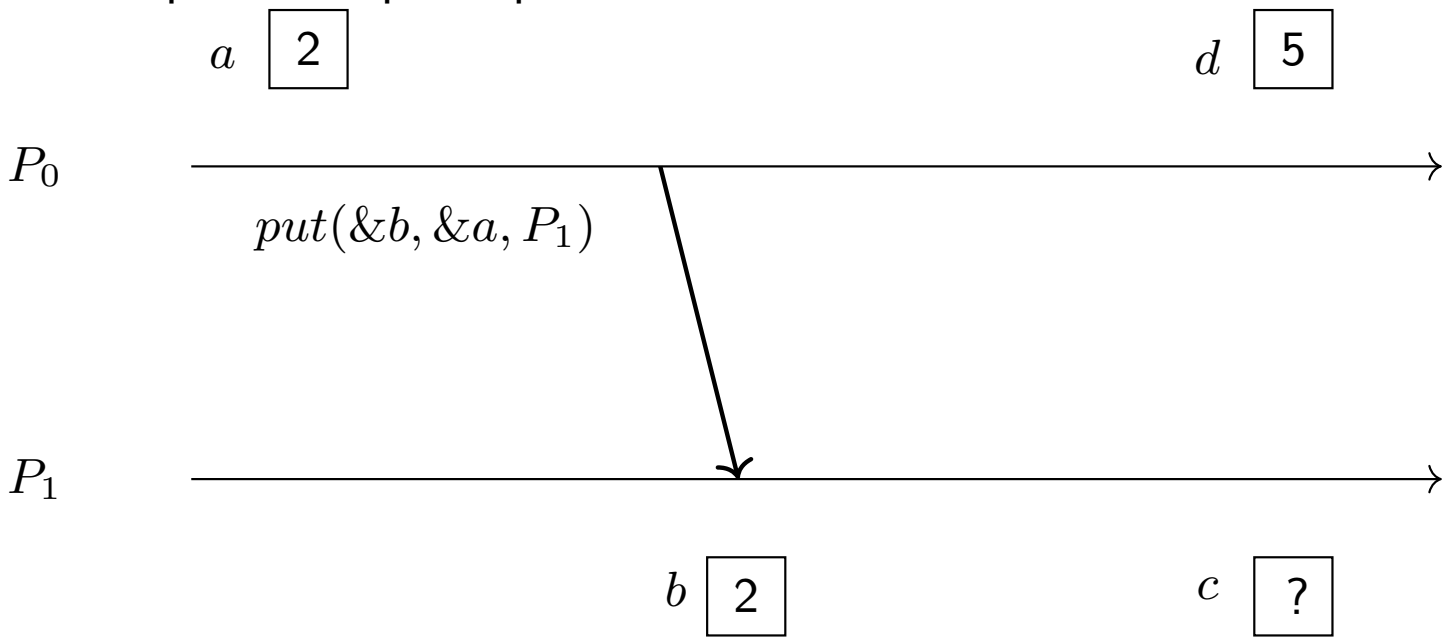


Communications unilatérales

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.

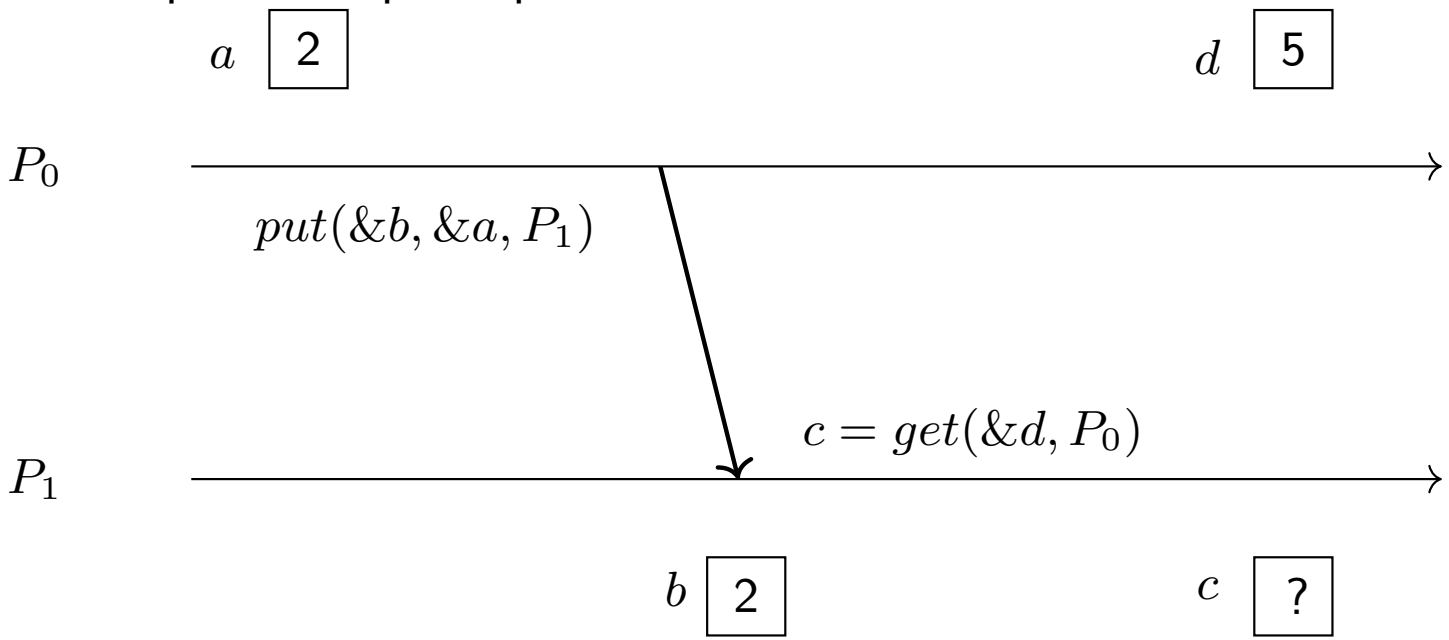


Communications unilatérales

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.

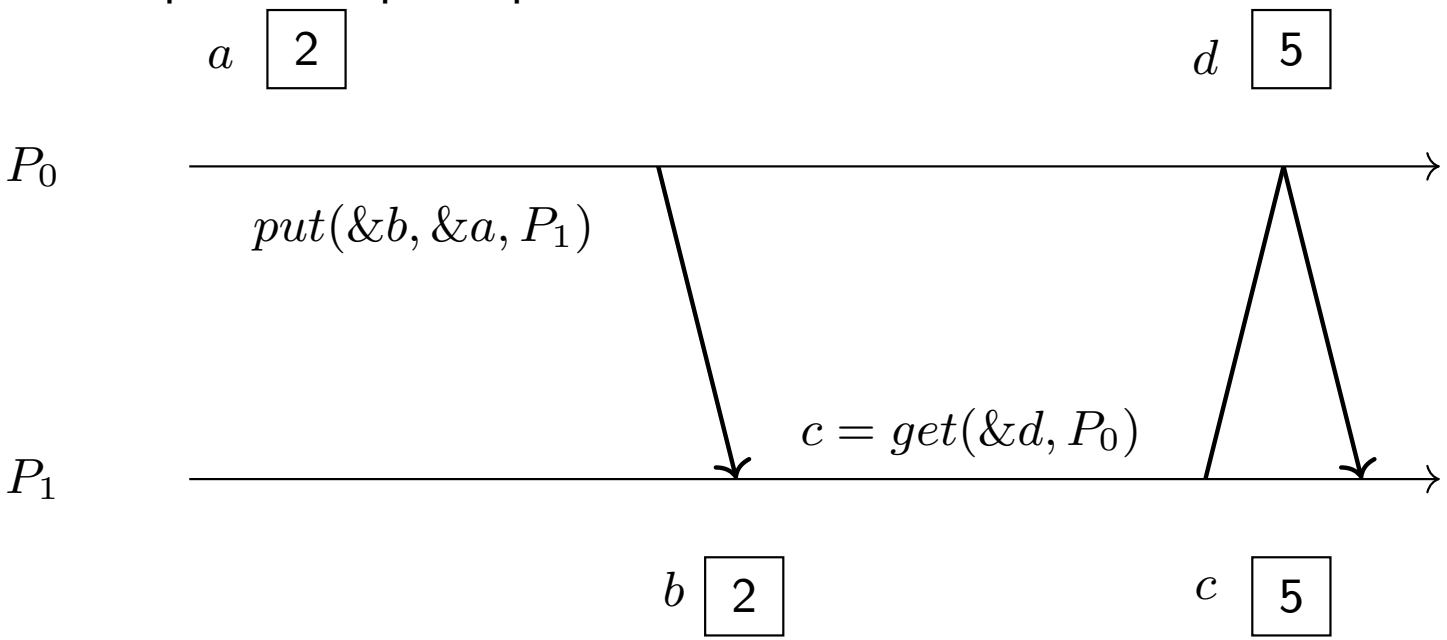


Communications unilatérales

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

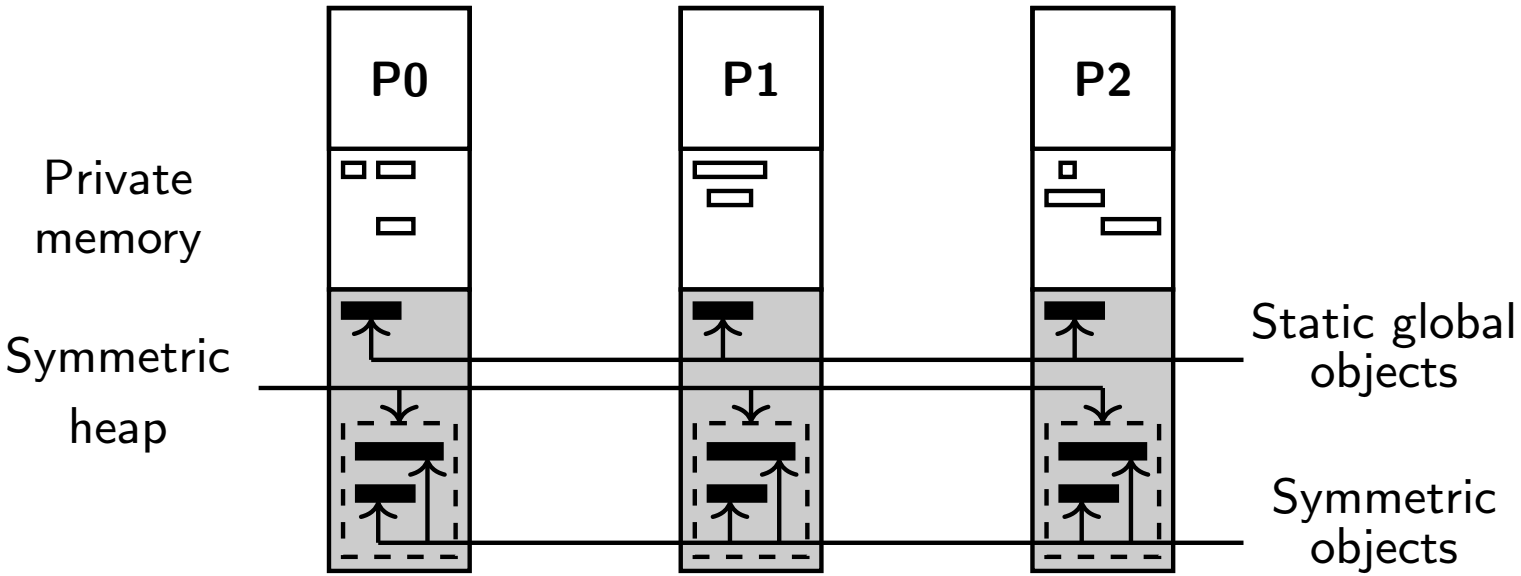
Inconvénients :

- 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*



Plan du cours

Plan du cours

Temps dans un système distribué

Sur un système centralisé

- Horloge centrale, unique
- Consultée quand on a besoin de l'heure : `gettimeofday()`, `clock_gettime()`...

Dans un système distribué : **pas d'horloge globale !**

- Pas de synchronisation
- Une heure globale a-t-elle un sens ?
- L'heure qu'il est ici vs l'heure à l'autre bout de la pièce

Chaque processus ne connaît que son état **local**

Propriétés des canaux de communications

Canal de communication entre les processus

- Uni ou **bidirectionnel**
- **Fiable** ou non
  - **Erreurs** : perte, duplication, création, erreur
- Modifie ou non l'ordre de réception
- Synchrone ou **asynchrone**
- **Tampons** d'envois et de réception : taille bornée ou infinie

Asynchrone : la communication se fait en un temps **fini non borné** .

Canal **FIFO** (*First In, First Out*) : les messages sont reçus dans l'ordre dans lequel ils ont été envoyés.

→ Comment avoir une **notion de temps** dans ces conditions ?

Notion de temps

Deux processus : P_0 et P_1

- P_0 effectue une action (affichage, envoi d'un message...)
- P_1 effectue aussi une action
- Comment savoir lequel l'a fait avant ?

Notion de temps

Deux processus : P_0 et P_1

- P_0 effectue une action (affichage, envoi d'un message...)
- P_1 effectue aussi une action
- Comment savoir lequel l'a fait avant ?

... on ne peut pas.

Plan du cours

Précédence causale

Dans un système distribué, un **évènement** est :

- Une émission de message
- Une réception de message
- Un calcul interne

Relation d'ordre entre ces évènements ?

Précédence causale

Dans un système distribué, un **évènement** est :

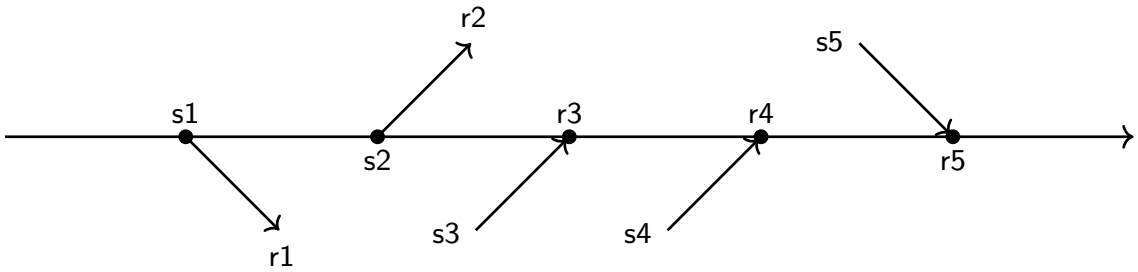
- Une émission de message
- Une réception de message
- Un calcul interne

Relation d'ordre entre ces évènements ?
Prenons la transmission d'un message :

- L'envoi précède *toujours* la réception

Si après une réception je fais un calcul sur le message que je viens de recevoir :

- La réception précède le calcul



Précédence causale

Dans un système distribué, un **évènement** est :

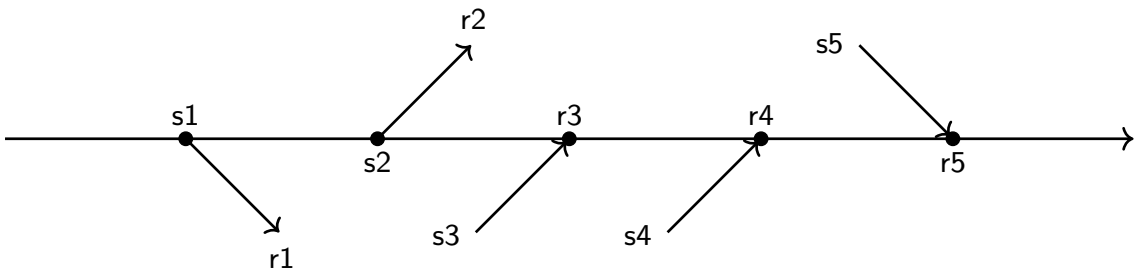
- Une émission de message
- Une réception de message
- Un calcul interne

Relation d'ordre entre ces évènements ?
Prenons la transmission d'un message :

- L'envoi précède *toujours* la réception

Si après une réception je fais un calcul sur le message que je viens de recevoir :

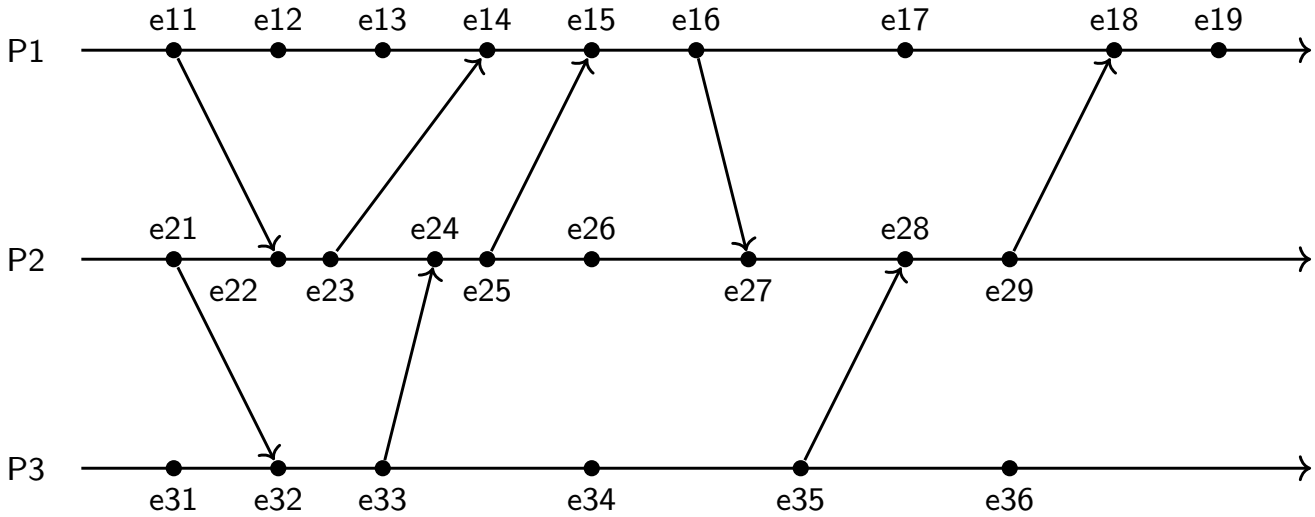
- La réception précède le calcul



Relation entre deux évènements : $\prec$

- Relation de **précédence causale**
- Appelée *happens before* (Lamport)
- Définit un ordre partiel entre les évènements

Précédence causale : exemple



Ordre causal

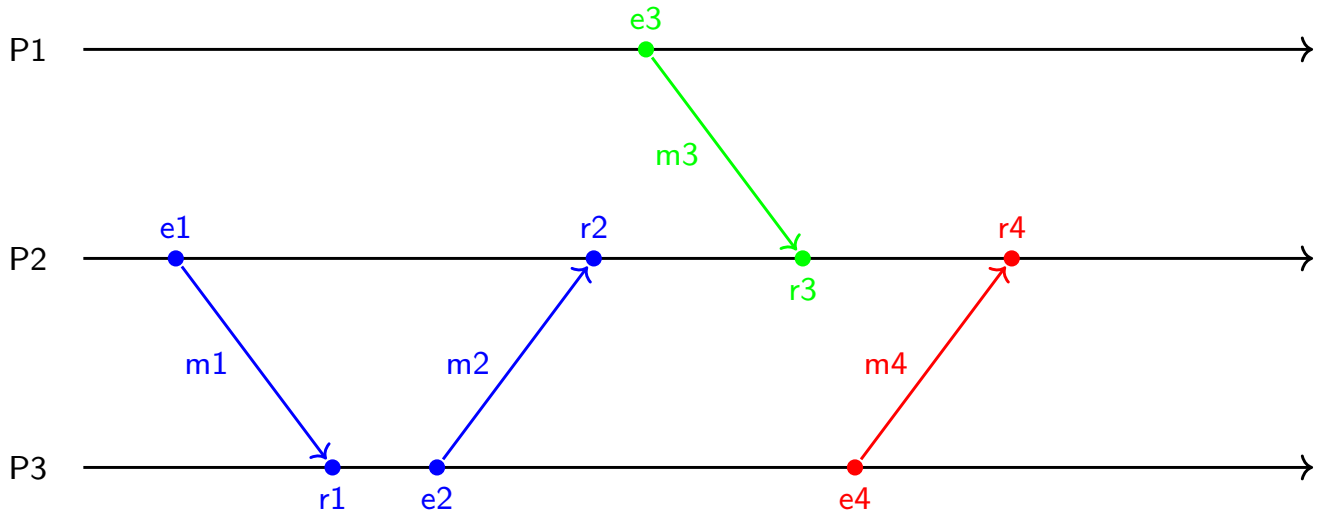
Soit e et e' deux évènements. On a $e \prec e'$ si au moins une des trois conditions suivantes est vraie :

- e et e' ont lieu sur le même processus avec e avant e'
- $e = envoyer(m)$ et $e' = recevoir(m)$ avec le même message
- Il existe un évenement e'' tel que $\prec e''$ et $e'' \prec e'$

Remarques :

- Si on ne peut pas définir de relation causale entre deux évenements : ils sont **concurrents**
  - Noté $e_1 || e_2$
- La relation d'ordre causal est **transitive**

Ordre causal : exemple



Existe-t-il un ordre causal entre

- r1 et e2 ?
- r1 et r2 ?
- r2 et r3 ?
- e2 et e3 ?
- e4 et r3 ?
- e2 et r4 ?
- e2 et r3 ?
- r2 et r4 ?

Notion de temps *logique*

Système de temps pour dater les évènements

- Donne les informations de causalité entre évènements
- Lamport

Horloge logique

- Compte les événements
- Incrémentée quand un événement arrive
  - en général : émission ou réception de message

Si $h(e_1) < h(e_2)$ alors $e_1 \prec e_2$

Signification de $<$ en pratique : dépend du type d'horloge.

Exemple : canal FIFO

- Deux processus P_0 et P_1
- Chacun a une horloge (scalaire) initialisée à 0
- À chaque envoi : le processus incrémente son horloge
- La valeur de l'horloge est incluse dans le message (*estampille*)

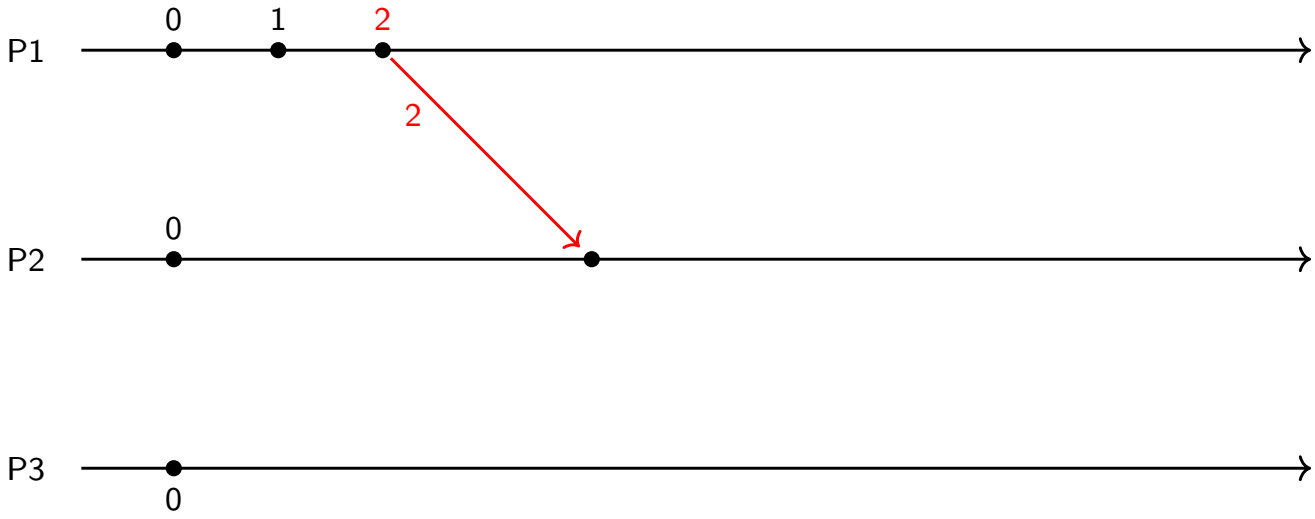
Plan du cours

Horloge de Lamport

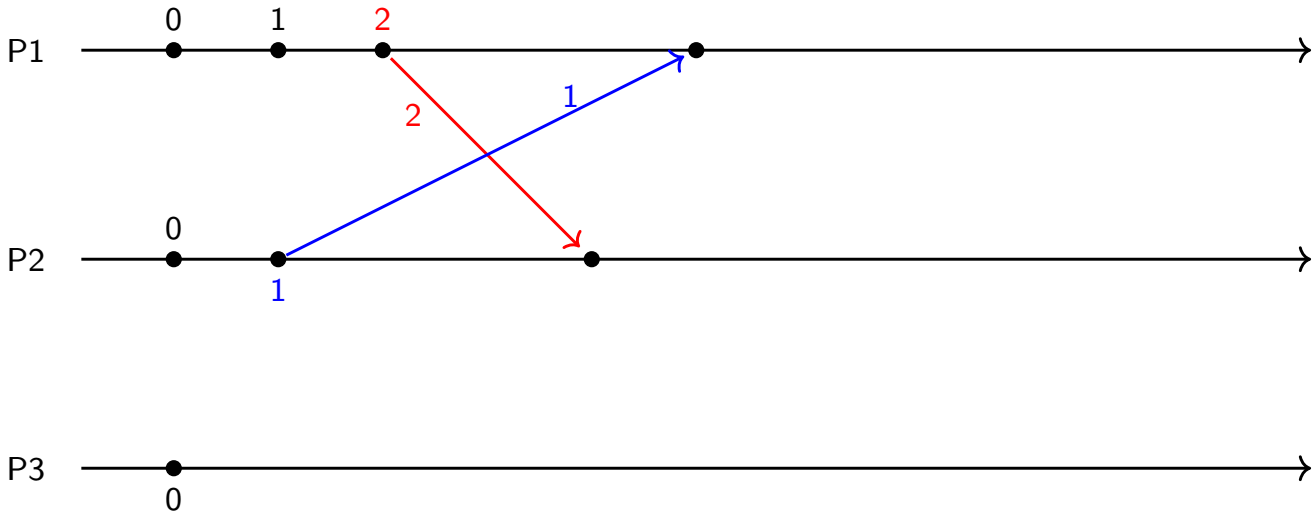
Horloge de Lamport = **horloge scalaire**

- Chacun a une horloge initialisée à 0
- À chaque événement : le processus incrémente son horloge
- La valeur de l'horloge est incluse dans le message
- En réception : le processus **cale son horloge sur l'horloge reçue**
  - $H_{locale} = max(H_{locale}, H_{reue}) + 1$
  - Prise en compte de l'évolution du reste du système depuis la dernière interaction

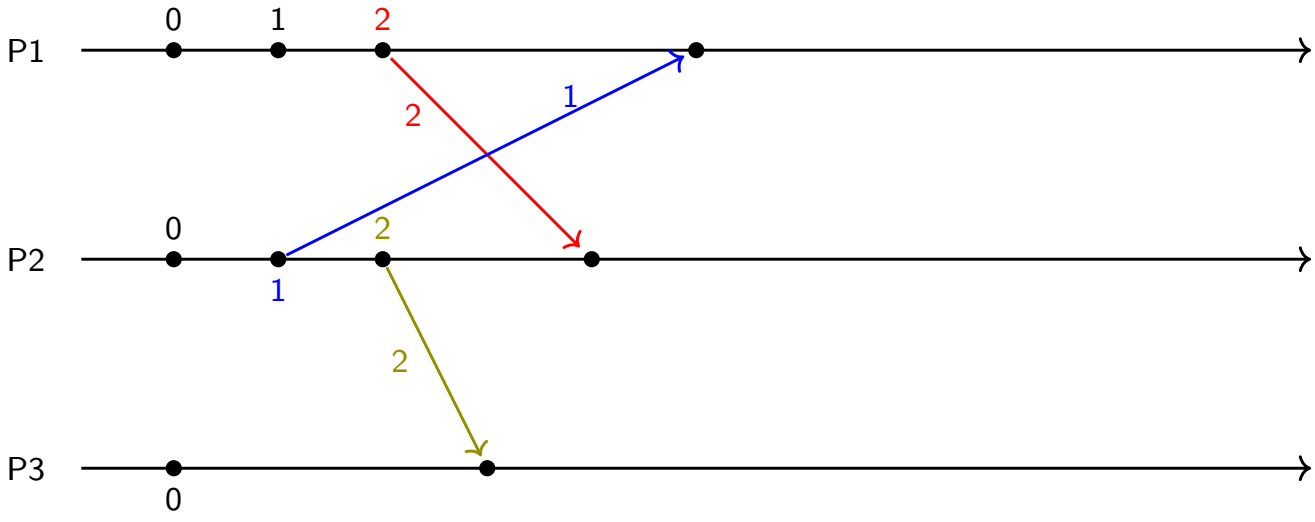
Exemple



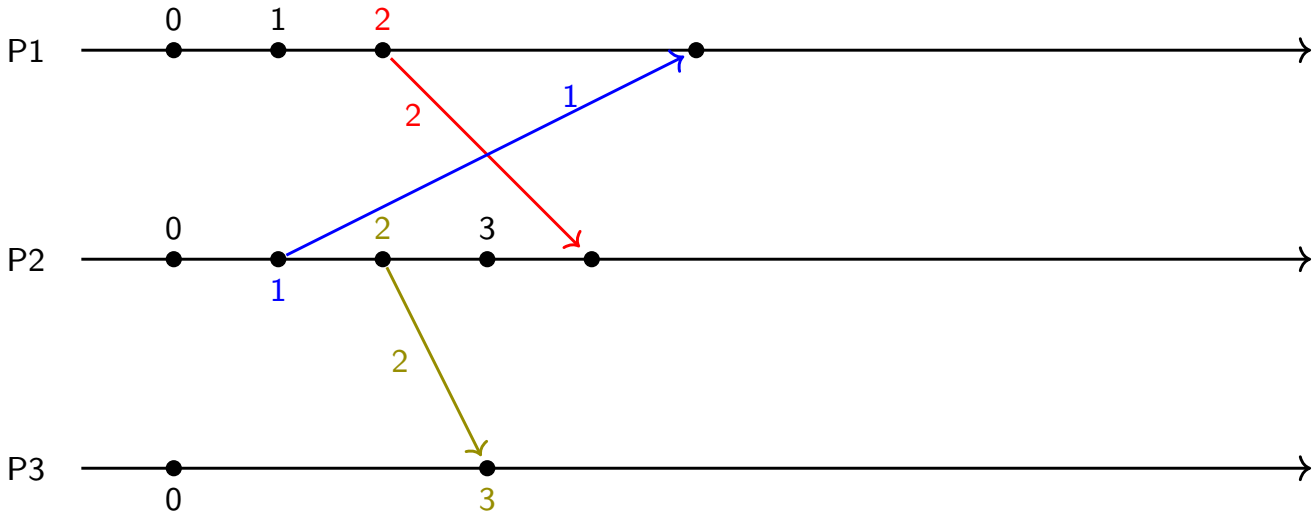
Exemple



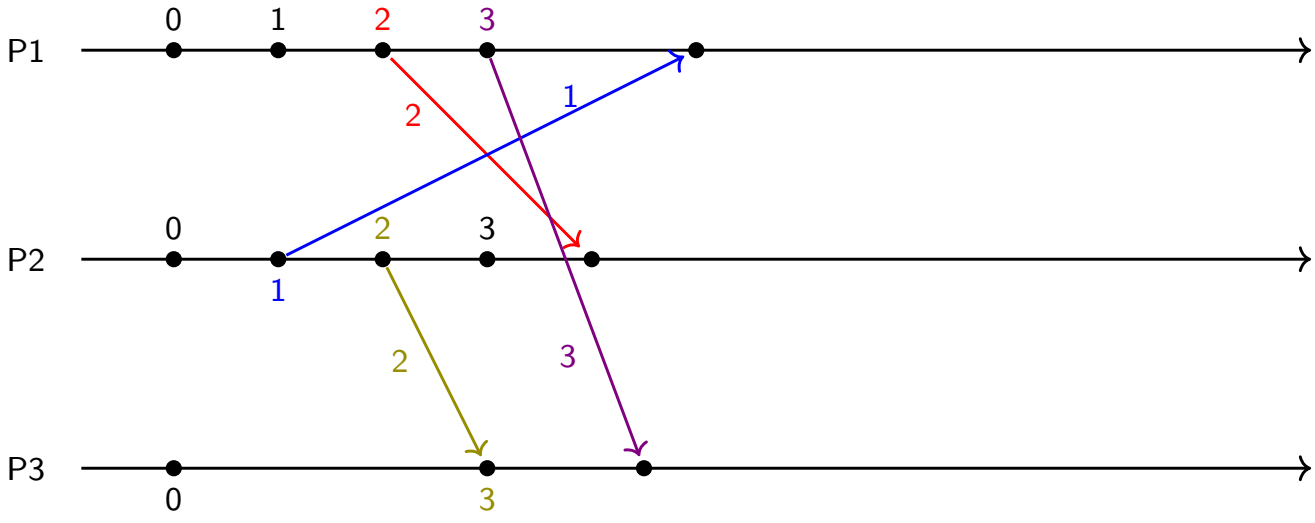
Exemple



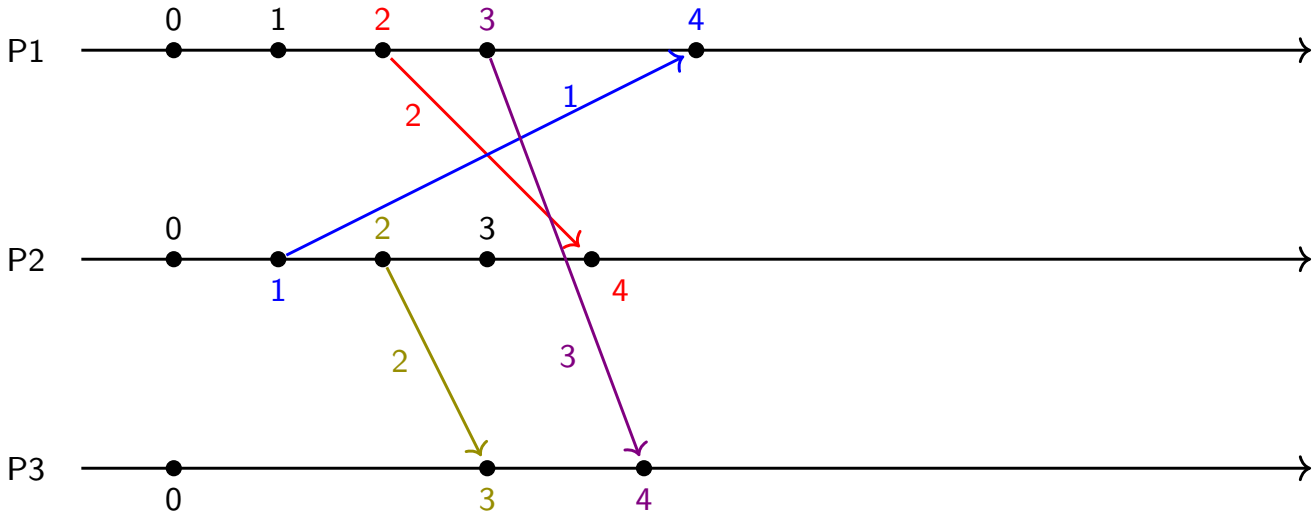
Exemple



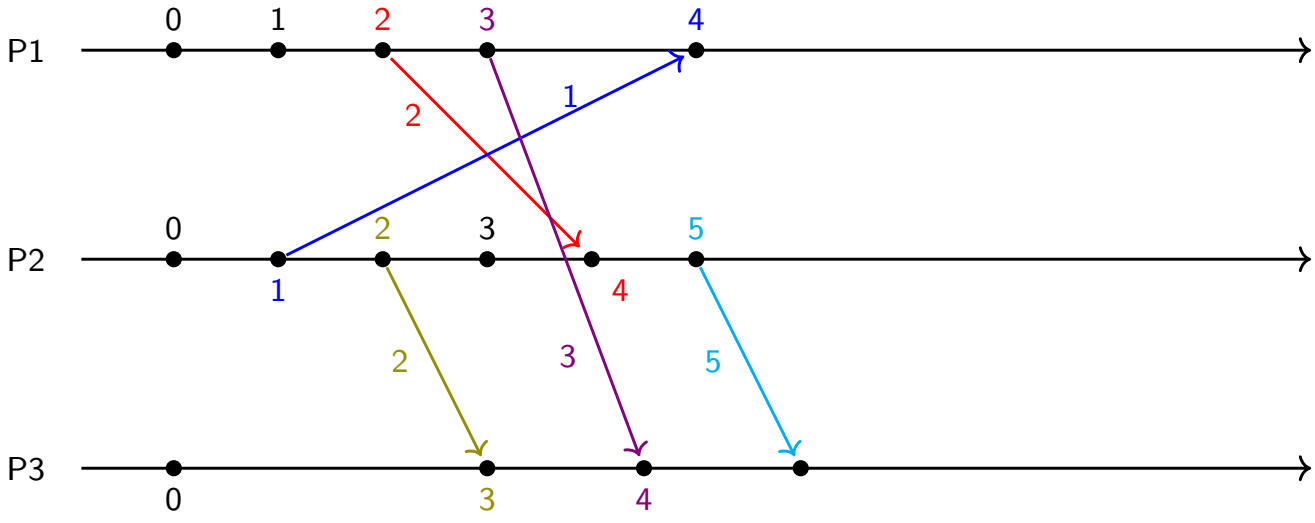
Exemple



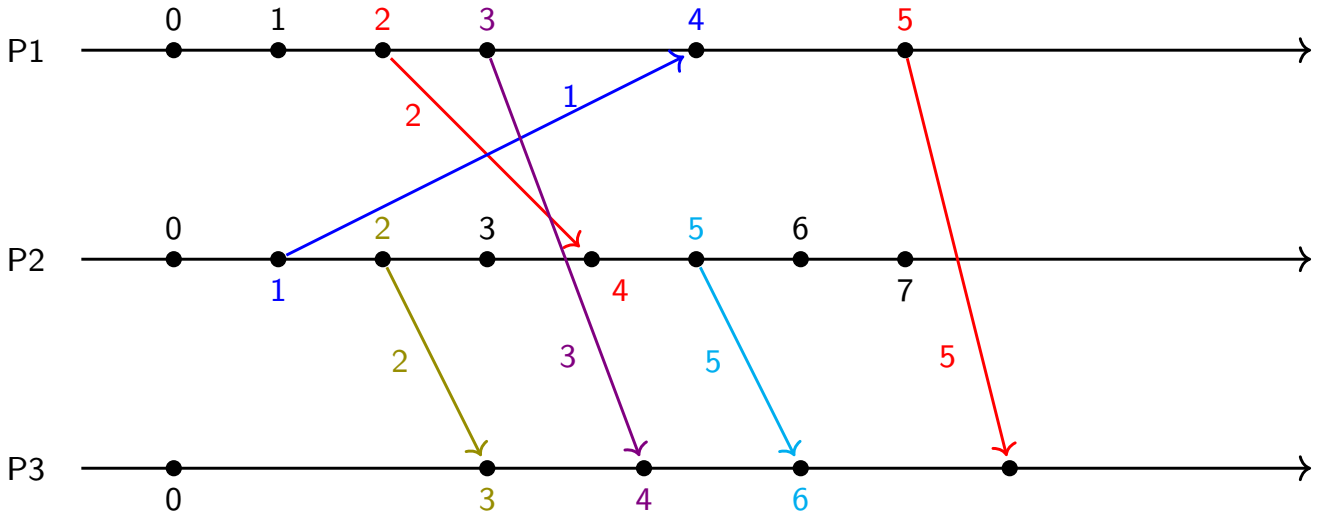
Exemple



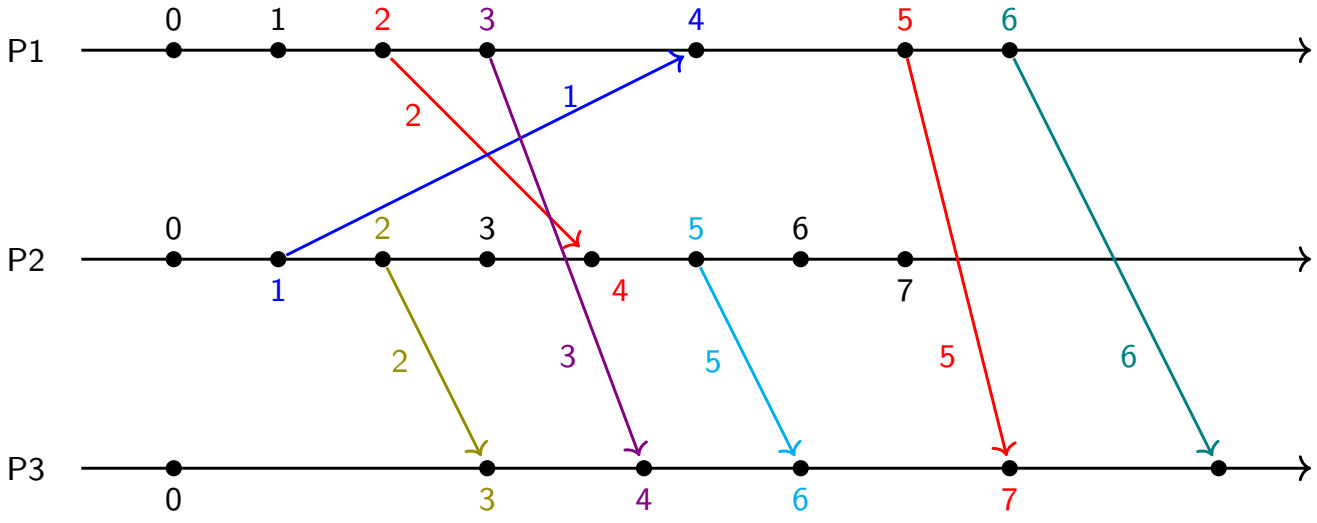
Exemple



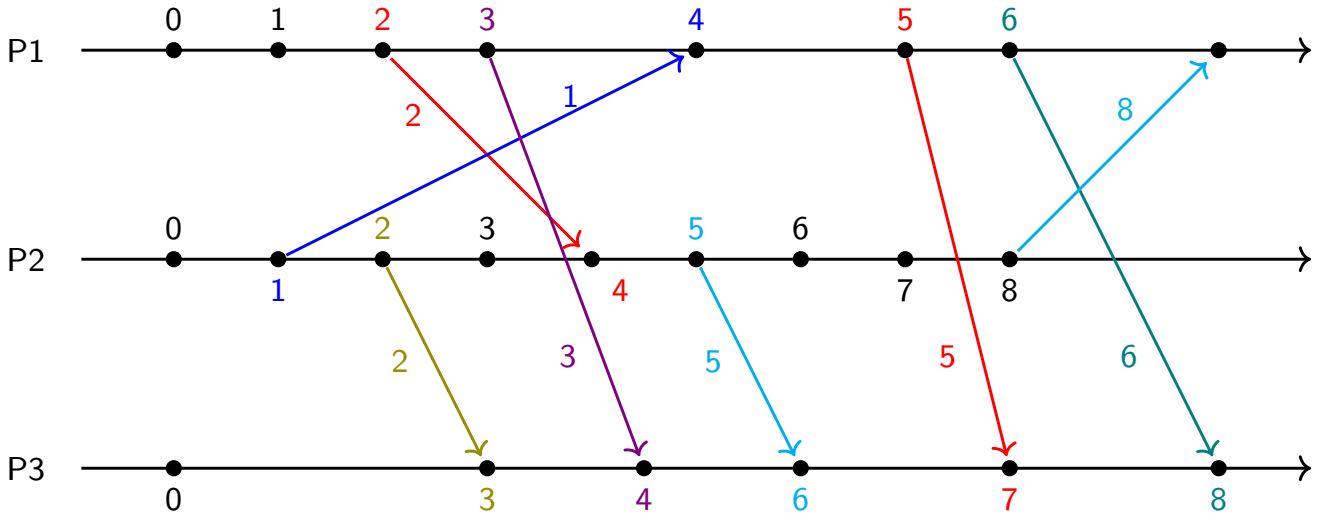
Exemple



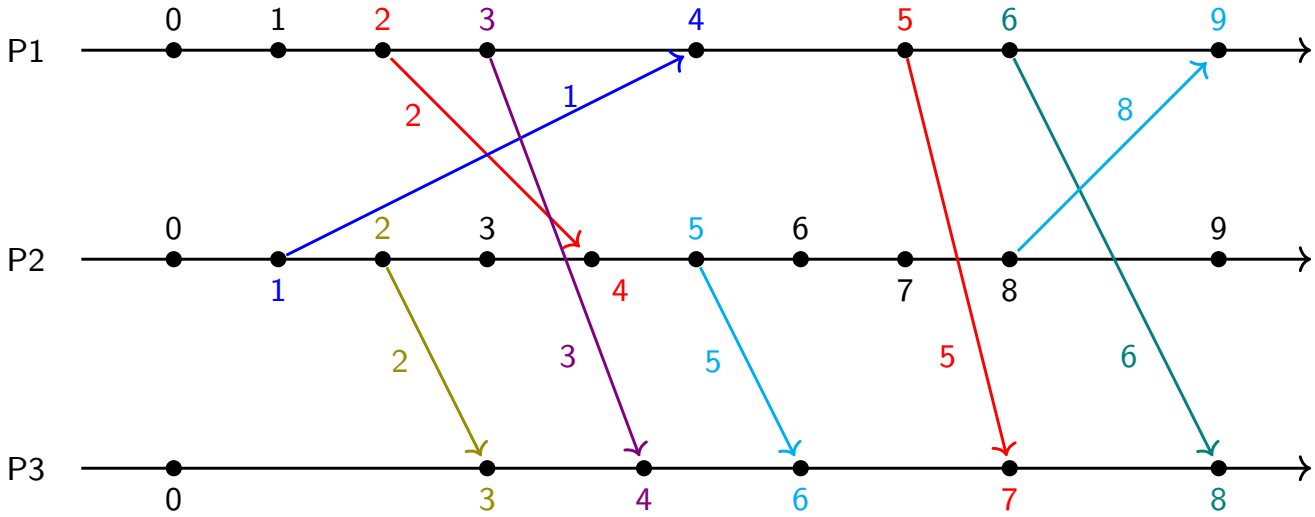
Exemple



Exemple



Exemple



Remarques sur les horloges de Lamport

Point de vue *global*

- Faisable sur un seul processus
- Plus compliqué sur plusieurs processus (pas de vision globale)

Pas deux évènements locaux avec la même horloge

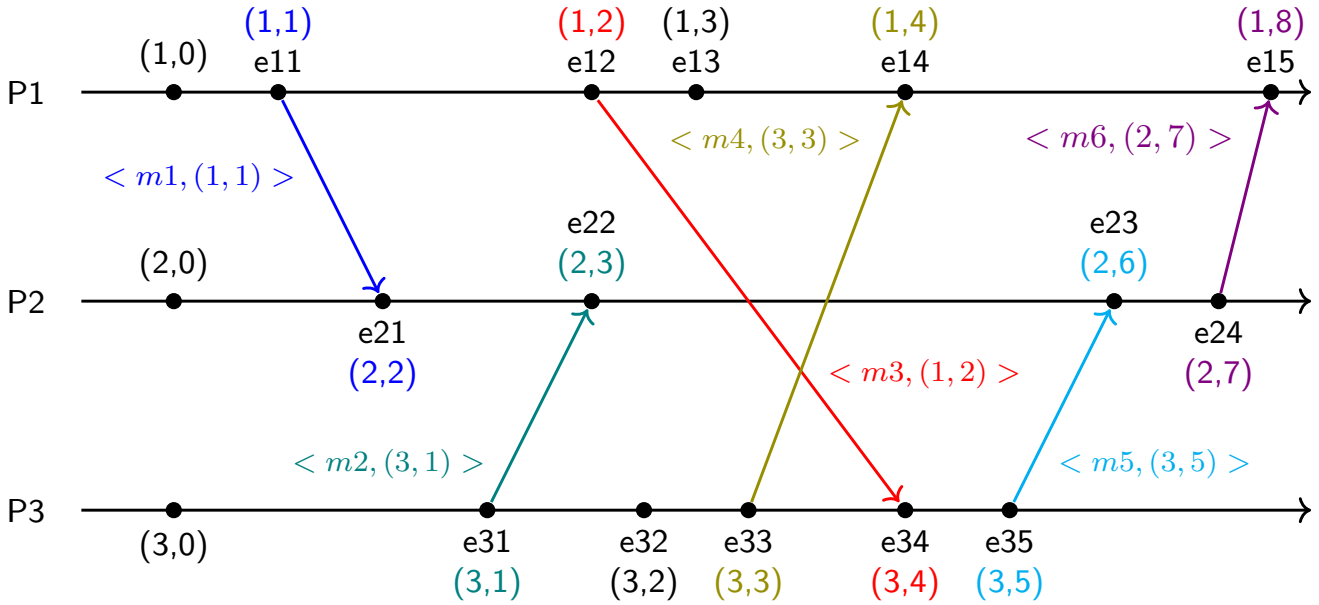
- Mais possibilité d'avoir deux évènements sur deux processus différents avec la même horloge (compteur *local*)

$\rightarrow$ Pas assez d'information !

Ordre des messages : estampille

Estampille des messages : (p, H_p)

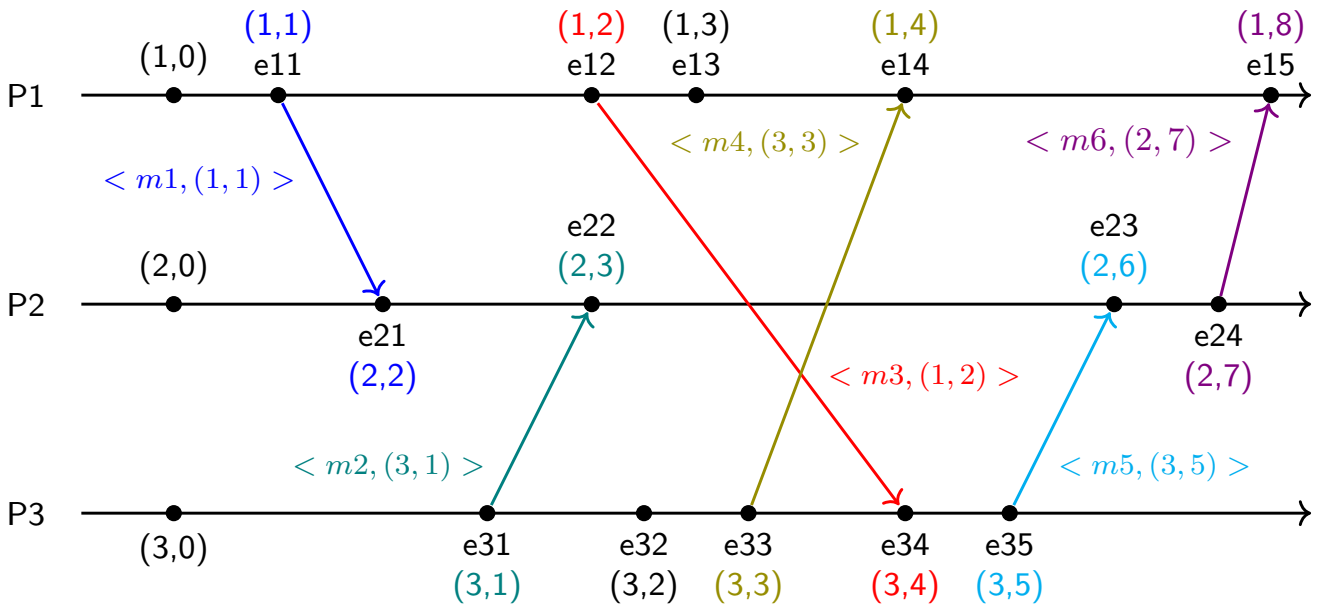
- p : numéro du processus
- H_p : horloge



Ordre des messages : estampille

Estampille des messages : (p, H_p)

- p : numéro du processus
- H_p : horloge



On obtient un ordonnancement *global* des évènements du système

- Ordre **total**
- Attention, c'est un ordre *arbitraire*
  - Si il y a dépendance causale entre deux évènements : l'ordre respecte la dépendance
  - Dans le cas contraire (pas de dépendance causale) : un ordre entre les deux est choisi arbitrairement (pas un problème car indépendants)

Plan du cours

Horloges vectorielles

Les horloges de Lamport scalaires manquent d'information

- Information sur l'ordre *local* des évenements
- État des horloges des autres processus ?

→ Utilisation d'horloges **vectorielles**

Également appelées ***horloge de Mattern***

- Chaque processus a un vecteur d'horloges
- Nombre d'éléments = nombre de processus

Chaque processus tient à jour sa propre horloge

- L' **horloge de l'émetteur** est transmise avec chaque message
- En réception : mise à jour de l' **horloge du destinataire**

Horloge de Mattern

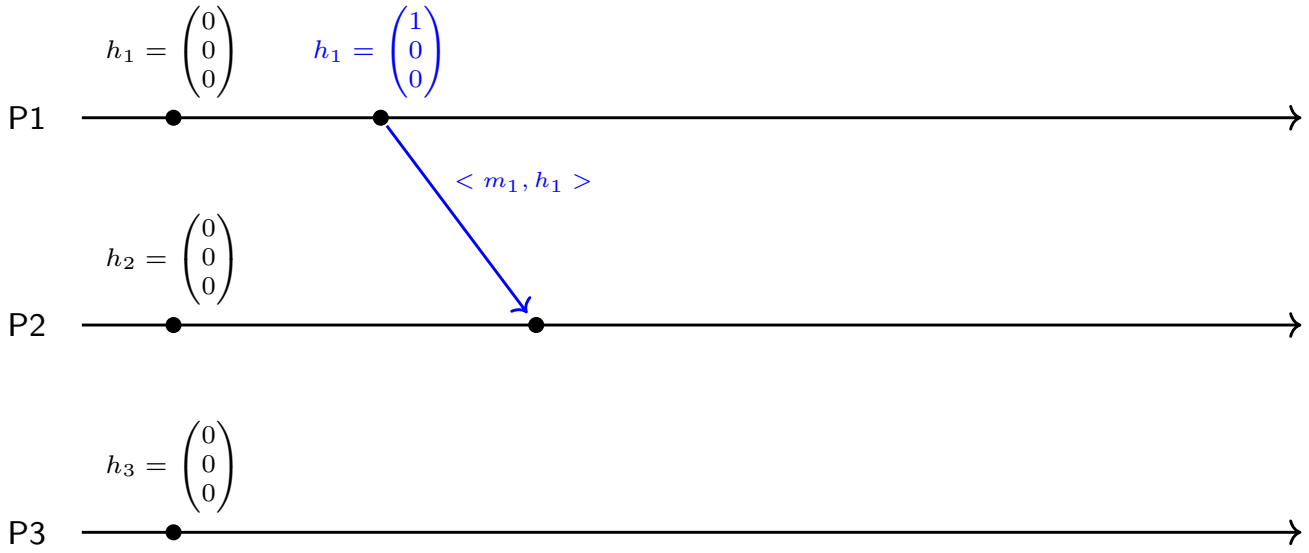
Mise à jour de l'horloge

- L'horloge du processus local est incrémentée (un évènement arrive)
- Pour les horloges des autres processus : max(H_{loc}, H_{reu})

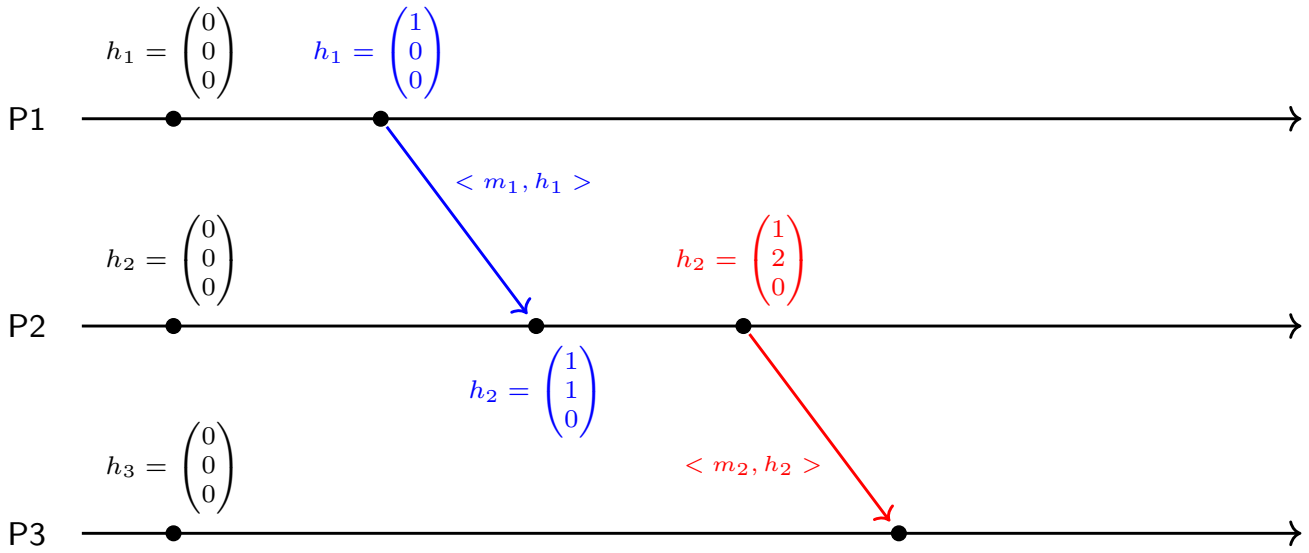
Exemples :

- Je suis le processus 0. J'ai [2, 4, 0, 2], je reçois [0, 2, 5, 1]
  - J'incrémente mon horloge locale : [3, 4, 0, 2]
  - Mise à jour : [3, 4, 5, 2]
- Je suis le processus 2. J'ai [3, 5, 3, 5], je reçois [0, 2, 1, 3]
  - J'incrémente mon horloge locale : [3, 5, 4, 5]
  - Mise à jour : [3, 5, 4, 5]

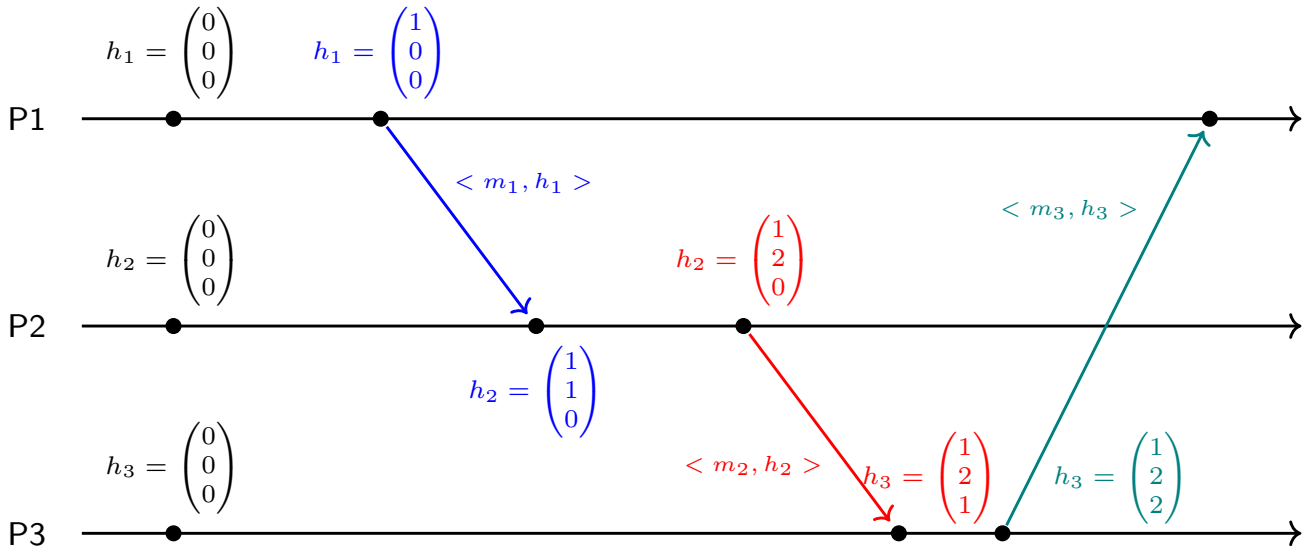
Horloges de Mattern : exemple



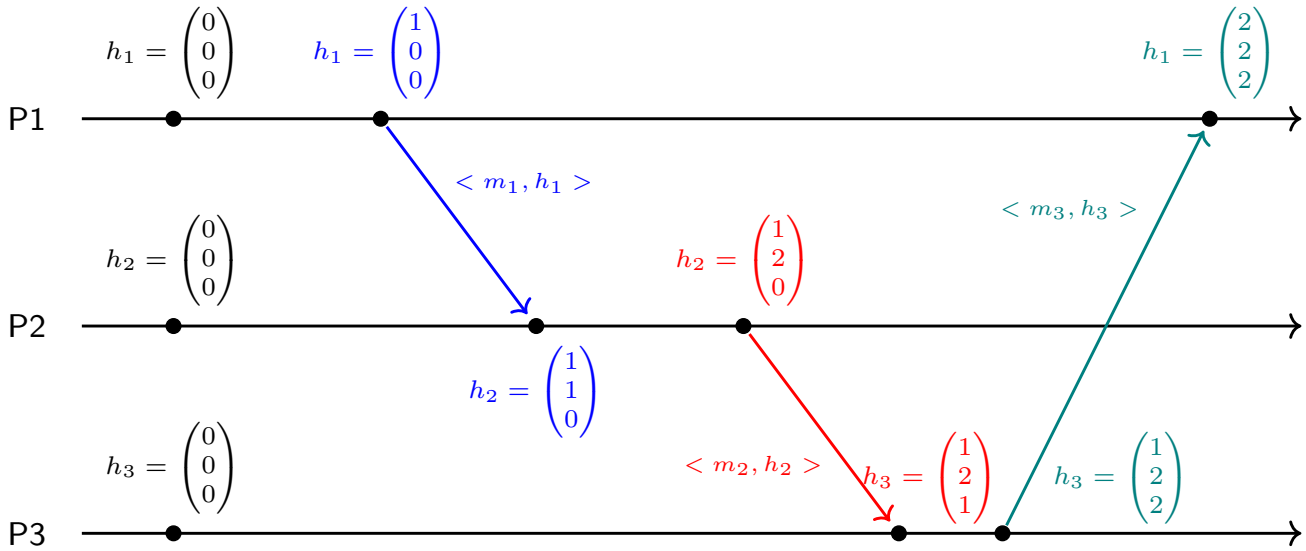
Horloges de Mattern : exemple



Horloges de Mattern : exemple



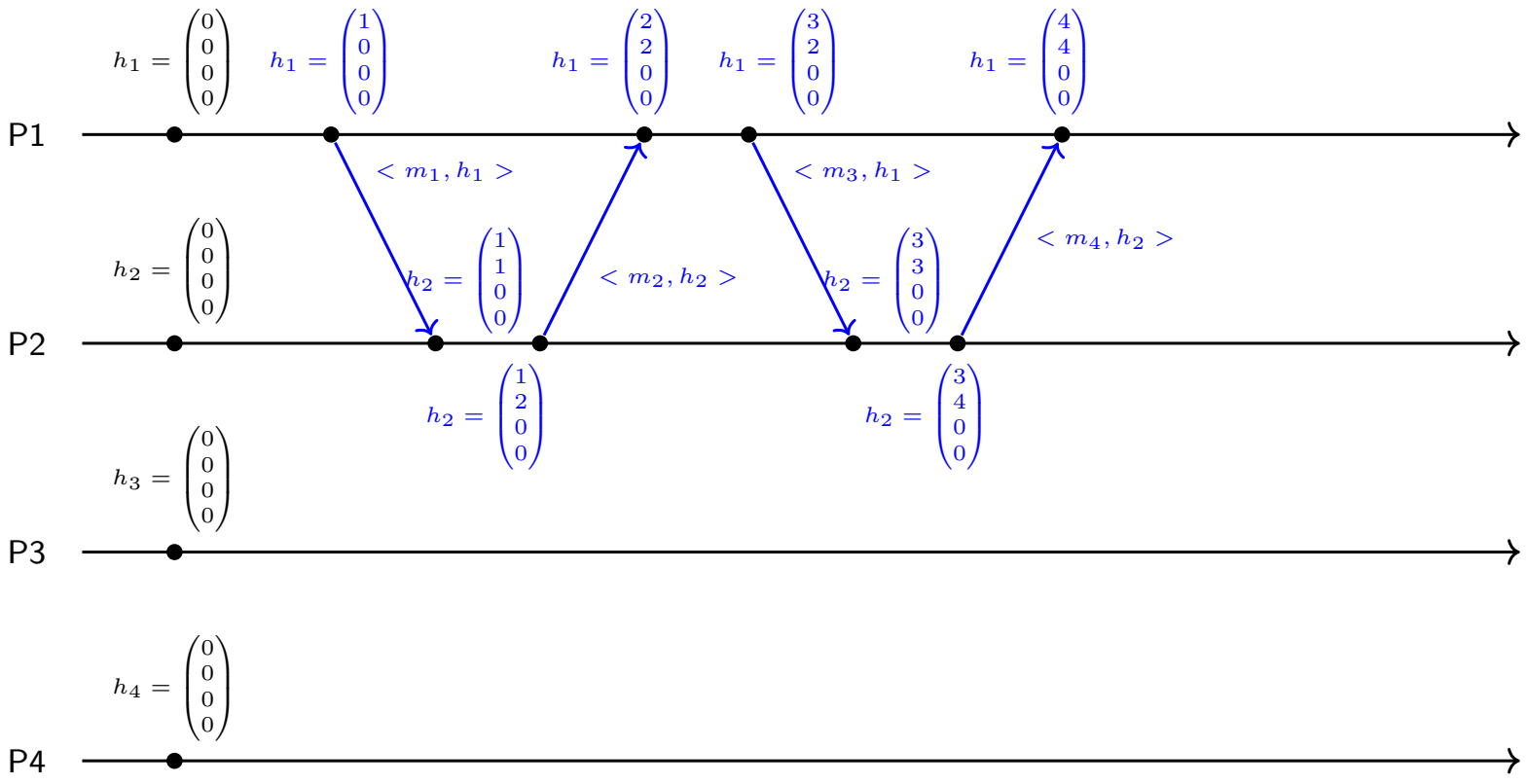
Horloges de Mattern : exemple



Horloges de Mattern

Remarques :

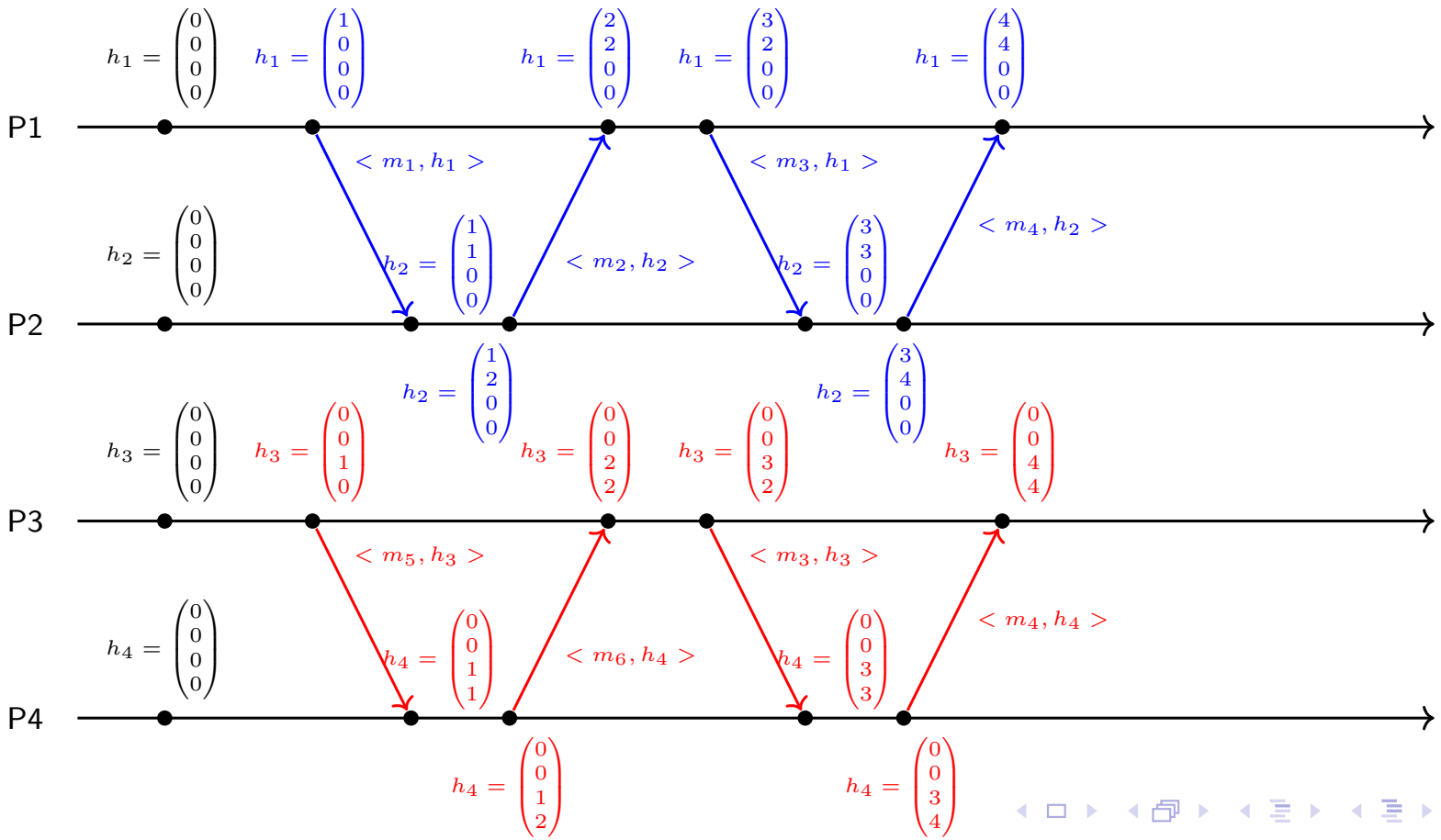
- Traduit la dernière fois qu'un processus a *eu une influence* sur un autre processus
  - Si un processus n'a pas eu de mise à jour de l'horloge d'un autre processus, c'est que ce dernier n'a pas eu d'influence sur lui
  - Transitif : la mise à jour est propagée



Horloges de Mattern

Remarques :

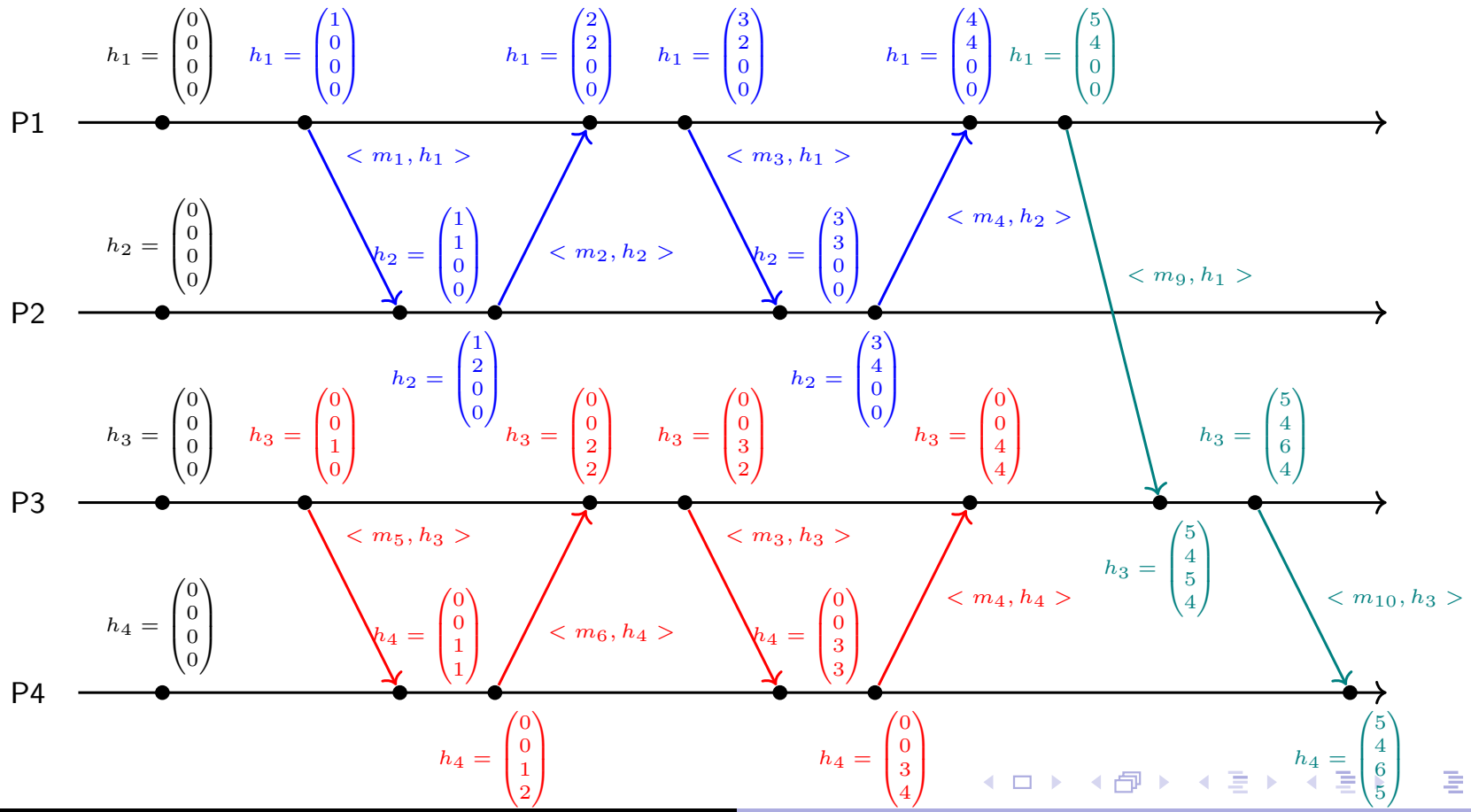
- Traduit la dernière fois qu'un processus a *eu une influence* sur un autre processus
  - Si un processus n'a pas eu de mise à jour de l'horloge d'un autre processus, c'est que ce dernier n'a pas eu d'influence sur lui
  - Transitif : la mise à jour est propagée



Horloges de Mattern

Remarques :

- Traduit la dernière fois qu'un processus a *eu une influence* sur un autre processus
  - Si un processus n'a pas eu de mise à jour de l'horloge d'un autre processus, c'est que ce dernier n'a pas eu d'influence sur lui
  - Transitif : la mise à jour est propagée



Utilisation des horloges de Mattern

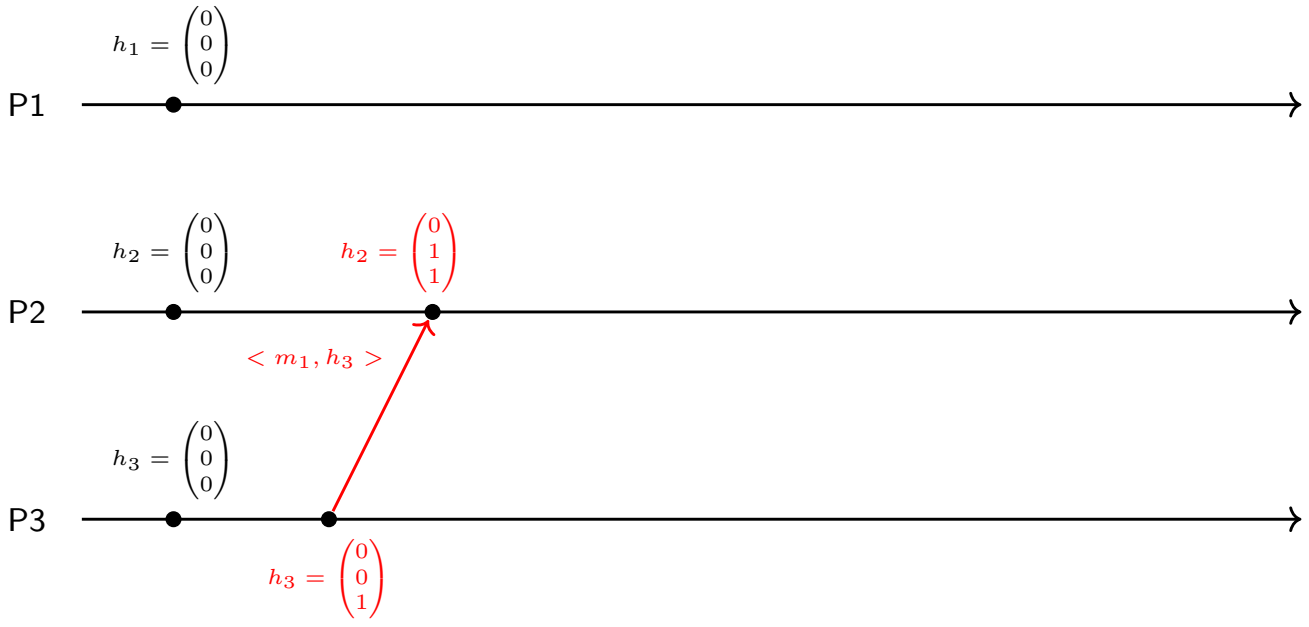
Relation d'ordre sur les horloges vectorielles

- $V \leq W \Leftrightarrow \forall i : V[i] \leq W[i]$
- $V < W \Leftrightarrow \forall i : V[i] \leq W[i]$ et $\exists j / V[j] < W[j]$

Informations de causalité

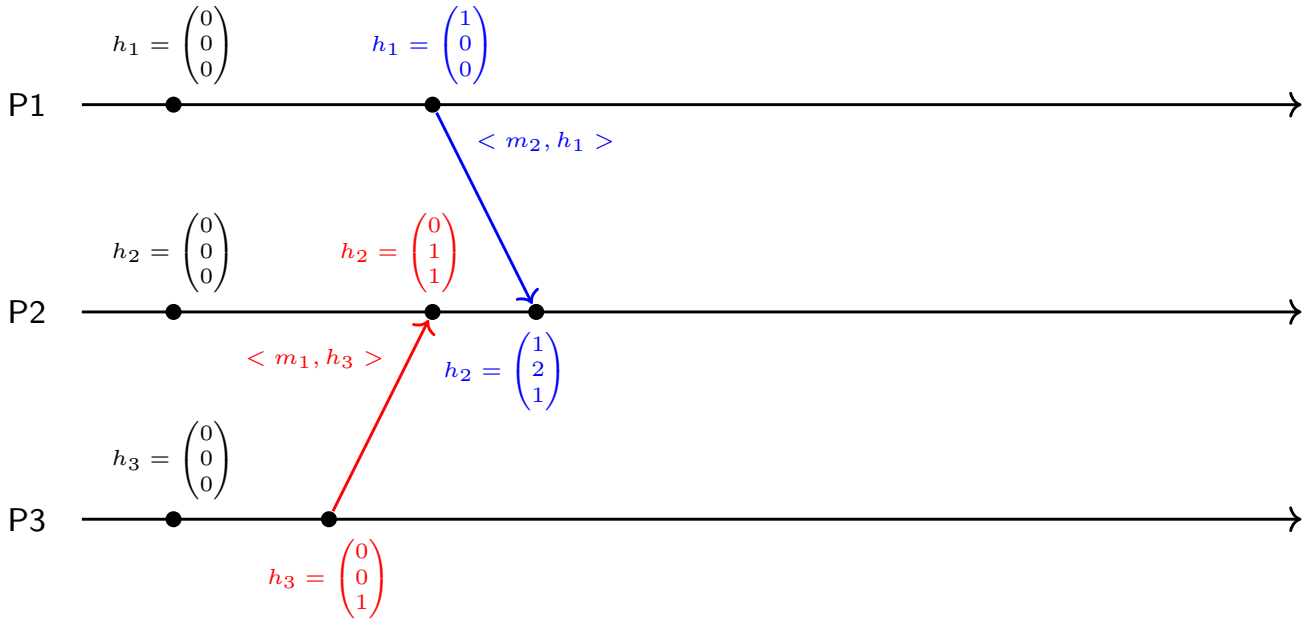
- Si les horloges de deux évènements sont comparables, alors **ils sont ordonnés causalement**
  - e et e' ; si $H(e) < H(e')$ alors $e \prec e'$
- Si on ne peut pas établir d'ordre entre les deux horloges : **les évènements sont concurrents** (indépendants causalement)
  - e et e' ; si on n'a ni $H(e) < H(e')$ ni $H(e) > H(e')$ alors $e||e'$

Ordre causal avec des horloges de Mattern



Ordre entre les messages :

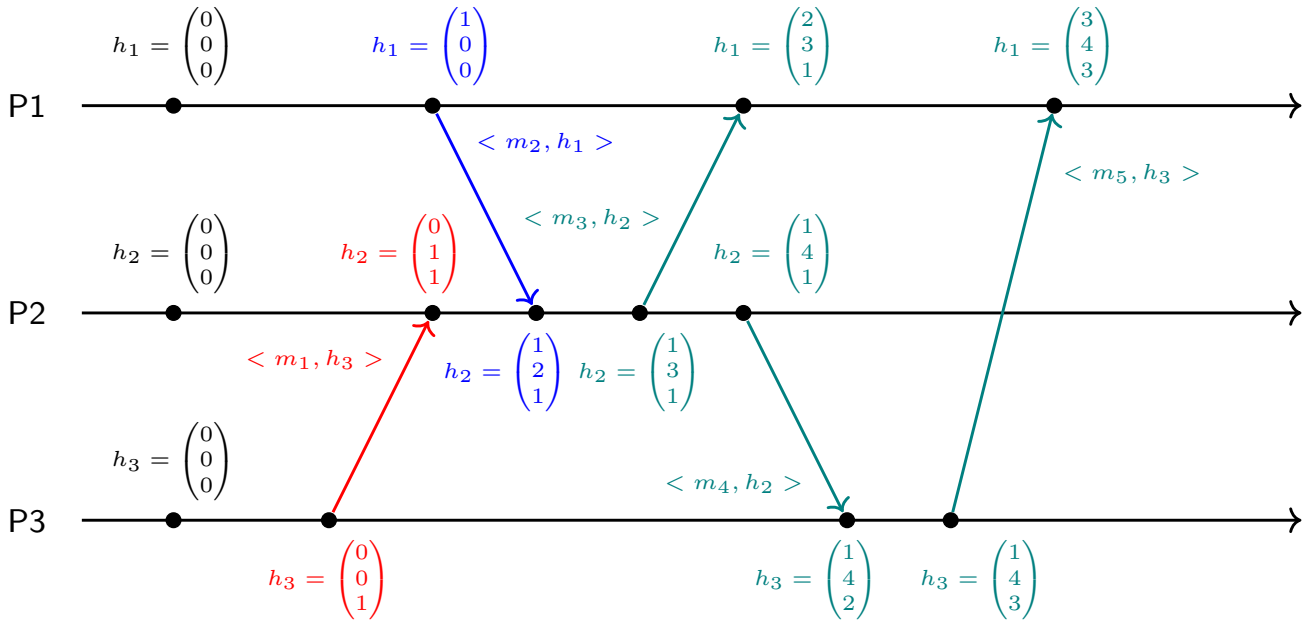
Ordre causal avec des horloges de Mattern



Ordre entre les messages :

- Quand m_2 est reçu, peut-on comparer h_1 et h_2 ?
  - m_1 et m_2 sont-ils ordonnés ?

Ordre causal avec des horloges de Mattern



Ordre entre les messages :

- Quand m_2 est reçu, peut-on comparer h_1 et h_2 ?
  - m_1 et m_2 sont-ils ordonnés ?
- Quand m_5 est reçu, peut-on comparer h_1 au moment de l'envoi de m_1 et h_3 ?
  - l'envoi de m_1 et la réception de m_5 sont-ils ordonnés ?
- Quand m_5 est reçu, peut-on comparer h_1 au moment de la réception de m_3 et h_3 ?
  - les réceptions de m_3 et de m_5 sont-elles ordonnées ?

Plan du cours

Horloge matricielle

Généralisation de l'horloge vectorielle

- Chaque processus maintient une **matrice** $p \times p$
- Chaque ligne k contient l' **horloge vectorielle du processus** k
- À chaque interaction du processus i avec le processus j :
  - Le processus i met à jour la case j de la ligne i

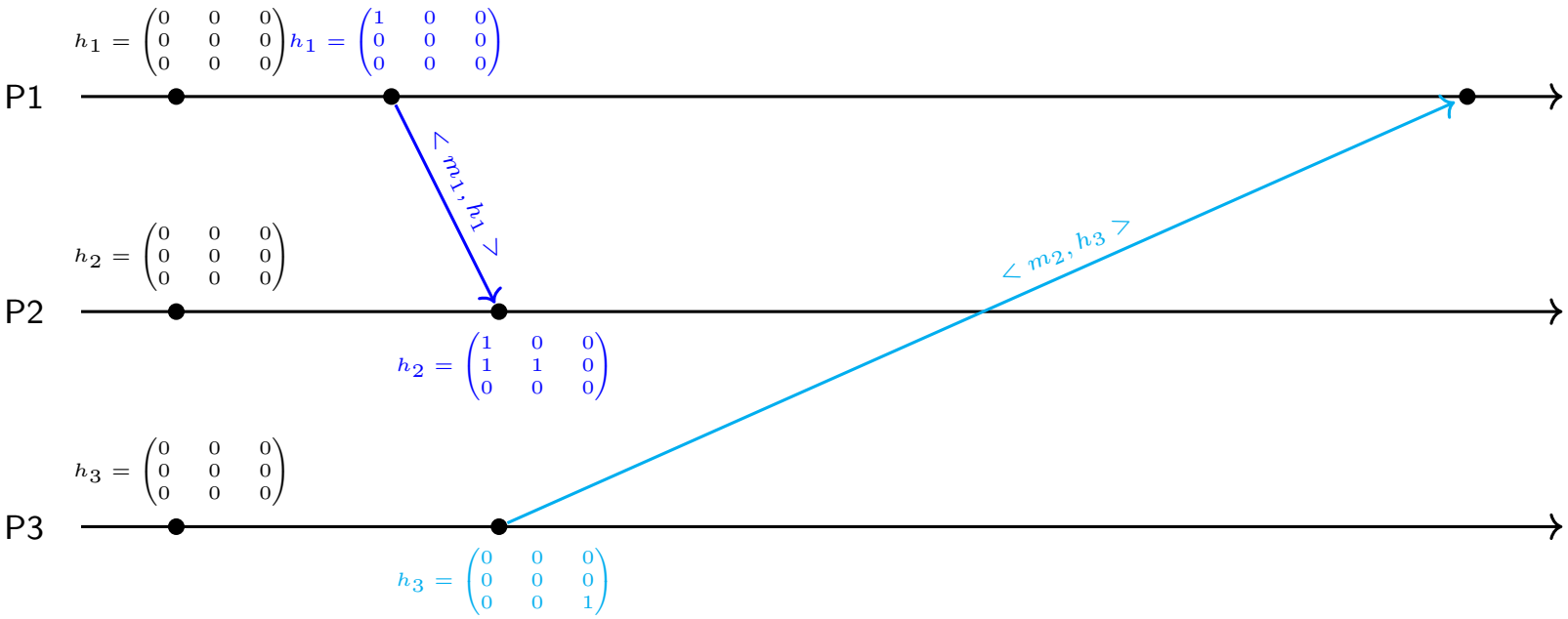
Transmission et mise à jour :

- Avant de réaliser un évènement sur le processus i :
  - On incrémente son horloge locale $M[i][i]$
- L'horloge du processus émetteur est transmise avec le message
- À la réception d'un message venant de j et de l'horloge qui l'accompagne :
  - On met à jour son information local en utilisant l'information du processus émetteur
    $\forall k : M[i][k] = max(M[i][k], M'[j][k])$
  - On met à jour l'information que l'on a sur le reste du système en utilisant l'information dont dispose le processus émetteur
    $\forall k, \forall n : M[k][n] = max(M[k][n], M'[k][N])$

Informations contenues dans une horloge matricielle, sur le processus i :

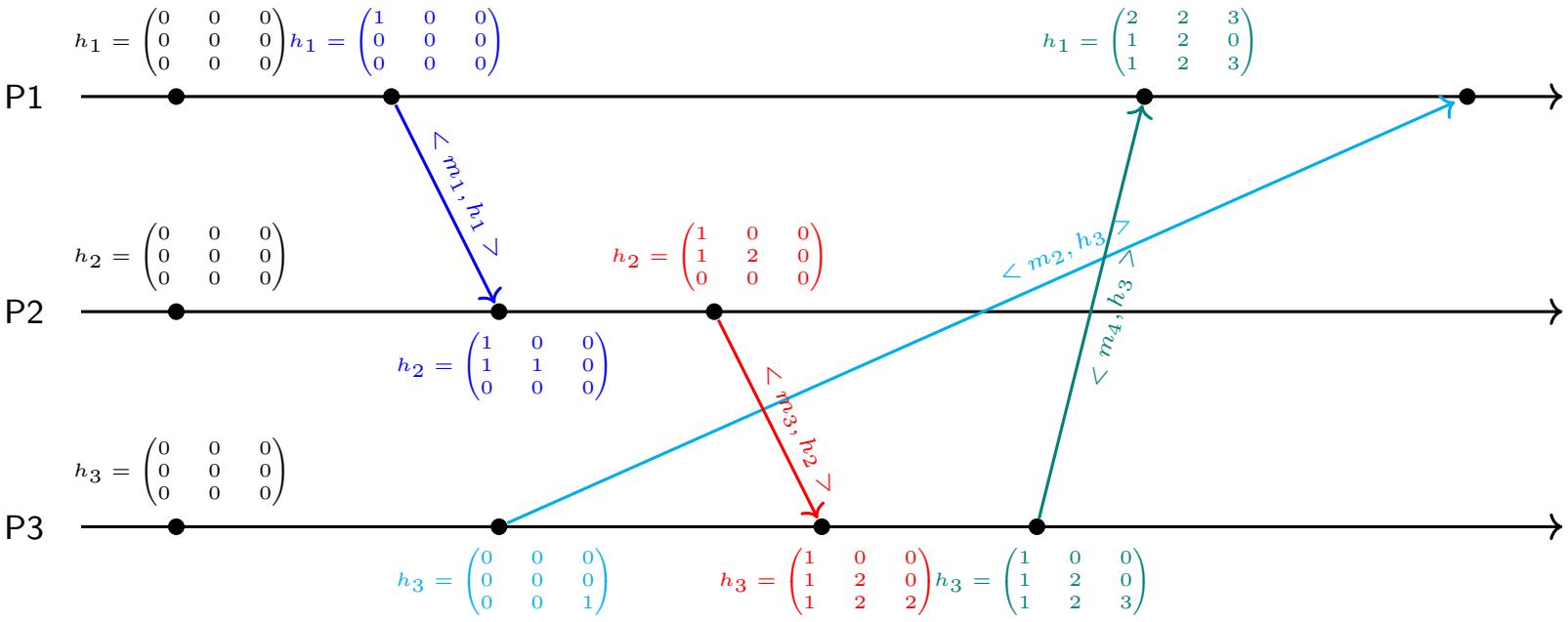
- $M[i][i]$: horloge logique **locale**
- $M[i][j]$: dernière information que l'on a sur l' **horloge logique de** j
- $M[j][k]$: connaissance que i a sur l' **information dont** j **dispose sur le processus** k

Horloge matricielle



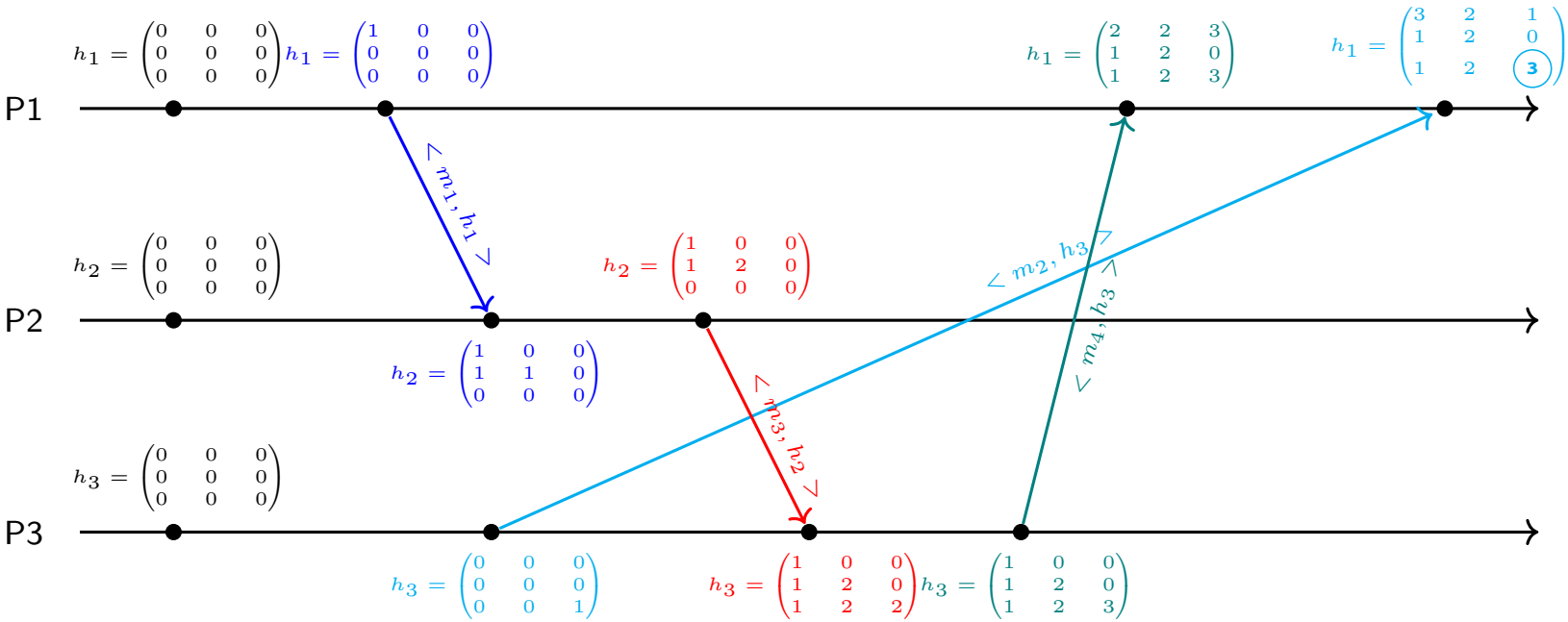
- Le message m_2 a un temps de transmission long

Horloge matricielle



- Le message m_2 a un temps de transmission long
- Le message m_4 arrive avant m_2
  - Quand m_4 arrive, $P1$ se rend compte que h_3 a un temps plus avancé sur lui-même (3ème ligne, 1ere colonne)
  - $P3$ a donc envoyé un autre message à $P1$ avant m_4

Horloge matricielle



- Le message m_2 a un temps de transmission long
- Le message m_4 arrive avant m_2
  - Quand m_4 arrive, $P1$ se rend compte que h_3 a un temps plus avancé sur lui-même (3ème ligne, 1ere colonne)
  - $P3$ a donc envoyé un autre message à $P1$ avant m_4
- Quand m_2 arrive, on voit que c'était le message manquant

Plan du cours

Comparaison des types d'horloges

Horloge scalaire (Lamport)

- Datation des évènements locaux
- Causalité des évènements locaux
- Au niveau global : fournit l'ordre des évènements
- Ordre arbitraire, pas causal

Horloge vectorielle (Mattern)

- Informations de causalité des évènements (dépendances causales)
- Informations de concurrence (indépendance causale)
- Datation de la dernière interaction d'une partie du système sur un processus

Horloge matricielle

- Informations sur la connaissance qu'a chacun du temps du système
- Informations de causalité *et* de chronologie
- Ordre de livraison des messages

Plan du cours

Plan du cours

Motivation

Besoin d'exclusivité

- Accès à une ressource partagée
  - Écriture sur un système de fichier partagé, accès à un système d'impression, ...

Dans un système centralisé :

- Verrou, sémaphore, etc...

Dans un système distribué : **pas de tel mécanisme !**

Propriétés souhaitées

On souhaite avoir deux propriétés :

- **Correction** : à tout instant, il ne peut y avoir qu'un seul processus au plus en section critique (par définition)
- **Vivacité** : tout processus demandant à entrer en section critique est sûr d'y entrer en un temps fini
  - Pas d'interblocage
  - Équitable

Un processus peut être dans un de ces 3 états :

- **Demandeur** : voulant entrer en section critique
- **Dedans** : dans la section critique
- **Dehors** : en-dehors de la section critique et non demandeur

L'algorithme d'exclusion mutuelle gère le **passage de demandeur à dedans** .

Types d'algorithmes d'exclusion mutuelle

Centralisés

- Un serveur centralisé donne les permissions
- Les clients s'adressent au serveur pour demander la permission

À jeton

- Les processus se passent un jeton
- Celui qui a le jeton peut entrer en section critique

Par permission

- Les processus donnent la permission à un processus d'entrer en section critique
- Permission individuelle ou permission collective

Plan du cours

Algorithme centralisé

Algorithme centralisé : un serveur **coordonne l'accès**

Algorithm 1: Algorithme du serveur

```
while Vrai do
    m = recevoir () ;
    if m == REQUEST then
        /* Envoyer les autorisations selon un ordre déterminé    */
        decider () ;
    if m == LIBERATION then
        /* Un client est sorti de la section critique, envoyer un
           autre dedans                                          */
        decider () ;
```

Algorithme centralisé (suite)

Algorithm 2: Algorithme du client

```
envoyer ( REQUEST ) ;
recevoir ( AUTORISATION ) ;
/* On peut entrer en section critique                    */
sectioncritique () ;
/* On prévient le serveur qu'on en sort                  */
envoyer( LIBERATION) ;
```

Propriétés de l'algorithme centralisé

Correction

- À tout instant, il ne peut y avoir qu'un seul processus au plus en section critique

Vivacité

- Tout processus demandant à entrer en section critique est sûr d'y entrer en un temps fini

$\rightarrow$ Propriétés assurées par le choix du serveur centralisé

Propriétés de l'algorithme centralisé

Correction

- À tout instant, il ne peut y avoir qu'un seul processus au plus en section critique

Vivacité

- Tout processus demandant à entrer en section critique est sûr d'y entrer en un temps fini

$\rightarrow$ Propriétés assurées par le choix du serveur centralisé

Avantages

- Simple

Inconvénients

- Approche centralisée : site particulier, goulet d'étranglement potentiel, point central de défaillance

Plan du cours

Circulation de jeton

Principe : **un jeton circule entre les processus**

- Celui qui a le jeton peut être en section critique
- Quand on sort de la section critique, on passe le jeton

Mise en place de l'algorithme

- Définition d'un ordre de **circulation du jeton**
- Définition d'une procédure d' **initialisation** : introduction du jeton
- Si un processus n'a pas besoin d'être en section critique : il **transfère le jeton**

Avantages

- Simple et flexible (topologie)

Inconvénients

- Problème si un processus injecte un autre jeton.

Jeton dans un anneau : algorithme

Le Lann, 1977

Algorithm 3: Jeton dans un anneau

```
/* Initialisation                                                    */
if INITIATEUR then
    envoyer ( jeton, droite ) ;
/* Boucle principale                                                 */
while Vrai do
    jeton = recevoir ( gauche ) ;
    /* On a le jeton                                                 */
    sectioncritique () ;
    /* On sort de la section critique : on libère le jeton           */
    envoyer ( jeton, droite ) ;
```

Jeton dans un anneau : propriétés

Correction

Jeton dans un anneau : propriétés

Correction

- Et si un processus injecte un autre jeton ?

Jeton dans un anneau : propriétés

Correction

- Et si un processus injecte un autre jeton ?

Vivacité

Jeton dans un anneau : propriétés

Correction

- Et si un processus injecte un autre jeton ?

Vivacité

- Et si on perd le jeton (panne) ?

Jeton dans un anneau : propriétés

Correction

- Et si un processus injecte un autre jeton ?

Vivacité

- Et si on perd le jeton (panne) ?

Nécessité de mettre en place :

- Détection de jetons dupliqués
- Détection de panne avec réinjection de jeton

Jeton dans un anneau : diffusion

Variante, Ricart & Agrawala 1983

- On n'est plus dans un anneau mais une topologie quelconque
- Au lieu d'attendre le jeton, on diffuse qu'on le veut
- On conserve *dans le jeton* le nombre de fois où chaque processus a eu le jeton

Jeton dans un anneau : diffusion

Variante, Ricart & Agrawala 1983

- On n'est plus dans un anneau mais une topologie quelconque
- Au lieu d'attendre le jeton, on diffuse qu'on le veut
- On conserve *dans le jeton* le nombre de fois où chaque processus a eu le jeton

Algorithm 5: Ricart & Agrawala : initialisation

```
/* Initialisation                                          */
H ← 0 ;
for i ← 0; i < N; i++ do
    nbreq[i] ← 0 ;
/* L'initiateur possède le jeton                           */
if INITIATEUR then
    for i ← 0; i < N; i++ do
        jeton[i] ← 0 ;
    jetonIci ← Vrai
else
    jetonIci ← Faux
etat ← dehors ;
```

Jeton dans un anneau : diffusion

Quand un processus veut le jeton, il le demande

Algorithm 6: Ricart & Agrawala : attente

```
/* Je reçois une requête venant de i                                  */
H_i, i = recevoir () ;
nbreq[i] ← nbreq[i] + 1 ;
/* Je mémorise que i a demandé à l'heure H_i                          */
demandes ← concatajouteener (demandes, i, H_i) ;
/* Je mets à jour mon horloge                                         */
jeton[moi] ← max ( H_i, jeton[moi] ) ;
if jetonIci == Vrai ET etat == dehors then
    j ← prioritaire ( demandes );
    envoyer ( jeton, j );
    jetonIci ← Faux ;
    demandes ← supprime ( j ) ;
```

Jeton dans un anneau : diffusion

Algorithm 7: Ricart & Agrawala : passage en section critique

```
/* Diffusion de la demande = horloge locale                    */
for j ← 0; j < N; j++ do
    if j != moi then
        envoyer ( H, j );
jeton = recevoir () ;
jetonIci ← Vrai ;
etat ← dedans ; /* On a le jeton                               */
sectioncritique () ;
/* On sort de la section critique : on libère le jeton         */
etat ← dehors ;
H ← H + 1 ;
jeton[moi] = H ;
/* Est-ce que quelqu'un veut le jeton                          */
for j ← 0; j < N; j++ do
    if j != moi then
        if nbreq[j] > jeton[j] then
            envoyer ( jeton, j ) ;
            jetonIci ← Faux ;
```

Fonctionnement de l'algorithme

C'est un système de **file d'attente distribuée**

- Les processus qui demandent le jeton sont mis en file d'attente
- Ordre fourni par l'horloge

Les processus maintiennent une **horloge**

- Horloge dans le jeton
- On incrémente son horloge quand on a le jeton
- On la transporte dans les requêtes et le jeton

Fonctionnement de l'algorithme

C'est un système de **file d'attente distribuée**

- Les processus qui demandent le jeton sont mis en file d'attente
- Ordre fourni par l'horloge

Les processus maintiennent une **horloge**

- Horloge dans le jeton
- On incrémente son horloge quand on a le jeton
- On la transporte dans les requêtes et le jeton

Correction

- Impossible d'y avoir deux processus dans la section critique (preuve par l'absurde)

Vivacité

- Garantie par le système d'horloges, qui en plus assure l'ordre des requêtes.

Fonctionnement de l'algorithme

C'est un système de **file d'attente distribuée**

- Les processus qui demandent le jeton sont mis en file d'attente
- Ordre fourni par l'horloge

Les processus maintiennent une **horloge**

- Horloge dans le jeton
- On incrémente son horloge quand on a le jeton
- On la transporte dans les requêtes et le jeton

Correction

- Impossible d'y avoir deux processus dans la section critique (preuve par l'absurde)

Vivacité

- Garantie par le système d'horloges, qui en plus assure l'ordre des requêtes.

Avantage

- Requêtes ordonnées.

Inconvénient

- Nombreux messages échangés.

Plan du cours

Algorithme de la boulangerie

Lamport 1974

Système de **ticket**

- Phase 1 : attribution du ticket
- Phase 2 : attente de son tour, entrée en section critique

Fonctionnement : variables locales et communications

- Variables : chaque processus a deux variables `choix` et `num`
- Communications entre les processus : un processus peut demander à un voisin les valeurs des siennes
- Aucun dispositif centralisé, complètement distribué

Inspiration : les systèmes de tickets

- La boulangerie (? ? ?), la sécu, le Mc Do, la préfecture de police...
- On récupère un ticket avec un numéro et on attend d'être appelé.

Algorithme de la boulangerie : phase 1

```
/* Initialisation                                                    */
choix ← Vrai ;
max ← 0 ;
compteur ← 0 ;
/* Calcul du max des tickets des autres processus                    */
for j ← 0; j < N; j++ do
    if j! = moi then
        compteur_j = obtenir ( compteur, j ) ;
        if max < c then
            max ← c ;
/* Attribution du nouveau ticket                                     */
compteur ← max + 1 ;
/* Fin de la phase d'attribution du ticket                           */
choix ← Faux ;
```

Notes :

- Il est possible que plusieurs processus aient le même max (si ils tournent en même temps)
- Dans cette phase, la variable booléenne $choix$ sert à savoir si on est en train de s'attribuer le ticket

Algorithme de la boulangerie : deuxième phase

```
/* On boucle sur tous les autres processus et on regarde si ils
   ont fini de s'attribuer le ticket                              */
for j ← 0; j < N; j++ do
    if j! = moi then
        choix_j ← obtenir ( choix, j ) ;
        while c == Vrai do
            choix_j ← obtenir ( choix, j ) ;
        /* Si le processus distant est plus prioritaire que moi :
           attendre                                                */
        compteur_j ← obtenir ( compteur, j ) ;
        while (compteur_j! = 0 ET compteur_j < compteur) OU
        (compteur_j == 0 ET j < moi) do
            compteur_j ← obtenir ( compteur, j ) ;
/* On y est !                                                     */
sectioncritique () ;
/* Sortie de la section critique                                  */
compteur ← 0 ;
```

Remarques sur l'algorithme de la boulangerie

Correction

- Si deux processus ont le même numéro de ticket (même max à l'issue de la première phase), celui avec le plus petit numéro a la priorité

$\rightarrow$ Au plus un processus à la fois dans la section critique

Vivacité

- Le numéro de ticket augmente, donc le tour de chacun finit par arriver.

Plan du cours

Plan du cours

Problème

Au départ, tous les processus sont **dans le même état**

- Pas de chef
- Processus non distinguables les uns des autres
- Chaque processus exécute **le même algorithme** (symétrie complète)

À la fin, il y a **un gagnant**

- ... et tous les autres processus savent qu'il est le gagnant !

Exemples : nommage de processus, détermination d'un initiateur pour un autre algorithme, etc.

Les processus sont :

- Au début : **dormant**
- Pendant l'exécution de l'algorithme : **candidat** (pas forcément tous)
- À la fin : **gagnant ou perdant**

Plan du cours

Élection dans un anneau

Le Lann 1977

- Circulation d'un jeton
- Possibilité d'avoir plusieurs initiateurs
- Les initiateurs sont candidats
- Chaque initiateur collecte la liste des identifiants des processus
- Les jetons contiennent l'identifiant de leur initiateur

Remarque : à comparer avec l'algorithme d'exclusion mutuelle de Le Lann.

Élection dans un anneau

Algorithm 8: Élection dans un anneau

```
/* Initialisation                                                    */
liste ← moi ;
etat ← dormant ;
if INITIATEUR then
    /* Initiateurs                                                  */
    etat ← candidat ;
    envoyer ( < jeton, moi >, droite ) ;
    id ← recevoir ( gauche );
    while id! = moi do
        liste ← ajouter ( liste, id );
        envoyer ( < jeton, id >, droite ) ;
        id ← recevoir ( gauche );
    if moi == min (list) then
        etat ← gagnant ;
    else
        etat ← perdant ;
else
    /* Autres                                                       */
    etat ← perdant ;
    while Vrai do
        id ← recevoir ( gauche );
        envoyer ( < jeton, id >, droite ) ;
```

Algorithme de Le Lann

Chaque processus initiateur p :

- Émet un message $< jeton, p >$
- Reçoit les messages $< jeton, q >$ de tous les autres initiateurs q
- Finit par recevoir $< jeton, p >$ (transféré après les autres)
- Calcule l'identité du plus petit initiateur
  - Comme ils reçoivent tous la liste, ils ont tous le même minimum

Plan du cours

Algorithme de Chang & Roberts

Principe :

- Chaque processus a un numéro unique
- Le gagnant est celui qui a le numéro de plus élevé

Topologie :

- **Anneau** (unidirectionnel)
- Chaque candidat **diffuse sa candidature** autour de l'anneau

Algorithme en deux phases :

- **Sélection** du chef (phase d'élection)
- **Diffusion** du résultat (phase d'annonce)

Donc : deux types de messages : `election` (pour annoncer les candidats) et `elu` (pour annoncer l'élu).

Algorithm 9: Initialisation

$participant \leftarrow Vrai$;
`envoyer` ($ELECTION$, moi, $droite$) ;

Algorithme de Chang & Roberts

Algorithm 10: Réception d'un message d'élection

$candidat =$ `recevoir` ($ELECTION$, $gauche$) ;
if $candidat > moi$ **then**
  `envoyer` ($ELECTION$, $candidat$, $droite$) ;
  $participant \leftarrow Vrai$;
if $candidat < moi$ *ET* $participant == Faux$ **then**
  `envoyer` ($ELECTION$, moi, $droite$) ;
  $participant \leftarrow Vrai$;
if $candidat == moi$ **then**
  `envoyer` (ELU, moi, $droite$) ;

Algorithm 11: Réception d'un message élu

$candidat =$ `recevoir` (ELU, $gauche$) ;
$gagnant \leftarrow candidat$;
$participant \leftarrow Faux$;
if $candidat! = moi$ **then**
  `envoyer` (ELU, $candidat$, $droite$) ;

Comportement de l'algorithme de Chang & Roberts

Nombre de messages envoyés

- Pendant la phase d'élection
  - Temps pour que la plus grande valeur fasse le tour
  - Dépend de l'ordre des identifiants (croissant, décroissant)
  - Au mieux n, au pire $2n - 1$ étapes
- Pendant la phase d'annonce
  - Au mieux n, au pire n^2 étapes

Élection dans un graphe binomial

Anneau bidirectionnel

- **Élections primaires** , 2 à 2, de proche en proche
- Utilisation de puissances de 2 croissantes
  - Chaque processus communique avec les processus à distance 2^k de lui
  - À chaque fois, on élit le chef entre les deux

Algorithm 12: Initiation de l'élection

$participant \leftarrow Vrai$;
$distance \leftarrow 1$;
`envoyerDroiteGauche` ($ELECTION$, moi, $distance$);

Élection dans un graphe binomial

Algorithm 13: Réception d'un message d'élection

$candidat =$ `recevoir` ($REPONSE$, $distance$) ;
if $candidat > moi$ **then**
  $distance \leftarrow distance + 1$;
  `envoyerDroiteGauche` ($ELECTION$, $candidat$, $distance$) ;
  $participant \leftarrow Vrai$;
if $candidat < moi$ *ET* $participant == Faux$ **then**
  $distance \leftarrow distance + 1$;
  `envoyerDroiteGauche` ($ELECTION$, moi, $distance$) ;
  $participant \leftarrow Vrai$;
if $candidat == moi$ **then**
  `envoyerDroiteGauche` (ELU, moi, $distance$) ;

Algorithm 14: Réception d'un message élu

$candidat =$ `recevoir` (ELU, $distance$) ;
$gagnant \leftarrow candidat$;
$distance \leftarrow distance - 1$;
$participant \leftarrow Faux$;
if $candidat! = moi$ **then**
  `envoyerDroiteGauche` (ELU, $candidat$, $distance$) ;

Généralisation dans un graphe quelconque

Graphe quelconque :

- Un processus a des voisins
- Pas forcément tous le même nombre

État : 3 valeurs (repos, en_cours, terminé)

Algorithm 15: Initialisation

if $etat == repos$ **then**
  $etat \leftarrow encours$;
  $chef \leftarrow moi$;
  `envoyervoisins` ($ELECTION, moi$) ;

Généralisation dans un graphe quelconque

Algorithm 16: Réception d'un message d'élection

```
candidat = recevoir ( ELECTION, voisin ) ;
if candidat < moi ET etat == repos then
    etat ← encours ;
    candidat ← moi ;
    if fairesuivre ( ELECTION, candidat, voisinsuivant) then
        etat ← termine ;
        envoyervoisins ( ELU, candidat ) ;
```

Algorithm 17: Réception d'un message élu

```
if candidat! = moi then
    etat ← termine ;
    fairesuivre ( ELU, candidat, voisinsuivant) ;
```

Plan du cours

Algorithme de la brute

Algorithme un peu différent des précédents

- On suppose qu' **on avait un leader**
- Le leader est tombé en panne
- Les survivants doivent **élire un nouveau leader**

3 types de messages :

- *Élection*
- *OK*
- *Élu*

Algorithme de la brute

Les processus qui ont détecté la panne lancent la nouvelle élection

- Chaque initiateur envoie un message $ELECTION$ aux processus d'identifiant supérieur
- Si aucun processus de répond OK :
  - Le processus gagne l'élection
  - Le processus envoie un message ELU à tous les processus
- Si on reçoit un OK :
  - Le processus a perdu l'élection
  - Attendre un message ELU
- Quand on reçoit des messages $ELECTION$:
  - Envoyer un message $ELECTION$ aux processus d'identifiant supérieur (démarrer une élection)
- Si on reçoit un ELU :
  - On sait qui a gagné

Plan du cours

Algorithme

L'algorithme de la brute fonctionne **dans un graphe complet**

- Et sur un arbre ?

Lorsqu'un processus détecte la panne

- Il envoie un message $DEBUT$ à tous les processus
- Le message $DEBUT$ est transmis dans tout l'arbre

Le message $DEBUT$ signifie le début de l'algorithme

Les feuilles débutent l'algorithme

- Même

Le message $DEBUT$ signifie le début de l'algorithme

Élection dans un arbre

Algorithm 18: Initialisation

for $i \leftarrow 0; i < N; i++$ **do**
  $recu[i] \leftarrow Faux$;

Algorithm 19: Vague : sur réception d'un paquet de démarrage

`/* On recoit de tous les voisins sauf 1 */`
while `compter` $(recu[*] == Faux) > 1$ **do**
  $c =$`recevoir` (v) ;
  $recu[v] \leftarrow Vrai$;
q tel que $recu[q] = Faux$;
`envoyer` (OK, q) ;

Élection dans un arbre

Algorithm 20: Désignation du leader

$paquet \leftarrow$ `recevoir` (p);
$s \leftarrow$ `source` (p) ;
$recu[s] \leftarrow Vrai$;
if `compter` $(recu[*] == Faux\ ) == 1$ *ET* $recu[y] == Faux$ **then**
  `envoyer` (OK, y) ;
if `compter` $(recu[*] == Faux\ ) == 0$ **then**
  `decision` () ;

Plan du cours

Plan du cours

Qu'est-ce que le consensus

Définition : Tous les processus *corrects* se mettent d'accord sur une valeur.

- Initialement, chaque processus propose une valeur.
- À la fin de l'algorithme, tous les processus décident d'une même valeur.

Propriétés :

- **Terminaison** : tout processus doit décider d'une valeur.
- **Intégrité** : tout processus décide une fois, et sa valeur est définitive.
- **Validité** : la valeur décidée a été proposée par un des processus (c'est l'une des valeurs proposées).
- **Accord** : tous les processus corrects décident de la même valeur.

Un protocole qui peut garantir ces propriétés en présence de moins de t pannes est dit **t-robuste** .

Variantes du problème

Consensus uniforme

- La valeur décidée est la même pour tous les processus qui décident

k-consensus

- Au plus k valeurs distinctes sont décidées pour l'ensemble des processus corrects

Consensus approximatif (ϵ-accord)

- Les valeurs décidées par les processus corrects doivent être à distance maximale ϵ l'une de l'autre

Problèmes dérivés

- Élection de leader, élection mutuelle
- Échec ou validation (abort ou commit) dans une transaction distribuée
- Consensus sur si un processus est en panne : détection de défaillances
- Systèmes de contrôle aérien : tous les avions doivent avoir la même vue
- Diffusion fiable : tous les processus délivrent le message ou pas
- ...

Beaucoup de problèmes dérivent du consensus !

Valence

Dans un système où les processus doivent se mettre d'accord entre deux valeurs 0, 1, une configuration peut-être :

- **0-valente** : à partir de cette configuration initiale, seul le résultat "0" est possible
- **1-valente** : à partir de cette configuration initiale, seul le résultat "1" est possible
- **Bivalente** : à partir de cette configuration initiale, les deux résultats "0" ou "1" sont possibles

Lemme 1

Tout protocole de consensus doit avoir un état initial *bivalent*.

Preuve : par contradiction

P0	P1	P2	P3	
0	0	0	0	← 0-valent
0	0	1	1	
0	1	1	1	
1	1	1	1	← 1-valent

Deux configurations successives, bien qu'univalentes, ne diffèrent que par la valeur d'un processus. Mais ne donnent pas le même résultat.
Si ce processus tombe en panne, elles seront identiques alors qu'elles ne donnent pas le même résultat.
C'est donc impossible.

Valence

Corollaire du lemme précédent

Aucune exécution partant d'un état 0-valent peut mener au consensus sur la valeur 1, et inversement.

Preuve : par définition des états 0-valent et 1-valent.

Accessibilité

Lemme 2

Dans un protocole de consensus, en démarrant de n'importe quel état initial bivalent S, il existe un état accessible bivalent T tel que toute action effectuée par un processus p dans l'état T amène vers soit un état 0-valent, soit un état 1-valent.

Preuve : si tous les états qui suivent S sont bivalents, alors on n'atteint jamais le consensus. Pour résoudre le problème du consensus, il faut qu'à partir d'un moment de l'exécution, on arrive à un état qui finira par donner un état univalent.

Corollaire

Dans un protocole de consensus, depuis n'importe quel état bivalent S, il existe une action possible telle que l'état suivant soit aussi bivalent.

Preuve : par contradiction, en cas de panne d'un processus.

Plan du cours

Théorème d'impossibilité

Théorème de Fischer, Lynch et Patterson (FLP)

Dans un système distribué asynchrone, il est impossible de résoudre le problème du consensus en présence d'au moins une panne.

Preuve : pour une configuration bivalente, on peut se retrouver dans une autre configuration bivalente (lemme 2 : accessibilité).
Par ailleurs, en asynchrone, une décision basée sur l'observation qu'un processus P_i est en panne peut être contredite plus tard par une action de P_i.
Donc on peut se retrouver dans une exécution infinie où les configurations bivalentes se succèdent : le protocole de consensus peut ne pas pouvoir décider d'une valeur. □

Théorème d'impossibilité

Théorème de Fischer, Lynch et Patterson (FLP)

Dans un système distribué asynchrone, il est impossible de résoudre le problème du consensus en présence d'au moins une panne.

Preuve : pour une configuration bivalente, on peut se retrouver dans une autre configuration bivalente (lemme 2 : accessibilité).
Par ailleurs, en asynchrone, une décision basée sur l'observation qu'un processus P_i est en panne peut être contredite plus tard par une action de P_i.
Donc on peut se retrouver dans une exécution infinie où les configurations bivalentes se succèdent : le protocole de consensus peut ne pas pouvoir décider d'une valeur. □

Autrement dit *(avec les mains)* :
Si un processus ne répond pas, on ne peut pas savoir si il est en panne ou très lent (asynchronisme).

Comment faire autrement

On contourne le problème :

- Modèles de systèmes “partiellement synchrones”
  - Introduction de synchronisme (opération de synchronisation des processus...)
  - Hypothèses sur les communications
- On atteint des consensus imparfaits
  - Pas forcément exact
  - Avec un timeout (réduction de l'asynchronisme)

Plan du cours

Principe de Paxos

Lamport, 1998

Les processus peuvent avoir trois rôles :

- Proposeur : envoie la valeur qu'il propose aux accepteurs
- Accepteur : décide d'une valeur (à plusieurs accepteurs)
- Apprenti : apprend la valeur choisie

Le protocole parle d'*agents*. Au cours de l'exécution, un processus peut jouer le rôle de plusieurs agents.

Approche naïve (non satisfaisante) :

- Un accepteur unique.
- Un proposeur envoie sa valeur à l'accepteur
- L'accepteur garde la première valeur reçue
- Cette valeur est utilisée pour le consensus

Approche **non satisfaisante** : en cas de panne de l'accepteur, le protocole est bloqué.

Paxos : choix de la valeur

Les *proposeurs* envoient leur valeur à **un ensemble d'accepteurs**

- On a plusieurs accepteurs
- Tous à la fois !

Choix de la valeur

- On veut que ça fonctionne même si une seule valeur est proposée
  - Donc les accepteurs gardent la première valeur qu'ils reçoivent
- Problème : si plusieurs proposeurs envoient leur valeur en même temps
- Système de vote entre les accepteurs, à majorité
- Utilisation de numéros de requêtes : horloges

Phase 1

- Un proposeur envoie une requête *prepare* portant un numéro n à un ensemble d'accepteurs (au moins la majorité)
- Si un accepteur reçoit une requête *prepare* portant un numéro n supérieur à toutes celles auxquelles il a répondu, alors le proposeur qui a émis cette requête promet qu'il n'acceptera pas de proposition avec un numéro inférieur à n

Paxos : décision de la valeur

Idée : vote parmi les accepteurs

- Le proposeur reçoit la réponse des accepteurs
- Il compte, et accepte la valeur la plus reçue / la plus récente

Phase 2

- Si un proposeur reçoit une réponse à sa requête *prepare* (numérotée n) d'une majorité d'accepteurs, alors il envoie une requête *accept* à chacun des accepteurs avec la valeur v, étant la valeur de la proposition au numéro le plus élevé des réponses reçues
- Si un accepteur reçoit une requête *accept* d'une proposition numérotée n, il l'accepte sauf si il a déjà répondu à une requête *prepare* de numéro plus élevé.

NB : un proposeur peut proposer plusieurs valeurs différentes.

Paxos : apprentissage de la valeur

Une fois la valeur décidée, les apprentis doivent **apprendre la valeur**

- savoir que les accepteurs se sont mis d'accord
- découvrir sur quelle valeur

Si les apprentis demandent aux proposeurs :

- Problème si un proposeur tombe en panne

Les proposeurs informent les apprentis

- Naïf : tous les proposeurs informent tous les apprentis
  - Coût de communications élevé !
  - Le plus rapide qui soit : les apprentis apprennent par le proposeur le plus rapide
- Protocole hiérarchique : les proposeurs informent un ensemble d'apprentis qui eux-mêmes informent les autres apprentis
  - Communications complexes
  - Tolère des pannes à chaque niveau

Paxos : terminaison

Paxos termine **si les accepteurs tiennent assez longtemps**

- Nécessité d'avoir un vote parmi les accepteurs
- Nécessité que les proposeurs aient la dernière réponse majoritaire

Ici on contourne le résultat d'impossibilité par :

- Le système d'horloge logique, qui ordonne les votes
- Une élection fiable
- Une hypothèse sur la survie des accepteurs (au moins un vit assez longtemps)
- Une hypothèse sur la survie des proposeurs

Plan du cours

Modèles de pannes

Pannes sur les **canaux de communications**

- Omission, duplication, création de messages

Pannes sur les **processus**

- Crash : modèle fail/stop, le processus fonctionne correctement jusqu'à ce qu'il s'arrête totalement et définitivement de fonctionner

Classification de **durée** :

- Pannes transitoires ou définitives

Erreurs Byzantines

- Tout ce qui précède est possible
- Le processus a l'air de bien se comporter, mais en fait non.

Ce qu'on peut faire en présence de pannes : dépend du type de pannes

- Par exemple : consensus possible avec N processus Byzantins si on a au moins $3N + 1$ **processus en tout** (Lamport, Shostak et Pease 1982)

Plan du cours

Définition et modèle

Problème : détecter quels processus sont en panne

- Ici : pannes franches
- Modèle asynchrone : pas de possibilité d'utiliser un timeout

On veut en fait :

- Savoir qui est mort et qui est vivant
- Que tous les processus vivants le sachent

C'est donc un **problème de consensus** $\rightarrow$ impossible

Propriétés recherchées :

- **Complétude** : tout processus en panne finit par être suspecté.
- **Correction** : aucun processus vivant n'est suspecté.

On ne **peut pas avoir un détecteur de défaillances parfait** .

Fonctionnement

Les processus maintiennent deux listes :

- Les processus vivants
- Les processus morts

On se ramène à un problème de **Group Membership Service**

- Accord entre les processus sur l'appartenance ou non d'un ensemble de processus à un groupe

En réalité : hypothèses sur l'appartenance à ces listes

- On *finit* par avoir une vision réelle

Détection des défaillances par battements et timeout

Chandra, Toueg 1991

Chaque processus surveille un **ensemble de voisins**

- Topologie choisie : dépend des propriétés réseau
- Doit être robuste mais performante

Chaque processus envoie périodiquement un message à ses voisins

- **Heartbeat**

Fonctionnement avec un **timeout** :

- Si on ne reçoit pas de heartbeat d'un voisin pendant un certain temps, on le considère comme mort
- Si on a à nouveau signe de vie, on le remet dans la liste des vivants

On prévient les autres processus qu'il est mort

- Possibilité d'utiliser un deuxième timeout ici
  - On déclare quand on le suspecte d'être mort depuis un certain temps
- Synchronisation logique dans la diffusion de l'information de sa mort
  - On n'est plus dans du "vrai" asynchrone

Détection des défaillances sans timeout

Aguilera, Chen, Toueg 1997

Chaque processus **compte les heartbeats** reçus de ses voisins

- Comparaison entre les compteurs des différents processus surveillés
- Quand le compteur d'un processus cesse d'augmenter alors que les autres augmentent, on le considère comme mort
  - Détection de la panne par création d'un décalage avec les autres

Même fonctionnement, avec les hypothèses que les communications sont fiables

Plan du cours

Plan du cours

Retour arrière sur point de reprise

Point de reprise (*checkpoint*) :

- Enregistre l'état d'un processus
- État de la mémoire (pile et tas), registres, program counter

Exemples : BLCR, SGI cpr, Condor...

On **enregistre l'état d'un processus**

- Possibilité de **redémarrer** dans l'état dans lequel on était

→ Migration, **redémarrage**

Tutoriel BLCR :
http://www-lipn.univ-paris13.fr/ coti/download.php?file=tutoblcr.pdf

Piecewise Deterministic Assumption

L'exécution d'un système distribué est, dans le cas général, **non-déterministe**

- Circulation de messages non-ordonnés causalement...

Pris individuellement, l'exécution d'un processus est non-déterministe

- Interactions avec le reste du système
- Réception de messages venant d'autre processus

On considère l'exécution des processus d'un système distribué comme **déterministe par morceaux**

- L'exécution de chaque processus est une **succession de segments déterministes**
- Les segments déterministes sont **séparés par des évènements non-déterministes**
  - Concrètement : interactions avec le reste du système

Application à la tolérance aux pannes

Tolérance aux pannes par retour arrière :

- Prise de checkpoints réguliers
- Stockage sur un support sûr
- Redémarrage sur un checkpoint sauvegardé : **retour arrière sur checkpoint**

Retour arrière d'un processus :

- Problème : les évènements non-déterministes
- Demander de les **rejouer** ?

→ Effet domino

État cohérent

État cohérent d'un système distribué

- État dans lequel le système n'a pas de message en attente
  - Envoi de message sans réception du destinataire
  - Attente de réception d'un message qui ne sera pas envoyé
- Processus en attente d'un message qui ne sera jamais renvoyé : **processus orphelin**

Problème : comment assurer la cohérence de l'état après une panne ?

- Retour arrière sur checkpoint après la panne
- Restauration de la cohérence de l'état

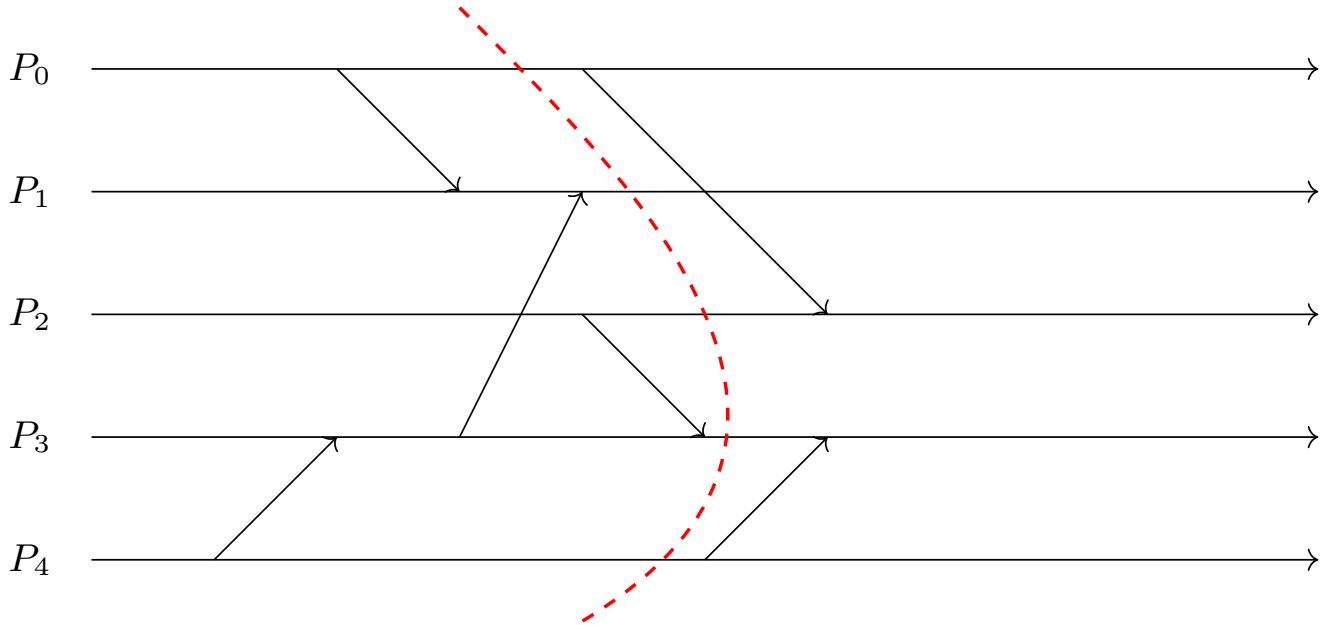
Plan du cours

État cohérent

Chandy & Lamport 1985 : enregistrement de l'état global du système

- Définition d'une **coupe cohérente**
- Coupe qui n'est pas traversée par l'envoi ou la réception d'un message

Définition :
Un état est dit **cohérent** si pour tous les messages m d'un processus P_i vers P_j , si le point de reprise sur P_j a été effectué après la réception de m alors le point de reprise de P_i a été effectué après l'émission de m.



Checkpoint distribué

État après retour arrière sur checkpoint coordonné : **ligne de recouvrement**

- Pour que l'état soit cohérent, aucun message ne doit franchir la ligne de recouvrement

Le **checkpoint distribué** de Chandy & Lamport constitue un état cohérent

- Problème : comment enregistrer un état cohérent ?
- Idée de base : s'assurer qu'aucun message ne coupe la ligne de recouvrement

Checkpoint distribué

- Enregistrement de l'état global du système
- Enregistrement de l'état de chaque processus du système
- Problème : messages en transit entre les processus ?

Prise de checkpoint distribué

Un processus est l' **initiateur**

- **Vague** de checkpoint
- Circulation d'un marqueur entre les processus

On parle de **checkpoint coordonné**

Chaque processus qui reçoit le marqueur

- Prend un checkpoint
- Envoie le marqueur aux autres processus

Tous les processus reçoivent le marqueur, tous les processus prennent un checkpoint.

Problème : **que faire des messages envoyés pendant la vague de checkpoint ?**

Checkpoint coordonné bloquant

Les canaux ont la propriété FIFO

- Aucun message ne peut doubler le marqueur

Prise de checkpoint **une fois qu'on a reçu le marqueur de tous les processus**

- Pas de messages envoyés pendant la vague de checkpoint
- Les messages envoyés avant la prise de checkpoint des processus P_i sont tous reçus avant la prise de checkpoint des processus P_j

On **bloque** les communications pendant la vague de checkpoint

Checkpoint coordonné non-bloquant

Les messages envoyés avant la vague de checkpoint et reçus après peuvent être **enregistrés**

- Enregistrés avec le checkpoint
- En cas de retour arrière : ils sont **rejoués**

On ne **bloque pas** les communications pendant la vague de checkpoint

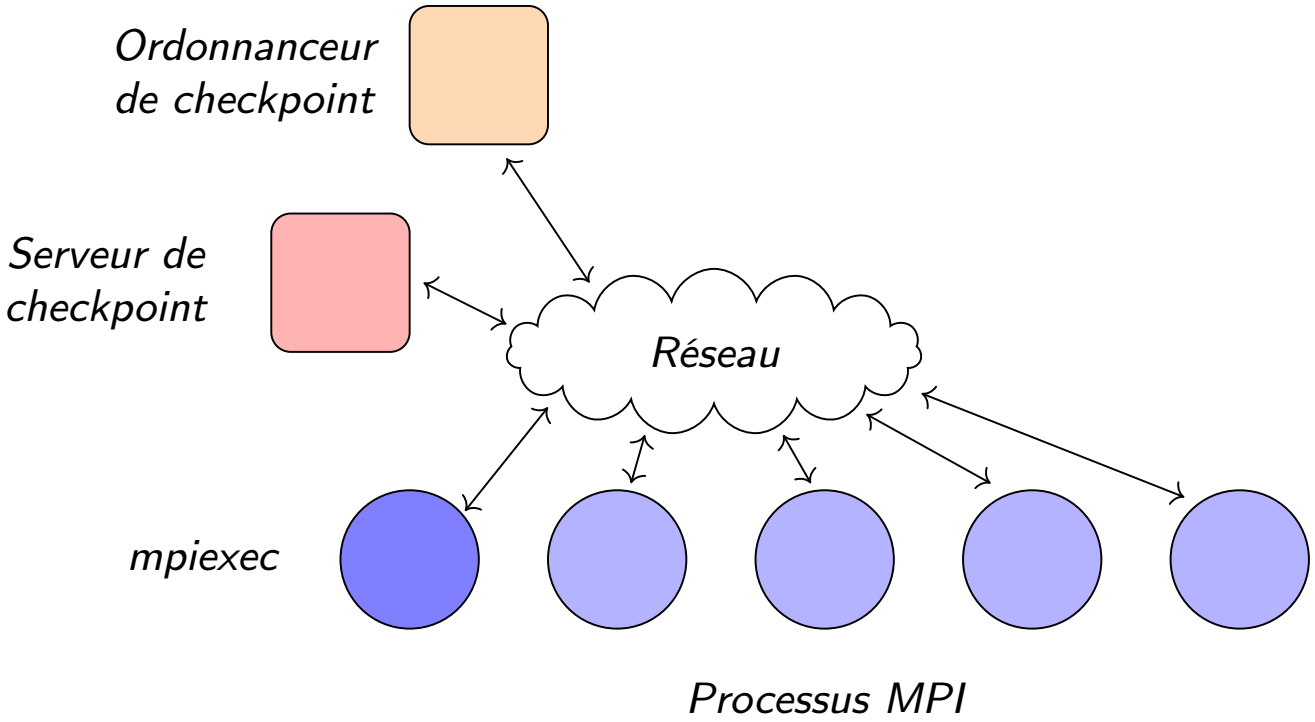
- Enregistrement des messages pendant la vague de checkpoint
- Prise de checkpoint à la fin

Utilisation ici encore de la propriété FIFO des canaux

- Aucun message ne peut doubler le marqueur
- Tous les messages qui traversent la ligne de recouvrement sont enregistrés

Implémentation pour MPI

- Ordonnanceur de checkpoint : décide quand la vague de checkpoint démarre
- Serveur de checkpoint : stockage stable



Comparaison : bloquant vs non bloquant

Comparaison des protocoles

- Bloquant : plus simple
- Non-bloquant : n'arrête pas les communications, et donc l'exécution

Comparaison des performances

- Implémentation dans MPICH, 2006
- Non-bloquant : l'enregistrement des messages ralentit les communications
- L'implémentation bloquante est plus rapide

Plan du cours

Checkpoint non coordonné

Checkpoint **non coordonné** : évite la vague de checkpoint

- Chaque processus prend ses checkpoints de façon indépendante
- Ne synchronise pas les processus

Problème : comment maintenir la cohérence de l'état ?

- **Journalisation des messages** (*message logging*)
- Après retour arrière : les messages sont rejoués
- Dans le **même ordre** ! $\rightarrow$ informations de causalité

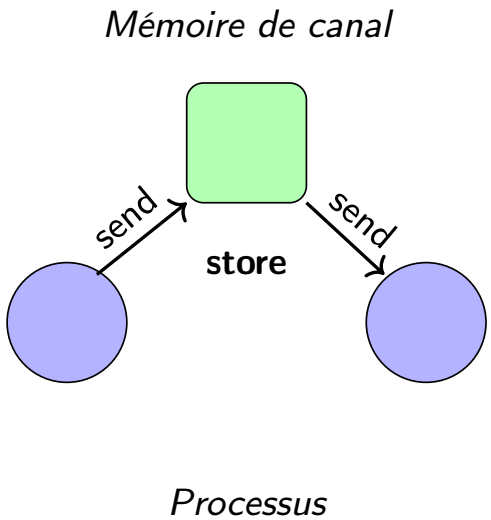
Mémoire de canal

Idée la plus simple : mise en place d'une **mémoire de canal**

- Enregistre tout ce qui passe sur le canal de communication

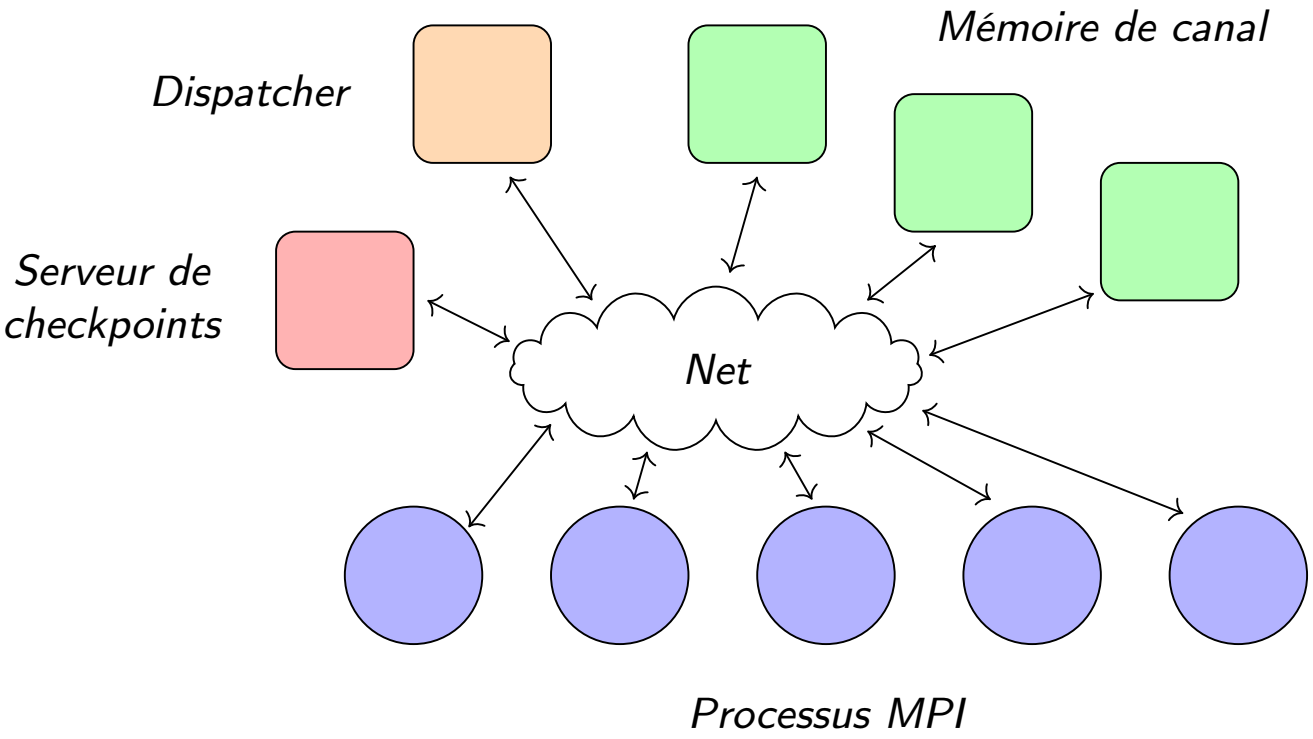
Les messages envoyés passent par la mémoire de canal

- Intermédiaire entre les deux processus
- Problème de passage à l'échelle
- Latence !



Mémoire de canal : implémentation pour MPI

- Utilisation de plusieurs mémoires de canal
- Serveur de checkpoints
- Dispatcher pour déclencher les checkpoints



Enregistrement sur un serveur

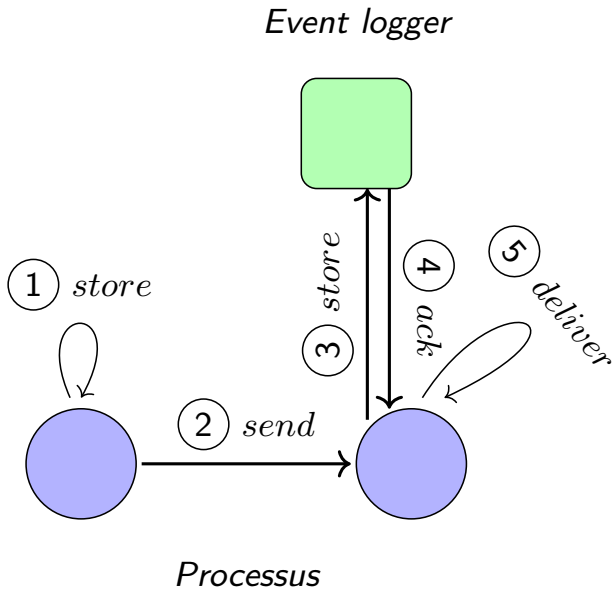
Enregistrement **local** des messages

- Enregistrés de façon fiable lors du checkpoint suivant
- Enregistrement des **informations de causalité** sur un serveur

Informations de causalité : **déterminant**

- Contient les horloges de l'émetteur et de la source
- Horloges de Lamport
- Permet de rejouer les messages dans l'ordre dans lequel ils ont été envoyés et reçus

$$[id\_source, id\_dest, H_s, H_d]$$



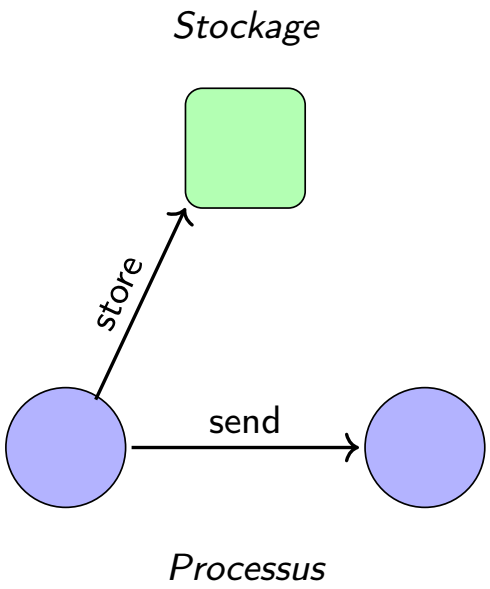
Enregistrement sur un serveur

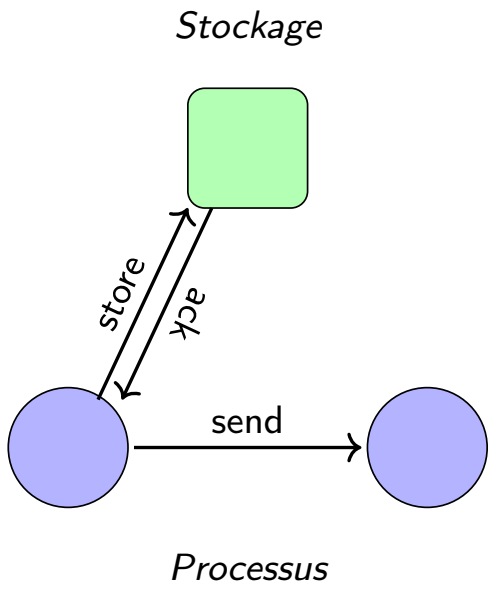
Problème : attente de l'acquittement

- Introduit de la latence !

Deux options :

- Stockage **optimiste** : on considère qu'aucune panne n'arrivera entre l'envoi du message et son stockage sûr
- Stockage **Pessimiste** : une panne peut arriver n'importe quand





Utilisation des messages pour transmettre des informations

Si un message est reçu sur P_j depuis P_i, alors P_j dépend causalement de P_i

- En cas de panne, ce message peut être orphelin : il a besoin d'être journalisé
- Les informations de causalité de cette interaction doivent donc être sauvegardées

Idée : **utiliser les messages pour conserver l'information**

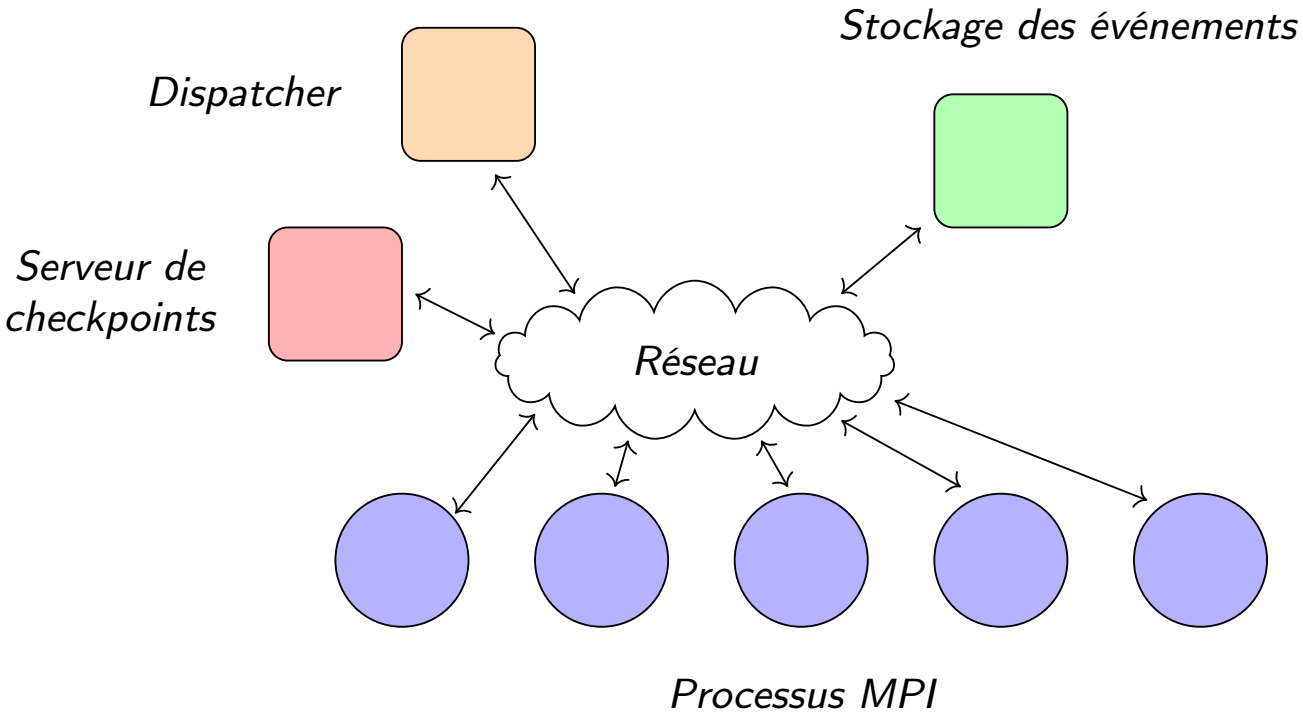
- On note d_{ij} le déterminant correspondant à la réception sur P_j d'un message en provenance de P_i
- Quand P_j envoie un message à P_k, il envoie aussi d_{ij} avec le message (*piggyback*)

Pas besoin d'attendre l'acquittement

- Les déterminants sont accumulés en queue des messages
- Élimination : lorsque le stockage est acquitté

Implémentation pour MPI

- Serveur de checkpoint : stockage stable
- Dispatcher pour déclencher les checkpoints
- Enregistrement des événements



Comparaison des performances

Vague de checkpoint : lourd

- **Synchronise** tous les processus
- Passe mal à l'échelle

Journalisation des messages : **pénalise les communications**

- Coût en latence
  - Variable selon le protocole
- Coût en bande passante (copie)

En cas de pannes :

- Checkpoint coordonné : fait repartir tous les processus en arrière
- Checkpoint non-coordonné : un seul processus repart en arrière
  - Synchronisations ultérieures ?

En-dehors des pannes :

- Checkpoint non-coordonné : pénalisation sur les communications
- Checkpoint coordonné : coût uniquement pendant la vague de checkpoint

Plan du cours

Critères pour la réplication

Réplication : utilisation de plusieurs ressources (les **réplicas**) qui maintiennent une copie de l'état à sauvegarder (processus, donnée...)

- En cas de panne, un replica est utilisé
- Aucune donnée n'est perdue

Linéarisabilité

- Donne au client l'impression qu'il n'y a pas de réplication
- Vu du client : une seule ressource

Déterminisme

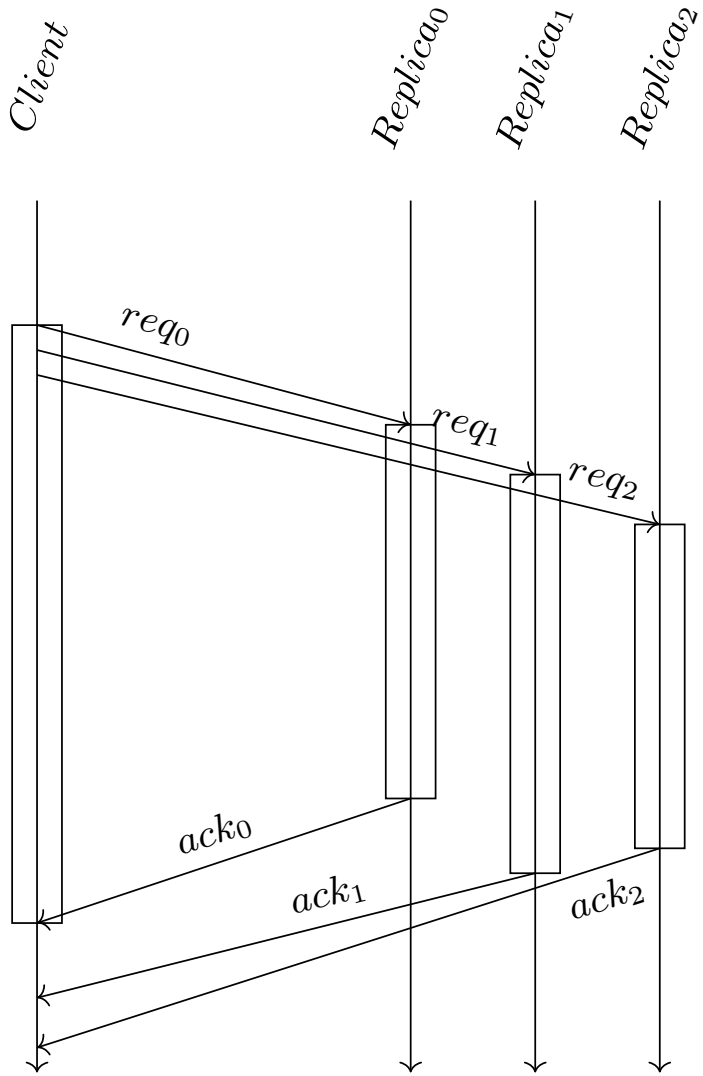
- L'issue de l'opération ne dépend que de l'état initial et de la séquence d'événements qui ont eu lieu.

Réplication active

Le client effectue lui-même la réplication

C'est le client qui s'occupe de redonder la donnée

- Il diffuse la donnée parmi les réplicas
- Si on considère la diffusion comme une opération abstraite : la linéarisabilité est préservée
- Sinon : il n'y a pas linéarisabilité
- Déterminisme : ok



Réplication par primaire/backup

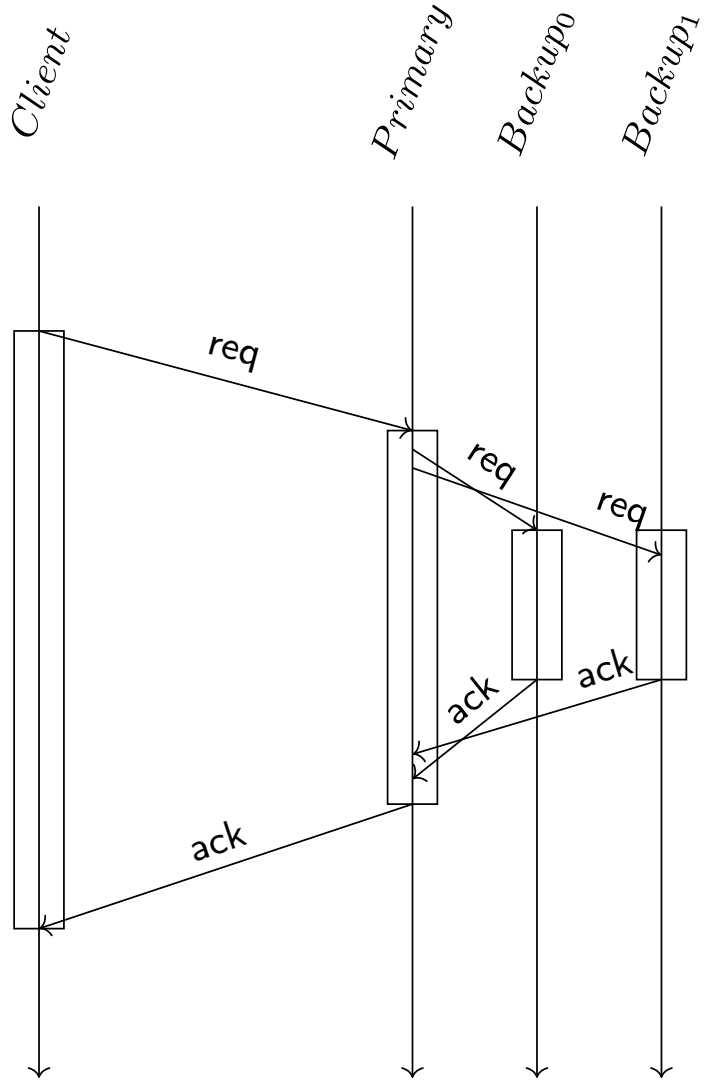
Utilisation d'un serveur primaire qui copie sur des replicas

Transparent pour le client

- ... tant que la copie primaire ne tombe pas en panne !
- En cas de panne : un replica devient primaire

C'est le serveur (primaire) qui s'occupe de la réplication.

- Linéarisabilité : ok
- Déterminisme : non (panne du serveur primaire)



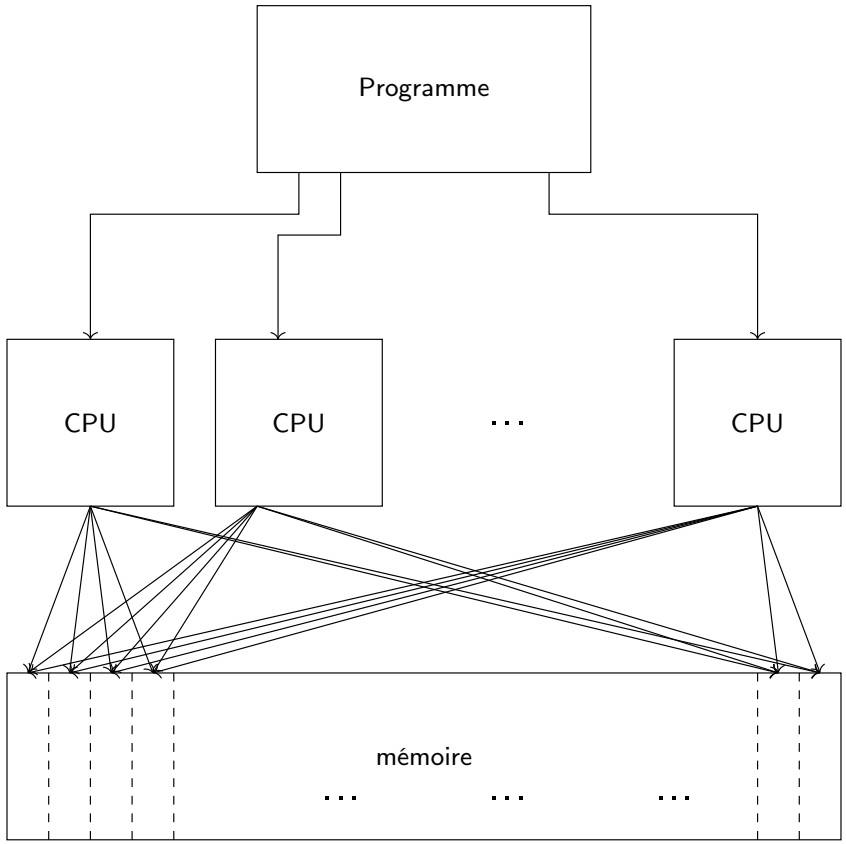
Plan du cours

Plan du cours

Mémoire partagée : PRAM

Mémoire partagée

- Unique banc mémoire
- Plusieurs processeurs de calcul
- Les processeurs accèdent tous à la mémoire



Caractéristiques de PRAM

Processeurs

- Nombre de processeurs fini, chacun connaissant son indice
- Tous les processeurs exécutent la **même** instruction
- ... dans la **même** unité de temps

Accès mémoires

- Tous les processeurs peuvent accéder à la mémoire
- Temps d'accès uniforme

C'est un modèle **théorique** de machine parallèle

- Déterministe : une seule exécution possible
- Synchrone : tous les processeurs exécutent la même instruction en même temps.

5 nuances de PRAM

Différents **modes d'accès mémoire**

EREW (exclusive read exclusive write)

Sur une case donnée, *un seul processus peut lire et écrire* à un moment donné

CREW (concurrent read exclusive write)

Sur une case donnée, *plusieurs processus peuvent lire* mais *un seul peut écrire* à un moment donné

CRCW (concurrent read concurrent write)

Plusieurs processus peuvent lire et écrire en même temps sur la même case

- mode arbitraire : tous les processus qui écrivent sur la même case écrivent la même valeur
- mode consistant : la dernière valeur écrite est prise en compte
- mode association/fusion : une fonction est appliquée à toutes les valeurs écrites simultanément (max, somme, XOR...)

Mise en œuvre

Rappel : c'est un modèle **théorique** de machine parallèle

Architectures s'en rapprochant : mémoire partagée

Machine multi-cœur

- Chaque cœur exécute une instruction différente
- Temps d'accès mémoire non-uniformes : NUMA

Processeur vectoriel

- Chaque cœur exécute la même instruction
- Des registres stockent les vecteurs de données (processeurs vectoriels *Load-Store*

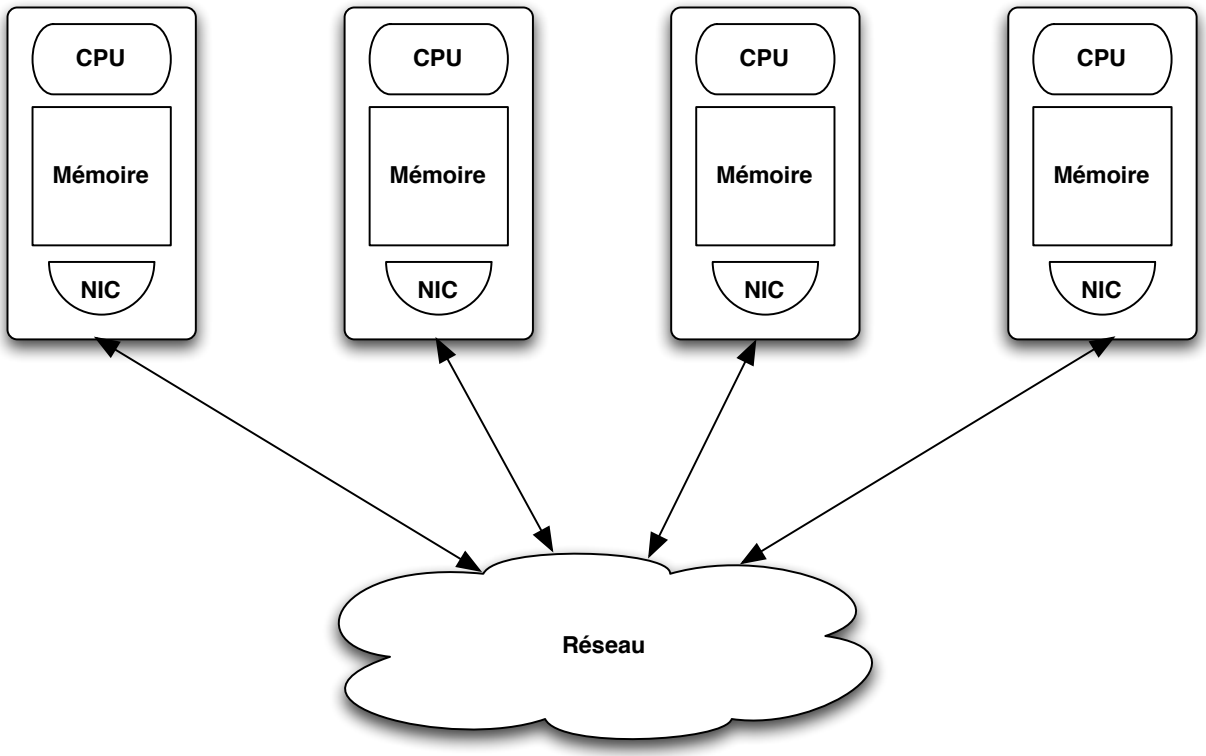
GPU

- Les cœurs sont répartis en *warps* (archi Fermi : 32 threads par warp)
- Chaque cœur d'un *warp* exécute la même instruction dans un *timestep*
- Accès à une mémoire commune

Mémoire distribuée

Nœuds de calcul distribués

- Chaque nœud possède un banc mémoire
- Lui seul peut y accéder
- Les nœuds sont reliés par un *réseau*



Mise en œuvre

Réseau d'interconnexion

Les nœuds ont accès à un *réseau d'interconnexion*

- Tous les nœuds y ont accès
- Communications point-à-point sur ce réseau

Espace d'adressage

Chaque processus a accès à sa mémoire propre *et uniquement sa mémoire*

- Il ne *peut pas* accéder à la mémoire des autres processus
- Pour échanger des données : communications point-à-point
  C'est au programmeur de gérer les mouvements de données entre les processus

Système d'exploitation

Chaque nœud exécute sa propre instance du système d'exploitation

- Besoin d'un middleware supportant l'exécution parallèle
- Bibliothèque de communications entre les processus

Avantages et inconvénients

Avantages

- Modèle plus réaliste que PRAM
- Meilleur passage à l'échelle des machines
- Pas de problème de cohérence de la mémoire

Inconvénients

- Plus complexe à programmer
  - Intervention du programmeur dans le parallélisme
- Temps d'accès aux données distantes

Taxonomie de Flynn

Classification des modèles

- Suivant les données
- Suivant les instructions

- **SISD** (Single Instruction, Single Data)
  - Modèle séquentiel, architecture de Von Neumann
- **SIMD** (Single Instruction, Multiple Data)
  - Processeur vectoriel
- **MISD** (Multiple Instruction, Single Data)
  - Peu d'implémentations en pratique, systèmes critiques (redondance)
- **MIMD** (Multiple Instruction, Multiple Data)
  - Plusieurs unités de calcul, chacune avec sa mémoire

SISD	SIMD
MISD	MIMD

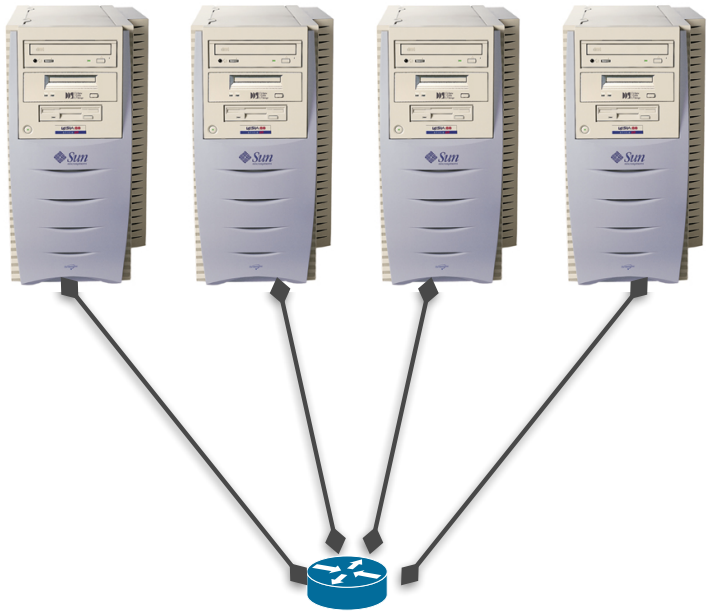
Plan du cours

Exemples d'architectures

Cluster of workstations

Solution *économique*

- Composants produits en masse
  - PC utilisés pour les nœuds
  - Réseau Ethernet ou haute vitesse (InfiniBand, Myrinet...)
- Longtemps appelé "le supercalculateur du pauvre"



Exemples d'architectures

Supercalculateur massivement parallèle (MPP)

Solution spécifique

- Composants spécifiques
  - CPU différent de ceux des PC
  - Réseaux spécifique (parfois propriétaire)
  - Parfois sans disque dur
- Coûteux

Exemples d'architectures

Exemple : Cray XT5m

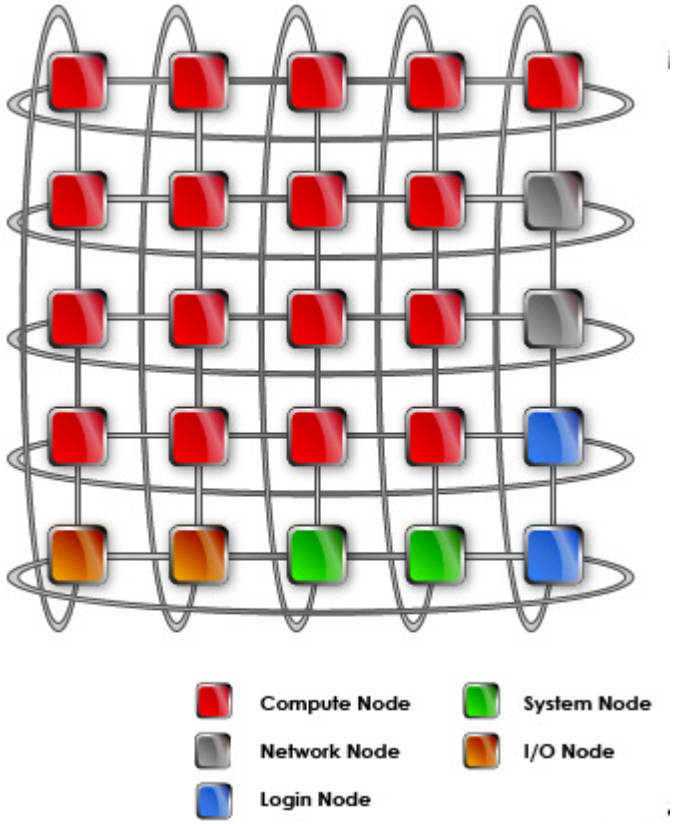
- CPU : deux AMD Istanbul
  - 6 cœurs chacun
  - 2 puces par machine
  - Empilées sur la même socket
  - Bus : crossbar
- Pas de disque dur

Réseau

- Propriétaire : SeaStar
- Topologie : tore 2D
- Connexion directe avec ses 4 voisins

Environnement logiciel

- OS : Cray Linux Environment
- Compilateurs, bibliothèques de calcul spécifiques (tunés pour l'architecture)
- Bibliothèques de communications réglées pour la machine



Top 500 www.top500.org

Classement des machines les plus rapides

- Basé sur un benchmark (LINPACK) effectuant des opérations typiques de calcul scientifique
- Permet de réaliser des statistiques
  - Tendances architecturales
  - Par pays, par OS...
  - Évolution !
- Depuis juin 1993, dévoilé tous les ans en juin et novembre

Dernier classement : novembre 2016

1. Sunway TaihuLight - Sunway MPP - National Supercomputing Center in Wuxi (Chine)
2. Tianhe-2 (MilkyWay-2) - NUDT - National Super Computer Center in Guangzhou
3. Titan - Cray XK7 - Oak Ridge National Lab
4. Sequoia - IBM BlueGene/Q - Lawrence Livermore National Lab, NNSA
5. Cori - Cray XC40 - Lawrence Berkeley National Laboratory, NERSC

Top 500 - Nombre de coeurs

Rang	Machine	Nb de coeurs	Rmax (TFlops)	Conso (kW)
1	Sunway	10 649 600	93 014.6	15 371
2	Tianhe-2	3 120 000	33 862.7	17 808
3	Titan	560 640	27 112.5	8 209
4	Sequoia	1 572 864	16 324.8	7 890
5	Cori	622 336	27 880.7	3 939

Anciens et actuels numéro 1 :

- Sunway : depuis juin 2016
- Tianhe-2 : de juin 2013 à novembre 2015
- Titan : numéro 1 en novembre 2012
- Sequoia : numéro 1 en juin 2012
- K : numéro 1 de juin à novembre 2011

Top 500 - Les numéro 1

Nom	Dates #1	Nb coeurs	Rmax
CM-5	06/93	1 024	59,7 Gflops
Numerical Wind Tunnel	11/93	140	124,2 Gflops
Intel XP/S 140 Paragon	04/93	3 680	143,40 Gflops
Numerical Wind Tunnel	11/94-12/95	167	170,0 Gflops
Hitachi SR2201	06/96	1 024	232,4 Gflops
CP-PACS	11/96	2 048	368,20 Gflops
ASCI Red	06/97 - 06/00	7 264	1,068 Tflops
ASCI White	11/00 - 11-01	8 192	4,9 - 7,2 Tflops
Earth Simulator	06/02 - 06/04	5 120	35,86 Tflops
BlueGene/L	11/04 - 11/07	212 992	478,2 Tflops
Roadrunner	06/08 - 06/09	129 600	1.026 - 1,105 Pflops
Jaguar	11/09 - 06/10	224 162	1,759 Pflops
Tianhe-1A	11/10	14 336 + 7 168	2.57 Pflops
K	06/11 - 11/11	548 352 - 705 024	8,16 - 10,51 Pflops
Sequoia	06/12	1 572 864	16,32 Pflops
Titan	11/12	552 960	17,6 Pflops
Tianhe-2	6/13 - 11/15	3 120 000	33,9 Pflops
Sunway	6/16 $\rightarrow$	10 649 600	93,0 Pflops

Plan du cours

Programmation parallèle

1er critère : **accès mémoire**

- Mémoire partagée ?
- Mémoire distribuée ?

→ *Dépend du matériel sur lequel on s'exécute*

2ème critère : **paradigme de programmation**

- Langage compilé ?
- Communications explicites ?
- Communications unilatérales ? Bilatérales ?

→ *Dépend du programmeur et du matériel*

Programmation sur mémoire partagée

Tous les threads ont accès à une mémoire commune
Utilisation de **processus**

- Création avec `fork()`
- Communication via un segment de mémoire partagée

Utilisation de **threads POSIX**

- Création avec `pthread_create()`, destruction avec `pthread_join()`
- Communication via un segment de mémoire partagée ou des variables globales dans le tas
  - Rappel : la pile est propre à chaque thread, le tas est commun

Utilisation d'un langage spécifique

- Exemple : **OpenMP, TBB, Cilk...**

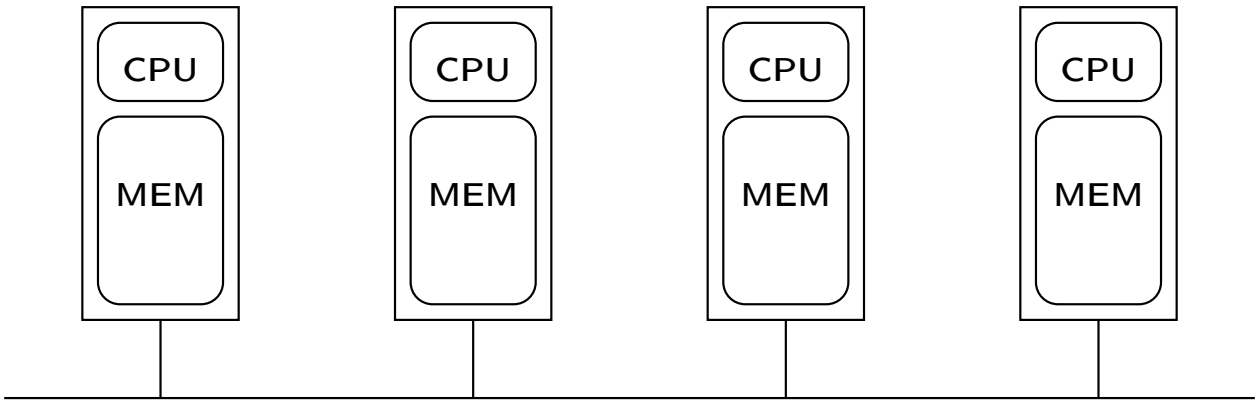
Programmation sur mémoire distribuée

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

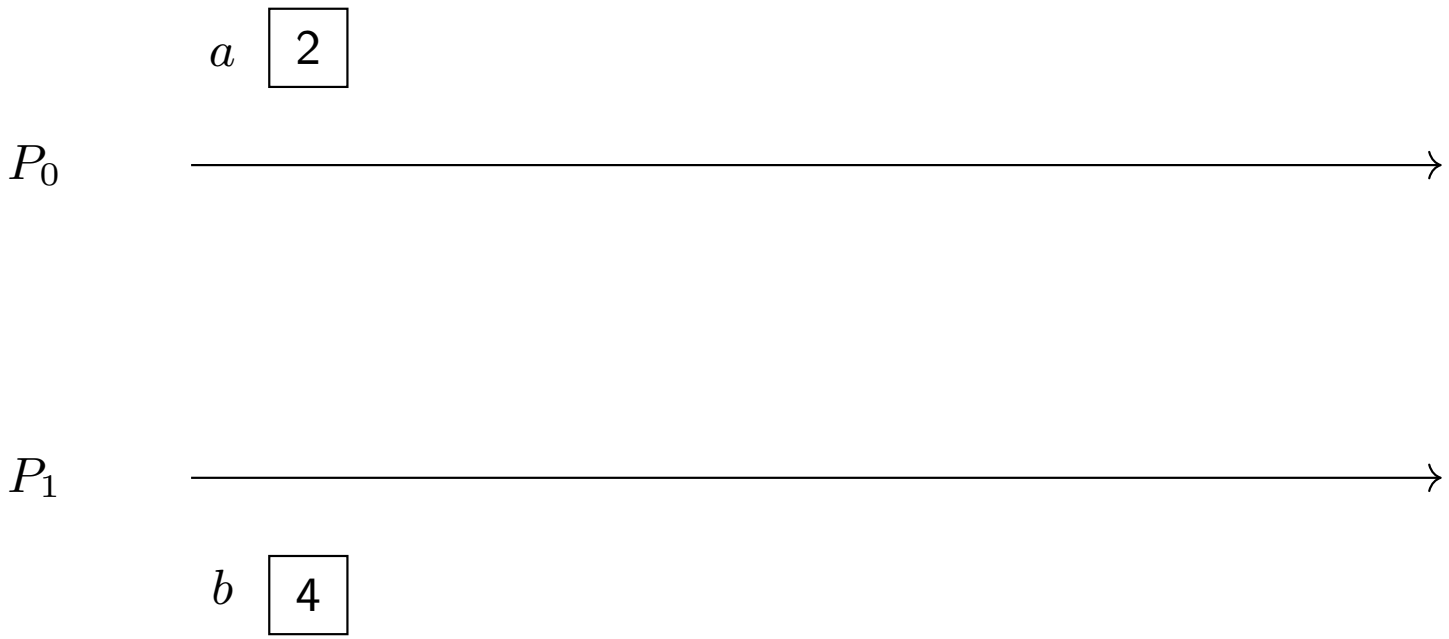


Communications bilatérales

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

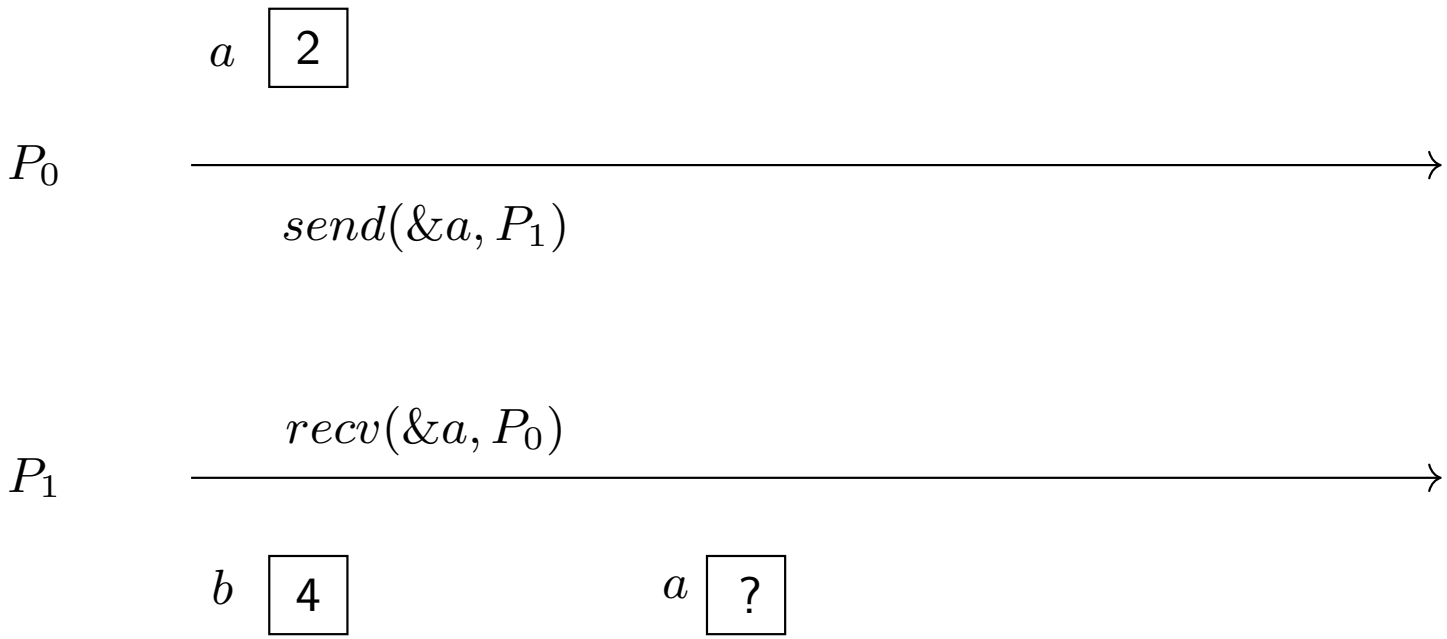


Communications bilatérales

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

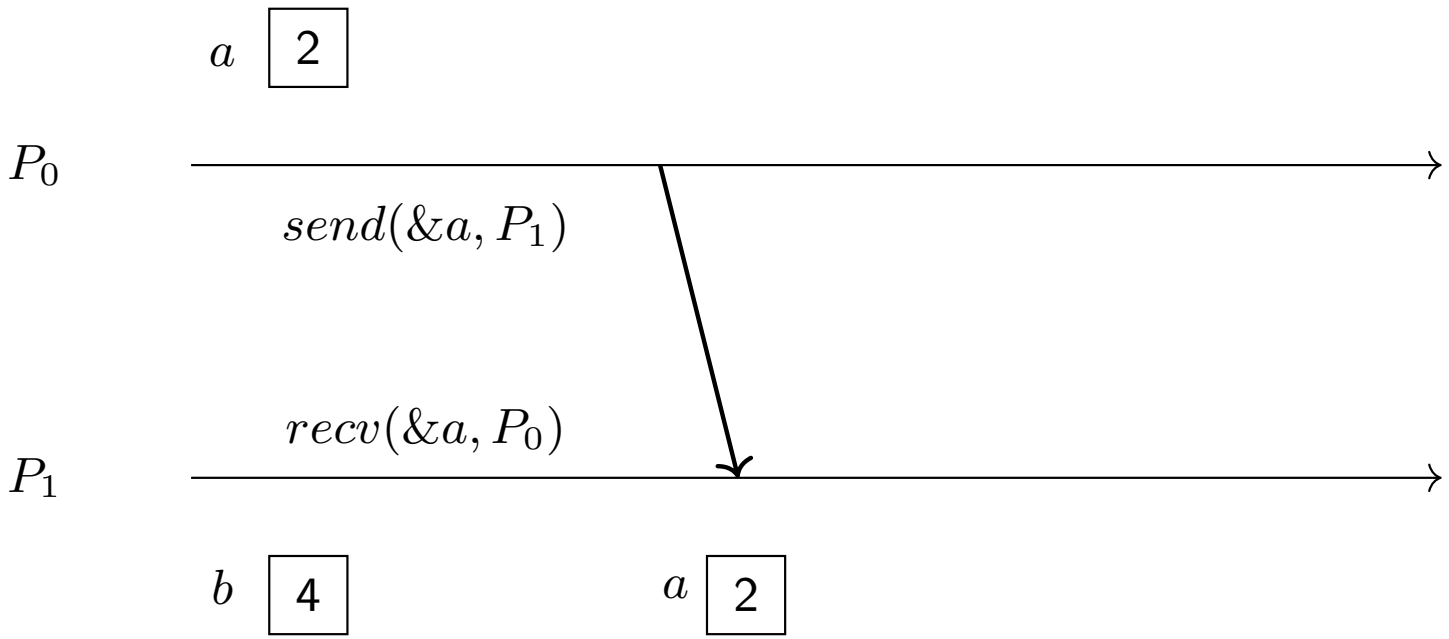


Communications bilatérales

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

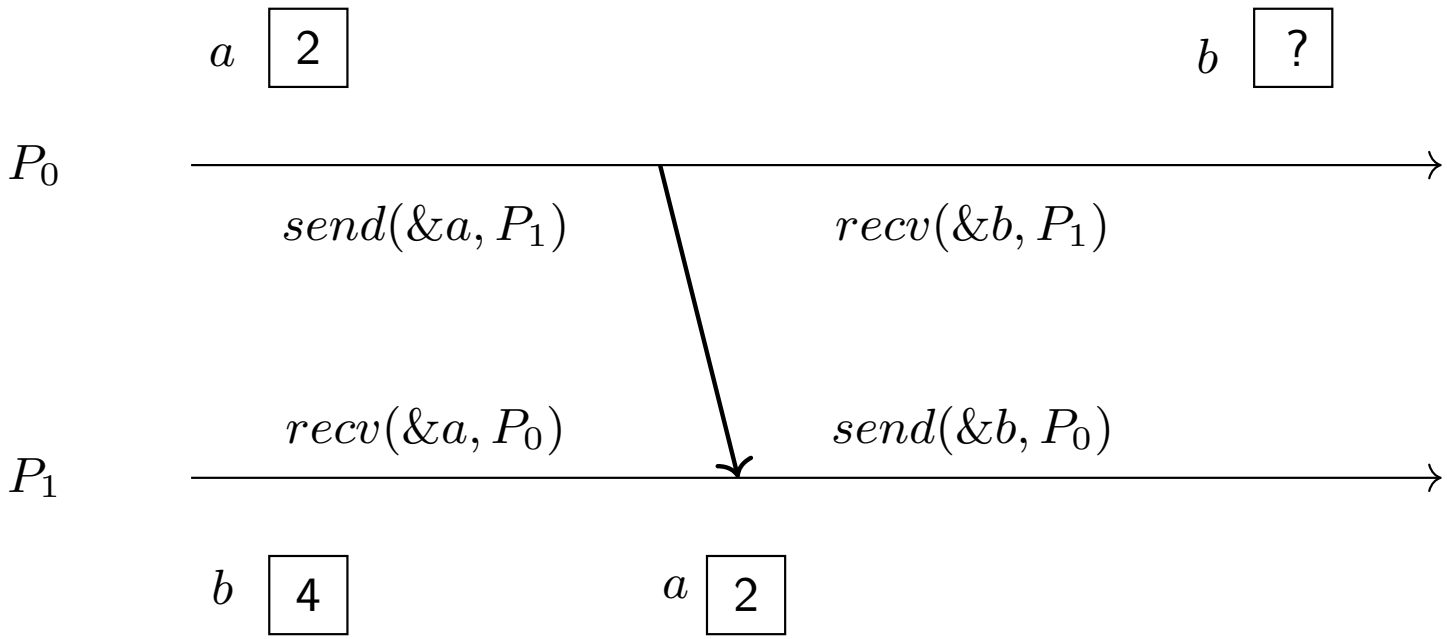


Communications bilatérales

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

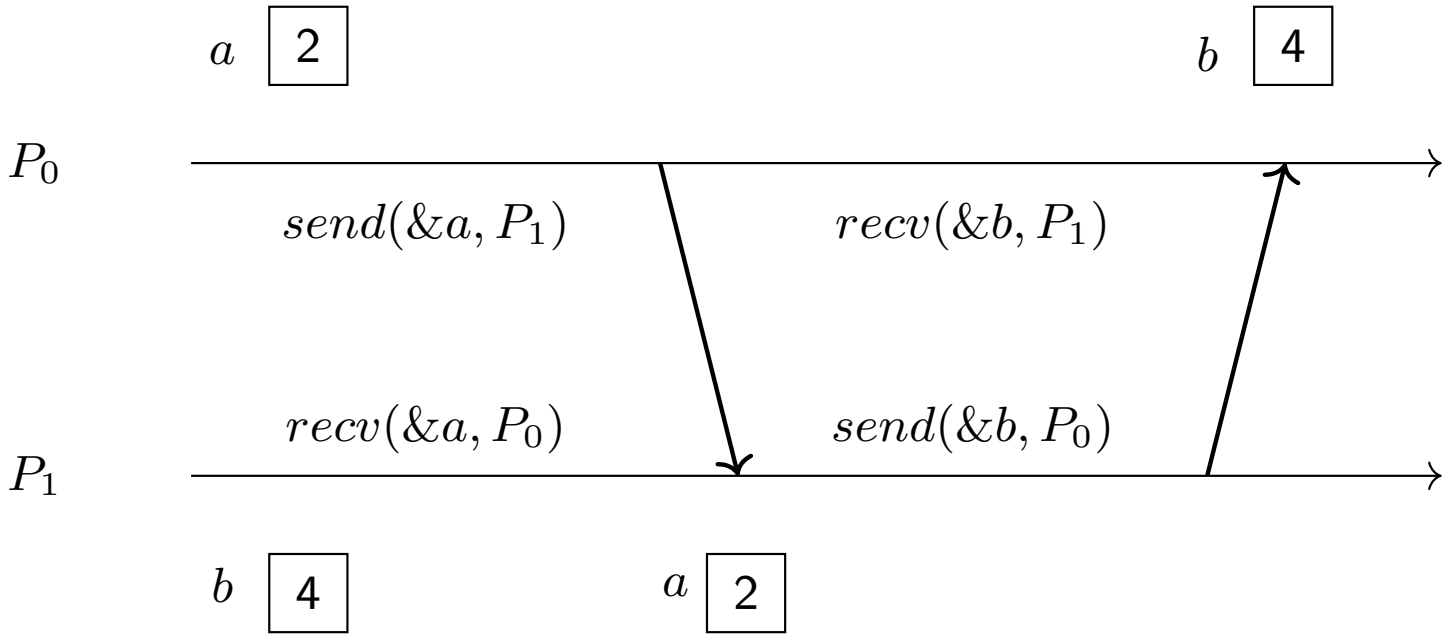


Communications bilatérales

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 :

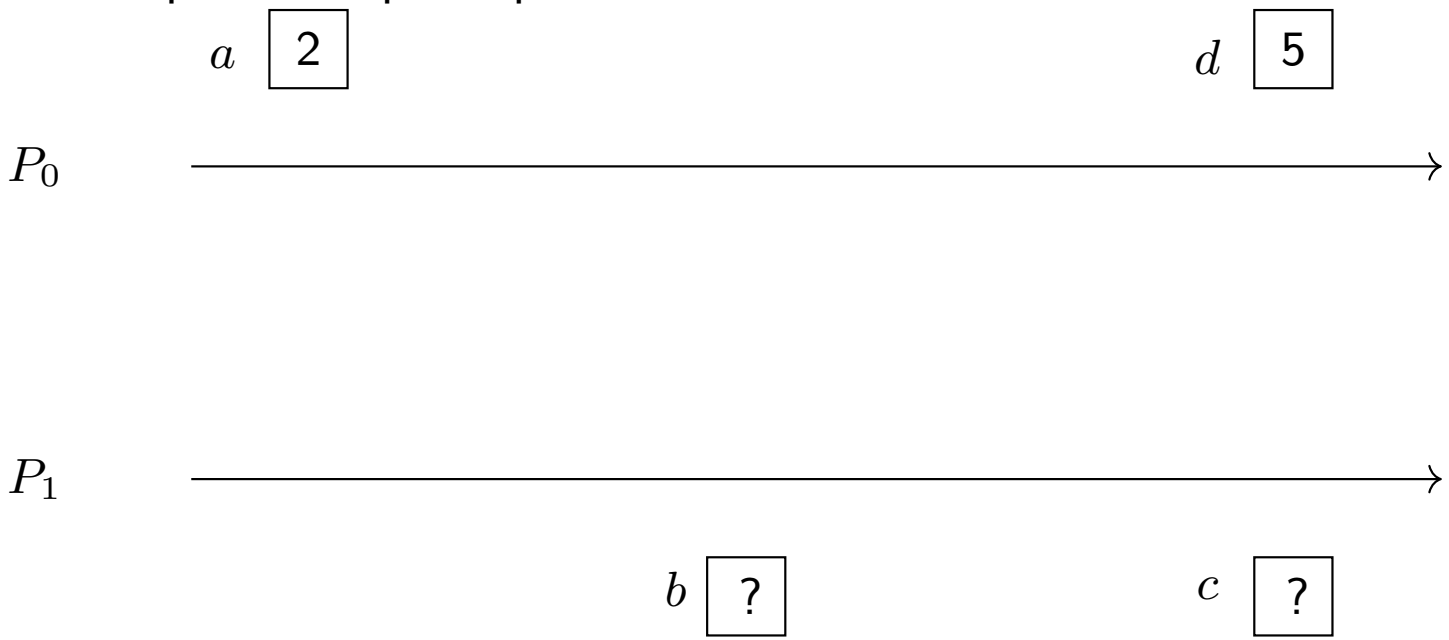
- Besoin de la coopération des deux processus : source et destination
- Fort synchronisme

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.

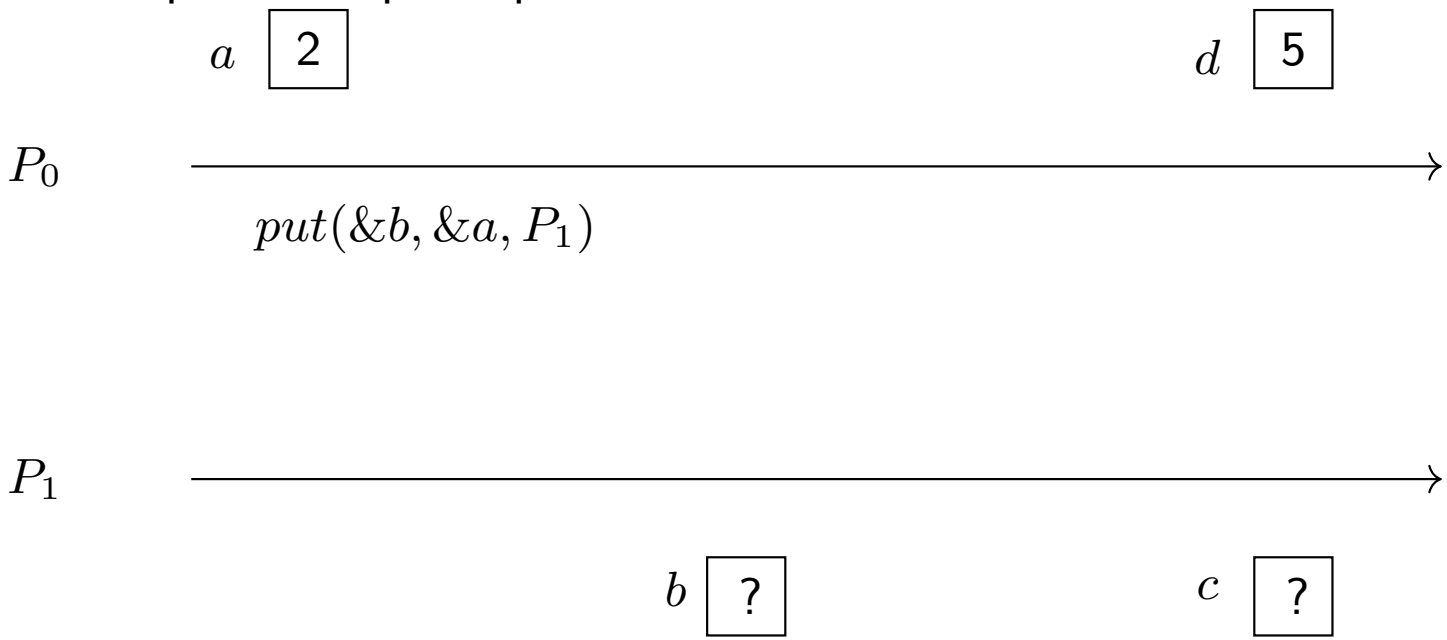


Communications unilatérales

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.

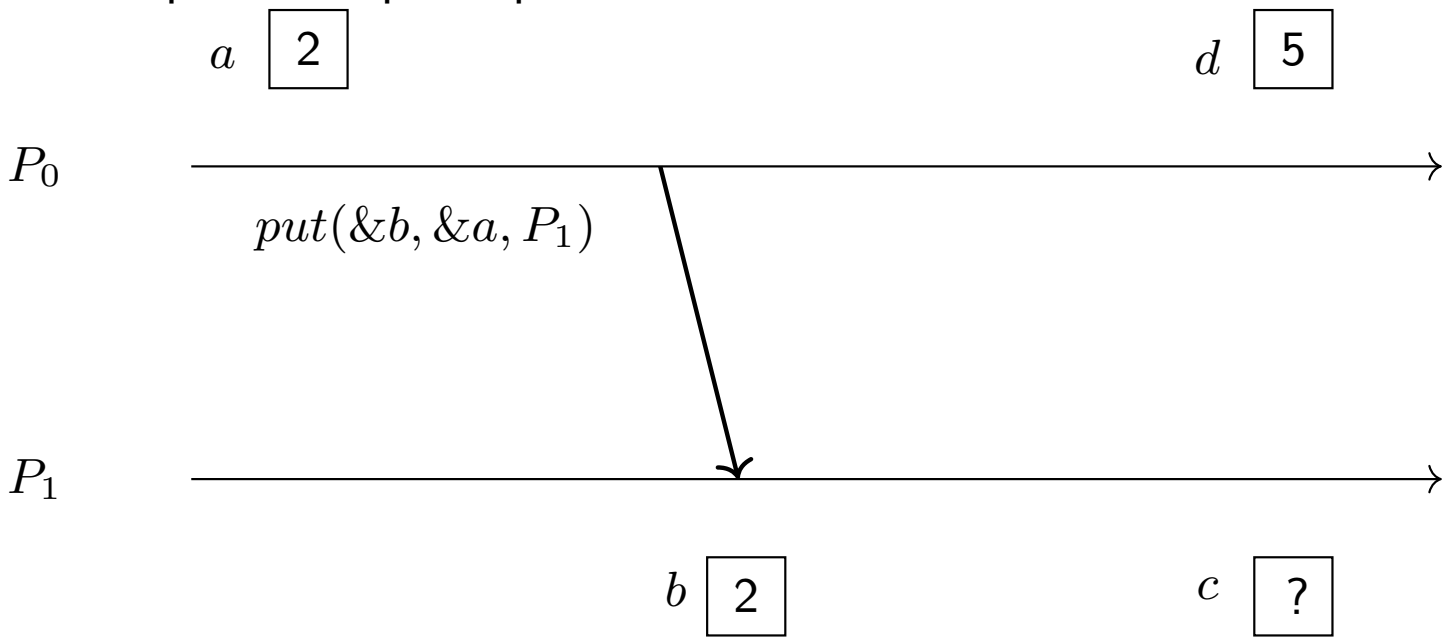


Communications unilatérales

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.

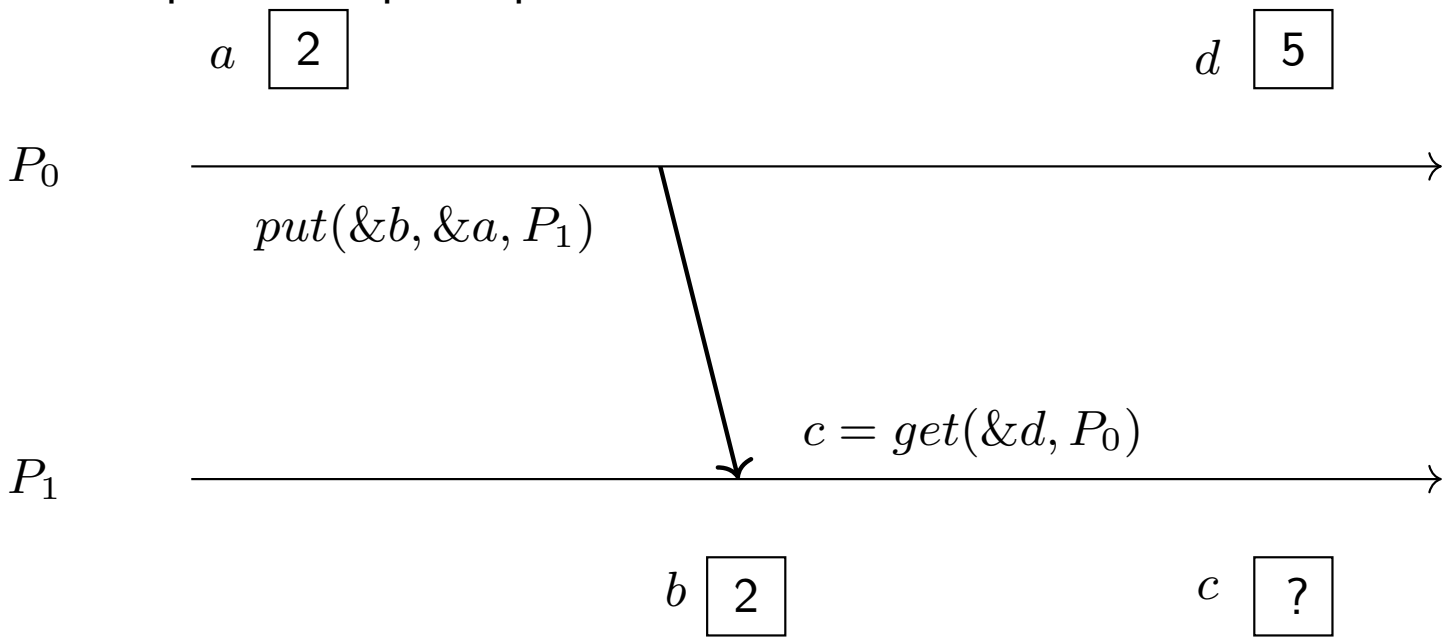


Communications unilatérales

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.

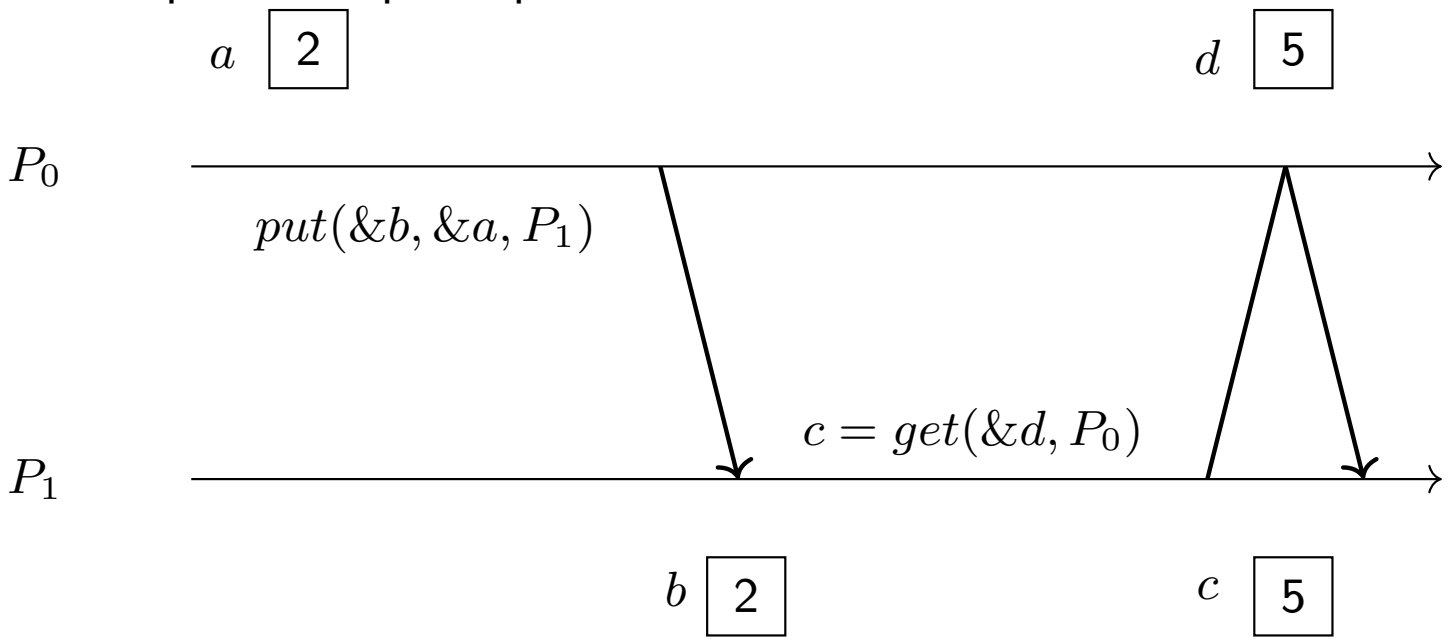


Communications unilatérales

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

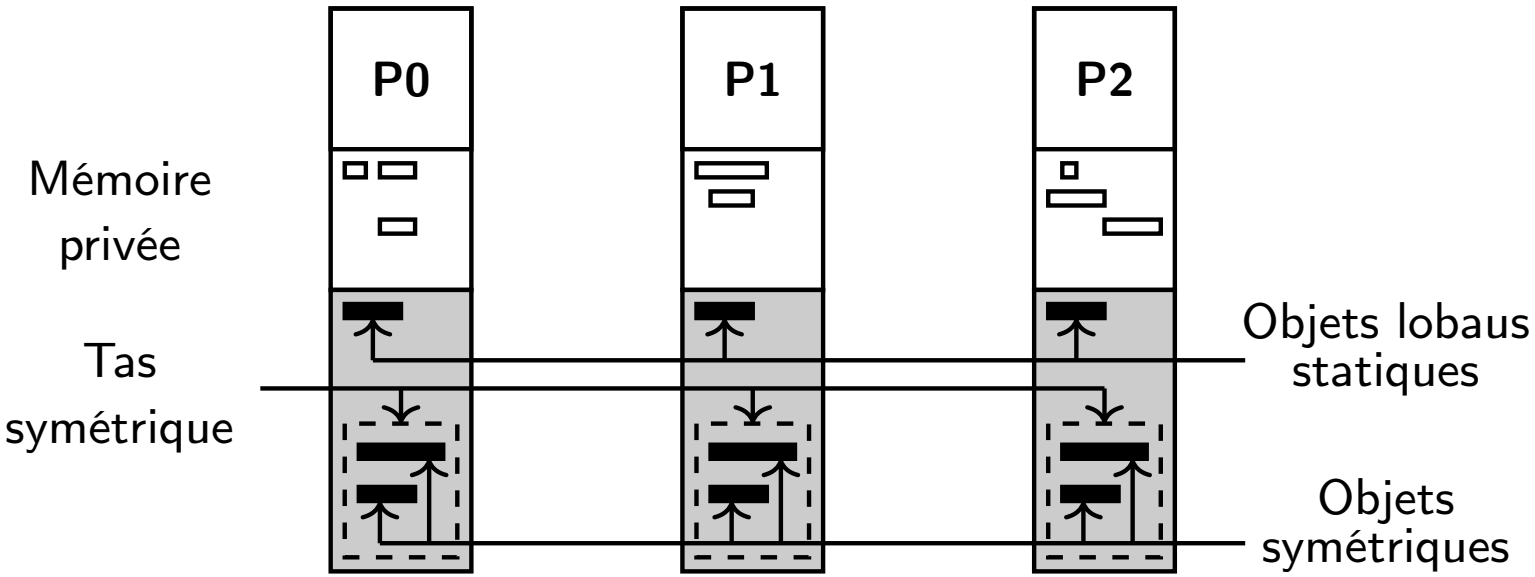
Inconvénients :

- 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*



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 :

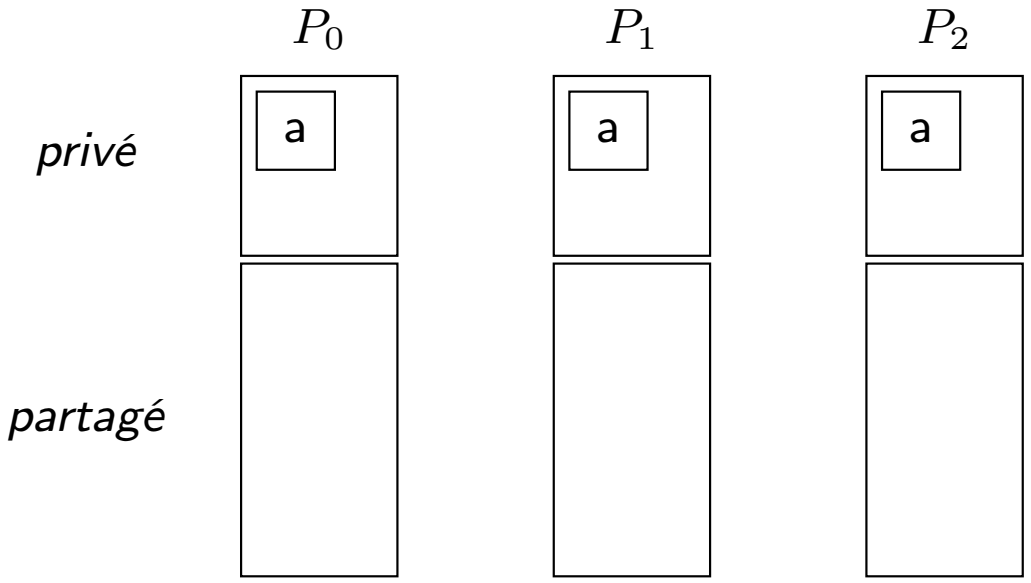
- Le programmeur déclare la **visibilité** de ses variables : privées (par défaut) ou **partagées**
- Pour les tableaux : le programmeur déclare la taille des blocs sur chaque processus
- Le compilateur se charge de
  - **répartir les données partagées** dans la mémoire des différents processus
  - **traduire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



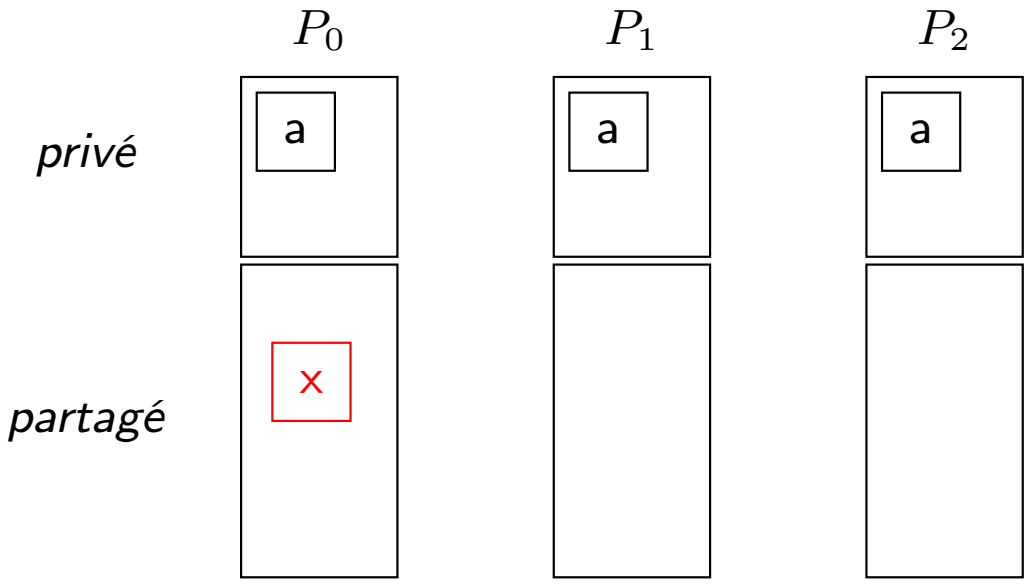
int a ;

shared int x ;

Exemples

Langages **PGAS** :

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



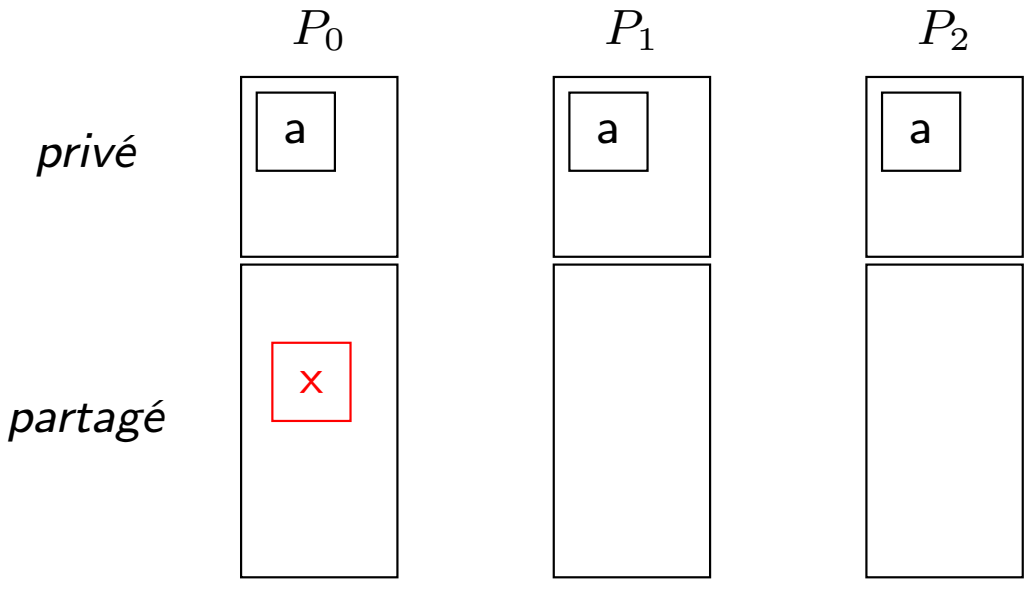
int a ;

shared int x ;

Exemples

Langages **PGAS** :

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

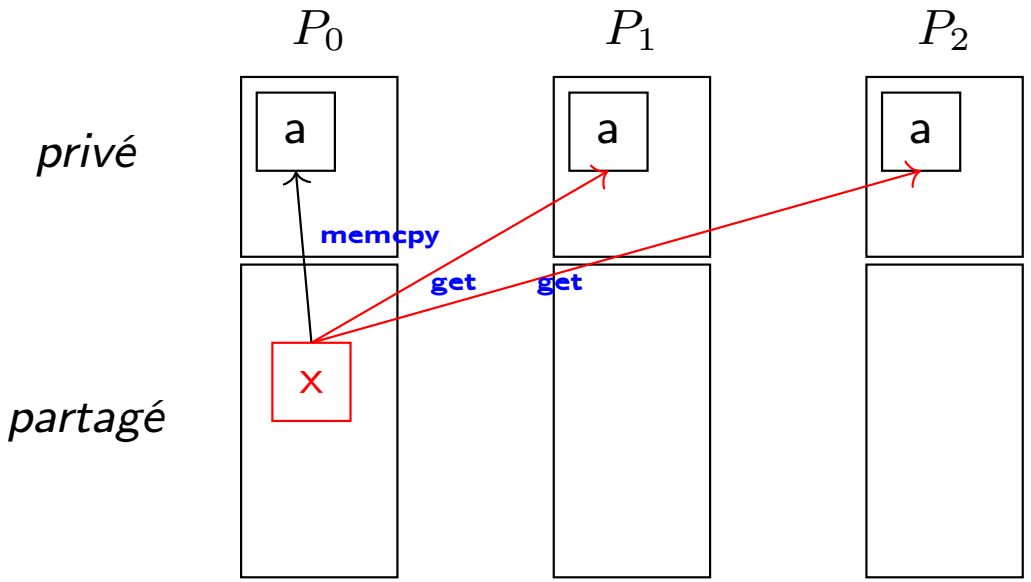



```
int a ;
shared int x ;
int a = x ;
```

Exemples

Langages **PGAS** :

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




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

Sac de tâches

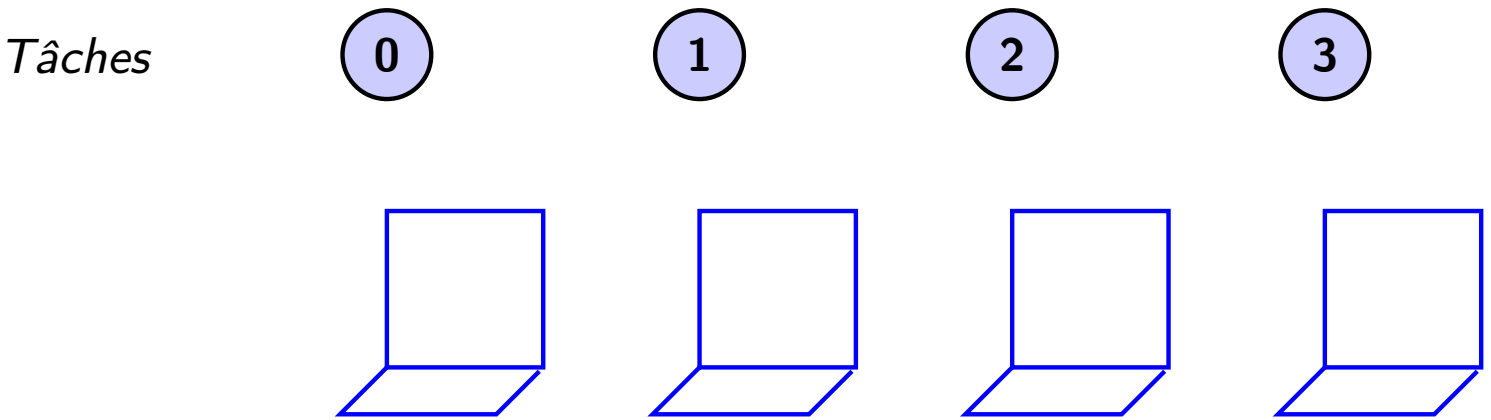
Qu'est-ce qu'un **sac de tâches** ?

- 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** ?

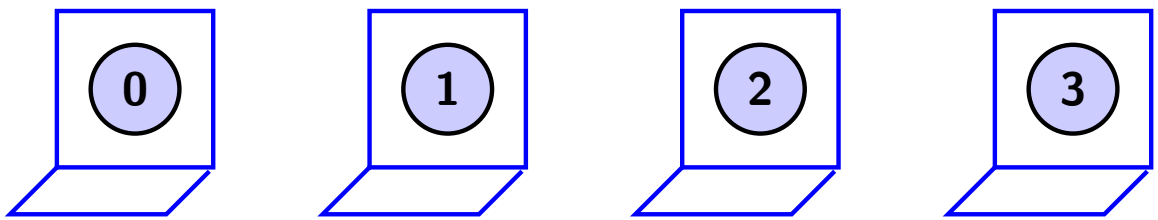
- Un ensemble de calculs à faire
- **Indépendants** les uns des autres

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

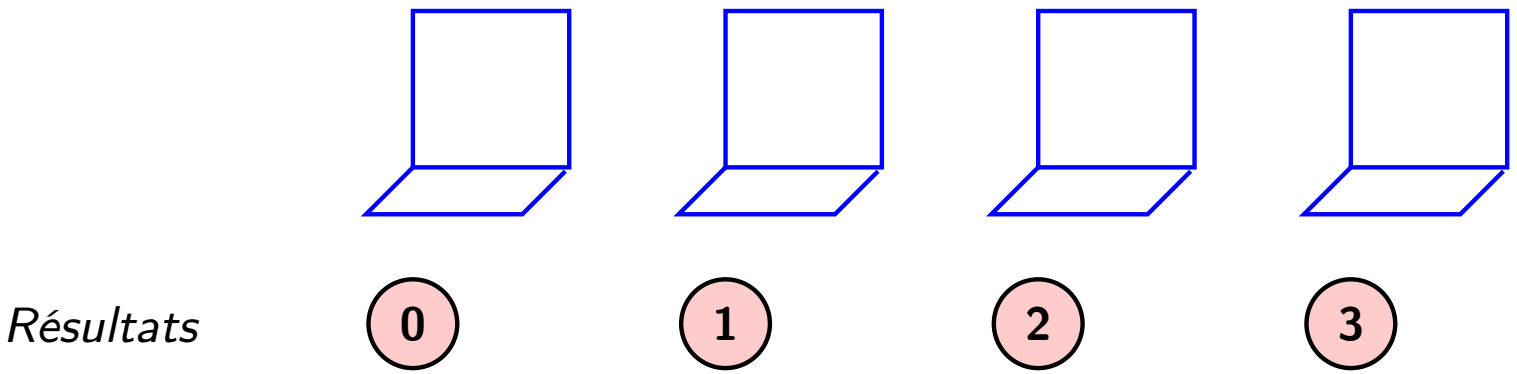
Qu'est-ce qu'un **sac de tâches** ?

- 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 !

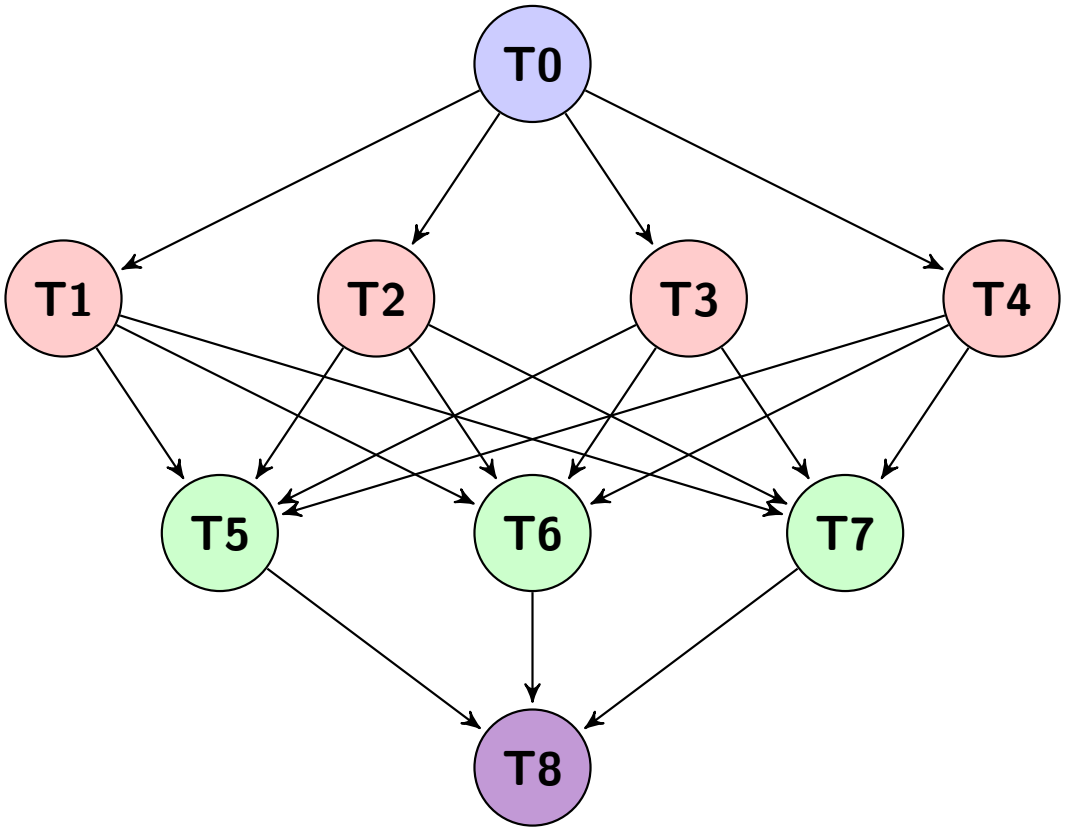
Pas de communications entre les processus exécutant les tâches.



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

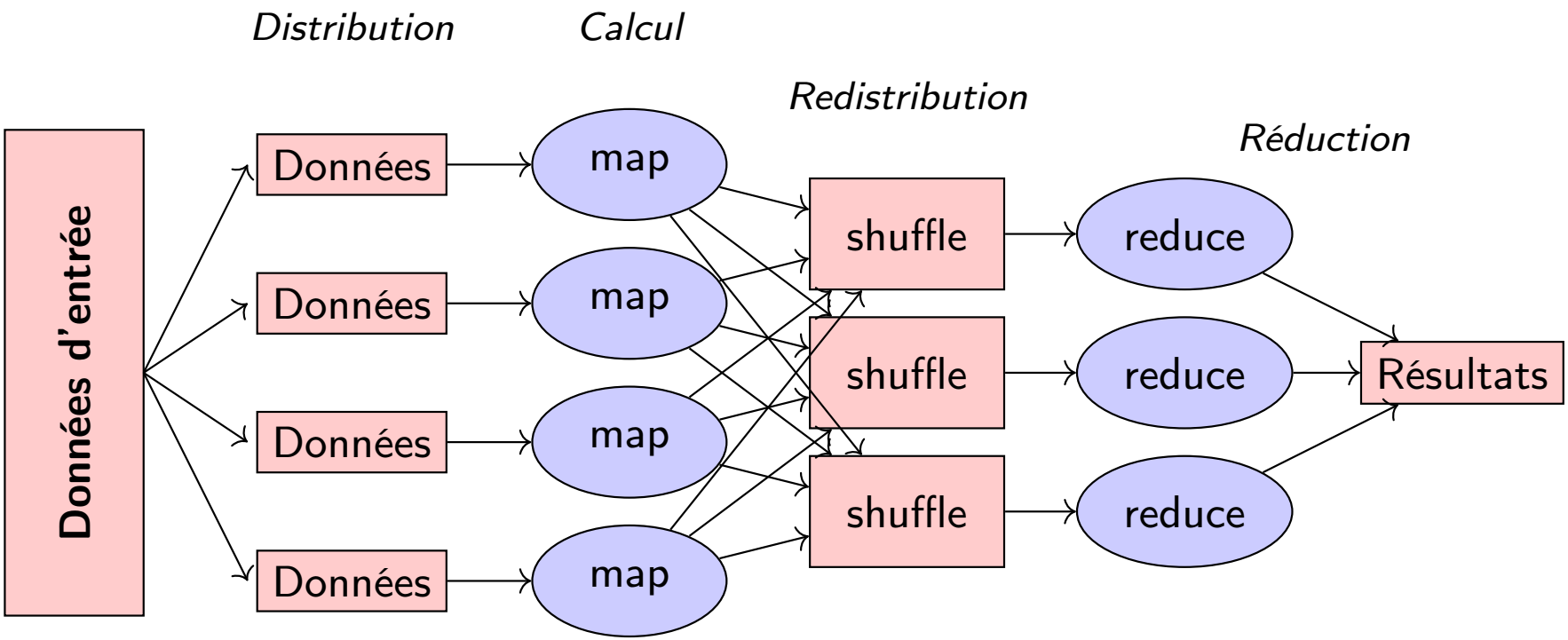
- Besoin d'un coordinateur qui distribue les tâches
- ... et qui récupère les résultats à la fin.

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*



Plan du cours

Plan du cours

Communications inter-processus

Passage de messages

Envoi de messages *explicite* entre deux processus

- Un processus A envoie à un processus B
- A exécute la primitive : `send( dest, &msgptr )`
- B exécute la primitive : `recv( dest, &msgptr )`

Les deux processus émetteur-récepteur doivent exécuter une primitive, de réception pour le récepteur et d'envoi pour l'émetteur

Nommage des processus

On a besoin d'une façon unique de désigner les processus

- Association adresse / port $\rightarrow$ portabilité ?
- On utilise un *rang de processus*, unique, entre 0 et N-1

Gestion des données

Tampons des messages

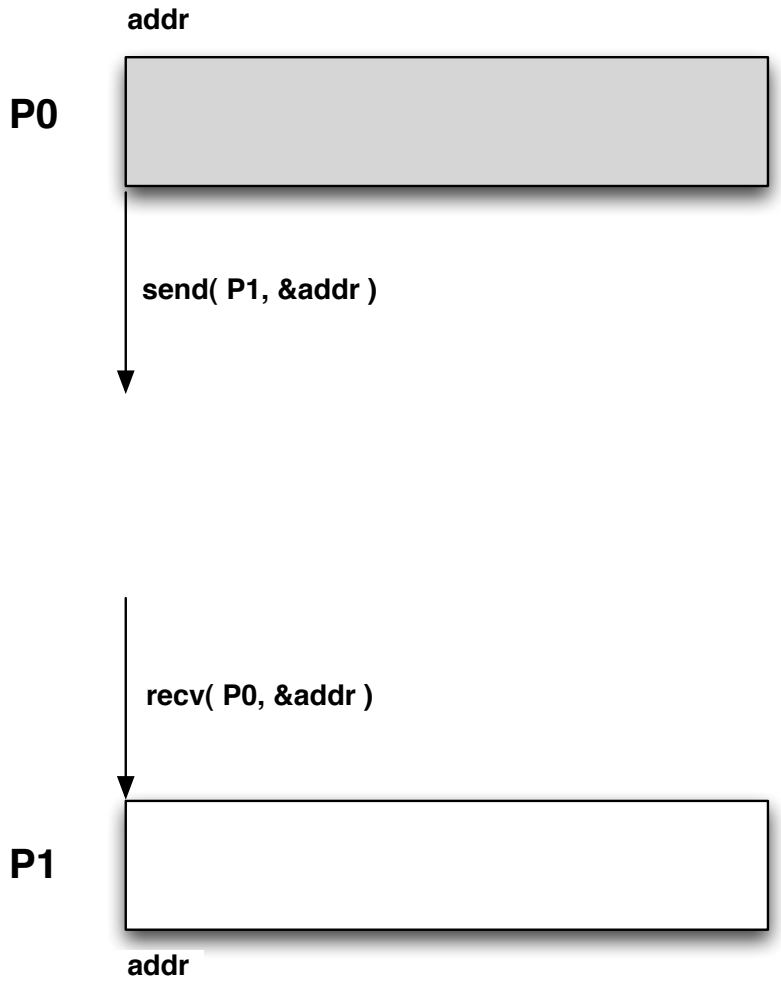
Chaque processus (émetteur et récepteur) a un tampon (buffer) pour le message

- La mémoire doit être allouée côté émetteur *et côté récepteur*
- On n'envoie pas plus d'éléments que la taille disponible en émission

Linéarisation des données

Les données doivent être sérialisées (marshalling) dans le tampon

- On envoie un tampon, un tableau d'éléments, une suite d'octets...



Modèle de communications

Asynchrones

- Délais de communications finis, non-bornés

Modes de communications

- Petits messages : *eager*
  - L'émetteur envoie le message sur le réseau et retourne dès qu'il a fini
  - Si le destinataire n'est pas dans une réception, le message est bufferisé
  - Quand le destinataire entre dans une réception, il commence par regarder dans ses buffers si il n'a pas déjà reçu
- Gros messages : *rendez-vous*
  - L'émetteur et le destinataire doivent être dans une communication
  - Mécanisme de rendez-vous :
    - Envoi d'un petit message
    - Le destinataire acquitte
    - Envoi du reste du message
  - L'émetteur ne retourne que si il a tout envoyé, donc que le destinataire est là : pas de mise en buffer

Plan du cours

La norme MPI

Message Passing Interface

Norme *de facto* pour la programmation parallèle par passage de messages

- Née d'un effort de standardisation
  - Chaque fabriquant avait son propre langage
  - Portabilité des applications !
- Effort commun entre industriels et laboratoires de recherche
- But : être à la fois portable et offrir de bonnes performances

Implémentations

Portabilité des applications écrites en MPI

- Applis MPI exécutables avec n'importe quelle implémentation de MPI
  - Propriétaire ou non, fournie avec la machine ou non

Fonctions MPI

- Interface définie en C, C++, Fortran 77 et 90
- Listées et documentées dans la norme
- Commencent par `MPI_` et une lettre majuscule
  - Le reste est en lettres minuscules

Historique de MPI

Évolution

- Appel à contributions : SC 1992
- 1994 : MPI 1.0
  - Communications point-à-point de base
  - Communications collectives
- 1995 : MPI 1.1 (clarifications de MPI 1.0)
- 1997 : MPI 1.2 (clarifications et corrections)
- 1998 : MPI 2.0
  - Dynamicité
  - Accès distant à la mémoire des processus (RDMA)
- 2008 : MPI 2.1 (clarifications)
- 2009 : MPI 2.2 (corrections, peu d'additions)
- En cours : MPI 3.0
  - Tolérance aux pannes
  - Collectives non bloquantes
  - et d'autres choses

Désignation des processus

Communicateur

Les processus communiquant ensemble sont dans un *communicateur*

- Ils sont tous dans MPI_COMM_WORLD
- Chacun est tout seul dans son MPI_COMM_SELF
- MPI_COMM_NULL ne contient personne

Possibilité de créer d'autres communicateurs au cours de l'exécution

Rang

Les processus sont désignés par un *rang*

- Unique dans un communicateur donné
  - Rang dans MPI_COMM_WORLD = rang absolu dans l'application
- Utilisé pour les envois / réception de messages

Déploiement de l'application

Lancement

`mpiexec` lance les processus sur les machines distantes

- Lancement = exécution d'un programme sur la machine distante
  - Le binaire doit être accessible de la machine distante
- Possibilité d'exécuter un binaire différent suivant les rangs
  - "vrai" MPMD
- Transmission des paramètres de la ligne de commande

Redirections

Les entrées-sorties sont redirigées

- `stderr`, `stdout`, `stdin` sont redirigés vers le lanceur
- MPI-IO pour les I/O

Finalisation

`mpiexec` retourne quand tous les processus ont terminé normalement ou un seul a terminé anormalement (plantage, défaillance...)

Hello World en MPI

Début / fin du programme

Initialisation de la bibliothèque MPI

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

Finalisation du programme

- `MPI_Finalize( );`

Si un processus quitte avant `MPI_Finalize( );`, ce sera considéré comme une erreur.

Ces deux fonctions sont OBLIGATOIRES !!!

Qui suis-je ?

Combien de processus dans l'application ?

- `MPI_Comm_size( MPI_COMM_WORLD, &size );`

Quel est mon rang ?

- `MPI_Comm_rank( MPI_COMM_WORLD, &rank );`

Hello World en MPI

Code complet

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

int main( int argc, char** argv ) {
  int size, rank;

  MPI_Init( &argc, &argv );

  MPI_Comm_size( MPI_COMM_WORLD, &size );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  fprintf( stdout, "Hello, I am rank %d in %d\n",
                   rank, size );

  MPI_Finalize();

  return EXIT_SUCCESS;
}
```

Hello World en MPI

Compilation

Compilateur C : `mpicc`

- Wrapper autour du compilateur C installé
- Fournit les chemins vers le `mpi.h` et la lib MPI
- Équivalent à
  `gcc -L/path/to/mpi/lib -lmpi -I/path/to/mpi/include`

```
mpicc -o helloworld helloworld.c
```

Exécution

Lancement avec `mpiexec`

- On fournit une liste de machines (`machinefile`)
- Le nombre de processus à lancer

```
mpiexec -machinefile ./machinefile -n 4 ./helloworld
Hello, I am rank 1 in 4
Hello, I am rank 2 in 4
Hello, I am rank 0 in 4
Hello, I am rank 3 in 4
```

Petite note sur Python

Les bindings Python sont **non-officiels**

- Ne font pas partie de la norme
- Par exemple : mpi4py

```
from mpi4py import MPI
```

Le script Python a besoin d'un interpréteur :

```
$ mpiexec -n 8 python helloWorld.py
```

Particularité de Python

Attention : il ne faut pas faire de `MPI_Init` ni de `MPI_Finalize`

Exemple Python

Communicateurs : `MPI.COMM_WORLD`, `MPI.COMM_SELF`, `MPI.COMM_NULL`

```
  #!/bin/python

from mpi4py import MPI

def main():
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()
    print "hello from " + str( rank ) + " in " + str( size )


if __name__ == "__main__":
    main()
```

Plan du cours

Communications point-à-point

Communications bloquantes

Envoi : `MPI_Send`

- `int MPI_Send( void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm )`

Réception : `MPI_Recv`

- `int MPI_Recv( void *buf, int count, MPI_Datatype datatype, int orig, int tag, MPI_Comm comm, MPI_Status *status )`

Communications point-à-point

Données

- buf : tampon d'envoi / réception
- count : nombre d'éléments de type `datatype`
- datatype : type de données
  - Utilisation de datatypes MPI
  - Assure la portabilité (notamment 32/64 bits, environnements hétérogènes...)
  - Types standards et possibilité d'en définir de nouveaux

Identification des processus

- Utilisation du couple communicateur / rang
- En réception : possibilité d'utilisation d'une wildcard
  - `MPI_ANY_SOURCE`
  - Après réception, l'émetteur du message est dans le status

Identification de la communication

- Utilisation du tag
- En réception : possibilité d'utilisation d'une wildcard
  - `MPI_ANY_TAG`
  - Après réception, le tag du message est dans le status

Ping-pong entre deux processus

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

int main( int argc, char** argv ) {
  int rank;
  int token = 42;
  MPI_Status status;

  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  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_Finalize();

  return EXIT_SUCCESS;
}
```

Ping-pong entre deux processus

Remarques

- À un envoi correspond *toujours* une réception
  - Même communicateur, même tag
  - Rang de l'émetteur et rang du destinataire
- On utilise le rang pour déterminer ce que l'on fait
- On envoie des entiers → MPI_INT

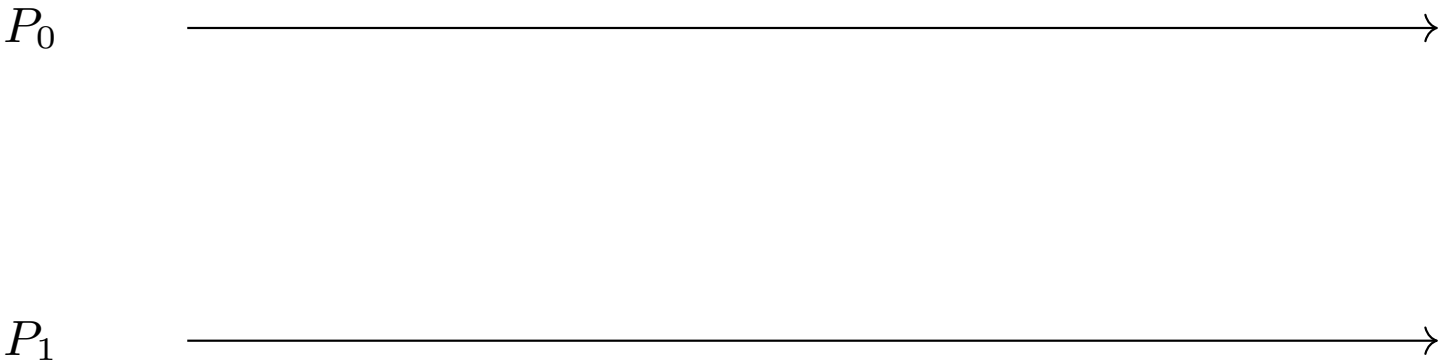
Sources d'erreurs fréquentes

- Le datatype et le nombre d'éléments doivent être identiques en émission et en réception
  - On s'attend à recevoir ce qui a été envoyé
- Attention à la correspondance MPI_Send et MPI_Recv
  - Deux MPI_Send ou deux MPI_Recv = deadlock !

Ping-pong illustré

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

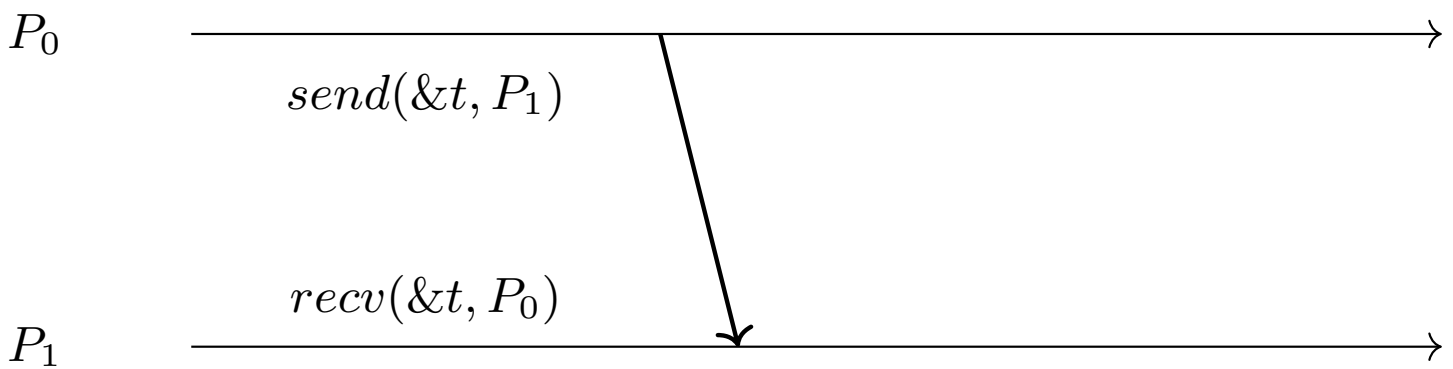
```
if( 0 == rank ) {
    MPI_Send( &token, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    MPI_Recv( &token, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &status );
} else if( 1 == rank ) {
    MPI_Recv( &token, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status );
    MPI_Send( &token, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
}
```



Ping-pong illustré

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

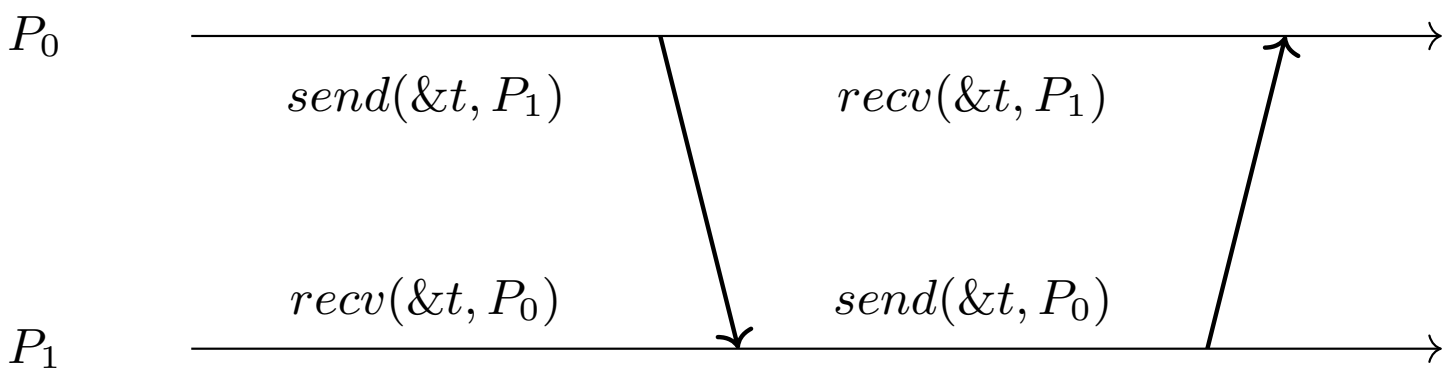
```
if( 0 == rank ) {
    MPI_Send( &token, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    MPI_Recv( &token, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &status );
} else if( 1 == rank ) {
    MPI_Recv( &token, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status );
    MPI_Send( &token, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
}
```



Ping-pong illustré

- 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 non-bloquantes

But

La communication a lieu pendant qu'on fait autre chose

- Superposition communication/calcul
- Plusieurs communications simultanées sans risque de deadlock

Quand on a besoin des données, on attend que la communication ait été effectuée complètement

Communications

Envoi : `MPI_Isend`

- `int MPI_Isend( void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request )`

Réception : `MPI_Irecv`

- `int MPI_Irecv( void *buf, int count, MPI_Datatype datatype, int orig, int tag, MPI_Comm comm, MPI_Request *request )`

Communications non-bloquantes

Attente de complétion

Pour une communication :

- `int MPI_Wait( MPI_Request *request, MPI_Status *status )`

Attendre plusieurs communications : `MPI_{Waitall, Waitany, Waitsome}`

Test de complétion

Pour une communication :

- `int MPI_Test( MPI_Request *request, int *flag, MPI_Status *status )`

Tester plusieurs communications : `MPI_{Testall, Testany, Testsome}`

Annuler une communication en cours

Communication non-bloquante identifiée par sa request

- `int MPI_Cancel(MPI_Request *request)`

Différences

- MPI_Wait est bloquant, MPI_Test ne l'est pas
- MPI_Test peut être appelé simplement pour entrer dans la bibliothèque MPI (lui redonner la main pour faire avancer des opérations)

Plan du cours

Plan du cours

Sémantique des communications collectives

Tous les processus participent à la communication collective

- En MPI : lié à la notion de communicateur
- On effectue une communication collective *sur un communicateur*
  - MPI_COMM_WORLD ou autre
  - Utilité de bien définir ses communicateurs !

Fin de l'opération

Bloquant (pour le moment)

Sémantique des communications collectives

Tous les processus participent à la communication collective

Fin de l'opération

Un processus sort de la communication collective une fois qu'il a terminé *sa participation* à la collective

- Aucune garantie sur l'avancée globale de la communication collective
- Dans certaines communications collectives, un processus peut avoir terminé avant que d'autres processus n'aient commencé
- Le fait qu'un processus ait terminé **ne signifie pas** que la collective est terminée !
- Pas de synchronisation (sauf pour certaines communications collectives)

Bloquant (pour le moment)

Sémantique des communications collectives

Tous les processus participent à la communication collective

Fin de l'opération

Bloquant (pour le moment)

- Quelques projets de communications collectives non-bloquantes (NBC, MPI 3)
- On entre dans la communication collective et on n'en ressort que quand on a terminé sa participation à la communication

Exemple de communication collective : diffusion avec MPI

Diffusion avec MPI : `MPI_Bcast`

- Diffusion d'un processus vers les autres : définition d'un processus racine
- On envoie un tampon de N éléments d'un type donné, sur un communicateur

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

int main( int argc, char** argv ) {
  int size, rank, token;
  MPI_Init( &argc, &argv );

  MPI_Comm_size( MPI_COMM_WORLD, &size );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  if( 0 = rank ) {
    token = getpid();
  }
  MPI_Bcast( &token, 1, MPI_INT, 0, MPI_COMM_WORLD );

  MPI_Finalize();
  return EXIT_SUCCESS;
}
```

Exemple de communication collective : diffusion avec MPI

```
MPI_Bcast( &token, 1, MPI_INT, 0, MPI_COMM_WORLD );
```

Le processus 0 diffuse un entier (`token`) vers les processus du communicateur MPI_COMM_WORLD

- Avant la communication collective : `token` est initialisé uniquement sur 0
- Tous les processus sauf 0 reçoivent quelque chose dans leur variable `token`
- Après la communication collective : tous les processus ont la même valeur dans leur variable `token` locale

Tous les processus du communicateur concerné doivent appeler `MPI_Bcast`

- Sinon : deadlock

Plan du cours

Modèle pour les communications

Modèle pour les communications point-à-point

$$T_{comm} = \lambda + \frac{s}{\beta} \quad (1)$$

Avec λ = latence, β = bande passante, s = taille du message

Comment représenter le temps pris par une communication collective ?

- Temps pour que *tous les processus* participent à la communication collective et en sortent
- Temps pour que *chaque processus* termine sa participation locale à la communication collective

Modèle pour les communications

Modèle pour les communications point-à-point

$$T_{comm} = \lambda + \frac{s}{\beta} \quad (1)$$

Avec λ = latence, β = bande passante, s = taille du message

Comment représenter le temps pris par une communication collective ?

- Temps pour que *tous les processus* participent à la communication collective et en sortent
- Temps pour que *chaque processus* termine sa participation locale à la communication collective

Comment quantifier et mesurer ?

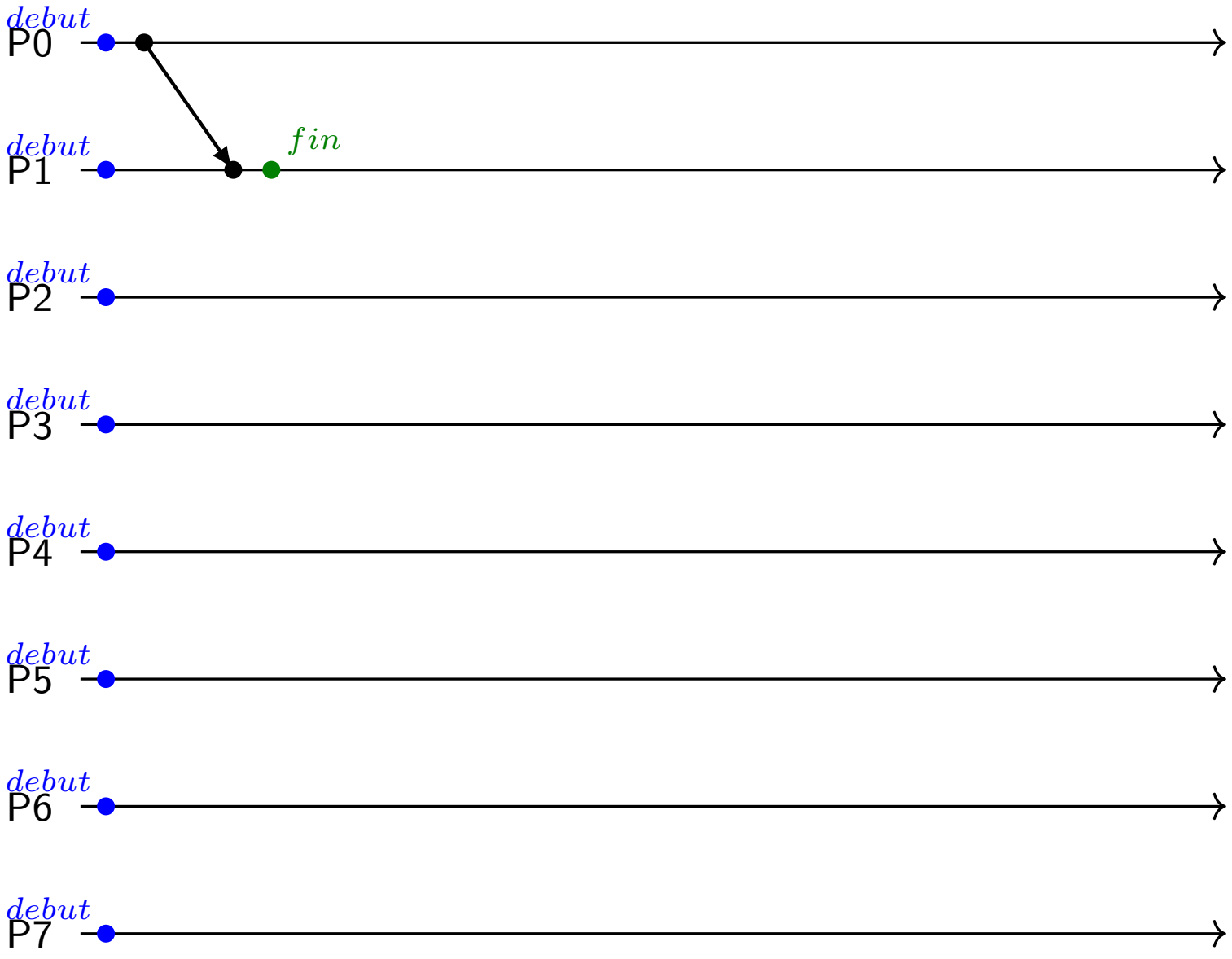
- Nombre de messages envoyés / reçus (latence)
- Utilisation de la bande passante

Paramètres :

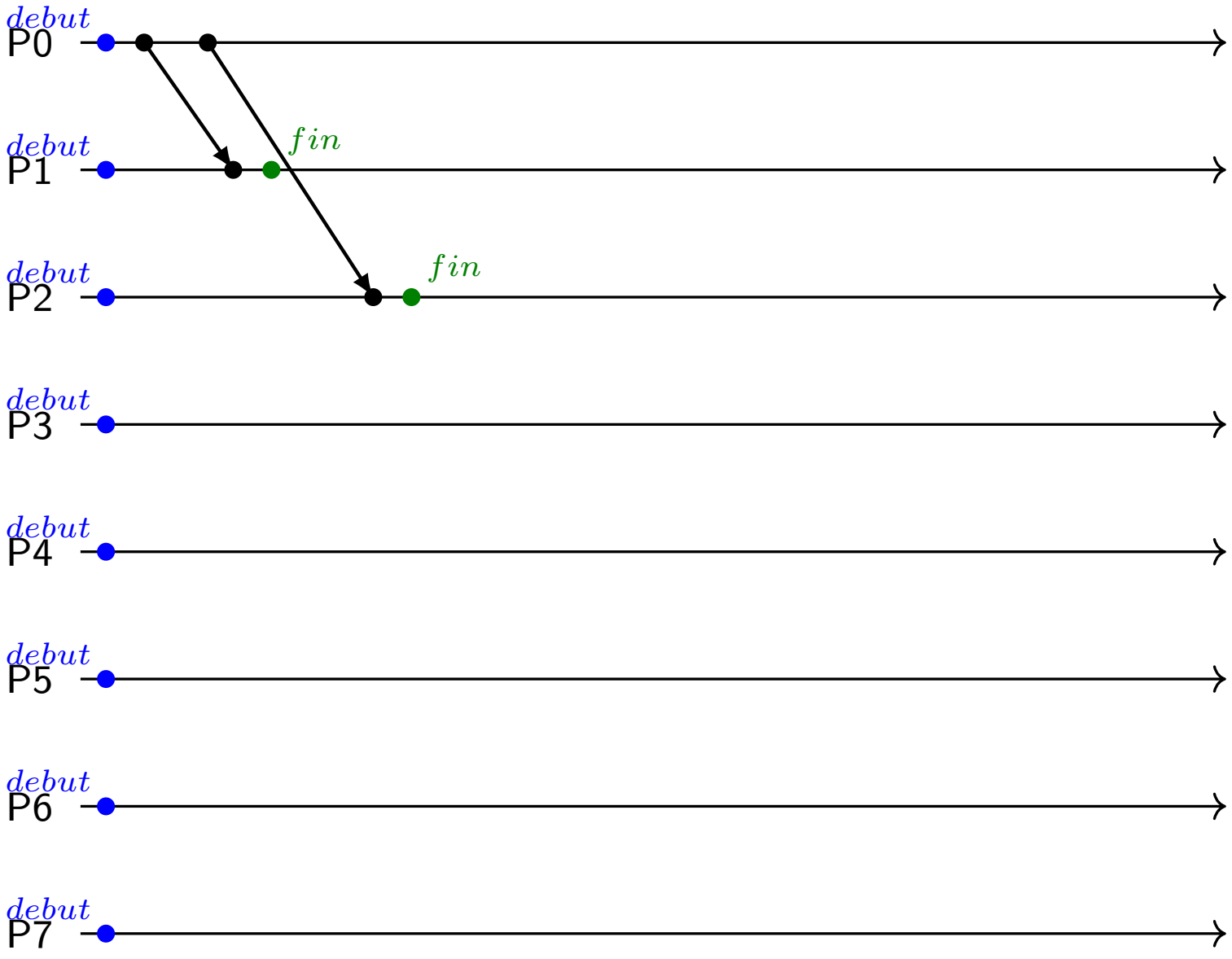
- Taille du message
- Nombre de processus impliqués !

Exemple : certains algorithmes seront plus performants sur des petits messages (bandwidth-bound), d'autres sur des gros messages (latency-bound), d'autres passent mieux à l'échelle...

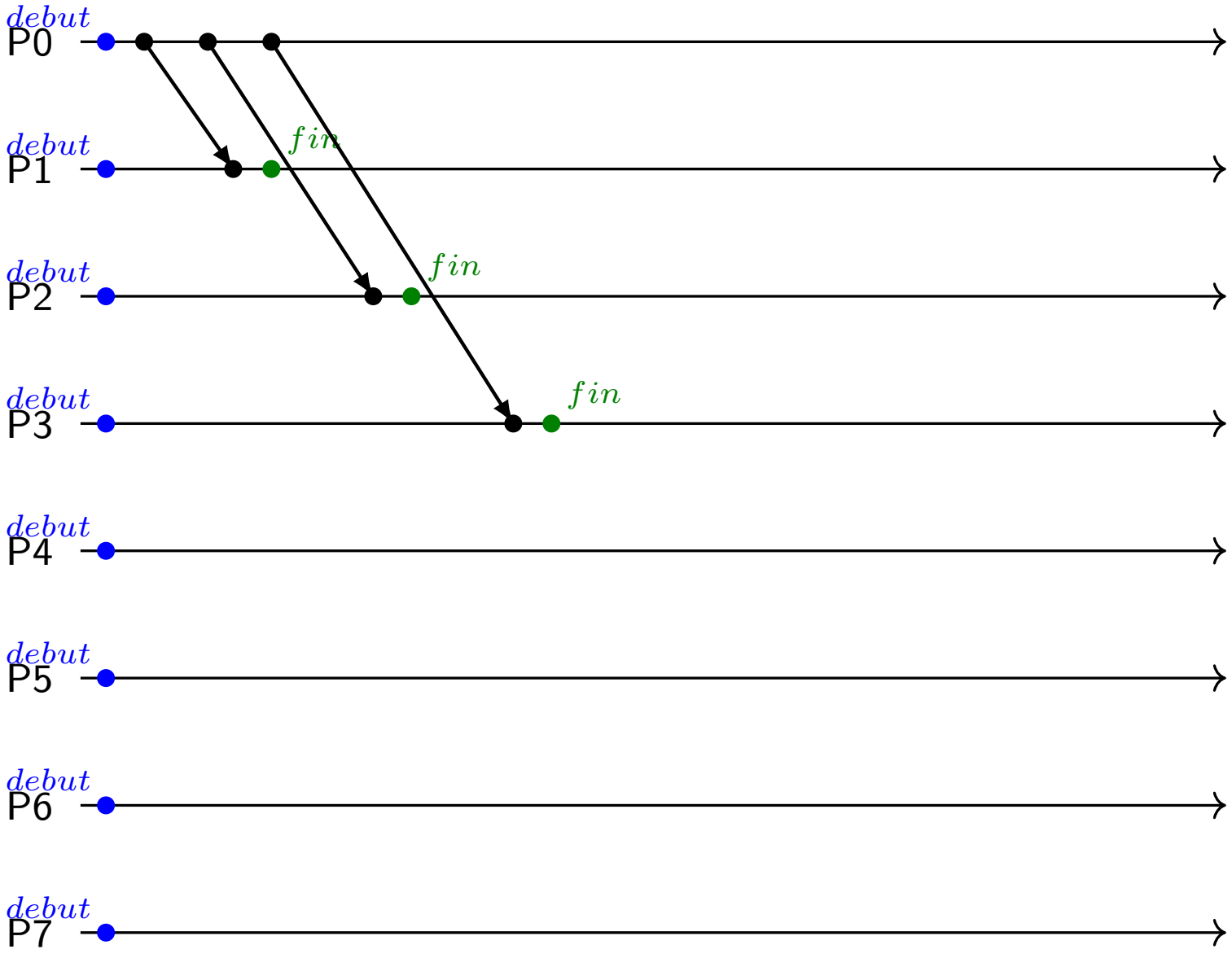
Modèle pour les communications : Bcast étoile



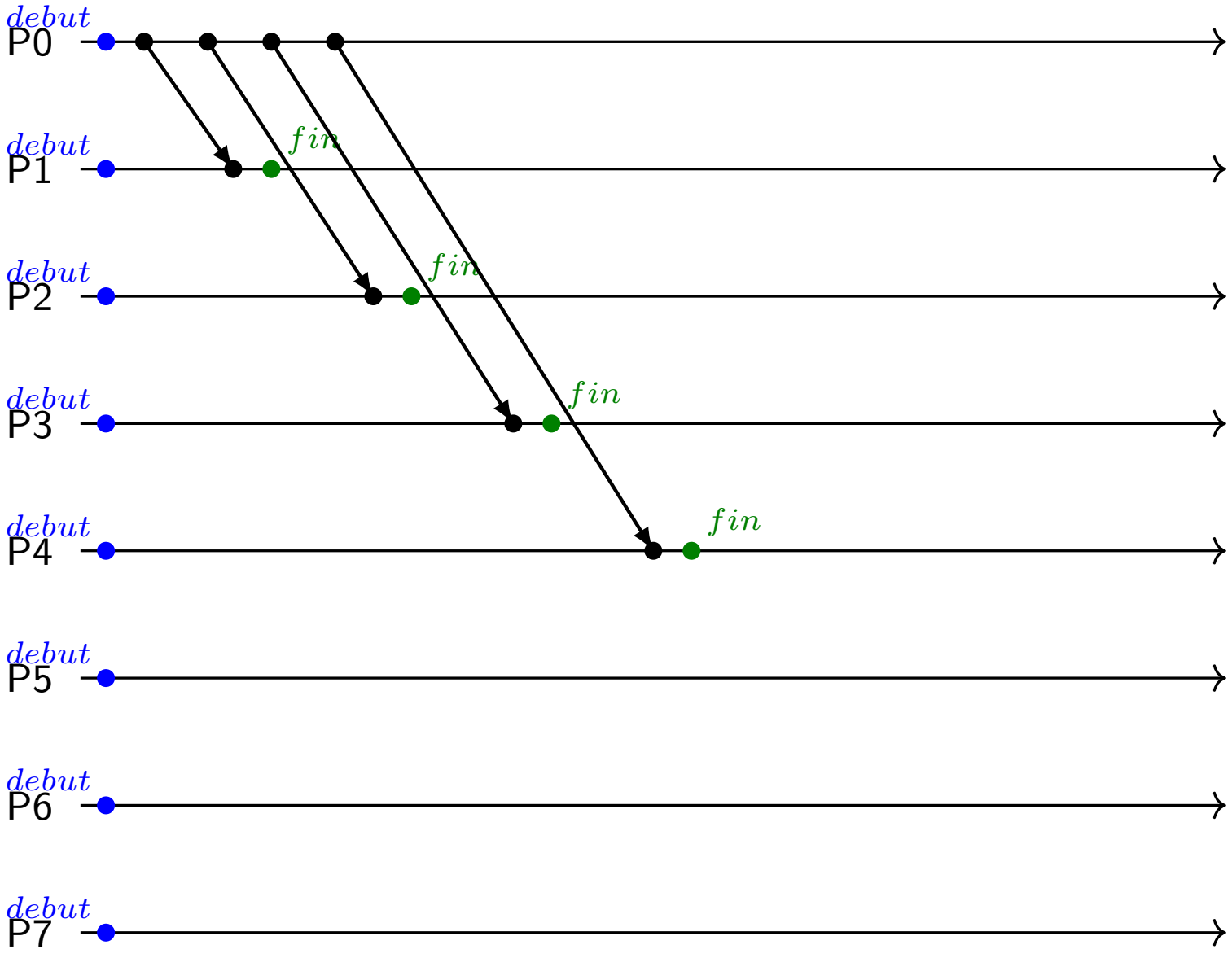
Modèle pour les communications : Bcast étoile



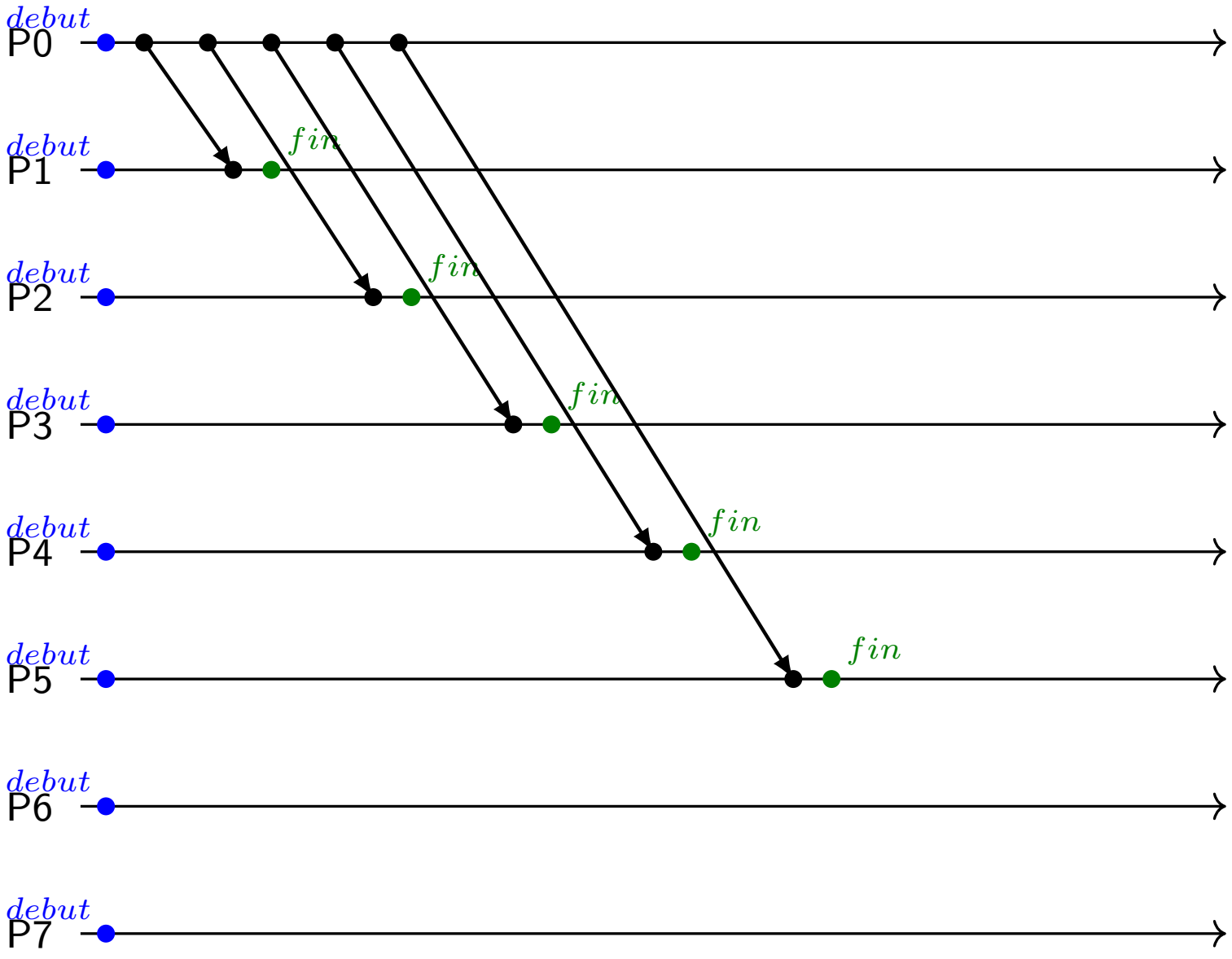
Modèle pour les communications : Bcast étoile



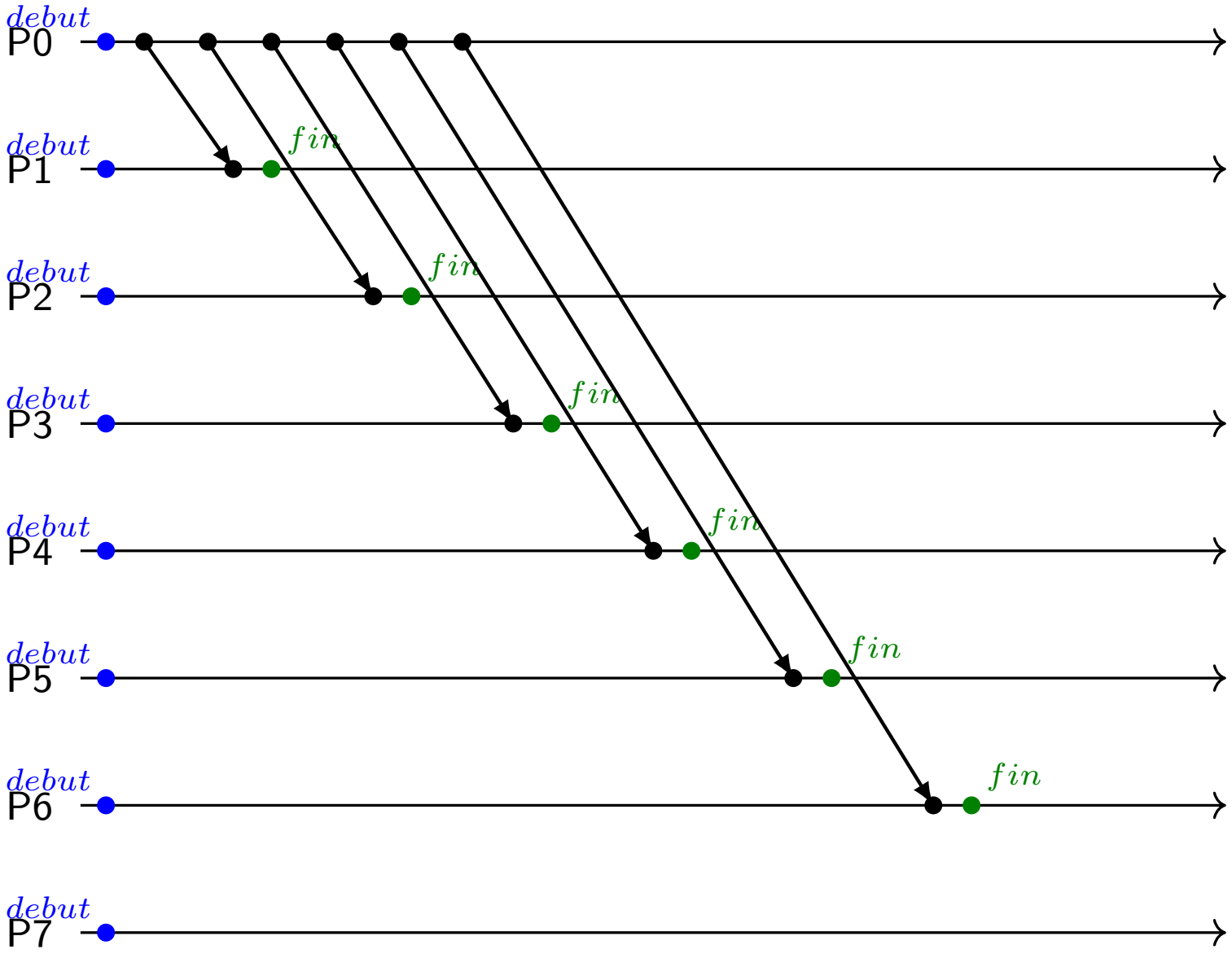
Modèle pour les communications : Bcast étoile



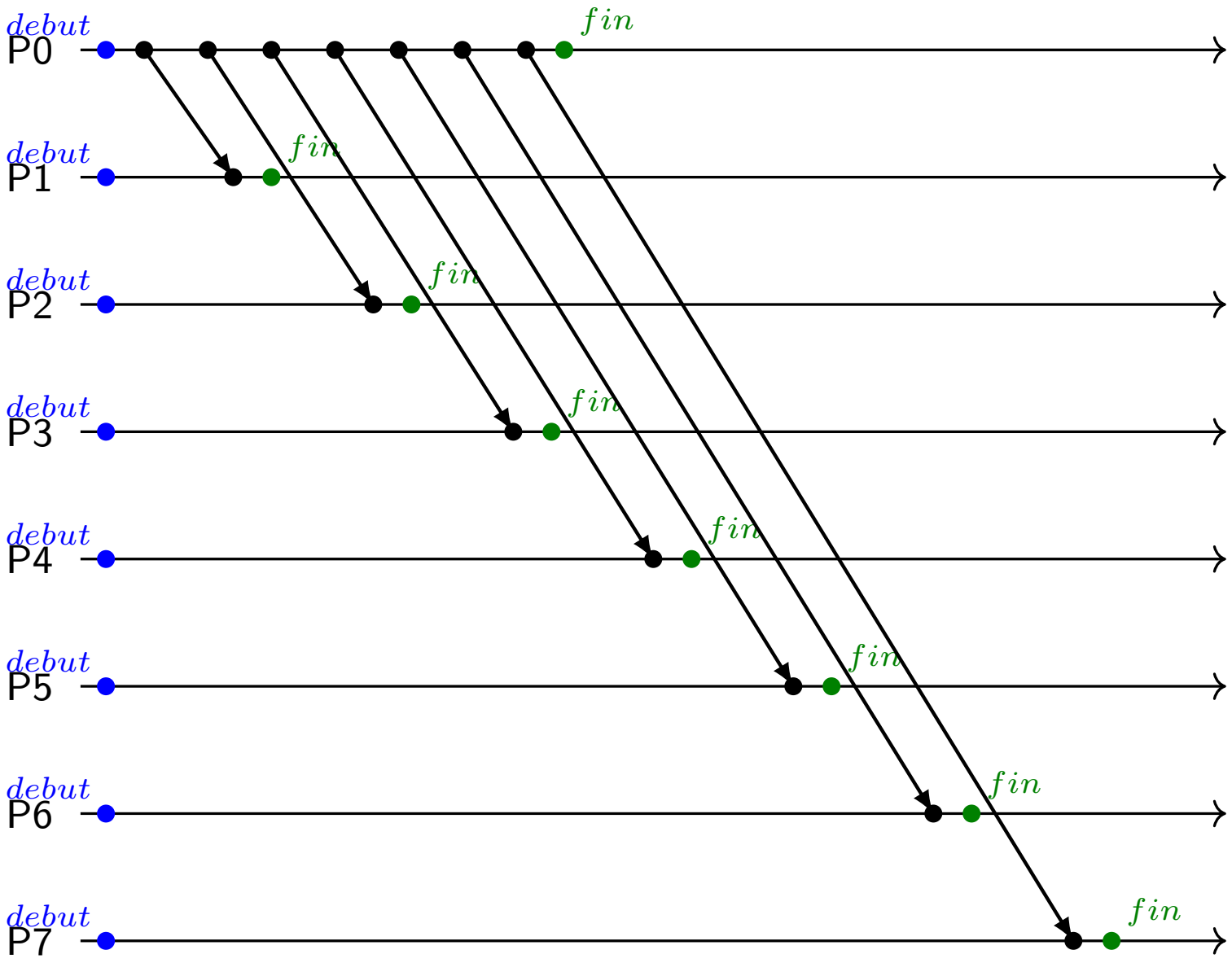
Modèle pour les communications : Bcast étoile



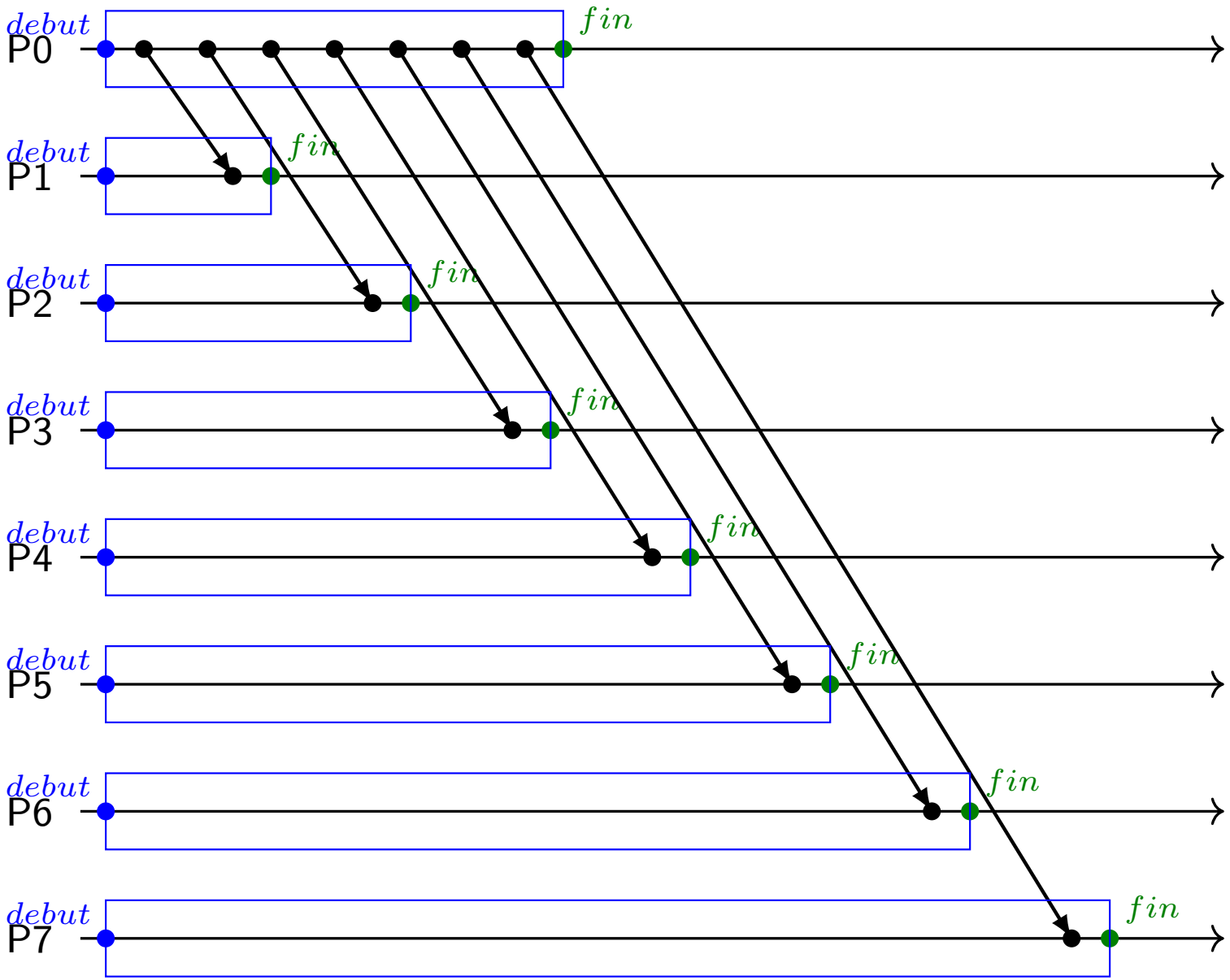
Modèle pour les communications : Bcast étoile



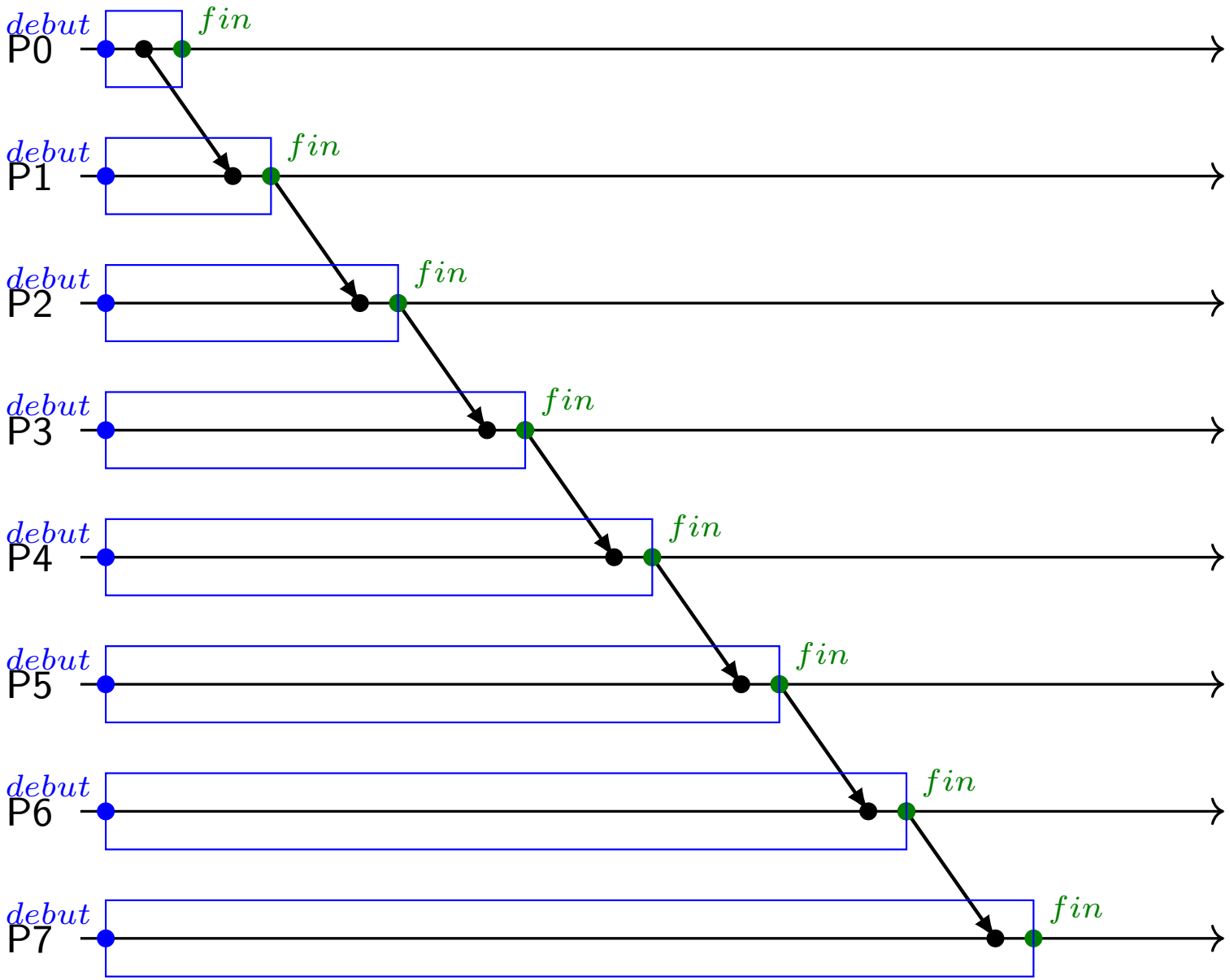
Modèle pour les communications : Bcast étoile



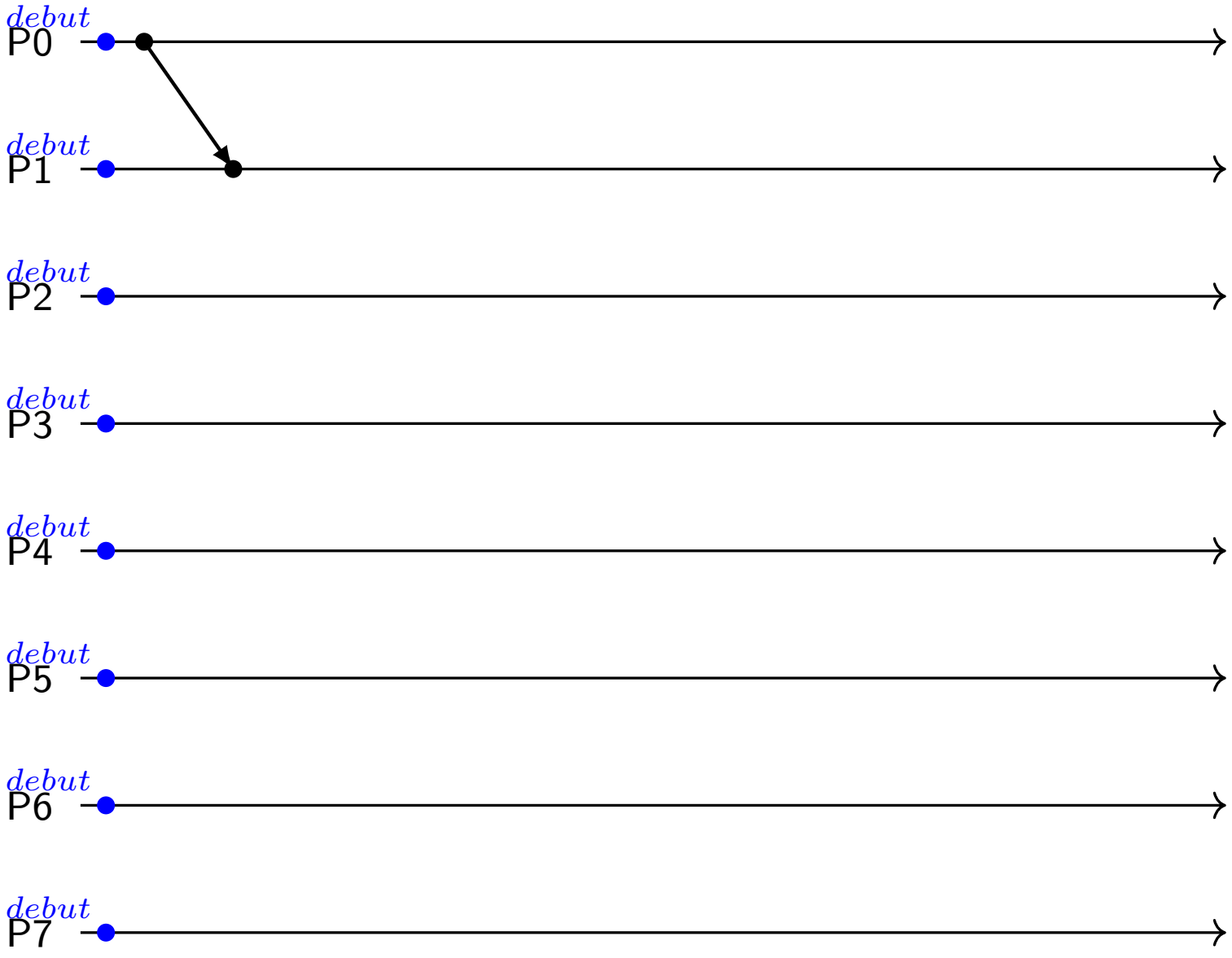
Modèle pour les communications : Bcast étoile



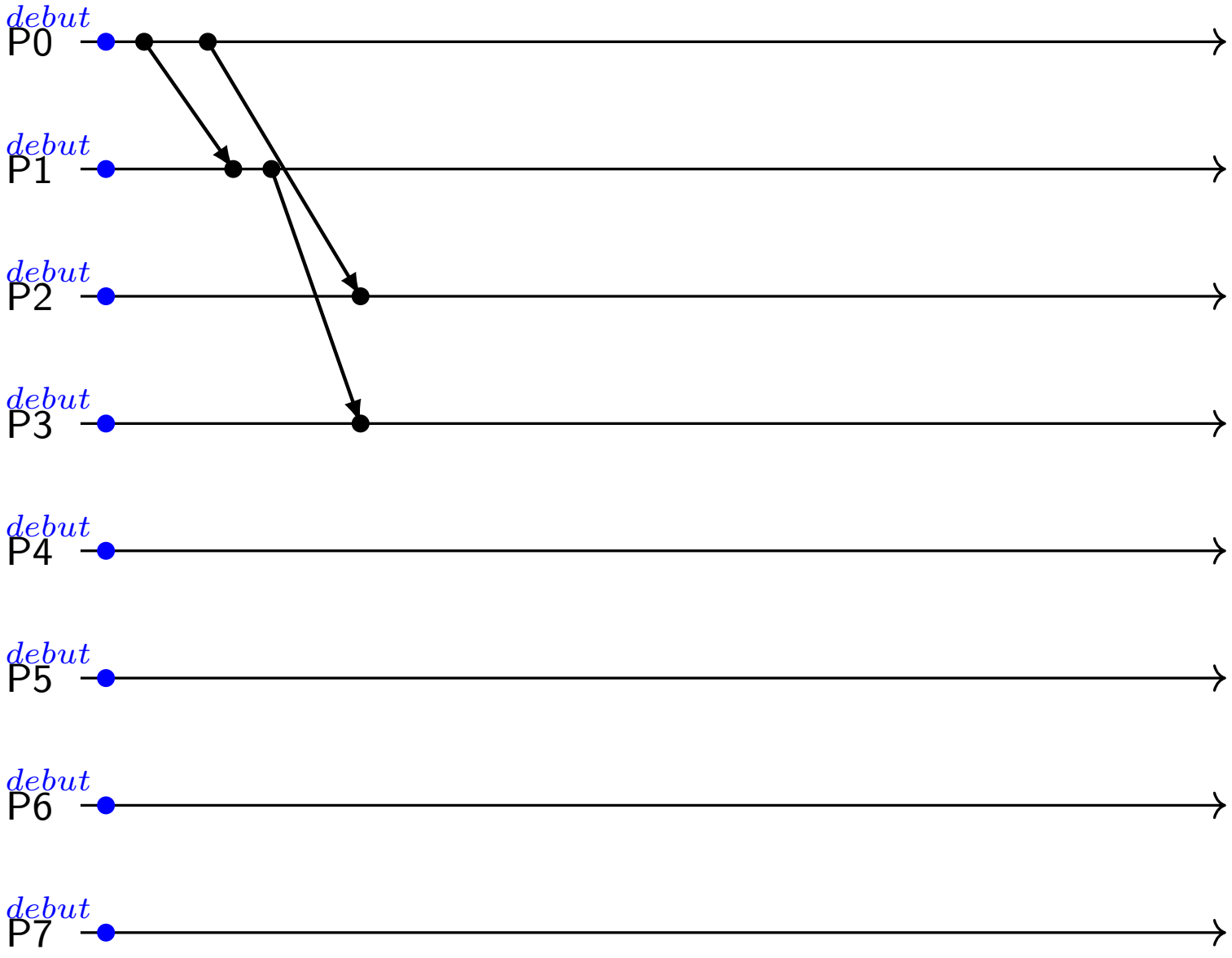
Modèle pour les communications : Bcast chaine



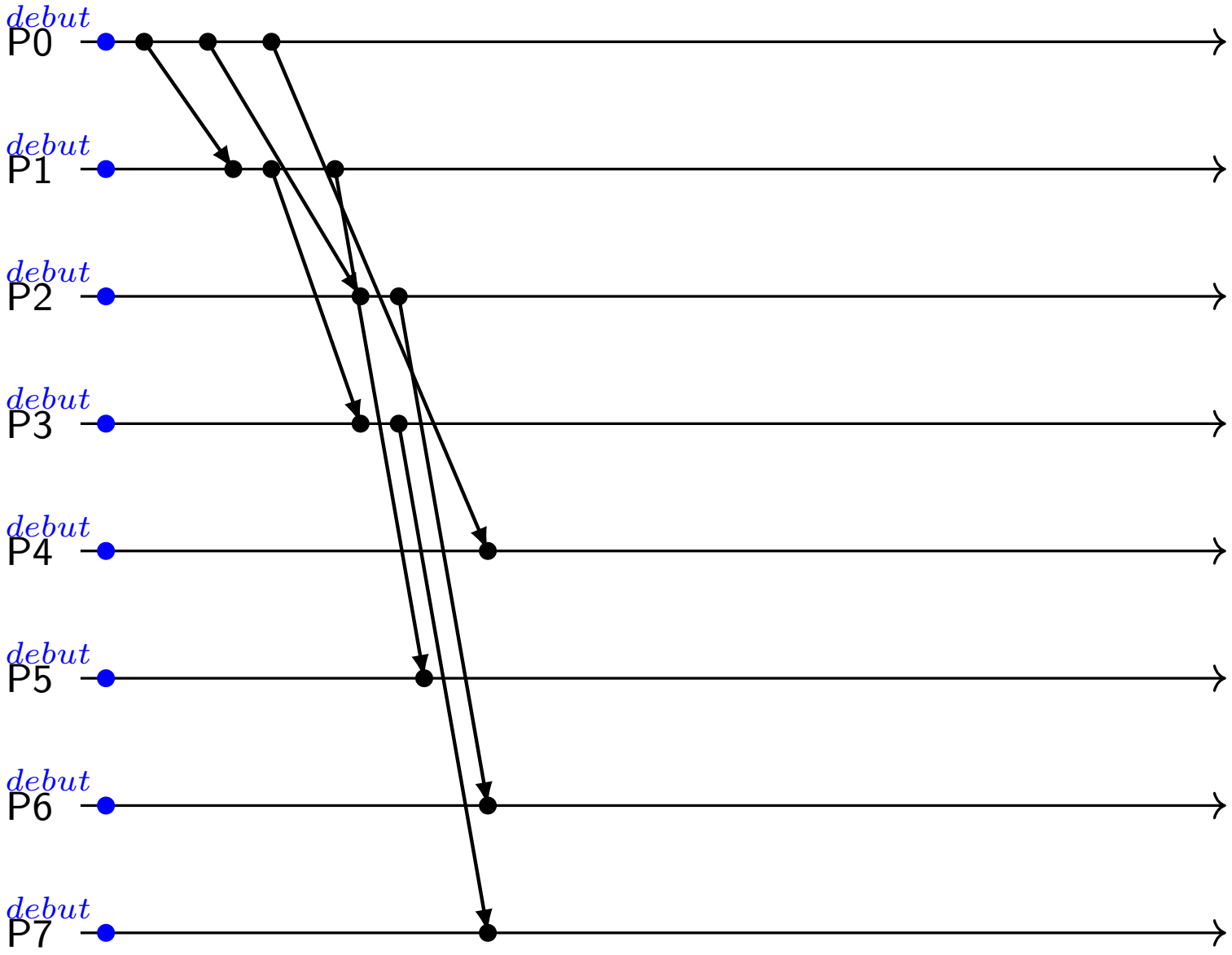
Modèle pour les communications : Bcast binomial



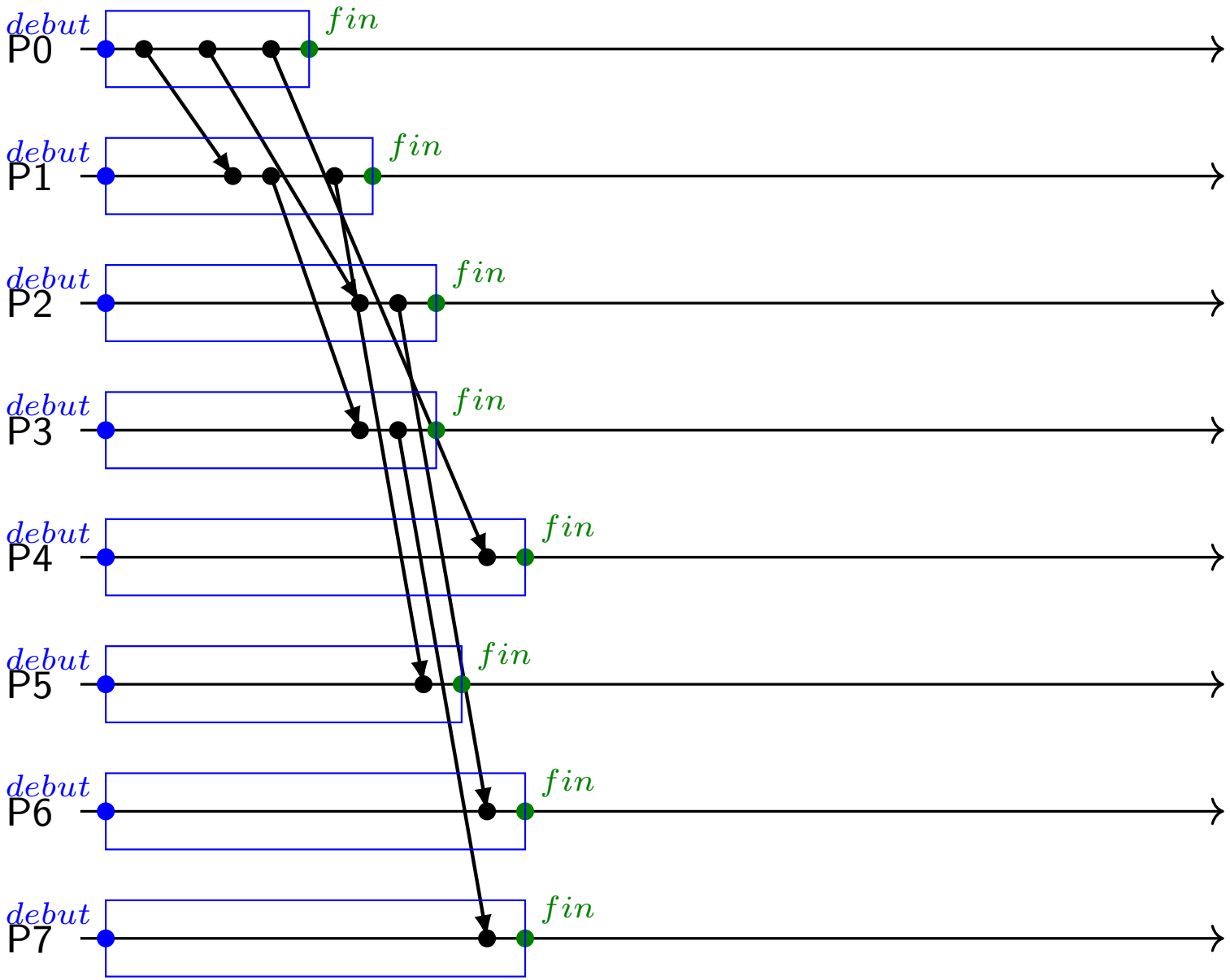
Modèle pour les communications : Bcast binomial



Modèle pour les communications : Bcast binomial



Modèle pour les communications : Bcast binomial



Communications collectives

Plan du cours

Communications collectives

Diffusion

Sémantique

Une diffusion envoie une donnée (le contenu d'un buffer)

- à partir d'un processus racine
- vers tous les processus du communicateur

```
int MPI\_Bcast( void *buffer, int count, MPI\_Datatype
  datatype, int root, MPI\_Comm comm );
```

Communications collectives

Algorithme de diffusion : l'étoile

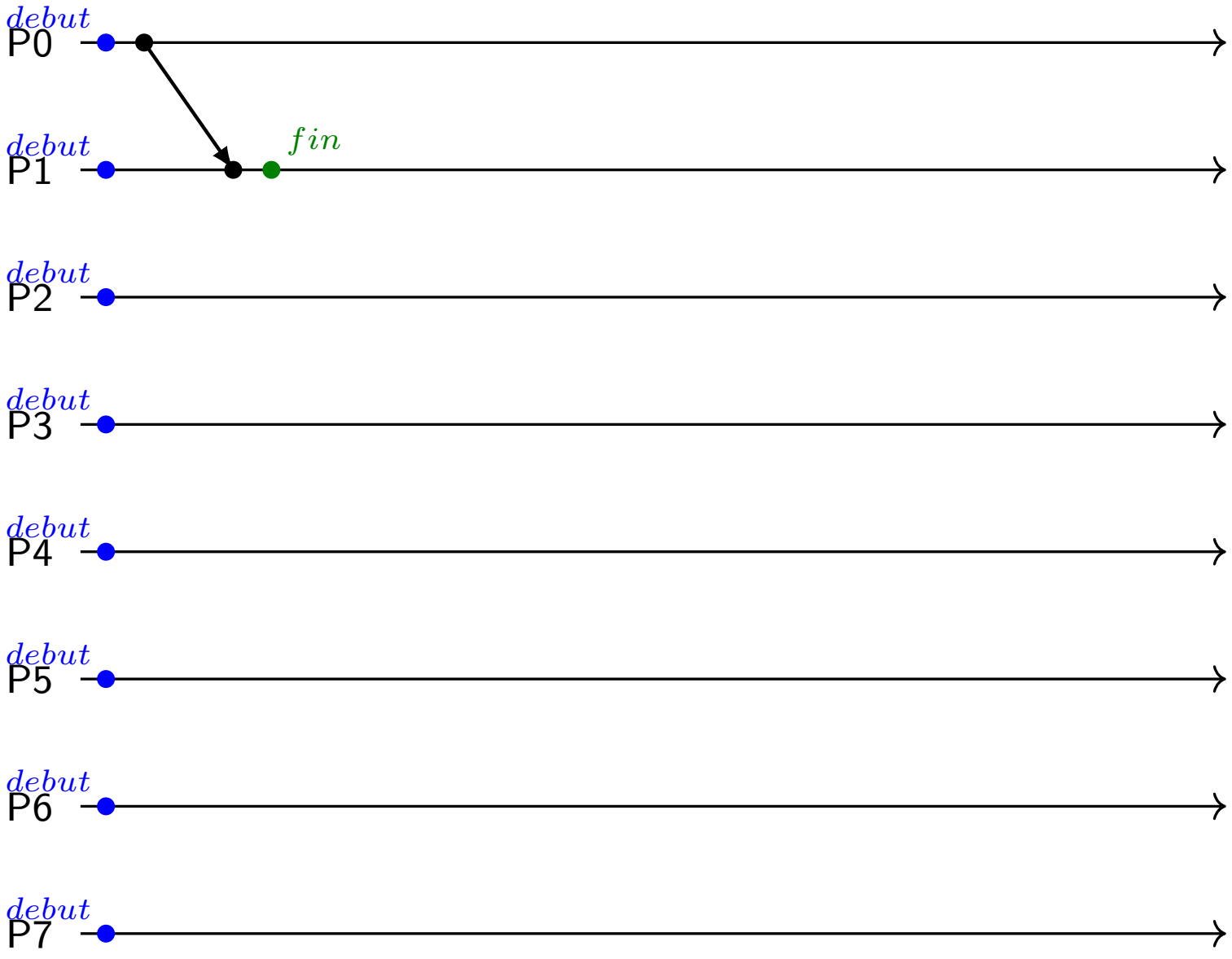
Le processus racine envoie à tous les autres processus du communicateur :

```
if( root == rank ) {
  for( i = 0 ; i < size ; i++ ) {
    if( i != root ) {
      send( message, i )
    }
  }
} else {
  recv( message, root )
}
```

- Nombre de messages ? $N - 1$ pour la racine, 1 pour les autres processus
- Mais le message de chaque processus n'arrive pas tout de suite !
  - Dépend du modèle N-port : N messages envoyés simultanément
- Complexités en $O(N)$: pas scalable ! ! !

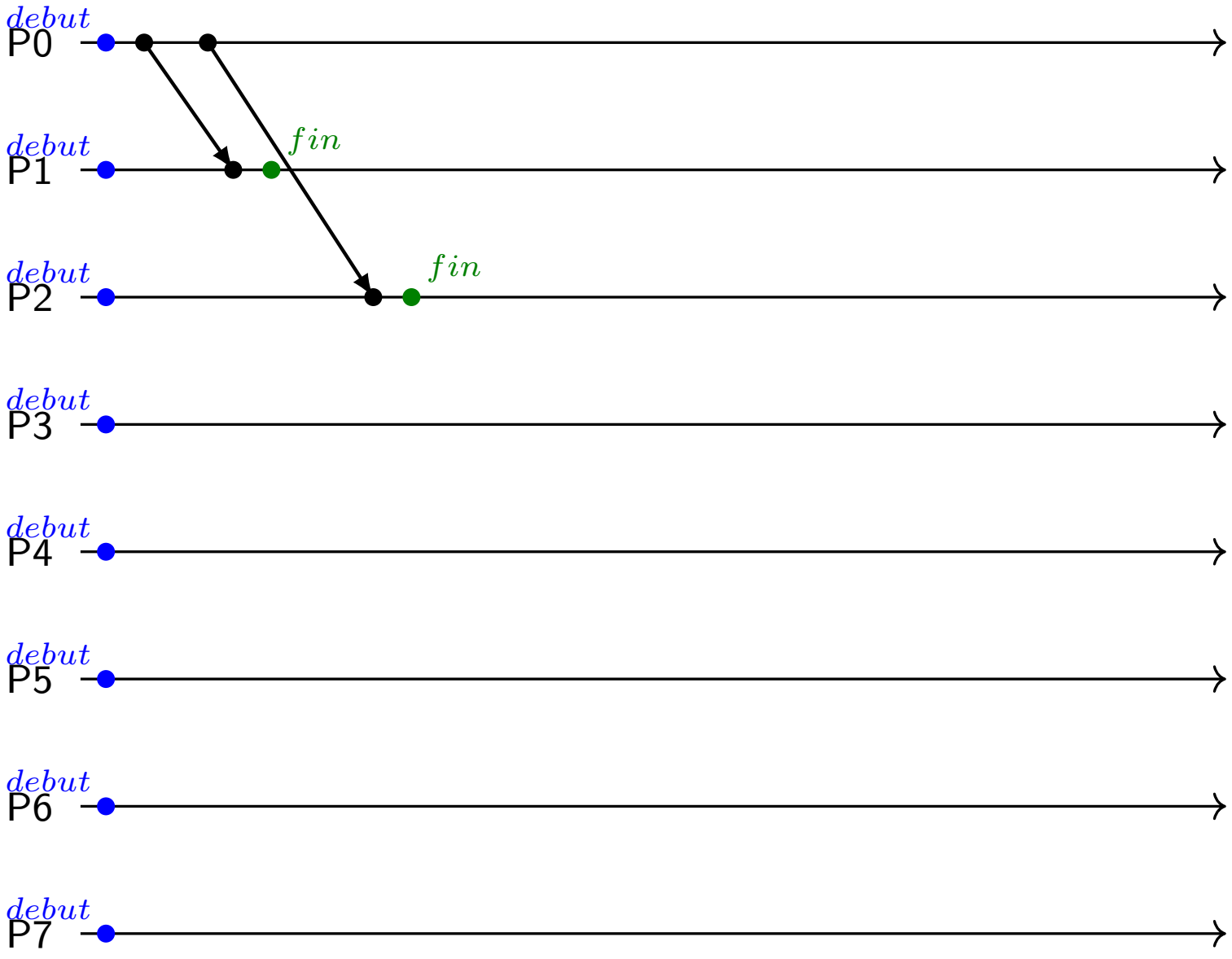
Communications collectives

Algorithme de diffusion : l'étoile



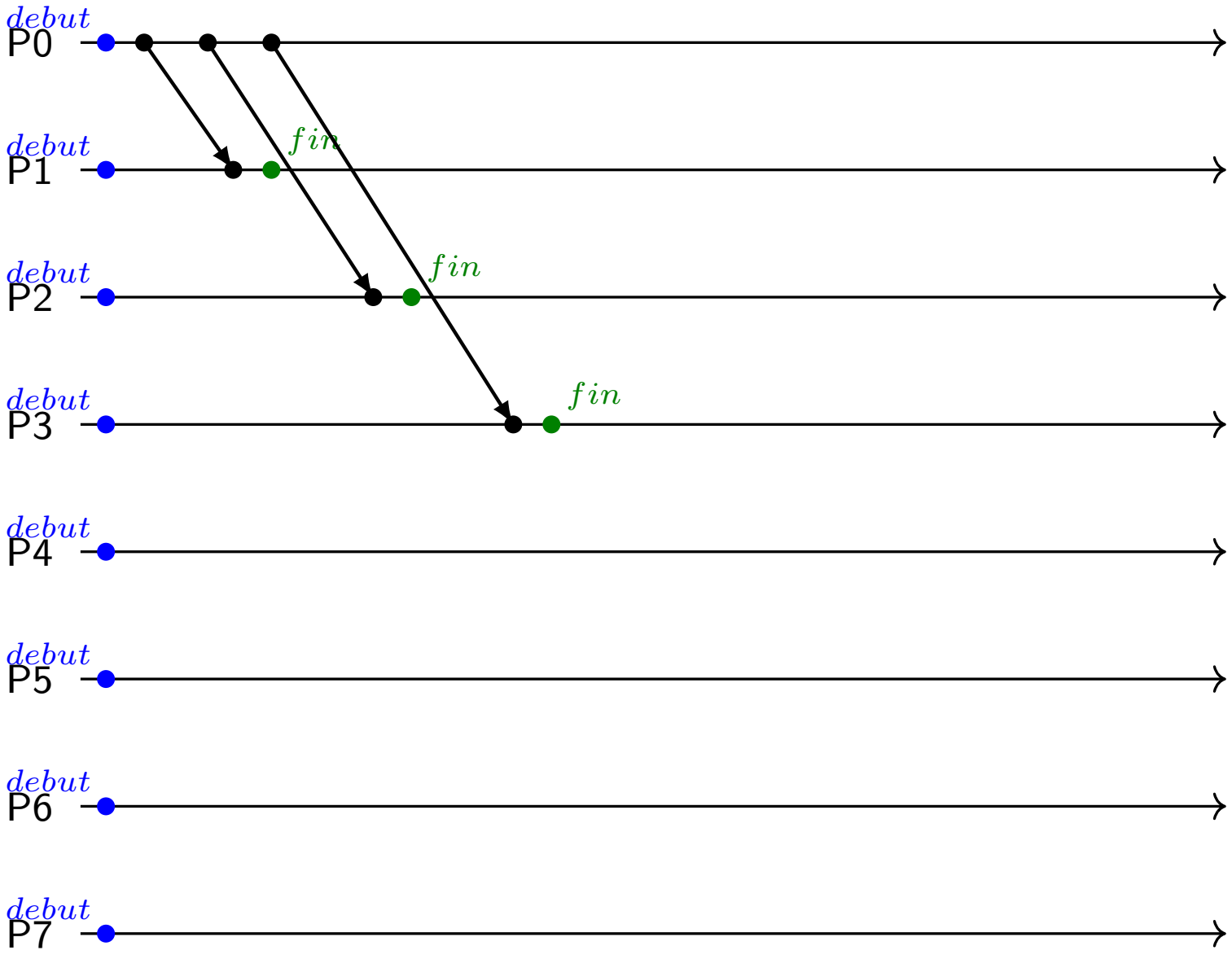
Communications collectives

Algorithme de diffusion : l'étoile

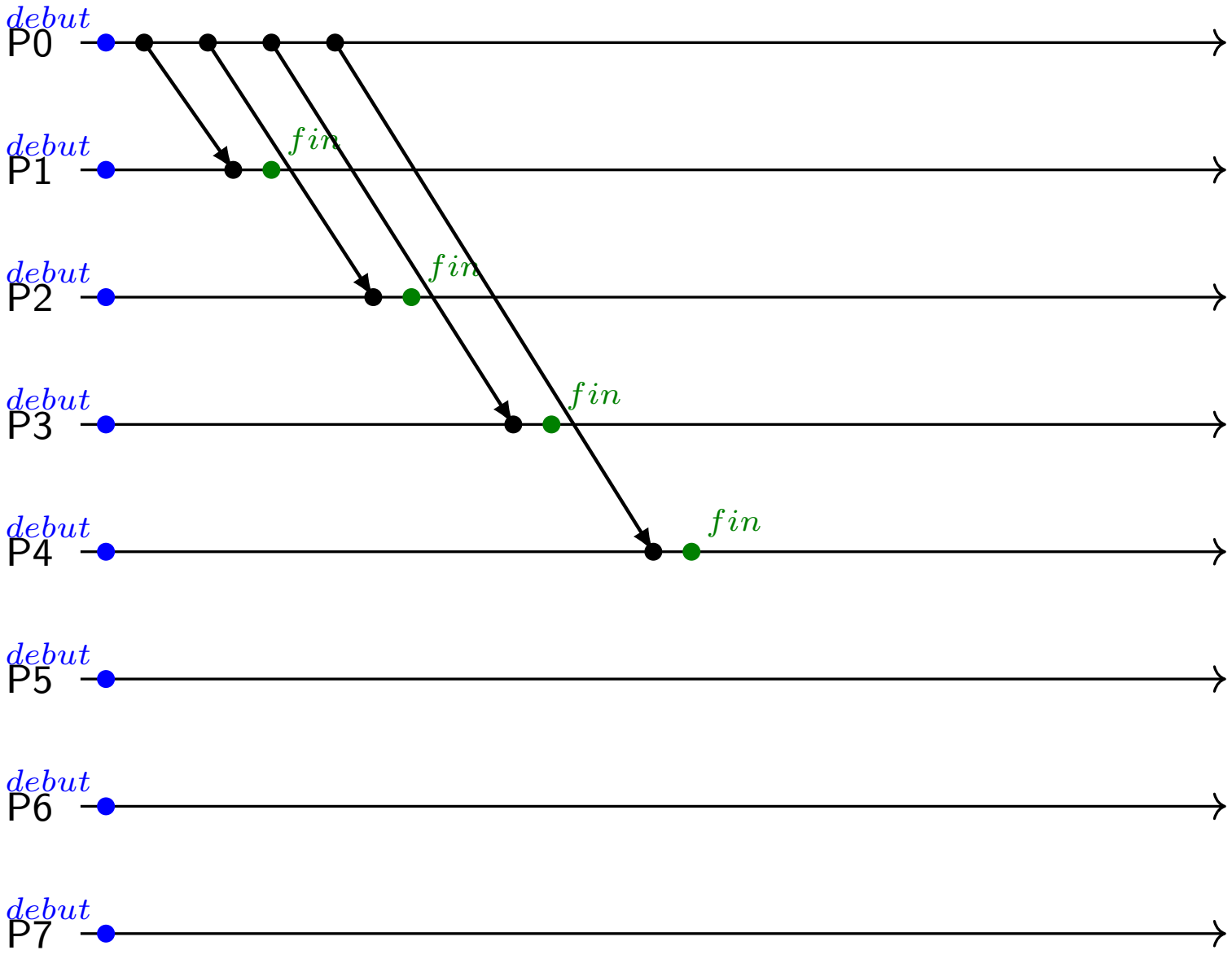


Communications collectives

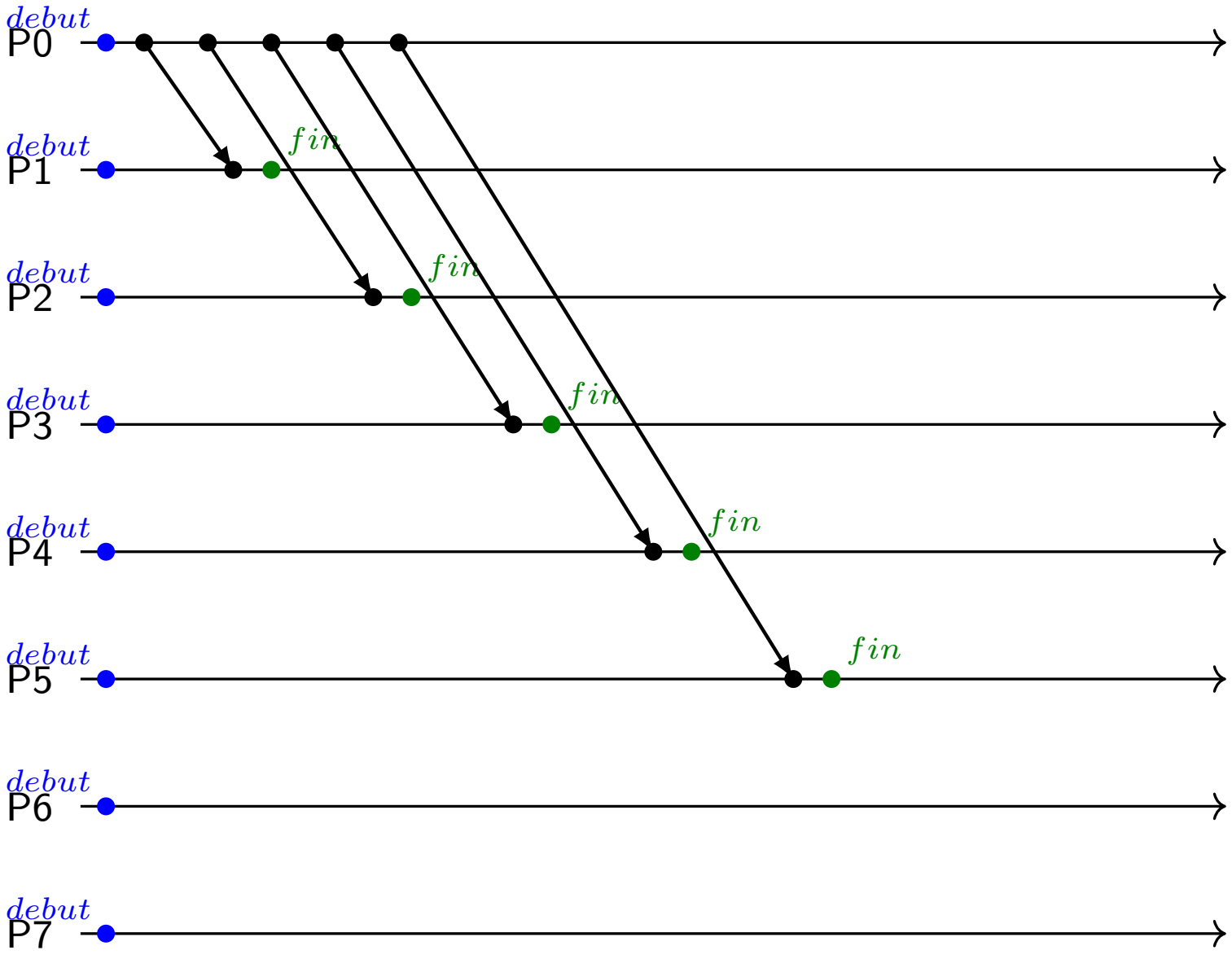
Algorithme de diffusion : l'étoile



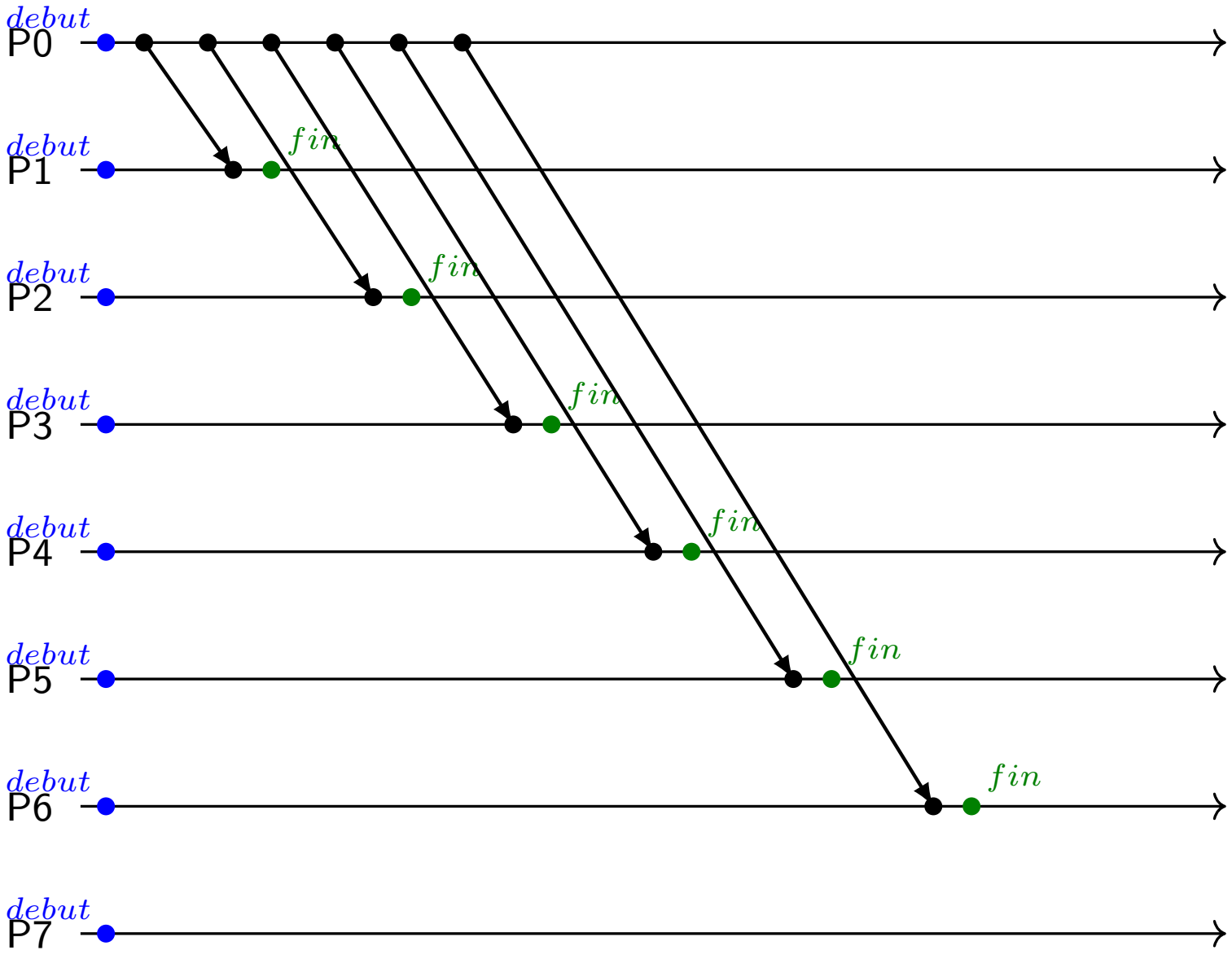
Algorithme de diffusion : l'étoile



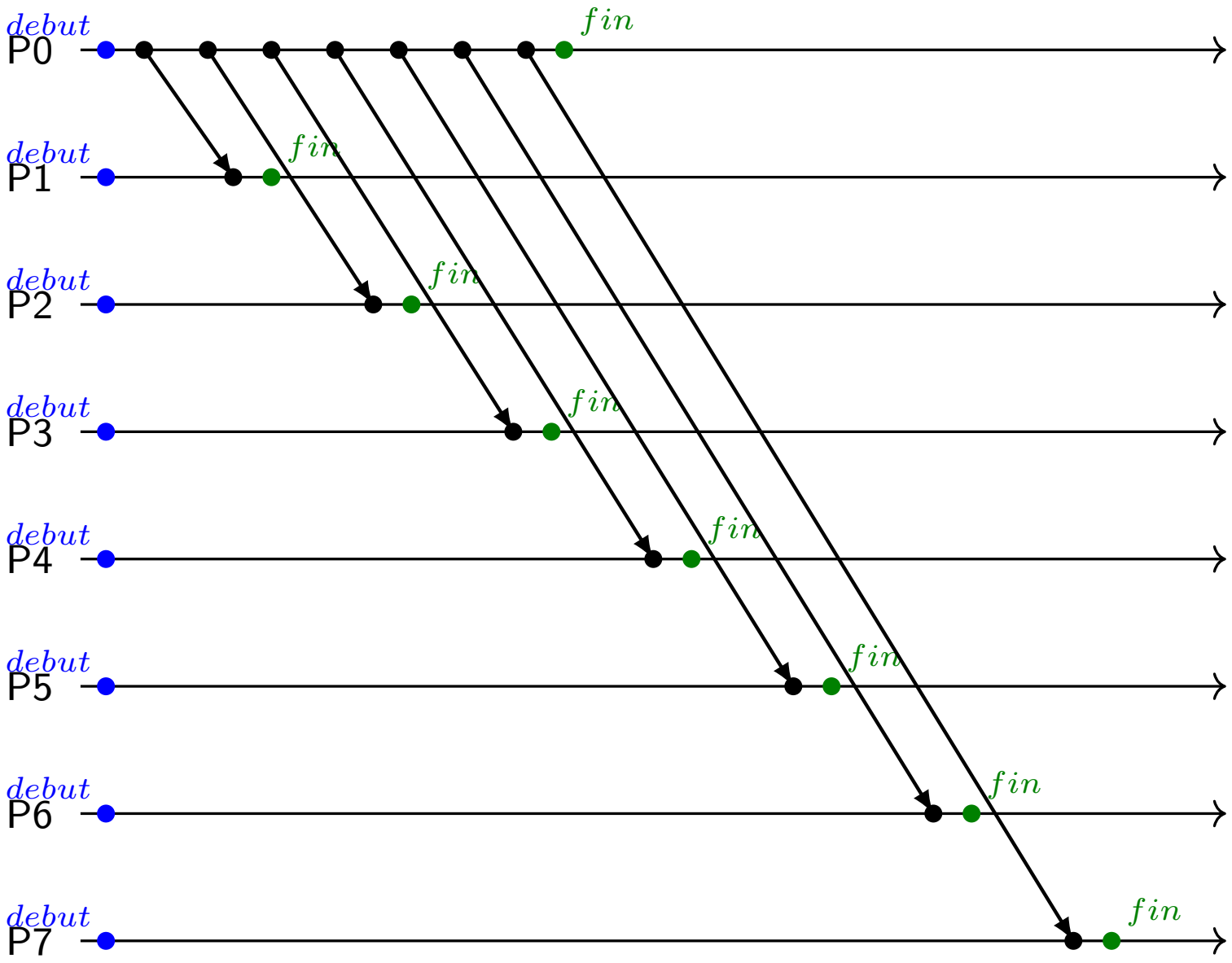
Algorithme de diffusion : l'étoile



Algorithme de diffusion : l'étoile

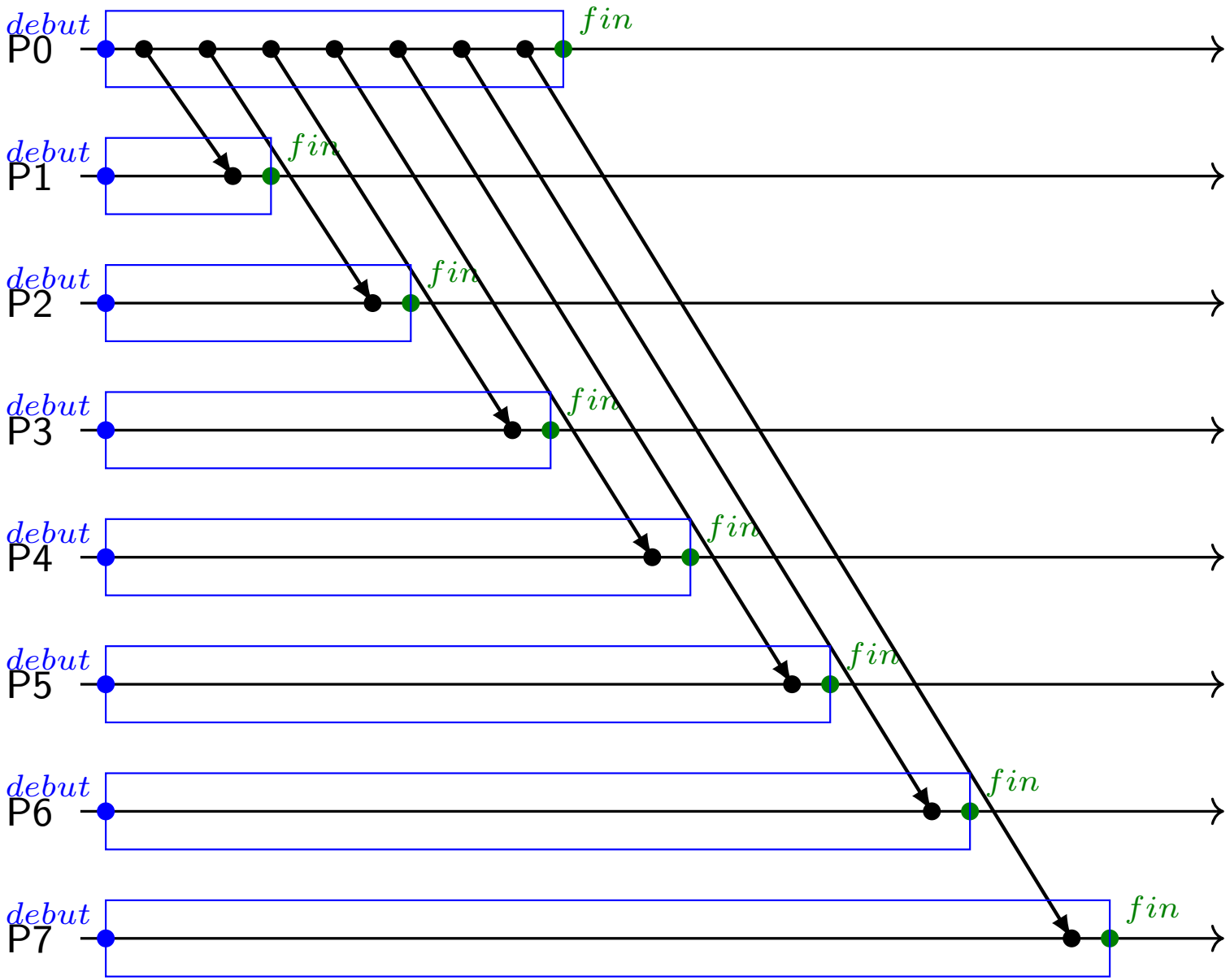


Algorithme de diffusion : l'étoile



Communications collectives

Algorithme de diffusion : l'étoile



Communications collectives

Algorithme de diffusion : arbre binomial

Idée de base : chaque processus disposant du message l'envoie à un autre processus

- 0 envoie à 1
- 0 envoie à 2, 1 envoie à 3
- 0 envoie à 4, 1 envoie à 5, 2 envoie à 6, 3 envoie à 7

Chaque processus de rang r_e envoie aux processus de rank

$$r_{dest} = r_e + 2^k \text{ avec } log_2(r_e) \leq k \leq log_2(size) \quad (2)$$

Communications collectives

Algorithme de diffusion : arbre binomial

Idée de base : chaque processus disposant du message l'envoie à un autre processus

- 0 envoie à 1
- 0 envoie à 2, 1 envoie à 3
- 0 envoie à 4, 1 envoie à 5, 2 envoie à 6, 3 envoie à 7

Chaque processus de rang r_e envoie aux processus de rank

$$r_{dest} = r_e + 2^k \text{ avec } log_2(r_e) \leq k \leq log_2(size) \quad (2)$$

Complexité :

- À chaque étape on envoie 2 fois plus de messages
- Donc $O(log_2 N)$ messages

Optimal en nombre de messages dans un modèle 1-port

- Un arbre binomial extrait le maximum de parallélisme possible dans la communication

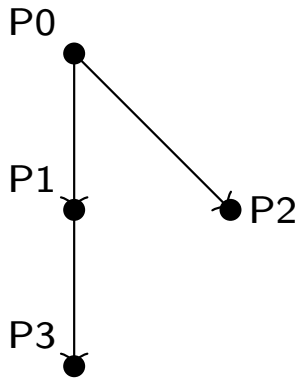
Communications collectives

Algorithme de diffusion : arbre binomial

Communications collectives

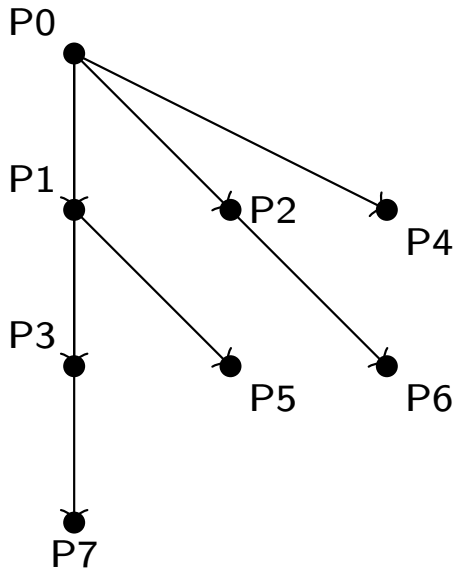
Algorithme de diffusion : arbre binomial

Algorithme de diffusion : arbre binomial



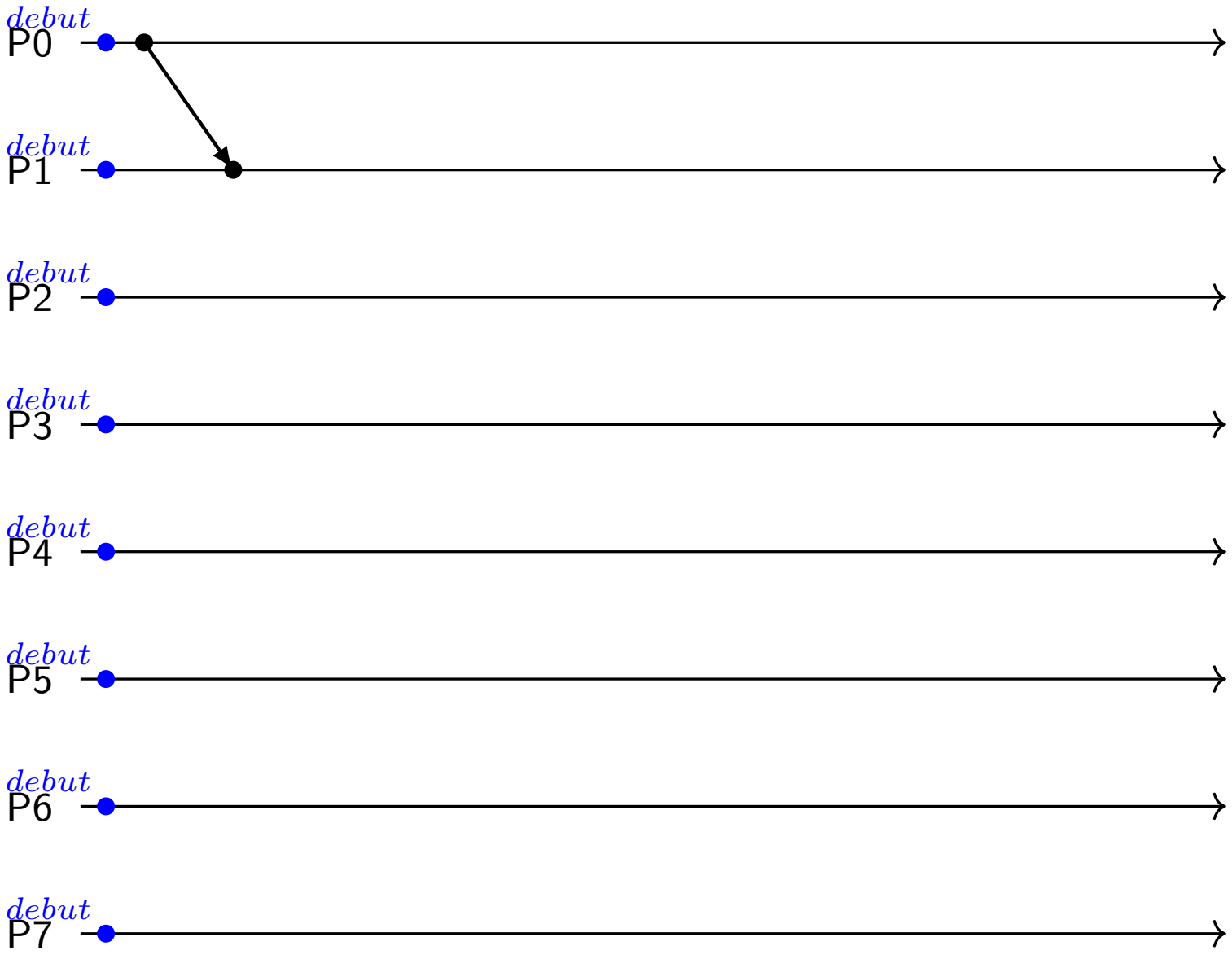
Communications collectives

Algorithme de diffusion : arbre binomial

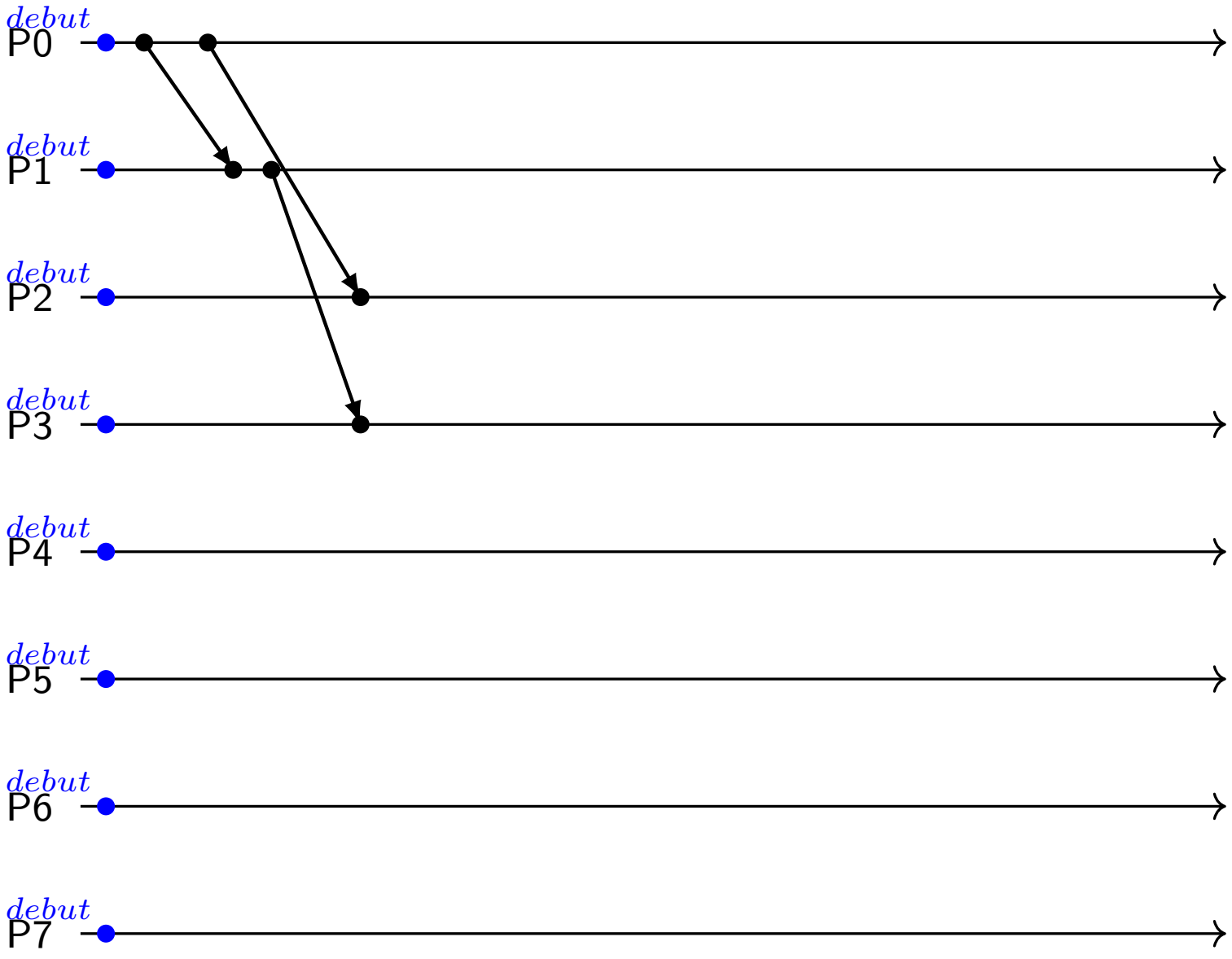


Communications collectives

Algorithme de diffusion : arbre binomial

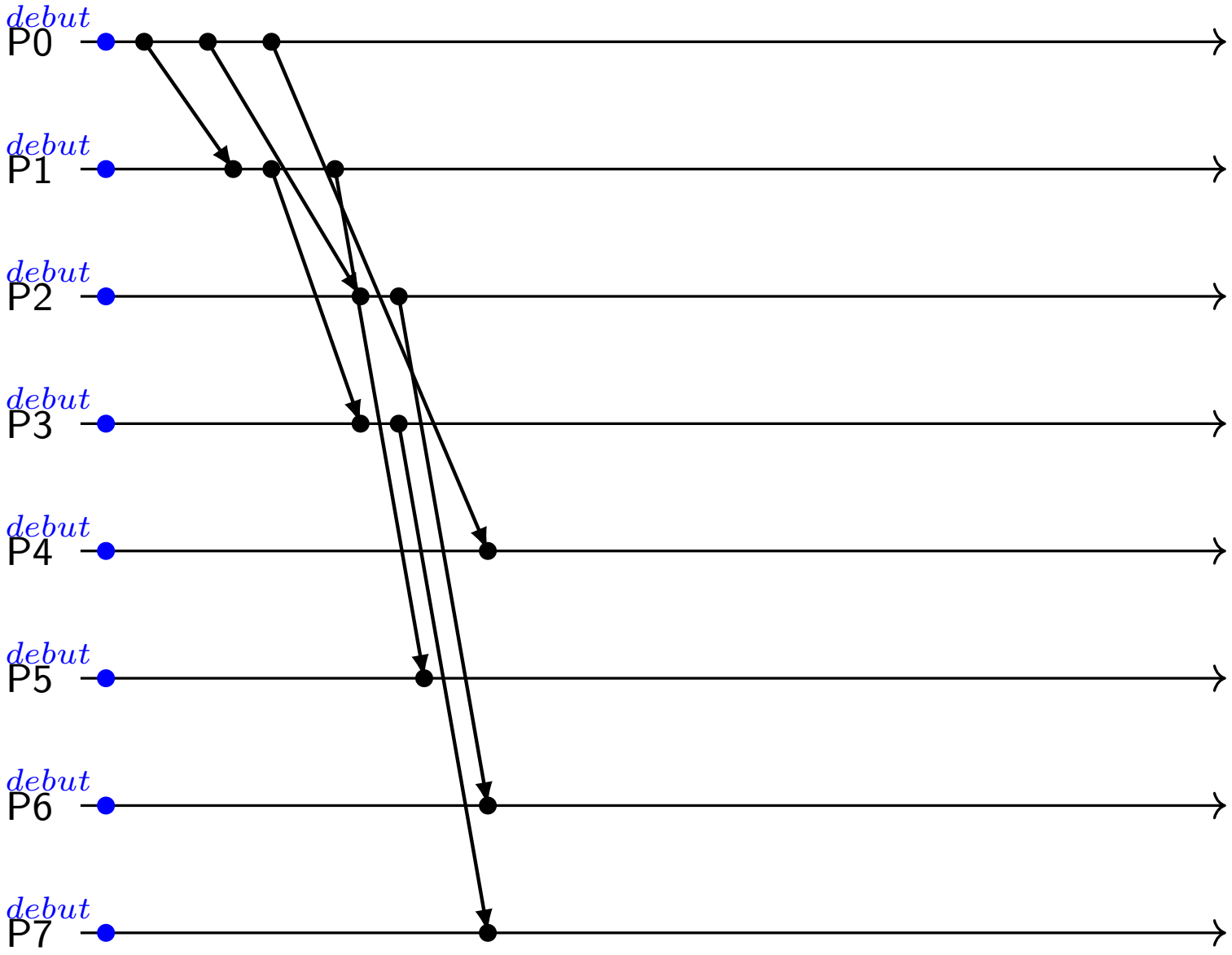


Algorithme de diffusion : arbre binomial



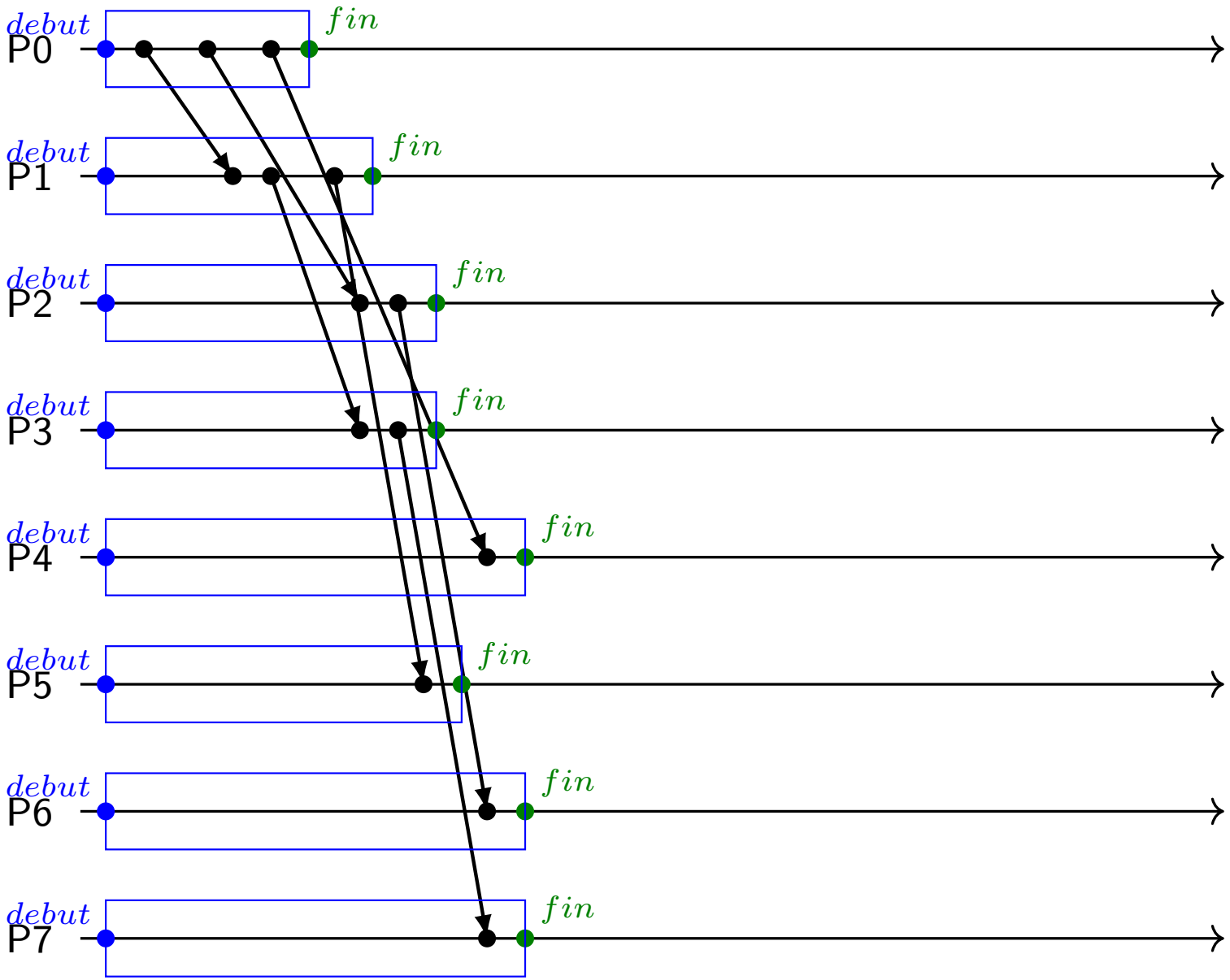
Communications collectives

Algorithme de diffusion : arbre binomial



Communications collectives

Algorithme de diffusion : arbre binomial



Communications collectives

Algorithme de diffusion : arbre de Fibonacci

Idée de base : chaque processus disposant du message l'envoie à k processus fils

- Arbre binaire = cas particulier d'arbre de Fibonacci (k=2)

Hauteur de l'arbre = $\lceil log_k(N) \rceil$
Bon dans un modèle k-port

- Les processus envoient de plus en plus de messages simultanéments quand on descend dans l'arbre
- Parallélisme de plus en plus important

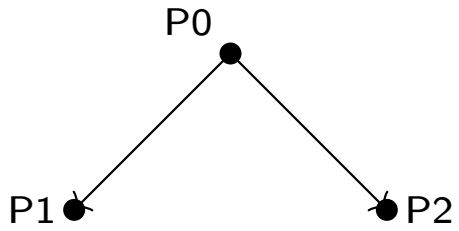
Moins intéressant que l'arbre binomial dans un modèle 1-port.

Communications collectives

Algorithme de diffusion : arbre de Fibonacci

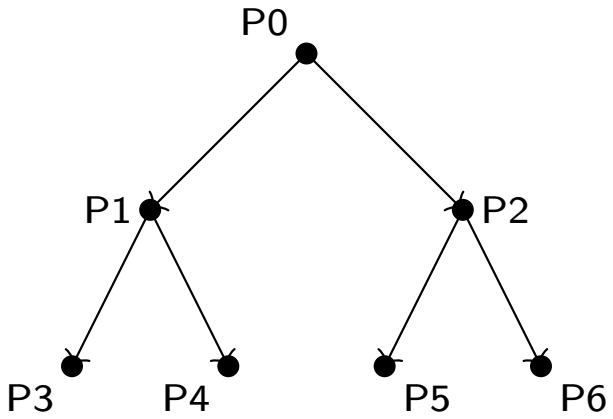
Communications collectives

Algorithme de diffusion : arbre de Fibonacci



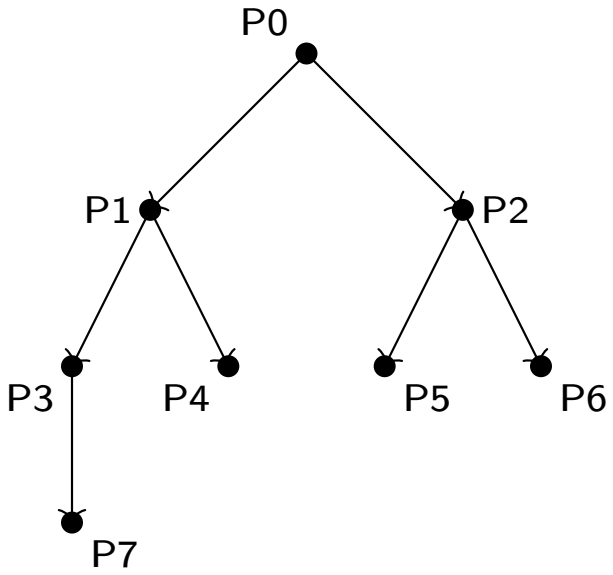
Communications collectives

Algorithme de diffusion : arbre de Fibonacci



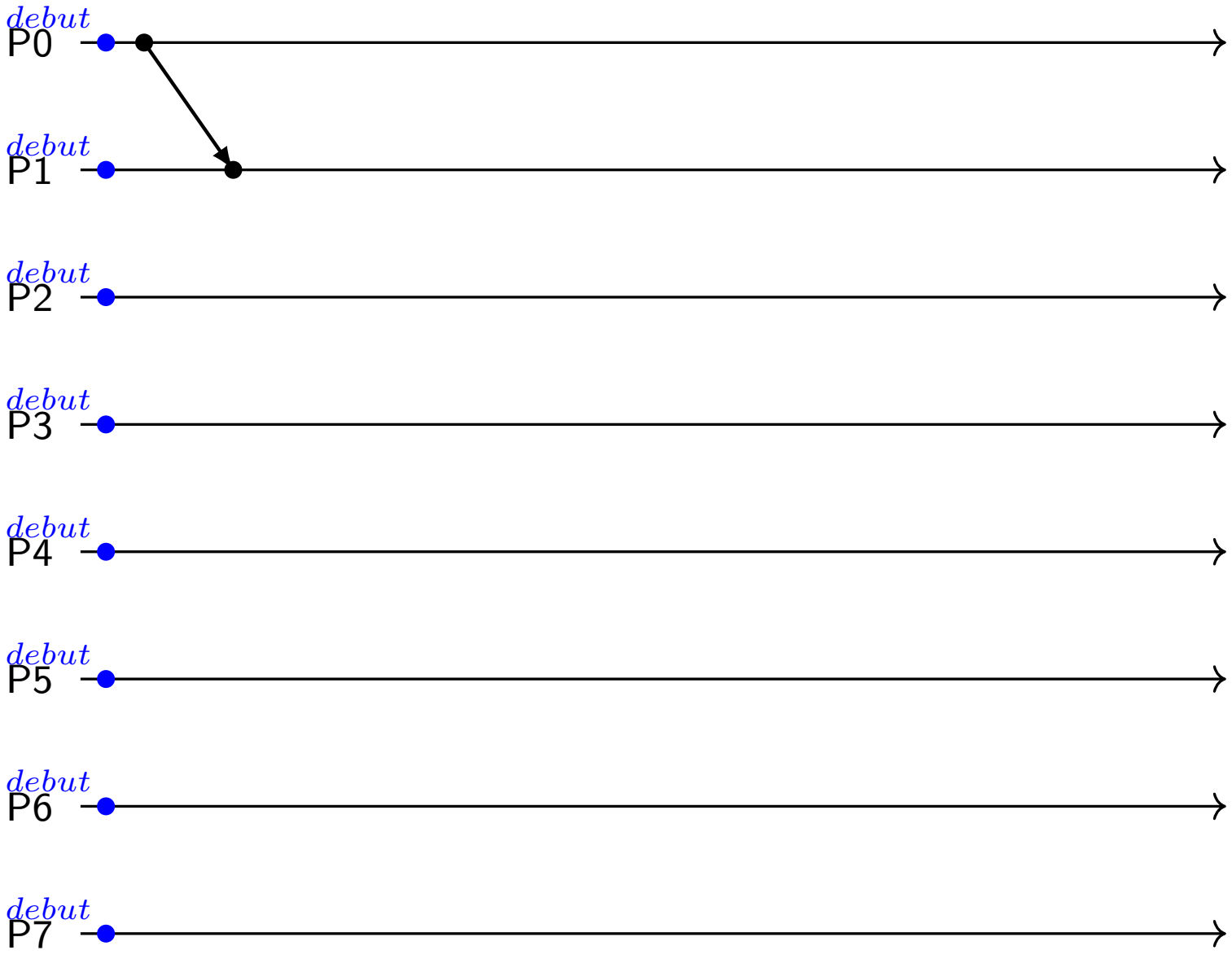
Communications collectives

Algorithme de diffusion : arbre de Fibonacci

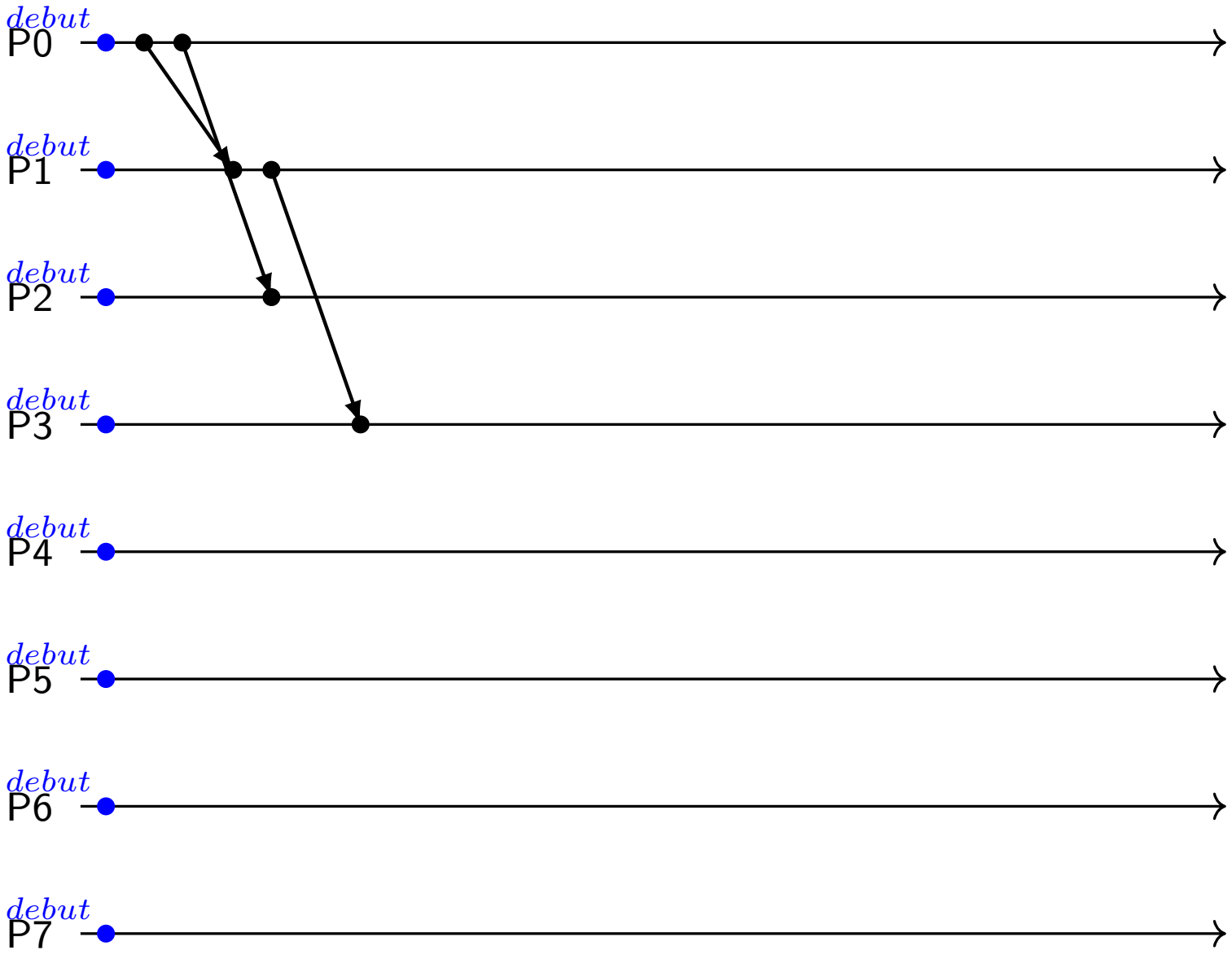


Communications collectives

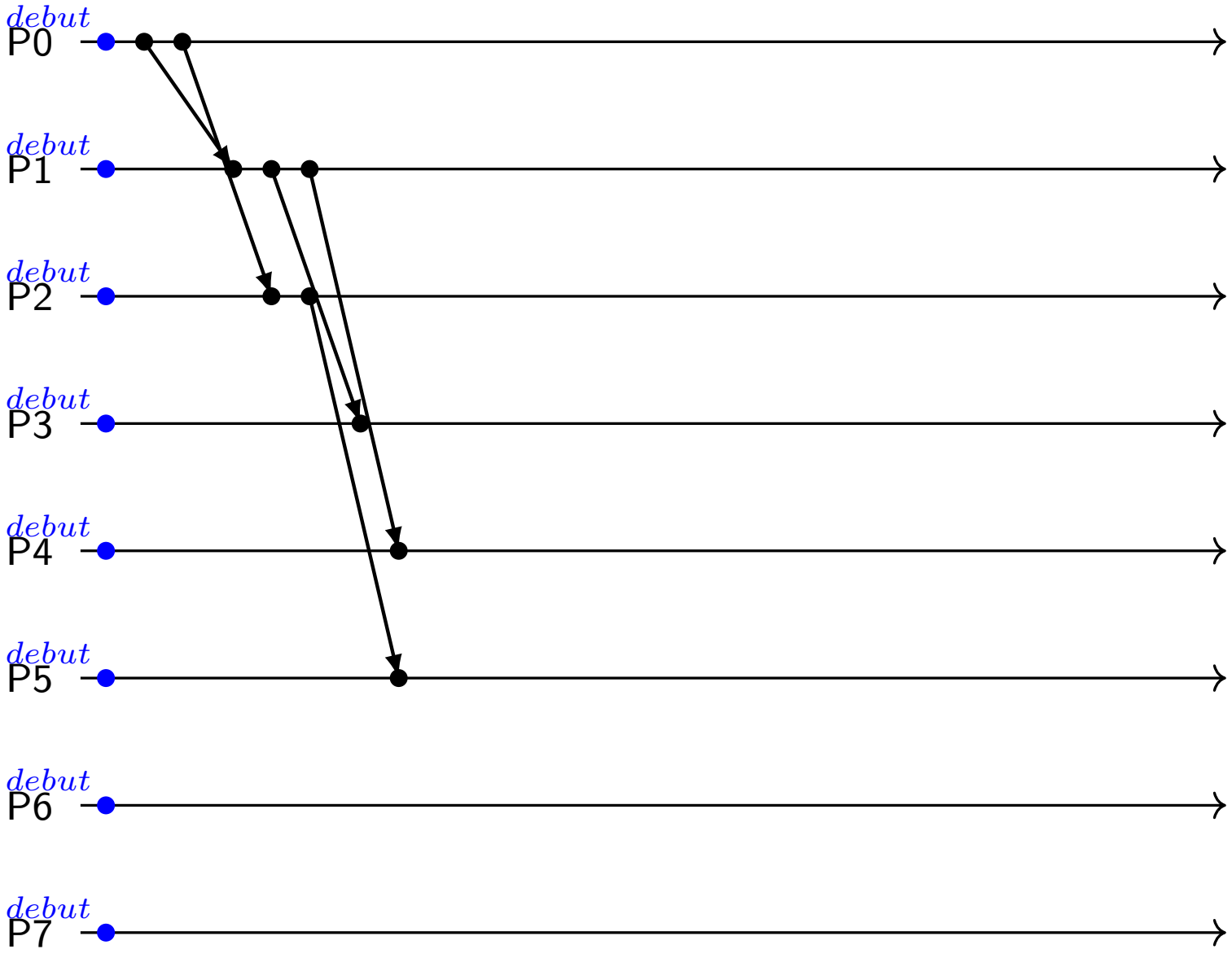
Algorithme de diffusion : arbre de Fibonacci



Algorithme de diffusion : arbre de Fibonacci

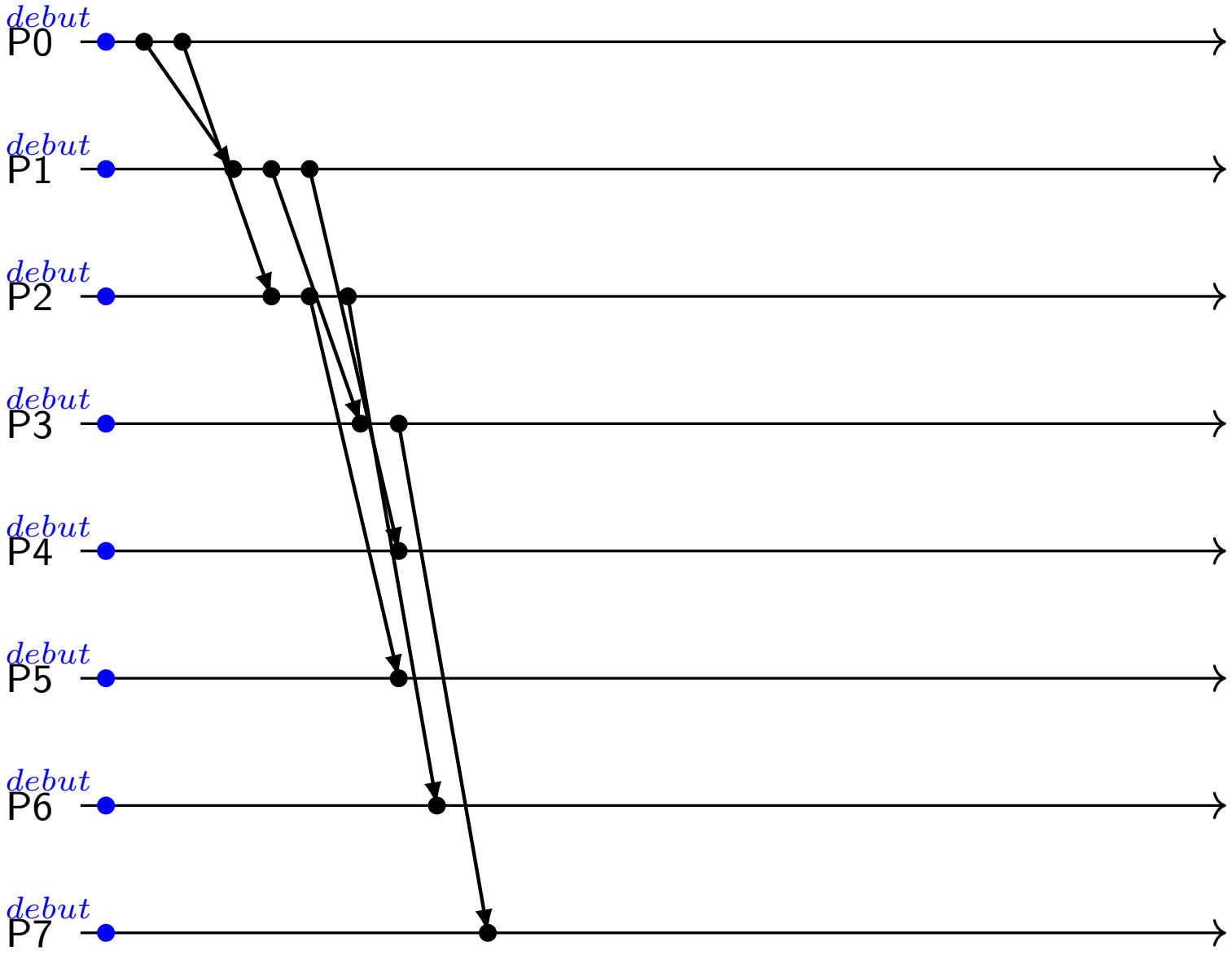


Algorithme de diffusion : arbre de Fibonacci



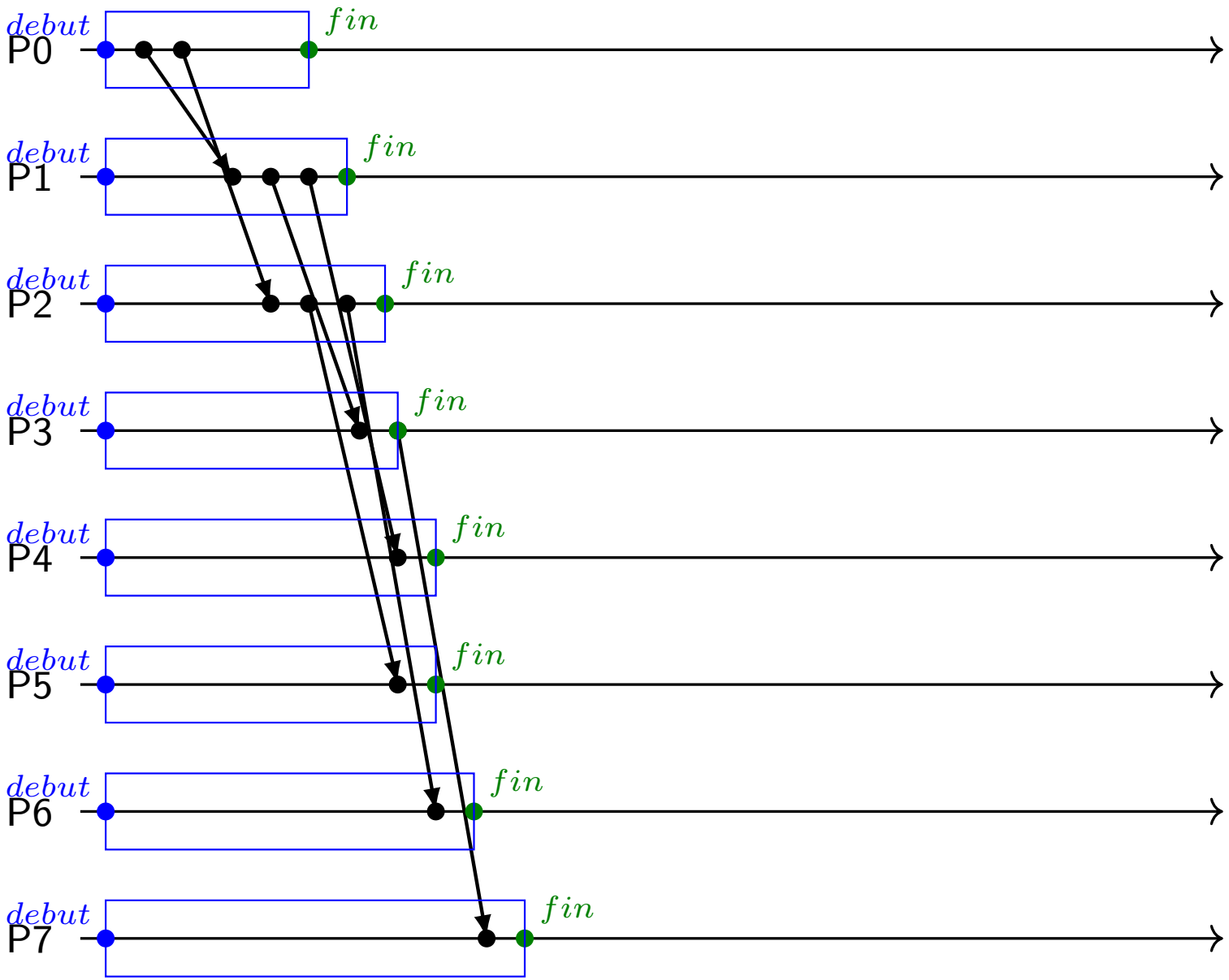
Communications collectives

Algorithme de diffusion : arbre de Fibonacci



Communications collectives

Algorithme de diffusion : arbre de Fibonacci



Communications collectives

Algorithme de diffusion : split-binary tree

Utilisation de la technique de division du message dans un arbire binaire

- La racine divise par 2 le message
- La première moitié est envoyée à son fils de droite
- La deuxième moitié est envoyée à son fils de gauche
- Les demi-messages sont transmis dans les deux sous-arbres
- Arrivé en bas, chaque processus du sous-arbre de droite échange son demi-message avec un processus du sous-arbre de gauche

Communications collectives

Algorithme de diffusion : split-binary tree

Utilisation de la technique de division du message dans un arbire binaire

- La racine divise par 2 le message
- La première moitié est envoyée à son fils de droite
- La deuxième moitié est envoyée à son fils de gauche
- Les demi-messages sont transmis dans les deux sous-arbres
- Arrivé en bas, chaque processus du sous-arbre de droite échange son demi-message avec un processus du sous-arbre de gauche

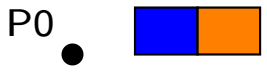
Complexités :

- Nombre de messages : un de plus qu'un arbre binaire
- La technique de division du message divise par 2 la taille du message à transmettre
- Donc on multiplie par 2 la bande passante disponible au coût d'un message supplémentaire

Intéressant pour les gros messages !

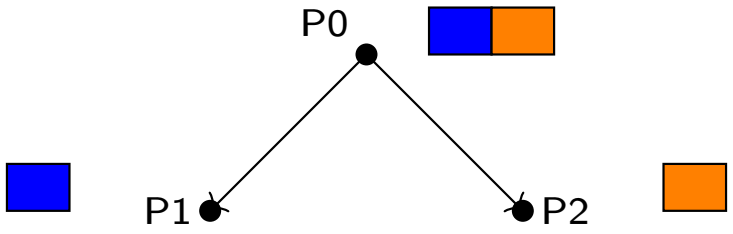
Communications collectives

Algorithme de diffusion : split-binary tree



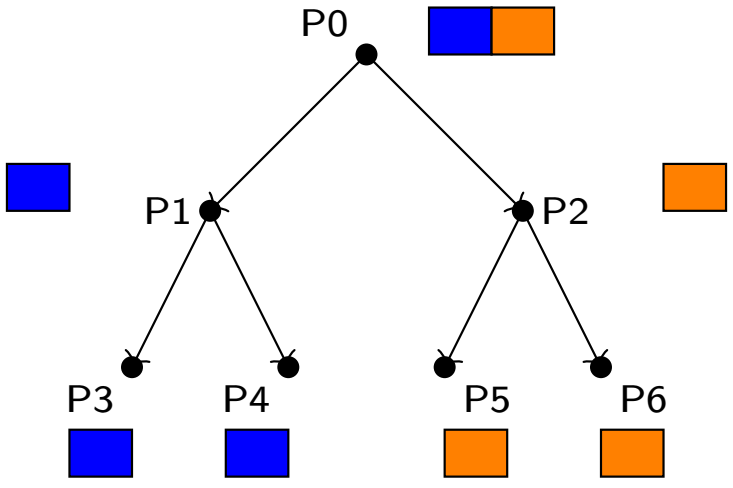
Communications collectives

Algorithme de diffusion : split-binary tree



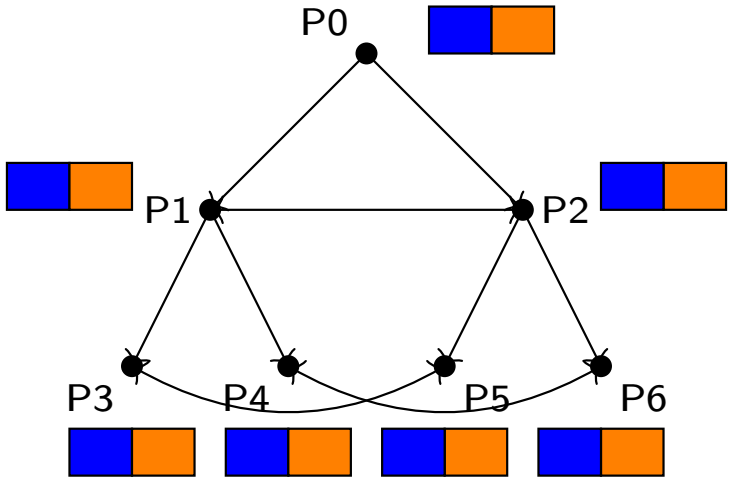
Communications collectives

Algorithme de diffusion : split-binary tree



Communications collectives

Algorithme de diffusion : split-binary tree



Communications collectives

Réduction

Réduction vers une racine

```
int MPI_Reduce( void *sendbuf, void *recvbuf, int count,
                MPI_Datatype datatype, MPI_Op op, int root,
                MPI_Comm comm );
```

- Effectue une opération (op)
  - Sur une donnée disponible sur tous les processus du communicateur
  - Vers la racine de la réduction (`root`)
- Opérations disponibles dans le standard (MPI_SUM, MPI_MAX, MPI_MIN...) et possibilité de définir ses propres opérations
  - La fonction doit être associative mais pas forcément commutative
- Pour mettre le résultat dans le tampon d'envoi (sur le processus racine) : `MPI_IN_PLACE`

Algorithme de réduction : arbre de Fibonacci

Approche naïve : considérer une réduction comme une diffusion inversée

- Pas toujours correct : l'opération peut prendre un temps non négligeable

Utilisation d'un arbre de Fibonacci

- Les processus fils envoient à leur père
- Le père fait le calcul une fois qu'il a reçu les données de ses fils
- Puis il transmet le résultat à son propre père

Mieux qu'un arbre binomial à cause de l'opération de calcul

- Doit être effectuée sur l'ensemble des données des fils
- On doit donc avoir reçu tous les buffers des fils, puis effectuer le calcul
- En ce sens, la réduction n'est donc pas une diffusion inversée !

Communications collectives

Algorithme de réduction : chaîne

Idée : établir un pipeline entre les processus

- Chaque processus découpe son buffer en plusieurs parties
- Les sous-buffers sont envoyés un par un selon une chaîne formée par les processus

Communications collectives

Algorithme de réduction : chaîne

Idée : établir un pipeline entre les processus

- Chaque processus découpe son buffer en plusieurs parties
- Les sous-buffers sont envoyés un par un selon une chaîne formée par les processus

Complexités :

- $O(N)$ messages
- Bande passante divisée par la longueur du pipeline !

Très intéressant pour des gros messages sur des petits ensembles de processus

Communications collectives

Réduction avec redistribution du résultat

Sémantique : le résultat du calcul est disponible sur tous les processus du communicateur

```
int MPI_Allreduce( void *sendbuf, void *recvbuf, int count,
                   MPI_Datatype datatype, MPI_Op op, MPI_Comm comm );
```

- Similaire à `MPI_Reduce` sans la racine

Équivalent à :

- Un Reduce
- Suivi d'un Broadcast (fan-in-fan-out)

Ce qui serait une implémentation très inefficace !

Communications collectives

Algorithme de réduction avec redistribution du résultat : Algorithme de Rabenseifner

Idée : on échange la moitié du message avec le processus à distance 2^k, avec $0 \leq k \leq log_2(size)$

- Chaque processus n'effectue donc que la moitié du calcul
- Parallélisation du calcul entre les processus !

Intéressant pour des petits messages :

- Nombre d'étapes : $O(log_2(N))$

Algorithme de réduction avec redistribution du résultat : anneau

Même principe que la chaîne de réduction

- Utilisation d'une chaîne pour calculer le résultat
- Une fois que le dernier processus de la chaîne a le résultat d'une portion du buffer, il l'envoie à son voisin dans l'anneau
- Le résultat circule pendant que le reste du buffer est calculé
- Établissement d'un pipeline de redistribution du résultat

Intéressant pour des gros messages : bande passante multipliée par la longueur du pipeline

Communications collectives

Distribution

Distribution d'un tampon vers plusieurs processus

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

- Des fractions de taille `sendcount` de tampon d'envoi disponible sur la racine sont envoyés vers tous les processus du communicateur
- Possibilité d'utiliser `MPI_IN_PLACE`

Communications collectives

Distribution

Distribution d'un tampon vers plusieurs processus

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

- Des fractions de taille `sendcount` de tampon d'envoi disponible sur la racine sont envoyés vers tous les processus du communicateur
- Possibilité d'utiliser `MPI_IN_PLACE`

Deux possibilités :

- Linéaire : la racine envoie à tous les autres noeuds
- Arbre binomial : on envoie à un noeud le tampon qui lui est destiné ainsi que ceux destinés à ses enfants

Communications collectives

Concaténation vers un point

Concaténation du contenu des tampons

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

- Les contenus des tampons sont envoyés vers la racine de la concaténation
- Possibilité d'utiliser des datatypes différents en envoi et en réception (attention, source d'erreurs)
- `recvbuf` ne sert que sur la racine
- Possibilité d'utiliser `MPI_IN_PLACE`

Communications collectives

Concaténation vers un point

Concaténation du contenu des tampons

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

- Les contenus des tampons sont envoyés vers la racine de la concaténation
- Possibilité d'utiliser des datatypes différents en envoi et en réception (attention, source d'erreurs)
- `recvbuf` ne sert que sur la racine
- Possibilité d'utiliser `MPI_IN_PLACE`

Même chose que pour la distribution :

- Linéaire : la racine collecte le tampon de tous les autres processus
- Arbre binomial : chaque processus envoie à son père, qui rassemble les tampons de tous ses enfants et envoie le résultat à son père

Communications collectives

Concaténation avec redistribution du résultat

```
int MPI_Allgather( void *sendbuf, int  sendcount,
                   MPI_Datatype sendtype, void *recvbuf, int recvcount,
                   MPI_Datatype recvtype, MPI_Comm comm );
```

- Similaire à `MPI_Gather` sans la racine

- Approche naïve : gather + broadcast
- Pour les gros messages : anneau
- Algorithme de Bruck

Communications collectives

Algorithme de Bruck

Idée : on concatène les tampons et on échange son tampon résultat avec un processus qui a les tampons d'autres processus

```
for( i = 0 ; i < log2(size) ; i++ ) {
  copain = rank XOR 2^i
  sendrecv( tamponlocal, resultat, copain )
  concatener( tamponlocal, resultat)
}
```

Complexité :

- $O(log_2(N))$ messages échangés
- Les messages sont de plus en plus gros
- Nécessité d'un tampon local temporaire

Intéressant pour les messages petits à moyens

Communications collectives

Barrière de synchronisation

Sémantique : un processus ne sort de la barrière qu'une fois que tous les autres processus y sont entrés

```
MPI_Barrier( MPI_Comm comm );
```

- Apporte une certaine synchronisation entre les processus : quand on dépasse ce point, on sait que tous les autres processus l'ont au moins atteint
- Équivalent à un Allgather avec un message de taille nulle

Algorithmes utilisés :

- gather / reduce suivi d'un broadcast = approche naïve
- Bruck = efficace sur petits messages donc bon pour la barrière

Communications collectives

Distribution et concaténation de données

Distribution d'un tampon de tous les processus vers tous les processus

```
int MPI_Alltoall( void *sendbuf, int sendcount,
            MPI_Datatype sendtype, void *recvbuf, int recvcount,
            MPI_Datatype recvtype, MPI_Comm comm );
```

- Sur chaque processus, le tampon d'envoi est découpé et envoyé vers tous les processus du communicateur
- Chaque processus reçoit des donnés de tous les autres processus et les concatène dans son tampon de réception
- PAS de possibilité d'utiliser `MPI_IN_PLACE`

Algorithmes :

- Bruck : originalement conçu pour Alltoall
- Linéaire (pairewise exchange) :

```
for( i = 0 ; i < size ; i++ ) {
  if( i != rank ) {
    sendrecv( envoi[i], resultat[i], i );
  }
}
```

Plan du cours

Plan du cours

Plan du cours

Utilisation des datatypes MPI

Principe

- On définit le datatype
  - `MPI_Type_contiguous`, `MPI_Type_vector`, `MPI_Type_hvector`, `MPI_Type_indexed`, `MPI_Type_hindexed`, `MPI_Type_struct`
- On le commit
  - `MPI_Type_commit`
- On le libère à la fin
  - `MPI_Type_free`

Combinaison des types de base

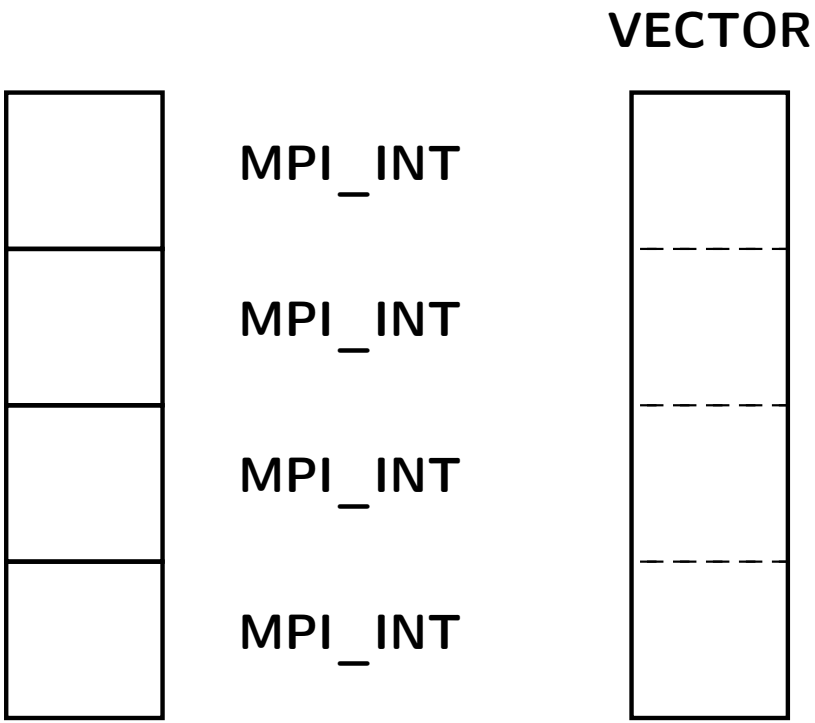
`MPI_CHAR`, `MPI_SHORT`, `MPI_INT`, `MPI_LONG`, `MPI_UNSIGNED_CHAR`, `MPI_UNSIGNED_SHORT`, `MPI_UNSIGNED_LONG`, `MPI_UNSIGNED`, `MPI_FLOAT`, `MPI_DOUBLE`, `MPI_LONG_DOUBLE`, `MPI_BYTE`

Construction de datatypes MPI

Données contiguës

On crée un block d'éléments :

- `int MPI_Type_contiguous( int count, MPI_Datatype oldtype, MPI_Datatype *newtype );`



Construction de datatypes MPI

Vecteur de données

On crée un vecteur d'éléments :

- `int MPI_Type_vector( int count, int blocklength, int stride MPI_Datatype oldtype, MPI_Datatype *newtype );`

On aggrège des blocs de *blocklength* éléments séparés par un vide de *stride* éléments.

Construction de datatypes MPI

Structure générale

On donne les éléments, leur nombre et l'offset auquel ils sont positionnés.

- `int MPI_Type_struct( int count, int *array_of_blocklengths, MPI_Aint *array_of_displacements, MPI_Datatype *array_of_types, MPI_Datatype *newtype);`

Exemple

On veut créer une structure MPI avec un entier et deux flottants à double précision

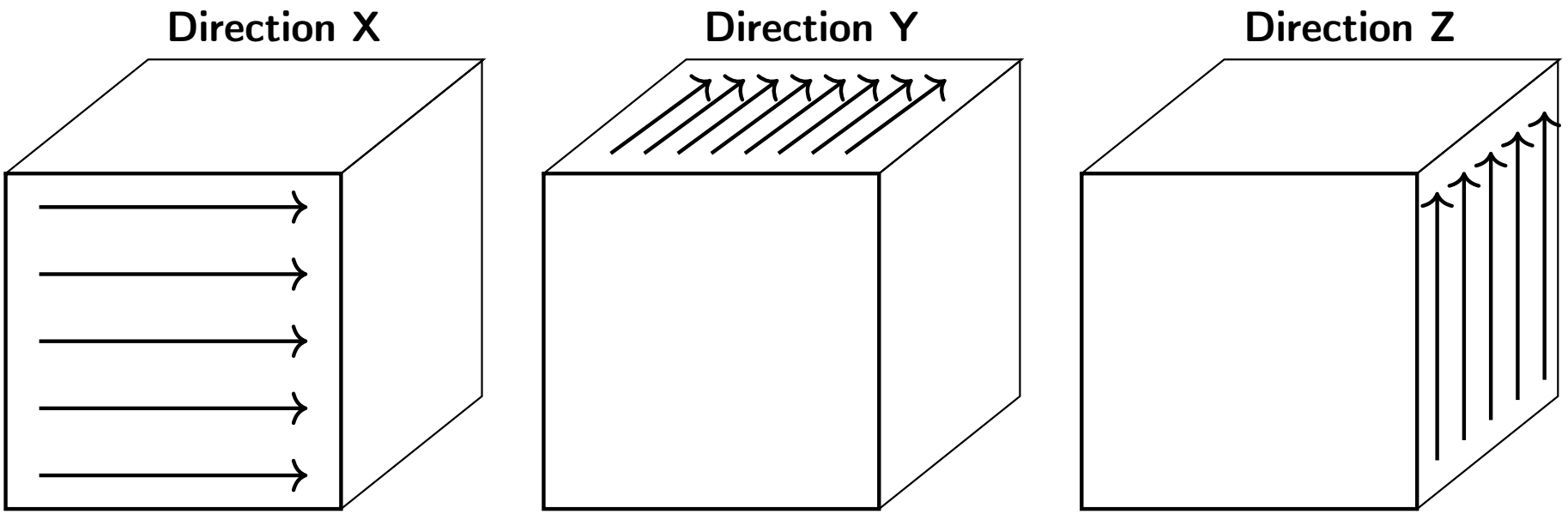
- count = 2 : on a deux éléments
- array_of_blocklengths = 1, 2 : on a 1 entier et 2 doubles
- array_of_displacements = 0, sizeof(int), sizeof(int) + sizeof(double) (ou utilisation de MPI_Address sur une instance de cette structure)
- array_of_types = MPI_INT, MPI_DOUBLE

Exemple d'utilisation : FFT 3D

La FFT 3D

Calcul de la transformée de Fourier 3D d'une matrice 3D

- FFT dans une dimension
- FFT dans la deuxième dimension
- FFT dans la troisième dimension



Parallélisation de la FFT 3D

Noyau de calcul : la FFT 1D

La FFT 1D est parallélisable

- Chaque vecteur peut être calculé indépendamment des autres

On veut calculer chaque vecteur *séquentiellement* sur *un seul processus*

- Direct pour la 1ere dimension
- Nécessite une transposition pour les dimensions suivantes

Transposition de la matrice

On effectue une rotation de la matrice

- Redistribution des vecteurs
  - All to All
- Rotation des points (opération locale)

Rotation des points

Algorithme

```
MPI_Alltoall(tab 2 dans tab 1 )
for( i = 0; i < c; ++i ) {
  for( j = 0; j < b; ++j ) {
    for( k = 0; k < a; ++k ) {
      tab2[i][j][k] = tab2[i][k][j];
    }
  }
}
```

Complexité

- Le Alltoall coûte cher
- La rotation locale est en $O(n^3)$

On essaye d'éviter le coût de cette rotation en la faisant faire par la bibliothèque MPI

Dataypes non-contigüs

Sérialisation des données en mémoire

La matrice est sérialisée en mémoire

- Vecteur$[0][0]$, Vecteur$[0][1]$, Vecteur$[0][2]$... Vecteur$[1][0]$, etc

Rotation des données sérialisées

x	x	x	x	$\rightarrow$	x	o	o	o
o	o	o	o		x	o	o	o
o	o	o	o		x	o	o	o
o	o	o	o		x	o	o	o

Utilisation d'un datatype MPI

C'est l'écart entre les points des vecteur qui change avant et après la rotation $\rightarrow$ on définit un dataype différent en envoi et en réception.
Avantages :

- La rotation est faite par la bibliothèque MPI
- Gain d'une copie (buffer) au moment de l'envoi / réception des éléments (un seul élément au lieu de plusieurs)

Plan du cours

Définition d'opérations

Syntaxe

- `int MPI_Op_create( MPI_User_function *function, int commute, MPI_Op *op );`

On fournit un pointeur sur fonction. La fonction doit être associative et peut être commutative ou pas. Elle a un prototype défini.

Exemple

- Définition d'une fonction :

```
void addem( int *, int *, int *, MPI\_Datatype * );
void addem( int *invec, int *inoutvec, int *len,
            MPI\_Datatype *dtype ){
   int i;
   for ( i = 0 ; i < *len ; i++ ){
       inoutvec[i] += invec[i];
   }
}
```

- Déclaration de l'opération MPI :

```
MPI_Op_create( (MPI_User_function *)addem, 1, &op );
```

Exemple d'utilisation : TSQR

Definition

La *décomposition QR* d'une matrice A est une décomposition de la forme

$$A = QR$$

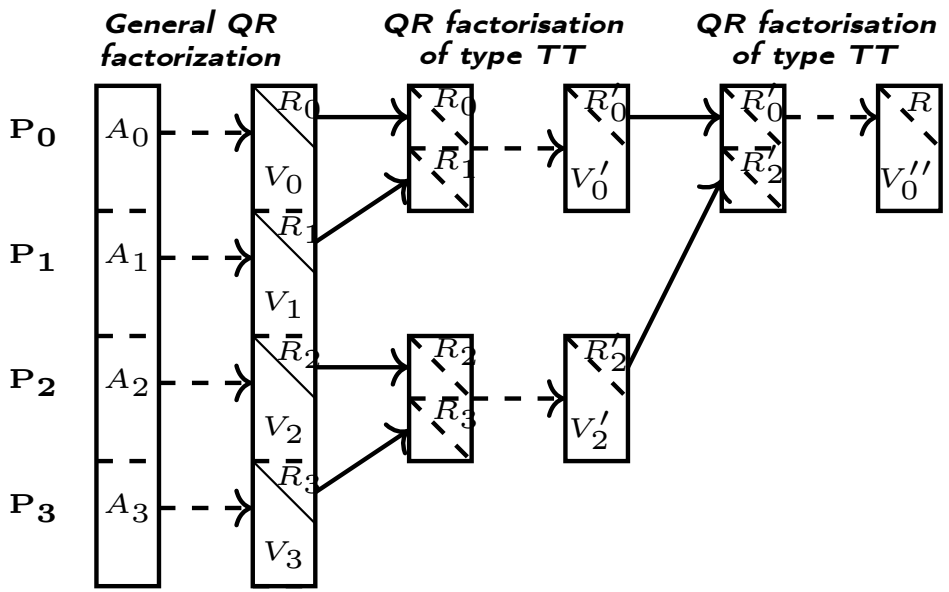
Où Q est une matrice orthogonale ($QQ^T = I$)
et R est une matrice triangulaire supérieure.

TSQR basé sur CAQR

Algorithme à évitement de communications pour matrices "tall and skinny" (hauteur $>>$ largeur)

- On calcule plus pour communiquer moins (optimal en nombre de communications)
  - Les communications coûtent cher, pas les flops
- Calcul de facto QR partielle sur des sous-domaines (en parallèle), recomposition 2 à 2, facto QR, etc

Algorithme TSQR



Algorithme

Sur un arbre binaire :

- QR sur sa sous-matrice
- Communication avec le voisin
  - Si rang pair : réception de $r + 1$
  - Si rang impair : envoi à $r - 1$
- Si rang impair : fin

Arbre de réduction

Opération effectuée

On effectue à chaque étape de la réduction une factorisation QR des deux facteurs R mis l'un au-dessus de l'autre :

$$R = QR(R_1, R_2)$$

- C'est une opération binaire
  - Deux matrices triangulaires en entrée, une matrice triangulaire en sortie
- Elle est associative

$$QR(R_1, QR(R_2, R_3)) = QR(QR(R_1, R_2), R_3)$$

- Elle est commutative

$$QR(R_1, R_2) = QR(R_2, R_1)$$

Utilisation de MPI_Reduce

L'opération remplit les conditions pour être une MPI_Op dans un MPI_Reduce

- Définition d'un datatype pour les facteurs triangulaires supérieurs R
- Utilisation de ce datatype et de l'opération QR dans un MPI_Reduce

Plan du cours

Plan du cours

Maître-esclave

Distribution des données

Le maître distribue le travail aux esclaves

- Le maître démultiplexe les données, multiplexe les résultats
- Les esclaves ne *communiquent pas* entre eux

Efficacité

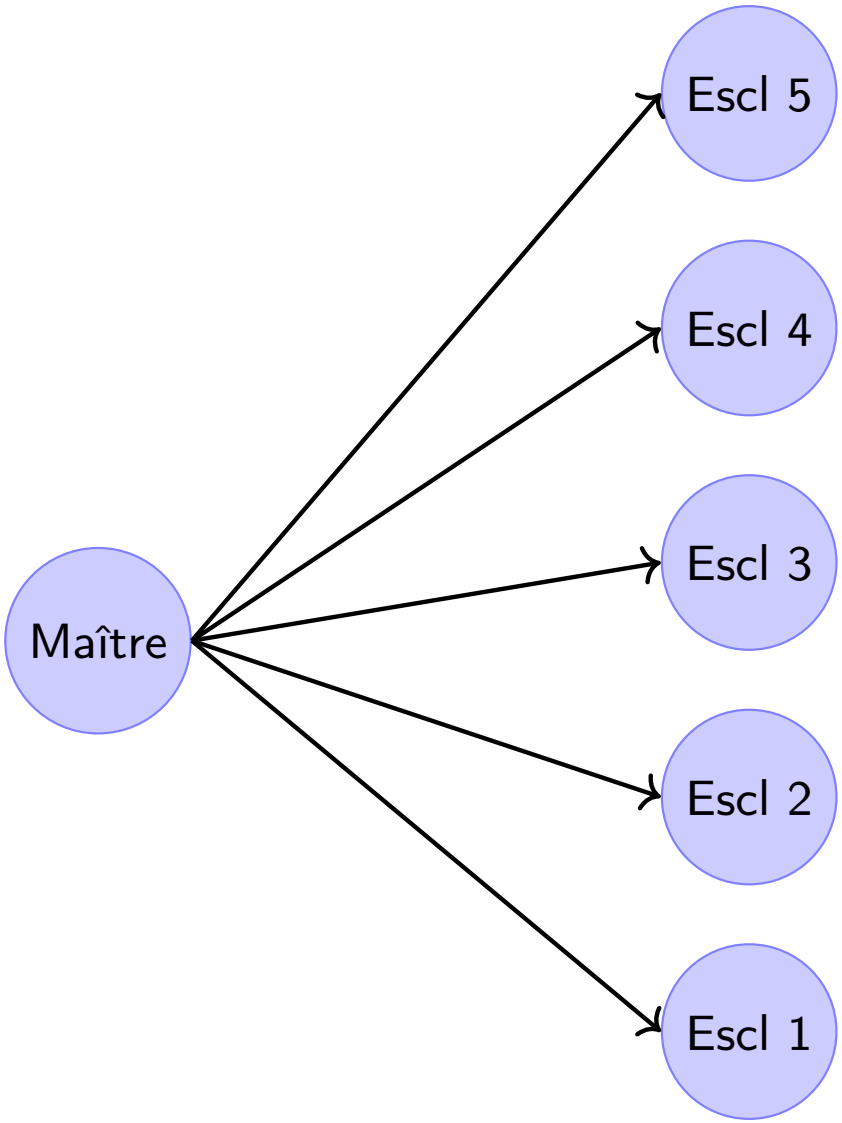
Files d'attentes de données et résultats au niveau du maître

- On retrouve la partie séquentielle de la loi d'Amdahl
- Communications : maître $\leftrightarrow$ esclaves
- Les esclaves ne travaillent pas quand ils attendent des données ou qu'ils envoient leurs résultats

Seuls les esclaves participent effectivement au calcul

- Possilibité d'un bon speedup à grande échelle (esclaves $>>$ maître) *si les communications sont rares*
- Peu rentable pour quelques processus
- Attention au bottleneck au niveau du maître

Maître-esclave



Équilibrage de charge

Statique :

- Utilisation de `MPI_Scatter` pour distribuer les données
- `MPI_Gather` pour récupérer les résultats

Dynamique :

- Mode *pull* : les esclaves demandent du travail
- Le maître envoie les chunks 1 par 1

Plan du cours

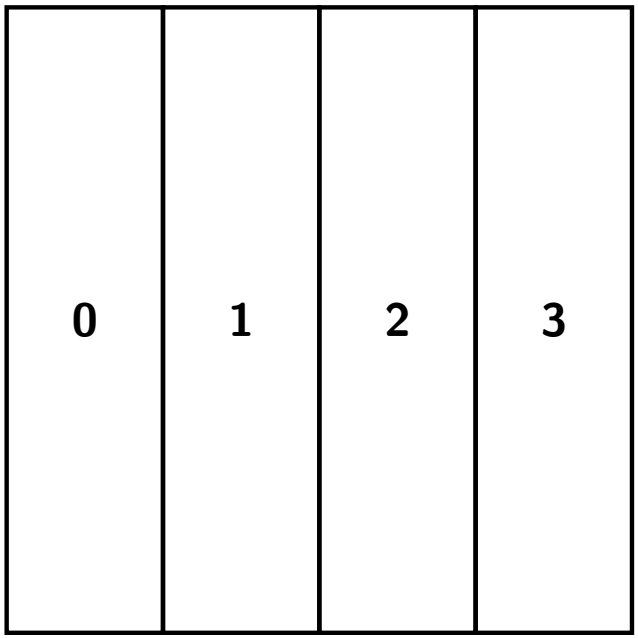
Découpage en grille

Grille de processus

On découpe les données et on attribue un processus à chaque sous-domaine

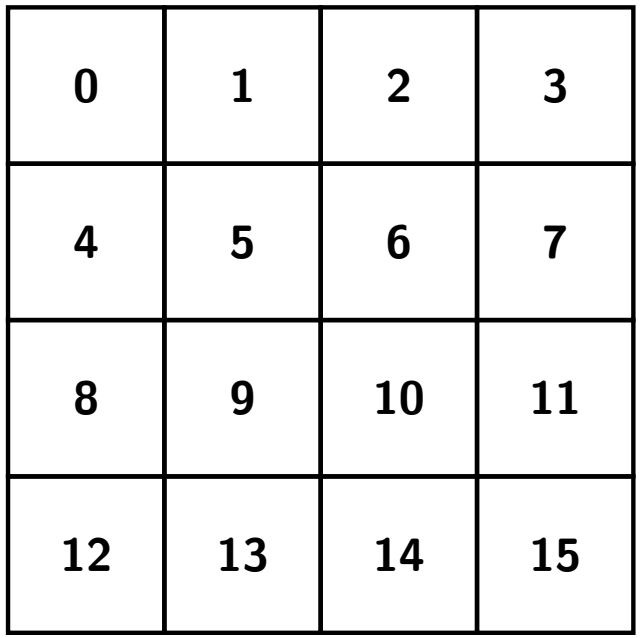
Décomposition 1D

Découpage en bandes



Décomposition 2D

Découpage en rectangles, plus scalable



Ghost region

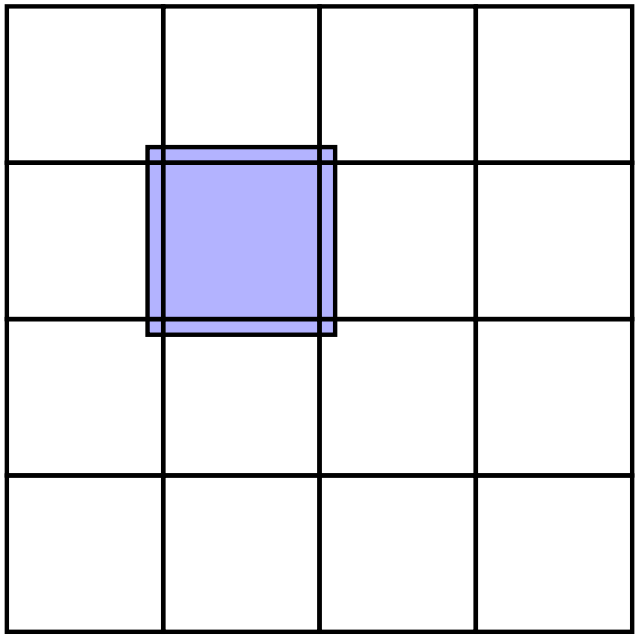
Frontières entre les sous-domaines

Un algorithme peut avoir besoin des valeurs des points voisins pour calculer la valeur d'un point

- Traitement d'images (calcul de gradiant...), automates cellulaires...

Réplication des données situées à la frontière

- Chaque processus dispose d'un peu des données des processus voisins
- Mise à jour à la fin d'une étape de calcul



Plan du cours

Utilisation d'une topologie

Décomposition en structure géométrique

On transpose un communicateur sur une topologie cartésienne

- `int MPI_Cart_create ( MPI_Comm comm_old, int ndims, int *dims, int *periods, int reorder, MPI_Comm *comm_cart );`

0	**1**	**2**	**3**
4	**5**	**6**	**7**
8	**9**	**10**	**11**
12	**13**	**14**	**15**

En 2D

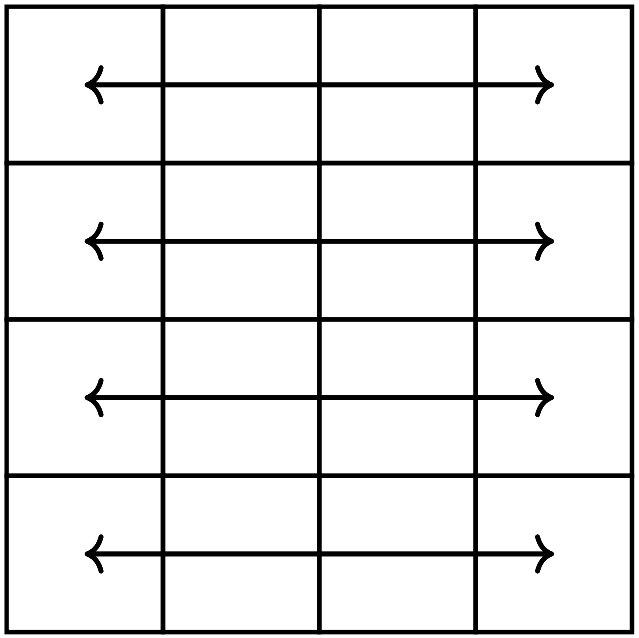
- `comm_old` : le communicateur de départ
- `ndims` : ici 2
- `dims` : nombre de processus dans chaque dimension (ici {4, 4})
- `periods` : les dimensions sont-elles périodiques
- `reorder` : autoriser ou non de modifier les rangs des processus
- `comm_cart` : nouveau communicateur

Utilisation d'une topologie

Extraction de sous-communicateurs

Communicateur de colonnes, de rangées...

- `int MPI_Cart_sub( MPI_Comm comm_old, int *remain_dims, MPI_Comm *comm_new );`



Communicateurs de lignes

- `comm_old` : le communicateur de départ
- `remain_dims` : quelles dimensions sont dans le communicateur ou non
- `comm_new` : le nouveau communicateur

Plan du cours

Plan du cours

Principe du RMA

Remote Memory Access

Accès mémoire distant

- Accès direct à la mémoire d'un autre processus
- Lecture (*get*) ou écriture (*put*)
- Le processus cible **n'intervient pas** dans le transfert

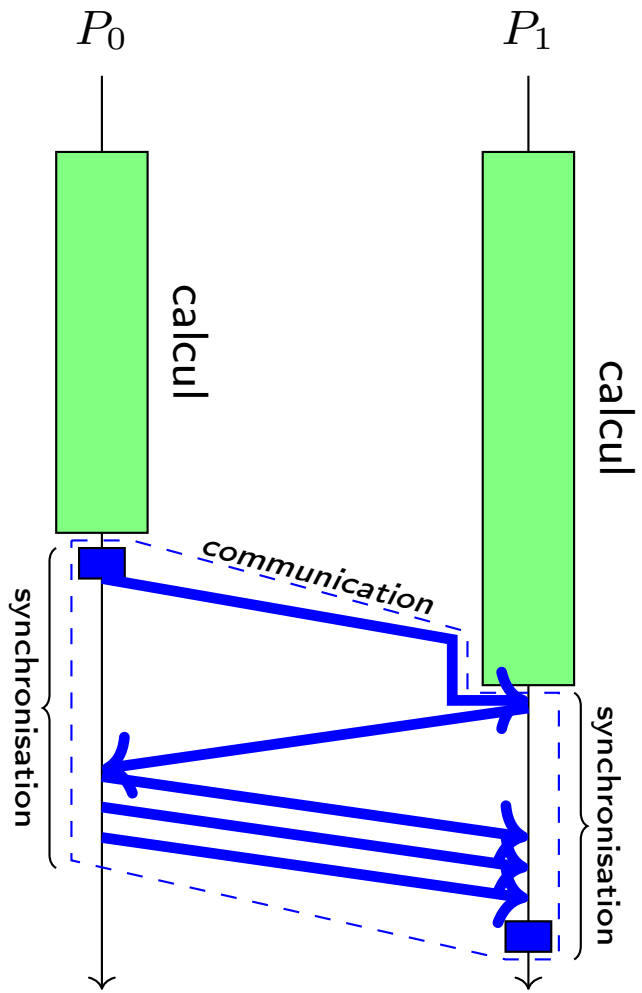

Permet de tirer parti de matériel spécialisé (DMA, coprocesseur, réseau rapide InfiniBand...)

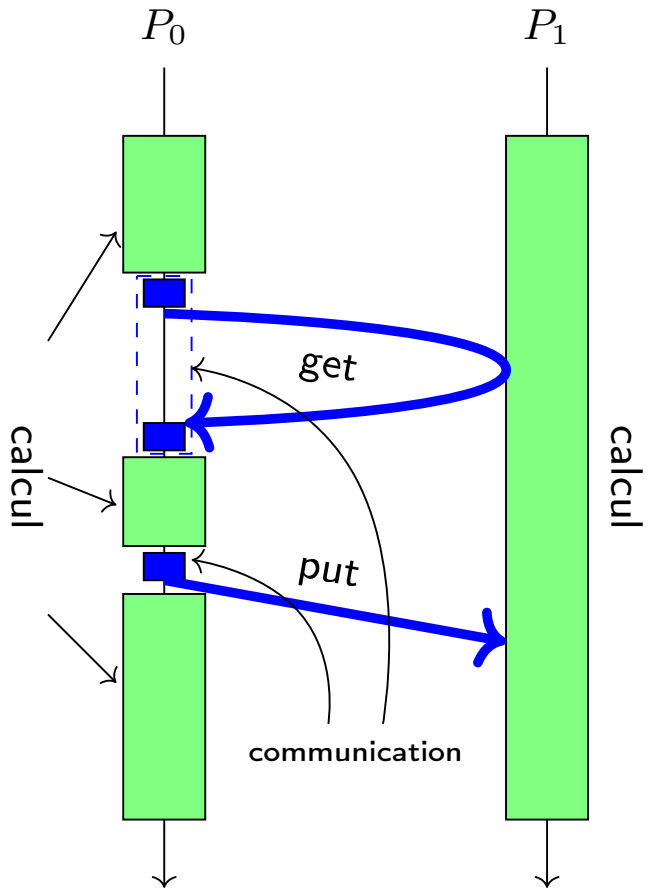

Plus efficace sur certains algorithmes (pas de synchronisation)


Plus complexe à programmer, risque d'erreurs, race conditions


Moins performant sur certains matériels (non adaptés)

Unilatérales / bilatérales



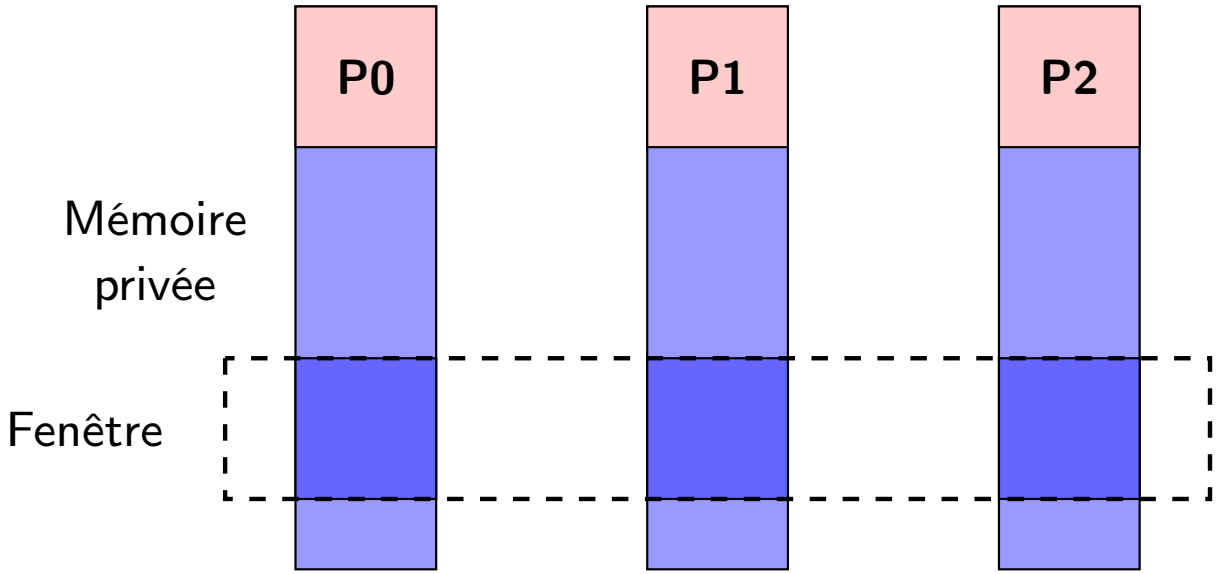


Plan du cours

Mémoire accédée

On n'accède pas à *toute* la mémoire des autres processus !

- Création de **fenêtres** accessibles



Fonctionnement global des communications unilatérales

1. Création d'une fenêtre mémoire
2. Accès distants en lecture et en écriture
3. Destruction de la fenêtre

Création de fenêtre

Création de fenêtre

- `MPI_Win_create` : création d'une fenêtre, on récupère un objet opaque de type `MPI_Win`.
  - La mémoire doit être déjà allouée !
- `MPI_Win_allocate` : allocation de mémoire et création d'une fenêtre, on récupère un objet opaque de type `MPI_Win` et un pointeur vers le début de la zone mémoire.
  - Alloue la mémoire.
- `MPI_Win_create_dynamic` : création d'une fenêtre sans zone mémoire associée
  - On associe de la mémoire plus tard, avec `MPI_Win_attach` et `MPI_Win_detach`
  - Utile pour ne pas faire l'allocation dans une collective, pour des zones mémoires non-contiguës...

Ces opérations sont **collectives** : tous les processus *du communicateur* doivent les appeler.

- La fenêtre créée ne peut être utilisée que par les processus du communicateur

Destruction de fenêtre

Création de fenêtre

Création de fenêtre

Ces opérations sont **collectives** : tous les processus *du communicateur* doivent les appeler.

- La fenêtre créée ne peut être utilisée que par les processus du communicateur

Destruction de fenêtre

- `MPI_Win_free`
  - Si la création de la fenêtre a alloué de la mémoire, `MPI_Win_free` la libère.

Exemple : création de fenêtre

```
int* token;
MPI_Win win;
MPI_Win_allocate( sizeof( int ), sizeof( int ),
                  MPI_INFO_NULL, MPI_COMM_WORLD, &token, &win );
/* [...] */
MPI_Win_free( &win );
```

Autre façon de faire :

```
int* token;
MPI_Win win;
MPI_Alloc_mem( 1*sizeof(int), MPI_INFO_NULL, &token );
MPI_Win_create( token, sizeof( int ), sizeof( int ),
                MPI_INFO_NULL, MPI_COMM_WORLD, &win );
/* [...] */
MPI_Win_free( &win );
MPI_Free_mem( token );
```

Plan du cours

Déplacements de données

Principe des communications

Découplage des transferts de données et des synchronisations

- Communications **non-bloquantes**
- Accès concurrents possibles, pas d'erreur mais **comportement indéfini**

Fonctions d'accès aux données

On déplace les données dans ou depuis **une fenêtre**

- On donne l'offset depuis le début de la fenêtre
- Début de la fenêtre : `MPI_BOTTOM`
- Calcul de la taille d'un datatype MPI : `MPI_Type_size`

```
MPI_Put( void *origin_addr, int origin_count, MPI_Datatype
         origin_datatype, int target_rank, MPI_Aint target_disp,
         int target_count, MPI_Datatype target_datatype, MPI_Win win);
MPI_Get( void *origin_addr, int origin_count, MPI_Datatype
         origin_datatype, int target_rank, MPI_Aint target_disp,
         int target_count, MPI_Datatype target_datatype, MPI_Win win);
```

Exemple : communications

```
int rank, peer;
int* token;
MPI_Win win;

MPI_Init( &argc, &argv );
MPI_Comm_rank( MPI_COMM_WORLD, &rank );

MPI_Win_allocate( sizeof( int ), sizeof( int ), MPI_INFO_NULL,
                  MPI_COMM_WORLD, &token, &win );

*token = getpid();
if( 0 == rank ) {
  peer = 1;
  MPI_Get( token, 1, MPI_INT, peer, 0, 1, MPI_INT, win );
}

MPI_Win_free( &win );
MPI_Finalize();
```

→ Quel est le problème ici ?

Accumulate

`MPI_Accumulate` : accumule des données dans un buffer distant et **effectue une opération**

- Type de l'opération : `MPI_Op` : possibilité d'en définir
- Mêmes opérations de base de `MPI_REDUCE`
- Possibilité d'utiliser `MPI_NO_OP`

Généralisation de "`fetch-and-add`"

- Applique une opération définie
- Remplace la valeur dans le buffer cible par le résultat
- **Atomique** !
- Modes d'ordonnancement dans les buffers (par défaut : pas de surprise)

```
int MPI_Accumulate( void *origin_addr, int origin_count,
                    MPI_Datatype origin_datatype, int target_rank,
                    MPI_Aint target_disp, int target_count,
                    MPI_Datatype target_datatype, MPI_Op op,
                    MPI_Win win );
```

Autres : `MPI_Get_accumulate`, `MPI_Fetch_and_op`, `MPI_Compare_and_swap`

Exemple : accumulate

```
MPI_Win win;
int rank, size, *a, *b, i;
MPI_Init( &argc, &argv );
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
MPI_Comm_size( MPI_COMM_WORLD, &size );
MPI_Alloc_mem( sizeof(int)*size, MPI_INFO_NULL, &b );
MPI_Win_create( b, size, sizeof(int), MPI_INFO_NULL,
                MPI_COMM_WORLD, &win );
for( i = 0; i < size; i++ ) a[i] = b[i] = rank * 100 + i;
MPI_Win_fence( MPI_MODE_NOPRECEDE, win );
for (i = 0; i < size; i++)
    MPI_Accumulate( &a[i], 1, MPI_INT, i, rank, 1, MPI_INT,
                    MPI_SUM, win );
MPI_Win_fence( (MPI_MODE_NOSTORE | MPI_MODE_NOSUCCEED), win );
MPI_Win_free(&win);
MPI_Free_mem(a);
MPI_Free_mem(b);
MPI_Finalize();
```

- On crée une fenêtre `win` à laquelle on associe le buffer `b`
- On accumule le contenu de `b` dans `a` à un offset `rank` en lui appliquant l'opération `MPI_SUM`.

Plan du cours

Synchronisations

Comment s'assurer de la fin d'une opération ?

- Mode actif sur la cible : le processus cible est impliqué dans la synchronisation (par exemple : PSCW)
- Mode passif : la synchronisation est uniquement au niveau du processus local (par exemple : lock)

Fonctions de transfert avec Request

`MPI_Rput`, `MPI_Rget` : similaires à `MPI_Put` et `MPI_Get` mais on **récupère une** `MPI_Request`

- On peut attendre sur cette `MPI_Request`

Attente de complétion

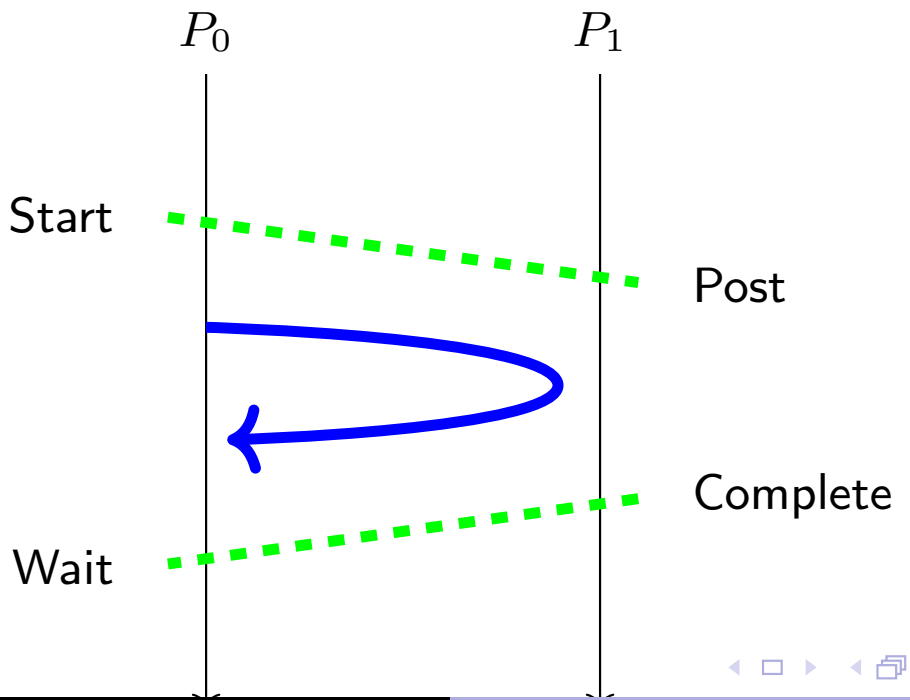
- `MPI_Win_fence` : synchronise toutes les communications RMA sur une fenêtre
- `MPI_Win_flush` : termine les opérations RMA sur le processus choisi
- `MPI_Win_flush_local` : termine les opérations RMA *locales* sur le processus choisi

Synchronisation PSCW

PSCW

- `MPI_Win_post` : démarre l'exposition d'une *epoch* à un groupe (non-bloquant)
- `MPI_Win_start` : démarre l'accès d'une *epoch* à un groupe
- `MPI_Win_complete` : termine l'accès à l'*epoch* qui précède
- `MPI_Win_wait` : attente de complétion

À noter aussi `MPI_Win_test` : test de complétion



Exemple de synchronisation PSCW

```
MPI_Win win;
MPI_Group group;
int i, tmp, size, *rank;
MPI_Comm_rank( MPI_COMM_WORLD, &tmp );
MPI_Comm_size( MPI_COMM_WORLD, &size );
rank = (int *) malloc ( sizeof(int) * size );
MPI_Win_create( rank, size*sizeof(int), sizeof(int), MPI_INFO_NULL,
                MPI_COMM_WORLD, &win );
MPI_Win_group( win, &group );

MPI_Win_post( group, 0, win );
MPI_Win_start( group, 0, win );

for ( i=0; i < size; i++ )
  MPI_Put( &tmp, 1, MPI_INT, i, tmp, 1, MPI_INT, win );

MPI_Win_wait( win );
MPI_Win_complete( win );
```

Synchronisations : les epoch

Notion d'**epoch**

Les communications initiés dans une **epoch** sont **terminées à la fin de l'epoch** .

- Début d'une **epoch** : `MPI_Win_lock`
- Fin d'une **epoch** : `MPI_Win_unlock`

Permet d'assurer qu'il n'y a pas deux opérations RMA d'un processus vers un autre en même temps.

Attention

- Ce n'est pas un verrou au sens pthread.
- Ça n'assure rien quant aux communications RMA dans l'autre sens.
- Possibilité de bloquer également un accès exclusif aux données : `MPI_LOCK_EXCLUSIVE`
- Sinon : `MPI_LOCK_SHARED`

Exemple de synchronisation lock/unlock

```
int rank, data, peer;
int* token;
MPI_Win win;

MPI_Init( &argc, &argv );
MPI_Comm_rank( MPI_COMM_WORLD, &rank );

MPI_Win_allocate( sizeof( int ), sizeof( int ), MPI_INFO_NULL,
                  MPI_COMM_WORLD, &token, &win );

*token = getpid();
if( 0 == rank ) {
  peer = 1;
  MPI_Win_lock( MPI_LOCK_SHARED, peer, 0, win );
  MPI_Get( token, 1, MPI_INT, peer, 0, 1, MPI_INT, win );
  MPI_Win_unlock( peer, win );
}

MPI_Win_free( &win );
MPI_Finalize();
```

Petites remarques

Performances : put plutôt que get

- Put = un aller simple
- Get = aller-retour, attente du retour

Attention aux datatypes

- Pas forcément le même sur la source et sur la cible
- Permet par exemple de changer l'espacement en réception

Accumulate permet d'émuler Put et Get

- Avec `MPI_NO_OP`
- Performances : très lent

Pas de garantie sur le moment exact où le mouvement de données est fait

- Pas forcément pendant le `MPI_Put` ou le `MPI_Get`
- Ni pendant le `MPI_Win_fence`

Attention aux accès concurrents !

- Si plusieurs processus font un accès distant à la même zone mémoire ?
- Aucune garantie.

Plan du cours