**Cours 1**

Le *benchmark* (une norme ou un point de référence) de performance en calcul scientifique est donné par le nombre d'opérations en virgule flottante par seconde (***FL****oating* ***O****perations* ***P****er* ***S****econd*). Les FLOPS mesurent la capacité d'un système informatique à effectuer des calculs mathématiques complexes. Plus le nombre de FLOPS est élevé, plus le système est capable d'exécuter des calculs rapidement et efficacement.

La "loi" de Moore (le nombre de transistors pour chaque génération de processeurs doublait tous les dix-huit mois, ce qui doublait la puissance de calcul) a comme limitations la miniaturisation des transistors et l'augmentation de leurs fréquences entraînent une production de chaleur plus élevée à l'intérieur du processeur.

De nos jours, la "loi” de Moore est toujours vérifiée, non pas en doublant le nombre de transistors à l'intérieur d'un cœur de calcul, mais plutôt **en augmentant le nombre de cœurs de calcul à l'intérieur d'un processeur ou d'un ordinateur**.

* Solution : Exécution concurrente de tâches interdépendantes sur plusieurs cœurs de calcul

**Remarque :** De nos jours, nous recherchons les **performances en FLOPS/Watt**

* Architecture à mémoire partagée

De nombreux cœurs de calcul partagent la même mémoire principale.

A diagram of a diagram

Description automatically generated

Problème d'accès à la mémoire :

* Optimisation de l'accès à la mémoire
* Accès en lecture/écriture simultanés à la même adresse mémoire
* Exemple de latence mémoire (architecture Haswell)

A table with numbers and text

Description automatically generated

* La mémoire devient plus lente en comparaison de l'exécution des instructions par le processeur
* Les niveaux de cache plus proches du processeur (comme le Cache L1 et le Cache L2) ont des latences plus faibles, ce qui signifie qu'ils peuvent fournir des données plus rapidement au processeur. Cependant, à mesure que l'on s'éloigne du processeur et que l'on atteint des niveaux de cache plus élevés (comme le Cache L3), ainsi que la RAM et le Swap, les latences augmentent. Par conséquent, **le processeur doit attendre plus longtemps pour accéder aux données dans ces niveaux de mémoire plus éloignés**.
* Cette disparité est exacerbée dans les architectures multicœurs
* Plusieurs cœurs de processeur fonctionnent simultanément et partagent souvent les mêmes ressources de mémoire, comme le Cache L3. **Un accès fréquent à cette mémoire peut entraîner des conflits d'accès et une augmentation de la latence, ce qui peut ralentir l'ensemble du système**.
* RAM

***R****andom* ***A****ccess* ***M****emory* (mémoire vive), est une forme de mémoire informatique volatile utilisée par les ordinateurs pour stocker des données et des instructions temporairement pendant qu'elles sont traitées par le processeur.

* Lorsque vous exécutez un programme ou effectuez une tâche sur votre ordinateur, les données nécessaires à cette tâche sont chargées depuis le disque dur ou un autre support de stockage dans la RAM. Cela inclut les programmes en cours d'exécution, les fichiers ouverts, les processus système et les données d'application.
* La RAM est conçue pour offrir un accès rapide aux données. Contrairement au disque dur ou au SSD, qui ont des temps d'accès plus lents, la RAM peut être lue et écrite très rapidement par le processeur. Cela permet au processeur d'accéder aux données nécessaires pour exécuter des programmes et effectuer des opérations de manière efficace.
* La RAM est organisée en une série de cellules, chacune capable de stocker un petit morceau de données. Chaque cellule a une adresse unique qui permet au processeur d'y accéder directement. C'est pourquoi on l'appelle "mémoire à accès aléatoire", car le processeur peut accéder à n'importe quelle cellule de manière aléatoire, sans devoir parcourir séquentiellement les données.
* La RAM est volatile, ce qui signifie que les données qu'elle contient sont effacées lorsque l'alimentation est coupée. Cela contraste avec le stockage non volatile comme les disques durs ou les SSD, qui conservent les données même lorsque l'alimentation est éteinte. C'est pourquoi les programmes et les données doivent être enregistrés sur un support de stockage permanent si vous souhaitez les conserver après avoir éteint votre ordinateur.

**Conséquences de l'architecture en grille pour les RAMS :** Plus la mémoire est grande, plus sa grille est grande, ce qui entraîne des accès en lecture et en écriture plus lents.

* RAMs intercalées

A diagram of a process

Description automatically generated

Mémoire intercalée :

* Plusieurs unités de mémoire physique intercalées par le bus mémoire
* Le nombre d'unités de mémoire physique ≡ **nombre de voies** (*canales*)
* Le nombre d'octets contigus dans une seule unité de mémoire physique ≡ **largeur de la voie** (*canales*)

Dans **l'accès mémoire classique**, les données sont récupérées depuis la mémoire de manière directe, linéaire et séquentielle. Le processeur émet une demande de mémoire, et la mémoire répond en fournissant les données demandées. Il existe une relation directe et un-à-un entre les adresses mémoire et les emplacements physiques de la mémoire.

En revanche, **l'accès mémoire intercalé sur quatre voies** divise la mémoire en quatre segments ou banques. Lorsque le processeur accède à la mémoire, chaque accès mémoire consécutif est réparti entre ces quatre banques de manière tournante, ce qui peut améliorer les performances d'accès à la mémoire dans certains scénarios.

L'avantage de l'accès mémoire intercalé est qu'il peut **réduire les conflits d'accès mémoire et améliorer la bande passante mémoire globale**, notamment dans les systèmes avec une demande mémoire élevée ou plusieurs demandes d'accès mémoire se produisant simultanément. Cependant, la mise en œuvre de l'accès mémoire intercalé peut également **augmenter la complexité et le coût du système**, car elle nécessite du matériel supplémentaire pour gérer le processus d'intercalation et assurer la cohérence des données entre les banques de mémoire.

* Mémoire Cache
* Petite unité de mémoire rapide utilisée pour stocker des données temporaires
* Lorsqu'il y a plusieurs accès à la même variable en peu de temps, cela accélère l'accès en lecture ou en écriture
* La mémoire cache est gérée par le CPU (bien que la mémoire cache des GPU puisse être gérée par le programmeur)

**Conséquence :** pour optimiser leur programme, le programmeur doit être conscient des stratégies utilisées par le CPU

La **RAM** est une mémoire principale utilisée pour **stocker des données et des instructions en cours d'utilisation par le processeur**. **Elle** **est plus grande mais plus lente que la mémoire cache**. La **mémoire cache**, en revanche, est une mémoire plus petite mais plus rapide située entre le processeur et la RAM. **Elle stocke des données fréquemment utilisées pour accélérer l'accès du processeur à ces données**.

Le **CPU** (**unité centrale de traitement**) est le **cerveau de l'ordinateur**, responsable de l'exécution des instructions et du contrôle des opérations de base. Il est conçu pour effectuer un petit nombre de tâches complexes très rapidement. Le **GPU** (**unité de traitement graphique**), quant à lui, est **spécialisé dans le traitement graphique** et est conçu pour effectuer de nombreuses tâches simples simultanément. Il est efficace pour les opérations parallèles et est souvent utilisé dans les applications de rendu graphique, de calcul scientifique et de l'intelligence artificielle.

Stratégie du CPU

* **Ligne de cache** : Se réfère à la façon dont les données sont stockées dans le cache du processeur. Les données contiguës en mémoire sont regroupées et stockées ensemble dans le cache.
* **Cache mémoire associatif** : Implique que chaque adresse de mémoire cache est associée à une adresse fixe dans la RAM, souvent **en utilisant l'opération de modulo pour déterminer l'emplacement dans la RAM**.

**Conséquences :**

* Avantageux d'avoir un accès contigu à la mémoire
* Préférable d'utiliser les données stockées dans le cache dès que possible
* **Localisation spatiale et temporelle des données**

A diagram of data processing

Description automatically generated

Dans un système avec une seule mémoire cache, les données sont toujours à jour.

Lorsque les données sont partagées entre plusieurs mémoires caches (comme dans les **processeurs multi-cœurs**), chaque cache vérifie si les données ont été modifiées par d'autres cœurs avant de les utiliser. Lorsqu'une mémoire cache modifie les données partagées, elle marque sa propre copie comme invalide. Cela est nécessaire car les autres caches pourraient avoir des versions plus anciennes des données et doivent être mises à jour avec les derniers changements.

Une fois que la mémoire cache marque sa copie comme invalide, elle s'assure que la mémoire principale (RAM) est mise à jour avec la dernière version des données. Cela garantit que toutes les mémoires caches ont accès aux informations les plus récentes.

Lorsqu'une mémoire cache détecte que ses données sont invalides (parce qu'elles ont été marquées comme telles ou qu'elle a besoin de la dernière version), elle récupère les données mises à jour depuis la RAM avant de les réutiliser. Cela garantit que toutes les mémoires caches disposent d'informations cohérentes et à jour, ce qui prévient les incohérences ou les erreurs de données dans les systèmes multi-cœurs.

A diagram of a computer network

Description automatically generated

Le même problème se pose avec la cohérence du cache, mais il existe un besoin de cohérence des données entre les niveaux de cache. La complexité augmente avec le nombre de niveaux de cache.

Un *thread* est une **séquence d'instructions exécutée par un processeur**. Il s'agit d'une **unité de base d'exécution d'un programme**. Plusieurs *threads* peuvent s'exécuter simultanément sur un processeur multi-cœurs ou un système multi-thread. Chaque *thread* peut exécuter des **tâches indépendantes**, ce qui permet un **traitement parallèle** et une **utilisation efficace** des **ressources du système**.

* Mémoire distribuée

A diagram of a network

Description automatically generated

* Chaque unité de calcul peut lire/écrire sur la RAM locale. **L'ensemble contenant l'unité de calcul et la RAM est appelé le** **nœud de calcul**.
* Les **données** sont **échangées** entre les nœuds de calcul **via un bus spécialisé** ou un lien Ethernet spécifique (il incombe au programmeur d'échanger explicitement des donnée).
* La limitation est seulement imposée par la **consommation d'électricité** (avec un coût linéaire).
* Gestion du contexte dans votre programme
* Appeler l'initialisation du contexte parallèle avant d'utiliser d'autres fonctions de la bibliothèque (MPI\_Init)
* Obtenir le nombre de processus contenus dans le communicateur (MPI\_Comm\_size)
* Récupérer le rang du processus dans le communicateur (MPI\_Comm\_rank)
* Après avoir appelé la dernière fonction de la bibliothèque, invoquer la terminaison du contexte parallèle pour synchroniser les processus (MPI\_Finalize)

A screenshot of a computer code

Description automatically generated

* Interblocage

L'interblocage est une situation où de nombreux processus attendent indéfiniment les uns les autres pour terminer leurs messages. Par exemple, le processus 1 attend de recevoir un message du processus 0, et le processus 0 attend de recevoir un message du processus 1.

**Règle générale :** Faites attention à vous assurer que **chaque envoi a une réception correspondante** avec le tag et l'expéditeur corrects.

A black text on a white background

Description automatically generated

**Message bloquant :** Attend la réception complète du message avant de retourner de la fonction.

**Message non bloquant :** Poste le message à envoyer ou à recevoir et retourne immédiatement de la fonction.

* Le statut d'un message non bloquant est **mis à jour dans une structure de requête** (pas encore envoyé/reçu, en cours d'envoi/réception, ou envoyé/reçu).
* Un schéma pour éviter les situations d'interblocage
* Effectuer les **réceptions** en mode **non bloquant**
* Effectuer les **envois** en mode **bloquant**
* **Synchroniser vos réceptions** (en attendant l'achèvement (*finalización*) ou en testant pour chevaucher (*superponer*) le coût des messages avec le traitement)

A screenshot of a computer code

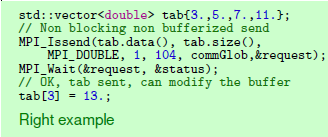
Description automatically generated

* Messages tamponnés (*buffered messages*)

Un **envoi non bloquant** est copié dans un tampon **avant d'être transmises au destinataire**. Cette copie dans le tampon permet à l'émetteur de continuer à travailler sur les données après l'envoi du message, sans attendre la fin de la transmission. Cependant, copier les données dans un tampon entraîne des coûts en termes de mémoire et de CPU. Les utilisateurs peuvent appeler des fonctions d'envoi qui ne copient pas les données dans un tampon.

A computer code with text

Description automatically generated



A screen shot of a computer code

Description automatically generated

La fonction MPI\_Wait est ensuite utilisée pour garantir que l'opération d'envoi initiée par MPI\_Issend est terminée avant de continuer. Cela garantit que les données ont été envoyées en toute sécurité à leur destination.

* Communication collective

Qu'est-ce que la communication collective :

* Diffusion des données d'un processus à tous les processus.
* Dispersion des données d'un processus à tous les processus.
* Rassemblement des données de tous les processus vers un processus.
* Réduction des données (avec une opération arithmétique) de tous les processus vers un processus ou tous les processus.
* Analyse des données (avec une opération arithmétique) de tous les processus vers tous les processus.
* Diffusion/dispersion de données de tous vers tous.

Pourquoi la communication collective :

* La communication point à point est suffisante pour de nombreux algorithmes.
* Cependant, pour certaines opérations parallèles (diffusion, réduction, dispersion), l'algorithme optimal dépend de la topologie du réseau.
* Les bibliothèques parallèles distribuées fournissent une communication collective optimisée pour différentes topologies de réseau.
* L'algorithme résultant est plus clair.
* Règles de parallélisme distribué
* Les échanges de données Ethernet sont très lents par rapport à l'accès à la mémoire : **minimiser autant que possible les échanges de données**.
* Pour masquer les coûts des échanges de données, il est préférable de calculer certaines valeurs pendant l'échange de données : **privilégier l'utilisation d'échanges de messages non bloquants**.
* Chaque message a un coût initial : **regrouper les données dans un tampon temporaire si nécessaire**.
* Assurer que tous les processus quittent le programme en même temps : **essayer de répartir la charge de calcul entre les nœuds de calcul**.

**Cours 2**

* A mathematical equation with numbers and symbols

  Description automatically generatedSpeed-up

*ts* : Temps d'exécution séquentielle

*tp(n)* : Temps d'exécution sur *n* unités de calcul

Remarque :

L'algorithme séquentiel est souvent différent de l'algorithme parallèle. Dans ce cas, la mesure du *speed-up* n'est pas évidente. En particulier, les questions suivantes doivent être posées, entre autres :

* L'algorithme séquentiel est-il optimal en termes de complexité ?
* L'algorithme séquentiel est-il bien optimisé ?
* L'algorithme séquentiel exploite-t-il au mieux la mémoire cache ?
* Loi d'Amdahl (elle fournit une limite pour le *speed-up*)

Soit *ts* le temps nécessaire pour exécuter le code séquentiellement.

Soit *f* la fraction du code qui peut être parallélisée.

Soit *S(n)* l'accélération obtenue en utilisant *n* unités de calcul.

* Cette loi est utile pour trouver un **nombre raisonnable de cœurs de calcul à utiliser** pour une application.

Limitation de la loi : La valeur de *f* peut changer avec le volume des données en entrée, et des données d'entrée plus importantes peuvent améliorer l'accélération.

* Loi de Gustafson (le comportement de le *speed-up* reste constant avec un volume fixe de données en entrée par processus)

Soit *ts* le temps pris par l'exécution de la partie séquentielle du code.

Soit *tp* le temps pris par l'exécution de la partie parallèle du code pour une quantité fixe de données.

Hypothèses :

* *ts* ≥ 0 : Le temps pour exécuter la partie séquentielle du code est **indépendant du volume de données en entrée**
* *tp* > 0 : Le temps pour exécuter la partie parallèle du code est **linéairement dépendant du volume de données en entrée**

Supposons que *ts* + *tp* = 1 (une unité de temps).

A math equation with numbers and symbols

Description automatically generated

* La **loi d'Amdahl** se concentre sur les **limitations de l'accélération** due à la **fraction séquentielle** d'un programme, tandis que la **loi de Gustafson** met en avant **l'amélioration de l'accélération** avec la **taille des données**. La première est axée sur les **limites de la parallélisation**, tandis que la seconde explore la **scalabilité des programmes parallèles**.
* Scalabilité

Pour un programme parallèle, la scalabilité représente le **comportement de l'accélération** lorsque nous augmentons le **nombre de processus** et/ou la **quantité de données en entrée**.

Comment évaluer la scalabilité ?

* Évaluer le pire *speed-up* : **Pour une quantité de données globalement fixe**, tracer la courbe de le *speed-up* en fonction du nombre de processus.
* Évaluer le meilleur *speed-up* : **Pour une quantité de données fixe par processus**, tracer la courbe de le *speed-up* en fonction du nombre de processus.
* Dans l'utilisation pratique du programme, le *speed-up* peut se situer entre les scénarios du pire et du meilleur.
* La **scalabilité** d'un programme parallèle se réfère à sa **capacité à maintenir une amélioration de la performance** lorsque le **nombre de processus** ou la **taille des données en entrée** est **augmenté**.
* Granularité

Le rapport entre **l'intensité de calcul** et **la quantité de données échangées** entre les processus :

* L'envoi et la réception de données peuvent être prohibitifs :
* **Coût initial d'un message :** Chaque message entraîne un coût initial, comprenant l'établissement de la connexion, l'établissement du même protocole, etc. **Ce coût est constant**.
  + **Coût du transfert de données :** Enfin, le coût du flux de données est linéaire avec le nombre de données à échanger.
* Ces coûts **dépassent le coût des opérations** **de mémoire** dans la RAM.
* Il vaut mieux **copier certaines données éparses dans un tampon et envoyer le tampon**, plutôt que d'envoyer des données dispersées avec de multiples envois et réceptions.
* Essayez de minimiser le nombre d'échanges de données entre les processus.
* Plus le ratio entre le nombre d'instructions de calcul et les messages à échanger est élevé, meilleur sera le *speed-up.*
* Un faible *speed-up* peut être amélioré avec des **échanges de données non bloquants**.
* La **granularité** se rapporte à **la taille des tâches de traitement par rapport à la quantité de données échangées entre les processus**. Une granularité **fine** signifie que les tâches sont petites et que de nombreuses données sont échangées entre les processus, ce qui peut **entraîner des** **frais de communication élevés**. Une granularité **grossière** signifie que les tâches sont grandes et que peu de données sont échangées, ce qui peut **limiter l'efficacité de la parallélisation**.
* Équilibrage de charge

Tous les processus exécutent une section de calcul du code avec la même durée. Le *speed-up* est fortement impactée si certaines parties du code sont mal équilibrées.

* Exemple 1 : Une fonction prend *t* secondes pour la moitié des processus et *t/2* pour les autres processus. Le *speed-up* maximale pour cette fonction sera :



* Exemple 2 : Une fonction prend *t/2* secondes pour *n-1* processus et *t* pour un processus. Le *speed-up* maximale pour cette fonction sera :



**Remarque :** Plus il faut de temps pour exécuter une fonction avec un mauvais équilibrage de charge, plus la pénalité est importante. **Ne vous inquiétez pas de l'équilibrage de charge pour les fonctions qui prennent très peu de temps à s'exécuter**.

* Assurer que les **charges de travail sont réparties de manière équitable** entre les différents processeurs ou *threads* dans un système parallèle. Cela garantit que **chaque unité de traitement est** **utilisée efficacement**, évitant ainsi les goulets d'étranglement (*cuellos de botella*) et **maximisant les performances globales** du système.
* Algorithme parallèle embarassant
* Chaque donnée utilisée et calculée est indépendante.
* Aucune course de données dans un contexte multithread.
* Aucune communication entre les processus dans un environnement distribué.

Propriétés :

* Dans un contexte parallèle distribué, aucune donnée ne doit être échangée entre les processus pour calculer les résultats.
* Dans un environnement parallèle partagé, la parallélisation est simple, mais attention aux calculs limités par la mémoire.
* Dans un environnement distribué, la limitation liée à la mémoire n'est pas un problème.
* Si les données sont contiguës et que l'algorithme est vectorisable, il peut être idéal sur les GPGPU pour les performances.
* Dans un algorithme parallèle embarassant, **chaque tâche peut être exécutée indépendamment des autres**. Il n'y a pas de conflits de données entre les *threads*, ni de besoin de communication entre les processus dans un environnement distribué. Ces algorithmes sont efficaces car **ils évitent les coûts liés à la synchronisation et à la communication**, permettant ainsi une **exécution rapide et autonome des tâches parallèles**.
* **Environnement distribué :** Les ressources sont réparties sur plusieurs nœuds connectés via un réseau. Chaque nœud fonctionne de manière autonome et les processus peuvent communiquer entre eux via le réseau.
* **Environnement partagé :** Tous les processus ont accès aux mêmes ressources sur une seule machine. Ils peuvent partager des données et communiquer plus facilement.
* Premier exemple : Addition de vecteurs

Ajoutez deux vecteurs réels de dimensions *N*



* Pour l'équilibrage de charge, répartissez les vecteurs en parties égales entre les *threads* ou les processus.
* Chaque processus/*thread* calcule une partie de l'addition des vecteurs.

Quelques propriétés :

* L'accès à la mémoire et les opérations de calcul ont la même complexité : **Sur une mémoire partagée, la limitation liée à la mémoire affecte les performances**.
* En **mémoire distribuée**, chaque processus utilise sa propre mémoire physique et **aucune donnée n'est échangée** : Lé *speed-up* peut être linéaire par rapport au nombre de processus (si l'intensité des données est suffisante).
* Deuxième exemple : Multiplication de matrices diagonales par blocs

Multiplication de matrices C = A.BB

A number of lines and numbers

Description automatically generated with medium confidence

où *di*= dim(*Aii*) = dim(*Bii*) (*n* multiplications de matrices indépendantes)

Problématique : Proche de la multiplication de vecteurs, mais :

* Les dimensions *di* des blocs diagonaux sont hétérogènes, et pour chaque bloc diagonal, la complexité de calcul est *di3*​.
* Comment répartir les blocs diagonaux entre les processus pour obtenir un équilibrage de charge presque optimal ?

Un algorithme pour distribuer les blocs diagonaux :

1. Trier les blocs diagonaux par ordre décroissant de dimensions.
2. Initialiser le "poids" à zéro pour chaque processus.
3. Distribuer les plus grands triplets de blocs *Aii* , *Bii* , *Cii* parmi les processus et ajouter chaque *di*au poids de chaque processus.
4. Tant que certains blocs diagonaux ne sont pas distribués :
   1. Ajouter le plus grand bloc qui n'est pas distribué au processus ayant le poids le plus faible.
   2. Ajouter la dimension *di*  relative au poids du processus.

**Remarque :** Tous les processus calculent la distribution des blocs diagonaux. Il est préférable de réaliser le même calcul pour tous les processus plutôt que d'avoir le processus 0 calculer la distribution et l'envoyer aux autres processus.

* Troisième exemple : Série de Syracuse

A black and white math equations

Description automatically generated with medium confidence

* Une séquence cyclique existe : 1 → 4 → 2 → 1 → · · ·
* Une conjecture **: Pour tout *u0*** ∈ ***N*, la série atteint le cycle ci-dessus en un nombre fini d'itérations**

**Longueur du vol :** nombre d'itérations nécessaires pour qu'une série atteigne la valeur 1

**Hauteur du vol :** la valeur maximale atteinte par une série

* L'objectif du programme : calculer la longueur et la hauteur du vol pour beaucoup de valeurs (impaires) de ***u0***

Problématique :

* Chaque processus calcule la longueur et la hauteur pour un sous-ensemble de valeurs initiales ***u0***
* L'intensité de calcul dépend de la longueur de chaque série de Syracuse
* Il est impossible de connaître la complexité de calcul d'une série avant de l'avoir calculée
* Le problème n'est pas naturellement bien équilibré
* Utiliser un algorithme dynamique sur le processus "racine" (le processus "maître") pour distribuer les séries parmi les autres processus ("esclaves")

A white background with black text and blue dots

Description automatically generated

* Algorithme presque parallèle embarassant

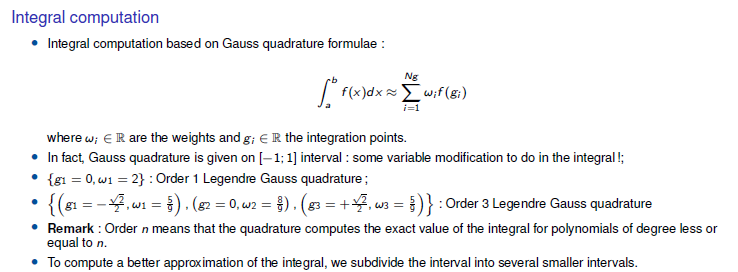
Calcul indépendant pour chaque processus avec une communication finale pour finaliser le calcul.

Exemples d'algorithmes presque parallèle embarassant :

* Produit scalaire de deux vecteurs dans *Rn*
* Calcul d'une intégrale
* Produit matrice-vecteur

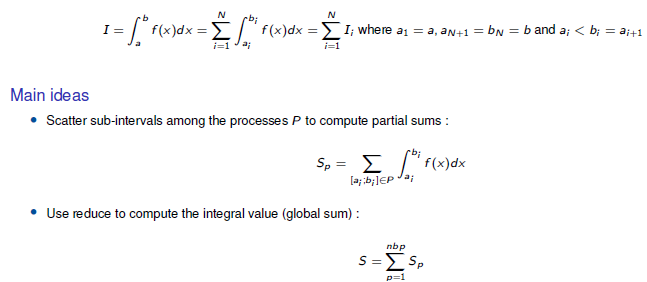
Exemples d'algorithmes NON presque parallèles embarassant

* Algorithmes de tri parallèle
* Produit matriciel
* Algorithmes basés sur des méthodes de décomposition de domaine



* Calcul intégral parallèle

Le calcul intégral parallèle implique la division d'une intégrale en plusieurs sous-intervalles, chacun étant calculé de manière indépendante par différents processus. Une fois les calculs effectués, les résultats sont généralement combinés pour obtenir la valeur finale de l'intégrale.



* Produit matrice-vecteur

Soit *A* ∈ *Rnxm* une matrice et *u* ∈ *Rm* un vecteur. L'objectif de cet algorithme est de calculer le produit matrice-vecteur :



Deux possibilités pour paralléliser cet algorithme :

* Partitionner la matrice par blocs de lignes
* Partitionner la matrice par blocs de colonnes et le vecteur *u* par blocs de même taille
* L'objectif est de répartir le calcul entre les processus et d'utiliser une opération de communication globale pour obtenir le résultat final
* Produit matrice-vecteur par division des lignes

A black text with black text

Description automatically generated with medium confidence

* Chaque processus possède certaines lignes de *A* et l'ensemble de *u*.
* Chaque processus calcule une partie de *v* : le processus *I* calcule *vI = AI ⋅ u ∈ Rn/N*
* Pour calculer un autre produit matrice-vecteur avec le nouveau vecteur, nous devons rassembler le vecteur dans tous les processus (**uniquement nécessaire pour l'algorithme parallèle distribué**)
* A rectangular object with text

  Description automatically generatedProduit matrice-vecteur par division des colonnes
* Chaque processus possède certaines colonnes de *A* et certaines lignes de *u*.
* Chaque processus calcule une somme partielle pour *v* : le processus *I* calcule *VI = AI ⋅ UI ∈ Rn*
* Finalement, une réduction de somme est effectuée pour obtenir le résultat final : A screenshot of a computer

  Description automatically generated

Le "Mandelbrot set" et le "Buddhabrot" (ou "Buddha set") sont deux ensembles fractals liés mais distincts.

* Ensemble de Mandelbrot

Le Mandelbrot set est un ensemble de points dans le plan complexe, déterminé par une règle mathématique simple mais puissante. Chaque point c du plan complexe est soumis à une itération récursive basée sur l'équation zn+1 = zn2 + c, où z0 = 0. Si cette suite itérative reste bornée (c'est-à-dire qu'elle ne tend pas vers l'infini), le point c appartient à l'ensemble de Mandelbrot. Les points qui n'appartiennent pas à l'ensemble sont ceux pour lesquels la suite diverge vers l'infini.

* L'ensemble de Mandelbrot est connu pour ses structures fractales complexes et auto-similaires, et il est souvent représenté graphiquement avec des couleurs pour mettre en évidence les différentes régions et les détails fins.
* Buddhabrot (Buddha set)

Le Buddhabrot est une variation du Mandelbrot set, qui se concentre sur la visualisation des chemins de fuite (fuga) des points qui ne sont pas dans l'ensemble de Mandelbrot, plutôt que de simplement marquer les points appartenant à l'ensemble. Il utilise des algorithmes de suivi de trajectoire pour déterminer quels chemins sont fréquemment empruntés par les points qui divergent.

* Cette variation permet d'explorer les structures et les motifs présents dans les itérations de la suite de Mandelbrot de manière plus détaillée et dynamique.

**Cours 3**

* Complexité des algorithmes de tri

Opérations de base :

* La complexité de **l'algorithme de comparaison** est supposée être en *O(1)*. Cependant, dans un contexte parallèle distribué, il faut tenir compte de la **distribution des données initiales** pour prendre en compte le **coût de l'échange de données entre les processus**.
* La complexité de **l'algorithme d'échange** est supposée être en *O(1)*. Cependant, **la même considération s'applique que pour l'algorithme de comparaison**.
* Algorithme séquentiel "compare-and-exchange" :

A computer code with text

Description automatically generated with medium confidence

* Potentiel de *speed-up*
* Les meilleurs algorithmes de tri séquentiels pour des séquences arbitraires de nombres ont une complexité temporelle moyenne de :
* Par conséquent, le meilleur gain de vitesse que l'on peut attendre de l'utilisation de *n* processeurs est :
* Il existe de tels algorithmes parallèles, mais la constante cachée est très grande.
* En général, trouver un algorithme pratique et utile peut être difficile.

A diagram of a data flow

Description automatically generated**Attention :** il peut être une mauvaise idée d'utiliser *n* processus pour trier *n* données (granularité).

* Parallélisation d'un algorithme naïf
* Compter le nombre d'éléments plus petits qu'un nombre donné *a* dans la liste.
* Cela détermine la position de *a* dans la liste triée.
* Cette procédure doit être répétée pour tous les éléments de la liste ; par conséquent, la complexité temporelle est :

(pas très efficace pour les algorithmes séquentiels)

Pour paralléliser cet algorithme, on peut répartir la liste entre plusieurs processus ou *threads*, chaque processus étant chargé de compter les éléments plus petits que *a* dans sa propre partie de la liste. Ensuite, les résultats partiels peuvent être combinés pour obtenir le résultat final. Cela peut réduire le temps d'exécution total de l'algorithme, en particulier pour de grandes listes.

A white background with text and symbols

Description automatically generated

**Attention :** L'implémentation fonctionne bien si la liste ne contient pas de répétitions de nombres. Cela est dû au fait que chaque élément est traité de manière unique dans la liste, et il n'y a pas de confusion sur la position de chaque élément par rapport à la valeur donnée *a*.

* Rank sort: Code parallèle

Algorithme "idéal" embarassant : Code parallèle, utilisant *n* processus (pour *n* valeurs à trier)



Complexité

* *n* processeurs travaillent en parallèle pour déterminer les rangs de tous les nombres dans la liste.
* La complexité temporelle parallèle est en *O(n)*, supérieure à tout algorithme de tri séquentiel.
* Convient aux unités GPGPU (General-Purpose Graphics Processing Unit).

**Attention :** is not efficient to use *n* or *n2* processors to sort *n* numbers => processor efficiency is very low

* Partitionnement des données

Généralement, le nombre *n* de valeurs est beaucoup plus grand que le nombre *p* de processus. Dans de tels cas, chaque processus gérera une partie des données (une sous-liste des données).

Conteneur trié distribué :

Le conteneur local est trié



Schéma global de l'algorithme de tri parallèle (pour chaque processus) :

* Trier ses données locales
* Exécuter un algorithme de tri fusion pour concaténer sa liste avec celle reçue d'un autre processus
* Conserver la moitié inférieure (ou la moitié supérieure) de la liste triée
* Opérations d'échange et de comparaison parallèles
* Algorithme Asymétrique
* Le processus *pi* envoie sa valeur locale *A* au processus *pj*.
* Le processus *pj* compare la valeur *A* avec ses valeurs locales *Bj*.
* Envoyer les valeurs *Bj* qui sont plus grandes (ou plus petites) que *A*. Si aucun *Bj* n'est plus grand (ou plus petit) que *A*, renvoyer *A*.
* Algorithme Symétrique
* Les processus *pi* et *pj* échangent des valeurs entre eux.
* Chaque processus compare sa propre valeur avec la valeur reçue.
* Chaque processus conserve sa valeur d'origine ou la valeur reçue en fonction du résultat de la comparaison.

**Remarques :**

* Les échanges de données entre les processus sont très coûteux, donc les algorithmes doivent minimiser les échanges de données.
* En général, l'opération de réception ne connaît pas le nombre de valeurs à recevoir. Par conséquent, il est nécessaire de sonder le message reçu pour déterminer le nombre de données à recevoir, allouer le tampon relatif, puis recevoir les données.

A screenshot of a computer

Description automatically generated

* Algorithme de tri à bulles (*bubble*) séquentiel

Simple, mais pas un algorithme de tri séquentiel efficace.

Complexité de la comparaison/échange :

A close-up of a number

Description automatically generated

* Algorithme de tri pair-impair
* Tri à bulles parallélisé
* Basé sur l'idée que les corps de la boucle principale peuvent se chevaucher

Algorithme "scalaire" : Itération entre les **phases Paires et Impaires**

A table with numbers and arrows

Description automatically generatedA screenshot of a computer code

Description automatically generated

A screenshot of a computer code

Description automatically generated

* Algorithme de tri *Shear* (tri bidimensionnel)
* Considérer le tableau comme un tableau bidimensionnel (une rangée par processus).
* L'objectif est de trier ce tableau 2D de **manière serpentine** : les rangées **paires augmentent**, les rangées **impaires diminuent**.
* Deux phases : dans la **phase paire, tri par rangée** ; dans la **phase impaire, tri par colonne** augmentant de haut en bas.
* Après *log2(N) + 1* phases, le tableau est trié en style serpentin.

**Remarques :**

Algorithme embarassant parallèle pour la mémoire partagée.

A close-up of a number

Description automatically generatedA close-up of a number

Description automatically generatedA close-up of numbers

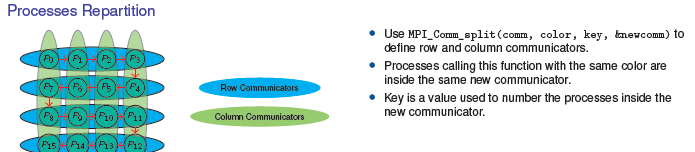
Description automatically generatedA close-up of a sign

Description automatically generatedA close-up of a logo

Description automatically generatedA close-up of a number

Description automatically generatedPas bien adapté pour une architecture parallèle distribuée telle quelle.

* Algorithme de tri *Shear* pour une architecture parallèle à mémoire distribuée
* Principe similaire à l'algorithme pair-impair : remplacer une valeur par des ensembles de valeurs *Si* (un ensemble par processus).
* Définir une relation d'ordre : *Si < Sj* si *max(Si) < min(Sj)* (Dans un ensemble, les valeurs sont ordonnées par ordre croissant).
* Utiliser l'algorithme pair-impair pour paralléliser la phase de tri par ligne ou par colonne.
* Regrouper les processus dans de nouveaux communicateurs par lignes et par colonnes.
* Jouer avec le numéro de rang pour alterner entre l'ordre croissant et l'ordre décroissant pour les lignes.



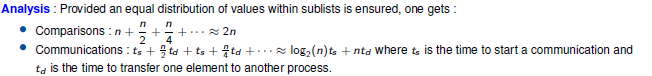
* Algorithme de tri rapide séquentiel

Rappel :

* Algorithme de tri séquentiel optimal, *O(n log2(n)) ,* basé sur la classe d'algorithmes diviser-pour-régner.
* Sélectionner un nombre *r* appelé pivot et diviser la liste en deux sous-listes : l'une avec tous les éléments inférieurs ou égaux à *r*, l'autre contenant tous les éléments supérieurs à *r*.
* Cette procédure est récursive, appliquée jusqu'à ce que des listes à un seul élément soient obtenues (qui sont triées).

A computer code with text

Description automatically generated with medium confidence

* Parallélisation naïve de l'algorithme de tri rapide
* La classe des algorithmes de tri rapide suggère d'appliquer une méthode de parallélisation diviser-pour-régner.
* Le principal problème avec cette approche est que la distribution de l'arbre (induite par les longueurs des sous-listes) dépend fortement de la sélection du pivot ; dans le pire des cas, l'arbre peut consister en un seul chemin (comme une liste).
* L'algorithme de tri Hyperquick

Basé sur la numérotation binaire de l'hypercube.

L'hypercube est un graphe utilisé pour modéliser les réseaux informatiques et les architectures parallèles. Chaque nœud de l'hypercube est associé à un numéro binaire unique. L'algorithme Hyperquick exploite cette structure pour effectuer un tri efficace. Il utilise la numérotation binaire des nœuds de l'hypercube pour organiser les données et effectuer des comparaisons en parallèle.

A diagram of a figure

Description automatically generatedEn attribuant à chaque élément à trier une adresse correspondant à un nœud de l'hypercube, l'algorithme peut comparer et échanger les éléments de manière efficace en exploitant les propriétés de l'hypercube. L'utilisation de la numérotation binaire de l'hypercube permet à Hyperquick d'atteindre des performances élevées en parallèle pour le tri de grandes quantités de données.

A black and green line with black text

Description automatically generatedA green and black text

Description automatically generated

A diagram of a number

Description automatically generatedA diagram of a diagram

Description automatically generated

Numérotation des Sommets de l'Hypercube :

* Les nombres binaires de deux nœuds liés ont une différence d'un bit
* La distance entre deux nœuds dans un hypercube (le nombre minimal de nœuds à parcourir pour aller du premier nœud au deuxième nœud) est le nombre de bits qui diffèrent dans leur nombre binaire
* C'est le numéro de **code Gray**

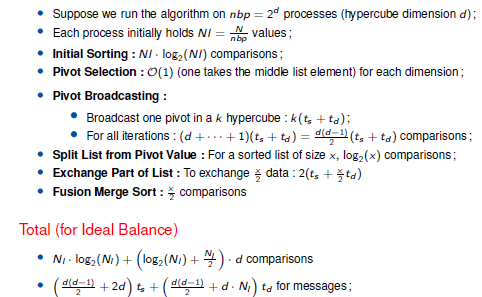
Idées de l'Algorithme de Tri Hyperquick :

* Initialement, les données sont réparties entre tous les processus
* Chaque processus trie ses données locales
* Boucle sur les dimensions de l'hypercube et pour la dimension *d*, considérez les paires de processus :
* Le processus *p* choisit sa valeur médiane comme pivot et envoie les valeurs inférieures au pivot au processus :

**

* Cet processus reçoit le pivot et les données du processus *p* et conserve les valeurs supérieures au pivot
* A diagram of a diagram

  Description automatically generatedChaque processus conserve des valeurs triées, en utilisant l'algorithme de fusion pour maintenir le tri



* Séquences bitoniques

Une séquence de valeurs qui peut être divisée en deux sous-séquences (de nombres consécutifs), l'une croissante et l'autre décroissante ; par exemple :



* Exemple : 3; 5; 8; 19; 17; 14; 12; 11

Ou une séquence qui peut être amenée (*estar obligada*) à cette forme par un décalage circulaire des éléments de la séquence.

* Exemple : 12; 11; 3; 5; 8; 19; 17; 14

**Remarques :**

* La première sous-séquence peut être croissante ou décroissante.
* Ainsi, une séquence bitonique (sans tenir compte du décalage circulaire) peut être croissante-décroissante ou décroissante-croissante.
* Une **séquence monotone bitonique** est une séquence **triée** (croissante ou décroissante).
* **Toutes les séquences avec trois éléments ou moins sont bitoniques**.

A math problem with numbers and equations

Description automatically generated with medium confidence

* Tri d'une séquence bitonique (algorithme **S**orting **B**itonic **S**equence)

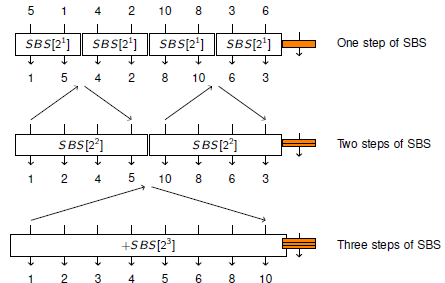
Appliquer la procédure de division sur la séquence bitonique et répéter cette procédure de division sur les sous-séquences jusqu'à obtenir un seul élément par sous-séquence pour obtenir une séquence triée.

A screenshot of a computer

Description automatically generated

Procédure pour construire une séquence bitonique :

1. Diviser la séquence en sous-séquences de deux éléments (qui sont bitoniques !)
2. Trier alternativement les sous-séquences dans l'ordre croissant et décroissant (en utilisant l'algorithme SBS)
3. Concaténer deux listes adjacentes pour obtenir une séquence bitonique plus longue
4. Répéter à partir de l'étape 2 jusqu'à ce que la liste complète devienne une séquence bitonique



Complexité du tri bitonique :

Avec *n = 2k*, il y a *k*  phases, chacune comprenant respectivement *1, 2, …, k* étapes.

Le nombre total d'étapes est :

A number and numbers with numbers and symbols

Description automatically generated with medium confidence

La complexité totale est :

Adapter le tri bitonique à une architecture parallèle distribuée :

* D'abord, trier les données locales avec l'algorithme de tri le plus rapide dans **l'ordre croissant** si le **rang est pair** et dans **l'ordre décroissant** si le **rang est impair**
* Regrouper les processus par paires à l'intérieur d'un sous-communicateur pour définir une séquence bitonique
* Appliquer l'algorithme de tri bitonique en regroupant les processus par quatre, huit, et ainsi de suite
* Les sous-communicateurs construits ici sont très similaires aux sous-communicateurs construits avec les algorithmes de tri hyperquick
* Algorithme de tri par compartiments (*Bucket Sort*)
* Mettre en place un **tableau de "compartiments"** initialement vides.
* **Dispersion :** Placer chaque élément de la liste dans le compartiment approprié.
* Trier les éléments à l'intérieur de chaque compartiment.
* **Rassemblement :** Visiter chaque compartiment dans l'ordre et remettre tous les éléments dans la liste d'origine.

A number of numbers in black and white

Description automatically generated with medium confidence

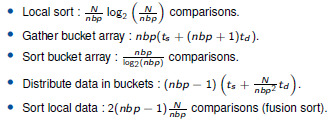


* Tri par compartiments parallèle

Chaque processus est considéré comme un compartiment.

Calculer les intervalles pour les compartiments :

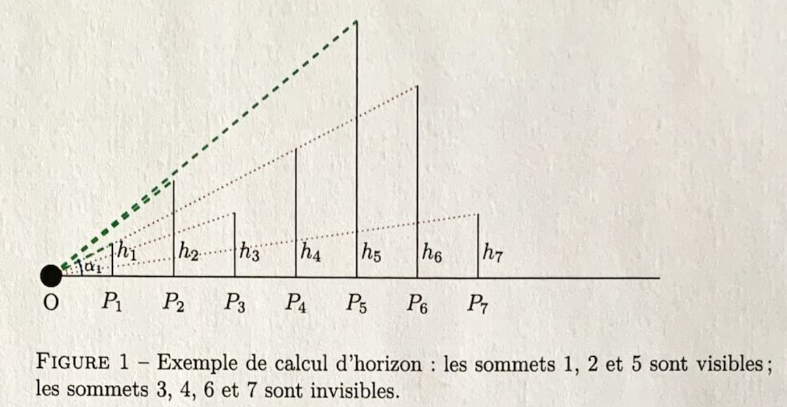
* Trier les données locales.
* Prendre *nbp + 1* valeurs à des intervalles réguliers.
* Rassembler les valeurs dans le tableau de compartiments.
* Extraire *nbp + 1* valeurs pour trouver les intervalles des compartiments.



**2023**

**A paper with text on it

Description automatically generated**



# Calcule les angles d'observation locaux pour chaque sommet

**angles\_loc = calcule\_angles(sommets\_loc)**

# Initialise les variables pour stocker les angles maximums

**prev\_init\_angl\_max = 0** # précédent

**cur\_init\_angl\_max = 0** # courant (à calculer)

# Si le processus n'est pas le premier

**if rank > 0:**

# Reçoit l'angle maximum calculé par le processus précédent

**prev\_init\_angl\_max = glob\_comm.recv(source=rank-1)**

**cur\_init\_angl\_max = prev\_init\_angl\_max**

# Parcours les angles locaux et met à jour l'angle maximum courant

**for i in range(sommets\_loc.shape[0]):**

# Reçoit l'angle du processus précédent

**a = glob\_comm.recv(source=rank-1)**

# Met à jour l'angle maximum courant si nécessaire

**cur\_init\_angl\_max = max(cur\_init\_angl\_max, a)**

# Si le processus n'est pas le dernier

**if rank < nbp-1:**

# Envoie l'angle maximum courant au processus suivant

**glob\_comm.send(cur\_init\_angl\_max, dest=rank+1)**

# Envoie chaque angle local au processus suivant

**for a in angles\_loc:**

# Envoie l'angle local au processus suivant

**glob\_comm.send(a, dest=rank+1)**

# Initialise l'angle maximum global

**angl\_max = cur\_init\_angl\_max**

# Initialise un tableau pour marquer les sommets visibles

**is\_visible = np.zeros(sommets\_loc.shape[0], dtype=np.int8)**

# Parcours les angles locaux

**for ia, a in enumerate(angles\_loc):**

# Vérifie si l'angle est plus grand que l'angle maximum courant

**if a > angl\_max:**

# Marque le sommet comme visible

**is\_visible[ia] = 1**

# Met à jour l'angle maximum global

**angl\_max = a**

* 1. Du point de vue purement mise en œuvre (sans optimiser l'algorithme lui-même), critiquez la façon dont la mise en œuvre parallèle a été faite en vous justifiant :
* **Communication Point-to-Point :** L'utilisation de communications point-à-point est raisonnable pour cet algorithme, car chaque processus doit communiquer directement avec ses voisins. Cependant, il est important de s'assurer que les communications sont efficaces et que les processus ne passent pas trop de temps à attendre les données.
* L'implémentation utilise des communications point-à-point pour échanger les angles entre les processus. Cela peut fonctionner efficacement pour un petit nombre de processus et une petite quantité de données. Cependant, pour un grand nombre de processus ou une grande quantité de données, les communications point-à-point pourraient devenir un goulot d'étranglement.
* **Synchronisation :** La synchronisation basée sur les communications directes entre les processus est simple et facile à implémenter. Cependant, cela peut entraîner des surcoûts de communication, en particulier si les processus doivent attendre les autres avant de pouvoir avancer. Une synchronisation plus fine-grain pourrait être nécessaire si les processus ont des charges de travail variables.
* L'approche utilisée pour synchroniser et mettre à jour les maximums locaux est simple mais efficace. Cependant, elle suppose que les maximums locaux peuvent être transmis à tous les autres processus sans problème. Cela pourrait fonctionner dans certains cas, mais cela peut également introduire des surcoûts de communication inutiles.
* **Gestion des Erreurs de Communication :** Bien que la gestion des erreurs de communication soit importante pour la robustesse de l'application parallèle, elle peut être considérée comme une optimisation de la fiabilité plutôt que de la performance pure.
* L'implémentation ne semble pas inclure de mécanismes de gestion des erreurs de communication. Si une erreur survient pendant la communication, le programme pourrait se retrouver dans un état instable.
* **Équilibrage de Charge :** Sans optimiser l'algorithme lui-même, l'équilibrage de charge peut être difficile à réaliser. Cependant, il est important de s'assurer que chaque processus reçoit une quantité de travail équitable pour éviter les goulets d'étranglement.
* L'implémentation ne semble pas prendre en compte l'équilibrage de charge entre les processus. Si les données sont réparties de manière inégale entre les processus, certains processus pourraient terminer leur travail plus rapidement que d'autres, ce qui sous-utiliserait les ressources disponibles.
* **Impact sur les Performances :** L'impact sur les performances dépendra de divers facteurs, y compris la taille des données, la topologie du réseau, etc. L'objectif ici serait de minimiser les surcoûts liés à la parallélisation tout en maximisant l'utilisation des ressources disponibles.
* Bien que l'objectif soit de ne pas optimiser l'algorithme lui-même, l'implémentation parallèle pourrait être optimisée pour réduire les surcoûts de communication et améliorer la scalabilité. Cela pourrait impliquer l'utilisation de communications collectives, une synchronisation plus fine-grain, ou des stratégies d'équilibrage de charge.
  1. Y avait-il un moyen d'améliorer l'algorithme? Quelle est la (les) donnée(s) vraiment nécessaire à transmettre entre processus? L'algorithme est\_il vraiment parallèle? Justifier votre révonse.
* **Améliorations de l'algorithme :**
* **Réduction des Communications :** L'algorithme actuel effectue beaucoup de communications entre les processus pour échanger les angles maximums. Une approche alternative pourrait être de calculer les maximums locaux sur chaque processus, puis de les réduire globalement pour obtenir le maximum global. Cela réduirait le nombre de communications nécessaires.
* **Équilibrage de Charge :** L'algorithme pourrait être amélioré en prenant en compte l'équilibrage de charge entre les processus. Si les données sont réparties de manière inégale, certains processus pourraient être surchargés tandis que d'autres seraient sous-utilisés.
* **Données Nécessaires à Transmettre :** Les données nécessaires à transmettre entre les processus sont principalement les angles maximums locaux. Cela permet à chaque processus de déterminer quels sommets sont visibles localement. Une fois que chaque processus a calculé son maximum local, il peut le transmettre aux processus voisins pour synchroniser les maximums globaux.
* **Parallélisme de l'Algorithme :** L'algorithme tel qu'il est actuellement mis en œuvre peut être considéré comme parallèle dans une certaine mesure. Chaque processus travaille de manière indépendante sur ses propres données locales et ne dépend pas directement des autres processus pour effectuer ses calculs. Cependant, la synchronisation des maximums locaux et la communication entre les processus introduisent des étapes de communication qui peuvent limiter le parallélisme effectif de l'algorithme.
  1. Quelle fonction de communication globale MPI aurait-pu servir pour simplifier et optimiser l'algorithme ?
* La fonction de communication globale MPI qui aurait pu servir dans ce cas est la fonction MPI\_Reduce(). Cette fonction permet de combiner les valeurs de chacun des processus pour obtenir une seule valeur agrégée. Dans le contexte de cet algorithme, chaque processus aurait pu calculer son maximum local et utiliser MPI\_Reduce() pour obtenir le maximum global parmi tous les processus.

**import numpy as np**

**from mpi4py import MPI**

# Initialisation de MPI

**comm = MPI.COMM\_WORLD**

**rank = comm.Get\_rank()**

**size = comm.Get\_size()**

# Supposons que chaque processus a des angles\_loc déjà calculés localement

# angles\_loc est un tableau numpy contenant les angles locaux sur chaque processus

# Calcule les angles d'observation locaux pour chaque sommet

**angles\_loc = calcule\_angles(sommets\_loc)**

# Calcule le maximum local des angles locaux

**max\_local = np.max(angles\_loc)**

# Utilise MPI\_Reduce() pour trouver le maximum global de tous les maximums locaux

**max\_global = comm.allreduce(max\_local, op=MPI.MAX)**

# Initialise un tableau pour marquer les sommets visibles

**is\_visible = np.zeros(angles\_loc.shape[0], dtype=np.int8)**

# Marque les sommets comme visibles si leurs angles locaux sont supérieurs au maximum global

**for i, a in enumerate(angles\_loc):**

**if a > max\_global:**

**is\_visible[i] = 1**

* Un résumé des **fonctions de communication globale MPI** les plus courantes et importantes pour **l'optimisation et la simplification des codes parallèles** :
* **MPI\_Bcast** : Permet de diffuser une même valeur depuis un processus (habituellement le root) vers tous les autres processus d'un groupe.
* **MPI\_Reduce** : Permet de réduire les données de tous les processus d'un groupe MPI vers un seul processus, en appliquant une opération spécifiée :
* MPI.SUM : Effectue la somme de tous les éléments.
* MPI.PROD : Effectue le produit de tous les éléments.
* MPI.MAX : Retourne le maximum de tous les éléments.
* MPI.MIN : Retourne le minimum de tous les éléments.
* MPI.LAND : Effectue un "et logique" sur tous les éléments.
* MPI.LOR : Effectue un "ou logique" sur tous les éléments.
* MPI.BAND : Effectue un "et bit à bit" sur tous les éléments.
* MPI.BOR : Effectue un "ou bit à bit" sur tous les éléments.
* MPI.MAXLOC : Retourne la valeur maximale et l'indice du premier élément ayant cette valeur.
* MPI.MINLOC : Retourne la valeur minimale et l'indice du premier élément ayant cette valeur.
* **MPI\_Allreduce** : Effectue une réduction globale comme MPI\_Reduce, mais les résultats sont renvoyés à tous les processus du groupe.
* **MPI\_Scatter** : Permet de diviser les données d'un processus (habituellement le root) en morceaux et de les distribuer à tous les autres processus d'un groupe.
* **MPI\_Gather** : Recueille les données de tous les processus d'un groupe MPI et les rassemble sur un processus (habituellement le root).
* **MPI\_Allgather** : Rassemble les données de tous les processus d'un groupe MPI et les distribue à tous les processus.
* **MPI\_Alltoall** : Distribue les données de chaque processus à tous les autres processus du groupe MPI, et reçoit les données des autres processus.

**2020**

A close-up of a math problem

Description automatically generated

Pour paralléliser le code **comp\_seuil** sur une machine à mémoire distribuée, nous devons diviser les données et distribuer les calculs entre les différents processeurs ou nœuds du système distribué. Voici comment nous pourrions procéder :

* **Partitionnement des données :**
* Nous divisons l'ensemble des données d'entrée **input** en sous-ensembles pour chaque processeur ou nœud du système distribué. Chaque nœud traitera une partie des données.
* **Distribution des données :**
* Une fois que les données sont partitionnées, nous les distribuons aux différents nœuds du système distribué. Chaque nœud reçoit sa propre partie des données à traiter.
* Parallélisation des calculs :
* Chaque nœud exécute la fonction **comp\_seuil** sur les données qui lui sont assignées localement. Cela signifie que chaque nœud parcourt seulement sa propre partie des données et effectue les calculs nécessaires.
* Synchronisation des résultats :
* Après que chaque nœud ait terminé le calcul, les résultats doivent être synchronisés. Cela peut impliquer la collecte des résultats partiels de chaque nœud et les regrouper pour obtenir le résultat final.
* Réunion des résultats :
* Une fois que tous les résultats partiels sont collectés, ils sont combinés pour former le résultat final de la fonction de seuil.

**#include <stdio.h>**

**#include <mpi.h>**

// Définition de la fonction de seuil

**void comp\_seuil(int local\_n, const double\* local\_input, double s, double\* local\_output) {**

// Parcours des données locales

**for (int i = 0; i < local\_n; ++i) {**

// Application du seuil

**if (local\_input[i] < s)**

**local\_output[i] = 0; // Si la valeur est inférieure au seuil, la sortie est 0**

**else**

**local\_output[i] = 1; // Sinon, la sortie est 1**

**}**

**}**

**int main(int argc, char \*argv[]) {**

**int rank, size;**

**const int n = 1000;** // Taille totale des données

**const double seuil = 5.0;**

**double input[n], output[n];**

**MPI\_Init(&argc, &argv);** // Initialisation de l'environnement MPI

// Récupération du rang (identifiant) du processus courant

**MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);**

// Récupération du nombre total de processus lancés

**MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);**

// Calcul du nombre d'éléments par processus

**int local\_n = n / size;**

// Déclaration des tableaux locaux pour les données d'entrée et de sortie

**double local\_input[local\_n];**

**double local\_output[local\_n];**

// Distribution des données d'entrée à tous les processus

**MPI\_Scatter(input, local\_n, MPI\_DOUBLE, local\_input, local\_n, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);**

// Calcul parallèle de la fonction de seuil sur les données locales

**comp\_seuil(local\_n, local\_input, seuil, local\_output);**

// Rassemblement des résultats de tous les processus sur le processus maître

**MPI\_Gather(local\_output, local\_n, MPI\_DOUBLE, output, local\_n, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);**

// Affichage des résultats sur le processus maître

**if (rank == 0) {**

**printf("Résultats :\n");**

**for (int i = 0; i < n; ++i) {**

**printf("%lf ", output[i]);**

**}**

**printf("\n");**

**}**

// Finalisation de l'environnement MPI

**MPI\_Finalize();**

**return 0;**

**}**

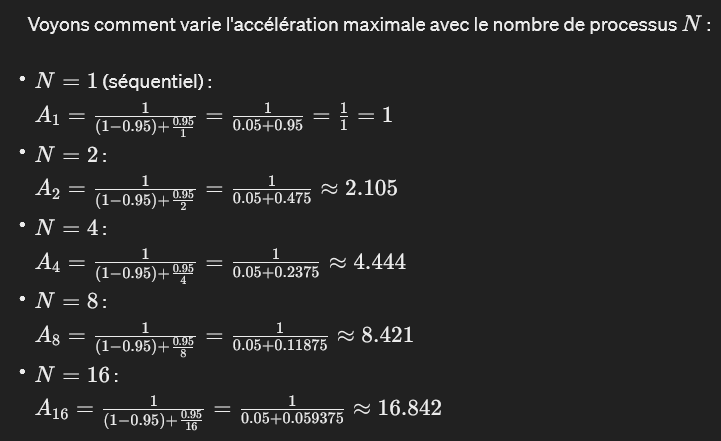
A close up of a text

Description automatically generated

A black background with white text

Description automatically generatedPour calculer l'accélération maximale et déterminer le nombre raisonnable de processus pour une bonne efficacité, nous devons utiliser la **loi de Amdahl**, qui décrit la relation entre l'accélération obtenue parallélisant une tâche et la fraction de cette tâche qui peut être parallélisée.

La loi de Amdahl est définie comme suit :



Plus le nombre de processus NN augmente, plus l'accélération maximale augmente. Cependant, il y a des limites à cela en raison de l'overhead de communication et des ressources matérielles disponibles.

A computer code on a white background

Description automatically generatedLe nombre raisonnable de processus pour une bonne efficacité dépendra de divers facteurs, tels que la capacité de la machine, la taille des données, les ressources matérielles et la capacité du programme à tolérer la parallélisation. Dans ce cas, étant donné que 95% du programme est parallélisable, une forte accélération peut être obtenue avec un nombre raisonnable de processus, tout en évitant un surcoût significatif en termes de temps d'exécution et d'utilisation des ressources.

Le problème dans le programme de Fabrice réside dans la logique de communication entre les processus MPI.

Voici les raisons pour lesquelles le programme ne fonctionne pas correctement avec plus de deux processus :

1. Blocage sur MPI\_Recv() dans le processus 0 :

* Lorsque le programme est exécuté avec plus de deux processus, tous les processus autres que le processus 0 n'envoient jamais de message.
* Le processus 0 utilise MPI\_Recv() pour recevoir un message. Cependant, il attend un message de n'importe quelle source avec n'importe quelle étiquette (MPI\_ANY\_SOURCE, MPI\_ANY\_TAG). Cela signifie que le processus 0 attend un message de n'importe quel autre processus.
* Comme les autres processus ne lui envoient jamais de message, le processus 0 reste bloqué indéfiniment sur MPI\_Recv().

1. Envoi sans initialisation de la variable token :

* Dans tous les processus autres que le processus 0, la variable token est utilisée dans MPI\_Send() sans avoir été initialisée au préalable.
* Cela peut entraîner un comportement indéterminé, car la valeur de token n'est pas définie avant d'être envoyée.

Pour corriger ces problèmes :

* Pour le blocage sur MPI\_Recv(), on doit modifier la logique du programme pour que le processus 0 ne commence à attendre un message que lorsqu'il a réellement besoin de le faire, c'est-à-dire après que les autres processus ont envoyé leur message.
* Pour l'envoi du message, initialiser la variable token avec une valeur appropriée avant d'appeler MPI\_Send().

**#include <iostream>**

**#include <mpi.h>**

**int main() {**

**int rank, nbp;**

**MPI\_Init(NULL, NULL);**

**MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);**

**MPI\_Comm\_size(MPI\_COMM\_WORLD, &nbp);**

**int token;**

**if (rank == 0) {**

**for (int source = 1; source < nbp; source++) {**

**MPI\_Recv(&token, 1, MPI\_INT, source, 101, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);**

**std::cout << "Token\_recu de " << source << " : " << token << std::endl;**

**}**

**} else {**

**token = rank;** // Par exemple, chaque processus envoie son propre rang

**MPI\_Send(&token, 1, MPI\_INT, 0, 101, MPI\_COMM\_WORLD);**

**}**

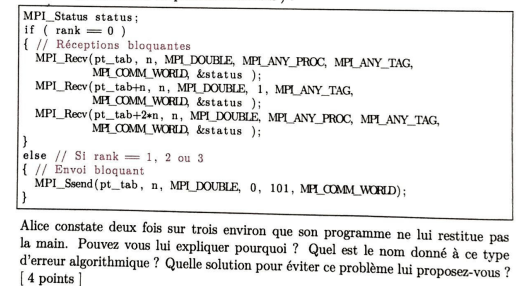
**MPI\_Finalize();**

**return 0;**

**}**

**2019**





Le problème que rencontre Alice est dû à ce qu'on appelle le "blocage", ou "deadlock" en anglais, dans le contexte des systèmes distribués et de la programmation parallèle.

Un deadlock se produit lorsque deux ou plusieurs processus sont incapables de continuer leur exécution parce qu'ils attendent chacun une ressource que les autres possèdent. Dans le cas présent, Alice utilise des réceptions bloquantes MPI\_Recv et des envois bloquants MPI\_Ssend.

Le deadlock survient lorsque le processus 0 attend les messages des processus 1, 2 et 3 via les réceptions bloquantes MPI\_Recv. Cependant, les processus 1, 2 et 3 sont tous bloqués en attente d'envoyer des données à 0 via MPI\_Ssend. Cela crée une impasse (*estancamiento*) où aucun processus ne peut progresser.

La raison principale de ce deadlock est que MPI\_Recv est une opération bloquante qui attend un message jusqu'à ce qu'il soit disponible. De plus, MPI\_Ssend est également bloquant et nécessite une réception correspondante pour compléter l'envoi.

Pour éviter ce problème, Alice peut utiliser des réceptions non-bloquantes MPI\_Irecv combinées avec MPI\_Wait pour vérifier si les messages ont été reçus. De cette façon, les processus peuvent continuer à exécuter d'autres tâches pendant qu'ils attendent les messages. Alternativement, elle peut utiliser MPI\_Send à la place de MPI\_Ssend, qui est non bloquant par défaut et ne nécessite pas de réception correspondante.

**MPI\_Status status;**

**if (rank == 0) {**

**// Réceptions non-bloquantes**

**MPI\_Irecv(pt\_tab, n, MPI\_DOUBLE, MPI\_ANY\_PROC, MPI\_ANY\_TAG, MPI\_COMM\_WORLD, &request[0]);**

**MPI\_Irecv(pt\_tab + n, n, MPI\_DOUBLE, 1, MPI\_ANY\_TAG, MPI\_COMM\_WORLD, &request[1]);**

**MPI\_Irecv(pt\_tab + 2 \* n, n, MPI\_DOUBLE, MPI\_ANY\_PROC, MPI\_ANY\_TAG, MPI\_COMM\_WORLD, &request[2]);**

**MPI\_Waitall(3, request, status);** // Attente de la réception de tous les messages

**}**

**else { // Si rank == 1, 2 ou 3**

**// Envoi bloquant**

**MPI\_Ssend(pt\_tab, n, MPI\_DOUBLE, 0, 101, MPI\_COMM\_WORLD);**

**}**

**A black text on a white background

Description automatically generated**

Il semble que Bob se heurte (*se enfrenta*) à la loi d'Amdahl. La loi d'Amdahl, formulée par Gene Amdahl en 1967, stipule que le gain de performance qu'on peut obtenir en parallélisant un programme est limité par la proportion de code séquentiel dans ce programme.

A black background with white text

Description automatically generatedLa loi d'Amdahl est formulée mathématiquement comme suit :

Où :

* Speedup représente le gain de performance obtenu en parallélisant le programme.
* P est la proportion du code pouvant être parallélisée.
* N est le nombre de processus.

Dans le cas de Bob, le fait qu'il ne parvienne pas à diviser le temps par plus de vingt, quel que soit le nombre de processus, indique qu'il rencontre une limite imposée par la partie séquentielle de son programme.

Si Bob obtient un speedup maximal de 20, cela signifie que la proportion de son code traité en parallèle est d'environ 95% et la proportion traitée séquentiellement est d'environ 5%. Cela signifie que même s'il ajoute davantage de ressources parallèles, la partie séquentielle de son programme continue à limiter le gain de performance global.

Il est crucial pour Bob de réduire la proportion du code séquentiel pour améliorer davantage les performances parallèles de son programme. Cela peut impliquer des efforts pour identifier et optimiser les parties séquentielles du code, ou même envisager des approches algorithmiques différentes pour réduire la dépendance séquentielle.

* on suppose que Bob a à disposition un nombre illimité de nœuds de calcul et que le temps passé à échanger les messages est négligeable devant le temps de traitement des données

Cela signifie qu'il peut exécuter son programme de manière parallèle sur un nombre très élevé de processeurs ou de nœuds sans rencontrer de contraintes matérielles liées aux ressources de calcul.

Cela signifie que les opérations de communication, telles que l'envoi et la réception de messages entre les nœuds, n'ont pas d'impact significatif sur le temps global d'exécution du programme par rapport au temps nécessaire pour effectuer le traitement réel des données.

Dans un environnement de calcul parallèle, les opérations de communication entre les nœuds peuvent souvent devenir un goulot d'étranglement, limitant les performances globales du programme. Cependant, dans le cas de Bob, cette limitation est supposée être insignifiante par rapport à la capacité de traitement parallèle offerte par l'infrastructure.

Si tout le code est séquentiel, il n'y aura aucun gain de vitesse réalisable en parallélisant le code. La loi d'Amdahl fournit un aperçu de ce scénario. Si tout le code est séquentiel, Ps=1 et Pp=0. Dans ce cas, la loi d'Amdahl se simplifie à Speedup=N, où N représente le nombre de processeurs. Cela signifie que le gain de vitesse serait égal au nombre de processeurs NN. Cependant, le temps d'exécution réel ne diminuerait pas linéairement avec le nombre de processeurs car le code est purement séquentiel. L'ajout de processeurs supplémentaires ne conduirait à aucune amélioration de performance.

D'autre part, si tout le code était parfaitement parallélisable, Ps=0 et Pp=1. Dans ce cas, la loi d'Amdahl se simplifie à Speedup=1, ce qui signifie que le programme fonctionnerait aussi vite qu'il le pourrait sur un seul processeur. Ajouter plus de processeurs ne fournirait aucun avantage car tout le code est déjà parallélisé et exécuté simultanément sur tous les processeurs disponibles.

A white background with black text

Description automatically generated

La réponse à cette question dépend de divers facteurs, notamment la taille des données à transférer, la structure de l'algorithme et les performances du système sous-jacent. Voici une analyse des deux approches :

* Copier dans un tableau dédié des valeurs non contiguës et l'envoyer à un autre processus :

Avantages :

* La copie des valeurs dans un tableau dédié peut permettre une manipulation plus facile et une gestion plus efficace des données avant l'envoi.
* L'envoi d'un seul tableau peut réduire le nombre d'appels aux fonctions de communication, ce qui peut être plus efficace dans certains cas.

Inconvénients :

* La copie des valeurs dans un tableau dédié peut nécessiter des opérations de mémoire supplémentaires, ce qui peut entraîner une surcharge de traitement, en particulier pour de grandes quantités de données.
* Envoyer ces mêmes valeurs une à une à un autre processus sans les copier préalablement dans un tableau dédié :

Avantages :

* Évite la surcharge de copie des données dans un tableau dédié.
* Peut être plus efficace pour les petits ensembles de données ou lorsque la structure de l'algorithme nécessite un traitement élément par élément.

Inconvénients :

* Le transfert de données élément par élément peut entraîner une surcharge de communication en raison du surcoût associé à chaque appel de fonction de communication.
* Peut être moins efficace pour les grandes quantités de données en raison de l'overhead de communication supplémentaire.

En général, si la taille des données à transférer est petite et qu'une manipulation simplifiée des données est souhaitée, l'approche 2 peut être plus efficace. Cependant, pour des ensembles de données plus importants ou des structures de données complexes, l'approche 1 peut offrir une meilleure performance en réduisant le nombre total d'opérations de communication et en permettant une manipulation plus efficace des données.