Parallélisme à base de thread

(PBT 2022-2023)

TD5

Programmation par tâches

OpenMP

marc.perache@cea.fr
hugo.taboada@cea.fr
romain.pereira@cea.fr

Les objectifs de ce TD sont :

— la mise en pratique du modèle de programmation par tâche avec dépendances

— l'expérimentation et l'analyse de résultats

I Commandes utiles

Pour se connecter sur le calculateur

```
ssh -CXY hpc.pedago.ensiie.fr
```

Pour charger l'environnement `MPC` (commit `d564cbd87`) :

```
source /home/romain.pereira/install/mpc-2023/mpcvars.sh
```

Pour envoyer un ficher sur le calculateur

```
scp fichier romain.pereira@hpc.pedago.ensiie.fr:/chemin/vers/fichier
```

Pour récupérer un ficher du calculateur

```
scp romain.pereira@hpc.pedago.ensiie.fr:/mon/fichier/est/ici fichier
```

Pour soumettre un job

```
OMP_PLACES="cores(4)" OMP_NUM_THREADS=4 srun -p calcul -N 1 -n 1 --exclusive ./app
```

II Factorisation de Cholesky

Soit A une matrice symétrique définie positive. La factorisation de Cholesky consiste à déterminer une matrice triangulaire inférieure L telle que : $A = LL^T$.

Le dossier **cholesky/** contient une implémentation par bloc de cet algorithme. Pour vos tests, utilisez une matrice de taille $n = 2048$ et des blocs de taille $ts = 512$ - c'est à dire **./cholesky 2048 512**.

Q.1: Dans le fichier **main.c**, et dans la fonction `cholesky_par`, encapsulez les noyaux d'algèbres `potrf`, `trsm`, `gemm` et `syrk` dans des tâches OpenMP.

Attention : utilisez les dépendances pour garantir l'ordre d'exécution

Q.2: Mesurez et commentez les temps d'exécution de **./cholesky 2048 512** pour

$$\texttt{OMP\_NUM\_THREADS} \in 1, 2, 3, 4, 5, 6, 7, 8$$

Q.3: L'environnement `MPC` permet de générer le graphe de dépendance de tâche d'une instance d'exécution et son chemin critique. Pour cela :

- `OMP_NUM_THREADS=4 MPCFRAMEWORK_OMP_TASK_TRACE=true MPCFRAMEWORK_OMP_TASK_TRACEAUTO=true ./cholesky 2048 512`
- `python3 scripts/main.py -i [dossier-mpc-omp-trace]`
- `dot -Tpdf traces.dot > graph.pdf`

Vous pouvez également nommer vos tâches avec la macro `MPC_OMP_TASK_SET_LABEL`.
À l'aide de cet outil, affinez votre analyse sur la question précédente.
Fournir le fichier `graph.pdf` dans le rendu.

III Simulation problème à N corps

« Le problème à N corps consiste à résoudre les équations du mouvement de Newton de N corps interagissant gravitationnellement, connaissant leur masses ainsi que leurs positions et vitesses initiales. » - source

Illustration - https://www.youtube.com/watch?v=fit1uX1HIlc

Le dossier **nbody/** contient 4 versions d'une simulation du problème à N corps.

- **sequential** - Code séquentiel
- **parallel-for** - Code parallélisé avec les boucles OpenMP
- **taskloop** - Code parallélisé avec les tâches OpenMP, avec une synchronisation implicite à chaque boucle
- **task** - Code incomplet, contenant une version parallélisée avec les tâches OpenMP supprimant les synchronisations implicites en fin de boucle à l'aide de dépendances

1 Parallélisme de tâches

1.1 Compréhension du code

Q.4: Ouvrir les fichiers **simulate.c**, **forces.c** et **integrate.c** de la version **sequential**. Donner la complexité temporelle en fonction de $n = N\_BODIES$, des fonctions `update_forces`, `update_position`, `update_velocity` et `update_position`.

Q.5: Ouvrir la version **parallel-for** de ces 3 fichiers. Partant de la fonction `simulate`, décrire le flot d'exécution pour une exécution sur 4 threads. Vous pourrez illustrer votre description à l'aide d'un schéma en faisant figurer les appels de fonction, les boucles, et les points de synchronisation.

1.2 Étude d'extensibilité

On note
— $T(n)$ le temps d'exécution sur n threads.
— $s(n) = \frac{T(1)}{T(n)}$ le speed-up pour n threads.
— $eff(n) = \frac{s(1)}{n}$ l'efficacité sur n threads.

Q.6: Entrez la commande `make parallel-for` pour compiler cette version. Mesurer $T(n)$ et calculer $s(n)$ et $eff(n)$ pour $n \in 1, 2, 3, 4$ pour cette version de l'application.

Q.7: Dans la version **taskloop**, quelles significations donnez-vous aux variables `TASKS_PER_LOOP` et `BODIES_PER_TASK` ?

Q.8: La commande `make taskloop` permet de compiler cette version. On fixe maintenant `OMP_NUM_THREADS` $= 4$ pour le reste du TD. Pour chaque découpage `TASKS_PER_LOOP` $\in \{1, 4, 16\}$, mesurer le temps $T(4)$.

Q.9: Comparer et commenter les temps obtenus entre les versions **parallel-for** et **taskloop**. *Indice : intéressez vous au découpage des boucles en tâches*

1.3 Diagramme de Gantt

Un diagramme de Gantt est adapté à la visualisation de l'exécution d'une application à base de tâches. `MPC` propose une génération automatique de diagramme. En figure 1 se trouve un diagramme de Gantt de la version **taskloop** pour `TASKS_PER_LOOP` $= 4$. Le diagramme se situe dans le fichier **files/taskloop-4.json** et peut être visualiser dans `Chrome` en vous rendant à l'URL `about:tracing`.

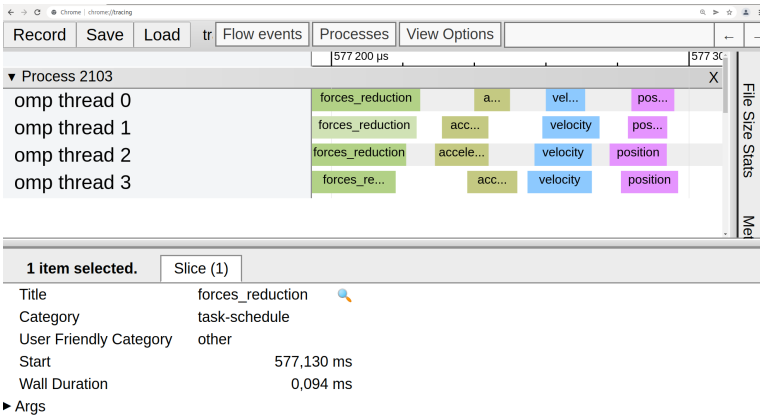

FIGURE 1 – Diagramme de Gantt généré avec MPC

Q.10: Ouvrir le diagramme de Gantt **files/taskloop-4.json** dans `Chrome`. Donner le temps d'exécution des 1ères tâches `forces_compute` et `acceleration` de chaque threads et commenter.

Q.11: **Bonus**. On note $F_M(n,t)$ le nombre de forces calculées par chaque tâche de la fonction `forces_compute` - pour un problème à n corps - pour un découpage en M tâches - les tâches étant numérotées $0 \leq t < M$ par leur ordre de création séquentiel.

- Donner une formule permettant de calculer $F_M(n,t)$.
- Comparer les temps mesurés à la question précédente avec $F_M(n,t)$ pour $M = 4$ et $t \in [0,3]$, et commenter.

Q.12: Générer le diagramme de la version **taskloop** pour `TASKS_PER_LOOP` $= 1$.

- `OMP_NUM_THREADS=4 MPCFRAMEWORK_OMP_TASK_TRACE=true MPCFRAMEWORK_OMP_TASK_TRACEAUTO=true ./taskloop 1`
- `python3 scripts/main.py -i [dossier-mpc-omp-trace] -o taskloop-1`
- `taskloop-1.json` contient un diagramme à ouvrir dans `Chrome`.

Fournir les fichiers `.json` dans le rendu.

Q.13: Même question pour `TASKS_PER_LOOP` $\in \{4, 16\}$.

Q.14: Visualisez les 3 diagrammes et affinez votre analyse **Q.9**. Vous pourrez également vous appuyer sur les métriques fournies par le script python.

2 Dépendances de tâches

Dans cette partie du TD, on souhaite supprimer les barrières de synchronisation de la version **taskloop**, tout en garantissant l'ordre d'exécution à l'aide de dépendances de tâches. On fixe `TASKS_PER_LOOP` $= 16$.

La version **task** contient une tentative ratée.

2.1 Correction du code

Q.15: Générer et visualiser le diagramme de Gantt de la version **task**.

— Zoomer sur les tâches `acceleration`, `velocity` et `position`
— Expliquer le problème

Q.16: Corriger le code en ajoutant des dépendances de tâches du fichier **integrate.c**.

Q.17: Générer et visualiser le diagramme de Gantt de la version corrigée.
— Zoomer sur les tâches `acceleration`, `velocity` et `position`
— Activer la visualisation des dépendances en cliquant sur « flow-events »
— Fournir une capture d'écran de cette section du diagramme de Gantt

2.2 Réduction des synchronisations

Q.18: Bonus (difficile). Modifier le fichier **forces.c** pour supprimer les barrières implicites de la directive `taskloop`, en utilisant la directive `task` et la clause `depend` pour garantir l'ordre d'exécution (ne pas utiliser d'autres mécanismes de synchronisation). Fournir le code dans le rendu.

Q.19: Le fichier **files/forces-bonus.c** contient une correction de la question précédente.
— Comparer la version **taskloop** à cette version **task** dépourvue de toute barrière.
— (Bonus) Si vous aviez réussi par vous même la question précédente, comparer votre implémentation à celle ci.
Il est attendu des comparaisons quantitatives : utilisez les métriques à votre disposition. Vous pouvez également illustrer vos propos à l'aide de figures.