Chapitre 6 : MPI

6.1 Une tâche MPI s'exécute sur deux noeuds. L'idée est d'échanger de l'information entre le noeud 0 et le noeud 1. Cependant, les tâches bloquent et donc ne se terminent jamais. Quel est le problème?

```
switch (rank) {
  case 0:
    MPI_Send(msg0, MSG_SIZE, MPI_INT, 1, 1, MPI_COMM_WORLD);
    MPI_Recv(msgr, MSG_SIZE, MPI_INT, 1, 1, MPI_COMM_WORLD,&status);
    break;
  case 1:
    MPI_Send(msg1, MSG_SIZE, MPI_INT, 0, 1, MPI_COMM_WORLD);
    MPI_Recv(msgr, MSG_SIZE, MPI_INT, 0, 1, MPI_COMM_WORLD,&status);
    break;
  default:
    break;
}
```

Les deux processus exécutent en même temps le send et bloqueront s'il n'y a pas assez de tampon disponible, espérant que l'autre processus se mette en réception. De manière plus générale, il faut premièrement éviter les blocages, ensuite éviter la sérialisation (chacun sauf le noeud 0 attend la réception du précédent avant d'envoyer au suivant, ce qui crée une longue chaîne). Ensuite, l'envoi avec tampon coûte possiblement une copie et l'envoi asynchrone demande une structure requête, un surcoût faible mais non nul. Les échanges sont un mécanisme intéressant (Send-Receive) et l'alternance noeuds pairs envoient, impairs reçoivent puis l'inverse est un bon moyen de transmettre l'information sans avoir de longues chaînes de dépendances.

6.2 Un programme MPI est réparti sur un grand nombre de noeuds, chaque noeud contenant un vecteur qui donne la chaleur pour chaque point en X, le long d'une ligne horizontale qui correspond à un Y donné. Pour simuler la diffusion de chaleur dans une plaque, chaque noeud doit obtenir le vecteur de ses deux voisins, celui avec Y juste en-dessous et celui avec Y juste au-dessus. Chaque noeud a une position, rank, qui va de 0 à size - 1 et qui correspond à la coordonnée en Y. La fonction `itère_chaleur` reçoit en argument rp (rangée précédente), le vecteur de chaleur du noeud de rang inférieur (ou NULL pour la première rangée), rs (rangée suivante), le vecteur de chaleur du noeud de rang supérieur (ou NULL pour la dernière rangée), et rc (rangée courante), le vecteur de chaleur du noeud courant. Cette fonction modifie l'argument rc en y plaçant les nouvelles valeurs de chaleur. La boucle de calcul de cette application est fournie. Complétez cette

boucle avec les énoncés (et commentaires appropriés) permettant d'effectuer efficacement ce travail.

```
int rank, size;
MPI_status status;
float rp[2048], rs[2048], rc[2048];
...
for(t = 0 ; t < max_time; t++) {
  /* Obtenir les valeurs des rangées précédentes et suivantes
     et leur fournir les nôtres. Nous sommes la rangée (noeud) rank,
     comprise entre 0 et (size - 1) et size est pair et > 1 */

  /* complétez cette section*/
  ...
  itere_chaleur(rp, rs, rc)
  ...
}
```

Plusieurs stratégies peuvent être utilisées pour réaliser ce travail. Une réalisation relativement simple est d'initier de manière asynchrone toutes les communications (envois de rc et réception de rp et rs). Il suffit ensuite d'attendre que les requêtes soient complétées. Il peut être un peu plus efficace de commander chaque opération sans aller de manière asynchrone, à condition de bien planifier toutes les communications pour éviter les blocages ou les dépendances en chaîne. Par exemple, à chaque envoi sur un noeud doit correspondre l'appel pour la réception sur le noeud de destination.

Une situation à éviter est la suivante:

```
for(t = 0 ; t < max_time; t++) {
  if(rank < (size - 1) {
    MPI_Sendrecv(rc, 2048, MPI_FLOAT, rank+1, 1, rs, 2048, MPI_FLOAT,
        rank+1, 0, MPI_COMM_WORLD);
  }
  if(rank > 0) {
    MPI_Sendrecv(rc, 2048, MPI_FLOAT, rank-1, 0, rp, 2048, MPI_FLOAT,
        rank-1, 1, MPI_COMM_WORLD);
  }
  itere_chaleur(rp, rs, rc)
}
```

Dans ce cas, tous les noeuds sauf le dernier commencent par échanger (du noeud courant, rang inférieur) avec le noeud de rang plus élevé. Il n'y a personne pour recevoir, sauf le dernier noeud qui passe directement au deuxième if et peut faire l'échange avec le noeud de rang size - 2. Pendant cet échange, tous les autres noeuds sont bloqués. Une fois que le noeud de rang size - 2 a terminé son échange avec size - 1, il passe au second if et peut compléter l'échange avec le

noeud de rang size - 3. Ceci continue ainsi de manière sérielle et tout l'avantage du parallélisme est perdu. Pour éviter ce problème, une stratégie très courante est de séparer les noeuds entre noeuds pairs et impairs. Dans un premier temps, les noeuds pairs échangent rs et rc avec le noeud impair de rang supérieur, et ensuite ils échangent rp et rc avec le noeud impair de rang inférieur.

```
for(t = 0 ; t < max_time; t++) {
  if(rank == 0) {
    /* Noeud 0, rangée paire, échanger avec le suivant seulement */
    MPI_Sendrecv(rc, 2048, MPI_FLOAT, rank+1, 1, rs, 2048, MPI_FLOAT,
        rank+1, 0, MPI_COMM_WORLD);
    itere_chaleur(NULL, rs, rc)
  }
  else if(rank == (size - 1)) {
    MPI_Sendrecv(rc, 2048, MPI_FLOAT, rank-1, 0, rp, 2048, MPI_FLOAT,
        rank-1, 1, MPI_COMM_WORLD);
    itere_chaleur(rp, NULL, rc)
  } else {
    if(rank % 2) {
      /*rangée impaire, échanger avec précédent puis avec suivant*/
      MPI_Sendrecv(rc, 2048, MPI_FLOAT, rank-1, 0, rp, 2048, MPI_FLOAT,
          rank-1, 1, MPI_COMM_WORLD);
      MPI_Sendrecv(rc, 2048, MPI_FLOAT, rank+1, 2, rs, 2048, MPI_FLOAT,
          rank+1, 3, MPI_COMM_WORLD);
    }
    else {
      /* Rangée paire, échanger avec suivant puis avec précédent */
      MPI_Sendrecv(rc, 2048, MPI_FLOAT, rank+1, 1, rs, 2048, MPI_FLOAT,
          rank+1, 0, MPI_COMM_WORLD);
      MPI_Sendrecv(rc, 2048, MPI_FLOAT, rank-1, 3, rs, 2048, MPI_FLOAT,
          rank-1, 2, MPI_COMM_WORLD);
    }
    itere_chaleur(rp, rs, rc)
  }
}
```

6.3 Chaque tâche MPI contient une matrice diagonale (tous les éléments sur la diagonale sont à i pour la tâche i et les autres à 0). Chaque tâche MPI doit envoyer le contenu de cette diagonale à la tâche de rang 0 qui place le contenu de la diagonale de la tâche i dans la rangée i de sa matrice. Le contenu de la matrice est ensuite imprimé par la tâche 0.

```
#include <stdio.h>
#include <mpi.h>

int main (int argc, char* argv[]) {
  int rank, size;
  MPI_Datatype staircase;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
```

```
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  float m[size][size];
  for (int i=0; i < size; i++) {
    for (int j=0; j < size; j++) {
      m[i][j] = (i==j) ? rank : 0;
    }
  }

  MPI_Type_vector(size, 1, size+1, MPI_FLOAT, &staircase)
  MPI_Type_commit(&staircase)
  MPI_Gather(m, 1, staircase, m, size, MPI_FLOAT, 0, MPI_COMM_WORLD);
  MPI_Type_free(&staircase);

  if (rank == 0)
    for (int i=0; i < size; i++) {
      for (int j=0; j < size; j++) {
        printf("%g ", m[i][j]);
        printf("\n");
      }
    }
  }

  MPI_Finalize()
  return 0;
}
```