Chapitre 3

Métriques de performance et de passage à l'échelle

La mesure de performance et l'inclusion de cette mesure dans la boucle de développement ne sont pas habituels en informatique, en dehors du HPC et d'une partie du Big Data. Ce sont pourtant des concepts essentiels pour aboutir à des codes *rapides* et *large échelle*. Ce chapitre introduit les principales métriques de performance et de passage à l'échelle : *speedup* (ou *scale in*), débit d'un flux de sortie (*throughput*), *size up* (ou *scale out*), courbes d'extensibilité (de passage à l'échelle), ... Elles sont utilisables pour le calcul massif mais aussi pour l'analyse de données.

3.1 Que faire avec plus de ressources informatiques ?

Consacrer à une même application plus de ressources informatiques, c'est-à-dire plus d'unités de calculs (cœurs, processeurs), plus de mémoire, plus d'espace de stockage (disques), peut se faire dans 3 directions comme illustré sur la figure 3.1 :

– On peut tout d'abord *exécuter concurremment l'application* sur les différents cœurs d'une machine, ou sur différentes machines. Un même utilisateur peut exécuter n_e fois l'application sur des jeux de données différents (ex : traiter des images différentes), ou sur un même jeu avec des paramètres de calculs différents (*investigation multi-paramètres*) et rechercher ensuite le meilleur résultat. Ou bien, n_u utilisateurs peuvent exécuter en même temps l'application sur leurs propres données. Enfin, n_u utilisateurs peuvent exécuter chacun n_e fois l'application pour faire des investigations multi-paramètres.

  Cette démarche est parfois qualifiée d'*embarrassingly parallel* (ou *happily parallel*). L'algorithmique de l'application élémentaire peut rester séquentielle alors qu'elle est exécutée plusieurs fois en pure concurrence. En conséquence, la mise en œuvre de cette démarche relève plutôt de la gestion de processus distribués (par un middleware adapté) que de la conception d'un algorithme parallèle.

– On peut aussi chercher à *accélérer l'exécution* d'une application en la *parallélisant* sur plusieurs ressources de calcul. On commence par identifier les parties de l'application qui sont les plus gourmandes en temps de calcul, et on répartit chacune de ces parties sur p unités de calculs (cœurs ou processeurs) qui travailleront en parallèle. On réduira ainsi les temps de calculs, mais il faut souvent faire communiquer les unités de calculs pour qu'elles s'échangent des données initiales ou des résultats intermédiaires.

  C'est une démarche d'algorithmique parallèle qui peut être très complexe. Au final, on espère obtenir une accélération (ou *speedup*) proche de p quand on utilise p ressources (voir

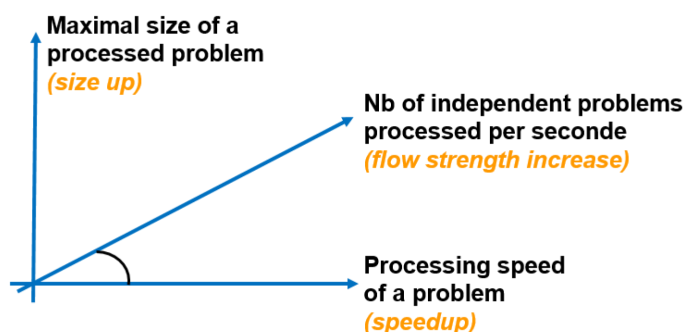

FIGURE 3.1 – Trois types de besoins menant à utiliser plus de ressources (de calcul et de stockage)

section 3.2), mais il existe toujours une limite à cette accélération quand on traite un même problème de taille fixée.

– On peut enfin chercher à *traiter un problème plus gros* qu'on ne pourrait le faire sur une seule machine. Lorsque l'on grossit le jeu de données, ou que l'on augmente la finesse des calculs (pour plus de précision), on peut manquer de mémoire sur une seule machine ou bien obtenir un temps de traitement prohibitif. On cherche alors à distribuer l'application sur plusieurs machines pour (1) être capable de traiter le problème (si on manquait de mémoire), et (2) maintenir un temps d'exécution constant quelle que soit la taille du problème.
Il s'agit d'une démarche de *size up*, qui demande aussi un important effort d'algorithmique parallèle, et qui n'est pas toujours couronnée de succès (voir section 3.3).

Ces trois façons d'utiliser un plus grand nombre de ressources informatiques sont détaillées dans les sections suivantes. Le *passage à l'échelle* complet requiert d'exploiter à la fois une démarche de *speedup* et de *size up*, et sera défini à la section 3.6.

3.2 Accélération d'un traitement (*speedup*)

3.2.1 Difficultés et contraintes sur le code applicatif

Pour accélérer l'exécution d'une application sur plusieurs ressources, il faut que son code soit parallélisé, c'est-à-dire que ses parties les plus gourmandes en temps de calcul soient découpées en sous-tâches pouvant s'exécuter en même temps (*en parallèle*). Mais deux problèmes majeurs peuvent empêcher un découpage efficace.

– Il arrive que des algorithmes soient *intrinsèquement* séquentiels : lorsque chaque petite étape de calcul dépend des résultats de la précédente. Dans ce cas, une révision de la modélisation est nécessaire pour aboutir à un autre jeu d'équations, mais il se peut que l'algorithme séquentiel initial possède ait une complexité optimale (la plus faible de tous les algorithmes traitant le problème). Il faut alors accepter de paralléliser un algorithme (un peu) moins bon pour profiter de plusieurs ressources de calcul, et pour au final être plus rapide que le code initial sur une seule ressource.

– Il se peut que le *surcoût de gestion* du parallélisme soit trop important. En effet, les sous-tâches d'un code parallèle ont besoin de communiquer et de se resynchroniser régulièrement. Ces opérations prennent du temps, et constituent un surcoût qui peut limiter ou même gâcher les gains obtenus par la répartition des calculs sur plusieurs ressources. En fait, la mise au point et l'implantation d'un schéma de communication et de synchronisation efficace demande une réelle expertise en calcul parallèle et distribué, qui reste inaccessible à beaucoup de développeurs.

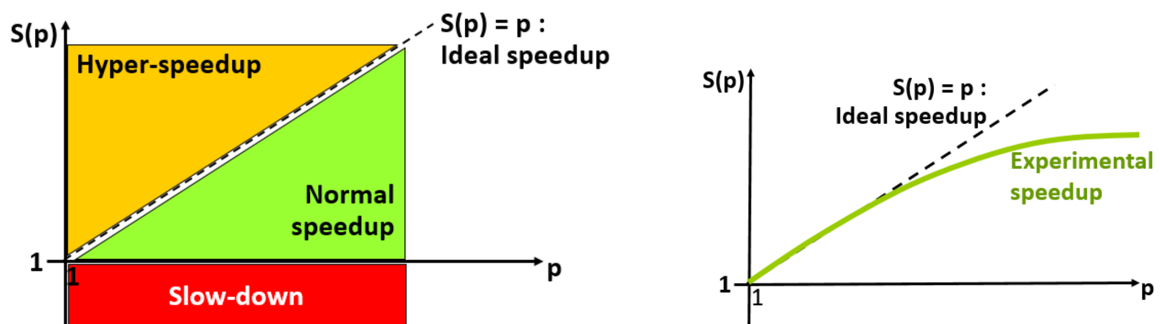

FIGURE 3.2 – Les trois types de *speedup* possibles (à gauche) et l'allure classique d'un *speedup* expérimental (à droite)

Le paradigme de programmation *Map-Reduce* est une tentative de réponse à ses problèmes pour le *Big Data*. Il propose un schéma de parallélisation à la fois adapté à de nombreux algorithmes d'analyse de données, et facilement exploitable par (presque) tous les développeurs (voir chapitres 5 et 6).

3.2.2 Métrique et courbe d'accélération

Le *speedup* est la métrique la plus connue pour caractériser la performance d'une parallélisation. Sa définition initiale est simple et permet de chiffrer le gain obtenu par un algorithme et une implantation parallèles :

$$S(p) = T(1)/T(p) \tag{3.1}$$

Où $T(1)$ et $T(p)$ sont respectivement les temps de traitement sur 1 et p *ressources*. Il peut s'agir de cœurs de calculs dans un processeur, de processeurs dans une machine à mémoire partagée, de nœuds (PCs) dans un cluster, de disques dans un système de stockage massif...

Le calcul du *speedup* suppose donc d'être capable de mesurer $T(1)$ en traitant le problème sur une seule ressource. Mais dans le cas du traitement de très gros problèmes, ceci n'est pas toujours possible. Par exemple, il peut ne pas y avoir assez d'espace mémoire ou d'espace disque associé à un seul PC pour y stocker toutes les données du problème. Ou bien le temps d'exécution sur une seule ressource peut être trop long pour que l'on puisse accepter l'expérience ! Il est toujours possible de prendre comme temps de référence $T(r_{min})$: le temps d'exécution sur le nombre minimal de ressources (r_{min}) à utiliser, et de calculer des accélérations relatives à $T(r_{min})$ (plutôt qu'à $T(1)$). Mais les analyses sont plus compliquées et la quantité de ressources r_{min} peut varier avec le type de ressources utilisées, rendant plus difficile la comparaison de courbes d'accélérations sur diverses architectures distribuées.

Quand on peut mesurer un *speedup* $S(p)$, on doit normalement obtenir : $1 < S(p) \leq p$, en espérant une valeur de $S(p)$ la plus proche possible de p (voir figure 3.2 gauche). Un *speedup* inférieur à 1 correspond en fait à un ralentissement, et donc normalement à une parallélisation ratée. Mais dans le cas du traitement de très gros problèmes, on peut être amené à raisonner différemment (voir la section 3.3 sur le *size up*). Un *speedup* $S(p)$ supérieur à p correspond à une *hyper-accélération*, et possède toujours une explication. Par exemple, en utilisant plus de PC on aura utilisé plus de cœurs de calcul, mais aussi plus de mémoire, plus de cache, plus de disques, et on aura finalement cumulé plusieurs gains.

L'allure classique d'une courbe de *speedup* d'un problème de taille fixée est celle de la figure 3.2 droite. Le début de la courbe est souvent proche de l'idéal ($S(p) = p$), puis s'en écarte, et

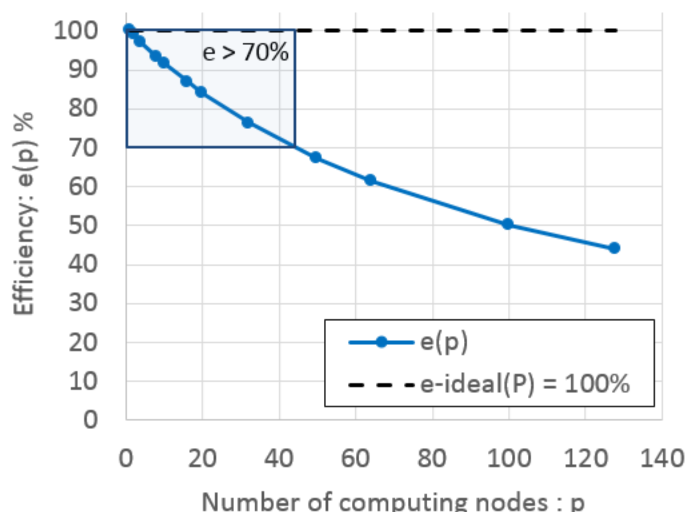

FIGURE 3.3 – Allure d'une courbe d'efficacité

tangente un plafond horizontal. La présence d'un plafond est en fait inévitable, même sur une machine idéale dès qu'une partie du problème n'est pas parallélisable (voir la section A.1 sur la loi de Amdhal), et aussi à cause du coût non négligeable des communications dans les machines réelles. Si l'on augmente encore le nombre de processeurs utilisés, on peut même obtenir une chute du *speedup* car les nouveaux processeurs ne seront presque pas utilisés et les temps de communications entre plus de processeurs deviendront plus importants. Il faut donc (généralement) utiliser un nombre de ressources de calcul raisonnable pour la taille de problème traité, ce que tente de mesurer l'efficacité (voir ci-après).

3.2.3 Métrique et courbe d'efficacité d'une parallélisation

L'efficacité est une métrique de rentabilité qui permet de chiffrer le taux de bonne utilisation des ressources utilisées dans une parallélisation :

$$e(p) = S(p)/p \tag{3.2}$$

Logiquement, on doit obtenir : $0\% < e(p) \leq 100\%$, en espérant une valeur de $e(p)$ la plus proche possible de 100% (voir figure 3.3). Par exemple, un *speedup* de seulement 70 avec 100 ressources donnera une efficacité de 70%, qui signifie que 30% du potentiel des ressources n'est pas (ou est mal) utilisé.

Souvent celui qui utilise l'application parallélisée/distribuée préférera maximiser à tout prix le *speedup*, signe de traitements toujours plus rapides. Mais celui qui finance l'achat des ressources et leur fonctionnement préférera maintenir l'efficacité au dessus d'un seuil minimal, en dessous duquel la mauvaise utilisation des ressources (le *gaspillage*) n'est pas acceptable. La gestion d'un *centre de calcul* peut prendre en compte l'efficacité des codes qui y sont exécutés : on peut ainsi limiter le nombre de nœuds alloués à des codes trop inefficaces pour les laisser à des codes mieux parallélisés.

3.3 Traitement de problèmes plus gros (*size up*)

En fait, on n'utilise pas souvent une grosse machine parallèle ou un gros système distribué pour accélérer encore plus le traitement d'un problème fixé, mais plutôt pour traiter dans le même temps

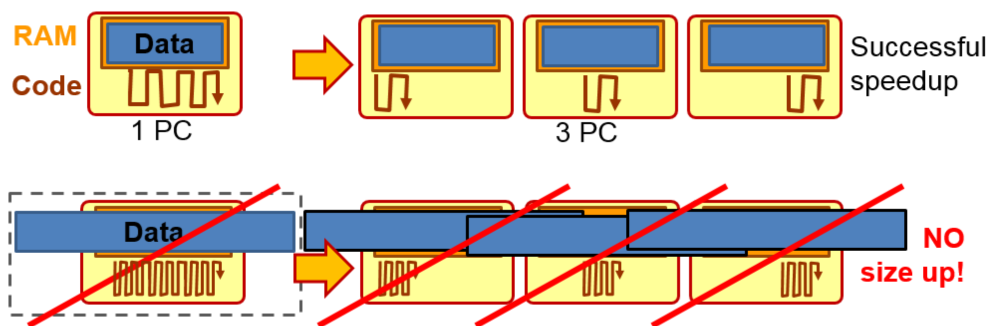

FIGURE 3.4 – Distribution répliquant les données et permettant un *speedup* mais pas de *size up*

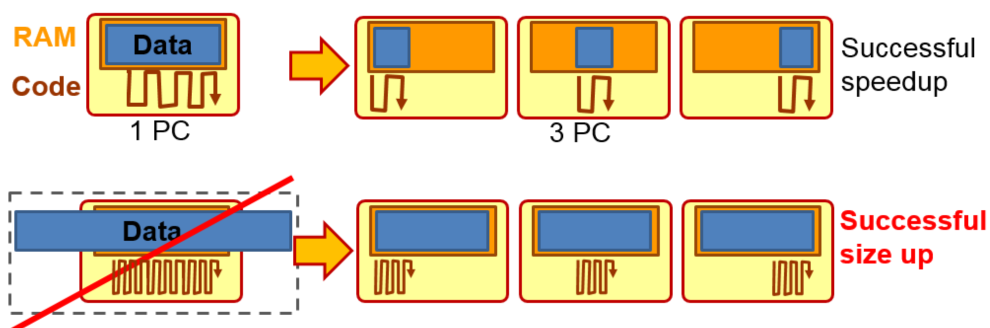

FIGURE 3.5 – Distribution partitionnant les données et permettant un *speedup* ET un *size up*

(ou presque) un problème plus gros (voir l'annexe A.2 sur la loi de Gustafson), ou pour traiter simultanément plusieurs problèmes indépendants, ou encore pour traiter simultanément plusieurs problèmes plus gros.

3.3.1 Difficultés et contraintes sur le code applicatif

La figure 3.4 montre une distribution par répartition des calculs et réplication (totale) des données sur chaque nœud. Dès lors les calculs peuvent se faire très simplement : chaque nœud peut accéder à tout moment à toutes les données dont il a besoin. Cependant, quand le problème grossit, chaque nœud continue à stocker la totalité des données dans sa mémoire, et la taille de problème traitable reste limitée par celle de la mémoire d'un seul nœud. Cette approche ne permet donc pas de traiter de plus gros problèmes en utilisant la mémoire de plusieurs nœuds : elle permet de réaliser un *speedup* mais pas un *size up*.

A l'opposé, si on arrive à *répartir* les calculs *et* les données sur un ensemble de nœuds, comme sur la figure 3.5, alors on peut traiter sur plusieurs nœuds des problèmes plus gros que sur un seul nœud. On peut alors adopter une démarche de *size up* en cherchant à augmenter la taille du problème traité, mais on peut aussi conserver une démarche de *speedup* en cherchant à accélérer un problème de taille fixe (le *size up* n'empêche pas le *speedup*).

Mais la mise au point d'un algorithme distribué avec un *partitionnement* des données est habituellement plus difficile qu'avec une réplication ! Il faut mettre au point un schéma et une implantation efficaces de *circulation* des données entre les nœuds, ce qui est parfois impossible. Il arrive que les accès aux données situées sur d'autres nœuds soient tellement fréquents que les communications engendrées effacent les gains de temps obtenus par la distribution des calculs.

L'obtention d'un *size up* n'est donc pas du tout garantie lorsque l'on entreprend de paralléliser et distribuer un algorithme. Toutefois, nous verrons plus loin que l'analyse de données commence

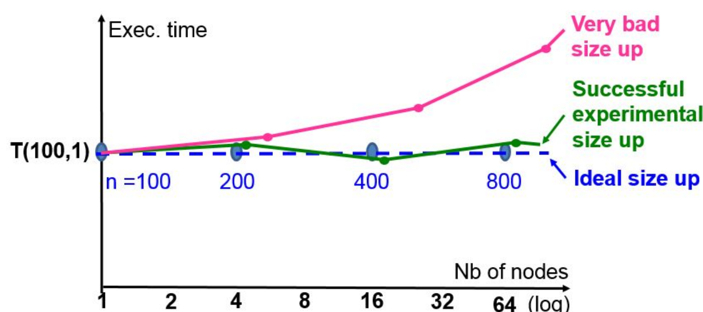

FIGURE 3.6 – Etude du maintien du temps d'exécution lors d'un *size up* : très réussi (en vert), et raté (en rose)

souvent par une phase de lecture et de filtrage de fichiers qui se prête naturellement au *size up* (voir section 5.2.3 sur les principes du paradigme de programmation Map-Reduce).

3.3.2 Métrique et courbe de temps d'un *size up*

Dans le cas où le code applicatif se prête bien à un *size up* (voir section précédente), alors un premier objectif de performance consiste à chercher à maintenir constant le temps d'exécution : *traiter dans le même laps de temps des problèmes plus gros en utilisant plus de ressources*. Ce qui se quantifie comme suit :

$$T(n_0, p_0) = T(n_1, p_1) = T(n_2, p_2) = Cte \tag{3.3}$$

La figure 3.6 illustre trois comportements différents d'une application distribuée ayant des calculs quadratiques (de complexité en $O(n^2)$) :

- La courbe bleue illustre le cas idéal où l'on arrive à trouver des couples (n_k = *taille de problème*, p_k = *nombre de nœuds*) tels que :

  $$T(n_k, p_k) = Cte, \qquad \text{avec } n_k = k.n_0, \qquad \text{et } p_k = k^2.p_0.$$

  C'est-à-dire tels que : $T(n_0, p_0) = T(2.n_0, 2^2.p_0) = T(k.n_0, k^2.p_0)$.
- La courbe verte représente une courbe de temps expérimentale d'un *size up* réussi :
  - la courbe est à peu près plate, les temps d'exécutions sont maintenus à peu près constants.
  - les nombres de ressources p_k sont à peu près ceux attendus, ils sont juste un peu plus grands que dans le cas idéal (voir section 3.3.3).
- En revanche, la courbe rose représente une courbe de *size up* où l'on n'arrive absolument pas à maintenir constant le temps d'exécution. Les points roses représentent les temps d'exécutions minimums obtenus pour des tailles de problèmes $n_k = k.n_0$, et l'on observe que ces temps minimums s'éloignent de plus en plus du temps de référence $T(n_0, p_0)$.

Il est fréquent d'arriver à traiter de plus gros problèmes en utilisant plus de ressources de calcul (notamment pour disposer de plus de RAM), mais sans arriver à maintenir constant le temps d'exécution global. Cette situation arrive notamment si la distribution de l'algorithme initial entraîne de nombreuses communications entre tâches.

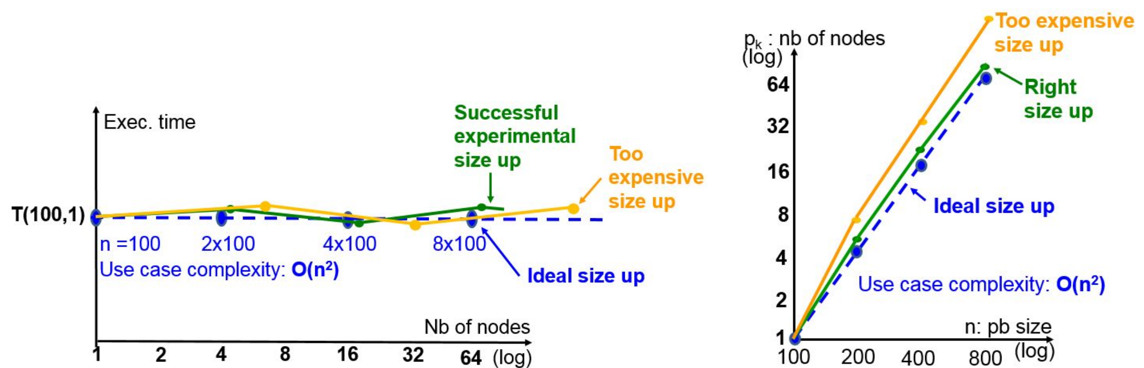

FIGURE 3.7 – Maintien du temps d'exécution d'un *size up* (figure gauche), et coût de ce maintien (figure droite) : très réussi (en vert), et trop coûteux (en orange)

3.3.3 Métrique et courbe de coût en ressources d'un *size up*

Dans le cas où le code applicatif est apte au *size up* et où l'on est arrivé à maintenir quasiment constant le temps d'exécution, alors on étudie le coût en ressources de ce *size up*. On vérifie si le nombre de ressources qui se sont avérées nécessaires était bien celui prévu, ou s'il en a fallu bien plus.

Si l'on connaît la complexité des calculs, on peut facilement estimer la quantité de travail engendrée par un problème de taille n_0 puis de taille $k.n_0$. Dans le cas d'une distribution idéale (parfaitement équilibrée et sans aucun surcoût de communication ou de synchronisation), on peut alors en déduire l'augmentation du nombre de ressources permettant de traiter le gros problème dans le même temps que le problème initial.

Soit :

- V : la vitesse d'un nœud de calcul en $operations/s$,
- $q(n)$: la quantité d'opérations engendrées par le traitement d'un problème de taille n,
- p : le nombre de nœuds de calculs utilisés,
- $T^{ideal}(n,p)$: le temps pour traiter un problème de taille n sur p nœuds de calculs, dans le cas d'une distribution *idéale* (parfaitement équilibrée sur les p nœuds avec des communications négligeables entre les nœuds).

On définit naturellement $T^{ideal}(n,p)$ par :

$$T^{ideal}(n,p) = \frac{q(n)}{p.V} \tag{3.4}$$

On cherche alors le nombre p de nœuds à utiliser pour traiter un problème de taille n aussi vite qu'un problème de taille n_0 sur 1 nœud :

$$\begin{aligned} T^{ideal}(n,p) &= T(n_0,1) \\ \frac{q(n)}{p.V} &= \frac{q(n_0)}{V} \\ p &= \frac{q(n)}{q(n_0)} \end{aligned} \tag{3.5}$$

A titre d'exemple on considère à nouveau un problème de complexité $O(n^2)$. Un problème de taille n_0 engendre alors $q(n_0) \approx \alpha.n_0^2$ opérations, et un problème de taille $2.n_0$ engendre

$q(2.n_0) \approx \alpha.(2.n_0)^2$ opérations. On obtient donc :

$$\frac{q(2 \times n_0)}{q(n_0)} = 4 \tag{3.6}$$

Selon l'équation 3.5 il nous faut utiliser $p = 4$ nœuds de calcul pour maintenir constant le temps d'exécution :

$$T^{ideal}(2.n_0, 4) = T(n_0, 1) \tag{3.7}$$

La figure 3.7 gauche montre 3 types de performances de *size up* ($T(n_k, p_k)$) sur notre exemple d'application de complexité en $O(n^2)$, et la figure 3.7 droite représente les coûts en ressources correspondants ($p_k(n_k)$).

- La courbe bleu de la figure 3.7 gauche illustre le cas idéal où $T(n_0, p_0) = T(2.n_0, 2^2.p_0) = T(k.n_0, k^2.p_0)$. Les temps d'exécutions sont constants et sont obtenus pour les nombres de ressources prévus ($p_k = k^2.p_0$). Cela se traduit par une droite de pente 2 sur la figure 3.7 droite en échelles logarithmiques.
- La courbe verte montre un cas pratique où l'on arrive à maintenir constant le temps d'exécution mais où les nombres de ressources à utiliser sont un peu supérieurs aux prévisions.
- Enfin la courbe orange montre un cas où l'on arrive à maintenir constant le temps d'exécution, mais où le nombre de ressources à utiliser augmente beaucoup plus vite que le laissait prévoir la théorie.

La courbe orange illustre donc une distribution qui fonctionne, et qui permet même de traiter des problèmes plus gros sur un plus grand nombre de machines, mais qui est très imparfaite. Les raisons les plus fréquentes sont des communications et opérations de synchronisation qui croissent plus vite que les calculs quand la taille du problème et le nombre de nœuds augmentent. Dès lors, réaliser un *size up* à temps d'exécution constant peut vite devenir très coûteux, voire insupportable, et l'on doit se contenter d'un *size up* à temps d'exécution croissant.

On dispose alors techniquement d'une solution pour traiter des problèmes de taille croissante (il est *techniquement possible* de faire les calculs), mais cette solution ne sera probablement pas utilisable fréquemment, de manière répétée, au sein d'un processus industriel complet. Mieux vaudrait rechercher un autre algorithme distribué, introduisant moins de surcoûts, même si l'on doit partir d'un algorithme initial un peu moins performant.

3.3.4 Exemple expérimental de *size up*

La figure 3.8 montre des mesures expérimentales de *size up* sur un problème de *co-simulation distribuée* [1]. Elle correspond à une application contenant des tâches hétérogènes, faisant de nombreux accès à la mémoire, et qui communiquent très fréquemment en s'échangeant de petits messages. Dans le cadre du HPC, il s'agirait d'un cas très défavorable pour lequel on chercherait absolument une nouvelle solution algorithmique avec une nouvelle modélisation si nécessaire. Mais dans le cadre de la *co-simulation*, il s'agit d'une application réelle, répondant à des besoins utilisateurs concrets, et qu'il convient de distribuer au mieux pour pouvoir réaliser des co-simulations de grandes tailles (en attendant d'avoir éventuellement une meilleure solution).

Sur les courbes de la figure 3.8 gauche, on voit que l'on arrive à maintenir le temps d'exécution constant sur les deux clusters en traitant des problèmes plus gros sur plus de nœuds. Mais dans ce problème, les calculs étaient prépondérants, les communications (même fréquentes) n'avaient presque pas d'impact, et finalement la distribution s'est avérée quasi-parfaite quel que soit le réseau d'interconnexion utilisé.

1. Travaux menées avec EDF R&D, au sein de l'institut RISEGrid

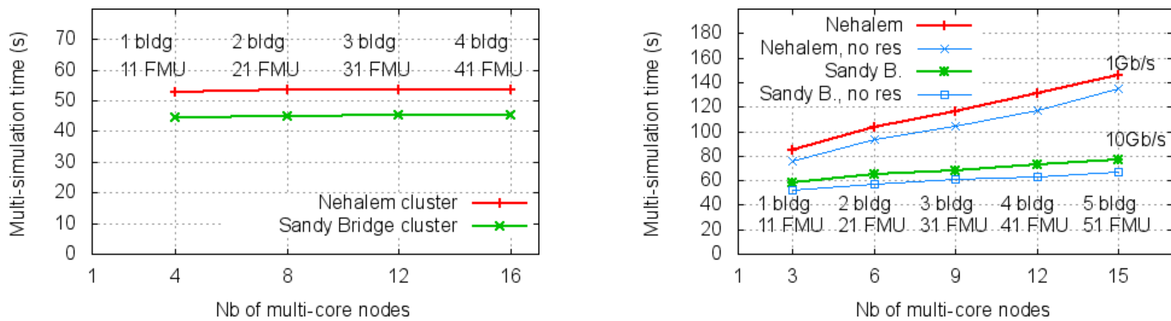

FIGURE 3.8 – Tests de *size up* sur deux problèmes de *co-simulation* réalisant beaucoup de calculs (à gauche) ou peu de calculs (à droite), et sur deux clusters de multi-cœurs avec des réseaux d'interconnexion de 1Gb/s (courbes rouges) et 10Gb/s (courbes vertes)

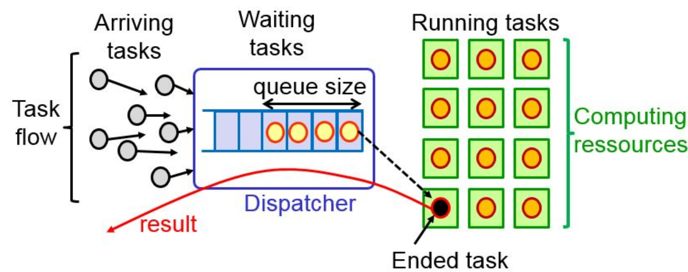

FIGURE 3.9 – Principe du traitement d'un flux de tâches dans une architecture HTC

En revanche, les courbes de la figure 3.8 droite montrent une co-simulation avec les mêmes communications mais avec des calculs moins approfondis. Cette seconde co-simulation contient donc proportionnellement plus de communications. Dès lors il s'est s'avéré impossible de maintenir constant le temps d'exécution quand le problème grossissait. Sur un réseau Ethernet 1Gb/s, l'échec est total (courbes rouges). Sur un réseau Ethernet 10Gb/s, la croissance du temps d'exécution est *supportable* (courbes vertes). Les courbes bleues correspondent à des tests de *size up* sans opérations d'E/S sur disques (pas de sauvegarde des résultats), et montrent que les E/S ne sont pas la source de l'échec du *size up*, car les temps d'exécution continuent de croître.

Cette expérience sur une application complexe réelle montre qu'il n'est pas toujours possible de réaliser un *size up* en temps constant.

3.4 Traitement de flux de requêtes plus intenses (autre *size up*)

3.4.1 Définition du *High Troughput Computing* (HTC)

Le calcul parallèle à haute performance (ou HPC) s'intéresse à accélérer au maximum l'exécution d'une application sur un ensemble de données. Le traitement de flux de tâches à haute performance (ou HTC) s'intéresse à traiter le plus de tâches possible dans un intervalle de temps, sans chercher à ce que chaque tâche soit traitée le plus vite possible. Il s'agit en fait d'accélérer globalement le traitement d'un flux de tâches pour supporter des flux plus importants [2].

Une salle de marché où un ensemble de *traders* soumettent quasiment en continu des requêtes de *pricing* en est un bon exemple. Si le fonctionnement du marché accepte qu'un *trader* prenne

2. On fait aussi parfois la différence entre le HTC (High Throughput Computing) et le MTC (Many Task Computing), selon que les temps de traitement des tâches sont longs ou courts.

$30s$ pour décider d'acheter ou de vendre un produit/contrat, chaque calcul d'un prix d'achat ou de vente pertinent doit aboutir en moins de 30s. Une solution consiste à insérer chaque requête de *pricing* dans une file d'attente, puis un mécanisme de répartition affecte chaque requête en tête de file sur une ressource de calcul disponible (voir figure 3.9). La soumission d'une requête, sa mise en file d'attente, puis son extraction et traitement doivent bien sûr se faire en $30s$ maximum.

Mais une fois ce premier objectif atteint, on préfère augmenter le nombre de ressources de calcul pour arriver à traiter plus de requêtes par minute (ou par seconde), plutôt que de diminuer encore le temps de traitement de chaque requête. On réalise ainsi un *size up* sur l'intensité du flux de requêtes.

Au départ, on considérait uniquement des tâches identiques (mêmes calculs) et séquentielles, appliquées à des données différentes. Mais les systèmes de gestion de flux de tâches d'aujourd'hui supportent des tâches hétérogènes, et éventuellement *multithreads*. Toutefois, une tâche est normalement destinée à être traitée sur un seul PC, et les tâches ne communiquent pas entre elles. Ainsi une grappe de PC, reliée à un pilote-répartiteur (*dispatcher*) par un réseau fiable et de bonne qualité, est souvent préférée à un supercalculateur pour traiter un flux de tâches.

3.4.2 Métrique de performance du traitement de flux de requêtes

L'objectif d'un système HTC est double : (1) garantir de traiter chaque tâche en un temps inférieur à une limite fixée, et (2) traiter un maximum de tâches en concurrence. On ne calcule donc pas de *speedup* pour caractériser et évaluer un système HTC, mais plutôt :

- la vitesse globale de traitement V, en $tasks/s$, ou $tasks/min$ ou même en $tasks/jour$,
- les temps de traitements maximums et moyens d'une tâche (T_{max}, T_{mean}).

Ainsi que deux critères caractérisant plus le fonctionnement interne du système :

- la taille de la file d'attente des tâches soumises mais pas encore démarrées ($Size_{queue}$),
- le temps de latence, ou temps entre la soumission et le démarrage du traitement d'une tâche ($T_{latency}$).

Lorsque la file d'attente s'allonge et que la latence de démarrage augmente, cela signifie naturellement que la grappe de ressources de calculs devient insuffisante vis-à-vis du flux de tâches, et qu'il faut augmenter les ressources de la grappe.

Evidemment, un flux de tâches peut être irrégulier, avec des pics de charges et des périodes creuses. Chacun des critères précédents peut alors être étudié en fonction du temps : $V(t)$ en $task/s$, $T_{max}(t)$ en s, $Size_{queue}(t)$ en *nombre de tâches*...

3.5 Différentes architectures pour supporter différents *size up*

Afin de faciliter des démarches de *size up*, on cherche donc à réaliser des architectures matérielles et logicielles *extensibles*. Mais ces architectures dépendent du type de *size up* visé : selon la taille du problème, l'intensité du flux de requêtes, ou le volume de données à traiter (cas du Big Data).

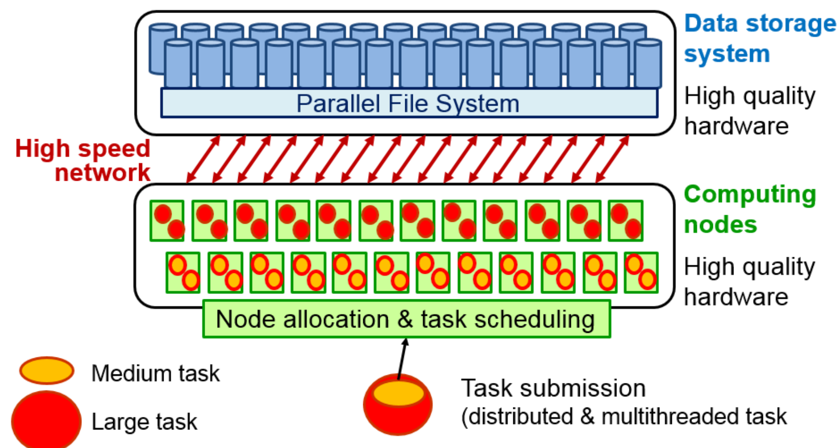

FIGURE 3.10 – Approche classique (en HPC) pour un passage à l'échelle sur la taille du problème traité

3.5.1 Architecture pour un *size up* en taille de problème

Dans une approche de calcul classique (HPC), on utilise des clusters de calculs, où une baie de serveurs de calculs et reliée à une baie de stockage par un réseau rapide (voir figure 3.10). Les nœuds ne sont habituellement pas partagés : l'exécution d'une application traitant un problème de taille n_k utilise p_k nœuds en totalité (en mode exclusif). Si le problème grossit ($n_h > n_k$) l'utilisateur demandera à allouer plus de nœuds pour l'exécuter ($p_h > p_k$), conformément à la démarche de *size up* introduite à la section 3.3.

Les clusters de calculs à haute performance, ou les supercalculateurs, sont en général capables de rester efficaces quand on utilise de plus en plus de leurs nœuds, et permettent un bon *size up* sur la taille du problème. En revanche, dans ce genre d'architecture traditionnelle, chaque partie de l'application tournant sur un nœud accède à la baie de stockage pour y récupérer sa part des données. Le réseau reliant les baies de calculs et de stockage devra donc supporter tous ces mouvements de données, et peut constituer une limitation au *size up* sur la taille du problème.

3.5.2 Architecture pour un *size up* sur un flux de requêtes

Dans une approche classique de traitement d'un flux de requêtes (HTC), on conserve la taille des requêtes mais on cherche à en traiter de plus en plus par seconde (ou par minute). Cela correspond à des utilisateurs différents lançant pleins de calculs en même temps, chaque calcul pouvant être lui-même séquentiel ou parallèle au sein d'un même nœud (voir section 3.4).

La figure 3.11 représente une architecture traditionnelle où une baie de stockage est reliée à un cluster de machines à mémoire distribuée (à gauche) ou bien à une grosse machine à mémoire partagée (à droite). Au démarrage du système, les données sont transférées des disques de la baie de stockage vers la mémoire de l'architecture de traitement. L'objectif est de sauvegarder les données sur un système redondant et fiable (la baie de stockage), mais de les transférer dans la mémoire de l'architecture de traitement en début de journée (*in memory databases*) pour réaliser des calculs *in memory*, puis de mettre à jour les données sur disque *de temps en temps* seulement ou en arrière plan des calculs. En complément, des mises-à-jour des données *in memory* peuvent avoir lieu en continu à partir de sources extérieures (cas des données de marché).

Comme vu à la section 3.4, un mécanisme répartit sur les nœuds du cluster les requêtes (séquentielles ou *multithreads*) soumises par les utilisateurs. Il vise surtout à équilibrer la charge de calcul des nœuds, mais peut tenir compte des données stockées dans la mémoire des différents

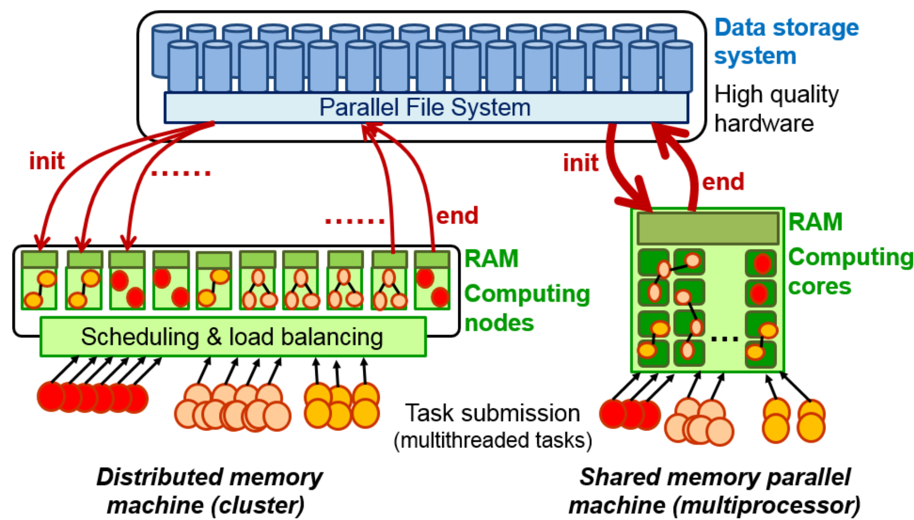

FIGURE 3.11 – Approche classique (en HTC) pour un passage à l'échelle du nombre de requêtes traitées par seconde

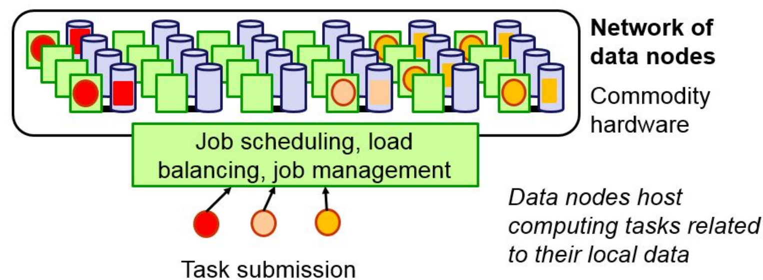

FIGURE 3.12 – Approche Big Data du passage à l'échelle en taille et en nombre de problèmes

nœuds pour éviter les communications entre nœuds. Dans le cas d'un gros multiprocesseur à espace mémoire partagé, la répartition peut sembler plus simple. Mais de telles machines ne sont pas homogènes, certains cœurs sont plus proches de certaines parties de la mémoire (architecture NUMA) et on retrouve des préoccupations de localisation du traitement des requêtes sur les unités de calculs les plus proches des données visées.

Cette problématique de réalisation des traitements sur les ressources de calcul proches des données prend de l'importance quand le volume des données augmente, jusqu'à être être au cœur des architectures Big Data (voir section suivante).

3.5.3 Architecture pour le *size up* du *Big Data*

Une troisième démarche de *size up* est celle où l'on doit gérer l'exécution de plus en plus de traitements manipulant des volumes de données croissants. Par exemple pour permettre à de plus en plus d'utilisateurs de soumettre des requêtes d'analyses de données issues d'un *data lake*, qui sont de plus en plus volumineuses. La figure 3.12 représente cette approche Big Data où les volumes de données lues sont vraiment énormes, et où les traitements ne sont pas toujours très longs comparés aux temps de lecture des données.

Dans cette approche Big Data, les nœuds sont des *nœuds de stockage de données*, qui se transforment momentanément en nœuds de calculs pour exécuter des traitements appliqués à leurs données locales. Sur la figure 3.12, on voit qu'en traversant le gestionnaire de *jobs*, la requête rouge

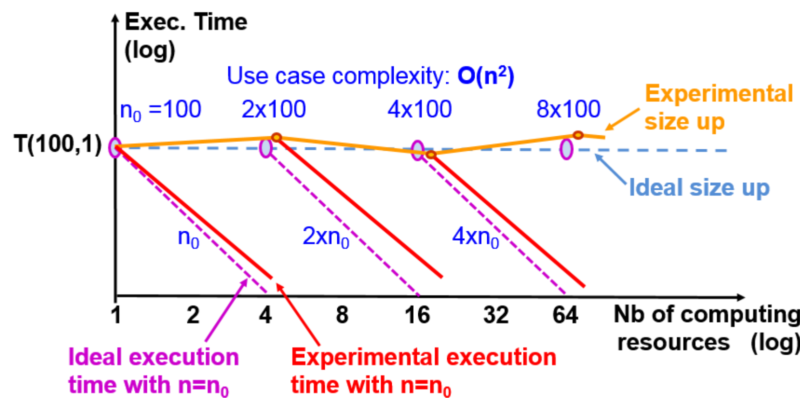

FIGURE 3.13 – Principe de passage à l'échelle par cumul du *size up* ET de la reproduction du profil d'accélération initial

donne naissance à deux tâches de calcul car les données auxquelles elle s'applique sont réparties sur deux nœuds de données différents. Les calculs sont donc amenés en priorité vers les données concernées, et non pas l'inverse, et c'est seulement si les nœuds concernés sont déjà très chargés que les traitements sont acheminés vers des nœuds voisins. Cette approche est typiquement celle de l'environnement *Hadoop* (voir chapitre 5).

Evidemment, on finit par avoir besoin d'énormément de ressources de stockage et de calculs ! Si la charge de calcul n'est pas régulière, et comporte des pics (par exemple en fin de mois, en fin de trimestre...), il est alors particulièrement intéressant de déployer cette approche Big Data dans un *Cloud*, où l'on pourra louer momentanément plus de ressources de calculs.

3.6 Passage à l'échelle en taille de problème traité

3.6.1 Définition d'un *passage à l'échelle* en taille de problème

Une démarche de *passage à l'échelle* en taille de problème (ou *scalability*) est le prolongement d'une démarche de *size up*. Partons d'une configuration permettant d'atteindre un bon *size up* : partitionnement des données sur les différentes ressources, temps d'exécution maintenu constant quand la taille du problème augmente, et coût en ressources maîtrisé (voir section 3.3). Un passage à l'échelle complet signifie en plus de reproduire le profil d'accélération initial pour chaque taille de problème, comme illustré sur la figure 3.13.

La courbe en pointillés bleus de la figure 3.13 représente un *size up* théorique idéal, et la courbe orange représente un *size up* expérimental réussi : temps d'exécution maintenu à peu près constant avec un coût en ressources voisin du coût théorique (comme introduit à la section 3.3).

Le temps d'exécution idéal du problème initial parallélisé sur p ressources de calculs est quantifié par une hyperbole d'équation : $T^{ideal}(n_0, p) = T(n_0, 1)/p$ (voir section 3.2.2). En échelle logarithmique en abscisse et en ordonnée, cette hyperbole devient une droite de pente -1, comme représenté par la courbe en pointillés violets la plus à gauche sur la figure 3.13. La courbe rouge légèrement au dessus représente le temps d'exécution parallèle expérimental, de pente habituellement un peu moins forte. Ces courbes de temps d'exécutions matérialisent en fait les *profils de parallélisation* (idéal et expérimental) de l'application. En fait, on ne peut pas tracer explicitement des courbes de *speedup*, car dès que la taille du problème croît, on ne peut plus exécuter le problème sur une seule ressource pour mesurer $T(1)$, et on ne peut donc plus calculer rigoureusement des accélérations (définies par $S(n, p) = T(n, 1)/T(n, p)$).

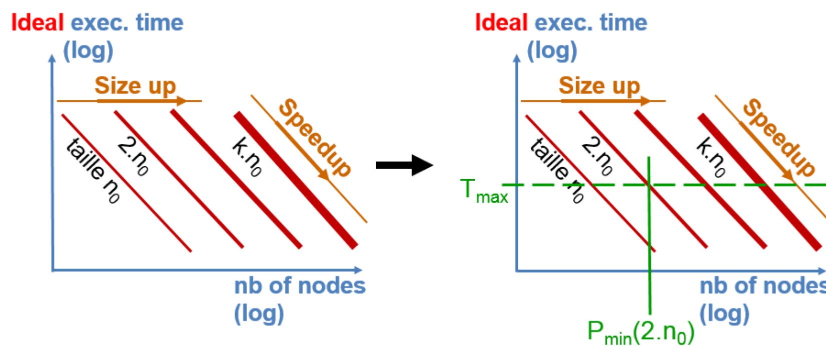

FIGURE 3.14 – Exemple d'un passage à l'échelle idéal (courbe de *scalability* à gauche), et utilisation en tant qu'abaque (à droite)

Dans un passage à l'échelle réussi, on retrouve donc des profils de parallélisation identiques et quasi-idéaux à partir de chaque taille de problème de la courbe de *size up*. Ce qui correspond aux couples successifs de courbes en pointillés violets et en trait plein rouge de la figure 3.13 (pour $n = 2.n_0$, $n = 4.n_0 \ldots$).

Une application passant à l'échelle exhibe donc un *size up* à coût maîtrisé, mais aussi un *speedup* régulier pour chaque taille de problème, ce qui facilite son exploitation quand elle est soumise à des contraintes de temps (voir sections suivantes).

3.6.2 Exploitation d'un passage à l'échelle complet en taille de problème

La figure 3.14 gauche représente des temps d'exécution d'une application distribuée sur un ensemble de nœuds de calculs, en échelles logarithmiques. Elle illustre un passage à l'échelle idéal dans lequel :

- un problème de taille donnée produit une courbe de temps d'exécution rectiligne de pente -1, donc une branche d'hyperbole parfaite, synonyme de *speedup* parfait,
- des problèmes de tailles croissantes produisent des droites rigoureusement parallèles, simplement translatées vers la droite, exhibant chacun un *speedup* parfait.
- l'ampleur de chaque translation est supposée respecter le coût théorique en ressources, exhibant un *size up* parfait.

Si une application distribuée produit ce type de courbes de temps d'exécution, alors il est toujours possible de trouver le nombre de nœuds nécessaires pour traiter un problème de taille donnée dans un temps maximum imposé.

La figure 3.14 droite illustre l'utilisation en *abaque* des courbes de passage à l'échelle : dans un laps de temps maximum de T_{max}, il est possible de traiter un problème de taille $2.n_0$ en utilisant au moins P_{min} nœuds de calculs. D'après la figure 3.14, il est possible d'accélérer encore ce traitement en utilisant encore plus de nœuds, si cela est nécessaire.

3.6.3 Exemple expérimental de passage à l'échelle

Des courbes de passage à l'échelle obtenues lors d'expérimentations d'applications réelles sont bien entendu moins lissées, moins régulières que les courbes idéales présentées sur la figure 3.14. La figure 3.15 montre ainsi des temps d'exécution réels d'une application de co-simulation

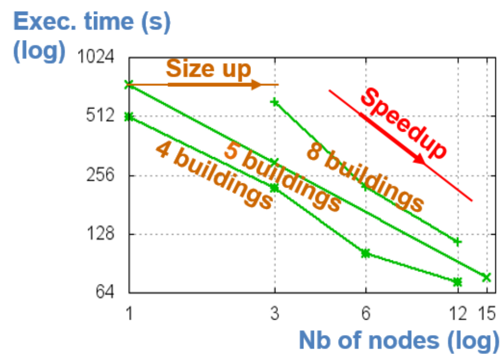

FIGURE 3.15 – Mesures réelles d'évaluation d'un passage à l'échelle

distribuée [3] de transfert de chaleur dans des bâtiments, exécutée sur un cluster de PC avec un interconnect Ethernet 10Gb/s [4].

Les pentes des différentes courbes de temps d'exécution ne valent pas -1, leurs pentes sont plus faibles en valeur absolue (accélérations moins forte que dans le cas idéal), et ces courbes ne sont pas exactement rectilignes, ni exactement parallèles. Néanmoins, le phénomène de passage à l'échelle est bien présent sur cette application distribuée. On peut planifier de traiter des problèmes plus gros dans le même temps en utilisant plus de nœuds de calculs, on peut aussi accélérer le traitement d'un problème donné en utilisant plus de nœuds, et on peut en déduire une première estimation du nombre de nœuds nécessaires pour respecter une contrainte de temps.

3.7 Exercices

3.7.1 Tri de documents et passage à l'échelle

On considère les pseudo-codes ci-après des tâches *ReaderTask* et *SortingTask*. Le premier code lit les enregistrements d'un fichier de données, et ne réécrit dans un fichier de bonnes données que ceux vérifiant une certaine condition. A chaque fichier de données correspondra ainsi un fichier de *bonnes données*. Le second code relit tous les fichiers de bonnes données, assemble toutes les bonnes données dans une liste unique, puis trie cette liste (avec un simple tri séquentiel), et enfin écrit la liste triée dans un fichier final.

```
ReaderTask(inputFileName)
{
   rightDataFileName = "RightData" + inputFileName
   inputFile = open(inputFileName,"r")
   rightDataFile = open(rightDataFileName,"w")

   while (Not EndOfFile(inputFile)) {
      data = ReadRecord(inputFile)
      if (data.date == 2000)
        write(record,rightDataFile)
   }

   close(inputFile)
   close(rightDataFile)
```

3. Une co-simulation est une application réalisant l'agrégation et le contrôle d'un ensemble de simulateurs élémentaires, exécutables en parallèle.

4. Travaux menées avec l'EDF Lab Saclay, au sein de l'institut RISEGrid.

```
}

SortingTask(listOfRightDataFileName, finalFileName)
{
   listAllRightData = []
   finalList = []

   for each rightDataFileName in listOfRightDataFileName
      listOfData = []
      rightDataFile = open(rightDataFileName,"r")
      listOfData = readAllRecords(rightDataFile)
      close(rightDataFile)
      listAllRightData.append(listOfData)
   }

   finalList = sort(listAllRightData)

   finalFile = open(finalFileName,"w")
   write(listAllRightData,finalFile)
   close(finalFile)
}
```

1. Si on a n_f gros fichiers de données initiales et au moins autant de PC, combien de tâches *ReaderTask* et *SortingTask* a-t-on intérêt à créer ? justifiez votre réponse.
2. Si chaque fichier de données initiales contient très peu de *bonnes données*, pensez-vous que ce programme passe a l'échelle ? pourquoi ?
3. Si chaque fichier de données initiales contient maintenant énormément de *bonnes données*, pensez-vous que ce programme passe a l'échelle ? pourquoi ?

Les exercices suivants nécessitent d'avoir étudié une partie de la suite du cours.

3.7.2 Map-Reduce et passage à l'échelle

Soit le programme Map-Reduce d'Hadoop suivant :

```
Mapper : < Offset, Record >
            for each < Offset, Record >
               Attribut = Record.extract("année de naissance")
               if (Attribut > 1980 and Attribut < 2000)
                 write < null-key, Record >
         < null-key, Record >

Reducer : < null-key, [R0, R1, R2...] (= lval) >
             lval2 = sort_nom(lval) // tri la liste lval par ordre
                                    // lexicographique sur les noms
             int i = 0
             for each R in lval2
               write < i, R >
               i++
          < index, Record >
```

1. Combien de Reducers conseillez-vous de créer ? pourquoi ?

2. Pensez-vous que ce programme Map-Reduce passe a l'échelle ? pourquoi ?

3.7.3 Passage à l'échelle d'un algorithme d'apprentissage

On considère que l'on doit utiliser un algorithme de Machine Learning, et l'entraîner, sur des données de tailles variées (fonction du cas d'utilisation). Ces phases d'entraînement prennent du temps, mais l'algorithme et l'implantation utilisés *passent à l'échelle parfaitement* sur une architecture distribuée.

1. Sera-t-il possible de diminuer le temps d'exécution d'un apprentissage de taille fixée en utilisant plus de nœuds de calcul ? Justifiez votre réponse.

2. Sera-t-il possible de maintenir constant le temps d'exécution d'un apprentissage sur un problème plus gros ? Justifiez votre réponse.

3. Sera-t-il possible de diminuer le temps d'exécution d'un apprentissage sur un problème plus gros ? Justifiez votre réponse.

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.