Le Parallélisme Pour Les Nuls

Laura Montagnier

February 2024

1 Cours 1

Introduction

La programmation utilisant le parallélisme s'oppose à la programmation séquentielle. Elle utilise plusieurs **threads** afin d'exécuter plusieurs tâches en même temps, en parallèle.

Important : La programmation en parallèle ne peut pas se faire de la même manière peu importe où on l'exécute. Il faut connaître son matériel : tout particulièrement l'architecture de la machine.

1.1 Pourquoi programmer en parallèle ?

On cherche à profiter du fait d'avoir plusieurs coeurs pour les utiliser simultannément pour la même application. Cela est possible en disposant de plusieurs processeurs accédant à la même mémoire ou en connectant avec des câbles Ethernet (rapidité) plusieurs ordinateurs.

Loi de Moore : en 1965, Gordon Moore a remarqué que le nombre de transitors dans les processeurs actuels doublait en 18 mois. Ce fût vrai jusque dans les années 2000, mais maintenant la loi est toujours vérifiée avec le nombre de coeurs de calcul dans un processeur ou un ordinateur.

1.2 Types d'architectures parallèles

1.2.1 Architecture à mémoire partagée

Cette architecture est présente dans les téléphones ou les tablettes. Elle comporte néanmoins des problèmes d'accès mémoire si l'on cherche à lire et écrire dans le même emplacement mémoire en même temps.
Exemple : architecture d'Haswell

Au plus il y a de coeurs, au plus les problèmes d'accès mémoires dans ce genre d'architecture sont fréquents.

1.2.2 Utilisation de RAM

La **RAM**, ou mémoire vive (Random Access Memory en anglais), est un type de mémoire informatique utilisée par les ordinateurs et d'autres dispositifs électroniques pour stocker temporairement des données qui sont actuellement en cours d'utilisation ou traitées activement par le processeur.

"Interleaved Rams" : On dispose de 3 RAMs. Le processus 1 utilise la RAM 1, le processus 2 utilise la RAM 2, le processus 3 utilise la RAM 3, et le processus 4... utilise la RAM 1, en comptant sur le fait que le processus 1 ait terminé !

Le nombre de "chemins" est le nombre de RAMs, et la largeur du chemin est le nombre d'octet que chaque RAM contient. Au plus on a de RAM, au moins on a de problèmes, mais au plus ça coûte cher !

1.2.3 Cache mémoire

Le problème des RAMs, c'est qu'elles ont une **architecture en grille**. Cela prend donc du temps de parcourir les colonnes et les lignes.

> La **mémoire cache** est une petite unité de mémoire vive utilisée rapidement et gérée par le CPU. Afin d'optimiser les temps d'exécution, le programmeur doit donc connaître les stratégies du CPU !

Le CPU utilise deux principales stratégies :

1. stocker les variables mémoires contigues dans le cache (de 64 octets pour un processeur Intel classique).
2. chaque adresse de mémoire cache est liée à une adresse fixe sur la RAM (en utilisant les modulos)

Il faut donc adopter 3 stratégies :

1. accès à la mémoire contigu
2. utiliser le contenu de la mémoire cache le plus rapidement possible pour s'en "débarasser" vite
3. **Voir les emplacements mémoires comme à un endroit dans l'ESPACE et dans le TEMPS.**

1.2.4 Outils aidant à la mémoire partagée

1. **OpenMP :** directives de compilation et une petite API en C (pragma).
2. **Librairies classiques en C++ :** threads, fonctions asynchrones, politiques d'exécution...
3. **oneTTB :** Librairie libre de droits produite par Intel.

1.2.5 Mémoire distribuée

Explications non contractuelles fournies par ChatGPT :

1. **Répartition de la mémoire :** Au lieu d'avoir une mémoire centrale partagée par tous les nœuds, chaque nœud dispose de sa propre mémoire locale. Cela permet d'augmenter la capacité totale de la mémoire en ajoutant simplement plus de nœuds au système.
2. **Coordination et communication :** Les nœuds dans un système de mémoire distribuée doivent être capables de communiquer entre eux. Des protocoles de communication et des mécanismes de synchronisation sont utilisés pour permettre aux nœuds de partager des informations, d'accéder à des données situées sur d'autres nœuds et de coordonner leur activité.
3. **Avantages de la distribution :** La mémoire distribuée offre plusieurs avantages, notamment la scalabilité, la résilience et la répartition de la charge de travail. La scalabilité signifie que vous pouvez augmenter la capacité du système en ajoutant simplement plus de nœuds.

1.3 Utiliser MPI

Managing the context in your program

- Call the initialization of the parallel context before using other functions in the library (`MPI_Init`).
- Obtain the number of processes contained in the communicator (`MPI_Comm_size`).
- Retrieve the rank of the process within the communicator (`MPI_Comm_rank`).
- After calling the last library function, invoke the termination of the parallel context to synchronize processes (`MPI_Finalize`). Failure to do so may result in program crashes.

```
#include <mpi.h>
int main(int nargs, char const* argv[])
{
    MPI_Comm commGlob;
    int nbp, rank;
    MPI_Init(&nargs, &argv);// Initialization of the parallel context
    MPI_Comm_dup(MPI_COMM_WORLD, &commGlob);// Copy global communicator in own communicator;
    MPI_Comm_size(commGlob, &nbp);// Get the number of processes launched by the used;
    MPI_Comm_rank(commGlob, &rank);// Get the rank of the process in the communicator commGlob.
    ...
    MPI_Finalize();// Terminates the parallel context
}
```

Nous allons utiliser python, grâce à la Library MPI, qui permet la programmation en parallèle.

1.4 Problèmes à éviter

Interlocking : le processus 0 attend un message du processus 1 avant d'envoyer un message, et le processus 1 attend un message du processus 0 avant de pouvoir envoyer son message.

Blocking message : attend la réception entière d'un message avant de retourner à la fonction.

Buffer/Tampon : Si un message envoie une donnée avant sa modification, sa première donnée est stockée dans un tampon. On perd donc du temps de stockage mémoire qui n'est pas nécessaire. Il faut donc éviter d'envoyer des données qui vont être modifiées.

Broadcast/Collective communication : Peut parfois être utile. C'est lorsqu'un processus envoie ou demande de l'information à tous les autres processus.

1.5 Règles indispensables de la programmation parallèle :

- L'échange de données via Ethernet est très lent comparé à l'accès à la mémoire : minimisez autant que possible les échanges de données.
- Pour masquer les coûts d'échange de données, il est préférable de calculer certaines valeurs pendant l'échange de données : privilégiez l'utilisation d'échanges de messages non bloquants.
- Chaque message a un coût initial : regroupez les données dans un tampon temporaire si nécessaire.
- Assurez-vous que tous les processus sortent du programme simultanément : essayez d'équilibrer la charge de calcul entre les nœuds de calcul.

2 Cours 2

Performance de parrallèlisation

2.1 Outils de performance

2.1.1 Le Speed-Up

On définit le speed-up par : $S(n) = \frac{Ts}{T(n)}$ avec Ts = Temps d'exécution séquentiel et T(n) = Temps d'exécution avec n unité de calcul.
C'est, en gros, l'accélération en passant de séquentiel en parallèle.

Remarque : Souvent, l'algorithme séquentiel est bien différent de l'algorithme parallélisé. Il est alors compliqué de calculer le speed-up. On peut se demander quelle est la complexité de l'algorithme séquentiel, est-ce qu'il utilise de manière optimale la mémoire cache, etc...

Loi d'Amdhal : si on pose ts le temps d'exécution du code en séquentielle et f la proportion de temps relative à l'exécution du code qui ne peut pas être mis en parallèle, alors on peut calculer la valeur limite du speed-up.

$$S(n) = \frac{t_s}{f.t_s + \frac{(1-f)t_s}{n}} = \frac{n}{1+(n-1)f} \xrightarrow[n\to\infty]{} \frac{1}{f}$$

C'est utile pour trouver le nombre n de coeurs à utiliser lors de la parallélisation d'un programme.

Loi de Gustafson : Le comportement du speed-up demeure constant avec un volume fixé de données d'entrées par processus.

La loi de Gustafson donne le calcul suivant :

- Let t_s be the time taken by the execution of the sequential part of the code ;
- Let t_p be the time taken by the execution of the parallel part of the code for a fixed amount of data.
- **Hypotheses** :
  - $t_s \geq 0$: The time to execute the sequential part of the code is independent of the volume of input data ;
  - $t_p > 0$: The time to execute the parallel part of the code is linearly dependent on the volume of input data.
  - Let's consider $t_s + t_p = 1$ (one unit of time).

$$S(n) = \frac{t_s + n.t_p}{t_s + t_p} = n + \frac{(1-n)t_s}{t_s + t_p} = n + (1-n).t_s$$

2.1.2 Scalability

La **Scalability** est le comportement du Speed-Up quand on modifie le nombre n de coeurs de calcul ou le nombre de données d'entrée.
On l'évalue pour le **pire Speed-Up :** on fixe le nombre de données, et on dessine la courbe du Speed-Up en fonction du nombre de processus.

On l'évalue pour le **meilleur Speed-Up :** on fixe le nombre de données PAR PROCESSUS, puis on trace la courbe du Speed-Up en fonction du nombre de processus.

2.1.3 La granularité

- Le **Gros Grain** ou Coarse Grain, est préférable sur un CPU. Il s'agit d'un mode de programmation parallèle où l'on met beaucoup de tâches dans une "unité" d'exécution.
- Le **Grain fin** ou Fine grain est préférable sur un GPU. Lorsqu'on réunit les tâches par petit nombre afin de former des "unités" d'exécution.

Lorsqu'il y a beaucoup de tâches, asymptotiquement, la programmation en parallèle n'améliore pas le temps d'exécution. En **overhead**, on passe son temps à faire de la préemption, et le séquentiel est plus rapide que le parallèle.

2.1.4 Load balancing

Load balancing peut se traduire par "équilibrer la charge". C'est lorsque tous les processus exécutent une section de calcul du code avec la même durée.

2.2 Algorithmes ennuyeusement parallèles

2.2.1 Propriétés

Définition :

- Chaque donnée utilisée et calculée est indépendante des autres.
- Pas de compétition des données dans un contexte de multithreads.
- Pas de communication entre les processus dans un environnement distribué.

Embarassing(=ennuyeux), c'est quand c'est vraiment évident niveau mémoire. Pas besoin de se partager des informations, et chaque processus a besoin d'une zone mémoire différente et écrit dans une zone mémoire différente.

Nearly embarassing(=presque ennuyeux), c'est comme embarassing mais il faut faire une opération spéciale pour regrouper la sortie de tes processeur. En d'autres termes, la "vraie" sortie des processus n'est pas vraiment indépendante.

Exemple d'algorithme pas embarassing : un produit de matrice A*B. En effet, dans la parallélisation par ligne par exemple, tout les process vont demander en mémoire B.

Shared evironment : Ils ont tous la même mémoire, donc il y a un gros problème quand ils veulent tous y avoir accès (*memory bound*) parce qu'il n'y a qu'un seul bus.

Distributed environment : Ils ont tous leur RAM, donc pas de problème pour l'accès mémoire. Néanmoins, il pourrait y avoir un problème s'ils doivent s'échanger des informations parce que le réseau (souvent ethernet) est évidemment beaucoup plus lent qu'avoir une RAM commune.

2.3 Exemples d'algorithmes ennuyeusement parallèles

2.3.1 Addition de deux vecteurs

Load balancing : chaque thread prend une partie des valeurs et les additionne.

En mémoire distribué, le résultat est éparpillé dans les différents processus !!

2.3.2 Multiplication de deux matrices diagonales par blocs

1. Trier les blocs avec des dimensions décroissantes.
2. Initialiser les poids des processus à zéro
3. Distribuer les plus gros blocs (celui de A, B et C la matrice résultat) aux processus et ajouter leur dimension aux poids des processus.
4. Tant que : certains blocs ne sont pas distribués (ajouter le block le plus grand au processus au poid le plus petit, et ajouter sa dimension au poid du processus).

2.3.3 Suite de Syracus et algorithmes maîtres et esclaves

Definition of Syracuse series

$$\begin{cases} u_0 \text{ chosen} \\ u_{n+1} = \begin{cases} \frac{u_n}{2} \text{ if } u_n \text{ is even} \\ 3.u_n + 1 \text{ if } u_n \text{ is odd} \end{cases} \end{cases}$$

Property of Syracuse series

- One cycle exists : $1 \to 4 \to 2 \to 1 \to \cdots$
- A conjecture : $\forall u_0 \in \mathbb{N}$, the series reaches the cycle above in a finite number of iterations

La suite de Syracus est une suite de nombres qui atteint forcément son "cycle" en un nombre fini d'itérations. On appelle "longueur du vol" le nombre d'itérations qu'il faut pour entrer dans le cycle, et "hauteur du vol" la plus haute valeur atteinte par la suite. L'algorithme que nous étudions a pour but de calculer la longueur et la hauteur du vol pour beaucoup de valeurs impaires de u_0.

Algoithme :

- Chaque processus calcule la longueur et la hauteur pour un sous-ensemble de valeurs de u_0.
- L'intensité de calcul pour chaque processus dépend de la longueur des suites. Il est impossible de connaître la complexité qu'implique une suite avant de la calculer.
- Le problème n'est pas bien équilibré.

On va utiliser un algorithme sur le processus racine, ou **maître**, pour distribuer les suites aux autres processus, ses **esclaves**.

Le maître : Envoie une série de suites à calculer à chaque processus, et attend que plusieurs aient fini pour leur en renvoyer.

L'esclave : Calcule les longueurs et hauteurs des suites qu'il reçoit, et dit quand il a fini au maître.

2.4 Algorithmes presque ennuyeusement parallèles

Nearly embarassing parallel algorithm : Calcul indépendant pour chaque processus avec une communication finale pour finaliser le calcul.

Exemples : produit vectoriel de deux vecteurs, produit matrice vecteur, calcul d'une intégrale...

2.4.1 Ensemble de Bouddha

Le code pour cet exemple est donné par le professeur.

Buddha set

Let's consider the complex recursive Mandelbrot series :

$$\begin{cases} z_0 = 0, \\ z_{n+1} = z_n^2 + c \text{ where } c \in \mathbb{C} \text{ chosen.} \end{cases}$$

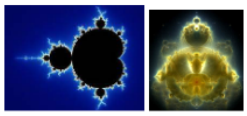


Figure – Mandelbrot (left) and Buddha (right) set

Property

- Series is divergent if $\exists n > 0; |z_n| > 2$;
- Region of interest : the disk $\mathcal{D}$ of radius 2 ;
- In some region of the disk, possible to prove convergence ;
- But chaotic convergence behaviour in some region of $\mathcal{D}$!

Mandelbrot and Buddha sets

- Mandelbrot's set :
  color c with "divergence speed" of relative series
- Buddha's set :
  Color orbit of divergent series

Buddha's set algorithm

Algorithm

- Draw N random values of c in the disk $\mathcal{D}$ where the relative series diverge ;
- Compute the orbit of this series until divergence and increment the intensity of the pixel representing each value of the orbit ;

Parallelization of the algorithm

- Master-slave algorithm to ensure load balancing ;
- For granularity, define a task as a pack of random values c ;

```python
# Calcul de l'ensemble de Mandelbrot en python
import numpy as np
from dataclasses import dataclass
from PIL import Image
from math import pi
from time import time
import matplotlib.cm
from mpi4py import MPI

twoPi = 2.*pi

@dataclass
class MandelbrotSet:
    max_iterations: int
    escape_radius : float = 2.0

    def __contains__(self, c: complex) -> bool:
        return self.stability(c) == 1

    def convergence(self, c:complex, clamp=True) -> float:
        value = self.count_iterations(c)[0]/self.max_iterations
        return max(0.0, min(value, 1.0)) if clamp else value

```

```
    def count_iterations(self, c: complex) :
        iter : int
        z = []
        z.append(c)
        for iter in range(self.max_iterations):
            z.append(z[-1]*z[-1] + c)
            if abs(z[-1]) > self.escape_radius:
                return iter, np.array(z[:-1],dtype=np.complex128)
        return self.max_iterations,np.array([],dtype=np.complex128)

# Definition d'une tâche prenant un sous paquet de samples à traiter :
def bhuddabort_task(nbSamples : int, maxIter : int, width : int, height : int ):
    radius = 2*np.random.rand(nbSamples)
    angle  = twoPi*np.random.rand(nbSamples)
    scaleX = 0.25*width
    scaleY = 0.25*height
    image = np.zeros((width, height),dtype=np.int64)
    cArr = radius*(np.cos(angle)+np.sin(angle)*1j)
    mandelbrot_set = MandelbrotSet(max_iterations=maxIter)
    for c in cArr:
        niter, orbit = mandelbrot_set.count_iterations(c)
        if niter < maxIter:
            x = np.asarray(scaleX*(orbit.real+2.),dtype=np.int32)
            y = np.asarray(scaleY*(orbit.imag+2.),dtype=np.int32)
            mask = np.logical_and(x<width, y<height)
            xfiltre = x[np.where(mask)]
            yfiltre = y[np.where(mask)]
            image[xfiltre,yfiltre] += 1
    return image
```

```python
# Bhuddabrot to test the chronometer
def bhuddabrot ( nbSamples : int, maxIter : int, width : int, height : int, comm : MPI.Comm ):
    packSize = 64
    nbp      = comm.size
    rank     = comm.rank

    nbPacks = (nbSamples+packSize-1)//packSize
    image     = np.zeros((width, height),dtype=np.int64)
    image_loc = np.zeros((width, height),dtype=np.int64)
    # Algorithme maître-escalve :
    if rank==0: # Algorithme maître distribuant les tâches

        iPack : int = 0
        for iProc in range(1,nbp):
            comm.send(iPack, iProc)
            iPack += 1
        stat : MPI.Status = MPI.Status()
        while iPack < nbPacks:
            done = comm.recv(status=stat)# On reçoit du premier process à envoyer un message
            slaveRk = stat.source
            comm.send(iPack, dest=slaveRk)
            iPack += 1
        iPack = -1 # iPack vaut maintenant -1 pour signaler aux autres procs qu'il n'y a plus de tâches à exécuter
        for iProc in range(1,nbp):
            status = MPI.Status()
            done = comm.recv(status=status)# On reçoit du premier process à envoyer un message
            slaveRk : int = status.source
            comm.send(iPack, dest=slaveRk)
        comm.Reduce([image_loc,MPI.INT64_T], [image,MPI.INT64_T], op=MPI.SUM, root=0)
```

```
83          else:
84              status : MPI.Status = MPI.Status()
85              iPack : int
86              res   : int = 1
87
88              iPack = comm.recv(source=0) # On reçoit un n° de tâche à effectuer
89              while iPack != -1:          # Tant qu'il y a une tâche à faire
90                  image_loc = bhuddabort_task(packSize, maxIter, width, height )
91                  req : MPI.Request = comm.isend(res,0)
92                  image += image_loc
93                  iPack = comm.recv(source=0) # On reçoit un n° de tâche à effectuer
94                  req.wait()
95              comm.Reduce([image,MPI.INT64_T], None, op=MPI.SUM, root=0)
96          return image
97
98      globCom = MPI.COMM_WORLD.Dup()
99      nbp     = globCom.size
100     rank    = globCom.rank
101     name    = MPI.Get_processor_name()
102
103     filename = f"Output{rank:03d}.txt"
104     out      = open(filename, mode='w')
105
106     # On peut changer les paramètres des deux prochaines lignes
107     width, height = 1024, 1024
108
109     # Calcul de chaque composante de Bhuddabrot
110     s1 = 1500_000 #150_000
111     s2 =  500_000 # 50_000
112     s3 =    30000 # 3_000
113     deb = time()
```

```
    deb = time()
    out.write("red\n")
    redOrbit   = bhuddabrot( s1,  2_000, width, height, globCom)
    out.write("green\n")
    greenOrbit = bhuddabrot(  s2, 10_000, width, height, globCom)
    out.write("blue\n")
    blueOrbit  = bhuddabrot(   s3, 10_000, width, height, globCom)
    fin = time()
    out.write(f"Temps du calcul de l'ensemble de Bhuddabrot : {fin-deb} secondes\n")

    if rank==0:
        # Constitution de l'image résultante :
        deb=time()
        b1 = np.sum(redOrbit)
        b2 = np.sum(greenOrbit)
        b3 = np.sum(blueOrbit)
        stride : int = width*height
        scal1 : float = 16.*stride/b1
        scal2 : float = 16.*stride/b2
        scal3 : float = 16.*stride/b3
        red   = np.array(np.clip((scal1*redOrbit).astype(np.uint8),0,255))
        green = np.array(np.clip((scal2*greenOrbit).astype(np.uint8),0,255))
        blue  = np.array(np.clip((scal3*blueOrbit).astype(np.uint8),0,255))
        pixels = np.stack((red,green,blue),axis=-1)
        image = Image.fromarray(pixels, 'RGB')
        fin = time()
        out.write(f"Temps de constitution de l'image : {fin-deb} secondes\n")
        image.save("bhudda.jpg")
```

3 Cours 3

Algorithmes de tri en parallèle

Les meilleurs algorithmes de tri séquentiels ont une complexité de O(n log(n)).

Par conséquent, le meilleur Speed-Up possible en utilisant n processeurs est de l'ordre :

$\frac{O(n*log(n))}{n} = log(n)$

Attention : Cela n'est pas forcément une bonne idée d'utiliser n processeurs pour trier n données...

3.1 Théorie des tris en parallèle

Tri Naïf : On compare les éléments 1 à 1. On peut partager les éléments entre différents threads. Chacun organise sa liste, puis il la merge avec un autre, avant d'en récupérer la partie haute ou la partie basse. Ainsi, les processeurs travaillent à chaque fois 2 à 2. On appelle cela le **data partitionning**, c'est-à-dire le partage de données.

Opérations "compare and exchange" : Il y a deux manières de faire. De manière symétrique, A et B s'envoient leurs valeurs. Ils les comparent de leur côté, et conservent la plus grande ou la plus petite, selon ce qu'on leur a demandé de faire. De manière asymétrique, A envoie à B sa valeur. B compare les deux, et conserve la plus petite (ou la plus grande) et renvoie l'autre à A. Comme ça, il n'y a qu'une comparaison à faire.

3.2 Différents algorithmes de tri en parallèle

3.2.1 Algorithme de tri pair-impair

C'est une parallélisation du tri à bulles !

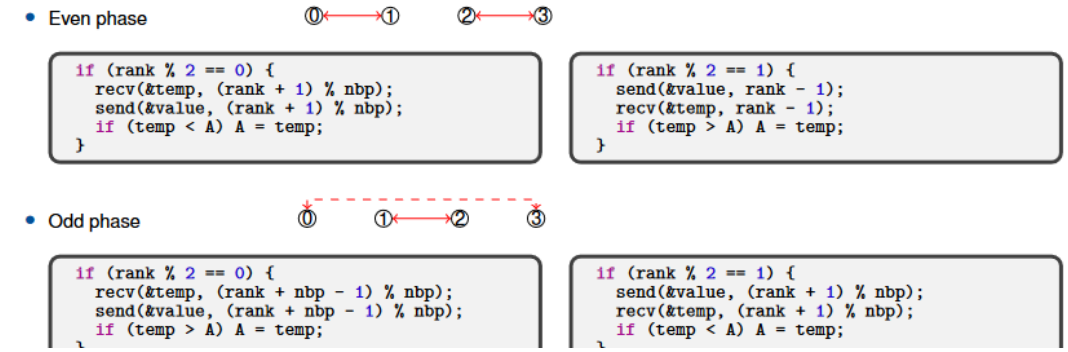
Bubble sort :

Le tri à bulles **séquentiel** ou "tri par propagation" est un algorithme de tri. Il consiste à comparer répétitivement les éléments consécutifs d'un tableau, et à les permuter lorsqu'ils sont mal triés. Il doit son nom au fait qu'il déplace rapidement les plus grands éléments en fin de tableau, comme des bulles d'air qui remonteraient rapidement à la surface d'un liquide.

Le tri à bulles est souvent enseigné en tant qu'exemple algorithmique, car son principe est simple. Mais c'est le plus lent des algorithmes de tri communément enseignés, et il n'est donc guère utilisé en pratique. (Complexité en O(n^2).)

Algorithme Pair-Impair : Basé sur l'idée que les séquences triées puissent s'entrecouper. Il permet de paralléliser le tri à bulles.

Explication de l'algorithme pair-impair.



On peut adapter cet algorithme en l'implémentant "par blocs", c'est-à-dire en donnant plus qu'un nombre à chaque processus. Il les trie avant de les trier avec les autres processus.

3.2.2 Algorithme de tri transparent

Problème de base : On veut trier un tableau comme un serpent. On trie les lignes paires par ordre croissant, et les lignes impaires par ordre décroissant. On procède par phase paire et phase impaire. En phase paire, on trie les lignes, en phase impaire, on trie les colonnes. Après log(n)+1 phases, le tableau est trié !

3.2.3 Algorithme de tri rapide : "Quick-Sort"

Quick-Sort séquentiel : On choisit un pivot, puis on trie ceux qui sont plus petits et plus grands que le pivot. Ensuite, on choisit un autre pivot dans chaque list de nombres supérieurs et inférieurs au pivot, jusqu'à n'avoir que des listes à un seul élément. C'est un principe réccursif. Sa complexité est optimale : O(n*log(n)).

Code séquentiel pour Quick-Sort

Sequential Code

```
void quicksort(T* list, T* start, T* end) {
    auto pivot = choosePivot(start, end);
    if (start < end) {
        split(list, start, end, pivot);
        quicksort(list, start, pivot-1);
        quicksort(list, pivot+1, end);
    }
}
```

Pour le paralléliser, on va tenter une approche "diviser pour mieux régner". On arrive à un tri spécifique : le tri **HYPERQUICK**. On commence par construire un "hypercube" dont les sommets sont numérotés en binaire.

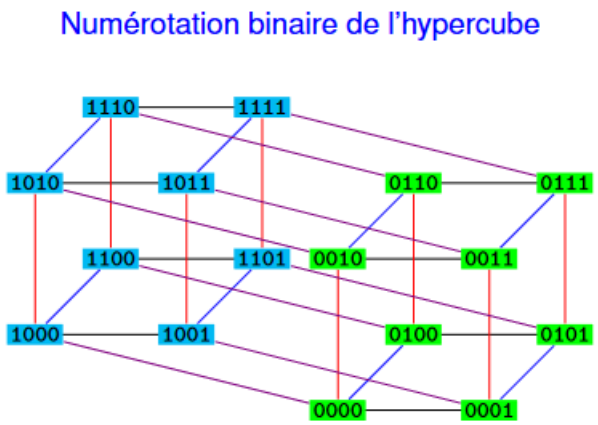


Figure – Dimension 4

Gray Code : La distance entre deux sommets est le nombre de bits de différence entre les deux. Deux sommets voisins ont un seul bit de différence par exemple.

Principe : Chaque processeur a un numéro : 0, 1, 2, etc... On les écrit en binaire, comme ça on peut passer de l'un à l'autre en ajoutant ou en retirant des bits. Il suffit que les processus échangent avec leurs voisins pour que le tableau soit trié. C'est le même principe que pour pair-impair, mais en plus complexe, et beaucoup plus rapide.

4 Cours 4

Décomposition des domaines

4.1 Généralités

4.1.1 Overlapping et Non-overlapping

Le principe est de diviser le domaine de calculs en sous-ensembles dans lesquels on va appliquer une méthode de calcul, puis rassembler nos résultats à l'issue des méthodes pour obtenir un résultat global.

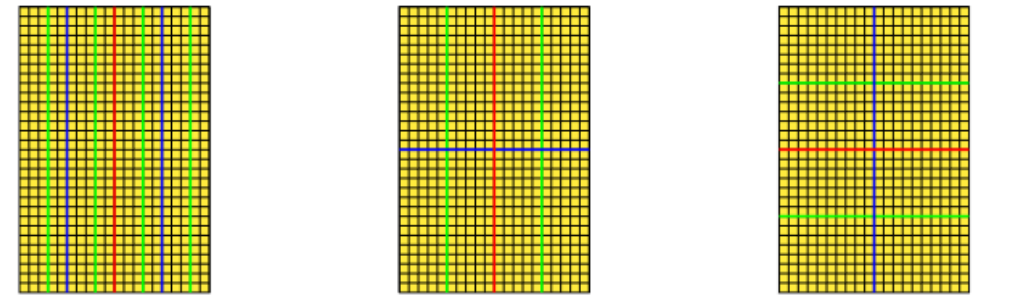
Méthode sans recouvrement : lorsque chaque élément appartient à un seul et unique sous-domaine (non-overlapping method)

Méthode avec recouvrement : les éléments à la frontière sont contenus par deux (ou plus) sous-domaines (overlapping method)

4.1.2 Méthodes de partage

On peut lister 3 méthodes de partage :

1. Partage unidirectionnel : on partage en deux chaque domaine puis sous-domaine, toujours dans la même direction.
2. Partage alterné : on partage dans un sens, puis dans l'autre, puis dans le premier sens, puis dans le second, etc...
3. Partage dans la direction adaptée : on minimise les frontières en partageant là où il y a le moins de cellules aux frontières. C'est le plus optimal en cas de parallélisation.



4.2 Méthodes pour domaines avec recouvrements

Les **ghost cells** : Soient P1 et P2 deux processus qui se partagent des domaines voisins. P1 pourrait avoir besoin des cellules calculées à l'itération précédente par P2 dans son domaine, notamment pour des phénomènes physiques tels que la propagation. Les cellules à la frontière doivent

être partagées par les 2 processus ou par un maître P0. P1 et P2 doivent donc s'attendre avant de faire des lectures mémoire.

On parle de la : **Méthode de Schwarz**. L'algorithme de Schwarz (décomposer les domaines en sous-domaines de manière itérative puis résoudre) converge, grâce à certains théorèmes mathématiques.

4.3 Méthodes pour domaines sans recouvrement

Non-overlapping domain : une partition mathématique (union recouvre le domaine, intersections deux à deux nulles)

Primal schur factorization : chaque processus se retrouve sur une partie de la partition. L'interface entre deux domaines est vue comme un troisième sous-ensemble. Comme il n'y a pas d'éléments commun au domaine 1 et au domaine 2, on peut mettre les noeuds sous la forme d'une matrice "arrowhead".

$$\begin{matrix} A_{11} & 0 & A_{13} \\ 0 & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{matrix}$$

On peut factoriser cette matrice. En parallèle, on inverse le système linéaire PENDANT qu'on met en place la factorisation.

4.4 Généralisation à plus de processus

On a vu comment procéder pour une seule interface entre 2 sous-domaines. De manière récursive, on peut séparer chacun de ces sous-domaines en deux, et ainsi de suite, afin d'obtenir autant de sous-domaines que l'on souhaite.

On appelle cela la **bissection emboîtée** : nested bissection.

Les emplacements des zéros sont très importants : ils représentent des données que l'on ne va pas avoir à stocker. Néanmoins, il faut faire attention à l'accélération, le speed up. Au plus on découpe, au moins il y a d'opérations à faire. Certains processus vont prendre moins de temps que d'autres. On peut se retrouver avec plus de processus que de processeurs (?) ce qui est un problème.

Approche algébrique de FETI : pas à l'examen, c'est plus pour la culture générale