Introduction à la Programmation Parallèle: MPI

Frédéric Gava et Gaétan Hains

L.A.C.L
Laboratoire d'**A**lgorithmique, **C**omplexité et **L**ogique

Cours du M2 SSI option PSSR

Plan

Plan

Plan

Plan

Déroulement du cours

Mémoire répartie

- Peu de systèmes informatiques offrent une véritable mémoire partagée. Même si c'est le cas (par ex. SGI Origin 2000 ou les Core Duo etc.) plusieurs facteurs technologiques et physiques font qu'un système multi-processeur a un temps d'accès à la mémoire qui est variable (*Non-uniform memory access* ou NUMA).
- Il vaut alors mieux construire, modéliser ou programmer le système comme un ensemble de blocs controle-processeur-mémoire, ce qui revient à répartir la mémoire sur un réseau d'ordinateurs séquentiels.

Mémoire répartie

- Peu de systèmes informatiques offrent une véritable mémoire partagée. Même si c'est le cas (par ex. SGI Origin 2000 ou les Core Duo etc.) plusieurs facteurs technologiques et physiques font qu'un système multi-processeur a un temps d'accès à la mémoire qui est variable (*Non-uniform memory access* ou NUMA).
- Il vaut alors mieux construire, modéliser ou programmer le système comme un ensemble de blocs controle-processeur-mémoire, ce qui revient à répartir la mémoire sur un réseau d'ordinateurs séquentiels.

Envoi de messages

Une méthode

Le modèle de programmation est alors appelé envoi de messages (*message passing* en anglais). Il consiste à faire exécuter à chaque processeur (c'est-à-dire chaque bloc controle-processeur-mémoire) un programme séquentiel faisant appel à des opérations **d'envoi** et de **réception** de messages. On suppose (pour simplifier) un unique processus par processeur et on utilise donc les termes *processeur* ou *processus* sans distinction [a] L'exécution des processus ne se synchronise qu'à travers les opérations d'envoi et réception de messages. Les données sont partagées via des messages qui les transmettent.

---

a. En réalité, on peut faire exécuter plus d'un processus par processeur physique ; ce qui ne change pas la sémantique des programmes mais leur performance.

Envoi de messages

Avantage

Portabilité et *extensibilité* : programmes valables sur 1, 2 ou des milliers de processeurs.

Inconvénient

Le programmeur doit raisonner en fonction d'une mémoire à deux niveaux : global vs local. Si par exemple on programme un algorithme qui agit sur une matrice de grande taille, il faudra la découper en P blocs et coordonner l'effet global des opérations sur la matrice.

Envoi de messages

Avantage

Portabilité et *extensibilité* : programmes valables sur 1, 2 ou des milliers de processeurs.

Inconvénient

Le programmeur doit raisonner en fonction d'une mémoire à deux niveaux : global vs local. Si par exemple on programme un algorithme qui agit sur une matrice de grande taille, il faudra la découper en P blocs et coordonner l'effet global des opérations sur la matrice.

Déroulement du cours

MPI : Message Passing Interface

Kesako ?

Un standard de bibliothèque d'envoi de messages. Défini par un "forum" de chercheurs et d'industriels du calcul haute performance. Plusieurs implantations gratuites, toutes portables.

- Plus d'info sur :
  - `http://www.mpi-forum.org/`
  - `http://en.wikipedia.org/wiki/Message_Passing_Interface`
- Deux implantations courament utilisées :
  - `http://www.mcs.anl.gov/mpi/`
  - `http://www.lam-mpi.org/`

MPI : Message Passing Interface

Kesako ?

Un standard de bibliothèque d'envoi de messages. Défini par un "forum" de chercheurs et d'industriels du calcul haute performance. Plusieurs implantations gratuites, toutes portables.

- Plus d'info sur :
  - `http://www.mpi-forum.org/`
  - `http://en.wikipedia.org/wiki/Message_Passing_Interface`
- Deux implantations courament utilisées :
  - `http://www.mcs.anl.gov/mpi/`
  - `http://www.lam-mpi.org/`

MPI : Message Passing Interface

Kesako ?

Un standard de bibliothèque d'envoi de messages. Défini par un "forum" de chercheurs et d'industriels du calcul haute performance. Plusieurs implantations gratuites, toutes portables.

- Plus d'info sur :
  - `http://www.mpi-forum.org/`
  - `http://en.wikipedia.org/wiki/Message_Passing_Interface`
- Deux implantations courament utilisées :
  - `http://www.mcs.anl.gov/mpi/`
  - `http://www.lam-mpi.org/`

MPI : Message Passing Interface

Pourquoi MPI ?

- Faciliter la programmation par envoi de message
- Portabilité des programmes parallèles
- Utilisable avec une architecture **hétérogène**
- Cible stable pour des outils/langages de plus haut niveau

Que contient MPI ?

MPI : Message Passing Interface

Pourquoi MPI ?

- Faciliter la programmation par envoi de message
- Portabilité des programmes parallèles
- Utilisable avec une architecture **hétérogène**
- Cible stable pour des outils/langages de plus haut niveau

Que contient MPI ?

1. Contrôle de l'environnement d'exécution parallèle
2. Opérations de communications point-à-point
3. Opérations de communications collectives
4. Groupes de processus
5. Interface de communications personnalisées
6. Topologies de processus

MPI : Message Passing Interface

Pourquoi MPI ?

- Faciliter la programmation par envoi de message
- Portabilité des programmes parallèles
- Utilisable avec une architecture **hétérogène**
- Cible stable pour des outils/langages de plus haut niveau

Que contient MPI ?

- Constantes, types et fonctions pour C et F77
- Opérations de communications point-à-point
- Opérations de communications collectives
- Groupes de processus (communicateurs)
- Topologies de processus
- Gestion de l'environnement et profilage

MPI : Message Passing Interface

Pourquoi MPI ?

- Faciliter la programmation par envoi de message
- Portabilité des programmes parallèles
- Utilisable avec une architecture **hétérogène**
- Cible stable pour des outils/langages de plus haut niveau

Que contient MPI ?

- Constantes, types et fonctions pour C et F77
- Opérations de communications point-à-point
- Opérations de communications collectives
- Groupes de processus
- Topologies de processus
- Environnements d'exécution

MPI : Message Passing Interface

Pourquoi MPI ?

- Faciliter la programmation par envoi de message
- Portabilité des programmes parallèles
- Utilisable avec une architecture **hétérogène**
- Cible stable pour des outils/langages de plus haut niveau

Que contient MPI ?

- Constantes, types et fonctions pour C et F77
- Opérations de communications point-à-point
- Opérations de communications collectives
- Groupes de processus
- Interface de suivi des performances
- Fonctions d'E/S parallèles (version 2)

MPI : Message Passing Interface

Pourquoi MPI ?

- Faciliter la programmation par envoi de message
- Portabilité des programmes parallèles
- Utilisable avec une architecture **hétérogène**
- Cible stable pour des outils/langages de plus haut niveau

Que contient MPI ?

- Constantes, types et fonctions pour C et F77
- Opérations de communications point-à-point
- Opérations de communications collectives
- Groupes de processus
- Interface de suivi des performances
- Fonctions d'E/S parallèles (version 2)

MPI : Message Passing Interface

Pourquoi MPI ?

- Faciliter la programmation par envoi de message
- Portabilité des programmes parallèles
- Utilisable avec une architecture **hétérogène**
- Cible stable pour des outils/langages de plus haut niveau

Que contient MPI ?

- Constantes, types et fonctions pour C et F77
- Opérations de communications point-à-point
- Opérations de communications collectives
- Groupes de processus
- Interface de suivi des performances
- Fonctions d'E/S parallèles (version 2)

MPI : Message Passing Interface

Pourquoi MPI ?

- Faciliter la programmation par envoi de message
- Portabilité des programmes parallèles
- Utilisable avec une architecture **hétérogène**
- Cible stable pour des outils/langages de plus haut niveau

Que contient MPI ?

- Constantes, types et fonctions pour C et F77
- Opérations de communications point-à-point
- Opérations de communications collectives
- Groupes de processus
- Interface de suivi des performances
- Fonctions d'E/S parallèles (version 2)

MPI : Message Passing Interface

Pourquoi MPI ?

- Faciliter la programmation par envoi de message
- Portabilité des programmes parallèles
- Utilisable avec une architecture **hétérogène**
- Cible stable pour des outils/langages de plus haut niveau

Que contient MPI ?

- Constantes, types et fonctions pour C et F77
- Opérations de communications point-à-point
- Opérations de communications collectives
- Groupes de processus
- Interface de suivi des performances
- Fonctions d'E/S parallèles (version 2)

MPI : Message Passing Interface

Pourquoi MPI ?

- Faciliter la programmation par envoi de message
- Portabilité des programmes parallèles
- Utilisable avec une architecture **hétérogène**
- Cible stable pour des outils/langages de plus haut niveau

Que contient MPI ?

- Constantes, types et fonctions pour C et F77
- Opérations de communications point-à-point
- Opérations de communications collectives
- Groupes de processus
- Interface de suivi des performances
- Fonctions d'E/S parallèles (version 2)

MPI : Message Passing Interface

Ce que ne contient pas MPI

- Opérations de mémoire partagée (donc chaque processus a son propre espace d'adressage)
- Outils de programmation, débogueur
- Gestion des processus
- RPC ou *Remote procedure call*
- Processus légers (en fait, certaines implantations de MPI ne sont pas "thread safe")

MPI : Message Passing Interface

Ce que ne contient pas MPI

- Opérations de mémoire partagée (donc chaque processus a son propre espace d'adressage)
- Outils de programmation, débogueur
- Gestion des processus
- RPC ou *Remote procedure call*
- Processus légers (en fait, certaines implantations de MPI ne sont pas "thread safe")

MPI : Message Passing Interface

Ce que ne contient pas MPI

- Opérations de mémoire partagée (donc chaque processus a son propre espace d'adressage)
- Outils de programmation, débogueur
- Gestion des processus
- RPC ou *Remote procedure call*
- Processus légers (en fait, certaines implantations de MPI ne sont pas "thread safe")

MPI : Message Passing Interface

Ce que ne contient pas MPI

- Opérations de mémoire partagée (donc chaque processus a son propre espace d'adressage)
- Outils de programmation, débogueur
- Gestion des processus
- RPC ou *Remote procedure call*
- Processus légers (en fait, certaines implantations de MPI ne sont pas "thread safe")

MPI : Message Passing Interface

Ce que ne contient pas MPI

- Opérations de mémoire partagée (donc chaque processus a son propre espace d'adressage)
- Outils de programmation, débogueur
- Gestion des processus
- RPC ou *Remote procedure call*
- Processus légers (en fait, certaines implantations de MPI ne sont pas "thread safe")

Programmation en mode SPMD

SPMD ?

SPMD = *Single Program Multiple Data*.

Fonctionnement ?

1. On écrit un programme C avec des opérations MPI pour les communications
2. On le compile et le lance avec `mpirun -np n` où n est le nombre de processus voulu.

Programmation en mode SPMD

SPMD ?

SPMD = *Single Program Multiple Data*.

Fonctionnement ?

1. On écrit un programme C avec des opérations MPI pour les communications
2. On le compile et le lance avec `mpirun -np` n où n est le nombre de processus voulu.
3. n copies de l'exécutable sont créées avec pour numéros de processus $0, 1, \ldots, n-1$.
4. Les processus s'exécutent en mode asynchrone et se synchronisent lors d'échanges de messages.
5. Les processus se terminent tous (s'il n'y a pas eu blocage).

Programmation en mode SPMD

SPMD ?

SPMD = *Single Program Multiple Data*.

Fonctionnement ?

1. On écrit un programme C avec des opérations MPI pour les communications
2. On le compile et le lance avec `mpirun -np` n où n est le nombre de processus voulu.
3. n copies de l'exécutable sont créées avec pour numéros de processus $0, 1, \ldots, n-1$.
4. Les processus s'exécutent en mode asynchrone et se synchronisent lors d'échanges de messages.
5. Les processus se terminent tous (s'il n'y a pas eu blocage).

Programmation en mode SPMD

SPMD ?

SPMD = *Single Program Multiple Data*.

Fonctionnement ?

1. On écrit un programme C avec des opérations MPI pour les communications
2. On le compile et le lance avec `mpirun -np` n où n est le nombre de processus voulu.
3. n copies de l'exécutable sont créées avec pour numéros de processus $0, 1, \ldots, n-1$.
4. Les processus s'exécutent en mode asynchrone et se synchronisent lors d'échanges de messages.
5. Les processus se terminent tous (s'il n'y a pas eu blocage).

Programmation en mode SPMD

SPMD ?

SPMD = *Single Program Multiple Data*.

Fonctionnement ?

1. On écrit un programme C avec des opérations MPI pour les communications
2. On le compile et le lance avec `mpirun -np` n où n est le nombre de processus voulu.
3. n copies de l'exécutable sont créées avec pour numéros de processus $0, 1, \ldots, n-1$.
4. Les processus s'exécutent en mode asynchrone et se synchronisent lors d'échanges de messages.
5. Les processus se terminent tous (s'il n'y a pas eu blocage).

Programmation en mode SPMD

SPMD ?

SPMD = *Single Program Multiple Data*.

Fonctionnement ?

1. On écrit un programme C avec des opérations MPI pour les communications
2. On le compile et le lance avec `mpirun -np` n où n est le nombre de processus voulu.
3. n copies de l'exécutable sont créées avec pour numéros de processus $0, 1, \ldots, n-1$.
4. Les processus s'exécutent en mode asynchrone et se synchronisent lors d'échanges de messages.
5. Les processus se terminent tous (s'il n'y a pas eu blocage).

Programmation en mode SPMD

SPMD ?

SPMD = *Single Program Multiple Data*.

Fonctionnement ?

1. On écrit un programme C avec des opérations MPI pour les communications
2. On le compile et le lance avec `mpirun -np` n où n est le nombre de processus voulu.
3. n copies de l'exécutable sont créées avec pour numéros de processus $0, 1, \ldots, n-1$.
4. Les processus s'exécutent en mode asynchrone et se synchronisent lors d'échanges de messages.
5. Les processus se terminent tous (s'il n'y a pas eu blocage).

Types d'appels de fonctions MPI

Local, bloquant ou non

1. **Local :** ne nécessite aucune communication. Par ex. pour générer un objet MPI local.
2. **Non-local :** nécessite l'exécution d'une procédure MPI sur un autre processus. Par ex. un envoi de message (nécessite réception).
3. **Bloquant :** à la fin d'un tel appel, le programme appelant peut réutiliser les ressources utilisées dans l'appel : par ex. un *buffer*.
4. **Non-bloquant :** à la fin d'un tel appel, le programme doit attendre que l'opération se termine avant de réutiliser les ressources utilisées. Par ex : un envoi de message non-bloquant peut libérer le processus émetteur avant que le message ne soit *envoyé*.

Types d'appels de fonctions MPI

Local, bloquant ou non

1. **Local :** ne nécessite aucune communication. Par ex. pour générer un objet MPI local.
2. **Non-local :** nécessite l'exécution d'une procédure MPI sur un autre processus. Par ex. un envoi de message (nécessite réception).
3. **Bloquant :** à la fin d'un tel appel, le programme appelant peut réutiliser les ressources utilisées dans l'appel : par ex. un *buffer*.
4. **Non-bloquant :** à la fin d'un tel appel, le programme doit attendre que l'opération se termine avant de réutiliser les ressources utilisées. Par ex : un envoi de message non-bloquant peut libérer le processus émetteur avant que le message ne soit *envoyé*.

Types d'appels de fonctions MPI

Local, bloquant ou non

1. **Local :** ne nécessite aucune communication. Par ex. pour générer un objet MPI local.
2. **Non-local :** nécessite l'exécution d'une procédure MPI sur un autre processus. Par ex. un envoi de message (nécessite réception).
3. **Bloquant :** à la fin d'un tel appel, le programme appelant peut réutiliser les ressources utilisées dans l'appel : par ex. un *buffer*.
4. **Non-bloquant :** à la fin d'un tel appel, le programme doit attendre que l'opération se termine avant de réutiliser les ressources utilisées. Par ex : un envoi de message non-bloquant peut libérer le processus émetteur avant que le message ne soit *envoyé*.

Types d'appels de fonctions MPI

Local, bloquant ou non

1. **Local :** ne nécessite aucune communication. Par ex. pour générer un objet MPI local.
2. **Non-local :** nécessite l'exécution d'une procédure MPI sur un autre processus. Par ex. un envoi de message (nécessite réception).
3. **Bloquant :** à la fin d'un tel appel, le programme appelant peut réutiliser les ressources utilisées dans l'appel : par ex. un *buffer*.
4. **Non-bloquant :** à la fin d'un tel appel, le programme doit attendre que l'opération se termine avant de réutiliser les ressources utilisées. Par ex : un envoi de message non-bloquant peut libérer le processus émetteur avant que le message ne soit *envoyé*.

Types d'appels de fonctions MPI

Local, bloquant ou non

1. **Local :** ne nécessite aucune communication. Par ex. pour générer un objet MPI local.
2. **Non-local :** nécessite l'exécution d'une procédure MPI sur un autre processus. Par ex. un envoi de message (nécessite réception).
3. **Bloquant :** à la fin d'un tel appel, le programme appelant peut réutiliser les ressources utilisées dans l'appel : par ex. un *buffer*.
4. **Non-bloquant :** à la fin d'un tel appel, le programme doit attendre que l'opération se termine avant de réutiliser les ressources utilisées. Par ex : un envoi de message non-bloquant peut libérer le processus émetteur avant que le message ne soit *envoyé*.

Types d'appels de fonctions MPI

Collectif et groupes

1. **Collectif** Nécessite l'appel de la même fonction par tous les processus (d'un même *groupe* de processus). Par ex : *broadcast*, diffusion d'une valeur d'un processus à tous les autres.
2. Pour simplifier, dans ce qui suit il n'y a qu'un seul groupe de processus communicants : situation par défaut

Types d'appels de fonctions MPI

Collectif et groupes

1. **Collectif** Nécessite l'appel de la même fonction par tous les processus (d'un même *groupe* de processus). Par ex : *broadcast*, diffusion d'une valeur d'un processus à tous les autres.
2. Pour simplifier, dans ce qui suit il n'y a qu'un seul groupe de processus communicants : situation par défaut

Types d'appels de fonctions MPI

Collectif et groupes

1. **Collectif** Nécessite l'appel de la même fonction par tous les processus (d'un même *groupe* de processus). Par ex : *broadcast*, diffusion d'une valeur d'un processus à tous les autres.
2. Pour simplifier, dans ce qui suit il n'y a qu'un seul groupe de processus communicants : situation par défaut

C et MPI

- Les noms MPI ont pour préfixe MPI_ : à éviter pour le reste du programme.
- Les fonctions rendent un code d'erreur. En cas de succès : MPI_SUCCESS
- Arguments tableaux indexés à partir de 0.
- Arguments "Flags" : entiers encodant un booléen. 0=faux, non-zéro=vrai.
- Arguments de types union : pointeurs de type **void***.
- Arguments-adresses : type défini MPI comme MPI_Aint. C'est un entier de taille suffisante pour une adresse de l'architecture cible.
- Les constantes MPI peuvent être affectées à des variables C.

C et MPI

- Les noms MPI ont pour préfixe MPI_ : à éviter pour le reste du programme.
- Les fonctions rendent un code d'erreur. En cas de succès : MPI_SUCCESS
- Arguments tableaux indexés à partir de 0.
- Arguments "Flags" : entiers encodant un booléen. 0=faux, non-zéro=vrai.
- Arguments de types union : pointeurs de type void*.
- Arguments-adresses : type défini MPI comme MPI_Aint. C'est un entier de taille suffisante pour une adresse de l'architecture cible.
- Les constantes MPI peuvent être affectées à des variables C.

C et MPI

- Les noms MPI ont pour préfixe MPI_ : à éviter pour le reste du programme.
- Les fonctions rendent un code d'erreur. En cas de succès : MPI_SUCCESS
- Arguments tableaux indexés à partir de 0.
- Arguments "Flags" : entiers encodant un booléen. 0=faux, non-zéro=vrai.
- Arguments de types union : pointeurs de type **void***.
- Arguments-adresses : type défini MPI comme MPI_Aint. C'est un entier de taille suffisante pour une adresse de l'architecture cible.
- Les constantes MPI peuvent être affectées à des variables C.

C et MPI

- Les noms MPI ont pour préfixe MPI_ : à éviter pour le reste du programme.
- Les fonctions rendent un code d'erreur. En cas de succès : MPI_SUCCESS
- Arguments tableaux indexés à partir de 0.
- Arguments "Flags" : entiers encodant un booléen. 0=faux, non-zéro=vrai.
- Arguments de types union : pointeurs de type **void***.
- Arguments-adresses : type défini MPI comme MPI_Aint. C'est un entier de taille suffisante pour une adresse de l'architecture cible.
- Les constantes MPI peuvent être affectées à des variables C.

C et MPI

- Les noms MPI ont pour préfixe MPI_ : à éviter pour le reste du programme.
- Les fonctions rendent un code d'erreur. En cas de succès : MPI_SUCCESS
- Arguments tableaux indexés à partir de 0.
- Arguments "Flags" : entiers encodant un booléen. 0=faux, non-zéro=vrai.
- Arguments de types union : pointeurs de type **void**$*$.
- Arguments-adresses : type défini MPI comme MPI_Aint. C'est un entier de taille suffisante pour une adresse de l'architecture cible.
- Les constantes MPI peuvent être affectées à des variables C.

C et MPI

- Les noms MPI ont pour préfixe MPI_ : à éviter pour le reste du programme.
- Les fonctions rendent un code d'erreur. En cas de succès : MPI_SUCCESS
- Arguments tableaux indexés à partir de 0.
- Arguments "Flags" : entiers encodant un booléen. 0=faux, non-zéro=vrai.
- Arguments de types union : pointeurs de type **void***.
- Arguments-adresses : type défini MPI comme MPI_Aint. C'est un entier de taille suffisante pour une adresse de l'architecture cible.
- Les constantes MPI peuvent être affectées à des variables C.

C et MPI

- Les noms MPI ont pour préfixe MPI_ : à éviter pour le reste du programme.
- Les fonctions rendent un code d'erreur. En cas de succès : MPI_SUCCESS
- Arguments tableaux indexés à partir de 0.
- Arguments "Flags" : entiers encodant un booléen. 0=faux, non-zéro=vrai.
- Arguments de types union : pointeurs de type **void***.
- Arguments-adresses : type défini MPI comme MPI_Aint. C'est un entier de taille suffisante pour une adresse de l'architecture cible.
- Les constantes MPI peuvent être affectées à des variables C.

Programmer en C+MPI

Inialisation

- Pour utiliser MPI, un programme doit inclure le header de MPI **#include** "mpi.h"
- On compile avec `mpicc`
- On initialise avec MPI_Init(argc,argv)

2 "constantes" très utiles

- Avoir la taille du groupe
  int MPI_Comm_size(MPI_Comm, **int** *size) et donc pour l'ensemble des processeurs
  MPI_Comm_size(MPI_COMM_WORLD,&nprocs)
- idem pour le "pid" du processeur dans un groupe
  int MPI_Comm_rank(MPI_Comm, **int** *rank)

Programmer en C+MPI

Inialisation

- Pour utiliser MPI, un programme doit inclure le header de MPI **#include** "mpi.h"
- On compile avec `mpicc`
- On initialise avec MPI_Init(argc,argv)

2 "constantes" très utiles

- Avoir la taille du groupe
  int MPI_Comm_size(MPI_Comm, **int** *size) et donc pour l'ensemble des processeurs
  MPI_Comm_size(MPI_COMM_WORLD,&nprocs)
- idem pour le "pid" du processeur dans un groupe
  int MPI_Comm_rank(MPI_Comm, **int** *rank)

Programmer en C+MPI

Terminaison et exécution

- On termine avec MPI_Finalize(**void**)
- On lance avec `mpirun option "prog" "arguments du programme"` et :
  1. `-np 6` pour avoir 6 processus
  2. `-nolocal` pour n'avoir des processus que sur les noeuds
  3. `-machinefile "machine"` un fichier contenant le nom des noeuds
  4. `-arch LINUX` pour préciser le type d'architecture (ici LINUX)
  5. etc. (faire `man mpirun` pour plus d'options)

En général, plus de détails sur

- `http://www.idris.fr/data/cours/parallel/mpi/choix_doc.html` (programmes en Fortran mais aisés à traduire en C)
- `http://ph.ris.free.fr/cours/mpi.html`
- `http://www.cines.fr/IMG/pdf/cours_mpi_mtp_051206.pdf`

Programmer en C+MPI

Terminaison et exécution

- On termine avec MPI_Finalize(**void**)
- On lance avec `mpirun option "prog" "arguments du programme"` et :
  1. `-np 6` pour avoir 6 processus
  2. `-nolocal` pour n'avoir des processus que sur les noeuds
  3. `-machinefile "machine"` un fichier contenant le nom des noeuds
  4. `-arch LINUX` pour préciser le type d'architecture (ici LINUX)
  5. etc. (faire `man mpirun` pour plus d'options)

En général, plus de détails sur

- `http://www.idris.fr/data/cours/parallel/mpi/choix_doc.html` (programmes en Fortran mais aisés à traduire en C)
- `http://ph.ris.free.fr/cours/mpi.html`
- `http://www.cines.fr/IMG/pdf/cours_mpi_mtp_051206.pdf`

Déroulement du cours

Envoi bloquant

Type

int MPI_Send(**void**∗ buf, **int** count, MPI_Datatype datatype, **int** dest, **int** tag, M

Arguments

Les arguments sont tous des paramètres d'entrée à la fonction :

- buf adresse du début du *buffer*
- count nombre de valeurs à envoyer
- datatype type de chaque valeur
- dest rang du processus de destination
- tag étiquette pour identifier le message
- comm *communicator* : groupe de processus

Envoi bloquant

Type

int MPI_Send(**void**∗ buf, **int** count, MPI_Datatype datatype, **int** dest, **int** tag, M

Arguments

Les arguments sont tous des paramètres d'entrée à la fonction :

- buf adresse du début du *buffer*
- count nombre de valeurs à envoyer
- datatype type de chaque valeur
- dest rang du processus de destination
- tag étiquette pour identifier le message
- comm *communicator* : groupe de processus

Envoi bloquant

Remarques

1. Dans le cas le plus simple, on fixe comm à MPI_COMM_WORLD
2. Comme un processus peut recevoir plusieurs messages (et éventuellement plusieurs d'un même émetteur), tag peut servir à les identifier. Si leur identité est connue (par le contexte du programme), on peut fixer tag à n'importe quelle valeur lors de l'émission et à MPI_ANY_TAG lors de la réception.
3. L'émetteur place les valeurs à émettre dans des lieux successifs de la mémoire locale avant d'appeler MPI_Send
4. Si on fixe count=0, le message ne contiendra que ses informations de controle source, destination, tag et communicateur et pourra servir à une synchronisation ou à signifier un événement.

Envoi bloquant

Remarques

1. Dans le cas le plus simple, on fixe comm à MPI_COMM_WORLD
2. Comme un processus peut recevoir plusieurs messages (et éventuellement plusieurs d'un même émetteur), tag peut servir à les identifier. Si leur identité est connue (par le contexte du programme), on peut fixer tag à n'importe quelle valeur lors de l'émission et à MPI_ANY_TAG lors de la réception.
3. L'émetteur place les valeurs à émettre dans des lieux successifs de la mémoire locale avant d'appeler MPI_Send
4. Si on fixe count=0, le message ne contiendra que ses informations de controle source, destination, tag et communicateur et pourra servir à une synchronisation ou à signifier un événement.

Envoi bloquant

Remarques

1. Dans le cas le plus simple, on fixe comm à MPI_COMM_WORLD
2. Comme un processus peut recevoir plusieurs messages (et éventuellement plusieurs d'un même émetteur), tag peut servir à les identifier. Si leur identité est connue (par le contexte du programme), on peut fixer tag à n'importe quelle valeur lors de l'émission et à MPI_ANY_TAG lors de la réception.
3. L'émetteur place les valeurs à émettre dans des lieux successifs de la mémoire locale avant d'appeler MPI_Send
4. Si on fixe count=0, le message ne contiendra que ses informations de controle source, destination, tag et communicateur et pourra servir à une synchronisation ou à signifier un événement.

Envoi bloquant

Remarques

1. Dans le cas le plus simple, on fixe comm à MPI_COMM_WORLD
2. Comme un processus peut recevoir plusieurs messages (et éventuellement plusieurs d'un même émetteur), tag peut servir à les identifier. Si leur identité est connue (par le contexte du programme), on peut fixer tag à n'importe quelle valeur lors de l'émission et à MPI_ANY_TAG lors de la réception.
3. L'émetteur place les valeurs à émettre dans des lieux successifs de la mémoire locale avant d'appeler MPI_Send
4. Si on fixe count=0, le message ne contiendra que ses informations de controle source, destination, tag et communicateur et pourra servir à une synchronisation ou à signifier un événement.

Envoi bloquant

Remarques

1. Dans le cas le plus simple, on fixe comm à MPI_COMM_WORLD
2. Comme un processus peut recevoir plusieurs messages (et éventuellement plusieurs d'un même émetteur), tag peut servir à les identifier. Si leur identité est connue (par le contexte du programme), on peut fixer tag à n'importe quelle valeur lors de l'émission et à MPI_ANY_TAG lors de la réception.
3. L'émetteur place les valeurs à émettre dans des lieux successifs de la mémoire locale avant d'appeler MPI_Send
4. Si on fixe count=0, le message ne contiendra que ses informations de controle source, destination, tag et communicateur et pourra servir à une synchronisation ou à signifier un événement.

Envoi bloquant

Suite des arguments

1. Le buffer est constitué de count valeurs successives en mémoire, à partir de l'adresse buf. Il s'agit bien de *valeurs* et pas d'octets, ce qui rend l'opération plus portable. Il faut spécifier le type des valeurs, mais pas leur taille. C'est ce que font les valeurs MPI de l'argument datatype.
2. Voici les principaux et leurs équivalents C :

MPI_Datatype	type C
MPI_CHAR	**signed char**
MPI_SHORT	**signed short int**
MPI_INT	**signed int**
MPI_LONG	**signed long int**
MPI_FLOAT	**float**
MPI_DOUBLE	**double**

Envoi bloquant

Suite des arguments

1. Le buffer est constitué de count valeurs successives en mémoire, à partir de l'adresse buf. Il s'agit bien de *valeurs* et pas d'octets, ce qui rend l'opération plus portable. Il faut spécifier le type des valeurs, mais pas leur taille. C'est ce que font les valeurs MPI de l'argument datatype.
2. Voici les principaux et leurs équivalents C :

MPI_Datatype	type C
MPI_CHAR	**signed char**
MPI_SHORT	**signed short int**
MPI_INT	**signed int**
MPI_LONG	**signed long int**
MPI_FLOAT	**float**
MPI_DOUBLE	**double**

Envoi bloquant

Suite des arguments

1. Le buffer est constitué de count valeurs successives en mémoire, à partir de l'adresse buf. Il s'agit bien de *valeurs* et pas d'octets, ce qui rend l'opération plus portable. Il faut spécifier le type des valeurs, mais pas leur taille. C'est ce que font les valeurs MPI de l'argument datatype.
2. Voici les principaux et leurs équivalents C :

MPI_Datatype	type C
MPI_CHAR	**signed char**
MPI_SHORT	**signed short int**
MPI_INT	**signed int**
MPI_LONG	**signed long int**
MPI_FLOAT	**float**
MPI_DOUBLE	**double**

Réception bloquante

Type

int MPI_Recv(**void**∗ buf, **int** count, MPI_Datatype datatype, **int** source, **int** tag

Arguments

Les arguments buf et status sont des paramètres de sortie, les autres sont des paramètres d'entrée :

- le buffer de réception est constitué de count valeurs successives de type datatype situées à partir de l'adresse buf. Si moins de valeurs sont reçues, elles sont placées au début. S'il y en a trop, une erreur de débordement est déclenchée.

Réception bloquante

Type

int MPI_Recv(**void**∗ buf, **int** count, MPI_Datatype datatype, **int** source, **int** tag

Arguments

Les arguments buf et status sont des paramètres de sortie, les autres sont des paramètres d'entrée :

- le buffer de réception est constitué de count valeurs successives de type datatype situées à partir de l'adresse buf. Si moins de valeurs sont reçues, elles sont placées au début. S'il y en a trop, une erreur de débordement est déclenchée.

Réception bloquante

Type

int MPI_Recv(**void**∗ buf, **int** count, MPI_Datatype datatype, **int** source, **int** tag

Arguments

Les arguments buf et status sont des paramètres de sortie, les autres sont des paramètres d'entrée :

- le buffer de réception est constitué de count valeurs successives de type datatype situées à partir de l'adresse buf. Si moins de valeurs sont reçues, elles sont placées au début. S'il y en a trop, une erreur de débordement est déclenchée.

Réception bloquante

Suite arguments

- le message sélectionné est le premier dans la file de réception dont l'étiquette et le rang d'émetteur sont tag et source ; on peut fixer tag à MPI_ANY_TAG et/ou source à MPI_ANY_SOURCE pour ne pas tenir compte d'une ou l'autre de ces valeurs.
- status est une structure de type MPI_Status contenant trois champs status.MPI_SOURCE, status.MPI_TAG et status.MPI_ERROR contenant l'émetteur effectif, l'étiquette et le code d'erreur.

Réception bloquante

Suite arguments

- le message sélectionné est le premier dans la file de réception dont l'étiquette et le rang d'émetteur sont tag et source ; on peut fixer tag à MPI_ANY_TAG et/ou source à MPI_ANY_SOURCE pour ne pas tenir compte d'une ou l'autre de ces valeurs.
- `status` est une structure de type MPI_Status contenant trois champs status.MPI_SOURCE, status.MPI_TAG et status.MPI_ERROR contenant l'émetteur effectif, l'étiquette et le code d'erreur.

Réception bloquante

Suite arguments

- le message sélectionné est le premier dans la file de réception dont l'étiquette et le rang d'émetteur sont tag et source ; on peut fixer tag à MPI_ANY_TAG et/ou source à MPI_ANY_SOURCE pour ne pas tenir compte d'une ou l'autre de ces valeurs.
- status est une structure de type MPI_Status contenant trois champs status.MPI_SOURCE, status.MPI_TAG et status.MPI_ERROR contenant l'émetteur effectif, l'étiquette et le code d'erreur.

Réception bloquante

Suite arguments

- le message sélectionné est le premier dans la file de réception dont l'étiquette et le rang d'émetteur sont tag et source ; on peut fixer tag à MPI_ANY_TAG et/ou source à MPI_ANY_SOURCE pour ne pas tenir compte d'une ou l'autre de ces valeurs.
- `status` est une structure de type MPI_Status contenant trois champs status.MPI_SOURCE, status.MPI_TAG et status.MPI_ERROR contenant l'émetteur effectif, l'étiquette et le code d'erreur.

Réception bloquante

Pour connaître le nombre exact de valeurs reçues, il faut "extraire" cette information de `status` en lui appliquant la fonction suivante :

int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, **int** *count)

dont les arguments *status et datatype sont en entrée et *count en sortie.

Il faut que datatype soit le type des valeurs spécifié dans l'appel à MPI_Recv et que *status soit la valeur obtenue.

Exemple Envoie/reception bloquante

Exemple Proc. 0 envoie un message à Proc. 1 :

```
char msg[20];
int myrank, tag = 99;
MPI_Status status;
/* trouver son rang */
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
/* envoyer ou recevoir le message */
if (myrank == 0) {
 strcpy( msg, "Hello_there");
 MPI_Send(msg, strlen(msg)+1, MPI_CHAR, 1,
                          tag, MPI_COMM_WORLD);
} else if (myrank == 1)
   MPI_Recv(msg, 20, MPI_CHAR, 0, tag,
                    MPI_COMM_WORLD, &status);
```

Exemple Envoie/reception bloquante

On suppose qu'il y a deux processus. Ils sont donc numérotés 0 et 1. L'appel à MPI_Comm_rank rend le rang du processus parmi les deux. On différencie alors l'action.

Il est aussi possible d'écrire un programme indépendant du nombre de processus. Il faut alors demander à MPI le nombre de processus et effectuer une conditionnelle qui compare le rang local à ce nombre.

Sémantique des opérations bloquantes

- L'envoi dit bloquant ne bloque pas (nécessairement) l'émetteur jusqu'à la réception du message. Ce terme spécifie le protocole local d'interaction avec MPI et son buffer. Le transport du message est asynchrone, comme pour du courrier électronique. **L'envoi peut se terminer avant même que la réception débute**.
- `MPI_Recv` peut être lancé avant le lancement du MPI_Send correspondant. Son appel se termine quand le buffer de réception contient les nouvelles valeurs.
- MPI_Send peut être lancé avant ou après l'appel du MPI_Recv correspondant. Son appel se termine quand le message a été copié vers le réseau et quand le programme C peut réutiliser le buffer d'envoi.

Sémantique des opérations bloquantes

- L'envoi dit bloquant ne bloque pas (nécessairement) l'émetteur jusqu'à la réception du message. Ce terme spécifie le protocole local d'interaction avec MPI et son buffer. Le transport du message est asynchrone, comme pour du courrier électronique. **L'envoi peut se terminer avant même que la réception débute**.
- `MPI_Recv` peut être lancé avant le lancement du MPI_Send correspondant. Son appel se termine quand le buffer de réception contient les nouvelles valeurs.
- MPI_Send peut être lancé avant ou après l'appel du MPI_Recv correspondant. Son appel se termine quand le message a été copié vers le réseau et quand le programme C peut réutiliser le buffer d'envoi.

Sémantique des opérations bloquantes

- L'envoi dit bloquant ne bloque pas (nécessairement) l'émetteur jusqu'à la réception du message. Ce terme spécifie le protocole local d'interaction avec MPI et son buffer. Le transport du message est asynchrone, comme pour du courrier électronique. **L'envoi peut se terminer avant même que la réception débute**.
- `MPI_Recv` peut être lancé avant le lancement du MPI_Send correspondant. Son appel se termine quand le buffer de réception contient les nouvelles valeurs.
- MPI_Send peut être lancé avant ou après l'appel du MPI_Recv correspondant. Son appel se termine quand le message a été copié vers le réseau et quand le programme C peut réutiliser le buffer d'envoi.

Ordre d'acheminement

MPI garantit qu'entre un même émetteur et une même destination, les messages sont transmis dans l'ordre (logiquement comme un canal FIFO). Les opérations de réception pour un même émetteur sont traitées dans l'ordre.

Cependant 1. le réseau peut mélanger des messages de source/destination différentes et 2. un récepteur peut mélanger les sources desquelles il reçoit.

Envoi et réception combinés

On a souvent besoin de “permuter” les valeurs disponibles sur les processeurs. Chaque processeur doit alors envoyer et recevoir. Il existe un opération MPI_SENDRECV qui combine les deux **et dont l'effet est équivalent** à créer deux processus légers par processus : un pour l'émission, l'autre pour la réception. Cela simplifie la programmation et évite certaines conditions de blocage (possible par manque d'espace dans les buffers).

MPI_SENDRECV exécute une envoi bloquant et une réception bloquante. Les deux doivent utiliser le même groupe de processus, mais doivent avoir des tag différents. Le buffer d'envoi et le buffer de réception doivent être disjoints et peuvent avoir des longueurs et types différents (pourquoi ?).

Envoi et réception combinés

La fonction

int MPI_Sendrecv(**void** *sendbuf, **int** sendcount, MPI_Datatype sendtype, **int** dest, **int** sendtag, **void** *recvbuf, **int** recvcount,

Arguments

Les arguments sont tous des paramètres d'entrée sauf recvbuf et status :

sendbuf	adresse de début du buffer d'envoi
sendcount	nombre de valeurs à envoyer
sendtype	type des valeurs d'envoi
dest	processus de destination pour l'envoi
sendtag	étiquette du message d'envoi
recvbuf	adresse de début du buffer de réception
recvcount	nombre max. de valeurs à recevoir
recvtype	type des valeurs à recevoir
source	processus émetteur de la réception
recvtag	étiquette du message à recevoir
comm	groupe de processus
status	structure status comme pour MPI_Recv

Envoi et réception combinés

La fonction

int MPI_Sendrecv(**void** *sendbuf, **int** sendcount, MPI_Datatype sendtype, **int** dest, **int** sendtag, **void** *recvbuf, **int** recvcount,

Arguments

Les arguments sont tous des paramètres d'entrée sauf recvbuf et status :

sendbuf	adresse de début du buffer d'envoi
sendcount	nombre de valeurs à envoyer
sendtype	type des valeurs d'envoi
dest	processus de destination pour l'envoi
sendtag	étiquette du message d'envoi
recvbuf	adresse de début du buffer de réception
recvcount	nombre max. de valeurs à recevoir
recvtype	type des valeurs à recevoir
source	processus émetteur de la réception
recvtag	étiquette du message à recevoir
comm	groupe de processus
status	structure `status` comme pour MPI_Recv

Envoi et réception combinés

Optimisation

Certaines architectures permettent de poursuivre les calculs entre le lancement d'une opération de communication et sa terminaison. MPI offre une version non-bloquante des opérations d'envoi et réception des messages qui permet cela en vue d'améliorer les performances (*recouvrement calcul- communication*).

Fonctionnement

1. l'émetteur place les données dans le buffer d'envoi et déclenche la communication. Cet appel est non-bloquant et le processus est alors libre de travailler à autre chose.
2. plus tard l'émetteur demande à MPI si le buffer a été vidé et ne reprend la main que lorsque c'est le cas (appel de type bloquant).

Envoi et réception combinés

Optimisation

Certaines architectures permettent de poursuivre les calculs entre le lancement d'une opération de communication et sa terminaison. MPI offre une version non-bloquante des opérations d'envoi et réception des messages qui permet cela en vue d'améliorer les performances (*recouvrement calcul- communication*).

Fonctionnement

1. l'émetteur place les données dans le buffer d'envoi et déclenche la communication. Cet appel est non-bloquant et le processus est alors libre de travailler à autre chose.
2. plus tard l'émetteur demande à MPI si le buffer a été vidé et ne reprend la main que lorsque c'est la cas : appel de type bloquant.
3. pendant ce temps le récepteur lance la copie des données dans le buffer de réception et, plus tard, obtient un signal de fin de réception en ayant profité du temps intermédiaire.

Envoi et réception combinés

Optimisation

Certaines architectures permettent de poursuivre les calculs entre le lancement d'une opération de communication et sa terminaison. MPI offre une version non-bloquante des opérations d'envoi et réception des messages qui permet cela en vue d'améliorer les performances (*recouvrement calcul- communication*).

Fonctionnement

1. l'émetteur place les données dans le buffer d'envoi et déclenche la communication. Cet appel est non-bloquant et le processus est alors libre de travailler à autre chose.
2. plus tard l'émetteur demande à MPI si le buffer a été vidé et ne reprend la main que lorsque c'est la cas : appel de type bloquant.
3. pendant ce temps le récepteur lance la copie des données dans le buffer de réception et, plus tard, obtient un signal de fin de réception en ayant profité du temps intermédiaire.

Envoi et réception combinés

Optimisation

Certaines architectures permettent de poursuivre les calculs entre le lancement d'une opération de communication et sa terminaison. MPI offre une version non-bloquante des opérations d'envoi et réception des messages qui permet cela en vue d'améliorer les performances (*recouvrement calcul- communication*).

Fonctionnement

1. l'émetteur place les données dans le buffer d'envoi et déclenche la communication. Cet appel est non-bloquant et le processus est alors libre de travailler à autre chose.
2. plus tard l'émetteur demande à MPI si le buffer a été vidé et ne reprend la main que lorsque c'est la cas : appel de type bloquant.
3. pendant ce temps le récepteur lance la copie des données dans le buffer de réception et, plus tard, obtient un signal de fin de réception en ayant profité du temps intermédiaire.

Envoi et réception combinés

Optimisation

Certaines architectures permettent de poursuivre les calculs entre le lancement d'une opération de communication et sa terminaison. MPI offre une version non-bloquante des opérations d'envoi et réception des messages qui permet cela en vue d'améliorer les performances (*recouvrement calcul- communication*).

Fonctionnement

1. l'émetteur place les données dans le buffer d'envoi et déclenche la communication. Cet appel est non-bloquant et le processus est alors libre de travailler à autre chose.
2. plus tard l'émetteur demande à MPI si le buffer a été vidé et ne reprend la main que lorsque c'est la cas : appel de type bloquant.
3. pendant ce temps le récepteur lance la copie des données dans le buffer de réception et, plus tard, obtient un signal de fin de réception en ayant profité du temps intermédiaire.

Déroulement du cours

Introduction

Qu'est-ce ?

Une communication collective est un schéma de communication (un squelette) nécessitant TOUT les processeurs. Certains algorithmes de base comme la diffusion et la réduction de valeurs sont pré-définis par des fonctions MPI.

C'est-à-dire ?

Ces opérations ont des définitions cohérentes avec celles de l'envoi/réception de messages avec les différences :

1. La taille du message doit correspondre exactement à la taille du buffer de réception ;
2. Toutes ces opérations sont bloquantes.

Introduction

Qu'est-ce ?

Une communication collective est un schéma de communication (un squelette) nécessitant TOUT les processeurs. Certains algorithmes de base comme la diffusion et la réduction de valeurs sont pré-définis par des fonctions MPI.

C'est-à-dire ?

Ces opérations ont des définitions cohérentes avec celles de l'envoi/réception de messages avec les différences :

1. La taille du message doit correspondre exactement à la taille du buffer de réception ;
2. Toutes ces opérations sont bloquantes.
3. Il n'y a pas d'argument tag.

Introduction

Qu'est-ce ?

Une communication collective est un schéma de communication (un squelette) nécessitant TOUT les processeurs. Certains algorithmes de base comme la diffusion et la réduction de valeurs sont pré-définis par des fonctions MPI.

C'est-à-dire ?

Ces opérations ont des définitions cohérentes avec celles de l'envoi/réception de messages avec les différences :

1. La taille du message doit correspondre exactement à la taille du buffer de réception ;
2. Toutes ces opérations sont bloquantes.
3. Il n'y a pas d'argument tag.

Introduction

Qu'est-ce ?

Une communication collective est un schéma de communication (un squelette) nécessitant TOUT les processeurs. Certains algorithmes de base comme la diffusion et la réduction de valeurs sont pré-définis par des fonctions MPI.

C'est-à-dire ?

Ces opérations ont des définitions cohérentes avec celles de l'envoi/réception de messages avec les différences :

1. La taille du message doit correspondre exactement à la taille du buffer de réception ;
2. Toutes ces opérations sont bloquantes.
3. Il n'y a pas d'argument tag.

Introduction

Qu'est-ce ?

Une communication collective est un schéma de communication (un squelette) nécessitant TOUT les processeurs. Certains algorithmes de base comme la diffusion et la réduction de valeurs sont pré-définis par des fonctions MPI.

C'est-à-dire ?

Ces opérations ont des définitions cohérentes avec celles de l'envoi/réception de messages avec les différences :

1. La taille du message doit correspondre exactement à la taille du buffer de réception ;
2. Toutes ces opérations sont bloquantes.
3. Il n'y a pas d'argument tag.

Barrière de synchronisation

Le problème

Habituellement, le retour d'une fonction d'envoi bloquante ne signifie pas que le correspondant a terminé (mais simplement que le buffer d'envoi est libéré). On ne peut donc pas *garantir* la progression globale des processus entre les phases successives de l'algorithme.

La solution

Une manière de le faire est de les synchroniser tous par une *barrière de synchronisation* : un appel collectif dont aucun processus ne repart avant que tous aient appelé la même fonction (modèle BSP) :

int MPI_Barrier(MPI_Comm comm)

Barrière de synchronisation

Le problème

Habituellement, le retour d'une fonction d'envoi bloquante ne signifie pas que le correspondant a terminé (mais simplement que le buffer d'envoi est libéré). On ne peut donc pas *garantir* la progression globale des processus entre les phases successives de l'algorithme.

La solution

Une manière de le faire est de les synchroniser tous par une *barrière de synchronisation* : un appel collectif dont aucun processus ne repart avant que tous aient appelé la même fonction (modèle BSP) :

int MPI_Barrier(MPI_Comm comm)

Diffusion

Type et arguments

int MPI_Bcast(**void**∗ buffer, **int** count, MPI_Datatype datatype, **int** root, MPI_C

Cette fonction émet un message du processu de rang root vers tous les autres du groupe comm. La valeur de root doit être la même partout. Le contenu du buffer d'envoi du processus émetteur est copié sur les autres processus. Les arguments sont tous d'entrée sauf buffer qui est "IN-OUT".

Example

Diffuser un tableau array de 100 entiers à partir du processus 0 :

```
MPI_Comm comm=MPI_COMM_WORLD;
int array[100];
int root=0;
...
MPI_Bcast(array, 100, MPI_INT, root, comm)
```

Diffusion

Type et arguments

int MPI_Bcast(**void**∗ buffer, **int** count, MPI_Datatype datatype, **int** root, MPI_C

Cette fonction émet un message du processu de rang root vers tous les autres du groupe comm. La valeur de root doit être la même partout. Le contenu du buffer d'envoi du processus émetteur est copié sur les autres processus. Les arguments sont tous d'entrée sauf buffer qui est "IN-OUT".

Example

Diffuser un tableau array de 100 entiers à partir du processus 0 :

```
MPI_Comm comm=MPI_COMM_WORLD;
int array[100];
int root=0;
...
MPI_Bcast(array, 100, MPI_INT, root, comm)
```

Collecte

Type et arguments

En sens inverse, on peut demander à chaque processus d'envoyer une valeur vers une même cible :

int MPI_Gather(**void**∗ sendbuf, **int** sendcount, MPIDatatype sendtype, **void**∗ re

Seul recvbuf est un argument de sortie.
Chaque processus (dont CodeCroot lui-même) envoie le contenu de son buffer d'envoi vers le processus root qui les trie en ordre du rang de l'émetteur. Le buffer de réception n'est pas utilisé par la fonction sur les processus autres que root et recvcount est le nombre de valeurs à recevoir *de chaque correspondant*.

Example

Chaque processus envoie 100 entiers au processus 0.

```
MPI_Comm comm=MPI_COMM_WORLD;
int gsize, sendarray[100];
int *rbuf;
...
MPI_Comm_Size(comm,&gsize);
rbuf = (int *)malloc(gsize*100*sizeof(int));
MPI_Gather(sendarray,100,MPI_INT,rbuf,100,MPI_INT,0,comm);
```

Collecte

Type et arguments

En sens inverse, on peut demander à chaque processus d'envoyer une valeur vers une même cible :

int MPI_Gather(**void**∗ sendbuf, **int** sendcount, MPIDatatype sendtype, **void**∗ re

Seul recvbuf est un argument de sortie.
Chaque processus (dont CodeCroot lui-même) envoie le contenu de son buffer d'envoi vers le processus root qui les trie en ordre du rang de l'émetteur. Le buffer de réception n'est pas utilisé par la fonction sur les processus autres que root et recvcount est le nombre de valeurs à recevoir *de chaque correspondant*.

Example

Chaque processus envoie 100 entiers au processus 0.

```
MPI_Comm comm=MPI_COMM_WORLD;
int gsize, sendarray[100];
int *rbuf;
...
MPI_Comm_Size(comm,&gsize);
rbuf = (int *)malloc(gsize*100*sizeof(int));
MPI_Gather(sendarray,100,MPI_INT,rbuf,100,MPI_INT,0,comm);
```

Calcul parallèle des préfixes

Qu'est-ce ?

L'algorithme de calcul des préfixes est pré-programmé sous forme d'une opération collective MPI. Chaque processus possède initialement une valeur, le tout constitue un tableau réparti. Le résultat est le tableau des sommes partielles réparti dans le même ordre que l'entrée .

Types et arguments ?

int MPI_Scan(**void***sendbuf, **void***recvbuf, **int** count, MPI_datatype datatype,
Chaque processus émet et reçoit, il y a donc deux buffers. Le buffer de réception du processus i contiendra la réduction des valeurs placeées dans sendbuf par les processus $0, 1, \ldots, i$.

Calcul parallèle des préfixes

Qu'est-ce ?

L'algorithme de calcul des préfixes est pré-programmé sous forme d'une opération collective MPI. Chaque processus possède initialement une valeur, le tout constitue un tableau réparti. Le résultat est le tableau des sommes partielles réparti dans le même ordre que l'entrée .

Types et arguments ?

int MPI_Scan(**void**∗ sendbuf, **void**∗ recvbuf, **int** count, MPI_datatype datatype,
Chaque processus émet et reçoit, il y a donc deux buffers. Le buffer de réception du processus i contiendra la réduction des valeurs placeées dans sendbuf par les processus $0, 1, \ldots, i$.

Calcul parallèle des préfixes

Opérateurs prédéfinies

MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	somme
MPI_PROD	produit
MPI_LAND	et-logique
MPI_LOR	ou-logique
MPI_BAND	et bit-à-bit
MPI_BOR	ou bit-à-bit
MPI_LXOR	ou exclusif logique
MPI_BXOR	ou exclusif bit-à-bit
MPI_MAXLOC	valeur maximale et son lieu
MPI_MINLOC	valeur minimale et son lieu

A la semaine prochaine