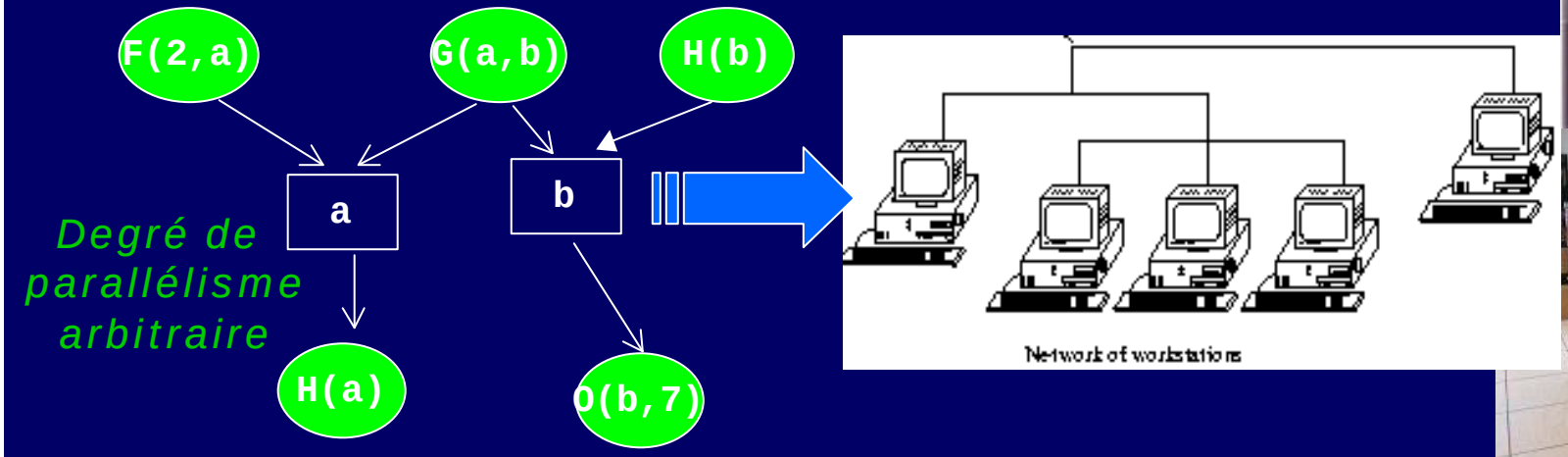
# Motivation (1/2)

## *Parallélisme et granularité*




Granularité « pratique » : gros grain

Découpe en p = #ressources disponibles

*Inconvénient : architecture hétérogène, dynamique*

Granularité « théorique » : grain fin

Parallélisme maximal avec #ops ~ tps séquentiel

*Inconvénient : surcoût d'ordonnancement sur l'architecture*

*Quel grain pour limiter le surcoût de la parallélisation ?*

# Motivation (2/2) *Algorithmique*

- Exemple : calcul des préfixes $P_i = *_{k=0}^{i} a_i \quad (i=0..n)$
  - Algorithme séquentiel :

```
for (i= 0 ; i <= n;  i++ )  P[ i ] = P[ i – 1 ] * a [ i ] ;
```

#ops * = n

  - Algorithme parallèle :

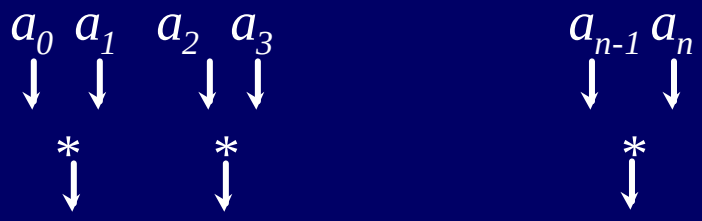
# Motivation (2/2) *Algorithmique*

- Exemple : calcul des préfixes $P_i = *_{k=0}^{i} a_i \quad (i=0..n)$
  - Algorithme séquentiel :

```
for (i= 0 ; i <= n;  i++ )  P[ i ] = P[ i – 1 ] * a [ i ] ;
```

#ops * = n

  - Algorithme parallèle :



# Motivation (2/2) *Algorithmique*

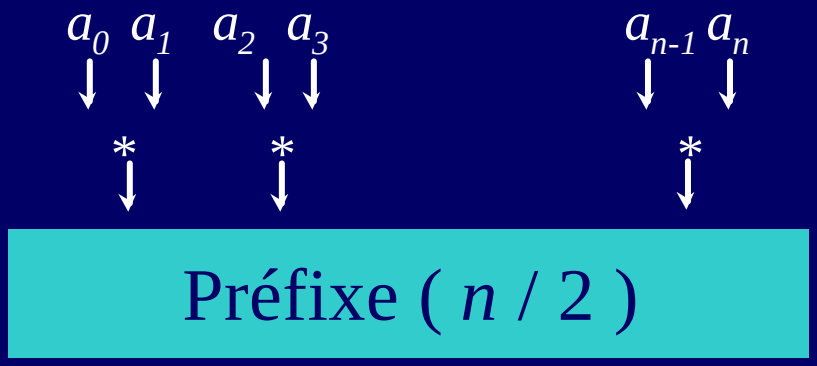
- Exemple : calcul des préfixes $P_i = *_{k=0}^{i} a_i \quad (i=0..n)$

  - Algorithme séquentiel :

```
for (i= 0 ; i <= n;  i++ )  P[ i ] = P[ i – 1 ] * a [ i ] ;
```

#ops * = n

  - Algorithme parallèle :



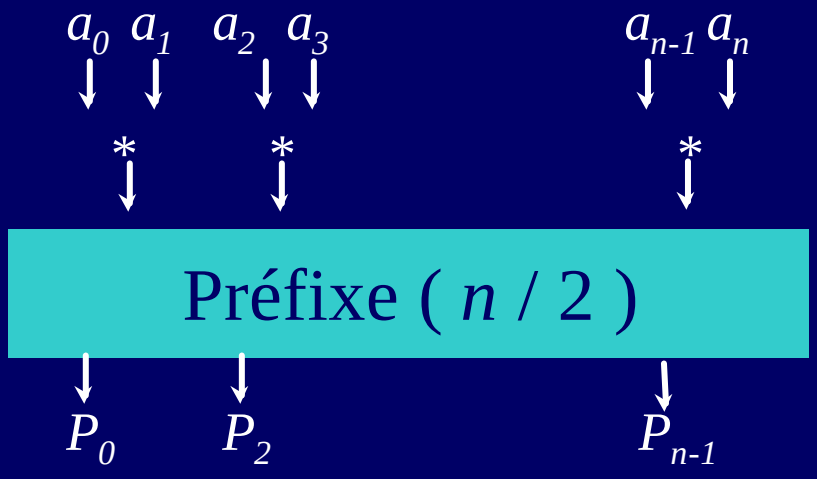
# Motivation (2/2) *Algorithmique*

- Exemple : calcul des préfixes $P_i = *_{k=0}^{i} a_i \quad (i=0..n)$

  - Algorithme séquentiel :

    for (i= 0 ; i <= n; i++ ) P[ i ] = P[ i – 1 ] * a [ i ] ; #ops * = n

  - Algorithme parallèle :



# Motivation (2/2) *Algorithmique*

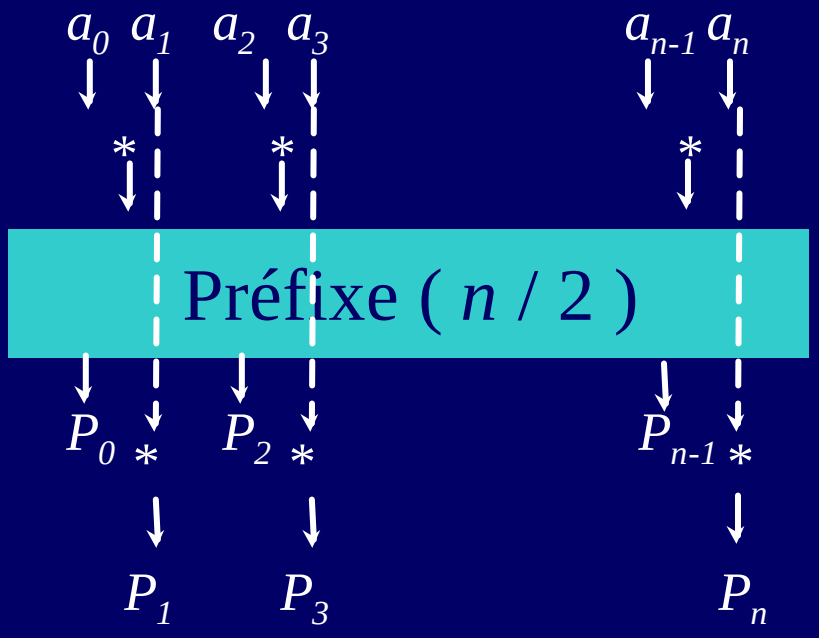
- Exemple : calcul des préfixes $P_i = *_{k=0}^{i} a_i \quad (i=0..n)$

  - Algorithme séquentiel :

```
for (i= 0 ; i <= n;  i++ )  P[ i ] = P[ i – 1 ] * a [ i ] ;
```

#ops * = n

  - Algorithme parallèle :



# Motivation (2/2) *Algorithmique*

- Exemple : calcul des préfixes $P_i = *_{k=0}^{i} a_i$ $(i=0..n)$
  - Algorithme séquentiel :

```
for (i= 0 ; i <= n;  i++ )  P[ i ] = P[ i – 1 ] * a [ i ] ;
```

#ops * = n

  - Algorithme parallèle :

```
dopar (i= 0 ; i <= n;  i+=2 )  b[ i ] = a[ 2.i ] * a [ 2.i+1 ] ;
PrefixePar( b[ 0 .. n/2], P[1, 3, 5, .., n-1] ) ;
dopar (i= 1 ; i <= n/2;  i++ )  P[ 2.i ] = P[2i-1] * a [ 2.i ] ;
```

=> $\#T_\infty$ = tps parallèle = 2. log n mais #ops * = 2.n

  - Parallèle => surcoût en nbre d'opérations

[Ladner-Fischer]

# Plan de la présentation

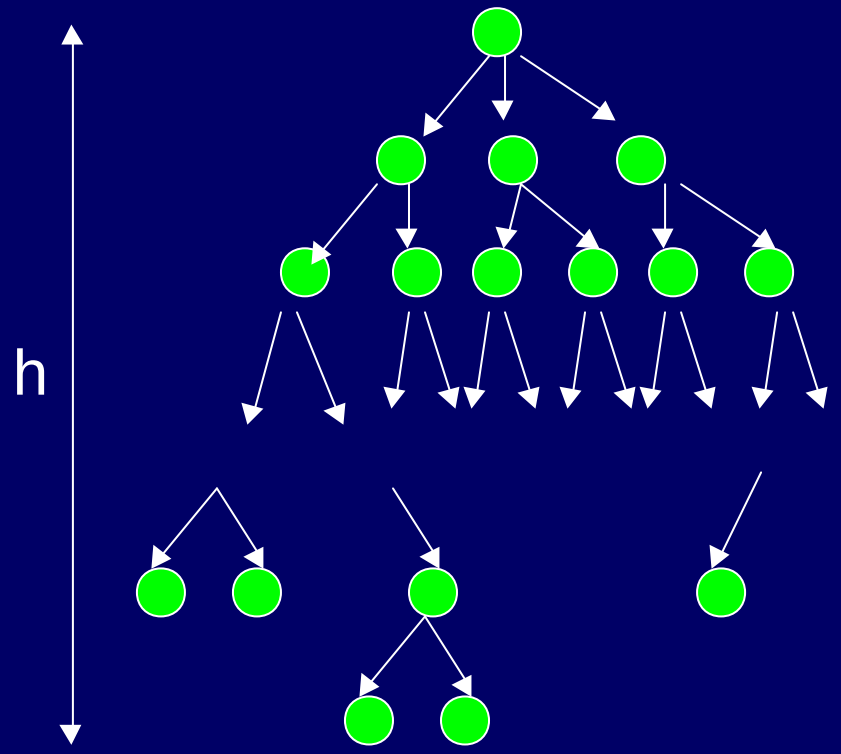
- 1. Découpe récursive et ordonnancement work-stealing
- 2. Parallélisation à grain adaptatif:
  - couplage algorithme séquentiel et algorithme parallèle
- 3. Application à la parallélisation de gzip

# 1. Parallélisme par découpe récursive

*exemple: Produit itéré, arborescence critique Branch&Bound, ...*

```
void ProdIter ( node  x, ...) {
  if (grain(x) < ...)  eval_seq(x)
  else {
     for    s ∈ fils(x)   dopar
          compute( s, ...) ;
} }
```

*task*




Notations :

$T_s$ = temps séquentiel

$T_1$ = travail parallèle

$T_¥$ = temps parallèle

Sur p processeurs [Brent]

$T_{exec(p)} \sim T_1 / p + T_\infty$ + **surcoût //**

# Implantation : ordo. par vol de travail (1/2)

- Ordonnancement par vol de travail [Graham] :
  - Graphe de tâches avec parallélisme ***dynamique***
  - Nombre de vols < p. $T_\infty$ => $T_p = T_{1/p} + T_\infty + O(p.T_\infty)$
  - Intérêt de la découpe récursive si $T_\infty$ petit

- Optimisation à l'exécution :

  ***work-first principle*** [Multilisp, Cilk, …]

  $\Rightarrow$ dégénération séquentielle de l'***algorithme parallèle*** pour éviter le surcoût de gestion de tâches parallèles inutiles

# Implantation : ordo. par vol de travail (2/2)

Hypothèse : ordo. séquentiel de l'algorithme parallèle valide

+ Exécution non-préemptive d'une tâche prête

Pile

```
f1() { ….
   fork 2 ; …
}
```

# Implantation : ordo. par vol de travail (2/2)

Hypothèse : ordo. séquentiel de l'algorithme parallèle valide

+ Exécution non-préemptive d'une tâche prête

```
f1() { ….
   fork 2 ; …
}
```


P

# Implantation : ordo. par vol de travail (2/2)

Hypothèse : ordo. séquentiel de l'algorithme parallèle valide

+ Exécution non-préemptive d'une tâche prête

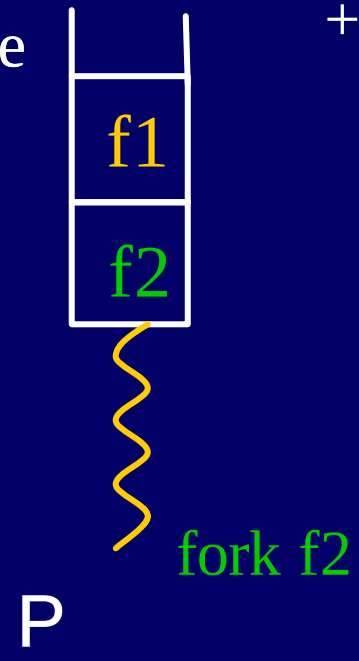
```
f1() { ….
   fork 2 ; …
}
```


# Implantation : ordo. par vol de travail (2/2)

Hypothèse : ordo. séquentiel de l'algorithme parallèle valide

+ Exécution non-préemptive d'une tâche prête

```
f1() { ….
   fork 2 ; …
}
```

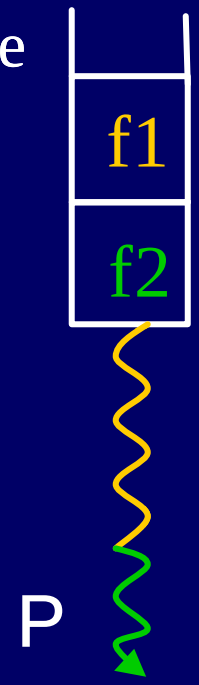


# Implantation : ordo. par vol de travail (2/2)

Hypothèse : ordo. séquentiel de l'algorithme parallèle valide

+ Exécution non-préemptive d'une tâche prête

```
f1() { ….
   fork 2 ; …
}
```

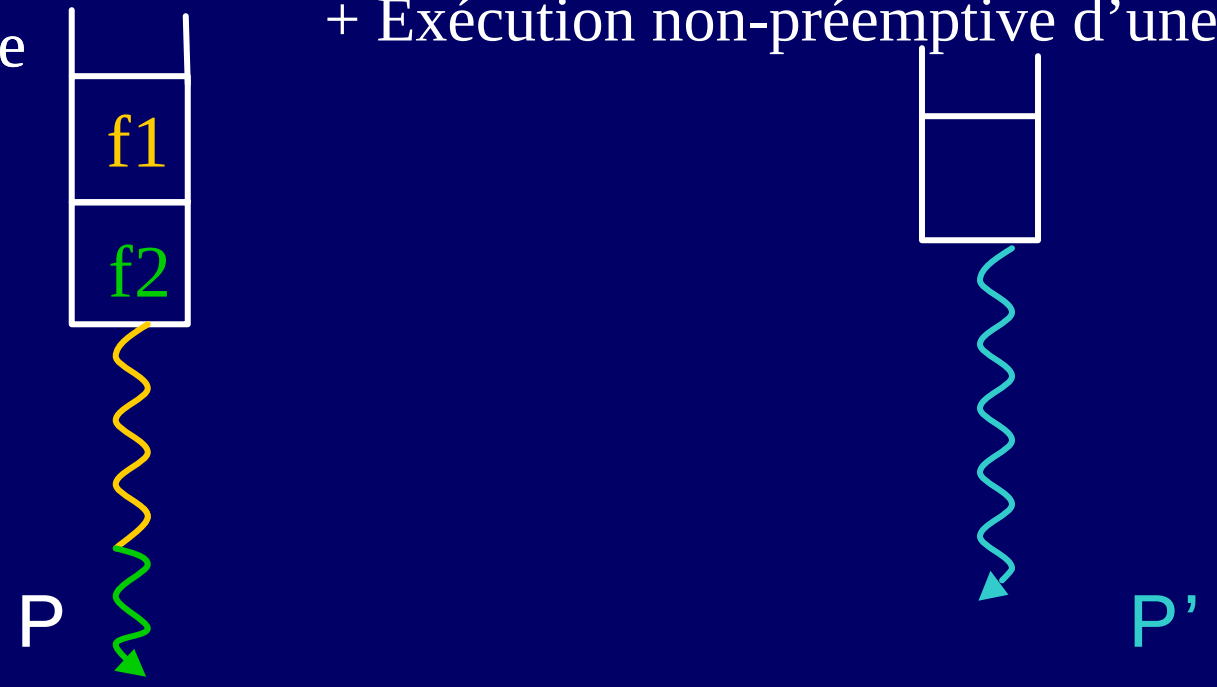


# Implantation : ordo. par vol de travail (2/2)

Hypothèse : ordo. séquentiel de l'algorithme parallèle valide

+ Exécution non-préemptive d'une tâche prête

```
f1() { ….
   fork 2 ; …
}
```

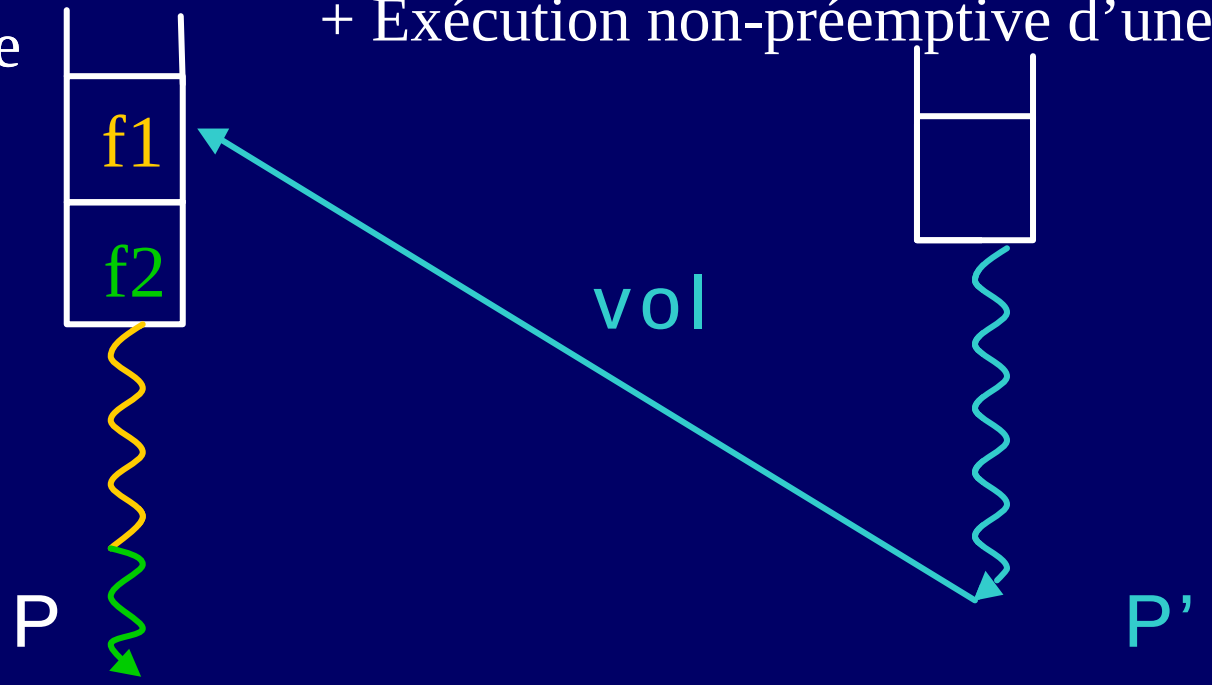


# Implantation : ordo. par vol de travail (2/2)

Hypothèse : ordo. séquentiel de l'algorithme parallèle valide

+ Exécution non-préemptive d'une tâche prête

```
f1() { ….
   fork 2 ; …
}
```

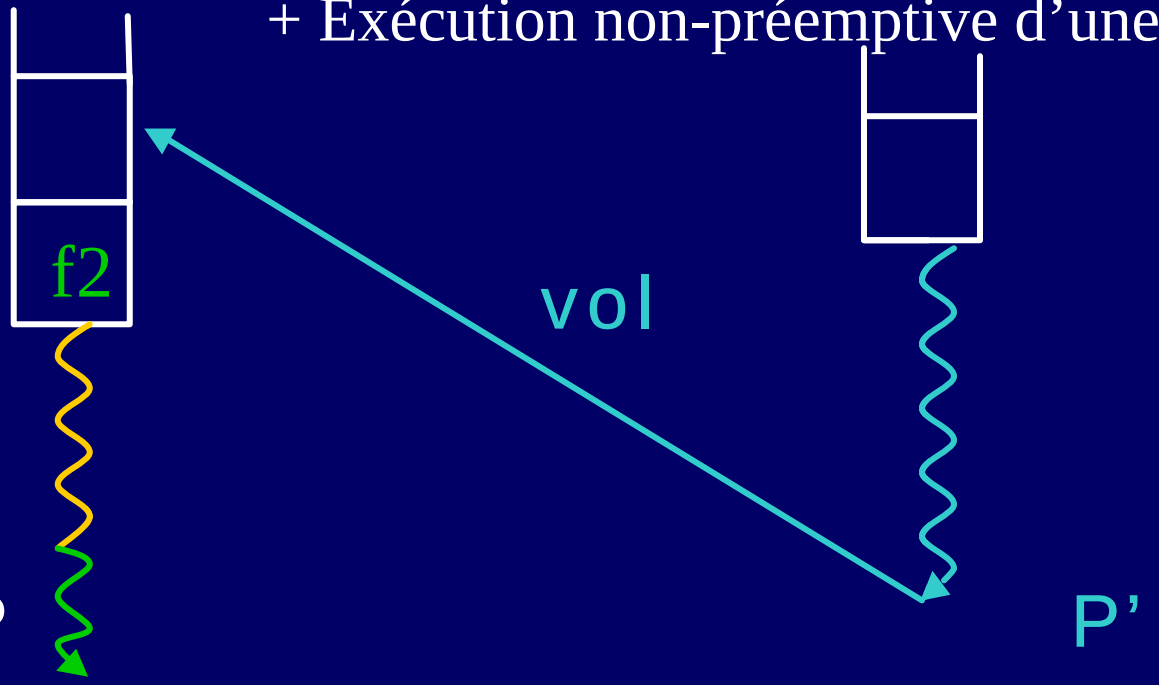


# Implantation : ordo. par vol de travail (2/2)

Hypothèse : ordo. séquentiel de l'algorithme parallèle valide

+ Exécution non-préemptive d'une tâche prête

```
f1() { ….
   fork 2 ; …
}
```

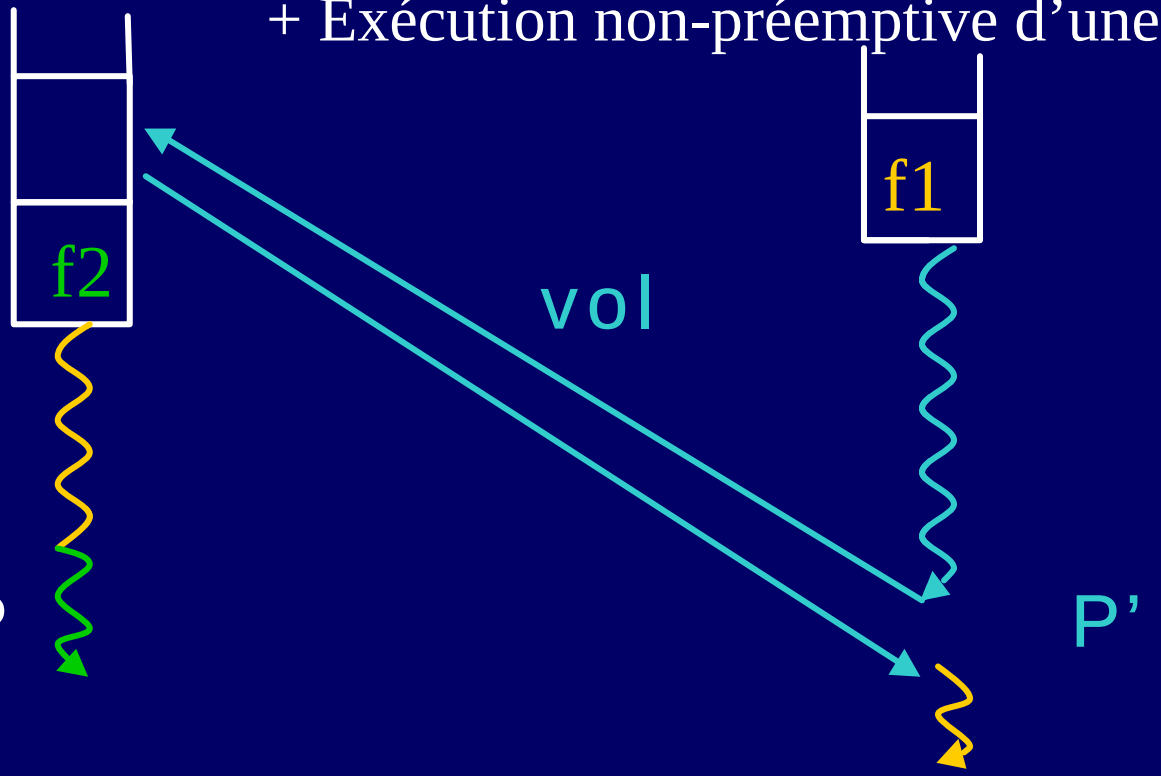


# Implantation : ordo. par vol de travail (2/2)

Hypothèse : ordo. séquentiel de l'algorithme parallèle valide

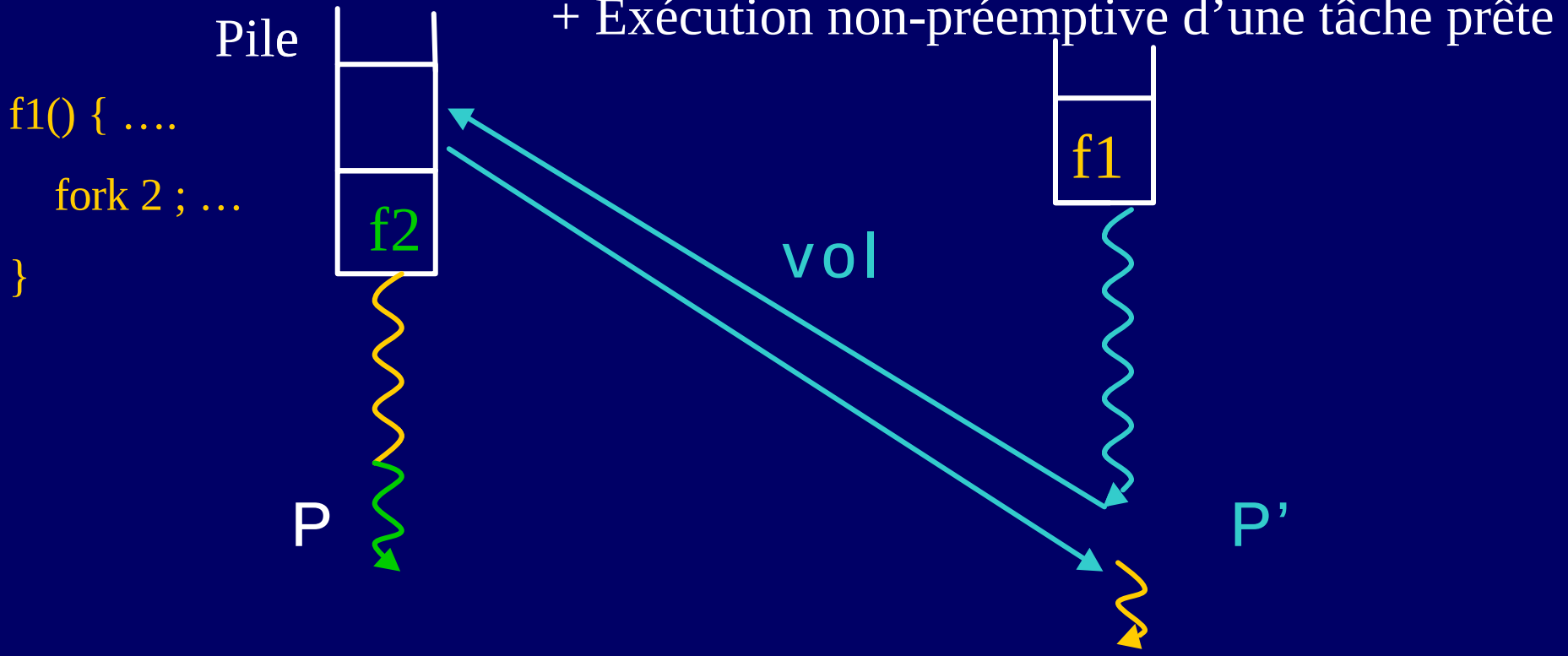
+ Exécution non-préemptive d'une tâche prête

```
f1() { ….
   fork 2 ; …
}
```



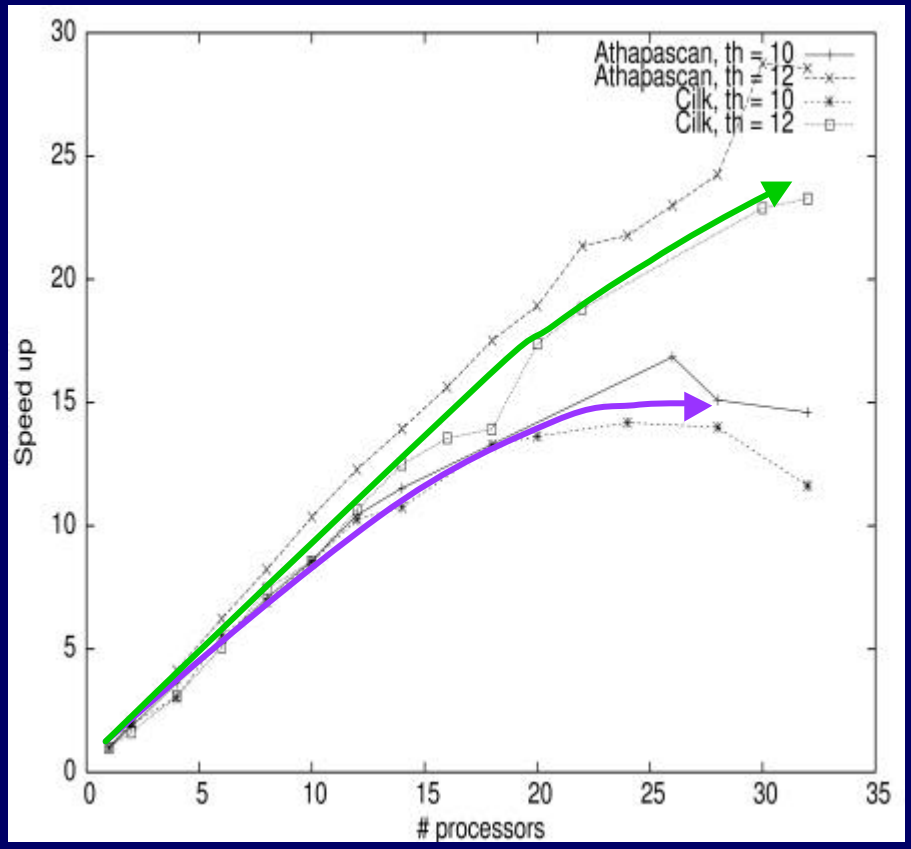
# Implantation : ordo. par vol de travail (2/2)

Hypothèse : ordo. séquentiel de l'algorithme parallèle valide
+ Exécution non-préemptive d'une tâche prête




- Intérêt : Grain fin « statique », mais contrôle dynamique
- Inconvénient: surcôut possible de l'algorithme parallèle [ex. préfixes]

# Expérimentations : knary




| #procs | Speed-Up |
|---|---|
| 8 | 7,83 |
| 16 | 15,6 |
| 32 | 30,9 |
| 64 | 59,2 |
| 100 | 90,1 |

## SMP Origin 3800

**Cilk / Athapascan**

## iCluster

**Athapascan**

$T_s = 2397$ s - $T_1 = 2435$

# 2. Parallélisation à grain adaptatif

- Principe : si un processeur devient inactif,
  => ***extraction de parallélisme*** sur un processeur en train d 'exécuter un ***algorithme séquentiel***

- Hypothèses : deux algorithmes :
  - 1 algo. séquentiel : *SeqCompute*
  - 1 algo. parallèle : *LastPartComputation* => à tout moment, possibilité d'extraire du parallélisme du travail restant à faire dans l'algo. séquentiel.
  - Après extraction : le travail extrait peut être réalisé par l'algo.séquentiel.

# 2. Parallélisation à grain adaptatif

- Principe : si un processeur devient inactif,
  => ***extraction de parallélisme*** sur un processeur en train d 'exécuter un ***algorithme séquentiel***

- Hypothèses : deux algorithmes :
  - 1 algo. séquentiel : *SeqCompute*
  - 1 algo. parallèle : *LastPartComputation* => à tout moment, possibilité d'extraire du parallélisme du travail restant à faire dans l'algo. séquentiel.
  - Après extraction : le travail extrait peut être réalisé par l'algo.séquentiel.

# 2. Parallélisation à grain adaptatif

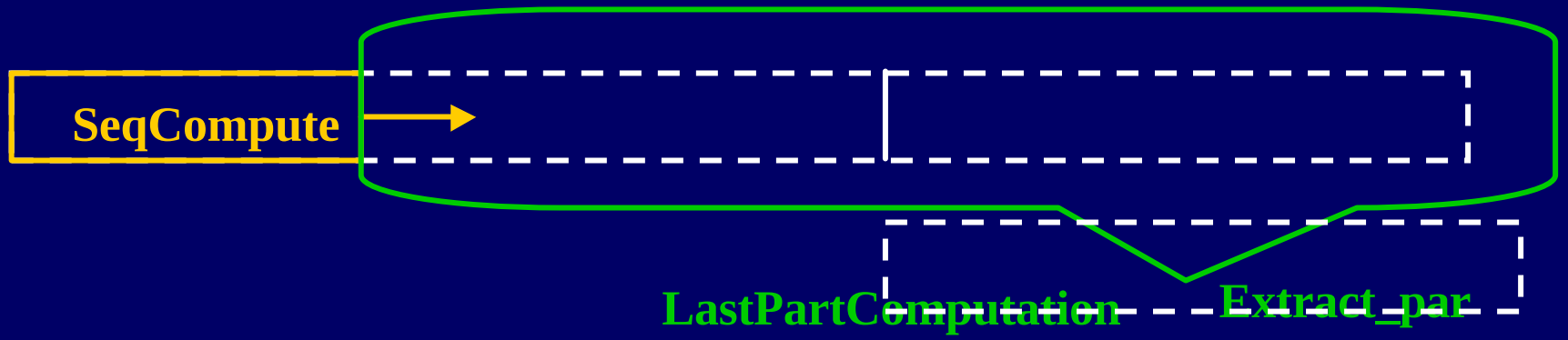
- Principe : si un processeur devient inactif,
  => ***extraction de parallélisme*** sur un processeur en train d 'exécuter un ***algorithme séquentiel***

- Hypothèses : deux algorithmes :
  - 1 algo. séquentiel : *SeqCompute*
  - 1 algo. parallèle : *LastPartComputation* => à tout moment, possibilité d'extraire du parallélisme du travail restant à faire dans l'algo. séquentiel.
  - Après extraction : le travail extrait peut être réalisé par l'algo.séquentiel.

# 2. Parallélisation à grain adaptatif

- Principe : si un processeur devient inactif,
  => ***extraction de parallélisme*** sur un processeur en train d 'exécuter un ***algorithme séquentiel***

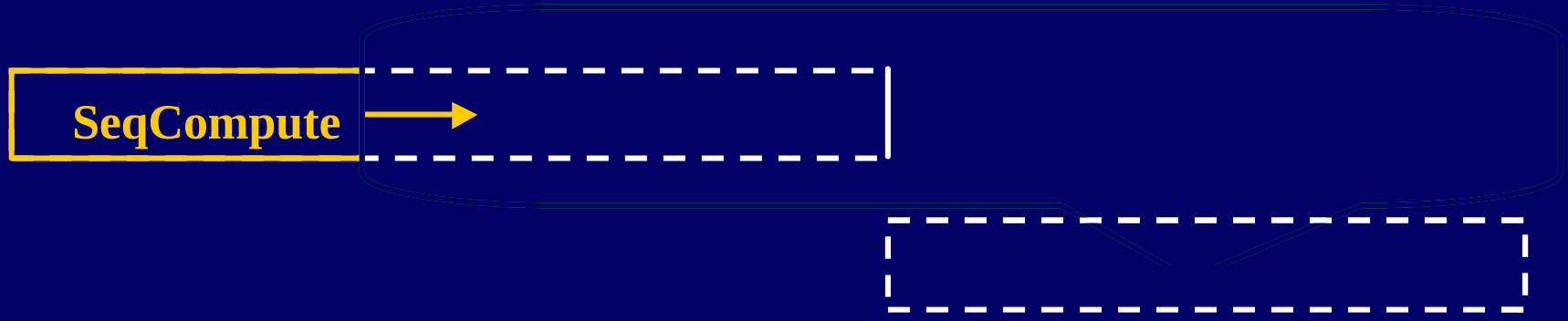
- Hypothèses : deux algorithmes :
  - 1 algo. séquentiel : *SeqCompute*
  - 1 algo. parallèle : *LastPartComputation* => à tout moment, possibilité d'extraire du parallélisme du travail restant à faire dans l'algo. séquentiel.
  - Après extraction : le travail extrait peut être réalisé par l'algo.séquentiel.



# 2. Parallélisation à grain adaptatif

- Principe : si un processeur devient inactif,
  => ***extraction de parallélisme*** sur un processeur en train d 'exécuter un ***algorithme séquentiel***

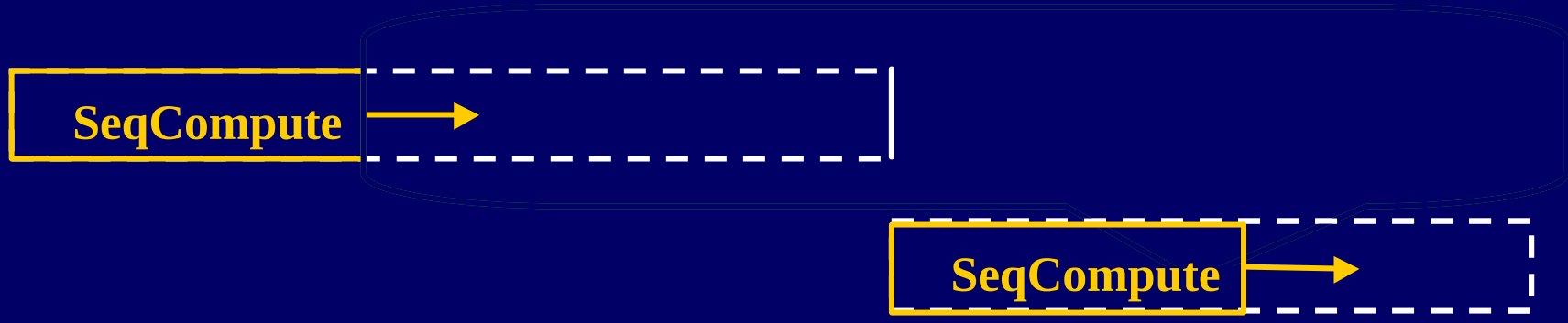
- Hypothèses : deux algorithmes :
  - 1 algo. séquentiel : *SeqCompute*
  - 1 algo. parallèle : *LastPartComputation* => à tout moment, possibilité d'extraire du parallélisme du travail restant à faire dans l'algo. séquentiel.
  - Après extraction : le travail extrait peut être réalisé par l'algo.séquentiel.



# 2. Parallélisation à grain adaptatif

- Principe : si un processeur devient inactif,
  => ***extraction de parallélisme*** sur un processeur en train d 'exécuter un ***algorithme séquentiel***

- Hypothèses : deux algorithmes :
  - 1 algo. séquentiel : *SeqCompute*
  - 1 algo. parallèle : *LastPartComputation* => à tout moment, possibilité d'extraire du parallélisme du travail restant à faire dans l'algo. séquentiel.
  - Après extraction : le travail extrait peut être réalisé par l'algo.séquentiel.

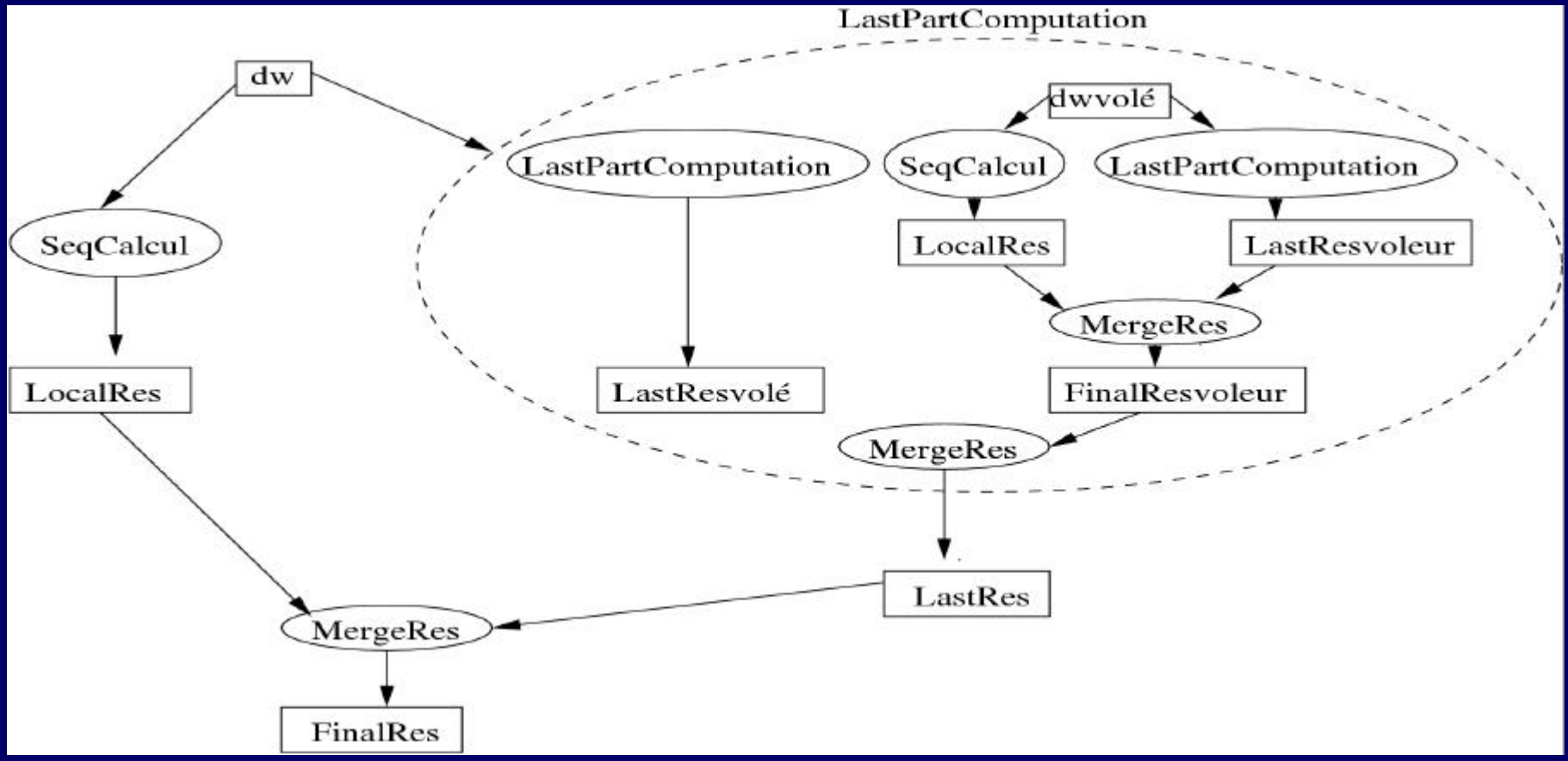


# Algorithme générique à grain adaptatif

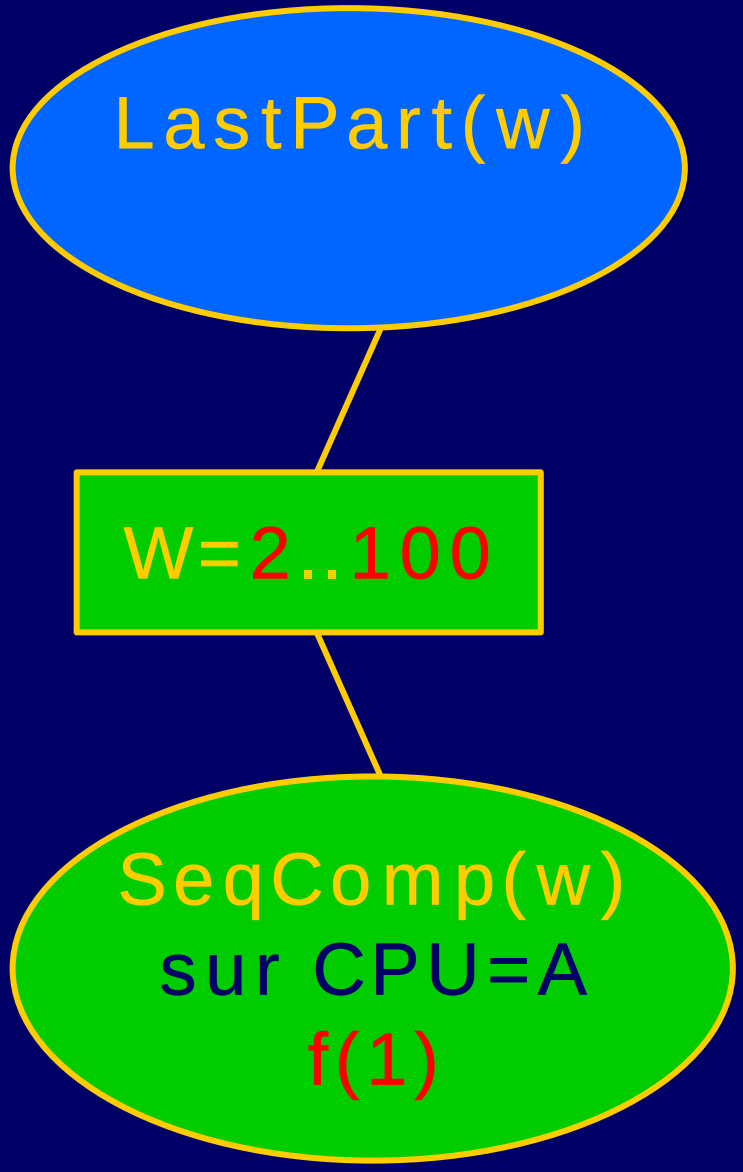
- Programmation : suppose un ordo. des tâches par vol de travail :

- ```
  LastPartComputation( work ) {
        wpar = Extract_par( work ) ;  // work est modifié
        fork SeqCompute ( wpar ) ;
        fork LastPartComputation ( wpar ) ;
        fork LastPartComputation ( work ) ;
  }
  ```

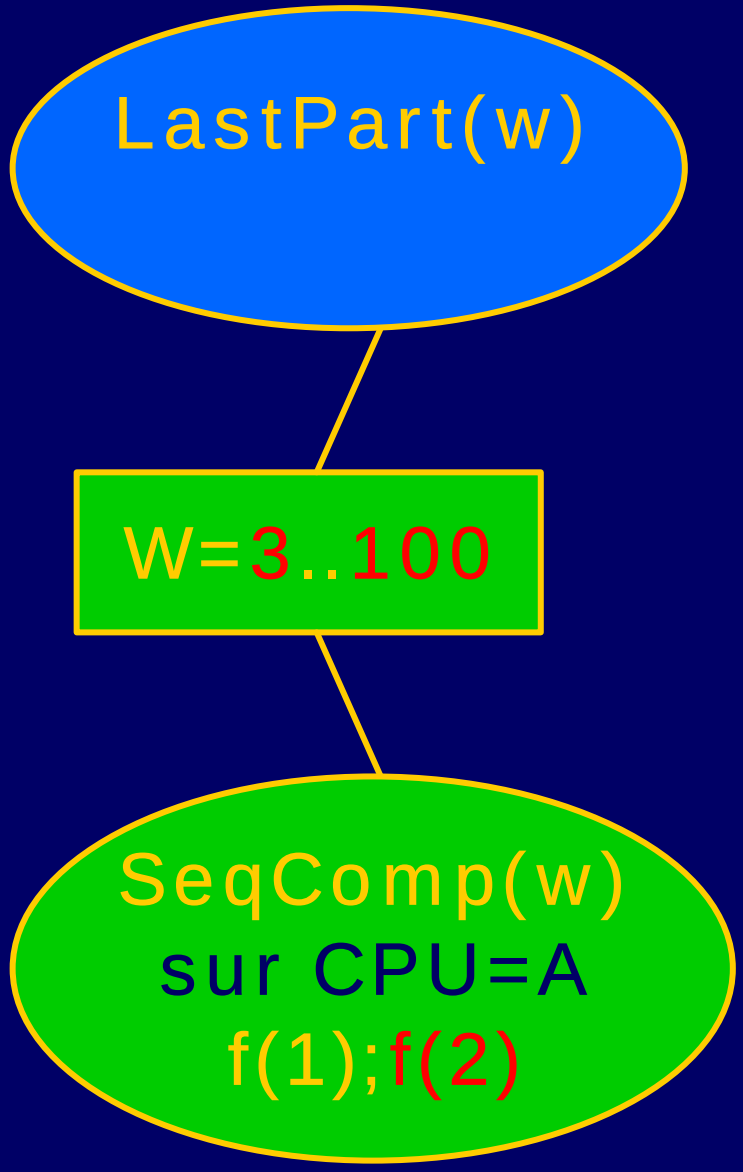
- Sémantique séquentielle :
  - LastPartComputation n'est lancée que si un processeur est inactif
  - Sinon, LastPartComputation est exécutée avec wpar = Vide
  ```


LastPartComputation
dw
dwvolé
LastPartComputation
SeqCalcul
LastPartComputation
SeqCalcul
LocalRes
LastResvoleur
MergeRes
LocalRes
LastResvolé
FinalResvoleur
MergeRes
LastRes
MergeRes
FinalRes

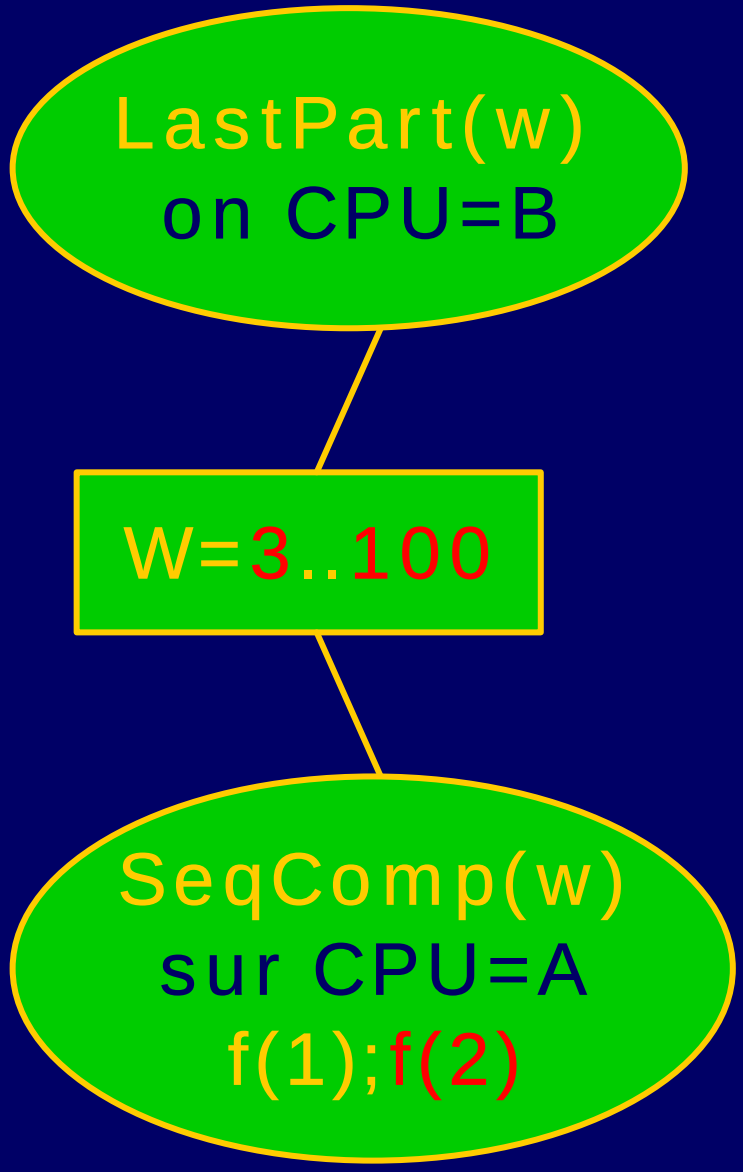
# E.g. Calcul parallèle f(i), i=1..100



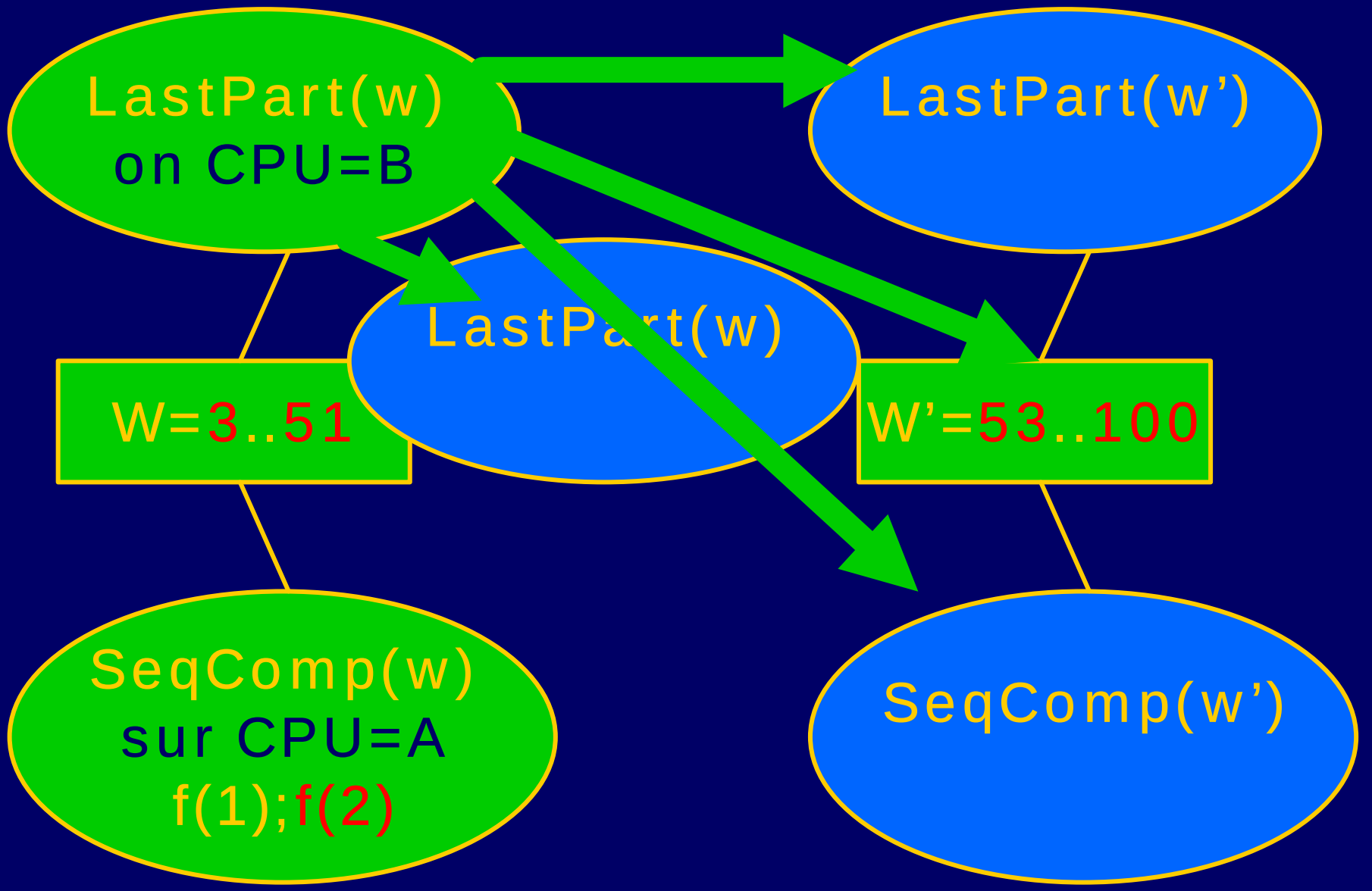
# E.g. Calcul parallèle f(i), i=1..100



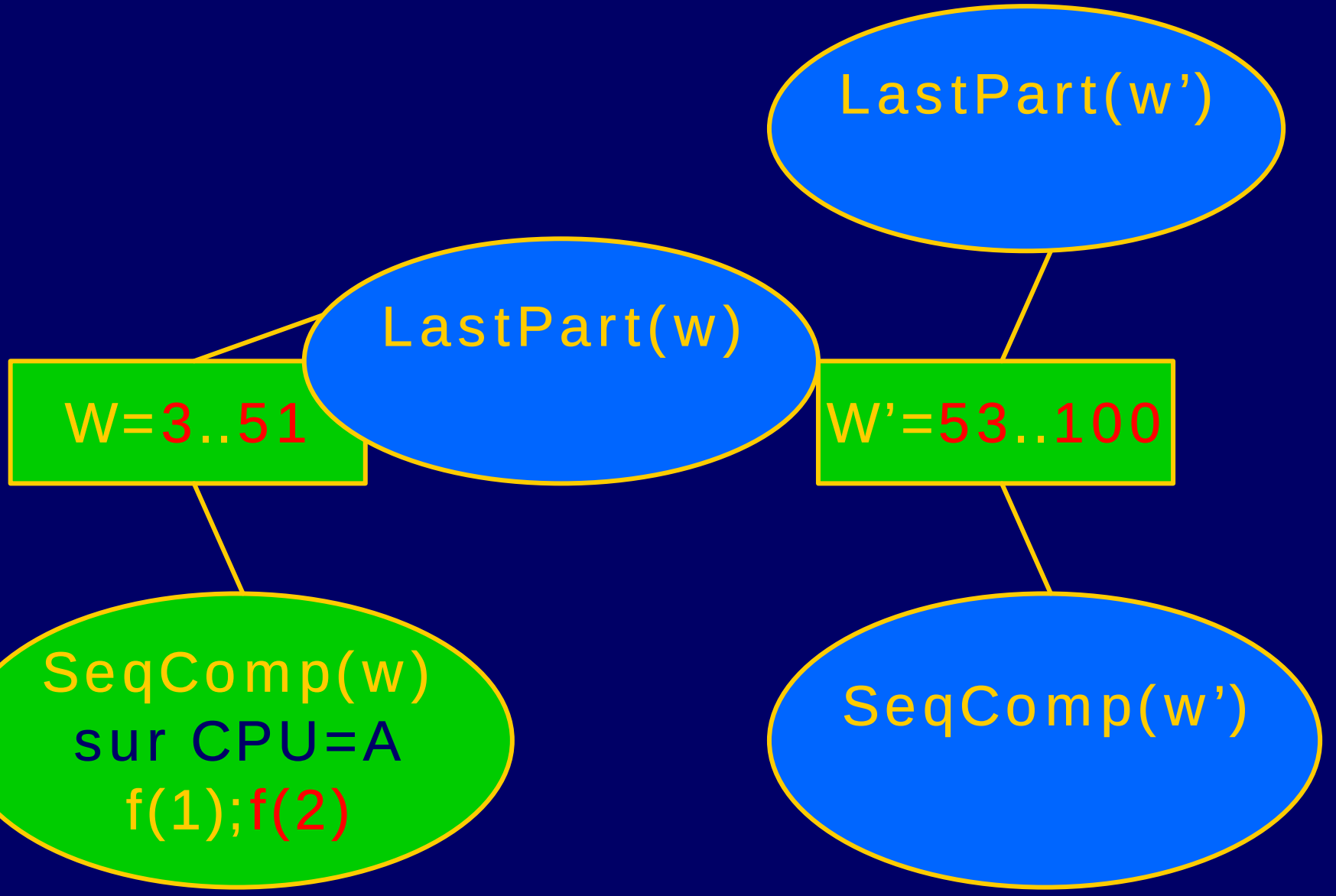
# E.g. Calcul parallèle f(i), i=1..100



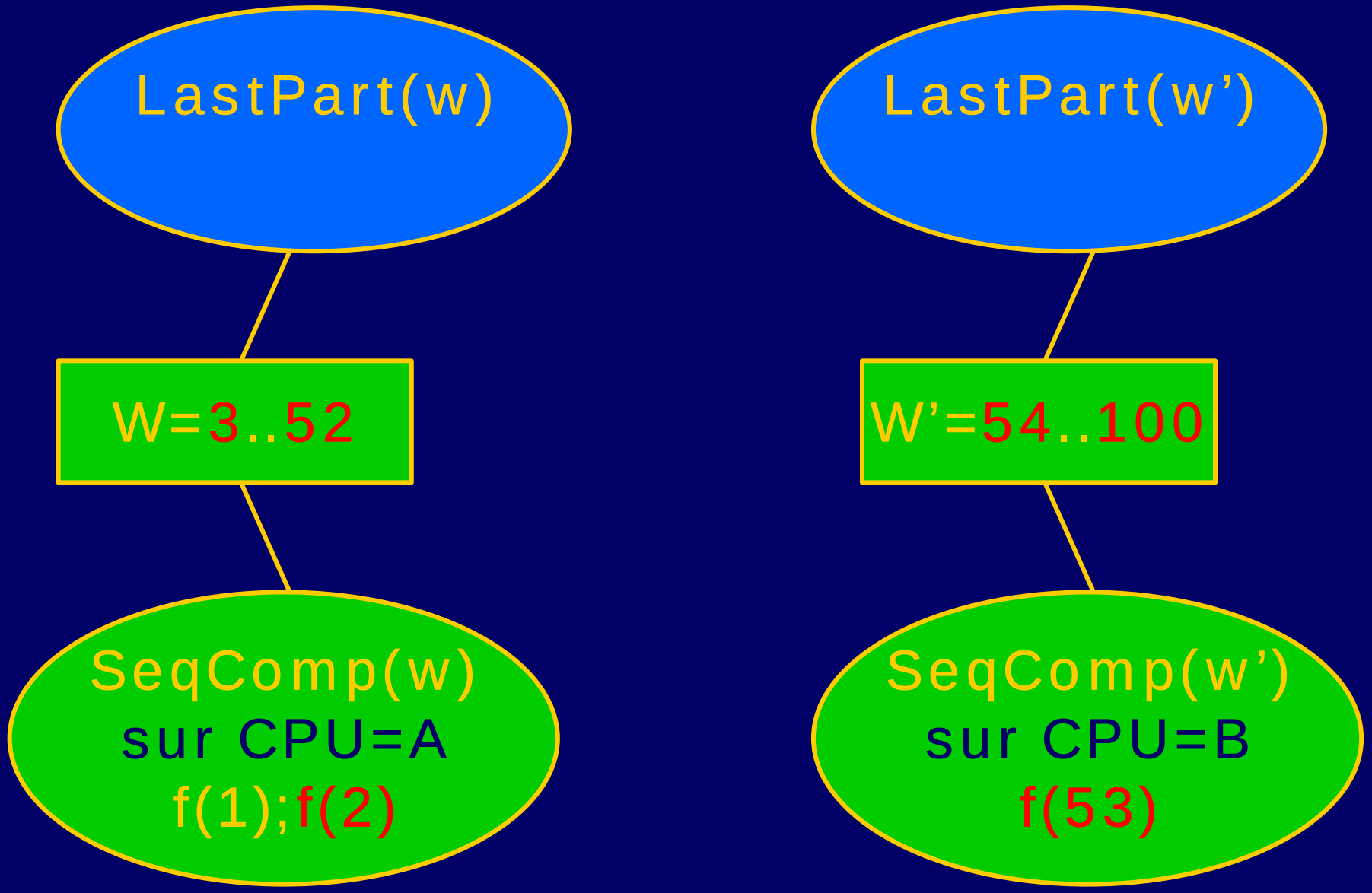
# E.g. Calcul parallèle f(i), i=1..100



# E.g. Calcul parallèle f(i), i=1..100



# E.g. Calcul parallèle f(i), i=1..100



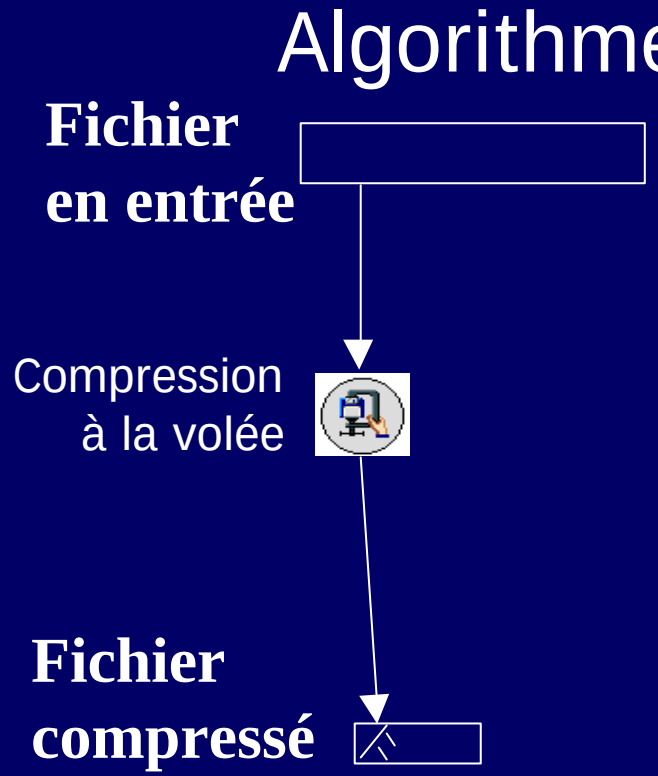
# 3. Application : parallélisation de gzip

- Gzip :

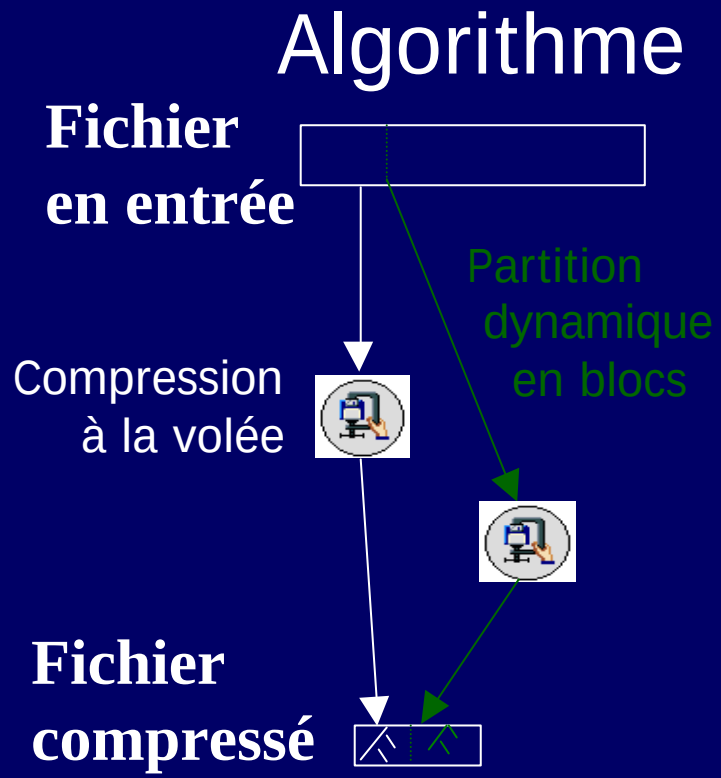
  - Utilisé (web) et coûteux bien que de complexité linéaire
  - Code source :10000 lignes C, structures de données complexes
  - Principe : LZ77 + arbre Huffman

- Pourquoi gzip ?
  - Problème P-complet, mais parallélisation pratique possible
  - Inconvénient: toute parallélisation (connue) entraîne un surcoût

# Comment paralléliser gzip ?



# Comment paralléliser gzip ?



# Comment paralléliser gzip ?

Algorithme

**Fichier en entrée**

Compression à la volée

Partition dynamique en blocs

**Fichier compressé**

=>

=>

**Partition statique en blocs**

***Compression parallèle***

**Blocs compressés**

# Comment paralléliser gzip ?

Algorithme

**Fichier en entrée**

Compression à la volée

Partition dynamique en blocs

**Fichier compressé**

=>

=>

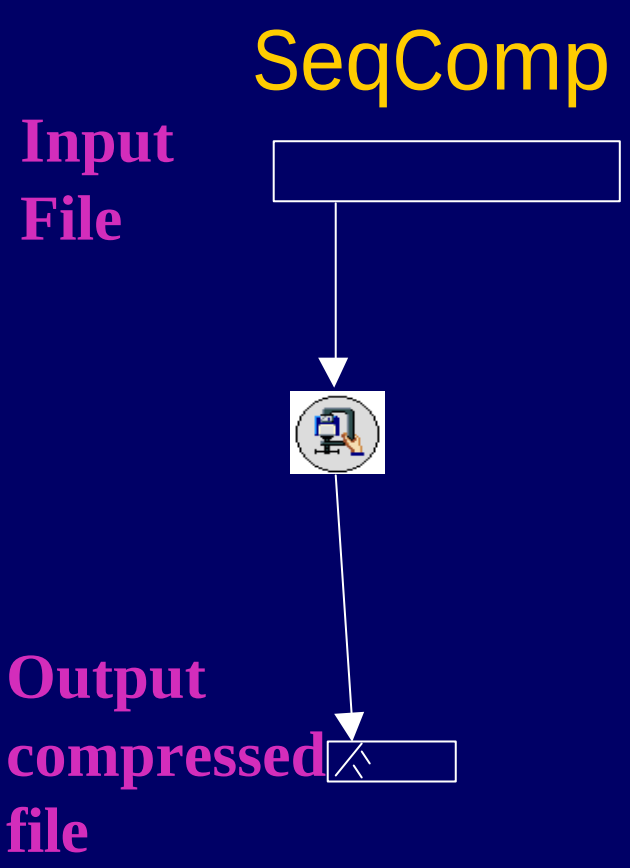
**Partition statique en blocs**

***Compression parallèle***

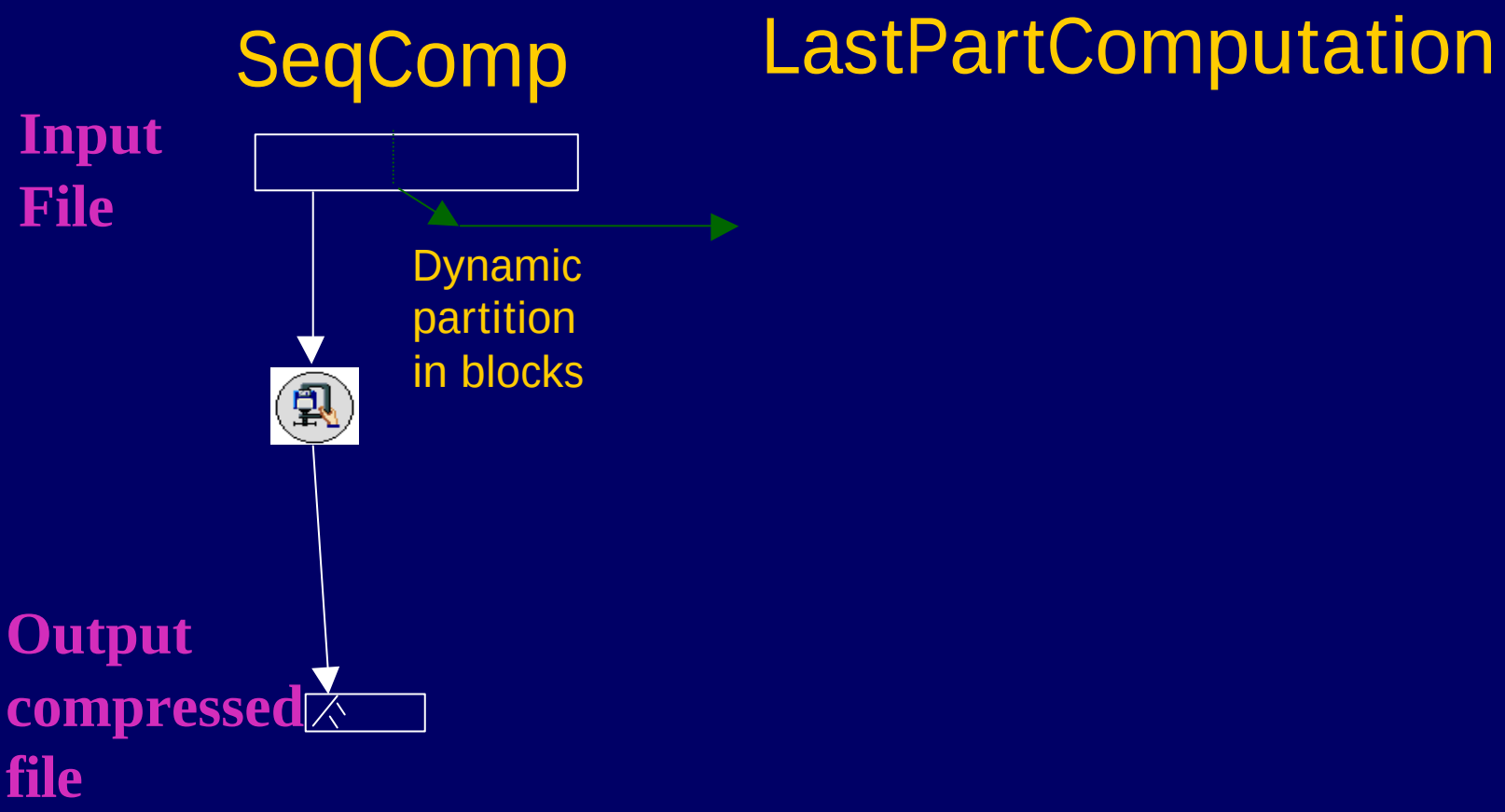
**Blocs compressés**

- Parallélisation « facile » ,100% compatible avec gzip/gunzip
- Problèmes : perte de taux de compression, grain par robuste, surcoût

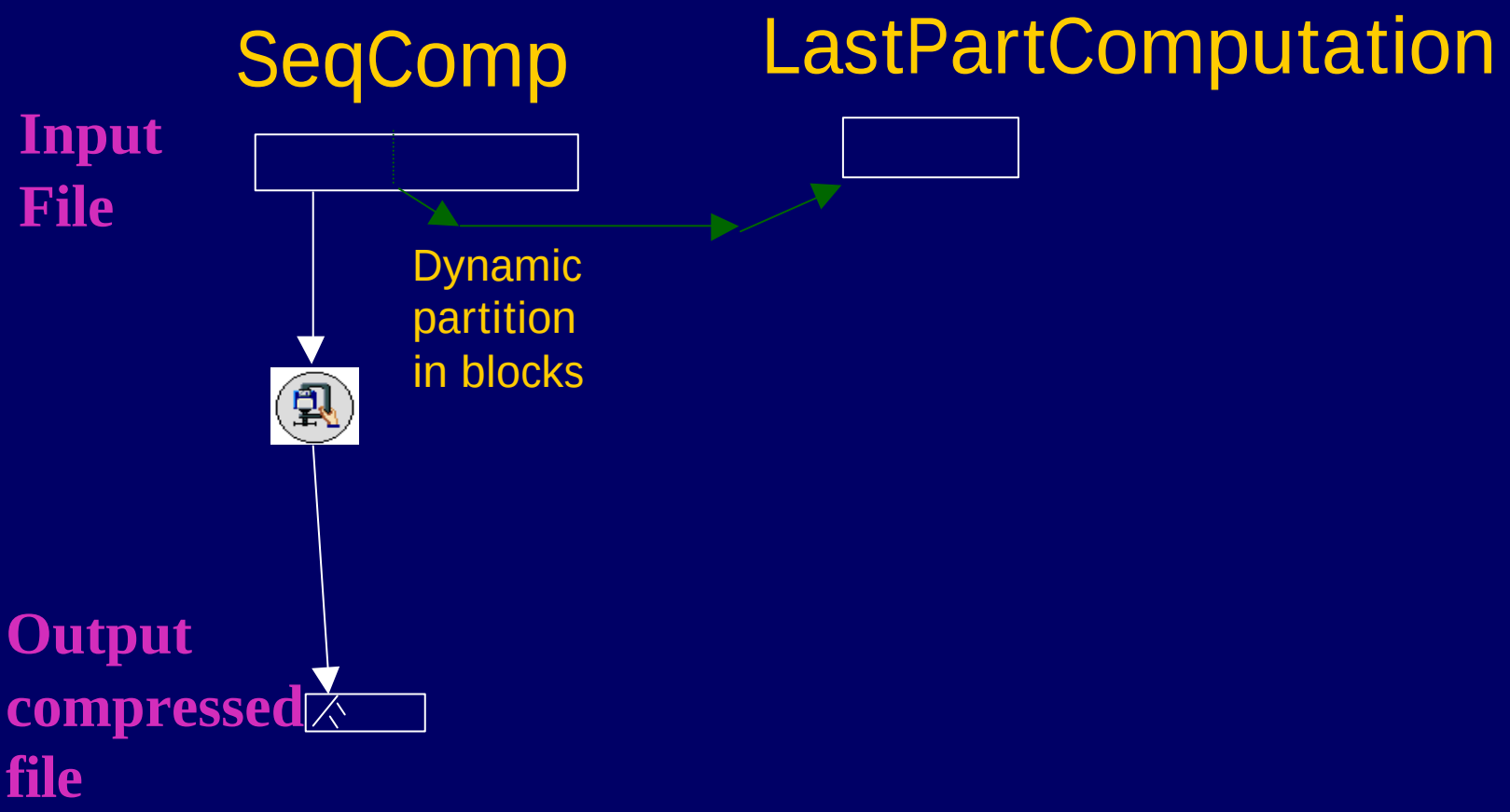
# Parallélisation gzip à grain adaptatif



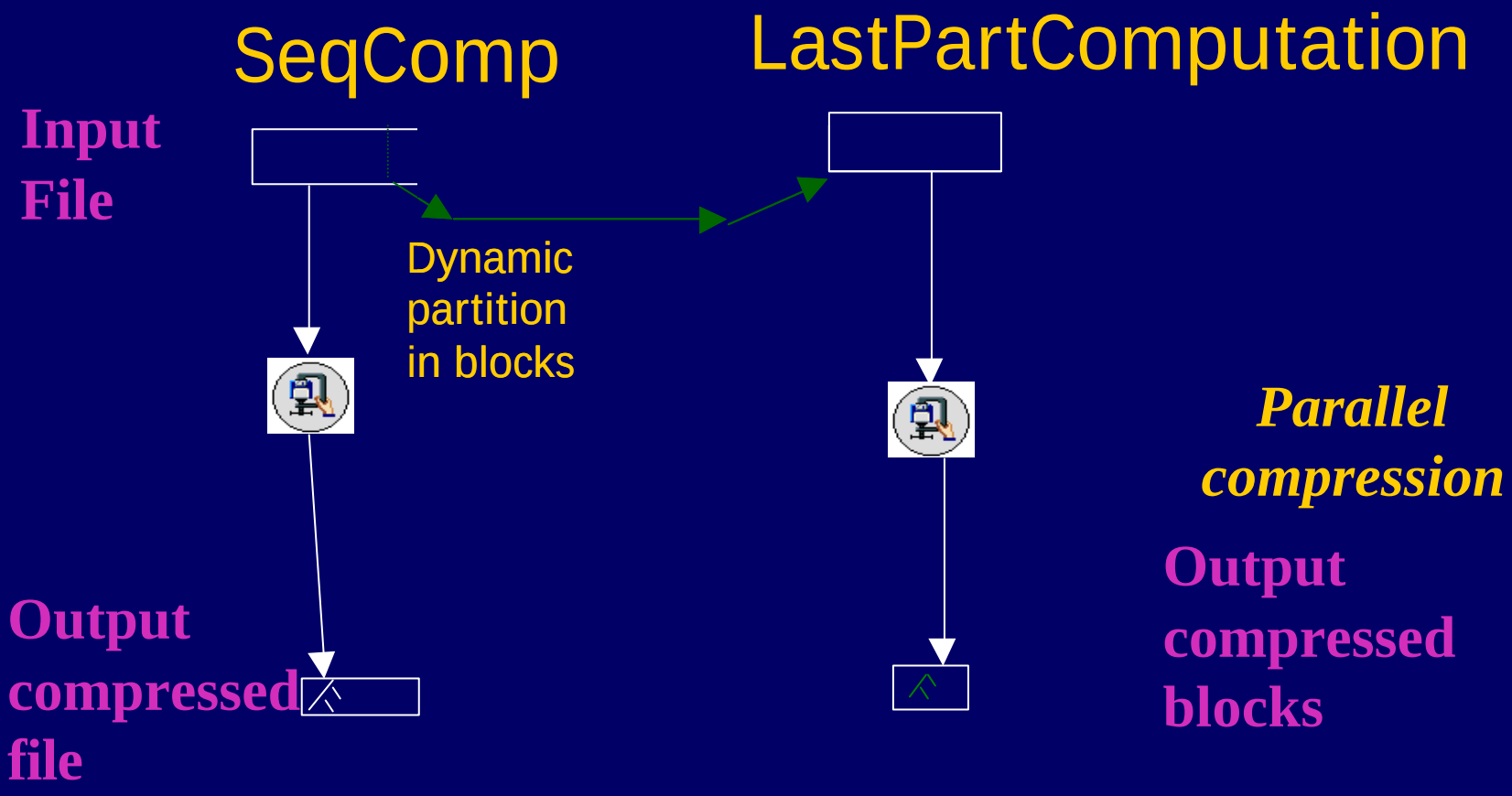
# Parallélisation gzip à grain adaptatif



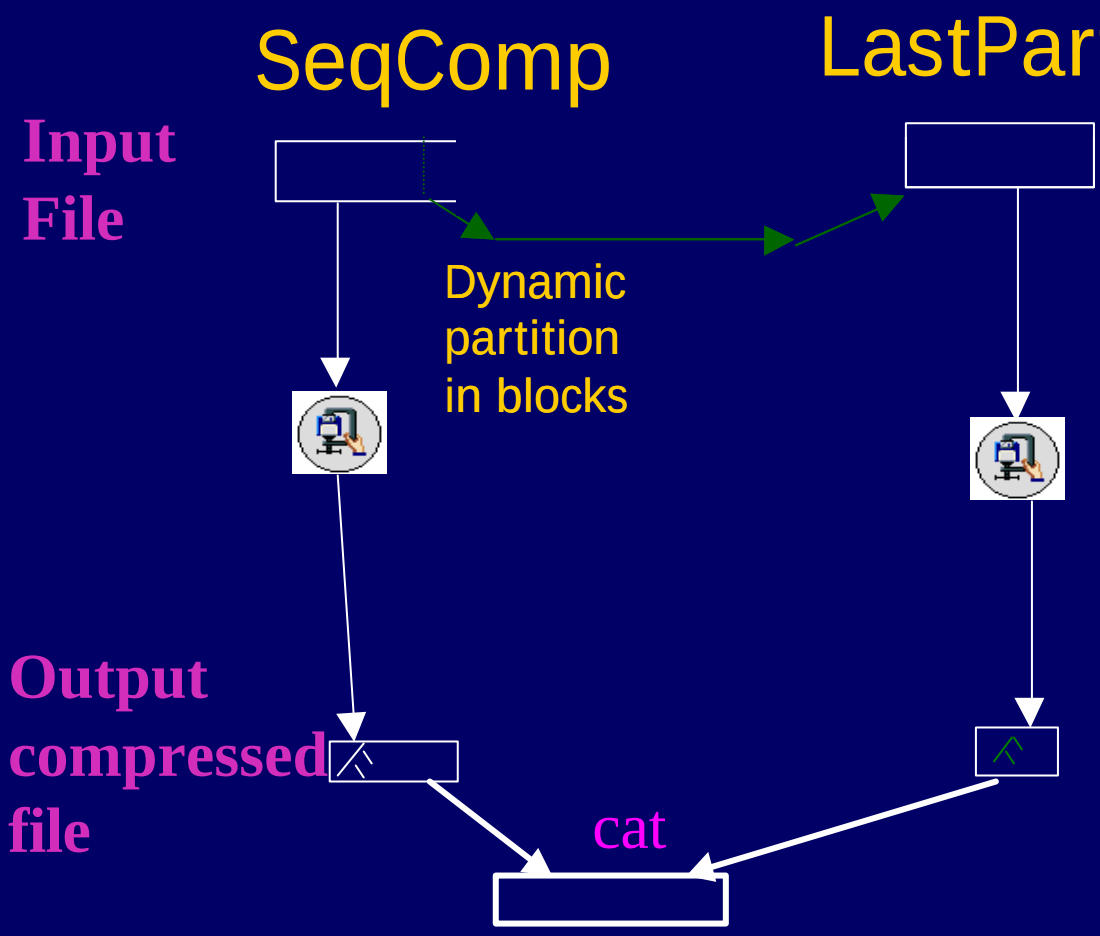
# Parallélisation gzip à grain adaptatif



# Parallélisation gzip à grain adaptatif



# Parallélisation gzip à grain adaptatif



# Conclusion

**Grain adaptatif**

Couplage de 2 algos : 1 séquentiel, 1 parallèle “récursif”
Génération de parallélisme que sur inactivité de ressources
Opérateur de base : ExtractPar de travail séquentiel en cours
Programmation générique, originale ... et simple !

**Intérêt** Réduction du surcoût lié au parallélisme :

- création de tâche, ordonnancement
- surcoût arithmétique intrinsèque
- Gain pratique: code PL inférence probabiliste [Mazer, SHARP]

**Perpsectives**

- Expérimentations SMP et distribuées : gzip, préfixes, ....
- Extension au cas distribué et hétérogène : ajout/résilience
- Extensions à d’autres algorithmes: Gauss, B&B, ...