CHPS0711
Programmation parallèle

Cours 2

Conception d'algorithmes parallèles
Modèle à passage de messages
Programmation MPI

Pierre Delisle
Université de Reims Champagne-Ardenne
Département de Math. Méca. et Informatique
Octobre 2018

Plan de la séance

- Le modèle Tâche/Canal (task/channel)
- Le modèle à passage de messages
- Programmation MPI

Le modèle tâche/canal

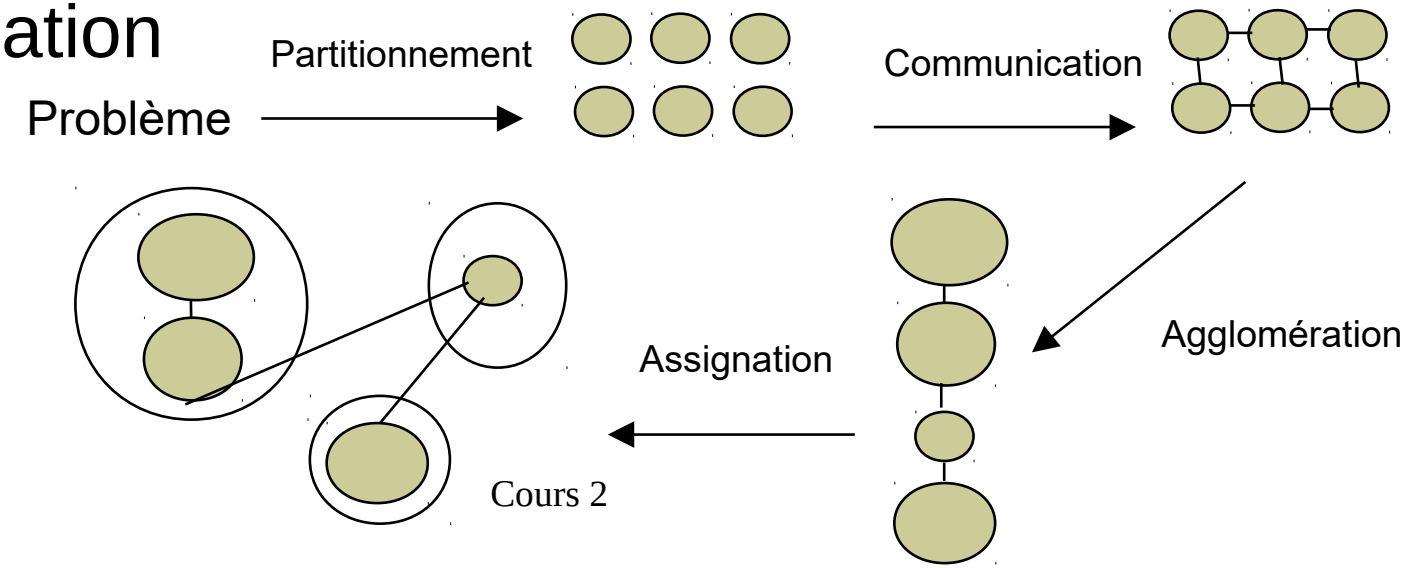
Présentation
Méthodologie de conception
Exemple

Modèle Tâche/Canal (Task/Channel)

- Développé par Foster en 1995
- Exécution parallèle : ensemble de tâches qui interagissent les unes avec les autres par des messages envoyés à travers des canaux
- Tâche : programme + mémoire + ports d'E/S
- Canal : file connectant le port de sortie d'une tâche avec le port d'entrée d'une autre tâche
- Réception de message -> bloquant
- Envoi de message -> non bloquant

Méthodologie de conception

- 4 étapes dans la conception d'un algorithme parallèle :
  - Partitionnement
  - Communication
  - Agglomération
  - Assignation



Partitionnement

- Division des opérations et des données en morceaux
- Peut être centré sur les données ou sur les instructions
- Décomposition de domaine
  - Division des données
  - Association des instructions aux données

Partitionnement

- Décomposition fonctionnelle
  - Division des instructions
  - Association des données aux instructions
- But : Identifier un max. de tâches primitives

4 critères pour évaluer la qualité d'un partitionnement

- Le nombre de tâches primitives est considérablement plus grand que le nombre de processeurs de l'ordinateur utilisé
- Les instructions et données redondantes sont minimisées (extensibilité -> taille de problème)
- Les tâches primitives sont de taille semblable (efficacité de l'ordonnancement des tâches)
- Le nombre de tâches primitives augmente en fonction de la taille du problème (extensibilité -> nombre de processeurs)

Communication

- 2 formes de communication entre les tâches
- Locale
  - Une tâche nécessite des données provenant d'un petit nombre d'autres tâches
  - Création de canaux partant des tâches fournissant les données, vers la tâche qui a besoin de ces données
- Globale
  - Une tâche nécessite des données provenant d'un grand nombre d'autres tâches
  - Gestion des canaux plus complexe et généralement traitée plus tard dans le processus de conception

Communication

- Les communications entre tâches d'un algorithme parallèle sont une source de coûts et doivent être minimisées autant que possible
- 4 critères permettent d'évaluer la structure des communications d'un algorithme parallèle
  - Communications équilibrées entre les tâches
  - Chaque tâche communique avec un petit nombre de tâches voisines
  - Les tâches peuvent communiquer de façon concurrente
  - Les tâches peuvent faire leurs calculs de façon concurrente

Agglomération

- À partir de cette étape, une architecture cible doit être identifiée
- Regroupement de tâches primitives en tâches de plus grande taille
  - Amélioration de la performance
  - Simplification de la programmation
- L'agglomération de tâches primitives communicantes :
  - Réduit la charge de communication
  - Augmente la localité de l'algorithme parallèle

7 critères pour évaluer la qualité de l'agglomération

- Augmentation de la localité
- La réplication de calcul prend moins de temps que les communications qu'elle remplacent
- La quantité de données répliquée est assez petite pour permettre l'extensibilité
- Les tâches agglomérées ont des coûts de calcul et de communication similaires
- Le nombre de tâches augmente avec la taille de problème
- Le nombre de tâches est aussi petit que possible, tout en étant plus grand que le nombre de processeurs
- L'agglomération choisie pour la parallélisation d'un code séquentiel implique des coûts raisonnables de modification

Assignation (mapping)

- Attribution des tâches aux processeurs
- Buts :
  - Maximiser l'utilisation des processeurs (% moyen du temps où les processeurs sont actifs durant l'exécution)
  - Minimiser les communications entre processeurs
- Ces 2 buts sont souvent contradictoires, un compromis est alors nécessaire

4 règles pour évaluer la qualité de l’assignation

- Considération d’une seule tâche par proc et de plusieurs tâches par proc
- Considération des allocations statiques et dynamiques des tâches aux procs
- Si une allocation dynamique a été choisie, vérification qu’elle n’est pas un goulot d’étranglement au niveau du système
- Si une allocation statique a été choisie, vérification que le ratio tâche/proc est sufissament grand

Exemple : problème d'addition de n nombres (réduction)

- Soit un ensemble de nombres a_0, a_1, a_2, … , a_{n-1}
- Trouver la somme $a_0 + a_1 + a_2 + \ldots + a_{n-1}$
- En séquentiel : n-1 opérations -> Θ(n)

- Partitionnement :
  - n valeurs -> n tâches avec une valeur chacune
  - But : trouver la somme des n valeurs

Exemple : problème d'addition de n nombres (réduction)

- Communication :
  - Des canaux doivent être créés entre les tâches pour transférer les données à additionner
  - Une tâche doit, à la fin, contenir le grand total (tâche racine)

Exemple : problème d'addition de n nombres (réduction)

- Premier essai : toutes les tâches envoient leur valeur à la tâche racine qui effectue l'addition
  - Temps d'exécution : $(n - 1)(\lambda + \chi)$
  - Plus lent que l'algorithme séquentiel !
- Conclusion : un meilleur équilibre doit être trouvé entre les calculs et les communications

λ (Lambda) = Temps de communication d'une tâche à une autre

χ (Khi) = Temps de calcul d'une tâche

Exemple : problème d'addition de n nombres (réduction)

- Deuxième essai : 2 semi-racines, chacune responsable de n/2 éléments
  - 2 communications en parallèle
  - 2 calculs d'addition en parallèle
  - Temps d'exécution : (n/2) (λ + χ)
  - Réduction de moitié !

Exemple : problème d'addition de n nombres (réduction)

- Troisième essai : 4 semi-racines, chacune responsable de n/4 éléments
  - Temps d'exécution : (n/4 + 1) (λ + χ)
  - Presque 4 fois plus rapide !

Exemple : problème d'addition de n nombres (réduction)

- En utilisant un schéma de communication en arbre binomial :
  - Addition de 2 valeurs = 1 passage de message
  - Addition de 4 valeurs = 2 passages de message
  - Addition de 8 valeurs = 3 passages de message
  - Addition de n valeurs = log n passages de message
- Exemple de 16 tâches
- Nombre de communications général : log n

Exemple : problème d'addition de n nombres (réduction)

- Agglomération et assignation :
  - Assignation du graphe de n tâches sur un ensemble de p processeurs, p < n
  - Résultat :
    - Minimisation des communications
    - Assignation de n/p tâches à chacun des p processeurs
  - Plus besoin de conserver la vision de tâches primitives sur un même processeur, alors :
    - n/p tâches primitives avec une seule valeur -> 1 tâche contenant n/p valeurs

Exemple : problème d'addition de n nombres (réduction)

- Analyse :
  - χ : temps requis pour effectuer une addition
  - λ : temps requis pour communiquer une valeur d'une tâche à une autre
  - Temps de calcul de chaque tâche : $(n/p - 1)\chi$
  - Temps pour une étape de réduction (communication + addition) : $\lambda + \chi$
  - Nombre de réductions : $\log p$
  - Temps de communication : $\log p\ (\lambda + \chi)$
  - Temps total de l'algorithme :
  - $(n/p - 1)\chi + \log p\ (\lambda + \chi)$

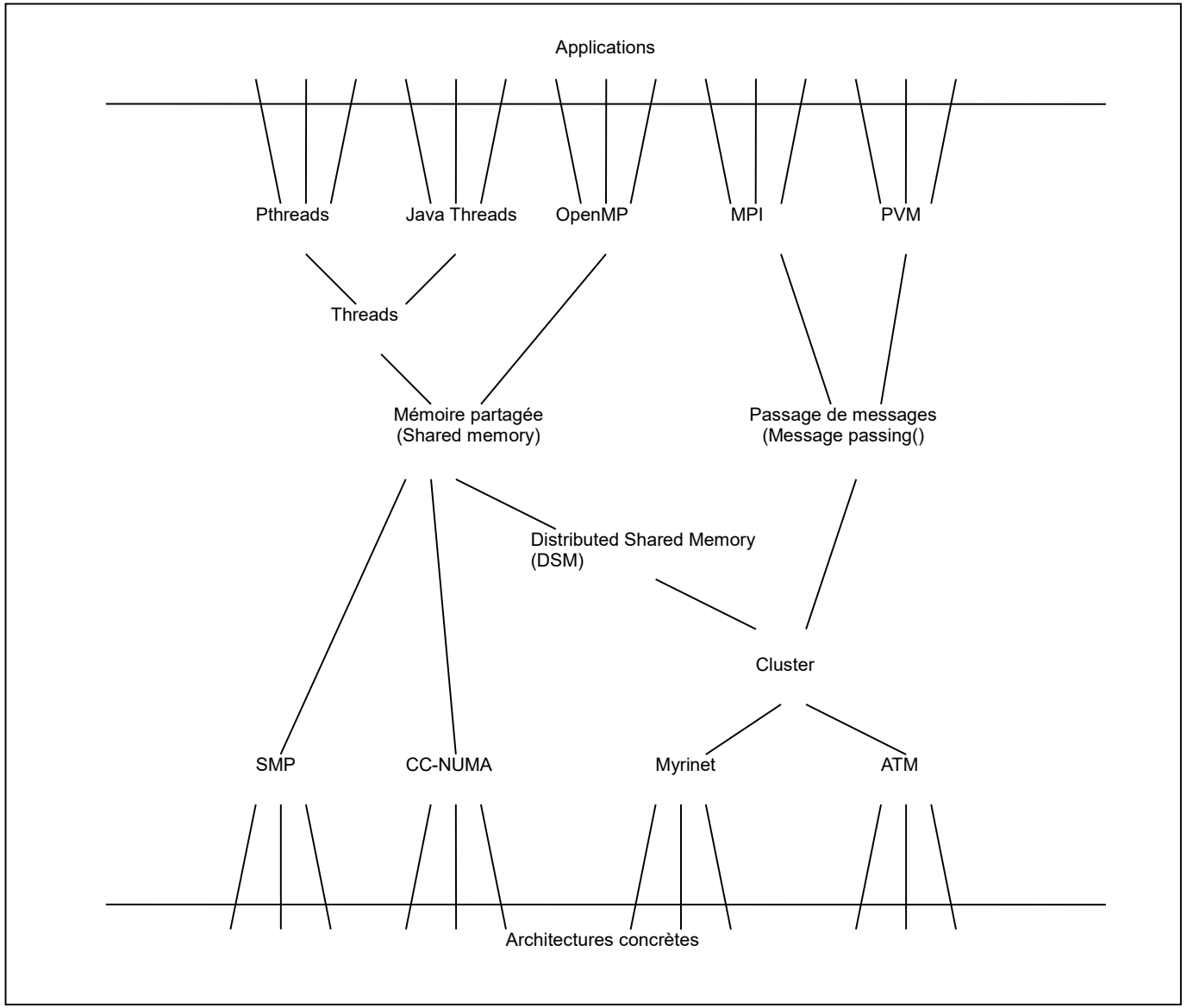
Sommaire

- Modèle tâche/canal
  - Façon efficace de modéliser un algorithme parallèle
  - 4 étapes :
    - Partitionnement, Communication
    - Agglomération, Assignation
  - But principal : déterminer un compromis équitable entre la maximisation de l'utilisation des processeurs et la minimisation des communications entre processeurs

Le modèle à passage de messages

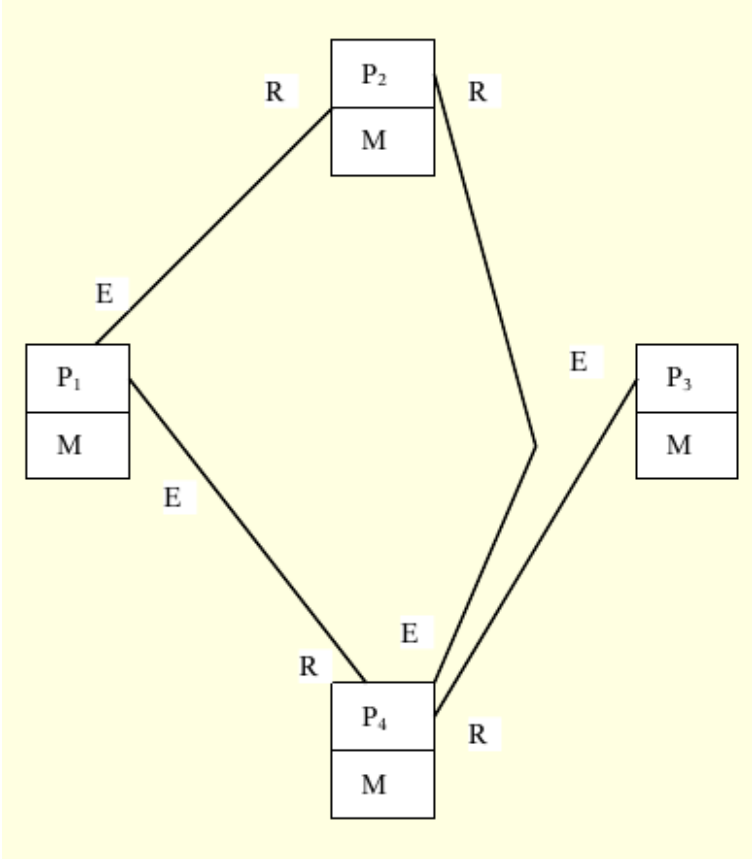
Introduction
Caractéristiques du modèle
Outils de développement
Description d'un programme MPI/C

Vue d'ensemble simplifiée du calcul parallèle des années 2000



Introduction

- Modèle à passage de messages
  - Plusieurs processus s'exécutent en parallèle
  - Processus attribués à des processeurs distincts
  - Chaque processeur possède sa propre mémoire (pas de mémoire partagée)
  - Communications par envois et réception de messages



Introduction

- Principales fonctions de l'échange de message
  - Échange de données
  - Synchronisation
- Particulièrement adapté aux architectures à mémoire distribuée (de type MIMD)
  - Multi-ordinateurs, Clusters et NOW
- Similaire au modèle tâche/canal
  - Tâche -> Processus -> Attribution à un processeur
  - Canal -> Possible entre tous les processeurs

Caractéristiques du modèle

- Difficulté de programmation
- Gestion explicite de
  - La distribution des données
  - L'ordonnancement des tâches
  - La communication entre les tâches
- Conséquences
  - Peut prendre beaucoup de temps
  - Peut mener à plusieurs erreurs
  - Coûts additionnels de développement

Caractéristiques du modèle

- Efficacité
  - Beaucoup de liberté laissée au programmeur
  - Possibilité d'optimisation et de « fine-tuning »
  - Conséquence -> si les ressources sont disponibles, une grande efficacité peut être obtenue

Caractéristiques du modèle

- Portabilité
  - Maturité du modèle
    - Popularité depuis plusieurs années -> standards établis
    - Standards implémentés sur plusieurs machines
  - Conséquences
    - Un programme écrit en respectant les standards peut être facilement transposé sur un autre ordinateur
    - Portabilité du code != Portabilité de la performance !

Développement d’applications

- 2 principaux outils
  - Message Passing Interface (MPI)
  - Parallel Virtual Machine (PVM)
- Englobent une grande proportion des applications parallèles existantes aujourd'hui
- Basées sur des bibliothèques permettant d'étendre des langages séquentiels existants
  - C, C++
  - Fortran

Développement d’applications

- Message Passing Interface (MPI)
  - Standard qui définit un ensemble de fonctions implémentant le passage de messages
  - Un programme comporte plusieurs processus (généralement 1 par processeur)
  - 1ère version : MPI-1 (1994)
  - 2ème version : MPI-2 (1997)

Développement d’applications

- Parallel Virtual Machine (PVM)
  - Standard semblable à MPI, mais moins évolué
  - Principale différence -> Gestion d’une plateforme de passage de messages sur des réseaux hétérogènes
  - 1ère version -> 1989
  - 2e version -> 1993

Description d'un programme MPI/C

```
#include <stdio.h>
#include <mpi.h>  /*bibliotheque MPI*/
int main(int argc, char *argv[]) {
    int id, p;
    int val;
    int somme;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    val = id + 1;
    MPI_Reduce(&val, &somme, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    if (id == 0) {
        printf("La reduction des valeurs donne %d\n", somme);
    }
    MPI_Finalize();
    return 0;
}
```

Inclusion de la bibliothèque MPI

Chaque processeur exécute une copie du programme

Chaque processeur possède ses propres variables

Description d'un programme MPI/C

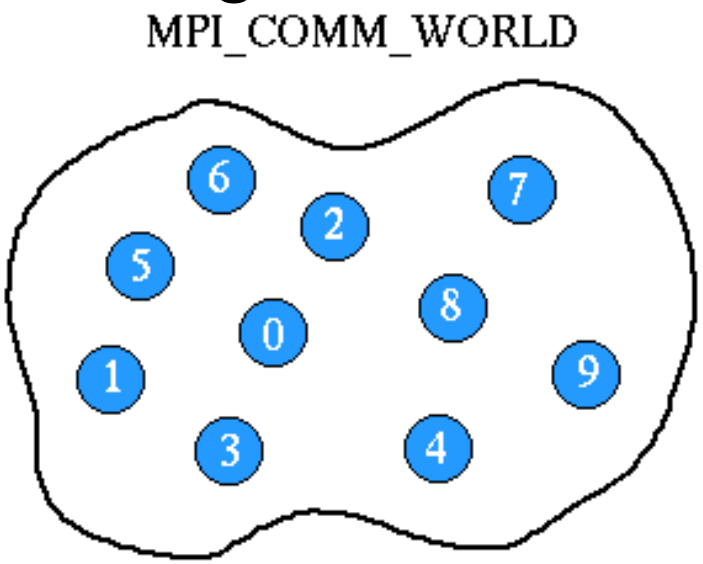
- Appel d'une fonction MPI
  - Format : rc = MPI_Xxx_xxx(paramètre, ...)
  - rc : Code d'erreur

MPI_Init (&argc, &argv)

- Initialise l'environnement d'exécution de MPI
- Doit être appelé
  - Une seule fois dans le programme
  - Avant l'appel d'une autre routine MPI (sauf MPI_Initialized)
- Peut être utilisé, dans certains cas, pour passer les arguments argc et argv du programme C à tous les processus

MPI_COMM_WORLD

- Communicateur : ensemble des processus qui peuvent communiquer les uns avec les autres
- MPI_COMM_WORLD : communicateur défaut
- Plusieurs routines MPI demandent un communicateur en argument



MPI_Comm_rank (MPI_comm comm, int *rank)

- À l'intérieur d'un communicateur, chaque processus a son propre rang/ID (0, 1,…, p-1)
- Permet d'obtenir le rang d'un processus
  - comm : communicateur (Ex.:MPI_COMM_WORLD)
  - rank : le rang est retourné à cette adresse
- Utilisé par le programmeur
  - Spécifier la source et la destination des messages
  - Pour attribuer des parties de programme différentes aux processeurs (if rang == 0 {} else {}

MPI_Comm_size (MPI_comm comm, int *size)

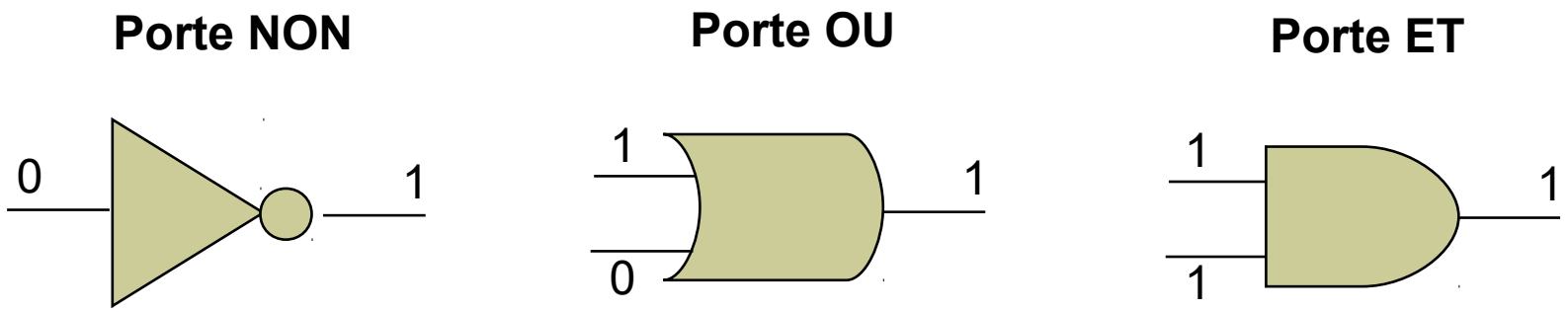
- Détermine le nombre de processus associés à un communicateur
- comm : communicateur (Ex : MPI_COMM_WORLD)
- size : le nombre de processus est retourné à cette adresse

MPI_Finalize ()

- Termine l’environnement d’exécution de MPI
- Doit être appelé une seule fois dans le programme
- Elle est la dernière routine MPI appelée dans un programme (sauf MPI_Initialized)

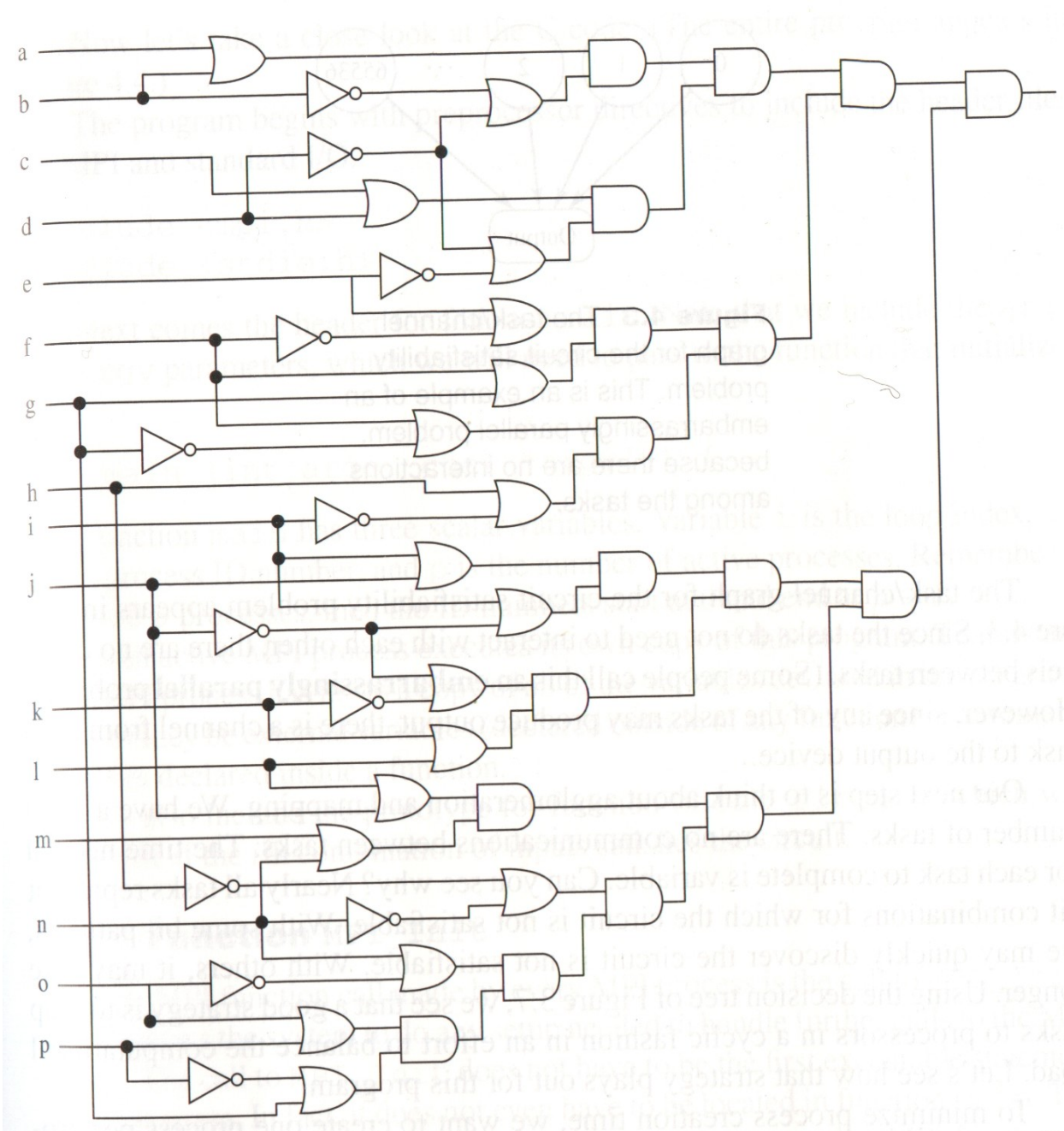
Problème de satisfiabilité d'un circuit

- « Étant donné un circuit combinatoire booléen composé de portes ET, OU et NON, est-il satisfiable ? »
- Un tel circuit est satisfiable s'il existe un ensemble de valeurs d'entrée x_i booléennes qui donne à la sortie du circuit la valeur 1



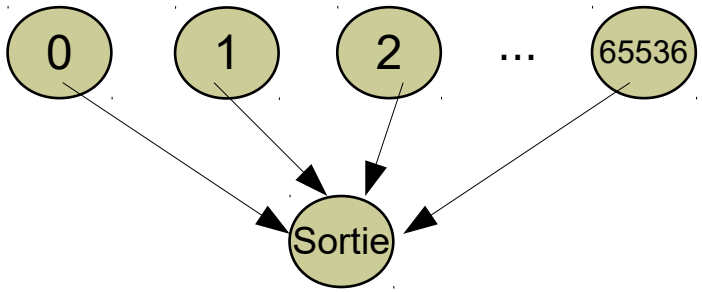
Problème de satisfiabilité d'un circuit

- Pour k entrées : 2^k affectations possibles
- Exemple:
  - 2^{16} = 65 536 possibilités !



Problème de satisfiabilité d'un circuit

- Partitionnement
  - Décomposition fonctionnelle
  - Tâche = Combinaison de 16 entrées
  - 65536 tâches indépendantes a exécuter en parallèle
- Communication
  - Aucun canal entre les tâches, sauf celui de la sortie



Problème de satisfiabilité d'un circuit

- Agglomération :
  - Un processus par processeur
  - n/p tâches par processeur
- Assignation
  - Nombre de tâches -> Statique
  - Temps de calcul par tâche -> Variable
- Résultat -> Allocation cyclique (Approche modulo)

Allocation cyclique

- Tâche k -> Processus k mod p
  - Si p = 6
    - Tâche 0 -> Processus 0
    - Tâche 1 -> Processus 1
    - Tâche 2 -> Processus 2
    - ...
    - Tâche 5 -> Processus 5
    - Tâche 6 -> Processus 0
    - Tâche 7 -> Processus 1
    - ...

Application pour résoudre le problème

- Voir énoncé, programme 1
- Approche modulo pour allouer chacune des 65 536 affectations aux p processeurs :

```
for (i = id; i < 65536; i += p)
    check_circuit (id, i);
```

Application pour résoudre le problème

- La fonction check_circuit (id, z)
  - id : rang du processus
  - z : représentation entière d'une affectation possible
  - la transformation de z en binaire donne les 16 valeurs d'entrée du circuit

```
for (i = 0; i < 16; i++)
    V[i] = EXTRACT_BITS(z, i);
```

  - le « if » de 9 lignes est le codage « à la dure » du circuit

Communications collectives

Ajout de communications collectives

- Voir énoncé – Programme 2
- Problème : déterminer le nb d'affectations possibles pour satisfaire le circuit
- Chaque processeur possède une variable « nbSolutions » qui effectue un compte local
- Une variable nbTotalSolutions permettra au processeur 0 de cumuler le total de solution
- Une opération de réduction doit être effectuée sur la variable nbSolutions

Ajout de communications collectives

- Opération de réduction en MPI
  - MPI_Reduce (void *operand, void *result, int count, MPI_Datatype type, MPI_Op operator, int root, MPI_Comm comm)
- Paramètres
  - operand : &NbSolutions (Compteur local)
  - result : &nbTotalSolutions (Compteur global)
  - count : 1 (Nombre d'éléments transférés → 1 seul entier)
  - type : MPI_INT (variable de type entier → int)

Ajout de communications collectives

- Opération de réduction en MPI
  - MPI_Reduce (void *operand, void *result, int count, MPI_Datatype type, MPI_Op operator, int root, MPI_Comm comm)
- Paramètres
  - operator : MPI_SUM (Somme)
  - root : 0 (rang du processus qui va conserver le résultat)
  - comm : MPI_COMM_WORLD
- Tous les processus doivent appeler la fonction

Les constantes MPI pour les types de données en C

- MPI_CHAR -> signed char
- MPI_DOUBLE -> double
- MPI_FLOAT -> float
- MPI_INT -> int
- MPI_LONG -> long
- MPI_SHORT -> short
- MPI_UNSIGNED -> unsigned int
- ....

Les opérateurs de réduction MPI

- MPI_SUM -> Somme
- MPI_PROD -> Produit
- MPI_MAX -> Maximum
- MPI_MIN -> Minimum
- MPI_BAND -> Et bit à bit
- ...

Calculer le temps d'exécution

- Voir Énoncé – Programme 3
- double MPI_Wtime (void)
  - Retourne le nombre de secondes passées depuis un moment arbitraire dans le passé
  - La différence entre deux valeurs retournées par cette fonction à deux endroits différents donne le nombre de secondes qui ont passé entre ces deux endroits

Calculer le temps d'exécution

- Il faut synchroniser les processeurs de façon globale avant l'appel
  - int MPI_Barrier (MPI_Comm comm)
  - Les processeurs vont attendre que tous ont atteint ce point avant de continuer leur exécution

Diffusion en MPI

- Permet à un processus d'envoyer une ou plusieurs données à tous les autres processus
- MPI_Bcast (void *buffer, int count, MPI_Data_Type datatype, int root, MPI_Comm comm)
- Paramètres
  - buffer : Adresse du premier élément à diffuser
  - count : nombre d'éléments à diffuser
  - datatype : type des éléments à diffuser
  - root : id du processus qui effectue la diffusion
  - comm : communicateur
- Tous les processus doivent appeler la fonction

Dispersion (scatter)

- Permet de répartir un ensemble de données (tableau) présent sur un processeur...
- … sur l'ensemble des processeurs

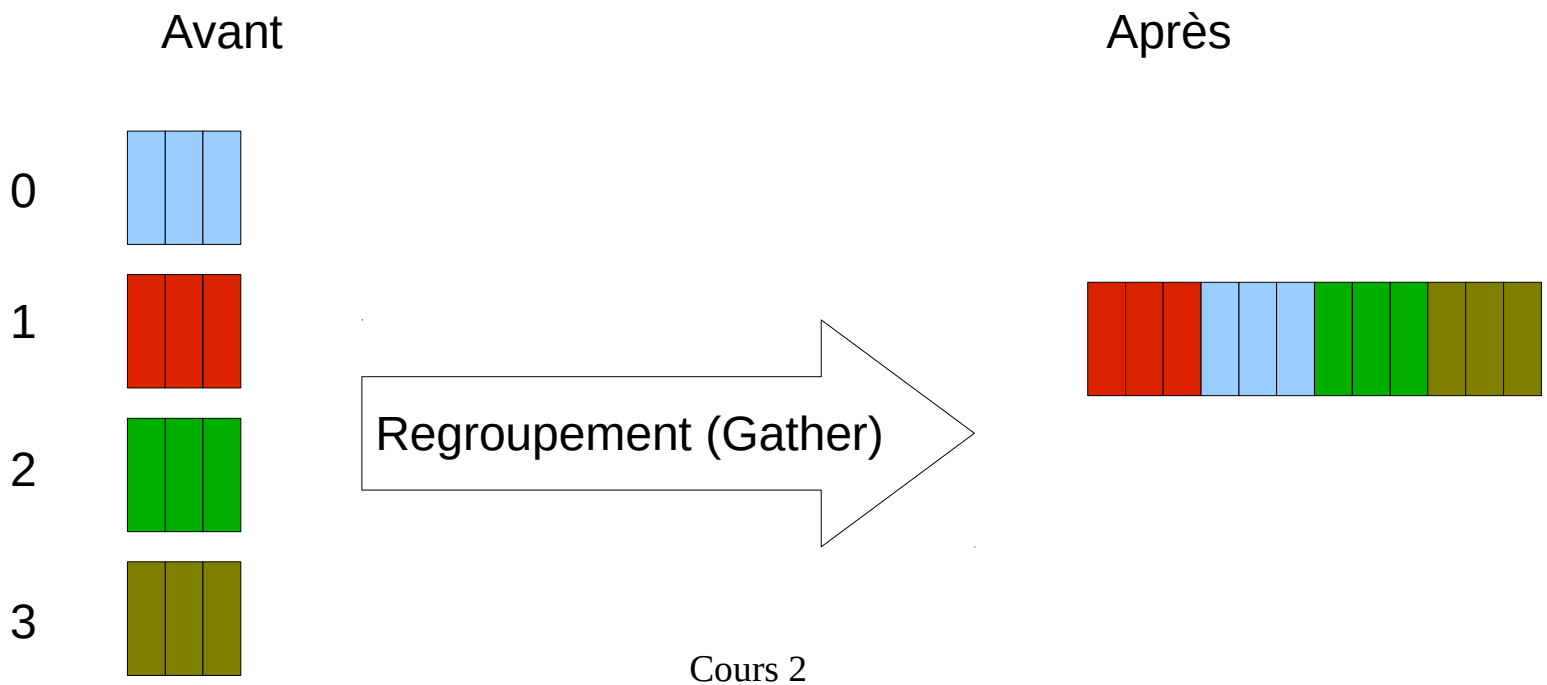
MPI_Scatter

- MPI_Scatter (void *sendbuf, int sendcnt, MPI_Datatype sendtype,
  void *recvbuf, int recvcnt, MPI_Datatype recvtype,
  int root, MPI_Comm comm)

- sendbuf : adresse du tableau d'envoi sur le processus root
- sendcnt : nombre d'éléments envoyés à chaque processus (significatif pour le processus root)
- recvcnt : nombre d'éléments reçus par chaque processus
- root : processus qui envoie les éléments

Regroupement (Gather)

- Permet de réunir des ensembles de données (tableaux) répartis sur les processeurs …
- … sur un seul processeur



MPI_Gather

- MPI_Gather (void *sendbuf, int sendcnt, MPI_Datatype sendtype,
  void *recvbuf, int recvcnt, MPI_Datatype recvtype,
  int root, MPI_Comm comm)

- sendbuf : adresse du tableau d'envoi de chaque processus
- sendcnt : nombre d'éléments envoyés au processus root
- recvbuf : adresse du tableau où les éléments sont stockés
- recvcnt : nombre d'éléments reçus par le processus root de la part de chaque autre processus
- root : processus qui reçoit les éléments

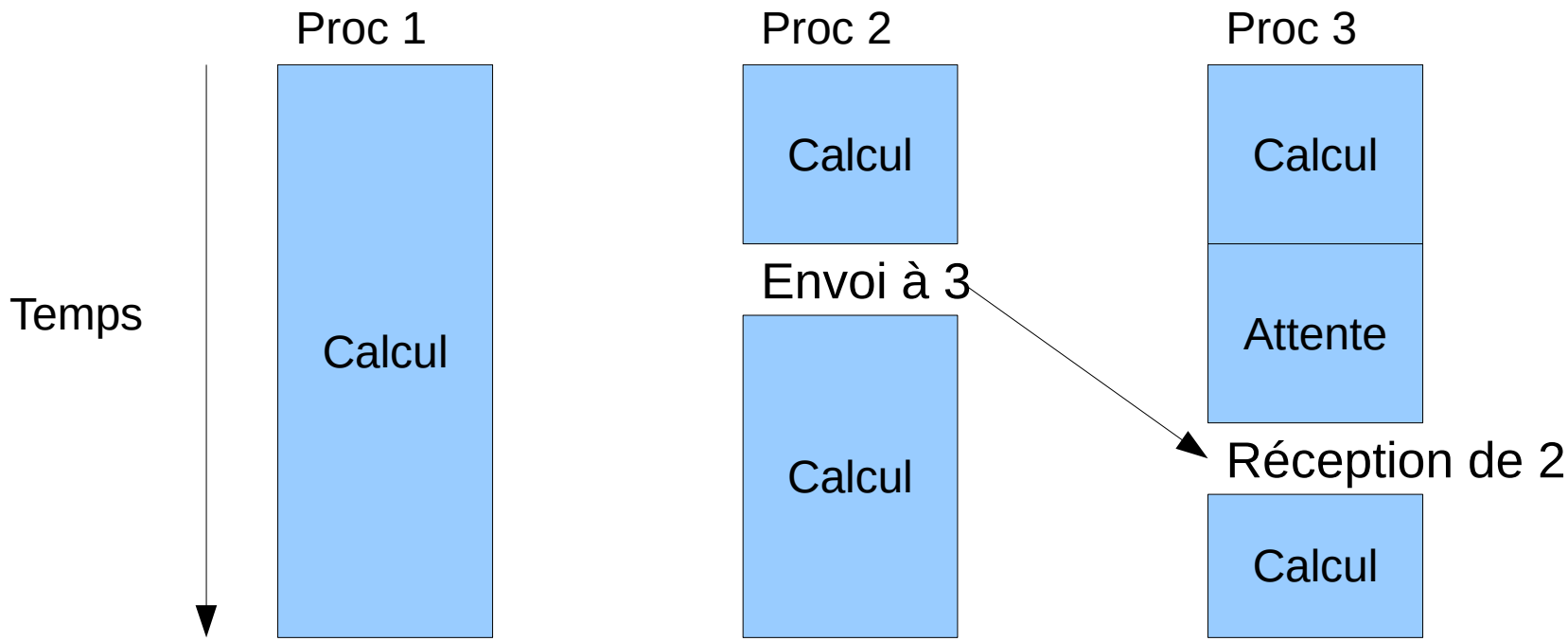
Communications collectives

- Il existes des variantes de gather et scatter pour des dimensions de tableau variables
  - MPI_Gatherv
  - MPI_Scatterv
- Et d'autres formes de communications collectives
  - All-gather
  - All-to-all exchange
- …

Communications point-à-point

Les communications point-à-point

- Implique 2 processus



Les communications point-à-point

- Souvent programmé par du code exécuté conditionnellement (if…then…else)
  - if (id == i)
    - Envoi d'un message à j
  - else if (id == j)
    - Réception d'un message de i
- Avec MPI
  - MPI_Send
  - MPI_Recv

Envoi de message

- MPI_Send (void *message, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
- Paramètres :
  - message : adr de la première donnée à envoyer
  - count : nombre de données à envoyer
  - datatype : le type des données
  - dest : le rang (id) du processus destinataire
  - tag : étiquette pour identifier le message
  - comm : communicateur (sous-groupe)
- Communication non bloquante
  - le processus peut reprendre ses calculs après envoi

Réception de message

- MPI_Recv (void *message, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *Status)
- Paramètres :
  - message : adr pour les données reçues
  - count : nb de données que le proc est prêt à recevoir
  - datatype : le type des données
  - source : rang (id) attendu du processus qui envoie
  - tag : étiquette attendue identifiant le message
  - comm : communicateur (sous-groupe)
  - Status : Enregistrement contenant de l'information additionnelle (MPI_source, MPI_tag, MPI_ERROR)
- Communication bloquante
  - le processus suspend son exécution tant que le message n'est pas reçu

Verrous mortels (deadlocks)

- Un processus est dans un état de deadlock s'il est bloqué en attendant une condition qui ne deviendra jamais vraie
- Des erreurs de conception et/ou programmation peuvent mener à cet état
  - a attend un msg de b et b attend un msg de a
  - Mauvaise spécification de tag
  - Erreur de destinataire ou de source

Exemple de verrou

```
float a = 1, b = 2, c = 3;
int id;
MPI_Status statut;
MPI_Comm_Rank(MPI_COMM_WORLD, &id);
if (id == 0) {
    MPI_Recv(&b,1,MPI_FLOAT,1,0,MPI_COMM_WORLD, &statut);
    MPI_Send(&a,1,MPI_FLOAT,1,0,MPI_COMM_WORLD);
}
else if (id == 1) {
    MPI_Recv(&a,1,MPI_FLOAT,0,0,MPI_COMM_WORLD, &statut);
    MPI_Send(&b,1,MPI_FLOAT,0,0,MPI_COMM_WORLD);
}
```

La semaine prochaine

Modèle à mémoire partagée
Programmation OpenMP