MTH 654/659
Parallel computing methods

Instructor:
Malgorzata Peszynska
Mathematics
Oregon State University

Parallel computing: Algorithm + Architecture + Software

- Algorithm = Computation + Communication
- Architecture = *PU + Memory + “cables”
- In our class we will use
  - Parallel cluster (COE+COS+…):
    - Distributed memory machine
    - Communication through Message Passing software (MPI)
  - Desktops with GPUs (in MLC)
    - Host (CPU) + device (GPU) model
    - Shared memory/threads model
    - Communication through memory read/write
- Software (for communication):
    - Must be able to exchange data between processors/threads
      - Send + receive
    - Must be able to perform “reduce” operations
      - Add results from all processors

Parallel computing: Algorithm + Architecture + Software

- Simple parallel example: compute PI (again !!!)
  - Algorithm: exchange partial sums
  - Architecture: cluster
  - Software: use MPI (calls from Fortran)
- Overlapping DD for solving PDE/ linear system
  - Algorithm: use block Jacobi, solve in *any* way on every processor
  - Architecture: cluster
  - Software: use MPI (calls from Fortran)
- Newton-CG
  - Algorithm: use special BLAS
  - Architecture: GPU (shated memory)
  - Software: use CUDABLAS

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: overlapping DD

- Either “real” Domain Decomposition
  - E.g., Schwarz methods
  - (we’ll talk about these later)
- Or … just simply a way to solve linear systems/numerical PDE on multiple processors
  - Use Jacobi method as an example
  - Show how to implement it in parallel
    - Decompose into subdomains; each processor owns a subdomain
    - What needs to be communicated between subdomains ?
      - » Send-receive ghost values (uleft, uright)
      - » Reduce operations: compute norm
  - The parallel version of Jacobi method can be generalized to an iterative method (block Jacobi) in which each processor/subdomain solves its own part of the linear system
    - Each iteration of such solver uses SAXPY, SDOT, SNRM2,SGEMV
    - For consistence, interface values need to be communicated

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 code)

```
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

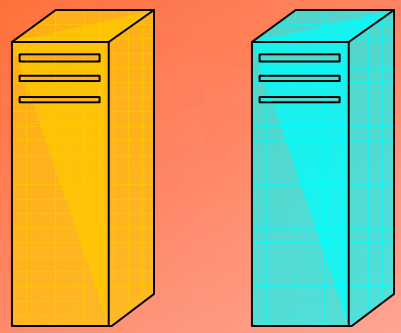
```
u = u0;
for iter = 1:maxiter
        myr = myb - myA*u;
        rerr = norm(myr);
        glob_err = rerr;
        fprintf('%d %g\n',iter,glob_err);

        %%% 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



myn **+ myn** **+ …..** **+ myn** **= n**

- paradigm:block Jacobi
- must be able to exchange data between processors

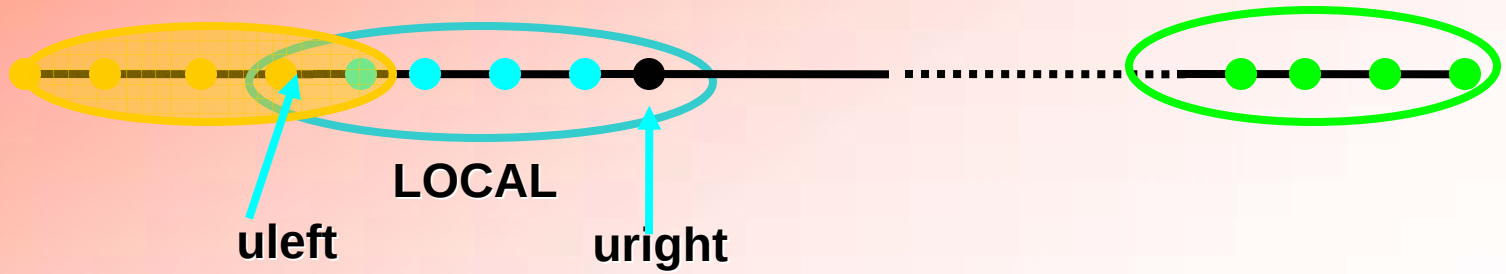
Decompose data

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

- analyze what is happening to data between two processors

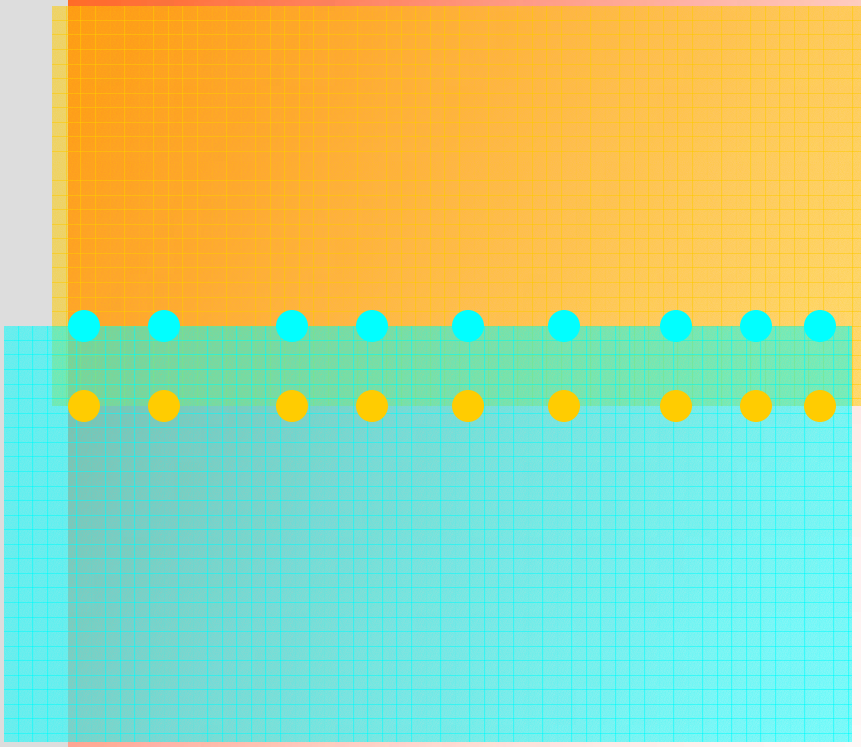
local to proc 1 local to proc 2 local to proc nproc

important to proc 1 important to proc 2 important to proc nproc

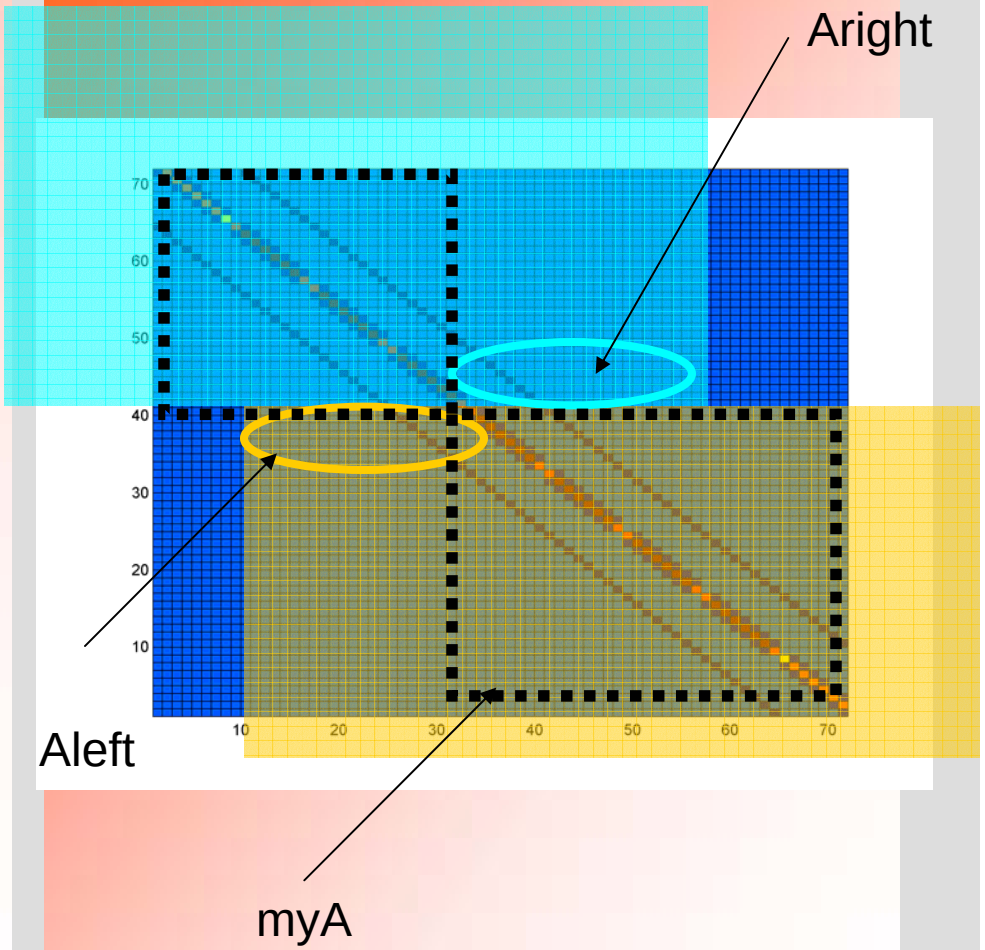


Domain decomposition: overlapping

- blue and yellow values lag between iterations



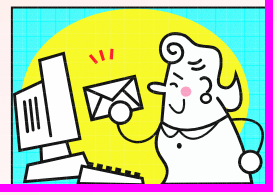
- similar to block Jacobi solver



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

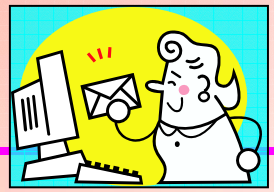


initialize

```
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;
```

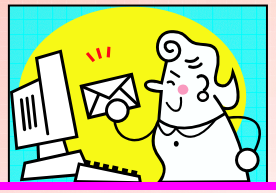
Jacobi iteration … parallel (1)

```
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
```

Example 2: how to start implementation

```
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: how to update vector

```
c exchaNnge data
         tag = 0
         if (p.lt.nproc) then
            call MPI_Send(u(myn), 1, MPI_DOUBLE_PRECISION, rank + 1,
     &        tag,MPI_COMM_WORLD,ierr );
         endif
         if (p.gt.1) then
            call MPI_Send(u(1), 1, MPI_DOUBLE_PRECISION, rank - 1,
     &        tag,MPI_COMM_WORLD,ierr );
         endif
         if (p.lt.nproc) then
            call MPI_RECV (uright, 1, MPI_DOUBLE_PRECISION, rank + 1,
     &        tag,MPI_COMM_WORLD,status,ierr );
         endif
         if (p.gt.1) then
            call MPI_RECV (uleft, 1, MPI_DOUBLE_PRECISION, rank - 1,
     &        tag,MPI_COMM_WORLD,status,ierr );
         endif
       write(6,*) 'Proc ',p,' data after ',uleft,(u(i),i=1,myn),uright
```

Now back to Example 2: how to implement using MPI ?

- Initialize MPI
- How to perform global reduce operations ?
- How to exchange data between processors ?
  - Send
  - Receive
- Also
  - Wait (for synchronization)
  - Broadcast
- MUST understand the logical flow so as not to lead to
  - Deadlock of all processors
    - Waiting for somehting that will never happen
  - Livelock (starvation) of a processor
    - Never getting the resources it needs

Details on parallel operations

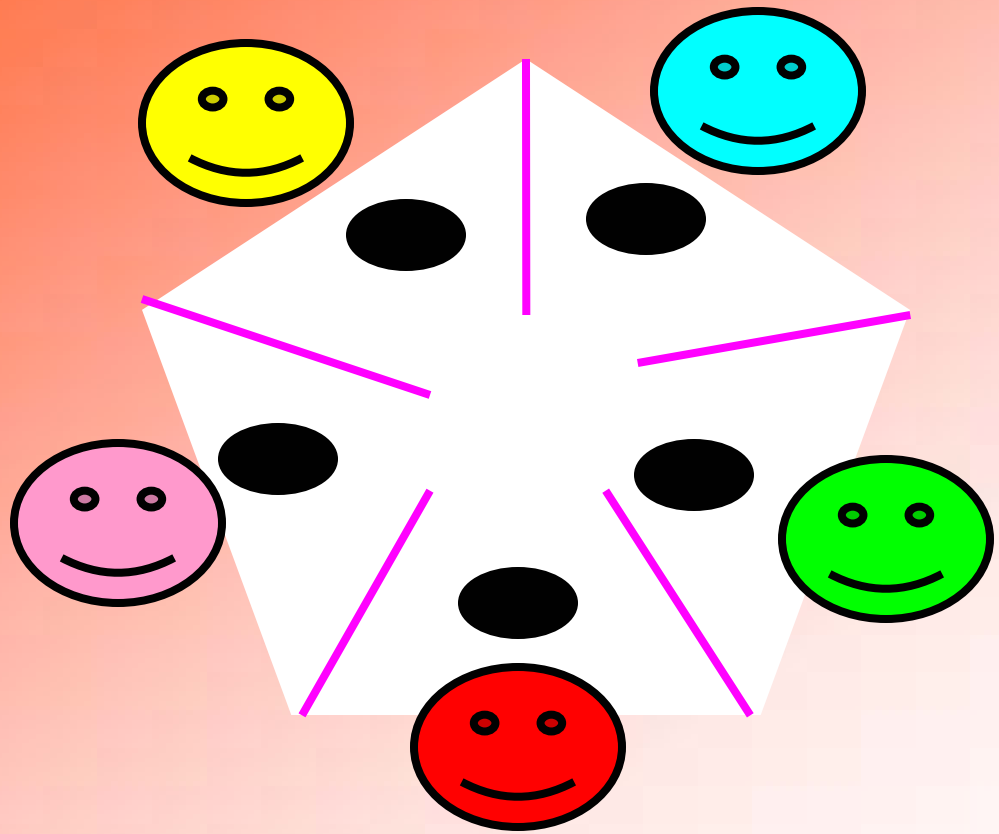
- use MPI (Message Passing Interface) developed/ described at
  - http://www-unix.mcs.anl.gov/mpi/
- we will use a few elementary operations as subroutine calls from Fortran
  - initialization/introduction/finalize operations
    - `call MPI_INIT (ierr)`
    - `call MPI_COMM_RANK(MPI_COMM_WORLD, rank, 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->four philosophers (mathematicians ?)



- Nobody ever gets to eat (DEADLOCK)
- Only proc. 1,3 eat (LIVELOCK, STARVATION)

Disaster example 1: what is wrong ?

- 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: mismatched calls

Disaster example 2: what is wrong ?

- 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: mismatched calls

Disaster example 3: what is wrong ?

- 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, 0,0,
   MPI_COMM_WORLD,status,ierr )
    call MPI_Send ( b, 1, MPI_DOUBLE_PRECISION,0,0,
           MPI_COMM_WORLD,ierr )
endif
```

- this is a classical DEADLOCK

Example 4: could things go wrong ?

- 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,1,0,
            MPI_COMM_WORLD,ierr )
    call MPI_RECV (b, 1, MPI_DOUBLE_PRECISION, 1,0,
   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 could result in a DEADLOCK, remedy: unblocking send/recv ?

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
  - E.g.: Gauss-Seidel inherently serial
  - multigrid has issues

How to run the MPI code (Ex.1 and 2)

- 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
  - do not submit multiple jobs before first one is done
  - ask for help when needed

Example 3: use GPUs

- Here we will use various online resources
  - Consult class webpage for links describing the GPU setup
- In our examples we will take advantage of highly optimized libraries for GPUs
  - cudaBLAS
- Our example is CG implementation
  - Uses BLAS or BLAS-like functions
    - SAXPY, SDOT, SNRM2, SGEMV
- We will use C in this example on Linux computers in MLC
  - Information through class website