

Non Uniform Memory Access

Une machine NUMA est une machine à mémoire commune constituée de nœuds (intégrant des processeurs et de la mémoire) reliés par un réseau. Dans ces machines la latence mémoire dépend de la distance entre le nœud du cœur et le nœud contenant la mémoire cible. Le programme `hwloc-distances` permet d'afficher le facteur NUMA entre différents nœuds et la topologie peut être visualisée via la commande `lstopo`.

Les machines NUMA du CREMI sont celles des salles 008 / 203 ou les serveurs AMD (boursouf, boursouflet et jolicoeur) ou Intel (xeonphi, tesla cocatris). Avant de lancer une commande sur un serveur, vérifier à l'aide de la commande `top` que celui-ci est libre et évitez de le monopoliser pendant plus de 30 secondes.

1 Placement de tâches

On s'intéresse au programme de détection d'objets vu au TP précédent. On cherche à mesurer l'influence du placement des tâches sur les temps d'exécution. Il s'agit de comparer les performances obtenues par la parallélisation à l'aide de tâches OpenMP à celles obtenues par un ordonnanceur¹ ad hoc qui permet d'affecter les tâches aux différents cœurs.

Dans notre exemple, on désigne le cœur qui va exécuter la tâche sur la macro-cellule d'indices (i, j) vers la ligne 160 du fichier :

```
1 static inline unsigned cpu (int i, int j, sens)
  {
3   return i % P;
  }
```

Question : Quelle est la politique de distribution utilisée ici ? Pourquoi n'est-elle pas stupide ?

1.1 Influence du cache L3

Est-ce que la stratégie first touch influe sur les résultats selon la taille du problème traité ?

Pour mettre en valeur l'influence du cache L3, on pourra modifier le code ainsi :

```
coeur = ((sens == 1) ? i + 6 : i) % P;
```

Sur les machine de la salle 008/203, cette modification aura pour effet de traiter montée et descente du max au sein d'une macro-cellule sur deux processeurs différents.

1.2 Influence des bancs mémoire

Pour mieux évaluer l'influence de la localité des données, on pourra (pour les machines de la salle 008) modifier le code ainsi : ainsi :

```
coeur = ((sens == 2) ? i + 6 : i) % P;
```

Les données seront alors allouées sur le nœud opposé au cœur les traitant. Conclure.

1. Le fichier `scheduler.c` contient un code qui permet de lancer un nombre paramétrable de threads appelés *workers*. Chaque worker est vissé sur un cœur (le worker 0 est sur le cœur 0, le worker 1 sur le cœur 1, etc) et dispose d'une file de tâches à exécuter.

2 Latence mémoire sur NUMA

Il s'agit de mesurer l'impact du placement thread / mémoire sur des machines NUMA. Ici nous proposons une expérience pour essayer de quantifier ce facteur NUMA et au passage d'apprécier les latences des différents caches.

Le principe du programme `test-numa coeur noeud` est de fixer un thread sur le cœur donné, d'alouer un tampon sur le nœud NUMA donné; ensuite on mesure le temps mis pour accéder presque *aléatoirement* au contenu du buffer un nombre constant de fois (ici 2 000 000 de fois). De plus on fait varier la taille du buffer entre 1ko et 64Mo et pour chaque taille du buffer on affiche le temps mis par une itération.

Lancez `test-numa 0 0` - ici cœur et mémoire sont sur le même nœud. Expliquez les sauts de latence observé.

Lancez `test-numa 0 1` maintenant cœur et mémoire ne sont plus sur le même nœud. Estimez le *facteur NUMA* (rapport entre latence d'accès distant et latence d'accès locale).

Réessayez sur un serveur NUMA (jolicœur, boursof, tesla), constatez que la latence varie selon les positions relatives du processeur et de la mémoire.

3 Faux partage

On va observer les effets de faux partage (False sharing) sur les machines de la salle 008.

La commande `test-line distance coeur1 [coeur2 coeur3 ...]`

lance des threads qui vont de manière concurrente incrémenter des variables différentes : le thread `i` incrémente la variable `(char *)tab + i*distance`. Le programme affiche le nombre de millions d'incrémentations que chaque thread parvient à faire chaque seconde (plus c'est grand mieux c'est).

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

Lancez maintenant deux threads, sur les cœurs 0 et 1 par exemple. 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). Déterminez expérimentalement la taille d'une ligne de cache.

Fixez la distance à 8 et le premier thread sur le processeur 0 et faites maintenant varier le numéro de processeur sur lequel vous lancez le second thread. Que remarquez-vous?

Toujours avec une distance de 8, comparez l'exécution de 4 threads et ce sur différentes combinaisons (tous sur le même processeur, 2 sur chaque processeur et 3 sur l'un / 1 sur l'autre). Le partage de la ligne de cache vous semble-t-il équitable?

Testez ce programme sur un (seul) des serveurs AMD.