Systèmes distribués

– M1 informatique –

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

Institut Galilée, Université Paris XIII

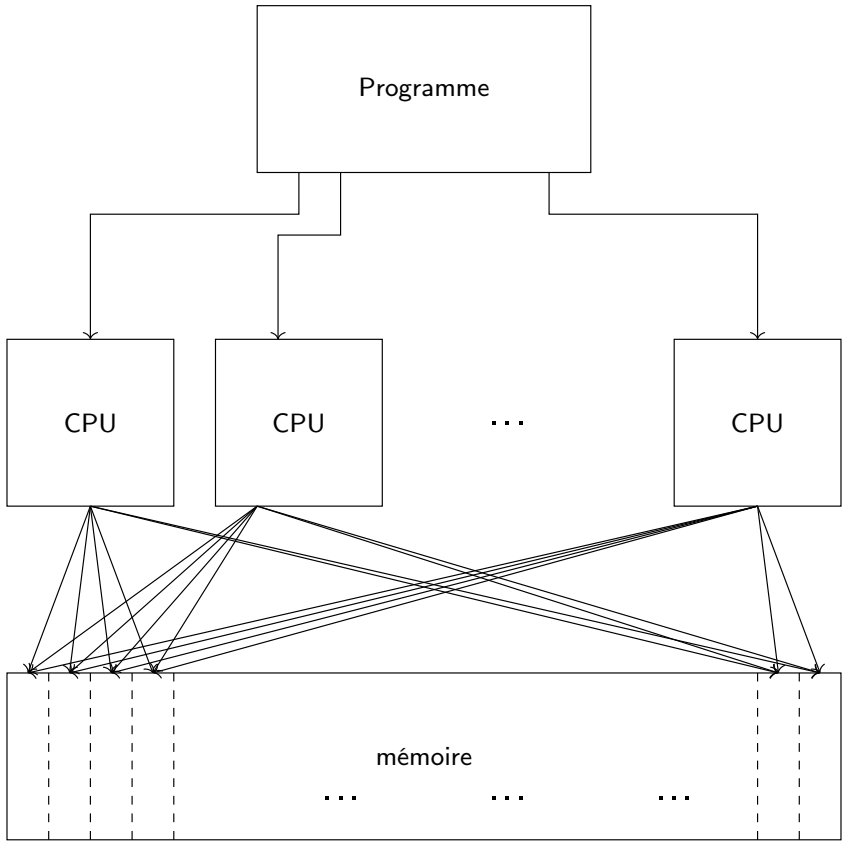
Plan du cours

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*

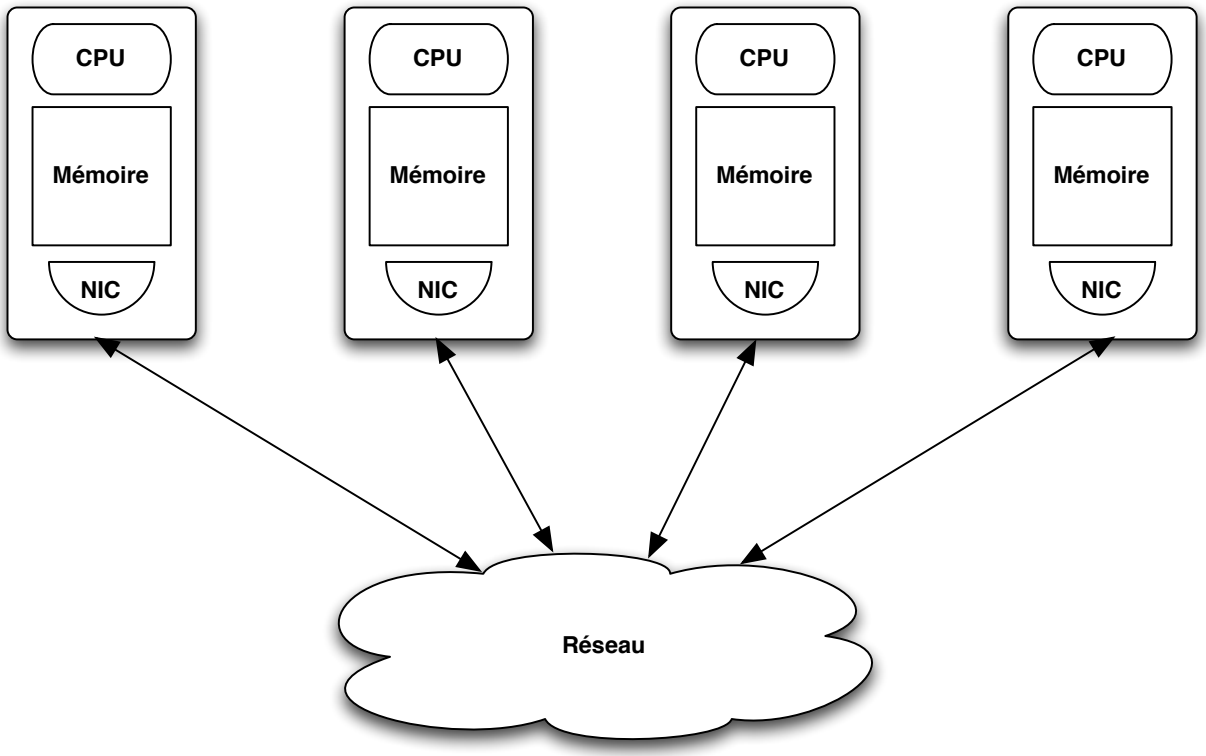
GPU

- 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

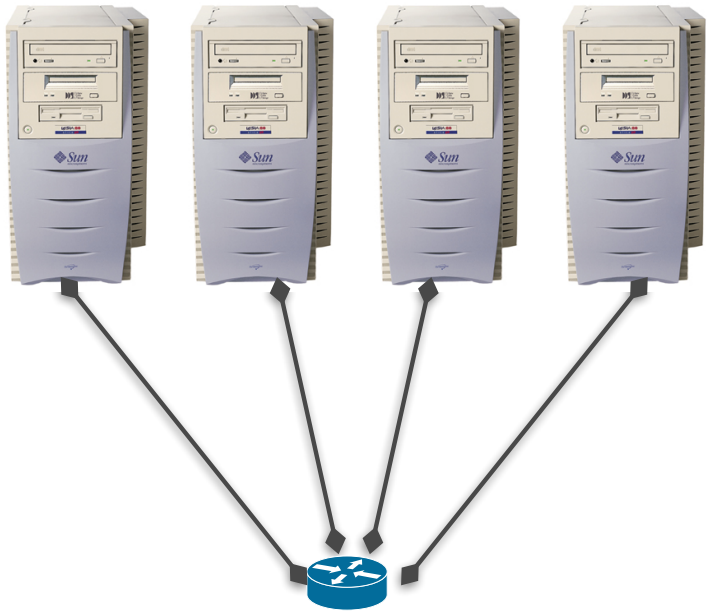
SISD	SIMD
MISD	MIMD

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

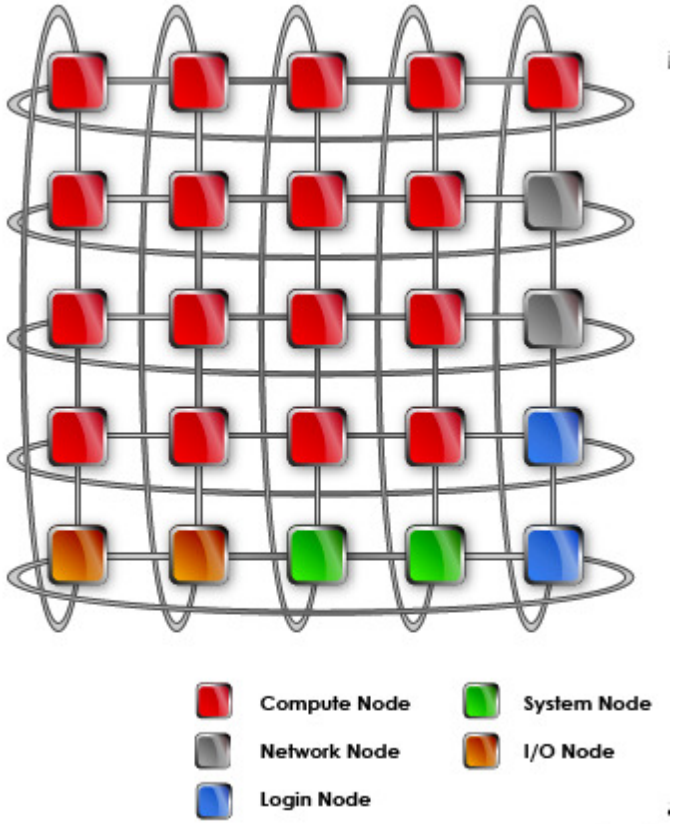
- CPU : deux AMD Istanbul
  - 6 cœurs chacun
  - 2 puces par machine
  - Empilées sur la même socket
  - Bus : crossbar
- Pas de disque dur

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

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...**

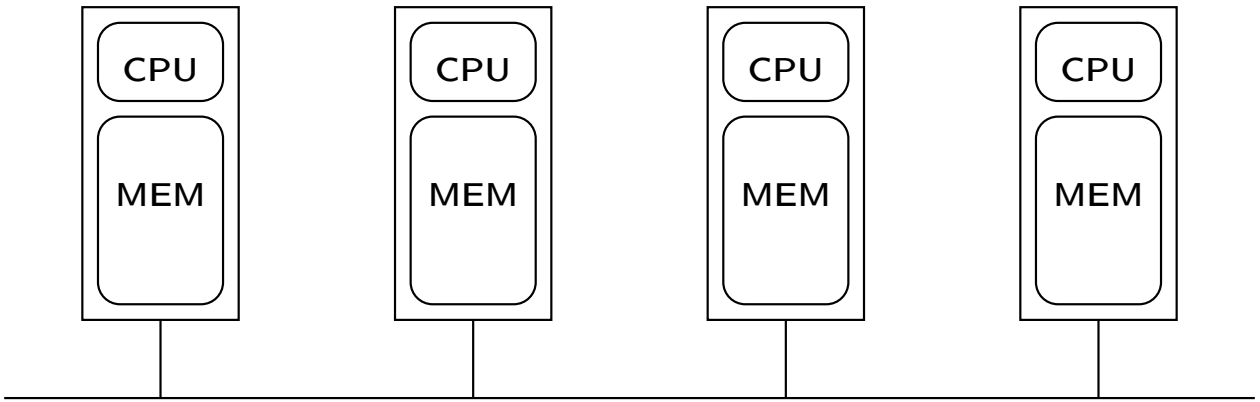
Programmation sur mémoire distribuée

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

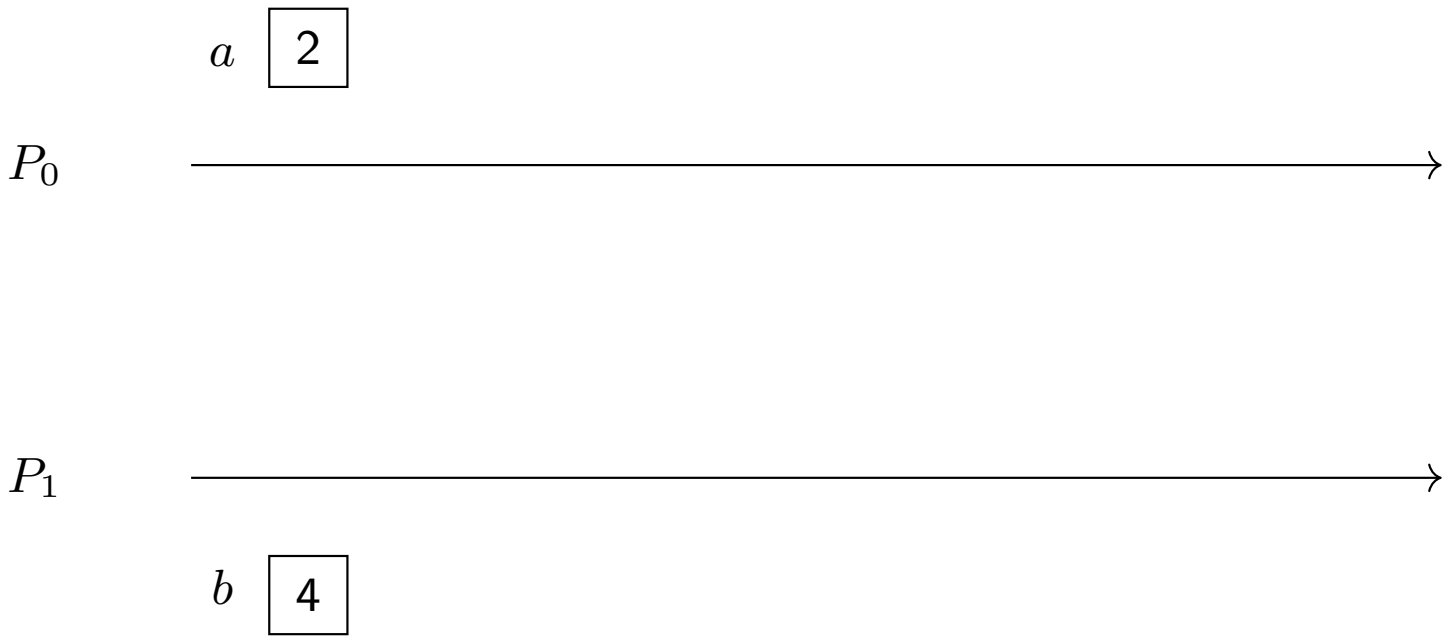


Communications bilatérales

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

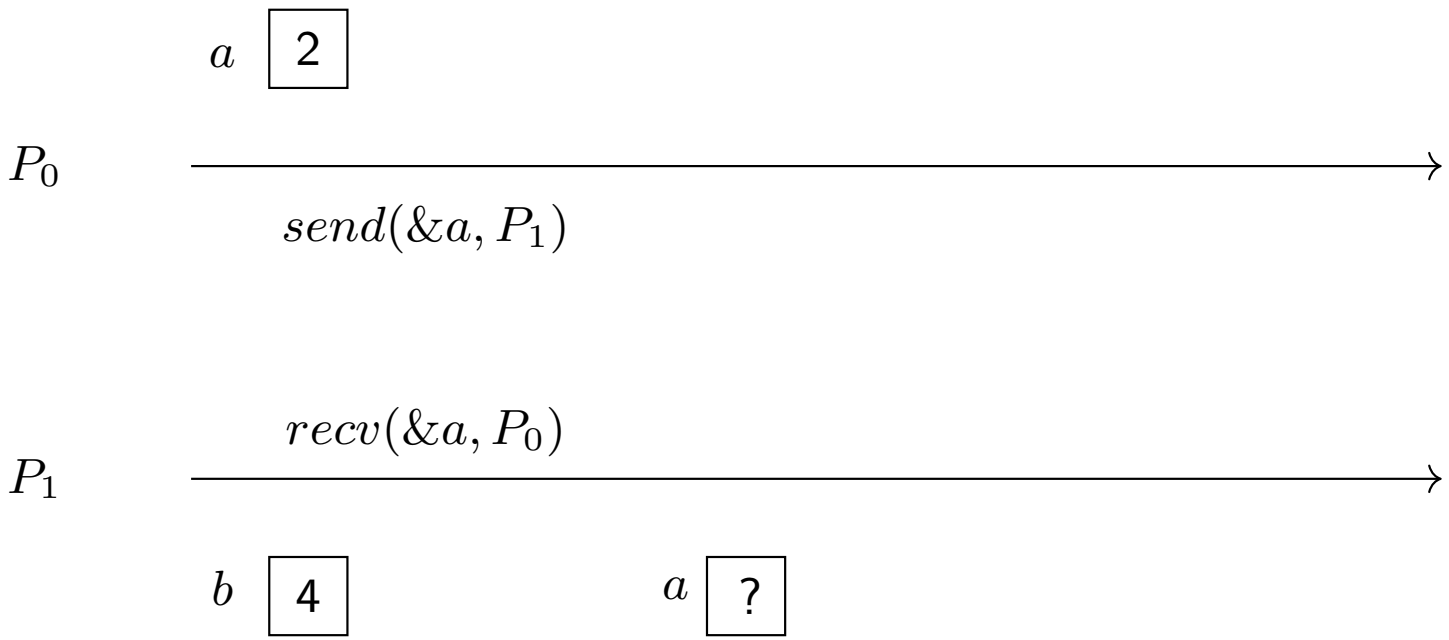


Communications bilatérales

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

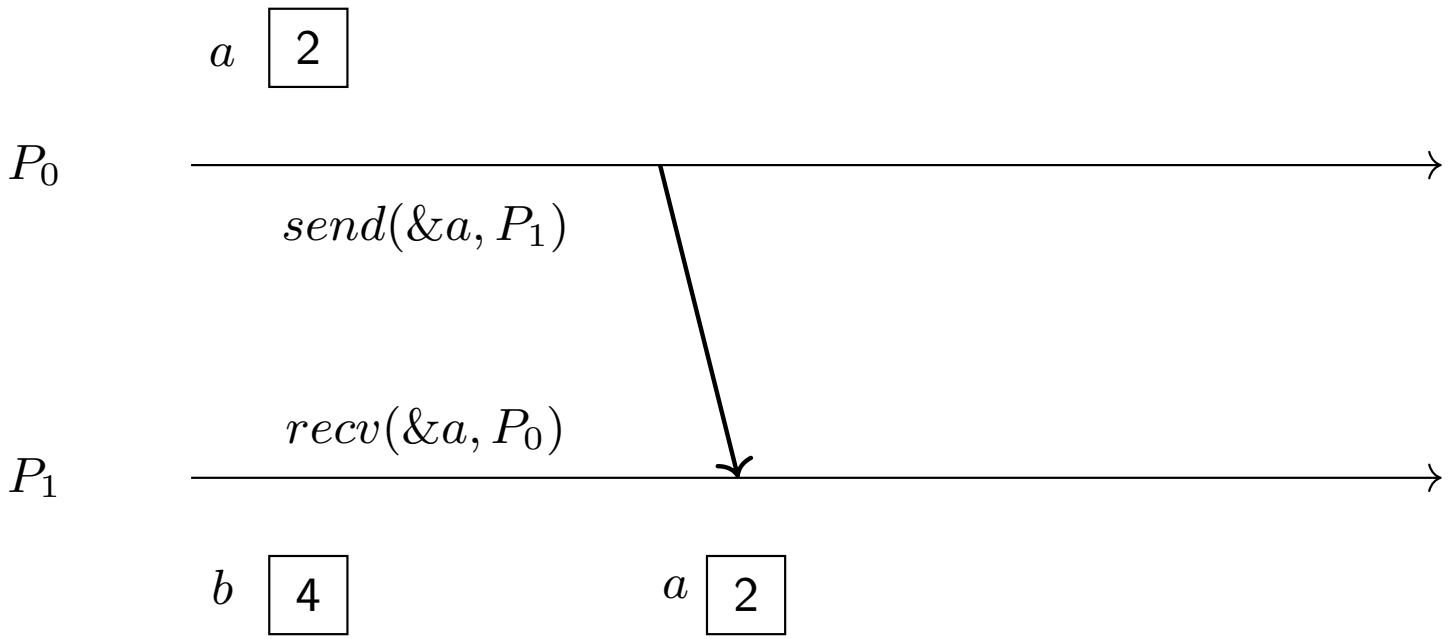


Communications bilatérales

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

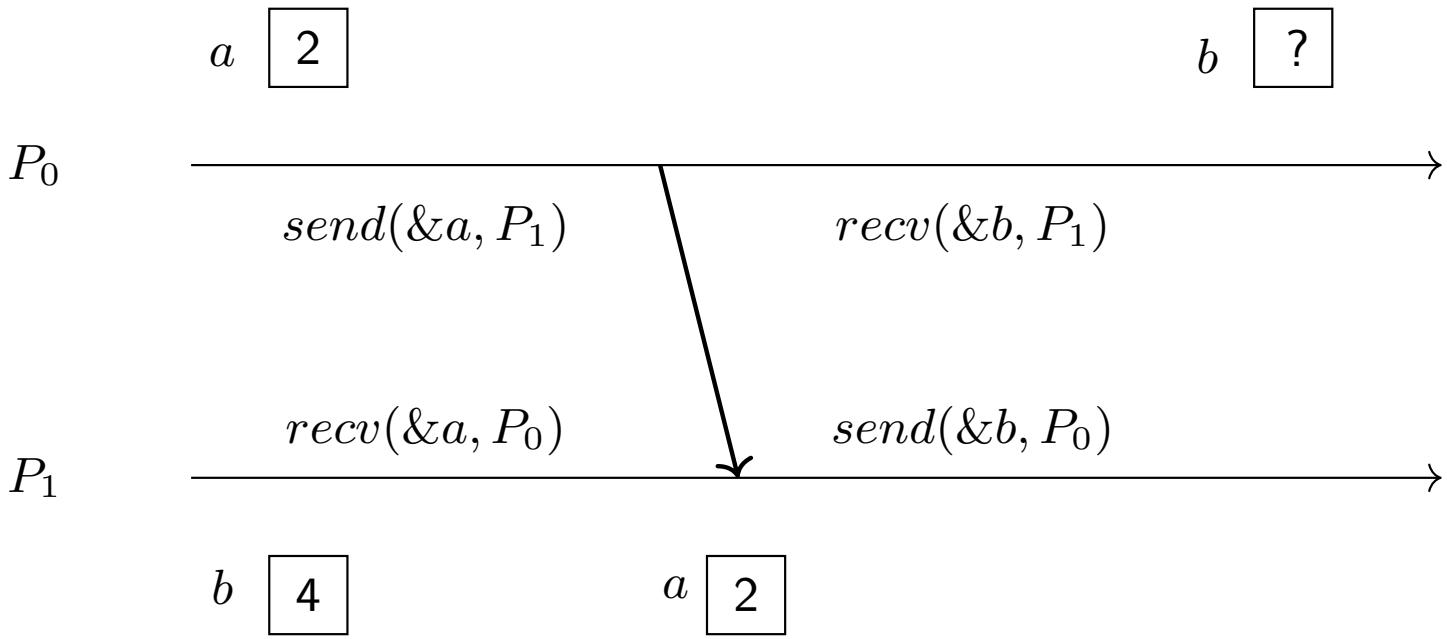


Communications bilatérales

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

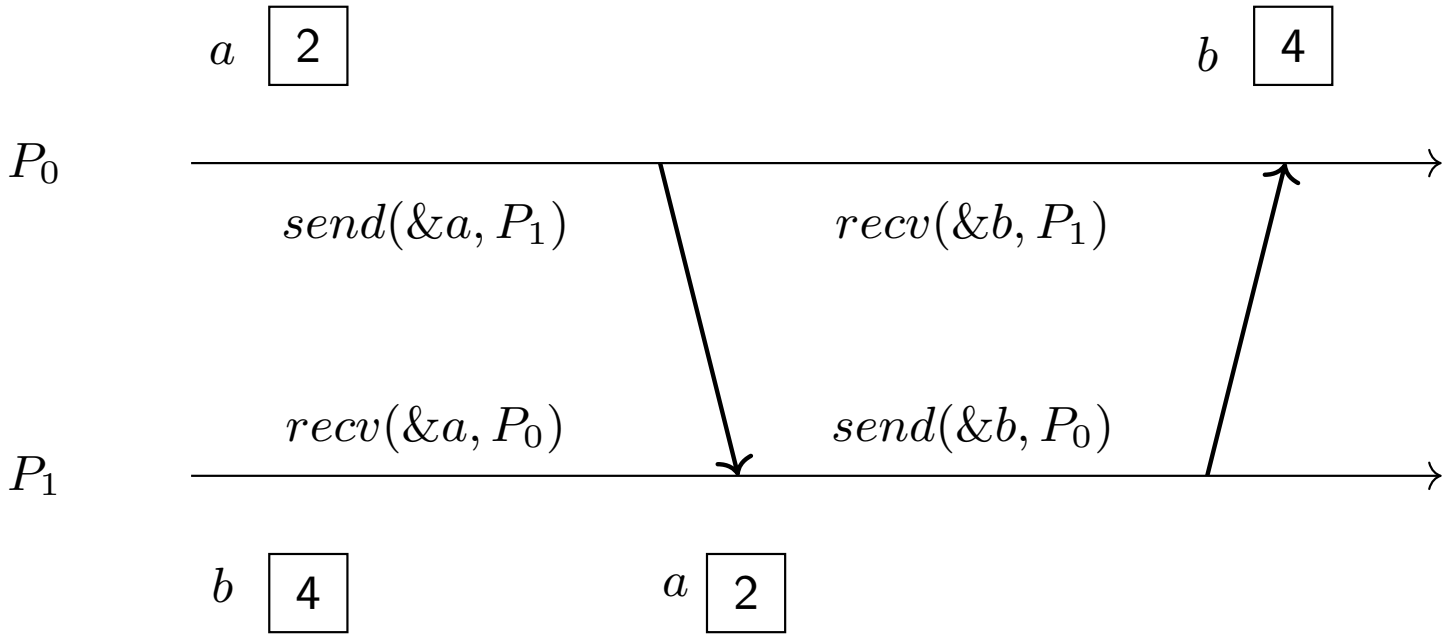


Communications bilatérales

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 :

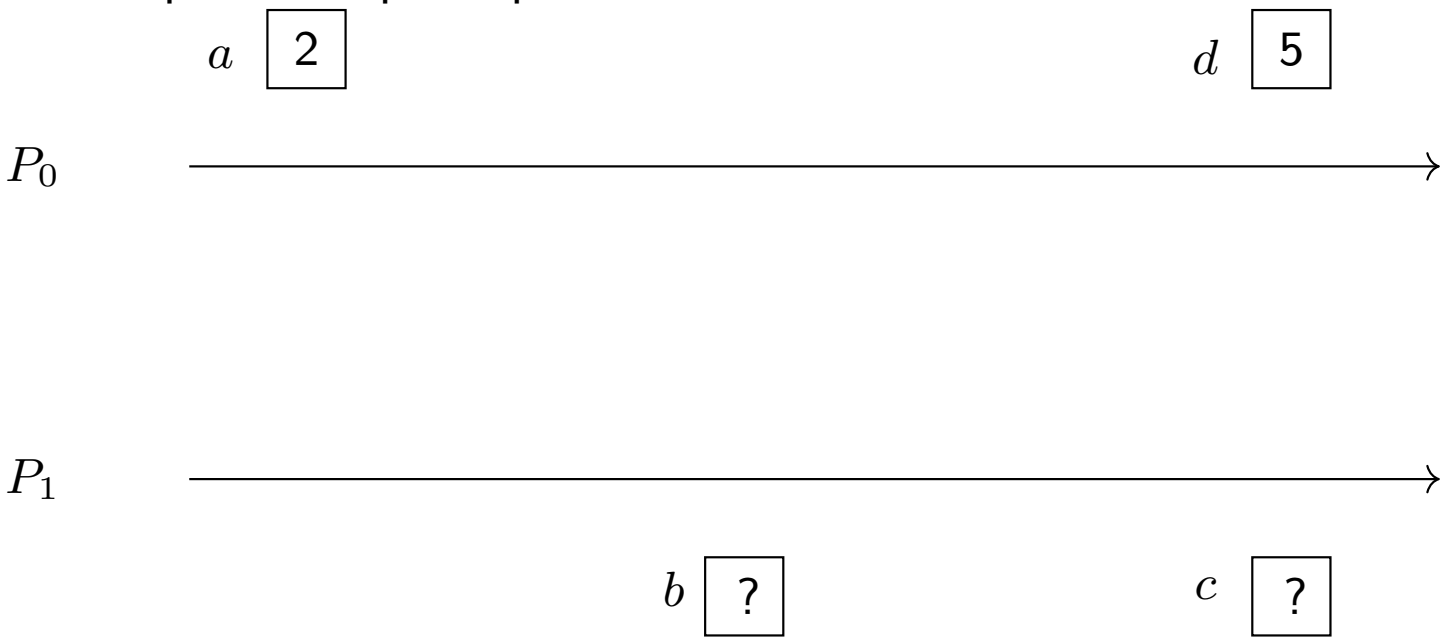
- Besoin de la coopération des deux processus : source et destination
- Fort synchronisme

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.

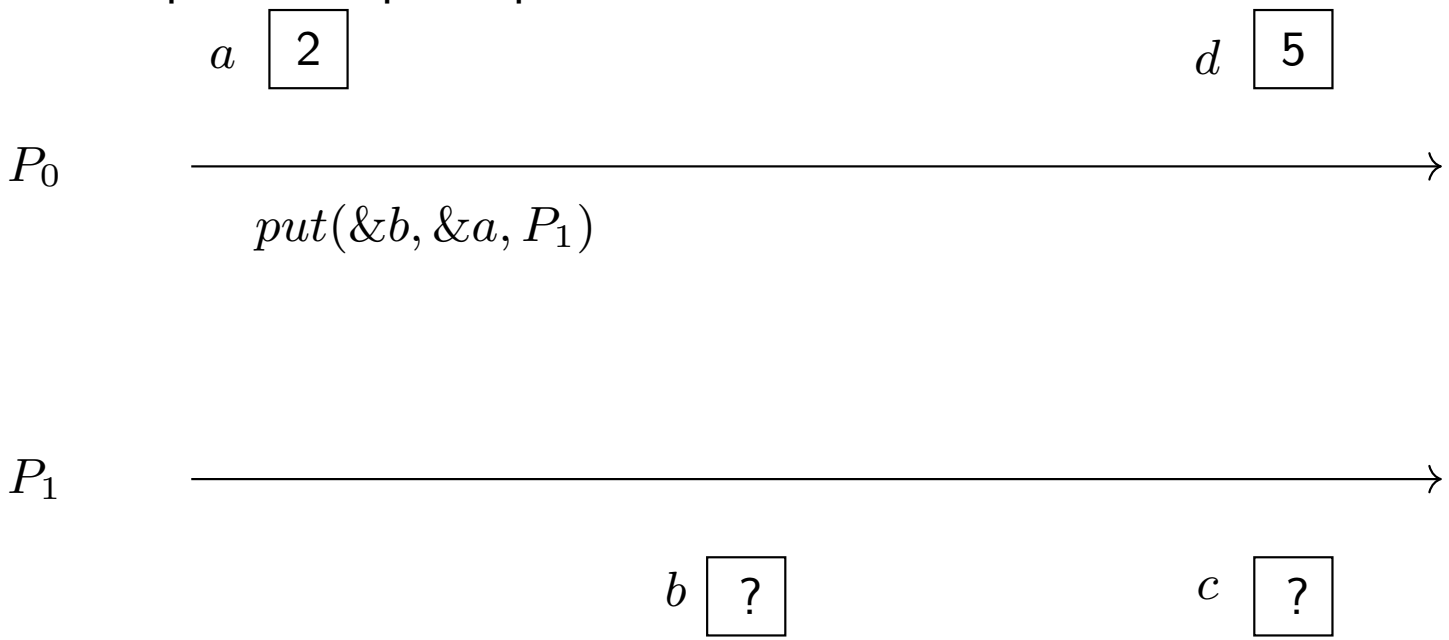


Communications unilatérales

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.

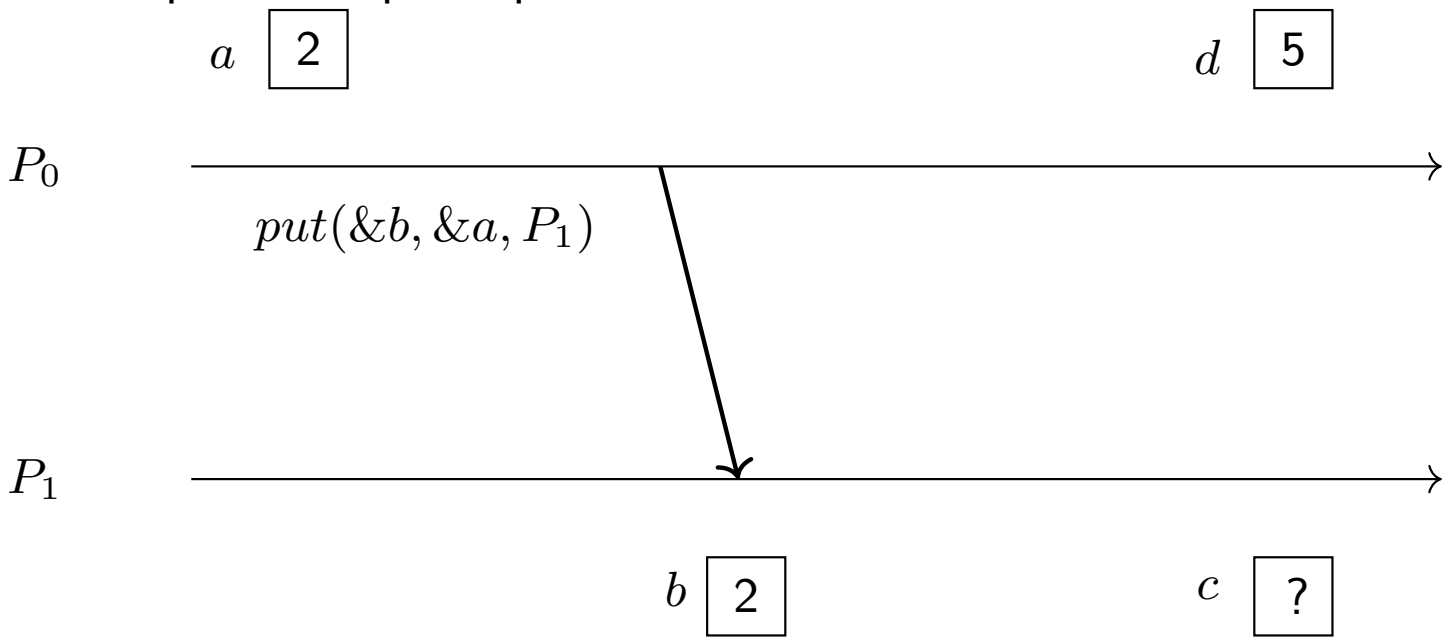


Communications unilatérales

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.

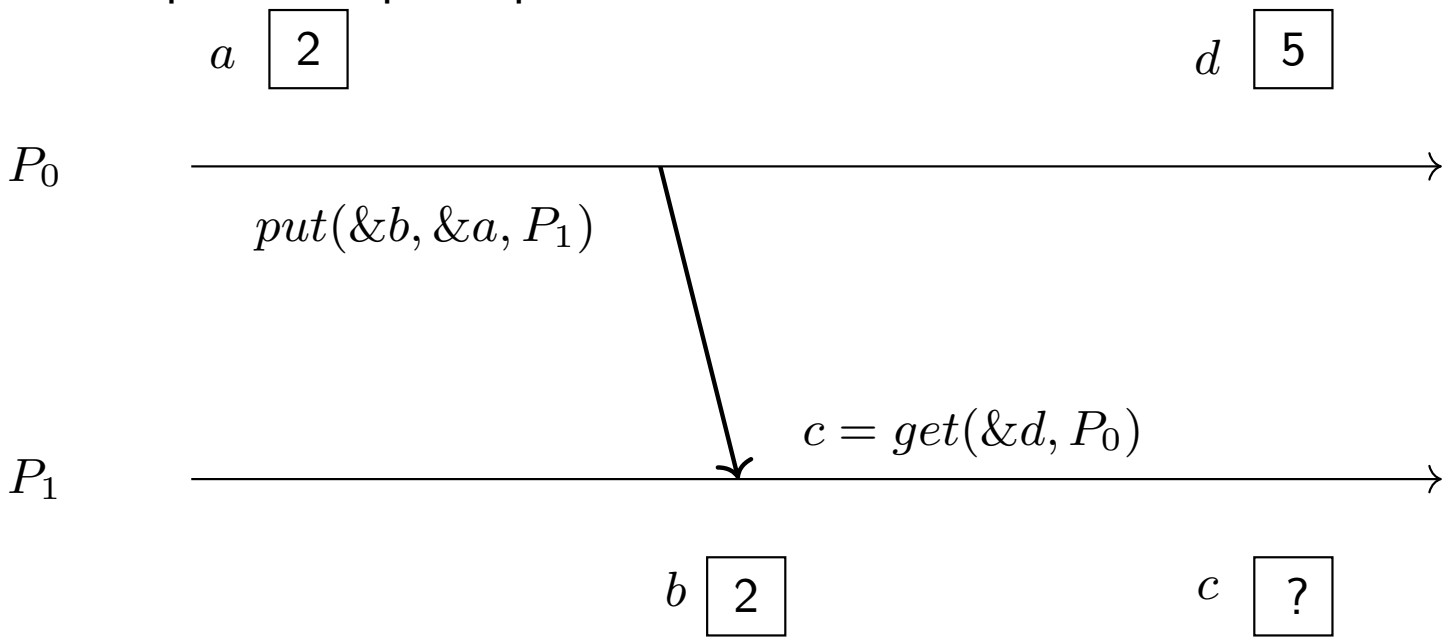


Communications unilatérales

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.

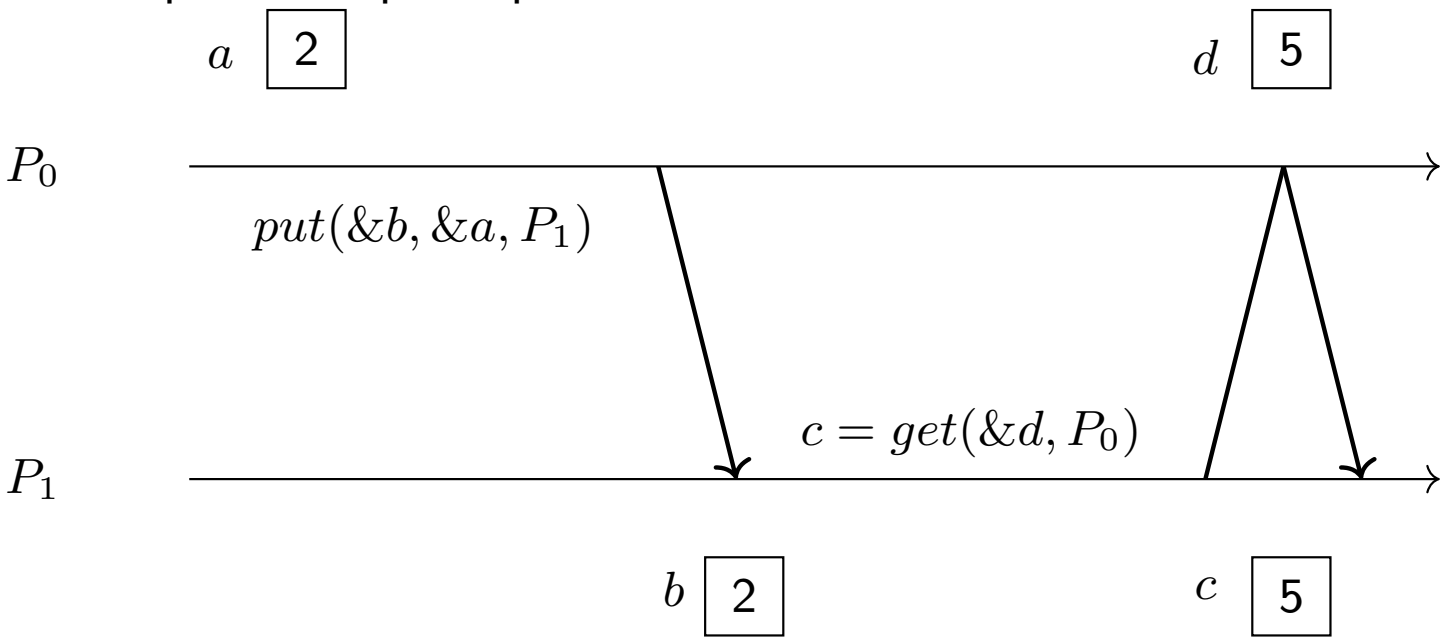


Communications unilatérales

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

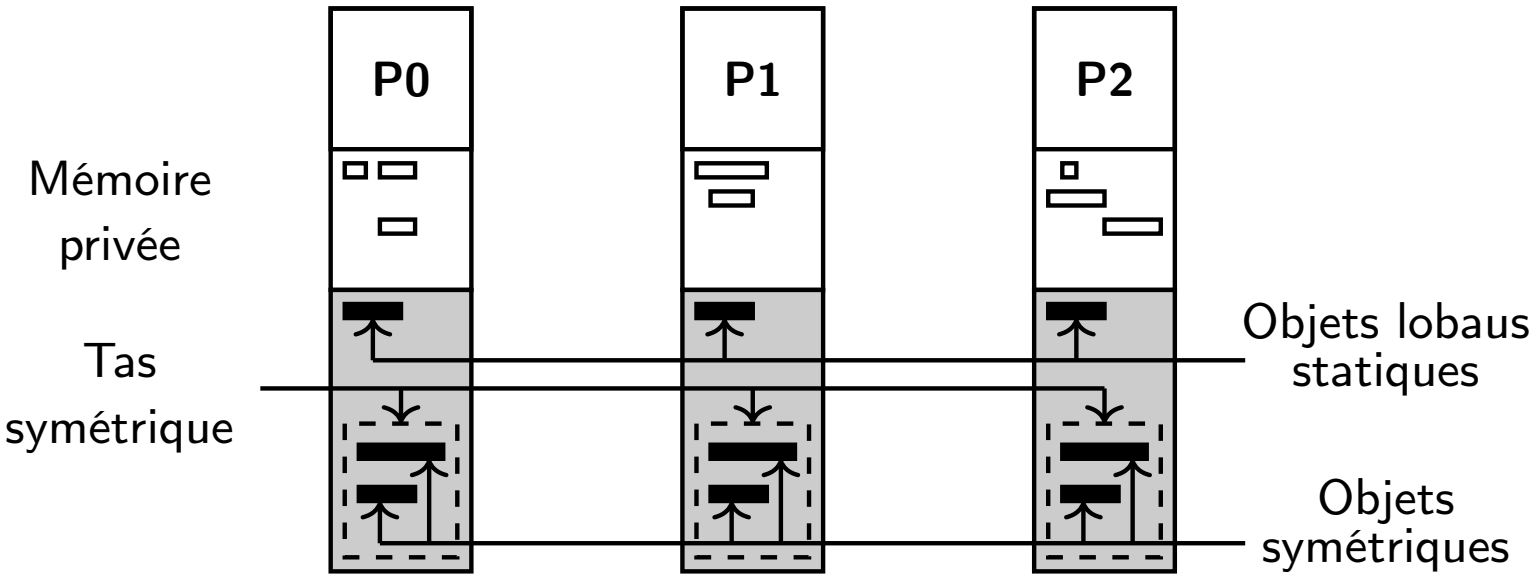
Inconvénients :

- 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 :

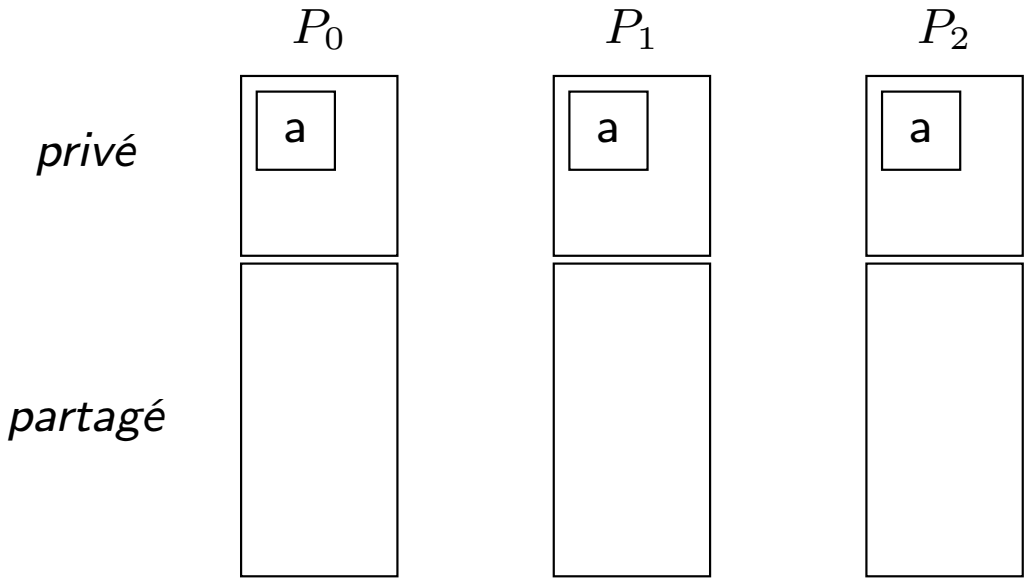
- Le programmeur déclare la **visibilité** de ses variables : privées (par défaut) ou **partagées**
- Pour les tableaux : le programmeur déclare la taille des blocs sur chaque processus
- Le compilateur se charge de
  - **répartir les données partagées** dans la mémoire des différents processus
  - **traduire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



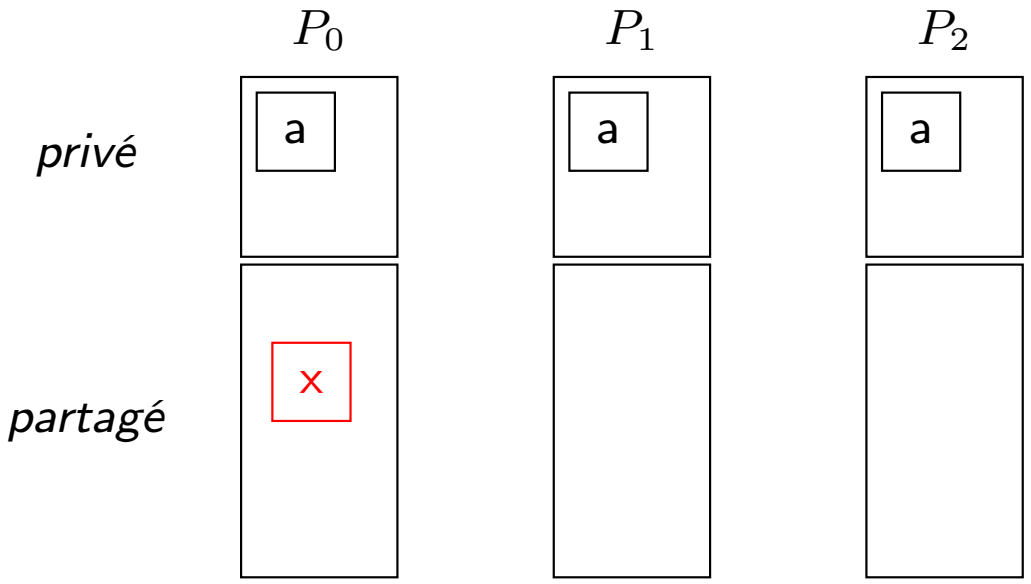
int a ;

shared int x ;

Exemples

Langages **PGAS** :

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



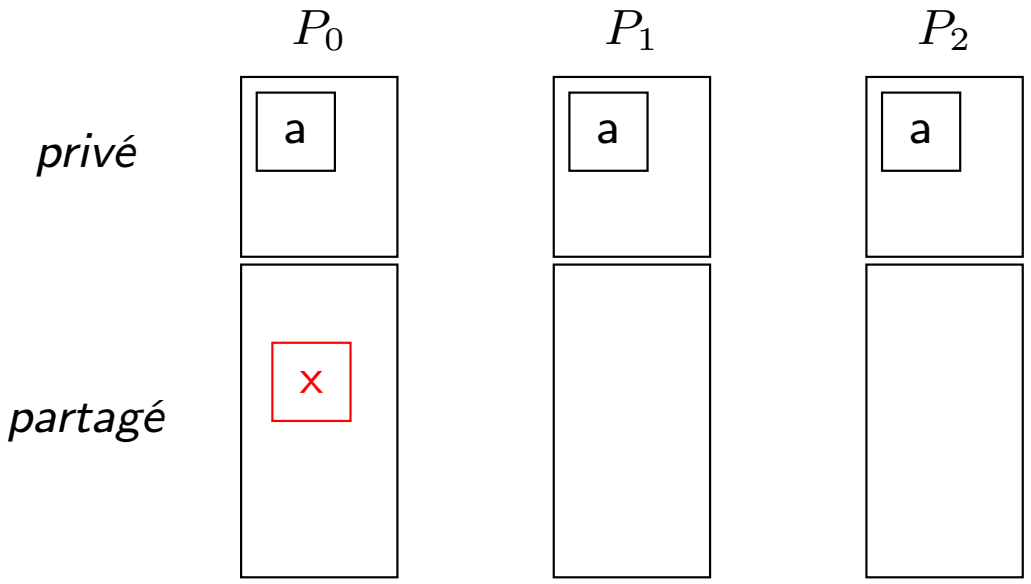
int a ;

shared int x ;

Exemples

Langages **PGAS** :

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

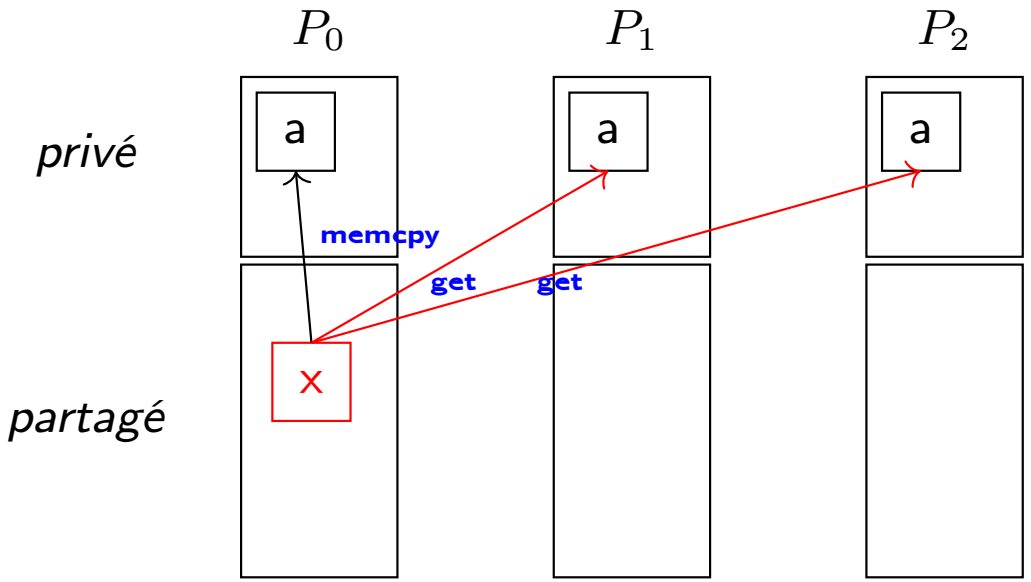



```
int a ;
shared int x ;
int a = x ;
```

Exemples

Langages **PGAS** :

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




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

Sac de tâches

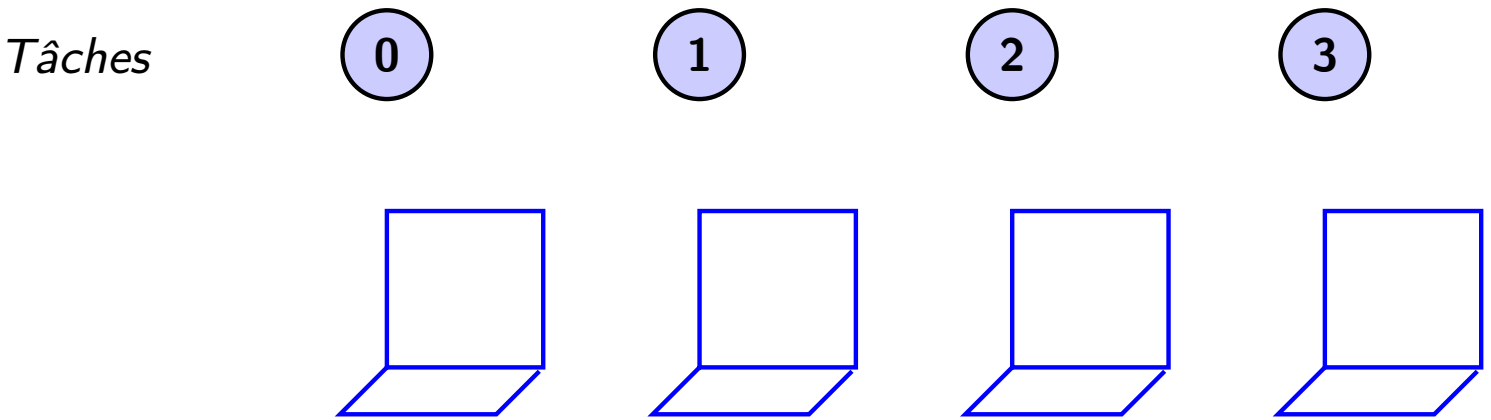
Qu'est-ce qu'un **sac de tâches** ?

- 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** ?

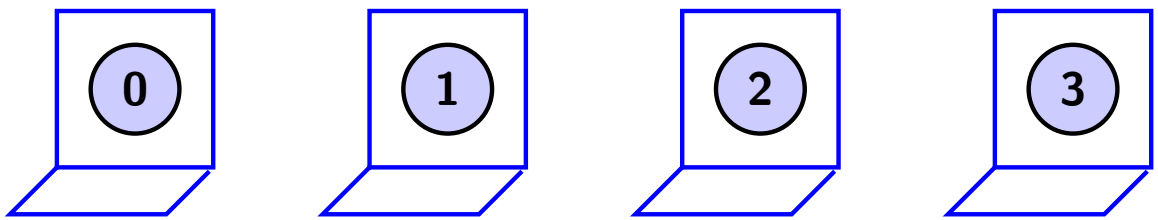
- Un ensemble de calculs à faire
- **Indépendants** les uns des autres

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

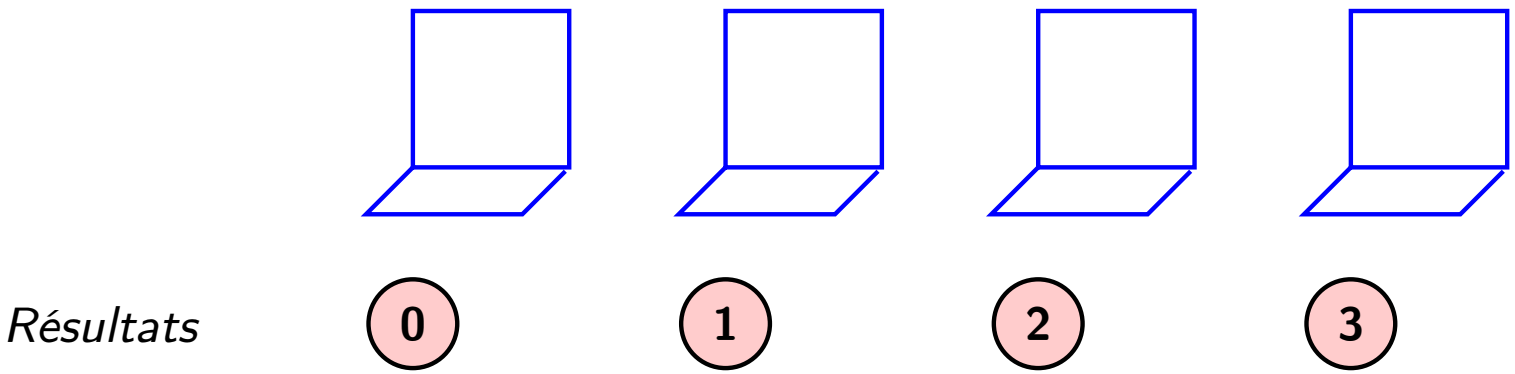
Qu'est-ce qu'un **sac de tâches** ?

- 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 !

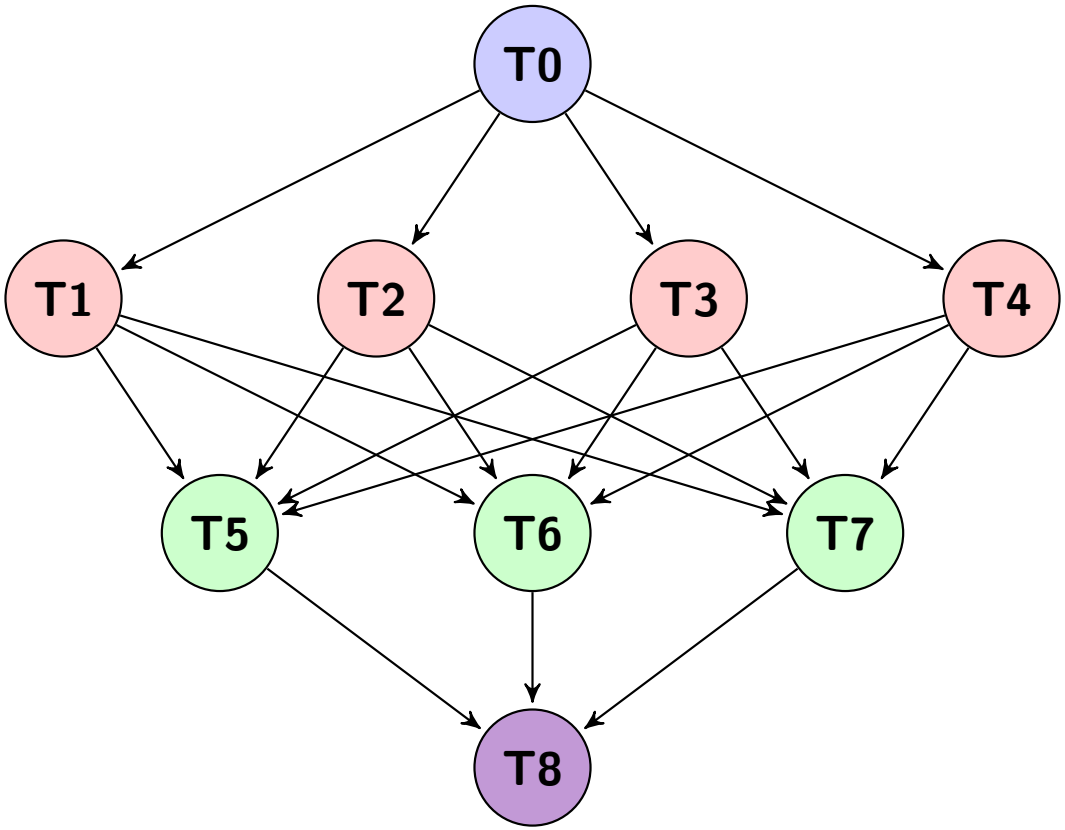
Pas de communications entre les processus exécutant les tâches.



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

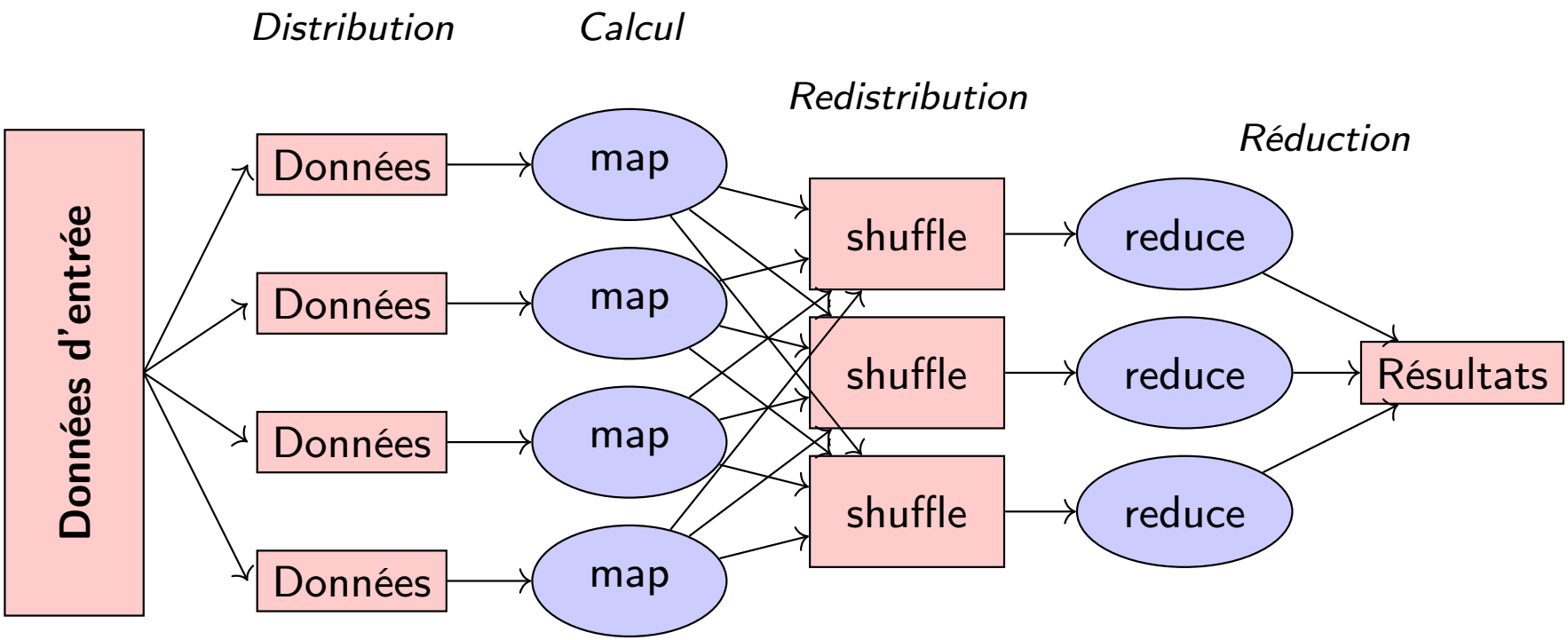
- Besoin d'un coordinateur qui distribue les tâches
- ... et qui récupère les résultats à la fin.

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

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

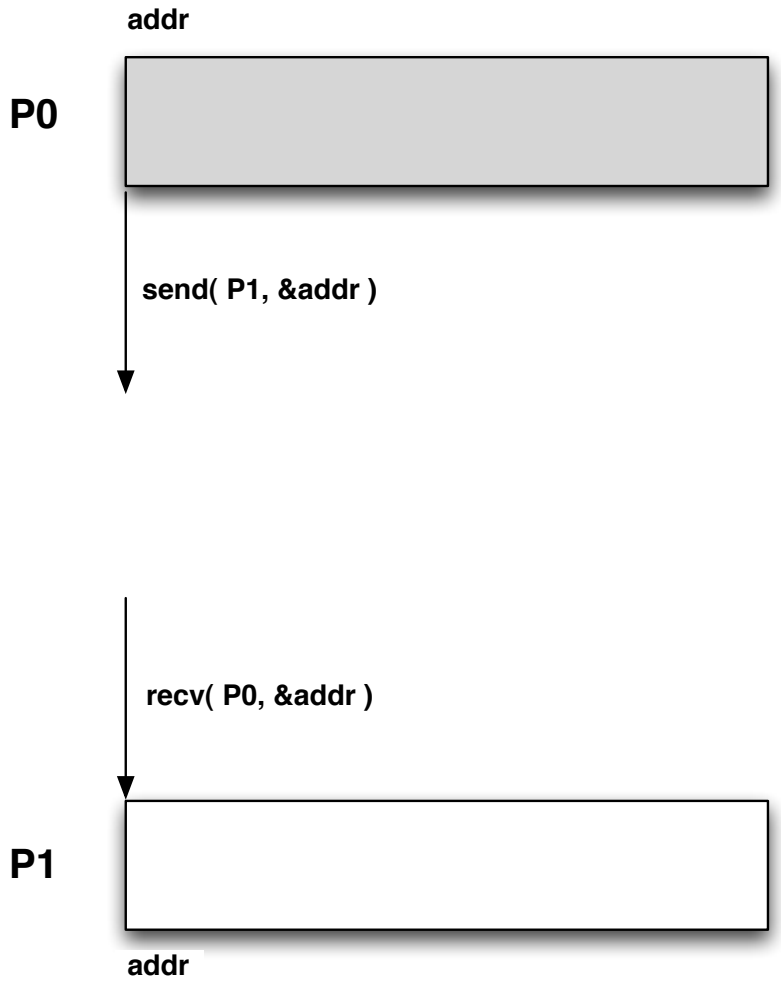
Chaque processus (émetteur et récepteur) a un tampon (buffer) pour le message

- 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...



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...)

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

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()
```

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

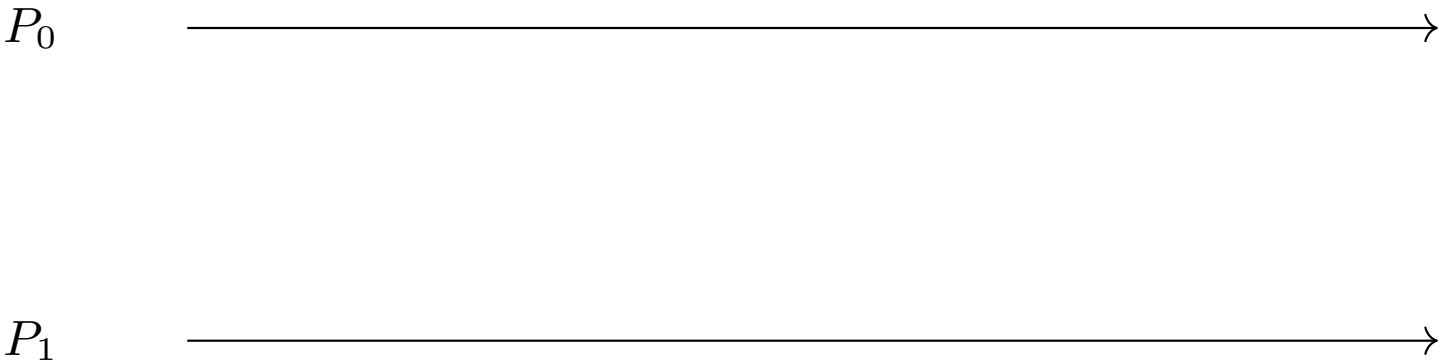
Sources d'erreurs fréquentes

- 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.

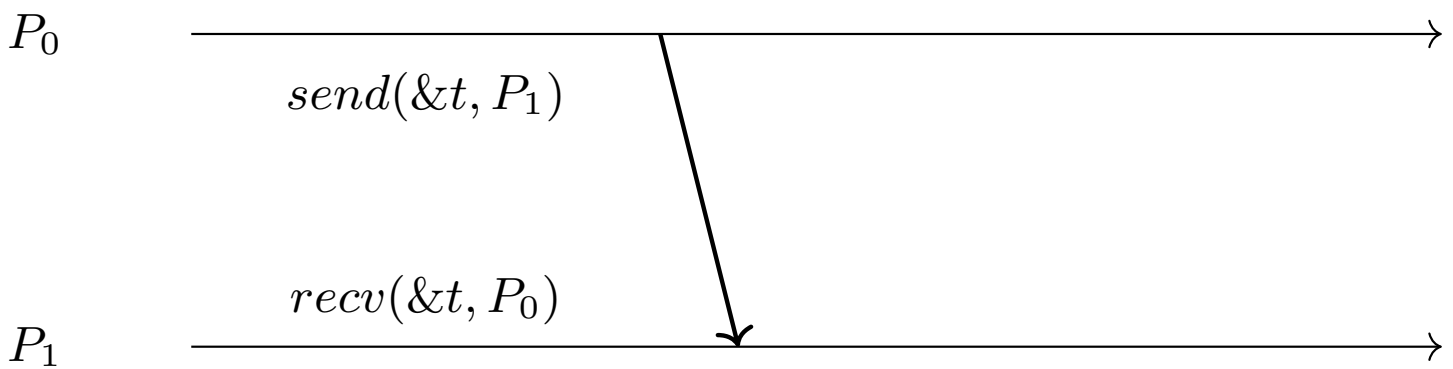
```
if( 0 == rank ) {
    MPI_Send( &token, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    MPI_Recv( &token, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &status );
} else if( 1 == rank ) {
    MPI_Recv( &token, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status );
    MPI_Send( &token, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
}
```



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.

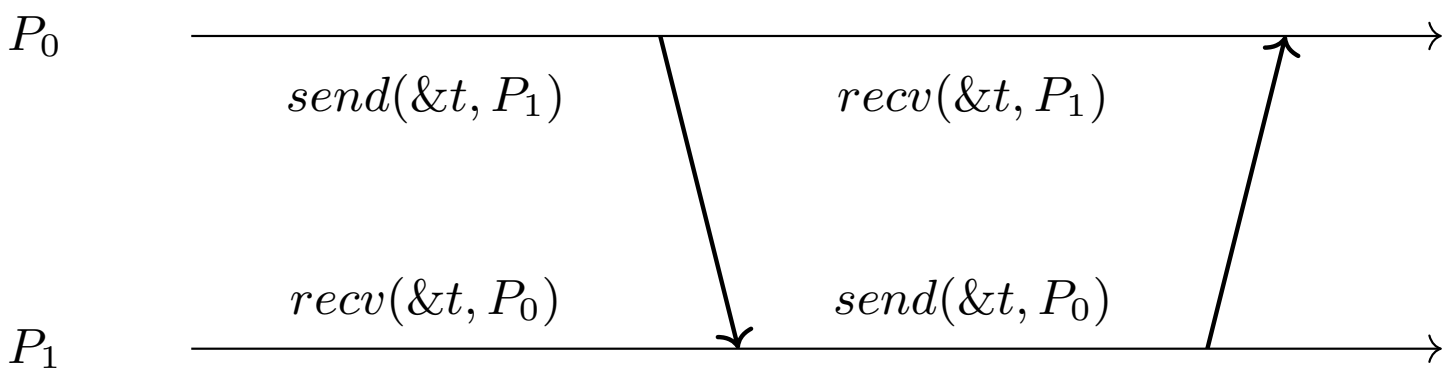
```
if( 0 == rank ) {
    MPI_Send( &token, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    MPI_Recv( &token, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &status );
} else if( 1 == rank ) {
    MPI_Recv( &token, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status );
    MPI_Send( &token, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
}
```



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)

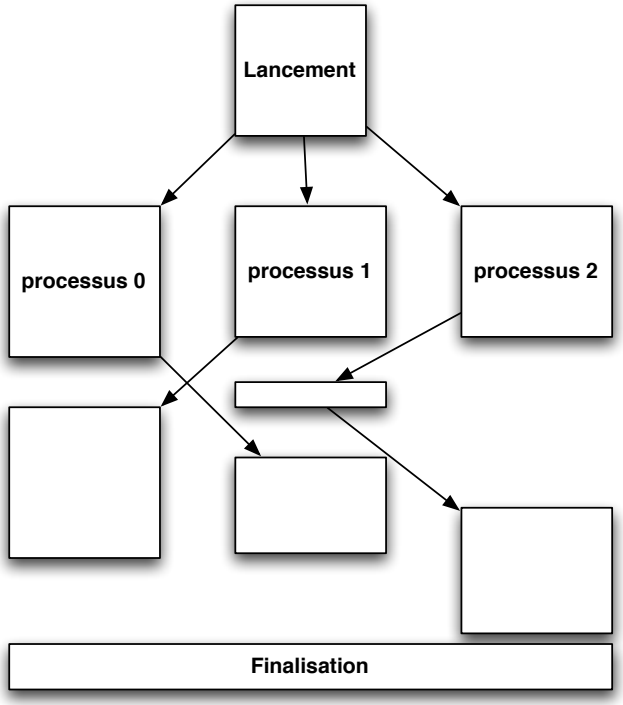
Plan du cours

Mesure de la performance des programmes parallèles

Comment définir cette performance

Pourquoi parallélise-t-on ?

- Pour diviser un calcul qui serait trop long / trop gros sinon
- Diviser le problème $\leftrightarrow$ diviser le temps de calcul ?



Sources de ralentissement

- Synchronisations entre processus
  - Mouvements de données
  - Attentes
  - Synchronisations
- Adaptations algorithmiques
  - L'algorithme parallèle peut être différent de l'algorithme séquentiel
  - Calculs supplémentaires

Efficacité du parallélisme ?

Accélération

Définition

L'accélération d'un programme parallèle (ou *speedup*) représente le gain en rapidité d'exécution obtenu par son exécution sur plusieurs processeurs.

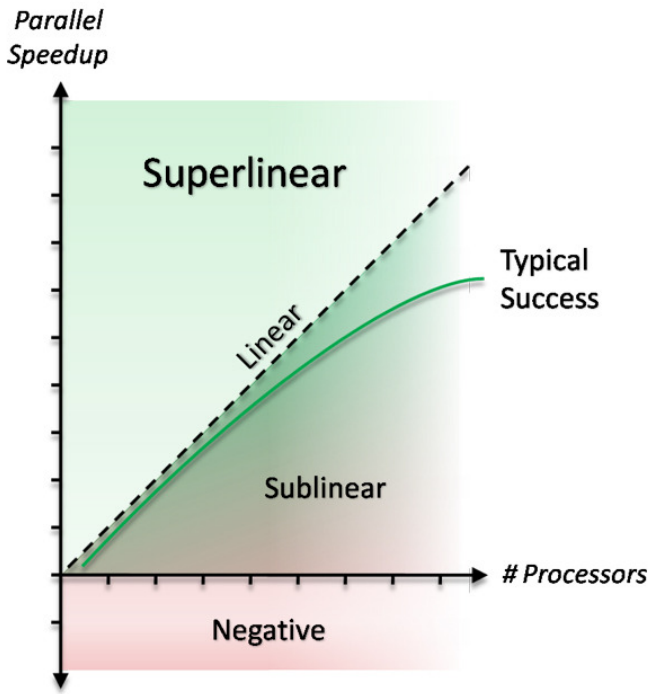
Mesure de l'accélération

On la mesure par le rapport entre le temps d'exécution du programme séquentiel et le temps d'exécution sur p processeurs

$$S_p = \frac{T_{seq}}{T_p}$$

Appréciation de l'accélération

- Accélération linéaire : parallélisme optimal
- Accélération sur-linéaire : attention
- Accélération sub-linéaire : ralentissement dû au parallélisme



Loi d'Amdahl

Décomposition d'un programme parallèle

Décomposition du temps d'exécution d'une application parallèle

- Une partie purement séquentielle ;
- Une partie parallélisable

Énoncé

On note s la proportion parallélisable de l'exécution et p le nombre de processus. Le rendement est donné par la formule :

$$R = \frac{1}{(1-s) + \frac{s}{p}}$$

Remarques

- si $p \to \infty$: $R = \frac{1}{(1-s)}$
  - L'accélération est *toujours* limitée par la partie non-parallélisable du programme
- Si $(1-s) \to 0$, on a $R \sim p$: l'accélération est linéaire

Passage à l'échelle (scalabilité)

Remarque préliminaire

On a vu avec la loi d'Amdahl que la performance augmente *théoriquement* lorsque l'on ajoute des processus.

- Comment augmente-t-elle en réalité ?
- Y a-t-il des facteurs limitants (goulet d'étranglement...)
- Augmente-t-elle à l'infini ?

Définition

Le *passage à l'échelle* d'un programme parallèle désigne l'augmentation des performances obtenues lorsque l'on ajoute des processus.

Obstacles à la scalabilité

- Synchronisations
- Algorithmes ne passant pas à l'échelle (complexité de l'algo)
  - Complexité en opérations
  - Complexité en communications

Passage à l'échelle (scalabilité)

La performance d'un programme parallèle a plusieurs dimensions

Scalabilité forte

On fixe la taille du problème et on augmente le nombre de processus

- Relative au speedup
- Si on a une hyperbole : scalabilité forte parfaite
  - On augmente le nombre de processus pour calculer plus vite

Scalabilité faible

On augmente la taille du problème avec le nombre de processus

- Le problème est à taille constante *par processus*
- Si le temps de calcul est constant : scalabilité faible parfaite
  - On augmente le nombre de processus pour résoudre des problèmes de plus grande taille

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

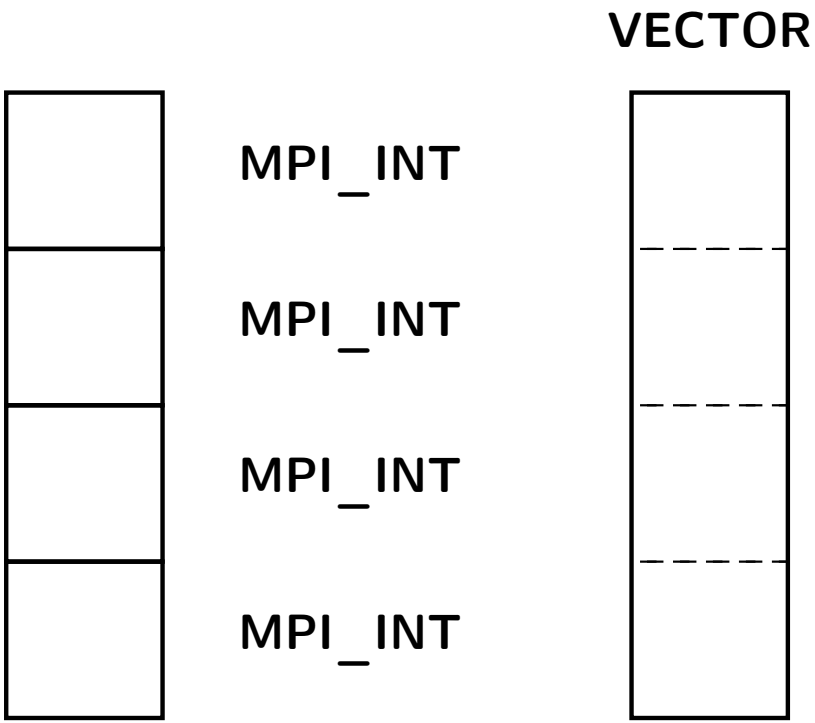
`MPI_CHAR`, `MPI_SHORT`, `MPI_INT`, `MPI_LONG`, `MPI_UNSIGNED_CHAR`, `MPI_UNSIGNED_SHORT`, `MPI_UNSIGNED_LONG`, `MPI_UNSIGNED`, `MPI_FLOAT`, `MPI_DOUBLE`, `MPI_LONG_DOUBLE`, `MPI_BYTE`

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

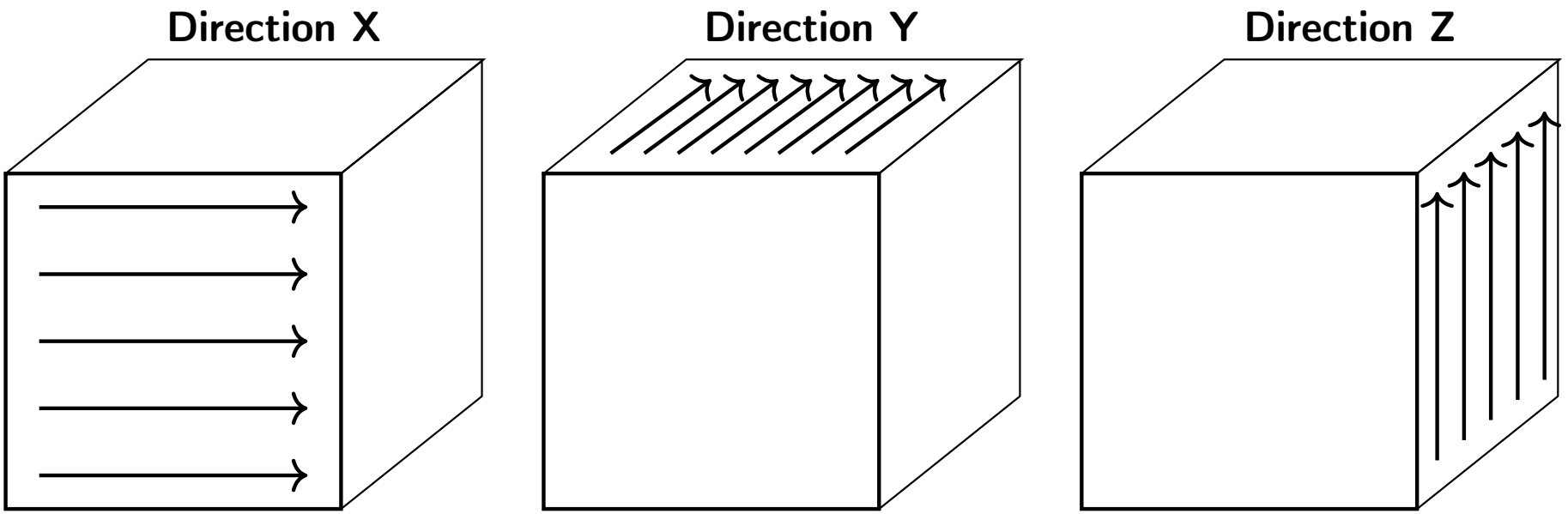
- count = 2 : on a deux éléments
- array_of_blocklengths = 1, 2 : on a 1 entier et 2 doubles
- array_of_displacements = 0, sizeof(int), sizeof(int) + sizeof(double) (ou utilisation de MPI_Address sur une instance de cette structure)
- array_of_types = MPI_INT, MPI_DOUBLE

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

$$\begin{array}{cccc} x & x & x & x \\ o & o & o & o \\ o & o & o & o \\ o & o & o & o \end{array} \rightarrow \begin{array}{cccc} x & o & o & o \\ x & o & o & o \\ x & o & o & o \\ x & o & o & o \end{array}$$

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)

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

$$A = QR$$

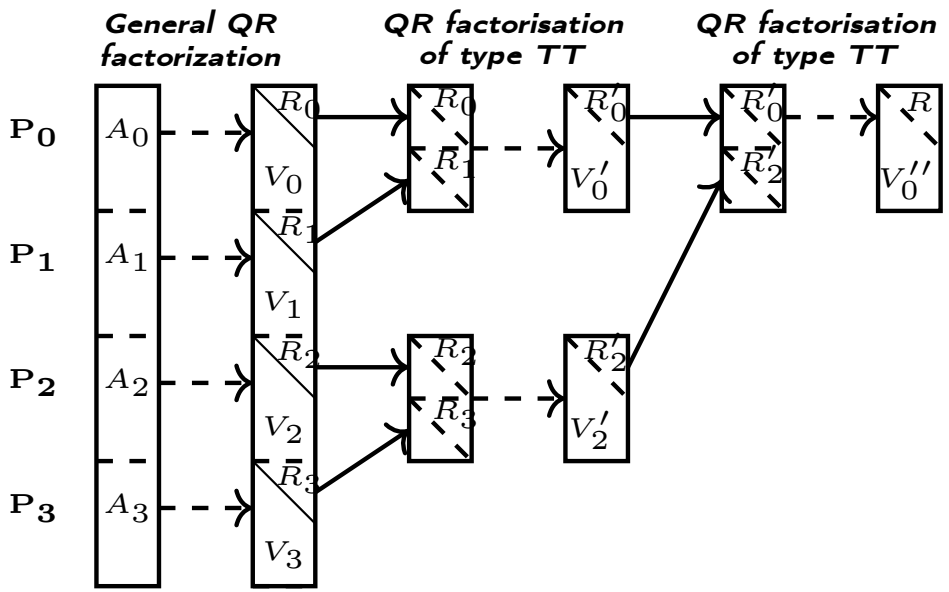
Où Q est une matrice orthogonale ($QQ^T = I$)
et R est une matrice triangulaire supérieure.

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

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

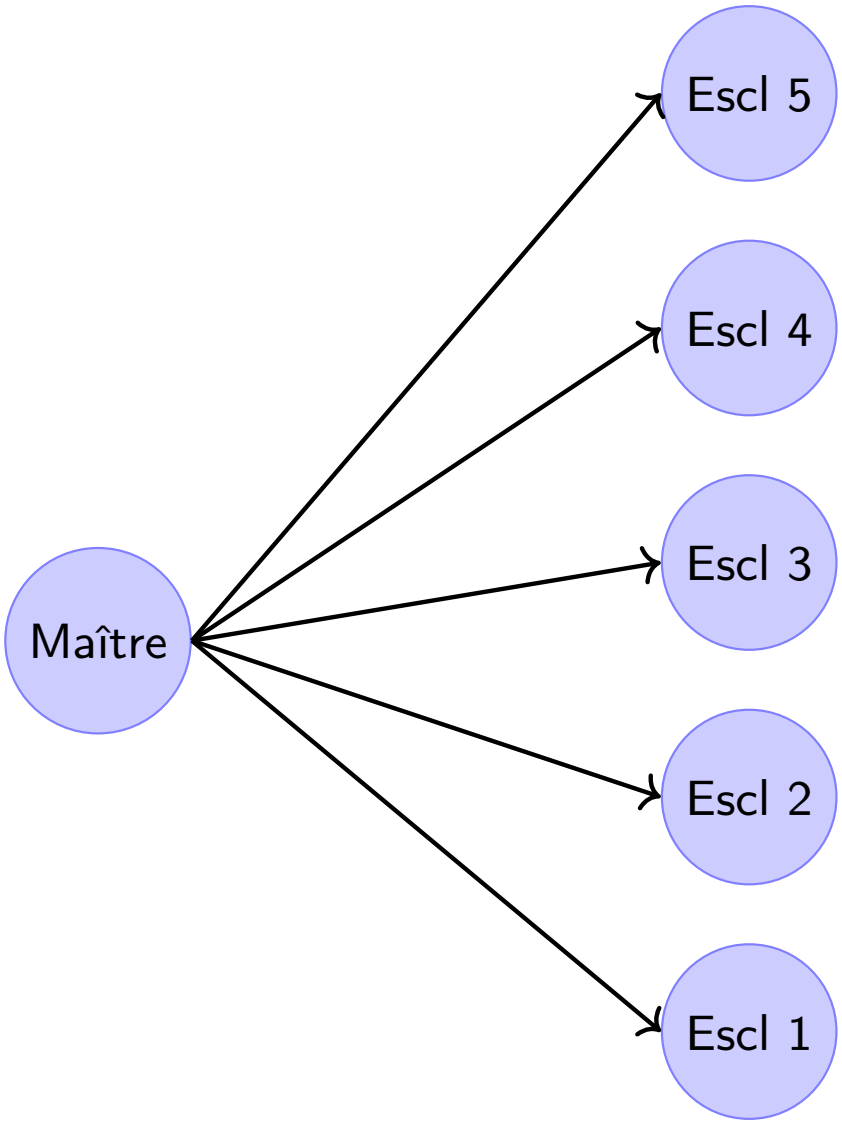
Files d'attentes de données et résultats au niveau du maître

- On retrouve la partie séquentielle de la loi d'Amdahl
- Communications : maître $\leftrightarrow$ esclaves
- Les esclaves ne travaillent que 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

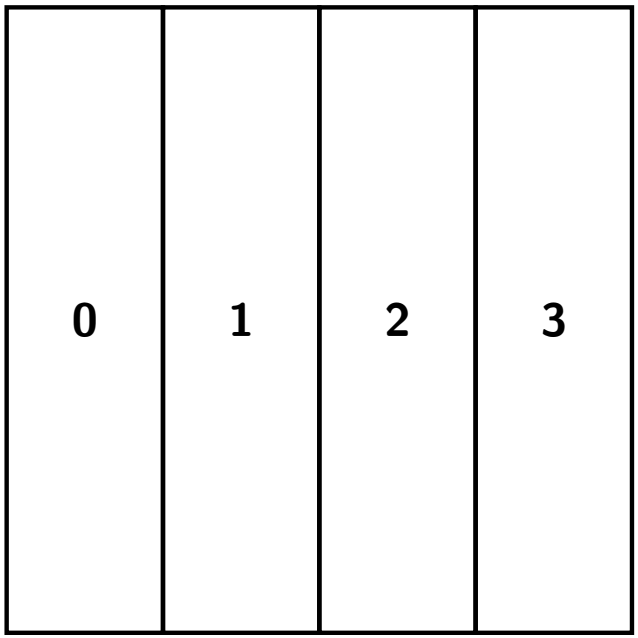
Découpage en grille

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

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

Ghost region

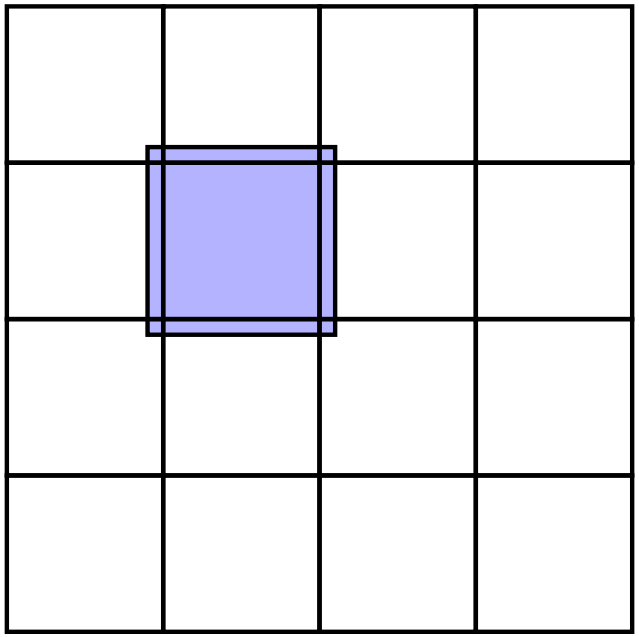
Frontières entre les sous-domaines

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



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

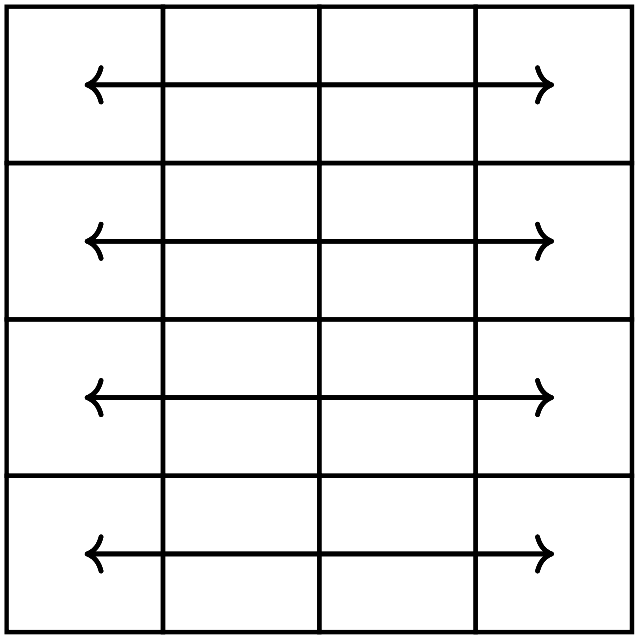
- `comm_old` : le communicateur de départ
- `ndims` : ici 2
- `dims` : nombre de processus dans chaque dimension (ici {4, 4})
- `periods` : les dimensions sont-elles périodiques
- `reorder` : autoriser ou non de modifier les rangs des processus
- `comm_cart` : nouveau communicateur

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

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

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 ?

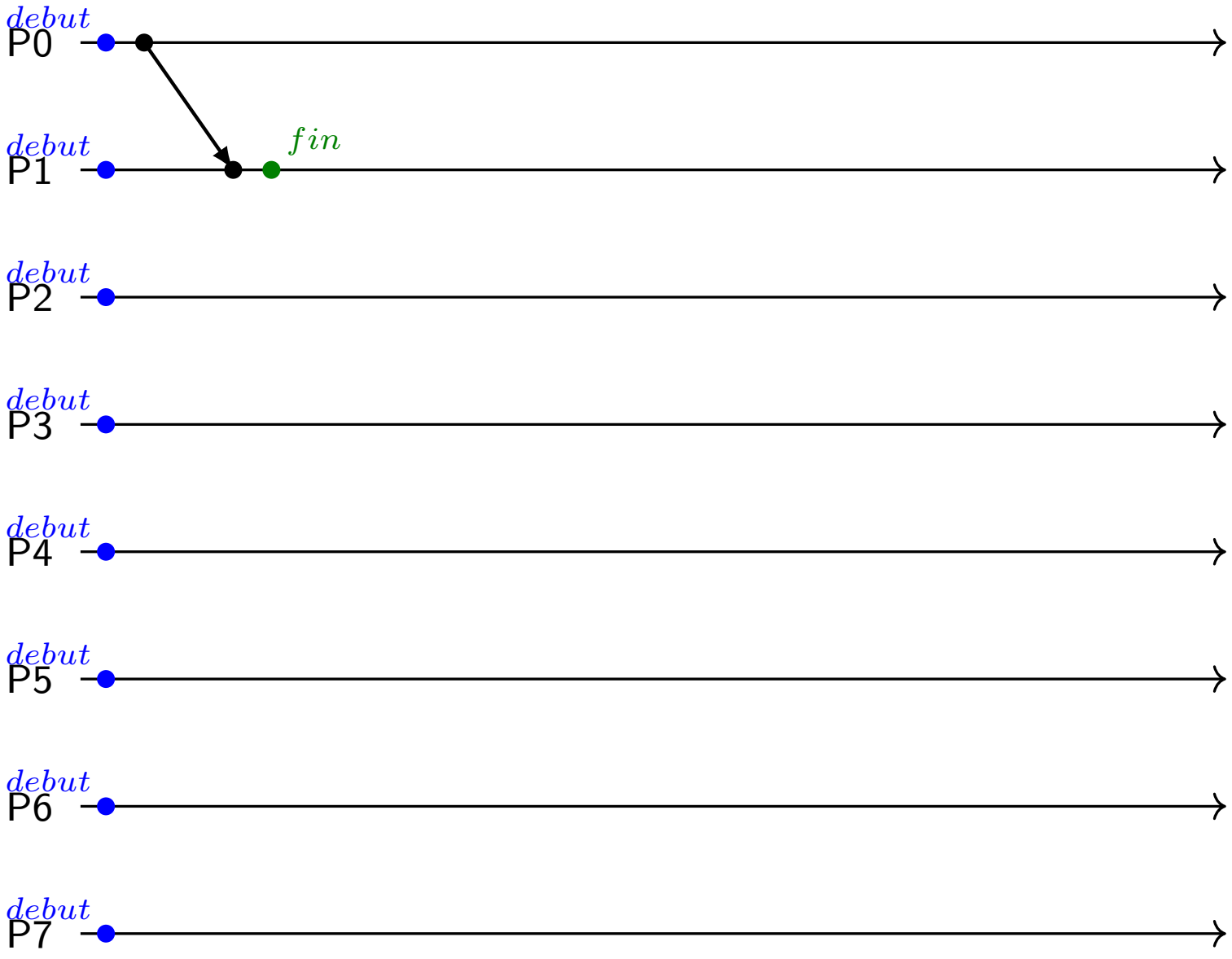
- Nombre de messages envoyés / reçus (latence)
- Utilisation de la bande passante

Paramètres :

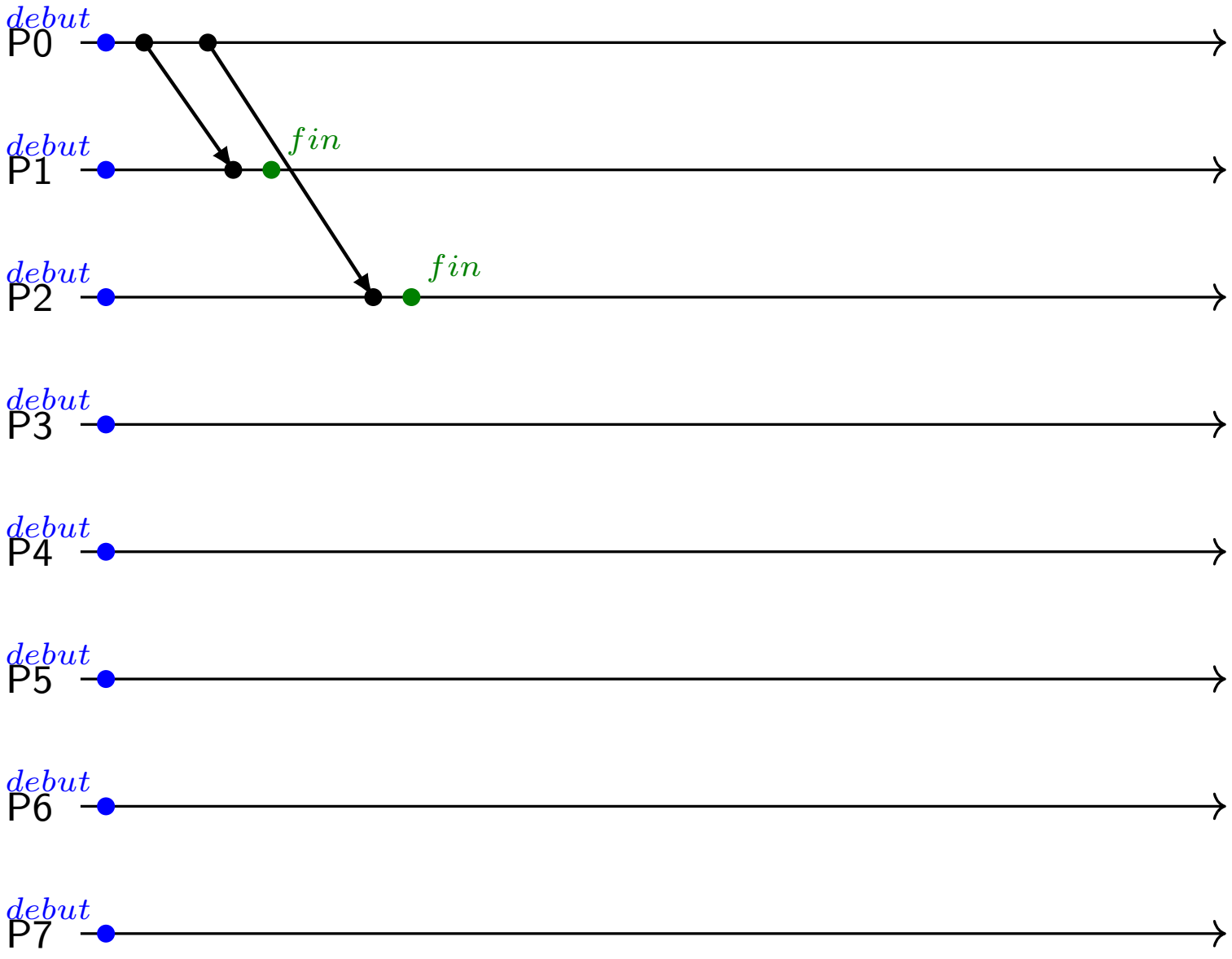
- Taille du message
- Nombre de processus impliqués !

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...

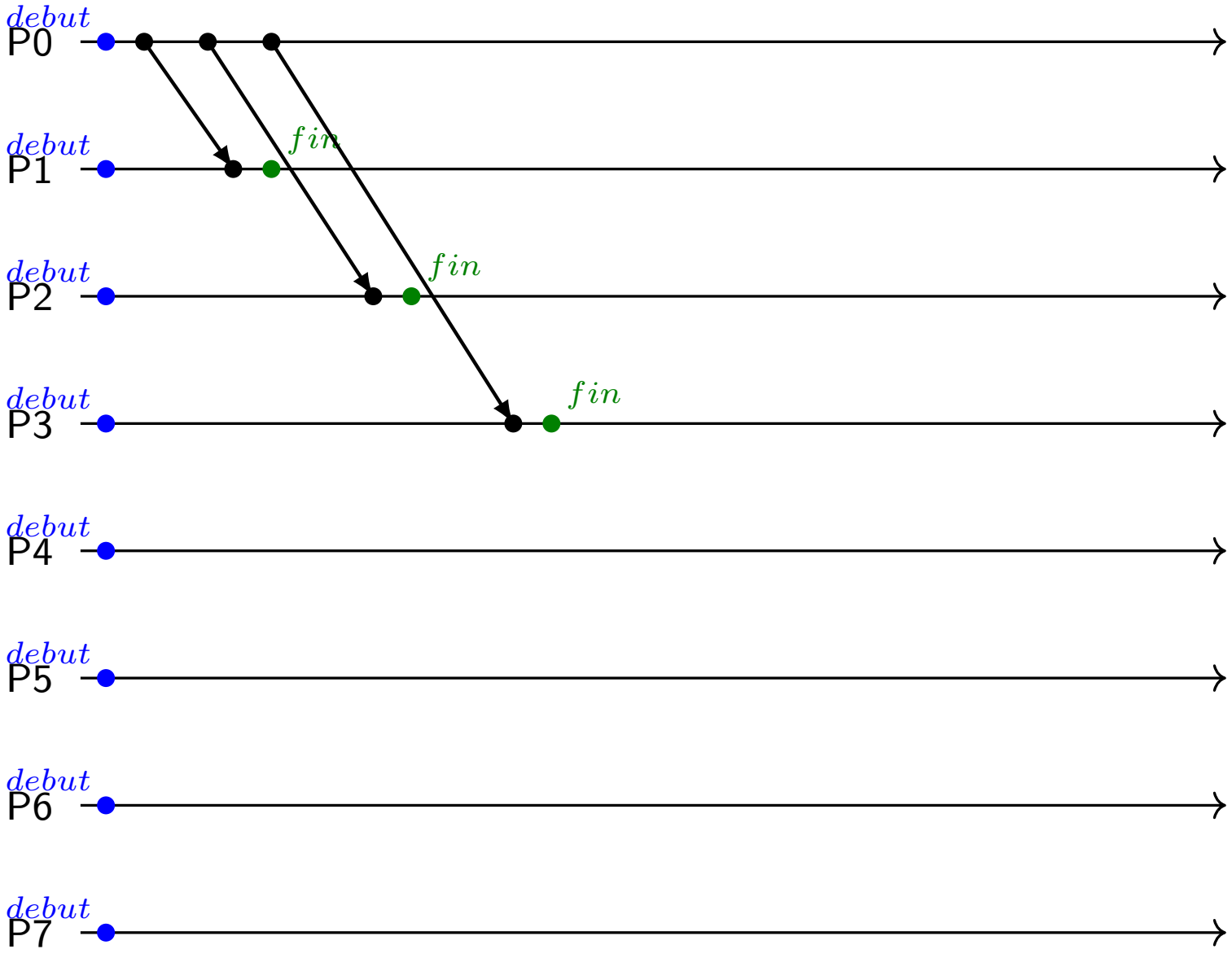
Modèle pour les communications : Bcast étoile



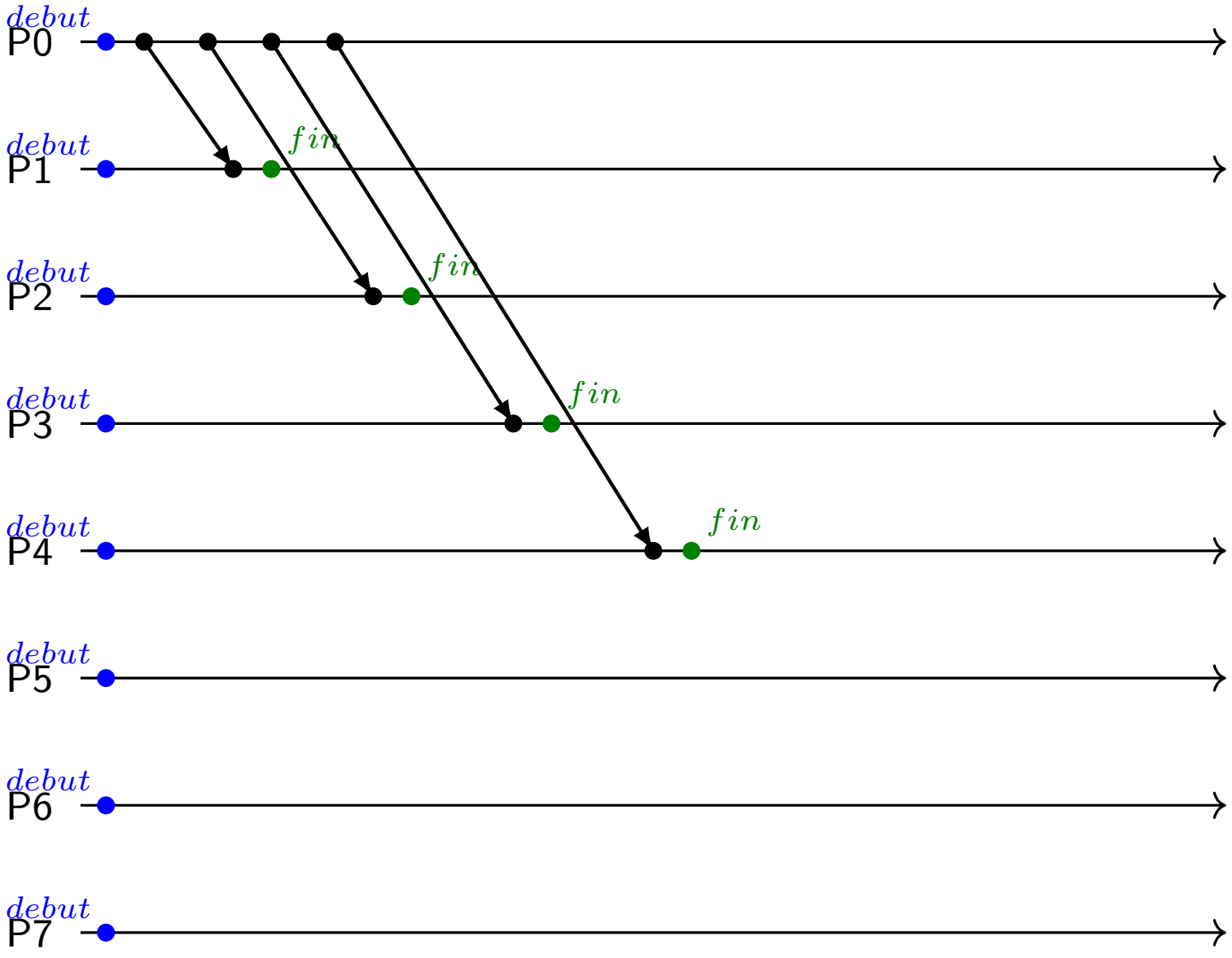
Modèle pour les communications : Bcast étoile



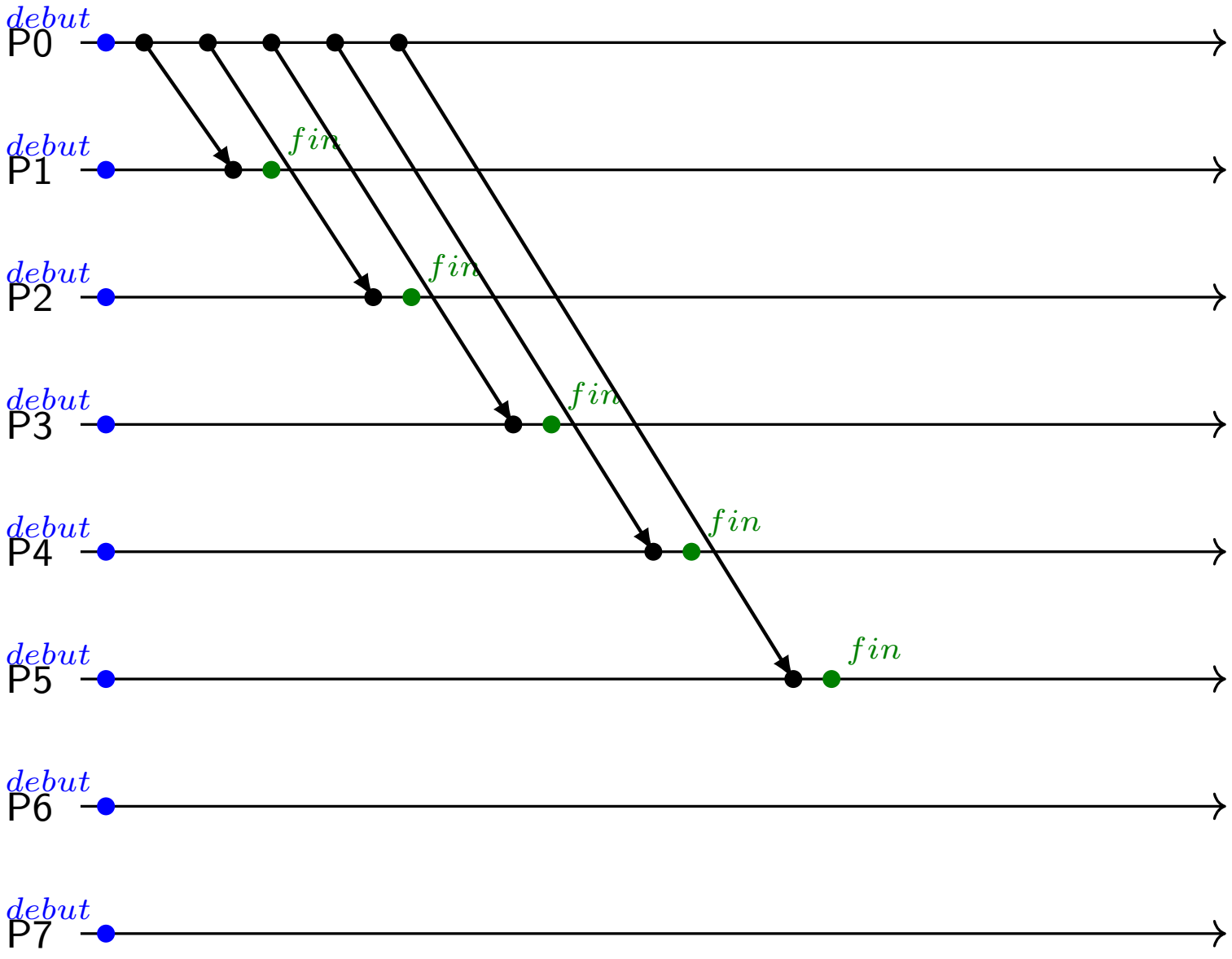
Modèle pour les communications : Bcast étoile



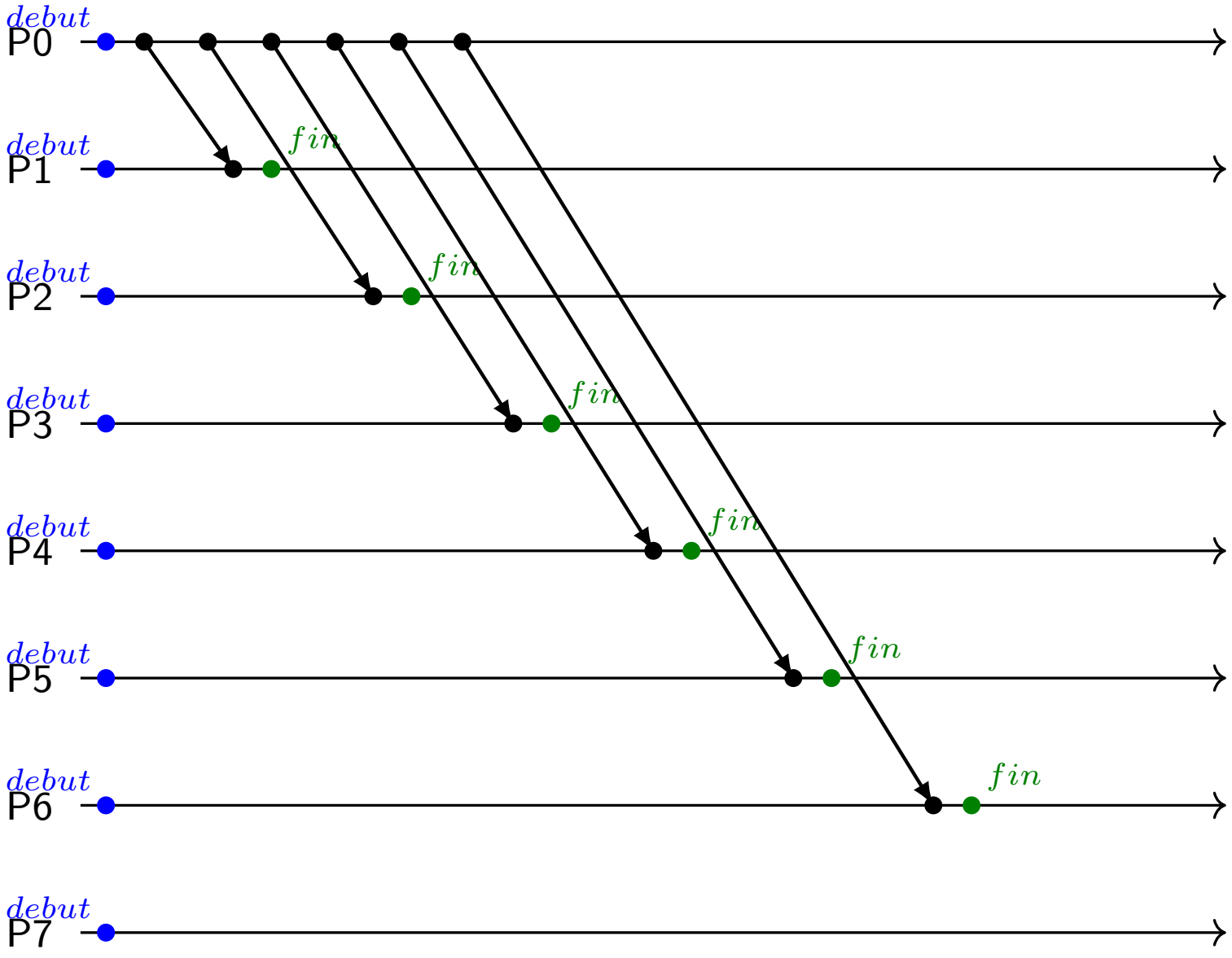
Modèle pour les communications : Bcast étoile



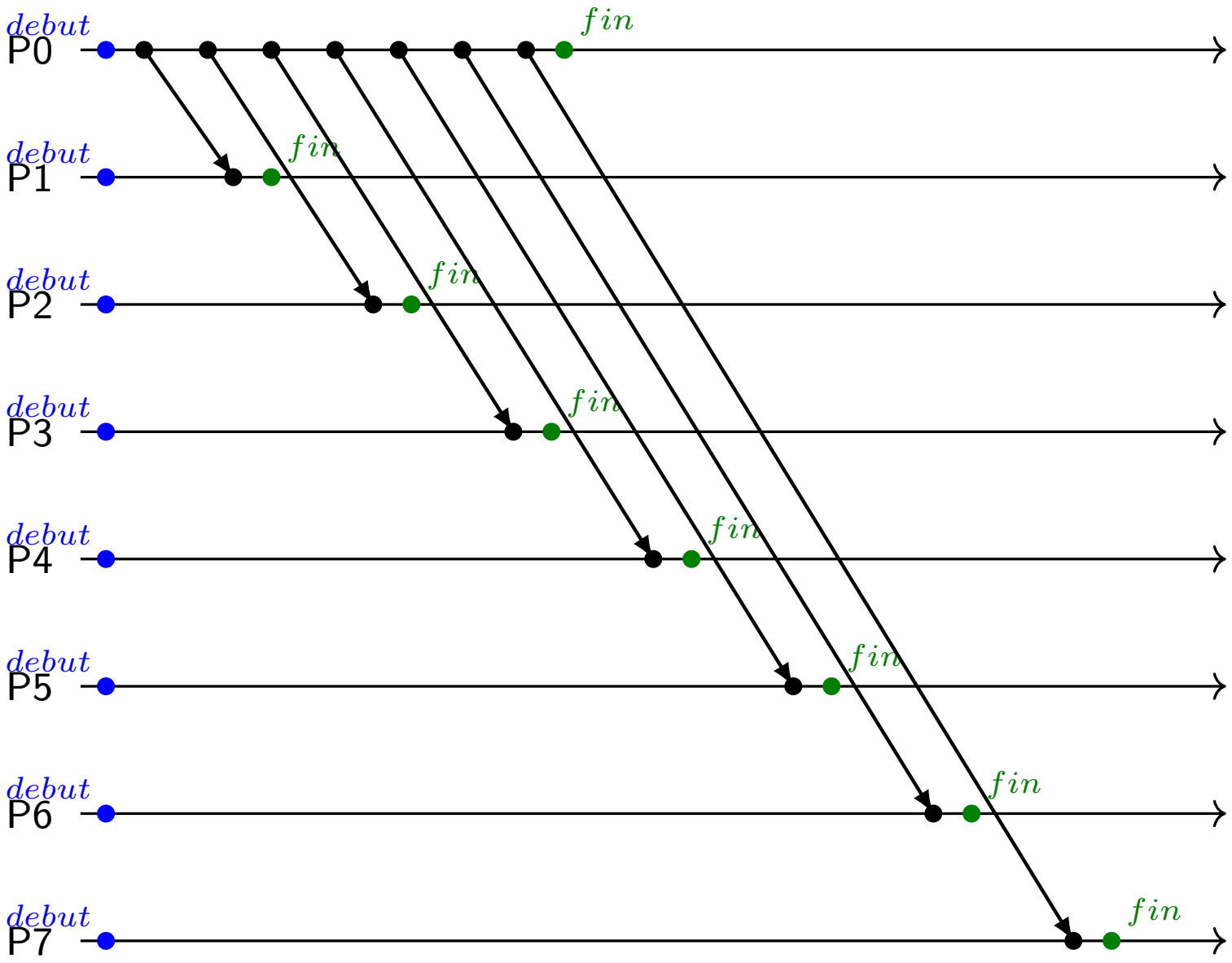
Modèle pour les communications : Bcast étoile



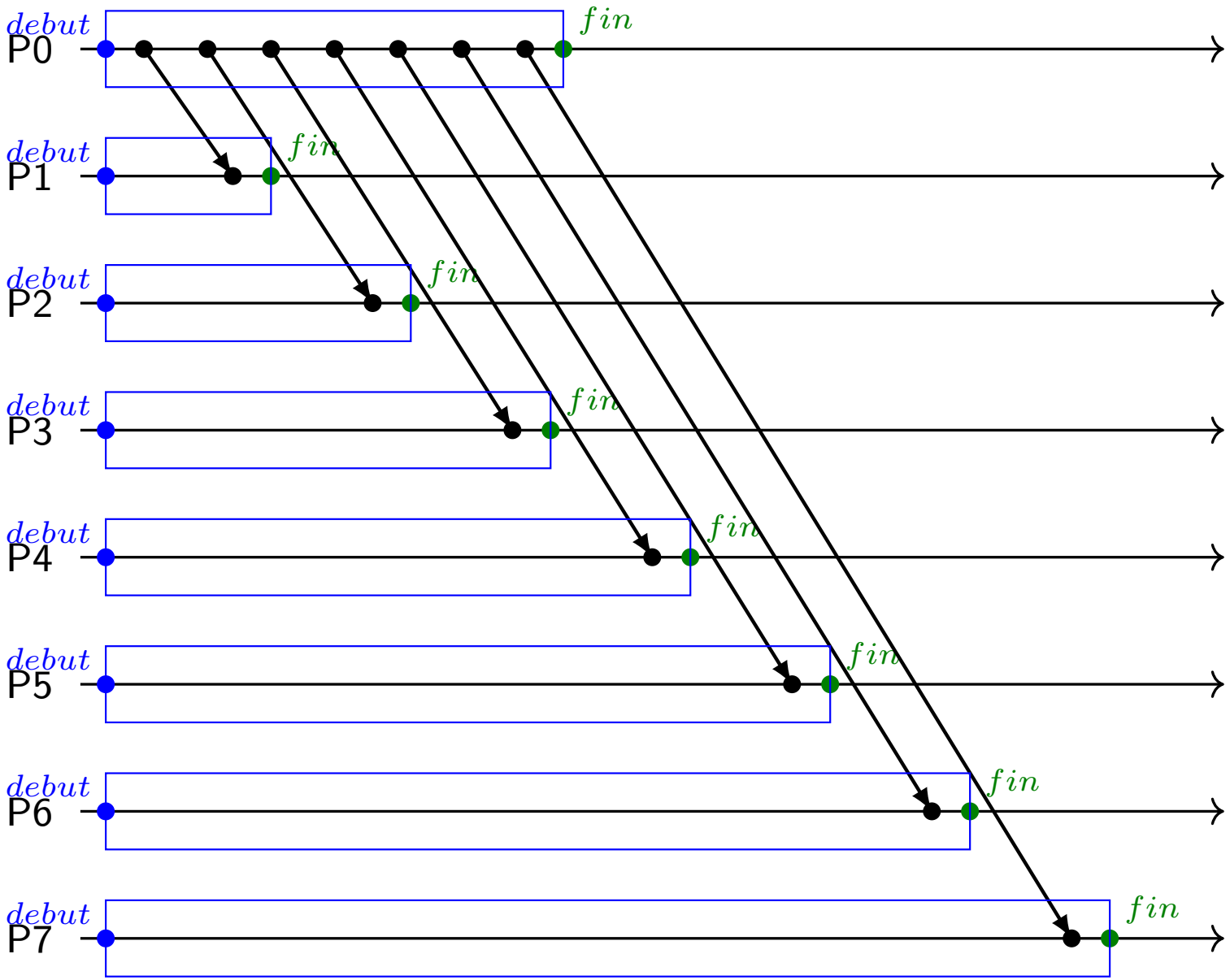
Modèle pour les communications : Bcast étoile



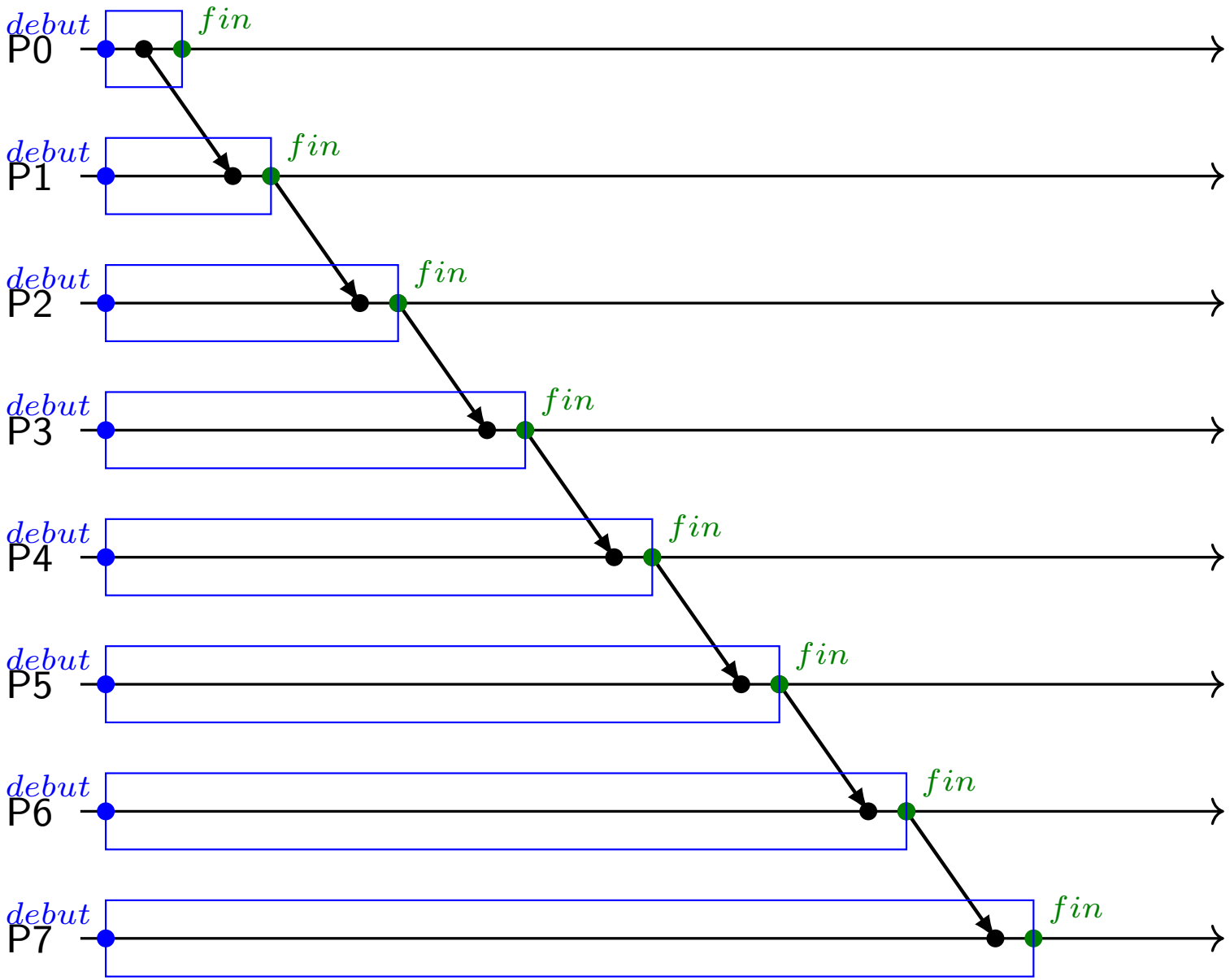
Modèle pour les communications : Bcast étoile



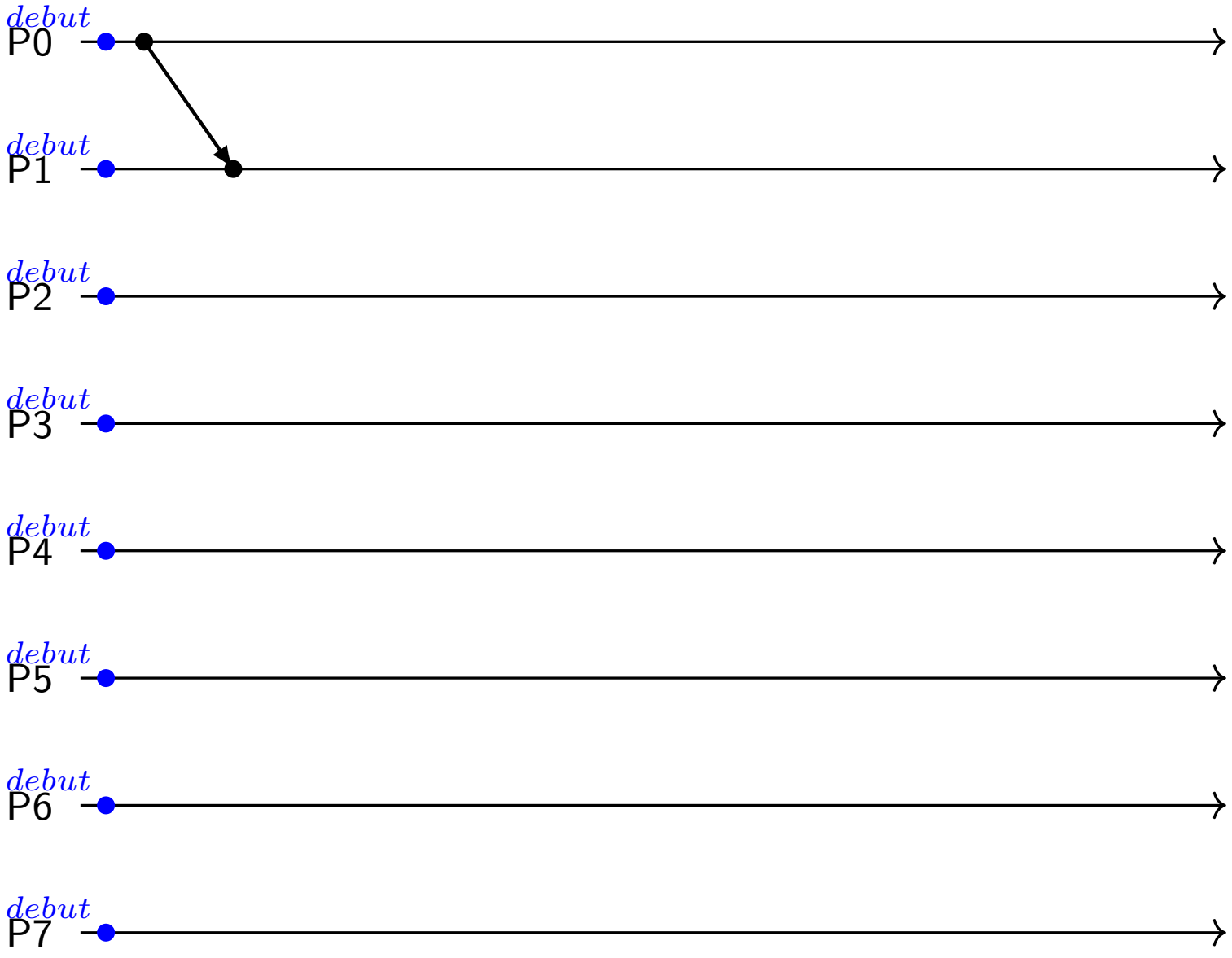
Modèle pour les communications : Bcast étoile



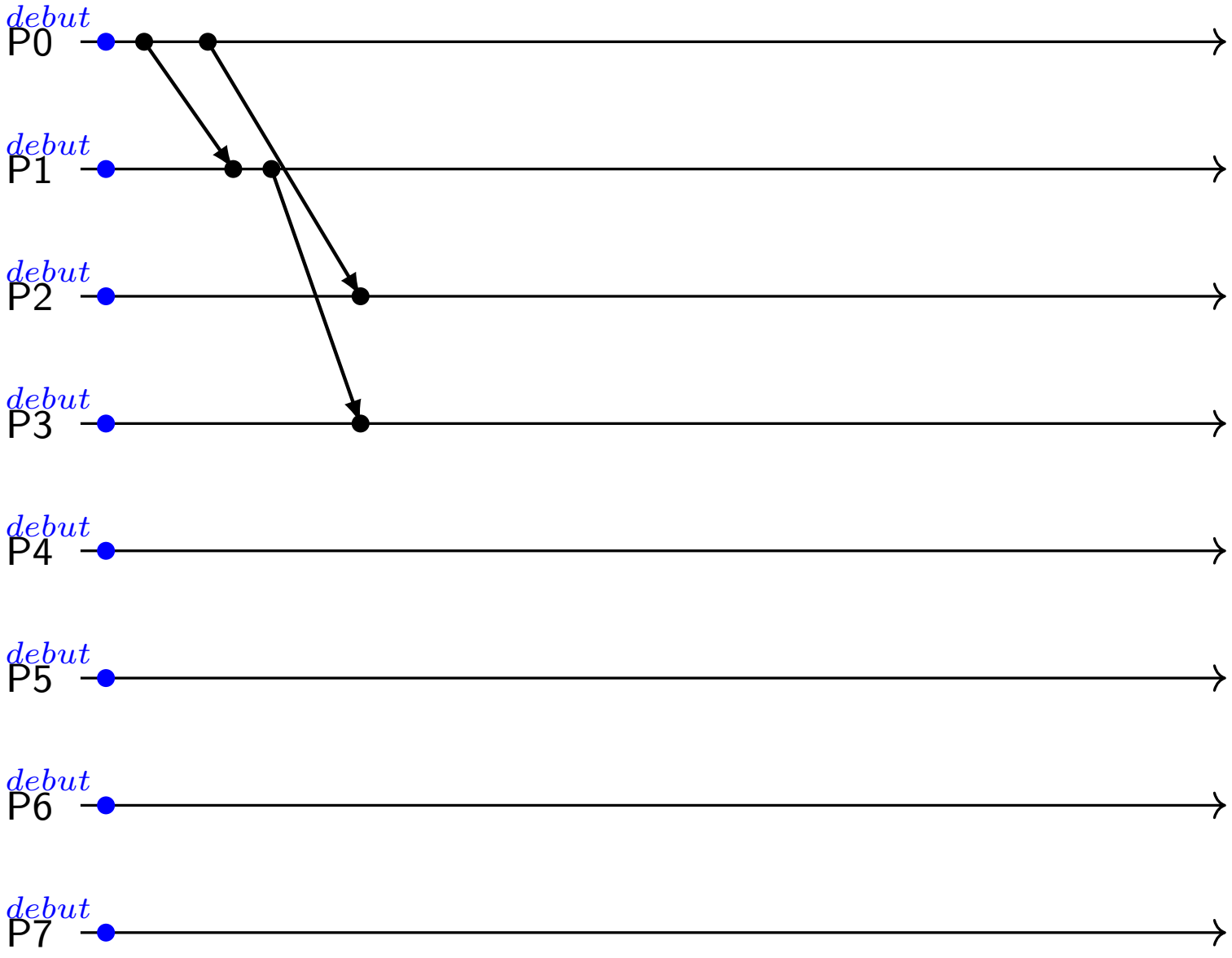
Modèle pour les communications : Bcast chaine



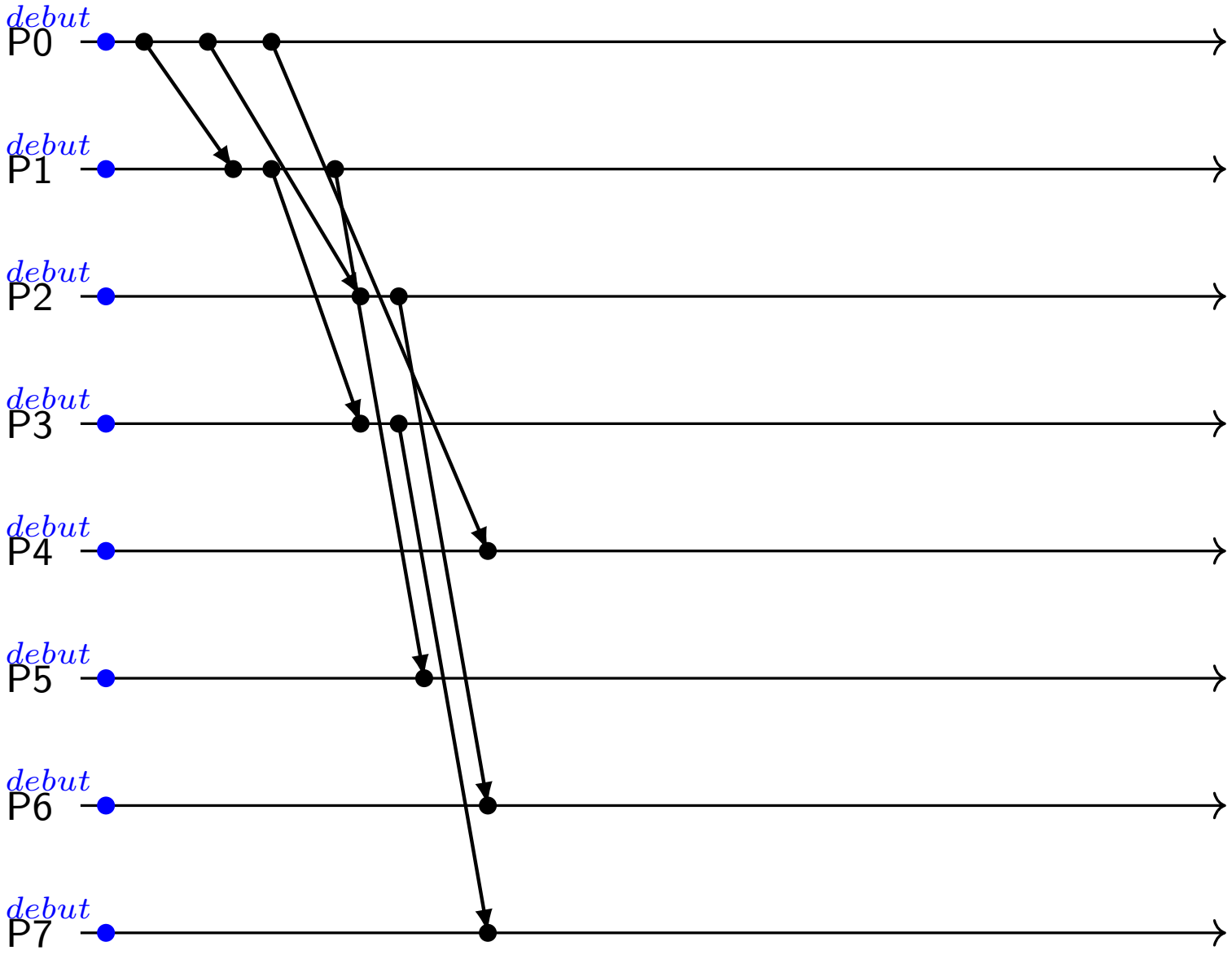
Modèle pour les communications : Bcast binomial



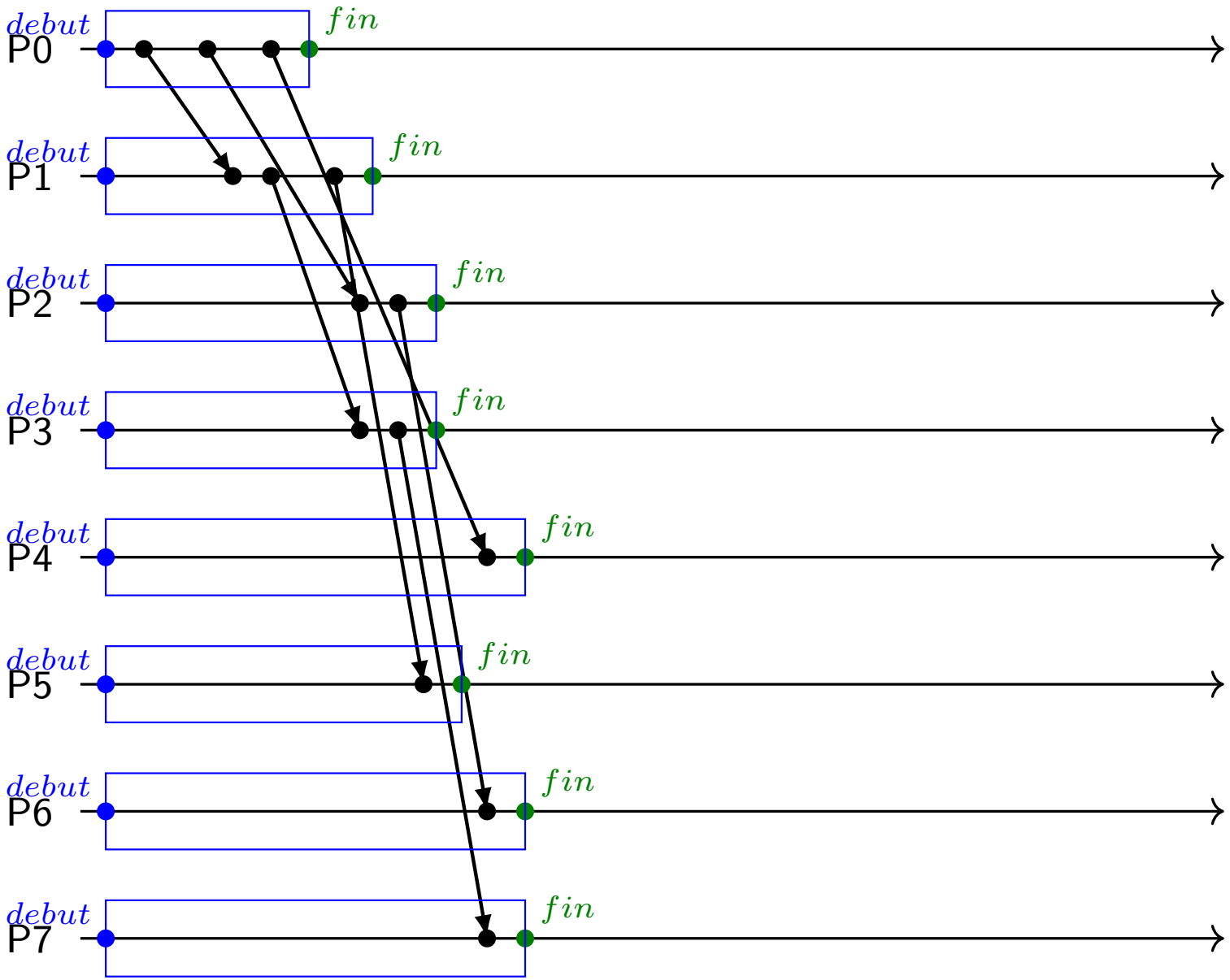
Modèle pour les communications : Bcast binomial



Modèle pour les communications : Bcast binomial



Modèle pour les communications : Bcast binomial



Communications collectives

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

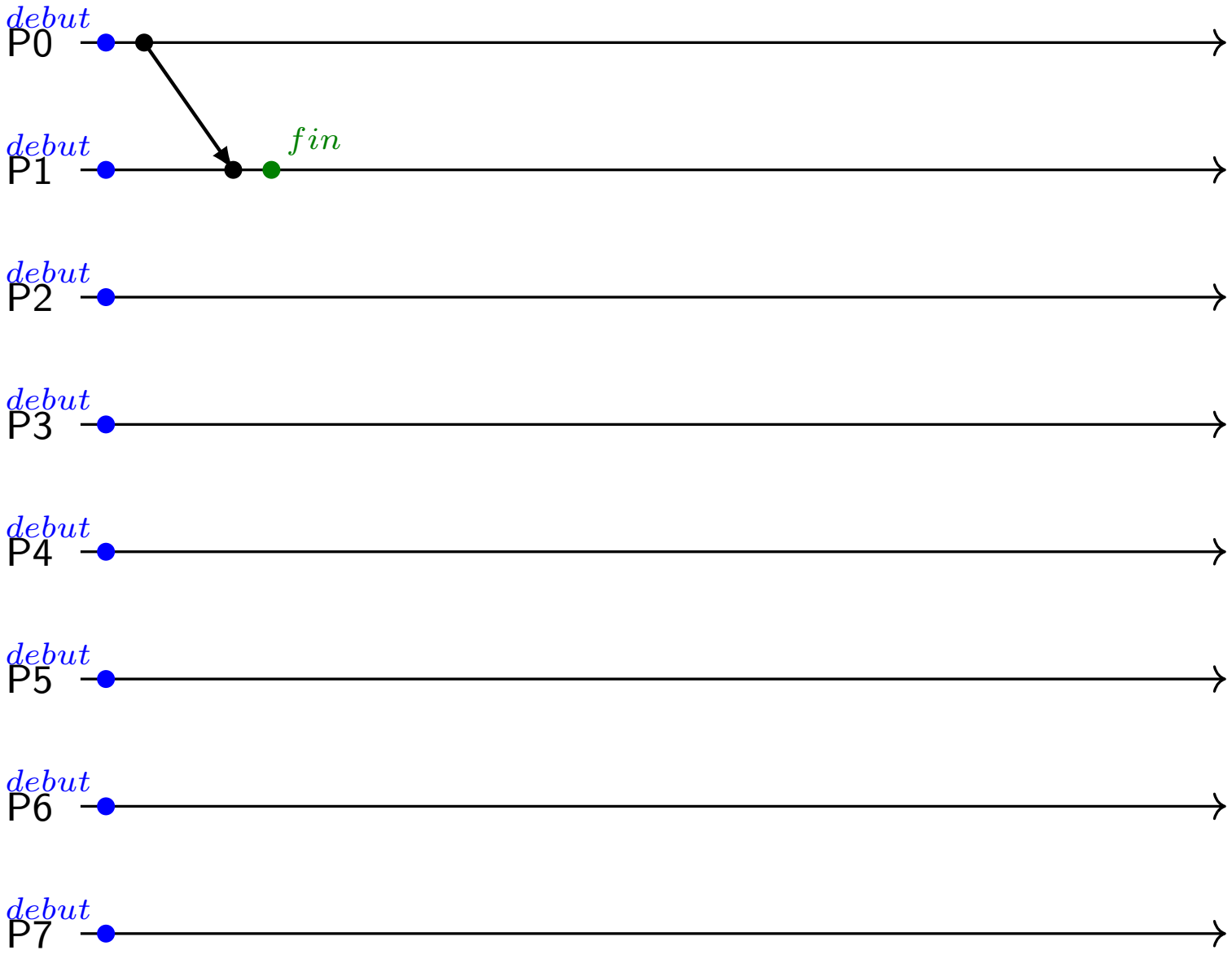
Le processus racine envoie à tous les autres processus du communicateur :

```
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 ! ! !

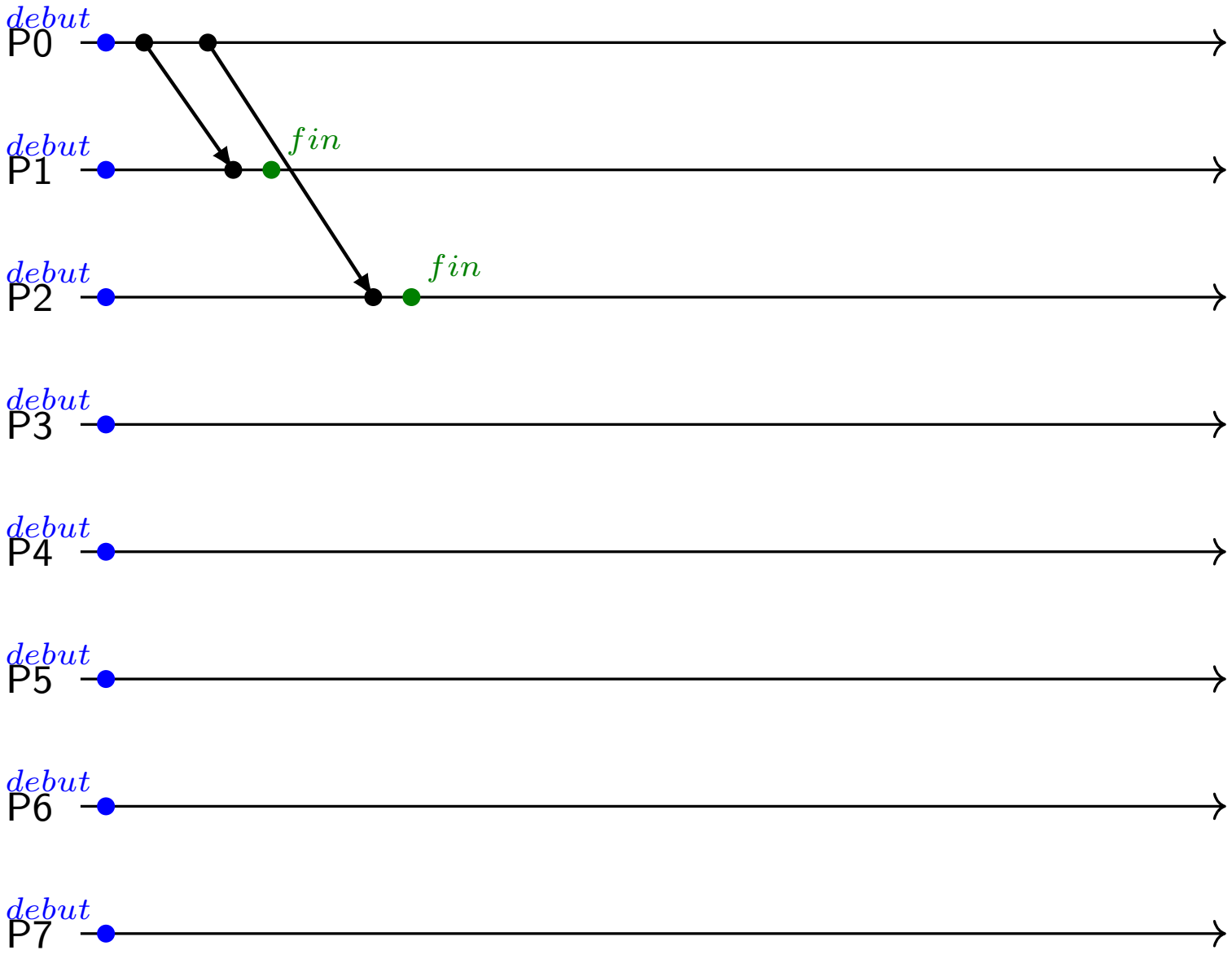
Communications collectives

Algorithme de diffusion : l'étoile



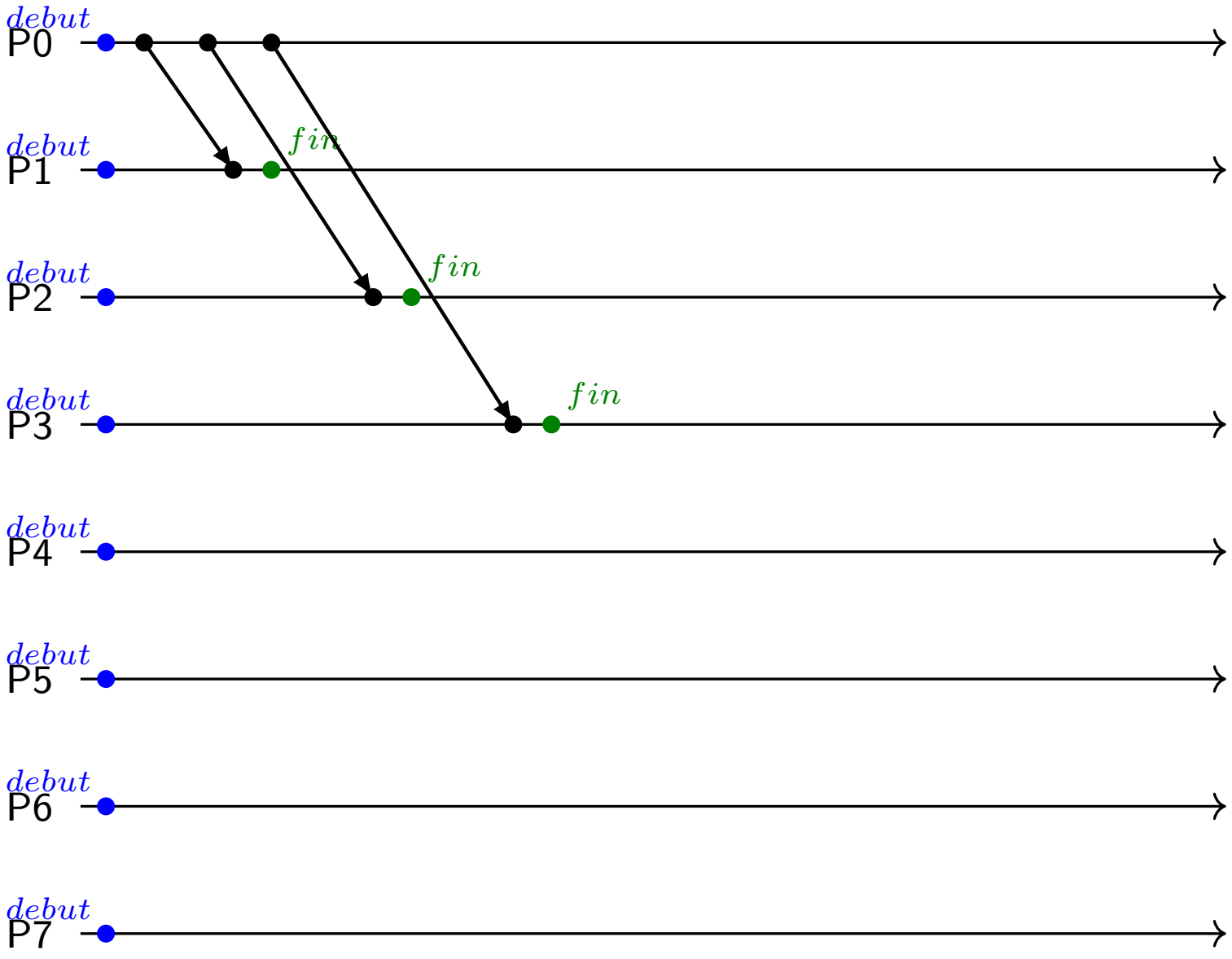
Communications collectives

Algorithme de diffusion : l'étoile

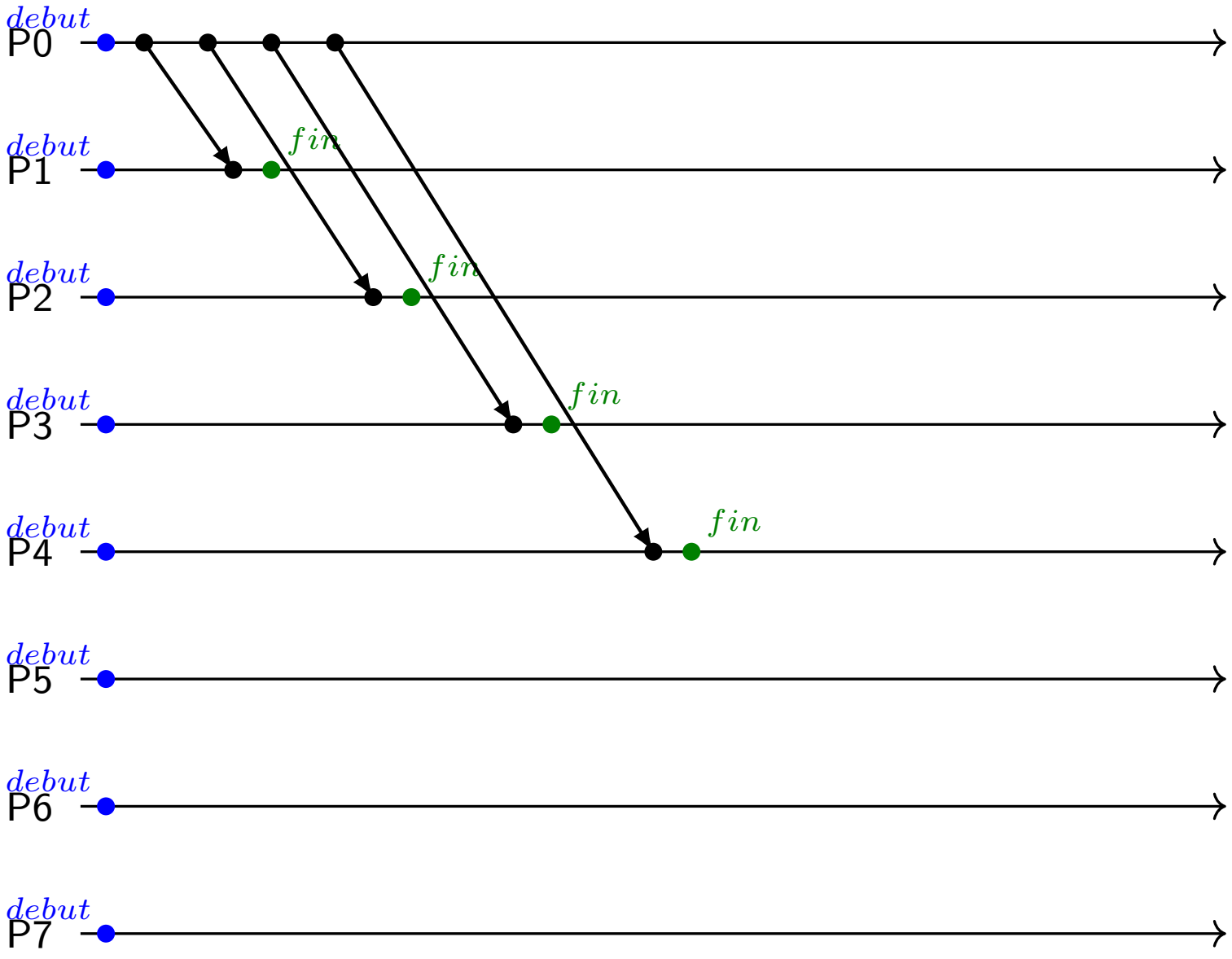


Communications collectives

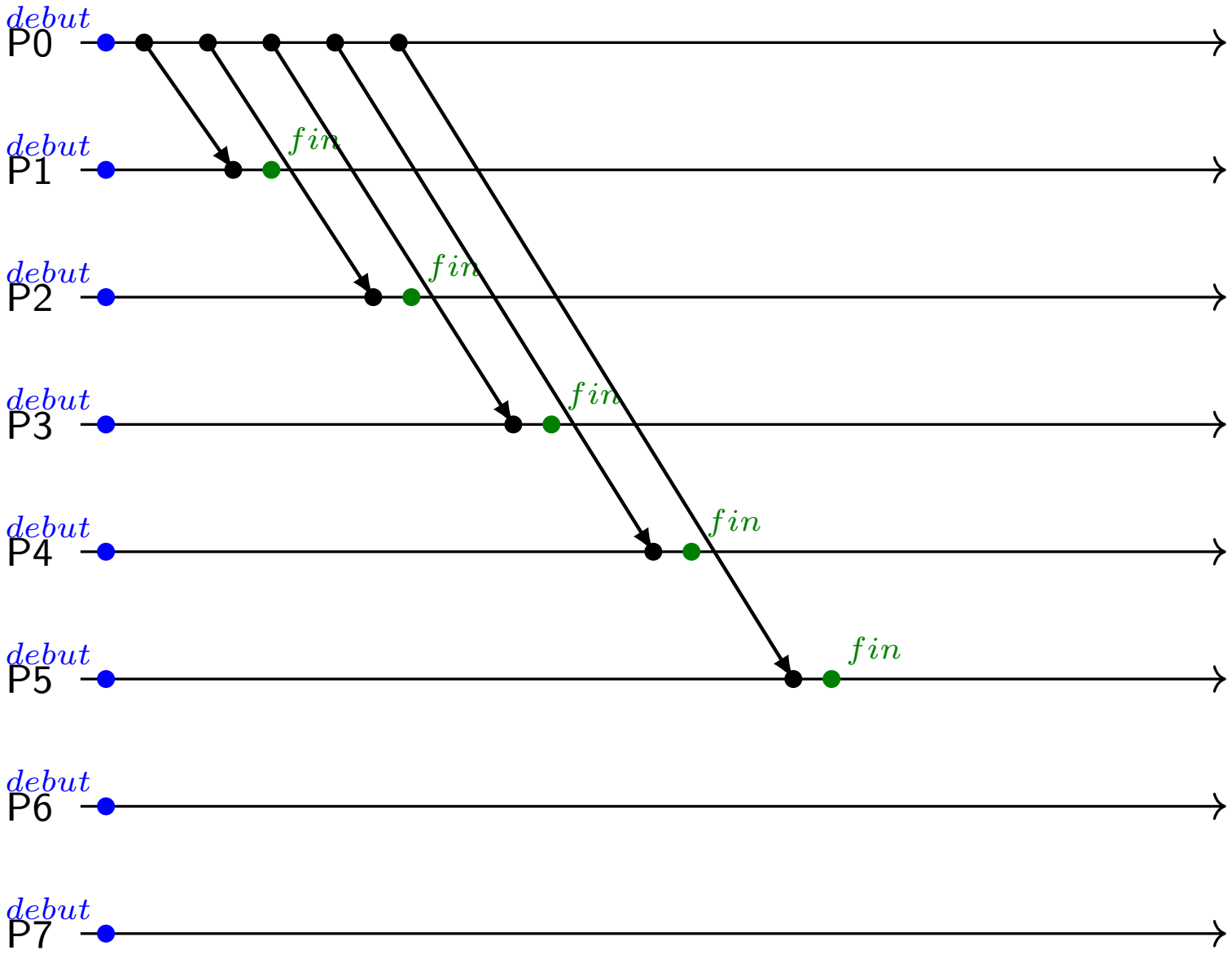
Algorithme de diffusion : l'étoile



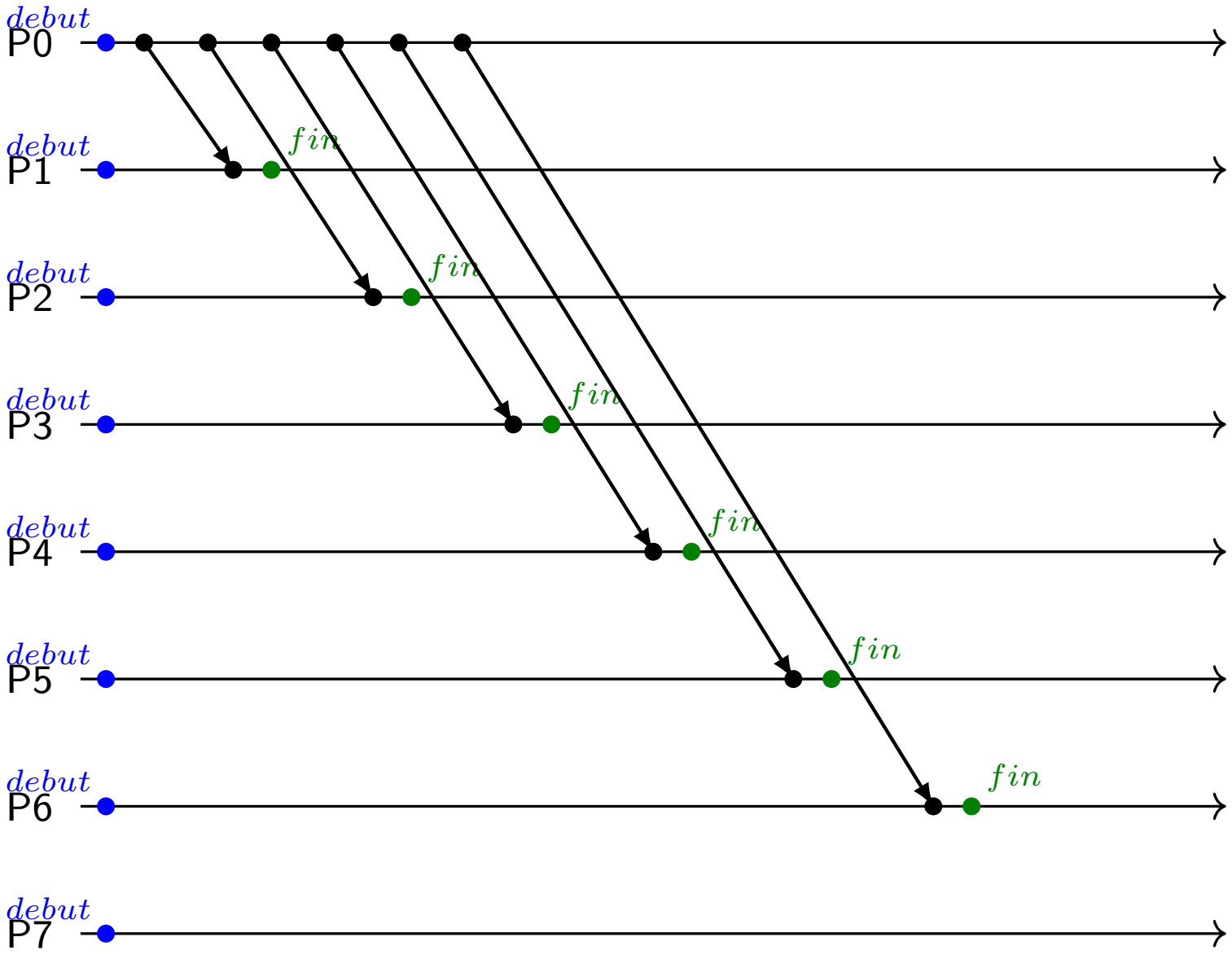
Algorithme de diffusion : l'étoile



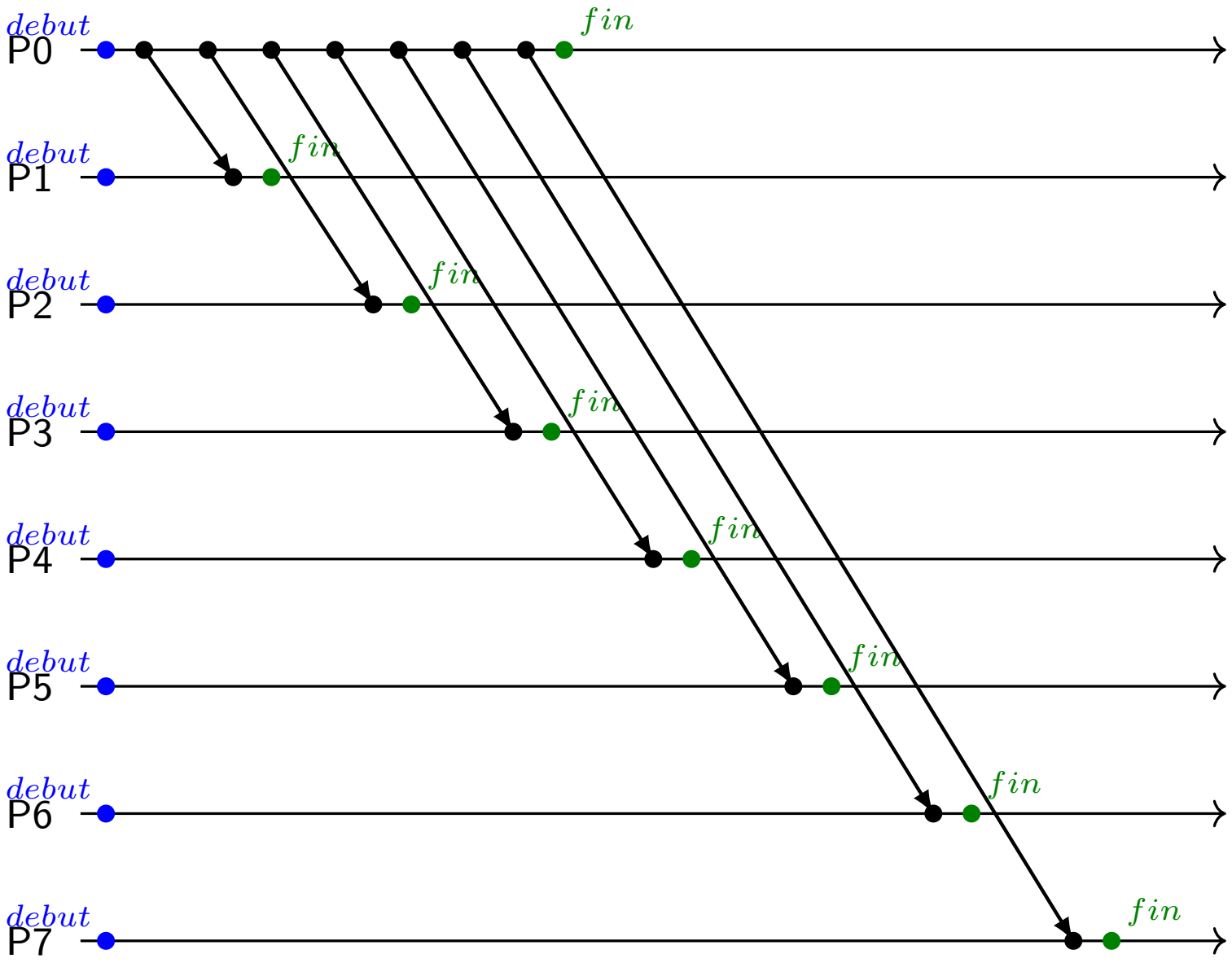
Algorithme de diffusion : l'étoile



Algorithme de diffusion : l'étoile

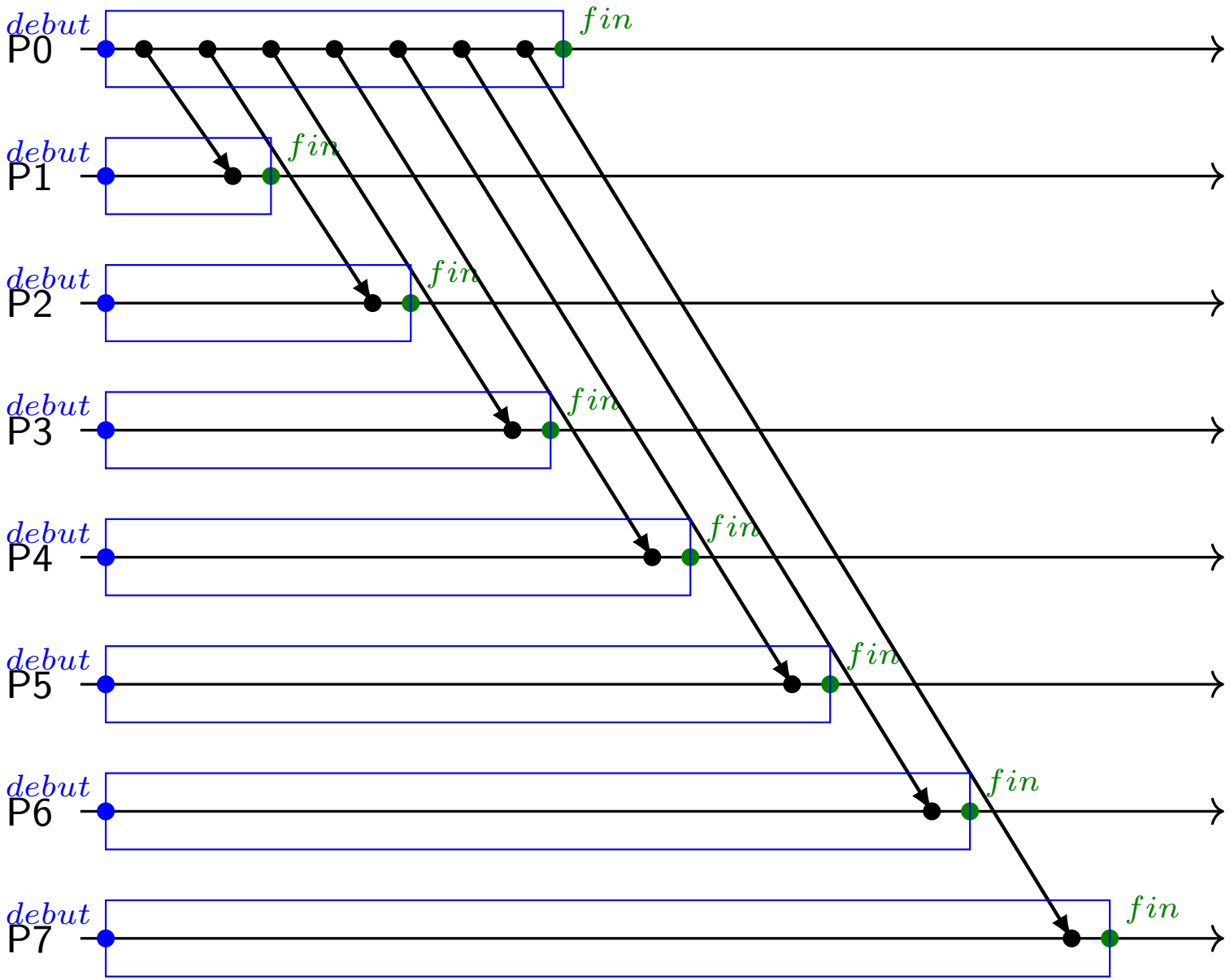


Algorithme de diffusion : l'étoile



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

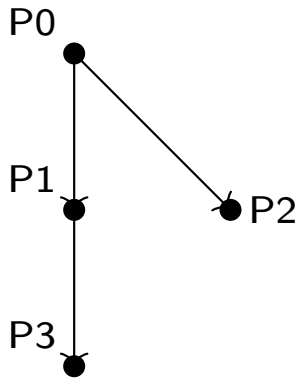
Communications collectives

Algorithme de diffusion : arbre binomial

Communications collectives

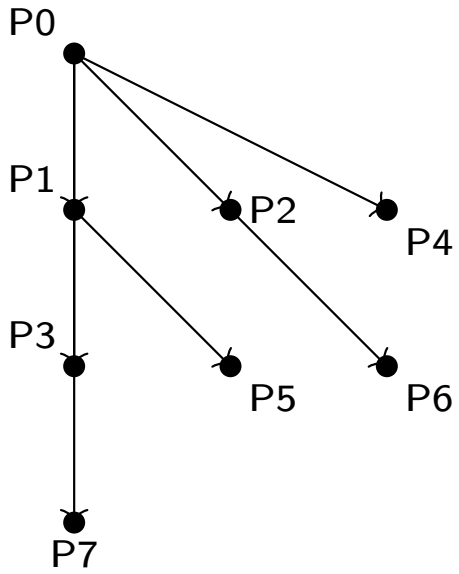
Algorithme de diffusion : arbre binomial

Algorithme de diffusion : arbre binomial



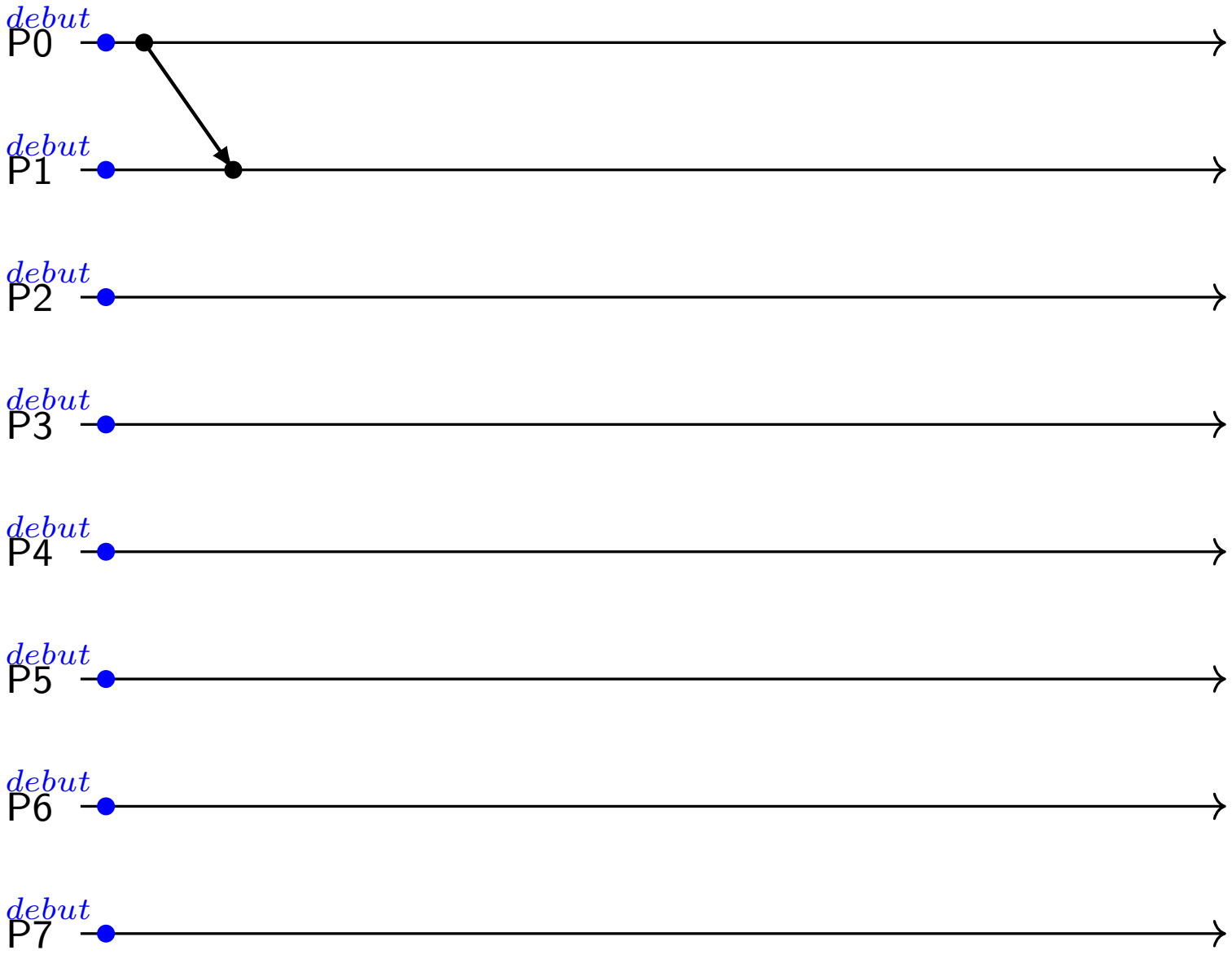
Communications collectives

Algorithme de diffusion : arbre binomial

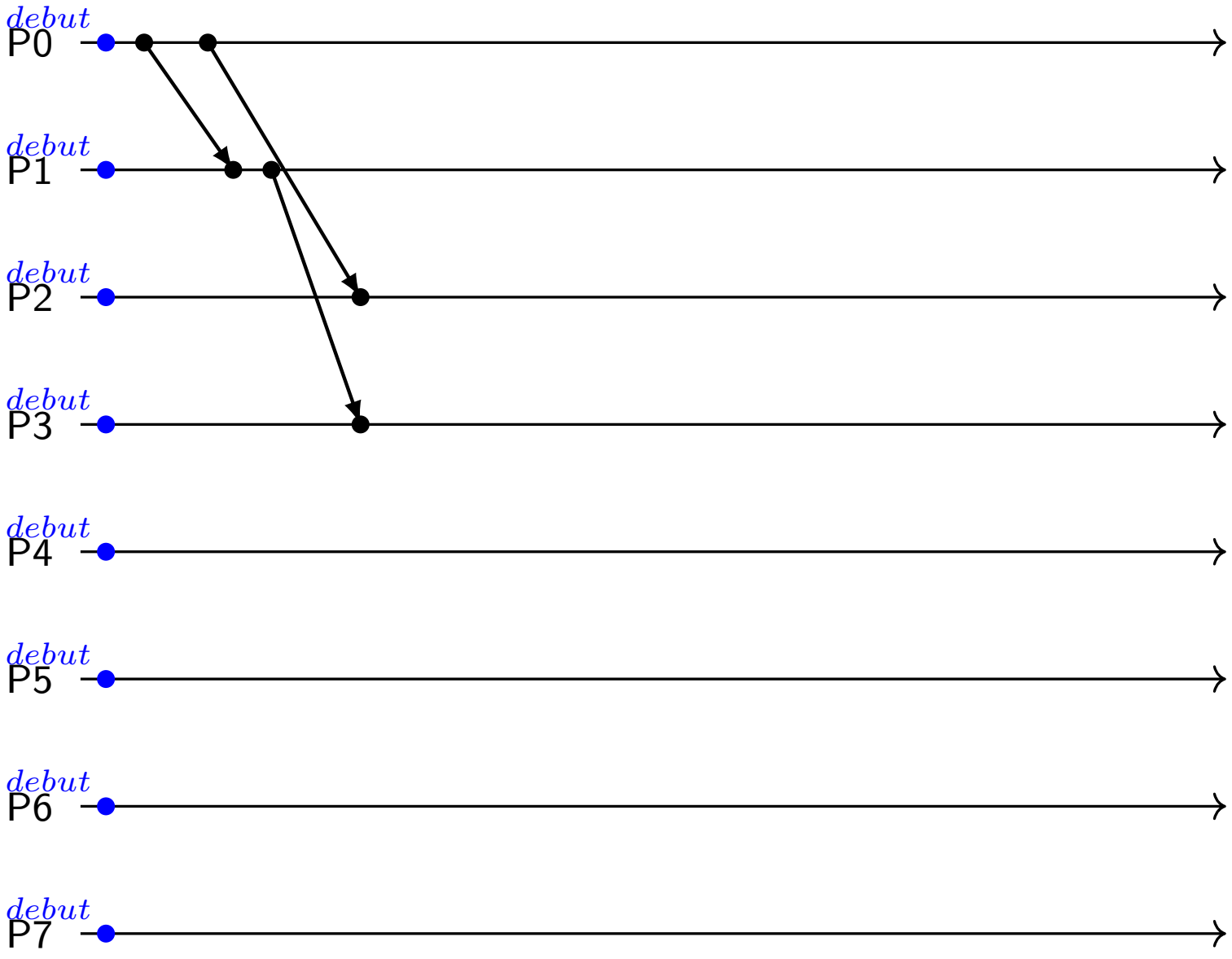


Communications collectives

Algorithme de diffusion : arbre binomial

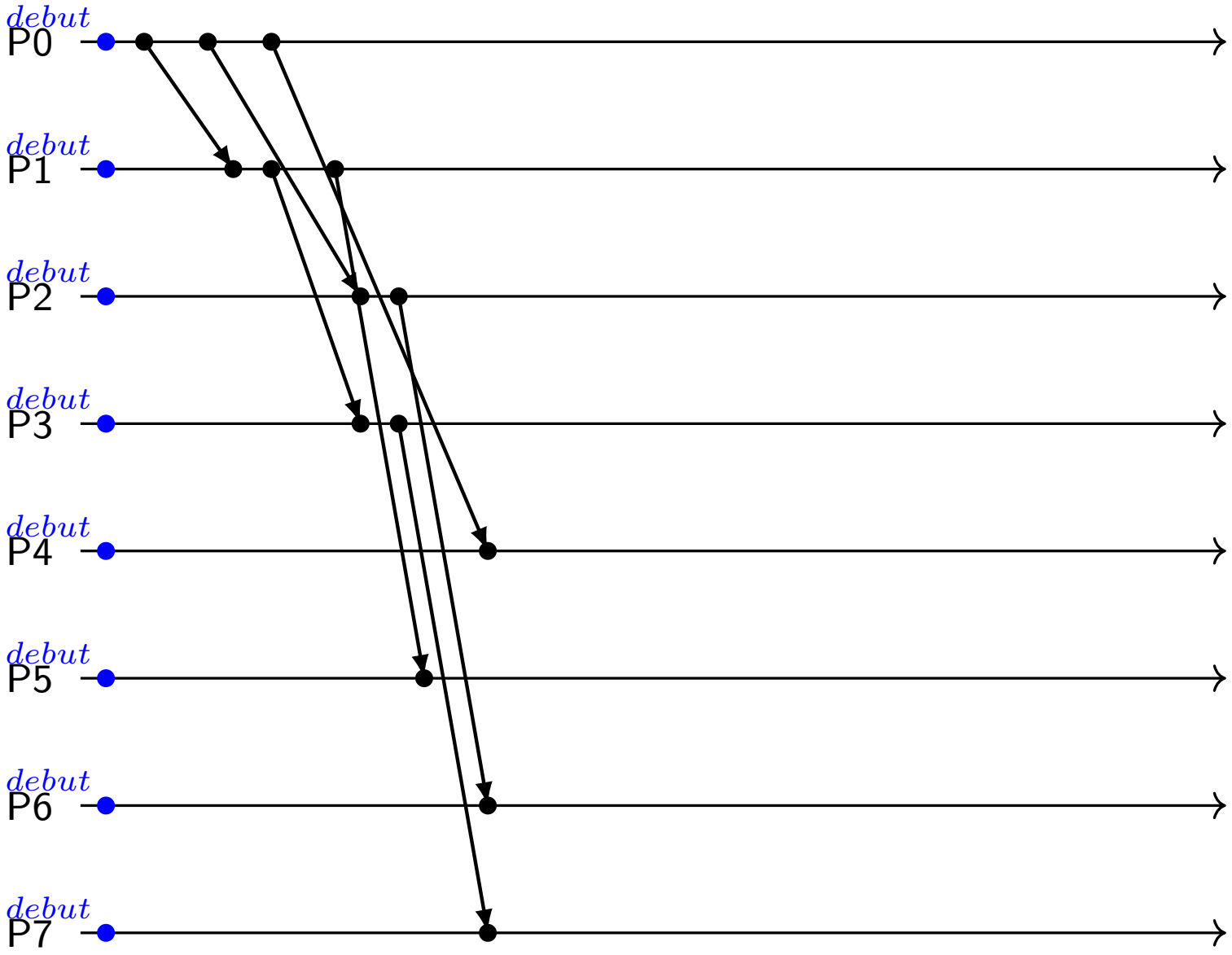


Algorithme de diffusion : arbre binomial



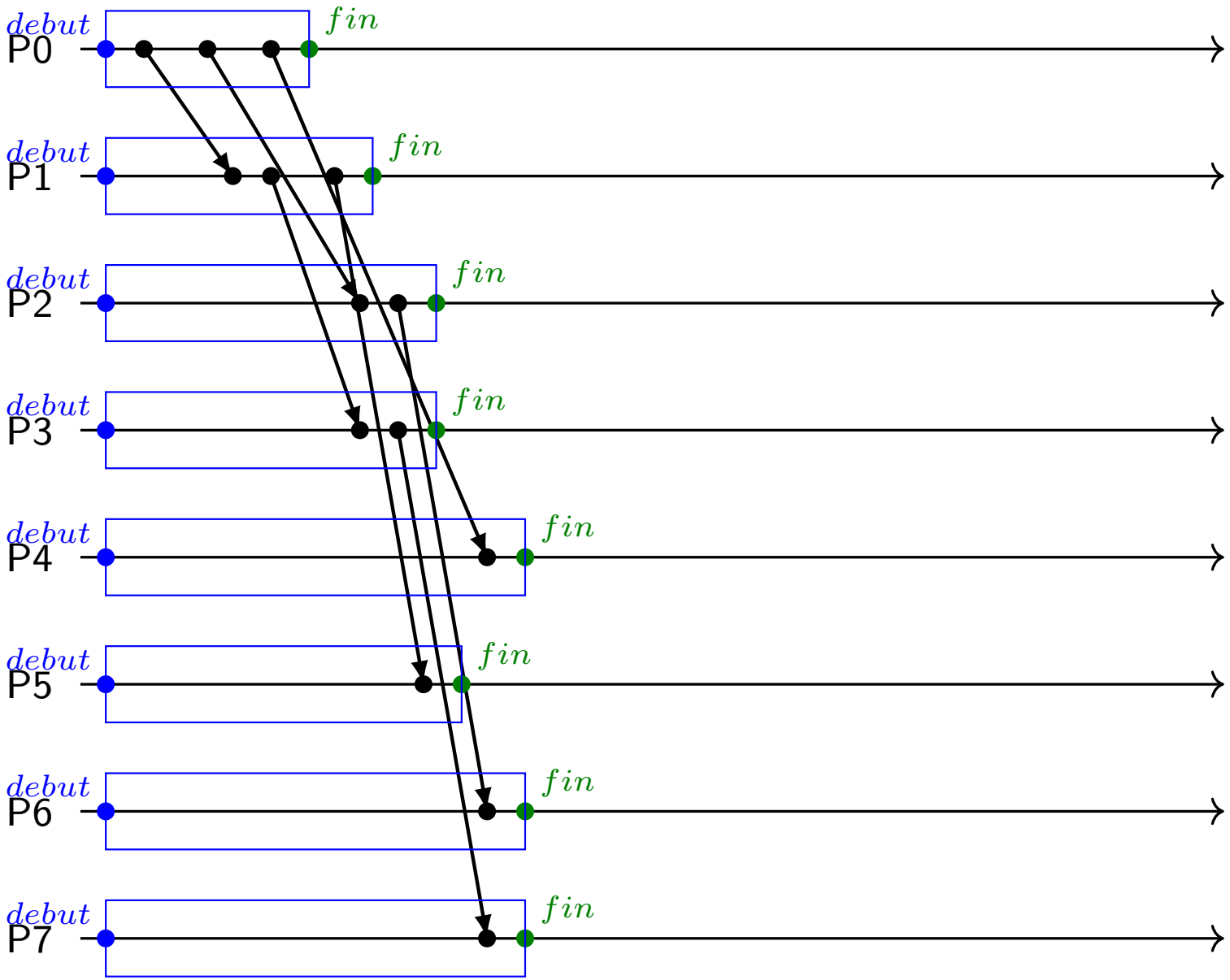
Communications collectives

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

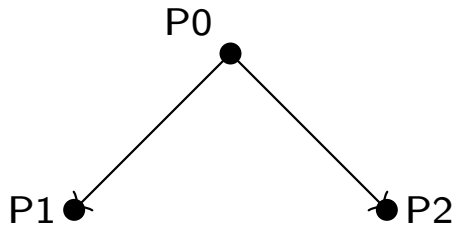
Moins intéressant que l'arbre binomial dans un modèle 1-port.

Communications collectives

Algorithme de diffusion : arbre de Fibonacci

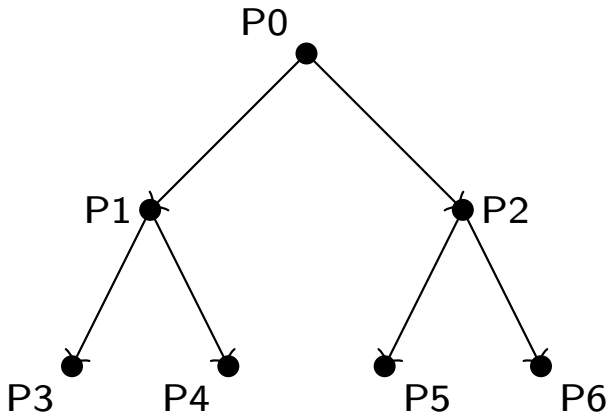
Communications collectives

Algorithme de diffusion : arbre de Fibonacci



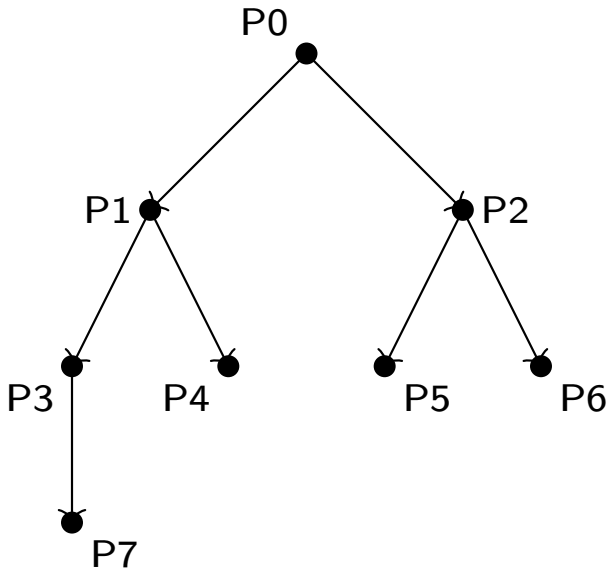
Communications collectives

Algorithme de diffusion : arbre de Fibonacci



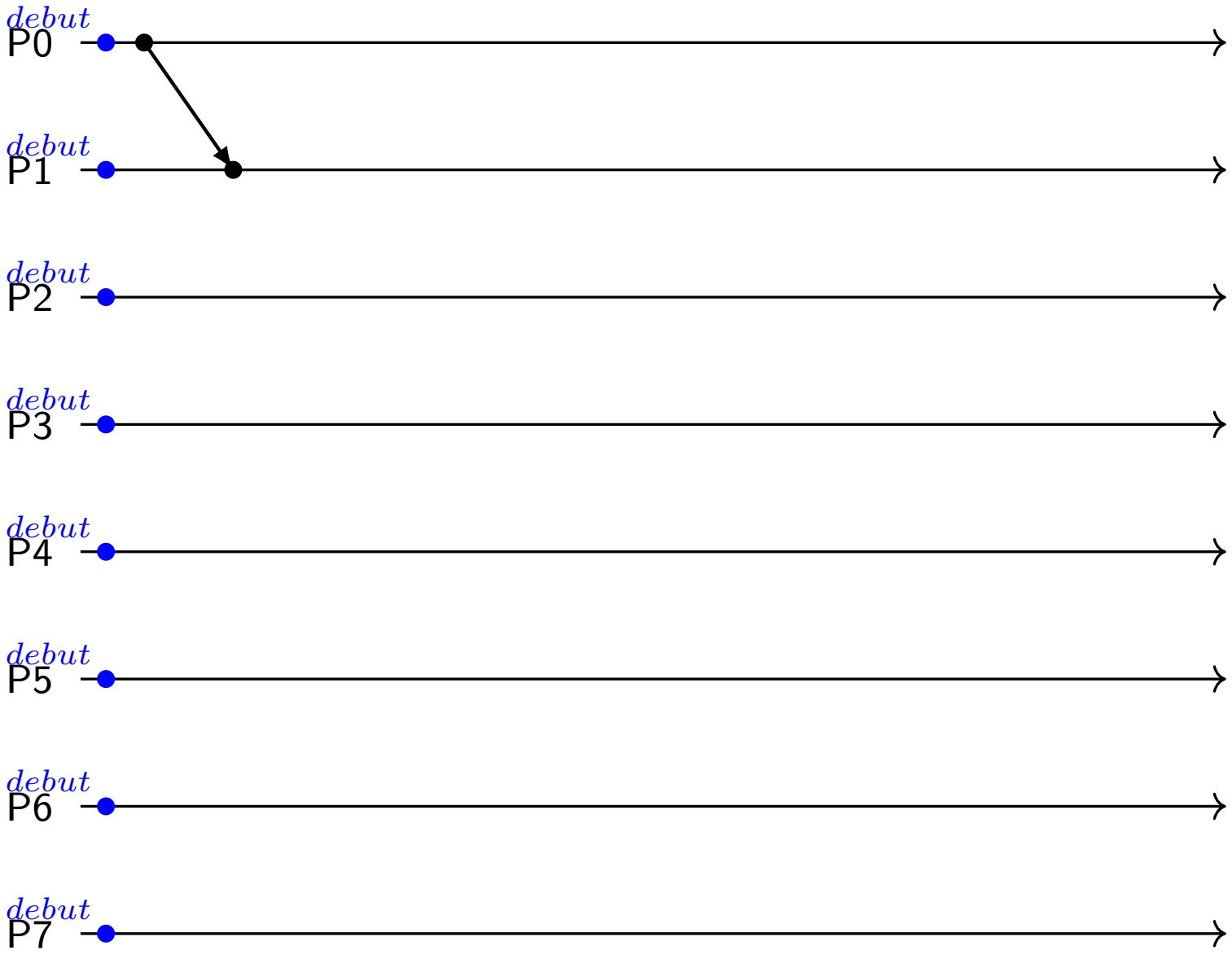
Communications collectives

Algorithme de diffusion : arbre de Fibonacci

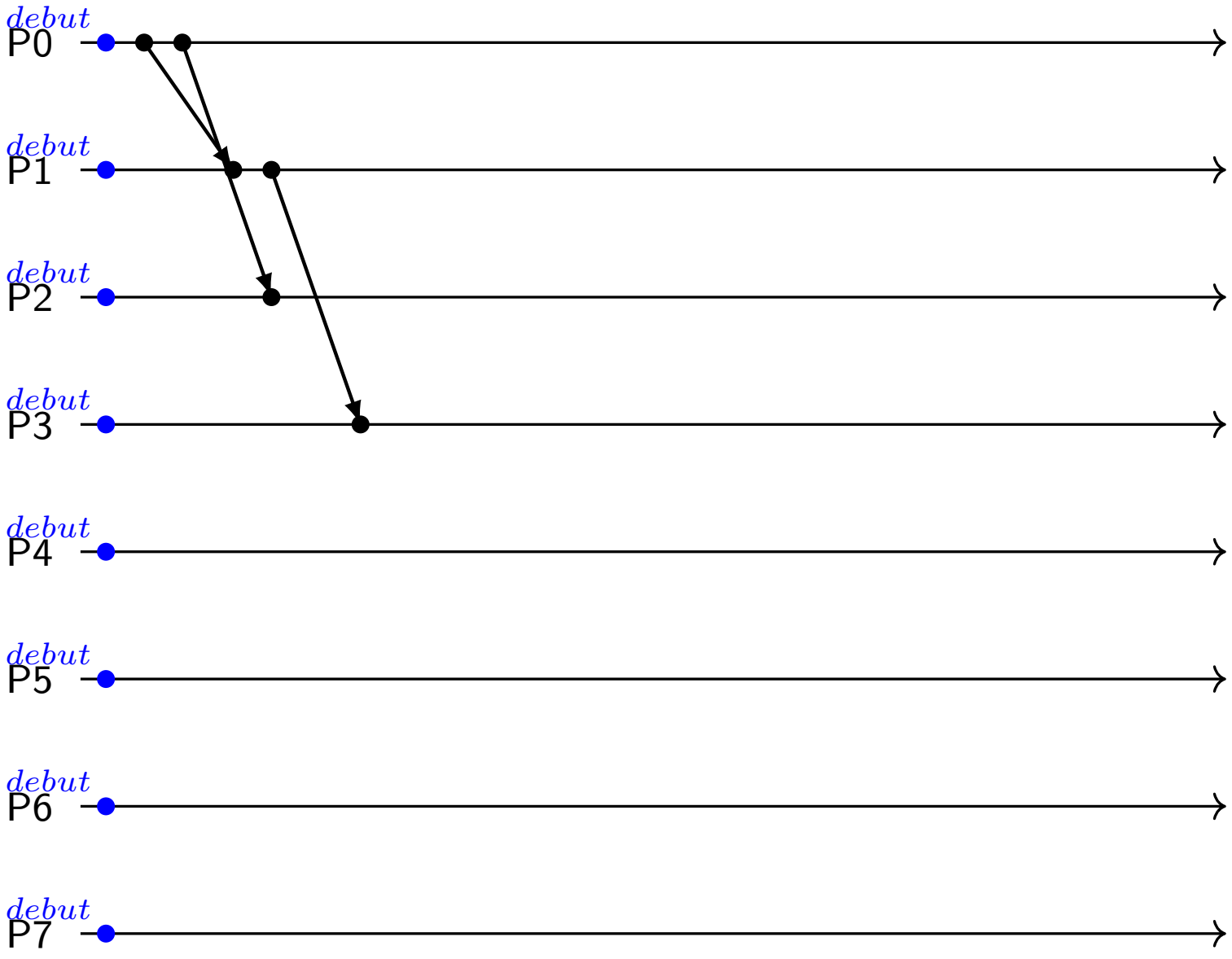


Communications collectives

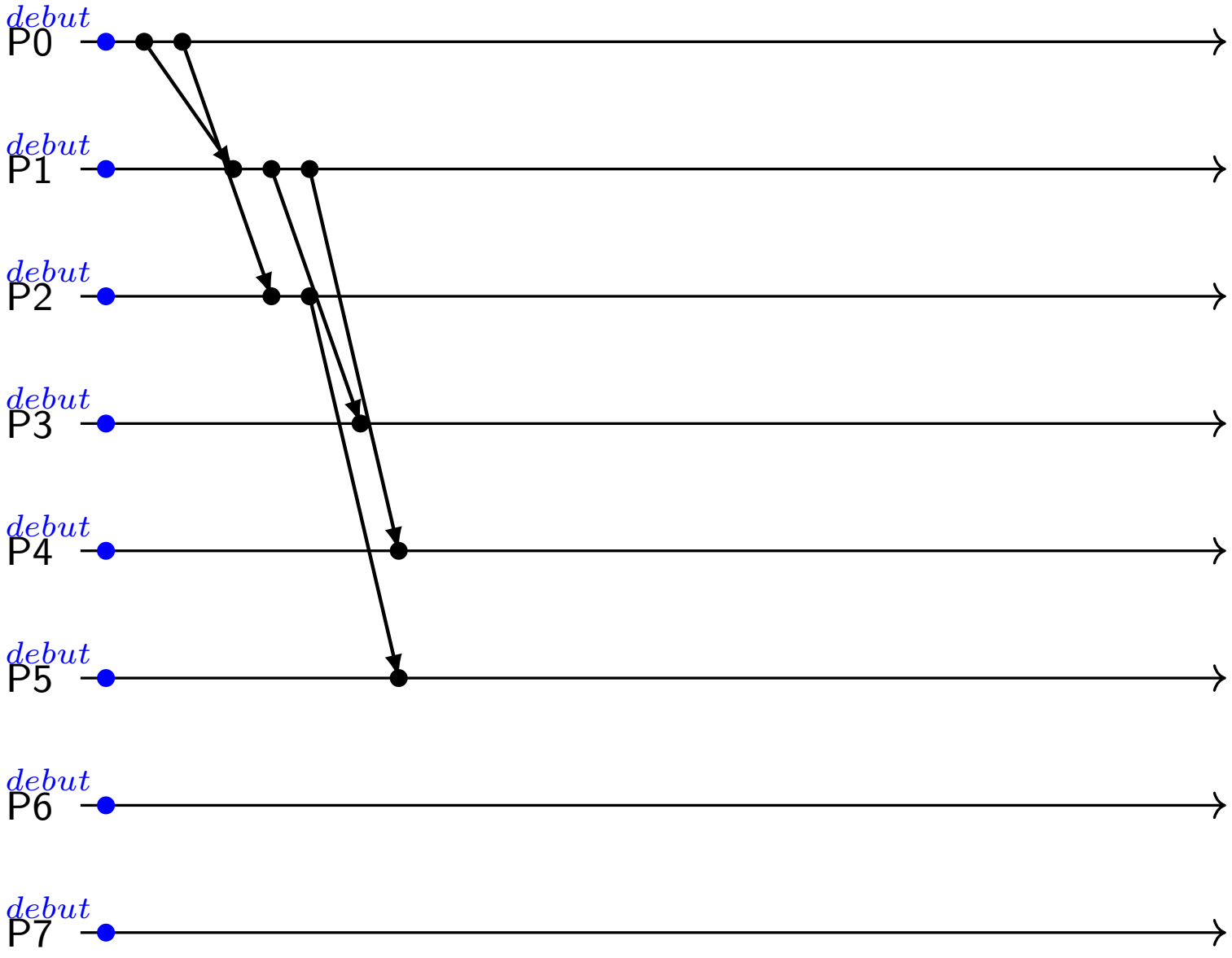
Algorithme de diffusion : arbre de Fibonacci



Algorithme de diffusion : arbre de Fibonacci

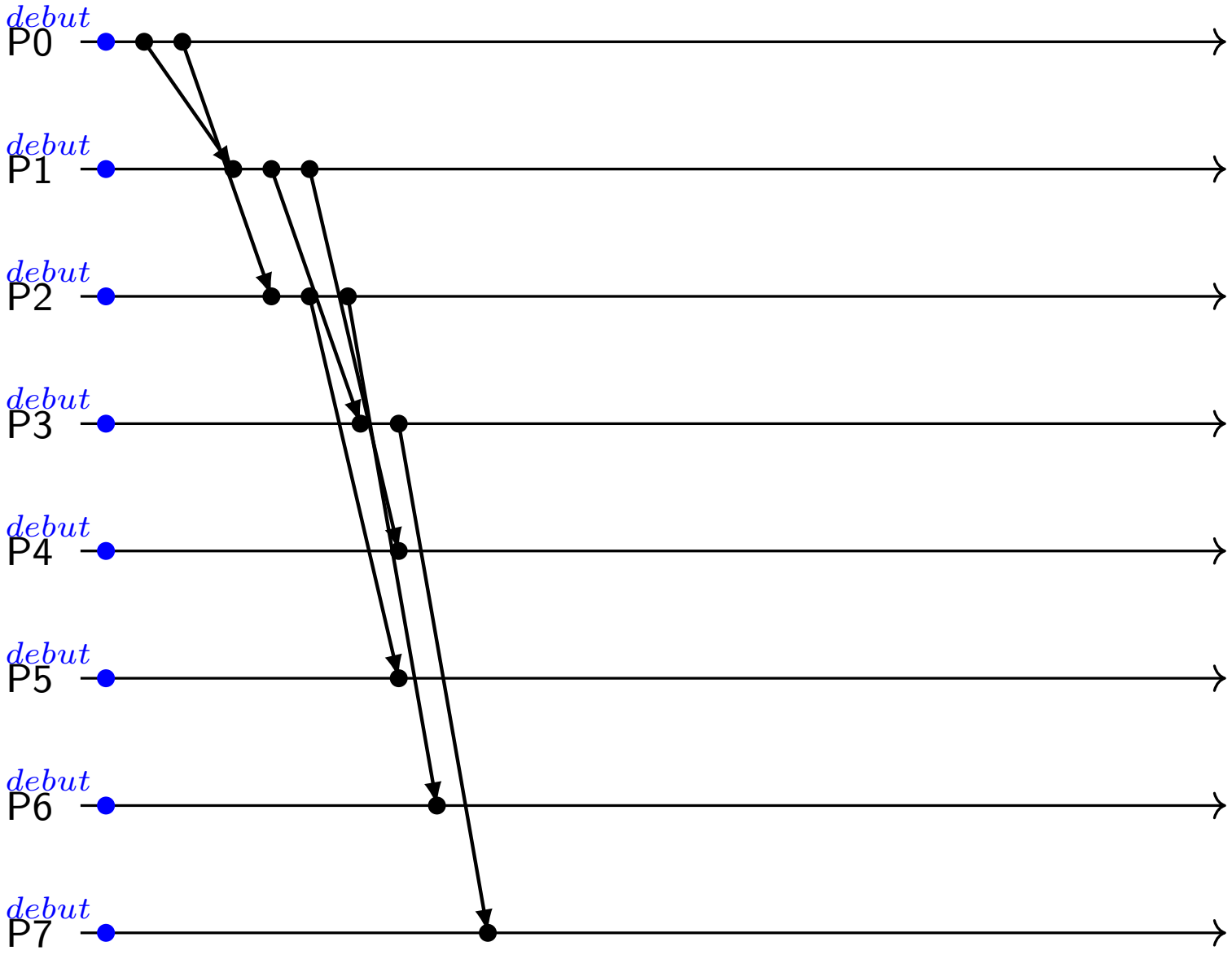


Algorithme de diffusion : arbre de Fibonacci



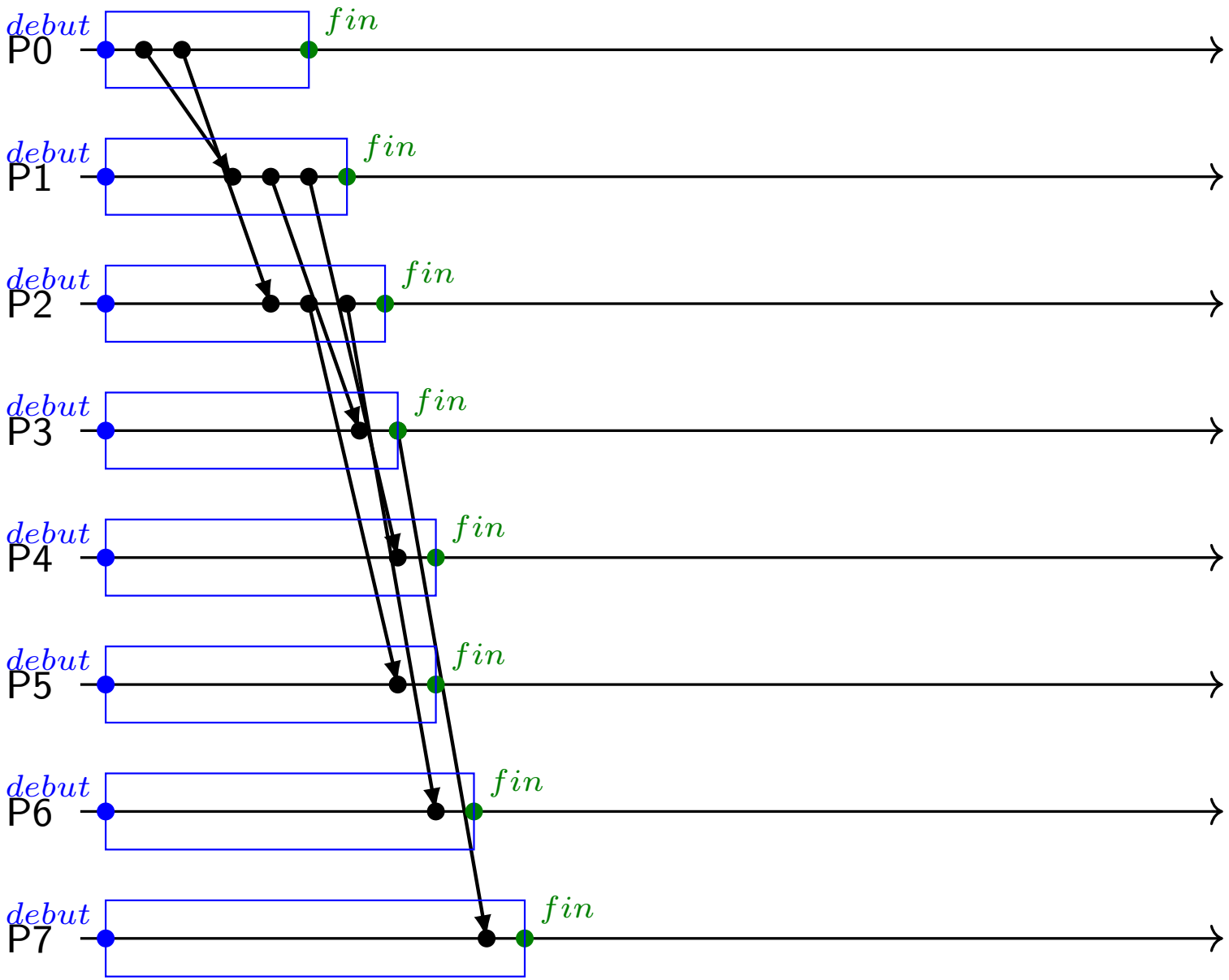
Communications collectives

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

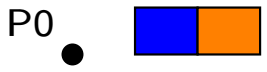
Complexités :

- 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 !

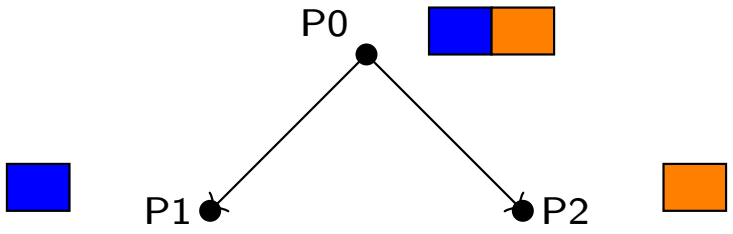
Communications collectives

Algorithme de diffusion : split-binary tree



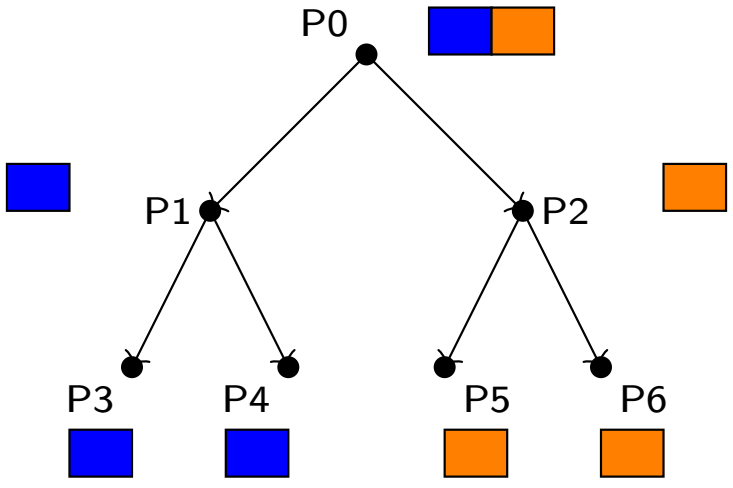
Communications collectives

Algorithme de diffusion : split-binary tree



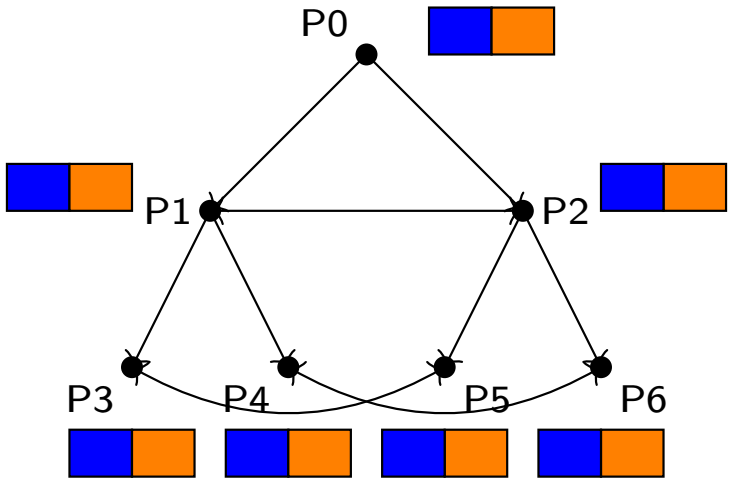
Communications collectives

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`

Communications collectives

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

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

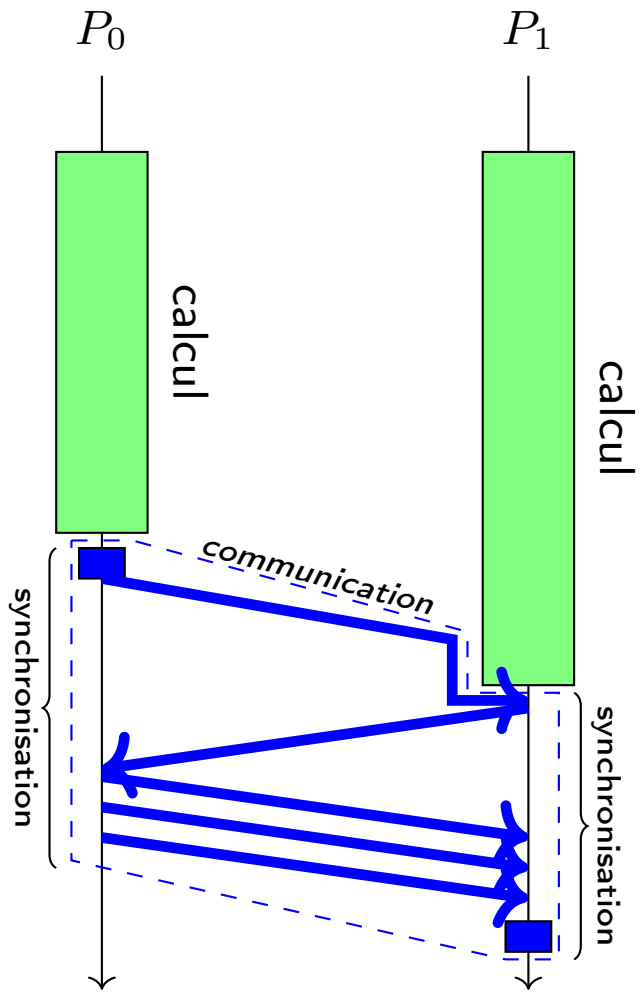

Permet de tirer parti de matériel spécialisé (DMA, coprocesseur, réseau rapide InfiniBand...)

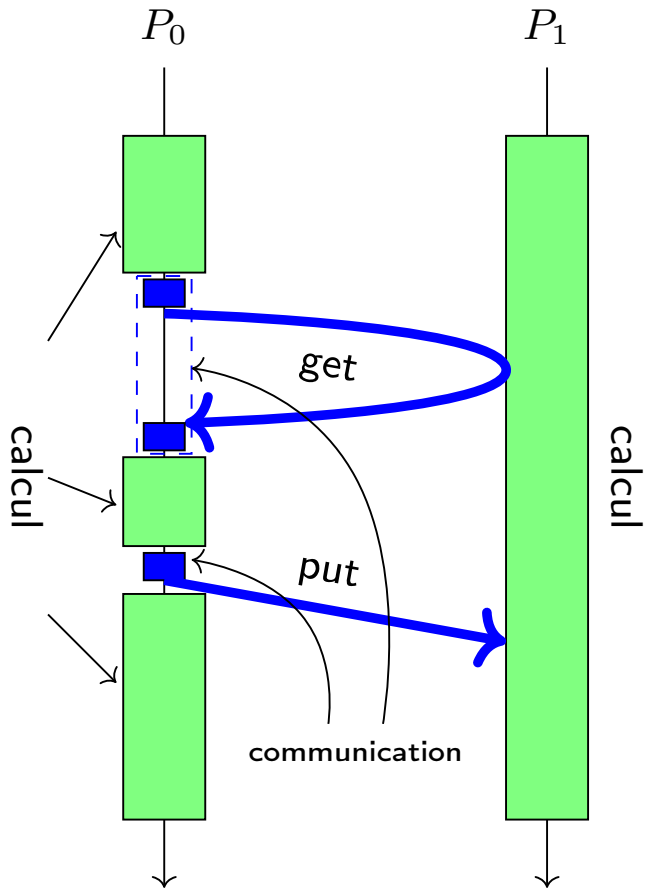

Plus efficace sur certains algorithmes (pas de synchronisation)


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


Moins performant sur certains matériels (non adaptés)

Unilatérales / bilatérales

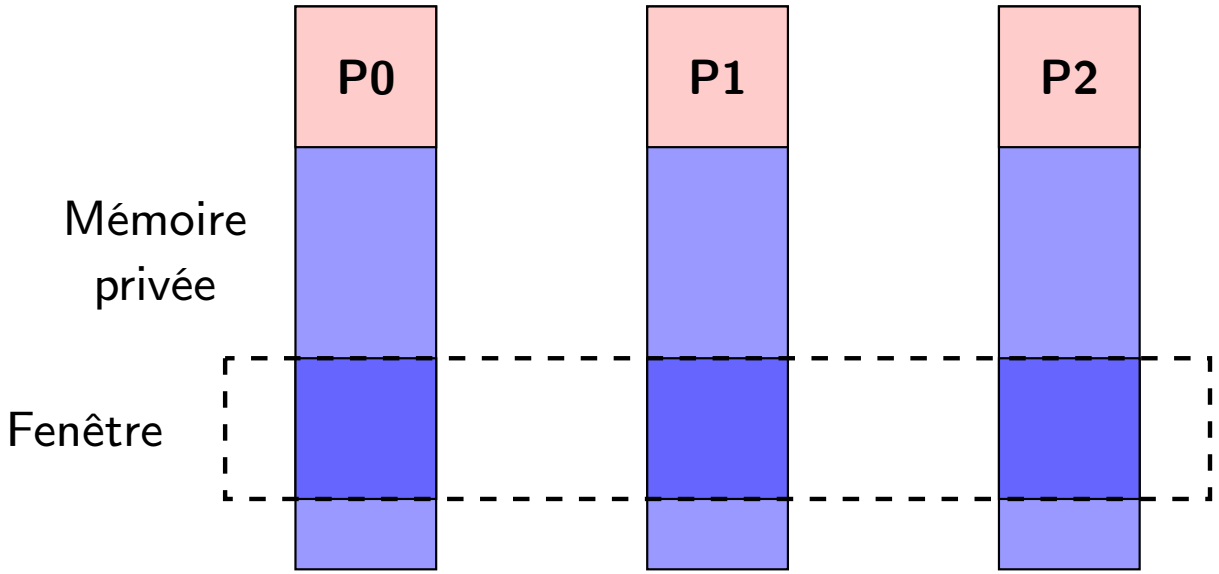




Mémoire accédée

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 );
```

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`.

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

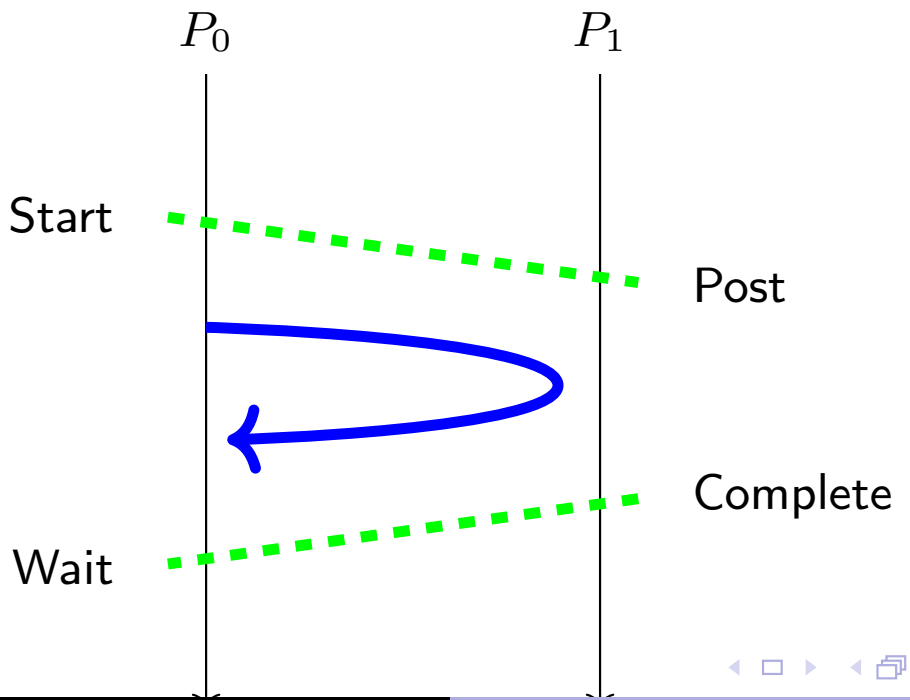
- `MPI_Win_fence` : synchronise toutes les communications RMA sur une fenêtre
- `MPI_Win_flush` : termine les opérations RMA sur le processus choisi
- `MPI_Win_flush_local` : termine les opérations RMA *locales* sur le processus choisi

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

Programmation multithreadée

Accès mémoire

Tous les threads ont accès à une mémoire commune

- Modèle PRAM

Techniques de programmation

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

Histoire d'OpenMP

Initialement : chaque constructeur avait son langage de programmation sur mémoire partagée

→ Problèmes de portabilité !

Volonté de standardisation : création de plusieurs standards implémentant différents paradigmes

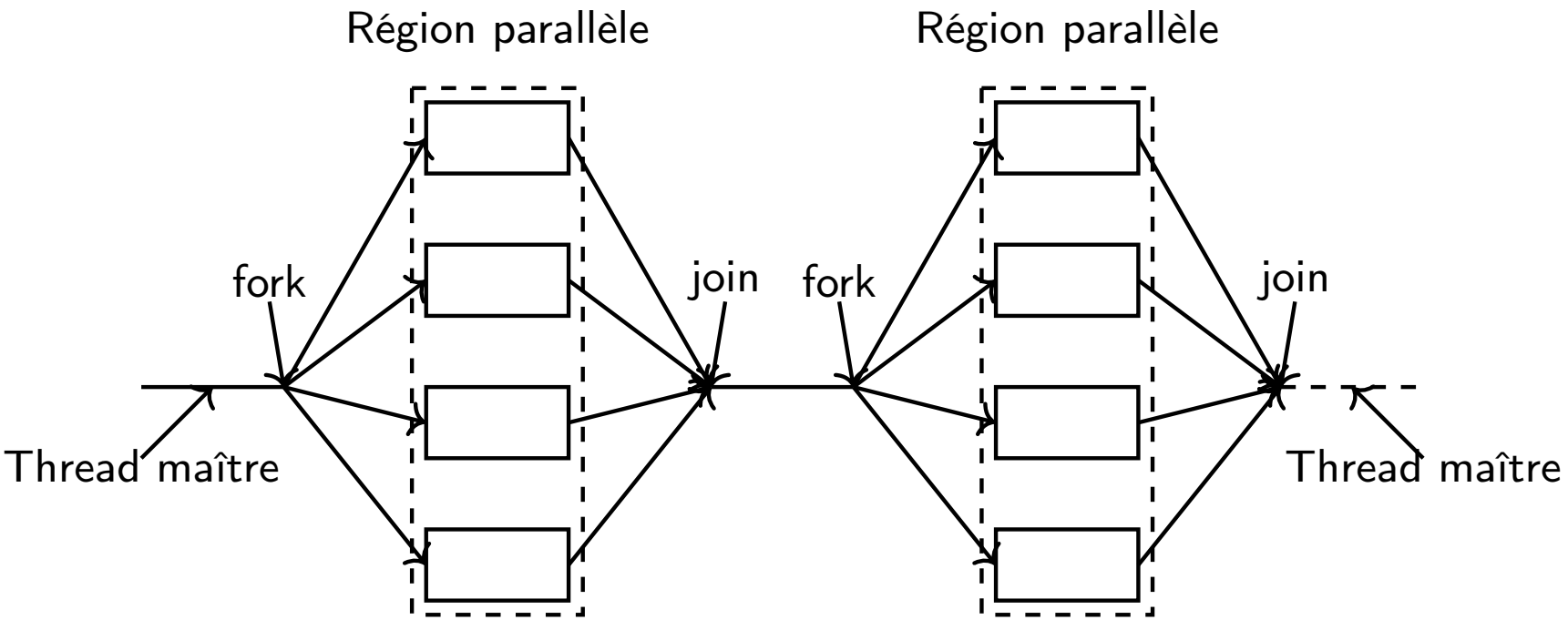
- SC 1991 : session Bird of a Feather proposant de standardiser un Fortran parallèle
- 1993 : publication de la norme HPF 1.0 (High Performance Fortran), basé sur Fortran 90
- 1994 : publication de Cilk (du MIT puis Intel), parallélisme par tâches
- 1997 : **OpenMP Fortran** 1.0 puis...
- 1998 : **OpenMP C/C++** 1.0 : parallélisation de boucles
- 2000 : OpenMP Fortran 2.0, 2002 OpenMP C/C++ 2.0
- 2008 : OpenMP 3.0, parallélisme par tâches, nouvelles politiques de placement de données
- 2013 : OpenMP 4.0, support d'accélérateurs, instructions atomiques, affinité...

OpenMP

Fonctionnement

Langage d'annotations

- Le code est relativement peu modifié
- Directives de compilation
  - Utilisation de `#pragma` : si le compilateur ne connait pas, OpenMP est débrayé et le code fonctionne en séquentiel



Qu'est-ce qu'OpenMP

Ensemble de **fonctions, variables d'environnement et directives de compilation**

- Programmation à haut niveau
- Forte implication du compilateur

Compilateur OpenMP

- Utilise les directives de compilation
- Chargé de générer les threads, partager le travail entre les threads, placer les données

Bibliothèque OpenMP

- Fournit un environnement d'exécution
- Chargé des fonctions dynamiques, à l'exécution

Variables d'environnement

- Permet à l'utilisateur de fixer des paramètres à l'exécution (nombre de threads...)
- Chargé de tout ce qui est spécifique à l'exécution : binding matériel, taille de la pile, etc

Exemple : parallélisation d'une boucle

Calcul d'un maximum global sur un tableau

Algorithm 1: Calcul séquentiel du maximum d'un tableau

begin
  Data: Un tableau de taille N d'entiers positifs tab[]
  Result: Un entier MAX
  $MAX = 0$;
  for $i \leftarrow 0$ **to** N **do**
    if $tab[i] > MAX$ **then** $MAX = tab[i]$;

Paréllélisation du calcul

Parallélisation de la boucle `for`

- On "tranche" l'intervalle sur lequel a lieu de calcul
- Les tranches sont réparties entre les threads

Annotations OpenMP

Sections parallèles

- `#pragma omp parallel` : début d'une section parallèle (fork)
- `#pragma omp for` : boucle for parallélisée

Synchronisations

- `#pragma omp critical` : section critique
- `#pragma omp barrier` : barrière de synchronisation

Visibilité des données

- Privée = visible uniquement de ce thread
- Partagée = partagée entre tous les threads
- Par défaut :
  - Ce qui est déclaré dans la section parallèle est privé
  - Ce qui est déclaré en-dehors est partagé

```
#pragma omp parallel private (tid) shared (result)
```

Compilation et exécution

En-têtes

```
#include <omp.h>
```

Compilation

Activation avec une option du compilateur

- Pour gcc : `-fopenmp`

Rappel : si l'option n'est pas activée, les annotations sont ignorées (pas les fonctions).

Exéuction

Nombre de threads :

- Par défaut : découverte du nombre de cœurs et utilisation de tous
- Fixé par l'utilisateur via la variable d'environnement `$OMP_NUM_THREADS`

Région parallèle OpenMP

On déclare la **région parallèle** avec

```
#pragma omp parallel
```

Chaque thread exécute ce qu'il y a dans le **bloc structuré**

- Attention, l'accolade ouvrante doit être en début de ligne
- Interdiction d'effectuer des branchements (par exemple goto vers l'intérieur ou l'extérieur d'une région parallèle

Hello World 0.1

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main(){
    printf("Hello from outside\n" );
#pragma omp parallel
    {
      printf("Hello World !\n" );
     }
    return EXIT_SUCCESS;
}
```

Portée des variables

Définition de la **portée des variables**

```
#pragma omp parallel private ( tid, numthreads )
#pragma omp parallel private ( a, b ) shared ( c, d )
```

Nombre de threads

- Par défaut : le nombre de cœurs du système
- Possibilité de le fixer
  - Variable d'environnement `OMP_NUM_THREADS`
  - Fonction `omp_set_num_threads()` (prioritaire)

Hello World 0.2

```
    /*...*/
    printf("Hello from outside\n" );
    omp_set_num_threads( 2 );
#pragma omp parallel
    {
        printf("Hello World !\n" );
    }
    omp_set_num_threads( 3 );
#pragma omp parallel
    {
        printf("Hello 2 !\n" );
    }
    /*...*/
```

Identification des threads

Système de nommage par rang :

- Numéro de thread : `omp_get_thread_num()`
- Nombre de threads dans le programme parallèle : `omp_get_num_threads()`

Hello World 1.0

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

int main(){
    int numthreads, tid;
#pragma omp parallel  private( tid, numthreads )
    {
        tid = omp_get_thread_num();
        numthreads = omp_get_num_threads();
        printf("Hello World from thread = %d / %d\n", tid, numthreads);
    }
    return EXIT_SUCCESS;
}
```

Une note sur les variables privées

Les variables privées ne **sont pas initialisées implicitement**

- Elles n'existent pas avant le début de la région parallèle

Exemple :

```
int  i = 10;
#pragma omp parallel  private(i)
{
    printf("thread %d: i = %d\n", omp_get_thread_num(), i);
     i = 1000 + omp_get_thread_num();
}
printf("i = %d\n", i);
```

Affichera :

```
thread 3: i = 0
thread 0: i = 0
thread 2: i = 0
thread 1: i = 32658
i = 10
```

Deux options d'interaction possibles entre la section parallèle et la partie séquentielle :

- `firstprivate` : initialisation avec la valeur que la variable avait **avant le début** de la section parallèle
- `lastprivate` : la valeur gardée à la fin du programme est celle qu'avait la variable **à la fin du dernier** thread à sortir de la partie parallèle

Utilisation d'une variable partagée

Hello World 2.0

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main(){
    int numthreads, tid;
#pragma omp parallel private( tid ) shared ( numthreads )
    {
        tid = omp_get_thread_num();
        printf("Hello World from thread = %d\n", tid);
        if( 0 == tid ) {
           numthreads = omp_get_num_threads();
        }
    }
    printf("Number of threads = %d\n", numthreads);
    return EXIT_SUCCESS;
}
```

Attention à l'exclusion mutuelle !

- Ici : un seul thread écrit dans la variable partagée et elle n'est lue qu'après la fin de l'exécution de tous les threads
- Plusieurs possibilités d'assurer l'exclusion mutuelle et l'ordre causal des opérations

Exclusion mutuelle

Plusieurs façons d'assurer l'exclusion mutuelle

- Définition de sections critiques : constructeur `critical`
- Verrous : `type omp_lock_t`
- Atomicité : constructeur `atomic`

Différences :

- Utilisabilité, syntaxe
- Restrictions : pouvoir en sortir, exceptions...
- Facilité d'utilisation et risques de bugs

Section critique

Protection d'une section critique :

```
#pragma omp critical (nom)
```

Optionnel : nom de la section critique (*global* au programme)

Sémantique : **un seul thread** accède à la section critique à la fois (bloquant).

Hello World 3.0

```
int main(){
    int numthreads, tid;
    numthreads = -1;
#pragma omp parallel private( tid ) shared (numthreads )
    {
        tid = omp_get_thread_num();
        printf("Hello World from thread = %d\n", tid);
#pragma omp critical ( nmthreads )
        {
            if( -1 == numthreads ){
                numthreads = omp_get_num_threads();
                printf("Number of threads = %d\n", numthreads);
            }
        } /* end critical */
    } /* end parallel */
    return EXIT_SUCCESS;
}
```

Verrous

5 fonctions de manipulation des verrous :

- `omp_init_lock()` : initialisation.
- `omp_destroy_lock()` : destruction (le verrou doit être libre)
- `omp_set_lock()` : prise du verrou. Bloquant : si le verrou est pris par un autre thread, attend jusqu'à ce qu'il soit libre.
- `omp_unset_lock()` : libère le verrou. Doit être appelé par le thread qui a pris le verrou.
- `omp_test_lock()` : tentative de prise du verrou. Retourne 0 si le verrou est déjà pris par un autre thread ; retourne 1 et prend le verrou si celui-ci est libre.

Verrous **imbriqués** (nested locks) : peuvent être pris *plusieurs fois* par le **même** thread

- `omp_init_nest_lock()`, `omp_destroy_nest_lock()`, ...

Verrous

Hello World 4.0

```
int main(){
    int numthreads, tid;
    omp_lock_t lock;
    numthreads = -1;
    omp_init_lock( &lock );
#pragma omp parallel private( tid )  shared (numthreads )
    {
        tid = omp_get_thread_num();
        printf("Hello World from thread = %d\n", tid);
        omp_set_lock( &lock );
         if( -1 == numthreads ) {
             numthreads = omp_get_num_threads();
             printf("Number of threads = %d\n", numthreads);
         }
        omp_unset_lock( &lock );
    }
    omp_destroy_lock( &lock );
    return EXIT_SUCCESS;
}
```

Atomicité

Opération atomique : **insécable**

- Conséquence : pas de problème si plusieurs threads concurrents

Constructeur :

```
#pragma omp atomic
```

Optionnel : le type d'opération

Quatre types d'opérations atomiques :

- Lecture (`read`)
- Écriture (`write`)
- Mise à jour (`update`) : par exemple `++x`, `x+= valeur`, `x «= valeur`,etc
- Capture : combine écriture et mise à jour, par exemple `var = x++`

Atomicité

Exemple : utilisation d'un compteur pour compter le nombre de threads

Hello World 5.0

```
int main(){
    int numthreads, tid;
    numthreads = 0;
#pragma omp parallel private( tid )  shared ( numthreads )
    {
        tid = omp_get_thread_num();
        printf("Hello World from thread = %d\n", tid);
#pragma omp atomic update
        numthreads++;
    }
    printf("Number of threads = %d\n", numthreads);
    return EXIT_SUCCESS;
}
```

Besoin de synchronisation entre les threads

Synchronisation des threads

- Permet de savoir que tous les threads ont atteint un point

Sémantique : *on ne sort de la barrière qu'une fois que tous les threads y sont entrés*

```
#pragma omp parallel
```

Attention aux barrières implicites ajoutées par le compilateur

- Fin d'une région parallèle
- Fin d'une boucle
- Fin d'une région `single`...

Barrière de synchronisation

Permet par exemple de s'assurer que personne ne va au-delà d'un certain point du programme avant les autres threads.

- Ici : personne ne va utiliser `numthreads` avant qu'elle ait été affectée.

Hello World 6.0

```
int main() {
    int numthreads, tid;
#pragma omp parallel private ( tid ) shared ( numthreads )
    {
        tid = omp_get_thread_num();
        if( 0 == tid )
            numthreads = omp_get_num_threads();
#pragma omp barrier
        printf("Hello World from thread = %d / %d\n", tid, numthreads);
    }
    return EXIT_SUCCESS;
}
```

Exécution par un seul thread

Possibilité de faire exécuter un bloc par un seul thread

- Sans préciser lequel : constructeur `single`
- Par le thread original (la plupart du temps le thread 0) : constructeur `master`

Hello World 7.0

```
int main() {
    int numthreads, tid;
#pragma omp parallel private ( tid ) shared ( numthreads )
    {
        tid = omp_get_thread_num();
#pragma omp single
        numthreads = omp_get_num_threads();
        printf("Hello World from thread = %d \n", tid );
    }
    printf("Number of threads = %d\n", numthreads);
    return EXIT_SUCCESS;
}
```

Exécution ordonnée

Permet de séquentialiser du code

- Un seul thread est dans la section ordonnée
- Les threads entrent dans l'ordre des itérations de boucle

Hello World 8.0

```
int main() {
    int numthreads, tid, i;
#pragma omp parallel private ( tid ) shared ( numthreads )
    {
        tid = omp_get_thread_num();
#pragma omp parallel private ( numthreads, tid )
    {
        numthreads = omp_get_num_threads();
        tid = omp_get_thread_num();
#pragma omp for ordered schedule( static, 5 )
        for( i = 0 ; i < numthreads; i++ ) {
#pragma omp ordered
            printf("Hello World from thread = %d \n", tid );
        }
    }
        return EXIT_SUCCESS;
}
```

Parallélisation d'une boucle

À l'origine : OpenMP utilisé pour paralléliser des boucles

- Éclater la boucle entre plusieurs threads
- Itérations indépendantes les unes des autres !

S'utilise **dans une section parallèle**

- Construction : `omp for`
- La boucle doit venir juste après

Carrés des entiers

```
int main() {
    int carre, i;
#pragma omp parallel private ( carre )
    {
#pragma omp for
        for( i = 0 ; i < NUM; i++ ) {
            carre = i * i;
            printf("carre = %d \n", carre );
        }
    }
    return EXIT_SUCCESS;
}
```

Synchronisation d'une boucle

Barrière implicite insérée à la fin d'une boucle

- Pour l'éviter : directive `nowait`
- Fonctionne à la fin des constructions `sections`, `for` et `single`
- Pas à la fin d'une boucle `ordered`

Carrés des entiers nowait

```
int main() {
    int carre, i;
#pragma omp parallel private ( carre )
    {
#pragma omp for nowait
        for( i = 0 ; i < NUM; i++ ) {
            carre = i * i;
            printf("carre = %d \n", carre );
        }
        printf( "Done\n" );
    }
    return EXIT_SUCCESS;
}
```

Exemple : Calcul d'un maximum global

Approche naïve

On peut parallélise la boucle et écrire le résultat directement dans une variable partagée

Algorithme

max = 0
parfor i = 0 à N-1 :
  si max $<$ tab$[i]$: alors max = $tab[i]$

Problème : les accès à max doivent se faire dans une section critique

- Solution : utilisation de `#pragma omp critical`
- Séquentialisation des accès $\rightarrow$ séquentialisation de la boucle !

Meilleure approche

Calcul d'un maximum local puis à la fin du calcul, maximum global des maximums locaux

Options de découpage des boucles (scheduling)

Static

- Le découpage est fait à l'avance
- Des tranches de tailles égales sont attribuées aux threads
- Adapté aux boucles dont les itérations ont des temps de calcul équivalent

Dynamic

- Le découpage est fait à l'exécution
- Le scheduler attribue une tranche aux threads libres
- Attention à la taille des tranches : si trop petites, seul le thread 0 travaillera

Guided

- Similaire à dynamic
- Les tailles des tranches diminuent durant l'exécution

Utilisation

```
#pragma omp for schedule (type, taille)
```

Réduction

Effectue une opération **globale** sur une **variable partagée**

- Atomique (mais pas besoin de le préciser)
- Autre méthode de communication entre threads

Pas de problème de concurrence... la bibliothèque s'en occupe.

On précise l' **opération**

- $+, -, |, ^, ||$
- $*, \&\&$
- $\&$
- min, max

Syntaxe :

```
reduction(operator:list)
```

Réduction

Exemple : calcul de $n!$

```
   int facto( int nb ) {
     int fac = 1;
     int i;
#pragma omp parallel for reduction( *:fac )
     for( n = 2 ; n <= nb ; n++ ) {
       fac *= n;
     }
     return fac;
   }
```

Si on l'avait fait sans réduction :

```
 int facto( int nb ){
   int fac = 1;
   int i;
#pragma omp parallel for
   for( n=2 ; n <= nb ; n++ ) {
#pragma omp atomic
     fac *= n;
   }
   return fac;
 }
```

Déclaration de réduction

Possibilité de généraliser l'opération

```
#pragma omp declare reduction( name:type:expression )
```

Optionnel : `initializer( expression )`

On déclare l'expression à utiliser dans la réduction :

```
struct BestInfo { unsigned size, param1, param2; };

#pragma omp declare reduction( isbetter:BestInfo: \
              omp_in.size<omp_out.size ? omp_out=omp_in : omp_out )\
              initializer(omp_priv = BestInfo{~0u,~0u,~0u})
```

Puis on l'utilise :

```
#pragma omp parallel for reduction(isbetter:result)
```

Sections parallèles

Section parallèle :

- Une section est exécutée **par exactement un thread**
- On peut définir plusieurs sections parallèles

```
#omp parallel
    {
#pragma omp sections
      {
#pragma omp section
         {
           fonction1();
         }
#pragma omp section
         {
            fonction2();
         }
      }
    }
```

Les deux sections parallèles sont exécutées

- Chacune par un thread
- Potentiellement en parallèle

Parallélisme imbriqué

On peut utiliser plusieurs niveaux de parallélisme

- Régions parallèles **imbriquées**
- Niveau d'imbrication arbitraire

S'active avec :

- Variable d'environnement `OMP_NESTED`
- Ou fonction `omp_set_nested( 1 )`

Applications :

- Boucles imbriquées
- Régions parallèles imbriquées
- Etc...

Nombre de threads : `num_threads( nb )`

```
#pragma omp parallel num_threads ( 2 )
{
#pragma omp parallel num_threads ( 2 )
     printf ( "Hello, world\n" ) ;
}
```

Remarques sur le parallélisme imbriqué

Fonction `omp_set_dynamic( nb )` : spécifie si le nombre de threads dans la prochaine région parallèle peut être **décidé à l'exécution**

- Si 0 : ne pas ajuster dynamiquement
- Si autre chose que 0 : ajuster dynamiquement, au max la valeur fixée par `omp_set_num_threads()` ou `OMP_NUM_THREADS`

Équivalent à la variable d'environnement `OMP_DYNAMIC` (mais la fonction est prioritaire).

- Possibilité de lire la valeur utilisée avec `omp_get_dynamic( )`

Attention aux **performances**

- Bien fixer le nombre de threads !

La clause `collapse` : paralléliser **plusieurs** boucles `for`

```
#pragma omp parallel for collapse(2)
 for(int y=0; y<25; ++y)
   for(int x=0; x<80; ++x) {
     tick(x,y);
   }
```

Remarques sur le parallélisme imbriqué

Fonction `omp_set_dynamic( nb )` : spécifie si le nombre de threads dans la prochaine région parallèle peut être **décidé à l'exécution**

- Si 0 : ne pas ajuster dynamiquement
- Si autre chose que 0 : ajuster dynamiquement, au max la valeur fixée par `omp_set_num_threads()` ou `OMP_NUM_THREADS`

Équivalent à la variable d'environnement `OMP_DYNAMIC` (mais la fonction est prioritaire).

- Possibilité de lire la valeur utilisée avec `omp_get_dynamic( )`

Attention aux **performances**

La clause `collapse` : paralléliser **plusieurs** boucles `for`

```
 int facto( int nb ){
   int fac = 1;
   int i;
#pragma omp parallel for
   for( n=2 ; n <= nb ; n++ ) {
#pragma omp atomic
     fac *= n;
   }
   return fac;
 }
```

Déclaration d'une *team*

Une section parallèle fait appel à une **team de threads**

- Possibilité de déclarer une **league de teams**

Chaque *league* a un **maître** (master)

- Rappelez-vous la construction `master`

Le maître de chaque team exécute la suite

- Création d'une région parallèle = exécutée par la team

S'utilise avec le constructeur `target`

```
int main(void) {
#pragma omp target teams
  {
    printf("test\n");
  }
  return 0;
}
```

Clauses d'une team

Optionnel : clauses

- `shared( list )` : les variables de la liste sont partagées entre les threads de la team
- `num_teams( exp )` : donne un nombre maximum de teams à créer
  - `omp_get_num_teams()` : nombre de teams existantes
- `thread_limit( nb )` : nombre maximum de threads par team
- et d'autres...

```
int main(){
    int tid, teamid;
#pragma omp target teams num_teams( 2 ) thread_limit(2)
    {
#pragma omp parallel private( tid, teamid )
        {
            teamid = omp_get_team_num() ;
            tid = omp_get_thread_num();
            if( teamid == 0){
                printf( "Hello from %d team %d\n", tid, teamid );
            }
            if( teamid == 1){
                printf( "Hello from %d team %d\n", tid, teamid );
            }
        }
    }
    return EXIT_SUCCESS;
}
```

SIMD

SIMD : Single Instruction Multiple Data

- Le processeur exécute la même instruction sur plusieurs données en même temps
- Vectorisation du calcul

```
#pragma omp simd
 for(int n=0; n<8; ++n) an += bn;
```

Possibilité de passer une clause :

- Collapse, reduction....

Parallélisme par tâches

Utilisation de **tâches** :

- Calculs indépendants les uns des autres
- Une tâche est exécutée par un thread
- Si pas de thread disponible : on attend et on exécute plus tard

On lance des tâches, qui sont exécutées par les threads du pool

- Calculs indépendants
- Possibilité de récursivité

Et on attend à la fin

```
#pragma omp task
```

Synchronisation : attente de la fin des tâches

```
#pragma omp taskwait
```

Exemple : Fibonacci

Attention : performances lamentables de cet exemple

- Très peu de calcul, beaucoup d'interactions

```
int  fib(int n){
  int i, j;
  if ( n < 2 ) return n;
  else {
#pragma omp task shared(i) firstprivate(n)
       i = fib(n-1);
#pragma omp task shared(j) firstprivate(n)
       j = fib(n-2);
#pragma omp taskwait
       return i+j;
    }
}
```

Appel :

```
#pragma omp parallel shared(n)
  {
#pragma omp single
    printf ("fib(%d) = %d\n", n, fib(n));
  }
```

Utilisation de tâches

Création de tâches = affectation à un thread

- Environnement de données : ça dépend
- À préciser avec des clauses : private firstprivate shared...

Ordonnancement des tâches :

- Utilisation de `taskyield`

```
#pragma omp taskyield
```

- Suspend la tâche courante pour donner la priorité à une autre tâche
- Utilisé pour éviter certains deadlocks, ou optimisation

```
void foo(omp_lock_t * lock, int n){
  for(int i = 0; i < n; i++)
#pragma omp task
  {
    something_useful();
    while( !omp_test_lock(lock) ) {
#pragma omp taskyield
     }
     something_critical();
     omp_unset_lock(lock);
   }
}
```

Dépendances de données

Variables partagées entre les tâches

- Habituels `shared`, `firstprivate`, `lastprivate`
- Attention aux variables partagées
  - Sections critiques, etc
- Utilisation de la valeur de retour !

Possibilité de définir des **dépendances de données** entre les threads

```
 for (int i = 0; i < T; ++i) {
#pragma omp task shared(x, ...) depend( out: x) // T1
    foo(...);
#pragma omp task shared(x, ...) depend( in: x) // T2
    bar(...);
#pragma omp task shared(x, ...) depend( in: x) // T3
    toto(...);
}
```

- T1 $\prec$ T2, T3
- T2 // T3

À partir de ces infos, l'environnement d'exécution **construit un DAG** et **ordonnance** sur les threads.

Types de tâches

final

- Les sous-tâches créées ensuite ne seront pas exécutées en parallèle
- Évite d'exploser le parallélisme dans des algos récursifs (perfs)

```
#pragma omp task final(...)
```

mergeable

- Fusionne l'environnement de la tâche avec la région appelante

```
#pragma omp task mergeable
```

Migration des tâches

Tâches filles d'une tâche : **pas de comportement défini**

- Peuvent être exécutées en parallèle de la tâche mère
- Peuvent être exécutées sur le même thread
- Pas précisé dans la spécification, dépend de l'implémentation

Par défaut, chaque tâche est **attachée à un thread**

- C'est toujours ce thread qui exécute la tâche

Attention aux directives qui **suspendent** une tâche

- `taskwait`, `barrier`...

Un thread qui exécute une tâche suspendue ne peut passer à une autre tâche que si il s'agit d'une **descendante de la tâche suspendue**
Solution : **détacher les tâches**

```
#pragma omp task untied
```

Attention, dangereux

- `omp_get_thread_num()` pas fiable
- Attention aux sections critiques et aux verrous

Plan du cours