Parallel Programming Sheet

Alors, qu'est-ce que le calcul parallèle ? -> Tout ordinateur où des tâches sont exécutées simultanément pour résoudre un problème

• Parallélisme embarrassant : pas de communication entre les tâches

• S'il y a une communication entre les tâches, il existe de nombreuses façons de le faire

• À ne pas confondre avec le multitraitement, où plusieurs tâches non liées exécuter

• Pourquoi le calcul parallèle ? -> Parce que tous les ordinateurs sont désormais parallèles !

• Combien de cœurs y a-t-il dans mon ordinateur portable ?

# History

* ACM Gordon Bell Prize
* 2022: 3 supercomputers – plasma simulation
* 2021: 41M cores, quantum circuit simulation- ?? MW
* 2020: 138M cores, 3.6 Pflops, COVID molecule – 13 MW
* 2019: 138M cores, 85 Pflops, quantum simulation – 13 MW
* 2018: 2.4M cores, 2.36 ExaOps, genomics research- 9.7 MW
* 2017: 10.5M cores, 19 Pflops, earthquake simulation – 15 MW
* 2016: 10.5M cores, 8 Pflops, 500 m resolution atmospheric model with 770 billion unknowns, 0.07 years per day – 15 MW
* 2015: 1.5M cores, earth’s mantle movement, 602 billion unknowns – 7MW
* 2014: Anton-2 custom machine for molecular dynamics: 23k atoms 85 us/day

# Physical limits

* Transistor count : more and more (betw 1E+03 in 1971 to more than 10E+10 in 2013)
* Core count : stagne to 1E+0 betw 1971 and 2004 and up to 1E+01 in 2013
* Frequency (MHz) : stagne betw 1E+03 and 1E+04 a partir de 2004
* Power (W) : stagne at 1E+02 a partir de 2004
* Top 500 #1 power (W) : to 1E+05 to 1E+07 betw 1971 and 2013

En théorie, utiliser davantage de ressources vous permettra d'obtenir la solution plus rapidement pour résoudre des problèmes plus vastes ou plus complexes. De nombreux problèmes sont trop complexes pour être résolus sur un seul ordinateur (par exemple, impossible tenir en mémoire, cela prendrait trop de temps)

Assurer la simultanéité -> Un seul ordinateur ne peut faire qu'une seule chose à la fois, plusieurs ressources peuvent le faire beaucoup de choses simultanément

# Parallel computing : Analogie avec un puzzle – Henry Neeman @ Oklahoma University

• Informatique série : supposons que vous vouliez le faire vous-même – 1 000 pièces prend environ une heure

• Supposons que votre ami s'assoie et essaie de vous aider : vous devez maintenant communiquer lorsque vous travaillez sur des pièces proches, et parfois vous atteignez la pile en même temps : luttez pour les ressources. Vous avez terminé. Le puzzle en 35 minutes au lieu de 30

Plus on est de fous, plus on rit? Supposons que deux autres amis s'assoient et essaient de vous aider : l'accélération n'est pas significativement inférieure à 4x – disons 3x et vous terminez en 20 minutes

Rendements décroissants : à mesure que de plus en plus d'amis se joignent, il y aura beaucoup plus de conflits et de besoins de communication – ajouter plus de ressources pour résoudre le même problème entraîne des rendements décroissants.

# Distributed parallelism

Essayons de diviser les pièces du puzzle entre deux tableaux et de résoudre le problème de cette façon : il n'y a plus de conflit, mais le coût de la communication est beaucoup plus élevé – vous devrez combiner vos résultats à la fin.

De plus, vous devez diviser vos pièces de puzzle (décomposer) au début. Selon le temps que vous avez consacré à un « bon partage », la communication ultérieure peut être plus ou moins.

Répartition parfaite avec des pièces sur les deux côtés du puzzle final, par rapport à une répartition aléatoire : il est beaucoup plus facile à la fin d'assembler les deux moitiés, mais beaucoup plus de travail au début

La décomposition initiale détermine l'équilibrage de charge : vous essayez de donner à chacun la même quantité de travail à faire. Si l’image est moitié herbe moitié ciel, c’est facile, et vous devrez communiquer uniquement sur la limite herbe-ciel

Certains problèmes se prêtent à de bonnes décompositions et à un bon équilibre de charge.

L'équilibrage de charge peut devenir difficile, en particulier si Jane est plus douée pour résoudre des énigmes que Joe.

# Serial computing / Parallel computing

Serial computing : problème décomposé en une suite d'instructions, exécutées en série et dans l'ordre sur un seul processeur. La réalité est différente depuis longtemps...

Parallel computing : Utilisation simultanée de plusieurs ressources. Problème divisé en parties pouvant être résolues simultanément, puis chacune en une séquence d'instructions. Coordination centrale.

Le problème informatique doit être capable de :

* Être divisé en tâches distinctes qui peuvent être résolues simultanément (indépendamment ?) ;
* Exécuter plusieurs instructions à tout moment ;
* Être résolu en moins de temps en utilisant plusieurs ressources qu'avec une seule ressource informatique.

Quelles sont les « ressources de calcul » :

* Un seul ordinateur avec plusieurs processeurs/cœurs.
* Un réseau de ces ordinateurs, connectés.

# Neumann architecture

Stored program computer – le programme et les données sont conservés dans la même mémoire.

* La mémoire stocke les instructions et les données
* La Control Unit récupère les instructions ou les données de la mémoire, décode les instructions et coordonne séquentiellement les opérations.
* L’Arithmetic Logic Unit (ALU) fait les calculs
* (Arithmetic Logic Unit + Control Unit = CPU)
* Ce sont les trois éléments auxquels vous devrez penser par rapport à votre projet.
* Les ordinateurs parallèles suivent toujours cette conception, simplement multipliés en unités.

# Comment faire des architectures Neumann parallèles ?

Les processeurs monocœur traditionnels ne sont plus évolutifs en raison de la limite de fréquence, mais les consommateurs s'attendent toujours à une croissance des performances.

Avec la mise à l'échelle des processus (32-16-12 nm), vous pouvez mettre plus de transistors sur la puce. Comment allez-vous les utiliser ?

* Mettez plusieurs cœurs de processeur sur la même puce – ils peuvent désormais fonctionner indépendamment, mais ils voient toujours la même mémoire
* Mais les cœurs de processeur ont de nombreux circuits logiques – une grande unité de contrôle – qui les aident à fonctionner rapidement
  + Exécution dans le désordre, prédiction de branchement, etc.
  + Vous avez beaucoup de temps système pour alimenter l'unité arithmétique et logique.
* Atténuez le temps système de l'unité de contrôle, en lui faisant contrôler de nombreuses ALU.
  + Calculez les mêmes choses (même instruction), juste sur des données différentes.
* Caches plus grands et plus rapides

# Classification des ordinateurs parallèles

Flynn’s taxonomy, deux dimensions : flux d'instructions et flux de données, chacun peut être unique ou multiple.

* SISD: Single Instruction stream, Sigle Data stream - Flux d'instructions unique, flux de données unique
  + Ordinateur série classique
  + Instruction unique : un seul flux d'instructions est traité par le processeur au cours d'un cycle d'horloge
  + Données uniques : un seul flux de données est utilisé comme entrée au cours d'un cycle d’horloge.

Une image contenant texte, capture d’écran, ligne, diagramme

Description générée automatiquement

* SIMD: Single Instruction stream, Multiple Data stream - …
  + Instruction unique : toutes les unités de traitement exécutent la même instruction à un cycle d'horloge donné.
  + Données multiples : chaque unité de traitement peut opérer sur une donnée différente
  + Exécution synchrone (lockstep), idéale pour les calculs très réguliers

Une image contenant texte, capture d’écran, Police, diagramme

Description générée automatiquement Une image contenant capture d’écran, ligne, Caractère coloré, nombre

Description générée automatiquement

* MISD: Multiple Instruction stream, Single Data stream - …
  + Instruction Multiple : chaque unité de traitement opère sur les données indépendantes
  + Données uniques : un flux de données unique est introduit dans plusieurs unités de traitement
  + Pas vraiment utilisé

Une image contenant texte, capture d’écran, Police, diagramme

Description générée automatiquement

* MIMD: Multiple Instruction stream, Multiple Data stream - …
  + Instruction multiple : chaque processeur exécute un flux d'instructions différent.
  + Données multiples : chaque processeur fonctionne avec un flux de données différent
  + Synchrone/asynchrone
  + La plupart des supercalculateurs, processeurs multicœurs

Une image contenant texte, capture d’écran, diagramme, Police

Description générée automatiquement

# Programming parallel hardware

Nous avons maintenant une idée du fonctionnement des ordinateurs parallèles et des différentes manières dont le parallélisme peut se produire. Comment écrivons-nous du code pour les utiliser ?

Il s'avère que les modèles de programmation parallèle sont très proches du fonctionnement du matériel parallèle. Ce qui est logique : plus ils correspondent, plus il est facile de mapper le code à un ou plusieurs flux d'instructions et meilleures sont les performances auxquelles vous pouvez vous attendre.

* SISD – tous les langages classiques, C/C++, Fortran, Java, etc.
* SIMD – méthodes de compilation (transparentes) et langages comme CUDA, OpenCL
* MIMD – multi-threading avec OpenMP/TBB, etc., plusieurs processus avec MPI.

Cependant, un calcul parallèle spécifiant n’est qu’un côté de la médaille, la spécification des communications est peut-être encore plus importante. -> Nous distinguons les modèles de mémoire partagée et les modèles de mémoire distribuée

# OpenMP

OpenMP, qui signifie Open Multi-Processing, est un ensemble de directives et d'API (Interfaces de Programmation d'Applications) pour la programmation parallèle sur des systèmes à mémoire partagée. Il est conçu pour simplifier le développement d'applications parallèles en fournissant un modèle de haut niveau pour exprimer le parallélisme dans les programmes C, C++ et Fortran. Examinons les principaux composants et concepts :

* Multi-traitement ouvert
* Trois composantes : Directives du compilateur, Routines de la bibliothèque d'exécution, Variables d'environnement
* Modèle de haut niveau -> Cartographie implicite et équilibrage de charge du travail
* Standard (norme)
* Portable (peut être exécuté sur divers systèmes sans modification significative.)

## OpenMP view of memory

Une image contenant texte, capture d’écran, ligne, Rectangle

Description générée automatiquement

## Directives du compilateur

Annotations spéciales ajoutées au code source pour guider le compilateur dans la génération de code parallèle. Elles apparaissent sous forme de commentaires dans le code et ignorées par le compilateur sauf indication contraire (indicateur du compilateur)

Les directives OpenMP sont utilisées à diverses fins :

* Génération d'une région parallèle
* Division des blocs de code entre les threads
* Distribution des itérations de boucle entre les threads
* Sérialisation sections de code • Synchronisation du travail entre les threads

Ces directives sont précédées du mot-clé #pragma et fournissent des informations au compilateur sur la manière de paralléliser des sections spécifiques du code.

Syntaxe : Sentinel directive-name [clause, ...] #pragma omp parallel default(shared) private(beta,pi)

## Routine de bibliothèque d'exécution

Aident à la gestion de l'exécution parallèle. Ces routines gèrent des tâches telles que la création et la gestion de threads parallèles, la synchronisation et le partage de données entre les threads.

## Variables d'environnement

OpenMP fournit plusieurs variables d'environnement pour contrôler l'exécution du code parallèle. Peut être utilisé pour :

* Définir le nombre de threads « export OMP\_NUM\_THREADS=4 »
* Spécifier comment les itérations de boucle sont divisées
* Lier les threads aux processeurs physiques
* Contrôler le parallélisme imbriqué, les threads dynamiques

## Vector add

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

## Chronométrer votre code

* Dans les cours précédents, nous avons rarement examiné le temps nécessaire pour exécuter quelque chose.
* Au cours de ce cours, c'est l'une des mesures les plus importantes de nos performances.
* Bibliothèque chrono C++11

Une image contenant texte, Police, capture d’écran

Description générée automatiquement

# Compiler le code

Installer g++ et lancer les commandes suivantes :

* g++ [fichier.cpp] -o [sortie.exe]
* ./[sortie.exe]

Compiling faster code -> Add the “-Ofast” flag

Compiling with OpenMP support -> Add the “-fopenmp” flag

# Utiliser Komondor - Hungarian supercomputer

* Based in Debrecen, 128 CPU server, 58 GPU servers (4x NVIDIA A100) (<https://docs.hpc.kifu.hu/>)
* Running stuff with SLURM: <https://docs.hpc.kifu.hu/first-steps/slurm.html>
* Interactive jobs & batch jobs
* S’enregistrer – <https://portal.hpc.kifu.hu>
* Connection SSH via VS Code : Two-factor authentication

Setting up

* Add the following to the ssh config
  + Host komondor
  + User p\_cov
  + HostName komondor.hpc.kifu.hu
* Save, then refresh
* Find this line after clicking details – open in browser, authenticate, back to VS code, hit enter in top dialog
* Créer un nouveau fichier/repertoire
* Pour le lancer, ouvrir une session interactive :
  + srun --reservation=p\_covidpre\_115 -p cpu -c 4 --mem-per-cpu=2000 -- time=1:00:00 --pty bash
    - “--reservation=p\_covidpre\_115” -> réservation toutes les semaines
    - “-c 4” -> 4 cores
    - “--mem-per-cpu=2000” -> 2GB/core
    - “--time=1:00:00” -> 1 heure
* Reservation during class
* Without --reservation for assignments (may have to wait)

# Cours 2

Pour paralléliser une application, il est essentiel de comprendre le problème sous-jacent et d'identifier les points chauds ou les goulots d'étranglement, c'est-à-dire les parties du code où la majorité du temps est dépensée et où le vrai travail est effectué. Une fois ces zones identifiées, il est possible d'analyser l'algorithme pour déterminer s'il peut être parallélisé. Certains algorithmes se prêtent mieux à la parallélisation que d'autres.

Pour déterminer la parallélisabilité d'un algorithme, il est important de considérer si les tâches peuvent être effectuées de manière indépendante et simultanée. Par exemple, le calcul de l'énergie potentielle pour différentes conformations moléculaires peut être parallélisé, tandis que le calcul de la suite de Fibonacci n'est pas parallélisable car il dépend des résultats précédents.

Les étapes typiques pour paralléliser une application comprennent l'identification des parties du travail pouvant être effectuées simultanément, la répartition de ces tâches sur des processeurs indépendants, la coordination des accès aux données partagées pour éviter les conflits et assurer le bon ordre de travail en utilisant la synchronisation. Dans certains cas, certaines de ces étapes ne sont pas nécessaires, et le mappage du travail sur les processeurs peut être effectué manuellement ou automatiquement, en fonction des besoins spécifiques de l'application.

Dans le contexte du produit matrice-vecteur dense, chaque élément du vecteur de sortie y peut être calculé de manière indépendante. Ainsi, il est possible de décomposer le produit matrice-vecteur dense en tâches, avec une tâche associée à chaque élément dans y. Cette approche présente une granularité uniforme des tâches et aucune dépendance de contrôle entre elles.

Une image contenant ligne, diagramme, carré, texte

Description générée automatiquementUne image contenant croquis, dessin, blanc, diagramme

Description générée automatiquement

Lors de la création de tâches, le travail est divisé en un certain nombre de tâches, et il est important d'identifier comment ces tâches sont interdépendantes. Il existe de nombreuses façons de décomposer les tâches, avec des tâches pouvant être de même taille, de tailles différentes ou variables. Cette conceptualisation peut souvent être représentée par un graphe acyclique orienté (DAG).

La granularité des tâches, ou la taille de chaque tâche, peut être fine ou grossière. Une granularité fine implique un grand nombre de petites tâches, tandis qu'une granularité grossière implique un petit nombre de grandes tâches. Pour le produit matrice-vecteur dense, une approche fine serait d'avoir une tâche par élément de y, tandis qu'une approche grossière pourrait impliquer une tâche par 3 éléments de y.

Une image contenant capture d’écran, ligne, diagramme, carré

Description générée automatiquement

Le degré de concurrence fait référence au nombre de tâches pouvant s'exécuter en parallèle à un moment donné. Ce nombre peut varier pendant l'exécution. Des métriques telles que le degré de concurrence maximal et le degré de concurrence moyen peuvent être utilisées pour évaluer la performance parallèle. Il existe une relation inverse entre le degré de concurrence et la granularité des tâches.

Le concept de chemin critique est important pour évaluer les performances parallèles. Dans un graphe des dépendances des tâches, une arête représente la sérialisation des tâches, et le chemin critique est le plus long chemin à travers le graphe. Il représente une limite inférieure sur le temps d'exécution parallèle et est crucial pour comprendre les performances potentielles de l'application parallèle.

Exemple 1 : Nombre de poignées de main

* En série : Lorsque N personnes se serrent la main en série, cela signifie que chaque personne serre la main de toutes les autres. Ainsi, le nombre total de poignées de main serait calculé par la somme des nombres de poignées de main entre chaque paire de personnes, ce qui est équivalent à la somme des nombres de 1 à N-1 (car une personne ne peut pas se serrer la main).
* En parallèle : Si toutes les personnes se serrent la main simultanément, chaque personne peut potentiellement serrer la main de toutes les autres en même temps, sans attendre. Ainsi, le nombre total de poignées de main pourrait être calculé de manière plus simple en utilisant une formule directe, sans considérer les interactions séquentielles entre les personnes.

Exemple 2 : Dynamique moléculaire

L'exemple des atomes dans une simulation de dynamique moléculaire implique de calculer la force totale exercée sur chaque atome.

Chaque atome exerce une force sur tous les autres atomes en fonction de leur distance et de leurs propriétés. En utilisant un modèle de force approprié, tel que le potentiel de Lennard-Jones ou les interactions électrostatiques, on peut calculer la force entre chaque paire d'atomes.

* En parallèle, cela pourrait signifier que chaque calcul de force entre chaque paire d'atomes peut être effectué indépendamment et simultanément, puis les forces peuvent être sommées pour chaque atome afin de déterminer la force totale sur chaque atome.
* En série, chaque calcul de force entre chaque paire d'atomes serait effectué séquentiellement, ce qui pourrait prendre beaucoup plus de temps si le système comporte un grand nombre d'atomes.

La loi d'Amdahl est une loi empirique utilisée pour prédire la vitesse d'exécution d'un programme parallèle en fonction de la proportion de travail qui peut être parallélisée. Elle est nommée d'après Gene Amdahl, un informaticien qui l'a développée en 1967.

La loi d'Amdahl stipule que même dans un programme parallèle, il y a souvent des parties qui doivent être exécutées séquentiellement et ne peuvent pas être parallélisées. Par conséquent, la vitesse totale d'exécution du programme dépend de la vitesse de ces parties séquentielles ainsi que de la vitesse des parties parallélisables.

Formellement, la loi d'Amdahl est exprimée comme suit :

Où :

* est la proportion du programme qui peut être parallélisée.
* est le nombre de processeurs ou de cœurs de calcul disponibles.

Lorsque P=0, cela signifie que tout le travail est séquentiel et ne peut pas être parallélisé. Dans ce cas, le speedup total est égal à 1, ce qui signifie qu'il n'y a pas de gain de vitesse parallèle.

Lorsque P=1, cela signifie que tout le travail est parallélisable. Dans ce cas, le speedup total est théoriquement infini, ce qui signifie que le programme pourrait théoriquement s'exécuter instantanément avec un nombre suffisamment grand de processeurs.

Plus est élevé, plus le gain de vitesse parallèle est important. Cependant, même avec une valeur élevée de , il y a toujours une limite au speedup total en raison des coûts de communication, de synchronisation et d'autres facteurs liés à la parallélisation.

Une image contenant texte, capture d’écran, Tracé, ligne

Description générée automatiquement

Dans de nombreux cas, tenter de résoudre un problème plus important est bénéfique : le surcoût sériel reste constant. Par exemple, pour une computation PDE à deux facteurs, pour une taille donnée : en doublant les points de discrétisation dans chaque dimension et en doublant les pas de temps, un calcul en 2D prendrait 85 secondes, avec 85% du temps utilisé pour le calcul et 15% pour le surcoût sériel. En doublant ces paramètres, le calcul en 2D prendrait 680 secondes, avec 97,84% du temps utilisé pour le calcul et 2,16% pour le surcoût sériel. Cela mène à la Loi de Gustafson.

Nous disposons de plusieurs mesures théoriques telles que le rapport entre les parties séquentielles et parallèles (Loi d'Amdahl), le degré moyen / maximal de parallélisme, la granularité des tâches, etc. Cependant, ces mesures n'aident que dans l'analyse des algorithmes parallèles et donnent des bornes théoriques. Elles offrent peu de prédictions sur la vitesse effective de notre implémentation. De plus, elles sont généralement axées sur les calculs, pas sur les mouvements de données. La manière la plus directe de mesurer la performance parallèle est de comparer les temps d'exécution du programme séquentiel avec celui parallèle. Le gain de vitesse (speedup) se calcule comme le temps séquentiel divisé par le temps parallèle.

La multiplication de matrices en OpenMP est simple :

#pragma omp parallel for

for (int i=0; i < N; i++)

for (int j=0; j < N; j++)

for (int k=0; k < N; k++)

c[i\*N+j] += a[i\*N+k] \* b[k\*N+j];

La granularité des tâches est choisie automatiquement. Cependant, il y a seulement une concurrence de N et N^3 opérations. Est-il possible d'améliorer cela d'une manière ou d'une autre ?

[EXERCICE]

Dans le modèle de mémoire partagée, les tâches (processus/fils d'exécution) partagent un espace d'adressage commun, auquel ils accèdent de manière asynchrone en lecture/écriture. Des mécanismes sont mis en place pour contrôler l'accès et prévenir les conditions de course (verrous, sémaphores). Il n'y a pas de "propriété" des données - l'accès est égal et donc assez simple. Cependant, la localisation des données est difficile à gérer. Le maintien des données locales à l'endroit où nous travaillons sur elles accélère les accès, mais il est difficile à contrôler.

La plupart des modèles de programmation parallèle fonctionnent dans un environnement de mémoire partagée. Tous les modèles SIMD sont naturellement en mémoire partagée. La question est de savoir comment contrôler le parallélisme, la synchronisation et l'accès aux données partagées et privées. Nous avons tendance à appeler les flux concurrents d'instructions capables d'accéder à la même mémoire partagée des "threads". Les threads sont générés (créés) par un processus maître, ils ont leur propre état (par exemple, où ils en sont dans le flux d'instructions) et certaines données privées. Différents modèles de programmation offrent différentes façons de contrôler les threads.

Du point de vue de l'implémentation des threads, nous avons besoin d'un ensemble d'appels d'API à partir du code source, ainsi que de directives du compilateur dans le code pour indiquer l'exécution parallèle des tâches. OpenMP est un exemple de cela. C'est une norme de l'industrie définie conjointement par un groupe d'organisations matérielles/logicielles. Basé sur des directives de compilateur, portable (OS et langages, C/Fortran) et facile à utiliser - peut être ajouté progressivement.

Open Multi-Processing (OMP) comprend trois composants : des directives de compilateur, des routines de bibliothèque d'exécution et des variables d'environnement. Il offre un modèle de haut niveau avec une allocation implicite et un équilibrage de la charge de travail, est standard, portable et facile à utiliser. Pour plus d'informations, vous pouvez consulter le site : <https://computing.llnl.gov/tutorials/openMP/>

Une image contenant texte, capture d’écran, ligne, Rectangle

Description générée automatiquement Une image contenant texte, capture d’écran, Police, ligne

Description générée automatiquement

Modèle de Fork-Join

Tous les programmes commencent sous la forme d'un seul processus : le thread maître. Il s'exécute séquentiellement jusqu'à la première construction de région parallèle. Dans la phase de FORK, le thread maître crée une équipe de threads parallèles. Les instructions à l'intérieur de la région parallèle sont exécutées en parallèle entre les différents threads. Une fois la région parallèle terminée, les threads se synchronisent et se terminent, laissant le thread maître.

Une image contenant texte, diagramme, ligne, Police

Description générée automatiquement

Dans le code ci-dessous, les directives de compilation sont utilisées pour délimiter les régions parallèles. Chaque région est précédée par la directive #pragma omp parallel for. Après chaque région, un message est imprimé pour indiquer l'étape correspondante.

#pragma omp parallel for

for (int i=0; i < N; i++)

c[i] = a[i] + b[i];

std::cout << "step 1\n";

#pragma omp parallel for

for (int i=0; i < N; i++)

d[i] = c[i] + b[i];

std::cout << "step 2\n";

#pragma omp parallel for

for (int i=0; i < N; i++)

e[i] = d[i] + c[i];

std::cout << "step 3\n";

Les directives de compilateur apparaissent sous forme de commentaires dans le code et sont ignorées par le compilateur à moins d'être explicitement activées (par un indicateur de compilation). Les directives OpenMP sont utilisées à diverses fins, telles que l'amorce d'une région parallèle, la division des blocs de code entre les threads, la distribution des itérations de boucle entre les threads, la sérialisation de sections de code et la synchronisation du travail entre les threads.

L'API OpenMP comprend un nombre croissant de routines de bibliothèque d’exécution utilisées pour diverses opérations telles que la configuration et la récupération du nombre de threads, l'interrogation de l'identifiant unique des threads, la configuration et l'interrogation des threads dynamiques, etc. Ces routines sont accessibles via l'en-tête

#include <omp.h>

int omp\_get\_num\_threads(void)

OpenMP propose plusieurs variables d'environnement pour contrôler l'exécution du code parallèle, telles que le nombre de threads, la spécification de la répartition des itérations de boucle, la liaison des threads aux processeurs physiques, le contrôle du parallélisme imbriqué, etc. Ces variables peuvent être définies dans l'environnement à l'aide de la commande

export OMP\_NUM\_THREADS=4

Les directives OpenMP suivent des règles strictes : elles sont sensibles à la casse, chaque directive ne peut contenir qu'un seul nom de directive et doit être suivie d'un bloc structuré. Les lignes de directives longues peuvent être continuées sur des lignes suivantes avec le caractère "\".

#pragma omp parallel for \

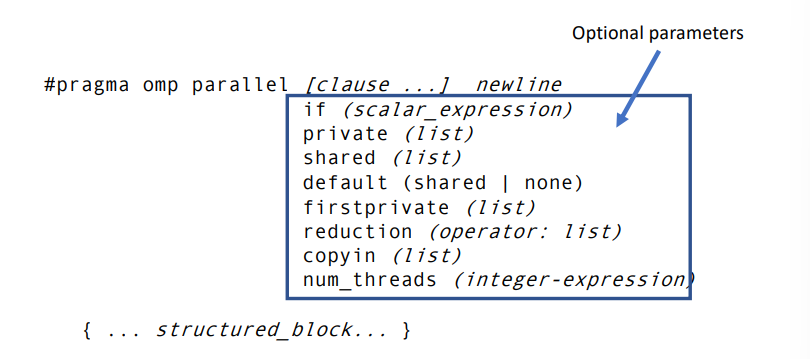
shared(a,b,c,chunk) private(i) \

schedule(static,chunk)

for (i=0; i < n; i++)

c[i] = a[i] + b[i];

Une région parallèle est un bloc de code qui sera exécuté par plusieurs threads. Il s'agit du concept fondamental de parallélisme OpenMP. La construction d'une région parallèle est introduite par la directive `#pragma omp parallel` suivie d'éventuelles clauses qui spécifient des aspects tels que la spécification du nombre de threads ou la portée des variables.



Lorsqu'un thread atteint une directive parallèle, il crée une équipe de threads et devient le maître de l'équipe. Le maître est un membre de l'équipe et a l'ID 0. À partir du début de la région, le code est "dupliqué" et tous les threads exécuteront ce code. Les variables définies à l'intérieur d'une région parallèle seront également répliquées. Il y a une barrière implicite (point de synchronisation) à la fin de la section parallèle, et seul le maître continue l'exécution au-delà de celle-ci. Si un thread se termine à l'intérieur d'une région parallèle, tous les threads de l'équipe se termineront et le travail effectué sera indéfini.

Exemple de code :

#include <omp.h>

int main() {

int var1, var2, var3;

// Code sériel

...

// Début de la section parallèle. Créer une équipe de threads. Spécifier le cloisonnement des variables

#pragma omp parallel private(var1, var2) shared(var3)

{

// Section parallèle exécutée par tous les threads

// Autres directives OpenMP

// Appels à la bibliothèque d'exécution

// Tous les threads rejoignent le thread maître et se disloquent

}

// Reprise du code sériel

...

}

Le cloisonnement (Scoping en anglais) est une façon de contrôler la portée des variables dans une région parallèle. Dans l'exemple précédent, `var1` et `var2` sont privées (chaque thread possède sa propre copie) tandis que `var3` est partagée (tous les threads accèdent à la même copie).

Les directives OpenMP ont une portée statique, qui est définie par le bloc de code structuré dans lequel elles apparaissent. Une directive orpheline est une directive OpenMP qui apparaît en dehors de la portée d'une autre directive. L'étendue dynamique inclut à la fois l'étendue statique et celle des directives orphelines.

Qu’est-ce qu’une directive orpheline concrètement ?

Une directive orpheline dans OpenMP est une directive qui apparaît en dehors de la portée d'une autre directive. En d'autres termes, elle n'est pas associée à une structure de contrôle telle qu'une boucle ou une région parallèle. Ces directives sont indépendantes et ne sont pas conditionnées par la portée d'autres directives.

#pragma omp parallel

{

// Code exécuté par tous les threads

}

#pragma omp barrier // Directive orpheline

Dans cet exemple, la directive `omp barrier` est une directive orpheline car elle n'est pas liée à la région parallèle précédente. Elle crée simplement une barrière de synchronisation entre tous les threads, mais elle n'est pas spécifiquement attachée à une région parallèle donnée.

Un autre exemple pourrait être :

#pragma omp parallel for

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

// Code exécuté par les threads de la boucle parallèle

}

#pragma omp critical // Directive orpheline

{

// Code exécuté de manière critique

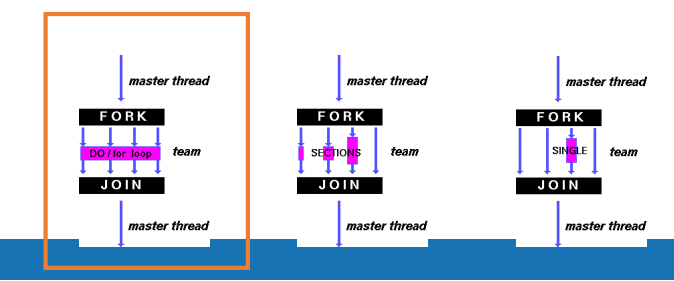
}

Ici, la directive `omp critical` est une directive orpheline car elle n'est pas incluse dans la boucle parallèle précédente. Elle crée une section critique où le code à l'intérieur ne peut être exécuté que par un thread à la fois, mais elle n'est pas directement liée à la boucle parallèle.

Le nombre de threads dépend de plusieurs facteurs, notamment la clause `if()` (doit être vraie, sinon sériel), la clause `num\_threads()`, l'utilisation de l'API `set\_omp\_num\_threads()`, la variable d'environnement `OMP\_NUM\_THREADS` et la valeur par défaut de l'implémentation (nombre de processeurs). Les threads sont numérotés de 0 (le maître) à N-1.

[EXERICE]

Les constructions de partage de travail répartissent l'exécution de la région de code englobante parmi les membres de l'équipe qui la rencontrent. Elles ne lancent pas de nouveaux threads et ne comportent pas de barrière implicite à l'entrée, mais il y en a une à la sortie. Elles doivent se produire à l'intérieur d'une région parallèle, rencontrée par tous les membres, dans le même ordre.



|  |  |  |
| --- | --- | --- |
| #pragma omp parallel  {  #pragma omp for  for (int i = 0; i < N; i++) {  // Chaque thread traite une itération différente de la boucle  // Les itérations sont réparties automatiquement entre les threads  // Par exemple, chaque thread peut effectuer des calculs sur un élément d'un tableau  process\_data(i);  }  } | #pragma omp parallel  {  #pragma omp sections  {  #pragma omp section  {  // Bloc de code exécuté par le premier thread  process\_section1();  }  #pragma omp section  {  // Bloc de code exécuté par le deuxième thread  process\_section2();  }  }  } | #pragma omp parallel  {  #pragma omp single  {  // Bloc de code exécuté par un seul thread  // Par exemple, initialisation de données ou impression de résultats  initialize\_data();  }  // Autres threads continuent ici  // Chaque thread exécute la suite du code de manière parallèle  process\_remaining\_data();  } |

La directive for spécifie que les itérations de la boucle immédiatement suivante doivent être exécutées en parallèle.

#pragma omp for [clause ...] newline

schedule (type [,chunk])

ordered

private (list)

firstprivate (list)

lastprivate (list)

shared (list)

reduction (operator: list)

collapse (n)

nowait

for\_loop

Clause Schedule : Cette clause spécifie comment les itérations de la boucle seront divisées entre les threads. Les différentes options sont :

* Static : Les itérations sont divisées en morceaux de taille fixe (chunk) et chaque morceau est attribué statiquement à un thread. Par défaut, la taille de chunk est size / num\_threads, où size est le nombre total d'itérations de la boucle et num\_threads est le nombre de threads.
* Dynamic : Les itérations sont divisées en morceaux de taille chunk, mais cette fois, la répartition est dynamique. Par défaut, chunk est 1, ce qui signifie qu'une seule itération est attribuée à chaque thread à la fois.
* Guided : Similaire à la planification dynamique, mais la taille du chunk diminue progressivement. Elle commence avec size / num\_threads, puis reste / num\_threads, etc.
* Runtime : La planification est laissée à la runtime, et dépend de la variable d'environnement OMP\_SCHEDULE.
* Auto : Laisse le compilateur ou le programme décider de la meilleure façon de planifier les itérations.

Clause Nowait : Cette clause indique qu'il n'y a pas de synchronisation à la fin de la boucle. Les threads peuvent continuer à exécuter le code suivant sans attendre que tous les threads aient terminé la boucle.

Clause Ordered : Cette clause garantit que certaines parties des itérations sont exécutées dans l'ordre, comme dans un programme sériel. Elle est utilisée avec la directive parallel for pour garantir que les itérations sont exécutées dans l'ordre défini par la boucle.

Clause Collapse : Cette clause spécifie le nombre de boucles imbriquées à fusionner en un seul espace d'itération, qui sera ensuite divisé entre les threads. Cela permet de paralléliser plus efficacement les boucles imbriquées en réduisant les frais généraux de parallélisation.

Besoin de synchronisation

|  |  |
| --- | --- |
| THREAD 1 : increment(x){  x = x + 1; } | THREAD 2 : increment(x){  x = x + 1; } |
| 10 CHARGEMENT A, (adresse x)  20 AJOUTER A, 1  30 STOCKER A, (adresse x) | 10 CHARGEMENT B, (adresse x)  20 AJOUTER B, 1  30 STOCKER B, (adresse x) |

Une séquence possible :

* Le thread 1 charge x dans le registre A ->
* Le thread 2 charge x dans le registre B ->
* Le thread 1 ajoute 1 à A ->
* Le thread 2 ajoute 1 à B ->
* Le thread 1 écrit A dans x ->
* Le thread 2 écrit B dans x

La directive maître et la directive critique sont des outils essentiels dans la programmation parallèle avec OpenMP pour contrôler l'exécution de sections de code critiques et sensibles aux threads. La directive maître spécifie une région qui ne doit être exécutée que par le thread principal de l'équipe. Contrairement à d'autres directives, elle ne crée pas de barrière implicite, ce qui signifie que l'exécution se poursuit sans attendre la fin de la région.

#pragma omp master newline

structured\_block

D'autre part, la directive critique délimite une région de code qui doit être exécutée par un seul thread à la fois. Si un thread se trouve dans cette section critique et qu'un autre thread arrive, celui-ci sera bloqué jusqu'à ce que le premier thread ait terminé son exécution. Il est également possible de nommer cette section critique, permettant ainsi de regrouper plusieurs sections dans le code en une seule. Les sections non nommées sont traitées comme une seule et même section critique.

Un exemple concret illustrant l'utilisation de la section critique serait une variable entière partagée entre plusieurs threads nécessitant une incrémentation sécurisée. Dans ce cas, la directive critique garantit que l'incrémentation se fait de manière atomique, évitant ainsi les problèmes de concurrence.

#pragma omp critical [ name ] newline

structured\_block

Exemple:

#include <omp.h>

main()

{

int x;

x = 0;

#pragma omp parallel shared(x)

{

#pragma omp critical

x = x + 1;

} /\* fin de la section parallèle \*/

printf("%d\n", x);

}

Cependant, bien que la section critique soit une solution efficace pour éviter les conflits d'accès aux données partagées, elle peut également entraîner des goulots d'étranglement en raison du blocage des threads concurrents. Dans certains cas, il peut être envisagé d'autres mécanismes de synchronisation ou de restructuration du code pour éviter ce problème.

typedef struct {

int index;

int value;

} Event;

...

#pragma omp parallel for

for(int i = 0; i < DATA\_SIZE; i++) {

if(data[i] > THRESHOLD) {

#pragma omp critical

{

rareEvents[eventCount].index = i;

rareEvents[eventCount].value = data[i];

eventCount++;

}

}

}

La directive barrière et la directive atomique sont des mécanismes essentiels pour synchroniser l'exécution des threads dans un programme parallèle avec OpenMP. Celle-ci synchronise tous les threads de l'équipe. Lorsque la directive barrière est atteinte, un thread attend que tous les autres threads aient atteint cette barrière. Ensuite, tous les threads reprennent l'exécution du code après la barrière. Il est important de noter que tous les threads (ou aucun) doivent rencontrer la barrière pour que la synchronisation soit effective.

#pragma omp parallel shared(image)

{

// Étape 1 : Appliquer un filtre à la section assignée

applyFilter(image\_section);

// Barrière : Attendre que tous les threads aient terminé l'étape 1

#pragma omp barrier

// Étape 2 : Appliquer le deuxième traitement

applyEnhancement(image\_section);

}

La directive atomique spécifie qu'un emplacement mémoire spécifique doit être mis à jour de manière atomique, plutôt que de permettre à plusieurs threads de tenter d'écrire dessus. C'est comme une mini section critique, s'appliquant à une seule (suivante) instruction.

#pragma omp parallel for

for(int i = 0; i < DATA\_SIZE; i++) {

int bin = data[i] \* NUM\_BINS / MAX\_VALUE; // Calcul simple du bin

#pragma omp atomic

histogram[bin]++;

}

Dans cet exemple, la directive atomique est utilisée pour garantir que l'incrémentation de l'histogramme se fait de manière atomique, évitant ainsi les conflits de données lors de l'accès concurrentiel à la même case mémoire.

*Que veut dire : « de manière atomique » ? :*

*Dans le contexte de la programmation parallèle, l'opération atomique signifie qu'une opération sur une variable est effectuée de manière indivisible, c'est-à-dire que l'opération entière se déroule sans interruption de la part d'autres threads. En d'autres termes, aucune autre opération ne peut interférer ou se produire simultanément avec cette opération atomique.*

*Lorsque vous déclarez une opération comme atomique dans un environnement parallèle, vous garantissez que cette opération est exécutée sans risque de corruption de données ou de conflits de ressources partagées entre les threads.*

*Dans l'exemple donné précédemment, l'opération atomique est utilisée pour garantir que l'incrémentation de l'histogramme se fait de manière sûre, même lorsque plusieurs threads tentent d'accéder à la même case mémoire simultanément. Cela évite les situations où deux threads pourraient essayer d'incrémenter la même case mémoire en même temps, ce qui entraînerait un comportement indéterminé ou des résultats incorrects.*

Partage de données

Il est important de comprendre comment les variables sont partagées et comment utiliser la portée des données dans un environnement parallèle. Par défaut, la plupart des variables sont partagées, sauf les variables d'index de boucle et tout ce qui est créé sur la pile à l'intérieur d'une région parallèle (c'est-à-dire les variables locales).

Les attributs de portée des données incluent `Private`, `Firstprivate`, `Lastprivate`, `Shared`, `Default`, `Reduction` et `Copyin`. Ces attributs sont combinés avec d'autres directives pour contrôler la portée des variables incluses.

La directive `Private` liste les variables qui doivent être privées pour chaque thread. Chaque thread possède une nouvelle copie de la variable, et toutes les références à l'original sont remplacées par des références au nouvel objet. Les variables privées sont généralement considérées comme non initialisées, sauf lorsqu'elles sont combinées avec `Firstprivate` pour les initialiser.

private (list)

firstprivate (list)

lastprivate (list)

La directive `Shared` déclare les variables énumérées comme partagées entre tous les threads de l'équipe. Ces variables existent dans une seule adresse mémoire et tous les threads peuvent y lire ou écrire. Il incombe au programmeur de garantir un accès sûr à ces variables partagées.

shared (list)

La directive `Reduction` effectue une réduction sur les variables de la liste. Une copie privée est créée pour chaque thread, et à la fin, elles sont réduites à la variable partagée globalement.

reduction (operator: list)

Exemple d'utilisation de la directive de réduction

#pragma omp parallel for \

default(shared) private(i) \

reduction(+:result)

for (i=0; i < n; i++)

result = result + (a[i] \* b[i]);

printf("Final result= %f\n",result);

* Les variables doivent être scalaires (pas de tableaux ou de structures).
* Elles doivent être déclarées partagées dans toute région englobante.
* La réduction suppose l'associativité des opérateurs utilisés.

Les opérateurs compatibles sont `+`, `\*`, `-`, `/`, `&`, `^`, `|`, `&&`, `||`, `++`, `--`.

*Pour plus de détail :*

*La directive `reduction` est utilisée dans OpenMP pour effectuer une opération de réduction sur une variable partagée. L'idée est d'effectuer une opération associative (comme l'addition ou la multiplication) sur les valeurs individuelles calculées par chaque thread, puis de combiner ces valeurs réduites en une seule valeur globale.*

*Par exemple, supposons que vous ayez une boucle parallèle où chaque thread calcule une valeur et que vous souhaitiez sommer ces valeurs pour obtenir une valeur globale. Sans la réduction, chaque thread mettrait à jour la variable partagée, ce qui entraînerait des problèmes de concurrence. La directive `reduction` résout ce problème en créant une copie privée de la variable pour chaque thread, en effectuant l'opération sur cette copie privée, puis en combinant toutes les valeurs réduites pour obtenir une seule valeur finale.*

*Voici un exemple de la directive de réduction pour sommer les valeurs calculées par chaque thread :*

*#pragma omp parallel for reduction(+:sum)*

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

*sum += array[i];*

*}*

*Dans cet exemple, `sum` est une variable partagée qui contient la somme des valeurs. Chaque thread a sa propre copie de `sum`, et à la fin de la boucle parallèle, toutes les valeurs de `sum` sont additionnées pour obtenir la somme finale. L'opération de réduction est faite ici avec l'opérateur `+`, indiqué par `+:sum`.*

Parallel Programming C3

Les dépendances de données sont un aspect crucial à considérer lors de la parallélisation de code. Dans la plupart des cas, la parallélisation consiste à diviser de grandes boucles for en morceaux indépendants que de nombreux processeurs peuvent traiter simultanément.

Dans un scénario sans dépendances de données, chaque processeur peut travailler sur une partie distincte de l'itération sans avoir besoin des résultats des autres processeurs. Par exemple, dans une addition de vecteurs, chaque processeur peut calculer une tranche spécifique des données sans nécessiter d'informations provenant des autres processeurs.

// Exemple sans dépendance de données : addition de vecteurs

#pragma omp parallel for

for (int i = 0; i < 100000; i++) {

c[i] = a[i] + b[i];

}

Cependant, dans certains cas, les itérations de la boucle ne sont pas indépendantes les unes des autres. Par exemple, dans une opération de scan inclusif, chaque élément de sortie dépend du résultat de l'élément précédent. Cette dépendance entre les itérations est appelée "dépendance de boucle portée" ou "dépendance de données".

// Exemple avec dépendance de données : scan inclusif

#pragma omp parallel for

for (int i = 1; i < 100000; i++) {

c[i] = c[i] + c[i-1];

}

Si le compilateur détecte une telle dépendance, il peut refuser de paralléliser la boucle pour garantir la validité des résultats.

Pour contourner cette limitation, il est possible de réorganiser le code pour éliminer les dépendances réelles ou potentielles, ou d'utiliser des directives spécifiques comme #pragma omp for pour outrepasser la décision du compilateur. Cependant, cela peut potentiellement conduire à des résultats incorrects si les dépendances ne sont pas correctement gérées.

Comprendre les dépendances de données est essentiel pour tirer parti de la parallélisation automatique offerte par les compilateurs, en particulier dans le contexte des architectures SIMD (Single-Instruction-Multiple-Data). Une bonne gestion des dépendances permet d'optimiser les performances tout en assurant la cohérence des résultats.

La structure de contrôle des processeurs parallèles peut varier considérablement en fonction du niveau de parallélisme impliqué, allant des instructions individuelles aux processus entiers. Voici quelques alternatives de structure de contrôle des processeurs :

* Travail indépendant : Chaque processeur fonctionne de manière autonome, exécutant ses propres tâches sans coordination avec les autres.
* Contrôle centralisé : Tous les processeurs fonctionnent sous le contrôle centralisé d'une seule unité de contrôle. Cela peut être efficace pour coordonner les tâches et garantir la synchronisation, mais cela peut aussi devenir un goulot d'étranglement dans les systèmes très parallèles. !!!!!
* MIMD (Multiple Instruction, Multiple Data) : Chaque processeur dispose de son propre jeu d'instructions et de données. Ils peuvent donc exécuter des instructions différentes sur des données différentes en même temps.
* SIMD (Single Instruction, Multiple Data) : Dans cette configuration, une seule instruction est envoyée à tous les processeurs, mais chacun travaille sur ses propres données. Cela est souvent utilisé pour des opérations où la même opération doit être effectuée sur un grand ensemble de données simultanément, comme le traitement multimédia ou l'algèbre linéaire.

Une image contenant texte, capture d’écran, diagramme, Police

Description générée automatiquement

Les unités SIMD sont particulièrement efficaces pour les calculs ayant une structure régulière, tels que le traitement multimédia et l'algèbre linéaire. Elles utilisent souvent des masques d'activité pour exécuter sélectivement des opérations sur certains processeurs.

Exemple : Vecteurs SIMD de 128 bits

Une image contenant capture d’écran, ligne, conception, rouge

Description générée automatiquement

Dans ce cas, les données sont organisées en vecteurs de 128 bits, où chaque instruction opère simultanément sur plusieurs éléments de données dans ce vecteur. Les données doivent être contiguës en mémoire et alignées pour assurer des performances optimales. Des instructions supplémentaires sont souvent nécessaires pour manipuler ces vecteurs, telles que des instructions de masquage pour sélectionner certaines données et des instructions de déplacement pour transférer des données entre différentes parties du vecteur.

Calcul avec des unités SIMD

* Traitement scalaire : Une opération produit un résultat.
* Unités vectorielles SIMD : Une opération produit plusieurs résultats simultanément.

Les opérations SIMD (Single Instruction, Multiple Data) sont conçues pour exécuter une même opération simultanément sur un ensemble de valeurs de données, souvent organisées en vecteurs. Voici quelques aspects clés des opérations SIMD :

* L’exécution SIMD permet d'effectuer une opération en parallèle sur un tableau de 2, 4, 8, 16 ou 32 valeurs, en fonction de la taille des valeurs et de l'architecture du processeur. Cela s'appelle également une opération parallèle de données.

Une image contenant texte, ligne, capture d’écran, Police

Description générée automatiquementUne image contenant texte, capture d’écran, ligne, diagramme

Description générée automatiquement

* + Les opérations SIMD peuvent inclure : Mouvement de données, Instructions arithmétiques, Instructions logiques, Instructions de comparaison, Instructions de conversion, Instructions de mélange (shuffle).
* Les opérations empaquetées s'appliquent en parallèle à 2, 4 ou 8 valeurs en virgule flottante, selon le format de données.
* Les opérations scalaires appliquent une opération sur une seule valeur en virgule flottante.

Le coût en termes de performances est similaire pour les deux types d'opérations.

L'exécution conditionnelle sur un processeur SIMD, ou unité SIMD, se réfère à la possibilité d'exécuter différentes branches de code en fonction d'une condition sur l'ensemble des données traitées simultanément.

Une image contenant texte, diagramme, Plan, capture d’écran

Description générée automatiquement

* Déclaration conditionnelle : Cela fait référence à une instruction conditionnelle qui évalue une condition donnée pour déterminer le chemin d'exécution du programme.
* Valeurs initiales : Avant d'exécuter la branche "alors" ou "sinon" d'une instruction conditionnelle, des valeurs initiales doivent souvent être définies pour les données sur lesquelles l'instruction conditionnelle sera appliquée.
* Exécution de la branche "alors" (then) : Si la condition donnée est évaluée comme vraie pour une partie des données traitées simultanément, alors la branche "alors" du code est exécutée pour ces données. Cela signifie que l'opération spécifiée dans cette branche sera effectuée sur ces données.

Exécution de la branche "sinon" (else) : Si la condition donnée est évaluée comme fausse pour une partie des données traitées simultanément, alors la branche "sinon" du code est exécutée pour ces données. Cela signifie que l'opération spécifiée dans cette branche sera effectuée sur ces données.

En résumé, l'exécution conditionnelle sur un processeur SIMD permet de gérer des instructions différentes en fonction de la condition évaluée pour chaque élément de données dans un vecteur SIMD, ce qui permet une plus grande flexibilité dans le traitement des données parallèles.

**Exemples SIMD**

* Historiquement : Ordinateurs SIMD
  + Machine de Connexion CM-1/2 : 65536 processeurs 1-bit
* Aujourd'hui : Unités ou accélérateurs SIMD
  + Unités vectorielles
    - AVX/2/512 – Extensions vectorielles avancées
    - Registres de 256 ou 512 bits (Sandy Bridge/Skylake)
  + Co-processeurs
    - GPUs NVIDIA
    - • Intel Xeon Phi

Intel Xeon Phi est un processeur spécialisé conçu pour des charges de travail intensives en calcul, telles que le calcul parallèle et le traitement de données massives. Voici quelques caractéristiques de base du Xeon Phi :

**64 cœurs** : Le Xeon Phi est doté de 64 cœurs de traitement, ce qui lui permet d'exécuter un grand nombre de tâches simultanément.

* Cache L1 de 64 Ko : Chaque cœur dispose d'une mémoire cache L1 de 64 Ko **pour un accès rapide aux données.**
* Cache L2 de 512 Ko : En plus de la mémoire cache L1, il y a une mémoire cache L2 de 512 Ko **partagée entre les cœurs**.
* 32 registres vectoriels de 512 bits : Ces registres permettent d'effectuer des opérations vectorielles sur de grandes quantités de données de manière efficace.
* 4 voies SMT (Simultaneous Multithreading) par cœur : Le SMT permet à chaque cœur d'exécuter plusieurs threads simultanément, améliorant ainsi l'utilisation des ressources du processeur.

Fréquence pouvant atteindre 1,5 GHz : Le Xeon Phi peut fonctionner jusqu'à une fréquence de 1,5 GHz, ce qui lui permet d'effectuer des calculs à une vitesse élevée.

16 Go de mémoire empilée : Le processeur est équipé de 16 Go de mémoire empilée, offrant une grande capacité de stockage de données.

* Débit mémoire maximal de 450 Go/s : Le Xeon Phi peut transférer des données à une vitesse maximale de 450 Go par seconde, ce qui est crucial pour des applications gourmandes en données.

3 TFlops : Le Xeon Phi est capable d'effectuer jusqu'à 3 téraflops (3 billions d'opérations en virgule flottante par seconde) de calcul, ce qui en fait une puissante solution de calcul parallèle.

Le **GPU NVIDIA A100** est une unité de traitement graphique (GPU) de pointe conçue pour des charges de travail intensives en calcul, telles que l'apprentissage machine, l'intelligence artificielle et les simulations scientifiques. Voici quelques-unes de ses spécifications principales :

* 108 Streaming Multiprocessors (SMX) : Le GPU est composé de 108 SMX, qui sont des unités de traitement fondamentales responsables de l'exécution des calculs parallèles.

Chaque SMX :

* 64 cœurs CUDA : Les cœurs CUDA sont les unités de calcul de base du GPU, responsables de l'exécution des opérations de calcul.
* Unité s FP et INT entièrement pipelinées : Ces unités sont spécialisées dans l'exécution de calculs en virgule flottante (FP) et entiers (INT).
* Quatre planificateurs de warp : Les planificateurs de warp gèrent l'exécution des groupes de threads (warps) sur le SMX.
* 32 groupes de threads (warps) : Les warps sont des groupes de threads qui sont exécutés simultanément sur le SMX.
* 4 warps émettent et exécutent simultanément : Cela signifie que quatre groupes de threads peuvent être exécutés en parallèle sur chaque SMX.
* 2 instructions/warp/cycle : Le SMX est capable de traiter jusqu'à deux instructions par warp à chaque cycle d'horloge.
* 64 unités de double précision : Ces unités sont spécialisées dans l'exécution de calculs en double précision, ce qui est crucial pour certaines applications scientifiques.
* 32 unités de chargement-stockage : Ces unités gèrent les opérations de lecture et d'écriture dans la mémoire du GPU.
* 32 unités de fonctions spéciales : Ces unités sont utilisées pour exécuter des fonctions mathématiques spéciales, telles que les fonctions trigonométriques et exponentielles.

Performances de calcul : 9,7/19,5 TFLOP : Le GPU NVIDIA A100 est capable d'effectuer jusqu'à 9,7 téraflops (calculs en virgule flottante à simple précision) ou 19,5 téraflops (calculs en virgule flottante à demi-précision) par seconde, ce qui en fait une puissante solution de calcul parallèle.

**La programmation pour les unités vectorielles** implique l'utilisation d'instructions vectorielles dans votre programme pour tirer parti de la capacité des processeurs à effectuer des opérations sur plusieurs éléments de données en parallèle. Voici trois façons différentes d'utiliser des instructions vectorielles dans votre programme :

* Vectorisation automatique par le compilateur :

Cela signifie que le compilateur est responsable de la vectorisation du code.

Aucune programmation vectorielle explicite n'est requise de la part du développeur.

Cependant, le code doit être structuré de manière à ce que le compilateur puisse identifier les possibilités de vectorisation.

* Exprimer le calcul sous forme d'expressions arithmétiques sur des types de données vectorielles :

Dans cette approche, vous déclarez des variables de type vectoriel.

Les calculs sont ensuite exprimés comme des expressions arithmétiques normales, mais appliquées à des variables de type vecteur.

* Utiliser des fonctions intrinsèques du compilateur pour les opérations vectorielles :

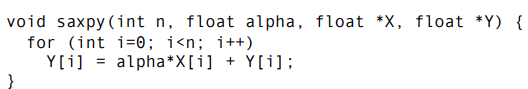
Il s'agit de fonctions fournies par le compilateur qui implémentent des instructions vectorielles dans un langage de haut niveau.

Cela nécessite une connaissance détaillée de l'ensemble d'instructions vectorielles spécifiques au processeur utilisé.

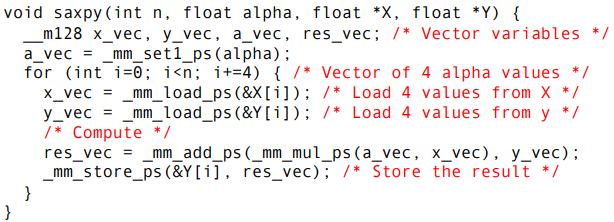
Les développeurs doivent utiliser ces fonctions pour exploiter pleinement les capacités vectorielles du processeur.

**La programmation avec des intrinsèques** est une méthode permettant d'exploiter directement les instructions vectorielles du processeur dans votre code source, sans avoir à écrire du code assembleur bas niveau. Voici une comparaison entre une implémentation classique d'une fonction SAXPY (Single-precision A\*X Plus Y) et une implémentation utilisant des intrinsèques SIMD (Single Instruction, Multiple Data) SSE (Streaming SIMD Extensions) pour la parallélisation :

Implémentation classique :



Implémentation avec intrinsèques SIMD :



Dans cette version utilisant des intrinsèques SIMD :

- Nous utilisons des types de données spécifiques (\_\_m128) pour représenter des vecteurs de 4 valeurs flottantes à simple précision.

- Nous utilisons des fonctions intrinsèques SSE comme `\_mm\_set1\_ps`, `\_mm\_load\_ps`, `\_mm\_mul\_ps`, `\_mm\_add\_ps`, et `\_mm\_store\_ps` pour effectuer des opérations vectorielles.

- La boucle for parcourt les tableaux X et Y par pas de 4, pour traiter quatre éléments à la fois.

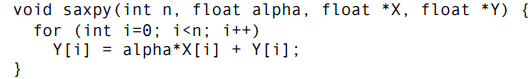
- À chaque itération, nous chargeons quatre valeurs de X et Y, effectuons le calcul vectoriel (SAXPY), et stockons le résultat dans Y.

Cela permet d'exploiter les capacités vectorielles du processeur pour accélérer les calculs, en réalisant plusieurs opérations en parallèle sur des jeux de données contigus.

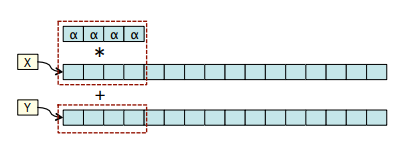
**La vectorisation automatique** est une technique où le compilateur reconnaît les opportunités pour paralléliser des opérations sur des données en les traitant comme des vecteurs. Voici quelques points à considérer concernant la vectorisation automatique :

* Requiert un compilateur avec des capacités de vectorisation : Tous les compilateurs ne sont pas équipés pour effectuer cette optimisation. Des compilateurs tels que GCC (g++) avec des options d'optimisation élevées comme -O3 ou -Ofast sont généralement nécessaires. Pour le compilateur Intel, la vectorisation est souvent de meilleure qualité.
* Reconnaissance automatique par le compilateur : Le compilateur est capable d'identifier automatiquement les boucles dans le code source qui peuvent être vectorisées.
* Complexité de la vectorisation automatique : Il s'agit d'une tâche complexe pour le compilateur car il doit garantir que le code généré est correct. Cela implique souvent des analyses approfondies du code pour détecter les dépendances de données et autres obstacles à la parallélisation.
* Aider le compilateur : Bien que la vectorisation automatique soit puissante, il peut être bénéfique d'aider le compilateur en lui fournissant des indications. Par exemple :
* Utilisation de pointeurs avec le qualificatif \_\_restrict pour indiquer qu'il n'y a pas d'aliasing entre pointeurs, ce qui simplifie l'analyse des dépendances de données.
* Alignement des tableaux sur des frontières de 32 octets pour faciliter le chargement des données dans les registres vectoriels, ce qui améliore les performances.

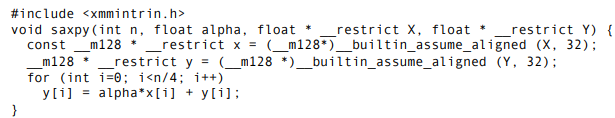
Dans cet exemple, nous avons une fonction SAXPY (Single-precision Alpha X Plus Y) qui effectue une opération vectorielle entre un scalaire (alpha) et deux vecteurs (X et Y) de nombres flottants. Voici le code non vectorisé



Pour exploiter la vectorisation, nous voulons effectuer les calculs sur 4 valeurs à la fois, ce qui peut améliorer considérablement les performances sur les architectures modernes.



Dans cet exemple, nous utilisons **des types de données vectorielles explicites** pour exploiter les instructions SIMD (Single Instruction, Multiple Data) et effectuer des calculs sur plusieurs éléments de données simultanément. Voici le code :



Dans ce code :

* Nous utilisons le type de données vectorielles **\_\_m128** pour représenter des vecteurs de 4 valeurs flottantes à simple précision (correspondant à un registre SIMD de 128 bits).
* Les fonctions intrinsèques SIMD **\_mm\_add\_ps** et **\_mm\_mul\_ps** sont utilisées pour effectuer respectivement l'addition et la multiplication vectorielles.
* **\_\_builtin\_assume\_aligned** est une fonction du compilateur (dans ce cas, GCC) qui permet d'indiquer au compilateur que les pointeurs **X** et **Y** sont alignés sur une frontière de 32 octets, ce qui est important pour les performances des opérations SIMD.
* La boucle parcourt les données en incréments de 4 (n / 4), ce qui signifie que chaque itération traite 4 éléments de **X** et **Y** en parallèle.
* **alpha** est un scalaire, donc il est multiplié par chaque élément de **x[i]** à l'aide de **\_mm\_set1\_ps(alpha)**, qui crée un vecteur contenant quatre valeurs alpha identiques.

En utilisant des types de données vectorielles explicites et des opérations SIMD, nous optimisons le code pour qu'il effectue les calculs sur plusieurs éléments de données en parallèle, ce qui peut considérablement accélérer les performances par rapport à une approche séquentielle.

Message Passing Interface – Advanced Topics

Dans cette leçon avancée sur l'interface de passage de messages (MPI), nous abordons des concepts essentiels concernant les groupes et les communicateurs, qui sont fondamentaux pour organiser et gérer les processus dans les environnements parallèles.

# Groupes et communicateurs

Au début, nous avons utilisé MPI\_COMM\_WORLD, qui regroupait tous les processus impliqués dans notre programme MPI. Cependant, pour une gestion plus fine, nous introduisons la notion de groupes MPI. Un groupe MPI est un ensemble ordonné de processus, chaque processus étant associé à un rang entier unique. Les valeurs de rang commencent à zéro et vont jusqu'à N-1, où N est la taille du groupe. Chaque groupe est toujours associé à un communicateur MPI, qui englobe un ensemble de processus pouvant communiquer entre eux. Du point de vue du programmeur, il n'y a pas de différence fondamentale entre MPI\_COMM\_WORLD et les autres communicateurs créés pour des groupes spécifiques.

# Objectifs des groupes MPI

Les groupes MPI ont plusieurs objectifs cruciaux :

1. Organisation des tâches : Les groupes permettent d'organiser les tâches en fonction de leur fonction dans le programme.
2. Communications collectives : Ils facilitent les opérations de communication collective sur un sous-ensemble de tâches liées
3. Topologies virtuelles : Ils fournissent une base pour implémenter des topologies virtuelles définies par l'utilisateur, permettant ainsi une gestion plus flexible des communications
4. Communications sécurisées : Les groupes offrent un moyen de communiquer de manière sûre, en limitant les interactions aux processus spécifiés.
5. Objets dynamiques : Les groupes sont des objets dynamiques pouvant être créés ou libérés à tout moment, ce qui permet une gestion adaptative des ressources.
6. Processus dans plusieurs groupes : Il est possible pour un processus de faire partie de plusieurs groupes, chaque groupe lui attribuant un identifiant unique.

# Processus pour travailler avec les groupes MPI

Pour travailler avec les groupes MPI, nous suivons une séquence d'étapes :

|  |  |
| --- | --- |
| 1. Extraction du gestionnaire de groupe : À partir de MPI\_COMM\_WORLD, nous obtenons le gestionnaire de groupe avec MPI\_Comm\_group. 2. Création d'un sous-ensemble de groupe global : Utilisant MPI\_Group\_incl, nous créons un sous-ensemble du groupe global selon nos besoins. 3. Création d'un nouveau communicateur : Avec MPI\_Comm\_create, nous créons un nouveau communicateur pour le nouveau groupe. 4. Détermination du nouveau rang : MPI\_Comm\_rank nous permet de déterminer le rang de chaque processus dans le nouveau groupe. 5. Communications au sein du groupe : Nous effectuons les communications nécessaires entre les processus du groupe. 6. Libération des ressources : Une fois les communications terminées, nous libérons les ressources associées en utilisant MPI\_Comm\_free et MPI\_Group\_free. | Une image contenant cercle, capture d’écran  Description générée automatiquement |

# Topologies virtuelles

Les topologies virtuelles décrivent un mapping ou un ordonnancement des processus MPI dans une forme géométrique. Il existe principalement deux types de topologies : les topologies cartésiennes et les topologies de graphe. Le terme "virtuel" est utilisé car il peut ne pas y avoir de relation directe avec la structure physique de la machine sur laquelle s'exécute le programme. Cependant, certaines implémentations tentent d'optimiser ces topologies, notamment si la communication avec des nœuds "lointains" est plus coûteuse. Les topologies virtuelles sont construites à partir des communicateurs et des groupes MPI, et elles sont particulièrement utiles lorsque le schéma de communication spécifique à l'application correspond à la structure de la topologie MPI.

## Topologie cartésienne

La topologie cartésienne est un type de topologie virtuelle qui peut être créée et manipulée à l'aide de fonctions MPI spécifiques :

|  |  |
| --- | --- |
| MPI\_Cart\_create : Cette fonction crée une nouvelle topologie cartésienne à partir d'un communicateur existant. Elle spécifie le nombre de dimensions du tableau de processus, la taille de chaque dimension, la périodicité des dimensions et si le réordonnancement est autorisé. Elle crée un nouveau communicateur pour représenter l'espace dimensionnel spécifié.  MPI\_Cart\_coords : Cette fonction permet d'obtenir les coordonnées cartésiennes d'un processus dans la topologie cartésienne. Elle prend en entrée le communicateur cartésien, le rang séquentiel du processus et renvoie ses coordonnées dans l'espace cartésien.  MPI\_Cart\_shift : Cette fonction permet de trouver les rangs des voisins d'un processus dans une dimension spécifique de la topologie cartésienne. Elle prend en entrée le communicateur cartésien, la dimension concernée, le décalage (offset) par rapport au processus actuel, et renvoie les rangs des voisins dans les directions positive et négative. | Une image contenant capture d’écran, texte, nombre, carré  Description générée automatiquement |

Ces fonctions offrent un moyen pratique de travailler avec des topologies cartésiennes dans MPI, ce qui permet de modéliser et de manipuler efficacement la communication entre les processus dans des configurations structurées.

## Exercise

* Take a lookatmpi\_cart.cpp
* Modify it to work with any number of processes.
* Use MPI\_Dims\_create
* Evaluate it for 1, 2, 4, 7, 8, 9
* What sort of decomposition do you get?
* Modify it so each process ends a message (its rank) to each of neighbors, receives a message (into in buff) from each of its neighbors, and prints it.

Pour modifier le code `mpi\_cart.cpp` afin qu'il fonctionne avec n'importe quel nombre de processus et implémenter la communication entre voisins, nous devons suivre quelques étapes :

1. Remplacer la définition fixe de la taille `SIZE` par une vérification du nombre de processus.
2. Utiliser `MPI\_Dims\_create` pour calculer les dimensions de la topologie cartésienne en fonction du nombre total de processus.
3. Modifier le code pour envoyer le rang de chaque processus à ses voisins et recevoir les messages de ses voisins.

Voici le code modifié :

#include <mpi.h>

#include <stdio.h>

#include <stdlib.h>

#define UP 0

#define DOWN 1

#define LEFT 2

#define RIGHT 3

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

int numtasks, rank, source, dest, outbuf, inbuf[4], nbrs[4], dims[2], periods[2] = {0, 0}, reorder = 0, coords[2];

MPI\_Request reqs[8];

MPI\_Status stats[8];

MPI\_Comm cartcomm;

MPI\_Init(&argc, &argv);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &numtasks);

**MPI\_Dims\_create(numtasks, 2, dims);**

**if (numtasks == dims[0] \* dims[1])** {

MPI\_Cart\_create(MPI\_COMM\_WORLD, 2, dims, periods, reorder, &cartcomm);

MPI\_Comm\_rank(cartcomm, &rank);

MPI\_Cart\_coords(cartcomm, rank, 2, coords);

MPI\_Cart\_shift(cartcomm, 0, 1, &nbrs[LEFT], &nbrs[RIGHT]);

MPI\_Cart\_shift(cartcomm, 1, 1, &nbrs[UP], &nbrs[DOWN]);

printf("rank= %d coords= %d %d neighbors(u,d,l,r)= %d %d %d %d\n", rank, coords[0], coords[1], nbrs[UP], nbrs[DOWN], nbrs[LEFT], nbrs[RIGHT]);

outbuf = rank;

**// Send my rank to all four neighbors**

**MPI\_Isend(&outbuf, 1, MPI\_INT, nbrs[UP], 0, cartcomm, &reqs[UP]);**

**MPI\_Isend(&outbuf, 1, MPI\_INT, nbrs[DOWN], 0, cartcomm, &reqs[DOWN]);**

**MPI\_Isend(&outbuf, 1, MPI\_INT, nbrs[LEFT], 0, cartcomm, &reqs[LEFT]);**

**MPI\_Isend(&outbuf, 1, MPI\_INT, nbrs[RIGHT], 0, cartcomm, &reqs[RIGHT]);**

**// Receive messages from neighbors**

**MPI\_Irecv(&inbuf[UP], 1, MPI\_INT, nbrs[UP], 0, cartcomm, &reqs[UP]);**

**MPI\_Irecv(&inbuf[DOWN], 1, MPI\_INT, nbrs[DOWN], 0, cartcomm, &reqs[DOWN]);**

**MPI\_Irecv(&inbuf[LEFT], 1, MPI\_INT, nbrs[LEFT], 0, cartcomm, &reqs[LEFT]);**

**MPI\_Irecv(&inbuf[RIGHT], 1, MPI\_INT, nbrs[RIGHT], 0, cartcomm, &reqs[RIGHT]);**

**// Wait for all communication to complete**

**MPI\_Waitall(4, reqs, stats);**

printf("rank= %d inbuf(u,d,l,r)= %d %d %d %d\n", rank, inbuf[UP], inbuf[DOWN], inbuf[LEFT], inbuf[RIGHT]);

} else {

printf("Must specify %d processors. Terminating.\n", dims[0] \* dims[1]);

}

MPI\_Finalize();

return 0;

}

Ce code modifié calculera automatiquement les dimensions de la topologie cartésienne en fonction du nombre total de processus, en utilisant `MPI\_Dims\_create`. Ensuite, chaque processus enverra son rang à ses voisins et recevra les rangs de ses voisins pour les imprimer.

## Grille cartésienne et Équations aux dérivées partielles (PDE)

Dans cette partie du cours, nous abordons l'utilisation des grilles cartésiennes pour la résolution numérique des équations aux dérivées partielles (EDP) (PDE en anglais). Une grille cartésienne est souvent utilisée pour représenter discrètement un domaine spatial où chaque point de la grille correspond à un emplacement dans l'espace physique.

### Mise à jour des éléments de la grille

On suppose que les éléments de la grille sont mis à jour à l'aide d'un schéma de différences finies à 5 points. Les points de la grille situés sur la bordure de la grille (en cyan) contiennent généralement des données constantes, tandis que les points à l'intérieur de la grille changent à chaque étape de calcul.

### Partitionnement de la grille

Pour paralléliser le calcul, on peut diviser la grille en sous-domaines. Cela peut être fait verticalement, horizontalement ou les deux. Dans le cadre de la résolution de problèmes parallèles, une attention particulière est portée à minimiser la communication entre les sous-domaines.

Une image contenant carré, Rectangle, motif, ligne

Description générée automatiquementUne image contenant carré, Caractère coloré, Rectangle, ligne

Description générée automatiquement

### Calcul

Pour mettre à jour un point de la grille, des informations sont nécessaires à partir de ses quatre voisins les plus proches, comme illustré par le stencil de différences finies. Toutes les données nécessaires pour la mise à jour d'un point se trouvent dans le sous-domaine local. Cependant, lorsqu'on se déplace au-delà des voisins directs, il peut être nécessaire de récupérer des données des sous-domaines adjacents.

Une image contenant carré, Rectangle, Caractère coloré

Description générée automatiquementUne image contenant carré, Rectangle, Caractère coloré

Description générée automatiquementUne image contenant carré, Rectangle

Description générée automatiquementUne image contenant carré, Rectangle, Caractère coloré

Description générée automatiquement

### Communication

Pour minimiser la communication entre les processus, on crée des cellules "fantômes" (ghost cells) supplémentaires pour contenir les copies des données des voisins. Avant chaque mise à jour, les données de la bordure intérieure sont envoyées aux processus travaillant sur les sous-domaines adjacents. Cela permet de limiter à nouveau les accès aux données locales.

Une image contenant carré, Caractère coloré, capture d’écran, Rectangle

Description générée automatiquementUne image contenant carré, Caractère coloré, Rectangle, motif

Description générée automatiquementUne image contenant carré, Caractère coloré, Rectangle, capture d’écran

Description générée automatiquement

### Application typique

Une classe d'applications importante utilisant ces schémas d'accès aux données est la solution numérique d'équations aux dérivées partielles. Ces programmes sont itératifs et mettent à jour les points de la grille avec diverses "passes". Entre chaque passe, les données de la bordure intérieure doivent être communiquées. Cela peut être fait soit par dimension (d'abord en x, puis en y, etc.), en utilisant des communications non bloquantes ou un modèle circulaire de MPI\_Sendrecv.

Cette approche permet d'effectuer efficacement des calculs parallèles sur des grilles complexes en minimisant la communication entre les processus, ce qui est essentiel pour obtenir de bonnes performances dans les applications de simulation numérique.

# Échange de données fantômes

L'échange de données fantômes est une technique qui permet de gérer les dépendances diagonales et les frontières périodiques de manière efficace. Elle utilise notamment la fonction MPI\_Sendrecv pour échanger des données avec des processus qui n'existent pas, en utilisant MPI\_PROC\_NULL comme destination pour éviter l'envoi inutile de données.

## Types de données dérivées

Bien que MPI propose de nombreux types de données intégrés qui reflètent principalement les types de données de C/C++ et Fortran, il peut être nécessaire de créer des types de données dérivées pour gérer des structures de données non contiguës. MPI offre plusieurs méthodes pour construire ces types de données :

* Contiguous : Une séquence contiguë d'éléments de même type.
* Vector, Hvector : Une séquence d'éléments avec un espacement fixe entre eux.
* Indexed, Hindexed : Une séquence d'éléments non contigus, avec des indices spécifiés.
* Indexed\_block : Une séquence d'éléments non contigus de longueurs identiques.
* Struct : Une structure de données composite avec des éléments de types différents.

Les routines marquées avec "H" diffèrent des autres en ce sens que les pas et les déplacements de bloc sont spécifiés en octets.

## Création et utilisation d'un nouveau type de données

La création et l'utilisation d'un nouveau type de données en MPI nécessitent deux étapes :

1. Créer le type en utilisant l'une des routines de construction de type de MPI.
2. Valider le type en utilisant MPI\_Type\_commit().

Une fois qu'un type a été validé, il peut être utilisé dans les opérations d'envoi, de réception et d'autres opérations de tampon. Il peut être libéré avec MPI\_Type\_free() une fois qu'il n'est plus nécessaire.

Ces fonctionnalités offrent une flexibilité accrue pour manipuler des données non contiguës dans MPI, ce qui est essentiel pour les applications nécessitant des structures de données complexes et des communications sophistiquées entre les processus.

## Types de données Contiguous et Vector

### Contiguous

Le type de données contiguous permet à un seul type de faire référence à plusieurs éléments contigus d'un type de données existant. Sa syntaxe est la suivante :

int MPI\_Type\_contiguous(

int count, // Nombre de répétitions

MPI\_Datatype oldtype, // Ancien type de données

MPI\_Datatype \*newtype // Nouveau type de données

);

Essentiellement, il crée un tableau de `count` éléments ayant le type `oldtype`. Par exemple, pour envoyer un tableau `a` de `n` éléments de type double en utilisant MPI\_Send, on peut également le faire en créant un nouveau type de données avec MPI\_Type\_contiguous, puis en l'utilisant pour l'envoi :

MPI\_Datatype rowtype;

MPI\_Type\_contiguous(n, MPI\_DOUBLE, &rowtype);

MPI\_Type\_commit(&rowtype);

MPI\_Send(a, 1, rowtype, dest, tag, MPI\_COMM\_WORLD);

### Vector

Le type de données vector est similaire au type de données contiguous, mais il permet un pas non unitaire constant entre les éléments. Sa syntaxe est la suivante :

int MPI\_Type\_vector(

int count, // Nombre de blocs

int blocklength, // Nombre d'éléments dans chaque bloc

int stride, // Nombre d'éléments entre chaque bloc

MPI\_Datatype oldtype, // Ancien type de données

MPI\_Datatype \*newtype // Nouveau type de données

);

Par exemple, si une grille cartésienne `nx \* ny` est stockée de manière contiguë par ligne, on peut créer des types de données pour les lignes et les colonnes de la grille comme suit :

MPI\_Datatype row, column;

MPI\_Type\_vector(nx, 1, 1, MPI\_DOUBLE, &row);

MPI\_Type\_vector(ny, 1, nx, MPI\_DOUBLE, &column);

MPI\_Type\_commit(&row);

MPI\_Type\_commit(&column);

## Types de données Indexed et Struct

### Indexed

Le type de données Indexed permet des pas variables entre les éléments. Sa syntaxe est la suivante :

int MPI\_Type\_indexed(

int count, // Nombre de blocs

int \*blocklengths, // Nombre d'éléments par bloc

int \*displacements, // Déplacement pour chaque bloc

MPI\_Datatype oldtype, // Ancien type de données

MPI\_Datatype \*newtype // Nouveau type de données

);

Il permet de créer des types de données où chaque bloc a une longueur différente et un décalage différent par rapport au début du tableau.

### Struct

Le type de données Struct permet de créer des types de données représentant des structures C/C++ générales. Sa syntaxe est la suivante :

int MPI\_Type\_create\_struct(

int count, // Nombre de blocs

int \*blocklengths, // Nombre d'éléments par bloc

MPI\_Aint \*displacements, // Décalage en octets de chaque bloc

MPI\_Datatype \*datatypes, // Type des éléments de chaque bloc

MPI\_Datatype \*newtype // Nouveau type de données

);

Il permet de définir des types de données complexes avec différents types et longueurs d'éléments, ainsi que des décalages en octets personnalisés.

Ces fonctions offrent une grande flexibilité pour créer des types de données adaptés à une variété de situations, ce qui est essentiel pour gérer efficacement les communications et les opérations de tampon dans MPI.

## Échange de données fantômes

L'échange de données fantômes est essentiel pour maintenir la cohérence des données dans des grilles ou des structures de données partagées entre plusieurs processus MPI. Voici comment cela peut être mis en œuvre pour une grille cartésienne bidimensionnelle.

### Échange des bords supérieurs et inférieurs

Pour échanger les bords supérieurs et inférieurs de la grille, nous utilisons la fonction MPI\_Sendrecv pour envoyer la dernière ligne (`ny-2`) vers le processus voisin supérieur (`next\_y`) et recevoir dans la première ligne de notre processus (`prev\_y`). Cela assure que les bords supérieurs sont mis à jour avec les données correctes provenant des processus voisins.

Une image contenant capture d’écran, Caractère coloré, carré, ligne

Description générée automatiquement

MPI\_Sendrecv(&arr[(ny-2)\*(nx)], 1, row, next\_y, 0, &arr[0], 1, row, prev\_y, 0, MPI\_COMM\_CART, MPI\_STATUS\_IGNORE);

Ensuite, nous échangeons la première ligne (`0+1\*(nx)`) vers le processus voisin inférieur (`prev\_y`) et recevons dans la dernière ligne de notre processus (`ny-1`). Cela assure que les bords inférieurs sont mis à jour correctement.

Une image contenant capture d’écran, Caractère coloré, carré, diagramme

Description générée automatiquement

MPI\_Sendrecv(&arr[0+1\*(nx)], 1, row, prev\_y, 0, &arr[(ny-1)\*(nx)], 1, row, next\_y, 0, MPI\_COMM\_CART, MPI\_STATUS\_IGNORE);

### Échange des bords gauche et droit

Pour échanger les bords gauche et droit de la grille, nous utilisons une approche similaire. Nous envoyons la dernière colonne (`nx-2`) vers le processus voisin de droite (`next\_x`) et recevons dans la première colonne de notre processus (`prev\_x`).

Une image contenant capture d’écran, Caractère coloré, carré, diagramme

Description générée automatiquement

MPI\_Sendrecv(&arr[nx-2], 1, column, next\_x, 0, &arr[0], 1, column, prev\_x, 0, MPI\_COMM\_CART, MPI\_STATUS\_IGNORE);

Ensuite, nous envoyons la deuxième colonne (`1`) vers le processus voisin de gauche (`prev\_x`) et recevons dans la dernière colonne de notre processus (`nx-1`).

Une image contenant Caractère coloré, capture d’écran, carré, Rectangle

Description générée automatiquement

MPI\_Sendrecv(&arr[1], 1, column, prev\_x, 0, &arr[nx-1], 1, column, next\_x, 0, MPI\_COMM\_CART, MPI\_STATUS\_IGNORE);

Ces échanges de données fantômes assurent la cohérence des données à travers les bords de la grille et sont cruciaux pour les applications utilisant des opérations itératives sur des données discrètes.

## Opérations non bloquantes

Les opérations non bloquantes sont une fonctionnalité clé de MPI qui permet à un processus d'exécuter d'autres tâches pendant qu'il attend la fin d'une opération de communication. Cela peut améliorer l'efficacité en permettant à un processus de continuer à travailler sur des calculs locaux pendant que des données sont échangées avec d'autres processus.

## Cellules fantômes et latence de communication

Dans certaines applications, certaines cellules intérieures n'ont pas besoin de la présence de cellules fantômes mises à jour immédiatement. On peut profiter de cela pour masquer la latence des communications. Par exemple, on peut commencer l'échange des cellules fantômes, puis commencer à calculer les nouvelles valeurs pour les cellules intérieures pendant que les données sont en transit. Une fois l'échange des cellules fantômes terminé, on attend leur réception avant de poursuivre les calculs.

# Perspectives - MPI 2 et 3

Le standard MPI 2 a introduit plusieurs nouvelles fonctionnalités, notamment la possibilité de créer et de supprimer des processus dynamiquement pendant l'exécution du programme, les communications unilatérales (one-sided), un ensemble étendu de collectifs, et la prise en charge de l'entrée/sortie parallèle.

Le standard MPI 3 a poursuivi cette évolution en ajoutant des opérations collectives non bloquantes, plus de communications unilatérales, des collectifs de voisinage spécialement conçus pour les topologies cartésiennes et graphiques, ainsi qu'une interface d'outil MPIT pour le profilage et le débogage des programmes MPI.

Ces améliorations ont permis à MPI de rester une bibliothèque de communication de pointe pour les applications parallèles et distribuées, offrant des fonctionnalités avancées pour les programmeurs afin d'optimiser les performances et la scalabilité de leurs applications.