Calcul scientifique à haute performance

J. Erhel
Inria, Rennes

Janvier 2012

1 Calculateurs parallèles

loi de Moore: augmentation régulière de la vitesse des processeurs (facteur 2 tous les 18 mois) mais limite physique

Il faut aussi accroître la vitesse d'accès aux données stockées dans la mémoire: hiérarchie de mémoire; gestion de cache; dessin

1.1 architectures parallèles

Pour accroître la vitesse de calcul, on multiplie les processeurs

multiprocesseurs

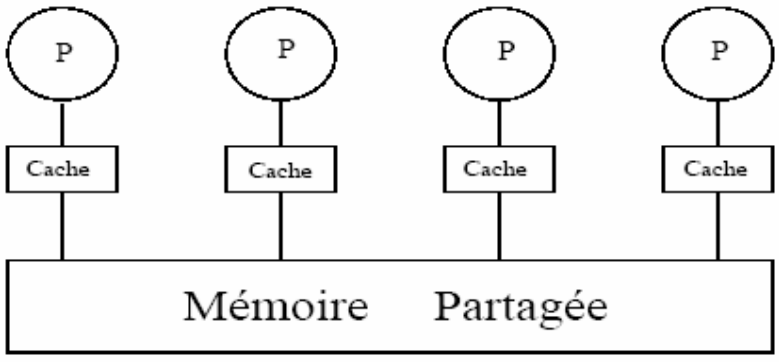
mémoire partagée (SMP: Symmetric MultiProcessor)

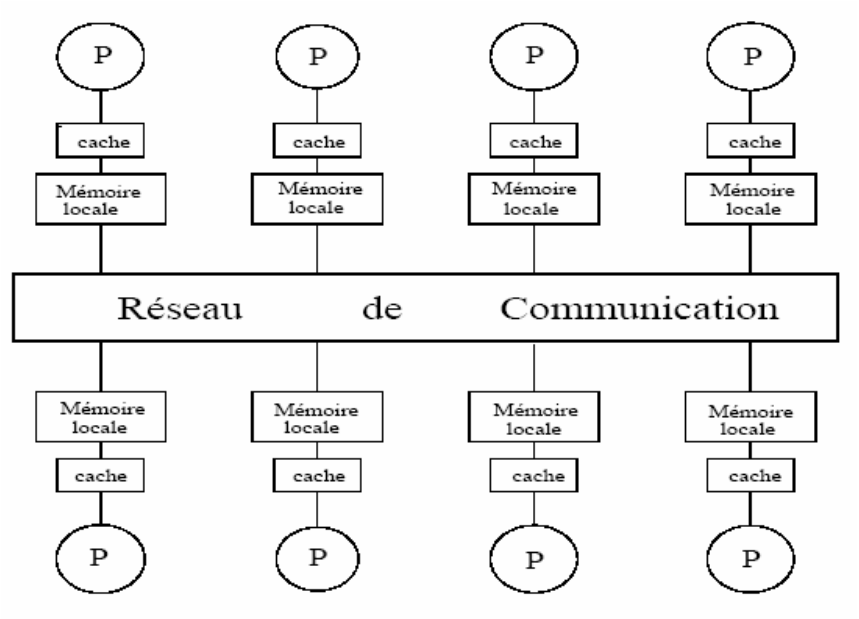
mémoire distribuée (cluster)

1.2 modèles de programmation

modèle SPMD: un seul programme, plusieurs données

mémoire partagée: variables partagées et locales; boucles parallèles (fork and join);





synchronisation pour l'accès aux variables partagées; dessin

mémoire distribuée: variables locales; communications par messages; rendez-vous point-à-point; barrière de synchronisation; diffusion un vers tous ou tous vers tous; dessins; exemples

1.3 performances parallèles

vitesse d'exécution mesurée en flops/seconde; ou en Mega Flops ou en Giga Flops, voire en Tera Flops

temps séquentiel : T_1; temps avec P processus parallèles: T_P

accélération $S_P = \frac{T_1}{T_P}$; efficacité $E_P = \frac{S_P}{P}$; objectif: $S_P = P$ et $E_P = 1$

parfois, $S_P > 1$ grâce à une augmentation des capacités mémoire

en général, $S_P < 1$ pour plusieurs raisons:

- partie non parallèle, exécutée en séquentiel sur un seul processus
- temps d'attente lorsque les tâches ne sont pas équilibrées
- temps de communication entre processus

La loi d'Amdhal modélise le surcoût dû à une partie séquentielle;

soit a la proportion de temps parallèle; on suppose que le temps de la partie parallèle est le temps séquentiel divisé par le nombre de processus, en négligeant les surcoûts. On a donc $T_P = a\frac{T_1}{P} + (1-a)T_1$ soit $S_P = \frac{P}{(1-a)P+a}$. On a $S_P = P$ si $a = 1$ mais $S_P = 1$ si $a = 0$. Pour a fixé, l'accélération est inférieure à la

valeur asymptotique (limite quand P tend vers ∞): $S_P \leq \frac{1}{1-a}$; par exemple, pour $a = 0.5$, on a $S_P \leq 2$ et pour $a = 0.9$, on a $S_P \leq 10$. Il faut donc essayer de paralléliser une fraction très importante de l'algorithme.

Les temps d'attente et de communication peuvent être fonction du nombre de processus. On suppose que $a = 1$ (on considère la partie parallèle). On a $T_P = \frac{T_1}{P} + T_1 c(P)$ et $S_P = \frac{P}{1+Pc(P)}$. On a aussi une valeur asymptotique $S_P \leq \frac{1}{c(P)}$. Il faut donc réduire les tems d'attente et de communication. Pour cela, les temps de calcul parallèle entre des points de communication ou de synchronisation doivent être assez longs: on parle de granularité pour caractériser ce temps parallèle. Cette granularité dépend de la taille N du problème considéré. Les temps de communication doivent aussi être réduits; en général, un temps de communication est la somme d'un temps fixe dit de latence et d'un temps de transfert proportionnel à la taille du message. Il faut donc réduire à la fois le nombre et le volume des communications. Les temps de communication dépendent aussi de la taille N.

Une autre façon d'analyser les performances est de mesurer l'extensibilité. On fait croître en même temps la taille du problème et le nombre de processeurs, on a donc un rapport $\frac{N}{P}$ constant. Cela permet d'avoir des ressources mémoire qui augmentent avec la taille de problème. L'objectif est d'avoir une efficacité constante.

1.4 algorithmes parallèles

diviser pour régner: l'objectif est de concevoir un algorithme où il est possible d'identifier des tâches pouvant s'exécuter simultanément. Une approche courante est de décomposer la séquence de calculs en parties indépendantes, qui échangent régulièrement des résultats pour continuer.

complexité des algorithmes: on évalue la complexité par le nombre de données stockées en mémoire et le nombre d'opérations flottantes. On distingue les algorithmes directs et itératifs. Souvent, les algorithmes directs ont une complexité polynomiale, par exemple $O(N^3)$, avec N la taille du problème. Les algorithmes itératifs ont une complexité unitaire de chaque itération, qui est multipliée par le nombre d'itérations. Celui-ci peut dépendre de la taille du problème.

gain en parallélisme versus gain en complexité: parfois, on modifie l'algorithme séquentiel pour gagner en parallélisme et en granularité. Mais c'est parfois au détriment de la complexité. Il faut alors mesurer le gain en comparant à la version séquentielle optimale.

précision des algorithmes: les calculs se font en arithmétique flottante, avec une erreur d'arrondi à chaque opération. Un algorithme est dit stable si l'erreur finale de calcul peut être bornée ou si le résultat calculé est finalement le résultat exact d'un problème avec des données légèrement perturbées. Parfois, pour accroître le parallélisme, l'algorithme est modifié et devient moins stable, plus sensible aux erreurs d'arrondi. Il faut alors quantifier cette perte de précision.

2 Etapes de calcul d'un problème de diffusion

- lecture des données
- maillage du domaine de calcul
- génération du champ de conductivité
- génération du système linéaire
- résolution du système linéaire
- calcul de la vitesse à partir de la charge

3 Génération parallèle du système linéaire de diffusion

3.1 cas stationnaire 2D

Le système discrétisé en espace est un système linéaire avec une matrice symétrique définie positive (spd). La matrice est de taille N, et est creuse: beaucoup de coefficients sont nuls.

domaine carré avec un maillage régulier: matrice pentadiagonale

domaine quelconque avec un maillage triangulaire par exemple: matrice non structurée

Les coefficients $a_{i,j}$ sont non nuls seulement pour des mailles voisines i et j; ils dépendent des conductivités K_i et K_j.

maillage régulier:

$$\begin{aligned} a_{i,j} &= -\frac{2K_iK_j}{K_i+K_j} \text{ pour } j \neq i \text{ et } j \text{ voisin de } i \\ a_{i,i} &= -\sum_{j \text{ voisin de } i} a_{i,j} \end{aligned}$$

ajustement aux bords avec les conditions aux limites

3.2 version séquentielle de l'algorithme

stockage creux de la matrice; formats coordinate et compressed sparse column

boucle sur les mailles i

maille i: calcul de la ligne i de la matrice

3.3 partition en sous-domaines et en sous-matrices

partition géométrique et algébrique

partition en sous-domaines : ensemble de mailles i dans un sous-domaine

partition en blocs des vecteurs: bloc de valeurs de conductivité K_i correspondant aux mailles du sous-domaine.

partition en blocs de lignes de la matrice: lignes associées aux cellules du sous-domaine.

distribution des données associée: données locales aux sous-domaines

calculs locaux dans les sous-domaines: calcul de $a_{i,j}$ et $a_{i,i}$ avec j voisin de i. Données locales K_i pour i dans le sous-domaine; besoin de K_j pour j voisin

sous-domaines voisins: interfaces entre voisins

cellules fantômes: recouvrement des sous-domaines

données K_i dans le sous-domaine et K_j à l'interface

autre solution: communications

algorithme:

envoyer K_i sur le bord aux processus voisins

recevoir K_j à l'interface des processus voisins

calculer $a_{i,j}$ et $a_{i,i}$