ENSERB 3ème Année Filière Informatique **Année 1999-2000**
D.E.A Informatique

Expression du Parallélisme

Durée 2h
Documents de cours autorisés

Exercice I

On considère le programme `HPF` suivant :

```
      PROGRAM ex_forall
      INTEGER, PARAMETER :: N = 100000
      INTEGER C(N,N), D(N,N)
      INTEGER I, J, s

!HPF$ ALIGN D WITH C
!HPF$ DISTRIBUTE C(BLOCK,BLOCK)
      ...

      s = N/2
      FORALL (I=1:N, J=1:N-s) C(I,J) = D(3, J+s)
      END
```

1) Décrire le placement des données de ce programme sur une grille de 4×4 processeurs.

2) Donner le principe général de génération du code SPMD utilisé par le compilateur `ADAPTOR` pour une instruction `FORALL`. Illustrer le pour l'instruction `FORALL` du programme `ex_forall` et détailler le en donnant le code intermédiaire généré par `ADAPTOR`.

3) Commenter le code SMPD généré par `ADAPTOR` (programme `cube.f`) pour le programme `HPF` précédent.
Détailler la traduction de l'instruction `FORALL` (en se basant sur le code intermédiaire fourni en **2)**).
Décrire les communications nécessaires dans le cas d'une grille 4×4 de processeurs (et en prenant $s = N/2$).

4) Plus généralement, décrire les communications nécessaires dans le cas d'une grille $P \times P$ de processeurs et s quelconque ($0 < s < N$).

5) On se place à présent dans le cadre d'une programmation parallèle par processus communiquant par transmission de messages, en utilisant `MPI`.

En utilisant les mêmes distributions de données que dans le programme `HPF`, écrire un code SPMD réalisant l'instruction `FORALL` du programme `HPF` (s quelconque tel que

$0 < s < N$) et qui implémente donc le schéma de communication entre processus qui a été mis en évidence dans la question **4)**.

Préciser les données locales des processus `MPI` et les primitives de communication `MPI` utilisées.

Exercice II

On s'interesse à la recherche d'une solution approchée de l'équation de Poisson $\Delta u = f$ en utilisant la méthode de Jacobi (sur une grille rectangulaire, avec f et les conditions aux limites donnés) :

$$u_{i,j}^{n+1} = 1/4 \times (u_{i+1,j}^{n} + u_{i-1,j}^{n} + u_{i,j+1}^{n} + u_{i,j-1}^{n} + f_{i,j})$$

Une programme incluant des directives `OpenMP` est donné (programme `poisson2` et module `mysubs`). Le programme séquentiel est obtenu en ignorant les directives `OpenMP`; les directives `OpenMP` permettent une parallélisation du programme sur machine à mémoire partagée.

Le code avec les directives `OpenMP` est incorrect (le code séquentiel est correct). Expliquer pourquoi (expliquer son fonctionnement et l'effet des directives).
Noter que les variables locales de la fonction `sweep` (variable `sweep` comprise), sont privées.

Proposer une correction.

Programme `cube.f`

```
      PROGRAM EX_FORALL
      INTEGER*4 N
      PARAMETER ( N = 100000 )
      INTEGER*4 C_DSP
      INTEGER*4 C (1:2)
      INTEGER*4 C_DIM2
      INTEGER*4 C_DIM1
      INTEGER*4 C_ZERO
      INTEGER*4 D_DSP
      INTEGER*4 D (1:2)
      INTEGER*4 D_DIM2
      INTEGER*4 D_DIM1
      INTEGER*4 D_ZERO
      INTEGER*4 I
      INTEGER*4 J
      INTEGER*4 S
      INTEGER*4 TMP1_DSP
      INTEGER*4 TMP1 (1:2)
      INTEGER*4 TMP1_DIM1
      INTEGER*4 TMP1_ZERO
      INTEGER*4 C_STOP1
      INTEGER*4 C_START1
      INTEGER*4 C_STOP2
      INTEGER*4 C_START2
      INTEGER*4 ISEC_DSP2
      INTEGER*4 dalib_0
      COMMON /dalib_data0/ dalib_0
      call dalib_init (4,4,4)
      call dalib_set_present (dalib_0)
      call dalib_start_subroutine ('SIMPLE',6)
      call dalib_array_make_dsp (C_DSP,2,4)
      call dalib_distribute (C_DSP,2,1,0,1,0)
      call dalib_array_define (C_DSP,1,N,1,N)
      call dalib_array_allocate (C_DSP,C,C_ZERO,C_DIM1,C_DIM2)
      call dalib_array_make_dsp (D_DSP,2,4)
      call dalib_align_source (D_DSP,C_DSP,4,
     $  1,0,1,4,2,0,1)
      call dalib_align_target (D_DSP,C_DSP,8,1,8,2)
      call dalib_array_define (D_DSP,1,N,1,N)
      call dalib_array_allocate (D_DSP,D,D_ZERO,D_DIM1,D_DIM2)
      S = N/2
      call dalib_array_make_dsp (TMP1_DSP,1,4)
      call dalib_align_source (TMP1_DSP,C_DSP,4,2,0,1)
      call dalib_align_target (TMP1_DSP,C_DSP,6,1,8,1)
      call dalib_array_define (TMP1_DSP,1,N-S)
```

```
 call dalib_array_allocate (TMP1_DSP,TMP1,TMP1_ZERO,TMP1_DIM1)
 call dalib_section_create (ISEC_DSP2,D_DSP,0,
$  3,0,0,1,1+S,N-S+S,1)
 call dalib_assign (TMP1_DSP,ISEC_DSP2)
 call dalib_section_free (ISEC_DSP2)
 call dalib_array_lslice (C_DSP,1,1,N,C_START1,C_STOP1)
 DO I=C_START1,C_STOP1
    call dalib_array_lslice (C_DSP,2,1,N-S,C_START2,C_STOP2)
    DO J=C_START2,C_STOP2
       C(C_ZERO+J*C_DIM1+I) = TMP1(TMP1_ZERO+J)
    END DO
 END DO
 call dalib_array_free (TMP1_DSP)
 call dalib_array_free (D_DSP)
 call dalib_array_free (C_DSP)
 call dalib_end_subroutine ()
 call dalib_exit ()
 END PROGRAM EX_FORALL
```

```
module mysubs

contains

  subroutine getsrc(q)
    use kinds
    implicit none

    real(kind=REAL8), dimension(:,:)  :: q
    integer  :: n
    integer  :: d1, d2
    integer  :: x1, x2, y1, y2

    !
    !
    !                              |         |
    !                    U = 1  -> |   D2    | <-  U = -1
    !          Y2  ----------------|         |-----------------
    !                                                      D1
    !
    !          Y1  ----------------|         |-----------------
    !                    U = -1 -> |         | <-  U = 1
    !                              |         |
    !             0                X1        X2
    !

    n  = size(q, 1)
    d1 = n/2
    d2 = n/4

    q = 0.0

    x1 = (n - d2)/2
    x2 = (n + d2)/2
    y1 = (n - d1)/2
    y2 = (n + d1)/2

    q(1:x1, y1) = -1.0
    q(1:x1, y2) =  1.0

    q(x2:n, y1) =  1.0
    q(x2:n, y2) = -1.0

    q(x1, 1:y1) = -1.0
    q(x2, 1:y1) =  1.0

    q(x1, y2:n) =  1.0
    q(x2, y2:n) = -1.0

    return
  end subroutine getsrc

  subroutine store(u)
    use kinds
    implicit none

    real(kind=REAL8), dimension(:,:)  :: u
    character(len=80), parameter      :: ofile = 'poisson.out'
    integer                           :: n, status, i, j

    n  = size(u, 1)
```

```
    open(1, file=ofile,iostat=status)
    if (status /= 0) then
       write (*,*) 'could not open file ', ofile
       stop
    end if
    write (1, *) n-2
    write (1,'(F10.5)') ((u(i,j), j=2, n-1), i=2, n-1)

    close(1)
    return
  end subroutine store

  function sweep(a, f, b)
    use kinds
    implicit none

    real(kind=REAL8)                  :: sweep
    real(kind=REAL8), dimension(:,:)  :: a, b, f
    real(kind=REAL8)                  :: resid
    integer                           :: n, i, j

    n  = size(a, 1)
    sweep = 0.0

    !$OMP DO
    do i = 2, n-1
       do j = 2, n-1
          resid = a(i-1,j) + a(i+1,j) + a(i,j+1) + a(i,j-1) + &
                  (-4.0)*a(i,j) - f(i,j)
          sweep = sweep + abs(resid)
          b(i,j) = a(i,j) + 0.25*resid
       end do
    end do
    !$OMP END DO

    return
  end function sweep

end module mysubs


program poisson2
  ! solve Poisson equation using Jacobi relaxation
  !! flawed !!

  use kinds
  use mysubs
  implicit none

  integer, parameter          :: n      = 200
  integer, parameter          :: maxits = 1000
  real(kind=REAL8), parameter :: eps    = 1.0e-4

  real(kind=REAL8), dimension(n,n) :: q, u1, u2
  real(kind=REAL8)                 :: error
  integer                          :: it

  call getsrc(q)        ! compute matrix of sources
  u1 = 0.0              ! initial solution
  u2 = 0.0
```

```
  !$OMP PARALLEL PRIVATE(error)
  do it = 1, maxits
     error = sweep(u1, q, u2)
     error = sweep(u2, q, u1)
     if (error/(n*n) < eps) exit
  end do

  !$OMP SINGLE
  print *, 'error after ', it, ' iterations: ', error/(n*n)
  !$OMP END SINGLE
  !$OMP END PARALLEL

  call store(u1)          ! store solution

  stop
end program poisson2
```

```
!
! portable data types
!

module kinds
  implicit none

  integer, parameter :: INT1  = selected_int_kind(2)   ! Single byte integer
  integer, parameter :: INT2  = selected_int_kind(4)   ! Two byte integer
  integer, parameter :: INT4  = selected_int_kind(9)   ! Four byte integer

  integer, parameter :: REAL4 = selected_real_kind(5)  ! Single precision
  integer, parameter :: REAL8 = selected_real_kind(12) ! Double precision

  integer, parameter :: COMPLEX8  = REAL4  ! complex kind from components
  integer, parameter :: COMPLEX16 = REAL8

end module kinds
```