Plus près du matériel parallèle

L'objectif de ce TP est d'observer quelques comportements dûs aux caches et à la nature du parallélisme SMP/SMT/NUMA...

Vérifiez avec top qu'il ne traîne pas de processus « fou » sur votre machine qui perturberait les mesures. Éventuellement, redémarrez votre machine. Les tests doivent absolument durer plusieurs secondes et être répétés plusieurs fois.

Faites attention à ne même pas bouger la souris pendant la mesure : surtout dans un environnement KDE ou gnome, toute action utilisateur entraîne une flopée de traitements :)

Pour observer l'organisation de votre machine, utiliser la commande `lstopo` (pensez à utiliser l'option -X de ssh pour les machines distantes).

1 Accès mémoire et cache

Lisez le programme `tab.c`. Son principe est d'allouer un tampon, d'aller taper dedans et de mesurer le temps d'accès à la mémoire. La variable *gold* est là juste pour taper *pseudo-aléatoirement* dans le tampon sans pour autant que ce soit long à calculer (demandez à Donald Knuth pour les détails).

Tracez avec gnuplot une courbe à l'échelle logarithmique (à l'aide de la commande `set logscale x 2`), à quoi correspond le saut de latence ?

2 Somme de matrices

On l'a déjà vu, les effets de cache peuvent avoir des répercussions importantes sur l'exécution d'un programme. Le programme `mat.c` effectue la somme de deux matrices (donc ayant très peu de calcul par rapport aux

accès mémoire), et la somme de cosinus et sinus de deux matrices (qui fait donc déjà bien plus de calculs). La façon classique de procéder consiste à parcourir les matrices de façon à minimiser le nombre de défauts de cache. Que ce passe-t-il si l'on inverse les deux boucles ? Réfléchissez à l'organisation des défauts de cache pour chaque cas.

Parallélisez ces deux fonctions à l'aide d'OpenMP. Tracez rapidement une courbe de speedup obtenu sur vos machines, en fonction du nombre de threads.

Connectez-vous à une machine NUMA (celles de la salle 008, `cocatrix`, `boursouf` ou encore `boursouflet`), et retracez rapidement une courbe de speedup. Le résultat pour l'addition simple n'est pas terrible... Parallélisez maintenant aussi l'initialisation des matrices, et définissez la variable d'environnement

```
export GOMP_CPU_AFFINITY=0-47
```

pour activer le binding de thread sur les cœurs, pour que les threads travaillent toujours sur le même processeur et sur les mêmes données que celles qu'ils ont initialisées (et que donc le noyau a alloué sur le nœud NUMA proche du processeur d'exécution). Retracez une courbe de speedup, constatez que c'est bien meilleur.

3 Faux partage

On va observer les effets de faux partage (False sharing) sur vos machines bi-quadcore hyperthreadée, dont la topologie peut être visualisée via la commande `lstopo -p`.

Lisez le programme `test-line.c` . Son principe est de lancer deux threads qui vont de manière concurrente taper dans des variables en mémoire.

Lancez d'abord un seul thread pour obtenir une valeur de référence : le thread tourne alors tout seul, et la variable dans laquelle il tape peut rester en permanence dans le cache.

Lancez maintenant un autre thread, sur le processeur 1 par exemple. Remarquez au passage pourquoi on utilise un sémaphore. Lorsque les variables confiées aux deux threads sont proches (même si pas confondues !), on a un faux partage, conduisant à un ping-pong de lignes de cache. Faites varier l'indice de la case confiée au deuxième thread (en veillant à toujours

utiliser un multiple de 4 pour conserver tout de même des accès bien alignés en mémoire). Quelle est la taille d'une ligne de cache ?

Utilisez le même indice de tableau pour les deux threads et faites maintenant varier le numéro de processeur sur lequel vous lancez le deuxième thread.

4 Niagara (facultatif)

La machine `graup` est à base d'un processeur Niagara doté de :
- 6 cores à 4 voies
- un cache L1 privé à chaque core de 8 ko d'associativité 4 voies (dont les lignes sont constituées de 16 octets)
- un cache L2 partagé de 3 Mo d'associativité 12 voies (dont les lignes contiennent 64 octets)

Lisez le programme `hyperthreading.c`, et mesurez son temps d'exécution en fonction du nombre de threads pour K={0,1}. Notez la présence d'un appel à `lockf` qui vous permet d'éviter de vous marcher les uns sur les autres pendant les mesures.

Commentez l'ensemble des temps d'exécution. Quel phénomène est ici mis en lumière ? Ce phénomène était-il prédictible ?

Sans modifier le travail réalisé par les threads ni leur nombre, modifiez leur lancement et attente de terminaison dans la fonction `main` afin d'améliorer les performances du programme sur le processeur Niagara pour le cas K=1.