TP : Programmation hybride MPI + X

Stencil : équation de la chaleur MPI + OpenMP.

Alexandre DENIS
Alexandre.Denis@inria.fr

ENSEIRB CISD, IT389 — 2019–2020

L'objectif de ce TP est d'expérimenter la programmation hybride mélant MPI et OpenMP. Une séance sera consacrée à ce TP, qui sera à finir en projet autonome et sera noté.

1 Présentation du stencil

Nous allons travailler avec une simulation de la transmission de la chaleur dans un solide. Après discrétisation en temps et en espace, on obtient le code de calcul que vous pouvez télécharger sur la page du cours que vous trouverez à partir de ma page d'enseignement [1]. Ce code utilise une condition de Dirichlet (conditions aux limites fixées) et itère jusqu'à l'obtention du règime stationnaire. Pour simplifier, nous travaillerons en 2 dimensions.

2 Travail à faire

Les travaux de ce TP feront l'objet d'un compte-rendu, qui décrira vos méthodes, approches, idées, observations, accompagné du code correspondant. Il sera tenu compte dans la notation de la clarté des explications, de la concision, de l'orthographe.

Merci d'envoyer le tout par e-mail à `Alexandre.Denis.inria.fr` au plus tard le 29 janvier 2020, dans un fichier `.tar.gz` contenant le rapport en PDF ainsi que l'ensemble du code.

2.1 Version séquentielle

La version séquentielle vous est donnée.

— Faites varier la taille du problème, mesurer la performance obtenue. Calculez la performance en *gigaflops* et en nombre de cases par seconde. Désactivez l'affichage de la matrice pour les grandes tailles !

— Qu'est-ce qui limite la performance du test de convergence ? Que pensez-vous du fait de faire ce test dans un parcours des données séparé du calcul ?

Vous constaterez que la performance obtenue est éloignée de la performance nominale du processeur. Dans un code plus réaliste, il faudrait procéder à quelques optimisations au préalable :

1. `http://dept-info.labri.fr/~denis/`

— *vectorisation, pour exploiter toutes les unités vectorielles du processeur ;*
— *calcul par blocs, pour optimiser l'utilisation du cache.*
— *test de convergence à la volée.*

2.2 Version OpenMP

— Transformer le code en version OpenMP. Qu'observez-vous sur les performances ?

2.3 Version MPI

— Transformer le code en version MPI pure.
— Pensez à définir les bons datatypes pour les halos.
— Utilisez (ou non) une topologie MPI cartésienne.
— Vous pouvez commencer par une version qui ne teste pas la convergence en faisant un nombre d'itérations fixe, puis ajouter la détection de convergence (`MPI_Reduce`) dans un deuxième temps.
— Tracer les courbes de scalabilité forte et de scalabilité faible pour des tailles de données qui vous semblent pertinentes.
— Que pensez-vous de cette version pure MPI avec un processus par coeur ?

2.4 Code hybride MPI+OpenMP

— Proposez une version hybride MPI + OpenMP en combinant votre version MPI et votre version OpenMP.
— Cherchez les paramètres optimaux en nombre de processus par noeud et nombre de coeurs par processus. La différence est-elle importante ?
— Tracer les courbes de scalabilité. Conclusion ?

2.5 Bonus : recouvrement calcul/communications

Pour amortir le coût des communications, nous allons recouvrir le temps de communication avec des calculs.

— Pour recouvrir le temps de communication pour l'échange des halos, nous pouvons procéder en deux temps : calculer la partie intérieure de la grille (qui n'utilise pas les halos) pendant les communications, puis dans un deuxième temps calculer la partie de la grille qui utilise les halos. *Bien entendu, dans une version par blocs, il suffit de parcourir les blocs dans le bon ordre.*
— Pour recouvrir le temps de communication du test de convergence, nous pouvons utiliser la collective non-bloquante `MPI_Ireduce`. Quelle concession doit-on faire pour que ce soit possible ?
— Comment placer les threads sur les coeurs de calcul pour que la progression soit efficace ? Pourquoi ?
— Essayer avec les deux bibliothèques MPI OpenMPI 3 et Intel MPI et comparer les résultats obtenus. Conclusion ?

3 Aide-mémoire des paramètres de placement pour `mpirun` et `salloc`

Le placement des processus sur les noeuds résulte de l'interaction entre `salloc`, fourni par Slurm, et de `mpirun` (ou `mpiexec`) fourni par la bibliothèque MPI. En particulier, Slurm réserve les noeuds et coeurs, puis `mpirun` utilise ces ressources. Note : les choix par défaut (allocation de tâches sur des noeuds différents ou non par `salloc`, processus MPI par noeud ou par coeur par `mpirun`) dépend de la version des logiciels et de la configuration locale du cluster. Il faut donc *toujours* préciser les paramètres de placement pour un comportement déterministe.

`salloc`. Les paramètres essentiels sont :

- **`-n`** nombre de tâches (i.e. nombre de processus)
- **`-N`** nombre de noeuds
- **`-exclusive`** utilise tous les coeurs de chaque noeud.

`mpirun`. Les paramètres dépendent de la bibliothèque MPI et ne font pas partie du standard. Dans leurs dernières versions, OpenMPI et MPICH convergent vers une syntaxe commune (`-map-by`, `-bind-to`), incompatible avec leur ancienne syntaxe propre, mais utilisable conjointement.

- **`-map-by core|socket|node`** niveau d'assignation des processus (un processus par coeur/socket/noeud). D'autres niveaux sont disponibles. Veuillez consulter la documentation de `mpirun` pour plus de précisions.
- **`-npernode`** (seulement OpenMPI) nombre de processus par noeud. Équivalent à `--map-by ppr:`N`:node`
- **`-bind-to core|socket|numa|...`** niveau de binding des processus. D'autres niveaux sont disponibles. Par défaut, OpenMPI binde les processus. Vérifiez que le binding est compatible avec l'utilisation des threads à l'intérieur du processus.

`srun`. Il est possible de lancer une application MPI sur un cluster Slurm en une seule commande avec `srun` en lui passant l'option `-mpi=`*type* adaptée. Consultez la documentation de `srun` pour plus de précisions.

Exemples pratiques.

— Lancer 8 noeuds, un processus sur chaque coeur :
```
% salloc -p mistral -N 8 --exclusive mpirun --map-by core ./a.out
```
à raison de 20 coeurs par noeud, on obtient 160 processus.

— Lancer 160 processus, un processus par coeur :
```
% salloc -p mistral -n 160 mpirun --map-by core ./a.out
```
à raison de 20 coeurs par noeuds, le job occupe 8 noeuds complets.

— Lancer 8 noeuds, un processus par noeud :
```
% salloc -p mistral -N 8 --exclusive mpirun --map-by ppr:1:node ./a.out
```

— Lancer 4 processus par noeud, sur 8 noeuds :
```
% salloc -p mistral -N 8 --exclusive mpirun --map-by ppr:4:node ./a.out
```

— Astuce : en cas de doute sur l'affectation des processus, on peut vérifier en lançant la commande `hostname` :

```
% salloc -p mistral -N 3 --exclusive mpirun --map-by ppr:4:node hostname
salloc: Granted job allocation 428555
mistral05.formation.cluster
mistral05.formation.cluster
mistral05.formation.cluster
mistral05.formation.cluster
mistral02.formation.cluster
mistral02.formation.cluster
mistral02.formation.cluster
mistral03.formation.cluster
mistral03.formation.cluster
mistral03.formation.cluster
mistral02.formation.cluster
mistral03.formation.cluster
```

On a bien 4 processus par noeud, sur 3 noeuds différents. Notez que les affichages peuvent être entrelacés.

— Pour vérifier le binding, on peut utiliser le paramètre `-report-bindings`