MTH 655/659
Large scale scientific computing methods
Overlapping DD: and its parallel implementation

Instructor:

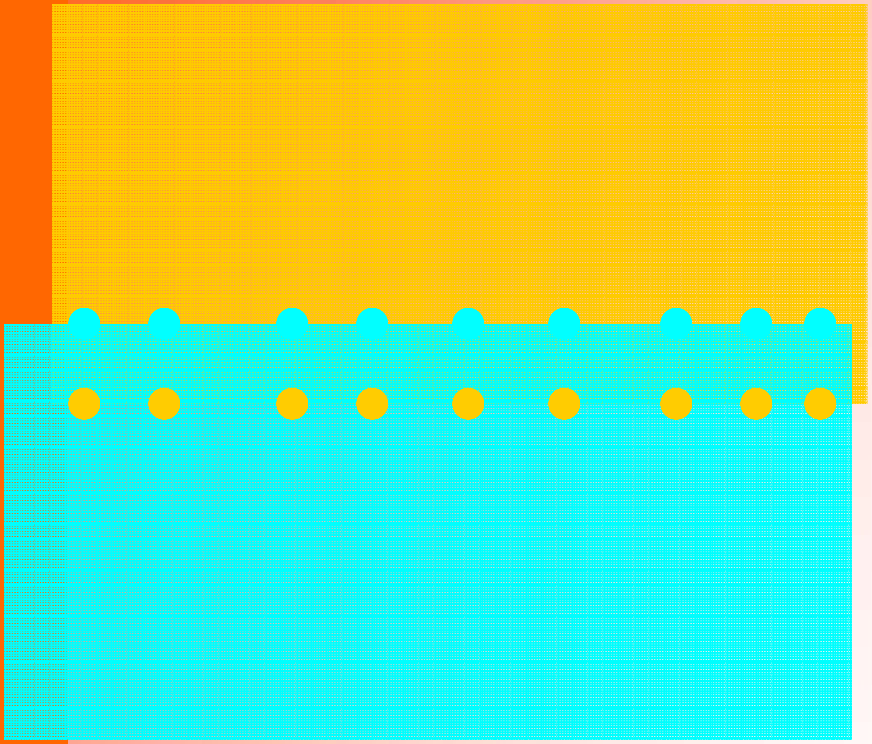
Malgorzata Peszynska

Mathematics

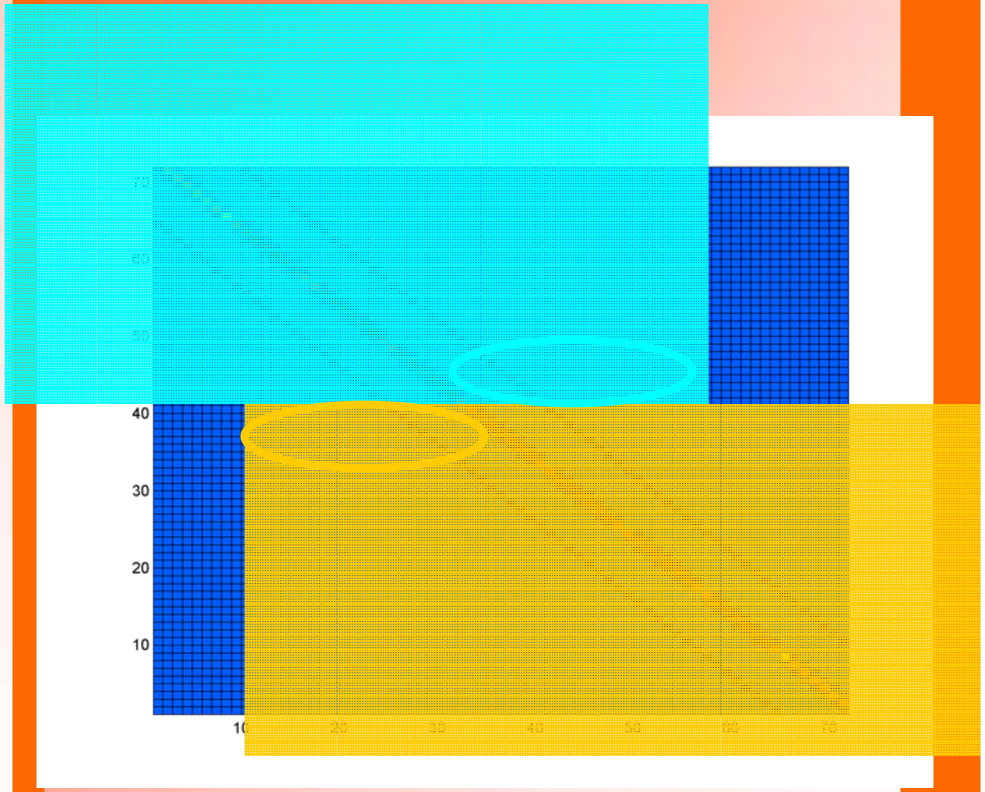
Oregon State University

Domain decomposition: overlapping

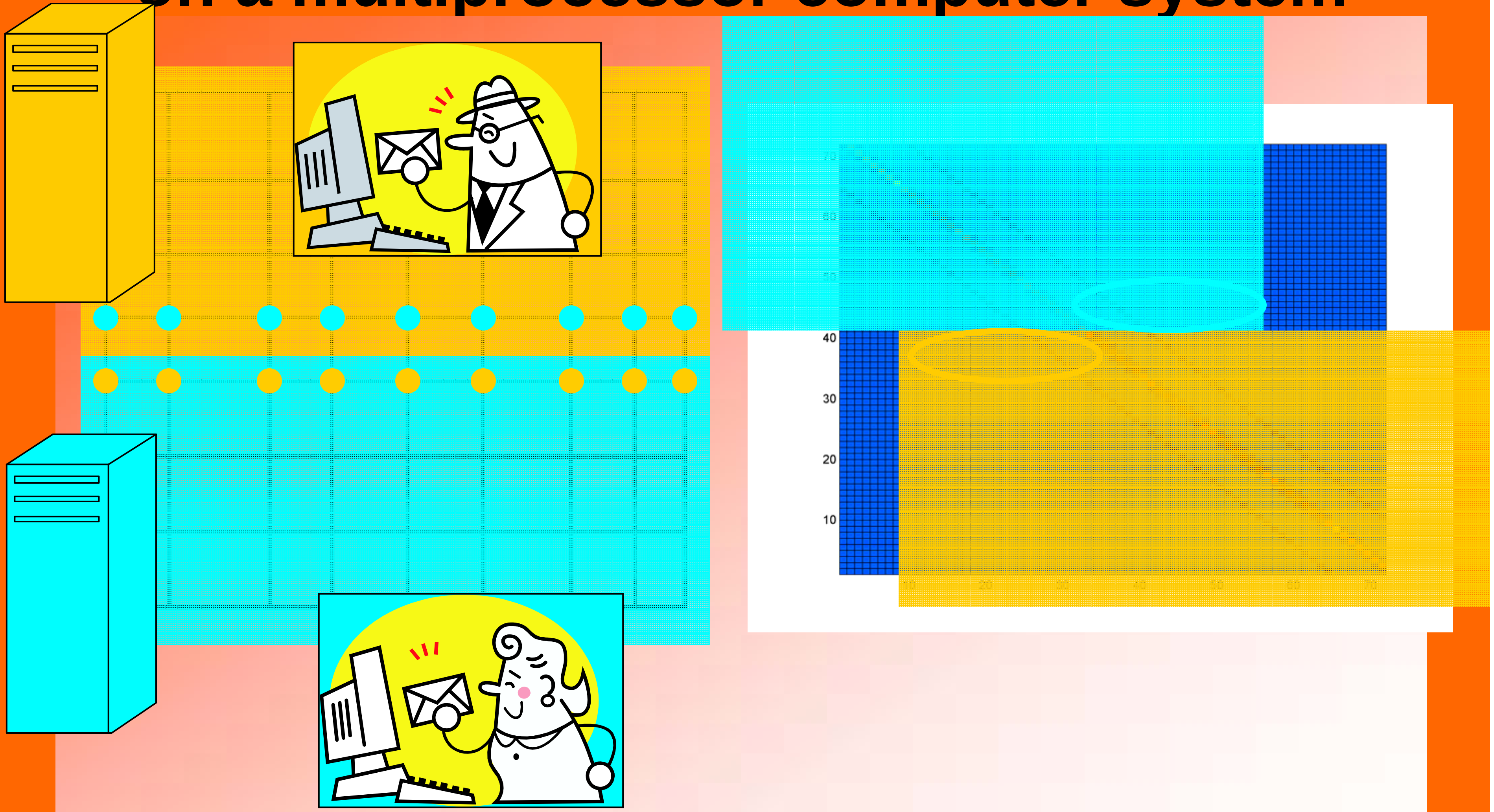
- blue and yellow values lag between iterations



- similar to block Jacobi solver



Solve system $\mathbf{AU} = \mathbf{b}$ on a multiprocessor computer system



- Need to communicate data between processors
    - distributed memory or shared memory ?

function iter = myjacobi (myn,tol,maxiter)

- %% myn = size of problem
- %% tol = tolerance for size of residual
- %% maxiter = max number of iterations acceptable

initialize (could be done outside)

```
n = myn; %% size of global problem
h = 1/(myn+1);    %% n,h are global
myA = zeros (myn,myn);
for i = 1:myn myA(i,i)=2;end;
for i = 2:myn myA(i,i-1)=-1;end;
for i = 1:myn-1 myA(i,i+1)=-1;end;
myb = h * h * ones(myn,1);
%%% initial guess: could be given externally
u0 = zeros(myn,1);
```

overall flow of Jacobi iteration

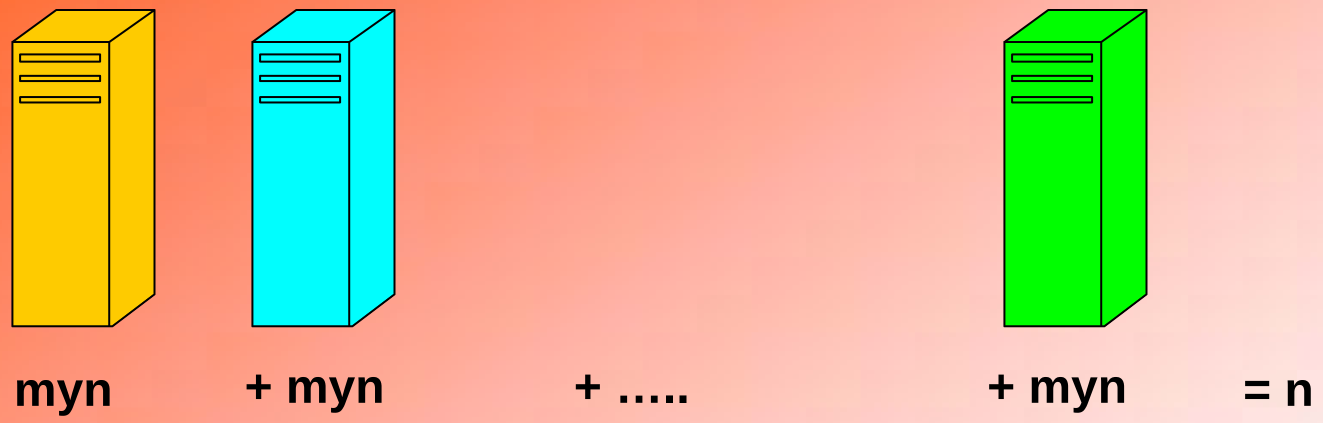
```
u = u0;
for iter = 1:maxiter
        myr = myb - myA*u;
        rerr = norm(myr);
        glob_err = rerr;
        fprintf('%d %g\n',iter,glob_err);

        %%% is it time to stop ?
        if glob_err < tol break;end;
```

perform the update …

```
    for k = 1:myn
        %% collect right hand side terms in every row
        s = 0;
        for j = 1:k-1 s = s + myA(k,j)*u(j);end;
        for j = k+1:myn s = s + myA(k,j)*u(j);end;
      %% compute new guess in row k
        unew (k,1) = (myb(k) - s)/myA(k,k);
    end;
    u = unew;
end
```

Do this in parallel: divide and conquer

- solve a problem of size n = myn * nproc

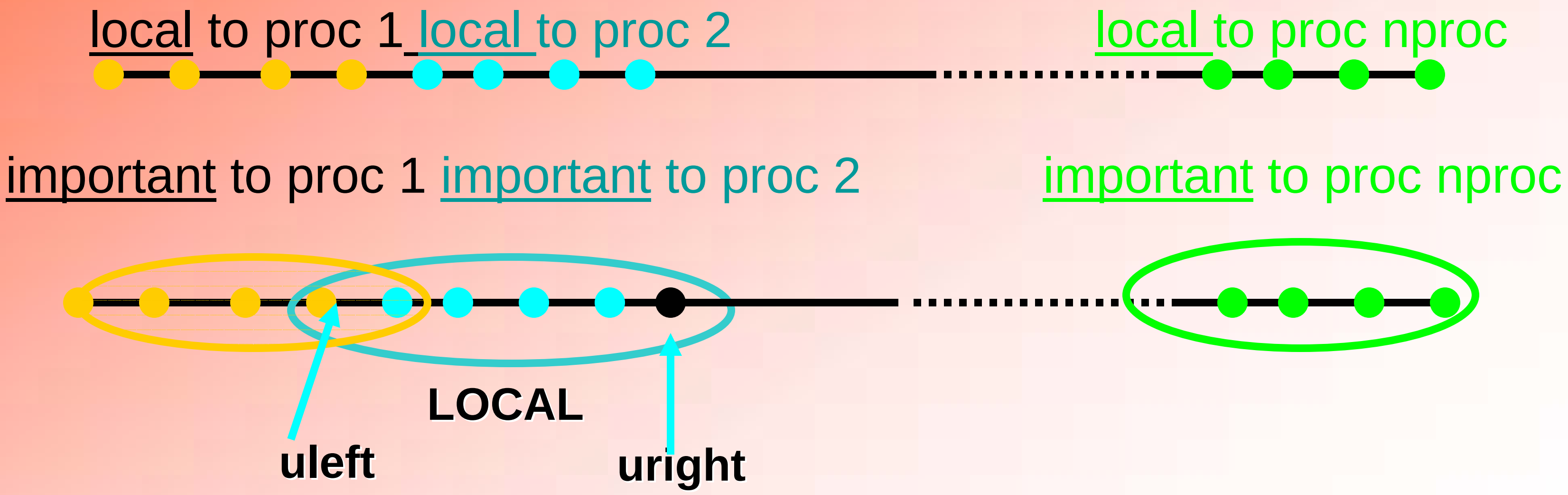


- paradigm:
  - same as block Jacobi, using Jacobi on individual processors (additive Schwartz, overlapping DD)
  - equivalently: perform iteration as if everything was done on “one processor”
- must be able to exchange data between processors

Decompose data

[1…myn]   [myn+1 … 2*myn] ….. [myn*(nproc-1)+1..myn*nproc=n

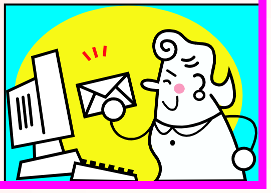
- analyze what is happening to data between two processors



function iter = myjacobi_parallel (myn,tol,maxiter)

- %% myn = size of local problem
- %% tol = tolerance for size of residual
- %% maxiter = max number of iterations acceptable

- EACH processor must know
  - its own number p
  - total number of processors nproc
- EACH processor must be able to access
  - its local data
  - all data important to her
- Processors must be able to communicate
  - with immediate neighbors
  - with everybody

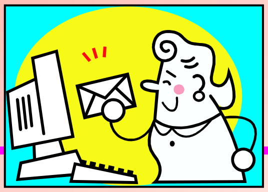


initialize

```
n = myn*nproc;   %% size of global problem
h = 1/(n+1);    %% n,h are global

%%% only local variables allocated
myA = zeros (myn,myn);
for i = 1:myn myA(i,i)=2;end;
for i = 2:myn myA(i,i-1)=-1;end;
for i = 1:myn-1 myA(i,i+1)=-1;end;
myb = h * h * ones(myn,1);

%%% off-diagnal blocks represented
Aright = -1;
Aleft = -1;

%%% initial guess: could be given externally
u0 = zeros(myn,1); u = u0;
```

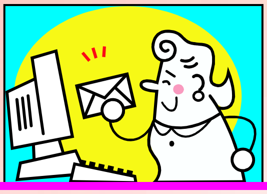
Jacobi iteration … parallel (1)

```
for iter = 1:maxiter
    %%% make sure every processor has current data in uleft, uright
    %%% PARALLEL send your own data: u(1), u(myn)
    %%% PARALLEL receive uleft, uright

    %%%   compute residual
    myr = myb - myA*u;
    %%% use important data on the left and right
    if p > 1 myr(1) = myr(1) - Aleft*uleft(p);end;
    if p < nproc myr(myn) = myr(myn) - Aright*uright(p);end;
```



Jacobi iteration … parallel (2)

```
%%%% compute norm of residual
rerr = norm(myr);
%%%% collect norms from all processors .. execute
%%%% PARALLEL reduce_ operation
glob_err = rerr;
fprintf('%d %g\n',iter,glob_err); %%%% only if p=1

%%%% is it time to stop ?
if glob_err < tol break;end;
```



perform the update …

```
for k = 1:myn
        %% collect right hand side terms in every row
        s = 0;
        for j = 1:k-1 s = s + myA(k,j)*u(j);end;
        for j = k+1:myn s = s + myA(k,j)*u(j);end;
        %%% incorporate values from left and right
       if k ==1 if p > 1       s = s + Aleft*uleft(p);end;end;
       if k == myn if p < nproc   s = s + Aright*uright(p); end; end;

    %% compute new guess in row k
        unew (k,1) = (myb(k) - s)/myA(k,k);
end;
u = unew;
end
```

Details on parallel operations

- use MPI (Message Passing Interface) developed/ described at
    - http://www-unix.mcs.anl.gov/mpi/

- we will use a few elementary operations as subroutine calls from Fortran
    - initialization/introduction/finalize operations
        - `call MPI_INIT (ierr)`
        - `call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)`
        - `call MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)`
        - `call MPI_FINALIZE(ierr)`

Details on parallel operations

- cd operations
  - PARALLEL ***reduce*** operation

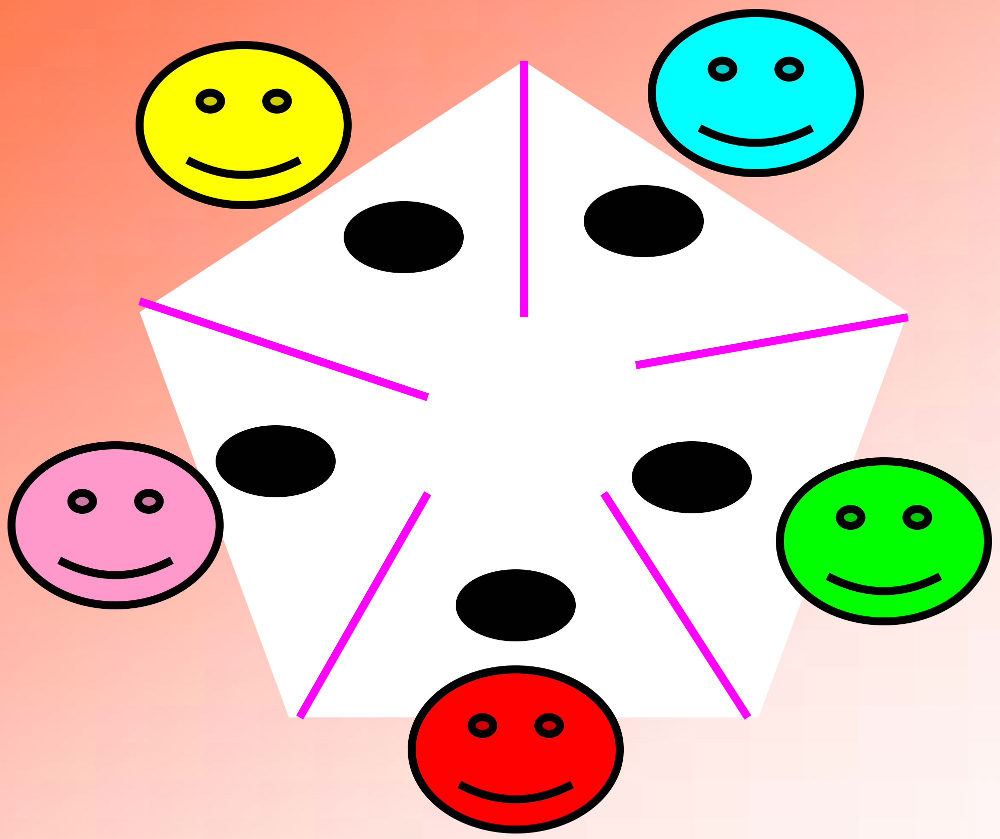
    -
      ```
      call MPI_ALLREDUCE(rerr,glob_err,1,MPI_DOUBLE_PRECISION,
                MPI_SUM,MPI_COMM_WORLD, ierr)
      ```
  - PARALLEL ***send*** operation

    -
      ```
      call MPI_Send ( what, count, MPI_DOUBLE_PRECISION,
            where,
           tag,
        MPI_COMM_WORLD,ierr )
      ```
  - PARALLEL ***receive*** operation

    -
      ```
      call MPI_RECV (what, howmuch, MPI_DOUBLE_PRECISION,
      wherefrom,
        tag,
         MPI_COMM_WORLD,status,ierr )
      ```

Caution suggested: parallel disasters

- Five philosophers (mathematicians ?)



- nobody ever gets to eat (DEADLOCK, LIVELOCK)

More technical disaster example

- Trying to compute c=(a+b)/2.
  - value of a belongs to proc 0, value of b belongs to proc 1
- try the code

```
if (rank.eq.0) then
    call MPI_RECV (b, 1, MPI_DOUBLE_PRECISION, 1,0,
   MPI_COMM_WORLD,status,ierr )
    call MPI_Send ( a, 1, MPI_DOUBLE_PRECISION,1,0,
           MPI_COMM_WORLD,ierr )
else
    call MPI_RECV (a, 1, MPI_DOUBLE_PRECISION, 1,0,
   MPI_COMM_WORLD,status,ierr )
    call MPI_Send ( b, 1, MPI_DOUBLE_PRECISION,1,0,
           MPI_COMM_WORLD,ierr )
endif
```

- this is a classical DEADLOCK

Another deadlock example

- Trying to compute c=(a+b)/2.
  - value of a belongs to proc 0, value of b belongs to proc 1
- try the code

```
if (rank.eq.0) then
    call MPI_Send ( a, 1, MPI_DOUBLE_PRECISION,1,1,
           MPI_COMM_WORLD,ierr )
    call MPI_RECV (b, 1, MPI_DOUBLE_PRECISION, 1,1,
   MPI_COMM_WORLD,status,ierr )

else
   call MPI_Send ( b, 1, MPI_DOUBLE_PRECISION,0,0,
           MPI_COMM_WORLD,ierr )
    call MPI_RECV (a, 1, MPI_DOUBLE_PRECISION, 0,0,
   MPI_COMM_WORLD,status,ierr )
endif
```

- this is a classical DEADLOCK

Example 1: compute Pi (part 1)

```
 program mypi
      implicit none
c%%%%% MPI declarations
      include 'mpif.h'
      integer nproc, rank, p, ierr, rc
c%%%%% problem declarations
      integer myn,n, istart, iend
      double precision h,s,x,glob_pi
      integer i
c%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
      call MPI_INIT(ierr)
      if (ierr .ne. MPI_SUCCESS) then
         print *,'Error starting MPI program. Terminating.'
         call MPI_ABORT(MPI_COMM_WORLD, rc, ierr)
      end if
c%%%% what is my number (rank+1) and total number of processors
      call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
c
      p = rank + 1
      if (rank.eq.0) then
          write(6,*) 'Number of processors=',nproc
      endif
```

Example 1: compute Pi (part 2)

```
c%%%% set size of subproblem as a constant
      myn = 100
      n = nproc * myn
      h = 1.D0/n
c%%% compute the integral using midpoint rule
      s = 0D0
      istart = rank*myn + 1
      iend = p*myn
      do i = istart, iend
         x = i*h - h/2D0
         s = s + 1D0 / (1D0+x*x)
      enddo
      s = s * 4.D0 * h
c %%% PARALLEL: must add all values
      call MPI_ALLREDUCE(s,glob_pi,1,MPI_DOUBLE_PRECISION,MPI_SUM,
     $      MPI_COMM_WORLD,ierr)
c %%% finished
      if (rank.eq.0) then
         write(6,*) 'Finished with ',n,' subitervals. Result=',glob_pi
         write(6,*) 'Error = ',abs( glob_pi-atan(1.0)*4D0 )
c%%%%%%%%%%%%%%%%%%%%%%%%%%%
         call MPI_FINALIZE(ierr)
      end
```

Example 2: update vector (as in before Jacobi iteration) (1)

```
c%%%% same initialization as in Example 1 ….
      p = rank + 1
      n = nproc * myn
      uleft = 0D0;
      uright = 0D0;
c%% %%% every processor records its number in vector u
      do i = 1, myn
         u(i) = p;
      enddo
c
   write(6,*) 'Proc ',p,' data before
   ',uleft,(u(i),i=1,myn),uright
c%%%%%%%%%%%%%%%%%%%%%%%%%%%% example of exchange data
```

Example 2: update vector (as in before Jacobi iteration) (2)

```
c exchaNnge data
           tag = 0
           if (p.lt.nproc) then
              call MPI_Send(u(myn), 1, MPI[redacted]
     &          tag,MPI_COMM_WORLD,ierr );
           endif
           if (p.gt.[redacted]
              call MP[redacted]1), 1, MPI_DOU[redacted]
     &          tag,MP[redacted]WORLD,ierr );
           endif
           if (p.lt.n[redacted]
              call M[redacted]
     &          t[redacted]
     &[redacted]
           endif
      write(6,*) 'Proc ',p,' data[redacted],myn),uright
```

The correct answer will be posted after the lab

How to run the code

- go to your cluster account
- get (scp) all files/type new program
- compile
  - mpif77
- submit a job to queue (uses mpirun ..)
  - must use a job file
  - input from file only
  - output can go to screen, will be saved in job log file
- wait for job to finish
  - qstat the queue
- PLEASE
  - wait until LAB 7 before using the cluster
  - do not submit multiple jobs before first one is done
  - ask for help when needed

Other issues in parallel performance

- load balancing
  - dynamic for FE meshes
  - dynamic for CFD
  - dynamic for transient problems
  - dynamic for nonlinear solvers and sophisticated preconditioners
- speedup and Amdahl’s law: T(N,p) >= T(N,1)/p
  - parallel efficiency
- scaled speedup (especially for out-of-core problems)
  - change N when p changes
- parallel implementations of solvers typically have worse properties than serial (single-processor) implementations
  - G-S inherently serial
  - multigrid has issues