SOMMAIRE

1. Présentation du Calcul Haute Performance
2. Les composants HPC - matériels
3. Les composants HPC - Logiciels
4. HPC - Performance et productivité
5. Mesures de performance
6. Qu'est-ce que le calcul parallèle ?
7. Performance parallèle
8. Limites de l'accélération
9. Loi d'Amdahl
10. Analyse d'une solution parallèle
11. Architecture HPC
12. Mémoire partagée vs. Mémoire distribuée
13. Mémoire partagée : UMA vs. NUMA
14. Clusters : mémoire distribuée
15. Architecture hybride
16. Paradigmes de programmation parallèle
17. Architecture, Paradigme, Modèle
18. Modèles de programmation
19. Besoins parallèlisme
20. Message Passing Interface
21. OpenMB
22. OpenMB vs MPI
23. Hybrid programming model
24. Problématique parallélisme
25. OpenPBS
26. Slurm
27. Implémentation OpenMB
28. Implémentation MPI

SOMMAIRE

28. Technologies scalables cloud : Container as a service, K8S, et Knative, Azure et AWS Batch, Parallel Cluster – GCP Cloud HPC Toulkit – Azure CycleCloud
29. L'utilisation de Apptainer pour la conteneurisation des jobs

Présentation du Calcul Haute Performance

- Dans sa structure la plus simple, les clusters HPC (calcul haute performance) sont conçus pour utiliser le calcul parallèle afin d'appliquer plus de force de processeur pour la résolution d'un problème.
- Les clusters HPC ont généralement un grand nombre d'ordinateurs (souvent appelés « nœuds ») et, en général, la plupart de ces nœuds seraient configurés de manière identique.

Superordinateurs vs. clusters HPC

Présentation du Calcul Haute Performance

- Les systèmes HPC tirent leur puissance de calcul en exploitant le parallélisme.
- Les programmes destinés aux systèmes HPC doivent être divisés en de nombreux sous-programmes plus petits qui peuvent être exécutés en parallèle sur différents processeurs.
- Les systèmes HPC peuvent offrir un parallélisme à une échelle beaucoup plus grande, avec des centaines ou des milliers, voire des millions de tâches s'exécutant simultanément.

Présentation du Calcul Haute Performance

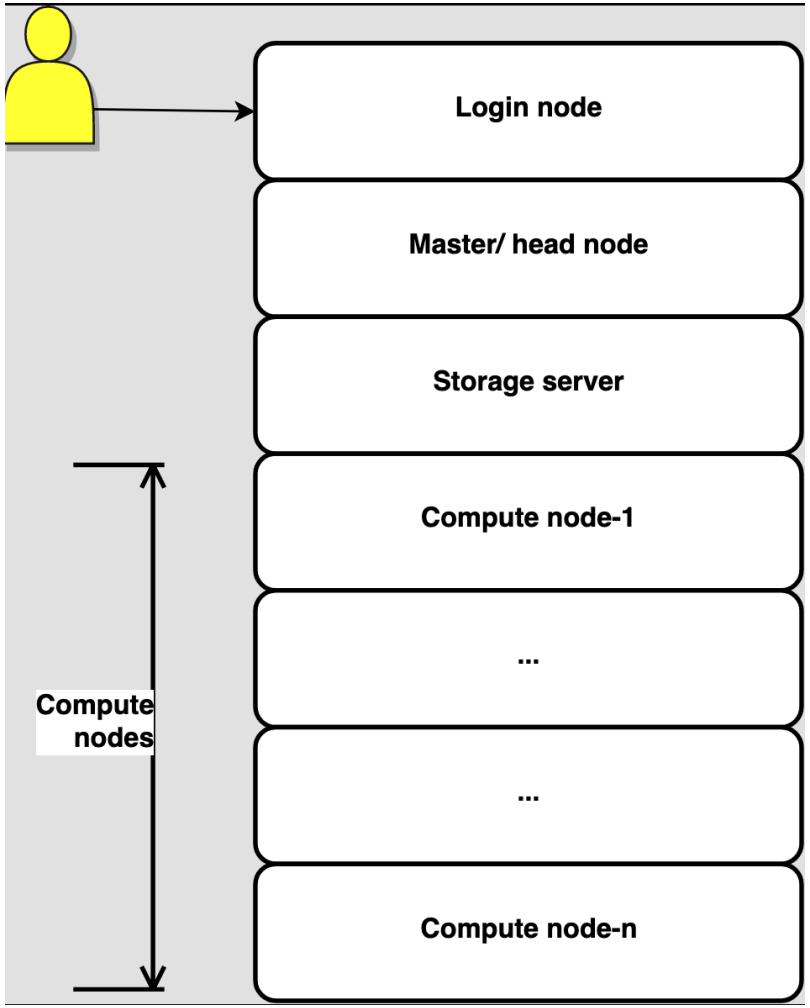
les HPC sont utiles :

- Un programme qui peut être recompilé ou reconfiguré pour utiliser des bibliothèques numériques optimisées disponibles sur les systèmes HPC mais pas sur votre propre système.
- Un problème parallèle, par exemple, vous avez une application unique qui doit être relancée plusieurs fois avec différents paramètres.
- Une application qui a déjà été conçue avec du parallélisme.
- Pour utiliser la grande mémoire disponible.
- Des solutions aux problèmes nécessitant des sauvegardes pour une utilisation future.

Quand ne pas utiliser les systèmes HPC ?

- Vous avez une tâche mono-thread qui ne s'exécutera qu'une seule tâche à la fois.
- Vous dépendez des SGBD / bases de données.
- Vous avez beaucoup de données à transférer entre votre machine locale et le HPC de manière continue (par exemple, par tâche).
- Vous avez besoin d'une interface graphique pour interagir avec votre programme.

Les composants HPC - matériels



Les composants HPC - matériels

- Un cluster est constitué de deux ordinateurs ou plus (souvent bien plus) qui fonctionnent comme un seul système logique pour fournir des services.
- La figure ci-dessus présente les fonctions logiques qu'un nœud physique dans un cluster peut fournir. Plusieurs fonctions logiques peuvent résider sur le même nœud physique, et dans d'autres cas, une fonction logique peut être répartie sur plusieurs nœuds physiques.
- En plus des nœuds du cluster (nœud de gestion, nœuds de calcul et nœuds de stockage) qui composent un cluster, plusieurs autres composants clés doivent également être pris en compte.
  - Commutateurs Ethernet : Des commutateurs Ethernet sont inclus pour fournir la communication nécessaire de nœud à nœud (1/10 GB).
  - Commutateur Infiniband : Pour des réseaux plus rapides (56/100 GB), principalement utilisés par les logiciels activant MPI.

Les composants HPC - matériels

1. Login Node

- Les nœuds individuels d'un cluster sont souvent sur un réseau privé qui ne peut pas être directement accessible de l'extérieur ou du réseau d'entreprise. Même s'ils sont accessibles, la plupart des nœuds de cluster ne seraient pas nécessairement configurés pour fournir une interface utilisateur optimale. Le nœud de connexion est le type de nœud qui est configuré pour fournir cette interface aux utilisateurs (éventuellement sur des réseaux extérieurs) qui peuvent accéder au cluster pour demander l'exécution d'une tâche, ou pour accéder aux résultats d'une tâche précédemment exécutée.

2. Master Node

- Surveiller l'état des nœuds individuels (typiquement les nœuds de calcul).
- Émettre des commandes de gestion aux nœuds individuels.
- Le nœud principal peut également fournir des services réseau qui aident les autres nœuds du cluster à travailler ensemble pour obtenir le résultat souhaité.
- La planification des tâches peut être un autre rôle important pour un nœud principal.

3. Compute Node

- Un nœud de calcul est l'endroit où le véritable calcul est effectué. La majorité des nœuds dans un cluster sont généralement des nœuds de calcul. Pour fournir une solution globale, un nœud de calcul peut exécuter une ou plusieurs tâches, en fonction du système de planification.

Les composants HPC - Logiciels

1. Protocol de connexion
2. Compilateurs de langages de programmation
3. Scripting
4. Formats de fichiers et gestion des données
5. Bibliothèques/Outils de programmation parallèle
6. Planificateurs

Les composants HPC - Logiciels - Modules

- Sur un système HPC, il est nécessaire de rendre disponibles un large choix de paquets logiciels dans plusieurs versions, ce qui peut rendre assez difficile la configuration de l'environnement utilisateur afin de toujours trouver les exécutables et les bibliothèques requis.
- Les modules d'environnement offrent un moyen de activer et de désactiver sélectivement les modifications de l'environnement utilisateur qui permettent de trouver des paquets particuliers et leurs versions.
- Les modules fonctionnent en définissant des variables d'environnement telles que PATH et LD_LIBRARY_PATH
- Pour utiliser un module dans un script de tâche en lot, il suffit de spécifier module load .

Les composants HPC - Logiciels - Job Schedulers

- Dans le domaine du HPC (High Performance Computing), nous parlons beaucoup de tâches. Il s'agit simplement de commandes que nous souhaitons exécuter et de demandes de ressources (par exemple, temps de calcul, espace disque, quantité de mémoire, environnements logiciels, etc.).
- Les tâches HPC sont généralement longues et nécessitent beaucoup de ressources, et se déroulent de manière non interactive.
- Le système de traitement par lots est un programme (qui se trouve généralement sur le nœud principal) qui vous permet d'ajouter et de retirer des tâches de la file d'attente et de surveiller cette file. C'est un programme piloté par script ou ligne de commande.
- Les ordonnanceurs de tâches gèrent les files d'attente de tâches.
  - Portable Batch System (PBS)
  - Simple Linux Utility for Resource Management (SLURM)
  - Moab
  - Univa Grid Engine
  - LoadLeveler, Condor
  - OpenLava

HPC - Performance et productivité

- **High Throughput Computing** : L'objectif est de maximiser le volume de calcul réalisé plutôt que la rapidité d'exécution d'une seule tâche.
- **High Availability Computing** :La capacité d'un système à rester opérationnel et accessible, minimisant ainsi le temps d'arrêt même en cas de défaillance. Cela est crucial pour les applications où il est essentiel que les services et les ressources soient continuellement disponibles.
- **Capacity Computing** : L'utilisation de systèmes informatiques pour gérer des charges de travail régulières et prévisibles à grande échelle.
- **Capability Computing** : L'accent est mis sur l'exécution des tâches de calcul les plus exigeantes et les plus complexes, exploitant pleinement les capacités d'un système pour résoudre des problèmes qui ne pourraient pas être abordés autrement.
- **High Productivity Computing** : HPC devrait signifier "High Productivity Computing" (Calcul de haute productivité) plutôt que simplement "High Performance Computing" (Calcul de haute performance).

HPC - Performance et productivité

- Productivité = (performance de l'application) / (effort de programmation de l'application)
- Les scientifiques dans le domaine du HPC ont des objectifs différents, donc des attentes et des définitions différentes de la productivité

Mesures de performance

- Quelle est la vitesse de calcul de mon CPU ?
- Combien de données puis-je stocker ?
- Quelle est la vitesse de déplacement des données ?
  - Depuis les CPU vers la mémoire ; depuis les CPU vers le disque ; depuis les CPU vers/dans différentes machines
  - Entre les ordinateurs : réseaux
    - par défaut (commodité) : 1 Gb/s
    - sur mesure (haute vitesse) : 10 Gb/s, 20 Gb/s et maintenant 40 Gb/s ou plus
  - Au sein de l'ordinateur :
    - CPU – Mémoire : milliers de Mb/s : 10 - 100 Gb/s
    - CPU – Disques : Mo/s : 50 ~ 100 Mo/s jusqu'à 1000 Mo/s

Qu'est-ce que le calcul parallèle ?

- Le calcul parallèle est l'exécution simultanée de la même tâche (divisée et spécialement adaptée) sur plusieurs processeurs afin d'obtenir des résultats plus rapidement
- Le processus de résolution d'un problème peut généralement être divisé en tâches plus petites, qui peuvent être effectuées simultanément avec une certaine coordination

Performance parallèle

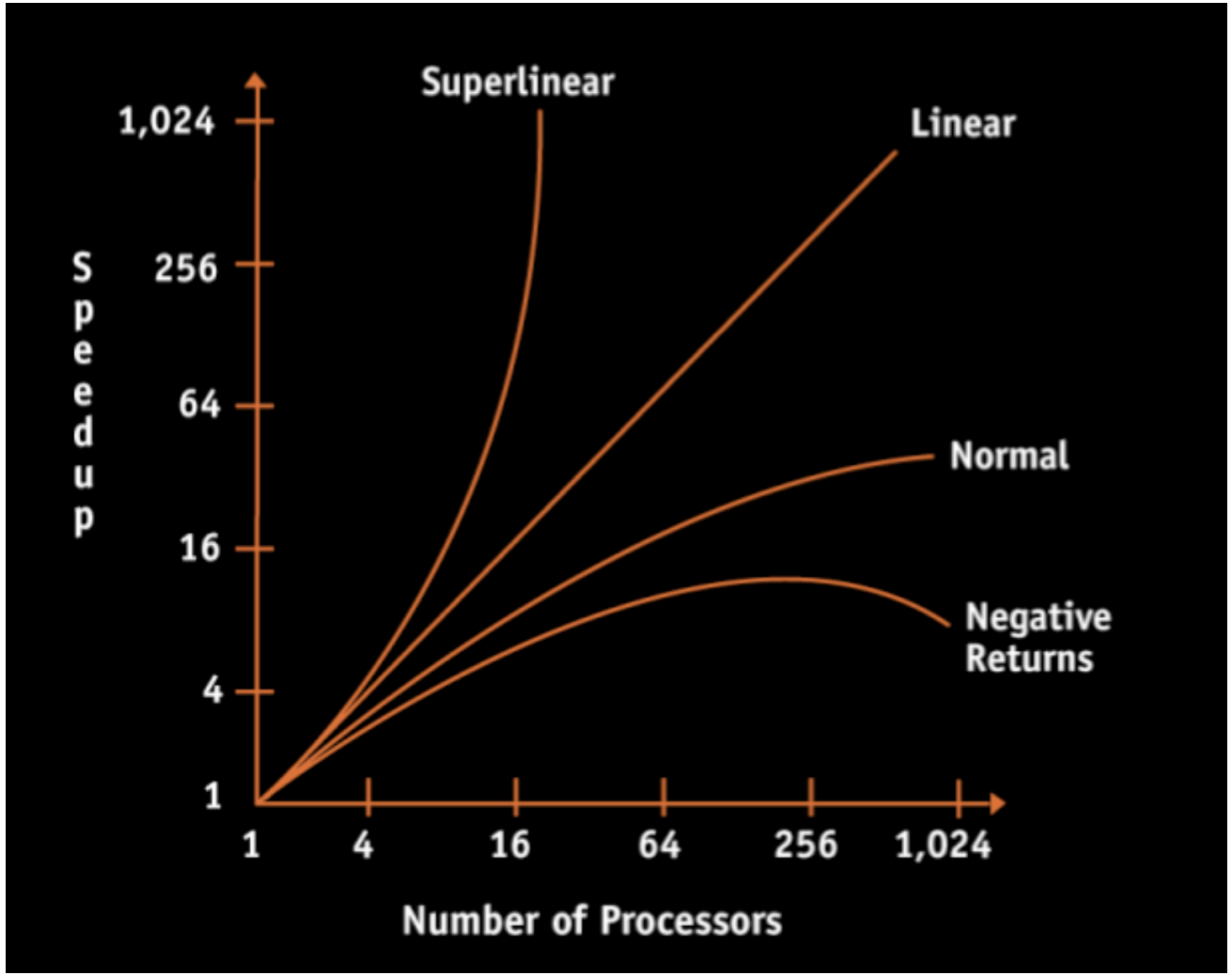
L'accélération d'une application parallèle, souvent mesurée par le "speedup":

- Le speedup d'une application parallèle est défini par la formule:

$$\text{Speedup}(p) = \frac{\text{Temps}(1)}{\text{Temps}(p)}$$

  - **Temps(1)** est le temps d'exécution de l'application sur un seul processeur.
  - **Temps(p)** est le temps d'exécution de l'application en utilisant (p) processeurs parallèles.
- Si **Speedup(p) = p**, cela signifie que l'accélération est parfaite, également connue sous le nom de scalabilité linéaire. Cela implique que l'ajout de processeurs réduit le temps d'exécution proportionnellement au nombre de processeurs ajoutés, sans aucune perte de performance due à la coordination ou la communication entre les processeurs.
- Pour une évaluation précise, il est recommandé de comparer :
  - Le temps d'exécution de la meilleure application sérielle sur un processeur.
  - Le temps d'exécution du meilleur algorithme parallèle sur (p) processeurs.

Performance parallèle



Question

- L'algorithme A et l'algorithme B résolvent en parallèle le même problème
- Nous savons que sur 64 cœurs :
  - Le programme A obtient une accélération de 50
  - Le programme B obtient une accélération de 4
- Lequel choisissez-vous ?
  - 1. programme A
  - 2. programme B
  - 3. Aucun des deux

Limites de l'accélération

- Tous les programmes parallèles contiennent :
  - Sections parallèles
  - Sections sérielles
- Les sections sérielles limitent l'accélération :
  - Manque de parallélisme parfait dans l'application ou l'algorithme
  - Équilibrage de charge imparfait (certains processeurs ont plus de travail)
  - Coût de la communication
  - Coût de la contention pour les ressources, par exemple, bus mémoire, E/S
  - Temps de synchronisation
- Comprendre pourquoi une application ne s'adapte pas linéairement aide à trouver des moyens d'améliorer les performances de l'application sur les ordinateurs parallèles

Loi d'Amdahl

La loi d'Amdahl est un principe fondamental en informatique qui sert à prédire l'amélioration théorique de la performance d'un système lorsqu'une partie de ce système est optimisée. Formulée par Gene Amdahl dans les années 1960, cette loi est particulièrement importante dans le domaine du calcul parallèle pour évaluer l'impact du parallélisme sur l'accélération globale d'un programme.
La loi d'Amdahl s'exprime comme suit :

$$\text{Speedup}_{\text{max}} = \frac{1}{(1-p) + \frac{p}{n}}$$

- (p) est la proportion du programme qui peut être parallélisée,
- (n) est le nombre de processeurs utilisés,
- (1 - p) est la proportion du programme qui doit rester séquentielle.
- L'idée centrale derrière la loi d'Amdahl est que l'amélioration de la performance d'un système est limitée par la partie du système qui ne peut pas être améliorée. Cela signifie que même si une grande partie d'un programme peut être parallélisée, la portion séquentielle restante limite le gain maximal en performance que le parallélisme peut apporter.

Performance parallèle efficace

- Il y a 3 façons d'améliorer les performances :
  - Utiliser du matériel plus rapide
  - Optimiser les algorithmes et les techniques utilisées pour résoudre les tâches de calcul
  - Utiliser plusieurs ordinateurs pour résoudre une tâche particulière
- Les trois stratégies peuvent être utilisées simultanément

Exercice

- L'objectif de cet exercice est de comparer les performances de la simulation de diffusion de chaleur en 2D en version séquentielle et parallèle.
- Calculer le speedup et l'efficacité en utilisant différents nombres de CPU, puis tirer des conclusions sur les gains de performance obtenus grâce au parallélisme.

Contexte de l'Exercice: La diffusion de chaleur est un phénomène physique où la chaleur se propage dans un milieu. En 2D, ce phénomène peut être modélisé par l'équation de la chaleur discrétisée sur une grille. Chaque point de la grille représente la température à cet endroit, et la température à chaque point est mise à jour en fonction des températures de ses voisins.

Analyse d'une solution parallèle

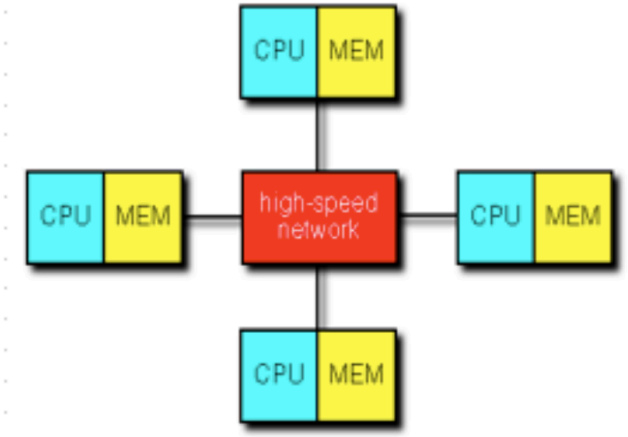
- Décomposition fonctionnelle :
  - Différentes personnes exécutent différentes tâches
- Décomposition de domaine :
  - Différentes personnes exécutent les mêmes tâches

Architecture HPC

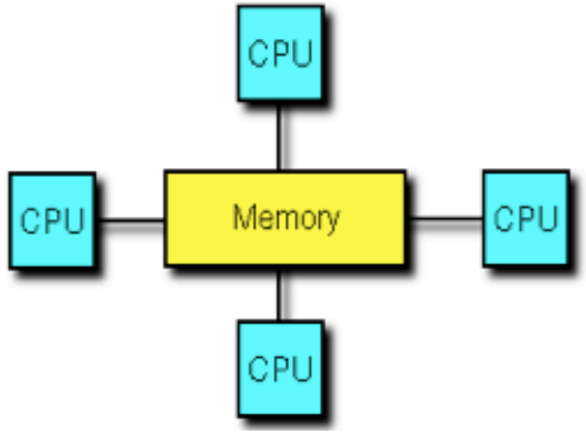
- La manière la plus simple et la plus utile de classer les ordinateurs parallèles modernes est par leur modèle de mémoire
  - Comment les CPU voient et peuvent accéder à la mémoire disponible ?
    - Mémoire partagée
    - Mémoire distribuée

Mémoire partagée vs. Mémoire distribuée

- Mémoire distribuée :
  - Chaque processeur a sa propre mémoire locale. Doit faire passer des messages pour échanger des données entre les processeurs
  - Multi-ordinateurs

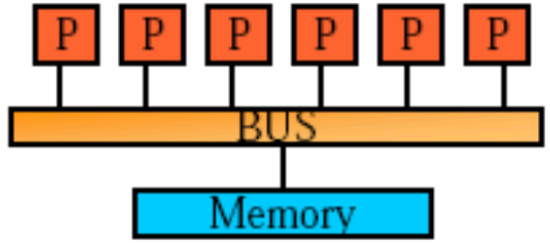


- Mémoire partagée :
  - Espace d'adresse unique. Tous les processeurs ont accès à un pool de mémoire partagée
  - Multi-processeurs

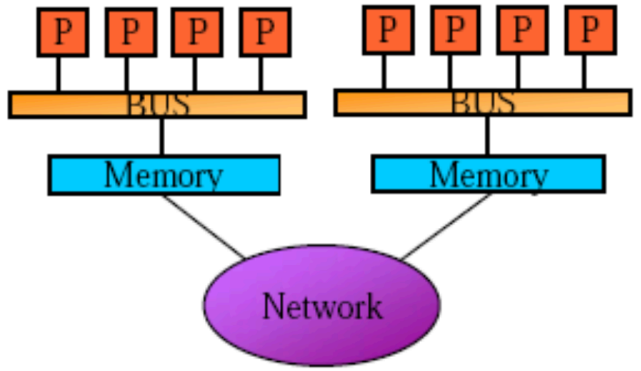


Mémoire partagée : UMA vs. NUMA

- Accès uniforme à la mémoire (UMA) : Chaque processeur a un accès uniforme à la mémoire. Aussi connu sous le nom de multiprocesseurs symétriques (SMP)

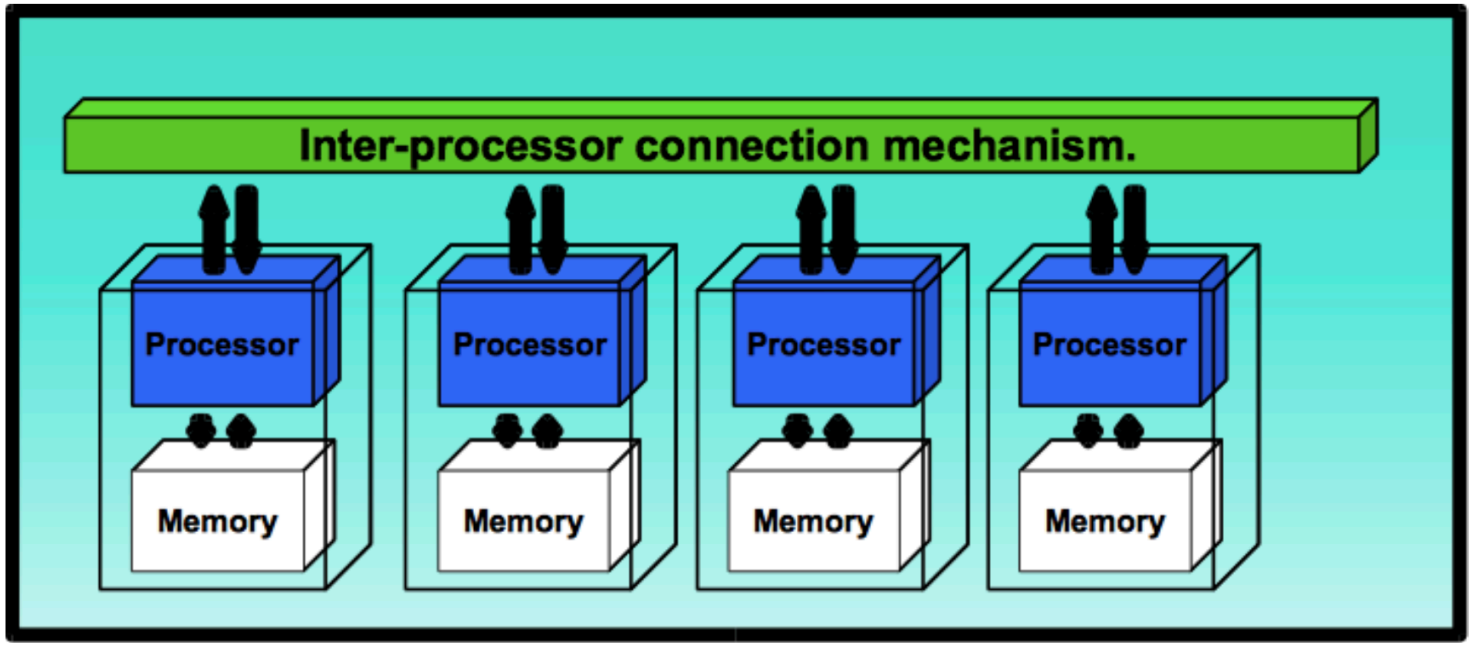


- Accès non uniforme à la mémoire (NUMA) : Le temps d'accès à la mémoire dépend de l'emplacement des données. L'accès local est plus rapide que l'accès non local



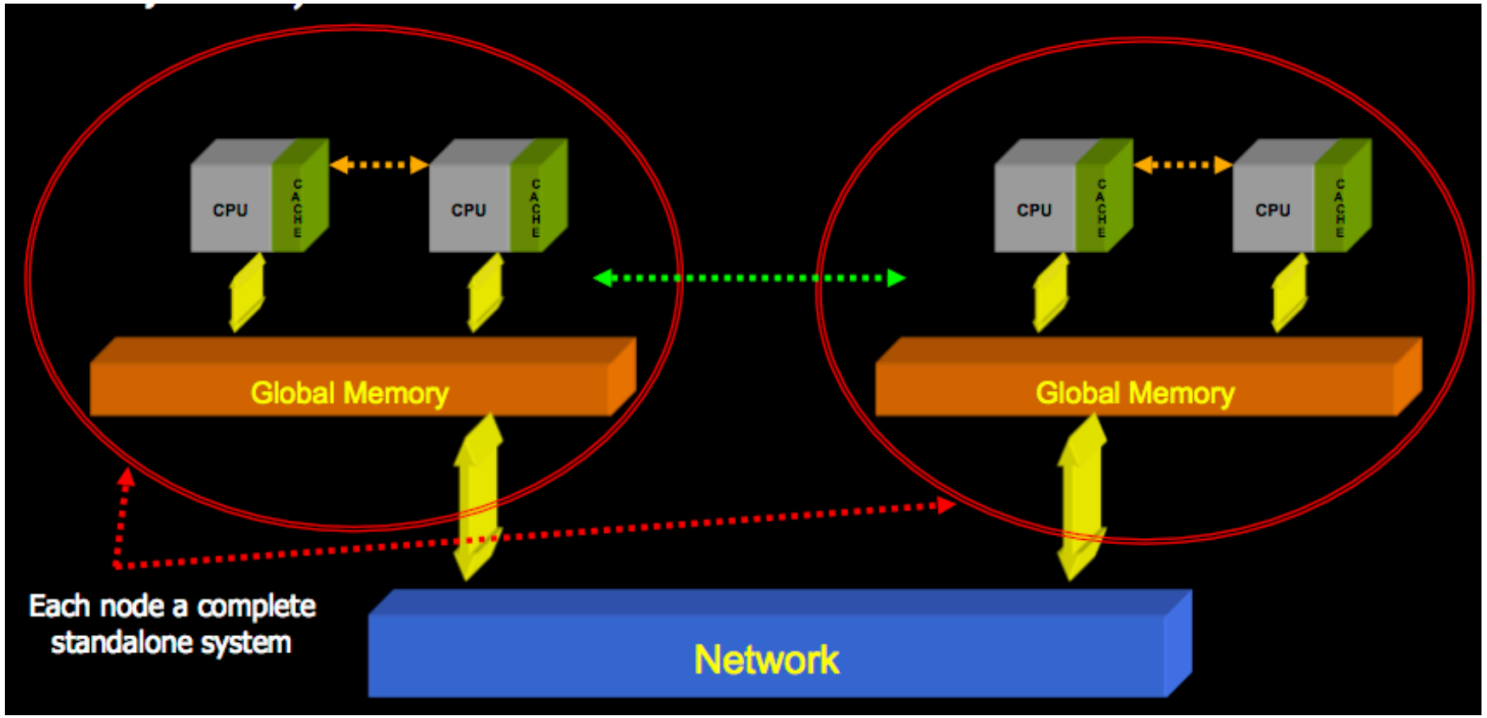
Clusters : mémoire distribuée

- Machines indépendantes combinées en un système unifié par des logiciels et des réseaux



Architecture hybride

- Tous les clusters modernes ont en réalité une architecture hybride
- Les CPU multi-cœurs rendent chaque nœud un petit système SMP



Paradigmes de programmation parallèle

- Les modèles de mémoire déterminent les paradigmes de programmation
- Mémoire distribuée
  - Passage de message
  - Tous les processus pourraient accéder directement à leur mémoire locale uniquement. Les messages explicites sont demandés pour accéder à la mémoire distante de différents processeurs
- Mémoire partagée
  - Parallélisme de données
  - Vue mémoire unique. Tous les processus (généralement des threads) peuvent accéder directement à l'ensemble de la mémoire

Architecture, Paradigme, Modèle

Mémoire distribuée

- Passage de message
- Décomposition de domaine

Mémoire partagée

- Parallélisme de données
- Décomposition fonctionnelle

Modèles de programmation

Décomposition de domaine

- Les données sont divisées en segments égaux et distribuées aux CPU disponibles.
- Chaque CPU traite ses propres données locales.
- Échange de données si nécessaire.

Décomposition fonctionnelle

- Le problème est décomposé en plusieurs sous-tâches.
- Chaque CPU exécute l'une des sous-tâches.
- Semblable au paradigme serveur/client.

Besoins parallèlisme

1. **Équilibrage de la charge**

- **S'applique aux calculs, aux opérations d'E/S, aux communications réseau** :
  - L'équilibrage de la charge concerne la répartition équitable du travail parmi les différentes ressources disponibles, qu'il s'agisse de calculs intensifs, d'entrées/sorties (E/S) ou de communications réseau. Cela garantit que aucune ressource n'est surchargée pendant que d'autres restent inactives.
- **Relativement facile pour la décomposition de domaine, pas si facile pour la décomposition fonctionnelle** :
  - Lorsqu'on utilise la décomposition de domaine, il est plus facile de diviser le travail de manière égale car les données peuvent être segmentées uniformément et distribuées aux différentes unités de traitement. En revanche, avec la décomposition fonctionnelle, il peut être plus difficile de s'assurer que chaque sous-tâche a une charge de travail équivalente, car certaines fonctions peuvent être plus complexes ou plus gourmandes en ressources que d'autres.

Besoins parallèlisme

2. **Minimisation de la communication**

- **Joindre les communications individuelles** :
  - Réduire le nombre de communications en les regroupant peut diminuer le temps consacré aux échanges de messages entre les unités de traitement. Par exemple, au lieu d'envoyer plusieurs petits messages, il est plus efficace d'envoyer un message unique et plus grand.
- **Éliminer la synchronisation – le processus le plus lent domine** :
  - La synchronisation entre les processus peut entraîner des ralentissements car tous les processus doivent attendre que le processus le plus lent termine avant de passer à l'étape suivante. En minimisant les points de synchronisation, on peut réduire ces temps d'attente et améliorer l'efficacité globale.

Besoins parallèlisme

3. Chevauchement des calculs et des communications

- **C'est essentiel pour un véritable parallélisme !** :
  - Pour atteindre un parallélisme efficace, il est crucial de chevaucher les phases de calculs et de communications. Cela signifie que pendant que certaines unités de traitement effectuent des calculs, d'autres peuvent simultanément gérer les communications nécessaires. En faisant cela, on utilise les ressources de manière optimale et on réduit les temps d'attente, maximisant ainsi les performances du système parallèle.

Message Passing Interface

- Les programmes parallèles consistent en des processus séparés, chacun ayant son propre espace d'adressage
  - **Le programmeur gère la mémoire en plaçant les données dans un processus particulier** :
  - Dans les programmes parallèles, chaque processus fonctionne indépendamment avec son propre espace mémoire. Il incombe au programmeur de décider dans quel processus les données doivent être placées.
- Les données sont envoyées explicitement entre les processus
  - **Le programmeur gère le mouvement de la mémoire** :
  - La transmission des données entre les processus doit être faite de manière explicite par le programmeur. Cela signifie que le programmeur doit coder manuellement la façon dont les données se déplacent d'un processus à un autre.

Message Passing Interface

- Opérations collectives
  - **Sur un ensemble arbitraire de processus** :
  - Les opérations collectives impliquent plusieurs processus qui coopèrent pour accomplir une tâche commune. Cela peut inclure des opérations telles que la diffusion de données, la collecte de résultats ou des opérations de réduction (par exemple, calculer la somme ou le maximum de valeurs réparties sur plusieurs processus).
- Distribution des données
  - **Également gérée par le programmeur** :
  - La répartition des données entre les différents processus doit également être gérée manuellement par le programmeur. Cela inclut la décision de quelles données doivent aller à quel processus pour optimiser la performance et l'efficacité de l'application parallèle.

OpenMP

- **Bonne pour les nœuds SMP, mais aussi possible sur des clusters via des systèmes de mémoire virtuelle partagée distribuée** :
  - La mémoire partagée est utile pour les systèmes multiprocesseurs symétriques (SMP) et peut également être mise en œuvre sur des clusters en utilisant des systèmes de mémoire virtuelle partagée distribuée.
- **Parallélisme obtenu grâce au partage de mémoire** :
  - Le parallélisme est atteint en permettant à plusieurs processus ou threads d'accéder à la même mémoire partagée, ce qui facilite la communication et la coordination entre eux.
- **Le programmeur est responsable de la synchronisation correcte** :
  - Il incombe au programmeur de s'assurer que l'accès concurrent à la mémoire partagée est correctement synchronisé pour éviter les conditions de course et autres problèmes liés à la concurrence.
- **Le programmeur est également responsable de la gestion de la mémoire (verrous)** :
  - La gestion de la mémoire partagée, y compris l'utilisation des verrous pour protéger les sections critiques du code, est également sous la responsabilité du programmeur. Cela inclut l'utilisation de mécanismes de verrouillage pour garantir que seules certaines parties du code peuvent accéder à la mémoire partagée à un moment donné pour prévenir les conflits.

OpenMP VS MPI

1. MPI (avantages) :

- **Portable sur les machines à mémoire distribuée et partagée** :
  - Les programmes utilisant MPI (Message Passing Interface) peuvent être exécutés sur des systèmes avec mémoire distribuée ainsi que sur des systèmes avec mémoire partagée.
- **S'étend au-delà d'un seul nœud** :
  - Les programmes MPI peuvent être exécutés sur plusieurs nœuds dans un cluster, permettant ainsi une grande scalabilité.
- **Pas de problème de placement des données** :
  - Avec MPI, chaque processus gère ses propres données, ce qui élimine les problèmes de placement de données associés à la mémoire partagée.

OpenMP VS MPI

2. Pure MPI (inconvénients) :

- **Difficile à développer et à déboguer** :
  - Le développement et le débogage des programmes MPI peuvent être complexes en raison de la gestion explicite des communications entre les processus.
- **Haute latence, faible bande passante** :
  - Les communications entre les processus peuvent souffrir de haute latence et de faible bande passante, ce qui peut ralentir les performances.
- **Communication explicite** :
  - Les communications doivent être programmées explicitement par le développeur, ce qui ajoute une charge de travail supplémentaire.
- **Granularité grossière** :
  - Les opérations de communication en MPI ont tendance à être de grande granularité, ce qui peut limiter la flexibilité.
- **Équilibrage de charge difficile** :
  - L'équilibrage de charge peut être difficile à réaliser avec MPI, car les processus peuvent avoir des charges de travail très différentes.

OpenMP VS MPI

3. OpenMP (avantages) :

- **Facile à implémenter le parallélisme** :
  - OpenMP (Open Multi-Processing) permet d'ajouter facilement du parallélisme aux programmes en utilisant des directives simples.
- **Faible latence, haute bande passante** :
  - Les threads OpenMP peuvent communiquer rapidement grâce à la mémoire partagée, ce qui entraîne une faible latence et une haute bande passante.
- **Communication implicite** :
  - La communication entre les threads est implicite, ce qui simplifie le code.
- **Granularité fine et grossière** :
  - OpenMP permet de travailler avec des granularités fines et grossières, offrant plus de flexibilité dans la gestion des tâches.
- **Équilibrage de charge dynamique** :
  - OpenMP supporte l'équilibrage de charge dynamique, permettant de mieux répartir les tâches entre les threads en fonction de la charge de travail.

OpenMP VS MPI

4. Pure OpenMP (inconvénients) :

- **Uniquement sur les machines à mémoire partagée** :
  - OpenMP ne fonctionne que sur des systèmes avec mémoire partagée, ce qui limite son utilisation sur les systèmes distribués.
- **S'étend à un seul nœud** :
  - OpenMP est limité à un seul nœud et ne peut pas être utilisé efficacement pour des systèmes multi-nœuds.
- **Problème possible de placement des données** :
  - Le placement des données peut devenir problématique, car toutes les données sont accessibles à tous les threads, ce qui peut entraîner des conflits.
- **Pas d'ordre spécifique des threads** :
  - Il n'y a pas de garantie d'ordre d'exécution des threads, ce qui peut compliquer la synchronisation et la gestion des ressources.

Hybrid programming model

- **Élégant en concept et en architecture : utilisant MPI entre les nœuds et OpenMP à l'intérieur des nœuds** :
  - Cette approche combine l'utilisation de MPI pour la communication entre les différents nœuds du cluster et OpenMP pour la parallélisation au sein de chaque nœud. Cela permet de tirer parti des avantages des deux modèles.
- **Bonne utilisation des ressources des systèmes à mémoire partagée (mémoire, latence et bande passante)** :
  - L'approche hybride maximise l'utilisation des ressources des systèmes à mémoire partagée, en optimisant l'accès à la mémoire, en réduisant la latence et en augmentant la bande passante.
- **Évite la surcharge de communication supplémentaire avec MPI à l'intérieur du nœud** :
  - En utilisant OpenMP à l'intérieur des nœuds, on évite les surcharges de communication qui pourraient survenir si MPI était utilisé pour la communication intra-nœud.

Hybrid programming model

- **OpenMP ajoute une granularité fine (tailles de messages plus grandes) et permet un équilibrage de charge accru et/ou dynamique** :
  - OpenMP permet de gérer des tâches de granularité plus fine, ce qui améliore l'efficacité des communications. Il facilite également un équilibrage de charge plus flexible et dynamique.
- **Certains problèmes ont naturellement un parallélisme à deux niveaux** :
  - Certaines applications se prêtent naturellement à une parallélisation à deux niveaux, ce qui est idéal pour une approche hybride.
- **Certains problèmes pourraient n'utiliser qu'un nombre restreint de tâches MPI** :
  - Il peut y avoir des cas où le nombre de tâches MPI est limité, ce qui rend l'approche hybride plus avantageuse.
- **Pourrait avoir une meilleure scalabilité que les modèles MPI pur et OpenMP pur** :
  - Le paradigme hybride peut offrir une scalabilité supérieure en combinant les points forts de MPI et d'OpenMP, dépassant ainsi les limites de chacun lorsqu'ils sont utilisés séparément.

Problématique parallélisme

- **Communication intra-nœud inutile (avec MPI)** :
  - Utiliser MPI pour la communication au sein d'un même nœud peut entraîner des communications inutiles et inefficaces.
- **Problème de saturation (avec MPI)** :
  - La saturation des communications peut se produire lorsque trop de messages sont envoyés simultanément, ce qui peut réduire les performances.
- **Threads en veille (avec OpenMP)** :
  - Avec OpenMP, certains threads peuvent rester en veille, ce qui signifie qu'ils ne sont pas utilisés de manière optimale.
- **Problème de bande passante inter-nœuds (avec hybride)** :
  - Dans une approche hybride, la bande passante limitée entre les nœuds peut devenir un goulot d'étranglement, affectant les performances globales.

Problématique parallélisme

- **Surcharge supplémentaire d'OpenMP (avec hybride)** :
  - **Démarrage/jonction des threads** :
    - Le démarrage et la jonction des threads OpenMP ajoutent une surcharge supplémentaire.
  - **Vidage du cache (thread source des données)** :
    - Le vidage du cache, nécessaire lorsque les données sont transférées entre threads, peut également ajouter une surcharge.
- **Problème d'application** :
  - Certaines applications peuvent ne pas se prêter facilement au chevauchement de la communication et des calculs.
- **Problème de programmation** :
  - La programmation pour chevaucher efficacement la communication et les calculs peut être complexe et sujette à des erreurs.
- **Problème d'équilibrage de charge** :
  - Un équilibrage de charge inefficace peut entraîner des temps d'attente et des inefficacités dans l'exécution parallèle.

Exercice

Contexte: Vous êtes un architecte logiciel chargé de concevoir une solution parallèle pour un projet de simulation scientifique complexe. Vous devez choisir le modèle de programmation parallèle approprié pour différentes sections de l'application et analyser les avantages et les inconvénients de chaque modèle en fonction de différents critères.

Étude de Cas : Simulation de Météo

Vous devez paralléliser une application de simulation de météo qui modélise les conditions climatiques sur une région géographique en utilisant une grille 3D (latitude, longitude, altitude). Les calculs incluent la simulation des mouvements de l'air, des températures, des précipitations, etc.

- **Question 1** : Quels sont les avantages et les inconvénients d'utiliser la mémoire partagée (OpenMP) pour cette application ?
- **Question 2** : Quels sont les avantages et les inconvénients d'utiliser la mémoire distribuée (MPI) pour cette application ?
- **Question 3** : Quels sont les avantages et les inconvénients d'utiliser un modèle hybride (MPI + OpenMP) pour cette application ?
- **Question 4** : Comment équilibreriez-vous la charge entre les différents processus ou threads dans chacun des modèles de programmation ?

Exercice

- **Question 5** : Quels seraient les moyens de minimiser la communication dans chacun des modèles de programmation ?
- **Question 6** : Comment géreriez-vous les ressources (mémoire, bande passante, temps de calcul) de manière efficace dans chacun des modèles de programmation ?
- **Question 7** : Quelle serait la complexité de développement pour chacun des modèles de programmation et quels outils ou techniques utiliseriez-vous pour faciliter le développement et le débogage ?
- **Question 8** : Quels problèmes potentiels pourriez-vous rencontrer avec la mémoire partagée et comment les résoudre ?
- **Question 9** : Quels problèmes potentiels pourriez-vous rencontrer avec la mémoire distribuée et comment les résoudre ?
- **Question 10** : Quels problèmes potentiels pourriez-vous rencontrer avec le modèle hybride et comment les résoudre ?
- **Question 11** : En comparant les trois modèles, lequel choisiriez-vous pour cette application spécifique et pourquoi ?

Introduction à PBS

Le Portable Batch System (PBS) est un logiciel informatique qui effectue la planification des tâches. Sa tâche principale est d'allouer des tâches de calcul, c'est-à-dire des travaux par lots, parmi les ressources informatiques disponibles. Les versions suivantes de PBS sont actuellement disponibles :

- OpenPBS
- TORQUE
- PBS Professional (PBS Pro)

Principales caractéristiques de PBS Pro

- Scalabilité : prend en charge des millions de cœurs avec une répartition rapide des tâches et une latence minimale ; testé au-delà de 50 000 nœuds.
- Planification basée sur des politiques : répond aux objectifs uniques du site et aux SLA en équilibrant le temps de traitement des tâches et l'utilisation avec un placement optimal des tâches.
- Résilience : inclut une architecture de basculement automatique sans point de défaillance unique - les travaux ne sont jamais perdus et continuent de s'exécuter malgré les pannes.
- Cadre de plugin flexible : simplifie l'administration avec une visibilité et une extensibilité améliorées ; personnalise les implémentations pour répondre aux exigences complexes.

PBS Commande basique

Du point de vue de l'utilisateur, le PBS permet d'effectuer trois actions :

- Ajouter un travail à la file d'attente ;
- Retirer un travail de la file d'attente ; et
- Voir où se trouve votre travail dans la file d'attente (stat).
  - qsub
  - qdel
  - qstat
  - qalter

Exercice : Simulation de Monte Carlo pour l'Évaluation des Options avec MPI

Objectif

- Apprendre à soumettre des jobs MPI via OpenPBS.
- Déboguer un programme MPI.
- Mesurer les performances et calculer le speedup.

Description: Vous allez utiliser un programme MPI pour effectuer une simulation de Monte Carlo afin d'évaluer le prix d'une option financière. Vous soumettrez ce programme via OpenPBS, déboguerez les éventuels problèmes et mesurerez les performances par rapport à une version séquentielle.

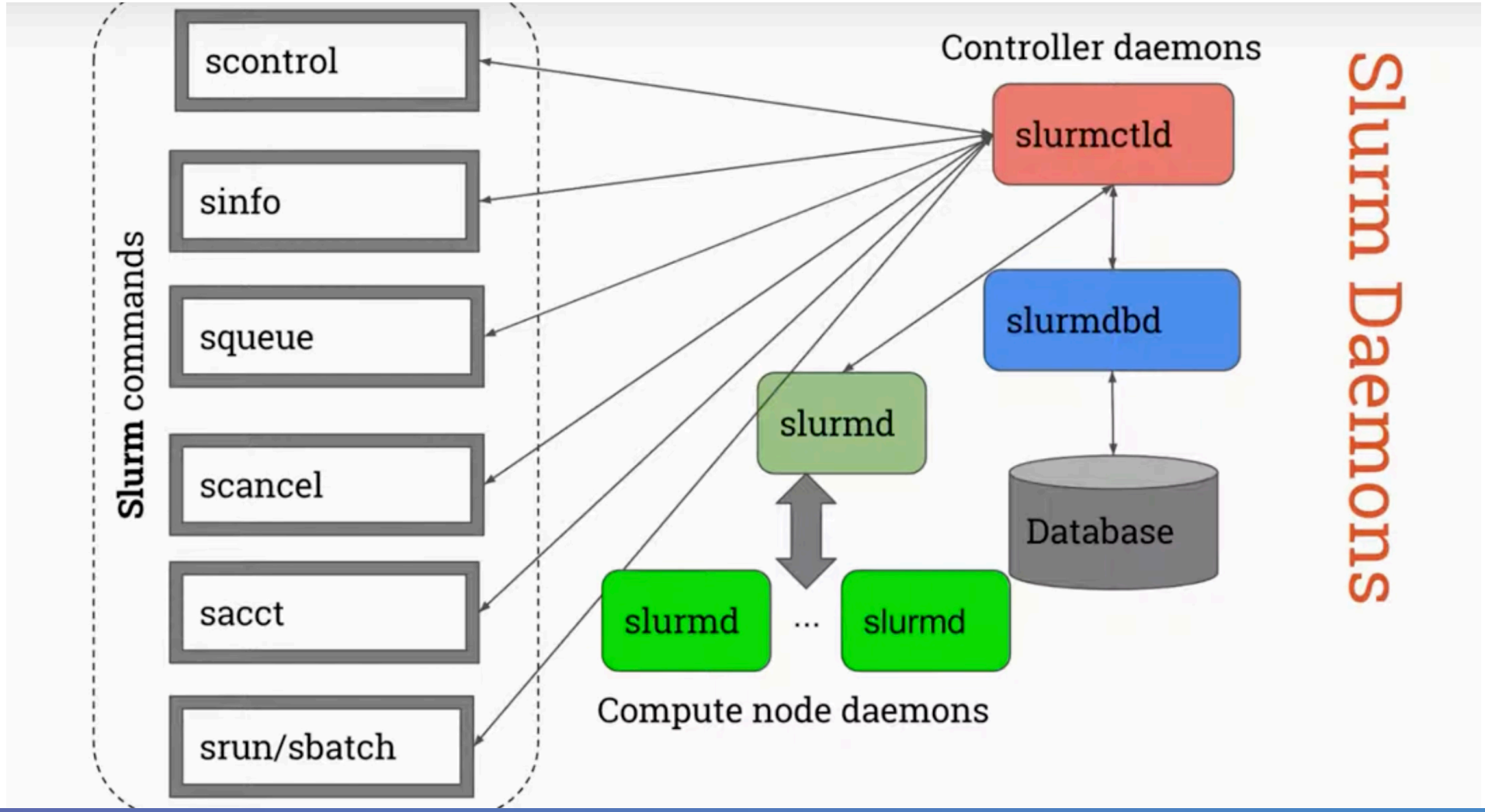
1. Compiler et exécuter le programme séquentiel pour la simulation de Monte Carlo.
2. Compiler et exécuter le programme parallèle MPI pour la simulation de Monte Carlo.
3. Soumettre les jobs MPI via OpenPBS.
4. Déboguer les éventuels problèmes.
5. Mesurer les performances et calculer le speedup.

SLURM

Le gestionnaire de charges de travail Slurm (anciennement connu sous le nom de Simple Linux Utility for Resource Management ou SLURM) est un ordonnanceur de tâches libre et open-source pour les noyaux Linux et Unix-like, utilisé par de nombreux supercalculateurs et clusters de calcul dans le monde. Il offre trois fonctions clés.

1. Tout d'abord, il alloue aux utilisateurs un accès exclusif et/ou non exclusif aux ressources (nœuds de calcul) pour une certaine durée afin qu'ils puissent effectuer des travaux.
2. Ensuite, il fournit un cadre pour démarrer, exécuter et surveiller les travaux (généralement un travail parallèle tel que MPI) sur un ensemble de nœuds alloués.
3. Enfin, il arbitre la concurrence pour les ressources en gérant une file d'attente de travaux en attente.

SLURM


scontrol
sinfo
squeue
scancel
sacct
srun/sbatch
Slurm commands
Controller daemons
slurmctld
slurmdbd
slurmd
Database
slurmd
...
slurmd
Compute node daemons
Slurm Daemons

SLURM

1. **Démons et architecture**:
   - **slurmd** : C'est le démon qui fonctionne sur chaque nœud de calcul dans le cluster. Il est responsable de l'exécution des tâches assignées à ce nœud. Il assure également des communications tolérantes aux pannes de manière hiérarchique.
   - **slurmctld** : C'est le démon central qui fonctionne sur un nœud de gestion ou principal. Il gère les demandes de ressources et contrôle les opérations de l'ensemble du cluster.
2. **Commandes utilisateur** :

- **sacct** : Utilisé pour rapporter les informations sur les travaux et les étapes de travaux terminés.
- **salloc** : Utilisé pour obtenir une allocation de ressources (nœuds de calcul) pour un travail.
- **sattach** : Permet à un utilisateur de se connecter à une tâche en cours d'exécution.
- **sbatch** : Soumet un script de lot à la planification.
- **sbcast** : Diffuse un fichier à tous les nœuds de calcul d'une allocation.
- **scancel** : Annule un ou plusieurs travaux.
- **scontrol** : Utilisé pour afficher ou modifier la configuration du système et les états des travaux.
- **sinfo** : Affiche les informations sur l'état des partitions et des nœuds.
- **smap** : Affiche une carte graphique des nœuds et des travaux.
- **squeue** : Affiche la liste des travaux en attente ou en cours d'exécution.
- **srun** : Utilisé pour soumettre et exécuter une tâche de calcul.
- **strigger** : Déclenche des actions basées sur des événements définis.
- **sview** : Fournit une interface graphique pour surveiller et gérer les travaux et les nœuds.

SLURM

1. **Nœuds** :
   - Ce sont les ressources physiques (ou virtuelles) de calcul disponibles dans le cluster. Chaque nœud peut exécuter des tâches en fonction des ressources qu'il possède (CPU, mémoire, etc.).
2. **Partitions** :
   - Ce sont des groupes de nœuds organisés en ensembles logiques. Contrairement aux files d'attente dans PBS, les partitions peuvent se chevaucher, c'est-à-dire qu'un même nœud peut appartenir à plusieurs partitions. Les partitions permettent de gérer les ressources en fonction de différents critères (par exemple, type de travail, priorité, etc.).
3. **Jobs** :
   - Ce sont des allocations de ressources (nœuds) attribuées à un utilisateur pour une période de temps spécifiée. Un travail peut inclure une ou plusieurs tâches à exécuter sur les nœuds alloués.
4. **Step Jobs** :
   - Ce sont des ensembles de tâches au sein d'un travail. Une étape de travail peut être composée de tâches parallèles, par exemple des processus MPI (Message Passing Interface). Les étapes de travail permettent de structurer et de gérer plus finement l'exécution des tâches dans un travail.

SLURM

- **Partitions comme files d'attente** :
  - Chaque partition peut être vue comme une file d'attente de travaux avec des contraintes spécifiques telles que la limite de taille des travaux (nombre de nœuds maximum qu'un travail peut utiliser), la limite de temps des travaux (durée maximale d'exécution), et les utilisateurs autorisés à soumettre des travaux à cette partition. Ces contraintes permettent d'organiser et de prioriser les ressources de manière efficace, en assurant que les ressources sont utilisées de manière optimale et que les travaux importants ou urgents peuvent être traités en priorité.

Exercice : Simulation de Monte Carlo pour l'Évaluation des Options avec MPI

Objectif

- Apprendre à soumettre des jobs MPI via OpenPBS.
- Déboguer un programme MPI.
- Mesurer les performances et calculer le speedup.

Description: Vous allez utiliser un programme MPI pour effectuer une simulation de Monte Carlo afin d'évaluer le prix d'une option financière. Vous soumettrez ce programme via SLURM, déboguerez les éventuels problèmes et mesurerez les performances par rapport à une version séquentielle.

1. Compiler et exécuter le programme séquentiel pour la simulation de Monte Carlo.
2. Compiler et exécuter le programme parallèle MPI pour la simulation de Monte Carlo.
3. Soumettre les jobs MPI via SLURM.
4. Déboguer les éventuels problèmes.
5. Mesurer les performances et calculer le speedup.

Implémentation OpenMB

-OpenMP est un modèle de programmation explicite (non automatique), offrant au programmeur un contrôle total sur la parallélisation. Il utilise le modèle fork-join d'exécution parallèle en plus des flux de programme basés sur des threads et des directives du compilateur. Tous les programmes OpenMP commencent comme un seul processus (thread maître). Le thread maître s'exécute séquentiellement jusqu'à ce que la première construction de région parallèle soit rencontrée. Le thread maître crée alors (fork) une équipe de threads parallèles. Lorsque les threads de l'équipe terminent les instructions dans la construction de région parallèle, ils se synchronisent (join) et se terminent, ne laissant que le thread maître.
Les trois principaux composants d'OpenMP sont :

- Les directives du compilateur
- Les routines de la bibliothèque d'exécution
- Les variables d'environnement

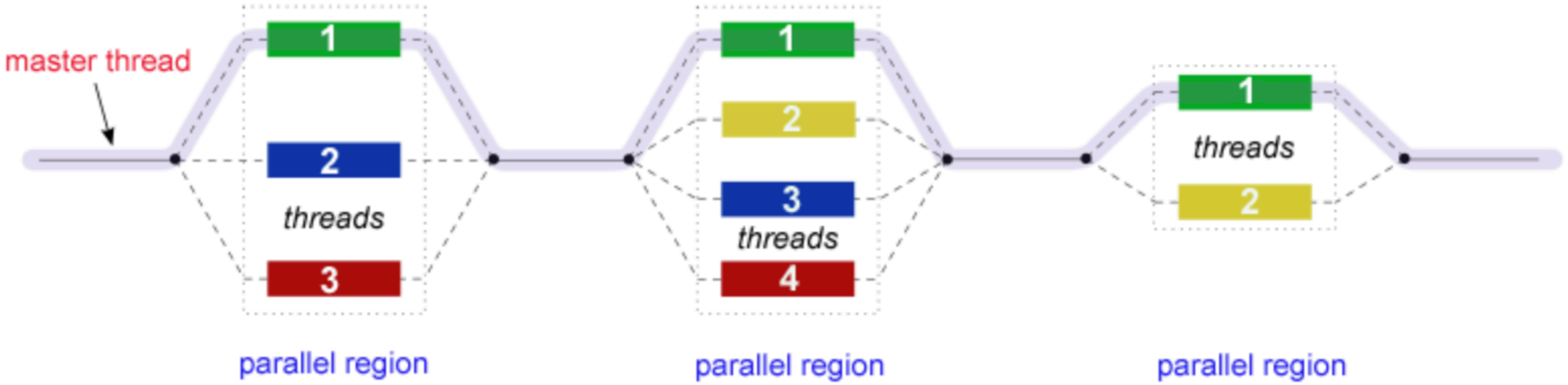
Les directives OpenMP apparaissent comme des commentaires dans votre code source et sont ignorées par les compilateurs sauf si vous leur indiquez le contraire. Les directives ont la syntaxe suivante : sentinelle nom-de-directive [clause, ...].

```
#pragma omp parallel default(shared) private(beta,pi)
```

OpenMP fournit plusieurs variables d'environnement pour contrôler l'exécution du code parallèle au moment de l'exécution.

```
export OMP_NUM_THREADS=8
```

Implémentation OpenMB



Implémentation OpenMB

- copyin: Permet aux threads d'accéder à la valeur du thread maître pour une variable threadprivate.
- copyprivate: Spécifie qu'une ou plusieurs variables doivent être partagées entre tous les threads.
- default: Spécifie le comportement des variables non définies dans une région parallèle.
- firstprivate: Spécifie que chaque thread doit avoir sa propre instance d'une variable, et que la variable doit être initialisée avec la valeur de la variable telle qu'elle existe avant la construction parallèle.
- if: Spécifie si une boucle doit être exécutée en parallèle ou en série.
- lastprivate: Spécifie que la version de la variable dans le contexte englobant est égale à la version privée du thread qui exécute l'itération finale (construction for-loop) ou la dernière section (#pragma sections).
- nowait: Supprime la barrière implicite dans une directive.
- num_threads: Définit le nombre de threads dans une équipe de threads.
- ordered: Requis sur une instruction de boucle parallèle for si une directive ordered doit être utilisée dans la boucle.
- private: Spécifie que chaque thread doit avoir sa propre instance d'une variable.
- reduction: Spécifie qu'une ou plusieurs variables privées à chaque thread font l'objet d'une opération de réduction à la fin de la région parallèle.
- schedule S'applique à la directive for.
- shared Spécifie qu'une ou plusieurs variables doivent être partagées entre tous les threads.

MPI - implémentation

1. Initialisation MPI (MPI_Init)

C: `MPI_Init(&argc, &argv);`
Fortran: `MPI_INIT(ierr);`

L'initialisation de MPI est la première étape de tout programme MPI. `MPI_Init` doit être appelé avant toute autre routine MPI. Cette fonction initialise l'environnement MPI, en prenant en arguments les paramètres de la ligne de commande (le cas échéant). Cela permet à MPI de configurer l'environnement de communication.

2. Taille du Communicateur (MPI_Comm_size)

C: `MPI_Comm_size(MPI_Comm comm, int *size);`
Fortran: `MPI_COMM_SIZE(comm, size, ierr);`

La fonction `MPI_Comm_size` détermine le nombre total de processus impliqués dans un communicateur donné. `comm` est le communicateur dont on veut connaître la taille, et `size` est un pointeur vers un entier où sera stocké le nombre de processus. Cette information est cruciale pour les tâches parallèles, car elle permet de savoir combien de processus sont disponibles pour le calcul.

3. Rang du processus (MPI_Comm_rank)

C: `MPI_Comm_rank(MPI_Comm comm, int *rank);`
Fortran: `MPI_COMM_RANK(comm, rank, ierr);`

La fonction `MPI_Comm_rank` détermine le rang du processus appelant dans le communicateur. Le rang est un identifiant unique (de 0 à p-1, où p est la taille du communicateur) pour chaque processus dans un communicateur. Cette identification permet de distribuer les tâches et de coordonner la communication entre les processus.

MPI - implémentation

4. Abandon des processus MPI (MPI_Abort)

C: `MPI_Abort(MPI_Comm comm, int errorcode);`

Fortran: `MPI_ABORT(comm, errorcode, ierr);`

`MPI_Abort` est utilisé pour terminer tous les processus MPI associés à un communicateur. Cette fonction est souvent appelée en cas d'erreur critique. `comm` est le communicateur dont les processus doivent être arrêtés, et `errorcode` est un code d'erreur spécifiant la raison de l'abandon.

5. Finalisation MPI (MPI_Finalize)

C: `MPI_Finalize();`

Fortran: `MPI_FINALIZE(ierr);`

La fonction `MPI_Finalize` termine l'environnement d'exécution MPI. Elle doit être la dernière routine MPI appelée dans un programme MPI. Une fois que `MPI_Finalize` a été appelée, aucune autre fonction MPI ne peut être utilisée. Cette fonction nettoie l'environnement MPI et libère les ressources allouées.

MPI - implémentation

6. Exécution des tâches MPI sous OpenMPI et systèmes batch

Les tâches MPI sous OpenMPI sont exécutées en utilisant la commande `mpirun`. Voici quelques points détaillés :

1. **Commande de base**: `mpirun` est utilisé pour lancer des programmes MPI. La syntaxe de base est `mpirun -np <num_processes> <program>`, où `<num_processes>` est le nombre de processus que vous souhaitez lancer et `<program>` est le programme MPI à exécuter.
2. **Détection automatique**: OpenMPI détecte automatiquement lorsqu'il est exécuté dans un environnement batch (comme un gestionnaire de ressources) et assigne les processus aux ressources appropriées.
3. **Placement des processus**: La politique de placement des processus peut être modifiée en utilisant divers arguments de `mpirun`, comme `--map-by` pour spécifier comment les processus sont mappés sur les nœuds et `--bind-to` pour définir comment les processus sont liés aux ressources matérielles (comme les cœurs de CPU).
4. **Options avancées**: Il existe de nombreuses autres options pour contrôler l'exécution des programmes MPI, comme `--hostfile` pour spécifier un fichier d'hôtes ou `--oversubscribe` pour permettre à plus de processus de s'exécuter que de cœurs disponibles.

Exercice MPI Échange de messages Ping-Pong avec MPI

Dans cet exercice, vous allez compléter un programme MPI pour réaliser un échange de messages de type “ping-pong” entre deux processus. Le but est d’envoyer un message entre deux processus jusqu’à ce qu’un certain nombre de “ping-pong” soit atteint.

Container as a Service (CaaS)

Container as a Service (CaaS) est une offre de service cloud permettant aux développeurs de gérer, déployer et exécuter des applications conteneurisées sans avoir à gérer l'infrastructure sous-jacente.

- **Isolation des applications** : Chaque conteneur fonctionne de manière isolée, ce qui réduit les conflits entre les applications et les dépendances.
- **Portabilité** : Les conteneurs peuvent être déplacés facilement entre les environnements de développement, de test et de production.
- **Gestion simplifiée** : Les fournisseurs de CaaS gèrent l'orchestration des conteneurs, la mise à l'échelle automatique, le réseau et la sécurité.

Exemple de services CaaS :

- **Google Kubernetes Engine (GKE)**
- **Amazon Elastic Kubernetes Service (EKS)**
- **Azure Kubernetes Service (AKS)**

Kubernetes (K8S)

Kubernetes (K8S) est une plateforme d'orchestration de conteneurs qui automatise le déploiement, la gestion et la mise à l'échelle des applications conteneurisées.

- **Orchestration avancée** : Kubernetes gère l'ordonnancement et le placement des conteneurs sur les nœuds du cluster pour optimiser l'utilisation des ressources.
- **Mise à l'échelle automatique** : Kubernetes peut automatiquement ajuster le nombre de réplicas d'une application en fonction de la charge de travail.
- **Gestion des états désirés** : Kubernetes assure que l'état des applications correspond à l'état souhaité défini par les fichiers de configuration.
- **Services de réseau** : Kubernetes fournit des services de mise en réseau pour la communication entre les conteneurs et l'exposition des applications au monde extérieur.

Knative

Knative est une extension de Kubernetes conçue pour faciliter le déploiement et la gestion des applications serverless et des conteneurs.

- **Autoscaling** : Knative peut automatiquement mettre à l'échelle les instances de conteneurs en fonction de la demande.
- **Gestion des événements** : Knative permet de déclencher des fonctions serverless en réponse à des événements, ce qui est utile pour des applications réactives.
- **Déploiement simplifié** : Knative simplifie le processus de déploiement en offrant des abstractions pour gérer le cycle de vie des conteneurs.

Azure Batch et AWS Batch

Azure Batch et **AWS Batch** sont des services de traitement par lots qui permettent de gérer et d'exécuter des travaux de calcul à grande échelle.

- **Planification des tâches** : Ces services permettent de définir des travaux à exécuter, de les décomposer en tâches plus petites et de les distribuer sur plusieurs instances de calcul.
- **Évolutivité** : Ils peuvent automatiquement ajuster les ressources en fonction du volume de travail, assurant une utilisation optimale des ressources.
- **Intégration facile** : Azure Batch et AWS Batch s'intègrent facilement avec d'autres services cloud pour le stockage, la gestion des données et la surveillance.

AWS ParallelCluster est un outil open-source qui simplifie le déploiement et la gestion de clusters HPC sur AWS.

- **Configuration par code** : Les clusters peuvent être définis et configurés via des fichiers de configuration YAML, ce qui facilite la reproductibilité et l'automatisation.
- **Support multi-instance** : ParallelCluster prend en charge différents types d'instances AWS pour s'adapter aux besoins spécifiques des charges de travail.
- **Intégration avec AWS** : Il s'intègre avec d'autres services AWS, tels que Amazon FSx pour le stockage haute performance et AWS Batch pour la gestion des travaux.

GCP Cloud HPC Toolkit

GCP Cloud HPC Toolkit est une collection d'outils et de services sur Google Cloud Platform conçus pour les charges de travail HPC.

- **Gestion des clusters** : Le toolkit fournit des outils pour créer et gérer des clusters HPC avec des configurations optimisées pour différentes applications.
- **Provisionnement des ressources** : Il offre des fonctionnalités pour provisionner dynamiquement des ressources de calcul en fonction des besoins.
- **Optimisation des performances** : Des outils pour surveiller et optimiser les performances des applications HPC, y compris le suivi des ressources et l'ajustement des configurations.

Azure CycleCloud

Azure CycleCloud est un service Azure qui facilite la création, la gestion et l'orchestration des clusters HPC dans le cloud.

- **Déploiement automatisé** : CycleCloud permet de déployer des clusters HPC avec une configuration automatisée et des scripts personnalisés.
- **Gestion des files d'attente** : Il offre des capacités de gestion des files d'attente de travaux, facilitant la soumission et le suivi des tâches de calcul.
- **Surveillance et optimisation** : CycleCloud fournit des outils pour surveiller les performances des clusters et optimiser l'utilisation des ressources pour les charges de travail HPC.

Apptainer

Apptainer (anciennement connu sous le nom de Singularity) est une plateforme de conteneurisation conçue spécifiquement pour répondre aux besoins de la communauté scientifique et des utilisateurs de HPC (High Performance Computing). Contrairement à d'autres solutions de conteneurisation comme Docker, Apptainer se concentre sur la portabilité, la reproductibilité et la sécurité dans des environnements multi-utilisateurs, souvent rencontrés dans les supercalculateurs et les clusters de calculs scientifiques.

- **Sécurité** : Exécution en mode utilisateur sans nécessiter de privilèges root.
- **Portabilité** : Les images conteneurisées sont des fichiers uniques (.sif) qui peuvent être facilement déplacés et partagés.
- **Reproductibilité** : Garantit que les environnements logiciels restent cohérents sur différents systèmes.
- **Intégration HPC** : Conçu pour fonctionner harmonieusement avec les gestionnaires de tâches de clusters comme Slurm et OpenPBS.

Apptainer

1. Exemple de fichier de définition pour une application MPI :

```
BootStrap: docker
From: ubuntu:20.04

%post
    apt-get update && apt-get install -y mpich

%environment
    export PATH=/usr/local/bin:$PATH

%runscript
    exec "$@"
```

2. Construire l'image Apptainer :

```
apptainer build mpi_image.sif mpi_definition.def
```

Apptainer

3. Écrire un script de soumission OpenPBS pour MPI :

```
#!/bin/bash
#PBS -N mpi_job
#PBS -l nodes=4:ppn=8
#PBS -l walltime=02:00:00
#PBS -j oe

cd $PBS_O_WORKDIR

# Charger le module MPI si nécessaire
module load mpich

# Exécuter l'application MPI dans le conteneur Apptainer
mpirun -np 32 apptainer exec mpi_image.sif /path/to/mpi_application
```

4. Soumettre le job :

```
qsub mpi_job.pbs
```

Exercice : Création et soumission d'un conteneur Apptainer avec OpenPBS et Slurm

1. Créer un conteneur Apptainer à partir d'un fichier de définition.
2. Soumettre un job exécutant un programme C parallèle avec OpenMP et MPI dans un conteneur Apptainer sur un cluster utilisant OpenPBS.
3. Soumettre un job similaire sur un cluster utilisant Slurm.

TP : Déploiement, Benchmark et Optimisation d'un modèle HPC pour la dynamique des fluides utilisant MPI et OpenMP avec Apptainer sur OpenPBS et Slurm

1. Créer un conteneur Apptainer pour une application de dynamique des fluides.
2. Compiler et exécuter un programme C utilisant MPI et OpenMP dans un conteneur Apptainer.
3. Soumettre un job sur un cluster utilisant OpenPBS.
4. Soumettre un job similaire sur un cluster utilisant Slurm.
5. Débuter les performances de l'application pour évaluer son efficacité.
6. Identifier les points de goulot d'étranglement et proposer des optimisations.

Étapes du TP

1. **Écrire le programme en C** :
   - Écrire un programme en C qui résout les équations de Navier-Stokes en 2D en utilisant MPI pour la communication entre processus et OpenMP pour la parallélisation au sein de chaque processus.
2. **Créer un fichier de définition pour Apptainer** :
   - Utiliser le format de fichier de définition Apptainer pour spécifier les dépendances et les configurations nécessaires pour compiler et exécuter le programme C.
3. **Construire le conteneur Apptainer** :
   - Utiliser la commande appropriée pour créer une image de conteneur Apptainer à partir du fichier de définition.
4. **Écrire un script de soumission pour OpenPBS** :
   - Créer un script de soumission PBS qui charge le conteneur Apptainer et exécute le programme C en parallèle sur un cluster utilisant OpenPBS.

5. **Écrire un script de soumission pour Slurm** :
   - Créer un script de soumission Slurm qui charge le conteneur Apptainer et exécute le programme C en parallèle sur un cluster utilisant Slurm.
6. **Soumettre les jobs sur OpenPBS et Slurm** :
   - Utiliser les scripts de soumission pour exécuter le programme sur les clusters OpenPBS et Slurm.
   - Collecter les temps d'exécution affichés dans les fichiers de sortie.
7. **Analyser les performances de l'application** :
   - Comparer les temps d'exécution pour différentes configurations de nœuds et de cœurs.
   - Identifier les fonctions où le programme passe le plus de temps.
   - Vérifier la balance de charge MPI et l'utilisation des threads OpenMP.
8. **Identifier les points de goulot d'étranglement et proposer des optimisations** :
   - Examiner les parties du code les plus consommatrices en temps ou en ressources.
   - Proposer des optimisations spécifiques pour améliorer les performances, telles que l'optimisation des boucles, la réduction des sections critiques, et l'utilisation efficace de la mémoire cache.
   - Valider les optimisations pour s'assurer qu'elles n'introduisent pas de bugs et améliorent effectivement les performances.