UNIVERSITY OF WASHINGTON
Department of Applied Mathematics

AMATH 483/583 — **Assigned:** 5/25/2018
Problem Set #8 — **Due:** 6/01/2018

1 Preliminaries

1.1 MPI with Docker

Our preliminary and warm up exercises will be done with MPI in docker. Although parallel speedup will be limited, regarding correctness and concurrency, MPI in docker will work basically like it would in a truly distributed memory system.

Recall that MPI is about communicating sequential *processes* – those processes can actually be on the same node. And, in fact, when processes are on the same node, most MPI implementation effect message passing via shared memory rather than over the network. Even on shared memory hardware, separate processes cannot directly read and write each other's memory and are thus communicating sequential processes and behave just as if the memory were distributed.

Rather than using the `amath583/base` image for this assignment, you will be using the image `amath583/openmpi`. This image is the same as the one we have been using so far, but also includes the Open MPI implementation of MPI. The steps for launching this image are the same as with the base one. (Make sure to update your vscode settings if you use docker through vscode.) Test out this image after you have launched it by running it and issuing the following command:

```
% mpirun --version
```

You should get back the following message:

```
mpirun (Open MPI) 1.10.2
```

1.2 Compiling and Running an MPI Program

As we have emphasized in lecture, MPI programs are not special programs in any way, other than that they make calls to functions in an MPI software library. An MPI program is a plain old sequential program, regardless of the calls to MPI functions. (This is in contrast to, say, an OpenMP program where there are special directives for compilation into a parallel program.)

Given that, we can compile an MPI program into an executable with the same C++ complier that we would use for a sequential program without MPI function calls. However, using a third-party library (not part of the standard library) requires specifying the location of include files, the location of library archives, which archives to link to, and so forth. Since MPI itself only specifies its API, implementations vary in terms of where their headers and archives are kept. Moreover, it is often the case that a parallel programmer may want to experiment with different MPI implementations with the same program.

It is of course possible (though not always straightforward) to determine the various directories and archives needed for compiling an MPI program. Most (if not all) implementations make this process transparent to the user by providing a "wrapper compiler" – a script (or perhaps a C/C++ program) that properly establishes the compilation environment for its associated MPI implementation. Using the wrapper compiler, as user does not have to specify the MPI environment. The wrapper compiler is simply invoked as any other compiler – and the MPI-associated bits are handled automagically.

The wrapper compilers associated with most MPI implementations are named along the lines of "mpicc" of "mpic++" for compiling C and C++, respectively. These take all of the same options as their underlying (wrapped) compiler. The default wrapped compiler is specified when the MPI implementation is built, but it can usually be over-ridden with appropriate command-line and/or environment settings. For MPI assignments in this course, the compiler is named `mpic++` – the same name is used by both MPICH and Open MPI.

1.3 mpirun

Launch the openmpi docker image. The first MPI program that you will run is not an MPI program at all. At the command prompt type the following:

```
% hostname
```

As in an early assignment, you will get back the hostname of the current docker container. Now, issue the following command:

```
% mpirun -np 4 hostname
```

Important: Under docker, you may see an inscrutable message along the lines of:

```
Unexpected end of /proc/mounts line `overlay / overlay rw,relatime,lowerdir=/var/lib/docker/over
lay2/l/T2BWD63PMBCVAHMK2IP42K4R2I:/var/lib/docker/overlay2/l/LKRUDUKGNDMA4K7I7KR46V
6GPD:/var/lib/docker/overlay2/l/4U2TFSRMUXZ33NTUHUCKTMS3PS:/var/lib/docker/overlay2
/l/JQ7BI4ATXNGCBF4RJFYGLXHKYG:/var/lib/docker/overlay2/l/AQC6WSWNLQECOWXWO3DVZPA5LG
:/var/lib/docker/overlay2/l/A45SA4GKAM5SNQ4WND5I6ZYU37:/var/lib/docker/overlay2/l/Z
2FQHXNCP6XEHOGIWVXLMYFFCL:/var/lib/docker/overlay2/l/32HPGZCE4C6PS'
```

Do not be alarmed by this message – it is innocuous and does not indicate that anything is wrong with your program or with mpirun.

The `-np 4` option instructs `mpirun` to launch four processes. This command (launching four processes running hostname) is a very useful diagnostic for determining whether `mpirun` is functioning (or how it is functioning). It can also be helpful to issue other commands like

```
% mpirun -np 4 pwd
```

What is important in these examples – and why they are diagnostically useful – is that `mpirun` invokes multiple copies (as specified by `-np`) of the indicated program. In a real cluster, these would be on different nodes and you would see different hostnames (although, as with what we are doing here, you might still have multiple processes on the same node). In a real cluster, the node where the MPI processes are launched may have a different working directory than the program where you run `mpirun`. (And remember, `mpirun` is only launching your programs – in a real cluster they probably will be run on different nodes than where you build and launch your programs.)

It is also important to note that we are using `mpirun` programs to launch programs that don't have any MPI function calls in them. Again, in some sense, there is no such thing as "an MPI program." There are only processes, some of which may use MPI library functions to communicate with each other. But `mpirun` does not in any way inspect what the processes actually do, so it will launch programs without MPI just as well as programs that do call MPI.

For your first program to try yourself with `mpirun` type in and compile the following program:

```cpp
#include <iostream>
#include <unistd.h>

int main(){

    std::cout << "Process ID is " << getpid() << std::endl;
    return 0;
}
```

You can compile by just executing `c++ yourprogram.cpp` – it doesn't matter what you name it, and if you don't specify an output name it will create `a.out`. Moreover, there are no MPI function calls, so you don't need to compile it with `mpic++`. Now, run the program by issuing

```
% mpirun -np 4 ./a.out
```

What gets printed? In all of the previous diagnostic programs, all of the processes responded with the same results, so we just got multiple copies of the same results. Now we finally have processes that give visibly different behavior. (In this example the program prints its own process id).

Experiment a bit with different values after `-np` – and with different programs given to `mpirun`. Note that you can pass arguments to the program you are launching just by tacking them onto the end of the command.

```
% mpirun -np 4 ls -CF
```

will run four copies of `ls -CF`.

2 Warm Up

2.1 Hello MPI World

For your first non-trivial (or not completely trivial) MPI program, try the hello world program we presented in lecture, provided as `hello.cpp`. The basic command to compile this program is

```
% mpic++ hello.cpp
```

Note however, that to use all of the other arguments that we have been using in this course with C++ (such as -std=c++11, -Wall, etc), we need to pass those in too.

Thus, to compile hello.cpp you should issue (at least) the following:

```
% mpic++ -std=c++11 hello.cpp
```

which will create an executable a.out. Once you have hello.cpp compiled (into a.out or some other executable), run it!

```
% mpirun -np 4 ./a.out
```

Experiment with different numbers of processes – though you probably don't want to launch more than ten or twelve on docker – since these are real processes that are being launched.

2.2 MPI Ping Pong

The point to point example we showed in class used `MPI::Comm::Send()` and `MPI::Comm::Recv()` to communicate – to bounce a message back and forth. Rather than including all of the code for that example in the text here, I refer you to the included program `pingpong.cpp`. Take a few minutes to read through it and determine how the arguments get passed to it and so forth.

Build an executable program from `pingpong.cpp`. (I suggest using `pingpong.exe` rather than `a.out` as the executable name and the Makefile should already be set up that way.) In this case the program can take a number of "rounds" – how many times to bounce the token back and forth. Note also that the token gets incremented on each "volley". Launch this program with `mpirun -np 2`. Note that we are passing command line information in to this program. Note also that we are not testing the rank of the process for the statement

```
  if (argc >= 2) rounds = std::stol(argv[1]);
```

What are we assuming in that case about how arguments get passed? Is that a valid assumption? How would you modify the program to either "do the right thing" or to recognize an error if that assumption were not actually valid?

Does this program still work if we launch it on more than two processes? Why or why not?

2.3 Iterative Solvers

At the core of most (if not almost all) scientific computing programs is a linear system solver. That is, to solve a system of PDEs, we need to discretize the problem in space to get a system of ODEs. The system of ODEs is discretized in time to get a sequence of nonlinear algebraic problems. The nonlinear problems are solved iteratively, using a linear approximation at each iteration. It is the linear problem that we can solve computationally – and there is an enormous literature on how to do that.

A linear system is canonically posed as

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b},$$

where our task is to find an $\boldsymbol{x}$ that makes the equation true. If $\boldsymbol{A}$ has full rank, the solution exists and is unique. The classical method for solving a linear system is to use Gaussian elimination (equivalently, LU factorization). Unfortunately, LU factorization is an $O(N^3)$ computation, with a structure similar to that of matrix-matrix product (in fact, LU factorization can be reduced to a series of matrix-matrix products, and high-performance implementations of matmat are used to make high-performance versions of linear system solvers).

One motivation that we discussed for using sparse matrix representations (and sparse matrix computations) is that many linear systems – notably those derived from PDEs using the pattern above – only involve a few variables in every equation. It is generally much more efficient in terms of computation and memory to solve linear systems iteratively using sparse matrix methods rather than using LU factorization. The most commonly used iterative methods (Krylov subspace methods) have as their core operation a sparse matrix-vector product rather than a dense matrix-matrix product. Besides being more efficient, sparse matrix-vector product is much more straightforward to parallelize than dense matrix-matrix product.

A prototypical problem for linear system solvers is Laplace's equation: $\nabla^2\phi = 0$. As illustrated in lecture, one interpretation of the discretized Laplace's equation is that the solution satisfies the property that the value the solution `phi(i,j)` at each grid point is equal to the average of its neighbors. We made two, seemingly distinct, observations based on that property.

First, we noted that we could express that property as a system of linear equations. If we map the 2D discretization `phi(i,j)` to a 1D vector `x(k)` such that

```
phi(i,j) == x(i*N + j)
```

(where `N` is the number of discrete points in one dimension of the discretized domain) then we can express the discretized Laplace's equation in the form $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$. In that case, each row of the matrix represents a linear equation of the form:

$$x_i - (x_{i-1} + x_{i+1} + x_{i-N} + x_{i+N})/4 = 0. \tag{1}$$

Second, we remarked that one could solve the discretized Laplace's equation iteratively such that at each iteration k we compute the updated approximate value of each grid point to be the average of that grid point's neighbors:

$$\phi_{i,j}^{k+1} = (\phi_{i-1,j}^{k} + \phi_{i+1,j}^{k} + \phi_{i,j-1}^{k} + \phi_{i,j+1}^{k})/4. \tag{2}$$

Note that once $\phi_{i,j}^{k+1} = \phi_{i,j}^{k}$ for all i and j, then we also have

$$\phi_{i,j}^{k} - (\phi_{i-1,j}^{k} + \phi_{i+1,j}^{k} + \phi_{i,j-1}^{k} + \phi_{i,j+1}^{k})/4 = 0, \tag{3}$$

which means ϕ^k in that case solves the discretized Laplace's equation.

Now, given the mapping between x and ϕ, equations (1) and (3) are saying the same thing. Or, in other words, iterating through the grid and updating values of the grid variables according to a defined stencil is equivalent to the product of matrix by a vector. In the former case, we are representing our unknowns on a 2D grid – in the latter case we are representing them in a 1D vector. But, given the direct mapping between the 2D and 1D representations, we have the same values on the grid as in the vector (and we know how to get from one to the other). And, most importantly, we know that we can compute the result of the matrix by vector product by iterating through the equivalent grid and applying a stencil.

That turns out to be a profound realization for two reasons. First, we don't have to create a 1D vector to represent knowns or unknowns in our problem. Our problem is described on 2D grids, we can keep it on 2D grids. Second, and here is the really significant part, *we don't need to form the matrix A to compute*

$A \times x$. Read that last sentence again. The consequence is perhaps even more profound. *We can solve* $Ax = b$ *without ever forming* A. (Since $\boldsymbol{A}$ was a construct to represent the stencil application, perhaps this isn't so surprising.)

So how do we do that? Again, with the class of Krylov solvers, we only need to form the product of $\boldsymbol{A}$ times $\boldsymbol{x}$ as a core operation to solve $\boldsymbol{Ax} = \boldsymbol{b}$. In the context of the solver, we don't need the matrix $\boldsymbol{A}$, we just need the result of multiplying $\boldsymbol{A}$ times $\boldsymbol{x}$ – which we can do with a stencil.

2.4 Writing Jacobi as an Iteration with Matrix-Vector Product

In lecture we noted that we could express the "in-place" iterative update of the grid variables this way:

```
void jacobi(Grid& X0, Grid& X1, size_t max_iters) {
  for (size_t iter = 0; iter < max_iters; ++iter) {
    for (size_t i = 1; i < X0.numX() - 1; ++i) {
      for (size_t j = 1; j < X0.numY() - 1; ++j) {
        X1(i, j) = (X0(i - 1, j) + X0(i + 1, j) + X0(i, j - 1) + X0(i, j + 1)) / 4.0;
      }
    }
    swap(X0, X1);
  }
}
```

(Note that this is simplified from the source code to save space in this document.)

An equivalent expression would be the following:

```
void jacobi(Grid& X0, size_t max_iters) {
  Grid R(X0);
  for (size_t iter = 0; iter < max_iters; ++iter) {
    for (size_t i = 1; i < X0.numX() - 1; ++i) {
      for (size_t j = 1; j < X0.numY() - 1; ++j) {
        R(i, j) = X0(i, j) - (X0(i - 1, j) + X0(i + 1, j) + X0(i, j - 1) + X0(i, j + 1)) / 4.0;
      }
    }
  }
  X0 = X0 - R;
}
```

But notice what the two inner loops are: we are computing `R` according to equation (3). That is, $R = A \times X0$!

Given that we know how to write matrix-vector product, let's refactor and rewrite `jacobi` in a slightly more reusable and modular fashion. First, let's define an operator between a stencil and a grid that applies the Laplacian.

```
Grid operator*(const Stencil& A, const Grid& x) {
  Grid y(x);

  for (size_t i = 1; i < x.numX()-1; ++i) {
    for (size_t j = 1; j < x.numY()-1; ++j) {
      y(i, j) = x(i,j) - (x(i-1, j) + x(i+1, j) + x(i, j-1) + x(i, j+1))/4.0;
    }
  }
  return y;
}
```

In general, we might want to carry this iteration with the stencil, and just dispatch to it from here (and use subtype or parametric polymorphism), but we are going to just use the Laplacian stencil here.

Now we can define a simple iterative solver ("ir" for iterative refinement):

```
size_t ir(const Stencil& A, Grid& x, const Grid& b, size_t max_iter) {
  for (size_t iter = 0; iter < max_iter; ++iter) {
    Grid r = b - A*x;
    x += r;
  }
  return max_iter;
}
```

Question for the reader. What interface do we need the stencil to provide in this case? How would we define it? (In the above, A is the stencil and x and b are Grids. What functions are used between A and x and/or b? Answer this question for yourself *before* you look into `Stencil.hpp`.

Second question for the reader. What functions do you need to define so that you can compile and run the code just as it is above – meaning, using matrix and vector notation? Again, answer this to yourself before looking in `Grid.cpp`.

(NB: Using the ir routine above is only equivalent to jacobi when we have unit diagonals in the matrix, otherwise we would need the additional step of doing that division (the general practice of which is known as *preconditioning*). But we've defined things here so that that they will work without needing to worry about preconditioning for now – even though preconditioning turns out to be a an essential issue in real solvers.)

2.5 Specialization, 5-point stencils and 9-point stencils

In C++, templates can be parameterized by type, but they can also be parameterized by integral values. For example, one could define a structure with compile time specified internal storage as follows:

```
template <int I>
struct n_array {
  char storage_[I];
}
```

Here, `n_array` is a class (struct) template, with an integer as its parameter. We can instantiate an `n_array` as follows:

```
int main() {

  n_array<10> ten;
  n_array<42> forty_two;

  std::cout << "size of ten is " << sizeof(ten) << std::endl;
  std::cout << "size of forty_two is " << sizeof(forty_two) << std::endl;

  return 0;
}
```

Compiling and executing this program will produce:

```
size of ten is 10
size of forty_two is 42
```

Functions templates can also use integral types as template parameters. Coupled with overloading, integral template values (which can be enumerated types in addition to numeric types) can provide powerful compile-time dispatching capabilities.

The stencil we have been considering in lecture for representing a PDE discretization has five points: the middle point and its four neighbors to the north, east, south, and west. But the Laplacian can also be discretized in other ways. For example, we can use a nine-point stencil, where we include neighbors to the northeast, southeast, southwest, and northwest in addition to the original four. In this case we do not take a strict average. Rather, we use the following averaging formula to compute the middle point:

$$\phi_{i,j}^{k+1} = \left(4(\phi_{i-1,j}^k + \phi_{i+1,j}^k + \phi_{i,j-1}^k + \phi_{i,j+1}^k) + (\phi_{i-1,j-1}^k + \phi_{i+1,j-1}^k + \phi_{i-1,j+1}^k + \phi_{i+1,j+1}^k)\right)/20. \quad (4)$$

Let's change the `Stencil` class and `operator*` function above to be a class and function template, respectively:

```
template <int I>
class Stencil { };

template <int I>
Grid operator*(const Stencil<I>& A, const Grid& x);
```

Now, we can create an `operator*` function for a five-point stencil and for a nine-point stencil. We use the integer template parameter to differentiate between the two. Although any two different values will serve to differentiate the instantiations, we'll go ahead and use 5 and 9 since those are the stencil sizes. To do this, we put the template declarations as above in Stencil.hpp. We can then define the following five point stencil as follows in Stencil.cpp:

```
template <>
Grid operator*(const Stencil<5>& A, const Grid& x) {
  Grid y(x);

  for (size_t i = 1; i < x.numX() - 1; ++i) {
    for (size_t j = 1; j < x.numY() - 1; ++j) {
      y(i, j) = x(i, j) - (x(i - 1, j) + x(i + 1, j) + x(i, j - 1) + x(i, j + 1)) / 4.0;
    }
  }
  return y;
}
```

The notation `template <>` indicates that the function template `operator*` is fully specialized. In this case, we instantiate it with 5 bound to its template parameter (notice the specialization of `Stencil` in the parameter list). We can similarly define another specialization, say for a nine-point stencil.

Finally, we parameterize the `ir` function to be a function template as well:

```
template <int I>
size_t ir(const Stencil<I>& A, Grid& x, const Grid& b, size_t max_iter, double tol)
```

Now, to invoke `ir` with a five-point stencil we would make the following call:

```
  Stencil<5> A;
  ir(A, X1, X0, max_iters, 1.e-6);
```

whereas to invoke it with a nine-point stencil we would instead do this:

```
  Stencil<9> A;
  ir(A, X1, X0, max_iters, 1.e-6);
```

NB: there must be a specialization for `operator*` for this to compile. Note that although there is an advanced use of templates in implementing this example, its use only involves specifying 5 or 9 to the `Stencil`.

The code in support of `Grid` is in `Grid.hpp` and `Grid.cpp`, the code for `Stencil` is in `Stencil.hpp` and `Stencil.cpp`, and the code for `ir` is in `ir.hpp` and `ir.cpp`. A driver program is provided in `ir_driver.cpp`. These files are in their subdirectory named "laplace".

3 Exercises

3.1 Ring

In the MPI ping-pong example we sent a token back and forth between two processes, which, while actually quite amazing, is still only limited to two processes.

Write a program `ring.cpp` (perhaps starting with and modifying `pingpong.cpp`) that instead of simply sending a token from process 0 to 1 and then back again, sends a token from process 0 to 1, then from 1 to 2, then from 2 to 3 – etc until it comes back to 0. As with the ping pong example above, increment the value of the token every time it is passed.

Hint: The strategy is that you want to receive from `myrank-1` and send to `myrank+1`. It is important to note that this will break down if `myrank` is zero or if it is `mysize-1`. These cases could be addressed explicitly. (Or, since with everything else in MPI, there may be capabilities in the library for handling this situation, since it is not uncommon.)

Deliverables

- A file `ring.cpp` that implements the aforementioned program.
- A `Makefile` that correctly compiles and creates the program `ring.exe` in response to `make ring.exe`.

3.2 mpiTwoNorm

We have had a running example of taking the Euclidean norm of a `Vector` throughout this course. It seems only fitting that we now develop an MPI version. Computing the Euclidean norm has two parts, and the difficult part isn't the one you would expect.

The mpiTwoNorm Function

The statement for this part of the problem is: Write a function `mpiTwoNorm` that takes a `Vector` and returns its Euclidean norm.

That is easy to state, but we have to think for a moment about what it means in the context of CSP. Again, this function is going to run on multiple separate processes. So, first of all, there is not really such a thing as "a Vector" or "its norm". Rather, each process will have its own Vector and each process can compute the norm of that Vector (or, it can compute the sum of squares of that Vector). Together, the different Vectors on each process can be considered to be one distributed Vector.

But now what do we mean by "its norm"? If we consider the local Vectors on each process to be part of a larger global Vector, then what we have to do is compute the local sum of squares, add all of those up, and then take the square root of the resulting global sum. This immediately raises another question though. We have some number of separate processes running. How (and where) are the individual sums "added up"? Where (and how) is the square root taken? Which process gets (or which processes get) the result?

Typically, we write individual MPI programs working with local data as if they were a single program working on the global data. That can be useful, but it is paramount to keep in mind what is actually happening.

With this in mind, generally, the local vectors are parts of a larger vector and we do a global reduction of the sum of squares and then either broadcast out the sum of squares (in which case all of the local processes can take the square root), or we reduce the sum of squares to one process, take the square root, and then broadcast the result out. Since the former can be done efficiently with one operation (all-reduce), that is the typical approach.

The precise statement of this part of the problem is: Write a function `mpiTwoNorm` that is executed on each process in a communicator (assume `MPI::COMM_WORLD`). The function should take a local `Vector` and return the Euclidean norm of a vector that would be the concatenation of all of the local vectors stored on the processes in `MPI::COMM_WORLD`. All of the processes calling the function should return the same result.

The mpiTwoNorm Driver

Now here is the harder question that I alluded to above. In previous assignments you have been asked to test your implementations of Euclidean norm with a driver. A random vector is usually created to compare the sequential result to the parallel result (or to differently ordered sequential results). But how do we generate a distributed random vector across multiple independent processes, each of which will be calling its own version of randomize? More importantly, in order to compare results to those obtained with sequential random vectors, how do we create a distributed random vector, such that the combination of them is exactly the same as a single sequential vector? Consider the following example:

```
int main() {
  MPI::Init();

  size_t myrank = MPI::COMM_WORLD.Get_rank();
  size_t mysize = MPI::COMM_WORLD.Get_size();
  Vector lx(1024);
  randomize(lx);
  double rho = twoNorm(lx);

  std::cout << myrank << ": " << std::sqrt(rho) << std::endl;

  MPI::Finalize();
  return 0;
}
```

Compile this with `mpic++` (per above instructions) and run it. Run this on 8 processes (say). What does it print out?

As opposed to the example above, `mpi2norm_simple.cpp` has a function that returns the norm of all of the combined global vector. Compile and run that program (you can `make mpi2norm_simple.exe`). It should print something like this:

```
# seq Pseq mpi diff
0 18.227 51.5536 51.5536 0
```

The columns are for the process rank, the sequential norm, the MPI computed norm, and the difference between the two. Read through the program and see if you can assess what it basically does. In this case, it creates a random vector on each process computes the sum of squares on each process, uses `MPI::COMM_WORLD::Reduce` to get a global sum, takes the square root and then prints the result on the root process (rank 0).

Next, consider the program `mpi2norm_simple_all.cpp`. Again, read through the program and compare it to the previous one. This program does the same computations as the previous one, but now, each process prints the same information that was previously printed just by rank 0.

There are a few things to notice. First, all of the processes except rank 0 do not have the correct global norm. Second, all of the processes (including rank 0) have exactly the same local norm (because each of the local Vectors is the same). Third, the printed output is unordered and may be somewhat scrambled (and may differ from run to run).

NB: Here is a situation where the SPMD local processes are quite different from the equivalent global process that we want. For taking a parallel 2-norm, we want to compute the norm of a single vector that is spread across multiple processes, with each node contributing its part to the total. However, `randomize` is a completely sequential function. It uses a pseudo-random number generator that starts with some seed number and then generates a deterministic sequence of (unpredictable) numbers. No matter how many copies of it we run, each copy generates the same sequence – and therefore populates the same values in each local vector. That is, each of the vectors on the node are identical. But this is not what we want at all! The local vectors are supposed to be parts of a single longer vector – and, for testing, one that we can replicate sequentially.

We want to address all three of the above problems.

First, let's get a truly distributed random Vector (so that each process has a piece of a larger Vector, not the same copy of a smaller Vector). One approach to dealing with issues such as this (the same kind of problem shows up when, say, reading data from a file) is to get all the data needed on one node (typically node 0) and then scatter it to the other nodes using `MPI::Comm::Scatter`.

Second, we need to get the computed norm to all of the processes rather than just to a single process.

Finally, we need to print the computed norms in an orderly fashion (using the inverse operation, say, of `MPI::Comm::Scatter`.)

Write a driver program `mpi2norm_driver.cpp` that takes as input the local size of a `Vector`. On rank 0 it should create a `Vector` of a global size that is equal to the input size (local size) times the number of ranks in `MPI::COMM_WORLD`. It should then scatter that vector out to all the other processes – into local vectors of the size given. (Scattering takes one line of code – one invocation of `MPI::Comm::Scatter`).

Write a function `mpiTwoNorm` with the following prototype:

```
double mpiTwoNorm(const Vector& x);
```

This should be an SPMD function. That is, every process will call it. And, on every process, it should return the norm of the global vector that is represented by the set of local vectors that each process passes in when it calls `mpiTwoNorm`.

Deliverables

- A file `mpi2norm_driver.cpp` that contains the `mpiTwoNorm` function and a `main` function to drive it, all as described above. The driver should first print a header row as in `mpi2norm_simple.cpp`. It should then print a line of information containing the rank, the sequential norm, the global norm, and the difference between the sequential norm and the global norm. The lines should be printed in order by rank: the first line should be for rank 0, the next for rank 1, and so on. For example:

  ```
  $ mpirun  -np 4 mpi2norm_driver.exe
  # seq mpi diff
  0 37.111 37.111 3.55271e-14
  1 37.111 37.111 3.55271e-14
  2 37.111 37.111 3.55271e-14
  3 37.111 37.111 3.55271e-14
  ```

- A `Makefile` that correctly compiles and creates the program `mpi2norm_driver.exe` in response to `make mpi2norm_driver.exe`.

3.3 Timing mpiTwoNorm (AMATH583 only)

In addition to testing, we also have been evaluating our programs for how they scale with respect to problem size and to number of threads/processes. We will continue this process for this problem. In distributed memory, however, the issue of timing becomes a little touchy, because there is no real notion of synchronized time across the different processes – but we need one number to measure to determine whether we are scaling or not.

Make a copy of your `mpi2norm_driver.cpp` and rename it `mpi2norm_timer.cpp`. Add the necessary code to print out the time required to compute the global two norm on rank 0 as well as the time between the scatter and gather for the parallel two norm.

As you have done before, your program should print a line of tab (or space) separated numbers. The first element in the line should be the number of processes, the second should be the size of the vector, the third the sequential execution time, the fourth the parallel execution time, the fifth the speedup, and finally the difference between the computed norms.

Deliverables

- A file `mpi2norm_timer.cpp` that contains the `mpiTwoNorm` function and a `main` function to time it, all as described above.

- A `Makefile` that correctly compiles and creates the driver program `mpi2norm_timer.exe` in response to `make mpi2norm_timer.exe`.

3.4 Extra Credit

Reminder – the files for the laplace solver are in the subdirectory "laplace", not "src". Compile and run the `jacobi_driver.exe` and `ir_driver.exe` programs. You can build them by issuing "make all" in the laplace subdirectory. The `ir_driver.exe` program should have been provided with a specialization for the five-point stencil. When you run the driver program, the ir solver will print the norm of residual at every iteration. The two programs should print the same sequence of residuals.

For this exercise, implement **`operator*`** for the nine-point stencil. A skeleton has been provided in `Stencil.cpp`. To switch to that version of **`operator*`**, simply change the 5 to a 9 in `ir_driver.cpp` and recompile.

We will be revisiting this problem in the final assignment.

4 Turning in The Exercises

All your cpp files, hpp files, and your `Makefile` should go in the tarball `ps7.tgz`. If necessary, include a text file ref7.txt that includes a list of references (electronic, written, human) if any were used for this assignment.

Before you upload the `ps7.tgz` file, it is **still very important** that you have confidence in your code passing the automated grading scripts. After testing your code with your own drivers, make use of the python script test_ps7.py by using the command `python test_ps7.py` in your ps7 directory. Once you are convinced your code is exhibiting the correct behavior, upload ps7.tgz to Canvas.

5 Learning Outcomes

At the conclusion of week 9 students will be able to

1. Explain the basic parallelization strategy of MPI
2. Describe communicating sequential processes and SPMD models
3. Name and describe the functions in both the collective and the point-to-point versions of "six-function MPI"
4. Describe the semantics of `MPI::Comm::Send` and `MPI::Comm::Recv`
5. Write "ping pong" programs using "six function MPI" that sends and receives an integer, a double, an array of integers, or an array of doubles
6. Describe what "ghost cells" are, where they arise, and what you do with them
7. Describe a scenario where MPI can deadlock
8. Describe three strategies for avoiding such deadlock
9. Describe the principal components of the MPI communicator abstraction
10. Explain `MPI::Comm::Scatter` and `MPI::Comm::Gather`
11. Explain Cannon's algorithm for distributed matrix-matrix product
12. Describe a strategy and implement a program for computing π using numerical quadrature
13. Describe the semantics of `MPI::Comm::Isend` and `MPI::Comm::Irecv`
14. Describe the semantics of `MPI::Comm::Ssend`