Diffusion avec MPI

Résumé: Dans ce TD, nous allons nous familiariser avec les fonctions de communications de MPI et essayer d'écrire nous-même la fonction de diffusion à partir d'envois et de receptions simples. Nous allons également nous rendre compte de la difficulté de l'écriture d'une opération de diffusion non bloquante avec MPI et ainsi souligner bon nombre de limitations de MPI.

1 Un mot d'histoire

Tout d'abord, il convient de rappeler quelques généralités sur ce qu'est MPI et sur ce que ce n'est pas. MPI n'est pas une bibliothèque de communication développée par une université ou une entreprise : c'est un standard qui est né au début des années 90 pour répondre à un besoin de clarification. En effet, à cette époque, la seule façon de faire du calcul parallèle efficace consistait à acheter une grosse machine parallèle propriétaire. Ces dernières étaient généralement livrées avec leur propre bibliothèque de communication qui n'était que rarement compatible avec celle de la machine précédente et jamais avec celle des concurrents. Il était donc très difficile de maintenir un programme à jour et un gros travail était nécessaire à chaque fois que l'on souhaitait changer de machine. Le standard MPI est donc né de la collaboration entre des universitaires et des industriels de tous domaines scientifiques. Cependant, si ce standard a permit de résoudre la majorité des problèmes que l'on pouvait avoir au moment de sa création, il souffre désormais d'un bon nombre de limitations et d'une inadéquation aux plateformes de calcul actuelles.

En effet, la mode actuelle en matière de calcul distribué et parallèle est à l'interconnection, via des liens à très haut débit, de grands centres de calculs disséminés à l'échelle d'un pays, d'un continent voire du monde. C'est ce que l'on appelle le *meta-computing*. Personne ne sait si on pourra un jour exploiter correctement de telle plateformes mais, dixit Al. Geist (responsable du projet PVM) il y a quelques jours[1] : "Il y a de toutes façons tellement d'argent investi dans ce domaine que cela aboutira forcément à quelque chose".

Mais revenons à MPI. Un des gros problème auquel on se heurte dès que l'on essaie d'utiliser un programme MPI sur une plateforme de *meta-computing* est que les bibliothèques MPI d'un centre de calcul à l'autre ne sont pas forcément identiques et rien dans le standard n'oblige une bibliothèque à perdre en performance pour être compatible avec une bibliothèque concurrente. À cela s'ajoutent la difficulté de déploiement d'un tel programme, la nécessité de prendre en compte (autant au niveau algorithmique qu'au niveau système) l'hétérogénéité des processeurs et des réseaux, le coût énorme des synchronisations et donc des opérations bloquantes, la non-tolérance aux pannes,... Même s'il serait plaisant de pouvoir programmer et utiliser ces plateformes comme une simple station de travail, nous en somme très loin actuellement et leur complexité semble rendre cette entreprise un peu utopique.

Ce portrait de MPI et du calcul parallèle actuel peut sembler un peu pessimiste mais tous ces projets ont aussi donné naissance à de très bonnes choses. Même si MPI n'est plus adapté aux plateformes de calcul telle qu'elles sont envisagées actuellement, l'aventure MPI est quand même un succès étant donné qu'elle a permis à bon nombre de personnes non informaticiennes de développer des programmes efficaces pour les plateformes de calcul parallèle classiques et de collaborer grâce à des codes portables. L'émulation créée par ces projets ambitieux de *global-computing* permet à bon nombre de scientifiques non informaticiens de résoudre des problèmes qu'ils n'auraient jamais pu espérer résoudre il y a de cela quelques années. Enfin, même si tous ces projets semblent un peu fous et extrêmement ambitieux, il est indéniable que la communauté scientifique a un besoin toujours croissant de puissance de calcul et de communication et que nous ne sommes actuellement pas en mesure de répondre à leurs besoins.

Un nouveau standard MPI-2 a été mis en place il y a quelques années et résout certains des problèmes précédemment évoqués. Il existe cependant encore très peu de bibliothèques mettant en œuvre l'intégralité du standard MPI-2.

[1] Et oui, on voit du beau monde dans les confs...

2 Introduction à l'utilisation de MPI

2.1 Initialisation et terminaison du programme

Un programme MPI commence en général par un appel de la fonction `MPI_Init` dont le prototypage est le suivant :

```
int MPI_Init(int *argc, char ***argv)
```

Cette fonction initialise les connections MPI en fonction des arguments passés à votre programme. C'est pourquoi avant de lire les arguments de votre programme, il convient de faire un appel à cette fonction.

Un programme MPI se termine généralement par l'appel de la fonction `MPI_Finalize`. Tous les processus doivent appeler ce programme avant leur terminaison. Cette opération est bloquante ne termine que lorsque toutes les opérations de communications en cours ou en attente sont terminées.

Il peut être utile de savoir combien de processus participent au calcul et quel numéro on a. Les fonctions `MPI_Comm_size` et `MPI_Comm_rank` dont le prototype est le suivant. Un `MPI_Comm` est un groupe de processus MPI. Pour l'instant, nous n'utiliserons que le groupe prédéfini `MPI_COMM_WORLD` qui regroupe l'intégralité des processus MPI participant au calcul.

```
int MPI_Comm_size ( MPI_Comm comm, int *size )
int MPI_Comm_rank ( MPI_Comm comm, int *rank )
```

2.2 Communications bloquantes

Les envois et les réceptions bloquantes se font grâce aux fonctions `MPI_Send` et `MPI_Recv` dont le prototype est le suivant :

```
int MPI_Send( void *buf, int count, MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm )
int MPI_Recv( void *buf, int count, MPI_Datatype datatype, int source,
              int tag, MPI_Comm comm, MPI_Status *status )
```

Quelques explications :

– `buf` représente l'adresse du buffer d'émission (pour `MPI_Send`) ou du buffer de réception (pour `MPI_Recv`).

– `count` est le nombre d'éléments à envoyer ou à recevoir.

– `datatype` est le type des éléments que l'on va envoyer (ou recevoir). On doit préciser le type des éléments car MPI peut effectuer des conversions de type si les architectures cibles ne représentent pas les objets de la même manière (*big indian/little indian*...). Il est possible de créer des types MPI de façon à faciliter la programmation mais nous n'utiliserons pour l'instant que le type `MPI_INT`.

– `dest` et `source` sont respectivement les indices des processeurs destinataires et sources mais vous l'aviez probablement deviné tout seul.

– `tag` est un nombre qui sert généralement à identifier un type de message. Choisissez-en un et essayez de vous y tenir...

– `comm` est le groupe de processus impliqués dans la communication. Comme cela a été signalé précédemment, nous nous contenterons de `MPI_COMM_WORLD`.

– `status` est un objet permettant de récupérer des informations sur la communication et ne nous servira pas pour l'instant.

Enfin une autre fonction qui s'avère souvent utile : la barrière de synchronisation.

```
int MPI_Barrier ( MPI_Comm comm )
```

Nous allons essayer de simuler la fonction `MPI_Bcast` dont le prototype est le suivant.

```
int MPI_Bcast ( void *buffer, int count, MPI_Datatype datatype, int root,
                MPI_Comm comm )
```

Cette fonction est bloquante. Lorsqu'un programme alterne des phases de calcul et des phases de communications, il y a donc intérêt à ce que l'équilibrage de charge soit parfait...

2.3 Communications non bloquantes

Les fonctions d'envoi et de réception non bloquantes `MPI_Isend` et `MPI_Irecv` ont le prototype suivant :

```
int MPI_Isend( void *buf, int count, MPI_Datatype datatype, int dest, int tag,
               MPI_Comm comm, MPI_Request *request )
int MPI_Irecv( void *buf, int count, MPI_Datatype datatype, int source,
               int tag, MPI_Comm comm, MPI_Request *request )
```

`request` est un *"numéro"* de communication. Cela permet de savoir si une communication est terminée ou pas, notamment grâce à la fonction `MPI_Test` ou à la fonction `MPI_Wait`.

```
int MPI_Test ( MPI_Request *request, int *flag, MPI_Status *status)
int MPI_Wait ( MPI_Request *request, MPI_Status *status)
```

`flag` reçoit la valeur vrai si la communication est effectivement terminée.

2.4 Lancement du programme

Un programme MPI minimal (`simple.c`) ainsi qu'un makefile permettant de compiler (en tapant `make`...) et de lancer des programmes MPI (en tapant `make run`) sont disponibles dans http://www-id.imag.fr/Laboratoire/Membres/Legrand_Arnaud/algopar/MPI/src/. Théoriquement, si vos clés ssh ont été posées, le lancement des applications ne devrait pas poser de problème. La bibliothèque MPI que nous allons utiliser s'appelle MPICH et est disponible sur une grande variété de plateformes.

3 Diffusion en MPI

▷ **Question 1.** Écrivez un programme où le processeur 0 diffuse aux autres processeurs un tableau d'entiers à l'aide de la fonction `MPI_Bcast`. Rajoutez un appel à la fonction `long_computation` juste après la diffusion et mesurez le temps nécessaire à ces opérations grâce à la fonction `ms_time`[2].

▷ **Question 2.** Recommencez en écrivant vous même la diffusion (en anneau) à l'aide d'opérations bloquantes. Que pensez-vous des performances, notamment quand vous faites varier le nombre de processus ?

▷ **Question 3.** Remplacez maintenant vos émissions bloquantes par des émissions non bloquantes. Que pensez-vous des performances ?

▷ **Question 4.** Rajoutez une barrière de synchronisation juste avant l'appel à `long_computation`. Que pensez-vous des performances ? Avez-vous une explication à proposer ?

▷ **Question 5.** Essayer de remédier au problème en utilisant autre chose qu'une barrière de synchronisation.

L'objectif de ce TD était de mettre en valeur un certain nombre de limitations de MPI. Il ne faut quand même pas oublier que ces bibliothèques sont quand même très efficaces sur des plateformes homogènes, qu'elles offrent une quantité d'opérations globales impressionnantes ainsi qu'un haut niveau d'abstraction, ce qui permet de développer très rapidement un programme parallèle.

Les fichiers C solutions au problème sont disponibles dans http://www-id.imag.fr/Laboratoire/Membres/Legrand_Arnaud (`broadcast1.c`, `broadcast2.c`, `broadcast3.c`, `broadcast4.c`, `broadcast5.c`).

[2] ces deux fonctions sont disponibles dans le programme MPI minimal évoqué précédemment

4 Réponses aux exercices

▷ **Question 1.** La Figure 1 représente les modifications à apporter au programme `simple.c` pour effectuer une diffusion utilisant la fonction `MPI_Bcast`

Utilisation de la fonction MPI_Bcast

```
 6  #include <stdlib.h>                                          #include <stdlib.h>
 7  #include <stdio.h>                                           #include <stdio.h>
 8  #include <unistd.h>                                          #include <unistd.h>
 9  #include <mpi.h>                                             #include <mpi.h>
10  #include <sys/time.h>                                        #include <sys/time.h>
11
12  #define BUFLEN 65536                                         #define BUFLEN 65536
13
14  long ms_time()                                               long ms_time()
15  {                                                            {
16    struct timeval tv;                                           struct timeval tv;
17
18    gettimeofday(&tv, NULL);                                     gettimeofday(&tv, NULL);
19    return(tv.tv_sec*1000 + tv.tv_usec/1000);                    return(tv.tv_sec*1000 + tv.tv_usec/1000);
20  }                                                            }
21
22  void long_computation()                                      void long_computation()
23  {                                                            {
24     long now=ms_time();                                          long now=ms_time();
25     while(ms_time()<now+4000)                                    while(ms_time()<now+4000)
26       {                                                            {
27       }                                                            }
28  }                                                            }
29
30  int main(int argc, char **argv)                              int main(int argc, char **argv)
31  {                                                            {
32    int nb_proc;                                                 int nb_proc;
33    int my_id;                                                   int my_id;
34                                                           >    int next;
35    int buffer[BUFLEN];                                          int buffer[BUFLEN];
36
37                                                           >    long start,end;
38                                                           >
39    MPI_Init(&argc,&argv);                                       MPI_Init(&argc,&argv);
40    MPI_Comm_size(MPI_COMM_WORLD,&nb_proc);                      MPI_Comm_size(MPI_COMM_WORLD,&nb_proc);
41    MPI_Comm_rank(MPI_COMM_WORLD,&my_id);                        MPI_Comm_rank(MPI_COMM_WORLD,&my_id);
42
43    printf("Je suis le processus P%d et nous sommes %d au total.\n",  |    if (my_id == nb_proc-1)
44           my_id, nb_proc);                                  |      next = 0;
45                                                           >    else
46                                                           >      next = my_id+1;
47                                                           >
48                                                           >    if (my_id == 0) start = ms_time();
49                                                           >
50                                                           >    if (my_id == 0)
51                                                           >      {
52                                                           >        int i;
53                                                           >        for(i=0; i<BUFLEN; i++) buffer[i]=i;
54                                                           >      }
55                                                           >
56                                                           >    MPI_Bcast(buffer, BUFLEN, MPI_INT, 0, MPI_COMM_WORLD);
57                                                           >    long_computation();
58
59    MPI_Finalize();                                              MPI_Finalize();
60                                                           >    if (my_id == 0) end = ms_time();
61                                                           >    if (my_id == 0) printf("Ça m'a prit %ld ms\n",(end-start));
62
63    return (0);                                                  return (0);
64  }                                                            }
```

Fig. 1 – Utilisation de la fonction `MPI_Bcast`

▷ **Question 2.** La Figure 2 représente les modifications à apporter au programme précédent pour simuler l'appel à `MPI_Bcast` par communications bloquantes. Au fur et a mesure que le nombre de processeur augmente, les performances se dégradent. Ceci est du au fait que la bibliothèque MPICH utilise un arbre de diffusion pour mettre en œuvre la fonction `MPI_Bcast`.

▷ **Question 3.** Les performances sont, contrairement à ce que l'on pourrait croire, complètement lamentables. La question suivante devrait éclaircir les choses.

Utilisation des fonctions de communications bloquantes

```
 6  #include <stdlib.h>
 7  #include <stdio.h>
 8  #include <unistd.h>
 9  #include <mpi.h>
10  #include <sys/time.h>
11
12  #define BUFLEN 65536
13
14  long ms_time()
15  {
16    struct timeval tv;
17
18    gettimeofday(&tv, NULL);
19    return(tv.tv_sec*1000 + tv.tv_usec/1000);
20  }
21
22  void long_computation()
23  {
24     long now=ms_time();
25     while(ms_time()<now+4000)
26       {
27       }
28  }
29
30  int main(int argc, char **argv)
31  {
32    int nb_proc;
33    int my_id;
34    int next;
35    int buffer[BUFLEN];
36
37
38    long start,end;
39
40    MPI_Init(&argc,&argv);
41    MPI_Comm_size(MPI_COMM_WORLD,&nb_proc);
42    MPI_Comm_rank(MPI_COMM_WORLD,&my_id);
43
44    if (my_id == nb_proc-1)
45      next = 0;
46    else
47      next = my_id+1;
48
49    if (my_id == 0) start = ms_time();
50
51    if (my_id == 0)
52      {
53        int i;
54        for(i=0; i<BUFLEN; i++) buffer[i]=i;
55
56
57
58
59
60
61
62
63
64
65      }
66
67    MPI_Bcast(buffer, BUFLEN, MPI_INT, 0, MPI_COMM_WORLD);
68    long_computation();
69
70    MPI_Finalize();
71    if (my_id == 0) end = ms_time();
72    if (my_id == 0) printf("Ça m'a prit %ld ms\n",(end-start));
73
74    return (0);
75  }
```

```
    #include <stdlib.h>
    #include <stdio.h>
    #include <unistd.h>
    #include <mpi.h>
    #include <sys/time.h>

    #define BUFLEN 65536

    long ms_time()
    {
      struct timeval tv;

      gettimeofday(&tv, NULL);
      return(tv.tv_sec*1000 + tv.tv_usec/1000);
    }

    void long_computation()
    {
       long now=ms_time();
       while(ms_time()<now+4000)
         {
         }
    }

    int main(int argc, char **argv)
    {
      int nb_proc;
      int my_id;
      int next;
      int buffer[BUFLEN];
>     MPI_Status status;

      long start,end;

      MPI_Init(&argc,&argv);
      MPI_Comm_size(MPI_COMM_WORLD,&nb_proc);
      MPI_Comm_rank(MPI_COMM_WORLD,&my_id);

      if (my_id == nb_proc-1)
        next = 0;
      else
        next = my_id+1;

      if (my_id == 0) start = ms_time();

      if (my_id == 0)
        {
          int i;
          for(i=0; i<BUFLEN; i++) buffer[i]=i;
>
>         MPI_Send(buffer, BUFLEN, MPI_INT, next, 666, MPI_COMM_WORLD);
>       }
>     else
>       {
>         MPI_Recv(buffer, BUFLEN, MPI_INT, MPI_ANY_SOURCE, 666, MPI_COMM_WORLD,
>                  &status);
>
>         if (my_id != nb_proc-1)
>           MPI_Send(buffer, BUFLEN, MPI_INT, next, 666, MPI_COMM_WORLD);
        }

<---
      long_computation();

      MPI_Finalize();
      if (my_id == 0) end = ms_time();
       if (my_id == 0) printf("Ça m'a prit %ld ms\n",(end-start));

      return (0);
    }
```

Fig. 2 – Utilisation des fonctions de communications bloquantes

Utilisation des fonctions de communications asynchrones

```
 6  #include <stdlib.h>
 7  #include <stdio.h>
 8  #include <unistd.h>
 9  #include <mpi.h>
10  #include <sys/time.h>
11
12  #define BUFLEN 65536
13
14  long ms_time()
15  {
16    struct timeval tv;
17
18    gettimeofday(&tv, NULL);
19    return(tv.tv_sec*1000 + tv.tv_usec/1000);
20  }
21
22  void long_computation()
23  {
24     long now=ms_time();
25     while(ms_time()<now+4000)
26       {
27       }
28  }
29
30  int main(int argc, char **argv)
31  {
32    int nb_proc;
33    int my_id;
34    int next;
35    int buffer[BUFLEN];
36    MPI_Status status;
37
38
39    long start,end;
40
41    MPI_Init(&argc,&argv);
42    MPI_Comm_size(MPI_COMM_WORLD,&nb_proc);
43    MPI_Comm_rank(MPI_COMM_WORLD,&my_id);
44
45    if (my_id == nb_proc-1)
46      next = 0;
47    else
48      next = my_id+1;
49
50    if (my_id == 0) start = ms_time();
51
52    if (my_id == 0)
53      {
54        int i;
55        for(i=0; i<BUFLEN; i++) buffer[i]=i;
56
57        MPI_Send(buffer, BUFLEN, MPI_INT, next, 666, MPI_COMM_WORLD);
58      }
59    else
60      {
61        MPI_Recv(buffer, BUFLEN, MPI_INT, MPI_ANY_SOURCE, 666, MPI_COMM_WORLD,
62                 &status);
63
64        if (my_id != nb_proc-1)
65          MPI_Send(buffer, BUFLEN, MPI_INT, next, 666, MPI_COMM_WORLD);
66      }
67
68    long_computation();
69
70    MPI_Finalize();
71    if (my_id == 0) end = ms_time();
72    if (my_id == 0) printf("Ça m'a prit %ld ms\n",(end-start));
73
74    return (0);
75  }
```

```
   #include <stdlib.h>
   #include <stdio.h>
   #include <unistd.h>
   #include <mpi.h>
   #include <sys/time.h>

   #define BUFLEN 65536

   long ms_time()
   {
     struct timeval tv;

     gettimeofday(&tv, NULL);
     return(tv.tv_sec*1000 + tv.tv_usec/1000);
   }

   void long_computation()
   {
      long now=ms_time();
      while(ms_time()<now+4000)
        {
        }
   }

   int main(int argc, char **argv)
   {
     int nb_proc;
     int my_id;
     int next;
     int buffer[BUFLEN];
     MPI_Status status;
>    MPI_Request request;

     long start,end;

     MPI_Init(&argc,&argv);
     MPI_Comm_size(MPI_COMM_WORLD,&nb_proc);
     MPI_Comm_rank(MPI_COMM_WORLD,&my_id);

     if (my_id == nb_proc-1)
       next = 0;
     else
       next = my_id+1;

     if (my_id == 0) start = ms_time();

     if (my_id == 0)
       {
         int i;
         for(i=0; i<BUFLEN; i++) buffer[i]=i;

|        MPI_Isend(buffer, BUFLEN, MPI_INT, next, 666, MPI_COMM_WORLD,&request);
       }
     else
       {
         MPI_Recv(buffer, BUFLEN, MPI_INT, MPI_ANY_SOURCE, 666, MPI_COMM_WORLD,
                  &status);

         if (my_id != nb_proc-1)
|          MPI_Isend(buffer, BUFLEN, MPI_INT, next, 666, MPI_COMM_WORLD,&request)
       }

     long_computation();

     MPI_Finalize();
     if (my_id == 0) end = ms_time();
      if (my_id == 0) printf("Ça m'a prit %ld ms\n",(end-start));

     return (0);
   }
```

Fig. 3 – Utilisation des fonctions de communications asynchrones

▷ **Question 4.** Encore une fois, les performances sont contraires à l'intuition. Ces problèmes sont dus au fait que cette bibliothèque MPI n'est pas *multi-threadée*. Les communications ne sont traitées que lorsque l'on rentre dans une fonction MPI. L'appel à la fonction `MPI_ISend` ne fait que préparer la communication à venir. Pour que la communication puisse se faire effectivement, il faut que les deux parties soient prêtes et il suffit donc en général de faire un appel à une fonction MPI quelconque pour déclancher la communication.

▷ **Question 5.** On peut également résoudre le problème en truffant les fonctions de calcul d'appels à la fonction `MPI_Test` (voir Figure 4). Imaginez maintenant que vous souhaitiez remplacer la réception bloquante par une réception asynchrone et vous aurez alors une idée des problèmes auxquels ont du faire face les développeurs de bibliothèques reposant sur MPI.

Truffer les calculs de MPI_Test

```
 6  #include <stdlib.h>
 7  #include <stdio.h>
 8  #include <unistd.h>
 9  #include <mpi.h>
10  #include <sys/time.h>
11
12  #define BUFLEN 65536
13
14  long ms_time()
15  {
16    struct timeval tv;
17
18    gettimeofday(&tv, NULL);
19    return(tv.tv_sec*1000 + tv.tv_usec/1000);
20  }
21
22  void long_computation()
23  {
24
25
26
27     long now=ms_time();
28     while(ms_time()<now+4000)
29       {
30
31       }
32  }
33
34  int main(int argc, char **argv)
35  {
36    int nb_proc;
37    int my_id;
38    int next;
39    int buffer[BUFLEN];
40    MPI_Status status;
41    MPI_Request request;
42
43    long start,end;
44
45    MPI_Init(&argc,&argv);
46    MPI_Comm_size(MPI_COMM_WORLD,&nb_proc);
47    MPI_Comm_rank(MPI_COMM_WORLD,&my_id);
48
49    if (my_id == nb_proc-1)
50      next = 0;
51    else
52      next = my_id+1;
53
54    if (my_id == 0) start = ms_time();
55
56    if (my_id == 0)
57      {
58        int i;
59        for(i=0; i<BUFLEN; i++) buffer[i]=i;
60
61        MPI_Isend(buffer, BUFLEN, MPI_INT, next, 666, MPI_COMM_WORLD,&request);
62      }
63    else
64      {
65        MPI_Recv(buffer, BUFLEN, MPI_INT, MPI_ANY_SOURCE, 666, MPI_COMM_WORLD,
66                 &status);
67
68        if (my_id != nb_proc-1)
69          MPI_Isend(buffer, BUFLEN, MPI_INT, next, 666, MPI_COMM_WORLD,&request)
70      }
71
72    long_computation();
73
74    MPI_Finalize();
75    if (my_id == 0) end = ms_time();
76    if (my_id == 0) printf("Ça m'a prit %ld ms\n",(end-start));
77
78    return (0);
79  }
```

```
   #include <stdlib.h>
   #include <stdio.h>
   #include <unistd.h>
   #include <mpi.h>
   #include <sys/time.h>

   #define BUFLEN 65536

   long ms_time()
   {
     struct timeval tv;

     gettimeofday(&tv, NULL);
     return(tv.tv_sec*1000 + tv.tv_usec/1000);
   }

|  void long_computation(MPI_Request *request)
   {
>    MPI_Status status;
>    int flag;
>
     long now=ms_time();
     while(ms_time()<now+4000)
       {
>        MPI_Test(request,&flag,&status);
       }
   }

   int main(int argc, char **argv)
   {
     int nb_proc;
     int my_id;
     int next;
     int buffer[BUFLEN];
     MPI_Status status;
     MPI_Request request;

     long start,end;

     MPI_Init(&argc,&argv);
     MPI_Comm_size(MPI_COMM_WORLD,&nb_proc);
     MPI_Comm_rank(MPI_COMM_WORLD,&my_id);

     if (my_id == nb_proc-1)
       next = 0;
     else
       next = my_id+1;

     if (my_id == 0) start = ms_time();

     if (my_id == 0)
       {
         int i;
         for(i=0; i<BUFLEN; i++) buffer[i]=i;

         MPI_Isend(buffer, BUFLEN, MPI_INT, next, 666, MPI_COMM_WORLD,&request);
       }
     else
       {
         MPI_Recv(buffer, BUFLEN, MPI_INT, MPI_ANY_SOURCE, 666, MPI_COMM_WORLD,
                  &status);

         if (my_id != nb_proc-1)
           MPI_Isend(buffer, BUFLEN, MPI_INT, next, 666, MPI_COMM_WORLD,&request)
       }

|    long_computation(&request);

     MPI_Finalize();
     if (my_id == 0) end = ms_time();
      if (my_id == 0) printf("Ça m'a prit %ld ms\n",(end-start));

     return (0);
   }
```

Fig. 4 – Truffer les calculs de `MPI_Test`