Université Bordeaux 1 Master Informatique **INF 312**
Examen du mercredi 2 mai 2007 Durée : 1h30 – Sans document

Programmation des Architectures Parallèles

Ne soyez effrayé par la longueur du sujet, il n'est pas nécessaire de résoudre toutes les questions pour obtenir une excellente note.

Scheduling OpenMP.

1. Quelles sont les caractéristiques des programmes OpenMP qui tirent avantage d'une distribution dynamique ? d'une distribution statique ? Expliquez.

2. D'après votre expérience quelle politique utiliseriez-vous pour le jeu de la vie sur machine SMP ? sur machine NUMA ? Justifiez.

Latence des programmes MPI. Parfois, on constate que la procédure MPI_Isend ne déclenche pas la transmission effective des données.

3. Décrivez une expérience mettant en évidence ce phénomène. Illustrez votre propos à l'aide d'un chronogramme des messages échangés.

4. En vous inspirant éventuellement de la description de l'implémentation du protocole MX donné en cours, expliquez pourquoi la plupart des implémentations MPI actuelles déclenchent le transfert effectif des données que lors d'un (autre) appel à une fonction MPI.

5. Montrez alors comment un programmeur dont le programme est fortement pénalisé par ce phénomène peut y pallier afin d'améliorer les performances de son programme.

6. Quelles solutions les concepteurs de bibliothèques de MPI pourraient-ils proposer afin de remédier à ce problème ?

Protocole MOESI. Les états du protocole de cohérence de cache MOESI sont ainsi décrits :

> Modified: la ligne de cache contient la donnée la plus récente qui ne se trouve dans aucune autre mémoire cache. De plus, la valeur contenue par la mémoire principale est incorrecte.
>
> Owned: la ligne de cache contient la donnée la plus récente qui peut se trouver aussi à l'état shared dans d'autres mémoires caches. En revanche, la valeur contenue par la mémoire principale est incorrecte.
>
> Exclusive: la ligne de cache contient la donnée la plus récente et celle-ci ne se trouve dans aucune des autres mémoires caches. La valeur contenue par la mémoire principale est correcte.
>
> Shared: la ligne de cache contient la donnée la plus récente et peut être aussi dans d'autres mémoires caches. La valeur contenue par la mémoire principale est correcte si et seulement si aucune mémoire cache ne détient la donnée à l'état Owned.
>
> Invalid: la ligne de cache ne contient pas la valeur actualisée de la donnée. Celle-ci se trouve en mémoire principale ou dans la mémoire cache d'un autre processeur.

7. Considérant l'automate du protocole MOESI, décrivez une transition arrivant dans l'état Owned. Il s'agit de donner l'état d'origine de la transition (que vous choisirez), l'événement déclenchant (que vous choisirez) et la liste des actions à réaliser.

8. Quelles sont les architectures qui tirent le plus avantages de l'introduction de l'état Owned ? Illustrez votre réponse.

Analyse de performance. Il s'agit d'optimiser l'exécution d'un programme sur une machine à base d'un processeur Niagara doté de :

- 6 cores à 4 voies ;
- un cache L1 privé à chaque core de 8ko d'associativité 4 voies (dont les lignes sont constituées de 16 octets) ;
- un cache L2 partagé de 3Mo d'associativité 12 voies (dont les lignes contiennent 64 octets).

```
int main (int argc, char **argv)
{
  pthread_t thread_id[256];
  int i;

   for(i=0; i < atoi(argv[1]); i++)
     pthread_create (thread_id+i, &attr, f, (void*) i);

   for(i=0; i < atoi(argv[1]); i++)
     pthread_join (thread_id[i], NULL);

  return EXIT_SUCCESS;
}
```

```
volatile int TAB[256 * 64 * 1024];

void *
f (void *arg)
{
  int i, j, K = 1 ;

  for(j=0; j<10000;j++)
     for(i=0; i < 64 * 1024; i+=16)
        TAB[i *K+ 64 * 1024 * (int) arg]=i;
  return NULL ;
}
```

Voici les temps d'exécution en secondes du programme en fonction du nombre de threads exécutés et de la constante K:

Argv[1]	K=0	K=1	Argv[1]	K=0	K=1	Argv[1]	K=0	K=1	Argv[1]	K=0	K=1	Argv[1]	K=0	K=1
1	0,7	0,7	7	0,73	0,74	13	1,08	2,62	19	1,57	13,85	30	2,55	23,98
2	0,7	0,7	8	0,74	0,74	14	1,16	5,19	20	1,66	15,22	36	3,03	28,73
3	0,7	0,7	9	0,75	0,75	15	1,25	7,21	21	1,73	16,43	42	3,56	33,65
4	0,71	0,71	10	0,83	0,75	16	1,33	9,07	22	1,82	17,56	48	3,99	38,74
5	0,71	0,71	11	0,92	0,75	17	1,41	10,79	23	1,9	18,58			
6	0,72	0,72	12	1	1,43	18	1,49	12,37	24	1,98	19,45			

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

10. Sans modifier le travail réalisé par les threads (ni leur nombre), modifiez leur gestion dans la fonction main afin d'améliorer les performances du programme sur le processeur Niagara pour le cas K=1. Donnez une estimation du temps mis par votre programme en fonction du nombre de threads exécutés.

Barrières distribuées. Il s'agit d'implémenter de façon économique une (et une seule) barrière en utilisant un réseau à capacité d'adressage tel SCI. Pour communiquer entre eux les N processus disposent d'une structure *local* et d'un tableau de pointeurs *remote :*

*struct barrier local,*remote[N];*

Le processus j peut alors accéder à un champ *local.var* du processus i via *remote[i]->var*

11. Après avoir détaillé le contenu de struct barrier, donner le pseudo-code de la barrière : barrier_init() et barrier_wait().

12. Donner le chronogramme des messages échangés pour synchroniser deux fois de suite 3 processus.

13. Donner si possible le nombre de messages échangés pour synchroniser n processus. En supposant que la latence du réseau est 1 unité de temps, évaluer le nombre d'unité de temps nécessaires à la synchronisation de n processus. Est-il possible de faire mieux ? Comment ?

1. Réaliser une distribution statique pour un programme (fortement) irrégulier c'est prendre le risque de mettre au chômage un nombre non négligeable de processeurs. De plus, lorsqu'il est possible de prévoir le comportement d'un programme, on peut distribuer la charge de travail aux différents threads de façon équilibrée et ce dès le démarrage voire dès la compilation ce qui évite les surcoûts induits à la répartition dynamique. Aussi plus un programme a un comportement irrégulier plus on a avantage à utiliser une répartition dynamique pour atteindre un bon équilibre des charges et donc de bonnes performances.
2. Dans cette question, il s'agissait avant tout d'argumenter, la réponse n'étant pas unique car elle dépends de votre expérience et aussi des machines utilisées. Voici une réponse possible. Suivant l'algorithme utilisé (optimisations, type de découpage) et suivant les données le jeu de la vie peut être ou ne pas être régulier. Une fonctionnalité d'OpenMP est la possibilité choisir dynamiquement la politique à utiliser (on peut en changer en cours d'exécution). On peut donc commencer par une politique statique (pour répartir les données) puis tester le coût d'une politique dynamique puis choisir la meilleure politique.
3. En faisant varier la taille d'un jeton tournant sur n processus à l'aide de la séquence « Isend() ; sleep(1) ; Wait() » permet de mettre en valeur le phénomène en question. Lorsque le volume de donnée est important la plupart des implémentations utilise le mode « rendez-vous » pour faire le transfert. Si on utilise le couple Isend/Wait on s'aperçoit que les données sont effectivement envoyées lors de l'appel au Wait, Isend envoyant uniquement la demande de RdV.
4. Simplement parce que la bibliothèque n'a pas la main pour traiter l'acquittement du RdV (appelé phase de « matching » sous MX) qui sera traité au prochain appel à une fonction de la bibliothèque, les données ne peuvent donc pas être transmises avant.
5. Le programmeur peut glisser des appels « algorithmiquement inutiles » à la bibliothèque afin qu'elle puisse réaliser le « matching »… Usuellement on insère des MPI_Test().
6. MX propose d'utiliser un thread pour faire progresser les communications, il est possible d'utiliser des interruptions cependant toutes les cartes ne sont pas capables d'en générer.
7. Transition Modified → Owned sur BusRd on transmet la donnée en signalant qu'elle ne provient pas de la mémoire.
8. [C'est le protocole utilisé par les AMD 64.] Les architectures hiérarchiques telles les NUMA (parce que la mémoire peut être éloignée des processeurs disposant de l'information) ou encore les architectures SMP disposant de plusieurs bus reliant eux mêmes plusieurs processeurs (parce qu'on risque moins de saturer l'accès à la mémoire)
9. Pour K=0 l'augmentation est linéaire à partir de 9 threads, les threads s'exécutent « harmonieusement », il n'y a pas de saut, la machine est donc capable d'exécuter un nombre croissant de tâches sans problème. Pour K=1 chaque thread balaye un espace de 256ko en sautant de ligne de cache en ligne de cache (4192 fois) et recommence (10000 fois). On constate une augmentation importante du temps d'exécution (facteur 2) entre 11 et 12 threads et aussi entre 12 et 13, etc… à partir de 18 threads les performances continuent à se dégrader mais linéairement cette fois-ci. Les dégradations de performance sont dues à la saturation des caches. La saturation du cache L1 n'est pas la cause du phénomène observé car l'exécution d'un seul thread suffit à le saturer (sa taille est de 8ko), son influence est donc négligeable dans cette expérience. Il s'agit donc du cache L2 partagé par tous les coeurs. On peut noter que cette saturation intervient lorsque la limite d'associativité est atteinte (12 way) alors qu'il pourrait rester de l'ordre 512 ko de cache libre : à chaque tour de boucle la ligne de cache d'un thread chassera celle d'un autre qui chassera probablement celle d'un troisième etc... le cache L2 n'est plus utilisé. Ce phénomène est en gros prédictible puisqu'on peut faire tourner 24 threads et que l'associativité du cache L2 est 12 ; cependant prévoir l'importance du phénomène et son seuil (entre 11 et 12 tâches) demande une analyse très fine.

10. Initialement exécuter 22 tâches prends 18s alors qu'exécuter 11 tâches prends 0,75s et donc exécuter 2 fois 11 tâches devrait prendre moins de 1,5s. On en conclut qu'il vaut mieux exécuter les threads par paquet de 11. Cela ce fait en introduisant une boucle pour lancer un paquet et une autre pour attendre la terminaison de chaque thread du paquet. On peut s'attendre à un temps d'exécution inférieure à 0,75 * arrondi_supérieur (n / 11) s.

11. On suppose que les objets sont initialisés à 0 a leur création. Il ne fallait surtout pas utiliser de mutex, ni de condition car on travaille ici de façon distribuée.

```
struct barrier {bool parity ; bool flag[N]} ;

void barrier_init(){ remote[my_id] = &local}

void barrier_wait()
{
  for(int i = 0 ; i < N ; i++)
        remote->flag[i] = !local.parity ;

for(i=0 ; i < N ;)
  if (local.flag[i] != local.parity)
     i++ ;
 local.parity = !local.parity ;
}
```

12. À faire… je n'aime pas dessiner !

13. On a ici N(N-1) messages échangés et au moins N-1 étapes de communication. Dans un réseaux point à point on peut faire mieux en utilisant un schéma de communication arborescent pour organiser la diffusion de l'information en 2 phases : remontée vers « la » racine du fait que tous les processus du sous arbre ont atteint la barrière puis diffusion du franchissement réussi de la barrière. Cela nécessitera de l'ordre de 2N messages communication et de l'ordre de 2 log(N) étapes. Pour la diffusion on a intérêt à utiliser si possible la technique du *multicasting*.