CPSC 424/524 Fall 2018
Assignment 1
Due Date: 9/12/2018, 9:00am

In this assignment, you will access the Omega Linux cluster (`omega.hpc.yale.edu`) and run C programs to investigate the characteristics of the processors on the cluster. For detailed information about the Yale clusters, please consult the website for the Yale Center for Research Computing (https://research.computing.yale.edu). (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 Omega, please see https://research.computing.yale.edu/support/hpc/clusters/omega.

Omega has two cluster login nodes (`omega1 and omega2`) and several hundred compute nodes, of which 32 are dedicated for our class to use this semester. In addition, you may use nodes in other partitions on the cluster, including the "interactive partition," which is used for editing, program building, certain types of executions, etc. The nodes we'll use for most of the semester have two 4-core Intel Xeon processors (Intel Xeon X5560) and 36 gigabytes of memory. (When we come to GPU programming, we'll use different nodes on the Grace cluster.)

While each node does have a small amount of local storage, the primary file storage facility on Omega is a large parallel disk system mounted on every node. Each user has a private home directory and a private scratch directory for files used in classwork. Both areas can be accessed from every node of the cluster. When you login, you will start in your home directory. You can reach your scratch directory via the `scratch` link in your home directory. Only your home directory is backed up, but files in your scratch space will remain until you remove them or you are no longer enrolled in the class.

The cluster 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 one core (same as "processor" or CPU) on a node in the interactive partition to build programs. If your request for a session in the interactive partition is not fulfilled within a few seconds, you can also build programs on one of the class's dedicated nodes. When you actually run codes, you should use the dedicated nodes only so that you are the only user of the nodes you use. Note that sessions on the interactive nodes are limited to 6 hours, while sessions on the dedicated nodes are limited to 30 minutes.

The web page at https://research.computing.yale.edu/support/hpc/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 Omega. (Accounts have been requested for all students in the class, so your account should already be created for you; if not, please send email to Dr. Sherman.) Once your account is created, 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 Omega, 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 (hidden) `.ssh` subdirectory of your home directory. (See the getting-started page for details.) **NOTE: If you already have an account on Omega 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. Your home directory will remain as it was before.**

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

1. From a terminal/command-line window on your local machine, login to Omega using "`ssh NetID@omega.hpc.yale.edu`" (substituting your own NetID, of course). [On a Windows machine, you may first have to install an ssh client; see the web page at https://research.computing.yale.edu/support/hpc/user-guide/connect-windows for current recommendations.] 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`. To run X Windows graphical programs, such as certain editors, or tools like TotalView (a graphical debugger for parallel programs), you must have an X Server on your local machine. Most Linux machines will have one out of the box. For MacOS, you can freely download the Xquartz server from the Internet. For Windows machines, we recommend MobaXterm, as described here: https://research.computing.yale.edu/support/hpc/user-guide/connect-windows.
2. Your `ssh` command will put you on an Omega 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. You **must not** execute computational programs (even compilers) on the login nodes because they are shared by many users. Instead, you should request an interactive or batch session on one or more compute nodes by using the Slurm resource management and scheduling system.
3. To start a one-hour command line terminal session on a compute node in the interactive partition, run the following Slurm command to request a single core on an interactive node:

   ```
   srun --pty --x11 -p interactive -c 1 -t 1:00:00 --mem-per-cpu=4100mb 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 1 hour of run time, the `--mem-per-cpu` option requests memory by specifying 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. 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` as the partition name, and limiting your runtime to 30 minutes (`-t 00:30:00`).
4. For this assignment, you can do everything you need in a single-core interactive session on one of the class nodes in the cpsc424 partition. For other assignments in the class, you will need to run a batch job. 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 the man pages for `srun` or `sbatch` and the YCRC website.

Setting up your Linux software environment

At Yale, we use the LMod 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 jobs, but you'll want to try this first in an interactive session to familiarize yourself with the LMod 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/15.0.2`, which is the one we'll use for this class. Now you can load the compiler module file with the command:

```
module load Langs/Intel/15.0.2
```

(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/15.0.2

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
icc --version
```

The output ought to be something like:

```
[ahs3@c15n01 a1]$ which icc
/home/apps/fas/Langs/Intel/2015_update2/composer_xe_2015.2.164/bin/intel64/icc
[ahs3@c15n01 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/15.0.2
```

Keep in mind, though, that some module files may conflict with others. 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 generally use elapsed or "wallclock" time. You'll find a sample timing routine (`timing.*`) in `/home/fas/cpsc424/ahs3/utils/timing` that you may use, 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);
```

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*. (Note that we'll generally ignore all operations other than floating point operations in assessing per-operation program efficiency in this class. Use the Intel compiler suite and report results using the following combinations of `icc` compiler options:

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 search the Intel software 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, the Hager book, or look up the Intel Nehalem architecture on the web.) I'm looking for a conceptual answer here, but, for full credit, you should provide some quantitative justification. 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 (expressed 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 2.8 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 you're using.)

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. 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 graphing program (e.g. gnuplot or Excel or Matlab, not necessarily on Omega) to create 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 changes on your plot. Try to explain these in terms of the processor architecture, by computing and discussing these changes in apparent memory bandwidth.

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
   ```

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 Omega.

Due Date and Late Policy

Due Date: **Wednesday, September 12, 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

Unless instructed otherwise, all submitted assignments must be your own individual work. Neither copied work nor work done in groups will be permitted or accepted without prior permission.

However….

You may discuss questions about assignments and course material with anyone. You may always consult with the instructor or TFs/ULAs on any course-related matter. You may seek general advice and debugging assistance from fellow students, the instructor, TFs/ULAs, Piazza conversations, and Internet sites.

However, except when instructed otherwise, the work you submit must be entirely your own. If you do benefit substantially from outside assistance, then you must acknowledge the source and nature of the assistance. (In rare instances, your grade for the work may be adjusted appropriately.) It is acceptable and encouraged to use outside resources to *inform* your approach to a problem, but *plagiarism is unacceptable* in any form and under any circumstances. If discovered, plagiarism will lead to adverse consequences for all involved.

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.