Systèmes Parallèles et Distribués

2. Introduction à la programmation de machines parallèles distribuées avec MPI

Franck Butelle Camille Coti

LIPN, Université Paris 13
Formation Ingénieurs SupGalilée Info 3

01/2014

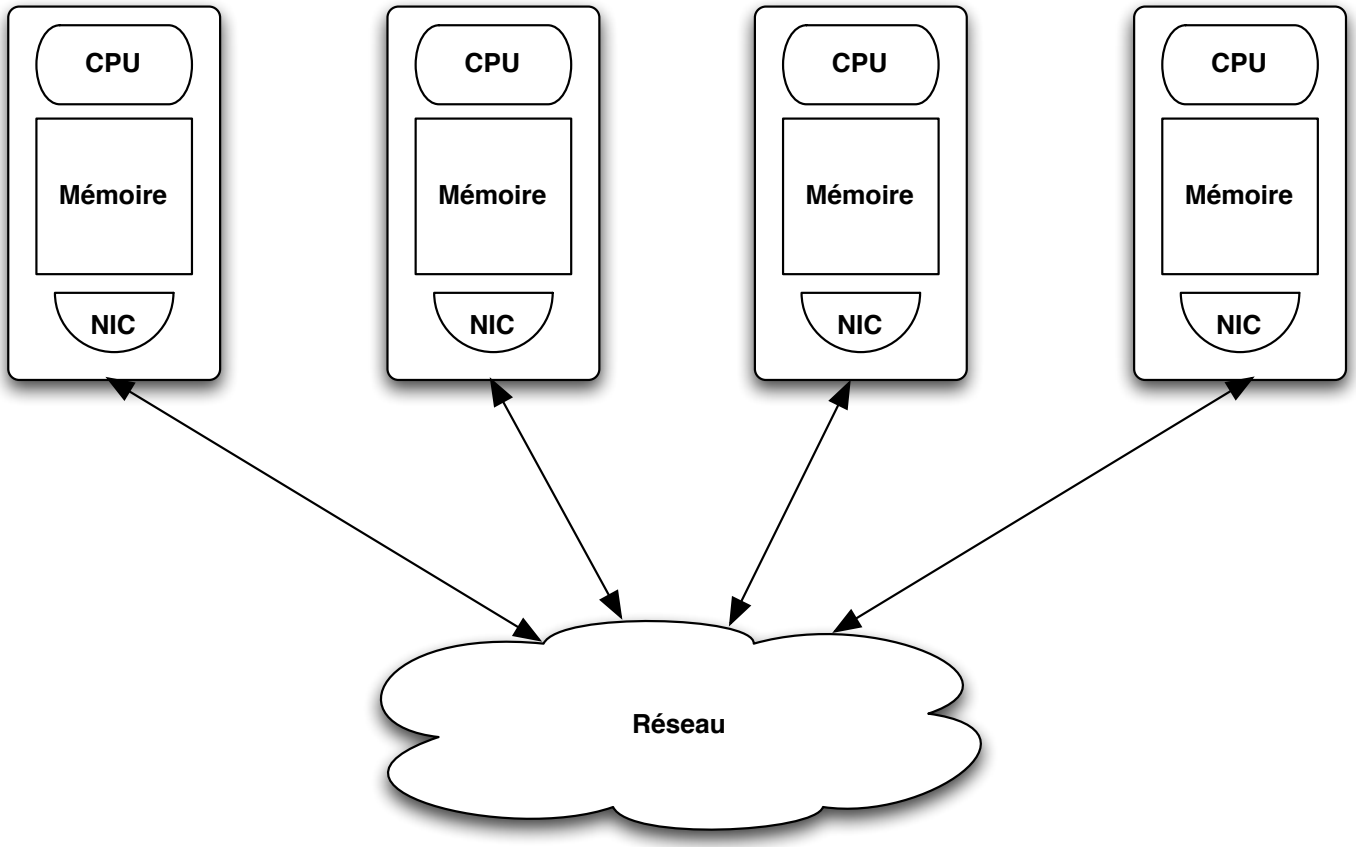
Plan

Plan

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

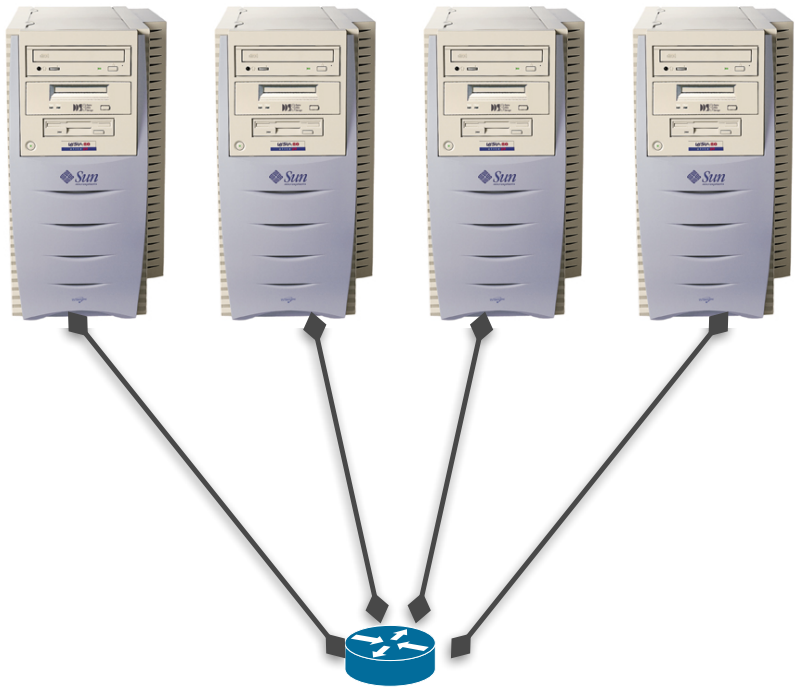
- Plus complexe à programmer
  - Intervention du programmeur dans le parallélisme
- Temps d'accès aux données distantes

Exemples d'architectures

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

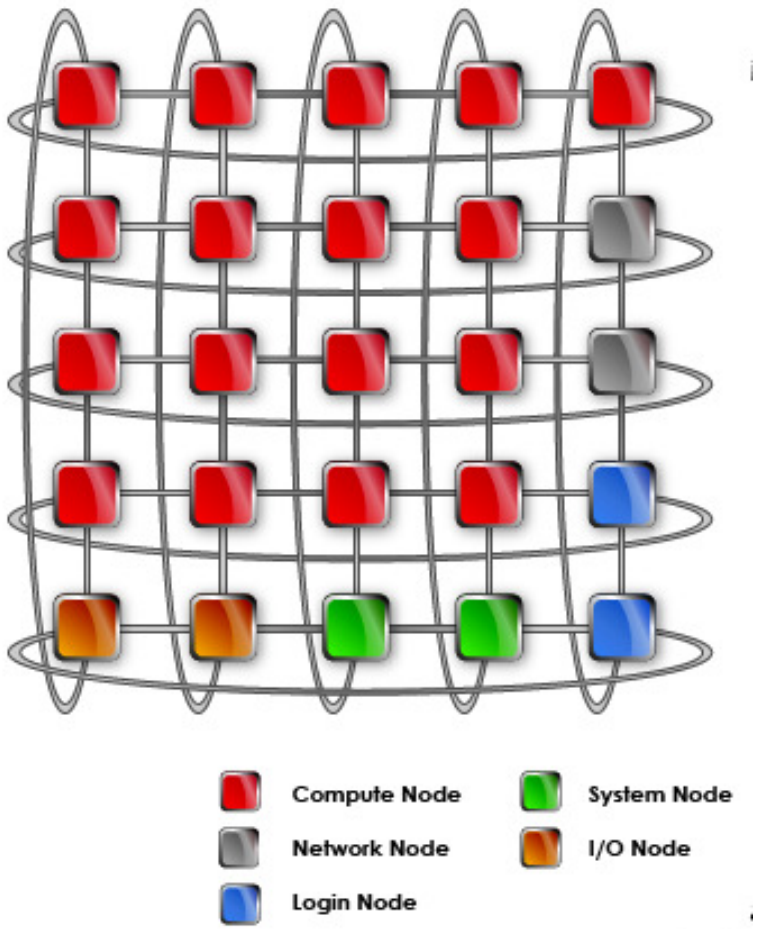
- CPU : deux AMD Istanbul
  - 6 cœurs chacun
  - 2 puces par machine
  - Empilées sur la même socket
  - Bus : crossbar
- Pas de disque dur

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 : novembre 2014

1. Tianhe-2 (MilkyWay-2) - NUDT - National Super Computer Center in Guangzhou
2. Titan - Cray XK7 - Oak Ridge National Lab
3. Sequoia - IBM BlueGene/Q - Lawrence Livermore National Lab
4. K computer - Fujitsu - RIKEN, Japon
5. Mira - IBM BlueGene/Q - Argonne National Lab

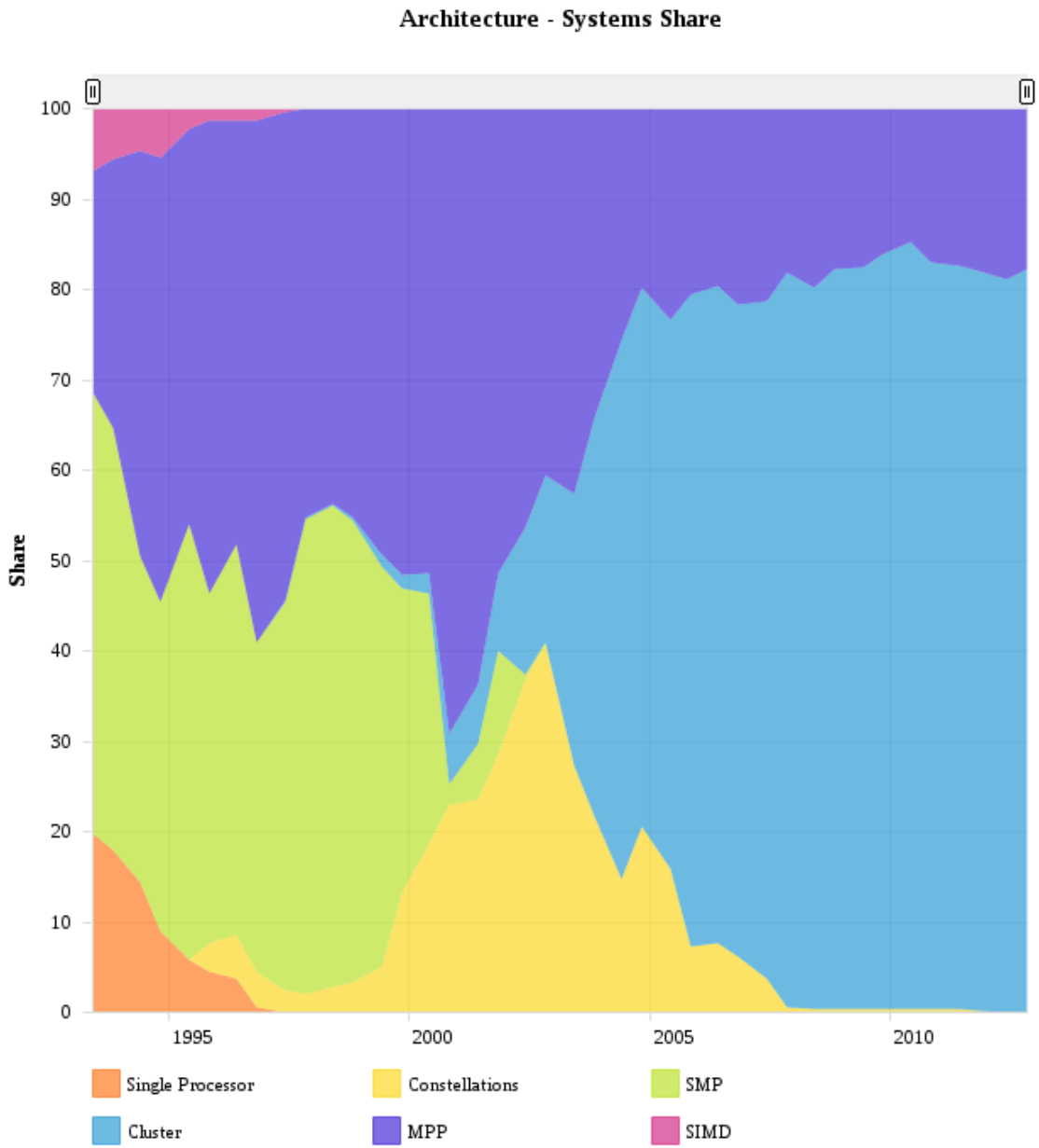
Top 500 - Nombre de coeurs

Rang	Machine	Nb de coeurs	Rmax (TFlops)	Conso (kW)
1	Tianhe-2	3 120 000	33 862.7	17 808.00
2	Titan	560 640	17 590,0	8 209
3	Sequoia	1 572 864	16 324,8	7 890
4	K Computer	705 024	10 510,0	12 660
5	Mira	786 432	8 162,4	3 945

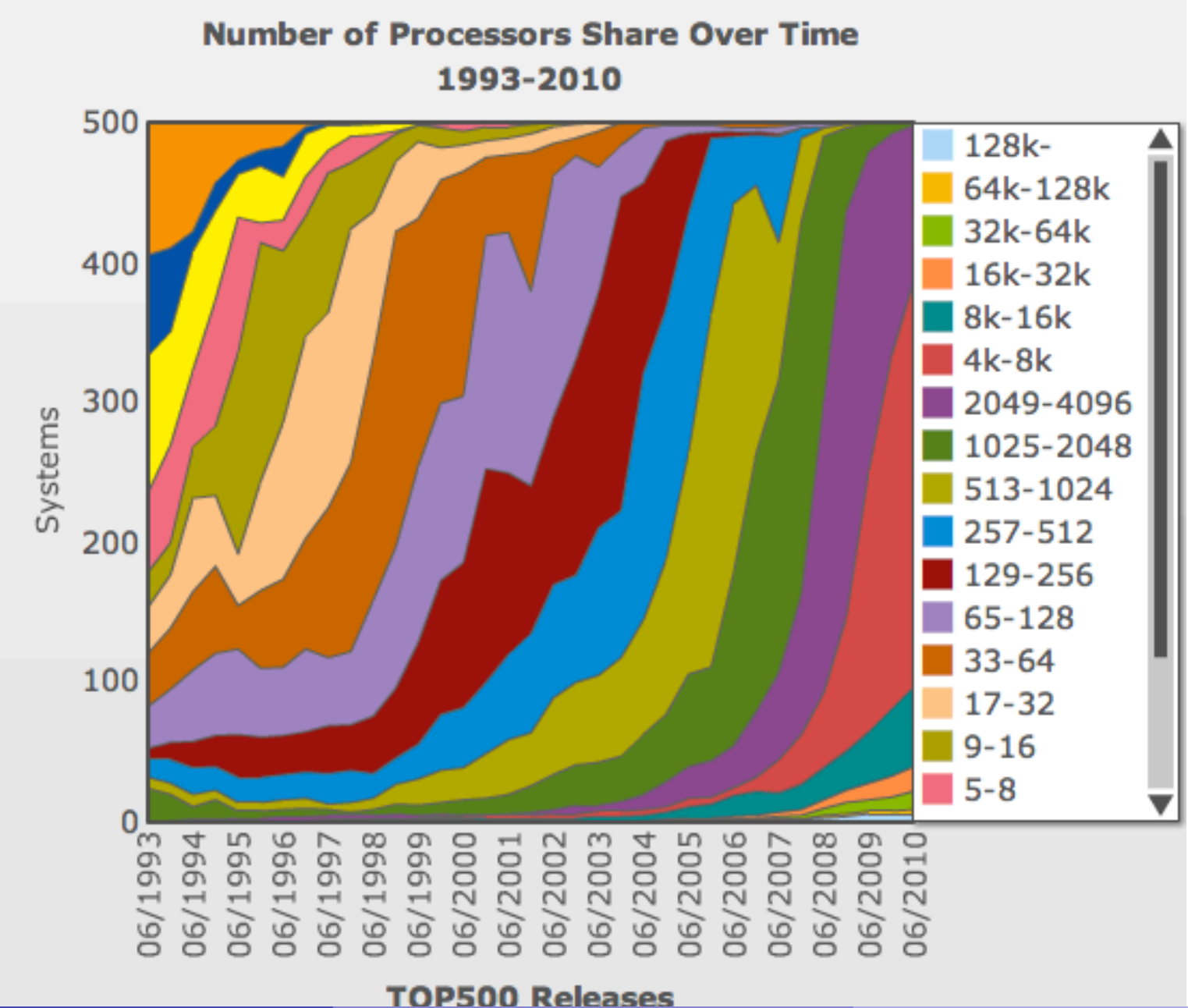
Anciens et actuels numéro 1 :

- Tianhe-2 : depuis juin 2013
- Titan : numéro 1 en novembre 2012
- Sequoia : numéro 1 en juin 2012
- K : numéro 1 de juin à novembre 2011

Top 500 - Type de systèmes



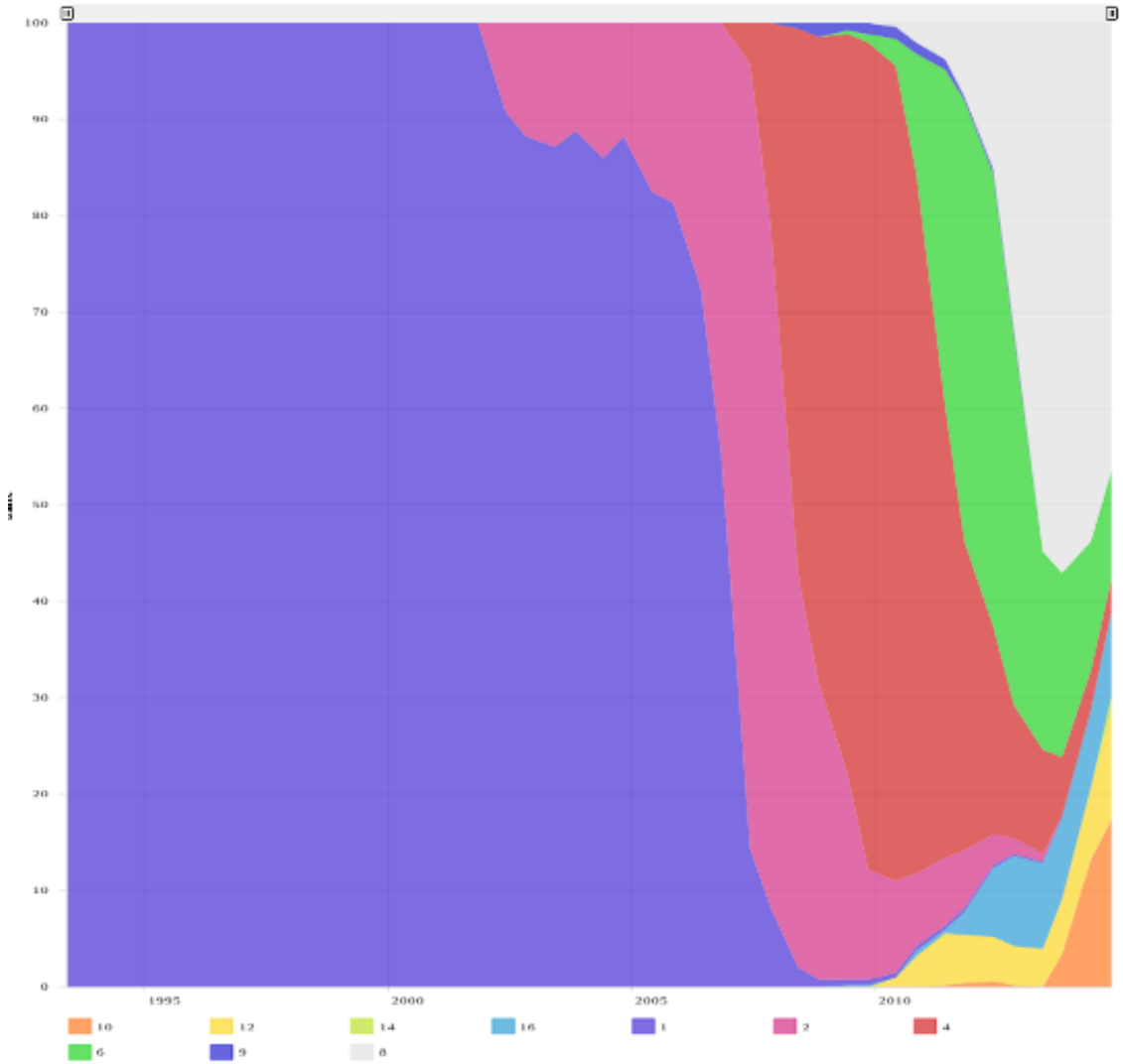
Top 500 - Nombre de CPU



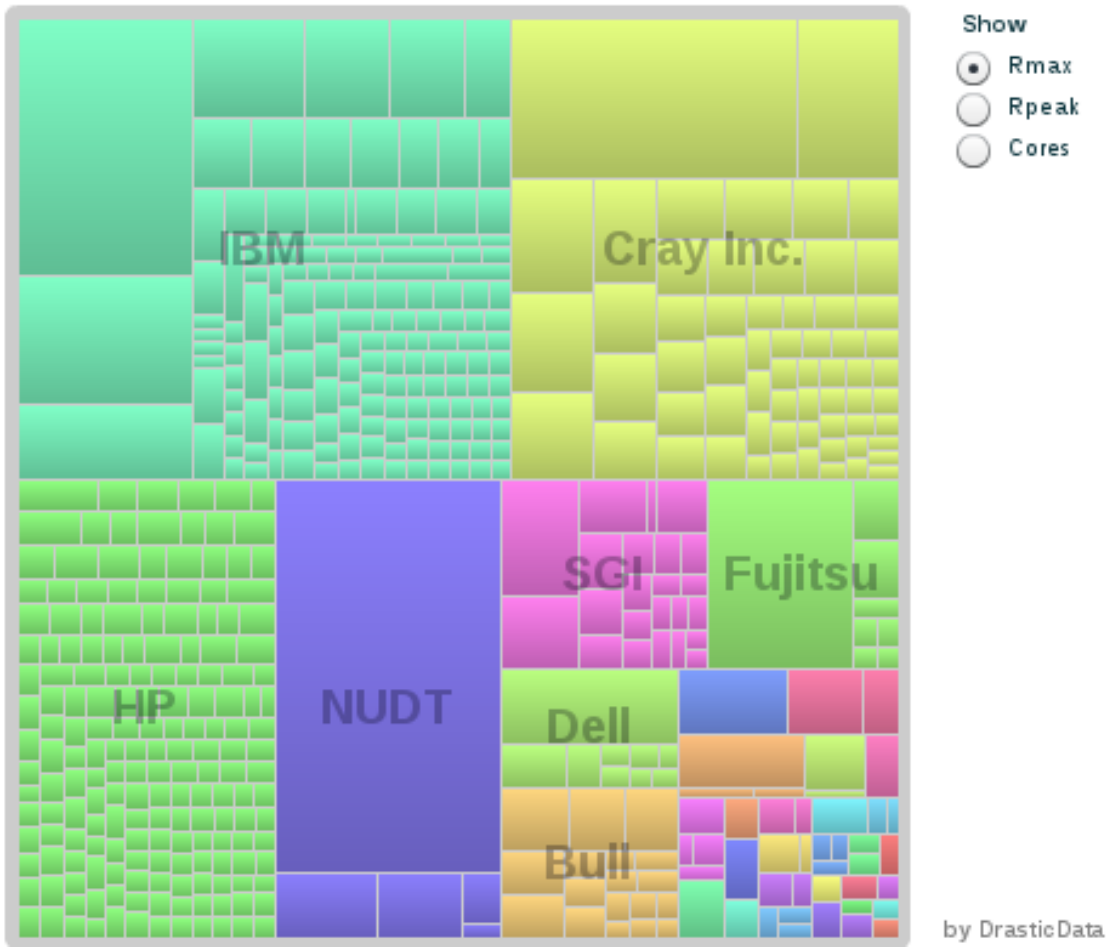
Top 500 - Nombre de CPU juin 2010

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**

Top 500 - nombre de coeurs par socket



Top 500 - Principaux constructeurs (novembre 2014)



Top 500 - Les numéro 1

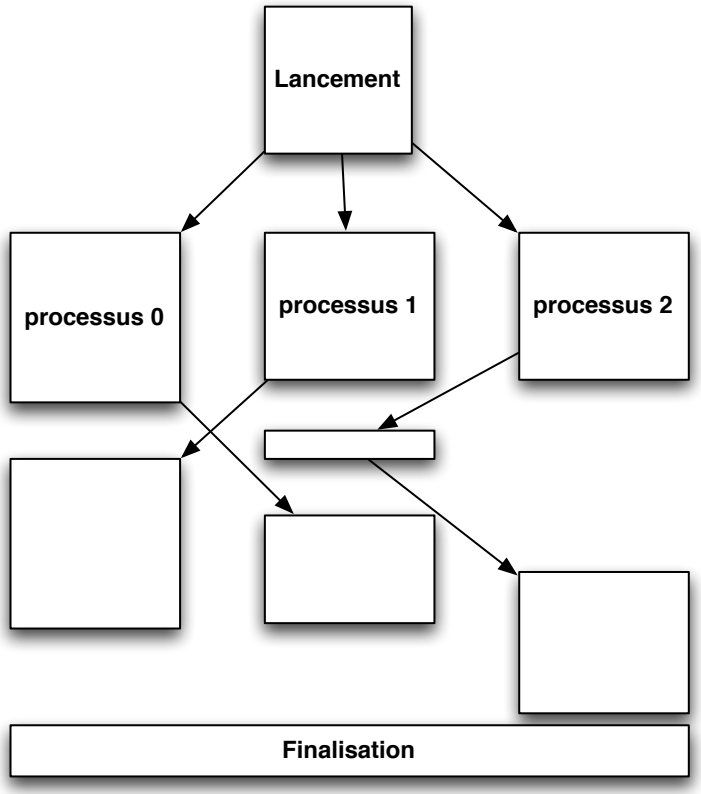
Nom	Dates #1	Nb coeurs	Rmax
CM-5	06/93	1 024	59,7 Gflops
Numerical Wind Tunnel	11/93	140	124,2 Gflops
Intel XP/S 140 Paragon	04/93	3 680	143,40 Gflops
Numerical Wind Tunnel	11/94-12/95	167	170,0 Gflops
Hitachi SR2201	06/96	1 024	232,4 Gflops
CP-PACS	11/96	2 048	368,20 Gflops
ASCI Red	06/97 - 06/00	7 264	1,068 Tflops
ASCI White	11/00 - 11-01	8 192	4,9 - 7,2 Tflops
Earth Simulator	06/02 - 06/04	5 120	35,86 Tflops
BlueGene/L	11/04 - 11/07	212 992	478,2 Tflops
Roadrunner	06/08 - 06/09	129 600	1.026 - 1,105 Pflops
Jaguar	11/09 - 06/10	224 162	1,759 Pflops
Tianhe-1A	11/10	14 336 + 7 168	2.57 Pflops
K	06/11 - 11/11	548 352 - 705 024	8,16 - 10,51 Pflops
Sequoia	06/12	1 572 864	16,32 Pflops
Titan	11/12	552 960	17,6 Pflops
Tianhe-2	6/13 →	3 120 000	33,9 Pflops

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 ?

Accélération

Définition

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

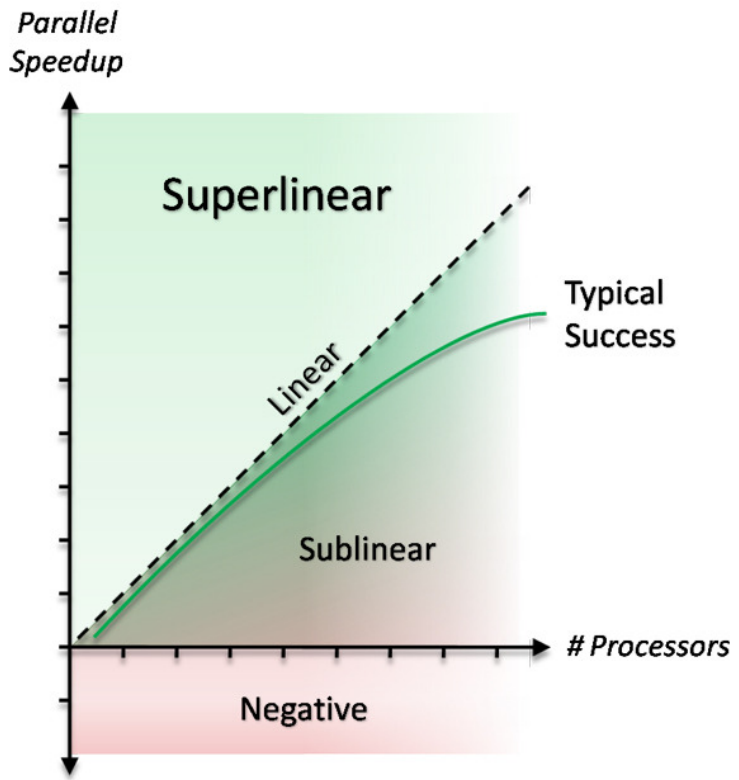
Mesure de l'accélération

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

Énoncé

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 *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

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

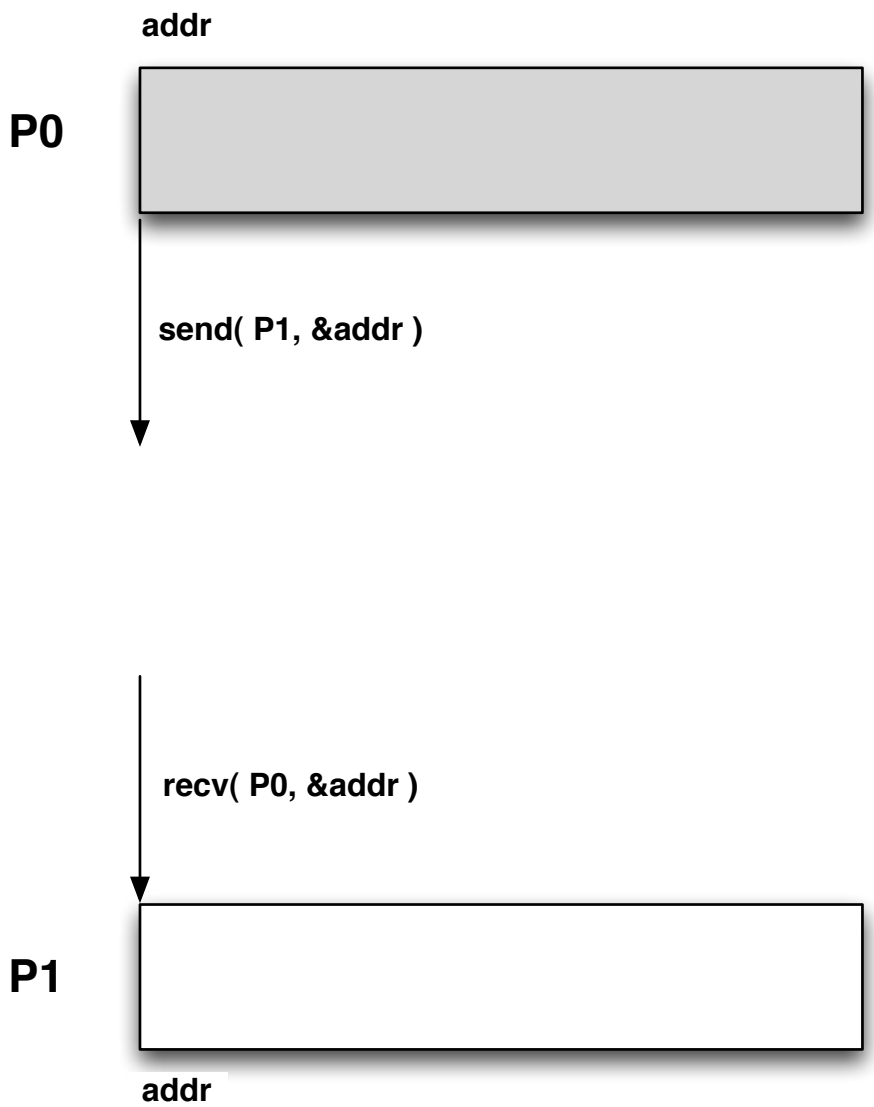
Chaque processus (émetteur et récepteur) a un tampon (buffer) pour le message

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

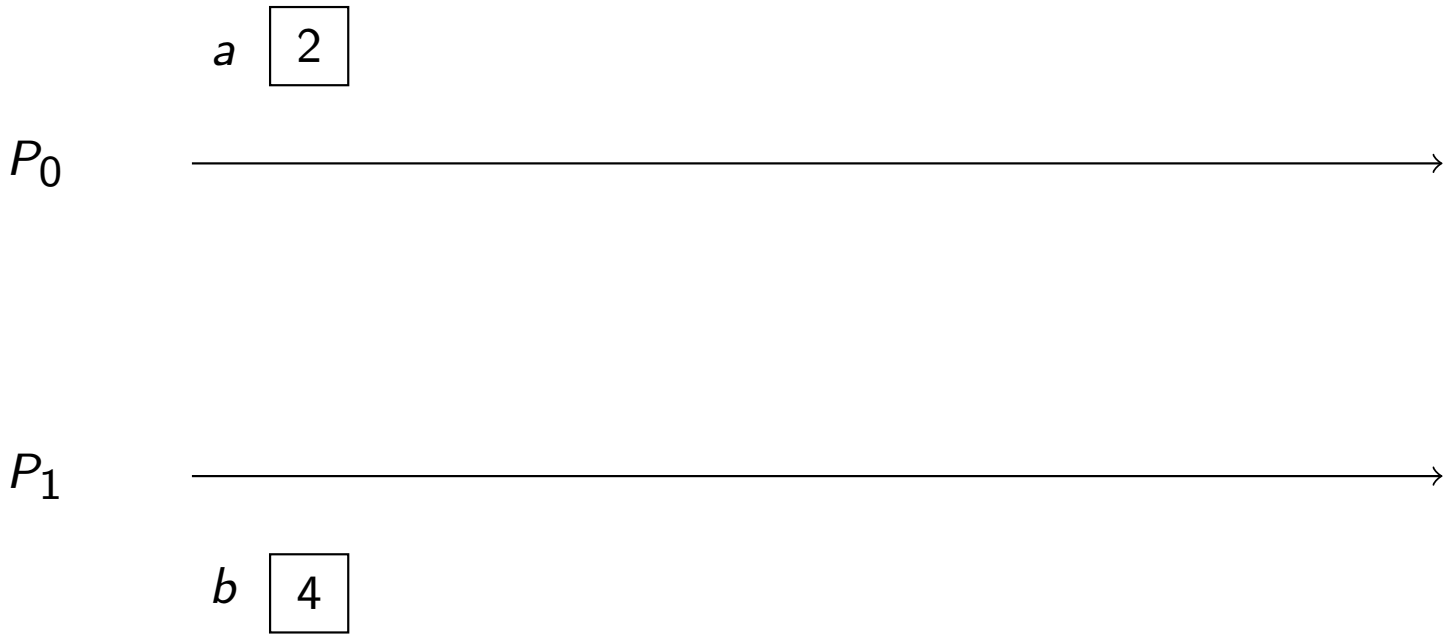


Communications bilatérales

Communications bilatérales

- Primitives *send*/*recv*
- Une primitive *send doit* matcher une primitive *recv* (et inversement)

Conséquence : quand on déplace des données entre deux processus, les deux processus participent *activement* au déplacement des données

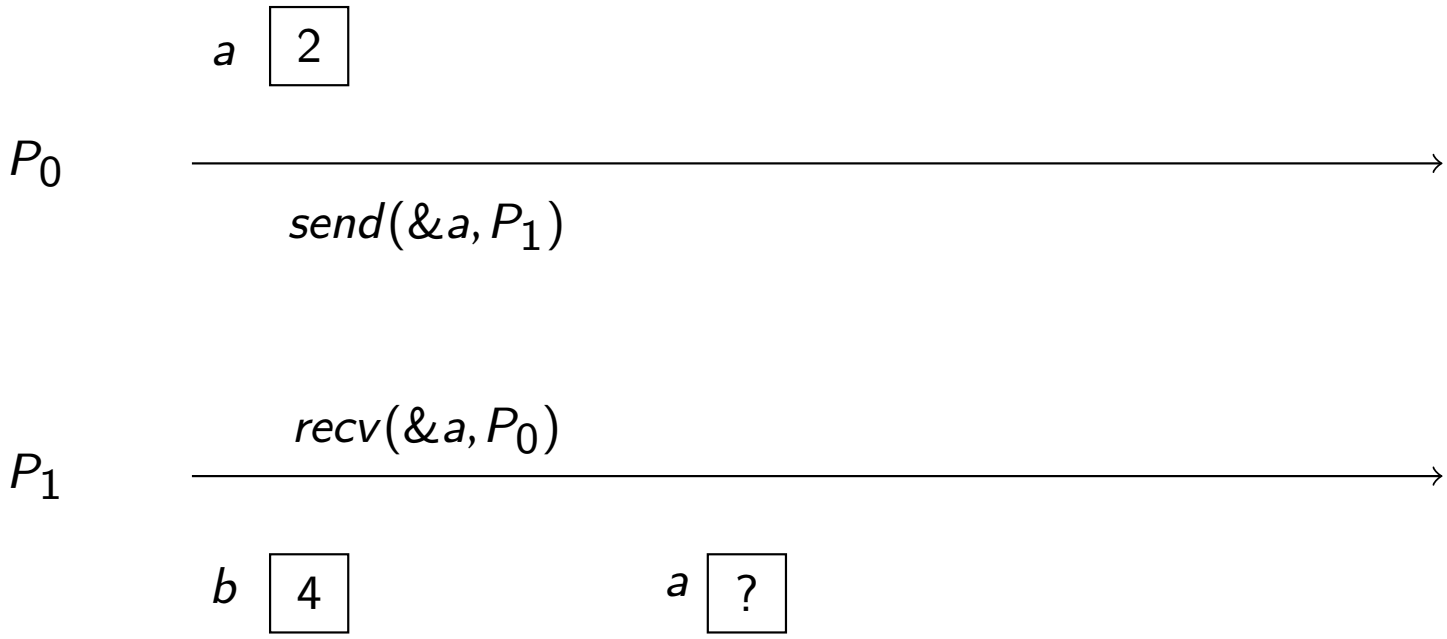


Communications bilatérales

Communications bilatérales

- Primitives *send* / *recv*
- Une primitive *send doit* matcher une primitive *recv* (et inversement)

Conséquence : quand on déplace des données entre deux processus, les deux processus participent *activement* au déplacement des données

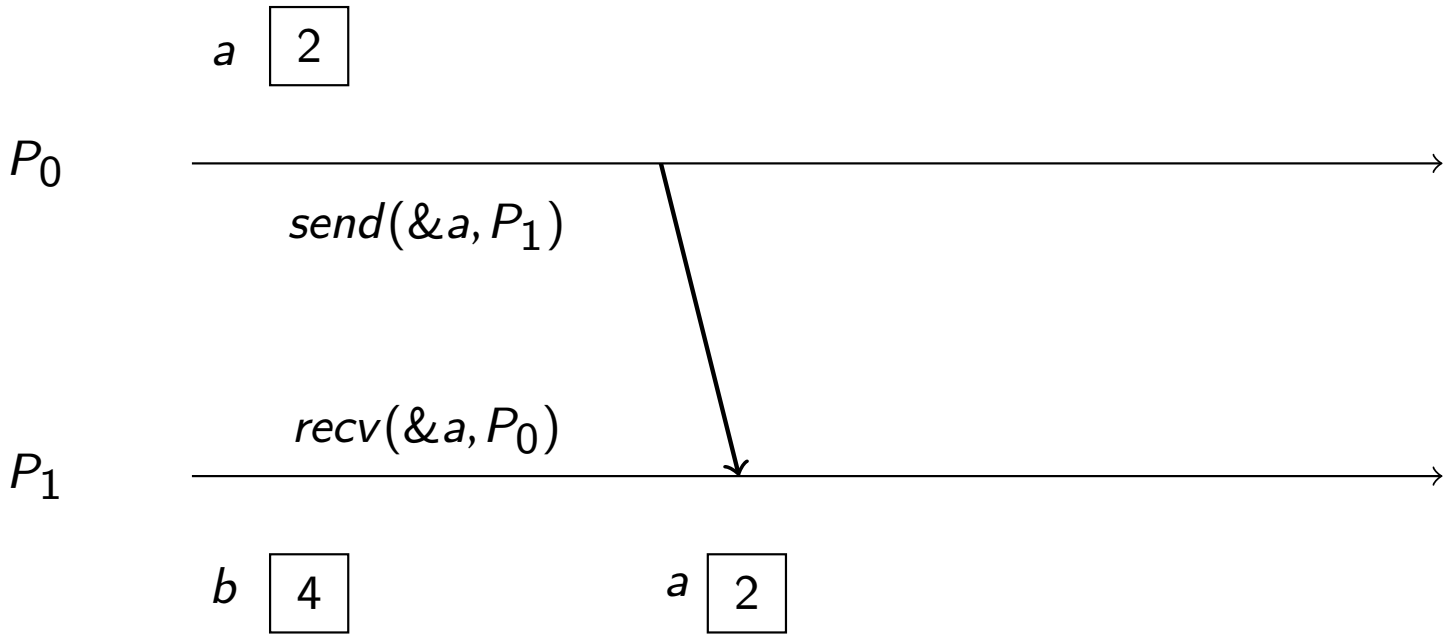


Communications bilatérales

Communications bilatérales

- Primitives *send*/*recv*
- Une primitive *send doit* matcher une primitive *recv* (et inversement)

Conséquence : quand on déplace des données entre deux processus, les deux processus participent *activement* au déplacement des données

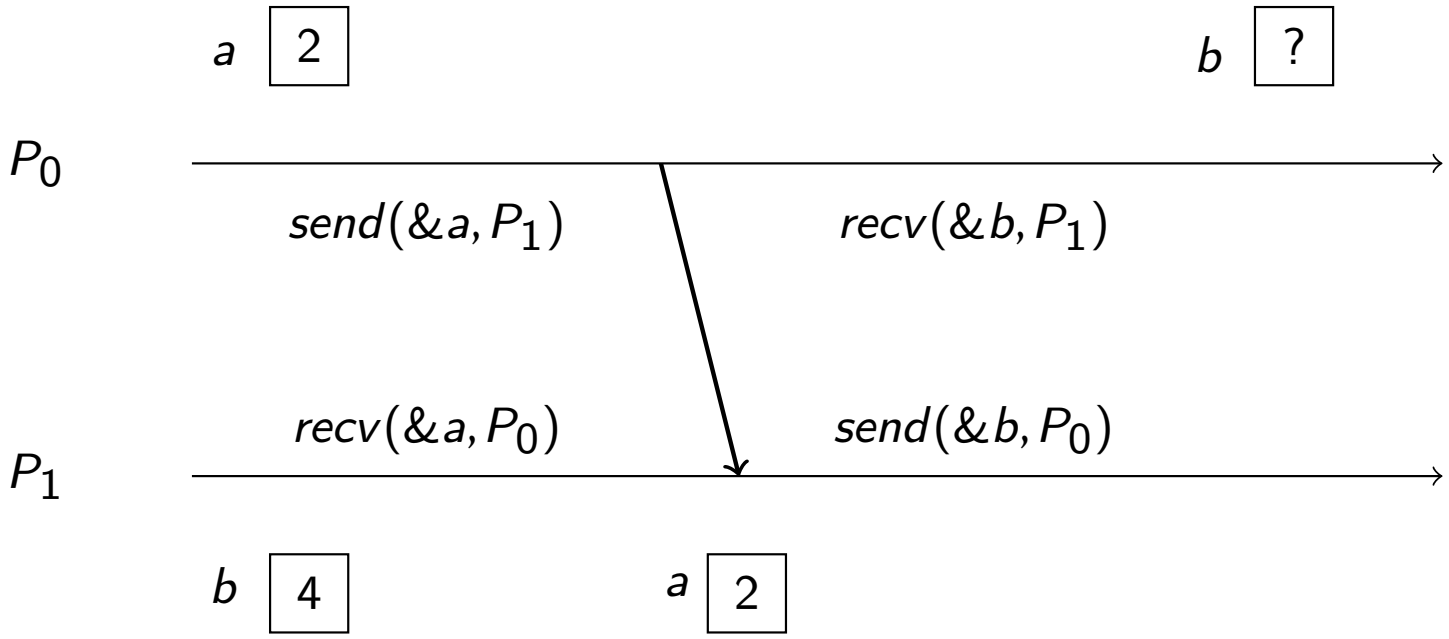


Communications bilatérales

Communications bilatérales

- Primitives *send*/*recv*
- Une primitive *send doit* matcher une primitive *recv* (et inversement)

Conséquence : quand on déplace des données entre deux processus, les deux processus participent *activement* au déplacement des données

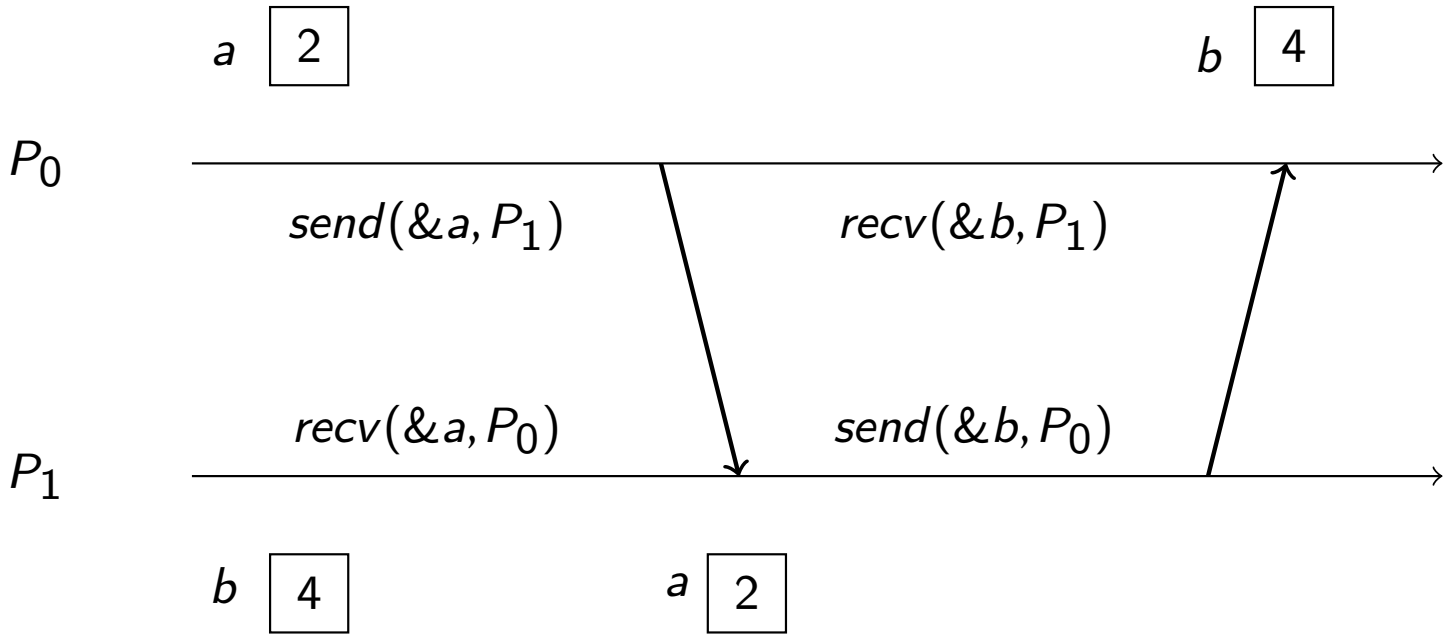


Communications bilatérales

Communications bilatérales

- Primitives *send*/*recv*
- Une primitive *send doit* matcher une primitive *recv* (et inversement)

Conséquence : quand on déplace des données entre deux processus, les deux processus participent *activement* au déplacement des données



Modèle de communications

Asynchrones

- Délais de communications finis, non-bornés

Modes de communications

- Petits messages : *eager*
  - L'émetteur envoie le message sur le réseau et retourne dès qu'il a fini
  - Si le destinataire n'est pas dans une réception, le message est bufferisé
  - Quand le destinataire entre dans une réception, il commence par regarder dans ses buffers si il n'a pas déjà reçu
- Gros messages : *rendez-vous*
  - L'émetteur et le destinataire doivent être dans une communication
  - Mécanisme de rendez-vous :
    - Envoi d'un petit message
    - Le destinataire acquitte
    - Envoi du reste du message
  - L'émetteur ne retourne que si il a tout envoyé, donc que le destinataire est là : pas de mise en buffer

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 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...

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)
- Exemples : InfiniBand, OpenSHMEM

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 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.

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

```c
#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 être appelé simplement pour entrer dans la bibliothèque MPI (lui redonner la main pour faire avancer des opérations)