Examen du cours I3
Programmation hybride et multi-cœurs

Vendredi 23 février 2018 - durée 3 heures
Supports de cours autorisés

Dans les questions où on demande d'écrire des lignes de code, les erreurs de syntaxe ne seront pas prises en compte (ponctuation, nom exact des fonctions, ordre des arguments, etc.). Du moment que vous indiquez clairement ce que fait chaque ligne de code ajoutée pour répondre aux questions.

Question de cours n° 1
Définissez les notions de localité temporelle et localité spatiale. Donnez un exemple simple pour illustrer chacune de ces deux notions.

Question de cours n° 2
Rappeler les différences principales entre la programmation OpenMP «grain fin» (fine-grain) et «gros grain» (coarse-grain).

Question de cours n° 3
Dans le modèle de programmation hybride MPI-OpenMP, décrivez les principaux avantages espérés par rapport à une programmation purement MPI et une programmation purement OpenMP.

Problème n° 1
Le but de cet exercice est de calculer une valeur approchée de π par intégration numérique en modèle de programmation mixte MPI+OpenMP.
La machine de callcul dispose de N nœuds, chaque nœud étant composé de P processeurs (on considère ici que 1 cœur = 1 processeur). Le nombre total de processeurs (cœurs) est donc de $N \times P$.
Soit la formule suivante utilisée pour calculer la valeur du nombre π :

$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$

Le programme séquentiel suivant permet de calculer une valeur approchée de cette intégrale en utilisant la méthode du trapèze. Cette méthode simple consiste à remplir la surface sous la courbe par une série de petits rectangles. Lorsque la largeur des rectangles tend vers zéro, la somme de leur surface tend vers la valeur de l'intégrale (et donc vers π).

```
1  #include <iostream>
2
3  int main() {
4    int lNumSteps = 1000000000;
5    double lStep = 1.0 / lNumSteps;
6    double lSum = 0,x;
7
8    for (int i =0; i < lNumSteps ; ++i ) {
9      x = (i+0.5) * lStep;
10     lSum += 4.0/(1.0 + x*x);
```

```
11      }
12    double pi = lSum*lStep;
13
14    std::cout.precision(15) ;
15    std::cout << "pi = " << pi << std::endl;
16
17    return 0;
18  }
```

Question n° 4
Ajouter une parallélisation OpenMP (type grain fin).

Question n° 5
Ajouter une parallélisation MPI pour obtenir une parallélisation hybride.

Problème n° 2
Le but de cet exercice est de calculer la solution de

$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}(\kappa(u)\frac{\partial u}{\partial x})$$

où $\kappa(u)$ est une fonction non-linéaire de u et on suppose que le temps nécessaire pour calculer $\kappa(u)$ est proportionnel à $|u|$.

Le schéma numérique utilisé ici, calcule une approximation u_i^n de $u(i\Delta x, n\Delta t)$.
Il s'écrit

$$u_i^{n+1} = u_i^n + \frac{\Delta t}{\Delta x^2}\left(-\kappa(\frac{u_i^n + u_{i-1}^n}{2})(u_i^n - u_{i-1}^n) + \kappa(\frac{u_{i+1}^n + u_i^n}{2})(u_{i+1}^n - u_i^n)\right)$$

pour calculer $u_i^{n+1}, i = 1, \ldots, N-1$ à partir de $u_i^n, i = 0, \ldots N$ (on supposera u nul au bord du domaine).
On supposera que le schéma est stable et sa précision acceptable.

En langage C, le schéma pour passer de `v[i]` $= u_i^n$ à `w[i]` $= u_i^{n+1}$ s'écrit

```
1       for (i=1; i<N; i++)
2           w[k] = v[i] + dt/(dx*dx) * (
3                   - k(0.5*(v[i] + v[i-1])) * (v[i] - v[i-1])
4                   + k(0.5*(v[i+1] + v[i])) * (v[i+1] - v[i]))
5
```

Question n° 6
Ajouter un pragma OpenMP avant la boucle sur `v[k]` pour paralléliser cette boucle sur plusieurs threads.

Question n° 7
Qu'est ce qui peut provoquer du déséquilibre de charge entre les différents threads dans la boucle ci-dessus ?

Question n° 8
Proposer une parallélisation type gros-grain qui essaie de mieux répartir la charge.