High Performance Computing

Autumn, 2018

Lecture 17

Imperial College
London

Prasun Ray

30 November 2018

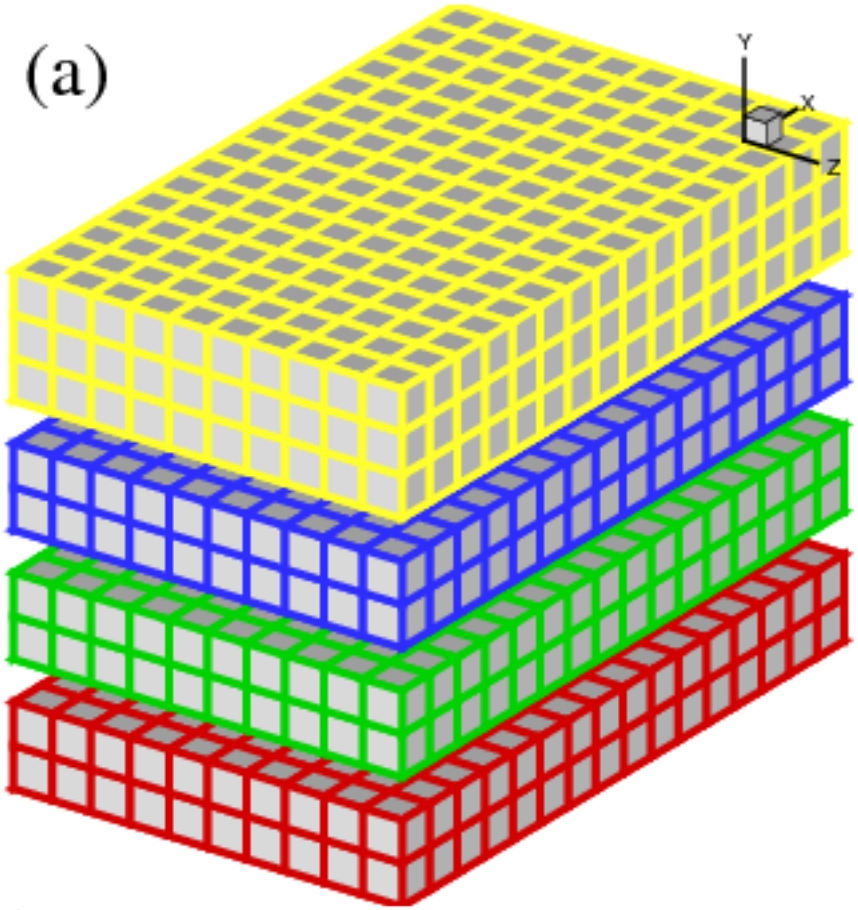
Comments

- **Final project will be posted online tomorrow evening ~8pm**
- **Homework 2 solutions have been posted online, marking will hopefully be completed by end of Friday**
- **No Panopto for lecture 16 – uploaded similar material from last year**
- **Be sure you can compile and run MPI midpoint code**
- **Adding** return ani **at the end of your animation function may (mysteriously) fix problems saving an** mp4
- **Running mpi on more processes than available cores:**
  1) create a text file and add the line: localhost slots=16
  (replace 16 with whatever you want)

2) $ mpiexec –n 8 --hostfile yourtextfile.txt something.exe

Send/Recv and domain decomposition

A parallel computation computes a potential field, *f(x,y,z,t)* on four processors.
P0, P1, P2, P3 solve for *f* in separate subdomains



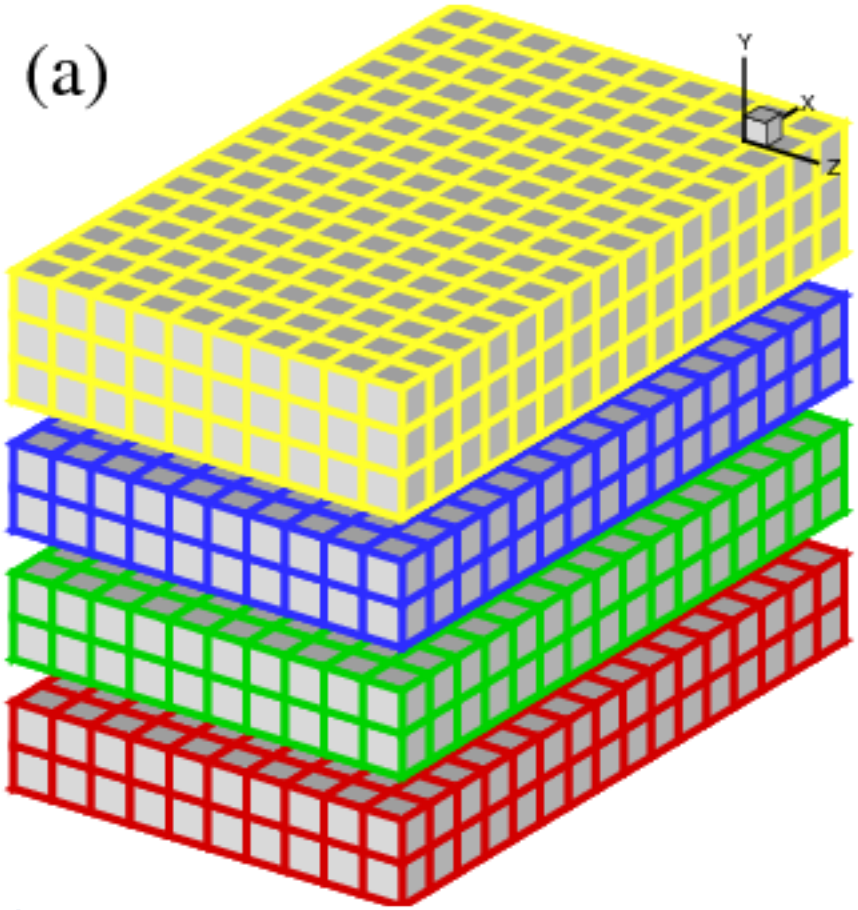
- **How would you compute the gradient?**

$$\nabla f = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right)$$

- **No problems with x and z directions**
- **But what about y?**

Last time: Send/Recv

A parallel computation computes a potential field, *f(x,y,z,t)* on four processors.
P0, P1, P2, P3 solve for *f* in separate subdomains



- **Send and Recv *must* be paired:**
  - **If yellow sends to blue, blue must recv from yellow**

- **Instead:**

Yellow → Blue
Blue → Green
Green → Red
Red → Yellow (if periodic)

and then the reverse

- **For large message sizes, sends will be blocking, this won't work!**

- **Can use** sendrecv **or** isend/irecv

Parallel differentiation example

```
!-----------------------------------------------------------
!Send data at top boundary up to next processor
!i.e. send f(nlocal+1) to myid+1 and store it there as f(1)
!data from myid=numprocs-1 is sent to myid=0
!-----------------------------------------------------------
    if (myid<numprocs-1) then
        receiver = myid+1
    else
         receiver = 0
    end if

    if (myid>0) then
        sender = myid-1
    else
        sender = numprocs-1
    end if

    call MPI_ISEND(f(Nlocal+1),1,MPI_DOUBLE_PRECISION,receiver,0,
                                        MPI_COMM_WORLD,request,ierr)
    call MPI_RECV(f(1) 1,MPI_DOUBLE_PRECISION,sender,MPI_ANY_TAG,
                                        MPI_COMM_WORLD,status,ierr)
```

Today

Solving the (steady) 2D heat equation

Heat equation

Task: Compute temperature distribution in a room

Governing equation: Heat equation (diffusion equation):

$$\frac{\partial T}{\partial t} = \alpha \nabla^2 T + S(\mathbf{x}, t)$$

$$T(\mathbf{x}, t = 0) = f(\mathbf{x})$$ **Initial condition**

Here, S **is a *heat source.* Boundary conditions should also be specified as appropriate.**

Problem: given the source, initial condition, and boundary conditions, solve for the temperature distribution, T(**x**,t)

1-D (steady) heat equation

First consider steady problem, e.g., S = S(x), a **and** b **are constants:**

$$\frac{\partial^2 T}{\partial x^2} + S(x) = 0$$ **Poisson equation**

Numerical method:

1. Discretize the derivative:

$$\frac{\partial^2 T}{\partial x^2} \approx \frac{T_{i+1} - 2T_i + T_{i-1}}{\Delta x^2}$$ **2nd-order, centered scheme**

$$x_i = i * \Delta x,\ i = 0, 1, 2, ..., N+1$$

$$(N+1) * \Delta x = 1$$

With boundary conditions: $T_0 = T_a,\ T_N = T_b$

Programming example

Equation for T_i: $$\frac{T_{i+1} - 2T_i + T_{i-1}}{\Delta x^2} = -S_i$$

In matrix form: AT = b

$$A = \begin{bmatrix} -2 & 1 & 0 & 0 & 0 & \dots & 0 \\ 1 & -2 & 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & -2 & 1 & 0 & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 & -2 & 1 \\ 0 & \dots & 0 & 0 & 0 & 1 & -2 \end{bmatrix}, \; b = \frac{1}{\Delta x^2} \begin{bmatrix} -\Delta x^2 T_a - S_1 \\ -S_2 \\ \vdots \\ -S_i \\ \vdots \\ -S_{N-1} \\ -\Delta x^2 T_b - S_N \end{bmatrix}$$

- **In 1-D, this is just a tridiagonal system of equations**
- **Easy to solve directly (with, say, *DGTSV*)**

Jacobi iteration

- Basic idea: rewrite Ax=b as $A_1x = A_2x + b$
- *Choose* A_1 so that it is easy to invert, then solve iterative system:
- $A_1x^{k+1} = A_2x^k + b$
  - Requires guess, x^0
- *Jacobi iteration*: Choose A_1 to be diagonal matrix (main diagonal of A):

$$\frac{T_{i+1}^{k-1} - 2T_i^k + T_{i-1}^{k-1}}{\Delta x^2} = -S_i$$

$$T_i^k = \frac{\Delta x^2}{2} S_i + \frac{1}{2}\left(T_{i+1}^{k-1} + T_{i-1}^{k-1}\right)$$

Main algorithm, easy to code!

Jacobi iteration in Fortran

- **One Fortran trick: set variables to be** dimension(0:N+1)
  - x(0)=0, x(N+1)=1, T(0)=a, T(N+1)=b
  - **Then, easy to compute T_1 using:**

$$T_i^k = \frac{\Delta x^2}{2} S_i + \frac{1}{2}\left(T_{i+1}^{k-1} + T_{i-1}^{k-1}\right)$$

Core part of code (see *jacobi1s.f90*):

```
do k1=1,kmax
   Tnew(1:n) = S(1:n)*dx2f + 0.5d0*(T(0:n-1) + T(2:n+1)) !Jacobi
   deltaT(k1) = maxval(abs(Tnew(1:n)-T(1:n))) !compute relative error
   T(1:n)=Tnew(1:n)    !update variable
   if (deltaT(k1)<tol) exit !check convergence criterion
end do
```

Parallel Jacobi

Let's now parallelize the solver with OpenMP

- **Look for loops that can be parallelized**
- **Look for vectorized operations that can be converted to loops that can be parallelized**

Parallel:

```
dmax=0.d0
!$omp parallel do reduction(max:dmax)
do i1=1,n
    Tnew(i1) = S(i1)*dx2f + 0.5d0*(T(i1-1) + T(i1+1))
    dmax = max(dmax,abs(Tnew(i1)-T(i1)))
end do
!$omp end parallel do
deltaT(k1) = dmax
```

2-D (steady) heat equation

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + S(x,y) = 0$$ **Poisson equation**

Numerical method:

1. Discretize the derivative:

$$\left(\frac{\partial^2 T}{\partial x^2}\right)_{i,j} \approx \frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{\Delta x^2}$$

$$\left(\frac{\partial^2 T}{\partial y^2}\right)_{i,j} \approx \frac{T_{i,j+1} - 2T_{i,j} + T_{i,j-1}}{\Delta y^2}$$

$$\Delta x = \Delta y = \Delta$$

2nd-order, centered scheme

With boundary conditions:

$$T(x=0,y) = L(y),\ T(x=1,y) = R(y)$$

$$T(x,y=0) = D(x),\ T(x,y=1) = U(x)$$

2-D (steady) heat equation

Numerical method:

1. Discretize the derivative:

$$\left(\frac{\partial^2 T}{\partial x^2}\right)_{i,j} \approx \frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{\Delta x^2}$$

$$\left(\frac{\partial^2 T}{\partial y^2}\right)_{i,j} \approx \frac{T_{i,j+1} - 2T_{i,j} + T_{i,j-1}}{\Delta y^2}$$

$$\Delta x = \Delta y = \Delta$$

2nd-order, centered scheme

(How does this change if dx dy are different?)

Final equation:

$$\frac{T_{i+1,j} + T_{i,j+1} - 4T_{i,j} + T_{i-1,j} + T_{i,j-1}}{\Delta^2} = -S_i$$

$$x_i = i * \Delta,\ i = 0, 1, 2, ..., N+1$$

$$y_j = j * \Delta,\ j = 0, 1, 2, ..., N+1$$

With b.c.'s imposed at i=0, i=N+1 **and** j=0, j=N+1

2-D (steady) heat equation

We now have system of n^2 linear equations, AT = B:

$$A = \begin{bmatrix} M & I & 0 & 0 & 0 & \dots & 0 \\ I & M & I & 0 & 0 & \dots & 0 \\ 0 & I & M & I & 0 & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & I & M & I \\ 0 & \dots & 0 & 0 & 0 & I & M \end{bmatrix}$$

- **1D: A was tridiagonal**
- **2D: A is now *block* tridiagonal**
- **M is a** n x n **tridiagonal matrix**

Note: 1st row of 1st M and last row of nth M will typically be different to enforce boundary conditions

$$M = \begin{bmatrix} -4 & 1 & 0 & 0 & 0 & \dots & 0 \\ 1 & -4 & 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & -4 & 1 & 0 & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 & -4 & 1 \\ 0 & \dots & 0 & 0 & 0 & 1 & -4 \end{bmatrix}$$

2-D (steady) heat equation

We now have system of n^2 linear equations, AT = B:

$$T = \begin{bmatrix} T_{11} \\ T_{21} \\ \vdots \\ T_{n1} \\ T_{12} \\ \vdots \\ T_{n2} \\ \vdots \\ T_{1n} \\ \dots \\ T_{nn} \end{bmatrix} \qquad B = \begin{bmatrix} S_{11}\Delta^2 - T_{01} - T_{10} \\ S_{21}\Delta^2 - T20 \\ \vdots \\ S_{n1}\Delta^2 - T_{n0} - T_{n+1,1} \\ S_{12}\Delta^2 - T_{02} \\ \vdots \\ S_{n2}\Delta^2 - T_{n+1,2} \\ \vdots \\ S_{n1}\Delta^2 - T_{n+1,1} \\ \vdots \\ S_{nn}\Delta^2 - T_{n+1,n} - T_{n,n+1}] \end{bmatrix}$$

Boundary conditions appear in appropriate elements of B (refer to example code)

2-D (steady) heat equation

We now have system of n^2 linear equations, $AT = B$

- **Direct solution of n x n matrix: $O(n^3)$ operations (LU decomposition of** A **+ back-substitution with** B**)**
- **We have n^2 x n^2 matrix, so on 100 x 100 grid, matrix is 1e4 x 1e4 and contains 1e8 elements (in double precision, that's 800 mb!)**
- **So, in 2D (and 3D) direct solution becomes expensive and memory-intensive**

2-D (steady) heat equation

We now have system of n^2 linear equations, $AT = B$

- **Direct solution of n x n matrix: $O(n^3)$ operations (LU decomposition of** A **+ back-substitution with** B**)**
- **We have n^2 x n^2 matrix, so on 100 x 100 grid, matrix is 1e4 x 1e4 and contains 1e8 elements (in double precision, that's 800 mb!)**
- **So, in 2D (and 3D) direct solution becomes expensive and memory-intensive**
- **But the matrix,** A**, is *sparse*, so iterative method will only need to store $O(n^2)$ elements (rather than n^4)**
- **A good iterative method (e.g. conjugate gradient, multigrid) will be faster as well**
- **Jacobi iteration is *inefficient,* but a good starting point for looking at iterative methods**

2-D (steady) heat equation

- **Moving to 2D, 3D means large matrices, large memory requirements**
  - **With finite-difference methods, these matrices are often *sparse***
  - **There are a few standard approaches for efficiently storing sparse matrices**
  - **Simplest: for each non-zero element, store the (i,j) coordinate and the value of that element.**
  - **Example from scipy.sparse:**

```
>>> # Constructing a matrix using ijv format
>>> row  = np.array([0, 3, 1, 0])
>>> col  =  np.array([0, 3, 1, 2])
>>> data = np.array([4, 5, 7, 9])
>>> coo_matrix((data, (row, col)), shape=(4, 4)).toarray()
array([[4, 0, 9, 0],
       [0, 7, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 5]])
```

2-D (steady) heat equation

Jacobi iteration:

$$T_{i,j}^{k+1} = \frac{\Delta^2}{4} S_i + \frac{1}{4}\left(T_{i+1,j}^{k} + T_{i,j+1}^{k} + T_{i-1,j}^{k} + T_{i,j-1}^{k}\right)$$

- **New temperature = Source contribution + average of surrounding temperatures**
- **How do we convert 1D code → 2D?**

2-D (steady) heat equation

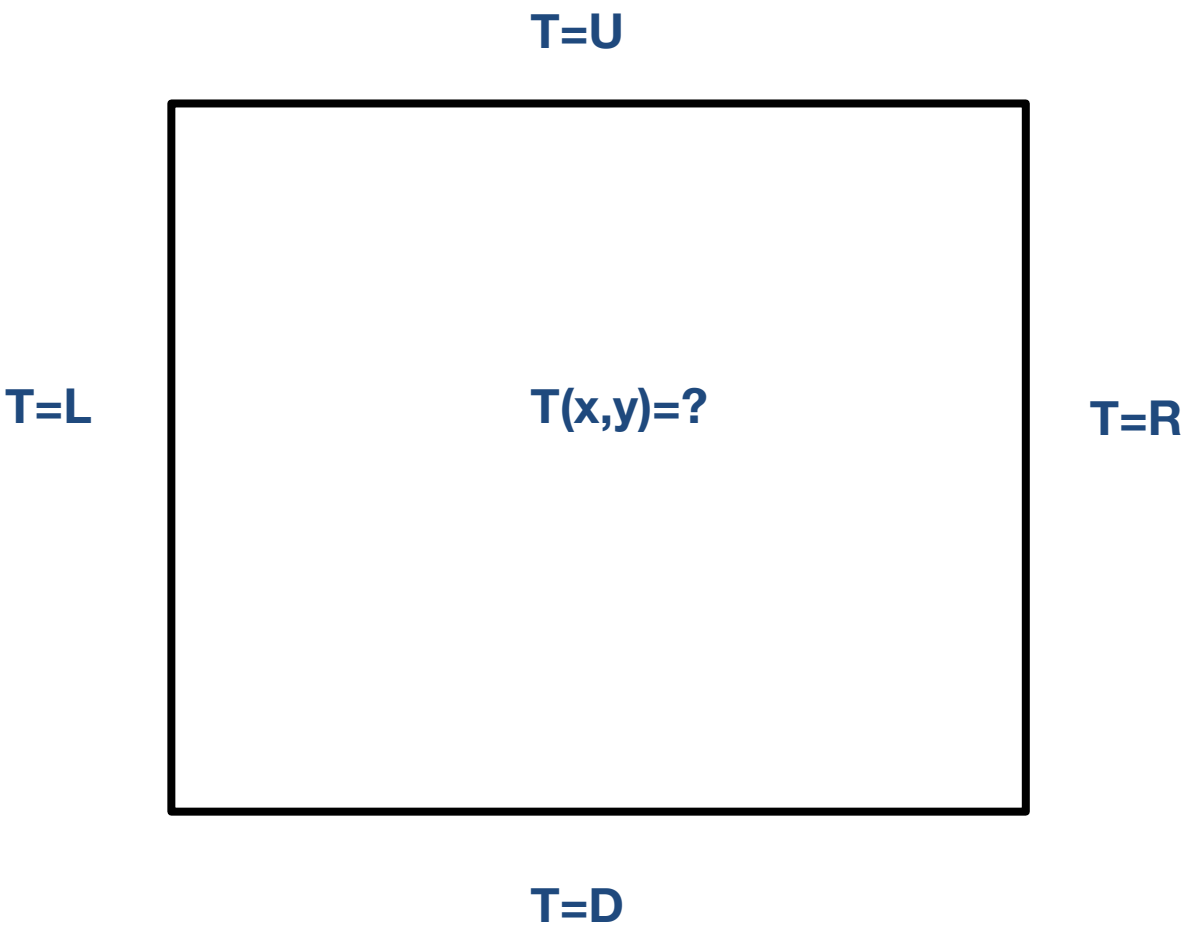
Jacobi iteration:

$$T_{i,j}^{k+1} = \frac{\Delta^2}{4} S_i + \frac{1}{4}\left(T_{i+1,j}^{k} + T_{i,j+1}^{k} + T_{i-1,j}^{k} + T_{i,j-1}^{k}\right)$$

- New temperature = Source contribution + average of surrounding temperatures
- How do we convert 1D code → 2D?
- Plan:
  1. Need 2D variables: x, y, S, T
  2. Initialize 2D field and apply boundary conditions
  3. During iterations, average in two dimensions rather than one

2-D (steady) heat equation

Final problem: Use Jacobi iteration to find $T(x,y)$ **on unit square with fixed temperature on boundaries and prescribed source,** $S(x,y)$



Jacobi iteration in Fortran

1D:

```
do k1=1,kmax
   Tnew(1:n) = S(1:n)*dx2f + 0.5d0*(T(0:n-1) + T(2:n+1)) !Jacobi
   deltaT(k1) = maxval(abs(Tnew(1:n)-T(1:n))) !compute relative error
   T(1:n)=Tnew(1:n)    !update variable
   if (deltaT(k1)<tol) exit !check convergence criterion
end do
```

Jacobi iteration in Fortran

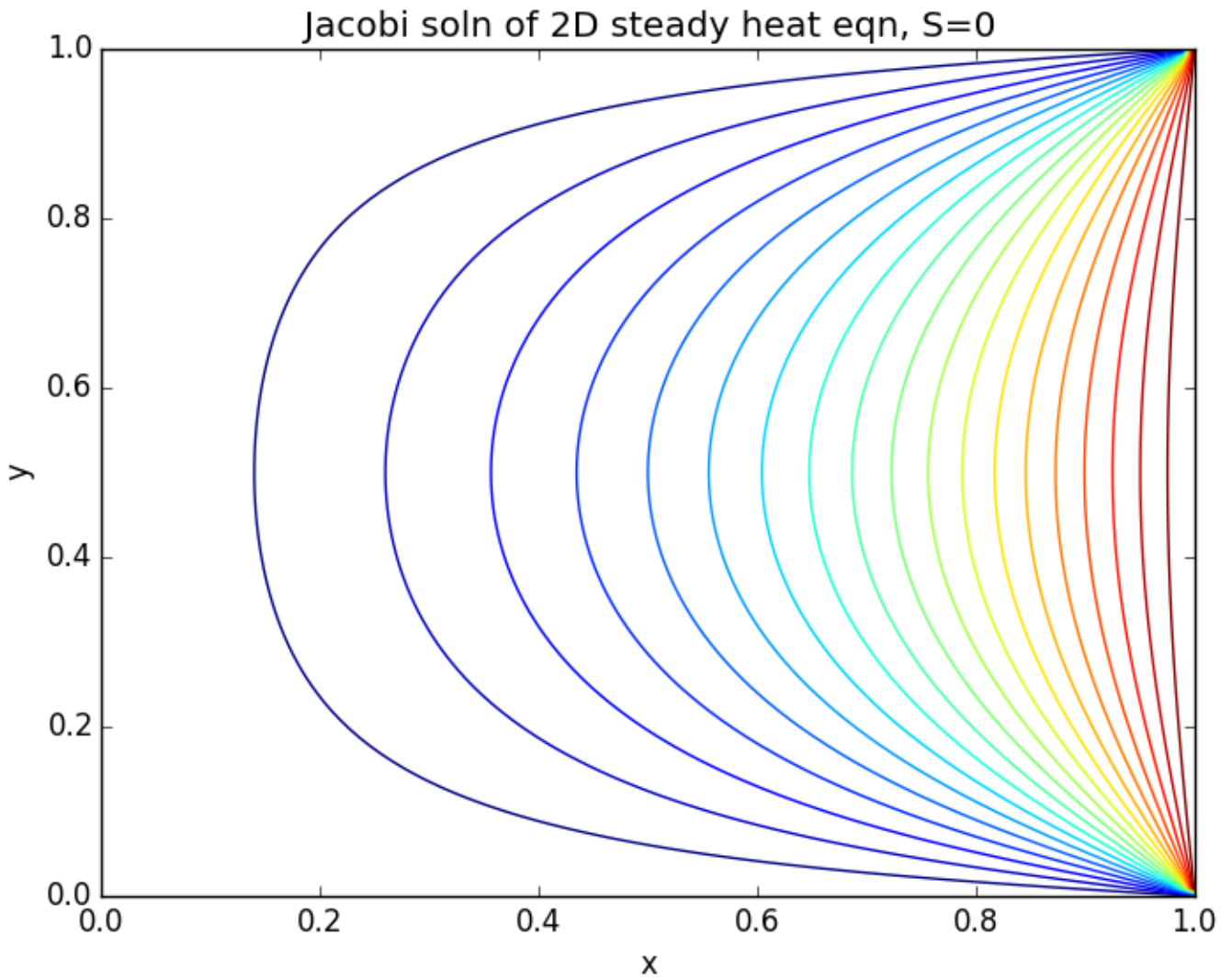
2D (see *jacobi2s.f90*):

```
 do k1=1,kmax
        Tnew(1:n,1:n) = S(1:n,1:n)*del2f +
0.25*(T(2:n+1,1:n) + T(0:n-1,1:n) + T(1:n,0:n-1) + T(1:n,2:n+1)) !Jacobi
deltaT(k1) = maxval(abs(Tnew(1:n,1:n)-T(1:n,1:n))) !compute relative
error
        T(1:n,1:n)=Tnew(1:n,1:n)    !update variable
        if (deltaT(k1)<tol) exit !check convergence criterion
    end do
```

- **Run code with** U=D=L=0, R=1
- $S = S_0 \sin(\pi x) \sin(\pi y)$

2-D (steady) heat equation

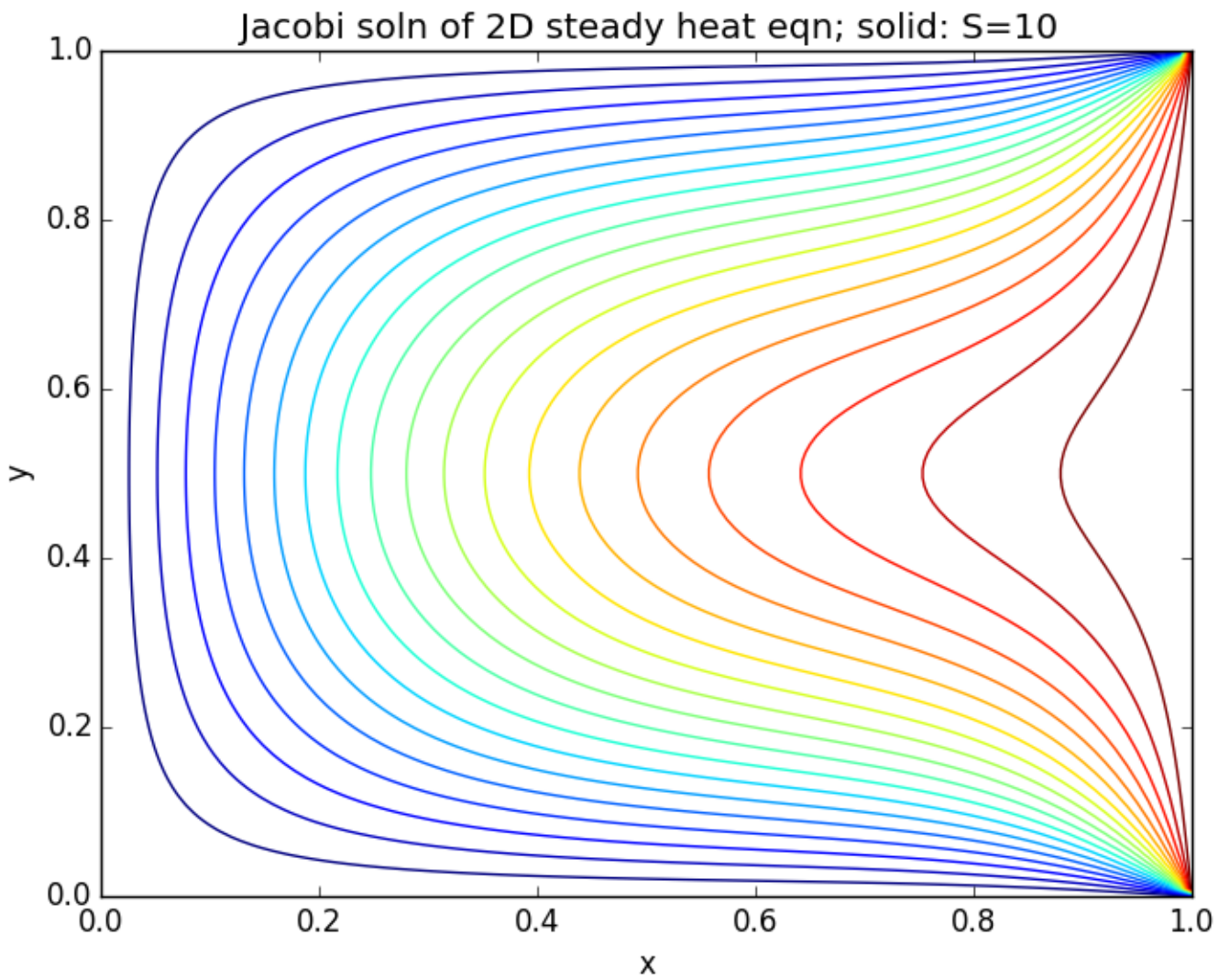
Homogeneous solution (S=0):



The hot wall sets the temperature distribution

2-D (steady) heat equation

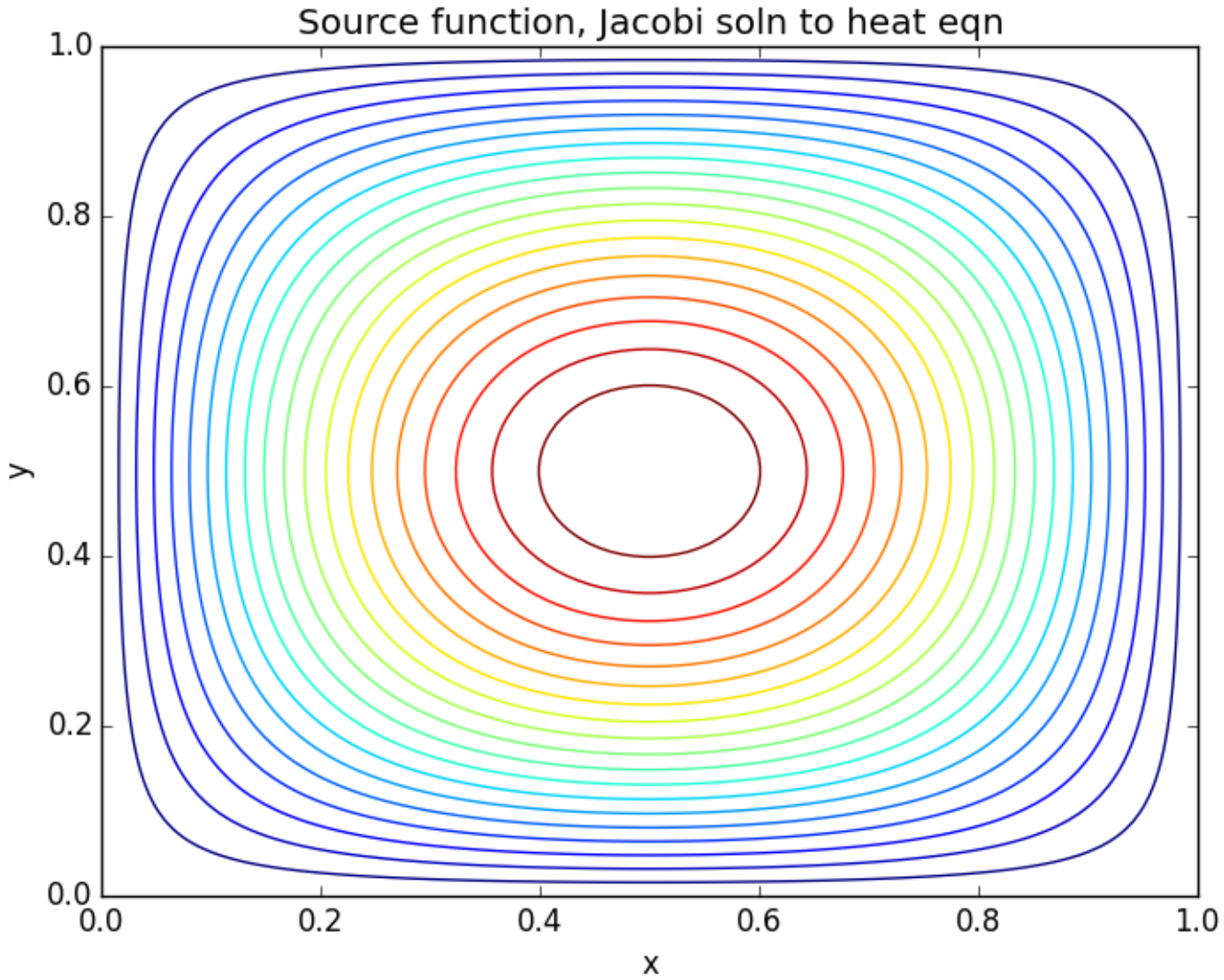
Particular solution (S=10):



Source in middle of room shifts temperature distribution

2-D (steady) heat equation

Source function:



Source in middle of room shifts temperature distribution

2-D (steady) heat equation

Parallelize with OpenMP:

- **Essentially the same as in 1D**
- **Parallel loop iterates across rows of** A
- **Reduction of deltaT to check for convergence**

2-D (steady) heat equation

1D:

```
dmax=0.d0
!$omp parallel do reduction(max:dmax)
do i1=1,n
    Tnew(i1) = S(i1)*dx2f + 0.5d0*(T(i1-1) + T(i1+1))
    dmax = max(dmax,abs(Tnew(i1)-T(i1)))
end do
!$omp end parallel do
deltaT(k1) = dmax
if (deltaT(k1)<tol) exit !check convergence criterion

!$omp parallel do
do i1=1,n
    T(i1) = Tnew(i1)
end do
!$omp end parallel do
```

2-D (steady) heat equation

2D (*see jacobi2s_omp.f90*):

```
dmax = 0.0
!$OMP parallel do reduction(max:dmax)
do j1=1,n
    Tnew(1:n,j1) = S(1:n,j1)*del2f + 0.25*(T(2:n+1,j1) +
  T(0:n-1,j1) + T(1:n,j1-1) + T(1:n,j1+1))!Jacobi iteration
    dmax = max(dmax,maxval(abs(Tnew(1:n,j1)-T(1:n,j1))))
end do
!$OMP end parallel do
deltaT(k1) = dmax
if (deltaT(k1)<tol) exit !check convergence criterion
!$OMP parallel do
do j1=1,n
    T(1:n,j1) = Tnew(1:n,j1)
end do
!$OMP end parallel do
```

2-D (steady) heat equation

- Moving from 1D serial to 2D parallel (with OpenMP) is straightforward
- Much more difficult if solving equations directly
- Moving to 2D, 3D means large matrices, large memory requirements

2-D (steady) heat equation

- Moving from 1D serial to 2D parallel (with OpenMP) is straightforward
- Much more difficult if solving equations directly