Systèmes Parallèles et Distribués

4. Communications collectives

Franck Butelle Camille Coti

LIPN, Université Paris 13
Formation Ingénieurs SupGalilée Info 3

01/2012

Plan

Plan

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 !

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 !

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)

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 !

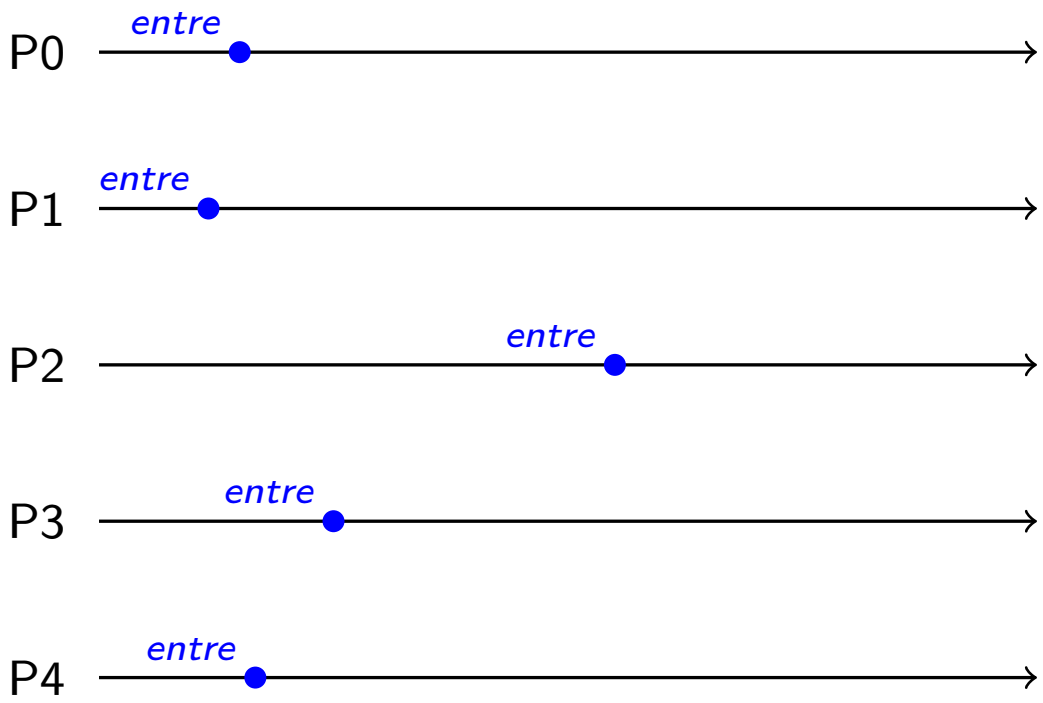
Un processus sort de la communication collective une fois qu'il a terminé **sa participation** à la collective

- 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)

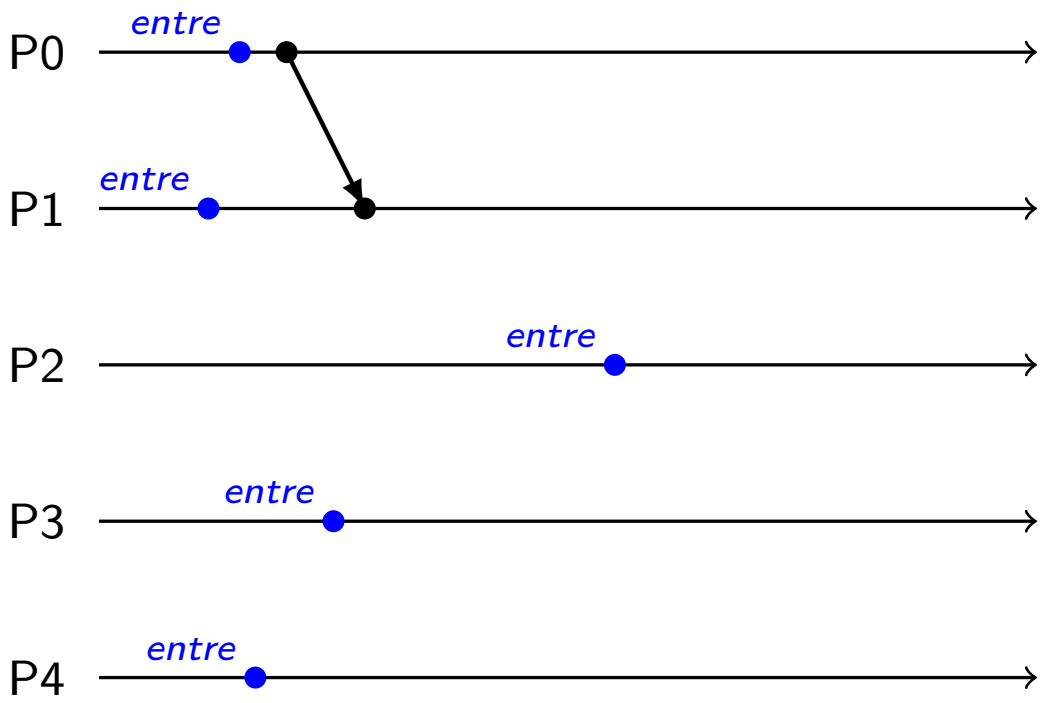
Bloquant (pour le moment)

- 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

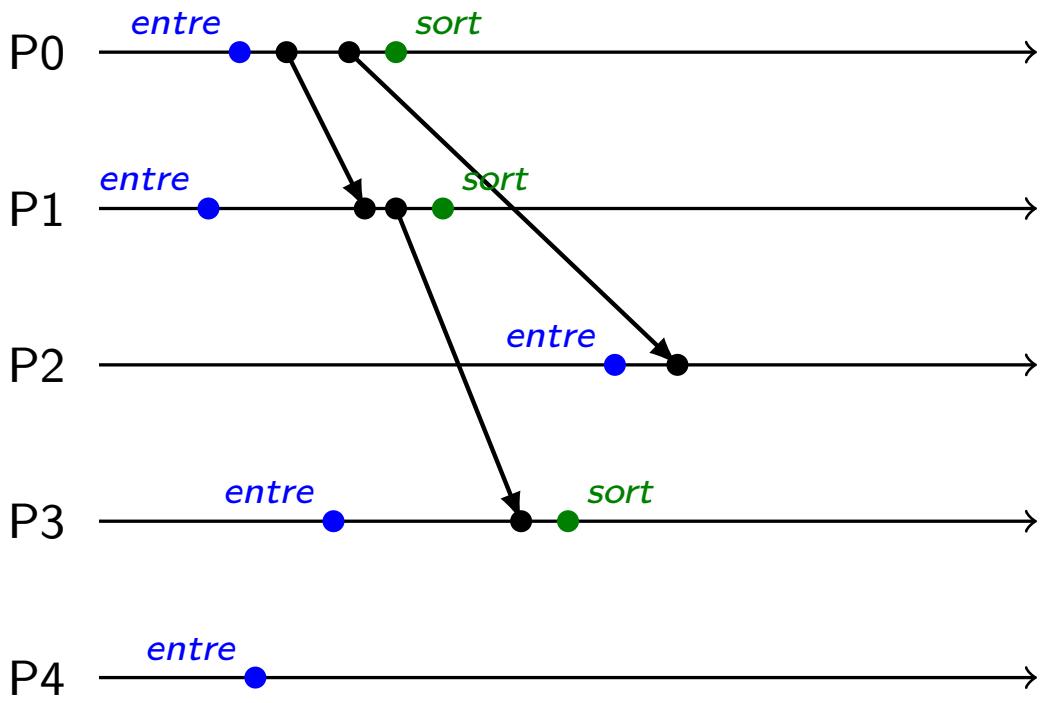
Sémantique des communications collectives



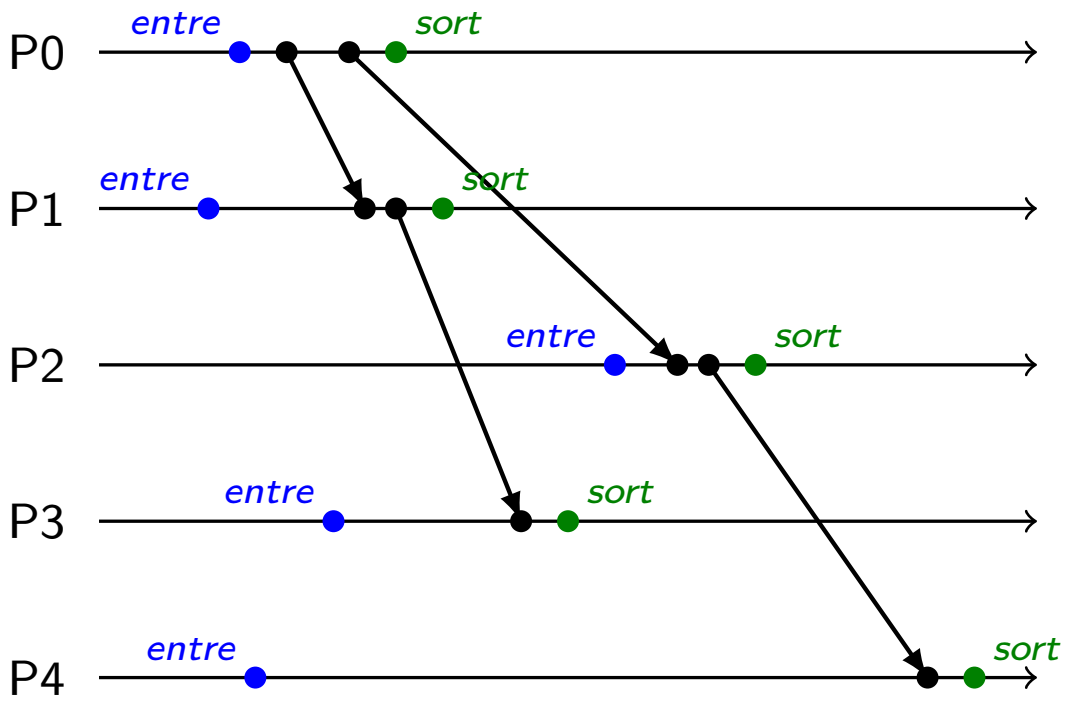
Sémantique des communications collectives



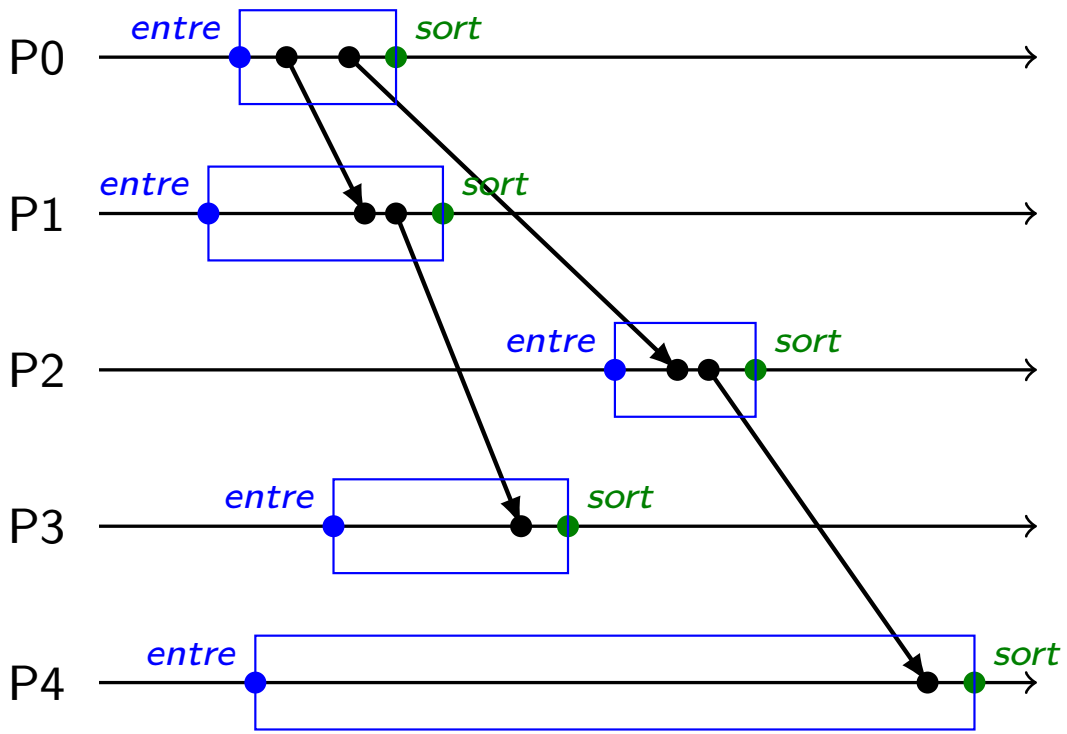
Sémantique des communications collectives



Sémantique des communications collectives



Sémantique des communications collectives



Exemple de communication collective : diffusion avec MPI

Diffusion avec MPI : `MPI_Bcast`

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

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

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

  MPI_Init( &argc, &argv );

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

  MPI_Finalize();

  return EXIT_SUCCESS;
}
```

Exemple de communication collective : diffusion avec MPI

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

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

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

Tous les processus du communicateur concerné doivent appeler `MPI_Bcast`

- Sinon : deadlock

Plan

Modèle pour les communications

Modèle pour les communications point-à-point :

$$T_{comm} = \lambda + \frac{s}{\beta} \tag{1}$$

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

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 ?

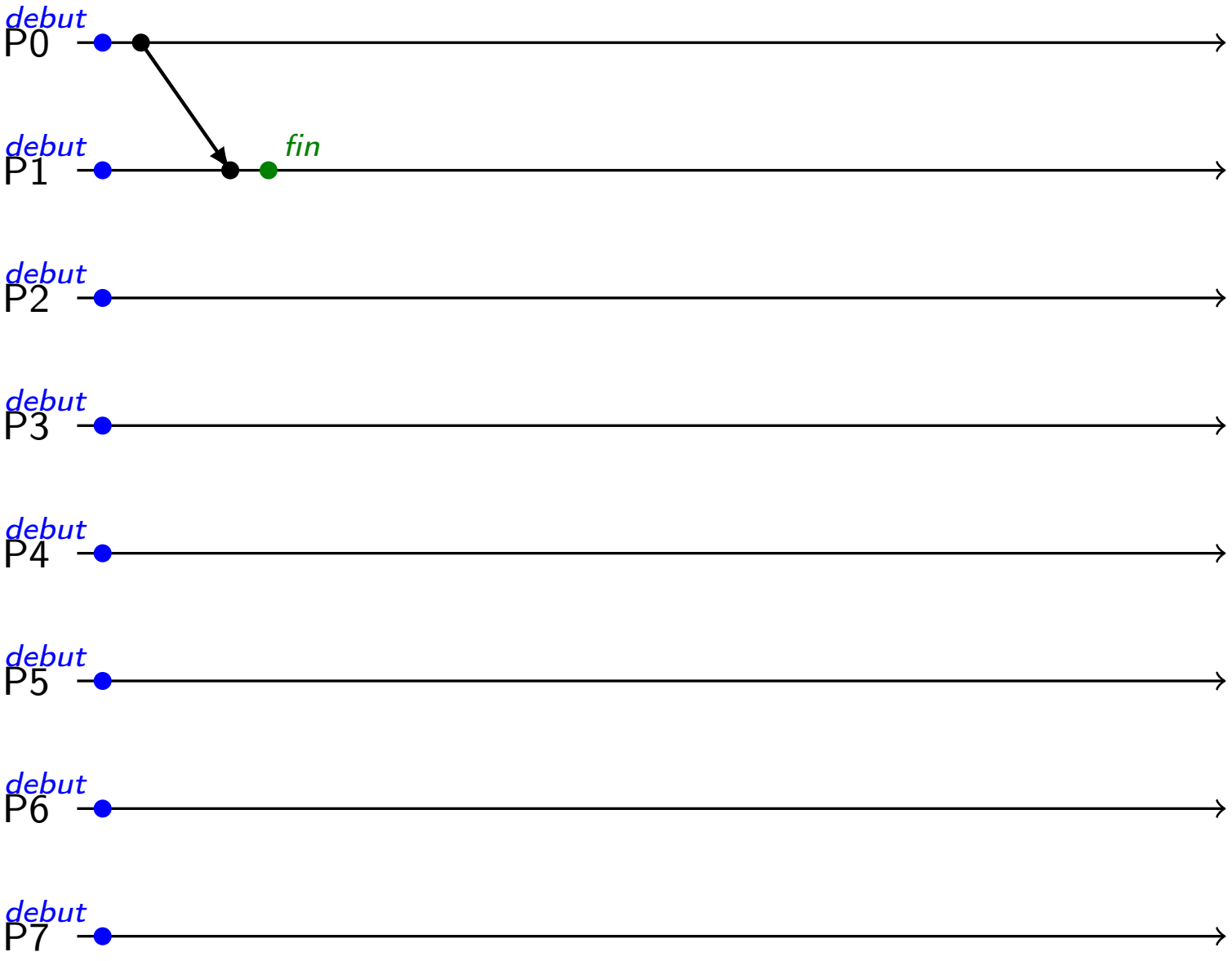
- Nombre de messages envoyés / reçus (latence)
- Utilisation de la bande passante

Paramètres :

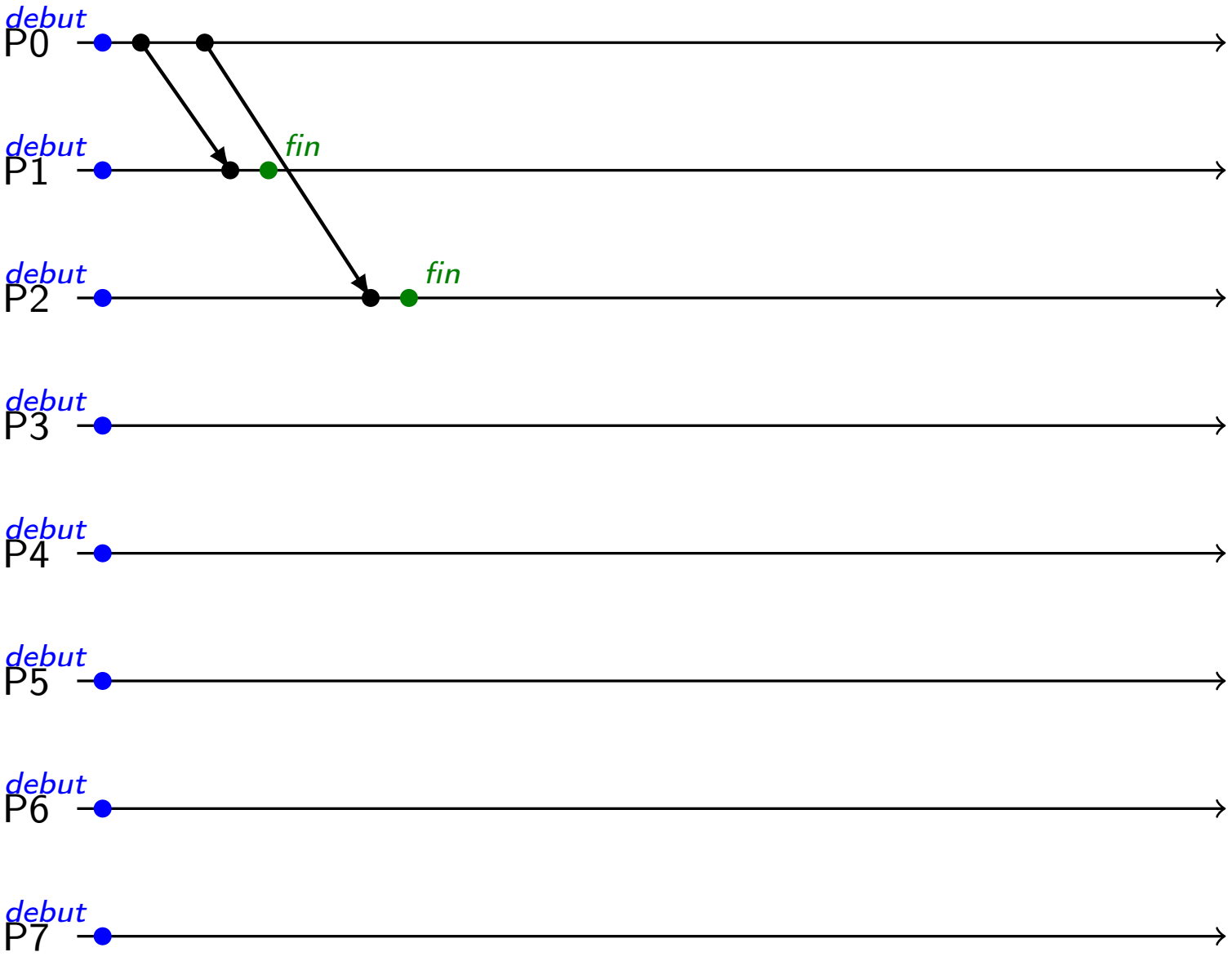
- Taille du message
- Nombre de processus impliqués !

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...

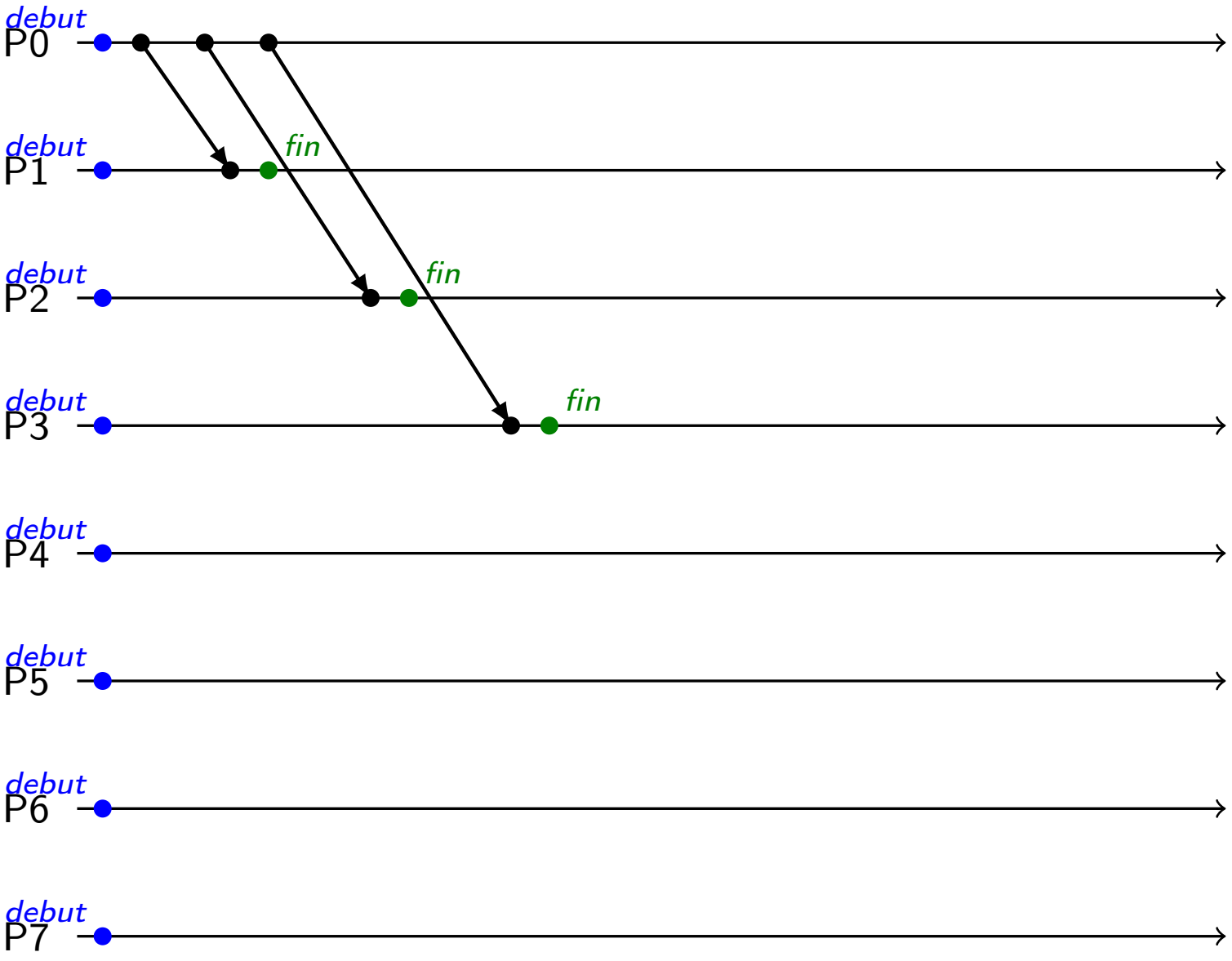
Modèle pour les communications : Bcast étoile



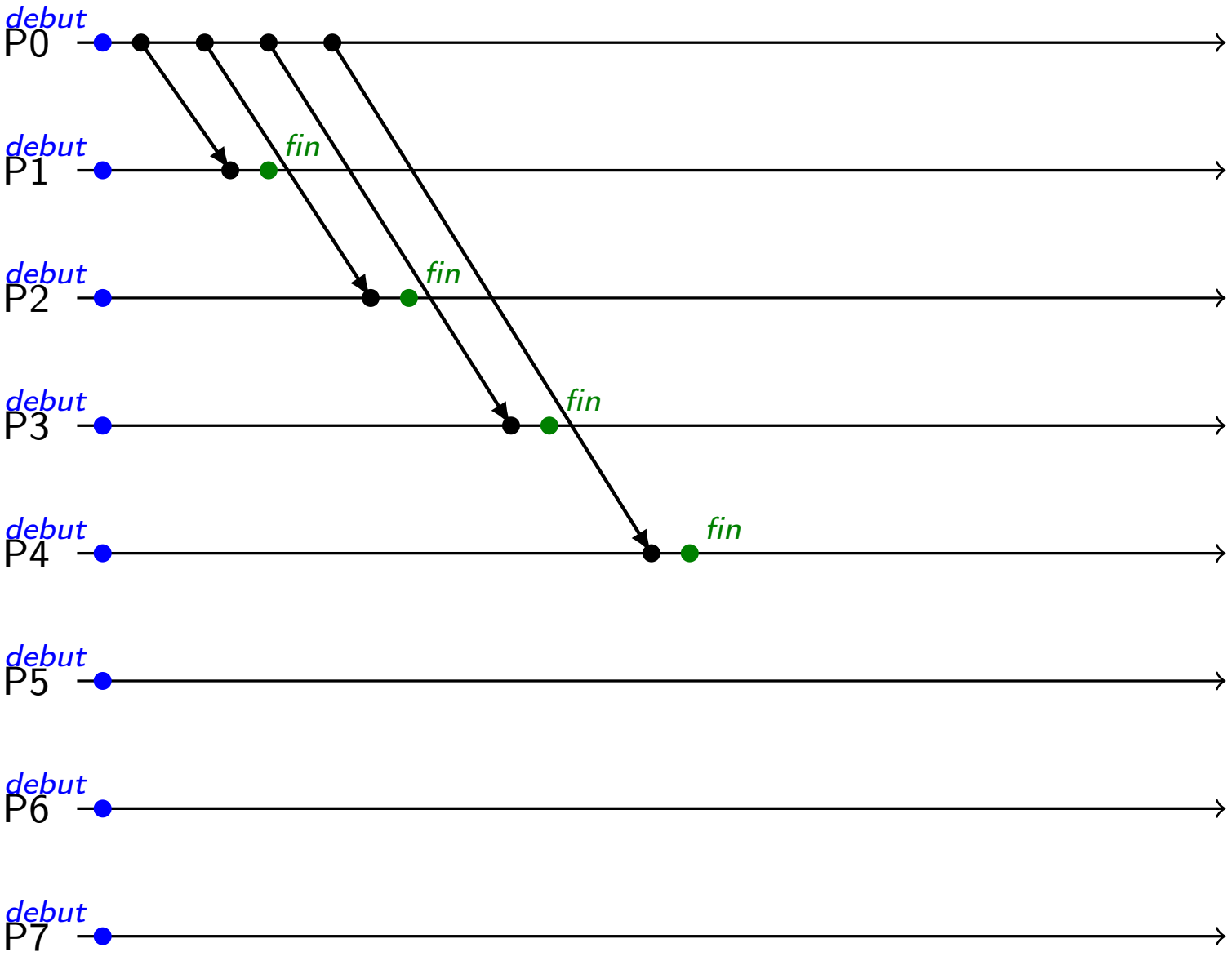
Modèle pour les communications : Bcast étoile



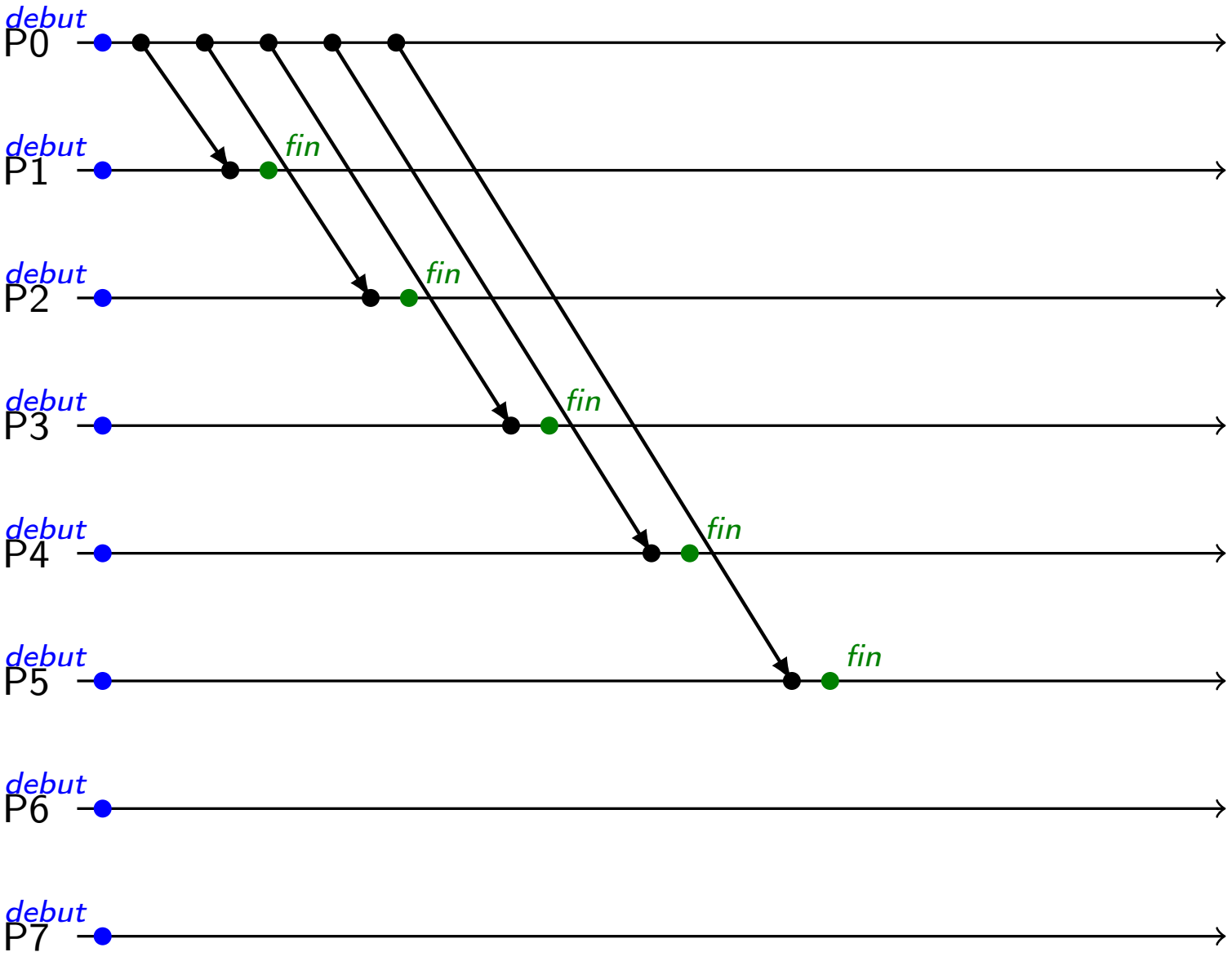
Modèle pour les communications : Bcast étoile



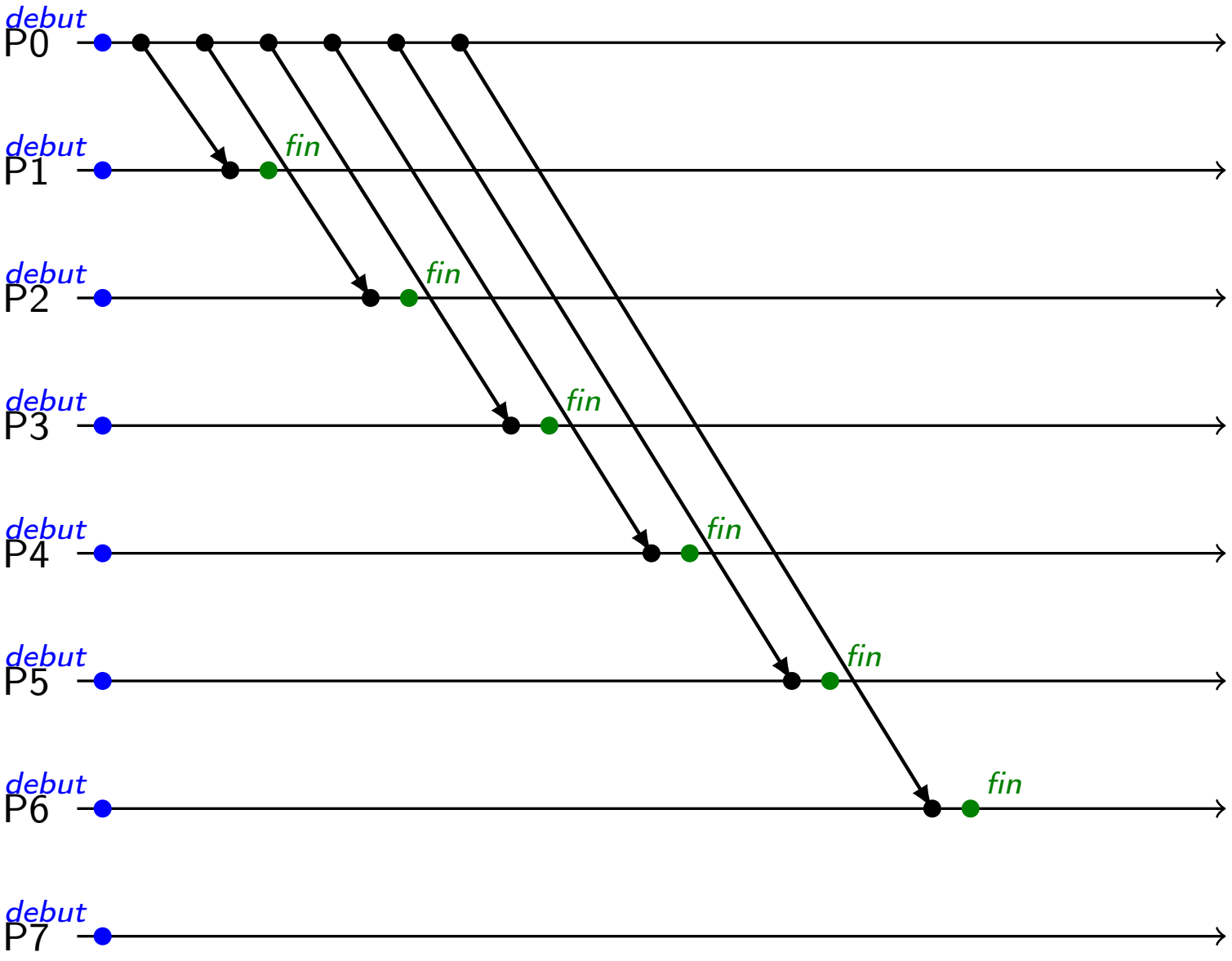
Modèle pour les communications : Bcast étoile



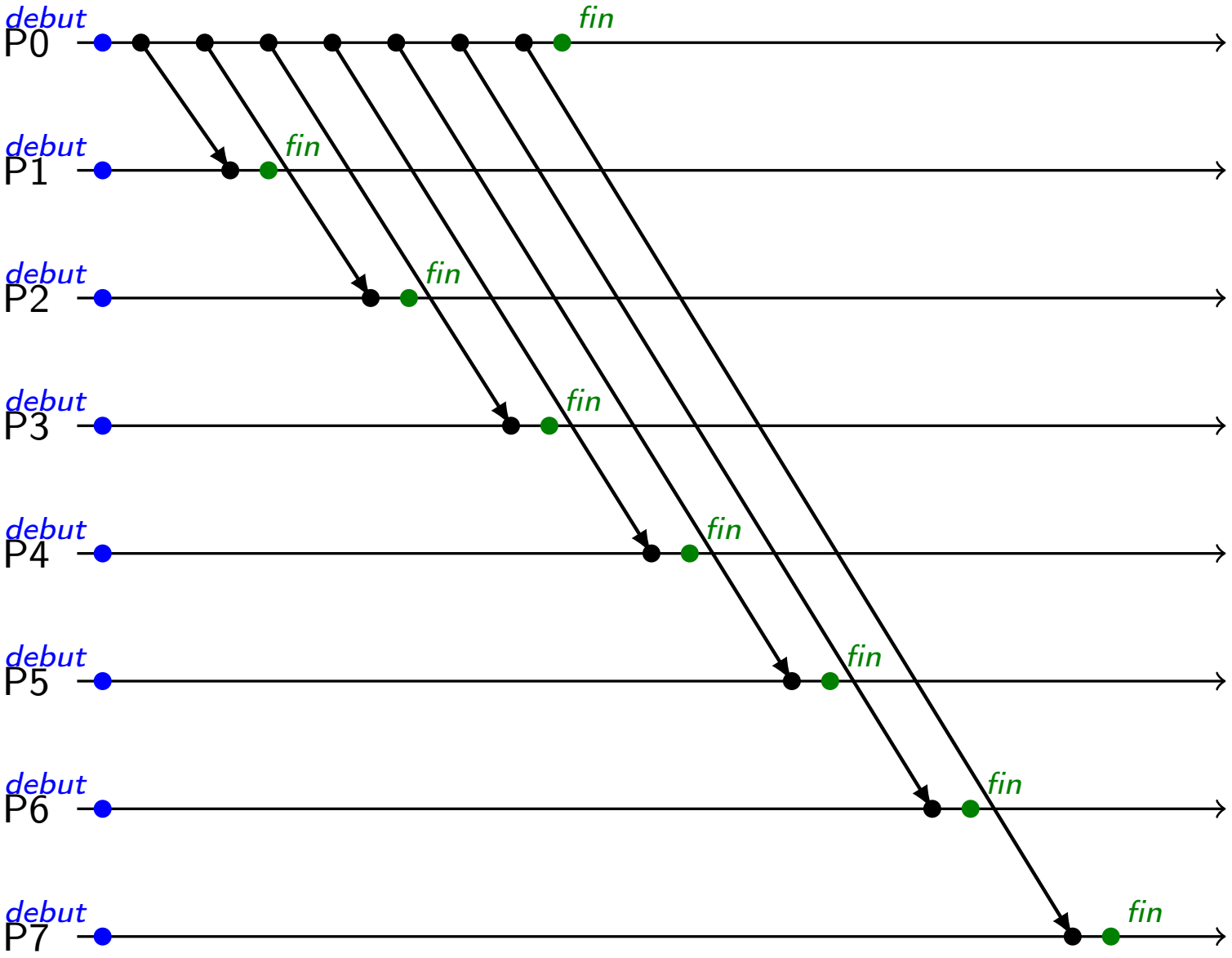
Modèle pour les communications : Bcast étoile



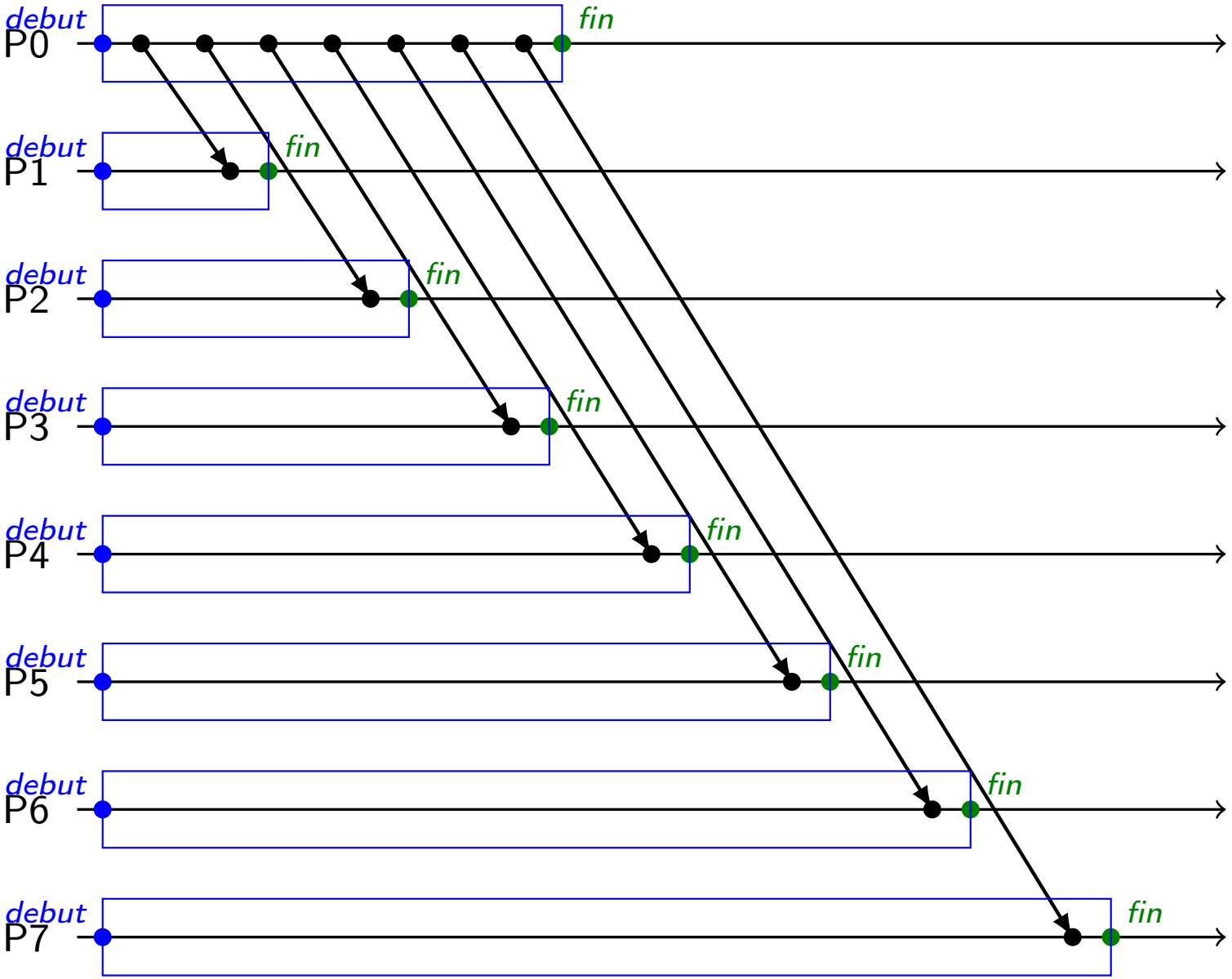
Modèle pour les communications : Bcast étoile



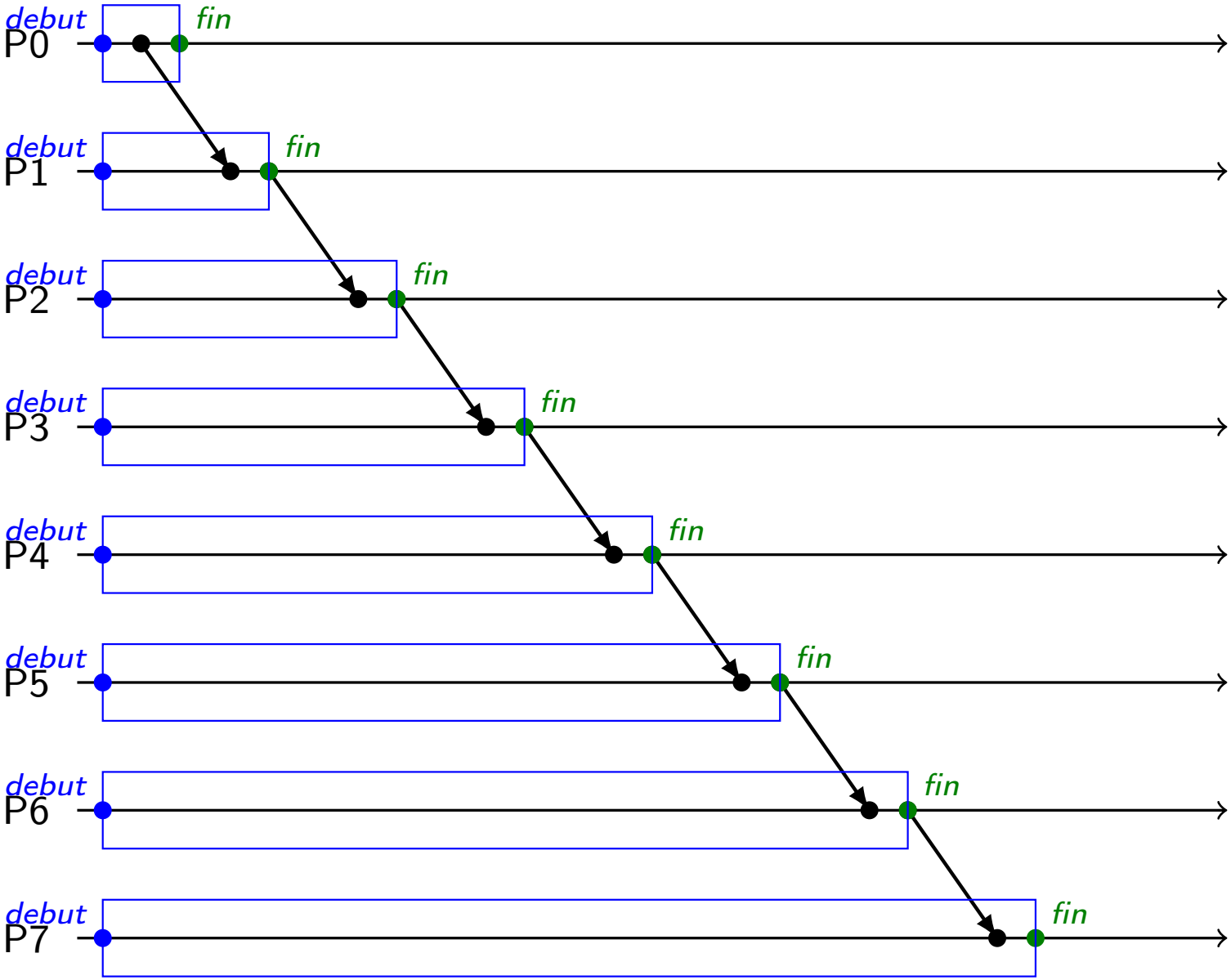
Modèle pour les communications : Bcast étoile



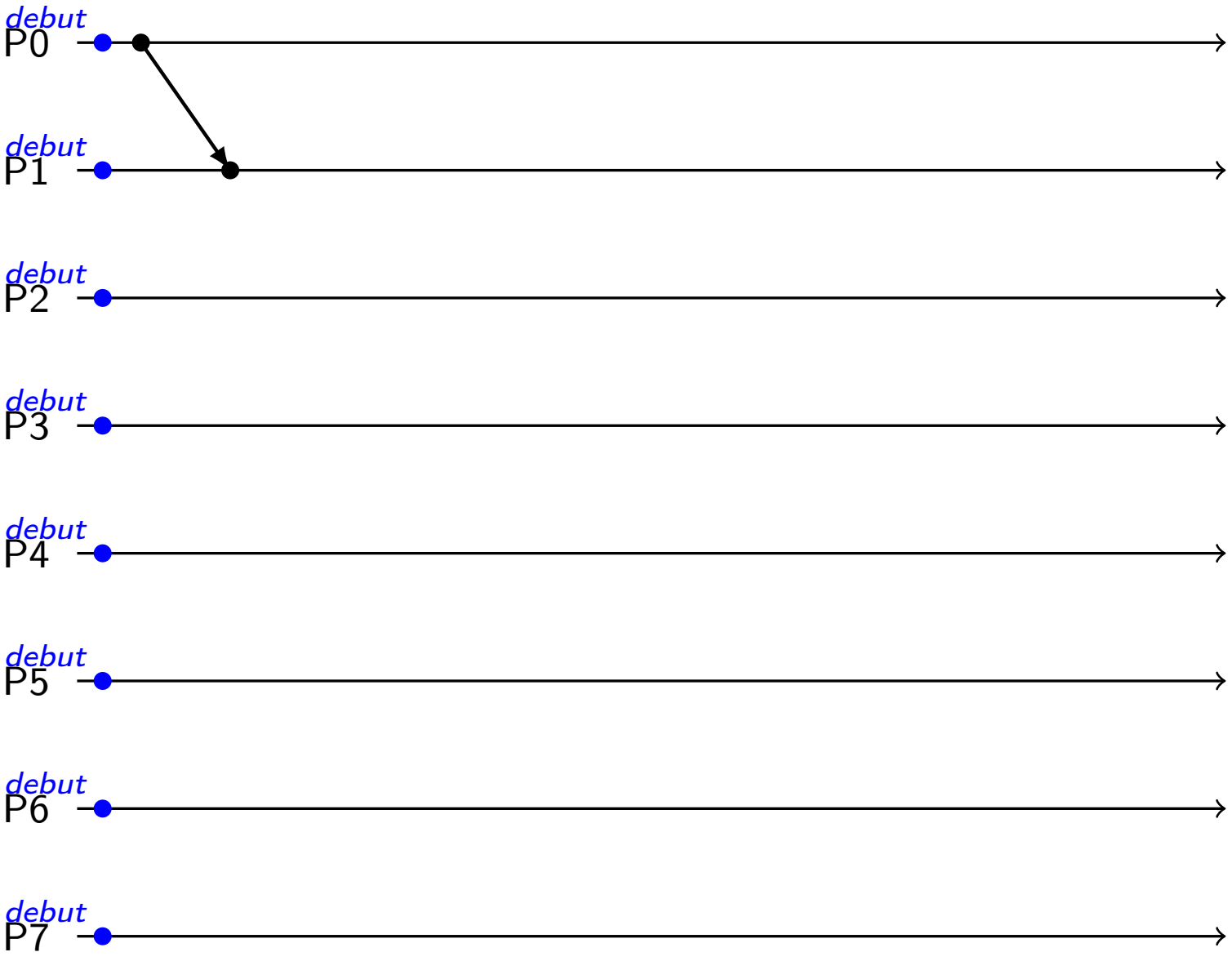
Modèle pour les communications : Bcast étoile



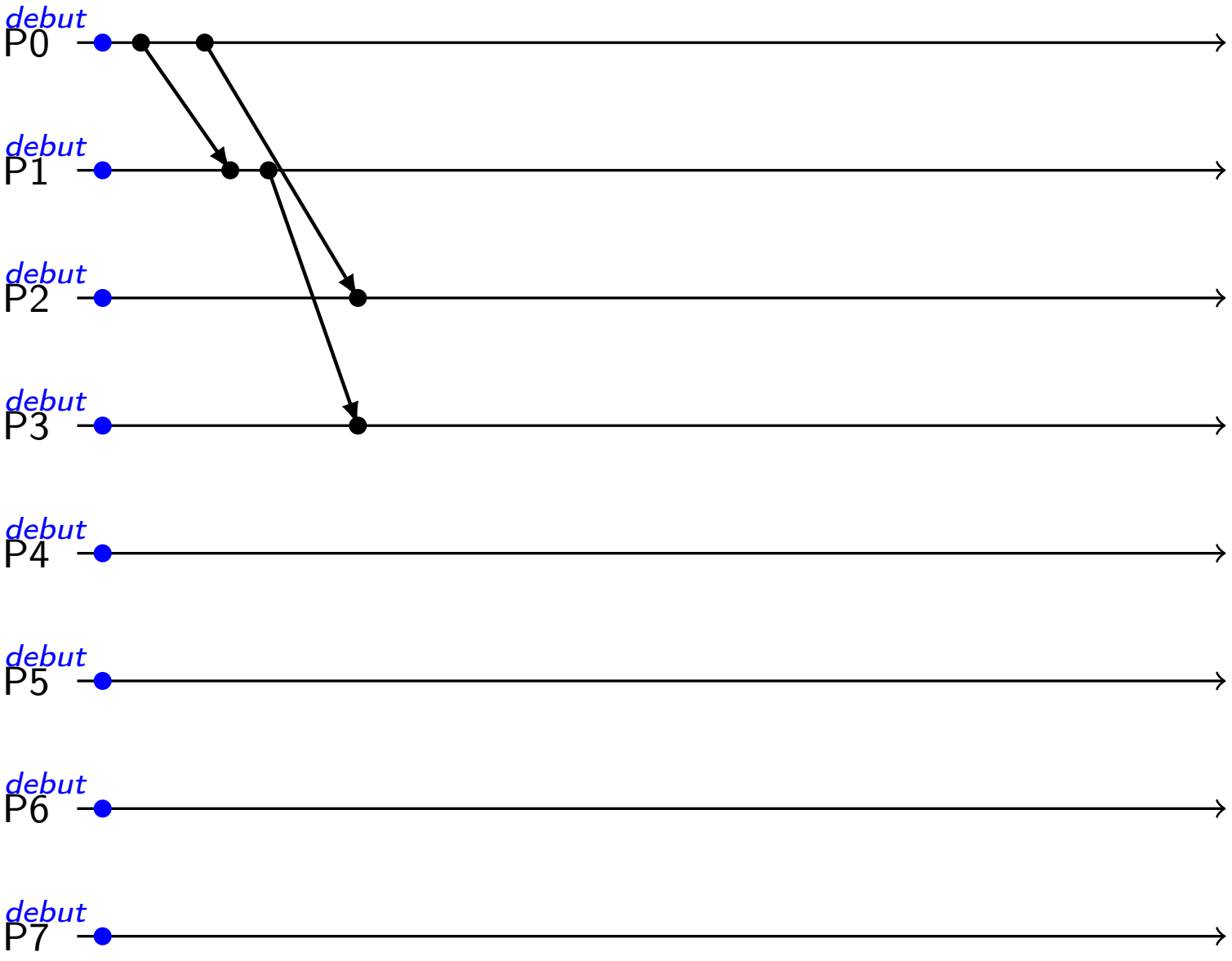
Modèle pour les communications : Bcast chaine



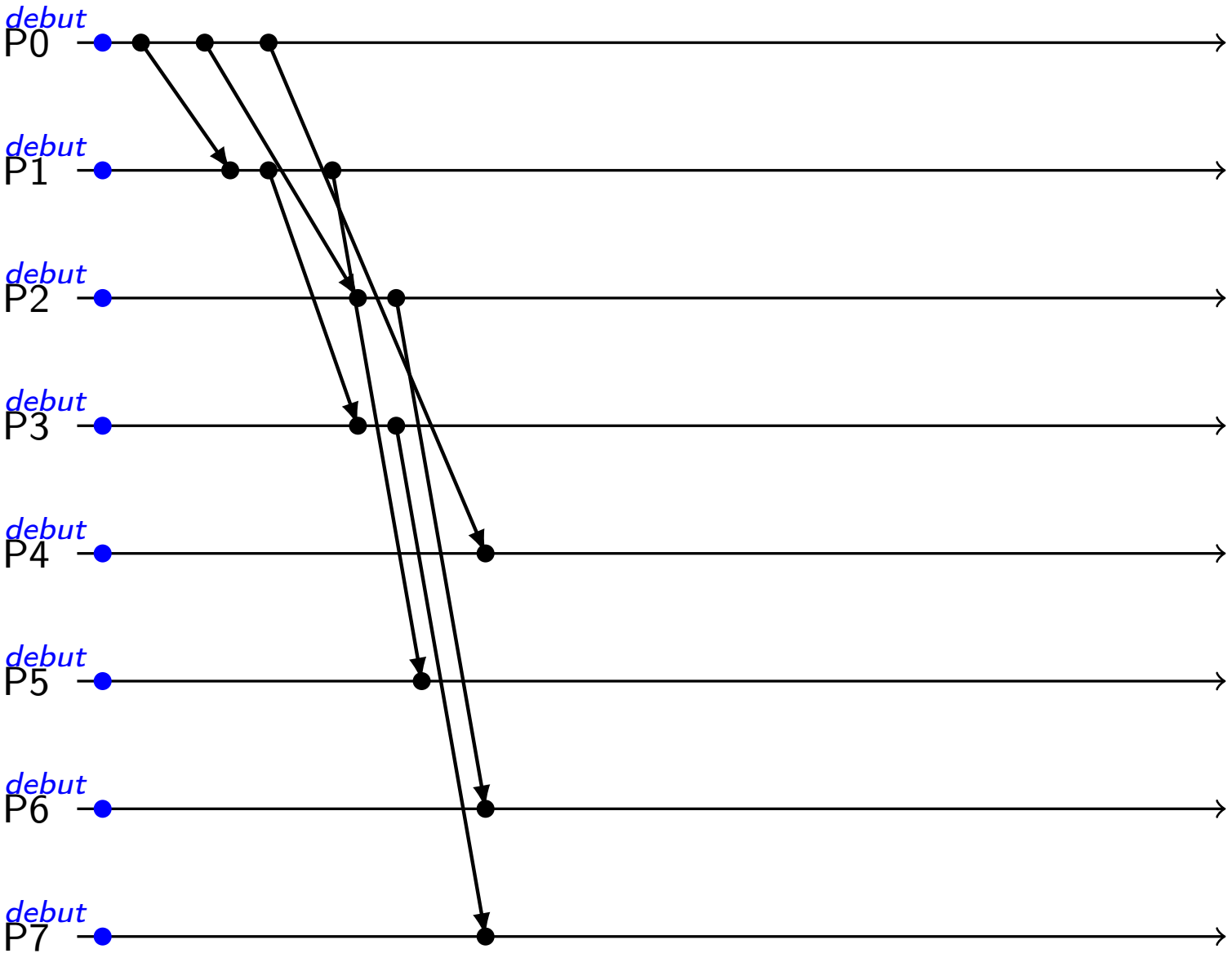
Modèle pour les communications : Bcast binomial



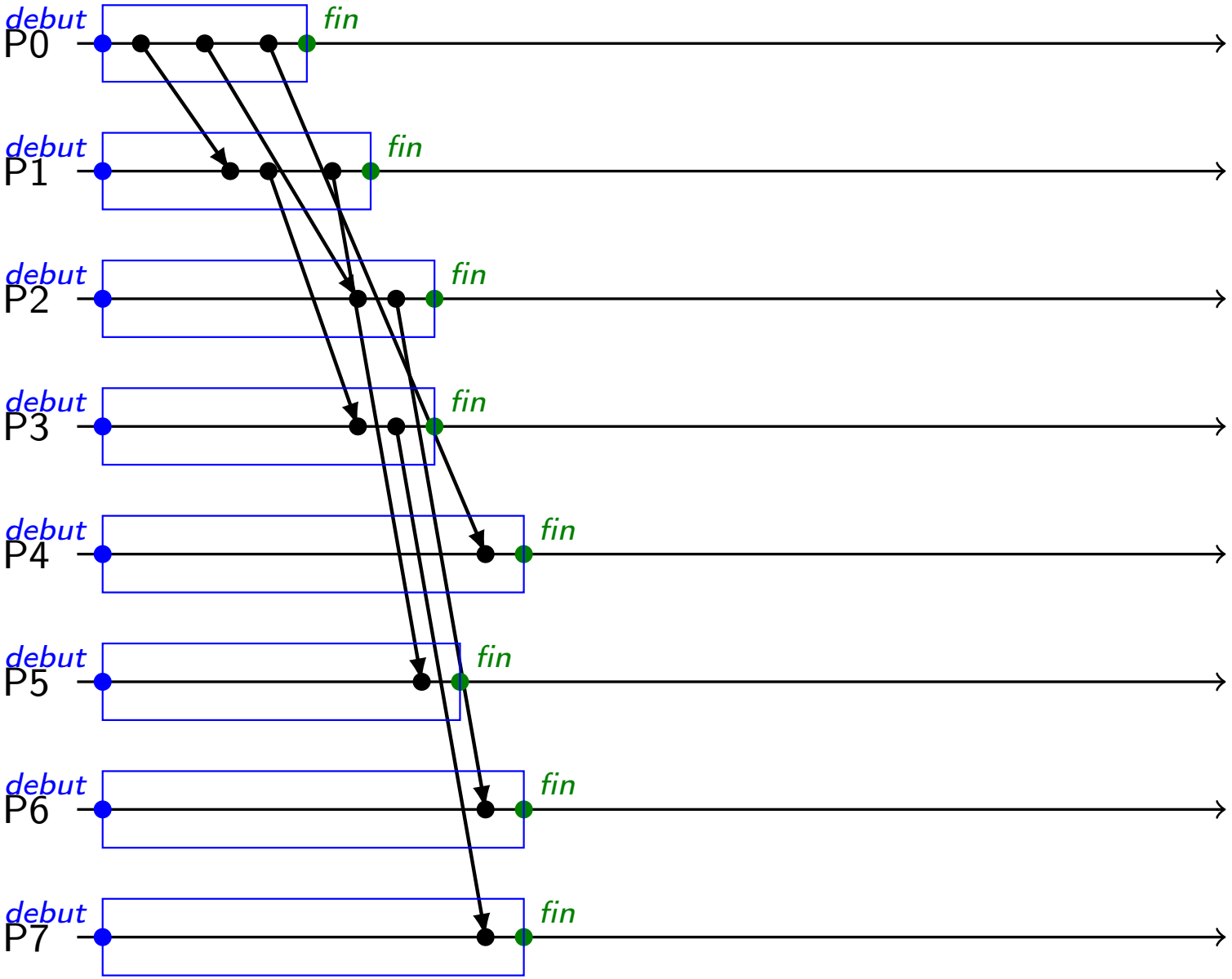
Modèle pour les communications : Bcast binomial



Modèle pour les communications : Bcast binomial



Modèle pour les communications : Bcast binomial



Plan

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 );
```

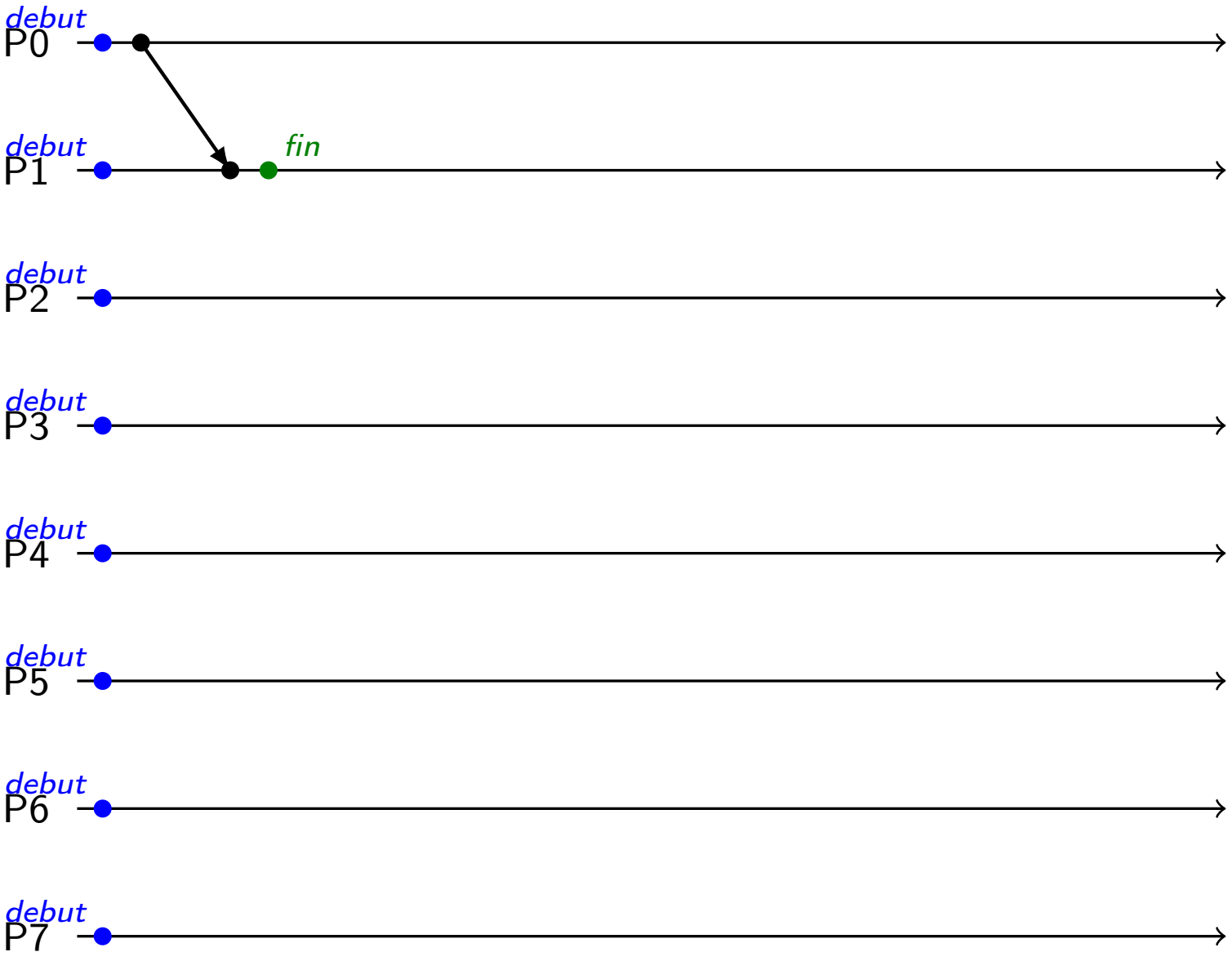
Algorithme de diffusion : l'étoile

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

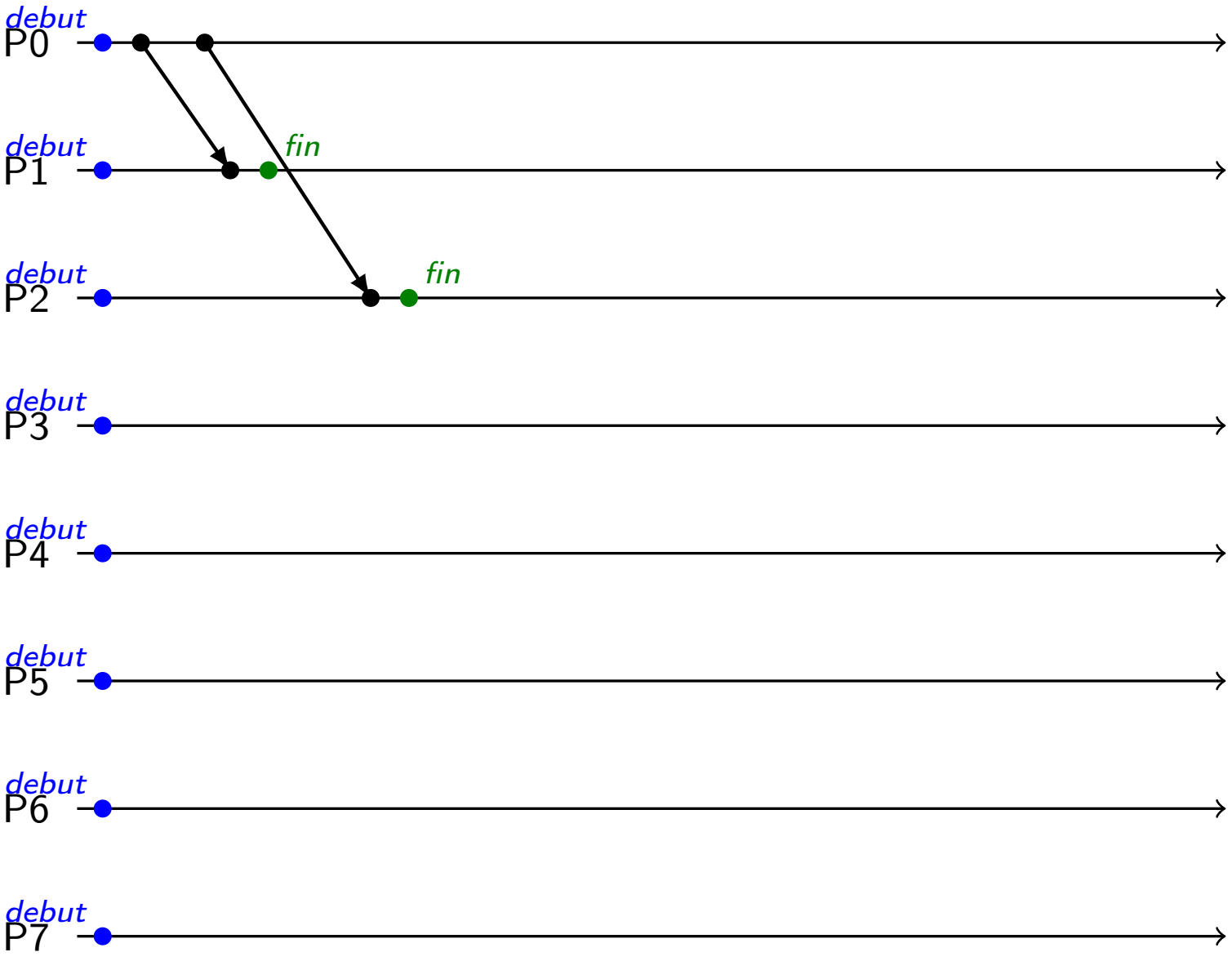
```
if( root == rank ) {
  for( i = 0 ; i < size ; i++ ) {
    if( i != root ) {
      send( message, i )
    }
  }
} else {
  recv( message, root )
}
```

- 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 !!!

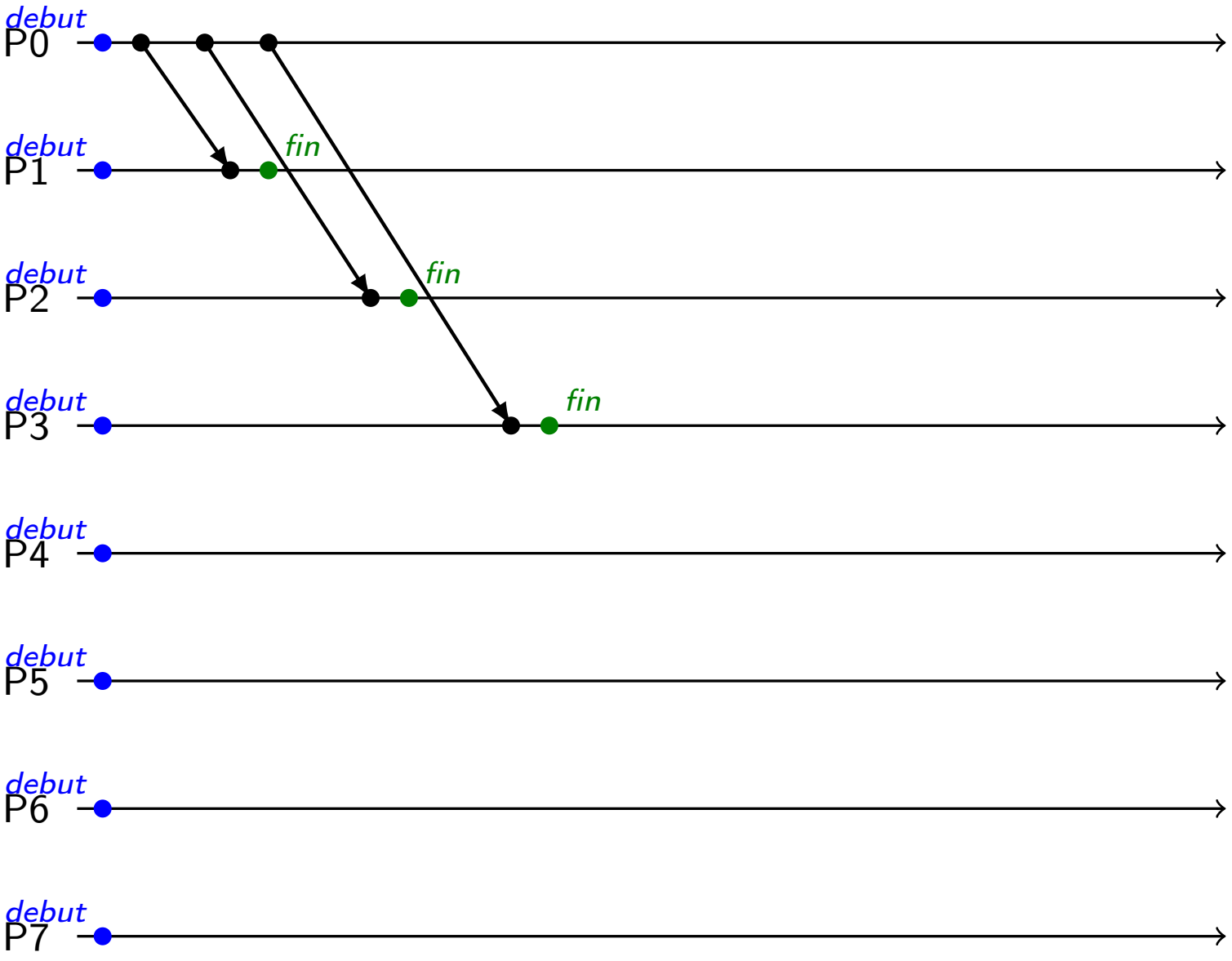
Algorithme de diffusion : l'étoile



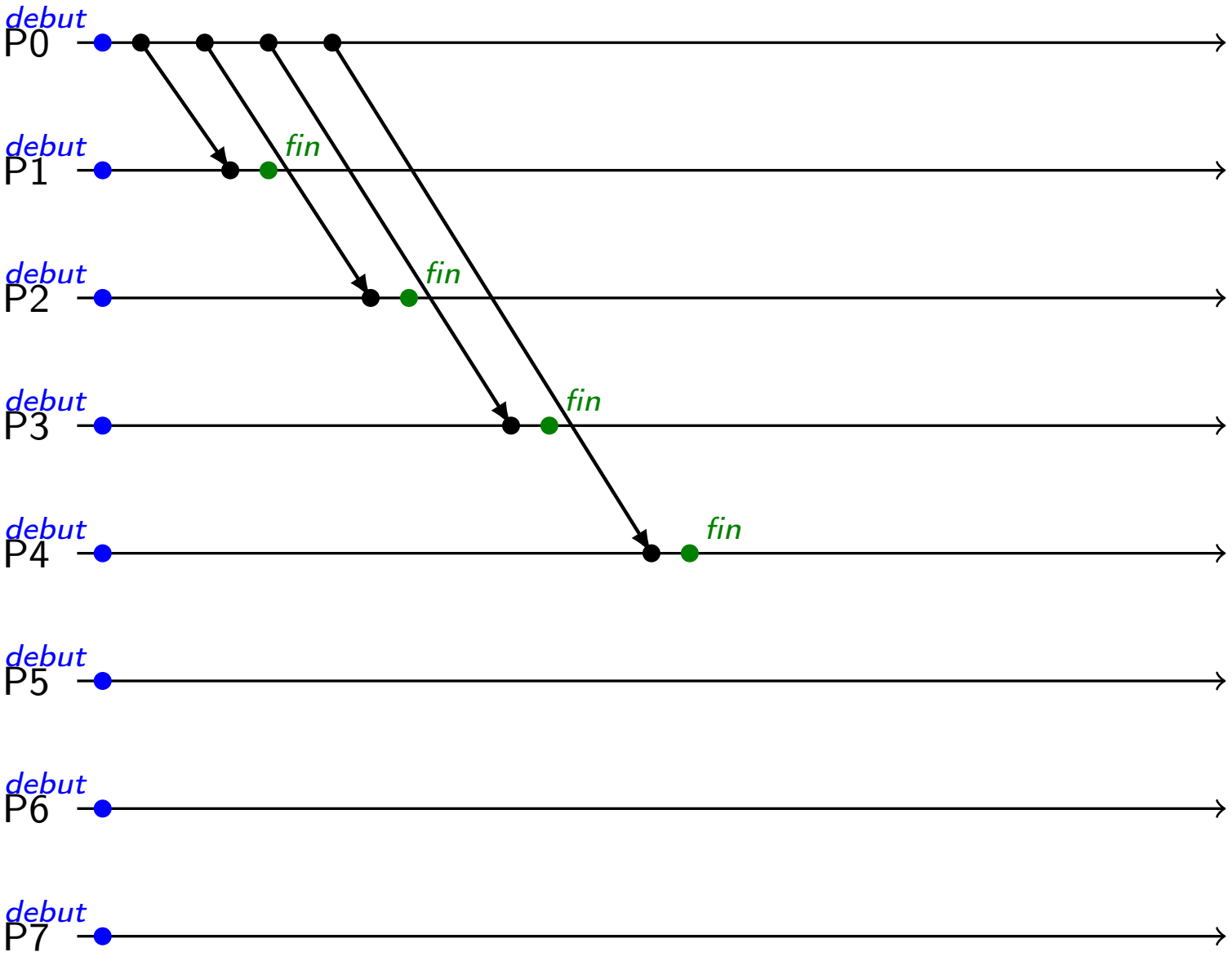
Algorithme de diffusion : l'étoile



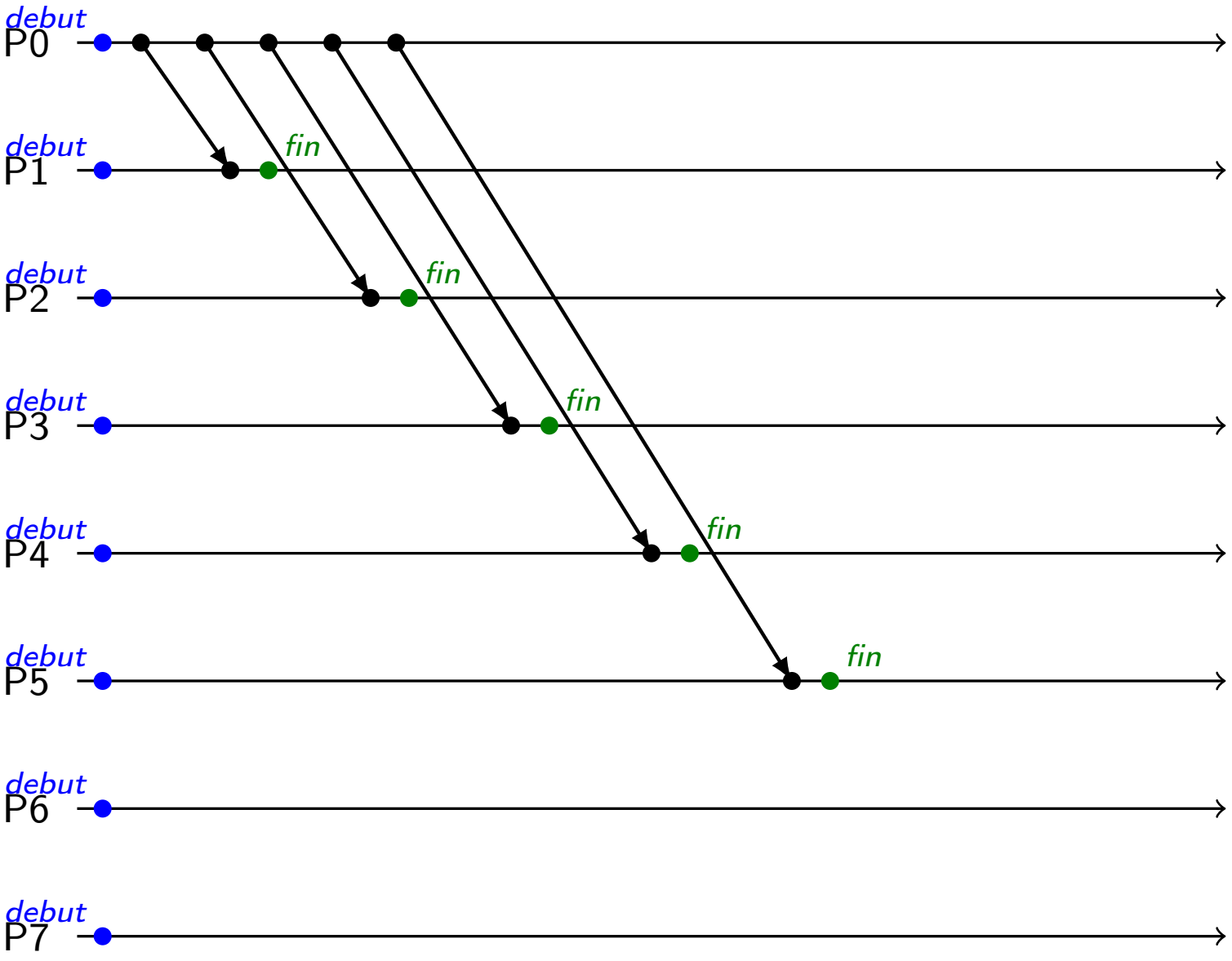
Algorithme de diffusion : l'étoile



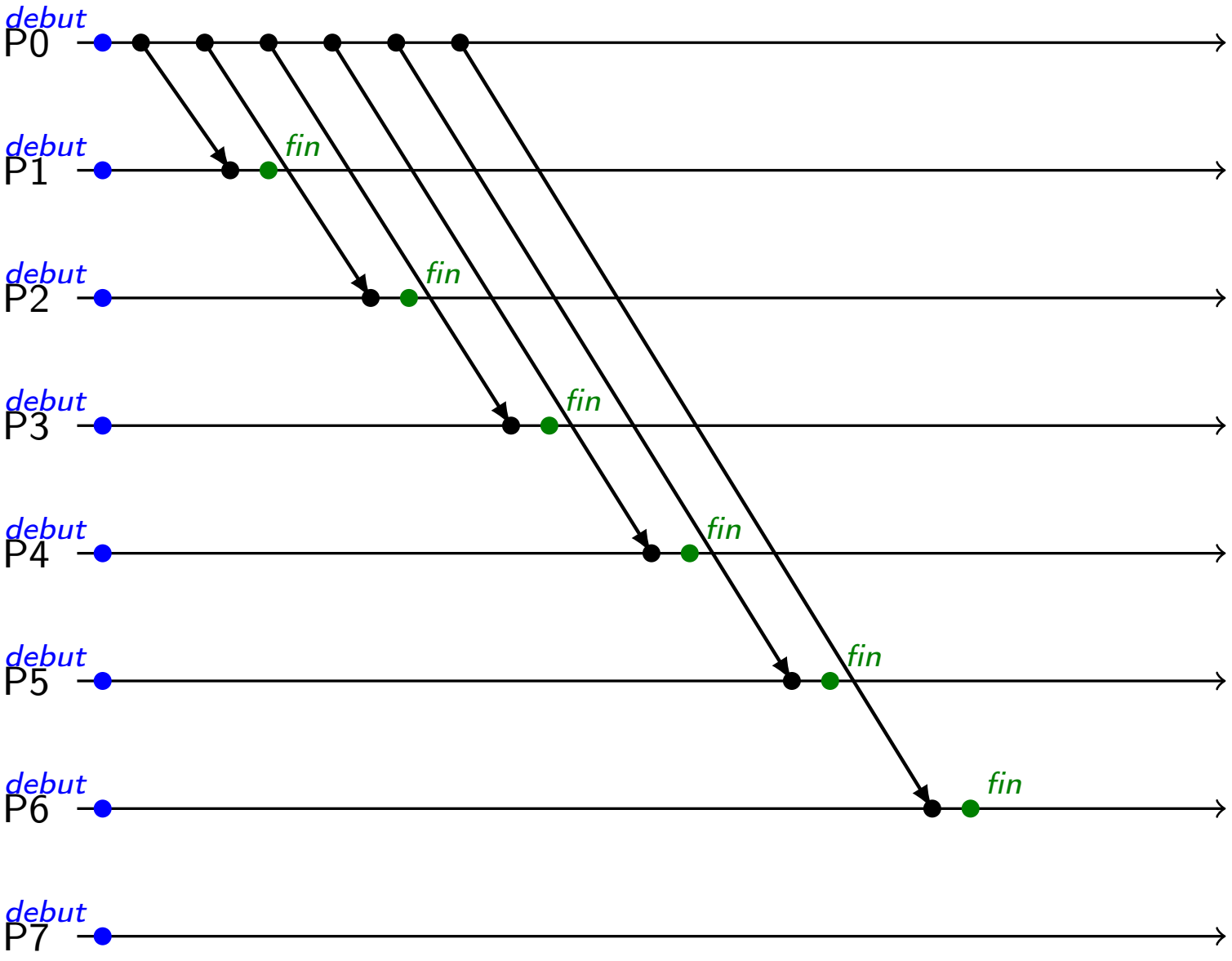
Algorithme de diffusion : l'étoile



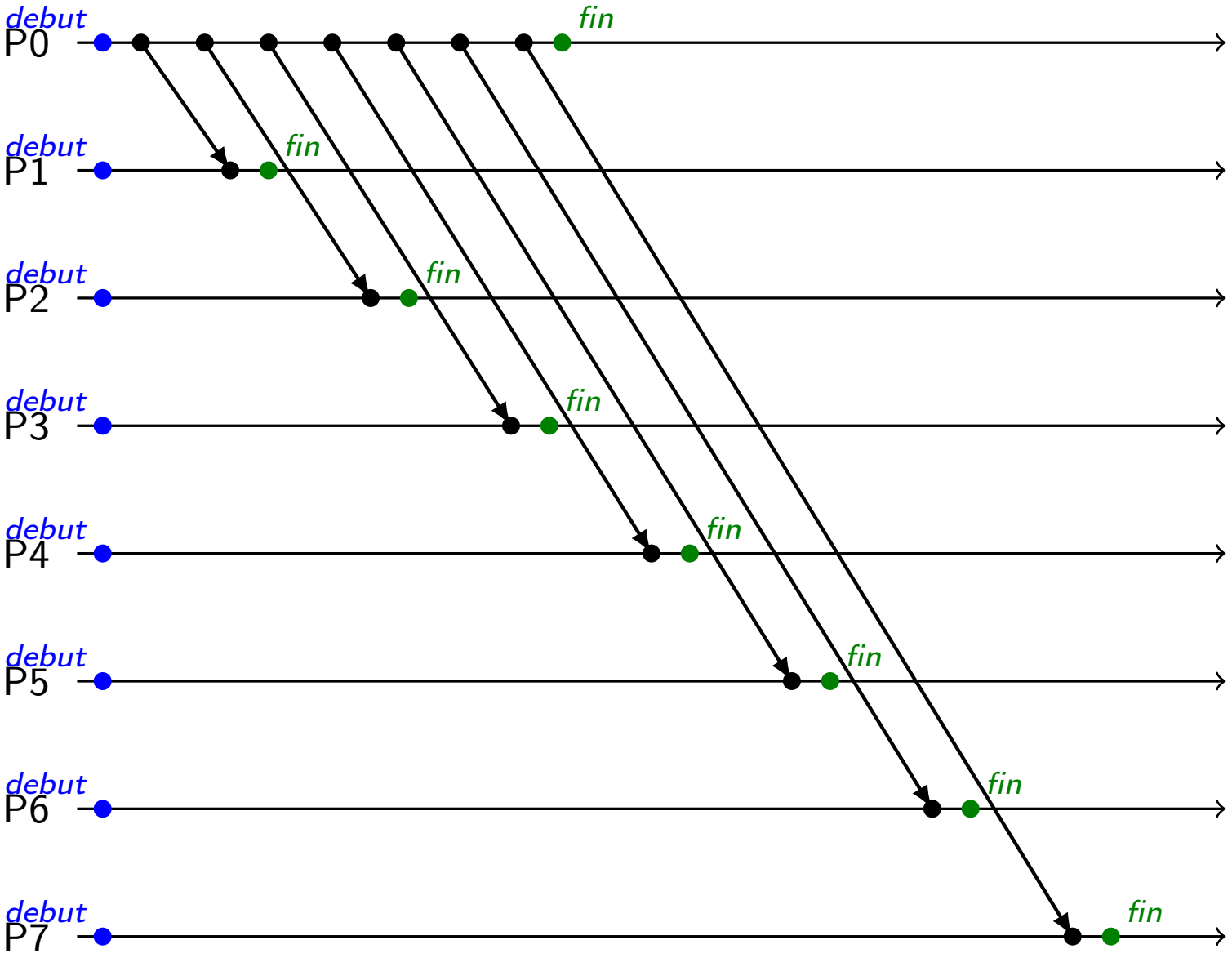
Algorithme de diffusion : l'étoile



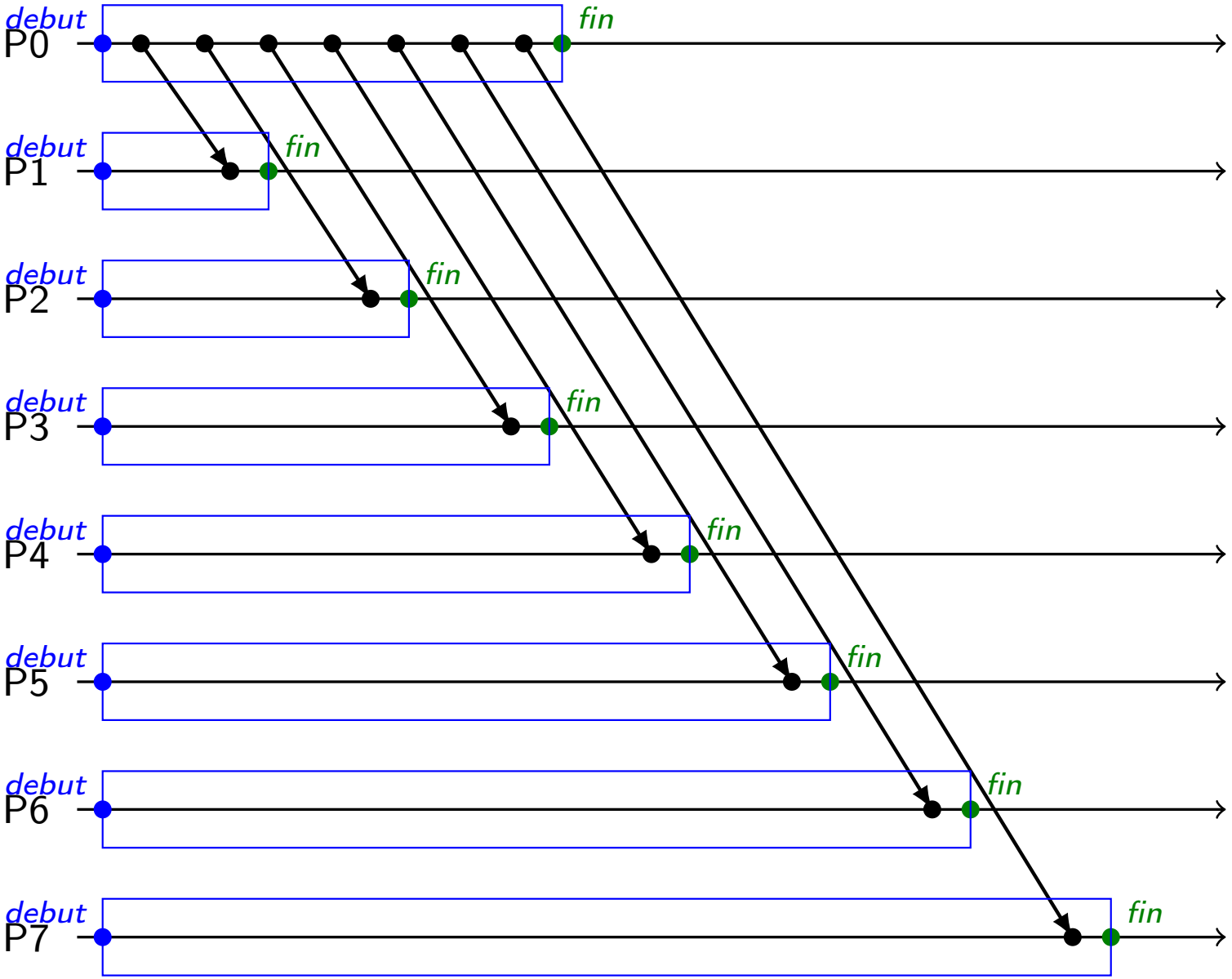
Algorithme de diffusion : l'étoile



Algorithme de diffusion : l'étoile



Algorithme de diffusion : l'étoile



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

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

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

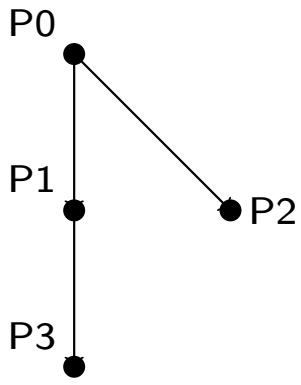
Optimal en nombre de messages dans un modèle 1-port

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

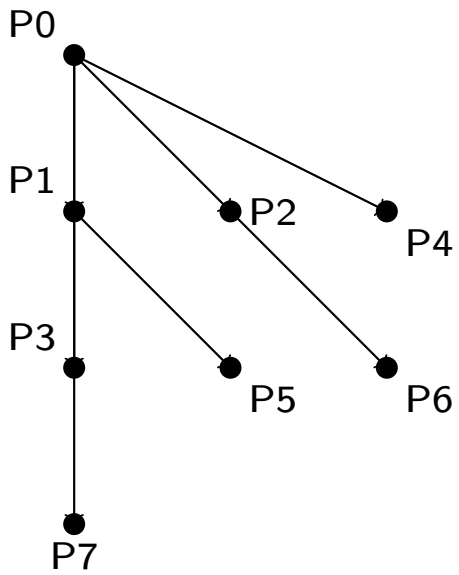
Algorithme de diffusion : arbre binomial

Algorithme de diffusion : arbre binomial

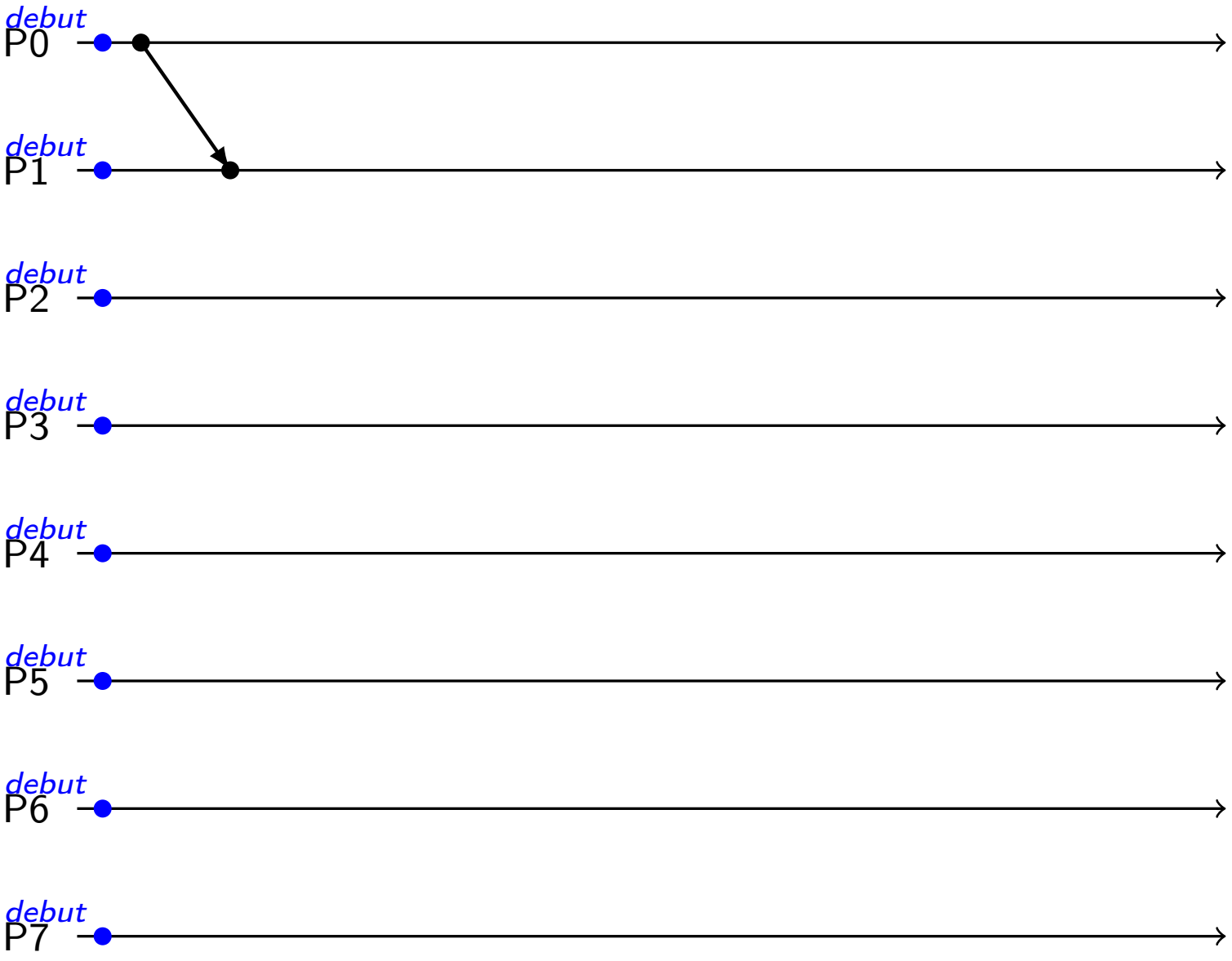
Algorithme de diffusion : arbre binomial



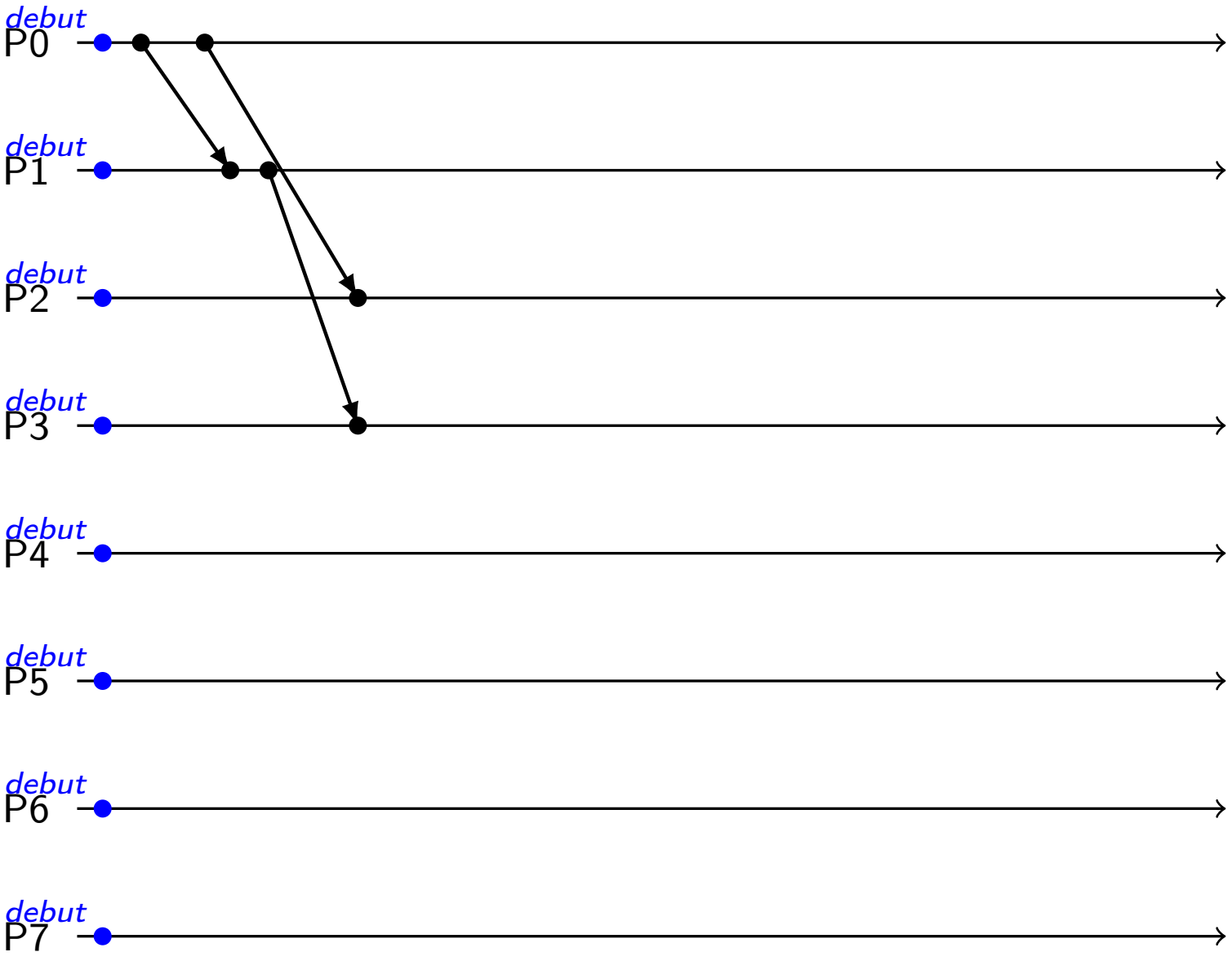
Algorithme de diffusion : arbre binomial



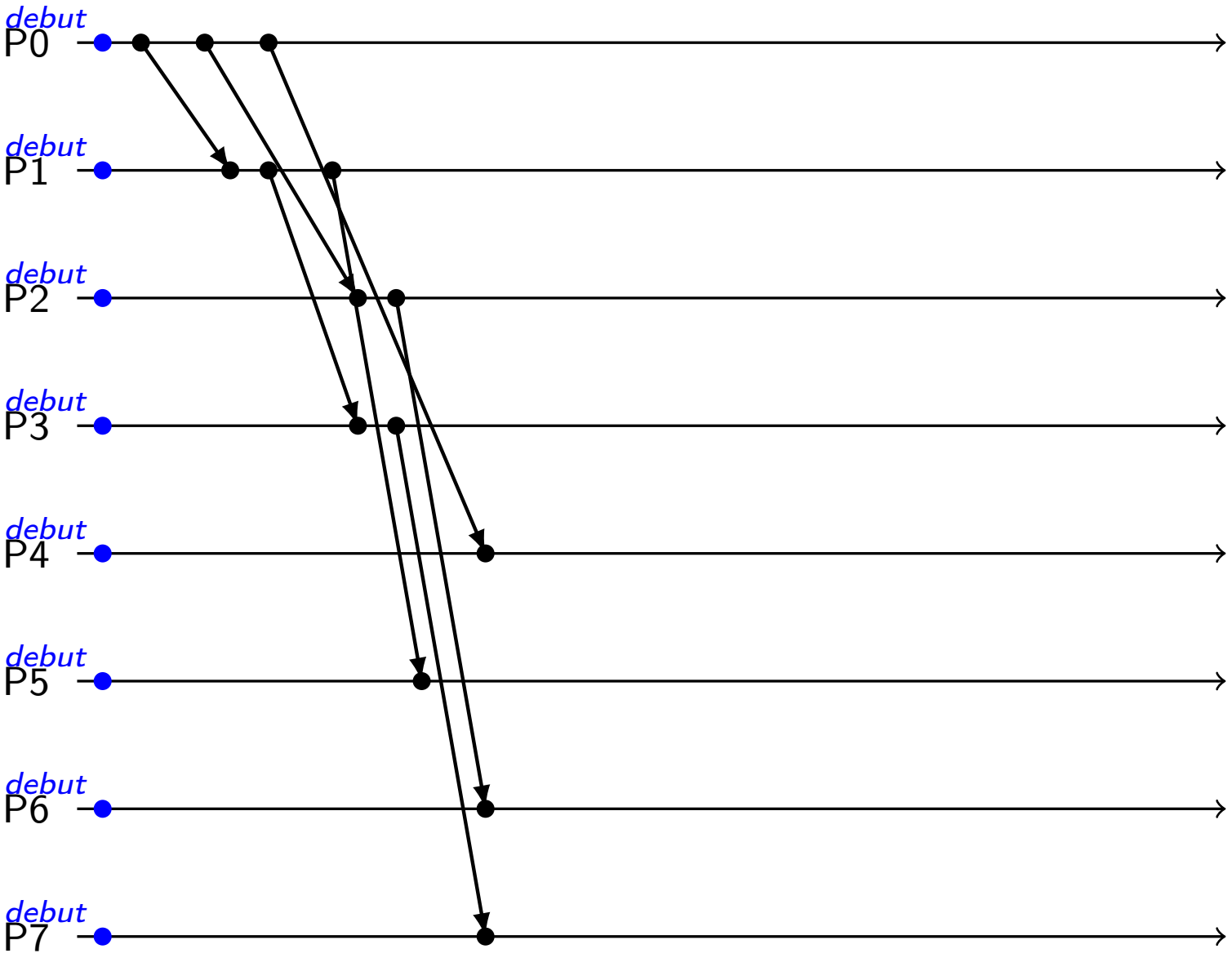
Algorithme de diffusion : arbre binomial



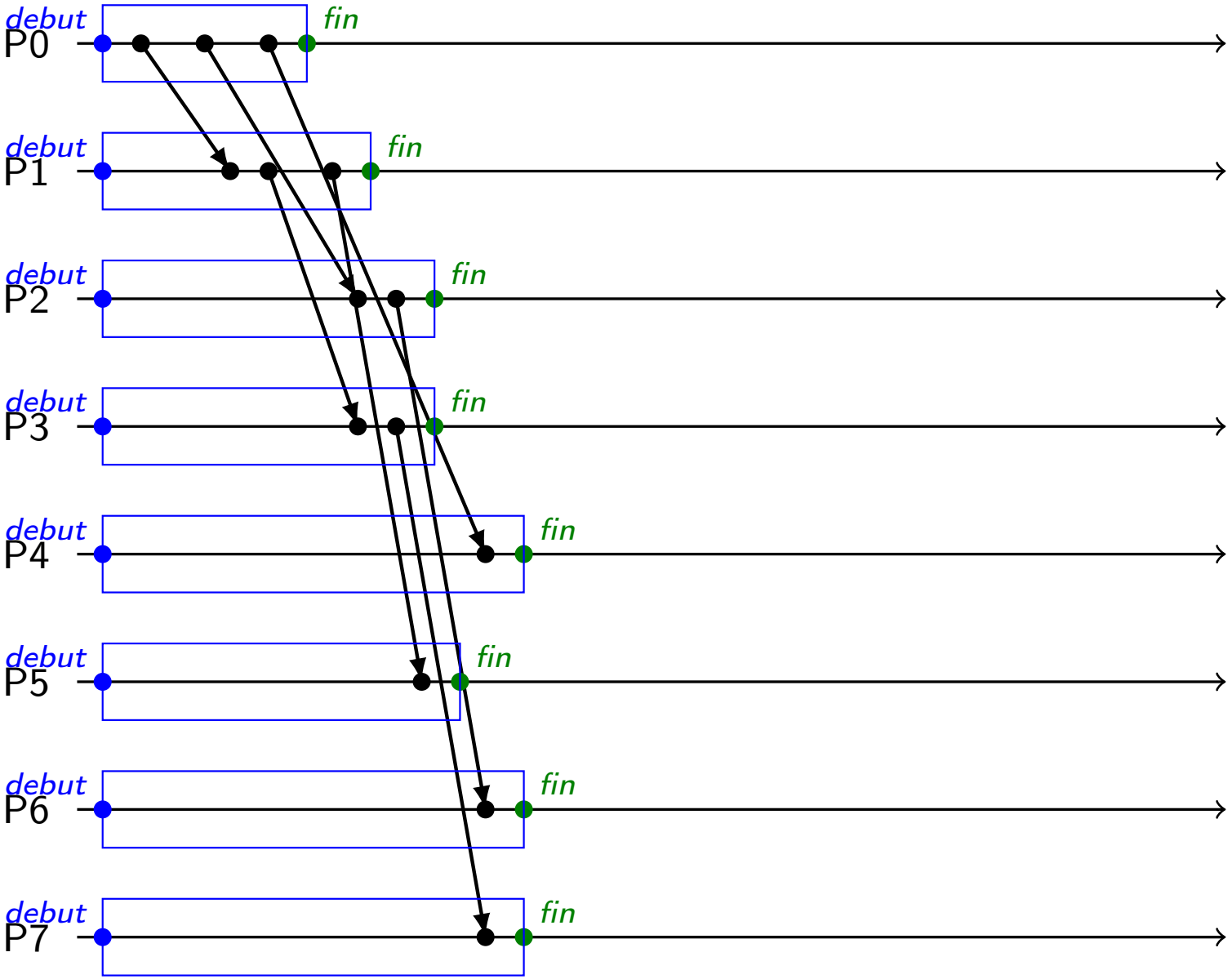
Algorithme de diffusion : arbre binomial



Algorithme de diffusion : arbre binomial



Algorithme de diffusion : arbre binomial



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)

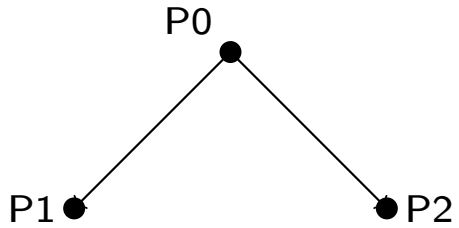
Hauteur de l'arbre = $\lceil log_k(N) \rceil$
Bon dans un modèle k-port

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

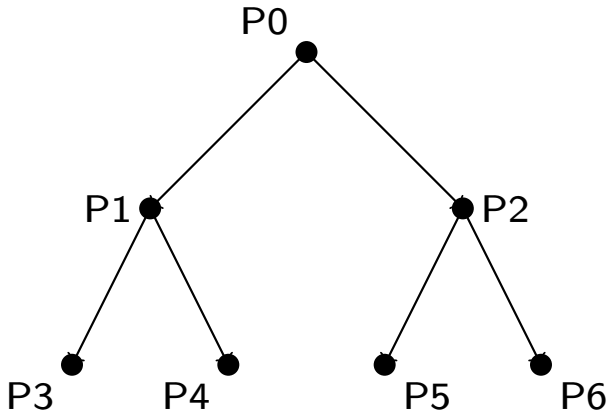
Moins intéressant que l'arbre binomial dans un modèle 1-port.

Algorithme de diffusion : arbre de Fibonacci

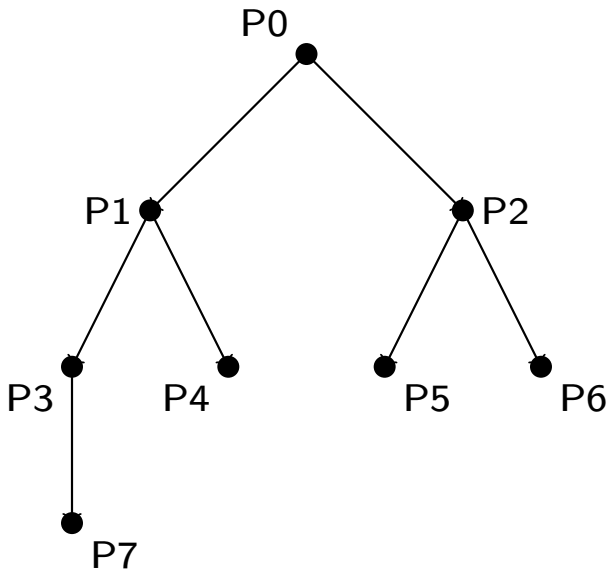
Algorithme de diffusion : arbre de Fibonacci



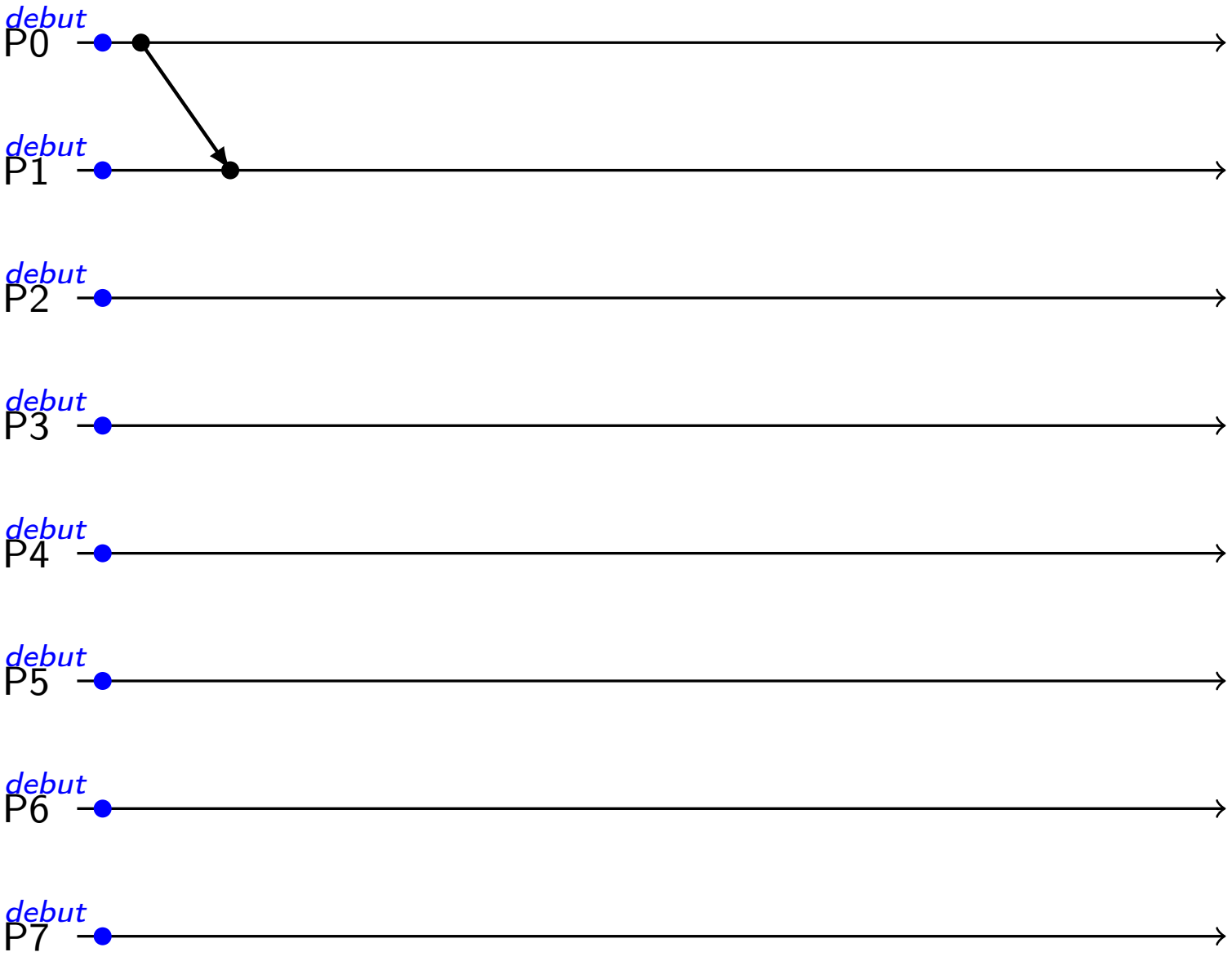
Algorithme de diffusion : arbre de Fibonacci



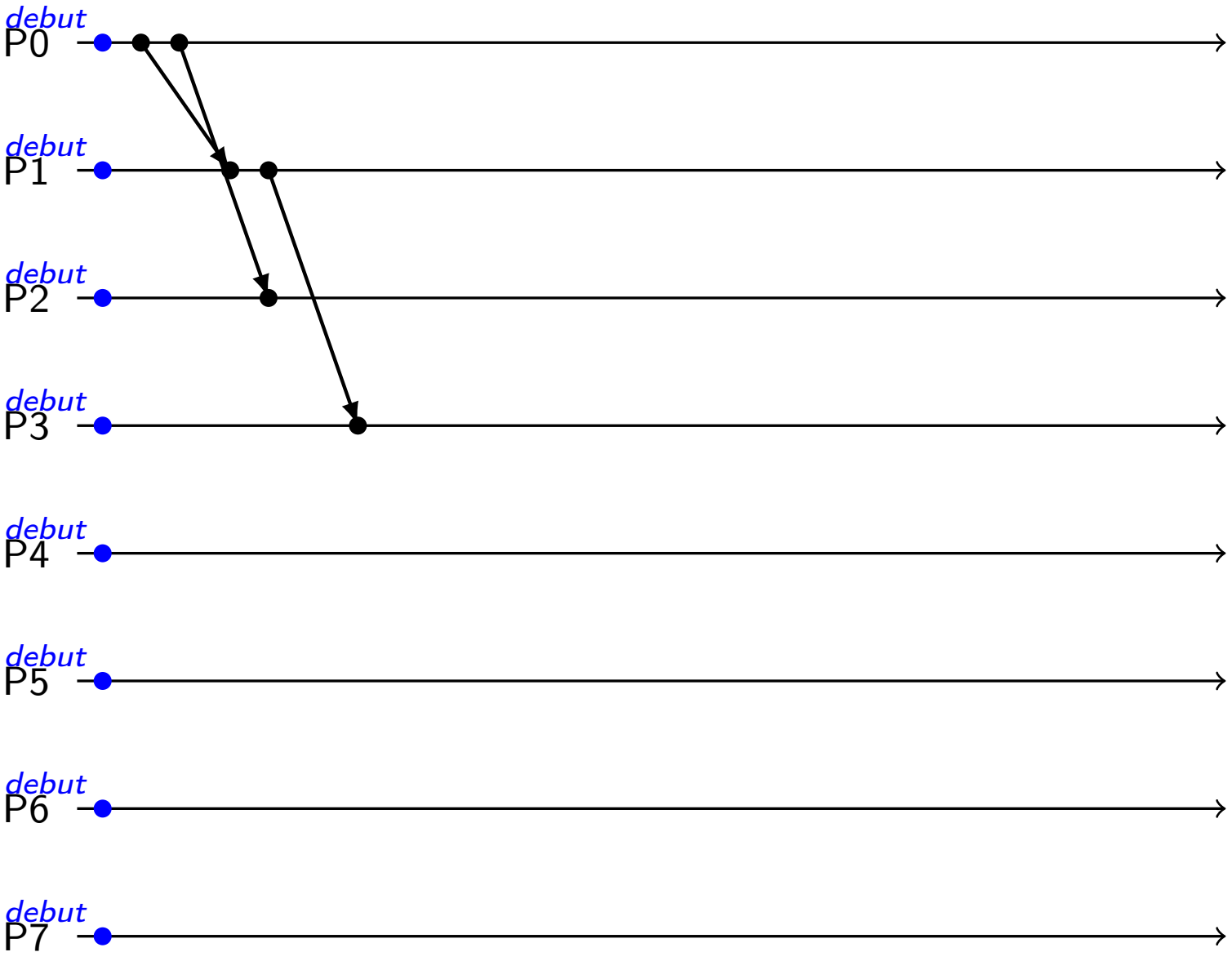
Algorithme de diffusion : arbre de Fibonacci



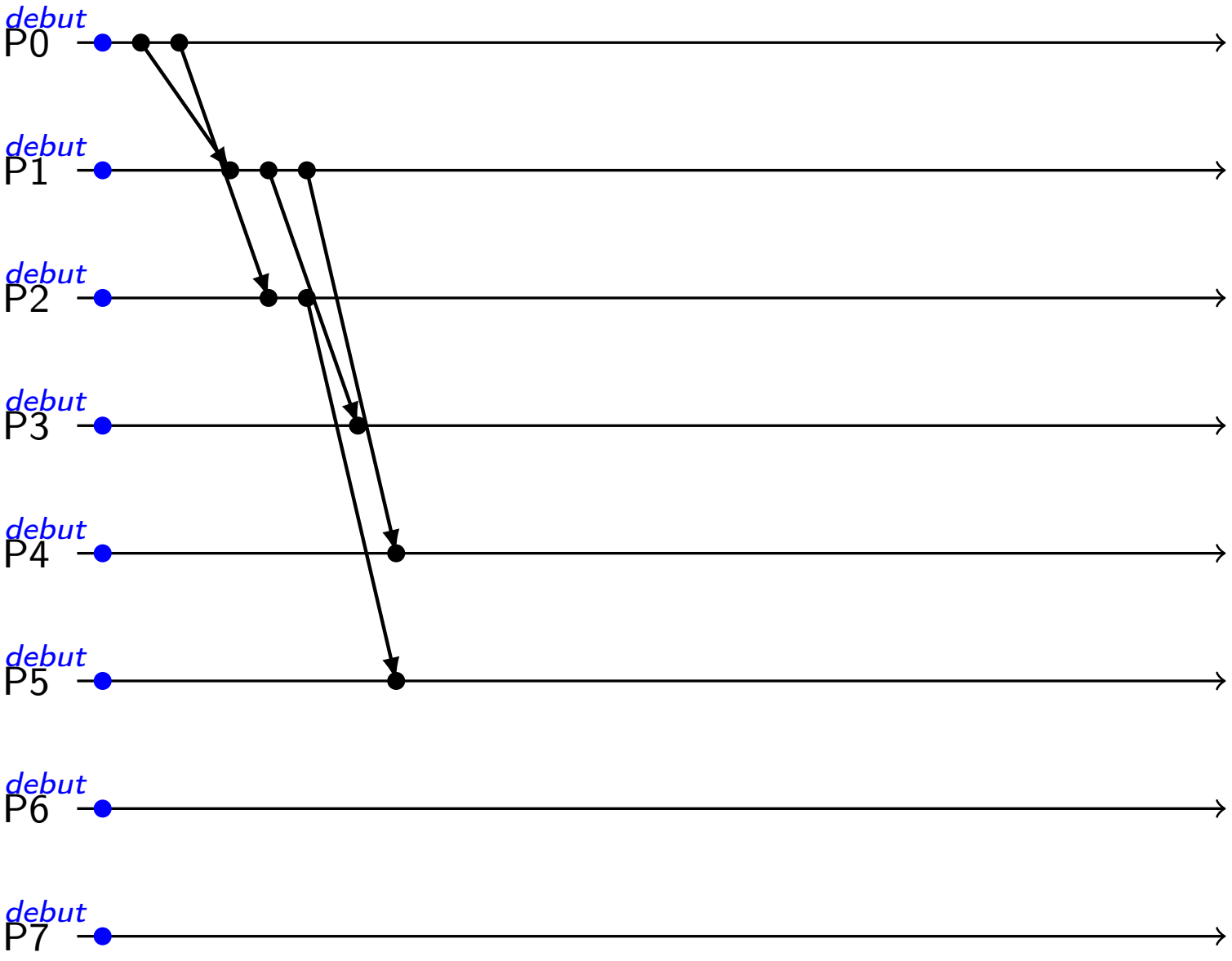
Algorithme de diffusion : arbre de Fibonacci



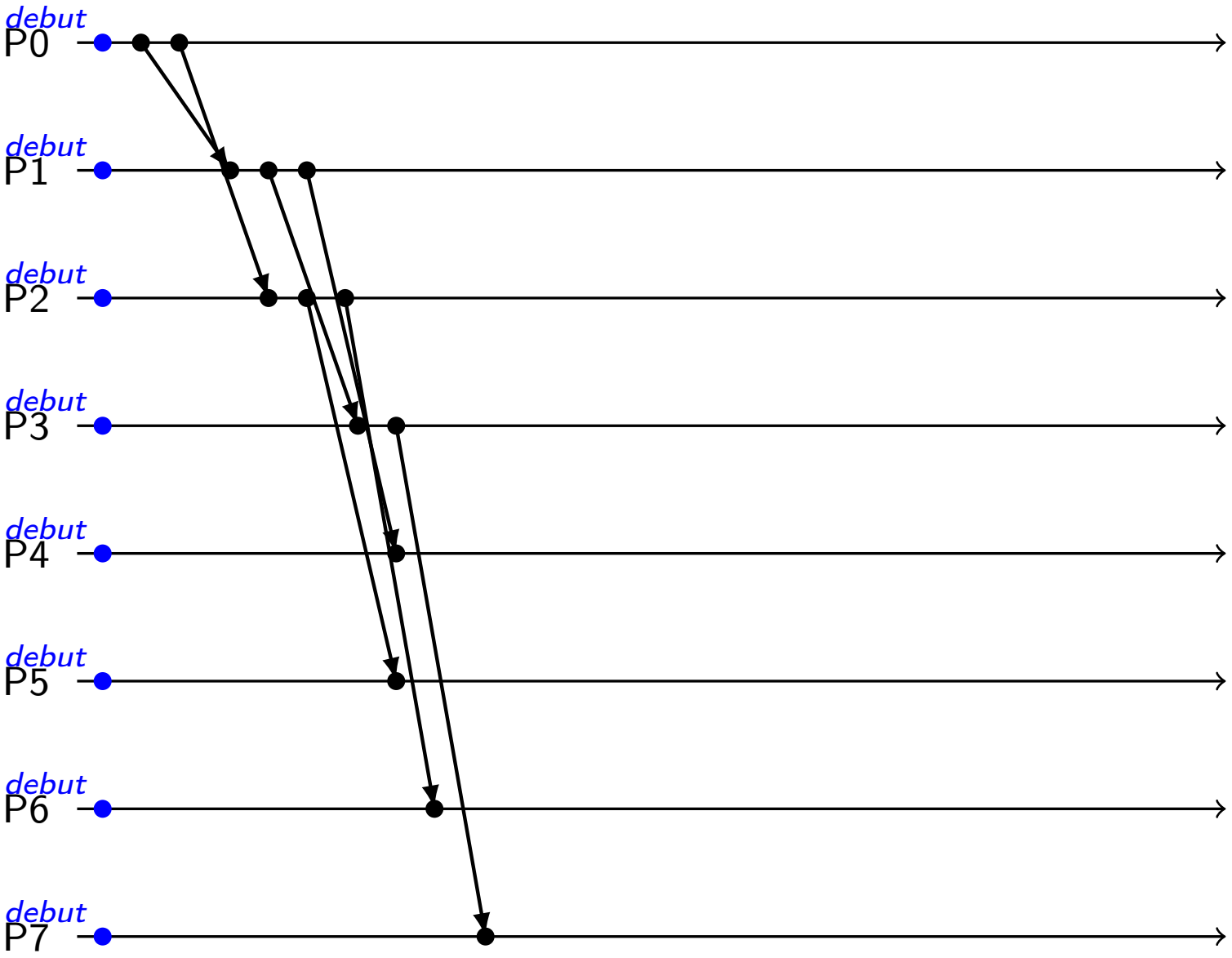
Algorithme de diffusion : arbre de Fibonacci



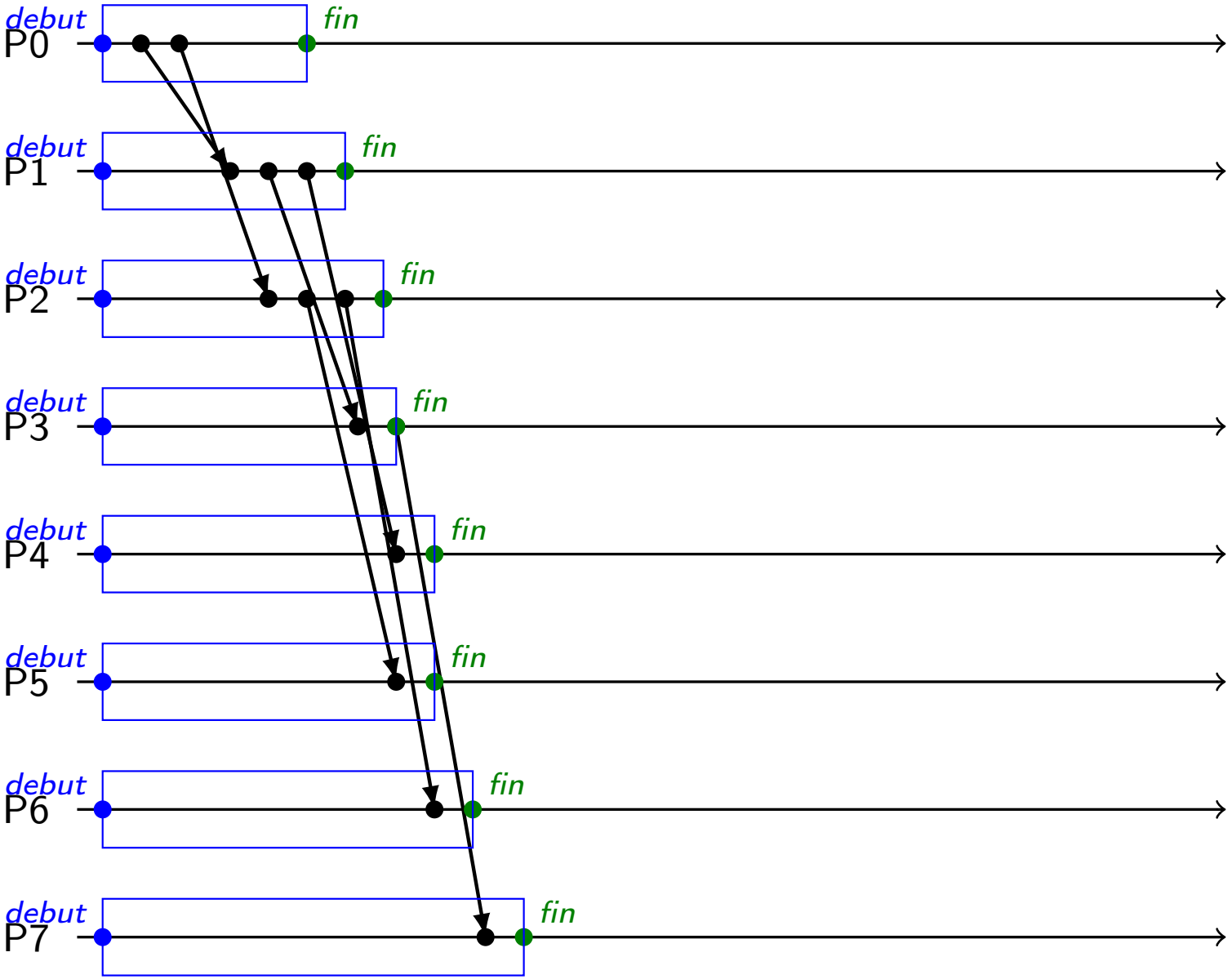
Algorithme de diffusion : arbre de Fibonacci



Algorithme de diffusion : arbre de Fibonacci



Algorithme de diffusion : arbre de Fibonacci



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

Algorithme de diffusion : split-binary tree

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

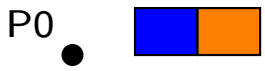
- La racine divise par 2 le message
- La première moitié est envoyée à son fils de droite
- La deuxième moitié est envoyée à son fils de gauche
- Les demi-messages sont transmis dans les deux sous-arbres
- Arrivé en bas, chaque processus du sous-arbre de droite échange son demi-message avec un processus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

Complexités :

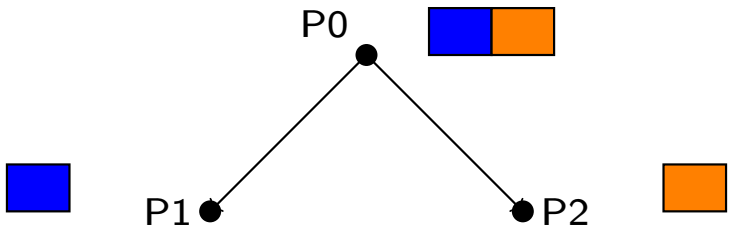
- Nombre de messages : un de plus qu'un arbre binaire
- La technique de division du message divise par 2 la taille du message à transmettre
- Donc on multiplie par 2 la bande passante disponible au coût d'un message supplémentaire

Intéressant pour les gros messages !

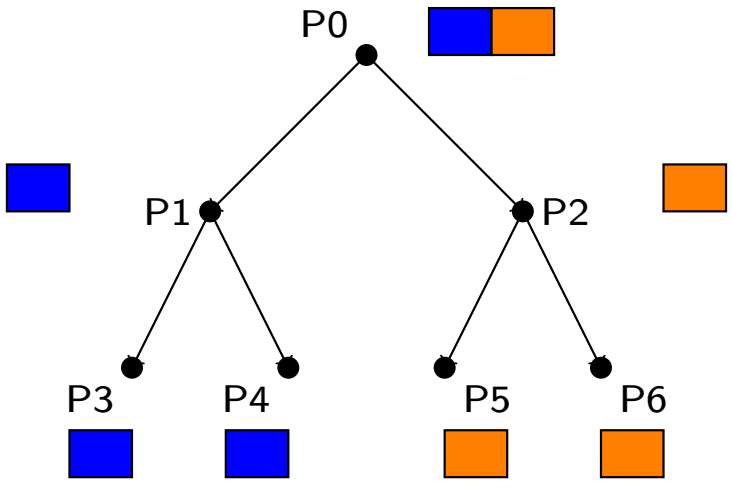
Algorithme de diffusion : split-binary tree



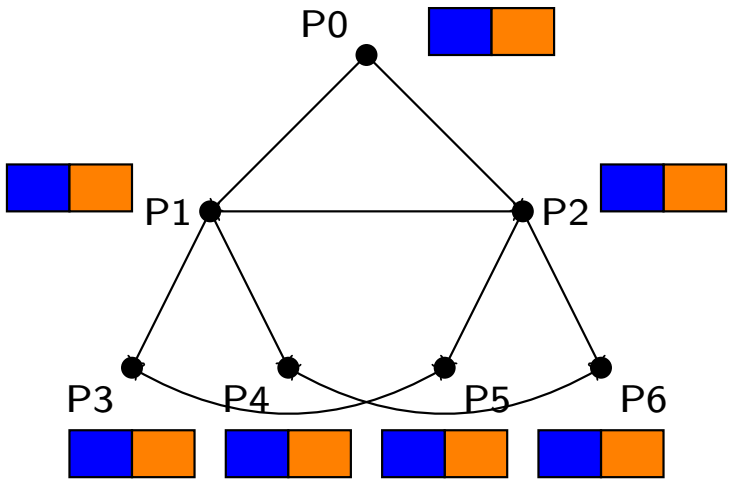
Algorithme de diffusion : split-binary tree



Algorithme de diffusion : split-binary tree



Algorithme de diffusion : split-binary tree



Réduction

Réduction vers une racine

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

Algorithme de réduction : arbre de Fibonacci

Approche naïve : considérer une réduction comme une diffusion inversée

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

Utilisation d'un arbre de Fibonacci

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

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

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

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

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

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 !

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

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`

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

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`

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

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

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

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

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 );
  }
}
```