CRSNG-FONCER

Programme de formation en Génie Par Simulation

Introduction à la programmation parallèle

Avec OpenMP et MPI

Guillaume Emond

Polytechnique Montréal
Montréal, QC

Novembre 2016

Table des matières

Table des matières

Loi d'Amdahl

La loi d'Amdahl permet de calculer l'accélération théorique(A), obtenu en parallélisant un programme séquentiel, selon le nombre de processeurs (n) utilisés et la fraction parallélisable du code(p).

$$A = \frac{1}{(1-p) + p/n} \tag{1}$$

Il s'agit toujours d'une borne supérieur de l'accélération réelle.

API de parallélisation

3 niveaux de parallélisation

- Thread : PThread,PTh,TBB, **openMP**
- Processus : **MPI**, PVM, LINDA
- GPU : Cuda, openCL

Pourquoi openMP ou MPI ?

- OpenMP et MPI sont simples d'utilisation
- Ce sont les api les plus répendus dans le monde scientifique
- Gratuits et portables

Processus

Un processus est un programme en cours d'exécution qui est constitué de :

- un numéro d'identification
- un espace d'adressage
- un état (Élu, Prêt, Bloqué)
- une priorité
- une liste d'instructions (Compteur Ordinal)
- descripteurs de fichier

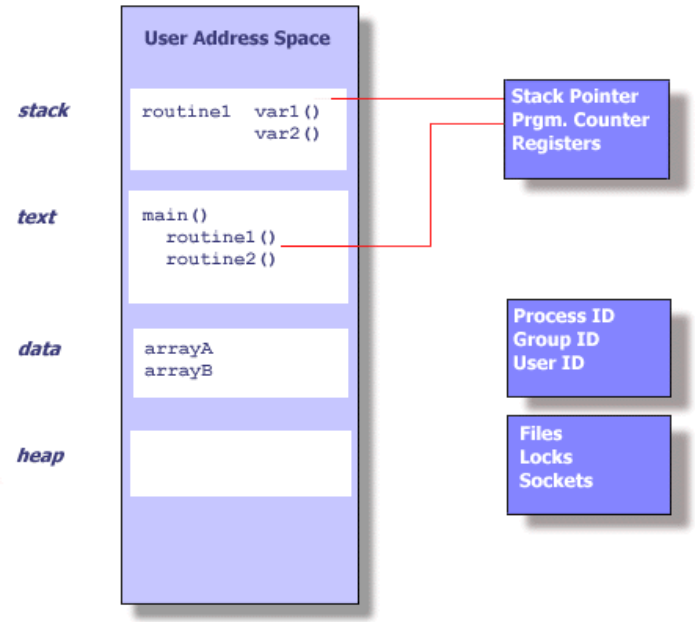


FIGURE – Espace d'adressage d'un processus

Thread

- Un même processus a la possibilité d'avoir plus d'un fil d'exécution (stack).
- Ces fils d'exécution partagent les ressources du processus.
- Chaque thread possède :
  - un identificateur
  - sa pile d'exécution
  - son compteur ordinal
  - un état

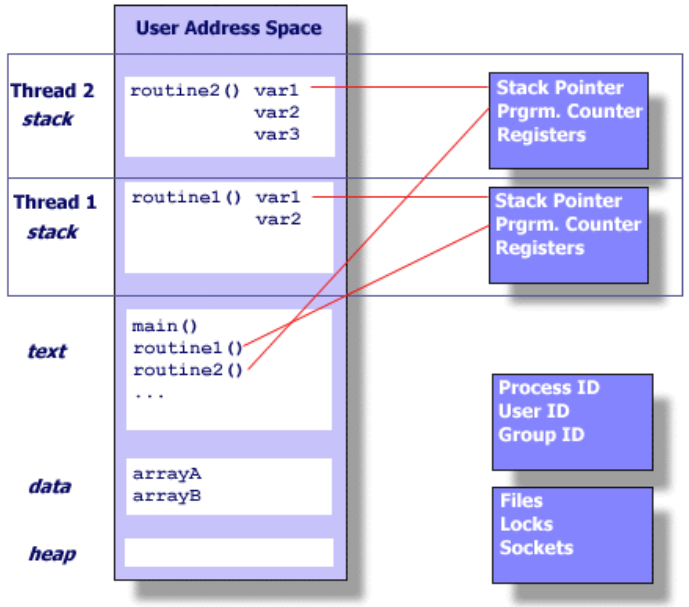


FIGURE – Multithreaded process

Avantages threads vs processus

- Le partage des ressources entre threads est beaucoup plus facile et efficace que pour les processus.
- Problèmes de concurrence des ressources.
- La création et les changements de contexte entre threads est beaucoup plus rapide.
- Les threads ne s'appliquent pas aux architectures à mémoire distribuée.

Table des matières

Open Multi-Processing

- Programmation en mémoire partagée
- API C,C++ et Fortran.
- Disponible sur Linux, Unix, Mac OS X, Microsoft Windows et Solaris (www.openmp.org)
- Permet de progressivement paralléliser un programme séquentiel sans restructurer l'entièreté du programme.

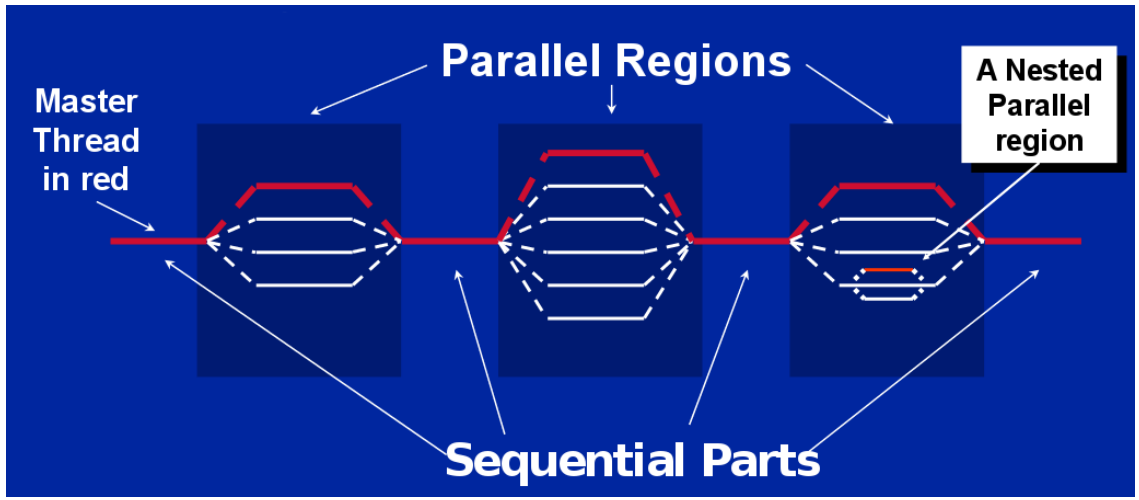
Quelques modèles

Modèle d'éxecution

- "Fork-Join"
- Open mp consiste à insérer des blocs parallèles dans un programme séquentiel.
- Possibilité d'avoir des régions parallèles imbriquées.

Modèle de mémoire

- Variables partagées.
- Variables privées.



Syntaxe

C/C++ : #pragma omp directive [clause]
Fortran : !$OMP DIRECTIVE [CLAUSE]
: !$OMP END DIRECTIVE [CLAUSE]

- Directives : parallel, for (parallel for), sections (parallel sections), single, critical, atomic, master, target, simd...
- Clauses : shared, private, firstprivate, lasprivate, default, reduction, copyin, if, ordered, schedule, nowait, safelen, linear, aligned, collasped, device, map...
- Les clauses disponibles pour chaque directive peuvent changer.

Fonctions de support : threads

- omp_(set/get)_num_threads : spécifie/retourne le nombre de threads.
- omp_get_max_threads : retourne le nombre maximal possible de threads.
- omp_get_thread_num : retourne le numéro du thread courant.
- omp_get_num_proc : retourne le nombre de processeurs disponibles.
- omp_in_parallel : pour savoir si on se trouve dans une région parallèle.
- omp_get_wtime : temps écoulé en secondes par thread.
- omp_get_wtick : temps écoulé en cycle d'horloge par thread.

Fonctions de support : threads

- omp_(set/get)_nested : permission d'avoir des régions parallèles imbriquées.
- omp_(set/get)_set_max_active_levels : profondeur maximale d'imbrication
- omp_get_active_level : retourne la profondeur courante de la région parallèle imbriquée.
- omp_stacksize : retourne la taille des piles pour les threads.

Table des matières

Parallel

- Création des fils d'exécution au début de la section parallèle.
- Le nombre de threads créés est généralement spécifié dans la variable d'environnement *OMP_NUM_THREAD* ou par défault (2 x core).
- Il y a une synchronisation à la fin de la section.
- Chaque thread exécute les instructions dans le bloc parallèle mais agit différement selon son identificateur.

Exemple : Parallel

```
// Hello World parallel

#include <stdio.h>
#include <omp.h>
int main( int argc, char **argv )
{
  #pragma omp parallel
  {
       int rank= omp_get_thread_num();
       int size= omp_get_num_threads();
       printf( "Hello world! I'm %d of %d\n",rank, size );
  }
  return 0;
}
```

export OMP_NUM_THREADS=4
gcc –fopenmp -o HelloWorld HelloWorld.c
./HelloWorld
Hello world ! I'm 0 of 4
Hello world ! I'm 1 of 4
Hello world ! I'm 3 of 4
Hello world ! I'm 2 of 4

For

- Parallélisation d'une boucle for (si chaque itération est indépendante des autres).
- Doit être appelé depuis un environnement parallèle ou avec omp parallel for.
- Chaque thread s'occupe d'un sous-intervalle de la boucle.
- Plusieurs types de division des sous-domaines possibles : static, dynamic, guided ou auto.

Exemple : for

```
// fonction effectuant la moyenne de chaque elements avec ses voisins directes
// output [1...n]
// input [0...n]

int parallelAverage(double* output, const double* input, const int length)
{
    #pragma omp parallel for
        for (int i=1; i<length-1; i++){
            output[i-1] = (input[i-1] + input[i] + input[i+1])/3;
        }
        return 0;
}
```

Single et Sections

- Single permet d'encapsuler un bloc d'instructions qui ne sera exécuté que par un seul thread.
- Sections permet de séparer des tâches différentes et de les exécuter respectivement par 1 seul fil d'exécution.
- Le thread qui exécutera une section ou le single est aléatoire.
- La directive master permet de faire un single en garantissant que le bloc sera exécuté par le thread parent (id=0).

Exemple :Single et Sections

Sections

```
#pragma omp parallel{
    function_0();
    #pragma omp sections{
    function_1();
        #pragma omp section{
            function_2();
        }
        #prgma omp section{
            function_3();
            function_4();
        }
    }
}
```

Single

```
#pragma omp parallel{
    #pragma omp for
    /* bloc parallel 1 */

    #pragma omp single{
    /* bloc sequentiel */
    }

    /*bloc parallel 2 */
}
```

Autres directives

- omp task
- omp taskloop
- omp taskloop simd
- omp simd / declare simd
- omp target / declare target
- omp teams
- etc...

Table des matières

shared, private...

La portée des variables doit être définie pour chaque région parallèle.

- shared : une seule copie de la variable pour tous les threads.
- private : chaque thread possède une copie de la variable. La variable visée est indéfinie avant et après la région.
- firstprivate : variable privée initialisée avec la valeur en entrée.
- lastprivate : la valeur de sortie est donnée par le thread qui effectue la dernière itération de la boucle.
- copyprivate : permet de propager une variable d'une région single aux autres threads

shared, private

```
// Produit Matrice-vecteur
#pragma omp parallel for default(none) private(i,j,sum) shared(m,n,a,b,c)
for (i=0; i<m; i++) {
    sum = 0.0;
    for (j=0; j<n; j++)
        sum += b[i][j]*c[j];
    a[i] = sum;
}
```

reduction

- Pour les boucles for.
- Permet de spécifier une variable pour une réduction avec un opérateur commutatif et associatif.
  $+, *, \&\&, \|,$

```
// faire une somme des elements d'un vecteur
      .
double somme(double* a, int size){
    int i;
    double sum =0;
    #pragma omp parallel for private(i)
         reduction(+:sum)
        for(i=0; i<size; i++){
        sum +=  a[i];
    }
    return sum;
}
```

collapse

- Dans un omp for, seule la première boucle est parallélisée.
- Avec la clause collapse, des boucles imbriquées seront considérées comme une grande boucle.

```
// faire une somme des elements d'une matrice
double somme(double* a, int width, int height){
    int i,j;
    double sum =0;
    #pragma omp parallel for private(i,j) collapse(2) reduction(+:sum)
    for(i=0; i<size; i++){
        for(j=0; j<size; j++)
            sum +=  a[i][j];
    }
    return sum;
}
```

if

- Le bloc parallèle attaché au if ne s'exécute que si la condition est vraie.
- Dans le cas contraire, la région s'exécute en séquentiel.
- Si la taille du problème est petite, il est parfois préférable de rester en séquentiel à cause du surcoût de opemMP.

nowait

- Permet d'enlever la barrière implicite à la fin d'un bloc. Ainsi l'exécution de deux régions parallèles distinctes peuvent se chevaucher.
- La directive omp barrier permet de synchroniser tous les threads de la région.

```
#pragma parallel shared(a,b,c,y,z)
{ #pragma omp for schedule(static) nowait
for (int i=0; i<n; i++) c[i] = (a[i] + b[i]) / 2.0;
#pragma omp for schedule(static) nowait
for (int i=0; i<n; i++) z[i] = c[i]*c[i];
#pragma omp barrier;
#pragma omp for schedule(static) nowait
for (int i=1; i<=n; i++) y[i] = z[i-1] + a[i];
}
```

Table des matières

Problème ? ? ?

```
#include <stdio.h>
#include <omp.h>
int main( int argc, char **argv ){
    int N = 100;
    double a[N];
    double total = 0;
    for(int i=0; i<N; i++) a[i]=i;

    #pragma omp parallel num_threads(4) {
        #pragma omp for
        for(int i=0; i<N; i++){
                total += a[i]
        }
    }
return total;
}
```

réponse actuelle : 3745

réponse attendue : 4950

critical et atomic

Afin de régler un problème de concurrence, Il faut que l'accès aux données ciblées soit protégé soit par des verrous ou en étant locales.

- #pragma omp critical [name] : Une seule région critique du même nom peut s'exécuter en même temps. Cette directive utilise des verrous afin de protéger la région.
- #pragma omp atomic {x operateur= expression}

```
#pragma omp parallel num_threads(4) {
    int i,j;
    #pragma omp for
    for(i=0; i<N; i++){
        #pragma omp critial{
                total += a[i]
        }
    }
}
```

```
#pragma omp parallel num_threads(4) {
    int i,j;
    #pragma omp for
    for(i=0; i<N; i++){
        #pragma omp atomic{
                total += a[i]
        }
    }
}
```

Quelques conseils

- Attention aux problèmes de concurrence. S'assurer que les fonctions appelées sont "thread safe"
- Choisir des morceaux assez gros pour minimiser le surcoût mais assez petits pour équilibrer le travail de chaque thread.
- Attention à l'odre des indices lors du parcours de matrice.

Table des matières

Message Passing Interface

- Ce n'est pas une librairie mais un standard. Il exite plusieurs implémentations différentes (openMPI, MPICH, MVAPICH, IBM MPI, etc...)
- Habituellement supporté en C,C++ et Fortran et sur la plupart des systèmes d'exploitations.
- Programmation sur architechture à mémoire distribuée.
- Un programme MPI est constitué de processus autonomes qui exécutent leur code respectif (MIMD) dans leur espace d'adressage respectif. MPI est un environnment servant uniquement à la communication entre ces processus.

Syntaxe

C/C++ : ierr = MPI_Xxxx(parametre1, ...)
ierr = MPI_Bsend(&buf,count,type,dest,tag,comm)

Fortran : MPI_XXXX(parametre1, ..., ierr)
MPI_BSEND(&buf,count,type,dest,tag,comm,ierr)

Fonctions de base

- MPI_INIT : initialisation de l'environnment MPI. Il ne doit être appelé qu'une seule fois dans le programme.
- MPI_FINALIZE : terminaison des communications.
- MPI_COMM_RANK : retourne le numéro du processus.
- MPI_COMM_SIZE : retourne le nombre de processus dans le communicateur (MPI_COMM_WORLD).
- MPI_ABORD : terminaison de tous les processus MPI.

Hello World MPI

mpiexec ./HelloWorldMPI.c -np 4

```
#include "mpi.h"
#include <stdio.h>
int main( argc, argv ){
  int rank, size;
  MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "Hello world! I'm %d of %d\n",rank, size );
  MPI_Finalize();
  return 0;
}
```

Hello world ! I'm 3 of 4
Hello world ! I'm 2 of 4
Hello world ! I'm 0 of 4
Hello world ! I'm 1 of 4

Table des matières

Contenu d'un message

Le contenu

- buffer : adresse de la variable qui est envoyée ou qui recevra les données.
- count : nombre d'éléments dans le buffer
- datatype : type de la donnée passée dans le message. Il ne peut y en avoir qu'un seul par message. Le système peut effectuer des conversions si nécessaire.

L'enveloppe

- source : L'identificateur de l'envoyeur.
- dest : L'identificateur du receveur.
- tag : Valeur entière identifiant le message (possibilité de wildcard à la réception : MPI_ANY_TAG).
- communicateur : Communicateur de l'envoyeur et du receveur.
- status : objet status indiquant l'état du message (pour la gestion des erreurs).

Exemple de communication

```
#include "mpi.h"
int main( int argc, char *argv[])
{
  char message[20];
  int myrank;
  MPI_Status status;
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &myrank );

  if (myrank == 0){ /* code for process zero */
  strcpy(message,"Hello, there");
  MPI_Send(message, strlen(message), MPI_CHAR, 1, 99, MPI_COMM_WORLD);
  }
  else if (myrank == 1){ /* code for process one */
  MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
  printf("received :%s:\n", message);
  }

  MPI_Finalize();
  return 0;
}
```

Mode de communication

Standard, Buffered, Synchronous, Ready

- MPI_XSEND(buf,count,datatype,dest,tag,comm)
  1. MPI_SEND : peut se comporter comme un BSEND ou SSEND selon le choix de MPI.
  2. MPI_BSEND : termine lorsque le message est complètement copié dans un tampon.
  3. MPI_SSEND : termine lorsqu'un RECV correspondant est appelé.
  4. MPI_RSEND : ne peut être appelé que si un RECV correspondant est en attente.
- MPI_RECV(buf,count,datatype,source,tag,comm,status)
  Termine lorsque le message est totalement copié. Un RECV peut terminer avant le SEND.

interblocage

```
// EXEMPLE 1
MPI_Comm_rank(comm, &rank)
if (rank==0){
    MPI_Bsend(sendbuf, count, MPI_DOUBLE, 1, tag1, comm)
    MPI_Ssend(sendbuf, count, MPI_DOUBLE, 1, tag2, comm)
}
elseif (rank==1){
    MPI_Recv(recvbuf, count, MPI_DOUBLE, 0, tag2, comm, status)
    MPI_Recv(recvbuf, count, MPI_DOUBLE, 0, tag1, comm, status)
}

// EXEMPLE 2
MPI_Comm_rank(comm, &rank)
if (rank==0){
    MPI_Send(sendbuf, count, MPI_REAL, 1, tag, comm)
    MPI_Recv(recvbuf, count, MPI_REAL, 1, tag, comm, status)
}
elseif (rank==1){
    MPI_Send(sendbuf, count, MPI_REAL, 0, tag, comm)
    MPI_Recv(recvbuf, count, MPI_REAL, 0, tag, comm, status)
}
```

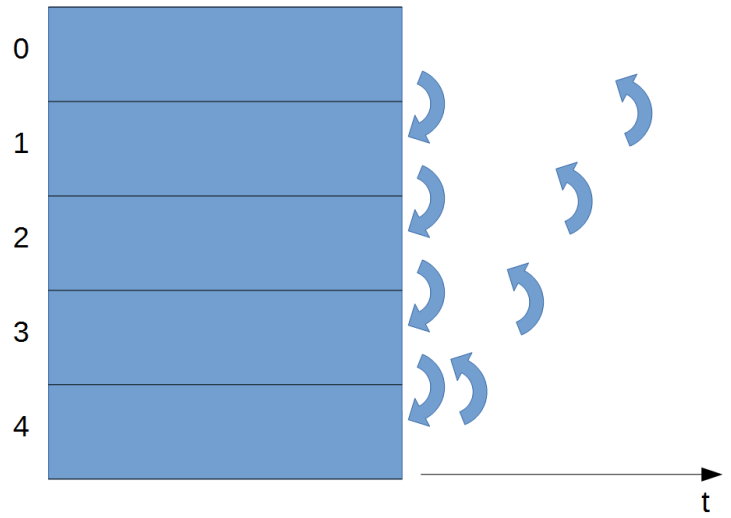
SendReceive

On combine les appels send et receive afin de simplifier les échanges de message.Cela évite les risque d'interblocages.

- MPI_SENDRECV(sendbuf,sendcount,sendtype,dest,sendtag, recvbuf,recvcount,recvtype,source,recvtag,comm,status)
- MPI_SENDRECV_REPLACE(buf,count,datatype,dest,sendtag, source,recvtag,comm,status)

Exemple : SendRecv

```
intrank,size;
MPI_statusstatus;
floatrp[2048],rs[2048],rc[2048];
...
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)
}
```



Exemple : SendRecv

```
intrank,size;
MPI_status status;
float rp[2048],rs[2048],rc[2048];
...
for(t=0;t<max_time;t++){
   if(rank==0){
      MPI_Sendrecv(rc,2048,MPI_FLOAT,rank+1,1,rs,2048,MPI_FLOAT,rank+1,0,MPI_COMM_WORLD);
      itere_chaleur(NULL,rs,rc)
   }
   elseif(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){  // noeud impaire
         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{   // noeud paire
         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)
   }
}
```

Exemple : SendRecv

Communication non-bloquante

Les opérations SEND et RECV sont séparées en deux appels. Le premier permet de commencer la communication sans attendre que l'écriture ou la lecture soit terminée. Le second permet de savoir quand l'opération est terminée. Cela permet de superposer les communications et les calculs.

- MPI_IXSEND(buf,count,datatype,dest,tag,comm,request)
- MPI_IRECV(buf,count,datatype,source,tag,comm,request)

L'objet request contient des informations sur le mode de communication, sur l'enveloppe et le status de la communication

Communication non-bloquante

Pour compléter une communication, on utilise les commandes suivantes :

- MPI_WAIT(request,status)
- MPI_WAITANY(count,array_of_request,index,status)
- MPI_WAITALL(count,array_of_request,array_status)
- MPI_WAITSOME(incount,array_request,outcount, array_index,array_status)

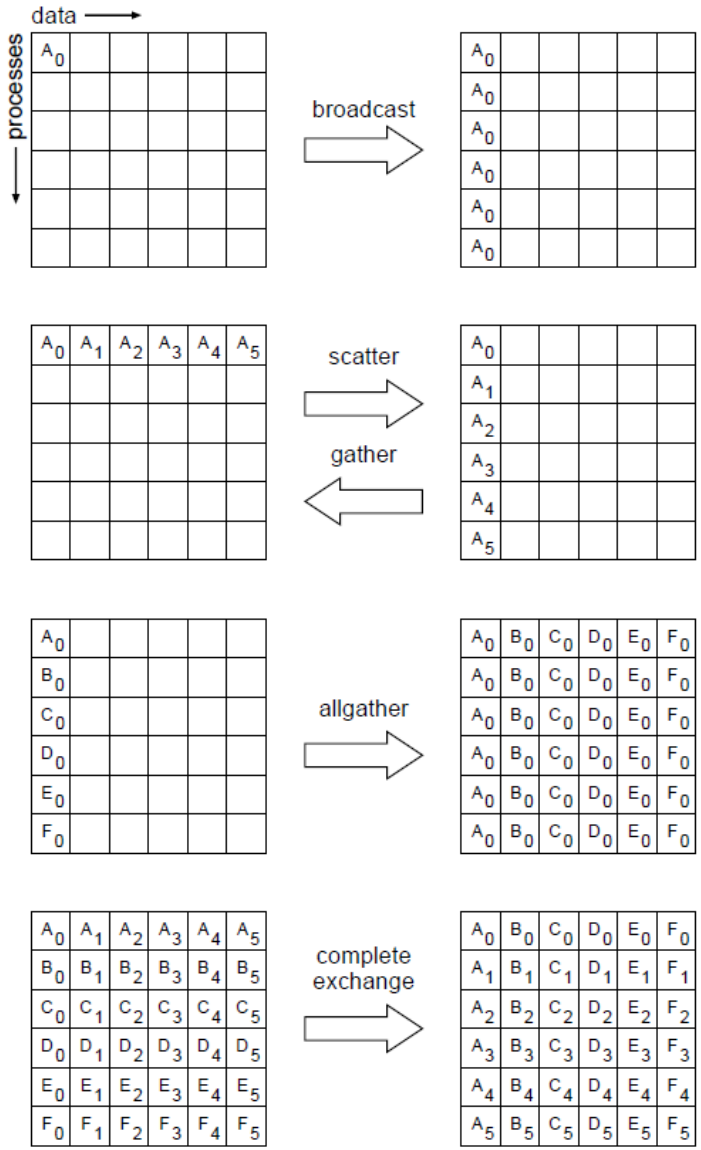
- MPI_TEST(request,flag,status)
- MPI_TESTANY(count,array_of_request,index,flag,status)
- MPI_TESTALL(count,array_of_request,flag,array_status)
- MPI_TESTSOME(incount,array_request,outcount, array_index,array_status)

Exemple : communication non-bloquante

```
//Communication circulaire

#include <stdio.h>
#include "mpi.h"
int main (int argc, char* argv[]){
        int numtasks, rank, next, prev, buf[2], tag1=1, tag2=2;
  MPI_Request reqs[4];
  MPI_Status stats[4];
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  prev = rank-1;    next = rank+1;
  if(rank == 0) prev = numtasks - 1;
  if(rank == (numtasks - 1))  next = 0;
  MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[0]);
  MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[1]);

  MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[2]);
  MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[3]);

  MPI_Waitall(4, reqs, stats);
  printf("Task %d communicated with tasks %d & %d \n",rank,prev,next);
MPI_Finalize();
}
```

Table des matières

communications collectives



Communications collectives

- MPI_BCAST(buf,count,datatype,root,comm)
- MPI_GATHER(sendbuf,sendcount,sendtype, recvbuf,recvcount,recvtype,root,comm)
- MPI_SCATTER(sendbuf,sendcount,sendtype, recvbuf,recvcount,recvtype,root,comm)
- MPI_ALLGATHER(sendbuf,sendcount,sendtype, recvbuf,recvcount,recvtype,comm)
- MPI_ALLTOALL(sendbuf,sendcount,sendtype, recvbuf,recvcount,recvtype,comm)

Réduction

Possibilité de faire une réduction avec des opérations prédéfinies ou définies par l'usager(MPI_OP_CREATE).

- MPI_REDUCE(sendbuf,recvbuf,count,datatype,op,root,comm)
- MPI_ALLREDUCE(sendbuf,recvbuf,count,datatype,op,comm)

Les opérations prédéfinies sont :
MAX,MIN,SUM,PROD,LAND,BAND,LOR,BOR,LXOR,BXOR,MAXLOC et MINLOC

Exemple : calcul de pi

```
//genere coordonnee x et y
void random_coord(double* x,double* y)

int main(int argc, char *argv[]){
    int rank, size, ierr;
    MPI_Status status;
    int root = 0;

    double N_try = 100000
    double hit_local =0;
    double hit_total = 0;

    ierr = MPI_Init(&argc,&argv);
    ierr = MPI_Comm_size(MPI_COMM_WORLD,&size);
    ierr = MPI_Comm_rank(MPI_COMM_WORLD,&rank);

    for(int i=0; i< N_try; i++){
        double x,y,norm;
        random_coord(&x,&y);

        norm = x*x +y*y
        if (norm < 1){hit_local++}
     }
     ierr = MPI_Reduce(&hit_local,&hit_total,1,MPI_DOUBLE,
                MPI_SUM,root,MPI_COMM_WORLD)
     if(rank == root){
        double pi = 4*hit_total/N_try;
     }
     ierr = MPI_Finalize();
     return 0;
```

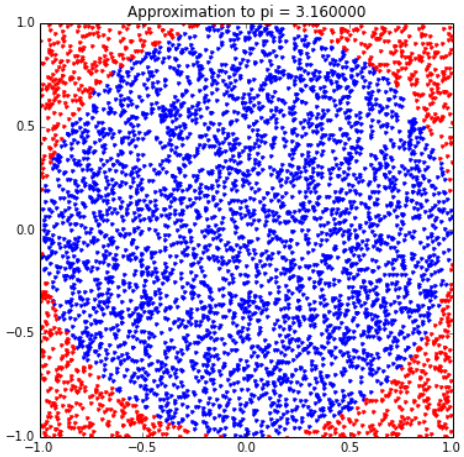


FIGURE – Calcul de pi

Autre type d'opération globale

- MPI_REDUCE_SCATTER(sendbuf,recvbuf,recvcount, datatype,op,comm) : redistribue le résultat de la réduction
- MPI_SCAN(sendbuf,recvbuf,count,datatype,op,comm) : réduction avec préfixe
- MPI_EXSCAN(sendbuf,recvbuf,count,datatype,op,comm)
- MPI_BARRIER(comm)

Les points importants

- Attention aux situations d'interblocage.
- Les communications non bloquantes permettent d'effectuer des calculs en même temps.
- S'assurer que les communications sont parallèles (si possible)

Table des matières

À retenir !

- La loi d'Amdhal.
- La différence entre un thread et un processus.
- OpenMP crée des groupes de threads afin d'insérer des blocs parallèles dans un code.
- MPI est un standard de communication inter-processus.
- Faites attention aux problèmes de concurrence et aux situations d'interblocages.
- Une panoplie de tutoriels sur openMPI et MPI sont disponibles sur internet.
- On retrouve les documentations officielles aux adresses suivantes :
  - www.openmp.org
  - mpi-forum.org

questions ?