Calculateurs MIMD à mémoire distribuée

Philippe d'Anfray CEA ANR-CI

2007-2008

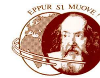
Philippe.d-Anfray@cea.fr

Formation d'Ingénieurs de l'Institut Galilée MACS 2

Partie 1

Calculateurs MIMD à mémoire distribuée

- MIMD Intro, Contexte; Modèle.
- Topologie réseau; Routage; Calculateur hôte.
- Modèle de programmation.
- Équilibrage, granularité



MIMD (1) Introduction

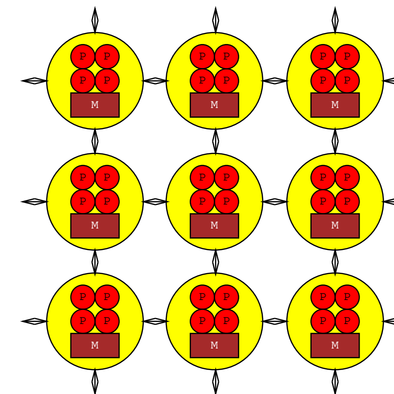
A chaque architecture correspond un modèle de programmation et donc des outils et des langages adaptés.

- vectoriel ou super scalaire;
- SIMD;
- MIMD à mémoire partagée;
- MIMD à mémoire distribuée.

Les machines actuelles permettent d'utiliser ces modèles simultanément.

MIMD (2)

Ainsi, un "réseau" dont chaque nœud est constitué de plusieurs processeurs vectoriels sur une mémoire partagée.



MIMD (3)

Dans le domaine du calcul haute performance il faut prendre en compte l'architecture "cible"

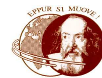
- un impact sur la modélisation;
- un impact sur la programmation;
- nécessite des outils et des méthodes spécifiques.

"à quel niveau optimiser un code en fonction de quelle caractéristique technique"

MIMD (4)

Mettre en évidence les modèles de programmation dont relèvent les différentes parties de son application.

1. le séquentiel (bien sûr).
2. le vectoriel, le super-scalaire (voire SIMD) concerne quelques lignes de code.
3. le MIMD à mémoire partagée concerne de petites unités de programme (quelques centaines de lignes).
4. le MIMD à mémoire distribuée (échange de messages) concerne toute l'application.
5. le MIMD à mémoire virtuelle partagée ou DSM concerne toute l'application.



MIMD mémoire distribuée le plus général, le reste c'est de l'"optimisation locale"

MIMD (5) Contexte

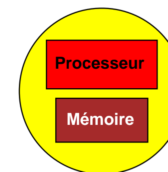
L'idée est bien de multiplier le nombre des processeurs pour augmenter la puissance de la machine.

On utilisera ici un **grand nombre de processeurs** (éventuellement peu puissants): le plus souvent quelques centaines.

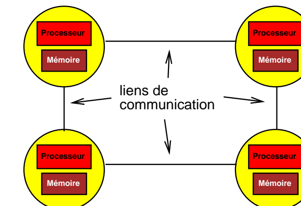
Rappel: Il n'est alors plus possible de partager la mémoire, le débit serait trop faible et les conflits d'accès trop nombreux.

MIMD (6)

Chaque processeur est indépendant et constitue avec sa mémoire le **nœud** d'un réseau dont les arcs sont les **liens de communication**.



nœud du réseau



un calculateur MIMD à mémoire distribuée

MIMD (7)

- Il n'y a pas de **contrôle centralisé**
  La machine fonctionne en mode **asynchrone**, en particulier chaque processeur possède sa propre horloge.
- La "mesure" de la taille des tâches exécutées en parallèle est appellée **granularité**.
  Celle-ci va, comme nous l'avons vu, de quelques lignes (vectoriel) à des programmes complets (MIMD à mémoire distribuée).

MIMD (8)

- Chaque processeur exécute donc son propre flot d'instructions sur ses propres données: **tâche** (ou **processus**).
- La mémoire est **distribuée**, les données sont dans les mémoires locales.
- L'accès à une donnée située sur un autre processeur se fait par l'intermédiaire d'un mécanisme d'échange de messages.

MIMD (9) Modèle

Sur les calculateurs MIMD à mémoire distribuée, la technique la plus utilisée est actuellement:

- **la programmation par échange de messages**

C'est un modèle très général, les autres apparaissent comme des **optimisations** -de plus en plus- locales.

Il existe des outils standards: `MPI`, `PVM` ...

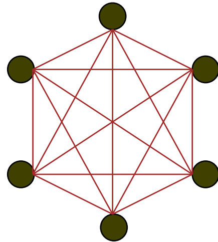
Partie 1-suite

Calculateurs MIMD à mémoire distribuée

- MIMD Intro, Contexte; Modèle.
- Topologie réseau; Routage; Calculateur hôte.
- Modèle de programmation.
- Équilibrage, granularité

Topologie du réseau (1)

Premier problème à résoudre: choix d'une **topologie** pour le réseau.
L'idéal: réseau **totalement connecté**. Irréaliste: trop grand nombre de liens.



Un réseau totallement connecté

Topologie du réseau (2)

La **distance entre deux nœuds** est le nombre de liens distincts à emprunter pour acheminer une information de l'un vers l'autre.

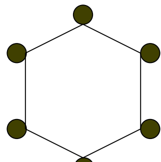
Dans un réseau **totalement connecté**, les processeurs sont à distance un.

La topologie sera un compromis entre:

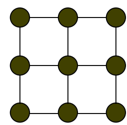
- la difficulté à construire la machine;
- la nécessité de maintenir une faible distance entre les nœuds.

Topologie du réseau (3)

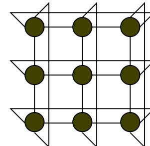
Toutes sortes de machines ont donc été conçues et/ou fabriquées:



Anneau



Grille 2D (ou 3D)

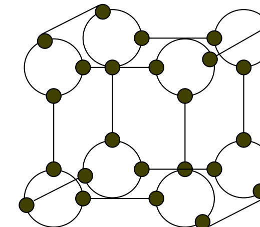


Tore 2D (ou 3D)

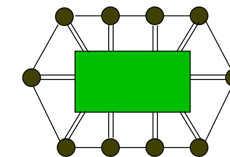
. . . plus ou moins bien adaptées à certains types d'applications. Le problème reste minimiser les liens.

Topologie du réseau (4)

On a envisagé des solutions “mixtes”, et des réseaux reconfigurables:



Topologie "mixte", cube dont chaque sommet est un anneau



Réseau reconfigurable. La base est un anneau. Chaque noeud possède deux liens vers un commutateur programmable.

Le constituant central d'un réseau reconfigurable est un commutateur programmable permettant de relier “à la demande” les nœuds entre eux.

Topologie du réseau (5)

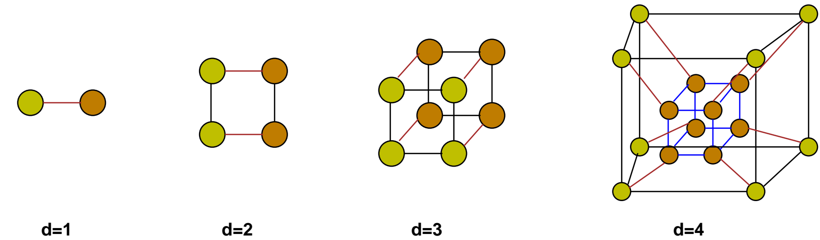
Un bon compromis: l'architecture hypercube qui a été beaucoup employé (Intel iPSC, N-cube,. . .) vers la fin des années 80 -le débit des liens est encore seulement de quelques Mo par seconde-.

Ce type d'architecture reste d'actualité (SGI).

Les hypercubes se construisent de façon récursive en reliant un à un les sommets de deux hypercubes de dimension inférieure.

Topologie du réseau (6)

Exemples d'hypercubes



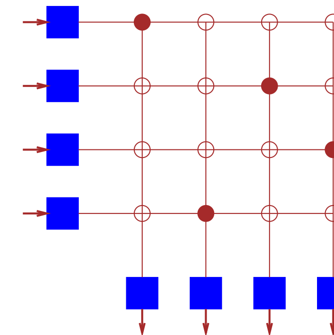
Topologie du réseau (7)

Si l'hypercube possède 2^d nœuds, la distance maximal e entre deux nœuds est de d.

- l'évolution de ces machines devient difficile dès que le n ombre de processeurs augmente (128, 256, 512, etc. . .);
- le partage entre plusieurs applications -qui doivent chacune utiliser un *sous cube*- est un mécanisme lourd qui complique l' exploitation en centre de calcul de ce type de m achine.

Topologie du réseau (7)

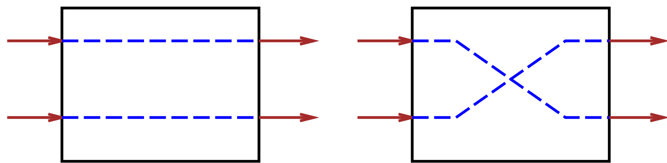
Les réseaux du type "*crossbar*" (cf. totalement connecté. . .)



utilisés aussi dans l'interconnexion Processeurs bancs-mémoires (CRAY)

Topologie du réseau (8)

Les réseaux à étages sont basés sur l'utilisation de composants: "*switch*" (à 2, 4, 8 voies, . . .):

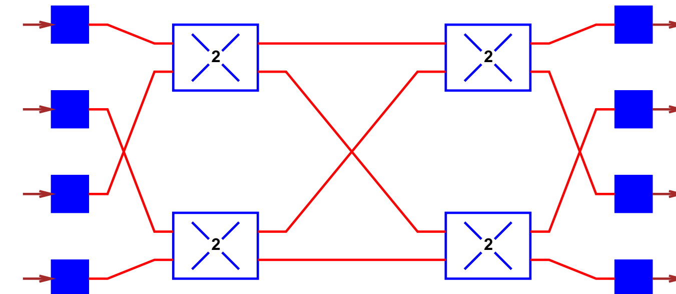


Les deux états d'un "switch" à deux voies

donnent une vision "totalement connecté"

Topologie du réseau (9)

Exemple d'un réseau à deux étages, 4 composants (*switch* à deux voies) permettent d'émuler un *crossbar* 4 par 4 (16 connexions).



progression logarithmique du nombre de composants, nombreuses variantes (*omega*, *perfect shuffle*, etc . . .)

Topologie du réseau (10)

Les réseaux sont maintenant plus rapides: plusieurs centaines de Mo voire plusieurs Go par seconde.

- la distance entre les nœuds n'est plus un problème;
- l'engorgement *a priori* non plus, tout au moins sur des systèmes de tailles limitée à quelques centaines de nœuds.

De plus les calculateurs actuels sont destinés à être utilisés en *exploitation* dans des centres de calcul. Il est donc nécessaire qu'elles soient facilement partageables entre plusieurs travaux.

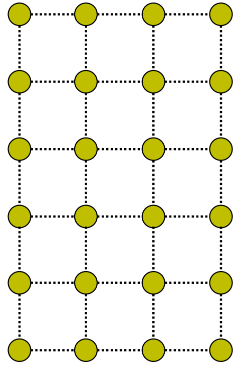
Topologie du réseau (11)

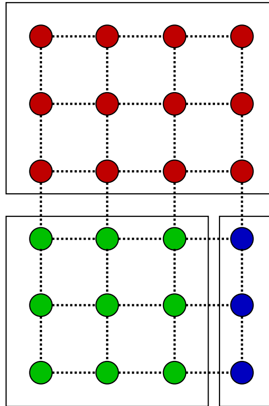
Les calculateurs actuels adoptent des topologies simples

- grille 2D pour l'INTEL PARAGON (années 80);
- tore 3D pour le CRAY T3E (années 90);
- architectures à étages (IBM, SGI, SUN, etc. . .).

Les politiques de routage sont simplifiées au maximun, la machine est facile à partager.

Topologie du réseau (12)



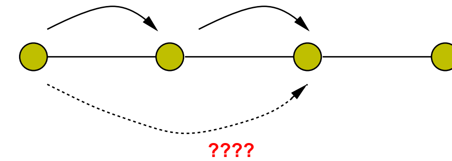


La machine est une grille 2D

Il n'y a pas d' "interférences" entre les partitions

Routage (1)

Si tous les nœuds ne sont pas à distance un, il se pose un **problème de routage**: (acheminer des informations entre deux nœuds distants).



Avant: écrire sur les nœuds intermédiaires un processus *relais* assurant le transfert du message.
Maintenant: les calculateurs comportent des **routeurs** qui assurent l'acheminement de l'information.

Routage (2)

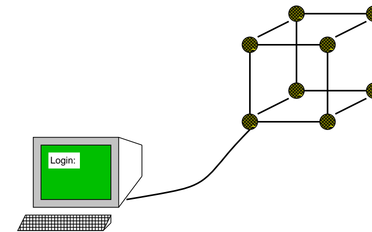
Pour l'utilisateur, la machine est totalement connectée **mais**:

- le coût peut néanmoins dépendre de la distance réelle à parcourir;
- de nombreux messages transitent par les mêmes liens, il peut se produire des phénomènes d'engorgement du réseau.

Le calculateur hôte (1)

L'accès aux calculateurs MIMD se faisait via un calculateur hôte.
Il n'était pas possible, d'avoir sur chacun des nœuds un système d'exploitation complet.

L'hôte se charge donc en général des entrées sorties , du lancement des tâches sur les nœuds etc. . .



Le calculateur hôte (2)

Sur les machines de dernière génération, le système d'exploitation est un **UNIX réparti** (*single system image*).

- pour l'utilisateur, il y a une seule ressource et il se connecte directement sur la machine;
- on trouve sur chaque nœud toutes les fonctionnalités d'UNIX.

Le calculateur hôte n'est plus nécessaire, mais reste néanmoins très utilisé.

Partie 1

Calculateurs MIMD à mémoire distribuée

- MIMD Intro, Contexte; Modèle.
- Topologie réseau; Routage; Calculateur hôte.
- Modèle de programmation.
- Équilibrage, granularité

Programmation (1)

Tâches communicantes.

Un programme est un ensemble de tâches indépendantes qui échangent de l'information.

- **parallélisme à gros grain**. Plus précisement, la granularité est liée au rapport:
  coût du calcul / coût de la communication
  la communication coûte cher, en général;
- l'utilisateur doit définit la taille de ses tâches en fonction du nombre de processeurs disponibles et de la réponse à la question:
  vais-je gagner quelque chose en coupant les tâches existantes ?

Programmation (2)

Une remarque fondamentale: le parallélisation efficace ne proviendra pas d'un travail sur le code lui-même. Il faudra souvent:

- développer de nouveaux algorithmes, exemple: les méthode par blocs;
- déveloper de nouvelles méthodes, exemple: les méthode de sous-domaine;
- travailler sur les modèles. . .

Programmation (3)

Le mot-clef dans ce domaine est **localité**.

On cherchera des algorithmes mettant en évidence des calculs sur des données locales en minimisant ainsi les transferts.

Le mécanisme d'échange de messages sert à traduire les transferts d'informations et les nécessaires synchronisations entre tâches.

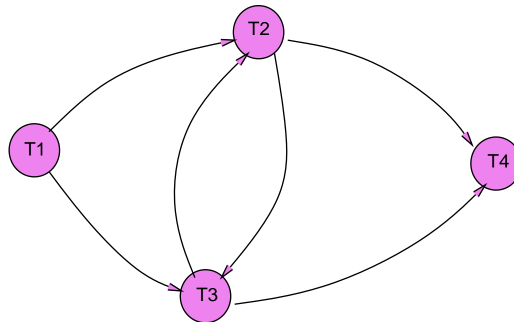
Programmation (4)

Dans le cas général, il s'agit:

- d'isoler les tâches de calcul;
- d'étudier leurs dépendances mutuelles:
  - synchronisations, communications. . .

l'application peut être vue comme un graphe appelé **graphe logique**.

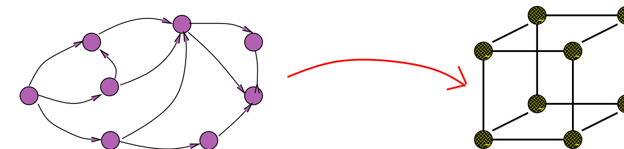
Programmation (5)



Rappel: le mécanisme déchange de messages traduit les transferts d'informations et les nécessaires synchronisations entre tâches.

Le placement (1)

Le graphe logique de l'application peut être complexe et il s'agit de le *placer* au mieux sur le **graphe physique** constitué par les nœuds de la machine et son réseau:



Quelles tâches exécuter sur quels nœuds pour minimiser le délai total d'exécution.

Le placement (2)

Le problème général de placement d'un graphe sur un graphe est très difficile à résoudre.

C'est même un problème *NP-complet*

On dispose souvent d'**heuristiques** permettant de trouver de bonnes solutions approchées.

Néanmoins on procède généralement plus simplement. . .

Modèle SPMD (1)

Le plus souvent, le découpage en tâches est guidé par les données.

On utilise un modèle **SPMD**: le même programme s'exécute sur chaque nœud de la machine.

Ce programme décrit ce qui se passe **localement sur le nœud** (pas l'ensemble de l'application)

Le problème principal reste d'équilibrer la charge de calcul sur les nœuds en minimisant les communications.

Approche objet (1)

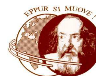
Digression: les méthodes orientés objets permettent de construire des logiciels de meilleure qualité.

- Question: **Y-a-t-il un lien avec le parallélisme: ?**
- Conclusion (optimiste):
  - Les stratégies de parallélisation sont souvent basées sur les données.
  - Les objets modélisent les éléments de ces décompositions.
  - On effectue (en majorité) des calculs **locaux** sur ces objets.

Partie 1

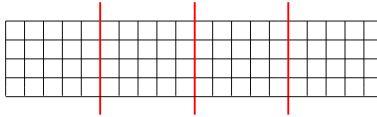
Calculateurs MIMD à mémoire distribuée

- MIMD Intro, Contexte; Modèle.
- Topologie réseau; Routage; Calculateur hôte.
- Modèle de programmation.
- Équilibrage, granularité.

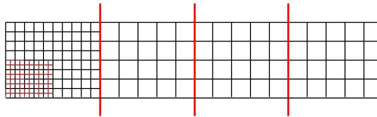


L'équilibrage (1)

Il n'est pas toujours possible de prévoir la taille des tâches à exécuter; ex) si l'on fait du "raffinement de maillage", un équilibrage *a priori* se révélera rapidement inefficace:



Répartition a priori d'un maillage sur quatre processeurs



Après raffinement local, les tâches deviennent déséquilibrées

L'équilibrage (2)

L'équilibrage de la charge de calcul doit donc , en général être un processus dynamique.

Le prôblème devrait être pris en compte au niveau du système d'exploitation de la machine (avec une gestion de file d'attente de tâches etc. . .).

L'équilibrage de la charge peut impliquer des redistribution des données de l'application.

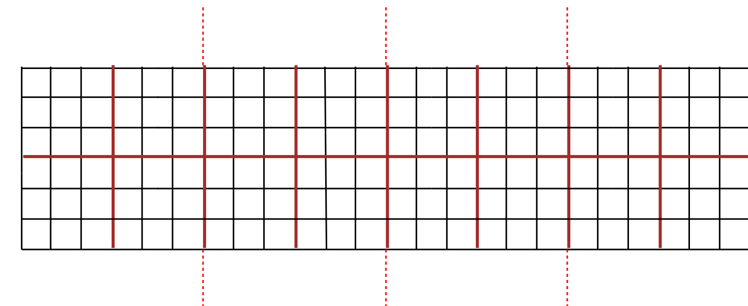
Équilibrage/Granularité (1)

Dans les méthodes numériques du type multigrille, on utilise plusieurs maillages imbriqués.

Le problème est résolu alternativement, jusqu'a convergence de la solution, sur les grilles "fines" et "grossières".

Dans le cas de la grille grossière, la **granularité** des tâches parallèles est insuffisante et il est plus rapide de résoudre sur un seul processeur.

Équilibrage/Granularité (2)



Grille "fine"

Grille "grossière"

Équilibrage/Granularité (3)

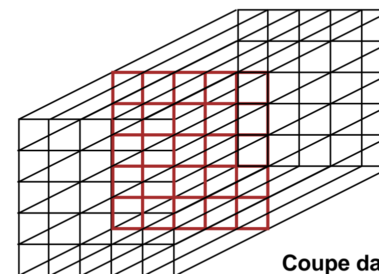
Au contraire, cas d'une application graphique sur un objet tridimentionnel.

La sélection d'un plan dans la structure conduira à effectuer des calculs éventuellement complexes sur des données localisées dans un seul processeur.

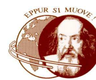
Ici il faudra répartir la tâche sur tous les processeur.

Il existe des *boites à outils* informatiques permettant de gérer facilement ces redistributions de données (notamment MPI).

Équilibrage/Granularité (4)



Coupe dans un objet 3d



Conclusion(s) (1)

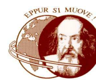
Programmation des "super-calculateurs" massivement parallèles points importants:

- calculateur `MIMD` à mémoire distribuée;
- modèle de tâches communicantes;
- notions de granularité, localité, équilibrage;
- programmation par échange de messages;
- approche `SPMD`.

Partie 2, les outils

Les outils, la bibliothèque `MPI`

- Introduction.
- Environnement.
- Communication point à point.
- Fonctions globales.
- Entrées Sorties.
- Mesures de temps, traces.



Les outils (1)

L'outil permet de programmer les modèles de tâches et d'échanges de messages. On y trouve:

- accès à l'environnement;
- communications point à point;
- opérations "globales": synchronisation, diffusion, réduction;
- prise en compte de la topologie de la machine;
- le reste. . . (notamment entrées sorties).

La bibliothèque `MPI` est un bon exemple (il existe aussi `PVM`, etc. . .).

Partie 2-suite

Les outils, la bibliothèque `MPI`

- Introduction.
- Environnement.
- Communication point à point.
- Fonctions globales.
- Entrées Sorties.
- Mesures de temps, traces.

Environnement (1)

Une tâche doit savoir sur quel nœud de la machine elle est exécutée:

```
moi = noeud_numero ();
```

et combien de nœuds il y a dans la machine:

```
nb_proc = combien_de_noeuds();
```

Les nœuds sont toujours numérotés de `0` à `N-1`

Partie 2-suite

Les outils, la bibliothèque `MPI`

- Introduction.
- Environnement.
- Communication point à point.
- Fonctions globales.
- Entrées Sorties.
- Mesures de temps, traces.

Communications point à point (1)

Deux sous-programmes (au moins) sont nécessaires pour envoyer ou recevoir un message.

Ils se présentent généralement sous cette forme:

```
envoyer  (ident, adresse, deplacement, dest);
           ...
recevoir (ident, adresse, deplacement );
```

Communications point à point (2)

Les messages sont identifiés par:

- des numéros `ident` (ou par l'expéditeur)

On envoie au `destinataire` le contenu d'une zone mémoire identifiée par:

- une `adresse`;
- et un `deplacement`.

Communications point à point (3)

Il y a plusieurs protocoles d'envoi et de réception de messages.

On peut les classer selon l'impact des requêtes (`envoyer`, `recevoir`) sur le déroulement de l'application (vision *software*):

- non bloquant;
- localement bloquant;
- bloquant.

Communications point à point (4)

La vision *hardware*, conduit à distinguer plusieurs modes de fonctionnement du calculateur:

- mode asynchrone (chaque processeur est indépendant);
- mode synchrone (notion de contrôle centralisé).

Certains protocoles de communication sont rendus possibles par les détails de l'architecture matérielle.

Communications point à point (5)

Le plus "standard" est l'asynchrone, localement bloquant:

- l'envoi ou la réception d'un message n'a pas d'impact sur le déroulement des autres processus (concernés ou non);
- il n'y a pas de notion de rendez-vous;
- l'ordre n'est pas forcement respecté.

Notion de boîte aux lettres.

Communications point à point (6)

Ce mécanisme suppose l'existence de zones mémoire “tampon” de taille à priori illimitées (ce qui n'est jamais le cas en pratique).

Le principe est le suivant:

- le sous-programme `envoy_lb` rend la main lorsque les données à émettre sont sauvegardées dans cette zone tampon.
- le sous-programme `recevoir_lb` rend la main lorsque les données à recevoir sont recopiée de la zone tampon dans la zone “utilisateur”.

Communications point à point (7)

Cela fonctionne comme bien un système de *boîtes aux lettres* dans lesquelle sont stockés les messages en attente d'émission ou de réception.

On peut généralement interroger le système (*sonder*) pour savoir si un message d'un certain `ident` est arrivé:

```
reponse = sonder (ident);
```

N.B. `ident` identifie le message.

Communications point à point (8)

On considérera l'asynchrone non bloquant comme le mécanisme “de base”.

Les autres protocoles, si ils existent serviront à optimiser le programme:

- “recouvrir”, c'est à dire effectuer simultanément calculs et comunications sur un même processeur: gain de temps.
- éviter d'utiliser les mémoires tampon: gain de temps et possibilité d'envoyer des messages de grande taille;

Communications point à point (9)

Asynchrone, non bloquant: les sous-programmes `envoyer_nb` et `recevoir_nb` rendent immédiatement le contrôle au programme appelant.

Les opérations se passent donc en deux temps:

```
ii= envoyer_nb (id_1, ad_i1, depl, j);
...
...
attendre (ii) ;
```

On peut ainsi "recouvrir" calculs et communications.
L'ensemble est équivalent à un envoi (resp. une réception) asynchrone localement bloquant(e).

Communications point à point (10)

Cas de l'émission: entre les deux appels, peuvent prendre place des calculs qui ne modifient pas les données à envoyer:

```
     real*8 x, y (100)
     ...
     ii= envoyer_nb (id1, x , 800, 2)
     ...
c === calculs ne modifiant pas x
     ...
     call attendre (ii)
     ...
c === calculs utilisant  x
     ...
```

Communications point à point (11)

Cas de la réception: entre les deux appels, peuvent prendre place des calculs qui n'utilisent pas les données à recevoir:

```
      real*8 x, y (100)
      ...
      jj= recevoir_nb (id1, y , 800)
      ...
c === calculs n'utilisant pas y (lecture et ecriture)
      ...
      call attendre (jj)
      ...
      c === calculs utilisant y
      ...
```

Communications point à point (12)

Synchrone, bloquant: ce protocole suppose un **rendez-vous** entre processus; impact sur l'ensemble de l'application.

On distingue deux variantes:

- rendez-vous "simple": `envoyer_s` ne peut commencer à s'exécuter que si le `recevoir_s` correspondant a commencé.
- rendez-vous "prêt": si le `recevoir_p` correspondant n' a commencé, `envoyer_p` provoquera une erreur.

Intérêt: ces échanges peuvent se faire sans mémoire tampon.

Communications point à point (13)

Récapitulation:

Protocole	Remarque
Asynchrone localement bloquant	Attention: suppose un tampon “infini”. . .
Asynchrone non bloquant	se passe en deux temps attente localement bloquante
Synchrone, rdv “prêt”	bloquant erreur si l’envoi avant réception
Synchrone, rdv “simple”	bloquant

Communications point à point (14)

Il existe d’autres protocoles plus spécialisés (messages actifs etc. . .)

Dans la pratique le code peut être conçu avec des envois et des réceptions asynchrones localement bloquants puis optimisé:

- en recouvrant calculs et communications lorsque cela est possible;
- en utilisant des rendez-vous lorsque de grands messages sont utilisés.

N.B. la machine ne “supporte pas toujours” tous les protocoles.

Partie 2-suite

Les outils, la bibliothèque `MPI`

- Introduction.
- Environnement.
- Communication point à point.
- Fonctions globales.
- Entrées Sorties.
- Mesures de temps, traces.

Fonctions Globales (1)

Un premier exemple, synchroniser: c’est une fonction globale.

Tous les nœuds appellent

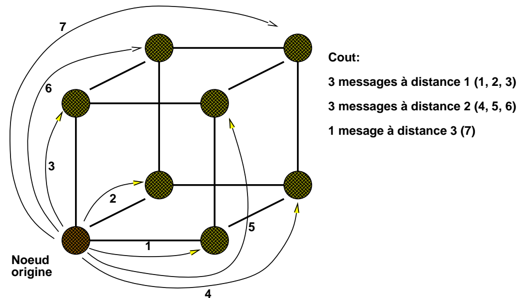
```
call synchro()
```

et redémarrent en même temps.

Souvent appelée "barrière de synchronisation" (ne pas en abuser. . .).

Fonctions Globales (2)

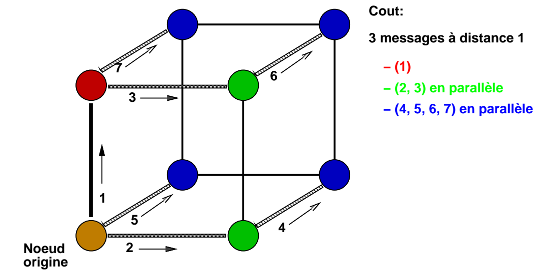
Il en existe bien d'autres, pourquoi??, ex: diffusion par un nœud (ex 0) d'unr information:



Coût: (en temps écoulé) **7 envois de messages**

Fonctions Globales (3)

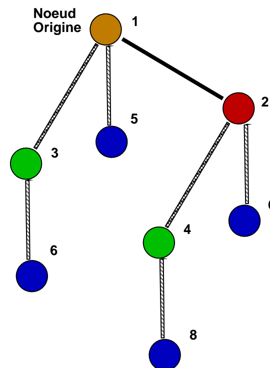
On peut faire beaucoup mieux !, tous les nœuds participent au processus:



Coût: (en temps écoulé) **3 envois de messages**

Fonctions Globales (4)

Comment faire? on construit un **arbre de recouvrement:**



Les autres **opérations globales** sont optimisées de la même façon.
Attention: opérations globales ==> **synchronisation des nœuds**.

Fonctions Globales (5)

Récapitulation, les principales fonctions:

- barrière de synchronisation (*Barrier*);
- diffusion (*Broadcast*);
- opération de réduction (*Reduce*);
- redistribution de données:
  - disperser des données (*Scatter*);
  - rassembler des données (*Gather*).

Partie 2-suite

Les outils, la bibliothèque `MPI`

- Introduction.
- Environnement.
- Communication point à point.
- Fonctions globales.
- Entrées Sorties.
- Mesures de temps, traces.

Entrées/Sorties (1)

Les tâches qui s'exécutent sur les nœuds peuvent accéder aux systèmes de fichiers UNIX des "frontales".

Pour de "petites" entrées/sorties il est possible d'utiliser les systèmes de fichiers de ces frontales.

l'accès doit se faire de préférence depuis **un seul des nœuds**.

Entrées/Sorties (2)

Un exemple:

Lecture de quelques données:
le nœud `0` lit puis diffuse les valeurs aux autres nœuds.

Ecriture de résultats:
le nœud `0` récupère et écrit dans l'ordre les résultats de tous les nœuds.

Pour des "grands" volumes de données:

- il est **impératif** d'utiliser le système d'entrées/sorties parallèle de la machine;
- il est nécessaire d'utiliser une interface de programmation adaptée (`MPI-IO`).

Partie 2-suite

Les outils, la bibliothèque `MPI`

- Introduction.
- Environnement.
- Communication point à point.
- Fonctions globales.
- Entrées Sorties.
- Mesures de temps; traces.

Mesure de temps (1)

On dispose généralement d'une fonction `horloge`, qui permet de mesurer des intervalle de temps sur chaque nœud:

```
synchronisation();
debut= horloge ();
/*  .. algorithme  */
temps_proc=horloge()-debut;
/* puis prendre le max sur tous les noeuds !! */
temps = reduction_max (temps_proc);
...
```

Rappel: pas d'horloge centralisée.

Les traces (1)

Le débogage des programmes parallèles est souvent très complexe. Il faut utiliser des informations collectées à l'exécution. Les traces permettent aussi d'évaluer l'activité des processeurs et du réseau et d'apprécier le caractère parallèle du programme.