Processus concurrents et parallélisme

Chapitre 6 - Notions de calcul parallèle et distribué

Gabriel Girard

17 octobre 2022

Chapitre 6 - Notions de calcul parallèle et distribué

Chapitre 6 - Notions de calcul parallèle et distribué

Défis

- Non-déterminisme
- Communication, synchronisation et interblocage
- Partitionnement et distribution des données
- Balancement de la charge
- Tolérance aux fautes
- Hétérogénéité
- Mémoire partagée et distribuée
- Condition de course (race condition)

Défis

- Communication
  Comment se fait la communication ?
  Comment réduire les coûts de communication (minimisation) ?
- Balancement de la charge
  Comment bien répartir la charge de travail ?
  Doit-on regrouper certaines tâches ?

Approches

- Implicite
- Explicite

Types de calcul

- Programmation concurrente (mono-processeur)
- Programmation concurrente (multiprocesseurs)
- Programmation concurrente (réseau)
  - Applications parallèles et distribuées
  - Calcul parallèle

Besoins

- Modèle algorithmique
- Modèle de communication
- Modèle de synchronisation
- Modèle architectural (mapping)
- Outils de développement

Chapitre 6 - Notions de calcul parallèle et distribué

Modèle algorithmique

- Malgré toutes les applications possibles, seulement quelques modèles existent
- Il y a deux façons de paralléliser une application
  - parallélisation par les données (data parallel programs)
  - parallélisation par les fonctions (task parallel programs)

Parallélisation selon les données

- Utilisée pour les applications parallèles et distribuées
- Modèles :
  - Parallélisation triviale (embarassingly parallel)
  - Parallélisation itérative
  - Parallélisation récursive
  - Parallélisation peer-2-peer

Parallélisation triviale (données)

- Séparation des données afin d'obtenir des traitements indépendants
- Fonctionne selon le principe administrateur/travailleurs
- L'administrateur sépare les données, les distribue aux travailleurs et récolte les résultats
- Une construction bien adapté : Map
- Exemples :
  - briser les mots de passe
  - seti@home
  - recherche de modèle dans ARN et protéines
  - multiplication de matrices

Administrateur/Travailleurs

- Simple
- Permet de répartir la charge
- Aussi appelé Bag of tasks, administrator/worker
- C'est la contrepartie du modèle peer-2-peer
- Cas possibles :
  - planification statique (nombre de processeurs $\geq$ nombre de tâches)
  - planification dynamique (nombre de processeurs $<$ nombre de tâches)

Administrateur/Travailleurs

- Problèmes à résoudre
  - que faire si un travailleur tombe en panne ?
  - que faire si le l'administrateur tombe en panne ?
  - goulot d'étranglement ?
- Avantages :
  - offre un balancement facile et efficace
  - chaque processeur exécute environ la même quantité de travail

Construction : Map

- Construction qui s'applique à tous les éléments d'une ≪liste≫ (potentiellement en parallèle)
- Principe de fonctionnement :

  Map(fonction, liste)
- Dans le monde séquentiel, présente surtout dans les langages fonctionnels
- Exemples de langages supportant ≪Map≫ parallèle :
  - OpenMP et Cilk (Boucle parallèle)
  - OpenCL et Cuda (kernel)
- Elle est souvent utilisé avec une autre construction : ≪Reduce≫

Parallélisation itérative (données)

- Ressemble à la parallélisation triviale
- Les traitements ne sont pas complètement indépendants
- Les travailleurs peuvent s'échanger de l'information
- Les travailleurs peuvent travailler par étapes

Parallélisation récursives (données)

- Applique le principe de diviser pour régner
- S'applique aux programmes qui divisent les données par récursivité et qui produisent des appels récursifs indépendants
- Exemple :
  - tri rapide, tri fusion
  - jeu d'échec (backtracking)
  - aire sous la courbe par approximation (adaptative quadrature)

Parallélisation peer-2-peer (données)

- Semblable au parallélisme itératif mais sans la relation Administrateur/Travailleurs
- Sert principalement à la prise de décision décentralisée
  - sémaphore distribué
  - horloge logique
  - élection
- La communication ne se fait pas nécessairement avec les voisins immédiats

Parallélisation par les fonctions

- Pas vraiment de modèles algorithmiques
- Dépend du problème à résoudre
- Exemples
  - clients/serveur
- Ce genre de parallélisation est principalement caractérisé par le modèle de communication

Modèle de communication

- Spécifie les règles pour l'échange d'information entre les différentes unités de calcul
- Plusieurs modèles existent

Clients/serveur

- Pour les applications réparties
- Parallélisation selon les fonctions

Administrateur/Travailleurs

- Pour le calcul parallèle
- Relation inverse du clients/serveur

Diffusion

- Parallélisation selon les données et les fonctions
- Principalement dans les applications réparties
- Pour la prise de décisions répartie
  - horloge logique
  - élection
  - sémaphore distribué

Anneau/jeton

- Parallélisation selon les données et les fonctions
- Pour les applications réparties
- Pour la prise de décisions répartie
  - horloge logique
  - élection
  - sémaphore distribué

Pipeline

- Parallélisation selon les données et les fonctions
- Pour les applications réparties, concurrentes et parallèles
- Exemples
  - ordinateur systolique
  - pipe dans Unix

Hartbeat

- Propage l'information par l'intermédiaire des voisins immédiats (dissémination) (exploration progressive de l'horizon)
- Modèle

```
loop
    envoi msg à tous les voisins
    reçoit msg de tous les voisins
```

- Cette information est envoyée et reçue périodiquement
- Implante une forme de barrière
- Fréquemment utilisé dans l'approche peer-2-peer

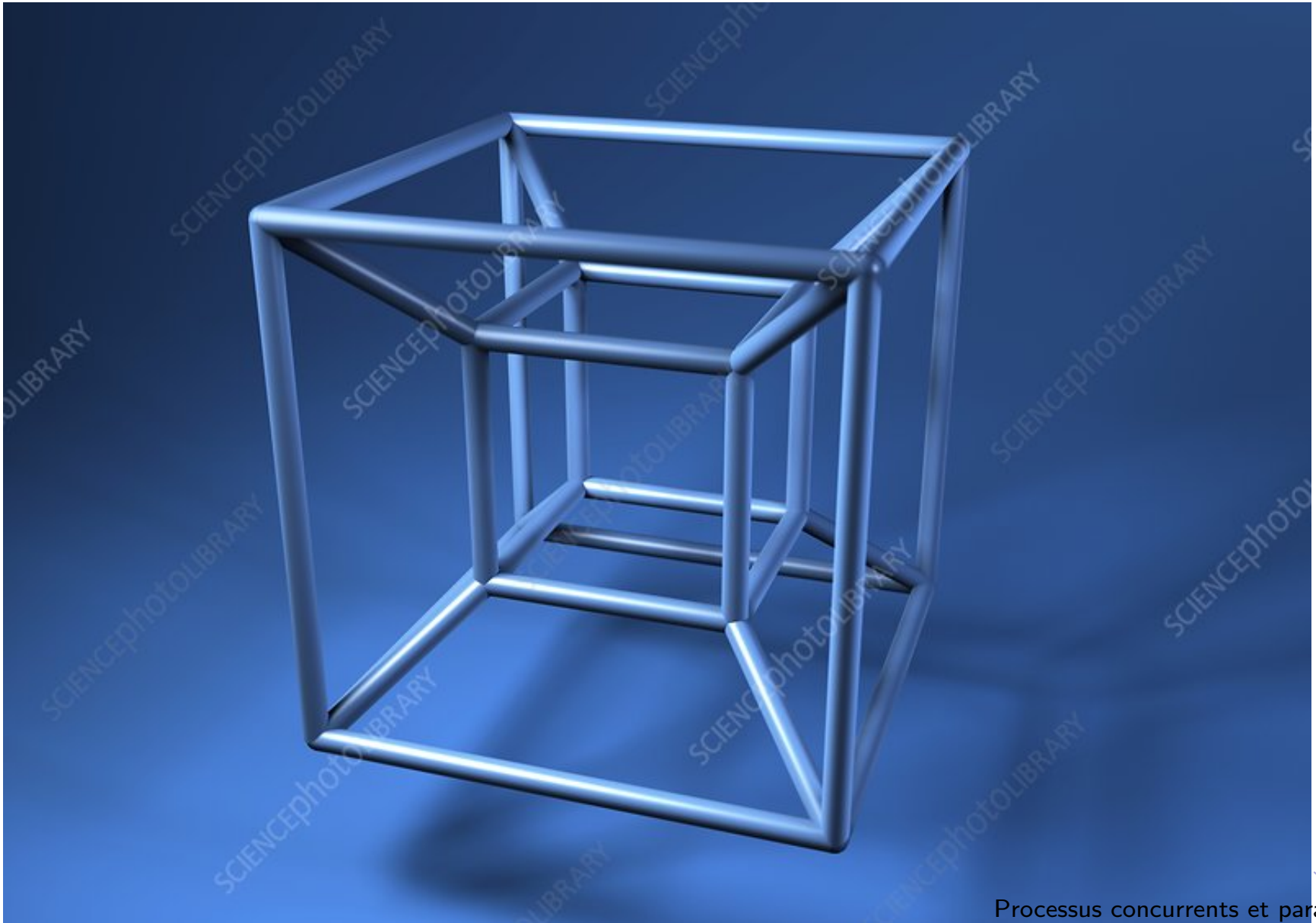
Probes/echoes

- Propage l'information par l'intermédiaire des voisins immédiats
- Pour les topologies en graphe ou en arbre
- Envoi des messages aux successeurs et reçoit des réponses de ces mêmes noeuds

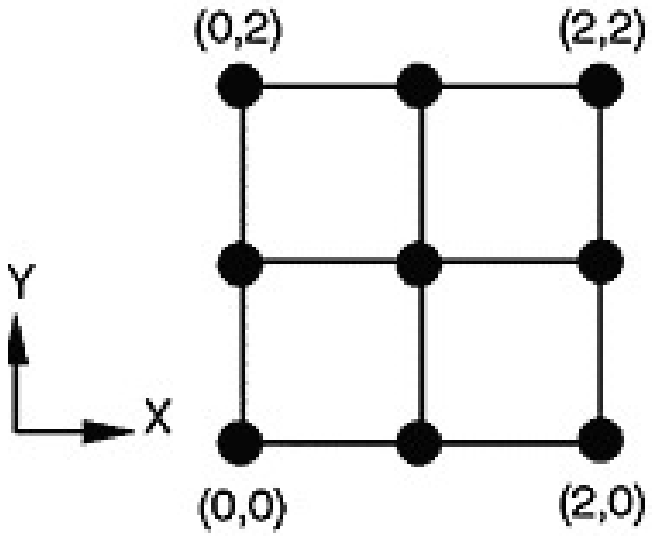
Autres

- Serveurs multiples (utilisé dans Grids)
- Hypercube
- Mesh

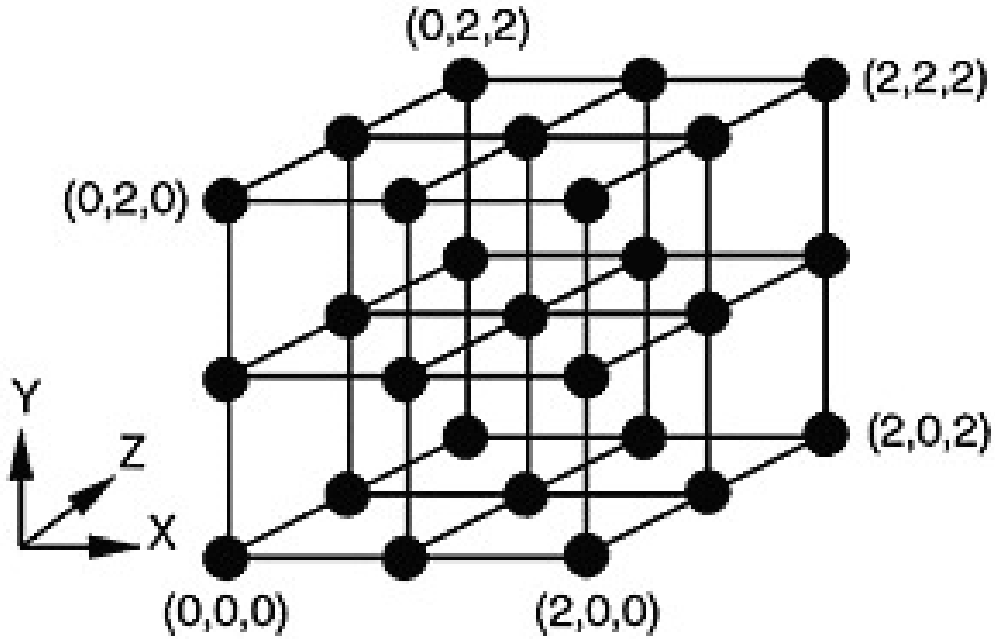
Autres



Autres



(a) 3 x 3 2D mesh



(b) 3 x 3 x 3 3D mesh

Modèle architectural

- Le modèle de communication doit être projeté sur le modèle architectural (matériel)
- Contraintes : coûts des communication et planification

Modèle architectural

- Ne concerne pas ce cours
- En pratique : mémoire commune et bus (réseau)
- Exemple de modèle architectural : Mesh, Hypercube, Tile

Modèle de synchronisation

- Synchrone
  - pour les problèmes itératifs
  - chaque travailleur exécute par étapes
  - les travailleurs se synchronisent à la fin de chaque étape
  - ce type de synchronisation s'appelle une barrière
- Asynchrone
  - chaque travailleur progresse à sa vitesse
  - algorithmes complexes (flots de données, ...)
  - exemple : simulation à événements discrets

Synchrone

```
process travailleur [i:= 1 to n]
begin
   while(true)
   begin
      code pour la tâche i
      -- attendre que les n tâches complètent
   end
end
```

Synchrone

Mauvais exemple.....Inefficace....

```
while(true)
  begin
    co [i:=1 to n]
      code pour la tâche i
    oc
  end
```

Barrière

- Il existe plusieurs implantations pour les barrières
  - basé sur les variables globales ou les messages
  - basé sur un coordonnateur ou une approche symétrique
  - basé sur d'autres structures de communication (arbre, ...)

Barrière

```
while(true)
  begin
     code
     --- cpt++
     --- attendre (cpt =n)
  end
```

Barrière

```
while(true)
  begin
     code
     --- envoi msg à tous
     --- reçoit msg de tous
  end
```

Chapitre 6 - Notions de calcul parallèle et distribué

Outils de programmation parallèle

- Peu importe le modèle utilisé et le type de concurrence visé, il est nécessaire d'utiliser un outil qui permet de les implanter correctement

Caractéristiques

Doivent fournir :

- transparence architecturale et de type de processeur
- transparence au niveau des comm. et du réseau
- facilité d'utilisation et fiabilité
- support pour la tolérance aux fautes
- support pour l'hétérogénéité
- portabilité
- support pour les langages traditionnels
- augmentation de performance
- transparence au niveau de la concurrence

Exemples d'outils

- MPI
- OpenMP
- MOM (JMS, Active MQ, ...)
- HPF (High Performance Fortran)
- Cuda, openCL
- PVM
- Langages : Sisal, PCN, CC++, Mentat, ...
- C++, C#, Java, Python, ... (applications réparties)
  - RMI, Corba, Pyro, Onc, ...

Chapitre 6 - Notions de calcul parallèle et distribué

Introduction

- Il y a de plus en plus d'applications concurrentes
  - À cause de la limite du séquentiel
  - À cause du coût peu élevé des multiprocesseurs, multi-coeurs et des grappes
  - À cause d'internet
  - ...
- Les applications sont de plus en plus variées

Exemples d'application

- Applications réparties
  - Serveurs Http, ftp, fichiers
  - Prise de décisions
  - ...
- Calcul scientifique
  - trois techniques fondamentales
  - calcul de grilles (grid computing), calcul de particules et calcul matricielle

Calcul scientifique

- Calcul de particules
  - modélisation des forces qu'exercent une sur l'autre les particules (molécules, planètes)
- Calcul matriciel
  - systèmes d'équations

Grid computing

- Solution numérique aux équations différentielles partielles (prévision météorologique, circulation d'air sur une aile, turbulence des fluides, ...)
  - on projette le problème sur une matrice
  - on applique un traitement répétitif jusqu'à convergence vers un état stable
- Traitement d'images
  - la matrice est initialisée avec les valeurs des pixels
  - on applique un traitement répétitif sur chaque pixel

Grid computing

- Exemple d'équation différentielle partielle : équation de Laplace
  - S'approxime numériquement avec l'itération de Jacobi, Gauss-Seidel et SOR (Successive Over Relaxation)
  - Jacobi
    - les bords de la matrice sont initialisés à la valeur cible
    - l'intérieur reçoit les valeurs de départ pour le calcul
    - on calcule de façon répétitive de nouvelles valeurs pour les valeurs internes (moyenne des voisins)
    - on termine après K itérations ou après un test de convergence (EPSILON)

Calcul parallèle : exemple 1 - Somme de n valeurs

Comment calculer la somme de N valeurs contenues dans un tableau.

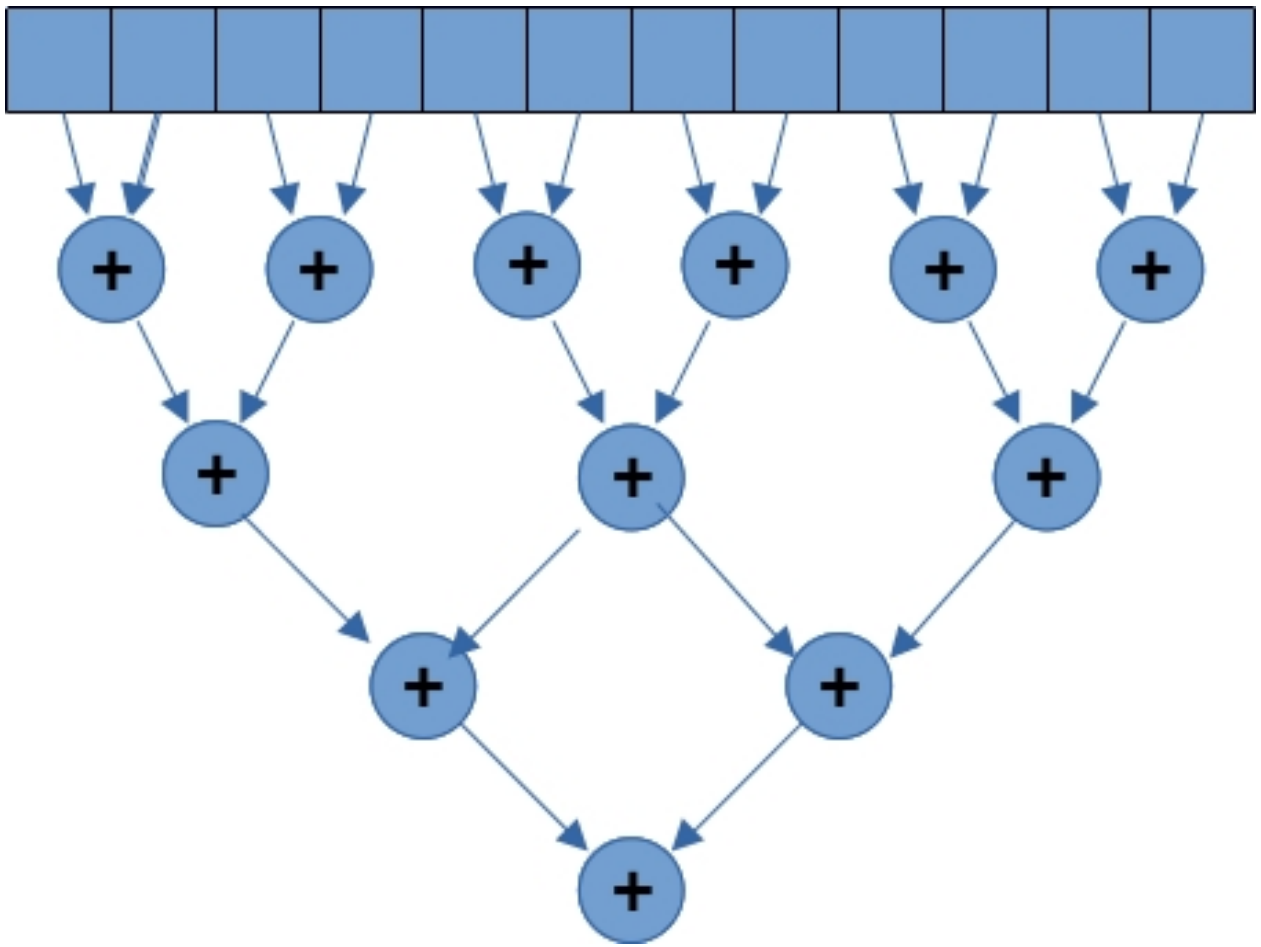
Exemple 1 - - Somme de n valeurs

```
resource main()
        const N := 8  # number of processes
        sem continue[N] := ([N] 0)
        sem done := 0
        var x[1:8]: int := (1,2,3,4,5,6,7,8)
        var nbr : int := 8

process barriere
  do true ->
       fa w := 1 to N -> P(done) af
       fa w := 1 to N -> V(continue[w]) af
  od
end

process calcul(i:= 1 to 8 )
   fa k := 1 to 3 ->
       if i <= nbr/(2**k) -> x[i] := x[2*i] + x[2*i-1]
       fi
       V(done); P(continue[i])
   af
   if i=1 -> write (i, " ", x[i]) fi
end calcul
end main
```

Exemple 1 - - Somme de n valeurs



Grid comp. : exemple - multiplication de matrices (1)

```
resource matrice()
    const n := 4
    var a[1:n,1:n]: int, b[1:n,1:n]: int, c[1:n,1:n]: int

    fa i := 1 to n ->
        fa j := 1 to n -> a[i,j]:=1;b[i,j]:=2  af
    af
```

Grid comp. : exemple - multiplication de matrices (1)

```
    fa i := 1 to n ->
       fa j := 1 to n ->
             c[i,j]:=0
             fa k := 1 to n ->
                c[i,j]:= c[i,j] + a[i,k]* b[k,j]
             af
        af
    af


    fa i := 1 to n ->
         fa j := 1 to n -> printf("% i ,", c[i,j] ) af
         write()
    af
end matrice
```

Grid comp. : exemple - multiplication de matrices (2)

```
resource matrice()
    const n := 4
    var a[1:n,1:n]: int, b[1:n,1:n]: int, c[1:n,1:n]: int
    fa i := 1 to n ->
         fa j := 1 to n ->
                 a[i,j]:=1; b[i,j]:=2; c[i,j]:=0
         af
    af
```

Grid comp. : exemple - multiplication de matrices (2)

```
process calcul(i := 1 to n )
      var j,k:int
      fa j := 1 to n ->
          c[i,j]:=0
          fa k := 1 to n ->
              c[i,j]:= c[i,j] + a[i,k]* b[k,j]

          af
      af
 end

final
 fa i := 1 to n ->
      fa j := 1 to n -> printf("% i ,", c[i,j] ) af
      write()
 af
end
end matrice
```

Grid comp. : exemple - multiplication de matrices (3)

```
resource matrice()
    const n := 4
    var a[1:n,1:n]: int, b[1:n,1:n]: int, c[1:n,1:n]: int
    fa i := 1 to n ->
         fa j := 1 to n ->
            a[i,j]:=1; b[i,j]:=2; c[i,j]:=0
         af
    af
```

Grid comp. : exemple - multiplication de matrices (3)

```
process calcul(i := 1 to n, j:=1 to n )
   var k:int
   c[i,j]:=0
   fa k := 1 to n -> c[i,j]:= c[i,j] + a[i,k]* b[k,j]
   af
   write("Processus ", i,j, " termine")
end

final
  fa i := 1 to n ->
     fa j := 1 to n -> printf("% i ,", c[i,j])
     af
     write()
  af
end
end matrice
```

Grid comp. : exemple - multiplication de matrices (4)

```
resource matrice()
    const n := 4
    var a[1:n,1:n]: int, b[1:n,1:n]: int, c[1:n,1:n]: int
    #initialisation de la matrice
    fa .... af
```

Grid comp. : exemple - multiplication de matrices (4)

```
procedure scalaire(i: int)
   var j,k:int
   fa j := 1 to n ->
       c[i,j]:=0
       fa k := 1 to n ->
             c[i,j]:= c[i,j] + a[i,k]* b[k,j]
       af
   af
end

co (i := 1 to n ) scalaire(i) oc

fa i := 1 to n ->
   fa j := 1 to n -> printf("% i ,", c[i,j] ) af
   write()
af
end matrice
```

Grid comp. : exemple - multiplication de matrices (5)

```
resource matrice()
  # Lire la taille de la matrice
  var n: int; getarg(1,n)
  # declarer et initialiser les matrices
  var a[n,n] := ([n] ([n] 1.0)),
      b[n,n] := ([n] ([n] 1.0)),
      c[n,n] : real
  # calculer le produit scalaire de a[i,*] * b[*,j]
```

Grid comp. : exemple - multiplication de matrices (5)

```
# calculer le produit scalaire de a[i,*] * b[*,j]
procedure scalaire(i,j: int)
  var somme := 0.0
  fa k := 1 to n -> somme +:= a[i,k]*b[k,j] af
  c[i,j] := somme
end

# calculer n**2 produit scalaire en parallele
co (i := 1 to n, j := 1 to n) scalaire(i,j) oc


# imprimer le résultat
fa i := 1 to n ->
  fa j := 1 to n -> writes(c[i,j], " ") af
  write()
af
end
```

Tri rapide

```
resource quick()
  op sort(var a[1:*]: int)  #  declaration de sort()

  var n: int; getarg(1,n)
  var a[1:n]: int

  # lecture des données
  fa i := 1 to n -> read(a[i]) af
  write("input:"); fa i := 1 to n -> write(a[i]) af

  sort(a)

  write("sorted:"); fa i := 1 to n -> write(a[i]) af
```

Tri rapide

```
  proc sort(a)
    if ub(a) <= 1 -> write("fin tri", ub(a)); return fi
    fa i := 1 to ub(a) -> writes(a[i], " ") af; write()
    var pivot := a[1]
    var lx := 2, rx := ub(a)

    do lx <= rx ->
        if a[lx] <= pivot -> lx++
        [] a[lx] >  pivot -> a[lx] :=: a[rx]; rx--
        fi
    od

    a[rx] :=: a[1]

    co sort(a[1:rx-1]) // sort(a[lx:ub(a)]) oc

  end
end quick
```

Sommes partielles

```
resource partiel()
    const n := 10
    var a[n]: int, somme[n]: int, temp[n]: int
    fa i := 1 to n -> a[i]=i af


    process sum(i := 1 to n)
         var d:int:=1
         somme[i] := a[i]
         barrier(i)
         do d<n ->
           begin
             temp[i] := somme[i]
             barrier(i)
             if i-d >= 0 ->somme[i] := temp[i-d]+ somme[i]
             fi
             barrier(i)
             d := d+d
           end
         od
    end
```

Sommes partielles

```
    final
        fa i := 1 to n -> printf("% i ,", somme[i] ) af
        write()
    end
end partiel
```

Grid computing : Jacobi séquentiel

```
resource jacobi()
    const EPSILON := .01
    const n := 4
    var grid[0:n+1,0:n+1]: real, temp[0:n+1,0:n+1]: real
    var diff:real, maxdiff : real:=1, borne:real
#.......initialisation de la grille...................
```

```
    do maxdiff > EPSILON -> begin
       fa i := 1 to n ->
         fa j := 1 to n ->
             temp[i,j]:=(grid[i-1,j]+ grid[i+1,j]+
                         grid[i,j-1]+grid[i,j+1])/4
         af
       af
      maxdiff :=0
       fa i := 1 to n ->
         fa j := 1 to n ->
             diff := abs(temp[i,j] - grid[i,j])
             if diff>maxdiff -> maxdiff := diff fi
         af
       af
       fa i := 1 to n ->
         fa j := 1 to n -> grid[i,j] := temp[i,j] af
       af
     end
    od
  #impression resultat
end jacobi
```

Grid computing : Jacobi parallèle

```
resource jacobi()
    const EPSILON := .01, n := 4
    var g[0:n+1,0:n+1]: real, t[0:n+1,0:n+1]: real
    var maxdiff[1:n,1:n]:real
    var borne:real
#.........initialisation de la grille.................
```

```
    process calcul(i :=1 to n, j:=1 to n)
      var k:int, l:int, fini:bool :=false;
      maxdiff[i,j]:=1
      do fini=false ->
        t[i,j]:=(g[i-1,j]+ g[i+1,j]+g[i,j-1]+g[i,j+1])/4
        maxdiff[i,j] := abs(t[i,j] - g[i,j])
        barrier(i,j)
        fini := true
        fa k := 1 to n ->
            fa l := 1 to n ->
               if maxdiff[k,l] > EPSILON -> fini := false fi
            af
        af
        barrier(i,j)
        fa i := 1 to n ->
          fa j := 1 to n -> g[i,j] := t[i,j]; af
        af
      od
    end

    final
       #impression resultat
    end
  end jacobi
```

Conclusion