CPSC 424/524 Spring 2018
Assignment 1
Due Date: 2/1/2018, 9:00am

In this assignment, you will access the Grace Linux cluster (**`grace.hpc.yale.edu`**) and run programs to investigate the characteristics of the processors on the cluster. Unless otherwise directed, you may use either C (but not C++) or Fortran for this and other regular programming assignments in the course. For detailed information about the Yale clusters, please consult http://research.computing.yale.edu/hpc-support. (Note: Parts of this website may only be accessible if you are on campus or use a VPN so that your computer has a Yale IP address. You may also need to login to CAS using your NetID and password.) For more information specifically about Grace, please see http://research.computing.yale.edu/grace.

Omega has two cluster login nodes (**`grace1 and grace2`**) and several hundred compute nodes, of which 6 are dedicated for our class to use this semester. In addition, you may use nodes in the "interactive partition" for editing, program building, certain types of executions, etc. The nodes we'll use for most of the semester have two 10-core Intel Xeon processors (Intel Xeon e5-2660 v2) and 128 gigabytes of memory. (When we come to GPU programming, we'll use different nodes.) While each node does have a small amount of local storage, the primary file storage facility is a large parallel disk system mounted on every node. Each user has a private home directory, a private "project space" for files used in classwork, and a private "scratch60 space" for short-term temporary files. All three areas can be accessed from every node of the cluster. When you login, you will start in your home directory. You can reach the other two spaces via the **`project`** and **`scratch60`** links in your home directory. Only your home directory is backed up, but files in your project space will remain until you remove them. Files in your scratch60 space will be deleted automatically after 60 days (with a 7-day warning).

On Grace, the login nodes are used solely for submitting and monitoring jobs, editing, and similar lightweight tasks. All program building (compilation, linking, etc.) and execution must take place entirely on compute nodes. The preferred approach is to use a node in the interactive partition to build and test programs. You should never have to wait for access to the interactive nodes, but your use of them will be limited to 4 cores (same as CPUs) on one node. (If you do have to wait for more than a few seconds, please let Dr. Sherman know.) When you actually run codes, you should use the 6 dedicated nodes, since you'll need to use more than one at a time in some cases.

The web page at http://research.computing.yale.edu/hpc-support/getting-started contains general information about how to get started on the Yale clusters. Your first task is to visit that page and familiarize yourself with how to login to Grace. (Your account should already be created for you; if not, please send email to Dr. Sherman.) Accounts have been requested for all students who have been accepted for enrollment in the class. Once your account is set up, you'll receive an email with further instructions about how to login. The instructions will differ somewhat depending on whether you come from a Linux, Windows, or Mac OS machine. We do not use passwords on the HPC clusters. To login to Grace, you will need to create an "ssh key pair" and then upload your public key (*not the private key*) so that it can be automatically installed in the **`.ssh`** subdirectory of your home directory. (See the getting-started page for details.) **NOTE: If you already have an account on Grace for research, nothing will change, except that you'll be added to the cpsc424 group and provided with an additional directory that you may use for this class if you wish.**

The usual workflow for using Grace is something like the following:

1. From a terminal window on either a Linux or Mac OS machine, login to Grace using the command "**`ssh`** `NetID@grace.hpc.yale.edu`" (substituting your own NetID, of course). [From a Windows machine, you will do the equivalent thing using a graphical tool like PuTTY; see the getting-started page mentioned above.] Since you'll use an **`ssh`** key, no password will be required. If you want to use any graphical X Windows programs on the cluster, be sure to include an **`ssh`** option for X forwarding. (For command-line **`ssh`**, the recommended option is **`-Y`**, though **`-X`** also works. For graphical **`ssh`** tools, like PuTTY on Windows, you should set corresponding preferences for the tool.) To run X Windows graphical programs, such as certain editors, or tools like TotalView (a graphical debugger), you must have an X Server on your machine. The ITS software site has a free one for Windows (XWin32), and for MacOS, you can freely download Xquartz from the Internet. Most Linux workstations or laptops will already have an X server installed to support the graphical desktop.

2. Your **ssh** command will put you on a Grace login node; it doesn't matter which one. On a login node, you can edit files, submit or check on batch jobs, or start sessions on compute nodes. All our HPC clusters use the Slurm resource management and scheduling system (see the YCRC website, or consult Chapter 5 of the text for more information about Slurm).

3. To start an interactive session on a compute node, run the following Slurm command to request a single core on an interactive node:

   ```
   srun --pty --x11 -p interactive -c 1 -t 6:00:00 --mem-per-cpu=6100mb bash
   ```

   If you're not planning to use a graphical tool (or don't have an X server on your local machine), omit the "**--x11**" option. The **--pty** option requests execution in pseudo terminal (interactive) mode, the **-c** option specifies 1 core; the **-t** option specifies 6 hours of run time, the **--mem-per-cpu** option specifies the number of megabytes of memory per core, and **bash** tells the system to run the bash shell. The **-p** option specifies the partition to use for this request ("**interactive**" in this instance). In the interactive partition, requests are limited to one session per user, 4 cores, and 6 hours of runtime, but you should not have to wait more than a few seconds for your interactive session to start. If you wish, you could start an interactive session on one of the class nodes using the same command as above, but using **cpsc424** instead of **interactive**.

4. For this assignment, you can do everything you need in a single-core interactive session. For other assignments in the class, you will need to run a batch session. To do that, you will create a batch job script and then submit it to Slurm using the **sbatch** command (instead of **srun**). For additional information on Slurm commands, see Chapter 5 of the text or look at the man pages for **srun** or **sbatch**.

Setting up your Linux software environment

At Yale, we use the Modules system to manage software environments. Whenever software is installed or updated, one or more "module files" are created to make it easy to set up environment variables and paths for the software. By default, only one module file is loaded when your session starts: "**StdEnv**". In order to use specific editors, compilers, libraries, and other tools or programs, you'll need to load additional module files. The information below applies to both interactive sessions and batch sessions, but you'll want to try this first in an interactive session to familiarize yourself with the Modules system.

To begin, run the command:

```
module list
```

This will list out the modules that are currently loaded. You should see something like:

```
Currently Loaded Modulefiles:
  1) StdEnv (S)

Where:
 S:  Module is Sticky, requires –force to unload or purge
```

To use a compiler, you need to load a compiler module file. Our module files are organized into sections such as "Langs" (programming or scripting languages), "Libs" (libraries), "Apps" (computational applications), "MPI" (for the MPI system), etc. For this assignment, you need to use the Intel compiler suite. Since you probably don't know the name of the right module file to load, run the command

```
module spider intel
```

that will list all the available module files whose paths contain the (case-insensitive) string "intel". (Running this command without an argument is one way to find out what software is available, although there are some software packages that are available but don't have module files.)

Among the listed module files, you'll find one named **Langs/Intel/2015_update2**, which is the one we want to use for this assignment. Now you can load the compiler module file with the command:

```
module load Langs/Intel/2015_update2
```

(tab completion is enabled for `module load`, so that you can type a partial string and hit tab (possibly several times) to see what module names start with what you typed. Now list out all the loaded module files again, and you should see something like:

```
Currently Loaded Modulefiles:
  1) StdEnv (S)    2) Langs/Intel/2015_update2

Where:
 S:  Module is Sticky, requires –force to unload or purge
```

To check that the right environment is loaded, run the following commands:

```
which icc
which ifort
icc --version
```

The output ought to be something like:

```
[ahs3@c01n01 a1]$ which icc
/gpfs/apps/hpc/Langs/Intel/2015_update2/composer_xe_2015.2.164/bin/intel64/icc
[ahs3@c01n01 a1]$ which ifort
/gpfs/apps/hpc/Langs/Intel/2015_update2/composer_xe_2015.2.164/bin/intel64/ifort
[ahs3@c01n01 a1]$ icc --version
icc (ICC) 15.0.2 20150121
Copyright (C) 1985-2015 Intel Corporation.  All rights reserved.
```

For additional information about the module system, you can use "`man module`".

If you always use some particular module files, you may want to put the module commands you need into your session initialization file. (For the bash shell, this is the file `.bashrc` in your home directory.) Simply type the commands on separate lines at the end of the file. In this case, you might add the line:

```
module load Langs/Intel/2015_update2
```

Keep in mind, though, that certain module files may conflict with others. For example, you wouldn't want to load two different versions of the same compiler suite. For that reason, you should be very careful about which module files you load in your `.bashrc` file, since they may interfere with other module files you need for a particular job. It's always a good idea to include a listing of loaded module files in your output.

Important Note About Timing

To measure performance in this class, we will use elapsed "wallclock time". You'll find a sample timing routine (`timing.*`) in `/home/fas/cpsc424/ahs3/utils/timing` that you may use, if you wish, though you are free to use other wallclock timing routines, if you prefer. The C function prototype for `timing()` is:

```
void timing(double* wcTime, double* cpuTime);
```

In Fortran, you can invoke the same timing function using:

```
call timing(wcTime, cpuTime)
```

When you build your code, simply include `timing.o` in the final link step. My timing function returns both the elapsed wallclock time from a particular pre-defined point in the past, and the cpu time consumed so far by your process. Neither of these is meaningful by itself, but differences between two wallclock or cpu times *may* be meaningful (though cpu time is often misleading). What most users care about is the elapsed wallclock time.

For this assignment, you may find that the chapters 1-3 of the Hager book are very helpful. We will cover some of that material in class, but you will find it beneficial to browse those chapters, as well.

Exercise 1: Division Performance (35 points)

Write and benchmark a code that approximates π by numerically integrating the function

$$f(x) = 1.0 / (1.0+x^2)$$

from 0 to 1 and multiplying the result by 4. You may use a very simple "mid-point" integration scheme that starts by creating a large number (N) of equally spaced points x_i covering the interval [0,1], with $x_0 = 1/(2N)$ and $x_i = x_{i-1} + 1/N$, for $1 \le i \le N$-1, and then approximating the integral as the sum of the areas of rectangles centered around each point. Each such rectangle has width $\Delta x = 1/N$ and height $f(x_i)$. For this assignment, using $N = 1000000000$ (1 billion) would be reasonable.

To create a complete benchmark program in C or Fortran, write code to implement the mid-point scheme, including suitable timing function calls (see above). Demonstrate that your program actually computes a reasonable approximation to π (there are a number of simple ways to do this, if you think about it a bit), and report runtime and performance in MFlops (millions of floating point operations per second) using *one core of one compute node*. Use the Intel compiler suite and report results using the following combinations of `icc` compiler options (or similar options for `ifort`):

a. `-g -O0 -fno-alias -std=c99`

b. `-g -O1 -fno-alias -std=c99`

c. `-g -O3 -no-vec -no-simd -fno-alias -std=c99`

d. `-g -O3 -xHost -fno-alias -std=c99` (Recommended for real codes.)

[For more information on the Intel compiler options, look at the man page for `icc` or `ifort`, or search the Intel website. You will need to load the compiler module file before you can find the man pages.]

Try to explain your results ***briefly*** by relating them to the architecture of the processor. (See class notes or look up the Intel Ivy Bridge architecture on the web. You could also look at the older X5560 Nehalem architecture, instead.) I'm looking for a conceptual answer here, preferably with some quantitative backup that justifies the answer. There may be more than one reasonable explanation.

Use timings of your code (and/or some modest variations of it) to estimate the latency of the divide operation (measured as a number of CPU cycles to obtain a divide output). Be sure to explain how you got your estimate. For this part of the problem, you may assume that the processor runs at clock rate of 2.2 GigaHertz, and that the cycle time is the reciprocal of the clock rate. (You can find out the base clock speed and lots more information about the node by running:

```
cat /proc/cpuinfo
```

to see what it tells you about the node.)

Exercise 2: Vector Triad Performance (65 points)

Write and benchmark a program that measures the performance in MFlops of the vector triad kernel:

```
a[i] = b[i] + c[i] * d[i]
```

Here `a`, `b`, `c`, and `d` are double precision arrays of length `N`. Allocate memory for these data structures on the heap, using `malloc()` or `calloc()` in C or `allocate()` in Fortran90. (For Fortran77, you'll have to precalculate how large the arrays need to be and allocate them accordingly in your main program.) You should initialize all data elements with valid random floating point numbers in [0,100]. (For fun, you could try running it with $N = floor(2.1^{25})$ in C using `calloc()` without initialization to see what happens.) Benchmark your code with $N = floor(2.1^{k})$, $k = 3...25$.

To generate random numbers in C, use the function `rand()` that generates random integers in the interval [0,RAND_MAX]. (See the man page for more information.) You can then convert these to double precision numbers `r` by using something like:

```
drand_max = 100.0 / (double) RAND_MAX;
r = drand_max * (double) rand();
```

For this exercise, you may wish to try all the compiler options from Exercise 1, but please use option (d) to generate the data for the plot requested below.

To get reasonable timing accuracy, insert an extra loop that ensures that, for each value of N, you time a computation that is at least 1 second in duration. That is, you want to run the kernel multiple times so that the total computation takes at least 1 second, and then scale the total time appropriately to calculate the time for a single execution of the kernel. It would be best to dynamically adjust the number of repetitions depending on the runtime of the kernel, along the lines of the following code fragment:

```
int repeat = 1;
double runtime = 0.0;
while(runtime < 1.0) {
  timing(&wcs, &ct);
  for (r=0; r<repeat; r++) {
    /* PUT THE KERNEL BENCHMARK LOOP HERE */
    if (CONDITION_NEVER_TRUE) dummy(a); // fools the compiler
  }
  timing(&wce, &ct);
  runtime = wce - wcs;
  repeat *= 2;
}
repeat /= 2;
```

You may need to be careful to make sure that the operations in the kernel actually get executed. (Compilers are smarter than you think!) A simple way to do this is to insert a fake conditional call to an opaque function. In the above example, the conditional call to `dummy()` serves this purpose. (Note that the opaque function must reside in a separate source file (and you must not let the compiler do too much interprocedural optimization). Also, you need to ensure that the compiler can't easily determine the result of the condition statement at compile time. One possible condition might be something like: "`if (a[N>>1]<0.)`", which will never be true if all the arrays are initialized with positive numbers.

Use your favorite plotting program (e.g. gnuplot or Excel or Matlab, not necessarily on Grace) to generate a plot of the performance in MFlops vs. N. **Use a logarithmic scale for N on the x-axis**. You should see a number of interesting performance variations on your plot. Try to explain the variations in terms of the processor architecture, by computing and discussing the apparent memory bandwidth that

your data show near the variations in your plot. [**Hint**: You can estimate the apparent bandwidth, often expressed in GB/sec, as the product of Gflops and bytes/flop.]

Additional Notes:

1. When a batch submission (not the job execution) succeeds, you will receive output from sbatch that gives you the job number. You can check on the status of that specific job using a command like:

   ```
   squeue -j your_job_number_here
   ```

   or, to check all your current jobs (either running or pending):

   ```
   squeue -u your_netid_here
   ```

   Most important to you is the status indicator (under heading "`ST`"), which will usually be either "`PD`" (if your job is pending/waiting to run) or "`R`" (if your job is running). You can find other information about `squeue` on its man page.

   If your job has finished running, then `squeue` may still report on the job for a brief time. More often, you'll need to use the `sacct` command, as in:

   ```
   sacct -u your_netid_here
   ```

   or

   ```
   squeue -j your_job_number_here
   ```

Procedures for Programming Assignments

For this class, we will use the Yale Canvas website to submit solutions to programming assignments.

Remember: While you may discuss the assignment with me, a ULA, or your classmates, the work you turn in must be yours alone and should not represent the ideas of others!

What should you include in your solution package?

1. **All source code files, Makefiles, and scripts that you developed or modified.** All source code files should contain proper attributions.
2. **A report in PDF format** containing:
   a. Your name, the assignment number, and course name/number.
   b. Information on building and running the code:
      i. A brief description of the software/development environment used. For example, you should list the module files you've loaded.
      ii. Steps/commands used to compile, link, and run the submitted code. Best is to use a Makefile for building the code and to submit an `sbatch` script for executing it. (If you ran your code interactively, then you'll need to list the commands required to run it.)
      iii. Outputs from executing your program.
   c. Any other information required for the assignment (e.g., in this case, answers to questions and the plot).

How should you submit your solution?

1. On the cluster, create a directory named "`NetID_ps1_cpsc424`". (For me, that would be "`ahs3_ps1_cpsc424`". Put into it all the files you need to submit.
2. Create a compressed tar file of your directory by running the following in its parent directory:

   ```
   tar -cvzf NetID_ps1_cpsc424.tar.gz NetID_ps1_cpsc424
   ```

3. To submit your solution, click on the "Assignments" button on the Canvas website and select this assignment from the list. Then click on the "Submit Assignment" button and upload your solution file `NetID_ps1_cpsc424.tar.gz`. (Canvas will only accept files with a "`gz`" or "`tgz`" extension.)You may add additional comments to your submission, but your report, including the plot, should be included in the attachment. You can use scp or rsync (Linux or Mac) or various GUI tools (e.g., WinSCP or CyberDuck) to move files back and forth to Grace.

Due Date and Late Policy

Due Date: **Thursday, February 1, 2018 by 9:00 a.m.**

Late Policy: On time submission: Full credit

Up to 24 hours late: 90% credit

Up to 72 hours late: 75% credit

Up to 1 week late: 50% credit

More than 1 week late: 35% credit

General Statement on Collaboration

Programming, like composition, is an individual creative process in which you must reach your own understanding of the problem and discover a path to its solution. During this time, discussions with others (including the instructional staff) are encouraged.

However, when the time comes to write code, such discussions are no longer appropriate—the program must be your own personal inspiration (although you may ask the instructional staff for help in writing and debugging).

Since code reuse is an important part of programming, you may incorporate published code (e.g., from text books or the net) in your programs, provided you give proper attribution, you are complying with the licenses (if any) associated with the code, and *THE BULK OF THE CODE SUBMITTED IS YOUR OWN*.

DO NOT UNDER ANY CIRCUMSTANCES COPY ANOTHER PERSON'S CODE—to do so is a clear violation of ethical/academic standards that, when discovered, will be referred to the appropriate university authorities for disciplinary action. Modifying code to conceal copying only compounds the offense.