Algorithmique Parallèle

Sylvain Contassot-Vivier

Université de Lorraine, LORIA, France

Introduction

Introduction

Les ordinateurs permettent de résoudre efficacement de nombreux problèmes

Cependant, la demande est toujours plus importante quant à la complexité et la taille des problèmes à traiter !

⇒ Apparition très tôt du concept de *parallélisme* :

- Exécuter plusieurs tâches *en même temps*
- Existe à plusieurs niveaux :
  - Processeurs : unités de calcul, pipelines, multi-cœurs
  - Machines : machines parallèles et stations multi-processeurs (SMP)
  - Au-delà : grappes de machines (COW/NOW), grilles
- *Double intérêt* :
  - Augmentation de la *puissance de calcul* (temps de calcul)
  - Augmentation de la *capacité de stockage* (taille de problème)

Plan

- **Modèles de parallélisme et systèmes parallèles**
  - Classifications de Flynn, Systèmes parallèles
- **Évaluation du parallélisme**
  - Accélération, travail, efficacité,...
- **Modèles de programmation parallèle**
  - Parallélisme de données, de contrôle, de flux
- **Algorithmique parallèle**
  - Programmation des systèmes à mémoire partagée
  - Programmation des systèmes à mémoire distribuée
- **Équilibrage de charge**
- **Programmation GPU**

Modèles de parallélisme

Plusieurs classifications possibles selon les critères utilisés

Celle de *Flynn* est sans doute la plus communément utilisée (instruction / données) :

- *SISD* (Single Instruction Single Data) :
  - Systèmes à processeurs scalaires séquentiels
- *SIMD* (Single Instruction Multiple Data) :
  - Systèmes à processeurs vectoriels : opérations sur vecteurs
  - Grand nombre de petites unités de calcul travaillant simultanément
- *MISD* (Multiple Instruction Single Data) :
  - Pas/peu de réalisations, champs d'application trop réduit
- *MIMD* (Multiple Instruction Multiple Data) :
  - *SPMD* (Single Program Multiple Data) : le plus utilisé
  - *MPMD* (Multiple Program Multiple Data) : algorithmes collaboratifs, couplage de code...

Systèmes parallèles

Classification basée sur la dimension physique :

- Machines mono-processeur :
  - Stations de travail, PCs mais aussi machines à processeur vectoriel
- Machines parallèles :
  - Machines multi-processeurs à mémoire partagée ou distribuée
- Grappes locales :
  - Machines indépendantes connectées via un réseau local
- Grappes distribuées ou grilles :
  - Machines de l'un des types précédents reliées entre elles par le réseau global (Internet)

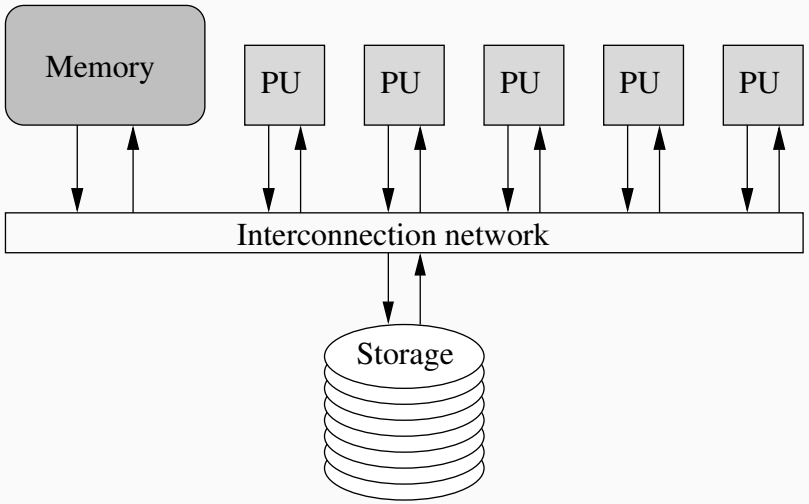
Le *type de mémoire* est important :

- *Partagée* ou *Distribuée*
  $\Rightarrow$ Influence directe sur la façon de programmer

Systèmes à mémoire partagée

Architecture :

- Unités de calcul reliées à une *mémoire commune unique*



Avantages :

- Pas de distribution des données
- Échanges d'informations entre les unités via la mémoire
  ⇒ Implicites et rapides !

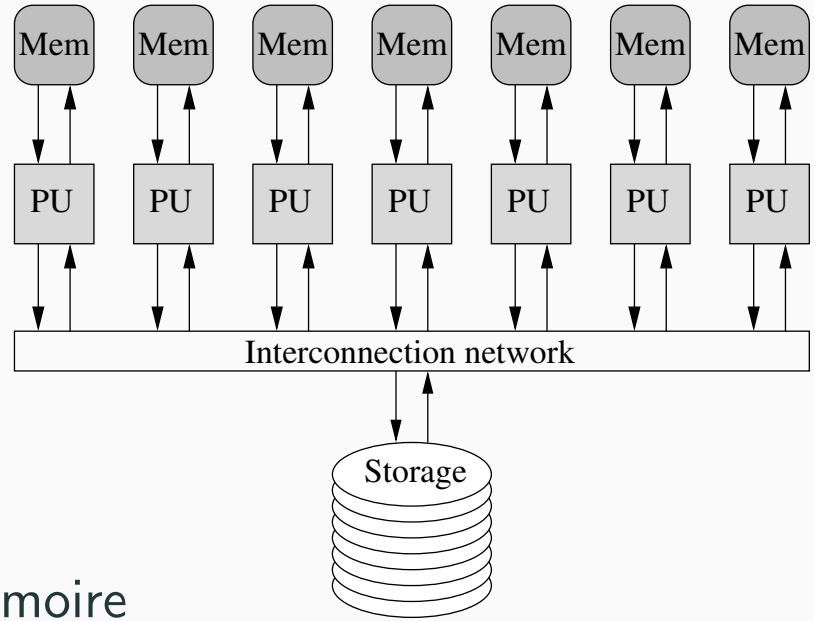
Inconvénients :

- Implique un réseau et une mémoire à large bande passante
- Gestion des accès concurrents *(exclusion mutuelle)*

Systèmes à mémoire distribuée

Architecture :

- Unités de calcul avec chacune une *mémoire locale*



Avantages :

- Pas d'accès concurrents à la mémoire
- Bande passante du réseau moins critique

Inconvénients :

- Implique une distribution des données
- Utilisation de *messages explicites* entre les unités

Grappes locales

Deux types :

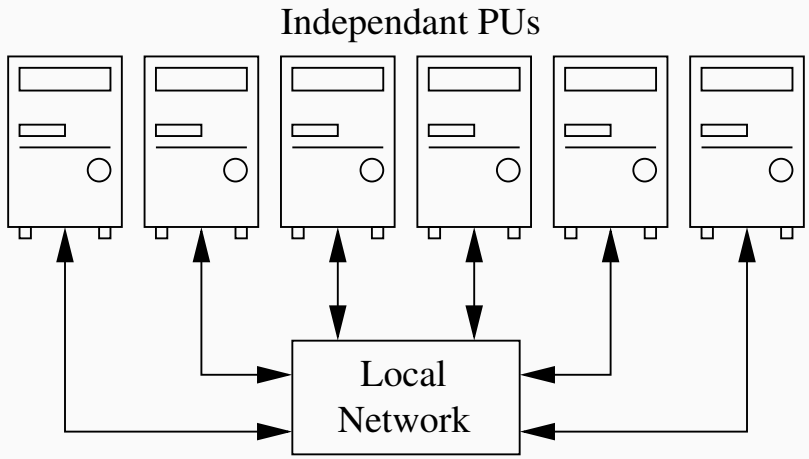
- *COWS/NOWS* :
  - Utilisation de matériel issu de la production de masse (coût réduit)
- *Systèmes intégrés* :
  - Conçus par les grands constructeurs de machines
  - Matériel spécifique : racks, réseau optimisé (coût plus élevé)
  - Environnement logiciel (système, développement,...)

Avantages :

- Flexibilité de configuration
- Maintenance plus facile

Inconvénients :

- Réseau relativement lent (tend à se réduire)



Grilles

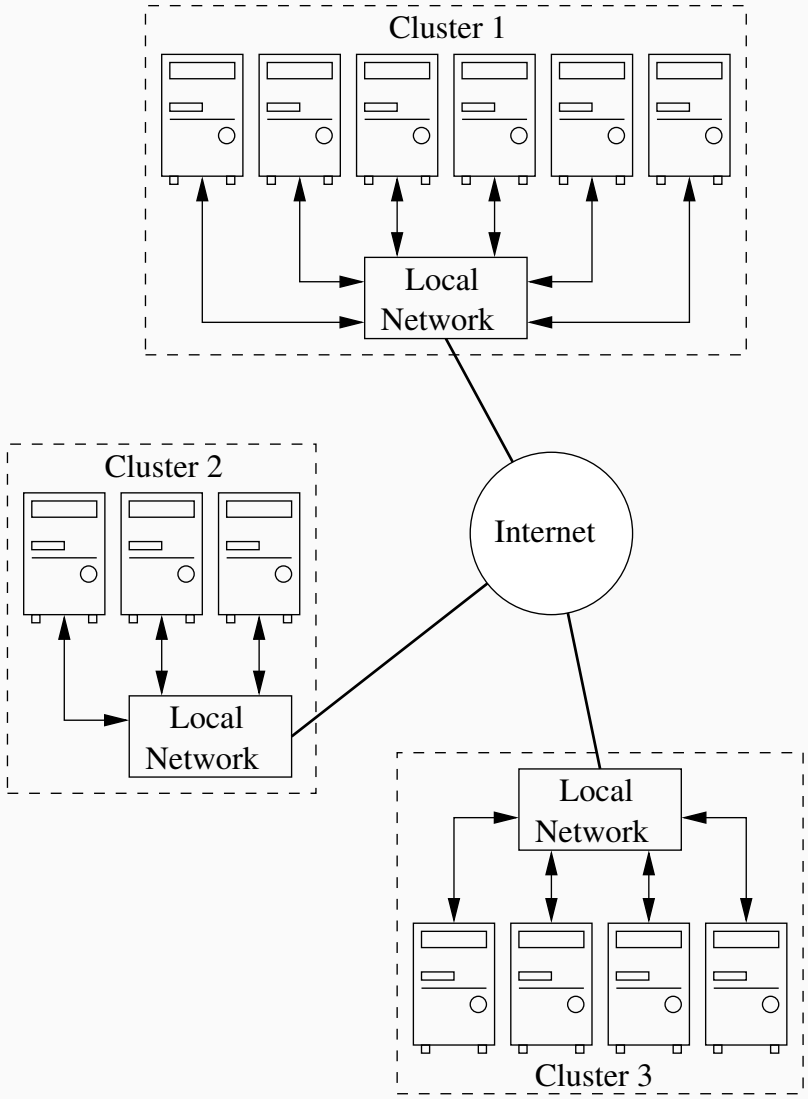
Interconnexion de systèmes géographiquement distants via le réseau global

Avantages :

- Puissance de calcul et capacité de stockage bien plus importants

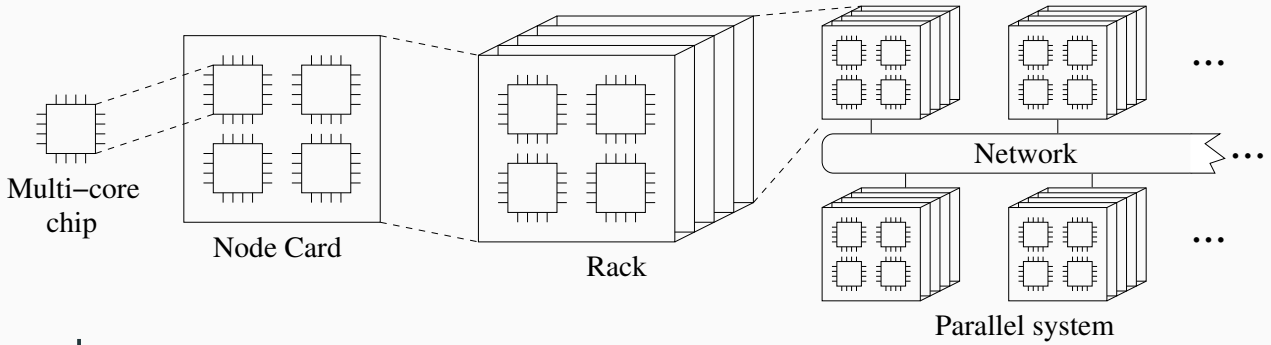
Inconvénients :

- Impact plus important du réseau
- Gestion plus complexe
- Fiabilité
- Sécurité
- ...



Bilan

Systèmes parallèles de plus en plus *hiérarchiques* avec plusieurs niveaux de parallélisme



Typiquement, on a :

- Un ensemble de machines en réseau intégrant :
  - plusieurs processeurs :
    - multi-coeurs
    - avec pipelines internes
  - et éventuellement des accélérateurs de calcul :
    - GPU, FPGA,...

⇒ Une exploitation efficace doit donc tenir compte de *tous* ces étages et de leurs *spécificités* :

- Mémoire partagée ou distribuée, architecture SIMD ou MIMD,...

Super-calculateurs

Liste mise à jour sur `https://www.top500.org`

R_{max} and R_{peak} values are in PFlop/s. For more details about other fields, check the TOP500 description.

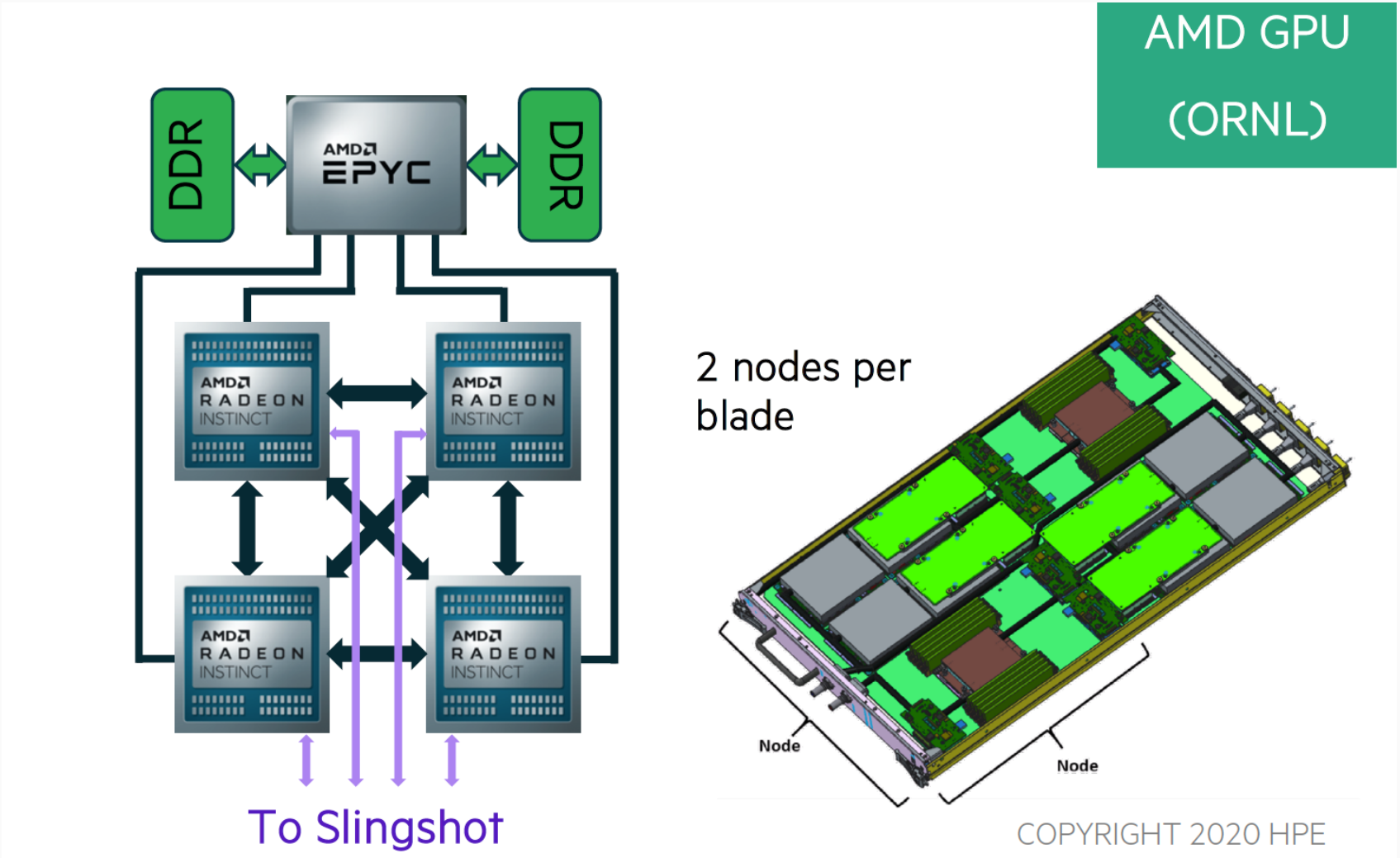
R_{peak} values are calculated using the advertised clock rate of the CPU. For the efficiency of the systems you should take into account the Turbo CPU clock rate where it applies.

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,730,112	1,102.00	1,685.65	21,100
2	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
3	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	2,220,288	309.10	428.70	6,016
4	**Leonardo** - BullSequana XH2000, Xeon Platinum 8358 32C 2.6GHz, NVIDIA A100 SXM4 64 GB, Quad-rail NVIDIA HDR100 Infiniband, Atos EuroHPC/CINECA Italy	1,463,616	174.70	255.75	5,610

Les 4 plus puissants super-calculateurs en 2022

`https://www.top500.org/lists/top500/list/2022/11/`

Détail d'un nœud de Frontier



Composition d'un noeud du super-calculateur **Frontier**

Concepts algorithmiques du parallélisme

Concepts algorithmiques

Au niveau algorithmique, on considère un système :

- avec un *nombre quelconque* d'unités de calcul (numérotées)
- sans hypothèse particulière sur l'architecture (selon contexte)

Élément algorithmique générique :

- *Boucle spatiale* :

```
1 pour cpt de déb à fin faire en parallèle
2    ... // Instructions exécutées sur les unités entre déb et fin
3 fpour
```

- Les itérations de la boucle sont *distribuées* aux unités dont les numéros sont entre les indices `déb` et `fin`
- Le compteur `cpt` a une valeur différente pour chaque unité

⚠ En *mémoire distribuée*, les *variables* mises en jeu sont *locales* à l'unité qui effectue l'action !

Exemples

Exemple 1 :

```
1 pour i de 0 à 99 faire en parallèle
2   tab[i] ← 0
3 fpour
```

- Mémoire partagée : initialisation parallèle du tableau `tab` à 0
- Mémoire distribuée : chaque unité modifie uniquement la case de `tab` d'indice son numéro, dans sa mémoire locale

Exemple 2 :

```
1 pour i de 0 à 99 faire en parallèle
2   val ← 0
3 fpour
```

- Mémoire partagée : la valeur val est mise à 0 (la boucle est inutile)
- Mémoire distribuée : chaque unité affecte 0 à sa copie locale de `val`

Exécution parallèle différenciée

On peut exprimer des *tâches différentes* en parallèle en utilisant une boucle sur les unités de calcul et des tests sur les indices des unités

```
1  // Calcul parallèle d'un polynôme en mémoire partagée
2  pour p de 1 à 3 faire en parallèle
3    selon que p est :
4      1 : res1 ← 3*z + 2          // Calcul sur unité 1
5      2 : res2 ← x^2 + 2*x - 5    // Calcul sur unité 2
6      3 : res3 ← 3*y^2 - 2*y + 7 // Calcul sur unité 3
7    fselon
8  fpour
9  // Retour au séquentiel
10 res ← res1 * res2 * res3
```

On peut aussi utiliser des `si alors sinon` à la place du `selon que`

Exclusion mutuelle

Parfois, les unités doivent exécuter des instructions de manière *exclusive*

Exemple de somme des éléments d'un tableau en mémoire partagée :

```
1 somme ← 0
2 pour i de 0 à n-1 faire en parallèle
3   somme ← somme + tab[i] // Erreur → écritures simultanées dans somme
4 fpour
```

Éléments algorithmiques nécessaires pour gérer l'*exclusion mutuelle* :

- *Opération atomique* : ne peut être interrompue par un autre processus
  - Mécanismes au niveau matériel : `Test_And_Set`, `Compare_And_Swap`,...
- *Section critique* : bloc d'instructions *protégé*
  - Ne peut être exécuté que par *une seule unité à la fois*

⇒ On construit des sections critiques avec les opérations atomiques

L'*algorithme de la boulangerie* (Lamport) permet de réaliser des mutex *sans* opérations atomiques lorsque l'on a un *nombre fixé d'unités*

Verrous pour l'exclusion mutuelle

On utilise des *verrous* : éléments partagés à *modifications exclusives*

Initialisation d'un verrou :

- `initVerrou(v)` :
  - *Initialise* le verrou `v` dans l'état libre (non vérouillé)
  - Doit être exécutée par *une seule unité*

On définit deux *opérations atomiques* de modification de verrou :

- `vérouille(v)` :
  - *Attend* tant que le verrou `v` est vérouillé (non libre)
  - *Vérouille* l'accès à `v` dès qu'il est libre
- `dévérouille(v)` :
  - *Libère* l'accès à `v`
  - Doit être exécutée par l'unité qui a vérouillé `v`

On peut ajouter une opération classique de *test* :

- `estVérouillé(v)` :
  - Renvoie `Vrai` si le verrou n'est pas libre
  - Retourne le résultat immédiatement (*non bloquant*)

Retour sur la somme des éléments d'un tableau

Somme en mémoire partagée avec exclusion mutuelle :

```
1 initVerrou(v)    // Initialisation du verrou
2 somme ← 0
3 pour i de 0 à n-1 faire en parallèle
4   vérouille(v)   // Récupération du verrou : début de section critique
5   somme ← somme + tab[i] // Écriture exclusive dans somme
6   dévérouille(v) // Libération du verrou : fin de section critique
7 fpour
```

Déroulement :

1. Toutes les unités vont tenter de vérouiller `v`
2. *Une seule* va l'obtenir → les autres attendent
3. Après son calcul, l'unité qui a le verrou le libère
4. Une autre unité le prend, et ainsi de suite...
5. ...jusqu'à ce que toutes les unités aient exécuté leur calcul

⚠ *Blocages* possibles si verrous mal utilisés ! ! ⚠

⚠ Effet de *séquentialisation des calculs* si trop utilisé ! ! ⚠

Évaluation du parallélisme

Évaluation du parallélisme

Pour un algorithme parallèle donné, on considère :

- $T_1(n)$ = temps pour résoudre un problème de taille n sur 1 unité
- $T_p(n)$ = temps pour résoudre le même problème sur p unités
  → c'est le temps *maximal* sur l'ensemble des unités !

Accélération :

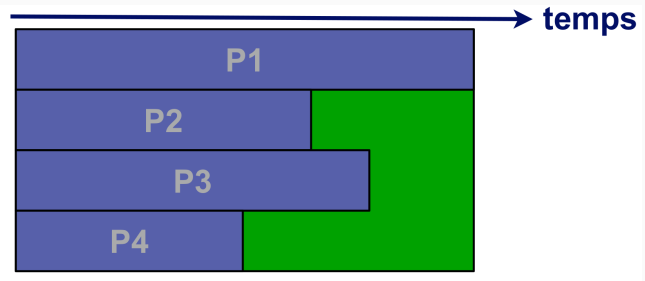
$$S_p(n) = \frac{T_1(n)}{T_p(n)}$$

- Au maximum elle est égale à p (problème totalement découplé)

Travail :

$$W_p(n) = p.T_p(n)$$

- Représente le *potentiel de calcul* des ressources monopolisées par l'algorithme
- Pas obligatoirement équivalent à l'utilisation réelle de ces ressources

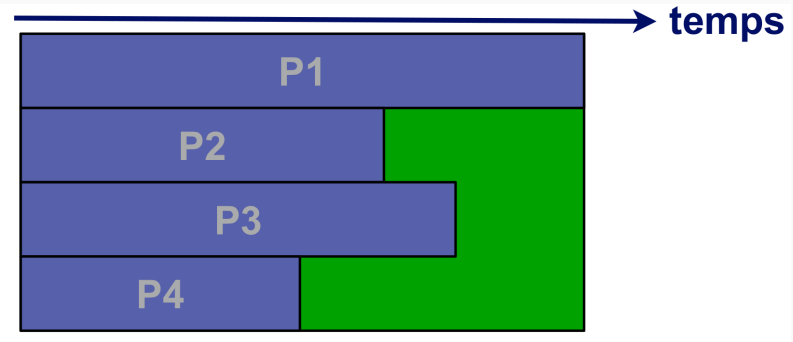


Évaluation du parallélisme

Efficacité :

$$E_p(n) = \frac{T_1(n)}{W_p(n)} = \frac{S_p(n)}{p}$$

- Représente la part des ressources *réellement utilisées* par l'algorithme
- Ratio de la partie utilisée (bleue) sur la surface totale du rectangle



- Valeur entre 0 et 1 : 1 = 100% d'efficacité
- L'objectif est donc de *minimiser la partie inutilisée* (verte)

Remarques

⚠ *Une bonne accélération n'est pas synonyme d'efficacité !*

Exemple : comparaison de deux exécutions parallèles

- $T_1(n) = 100$
- $T_4(n) = 50 \Rightarrow S_4(n) = 2 \Rightarrow W_4(n) = 200 \Rightarrow E_4(n) = 0,5$
- $T_8(n) = 33 \Rightarrow S_8(n) = 3 \Rightarrow W_8(n) = 264 \Rightarrow E_8(n) = 0,38$

S et E dépendent de n mais aussi de p !

On peut parfois avoir des résultats *super-linéaires* :

- Cela provient généralement de problèmes de *cache mémoire*
- Problème trop gros pour être stocké complètement sur peu de machines
- L'utilisation des caches externes ralentie l'exécution de l'algorithme
- Provoque une accélération supplémentaire dès que les caches ne sont plus utilisés lorsque le nombre de machines est suffisant
- Plus rarement, on peut avoir une *économie d'instructions* dans la version parallèle (notamment en calcul vectoriel)

Exemple de gain d'instructions

Algorithme séquentiel :

```
1 pour i de 0 à 99 faire // test de l'indice = 100 ops
2   tab[i] ← 0           // affectation = 100 ops
3 fpour                  // incrément de l'indice = 100 ops
```

Algorithme vectoriel (mémoire partagée) sur 100 unités :

```
1 pour i de 0 à 99 faire en parallèle // calcul de l'indice = 1 op
2   tab[i] ← 0                        // affectation case tableau = 1 op
3 fpour                               // aucune opération = 0 op
```

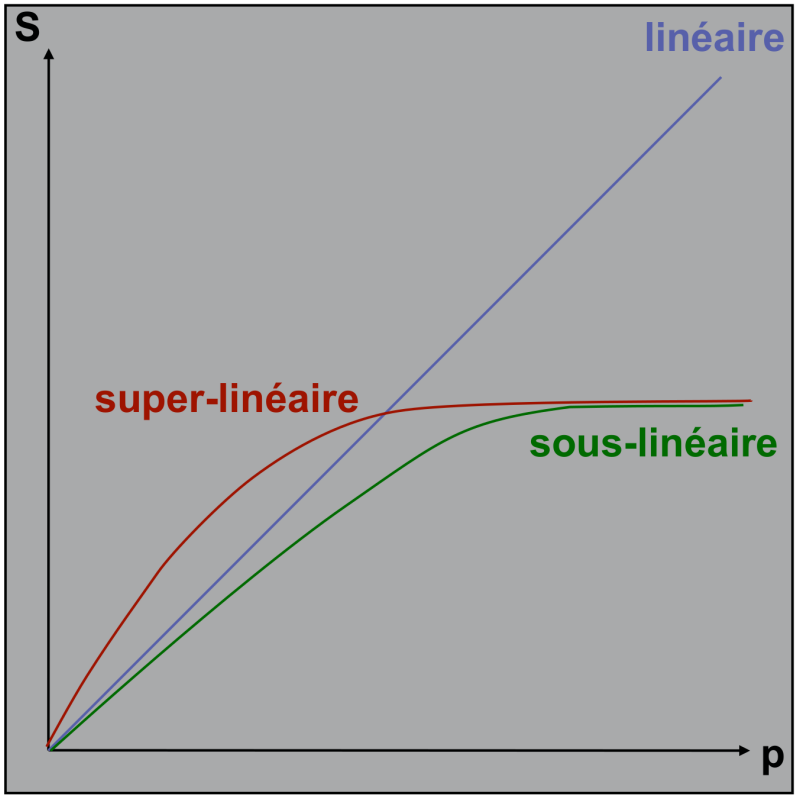
On a donc :

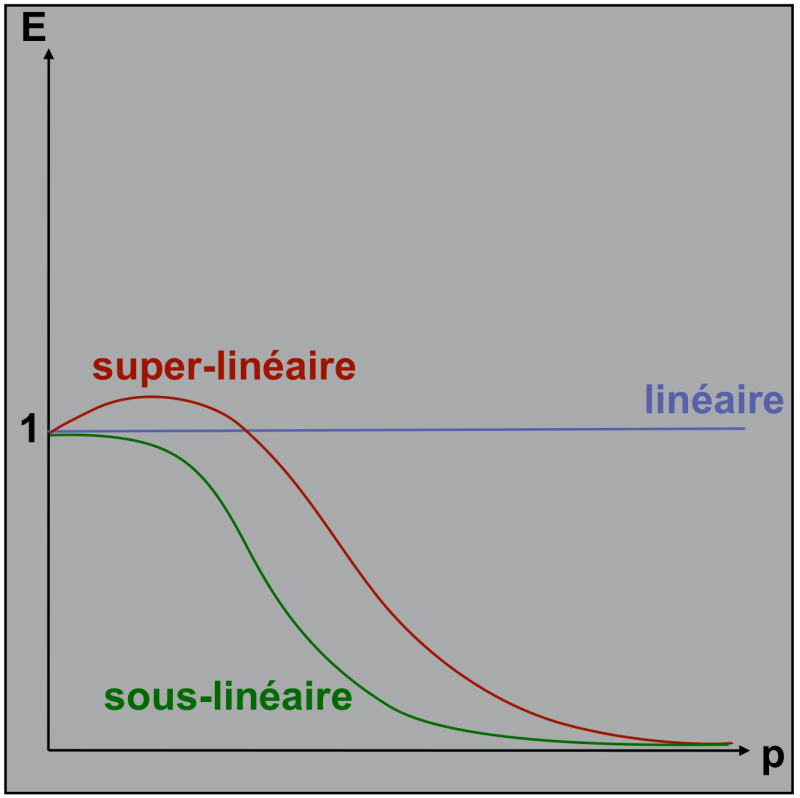
- $T_1(A) = 300$ et $T_{100}(A) = 2$

$\Rightarrow S_{100}(A) = \frac{300}{2} = 150$

$\Rightarrow E_{100}(A) = \frac{150}{100} = 1,5 > 1$, donc *super-linéaire* !

Courbes typiques pour des algorithmes linéaires





Exemple plus concret de super-linéarité

Contexte :

- Problème de taille n
- Algorithme A avec temps séquentiel en $\mathcal{O}\left(n^2\right)$
- P unités de calcul (bien inférieur à n)
- Décomposition en P parties de taille $\frac{n}{P}$

  $\Rightarrow$ Chaque partie est traitée en temps $\mathcal{O}\left(\frac{n^2}{P^2}\right)$
- Recomposition des résultats partiels en temps $\mathcal{O}\left(P.n\right)$

On a donc :

- $T_1(A) = n^2$ et $T_P(A) = \frac{n^2}{P^2} + P.n$

$\Rightarrow$ $\boxed{S_P(A)} = \frac{n}{\frac{n}{P^2}+P} \rightarrow \boxed{P^2}$ quand $n >> P$ ($\rightarrow 1$ quand $P \approx n$)

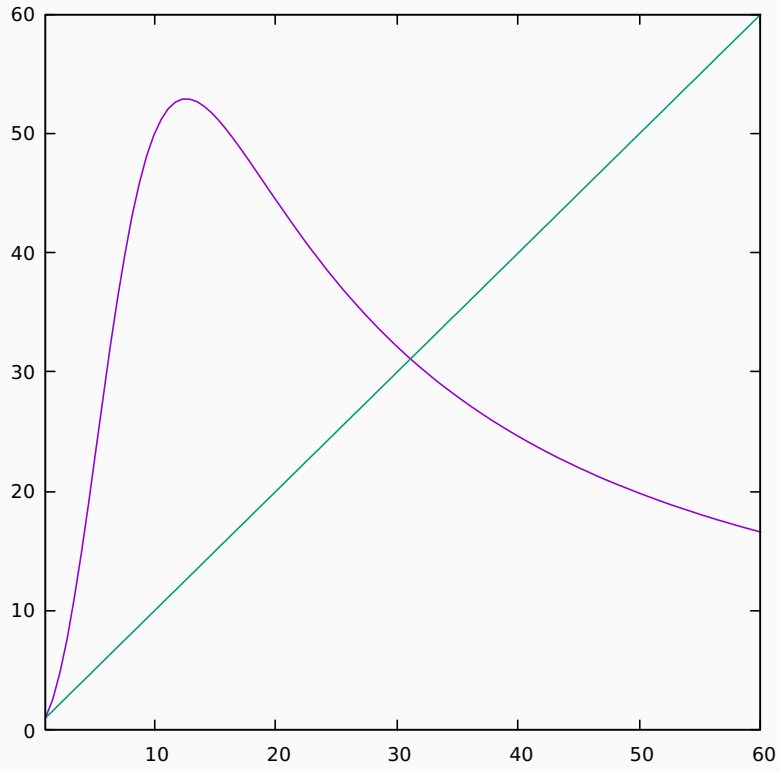
$\Rightarrow$ $\boxed{E_P(A)} \rightarrow \frac{P^2}{P} = \boxed{P}$, donc *super-linéaire* !

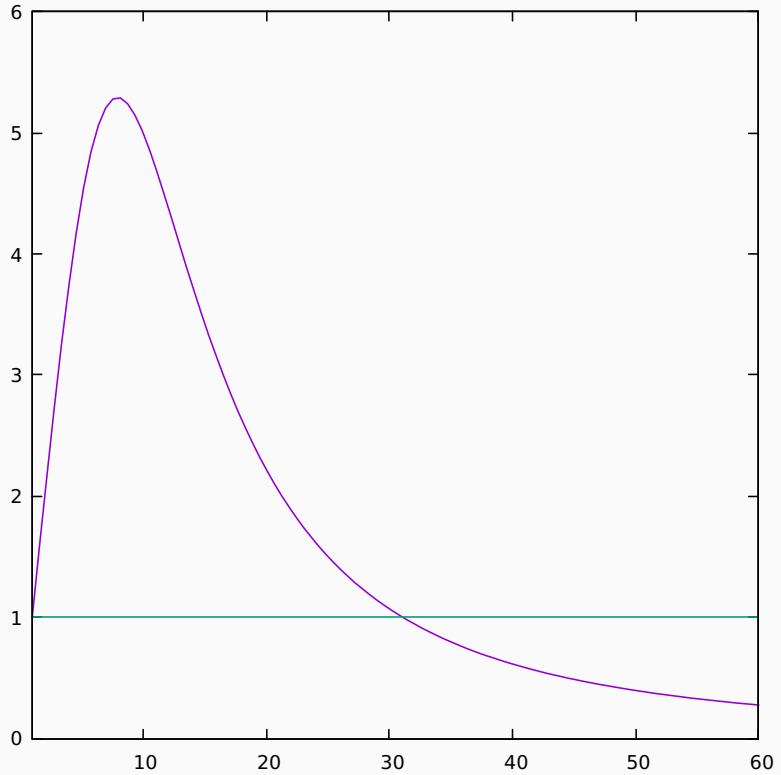
On peut alors introduire la notion d'efficacité *quadratique* $\frac{S_P(A)}{P^2}$

Relation entre n et P pour avoir accélération $\geq P$?

On déduit que $n \geq \left\lceil \frac{P^3}{P-1} \right\rceil \Rightarrow$ P = $\boxed{\left\lfloor \sqrt{n} \right\rfloor \text{ ou } \left\lfloor \sqrt{n} \right\rfloor - 1}$

Courbes pour l'exemple de l'algorithme quadratique





Résultats pour n=1000 :

- Accélération de 52.86 et efficacité de 4.06 pour 13 unités
- Accélération de 42.32 et efficacité de 5.29 pour 8 unités

Discussion sur la super-linéarité

En fait, cela dépend de ce que l'on évalue :

- *Séquentiel vs Parallèle* : meilleur algo dans chaque contexte
  - Identifier explicitement les différences algorithmiques/comportementales

  $\Rightarrow$ Super-linéarité possible
- *Algo parallèle face à lui-même* : qualité intrinsèque
  - Comparer le temps d'*exécution parallèle* sur p unités au temps de la *simulation séquentielle* de ce programme sur une seule unité
    - Si on ne tient pas compte des problèmes de mémoire, la simulation prendra au plus p fois le temps de l'exécution parallèle

    $\Rightarrow$ Accélération maximale égale à p
    - Exemple précédent du calcul vectoriel :
      Simulation séquentielle du programme vectoriel = 200 ops
      L'accélération est donc $\frac{200}{2} = 100$ et l'on a bien une efficacité de 1
  - Comparaison des temps du *même* algo entre 1 et p unités :

    $\Rightarrow$ Loi de *Amdahl*

Loi de Amdahl

Souvent, on peut identifier deux parties distinctes dans un algorithme :

- Une partie *purement séquentielle* (non parallélisable) :
  - On lui associe un ratio (pourcentage) de l'algorithme complet : $0 \leq R_s(n) \leq 1$
- Une *partie parallélisable* :
  - Son ratio est égal à $1-R_s(n)$ (complémentaire de la partie séquentielle)

On a donc pour un problème de taille n :

- $T_s(n) = R_s(n).T_1(n)$ = temps de la partie purement séquentielle
- $T_{//}(n) = (1 - R_s(n)).T_1(n)$ = temps de la partie parallélisable
- $T_1(n) = T_s(n) + T_{//}(n) = R_s(n).T_1(n) + (1 - R_s(n)).T_1(n)$

Et le *temps minimal théorique d'exécution* sur p unités est :

$$T_p(n) = T_s(n) + \frac{T_{//}(n)}{p} = \left(R_s(n) + \frac{1 - R_s(n)}{p} \right) \times T_1(n)$$

Loi de Amdahl (suite)

Ce qui donne :

$$S_p(n) = \frac{T_1(n)}{\left(R_s(n)+\frac{1-R_s(n)}{p}\right)\times T_1(n)} = \frac{1}{R_s(n)+\frac{1-R_s(n)}{p}}$$

$$E_p(n) = \frac{1}{p\times\left(R_s(n)+\frac{1-R_s(n)}{p}\right)} = \frac{1}{1+(p-1)\times R_s(n)}$$

Et nous obtenons les limites suivantes :

$$\lim_{p\to\infty} S_p(n) = \frac{1}{R_s(n)} \quad \Rightarrow \quad \text{pour } R_s(n) = 10\%, \text{ on ne dépassera pas une accélération de 10}$$

$$\lim_{p\to\infty} E_p(n) = 0|1 \quad \Rightarrow \quad \text{l'efficacité devient nulle s'il y a une partie non parallélisable}$$

Principe de Brent

Extension de la simulation séquentielle (sur 1 unité) :

- Simulation sur p unités d'un algorithme parallèle utilisant un nombre indéterminé d'unités :
  - Algorithme A avec coût séquentiel C_1 et temps parallèle T_∞
  - Il est possible de simuler A sur p unités identiques en :

$$T_p = \mathcal{O}\left(\frac{C_1}{p} + T_\infty\right)$$

  - Comment ?
    - À chaque instant i, A exécute $C_1(i)$ opérations : $C_1 = \sum_{i=1}^{T_\infty} C_1(i)$
    - Chaque instant est simulé sur p unités en $\left\lceil \frac{C_1(i)}{p} \right\rceil \leq \frac{C_1(i)}{p} + 1$
    - Si l'on somme les instants, on retrouve le résultat

$\Rightarrow$ *Prédiction des performances* lorsque l'on réduit l'ordre de grandeur du nombre d'unités

Grain et degré de parallélisme

Grain :

- Taille moyenne des tâches élémentaires du processus parallèle
- Choix lié à l'architecture cible :
  - Échelle : bit, opérateur, expression, instruction, fonction, programme
  - *Grain fin* : généralement SIMD en mémoire partagée
  - *Gros grain* : généralement MIMD en mémoire distribuée
- Plusieurs grains possibles dans un même algo (hiérarchie)

Degré :

- Mesure le nombre d'opérations possibles simultanément (≥ 1)
- Peut varier pendant le déroulement du programme
  (notion de parties distinctes avec degrés différents)
- Les parties de degré 1 représentent la part purement séquentielle

$\Rightarrow$ Notions de degré min, max et moyen

- Donne une information sur le nombre d'unités à utiliser et permet d'évaluer la performance selon le nombre d'unités

Interprétation du degré

Considérons un programme parallèle décomposable en n parties de degrés respectifs d_i et de temps d'exécution t_i tel que : $T_1 = \sum_{i=1}^{n} d_i t_i$

On a alors :

$$S_p = \frac{\sum_{i=1}^{n} d_i t_i}{\sum_{i=1}^{n} \left\lceil \frac{d_i}{p} \right\rceil t_i}$$

Si $p \geq \max_i d_i$ alors on a l'accélération maximale $S_p = \frac{\sum_{i=1}^{n} d_i t_i}{\sum_{i=1}^{n} t_i}$

Mais cela ne correspond pas obligatoirement à l'*efficacité maximale*

Exemple : programme en 3 parties

d_i	2	7	3
t_i	5	20	7

Calculer le nombre d'unités donnant la meilleur efficacité

Interprétation du degré

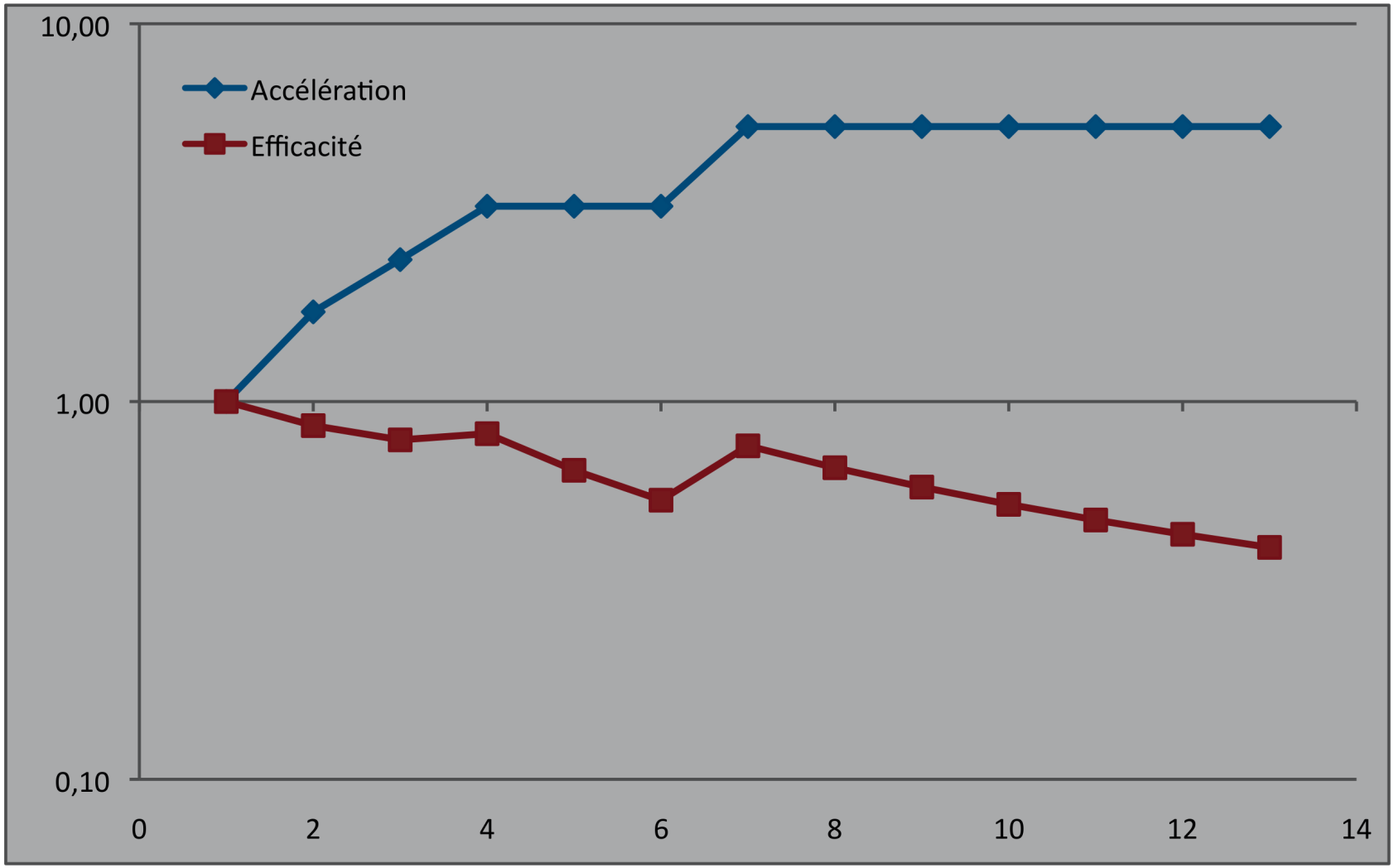
Exemple : programme en 3 parties

d_i	2	7	3
t_i	5	20	7

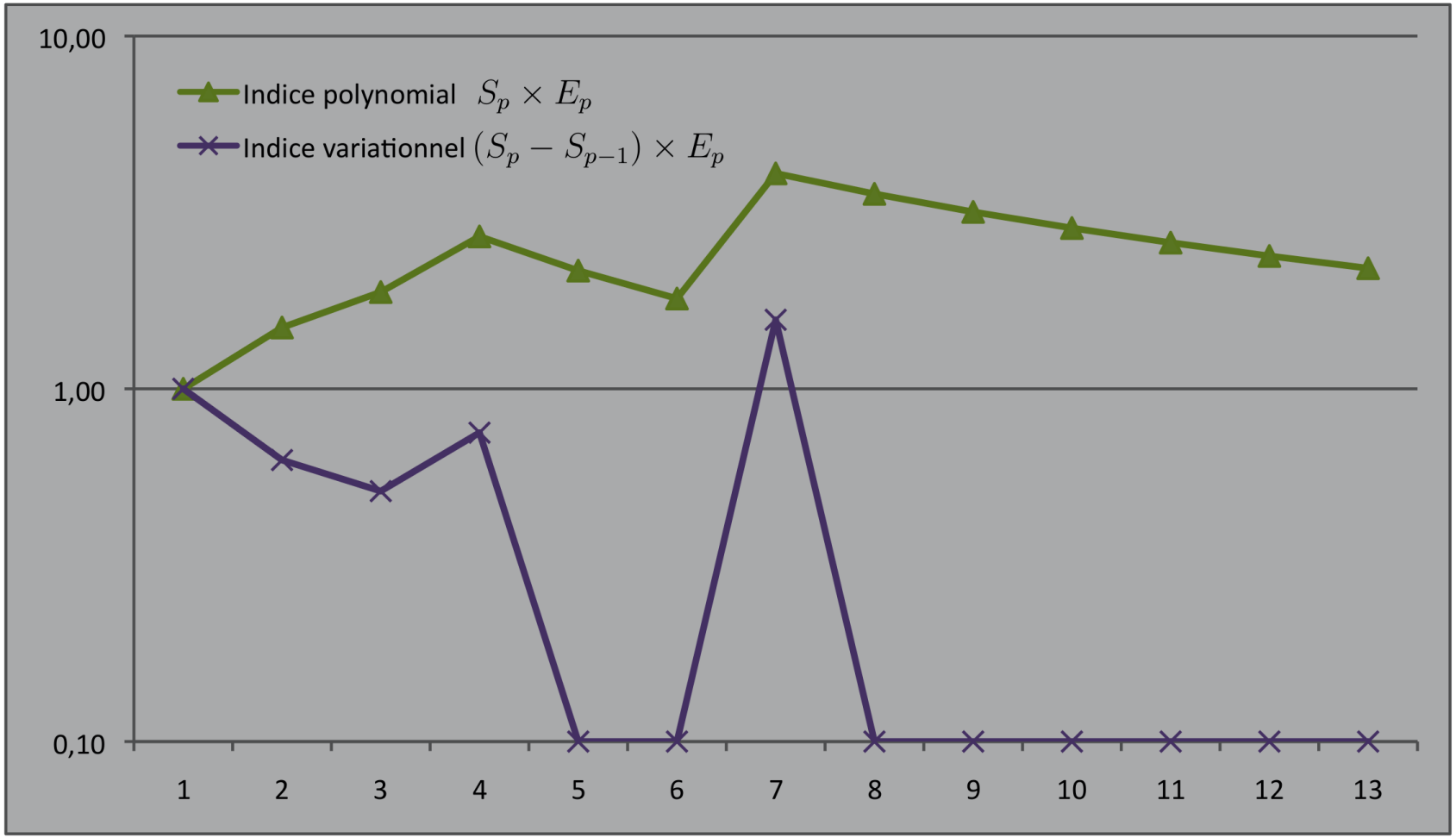
Évolution des performances en fonction du nombre d'unités :

nb unités	1	2	3	4	5	6	7	8	9
temps	171	99	72	52	52	52	32	32	32
accélération	1	1,73	2,38	3,29	3,29	3,29	**5,34**	5,34	5,34
efficacité	1	**0,86**	0,79	0,82	0,66	0,55	0,76	0,67	0,59

Courbes de performances



Indices globaux



Modèles de programmation parallèle

Modèles de programmation parallèle

Parallélisme de données : *data parallelism*

- Distribution des données dans le système
- Exploite généralement la régularité des données
- Application d'un même calcul à des données différentes

Parallélisme de contrôle ou de tâches : *control/task parallelism*

- Décomposition/Distribution des calculs dans le système
- Chaque unité calcule une partie du résultat attendu
- Les données peuvent être dupliquées

Parallélisme de flux : *pipeline*

- Découpage d'un traitement en tâches successives : travail à la chaîne
- Les tâches doivent avoir des durées les plus proches possibles

Mixage des schémas précédents :

- Répartition des calculs et des données dans le système
- Gestion généralement plus complexe (traitements supplémentaires)

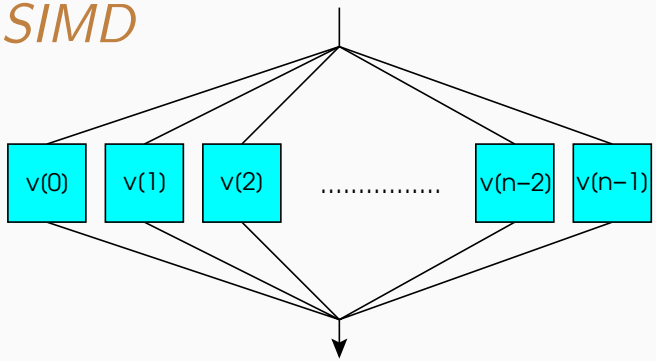
Exemple comparatif

Calcul de n valeurs d'un polynôme donné :

```
1 pour i de 0 à n-1 faire
2   v[i] ← a + b.x[i] + c.x[i]^2 + d.x[i]^3 + e.x[i]^4 + f.x[i]^5
3 fpour
```

Parallélisme de données : plutôt sur machines *SIMD*

```
1 pour i de 0 à n-1 faire en parallèle
2   v[i] ← a + b.x[i]   + c.x[i]^2 + d.x[i]^3
3            + e.x[i]^4 + f.x[i]^5
4 fpour
```



- Nécessite n unités au plus : au-delà, ça ne sert à rien !
- Si l'on a moins de n unités, chacun calcule plusieurs `v[i]`

$\Rightarrow$ Attribuer des paquets de données à chaque unité :

Exemple pour $n = 20$ et $p = 3$, les calculs pourront être répartis en :

- P0 : v[0] à v[6] (7 éléments)
- P1 : v[7] à v[13] (7 éléments)
- P2 : v[14] à v[19] (6 éléments)

si $n\%p \neq 0$, chaque unité traite

$\left\lceil \frac{n}{p} \right\rceil$ ou $\left\lfloor \frac{n}{p} \right\rfloor$ éléments

Exemple comparatif

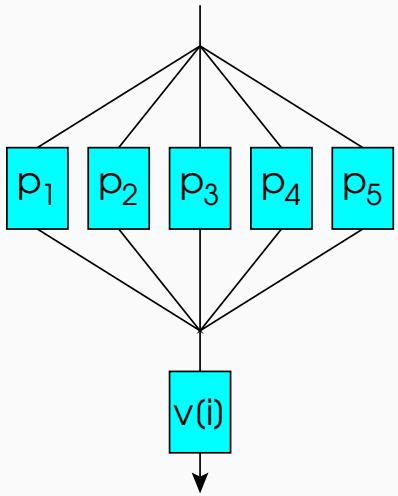
Parallélisme de tâches : plutôt sur machines *MIMD*

- Réécriture du polynôme, par exemple :

```
v[i] = (a + b.x[i]) + c.x[i]^2 + x[i]^3 . ((d + e.x[i]) + (f.x[i]^2))
```

- Menant à l'algorithme suivant :

```
1  pour i de 0 à n-1 faire
2    pour p de 1 à 5 faire en parallèle
3      selon que p est :
4      1 : p1 ← a + b.x[i]
5      2 : p2 ← c.x[i]^2
6      3 : p3 ← x[i]^3
7      4 : p4 ← d + e.x[i]
8      5 : p5 ← f.x[i]^2
9      fselon
10   fpour
11   v[i] ← p1 + p2 + p3.(p4 + p5)
12 fpour
```

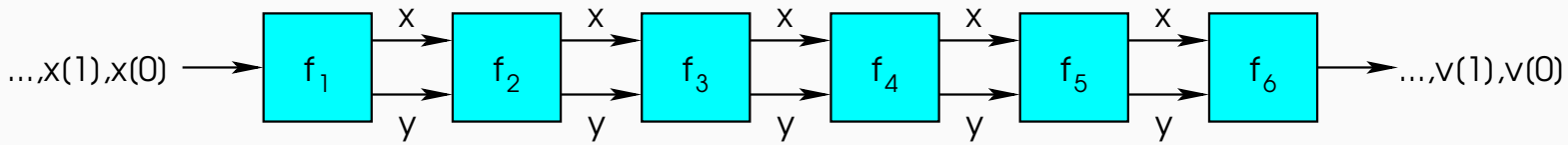


- Parallélisme limité à 5 unités dans ce cas là
- Souvent nécessaire de fusionner les résultats partiels ⇒*synchro !*
  - Important d'avoir des tâches de *durées identiques* (même complexité)

Exemple comparatif

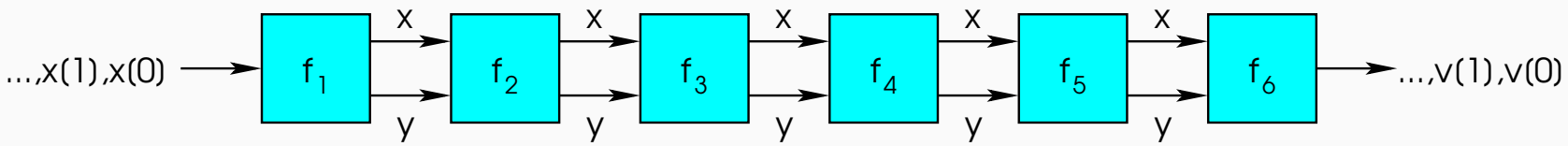
Parallélisme de flux : pipeline

- Réécriture sous forme d'une *composition de fonctions* :
  $f_1(x) = (x, f + x.0), f_2(x, y) = (x, e + x.y), f_3(x, y) = (x, d + x.y)),$
  $f_4(x, y) = (x, c + x.y), f_5(x, y) = (x, b + x.y), f_6(x, y) = a + x.y$
- Menant au calcul suivant pour chaque `v[i]` :
  $v[i] \leftarrow f_6(\, f_5(\, f_4(\, f_3(\, f_2(\, f_1(x[i])\,)\,)\,)\,)\,)$



- Chaque fonction correspond à une *étage* du pipeline (une étape)
- Il y a plusieurs phases de fonctionnement :
  - *Remplissage* : remplissage de tous les étages du pipeline
  - *Régime plein* : tous les étages sont actifs
  - *Vidage* : fin du flux de données
- Degré de parallélisme donné par le nombre d'étages
- Efficacité maximale lorsque les f_i ont toutes la *même durée*

Exemple de fonctionnement avec 8 données



Remplissage	x[0]						
	x[1]	x[0]					
	x[2]	x[1]	x[0]				
	x[3]	x[2]	x[1]	x[0]			
	x[4]	x[3]	x[2]	x[1]	x[0]		
Plein	x[5]	x[4]	x[3]	x[2]	x[1]	x[0]	
	x[6]	x[5]	x[4]	x[3]	x[2]	x[1]	**v[0]**
	x[7]	x[6]	x[5]	x[4]	x[3]	x[2]	**v[1]**
Vidage		x[7]	x[6]	x[5]	x[4]	x[3]	**v[2]**
			x[7]	x[6]	x[5]	x[4]	**v[3]**
				x[7]	x[6]	x[5]	**v[4]**
					x[7]	x[6]	**v[5]**
						x[7]	**v[6]**
							v[7]

Parallélisme de données

α-notation : α(opérateur, v1, v2)

- Opérateur k-aire appliqué à k vecteurs en parallèle
- Idéalement, une unité calcule un élément du vecteur résultat
- Vérifie les *conditions de Bernstein* (indépendance entre les calculs) :
  - $L(C_1) \cap E(C_2) = \varnothing$ avec $L(A)$ = variables lues pendant A
  - $E(C_1) \cap L(C_2) = \varnothing$ et $E(A)$ = variables modifiées pendant A
  - $E(C_1) \cap E(C_2) = \varnothing$

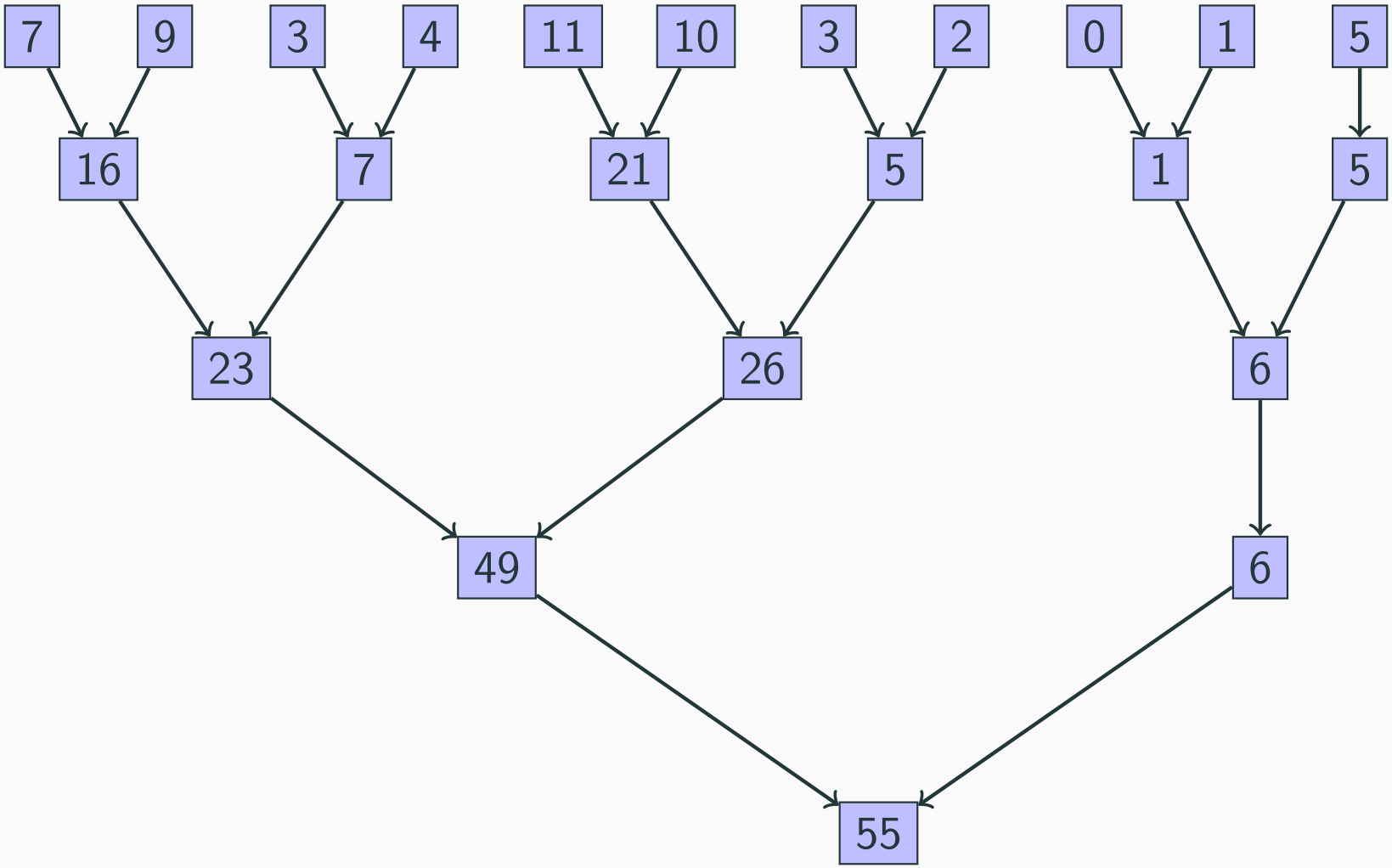
β-réduction logarithmique : β(opérateur, v)

- Issue de la β-réduction :
  - Fonction binaire appliquée successivement aux éléments d'un vecteur
  - Pas directement parallèle mais utile avec les fonctions associatives

$\Rightarrow$ Succession de regroupements par couples distincts : *arbres binaires*

- Réduction par 2 à chaque étape du nombre de données à traiter
  $\Rightarrow \mathcal{O}(log_2(n))$ étapes pour traiter n données

Exemple de somme



Principe de Brent appliqué à la réduction

- Coût séquentiel : $\mathcal{O}(n)$
- Temps parallèle : $\mathcal{O}(log(n))$
- Nombre d'unités : $\mathcal{O}(n)$
- Même temps parallèle avec moins de $\mathcal{O}(n)$ unités ?
- Avec p unités, on a $T_p = \mathcal{O}\left(\frac{n}{p} + log(n)\right)$
- En choisissant $\boxed{p = \frac{n}{log(n)}}$, on obtient $\boxed{T_p = \mathcal{O}(log(n))}$
- On peut donc avoir des temps d'exécution de *même complexité* avec *moins d'unités* ! !

$\Rightarrow$ Améliore l'efficacité du parallélisme (lorsque non maximale)

Exercice

Multiplication de matrices : $A(m, l) \times B(l, n)$

```
1 pour i de 0 à m-1 faire
2   pour j de 0 à n-1 faire
3     C[i][j] ← 0
4     pour k de 0 à l-1 faire
5       C[i][j] ← C[i][j] + A[i][k] * B[k][j]
6     fpour
7   fpour
8 fpour
```

Version parallèle avec les fonctions `lig(M, num)` et `col(M, num)`

```
1 pour i de 0 à m-1 faire en parallèle
2   pour j de 0 à n-1 faire en parallèle
3     C[i][j] ← β(+, α(×, lig(A,i), col(B,j)))
4   fpour
5 fpour
```

Évaluation

Complexités :

- Version séquentielle : $m.n.l\ (\times) + m.n.l\ (+)$
- Version parallèle : $1\ (\times) + \lceil log_2(l) \rceil\ (+)$

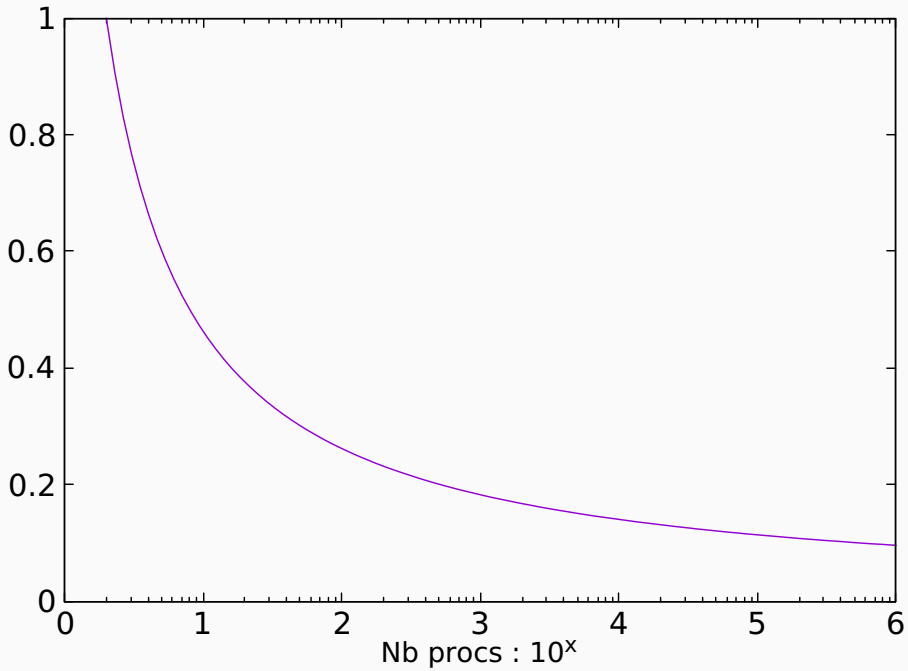
$\Rightarrow$ Accélération supérieure à $m.n$!

Surface :

- Version séquentielle : 1
- Version parallèle : $m.n.l$

Si $\times \approx +$ alors efficacité : $\frac{2}{\lceil log_2(l) \rceil + 1}$

$\rightarrow$ *Efficacité décroît rapidement* quand l augmente !



Avec le principe de Brent

Si on applique le principe de Brent, on peut utiliser *moins d'unités* :

- Surface : $m.n.\frac{l}{\lceil log_2(l) \rceil} = \frac{m.n.l}{\lceil log_2(l) \rceil}$
- Temps : $\frac{l}{\frac{l}{\lceil log_2(l) \rceil}} + \lceil log_2(l) \rceil \approx \lceil log_2(l) \rceil + \lceil log_2(l) \rceil = 2.\lceil log_2(l) \rceil$
- Accélération : $\frac{2.m.n.l}{2.\lceil log_2(l) \rceil} = \frac{m.n.l}{\lceil log_2(l) \rceil}$
- Efficacité : $\frac{m.n.l}{\lceil log_2(l) \rceil} \times \frac{\lceil log_2(l) \rceil}{m.n.l} = 1$

$\Rightarrow$ On retrouve une *efficacité maximale* en utilisant $\frac{l}{\lceil log_2(l) \rceil}$ unités !

Autres variantes

Parallélisme de données *seul* :

- Complexité : l $(\times) + l$ $(+)$
- Surface : $m.n$ unités
- Accélération : $m.n$
- Efficacité : 1

Mixage sans la β-réduction :

- Complexité : 1 $(\times) + l$ $(+)$
- Surface : $m.n.l$ unités
- Accélération : $\frac{2.m.n.l}{1+l}$
- Efficacité : $\frac{2}{1+l}$

$\Rightarrow$ *Pas intéressant !*

Parallélisme de tâches *seul* :

- Complexité : $m.n.(1\ (\times) + log_2(l)\ (+))$ / $m.n.(log_2(l)\ (\times) + log_2(l)\ (+))$
- Surface : l unités / $\frac{l}{log_2(l)}$ unités
- Accélération : $\frac{2.l}{1+log_2(l)}$ / $\frac{l}{log_2(l)}$
- Efficacité : $\frac{2}{1+log_2(l)}$ / 1

Mixage sans l'α-multiplication :

- Complexité : l $(\times) + log_2(l)$ $(+)$
- Surface : $m.n.l$ unités / $\frac{m.n.l}{log_2(l)}$ unités
- Accélération : $\frac{2.m.n.l}{l+log_2(l)}$
- Efficacité : $\frac{2}{l+log_2(l)}$ / $\frac{2.log_2(l)}{l+log_2(l)}$

$\Rightarrow$ *Pas intéressant !*

$\Rightarrow$ *Toujours préférable d'utiliser au maximum les unités disponibles !*

Répartition des données

La *répartition de données* correspond à l'affectation des données aux unités de calcul

En *mémoire partagée* :

- On peut autoriser qu'une même donnée soit lue par n'importe quelle unité
- Il faut s'assurer qu'une donnée n'est *modifiée* que *par une seule unité* !

⇒ La répartition indique quelle unité a le droit d'écrire dans quelle donnée

En *mémoire distribuée* :

- Les données ne peuvent pas toujours être stockées entièrement dans chaque nœud

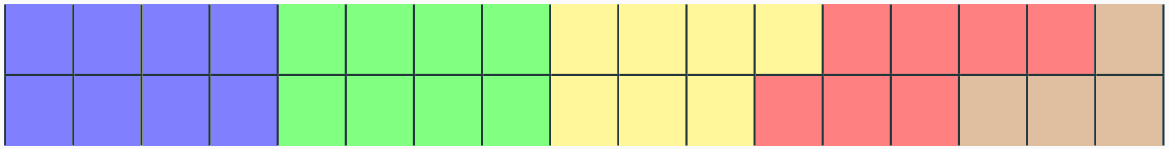
⇒ La répartition indique comment on distribue les données (lues et écrites) sur les nœuds

Répartitions classiques

Pour n données et P unités :

- *Blocs* :
  - Si $n\%P = 0$ alors les données i du bloc b vérifient $b.\frac{n}{P} \leq i \leq (b+1).\frac{n}{P}$

  - Sinon, il faut répartir le *reste* sur les premières ou dernières unités



- *Cyclique* :
  - Les données du bloc b vérifient $i\%P = b$

  - Équilibre implicite !
- *Blocs cycliques* :
  - Les données du bloc b de taille k vérifient $\left\lfloor \frac{i}{k} \right\rfloor \%P = b$

  - Déséquilibre possible selon le choix de k !

Répartitions spécifiques

Dictées par les *dépendances* entre les données

Exemple :

```
1 pour i de 0 à n-1 faire
2   a[i] ← f(i)            // synchro nécessaire
3   b[i] ← a[i] + a[n-1-i] // entre les deux calculs
4 fpour
```

⇒ Pour limiter les attentes, il faut *minimiser les dépendances* entre unités (unité qui utilise des données modifiées par d'autres unités)

La répartition s'exprime par une *sélection des données* en fonction du numéro d'unité via des tests ou des boucles

```
1 pour p de 0 à P-1 faire en parallèle
2   pour i de p.n/(2.p) à (p+1).n/(2.p) faire
3     a[i] ← f(i)
4     a[n-1-i] ← f(n-1-i)
5     b[i] ← a[i] + a[n-1-i]
6     b[n-1-i] ← b[i]
7   fpour
8 fpour
```

Bilan du parallélisme de données

Lié aux *structures de données régulières* :

- Degré de parallélisme potentiellement très élevé
- Identification relativement aisée du parallélisme

Mais :

- Les *dépendances* entre données détériorent le degré de parallélisme
- Le *degré* peut varier fortement entre les parties d'un algorithme :
  - Bonnes accélérations mais efficacité moyenne
- Coût important en *ressources matérielles* (unités)
- Souvent limité au traitement de *données statiques* :
  - Moins adapté aux données dynamiques

Parallélisme de tâches

Le principe est de découper les calculs en *tâches* à faire en parallèle

Critères importants :

- *Grain* :
  - Dépend de l'architecture cible
  - Prise en compte des transferts de données (mémoire distribuée)
  - Et des durées des tâches obtenues
- *Dépendances* entre tâches : *DAG*
  - Analyser les dépendances pour déduire les tâches simultanées
  - Choix du placement des tâches sur les unités (*ordonnancement*)
  - Démarrer les tâches le plus tôt possible
  - Prendre en compte les communications éventuelles

Exemple

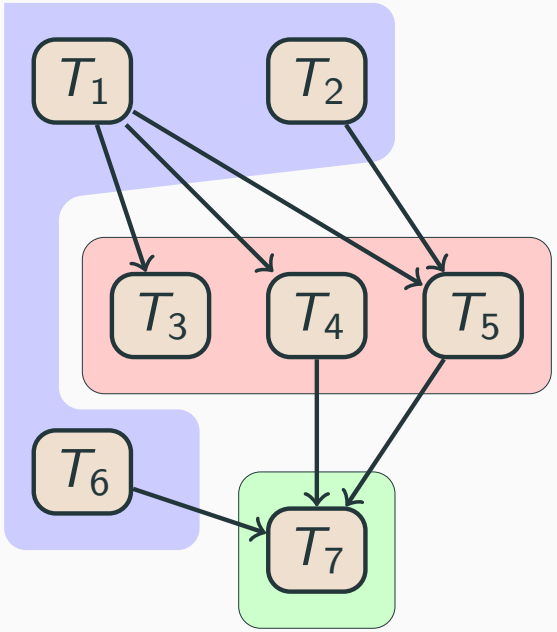
Une façon simple (non optimale) est d'identifier des groupes de tâches exécutables en parallèle :

1. Tâches sans dépendances
2. Tâches avec dépendances satisfaites

→ Contraintes d'ordre entre les groupes :

- (T_1, T_2, T_6) avant (T_3, T_4, T_5) avant (T_7)

⇒ Bien si les tâches d'un même groupe ont des durées proches
On peut être plus précis si on a une estimation de la durée des tâches



Exemple selon le nombre d'unités :

- 1 unité : $T_1, T_2, T_3, T_4, T_5, T_6, T_7$
- 2 unités : $(T_1, T_2), (T_3, T_4), (T_5, T_6), (T_7)$
- 3 unités : $(T_1, T_2, T_6), (T_3, T_4, T_5), (T_7)$
- 4 unités : $(T_1, T_2), (T_3, T_4, T_5, T_6), (T_7)$
- 5 unités et plus : pas mieux car pas plus de 4 tâches indépendantes

Ordonnancement

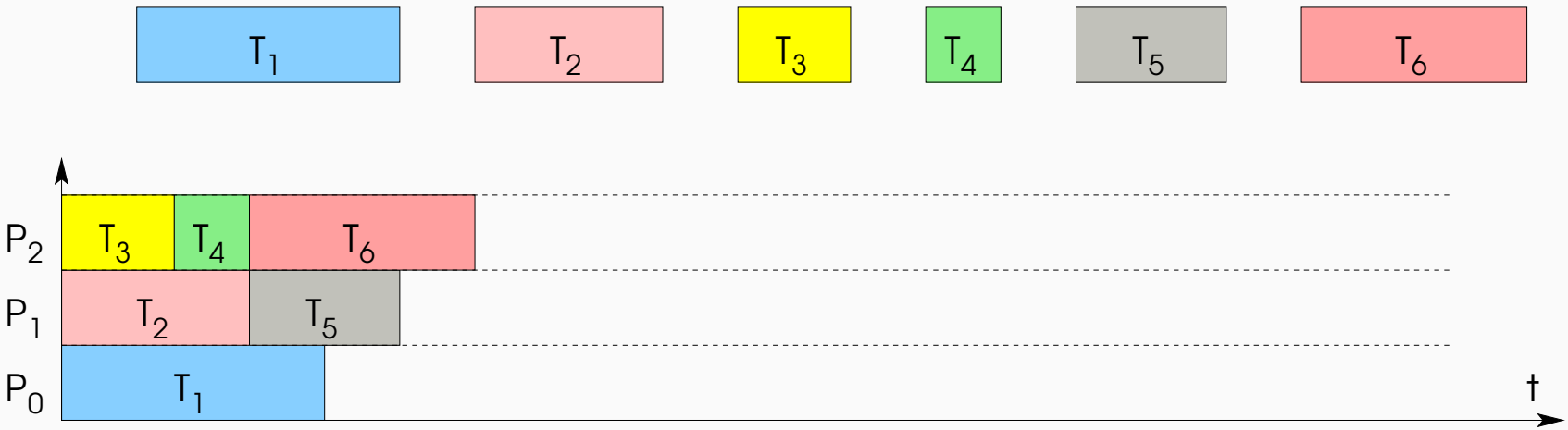
Problème *difficile* ⇒ recours à des *heuristiques*

On arrive parfois à trouver des heuristiques proches de *l'optimal*

Cas simple : tâches *indépendantes* sur unités identiques

Algorithme de liste :

- On place la tâche suivante sur la première ressource libre



Efficacité de l'algorithme de liste

Compétitivité d'une heuristique :

- Rapport entre la valeur de la solution produite par l'heuristique et la solution optimale

Théorème de Graham (1966)

Théorème 1. *Pour un ensemble de tâches indépendantes à placer sur P unités, tout algorithme de liste a un rapport de compétitivité inférieur ou égal à* $2 - \frac{1}{P}$.

Retrouver ce résultat schématiquement :

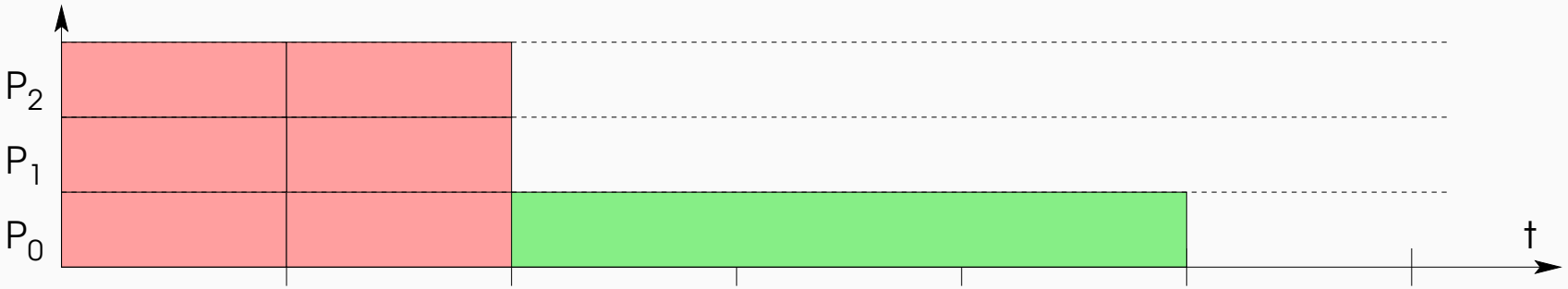


Schéma de preuve

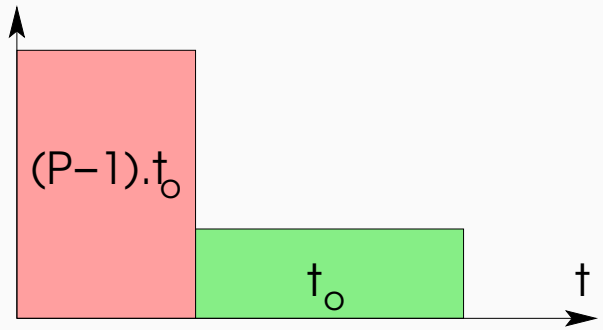
On construit une *configuration* dont on connaît une solution *optimale* :

- $n-1$ tâches :
  - placées de manière optimale sur $P-1$ unités avec un temps total t_o
  - placées de manière optimale sur P unités avec un temps total $\frac{(P-1).t_o}{P}$
- 1 tâche de durée t_o

On déduit la *pire configuration* produite par l'algorithme de liste :

- Placer la tâche longue en dernier

$\Rightarrow$ Temps total : $\frac{(P-1).t_o}{P} + t_o = \frac{(2P-1).t_o}{P}$



Le *rapport entre le pire cas et l'optimal* est : $\frac{(2P-1).t_o}{P.t_o} = \boxed{2 - \frac{1}{P}}$

Bilan du parallélisme de tâches

Lié aux *tâches indépendantes* dans les calculs :

- Le degré de parallélisme dépend de ce nombre de tâches
- Approche plus souple que pour les données :
  - Les tâches peuvent être différentes
  - Les données peuvent être dynamiques

Mais :

- Le *degré* reste souvent limité
- L'*identification* du parallélisme est moins aisée

Souvent utilisé en combinaison avec le parallélisme de données →*mixage*

- Exemple de la multiplication de matrices :
  - Parallélisme de données sur l'ensemble des `C[i][j]`
  - Parallélisme de tâches pour le calcul de chaque `C[i][j]`

Parallélisme de flux

Décomposition des calculs en *tâches successives* :

- Le résultat d'une tâche est la donnée de la tâche suivante

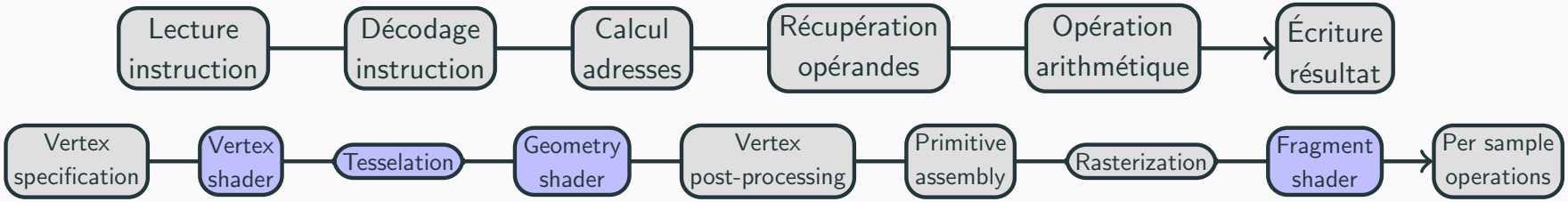
Accélération idéale pour n données, k étages et temps t par étage :

- Temps séquentiel : $n.k.t$
- Temps parallèle :
  - Traversée de la 1ère donnée $= k.t$
  - Traversée des autres données $= (n-1).t$
- Accélération :

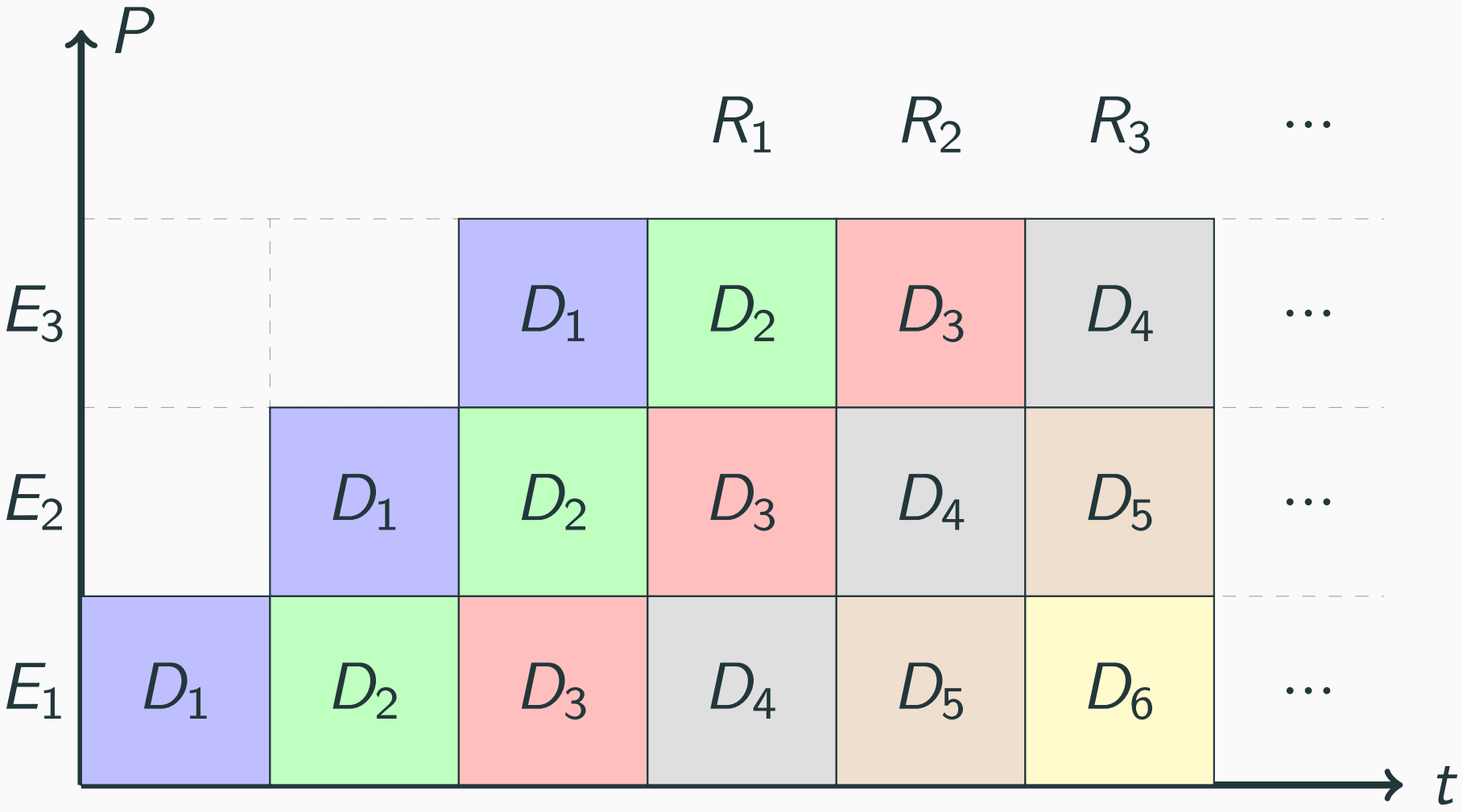
$$S_k(n) = \frac{n.k.t}{k.t + (n-1).t} = \frac{n.k}{k+n-1} \quad \Rightarrow \quad \boxed{\lim_{n\to\infty} S_k(n) = k}$$

Utilisation principale au niveau matériel :

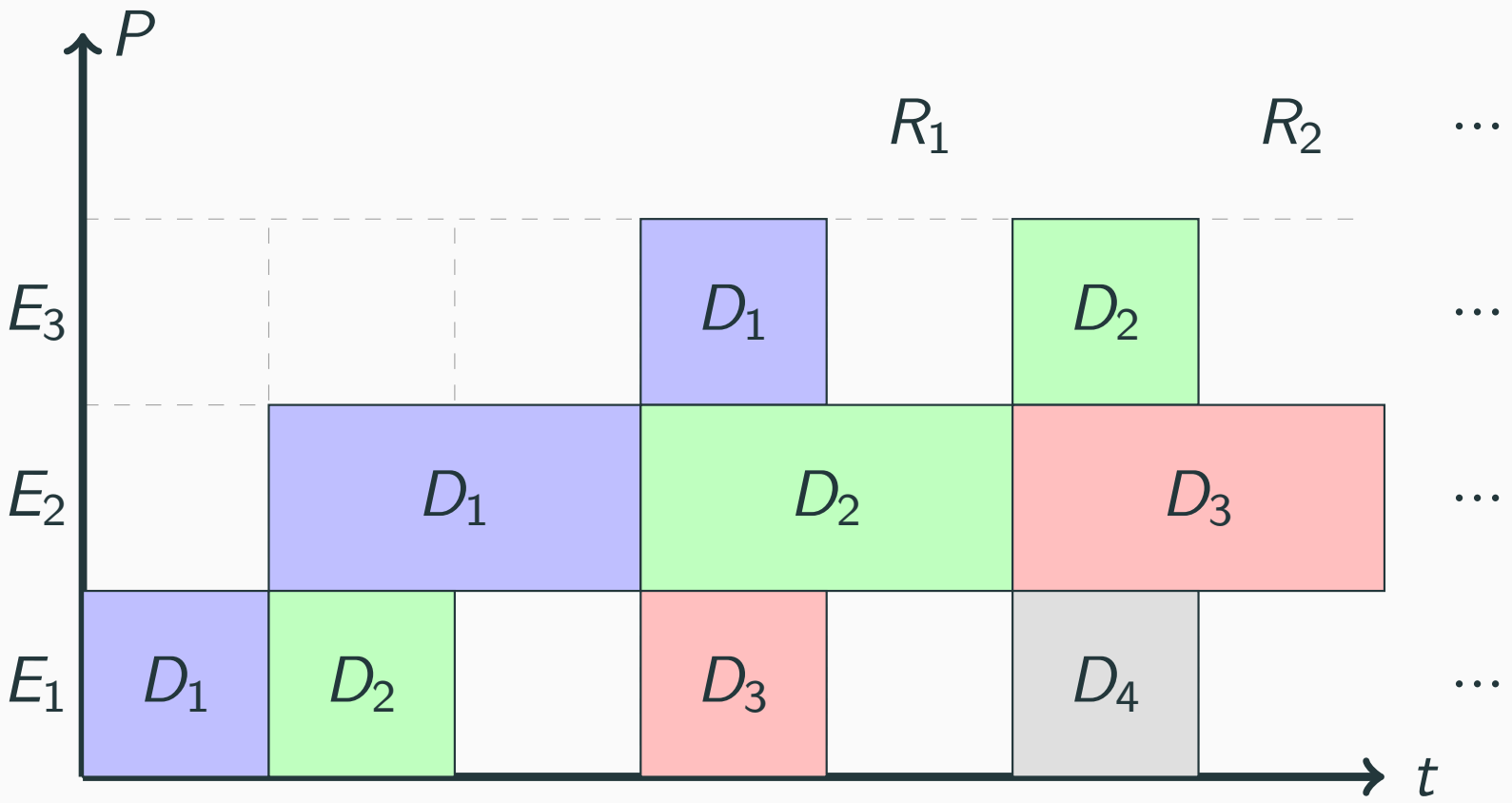
- Séquenceur d'instruction, pipeline graphique,...



Déroulement



Déroulement avec E_2 2 fois plus long que E_1 et E_3



Étages de durées différentes

Difficile de découper en blocs de même durée :

- k étages de durées respectives t_i, $1 \leq i \leq k$
- Synchronisation entre chaque étage
- Délai de sortie en régime plein : $\max\limits_{1 \leq i \leq k} t_i$

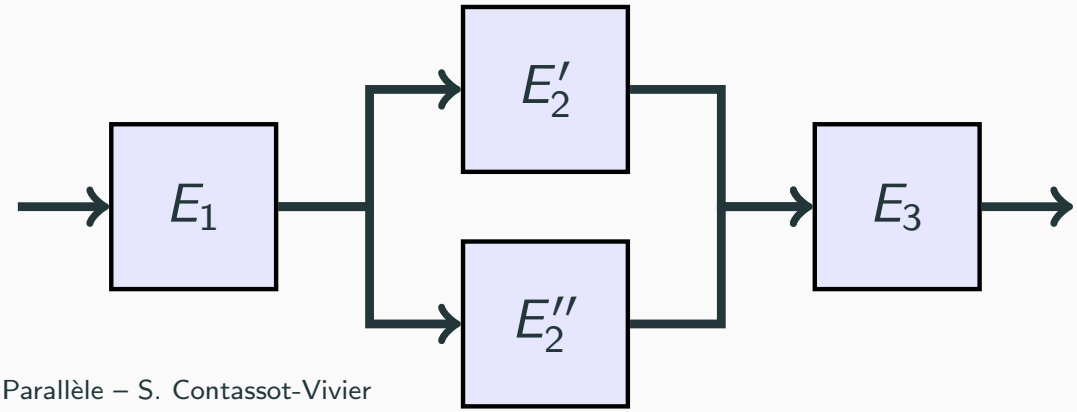
L'accélération devient :

$$S_k(n) = \frac{n \sum_{i=1}^{k} t_i}{\sum_{i=1}^{k} t_i + (n-1).\max_{1 \leq i \leq k} t_i} \quad \Rightarrow \quad \boxed{\lim_{n \to \infty} S_k(n) \leq k}$$

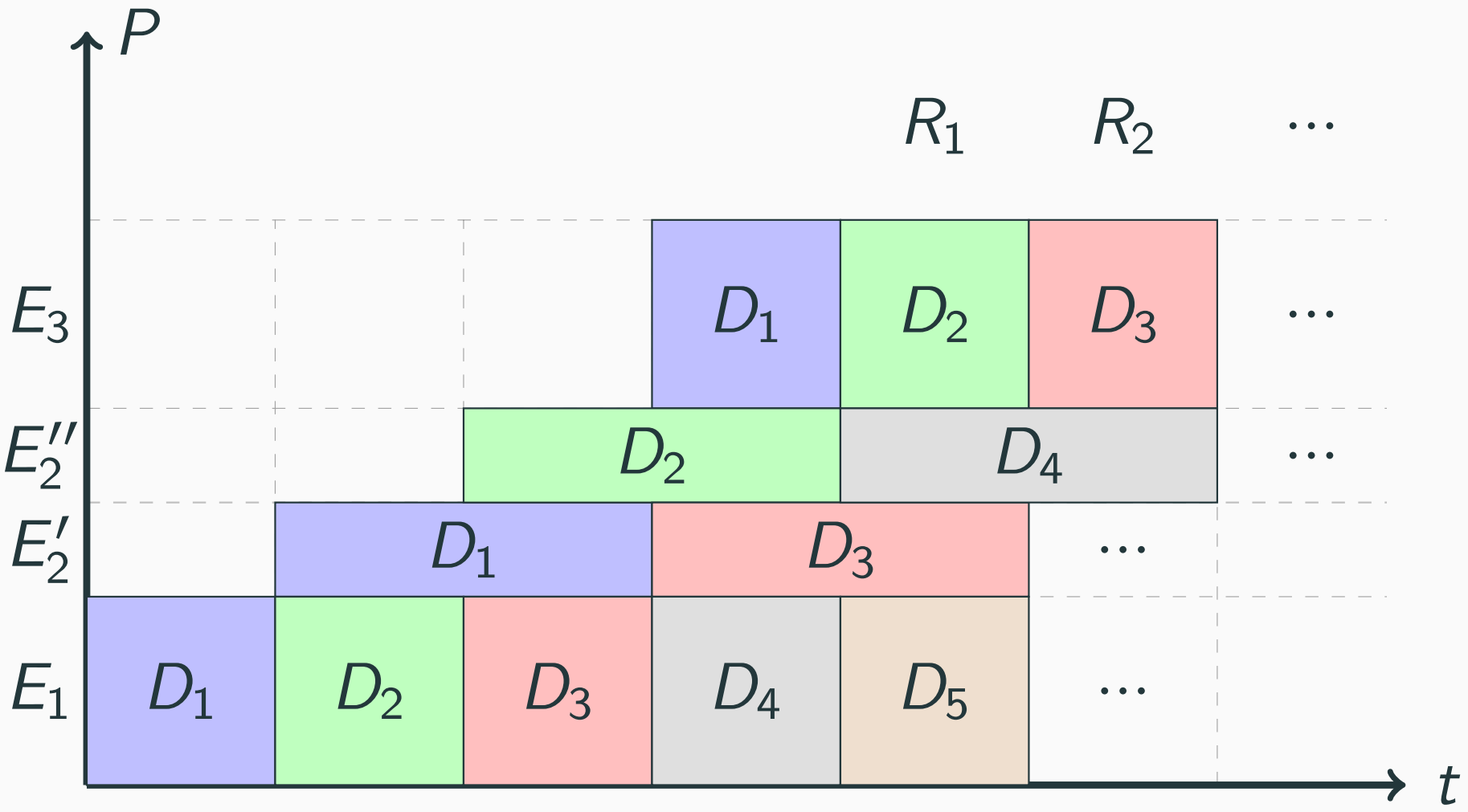
Duplication des étages :

- Exemple avec $t_2 = 2.t_1 = 2.t_3$

$\Rightarrow$ Dupliquer les étages plus longs et alterner l'utilisation



Déroulement



Utilisation dans les réseaux

Le *traitement* est le transfert d'un message de `Source` à `Destination` en passant par P liens successifs ($P-1$ nœuds intermédiaires)

Le *découpage* des messages en paquets et la retransmission *à la volée* permettent de *décomposer le traitement* en P parties successives
$\Rightarrow$*pipeline à P étages*

Exemple avec message de longueur L et P liens successifs de débit D :

- Sans découper le message, on doit le recevoir en entier sur chaque routeur avant de le transmettre $\Rightarrow$ délai total $= \frac{P.L}{D}$
- Si on découpe le message en k parties, on a :
  - Délai d'acheminement du 1er paquet : $\frac{P.L}{k.D}$
  - Délai d'acheminement des paquets suivants : $\frac{(k-1).L}{k.D}$

  $\Rightarrow$ Délai total : $\frac{(P+k-1).L}{k.D}$
- Lorsque k est suffisamment grand, le délai tend vers $\frac{L}{D}$

$\Rightarrow$ *Division par P du temps de transfert !*

Bilan du parallélisme de flux

Lié à l'*enchaînement de tâches* :

- Permet de traiter efficacement un *flot de données*
- Adapté au calcul vectoriel et aux circuits électroniques

Mais :

- Lorsque les données sont entièrement disponibles
  $\Rightarrow$ Parallélisme de données peut être préférable
- Le *découpage en tâches* de durées identiques n'est pas aisé :
  - *Duplication* éventuelle des étages plus longs

Surtout utile lorsque :

- Le flot de données est *séquentiel*
- Les *tâches* sont similaires ou de *même durée*

Programmation des systèmes à mémoire partagée

Programmation des systèmes à mémoire partagée

Principaux contextes de mémoire partagée :

- Machines multi-processeurs
- Processeurs multi-cœurs
- GPUs, Xeon-Phi

Utilisation de *threads* :

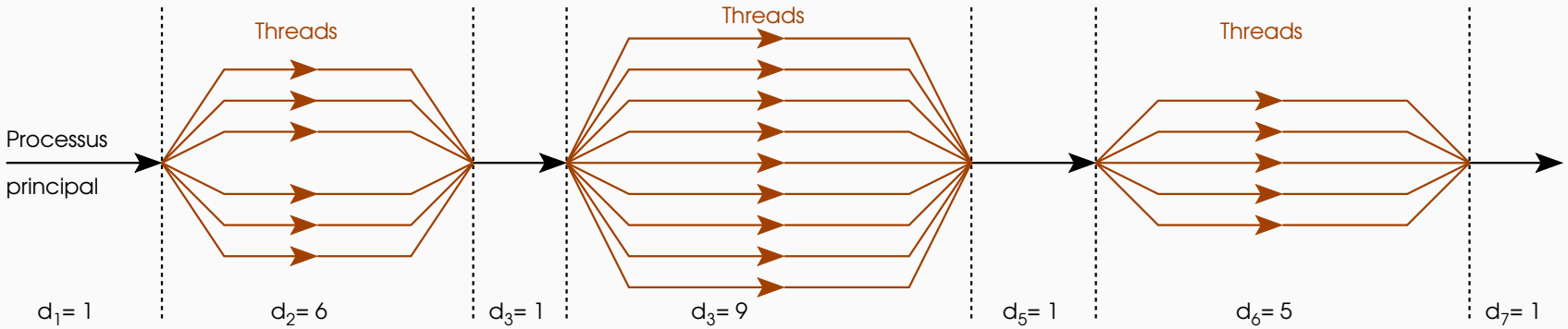
- Processus légers :
  - Partagent la mémoire du processus père
  - Peuvent être exécutés sur n'importe quel cœur
- Programmation directe :
  - Lourd et pas toujours efficace (optimisation)

⇒ Utilisation de bibliothèques spécifiques : *OpenMP*

- Directives de compilation : `#pragma omp ...`
- Enrichit un code séquentiel : deux versions en une !

Principes de base d'OpenMP

Utiliser des *threads* selon le *degré* de chaque partie d'un algorithme



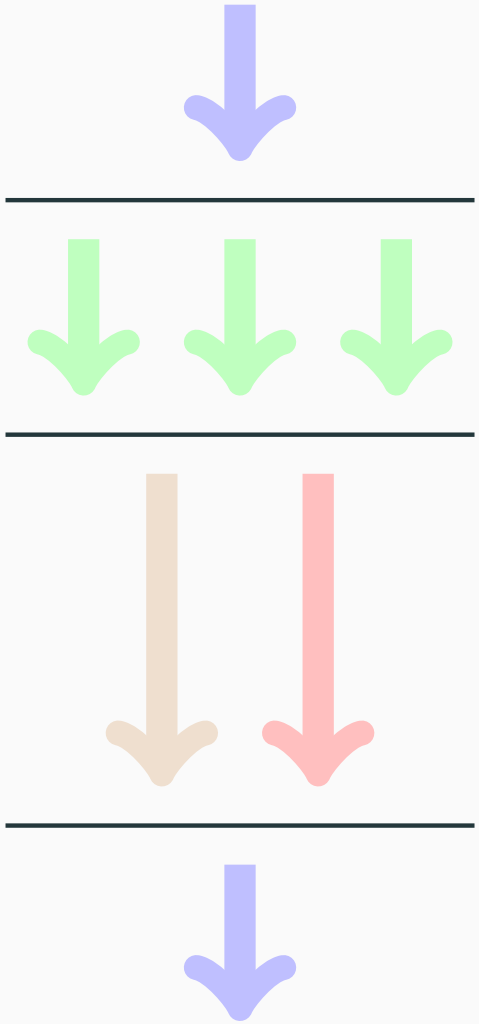
On peut partir d'un algorithme séquentiel :

- Parallélisation progressive (partie par partie)
- Validation possible après chaque partie parallélisée

Mais il est parfois nécessaire de *ré-organiser l'algorithme* initial pour obtenir une version parallèle efficace

Syntaxe et sémantique

```
int main()
{
  ...          // Partie séquentielle
  #pragma omp parallel num_threads(3)
  {            // Création de 3 threads
    ...        // Duplication du code sur 3 threads
    #pragma omp sections
    {          // Distribution de travail
      #pragma omp section
      {
        ... // Travail effectué par 1 seul thread
      }
      #pragma omp section
      {
        ... // Travail effectué par 1 autre thread
      }
    }
  }
  ...          // Retour au séquentiel
}
```



API de OpenMP

Ensemble de fonctions accessibles via :

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

Pour :

- Obtenir des informations :

```
int omp_get_num_threads() // nombre de threads actifs
int omp_get_thread_num()  // numéro du thread courant
double omp_get_wtime()    // horloge (secondes)
```

- Effectuer des réglages :

```
void omp_set_num_threads(...) // règle le nombre de threads à créer
```

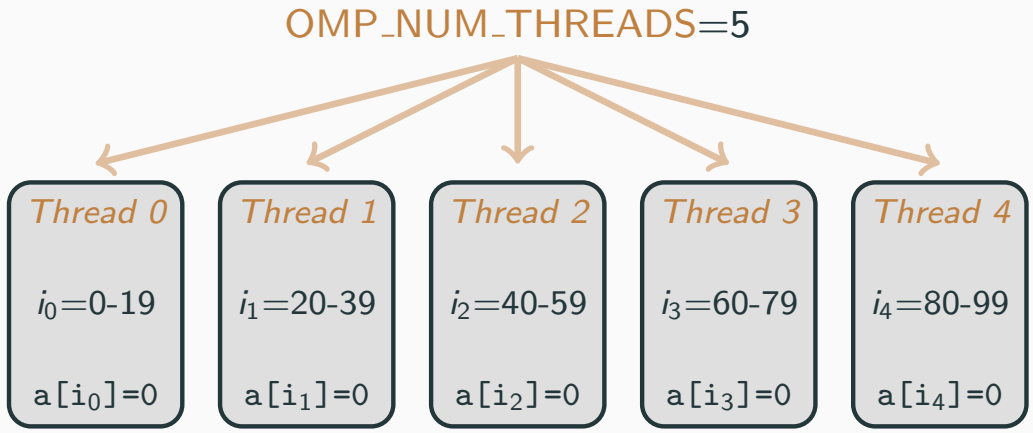
- Gérer des verrous :

```
void omp_init_lock(...)    // initialise un lock
void omp_set_lock(...)     // demande le lock (bloquant)
void omp_unset_lock(...)   // libère le lock
 int omp_test_lock(...)    // demande non bloquante de lock
void omp_destroy_lock(...) // destruction du lock
```

Directive de boucle

Répartition des itérations d'une boucle sur les threads

```
#pragma omp parallel
{
  ...
  #pragma omp for [clause,...]
  for(i=0; i<100; i++){
    a[i] = 0;
  }
  ...
}
```



La clause optionnelle permet de gérer les variables partagées :

- `private(var)` :
  - Une copie *privée* de `var` pour chaque thread
  - Spécification de l'initialisation et de la valeur finale (après la boucle)
- `shared(var)` :
  - `var` est la *même* variable pour *tous* les threads

  ⚠ *Attention* à la cohérence des manipulations ! !
  - Mode *par défaut* modifiable : `default(none)` / `default(private)`

Ordonnancement statique

```
#pragma omp for schedule(static, taille)
for(i=0; i<50; i++){
  mon_calcul(i);
}
```

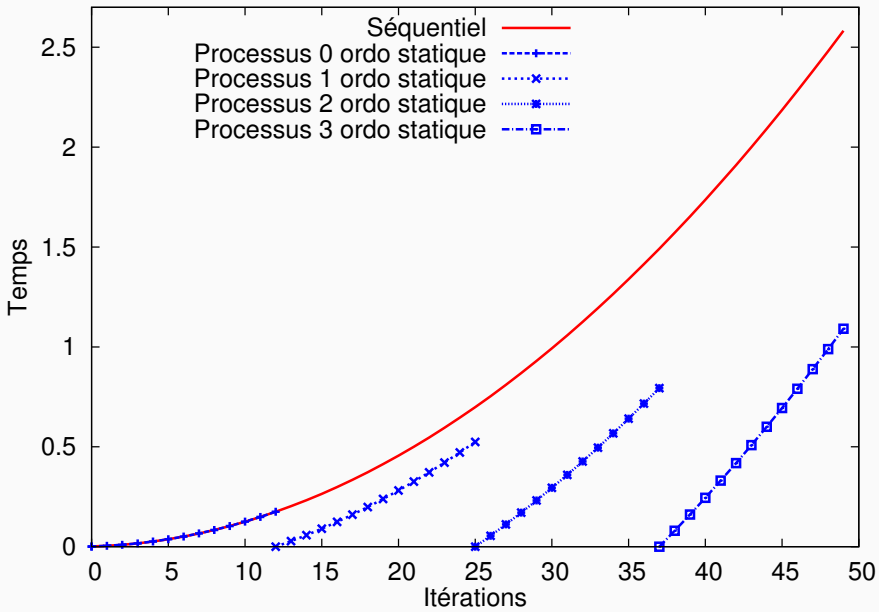
La clause `schedule(static,...)` réalise une *distribution par blocs* de la taille spécifiée :

- C'est le mode par défaut
- Si pas de taille, nombre d'itérations divisé par nombre de threads
- L'affectation des blocs suit l'*ordre cyclique* des threads

→ Bien si itérations avec *coût similaires*

Déséquilibre possible sinon :

- Itérations avec coûts croissants, exécutées sur 4 cœurs



Ordonnancement dynamique

```
#pragma omp for schedule(dynamic, taille)
for(i=0; i<50; i++){
  mon_calcul(i);
}
```

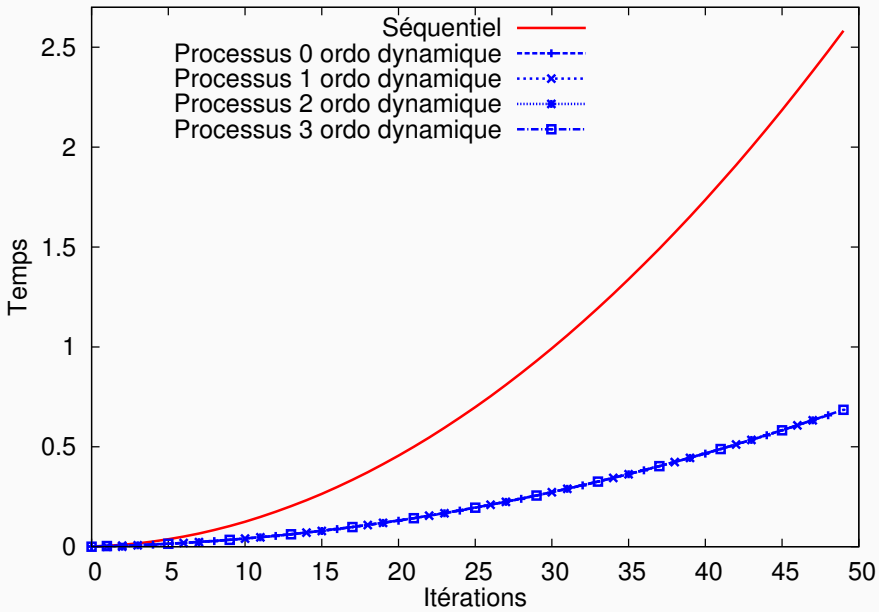
La clause `schedule(dynamic,...)` réalise une *distribution à la volée* de blocs d'itérations de la taille spécifiée (1 par défaut) :

- Chaque bloc d'itérations est attribué au *premier thread disponible*
- Surcoût dû à la gestion dynamique
- Risque de *ralentissement* si itérations similaires

→ Bien si itérations avec *coût différents*

Équilibrage implicite :

- Itérations avec coûts croissants, exécutées sur 4 cœurs



Gestion des blocs d'instructions

Plusieurs directives pour gérer l'exécution des blocs d'instructions :

- `single` :
  - Le bloc est exécuté par *un seul* thread
  - Les autres threads attendent la fin de l'exécution (*synchronisation*)
  - Utile pour éviter les interruptions de sections parallèles (*coûteux*)
- `master` :
  - Le bloc est exécuté uniquement par *le maître* (thread 0)
  - Pas de synchronisation avec les autres threads
- `critical` (ou `atomic` pour une seule instruction) :
  - Le bloc est exécuté par *tous* les threads, mais *un seul à la fois*
  - Utile pour gérer l'*exclusion mutuelle*
- `barrier` :
  - Synchronisation explicite entre les threads
  - Il y en a aussi à la fin des directives `parallel`, `sections`, `single`
  - Utile pour assurer la cohérence entre des calculs consécutifs

Illustrations

```
#pragma omp parallel
{
  int num = omp_get_thread_num();
  int nbT = omp_get_num_threads();

  printf("Thread␣%d␣parmi␣%d\n", num, nbT);

  // Partie séquentielle
  #pragma omp single
  {
    printf("Thread␣%d␣seul\n", num);
  }

  printf("Thread␣%d␣parmi␣%d\n", num, nbT);
}
```

```
#pragma omp parallel
{
  int num = omp_get_thread_num();
  int nbT = omp_get_num_threads();

  printf("Thread␣%d␣parmi␣%d\n", num, nbT);

  // Partie séquentielle
  #pragma omp master
  {
    printf("Thread␣%d␣seul\n", num);
  }

  printf("Thread␣%d␣parmi␣%d\n", num, nbT);
}
```

```
int nbT = 0; // Nombre de threads

#pragma omp parallel
{
  #pragma omp critical
  {
    nbT++;
  }
}

printf("%d␣threads␣exécutés\n", nbT);
```

```
#pragma omp parallel
{
  #pragma omp sections
  {
    ...
  }

  #pragma omp barrier

  #pragma omp for
  for(int i=0; i<N; ++i){
    ...
  }
}
```

Réduction et tâches

La clause `reduction` :

- Applicable aux directives `parallel`, `sections`, `for`
- 1er paramètre : *opération* de réduction (+, ×, min, max, &, |,...)
- 2nd paramètre : liste des *variables* (partagées) à réduire

Création dynamique de tâches : `#pragma omp task`

- Le bloc associé peut être exécuté par le thread propriétaire ou un autre thread disponible dans la section parallèle courante
- Utile pour distribuer des *tâches créées dynamiquement*
- Souvent, un seul thread crée les tâches :
  - Utilisation de `task` dans une section `single`
  - Initialisation → création des tâches → attente éventuelle (`taskwait`)

Toujours préférable d'utiliser la directive `for` quand cela est possible

Illustrations

```
int cumul = 0;

#pragma omp parallel
{
  #pragma omp for reduction(+:cumul)
  for(int i=0; i<N; ++i){
    cumul++;
  }
}

printf("Cumul = %d\n", cumul);
```

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

```
int fibo(int n)
{
  if ( n == 0 || n == 1 ) { return n; }

  int fnm1, fnm2;

  #pragma omp task shared(fnm1)
  fnm1 = fibo(n-1);

  #pragma omp task shared(fnm2)
  fnm2 = fibo(n-2);

  #pragma omp taskwait

  return fnm1 + fnm2;
}
```

Condition et désynchronisation

Parallélisme conditionnel : `#pragma omp parallel if(condition)`

- Crée les threads *seulement si* la condition est vérifiée
  Sinon, le bloc est exécuté par le processus principal →*séquentiel*
- La création des threads a un coût !
  - ⇒ Pas intéressant si pas assez de calculs à effectuer
    - On risque d'avoir *gain(parallélisme) < surcoût(parallélisme)*
- La condition porte souvent sur le nombre de données à traiter
- Le réglage nécessite de connaître les performances du système cible
  - Performance théorique, tests de référence, mesures dynamiques

Désynchronisation : `nowait`

- Clause *optionnelle* de certaines directives
- *Supprime les synchronisations* implicites en fin de bloc
- Utile pour ne pas bloquer les threads lorsque ça n'est pas nécessaire

Illustrations

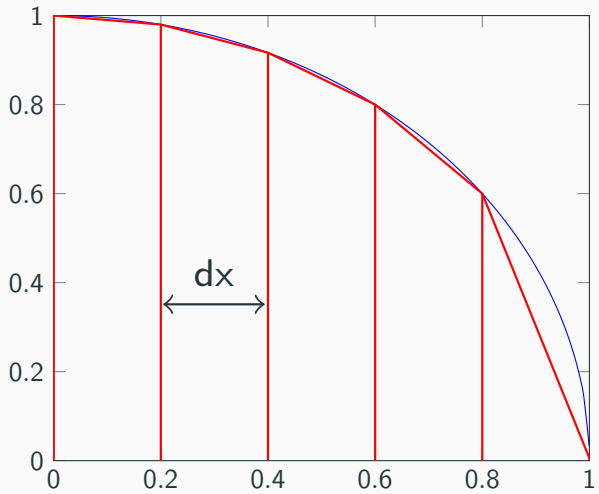
```
#pragma omp parallel if(N > 10000)
{
  #pragma omp for
  for(int i=0; i<N; ++i){
    tab[i] = f(i);
  }
}
```

```
#pragma omp parallel
{
  #pragma omp for nowait
  for(int i=0; i<N; ++i){
    a[i] = f(i);
  }

  #pragma omp for
  for(int i=0; i<N; ++i){
    b[i] = g(i); // a[] n'intervient pas
  }
}
```

Exemple du calcul de π

Calcul de π par la *méthode des trapèzes* :

- `nbTr` est le nombre de trapèzes utilisés
- `dx = 1/nbTr` est la largeur de chaque trapèze




```
pi = 0.5; // (f(0.0) + f(1.0)) / 2.0
#pragma omp parallel for private(x) reduction(+:pi)
for(i=1; i<nbTr-1; i++){
  x = dx * i;
  pi += sqrt(1.0 - x * x);
}
pi = 4.0 * dx * pi;
printf("L'approximation de PI est : %f\n", pi);
```

Bilan sur la programmation en mémoire partagée

Points forts :

- Déduction rapide d'une version parallèle à partir d'une version séquentielle
  - ⇒ Efficace si les calculs sont *réguliers* et indépendants
    - Parfois nécessaire de *ré-agencer les calculs* pour mieux paralléliser
- Possible de concevoir des algorithmes parallèles directement
- *Développement progressif* permettant la *validation par étapes*
- Peut être utilisé sur la plupart des matériels informatiques :
  - Machines multi-cœurs : ordinateurs, téléphones, tablettes,...

Points faibles :

- La gestion des *accès concurrents* est délicate et peut limiter l'efficacité
- Le *nombre d'unités* de calcul est limité à quelques dizaines
- La vitesse d'*accès à la mémoire* et sa quantité peuvent être limitant

⇒ Le recours à un parallélisme de *plus grande échelle* est souvent nécessaire

- Systèmes à mémoire distribuée, parallélisme multi-niveaux

Programmation des systèmes à mémoire distribuée

Programmation des systèmes à mémoire distribuée

Principaux contextes de mémoire distribuée :

- Machines indépendantes en réseau, super-calculateurs

$\Rightarrow$ Pas de synchronisation implicite globale !

Utilisation de *messages explicites* via des communications :

- *Point à point* :
  - Entre deux machines identifiées (*source*, *destination*)
  - *Synchrones* : avec rendez-vous (attente) entre les machines
  - *Asynchrones* : sans rendez-vous (bloquantes ou non bloquantes)
- *Communications globales* :
  - *one-to-all* : diffusion ou distribution
  - *all-to-one* : réduction ou rassemblement
  - *all-to-all* : multi-diffusion, multi-distribution,...
  - *synchronisation* : pas d'échange de données

Communications point-à-point

Deux fonctions :

- *Envoi* : `send(données, taille, dest, type)`
  - `données` : tableau contenant les données à envoyer
  - `taille` : taille des données à envoyer (en unités d'info)
  - `dest` : numéro de la machine destinataire
  - `type` : entier permettant d'appliquer un filtrage en réception
- *Réception* : `recv(données, taille, src, type)`
  - `données` : tableau où l'on stocke les données à recevoir (*déjà alloué*)
  - `taille` : taille des données à recevoir (en unités d'info)
  - `src` : numéro de la machine qui envoie les données
  - `type` : sélection du message selon l'entier spécifié

Communications *FIFO* :

- Entre une source et un destinataire, l'ordre des réceptions sur le destinataire, respecte l'ordre des envois de la source

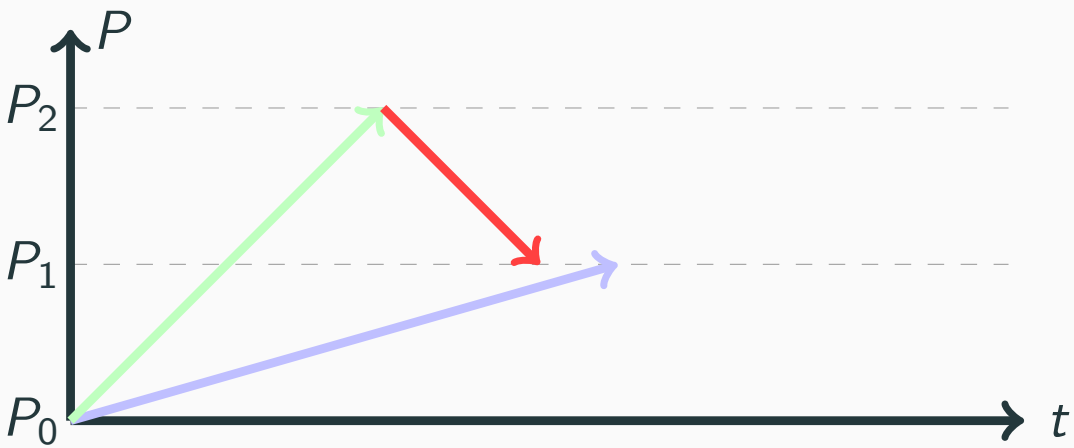
$\Rightarrow$ *Pas de croisement !*

Indéterminisme induit par les délais de communication

Les délais de communication peuvent modifier l'ordre des réceptions

Exemple :

- 3 machines avec P_0 qui envoie des données à P_1 et P_2
- Quand P_2 reçoit les données, il doit envoyer des résultats à P_1
- Quand P_1 reçoit les données de P_0, il attend les données de P_2



$\Rightarrow$ P_1 peut recevoir les résultats de P_2 *avant* les données de P_0 !

Gérer l'ordre des messages

Mémoriser les messages reçus :

- Selon leur source et leur numéro d'envoi
- Lors d'une réception, on récupère le message correspondant au numéro d'envoi suivant

$\Rightarrow$ Peut être *coûteux en mémoire* si $\text{fréq}_{arr} > \text{fréq}_{rec}$!

- Parfois nécessaire de *jeter* des messages

Imposer des *synchronisations explicites* :

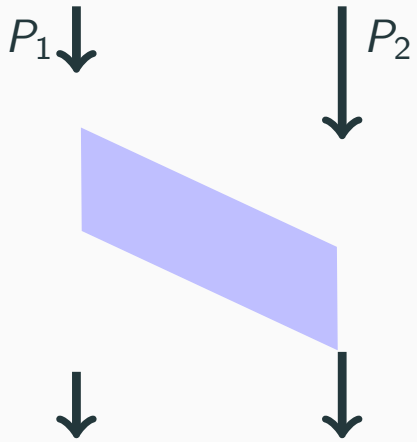
- Communications *synchrones*
- Barrières de synchronisation

$\Rightarrow$ Généralement *coûteuses en temps* !

Modes de communication

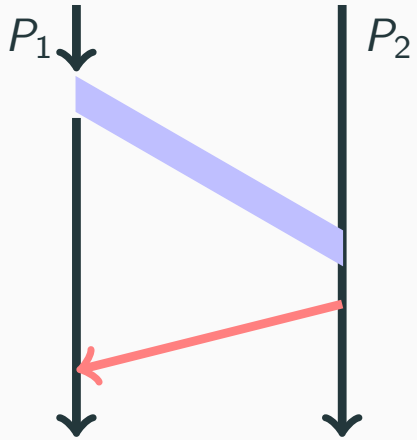
Synchrone : `ssend` et `srecv`

- Opérations *simultanées* d'envoi et de réception
  - Similaire au téléphone
  - Les deux intervenants doivent être prêts à communiquer (rendez-vous)
  - Après l'envoi, l'émetteur est sûr que le destinataire a reçu



Asynchrone :

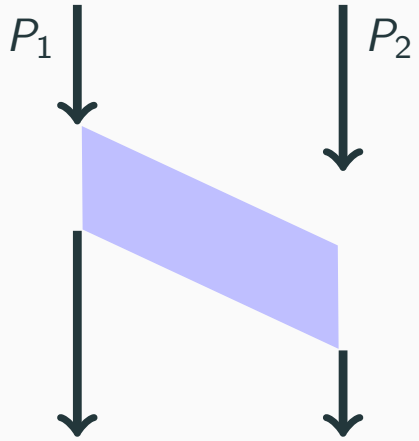
- Réception *dissociée* de l'envoi :
  - Similaire au courrier (postal/électronique)
  - Attente éventuelle du côté récepteur
  - L'émetteur ne sait pas quand le destinataire reçoit les données
    - Acquittement possible pour avoir confirmation



Modes asynchrones bloquant et non bloquant

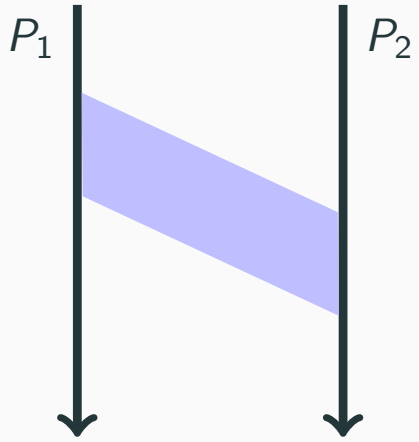
Mode bloquant : `bsend` et `brecv`

- *Attente* de la fin de l'opération de communication
- La fin de l'envoi sur la source n'implique pas que les données sont reçues sur le destinataire



Mode non bloquant : `nsend` et `nrecv`

- *Pas d'attente* de la fin de la communication
  - Déléguée à l'interface réseau
  - Capacités matérielles nécessaires : calculs *et* communications
- Permet de faire du *recouvrement calculs/communications*
- Contrôle plus délicat :
  - Assurer la *cohérence* des opérations
  - Fonctions `comDone` et `comWait`



Fonctions du mode non bloquant

Les fonctions `nsend` et `nrecv` retournent un *identifiant* de *requête*

Test de la terminaison d'une communication : `comDone(id)`

- Retourne `Vrai` si la communication `id` est finie et `Faux` sinon

Attente de la terminaison d'une communication : `comWait(id)`

- Bloque le processus tant que la communication `id` n'est pas finie

Lien entre les modes :

```
bsend(...) ≡ id ← nsend(...)
             comWait(id)
```

L'intérêt est d'*intercaler des calculs* entre le `nsend` et `comWait`

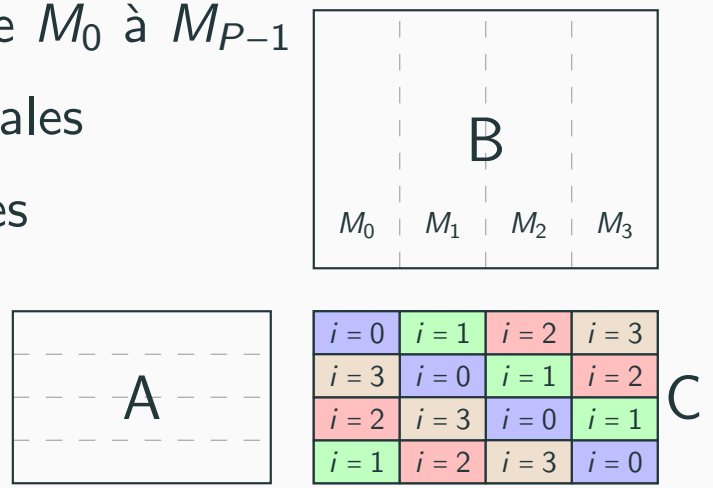
⇒*Gain de temps !*

Exemple avec la multiplication de matrices

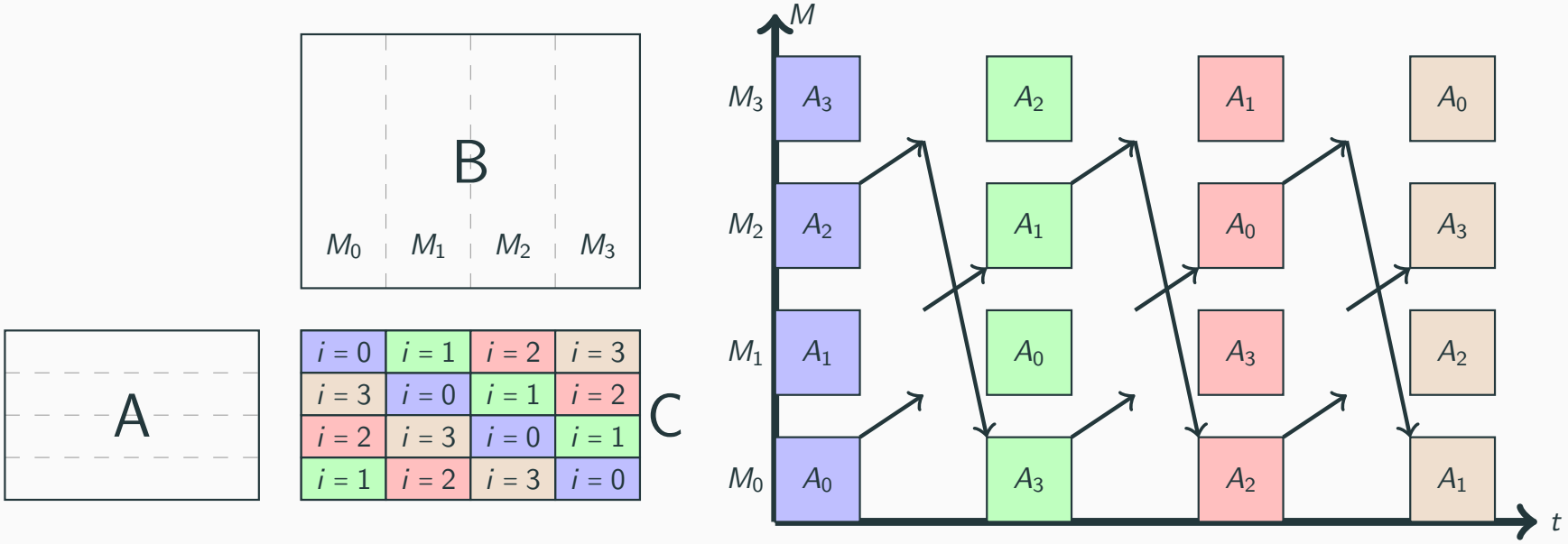
Calcul de $C = A \times B$ avec $A(m, l)$ et $B(l, n)$ sur P machines :

- Les machines sont organisées en *anneau* de M_0 à M_{P-1}
- Décomposition de A en P bandes horizontales
- Décomposition de B en P bandes verticales
- Décomposition de C en $P \times P$ blocs
- Chaque machine k $(0 \leq k \leq P-1)$:
  - Conserve la bande k de B en mémoire
  - Conserve les blocs de la colonne k de C en mémoire
  - Démarre avec la bande k de A en mémoire
- À chaque itération i $(0 \leq i \leq P-1)$, la machine k :
  - Calcule le bloc de C associé aux bandes $k+i$ de A et k de B
  - Envoie la bande $k+i$ de A à la machine suivante (cyclique)
  - Reçoit la bande $k+i-1$ de A de la machine précédente (cyclique)

$\Rightarrow$ Il faut P itérations pour réaliser la calcul complet

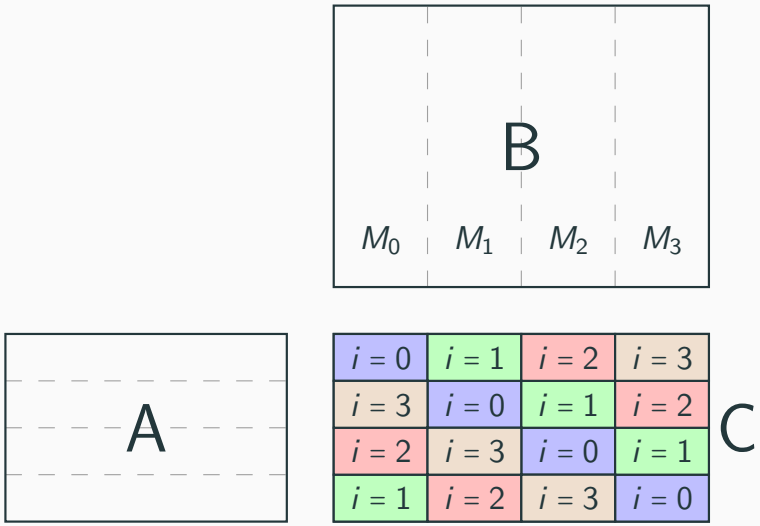


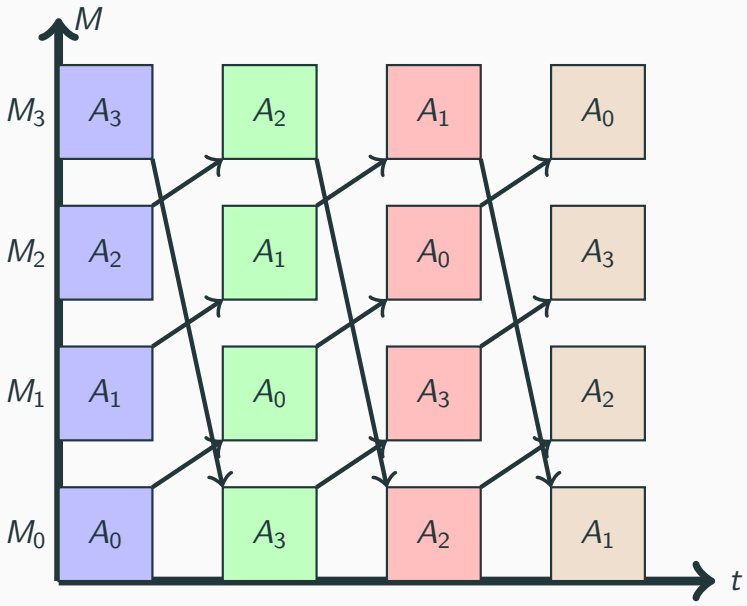
Algorithme distribué sans recouvrement




```
pour M de 0 à P-1 faire en parallèle
  pour i de 0 à P-2 faire                          // Boucle de circulation des bandes de A
    blocC ← CM[(M+P-i)%P]                          // Bloc de la colonne M de C à calculer
    multiplication(bandeAcrt, bandeB, blocC)       // Calcul du bloc de C associé aux bandes de A et B
    si M%2 = 0 alors                               // Concordance des envois/réceptions entre voisins
      ssend(bandeAcrt, (m/P)*l, (M+1)%P, 1)        // Envoi synchrone de la bande de A locale
      srecv(bandeAsvt, (m/P)*l, (M+P-1)%P, 1)      // Réception synchrone de la bande de A suivante
    sinon
      srecv(bandeAsvt, (m/P)*l, (M+P-1)%P, 1)      // Réception synchrone de la bande de A suivante
      ssend(bandeAcrt, (m/P)*l, (M+1)%P, 1)        // Envoi synchrone de la bande de A locale
    fsi
    échange(bandeAcrt, bandeAsvt)                  // Échange des bandes locales de A ...
  fpour                                            // ... pour l'itération suivante
  multiplication(bandeAcrt, bandeB, blocC)         // Calcul du dernier bloc de C pour la colonne M de B
fpour
```

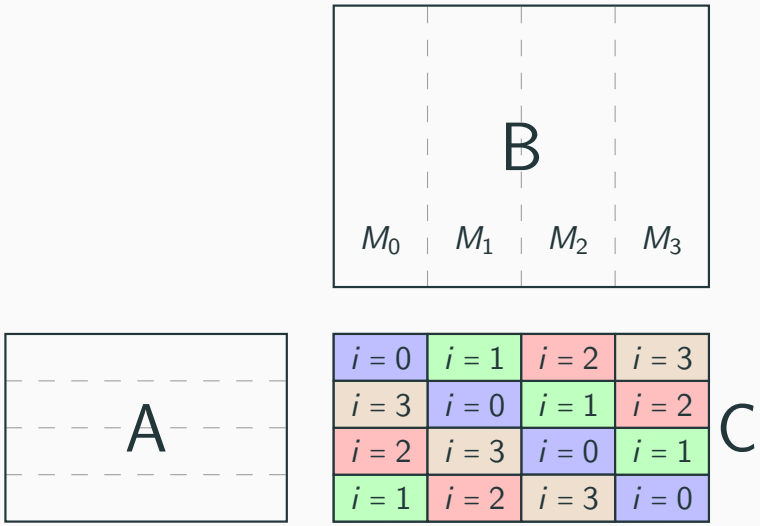
Algorithme distribué avec recouvrement des communications

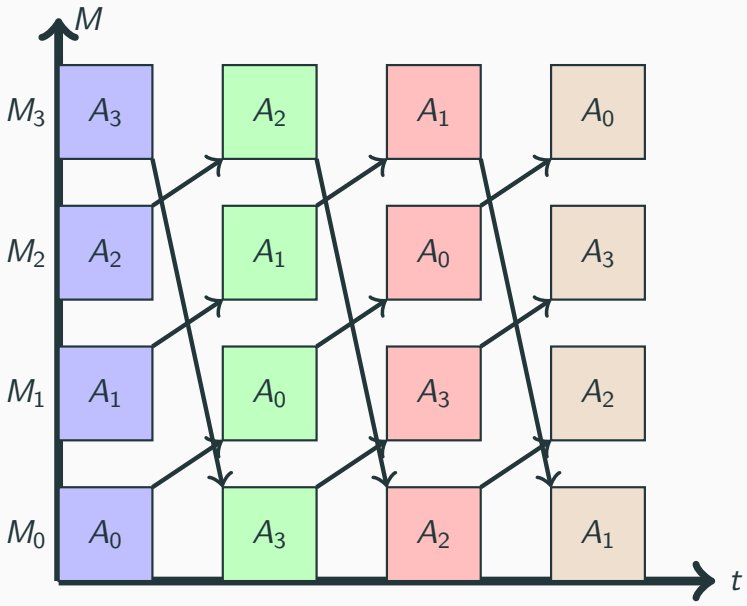





```
pour M de 0 à P-1 faire en parallèle
  pour i de 0 à P-2 faire                        // Boucle de circulation des bandes de A
    blocC ← CM[(M+P-i)%P]                        // Bloc de la colonne M de C à calculer
    multiplication(bandeAcrt, bandeB, blocC)     // Calcul du bloc de C associé aux bandes de A et B
    //
    // Version avec envois non bloquants et réceptions bloquantes
    //
    idS ← nsend(bandeAcrt, (m/P)*l, (M+1)%P, 1) // Envoi non bloquant de la bande de A locale
    brecv(bandeAsvt, (m/P)*l, (M+P-1)%P, 1)      // Réception bloquante de la bande de A suivante
    comWait(idS)                                 // Attente de la fin de l'envoi
    échange(bandeAcrt, bandeAsvt)                // Échange des bandes locales de A ...
  fpour                                          // ... pour l'itération suivante
  multiplication(bandeAcrt, bandeB, blocC)       // Calcul du dernier bloc de C pour la colonne M de B
fpour
```

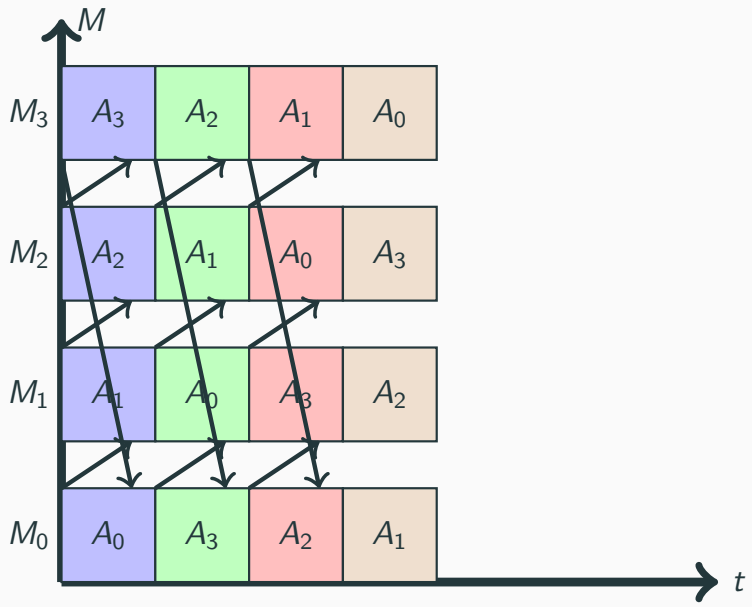
Algorithme distribué avec recouvrement des communications

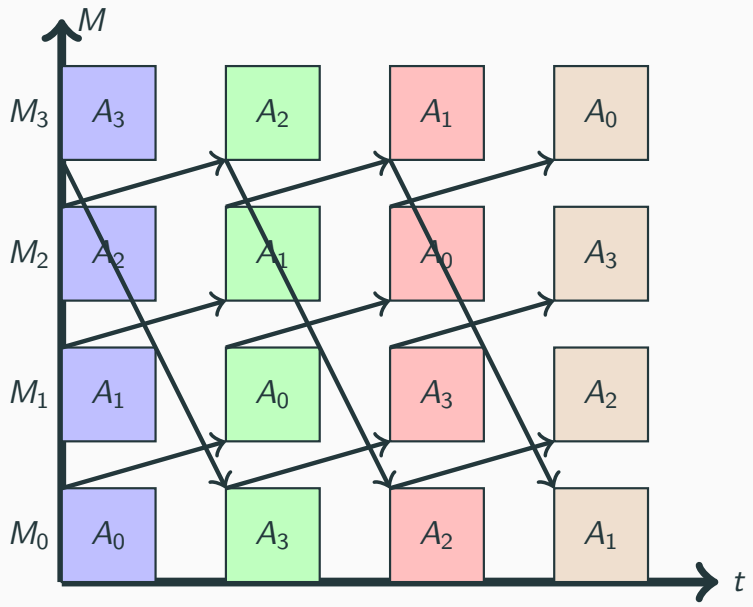





```
pour M de 0 à P-1 faire en parallèle
  pour i de 0 à P-2 faire                        // Boucle de circulation des bandes de A
    blocC ← CM[(M+P-i)%P]                        // Bloc de la colonne M de C à calculer
    multiplication(bandeAcrt, bandeB, blocC)     // Calcul du bloc de C associé aux bandes de A et B
    //
    // Version avec réceptions non bloquantes et envois bloquants
    //
    idR ← nrecv(bandeAsvt, (m/P)*l, (M+P-1)%P, 1) // Réception non bloquante de la bande de A svte
    bsend(bandeAcrt, (m/P)*l, (M+1)%P, 1)          // Envoi bloquant de la bande de A locale
    comWait(idR)                                   // Attente de la fin de la réception
    échange(bandeAcrt, bandeAsvt)                  // Échange des bandes locales de A ...
  fpour                                            // ... pour l'itération suivante
  multiplication(bandeAcrt, bandeB, blocC)     // Calcul du dernier bloc de C pour la colonne M de B
fpour
```

Algorithme avec recouvrement calculs/communications






```
pour M de 0 à P-1 faire en parallèle
  pour i de 0 à P-2 faire                              // Boucle de circulation des bandes de A
    blocC ← CM[(M+P-i)%P]                              // Bloc de la colonne M de C à calculer
    idS ← nsend(bandeAcrt, (m/P)*l, (M+1)%P, 1)        // Requête d'envoi de la bande de A locale
    idR ← nrecv(bandeAsvt, (m/P)*l, (M+P-1)%P, 1)      // Requête de réception de la bande de A suivante
    multiplication(bandeAcrt, bandeB, blocC)           // Calcul du bloc local de C
    comWait(idS)                                       // Attente de la fin de l'envoi
    comWait(idR)                                       // Attente de la fin de la réception
    échange(bandeAcrt, bandeAsvt)                      // Échange des bandes locales de A ...
  fpour                                                // ... pour l'itération suivante
  multiplication(bandeAcrt, bandeB, blocC)    // Calcul du dernier bloc de C pour la colonne M de B
fpour
```

Évaluation des performances

On pose : t_c = temps de calcul d'un bloc de C

t_b = temps de transfert d'une bande de A

Version *sans recouvrement* :

- $T_S = P.t_c + 2(P-1).t_b$

Version *avec recouvrement des communications* :

- $T_D = P.t_c + (P-1).t_b$

Version *avec recouvrement calculs/communications* :

- $T_C = (P-1).max(t_c, t_b) + t_c$

Gains →	D	C
S	$T_S - T_D = (P-1).t_b$	si $t_b \leq t_c$: $T_S - T_C = 2(P-1).t_b$ si $t_b > t_c$: $T_S - T_C = (P-1).(t_c + t_b)$
D		$T_D - T_C = (P-1).min(t_c, t_b)$

⇒ Le recouvrement permet d'*effacer l'action la moins longue* entre les calculs et les communications !

Exemples d'exécutions avec une version MPI

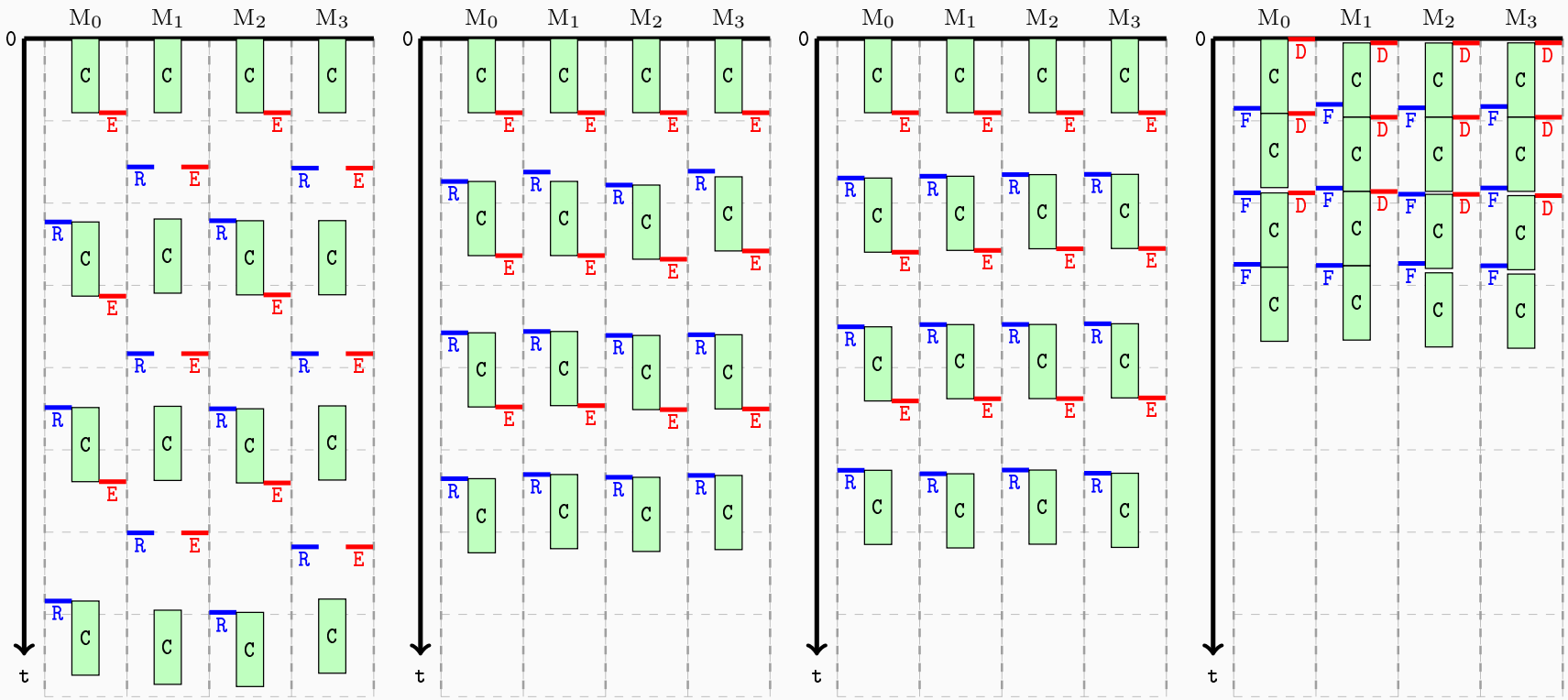
Sans recouvrement

Envois non bloquants

Réceptions non bloquantes

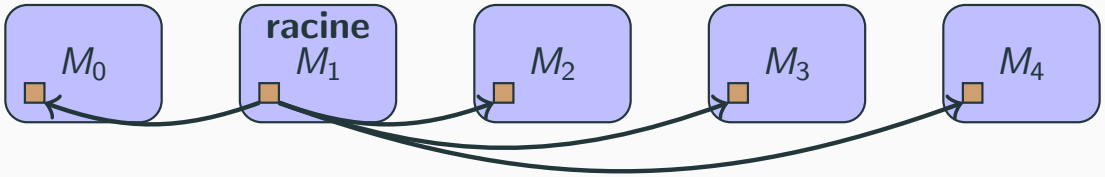
Recouvrement calculs/coms

2 threads OpenMP

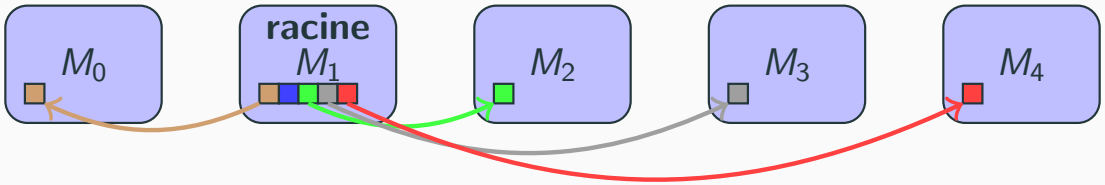


Communications globales

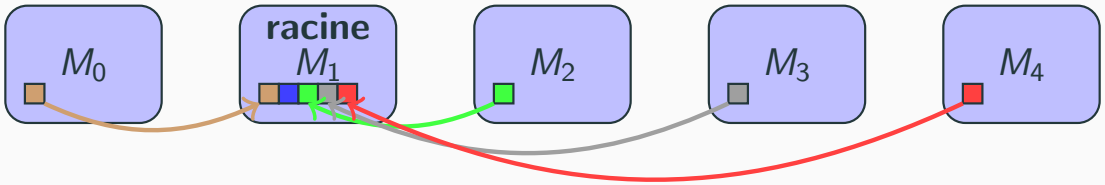
Diffusion : Un processus `racine` *envoie* les *mêmes* données à *tous les autres*



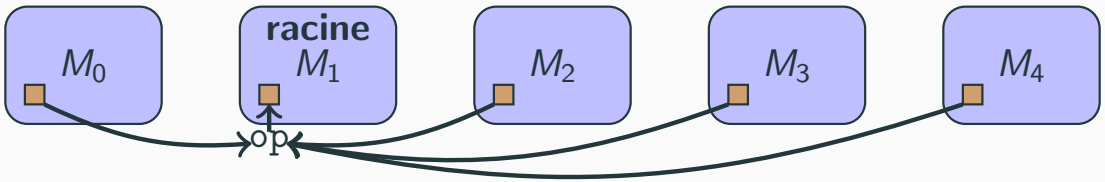
Distribution : Un processus `racine` *envoie* des données *différentes* à *chacun*



Regroupement : Un processus `racine` *agrège* des données *reçues* de *tous les autres*



Réduction : Un processus `racine` *combine* des données *reçues* de *tous les autres* via une *opération* op



Exemple de la diffusion

```
fonction diffusion(données, taille, racine)
DÉBUT
  numP ← numéroProcessus() // Récupère le numéro du processus
  nbP ← nombreProcessus()  // Récupère le nombre de processus

  si numP = racine alors // Processus racine
    // Boucle qui envoie les données aux autres processus
    pour dest de 0 à nbP-1 do
      si numP ≠ racine alors // Exclusion de la racine
        send(données, taille, dest, dtag) // valeur dtag réservée
                                          // à la diffusion
      finsi
    finpour
  sinon                    // Autres processus
    // Réception sur chaque processus différent de la racine
    recv(données, taille, racine, dtag)
  finsi
FIN
```

La synchronisation

Intérêt :

- Permet de contrôler le passage entre les étapes d'un algorithme
- Après une synchronisation, les unités savent que les autres ont également passé ce point
- Nécessaire au moins pour le démarrage et l'arrêt global

Inconvénients :

- Ralentit le déroulement global (attentes)
- Certaines communications globales ont un effet synchronisant

Principe :

- Une unité *maître* :
  - Attend un message de tous les autres
  - Une fois *tous* les messages reçus, envoie un acquittement aux autres
- Les autres unités :
  - Envoient un message au maître
  - Attendent l'acquittement

Schéma temporel

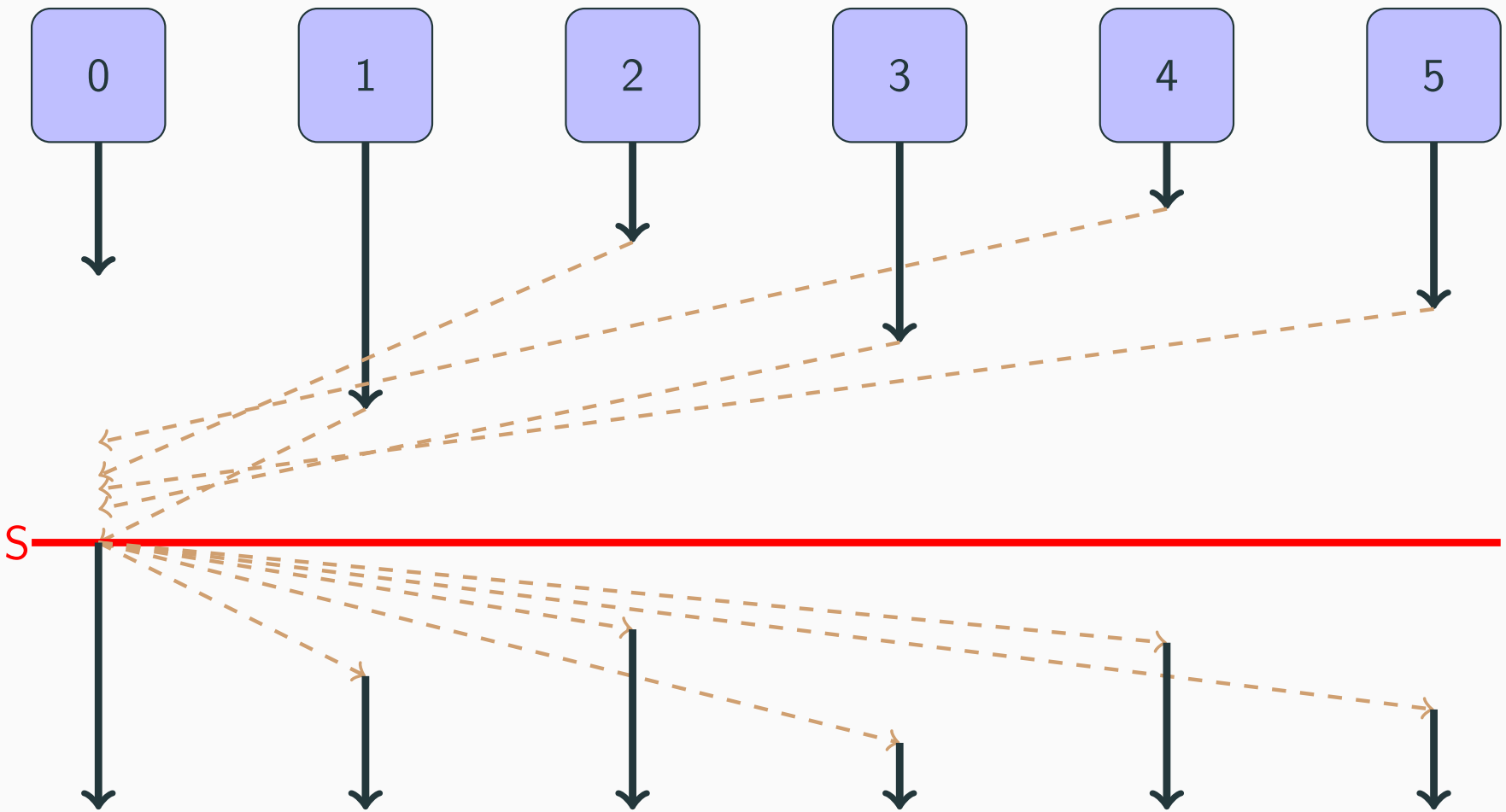


Schéma algorithmiques en mémoire distribuée

Décomposition du problème :

- Extraire un maximum de parallélisme à grain moyen/gros
- Plusieurs approches possibles :
  - Décomposition de *domaine* dirigée par les données
  - Décomposition en *tâches* (pipeline ou graphe orienté)

Répartition des données :

- *Duplication* :
  - ☺ Gestion plus simple
  - ☺ Réduction des communications → performances souvent meilleures
  - ☹ Limitant sur la taille des problèmes traités
- *Distribution* :
  - ☺ Meilleure exploitation des ressources → problèmes plus grands
  - ☹ Génère souvent plus de communications
  - ☹ Gestion plus difficile

Mise en œuvre

Recours à des bibliothèques de communication :

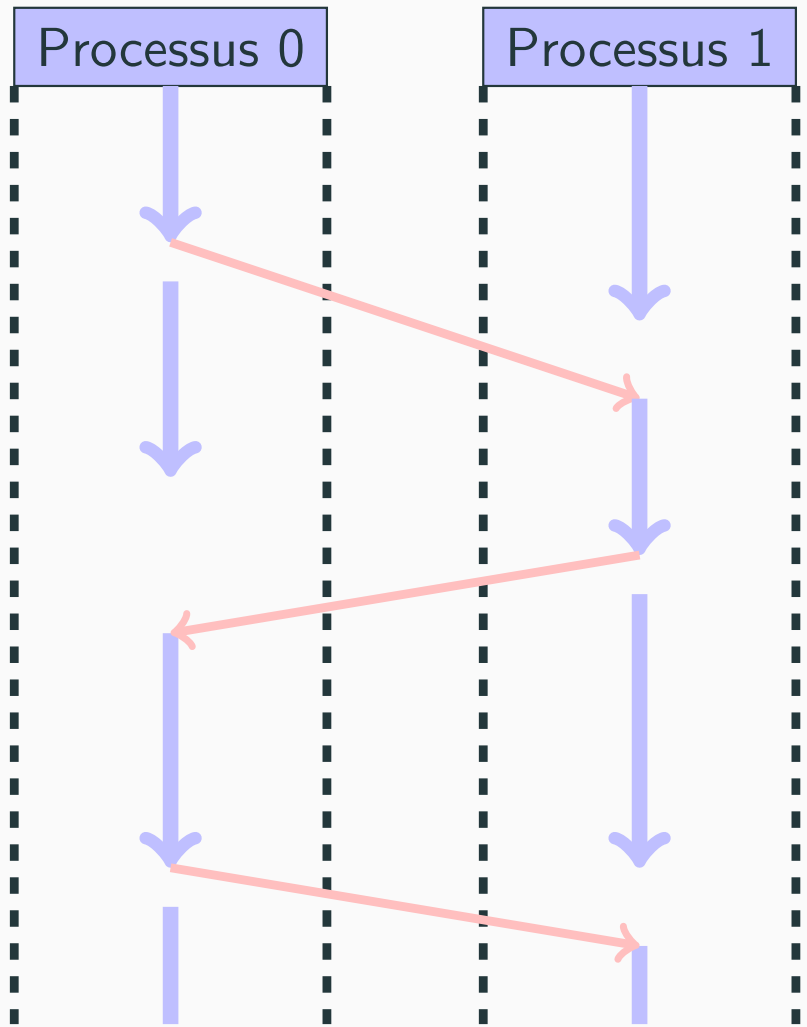
- *Explicites* : `MPI`, `PVM`
  - Fonctions d'envois et de réceptions de messages
- *Implicites* :
  - Environnements basés sur les *RPC* (Remote Procedure Call)
    - Appel d'une fonction potentiellement localisée sur une autre machine
    - Les paramètres de la fonction sont envoyées via un message
    - Les résultats de la fonction sont reçus via un message
  - Environnements à *mémoire partagée virtuelle* :
    - Les machines voient une seule *mémoire globale partagée* (virtuelle)
    - Génération auto de messages pour accéder aux données non locales
    - Permet de réutiliser facilement des codes MP
      mais pas toujours de bonnes performances

Principes de base de MPI

Plusieurs processus indépendants échangent des données via des messages

Schéma *SPMD* :

- Un *même programme* exécuté par tous les processus
- Différents branchements pour *différencier le travail* à effectuer :
  - Calculs, communications (envois/réceptions)
- *Correspondance des envois/réceptions* pour les modes synchrones et bloquants
- Souvent, un processus joue un rôle central :
  - *Maître - Travailleurs*
  - Interface avec extérieur (E/S)



Fonctions principales de MPI en C

Inclusion de l'API :

- `#include <mpi.h>`

Initialisation et fermeture :

- `MPI_Init(&argc, &argv);` démarre l'environnement de communication
- `MPI_Finalize();` arrête l'environnement de communication
- *Toutes* les communications doivent être entre ces deux appels !

Informations :

- `MPI_Comm_size(MPI_COMM_WORLD, &nbP);` nombre de processus
- `MPI_Comm_rank(MPI_COMM_WORLD, &numP);` numéro du processus
- `MPI_Wtime();` horloge (réel)

Communications point à point :

- On retrouve les modes synchrone et asynchrone bloquant ou non

Synchrone :	`MPI_Ssend()`	`MPI_Recv()`
Bloquant :	`MPI_Send()`	`MPI_Recv()`
Non bloquant :	`MPI_Isend()`	`MPI_Irecv()`

Fonctions élémentaires d'envoi/réception

Envoi : `MPI_Send(données, nombre, type, dest, tag, comm)`

- `données` : tableau des données à envoyer
- `nombre` : nombre de données à envoyer
- `type` : type des données à envoyer parmi une liste pré-définie
  - `MPI_CHAR`, `MPI_BYTE`, `MPI_INT`, `MPI_DOUBLE`, ...
- `dest` : numéro du processus destinataire dans le groupe `comm`
- `tag` : étiquette du message permettant un filtrage
- `comm` : groupe de processus (`MPI_COMM_WORLD` pour *tous*)

Réception : `MPI_Recv(données, nombre, type, src, tag, comm, état)`

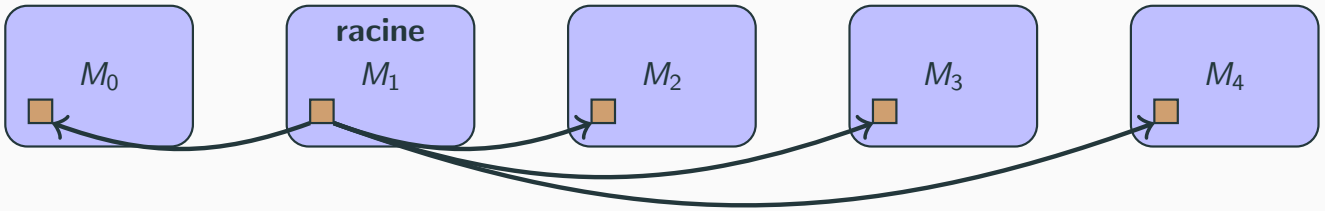
- `données` : tableau où sont stockées les données à recevoir

  $\Rightarrow$ Espace *alloué avant* la réception et de *taille suffisante*
- `src` : numéro du processus source dans le groupe `comm`
- `tag` : filtrage des messages selon cette étiquette
- `état` : informations sur l'état de la réception (`NULL` si non utilisé)

Communications globales un-vers-tous (one-to-all)

Diffusion : `MPI_Bcast(données, nombre, type, racine, comm)`

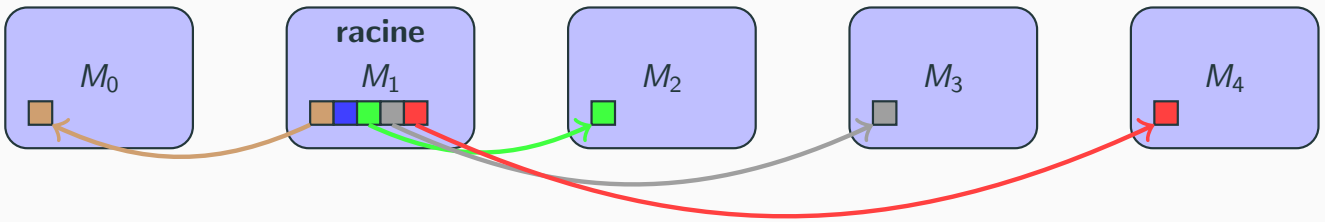
- `données` : *envoyées* par `racine` et *reçues* par les autres
- `racine` : numéro du processus qui diffuse aux autres
- Actions différentes selon les processus →*masquage*



Distribution :

`MPI_Scatter(donE, nbE, typeE, donR, nbR, typeR, racine, comm)`

- `donE` : tableau des données à distribuer (racine comprise)
- `nbE` : nombre de données à distribuer par processus
- `donR` : espace mémoire des données à recevoir
- `racine` : numéro du processus qui distribue les données



MPI_Bcast avec MPI_Send et MPI_Recv

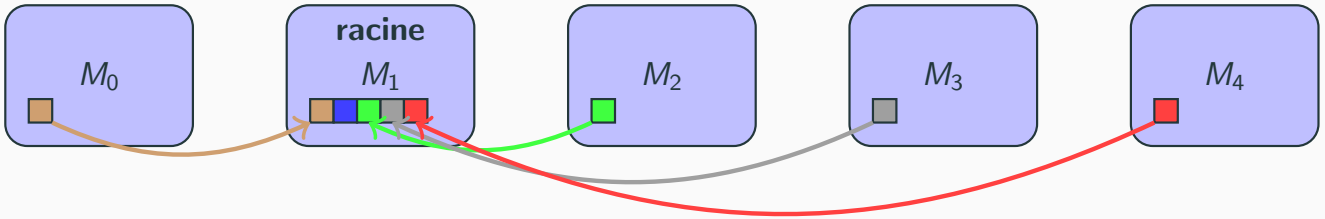
```
void MPI_Bcast(void *donnees, int nombre, MPI_Datatype type,
                              int racine, MPI_Comm comm)
{
  int i, num, nbP; // Compteur, num processus et nombre de processus
  MPI_Status etat; // État de la réception

  MPI_Comm_rank(comm, &num); // Numéro du processus
  MPI_Comm_size(comm, &nbP); // Nombre de processus

  if(num == racine){ // Processus racine
    // Boucle d'envoi aux autres processus
    for(i=0; i<nbP; ++i){
      if(i != racine){ // Exclusion de la racine (a déjà les données)
        MPI_Send(donnees, nombre, type, i, 1, comm);
      }
    }
  }else{ // Autres processus
    // Réception sur chaque processus autre que la racine
    MPI_Recv(donnees, nombre, type, racine, 1, comm, &etat);
  }
}
```

Communications globales tous-vers-un (all-to-one)

Regroupement :

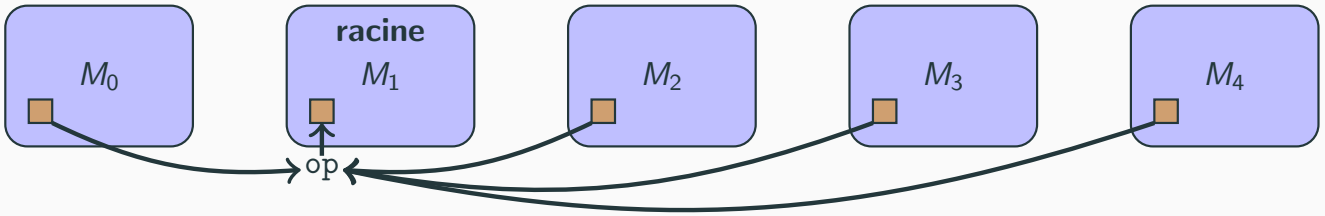
```
MPI_Gather(donE, nbE, typeE, donR, nbR, typeR, racine, comm)
```

- `donR` : doit pouvoir contenir *toutes* les données à recevoir
- `nbR` : nombre de données à recevoir de *chaque* processus



Réduction : `MPI_Reduce(donE, donR, nb, type, op, racine, comm)`

- `donE` : peut être un tableau → résultat est un tableau (op élt à élt)
- op : parmi un ensemble prédéfini : `MPI_MAX`, `MPI_SUM`, `MPI_PROD` ,...



Exemple du calcul de π par la méthode des trapèzes

Version avec *deux niveaux de parallélisme* :

→ Ensemble de machines (mémoire *distribuée*) : `MPI`

→ Ensemble de cœurs (mémoire *partagée*) : `OpenMP`

```
MPI_Comm_rank(MPI_COMM_WORLD, &num); // Numéro du processus MPI
MPI_Comm_size(MPI_COMM_WORLD, &nbP); // Nombre de processus MPI
piLoc = 0.0;                         // Valeur partielle de pi calculée par num

// Diffusion du nombre total de trapèzes
MPI_Bcast(&nbTr, 1, MPI_INT, 0, MPI_COMM_WORLD);

dx = 1.0 / nbTr;      // Déduction de la largeur des trapèzes
nbTrLoc = nbTr / nbP; // Cas simple où la division tombe juste

// Intégrale partielle de pi associée au processus num avec nbT threads OpenMP
#pragma omp parallel for private(x) reduction(+:piLoc) num_threads(nbT)
for(i=num*nbTrLoc; i<(num+1)*nbTrLoc; i++){
  x = dx * i;
  piLoc += sqrt(1.0 - x * x);
}

// Somme des intégrales partielles de pi de tous les processus MPI vers le processus 0
MPI_Reduce(&piLoc, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

if(num == 0){ // Calcul et affichage du résultat global par le processus 0
  pi = 4.0 * dx * (pi + 0.5);                    // Résultat final (uniquement sur le processus 0)
  printf("L'approximation de PI est : %f\n", pi); // Affichage
}
```

Fonctions non bloquantes d'envoi/réception

Envoi : `MPI_Isend(données, nombre, type, dest, tag, comm, requête)`

- Retour immédiat de la fonction →*délégation* de l'envoi des données
- Paramètres similaires à `MPI_Send` sauf le dernier
- `requête` : pointeur sur une variable de type `MPI_Request` (identifiant de la communication)

Réception : `MPI_Irecv(données, nombre, type, src, tag, comm, requête)`

- Similaire à l'envoi mais côté réception

Test : `MPI_Test(requête, réponse, état)`

- `requête` : pointeur sur la requête de communication à tester
- `réponse` : pointeur sur le résultat du test (entier 1 si finie, 0 sinon)
- `état` : pointeur sur les informations d'état de la communication

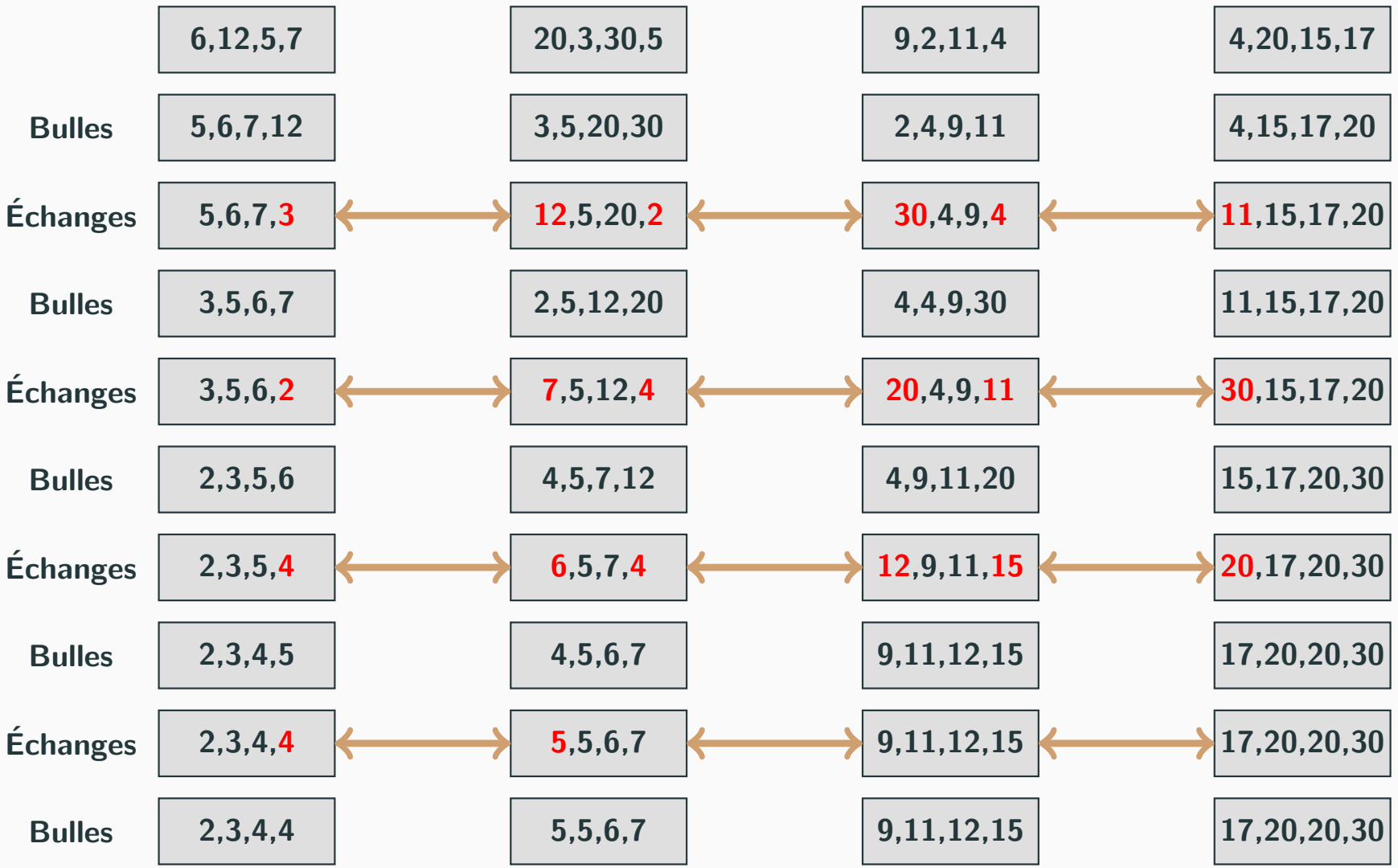
Attente : `MPI_Wait(requête, état)`

- Blocage en attente tant que la communication n'est pas terminée
- `requête` : pointeur sur la requête de communication à attendre

Exemple du double tri à bulles

```
MPI_Request reqP, reqD; // Requêtes d'envoi du premier/dernier élt
char triOK = 0;         // Booléen indiquant si les données sont globalement triées
nbLoc = TAILLE/nbP;     // Taille du tableau local tLoc
// Distribution des données initiales de tab aux processus MPI
MPI_Scatter(tab, nbLoc, MPI_INT, tLoc, nbLoc, MPI_INT, 0, MPI_COMM_WORLD);
while(!triOK){ // Succession de parcours avec remontée des bulles
               // jusqu'à ce que les données soient globalement triées
  for(j=0; j<nbLoc-1; ++j){ // Parcours gauche -> droite
    if(tLoc[j] > tLoc[j+1]){
      ... // Échange des valeurs
    }
  } // À la fin de cette boucle, tLoc[nbLoc-1] contient le max local
  for(j=nbLoc-3; j>0; --j){ // Parcours droite -> gauche
    if(tLoc[j] > tLoc[j+1]){
      ... // Échange des valeurs
    }
  } // À la fin de cette boucle, tLoc[0] contient le min local
  if(num > 0){ // Envoi du 1er élt (min) à gauche et réception du dernier élt (max) de gauche
    MPI_Isend(&tLoc[0], 1, MPI_INT, num-1, 1, MPI_COMM_WORLD, &reqP);
    MPI_Recv(&prec, 1, MPI_INT, num-1, 1, MPI_COMM_WORLD, &etat);
    MPI_Wait(&reqP, &etat); // Attente de la fin d'envoi du 1er élt
    if(prec > tLoc[0]) tLoc[0] = prec; // Remplacement éventuel du 1er élt
  }
  if(num < nbP - 1){ // Envoi du dernier élt (max) à droite et réception du 1er élt (min) de droite
    MPI_Isend(&tLoc[nbLoc-1], 1, MPI_INT, num+1, 1, MPI_COMM_WORLD, &reqD);
    MPI_Recv(&svt, 1, MPI_INT, num+1, 1, MPI_COMM_WORLD, &etat);
    MPI_Wait(&reqD, &etat); // Attente de la fin d'envoi du dernier élt
    if(svt < tLoc[nbLoc-1]) tLoc[nbLoc-1] = svt; // Remplacement éventuel du dernier élt
  }
  triOK = estGlobalementTrie(); // Vérifie si les données sont globalement triées
}
```

Schéma d'exécution



Fonctions principales de MPI en Python

Inclusion de l'API : `from mpi4py import MPI`

Initialisation et fermeture :

- Prises en charge via le chargement du module `MPI` de `mpi4py`

Groupe de communication :

- Global : `comm = MPI.COMM_WORLD`

Informations :

- Nombre de processus : `comm.Get_size()`
- Numéro du processus : `comm.Get_rank()`
- Horloge (s) : `MPI.Wtime()`

Communications point à point :

- On retrouve les modes asynchrones bloquant et non bloquant

  Bloquant : `comm.send()` `comm.recv()`

  Non bloquant : `comm.isend()` `comm.irecv()`
- Distinction entre communication d'*objets python* et de *tableaux*

  Version objets (`comm.send`) et version tableaux (`comm.Send`)

Fonctions standard d'envoi/réception

Send : `comm.send(objet, dest=..., tag=...)`

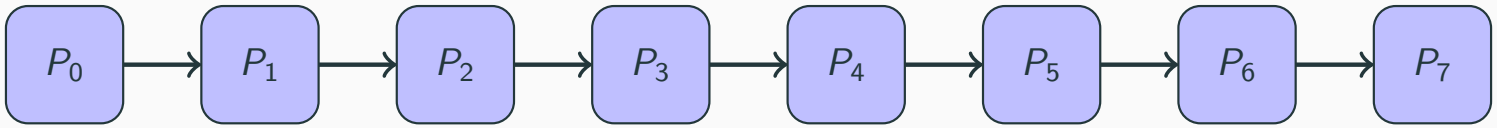
- `comm` : *groupe* de communication
- `objet` : *objet* Python à envoyer
- `dest` : processus *destinataire* dans le groupe `comm`
- `tag` : *étiquette* du message (entier)

Receive : `objet = comm.recv(source=..., tag=...)`

- `comm` : *groupe* de communication
- `source` : processus *source* dans le groupe `comm`
- `tag` : *étiquette* de *filtrage* de réception
- `objet` : *objet* Python reçu

Exemple de circulation d'entier du 1er au dernier processus

Organisation *logique* en *ligne* : exemple avec 8 processus




```
comm = MPI.COMM_WORLD # Définition du groupe de communication
idP = comm.Get_rank() # ID du processus
nbP = comm.Get_size() # Nombre total de processus

if (idP > 0):
   # Les processus qui ont un prédecesseur
   # ATTENDENT un message de celui-ci (com BLOQUANTE)
   val = comm.recv(source=idP-1, tag=10)
   # Affichage de la valeur recue
   print(idP, ":", val)
else:
   print()

# Chaque processus affiche son ID et la taille du groupe
print("Salut depuis %d parmi %d" % (idP, nbP))

if (idP < nbP-1):
   # Les processus qui ont un successeur lui envoient leur ID
   # --> Le processus 0 débloque le processus 1, et ainsi de suite...
   comm.send(idP, dest=idP+1, tag=10)
   # L'étiquette doit correspondre à celle de la réception
```

Bilan sur la programmation en mémoire distribuée

Points forts :

- Possibilité d'agréger une puissance de calcul très importante :
  ⇒ *Passage à l'échelle* pour traiter des problèmes de grande taille
- Mécanismes relativement simples pour transférer des données
- Possibilité de *recouvrement calculs/communications*
  ⇒ Recours éventuel à plusieurs threads

Points faibles :

- *Coût* des communications !
- Correspondance nécessaire entre envois et réceptions pour les échanges synchrones ou bloquants
  ⇒ Risque d'*inter-blocages* !
- *Distribution* des données souvent *fastidieuse*
  - Identification d'une distribution efficace pas toujours aisée

⇒ **Parallélisme à grande échelle mais qui nécessite généralement un parallélisme interne aux machines pour une efficacité maximale**

Équilibrage des charges de travail

Équilibrage des charges de travail

Dans un système parallèle, la *répartition homogène* des tâches ou données n'est *pas toujours efficace*

Des *déséquilibres* peuvent provenir de :

- L'*hétérogénéité des machines* :
  - Vitesses ou charges différentes des unités de calcul
- L'*hétérogénéité des traitements* :
  - Tâches ayant des quantités de calcul différentes
  - Tâches générées dynamiquement pendant l'exécution de l'algorithme
- L'*hétérogénéité des données* :
  - Évolution dynamique du nombre de données à traiter
  - Le traitement d'une donnée peut générer d'autres données à traiter

$\Rightarrow$ L'objectif de l'équilibrage de charges consiste à répartir le travail afin que toutes les unités *terminent en même temps*

Exemple simple

On considère :

- Trois unités P_0, P_1 et P_2 avec les vitesses suivantes :
  - $V_0 = V_1 = V$ et $V_2 = 2V$ op/s
- Un calcul parallèle C dont le nombre d'opérations par donnée est α

Distribution homogène de N données sur les trois unités :

- $T(C(\frac{N}{3}, P_0)) = T(C(\frac{N}{3}, P_1)) = \frac{\alpha.N}{3V}$ et $T(C(\frac{N}{3}, P_2)) = \frac{\alpha.N}{6V}$

$\Rightarrow$ $\boxed{T_3(N) = \frac{\alpha.N}{3V}}$

Peut-on faire mieux ?

- *Oui*, en attribuant plus de données à P_2
- Il faut que : $T(C(N_0), P_0)) = T(C(N_1), P_1)) = T(C(N_2), P_2)$

$\Rightarrow \frac{N_0}{V} = \frac{N_1}{V} = \frac{N_2}{2V} \Rightarrow N_0 = N_1$ et $N_2 = 2N_0 = 2N_1 \Rightarrow \boxed{T_3(N) = \frac{\alpha.N}{4V}}$

Allocation statique de tâches identiques

On a N tâches indépendantes de coûts identiques à répartir sur M unités

Le *principe* consiste à :

- Évaluer les *vitesses respectives* des unités : V_i op/s
- Déduire la *puissance totale* du système : $P = \sum_{j=0}^{M-1} V_j$
- Les *puissances relatives* des unités : $P_i = \frac{V_i}{P}$
- Et la *charge* C_i (nb de tâches) à allouer à chaque unité selon son P_i :

$\Rightarrow$ $\boxed{C_i = P_i.N}$

```
P ← somme(V, 0, M-1)   // Somme des valeurs de V
pour i de 0 à M-1 faire // Calcul des charges de chaque unité
  C[i] ← floor(V[i] * N / P)
fpour
tant que somme(C, 0, M-1) < N faire // Distribution des tâches restantes
  k ← argmin_j((C[j] + 1) / V[j])   // Similaire à l'algo de liste
  C[k] ← C[k] + 1
ftant
```

$\Rightarrow$ Algorithme actualisant la distribution optimale lorsque N augmente

Équilibrage dynamique par file d'attente

Utilisation d'une *file d'attente* de type *FIFO* :

- Pour stocker les tâches à effectuer
- Pour stocker les données à traiter

Principe :

- Les unités se servent dans la file pour récupérer un/e travail/donnée
- À la fin d'un traitement, de nouvelles tâches ou données sont éventuellement ajoutées à la file

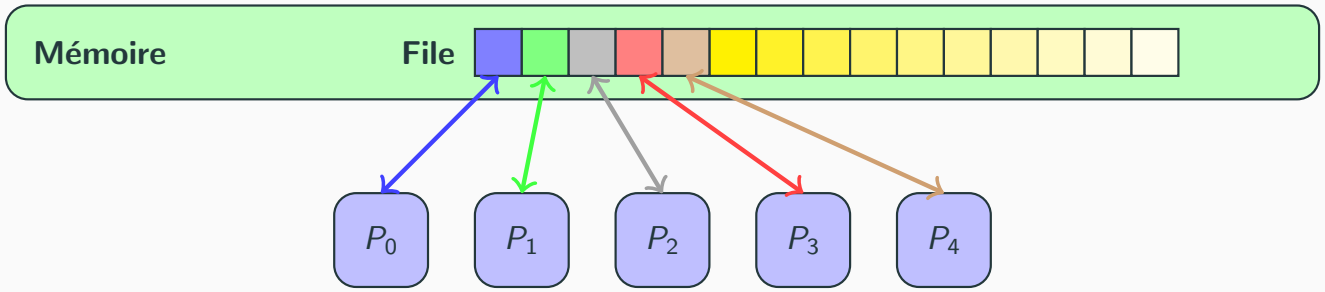
Terminaison :

- La file doit être vide et toutes les unités inactives

$\Rightarrow$ Le système de file réalise un *équilibrage implicite* entre les unités

File en mémoire partagée

Principe relativement simple et efficace



Mais :

- Toutes les unités accèdent à la *même* file ! !

⇒ Il faut assurer la *cohérence des accès* :

- *Exclusion mutuelle* nécessaire pour chaque accès à la file

→ lectures, retraits, insertions doivent être dans une *section critique*

Les *lectures* sont nécessaires pour :

- Tester si la file contient quelque chose
- Récupérer une tâche/donnée à traiter

Les *modifications* sont utilisées pour :

- Le retrait d'une tâche/donnée
- L'ajout éventuel d'une nouvelle tâche/donnée

Algorithme de gestion de la file

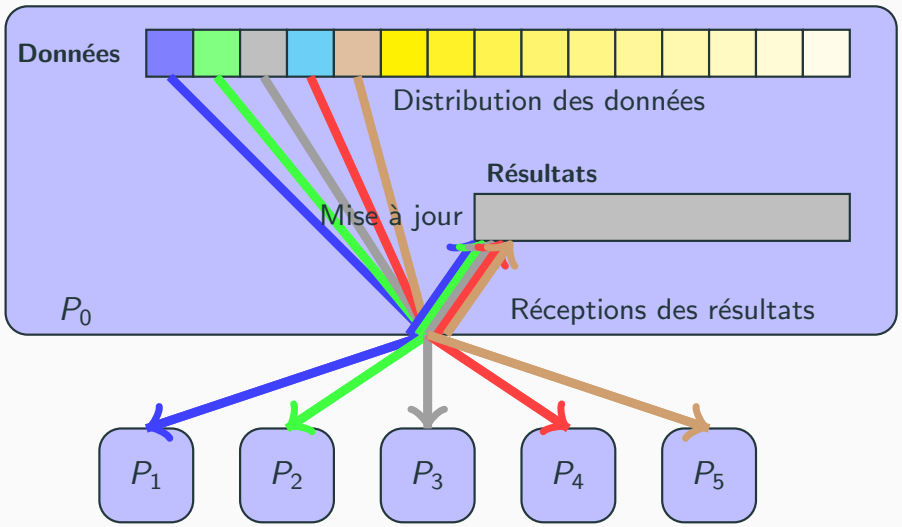
```
pour p de 0 à n-1 faire en parallèle
  si p = 0 alors // Un seul processus initialise les données
    file ← ConstructionFile(listeTaches)
    initVerrou(v)
    nbActifs ← 0
  fsi
  barrière()    // Empêche les autres processus d'accéder aux données avant leur initialisation
  fini_p ← Faux // Indique si le processus p doit arrêter la boucle de travail
  tache_p ← Nil                // Tâche (ou donnée) à traiter
  tant que !fini_p faire       // Boucle de travail du processus p
    vérouiller(v)              // Entrée en section critique pour l'accès à la file
    si ¬ estVide(file) alors   // S'il reste au moins une tâche...
      tache_p ← défiler(file)  // ...on la récupère
      nbActifs ← nbActifs + 1 // ...et on comptabilise le processus dans les processus actifs
    sinon                      // Sinon...
      si nbActifs = 0 alors    // ...on vérifie qu'il ne reste plus de travail en cours
        fini_p ← Vrai          // ...et dans ce cas, on arrête le processus p
      fsi
    fsi
    dévérouiller(v)            // Sortie de la section critique
    si tache_p ≠ Nil alors     // Si on a récupéré une tâche
      exécuter(tache_p)        // ...on l'exécute
      vérouiller(v)
      nbActifs ← nbActifs - 1 // ...et on enlève le processus p des processus actifs
      dévérouiller(v)
    fsi
  ftant
fpour
```

File en mémoire distribuée

Principe du *maître/travailleurs* :

- Une machine *maître* gère la file d'attente
- Les autres machines :
  - Lui demandent des travaux
  - Lui renvoient les résultats

⇒ *Schéma centralisé*



Avantages :

- Très *simple* à mettre en œuvre
- Accès *unique* à la file

Inconvénients :

- *Goulot d'étranglement* au niveau du maître :
  - Limite le nombre de travailleurs
  - Limite le grain des tâches à distribuer

Évaluation des performances dans un cas simple

Modèle :

- n tâches similaires à traiter avec P unités identiques
- t_t = temps de traitement d'une donnée
- t_c = temps d'une communication (1 aller ou 1 retour)

Accélération et efficacité selon P si le maître *distribue seulement* :

- $T_P(n) \approx \frac{n.(t_t+2t_c)}{P-1} \Rightarrow S_P(n) \approx \frac{(P-1).t_t}{t_t+2t_c}$ et $E_P(n) \approx \frac{(P-1).t_t}{P.(t_t+2t_c)}$

Accélération et efficacité selon P si le maître *traite aussi des tâches* :

- Pas de communications pour le maître :
  $\Rightarrow$ temps t_t pour chaque tâche traitée $\Rightarrow$ vitesse $V_0 = \frac{1}{t_t}$
- Les $P-1$ autres unités ont des communications :
  $\Rightarrow$ temps $t_t + 2t_c$ par tâche $\Rightarrow$ vitesse $V_{i>0} = \frac{1}{t_t+2t_c}$

$\Rightarrow$ Contexte similaire à l'équilibrage statique de charges :

$\Rightarrow$ $C_0 = \frac{n.(t_t+2t_c)}{P.t_t+2t_c}$ et $C_{i>0} = \frac{n.t_t}{P.t_t+2t_c}$

- $T_P(n) \approx \frac{n.t_t.(t_t+2t_c)}{P.t_t+2t_c} \Rightarrow S_P(n) \approx \frac{P.t_t+2t_c}{t_t+2t_c}$ et $E_P(n) \approx \frac{P.t_t+2t_c}{P.(t_t+2t_c)}$

Évaluation des performances dans un cas simple

On voit donc qu'il faut que $t_c << t_t$ *pour avoir de bonnes performances* !

En pratique, il faut également éviter les phénomènes de *famine* :

- Le maître doit servir *toutes* les unités *avant* une nouvelle demande
- Le temps de travail d'un tâche doit donc être suffisamment grand

On peut également ajouter de la *tolérance aux pannes* :

- *Redistribuer une tâche* si le résultat ne revient pas assez vite (coupure réseau, panne machine,...)

Et de la *concurrence* si les machines ont des vitesses différentes :

- Distribuer une *même tâche à plusieurs unités*

$\Rightarrow$ On récupère le résultat de l'unité la plus rapide

$\Rightarrow$ On annule le travail des autres unités (envois d'autres tâches)

Équilibrage dynamique par redistributions

Les tâches sont *régulièrement redistribuées* entre les unités :

- Évite la file centralisée
- Prend en compte l'évolution du système pendant l'exécution

Principe :

- Distribution initiale de la charge (statique)
- Entre les grandes étapes ou à intervalles de temps réguliers :
  - Évaluer les charges/performances de chaque unité
  - Calculer une nouvelle distribution équilibrée
  - Redistribuer les données/tâches globalement

⚠ *!! Redistributions globales coûteuses en mémoire distribuée !!* ⚠

Nécessité d'un réglage approprié de la *fréquence des redistributions* :

- Si *trop fréquentes* : le surcoût des redistributions annule le gain d'équilibrage (voir même pire...)
- Si *pas assez fréquentes* : risque de déséquilibre prolongé

Partage local de charge

Les machines sont organisées selon un *graphe logique d'inter-connexion* :

- Une unité qui n'a plus (assez) de travail, en demande à ses voisins

⇒ La *charge locale* d'une unité doit être *divisible*
→ Choix du grain de parallélisme, plusieurs tâches par unité

Avantages :

- Transferts de charge uniquement entre voisins

⇒ On privilégie les *communications locales* (plus rapides)

Règles à suivre pour que ce mécanisme soit efficace :

- Éviter les *va-et-vient* (ou *ping-pong*)
- Éviter les *famines* en déchargeant trop les unités qui donnent
- Éviter les transferts trop nombreux
→ Surcharge réseau, ralentissement des calculs

Programmation des GPU

Architecture des GPU

Un GPU est un *co-processeur de calcul* comprenant :

- Un ensemble de *multi-processeurs (SM)* de type SIMD avec chacun :
  - Un ensemble d'unités arithmétiques (ALU)
  - Un décodeur d'instructions
  - Un ensemble de registres partagés
  - Trois mémoires internes : partagée, constantes et textures
- Partageant une mémoire globale :
  - Point de passage pour les transferts de données entre CPU et GPU

Comparaison :

- Ryzen 9 5950X : $\approx 4,15$ Gt, 16 Coeurs, 105W
- GPU Oberon Plus (PS5) : $\approx 10,6$ Gt, ≈2500 unités, 225W

Plusieurs langages de programmation permettent d'utiliser les GPU :

- CUDA (Nvidia), OpenCL, OpenACC, Compute Shaders,...
- Imposent une *vision logique* du GPU

Organisation logique de CUDA (Nvidia)

CUDA voit les GPU comme :

- Une *grille 3D de blocs* contenant chacun :
  - Une *grille 3D de threads* avec chacun :
    - Des registres
    - Une mémoire locale
  - Une *mémoire partagée* entre les threads
    avec des temps d'*accès rapides*
  - Les threads peuvent aussi accéder aux *mémoires supérieures*
    (globale (Go), constantes (Ko), texture (Go))
    mais avec des temps d'*accès plus lents*
- Des transferts possibles de la *RAM vers les trois mémoires GPU*
  mais avec des temps d'*accès très lents*

$\Rightarrow$ Une des *difficultés principales* vient de la *bonne gestion des mémoires*

Principe de fonctionnement

Les GPU exécutent des *kernels* :

- Fonctions écrites en CUDA (fichiers `.cu`)
- Lancés par le CPU selon *deux modes possibles* :
  - Synchrone ou asynchrone (par défaut)

Schéma simplifié :

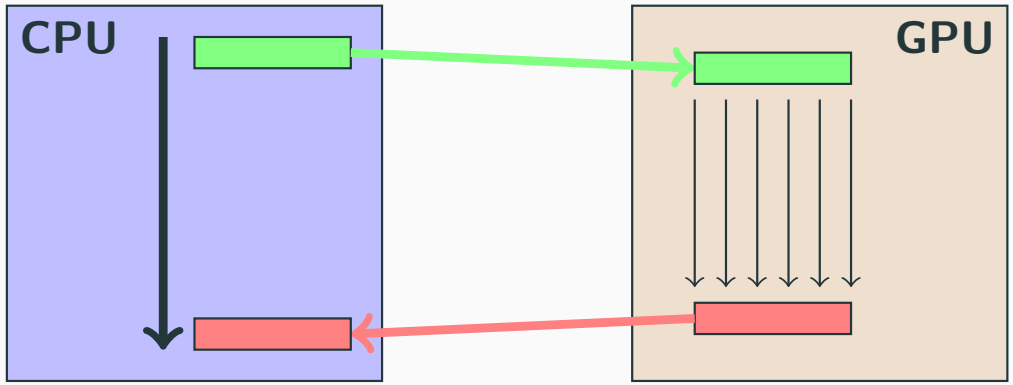


Schéma réel :

- Chaque bloc est exécuté sur un seul SM
- L'ordonnanceur de blocs répartit les blocs sur les SM

$\Rightarrow$ Plus le GPU a de SM, plus il traite l'ensemble des blocs rapidement
→ Problème classique d'ordonnancement de n tâches sur P unités !

Définition de kernels et utilisation

Exemple d'addition de deux matrices carrées de tailles $N \times N$ dans une troisième

```
// Définition du Kernel à exécuter sur le GPU
__global__ void MatAdd(int N, float *A, float *B, float *C)
// Le préfixe __global__ indique un kernel GPU
{
  int i = threadIdx.x; // Récupération de l'indice x du thread courant dans le bloc
  int j = threadIdx.y; // Récupération de l'indice y du thread courant dans le bloc
  C[i*N+j] = A[i*N+j] + B[i*N+j]; // Calcul d'un élément de la matrice C (mémoire globale)
}
```

```
// Définition du programme exécuté sur le CPU
int main()
{
  int N;                    // Taille des matrices à traiter (divisible par 16)
  float *Acpu, *Agpu, ...; // Pointeurs sur les matrices pour le CPU et le GPU
  dim3 Db(16, 16, 1);      // Description des dimensions des blocs (2D avec 256 threads)
  dim3 Dg(N/16, N/16, 1); // Description des dimensions de la grille (2D avec N/16 lignes et colonnes)

  // Allocation de A sur le GPU et transferts des données de Acpu dans Agpu vers le GPU
  cudaMalloc((void**) &Agpu, N * N * sizeof(float));
  cudaMemcpy((void*) Agpu, Acpu, N * N * sizeof(float), cudaMemcpyHostToDevice);
  ...
  MatAdd<<<Dg, Db>>>(N, Agpu, Bgpu, Cgpu); // Lancement du kernel depuis le CPU selon la gille
                                           // et les blocs spécifiés par <<<Dg, Db>>>
  ... // Calculs éventuels sur le CPU en attendant l'exécution du GPU
  // Récupération des données de Cgpu dans Ccpu depuis le GPU
  cudaMemcpy((void*) Ccpu, Cgpu, N * N * sizeof(float), cudaMemcpyDeviceToHost);
  ... // Suite du programme
}
```

Organisation logique de OpenACC

OpenACC suit une logique proche de celle de OpenMP :

- Identification de régions à exécuter en parallèle (sur CPU ou GPU)
- Utilisation de *directives*

Mais OpenACC a son propre *modèle d'exécution* :

- *Un thread hôte* qui gère le parallélisme
- Envoi des calculs sur un *device* particulier (coeurs CPU ou GPU)
- Une décomposition en *trois niveaux de parallélisme* :
  - Groupes (`gang`) : grandes tâches (gros grain)
  - Travailleurs (`workers`) : tâches moyennes (grain moyen)
  - Vecteurs (`vector`) : petites tâches (grain fin = SIMD)
- Ces trois niveaux peuvent être activés ensemble ou non
- *Pas* de barrières ou sections critiques entre gangs, workers ou vectors

Et son *modèle mémoire* :

- *Transferts explicites* de données entre hôte et GPU via des directives

Exemple d'utilisation d'OpenACC

Exemple d'addition de deux matrices carrées de tailles $N \times N$ dans une troisième

```
#define N 1000
float A[N][N], B[N][N], C[N][N];

int main()
{
  int i, j;

  // Initialisation des matrices A et B
  ...

  #pragma acc data copyin(A,B) copy(C) // Transfert des matrices A et B vers le GPU
                                       // et récupération du résultat C depuis le GPU
  #pragma acc kernels                  // Transforme le code suivant en kernel GPU
  #pragma acc loop tile(16,16)         // Parallélisation de la boucle par blocs de taille 16x16
  for(i=0; i<N; ++i){
    for(j=0; j<N; ++j){
      C[i][j] = A[i][j] + B[i][j];
    }
  }
}
```

Bilan sur la programmation GPU

Petit aperçu avec deux langages très utilisés :

- CUDA : langage *proche du matériel*, spécifique aux cartes NVIDIA
- OpenACC : langage *plus général* (pas que GPU), basé sur directives

Points forts :

- Accès à une *puissance de calcul* très importante
- Décharge le CPU pour faire d'autres calculs entre temps

Points faibles :

- Contraintes matérielles importantes
- *Difficulté de programmation* pour obtenir un code efficace
- *Temps de transferts* entre hôte et GPU

Conclusion

Conclusion

Le parallélisme permet :

- Un *gain de temps* d'exécution
- Le traitement de *problèmes de grandes tailles*

Il y a plusieurs types de parallélismes :

- Données, tâches, flux
- Souvent une *composition hiérarchique* de ces différents types

Les principaux obstacles sont :

- Les *dépendances* entre calculs
- La *concurrence des accès* en mémoire partagée
- Les surcoûts dûs aux *communications* en mémoire distribuée
- La gestion des différents niveaux de parallélisme imbriqués

L'exploitation efficace des systèmes de calcul reste un vrai **challenge** !