Programmation Parallèle

Parallélisme de données

Plan

- Introduction aux langages data parallèles
  - Principe
  - Collections
  - Opérateurs
- High Performance Fortran (HPF)
  - Itérateurs
  - Directives de placement
  - Introduction à la compilation
  - Extensions

Principes

- Modèle de programmation SEQ(PAR)
  - Flot de contrôle séquentiel
  - Le parallélisme est décrit par des types de données collections
- Collections
  - Type décrivant un ensemble (au sens non-technique) de données
  - Munies d'opérateurs
    - Parallèles génériques
    - Spécifiques

```
real A(N), B(N), C(N)
do i = 1 , n
   A(i) = B(i) + C(i)
end do
```

```
real A(N), B(N), C(N)
A = B + C
```

Collections : typologie

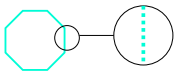
	Répétition valeurs	Pas de répétition valeurs
Accès	Tableau	Dictionnaire (Map)
Pas d'accès	Liste	Ensemble (Set)

Exemple : les collections java
http://java.sun.com/docs/books/tutorial/collections
⚠ – Pas d'exécution parallèle

4

Collections : typologie

- Les éléments peuvent-ils être des collections ?
  - Décrit plusieurs niveaux de parallélisme



  - Listes de listes
  - Tableau irrégulier (ragged array)

5

Opérateurs parallèles

- α-extension : étendre à la collection un opérateur valide pour le type des éléments

  α+{1,2,3}{4,5,6} => {5,7,9}
  - Contraintes de cohérence : tableaux conformes, éléments de même clé,…
- Syntaxe
  - Explicite : *Lisp : !! (+!! A(!!1))
  - Surcharge d'opérateurs A = B+ C
  - Polymorphisme : Fonctions
  - Itérateurs : énumèrent les éléments de la collection auxquels s'applique l' α-extension

6

Opérateurs parallèles

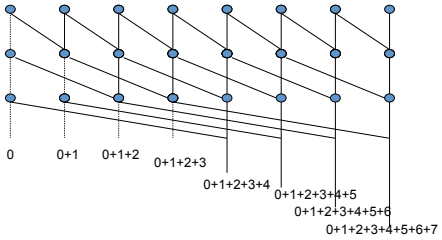
- Réduction
- Préfixe parallèle

7

Préfixe parallèle

```
pour j := 0 to lg(n-1)
  pour i := 2^j to n-1 (en parallèle)
    s[i] := f(s[i-2^j], s[i])
```

f est associative et commutative



8

Opérateurs spécifiques

- Point de vue collection pur
  - insertion, suppression
  - Listes : tri
  - Dictionnaires : accès par clé
- Tableaux : permutation, combinaison
  - Gather :
    B = gather (A,L) : pour tout i, B(i) = A(L(i))
  - Scatter :
    B = scatter (A,L) : pour tout i, B(M(i)) = A(i)
    - Contraintes de conformité
    - Collision : sens si M(i) = M(j) ?
    - Si M est injective, trie A suivant M

9

Opérateurs spécifiques

- Tableaux : sélection
  T(a:b:c) est le tableau des
  T(a + kc) pour 0 <= k <= (b-a)/c
- Et beaucoup d'autres primitives

10

Plan

- Introduction aux langages data parallèles
  - Principe
  - Collections
  - Opérateurs
- High Performance Fortran (HPF)
  - Itérateurs
  - Directives de placement
  - Introduction à la compilation
  - Extensions

11

Histoire

- Fortran 77 : séquentiel
- Fortran 90 : Vectoriel + encapsulation, polymorphisme
- Proposition de spécification réalisée par le HPF forum
  - HPF 1 : 94
  - HPF 2 : 97

12

L'instruction FORALL

- Syntaxe

  FORALL (*index-spec-list* [, *mask-expr*]) *forall-assignment*

  – *index-spec-list* : a1:b1:c1,a2:b2:c2, …, an:bn:cn
  – *forall-assignment* : affectation
- Sémantique : itérateur sur un sous-ensemble d'un pavé de Z^n
  – Evaluer l'ensemble des indices valides : ceux sur lesquels le masque est vrai
  – Pour tous les indices valides, évaluer le membre droit
  – Pour tous les indices valides, réaliser l'affectation

13

L'instruction FORALL

```
Forall (i=2:5:2, j=1:2)
  A(i,j) = A(i+2,j) + B(i, j)
```

synchronisation

```
do i=2:5:2
  do j= 1,2
    Tmp(i,j) = A(i+2,j) + B(i,j)
  end do
end do
do i=2:5:2
  do j= 1,2
    A(i,j) = tmp(i,j)
  end do
end do
```

14

Exemples

Affectation Forall (i=1:2,j=1:3) A(i,j) = i+j	Multidiffusion Forall (i=1:2, j=1:4) Y(i) = A(i,i)
Permutation Forall (i=2:5) X(i) = X(i-1) Forall (i=1:4) Y(i, L(i)) = Z(i) L = /1, 2, 2, 4/	Sélection Forall (i=1:3, X(i) != 0) Y(i) = 1/X(i)

15

Collisions

- En relation avec la vision espace d'adressage unique
- Les affectations multiples ne sont pas admises – le comportement est non-prédictible
- La correction ne peut pas être vérifiée par le compilateur
  - Incorrect : forall (i=1:n, j=1:n) A(i+j) = B(i,j)
  - Peut être correct : forall (i=1:n) A(L(i)) = B(i)
  - Toujours correct : forall (i=1:n) A(i) = B(L(i))

16

Relaxation de Jacobi

```
u = 0.0
unew = 0.0
err = tol + 1.0
iter = 0
DO WHILE (err > tol .and. iter < NITER)
  iter = iter + 1
  FORALL ( i=1:nx-1, j=1:ny-1 )
    unew(i,j) = (u(i-1,j)+u(i+1,j)+u(i,j-1)+u(i,j+1)+dx*dx*f(i,j)) / 4
  END FORALL
  err = MAXVAL( ABS(unew-u) )
  u = unew
END DO
```

17

Fonctions

- Quelles conditions doit vérifier la fonction FOO pour que l'appel

  Forall (i=1:n) A(i) = FOO(i)

  soit cohérent avec la sémantique du forall ?

- FOO ne doit pas produire d'effet de bord
  - Vérifié par le compilateur
  - Contraintes syntaxiques sur-restrictives
    - Pas de pointeurs
    - Pas de paramètres OUT

18

La construction FORALL

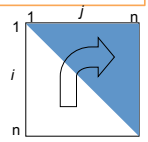
- Syntaxe
  FORALL (*index-spec-list* [, *mask-expr*])
  forall-body-list
  END FORALL
  - *forall-body* : *forall-assignment* ou FORALL ou WHERE
- Sémantique
  - Imbrication de forall : espace d'itération non rectangulaire
  - Exécution dans l'ordre syntaxique des instructions du *forall-body*

19

La construction FORALL

- Imbrication

```
Forall i=1:n
   Forall j= i+1:n
     A(i,j) = A(j,i)
   End Forall
End Forall
```



- Séquence

```
Forall i=2:4
   A(i) = A(i-1) + A(i+1)
   C(i) = B(i) + A(i+1)
End Forall
```

synchronisation

synchronisation

synchronisation

20

Placement des données

- Architecture matérielle cible : espaces d'adressages multiples
- Le placement des données détermine
  - Le placement des calculs : par exemple Owner Computes Rule
  - Donc la distribution de charge
  - Le volume et le nombre de communications : surcoût
- Choix HPF :
  - Placement explicite : le compromis localité/équilibrage de charge/surcoût est contrôlé par l'utilisateur
  - Par directives : pas un type – difficultés d'interprétation pour le passage des paramètres
  - Alignement + Distribution

21

Alignement

- Position relative des tableaux
  - Indépendante du nombre de processeurs
- Syntaxe

!HPF$ ALIGN array(*source-list*) WITH target (*subscript-list*)

  - *Subscript* est
    - Une fonction d'UNE variable de source-list
    - Un triplet a:b:c
    - *
- Sémantique
  - Les éléments alignés sont placés sur le même processeur
  - * en destination indice la réplication
  - * en source indique la séquentialisation

22

Alignement

Individuel

```
REAL A(3,3), B(3,3), X(2)
!HPF$ ALIGN A(i,j) WITH B(j,i)
!HPF$ ALIGN X(i) WITH A(i+1,2)
```

Séquentialisé

```
REAL A(3,3), Y(3)
!HPF$ ALIGN A(i,j) WITH Y(i)
Ou
!HPF$ ALIGN A(i,*) WITH Y(i)
```

Répliqué

```
REAL A(3,3), Y(3)
!HPF$ ALIGN Y(i) WITH A(i,*)
```

23

Distribution : la directive PROCESSORS

- Syntaxe

  !HPF$ PROCESSORS array-decl
- Sémantique

  Définit une géométrie rectangulaire de processeurs limitée par le nombre de processeurs physiquement disponibles
- Exemple A 64 processeurs

  !HPF$ PROCESSORS PC(4,4,4) ou PC(2,4,8) etc.

  !HPF$ PROCESSORS PP(8,8) ou P(4,16) etc.

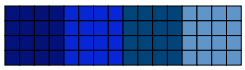
  !HPF$ PROCESSORS PL(64)

24

Distribution : la directive DISTRIBUTE

- Syntaxe
  !HPF$ DISTRIBUTE array (*dist-format-list*) [ONTO procs]
- Sémantique
  - Chaque dimension du tableau est distribuée suivant le schéma correspondant de dist-format-list
  - *dist-format* est
    - BLOCK : fragments contigus de taille égale
    - CYCLIC : circulaire
    - BLOCK(k) : fragments contigus de taille k
    - CYCLIC(k) : circulaire des blocks de taille k
    - * : séquentialisé
  - *procs* est un tableau de processeurs (calculé par le compilateur si absent)
- Parallélisation possible des calculs associés aux données distribuées sur des processeurs différents

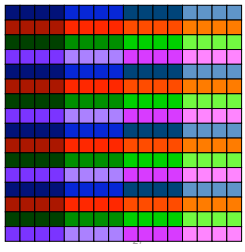
Distribution : la directive DISTRIBUTE

REAL A(16) !HPF$ PROCESSORS P(4) !HPF$ DISTRIBUTE A (*block)* 	REAL A(16) !HPF$ PROCESSORS P(4) !HPF$ DISTRIBUTE A (*cyclic)* 
REAL A(16) !HPF$ PROCESSORS P(4) !HPF$ DISTRIBUTE A (*cyclic(2))* 	REAL B(4,16) !HPF$ PROCESSORS P(4) !HPF$ DISTRIBUTE B(*,*block)* 

Distribution : la directive DISTRIBUTE

REAL B(16,16)
!HPF$ PROCESSORS P(4,4)
!HPF$ DISTRIBUTE B(cyclic,block*)*

Remarque



Alignement et distribution

- Chaque tableau a une cible d'alignement ultime
- Seules les cibles ultimes peuvent être distribuées
- Des templates peuvent être déclarés pour simplifier l'alignement

  !HPF$ TEMPLATE *array-decl*

28

Alignement et distribution

- Les distributions ne sont pas fermées vis-à-vis
  - De l'alignement
  - Des sections de tableaux

```
REAL A(12), B(6)
!HPF$ PROCESSORS P(4)
!HPF$ ALIGN B(i) WITH A(2*i –1)
!HPF$ DISTRIBUTE A(BLOCK) ONTO P
```

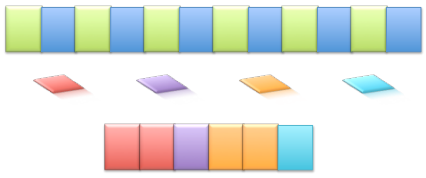
Ou A(1:12:2)

29

Alignement et distribution

Les distributions ne sont pas fermées vis-à-vis
- De l'alignement
- Des sections de tableaux

```
REAL A(12), B(6)
!HPF$ PROCESSORS P(4)
!HPF$ ALIGN B(i) WITH A(2*i –1)
!HPF$ DISTRIBUTE A(BLOCK) ONTO P
```



30

Conclusion

- Echec en tant que normalisation : n'a jamais été intégré dans le standard Fortran
- Concurrent OpenMP
- Explication socio-économique
  - L'essentiel des résultats pour une compilation efficace existe
  - Pas de marché
  - Problème de leadership
- Une niche au Japon, dans le programme Earth Simulator
  - Gordon Bell prize 2002 !

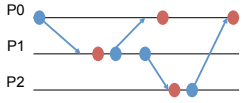
31

Programmation Parallèle

Passage de messages

Modèle de programmation à passage de messages

- Espaces d'adressage multiples
  - Toutes les données sont privées
  - Toutes les tâches peuvent communiquer
  - La communication et la synchronisation sont explicites
- La réalisation d'un modèle de calcul processus+ canaux
  - Identique : Messages typés-délimités et non flot comme en sockets SOCK_STREAM
  - Différence essentielle : pas de réception implicite



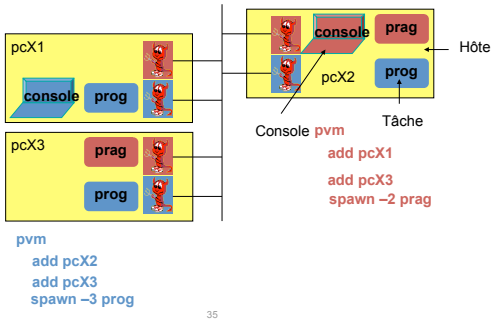
33

Parallel Virtual Machine

- Composants
  - Une librairie de passage de messages
  - Un environnement de gestion d'une machine parallèle virtuelle dynamique
- Une API et des implémentations
  - Domaine public : protocole sous-jacent TCP
  - Propriétaires : protocole sous-jacent propriétaire
  - Pour la plupart des machines parallèles et les grappes
- Histoire
  - Sous la direction de Jack Dongarra -> Top 500
  - PVM 1.0 : 1989, interne Oak Ridge National Laboratory
  - PVM 2.0 : 1991, University of Tennessee
  - PVM 3.4.5 : 2004

34

La machine virtuelle : version interactive



35

La machine virtuelle

- Ajout automatique de machine :
  - pvmd <hostfile>
  - Une machine par ligne
- L'ajout d'une machine crée les fichiers
  - /tmp/pvmd.<uid>
    - numéro de port
    - son existence interdit de relancer le démon
  - /tmp/pvml.<uid>
    - stdin et stdout des tâches lancées par la console

36

Création dynamique de tâches : pvm_spawn

int numt = pvm_spawn(char *task, char **argv, int flag, char *where, int ntask, int *tids

- task : nom de l'exécutable . Path par défaut $HOME/pvm3/bin/$PVM_ARCH/
- argv : arguments de l'exécutable
- flag : options de placement somme de
  - PvmTaskDefault 0 : quelconque
  - PvmTaskHost 1 : *where* spécifie une machine hôte
  - …
- ntask : Nombre de copies de l' exécutable à démarrer
- tids : Tableau[*ntask*] retourne les tids des processus PVM spawnés.
- numt Nombre de tâches spawnées.

37

Identification

int tid = pvm_mytid(void)

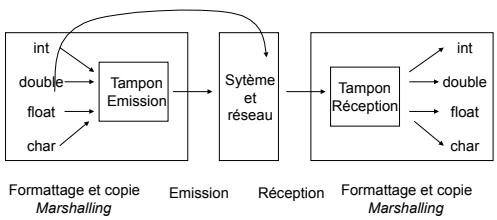
- Identifiant unique dans la MV
- « enrôle » la tâche dans la MV

int tid = pvm_parent(void)

- Identifiant du parent dans la MV
- Eventuellement PvmNoParent

38

Communication



39

Sérialisation

- Quel support pour la communication
  - de types complexes
  - d'objets pour les appels de méthodes à distance
  - pointeurs sous-jacents

40

Emission : fonctions de base

- Allocation d'un tampon
  Int bufid = pvm_initsend(int encoding)
  - Encoding : méthode de formattage des données
  - PvmDataInPlace : pas de copie, les données ne doivent pas être modifiées jusqu'au retour de l émission correspondante
- Formattage – sérialisation
  int info = pvm_pckxxx(<type> *p, int cnt, int std)
  - Limité aux sections de tableau de types primitifs
  - Un même tampon peut recevoir plusieurs pck sur des types différents
- Emission
  int info = pvm_send(int tid, int msgtag)
  - tid : identifiant destinataire
  - msgtag : un typage utilisateur
  - L'émission est non-bloquante : transfert vers l'agent de communication
  - info < 0 indique une erreur.
- Pour le formattage et l'émission, le tampon est implicite

41

Réception : fonctions de base

- Réception
  int bufid = pvm_recv(int tid, int msgtag)
  - tid : identifiant émetteur ; -1 = non spécifié
  - msgtag : un typage utilisateur; -1 = non spécifié
  - bloquante : attente sur un message émis par tid avec le tag msgtag.
- Formattage – désérialisation
  int info = pvm_upckxxx(<type> *p, int cnt, int std)
  - Copie dans le tableau p les données du buffer, avec le pas std

42

Réception

- Canaux FIFO
  - Si A envoie deux messages successifs de même tag à B, les réceptions s'effectuent dans l'ordre d'émission
  - Rien n'est garanti sur l'ordre de réception de messages provenant de processeurs différents
  - Le non déterminisme est possible
  - Le blocage en attente infinie est possible
  - Compromis
    - Programmation plus simple : Identifier les messages par tid et msgtag
    - Introduit des synchronisations inutiles

A	B	C
envoyer 2 à B	recevoir(-1)	envoyer 3 à B

Peut être 2 ou 3

Réceptions

- pvm_probe
  - Crée un tampon
  - >0 si un message est arrivé
  - Évite l'attente, ou permet de ne pas recevoir
- pvm_bufinfo
  - Information sur le message arrivé
- pvm_nrecv : réception non bloquante

```
tant que (pvm_probe (…) == -1)
    travail utile
Ftq
info = pvm_bufinfo (…)
flag = attendu(info)
traitement suivant flag
```

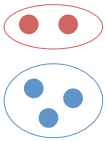
44

Méthodes d'optimisation

- Déséquilibre entre les performances de communication et celles de calcul
  - En latence : pénalise les accès isolés
  - En débit, à un degré moindre, mais irrécupérable
  - Rien de nouveau !
- Diminuer le nombre de communications
  - A décomposition constante : Vectorisation des communications
  - Modifier la décomposition : Vs équilibrage de charge
- Diminuer le volume de communication
  - Agrégation: vs simplicité
- Recouvrement communication/calcul
  - Si le surcoût de démarrage n'est pas dominant

Synchronisation

- Groupes de tâches
  - Syntaxe : voir group operations
  - Numéros unique séquentiels
  - Fonctions de conversion tid <-> inum
  - Seule géométrie prédéfine anneau
    - Conversion en géométrie nD triviale
    - En 2D : xCoord = inum/N, yCoord= inum%N



- Barrière

  int info = pvm_barrier (char*group, int count)
  - Toutes les tâches appelantes attendent que count tâches aient appelé la fonction

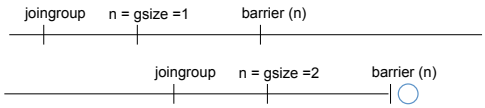
46

Synchronisation

- Un piège classique

  Size = pvm_gsize(mygroup)

  Info = pvm_barrier(gsize)
- Peut aboutir à une attente infinie



47

Passage de message vs Parallélisme de données

- Optimisation fine des communications
  - Recouvrement communication-calcul
  - Minimiser le volume et le nombre de communications
  - Distributions irrégulières des données
- Mais
  - Le parallélisme n'est pas explicite
  - Indéterminisme
    - Non reproductibilité des erreurs
    - Debug et maintenance difficiles
  - Code fastidieux

48

MPI

- Le standard de facto en librairies de communication
- Sérialisation de types complexes
- Géométries intégrées
- Uniquement une librairie de communication
  - Pas de création dynamique de processus
  - Toute la gestion de processus est reportée sur l'exécutif (environnement) : mpi-run
  - Pas de tolérance aux fautes. La réalisation de la tolérance aux fautes fait l'objet de nombreux projets.

http://www.mpi_forum.org

49