Clustering sous Fedora

Table des matières

Notion de cluster

Un « cluster » (en français « **grappe** ») est une architecture composée de plusieurs ordinateurs formant des noeuds, où chacun des noeuds est capable de fonctionner indépendamment des autres.

Il existe deux principaux usages des clusters :

- Les **clusters de haute disponibilité** permettent de répartir une charge de travail parmi un grand nombre de serveurs et de garantir l'accomplissement de la tâche même en cas de défaillance d'un des noeuds ;
- Les **clusters de calcul** permettent de répartir une charge de travail parmi un grand nombre de serveurs afin d'utiliser la performance cumulée de chacun des noeuds.

Installation du cluster

Premièrement, on devra installer au moins 2 ordinateurs avec la distribution de votre choix. Dans l'exemple qui suit, nous avons installé la distribution Fedora 20 sur nos appareils.

Pour avoir un cluster, nous devons posséder minimalement 2 ordinateurs. Un ordinateur sera le noeud principal et les autres deviendront des noeuds secondaires.

1. Installation du noeud principal
    - Procéder à installer Fedora 20 ou autre version sur l'ordinateur comme normalement.
    - À l'écran de création des usagers, créez un usager nommé « mpiuser » qui aura le mot de passe « mpiuser ».
    - Lorsque l'installation est terminé :
    - Donnez une adresse statique à la carte réseau du réseau interne.
    - Allez dans le répertoire /etc/sysconfig/network-scripts
    - Éditer le fichier ifcfg-em1 ou ifcfg-exxx (la première lettre commence par « e » après le tiret).

ATTENTION
L'adresse IP ajoutée ici est spécialement utilisée pour l'environnement du collège.

- Ajoutez : IPADDR=172.17.8.200
- NETMASK=255.255.255.0
- GATEWAY=172.17.8.1
- DNS1=172.17.200.11
- DNS2=172.17.200.12
- Changez BOOTPROTO=dhcp pour BOOTPROTO=static
- Sauvegardez le fichier
- redémarrez l'ordinateur

2. Installation des noeuds secondaires
   - Procédez à installer Fedora 20 de la même façon que pour le noeud principal.
   - Allez dans le répertoire /etc/sysconfig/network-scripts
   - Éditer le fichier ifcfg-em1 ou ifcfg-exxx (la première lettre commence par « e » après le tiret).
   - IPADDR=172.17.8.201
   - NETMASK=255.255.255.0
   - GATEWAY=172.17.8.1
   - DNS1=172.17.200.11
   - DNS2=172.17.200.12

Redémarrez l'ordinateur.

Procédez de la même façon pour les noeuds secondaires subséquents.

Configuration de SSH sans mot de passe

Branchez-vous avec le compte “mpiuser” sur l'ordinateur qui constitue le noeud principal.

Exécutez les commandes suivantes:

ssh-keygen -t rsa

À la question: Enter file in which to save the key (/home/mpiuser/.ssh/id_rsa):

Tapez la touche “enter” pour confirmer.

À la question : Enter passphrase (empty for no passphrase) :

tapez la touche « enter » pour ne rien entrer.

Confirmez une deuxième fois

Copiez les clés générées sur les noeuds secondaires.

1. ssh-copy-id -i .ssh/id_rsa.pub mpiuser@adresse_IP_du_noeud_1
2. Exemple avec le noeud 1 possédant l'adresse IP 172.17.8.201 :

   ssh-copy-id -i .ssh/id_rsa.pub mpiuser172.17.8.201

   Si vous obtenez une question qui demande de confirmer la connection comme ci-dessous, répondez yes.

   Are you sure you want to continue connecting (yes/no) ?

   Entrez le mot de passe de l'usager « mpiuser » lorsque demandé.

   Le système vous répondra qu'il a copié la clé correctement. Vous devriez voir :

   Number of key(s) added : 1
3. Refaites les 2 étapes précédentes pour chaque noeuds secondaires de votre cluster (en remplaçant les adresse ip évidemment)

Faites un test pour savoir si vous pouvez effectivement vous connecter sur le noeud secondaire :

ssh mpiuser@192.168.1.101

Si vous obtenez un prompt sans aucune demande de mot de passe, c'est que la procédure a probablement fonctionné. Entrez la commande suivante pour vous en assurer :

ifconfig

et constatez que l'adresse IP de la carte réseau est bel et bien celle de l'autre ordinateur (noeud secondaire).

Tapez exit pour quitter la connexion.

Installation de MPICH

Sur tous les ordinateurs de votre cluster (noeud principal et secondaires) vous devez installer la librairie de programmation MPICH.

À partir du noeud principal, entrez la commande suivante:

sudo yum install mpich2 mpich2-devel

Modifiez le chemin du système pour prendre en compte le chemin où se trouve les binaires et les librairies de mpi.

1. Assurez-vous d'être dans votre répertoire maison (cd)
2. Éditer le fichier « .bash_profile » et modifiez la ligne qui commence par PATH=. Ajoutez à la fin de cette ligne les éléments suivants :

/usr/lib64/mpich/bin:/usr/lib64/mpich/lib

ATTENTION

Remplacer le répertoire lib64 par lib si vous avez un système 32 bits.

Vous aurez donc une ligne qui ressemblera à :

PATH=$PATH:$HOME/.local/bin:$HOME/bin:/usr/lib64/mpich/bin:/usr/lib64/mpich/lib

Désactivation de selinux et du mur coupe-feu

Pour le bon fonctionnement de la démonstration et afin de faciliter les échanges entre les ordinateurs du réseau de notre cluster, nous allons désactiver SELinux et le mur coupe-feu (firewall) de toute les machines du cluster.

Pour chaque machine, ceci comprend le noeud principal et les noeuds secondaires, réalisez les étapes suivantes :

Désactiver SELINUX

1. Éditer (avec sudo) le fichier /etc/selinux/config
2. modifier la ligne selinux=enforcing et changez-là pour : selinux=disabled
3. Sauvegarder le fichier.

Désactiver le mur coupe-feu

1. sudo systemctl disable firewalld
2. sudo systemctl stop firewalld

Pour activer tous ces changements, vous devez redémarrer l'ordinateur.

Répéter les étapes précédentes pour tous les noeuds secondaires.

Test du cluster

Nous allons tester le cluster en envoyant une commande à tous les noeuds pour s'assurer qu'ils répondent adéquatement.

Nous allons ensuite tester ce même cluster en exécutant un fichier en parallèle.

Test par l'envoie d'une commande

Branchez-vous sur le noeud principal avec ssh.

ssh mpiuser@172.17.8.200

Créez un fichier dans ce répertoire et entrez les adresses IP de chaque noeud du cluster. Une ligne par adresse IP. Voici un exemple avec 2 machines :

172.17.8.200

172.17.8.201

Sauvegardez le fichier sous le nom « clusterip ».

Réalisez ensuite les commandes suivantes :

sudo nano /etc/hosts

ajouter les lignes :

172.17.8.200 Master

172.17.8.201 Node1

Sauvegardez le fichier.

Entrez la commande :

mpirun -n 2 hostname

Vous devriez recevoir la sortie suivante :

master

master

Essayez encore avec la commande suivante :

mpirun -n 8 hostname

master

master

et ainsi de suite 6 autres fois.

Retaper la commande mais cette fois-ci, en utilisant le fichier clusterip :

mpirun -f clusterip -n 2 hostname

master

node1

mpirun -f clusterip -n 8 hostname

master

3 autre fois avec master

node1

et 3 autre fois avec node1

Test de l'exécution d'un programme parallèle

- Branchez-vous avec le compte « mpiuser » sur le noeud principal

Avec votre éditeur de texte préféré, entrez les lignes suivantes :

```
#include <mpi.h>

int main(int argc, char** argv)
{
      int Nbre_Proc;  // Pour le nombre total de processeur disponible
                      // dans le cluster
      int Rang_Proc;    // Le numéro du processeur(de la machine)
                        // dans le cluster
      char Nom_Machine[MPI_MAX_PROCESSOR_NAME);
      // Le nom de la machine qui participe au cluster

      int Longueur_Nom_Machine;     // Longueur du nom de la machine
                                    // faisant partie du cluster


  // Initialise l'environnement de MPI
  MPI_Init(NULL, NULL);

  // Cherche le nombre de processeur à la disposition du cluster

  MPI_Comm_size(MPI_COMM_WORLD, &Nbre_Proc);

  // Va chercher le numéro de la machine

  MPI_Comm_rank(MPI_COMM_WORLD, &Rang_Proc);

  // Récupère le nom de la machine (opération non essentielle)

  MPI_Get_processor_name(Nom_Machine, &Longueur_Nom_Machine);

  cout << "Bonjour Cluster de la part du processeur " << Nom_Machine <<
" # " << Rang_Proc << " sur " << Nbre_Proc << endl;

  // Fermer adéquatement l'environnement MPI.
  MPI_Finalize();
}
```

Compiler le programme c++ avec la commande mpic++ :

```
mpic++  -o  nom_exécutable  nom_source.cpp
```

par exemple, pour le programme précédent qui se nomme ex1.cpp :

```
mpic++  -o  ex1   ex1.cpp
```

Exécutez avec mpirun :

```
mpirun  -n 2  ./ex1
```

Copiez le fichier exécutable sur le noeud secondaire :

```
scp  ex1  mpiuser@192.168.1.101:~/
```

Exécutez en utilisant le fichier « clusterip »:

```
mpirun  -f clusterip -n 2  ./ex1
```

En conclusion

Que dois-je faire lorsque je veux développer un programme parallèle ?

1. Développer le programme sur votre poste.
2. Compiler et débogguer avec mpic++
   mpic++ -0 executable nom_source.cpp

3. Envoyez le programme sur le poste du noeud principal.
   scp nom_executable 172.17.8.200:~

4. Branchez-vous sur le noeud principal.
   ssh mpiuser@172.17.8.200

5. Envoyez enuite, à partir du noeud principal, l'exécutable aux noeuds secondaires du cluster.

   Pour chaque noeud: (par exemple, s'il y a 3 noeuds)
   scp executable 172.17.8.201:~/
   scp executable 172.17.8.202:~/
   scp executable 172.17.8.203:~/

6. à partir du noeud principal, exécutez le programme par le cluster

   mpirun -n #processeur -f listeOrdiCluster ./nom_executable où

   #processeur est le nombre de processeur voulu dans le cluster

   ListeOrdiCluster est le fichier contenant les adresses IP des machines faisant partie du cluster.

Tout programme faisant usage de MPI doit inclure en c++ un fichier d'entête. Ce fichier se nomme “mpi.h”.

Le programme MPI à sa plus simple expression doit contenir les éléments suivants:

```
#include <mpi.h>

#include <mpicxx.h>  si ce sont les classes c++ qui sont utilisées.

#include <mpi.h>

int main()
{
   int Nbre_Processeur;
   int Rang_Processeur;

   MPI_Init(NULL, NULL);

   MPI_Comm_size(MPI_COMM_WORLD, &Nbre_Processeur);

  MPI_Comm_rank(MPI_COMM_WORLD, &Rang_Processeur);

  MPI_Finalize();

}
```

On compile avec la commande suivante: mpic++ -o exemple1 exemple1.cpp

Exemple 2: Faisons parler les ordinateurs du clusters

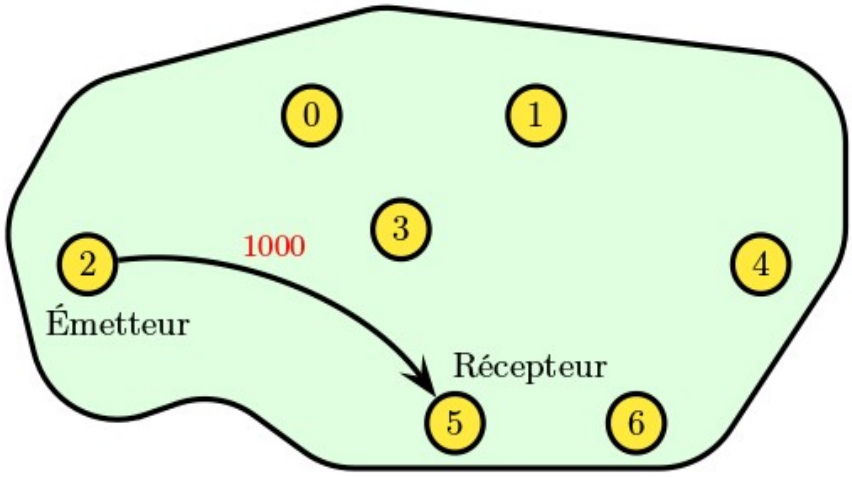
```
#include <mpi.h>

int main()
{
   int Nbre_Processeur;
   int Rang_Processeur;

   MPI_Init(NULL, NULL);

   MPI_Comm_size(MPI_COMM_WORLD, &Nbre_Processeur);

  MPI_Comm_rank(MPI_COMM_WORLD, &Rang_Processeur);
  cout << “Je suis le processus # “ << Rang_Processeur << “ sur “ << Nbre_Processeur
<< “ disponible.” << endl;

  MPI_Finalize();

}
```

Les communicateurs

Un communicateur désigne un ensemble de processus pouvant communiquer ensemble, et deux processus ne pourront communiquer que s'ils sont dans un même communicateur. Un communicateur initial englobe tous les processus (MPI_COMM_WORLD), qu'il est possible de subdiviser en communicateurs plus petits correspondants à des entités logiques. Il existe deux types de communicateurs : les intracommunicateurs et les intercommunicateurs. Les intracommunicateurs sont les communicateurs standards, alors que les intercommunicateurs servent à créer un pont entre deux intracommunicateurs. MPI-2 améliore grandement l'usage des intercommunicateurs en leur permettant de réaliser des communications collectives.

Communication Point à Point

Une communication dite point à point a lieu entre deux processus, l'un appelé processus émetteur et l'autre processus récepteur (ou destinataire).



L'émetteur et le récepteur sont identifiés par leur rang dans le communicateur. Ce que l'on appelle l'enveloppe d'un message est constituée :

- du rang du processus émetteur ;
- du rang du processus récepteur ;
- de l'étiquette (tag) du message ;
- du communicateur qui définit le groupe de processus et le contexte de communication.

Les données échangées sont typées (entiers, réels, etc ou types dérivés personnels). Il existe dans chaque cas plusieurs modes de transfert, faisant appel à des protocoles différents.

Fonction MPI_send()

Cette fonction permet l'envoie d'un message à un destinataire du cluster à partir d'une autre machine de ce cluster.

Syntaxe:
int MPI_Send(void *buf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm comm)

Remarque :
Cette opération est bloquante : l'exécution reste bloquée jusqu'à ce que le contenu de message puisse être réécrit sans risque d'écraser la valeur qui devait être envoyée

Paramètre:

void *buf: Message à envoyer
int count: Longueur du message
MPI_Datatype datatype: Type de donné (MPI_INT, MPI_DOUBLE...)
int dest: Destinataire du message
int tag: Un numéro commun identifiant la connexion.
MPI_Comm comm: Le communicateur utilisé (habituellement MPI_COMM_WORLD)

Fonction MPI_recv()

Cette fonction permet la réception d'un message qui parvient d'une autre machine du cluster.

Syntaxe:

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

Remarques:

statut reçoit des informations sur la communication : rang_source, etiquette,code, ...

L'appel MPI_RECV ne pourra fonctionner avec une opération MPI_SEND que si ces deux appels ont la même enveloppe (rang_source, rang_dest, etiquette, comm).

Cette opération est bloquante : l'exécution reste bloquée jusqu'à ce que le contenu de message corresponde au message reçu.

Paramètres:

void *buf: Message à recevoir
int count: Longueur du message
MPI_Datatype datatype: Type de donné (MPI_INT, MPI_DOUBLE...)
int source: Source du message (De qui vient le message)
int tag: Un numéro commun identifiant la connexion. (Le même que pour MPI_send)
MPI_Comm comm: Le communicateur utilisé (habituellement MPI_COMM_WORLD)
MPI_Status *status: L'état de la communication.

Exemple 3

Envoie d'un message de l'ordinateur #0 à l'ordinateur #1.

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

using namespace std;

int main()
{
   int Nbre_Processeur;
   int Rang_Processeur;
   int Valeur_Envoyee = 255, Valeur_Recue;
   int Tag = 1000;

   MPI_Status  Statut;

   MPI_Init(NULL, NULL);

   MPI_Comm_size(MPI_COMM_WORLD, &Nbre_Processeur);

  MPI_Comm_rank(MPI_COMM_WORLD, &Rang_Processeur);

  if (Rang_Processeur == 0)
  {
     MPI_send(&Valeur_Envoyee, 1, MPI_INT, 1, Tag, MPI_COMM_WORLD);
  }
  else
  {
    if (Rang_Processeur == 1)
    {
      MPI_Recv(&Valeur_Recue, 1, MPI_INT, 0, Tag, MPI_COMM_WORLD, &Statut);
      cout << “j'ai reçue la valeur “ << Valeur_Recue << “ de l'ordi #0” << endl;
    }
  }

  MPI_Finalize();

}
```

Compilation: mpic++ -o exemple3 exemple3.cpp

Exécution: mpirun -n 2 -f listordi ./exemple3

Type de donnée de base en MPI

Type MPI	Type C/C++
MPI_CHAR	signed char
MPI_SHORT	signed short
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double

Exemple 4: Envoi d'un tableau de 10 éléments à un autre ordinateur.

```
#include "mpi.h"
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
   int Rang_Processeur, Nbre_Processeur, i;
   int Tableau[10];
   MPI_Status status;
  int Tag = 1000;

   MPI_Init(NULL, NULL);
   MPI_Comm_size(MPI_COMM_WORLD, &Nbre_Processeur);
   MPI_Comm_rank(MPI_COMM_WORLD, &Rang_Processeur);
   if (Nbre_Processeur  <  2)
   {
       cout << “S.v.p.  Veuillez utiliser plus d'une machine.” << endl;
      MPI_Finalize();
      return 0;
   }
   if (Rang_Processeur == 0)
   {
      for (i=0; i<10; i++)
         Tableau[i] = i;
          MPI_Send(Tableau, 10, MPI_INT, Rang_Processeur+1, Tag, MPI_COMM_WORLD);
   }
```

```
   if (Rang_Processeur == 1)
   {
       for (i=0; i<10; i++)
          Tableau[i] = -1;
        MPI_Recv(Tableau, 10, MPI_INT, 0, Tag, MPI_COMM_WORLD, &status);
       for (i=0; i<10; i++)
       {
            cout << "tableau[" << i << "] = " << Tableau[i] << endl;
       }

   }
   MPI_Finalize();

    return 0;
}
```

Communication Collective

Les communications collectives permettent de faire en une seule opération une série de communications point à point. Une communication collective concerne toujours tous les processus du communicateur indiqué. Pour chacun des processus, l'appel se termine lorsque la participation de celui-ci à l'opération collective est achevée, au sens des communications point-à-point (donc quand la zone mémoire concernée peut être modifiée).

La gestion des étiquettes dans ces communications est transparente et à la charge du système. Elles ne sont donc jamais définies explicitement lors de l'appel à ces sous-programmes. Cela a entre autres pour avantage que les communications collectives n'interfèrent jamais avec les communications point à point.

Fonctions assurants la communication collectives

Il y a trois types de sous-programmes :

- celui qui assure les synchronisations globales : MPI_BARRIER()
- ceux qui ne font que transférer des données :
    - diffusion globale de données : MPI_BCAST()
    - diffusion sélective de données : MPI_SCATTER()

- collecte de données réparties : MPI_GATHER()
- collecte par tous les processus de données réparties : MPI_ALLGATHER()
- diffusion sélective, par tous les processus, de données réparties MPI_ALLTOALL()

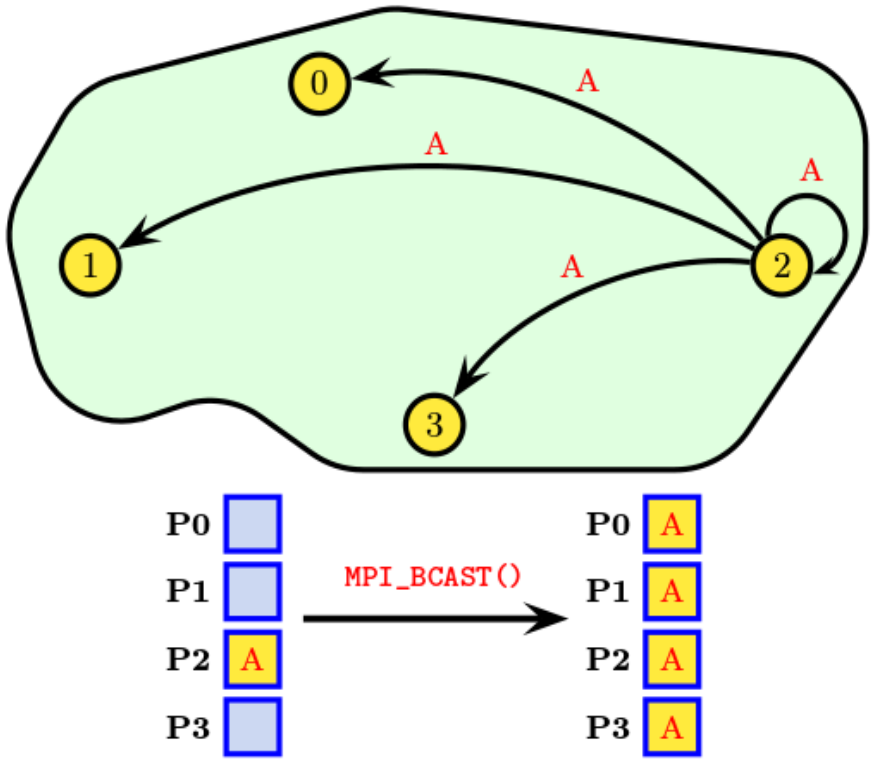
ceux qui, en plus de la gestion des communications, effectuent des opérations sur les données transférées :

- opérations de réduction (somme, produit, maximum, minimum, etc.), qu'elles soient d'un type prédéfini ou d'un type personnel : MPI_REDUCE()

- opérations de réduction avec diffusion du résultat (équivalent à un MPI_REDUCE() suivi d'un MPI_BCAST()) : MPI_ALLREDUCE()

Usage du Broadcasting

Graphiquement, l'opération de "broadcasting" se représente de la façon suivante:



Fonction MPI_Bcast()

Envoi un message à partir d'un processus à tous les autres processus du cluster.

Syntaxe:
MPI_Bcast(&Donneer,Compte,TypeDonnee,Racine, Comm)

où:

Donnee: La donnée à envoyer aux autres processus.
Compte: Le nombre d'octet à envoyer.
TypeDonnee: Le type des données qui sont envoyées.
Racine: Le processus émmeteur.
Comm: Le groupe de communication (MPI_COMM_WORLD)

Exemple 5: Envoi d'un entier à tous les autres processus du cluster et affichage de la donnée reçue.

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

using namespace std;

int main(int argc, char *argv[])
{

   int Rang, NbreProc;
   int Valeur;

   MPI_Init(&argc, &argv);

   MPI_Comm_rank(MPI_COMM_WORLD, &Rang);
   MPI_Comm_size(MPI_COMM_WORLD, &NbreProc);

   if (NbreProc < 2)
   {
      cout << "Il faut au moins 2 machines pour exécuter ce programme" << endl;
      return 0;
   }
   if(Rang == 0)
   {
     Valeur = 100;
   }

    MPI_Bcast(&Valeur, 1, MPI_INT, 0, MPI_COMM_WORLD);
    cout << "J'ai recu la valeur: " << Valeur << " du processus #0" << endl;

   MPI_Finalize();

   return 0;
}
```

Fonction MPI_Reduce()

Une réduction est une opération appliquée à un ensemble d'éléments pour en obtenir une seule valeur. Des exemples typiques sont la somme des éléments d'un vecteur ou la recherche d'un maximum ou d'un minimum dans un vecteur.

Syntaxe :

int **MPI_Reduce**(void * sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm communicator);

où:

sendbuf:	La ou les données envoyées.
recvbuf:	La ou les données reçues.
Count:	Le nombre d'octet à envoyer.
Datatype:	Le type de la donnée.
Op:	L'opérateur de réduction. (Voir tableau ci-dessous)
root:	Le numéro de rang de la machine qui reçoie les résultats (habituellement la machine #0)
communicator:	Le groupe de communication. (MPI_COMM_WORLD)

Dans la forme MPI_Reduce() seul le processeur root reçoit le résultat

Il existe la forme MPI_AllReduce(), ou tous les processus reçoivent le résultat

Type d'opération que l'on peut exécuter avec une réduction

Nom	Opération
MPI_SUM	Somme des éléments
MPI_PROD	Produit des éléments
MPI_MAX	Recherche du maximum
MPI_MIN	Recherche du minimum
MPI_MAXLOC	Recherche de l'indice du maximum
MPI_MINLOC	Recherche de l'indice du minimum
MPI_LAND	ET logique
MPI_LOR	OU logique
MPI_LXOR	OU exclusif logique

Exemple 6: Somme des nombre de 1 à 1000.

Version sérielle:

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

using namespace std;

int main(int argc, char *argv[])
{
  int Somme = 0;
  int ValeurFinale = 1000;

  for (int i = 0; i < ValeurFinale; i++)
  {
    Somme += i;
  }
  cout << "La somme de 1 à 1000 :  " << Somme  << endl;
```

Version parallèle:

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

using namespace std;

int main(int argc, char *argv[])
{

   int Rang, NbreProc;
   int ValeurFinale = 10;
   int IntervalleDebut, IntervalleFin;
   int SommeIntermediaire = 0;
   int SommeTotale = 0;

   MPI_Init(&argc, &argv);

   MPI_Comm_rank(MPI_COMM_WORLD, &Rang);
   MPI_Comm_size(MPI_COMM_WORLD, &NbreProc);

   if (NbreProc < 2)
   {
      cout << "Il faut au moins 2 machines pour exécuter ce programme" << endl;
      return 0;
   }
   IntervalleDebut = Rang * (ValeurFinale / NbreProc) + 1;

   if(Rang == (NbreProc - 1) )
   {
     IntervalleFin = ValeurFinale;
   }
   else
   {
      IntervalleFin = IntervalleDebut + (ValeurFinale / NbreProc) - 1;
   }

   for (int i = IntervalleDebut; i <= IntervalleFin; i++)
   {
      SommeIntermediaire += i;
   }

    MPI_Reduce(&SommeIntermediaire, &SommeTotale, 1, MPI_INT, MPI_SUM, 0,
MPI_COMM_WORLD);

    cout << "Rang# " << Rang << " Somme intermediaire: " << SommeIntermediaire << " Somme
totale: " << SommeTotale << endl;

   MPI_Finalize();

   return 0;
}
```