UNIVERSITÉ DE
SHERBROOKE

Département d'informatique
Faculté des sciences

**IFT 630 - Processus concurrents et parallélisme**

---

# Chapitre 6

## Calcul parallèle

---

Gabriel Girard[1]

Sherbrooke

14 février 2023

[1] Gabriel.Girard@usherooke.ca

# Table des matières

# Chapitre 6

# Calcul parallèle

*Ce chapitre est inspiré du manuel de Andrews. Les documents suivants ont aussi été consultés [6]*

Comme nous l'avons déjà mentionné, la programmation parallèle présente un certain nombre de défis, absents de la programmation séquentielle tels que :

- Le non déterminisme ;
  Traiter avec le fait qu'il est potentiellement impossible de connaître les vitesses relatives d'exécutions des programmes parallèles s'avère parfois complexe au niveau du développement d'une solution et de sa mise au point.
- la communication, la synchronisation et l'interblocage ;
  Instaurer une communication efficace tout en minimisant les coûts qu'elle entraîne présente un certain défi. Cela peut impliquer, par exemple, des regroupements de tâches ou une planification plus complexes.
- Le partitionnement et la distribution des données ;
- Le balancement de la charge ;
  Il s'avère parfois complexe de bien répartir la charge de travail entre tous les nœuds de calcul. Cette planification doit se baser sur les capacités de chaque nœud (puissance du processeur, quantité de mémoire, ...), les besoins en communication et autres considérations. En particulier, il est important dans certaines situations d'analyser la nécessité de regrouper certaines tâches.
- La tolérance aux fautes ;
- L'hétérogénéité ;
- La mémoire partagée et distribuée ;
- Les conditions de course (race condition).

Pour produire un programme parallèle, il existe deux approches : l'approche explicite et l'approche implicite. Selon l'approche implicite, le compilateur détecte le parallélisme potentiel dans un programme et génère automatiquement les programmes parallèles. L'approche explicite, dans laquelle le concepteur se charge de déterminer les tâches à exécuter en parallèle, est celle la plus couramment utilisée.

Le but de ce chapitre est de présenter des outils aux personnes cherchant à développer des programmes parallèles. Par programmation parallèle, nous couvrons les domaines suivants :

- la programmation concurrente (mono-processeur) ;

  Ce modèle est courant sur les systèmes à temps partagé sur ordinateur mono-processeur (ce qui n'est plus très courant). Cela entraîne de la programmation multi-fils ou multi-tâches et une communication par mémoire commune ou par messages.
- la programmation concurrente (multiprocesseurs) ;

  Ce modèle est une extension du premier dans lequel on ajoute plusieurs processeurs partageant la même mémoire centrale (multiprocesseurs et multi-cœurs). Ce modèle requiert des outils de synchronisation plus complexes (des constructions matérielles sont souvent requises).
- la programmation concurrente (réseau).
  - Applications parallèles et distribuées ;

    Dans ce modèle, les programmes s'exécutent sur plusieurs ordinateurs connectés par un réseau de communication. La communication entre les différents processus est basée entièrement sur le passage de messages (il n'y a aucune mémoire commune). Voici des exemples de telles applications :

    * Les serveurs de fichiers ;
    * Les serveurs Web et FTP ;
    * Les bases de données dans les banques ou les lignes aériennes ;
    * Les systèmes tolérants aux fautes (système «*non-stop*», téléphonie, ...).

    Ce type d'applications sert principalement à répartir la charge de travail, accéder à des données (ou ressources) éloignées ou distribuées et augmenter globalement la vitesse et la fiabilité du système.
  - Calcul parallèle.

    L'objectif du calcul parallèle est de résoudre des problèmes complexes beaucoup plus rapidement grâce à des grappes d'ordinateurs. Voici des exemples où ce modèle de calcul est appliqué :

    * Le calcul scientifique qui modélise ou simule des phénomènes tels que le climat, l'évolution du système solaire, les effets de nouvelles drogues, le comportement des gènes (bioinformatique), etc.
    * Le graphisme et le traitement d'images qui sert en particulier dans les jeux vidéos et pour les effets spéciaux.
    * Les problèmes d'optimisation et de combinatoire qui permettent en particulier la modélisation d'un système économique ou la planification de horaires de vol d'une compagnie aérienne.

Peu importe le type de programmation recherché, il est nécessaire d'avoir une méthode ou des modèles pour nous guider dans notre conception. De même, il est aussi pertinent d'avoir des outils pour nous aider dans cette tâche. Les prochaines sections présentent une série d'approches et modèles pour la conception d'algorithmes ainsi que des outils pour leur mise en œuvre.

## 6.1 Conception d'algorithmes

La traduction de la solution à un problème (sa spécification) en un algorithme parallèle est un défi en soi. Lors des cours précédant (en particulier «algorithmique»), vous avez appris plusieurs méthodes permettant de concevoir des algorithmes séquentiels. Vous avez également pu constater qu'il existe souvent plusieurs algorithmes pour résoudre le même problème (par exemple les algorithmes de tri).

La conception d'algorithmes parallèles ajoute une toute autre dimension à l'exercice, ce qui rend difficile l'utilisation des approches abordées dans le cours d'algorithmique (mais pas impossible). En particulier, la traduction des algorithmes séquentiels existants en une version parallèle n'est souvent pas une solution. En effet, comme en séquentiel, il existe souvent plusieurs solutions parallèles à un problème et les meilleurs d'entre elles ne correspondent généralement pas aux meilleures solutions séquentielles.

En fait, il existe peu de recettes pour la conception de tels algorithmes et la seule solution est fréquemment de faire appel à notre esprit créatif. Toutefois, afin de guider cette démarche créative, plusieurs auteurs [14, 3, 22, 8] ont tout de même proposé certaines approches qui permettent de mieux cerner le problème et d'évaluer les solutions. Ces approches divisent généralement la conception d'algorithmes est plusieurs étapes (inspirées de [14, 3] ) :

1. le partitionnement ou modèle algorithmique ;
2. la communication ou modèle de communication ;
3. Le modèle de synchronisation ;
4. Le regroupement ;
5. La mise en correspondance et les modèles architecturaux (mapping).
   Les modèles architecturaux ne sont pas vraiment le sujet du cours et ne seront pas abordés en profondeur.

### 6.1.1 Le partitionnement ou modèle algorithmique

Lors de cette étape, on décompose les données et les tâches d'un problème afin d'obtenir de multiples sous-tâches plus simples et plus rapides à traiter. Les éléments plus pratiques tels que le nombre de processeurs sont ignorés à cette étape. On tente seulement d'identifier toutes les opportunités de parallélisme.

Malgré la grande variété d'applications parallèles et réparties, le développement de celles-ci est soutenu par un nombre restreint de méthodes de partitionnement, que l'on appelle aussi modèles algorithmiques. En fait, il y a seulement deux grandes catégories d'algorithmes ou programmes parallèles (ou méthodes de partitionnement) qui sont couramment utilisés :

- la parallélisation par les données (data parallel programs ou domain decomposition)
- la parallélisation par les fonctions (task parallel programs ou functional decomposition)

Il est potentiellement possible aussi de combiner les deux approches.

#### Parallélisation selon les données

La parallélisation selon les données est une pratique courante dans le calcul parallèle. En fait c'est le modèle le plus utilisé. On le retrouve également dans plusieurs applications distribuées.

Ce modèle est tellement populaire qu'il a été subdivisé en plusieurs sous-modèles pour mieux correspondre aux besoins. Parmi ceux-ci on retrouve :

1. La parallélisation triviale («embarassingly parallel») ;

   La parallélisation triviale concerne les applications dans lesquelles on divise les données en sous-groupes afin que les traitements soient entièrement indépendants. Ces applications fonctionnent selon le principe d'administrateur/travailleurs. L'administrateur sépare les données, les distribue sous forme de tâches aux travailleurs, pour ensuite récolter les résultats tel qu'illustré à la figure 6.1.

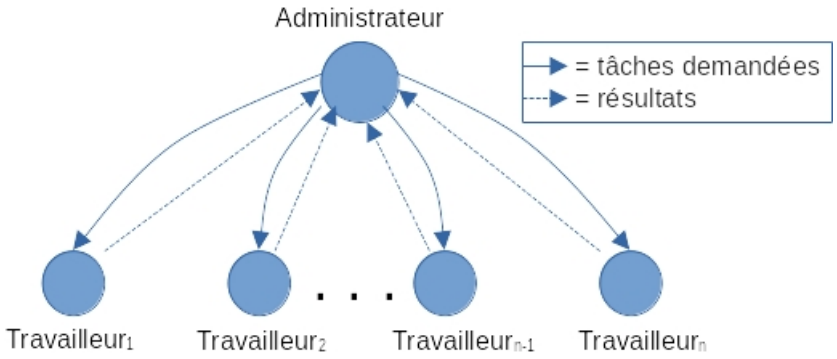



**Figure 6.1** – Relation administrateur-travailleurs

   Voici divers domaines où la parallélisation triviale s'applique régulièrement :

   - Briser les mots de passe ;
   - Seti@home, folding@home, etc ;
   - La multiplication de matrices ;
   - La recherche de modèle dans ARN et protéines
   - Le calcul de $\pi$ (voir annexe A) ;
   - La construction «Map».
     Cette construction applique un même traitement à tous les éléments d'une «liste» (potentiellement en parallèle). Le principe de fonctionnement est le suivant :

     Map(fonction, liste)

     En mode séquentiel, cette fonctionnalité est apparue très tôt dans les langages fonctionnels. Elle est aujourd'hui présente dans plusieurs langages communs dont Python. Des exemples de langages ou environnements qui supportent un «Map» parallèle :
     - OpenMP et Cilk (Boucle parallèle)
     - OpenCL et Cuda (kernel)

     Cette fonctionnalité est souvent utilisée avec une autre construction : «Reduce».
     Pour plus de détails sur la construction «Map/Reduce», veuillez consulter l'annexe B.

2. La parallélisation itérative ;

   La parallélisation itérative est analogue au mode administrateur/travailleurs sauf qu'ici, les traitements ne sont pas complètement indépendants. Les travailleurs, par l'intermédiaire de l'administrateur ou non, s'échangent de l'information régulièrement pour effectuer un travail par étapes.

## Le modèle administrateur/travailleurs

Le modèle administrateur-travailleurs est l'un des modèles les plus courants en calcul parallèle. Il est relativement simple et permet de bien répartir la charge de travail entre les différents travailleurs (cœurs d'un ordinateur ou processeurs sur le réseau). On y refère aussi sous l'expression «*bag of tasks*».

La planification des tâches dans ce modèle adopte un des deux cas de figures suivants :

- si le nombre de processeurs $\geq$ nombre de tâches
  Dans ce cas, on effectue une planification statique qui consiste à assigner les tâches au début et de ne plus rien changer par la suite.
- si le nombre de processeurs $<$ nombre de tâches
  Dans ce cas, on doit faire une planification dynamique, soit considérer que le nombre de tâches est inconnu et les attribuer dès qu'un travailleur devient disponible.

Pour que ce modèle fonctionne bien, il est important de prévoir les problèmes potentiels suivants :

- Que faire si un travailleur tombe en panne ?
  D'abord il faut pouvoir détecter la panne. Ensuite, il faut prendre action en reconfigurant notre système ou en réaffectant la tâche à un autre site. À la reprise, il est possible qu'il y ait une duplication des tâches. Une reconfiguration est de nouveau nécessaire.
- Que faire si l'administrateur tombe en panne ?
  Dans ce cas, le travail effectué est perdu. Il sera parfois nécessaire de reconfigurer le système pour nommer un nouvel administratuer. Dans ce cas, une «élection» est nécessaire.
- Comment traiter un goulot d'étranglement ?
  L'administrateur peut rapidement devenir un goulot d'étranglement si le nombre de tâches et de travailleurs à gérer devient trop grand.

Les avantages de ce modèle sont :

- d'offrir un balancement de charge facile et efficace ;
- d'assurer une répartition du travail relativement équivalente entre les processeurs.

Il est à noter que le modèle administrateur/travailleurs est en fait la contrepartie (à l'opposé fonctionnellement parlant) du modèle pair-à-pair (peer-to-peer).

Dans ce modèle, les processus sont des programmes itératifs qui communiquent et se synchronisent à la fin de chaque étape.

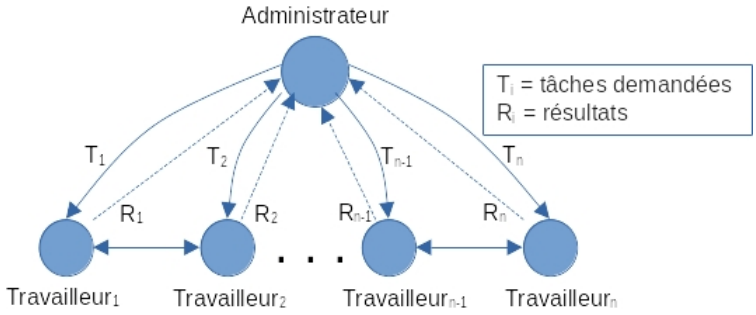



**Figure 6.2** – Parallélisation itérative

3. La parallélisation récursive (Diviser pour régner)

   La parallélisation récursive applique le principe de diviser pour régner que vous avez déjà abordé dans le cours d'algorithmique. Il sied bien aux programmes qui divisent les données par récursivité et qui produisent des appels récursifs indépendants. Le tri rapide, le tri fusion, la recherche dichotomique, les jeux d'échec (*backtracking*) et le calcul de l'aire sous la courbe par approximation (adaptative quadrature) sont des exemples d'algorithmes sur lesquels ce principe s'applique très bien.

   La figure 6.3 illustre ce concept.

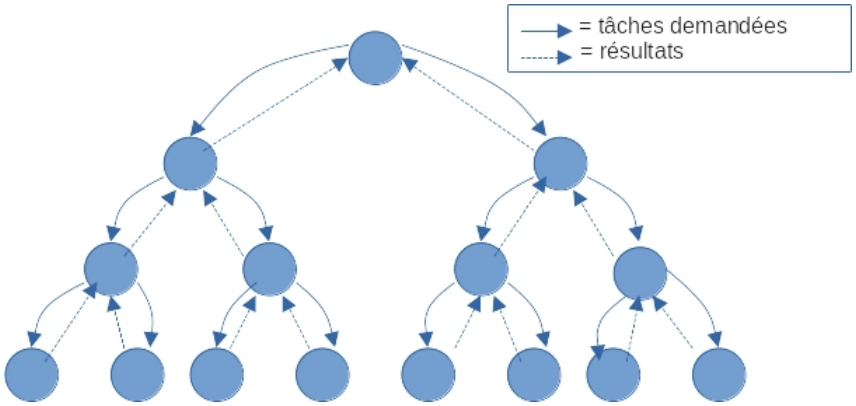



**Figure 6.3** – Parallélisation récursive

4. La parallélisation pair-à-pair (*peer-2-peer*)

   La parallélisation pair-à-pair, illustré à la figure 6.4, est un mode de décomposition analogue au parallélisme itératif mais sans la relation Administrateur/Travailleurs. Il sert principalement à la prise de décision décentralisée telle qu'on la rencontre dans les sémaphores distribués, les horloges logiques et les élections.

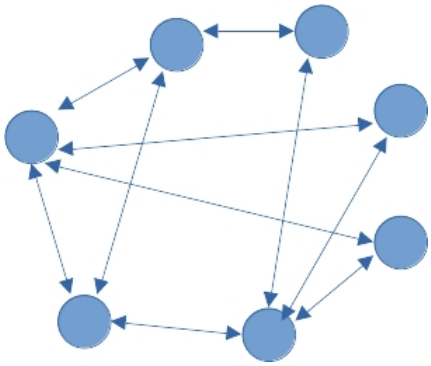


**Figure 6.4** – Parallélisation pair-à-pair

Il est important de noter que la communication ne se fait pas nécessairement avec les voisins immédiats. Nous verrons plus loin des modèles de communication adaptés à ce mode de parallélisation.

### Parallélisation par les fonctions

Les applications classées dans la catégorie dites parallélisation par les fonctions ne suivent aucun modèle algorithmique particulier. Tout dépend du problème à résoudre. Selon ce modèle, différents calculs se feront en parallèle sur les mêmes données ou sur des données différentes. Les calculs peuvent être de la même «durée» ou de durées différentes. Le degré de parallélisme que l'on peut obtenir avec cette approche est généralement modeste.

Il existe plusieurs situations où ce mode de parallélisation est appliquée, principalement sur les systèmes répartis. L'exemple le plus courant est une relation clients/serveur dans laquelle les clients sont fonctionnellement très différents du serveur.

Ce type de parallélisation est principalement caractérisé par le modèle de communication car chacune des tâches doit échanger de l'information avec les autres.

Paralléliser un système selon ce mode s'avère parfois complexe dû à l'absence de modèle adéquat.

### Autres approches

Il existe d'autres classifications pour cette décomposition. Certains auteurs mentionnent :

- décomposition récursive, selon les données, exploratoire ou spéculative [21, 22].
- diviser-pour-régner, techniques sur les pointeurs parallèles ou l'approche aléatoire [8]
- les approches séquentielles standards [55] ;
  Le site TutorialPoint de son côté propose d'adapter les approches standards (vue pour la plupart dans le cours d'algorithmique) pour la conception d'algorithmes, sans toutefois donner de précision sur le «comment». Les approches suggérées sont :
  - diviser pour régner ;
  - l'approche gloutonne ;
    Selon cette approche, la meilleure solution est choisi à tout moment.
  - la programmation dynamique ;

Cette approche ressemble à l'approche diviser pour régner sauf qu'elle réutilise la solution à un sous-problème à plusieurs reprises. L'algorithme pour trouver les séries de Fibonnaci est un exemple d'algorithme de programmation dynamique.
- l'approche avec retour en arrière (Backtracking Algorithm) ;
- la méthode par séparation et évaluation (Branch and Bound)
- la programmation linéaire.

### 6.1.2 La communication ou le modèle de communication

Dans une application parallèle, la communication entre les différents processus est primordiale pour accomplir une tâche. Selon le modèle introduit par Foster [14], il y a plusieurs aspects à considérer pour le communication tels que :

- la localité ou la globalité ;
  Si les communications sont locales, une tâche communique avec un petit sous-ensemble des tâches (généralement ses voisins). Si les communications sont globales la tâches communiquent avec de multiples autres tâches.
- la structure ;
  Si les communications sont «structurées», les messages suivent un «architecture» fixe tels un arbre ou une grille. Autrement, les communications peuvent suivre un structure de graphe arbitraire.
- l'aspect statique ou dynamique ;
  Si les communications sont statiques, les correspondants d'une tâche sont connus à l'avance et ne changent jamais. Si elles sont dynamiques, les correspondants changent selon le calcul effectué.
- l'aspect synchrone ou asynchrone ;
  Nous avons déjà abordé les aspects synchrones ou non des communications.

En respectant ou incorporant tous ces aspects, plusieurs modèles de communication ont été développés pour convenir aux différentes applications à traiter ou au modèle algorithmique utilisé. Dans cette section, les schémas les plus populaires sont présentés.

#### Clients/serveur

Le modèle de communication clients/serveur convient bien aux applications réparties basées sur une parallélisation selon les fonctions. La figure 6.5 illustre ce mode de communication.

#### Administrateur/Travailleurs

Le modèle de communication administrateur/travailleurs, correspondant au modèle de calcul du même nom, est appliqué principalement au calcul parallèle. Comme il est possible de le constater à la figure 6.6, c'est la relation inverse du modèle clients/serveur.

#### Diffusion

Le modèle de communication dit «par diffusion» implique que tous les processus envoient leurs messages à tous les autres processus, comme l'illustre la figure 6.7. Ce modèle est pratique autant pour la parallélisation selon les données que selon les fonctions. Il est cependant le modèle de choix

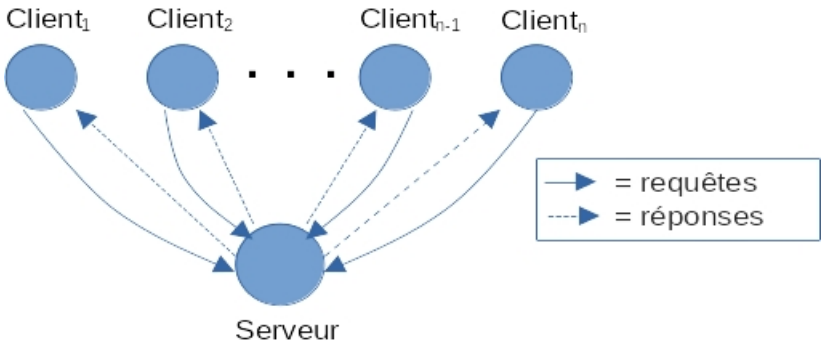



**Figure 6.5** – Modèle de communication clients/serveur

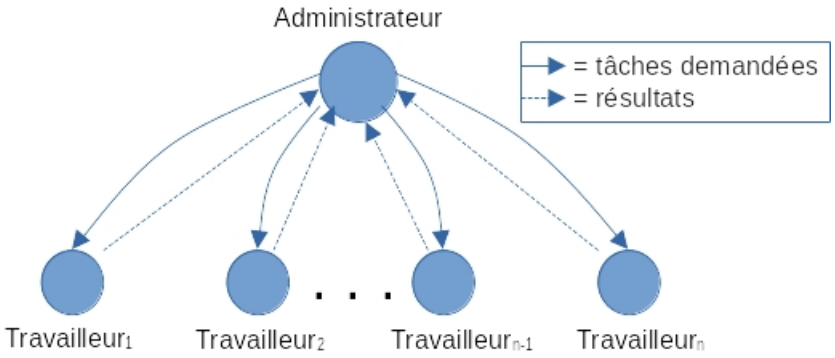



**Figure 6.6** – Modèle de communication Administrateur/Travailleurs

dans les applications réparties telles que les algorithmes de prise de décisions réparties (horloge logique, élection, sémaphore distribué).

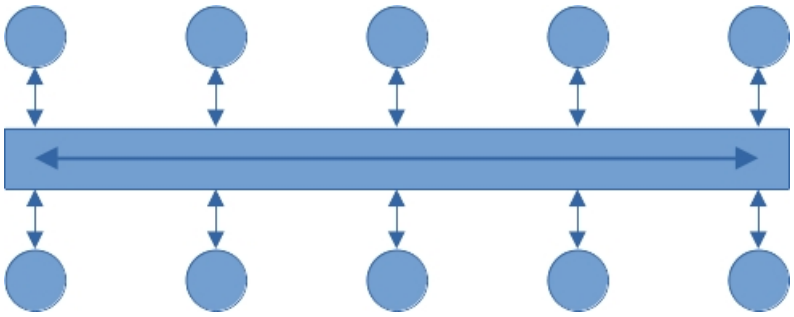


**Figure 6.7** – Modèle de communication par diffusion

### Anneau/jeton

Le modèle de communication par «anneau et jeton» suppose que tous les processus sont connectés par un réseau de communication sous forme d'un anneau. Un jeton unique circule en permanence sur l'anneau. La possession du jeton autorise le détenteur à entreprendre des actions dont l'envoi de messages. Le jeton est essentiel au bon fonctionnement de ce système. Il faut donc prévoir des mécanismes pour régénérer le jeton si celui-ci est perdu ou éliminer un jeton si jamais plus d'un circulent sur l'anneau.

Tous les messages circulent sur l'anneau de la source vers la ou les destinations. Cette organisation permet donc de fournir autant la communication un-à-un que la diffusion. Ainsi, tout comme le modèle de communication par diffusion, il est bien adapté à la parallélisation selon les données que selon les fonctions. Il est aussi un modèle de choix dans les applications réparties telles que les algorithmes de prise de décisions réparties (horloge logique, élection, sémaphore distribué).

### Pipeline

Dans un modèle de communication de type pipeline, la communication se fait avec les deux voisins immédiats et, par la suite, le message circule d'un voisin à l'autre jusqu'à sa destination finale. La figure 6.9 illustre ce mode de communication.

Ce modèle est exploité autant en parallélisation selon les données que selon les fonctions. Il permet d'implanter des applications réparties, concurrentes et parallèles. Les architectures systoliques (parallélisation selon les données), les suites de commandes dans Unix (parallélisation selon les fonctions) ou les suites de tâches en traitement par lots constituent les applications ou architectures les mieux adaptées à ce modèle.

### Battements de cœurs/commérages/vagues

Ces trois modes de communication sont similaires et permettent principalement la diffusion de l'information dans un réseau. Celle-ci se propage à partir d'un nœud par l'intermédiaire des voisins immédiats (dissémination ou exploration progressive de l'horizon). C'est une forme de diffusion

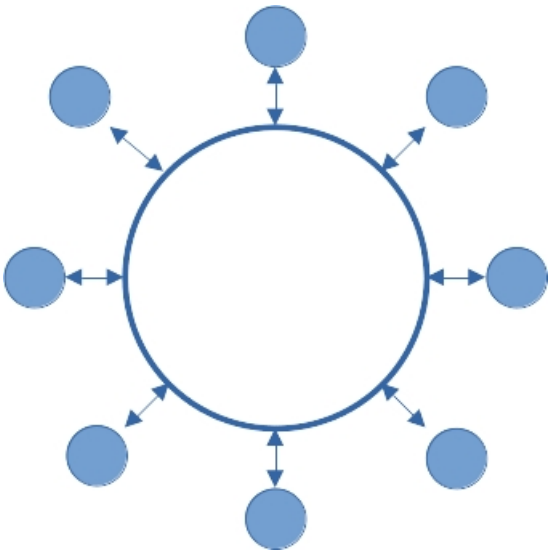


**Figure 6.8** – Modèle de communication par anneau/jeton

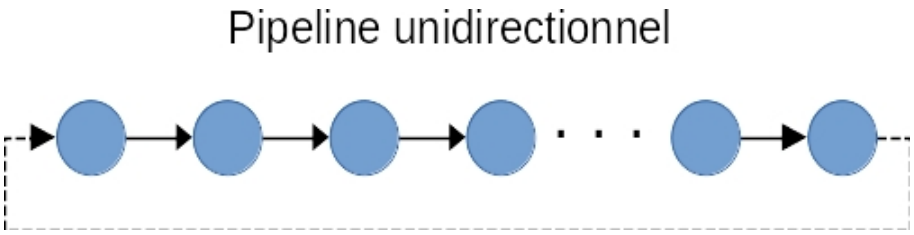
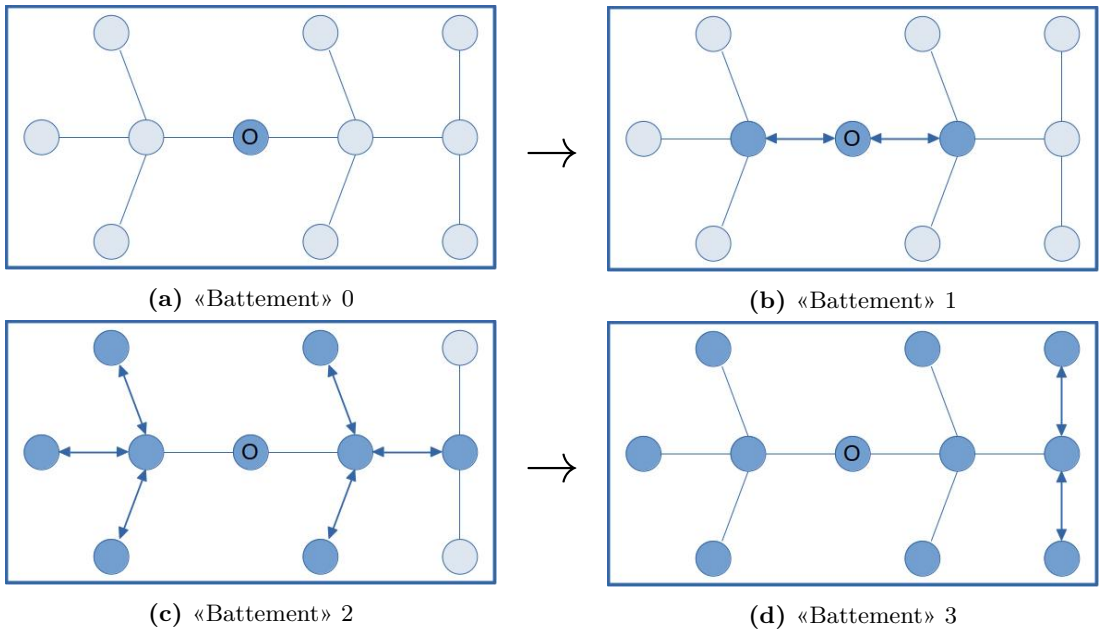



**Figure 6.9** – Modèle de communication de type pipeline

qui procède par étapes dans laquelle l'information est envoyée et reçue périodiquement, tel que décrit par le programme 6.1. La figure 6.10 illustre comment l'information est disséminée à tous les membres du réseau par l'intermédiaire des voisins.

```
1           loop
2                envoi msg à tous les voisins
3                reçoit msg de tous les voisins
4                mise à jour
```

**Programme 6.1** – Algorithme de type battement de coeur




**(a)** «Battement» 0 **(b)** «Battement» 1

**(c)** «Battement» 2 **(d)** «Battement» 3

**Figure 6.10** – Modèle de communication de type battements de cœurs

Ces approches implantent une forme de barrière. Celles-ci sont privilégiées lorsque les processus sont dans une relation pair-à-pair (peer-2-peer). Pour plus d'informations sur ces types de communication, consultez l'annexe C.

### Sondes/échos (Probes/echos)

Plusieurs applications et algorithmes séquentiels ont recours aux arbres et aux graphes pour traiter l'information. Le protocole dit par sondes et échos («probes/echos») propage l'information d'une façon similaire au protocole par battements de cœur, mais il est adapté aux topologies en arbre ou en graphe.

Une sonde (probe) est un message envoyé par un nœud à ses successeurs (voisins) dans l'arbre ou dans le graphe. Les messages sont envoyés en parallèle à tous les successeurs. Un «écho» est la réponse à cette sonde.

**Autres**

Plusieurs architectures ou applications particulières ont donné naissance à des modes de communication adaptés.

- Serveurs répliqués

  Pour rendre un environnement plus rapide et plus fiable, de nombreuses situations nécessitent que plusieurs serveurs soient chargés de fournir le même service. Dans ces cas, les serveurs doivent communiquer efficacement entre eux pour éviter les incohérences. Les modèles de communication généralement employés alors sont dérivés du client/serveur jusqu'au protocole par commérages.

- Maillage en réseau, grille et grille torique (*Mesh* et *Torus*)

  Les modèles par maillage et grille [66, 68, 67, 49, 26, 23, 18, 42] sont particuliers car ils peuvent autant servir de modèle de calcul (algorithmique), de modèle de communication que de modèle architectural. Ils sont très similaires au pipeline ou plutôt à une extension de ce dernier à de multiples dimensions.

  Il est possible d'imaginer que dans un environnement spécifique tous les processus soient connectés à tous les autres processus dans le système comme l'illustre la figure 6.11.Toutefois dû au nombre excessif de connexions que ce modèle génère, des dérivés, appelées réseaux maillés (*Mesh*) et exigeant moins d'interconnexions, ont été développés.

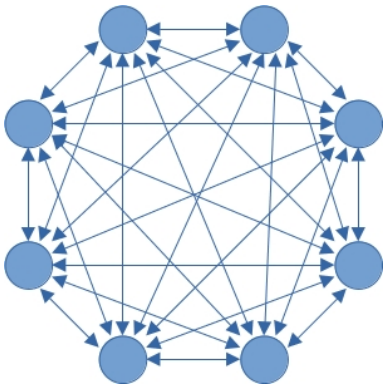


**Figure 6.11** – Réseau de processus entièrement connecté

Le terme *Mesh* [66, 68], que j'appellerai maillage réseau ou réseau maillé, est d'usage dans le monde des réseaux physiques (filaire ou sans fil). Il identifie une topologie dans laquelle chacun des nœuds se connecte dynamiquement et directement au plus grand nombre de nœuds possibles sans structure ni hiérarchie particulière. Comme les connexions sont établies dynamiquement, une réorganisation du réseau en cas de pannes ou d'ajouts est simplifiée.

Cette topologie s'approche d'un réseau complètement connecté sans toutefois nécessiter autant de connexions. Il est donc possible que la transmission des messages requiert un routage de la source vers la destination. Cette topologie est particulièrement bien adaptée et optimisée pour les réseaux sans fil.

Dans le monde du calcul parallèle, le terme «mesh» identifie plutôt un réseau maillé «rectilinéaire» de dimension $N$ ($N = 1, 2, 3, ...$) dans lequel chacun des processus ou nœuds est connecté seulement à ses voisin immédiats.

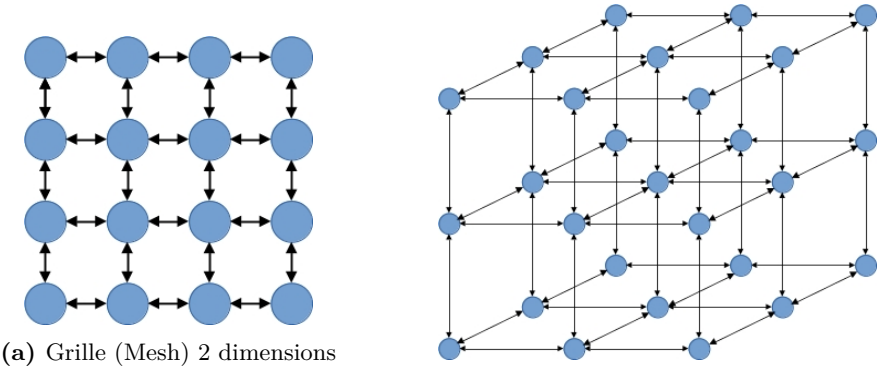
Le réseau linéaire est le modèle de réseau maillé le plus simple. Il est équivalent à un pipeline bidirectionnel. Si l'on interconnecte les processus aux extrémités du réseau, on obtient une grille torique à une dimension (1D). Ces deux organisations sont illustrés à la figure 6.12

**(a)** Réseau linéaire

**(b)** Grille torique à une dimension

**Figure 6.12** – Modèle de communication de type Mesh

Le modèle *mesh* en calcul parallèle [23, 18, 42, 67, 49, 26], que j'identifie comme une grille, défini un réseau linéaire à multiples dimensions. Un nœud communique donc avec ses voisins immédiats dont le nombre est déterminé par le nombre de dimensions. Ainsi pour un grille 2D, chaque nœud possède au maximum quatre voisins, tandis que pour une grille 3D, chaque nœud possède au maximum six voisins. La figure 6.13 illustre la structure d'une grille. En reliant les nœuds aux extrémités de la grilles, on obtient des grilles toriques en 2D ou 3D [67, 49, 26]. Comme l'illustre la figure 6.14, dans des grilles toriques, le nombre de voisins de chaque nœud est identique pour tous, soit 4 et 6 respectivement.



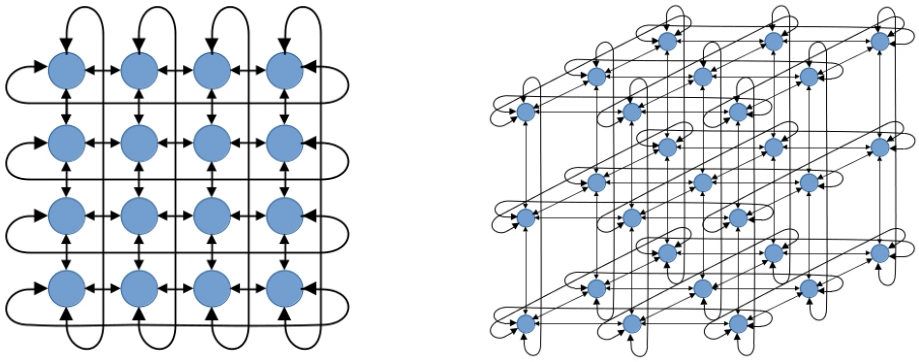
**(a)** Grille (Mesh) 2 dimensions

**(b)** Grille (Mesh) 3 dimension

**Figure 6.13** – Modèle de communication de type Mesh

- Hypercube

  Le modèle de communication par hypercube désigne aussi à la fois un modèle de communication et un modèle de calcul. Ici, le réseau est formé de cubes imbriqués. Dans cette structure, un nœud membre d'un hypercube de dimension D aura D voisins. La figure 6.15 affiche la structure d'un hypercube de dimension 4.



**(a)** Grille torique à 2 dimensions

**(b)** Grille torique à 3 dimension

**Figure 6.14** – Modèle de communication de type Mesh

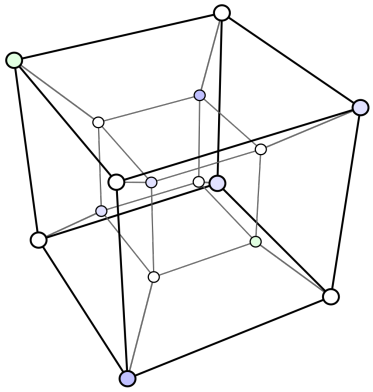


**Figure 6.15** – Modèle de communication de type Hypercube (tiré de [63])

#### Conclusion

Le modèle de communication peut être défini par un modèle abstrait afin de concevoir un algorithme parallèle. Si c'est le cas, lors de l'implantation, il est nécessaire projeter ce modèle de communication sur le modèle architectural. Cette situation implique un travail de planification afin de minimiser les coûts de communication.

### 6.1.3 Modèle de synchronisation

Dans toutes les situations de calculs, il est courant que les différentes tâches doivent se synchroniser. Pour ce faire, deux modèles de synchronisation sont disponibles : les algorithmes synchrones et asynchrones.

#### Les algorithmes synchrones

Les algorithmes parallèles synchrones sont requis pour implanter des méthodes de calcul itératives. Fréquemment, en mathématiques, la recherche d'une solution est effectuée en utilisant des techniques d'analyse numérique, i.e. par approximations successives qui convergent vers une solution. Cette recherche itérative s'applique jusqu'à l'obtention de la réponse finale ou de la précision recherchée.

En général, plusieurs processus sont affectés à ce calcul, mais la tâche est décomposée de telle façon à ce que chacun d'eux travaille sur une sous-tâche distincte. Ces sous-tâches sont également divisés en étapes à la fin desquelles tous les processus doivent se synchroniser. La synchronisation à la fin de chaque étape est appelée une barrière. La figure 6.16 illustre le fonctionnement de ces algorithmes.

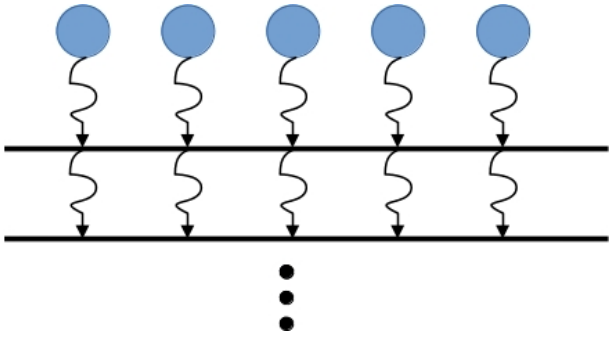


**Figure 6.16** – Mode de fonctionnement d'un algorithme synchrone

Le programme 6.2 expose la structure générale de ces algorithmes. Quant à lui, le programme 6.3 présente une approche différente mais non conseillée car totalement inefficace. En effet, à la fin chaque étape, tous les processus sont détruits et recréés au début de la suivante.

#### Les barrières

Les barrières constituent un type de synchronisation très particulier car elles permettent de synchroniser l'exécution d'un groupe de processus. Les implantations possibles des barrières se

```
1 Process travailleur[i:= 1 to N]
2 {
3    while (true)
4    {
5       Code à exécuter pour la tâche «i»
6       Barrière -----> Attendre que les N processus aient complété l'étape
7    }
8 }
```

**Programme 6.2** – Structure générale d'un algorithme synchrone

```
1 while (true)
2 {
3    co [ i:=1 to N]
4        code à exécuter pour la tâche i
5    oc // <--------- Barrière
6 }
```

**Programme 6.3** – Structure générale d'un algorithme synchrone (inefficace)

basent sur :

- les variables communes ou le passage de messages ;
- la présence ou non d'un coordonnateur (approche symétrique de type peer-2-peer ou asymétrique) ;
- d'autres structures de communication (arbre, ...).

Le programme 6.4 présente une première solution basée sur des variables communes, tandis que le programme 6.5 en implante une deuxième basée sur le passage de messages.

```
1 while(true)
2 begin
3   code
4   --- cpt++
5   --- attendre (cpt =n)
6 end
```

**Programme 6.4** – Implantation d'une barrière basée sur les variables communes

```
1 while(true)
2 begin
3   code
4   --- envoi msg à tous
5   --- reçoit msg de tous
6 end
```

**Programme 6.5** – Implantation d'une barrière basée sur le passage de messages

### Les algorithmes asynchrones

Avec ce type d'algorithme, il n'y a aucune synchronisation entre les processus, chacun progressant à sa vitesse. Il n'y a pas de synchronisation de fin d'étapes. De tels algorithmes s'avèrent

fréquemment complexes à concevoir.

Ainsi, les algorithmes de simulation à événements discrets sont généralement implantés avec des algorithmes asynchrones.

### 6.1.4 Le regroupement et les modèles architecturaux

Dans les étapes précédentes, la traitement a été découpé en sous-tâches abstraites et un modèle de communication a été défini. Le résultat est toutefois encore un modèle très abstrait et parfois peu pratique. À cette étape, l'architecture de l'environnement (modèle architectural) sous-jacent entre en ligne de compte (nombre de processeurs, capacité du réseau, ...). Par exemple, il est possible que ce premier partitionnement définisse beaucoup trop de tâches par rapport au nombre de processeurs dans le système. De même, le modèle de communication peut surcharger le réseau s'il n'y est pas bien adapté.

**Exemple : la multiplication de matrices**

Utilisons la multiplication de matrices pour illustrer la nécessité de regroupement.

Le partitionnement le plus simple et évident pour la multiplication de matrices est de créer autant de tĉahes que de produit scalaire (un tâche pour chaque valeur en sortie).

Le but de la présente étape est de regrouper des tâches (si nécessaire) pour s'approcher de quelque chose de plus pratique qui s'adapte à l'architecture de notre environnement. Il faut revisiter le partitionnement pour produire un algorithme qui respecte les capacités de l'environnement et s'exécute efficacement. Si possible, il faut aussi penser à réduire les coûts de communication (ou même à les éliminer).

Présenter en détails les modèles architecturaux sort du cadre de ce cours. Toutefois il est d'intérêt général d'énumérer quelques modèles architecturaux. Rappelons que ce modèle décrit comment les nœuds sont physiquement reliés et comment l'architecture finale peut supporter de processeurs (mise à l'échelle).

Les principaux modèles sont :

1. Les bus ;
   La plupart des environnement multiprocesseurs que nous utilisons sont basées sur cette architecture.
2. L'anneau ;
3. L'hypercube ;
4. Le réseau maillé, la grille ou la grille torique (*Mesh* et *torus*) ;
5. Le réseau de tuile (*tile* [65]) ;
   Ce modèle est similaire au *Mesh* mais il est employé pour relier de multiples cœurs sur une puce ($> 100$). C'est un modèle par couches, chaque couche ressemblant à une grille 2D, afin de permettre une expansion. Ce modèle possède une très bonne capacité de mise à l'échelle.
6. Autres.
   Dans les réseaux filaires, on retrouve plusieurs autres architectures dont celles en arbre et en étoile.

**Exemple : la multiplication de matrice**

Le partitionnement fait pour la multiplication de matrices présente au moins deux grands problèmes :

1. Elle produit trop de tâches ;
   Ainsi, si la matrice devient grande (1000 x 1000), cela produira 1000000 tâches. Si votre environnement ne possède pas un million de processeurs, des regroupements sont nécessaires. Un premier regroupement pourrait être que chaque tâche effectue plusieurs produits scalaires afin de calculer un ligne complète en sortie. Cela produira 1000 tâches. Encore une fois, si votre environnement ne possède pas 1000 processeurs, d'autre regroupements seront nécessaires.
2. Elle produit des tâches trop simples ;
   Le temps d'exécution d'un simple produit scalaire s'avère généralement trop court par rapport aux temps de latence (i.e le temps pris pour démarrer les multiples tâches) et de communication. Le regroupement est alors nécessaire pour produire des tâches dont le temps de calcul est beaucoup plus long que les temps de communication et de latence combinés.

### 6.1.5 La mise en correspondance ou planification

Lors de cette étape, on planifie les tâches dans l'environnement. Ainsi, on détermine sur quel processeur chaque tâche s'exécute.

Le but de cette planification est de minimiser le temps d'exécution et minimiser les communications. Par exemple, placer les tâches concurrentes sur des processeurs différents et les tâches qui communiquent intensément sur le même processeur. C'est à cette étape qu'interviennent les algorithmes effectuant le balancement de la charge.

### 6.1.6 Conclusion

Nous avons présenté dans cette section, une méthode structuré pour aider au développement d'un programme parallèle. Toutefois d'autres méthodes ou classifications ont été proposées. Ainsi, Barney et Frederick [6] définissent des classes différentes mais surtout moins abstraites plutôt basées sur des approches communément utilisées dans les environnements parallèles. Cela ressemble cependant beaucoup plus à des environnements que des modèles de conception. Ainsi, ils proposent la classification suivante :

- modèle à mémoire partagée (sans fils d'exécution) ;
  Selon ce modèle, les processus partagent un espace d'adresses commun qu'ils accèdent en lecture/écriture de façon asynchrone. Divers mécanismes de synchronisation sont utilisés pour contrôler l'accès. L'avantage de ce modèle est que tous les processus ont un accès égal à la mémoire (aucune hiérarchie), ce qui simplifie le développement. Toutefois, le contrôle de la cohérence des données et de la synchronisation peut devenir complexe.
  Ce modèle est disponible nativement sur plusieurs systèmes d'exploitation. Par exemple, UNIX fournit des outils pour partager des segments de données en mémoire (primitives `shmget, shmat, shmctl,` ...). Nous abordons, dans un prochain chapitre, le diverses fonctionnalités qui sont fournies avec la mémoire distribuée.

- modèle basé sur les fils d'exécution ;
  Ce modèle est très similaire au précédent. Seul l'existence de fils d'exécution s'ajoute pour remplacer les processus afin d'alléger différentes phases de création et de communication.
  Ce modèle est disponible dans la plupart des langages de programmation et dans des bibliothèques tels POSIX et OpenMP.
- modèle basé sur la mémoire distribué et le passage de messages ;
  Selon ce modèle, plusieurs processus ou fils s'exécutent soit en partie sur un même site et en partie sur plusieurs. Ils communiquent principalement par l'intermédiaire d'envois et de réceptions de messages sauf potentiellement pour certains fils s'exécutant sur le même site.
  Ce modèle est disponible dans des bibliothèques tels MPI.
- modèle de parallélisation selon les données ;
  Selon ce modèle, aussi appelé PGAS (Partitionned Global Address Space), la grande partie du travail parallèle s'oriente sur l'exécution d'opérations sur un seul ensemble de données. Chaque tâche travaille sur une partie distincte de l'ensemble.
  Ce modèle se retrouve dans certains langages de programmation tels *Coarray Fortran* et *Unified paralllel C* (UPC).
- modèle hybride ;
  Un modèle hybride intègre plusieurs des modèles précédents. Des exemples courants de tels modèles sont l'utilisation conjointe de OpenMP et MPI, de MPI avec un GPU et de MPI avec les fils d'exécution.
- modèle SPMD (Single Program Multiple Data) ;
  Ce modèle est considéré comme plus abstrait que les précédents car il peut être construit sur une combinaison des modèles précédents. Il comprend un programme unique qui opèrent sur des données distinctes.
- modèle MPMD (Multiple Program Multiple Data) ;
  Ce modèle est similaire au SPMD sauf que les processus exécutent des programmes différents.

Comme vous pouvez le constater, la conception de programmes parallèles est un problème complexe et de multiples méthodes ont été proposées pour en supporter le développement. Nous en avons présenté quelques unes mais soyez conscient qu'il en existe plusieurs autres.

## 6.2 Outils de programmation parallèle

Quelque soit l'approche ou le modèle choisi (algorithmique, communication ou synchronisation) et le type de concurrence visé, il est nécessaire de recourir à un outil qui permet et surtout facilite une implantation adéquate. Idéalement, celui-ci doit fournir :

- une transparence architecturale autant au niveau du système que du processeur ;
- une transparence au niveau des communication et du réseau ;
- une facilité d'utilisation ;
- un support pour la tolérance aux fautes (fiabilité) ;
- un support pour l'hétérogénéité ;
- la portabilité ;
- un support pour les langages traditionnels ;
- une augmentation de performance ;
- la transparence au niveau de la concurrence.

### 6.2.1 Exemples

Voici quelques outils conçus pour faciliter le développement d'applications parallèles et distribuées :

- MPI - Message Passing Interface
  Bibliothèque pour faciliter la programmation d'applications sur des grappes de calcul.
- OpenMP
  Bibliothèque pour faciliter la programmation d'applications parallèles sur multi-processeurs.
- MOM (JMS, Active MQ, ...) - *Message Oriented Middleware*
  Bibliothèques pour faciliter la communication entre applications.
- HPF (High Performance Fortran)
- Cuda, openCL
  Bibliothèques pour la programmation d'applications parallèles, principalement sur des cartes graphiques. Cuda fonctionne seulement sur les cartes Nvidia. OpenCL est une norme qui permet de développer des programmes qui fonctionnent sur de multiples environnements.
- PVM - Parallel Virtual Machine
  Bibliothèque pour faciliter la programmation d'applications sur des grappes de calcul. Cet environnement a disparu au profit de MPI.
- Langages : Sisal, PCN, CC++, Mentat, ...
  Langages spécialisés supportant la programmation parallèle.
- C++, C#, Python, Java, ...
  Langages offrant des facilités (principalement le multi-fils) pour le développement d'applications parallèles et réparties.

## 6.3 Applications et algorithmes

### 6.3.1 Introduction

Les limites du séquentiel, le faible coût des systèmes multi-cœurs et multi-processeurs, l'accès facile à des grappes de calcul et à l'internet, toutes ces raisons incitent de plus en plus au développent de solutions concurrentes et ce, pour tous types d'applications.

Voici quelques exemples d'applications parallèles et réparties.

#### Applications réparties

- Serveurs Web (Http) ;
- Serveur de transfert de fichiers (ftp, ftps, sftp) ;
- Systèmes de fichiers fichiers (Samba, NFS, ...) ;
- Prise de décisions ;
- Infonuagique ;
- ...

#### Calcul scientifique

En calcul scientifique, trois techniques fondamentales sont principalement employées :

- le calcul de grilles (grid computing)

Plusieurs problèmes en traitement d'images et en modélisation scientifique peuvent être résolus par des calculs sur des grilles qui sont représentées par des tableaux à plusieurs dimensions (vecteurs, matrices, ...). L'idée de base du calcul sur une grille consiste à employer une matrice de points que l'on projette sur la grille dans une région spatiale.
En traitement d'images, la matrice est initialisée avec des valeurs de pixels, et le but pourrait être, par exemple, de trouver un ensemble de points ayant la même intensité. Dans d'autres situations, on appliquerait un traitement répétitif sur chacun des pixels.
Un exemple de modélisation scientifique est l'application d'une solution numérique à la résolution d'équations différentielles partielles. Ces équations servent à décrire plusieurs phénomènes et sont particulièrement utiles dans le domaine de la prévision météorologique, la simulation de la circulation de l'air sur une aile d'avion, la simulation de la turbulence des fluides, les mathématiques financières, ... Dans ce cas, on projette les données du problème en main sur une matrice sur laquelle on applique un traitement répétitif jusqu'à convergence vers un état stable.
L'équation de Laplace est un exemple d'équation différentielle partielle. Sa solution s'approxime numériquement avec l'itération de Jacobi, Gauss-Seidel et SOR (Successive Over Relaxation). L'algorithme de Jacobi fonctionne de la façon suivante :

  - les bords de la matrice sont initialisés à la valeur cible ;
  - l'intérieur reçoit les valeurs de départ pour le calcul ;
  - on calcule de façon répétitive de nouvelles valeurs pour les valeurs internes (moyenne des voisins) ;
  - on termine après $K$ itérations ou après un test de convergence ($\epsilon$)

- le calcul de particules
  En calcul de particules, on modélise les forces qu'exercent l'une sur l'autre les particules (molécules, planètes)
- Le calcul matriciel
  La calcul matriciel sert à résoudre des systèmes d'équations.

### 6.3.2 Exemple 1 - Somme de n valeurs

Le premier cas que nous allons étudier est le calcul de la somme de $N$ valeurs contenues dans un tableau. C'est un problème qui se solutionne facilement en séquentiel (d'une complexité $\mathcal{O}(n)$). Le programme 6.6, écrit en SR, implante une solution parallèle à ce problème. Cette solution, comme illustrée à la figure 6.17 procède par étapes. À chaque étape, chacun des processus effectue la somme de deux valeurs. Le programme définit donc une fonction, `barriere`, permettant de synchroniser tous les processus à la fin de chaque étape. La somme est calculée après $\mathcal{O}(log(n))$ étapes).

### 6.3.3 Exemple 2 - Multiplication de matrices (calcul de grille)

La multiplication de matrices est un exemple simple de calcul de grille. C'est un problème de parallélisme trivial qui se résout sans la nécessité de fonctionner par étapes. Le programme 6.7 lui implante une solution séquentielle en SR. La solution s'exécute en un temps proportionnel à $\mathcal{O}(n^3)$

Il existe plusieurs solutions parallèles à ce problème. Certaines emploient une relation administrateur/travailleurs (surtout si on utilise la communication par messages) et d'autres sur une

```
1  resource main()
2         const N := 8  # nombre de processus
3         sem continue[N] := ([N] 0)
4         sem termine := 0
5         var x[1:8]: int := (1,2,3,4,5,6,7,8)
6         var nbr : int := 8
7
8  process barriere
9    do true ->
10        fa w := 1 to N -> P(termine) af
11        fa w := 1 to N -> V(continue[w]) af
12   od
13 end
14
15 process calcul(i:= 1 to 8 )
16    fa k := 1 to 3 ->
17       if i <= nbr/(2**k) -> x[i] := x[2*i] + x[2*i-1]
18       fi
19       V(termine); P(continue[i])
20    af
21    if i=1 -> write (i, " ", x[i]) fi
22 end calcul
23 end main
```

**Programme 6.6** – Somme de $n$ valeurs en parallèle

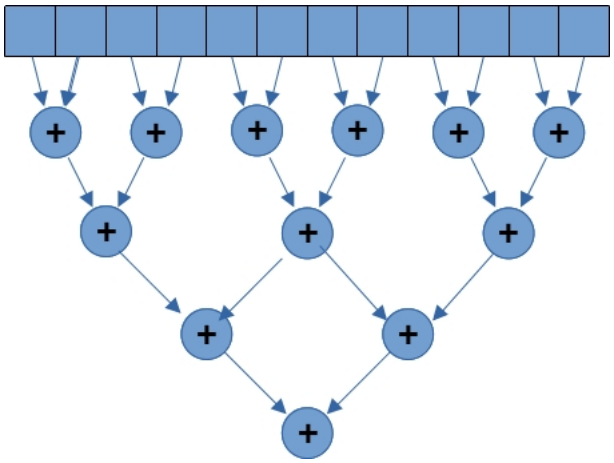


**Figure 6.17** – La somme des éléments d'un tableau en parallèle

```
1  resource matrice()
2     const n := 4
3     var a[1:n,1:n]: int, b[1:n,1:n]: int, c[1:n,1:n]: int
4     # initialisation des matrices
5     fa i := 1 to n ->
6         fa j := 1 to n -> a[i,j]:=1;b[i,j]:=2  af
7     af
8     # Multiplication
9     fa i := 1 to n ->
10       fa j := 1 to n ->
11             c[i,j]:=0
12             fa k := 1 to n ->
13                c[i,j]:= c[i,j] + a[i,k]* b[k,j]
14             af
15        af
16    af
17    # Affichage
18    fa i := 1 to n ->
19         fa j := 1 to n -> printf("% i ,", c[i,j] ) af
20         write()
21    af
22 end matrice
```

**Programme 6.7** – Multiplication de matrices en séquentiel

relation pair-à-pair (avec la mémoire partagée). Les solutions présentées dans cette section sont basées sur une relation pair-à-pair et sur la mémoire partagée. Elles sont toutes inspirés des solutions proposées par Andrews [4].

Le programme 6.8 présente une première version parallèle de la multiplication de matrices. Dans cette solution, $n$ processus sont créés, chacun calculant une ligne de la matrice résultante. Le calcul dans ce cas s'effectue en un temps proportionnel à $\mathcal{O}(n^2)$. Le programme 6.9 implante une solution similaire qui utilise une procédure et la notation «`co ... oc`» permettant d'exécuter des énoncés en parallèle.

Le programme 6.10 propose une autre version parallèle de la multiplication de matrices. Dans cette solution, $n^2$ processus sont créés, chacun d'eux calculant une seule valeur de la matrice résultante (produit scalaire). Le calcul dans ce cas ci est traité en un temps proportionnel à $\mathcal{O}(n)$). Le programme 6.11 fournit une solution équivalent qui fait appel à une procédure et la notation «`co ... oc`» permettant d'exécuter des énoncés en parallèle.

### 6.3.4 Exemple 3 - Tri rapide

Le tri rapide est un algorithme de tri séquentiel qui s'exécute en un temps proportionnel à $\mathcal{O}(n \times log(n))$. La solution parallèle du programme 6.12 démarre un processus pour chaque «appel récursif».

### 6.3.5 Exemple 4 - Sommes partielles

Le calcul de la somme partielle consiste à produire dans chaque case du tableau la somme de tous les éléments qui précèdent l'élément courant. Ainsi, le calcul de la somme partielle du tableau suivant :

| | | | | | |
|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 |

```
1  resource matrice()
2     const n := 4
3     var a[1:n,1:n]: int, b[1:n,1:n]: int, c[1:n,1:n]: int
4     # Initialisation des matrices
5     fa i := 1 to n ->
6          fa j := 1 to n ->
7                a[i,j]:=1; b[i,j]:=2; c[i,j]:=0
8          af
9     af
10    # Multiplication
11    process calcul(i := 1 to n )
12         var j,k:int
13         fa j := 1 to n ->
14             c[i,j]:=0
15             fa k := 1 to n ->
16                 c[i,j]:= c[i,j] + a[i,k]* b[k,j]
17
18             af
19         af
20    end
21    # Affichage
22    final
23    fa i := 1 to n ->
24         fa j := 1 to n -> printf("% i ,", c[i,j] ) af
25         write()
26    af
27   end
28 end matrice
```

**Programme 6.8** – Multiplication de matrices en parallèle

```
1  resource matrice()
2     const n := 4
3     var a[1:n,1:n]: int, b[1:n,1:n]: int, c[1:n,1:n]: int
4     #initialisation de la matrice
5     fa .... af
6
7     procedure scalaire(i: int)
8        var j,k:int
9        fa j := 1 to n ->
10           c[i,j]:=0
11           fa k := 1 to n ->
12                 c[i,j]:= c[i,j] + a[i,k]* b[k,j]
13           af
14       af
15    end
16
17    co (i := 1 to n ) scalaire(i) oc
18
19    fa i := 1 to n ->
20       fa j := 1 to n -> printf("% i ,", c[i,j] ) af
21       write()
22    af
23 end matrice
```

**Programme 6.9** – Multiplication de matrices en parallèle

```
resource matrice()
   const n := 4
   var a[1:n,1:n]: int, b[1:n,1:n]: int, c[1:n,1:n]: int
   # Initialisation des matrices
   fa i := 1 to n ->
        fa j := 1 to n ->
           a[i,j]:=1; b[i,j]:=2; c[i,j]:=0
        af
   af
   # Multiplication
   process calcul(i := 1 to n, j:=1 to n )
      var k:int
      c[i,j]:=0
      fa k := 1 to n -> c[i,j]:= c[i,j] + a[i,k]* b[k,j]
      af
      write("Processus ", i,j, " termine")
   end
   # Affichage du résultat
   final
     fa i := 1 to n ->
        fa j := 1 to n -> printf("% i ,", c[i,j])
        af
        write()
     af
   end
end matrice
```

**Programme 6.10** – Multiplication de matrices en parallèle

```
resource matrice()
  # Lire la taille de la matrice
  var n: int; getarg(1,n)
  # declarer et initialiser les matrices
  var a[n,n] := ([n] ([n] 1.0)),
      b[n,n] := ([n] ([n] 1.0)),
      c[n,n] : real
  # calculer le produit scalaire de a[i,*] * b[*,j]
  procedure scalaire(i,j: int)
    var somme := 0.0
    fa k := 1 to n -> somme +:= a[i,k]*b[k,j] af
    c[i,j] := somme
  end

  # calculer n**2 produit scalaire en parallele
  co (i := 1 to n, j := 1 to n) scalaire(i,j) oc


  # imprimer le résultat
  fa i := 1 to n ->
    fa j := 1 to n -> writes(c[i,j], " ") af
    write()
  af
end
```

**Programme 6.11** – Multiplication de matrices en parallèle

```
1  resource quick()
2    op sort(var a[1:*]: int)  #  declaration de sort()
3
4    var n: int; getarg(1,n)
5    var a[1:n]: int
6
7    # lecture des données
8    fa i := 1 to n -> read(a[i]) af
9    write("input:"); fa i := 1 to n -> write(a[i]) af
10   sort(a)
11
12   write("sorted:"); fa i := 1 to n -> write(a[i]) af
13
14   proc sort(a)
15     if ub(a) <= 1 -> write("fin tri", ub(a)); return fi
16     fa i := 1 to ub(a) -> writes(a[i], " ") af; write()
17     var pivot := a[1]
18     var lx := 2, rx := ub(a)
19     do lx <= rx ->
20         if a[lx] <= pivot -> lx++
21         [] a[lx] >  pivot -> a[lx] :=: a[rx]; rx--
22         fi
23     od
24     a[rx] :=: a[1]
25
26     co sort(a[1:rx-1]) || sort(a[lx:ub(a)]) oc
27   end
28 end quick
```

**Programme 6.12** – Tri rapide en parallèle

produit le tableau de résultats suivants :

| | | | | | |
|---|---|---|---|---|---|
| 1 | 3 | 6 | 10 | 15 | 21 |

Le programme 6.13 présente une solution séquentielle à ce problème qui s'exécute en un temps proportionnel à $\mathcal{O}(n)$.

```
1  resource partiel()
2     const n := 10
3     var a[n]: int, somme[n]: int
4     fa i := 1 to n -> a[i]=i af
5
6     somme[0] := a[0]
7     fa i := 1 to n ->
8           sum[i] := sum[i-1] + a[i]
9     af
10    fa i := 1 to n -> printf("% i ,", somme[i] ) af
11    write()
12 end partiel
```

**Programme 6.13** – Calcul de la somme partielle en parallèle

Le programme 6.14 propose une version parallèle de la somme partielle. Son fonctionnement est le suivant :

1. Addition en parallèle de toutes les paires voisines (voir la figure 6.18).

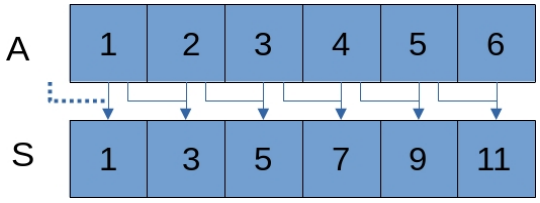



**Figure 6.18** – La première étape d'une somme partielle

2. Pour compléter les sommes partielles, on combine les résultats obtenus en 1 dans le tableau $S$. Il ne faut toutefois pas utiliser les voisins directs mais plutôt les voisins distants de 2 positions. De fait, comme $S[2] = A[1] + A[2]$ et $S[4] = A[3] + A[4]$, pour compléter la somme partielle de $S[4]$, il faut évaluer $S[4] = S[2] + S[4]$ $(= A[1] + A[2] + A[3] + A[4])$. La figure 6.19 décrit cette deuxième étape.

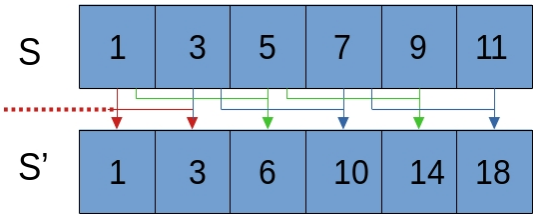



**Figure 6.19** – La seconde étape d'une somme partielle

3. Ces additions se poursuivent en doublant la distance entre les indices des éléments à chacune des étapes. Ainsi, à l'étape 1, nous utilisions une distance de 1, une distance de 2 à l'étape 2, une distance de 4 à l'étape 3, une distance de 8 à l'étape 4, ... La somme se termine quand la distance est plus grande que la dimension du tableau. La figure 6.20 illustre toutes les étapes de l'addition de cet exemple.

Le programme 6.14 implante cet algorithme. Il calcule les sommes partielles en un temps proportionnel à $\mathcal{O}(log_2(n))$.

### 6.3.6 Exemple 5 -Algorithme de Jacobi (calcul de grille)

Comme nous l'avons déjà mentionné, plusieurs problèmes en traitement d'images ou en modélisation scientifique sont résolus par des calculs sur les grilles (*mesh*).

Dans cette section, nous implantons l'équation de Laplace qui sert à résoudre des équations différentielles partielles (PDE). L'équation de Laplace est relativement simple à implanter car elle s'approxime avec des algorithmes tels que l'itération de Jacobi, Gauss-Seidel ou SOR (Successive Over Relaxation). Tous ces algorithmes ont recourt à des matrices. Les «bords» de ces matrices sont initialisés aux conditions limites et le but est de calculer une approximation de la valeur de chaque point interne. Cela correspond grossièrement à trouver un état stable à une équation.

Le programme 6.15 présente le modèle de calcul choisi. Dans ce modèle, on remarque qu'une nouvelle valeur est calculée pour chaque point, et ce, à chaque itération. Le calcul de chaque nouveau

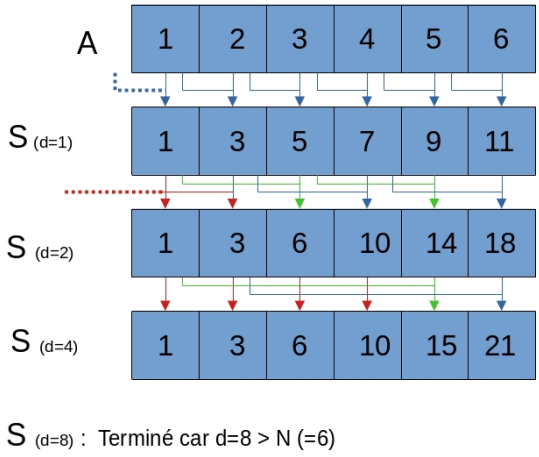



**Figure 6.20** – Toutes les étapes d'une somme partielle

```
1  resource partiel()
2     const n := 10
3     var a[n]: int, somme[n]: int, temp[n]: int
4     fa i := 1 to n -> a[i]=i af
5
6     process sum(i := 1 to n)
7           var d:int:=1
8           somme[i] := a[i]
9           barrier(i)
10          do d<n ->
11            begin
12              temp[i] := somme[i]
13              barrier(i)
14              if i-d >= 0 ->somme[i] := temp[i-d]+ somme[i]
15              fi
16              barrier(i)
17              d := d+d
18            end
19          od
20    end
21   final
22       fa i := 1 to n -> printf("% i ,", somme[i] ) af
23       write()
24   end
25 end partiel
```

**Programme 6.14** – Calcul de la somme partielle en parallèle

point peut se faire en parallèle car généralement ce sont des tâches indépendantes (parallélisme trivial). Les itérations se terminent lorsque qu'une certaine condition est atteinte (précision du calcul, nombre d'itérations maximal, ...)

```
Initialisation de la matrice

Tant que ce n'est pas terminé
    1 - Calculer une nouvelle valeur pour chaque point
    2 - Vérifier si on a terminé
    3 - Barrière si on utilise un calcul parallèle
```

**Programme 6.15** – Modèle de calcul

Le programme 6.16 implante une solution séquentielle simple pour résoudre une équation de Laplace en 2D. Elle fait appel à une méthode de différenciation finie appelée Jacobi. Dans cet algorithme, le nouveau point est tout simplement la moyenne des quatre points voisins dans la matrice (de l'itération précédente). L'algorithme se termine lorsque la précision requise est atteinte, i.e. lorsque la différence entre deux points consécutifs sera inférieure à un certain $\epsilon$ pour toute la grille. Il est aussi possible d'arrêter le calcul après un certain nombre $K$ d'itérations.

```
resource jacobi()
   const EPSILON := .01
   const n := 4
   var grid[0:n+1,0:n+1]: real, temp[0:n+1,0:n+1]: real
   var diff:real, maxdiff : real:=1, borne:real
   #.......initialisation de la grille..................
   do maxdiff > EPSILON ->
       fa i := 1 to n ->
          fa j := 1 to n ->
             temp[i,j]:=(grid[i-1,j]+ grid[i+1,j]+grid[i,j-1]+grid[i,j+1])/4
          af
       af
       maxdiff :=0
       fa i := 1 to n ->
          fa j := 1 to n ->
              diff := abs(temp[i,j] - grid[i,j])
              if diff>maxdiff -> maxdiff := diff fi
          af
       af
       fa i := 1 to n ->
           fa j := 1 to n -> grid[i,j] := temp[i,j] af
       af
   od
  #impression resultat
end jacobi
```

**Programme 6.16** – Calcul de Jacobi en séquentiel

Le programme 6.17 implante une version parallèle de cette solution.

```
1  resource jacobi()
2     const EPSILON := .01, n := 4
3     var g[0:n+1,0:n+1]: real, t[0:n+1,0:n+1]: real
4     var maxdiff[1:n,1:n]:real
5     var borne:real
6     #.........initialisation de la grille.................
7     process calcul(i :=1 to n, j:=1 to n)
8        var k:int, l:int, fini:bool :=false;
9        maxdiff[i,j]:=1
10       do fini=false ->
11           t[i,j]:=(g[i-1,j]+ g[i+1,j]+g[i,j-1]+g[i,j+1])/4
12           maxdiff[i,j] := abs(t[i,j] - g[i,j])
13           barrier(i,j)
14           fini := true
15           fa k := 1 to n ->
16               fa l := 1 to n ->
17                  if maxdiff[k,l] > EPSILON -> fini := false fi
18               af
19           af
20           barrier(i,j)
21           fa i := 1 to n ->
22               fa j := 1 to n -> g[i,j] := t[i,j]; af
23           af
24       od
25    end
26    final
27        #impression resultat
28    end
29 end jacobi
```

**Programme 6.17** – Calcul de Jacobi en parallèle

# Annexe A

# Le calcul de $\pi$ en parallèle

## A.1 Le calcul de Pi

> Contrairement à la croyance populaire et à ce que la plupart des mathématiciens vous diront, tous les chiffres décimaux du développement de $\pi$ sont connus. Ce sont 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. C'est l'ordre dans lequel ils apparaissent qui n'est pas connu.
>
> – Laurence Turner[53]

Tout le monde connaît le nombre $\pi$ dont la valeur est (à la $20^e$ décimale) 3,14159265358979*323846*, mais, en général, c'est sans toutefois connaître la façon de la calculer.

Plusieurs techniques permettent d'évaluer $\pi$ [53, 48, 20, 57, 1, 2]. On choisit ici d'en présenter trois qui se parallélisent très bien. Cependant celles-ci n'offrent pas des performances équivalentes, ni ne produisent des résultats de même précision. En fait, aucune d'elles n'a pu produire un résultat significativement satisfaisant, le meilleur obtenu ne dépassant pas la $14^e$ décimale. Sans investiguer pour autant, on suppose que la représentation des nombres sur l'ordinateur est l'une des causes.

### A.1.1 Le calcul de $\pi$ avec la méthode de Monte-Carlo

Wikipedia mentionne [69] que l'expression «Méthodes de Monte-Carlo» désigne une famille de méthodes algorithmiques, inventée en 1947, visant à calculer une valeur numérique approchée en utilisant des procédés aléatoires, c'est-à-dire des techniques probabilistes. Ce nom fait référence aux jeux de hasard pratiqués au casino de Monte-Carlo. La méthode sert dans plusieurs domaine dont notamment pour dans le calcul des intégrales de dimensions plus grandes que un, en physique des particules, en statistique pour la répartition du risque, mais le véritable développement de ces méthodes s'est effectué sous l'impulsion de John von Neumann et Stanislaw Ulam, lors de la seconde guerre mondiale, dans le cadre, notamment, des recherches sur la fabrication de la bombe atomique.

Revenons à notre estimation de $\pi$. La méthode de Monte-Carlo est simple, se parallélise très bien, mais n'est guère précise en plus d'être relativement lente. Son principe de base consiste à inscrire un cercle, de rayon $r$, à l'intérieur d'un carré dont la dimension des côtés est de $2r$ tel

qu'illustré à la figure A.1.

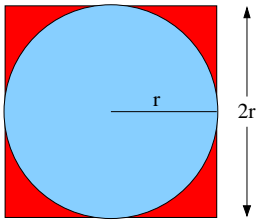



**Figure A.1**

On se sert de la relation entre le cercle et le carré pour construire une formule pour calculer $\pi$. Soit $A_s$, l'aire du carré et $A_c$, l'aire du cercle. On a donc :

1. $A_s = (2 \times r)^2 = 4 \times r^2$ et
2. $A_c = \pi \times r^2$.

De (1) on tire

$$r^2 = A_s/4$$

d'où (2) devient

$$A_c = \frac{\pi \times A_s}{4} \implies \pi = \frac{4 \times A_c}{A_s} \qquad (3)$$

À partir du dernier résultat (3), on élabore la méthodologie suivante pour approximer $\pi$ :

1. générer aléatoirement des points à l'intérieur du carré ;
2. compter le nombre de points à l'intérieur du carré et celui à l'intérieur du cercle, disons $nA_s$ et $nA_c$ respectivement ;
3. générer $v$, défini par $nA_s/nA_c$ (la proportion du nombre de points dans le cercle sur le nombre de points dans le carré) ;
4. estimer $\pi$ en évaluant $\pi = 4 * v$.

Ce procédé procure une approximation assez grossière de $\pi$. En effet, la valeur la plus précise obtenue selon cette approche est $3,1415851$ et ce, en générant aléatoirement un milliard de points à l'intérieur du carré. En général, le résultat est correct jusqu'à la $5^e$ décimale.

Cet algorithme a toutefois l'avantage de se paralléliser très facilement (parallélisme trivial). Les programmes A.1 et A.2 illustrent respectivement les versions séquentielle et parallèle de l'algorithme de Monte-Carlo.

### Analyse des résultats

La particularité de l'algorithme de Monte-Carlo est de devoir générer des points de façon aléatoire à l'intérieur du carré. Comme il existe de multiples bibliothèques de génération de nombres aléatoires, nous en avons testé quelques unes pour vérifier leur comportement dans des programmes parallèles. Les résultats obtenus selon ces différentes versions sont compilés au tableau A.1. À chaque fois, nous avons généré un milliard de points et utilisé un ordinateur portable Dell Latitude D6430 (Intel i7) pour effectuer les calculs.

```cpp
int main() {
    srand(time(NULL));
    long long int circle = 0;
    long long int j;
    for (j = 0; j < NB; j ++) {
        double x = static_cast<double>(rand()) / static_cast<double>(RAND_MAX);
        double y = static_cast<double>(rand()) / static_cast<double>(RAND_MAX);
        if (x * x + y * y <= 1.0) circle ++;
    }
    cout << (double)circle / j * 4.0 << endl;
    return 0;
}
```

**Programme A.1** – Version séquentielle de l'algorihme de Monte-Carlo

```cpp
long long int total[4];

int calculPI(int no, long long int cal) {
    srand(time(NULL));
    total[no] = 0;
    int j;
    for (j = 0; j < cal; j++) {
        double x = static_cast<double>(rand()) / static_cast<double>(RAND_MAX);
        double y = static_cast<double>(rand()) / static_cast<double>(RAND_MAX);
        if (x * x + y * y <= 1.0) total[no] ++;
    }
    return 0;
}
int main() {
    long long int nb_calcul;
    int NB_PCSR = 2;
    double resul = 0.0;
    nb_calcul = NB / NB_PCSR;
    vector<thread> threads;
    for(int i = 0; i < NB_PCSR; ++i) { threads.push_back(thread(calculPI, i, nb_calcul)); }
    for(auto& thread : threads)      { thread.join(); }
    for(int i = 0; i < NB_PCSR; ++i) { resul = resul + total[i]; }
    resul = resul/(float)NB * 4.0;
    std::cout << NB << "        PI = " << resul << std::endl;
    return 0;
}
```

**Programme A.2** – Version parallèle de l'algorihme de Monte-Carlo

Pour la première version, on a eu recourt à la fonction de génération de nombres aléatoires standard du langage C (et aussi C++), `rand` . Celle-ci produit de très bons résultats en mode séquentiel mais désastreux en mode parallèle. Le programme séquentiel opérant avec `rand` s'exécute en 20 secondes environ, alors qu'en parallèle avec deux processeurs, le programme requiert plus d'une minute. Ce résultat est d'abord déroutant (le but du parallélisme n'est certainement pas d'être moins efficace que le séquentiel en monopolisant plus de ressources), mais s'explique par le fait que la fonction `rand` n'est ni compatible avec le multi-fils («*thread-safe*») ni ré-entrante.

On constate aussi que les fonctions récentes de C++ 11 n'améliorent guère la qualité du résultat. Ainsi, les versions séquentielles nécessitent plus de cinq minutes pour calculer une estimation de $\pi$. Toutefois, comme ces fonctions sont compatibles avec le multi-fils et ré-entrantes, la version parallèle, s'exécutant sur deux processeurs, prend deux fois moins de temps.

Une autre évaluation, cette fois à l'aide de la fonction `rand_r` (une version ré-entrante de `rand` qui supporte le multi-fils), toujours pour un milliard de points, aboutit en un temps de 14,69 secondes en version séquentielle, alors qu'en version parallèle elle se fait en 7,4 secondes sur deux processeurs, comparativement à 4,28 secondes sur 4 processeurs.

La conclusion s'impose. Choisir judicieusement les fonctions est d'une grande importance en ce qui concerne la performance autant en programmation séquentielle qu'en parallèle.

**Table A.1** – Résultats pour l'algorithme de Monte-Carlo

| Algorithme | Séquentiel | Parallèle (2) | Parallèle (4) | Parallèle (8) |
|---|---|---|---|---|
| Monte-carlo (rand) | 20,343 | 10:56,767 (7:52,540) | | |
| Monte-Carlo (rand_r) | 14,331 | 7,401 (14,768) | 5,055 (17,692) | 2,913 (22,892) |
| Monte-Carlo (default C++) | 5:13,387 | 2:37,992 (5:15,012) | 1:27,091 (5:43,080) | 1:00,111 (7:46,472) |
| Monte-Carlo (mt19937) | 5:43,298 | 2:58,005 (5:55.432) | 1:34,263 (6:16,156) | 1:04,911 (8:29,564) |

### A.1.2 Le calcul de $\pi$ par le développement de McLaurin

Une seconde méthode pour évaluer $\pi$ consiste à faire appel au développement en série de McLaurin de la fonction $\arctan(x)$.

Soit :

$$\arctan(x) = x - x^3/3 + x^5/5 - x^7/7 + x^9/9 - x^{11}/11 + \dots$$

Comme :

$$1 = \tan(\pi/4)$$

Nous pouvons en déduire que :

$$\pi = 4 \times \arctan(1) = 4 \times (1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots)$$

$$\pi = 4 \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1}$$

Cette formule est simple, se code très bien, mais converge extrêmement lentement. Le programme A.3 présente une implantation triviale de cet algorithme. Les programmes A.4 et A.5 sont des versions «améliorées» du premier. Finalement, le programme A.6 implante la version parallèle.

```
1  int main()  {
2      long double QUAT = 4.0;
3      long double UN = 1.0;
4      long double DEUX = 2.0;
5      long double pi;
6      long double x=0;
7      long double sum = 0.0;
8      for (int i=0; i < NB; ++i)   {
9          x = pow(-UN,i)/(DEUX*i+1);
10         sum = sum + QUAT*x;
11     }
12     pi = sum;
13     cout << "Pi = " << pi << endl;
14 }
```

**Programme A.3** – Calcul de $\pi$ par une série de McLaurin (version naïve)

```
1  int main()
2  {
3      ...
4      for (int i=0; i < NB; ++i) {
5         int j = i%2;
6         x = pow(-UN,j)/(DEUX*i+1);
7         sum = sum + QUAT*x;
8      }
9      pi = sum;
10     cout << "Pi = " << pi << endl;
11 }
```

**Programme A.4** – Calcul de $\pi$ par une série de McLaurin (version améliorée)

### Analyse des résultats

Pour obtenir des résultats «comparables» à ceux obtenus par la méthode de Monte-carlo, nous avons dû utiliser un milliard d'itérations pour générer $\pi$ sur le même ordinateur. La tableau A.2 présente les temps d'exécution des différentes implantations.

| Algorithme | Séquentiel | parallèle (2) | Parallèle (4) | Parallèle (8) |
|---|---|---|---|---|
| (1) Mc Laurin (pow + exposant) | 272,628 | 136,976 (273,880) | 71,631 (285,656) | 46,043 (364,316) |
| (2) Mc Laurin (pow) | 48,632 | 24,803 (49516) | 12,700 (50,608) | 8,421 (1:4,816) |
| (3) Mc Laurin (sans pow) | 9,239 | 4,689 (9,332) | 2,825 (10,340) | 2,484 (16,092) |

**Table A.2** – Comparaison des trois algorithmes

```
1  int main()
2  {
3      ...
4      for (int i=0; i < NB; ++i)   {
5          long double val;
6          int j = i%2;
7          if (j==0) val = 1.0;
8          else val = -1.0;
9          x = val/(DEUX*i+1);
10         sum = sum + QUAT*x;
11     }
12     pi = sum;
13     cout << "Pi = " << pi << endl;
14 }
```

**Programme A.5** – Calcul de $\pi$ par une série de McLaurin (version optimale)

On remarque que l'implantation naïve (1) de cet algorithme s'exécute en 272.628 secondes. Notons aussi que l'expression $(-1)^n$ présente lors du calcul d'une série de McLaurin (qui ne sert qu'à alterner le signe d'un terme à l'autre selon que $n$ est pair ou impair) provoque l'évaluation d'un nombre à une puissance toujours de plus en plus astronomique. En évitant ce lourd calcul inutile, la version s'améliore (2) et le travail est réalisé en 48,632 secondes. Finalement, en renonçant à la fonction `pow`, la version devient optimale (3) et ne requiert plus qu'un temps de 9,120 secondes.

L'un des avantages de cet algorithme, c'est la facilité à la paralléliser tout en obtenant de très bons résultats (temps d'exécution), car en fait, il suffit seulement de diviser le calcul entre chaque fil d'exécution. La version parallèle (programme A.6) s'exécute en 4,805 secondes sur deux processeurs, en 2,672 secondes sur 4 processeurs et en 2,199 secondes sur 8 processeurs (dont 4 virtuels par *hyperthreading*).

### A.1.3 Le calcul de $\pi$ par intégration numérique

La troisième méthode proposée ici pour estimer $\pi$ est par une intégration numérique [35].
Sachant que $\pi = 4 \times \arctan(1)$ et que

$$\frac{d}{dx}(\arctan(x)) = (\arctan(x))' = \frac{1}{1+x^2}$$

On estime $\pi$ en évaluant l'intégrale définie suivante :

$$\pi = 4 \times \arctan(1) = 4 \times \int_0^1 \frac{1}{1+x^2}\, dx$$

Selon le théorème fondamental, il est possible d'évaluer (d'approximer) une intégrale par la somme de Reimann, ce qui produit, dans notre cas particulier, la somme :

$$\int_0^1 f(x)\, dx = \lim_{n\to\infty} \sum_{k=0}^{n} f(x_k^*)\, \Delta x_k = 4 \times \lim_{n\to\infty} \sum_{k=0}^{n} \frac{1}{1+(x_k^*)^2}\, \Delta x_k$$

où les $x_k^*$ sont des points quelconques dans l'intervalle $[x_k, x_{k+1}]$ et $\Delta x_k$ représente la largeur de la $k^e$ intervalle.

```cpp
long double total[4];

int calculPI(int no, long long int cal)
{
    long double QUAT = 4.0;
    long double UN = 1.0;
    long double DEUX = 2.0;
    long double pi;
    long double x=0;
    long double sum = 0.0;

    total[no] = 0;
    cout << " ==> " << cal << endl;

    for (int i=no*cal; i < (no+1)*cal; ++i)
    {
        long double val;
        int j = i%2;
        if (j==0) val = 1.0;
        else val = -1.0;
        x = val/(DEUX*i+1);

        sum = sum + QUAT*x;
    }
    total[no]  = sum;
    cout << "fin "<< no <<  "  total = " << total[no] << endl;
    return 0;
}
int main()
{
    long long int nb_calcul;
    int NB_PCSR = 8;
    long double resul = 0;
    nb_calcul = NB / NB_PCSR;
    cout.precision(20);

    cout << " ==> " << nb_calcul << endl;

    vector<thread> threads;
    for(int i = 0; i < NB_PCSR; ++i)
    {
        threads.push_back(thread(calculPI, i, nb_calcul));
    }
    for(auto& thread : threads)
    {
        thread.join();
    }
    resul = 0.0;
    for(int i = NB_PCSR-1; i >=0; i--)
    {
        cout << "Somme processuer " << i << endl;
        resul = resul + total[i];
    }
    std::cout << NB << "        PI = " << resul << std::endl;
    return 0;
}
```

**Programme A.6** – Calcul de $\pi$ en parallèle via une série de McLaurin

Comme nous séparons l'intervalle $[0, 1]$ en sous-intervalle de taille identique de $\frac{1}{n}$, la formule devient :

$$\pi = 4 \times \frac{1}{n} \times \lim_{n\to\infty} \sum_{k=0}^{n} \frac{1}{1 + (x_k^*)^2}$$

Nous choisissons $x_k^*$ comme le point milieu du sous-intervalle. La figure A.2 illustre comment procède ce calcul. Nous faisons la somme des points suivants :

1. $x_0^* = \frac{\Delta x_0}{2} = \frac{1}{2n}$
2. $x_1^* = \Delta x_0 + \frac{\Delta x_1}{2} = \frac{1}{n} + \frac{1}{2n} = \frac{3}{2n}$
3. $x_1^* = \Delta x_0 + \Delta x_1 + \frac{\Delta x_2}{2} = \frac{1}{n} + \frac{1}{n} + \frac{1}{2n} = \frac{5}{2n}$
4. ...
5. $x_k^* = \frac{2k+1}{2n}$

La formule à implanter pour notre approximation est la suivante : :

$$\pi = 4 \times \frac{1}{n} \times \lim_{n\to\infty} \sum_{k=0}^{n} \frac{1}{1 + (\frac{2k+1}{2n})^2}$$

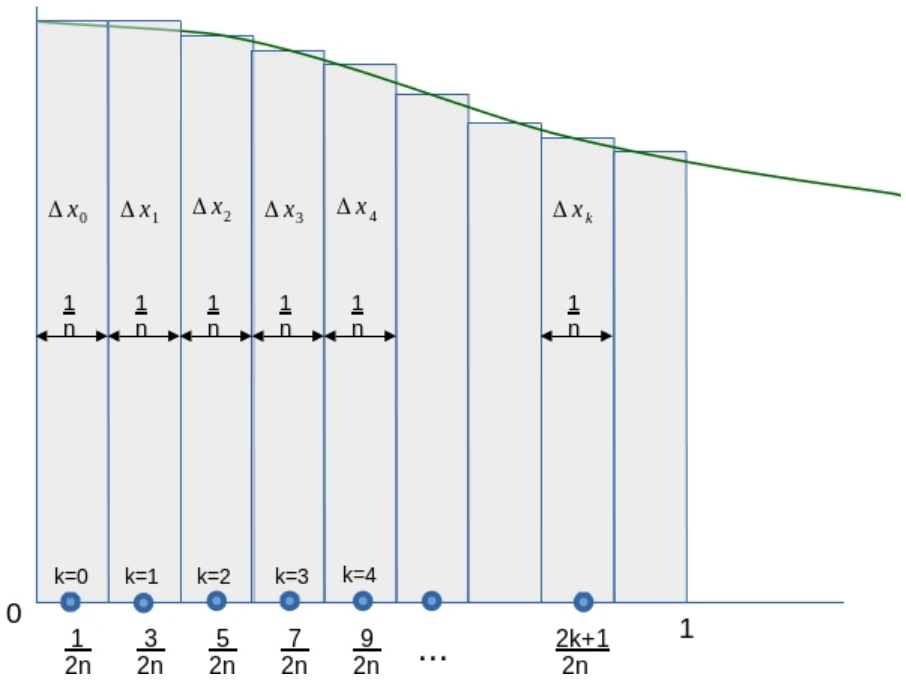



**Figure A.2**

La technique est assez simple à implanter. Le programme A.7 fournit une implantation de cet algorithme en C++. Sur un milliard d'itérations, il prend 6,989 secondes à s'exécuter.

Ce programme se parallélise facilement et donne de très bons résultats. Le programme A.8 présente la version parallèle de cet algorithme. Cette version prend 3,945 secondes à s'exécuter sur 2 processeurs, 2,420 secondes sur 4 processeurs et 1,859 secondes sur 8 processeurs.

```
1  int main()
2  {
3      long double QUAT = 4.0;
4      long double UN = 1.0;
5      long double DEMI = 0.5;
6      long double pi;
7      long double x=0;
8      long double sum = 0.0;
9      long double step = UN/(long double) NB;
10     cout.precision(20);
11
12     for (long long int i=0; i <NB; ++i) {
13         x = (i+DEMI)*step;
14         sum = sum + QUAT/(UN+x*x);
15     }
16     pi = step * sum;
17     cout << "Pi = " << pi << endl;
18 }
```

**Programme A.7** – Calcul de $\pi$ par la somme de Reimann

## A.2 Conclusion

Dans ce document, nous vous avons présenté diverses techniques pour approximer $\pi$. Celles-ci se distinguent par leur facilité d'implantation et leur efficacité tant pour leur version séquentielle que parallèle. La tableau A.3 compare les résultats des trois algorithmes.

| Algorithme | Séquentiel | parallèle (2) | Parallèle (4) | Parallèle (8) |
|---|---|---|---|---|
| Monte-carlo (rand) | 20,343 | 10:56,767 (7:52,540) | | |
| Monte-Carlo (rand_r) | 14,331 | 7,401 (14,768) | 5,055 (17,692) | 2,913 (22,892) |
| Monte-Carlo (default C++) | 5:13,387 | 2:37,992 (5:15,012) | 1:27,091 (5:43,080) | 1:00,111 (7:46,472) |
| Monte-Carlo (mt19937) | 5:43,298 | 2:58,005 (5:55.432) | 1:34,263 (6:16,156) | 1:04,911 (8:29,564) |
| Mc Laurin (pow + exposant) | 4:32,628 | 2:16,976 (4:33,880) | 1:11,631 (4:45,656) | 46,043 (6:4,316) |
| Mc Laurin (pow) | 48,632 | 24,803 (49516) | 12,700 (50,608) | 8,421 (1:4,816) |
| Mc Laurin (sans pow) | 9,239 | 4,689 (9,332) | 2,825 (10,340) | 2,484 (16,092) |
| Reimann | 6,989 | 3,945 (7,852) | 2,420 (8,992) | 1,859 (13,440) |

**Table A.3** – Comparaison des trois algorithmes

```cpp
long double total[4];

int calculPI(int no, long long int nb_cal, int nb_pcsr )
{
    long double QUAT = 4.0;
    long double UN = 1.0;
    long double DEMI = 0.5;
    long double pi;
    long double x=0;
    long double sum = 0.0;
    long double step = UN/(long double) nb_cal;
    total[no] = 0;
    long long int cal = nb_cal/nb_pcsr;

    for (int i=no*cal; i < (no+1)*cal; ++i) {
        x = (i+DEMI)*step;
        sum = sum + QUAT/(UN+x*x);
    }
    total[no]  = step * sum;
    return 0;
}

int main()
{
    long long int nb_calcul;
    int NB_PCSR = 2;
    long double resul = 0;
    nb_calcul = NB / NB_PCSR;
    cout.precision(20);

    cout << " ==> " << nb_calcul << endl;

    vector<thread> threads;
    for(int i = 0; i < NB_PCSR; ++i)
    {
        threads.push_back(thread(calculPI, i, NB, NB_PCSR));
    }
    for(auto& thread : threads)
    {
        thread.join();
    }
    resul = 0.0;
    for(int i = NB_PCSR-1; i >=0; i--)
    {
        cout << "Somme processuer " << i << endl;
        resul = resul + total[i];
    }
    std::cout << NB << "        PI = " << resul << std::endl;
    return 0;
}
```

**Programme A.8** – Calcul de $\pi$ en parallèle par la somme de Reimann

# Annexe B

# La construction «Map/Reduce»

## B.1 La construction «Map»

Dans plusieurs langages de programmation, la construction «Map» [58] est une fonction d'ordre supérieure (Higher order function) qui prend une fonction prédéfinie en paramètre et l'applique à chaque élément d'une structure tel une liste ou un tableau. Elle retourne une liste de résultats du même ordre de grandeur que la structure.

Le principe d'utilisation d'un construction «Map» ressemble au suivant :

```
map(fonction, liste d'éléments)
```

Les programmes 1, 2 et 3 montre l'utilisation du «Map» en Python.

```
# Utilisation du map pour convertir des températures

def fahrenheit(T): return ((float(9)/5)*T + 32)

temp = (36.5, 37, 37.5,39)

F = list(map(fahrenheit, temp))
print(F)

#-------------------------------------
        Résultats :
[97.7, 98.60000000000001, 99.5, 102.2]
```

**Programme 1** – Exemple d'utilisation du «map» : conversion de températures.

La construction «Map» (ou patron de conception) est utilisée en programmation parallèle[59] pour résoudre des problèmes reliés au parallélisme trivial. En rappel, les problèmes de parallélisme sont dits triviaux s'il n'y a aucune donnée partagée et s'ils ne requièrent aucune synchronisation ou communication, sauf une barrière ou un «join» à la fin.

Dans les exemples précédents, la «fonction» est appliquée à tous les éléments de la liste (comme une boucle le ferait dans le monde du séquentiel). Si les multiples instances de la fonction s'exécute en parallèle, l'utilisation de fils d'exécution, d'hyper-fils (hyperthreads), de SIMD lanes, de multiples coeurs ou de multiples ordinateurs est nécessaire pour traiter chaque élément de la liste.

```python
# Utilisation du map pour des fonctions mathématiques

def sqr(x): return x ** 2

items = [1, 2, 3, 4, 5]

liste = list(map(sqr, items))
print(liste)

print(list(map(sqr, range(10))))
print(list(map(math.sin, items)))

#----------------------------------------
         Résultats :
[1, 4, 9, 16, 25]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
[0.84..., 0.90..., 0.14..., -0.75..., -0.95...]
```

**Programme 2** – Exemple d'utilisation du «map» : fonctions mathématiques

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

# Utilisation du map pour encoder/décoder des messages

import functools

def encode(x) : return (chr(ord(x)+3))
def toString(l) : return "".join(l)
def decode(x) : return (chr(ord(x)-3))

def main():
    #print(f("a"))
    #print(f(" "))

    msg1 = input("Entrez le message à encrypter : ")
    a = list(map(encode,msg1))
    #print(a)

    b = toString(a)
    print("Le message secret est : ", b)

    a = list(map(decode,a))
    #print(a)

    b = toString(a)
    print("Le message decrypte est : ", b)


if __name__ == "__main__":
    main()
```

**Programme 3** – Exemple d'utilisation du «map» : encodage/décodage d'un message

Étant donné que cette construction est conçue pour régler les problèmes de parallélisme triviaux, il n'est pas surprenant qu'elle soit d'abord apparue dans les langages fonctionnels (dans lesquels il n'y a aucune variable partagée). Cette construction se retrouve aujourd'hui dans plusieurs langages séquentiels non fonctionnels (grâce à des extensions fonctionnelles) tel Python.

Plusieurs environnements parallèles supportent ce type de construction :

- OpenMP (sous forme de boucles parallèles)
- Cilk (sous forme de boucles parallèles)
- OpenCL et Cuda (kernel)
- MPI

Cette construction est souvent combinée à une autre construction populaire, la construction «Reduce». En effet, comme la construction «Map» a le potentiel de retourner une multitude de résultats, la construction «Reduce» permet de combiner les résultats.

## B.2 La construction «Reduce»

La construction «Reduce» peut être vue comme une fonction d'ordre supérieure (Higher order function) qui prend une fonction prédéfinie en paramètre et l'applique à chaque élément d'une structure tel une liste ou un tableau. Toutefois, à la différence du «Map», elle retourne un seul résultat qui est une «réduction» des données en entrée.

Il vous est probablement déjà arrivé de coder une fonction qui calcule la somme de tous les éléments d'une liste (voir programme 4). La construction «Reduce» peut être vue comme une généralisation de ce principe. Ce serait donc une fonction à laquelle on passe :

- une fonction de réduction ;
- une liste de valeurs sur lesquelles on applique la réduction.

Le programme 5 donne un exemple d'utilisation de la construction «Reduce». Plusieurs langages, tel Python, offre une telle construction.

```
def somme(liste):
    accumulateur = 0
    for elt in liste:
        accumulateur = accumulateur + elt
    return accumulateur

maliste = [1,2,3,4,5]
somme(maliste)
#-------------------------------------------
         Résultats :
15
```

**Programme 4** – Exemple de programme qui fait la somme.

Dans le cadre de la programmation parallèle [60], la construction «Reduce» est vue comme une primitive de communication qui combine plusieurs vecteurs ou éléments d'un vecteur en une seule entité (vecteur ou valeur) en utilisant un opérateur binaire associatif. Au départ, chaque

```
from functools import *

def somme(x,y) : return x+y
def mult(x,y) : return x*y
def m3(x) : return 3*x

v1 = [1,2,3,4,5]
v2 = [5,4,3,2,1]

# Utilisation de la construction map
res_map_1 = list(map(m3, v2))
res_map_2 = list(map(mult, v1, v2))
res_map_3 = list(map(somme, v1, v2))
print("res_map_1 = ",res_map_1)
print("res_map_2 = ",res_map_2)
print("res_map_3 = ",res_map_3)

# Utilisation de la construction reduce
res_red_1 =  reduce(somme, res_map_3)
res_red_2 = reduce(mult, v1)
print("res_red_1 = ",res_red_1)
print("res_res_2 = ",res_red_2)
#-----------------------------------------
         Résultats :
res_map_1 =  [15, 12, 9, 6, 3]
res_map_2 =  [5, 8, 9, 8, 5]
res_map_3 =  [6, 6, 6, 6, 6]
res_red_1 =  30
res_res_2 =  120
```

**Programme 5** – Exemple d'utilisation du «Reduce» : la somme

vecteur/valeur est présent dans un processeur distinct. Le rôle de la construction «Reduce» consiste à appliquer l'opérateur sur chaque processeur (dans l'ordre) jusqu'à ce qu'il ne reste qu'un seul processeur.

La réduction d'un ensemble d'éléments est une partie intégrale de modèle de programmation «Map/Reduce» qui consiste à appliquer une fonction à tous les éléments avant d'appliquer la réduction.

Des environnements tels, MPI et OpenMP, offrent de opérateurs de réduction (MPI_reduce, MPI_Allreduce, ...).

## B.3 La construction «MapReduce»

La construction «MapReduce» [34, 43, 54, 30, 25, 44, 13, 33, 29, 38, 10, 27] est un patron de conception (inventé par Google) dont l'objectif est d'effectuer des calculs parallèles ou distribués sur des données potentiellement très volumineuses. La construction «MapReduce» a aussi pour objectif de simplifier la programmation parallèle/distribuée et de permettre aux développeurs de décrire le traitement qu'ils désirent effectuer en terme des fonctions de mise en correspondance («*Map*») et de réduction («*Reduce*»). En fait, l'introduction de cette construction est basée sur le principe que toute parallélisation de traitement sur des données massives peut s'effectuer à l'aide des ces deux opérations : «*Map*» et «*Reduce*».

La construction «MapReduce» est basée sur un concept bien connu en algorithmique, le concept diviser pour régner, qui se divise en trois étapes :

1. Diviser : étape qui consiste à découper le problème initial en sous-problèmes.

2. Régner : étape qui consiste à résoudre les sous-problèmes indépendamment soit de manière récursive ou directement s'ils sont de petites tailles.

3. Combiner : étape qui consiste à construire la solution du problème initial en combinant les solutions des sous-problèmes.

On considère donc que la construction «MapReduce» est le diviser pour régner des données massives.

La construction «MapReduce» se divise donc en un certains nombre d'étapes :

1. Division des données et du calcul

   Lors de cette première étape, le maître (*JobTracker* dans Hadoop) analyse le problème (les données), le découpe en sous-problèmes (blocs de données) et délègue le calcul à d'autres processeurs/nœuds (*TaskTrackers* dans Hadoop).

   C'est lors de cette première étape qu'il faut structurer/décomposer les données. En effet, afin de pouvoir traiter des données avec ce patron de conception («MapReduce»), il est nécessaire de structurer les données en paires (clé, valeur). Dans certains cas, cette structuration est facile, mais dans d'autres cela peut être assez complexe. On verra dans nos exemples comment structurer les données en paires.

   La division des données en blocs dépend du nombre de nœuds pouvant effectuer l'opération de correspondance. Ainsi, avec 1000 enregistrements de données et 100 «mapper» on assigne à chacun 10 enregistrements de données. Si non a 50 mapper, on assigne à chacun 20 enregistrements. etc.

   Cette première étape est parfois combiner avec la seconde qui combine alors la division du calcul et la mise en correspondance.

   Prenons comme exemple un programme consistant à compter les mots dans un ensemble de documents. Lors de cette première phase, la tâche sera divisée en sous-tâches consistant chacune à trouver le nombre d'occurrences de chaque mot dans un document. Le couple de données en entrée de la fonction de mise en correspondance sera donc le nom du document (clé) et son contenu (valeur). Chaque sous-tâche sera envoyée à un travailleur distinct.

2. Mise en correspondance (Map)

   La fonction «Map» applique un certain traitement à chaque élément.

   Les processeurs/nœuds traitent le sous-problème avec une fonction *Map* qui de façon générique prend en entrée un couple ($\text{clé}_1$, $\text{valeur}_1$) et produit en sortie un ensemble de couples ($\text{clé}_2$,$\text{valeur}_2$) :

   $$\texttt{map(clé}_1\texttt{, valeur}_1\texttt{)} \rightarrow \texttt{liste(clé}_2\texttt{, valeur}_2\texttt{).}$$

   Si on reprend notre exemple consistant à compter les mots, chaque travailleur prendra en entrée un couple comprenant le nom du document (fichier) et son contenu, et produira un ensemble de couples <mot, nb. d'occurrences>. Le programme 6, inspiré d'exemples en python[11, 52, 41], donne le pseudo-code de la fonction «Map».

```
def map_mots(fichier, texte):
   compte=defaultdict(int)
   liste_mots = texte.split()     # sépare le texte en mots
   for mot in liste_mots:         # pour chacun des mots
        compte[mot.lower()] +=1   # on compte les occurence
   return liste_couples
```

**Programme 6** – Code du «Map»

3. Tri et regroupement (Sort and shuffle)

   Cette étape consiste à mélanger tous les résultats (regroupe) et à trier les couples selon leur clé pour finalement former des groupes avec les couples ayant les mêmes clés. Chaque groupe ainsi formé est transmis à une processus de réduction.

   Dans notre exemple, les couples sont triés par ordre alphabétique (les mots sont la clé de triage) et des regroupements de mots identiques sont formés.

4. Réduction (Reduce)

   La fonction de réduction («Reduce») permet de réduire la forme finale des résultats. Elle prend en entrée un liste de couples (contenant généralement tous la même clé) et retourne souvent un seul couple ou du moins une liste de couple réduite.

   Pour notre exemple, cette phase consiste à réduire la sortie du «*Map*» (nombre d'occurrences par document) pour compter le nombre d'occurrences de chaque mot dans l'ensemble des documents. Chaque processus de réduction fait la somme des occurrences. Il reçoit en entrée une liste de couples (mot, occurences), ayant tous le même mot, et produit un couple (mot, total des occurrences).

   La fonction «Reduce» peut se modéliser de la façon suivante :

   $$\texttt{reduce(clé2,liste(valeur2))} \rightarrow \texttt{valeur2} \ .$$

   Le programme 7 donne le pseudo-code de la fonction «Reduce».

```
def reduce_mots(mot, liste_couples):
   compte=0
   for couple in liste_couples:
          compte +=couple[1]
   couple_final = [mot, compte]
   return couple_final
```

**Programme 7** – Code du «Reduce»

Il n'est pas nécessaire d'attendre que tous les «*Map*» soient terminés pour débuter la phase de réduction. Cette dernière peut débuter aussitôt que les premiers résultats de la mise en correspondance sont disponibles. Cela peut fonctionner sous forme de pipeline.

Dans les sections qui suivent, nous donnons quelques exemples d'utilisation de la construction «Map/Reduce» dont notre exemple pour compter des mots et la multiplication matrice/vecteur qui est le calcul utilisé par l'algorithme «*PageRank*» de Google.

## B.4 Exemple 1 : la température des villes

Soit cinq fichiers contenant des noms de villes (les clés) et des températures mesurées (les valeurs). Un fichier contient les informations montrées à la figure B.1. Il est important de noter que les villes peuvent se répéter dans un même fichier.

```
Toronto  20
Montréal 25
New York 22
Rome     33
..
Montréal 22
Rome     34
...
```

**Figure B.1** – Fichier de températures par ville.

Nous voulons extraire de ces fichiers la température maximale pour chacune des villes en utilisant la construction «Map/Reduce».

La première étape consiste à séparer le calcul. Le plus simple consiste à démarrer le calcul sur cinq sites, chacun traitant les données d'un des fichiers de températures. Chaque fonction «*Map*» parcoure les données et retourne la température maximale pour chaque ville. La figure B.2 donne un exemple de résultats pour chaque fonction «*Map*».

```
Map 1 : <Toronto, 20>, <Montréal, 25>, <New York, 22>, <Rome, 34>,
Map 2 : <Toronto, 23>, <Montréal, 26>, <New York, 32>, <Rome, 31>,
Map 3 : <Toronto, 21>, <Montréal, 29>, <New York, 27>, <Rome, 24>,
Map 4 : <Toronto, 19>, <Montréal, 15>, <New York, 28>, <Rome, 29>,
Map 5 : <Toronto, 30>, <Montréal, 28>, <New York, 33>, <Rome, 35>,
```

**Figure B.2** – Résultats après le «*Map*».

Les cinq résultats sont alors triés et regroupés par ville, et envoyés à quatre fonctions de réductions (une par ville ; car nos fichiers contiennent les températures pour seulement quatre villes). Chaque fonction de réduction retourne une seule valeur par ville. Le résultat est donné à la figure B.3

```
<Toronto, 30>, <Montréal, 29>, <New York, 33>, <Rome, 35>,
```

**Figure B.3** – Résultat après la réduction.

## B.5 Exemple 2 : Multiplication d'une matrice par un vecteur

Cet exemple est présenté par Hudelot et Behmo [28] dans un cours en ligne (OpenClassroom).

La multiplication de matrice par un vecteur (comme la multiplication de matrices en général) est un exemple standard de calcul parallèle trivial. Il faut savoir aussi que la multiplication de matrice

est une opération nécessaire au calcul du fameux *PageRank*, utilisé par Google pour ordonnancer les résultats d'une recherche sur le Web. En fait Google a conçu la construction *MapReduce* en grande partie pour résoudre ce problème. Dans ce cas, la taille de la matrice et du vecteur représente le nombre de pages web indexées. Finalement, la multiplication d'une matrice par un vecteur est une opération très commune dans les algorithmes utilisées en sciences des données et même, de façon générale, en calcul scientifique.

Cet exemple est intéressant en ce sens qu'il est plus complexe dans ce cas de structurer les données en forme de `<clé, valeur>`.

Nous voulons donc multiplier une matrice $A$ de grande taille $(n \times n)$ par un vecteur $v$ de taille $n$. Il s'agit donc de calculer

$$A \times v = x$$

avec

$$x = (x_1, ..., x_n)$$

et

$$x_i = \sum_{j=1}^{n} a_{ij} \times v_j$$

Dans ce cas particulier, la question qui se pose est comment représenter la matrice $A$ pour la présenter en entrée de la construction «*Map/Reduce*».

Très souvent, pour ce type de problèmes, nous sommes en présence de matrices creuses dans lesquelles on ne représente pas les zéros. Ici, nous allons donc considérer que la matrice $A$ est emmagasinée sous la forme de triplets $(i, j, a_{ij})$ (les coordonnées sont explicites). De même, le vecteur $v$ est emmagasiné sous la forme de paires $(j, v_j)$. Cette représentation permet de résoudre le problème d'organisation des données.

L'autre difficulté pour ce problème est la taille du vecteur $v$. En particulier, deux cas vont devoir être considérés selon la taille du vecteur. Soit le vecteur en entier peut tenir dans la mémoire du nœud de calcul soit il ne tient pas. Dans ce document nous parlons seulement du cas où il tient dans la mémoire d'un seul site sachant que dans le second cas il suffit de découper le vecteur et ensuite d'appliquer la même stratégie que celle que nous allons présenter. Je vous réfère à l'article de Hudelot et Behmo [27, 28] pour plus de détails.

Dans le cas où le vecteur $v$ tient dans la mémoire d'un nœud de calcul, l'opération «*Map*» prendra en entrée une ligne de la matrice et le vecteur $v$. utilisera comme clé en entrée un numéro correspondant à une ligne de la matrice ainsi que le

prendra en entrée est relativement simple à écrire. comme en entrée le vecteur $v$ en entier et un élément non vide de la matrice, c'est-à-dire un triplet $(i, j, a_{ij})$. En effet, pour chaque élément de la matrice, l'opération «*Map*» va juste générer la paire $(i, a_{ij} \times v_j)$. en On a donc choisi de prendre comme clé pour «*Map*», un numéro correspondant à une ligne de la matrice. C'est plutôt logique si on se rapporte à la formule ci-dessus car on somme sur les lignes.

Comme pour WordCount, nous pouvons utiliser notre baguette magique et l'opération SHUFFLE and SORT regroupe toutes les valeurs associées à la même clé $i$ dans une paire $(i, [a_{i1}v_1, ..., a_{in}v_n])$.

L'opération «*Reduce*» est donc aussi très évidente, il suffit de faire la somme de toutes les valeurs associées à une clé donnée.

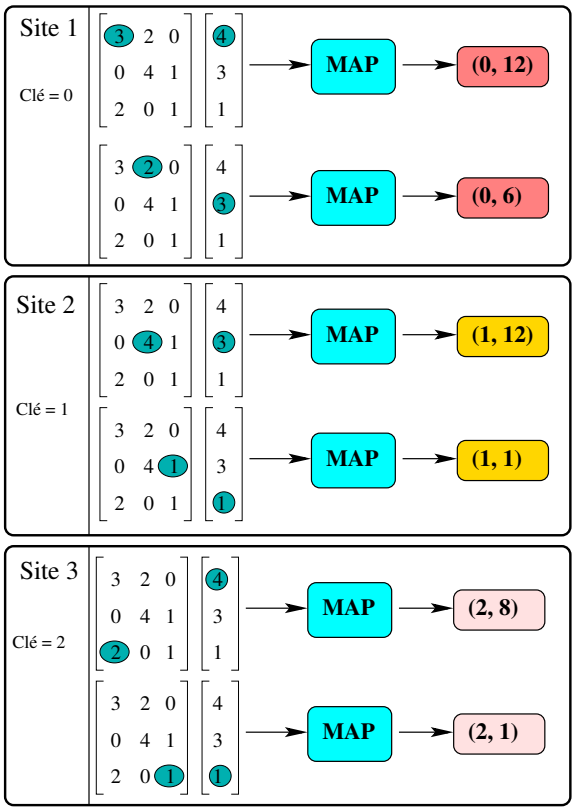



**Figure B.4** – Illustration de l'opération «*Map*» pour un exemple de multiplication d'une matrice par un vecteur.

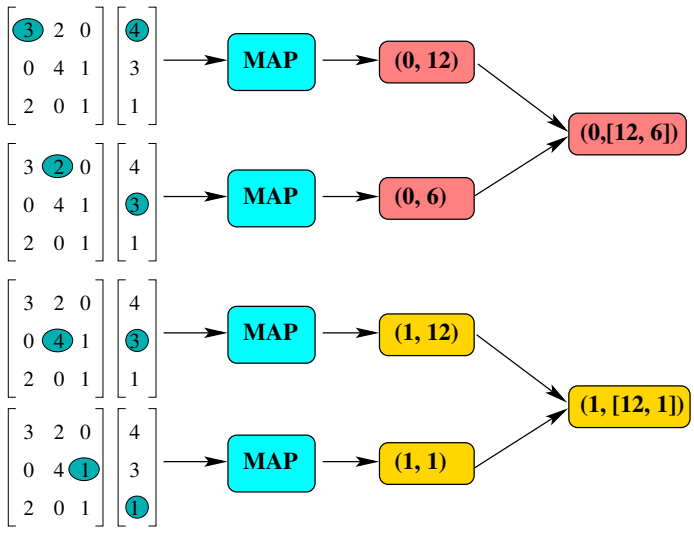



**Figure B.5** – Illustration de l'opération SHUFFLE pour un exemple de multiplication d'une matrice par un vecteur.

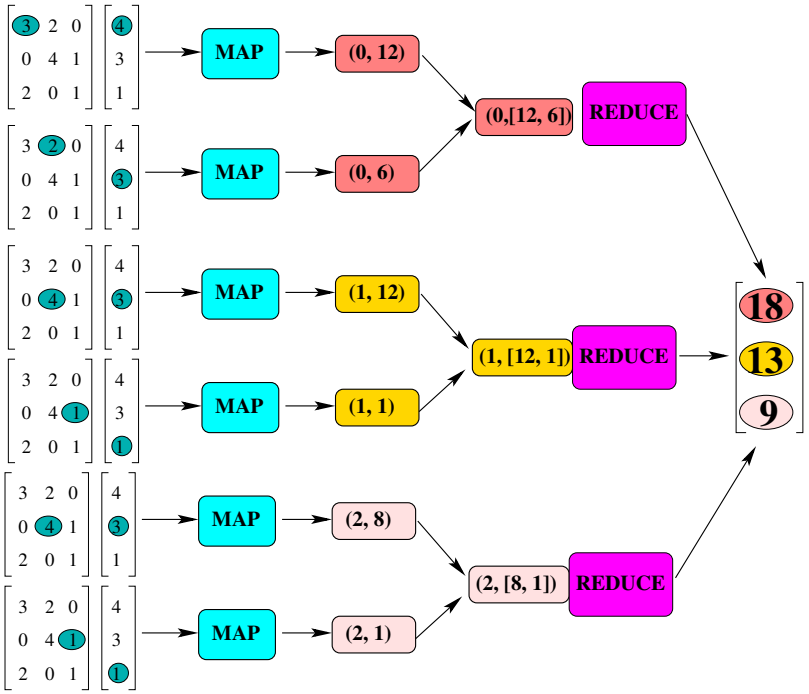



**Figure B.6** – Illustration de MapReduce pour un exemple de multiplication d'une matrice par un vecteur.

## B.6 Exemple 3 : Compter l'occurrence des mots dans un texte

Revenons avec notre exemple de programme qui compte le nombre de mots dans un ou des documents. Soit le texte suivant qui est divisé dans quatre documents :

*Document 1 :* `Bienvenue dans Hadoop`
*Document 2 :* `Hadoop est bon pour le nuage`
*Document 3 :* `Hadoop est bon pour le forage`
*Document 4 :* `Hadoop est bon bon bon`

Ce texte sera traité de la façon suivante :

1. Le texte est d'abord séparé en quatre sous-textes. La division évidente est de les séparer par document.

2. Chaque sous-texte est envoyé à un «*Map*» différent qui comptera le nombre de mots.

   Les résultat de chacun est donnée à la figure B.7.

```
MAP 1 : <Bienvenue, 1>, <dans, 1>, <Hadoop, 1>
MAP 2 : <Hadoop, 1>, <est, 1>, <bon, 1>, <pour, 1>, <le, 1>, <nuage, 1>
MAP 3 : <Hadoop, 1>, <est, 1>, <bon, 1>, <pour, 1>, <le, 1>, <forage, 1>,
MAP 4 : <Hadoop, 1>, <est, 1>, <bon, 3>
```

**Figure B.7** – Résultat après le Map.

3. L'étape suivante consiste à trier et regrouper les couples. Ainsi toutes les clés identiques seront expédiées à un processus de réduction distinct.

4. La réduction consiste à combiner les valeurs identiques pour compter le nombre total d'occurrences des mots dans le texte. Le résultat est illustré à la figure B.8

```
<Bienvenue, 1>, <bon, 5>, <dans, 1>,  <est, 3>,  <forage, 1>, <Hadoop, 4>,
<le, 2>, <nuage, 1>, <pour, 2>
```

**Figure B.8** – Résultat après le Reduce.

La figure B.9 illustre tout le processus du fonctionnement du «*Map/Reduce*» pour cet exemple.

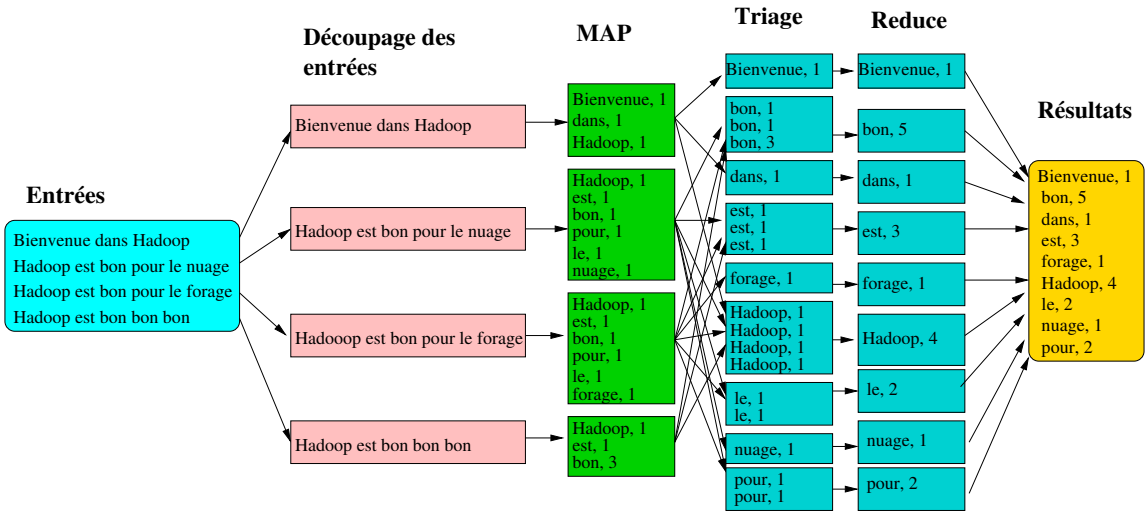



**Figure B.9** – Fonctionnement du map/reduce.

Les programmes 8, 10 et 11 introduisent des exemples de code implantant un «MapReduce» pour compter des mots. Le programme 8 implante une version purement séquentiel du «MapReduce» afin de compter le nombre d'occurrences de mots. Les programmes 10 et 11 utilisent la bibliothèque présentée au programme 9 et deux formes de parallélisme distinct de Python. Le programme 10 est basé sur les groupes de fils (Threadpool) et le programme 11 sur les processus.

```python
import ...
#
# Fonction qui organise l'occurence de chaque mot
#    Entrée : fichier contenant le texte (clé)
#    Sortie : liste de couples (mot, nb occurences)
#
def map1_mots(fichier):
    mon_fichier = open(fichier, "r")  # le fichier (clé)
    texte = mon_fichier.read()        # le contenu (valeur)
    mon_fichier.close()
    liste_couples = []                # liste de couples (mot, compte)
    couple = []
    liste_mots = texte.split()        # sépare le texte en mots
    for mot in liste_mots:            # pour chacun des mots
            couple = [mot, 1]
            liste_couples.append(couple)
    return liste_couples
#
# Fonction réduit la liste pour qu'un mot apparaisse une seule fois
# Entrée : mot, list d'occurence de ce mot)
# Sortie : couple comprenant (mot, nb occurences total)
#
def reduce1_mots(liste_couples):
    compte=defaultdict(int)
    for couple in liste_couples:
    compte[couple[0].lower()] +=1
    return compte
#
# Programme maitre qui distribue le travail
#
def maitre():
    # Étape 1 : mise en correspondance (MAP)
    liste_couples =  map1_mots("texte")
    # Étape 2 : trie... non nécessaire en séquentiel...
    # Étape 3 : réduction des données (reduce)
    compte_mots = reduce1_mots(liste_couples)
    # Impression du résultats
    for mot in compte_mots :
        print( "{:.<12} {: <20}".format(mot, compte_mots[mot]))
#-------------------------------------------------------------------
maitre()
```

**Programme 8** – MapReduce en séquentiel pour compter les mots

```python
import sys
from itertools import groupby
from operator import itemgetter
from collections import defaultdict
from multiprocessing.pool import ThreadPool
from multiprocessing import Process
from multiprocessing import Queue
#-----------------------------------------------------------------------
# Fonction qui compte l'occurence des mots dans un texte
#   Entrée : fichier contenant le texte
#   Sortie : liste de couples (mot, nb occurences)
#
def map_mots(fichier):
    mon_fichier = open(fichier, "r")  # Le fichier  (clé)
    texte = mon_fichier.read()        # le contenu (valeur)
    mon_fichier.close()
    liste_couples=[]                  # création d'une liste (mot, compte)
    compte=defaultdict(int)           # création d'un dictionnaire pour compter
    liste_mots = texte.split()        # sépare le texte en mots
    for mot in liste_mots:            # pour chacun des mots
        compte[mot.lower()] +=1    # on compte les occurence
    # On recopie la liste
    liste_couples += [[k,v] for k, v in compte.items()]
    return liste_couples
#-----------------------------------------------------------------------
# Fonction intermédiaiare qui trie et regroupe les mots identiques
#   Entrée : liste de couples (mot, nb occurences)
#   Sortie : liste de couples (mot, list d'occurence de ce mot)
#
def inter(liste_couples):
    # On trie la liste
    liste_triee = sorted(liste_couples, key = lambda x: x[0])
    # On regroupe par mot
    dic = {}
    for k, v in groupby(liste_triee, itemgetter(0)):
           dic[k] = list(v)
    return dic
#-----------------------------------------------------------------------
# Fonction réduit la liste pour qu'un mot apparaisse une seule fois
#    Entrée : mot, list d'occurence de ce mot)
#    Sortie : couple comprenant (mot, nb occurences total)
#
def reduce_mots(mot, liste_couples):
    compte=0
    for couple in liste_couples:
        compte +=couple[1]
    couple = [mot, compte]
    return couple
```

**Programme 9** – Bibliothèque MapReduce pour les exemples 2 et 3

```
#*****************************************************************************
# Programme qui implante un map reduce pour compter le nombre de mots dans
# plusieurs fichiers.
# Pour y parvenir, il utilise un groupe de fils (Threadpool)
#*****************************************************************************

import ...

#  Liste qui contiendra les couples (mots, nb d'occurrences)
res1=[]
#
#  Fonction de rappel pour accumuler les résultats du groupe de fils
#
def accumuler(resultat):
    res1.append(resultat)
#-----------------------------------------------------------------
# Programme maitre qui planifie le map/reduce
#
def maitre():
    res = []
    #-------------------------------------------------------------
    # Étape 1 : préparation des données
    #
    fichiers = ["ex1", "ex2", "ex3", "ex4"]
    numOfThreads = 4
    pool = ThreadPool(numOfThreads)
    #-------------------------------------------------------------
    # Étape 2 : on effectue le map dans le groupe de fils (threadpool)
    #
    results = pool.map(map_mots, fichiers)
    for result in results: # on met le résultat dans un seul tableau
        res.extend(result)
    #-------------------------------------------------------------
    # Étape 3 : on tri et regroupe les résultats par mots
    #
    res2 = inter(res)
    #-------------------------------------------------------------
    #  Étape 4 : on regoupe par mots avec le groupe de fils
    #
    for mot, couple in res2.items() :
        pool.apply_async(reduce_mots, (mot, couple), callback=accumuler)
    pool.close()
    pool.join()
    #
    # Impression du résultat
    #
    for r in res1 :
            print( "{:.<12} {: <20}".format(r[0], r[1]))
#-----------------------------------------------------------------
maitre()
```

**Programme 10** – MapReduce pour compter les mots avec un groupe de fils (threadpool)

```python
#*****************************************************************************
# Programme qui implante un map reduce pour compter le nombre de mots dans
# plusieurs fichiers.
# Pour y parvenir, il utilise des processus pour le calcul.
#
#*****************************************************************************
import ...

#
#  Fonction chapeau servant à récupérer les résultats
#
def travailleur(queue, f, *argv):
    queue.put(f(*argv))
#-------------------------------------------------------
# Programme maitre qui planifie le map/reduce
#
def maitre():
    q_res = Queue()
    map_res = []
    map_pcs = []
    #-----------------------------------------------
    # Étape 1 : on prépare les données
    fichiers = ["texte", "texte1", "texte2", "texte3"]
    #-----------------------------------------------
    # Étape 2 : on fait la mise en correspondance ("map") grâce à des processus
    for fichier in fichiers :
        p = Process(target=travailleur, args=(q_res, map_mots, fichier))
        map_pcs.append(p)
        p.start()
    # On collecte les résultats et on attend que les processus terminent
    for p in map_pcs:
        map_res.extend(q1.get())
        p.join()
    #-----------------------------------------------
    # Étape 3 : on tri et regourpe les résultats par mots
    res2 = inter(map_res)
    #-----------------------------------------------
    #  Étape 4 : On lance la réduction
    #
    redu_res=[]
    redu_pcs = []
    # On lance les processus
    for mot, couple in res2.items() :
        p = Process(target=travailleur, args=(q_res, reduce_mots, mot, couple))
        redu_pcs.append(p)
        p.start()
    # On récupère les résultats et on attend la fin des processus
    for p in redu_pcs:
        redu_res.append(q_redu.get())
        p.join()
    # impression du résultat
    for r in redu_res :
        print( "{:.<12} {: <20}".format(r[0], r[1]))
#-----------------------------------------------
maitre()
```

**Programme 11** – MapReduce pour compter les mots avec un groupe de fils (threadpool)

# Annexe C

# Les protocoles par battements de cœurs, vagues ou commérages

Les trois types de protocoles ou algorithmes présentés ici sont similaires en ce sens qu'ils communiquent tous avec des «voisins» afin de diffuser de l'information ou de prendre des décisions. Dans plusieurs situations, ces protocoles/algorithmes sont tellement similaires que le même vocable sert parfois à désigner des algorithmes distincts.

## C.1 Algorithmes/protocoles par battements de cœur

Une certaine distinction est occasionnellement sous-entendue entre protocoles et algorithmes par battements de cœur.

Dans façon générale, un algorithme pas battements de cœur décrit un signal périodique généré par le matériel ou le logiciel **pour indiquer un fonctionnement normal** ou **pour synchroniser d'autres composants du système**.

### C.1.1 Protocole de type battements de cœur pour la tolérance aux pannes

Soit une grappe d'ordinateurs comprenant des serveurs et des clients travaillant conjointement sur des tâches communes telles que la gestion d'une base de données distribuée, la planification, le stockage des données et la recherche d'information. Dans un tel environnement, la détection d'une panne est un défi qui doit être adressé. Plusieurs solutions existent pour ce faire dont les protocoles par battements de cœur.

Un protocole par battements de cœur, selon Wikipedia [62] et Fowler [15, 16], permet de connaître la disponibilité d'un serveur en émettant périodiquement des messages à tous les autres serveurs. Ce procédé est particulièrement utile lorsqu'on utilise une grappe de serveurs se partageant la tâche d'emmagasiner de l'information. La détection rapide des pannes est fort importante afin de s'assurer que des actions correctrices seront mises en œuvre.

Afin de négocier et surveiller la disponibilité d'une ressource, un message de type «battements de cœur» est transmis à tous les sites à intervalle régulier, exemple à toutes les secondes, indiquant

que le processus est toujours actif. Un protocole par battements de cœur utilise de tels messages pour chacun des processus et est caractérisé par une expansion rythmique et une contraction de l'information. Un processus muni de ce protocole travaille comme un cœur qui bat.

Si un processus ne reçoit pas ce message après un temps pré-déterminé (un intervalle de quelques battements de cœur), le site émetteur fautif est considéré en panne. Les messages de type battements de cœur sont transmis sans interruption et à période fixe par le site émetteur pendant toute sa durée de fonctionnement (du démarrage jusqu'à l'arrêt). Lorsque la destination identifie un manque aux messages de la part du site d'origine, elle conclut à une panne, un arrêt ou une indisponibilité de la part du site d'origine. Ce type de messages est généralement utilisé pour les systèmes hautement tolérant aux pannes.

Comme les battements de cœur sont destinés à déterminer l'état de santé d'un site, il est important que ce protocole, de même que le «réseau» sous-jacent, soient aussi fiables que possible. Une fausse alarme est effectivement indésirable, selon la ressource gérée. Il est tout aussi primordial de pouvoir réagir rapidement à une panne réelle, d'où la nécessité de se munir d'un protocole fiable. Dans ce but, il est fréquent que le protocole par battements de cœur fasse appel à plus d'un médium de communication, tel Ethernet utilisant UDP/IP et un lien série.

Lorsque la taille de la grappe est restreinte, un unique serveur centralisé (de type chef/exécutants ou «leader/followers» [16]) suffit à gérer la détection des pannes. Cette approche présente cependant plusieurs problèmes lorsque la grappe devient relativement imposante, allant jusqu'à plusieurs milliers de nœuds. Ce serveur devient alors un goulot d'étranglement, pouvant parfois entraîner une latence importante, en plus d'être un unique point de défaillance. Si la grappe contient un très grand nombre d'ordinateurs, un protocole par commérages sera mieux adapté à la situation.

Les systèmes Hashicrop, Akka Actors, Cassandra (Apache), RAFT et Zookeeper appliquent ce type de protocole.

### C.1.2 Algorithme par battements de cœur : version 1

Dans cette version de l'algorithme par battement de cœur [5, 3, 9], présentée originalement par Andrews [5, 3], chaque processus envoie périodiquement des informations à ses «voisins» (envois de messages) et en reçoit d'eux en retour. Cet algorithme fonctionne par «tours». À chaque tour, chacun des processus collecte les données de ses voisins et les traite. Au tour suivant, tous les voisins sont informés des nouveaux résultats.

Ainsi, lors du premier tour, chaque processus connaît ses voisins. Au second tour chaque processus connaît ses voisins et les voisins de ces derniers. Ainsi à chacun des tours, l'horizon s'élargit d'un «hop». Après «D» tours, où D représente le diamètre du réseau, chaque processus possède les informations de tout le réseau. Ce mode de communication propage l'information par l'intermédiaire des voisins immédiats (dissémination ou exploration progressive de l'horizon). C'est une forme de diffusion mais par étapes dans lesquelles l'information est envoyée et reçue périodiquement.

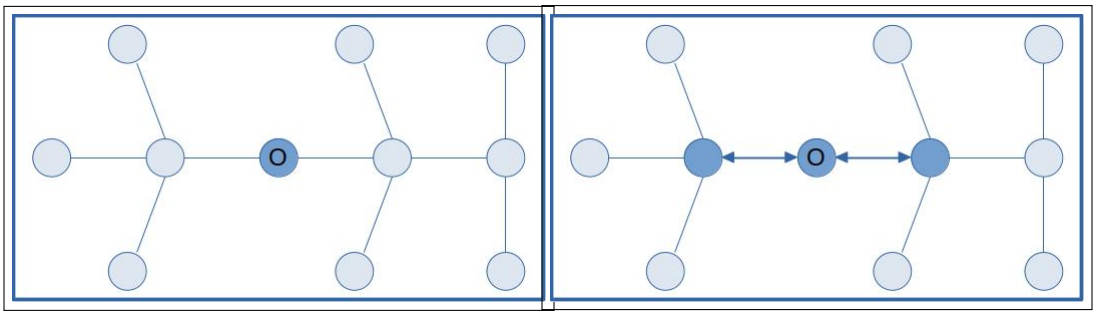
Le programme C.1 décrit le fonctionnement général du mode de communication dit «par battements de cœur». La figure C.1 illustre comment l'information est disséminée à tous les membres du réseau par l'intermédiaire des voisins.

Les algorithmes par battements de cœur sont en mesure de diffuser ou collecter de l'information sur un système distribué.

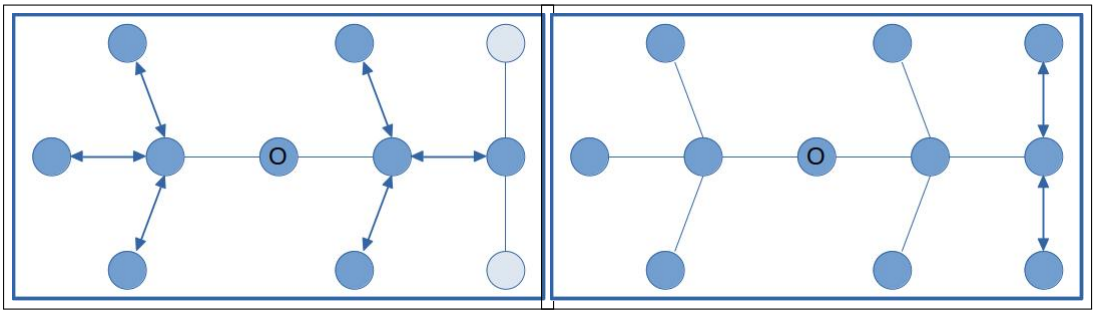
Par exemple, l'algorithme «FloodMax» en est un d'élection qui détermine le «nouvel élu» en sélectionnant le processus avec le plus grand identificateur dans le système. Ainsi, à chaque passage (tour !) de cet algorithme, chacun des processus fournit à ses voisins le plus grand identificateur

```
process Travailleur[i = 1 to nbTravailleurs]
{
   Déclarations des variables locales;
   Initialisation;
   while (tour < D)
   {
      send valeurs to "voisins immédiats";
      receive valeurs from "voisins immédiats";
      mise à jour;
      // prochain tour
      tour := tour + 1
   }
}
```

**Programme C.1** – Algorithme de type battements de coeur




**(a)** Le site «O» veut transmettre l'information. **(b)** Étape 1 : transmission aux voisins immédiats.




**(c)** Étape 2 : transmission aux voisins suivants. **(d)** Étape 3 : transmission aux derniers voisins.

**Figure C.1** – Algorithme par battements de cœur

connu à cet étape et lui-même reçoit en retour cette même information de ces voisins. Suite à la réception des nouvelles données, il met à jour son état.

Ce type d'algorithme est particulièrement utile lorsque de multiples travailleurs doivent s'échanger en continu de l'information afin de compléter un traitement (algorithmes itératifs). Il s'avère très avantageux notamment dans les cas où les données sont réparties entre les travailleurs qui ont aussi chacun la responsabilité de mettre à jour leurs données afin de poursuivre leurs calculs lorsque ceux-ci dépendent des valeurs détenues par ses voisins immédiats.

Cette type d'algorithme implante aussi une forme de barrière. Elle est à privilégier lorsque les processus ont une relation de type pair-à-pair (peer-2-peer).

**Exemple : découverte de la topologie d'un réseau [3]**

Le problème soumis dans cette section consiste à découvrir la topologie d'un réseau formé d'un certain nombre de processeurs connectés par des canaux de communication bi-directionnels. Chacun des processeurs ne peut que communiquer avec ses voisins et et ce uniquement pas les canaux le reliant à ceux-ci. L'enjeu est donc, que chacun des processeurs, découvre (sous ces conditions) la topologie du réseau entier (supposée constante pendant tout le calcul).

Dans le but d'illustrer le fonctionnement de ce type d'algorithme, on fait l'hypothèse qu'il n'y a aucun processeur central connu de tous pour recevoir l'information.

Le principe de l'algorithme est le suivant. Initialement, un nœud $p$ connaît les canaux vers ses voisins. Si $p$ envoie un message à ses voisins les interrogeant sur leurs propres voisins, il sera instruit des canaux entre ses voisins et lui, mais aussi de ceux reliant ses voisins à leurs propres voisins, donc il connaîtra la topologie du réseau jusqu'à deux canaux distants. À l'étape suivant, la seconde, il aura obtenu les informations lui permettant de connaître la topologie du réseau jusqu'à trois canaux de distance. Ainsi, après un certain nombre d'étapes, chaque processeur aura identifié la topologie entière du réseau. Si le diamètre du réseau est de «D», alors la topologie du réseau sera connue de tous après «D» étapes, comme l'illustre le programme

Le souci majeur de cette solution est l'ignorance de chacun des nœuds sur la valeur du diamètre $D$ du réseau, celle-ci ne leur étant pas connu à l'avance. De plus, elle n'est pas optimale en terme de nombre de messages échangés. En effet, certains processeurs sont situés de telle façon dans le réseau (par exemple «au centre» du réseau), qu'ils seront instruits de la topologie entière du réseau en un nombre d'étapes «$k$», tel que $k < D$. Pourtant, lors des dernières étapes, ceux-ci n'apprendront rien de neuf ($topologie = newtop$), mais continueront à échanger de l'information. Une solution est proposée par Andrews [3] pour régler ces problèmes.

### C.1.3 Algorithme par battements de cœur : version 2 [9]

Un algorithme de type battements de cœur ou systolique est exécuté sur un tableau de cellules (Systolique) où chacune d'elles a la capacité de faire certains traitements et de communiquer localement. C'est un cas particulier d'algorithmes synchrones dans lequel chaque processus/processeur échange des données mais seulement avec ses voisins adjacents, lesquelles circulent de façon synchrone à travers le tableau. Les noms «battements de cœur» et «systolique» sont choisis bien sûr, par analogie avec le pompage régulier du sang par le cœur dans un organisme vivant mais aussi parce que systolique est un mot grecque signifiant «contraction et battements de cœur». Notons que la version asynchrone d'un algorithme par battements de cœur est appelé «algorithme par phase».

```
canaux voisins[1..n](topologie[1:n, 1:n]:bool)

process p ()
  var canaux-valides[1:n] : bool

  # initialisé de façon à ce que canaux[q] est vrai si q is voisin de p]
  var topologie[1:n, 1:n] : bool := ([n*n] false) # vue locale du réseau
  var r:int:=0
  var newtopologie[1:n,1:n] : bool

  topologie[p] := links # initialise avec sa connaissance locale

  while (r < D)
    # Envoi la connaissance locale du réseau aux voisins
    fa q := 1 to n st canaux-valides[q] -> send to  voisins[q](topologie)
    af

    # Réception des informations des voisins
    fa q := 1 to n st canaux-valides[q] -> receive from voisins[p](newtopologie)

    # Mise à jour de notre topologie
    top := top or newtop
    af
```

**Programme C.2** – Algorithme de découverte du réseau

## C.2 Algorithme par vagues

Un algorithme par vagues [19, 51, 39, 40] se révèle des plus similaires, voire identiques dans certains cas aux algorithmes par battements de cœur [3]. En effet, tout comme dans ces derniers, l'information s'y propage en «vagues» d'un nœud vers ses voisins, puis vers les voisins des voisins et ce, jusqu'à ce qu'elle atteigne tous les processus.

Les algorithmes par vagues réfèrent donc à une classe d'algorithmes dans laquelle un processus initiateur démarre le calcul, lequel à son tour déclenche d'autres actions dans les processus non initiateurs adjacents. En un temps fini, lorsqu'une condition prédéterminée est satisfaite localement, chacun des processus termine son exécution. La chaîne d'événements résultante ressemble à la croissance et la décroissance des vagues à la surface d'une étendue d'eau calme, d'où le nom d'algorithme par vagues.

Les algorithmes par vagues constituent une généralisation de ceux proposés pour la détection de prédicats et la prise de décision basée sur l'exploration d'états.

Le calcul par vagues doit satisfaire les critères suivants :

- Un ou plusieurs nœuds initiateurs démarrent le traitement ;
- Le traitement dans un nœud non initiateur est déclenché par un traitement dans un nœud adjacent ;
- La vague doit se terminer i.e. :
  - Chaque nœud termine localement le traitement lorsqu'un but local prédéfini est atteint (événements locaux dans chaque processus qui précède la décision finale qui sera prise par l'initiateur) ;
  - L'initiateur de la vague prend une décision afin de terminer le traitement globalement. Celle-ci est précédée par un événement dans chaque processus non initiateur. Cette fin

du traitement se produit après un nombre fini d'étapes.

**Exemple : diffusion d'un message sur un anneau**

Soit un processus désirant diffuser un message M à tous les processus sur un anneau unidirectionnel. L'initiateur envoie un message à son voisin, qui à son tour le transfère à son voisin et ainsi de suite. Quand l'initiateur reçoit le message M, il arrête le transfert du message et la vague se termine. C'est un événement décisionnel à la suite duquel l'initiateur peut débuter une autre vague avec un objectif différent.

**Exemple : synchronisation de type barrière**

Soit un calcul par étapes. Un processus initiateur démarre l'étape 0 et tous les processus exécutent ensuite cette même étape. La contrainte d'une synchronisation de type barrière est qu'aucun processus ne doit démarrer une étape i (i > 0) avant que tous les processus aient terminé l'étape i-1. La fin d'un étape constitue l'événement décisionnel dans cet algorithme.

**Exemple : propagation d'informations avec rétroaction**

La propagation d'informations avec rétroaction consiste en une méthode de diffusion d'un message munie d'un mécanisme qui confirme sa réception. Ce dernier sert notamment à réveiller des nœuds et à démarrer l'exécution d'un nouveau traitement. L'algorithme suivant est décrit par Ghosh [19].

Soit un réseau $G = (V, E)$ dans lequel $V$ représente l'ensemble de nœuds et $E$ un ensemble de liens (arcs) bidirectionnels. Un nœud initiateur $i \in V$ désire envoyer un message $M$ à tous les autres nœuds et recevoir une confirmation. Dans cet l'algorithme, présenté au programme C.3, un nœud retarde l'envoi d'une copie du message $M$ à son parent (l'équivalent d'un accusé de réception) jusqu'à ce qu'il reçoive une copie de $M$ de tous les nœuds enfants.

```
1  program propagation {pour l'initiateur i}
2      cpt : integer
3      N(i): ensemble des voisins du processus i
4
5      foreach v in N(i)
6           send M to v;
7      cpt := |N(i)|
8      while  count != 0
9          receive M
10         cpt: = count - 1
11
12 {Programme pour un noeud j!= i non-initiateur}
13     if message M est reçu de E ->
14              parent := E
15              send M à chaque voisin sauf le parent;
16              cpt := |N(j)|;
17     [] cpt>0 and message M est reçu ->
18              cpt := cpt-1
19     [] cpt = 0  ->
20              send M to parent
21 fi
```

**Programme C.3** – Algorithme de propagation avec rétroaction

Pour chaque nœud, la condition $cpt = 0$ indique la fin de l'algorithme. C'est un confirmation que tous les nœuds ont bien reçu le message.

**Exemple : algorithme de diffusion**

Soit un réseau de $N$ processus fortement connectés numérotés de 0 à $N-1$. Chaque processus $i$ possède une valeur stable $s(i)$ qui lui est associée. Le but est de déterminer un algorithme dans lequel chaque processus $i$ peut diffuser sa valeur $s(i)$. À la fin, chaque processus $i$ détiendra un ensemble de valeur $V.i = \forall_k : 0 \leq k \leq N-1 : s(k)$.

Initialement, $V.i = s(i)$. Pour compléter la diffusion, chaque processus $i$ (1) envoie périodiquement $V.i$ à tous ses voisins et (2) reçoit de tous ses voisins $j$ leur propre valeur de $V.j$. L'opération ressemble au pompage du sang dans le cœur d'où le nom d'algorithme par battements de cœur généralement associé à cet algorithme.

Il y a deux complications importantes à considérer. D'abord, prévoir comment l'algorithme termine ainsi que sa complexité en termes de messages échangés. Pour être efficace, il est inutile d'envoyer $V.i$ s'il n'a pas changé depuis le dernier envoi. D'ailleurs, même dans cette éventualité ($V.i$ a changé), il suffit d'envoyer seulement les modifications. Pour y parvenir, on associe deux ensembles de valeurs à chacun des processus $i$, soit l'ensemble $V.i$, qui contient l'ensemble des valeurs collectées, et l'ensemble $W.i$ qui représente la dernière valeur de $V.i$ envoyée aux voisins. Dénotons par $(i, j)$, le canal entre les processus $i$ et $j$. L'algorithme prend donc fin lorsqu'il n'y a plus aucune nouvelle valeur à transmettre entre les processus et que tous les canaux sont vides (plus aucun message en transit). Le programme C.4 présente l'algorithme.

```
1  program diffusion
2      // Diffusion initiale par le processus i
3      V.i, W.i: set of values;
4      V.i = {s(i)}, W.i = {} {et les canaux de commuication sont tous vides}
5      do  V.i != W.i ->
6                 send (V.i \ W.i) à chaque voisin;
7                 W.i := V.i
8      []  ! empty (k, i) ->
9                 receive X from channel (k, i);
10                V.i := V.i union X
11     od
```

**Programme C.4** – Algorithme de propagation avec rétroaction

## C.3 Protocole par commérages (bavardages) (Gossip)

Un protocole par commérages (*gossip protocol*) [12, 17, 37, 36, 47, 64, 61, 31, 7, 17, 50, 45, 46, 32, 56], aussi appelé protocole «épidémique» ou par «bavardages», est un protocole servant à diffuser de l'information sur un réseau inspiré de la manière dont se transmettent les commérages entre personnes ou sur les réseaux sociaux ou encore qu'une épidémie se propage. Ce protocole résout le problème de diffusion même si chaque nœud communique seulement avec certains nœuds «voisins». En fait, c'est un algorithme distribué principalement utilisé dans les réseaux informatiques «pair-à-pair». La première formulation d'un protocole de bavardage est attribuée à Demers et al [12].

### C.3.1 Fonctionnement

Le protocole de communication par commérages est un protocole «pair-à-pair» dont le principe est relativement simple. Un nœud recevant une mise à jour désire la propager. Pour y parvenir, il sélectionne au hasard «V» nœuds voisins et leur envoie un message. «V» est appelé le taux de

### Comment se répand une rumeur

Le mode de fonctionnement de ce protocole est aussi comparé à la façon dont une rumeur se répand. Une personne «A» apprend une nouvelle rumeur. «A» appelle alors une personne «B» au téléphone pour lui transmettre la rumeur. Une fois cet appel terminé, «B» appelle une troisième personne «C» pendant que «A» en contacte une autre («D») avec pour objectif de répandre la rumeur. Ce processus continue jusqu'à ce que tout le monde en ait pris connaissance. La rumeur se répand donc à grande vitesse pour finalement atteindre chaque membre de la communauté, et cela sans la nécessité d'un coordonnateur central.
Il est manifeste que cette technique peut servir à diffuser rapidement de l'information sur un réseau.

### Virus et épidémiologie

Comment un virus se répand-il ? lorsqu'une personne est infectée par un virus, elle le transmet à un certain nombre d'individus au hasard. Le même mode de transmission s'applique également aux individus infectés. Ainsi le nombre de personnes infectées croit exponentiellement. L'épidémiologie étudie ce genre de tranmission dans l'espace et le temps. (théorie des épidémies).
Le protocole par commérage ou épidémique est basé sur cette théorie. Celle-ci montre mathématiquement qu'une épidémie, débutant sur un seul site, infectera éventuellement toute une population et cela en un temps proportionnel au logarithme de la taille de la population.

dispersion (*fanout*). Les nœuds «voisins» recevant cette nouvelle information font de même. Cette procédure, appelé un cycle ou tour, est répétée jusqu'à ce que l'information soit connue de tous.

C'est un protocole dont la base est simple et robuste. Il y a toutefois des subtilités à considérer dans son implantation. Aussi différents modes de fonctionnement ont été développés spécifiquement pour répondre aux besoins, aux nombres de cycles, à la direction des communications, aux choix des voisins, etc.

Notons d'abord que dans ce protocole, un nœud peut prendre un des trois états suivants :

- Infecté : c'est un nœud qui a reçu une mise à jour qu'il désire propager ;
- Prédisposé : c'est un nœud qui n'a pas reçu la mise à jour (il n'est pas infecté) ;
- Retiré : c'est un nœud qui a reçu la mise à jour mais qui ne désire plus la partager.
  Cet état est particulièrement complexe à gérer car il est difficile de déterminer quand un nœud doit arrêter de propager les mises à jour. Idéalement, un nœud devrait arrêter de propager de l'information quand tous ses «voisins» ont reçu la mise à jour.

Malgré son apparente simplicité, on se doit de distinguer les différents modèles qui répondent précisément aux décisions adoptées lors de la conception. Parmi celles-ci, on retrouve :

1. Le nombre de répétitions de la procédure (nombre de cycles ou tours) ;
   Deux modèles influencent le nombre de cycles du protocole :
   - le modèle anti-entropie (modèle SI)
     Le nom anti-entropie est utilisé car le protocole permet de réduire l'entropie entre les multiples copies, i.e. diminue la différence entre les copies. C'est un mode épidémique simple dans lequel un nœud est toujours soit «prédisposé», soit «infecté». Il n'y a pas d'état «retiré». Dans un protocole anti-entropie, un nœud infecté essaie de partager sa nouvelle information à chaque cycle. Un nœud ne partage pas seulement la dernière mise à jour, mais aussi toute l'information qu'il détient. Diverses techniques telles que les *checksum*, les listes à jour récentes, les arbres de Merkle, etc permettent à un nœud d'être au fait des différences entre l'état de deux nœuds avant de transmettre l'entièreté de l'information. Cette approche assure une dissémination éventuellement parfaite. Ce protocole ne termine pas. Il n'y a donc pas de limite au nombre de messages expédiés.
   - le modèle marchand de rumeurs (modèle SIR)
     C'est un mode épidémique complexe dans lequel un nœud peut être à l'état «prédisposé», «infecté» ou «retiré». Selon cette approche, un nœud envoie seulement les informations modifiées. Les cycles peuvent donc être plus fréquents qu'avec le modèle anti-entropie puisqu'ils demandent moins de ressources. Le modèle marchand de rumeurs répand les mises à jour rapidement tout en générant peu de trafic sur le réseau. À un certain moment, une rumeur est marquée comme retirée et n'est plus répandue. Grâce à cela, les nombre de messages échangés est limité. Toutefois, il est possible qu'une mise à jour n'atteigne pas tous les sites même si la probabilité en est faible. Le principal problème de cette approche est de déterminer quand un nœud passe à l'état «retiré».
2. La direction de l'information (le mode de communication)
   - Le mode *PUSH*
     Dans ce mode, les nœuds infectés sont ceux qui transmettent l'information à leurs voisins et infectent les nœuds prédisposés, et cela qu'ils le veuillent ou non.

Ce mode est très efficace quand il y a peu de mises à jour.

- Le mode *PULL*

  Dans ce mode, tous les nœuds demandent activement les mises à jour à leurs voisins. Comme les nœuds ne peuvent connaître à l'avance les nouvelles mises à jour, ils doivent demander constamment aux voisins.

  Ce mode est très efficace quand il y a un grand nombre de mises à jour.

- Le mode *PUSH-PULL* :

  Selon ce mode, les nœuds infectés transmettent l'information et les nœuds prédisposés demandent les mises à jour.

  Les nœuds s'échangent donc de l'information.

En présence d'une multitudes de mises à jour, le mode *PULL* est le meilleur choix car il est for probable qu'un nœud découvre rapidement un autre nœud possédant déjà les mises à jour. Lorsque le nombre de mises à jour est peu élevé, le mode *PUSH* s'avère le mieux adapté car il n'entraîne pas une surcharge de communication.

Durant la phase initiale, le mode *PUSH* est le plus efficace car les nœuds infectés ont nombreux. Transmettre les mises à jour s'avérerait donc peu efficace. Le mode *PULL* le devient lui lors de la dernière phase dans laquelle plusieurs nœuds sont infectés et qu'il est alors aisé d'obtenir une mise à jour. Pour la même raison, le mode *PUSH* est peu rentable dans la phase finale car alors les envois ne parviendraient quasiment qu'à des nœuds déjà infectés. C'est pourquoi d'autres modes ont été introduits, tels le mode «*Fisrt-Push-Then-Pull*», qui tentent de tirer avantage de ces caractéristiques.

Ces deux modes s'appliquent autant dans le modèle SI que dans le modèle SIR. Ainsi, pour le modèle SI, qui ne termine jamais, le mode *PUSH* implique que les nœuds envoient les mises à jour même si tous les autres nœuds les ont déjà reçues. Avec le mode *PULL*, les nœuds demandent constamment des mises à jour même s'il n'y a rien de nouveau. Concernant le modèle SIR, une information marquée «retirée» ne sera plus transmise sur le réseau, et cela peu importe le mode de communication utilisé.

3. Le «moment» du flux d'information

   Le «moment» du flux d'information se réfère au moment où les cycles démarrent. Il y a possibilités :

   - chaque nœud envoie / reçoit des informations à des moments précis (protocole synchrone) ;
   - chaque nœud envoie / reçoit des informations à tout moment (protocole asynchrone).

4. Le choix des voisins

   La technique utilisée pour choisir les «voisins» est importante. Elle influence la vitesse à laquelle l'information sera propagée sur le réseau et assure que tous les nœuds recevront bien cette information. Avec un bon choix de voisin, l'information sur le réseau deviendra éventuellement cohérente et cela très rapidement. Il est possible de faire appel à des modèles mathématiques pour identifier une borne supérieure au nombre de cycles requis pour une diffusion complète. Cette borne dépend non seulement de la manière de procéder quant au choix des voisins, mais aussi du taux de dispersion. Par exemple, si un nœud sélectionne au hasard qu'un seul voisin à chaque étape, $O(logN)$ étapes sont nécessaires pour que la mise à

jour atteigne tous les sites, où N est le nombre de nœuds du réseau.

Vu l'intérêt de désigner au hasard les voisins, il existe un service dédié qui fournit des interlocuteurs appelé «*peer samplign service*».

Il est primordial de noter que dans la plupart des implantations, un nœud ne possède qu'une vue partielle du réseau. Afin d'obtenir une vue globale et de sélectionner judicieusement ses «voisins», un nœud doit maintenir à jour continuellement la liste complète des membres du réseau. Par exemple, si un site tombe en panne, celui-ci doit impérativement être retiré de la liste. Pour un bon déroulement, un algorithme par bavardages doit se référer à une liste fiable de tous les membres, mais maintenir cette liste à jour est un enjeu en soi.

**Exemple de fonctionnement**

Soit un protocole par commérage asynchrone basé sur le modèle «marchand de rumeurs» et utilisant le mode *PUSH*. Celui-ci fonctionne de la façon suivante :

1. Événement de mise à jour (infection)

   Initialement un nœud veut partager de l'information avec les autres nœuds sur le réseau. Cela se produit soit lors d'un changement d'état, soit lorsqu'un nœud reçoit de l'information d'un autre nœud. Il passe alors à l'état «infecté».

2. Choix des voisins

   Aussitôt qu'il recevra la mise à jour, il deviendra «infecté» et commencera à disséminer l'information.

   Un nœud infecté essaiera aussitôt de transmettre de l'information périodiquement à un ou plusieurs voisins désignés aléatoirement. Il choisit donc périodiquement au hasard N ($N \geq 1$) nœuds. La valeur $N$ est appelée le taux de dispersion (*fanout*)

3. Dissémination

   Lorsque les $N$ voisins sont connus, le nœud leur envoie l'information.

4. Réception de l'information par les voisins

   Les destinataires reçoivent et traitent l'information. Si ces voisins sont «prédisposés», c'est qu'ils n'avaient pas encore reçu l'information, i.e. ils n'étaient pas infectés. Ils le deviendront alors et commenceront, à leur tour, à disséminer l'information. Ils transmettront donc l'information à $N$ voisins choisis au hasard.

5. Ces étapes sont répétées périodiquement par tous les nœuds du réseau de façon à disséminer l'information.

6. Ces répétitions se poursuivent jusqu'à ce que l'information soit diffusée sur tout le réseau.

   La diffusion se termine quand tous les nœuds sont à l'état «retirés». Un nœud «retiré» est celui qui connaît l'information mais qui ne la propage plus. Cela se produit, par exemple, lorsque tous ses nœuds voisins connaissent déjà l'information.

### C.3.2 Caractéristiques des protocoles par commérages

Il y a certes eu plusieurs tentatives pour formellement définir les protocoles par commérages mais il semble que celles-ci n'aient pas eu le succès escompté car il n'y a aucune définition standardisée pour exprimer ce qu'ils sont. Toutefois les propriétés recherchées pour de tels protocoles sont claires.

**Exemple de propagation**

Soit cinq nœuds d'un système distribué, tous initialement synchronisés, i.e. tous dans le même état. On suppose que le noeud 2 devient infecté (son état change). Il doit alors faire savoir aux autres que son état est devenu différent. Il sélectionne donc au hasard deux autres noeuds, 3 et 4, pour transmettre une copie de son état.
Les nœuds 3 et 4 sont alors avertis du changement et leur propre état est mis à jour. Ces deux nœuds choisissent à leur tour deux nœuds au hasard pour envoyer une copie de leur état. On peut donc prédire ce qui se produira ensuite. Même si parfois un nœud en sélectionne un déjà à jour, à la fin, tous sont au courant du changement et la surcharge est négligeable. Dans un tel système distribué, on communique aux nœuds tout changement survenu dans l'état de l'un d'eux.
Après un certain nombre d'étapes, tous les noeuds seront synchronisés. C'est là une implantation simplifiée d'un protocole de bavardage.

Les voici :

- La base du protocole implique des interactions périodiques entre deux sites «voisins» ;
- L'information échangée lors de ces interactions est de petite taille ;
- La sélection du nœud voisin doit être aléatoire ou au moins garantir une diversité de «voisins» ;
- Seulement de l'information locale est disponible à chaque nœud (les nœuds ne sont pas conscient de l'état global du système et agissent sur la base des connaissances locales seulement) ;
- La capacité de traitement entre chaque cycle est limitée ;
- Tous les nœuds exécutent le même protocole ;
- Lorsque des nœuds interagissent, l'état d'un nœud, ou des deux, change pour refléter l'état de l'autre. Par exemple, si «A» communique avec «B» juste pour mesurer le «round-trip time», ce n'est pas une interaction de commérage.
- On assume une communication fiable ;
- La fréquence des communications doit être basse par rapport à la latence des messages de façon à ce que le coût du protocole soit négligeable.

### C.3.3 Utilité des protocoles par commérages

Les protocoles par commérages se sont avérés très utiles pour maintes applications et sont particulièrement efficace pour :

- diffuser l'information et permettre ainsi à des processus de s'échanger les nouvelles mises à jour.
- l'agrégation d'informations, i.e. que des processus collaborent lors d'une évaluation en partitionnant les calculs entre eux ;

  Par exemple : lors de l'évaluation d'une moyenne, d'une somme ou de la plus grande valeur de données localisées sur différents sites.

- pour découvrir la topologie d'un réseau seulement par des interactions locales.
- permettre l'auto-organisation dans les réseaux informatiques sans coordonnateur central, aussi longtemps que les processus sont suffisamment actifs et connectés.
- maintenir la cohérence dans les bases de données ayant de multiples copies sur plusieurs sites (plusieurs centaines de sites dans certains cas). Les protocoles par commérages servent à maintenir un modèle de cohérence dit détendu, i.e. que la cohérence sera éventuellement atteinte pour une grand nombre de processus sur un réseau.
- la conception de réseau superposé (réseau virtuel au-dessus d'un réseau existant).
- déterminer l'appartenance d'un nœud à un groupe.
- la détection des pannes.

### C.3.4 Systèmes utilisant les protocoles par commérages

Les protocoles par commérages sont utilisés dans plusieurs logiciels tels que :

- Cassandra

  Développée par Apache, Cassandra est une base de données distribuée NoSQL «*open source*». Elle offre des capacités de mise à l'échelle linéaire et la tolérance aux pannes sans compromettre la performance. Un protocole par commérages est utilisé pour optimiser le routage ainsi que pour sa tolérance aux pannes. Chaque nœud choisit trois pairs actifs au hasard pour bavarder. À chaque seconde, il informe ses pairs de son état et de celui des nœuds qu'il connaît par bavardage. De cette façon, tous les nœuds sont renseignés rapidement sur l'état de tous les autres nœuds de la grappe.

- *Amazon Simple Storage Service (S3)*

  S3 est un service infonuagique qui se fie sur un algorithme par commérages pour améliorer sa tolérance aux pannes, spécifiquement pour la détection de serveurs défaillants et la diffusion de cet état de fait.

- Riak

  Riak est une base de données distribuée NoSQL qui offre la tolérance aux pannes. La gestion distribuée des données se fait à l'aide d'un anneau et de «*bucket*». Elle se sert d'un protocole par commérages afin de partager et de communiquer l'état de l'anneau et les propriétés des «*buckets*» sur une grappe.

- Tribler (client BitTorrent)

  Le client Tribler incorpore des capacités de recherche de fichiers «.torrent» grâce à un protocole par commérages ajouté au protocole BitTorrent.

- Dynamo

  Dynamo est un logiciel de collaboration basé sur l'infonuagique. Il emploie un protocole par commérages pour détecter les pannes et mettre à jour la liste des membres. Chaque nœud contacte un nœud pair, choisi au hasard, à toutes les secondes afin que tous deux s'échangent de l'information sur leur liste de membres respective. De cette façon, les modifications à la liste sont rapidement propagées. Le protocole assure une cohérence éventuelle de la liste. Cette implantation est basée sur le protocole de détection de pannes présenté par Gupta et al.[24].

- Consul

  Consul est un système de maillage de services distribués offrant des garanties de disponibilités même en cas de pannes. Le logiciel fournit une interface de contrôle avec des fonctionnalités de découverte, de configuration et de segmentation des services. Consul emploie un protocole par commérages appelée SERF pour la découverte des membres, la détection des pannes et la diffusion de l'information.

### C.3.5 Avantages et inconvénients des protocoles par commérages

Les avantages des protocoles par commérage sont :

- Mise à l'échelle

  Les protocoles par commérages ont la capacité d'évoluer afin de supporter des millions de nœuds. En effet, leur mise à l'échelle est possible car le nombre de cycles nécessaires pour diffuser de l'information est de l'ordre de $O(logN)$ où $N$ est le nombre de nœuds du réseau. De plus, même en présence d'une multitudes de sites, il n'y aura aucune surcharge sur le réseau car chaque nœud communique avec seulement quelques pairs, envoie une quantité fixe de messages (quantité indépendante du nombre de nœuds sur le réseau) et n'attend pas pour un accusé de réception.

- Tolérance aux pannes

  Ce protocole a la capacité de fonctionner sur un réseau irrégulier dont on ne connaît pas la connectivité. Comme un nœud partage la même information avec plusieurs autres nœuds, si l'un d'eux devient inaccessible l'information pourra lui parvenir par un autre chemin (nœud). Dit autrement, il existe plusieurs chemins par lesquels l'information peut être acheminée à destination.

- Robustesse

  Comme aucun nœud ne joue un rôle spécifique, un nœud en panne n'empêchera pas les autres de communiquer. Il est possible pour chaque de se joindre ou de quitter le réseau sans que cela nuise de façon importante au bon fonctionnement du système. Ceci dit, la robustesse n'est pas assurée en toutes circonstances, par exemple, face à des actions malicieuses d'un site.

  Une telle situation s'est produite avec Amazon S3 en juillet 2008. Une panne s'est alors produite car l'information partagée par le protocole de commérages avait subi une corruption sur un seul bit. Le message étant encore intelligible, il a été diffusé. Cependant, due à cette corruption, l'état du système entier est devenu incorrect, nécessitant un arrêt du système pour corriger l'état sur le disque.

- Cohérence éventuelle

  La cohérence des donnes fourni par les protocoles par commérages n'est pas très stricte. Toutefois comme ces derniers propagent l'information efficacement, le système converge très rapidement (éventuellement) vers un état globalement cohérent suite à une mise à jour.

- Décentralisation

  Les systèmes basés sur les protocoles par commérages offrent un niveau de décentralisation élevé, i.e. qu'ils ne nécessitent aucun contrôle central. Ils fournissent une fonctionnalité de découverte d'information extrêmement décentralisée qui n'est toutefois pas parfaite. La latence (la vitesse à laquelle on obtient une nouvelle information) est acceptable dans le cas où l'information n'est pas requise dans un court délai.

# Bibliographie


[1] Nombre pi. http://www.nombrepi.com/, 2020.

[2] A very simple simulation program : Parallel computation of pi. https://www.appentra.com/parallel-computation-pi/, 2020.

[3] Gregory R. Andrews : Paradigms for process interaction in distributed programs. *ACM Comput. Surv.*, 23(1):49–90, 1991.

[4] Gregory R. Andrews et Ronald A. Olsson : *The SR Programming Language : Concurrency in Practice*. Benjamin/Cummings Pub. Co, 1993.

[5] Gregpry R. Andrews : *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley Longman Publishing Co., Inc., USA, 1st édition, 1999.

[6] Blaise Barney et Donald Frederick : Introduction to parallel computing tutorial. https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial##MemoryArch, 2022.

[7] Ken Birman : The promise, and limitations, of gossip protocols. *SIGOPS Oper. Syst. Rev.*, 41(5):8–13, octobre 2007.

[8] Guy E. Blelloch et Bruce M. Maggs : *Parallel Algorithms*, page 25. Chapman & Hall/CRC, 2 édition, 2010.

[9] CAS-Wiki : Heart beat algorithm. http://wiki.cas-group.net/index.php?title=Heart_Beat_Algorithm, 2011.

[10] Geoffrey Challen : What is mapreduce ? https://www.youtube.com/watch?v=43fqzaSH0CQ, 2016.

[11] Devji Chhanga : Map reduce word count with python : Learn data science. https://idevji.com/2017/11/03/map-reduce-word-count-with-python/, 2017.

[12] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart et Doug Terry : Epidemic algorithms for replicated database maintenance. *In Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '87, page 1–12, New York, NY, USA, 1987. Association for Computing Machinery.

[13] Alain Fernandez : Mapreduce, définition. https://www.piloter.org/business-intelligence/map-reduce.htm, 2020.

[14] Ian T. Foster : *Designing and building parallel programs - concepts and tools for parallel software engineering.* Addison-Wesley, 1995.

[15] Martin Fowler : Heartbeat. https://martinfowler.com/articles/patterns-of-distributed-systems/heartbeat.html, 2020.

[16] Martin Fowler : Leader and followers. https://martinfowler.com/articles/patterns-of-distributed-systems/leader-follower.html, 2020.

[17] Martin Fowler : Gossip dissemination. https://martinfowler.com/articles/patterns-of-distributed-systems/gossip-dissemination.html, 2021.

[18] Fayez Gebali : *Algorithms and Parallel Computing.* Wiley, 2011.

[19] Sukumar Ghosh : *Distributed Systems : An Algorithmic Approach, Second Edition.* Chapman & Hall/CRC, 2nd édition, 2014.

[20] Boris Gourévitch : L'univers de $\pi$. http://www.pi314.net/fr/, 2019.

[21] A. Grama, V. Kumar, A. Gupta et G. Karypis : *Introduction to Parallel Computing.* Pearson Education. Addison-Wesley, 2003.

[22] A. Grama, V. Kumar, A. Gupta et G. Karypis : Principles of parallel algorithm design, 2003. [En ligne ; accédé le 21 novembre 2022].

[23] Ananth Grama, George Karypis, Vipin Kumar et Anshul Gupta : *Introduction to Parallel Computing.* Addison-Wesley, second édition, 2003.

[24] Indranil Gupta, Tushar D. Chandra et Germán S. Goldszmidt : On scalable and efficient distributed failure detectors. *In Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing*, PODC '01, page 170–179, New York, NY, USA, 2001. Association for Computing Machinery.

[25] Guru99 : Apache mapreduce. https://www.guru99.com/introduction-to-mapreduce.html, 2020.

[26] Olfa Hamdi-Larbi : Architectures et algorithme parallèle, 2012. [En ligne ; accédé le 19 février 2022].

[27] Céline Hudelot et Régis Behmo : Réalisez des calculs distribués sur des données massives : Divisez (et distribuez) pour régner. https://openclassrooms.com/fr/courses/4297166-realisez-des-calculs-distribues-sur-des-donnees-massives/4308626-divisez-et-distribuez-pour-regner, 2020.

[28] Céline Hudelot et Régis Behmo : Réalisez des calculs distribués sur des données massives : Parcourez les principaux algorithmes mapreduce. https://openclassrooms.com/fr/courses/4297166-realisez-des-calculs-distribues-sur-des-donnees-massives/4308631-parcourez-les-principaux-algorithmes-mapreduce, 2020.

[29] Cécile Hui-Bon-Hoa : Comprendre mapreduce. https://blog.soat.fr/2015/05/comprendre-mapreduce/, 2015.

[30] IBM : Apache mapreduce. https://www.ibm.com/analytics/hadoop/mapreduce, 2020.

[31] Anne-Marie Kermarrec et Maarten van Steen : Gossiping in distributed systems. *SIGOPS Oper. Syst. Rev.*, 41(5):2–7, octobre 2007.

[32] Amir Keshavarz : Introduction to gossip/epidemic protocol and memberlist. https://medium.com/@satrobit/introduction-to-gossip-epidemic-protocol-and-memberlist-5424352cdce0, 2020.

[33] Ravi Kiran : Mapreduce tutorial – fundamentals of mapreduce with mapreduce example. https://www.edureka.co/blog/mapreduce-tutorial/, 2020.

[34] Bastien L. : Mapreduce : tout savoir sur le framework hadoop de traitement big data. https://www.lebigdata.fr/mapreduce-tout-savoir, 2020.

[35] Linus : Computing pi as a riemann sum. https://dotink.co/posts/pi-by-riemann-sum/, 2020.

[36] Félix Lopez : Just my thoughts : Introduction to gossip. https://managementfromscratch.wordpress.com/2016/04/01/introduction-to-gossip/, 2016.

[37] Félix Lopez Luis : Understanding gossip protocol. https://codesync.global/media/understanding-gossip-protocols-felix-lopez/, 2018.

[38] MAPR : Mapreduce. https://mapr.com/products/product-overview/mapreduce/, 2020.

[39] Jeff Matocha et Tracy Camp : A taxonomy of distributed termination detection algorithms. *Journal of Systems and Software*, 43(3):207–221, 1998.

[40] Robert McCurley et Fred B. Schneider : Derivation of a distributed algorithm for finding paths in directed networks. *Sci. Comput. Program.*, 6(1):1–9, 1986.

[41] Michael G. Noll : Writing an hadoop mapreduce program in python. https://www.michael-noll.com/tutorials/writing-an-hadoop-mapreduce-program-in-python/, 2010.

[42] Peter Pacheco : *An Introduction to Parallel Programming*. Morgan Kaufmann, 2011.

[43] Shana Pearlman : Mapreduce – présentation générale. https://fr.talend.com/resources/what-is-mapreduce/, 2019.

[44] Margaret Rouse : Mapreduce. https://www.lemagit.fr/definition/MapReduce, 2020.

[45] High Scalability : Using gossip protocols for failure detection, monitoring, messaging and other good things. http://highscalability.com/blog/2011/11/14/using-gossip-protocols-for-failure-detection-monitoring-mess.html, 2011.

[46] Dhruv Sharma : Gossiping algorithms : Even they gossip, that too in an amazing and fascinating way. https://dhruvsharma-50981.medium.com/gossiping-algorithms-even-they-gossip-that-too-in-an-amazing-and-fascinating-way-1bce3bed6dbf, 2019.

[47] Swarnim SINGHAL : Implementing cassandra's gossip protocol : Part 1. https://medium.com/@swarnimsinghal/implementing-cassandras-gossip-protocol-part-1-b9fd161e5f49, 2019.

[48] STACKEXCHANGE : Calculate $\pi$ precisely using integrals ? https://math.stackexchange.com/questions/22777/calculate-pi-precisely-using-integrals, 2019.

[49] STRINGFIXER : Interconnexion de tore, 2012. [En ligne ; accédé le 19 février 2022].

[50] Rajagopal SUBRAMANIYAN, Pirabhu RAMAN, Alan D. GEORGE et Matthew RADLINSKI : Gems : Gossip-enabled monitoring service for scalable heterogeneous distributed systems. *Cluster Comput*, page 2006.

[51] Gerard TEL : *Introduction to Distributed Algorithms*. Cambridge University Press, 2 édition, 2000.

[52] Dominique THIEBAUT : Hadoop tutorial 2 – running wordcount in python. http://www.science.smith.edu/dftwiki/index.php/Hadoop_Tutorial_2_--_Running_WordCount_in_Python, 2010.

[53] Lawrence TURNER : Arctan formulae for computing $\pi$. http://turner.faculty.swau.edu/mathematics/materialslibrary/pi/piforms.html, 2019.

[54] TUTORIALSPOINT : Hadoop - mapreduce. https://www.tutorialspoint.com/hadoop/hadoop_mapreduce.htm, 2020.

[55] TUTORIALSPOINT : Parallel algorithm tutorial. https://www.tutorialspoint.com/parallel_algorithm/design_techniques.htm, 2021.

[56] Alvaro VIDELA : Gossip protocols, where to start. https://alvaro-videla.com/2015/12/gossip-protocols.html, 2015.

[57] WIKIPEDIA : Formule de machin. https://fr.wikipedia.org/wiki/Formule_de_Machin, 2019.

[58] WIKIPEDIA : Map (higher-order function)). https://en.wikipedia.org/wiki/Map_(higher-order_function), 2020.

[59] WIKIPEDIA : Map (parallel pattern). https://en.wikipedia.org/wiki/Map_(parallel_pattern), 2020.

[60] WIKIPEDIA : Reduce (parallel pattern). https://en.wikipedia.org/wiki/Reduce_(parallel_pattern), 2020.

[61] WIKIPEDIA : Gossip protocol. https://en.wikipedia.org/wiki/Gossip_protocol, 2021.

[62] WIKIPEDIA : Heartbeat (computing). https://en.wikipedia.org/wiki/Heartbeat_(computing), 2021.

[63] WIKIPEDIA : Hypercube, 2021. [En ligne ; accédé le 19 février 2022].

[64] WIKIPEDIA : Protocole de bavardage. https://fr.wikipedia.org/wiki/Protocole_de_bavardage, 2021.

[65] Wikipedia : Tile processor, 2021. [En ligne ; accédé le 19 février 2022].

[66] Wikipedia : Topologie mesh, 2021. [En ligne ; accédé le 19 février 2022].

[67] Wikipedia : Torus interconnect, 2021. [En ligne ; accédé le 19 février 2022].

[68] Wikipedia : Mesh networking, 2022. [En ligne ; accédé le 19 février 2022].

[69] wikipedia : Méthode de monte-carlo. `https://fr.wikipedia.org/wiki/M%C3%A9thode_de_Monte-Carlo`, 2022.