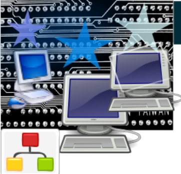


Introduction à la Programmation parallèle

– Institut Galilée –
— INFO 3 —

Camille Coti[1]
camille.coti@lipn.univ-paris13.fr

[1] Université de Paris XIII, CNRS UMR 7030, France

Plan du cours

1. **Introduction aux machines parallèles**
   - Pourquoi des machines parallèles ?
   - Modèles de mémoire pour machines parallèles
   - Exemples d'architectures
   - Parallélisation d'un programme
   - Performance du calcul parallèle
2. **Programmation de machines en mémoire partagée**
   - Programmation multithreadée
   - OpenMP
   - Scheduling
3. **Programmation de machines en mémoire distribuée**
   - Passage de messages
   - La norme MPI
   - Communications point-à-point
   - Communications collectives

Pourquoi des machines parallèles ?

Comment augmenter la puissance de calcul des machines

- Augmenter la fréquence d'horloge
- Augmenter le nombre d'opérations effectuées en un cycle

On se heurte aux limitation des techniques de fabrication des puces

Autre solution : augmenter le nombre d'unités de calcul

- Plusieurs pipelines
  - Architectures superscalaires
- Plusieurs processeurs logiques (avec leurs propres registres et PC, IC)
  - HyperThreading
- Plusieurs unités de calcul par puce
  - multi-cœur
- Plusieurs processeurs par carte mère
  - Symmetric Multi-Processor (SMP)
- Plusieurs machines reliées par un réseau
  - Clusters

Mémoire partagée

Plusieurs unités de calcul ont directement accès à la mémoire

- Threads
- Processus

Au niveau système :

- La mémoire doit être physiquement accessible par toutes les unités de calcul
- Une seule instance du système d'exploitation orchestre les accès mémoire

Exemple

Machine multi-cœur

- Un thread par cœur
- Les threads ont tous accès à la mémoire
- Programmation : pthread par exemple
  - Mémoire partagée

Mémoire distribuée

Chaque unité de calcul a son propre banc de mémoire

- Un processus par unité de calcul
- Les processus ne peuvent accéder qu'à *leur* banc de mémoire

Au niveau système :

- Les échanges de données entre processus sont explicites (passage de messages)
- Chaque unité de calcul est gérée par une instance du système d'exploitation

Exemple

Cluster

- Un processus par nœud
- Un CPU, un banc de mémoire et une carte réseau par nœud
- Programmation utilisant une bibliothèque de communications
  - Échanges de données via le réseau

Comparaison des deux modèles

Mise en œuvre

La mémoire partagée est peu scalable

- Nombre limité de cœurs par carte mère
- Scalabilité du scheduler du système d'exploitation

Programmation

La mémoire distribuée demande plus de travail au programmeur

- Mouvements de données *explicites*

La mémoire partagée est parfois faussement facile

- Deadlocks, race conditions...

Machines hybrides

Clusters de multi-cœurs:

- Plusieurs cœurs par nœud → mémoire partagée
- Plusieurs nœuds reliés par un réseau → mémoire distribuée

Cluster of workstations

Solution **économique**

- Composants produits en masse
  - PC utilisés pour les nœuds
  - Réseau Ethernet ou haute vitesse (InfiniBand, Myrinet...)
- Longtemps appelé "le supercalculateur du pauvre"

Exemples d'architectures

Supercalculateur massivement parallèle (MPP)

Solution spécifique

- Composants spécifiques
  - CPU différent de ceux des PC
  - Réseaux spécifique (parfois propriétaire)
  - Parfois sans disque dur
- Coûteux

Exemples d'architectures

Exemple : Cray XT5m

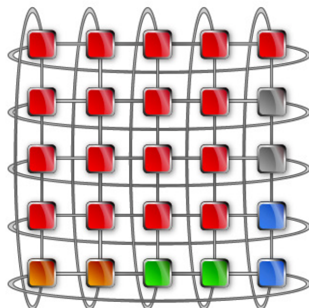
- CPU : deux AMD Istanbul
  - 6 cœurs chacun
  - 2 puces par machine
  - Empilées sur la même socket
  - Bus : crossbar
- Pas de disque dur

Réseau

- Propriétaire : SeaStar
- Topologie : tore 2D
- Connexion directe avec ses 4 voisins

Environnement logiciel

- OS : Cray Linux Environment
- Compilateurs, bibliothèques de calcul spécifiques (tunés pour l'architecture)
- Bibliothèques de communications réglées pour la machine



Top 500 www.top500.org

Classement des machines les plus rapides

- Basé sur un benchmark (LINPACK) effectuant des opérations typiques de calcul scientifique
- Permet de réaliser des statistiques
  - Tendances architecturales
  - Par pays, par OS...
  - Évolution !
- Depuis juin 1993, dévoilé tous les ans en juin et novembre

Dernier classement : juin 2010

1. Jaguar - Cray XT5-HE - Oak Ridge National Lab
2. Nebulae - Dawning TC3600 Blade - National Supercomputing Centre in Shenzhen (NSCS)
3. Roadrunner - IBM BladeCenter QS22/LS21 Cluster - Los Alamos National Lab
4. Kraken - Cray XT5-HE - Univ. of Tennessee / Oak Ridge National Lab
5. JUGENE - IBM Blue Gene/P - Forschungszentrum Juelich (FZJ)

Top 500 - Type de systèmes

Architecture Share Over Time
1993-2010

Systems: 0, 100, 200, 300, 400, 500

06/1993, 06/1994, 06/1995, 06/1996, 06/1997, 06/1998, 06/1999, 06/2000, 06/2001, 06/2002, 06/2003, 06/2004, 06/2005, 06/2006, 06/2007, 06/2008, 06/2009, 06/2010

TOP500 Releases

- MPP
- Cluster
- SMP
- Constellations
- Single Processor
- Others

Top 500 - Nombre de CPU

Number of Processors Share Over Time
1993-2010

Systems: 500, 400, 300, 200, 100, 0

Legend: 128k-, 64k-128k, 32k-64k, 16k-32k, 8k-16k, 4k-8k, 2049-4096, 1025-2048, 513-1024, 257-512, 129-256, 65-128, 33-64, 17-32, 9-16, 5-8

06/1993, 06/1994, 06/1995, 06/1996, 06/1997, 06/1998, 06/1999, 06/2000, 06/2001, 06/2002, 06/2003, 06/2004, 06/2005, 06/2006, 06/2007, 06/2008, 06/2009, 06/2010

TOP500 Releases

Top 500 - Nombre de CPU juin 2010

Number of Processors	Count	Share %	Rmax Sum (GF)	Rpeak Sum (GF)	Processor Sum
1025-2048	2	0.40 %	148800	164762	3072
2049-4096	111	22.20 %	3483053	4409819	380490
4k-8k	291	58.20 %	10257519	17638274	1660170
8k-16k	57	11.40 %	4605306	5827386	638093
16k-32k	18	3.60 %	3393408	4811462	497532
32k-64k	13	2.60 %	3963187	7252388	605494
64k-128k	3	0.60 %	2646400	3377921	303248
128k-	5	1.00 %	3937011	4988484	1043362
Totals	**500**	**100%**	**32434683.70**	**48470495.53**	**5131461**

Parallélisation d'un programme

Parallélisation du calcul entre les unités de calcul

But : diviser le travail entre les processus ou les threads disponibles

- Approche "divide and conquer"

Types de programmes parallèles

Classification de Flynn (1969)

- Single Instruction Single Data (SISD) : calcul séquentiel
- Multiple Instruction Single Data (MISD) : calcul vectoriel
- *Single Instruction Multiple Data (SIMD) : calcul parallèle*
- *Multiple Instruction Multiple Data (MIMD) : calcul parallèle*

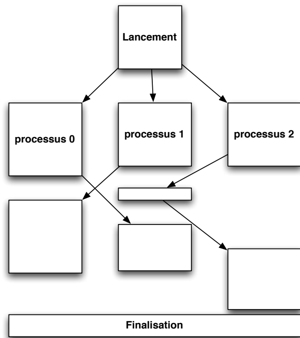
On traite plusieurs flux de données en parallèle

Mesure de la performance des programmes parallèles

Comment définir cette performance

Pourquoi parallélise-t-on ?

- Pour diviser un calcul qui serait trop long / trop gros sinon
- Diviser le problème ↔ diviser le temps de calcul ?



Sources de ralentissement

- Synchronisations entre processus
  - Mouvements de données
  - Attentes
  - Synchronisations
- Adaptations algorithmiques
  - L'algorithme parallèle peut être différent de l'algorithme séquentiel
  - Calculs supplémentaires

Efficacité du parallélisme ?

Définition

L'accélération d'un programme parallèle (ou *speedup*) représente le gain en rapidité d'exécution obtenu par son exécution sur plusieurs processeurs.

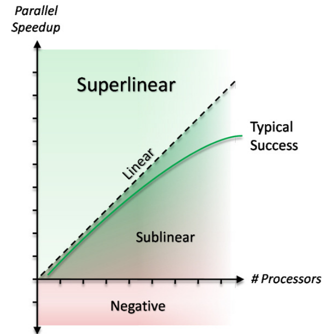
Mesure de l'accélération

On la mesure par le rapport entre le temps d'exécution du programme séquentiel et le temps d'exécution sur p processeurs

$$S_p = \frac{T_{seq}}{T_p}$$

Appréciation de l'accélération

- Accélération linéaire : parallélisme optimal
- Accélération sur-linéaire : attention
- Accélération sub-linéaire : ralentissement dû au parallélisme



Loi d'Amdahl

Décomposition d'un programme parallèle

Décomposition du temps d'exécution d'une application parallèle

- Une partie purement séquentielle ;
- Une partie parallélisable

Énnoncé

On note s la proportion parallélisable de l'exécution et p le nombre de processus. Le rendement est donné par la formule :

$$R = \frac{1}{(1-s) + \frac{s}{p}}$$

Remarques

- si $p \to \infty$: $R = \frac{1}{(1-s)}$
  - L'accélération est *toujours* limitée par la partie non-parallélisable du programme
- Si $(1-s) \to 0$, on a $R \sim p$: l'accélération est linéaire

UNIVERSITÉ PARIS 13 NORD

Passage à l'échelle (scalabilité)

Remarque préliminaire

On a vu avec la loi d'Amdahl que la performance augmente **théoriquement** lorsque l'on ajoute des processus.

- Comment augmente-t-elle en réalité ?
- Y a-t-il des facteurs limitants (goulet d'étranglement...)
- Augmente-t-elle à l'infini ?

Définition

Le *passage à l'échelle* d'un programme parallèle désigne l'augmentation des performances obtenues lorsque l'on ajoute des processus.

Obstacles à la scalabilité

- Synchronisations
- Algorithmes ne passant pas à l'échelle (complexité de l'algo)
  - Complexité en opérations
  - Complexité en communications

Passage à l'échelle (scalabilité)

La performance d'un programme parallèle a plusieurs dimensions

Scalabilité forte

On fixe la taille du problème et on augmente le nombre de processus

- Relative au speedup
- Si on a une hyperbole : scalabilité forte parfaite
  - On augmente le nombre de processus pour calculer plus vite

Scalabilité faible

On augmente la taille du problème avec le nombre de processus

- Le problème est à taille constante *par processus*
- Si le temps de calcul est constant : scalabilité faible parfaite
  - On augmente le nombre de processus pour résoudre des problèmes de plus grande taille

1. Introduction aux machines parallèles
   - Pourquoi des machines parallèles ?
   - Modèles de mémoire pour machines parallèles
   - Exemples d'architectures
   - Parallélisation d'un programme
   - Performance du calcul parallèle
2. **Programmation de machines en mémoire partagée**
   - Programmation multithreadée
   - OpenMP
   - Scheduling
3. Programmation de machines en mémoire distribuée
   - Passage de messages
   - La norme MPI
   - Communications point-à-point
   - Communications collectives

Accès mémoire

Tous les threads ont accès à une mémoire commune

- Modèle PRAM

Techniques de programmation

Utilisation de processus

- Création avec `fork()`
- Communication via un segment de mémoire partagée

Utilisation de threads POSIX

- Création avec `pthread_create()`, destruction avec `pthread_join()`
- Communication via un segment de mémoire partagée ou des variables globales dans le tas
  - Rappel : la pile est propre à chaque thread, le tas est commun

Utilisation d'un langage spécifique

- Exemple : OpenMP

Exemple : OpenMP

Fonctionnement

Langage d'annotations

- Le code est relativement peu modifié
- Directives de compilation
  - Utilisation de `#pragma` : si le compilateur ne connait pas, OpenMP est débrayé et le code fonctionne en séquentiel

Exemple : parallélisation d'une boucle

Calcul d'un maximum global sur un tableau

Algorithm 1: Calcul séquentiel du maximum d'un tableau

begin
- **Data**: Un tableau de taille N d'entiers positifs tab[]
- **Result**: Un entier MAX
- $MAX = 0$;
- **for** $i \leftarrow 0$ **to** N **do**
  - **if** $tab[i] > MAX$ **then** $MAX = tab[i]$;

end

Paréllélisation du calcul

Parallélisation de la boucle `for`

- On "tranche" l'intervalle sur lequel a lieu de calcul
- Les tranches sont réparties entre les threads

Annotations OpenMP

Sections parallèles

- `#pragma omp parallel` : début d'une section parallèle (fork)
- `#pragma omp for` : boucle for parallélisée

Synchronisations

- `#pragma omp critical` : section critique
- `#pragma omp barrier` : barrière de synchronisation

Visibilité des données

- Privée = visible uniquement de ce thread
- Partagée = partagée entre tous les threads
- Par défaut :
  - Ce qui est déclaré dans la section parallèle est privé
  - Ce qui est déclaré en-dehors est partagé

```
#pragma omp parallel private (tid) shared (result)
```

Compilation et exécution

En-têtes

```
#include <omp.h>
```

Compilation

Activation avec une option du compilateur

- Pour gcc : `-fopenmp`

Exéuction

Nombre de threads :

- Par défaut : découverte du nombre de cœurs et utilisation de tous
- Fixé par l'utilisateur via la variable d'environnement `$OMP_NUM_THREADS`

Exemple : HelloWorld

Code complet du Hello World

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int main() {
  int nbthreads, tid;

#pragma omp parallel private ( tid ) shared ( nbthreads )
  {
    tid = omp_get_thread_num();
    if( 0 == tid )
      nbthreads = omp_get_num_threads();
#pragma omp barrier
    printf( "Hello World!  I am thread %d/%d\ n", tid, nbthreads
);
  }

  return EXIT_SUCCESS;
}
```

Exemple : Calcul d'un maximum global

Approche naïve

On peut parallélise la boucle et écrire le résultat directement dans une variable partagée

Algorithme

max = 0
parfor i = 0 à N-1 :
  si max $<$ tab$[i]$: alors max $= tab[i]$

Problème : les accès à max doivent se faire dans une section critique

- Solution : utilisation de `#pragma omp critical`
- Séquentialisation des accès $\rightarrow$ séquentialisation de la boucle !

Meilleure approche

Calcul d'un maximum local puis à la fin du calcul, maximum global des maximums locaux

- code `parmax.c`

Options de découpage des boucles (scheduling)

Static

- Le découpage est fait à l'avance
- Des tranches de tailles égales sont attribuées aux threads
- Adapté aux boucles dont les itérations ont des temps de calcul équivalent

Dynamic

- Le découpage est fait à l'exécution
- Le scheduler attribue une tranche aux threads libres
- Attention à la taille des tranches : si trop petites, seul le thread 0 travaillera

Guided

- Similaire à dynamic
- Les tailles des tranches diminuent durant l'exécution

Utilisation

```
#pragma omp for schedule (type, taille)
```

1 Introduction aux machines parallèles
- Pourquoi des machines parallèles ?
- Modèles de mémoire pour machines parallèles
- Exemples d'architectures
- Parallélisation d'un programme
- Performance du calcul parallèle

2 Programmation de machines en mémoire partagée
- Programmation multithreadée
- OpenMP
- Scheduling

3 **Programmation de machines en mémoire distribuée**
- Passage de messages
- La norme MPI
- Communications point-à-point
- Communications collectives

Passage de messages

Envoi de messages *explicite* entre deux processus

- Un processus A envoie à un processus B
- A exécute la primitive : `send( dest, &msgptr )`
- B exécute la primitive : `recv( dest, &msgptr )`

Les deux processus émetteur-récepteur doivent exécuter une primitive, de réception pour le récepteur et d'envoi pour l'émetteur

Nommage des processus

On a besoin d'une façon unique de désigner les processus

- Association adresse / port $\rightarrow$ portabilité ?
- On utilise un **rang de processus** , unique, entre 0 et N-1

Tampons des messages

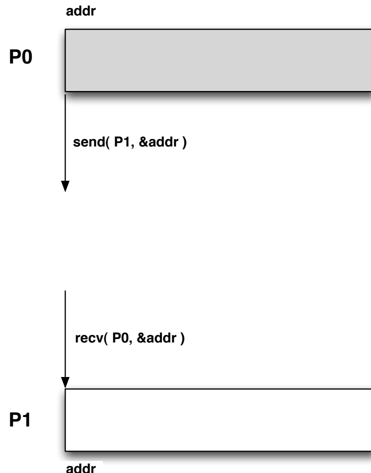
Chaque processus (émetteur et récepteur) a un tampon (buffer) pour le message

- La mémoire doit être allouée côté émetteur *et côté récepteur*
- On n'envoie pas plus d'éléments que la taille disponible en émission

Linéarisation des données

Les données doivent être sérialisées (marshalling) dans le tampon

- On envoie un tampon, un tableau d'éléments, une suite d'octets...



La norme MPI

Message Passing Interface

Norme *de facto* pour la programmation parallèle par passage de messages

- Née d'un effort de standardisation
  - Chaque fabriquant avait son propre langage
  - Portabilité des applications !
- Effort commun entre industriels et laboratoires de recherche
- But : être à la fois portable et offrir de bonnes performances

Implémentations

Portabilité des applications écrites en MPI

- Applis MPI exécutables avec n'importe quelle implémentation de MPI
  - Propriétaire ou non, fournie avec la machine ou non

Fonctions MPI

- Interface définie en C, C++, Fortran 77 et 90
- Listées et documentées dans la norme
- Commencent par `MPI_` et une lettre majuscule
  - Le reste est en lettres minuscules

Historique de MPI

Évolution

- Appel à contributions : SC 1992
- 1994 : MPI 1.0
  - Communications point-à-point de base
  - Communications collectives
- 1995 : MPI 1.1 (clarifications de MPI 1.0)
- 1997 : MPI 1.2 (clarifications et corrections)
- 1998 : MPI 2.0
  - Dynamicité
  - Accès distant à la mémoire des processus (RDMA)
- 2008 : MPI 2.1 (clarifications)
- 2009 : MPI 2.2 (corrections, peu d'additions)
- En cours : MPI 3.0
  - Tolérance aux pannes
  - Collectives non bloquantes
  - et d'autres choses

Désignation des processus

Communicateur

Les processus communiquant ensemble sont dans un **communicateur**

- Ils sont tous dans MPI_COMM_WORLD
- Chacun est tout seul dans son MPI_COMM_SELF
- MPI_COMM_NULL ne contient personne

Possibilité de créer d'autres communicateurs au cours de l'exécution

Rang

Les processus sont désignés par un **rang**

- Unique dans un communicateur donné
  - Rang dans MPI_COMM_WORLD= rang absolu dans l'application
- Utilisé pour les envois / réception de messages

Lancement

`mpiexec` lance les processus sur les machines distantes

- Lancement = exécution d'un programme sur la machine distante
  - Le binaire doit être accessible de la machine distante
- Possibilité d'exécuter un binaire différent suivant les rangs
  - "vrai" MIMD
- Transmission des paramètres de la ligne de commande

Redirections

Les entrées-sorties sont redirigées

- `stderr`, `stdout`, `stdin` sont redirigés vers le lanceur
- MPI-IO pour les I/O

Finalisation

`mpiexec` retourne quand tous les processus ont terminé normalement ou un seul a terminé anormalement (plantage, défaillance...)

Hello World en MPI

Début / fin du programme

Initialisation de la bibliothèque MPI

- `MPI_Init( &argc, &argv );`

Finalisation du programme

- `MPI_Finalize( );`

Si un processus quitte avant `MPI_Finalize( );`, ce sera considéré comme une erreur.

Ces deux fonctions sont OBLIGATOIRES !!!

Qui suis-je ?

Combien de processus dans l'application ?

- `MPI_Comm_size( MPI_COMM_WORLD, &size );`

Quel est mon rang ?

- `MPI_Comm_rank( MPI_COMM_WORLD, &rank );`

Hello World en MPI

Code complet

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int main( int argc, char** argv ) {
  int size, rank;
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  MPI_Comm_size( MPI_COMM_WORLD, &size );
  fprintf( stdout, "Hello, I am rank %d in %d\n'', rank, size );
  MPI_Finalize();
  return EXIT_SUCCESS;
}
```

Hello World en MPI

Compilation

Compilateur C : `mpicc`

- Wrapper autour du compilateur C installé
- Fournit les chemins vers le `mpi.h` et la lib MPI
- Équivalent à
  `gcc -L/path/to/mpi/lib -lmpi -I/path/to/mpi/include`

```
mpicc -o helloworld helloworld.c
```

Exécution

Lancement avec `mpiexec`

- On fournit une liste de machines (`machinefile`)
- Le nombre de processus à lancer

```
mpiexec --machinefile ./machinefile -n 4 ./helloworld
Hello, I am rank 1 in 4
Hello, I am rank 2 in 4
Hello, I am rank 0 in 4
Hello, I am rank 3 in 4
```

Communications point-à-point

Communications bloquantes

Envoi : `MPI_Send`

- `int MPI_Send( void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm )`

Réception : `MPI_Recv`

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

Données

- buf : tampon d'envoi / réception
- count : nombre d'éléments de type `datatype`
- datatype : type de données
  - Utilisation de datatypes MPI
  - Assure la portabilité (notamment 32/64 bits, environnements hétérogènes...)
  - Types standards et possibilité d'en définir de nouveaux

Identification des processus

- Utilisation du couple communicateur / rang
- En réception : possibilité d'utilisation d'une wildcard
  - `MPI_ANY_SOURCE`
  - Après réception, l'émetteur du message est dans le status

Identification de la communication

- Utilisation du tag
- En réception : possibilité d'utilisation d'une wildcard
  - `MPI_ANY_TAG`
  - Après réception, le tag du message est dans le status

Code complet

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

int main( int argc, char** argv ) {
  int rank;
  int token = 42;
  MPI_Status status;
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  if( 0 == rank ) {
    MPI_Send( &token, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    MPI_Recv( &token, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &status );
  } else if( 1 == rank ) {
    MPI_Recv( &token, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status );
    MPI_Send( &token, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
  }
  MPI_Finalize();
  return EXIT_SUCCESS;
}
```

Ping-pong entre deux processus

Remarques

- À un envoi correspond **toujours** une réception
  - Même communicateur, même tag
  - Rang de l'émetteur et rang du destinataire
- On utilise le rang pour déterminer ce que l'on fait
- On envoie des entiers → MPI_INT

Sources d'erreurs fréquentes

- Le datatype et le nombre d'éléments doivent être identiques en émission et en réception
  - On s'attend à recevoir ce qui a été envoyé
- Attention à la correspondance MPI_Send et MPI_Recv
  - Deux MPI_Send ou deux MPI_Recv = deadlock !

Communications non-bloquantes

But

La communication a lieu pendant qu'on fait autre chose

- Superposition communication/calcul
- Plusieurs communications simultanées sans risque de deadlock

Quand on a besoin des données, on attend que la communication ait été effectuée complètement

Communications

Envoi : `MPI_Isend`

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

Réception : `MPI_Irecv`

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

Communications non-bloquantes

Attente de complétion

Pour une communication :

- `int MPI_Wait( MPI_Request *request, MPI_Status *status )`

Attendre plusieurs communications : `MPI_{Waitall, Waitany, Waitsome}`

Test de complétion

Pour une communication :

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

Tester plusieurs communications : `MPI_{Testall, Testany, Testsome}`

Annuler une communication en cours

Communication non-bloquante identifiée par sa request

- `int MPI_Cancel(MPI_Request *request)`

Différences

- MPI_Wait est bloquant, MPI_Test ne l'est pas
- MPI_Test peut être appelé simplement pour entrer dans la bibliothèque MPI (lui redonner la main pour faire avancer des opérations)

Intérêt

On peut avoir besoin d'effectuer une opération sur *tous* les processus

- Synchronisation
- Déplacement de données
- Calcul global

Certaines communications collectives effectuent plusieurs de ces actions

Caractéristiques communes

Les collectives sont effectuées sur un *communicateur*

- Tous les processus de ce communicateur l'effectuent

Pour la norme MPI 1.x et 2.x, les collectives sont *bloquantes*

- Synchronisation avec effets de bord (attention à ne pas s'y fier)
- Pas de tag donc une seule collective à la fois

Synchronisation

Barrière

- `MPI_Barrier( MPI_Comm comm );`
- On ne sort de la barrière qu'une fois que tous les processus du communicateur y sont entrés
  - Assure une certaine synchronisation entre les processus

Diffusion d'une donnée

Broadcast

- `int MPI_Bcast( void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm );`
- Envoie une donnée (`buffer`)
  - À partir d'un processus racine (`root`)
  - Vers tous les processus du communicateur

Calcul global

Réduction vers une racine

- `int MPI_Reduce( void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm );`
- Effectue une opération (op)
  - Sur une donnée disponible sur tous les processus du communicateur
  - Vers la racine de la réduction (`root`)
- Opérations disponibles dans le standard (MPI_SUM, MPI_MAX, MPI_MIN...) et possibilité de définir ses propres opérations
  - La fonction doit être associative mais pas forcément commutative
- Pour mettre le résultat dans le tampon d'envoi (sur le processus racine) : `MPI_IN_PLACE`

Calcul global avec résultat sur tous les processus

Réduction globale

- `int MPI_Allreduce( void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm );`
- Similaire à `MPI_Reduce` sans la racine

Rassemblement

Concaténation du contenu des tampons

- `int MPI_Gather( void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm );`
- Les contenus des tampons sont envoyés vers la racine de la concaténation
- Possibilité d'utiliser des datatypes différents en envoi et en réception (attention, source d'erreurs)
- `recvbuf` ne sert que sur la racine
- Possibilité d'utiliser `MPI_IN_PLACE`

Rassemblement avec résultat sur tous les processus

Concaténation globale du contenu des tampons

- `int MPI_Allgather( void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm );`
- Similaire à `MPI_Gather` sans la racine

Distribution de données

Distribution d'un tampon vers plusieurs processus

- `int MPI_Scatter( void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm );`
- Des fractions de taille `sendcount` de tampon d'envoi disponible sur la racine sont envoyés vers tous les processus du communicateur
- Possibilité d'utiliser `MPI_IN_PLACE`

Distribution et concaténation de données

Distribution d'un tampon de tous les processus vers tous les processus

- `int MPI_Alltoall( void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm );`
- Sur chaque processus, le tampon d'envoi est découpé et envoyé vers tous les processus du communicateur
- Chaque processus reçoit des donnés de tous les autres processus et les concatène dans son tampon de réception
- PAS de possibilité d'utiliser `MPI_IN_PLACE`