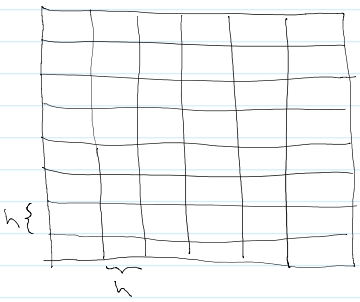
Mesh based solutions: Poisson Eqn

January 28, 2014 9:21 PM

Poisson Equation

Finite Difference Schemes for Poisson Equation

$\partial_x^2 u + \partial_y^2 u = f$ B.C.s: u = const on boundary



$x_l = a + lh \quad l = 0, \dots, L$

$y_m = c + mh \quad m = 0, \dots, M$

finite difference approximation to derivatives

$$\frac{\delta_x^2 u}{h^2} = \frac{u(x_{l+1}, y_m) - 2u(x_l, y_m) + u(x_{l-1}, y_m)}{h^2} + O(h^2)$$

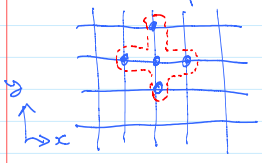
will work on interior mesh points, label $u(x_l, y_m) \equiv v_{l,m}$

$\rightarrow \delta_x^2 u + \delta_y^2 u = h^2 f$

or, assuming $h_x = h_y = h$

$$\text{(*)} \quad \frac{1}{h^2}\left(v_{l+1,m} + v_{l-1,m} + v_{l,m+1} + v_{l,m-1} - 4v_{l,m}\right) = f_{l,m}$$

this is often referred to as the 5-point stencil:

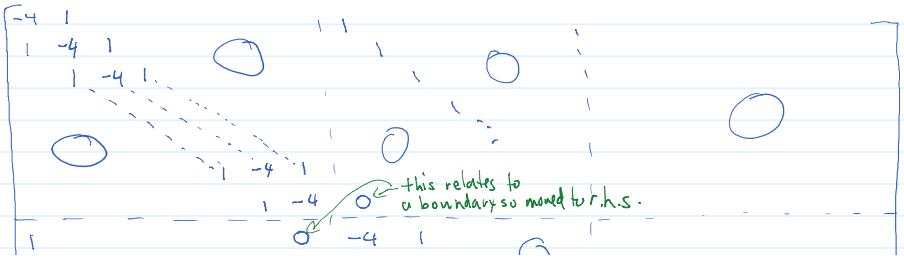


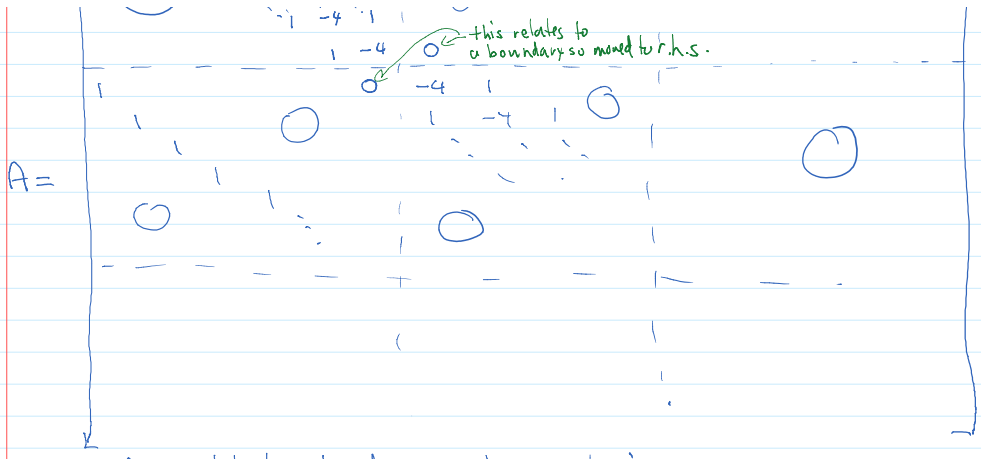
Note: (*) is a banded matrix equation with bandwidth $2L-1$ (or $2M-1$) } matrix eqn $Av = F$

$$v = \begin{bmatrix} v_{1,1} \\ v_{1,2} \\ \vdots \\ v_{1,M-1} \\ v_{2,1} \\ v_{2,2} \\ v_{2,3} \\ \vdots \\ v_{2,M-1} \\ v_{3,1} \\ \vdots \\ v_{L-1,M-1} \end{bmatrix}$$

boundary values (known assuming Dirichlet B.C.'s)

$$F = \begin{bmatrix} h^2 f_{1,1} - v_{0,1} - v_{1,0} \\ h^2 f_{1,2} - v_{0,2} \\ \vdots \\ h^2 f_{1,M-1} - v_{0,M-1} - v_{1,M} \\ h^2 f_{2,1} - v_{2,0} \\ h^2 f_{2,2} \\ h^2 f_{2,3} \\ \vdots \end{bmatrix}$$





a block tridiagonal matrix

note: the bands closest to the diagonal have boundary terms moved to r.h.s. so have zero entries for some elements.

The matrix A is negative definite. In order to make use of Lapack routines, which have specialized routines for positive definite matrices, we will instead solve

$$(-A)v = (-F)$$

as $-A$ will be positive definite.

Positive definite matrices are symmetric and so we only need to store the upper triangular part.
We can factor $-A$ using Cholesky factorization

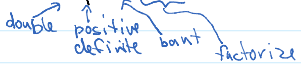
i.e. $-A = U^T U = L L^T$ where U, L are upper/lower triangular

and then use the factorization to solve. (this is a special form of LU factorization for positive definite matrices)

Note: Most of the work is in the factorization so once this is done you could use the factors to solve for multiple R.H.S. without much extra effort.

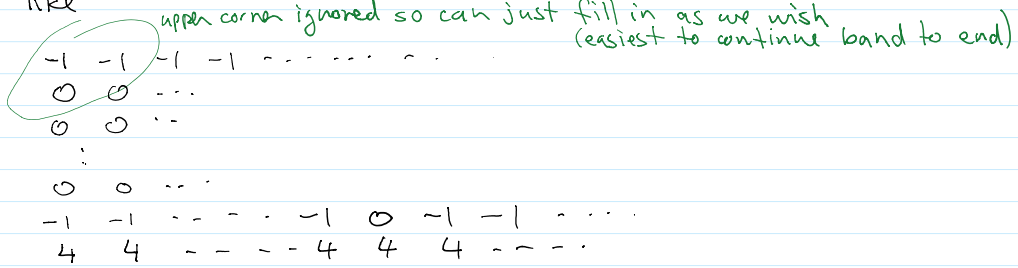
Routines from Lapack to do this are

dpbtrf - factorize



dpbtrs - solve using factorization

The matrix $-A$ is stored in band form and only requires the upper triangular region so looks something like



eg

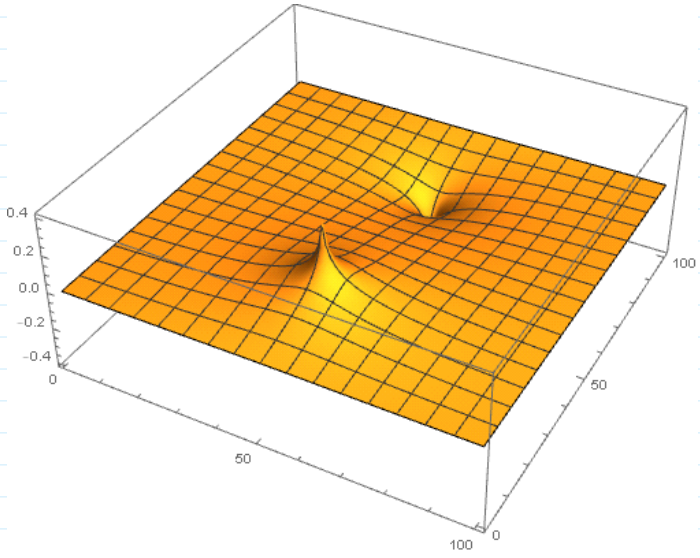
```
#include <iostream>
```

```
#include "boost/multi_array.hpp"

// Program to compute solution to -laplacian u = -F
// compile with
// g++ PoissonLAPACK.cpp -lblas -llapack

// LAPACK library files
extern "C" {
  // general band factorize routine
  extern int dpbtrf_(char *, int*, int*,double *, int*,int*);
  // general band solve using factorization routine
  extern int dpbtrs_(char*, int*,int*,int*,double*,int*,double*,int*,int*);
}


void AbInit(int N, boost::multi_array<double,2> &Ab)
{
 // Coefficient Matrix: initialize  (band format)
  for (int i=0; i < N*N; i++)
    for (int j=0; j< N+1; j++) {
      Ab[j][i]=0.0; <- most entries are zero so set as default
      if (j == 0)
        Ab[j][i] = -1.0;
      if (j == N-1 && i%N)   <- only 0 when i is divisible by N,
        Ab[j][i] = -1.0;        exactly where we need the 0's
      if (j == N)
        Ab[j][i] =  4.0; <- main diagonal
    }
  //std::cout << "Ab initialized \n";
  // Printout Ab for testing, small N only
  //for (int i=0; i < N+1; i++) {
  //  std::cout << Ab[i][0];
  //  for (int j=1; j < N*N; j++) {
  //    std::cout << " " << Ab[i][j];
  //  }
  //  std::cout << "\n";
  //}
}


void RHSInitialize(int N, std::vector<double> &F, double bcx, double bcy)
{
  // RHS: fill in boundary condition values
  for (int i = 0; i < N; i++) {
    F[i] += bcx;         //bottom boundary
    F[N*N-i-1] += bcx;   //top boundary
    F[i*N] += bcy;       //left boundary
    F[i*N+N-1] += bcy;   //right boundary
  }
  //std::cout << "RHS initialized\n";

  // RHS: fill in actual right-hand side, some "charges", actually h^2*charge
  F[N/4*N+N/2] += 0.5;
  F[3*N/4*N+N/2] += -0.5;
}

int main (void)
{ // Coefficient Matrix: declare
  const int N=100;
  int M = N+1;
  int ABcols = N*N;                                   only Ab in fortran form
  boost::multi_array<double,2> Ab(boost::extents[M][ABcols],boost::fortran_storage_order());

  AbInit(N,Ab);  // Initialize coefficient matrix

  // Coefficient Matrix: factorize
  char uplo = 'U';
  int KD = N;
  int info;
  dpbtrf_(&uplo, &ABcols, &KD, &Ab[0][0], &M, &info);
  if (info) {
    std::cout << "Ab failed to factorize, info = " << info << "\n";
    exit(1);
  }
```

```
  // RHS: declare
  const double bcx = 0.0, bcy = 0.0; // boundary conditions along x and y assume same on both sides
  std::vector<double> F(N*N,0.0);

  RHSInitialize(N, F, bcx, bcy); // set up boundary conditions and right hand side

  // Solve system
  int Bcols=1;
  dpbtrs_(&uplo,&ABcols,&KD, &Bcols, &Ab[0][0],&M,&F[0],&ABcols,&info);
  if (info) {
    std::cout << "System solve failed, info = " << info << "\n";
    exit(1);
  }

  // Output solution
  for (int i=0; i < N; i++) {
    std::cout << F[i*N];
    for (int j=1; j < N; j++) {
      std::cout << " " << F[i*N+j];
    }
    std::cout << "\n";
  }

  return 0;
}
```

(ordered in C-style, as in notes above)

Solution looks like:



Note: If we wanted to move the charges around and recompute the potential, we would re-use the factorization. Only change F and call dpbtrs each time you change F.

As you will see in HW, if this direct solution is not practical for large systems. We need to consider something else.

Jacobi Method

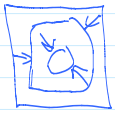
Solve ⊛ for $v_{l,m}$ and iterate:

$$v_{l,m}^{k+1} = \frac{1}{4}\left(v_{l+1,m}^{k} + v_{l-1,m}^{k} + v_{l,m+1}^{k} + v_{l,m-1}^{k}\right)$$

(this is equivalent to solving $\partial_t u = \nabla^2 u$ using forward-time central-space and ...)

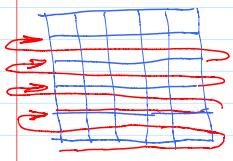
$\Delta t = \frac{1}{4}h^2$ and then iterating to steady-state)

We stop when $|v_{i,m}^{k+1} - v_{i,m}^{k}| < tol$ (for all sites)



solution converges from boundaries (which fixed by B.C.s)

Gauss-Seidel



If we go through mesh in lexicographic order then we will have updated $v_{i-1,m}$ and $v_{i,m-1}$ when we come to update $v_{i,m}$ in Jacobi method. If we use an immediate replacement convergence will faster (and we don't need to keep "old" and "new" values):

$$v_{i,m}^{k+1} = \frac{1}{4}\left(v_{i+1,m}^{k} + v_{i-1,m}^{k+1} + v_{i,m+1}^{k} + v_{i,m-1}^{k+1} \right)$$

Successive Over-Relaxation (SOR)

Take "bigger" step towards solution:

$$v_{i,m}^{k+1} = v_{i,m}^{k} + \omega\left[\frac{1}{4}\left(v_{i+1,m}^{k} + v_{i-1,m}^{k+1} + v_{i,m+1}^{k} + v_{i,m-1}^{k+1}\right) - v_{i,m}^{k}\right] \quad (*)$$

NB: $\omega = 1 \Rightarrow$ SOR = Gauss-Seidel
generally we will take $\omega > 1$ to converge faster

How do we pick ω for fastest convergence?
Typically $1 \le \omega \le 2$ works, so pick something in this range
Find by trial and error for small system (large h)
then use:

$$\omega^* \approx \frac{2}{1 + Ch}$$

usually good for more general cases.

To determine C, try optimizing convergence for small sytem (large h), then use that C for big system.

NB: It is better to overestimate ω than to underestimate
NB: Taking ω too large can cause problems in a parallel implementation where "boundaries" are forced to use Gauss-Seidel.

We could even divide the system into small blocks and solve small blocks exactly. As the boundary values will be from the previous iteration, we will still need to iterate to convergence.
In general, we will end up using some Jacobi/Gauss-Seidel hybrid scheme.

While the Jacobi scheme is not a great algorithm, it is very simple and illustrates how we can parallize a mesh-based problem.

Parallel Jacobi Method

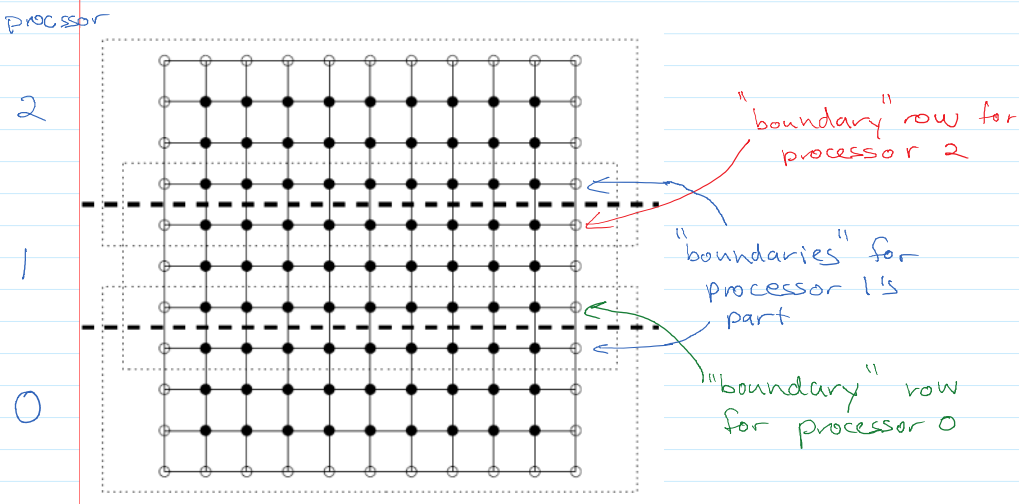
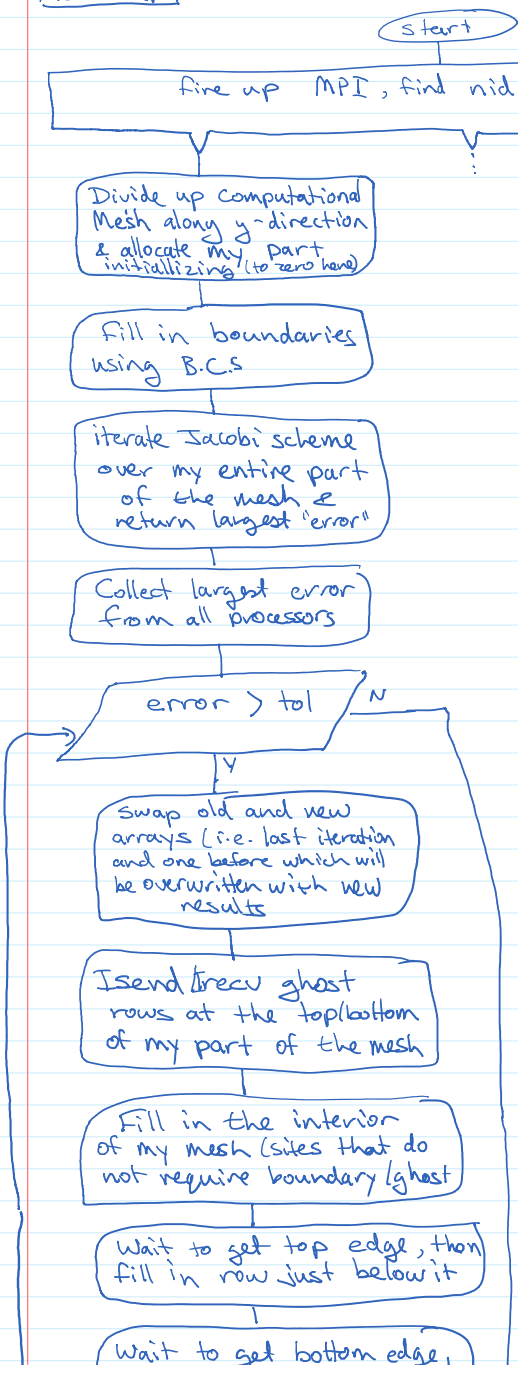
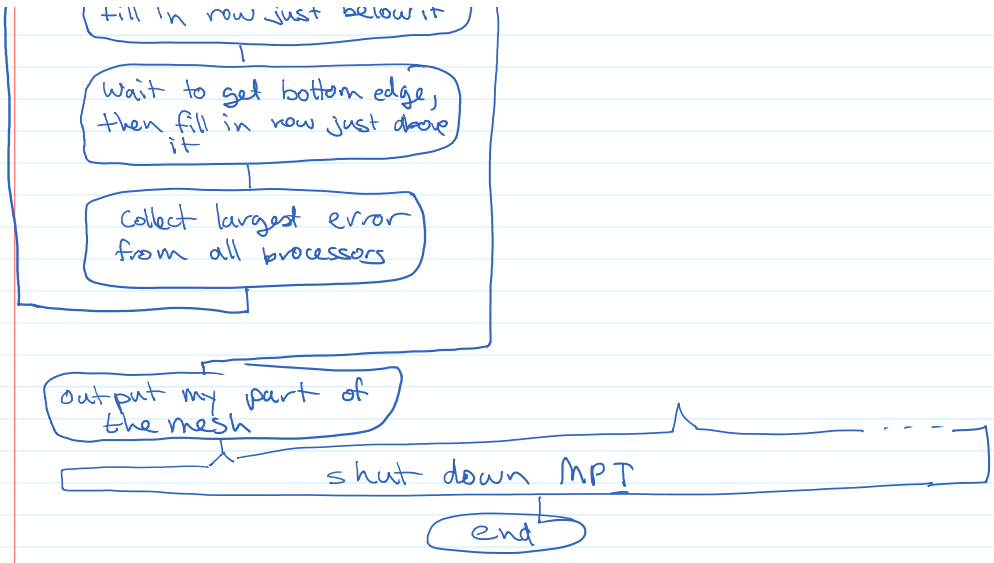


Figure 1: Partition of a finite difference mesh across 3 processes.

Flowchart






```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mpi.h"
// Solve Poisson's equations using the Jacobi method, with MPI parallization
// dividing the grid into a 1D array of processors
// Compile with mpicc PoissonJacobi1DMPI.c -lm

#define N  100
#define Tol  0.000001

// MPI Stuff
struct mpi_vars {
  int NProcs;
  int MyID;
};

struct mpi_vars mpi_start(int argc, char** argv)
{
  struct mpi_vars this_mpi;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &this_mpi.NProcs);
  MPI_Comm_rank(MPI_COMM_WORLD, &this_mpi.MyID);

  return this_mpi;
}

// breakup array size N in 1D into parts for each processor
int compute_my_size(struct mpi_vars the_mpi)
{
  int remainder = (N - 1) % the_mpi.NProcs;
  int size = (N - 1 - remainder)/the_mpi.NProcs;
  if(the_mpi.MyID < remainder)  // extra_rows added for MyID < remainder
    size = size + 2;
  else
    size = size + 1;
  return size;
}

double **matrix(int m, int n)
{
    /* Note that you must allocate the array as one block in order to use */
    /* MPI derived data types on them.  Note also that calloc initializes */
    /* all entries to zero. */

    double ** ptr = (double **)calloc(m, sizeof(double *));
    ptr[0]=(double *)calloc(m*n, sizeof(double));
    for(int i = 1; i < m ;i++)
      ptr[i]=ptr[i-1]+n;
    return (ptr);
}

void initialize(double **old, double **new, int size, struct mpi_vars the_mpi)
{ /* Assume interior is already initialized, just need to set boundaries */
  /* B.Cs set to a constant, 1 here */

  for(int i = 0; i < size + 1; i++)
    new[i][0] = new[i][N] = old[i][0] = old[i][N] = 1;
  if(the_mpi.MyID == 0)
```

```
    for(int j = 1; j < N; j++)
      new[0][j] = old[0][j] = 1;
  if(the_mpi.MyID == the_mpi.NProcs - 1)
    for(int j = 1; j < N; j++)
      new[size][j] = old[size][j] = 1;
}

double iteration(double **old, double **new, int start, int finish)
{   /* Jacobi iteration */
    double maxerr = 0;
    for(int i = start; i < finish; i++)
       for(int j = 1; j < N; j++){
         new[i][j] = 0.25*(old[i+1][j] + old[i-1][j] +
                           old[i][j+1] + old[i][j-1]);

         maxerr = fmax(maxerr, fabs(new[i][j] - old[i][j]));
       }
    return (maxerr);
}

void output(double **new, int size, struct mpi_vars the_mpi)
{ /* Output result */
  char str[20];
  FILE *fp;

  sprintf(str,"Solution%d.Txt",the_mpi.MyID);
  fp = fopen(str,"w");
  if(the_mpi.MyID == 0) {
    for(int j = 0; j < N + 1; j++)
      fprintf(fp,"%6.4f ",new[0][j]);
    fprintf(fp,"\n");
  }
  for(int i = 1; i < size; i++) {
    for(int j = 0; j < N + 1; j++)
      fprintf(fp,"%6.4f ",new[i][j]);
    fprintf(fp,"\n");
  }
  if(the_mpi.MyID == the_mpi.NProcs - 1){
    for(int j = 0; j < N + 1; j++)
      fprintf(fp,"%6.4f ",new[size][j]);
    fprintf(fp,"\n");
  }
  fclose(fp);
}

int main(int argc, char** argv)
{
  struct mpi_vars the_mpi = mpi_start(argc, argv);

  MPI_Status status;
  MPI_Request req_send10, req_send20, req_recv10, req_recv20;

  /* breakup compute grid amoung processors and initialize*/
  int size = compute_my_size(the_mpi);
  double **new = matrix(size+1, N+1);
  double **old = matrix(size+1, N+1);
  initialize(old, new, size, the_mpi);
  double **tmp; //used for swapping old and new arrays

  /* do one iteration and work out global error */
  double maxerrG;
  double maxerr = iteration(old, new, 1, size);
  MPI_Allreduce(&maxerr, &maxerrG, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);

  /* Main loop */
  while(maxerrG > Tol){
    tmp = new;
    new = old;
    old = tmp;
    /* send/receive at top of compute block */
    req_send10 = req_recv20 = MPI_REQUEST_NULL;
    if(the_mpi.MyID < the_mpi.NProcs - 1){
      MPI_Isend(&old[size-1][1], N-1, MPI_DOUBLE, the_mpi.MyID+1, 10, MPI_COMM_WORLD, &req_send10);
      MPI_Irecv(&old[size][1], N-1, MPI_DOUBLE, the_mpi.MyID+1, 20, MPI_COMM_WORLD,&req_recv20);
    }
    /* send/receive at bottom of compute block */
    req_send20 = req_recv10 = MPI_REQUEST_NULL;
    if(the_mpi.MyID > 0){
      MPI_Isend(&old[1][1], N-1, MPI_DOUBLE, the_mpi.MyID-1, 20, MPI_COMM_WORLD, &req_send20);
      MPI_Irecv(&old[0][1], N-1, MPI_DOUBLE, the_mpi.MyID-1, 10, MPI_COMM_WORLD, &req_recv10);
    }
    /* update interior of compute block excluding beside boundaries */
    maxerr = iteration(old, new, 2, size-1);
```

range of rows to do Jacobi iteration on

note each processor will output their piece of the grid to a seperate file

immediate send/recv, don't wait for response

do later after we have recv boundary

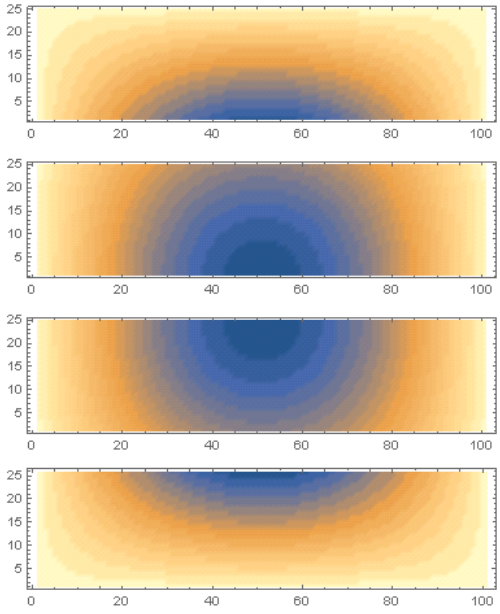
```
    /* update compute block beside boundaries as available */
    /* top edge */
    if(the_mpi.MyID < the_mpi.NProcs - 1)
      MPI_Wait(&req_recv20, &status);
    maxerr = fmax(maxerr, iteration(old, new, size-1, size));

    /* bottom edge */
    if(the_mpi.MyID > 0)
      MPI_Wait(&req_recv10, &status);
    maxerr = fmax(maxerr, iteration(old, new, 1, 2));

    /* find global error */
    MPI_Allreduce(&maxerr, &maxerrG, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
  }

  /* Output result */
  output(new, size, the_mpi);

  free(old[0]);
  free(old);
  free(new[0]);
  free(new);
  MPI_Finalize();
  return 0;
}
```

now do top/bottom

Results:

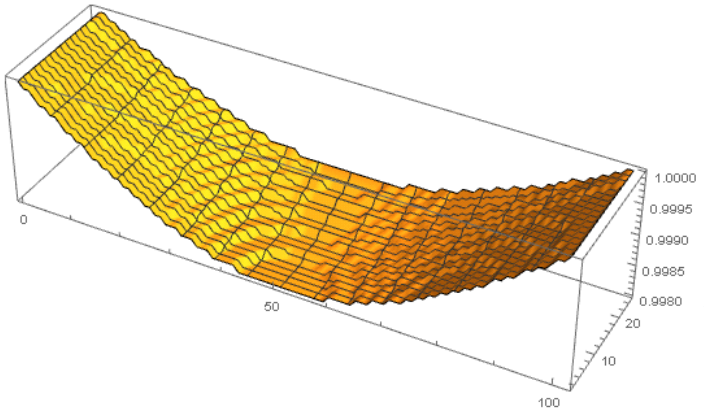
```
In[38]:= GraphicsColumn[
          {ListDensityPlot[dat6, AspectRatio → Dimensions[dat6][[1]] / Dimensions[dat6][[2]]],
           ListDensityPlot[dat5, AspectRatio → Dimensions[dat5][[1]] / Dimensions[dat5][[2]]],
           ListDensityPlot[dat4, AspectRatio → Dimensions[dat4][[1]] / Dimensions[dat4][[2]]],
           ListDensityPlot[dat3, AspectRatio → Dimensions[dat3][[1]] / Dimensions[dat3][[2]]]}]
```

Out[38]=



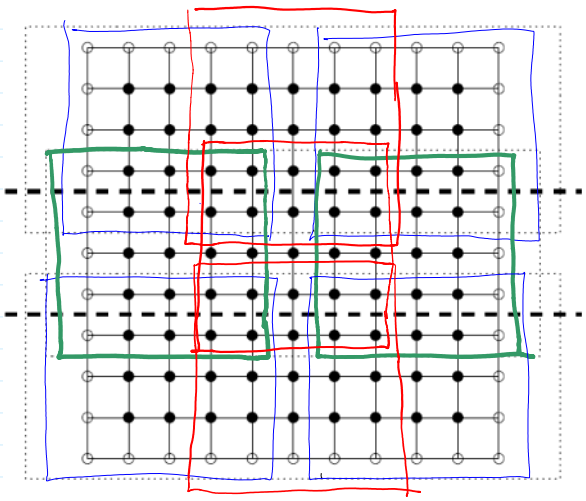

```
In[39]:= ListPlot3D[dat5, PlotRange → All, BoxRatios → {4, 1, 1}]
```

Out[39]=



note: exact solution here is $u = 1$ so error ~ 0.001, considerably larger than the change in one iteration.

2D Decomposition



9 overlapping meshs

Figure 1: Partition of a finite difference mesh across 9 processes.

Q1: We now need to create a mapping from the list of processors to the topology of the system.

Q2: If the computer architecture or network has some natural topology matching our problem we may want to take advantage of it.

How can we do this?

- **MPI_Cart_create** (***MPI_Comm*** comm_old, ***int*** ndims, ***int*** *dims, ***int*** *periods, ***int*** reorder, ***MPI_Comm*** *comm_cart)

Makes a new communicator to which topology information has been attached

MPI_Cart_create creates a Cartesian decomposition of the processes.
comm_old input communicator (handle)
ndims number of dimensions of cartesian grid (integer)
dims integer array of size ndims specifying the number of processes in each dimension
periods logical array of size ndims specifying whether the grid is periodic (true) or not (false) in each dimension
reorder ranking may be reordered (true) or not (false) (logical)

e.g. MPI_Cart_create(MPI_COMM_WORLD, 2, dims, isperiodic, 1, &comm2d);

creates 3x3 decomposition like above.

start with default communicator

2 dimensional decomposition

array {3,3}

array {0,0}

reorder to take advantage of any system topology

our new communicator (should be previously declared with type MPI_Comm

NB: 1) MPI_Comm_rank should be called after the create command, as rank could change with reorder.
2) The new communicator "comm2d" should now replace MPI_COMM_WORLD in MPI commands.

- **MPI_Cart_coords** (MPI_Comm comm, ***int*** rank, ***int*** maxdims, ***int*** *coords)

Determines process coords in Cartesian topology given ranks in group

MPI_Cart_coords takes input as a rank in a communicator, returns the coordinates of the process with that rank. MPI_Cart_coords is the inverse to MPI_Cart_Rank; it returns the coordinates of the processes with rank rank in the Cartesian communicator comm. Note that both of these functions are local.

- **MPI_Cart_rank** (***MPI_Comm*** comm, ***int*** *coords, ***int*** *rank)

Determines process rank in communicator given Cartesian location

MPI_Cart_rank returns the rank in the Cartesian communicator comm of the process with Cartesian coordinates. So coordinates is an array with order equal to the number of dimensions in the Cartesian topology associated with comm.

- **MPI_Cart_get** (***MPI_Comm*** comm, ***int*** maxdims, ***int*** *dims, ***int*** *periods, ***int*** *coords)

Retrieve Cartesian topology information associated with a communicator

MPI_Cart_get retrieves the coordinates of the calling process in communicator.

- **MPI_Cart_shift** (***MPI_Comm*** comm, ***int*** direction, ***int*** disp, ***int*** *rank_source, ***int*** *rank_dest)

Returns the shifted source and destination ranks given a shift direction and amount

MPI_Cart_shift returns rank of source and destination processes in arguments rank_source and rank_dest respectively.

Returns the shifted source and destination ranks given a shift direction and amount

MPI_Cart_shift returns rank of source and destination processes in arguments rank_source and rank_dest respectively.

e.g. MPI_Cart_shift(comm2d, 0, 1, &nbrleft, &nbrright);

x-direction; step; processor to left of me (this is the output); processor to right of me (also output of this routine)

i.e. this allows you to find out who the neighboring processors in the topology are.

In the 1D decomposition, we sent boundary rows to neighboring processors. In 2D we will need to send both rows (to upper/lower neighbor) and boundary columns to neighbors to right & left.

eg:

```
MPI_Isend(&old[size-1][1],N-1,MPI_FLOAT,nid+1,10,MPI_COMM_WORLD,&req_send10);
```

beginning of boundary row; # elements in the row

This worked as elements of a row are in consecutive memory locations. This is not true of a column.

We resolve this problem by creating a new data type:

MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype)

[IN count] number of blocks (nonnegative integer)
[IN blocklength] number of elements in each block (nonnegative integer)
[IN stride] number of elements between start of each block (integer)
[IN oldtype] old datatype (handle)
[OUT newtype] new datatype (handle)

MPI_Type_commit(&newtype)

Tell the system about our new type and optimize for communication.

eg.

29	30	31	32	33	34	35
22	23	24	25	26	27	28
15	16	17	18	19	20	21
8	9	10	11	12	13	14
1	2	3	4	5	6	7

#'s indicate consecutive memory locations

suppose we have data in a 2d array declared as double dat[5][7]

To send a column of dat, say the one indicated in yellow, we first define a new type "stridetype" via

MPI_Type_vector(5, 1, 7, MPI_DOUBLE, &stridetype);

previously declared as type MPI_Datatype

then

MPI_Type_commit(&stridetype);

now when you want to send the column, you could

MPI_Isend(&dat[1][4], 1, stridetype, procforecv, tag, commwie, &req);

note this is 1 of this type & that type gets whole column

this is just a regular double array (data is "reinterpreted" as of type stridetype)

note the implications of being allowed to do this: there is no checking you are not overstepping the array bounds.

```
// Solve Poisson's equations using the Jacobi method, with MPI parallization
```

```
// dividing the grid into a 1D array of processors
// Compile line:
// mpicc -O3 PoissonJacobi2DMPI.c -lm -o pois2d

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <float.h>
#include <limits.h>
#include "mpi.h"
#define N  8              /* N x N is size of interior grid */
#define Tol  0.00001

int isperiodic[2]={0,0};  // Flags for periodic boundary conditions are off

// MPI Stuff
struct mpi_vars {
  int NProcs;
  int MyID;
  MPI_Comm D1Comm; // 1D communicator
  int nbrdown, nbrup;

  MPI_Comm D2Comm; // 2D communicator
  int dims[2], IDcoord[2]; //  processors in each direction in 2D
  int nbrleft, nbrright;
};

struct mpi_vars mpi_start(int argc, char** argv, int dim)
{
  struct mpi_vars mpi;

  MPI_Init(&argc, &argv);

  // We always want a 1D array of processors so just duplicate default
  MPI_Comm_dup(MPI_COMM_WORLD, &mpi.D1Comm);

  MPI_Comm_size(mpi.D1Comm, &mpi.NProcs);
  if (dim == 1) {
    MPI_Comm_rank(mpi.D1Comm, &mpi.MyID);
    mpi.nbrup = mpi.MyID + 1;
    mpi.nbrdown = mpi.MyID -1;

    mpi.D2Comm = MPI_COMM_NULL;
  }

  // We sometimes want a 2D array so define if requested
  if (dim == 2) {
    // Find an integer closest to the square root of the processor number
    mpi.dims[0] = (int) (sqrt(((double)mpi.NProcs))+2.*FLT_EPSILON);
    // Assign as many as possible to other dimension (could leave some unused)
    mpi.dims[1] = mpi.NProcs/mpi.dims[0];

    /* Create cartesian grid of processors */
    MPI_Cart_create(MPI_COMM_WORLD, 2, mpi.dims, isperiodic, 1, &mpi.D2Comm);
    MPI_Comm_rank(mpi.D2Comm, &mpi.MyID); //need rank from 2D Comm
    if (mpi.MyID == 0)
      printf("number of processors %i: %i x %i\n",
        mpi.NProcs, mpi.dims[0], mpi.dims[1]);

    MPI_Cart_coords(mpi.D2Comm, mpi.MyID, 2, mpi.IDcoord);
    MPI_Cart_shift(mpi.D2Comm, 0, 1, &mpi.nbrleft, &mpi.nbrright);
    MPI_Cart_shift(mpi.D2Comm, 1, 1, &mpi.nbrdown, &mpi.nbrup);
  }

  return mpi;
}


// breakup array size N x N in 2D into parts for each processor
void compute_my_size(int *xsize, int *ysize, struct mpi_vars mpi)
{
  int xremainder = N  % mpi.dims[0];
  *xsize = (N - xremainder)/mpi.dims[0];
  if(mpi.IDcoord[0] < xremainder)
```

explicitly add a 1D communicator (copy of MPI_COMM_WORLD) but makes this independent of any future change to our program into a routine for another program

NEW

who is above/below us (note: no bounds check)

attempt to create a square grid of processors (or as close as possible) obviously 4, 9, 16, ok processors will be best

push up to avoid roundoff & chop dropping us down

will use this communicator mostly

who are the neighbors?

division along x & y

```
    *xsize = *xsize + 2;
  else
    *xsize = *xsize + 1;

  int yremainder = N  % mpi.dims[1];
  *ysize = (N - yremainder)/mpi.dims[1];
  if(mpi.IDcoord[1] < yremainder)
    *ysize = *ysize + 2;
  else
    *ysize = *ysize + 1;

  printf("compute grid  size (%i + 1) x (%i + 1) set on processor %i \n",
   *xsize,*ysize,mpi.MyID);
}


double **matrix(int m, int n)
{
    /* Note that you must allocate the array as one block in order to use */
    /* MPI derived data types on them.  Note also that calloc initializes */
    /* all entries to zero. */

    double ** ptr = (double **)calloc(m, sizeof(double *));
    ptr[0]=(double *)calloc(m*n, sizeof(double));
    for(int i = 1; i < m ;i++)
      ptr[i]=ptr[i-1]+n;
    return (ptr);
}


void initialize(double **old,double **new,int xsize,int ysize,struct mpi_vars mpi)
{ /* Assume interior is already initialized, just need to set boundaries */
  /* B.C.s set to a constant, 1 here */
  /* left edge */
  if (mpi.IDcoord[0] == 0)
    for (int i=0; i< ysize + 1; i++)
      new[i][0]=old[i][0]=1.0;
  /* right edge */
  if (mpi.IDcoord[0] == mpi.dims[0]-1)
    for (int i=0; i< ysize + 1; i++)
      new[i][xsize]=old[i][xsize]=1.0;
  /* bottom edge */
  if(mpi.IDcoord[1] == 0)
    for(int j = 1; j < xsize+1; j++)
      new[0][j] = old[0][j] = 1.0;
  /* top edge */
  if(mpi.IDcoord[1] == mpi.dims[1] - 1)
    for(int j = 1; j < xsize+1; j++)
      new[ysize][j] = old[ysize][j] = 1.0;

  printf("boundary conditions set on processor %i\n",mpi.MyID);
}


double iteration(double **old, double **new, int xstart, int xfinish,
     int ystart, int yfinish)
{  /* Jacobi iteration */

  double maxerr = 0;
  for(int i = ystart; i < yfinish; i++)
    for(int j = xstart; j < xfinish; j++){
      new[i][j] = 0.25*(old[i+1][j] + old[i-1][j] +
        old[i][j+1] + old[i][j-1]);
      maxerr = fmax(maxerr,fabs(new[i][j] - old[i][j]));
    }
  return (maxerr);
}


/* Output result */
void output(double **new, int xsize, int ysize, struct mpi_vars mpi)
{
  char str[20];
  FILE *fp;
```

could comment this out but useful for debugging

```
  sprintf(str,"result%d_%d.txt",mpi.IDcoord[0],mpi.IDcoord[1]);
  fp = fopen(str,"wt");
  if(mpi.IDcoord[1] == 0) {
    if (mpi.IDcoord[0] == 0)
      fprintf(fp,"%6.4f ",new[0][0]);
    for(int j = 1; j < xsize; j++)
      fprintf(fp,"%6.4f ",new[0][j]);
    if (mpi.IDcoord[0] == mpi.dims[0]-1)
      fprintf(fp,"%6.4f ",new[0][xsize]);
    fprintf(fp,"\n");
  }
  for(int i = 1; i < ysize; i++) {
    if (mpi.IDcoord[0] == 0)
      fprintf(fp,"%6.4f ",new[i][0]);
    for(int j = 1; j < xsize; j++)
      fprintf(fp,"%6.4f ",new[i][j]);
    if (mpi.IDcoord[0] == mpi.dims[0]-1)
      fprintf(fp,"%6.4f ",new[i][xsize]);
    fprintf(fp,"\n");
  }
  if(mpi.IDcoord[1] == mpi.dims[1] - 1) {
    if (mpi.IDcoord[0] == 0)
      fprintf(fp,"%6.4f ",new[ysize][0]);
    for(int j = 1; j < xsize; j++)
      fprintf(fp,"%6.4f ",new[ysize][j]);
    if (mpi.IDcoord[0] == mpi.dims[0]-1)
      fprintf(fp,"%6.4f ",new[ysize][xsize]);
    fprintf(fp,"\n");
  }
  fclose(fp);
}


int main(int argc, char** argv)
{
  // Start MPI with a 2D cartesian grid of processors
  struct mpi_vars mpi = mpi_start(argc, argv, 2);

  MPI_Status status;
  MPI_Request req_send10, req_send20, req_recv10, req_recv20;
  MPI_Request req_send30, req_send40, req_recv30, req_recv40;

  int maxit=200;

  /* breakup compute grid amoung processors */
  int xsize,ysize;
  compute_my_size(&xsize, &ysize, mpi);
  double **new = matrix(ysize+1,xsize+1);
  double **old = matrix(ysize+1,xsize+1);
  double **tmp; //used for swapping old and new arrays
  initialize(old, new, xsize, ysize, mpi);

  /* Create data type to send columns of data */
  /* If you didn't get this to work, look at the comment in the matrix */
  /* allocation routine */
  MPI_Datatype stridetype;
  MPI_Type_vector(ysize-1,1,xsize+1,MPI_DOUBLE,&stridetype);
  MPI_Type_commit(&stridetype);



  /* do one iteration and work out global error */
  double maxerrG;
  double maxerr = iteration(old,new,1,xsize,1,ysize);
  MPI_Allreduce(&maxerr,&maxerrG,1,MPI_DOUBLE,MPI_MAX,mpi.D2Comm);
  if (mpi.MyID == 0) printf("initial error %f\n",maxerrG);

  /* Main loop */
  int iter=0;
  while(maxerrG > Tol && iter < maxit){
    tmp = new;
    new = old;
    old = tmp;
```

```
    /* send/receive at top of compute block */
    req_send10 = req_recv20 = MPI_REQUEST_NULL;
    if(mpi.IDcoord[1] < mpi.dims[1]-1){
      MPI_Isend(&old[ysize-1][1],xsize-1,MPI_DOUBLE,mpi.nbrup,10,mpi.D2Comm,
&req_send10);
      MPI_Irecv(&old[ysize][1],xsize-1,MPI_DOUBLE,mpi.nbrup,20,mpi.D2Comm,&req_recv20);
    }
    /* send/receive at bottom of compute block */
    req_send20 = req_recv10 = MPI_REQUEST_NULL;
    if(mpi.IDcoord[1] > 0){
      MPI_Isend(&old[1][1],xsize-1,MPI_DOUBLE,mpi.nbrdown,20,mpi.D2Comm,&req_send20);
      MPI_Irecv(&old[0][1],xsize-1,MPI_DOUBLE,mpi.nbrdown,10,mpi.D2Comm,&req_recv10);
    }
    /* send/receive at right of compute block */
    req_send30 = req_recv40 = MPI_REQUEST_NULL;
    if(mpi.IDcoord[0] < mpi.dims[0]-1){
      MPI_Isend(&old[1][xsize-1],1,stridetype,mpi.nbrright,30,mpi.D2Comm,&req_send30);
      MPI_Irecv(&old[1][xsize],1,stridetype,mpi.nbrright,40,mpi.D2Comm,&req_recv40);
    }
    /* send/receive at left of compute block */
    req_send40 = req_recv30 = MPI_REQUEST_NULL;
    if(mpi.IDcoord[0] > 0){
      MPI_Isend(&old[1][1],1,stridetype,mpi.nbrleft,40,mpi.D2Comm,&req_send40);
      MPI_Irecv(&old[1][0],1,stridetype,mpi.nbrleft,30,mpi.D2Comm,&req_recv30);
    }
    /* update interior of compute block except beside boundaries */
    maxerr = iteration(old,new,2,xsize-1,2,ysize-1);

    /* fill in compute block beside boundaries as available */
    /* top edge */
    if(mpi.IDcoord[1] < mpi.dims[1]-1) MPI_Wait(&req_recv20,&status);
    maxerr = fmax(maxerr, iteration(old,new,2,xsize-1,ysize-1,ysize));

    /* bottom edge */
    if(mpi.IDcoord[1] > 0) MPI_Wait(&req_recv10,&status);
    maxerr = fmax(maxerr, iteration(old,new,2,xsize-1,1,2));

    /* right edge */
    if(mpi.IDcoord[0] < mpi.dims[0]-1) MPI_Wait(&req_recv40,&status);
    maxerr = fmax(maxerr, iteration(old,new,xsize-1,xsize,1,ysize));

    /* left edge */
    if(mpi.IDcoord[0] > 0) MPI_Wait(&req_recv30,&status);
    maxerr = fmax(maxerr, iteration(old,new,1,2,1,ysize));

    /* find global error */
    MPI_Allreduce(&maxerr,&maxerrG,1,MPI_DOUBLE,MPI_MAX,mpi.D2Comm);
    if (mpi.MyID == 0) printf(" error %f\n",maxerrG);

    iter++;
  }

  /* Output result */
  output(new, xsize, ysize, mpi);

  free(old[0]);
  free(old);
  free(new[0]);
  free(new);
  MPI_Finalize();
  return 0;
}
```

Similar to row interchange except using stridetype for columns

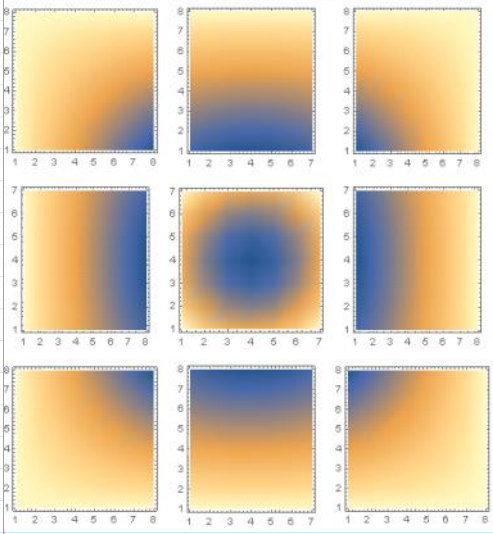
Which can be read in and plotted as: (here using Mathematica)

```
dat0 = ReadList["result0_0.txt", Number, RecordLists → True];
dat1 = ReadList["result0_1.txt", Number, RecordLists → True];
dat2 = ReadList["result0_2.txt", Number, RecordLists → True];
dat3 = ReadList["result1_0.txt", Number, RecordLists → True];
dat4 = ReadList["result1_1.txt", Number, RecordLists → True];
dat5 = ReadList["result1_2.txt", Number, RecordLists → True];
dat6 = ReadList["result2_0.txt", Number, RecordLists → True];
dat7 = ReadList["result2_1.txt", Number, RecordLists → True];
dat8 = ReadList["result2_2.txt", Number, RecordLists → True];

GraphicsGrid[
 {{ListDensityPlot[Transpose[dat6], AspectRatio → Dimensions[dat4][[1]] / Dimensions[dat4][[2]]],
   ListDensityPlot[Transpose[dat7], AspectRatio → Dimensions[dat3][[1]] / Dimensions[dat3][[2]]],
   ListDensityPlot[Transpose[dat8], AspectRatio → Dimensions[dat3][[1]] / Dimensions[dat3][[2]]]},
  {ListDensityPlot[Transpose[dat3], AspectRatio → Dimensions[dat3][[1]] / Dimensions[dat3][[2]]],
   ListDensityPlot[Transpose[dat4], AspectRatio → Dimensions[dat4][[1]] / Dimensions[dat4][[2]]],
   ListDensityPlot[Transpose[dat5], AspectRatio → Dimensions[dat3][[1]] / Dimensions[dat3][[2]]]},
  {ListDensityPlot[Transpose[dat0], AspectRatio → Dimensions[dat6][[1]] / Dimensions[dat6][[2]]],
   ListDensityPlot[Transpose[dat1], AspectRatio → Dimensions[dat5][[1]] / Dimensions[dat5][[2]]],
   ListDensityPlot[Transpose[dat2], AspectRatio → Dimensions[dat4][[1]] / Dimensions[dat4][[2]]]}}]
```

note we need to transpose data as this program expects opposite ordering of rows/columns

3x3 grid note ordering:

20 21 22
10 11 12
00 01 02



Output

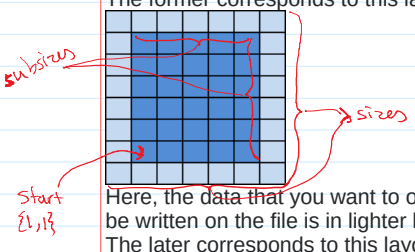
Our output routines are still fairly inadequate. Rather than output each local grid into a separate file it would be much better to get the complete global grid into a single output file.

There are a couple of issues with doing this:
a) We don't actually want the entire local grid, just it's interior (i.e. excluding ghost/halo boundaries)
b) The local grids may be of unequal size

It turns out to be easier to adjust the lattice spacing (Δx or h) to ensure all local grids are the same than to have to deal with different sizes so we will assume they are all the same.

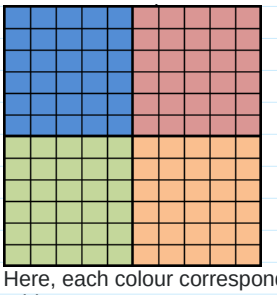
You have actually two very distinct *masks* to consider:
1. The mask for the local data, excluding the halo layers; and
2. The mask for the global data, as it should be once collated into the file.

The former corresponds to this layout:



local grid on a single processor

Here, the data that you want to output on the file for a given process in in dark blue, and the halo layer that should not be written on the file is in lighter blue.
The later corresponds to this layout:



how global mesh is assembled from 2x2 grid of processors

Here, each colour corresponds to the local data coming from a different process, as distributed on the 2D Cartesian grid.
To understand what you need to create to reach this final result, you have to think backwards:

1. Your final call to the IO routine will be `MPI_File_write_all(fh, &A[0][0], 1, interior, MPI_STATUS_IGNORE);`. So you have to have your interior type defined such as to exclude the halo boundary. We will define a MPI type grid to do this. (file functions will be discussed below)
2. But now, you have to have the view on the file to allow for this `MPI_File_write_all()` call. So the view must be as described in the second picture. We will therefore create a new MPI type representing it. For that, `MPI_Type_create_subarray()` is what we need.

Here is the synopsis of this function:

```
int MPI_Type_create_subarray(int ndims,
                             const int array of sizes[],
                             const int array of subsizes[],
                             const int array_of_starts[],
                             int order,
                             MPI_Datatype oldtype,
                             MPI Datatype *newtype)
Create a datatype for a subarray of a regular, multidimensional array
INPUT PARAMETERS
  ndims    - number of array dimensions (positive integer)
  array_of_sizes
           - number of elements of type oldtype in each
             dimension of the full array (array of positive integers)
  array_of_subsizes
           - number of elements of type oldtype in each dimension of
             the subarray (array of positive integers)
  array of starts
           - starting coordinates of the subarray in each dimension
             (array of nonnegative integers)
  order    - array storage order flag (state)
  oldtype - array element datatype (handle)
OUTPUT PARAMETERS
  newtype - new datatype (handle)
```

Based on: <https://stackoverflow.com/questions/33537451/writing-distributed-arrays-using-mpi-io-and-cartesian-topology>

For our Jacobi solver using a 2D array of processors the local subarray type is defined by:

```
  /* Create derived datatype for local interior grid (output grid) */
  MPI_Datatype grid;
  int start[2] = {1, 1};  // indices of interior "origin"
  int arrsize[2] = {xsize+1, ysize+1};  // full local array size
  int gridsize[2] = {xsize+1 - 2, ysize+1 - 2};  // size of interior

  MPI_Type_create_subarray(2, arrsize, gridsize,
           start, MPI_ORDER_FORTRAN, MPI_DOUBLE, &grid);
  MPI_Type_commit(&grid);
```

Note: I picked MPI_ORDER_FORTRAN rather than MPI_ORDER_C. This is because we picked our indices as A[y-index][x-index] which is consistent with A[row-index][column-index] for a matrix (except that row-index=0 is at the top and y-index=0 is at the bottom). However, when interpreted as spatial indices C-order is assumed to be A[x][y] so our order is actually Fortran equivalent.

For our global array we also need a subarray type describing how the local grids fit into the global grid.

We will use this type for the "file view", which we will simply apply with `MPI_File_set_view(fh, 0, MPI_DOUBLE, view, "native", MPI_INFO_NULL);` and the magic is done. More details on this are given below.

```
/* Create derived type for file view, how local interior fits into global system */
 MPI_Datatype view;
  int nnx = xsize+1-2, nny = ysize+1-2;
  int startV[2] = { mpi.IDcoord[0]*nnx, mpi.IDcoord[1]*nny };
  int arrsizeV[2] = { mpi.dims[0]*nnx, mpi.dims[1]*nny };
  int gridsizeV[2] = { nnx, nny };

  MPI_Type_create_subarray(2, arrsizeV, gridsizeV,
                           startV, MPI_ORDER_FORTRAN, MPI_DOUBLE, &view);
  MPI_Type_commit(&view);
```

Some of the stuff below based on: https://warwick.ac.uk/fac/sci/csc/people/computingstaff/chris_brady/ampi.pdf

Weaknesses of MPI Type create subarray

- At first sight, MPI Type create subarray looks like it solves all problems, but it has problems all of its own
- array of sizes must be the same on every process
- array of subsizes must be the same on every process
- Therefore, MPI Type create subarray is simply to do uniform, even subdivision of an array, with an identical fraction of the array being referred to on each processor
- In fact, it's main purpose is in MPIIO, where it is used to represent the subsection of a global array held by each processor
- It does still work in communication, although it's not quite as useful as it might be.

MPIIO Advantages

- Improved output speed in large parallel environments
- Output to a single file for any number of processors (easily)

MPIIO Disadvantages

- Syntax is not exactly like either C or FORTRAN IO (although similar in concept)
- Can be slower on desktop machines
- Produces C type binary output

MPIIO Concepts

- In most senses MPIIO is the same as conventional binary IO
- There are commands to open and close files, read and write data and move file pointers
- There are things called file views which describe the layout of data across processors
- There are some commands to help with writing simple data layouts more easily than using file views, but they will not be covered.
- File views are described using MPI_types

```
int MPI_File_open(MPI_Comm Comm, char *filename, int amode, MPI_Info info, MPI_File *mpi_fh)
```

Description

- Opens a file using MPIIO and returns a file handle mpi_fh.
- This is a collective operation. All processes must have the same amode and the filename must reference the same file (does not have to be the same filename)
- Choosing whether opening the file for reading or writing is via constants passed as part of amode as normal
- info is used to pass additional parameter, which will generally vary from system to system. You create MPI Info objects using the MPI Info commands. Most generally, you can use MPI_INFO_NULL to open the file generically.

MPI File open modes

- MPI_MODE_RDONLY - Open for reading
- MPI_MODE_RDWR - Open for reading and writing
- MPI_MODE_WRONLY - Open for writing
- MPI_MODE_CREATE - Create the file if it does not exist,
- MPI_MODE_EXCL - Throw error if file exists
- MPI_MODE_DELETE_ON_CLOSE - Delete the file when its closed
- MPI_MODE_UNIQUE_OPEN - Throw error if file opened anywhere else
- MPI_MODE_SEQUENTIAL - Sequential mode (tapes etc.)
- MPI_MODE_APPEND - Set initial position of all file pointers to end of file

```
int MPI_File_close(MPI_File *mpi_fh)
```

Description

- Closes and frees the file handle mpi_fh
- File handles must be closed before MPI is finalized. Otherwise behaviour is undefined.

```
int MPI_File_write_all(MPI_File mpi_fh, void *buf, int count, MPI_Datatype datatype,
```

MPI_Status *status)
Description
• Writes to the file pointed to by the handle mpi_fh using the individual file pointer
• This is a collective operation and all the processors in the MPI_Comm given to MPI_File open must call MPI_File_write_all or the code will lock (i.e. this is a blocking operation).

What are file views?
• If you wish to have more control over where a given processor will write it's data then you have to use file views.
• File views use MPI types to describe the section of the global data that the current processor has, and it's location in the final file on disk
• Makes it easier to write multidimensional arrays into a single file

```
int MPI_File_set_view(MPI_File mpi_fh, MPI_Offset offset, MPI_Datatype etype,
MPI_Datatype filetype, char* datarep, MPI_Info info)
```
Description
• Sets the file view on file handle mpi_fh
• offset is the offset from the start of the file at which to apply the file view
• datarep is a string describing the data representation, usually "native"
• info contains additional information, will usually be MPI_INFO_NULL
• etype is the basic datatype being written to the file
• filetype is (normally) a derived datatype which describes the layout of the data for the current processor on the disk

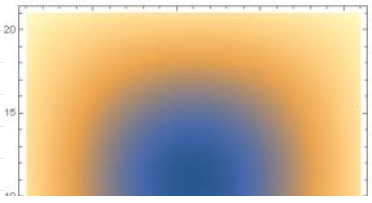
So our final output routine will be:

```
void output_onefile(double **new, int xsize, int ysize, struct mpi_vars mpi)
{
  /* Create derived datatype for local interior grid (output grid) */
  MPI_Datatype grid;
  int start[2] = {1, 1};  // indices of interior "origin"
  int arrsize[2] = {xsize+1, ysize+1};  // full local array size
  int gridsize[2] = {xsize+1 - 2, ysize+1 - 2};  // size of interior

  MPI_Type_create_subarray(2, arrsize, gridsize,
           start, MPI_ORDER_FORTRAN, MPI_DOUBLE, &grid);
  MPI_Type_commit(&grid);

  // Create derived type for file view, how local interior fits into global system
  MPI_Datatype view;
  int nnx = xsize+1-2, nny = ysize+1-2;
  int startV[2] = { mpi.IDcoord[0]*nnx, mpi.IDcoord[1]*nny };
  int arrsizeV[2] = { mpi.dims[0]*nnx, mpi.dims[1]*nny };
  int gridsizeV[2] = { nnx, nny };

  MPI_Type_create_subarray(2, arrsizeV, gridsizeV,
                           startV, MPI_ORDER_FORTRAN, MPI_DOUBLE, &view);
  MPI_Type_commit(&view);

  /* MPI IO */
  MPI_File fp;

  MPI_File_open(mpi.D2Comm, "output.bin", MPI_MODE_CREATE | MPI_MODE_WRONLY,
                MPI_INFO_NULL, &fp);

  MPI_File_set_view(fp, 0, MPI_DOUBLE, view, "native", MPI_INFO_NULL);
  MPI_File_write_all(fp, &new[0][0], 1, grid, MPI_STATUS_IGNORE);
  MPI_File_close(&fp);
}
```

Note: i) This should only be used when all local grids are the same

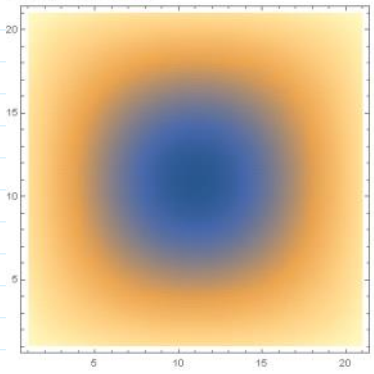
With xremainder= yremainder = 0 in Jacobi 2D this gives:

```
dat2 = BinaryReadList["output.bin", Table["Real64", {i, 1, 21}]];
ListDensityPlot[dat2]
```



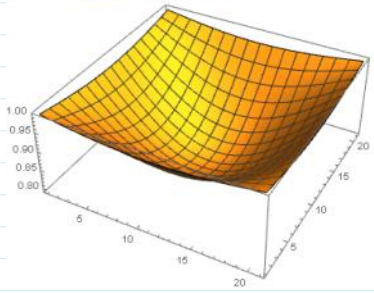
Note: Output is now in binary not text.

```
dat2 = BinaryReadList["output.bin", Table["Real64", {i, 1, 21}]];
ListDensityPlot[dat2]
```



Note: Output is now in binary not text.

```
ListPlot3D[dat2, PlotRange → All]
```



If you really want to solve this for a large system a better algorithm is Conjugate-Gradient Method

Consider $f(x) = \frac{1}{2} x^t A x - b^t x + c$ (suppose A is positive definite) in tensor notation

$f(x) = \frac{1}{2} x_\alpha A_{\alpha\beta} x_\beta - b_\alpha x_\alpha + c$ (summation convention assumed)

$\partial_\gamma f = \frac{1}{2}\delta_{\alpha\gamma} A_{\alpha\beta} x_\beta + \frac{1}{2} x_\alpha A_{\alpha\beta}\delta_{\beta\gamma} - b_\alpha \delta_{\alpha\gamma}$

$= \frac{1}{2} A_{\gamma\beta} x_\beta + \frac{1}{2} A_{\alpha\gamma} x_\alpha - b_\gamma$

$= A_{\gamma\beta} x_\beta - b_\gamma$

$\Rightarrow$ solving $\partial_\gamma f = 0$ for all γ (i.e. to minimize f)

is equivalent to solving $Ax = b$ as $\nabla f = Ax - b$.

Let's try to find approximate ways to minimize f:

Line Search

$x_\alpha^{(m+1)} = x_\alpha^{(m)} + \lambda_m d_\alpha^{(m)}$ (iteration label; vector coordinate)

and pick the scalar λ_m to minimize f along the line determined by the vector d.

(drop (m) index temporarily)

$f(x_\alpha + \lambda_m d_\alpha) = \frac{1}{2}(x_\alpha + \lambda d_\alpha) A_{\alpha\beta}(x_\beta + \lambda d_\beta) - b_\alpha(x_\alpha + \lambda d_\alpha) + c$

note A is symmetric so $d_\alpha A_{\alpha\beta} x_\beta = x_\alpha A_{\alpha\beta} d_\beta$ so

$= f(x_\alpha) + \lambda^2\left(\frac{1}{2} d_\alpha A_{\alpha\beta} d_\beta\right) + \lambda d_\alpha (A_{\alpha\beta} x_\beta - b_\alpha)$

$f(x_\alpha)$: some number; $\frac{1}{2} d_\alpha A_{\alpha\beta} d_\beta > 0$ as A +ive definite; $A_{\alpha\beta} x_\beta - b_\alpha \equiv r_\alpha^{(m)} = \partial_\alpha f(x^{(m)})$

$\Rightarrow$ upward parabola if plotted as function of λ

Min at $\frac{\partial f}{\partial \lambda} = 0 \Rightarrow \lambda d_\alpha^{(m)} A_{\alpha\beta} d_\beta^{(m)} + d_\alpha^{(m)} r_\alpha^{(m)} = 0$

$$\Rightarrow \lambda_m = - \frac{d_\alpha^{(m)} r_\alpha^{(m)}}{d_\alpha^{(m)} A_{\alpha\beta} d_\beta^{(m)}} \quad \text{for min.}$$

What direction should we search in?
It makes sense to take $d_\alpha^{(0)}$ along (-) gradient $-\partial_\alpha f(x^{(0)}) = -r_\alpha^{(0)}$ i.e. in downhill direction
Subsequent searches are along:

$$d_\alpha^{(m+1)} = -r_\alpha^{(m+1)} + \alpha_m d_\alpha^{(m)}$$

($r_\alpha^{(m+1)}$: gradient at current min; $\alpha_m d_\alpha^{(m)}$: remove directions we have already searched.)

How? Choose $d^{(m+1)}$ to be A-conjugate to $d^{(m)}$

i.e. $d_\alpha^{(m+1)} A_{\alpha\beta} d_\beta^{(m)} = 0$

$$\Rightarrow \left(-r_\alpha^{(m+1)} + \alpha_m d_\alpha^{(m)}\right) A_{\alpha\beta} d_\beta^{(m)} = 0$$

$$\text{or} \quad \alpha_m = \frac{r_\alpha^{(m+1)} A_{\alpha\beta} d_\beta^{(m)}}{d_\alpha^{(m)} A_{\alpha\beta} d_\beta^{(m)}}$$

This gives the following algorithm:

initial residual — initial guess

$r^{(0)} = A x^{(0)} - b$

$d^{(0)} = -r^{(0)}$ — downhill direction

for $m = 0, 1, 2, \ldots$

$$\lambda_m = \frac{-d^{(m)T} r^{(m)}}{d^{(m)T} A d^{(m)}}$$ — matrix-vector product

$x^{(m+1)} = x^{(m)} + \lambda_m d^{(m)}$

$r^{(m+1)} = A x^{(m+1)} - b$

if $\sqrt{r^{(m+1)T} r^{(m+1)}} < tol$

  output $x^{(m+1)}$

  stop

$$\alpha_m = \frac{r^{(m+1)T} A d^{(m)}}{d^{(m)T} A d^{(m)}}$$ — same product as before, so only do once & store in a vector

$d^{(m+1)} = -r^{(m+1)} + \alpha_m d^{(m)}$ — a different matrix-vector product

note that $x^{(m+1)} = x^{(m)} + \lambda_m d^{(m)}$

so $\underbrace{Ax^{(m+1)} - b}_{r^{(m+1)}} = \underbrace{Ax^{(m)} - b}_{r^{(m)}} + \lambda_m A d^{(m)}$

$$r^{(m+1)} = r^{(m)} + \lambda_m A d^{(m)}$$

same as other matrix-vector product

i.e. should replace $r^{(m+1)}$ calculation to make use of matrix-vector product already used.

It is also possible to show that:

i) $d^{(m)T} r^{(m+1)} = 0$

i.e. previous search direction & new residual are orthogonal

ii) $r^{(m)T} r^{(l)} = 0 \quad$ for $m \neq l$

i.e. all residuals are orthogonal
(⇒ we are eliminating residuals along a specific direction at every step)

iii) With i) & ii), it is possible to show that

$$\lambda_m = \frac{r^{(m)T} r^{(m)}}{d^{(m)T} A d^{(m)}}$$

$$\alpha_m = \frac{r_\alpha^{(m+1)} r_\alpha^{(m+1)}}{r_\beta^{(m)} r_\beta^{(m)}}$$

With these results we can simplify the algorithm to get

$r^{(0)} = A x^{(0)} - b$

$d^{(0)} = -r^{(0)}$

$\delta^{(0)} = r^{(0)T} r^{(0)}$

for $m = 0, 1, 2, \ldots$

- $u = A d^{(m)}$ ← note: • A is sparse, so this should be written out, not as a true matrix product. • THIS IS THE PARALLIZABLE PART
- $\lambda_m = \dfrac{\delta^{(m)}}{d^{(m)T} u}$ ← do locally then AllReduce
- $x^{(m+1)} = x^{(m)} + \lambda_m d^{(m)}$
- $r^{(m+1)} = r^{(m)} + \lambda_m u$ → need only local values
- $\delta^{(m+1)} = r^{(m+1)T} r^{(m+1)}$ → do locally then AllReduce
- if $\sqrt{\delta^{(m+1)}} < tol$
  - output $x^{(m+1)}$
  - stop
- $\alpha_m = \dfrac{\delta^{(m+1)}}{\delta^{(m)}}$
- $d^{(m+1)} = -r^{(m+1)} + \alpha_m d^{(m)}$ → local

How many iterations are typically needed?

Not many! Why?

Suppose initial guess is $x^{(0)} = \begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix}$, then

m

Suppose initial guess is $x^{(0)} = \begin{pmatrix} \vdots \\ 0 \end{pmatrix}$, then

$$x^{(m)} = \sum_{k=1}^{m} c_k d^{(k)}$$

but all $d^{(k)}$ are linearly dependent (from i) & ii) above) $\Rightarrow$ if $A = n \times n$, then we can need at most n iterations, and then should terminate.

eg. This leads to the serial code (in C here):

```
/* Compile line:                                */
/*  gcc -O3 CG.c -lm                            */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define N  500
#define Tol  0.0001
#define maxitr 2*N
#define h 0.1

double **matrix(int m, int n)
{
  /* Note that you must allocate the array as one block in order to use */
  /* MPI derived data types on them.  Note also that calloc initializes */
  /* all entries to zero. */

  double **ptr = (double **)calloc(m, sizeof(double *));
  ptr[0]=(double *)calloc(m*n, sizeof(double));
  for(int i = 1; i < m ;i++)
    ptr[i]=ptr[i-1]+n;
  return (ptr);
}


void initialize(double ** r, double **x, int size)
{
  /* Subtract off b from a particularly boring set of boundary conditions where */
  /* x=1 on the boundary */
  for(int i = 1; i < size; i++){
    r[i][1] -= 1;
    r[i][size-1] -= 1;
    x[i][0]=1;    // we won't use x on the boundary but useful for output at end
    x[i][size]=1;
  }
  for(int j = 1; j < size; j++){
    r[1][j] -= 1;
    r[size-1][j] -= 1;
    x[0][j]=1;
    x[size][j]=1;
  }
  x[0][0]=x[0][size]=x[size][0]=x[size][size]=1;    /* fill in the corners */
}
```

← similar to iteration routine in Jacobi algorithm (Ad similar to "new", d similar to "old")

```
void AD(double **Ad, double **d, int start, int finish)
{ // Compute Ad= A*d using the auxiliary layer around the outside that is all zero
  // for d to account for boundary */
  int i, j;
  for(i = start; i < finish; i++)
    for(j = 1; j < N; j++){
      Ad[i][j] = -(d[i+1][j] + d[i-1][j] + d[i][j+1] + d[i][j-1]-4.0*d[i][j]);
    }
  return;
}


void output(double **x, int size)
{
  char str[20];
```

```
  FILE *fp;

  sprintf(str,"Solution.Txt");
  fp = fopen(str,"wt");

  for(int i = 0; i < size+1; i++)
    for(int j = 0; j < size + 1; j++)
      fprintf(fp,"%6.4f\n",x[i][j]);

  fclose(fp);
}


int main(int argc, char** argv)
{
  double delta, lambda, olddelta, alpha;

  /* create arrays and initialize to all zeros */
  /* Note: Although these are column vectors in the algorithm, they represent */
  /* locations in 2D space so it is easiest to store them as 2D arrays */
  double **x = matrix(N+1,N+1);
  double **r = matrix(N+1,N+1);
  double **d = matrix(N+1,N+1);
  double **u = matrix(N+1,N+1);

  /* Work out A x(0), note that we make use of the x being zero on the boundary */
  /* so we don't need the boundaries to be special cases in the iteration */
  AD(r,x,1,N);

  initialize(r,x,N); // setup up boundary conditions (from the -b part of r=Ax-b)

  /* Fill in d */
  for (int i=1; i<N; i++)
    for (int j=1; j<N; j++)
      d[i][j]=-r[i][j];

  /* Compute first delta */
  delta=0.0;
  for (int i=1; i<N; i++)
    for (int j=1; j<N; j++)
      delta+= r[i][j]*r[i][j];
  printf("itr = %i delta = %g\r",0,delta);

  /* Main loop */
  int itr;
  for (itr=0; itr< maxitr; itr++){
    AD(u,d,1,N);   <- this could be parallized similar to Jacobi
                      with new<->u  old<->d

    lambda=0.0;
    for (int i=1; i<N; i++)
      for (int j=1;j<N; j++)
        lambda+= d[i][j]*u[i][j];
    lambda=delta/lambda;   <- would need to collect/sum results here with
                              an All reduce

    for (int i=1; i<N; i++)
      for (int j=1;j<N; j++) {
        x[i][j] += lambda*d[i][j];
        r[i][j] += lambda*u[i][j];
      }

    olddelta=delta;
    delta=0.0;
    for (int i=1; i<N; i++)
      for (int j=1; j<N; j++)
        delta+= r[i][j]*r[i][j];
    printf("itr = %i delta = %g\r",itr,delta);

    if (sqrt(delta) < Tol)
      break;

    alpha=delta/olddelta;
    for (int i=1; i<N; i++)
      for (int j=1;j<N; j++) {
        d[i][j] = -r[i][j]+alpha*d[i][j];
      }
```

local, no change for parallelization

local, no change for parallelization

```
    }
  }

  printf("\nComplete in %d iterations \n",itr);

  /* Output result */
  output(x,N);

  free(r[0]);
  free(r);
  free(d[0]);
  free(d);
  free(u[0]);
  free(u);
  free(x[0]);
  free(x);
  return 0;
}
```

Parallelization of this code is very similar to what we did before:

1. The "matrix-vector" product for the case where A is defined from the Poisson eqn is almost identical to what we did for an "iteration" of the Jacobi algorithm
2. Most other computations are local except for a few dot-products/norms which can easily be done locally and then collected together using an MPI_AllReduce.