Université Bordeaux 1 Master Informatique **INF 477**
Examen du mercredi 14 avril 2011 Durée : 1h30 – Sans document

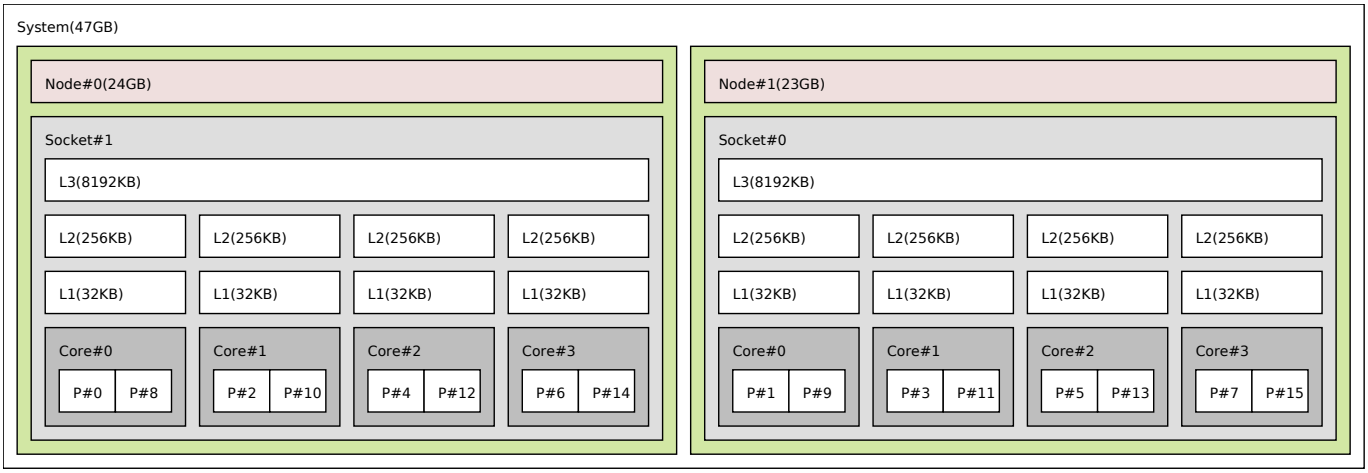
Programmation des Architectures Parallèles

Le barème est indicatif

On dispose d'un film composé d'un millier d'images codées sur 32 bits au format 1280×1024, chaque image pèse donc 5Mo. Pour chaque couple d'images consécutives on cherche à calculer le nombre de pixels différents. Les `T` *images sont présentes dans un tableau déclaré* `pixel film[T][1024][1280]` *et le résultat dans un tableau noté* `long int diff[T]` *(*`diff[i]` *contiendra le nombre de pixels différents entre les images i et i+1). Voici un exemple de code calculant ce tableau.*

```
for(int n=0; n < T-1; n++)
  for(int i=0; i < 1024; i++)
    for(int j=0; j < 1280; j++)
      if (film[n][i][j] != film[n+1][i][j])
        diff[n]++;
```

Nous allons chercher à optimiser ce code avant de le paralléliser en utilisant successivement une machine NUMA, une carte graphique puis un réseau de multiprocesseurs. Notre première cible sera une machine à base de Xeon Nehalem 8 cœurs hyperthreadés (telles les machines infini) dont voici les caractéristiques physiques :



Question 1 (1 pt) Quels sont les nombres de cycles nécessaires pour accéder à une donnée située dans un cache L1, L2, L3 et dans la mémoire sur une telle machine ? *On pourra se contenter de donner un ordre de grandeur de chaque latence.*

2 à 4 cycles, 9 cycles, 30-40 cycles et 240 cycles

Question 2 (1 pt) Admettons que l'on ait trois images consécutives à traiter sur un seul cœur. Quels sont les défauts de cache regrettables (et en fait tous évitables) générés par l'exécution du programme ci-dessus ? Pour rappel, les tailles de cache sont 32ko, 256ko et 8Mo et la taille d'une image est de 5Mo.

Il est nécessaire de lire chaque image au moins une fois, les défauts de cache issus de cette première lecture sont donc inévitables. On observe que les pixels de la seconde image sont utilisés deux fois, ce sont ces pixels qu'on peut éviter de perdre avant la seconde comparaison. Or le cache L3 n'est pas assez grand pour contenir 2 images à la fois, des pixels de la deuxième images seront chassés du cache L3. Ces pixels chassés trop tôt devront être rechargés pour la seconde comparaison.

Question 3 (1 pt) Comment réorganiser le calcul afin de minimiser le nombre de défauts de cache et de TLB ? *Il s'agit de décrire un schéma de calcul favorisant la localité des données en assurant que tout pixel ne soit chargé en cache qu'une seule fois en cache L1. Ne restez pas « bloqué » sur cette question trop longtemps.*

D'après la question 2, il faut éviter de comparer les couples d'images en une fois. Il faut donc traiter les images par morceaux, forcément par ligne pour optimiser les accès au cache (organisé en lignes).

La solution la plus simple est d'inverser les boucles *i* et *n* : on peut espérer un bon comportement car une ligne pesant 5 ko, statistiquement 3 portions d'images pourront cohabiter dans le cache L1 sans problème ce qui laisse une chance à l'algorithme LRU d'évincer les pixels inutiles. Cette solution n'optimise pas les défauts de TLB. Puisqu'une une ligne occupe 1,25 page (5ko).

Pour éviter les défauts de TLB on traitera les images par tranche de 4ko. Pour simplifier le codage, il suffit de considérer l'image comme un tableau à une seule dimension.

Question 4 (2 pts + 1 pt si vous tenez compte de Q3) Écrire un code séquentiel optimisé pour calculer le tableau `diff`. Il s'agit ici d'optimiser au mieux votre code tout en déléguant certains points au compilateur. Quels types d'optimisation utiliseriez-vous à la compilation ? Expliquer en quelques mots l'intérêt de chaque optimisation (les vôtres comme celles que vous demanderiez au compilateur) et les classer suivant leur impact sur les performances. *Il ne s'agit pas de répondre « -O3 » mais de préciser les transformations de codes attendues, vous n'avez donc pas besoin de connaître les options du compilateur pour répondre à cette question.*

```
inline void calculer_diff(int im, int it, int bloc, long int *d)
{
  int *flim1 = (int *) film[im];
  int *flim2 =(int *) film[im+1];
  long int madiff1 = 0;
  for(int i=it*bloc; i < (it+1) *bloc ; i++)
    {
      madiff += flim1[i]!=flim2[i];
    }
  *diff += madiff;
}

void difference(int taille_bloc)
{
  int iteration;
  int max = 1024*1280 / taille_bloc;

  for(iteration = 0; iteration < max; iteration++)
    for(int i=0; i < N-1; i++)
      calculer_diff(i,iteration,taille_bloc,&diff[i]);
}
```

On peut demander au compilateur de dérouler les boucles (dans ce cas on a intérêt à utiliser des constantes pour les bornes) et d'utiliser des instructions vectorielles.

Expérimentalement la version séquentielle optimisée est plus rapide que la version parallèle triviale (insertion au début du programme d'un #pragma omp parallel for)

Question 5 (2 pts + 1pt si vous tenez compte de Q3) Paralléliser le code obtenu en question 4 à l'aide d'OpenMP. *Ne pas recopier intégralement le code de la question 4.*

```
inline void calculer_diff(int im, int it, int bloc, long int *d)
{
  int *flim1 = (int *) film[im];
  int *flim2 =(int *) film[im+1];
  long int madiff = 0;
  for(int i=it*bloc; i < (it+1) *bloc ; i++)
    {
      madiff += flim1[i]!=flim2[i];
    }
#pragma OMP atomic
  *diff += madiff;
```

```
}
void difference(int taille_bloc)
{
  int iteration;
  int max = N1N2 / taille_bloc;

 #pragma OMP parallel for
  for(iteration = 0; iteration < max; iteration++)
    for(int i=0; i < N-1; i++)
      calculer_diff(i,iteration, taille_bloc,&diff[i]);
}
```

Question 6 (2 pts) D'après votre expérience, quelle accélération peut–on espérer de cette parallélisation ? Quel est le principal facteur limitant cette accélération ? Comment réduire l'impact de ce facteur ? Utiliseriez-vous *l'hyperthreading* ? Justifier.

Ce problème est proche de celui de la somme de deux matrices où l'on accède intensivement à la mémoire sans pour autant faire beaucoup de calcul. Le speed-up obtenu en TD sur ce type d'algorithme est toujours bien inférieur au nombre de cœurs disponibles sur la machine puisqu'on est limité par la bande passante de la mémoire. Par exemple le speed-up de ce programme se situe autour de 4,5 sur les machines infini (8 cœurs) ainsi que sur la machine cocatrix (nehalem 12 cœurs). Pour améliorer cela on peut essayer de répartir chaque image du film sur toutes les mémoires de la machine (le speed-up passe à 6,5 sur cocatrix). L'hyperthreading ne permet pas de résoudre ce problème de contention mémoire.

Question 7 (3 pts) Sans écrire de code, expliquer quelle stratégie de parallélisation efficace utiliser sur un GPU. Pour illustrer les choses, faites un petit dessin pour montrer comment vous pensez organiser les blocs de threads, en précisant sur quelles données chaque thread travaillera. Pensez à économiser les accès à la mémoire globale de la carte.

Il faut chercher à ne pas être pénalisé par la latence de la mémoire globale (~600 cycles) en utilisant un grand nombre de threads (pour recouvrir la latence) et la mémoire dite partagée (d'accés rapide mais de taille limitée). Pour cela il s'agit de réaliser des accès coalescés à la mémoire et d'utiliser au mieux la mémoire partagée (shared memory ou registres) en optimisant le volume de mémoire partagée occupé par thread (plus ce volume est petit plus on peut exécuter efficacement de threads en parallèle).

Ici il s'agit de réutiliser les pixels chargés, un thread va donc suivre l'évolution d'un pixel (x,y) sur une séquence d'images. Chaque thread utilisera 2 registres (pixel_pair, pixel_impair) pour suivre son pixel sur des images consécutives. Pour favoriser les accès coalescés, on constituera des blocs de 16 (ou 32 threads) où deux threads consécutifs traiteront deux pixels (32 bits) consécutifs. Ensuite il s'agit de stocker les différences entre les images, pour cela chaque thread disposera d'un tableau en mémoire partagée contenant les différences qu'il calcule. A la fin, il faudra faire une réduction pour calculer les différences via un nouveau kernel.

Il faudra régler le nombre d'images et le nombre de pixels qu'un thread traite pour optimiser l'utilisation de la mémoire partagée et obtenir de bonnes performances.

On s'intéresse maintenant à la version distribuée du programme parallèle pour la faire tourner sur grappe de machines (cluster). Les images sont initialement disponibles uniquement au processus maître et le résultat final doit résider sur le processus maître. On décide d'attribuer une séquence de film par processus esclave – on pourra supposer que T et le nombre d'esclaves sont des constantes puissances de 2 connues d'avance

Question 8 (4 pts) Donner les bouts de code contenant les requêtes de communication MPI du maitre et ceux des esclaves en précisant bien les paramètres nécessaires. Quelle amélioration apporteriez-vous à ce code si vous aviez plus de temps ?

```
// TRANCHE = nbre d'images à traiter
 // TAILLE = TRANCHE * 1024 * 1280

 // MAITRE
 for (int i = 1 ; i<= NBesclaves; i++)
```

```
    MPI_Isend(&film[i*TRANCHE],TAILLE,MPI_INT,i,0,MPI_COMMWORLD, &req[i]);
  for (int i = 1 ; i<= NBesclaves; i++)
    MPI_Irecv(&diff[i*TRANCHE],TRANCHE,MPI_INT,i,0,MPI_COMMWORLD, &req[i+NBesclaves]);
  // calculer les diff des images frontières
  MPI_Waitall(2*NBesclaves, req,&status);


  // code esclave
  int slave_diff[TRANCHE];
  MPI_Recv(slave_diff,TAILLE,MPI_INT,0,0,MPI_COMMWORLD, &slave_status);
  // calculer
  MPI_Send(slave_diff,TRANCHE,MPI_INT,0,0,MPI_COMMWORLD);
```

Question 9 (1 pt) A votre avis mieux vaut-il un processus par cœur ou un processus multi-threadé par machine ? Justifier.

L'intérêt d'une version mutlithreadée est limité puisque les threads d'un même processus ne partagent pas de données. Néanmoins l'utilisation de threads permet de diminuer le nombre de requêtes, ce qui peut être intéressant si l'on dispose de beaucoup de cœurs par machine.

Question 10 (1 pt) Quelle sera la qualité de l'accélération obtenue par l'exécution de votre programme OpenMP (cf. question 6) sur un environnement reposant sur une mémoire virtuellement partagée à base de pages (tel Cluster-OpenMP). Comment améliorer le programme pour ce type d'utilisation ? Justifier.

Dans ce cadre il faut éviter le ping-pong de pages. Nous avons déjà optimisé l'accès par page aux données du film pour réduire les défauts de TLB ; on obtiendra de ce côté-là de bonnes performances. Par contre la réduction pose problème. La solution la plus simple est de laisser le thread maître la réaliser en utilisant un tableau thread_diff[thread][image]. Autrement on peut adopter la stratégie utilisée pour MPI ou s'amuser à réaliser la réduction de façon hiérarchique.

ANNEXES

```
int MPI_[R,S,B]send(void *buf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm comm)

int MPI_Recv(void *buf,int count,MPI_Datatype datatype,int source,int tag,MPI_Comm
       comm,MPI_Status *status)

int MPI_I[r,s,b]send(void *buf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm
       comm,MPI_Request * request)

int MPI_Irecv(void *buf,int count,MPI_Datatype datatype,int source,int tag,MPI_Comm
       comm,MPI_Request *request)

int MPI_Wait(MPI_Request *request,MPI_Status *status)

int MPI_Test(MPI_Request *request,int *flag,MPI_Status *status)

int MPI_Waitall(int count,MPI_Request *array_of_requests, MPI_Status *array_of_statuses)

int MPI_Barrier(MPI_Comm comm)

MPI_ANY_TAG, MPI_ANY_SOURCE

status.source status.tag status.error status.length status.size status.bytes
```