Correction du Partiel de PP, 2019-2020

Polio, Promo 2021

23 novembre 2019

Durée : 1h30. Ce partiel est découpé en 5 exercices.
Correction par Polio (promo 2021), avec l'énoncé transcrit avant chaque réponse.

1 Exercice I : Petit programme MPI

Soit la section de code suivante exécutée par tous les processus d'un programme MPI (il y a au moins 2 processus) :

```
int rang, som;

MPI_Comm_rank(MPI_COMM_WORLD, &rang);

som = 0;
if (rang == 0)
  som = 1;
som += 2;
if (rang == 1)
  som += 3;
```

Suite à l'exécution de ces lignes, quelle est la valeur de la variable `som` en fonction du rang MPI (justifiez brièvement) ?

Tous les processus passent par `som = 0` et `som += 2`. Ainsi les processus dont le rang est supérieur ou égal à `2` auront pour valeur finale `som: 2`.
Rang 0 : Passe par `som = 1` après le `som = 0`, la valeur finale sera donc `som: 3`.
Rang 1 : Passe par `som += 3` après le `som += 2`, la valeur finale sera donc `som: 5`.

2 Exercice II : Cherchez l'erreur

Pour chacune des sections de code suivantes (numérotées **a** à **d**), dites :
- si elle fait l'object d'un risque de bloquage ("*deadlock*") ;
- ou bien si les contenus des messages risquent d'être erronés ;

Justifiez vos choix et apportez une correction pour que la section soit correcte.
Dans tout ce qui suit, on désigne par `rang` le rang du processus MPI et par `P` la taille du communicateur `MPI_COMM_WORLD`. On suppose que le programme a correctement initialisé MPI.

2.1 Section A

On veut calculer le nombre total d'éléments nuls dans un tableau distribué.

```
int i, Nnuls = 0, Nnuls_tot;
/* tabl est un tableau distribue sur tous les processus et prealablement initialise
 * N = dimension locale du tableau
 */
for (i = 0; i < N; i++)
  if (tabl[i] == 0)
    Nnuls++;

if (Nnuls > 0) {
  MPI_Allreduce(&Nnuls, &Nnuls_tot, 1, MPI_INT, MPI_SUM MPI_COMM_WORLD);
  printf("Nb d'elements nuls dans le tableau distribue : %d\n", Nnuls_tot);
}
```

`MPI_Allreduce` est une opération collective : tous les processus compris dans `MPI_COMM_WORLD` doivent l'appeler. Ici, seuls les processus ayant trouvé au moins un élément nul l'appellent, il y aura donc un *deadlock*.

Pour corriger, il faut supprimer le `if (Nnuls > 0)`.

2.2 Section B

Le processus 0 veut distribuer un entier différent à chacun des autres processus

```
MPI_Request req[P];
int entier;

if (rang == 0)
{
  for (i = 1; i < P; i++)
  {
    entier = i * i;
    MPI_Isend(&entier, 1, MPI_INT, i, 1000, MPI_COMM_WORLD, req + (i - 1));
  }
  MPI_Waitall(P - 1, req, MPI_STATUSES_IGNORE);
}
else
{
  MPI_Recv(&entier, 1, MPI_INT, 0, 1000, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

La variable `entier` est utilisée comme stockage pour chaque `MPI_Isend(&entier, [...])`. Si un de ces envois n'est pas complété avant que `entier` ne soit réécrit par le tour de boucle suivant, la valeur envoyée sera alors erronée.

Une correction consiste à déclarer un tableau `int entiers[P-1]` et à remplacer le contenu de la boucle par :

```
entiers[i - 1] = i * i;
MPI_Isend(entiers + (i - 1), 1, MPI_INT, i, 1000, MPI_COMM_WORLD, req + (i - 1));
```

Il faut toutefois laisser la variable `entier` afin qu'elle puisse être utilisée pour la réception pour les autres processus.

2.3 Section C

But : faire circuler un message (le contenu de `m`) dans un anneau initialisé par le processus `0`. Le message circule dans le sens croissants des rangs modulo `P`.

```
int droite, int gauche, m;

if (rang == 0)
{
  m = 0;
}

droite = (rang + 1) % P;
gauche = (rang - 1) % P;

MPI_Recv(&m, 1, MPI_INT, gauche, 1000, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
MPI_Send(&m, 1, MPI_INT, droite, 1000, MPI_COMM_WORLD);
```

`MPI_Recv` (et `MPI_Send`) est bloquante, hors tous les processus commencent par l'appeler : aucun `MPI_Send` ne sera fait. Il y a donc un *deadlock*.

Une correction possible consiste à spécialiser le code pour le processus 0 :

```
if (rang == 0)
{
  MPI_Send(&m, 1, MPI_INT, droite, 1000, MPI_COMM_WORLD);
  MPI_Recv(&m, 1, MPI_INT, gauche, 1000, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
else
{
  MPI_Recv(&m, 1, MPI_INT, gauche, 1000, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  MPI_Send(&m, 1, MPI_INT, droite, 1000, MPI_COMM_WORLD);
}
```

2.4 Section D

Le processus 0 veut envoyer le message `m = 100` au processus 1.

```c
if (rang == 0)
{
  m = 100;
  MPI_Request req;
  MPI_Isend(&m, 1, MPI_INT, 1, 1000, MPI_COMM_WORLD, &req);
}
else if (rang == 1)
{
  MPI_Recv(&m, 1, MPI_INT, 0, 1000, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

`MPI_Isend` est non bloquant : le message risque de ne jamais être envoyé et donc de bloquer le processus 1 indéfiniment.

Pour corriger il est possible d'ajouter la ligne `MPI_Wait(&req, MPI_STATUS_IGNORE);` en dessous de la ligne d'envoi.

3 Exercice III : Quelle est la bonne valeur ?

Soit la section de code suivante exécutée par tous les processus d'un programme MPI (il y a au moins 3 processus) :

```c
int rang, val, valA, valB;
MPI_Request req[2];

MPI_Comm_rank(MPI_COMM_WORLD, &rang);

if (rang == 0)
{
  MPI_Irecv(&valA, 1, MPI_INT, MPI_ANY_SOURCE, 1000, MPI_COMM_WORLD, req + 0);
  MPI_Irecv(&valB, 1, MPI_INT, MPI_ANY_SOURCE, 1111, MPI_COMM_WORLD, req + 1);
  MPI_Waitall(2, req, MPI_STATUSES_IGNORE);
}
else if (rang == 1)
{
  val = rang;
  MPI_Send(&val, 1, MPI_INT, 0, 1111, MPI_COMM_WORLD);
}
else if (rang == 2)
{
  val = rang;
  MPI_Send(&val, 1, MPI_INT, 0, 1000, MPI_COMM_WORLD);
}
```

Question : Suite à l'exécution de ces lignes, quelles sont les valeurs des variables `valA` et `valB` du processus de rang 0 (justifiez) ?

La différenciation sur les réceptions du processus 0 se fait sur les tags (`1000` et `1111`).
Le processus 1 envoie sur le tag 1111 donc `valB = 1`. Le processus 0 envoie sur le tag 1000 donc `valA = 2`.

4 Exercice IV : Implémentation d'un `allgather`

Voici le code de la fonction **`allgather`** et son programme principal : complétez-le.

```c
void allgather(int in, int *out)
{
  int rang, P;

  /* 1. A COMPLETER: il faut affecter le nombre de processus dans `P` */
  /* 2. A COMPLETER: il faut affecter la variable `rang` */

  if (rang == 0)
  {
    MPI_Status sta;
    out[0] = in;
    for (int p = 1; p < P; p++)
    {
      int val;
      /* 3. A COMPLETER: reception dans `val` de la valeur envoyee par le processus de rang `p` */
      /* 4. A COMPLETER: mise a jour du tableau `out` */
    }
  }
  else
  {
    /* 5. A COMPLETER: envoi de `in` au processus 0 */
  }

  /* 6. A COMPLETER: le processus 0 diffuse `out` aux autres processus */
}

int main(int argc, char *argv[])
{
  int rang, P, *out;

  /* 7. A COMPLETER */
  /* 8. A COMPLETER: il faut affecter le nombre de processus dans `P` */
  /* 9. A COMPLETER: il faut affecter la variable `rang` */

  out = (int *)malloc(P * sizeof(int));
  allgather(rang, out);

  for (int p = 0; p < P; p++)
    printf("P%d, out[%d] = %d\n", rang, P, out[p]);
  free(out);

  /* 10. A COMPLETER */

  return 0;
}
```

Soit la fonction `void allgather(int in, int *out)` appelée en même temps par tous les processus MPI. Elle collecte dans le tableau `out` le contenu de toutes les variables `in` de tous les processus `MPI`. Tous les processus doivent récupérer ensuite les résultats. Le tableau `out` est préalable alloué au nombre total de processus.
La fonction `allgather` est implémentée dans le bloc de code ci-dessus, ainsi qu'un programme principal qui utilise cette fonction. Cependant, ce code n'est pas complet.
Travail à faire : compléter les lignes `/* À COMPLÉTER */` pour que la fonction `allgather` et le programme principal soient corrects.

— 1 et 8 : `MPI_Comm_size(MPI_COMM_WORLD, &P);`
— 2 et 9 : `MPI_Comm_rank(MPI_COMM_WORLD, &rang);`
— 3 : `MPI_Recv(&val, 1, MPI_INT, p, 1000, MPI_COMM_WORLD, &sta);`
— 4 : `out[p] = val;`
— 5 : `MPI_Send(&in, 1, MPI_INT, 0, 1000, MPI_COMM_WORLD);`
— 6 : `MPI_Bcast(out, P, MPI_INT, 0, MPI_COMM_WORLD);`
— 7 : `MPI_Init(&argc, &argv);`
— 8 : `MPI_Finalize();`

5 Exercice V : Recouvrement

Définition de la fonction `collect_trait` :

```c
void collect_trait(int *tab, int N) {
  int rank, P, k;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &P);

  if (rank == 0) {
    MPI_status sta;
    int *buf2 = (int *)malloc(N * sizeof(int));
    trait(tab, N);
    for (k = 1; k < P; k++) {
      MPI_Recv(buf2, N, MPI_INT, k, 1000, MPI_COMM_WORLD, &sta);
      trait(buf2, N);
    }
    free(buf2);
  }
  else {
    MPI_Send(tab, N, MPI_INT, 0, 1000, MPI_COMM_WORLD);
  }
}
```

On considère une fonction `collect_trait` dont la définition se trouve ci-dessus. Elle est utilisée dans les conditions suivantes :

1. Cette fonction est appelée par tous les processus MPI (comme une opération collective) ;
2. Chaque processus a pré-alloué et rempli son propre tableau de `N` entiers ;
3. La valeur `N` est identique pour tous les processus MPI ;
4. La fonction `trait(int *tab, int N)` (appelée par le processus 0) effectue un traitement coûteux à partir des données du tableau `tab` passé en argument, sans modifier le contenu de ses données.

Question 1 : Quelle est la modification à apporter à la réception pour que le processus 0 reçoive un message à partir de n'importe quel processus ?

Il faut remplacer le `k` utilisé pour indiquer depuis quel processus réceptionner par `MPI_ANY_SOURCE`.

Question 2 : Expliquez brièvement pourquoi la réception (par le processus 0) n'est pas recouverte par le traitement `trait`.

`MPI_Recv` est bloquant : la réception est donc forcément effectuée avant que le traitement des données reçues ne s'effectue.

Question 3 : Modifiez la section de code exécutée par le processus 0 pour recouvrir la réception par le traitement `trait` tout en recevant les messages dans n'importe quel ordre.
Indication : utilisez deux buffers de `N` entiers, un pour recevoir les données, l'autre pour traiter les données déjà reçues.

```c
MPI_status sta;
// Nouvelles variables utilisees pour le recouvrement
MPI_Request req;
int n;
int *buf2 = (int *)malloc(N * sizeof(int));
int *recvbuf = (int *)malloc(N * sizeof(int));
// Copie du tableau du rang 0 pour qu'il soit traite pendant la premiere reception
for (n = 0; n < P; n++) buf2[n] = tab[n];
// Recouvrement avec reception depuis toutes les sources
for (k = 1; k < P; k++) {
  MPI_Irecv(recvbuf, N, MPI_INT, MPI_ANY_SOURCE, 1000, MPI_COMM_WORLD, &req);
  // Traitement du tableau precedent pendant la reception du suivant
  trait(buf2, N);
  MPI_Wait(&req, &sta);
  for (n = 0; n < P; n++) buf2[n] = recvbuf[n]; // Copie
}
// Traitement du dernier tableau qui n'a pas ete fait apres la derniere copie: il faut donc le
    faire apres la boucle
trait(buf2, N);
free(buf2);
free(recvbuf); // Ne pas oublier de liberer le nouveau buffer
```