

# TP 1,2 & 3

Master 2 SID

Benoist GASTON

[benoist.gaston@univ-rouen.fr](mailto:benoist.gaston@univ-rouen.fr)



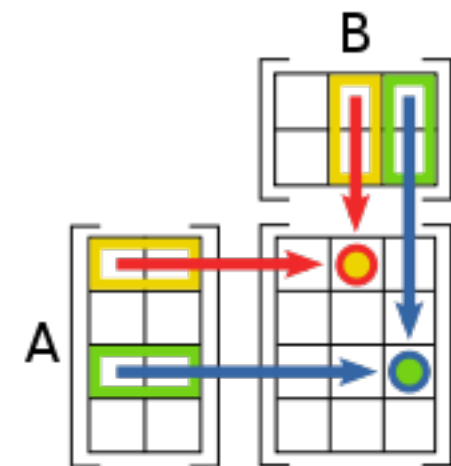
# OpenMP Matricielle

- Considérer le programme **prodmat** (<https://github.com/benoistgaston/m2sid-2020.git>) qui effectue une multiplication de deux matrices A et B en stockant le résultat dans une matrice C. Il est composé de plusieurs séquences de calcul sous forme de boucles sur les indices des matrices.
- On se propose de partager les calculs entre différents threads OpenMP.
- **Questions**
  1. Prendre en main le code ; le compiler à l'aide du makefile.
  2. Identifier les boucles à paralléliser et positionner les directives OpenMP **parallel** et **for** (en utilisant un **schedule runtime**)
  3. Modifier le makefile afin d'intégrer l'option openMP
  4. Compiler et exécuter en jouant à l'aide de variable d'environnement sur le nombre de threads et sur le **schedule**

$$A = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix} B = \begin{pmatrix} b_{11} & \cdots & b_{1p} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{np} \end{pmatrix}$$

$$AB = C = (c_{ij})_{n \times p}$$

$$c_{ij} = \sum_{k=0}^n a_{ik} \times b_{kj}$$



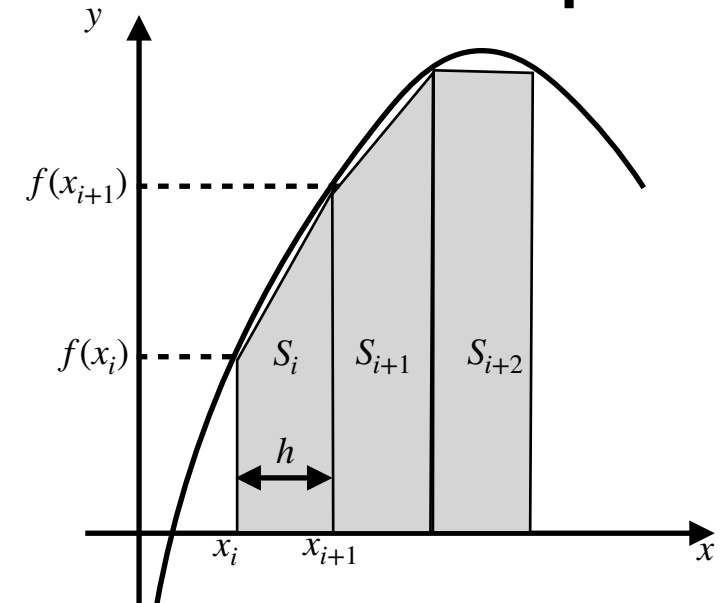
# OpenMP Calcul intégral

- Considérer le programme `integcos` (<https://github.com/benoistgaston/m2sid-2020.git>) qui effectue le calcul de l'intégrale la fonction  $\cos^2$  sur l'intervalle  $[0, \dots, \pi/4]$  par la méthode des trapèzes.
- Rappel : la valeur de cette intégrale est égale à  $\pi/8 + 1/4$
- On se propose de partager ce calcul entre différents threads OpenMP.

## • Questions

1. Prendre en main le code ; le compiler à l'aide du makefile.
2. Insérer les directives OpenMP appropriées dans le fichier `integcos.c`. La zone parallèle est déjà définie, il reste à insérer les directives de partage des données et du travail. On utilisera les directives : **section**, **single**, **for** et **reduction**.
3. Analyser les performances de la version parallèle.

## Méthode des trapèzes



## Formule pour $\cos^2$

$$\int_0^{\pi/4} \cos^2(x) dx = \frac{1}{2} \cos^2(0) + \cos^2(h) + \cos^2(2h) + \dots + \cos^2((n-1)h) + \frac{1}{2} \cos^2(nh)$$

# OpenMP Fibonacci

- Considérer le programme `fib.c` (<https://github.com/benoistgaston/m2sid-2020.git>) qui calcul de manière récursive la suite de fibonacci.

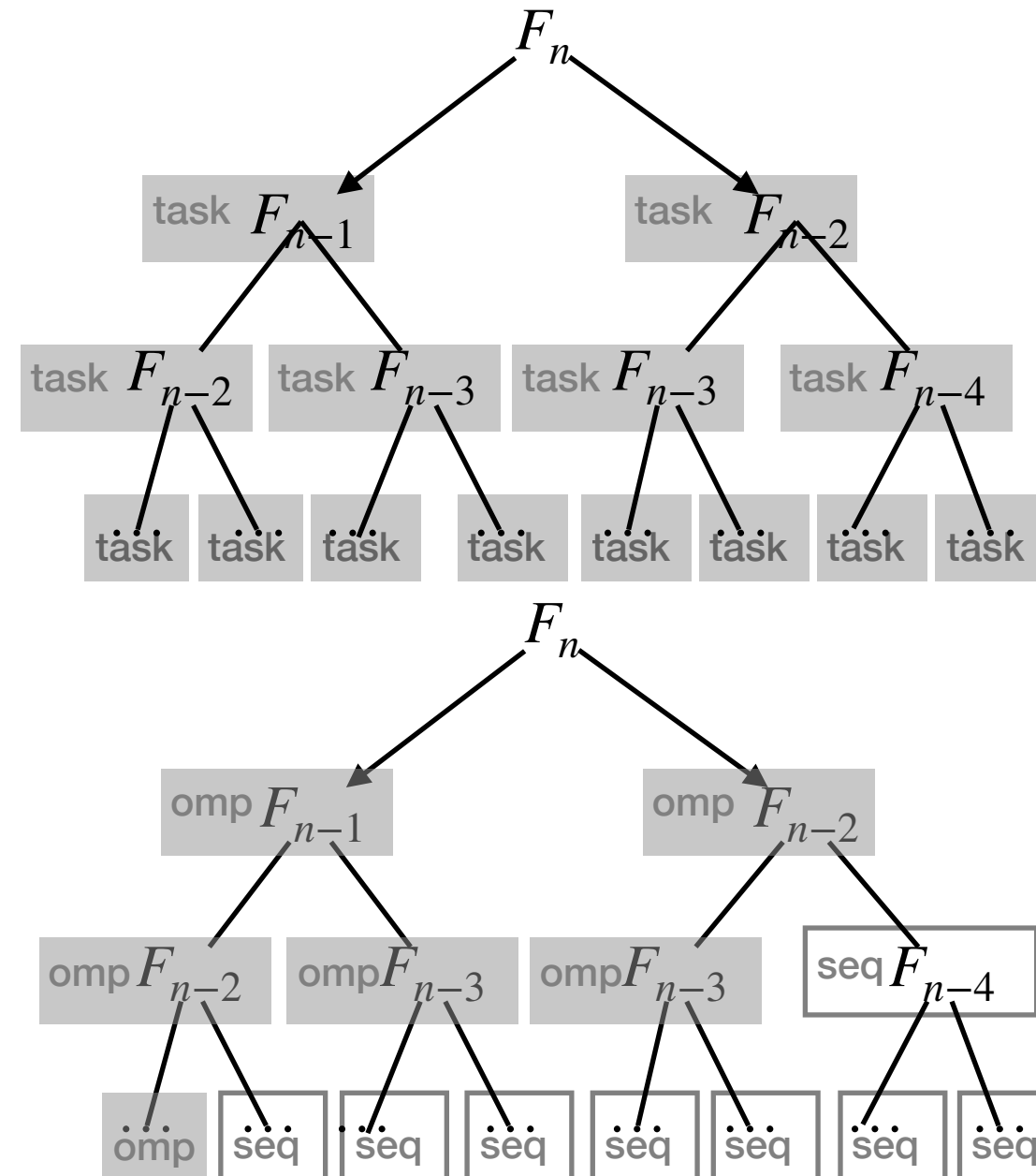
$$F_n = n, n = 0, 1$$

$$F_n = F_{n-1} + F_{n-2}, n \geq 2$$

- On se propose de paralléliser la fonction `fib_rec()` avec openmp sur le paradigme de distribution de tâches. Ce paradigme reprend l'exemple divide and conquer présenté en cours.

- **Questions**

1. Prendre en main le code ; le compiler à l'aide du makefile. Faire tourner pour des valeurs de  $n$  10, 20, 30 40.
2. Sur la base de la fonction `fib_rec()`, écrire une fonction `fib_omp()` parallélisant à l'aide de tâche.
3. Observer les performances de la version parallèle (utiliser la commande système `time`).
4. Pour résoudre le problème de performance constaté, définir dans `fib_omp()` un seuil en dessous duquel la fonction `fib_rec()` sera appelée à la place de `fib_omp()`.
5. Observer les performances de cette version hybride.



# OpenMP Bubble Sort

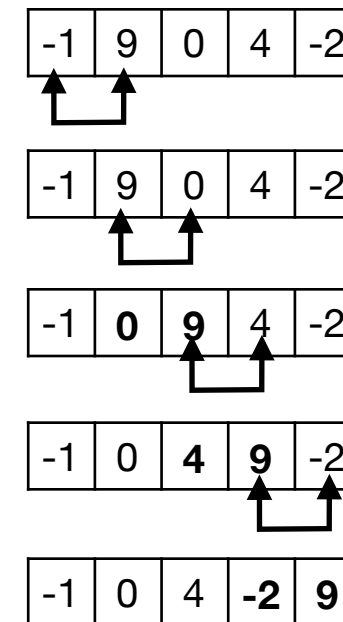
- Considérer le programme `b-sort.c` (<https://github.com/benoistgaston/m2sid-2020.git>) qui propose une implémentation du tri à bulle.

```
tri_à_bulles(Tableau T)
  pour i allant 1 de (taille de T)-1
    pour j allant de 0 à i-1
      si T[j+1] < T[j]
        échanger(T[j+1], T[j])
```

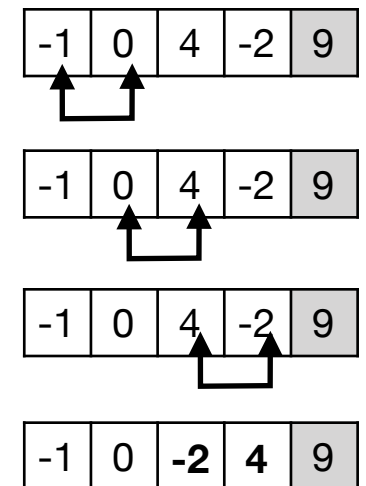
- On se propose de paralléliser la fonction `b-sort()` avec openmp.

## Questions

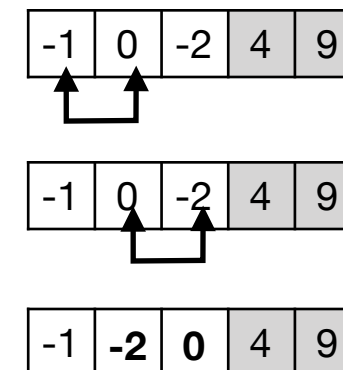
- Prendre en main le code ; le compiler à l'aide du makefile.
- Tenter une directive `parallel for` sur la boucle interne.
- Y a-t-il une erreur à la compilation ? À l'exécution (tester plusieurs fois) ?
- Comment peut-on modifier l'algorithme pour la boucle `for` parallélisable ?
- Modifier l'algorithme et le paralléliser.



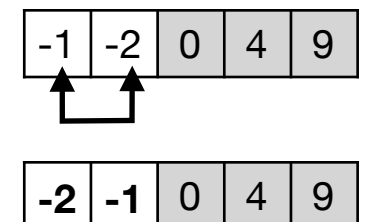
Étape 1



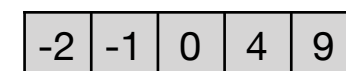
Étape 2



Étape 3



Étape 4



Final

# CUDA prise en main

L'ensemble des TPs sont proposés sous forme de notebook (<https://github.com/benoistgaston/m2sid-2020.git>).

Les notebooks peuvent être joués sur l'environnement colab research de google :

<https://colab.research.google.com>

Nécessite un compte google.

## Addition de matrice : `tp01-addmat.ipynb`


L'objectif est d'additionner deux matrices terme à terme sur GPU. On considérera des threads organisés en 1 seul bloc 2D.

1. Compléter le kernel `add_mat` afin que chaque thread prenne en charge une et une seule addition
2. Compléter les transferts mémoire entre le host et le device
3. Appeler le kernel pour les matrices `g_A` et `g_B` en utilisant un bloc correctement dimensionné

On se propose de traiter des matrices plus grosses

4. Sur la base du kernel `add_mat` créer un kernel `add_huge_mat` dans lequel chaque thread prend en charge `N` additions terme à terme
5. Appeler le kernel sur les matrices `g_HA` et `g_HB` en utilisant un bloc correctement dimensionné

## Addition de matrice

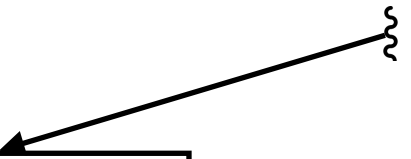
$$\begin{bmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \dots & a_{mn} \end{bmatrix} + \begin{bmatrix} b_{11} & \dots & b_{1n} \\ \vdots & \ddots & \vdots \\ b_{m1} & \dots & b_{mn} \end{bmatrix} = \begin{bmatrix} a_{11} + b_{11} & \dots & a_{1n} + b_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} + b_{m1} & \dots & a_{mn} + b_{mn} \end{bmatrix}$$


## Multiplication de matrice : `tp02-multmat.ipynb`

L'objectif est de réaliser une multiplication de deux matrices terme à terme sur GPU. On considérera des threads organisés en bloc 1D ou 2D et en grille 1D ou 2D.

1. Compléter le kernel `mult_mat` afin que chaque thread prenne en charge une boucle interne.
2. Appeler le kernel pour les matrices `g_a` et `g_b` en utilisant des blocs et des grilles correctement dimensionnés.

```
for i in range(dim1):  
    for j in range(dim2):  
        for k in range(dim3):  
            c[i][j] += a[i][k] * b[k][j]
```



# CUDA prise en main

## Addition de matrice

### Produit scalaire N-vecteur : tp03-dotvec.ipynb

Objectif : réaliser le produit scalaire de deux vecteurs 1D sur GPU. Le produit scalaire est donné par la formule et l'algorithme ci contre.

Dans un premier temps on ne réalisera sur GPU que les produits terme à terme de chaque vecteur ; **la somme est réalisée sur le host**.

1. Écrire le kernel `dot_vec` afin que chaque thread prenne en charge une et une seule multiplication terme à terme.
2. Appeler le kernel pour les vecteurs `g_va` et `g_vb` en utilisant un seul bloc correctement dimensionné.

Réfléchir à un ou plusieurs algorithmes permettant de réaliser la somme finale sur le device:

- Une première solution consiste à réaliser la somme par un seul thread.
- Une seconde solution peut s'inspirer du schéma ci-contre.

**Note:** on peut opérer si une synchronisation (barrière) des threads d'un bloc par la fonction `cuda__syncthreads()`

3. Sur la base du kernel `dot_vec` créer un kernel `complete_dot_vec` qui permet de réaliser les produit terme à terme et la somme finale.
4. Tester les différents algorithmes.
5. Selon vous, quelle sont les limites de ces méthodes de réduction sur GPU ?

$$[a_1, a_2, \dots, a_n] \times [b_1, b_2, \dots, b_n] = a_1b_1 + a_2b_2 + \dots + a_nb_n$$

```
res=np.empty_like(va)
for i in range(va.size):
    res[i]=va[i]*vb[i]
dotprod=sum(res[:])
```

