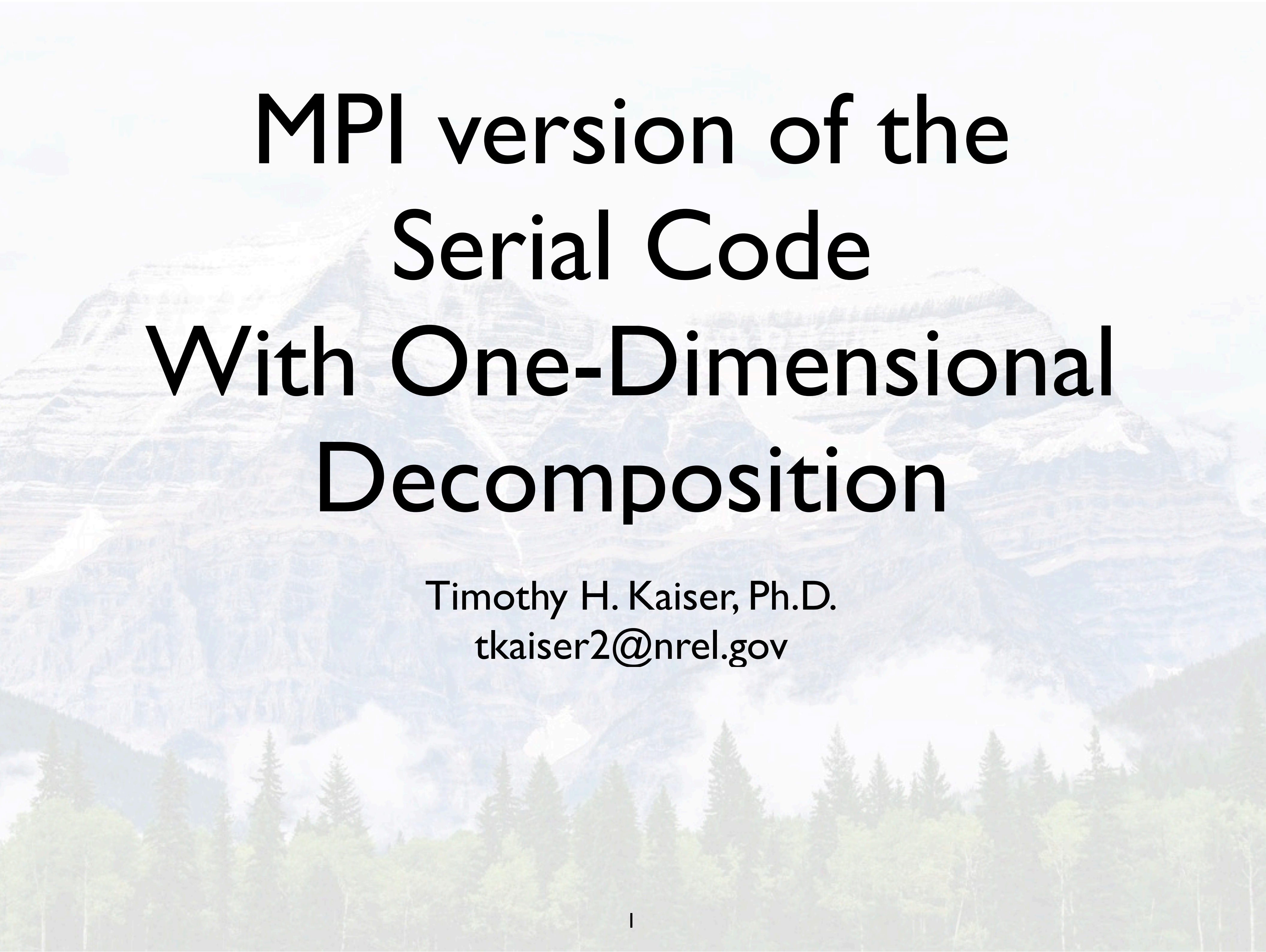


MPI version of the Serial Code With One-Dimensional Decomposition

Timothy H. Kaiser, Ph.D.
tkaiser2@nrel.gov

Slides at:

https://github.com/timkphd/slides

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 mpi4py version of the program there are C and Fortran versions also

Python is !!!!MUCH!!!! slower

- Our “big” example calculation, scp.py/stc_03.c
    - Python 4720 seconds
    - C/Fortran 3 seconds
- You can however, call C and Fortran compiled subroutines from Python
- You can mix Python and C/Fortran in MPMD fashion
    - MPI tasks 0 to n-2 might be Fortran or C
    - MPI task n-1 could be a Python graphics program
        - Have example
        - mpiexec -n 3 ./ccalc : -n 1 pwrite.py < small.in

mpi4py Examples at

https://petra.acns.colostate.edu/docs/examples/

To just get these examples:

```
mkdir examples
cd examples
curl https://petra.acns.colostate.edu/docs/examples/mpi/mpi4py/mpi4py.tgz | tar -xz
```

STEP1: introduce the MPI environment

- Need to include “mpif.h” or use mpi to define MPI constants
- Need to define our own constants
    - numnodes - how many processors are running
    - myid - Which processor am I
    - mpi_err - error code returned by most calls
    - mpi_master - the id for the master node

STEP1: introduce the MPI environment

```
from math import pi,sin
from math import fabs as abs
from numpy import empty
import numpy
from time import time as walltime
global vals,cons
global psi,new_psi,forf
import sys
global a1,a2,a3,a4,a5,a6,dx,dy
global r1,r2
global ttol
from write_grid import *
from copy import deepcopy
#http://mpi4py.scipy.org/docs/apiref/frames.html
#http://mpi4py.scipy.org/docs/usrman/tutorial.html
from mpi4py import MPI
. . .
```

STEP1: Start the MPI environment

We add the following to start MPI:

```
if __name__ == '__main__':
# do init
   global numnodes,myid,mpi_err
   comm=MPI.COMM_WORLD
   myid=comm.Get_rank()
   numnodes=comm.Get_size()
   name = MPI.Get_processor_name()
```

And the following to shut it down:

```
def myquit(mes):
   MPI.Finalize()
   print(mes)
   sys.exit()
```

Input

We read the data on processor 0 and send to the others

```
if (myid == 0):

    vals=input()
    vals.Read()
  else:
    vals=input()
  vals.nx=comm.bcast(vals.nx, root=0)
  vals.ny=comm.bcast(vals.ny, root=0)
  vals.lx=comm.bcast(vals.lx, root=0)
  vals.ly=comm.bcast(vals.ly, root=0)
  vals.alpha=comm.bcast(vals.alpha, root=0)
  vals.beta=comm.bcast(vals.beta, root=0)
  vals.gamma=comm.bcast(vals.gamma, root=0)
  vals.steps=comm.bcast(vals.steps, root=0)
```

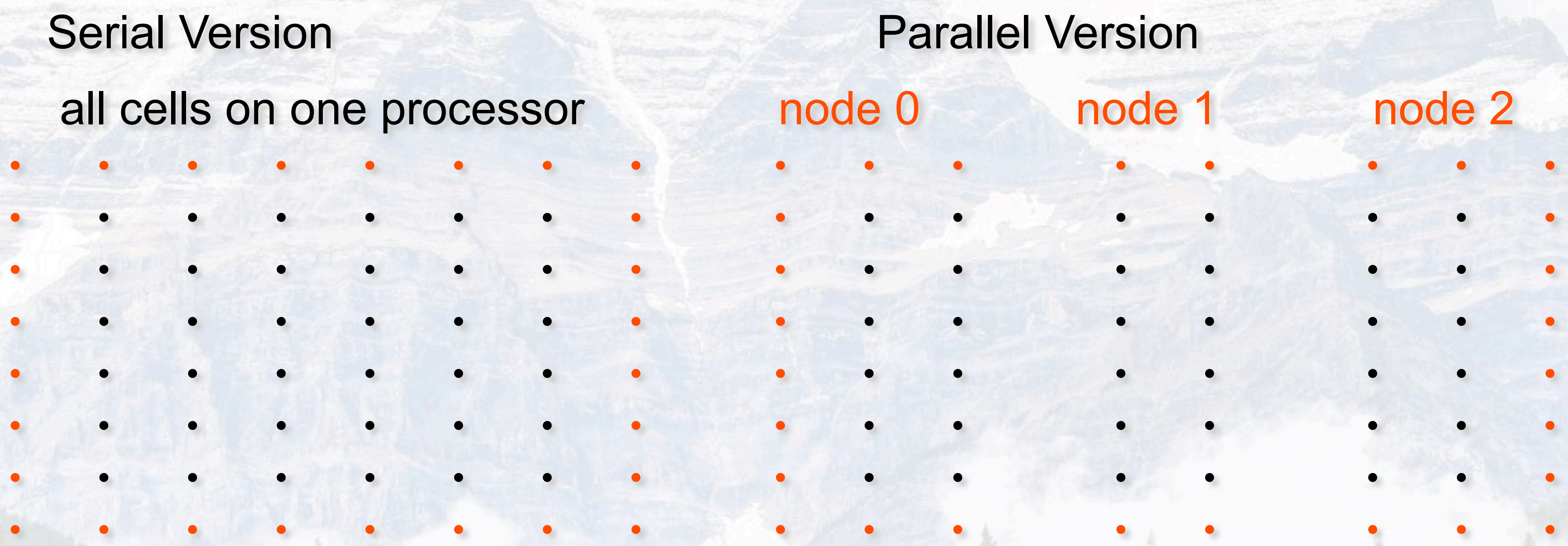
We use MPI_BCAST to send the data to the other processors

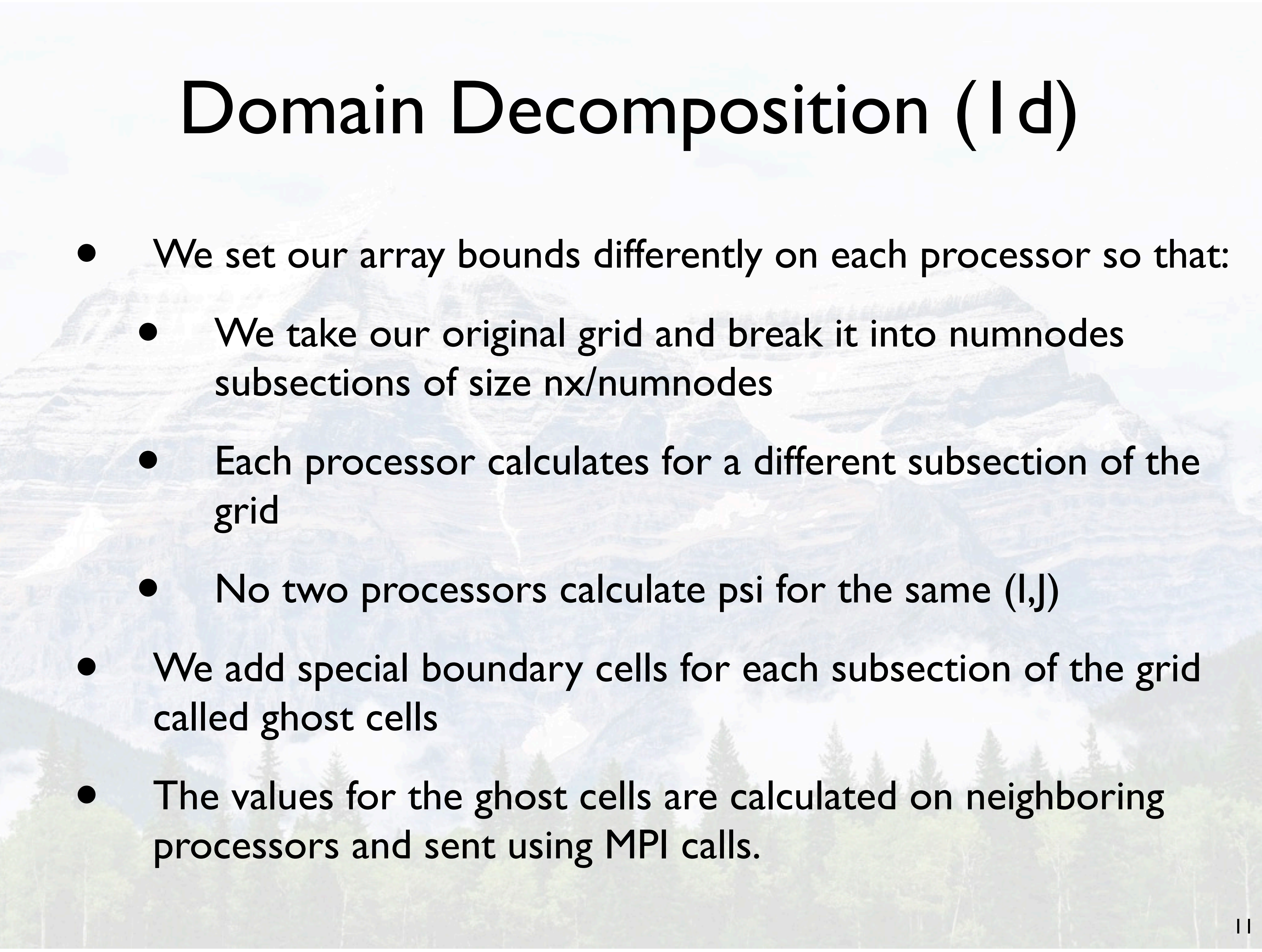
We use 8 calls

Can you do it in 2?

Domain Decomposition (1d)

Physical domain is sliced into sets of columns so that computation in each set of columns will be handled by different processors. Why do columns and not rows?



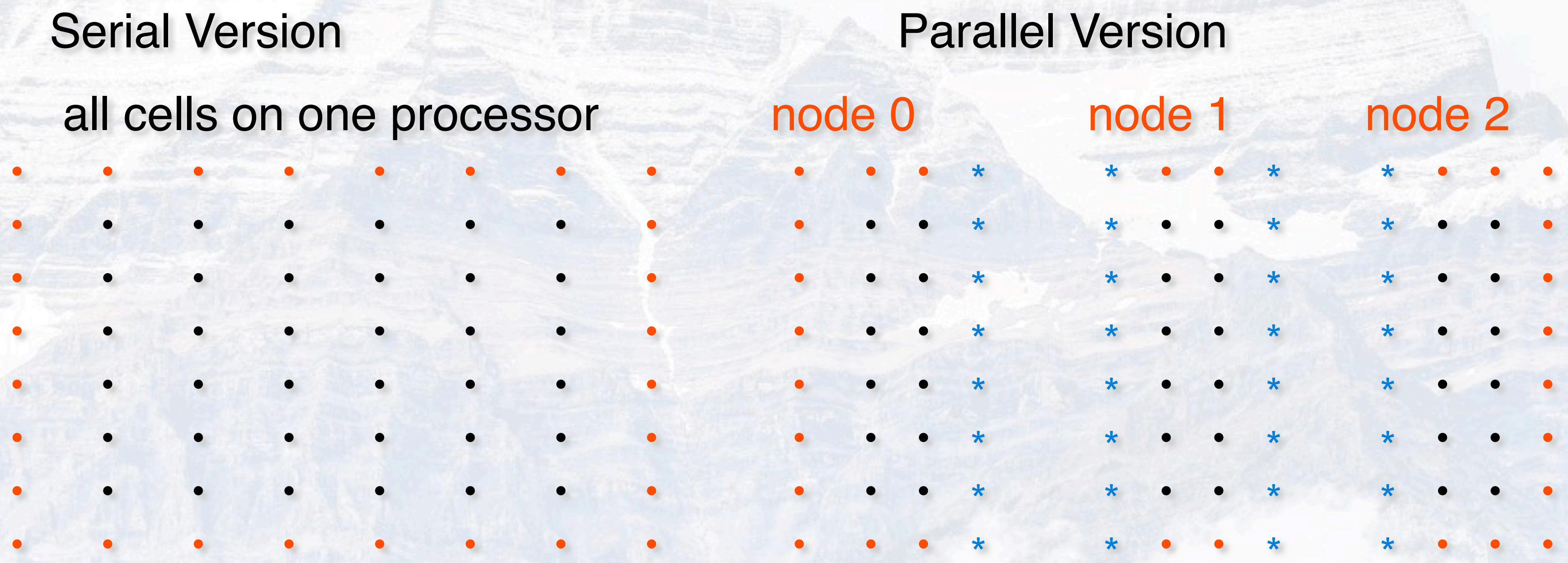


Domain Decomposition (1d)

- We set our array bounds differently 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
- The values for the ghost cells are calculated on neighboring processors and sent using MPI calls.

Domain Decomposition (1d)

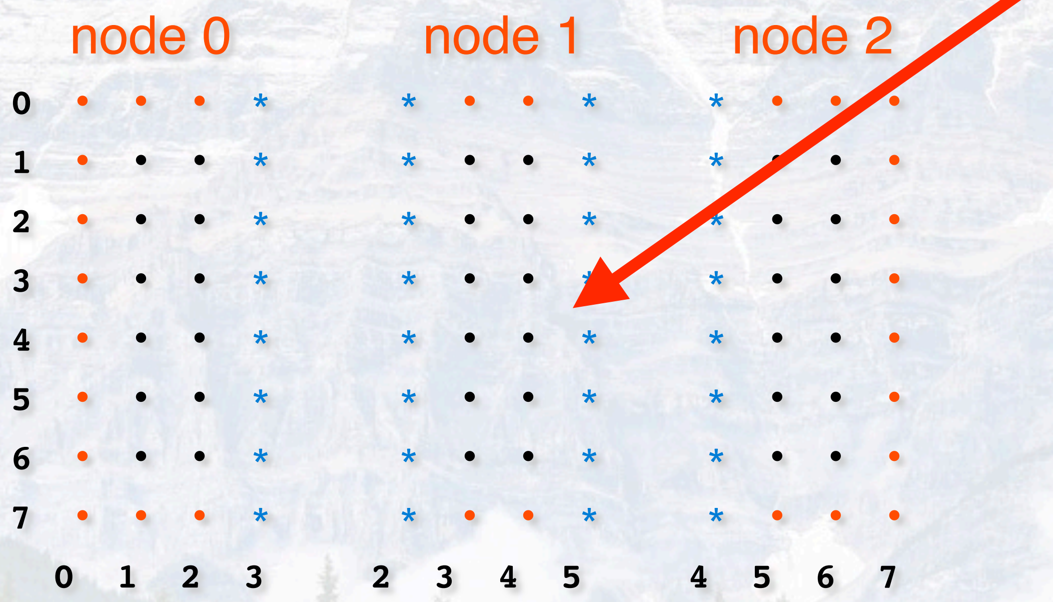
With ghost cells our decomposition becomes...



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)

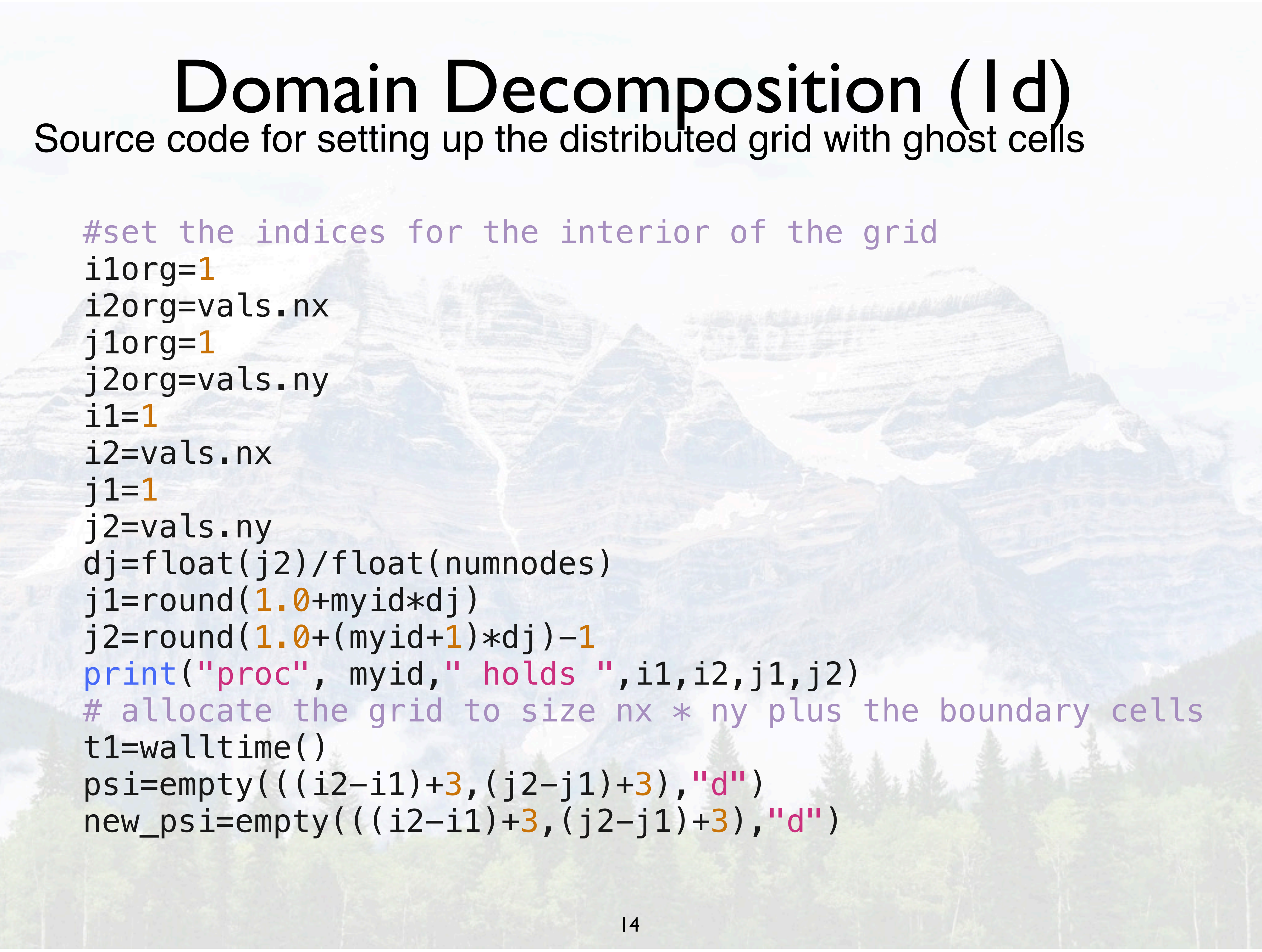


To calculate the value for psi(4,4) node1 requires the value from psi(4,3),psi(5,4),psi(3,4),psi(4,5)

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

Domain Decomposition (1d)

Source code for setting up the distributed grid with ghost cells



```
#set the indices for the interior of the grid
i1org=1
i2org=vals.nx
j1org=1
j2org=vals.ny
i1=1
i2=vals.nx
j1=1
j2=vals.ny
dj=float(j2)/float(numnodes)
j1=round(1.0+myid*dj)
j2=round(1.0+(myid+1)*dj)-1
print("proc", myid," holds ",i1,i2,j1,j2)
# allocate the grid to size nx * ny plus the boundary cells
t1=walltime()
psi=empty(((i2-i1)+3,(j2-j1)+3),"d")
new_psi=empty(((i2-i1)+3,(j2-j1)+3),"d")
```

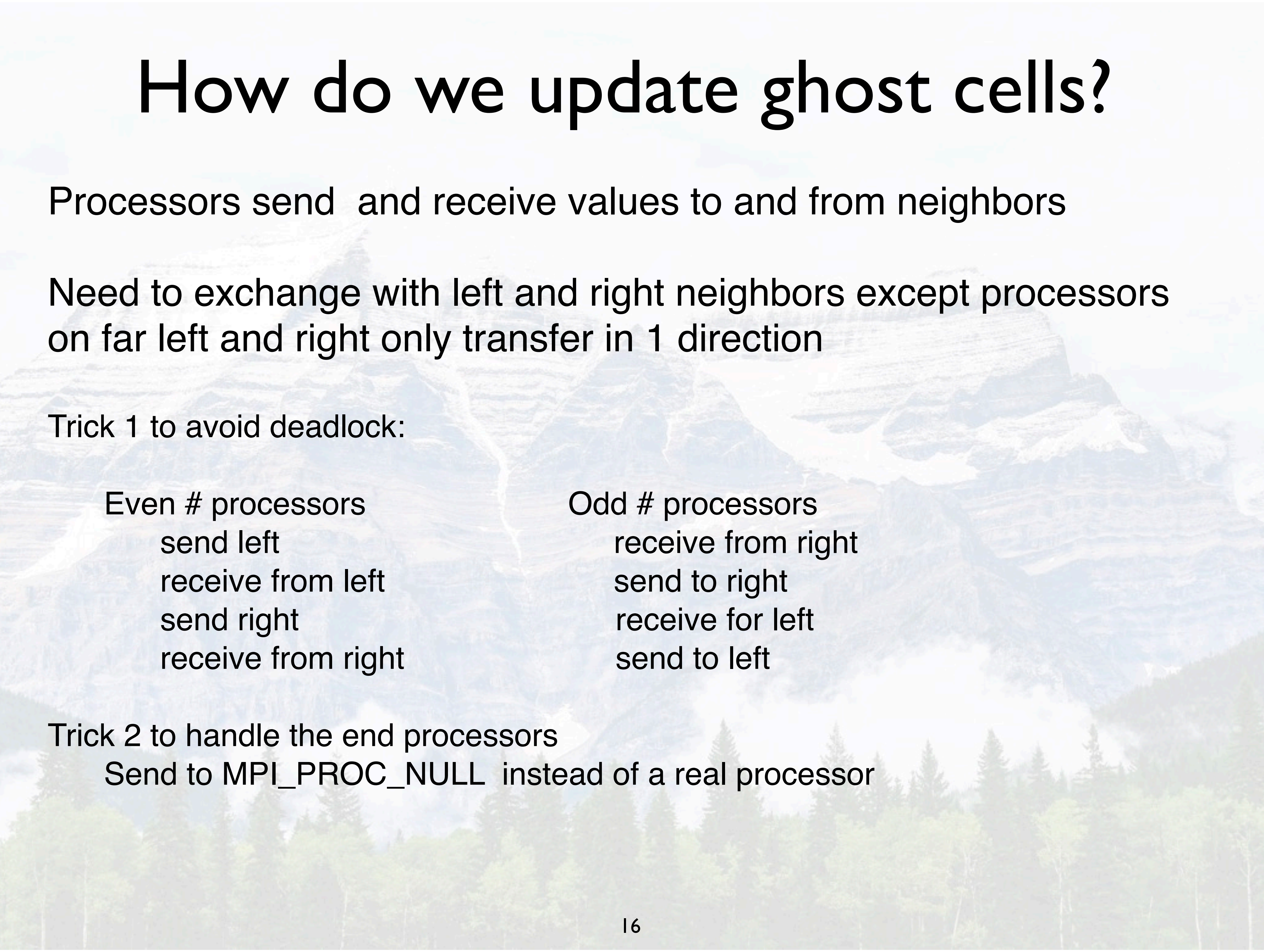
Ghost cell updates

When do we update ghost cells?

Each trip through our main loop we call do_transfer to update the ghost cells

Our main loop becomes...

```
r1=range(1,(i2-i1)+2)
r2=range(1,(j2-j1)+2)
ttot=0
do_transfer(psi,i1,i2,j1,j2)
for i in range(0,vals.steps):
  diff=do_jacobi(psi,new_psi,i1,i2,j1,j2)
  diff=comm.reduce(diff)
  do_transfer(psi,i1,i2,j1,j2)
  if ((i+1) % iout) == 0  and myid == 0:
    print("%8d %18.6e %10.3f" %(i+1,diff,walltime()-t1))
```



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:

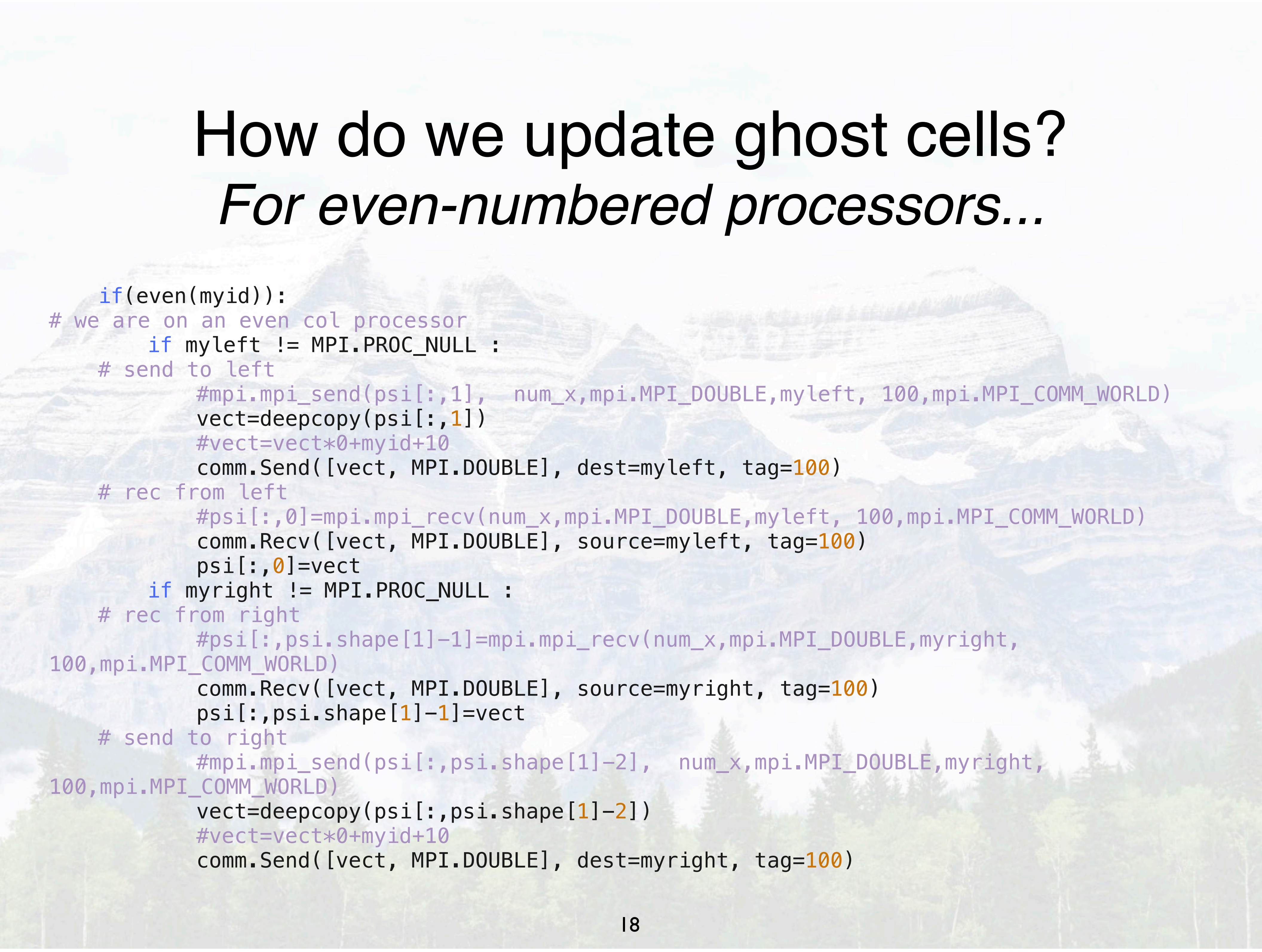
Even # processors
- send left
- receive from left
- send right
- receive from right

Odd # processors
- receive from right
- send to right
- receive for left
- 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?

```
def do_transfer(psi,i1,i2,j1,j2):
  global numnodes,myid,mpi_err
  num_x=i2-i1+3
  myleft=myid-1
  myright=myid+1
  if(myleft <= -1):
    myleft=MPI.PROC_NULL
  if(myright >= numnodes):
    myright=MPI.PROC_NULL
  vlen=psi[:,1].shape[0]
  vlen=psi.shape[0]
  vect=empty(vlen,"d")
```



How do we update ghost cells?

For even-numbered processors...

```
    if(even(myid)):
# we are on an even col processor
        if myleft != MPI.PROC_NULL :
    # send to left
            #mpi.mpi_send(psi[:,1],  num_x,mpi.MPI_DOUBLE,myleft, 100,mpi.MPI_COMM_WORLD)
            vect=deepcopy(psi[:,1])
            #vect=vect*0+myid+10
            comm.Send([vect, MPI.DOUBLE], dest=myleft, tag=100)
    # rec from left
            #psi[:,0]=mpi.mpi_recv(num_x,mpi.MPI_DOUBLE,myleft, 100,mpi.MPI_COMM_WORLD)
            comm.Recv([vect, MPI.DOUBLE], source=myleft, tag=100)
            psi[:,0]=vect
        if myright != MPI.PROC_NULL :
    # rec from right
            #psi[:,psi.shape[1]-1]=mpi.mpi_recv(num_x,mpi.MPI_DOUBLE,myright,
100,mpi.MPI_COMM_WORLD)
            comm.Recv([vect, MPI.DOUBLE], source=myright, tag=100)
            psi[:,psi.shape[1]-1]=vect
    # send to right
            #mpi.mpi_send(psi[:,psi.shape[1]-2],  num_x,mpi.MPI_DOUBLE,myright,
100,mpi.MPI_COMM_WORLD)
            vect=deepcopy(psi[:,psi.shape[1]-2])
            #vect=vect*0+myid+10
            comm.Send([vect, MPI.DOUBLE], dest=myright, tag=100)
```

How do we update ghost cells?

For odd-numbered processors...

```
    else:
# we are on an odd col processor
        if myright != MPI.PROC_NULL :
    # rec from right
            #psi[:,psi.shape[1]-1]=mpi.mpi_recv(num_x,mpi.MPI_DOUBLE,myright,
100,mpi.MPI_COMM_WORLD)
            comm.Recv([vect, MPI.DOUBLE], source=myright, tag=100)
            psi[:,psi.shape[1]-1]=vect

    # send to right
            #mpi.mpi_send(psi[:,psi.shape[1]-2],  num_x,mpi.MPI_DOUBLE,myright,
100,mpi.MPI_COMM_WORLD)
            vect=deepcopy(psi[:,psi.shape[1]-2])
            #vect=vect*0+myid+10
            comm.Send([vect, MPI.DOUBLE], dest=myright, tag=100)
        if myleft != MPI.PROC_NULL :
    # send to left
            #mpi.mpi_send(psi[:,1],  num_x,mpi.MPI_DOUBLE,myleft, 100,mpi.MPI_COMM_WORLD)
            vect=deepcopy(psi[:,1])
            #vect=vect*0+myid+10
            comm.Send([vect, MPI.DOUBLE], dest=myleft, tag=100)
    # rec from left
            #psi[:,0]=mpi.mpi_recv(num_x,mpi.MPI_DOUBLE,myleft, 100,mpi.MPI_COMM_WORLD)
            comm.Recv([vect, MPI.DOUBLE], source=myleft, tag=100)
            psi[:,0]=vect
```

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

Only a few other modifications

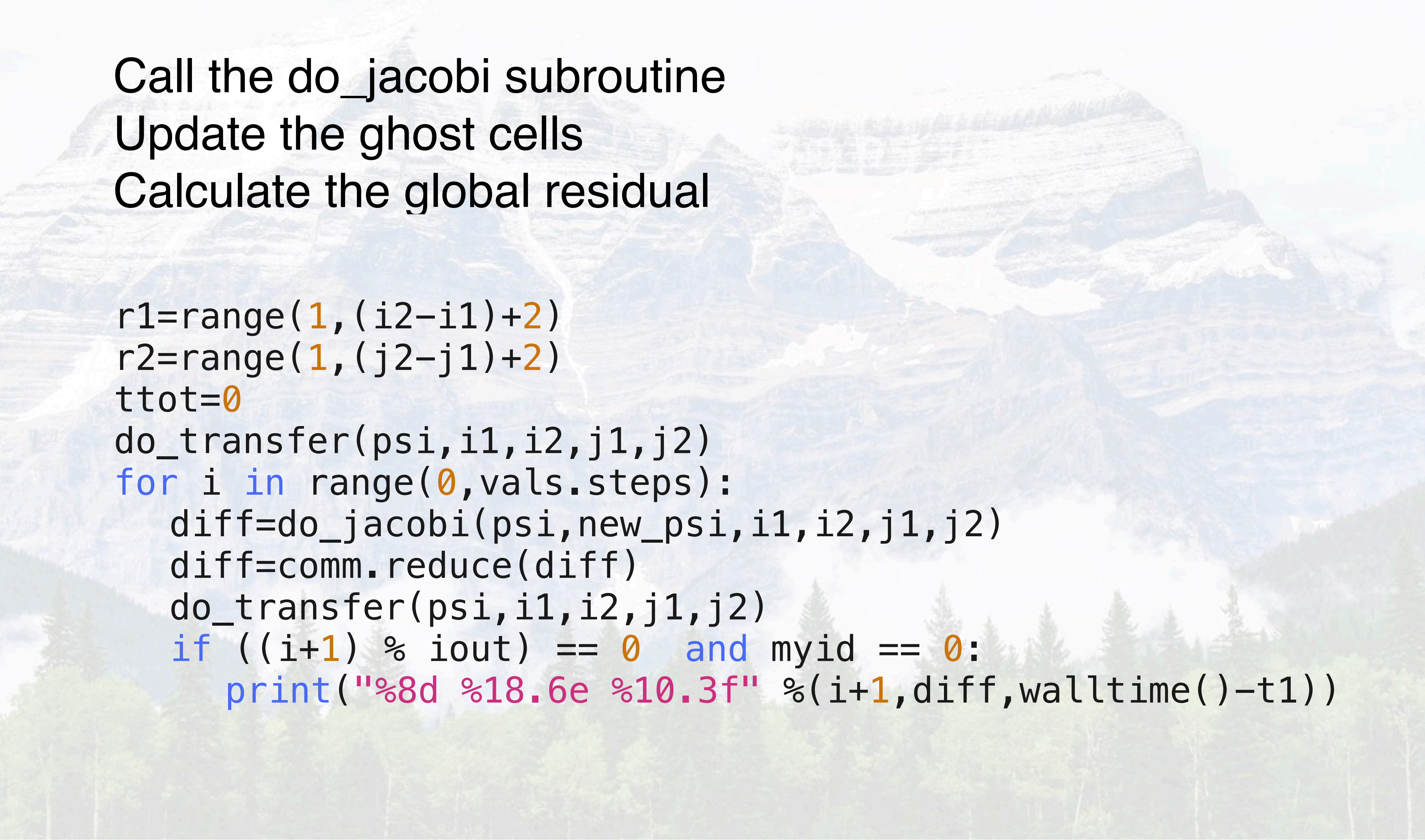
Force and do_jacobi are not modified
We modify the boundary condition routine only
to set value for true boundaries and ignore ghost cells

```
def bc(psi,i1,i2,j1,j2):
    global cons,vals
    if (i1 == 1):
        psi[i1-1,:]=0.0
    if (i2 == vals.ny):
        psi[psi.shape[0]-1,:]=0.0
    if (j1 == 1):
        psi[:,j1-1]=0.0
    if (j2 == vals.nx):
        psi[:,psi.shape[1]-1]=0.0
```

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

Our main loop is now...



Call the do_jacobi subroutine
Update the ghost cells
Calculate the global residual

```
r1=range(1,(i2-i1)+2)
r2=range(1,(j2-j1)+2)
ttot=0
do_transfer(psi,i1,i2,j1,j2)
for i in range(0,vals.steps):
   diff=do_jacobi(psi,new_psi,i1,i2,j1,j2)
   diff=comm.reduce(diff)
   do_transfer(psi,i1,i2,j1,j2)
   if ((i+1) % iout) == 0  and myid == 0:
      print("%8d %18.6e %10.3f" %(i+1,diff,walltime()-t1))
```

Final change (version #1)

We add the write_grid subroutine so that each node writes its part of the grid to a different file.

```python
def write_each(psi,i1, i2, j1, j2,nx,ny,comm):
    from numpy import empty
    from mpi4py import MPI
    myid=comm.Get_rank()
    numnodes=comm.Get_size()
    if(i1==1):
        i0=0
    else :
        i0=i1
    if(i2==nx):
        i3=nx+1
    else :
        i3=i2
    if(j1==1):
        j0=0
    else :
        j0=j1
    if(j2==ny):
        j3=ny+1
    else :
        j3=j2
    fname="out"+str(myid)
    eighteen=open(fname,"w")
```

Final change (version #1)

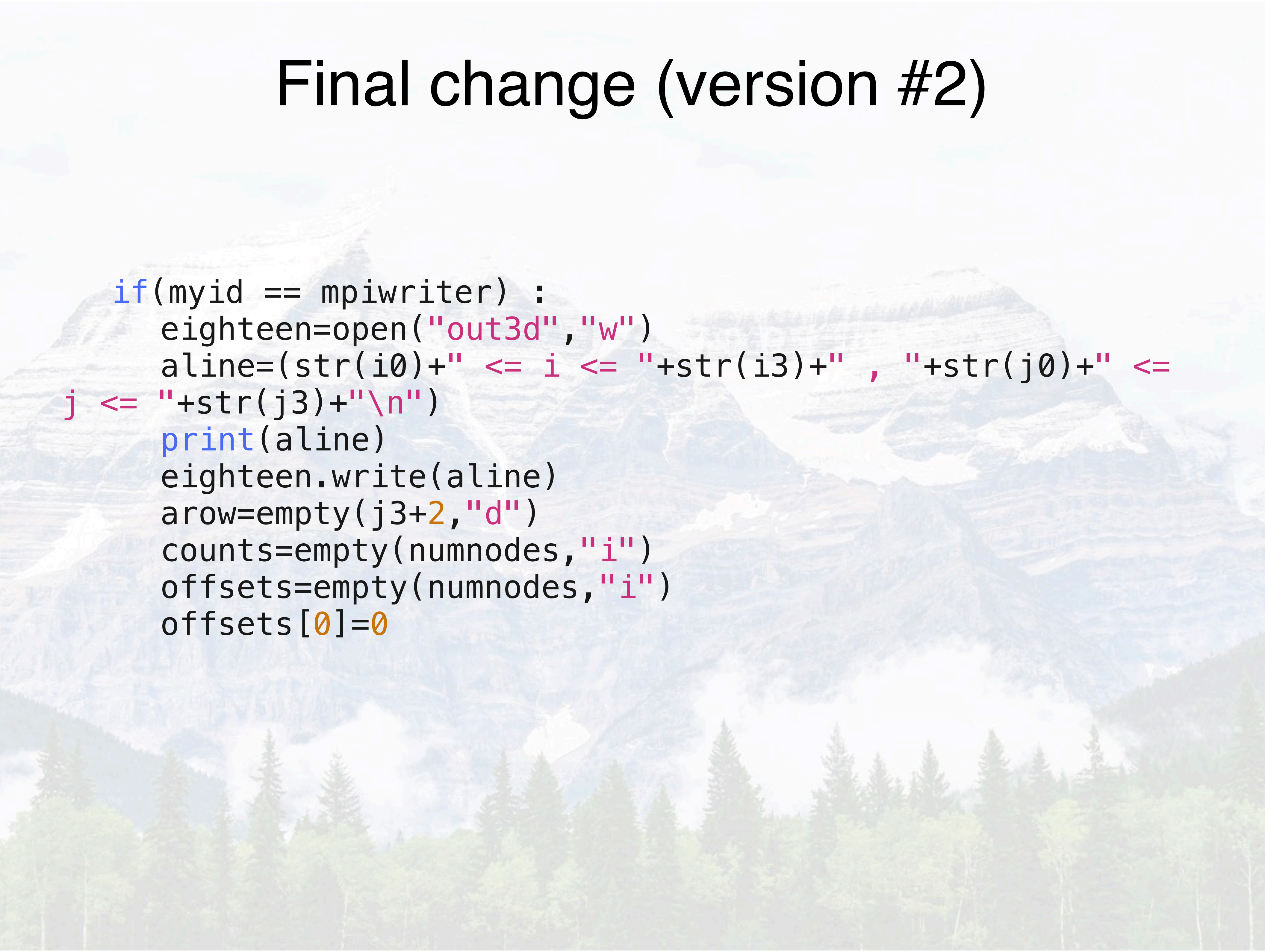
```
aline=("%d %d %d %d %d %d\n" % (i1, i2, j1, j2,nx,ny))
eighteen.write(aline)
aline=(str(psi.shape)+"\n")
eighteen.write(aline)
aline=("%d %d %d %d\n" % (i0, i3+1, j0, j3+1))
eighteen.write(aline)
eighteen.write(str(psi)+"\n")
(imax,jmax)=(psi.shape)
for i in range(0,imax) :
   for j in range(0,jmax) :
     vout=("%18.5f" % (psi[i][j]))
     eighteen.write(vout)
#    eighteen.write(str(psi[i][j]))
     if(j != jmax-1):
        eighteen.write(" ")
   eighteen.write("\n")
eighteen.close()
```

Final change (version #2)

We add the write_one subroutine so that each node sends its grid to the master to write.

```python
def write_one(psi,i1,i2,j1,j2,nx,ny,comm):
# 1-d version -> every processor holds a portion of a line
    from numpy import empty,array
    from mpi4py import MPI
    myid=comm.Get_rank()
    numnodes=comm.Get_size()
    counts=None
    offsets=None
    arow=None
    (id,jd)=psi.shape
    if myid == 0 :
        jstart=0
    else:
        jstart=1
    if myid == (numnodes-1) :
        jend=jd
    else:
        jend=jd-1
    i0=0
    j0=0
    i3=nx+1
    j3=ny+1
    mpiwriter=numnodes-1
```

Final change (version #2)



```
   if(myid == mpiwriter) :
      eighteen=open("out3d","w")
      aline=(str(i0)+" <= i <= "+str(i3)+" , "+str(j0)+" <=
j <= "+str(j3)+"\n")
      print(aline)
      eighteen.write(aline)
      arow=empty(j3+2,"d")
      counts=empty(numnodes,"i")
      offsets=empty(numnodes,"i")
      offsets[0]=0
```

Final change (version #2)

```
   for i in range(0,i3+1):
      dj=jend-jstart
      comm.Gather(sendbuf=[array(dj),1,MPI.INT],
recvbuf=[counts,1,MPI.INT],root=mpiwriter)
      if(myid == mpiwriter):
         for k in range(1,numnodes) :
            offsets[k]=counts[k-1]+offsets[k-1]
      comm.Gatherv(sendbuf=[psi[i,jstart:jend],
(dj),MPI.DOUBLE_PRECISION], recvbuf=[arow, (counts,offsets),
MPI.DOUBLE_PRECISION],root=mpiwriter)
      if(myid == mpiwriter):
         scounts=sum(counts)
         for j in range(0,scounts):
            vout=("%18.5f" % (arow[j]))
            eighteen.write(vout)
            if(j != scounts-1):
               eighteen.write(" ")
         eighteen.write("\n")
      #endif
   #endfor
   if(myid == mpiwriter): eighteen.close()
```

Stommel Code

We have 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. It has 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.
For a description of the Fortran and C versions of this program see:
http://geco.mines.edu/prototype/Show_me_some_local_HPC_tutorials/stoma.pdf
http://geco.mines.edu/prototype/Show_me_some_local_HPC_tutorials/stomb.pdf

The python version, stp.py, follows this C version except it does a 1d decomposition. The C version is 1500x faster than the python version.

pcalc.py and ccalc.c are similar except they create a new communicator that contains N-1 tasks. These tasks do the calculation and pass data to the remaining task to be plotted. Thus we can have "C" do the heavy calculation and python do plotting.

File	**Comment**
ccalc.c	**parallel**
stc_03.c	**parallel**
pcalc.py	**parallel**
stp_00.py	**serial**
stp.py	**parallel**
tiny.in	**tiny input file**
small.in	**small input file**
st.in	**regular input file**

“Extra version”

- Python version is very slow
- Python has nice graphics libraries
- We have a C (and python) version that can run N-1 tasks in C and 1 task in python.
- The “extra” python task outputs plots
- Calling:

```
mpiexec –n 5 ./ccalc : –n 1 ./pwrite.py < st.in
```