Remarks on

Data Structures and Algorithms

Uwe Küster

grids for the solution of partial differential equations

structured grids

unstructured grids

quad-trees, oct-trees

structured grids

coordinates: real(kind=8) :: xx(imax , jmax , kmax , 3)

states: real(kind=8) ::state(imax , jmax , kmax , 5)

differences: real(kind=8) ::d(imax , jmax , kmax)

neighbourhood for difference operators:

state(i , j , k +1, 1)

state(i , j+1 , k , 1)

state(i-1 , j , k , 1) state(i , j , k , 1) state(i+1 , j , k , 1)

state(i , j-1 , k , 1)

state(i , j , k-1 , 1)

neighbourhood is implicitly defined by array structure

cheap and fast

structured grids: finite differences in nested loops

```
do k=2,kmax-1
  do j=2,jmax-1
    do i=2,imax-1
      dd( i , j , k ) = state( i , j +1 , k , 1 ) - state( i , j - 1 , k , 1 )
    enddo
  enddo
enddo
```

loops are relatively small

imax, jmax, kmax ~ 200 → grid size~ 8000000 points

structured grids: definition of an inner index

```
integer :: ijk(imax*jmax*kmax)
ll=0
do k=2,kmax-1
  do j=2,jmax-1
     do i=2,imax-1
       ll=ll+1

       ijk(ll) = i + imax*(j-1) + imax*jmax*(k-1)   ! index definition
     enddo
   enddo
enddo
inner_size=ll
```

to be done one time for the inner grid points

structured grids: unrolling the nested loops

```
do ll=1, inner_size

   dd( ijk(ll) ,1, 1) = state( ijk(ll)+jmax ,1,1,1) - state( ikj(ll) - jmax ,1,1,1)
enddo
```

loop is as large as possible

loop is vectorizable and parallelizable

much faster on small sized imax, jmax, kmax for vector computer

index vector is reused

some overhead in memory in bandwidth

structured grids: finite differences in nested loops

```
real  :: dd(imax,jmax),a(imax,jmax,2),state(imax,jmax)
do j=2,jmax-1
      do i=2,imax-1
        dd( i , j ) = al i ,  j , 1)*state( i ,  j +1 ) - a( i , j , 2)*state( i ,  j - 1 )
      enddo
enddo
```

equivalent to

```
do i j=1+imax,imax*(jmax-1)
dd( i j, 1) = al i j, 1 , 1)*state( i j +imax,1 ) - a( i j, 1 , 2)*state( i j - imax , 1 )
enddo
```

a(i, j,1:2) = 0 on the boundaries

state(i , j+1) = state(i+imax*(j-1) + imax)

unstructured grids

triangles and tetraedras

handling self adaptive refinement

handling arbitrary geometrical shapes

neighbourhood must be described explicitly

Triangle with integer references: data strucure

```
type state_type
   real(kind=real_kind),dimension(3)       :: var
end type state_type

type triangle_type
   type(state_type)                        :: state
   integer,dimension(3)                    :: neighbour
end type triangle_type
```


Triangle with integer references: collecting loop

```
type(triangle_type),dimension(mmax)        :: triangles

do mm=1,mmax

   mm_1=triangles(mm)%neighbour(1)
   mm_2=triangles(mm)%neighbour(2)
   mm_3=triangles(mm)%neighbour(3)

   triangles(mm)%state%var(1)=1._real_kind / 3._real_kind &
                              & *(&
                              & +triangles(mm_1)%state%var(2) &
                              & +triangles(mm_2)%state%var(2) &
                              & +triangles(mm_3)%state%var(2) &
                              & )
enddo
```

Triangle with pointer references: data structure

```
type state_type
  real(kind=real_kind),dimension(3)      :: var
end type state_type

type triangle_type
  type(state_type)                        :: state
  type(triangle_type),pointer             :: neighbour_1
  type(triangle_type),pointer             :: neighbour_2
  type(triangle_type),pointer             :: neighbour_3
end type triangle_type
```


Triangle with pointer references: collecting loop

```
do mm=1,mmax

  triangles(mm)%state%var(1)=1._real_kind/3._real_kind &
            & *(&
            & +triangles(mm)%neighbour_1%state%var(2) &
            & +triangles(mm)%neighbour_2%state%var(2) &
            & +triangles(mm)%neighbour_3%state%var(2) &
            & )
enddo
```

integer or pointer references

- Fortran 90/95:
  most compilers generate
  faster code for integer references
  not IBM xlf90

- C,C++:
  pointer references are natural
  a[n] syntax is very near to pointers

sparse matrices: usage

conjugate gradient and Krylov space algorithms

direct LU and Cholesky decomposition

formulation of neighbourhoods or graphs

sparse matrices: row ordered example

row_number	
0	5
1	11
2	7
3	8
4	2
5	4
6	16
7	9
8	12

begin	
0	0
1	3
2	5
3	8
4	11
5	13
6	16
7	19
8	21

index		
0 11	1 7	2 8
3 4	4 5	
5 5	6 16	7 8
8 5	9 7	10 2
11 8	12 12	
13 11	14 16	15 9
16 4	17 12	18 7
19 4	20 12	
21 2	22 16	23 9

end	
0	2
1	4
2	7
3	10
4	12
5	15
6	18
7	20
8	23

index arrays beginning with 0

sparse matrix example

current_number	element	indices		
0	5	11	8	7
1	11	4	5	
2	7	5	16	8
3	8	5	7	2
4	2	8	12	
5	4	11	16	9
6	16	4	12	7
7	9	4	12	
8	12	2	16	9

matrix values are omitted, real array analogous to integer indices array

sparse matrices: jagged diagonal example

row_number

0	5
1	7
2	8
3	4
4	16
5	12
6	11
7	2
8	9

begin

0	0
1	9
2	18

index

0	11	9	7	18	8
1	5	10	16	19	8
2	5	11	7	20	2
3	11	12	16	21	9
4	4	13	12	22	7
5	2	14	16	23	9
6	4	15	5		
7	8	16	12		
8	4	17	12		

end

0	8
1	17
2	23

index arrays beginning with 0

row ordered and jagged diagonal matrices: data structure

```
type sparse_matrix_type

logical                        :: fleeting                 ! .true. if function result
integer                        :: type_of_relation         !
integer                        :: number_of_rows           ! dim of row_number
integer                        :: number_of_indices        ! dim of index, values
integer                        :: maximal_neighbourhood    ! max number cols
integer,pointer,dimension(:)   :: begin                    ! offset of row, pseudo col
integer,pointer,dimension(:)   :: length                   ! length row, pseudo col
integer,pointer,dimension(:)   :: index                    ! index array
integer,pointer,dimension(:)   :: row_number               ! row_number
integer,pointer,dimension(:)   :: value                    ! matrix values

end type sparse_matrix_type
```

row ordered matrix: matrix vector multiplication

```
subroutine  matrix_mult_vector_row_ord(matrix,vector,result_vector)
type(sparse_matrix_type)                                  :: matrix
type(vector_type)                                         :: vector, result_vector
real(kind=real_kind),dimension(matrix%number_of_rows)  :: temp
integer                                                   :: cn, no, pseudo_col, index

     temp=0.
do cn=1,matrix%number_of_rows
          pseudo_col=matrix%begin(cn)
  do no=1,matrix%length(cn)
       index=              matrix%index(no + pseudo_col)
   temp(cn)=temp(cn)+matrix%value(no + pseudo_col)*vector%value(index)
  enddo
enddo
  result_vector%value(matrix%row_number)=temp
end subroutine  matrix_mult_vector_row_ord
```


jagged diagonal matrix: matrix vector multiplication

```
subroutine  matrix_mult_vector_jaggdiag(matrix,vector,result_vector)
type(sparse_matrix_type)                                  :: matrix
type(vector_type)                                         :: vector, result_vector
real(kind=real_kind),dimension(matrix%number_of_rows)  :: temp
integer                                                   :: cn, no, pseudo_col, index

     temp=0.
do no=1,matrix%maximal_neighbourhood
     pseudo_col=matrix%begin(no)
  do cn=1,matrix%length(no)
          index=              matrix%index(cn+ pseudo_col)
     temp(cn)=temp(cn)+matrix%value(cn+ pseudo_col)*vector%value(index)
  enddo
enddo
  result_vector%value(matrix%row_number)=temp
end subroutine  matrix_mult_vector_jaggdiag
```

row ordered and jagged diagonal matrix formulation: performance

- row ordered
  - short loops
  - better on cache machines
  - simpler to use
  - redistribution not necessary in many cases
- jagged diagonal
  - long loops
  - better on vector machines
  - redistribution necessary
- combination of both
  - very flexibel
  - sophisticated