Chapitre 2

Terminologie des architectures matérielles et logicielles distribuées

Ce chapitre est un rappel de certains concepts traditionnels d'architecture informatique, et une présentation d'autres spécifiques aux systèmes distribués. L'objectif premier est de présenter et de différencier les concepts matériels (cœurs, nœuds, cluster, clouds, réseaux, Bw, latence...) et les concepts logiciels (threads, processus, middleware, client/serveur, latence applicative...). Le second objectif est d'introduire la notion de topologie virtuelle, et le problème de son déploiement (ou *mapping*) sur l'architecture matérielle réelle, avec notamment des préoccupations de performances, de tolérance aux pannes et de passage à l'échelle.

2.1 Composants matériels

2.1.1 Le processeur et ses unités de calcul

Un processeur standard est aujourd'hui composé de multiples entités, notamment : des cœurs physiques fonctionnant en parallèle, une hiérarchie de mémoires caches internes, et des unités de calcul vectoriel dans chaque cœur.

En fait un cœur est un petit processeur autonome, qui peut exécuter un code différent de ceux des autres cœurs, à une fréquence d'horloge éventuellement différente. De plus, chaque cœur contient et exploite une petite mémoire cache locale, et des unités de calcul vectoriel (typiquement 4 ou 8), qui lui permettent d'exécuter une même instruction en parallèle sur des données différentes. Voir figure 2.1.

Un processeur est aujourd'hui un système avec beaucoup de composants et de potentiel, mais l'utiliser au maximum de ses capacités est devenu extrêmement complexe. Il faut notamment :

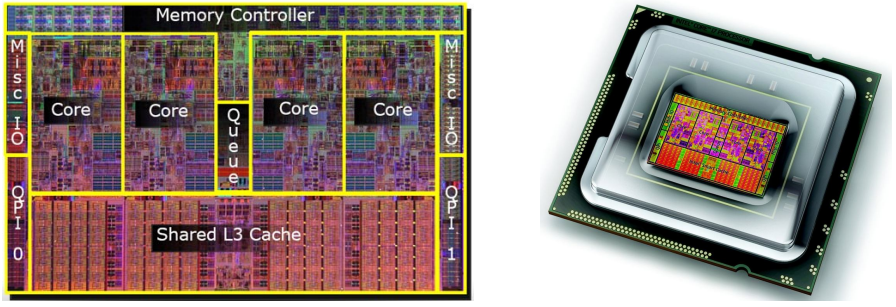

FIGURE 2.1 – Exemples de cœurs de calcul et de processeur multi-cœurs (Intel)

– lui faire exécuter à tout moment suffisamment de tâches pour occuper tous ses cœurs,
– organiser les données et leur accès de manière à ce que chaque cœur exploite bien sa mémoire cache, et fasse le moins possible de défaut de cache (aller chercher en mémoire RAM des données non anticipées et risquer de n'avoir rien à faire en attendant),
– organiser les calculs et les données de manière à réaliser des opérations vectorielles au sein de chaque cœur.

Un objectif des développeurs HPC est de maximiser le taux d'utilisation de chaque processeur, plutôt que d'en utiliser partiellement un plus grand nombre. Cet objectif n'était pas initialement présent dans les applications d'analyse de données, mais tend à le devenir dans les bibliothèques de *machine learning* intensif.

Un peu de détail : *les mémoires "cache" sont des mémoires rapides, mais plus gourmandes en énergie et en surface de silicium, et sont plus onéreuses que les mémoires classiques. Elles sont utilisées par petites quantités pour stocker près des unités de calcul les données les plus récemment utilisées (et susceptibles d'être ré-utilisées d'ici peu) ou dont on prévoit l'utilisation très prochaine. Les processeurs modernes disposent d'une hiérarchie de mémoires caches de plus en plus petites mais de plus en plus rapides entre la RAM et leurs unités de calcul. Chaque cœur possède sa propre mémoire cache de niveau 1 (le niveau le plus proche des unités de calcul), mais les différents cœurs partagent les mémoires cache de niveau supérieur, et peuvent donc se gêner mutuellement dans l'exploitation globale du cache. Une bonne exploitation de la mémoire cache reste cependant la première clé de l'obtention de performances au sein d'un processeur.*

Les unités de calcul vectoriel des CPU deviennent de plus en plus puissantes, et doivent absolument être exploitées lors de calculs sur des tableaux de valeurs. Il y a toutefois de nombreuses contraintes sur l'organisation des calculs et le stockage des données pour que l'utilisation des unités vectorielles soit possible et efficace. La programmation de ces unités reste donc fastidieuse, et tend à être déléguée au compilateur. Mais il faut apprendre à écrire des codes de haut niveau favorables à la vectorisation, et à ajouter dans le code source des directives de compilation guidant le compilateur.

2.1.2 Le nœud de calcul et sa mémoire partagée

Un nœud de calcul est un ordinateur, un PC, avec son/ses processeurs, sa mémoire RAM, son/ses accès réseau, et éventuellement un/des disques locaux. Un nœud peut ainsi avoir un ou plusieurs processeur (chacun ayant plusieurs cœurs) qui partagent sa mémoire : sauf cas original, un *nœud* est aujourd'hui une machine parallèle à mémoire partagée. Voir figure 2.2.

On peut cependant classer les nœuds de calcul en deux grandes catégories, selon que les accès mémoire se font toujours avec le même délai, ou que chaque processeur est plus proche de certains bancs mémoire. On parle alors d'architecture à accès mémoire non uniforme ou NUMA (*non*

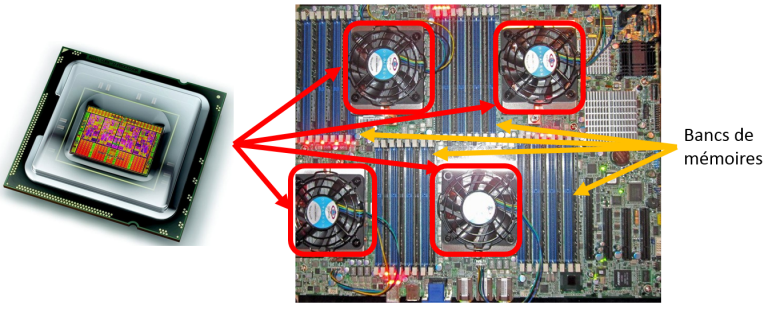

FIGURE 2.2 – Exemple de carte mère équipée de 4 processeurs multi-cœurs

FIGURE 2.3 – Exemple de clusters de PC.

uniform memory architecture). La tendance actuelle est aux nœuds NUMA, parfois à partir de 2 processeurs seulement. Ces détails sont invisibles au développeur de haut niveau (utilisant des bibliothèques parallèles), ou à celui qui ne cherche pas à optimiser la performance. En revanche, le développeur HPC optimisera l'allocation de ses données pour que chaque processeur accède par la suite à des données allouées par lui et donc *proches de lui*.

Un peu de détail : *Les nœuds avec peu de processeurs supportent que chaque processeur accède à n'importe quelle partie de la mémoire avec le même délais (la même* latence*, voir section 2.1.3) et le même débit (la même bande passante). On parle alors de machine à mémoire partagée uniforme et parfois d'architecture SMP (Symetric Multi-Processor : tous les processeurs sont équivalents). En revanche, quand le nombre de processeurs augmente, il n'est plus possible de maintenir efficacement une mémoire uniforme. Un petit réseau d'interconnexion relie les différents processeurs aux divers bancs mémoires, et chaque processeur se retrouve plus proche de certains bancs. On est alors en présence d'une architecture NUMA. Il s'agit bien d'une mémoire partagée entre tous les processeurs, mais avec des temps d'accès hétérogènes.*

2.1.3 Le cluster de calcul et son réseau d'interconnexion

Conceptuellement un cluster de calcul n'est qu'un ensemble de nœuds de calculs, installés dans des baies et reliés par un réseau local d'interconnexion (*l'interconnect*), voir figure 2.3. Il existe des clusters avec des nœuds très rapides et beaucoup de cœurs, avec beaucoup de mémoire et beaucoup d'espace disque par nœud,... mais il existe aussi des clusters avec des nœuds ressemblant à des serveurs standards. Tout dépend du type de calculs que l'on vise prioritairement et pour lequel on choisit un matériel le plus adapté possible. Mais dans la plupart des cas, le réseau d'interconnexion apparaît comme le composant clé du cluster.

Deux métriques sont particulièrement importantes pour évaluer la qualité d'un interconnect : la *bande passante* entre deux nœuds de calculs, mais aussi la *latence* entre deux nœuds. Il s'agit du temps écoulé entre le début de l'envoie d'un message et le début de son arrivée sur le nœud destination. Aujourd'hui un nœud peut faire énormément de calculs pendant qu'un petit message lui est routé sur l'interconnect. S'il est régulièrement bloqué en attente d'une donnée d'un autre nœud (n'ayant plus d'autres calculs à faire), alors beaucoup de temps CPU sera gâché et la parallélisation sera inefficace.

Un peu de détail : *Un interconnect bon marché, comme un réseau Ethernet à 1Gbit/s, ne permet pas de faire du calcul parallèle efficace avec les nœuds d'aujourd'hui. Les échanges de données et de résultats intermédiaires entre les nœuds prendraient trop de temps. Les nœuds traiteraient les données et produiraient des résultats intermédiaires plus vite qu'ils ne pourraient les échanger. On obtiendrait plutôt un ensemble de PC destinés à exécuter chacun des programmes indépendant. A l'opposé, un interconnect performant, comme un réseau Infiniband, constitue une partie non né-*

gligeable du coût du cluster. Mais il permet réellement de répartir différentes parties d'un même calcul sur différents PC, d'échanger régulièrement des données et résultats intermédiaires sans trop pénaliser le temps d'exécution, et d'accélérer des applications complexes. Evidemment, les temps d'accès à des données distantes à travers un réseau d'interconnexion restent longs vis-à-vis des temps d'accès à des données locales en RAM, et on cherche toujours à minimiser les communications inter-nœuds dans les algorithmes distribués. Enfin, les technologies des réseaux locaux classiques basées sur une pile TCP/IP sont inadaptées pour être des interconnects performants sur cluster de calculs, elles présentent notamment des temps de latences trop élevés. On leur préfère par exemple des réseaux au standard Infiniband, possédant à la fois des bandes passantes élevées et de faibles latences, mais plus chers qu'un simple réseau TCP/IP.

2.1.4 Les accélérateurs matériels

Les accélérateurs matériels sont en dehors du programme de ce cours, néanmoins ils sont très utilisés pour paralléliser des calculs de *machine learning*. Des algorithmes de *deep learning* ont notamment été portés et parallélisés sur accélérateurs, et se sont révélés très performants sur GPU.

La plupart du temps un *accélérateur matériel* se présente comme une carte fille installée dans un PC (voir figure 2.4), avec sa propre mémoire, ses *nombreuses petites* unités de calcul, et une architecture nécessitant une programmation massivement parallèle. On peut ainsi percevoir un accélérateur comme un co-processeur de calcul scientifique. Mais depuis peu, certains accélérateurs peuvent devenir des processeurs à part entière sur la carte mère, particulièrement efficaces pour traiter des problèmes contenant un fort taux de parallélisme.

Dans tous les cas, si l'on dispose de PCs équipés d'accélérateurs et de bibliothèques de calculs adaptées, alors l'utilisation des accélérateurs peut être totalement transparente pour un *data scientist*, et lui fournir de la performance bon marché. En revanche, s'il est nécessaire de développer un code de calcul spécifique sur accélérateur, cela reste du domaine de l'expert de ce type de matériel.

Un peu de détail : *Les GPU (processeurs graphiques) sont devenus de très bons co-processeurs vectoriels. Ils demandent une programmation particulière et sont adaptés surtout à des programmes réalisant de nombreux calculs identiques sur des tableaux de données. Sur ce type de problèmes ils peuvent être très rapides et présenter un très bon rapport performance/prix. A l'opposé, les processeurs Xeon-phi d'Intel étaient bâtis autour d'un ensemble d'environ* 80 *cœurs* $x86$*, similaires à ceux des Xeon traditionnels, bien que de fréquence plus faible, et demandant une programmation proche de celle d'un Xeon multi-cœurs. Toutefois, leur programmation apparut plus complexe dès lors que l'on cherchait à obtenir une performance élevée, et en 2018 Intel stoppa cette gamme d'accélérateurs. Enfin, les composants FPGA peuvent devenir d'excellents co-processeurs, mais restent plus complexes à programmer (on programme la configuration d'une architecture/d'un*

Intel Xeon-phi 80 cores (« Knight Landing ») | GPU NVIDIA (« TESLA ») | Intel FPGA accelerator (« PSG Arria »)

FIGURE 2.4 – Exemples d'accélérateurs matériels à insérer dans des PC, ou devenus processeur principal

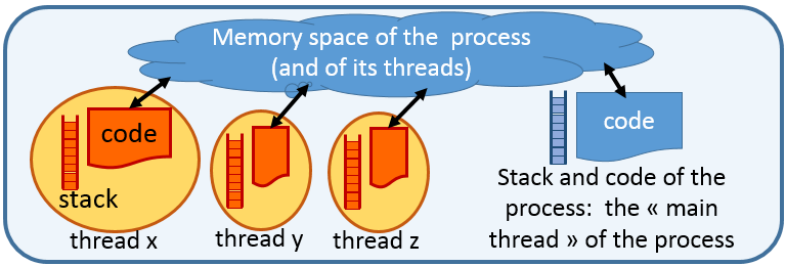

FIGURE 2.5 – Un processus avec son espace mémoire et ses threads qui le partagent

chemin de données adapté au problème).

2.2 Composants logiciels

2.2.1 Les *threads* et processus

A un niveau algorithmique un peu abstrait on conçoit des algorithmes qui créent et synchronisent des *tâches* ayant chacune son activité propre. Puis, dans une étape plus avancée du développement on implante ces tâches sous forme de processus ou de threads.

Les processus sont un concept ancien. Un programme qui s'exécute, comme un éditeur de texte ou un compilateur, est un processus. Il possède son espace mémoire propre, où il stocke ses données et sa pile (qu'il utilise notamment pour enchaîner ses appels de fonctions). Deux processus peuvent partager de la mémoire en demandant explicitement au système d'exploitation de leur allouer un espace mémoire partagée. Par défaut les espaces mémoires des processus ne sont pas partagés.

Les threads sont apparus plus récemment que les processus. Chaque thread a bien une pile propre, mais il s'exécute dans l'espace mémoire de son processus hôte, voir figure 2.5. En fait, un processus peut créer plusieurs threads, et par défaut tous les threads de ce processus partagent son espace mémoire et ses données. Pour le système d'exploitation, passer d'un thread à un autre est plus rapide que de changer de processus (l'espace d'adressage restant notamment le même) et l'on désigne un thread comme étant un *processus léger*.

Aujourd'hui, pour paralléliser une application et répartir ses calculs sur plusieurs cœurs d'un même processeur, on préfère créer des threads au sein d'un processus, que de créer plusieurs processus. Pour occuper tous les cœurs d'un nœud multi-processeurs on peut aussi utiliser un seul processus et de nombreux threads, mais à cause des architectures NUMA (voir section 2.1.2) on préfère souvent déployer un processus par processeur et des threads seulement à l'intérieur de chaque processeur (pour des raisons d'efficacité des accès mémoire)... Actuellement, la plupart des langages de programmation permettent de créer des threads, et les systèmes d'exploitation peuvent facilement répartir et ordonnancer des milliers de threads (ou plus) sur des dizaines de cœurs.

2.2.2 Les outils élémentaires de synchronisation de tâches

La synchronisation des tâches est indispensable dans un programme parallèle au sein de toute machine à mémoire partagée (comme un simple PC multi-cœurs). Plusieurs outils permettent d'implanter des protocoles de synchronisation (ou principes algorithmiques) souvent dérivés de protocoles génériques bien connus. Mais dans le cadre d'une programmation parallèle visant à exécuter des tâches sur plusieurs cœurs, dans le but de réduire le temps d'exécution, il est important d'implanter des protocoles de synchronisation qui ne re-séquentialisent pas systématiquement les

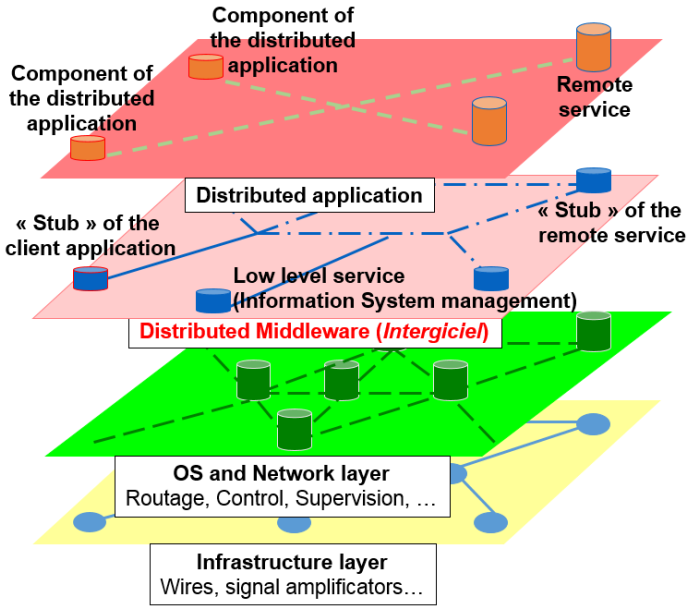

FIGURE 2.6 – Positionnement d'un middleware (ou intergiciel)

tâches. **La *mutex* et le *moniteur* (sorte de *mutex* sur des portions de codes) sont notamment à éviter.** Leur utilisation conduit souvent à des exécutions quasi-séquentielles des applications multi-threads. Les protocoles/algorithmes de *producteurs/consommateurs*, *lecteurs/rédacteurs*, *cohortes* et *barrières de synchronisation* sont beaucoup plus pertinents en calcul parallèle.

Remarque : curieusement, les *sémaphores* sont peu appréciés des développeurs applicatifs, alors qu'avec un peu d'entraînement ils permettent de réaliser des protocoles de synchronisation élégants et rapides.

Un peu de détail : *En ce qui concerne les outils, au niveau assembleur on trouve le* test-and-set*, et au niveau des appels systèmes et du systèmes d'exploitation on trouve notamment les* sémaphores *et* verrous*. Ces outils permettent d'implanter des protocoles de synchronisation comme* l'exclusion mutuelle *(ou* mutex*), les* producteurs/consommateurs*, les* lecteurs/rédacteurs*, la* cohorte*... qui sont étudiés dans d'autres cours que celui-ci. Notons que les* variables conditionnelles *associées à des mutex ont remplacé les sémaphores dans certains environnement (nous n'entrerons pas dans les détails ici). Au niveau de la sémantique des langages de programmation et des bibliothèques de parallélisme on trouve les* barrières de synchronisation *et les* moniteurs*, qui peuvent être perçus à la fois comme des outils et des protocoles de synchronisation.*

2.2.3 Les *middlewares* et *frameworks*

Un *framework* peut se percevoir comme un environnement de développement visant à simplifier le développement de certains types d'applications ou de composants logiciels. Il est plus générique qu'une simple bibliothèque, mais est conçu pour orienter les développements dans certaines voies. Un *framework* se perçoit depuis un langage de développement. Un *middleware* peut se voir plutôt comme un environnement d'exécution qui cherche à simplifier la communication entre des composants logiciels, pour faciliter leur assemblage et donc la construction d'applications complexes (et souvent distribuées). Le *middleware* se situe au dessus du système d'exploitation et des couches réseaux, voir figure 2.6.

Il est fréquent de développer des applications distribuées dans un environnement comprenant un *framework* et un *middleware* compatibles. Si l'on considère l'environnement *Hadoop* du Big

Data, son *framework* permet de développer facilement en Java des composants destinés à être insérés dans une chaîne logicielle réalisant un *Map-Reduce* (paradigme de programmation d'*Hadoop*). L'exécution de l'application distribuée obtenue s'appuiera ensuite sur un *middleware* comprenant notamment le système de fichiers distribués d'*Hadoop* (HDFS), et ses mécanismes de distribution et de gestion de tâches. Au final, le développeur applicatif ne percevra pas les détails de mise en oeuvre sur une ou plusieurs machines, ni les complications des mécanismes de tolérance aux pannes. En revanche, il sera limité à développer des applications respectant le paradigme *Map-Reduce*.

2.3 *Cloud* vs *cluster*

Les *clouds* et les *cluster* permettent tous deux d'allouer des PCs pour un temps donné, mais a priori pour des usages différents. Un *cluster* permet de disposer d'un ensemble de PCs performants, reliés par un réseau à haute performance (en latence et bande passante), sur lequel on pourra exécuter des tâches appartenant à une même application et échangeant de nombreuses données et résultats intermédiaires. Les *clusters* sont devenus les machines parallèles standard du HPC. A l'opposé, un *cloud* permet d'allouer des PCs, ou simplement des cœurs, mais habituellement aucune hypothèse ne peut être faite sur la proximité des ressources allouées ou sur la qualité de l'interconnect.

Allouer des ressources dans un *cloud* standard pour y stocker des données et exécuter un code HPC peut ainsi s'avérer désastreux en performances, mais être un très bon rapport qualité/prix pour stocker des photos et appliquer un traitement indépendant sur chacune... Il est également possible d'allouer des ressources dans un *cloud HPC*, dans quel cas on obtient en fait une tranche de cluster HPC ! Mais la location des ressources d'un *cloud HPC* est beaucoup plus chère que celle d'un *cloud* standard.

Pour des applications de Big Data qui suivraient un schéma de calcul Map-Reduce, où les tâches des phases de Map et de Reduce sont indépendantes, et où les communications entre les deux phases peuvent être partiellement recouvertes, un *cloud* standard peut présenter un très bon rapport qualité/prix. En revanche l'accélération d'algorithmes d'apprentissage (*machine learning*), sur des données filtrées et rassemblées préalablement par un pré-traitement en Map-Reduce, est habituellement plus indiquées sur des architectures HPC.

2.4 Problématique du déploiement ou *mapping*

Une application distribuée suit un algorithme qui exploite une *topologie distribuée virtuelle*. Par exemple, les différentes tâches de l'application distribuée communiquent selon un schéma en anneau mono- ou bi-directionnel, ou en étoile, ou en tore-2D ou 3D, ou en *Map-Reduce*... Cette topologie virtuelle est liée à l'algorithme implanté. Il faut néanmoins *déployer* cette topologie virtuelle sur la topologie réelle de l'architecture physique.

Tout d'abord il ne faut pas oublier que certaines parties d'une architecture distribuée peuvent être en panne de longue durée, ou en maintenance planifiée, ou être tombées en panne peu avant le lancement de l'application à déployer, et d'autres peuvent être déjà très chargées par d'autres tâches provenant d'autres applications. Il est donc important de sélectionner des ressources parmi une liste des *ressources disponibles* maintenue à jour en permanence, ce qui repose en général sur des mécanismes de monitoring par le système d'exploitation de chaque nœud et par une couche de middleware distribué qui fournit une vue d'ensemble sur toutes les ressources.

Ensuite il faut bien entendu qu'une tâche (processus ou thread) soit installée sur une ressource de calcul lui donnant accès aux données dont elle à besoin, et lui offrant une capacité de calcul

suffisante pour traiter ses données. Mais deux objectifs importants viennent compliquer le déploiement :

- La *recherche de la performance* : vitesse de calcul, vitesse d'accès en lecture et/ou écriture aux données, qui impose d'installer la tâche sur une ressource assez proche des données et assez puissante. Le déploiement doit donc tenir compte des besoins de la tâche et des caractéristiques des ressources disponibles pour choisir la ressource la plus adaptée à la tâche (voir section 2.4.1). Dans certains cas le déploiement s'applique à tous types de tâches mais est peu optimisé ou doit être aiguillé par l'utilisateur, dans d'autres cas (notamment en *Data Engineering* à large échelle) il est automatisé et suit des stratégies préétablies pour des applications d'analyse de données.
- La *mise en œuvre d'une tolérance aux pannes* car une ressource de calcul ou de données peut très bien tomber en panne au cours de l'exécution de la tâche qu'elle héberge, ce qui arrive d'autant plus fréquemment que les ressources utilisées par l'application complète le sont en grand nombre et pour une longue durée (voir section 2.4.2). Il existe différentes stratégies et mécanismes de tolérance aux pannes, selon la nature des applications à exécuter, selon la nature des ressources de données et de calculs disponibles, et selon le niveau de tolérance aux pannes que l'on souhaite atteindre. Mais il est classique que la solution adoptée impacte le déploiement de l'application distribuée.

S'ajoute ensuite deux autres objectifs plus complexes :

- Assurer une *capacité de passage à l'échelle* qui demande la conception d'une solution *extensible* tant au niveau de l'algorithme et de l'implantation de l'application, que du mécanisme de déploiement (voir section 2.4.3).
- Maintenir une *maîtrise du coût d'exécution* quand on souhaite augmenter la quantité de ressources allouées et la durée des traitements. Un bon outil de déploiement trouvera une solution pour satisfaire les besoins de l'application et des utilisateurs (tant qu'il y aura des ressources), mais le coût du traitement peut s'envoler si les utilisateurs n'ont pas conscience des ressources engagées (voir section 2.4.4).

Par expérience, la conception et la réalisation d'un déploiement satisfaisant tout ou partie de ses objectifs peut représenter jusqu'à 30% du temps de développement total.

2.4.1 Objectif de performance

En HPC la vitesse de calcul est l'objectif principal. Au niveau du déploiement cela revient souvent à chercher à équilibrer la charge de calcul des différents nœuds et à minimiser les temps de communications inter-nœuds (qui sont les plus coûteux). Ces optimisations dépendent étroitement de la nature des calculs parallèles de l'application. En amont, les codes sont censés être implantés pour maximiser une bonne utilisation des mémoires caches, et permettre à un cœur de calcul d'exécuter une tâche en minimisant ses accès à la mémoire globale, et donc en gênant un minimum les calculs des autres cœurs. Néanmoins, cet objectif n'est pas toujours atteint. Il est alors nécessaire d'identifier (souvent expérimentalement) le nombre optimal de tâches à exécuter simultanément sur un même nœud. Ce nombre peut s'avérer inférieur au nombre de cœurs disponibles si les quelques canaux d'accès à la mémoire sont trop fortement sollicités, ou si la quantité de mémoire cache est trop limitée pour l'ensemble des calculs exécutés par les cœurs.

En Big Data on peut retrouver les mêmes stratégies qu'en HPC dans le cas de parallélisations d'algorithmes d'apprentissage. Mais dans le cas de lectures et filtrages de gros volumes de fichiers, les opérations les plus coûteuses risquent d'être les opérations d'entrée/sorties. Le déploiement consistera alors à placer les tâches de traitement de manière à maximiser l'utilisation de la bande

passante des disques de stockage. Une stratégie par défaut existe en *Hadoop* pour le déploiement des tâches *map* (d'un algorithme *map-reduce*) qui lisent les données, mais elle peut toutefois être influencée par le développeur.

2.4.2 Objectif de tolérance aux pannes

Dès qu'un système distribué est un peu gros, on peut considérer que ses ressources ne sont jamais toutes disponibles en même temps. Certaines machines tombent en panne, certaines unités de stockage subissent des dommages et leurs données deviennent corrompues, certaines parties du réseau tombent en panne ou deviennent très chargées et lentes...

Imaginez que pour déployer votre application distribuée vous ayez absolument besoin que toutes les machines de TP du campus soient en état de marche et le restent pendant 24h. Pensez-vous que vous arriveriez à déployer votre application et à obtenir des résultats ? Seule une application et un déploiement tolérants aux pannes vous permettront d'y arriver.

En HPC on utilise souvent des unités de stockage dupliquées au niveau matériel dans une but de redondance (ex : disques associés en technologie RAID-1 ou RAID-5). Puis habituellement on pratique aussi des sauvegardes intermédiaires de l'état de l'application, permettant de redémarrer les calculs depuis la dernière sauvegarde en cas de dysfonctionnement de certaines ressources. Il s'agit de la stratégie de *checkpoint/restart*. Dans le domaine du Big Data, on stocke souvent les données de manière redondante sur des machines différentes, de façon à ce qu'elles soient toujours accessibles, et que l'on puisse toujours envoyer des tâches de traitement sur des machines stockant les données voulues. Un environnement comme *Hadoop* rend cet aspect de la tolérance aux pannes invisible au développeur, mais pas entièrement au responsable des ressources qui doit préciser le facteur de réplication des données (fonction de la fiabilité de son matériel et de la sensibilité des applications supportées).

Les diverses approches de la tolérance aux pannes sont adaptables, et composables, pour répondre au mieux aux besoins de l'application et de son usage. Mais dans tous les cas, elles affectent le déploiement de l'application distribuée, et le nombre de ressources physiques mobilisées.

2.4.3 Objectif de passage à l'échelle

Il est souvent plus difficile de déployer une même application sur 100 machines pour traiter 100To de données que sur 10 machines pour traiter 10To, mais il est surtout difficile de mettre au point une application et un déploiement *extensibles* pour traiter des données sans limite de taille connue (voir sections 3.3 et 3.6).

Par exemple, une application distribuée peut contenir une petite partie qui reste séquentielle et s'exécute dans la mémoire d'une seule machine, et qui devient insupportable par cette dernière quand le problème grossit. Une technologie de réseau d'interconnexion dans un cluster peut être très rapide, et répondre aux besoins pointus d'une application, mais ne pas exister pour un cluster 100 fois plus grand. Un mécanisme de déploiement peut être opérationnel sur un cluster, mais pas sur un cluster de clusters...

Exploiter automatiquement et sans limite plus de ressources pour traiter des problèmes plus importants, c'est-à-dire *passer à l'échelle*, est un problème très compliqué.

2.4.4 Objectif de maîtrise du coût d'exécution

Enfin, l'exécution d'une application distribuée peut être soumise à des contraintes de coût et de rentabilité. Vouloir gagner un peu plus en performance ou en tolérance aux pannes peut s'avérer très coûteux. Par exemple, allouer des cœurs de calcul très puissants et chers dans un *cloud*, alors

que c'est la bande passante des disques qui limitera les performances, sera un gaspillage d'argent. Allouer des nœuds de calculs équipés d'accélérateurs matériels (comme des GPU) coûtera plus cher et dissipera plus de puissance électrique. Si les calculs à effectuer n'y obtiennent qu'un faible gain, alors ce déploiement un peu plus rapide ne sera pas (du tout) rentable.

Un bon déploiement veillera à toujours cibler des ressources adaptées au problème, et à moindre coût.

Bibliographie

[1] G. Amdahl. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483–485, 1967.

[2] B. Azarmi. *Scalable Big Data Architecture*. Apress, 2016.

[3] R. Bruchez. *Les bases de données NoSQL et le Big Data*. Eyrolles, 2ème edition, 2016.

[4] R. Bruchez and M. Lutz. *Data science : fondamentaux et études de cas*. Eyrolles, 2015.

[5] B. Chapman, G. Jost, R. Van Der Pas, and D. Kuck. *Using OpenMP*. The MIT Press, 2008.

[6] K. Chodorow. *MongoDB, the Definitive Guide*. O'Reilly, 2ème edition, 2013.

[7] K. Dowd and Ch. Severance. *High Performance Computing*. O'Reilly, 2nd edition, 2008.

[8] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI*. The MIT Press, 1999.

[9] J.L. Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31 :532–533, 1988.

[10] H. Karau, A. Konwinski, P.Wendell, and M.Zaharia. *Learning Spark*. O'Reilly, 1st edition, 2015.

[11] H. Karau and R. Warren. *High Performance Spark*. O'Reilly, 1st edition, 2017.

[12] M. Kirk. *Thoughtful Machine Learning with Python*. O'Reilly, 2017.

[13] P. Lemberger, M. Batty, M. Morel, and J-L. Raffaelli. *Big Data et Machine Learning*. Dunod, 2015.

[14] D. Miner and A. Shook. *MapReduce Design Patterns*. O'Reilly, 2013.

[15] T. White. *Hadoop. The definitive Guide*. O'Reilly, 3rd edition, 2013.

[16] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M.J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets : A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, 2012.