Sorbonne Université

Polytech – EI-SE5

HPC – TP3 : Initiation à MPI

Dans ce TP nous allons nous focaliser sur l'écriture d'un code multi-cœur et multi-nœud. Pour y parvenir, nous allons utiliser la bibliothèque *Message Passing Interface* (MPI) qui est aujourd'hui le standard dans la plupart des supercalculateurs. Cette bibliothèque peut aussi bien être utilisée pour exploiter le parallélisme multi-cœur et que pour exploiter le parallélisme multi-nœud. MPI est basée sur le parallélisme dit « *Multiple Program Multiple Data* » (MPMD), en d'autres termes, plusieurs programmes (= processus) vont être instanciés sur des données différentes. Comme nous n'avons pas d'accès à un supercalculateur, les exercices de ce TP utiliserons les cœurs CPU de votre machine. Il est bon de noter que nous ne verrons pas les problèmes dus aux communications réseaux entre différents nœuds (nous resterons sur un nœud unique : votre machine).

1 Mise en bouche

1.1 Récupérer la version C++ d'EasyPAP

Il est nécessaire de re-télécharger EasyPAP sur votre disque avant de commencer le TP. Des modifications ont été apportées à EasyPAP par rapport au dernier TP. Notamment, dans cette nouvelle version, un nouveau *kernel* a été ajouté : `heat`. Vous devez cloner le dépôt Git d'EasyPAP :

```
git clone -b mpi --recursive https://gitlab.lip6.fr/cassagnea/easypap-se.git easypap-se-mpi
```

Pour compiler, rien n'a changé :

```
cd easypap-se-mpi; make -j4
```

1.2 Le noyau de calcul `heat`

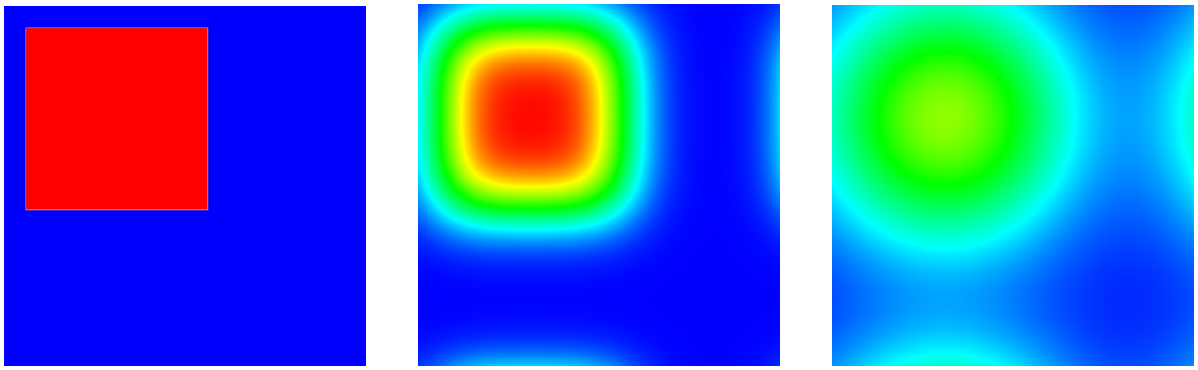
Dans ce TP nous allons utiliser EasyPAP et plus particulièrement nous allons travailler sur le noyau de calcul (ou *kernel*) `heat`. Ce noyau de calcul est une approximation de l'équation de la chaleur. C'est un *stencil* qui ressemble beaucoup au *kernel* `blur` que nous avons vu dans le TP1. Il y a cependant quelques différences :

- il y a 4 voisins (haut, bas, gauche, droite) plus le pixel courant qui sont considérés (contrairement aux 8 voisins du *kernel* `blur`),
- la gestion des bords est périodique, contrairement au *kernel* `blur` où elle était fixe,
- les calculs concernent une grille de valeurs flottantes (les températures) alors que dans le *kernel* `blur` on travaillait directement sur les pixels de l'image.

La Figure 1 illustre le *kernel* `heat`. La couleur rouge représente une très grande chaleur (valeur maximale : `1.f`), la couleur bleue représente une très faible chaleur (valeur minimale : `0.f`).
Le Code 1 présente l'implémentation actuelle de l'algorithme `heat`. Comme vous pouvez le voir, la fonction `heat_compute_seq` est appelée en premier (c'est le point d'entrée du *kernel* EasyPAP), puis la fonction `heat_do_tile_default` est appelée ensuite.
Vous pouvez dès à présent lancer le noyau de calcul dans EasyPAP :

```
./run -k heat -r 500 -i 5000 -si -v seq -wt default
```

La commande précédente permet de lancer 5000 itérations du *kernel* `heat`. Ici on affiche toutes les 500 itérations uniquement. `-si` permet d'afficher le numéro de l'itération courante et `-wt` permet de spécifier le type de tuile utilisée (ici `default`, voir ligne 10 dans le Code 1).
Dans ce TP nous allons écrire plusieurs variantes de la fonction `heat_compute_seq`. Pour cela vous serez invités à copier coller cette même fonction et à remplacer la fin du nom (ici `seq`) par le nouveau nom. Vous pourrez ensuite comparer les temps d'exécution des différentes versions en utilisant les paramètres `-v` et `-wt` d'EasyPAP.



(a) Conditions initiales. (b) Après 20000 itérations. (c) Après 100000 itérations.

FIGURE 1 – Illustration du noyau de calcul `heat` (équation de la chaleur simplifiée).

```c
static float *tgrid0; // grille de températures numéro 0
static float *tgrid1; // grille de températures numéro 1
void heat_init() {
  // allocations de `tgrid0` et `tgrid1` (`DIM * DIM` nombres flottants en simple précision)
  // conditions initiales de la grille :
  //   - un gros point chaud en haut à gauche,
  //   - tout le reste est très froid.
}

int heat_do_tile_default(int x, int y, int width, int height) {
  float c, l, r, b, t; // center, left, right, bottom and top values
  for (int i = y; i < y + height; i++)
    for (int j = x; j < x + width; j++) {
      c =                 tgrid0[(i + 0) * DIM + (j + 0)];
      l = (j - 1) >=  0 ? tgrid0[(i + 0) * DIM + (j - 1)] : tgrid0[(  i + 0) * DIM + (DIM - 1)];
      r = (j + 1) < DIM ? tgrid0[(i + 0) * DIM + (j + 1)] : tgrid0[(  i + 0) * DIM + (      0)];
      b = (i - 1) >=  0 ? tgrid0[(i - 1) * DIM + (j + 0)] : tgrid0[(DIM - 1) * DIM + (  j + 0)];
      t = (i + 1) < DIM ? tgrid0[(i + 1) * DIM + (j + 0)] : tgrid0[(      0) * DIM + (  j + 0)];
      tgrid1[i * DIM + j] = (c + l + r + t + b) * 0.2f;
    }

  return 0;
}

unsigned heat_compute_seq(unsigned nb_iter) {
  for (unsigned it = 1; it <= nb_iter; it++) {
    do_tile(0, 0, DIM, DIM, 0);
    // échange tgrid0 <-> tgrid1
    float *tmp = tgrid1;
    tgrid1 = tgrid0;
    tgrid0 = tmp;
  }

  // pour l'affichage, conversion de la température en pixel
  for (unsigned i = 0; i < DIM; i++)
    for (unsigned j = 0; j < DIM; j++)
      cur_img(i, j) = heat_to_rgb(tgrid0[i * DIM + j]);

  return 0;
}
```

Code source 1 – Code du noyau de calcul `heat` dans EASYPAP.

À la différence des précédents TP, ici les calculs ne sont pas directement faits sur les pixels d'une image. L'image permet simplement de vérifier nos résultats. La fonction `heat_to_rgb` (voir ligne 37 dans le Code 1) permet de convertir une température comprise entre 0 et 1 vers une couleur.
Vous noterez aussi que les tableaux `tgrid0` et `tgrid1` (lignes 1-2 dans le Code 1) sont sur une seule dimension. Cependant ils représentent bien une grille 2D. Simplement, les accès sont linéarisés (voir lignes 14-19 dans le Code 1). Par exemple, ligne 14 : `tgrid0[i * DIM + j]`, `i` représente y et `j` représente x.

Travail à faire Lancer le *kernel* `heat` sur votre machine, comprendre le code.

1.3 Temps d'exécution en fonction des options d'optimisation

La compilation du code EasyPAP est définie dans le ficher `Makefile` à la racine. Les options de compilations sont définies à la ligne 57 :

```
CFLAGS += -O3 -march=native -Wall -Wno-unused-function -std=c++11
```

Comme vous pouvez le voir l'option `-O3` est présente, c'est l'option d'optimisation de niveau 3.

Travail à faire Compiler le code avec les différents niveaux d'optimisation disponibles (`-O0`, `-O1`, `-O2` et `-O3`) puis comparer et noter les différents temps d'exécution pour 5000 itérations. Pour voir le temps d'exécution il faut utiliser l'option `--no-display` d'EasyPAP.

Pour la suite du TP, il faut utiliser le niveau d'optimisation « trois » (`-O3` = toutes les optimisations).

2 Modifier le code pour diminuer le temps d'exécution

2.1 Gestion alternative des bords (`seq_bv2`)

Dans cette partie nous allons modifier la gestion des bords pour supprimer les conditions ternaires lignes 15-18 dans le Code 1. Pour y parvenir, nous allons allouer une grille de taille `(DIM + 2) * (DIM + 2)`. Une partie de cette modification du code vous est donnée dans le Code 2 où les emplacement avec écrit `/* TODO */` sont à compléter. Il faut bien garder en tête que les dimensions de l'image à afficher sont toujours `DIM * DIM`. Par contre, les dimensions de la grille des températures sont maintenant `(DIM + 2) * (DIM + 2)`.

Travail à faire 1 Allouer correctement les buffers `tgrid0` et `tgrid1` dans la fonction `heat_init_seq_bv2` (lignes 2-3 dans le Code 2). Initialiser les grilles des températures à 0 (ligne 5 dans le Code 2).

Travail à faire 2 Initialiser correctement le point chaud dans la grille des températures `tgrid0` (lignes 15 dans le Code 2). Pensez bien que la première ligne de l'image correspond maintenant à la deuxième ligne dans la grille des températures. De même, la première colonne de l'image correspond à la deuxième colonne dans la grille des températures.

Travail à faire 3 Calculer correctement les indices dans la fonction `heat_do_tile_bv2` (lignes 22-27 dans le Code 2). Pensez bien que la première ligne de l'image correspond maintenant à la deuxième ligne dans la grille des températures. De même, la première colonne de l'image correspond à la deuxième colonne dans la grille des températures.

Travail à faire 4 Calculer correctement les indices dans la fonction `heat_compute_seq_bv2` (lignes 52 dans le Code 2). Pensez bien que la première ligne de l'image correspond maintenant à la deuxième ligne dans la grille des températures. De même, la première colonne de l'image correspond à la deuxième colonne dans la grille des températures.

```
void heat_init_seq_bv2() {
  tgrid0 = (float *)malloc(sizeof(float) * /* TODO */);
  tgrid1 = (float *)malloc(sizeof(float) * /* TODO */);

  for (unsigned i = 0; i < /* TODO */ ; i++) {
    tgrid0[i] = 0.f;
    tgrid1[i] = 0.f;
  }

  unsigned ss = DIM / 2;  // square size
  unsigned shift = DIM / 16;

  for (unsigned i = 0; i < ss; i++)
    for (unsigned j = 0; j < ss; j++)
      tgrid0[((/* TODO */) + shift) * (/* TODO */) + (/* TODO */) + shift] = 1.f;
}

int heat_do_tile_bv2(int x, int y, int width, int height) {
  float c,l,r,b,t;
  for (int i = y; i < y + height; i++)
    for (int j = x; j < x + width; j++) {
      c = tgrid0[(/* TODO */ + 0) * (/* TODO */) + (/* TODO */ + 0)];
      l = tgrid0[(/* TODO */ + 0) * (/* TODO */) + (/* TODO */ - 1)];
      r = tgrid0[(/* TODO */ + 0) * (/* TODO */) + (/* TODO */ + 1)];
      b = tgrid0[(/* TODO */ - 1) * (/* TODO */) + (/* TODO */ + 0)];
      t = tgrid0[(/* TODO */ + 1) * (/* TODO */) + (/* TODO */ + 0)];
      tgrid1[(/* TODO */) * (/* TODO */) + (/* TODO */)] = (c + l + r + t + b) * 0.2f;
    }

  return 0;
}

unsigned heat_compute_seq_bv2(unsigned nb_iter) {
  for (unsigned it = 1; it <= nb_iter; it++) {
    // copie des bord
    for (int i = 0; i < (int)DIM; i++) {
      tgrid0[       0  * (DIM + 2) +   i + 1] = tgrid0[   DIM  * (DIM + 2) + i + 1]; // t <- b
      tgrid0[(DIM + 1) * (DIM + 2) +   i + 1] = tgrid0[     1  * (DIM + 2) + i + 1]; // b <- t
      tgrid0[  (i + 1) * (DIM + 2)       + 0] = tgrid0[(i + 1) * (DIM + 2) +   DIM]; // l <- r
      tgrid0[  (i + 1) * (DIM + 2) + DIM + 1] = tgrid0[(i + 1) * (DIM + 2) +     1]; // r <- l
    }

    do_tile(0, 0, DIM, DIM, 0);

    float *tmp = tgrid1;
    tgrid1 = tgrid0;
    tgrid0 = tmp;
  }

  for (unsigned i = 0; i < DIM; i++)
    for (unsigned j = 0; j < DIM; j++)
      cur_img(i, j) = heat_to_rgb(tgrid0[(/* TODO */) * (/* TODO */) + (/* TODO */)]);

  return 0;
}
```

Code source 2 – Nouvelle gestion des bords.

Travail à faire 5 Maintenant il ne devrait plus y avoir de `/* TODO */` dans votre code. Vérifier que votre code fonctionne correctement, pour cela vous pouvez utiliser la commande suivante :

```
./run -k heat -v seq_bv2 -si -wt bv2 -r 500
```

Une fois que votre code fonctionne correctement vous pouvez recueillir son temps d'exécution avec la commande suivante :

```
./run -k heat -v seq_bv2 -wt bv2 -i 5000 --no-display
```

Comparer ce temps avec le code initial.

Tips Dans la fonction `heat_compute_seq_bv2`, la recopie des bords vous est donnée (lignes 35-41), vous pouvez vous en inspirer pour les autres questions de cet exercice.

2.2 MPI avec 2 processus : communications point à point bloquantes (`mpi_v0`)

Nous allons maintenant mettre en place une version parallèle MPI avec 2 processus. Nous allons découper notre domaine horizontalement : chaque processus sera en charge de calculer une bande horizontale (= un bloc horizontal) de même taille mais à différentes positions y dans l'image.

```
1  static int mpi_y = -1, mpi_h = -1, mpi_rank = -1, mpi_size = -1;
2
3  void heat_init_mpi_v0(void) {
4    heat_soluce_init_seq_bv2();
5    easypap_check_mpi();  // check if MPI was correctly configured
6
7    MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);
8    MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);
9
10   mpi_y = mpi_rank * (DIM / /* TODO */);
11   mpi_h = (DIM / /* TODO */);
12 }
13
14 unsigned heat_compute_mpi_v0(unsigned nb_iter) {
15   for (unsigned it = 1; it <= nb_iter; it++) {
16     // ...
17     do_tile(0, mpi_y, DIM, mpi_h, 0);
18
19     float *tmp = tgrid1;
20     tgrid1 = tgrid0;
21     tgrid0 = tmp;
22
23     /* TODO: écrire les communications MPI ici */
24   }
25
26   for (int i = /* TODO */; i < /* TODO */; i++)
27     for (int j = 0; j < (int)DIM; j++)
28       cur_img(i, j) = heat_to_rgb(tgrid0[(i + 1) * (DIM + 2) + (j + 1)]);
29
30   return 0;
31 }
```

Code source 3 – Implémentation MPI avec 2 processus (communications point à point bloquantes).

Le Code 3 propose une implémentation à trous de ce qui est attendu. Il y a plusieurs variables globales :

- `mpi_size` : le nombre de processus MPI (initialisé L8 dans le Code 3),
- `mpi_rank` : l'identifiant du processus MPI courant (initialisé L7 dans le Code 3), c'est un nombre compris entre 0 et `mpi_size` -1,
- `mpi_y` : l'indice y du début du bloc horizontal dans l'image (initialisé L10 dans le Code 3),
- `mpi_h` : le nombre de lignes à calculer dans un bloc horizontal (initialisé L11 dans le Code 3).

Travail à faire 1 Compléter le calcul de `mpi_y` et `mpi_h` aux lignes L10-11 dans le Code 3.

Travail à faire 2 Copier uniquement les lignes sur lesquelles on calcule dans le processus MPI courant (L26 dans le Code 3).

Vous devriez maintenant être capable d'exécuter le code avec la commande suivante :

```
./run -k heat -v mpi_v0 -wt bv2 --mpirun "-np 2" -r 500 --debug-flags M --monitoring
```

`--mpirun "-np 2"` permet de spécifier 2 processus MPI, `--debug-flags M` permet d'afficher une fenêtre par processus MPI et `--monitoring` permet de voir quel cœur calcule sur quelle zone de l'image.
En l'état le code devrait afficher quelque chose mais il n'est pas fonctionnel. Cela est du au fait que les processus MPI n'échangent pas de données, or ils ont besoin de le faire. En effet, chaque processus a besoin de connaître les valeurs de la ligne du dessus et les valeurs de la ligne du dessous. **Il faut donc que les processus communiquent ces lignes.**
Pour y parvenir vous aurez besoin des communications point à point bloquantes suivantes :

- `int MPI_Send(const void *mess, int count, MPI_Datatype type, int dest, int tag, MPI_Comm c);`
- `int MPI_Recv(void *mess, int count, MPI_Datatype type, int src, int tag, MPI_Comm c, MPI_Status *s);`

Travail à faire 3 Ajouter les communications MPI nécessaires pour qu'après chaque itération, les processus échangent leur lignes du dessus et du dessous (à la ligne 23 dans le Code 3).

Si vous avez correctement implémenté les communications MPI, le code devrait maintenant calculer correctement. Le seul souci c'est que aucun des processus n'est capable d'afficher l'image complète. Pour y parvenir il faut rassembler tous les blocs horizontaux (ici 2 blocs uniquement) dans le processus MPI 0. Il existe une routine MPI dédiée à ce genre d'opérations : `MPI_Gather` (voir Code 4).

```
1 int MPI_Gather(const void *message_emis, int longueur_message_emis,
2                MPI_Datatype type_message_emis, void *message_recu,
3                int longueur_message_recu, MPI_Datatype type_message_recu,
4                int rang_dest, MPI_Comm comm);
```

Code source 4 – Déclaration de la routine `MPI_Gather`.

Travail à faire 4 Sachant que quand vous utilisez la macro `cur_img(i, j)` vous écrivez en fait dans la variable globale `image[i * DIM + j]`, utilisez la routine `MPI_Gather` pour reconstruire l'image complète dans la processus MPI 0.

Travail à faire 5 Vérifier que votre code fonctionne correctement, pour cela vous pouvez utiliser la commande suivante :

```
./run -k heat -v mpi_v0 -si -wt bv2 --mpirun "-np 2" -r 500
```

Une fois que votre code fonctionne correctement vous pouvez recueillir son temps d'exécution avec la commande suivante :

```
./run -k heat -v mpi_v0 -wt bv2 --mpirun "-np 2" -i 5000 --no-display
```

Comparer ce temps avec les implémentations précédentes.

2.3 Généralisation à 2^n processus (`mpi_v1`)

Travail à faire 1 En partant du code précédent, généralisez pour 2^n processus.

Travail à faire 2 Remplacer les couples `MPI_Send`/`MPI_Recv` par un seul appel à la routine `MPI_Sendrecv`.

Travail à faire 3 Notez les temps d'exécution pour 1 processus, puis 2, puis 4, puis 8 processus. Dessinez une courbe de *speedup* en fonction du nombre de cœurs utilisés.

2.4 Communications point à point non-bloquantes (`mpi_v2`)

```
1 int MPI_Isend(const void*valeurs, int taille, MPI_Datatype type_message, int dest,
2               int etiquette, MPI_Comm comm, MPI_Request *req);
3 int MPI_Irecv(void *valeurs, int taille, MPI_Datatype type_message, int source,
4               int etiquette, MPI_Comm comm, MPI_Request *req);
5 int MPI_Wait (MPI_Request *req, MPI_Status *statut);
```

Code source 5 – Communications non-bloquantes : `MPI_Isend`, `MPI_Irecv` et `MPI_Wait`.

Travail à faire 1 Le Code 5 présente les déclarations des routines MPI pour implémenter une version non-bloquante. Proposez une nouvelle version avec ce nouveau type de communications MPI.

Travail à faire 2 Comparer l'implémentation avec communications non-bloquantes avec la version précédente (communications bloquantes).