Programmation parallèle sur mémoire distribuée

– MASTER MIHPS –

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

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

Plan du cours

Modèle de mémoire
- Mémoire distribuée
- Exemples
- Performance du calcul parallèle

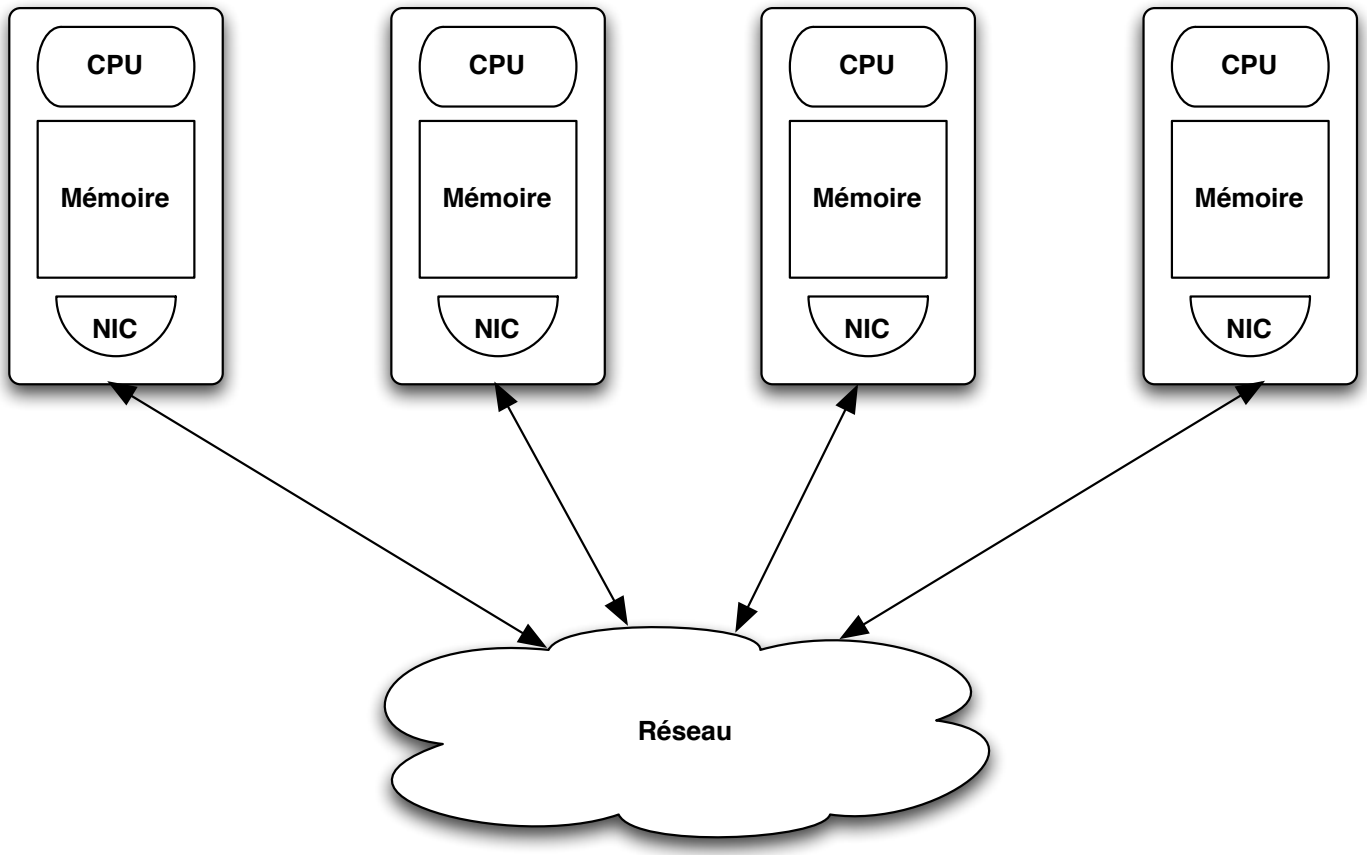
Communications
- Passage de messages
- La norme MPI

Quelques algorithmes
- Communications collectives

Mémoire distribuée

Nœuds de calcul distribués

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



Mise en œuvre

Réseau d'interconnexion

Les nœuds ont accès à un

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

Espace d'adressage

Chaque processus a accès à sa mémoire propre

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

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

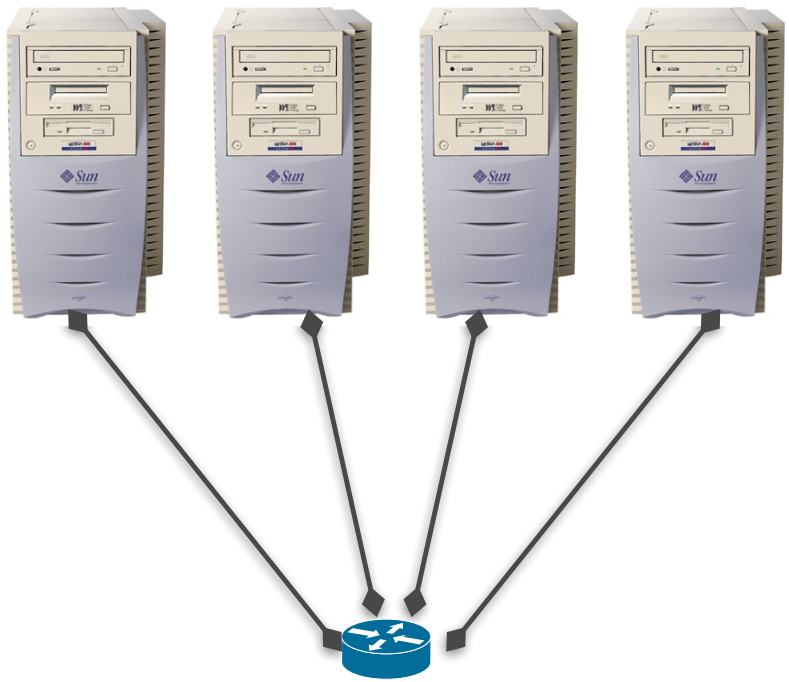
Inconvénients

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

Cluster of workstations

Solution

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



Supercalculateur massivement parallèle (MPP)

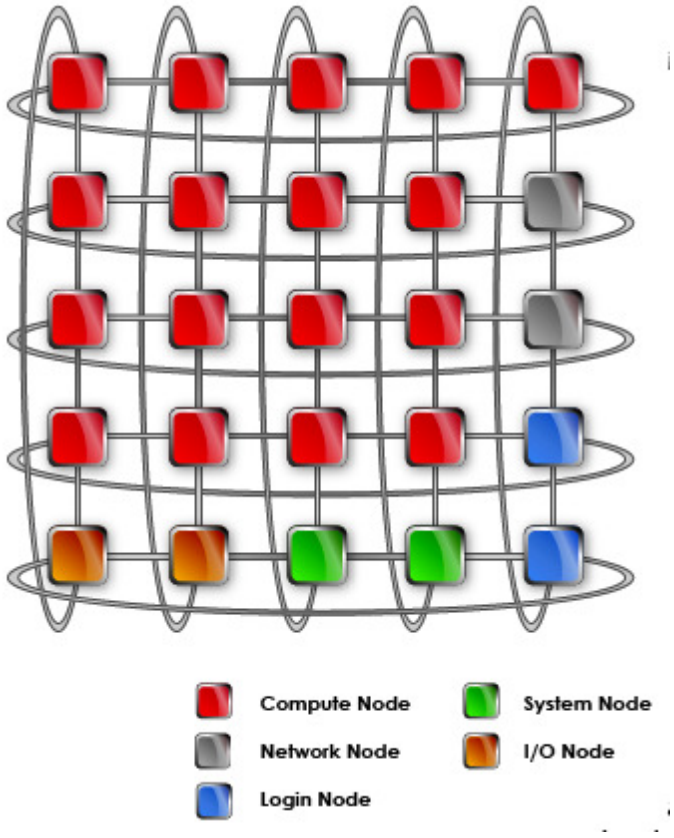
Solution spécifique

- 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

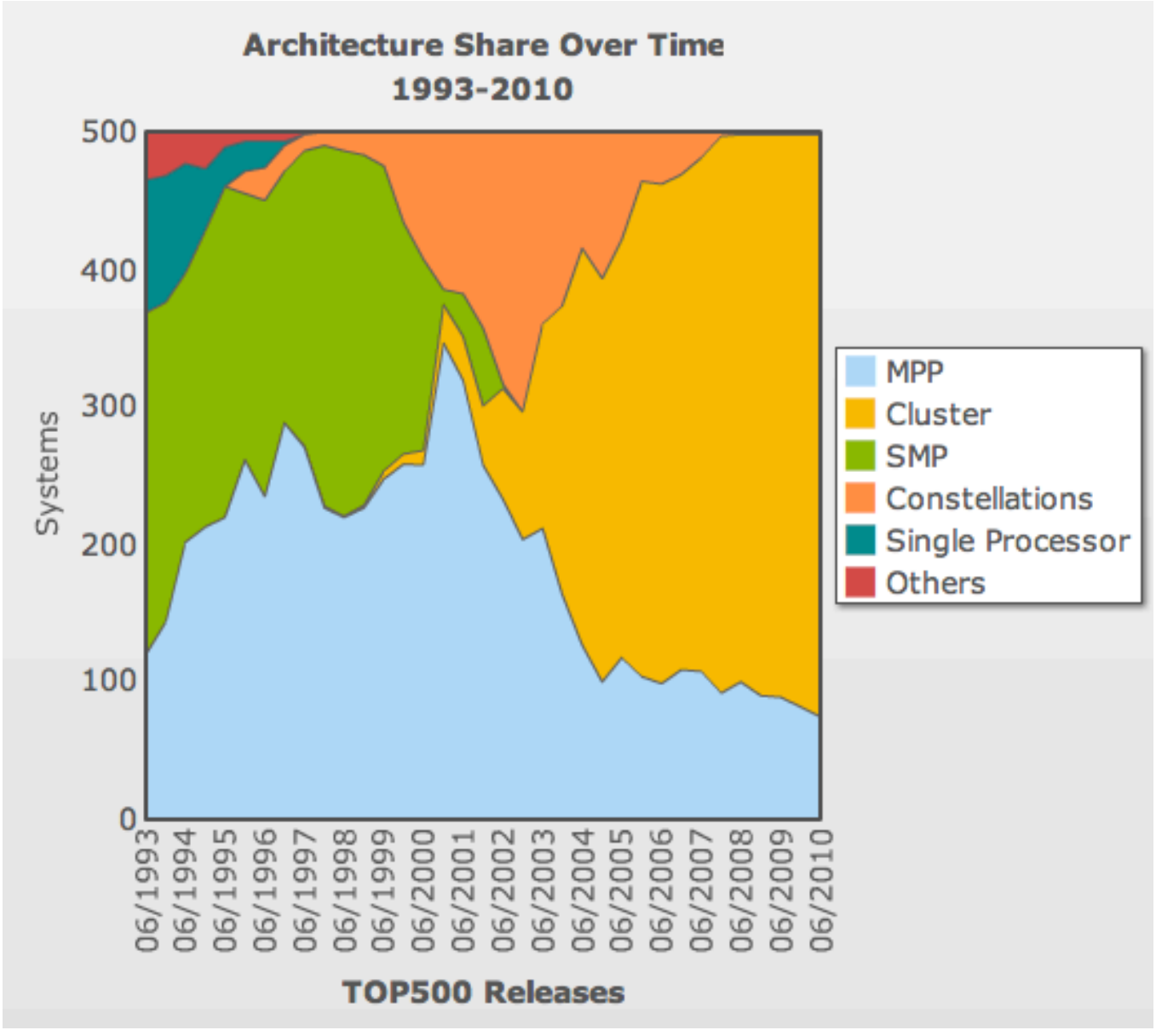
Classement des machines les plus rapides

- 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

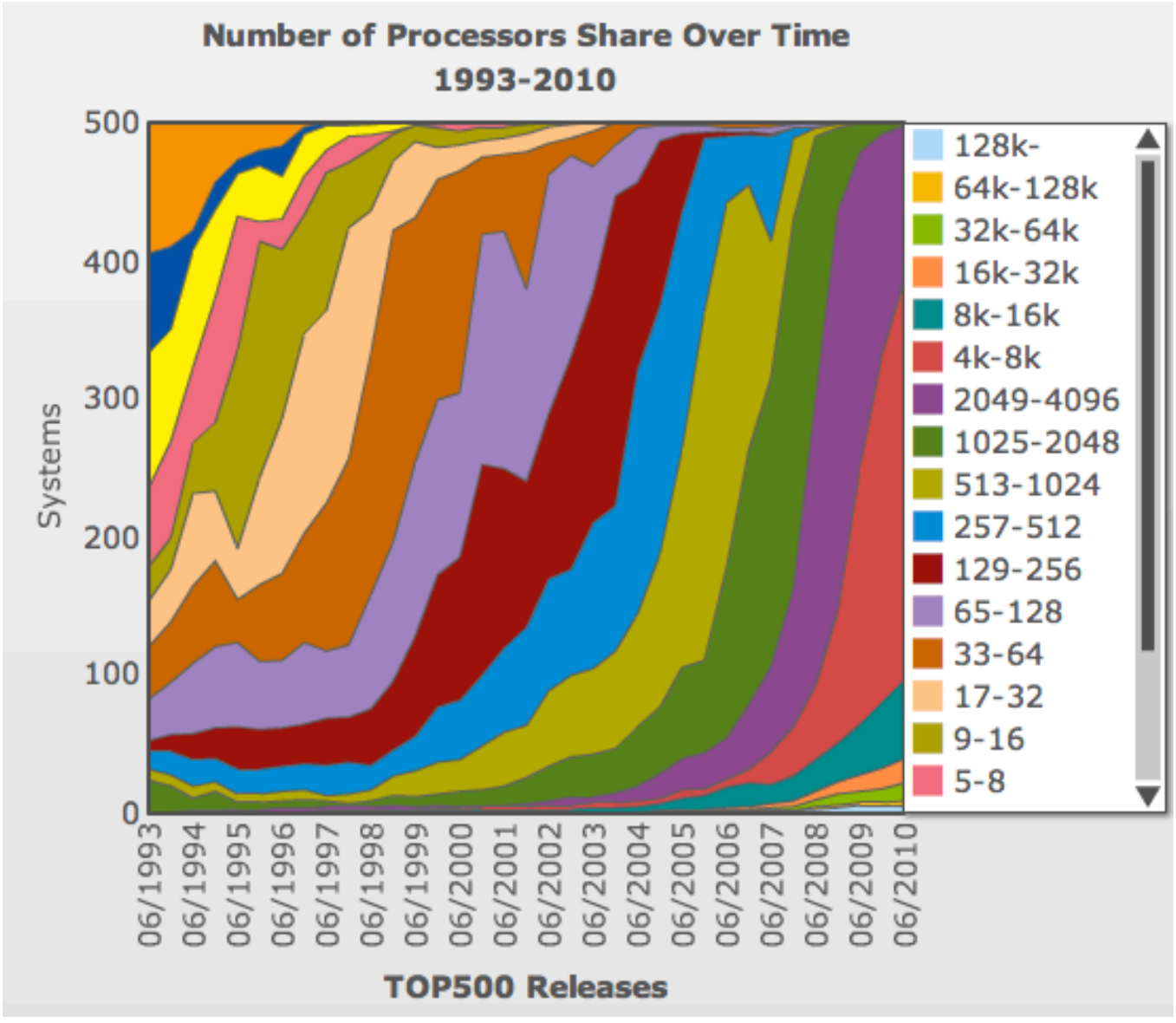
Dernier classement : juin 2010

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



Top 500 - Nombre de CPU



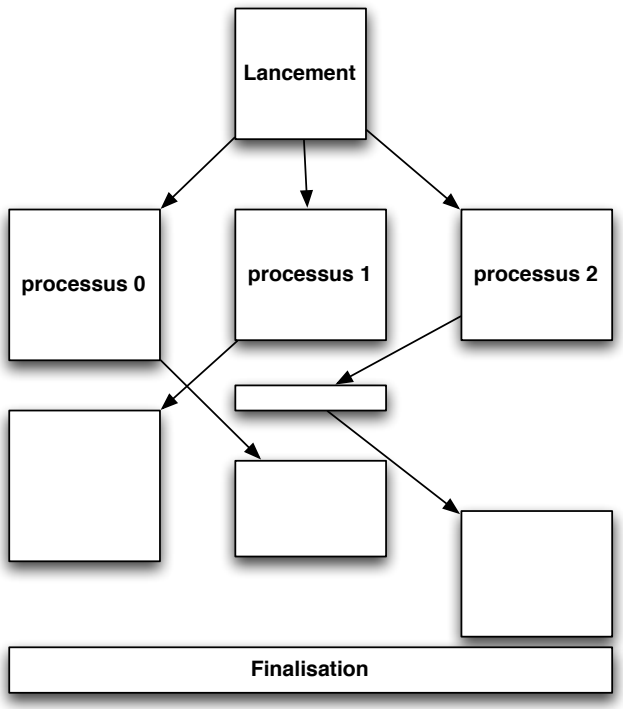
Top 500 - Nombre de CPU juin 2010

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**

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

Accélération

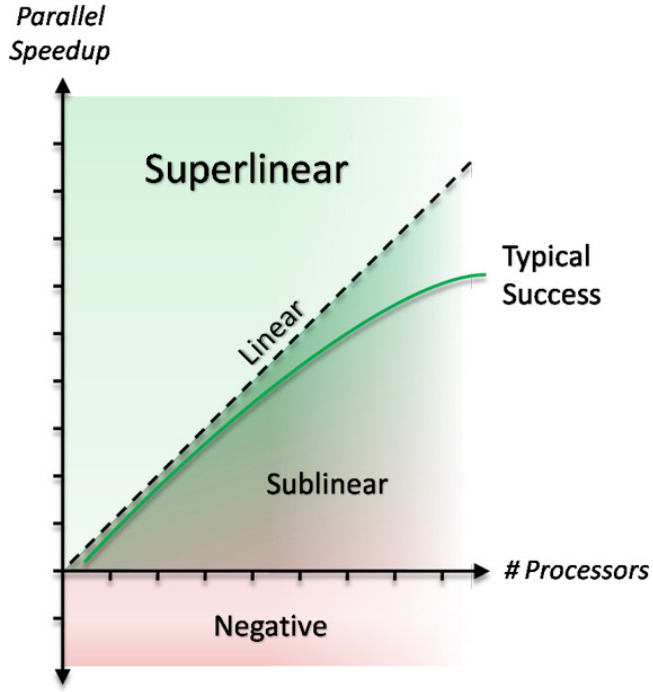
Définition

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

Passage à l'échelle (scalabilité)

Remarque préliminaire

On a vu avec la loi d'Amdahl que la performance augmente 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)

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

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 )`

Nommage des processus

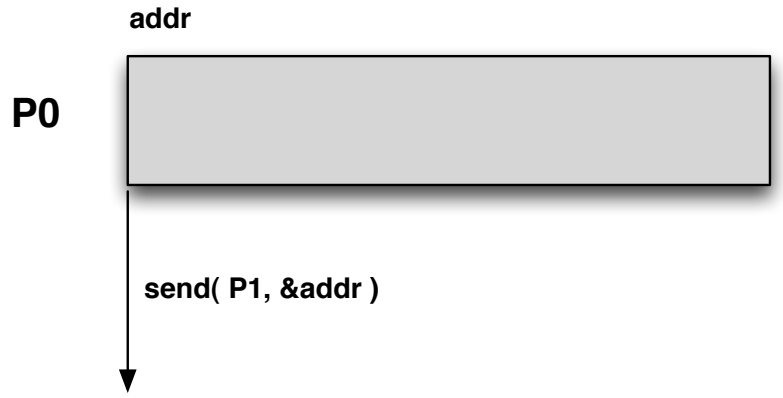
On a besoin d'une façon unique de désigner les processus

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

Tampons des messages

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

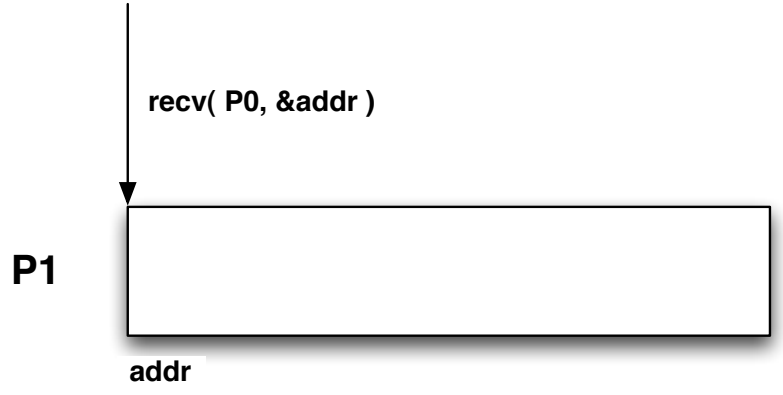
- La mémoire doit être allouée côté émetteur *et côté récepteur*
- On n'envoie pas plus d'éléments que la taille disponible en émission



Linéarisation des données

Les données doivent être sérialisées (marshalling) dans le tampon

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



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 façon 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...

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

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

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

- 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

- 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 être 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.

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 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 )`

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 <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 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 $\rightarrow$ 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)

Communications collectives

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`

Travaux dirigés 1

Utilisation de MPI_Send et MPI_Recv

Écrire une implémentation des opérations collectives en utilisant MPI_Send et MPI_Recv.

Utilisation de MPI_Isend et MPI_Irecv

Écrire une implémentation des opérations collectives en utilisant MPI_Isend et MPI_Irecv. Quelles seront les différences avec la version utilisant MPI_Send et MPI_Recv ?

Combinaison de collectives

Quelles opérations collectives peuvent être exprimées en combinant d'autres opérations collectives ?

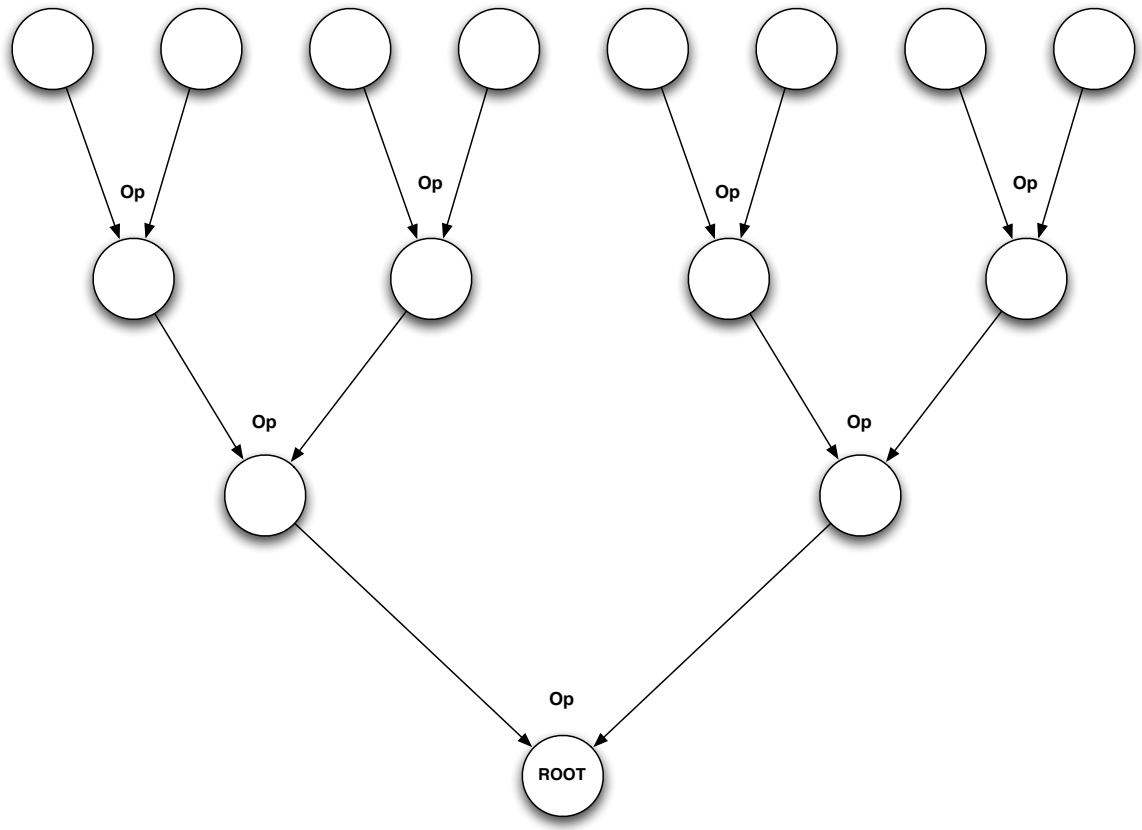
Arbre de réduction

Parallélisme dans la réduction

On cherche à mettre en place du parallélisme

- Dans la communication
- Dans le calcul

$\rightarrow$ réduction deux à deux selon un arbre binaire

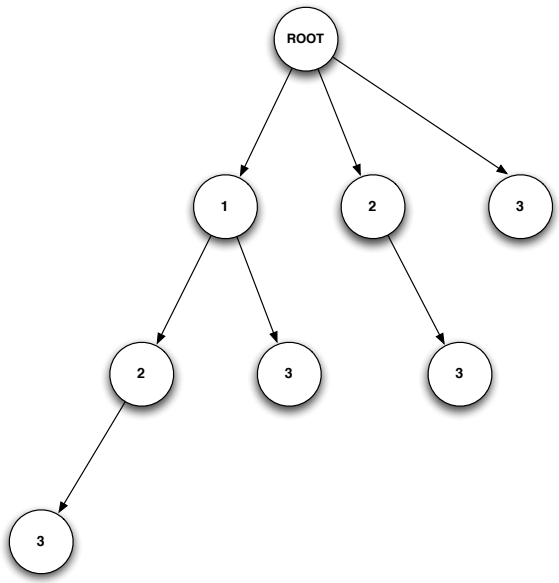


Diffusion en arbre

Principe de l'arbre

- Dès que l'on a reçu la donnée on peut l'envoyer à son tour
- Mise en place d'un parallélisme
- Comment obtenir un parallélisme maximal ?

$\rightarrow$ arbre binomial



Parallélisme dans un arbre binomial

Optimal dans un modèle 1-port

Allgather

Algorithme de Bruck

On concatène deux-à-deux les tampons

- À l'étape k :
  - Send/Recv avec le processus de rang $r + 2^k$
  - Send/Recv avec le processus de rang $r - 2^k$
- nombre d'étapes : $O(log_2 P)$ si on a P processus