Evaluation de performances

Soraya Zertal

Table des matières

1 Introduction

Pourquoi faire un évaluation de performances ? Le but de tout système est de fournir le meilleur rapport $\frac{cout}{performance}$; d'où la nécessité d'évaluer cette performance à toutes les étapes de la vie du système :

- Conception : Prédiction des performances
- Réalisation : Choix matériel (en fonction de la puissance crête) et logiciel (compilateurs, besoin applicatifs).
- Commercialisation : Choisir les bonnes métriques (puissance crête, pire cas, utilisation courante)
- De tuning est possible à l'utilisation (exemple : utilisation d'un RAID6 pour garantir une double défaillance).
- Maintenance et mise à jour : contrôle et monitoring

Il est possible d'obtenir des valeurs et des métriques par simulation qui serviront d'argument commercial avant la réalisation du produit.

L'évaluation de performances permet de mieux connaître la faisabilité et l'intérêt d'un projet, en particulier au niveau de l'investissement. Elle permet également de détecter la faiblesses du système (composant logiciel ? Matériel ?). Cela permet également de choisir correctement les ressources à ajouter, que ce soit en terme d'unités de calculs que de stockage (problématique d'accès parallèles et de distribution de charge : striping et usure matérielle).

Il convient de respecter un méthodologie logique :

- Définir le système à évaluer et *le besoin* : aller plus vite ? Etre fiable ? Etre plus précis ?
- Sélectionner les métriques adaptées en fonction de l'étape précédente.

Il faut alors

- Lister les paramètres
- Sélectionner les facteurs et les intervalles pertinents pour le problème.

On va donc sélectionner les charges (synthétiques ou réelles), concevoir de jeux de tests les plus rapides et représentatifs possibles ; analyser et interpréter les données ainsi obtenues sous une présentation clair et lisible.

2 Métriques

2.1 Concepts de base

Définition 1 (Temps de réponse)**.** *Le temps de réponse séparant la requête de l'utilisateur de la réponse du système. Le temps de réponse augmente avec la charge du système.*

Selon le système, le taux de requête prend des unités différentes : requêtes/s, MFLOPS ou MPIPS, job/s en batch, le packet-bits/s en réseau (pps ou bps), et les transactions par secondes (TPS)pour les systèmes transactionels.

Le *débit* est une mesure de On différenciera le *débit/capacité nominal* qui est le débit maximum lorsque le système est subit à une charge idéale ; et le *débit/capacité d'usage* le débit maximal pouvant être obtenu en respectant une limite imposée au temps de réponse ($\rightarrow$ systèmes temps réel).

Un MIPS est un Million D'instruction Par Seconde sans compter le temps de chargement des instructions en mémoire :

$$MPIS = \frac{nb\_instr}{10^6 \times tps\_execution}$$

L'accélération est une métrique de performance entre deux systèmes ; elle n'est pas toujours linéaire :

$$Acc = Speedup = \frac{tpsConfigInitial}{tpsNvlleConfig}$$

En HPC, on utilisera quasi toujours

$$\frac{tps\_sequentiel}{tps\_parallele}$$

Le calcul de l'accélération par la loi l'Amdahl simple :

$$Acc = \frac{1}{(1 - F_a) + \frac{F_a}{p}}$$

avec F_a la fraction parallélisable et p le nombre de processeurs.

Le calcul de l'accélération est plus juste par la *loi d'Amdahl enrichie* :

$$Acc = \frac{1}{(1 - F_a} + \frac{F_a}{p} + C_t(F_a, p)$$

Ou $C_t(F_a, p)$ est le coût liée à la parallélisation du programme (dispatch, etc).

L'efficacité est définie par

$$E = \frac{Acc}{p}$$

Avec Acc le speedup et P le nombre de processeurs. Plus elle est proche de 1, plus l'accélération est optimale.

Le *taux d'utilisation* d'une ressource est la fraction du temps pendant laquelle la ressource est utilisée :

$$\frac{tempsOccupe}{TempsTotal}$$

On cherche à maximiser le taux d'utilisation et équilibrer les charges (Load Balancing).

Définition 2. *La* fiabilité *est la capacité d'un système à effectuer ses fonctions, de fournir des résultats non erronés et de maintenir ce fonctionnement en toutes circonstance pendant un temps T.*

On différencie *Data Reliability* et *Network Reliability* suivant que l'on cherche un accès ou un transport de données non erronées.

On peut utiliser des systèmes *redondant*, totaux ou partiels. Dans le cas total, on va faire une copie totale des données utilisées, ce qui peut être coûteux en espace. Pour la redondance partielle, on utilise un bit de parité ou un code de Reed-Solomon pour pourvoir retrouver les informations manquantes en cas de panne.

La fiabilité se mesure généralement par la probabilité d'occurrence d'erreur (on parle de MTBE ou MTBF : mean time between error/failure).

Une bonne métrique doit indiquer une performance linéairement proportionnelle à la performance actuelle (extrapolation possible + comparaison par facteur). La métrique doit également être capable de refléter correctement l'écart de deux machines.

Exemple Non fiabilité du MIPS

le benchmark *whetstone* est exécuté sur une machine pouvant utiliser un coprocesseur flottant. Une itération de ce benchmark dure 1.08s pour 1.6 MIPS avec le copro, mais 13.6s pour 2.7 MIPS sans. On a un temps supérieurs pour plus de MIPS : ce n'est donc pas une bonne métrique.

Une bonne métrique doit également être facile à mesurer afin d'éviter les erreurs et le bruit dan la prise de mesure.

Trois méthodes sont possibles :

— Analytique (math)
— Simulation
— Mesures

Chacune étant adaptée à un moment de la vie du système, avec des outils et des coûts différents.

2.2 Pour le HPC

Deux objectifs principaux : être au maximum proche du matériel (vision constructeur de composants) par des mesures élémentaires. On stress alors les composants afin de tester les comportements en condition maximales d'utilisation. On peut avoir recours à de l'assembleur et des études en profondeur en fonction de benchmarks clients.

Pour le HPC, il faut voir si la machine prend parti du parallélisme de données, éviter l'usure, éviter les effets NUMA (localité des données), voir le dimensionnement de la hiérarchie mémoire et la stratégie d'allocation (le taux de miss est une métrique classique par exemple).

On dénombre trois classes de systèmes :
— Compute Bound
— Memory Bound : taux de miss élevé
— IO Bound
— Latency bound : il s'agit d'un cas particulier de memory bound dans lequel la latence est le facteur de réduction des performances

Souvent, les industrielles n'utilisent pas de métriques exactes mais des ordres de grandeur.

On se place dans un contexte massivement parallèle. Comment paralléliser ?

$$tpsTot = tpsSeq + \frac{tpsPara}{N}$$

Pour S la fraction séquentielle, on tend donc vers un temps de $1/S$ quand $n \to +\infty$

Pour exprimer le parallélisme deux possibilités existent :
— Explicite (MPI, OpenMP)
— Implicite : Plusieurs jobs en parallèle ; Monte Carlo ; Exploration paramétrique

On parle de *scalabilité* pour la capacité à passer à l'échelle, c'est à dire supporter un plus grand nombre de tâches en parallèle. Par exemple, supporter plus d'utilisateurs, plus de threads, plus de calculs, etc...

On s'intéresse également à la capacité à accélérer le calcul à utilisation constante de la machine. On peut utiliser également MPI/OpenMP. En moyenne, en doublant le nombre de CPU, le speedup est multiplié par 1,7 (variant typiquement de 0,8 à 2). On retrouve le même facteur pour l'augmentation de charge.

Exploration paramétrique Cela consiste à lancer n applications avec des paramètres d'entrée différents. On le fait très souvent sur plusieurs machines en parallèle plutôt qu'en série.

Monte Carlo Ordre du milliard d'expérience ; pas de dépendance (tests sans mémoire). On peut lancer ces tests par un script shell. Cette méthode permet d'augmenter la précision.

Comment évaluer la scalabilité d'un code déjà parallèle ? On prend en considération le nombre de noeuds, le nomre de processeur/nœud, le temps de calcul, le temps de communication.

Comment faire augmenter la taille de problème avec le nombre de processeurs ?

Certains codes sont très scalables, comme *Linpack*. D'autres le son nettement moins, notamment à causes de points de synchronisations.

Le *workload* est l'espace occupée par une application (instruction et données). Pour les plus petites applications, tout va réussir à rentrer dans le cache ; mais ce n'est pas le cas réel. Il se produit alors des *cache miss* : on calcule alors le miss ration : le pourcentage d'occurrence de miss. La localité des instruction étant forte, il y a très peu de miss de la part des instructions.

On parle de *latence* ou *temps de réponse* pour la durée entre la requête et sa réponse ; le *débit* pour indiquer une quantité par unité de temps, et le *temps d'occupation* pour quantifié l'utilisation d'une ressource.

On utilise également l'efficacité, la fiabilité, la disponibilité, le rapport performance/prix et le rapport performance/consommation énergétiques (Flop/watt).

Les codes IO-bounds sont très fréquent en HPC alors que cela ne devrait jamais arriver (HPC = calcul!). Le problème est qu'une fois les machines très massivement parallèles achetées, on travaille très souvent sur les IO plutôt que le calcul. On dédie même des nœuds entier aux IO afin de temporiser ces pb, et on utilise des modèles (markoviens, éléments statistiques, séries temporelles) afin d'anticiper certain comportements.

On peut toujours avoir aussi des problèmes liés à la mémoire et au calcul, dans ce cas on doit détecter si l'on est memory ou CPU bound. Il suffit alors de changer graduellement la fréquence CPU via le CPU governor : si les performances baisse, vous êtes CPU bound. Sinon, on peut soupçonner que l'on est memory-bound. Il faut par contre faire très attention aux extrapolations linéaires (avec un processeur deux fois plus puissant, j'aurais...) ! Si l'on a changé de processeur pour palier à la borne CPU, il est possible que le portage du code soit très difficile (gros travail d'optimisation nécessaire). La tailles du jeu de donnée peu également faire passer d'un CPU à un Memory bound (effet de saturation du code).

John D. McCalpin a caractérisé différents profil d'accès mémoire en fonction des flux (streams) :
— `Copy : c[i] = a[i];`
— `Scale : c[i]=scalar*a[i];`
— `Add : c[i] ) a[i] + b[i];`
— `Triad : c[i] = a[i] + scalar*b[i];`
— `Tot = tot+a[i];`
MIPS = Meaningless Operation per Second ; on préfèrera les FLOPS.

Exemple Modélisation de puissance CPU (cf slides). En moyenne, 1/3 de temps est passé en CPU, 1/3 dans les caches et 1/3 en mémoire.

Pour obtenir les valeurs, on peut utiliser des benchmarks ciblés pour obtenir des mesures élémentaires (variante de mesure mémoires, latence de cache, latence mémoire. On peut également utiliser des compteurs de performances (des registres dédiés aux mesures d'évènements) tels Perfmon, Perfctl et PAPI.

Se donner des objectifs clairs :
— Pas de but
— But biaisé : démontrer qu'on a raison / faire plaisir
— Approche non systématique (surtout, ne pas tout analyser !, cibler au contraire !)
— Analyser sans comprendre le problème
— Pas la bonne métrique
— Workload non significatif (calcul scientifique alors que l'on utilise des transactions..)
— Mauvaise technique d'évaluation
— Paramètres (en oublier/trop/trop peu ou trop de biais
— Mauvais niveau de détails (ne pas simuler un processeur porte logique par porte logique
Pour analyser, il ne faut pas
— Ne pas collecter les données (souvent difficile)
— Analyse erronée
— Manque d'analyse de sensibilité
— Traitement de valeur bizarres
— Etude de variabilités