Cours 3 : (Un peu) d'algorithmique parallèle

Charles Bouillaguet
(charles.bouillaguet@lip6.fr)

2021-02-12

Comment obtenir un programme parallèle efficace ?

Les grands principes

- **localité des données** : *répartir les données de sorte que chaque processeur dispose localement d'un maximum de données à traiter.*
- **équilibrage de charge** (*load balancing*) : *attribuer au mieux les charges de calcul en fonction des caractéristiques de chaque processeur, afin de limiter les périodes d'inactivité.*
- **recouvrement des communications par le calcul** : *éviter que les processeurs restent passifs pendant les phases de communications.*

Équilibrages de charge

Charge de calcul prédictible

$\Rightarrow$ équilibrage de charge **statique**.

- Toutes les données nécessitent le même temps de calcul.
  $\rightarrow$ distribution bloc, cyclique, ...
- Données régulières présentant des temps de calcul différents
  $\rightarrow$ utilisation d'une fonction de coût + distribution bloc, cyclique, ...

Charge de calcul non prédictible

$\Rightarrow$ équilibrage de charge **dynamique**

- modèle ~~maître-esclave~~ patron-ouvrier
- modèle auto-régulé

Modèle patron-ouvrier (maître-esclave)

- Le patron connaît les données et le travail.
- Le patron envoie aux ouvriers des ordres de travail ou un ordre de fin.

Limites

- Si le patron doit charger toutes les données, il lui faut beaucoup de RAM.
- 2 envois de messages par tâche (A/R) $\longrightarrow$ grosse granularité.
- Trop d'ouvriers $\rightarrow$ le patron peut être un goulet d'étranglement.

Avantages :

- l'équilibrage de charge peut se faire en fonction de l'hétérogénéité du matériel, même si elle varie dans le temps.
- *checkpointing* assez facile.

Modèle auto-régulé

Principe du « vol de travail » (*work stealing*) :

- chaque processeur gère sa propre liste de travaux à effectuer ;
- si la liste de travail d'un processeur est vide, il récupère une partie de la liste des autres processeurs.
  - \+ meilleure gestion mémoire
  - \+ tous les processeurs participent au calcul
  - – pas facile de détecter qu'il n'y a plus de travail
  - – difficulté de programmation

Parallélisme de données

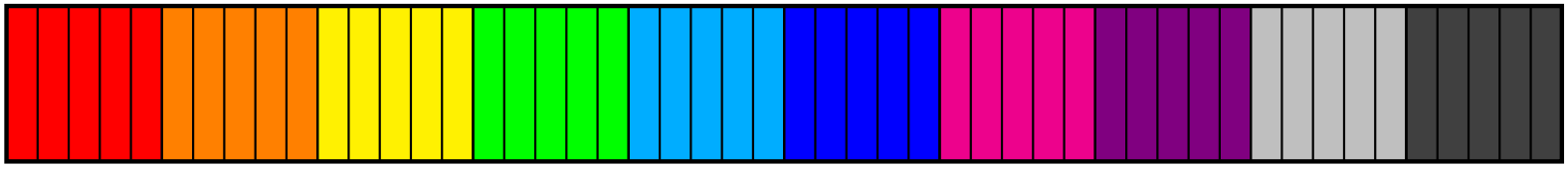
Exemple classique : *map*

```
for (int i = 0; i < n; i++)
    B[i] = f(A[i], i)
```

- Répartition des données ?
- Équilibrage de charge

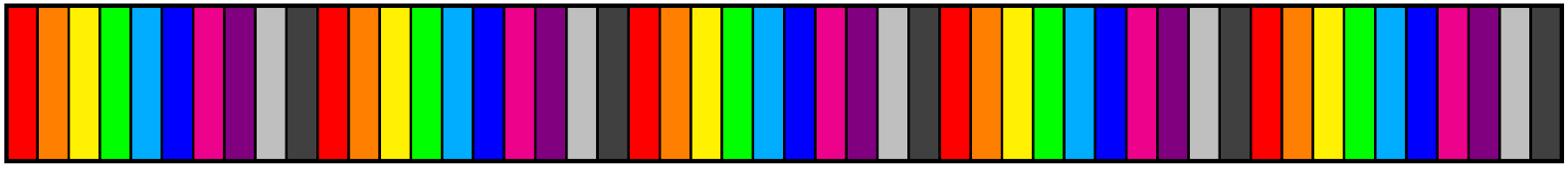
Répartition 1D

Par blocs :



- Le plus simple !
- Favorisé par MPI

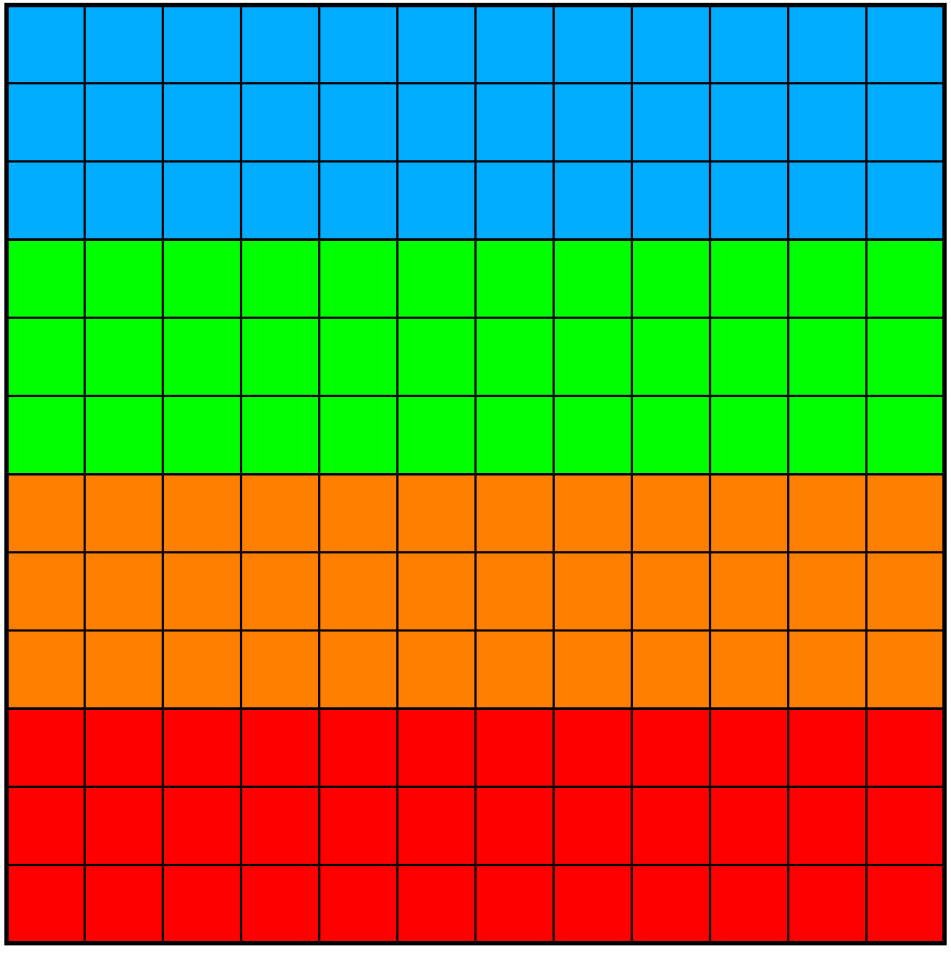
Cyclique :



- Améliore parfois l'équilibrage de charge.
- Possible aussi avec MPI (`MPI_Type_vector`...)

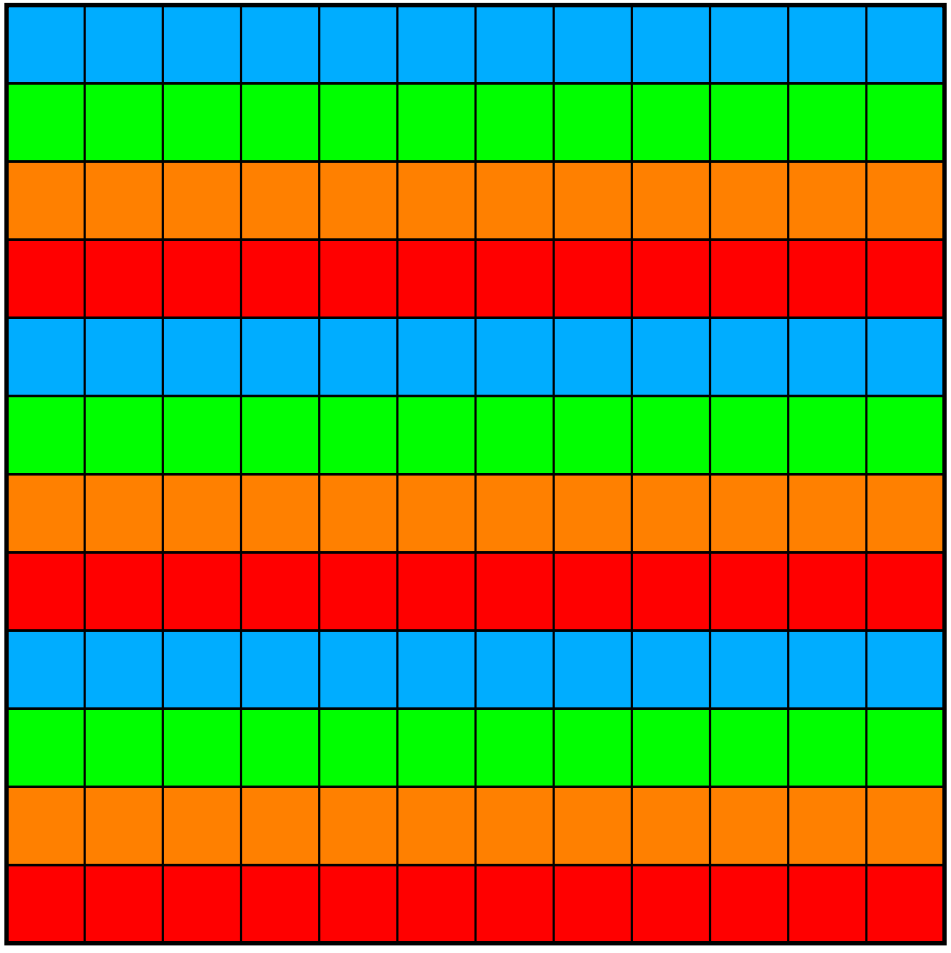
Répartition 1D (de données 2D)

Par blocs :



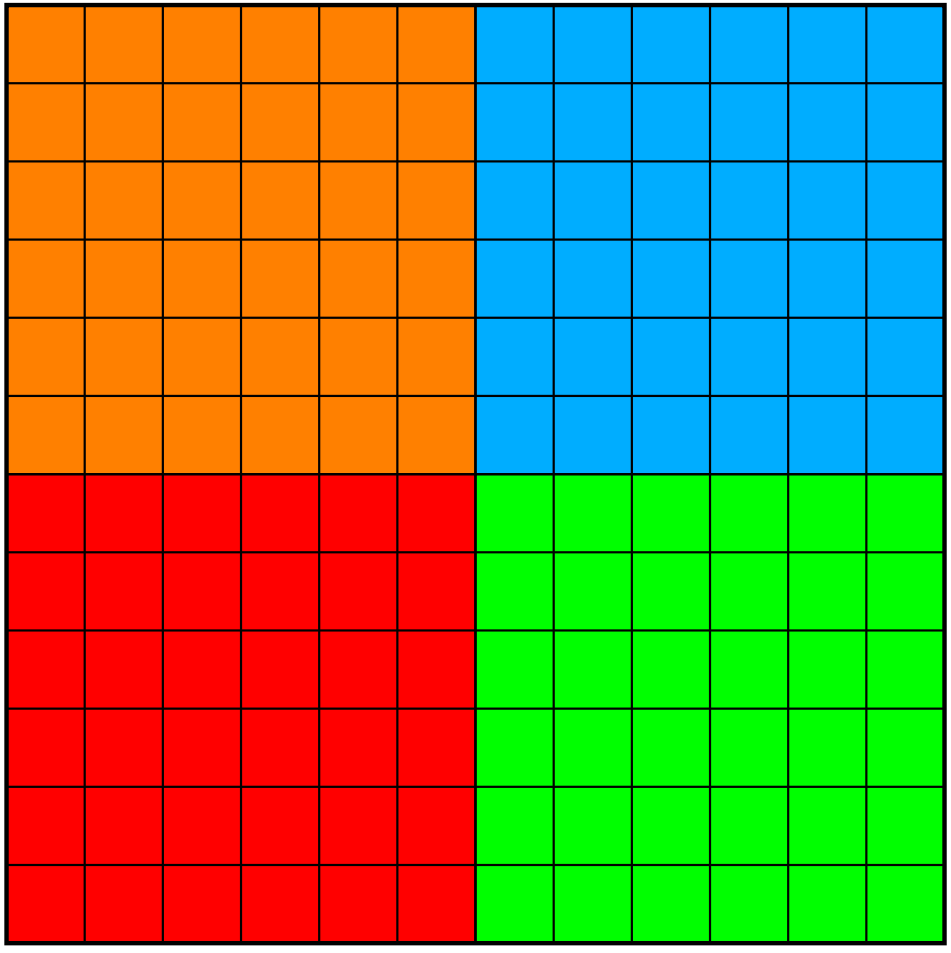
Répartition 1D (de données 2D)

Cyclique :



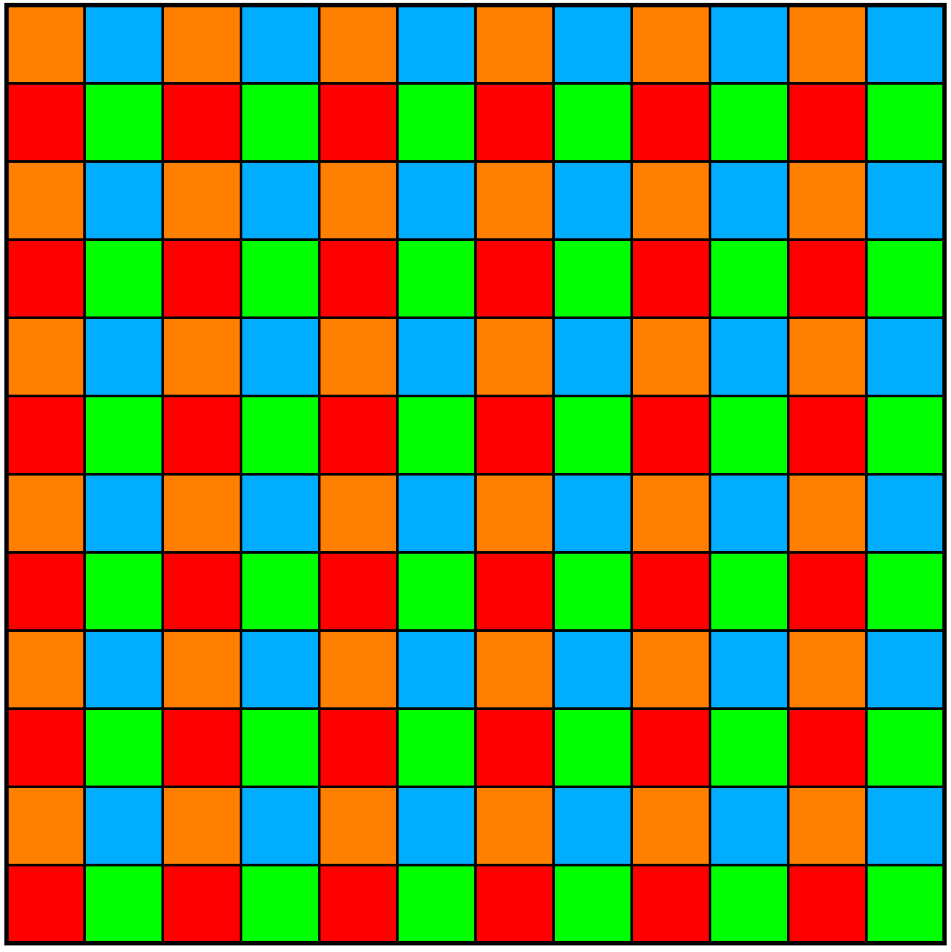
Répartition 2D (de données 2D)

Par blocs :



Répartition 2D (de données 2D)

Cyclique :



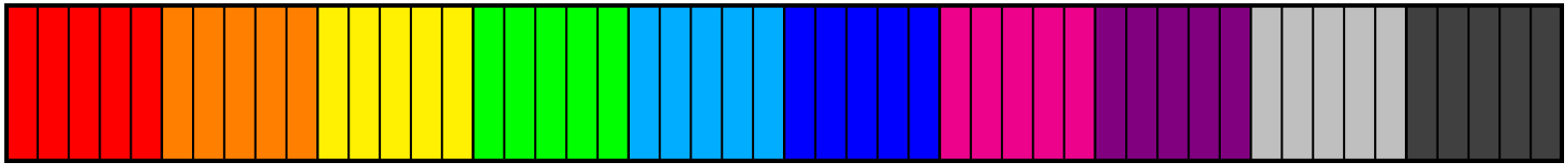
Parallélisme de données

Exemple classique : *reduce*

```
sum = 0
for (int i = 0; i < n; i++)
    sum = sum + A[i]
```

- Mémoire distribuée $\rightarrow$ communications
- Algorithme « classique » en arbre.

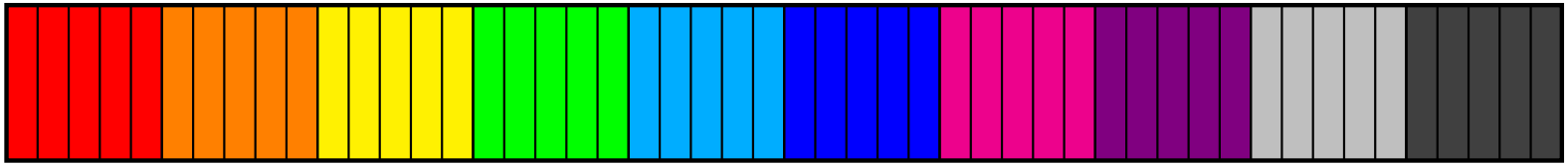
Algorithme (mémoire partagée) pour `reduce`



1. Tableau `Scratch` de taille p.
2. P_i fait : `Scratch[i]` $\leftarrow$ somme de *ses* donnée.
3. Barrière
4. P_0 calcule la somme de `Scratch` puis l'écrit dans `sum`
5. Barrière

$$T = \frac{n}{p} + p$$
$$\geq 2\sqrt{n} \qquad \text{(optimal atteint avec } \sqrt{n} \text{ processeurs)}$$

Algorithme (mémoire partagée) pour `reduce`



`reduce`(A, n) :

1. Si $n = 1$, renvoyer $A[0]$.
2. Allouer un tableau `Scratch` de taille $n/2$.
3. Pour tout $0 \leq i < n/2$, faire (en parallèle) :
   - `Scratch`$[i] \leftarrow A[2i] + A[2i+1]$.
4. renvoyer : `reduce`(`Scratch`, $n/2$).

$$
\begin{aligned}
T &= \frac{n}{p} + \log_2 p \\
&\geq 1 + \log_2 p \qquad \text{(optimal atteint avec } n/2 \text{ processeurs)}
\end{aligned}
$$

Durée des communications ?

Envoi d'un message du rang *i* au rang *j*

$$T = \alpha + n \times \beta$$

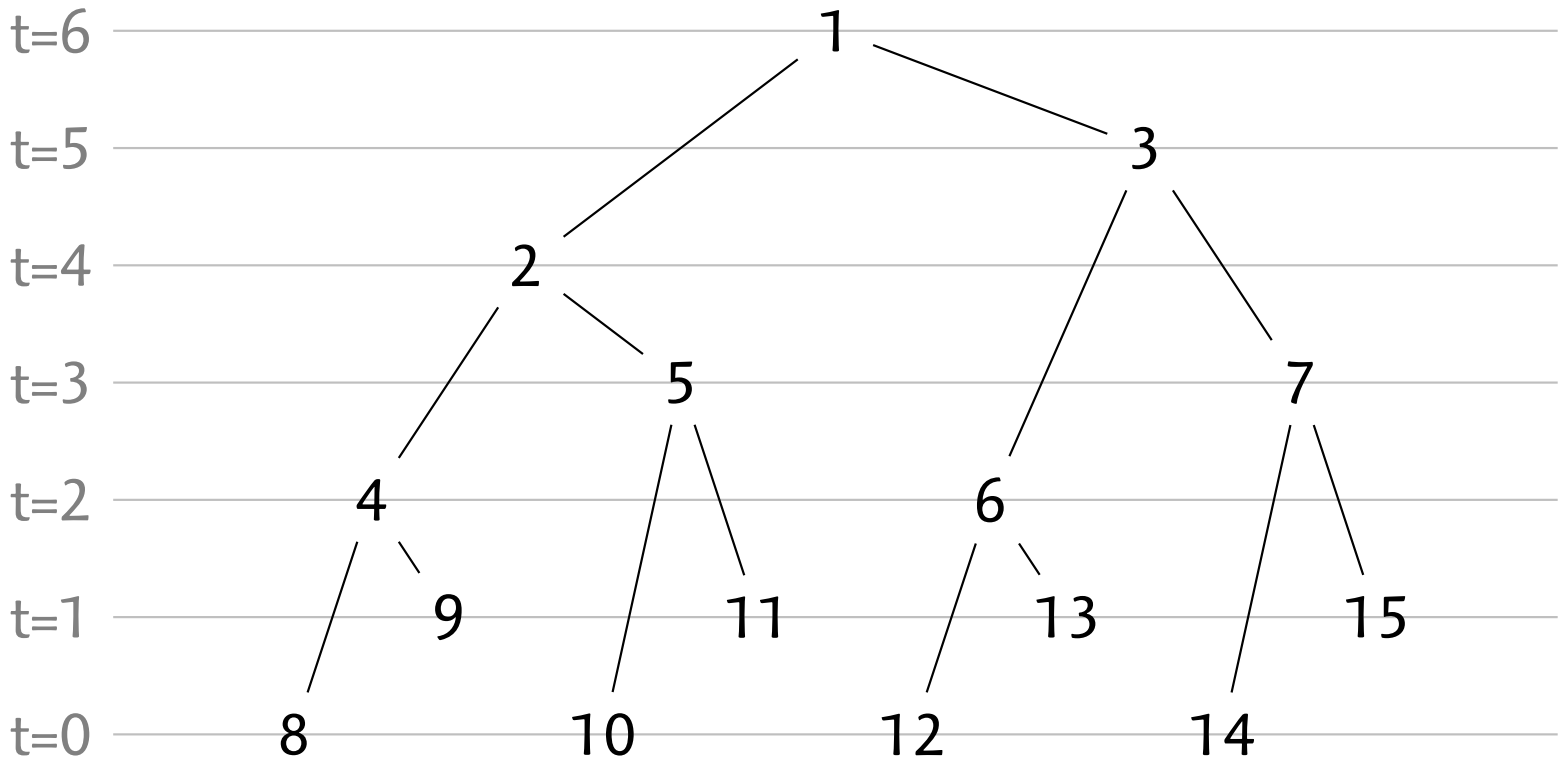
- $n =$ taille des données (bit)
- $\alpha =$ latence (s)
- $\beta \approx$ débit (s / bit).

$1/\beta$: bit / s (bande passante).

Modèle simplifié !

- T indépendant de i et j (topologie ?)
- Indépendant des autres envois (congestion ?)

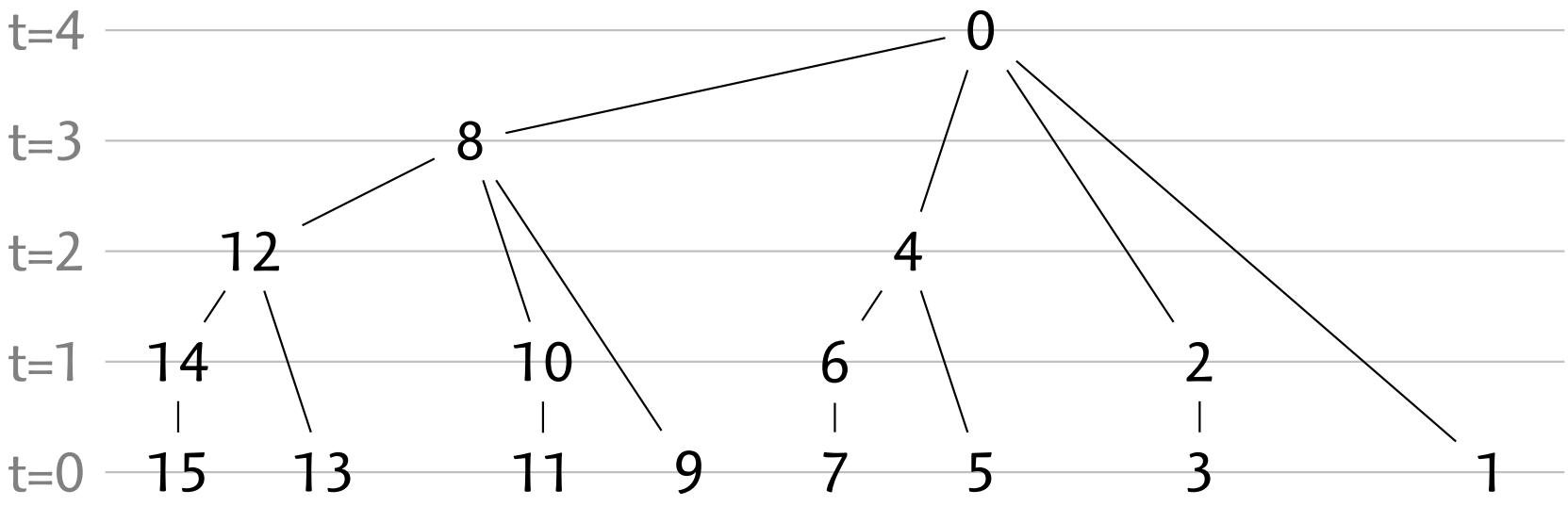
`reduce` : méthode de l'arbre binaire



- Contribution de $2^h - 1$ reçue par 1 après $2(h-1)$ messages successifs

$\Rightarrow$ $T \geq 2\alpha(\lceil \log_2 P \rceil - 1)$

`reduce` : méthode de l'arbre binomial



- Toutes les contributions reçues après h envois successifs.

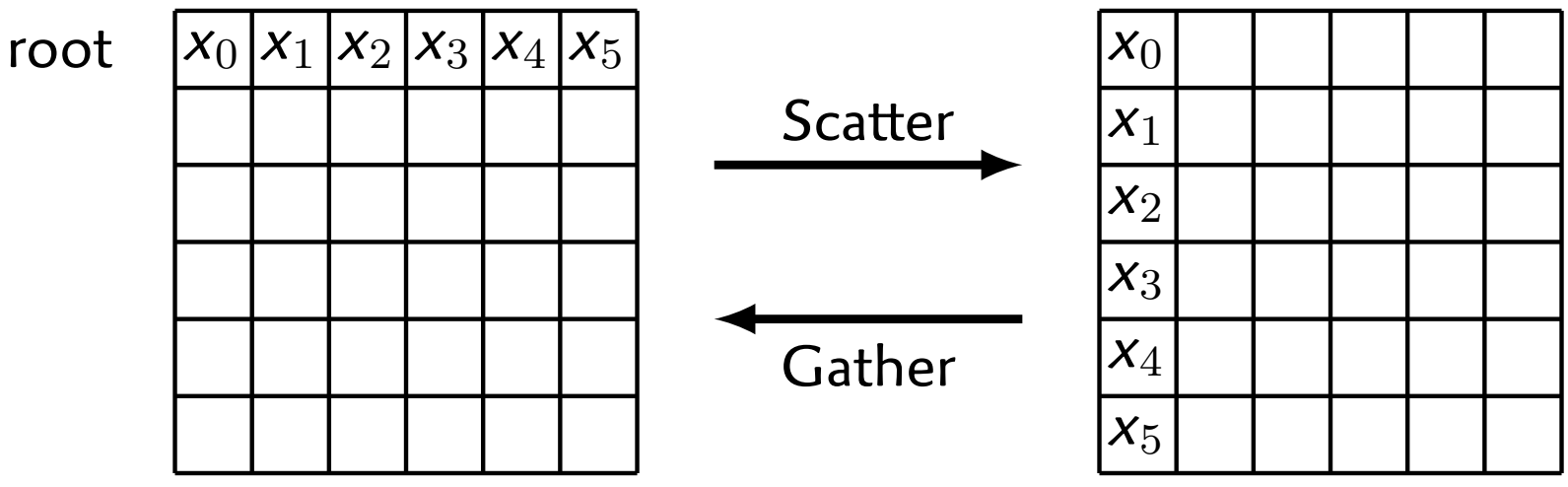
$\Rightarrow$ $T \geq \alpha h$.

`reduce`

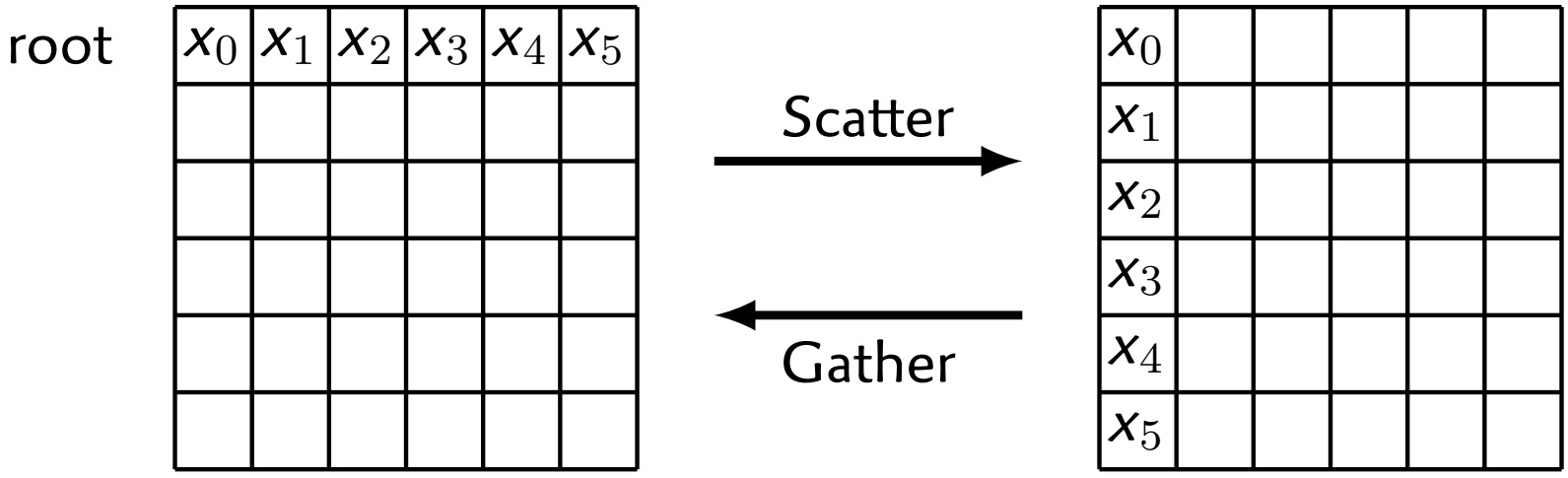
$T =$ envoi de $\lceil \log_2 P \rceil$ messages de taille 1 $= \lceil \log_2 P \rceil (\alpha + n\beta)$

Retour sur MPI : Gather / Scatter / Reduce

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



Borne inférieure pour Scatter/Gather

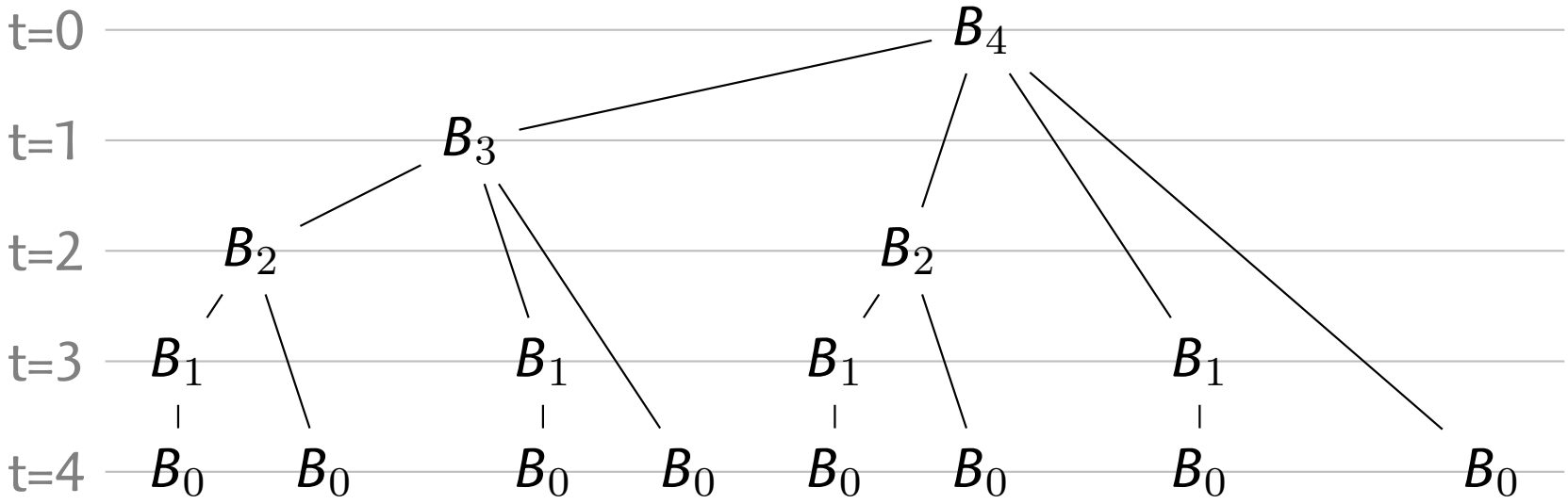


Sont inévitables...

1. $(p-1)\frac{n}{p}$ données doivent sortir (resp. entrer) de `root`.
   - $T \geq (p-1)\frac{n}{p}\beta$
2. Atteindre tout le monde.
   - $\log_2 p$ messages successifs (« **épidémie** »)

$\Rightarrow$ $T \geq \lceil \log_2 p \rceil \alpha + (p-1)\frac{n}{p}\beta$

Méthode de l'arbre binomial



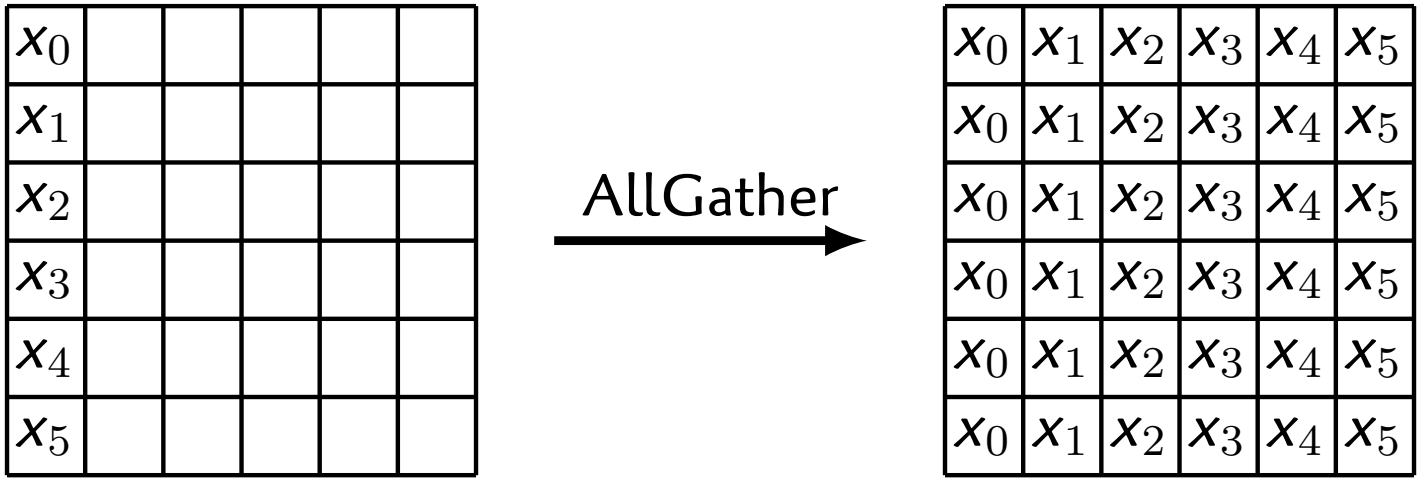
- Tous les B_i sont contactés en même temps
- Chaque B_i reçoit $2^i \frac{n}{p}$ données.

Scatter/Gather

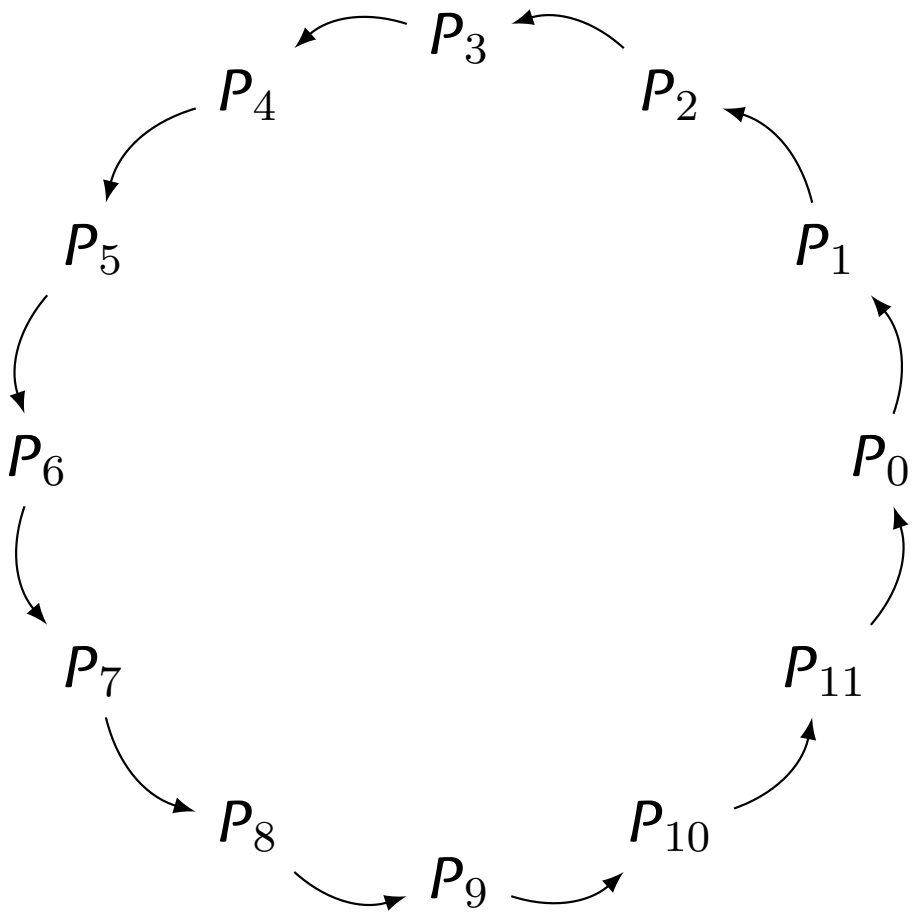
$$T = \text{message de taille } 1 + \cdots + \text{message de taille } n/2$$
$$= \alpha \lceil \log_2 p \rceil + (p-1)\frac{n}{p}\beta \qquad \text{(optimal)}$$

MPI : AllGather

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



AllGather : algorithme de l'anneau



algorithme

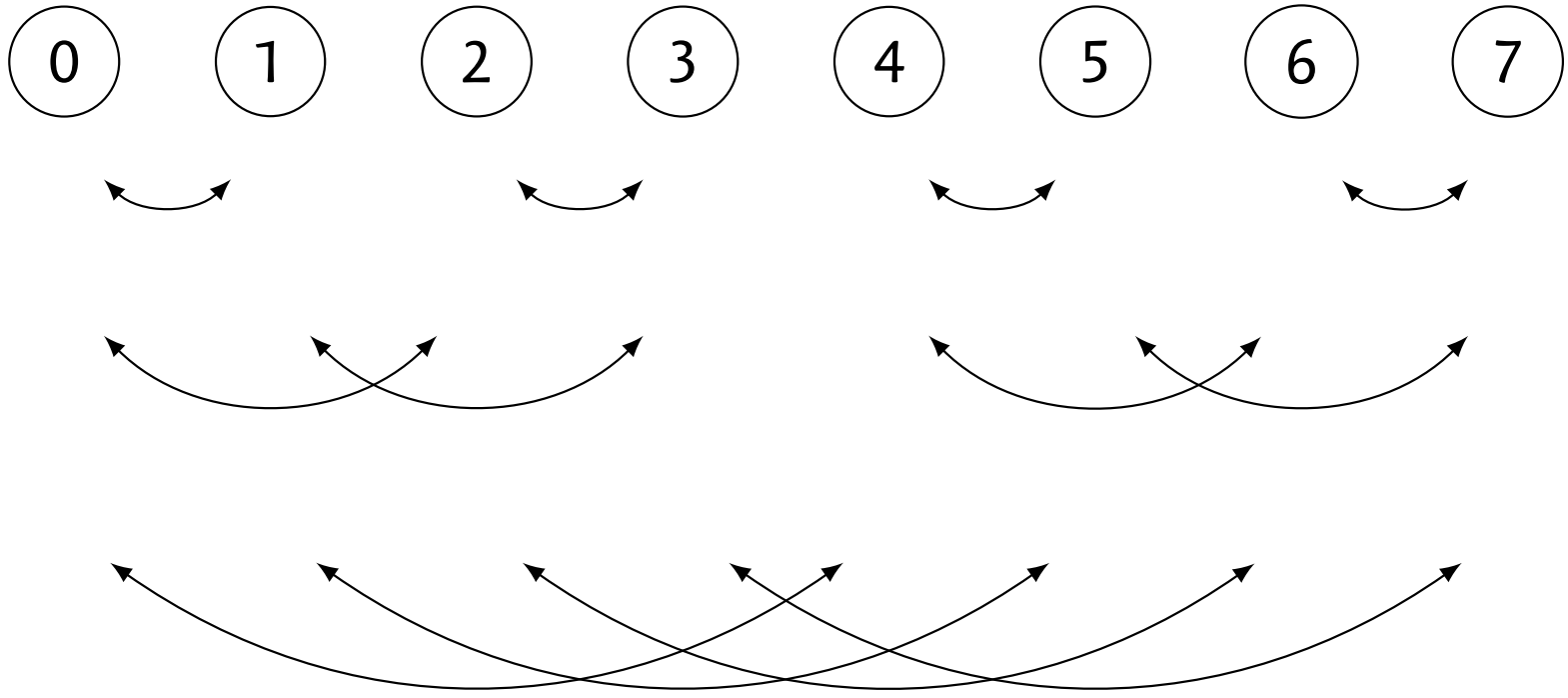
- Phase 0 :
  envoie « ses » données
- Phase $i+1$:
  envoie msg reçu phase i

Analyse

- 1 msg = n/p données
- 1 phase = p messages //
- AllGather = $p-1$ phases

$\Rightarrow T = (p-1)(\alpha + \frac{n}{p}\beta)$

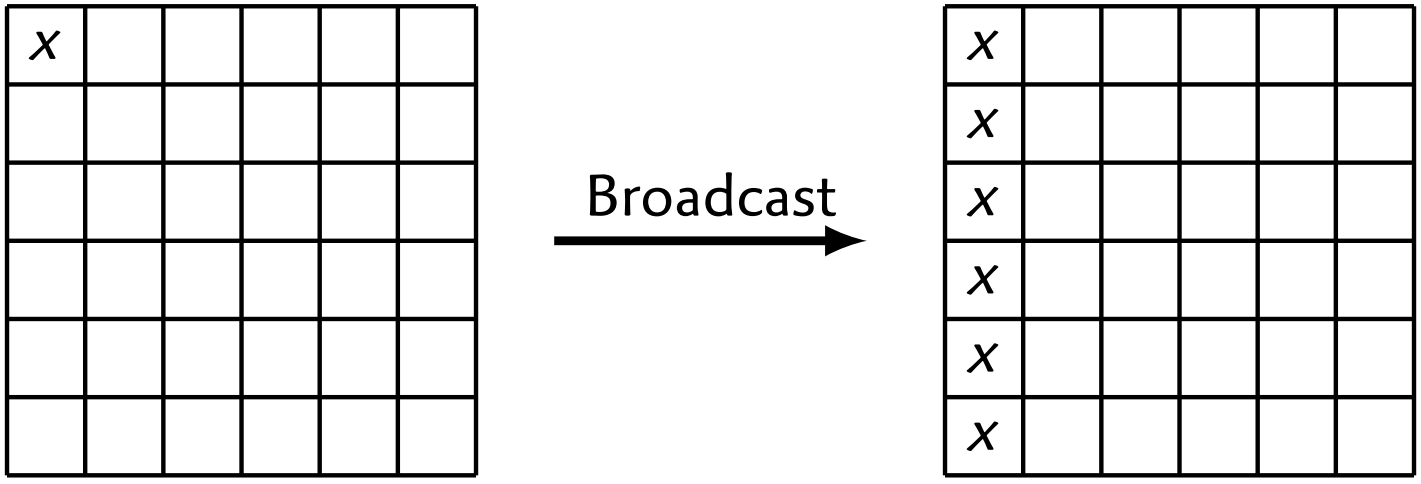
AllGather : algorithme du doublage récursif



$$T = \alpha \log_2 p + (p-1)\frac{n}{p}\beta \qquad \text{(optimal)}$$

MPI : Broadcast

```
int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype,
              int root, MPI_Comm comm);
```



Borne inférieure

1. n données doivent sortir de `root`.
2. Atteindre tout le monde.

$\Rightarrow T \geq \lceil \log_2 p \rceil \alpha + n\beta$

Algorithmes pour Broadcast

Arbre binomial

- $T = \lceil \log_2 p \rceil (\alpha + n\beta)$
- Terme « débit » sous-optimal
- (mauvais pour grands messages)

Van de Geijn

- Broadcast = Scatter puis AllGather (2rec)
- $T = 2(\log_2 p)\alpha + 2\left(1 - \frac{1}{p}\right) n\beta$

$\rightarrow$ $2\times$ borne inf'

Parallélisme de données

Exemple classique : *prefix-sum* (“scan”) (en place)

```
for (int i = 1; i < n; i++)
    A[i] = A[i] + A[i - 1];
```

- **Dépendance de données**
- Chaque itération a besoin du résultat de la précédente...
- Changer l’algorithme

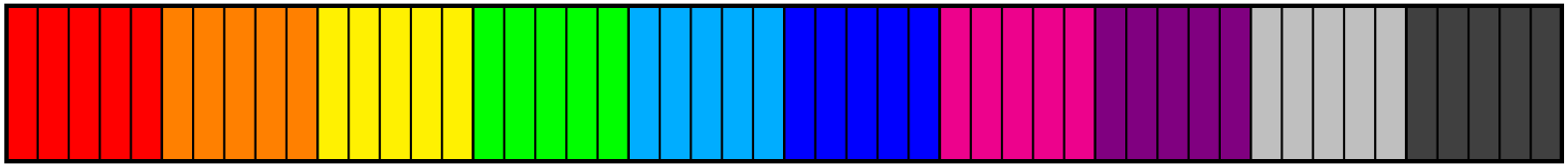
`prefix-sum` : Cas de la mémoire partagée

Exemple classique : *prefix-sum* (en place)

```
double *B;

void prefix_sum(double * A, int n)
{
    if (n < 2)
        return;
    for (int i = 0; i < n / 2; i++)
        B[i] = A[2 * i] + A[2 * i + 1];
    prefix_sum(B, k);
    for (int i = 1; i < n; i += 2) {
          A[i] = B[i / 2];
          A[i + 1] = B[i / 2] + A[i + 1];
    }
}
```

Algorithme MIMD-DM pour `prefix-sum`

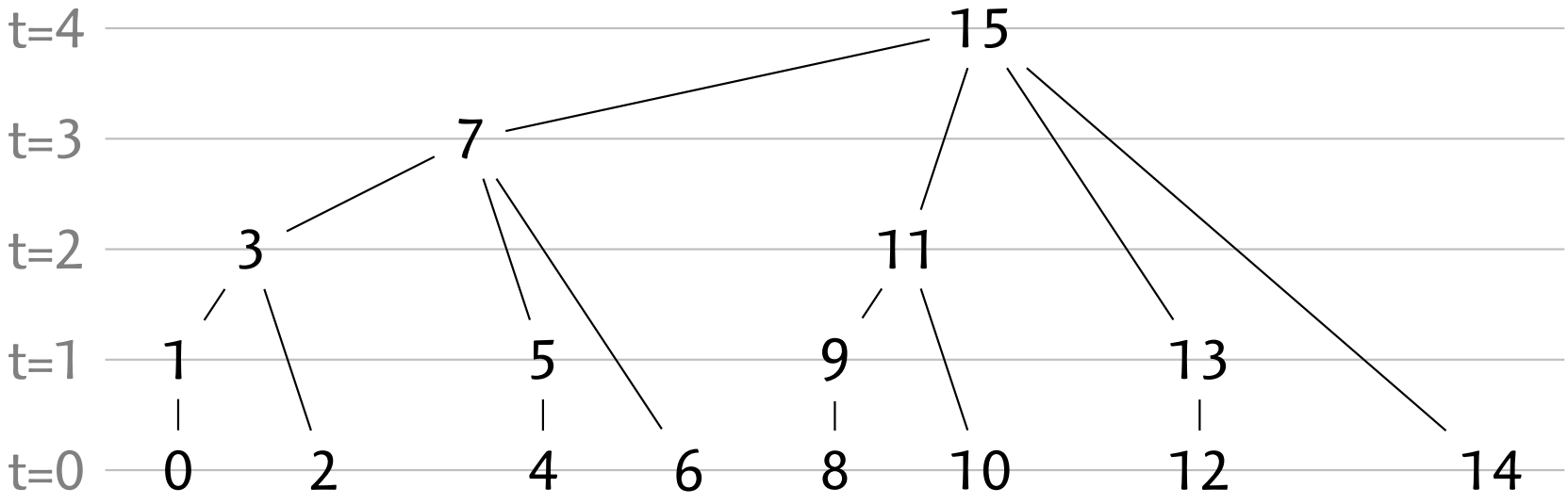


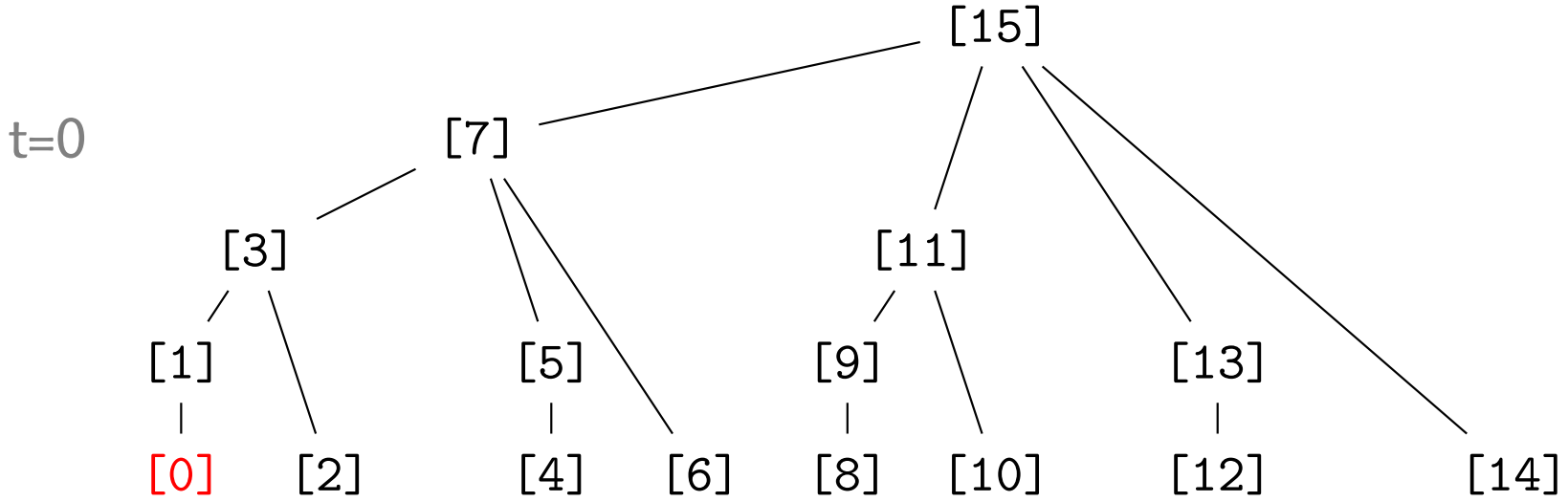
1. P_i calcule la somme S_i de *ses* données. [local]
2. Ils font (collectivement) $T_i \leftarrow$ `prefix-sum`(S).
   - `MPI_Scan`

$\rightarrow$ P_i obtient $T_i =$ somme des données des P_j (pour $j < i$).

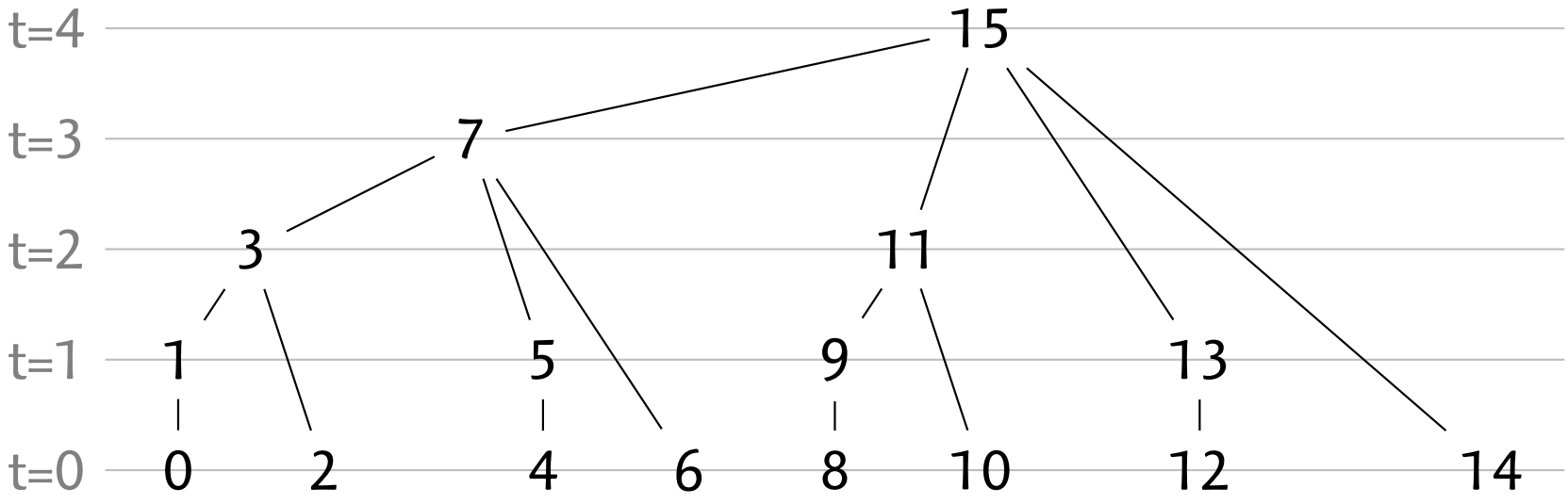
3. P_i `prefix-sum` ses données en ajoutant T_i. [local]

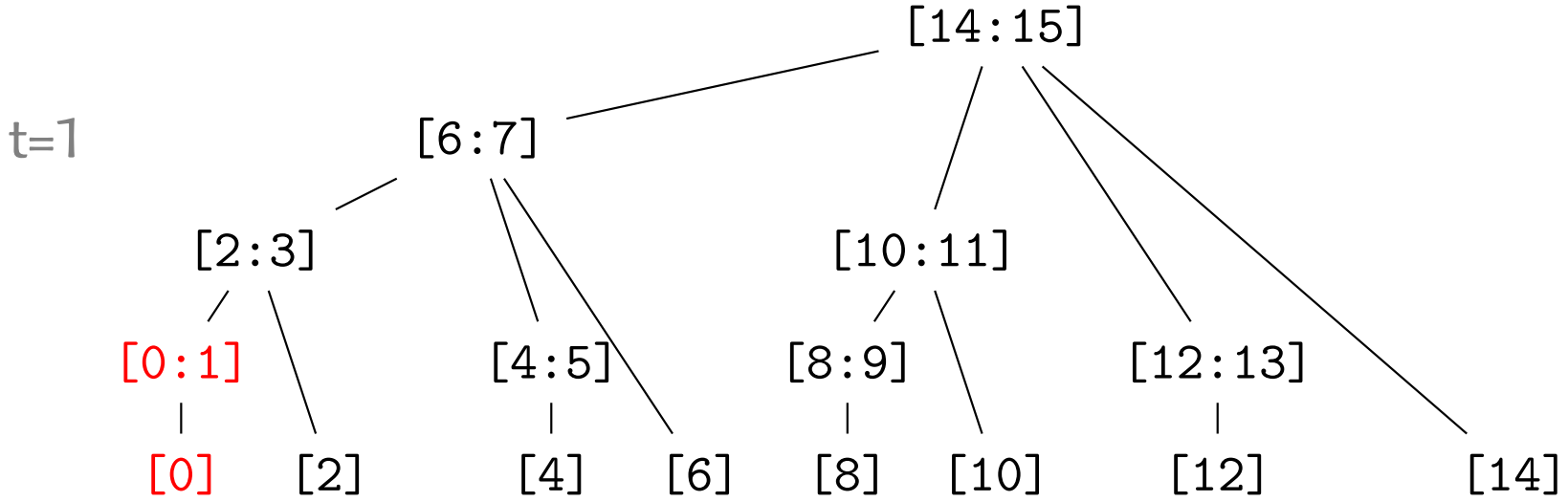
Rappel : `reduce` par la méthode de l'arbre binomial



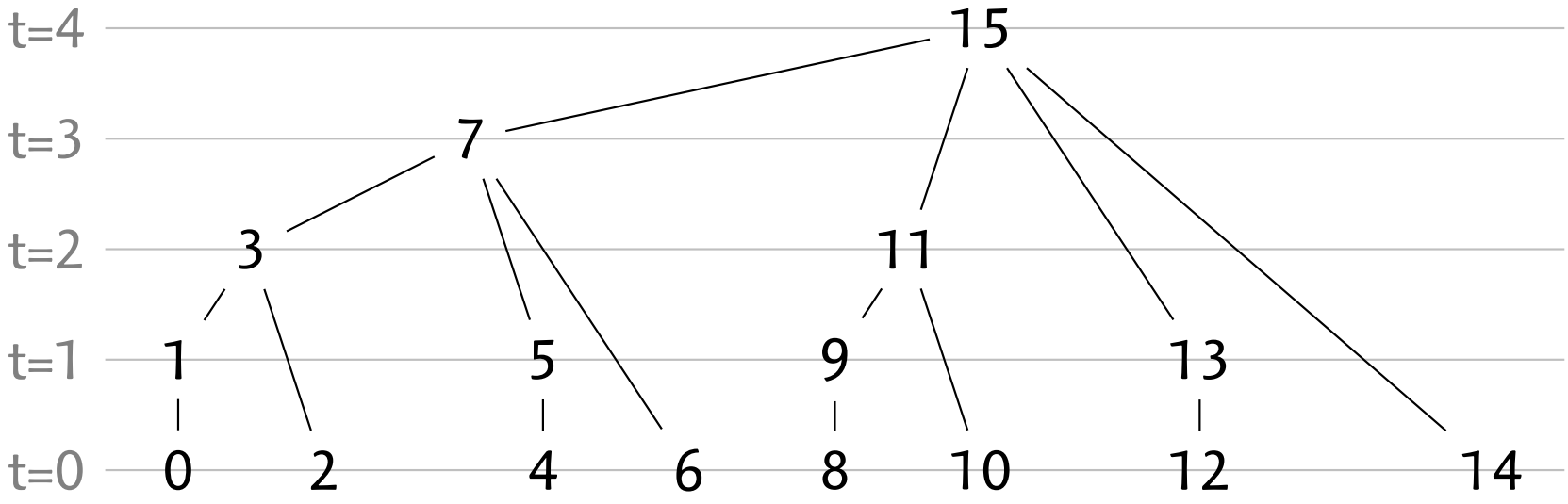


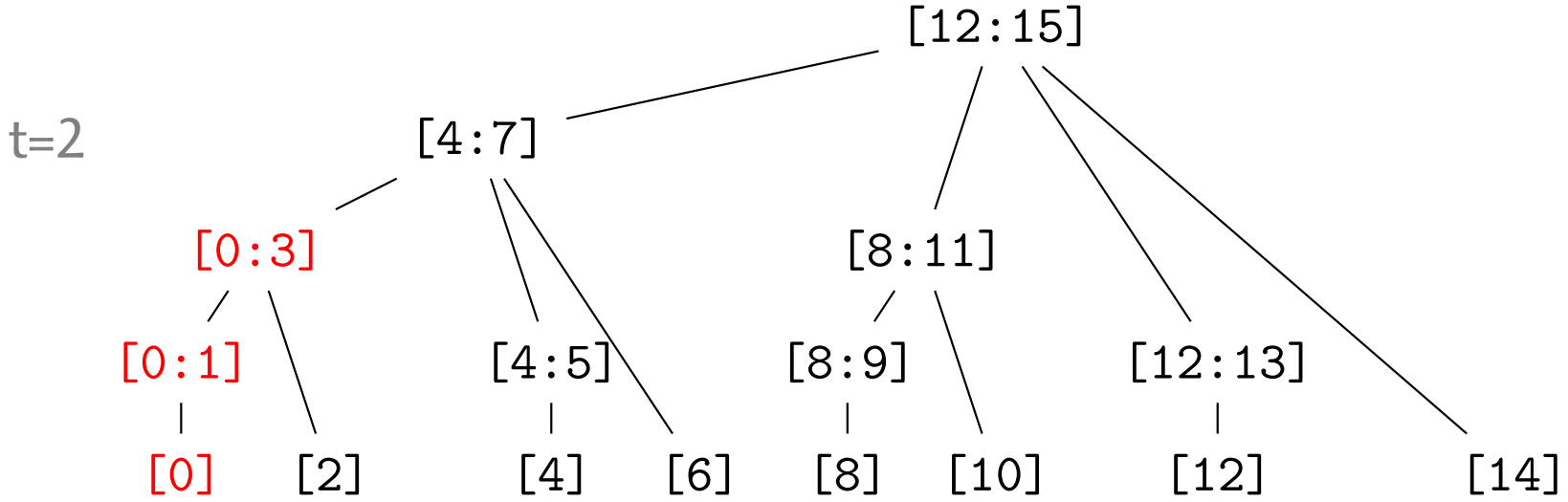
Rappel : `reduce` par la méthode de l'arbre binomial



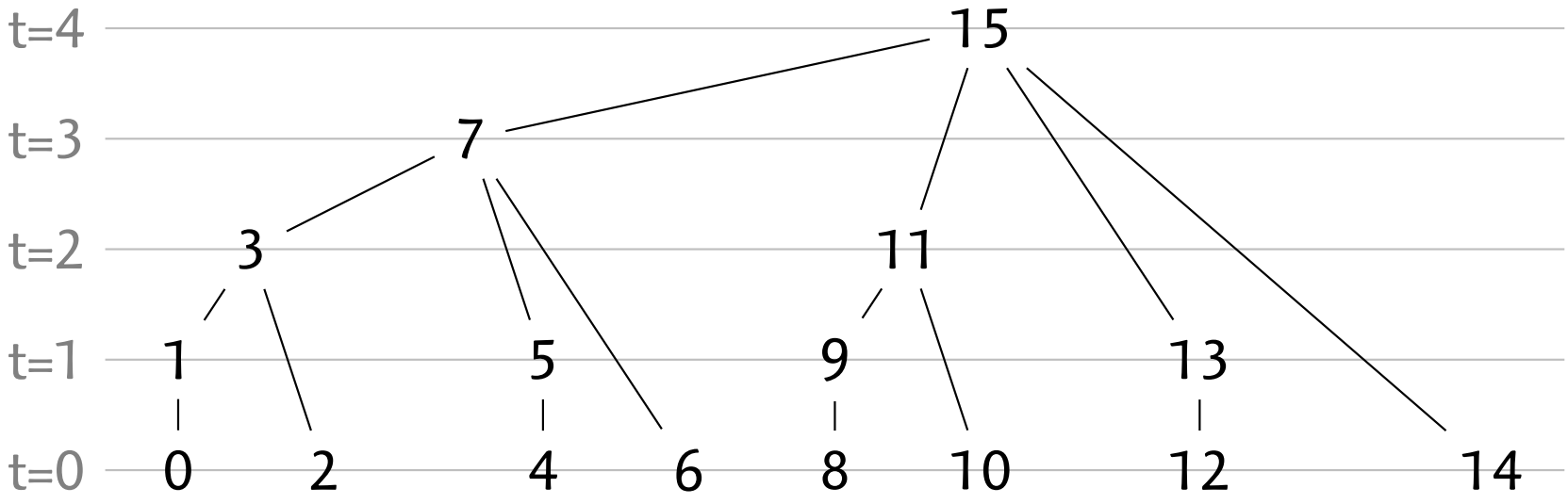


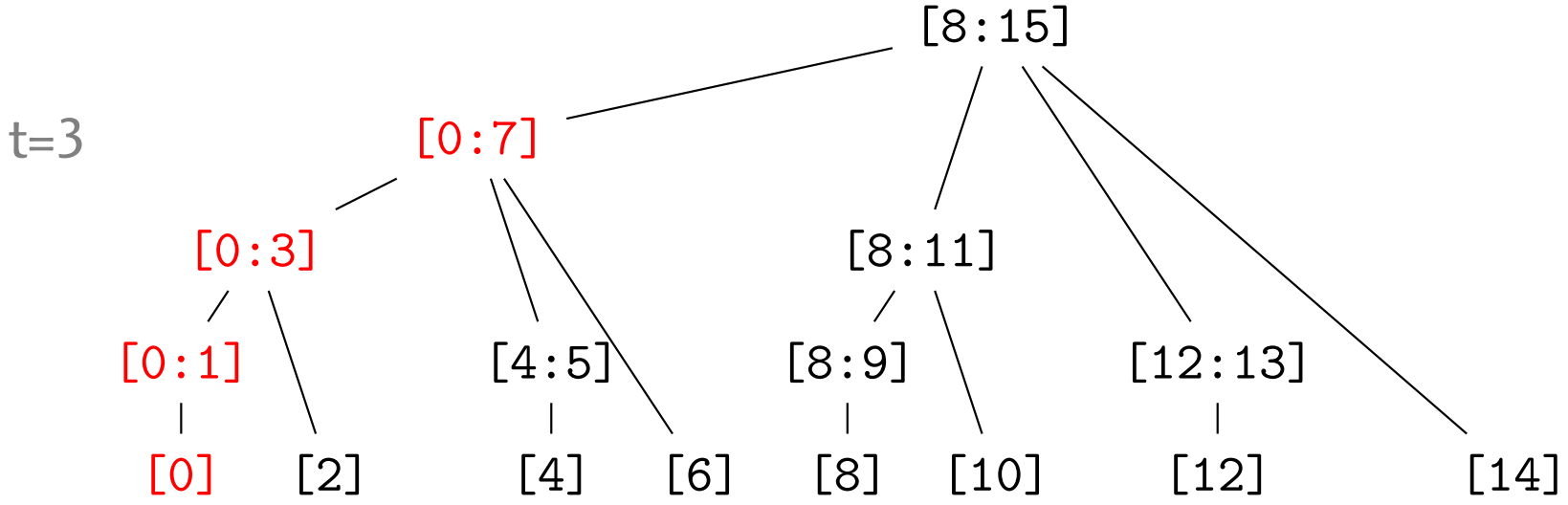
Rappel : `reduce` par la méthode de l'arbre binomial



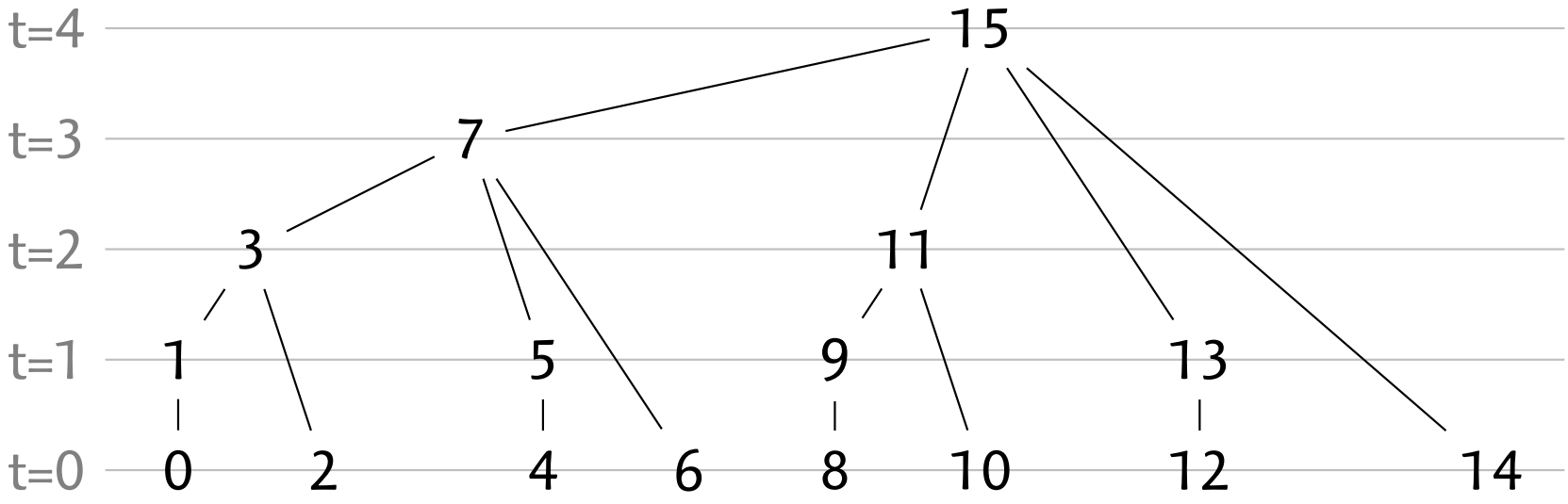


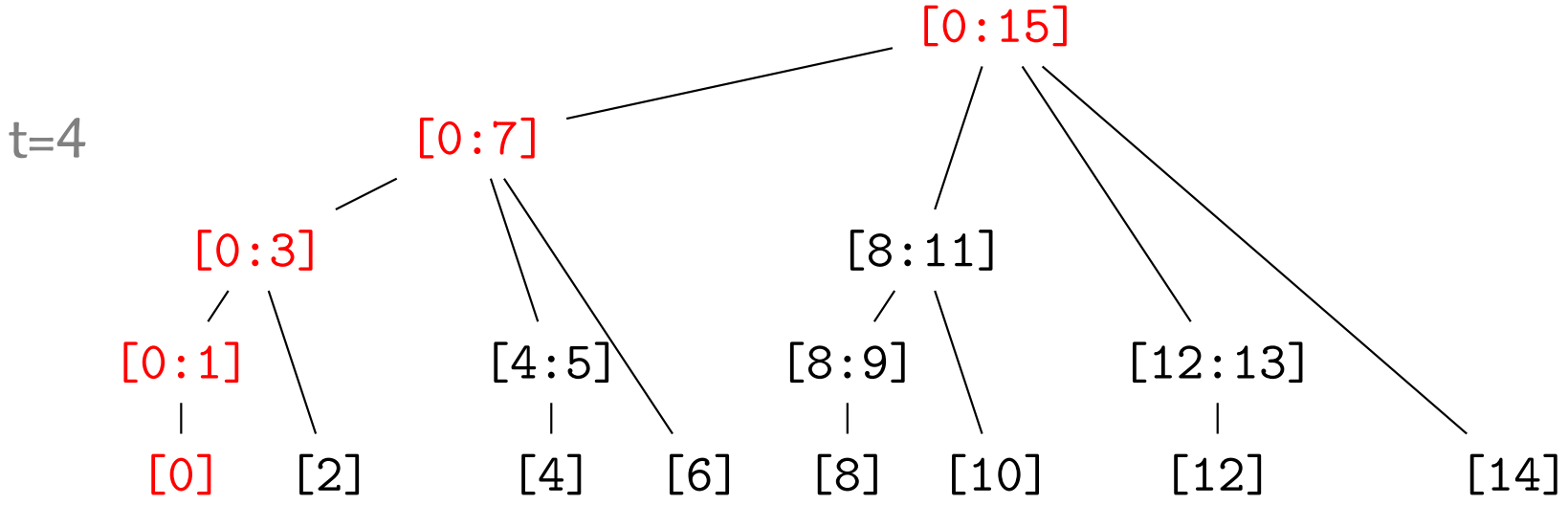
Rappel : `reduce` par la méthode de l'arbre binomial





Rappel : `reduce` par la méthode de l'arbre binomial





`prefix-sum` en mémoire distribuée (arbre binomial)

Phase 1 : `reduce`

Chaque noeud :

1. Récupère (et stocke) les valeurs de ses enfants.
2. Calcule la somme, ajoute sa propre valeur, envoie à son père.

$\rightarrow \log_2$ messages successifs.

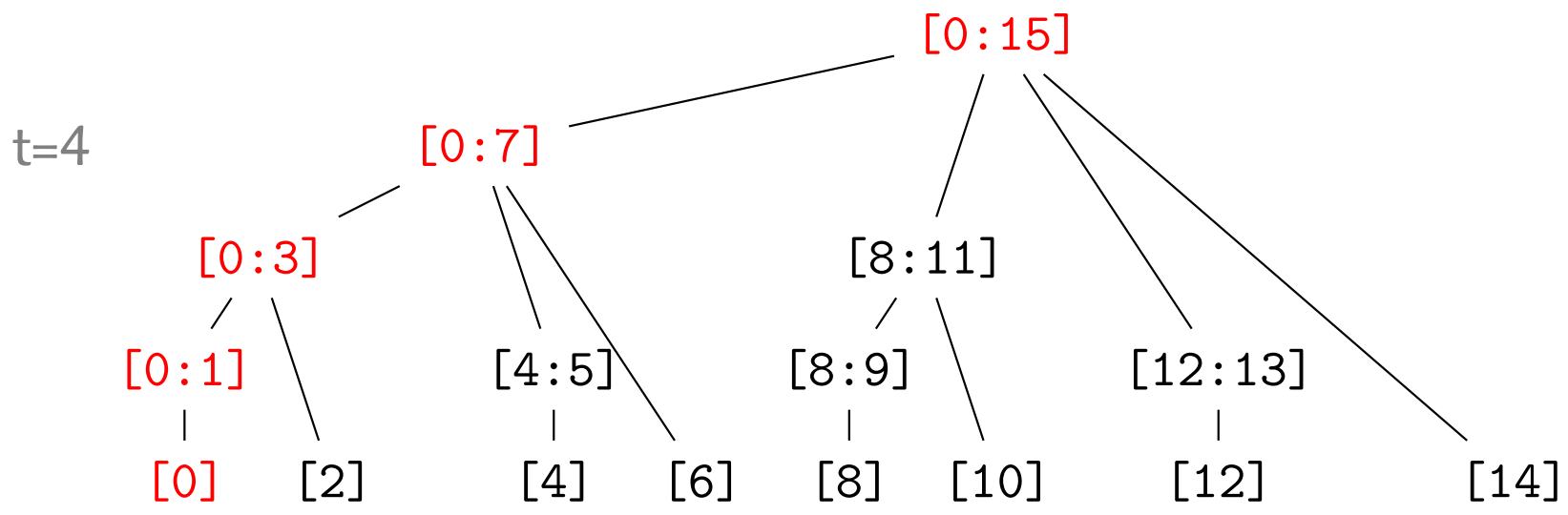
Phase 2 : `prefix-sum`

Chaque noeud :

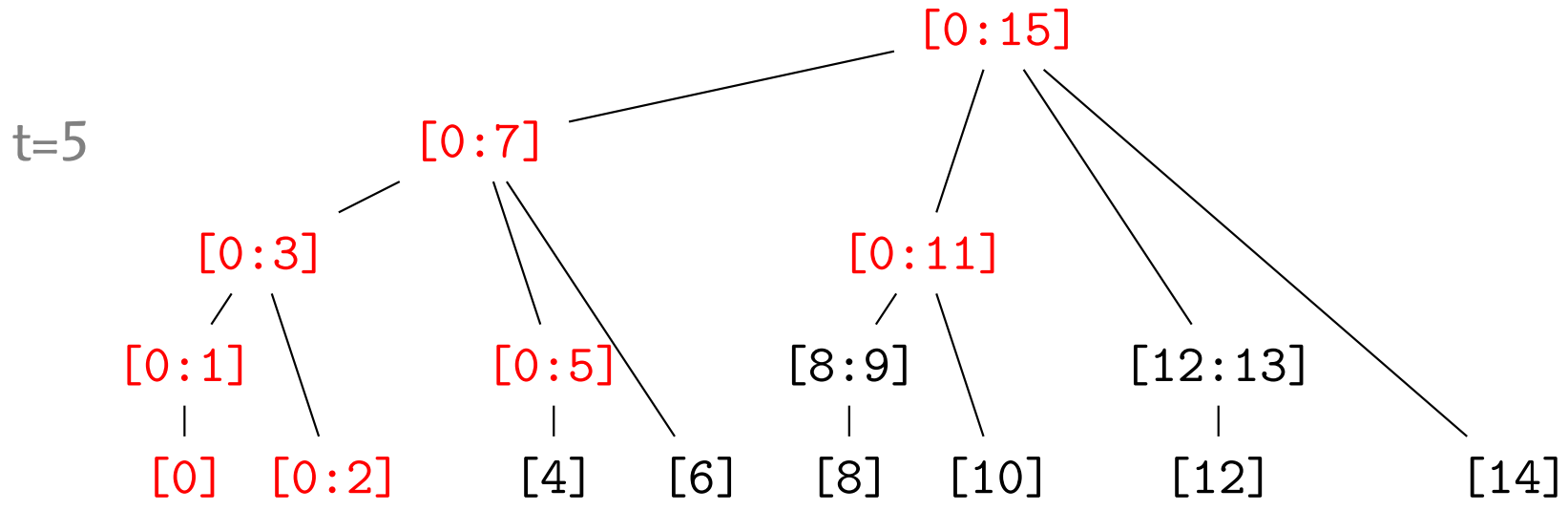
1. Reçoit une valeur de son père.
2. Envoie à chacun de ses fils la somme des valeurs remontées par ses frères *gauches*, plus la valeur reçue du père.

$\rightarrow \log_2$ messages successifs.

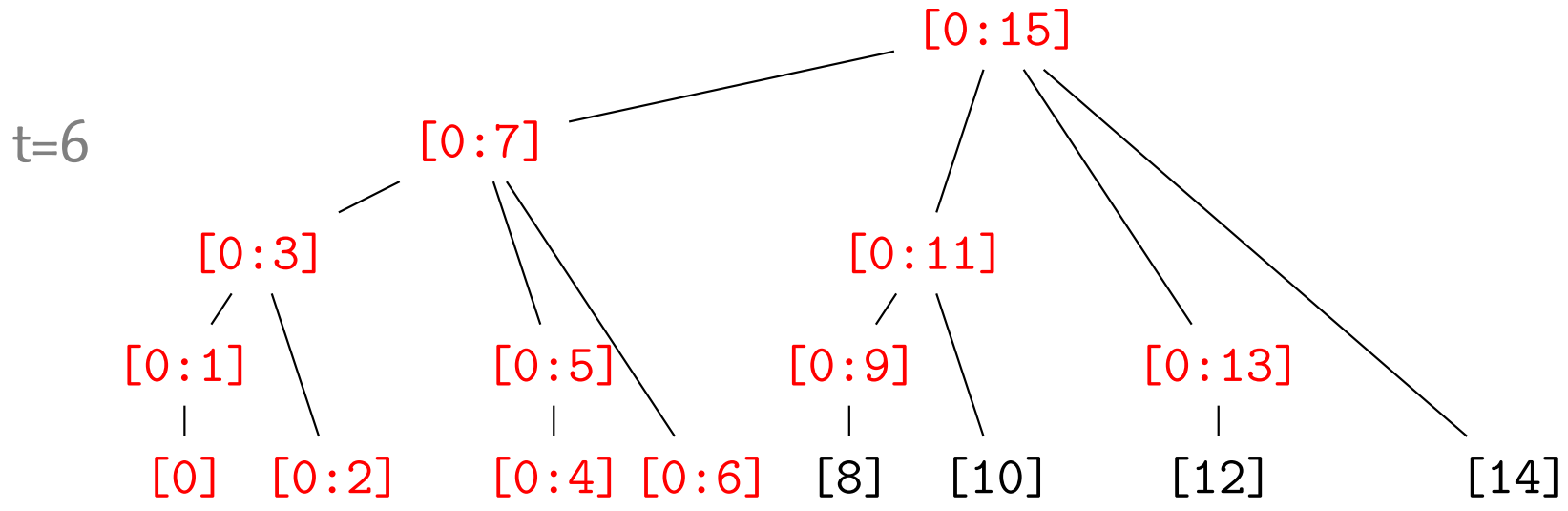
`prefix-sum` par la méthode de l'arbre binomial



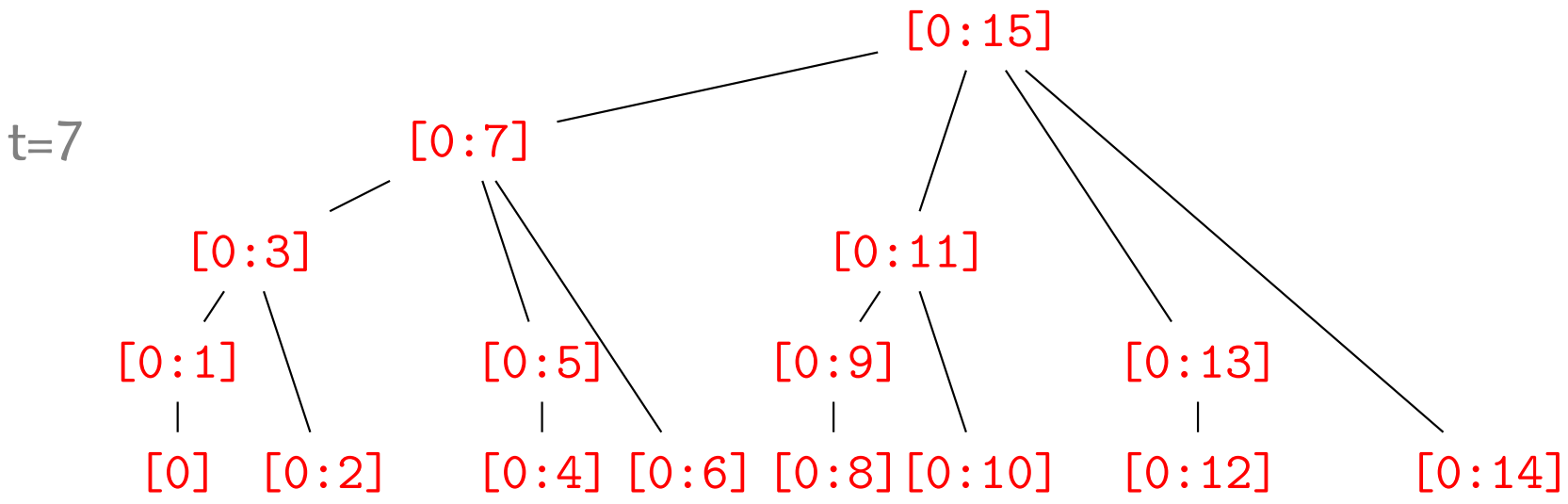
`prefix-sum` par la méthode de l'arbre binomial



`prefix-sum` par la méthode de l'arbre binomial



`prefix-sum` par la méthode de l'arbre binomial



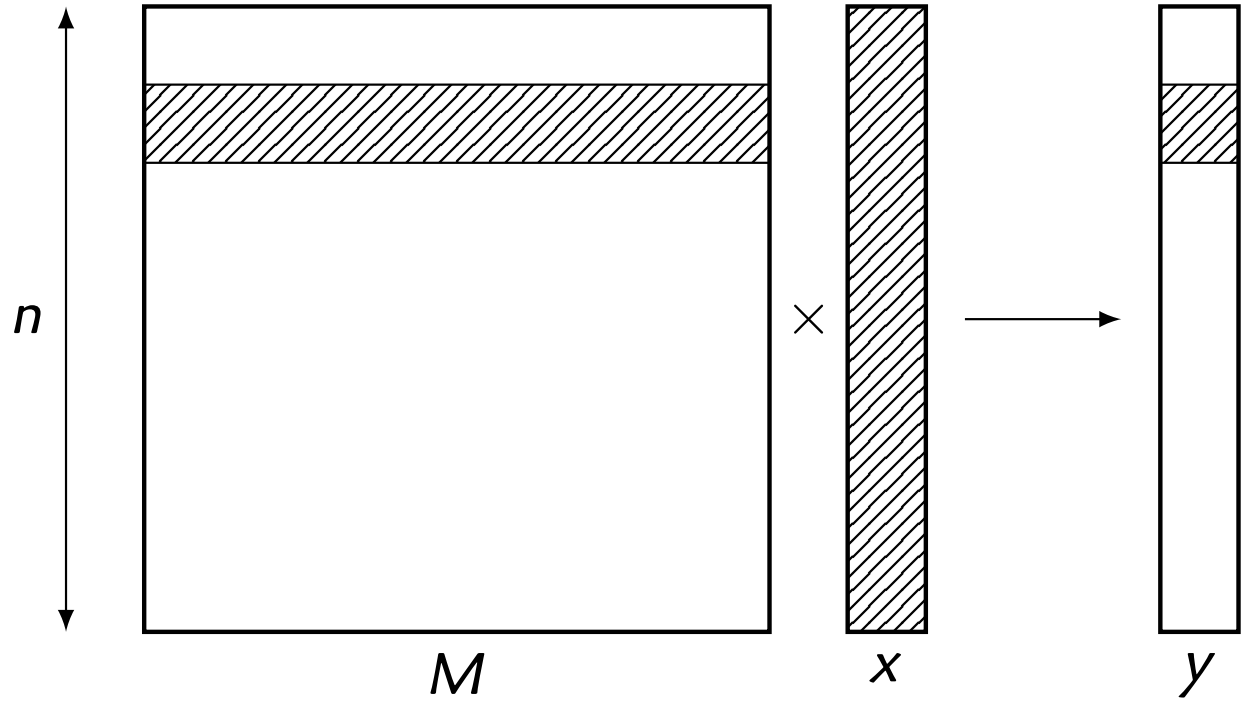
Produit matrice-vecteur (dense)

Répartition

M : Distribution 1D (par lignes)

x : possédé par tous au début

y : possédé par tous à la fin



`MPI_Allgather` pour la distribution correcte de y.

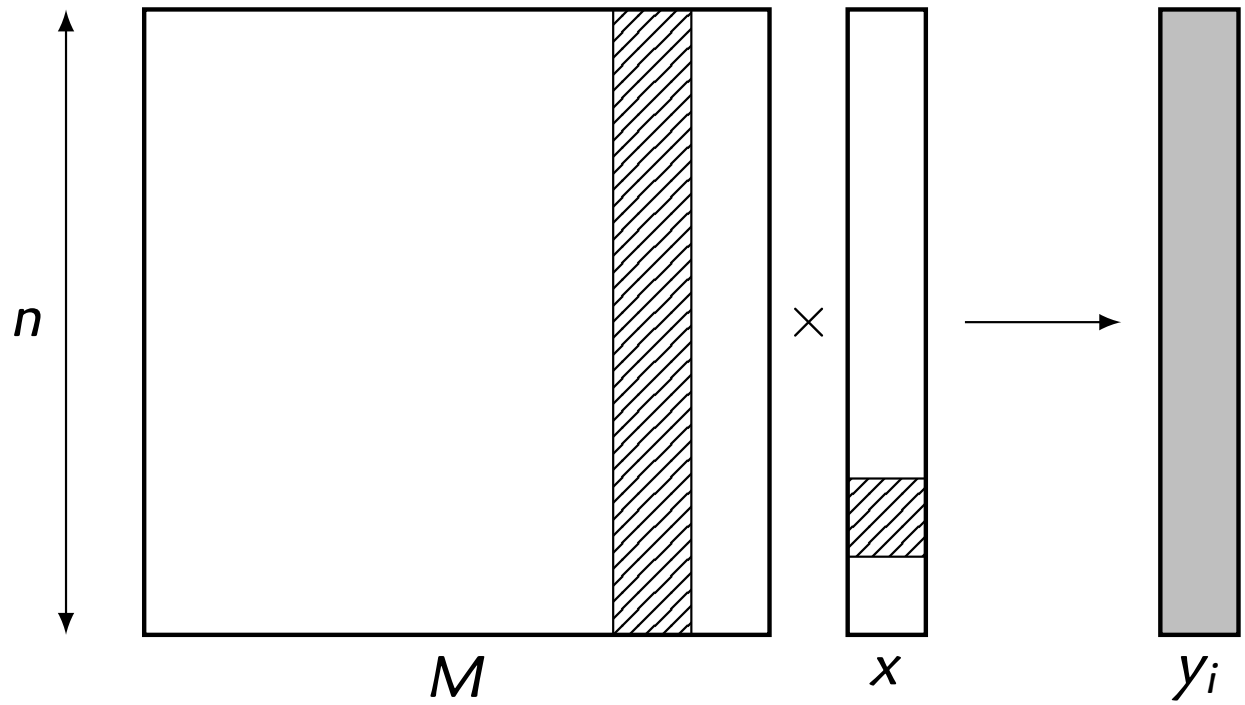
Produit matrice-vecteur (dense)

Répartition

M : Distribution 1D (par colonnes)

x : Distribution 1D au début

y : Distribution 1D à la fin



`MPI_Reduce_Scatter` pour la distribution correcte de y.

Produit matrice-vecteur (dense)

Répartition

M : Distribution 2D (par blocs)

x : Distribution 1D au début

y : Distribution 1D à la fin