A Prototype Finite Difference Model

Timothy H. Kaiser, Ph.D.
tkaiser@mines.edu
A Prototype Model

- We will introduce a finite difference model that will serve to demonstrate what a computational scientist needs to do to take advantage of Distributed Memory computers using MPI.
- The model we are using is a two dimensional solution to a model problem for Ocean Circulation, the Stommel Model.
- Examples
  - geco.mines.edu/workshop
The Stommel Problem

- Wind-driven circulation in a homogeneous rectangular ocean under the influence of surface winds, linearized bottom friction, flat bottom and Coriolis force.

- Solution: intense crowding of streamlines towards the western boundary caused by the variation of the Coriolis parameter with latitude.
Governing Equations Model Constants

\[ \psi = 0 \]

\[ \gamma \left( \frac{\partial^2 \psi}{\partial x^2} + \frac{\partial^2 \psi}{\partial y^2} \right) + \beta \frac{\partial \psi}{\partial x} = f \]

\[ f = -\alpha \sin \left( \frac{\pi y}{2L_y} \right) \]

\[ L_x = L_y = 2000 \text{Km} \]

\[ \gamma = 3 \times 10^{-6} \]

\[ \beta = 2.25 \times 10^{-11} \]

\[ \alpha = 10^{-9} \]
The steady state solution
Domain Discretization

Define a grid consisting of points \((x_i, y_j)\) given by

\[
x_i = i\Delta x, \quad i = 0, 1, \ldots, nx + 1
\]
\[
y_j = j\Delta y, \quad j = 0, 1, \ldots, ny + 1
\]

\[
\Delta x = \frac{L_x}{nx + 1}
\]
\[
\Delta y = \frac{L_y}{ny + 1}
\]
Domain Discretization

Seek to find an approximate solution

\[ \psi(x_i, y_j) \text{ at points } (x_i, y_j): \]
\[ \psi_{i,j} \approx \psi(x_i, y_j) \]
Centered Finite Difference Scheme for the Derivative Operators

\[ \frac{\partial \psi}{\partial x} \approx \frac{\psi_{i+1,j} - \psi_{i-1,j}}{2\Delta x} \]

\[ \frac{\partial^2 \psi}{\partial x^2} \approx \frac{\psi_{i+1,j} - 2\psi_{i,j} + \psi_{i-1,j}}{(\Delta x)^2} \]

\[ \frac{\partial^2 \psi}{\partial y^2} \approx \frac{\psi_{i,j+1} - 2\psi_{i,j} + \psi_{i,j-1}}{(\Delta y)^2} \]
Governing Equation
Finite Difference Form

\[
\psi_{i,j} = a_1 \psi_{i+1,j} + a_2 \psi_{i-1,j} + a_3 \psi_{i,j+1} + a_4 \psi_{i,j-1} - a_5 f_{i,j}
\]

\[
a_1 = \frac{\Delta y^2}{2(\Delta x^2 + \Delta y^2)} + \frac{\beta \Delta x^2 \Delta y^2}{4 \gamma \Delta x(\Delta x^2 + \Delta y^2)}
\]

\[
a_2 = \frac{\Delta y^2}{2(\Delta x^2 + \Delta y^2)} - \frac{\beta \Delta x^2 \Delta y^2}{4 \gamma \Delta x(\Delta x^2 + \Delta y^2)}
\]

\[
a_3 = \frac{\Delta x^2}{2(\Delta x^2 + \Delta y^2)}
\]

\[
a_4 = \frac{\Delta x^2}{2(\Delta x^2 + \Delta y^2)}
\]

\[
a_5 = \frac{\Delta x^2 \Delta y^2}{2 \gamma (\Delta x^2 + \Delta y^2)}
\]
Five-point Stencil Approximation

\[ \psi_{i,j} = a_1 \psi_{i+1,j} + a_2 \psi_{i-1,j} + a_3 \psi_{i,j+1} + a_4 \psi_{i,j-1} - a_5 f_{i,j} \]

\[ \psi_{i,0} = \psi_{i,n_y+1} = 0; \quad \psi_{0,j} = \psi_{nx+1, j} = 0; \]
Jacobi Iteration

Start with an initial guess for \( \psi_{i,j} \)

Repeat the process

\[
\text{do } i = 1, \text{nx}; j = 1, \text{ny} \\
\psi_{i,j}_{\text{new}} = a_1\psi_{i+1,j} + a_2\psi_{i-1,j} + a_3\psi_{i,j+1} + a_4\psi_{i,j-1} - a_5f_{i,j} \\
\psi_{i,j} = \psi_{i,j}_{\text{new}} \\
\text{end do}
\]
A Prototype Finite Difference Model (Philosophy)
Overview

- Model written in Fortran 90
- Uses many new features of F90
  - Free format
  - Modules instead of commons
  - Module with kind precision facility
  - Interfaces
  - Allocatable arrays
Free Format

- Statements can begin in any column
- ! Starts a comment
- To continue a line use a “&” on the line to be continued
Modules instead of commons

- Modules have a name and can be used in place of named commons
- Modules are defined outside of other subroutines
- To “include” the variables from a module in a routine you “use” it
- The main routine stommel and subroutine jacobi share the variables in module “constants”

```fortran
module constants
    real dx, dy, a1, a2, a3, a4, a5, a6
end module

program stommel
    use constants
    ...                  subroutine jacobi
    use constants      use constants
    ...                 ...                 end subroutine jacobi
end program
```

Kind precision facility

Instead of declaring variables

\[
\text{real*8 } x, y
\]

We use

\[
\text{real(b8) } x, y
\]

Where b8 is a constant defined within a module

```fortran
module numz
    integer, parameter :: b8 = selected_real_kind(14)
end module
```

```fortran
program stommel
    use numz
    real(b8) x, y
    x = 1.0_b8
    ...
```
Kind precision facility Why?

Legality
Portability
Reproducibility
Modifiability

Declaring variables “double precision” will give us 16 byte reals on some machines

\[
\text{integer, parameter :: b8 = selected_real_kind(14)}
\]
\[
\text{real(b8) x, y}
\]
\[
\text{x = 1.0_b8}
\]
Allocatable arrays

- We can declare arrays to be allocatable
- Allows dynamic memory allocation
- Define the size of arrays at run time

```fortran
real(b8), allocatable:: psi(:, :) ! our calculation grid
real(b8), allocatable:: new_psi(:, :) ! temp storage for the grid

allocate(psi(0:nx+1, 0:ny+1))
allocate(new_psi(0:nx+1, 0:ny+1))
```

! allocate the grid to size nx * ny plus the boundary cells
Interfaces

• Similar to C prototypes
• Can be part of the routines or put in a module
• Provides information to the compiler for optimization
• Allows type checking

module face
    interface bc
        subroutine bc (psi,i1,i2,j1,j2)
            use numz
            real(b8),dimension(i1:i2,j1:j2):: psi
            integer,intent(in):: i1,i2,j1,j2
        end subroutine
    end interface
end module

program stommel
    use face
    ...

Array Syntax

Allows assignments of arrays without do loops

! allocate the grid to size nx * ny plus the boundary cells
allocate(psi(0:nx+1,0:ny+1))
allocate(new_psi(0:nx+1,0:ny+1))

! set initial guess for the value of the grid
psi=1.0_b8

! copy from temp to main grid
psi(i1:i2,j1:j2)=new_psi(i1:i2,j1:j2)
Program Outline (1)

- Module NUMZ - defines the basic real type as 8 bytes
- Module INPUT - contains the inputs
  - nx, ny (Number of cells in the grid)
  - lx, ly (Physical size of the grid)
  - alpha, beta, gamma (Input calculation constants)
  - steps (Number of Jacobi iterations)
- Module Constants - contains the invariants of the calculation
Program Outline (2)

- Module face - contains the interfaces for the subroutines
  - bc - boundary conditions
  - do_jacobi - Jacobi iterations
  - force - right hand side of the differential equation
  - Write_grid - writes the grid
Program Outline (3)

- Main Program
  - Get the input
  - Allocate the grid to size nx * ny plus the boundary cells
  - Calculate the constants for the calculations
  - Set initial guess for the value of the grid
  - Set boundary conditions using
  - Do the jacobi iterations
  - Write out the final grid
C version considerations

- To simulate the F90 numerical precision facility we:
  - `#define FLT double`
- And use FLT as our real data type throughout the rest of the program
- We desire flexibility in defining our arrays and matrices
  - Arbitrary starting indices
  - Contiguous blocks of memory for 2d arrays
- Use routines based on *Numerical Recipes in C*
Vector allocation routine

FLT *vector(INT nl, INT nh)
{
/* creates a vector with bounds vector[nl:nh] */
    FLT *v;
/* allocate the space */
    v=(FLT *)malloc((unsigned) (nh-nl+1)*sizeof(FLT));
    if (!v) {
        printf("allocation failure in vector()\n");
        exit(1);
    }
/* return a value offset by nl */
    return v-nl;
}
Matrix allocation routine

FLT **matrix(INT nrl,INT nrh,INT ncl,INT nch)
/* creates a matrix with bounds matrix[nrl:nrh][ncl:nch] */
/* modified from the book version to return contiguous space */
{
    INT i;
    FLT **m;
    /* allocate an array of pointers */
    m=(FLT **) malloc((unsigned) (nrh-nrl+1)*sizeof(FLT*));
    if (!m){
        printf("allocation failure 1 in matrix()"); exit(1);}
    m -= nrl; /* offset the array of pointers by nrl */
    for(i=nrl;i<=nrh;i++) {
        if(i == nrl){
            /* allocate a contiguous block of memroy*/
            m[i]=(FLT *) malloc((unsigned) (nrh-nrl+1)*(nch-ncl+1)*sizeof(FLT));
            if (!m[i]){  
                printf("allocation failure 2 in matrix()"); exit(1); }
            m[i] -= ncl; /* first pointer points to beginning of the block */
        }  
        else {
            m[i]=m[i-1]+(nch-ncl+1); /* rest of pointers are offset by stride */
        }
    }
    return m;
}
Digression... a 3d Volume allocation routine

Same idea but we allocate an array of slices

```c
FLT ***cube(INT nslice1, INT nslice2, INT nrow1, INT nrow2, INT ncol1, INT ncol2) {
    FLT **slice(INT nrow1, INT nrow2, INT ncol1, INT ncol2, FLT **temp);
    FLT *temp,**mcube;
    INT i;
    mcube=(FLT ***) malloc((unsigned) (nslice2-nslice1+1)*sizeof(FLT*));
    if (!mcube){
        printf("allocation failure at 1 in cube()\n");
        return NULL;
    }
    mcube -= nslice1;
    temp=(FLT*) malloc((unsigned) (nslice2-nslice1+1)*
                      (nrow2-nrow1+1)*
                      (ncol2-ncol1+1)*sizeof(FLT));
    if (*temp){
        printf("allocation failure at 2 in cube()\n");
        return NULL;
    }
    for(i=nslice1;i<=nslice2;i++) {
        mcube[i]=slice(nrow1, nrow2, ncol1, ncol2,&temp);
        if(!mcube[i]) return NULL;
        temp += (nrow2-nrow1+1)*(ncol2-ncol1+1);
    }
    return mcube;
}
```
Our slice allocation routine

```c
FLT **slice(INT nrow1,INT nrow2,INT ncoll1,INT ncol2,FLT **temp) {
    INT i;
    FLT **mslice;
    mslice=(FLT **) malloc((unsigned) (nrow2-nrow1+1)*sizeof(FLT*));
    if (!mslice){
        printf("allocation failure at 3 in slice()\n");
        return NULL;
    }
    mslice -= nrow1;
    for(i=nrow1;i<=nrow2;i++) {
        if(i == nrow1){
            mslice[i]=*temp;
            mslice[i] -= ncoll1;
        }
        else {
            mslice[i]=mslice[i-1]+(ncol2-ncoll1);
        }
    }
    return mslice;
}
```

Digression... a 3d version of this routine