**UNIVERSITY OF WASHINGTON**
**Department of Applied Mathematics**

**AMATH 483/583** **Assigned:** 06/02/2018
**Problem Set #Final** **Due:** 06/08/2018

## 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*:

- **The exam may not be discussed with anyone except course instructors.**
  Do not discuss it with your classmates, with your friends, with your enemies. You *may* refer to your previous assignments, AMATH 483/583 slides and lectures. You may ask questions publicly on Piazza to clarify content of the final. The same policies and penalties for plagiarism apply for the final as for the regular assignments.
  We have had problems in the past 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.
- **No final exams will be accepted after 11:59pm PDT on 6/8**
  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 like until Canvas is closed at 11:59pm PDT.

Also note that the final will be graded **BY HAND**. So please add comments properly in the code if you are not confident with whether it will work correctly. This can help us to give you partial credit for a problem.

### 1.1 Compute cluster

The compute cluster for this assignment is 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 somewhat 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, whereas execution of the job is done on the compute 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.

Our AWS cluster consists of one head node and an elastic number of 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 **`54.71.183.17`.**

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. In fact, the fleet of compute nodes will grow and shrink dynamically in response to demand. 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 Cluster

A login has been created for you on the cluster, 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. Using `ssh` is much more secure than using passwords – and, in fact, passwords are disable altogether for the cluster. You can *only* connect to the cluster with `ssh`. The `ssh` client-server system fills two roles for us. 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.

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 during the morning of June 2, 2018 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.

The compute cluster is a shared compute resource, which sharing includes the compute resources themselves, as well as the file system. When you are logged in to the head node, other students will also be logged in at the same time, so be considerate of your usage. In particular,

- Use the cluster only for the final exam,
- Do not run anything compute-intensive on the head node,
- Do not run anything using `mpirun` on the head node,
- Do not put any really large files on the cluster, and
- Do not attempt to look at other students' work. Basic permissions have been established so that you should not be able to view the contents of any home directory but your own. You are responsible for maintaining (basic) control over your work, so don't change the permissions of your actual home directory (`/hfs/home/[yourid]` and don't copy your files anywhere that is not protected (such ast to `/tmp`).

#### Actually Connecting

Detailed walkthroughs on connecting to the cluster once you have your private key will also be made available on Panopto during the morning of June 2, 2018. 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.

Both of these parts will be covered. Targeted videos will be made to cover the process on Windows vs Mac OS, but you may want to watch all of the videos.

## Getting Your Files onto the Cluster

AWS is not easily 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 between your laptop and docker.

One way of getting files back and forth from your laptop to the cluster is to explicitly transfer them, e.g., using the `scp` or `rsync` command. Another alternative is to keep your files once to the cluster and edit them directly there, using a text-based editor such as `nano`, `vim`, or `emacs`. We recommend the `nano` editor if you have not used a text-based editor before (or if you are not familiar with `vim` or `emacs`).[1] To edit, e.g;, a file named `amath583.cpp`, you simply type `nano amath583.cpp` to open and edit it. A third alternative is to use the "remote fs" extension in Visual Studio Code, which will let you edit your files on the cluster via Visual Studio Code running on your laptop.

Of these options, it is recommended (although not necessary) that you keep your files on the cluster and edit them there (either with a text-based editor or with remote fs with Visual Studio Code.) Editing on your laptop and then copying to the cluster will cause numerous difficulties in maintaining consistency between your laptop files and your cluster files unless you use a version control system or a simpler tool like `rsync`. 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).

These different ways of editing and deploying your files on the cluster will be described in the videos.

**Important:** A number of files have been installed and made available for you on the clusters in the directory `/nfs/home/amath583`:

- `/nfs/home/amath583/trove` has files you are probably fairly familiar with by this point in the course and which you may use for reference in this assignment.
- `/nfs/home/amath583/src` has the files you will be working with and will need to complete for this assignment.

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, again, 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.

For your own use of the files, and for structuring your code for this assignment, I also recommend that you use a similar structure to that in `/nfs/home/amath583`:. Namely, you should have your own `trove` and `src` subdirectories "next to" each other. The `Makefile` that is provided in these subdirectories uses a somewhat advanced feature of `make` called "VPATH", which lets you keep your files in different subdirectories, but allows make to treat them just as if they were in the same subdirectory.

## Sanity Check

Once you are connected to the head node, 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++ --version
```

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

[1]Between vim and emacs, the answer is always "emacs".

```
% mpirun --version
```

should print out the same message as it did in docker, indicating that it is Open MPI.

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. Yo may see a list that looks something like this (though it may be longer or shorter):

```
HOSTNAME                ARCH         NCPU NSOC NCOR NTHR  LOAD  MEMTOT  MEMUSE  SWAPTO  SWAPUS
----------------------------------------------------------------------------------------------
global                  -               -    -    -    -     -       -       -       -       -
ip-10-11-12-101         lx-amd64        2    1    1    2  0.00    3.7G  357.8M     0.0     0.0
ip-10-11-12-12          lx-amd64        2    1    1    2  0.00    3.7G  360.6M     0.0     0.0
ip-10-11-12-127         lx-amd64        2    1    1    2  0.00    3.7G  358.0M     0.0     0.0
ip-10-11-12-128         lx-amd64        2    1    1    2  0.00    3.7G  356.2M     0.0     0.0
ip-10-11-12-131         lx-amd64        2    1    1    2  0.00    3.7G  361.6M     0.0     0.0
ip-10-11-12-144         lx-amd64        2    1    1    2  0.00    3.7G  361.8M     0.0     0.0
ip-10-11-12-153         lx-amd64        2    1    1    2  0.00    3.7G  359.2M     0.0     0.0
ip-10-11-12-155         lx-amd64        2    1    1    2  0.00    3.7G  362.8M     0.0     0.0
ip-10-11-12-180         lx-amd64        2    1    1    2  0.00    3.7G  360.2M     0.0     0.0
ip-10-11-12-206         lx-amd64        2    1    1    2  0.00    3.7G  359.8M     0.0     0.0
ip-10-11-12-21          lx-amd64        2    1    1    2  0.00    3.7G  362.0M     0.0     0.0
ip-10-11-12-216         lx-amd64        2    1    1    2  0.00    3.7G  361.8M     0.0     0.0
ip-10-11-12-232         lx-amd64        2    1    1    2  0.00    3.7G  360.9M     0.0     0.0
ip-10-11-12-246         lx-amd64        2    1    1    2  0.00    3.7G  357.7M     0.0     0.0
ip-10-11-12-247         lx-amd64        2    1    1    2  0.00    3.7G  361.4M     0.0     0.0
ip-10-11-12-248         lx-amd64        2    1    1    2  0.00    3.7G  360.2M     0.0     0.0
ip-10-11-12-254         lx-amd64        2    1    1    2  0.00    3.7G  362.2M     0.0     0.0
ip-10-11-12-26          lx-amd64        2    1    1    2  0.00    3.7G  358.4M     0.0     0.0
ip-10-11-12-29          lx-amd64        2    1    1    2  0.01    3.7G  361.9M     0.0     0.0
ip-10-11-12-36          lx-amd64        2    1    1    2  0.00    3.7G  360.0M     0.0     0.0
ip-10-11-12-39          lx-amd64        2    1    1    2  0.00    3.7G  357.7M     0.0     0.0
ip-10-11-12-4           lx-amd64        2    1    1    2  0.00    3.7G  357.9M     0.0     0.0
ip-10-11-12-50          lx-amd64        2    1    1    2  0.00    3.7G  362.4M     0.0     0.0
ip-10-11-12-54          lx-amd64        2    1    1    2  0.00    3.7G  357.7M     0.0     0.0
ip-10-11-12-58          lx-amd64        2    1    1    2  0.00    3.7G  362.0M     0.0     0.0
ip-10-11-12-65          lx-amd64        2    1    1    2  0.00    3.7G  358.1M     0.0     0.0
ip-10-11-12-71          lx-amd64        2    1    1    2  0.00    3.7G  356.1M     0.0     0.0
ip-10-11-12-74          lx-amd64        2    1    1    2  0.00    3.7G  358.2M     0.0     0.0
ip-10-11-12-80          lx-amd64        2    1    1    2  0.00    3.7G  357.8M     0.0     0.0
ip-10-11-12-87          lx-amd64        2    1    1    2  0.00    3.7G  361.3M     0.0     0.0
ip-10-11-12-9           lx-amd64        2    1    1    2  0.00    3.7G  355.9M     0.0     0.0
ip-10-11-12-99          lx-amd64        2    1    1    2  0.00    3.7G  353.7M     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
```

## 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 Open MPI. 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 will also use a 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.

#### 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 A note on encryption for those interested

Since the bit sequences in the private and public key used by ssh are obviously related to each other (one encrypts, one decrypts), 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.[2]

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

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

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.[3]

## 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
```

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 your job's number in the queue). 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 command a few times (experiment with various numbers of nodes – not too many).

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:

```bash
#!/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. You also won't get speed-up since you are using only one node (the front-end node).

[3]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?

### 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.

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 `mpi2norm_timer.cpp` (provided by us) to your home directory. Run this on 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 `final.txt`:

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

Create a single plot showing *strong scaling* and *weak scaling* execution time from using one to 32 MPI ranks. 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 need to choose problem size properly in order to get consistent and reliable timing results. Name the file `mpiTwoNormMPI.pdf` and include it with your other materials for this problem.

**Deliverable(s)**

- A text file named `final.txt` with the above questions and answers to them.
- A PDF file named `mpiTwoNormMPI.pdf` that has a plot showing both strong and weak scaling executing times. You can also include tables that show relevant numbers and any discussion you might have on this.

You are not being graded on your python or plotting skills in this course so you are free to use whatever resources (Matlab, Python, Excel etc.) to create these graphs.

**Extra Credit** Repeat the exercise but for vectors 4 times larger and 4 times smaller than the vector you used above. Put the results (plots, tables, discussion etc.) in the same `mpiTwoNormMPI.pdf` file. Please label and name you plots properly.

**More Extra Credit** In previous assignments, we parallelized our programs for shared-memory, using threads, tasks, and OpenMP to effect the parallelization. Unfortunately, as most of us found, the limitations imposed by docker, small numbers of cores, and laptop hardware limited the amount of speedup that we were able to obtain.

For 48 hours – June 5 and June 6 – I will create another cluster that will have only one compute node – but the compute node will have at least 8 (and hopefully many more) cores. You may repeat up to two previous parallel twoNorm experiments (out of threading, tasking, OpenMP) and run them on this machine. (I am setting it up with batch scheduling so runs by different students don't interfere with each other.) I will publish the IP address for the head node of that cluster when I open it (I am only running it for a few days to save money and so that everyone focuses first on MPI). Your home directory (and files) from the other cluster will be available here as well – jobs will be submitted in the same way, with qsub, though you will not need to use mpirun since you will only be running multi-threaded jobs. Create a scaling plot in the same manner as prescribed for the MPI portion just above. How does running 8 threads compare to running 8 MPI processes on the same problem? (Same question repeated for numbers larger than 8 if we are able to provide a larger multicore machine.)

Put the results (plots, tables, discussion etc.) in the same `mpiTwoNormMPI.pdf` file. Please label and name you plots properly.

## 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/home/amath583/src/ir-seq.cpp`.*

**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`:

   (a) 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.

   (b) 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.

3. `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.exe` in response to `make ir-mpi.exe`
- All additional source files included in any of the above files or required to build the executable.

### 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.exe` in response to `make cg-mpi.exe`
- All additional source files included in any of the above files or required to build the executable.

## 4 Turning in The Exercises

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

1. Be a tarball (i.e. use the `tar -czf` command, not zip or other commands)

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 Canvas. If your program need source code in `src`, `trove` and `test` directories on AWS, you can download all of them to your machine and do `tar -czf final.tgz trove src test`.

## 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