CPSC 424/524 Fall 2023
Assignment 1
Due Date: September 21, 2023, 11:59pm

In this assignment, you will access the Grace Linux cluster (**grace.hpc.yale.edu**) and run C programs to investigate the characteristics of the processors on the cluster. For information about Grace and other clusters run by the Yale Center for Research Computing (YCRC), visit the YCRC's technical website at https://docs.ycrc.yale.edu/. A good place to start is the general clusters page that is located at https://docs.ycrc.yale.edu/clusters-at-yale/. For information specifically about Grace, please see https://docs.ycrc.yale.edu/clusters-at-yale/clusters/grace/. In order to use Grace, you must either be on campus or logged into Yale's virtual private network (VPN).

Grace has two login nodes and hundreds of compute nodes containing thousands of cores/CPUs, of which a few nodes are reserved for our class to use this semester. All the cluster nodes are managed by a scheduler/resource manager ("Slurm"), and all the class nodes are part of a single scheduler "reservation" named `cpsc424`. The nodes we'll use for most of the semester have two 48-core Intel Xeon processors (Intel Xeon Platinum 8268, known as "Cascade Lake" processors) and 384 gibibytes (GiB) of memory. (When we get to GPU programming, we'll use different nodes.)

While each node does have a small amount of local temporary storage, the primary file storage facilities are large parallel disk systems mounted on every node. Each user has a private home directory (`/home/cpsc424_netid`) with a storage quota of 125 GiB. In addition, users have access to two shared file spaces that may be used to do class assignments. Your "project space" (`~/project`) may be used for files that you want to keep throughout the semester. Your shared "scratch space" (`~/palmer_scratch`) may be used for larger amounts of data necessary to make a particular run, but keep in mind that files in `palmer_scratch` are considered temporary and will be erased after 60 days. In addition, we will remove **all** class-related files one month after the end of exam period each semester.

Important Notes About Storage:

1. The space in `project` and `palmer_scratch` is shared by everyone in the class, and we have set class quotas of 1 TiB for `project` and 10 TiB for `palmer_scratch`. In most cases, you will not need to use the shared spaces for your own files, since your home directory should suffice for your classwork. However, some shared files for use by the class will be stored in `project`, and if you need to use particularly large files temporarily, you may need to use one of the shared spaces. If you do use a shared space, you should set permissions for your shared-space directory to "700" to make your directory a private one for yourself. (The directory meant here is either `/gpfs/gibbs/project/cpsc424/cpsc424_netid` for `project` or `/vast/palmer/scratch/cpsc424/cpsc424_netid` for `palmer_scratch`.)
2. For now, you should assume that only your home directory will be backed up!! It's very rare that we lose data in the `project` or `palmer_scratch` spaces, but it could happen. If you do use the shared spaces, keep important files (or copies of them) in your home directory, just in case.
3. To see how much of your storage space is in use, you can run the `getquota` command.

On Grace, the login nodes are used solely for submitting and monitoring batch jobs, editing, and similar lightweight tasks. Everything else should take place on compute nodes or, in some cases, on data transfer nodes. This includes all program building (compilation, linking, etc.) and execution. To make things simpler, we recommend that you access Grace using "Open OnDemand" (OOD), a browser-based graphical user interface. (See the "OOD for Courses" section on the YCRC's OOD webpage: https://docs.ycrc.yale.edu/clusters-at-yale/access/ood/.) To get started, you may find it helpful to view the YCRC's Youtube training video at https://www.youtube.com/watch?v=w1hbOppyUUc.

To login to OOD on Grace using your web browser, simply visit the class OOD web page: `https://cpsc424.ycrc.yale.edu` and enter your netid and password. (If you're off campus, remember to login to Yale's VPN before you do this.) Once you login, you'll mainly use these OOD tools: a Linux Remote Desktop tool that opens a graphical desktop environment on one or more compute nodes, a graphical file browser (on the `Files` pull-down) that allows you to manage files and transfer them between Grace and your personal computer, and a Shell tool (on the `Clusters` pull-down) that opens a command-line session on a login node,

For this assignment, you must do everything in a single-core session on a compute node, either interactively or in a batch job. For most assignments later in the course, you will almost always need to use batch jobs. To do that, you will create a batch job script and then submit it to Slurm using the `sbatch` command. For additional information on Slurm commands, search online or look at the YCRC website or the `man` pages for `slurm, sbatch` or `salloc`. In this assignment, we have provided you with a sample batch script that you could use for Exercise 1 (below).

Creating an interactive Remote Desktop session

To begin, login to OOD and create an interactive desktop session by clicking on the Remote Desktop tool in OOD, as discussed in the OOD video at https://www.youtube.com/watch?v=w1hbOppyUUc. This will open a form that you can fill out to specify settings for the Remote Desktop. For this assignment, you will need to specify the following:

Parameter Label	Setting
Number of hours	6 or as needed (up to 24 hours)
Number of CPU cores per node	1 for this assignment, or up to 48 for future assignments
Number of GPUs per node	0
Memory per CPU core in GiB	7
Partition	`day` (lower case)
Reservation	`cpsc424` (lower case)
Slurm account	Not needed for this assignment. Leave blank
Additional job options	Not needed for this assignment. Leave blank

Once you have filled these out, click the Launch button. You will see a new window showing your Remote Desktop request, which will be labelled either "Queued" or "Running." Once it is running, you can click on the button labelled "Launch Remote Desktop" to start your session and display the remote desktop with a command-line window in it. The command-line prompt will be similar to:

```
[cpsc424_ahs3@r918u05n01.grace ~]$
```

This shows that you're running in your home directory ("`~`") on the compute node named `r918u05n01`. You can resize or move the command-line window just as in any other graphical desktop environment.

For basic information about the nodes we are using, you might try running commands like `lscpu` or `lsmem`. Note that our nodes on Grace are configured with the hyperthreading option turned off. So the number of cores reported are actual physical cores. (If you don't know what hyperthreading is, look for information on line.)

Setting up your Linux software environment

At Yale, we use the LMod module system to manage software environments. Whenever Linux software is installed or updated, a "module file" is 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 LMod system.

In the session you just created, 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 module file. For this assignment, you need to use the Intel compiler suite. Since you probably don't know the name of the module file to load, run the command

```
module avail intel
```

to list all the available module files whose names contain the (case-insensitive) string "`intel`". (There are many of them, so you may need to hit the space bar several times to get all the way to the end.)

Among the listed module files, you'll find one named `intel/2022b` (near the bottom), which is the one we want to use for this assignment. Now you can load the compiler module file with the command:

```
module load intel/2022b
```

Actually, you could omit the "`/2022b`," since that version is the default, denoted by "`(D)`" in the module list. Now list out all the loaded module files again, and you should see a long list of modules with the one you just loaded at the end. (Except for `StdEnv` all the other modules in the list are dependencies of the `intel/2022b` module file that were loaded automatically by LMod.)

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

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

The output ought to be something like:

```
[ahs3@c01n01 a1]$ which icc
/vast/palmer/apps/avx2/software/intel-
compilers/2022.2.1/compiler/2022.2.1/linux/bin/intel64/icc


[ahs3@c01n01 a1]$ icc --version
icc (ICC) 2021.7.1 20221019
Copyright (C) 1985-2022 Intel Corporation.  All rights reserved.
```

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

At this point, you could edit, compile and run code, or do anything else you might need to do in an interactive session on a compute node. (Normally, you shouldn't run long jobs interactively. Instead, you should submit a batch job, as we'll discuss later.) You can reach a long list of tools, editors, and utilities via the Applications pull-down menu at the top left of the desktop.

Whenever you are done with your Remote Desktop session, select the "Log Out" option from the System menu somewhat to the right of the Applications menu. Then select the "Log Out" option from the dialog box to confirm that you want to log out. Finally, close the browser tab that you've used for your Remote Desktop session. (This should leave you in the "My Interactive Sessions" tab.) Logging out of the Remote Desktop session does not terminate your OOD session, so you could continue your work using any of the OOD tools. To fully terminate your entire OOD session, click on the logout icon on the far right of the Grace-onDemand menu bar, and then close your browser. If you are using a public computer, you should also clear the browser's cache. (Alternatively, you could use incognito mode to avoid this step, if you wish.)

Important Note About Timing

To measure performance, we will use elapsed "wallclock time". You'll find a sample timing routine (`timing.*`) in `/gpfs/gibbs/project/cpsc424/shared/utils/timing`. If you wish, though, you may use other **wallclock** timing routines. The C function prototype for the timing routine we've provided is:

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

When you build your code, simply include `timing.o` in the final link step. The timing function we've provided 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 chapters 1-3 of the Hager book are helpful, especially chapter 1. (See the slides from the first lecture of the semester for a list of course resources.) Lectures will cover some, but not all, of that material.

Exercise 1: Division Performance (25 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. It 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 $1 \le i \le N-1$. Then, it approximates 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. All floating point computations should use double precision arithmetic.

To create a complete benchmark program in C, 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 experiment with a variety of compiler options. Report results using at least the following combinations of `icc` compiler options:

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

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

c. `-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 website. You will need to load the compiler module file before you can find the man pages.]

Here are some questions for you to address in your report:

1. The performance of the pi calculation is probably dominated by the cost of division operations. Suppose you assume that floating point division cannot be pipelined. In that case, can you use the performance results from the pi calculation to estimate the latency of the divide operation? Does it matter which compiler options you use? (Explain your answer.)
2. Is the observed performance what you expected for the different compiler options (the ones above, plus any others you care to try)? Try to explain the comparative performance for these runs by relating them to your conceptual understanding of the processor architecture. To inform your thinking about the performance, you may want to search online for information on the Intel Cascade Lake architecture and the effect of compiler options like `-no-vec` and `-no-simd`.

To help you get started, I've provided several files in the following directory: `/gpfs/gibbs/project/cpsc424/shared/assignments/assignment1`. One of the files is a Makefile that can be used to build the codes for both Exercise 1 and Exercise 2). Another is a sample batch script that you could submit to Slurm using `sbatch`. One important point to note is that Slurm starts batch jobs in the directory from which you submit them, not from your home directory.

Exercise 2: Vector Triad Performance (75 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 $N = floor(2.1^{26})$ in C using `calloc()` without initialization to see what happens.) Benchmark your code with $N = floor(2.1^{k}), k = 3...25$. Note: You may need to modify the `--mem-per-cpu` option in your batch script to increase the amount of memory requested for your job to be able to allocate memory for the larger vectors.

To generate random numbers in C, use the function `rand()` that generates random integers in the interval [0,RAND_MAX]. (See its 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 some of the compiler options from Exercise 1, but please use option (c) 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 may be 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 should reside in a separate source file (and you should 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 conditional statement at compile time. One possible conditional might be something like: "`if (a[N>>1]<0.)`", which will never be true if all the arrays are initialized with positive numbers. I've provided a `dummy.c` file that you can use for this purpose. (It's in `/gpfs/gibbs/project/cpsc424/shared/assignments/assignment1`.)

Use your favorite plotting program (e.g., gnuplot or Excel or MATLAB) to generate a plot of the performance in MFlops vs. N. **Use a logarithmic scale for N on the x-axis, along with a standard scale for MFlops**. Explain what you see in your plot and try to tie it quanitatively to characteristics of the Cascade Lake CPU that you're using. (You need not generate the plot on Grace, though MATLAB can be run from OOD.)

Additional Notes:

1. When a batch submission (not the actual job execution) succeeds, you will receive output from Slurm that tells you the job number. You can then 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_username_here -a
   ```

   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 --me
   ```

   or

   ```
   sacct -j your_job_number_here
   ```

   Note that OOD also contains a tool to check on your submitted batch jobs. You can find it on the Jobs pull-down menu on the "My Interactive Sessions" tab.

2. I have provided several files for you to use in this assignment:

   a. Timing routines: Located in `/gpfs/gibbs/project/cpsc424/shared/utils/timing`

   b. Makefile, batch script, dummy.c: Located in:

      `/gpfs/gibbs/project/cpsc424/shared/assignments/assignment1`

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 the instructor, a ULA or your classmates, the work you turn in must be yours alone!

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, as needed. In addition, make sure that your source code contains suitable documentation (e.g., comments) so that someone looking at your code can understand what it's trying to do. In part, your grade will depend on the documentation, code organization, and comments for your code.
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 Slurm script that you submit using `sbatch`. The script could use a Makefile to build the code and would then run it multiple times so that you can report average timings. (If you ran your code interactively, then your report will 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, including your report.
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`. (We will set Canvas to accept only 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 OOD, tools like scp or rsync, or various GUI tools (e.g., CyberDuck) to move files back and forth to Grace.

Due Date and Late Policy

Due Date:	**Thursday, September 21, 2022 by 11:59 p.m.**
Late Policy:	On time submission: Full credit
	Up to 24 hours late: 85% credit
	Up to 72 hours late: 50% credit
	Up to 1 week late: 25% credit
	More than 1 week late: No 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 ULAs on any course-related matter. You may seek general advice and debugging assistance from fellow students, the instructor, ULAs, Canvas ED discussions, and Internet sites, including https://ask.cyberinfrastructure.org/c/yale/40 or similar sites.

However, except when instructed otherwise, the work you submit (e.g., source code, reports, etc.) 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.