Workshop On Scientific Problem Solving In Fortran 90/95 — Multigrid Code

Spring, 1999
Richard C. Allen, SNL
Paul M. Alsing, UNM/AHPCC
Andrew C. Pineda, UNM/AHPCC
Brian T. Smith, UNM/AHPCC/CS
Topics To Be Covered

- Description of the multigrid software
  - representing grids
  - computational modules
    - grid definitions
    - computational routines
    - data initialization for the specific problems
  - package structure and motivation
    - data abstraction
    - computation flexibility with the methods
    - iteration versus recursion
From the multigrid theory, for each grid,

- need only the interval width
- the number of points in the interval
  - the number of points is actually the extent of the solution vector and so need no explicitly appear in the grid node definition
- the current solution on the grid
- the current right hand side
  - the driving function $f$ on the finest grid
  - the restricted residual on all other grids
- pointers to the finer and coarser grids
**Definition Of A Grid Node**

The definition of a grid node `mesh`:

```fortran
  type mesh
    private
    type (mesh), pointer :: finer
    real h
    real, pointer, dimension(:) :: solution, f
    type (mesh), pointer :: coarser
  end type mesh
```
Subtleties Of The Grid Representation

Consider the following example

finest grid is 49 points, \( N = 48 \)

- the solution need only be computed for the interior points, actually only 47 points
  - the reason is that the solution is known to be equal to the boundary conditions at the end points for all iterations

Thus, the size of the solution and the driving function \( f \) (or residual) is 47

- if node is one of the grid nodes, then:
  - \( \text{size( head\%f )} \) is thus 47
Grid Definitions — Creating A Linked Grid

First create the head node with:

- the interval width for the finest grid
- initial guess at the solution on the finest grid
- the values of the driving function
- pointers to the finer and coarser grids
  - no finer grid so point this one to NULL
  - currently no coarser grid so point it to NULL

Next create grid nodes linked to each other and the head with:

- initial values of interval width, solutions, and residuals
subroutine create_head( h, finest, solution, finest_node, fcn )

    implicit none
    real h
    type( mesh ), pointer :: finest_node
    real, dimension(:) :: finest, solution
    interface
        function fcn(x)
            implicit none
            real, dimension(:), intent(in) :: x
            real, dimension(size(x)) :: fcn
        end function fcn
    end interface
    allocate( finest_node )
    allocate( finest_node%solution(size(finest)-1) )
    allocate( finest_node%f(size(finest)-1) )

    finest_node%h = h
    finest_node%solution = solution
    finest_node%f = h*h * fcn(finest)

    nullify( finest_node%finer )
    nullify( finest_node%coarser )

end subroutine create_head
subroutine create_double_linked_grids( depth, head )

    implicit none
    integer, intent(in) :: depth
    type (mesh), pointer :: head

    type (mesh), pointer :: current_node
    real, dimension(size(head%f)) :: f
    real h
    integer i, n, nw

    n = size(head%f)
    h = head%h
    f = head%f
    current_node => head

    do i = 1, depth
        if( n < 3 )  exit

        nw = n/2
        h = h/2
        f(1:nw) = f(2:n-1:2)

        n = nw
        current_node%coarser => set_node( f(1:n), h, current_node )
        current_node => current_node%coarser

    enddo

end subroutine  create_double_linked_grids
Computational Procedures

The computations are divided into:

- relax on a grid
  - compute the approximate solution to $Ax=b$ for appropriate $b$ and initial guess $x$
    - iterate a specified number of times
    - possible methods are: weighted Jacobi, Gauss-Seidel, Red-Black Gauss-Seidel, …
- compute residual on the finest grid
- restrict residual to a coarser grid
- correct solution on a finer grid with a residual estimated from a coarser grid
  - inject and correct solution
**Weighted Jacobi Method**

This method uses the following iteration:

\[ v^{(1)} = ((1-w)I + wD^{-1}(L+U))v^{(0)} + wD^{-1}f \]

where D is the diagonal matrix of 2s  
L is a subdiagonal of -1s  
U is a superdiagonal of -1s  
w is a weighting factor, w=0.66  

to solve \( (D-L-U)x = f \)
subroutine weighted_jacobi( no_iters, alpha, solution, rhs )

    implicit none
    integer, intent(in) :: no_iters
    real, dimension(:), intent(inout) :: solution
    real, dimension(:), intent(in) :: alpha, rhs

    integer k, n
    real  save_1, save_nm1, weightc
    intrinsic  abs, maxval, size

    weightc = 1.0 - weight
    n = size(solution)

    do k = 1, no_iters

        save_nm1 = solution(n-1)
        save_1 = weightc*solution(1) + weight*(rhs(1) + solution(2))/alpha(1)

        solution(2:n-1) = weightc*solution(2:n-1) + weight*(rhs(2:n-1) +
                        solution(1:n-2) + solution(3:n))/alpha(2:n-1)

        solution(1) = save_1
        solution(n) = weightc*solution(n) + weight*(rhs(n) + save_nm1)/alpha(n)

    enddo

end subroutine weighted_jacobi
This method uses the following iteration:
subroutine gauss_seidel( no_iters, alpha, solution, rhs )

implicit none
integer, intent(in) :: no_iters
real, dimension(:), intent(inout) :: solution
real, dimension(:), intent(in) :: alpha, rhs

integer i, k, n
real vj
intrinsic abs, maxval, size

! NOTE -- this code is assuming zero boundary conditions.

n = size(solution)

do k = 1, no_iters
  vj = solution(2)
  solution(1) = (rhs(1) + vj)/alpha(1)
  do i = 2, n-1
    vj = solution(i-1) + solution(i+1)
    solution(i) = (rhs(i) + vj)/alpha(i)
  enddo
  vj = solution(n-1)
  solution(n) = (rhs(n) + vj)/alpha(n)
endo
Red-Black Gauss-Seidel Method

This method uses the following iteration:
subroutine red_black( no_iters, alpha, solution, rhs )

implicit none
integer, intent(in) :: no_iters
real, dimension(:), intent(inout) :: solution
real, dimension(:), intent(in) :: alpha, rhs

integer k, n
logical neven
intrinsic mod, size

n = size(solution)
neven = mod(n,2) == 0

do k = 1, no_iters

  solution(2:n-1:2) = (rhs(2:n-1:2) + solution(1:n-2:2) + &
                      solution(3:n:2))/alpha(2:n-1:2)

  if( neven )  solution(n) = (rhs(n) + solution(n-1))/alpha(n)
  solution(1) = (rhs(1) + solution(2))/alpha(1)

  solution(3:n-1:2) = (rhs(3:n-1:2) + solution(2:n-2:2) + &
                        solution(4:n:2))/alpha(3:n-1:2)

  if( .not. neven )  solution(n) = (rhs(n) + solution(n-1))/alpha(n)

endo

end subroutine red_black
The residual on a fine grid must be transferred to the coarser grid:

- this is a transformation from a fine grid space to a coarse grid space
- for this implementation, we are assuming the coarser grid is a proper subset of the finer grid
  - actually all the odd grid points of the fine grid represent the coarse grid

This is done with a full weighting operator:

- the operator maps from fine space to coarse space
  - it has roughly half as many rows as columns
Full Weighting Operator Continued

The operator \(((N/2-1) \times (N-1))\) is:

\[
\begin{pmatrix}
c_1 \\
c_2 \\
\vdots \\
c_{N/2-1}
\end{pmatrix}
= \frac{1}{4}
\begin{pmatrix}
1 & 2 & 1 & 0 & \cdots \\
1 & 2 & 1 & 0 & \cdots \\
\vdots & & & & \\
1 & 2 & 1 \\
\end{pmatrix}
\begin{pmatrix}
f_1 \\
f_2 \\
f_3 \\
\vdots \\
f_{N-1}
\end{pmatrix}
\]
subroutine compute_residual( alpha, fsol, frhs, fres )

! This procedure computes the residual fres = frhs - A*fsol.

implicit none
real, dimension(:), intent(in) :: alpha, fsol, frhs
real, dimension(:), intent(out) :: fres

integer fsz, i
real vj

fsz = size(frhs)

vj = alpha(1)*fsol(1) - fsol(2)
frhs(1) = fres(1) = frhs(1) - vj

do i = 2, fsz-1
  vj = alpha(i)*fsol(i) - fsol(i-1) - fsol(i+1)
  fres(i) = frhs(i) - vj
enddo

vj = alpha(fsz)*fsol(fsz) - fsol(fsz-1)
frhs(fsz) = fres(fsz) = frhs(fsz) - vj

end subroutine compute_residual
Code For Restricting To The Coarser Grid

subroutine restrict_res_to_coarser_grid( finer, coarser )

use problem_data_module
implicit none
type( mesh ), pointer :: coarser, finer

intrinsic size
real, dimension(:), pointer :: alpha, crhs, frhs, fsol
real, dimension(:), allocatable :: fres
integer csz, fsz

crhs => get_rhs( coarser )
frhs => get_rhs( finer )
fsol => get_solution( finer )
alpha => get_interval_width( finer )
csz = size( crhs )
fsz = size( frhs )
allocate( fres(fsz) )
alpha = 2.0 + alpha * problem_sigma

call compute_residual( alpha, fsol, frhs, fres )
crhs(1:fsz/2) = 0.25*(fres(1:fsz-2:2) + 2.0*fres(2:fsz-1:2) + fres(3:fsz:2))
deallocate( alpha, fres )

end subroutine restrict_res_to_coarser_grid
The residual on a coarse grid is transferred to the coarser grid and then used to correct the estimated solution on the fine grid.

- this is a transformation from a coarse grid space to a fine grid space
- for this implementation, we are assuming the coarser grid is a proper subset of the finer grid
  - actually all the odd grid points of the fine grid represent the coarse grid

This is done with interpolation operator:

- the operator maps from coarse space to fine space
  - it has roughly twice as many rows as columns
  - recall it is proportional to the transpose of the full weighting operator
The operator is:

\[
\begin{pmatrix}
  f_1 \\
  f_2 \\
  f_3 \\
  \vdots \\
  f_{N-1}
\end{pmatrix}
= \frac{1}{2}
\begin{pmatrix}
  1 & 0 & \cdots \\
  2 & 0 & \cdots \\
  1 & 1 & 0 & \cdots \\
  \vdots & \ddots & \ddots & \ddots \\
  1 & 1 & 0 & \cdots & 1
\end{pmatrix}
\begin{pmatrix}
  c_1 \\
  c_2 \\
  \vdots \\
  c_{N/2 - 1}
\end{pmatrix}
\]
subroutine correct_to_finer_grid( coarser, finer )

  implicit none
  type( mesh ), pointer :: coarser, finer

  real, dimension(:,), pointer :: csol, fsol
  real, parameter :: SCALE = 4.0
  integer csz, fsz
  intrinsic size

  csol => get_solution( coarser )
  fsol => get_solution( finer )
  csz = size( csol )
  fsz = size( fsol )

  fsol(1) = fsol(1) + 0.5*SCALE*csol(1)
  fsol(3:2*csz-1:2) = fsol(3:2*csz-1:2) + 0.5*SCALE*(csol(1:csz-1) + csol(2:csz))
  fsol(2:2*csz:2) = fsol(2:2*csz:2) + SCALE*csol(1:csz)
  fsol(fsz) = fsol(fsz) + 0.5*SCALE*csol(csz)

  csol = 0.0

end subroutine correct_to_finer_grid
The operator on the coarser grid is:

The bottom line is that the operator on the coarse grid is a scaling by 1/4 of the same matrix reduced in dimension by roughly 2.
V-Cycle Using Loops

subroutine compute_solution_using_V_cycle( no_iters_to_c, no_iters_to_f, &
current_grid, iterative_solve )

implicit none
integer, intent(in) :: no_iters_to_c, no_iters_to_f
type( mesh ), pointer :: current_grid
interface
  subroutine iterative_solve( no, alpha, solution, rhs )
    integer, intent(in) :: no
    real, dimension(:), intent(in) :: alpha, rhs
    real, dimension(:), intent(inout) :: solution
  end subroutine iterative_solve
end interface

! Local variables.
type( mesh ), pointer :: last_grid

! Calculate the solution on the finest mesh.
call relax_on_a_grid( no_iters_to_f, current_grid, iterative_solve )

! Calculate the solution on successively coarser meshes down to the
! coarsest mesh.

...  GOING DOWN THE V

! Calculate the solution on successively finer meshes up to the finest
! mesh.

...  GOING UP THE V

end subroutine compute_solution_using_V_cycle
Last_grid => current_grid
current_grid => get_coarser_node( current_grid )

do while( associated( current_grid ) )

! Compute the residual on the last grid and project the residual
! onto the current grid.

call restrict_res_to_coarser_grid( last_grid, current_grid )

! Calculate and improve solution on the coarser grid.

call relax_on_a_grid( no_iters_to_c, current_grid, iterative_solve )

! Move to the next coarser grid, if there is one.

last_grid => current_grid
current_grid => get_coarser_node( current_grid )

enddo
current_grid => get_finer_node( last_grid )

do while( associated( current_grid ) )

! Correct the approximate solution on the finer grid to obtain an initial
! solution for the next round of relaxations.

call correct_to_finer_grid( last_grid, current_grid )

! Calculate and improve solution on the finer grid.

call relax_on_a_grid( no_iters_to_f, current_grid, iterative_solve )

! Move to the next finer grid, if there is one.

last_grid => current_grid
current_grid => get_finer_node( current_grid )

enddo
Exercises

- Add a zero index and SZ index to the arrays
  - alpha, solution, rhs, f, ...
  - place the boundary conditions in these elements
  - fix the computational loops so that they use these boundary conditions

- Implement compute_using_V_cycle as a recursive procedure
  - change the loops to recursive calls