Programmation de machines parallèles

Camille Coti
camille.coti@lipn.univ-paris13.fr

École Doctorale, Institut Galilée, Université de Paris XIII

Plan du cours

1. Programmation de système à mémoire partagée
2. Programmation de système à mémoire distribuée
3. Fonctionnalités avancées de MPI
4. Exemples d'applications MPI
5. Performance du calcul parallèle
6. Outils de profiling

Plan du cours

1. Programmation de système à mémoire partagée
   - Modèle : multithread et mémoire partagée
   - OpenMP
   - Scheduling
2. Programmation de système à mémoire distribuée
3. Fonctionnalités avancées de MPI
4. Exemples d'applications MPI
5. Performance du calcul parallèle
6. Outils de profiling

Programmation multithreadée

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

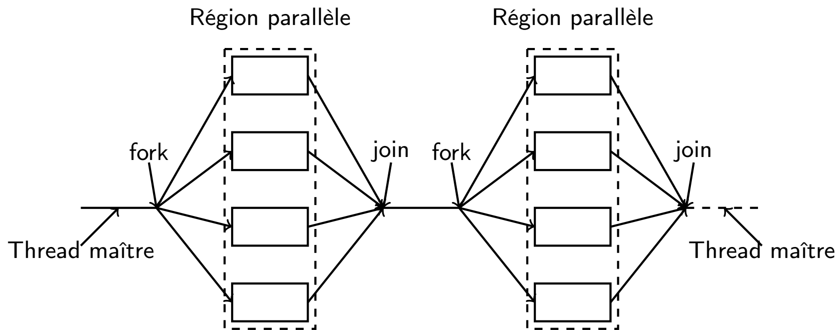
- Exemple : OpenMP

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** $MAX = tab[i]$;

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)
```

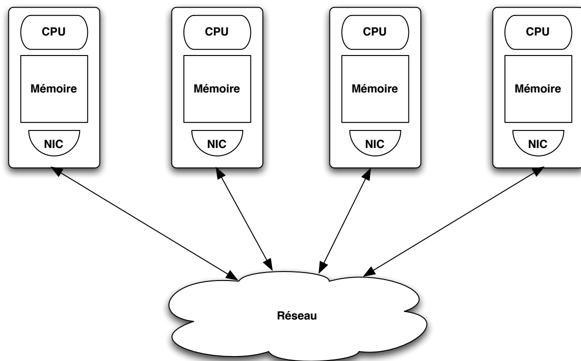
Plan du cours

1. Programmation de système à mémoire partagée
2. Programmation de système à mémoire distribuée
   - Modèle de mémoire distribuée
   - Passage de messages
   - La norme MPI
   - Communications point-à-point
   - Communications collectives
   - Distribution
   - MPI en Python
3. Fonctionnalités avancées de MPI
4. Exemples d'applications MPI
5. Performance du calcul parallèle
6. Outils de profiling

Mémoire distribuée

Nœuds de calcul distribués

- Chaque nœud possède un banc mémoire
- Lui seul peut y accéder
- Les nœuds sont reliés par un *réseau*



Mise en œuvre

Réseau d'interconnexion

Les nœuds ont accès à un *réseau d'interconnexion*

- Tous les nœuds y ont accès
- Communications point-à-point sur ce réseau

Espace d'adressage

Chaque processus a accès à sa mémoire propre *et uniquement sa mémoire*

- Il ne *peut pas* accéder à la mémoire des autres processus
- Pour échanger des données : communications point-à-point

C'est au programmeur de gérer les mouvements de données entre les processus

Système d'exploitation

Chaque nœud exécute sa propre instance du système d'exploitation

- Besoin d'un middleware supportant l'exécution parallèle
- Bibliothèque de communications entre les processus

Avantages et inconvénients

Avantages

- Modèle plus réaliste que PRAM
- Meilleur passage à l'échelle des machines
- Pas de problème de cohérence de la mémoire

Inconvénients

- Plus complexe à programmer
  - Intervention du programmeur dans le parallélisme
- Temps d'accès aux données distantes

Communications inter-processus

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 → portabilité ?
- On utilise un *rang de processus*, unique, entre 0 et N-1

Gestion des données

Tampons des messages

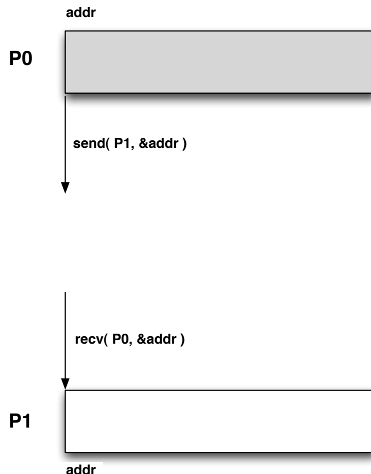
Chaque processus (émetteur et récepteur) a un tampon (buffer) pour le message

- La mémoire doit être allouée côté émetteur *et côté receveur*
- 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...



Autres modèles

Espace d'adressage global

Utilisation d'une mémoire partagée virtuelle

- Virtuelle car elle est en fait distribuée !
- Support d'un compilateur spécifique
  - Traduction des accès aux adresses distantes en communications
  - Passage de message "caché" à l'utilisateur, géré de faÃ§on transparente par l'environnement d'exécution
- Plus facile à programmer en apparence, mais difficile si on veut de bonnes performances
- Exemples : UPC, CoArray Fortran, Titanium...

Autres modèles

Accès distant à la mémoire

Les processus accèdent directement à la mémoire les uns des autres

- Les processus déposent des données dans la mémoire des autres processus ou vont lire dedans
- Nécessité d'un matériel particulier
- Gestion de la cohérence mémoire (risque de race conditions)
- Exemple : InfiniBand

NUMA en réseau

L'interconnexion entre les processeurs et les bancs de mémoire est fait par un réseau à faible latence

- Exemple : SGI Altix

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

Déploiement de l'application

Lancement

`mpiexec` lance les processus sur les machines distantes

- Lancement = exécution d'un programme sur la machine distante
  - Le binaire doit Ãªtre accessible de la machine distante
- Possibilité d'exécuter un binaire différent suivant les rangs
  - "vrai" MPMD
- 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_size( MPI_COMM_WORLD, &size );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  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 orig, int tag, MPI_Comm comm, MPI_Status *status )`

Communications point-à-point

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

Ping-pong entre deux processus

Code complet

```
#include <stdio.h>
#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 orig, 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 Ãªtre appelé simplement pour entrer dans la bibliothèque MPI (lui redonner la main pour faire avancer des opérations)

Sémantique des communications collectives

Tous les processus participent à la communication collective

- En MPI : lié à la notion de communicateur
- On effectue une communication collective *sur un communicateur*
  - MPI_COMM_WORLDou autre
  - Utilité de bien définir ses communicateurs !

Sémantique des communications collectives

Tous les processus participent à la communication collective

- En MPI : lié à la notion de communicateur
- On effectue une communication collective *sur un communicateur*
  - MPI_COMM_WORLDou autre
  - Utilité de bien définir ses communicateurs !

Un processus sort de la communication collective une fois qu'il a terminé *sa participation* à la collective

- Aucune garantie sur l'avancée globale de la communication collective
- Dans certaines communications collectives, un processus peut avoir terminé avant que d'autres processus n'aient commencé
- Le fait qu'un processus ait terminé ne signifie pas que la collective est terminée !
- Pas de synchronisation (sauf pour certaines communications collectives)

Sémantique des communications collectives

Tous les processus participent à la communication collective

- En MPI : lié à la notion de communicateur
- On effectue une communication collective *sur un communicateur*
  - MPI_COMM_WORLDou autre
  - Utilité de bien définir ses communicateurs !

Un processus sort de la communication collective une fois qu'il a terminé *sa participation* à la collective

- Aucune garantie sur l'avancée globale de la communication collective
- Dans certaines communications collectives, un processus peut avoir terminé avant que d'autres processus n'aient commencé
- Le fait qu'un processus ait terminé ne signifie pas que la collective est terminée !
- Pas de synchronisation (sauf pour certaines communications collectives)

Bloquant (pour le moment)

- Quelques projets de communications collectives non-bloquantes (NBC, MPI 3)
- On entre dans la communication collective et on n'en ressort que quand on a terminé sa participation à la communication

Exemple de communication collective : diffusion avec MPI

Diffusion avec MPI : `MPI_Bcast`

- Diffusion d'un processus vers les autres : définition d'un processus racine (root)
- On envoie un tampon de N éléments d'un type donné, sur un communicateur

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

int main( int argc, char** argv ) {
  int size, rank, token;

  MPI_Init( &argc, &argv );

  MPI_Comm_size( MPI_COMM_WORLD, &size );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  if( 0 == rank ) {
    token = getpid();
  }
  MPI_Bcast( &token, 1, MPI_INT, 0, MPI_COMM_WORLD );

  MPI_Finalize();

  return EXIT_SUCCESS;
}
```

Exemple de communication collective : diffusion avec MPI

```
MPI_Bcast( &token, 1, MPI_INT, 0, MPI_COMM_WORLD );
```

Le processus 0 diffuse un entier (`token`) vers les processus du communicateur MPI_COMM_WORLD

- Avant la communication collective : `token` est initialisé uniquement sur 0
- Tous les processus sauf 0 reçoivent quelque chose dans leur variable `token`
- Après la communication collective : tous les processus ont la mÃªme valeur dans leur variable `token` locale

Tous les processus du communicateur concerné doivent appeler `MPI_Bcast`

- Sinon : deadlock

Réduction

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`

Réduction avec redistribution du résultat

Sémantique : le résultat du calcul est disponible sur tous les processus du communicateur

- `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

Équivalent à :

- Un Reduce
- Suivi d'un Broadcast (fan-in-fan-out)

Ce qui serait une implémentation très inefficace !

Distribution

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`

Concaténation vers un point

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`

Concaténation avec redistribution du résultat

- `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

Barrière de synchronisation

Sémantique : un processus ne sort de la barrière qu'une fois que tous les autres processus y sont entrés

- `MPI_Barrier( MPI_Comm comm );`
- Apporte une certaine synchronisation entre les processus : quand on dépasse ce point, on sait que tous les autres processus l'ont au moins atteint
- Équivalent à un Allgather avec un message de taille nulle

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`

Utilisation de MPI en Python

Bindings non-officiels

- Par exemple : mpi4py

Lancement : on lance l'interpréteur avec `mpiexec`

```
$ mpiexec -n 8 python helloWorld.py
```

Import du module :

```
from mpi4py import MPI
```

Appels de fonctions MPI en Python

Les opérations sont toujours effectuées **sur un communicateur donné**

- Communicateurs fournis par la norme :
  - MPI.COMM_WORLD
  - MPI.COMM_SELF
  - MPI.COMM_NULL
- Objet de type mpi4py.MPI.Intracomm
  - On appelle les fonctions de cet objet
  - La classe Intracomm hérite les méthodes de communications point-à-point et collectives de la classe Comm

Attention : **pas** d'appel à MPI_Init(), MPI_Finalize()

```
#!/bin/python
from mpi4py import MPI

def main():
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()
    print "hello from " + str( rank ) + " in " + str( size )


if __name__ == "__main__":
    main()
```

Communications point-à-point

Deux familles de routines de communications :

- pour envoyer des objets Python : le nom de la routine commence par une minuscule
- pour envoyer des buffers (plus rapide) : le nom de la routine commence par une majuscule

Attention : il existe des fonctions d'envoi non-bloquantes (Isend/isend), mais pas de fonctions de réception.

```
#!/bin/python
from mpi4py import MPI

def main():
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    TAG = 40
    if 0 == rank:
        token = "Bonjour"
        comm.send( token, dest = 1, tag = TAG )
    if 1 == rank:
        token = comm.recv( source = 0, tag = TAG )
    print str( rank ) + "] Token : " + token

if __name__ == "__main__":
    main()
```

Plan du cours

1. Programmation de système à mémoire partagée
2. Programmation de système à mémoire distribuée
3. Fonctionnalités avancées de MPI
   - Datatypes
   - Réductions
4. Exemples d'applications MPI
5. Performance du calcul parallèle
6. Outils de profiling

Utilisation des datatypes MPI

Principe

- On définit le datatype
  - MPI_Type_contiguous, MPI_Type_vector, MPI_Type_hvector, MPI_Type_indexed, MPI_Type_hindexed, MPI_Type_struct
- On le commit
  - MPI_Type_commit
- On le libère à la fin
  - MPI_Type_free

Combinaison des types de base

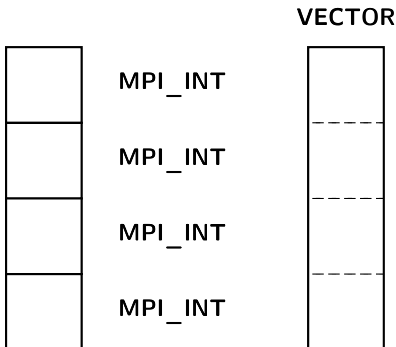
MPI_CHAR, MPI_SHORT, MPI_INT, MPI_LONG, MPI_UNSIGNED_CHAR, MPI_UNSIGNED_SHORT, MPI_UNSIGNED_LONG, MPI_UNSIGNED, MPI_FLOAT, MPI_DOUBLE, MPI_LONG_DOUBLE, MPI_BYTE

Construction de datatypes MPI

Données contigües

On crée un block d'éléments :

- `int MPI_Type_contiguous( int count, MPI_Datatype oldtype, MPI_Datatype *newtype );`



Travaux pratiques 5

Implémenter l'anneau à jeton en faisant circuler des vecteurs en utilisant un MPI_Datatype (un seul élément sera envoyé à chaque communication, quel que soit la taille des données envoyées).

Construction de datatypes MPI

Vecteur de données

On crée un vecteur d'éléments :

- `int MPI_Type_vector( int count, int blocklength, int stride MPI_Datatype oldtype, MPI_Datatype *newtype );`

On aggrège des blocs de *blocklength* éléments séparés par un vide de *stride* éléments.

Construction de datatypes MPI

Structure générale

On donne les éléments, leur nombre et l'offset auquel ils sont positionnés.

- `int MPI_Type_struct( int count, int *array_of_blocklengths, MPI_Aint *array_of_displacements, MPI_Datatype *array_of_types, MPI_Datatype *newtype);`

Exemple

On veut créer une structure MPI avec un entier et deux flottants à double précision

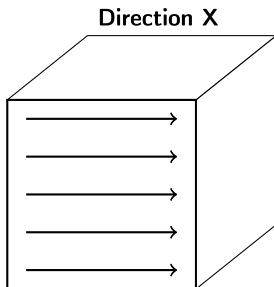
- count = 2 : on a deux éléments
- array_of_blocklengths = 1, 2 : on a 1 entier et 2 doubles
- array_of_displacements = 0, sizeof(int), sizeof(int) + sizeof(double) (ou utilisation de MPI_Address sur une instance de cette structure)
- array_of_types = MPI_INT, MPI_DOUBLE

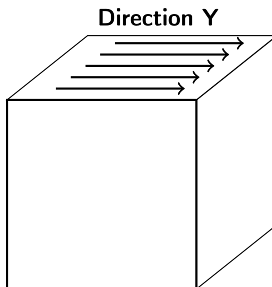
Exemple d'utilisation : FFT 3D

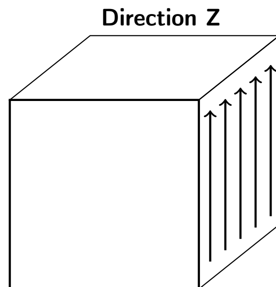
La FFT 3D

Calcul de la transformée de Fourier 3D d'une matrice 3D

- FFT dans une dimension
- FFT dans la deuxième dimension
- FFT dans la troisième dimension







Parallélisation de la FFT 3D

Noyau de calcul : la FFT 1D

La FFT 1D est parallélisable

- Chaque vecteur peut être calculé indépendamment des autres

On veut calculer chaque vecteur *séquentiellement* sur *un seul processus*

- Direct pour la 1ere dimension
- Nécessite une transposition pour les dimensions suivantes

Transposition de la matrice

On effectue une rotation de la matrice

- Redistribution des vecteurs
  - All to All
- Rotation des points (opération locale)

Rotation des points

Algorithme

```
MPI_Alltoall(tab 2 dans tab 1 )
for( i = 0 ; i < c ; ++i ) {
  for( j = 0 ; j < b ; ++j ) {
    for( k = 0 ; k < a ; ++k ) {
      tab2[i][j][k] = tab2[i][k][j];
    }
  }
}
```

Complexité

- Le Alltoall coûte cher
- La rotation locale est en $O(n^3)$

On essaye d'éviter le coût de cette rotation en la faisant faire par la bibliothèque MPI

Dataypes non-contigüs

Sérialisation des données en mémoire

La matrice est sérialisée en mémoire

- Vecteur$[0][0]$, Vecteur$[0][1]$, Vecteur$[0][2]$... Vecteur$[1][0]$, etc

Rotation des données sérialisées

x	x	x	x	$\rightarrow$	x	o	o	o
o	o	o	o		x	o	o	o
o	o	o	o		x	o	o	o
o	o	o	o		x	o	o	o

Utilisation d'un datatype MPI

C'est l'écart entre les points des vecteur qui change avant et après la rotation
$\rightarrow$ on définit un dataype différent en envoi et en réception.
Avantages :

- La rotation est faite par la bibliothèque MPI
- Gain d'une copie (buffer) au moment de l'envoi / réception des éléments (un seul élément au lieu de plusieurs)

Définition d'opérations

Syntaxe

- `int MPI_Op_create( MPI_User_function *function, int commute, MPI_Op *op );`

On fournit un pointeur sur fonction. La fonction doit être associative et peut être commutative ou pas. Elle a un prototype défini.

Exemple

- Définition d'une fonction :

```
void addem( int *, int *, int *, MPI_Datatype * );
void addem( int *invec, int *inoutvec, int *len, MPI_Datatype
*dtype ){
  int i;
  for ( i = 0 ; i < *len ; i++ )
    inoutvec[i] += invec[i];
}
```

- Déclaration de l'opération MPI :

```
MPI_Op_create( (MPI_User_function *)addem, 1, &op );
```

Exemple d'utilisation : TSQR

Definition

La *décomposition QR* d'une matrice A est une décomposition de la forme

$$A = QR$$

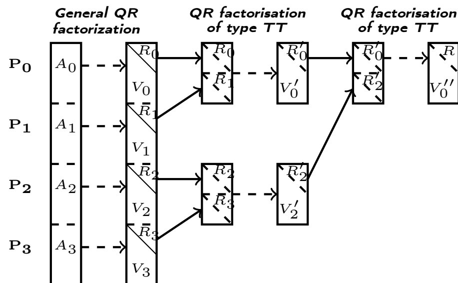
Où Q est une matrice orthogonale ($QQ^T = I$)
et R est une matrice triangulaire supérieure.

TSQR basé sur CAQR

Algorithme à évitement de communications pour matrices "tall and skinny" (hauteur $>>$ largeur)

- On calcule plus pour communiquer moins (optimal en nombre de communications)
  - Les communications coûtent cher, pas les flops
- Calcul de facto QR partielle sur des sous-domaines (en parallèle), recomposition 2 à 2, facto QR, etc

Algorithme TSQR



Algorithme

Sur un arbre binaire :

- QR sur sa sous-matrice
- Communication avec le voisin
  - Si rang pair : réception de $r+1$
  - Si rang impair : envoi à $r-1$
- Si rang impair : fin

Arbre de réduction

Opération effectuée

On effectue à chaque étape de la réduction une factorisation QR des deux facteurs R mis l'un au-dessus de l'autre :

$$R = QR(R_1, R_2)$$

- C'est une opération binaire
  - Deux matrices triangulaires en entrée, une matrice triangulaire en sortie
- Elle est associative

$$QR(R_1, QR(R_2, R_3)) = QR(QR(R_1, R_2), R_3)$$

- Elle est commutative

$$QR(R_1, R_2) = QR(R_2, R_1)$$

Utilisation de MPI_Reduce

L'opération remplit les conditions pour être une MPI_Op dans un MPI_Reduce

- Définition d'un datatype pour les facteurs triangulaires supérieurs R
- Utilisation de ce datatype et de l'opération QR dans un MPI_Reduce

Plan du cours

1. Programmation de système à mémoire partagée
2. Programmation de système à mémoire distribuée
3. Fonctionnalités avancées de MPI
4. Exemples d'applications MPI
   - Maître-esclaves
   - Découpage en grille
   - Utilisation d'une topologie
5. Performance du calcul parallèle
6. Outils de profiling

Maître-esclave

Distribution des données

Le maître distribue le travail aux esclaves

- Le maître démultiplexe les données, multiplexe les résultats
- Les esclaves ne *communiquent pas* entre eux

Efficacité

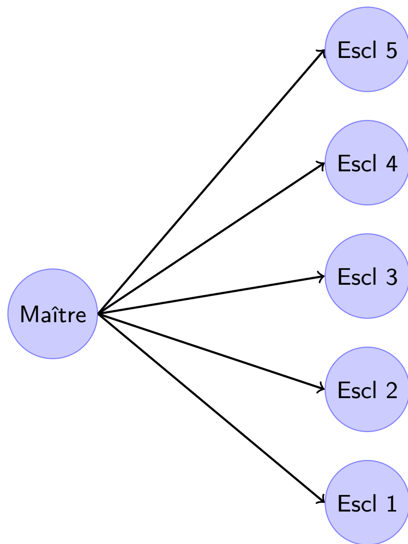
Files d'attentes de données et résultats au niveau du maître

- On retrouve la partie séquentielle de la loi d'Amdahl
- Communications: maître $\leftrightarrow$ esclaves
- Les esclaves ne travaillent que quand ils attendent des données ou qu'ils envoient leurs résultats

Seuls les esclaves participent effectivement au calcul

- Possilibité d'un bon speedup à grande échelle (esclaves $>>$ maître) *si les communications sont rares*
- Peu rentable pour quelques processus
- Attention au bottleneck au niveau du maître

Maître-esclave



Équilibrage de charge

Statique :

- Utilisation de `MPI_Scatter` pour distribuer les données
- `MPI_Gather` pour récupérer les résultats

Dynamique :

- Mode *pull* : les esclaves demandent du travail
- Le maître envoie les chunks 1 par 1

Travaux pratiques 2

Implémenter un squelette de maitre-esclave utilisant les deux méthodes. Avantages et inconvénients ?

Découpage en grille

Grille de processus

On découpe les données et on attribue un processus à chaque sous-domaine

Décomposition 1D

Découpage en bandes

0	1	2	3

Décomposition 2D

Découpage en rectangles, plus scalable

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Ghost region

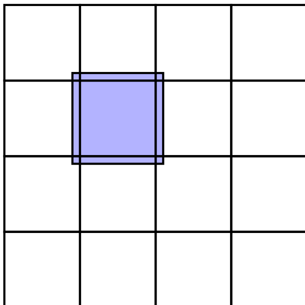
Frontières entre les sous-domaines

Un algorithme peut avoir besoin des valeurs des points voisins pour calculer la valeur d'un point

- Traitement d'images (calcul de gradiant...), automates cellulaires...

Réplication des données situées à la frontière

- Chaque processus dispose d'un peu des données des processus voisins
- Mise à jour à la fin d'une étape de calcul



Travaux pratiques 3

Utilisation d'une ghost region dans le maître-esclave dynamique en utilisant une décomposition 1D.

Utilisation d'une topologie

Décomposition en structure géométrique

On transpose un communicateur sur une topologie cartésienne

- `int MPI_Cart_create ( MPI_Comm comm_old, int ndims, int *dims, int *periods, int reorder, MPI_Comm *comm_cart );`

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

En 2D

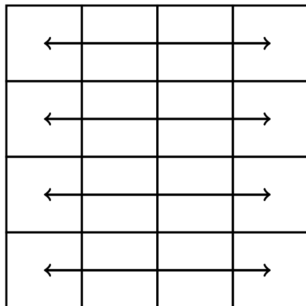
- `comm_old` : le communicateur de départ
- `ndims` : ici 2
- `dims` : nombre de processus dans chaque dimension (ici {4, 4})
- `periods` : les dimensions sont-elles périodiques
- `reorder` : autoriser ou non de modifier les rangs des processus
- `comm_cart` : nouveau communicateur

Utilisation d'une topologie

Extraction de sous-communicateurs

Communicateur de colonnes, de rangées...

- `int MPI_Cart_sub( MPI_Comm comm_old, int *remain_dims, MPI_Comm *comm_new );`



Communicateurs de lignes

- `comm_old` : le communicateur de départ
- `remain_dims` : quelles dimensions sont dans le communicateur ou non
- `comm_new` : le nouveau communicateur

Travaux pratiques 4

Implémenter la grille 2D de gauche et faire un broadcast sur chaque rangée.

Plan du cours

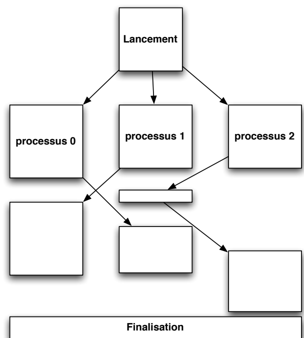
1. Programmation de système à mémoire partagée
2. Programmation de système à mémoire distribuée
3. Fonctionnalités avancées de MPI
4. Exemples d'applications MPI
5. Performance du calcul parallèle
6. Outils de profiling

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 $\leftrightarrow$ 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 ?

Accélération

Définition

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

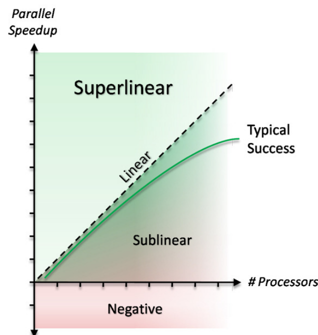
Mesure de l'accélération

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

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

Plan du cours

1. Programmation de système à mémoire partagée
2. Programmation de système à mémoire distribuée
3. Fonctionnalités avancées de MPI
4. Exemples d'applications MPI
5. Performance du calcul parallèle
6. Outils de profiling

Traces et analyses de traces

Scalasca : outil de profiling

- Comilation : ajout d'éléments d'instrumentation
- Non-invasif : on ne modifie pas le code

Compilation en vue d'une instrumentation avec Scalasca :

```
$ scalasca --instrument mpicc -o matmul matmul.c
```

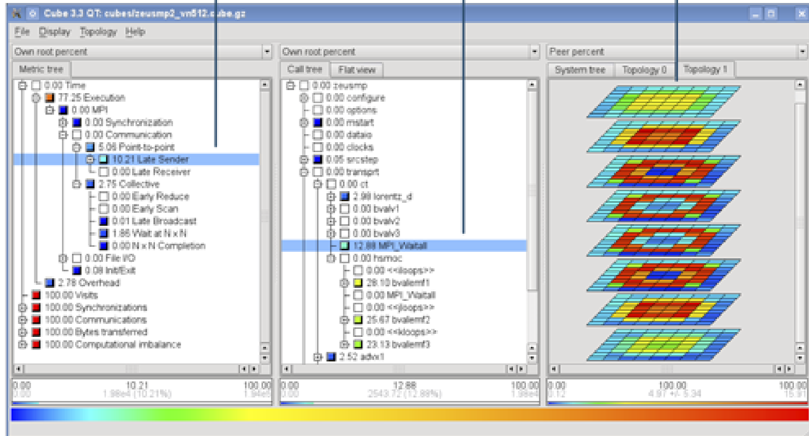
Réservation en mode interactif :

```
$ salloc -N 8 bash
```

Lancement d'un programme MPI :

```
$  scalasca --analyze mpiexec /home/dist/nicolas.greneche/O
  S=C=A=N: Scalasca 1.4.1 runtime summarization
  S=C=A=N: Warning: Number of processes not known!
  S=C=A=N: ./epik_matmul_O_sum experiment archive
  S=C=A=N: Mon Jun 11 11:33:18 2012: Collect start
```

Scalasca : affichage



Vampir

Outil d'analyse graphique

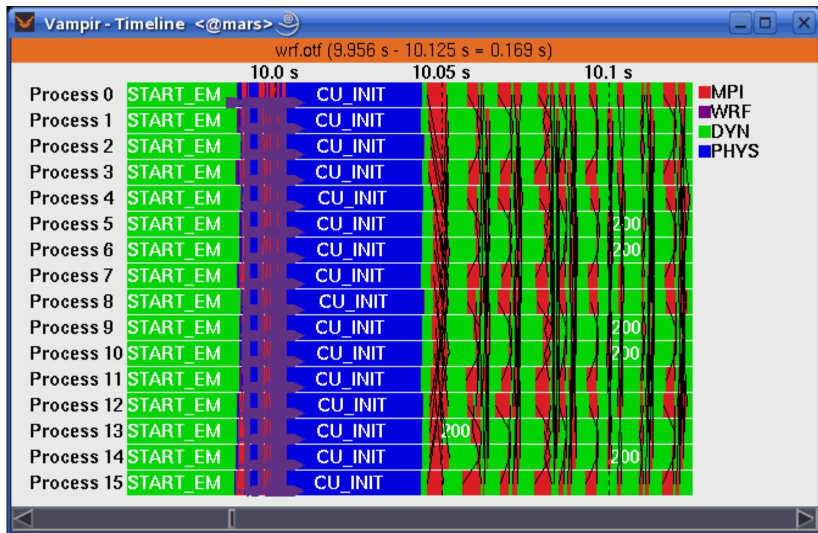
- Affiche des **évènements**
- Supporte le format OpenTrace

Affichages : diagrammes d'états, lignes d'exécution, statistiques...

Outil d'instrumentation générant ces traces :

- TAU
- VampirTrace
- KOJAK
- ...

Vampir : affichage



Profiling des communications

Bibliothèque mpiP

- Utilise l'interface de profiling des fonctions MPI
- Se place entre l'application et la bibliothèque MPI

Génère des statistiques sur les appels de fonctions MPI

- Nombre d'appels par fonction, par endroit dans le code
- Temps passé par appel, par processus
- Volumes de données envoyées

Bon outil de diagnostic pour détecter des déséquilibres

- Exemple : un processus qui attent les autres pour communiquer
- Temps passé dans une fonction de communication démesuré
- Maître-esclave déséquilibré :
  - le maître passe son temps à traiter les données et les résultats
  - les esclaves passent leur temps à attendre des données du maître : grosse proportion du temps passé dans les Recv côté esclaves

Utilisation de mpiP

Compilation : ajout de (au moins) la lib mpiP au link

```
$ mpicc -o mon_prog mon_proc.c -L${mpiP_root}/lib \
                -lmpiP -lm -lbfd -liberty -lunwind
```

Exécution :

- Mode non-invasif : le code n'est pas modifié, tout le programme est instrumenté
- Mode contrôlé : ajout de commandes mpiP dans le code

```
MPI_Pcontrol(1);
```

Argument	Comportement
0	Désactiver le profiling
1	Activer le profiling
2	Remettre à zéro les données
3	Générer un rapport complet
4	Générer un rapport concis

Rapport généré par mpiP

Début : infos sur le programme qui a été exécuté

```
@ mpiP
@ Command : NAS_test/bin/cg.C.16
@ Version                      : 3.1.2
@ MPIP Build date              : Apr 17 2010, 01:53:36
@ Start time                   : 2010 04 17 02:03:22
@ Stop time                    : 2010 04 17 02:04:27
@ Timer Used                   : PMPI_Wtime
@ MPIP env var                 : [null]
@ Collector Rank               : 0
@ Collector PID                : 31805
@ Final Output Dir             : .
@ Report generation            : Single collector task
```

Ensuite : infos sur le mapping des processus sur les machines

```
@ MPI Task Assignment          : 0 borderline-6
@ MPI Task Assignment          : 1 borderline-6
@ MPI Task Assignment          : 2 borderline-6
@ MPI Task Assignment          : 3 borderline-6
[...]
```

Rapport généré par mpiP

Infos générales : temps passé dans les routines MPI processus par processus

```
---------------------------------------------------------------------------
@--- MPI Time (seconds) ---------------------------------------------------
---------------------------------------------------------------------------
Task    AppTime    MPITime     MPI%
   0       65.6       24.3    37.01
   1       65.6       14.2    21.60
   2       65.6       24.6    37.50
   3       65.6       15.8    24.09
   4       65.6       24.7    37.57
   5       65.6       15.3    23.28
   6         65       24.2    37.28
   7       65.6       26.7    40.74
   8         65         17    26.09
   9       65.6         23    35.08
  10       65.6       22.5    34.34
  11       65.6       19.6    29.87
  12       65.6       24.1    36.69
  13       65.6       17.8    27.13
  14       65.6       24.4    37.14
  15       65.6       11.6    17.70
   *   1.05e+03        330    31.44
```

Puis fonction par fonction, processus par processus, appel par appel