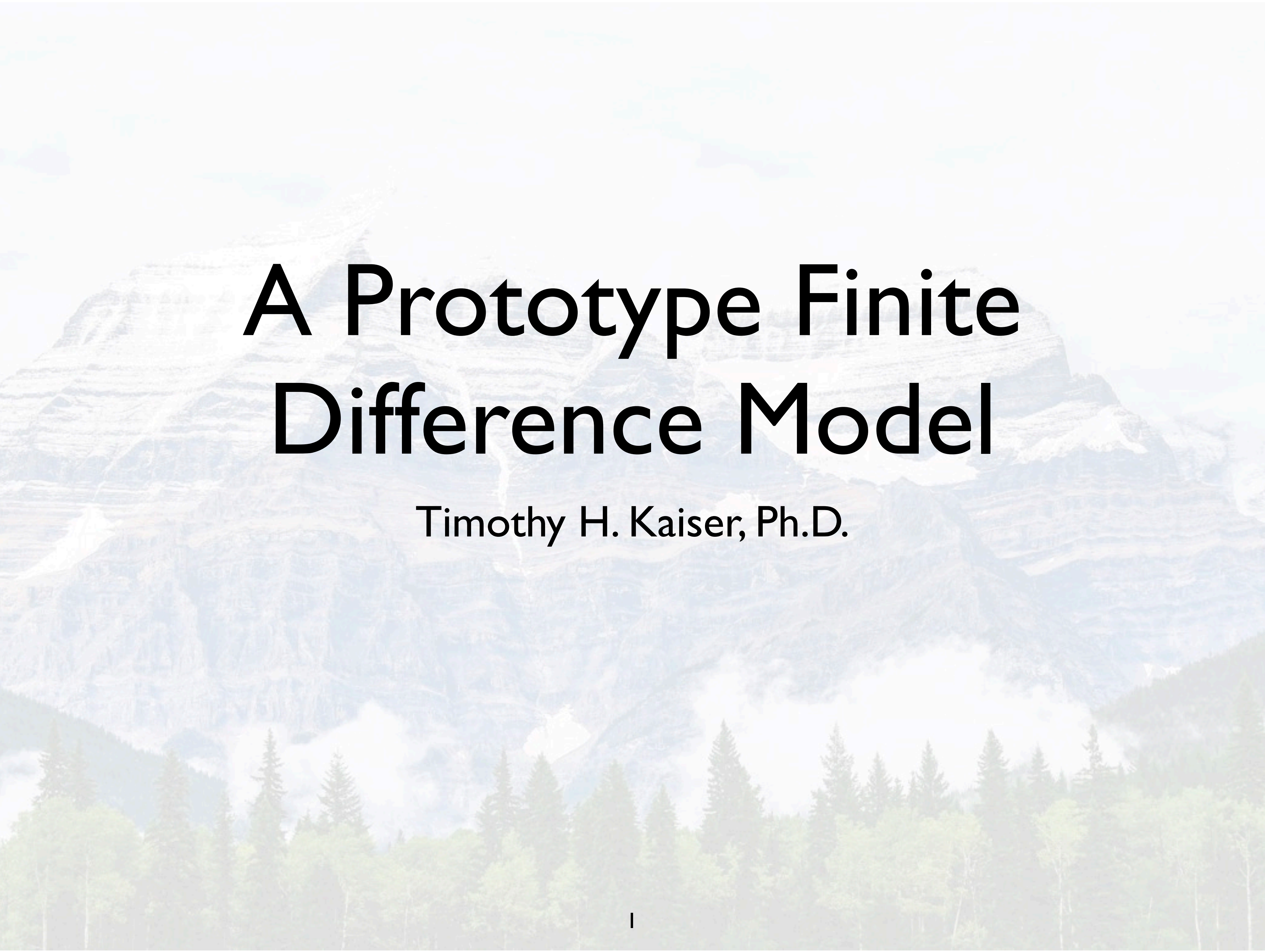


A Prototype Finite Difference Model

Timothy H. Kaiser, Ph.D.

Slides:

https://github.com/timkphd/slides

Source

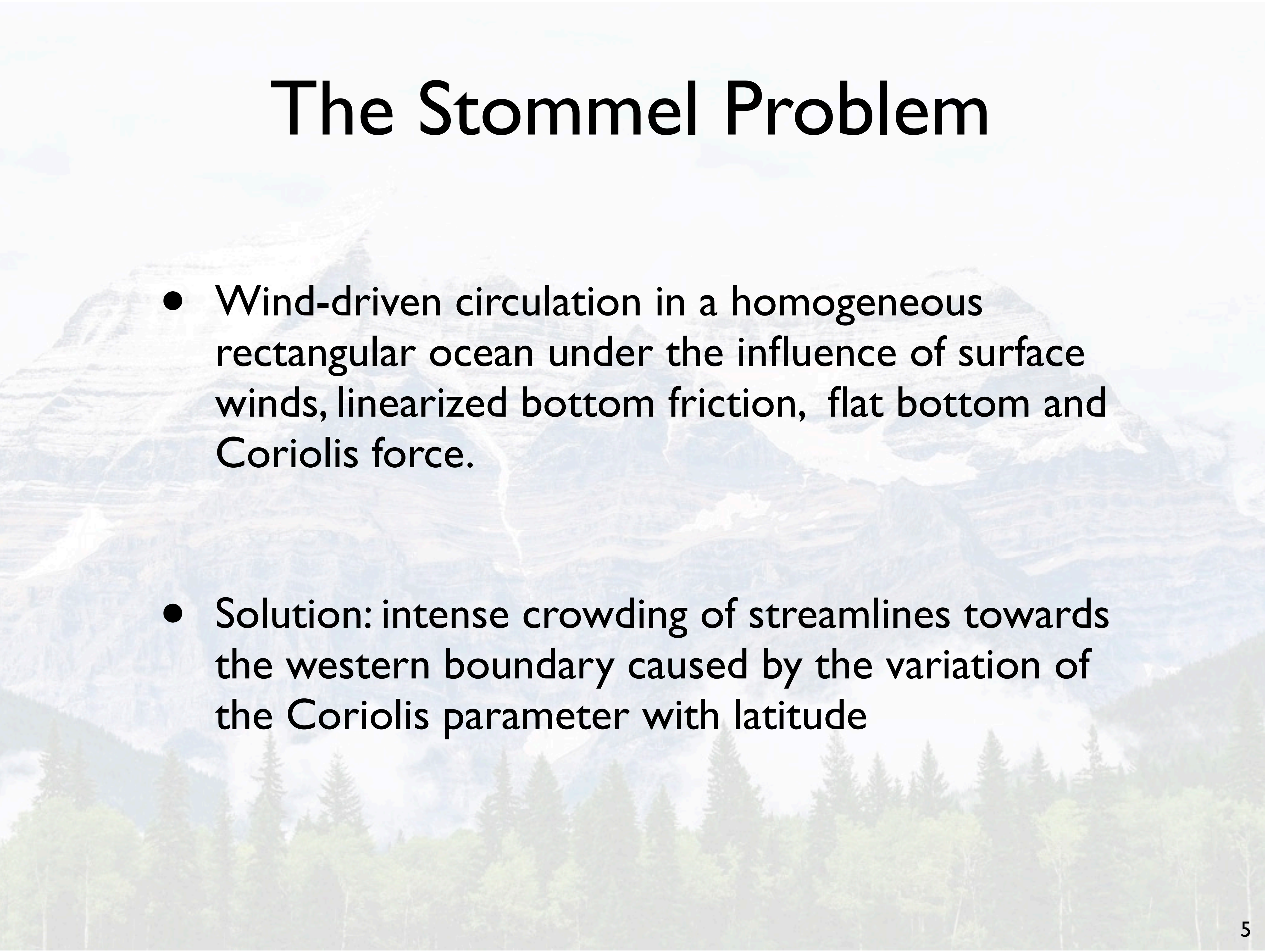
```
hexagon:~ tkaiser$ mkdir tut
hexagon:~ tkaiser$ cd tut
hexagon:tut tkaiser$ git clone https://github.com/timkphd/examples.git
Cloning into 'examples'...
remote: Enumerating objects: 83, done.
remote: Counting objects: 100% (83/83), done.
remote: Compressing objects: 100% (63/63), done.
remote: Total 1236 (delta 39), reused 45 (delta 19), pack-reused 1153
Receiving objects: 100% (1236/1236), 2.47 MiB | 3.64 MiB/s, done.
Resolving deltas: 100% (390/390), done.
hexagon:tut tkaiser$ mv examples/r .
hexagon:tut tkaiser$ rm -rf examples
hexagon:tut tkaiser$ cd r

hexagon:r tkaiser$ ./getdat.py
downloading file
True
split -l 158563 start start
hexagon:r tkaiser$ tar -xzf laser.tgz
hexagon:r tkaiser$

hexagon:r tkaiser$ Rscript doinstall.R
. . .
. . .
hexagon:r tkaiser$ R CMD SHLIB mapit.c
```

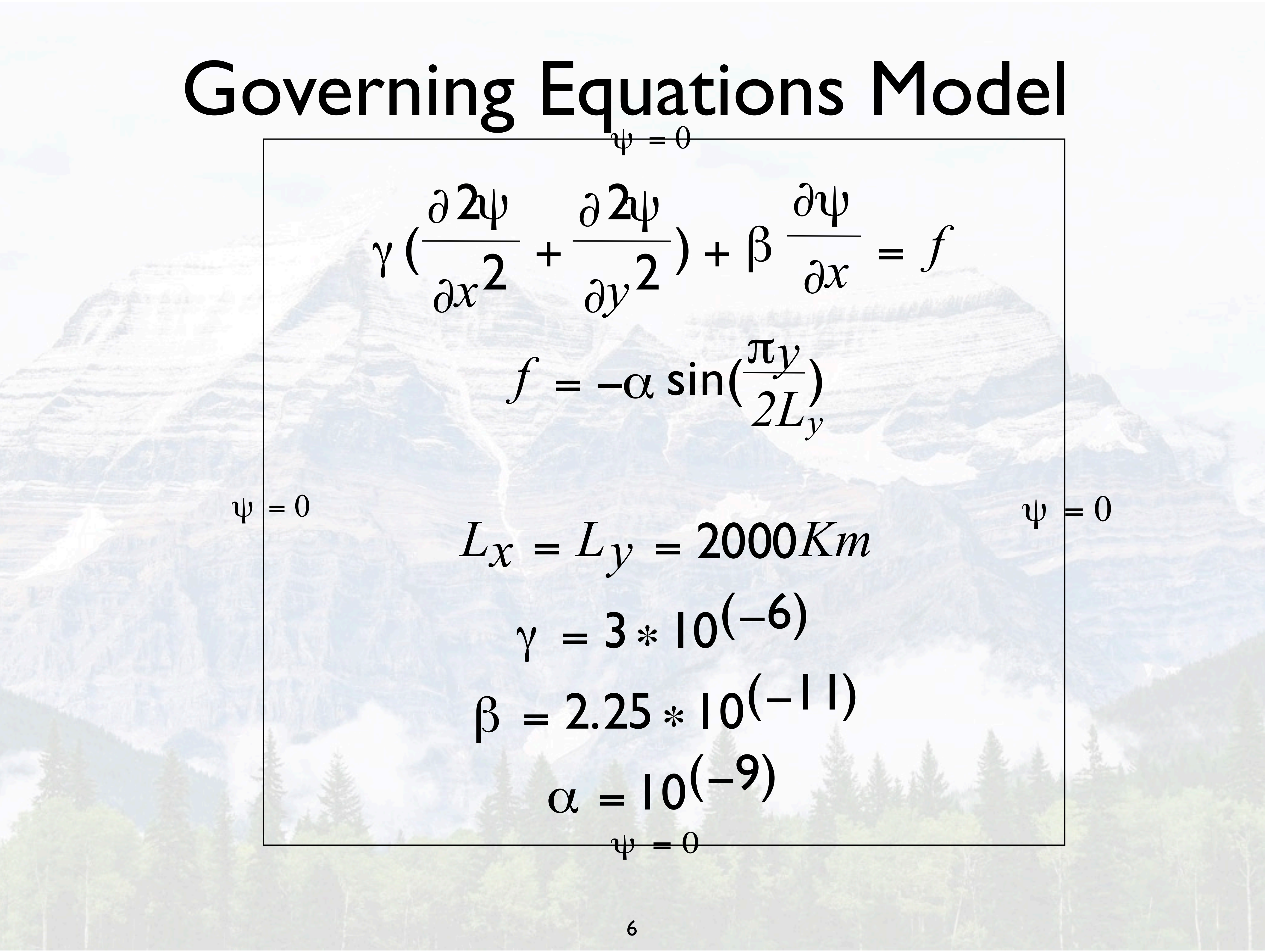
A Prototype Model

- 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



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

$\psi = 0$

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

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

$\psi = 0$ $\psi = 0$

$$L_x = L_y = 2000Km$$

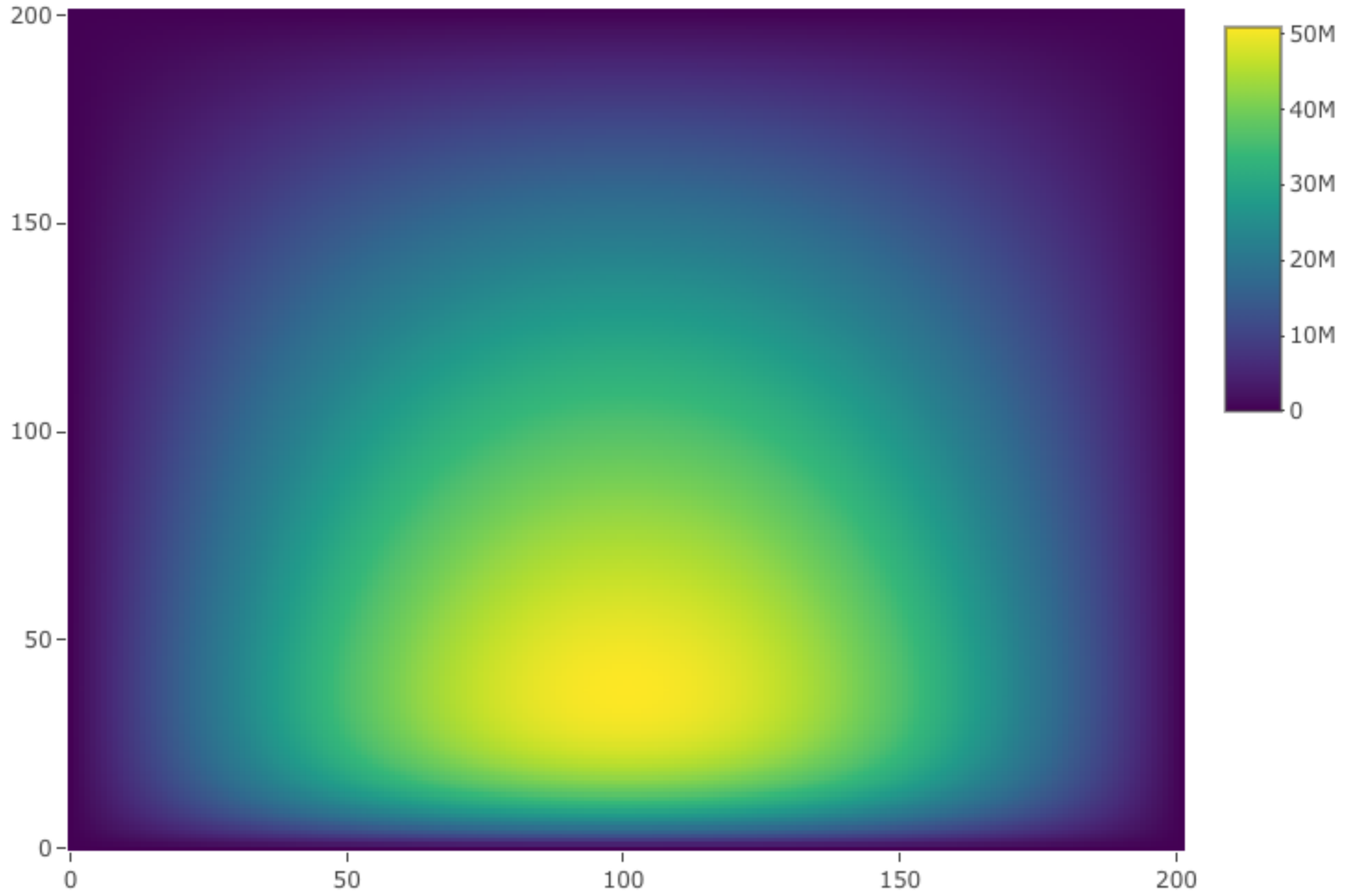
$$\gamma = 3 * 10^{(-6)}$$

$$\beta = 2.25 * 10^{(-11)}$$

$$\alpha = 10^{(-9)}$$

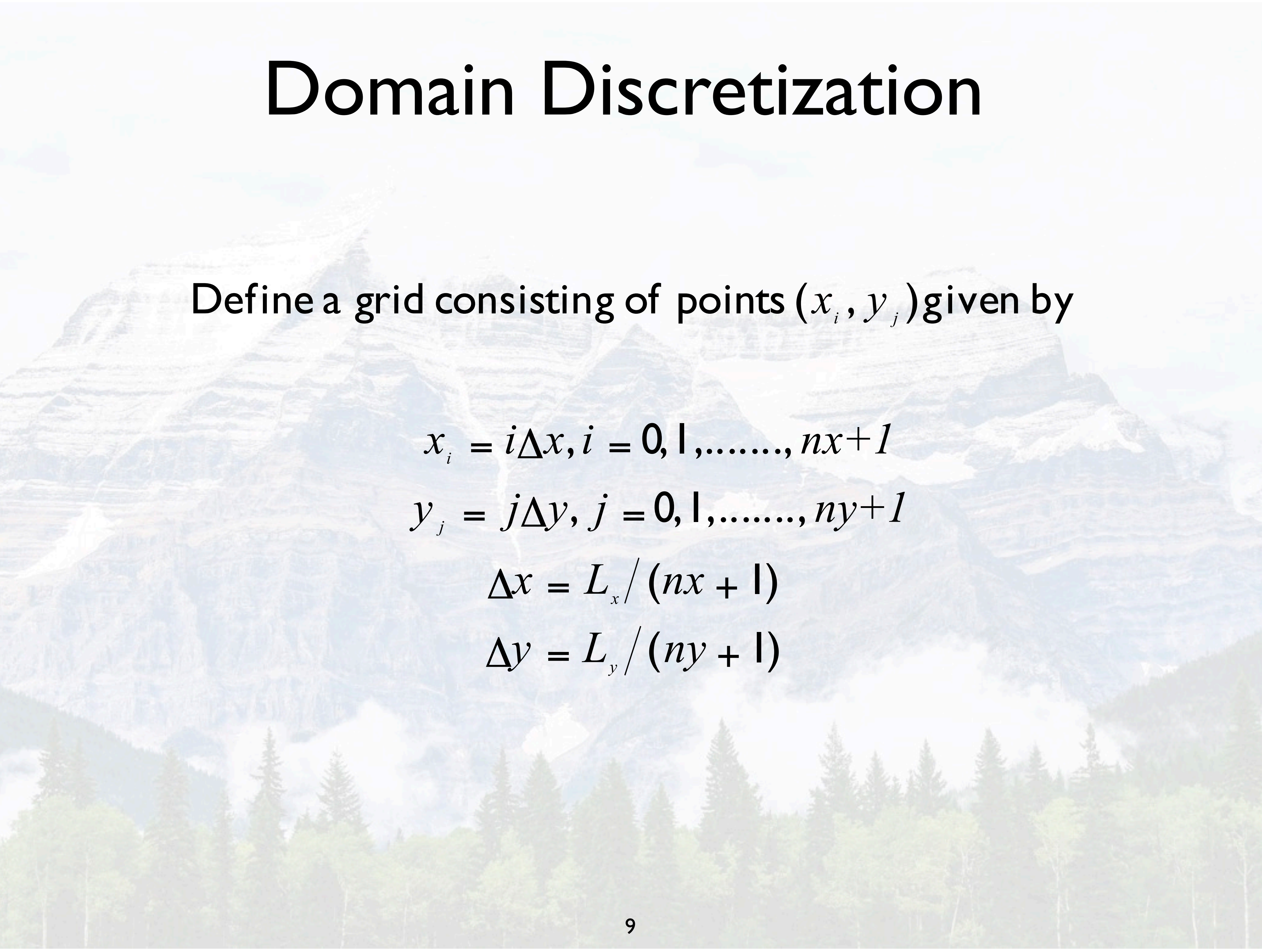
$\psi = 0$

The steady state solution



R is !!!!MUCH!!!! slower

- Our “normal” example calculation 200x200 grid 75k steps
  - R > 2000 seconds
  - C/Fortran 3 seconds
- You can however, call C and Fortran compiled subroutines from R
- You can mix R and C/Fortran in MPMD fashion
  - MPI tasks 0 to n-2 might be Fortran or C
  - MPI task n-1 could be a R graphics program
    - Have example
    - mpiexec -n 3 ./ccalc : -n 1 plot.R < small.in



Domain Discretization

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

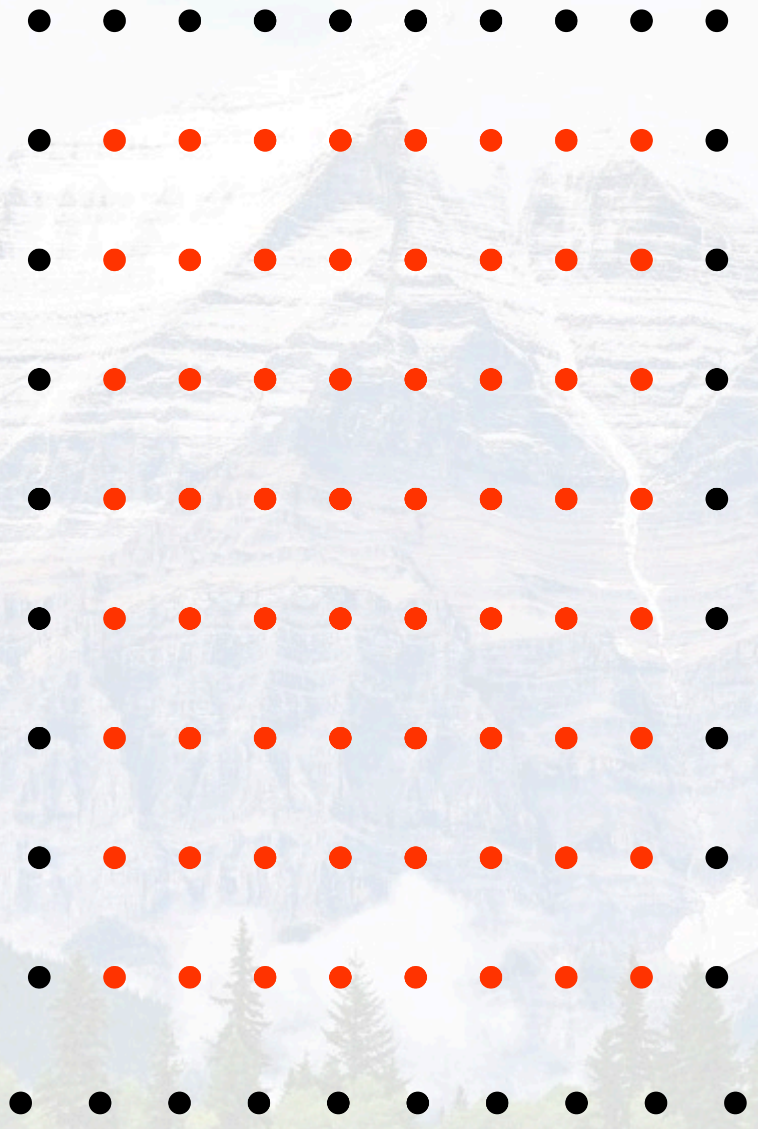
$$x_i = i\Delta x, i = 0, 1, \ldots\ldots., nx+1$$

$$y_j = j\Delta y, j = 0, 1, \ldots\ldots., ny+1$$

$$\Delta x = L_x / (nx + 1)$$

$$\Delta y = L_y / (ny + 1)$$

Domain Discretization

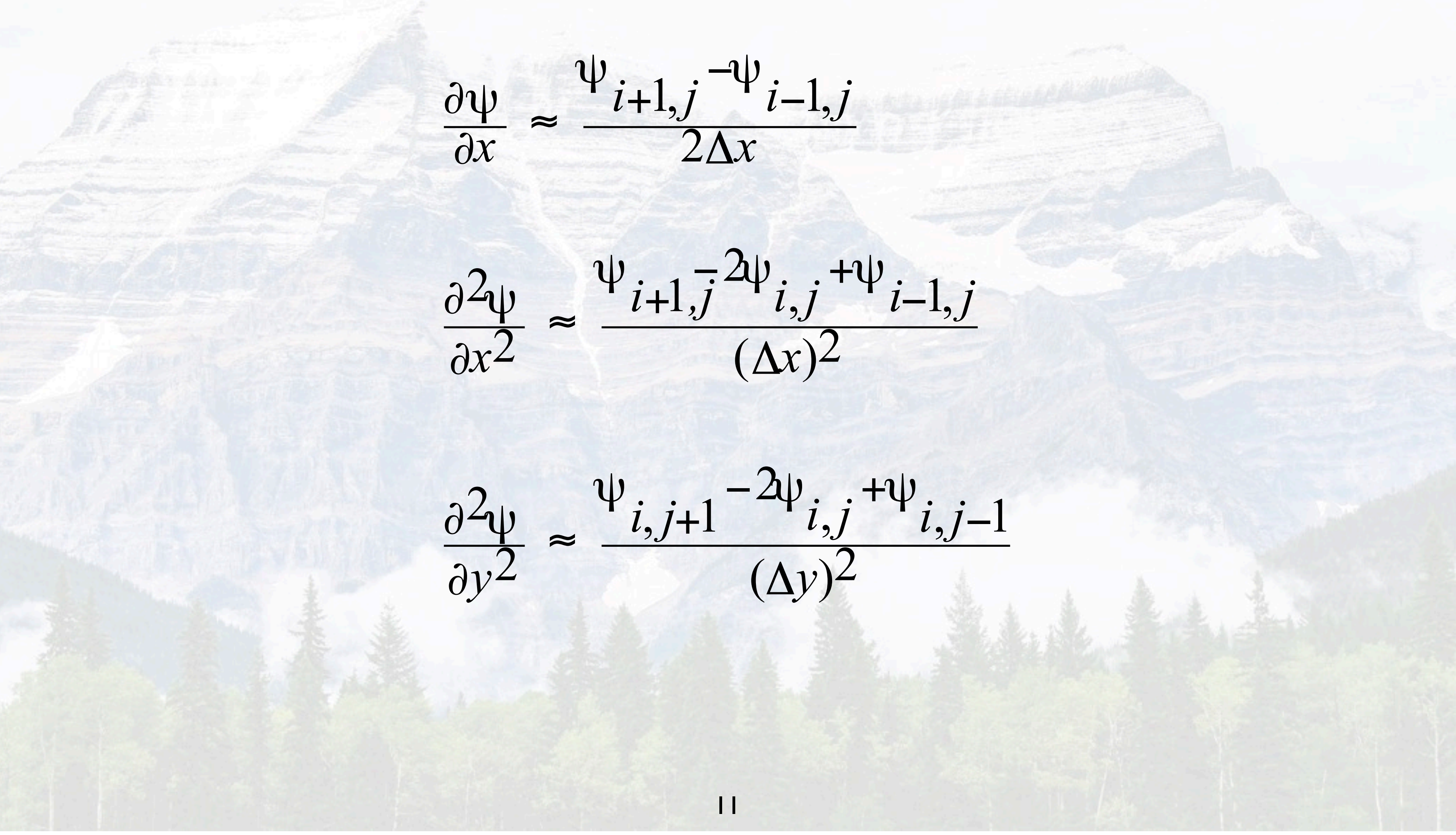


Seek to find an approximate solution

$\psi\,(x_i, y_j)$ at points (x_i, y_j):

$$\psi_{i,j} \approx \psi\,(x_i, y_j)$$

Centered Finite Difference Scheme for the Derivative Operators

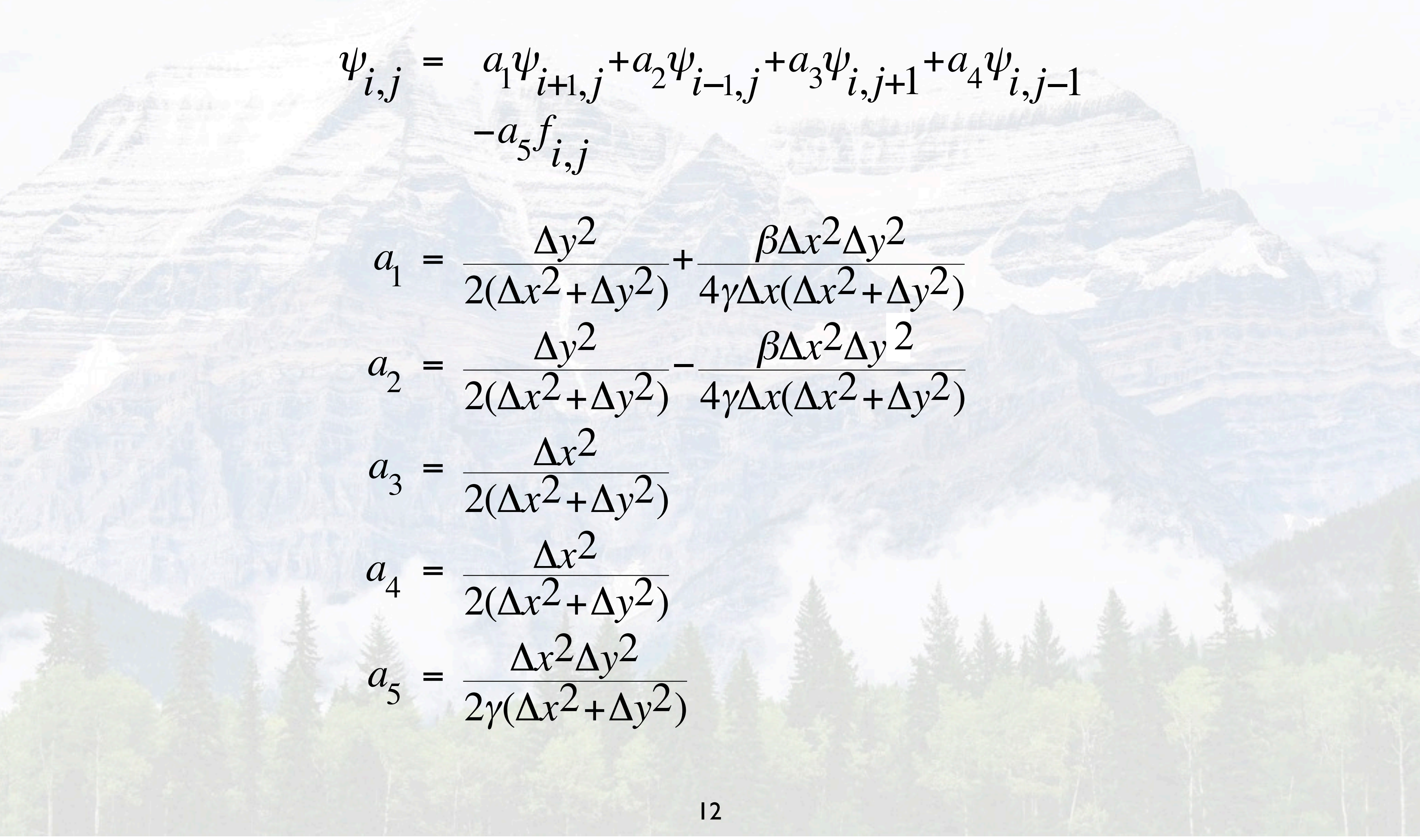


$$\frac{\partial\psi}{\partial x} \approx \frac{\psi_{i+1,j} - \psi_{i-1,j}}{2\Delta x}$$

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

$$\frac{\partial^2\psi}{\partial y^2} \approx \frac{\psi_{i,j+1} - 2\psi_{i,j} + \psi_{i,j-1}}{(\Delta y)^2}$$

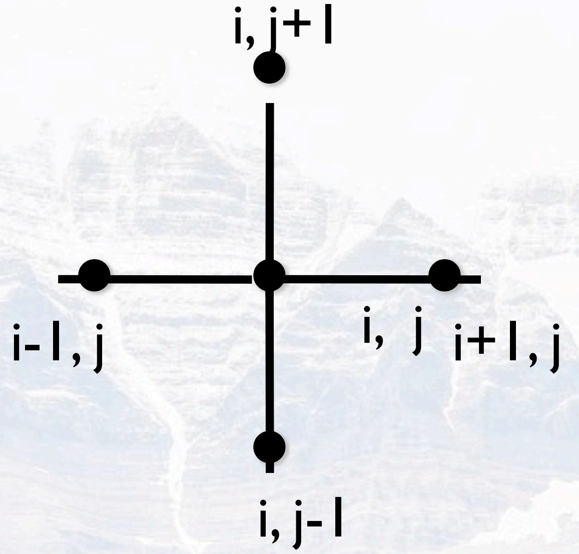
Governing Equation Finite Difference Form



$$
\begin{aligned}
\psi_{i,j} &= a_1\psi_{i+1,j} + a_2\psi_{i-1,j} + a_3\psi_{i,j+1} + a_4\psi_{i,j-1} \\
&\quad - a_5 f_{i,j}
\end{aligned}
$$

$$
\begin{aligned}
a_1 &= \frac{\Delta y^2}{2(\Delta x^2 + \Delta y^2)} + \frac{\beta \Delta x^2 \Delta y^2}{4\gamma \Delta x(\Delta x^2 + \Delta y^2)} \\
a_2 &= \frac{\Delta y^2}{2(\Delta x^2 + \Delta y^2)} - \frac{\beta \Delta x^2 \Delta y^2}{4\gamma \Delta x(\Delta x^2 + \Delta y^2)} \\
a_3 &= \frac{\Delta x^2}{2(\Delta x^2 + \Delta y^2)} \\
a_4 &= \frac{\Delta x^2}{2(\Delta x^2 + \Delta y^2)} \\
a_5 &= \frac{\Delta x^2 \Delta y^2}{2\gamma(\Delta x^2 + \Delta y^2)}
\end{aligned}
$$

Five-point Stencil Approximation



interior grid points: i=1,nx;
j=1,ny

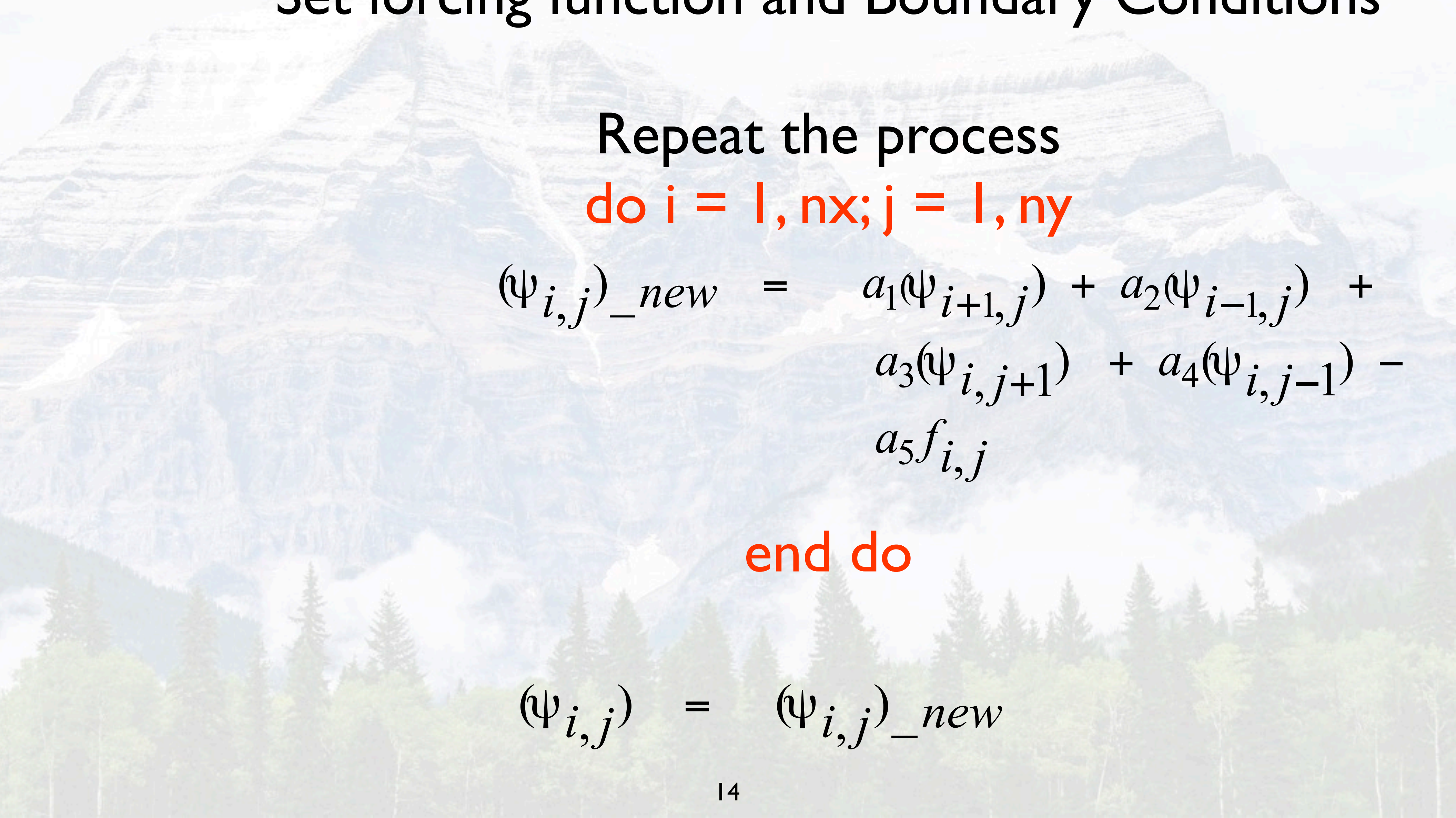
boundary points:

(i,0) & (i,ny+1) ; i=0,nx+1

(0,j) & (nx+1,j) ; j=0,ny+1

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

$$\psi_{i,0} = \psi_{i,ny+1} = 0; \quad \psi_{0,j} = \psi_{nx+1,j} = 0;$$



Jacobi Iteration

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

Set forcing function and Boundary Conditions

Repeat the process

```
do i = 1, nx; j = 1, ny
    $(\psi_{i,j})\_new = a_1(\psi_{i+1,j}) + a_2(\psi_{i-1,j}) +$
        $a_3(\psi_{i,j+1}) + a_4(\psi_{i,j-1}) -$
        $a_5 f_{i,j}$
end do
$(\psi_{i,j}) = (\psi_{i,j})\_new$
```

R Version - Forcing Function

```
do_force<-function(i1,i2,j1,j2,alpha){
    forf<-matrix(0.0,nrow=((i2-i1)+3),ncol=((j2-j1)+3))
    r1<-range(0,(i2-i1)+2)
    r2<-range(0,(j2-j1)+1)
    for (i in r1[1]:r1[2]){
        for (j in r2[1]:r2[2]){
            y<-(j+j1-1)*dy
            forf[i+1,j+1]<-(-alpha*sin(y*a6))
        }
    }
    return(forf)
}
```

R Version - Boundary Conditions

```r
bc<-function(psi,i1,i2,j1,j2,nx,ny){
   myid <<-0
   numprocs <<- 1
   myleft<-myid-1
   myright<-myid+1
   if(myleft <= -1){myleft<-(-1)}
   if(myright >= numprocs){myright=(-1)}
   if(myleft == -1){
      psi[,1]=0.0

   }
   if(myright == -1){
      psi[,ncol(psi)]=0.0
   }
      psi[1,]=0.0
      psi[nrow(psi),]=0.0
   return(psi)
}
```

R Version - Set Up

```
ngrid<-scan("st.in",integer(),2)
grid<-scan("st.in",double(),2,skip=1)
const<-scan("st.in",double(),3,skip=2)
steps<-scan("st.in",integer(),1,skip=3)
nx<-ngrid[1]
ny<-ngrid[2]
lx<-grid[1]
ly<-grid[2]
alpha<-const[1]
beta<-const[2]
gamma<-const[3]


numprocs<-1
myid<-0
i1org<-1
i2org<-nx
j1org<-1
j2org<-ny
i1<-1
i2<-nx
j1<-1
j2<-ny
dj<-as.double(j2)/as.double(numprocs)
j1<-round(1.0+myid*dj)
j2<-round(1.0+(myid+1)*dj)-1
```

R Version - Calculation Function

```
do_jacobi<-function(psi){
# does a single Jacobi iteration step
# input is the grid and the indices for the interior cells
# new_psi is temp storage for the the updated grid
# output is the updated grid in psi and diff which is
# the sum of the differences between the old and new grids
    gdiff<<-0.0
    is=2
    ie=dim(psi)[1]
    js=2
    je=dim(psi)[2]
    new_psi[,je]<-psi[,je]
    new_psi[,1]<-psi[,1]
    new_psi[ie,]<-psi[ie,]
    new_psi[1,]<-psi[1,]
    je=je-1
    ie=ie-1
    for (i in is:ie){
        for (j in js:je){
            new_psi[i,j]<-a1*psi[i+1,j] + a2*psi[i-1,j] + a3*psi[i,j+1] +
a4*psi[i,j-1] - a5*forf[i,j]
            gdiff<<-gdiff+abs(new_psi[i,j]-psi[i,j])
            }
    }
    return(new_psi)
}
```

R Version - Main Loop

```
iout=steps/100
if(iout  == 0){iout=1}
tymer(reset=T)
for(i in 1:steps){
   psi<-do_jacobi(psi)
   if(myid == 0){
         if (((i) %% iout) == 0){
         print(paste(i,gdiff,tymer()))
         }
   }
}



writeBin(as.vector(psi),"stom00.out")
```

MPI version of the Serial Code

Timothy H. Kaiser, Ph.D.

Overview

We will choose one of the two dimensions and subdivide the domain to allow the distribution of the work across a group of distributed memory processors

We will focus on the principles and techniques used to do the MPI work in the model

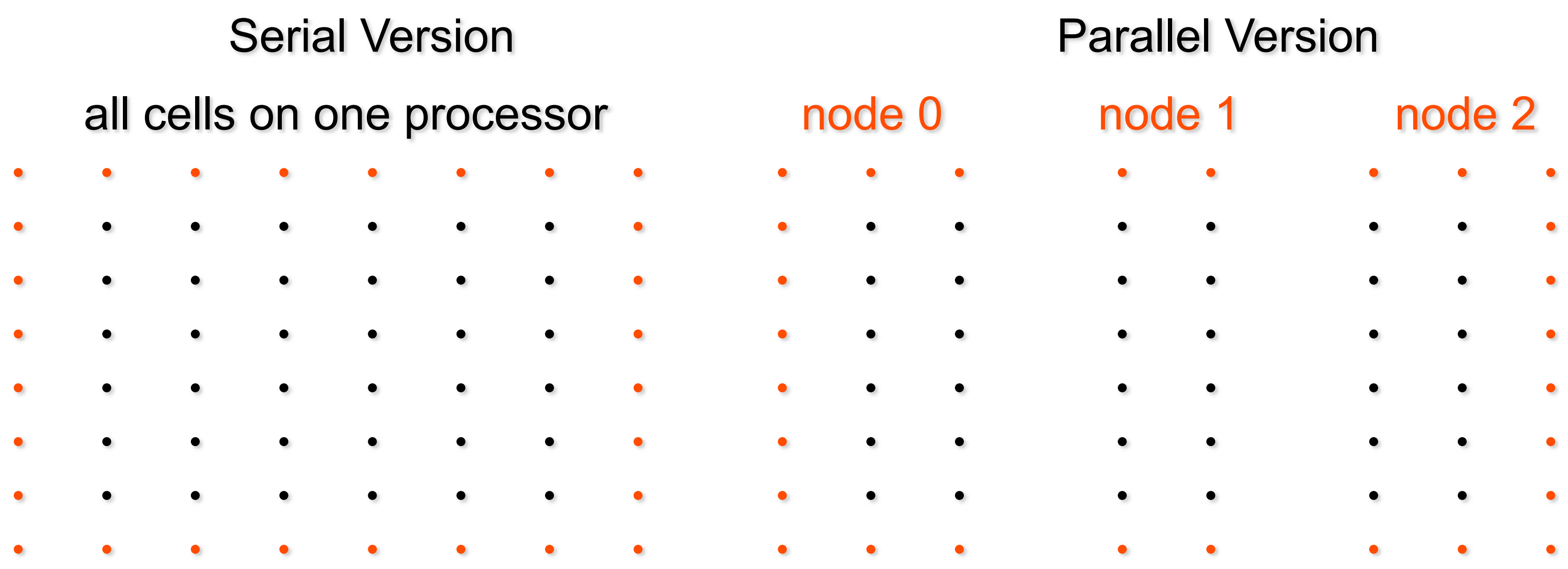
We will discuss the Rmpi version of the program there are C, PYthon, and Fortran versions also

Domain Decomposition (1d)

- We set our array bounds on each processor so that:
  - We take our original grid and break it into numnodes subsections of size nx/numnodes
  - Each processor calculates for a different subsection of the grid
  - No two processors calculate psi for the same (I,J)
- We add special boundary cells for each subsection of the grid called ghost cells

Domain Decomposition (1d)

Physical domain is sliced into sets of columns so that computation in each set of columns will be handled by different

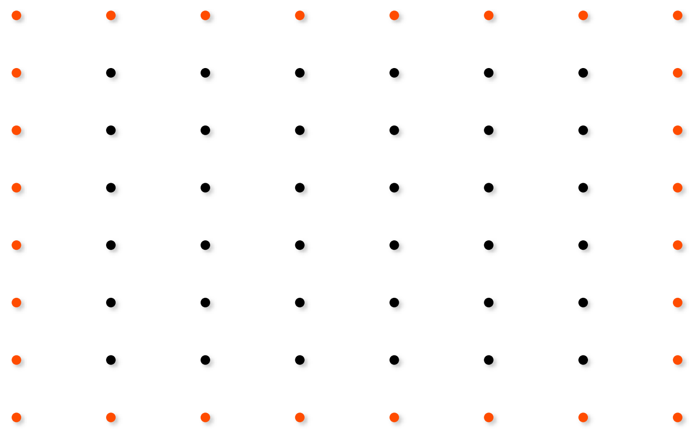


Domain Decomposition (1d)

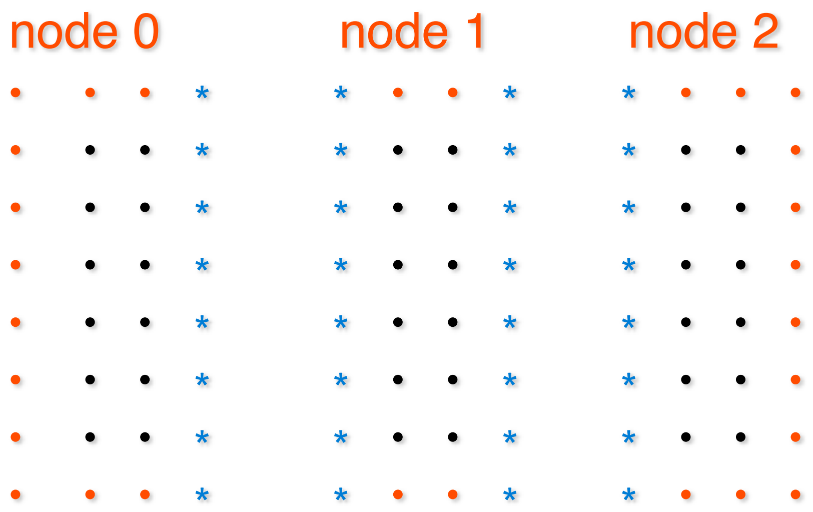
With ghost cells our decomposition becomes...

Serial Version

all cells on one processor



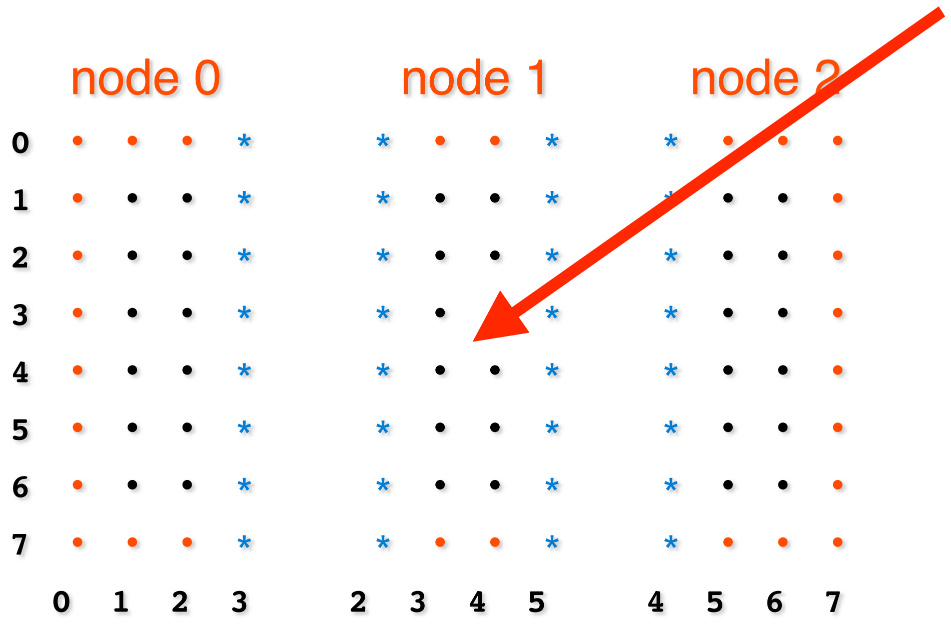
Parallel Version



Domain Decomposition (1d)

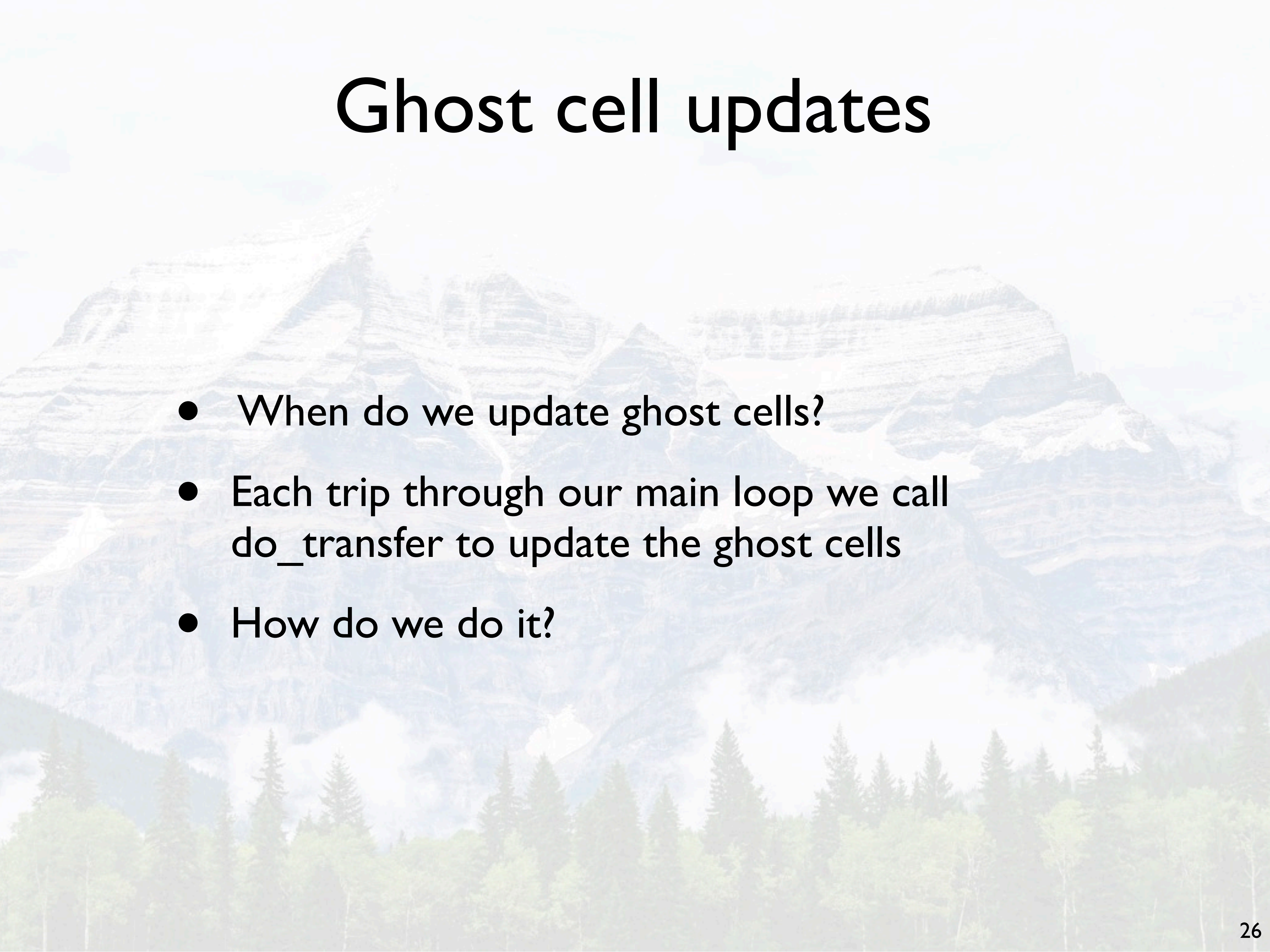
How and why are ghost cells used?

Node 0 allocates space for psi(0:7,0:3) but calculates psi(1:6,1,2)
Node 1 allocates space for psi(0:7,2:5) but calculates psi(1:6,3,4)
Node 2 allocates space for psi(0:7,4:7) but calculates psi(1:6,5,6)



To calculate the value for psi(4,4) node1 requires the value from psi(4,3),psi(5,4),psi(3,4),psi(4,5)

Where does it get the value for psi(4,5)? From node2, and it holds the value in a ghost cell



Ghost cell updates

- When do we update ghost cells?
- Each trip through our main loop we call do_transfer to update the ghost cells
- How do we do it?

How do we update ghost cells?

Processors send and receive values to and from neighbors

Need to exchange with left and right neighbors except processors on far left and right only transfer in 1 direction

Trick 1 to avoid deadlock:

Even # processors	Odd # processors
send left	receive from right
receive from left	send to right
send right	receive for left
receive from right	send to left

Trick 2 to handle the end processors
Send to MPI_PROC_NULL instead of a real processor

How do we update ghost cells? It's a 4-stage operation

Example with 4 nodes:

	Proc 0	Proc 1	Proc 2	Proc 3
Stage 1	Send left to MPI_PROC_NULL	Receive right from Proc 2	Send left to Proc 1	Receive right from MPI_PROC_NULL
Stage 2	Receive left from MPI_PROC_NULL	Send right to Proc 2	Receive left from Proc 1	Send right to MPI_PROC_NULL
Stage 3	Receive right from Proc 1	Send right to Proc 0	Receive right from Proc 3	Send right to Proc 2
Stage 4	Send right to Proc 1	Receive right from Proc 0	Send right to Proc 3	Receive right from Proc 2

do_transfer

```
do_transfer<-function(psi,i1,i2,j1,j2){

    mystat=0

    myid <<- mpi.comm.rank(comm=mpi_comm_world)

    myleft<-myid-1

    myright<-myid+1

    if(myleft <= -1){myleft<-(-1)}

    if(myright >= numprocs){myright=(-1)}

    lc=dim(psi)[2]

    if(is.even(myid)){
        ...
    }else{
        ...
    }
return(psi)

}
```

STEP1& 2: introduce the MPI environment and read data

```
mpi_comm_world<-0
myid <<- mpi.comm.rank(comm=mpi_comm_world)
numprocs <<- mpi.comm.size(comm=mpi_comm_world)
myname <- mpi.get.processor.name()

if (myid != 0) {
    nx<-0
    ny<-0
    lx<-0.0
    ly<-0.0
    alpha<-0.0
    beta<-0.0
    gamma<-0.0
    steps=0
}else{
    ngrid<-scan("st.in",integer(),2)
    grid<-scan("st.in",double(),2,skip=1)
    const<-scan("st.in",double(),3,skip=2)
    steps<-scan("st.in",integer(),1,skip=3)
    #bcast should be here
    nx<-ngrid[1]
    ny<-ngrid[2]
    lx<-grid[1]
    ly<-grid[2]
    alpha<-const[1]
    beta<-const[2]
    gamma<-const[3]
}
```

STEP 3: bcast the data

```
##  mpi.bcast(nx,    type=1,comm = mpi_comm_world)
nx<-mpi.bcast(nx,    type=1,comm = mpi_comm_world)
##  mpi.bcast(ny,    type=1,comm = mpi_comm_world)
ny<-mpi.bcast(ny,    type=1,comm = mpi_comm_world)
##     mpi.bcast(steps, type=1,comm = mpi_comm_world)
steps<-mpi.bcast(steps, type=1,comm = mpi_comm_world)
lx<-mpi.bcast(lx,    type=2,comm = mpi_comm_world)
ly<-mpi.bcast(ly,    type=2,comm = mpi_comm_world)
alpha<-mpi.bcast(alpha, type=2,comm = mpi_comm_world)
beta<-mpi.bcast(beta,  type=2,comm = mpi_comm_world)
gamma<-mpi.bcast(gamma, type=2,comm = mpi_comm_world)
print(paste(myid,nx,ny,steps,lx,ly,alpha,beta,gamma))
```

STEP4 : Set the constants

```
i1org<-1

i2org<-nx

j1org<-1

j2org<-ny

i1<-1

i2<-nx

j1<-1

j2<-ny

dj<-as.double(j2)/as.double(numprocs)

j1<-round(1.0+myid*dj)

j2<-round(1.0+(myid+1)*dj)-1
```

```
dx<<-lx/(nx+1.0)

dy<<-ly/(ny+1.0)

dx<<-lx/(nx+1.0)

dx2<-dx*dx

dy2<-dy*dy

bottom<-2.0*(dx2+dy2)

a1<<-(dy2/bottom)+(beta*dx2*dy2)/(2.0*gamma*dx*bottom)

a2<<-(dy2/bottom)-(beta*dx2*dy2)/(2.0*gamma*dx*bottom)

a3<<-dx2/bottom

a4<<-dx2/bottom

a5<<-dx2*dy2/(gamma*bottom)

a6<<-pi/(ly)

gdiff<<-0.0
```

Step 5: Set up

```
psi<-matrix(1.0,nrow=((i2-i1)+3),ncol=((j2-j1)+3))

print(paste(myid,"covers",i1,i2,j1,j2))

psi<-bc(psi)

new_psi<<-matrix(0.0,nrow=((i2-i1)+3),ncol=((j2-j1)+3))
```

```
bc<-function(psi,i1,i2,j1,j2,nx,ny){
    myid <<-0
    numprocs <<- 1
    myleft<-myid-1
    myright<-myid+1
    if(myleft <= -1){myleft<-(-1)}
    if(myright >= numprocs){myright=(-1)}
    if(myleft == -1){
        psi[,1]=0.0

    }
    if(myright == -1){
        psi[,ncol(psi)]=0.0
    }
        psi[1,]=0.0
        psi[nrow(psi),]=0.0
    return(psi)
}
```

```
bc<-function(psi,i1,i2,j1,j2,nx,ny){
    myid <<- mpi.comm.rank(comm=mpi_comm_world)
    numprocs <<- mpi.comm.size(comm=mpi_comm_world)
    myleft<-myid-1
    myright<-myid+1
    if(myleft <= -1){myleft<-(-1)}
    if(myright >= numprocs){myright=(-1)}
    if(myleft == -1){
        psi[,1]=0.0

    }
    if(myright == -1){
        psi[,ncol(psi)]=0.0
    }
        psi[1,]=0.0
        psi[nrow(psi),]=0.0
    return(psi)
}
```

Residual

- In our serial program, the routine do_jacobi calculates a residual for each iteration
- The residual is the sum of changes to the grid for a jacobi iteration
- Now the calculation is spread across all processors
- To get the global residual, we can use the MPI_Reduce function

Final Results

- In our parallel program, the grid is spread across all processors
- Each processor can print its portion of the grid
- Regular MPI has special parallel IO routines (difficult)
- Here we "just" bring all of the data back to process 0 and save it

Our Main Loop

```
iout=steps/100
if(iout  == 0){iout=1}
if(myid == 0){tymer(reset=T)}
for(i in 1:steps){
    psi<-do_transfer(psi,i1,i2,j1,j2)
    psi<-do_jacobi(psi,i1,i2,j1,j2)
    mydiff<-gdiff
    todiff<-mpi.reduce(mydiff, type=2, op="sum",dest = 0, comm = mpi_comm_world)
    if (((i) %% iout) == 0){
        print(paste(myid,max(psi),min(psi)))
        if(myid == 0){
            print(paste(i,todiff,tymer()))
        }
    }
}
getresults(psi,nx,ny)

bonk<-mpi.finalize()
```

do_transfer

```
do_transfer<-function(psi,i1,i2,j1,j2){

    mystat=0

    myid <<- mpi.comm.rank(comm=mpi_comm_world)

    myleft<-myid-1

    myright<-myid+1

    if(myleft <= -1){myleft<-(-1)}

    if(myright >= numprocs){myright=(-1)}

    lc=dim(psi)[2]

    if(is.even(myid)){
        ...
    }else{
        ...
    }
return(psi)

}
```

do_transfer

```
    if(is.even(myid)){
# we are on an even col processor
# send to left
        if(myleft != -1){
            #print(paste(myid,"to",myleft))
            mpi.send(psi[,2], type=2, dest=myleft, tag=100,  comm = mpi_comm_world)
# rec from left
            #print(paste(myid,"from",myleft))
            psi[,1]<-mpi.recv(psi[,1], type=2,source=myleft,tag=100,comm = mpi_comm_world,status = mystat)
        }
# rec from right
        if(myright != -1){
            #print(paste(myid,"from",myright))
            psi[,lc]<-mpi.recv(psi[,lc],type=2,source=myright, tag=100,comm = mpi_comm_world,status = mystat)
# send to right
            #print(paste(myid,"to",myright))
            mpi.send(psi[,lc-1], type=2, dest=myright, tag=100,  comm = mpi_comm_world)

            }

    }
```

do_transfer

```
else{
# we are on an odd col processor
# rec from right
        if(myright != -1){
            psi[,lc]<-mpi.recv(psi[,lc],type=2,source=myright,tag=100,comm = mpi_comm_world, status = mystat)

# send to right
            mpi.send(psi[,lc-1],type=2,dest=myright,tag=100,comm = mpi_comm_world)
        }
# send to left
        if(myleft != -1){
            mpi.send(psi[,2], type=2, dest=myleft, tag=100,  comm = mpi_comm_world)

# rec from left
            psi[,1]<-mpi.recv(psi[,1],type=2,source=myleft,tag=100,comm = mpi_comm_world, status = mystat)

        }

    }
```

getresults

```
getresults<-function(psi,nx,ny){
    myid <<- mpi.comm.rank(comm=mpi_comm_world)
    tosend=(nrow(psi)-2)*(ncol(psi)-2)
    rcounts<-vector("integer",numprocs)
    rcounts<-mpi.gather(tosend, 1, rcounts,root = 0, comm = mpi_comm_world)
    typeof(rcounts)
    typeof(psi)
    tot<-sum(rcounts)
    full<-vector("double",tot)
    full<-mpi.gatherv(psi[(2:(nrow(psi)-1)),(2:(ncol(psi)-1))] ,2, full,
                                            rcounts, root = 0, comm = mpi_comm_world)
    if(myid == 0){writeBin(full,"stom01.out")}

}
```

```
    #install.packages("plotly", lib="/Library/Frameworks/R.framework/Versions/3.6/Resources/library")
    #result<-readBin("stom01.out",double(),10000)
    #grid<-matrix(result,100,100)
    #library(plotly)
    #plot_ly(z =grid)
```