Université Bordeaux 1 Master Informatique **IN8W16**
Examen du vendredi 26 avril 2013 Durée : 1h30 – Sans document

Programmation Multicœur et GPU

L'énoncé comporte un problème et un exercice indépendants (4 pages et une annexe).

Tas de sable abélien

On cherche à modéliser la dynamique d'éboulement des grains de sable organisé en tas sur une table. Pour cela on discrétise l'espace de la table au moyen d'un tableau 2D et on impose les règles suivantes :

- Tout grain appartient à exactement une case du tableau ;
- Toute case du bord peut accepter un nombre quelconque de grains (cela simule la chute des grains de la table) ;
- Toute case interne contenant plus de 4 grains s'éboule en cédant 1 grain à chacune de ces voisines.

On peut montrer que la situation se stabilise après un nombre fini d'éboulements et même que le tas de sable admet une forme limite qui ne dépend pas de l'ordre d'éboulement des cases. Le programme suivant permet d'atteindre cette forme limite :

```
int table[DIM+1][DIM+1];

void initialiser()
{
  ……
}

void
afficher()
{
  …
}

int main()
{
initialiser();

  int i=0;
do
   {
     //printf("**** %d ****\n",i++);
     //afficher();
   } while(traiter(1,1,DIM,DIM));

return 0;
}
```

```
int traiter(int i_d, int j_d, int i_f, int j_f)
{
  int i,j;
  int changement = 0;
  for (j=j_d; j < j_f; j++)
      for (i=i_d; i < i_f; i++)
            if (table[i][j] >= 4)
              {
                int mod4 = table[i][j] % 4;
                int div4 = table[i][j] / 4;
                table[i][j] = mod4;
                table[i-1][j] += div4;
                table[i+1][j] += div4;
                table[i][j-1] += div4;
                table[i][j+1] += div4;
                changement = 1;
              }
  return changement;
}
```

```
Trace d'exécution :
********* 0 *********
0 0 0 0 0
0 0 7 0 0
0 7 7 7 0
0 0 7 0 0
0 0 0 0 0
********* 1 *********
0 0 1 0 0
0 2 5 3 0
1 5 5 2 2
0 3 2 0 1
0 0 2 1 0
********* 2 *********
0 0 2 1 0
0 4 3 1 1
2 3 5 0 3
1 1 0 2 1
0 1 3 1 0
********* 3 *********
0 1 3 1 0
1 2 1 2 1
3 1 3 1 3
1 2 1 2 1
0 1 3 1 0
```

NB. Dans cette version on a accéléré le calcul en utilisant le reste et la division euclidienne par 4.

Optimisation du programme séquentiel (5 points)

On suppose que l'on dispose d'un cache L2 de 128 ko et dont les lignes font 64 octets (il y a donc 2048 entrées de 64 octets). Le cache est géré par la politique LRU (évincement de la ligne de cache la moins récemment utilisée).

On s'intéresse au nombre de défauts de cache provoqués par l'exécution de cet algorithme dans le pire cas (c'est à dire lorsque toutes les cellules s'éboulent à chaque étape). Plus précisément on cherche à quantifier le nombre moyen de fois qu'une ligne de cache est chargée pour l'exécution de *n* étapes consécutives. On note ce nombre moyen *Défauts(XDIM,YDIM,n)* pour un tableau d'entiers de 4 octets de dimensions *XDIM* × *YDIM*.

Q1) Donner une estimation de *Défauts(128,128,n)* et *Défauts(4096,16,n)*.
[128 ×128 × 4o = 64 ko ; 4096 × 16 × 4o = 256ko]

Q2) Proposer des optimisations du corps de boucle (calcul, test) et de la gestion du cache. Expliquer brièvement en discutant de la portée des optimisations (dans quelles cas l'optimisation est-elle intéressante – peut-elle être parfois pénalisante ?).

Q3) Donner une estimation de *Défauts(128,128,n)*, *Défauts(4096,16,n) et Défauts(16 × 2048, 8,n)* pour votre algorithme. Peut-on faire mieux ? [une ligne de la matrice = 16 × 2048 × 4o = 128 ko]

OpenMP (5 points)

Une parallélisation de l'algorithme est possible car on sait que l'ordre des calculs n'a pas d'impact sur le résultat final tant qu'on arrive à un état stable. Il faut cependant veiller à ne pas perdre de grains de sable en route et donc d'être vigilant aux frontières des zones de travail des threads.

Q4) Quelles sont les données écrite par plusieurs threads ? Préciser pour chacune d'elles la meilleure technique d'exclusion mutuelle à employer [reduction ; atomic ; critical ; omp_set_lock()/omp_unset_lock()] pour protéger leurs accès – expliquer.

Q5) Paralléliser à l'aide d'OpenMP le code obtenu en Q2 (à défaut paralléliser le code initial).

MPI (5 points)

Il s'agit d'écrire un pseudo code MPI permettant de distribuer efficacement cet algorithme en affectant une zone de travail à chaque esclave puis en faisant coopérer les esclaves pour terminer le calcul. On considère le cas 512 × 512 et 16 processus.

Pour faciliter la programmation on pourra s'aider du code suivant qui permet de prendre en compte la contribution des processus voisins lorsque l'on considère un découpage en bande.

```
// domaine de travail d'un esclave
DIMP = (DIM+1) / 16 +1 ;
int table[(DIMP)[ (DIM+1)] ;

// buffers de réception du contenu des bords
int voisin_du_haut[DIM+1], voisin_du_bas[DIM+1] ;
...
// exemple de routine de traitement du sous domaine
traiter_frontiere(table[1],voisin_du_haut) ;
...
traiter(1,DIMP,1,DIM)
...
traiter_frontiere(table[DIM-1],voisin_du_bas) ;

// autre exemple de routine
// traiter_frontiere(table[1],voisin_du_haut) ;
//...
// traiter_frontiere(table[DIM-1],voisin_du_bas) ;
//...
// traiter(1,DIMP,1,DIM)

.....
}
```

```
int traiter_frontiere(int mon_bord[], int bord_voisin[])
{
  int  j;
  int changement = 0;
  for (j=1; j < DIM; j++)
    if (bord_voisin[j] >= 4)
      {
            int div4 = bord_voisin[j] / 4;
            mon_bord[j] += div4;
            changement = 1;
      }
  return changement;
}
```

Q6) Décrire l'algorithme général, il s'agit d'ajouter une boucle et de préciser les fonctions MPI utilisées à la place des points de suspension (vous pouvez modifier le code à loisir). Ne pas détailler des paramètres mais préciser destinataires et buffers.

Q7) Préciser la (les) requête(s) MPI de la Q6 permettant l'émission de la matrice par le maitre – bien préciser les paramètres.

Q8) Préciser la (les) requête(s) de la Q6 de réception nécessaire(s) à l'échange d'information entre les esclaves – bien préciser les paramètres et le contenu des messages échangé.

Q9) Montrer comment une seule requête de réduction MPI suffit pour détecter / diffuser la terminaison de l'algorithme.

OpenCL (5 points)

Le noyau OpenCL suivant implémente les interactions entre toutes les paires d'atomes de la simulation étudiée en TP (suivant le potentiel de Lennard-Jones, mais on ne s'intéresse pas à la manière de le calculer dans cet exercice). NB: Ce code fonctionne parfaitement.

```
__kernel
void lennard_jones(__global float *pos, __global float
*speed)
{
  int index = get_global_id(0);
  int N = get_global_size(0);
  int offset = ROUND(N);
  float3 force = {0.0, 0.0, 0.0};
  float3 mypos;

  mypos.x = pos[index];
  mypos.y = pos[index + offset];
  mypos.z = pos[index + 2*offset];

  for(int i = !index; i < N; (i == index-1 ? i+=2 : i++)) {
    float3 opos;
    float r, intensity;

    opos.x = pos[i];
    opos.y = pos[i + offset];
    opos.z = pos[i + 2*offset];

    // compute intensity from lennard-jones potential
    r = distance(mypos, pos);
    intensity = compute_lennard_jones(r);

    // accumulate resulting force
    force.x += ((mypos.x - opos.x)/r)*intensity;
    force.y += ((mypos.y - opos.y)/r)*intensity;
    force.z += ((mypos.z - opos.z)/r)*intensity;
  }

  // update speed
  speed[index] += force.x;
  speed[index + offset] += force.y;
  speed[index + 2*offset] += force.z;
}
```

Q10) Expliquez comment varie l'indice de boucle 'i'. Bien que les interactions entre atomes soient symétriques (i.e. la force exercée par un atome A_1 sur un atome A_2 est de même intensité que celle exercée par A_2 sur A_1, dans une direction opposée) on calcule deux fois ces forces en définitive. Expliquez quelles difficultés poserait une implémentation où on chercherait à ne calculer l'intensité de la force entre deux atomes qu'une seule fois.

Q11) En l'état, de nombreux accès à la mémoire globale du GPU pourraient être évités en utilisant de la mémoire locale exploitée comme un cache. Modifiez la fonction en supposant que les threads appartiennent à des *workgroups* OpenCL de taille 32 (pour simplifier, on supposera que le nombre d'atomes est un multiple de 32). L'idée, pour un groupe de threads, est de progresser « tuile par tuile » en préchargeant les coordonnées des atomes en début de tuile...

ANNEXES

1 - Envionnement
Initialiser MPI et quitter MPI :
int MPI_Init(int *argc,char*** argv)
int MPI_Finalize(void)
Quitter brutalement MPI :
int MPI_Abort(MPI_Comm comm)
Savoir si un processus à fait un MPI_Init :
int MPI_Initialized(int *flag)
Récupérer la chaine de caractères associée au code d'erreur err:
int MPI_Error_string(int errcode,char *chaine,int *taille_chaine)

2 - Communications point à point bloquantes
Envoyer un message à un processus :
int MPI_[R,S,B]send(void *buf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm comm)
Recevoir un message d'un processus :
int MPI_Recv(void *buf,int count,MPI_Datatype datatype,int source,int tag,MPI_Comm comm,MPI_Status *status)
Envoyer et recevoir un message :
int MPI_Sendrecv(void *sendbuf,int sendcount,MPI_Datatype sendtype,int dest,int sendtag,void *recvbuf,int recvcount,MPI_Datatype recvtype,int source,int recvtag,MPI_Comm comm,MPI_Status *status)
Compter le nombre d'éléments reçus :
int MPI_Get_count(MPI_status *status,MPI_Datatype datatype,int *count)
Tester l'arrivée d'un message :
int MPI_Probe(int source,int tag,MPI_Comm comm,MPI_Status *status)

3 - Communications point à point non bloquantes
Commencer à envoyer un message :
int MPI_I[r,s,b]send(void *buf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm comm,MPI_Request * request)
Commencer à recevoir un message :
int MPI_Irecv(void *buf,int count,MPI_Datatype datatype,int source,int tag,MPI_Comm comm,MPI_Request *request)
Compléter une opération non bloquante :
int MPI_Wait(MPI_Request *request,MPI_Status *status)
Tester une opération non bloquante :
int MPI_Test(MPI_Request *request,int *flag,MPI_Status *status)
Libérer une requête avant de la réutiliser :
int MPI_Request_free(MPI_Request *request)
MPI_Test checks the status of a specified non-blocking send or receive operation. The "flag" parameter is returned logical true (1) if the operation has completed, and logical false (0) if not. For multiple non-blocking operations, the programmer can specify any, all or some completions.
int MPI_Testany(int count,MPI_Request *array_of_requests,int *index, int *flag,MPI_Status *status)
int MPI_Testall(int count,MPI_Request *array_of_requests, int *flag,MPI_Status *array_of_statuses)
int MPI_Testsome(int incount,MPI_Request *array_of_requests,
int *outcount,int *array_of_indices, MPI_Status *array_of_statuses)
int MPI_Iprobe(int source,int tag,MPI_Comm comm,int *flag, MPI_Status *status)
int MPI_Cancel(MPI_Request *request);
MPI_Waitall(), MPI_Cancel MPI_Iprobe(), MPI_Test_cancelled()

4 - Communications persistantes
Décrire un schéma persistant :
int MPI_[R,S,B]send_init(void *sendbuf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm comm,MPI_Request *request)
int MPI_Recv_init(void *recbuf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm comm,MPI_Request *request)
Démarrer une communication persistante :
int MPI_Start(MPI_Request *request)
Autres fonctions
MPI_Startall(), MPI_Request_free()

5 - Communications collectives
Barrière :
int MPI_Barrier(MPI_Comm comm)
Diffusion générale d'un message :
int MPI_Bcast(void *buf,int count,MPI_Datatype datatype,int root,MPI_Comm comm)
Collecte de données :
int MPI_Gather(void *sendbuf,int sendcount,MPI_Datatype sendtype,void *recvbuf,int recvcount,MPI_Datatype recvtype ,int root,MPI_Comm comm)
Diffusion sélective d'un message :
int MPI_Scatter(void *sendbuf,int sendcount,MPI_Datatype sendtype,void *recvbuf,int recvcount,MPI_Datatype recvtype ,int root,MPI_Comm comm)
Collecte de données et rediffusion :
int MPI_Alltoall(void *sendbuf,int sendcount,MPI_Datatype sendtype,void *recvbuf,int recvcount,MPI_Datatype recvtype,MPI_Comm comm)
Calcul d'une réduction :
int MPI_Reduce(void *sendbuf,void *recvbuf,int count,MPI_Datatype datatype,MPI_Op operation,int root,MPI_Comm comm)
Calcul d'une réduction et rediffusion du résultat :
int MPI_Allreduce(void *sendbuf,void *recvbuf,int count,MPI_Datatype datatype,MPI_Op operation,MPI_Comm comm)

Opérateurs de MPI_Reduce et MPI_Allreduce :
MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, MPI_BAND, MPI_BOR, MPI_BXOR, MPI_LAND, MPI_LOR, MPI_LXOR

6 - Constantes
Jokers :
MPI_ANY_TAG, MPI_ANY_SOURCE
Datatypes élémentaires :
MPI_CHAR, MPI_SHORT, MPI_INT, MPI_LONG, MPI_FLOAT, MPI_DOUBLE, MPI_LONG_DOUBLE,
MPI_UNSIGNED, MPI_UNSIGNED_CHAR, MPI_UNSIGNED_SHORT ,MPI_UNSIGNED_LONG,
MPI_LOGICAL, MPI_BYTE, MPI_PACKED
Constantes réservées :
MPI_PROC_NULL, MPI_UNDEFINED
Communicateurs reservés :
MPI_COMM_WORLD, MPI_COMM_SELF

7 – MPI_Status

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