UNIVERSITY OF WASHINGTON
Department of Applied Mathematics

AMATH 483/583 **Assigned:** 5/30/2017
Problem Set #Final **Due:** 6/7/2017

1 The Rules and Preliminaries

This is a take-home final exam for AMATH 483/583. The same rules for homework assignments apply for the final – *with the following exception*:

- **No final exams will be accepted after 11:59pm PDT on 6/7**
  We gave a little leeway on this for the midterm. Due to the time constraints of grading the final so the overall course grades can be computed and submitted on time, **there will be NO leeway for the final.** It is recommended to plan on turning in your exam in by **10:00pm PDT** so that you have a couple hours to debug any submission issues. You can submit as many times as you'd like until the Dropbox is closed at 11:59pm PDT.

Note that, unlike the midterm, you are allowed to discuss the problems with your fellow students. This **does not mean** you should show your code to other students. We had a problem with similar/identical code being turned in for the midterm that resulted in heavy consequences. Receiving a zero on the final with potential disciplinary action is not a good way to wrap up the quarter.

1.1 Clusters

Better late than never, we were able to set up three clusters for use in AMATH583 for the next week. The clusters are set up in the Amazon Web Services (AWS) cloud just for this course and will be taken down once the course is completed. (NB: If you are interested in experimenting with parallel computing on your own or for a research project after this course, please let me know directly and I will try to see if we can make some arrangements to keep some of the clusters running a bit longer.)

Using a cluster is a bit different than what we have done so far in this course to use parallel computing resources. A compute cluster typically consists of some number of front-end nodes and then some (much larger) number of compute nodes. Development and compilation are typically done on the front-end nodes. A cluster is a shared system. When you ran parallel programs on your own computer, you didn't have to worry about the fact that someone else might want to run a parallel program with your computer. Or that other programs running on your computer would interfere with the performance of your jobs. To make efficient use of the resources – and to allocate them fairly, once a program is compiled and ready to execute, it is passed off to the compute nodes via a queueing system. The queueing system makes sure that the compute nodes are fairly used and that the resources that you need to run your parallel job are available.

There are three clusters available to you:

1x32 This cluster consists of a head node and one (or two) large SMP nodes in the cluster. The SMP nodes have 36 cores. The intent of this cluster is to allow you to try out previous examples on a real, and large, multicore machine. The IP address that you will use to connect to the head node of this cluster is 34.208.45.5.

32x1 This cluster is the complementary configuration to 1x32. It consists of a head node and 32 single-core compute nodes. The intent of this cluster is to allow you to run distributed memory jobs over a non-trivial number of nodes and observe scaling. The IP address that you will use to connect to the head node of this cluster is 52.38.172.236.

8x4 This cluster consists of a head node and 8 four-core nodes. The intent here is to provide a middle ground between 1x32 and 32x1. This is also the cluster that you should use for testing and debugging of your codes before trying to scale them up in shared or distributed memory. If everyone keeps their submissions to 4 cores/nodes are fewer, we should get sufficient throughput so that everyone can run here without significant delays. The head node of this cluster is 34.209.162.111.

The clusters all reference the same network file system for home directories. Your home directory (and all of its files) are the same, regardless of which cluster head-node you log in to. So you don't have to worry about moving files between the clusters.

If it turns out we need to add more capacity so that everyone can get their jobs run, we can set that up fairly quickly (now). Nevertheless, you are encouraged to start this assignment as early as you can so that there isn't major contention for these resources close to the assignment due date.

1.2 Accessing the Clusters

An account has been made for you on the clusters, with the same name as your UW netID. Authentication on this cluster is not done through UW however (this is possible, but requires significant administrative lead time). Instead, we are going to use the Secure Shell (ssh) to connect to the clusters. Using ssh is much more secure than using passwords – and, in fact, passwords are disable altogether for the clusters. You can *only* connect with ssh. The ssh client-server system fills two roles for us in using the clusters. First, it provides a secure mechanism to authenticate – to establish your identity to the system so that you can login and access the privileges associated with your account. Second, ssh encrypts the transmission between your client and the server you connect to (in this case, the clusters).

Ssh is built on public-private key cryptography. Public-private key cryptography is asymmetric – meaning one key is used to encrypt data and another is used to decrypt it. One of the keys is denoted the "public" key and one the "private" key. The keys are strings of bits (and related to each other) such that the public key can be used to encrypt a message and the private key can be used to decrypt it. The public key can be widely disseminated and anyone can use it to encrypt a message. But, the holder of the private key does not share that – and only that key can be used to decrypt the message.

To distribute your private key I will be using Google Drive. Your invitation to access the document containing your key will be sent by email. You will have to authenticate with Google Drive to get the document. Sometime on the afternoon of May 30, 2017 you will receive an email from me (Andrew Lumsdaine (via Google Drive)) the subject "[netid]_id_rsa" – where [netid] will be your UW netid. When you get that email, you will see an "open" button in it. If you trust that the email is actually from me, then you can click on the "open" button and you will be taken to Google drive where you can download the document. Download it to a secure location on your computer – and remember where you put it because you will need it in the next steps to enable your connection to the clusters.

You may be asked to authenticate in the process of getting the file from Google drive. Again, if you are pretty sure that the email is from me, and pretty sure that you are really at the UW authentication service, go ahead and do that. If you are more cautious about clicking on links in your emails, then you can go to drive.google.com and authenticate there first. Then you may see a link in the Google drive page in the left hand column that says "Shared with me". If you follow that link you should see the private key document. If not, now that you are authenticated directly with the actual Google drive page (probably), you can safely click on the "open" button in your invitation email and it should take you directly to your private key. In general you should be very skeptical of following any links in your emails – and triply skeptical of entering authentication information in the pages that such links take you to.

Actually Connecting

Detailed walkthroughs on connecting to the cluster once you have your private key are available on Panopto. There are two parts to this process: importing your private key into your ssh client and then using your ssh client to connect to one of the hosts above.

Sanity Check

Once you are connected to any of the three head nodes, there are a few quick sanity checks you can do to check out (and learn more about) the cluster environment.

To find out more about the head node itself, try

```
% more /proc/cpuinfo
```

This will return copious information about the cores on the head node. Of interest are how many cores there are.

You should also double check the C++ compiler, the mpic++ compiler, and mpirun.

```
% c++ --version
% mpic++ -cxx=clang++ --version
```

These should both print information about the installed version of clang (3.8). And, finally, issuing

```
% mpirun --version
```

should print out the same message as it did in docker – "HYDRA build details" indicating that it is MPICH.

We will discuss the queueing system in more detail below, but to get some information about it, you can issue the command

```
% qhost
```

This will print a list of the compute nodes along with some basic details about their capabilities. On the 8x4 cluster, qhost prints the following:

```
HOSTNAME                ARCH         NCPU NSOC NCOR NTHR  LOAD  MEMTOT  MEMUSE  SWAPTO  SWAPUS
----------------------------------------------------------------------------------------------
global                  -               -    -    -    -     -       -       -       -       -
ip-10-11-16-167         lx-amd64        4    1    2    4  0.00    7.3G  440.1M     0.0     0.0
ip-10-11-16-168         lx-amd64        4    1    2    4  0.00    7.3G  437.9M     0.0     0.0
ip-10-11-16-170         lx-amd64        4    1    2    4  0.00    7.3G  444.1M     0.0     0.0
ip-10-11-16-225         lx-amd64        4    1    2    4  0.00    7.3G  439.5M     0.0     0.0
ip-10-11-16-226         lx-amd64        4    1    2    4  0.00    7.3G  440.0M     0.0     0.0
ip-10-11-16-30          lx-amd64        4    1    2    4  0.00    7.3G  442.0M     0.0     0.0
ip-10-11-16-48          lx-amd64        4    1    2    4  0.00    7.3G  438.1M     0.0     0.0
ip-10-11-16-79          lx-amd64        4    1    2    4  0.00    7.3G  442.3M     0.0     0.0
```

("man qhost" to access documentation to explain the different columns.)

The command "qstat" on the other hand will provide information about what jobs are currently running (or waiting to be run) on the nodes.

```
% qstat
% qstat -f
```

Getting Your Files onto the Cluster

There is good news and bad news about working on the AWS clusters. The good news is that the filesystem with your home directory is shared among the three clusters so everything and anything you do when logged into one cluster will be there when you log into any of the other clusters. The bad news is that AWS (deliberately, apparently) is not easily (as in, at all) shareable with the outside world. You won't be able to share the files and folders on your laptop with the clusters in the same way that you did with docker.

(Intrepid users may wish to experiment with sshfs to access the remote filesystem as a folder on your laptop. However, we won't be able to provide any support for this.)

You will need to explicitly transfer files (e.g., using scp or rsync) between your laptop and the cluster. I recommend that you do editing of the files directly on the cluster rather than trying to edit them on your laptop and then uploading them to the cluster. It will be very difficult to maintain consistency between your laptop copies and your cluster copies unless you use a version control system or a simpler tool like rsync. Again, the cluster itself has a shared filesytem, so the same files will be available to you from any of the head nodes (and from the compute nodes). Documentation on scp and/or rsync is available in a number of on line sources (as well as with "man" on docker and on the clusters).

Important: A number of files have been installed and made available for you on the clusters in the directory /nfs/home/trove/amath583. If you want to copy from them or use them directly, that is what they are there for. You will not need to write very much code beyond what is there, so I recommend that you do all of your editing directly on the cluster. You may want to make copies of your work from time to time and download those somewhere safe, just in case.

1.3 Compiling and Running an MPI Program

The development environment you will be using on the cluster is identical to the one you have been using in docker (except for the file sharing issue we just mentioned). The OS is ubuntu 16.04, the compiler is clang, the MPI is mpich. The mpi programs you built under docker should run (you may not even need to recompile) without modification. There is a difference in how you run programs however. In our previous assignment, we used mpirun to launch an MPI job. In our queue-based cluster environment, we still use mpirun to launch the actual parallel job – but we use the queueing system to launch mpirun.

1.4 qsub

There are a number of HPC queueing systems used in production clusters, all of which provide essentially the same functionality. The default for the AWS cluster setups is the Sun Grid Engine (SGE) by Oracle. Other popular queueing systems include slurm and torque. The basic paradigm in using a queuing system is that you request the queueing system to run something for you. The thing that is run can be a program, or it can be a command script (with special commands / comments to control the queueing system itself). Command line options (or commands / directives in the script file) are used to specify the configuration of the environment for the job to run, as well as to tell the queueing system some things about the job (how much time it will need to run for instance). When the job has run, the queueing system will also provide its output in some files for you – one for the output from cout and the other for the output from cerr.

A reasonable tutorial on the basics of using SGE can be found here: http://bit.ly/2qCw03h. The most important commands we have already seen above and in the walkthroughs: qsub, qstat, and qhost. The most important bits to pay attention to in that tutorial are about those commands. The discussion about SGE and Open MPI basically applies to MPICH as well.

Pseudo-Interactive Submissions

When you are debugging and testing *small* and short running jobs, you may want to get your results back on your screen rather than in a file. To do this you can use the qrsh command instead of qsub. Please do not use qrsh to actually log in to nodes on the cluster as you will then be blocking others from using those nodes and we really need to use those only under direction of the queueing system so that we can make most efficient use of them.

1.5 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. 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 (e.g., Lecture 17), 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 $\boldsymbol{Ax} = \boldsymbol{b}$. 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 solve $Ax = b$*. Read that last sentence again. We can solve $Ax = b$ without every forming A.

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.

1.6 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:

```
int jacobi(Grid& Phi0, size_t max_iters) {
  Grid Phi1(Phi0);
  for (size_t iter = 0; iter < max_iters; ++iter) {
    for (size_t i = 1; i < Phi0.numX()-1; ++i) {
      for (size_t j = 1; j < Phi.numY()-1; ++j) {
        Phi1(i, j) = (Phi0(i-1, j) + Phi0(i+1, j) + Phi0(i, j-1) + Phi0(i, j+1))/4.0;
      }
    }
    swap(Phi0, Phi1);
  }
  return max_iters;
}
```

An equivalent expression would be the following:

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

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

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 do we need stencil to provide in this case? How would we define it? (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 very important issue in real solvers.)

1.7 A note on encryption for those interested:

Since the bit sequences in the private and public key are obviously related to each other, one might ask whether the private key can be derived somehow from the public key. One could imagine, for instance,

encrypting many known messages with the public key and comparing them to the original message in order to reverse-engineer the private key. Fortunately, as far as we know, this is not possible. The difficulty of doing so is exponential in the number of bits in the keys. With a sufficient number of bits, the task becomes one that could not be completed until many lives of the current universe have come and gone.[1]

Since public-private key encryption is relatively expensive compared to symmetric schemes, ssh does not necessarily use your keys for the actual encryption of the channel, but rather to securely exchange symmetric keys between client and server. The symmetric key is then used for encryption and decryption.

The ssh program itself works by keeping your private key securely on your client computer and then putting your public key onto any computer you would like to log into. The exact mechanisms vary depending on the ssh client and server as well as on the OS being used. In the Linux server that you will be connecting to, your public key is stored in a file named `authorized_keys` that is kept in the `.ssh` subdirectory in your home directory. Once you obtain your private key from me and import it into the ssh client of your choice, you will be able to connect directly to the clusters.

Transmitting a Private Key

So now there is a slight problem. I have generated your public-private key pair and the public key does not have to be kept securely. But, your private key does. Private keys always need to be distributed in some kind of secure fashion. Email won't cut it (email is the worst possible way to send anything sensitive). One can leverage existing secure communication mechanisms – but then that needs to be set up in some kind of secure fashion so that the information is only available to the intended recipient. How secure keys are distributed will vary from situation to situation. If you generate secure key pairs on the AWS web site, Amazon will keep your public key, but your private key will be downloaded to your computer over a secure https connection. In settings where one agent creates keys for another, getting the keys to the recipient may be done with encrypted USB drives – but, again, there is some security/encryption involved to make that transfer. And that security also requires establishing identity of the recipient.

Fortunately there are some existing mechanisms to leverage at UW, in particular your UW netid. Your UW netid and associated password are used in a number of services at UW – email, web services, network drives, shared folders, and so forth. When a service is requested for a particular netid and is presented with the correct password, then access is granted. Doing this in a consistent and coherent way across an entire enterprise can be a challenge. Most large (or even small) organizations use some kind of central authentication service. Rather than having each service on campus have its own authentication mechanisms, they all – securely – check credentials with a central server (the central authentication service (CAS)). This is one of those things that "just works" – and you can use your same netid and login at websites in different departments, for your Husky card, and even for some affiliated external services such as Google drive. This is rather remarkable – you never created an account on any of the distinct and separated services that you use around campus – but yet you can use them with the same netid and password.[2]

2 Warm Up

2.1 Investigating the Cluster Queues

In the Panopto walkthrough I demonstrated a few quick examples of launching some illustrative programs on the clusters. One program that is interesting to run across the cluster is the hostname command because it will do something different on different nodes (namely, print out the unique hostname of that machine). This can be very useful for quickly diagnosing issues with a cluster – and for just getting a feeling for how the queuing system works.

As an example, try the following:

```
% qsub -V -b y -cwd hostname
```

[1] This does pre-suppose $P \neq NP$, which has not been proven or disproven.

[2] There is an interesting existential question here. Are *you* authenticating when you enter your netid and your password? Is that transaction proving to the system that you are who you say you are? Or is your identity illusory? Is the system just responding to presentation of credentials without regard to the "identity" of the presenter?

This will run the hostname command on one of the cluster nodes and return the cout and cerr results in hostname.ox and hostname.ex (where the "x" is some number). To run the same thing on different nodes in the cluster, try the following:

```
% qsub -V -b y -cwd -pe mpi 8 mpirun hostname
```

The "-pe mpi 8" argument indicates that qsub should run the indicated program – in this case, mpirun, on 8 nodes and should prepare the environment for MPI. The results from this will be in files mpirun.ox and mpirun.ex (again, the "x" indicating your job's number in the queue).

Try this same command on each of three different clusters. What hostnames get printed out? Again, the 32x1 cluster has 32 unique single-core nodes, so you should get 8 different names. The 8x4 cluster has 8 unique four-core nodes, so you should get four of one name and four of another. Finally, the 1x32 cluster is just a single 36 node machine – you should get 8 of the same name.

Since qsub basically just takes what you give it and runs that it is usually much more productive in the long run to specify what you want to do inside of a script. For example, with the above, you could implement it as follows:

```
#!/bin/bash
echo "Hello from SGE"
mpirun hostname
echo "Goodbye"
```

Put this in a script file, say hello.sh and launch it as:

```
% qsub -V -cwd -pe mpi 8 hello.sh
```

NB: If you run mpirun directly on the front-end node – it will run – but will launch all of your jobs on the front-end node. Where everyone else will be working. This is not a way to win friends.

2.2 Hello MPI World, MPI Ping Pong, and MPI Ring

In the previous assignment, you wrote and/or ran a few introductory MPI programs. Before jumping into the main part of this assignment, you should copy your files to the cluster and try them out with different numbers of nodes and on the different clusters. There are also versions of these programs (along with some examples of the pi program) in the /nfs/home/trove account under the mpi subdirectory.

Experiment with these programs using qsub directly and also with a script. How does the behavior compare with docker?

3 Exercises

3.1 Timing mpiTwoNorm (reprise)

Copy `/nfs/home/trove/amath583/mpi2norm_timer.cpp` to your home directory. You should run this on each of the clusters using the `-pe mpi 8` for various vector sizes and observe what kinds of speedup you get. Copy and answer each of the following questions in a file named `ex1.txt`:

1. On the **1x32** cluster: How do you expect the 8 MPI processes to be distributed across the cluster (how many processes per node)?
2. On the **1x32** cluster: At what vector length did you start seeing speedup? What were the sequential and parallel runtimes for this vector length?
3. On the **32x1** cluster: How do you expect the 8 MPI processes to be distributed across the cluster (how many processes per node)?
4. On the **32x1** cluster: At what vector length did you start seeing speedup? What were the sequential and parallel runtimes for this vector length?

5. On the **8x4** cluster: How do you expect the 8 MPI processes to be distributed across the cluster (how many processes per node)?

6. On the **8x4** cluster: At what vector length did you start seeing speedup? What were the sequential and parallel runtimes for this vector length?

7. Explain the differences, if any, of vector lengths and runtimes between the clusters. Your answer should show your knowledge of distributed parallel computing (MPI) given the structure of each cluster.

Deliverable(s)

- A text file named `ex1.txt` with the above questions and answers to them.

Extra Credit Create a single plot showing strong scaling and weak scaling execution time from one to 32 cores. For strong scaling you will need to run a series of tests on different numbers of cores such that the global problem size stays the same. So, as you add more nodes, you need to decrease the local problem size (passed in as the command line argument). You should use a global problem size that takes at least 3 seconds on one node. For the weak scaling, you should use a local problem size that takes about 200 mseconds on one node. Name the file `mpiTwoNormMPI.pdf` and include it with your other materials for this problem.

Deliverable(s)

- A PDF file named `mpiTwoNormMPI.pdf` a plot showing both strong and weak scaling executing times.

More Extra Credit For up to two of threads/tasks/OpenMP-threads, repeat the corresponding twoNorm problem in the associated earlier problem set. Create a scaling plot in the same manner as prescribed for the MPI portion just above.

Deliverable(s)

- A PDF file named `mpiTwoNormOpenMP.pdf` a plot showing both strong and weak scaling executing times.

You are not being graded on your python or plotting skills in this course so you are free to use whatever resources (including classmates) to create these graphs. In fact, I encourage a nice group solution for this.

3.2 Iterative Refinement / Jacobi Iteration

In the trove you will find files `ir.cpp` and `ir.hpp` that implement iterative refinement as described above. Note that I have also provided some supporting files for the Grid class (`Grid.hpp` and `Grid.cpp`) so that ir (and cg, below) are implemented using operator notation. The operations for the Stencil class are in `Stencil.hpp` and `Stencil.cpp`. *Your assignment is to parallelize – with MPI – the iterative program carried out in the file `/nfs/trove/finalir-seq.cpp`.* There are a few distinct files you need to create to implement this parallelization:

Requirements

1. `mpiStencil.cpp`: an implementation of the `operator*` function declared in `mpiStencil.hpp` provided in the trove. Since we are parallelizing with SPMD (as we showed in class), the stencil part is the same as for the sequential case. BUT – the boundaries need to be updated appropriately before applying the stencil operation.

2. `Final.cpp`: an implementation of the ir(...) function declared in the header file `Final.hpp` provided in the trove. It is an overload of the ir(...) function in `ir.cpp` – it just takes an mpiStencil rather than a plain Stencil. In looking a the different functions called in the sequential version of ir(...) – what needs to change here to make it a parallel ir(...)? We have discussed everything you need for this in lecture. You may want to change the names of some functions that are called here but you probably do not need any MPI code in there directly. There is a Big Hint in the functions that are in the supplied header files.

3. `Final.cpp`: an implementation of the mpiDot(...) function declared in the header file `Final.hpp` provided in the trove. Think very carefully about this one. In this operation the distinction between real boundaries in your problem and the "pseudo-boundaries" realized by the ghost cells does become important.

4. `ir-mpi.cpp`: a driver program that sets up the stencils to operate in SPMD fashion just as was shown in lecture. You may wish to copy from `ir-seq.cpp` as a starting point (available in the trove). The command line arguments will give the global problem size, so you need to divide up the problem accordingly, based on how many ranks there are. Assume, as always, that things divide evenly. Instead of calling ir(...) with a Stencil, you will call ir(...) with an mpiStencil. You should change / edit the include files as needed.

Deliverable(s)

- A source code file named `mpiStencil.cpp` that satisfies the requirements above.
- A `Makefile` that creates an object file named `mpiStencil.o` in response to `make mpiStencil.o`
- A source code file named `Final.cpp` that satisfies the requirements above.
- A `Makefile` that creates an object file named `Final.o` in response to `make Final.o`
- A source code file named `ir-mpi.cpp` that satisfies the requirements above.
- A `Makefile` that creates an object file named `ir-mpi.o` in response to `make ir-mpi.o`
- A `Makefile` that creates an executable file named `ir-mpi` in response to `make ir-mpi`
- All additional source files included in any of the above files.

3.3 The Conjugate Gradient Algorithm (AMATH583 Only)

The conjugate gradient algorithm is one of the most celebrated and important algorithms in computer science / applied mathematics. It uses the same core operation as ir – a matrix-vector product. But through an extremely clever orthogonalization scheme, the CG algorithm is able to converge much more rapidly than ir. *Your assignment is to parallelize – with MPI – the iterative program carried out in the file cg-seq.cpp.* The pieces of this problem are identical to ir above. If you have carried out the above functions carefully enough, they should just drop in:

Requirements

1. `Final.cpp`: an implementation of the cg(...) function declared in the header file `Final.hpp` provided in the trove.

2. `cg-mpi.cpp`: a driver program similar to `cg-mpi.cpp`, only it calls cg(...) instead of ir(...).

Deliverable(s)

- A source code file named `mpiStencil.cpp` that satisfies the requirements from 3.2.
- A `Makefile` that creates an object file named `mpiStencil.o` in response to `make mpiStencil.o`
- A source code file named `Final.cpp` that satisfies the requirements above.
- A `Makefile` that creates an object file named `Final.o` in response to `make Final.o`
- A source code file named `cg-mpi.cpp` that satisfies the requirements above.
- A `Makefile` that creates an object file named `cg-mpi.o` in response to `make cg-mpi.o`
- A `Makefile` that creates an executable file named `cg-mpi` in response to `make cg-mpi`
- All additional source files included in any of the above files.

4 Turning in The Exercises

You are to turn in a tarball `final.tgz`, containing all the deliverables described above, to the Collect It Dropbox. The tarball needs to

1. be a tarball (i.e. use the `tar -czf` command, not zip or other commands)
2. contain only files and not any directories

Once created on your AWS directory, you can pull it down to your machine using the `scp` command (or other file-transfer methods you are familiar with). From there, you will upload the tarball to the Collect It Dropbox.

5 Learning Outcomes

At the conclusion of week 10 students will be able to

1. Enjoy the summer
2. Be well
3. Do good work
4. Keep in touch