# Final Exam: CSE 443/543 Spring 2020

The final exam for CSE443/543 takes place on Monday, May 11:

* Section A: 12:45 PM – 2:45 PM
* Section B: 3:00 PM – 5:00 PM

It will be administered via Canvas and proctored via Proctorio as the second exam was.

Like the first two exams, this exam is closed book and no notes are permitted. You cannot look up information online during the exam. Proctorio will limit you to a single Google Chrome window.

The exam will consist of multiple choice, matching, true/false, and short answer questions. You will need to read code (C++ and bash shell scripts) and determine what its output will be or identify key concepts in the code. You will also have to identify errors in code and how to remedy them. You will need to perform simple mathematics and may use the calculator built into Proctorio for this.

The exam will emphasize material on MPI, covered since Exam 2, but will include topics from the entire course.

This is a summary of the key concepts you should understand for this exam:

## High Performance Computing architecture

* Performance of processors and HPC systems (FLOPS)

FLOPS – Floating Point Operations Per Seconds, are the standard metric for determining how well a processor can perform.

Theoretical FLOPS = CPU Clock Frequency / Cycles Per Instruction (CPI)

For an entire system, sum up the theoretical FLOPS for a whole system

Summit runs at 148.6 PetaFlops

* Architectural components of HPC systems
  + Hierarchical storage
    - In essence, there is a place for everything, and everything has a place.
    - All nodes have access to home directories via NFS(Network File System)
    - All nodes have access to some local storage
    - Some or all nodes have access to temporary storage
    - Some or all nodes have access to high speed flash storage
  + Dedicated networks
    - These are in-house or proprietary systems that manage storage for all the nodes
    - Infiniband and Fibre Channel are common proprietary system
    - Large, dedicated storage servers provide data redundancy, increase access speed, and enable large amounts of data to be available
  + Multiple types of nodes in an HPC system
    - Interactive Nodes – Users can log in to edit programs, submit jobs, check job statuses
    - Debug Nodes – Users can submit jobs during the development process
    - Compute Nodes – Production jobs are run here. Most of these nodes have the same configuration in terms of processor type, memory, and local storage available

## Processor architecture

* Moore’s Law
  + Limits to processor performance
    - Moore’s law states that every two years, the number of transistors on microprocessors should double every two years
    - This amount has slowed down to every 2.5 years
* CPU clock and cycles per instruction
  + Time/frequency units
    - CPU clock boils down to a 0 or 1 binary signal, used to determine CPU operations
    - CPU operations are timed based on how many cycles it takes to perform them
    - Note that time is inversely proportionate to frequency
    - Operations are measured in nanoseconds (10-9 seconds) and gigahertz (109 Hz)
  + Processing power
    - Clock speed is measured in GHz, representative of how fast a CPU can process instructions
* Components of a modern processor
  + Hierarchical memory
    - Memory, Cache, Registers
  + Compute cores (ALUs)
  + Other kinds of units
    - Fetch, decode, write-back
    - Control unit
    - Memory management unit
  + Superscalar architecture
    - Architecture that allows the simultaneous processing of multiple instructions at the same time by introducing additional hardware in each core/CPU. Instructions can be executed out of order if there are no contingencies

## Linux commands

* Commands from the “Common Linux Shell Commands” document, and others introduced in subsequent labs and homework problems

## Benchmarking/Profiling

* Collecting data for benchmarking
  + /usr/bin/time command
    - /usr/bin/time -f "%U %S %e" ./f\_serial 65536 2>> allTestRuns.txt
    - A command to grab user, system, and elapsed time for the runtime of a program
* Types of profilers: Uses, pros and cons
  + Hardware
    - Use specialized hardware to monitor the CPU and the programs running on it
    - Adv: Most accurate, observe minute details without skewing system behavior, only way to troubleshoot hardware -> software issues
    - Dis: Tend to be expensive, require physical connections to a computer (many computers don’t have a JTAG connector), require a good understanding of hardware and software
  + Hardware assisted
    - Use specialized profiling features available on CPUs to monitor programs running on them (Periodic sampling is used, so some approximation is used)
    - Adv: Fast. Can be used to profile OS internals (under admin privileges) like syscalls and device drivers, can also provide data on a per-core basis
    - Dis: Polling/sampling rate influences effectiveness, interpreting detailed profile data can be daunting, requires a good understanding of the hardware (assembly) in order to be used
  + Software profilers
    - Instrument the program to record its behavior
    - Adv: Cheaper than hardware profilers, do not require hardware modifications so a programmer or end-users can readily use them
    - Dis: Cannot be used to profile hardware devices, additional code used to profile can cache and skew results, not as accurate (typically around ~95%), slows programs down by 4 to 5 times, up to even 20, bad for games
  + Hypervisors or simulators
    - Software that provides a virtual machine for running programs, in this case, a simulator that interprets and executes instructions
    - Adv: No source or binary code modifications, can generate more accurate trace than software profilers, can be used to emulate non-native hardware or CPUs
    - Dis: Cannot be used to profile hardware devices, developing a simulator is time consuming, time for profiling can be very high, simulators slow down program by 50 times or more, slow execution prevents use for programs that require real time interaction with user
  + Hybrid profiler

Any combination of the above, suffering similar advantages and disadvantages

* Linux perf command

A hardware-assisted profiler.

perf stat -e branches, branch-misses ./executable

Or

perf stat -d -d -d ./executable

What we used in class to get information on the specifics

## **Instruction Level Parallelism**

* Definition of parallelism and its goal

Parallelism – The ability to perform two or more tasks at the same time using independent computational devices. Devices may include

* + Separate ALUs
  + Separate Cores
  + Separate Processors
  + Separate Computers

Goal: Utilize concurrency in a task to reduce overall computational time to process said task

* Definition of instruction level parallelism (ILP)

ILP – Task concurrency performed automatically by a microprocessor (several instructions executed at the same time)

* + What is pipelining and how can it speed up execution

Processing of an instruction is divided into several independent stages, various stages of different instructions are overlapped, enabling processing of multiple instructions to proceed simultaneously

* + Superscalar execution

Multiple Instructions are processed in parallel by introducing additional hardware components in each core/CPU. Instructions may even be executed out of order

* Processing cycle (Fetch, Decode, Execute, Write Back)

1. Fetch – Retrieve instruction from cache or memory
2. Decode – Convert instruction to micro program, grab additional operands if necessary
3. Execute – Process the instruction using an ALU, possible also the FPU (Floating Point Unit)
4. Write Back – Store results back into memory or registers

* Hazards

Otherwise known as a stall, anything that forces the pipeline to skip processing instructions in a given cycle.

* + Data hazard and operand forwarding

A data hazard occurs when there are interdependencies between instructions. For example, if there are two instructions, and the second needs output from the first to use, that means it is dependent on the first and it must wait on the first instruction to complete

Operand forwarding – Occasionally, the CPU can detect a data dependency that would cause a stall, forward that data ahead, and ‘catch up’ later on.

* + Control hazard and branch prediction

Control hazards occur due to branching. Every time we branch, new instructions need to be fetched and the pipeline must be flushed and refilled (huge time deficit)

Branch Prediction – The CPU’s way of guessing which way a branch (a conditional statement like an if) is going to go (true or false). If we guess right, we gain efficiency, if we guess wrong, we are worse off than wen we started

* + - Branch misses and how to measure them

Branch Misses - When the branch predictor does not guess correctly

Measured using:  
perf stat -e branches, branch-misses ./executable

Or

perf stat -d -d -d ./executable

* + Loop unrolling

Loop Unrolling:

1. Completely unrolling a loop (like a for) into a massive chain of if statements to decrease overhead of incrementing loop index and testing loop conditions
2. Or, increasing the increment index and adding more statements in the body of a loop
   * Method inlining

Method Inlining- Method calls take up a large amount of overhead. In the case of small method calls, it is more efficient to simply write the body of the method out, even if it is repeated elsewhere.

Note \*\*: This practice and Loop Unrolling violate good coding practices.

* + Structural hazard

Structural Hazard – Any hazard that arises due to the limitations of hardware. This may include: needing to read & write to a single memory location simultaneously, two instructions may need to write data at the same time, or memory may not keep up with CPU speed.

## **Parallel Computing**

* Levels of tasks (application, program, thread, method/function, instruction)

1. Application – Multiple programs or processes running simultaneously
2. Program/Process – Multiple treads running at one time
3. Thread – Multiple methods/functions running at one time
4. Method/function – Several Instructions running at the same time
5. Instruction – Multiple micro-instructions running at the same time

* Concurrency

Two tasks are said to be ‘concurrent’ if there is no dependency or relationship between them.

* Definitions of implicit and explicit parallelism
  + Implicit Parallelism – Parallelism that is automatically or semi-automatically performed. Scope is typically limited to a single processor or single machine. Typically operating at the multi-thread or multi-instruction level. Requires very little effort on the programmer’s end
  + Explicit Parallelism – Parallelism is manually or programmatically performed. The programmer is responsible for developing the program to run in parallel. Typically operates at multi-thread or multi-processes level. Requires significant programming efforts. Can be applied to multiple processors, machines, or supercomputing clusters.
* Definitions of Flynn’s Taxonomy:  SISD, SIMD, MISD and MIMD

Flynn’s taxonomy – How we classify modern CPUs and Computer Systems based on the number of instruction and data streams they use

* + SISD (Single Instruction Single Data) – One instruction at a time operates on exactly one unit of data
  + MISD (Multiple Instructions Single Data)  - Rare architecture, multiple processors perform same or different operations on the same data. Designed for fault tolerance – the space shuttle used MISD processing.
  + SIMD (Single Instruction Multiple Data) – A single instruction performs the same operation on multiple data elements in parallel (performed by microprocessors automatically)
  + MIMD (Multiple Instructions Multiple Data) – Multi-core processors and supercomputing clusters fall in this category. Multiple CPUs perform different instructions on multiple data simultaneously.
* Profile-guided optimization

A way of using the runtime behavior of a program for optimization. The compiler can reorganize instruction to reduce data hazards, unroll loops to minimize control hazards, restructure if-else statements to place most frequently taken paths first, reorganize instructions to effectively utilize superscalar architecture

In practice:

g++ -fprofile-generate –O3 –std=c++14 –g –Wall prog.cpp –o prog ./prog

g++ -fprofile-use –O3 –std=c++14 –g –Wall prog.cpp –o prog

* Explicit threading
  + Threads and how they are similar to/different than processes

Both threads and processes are independent sequences of execution (multiple threads can be running at one time, the same as multiple processes can be running at one time)

The main differences between the two is that processes have shared memory space but threads do not

* + Use cases for explicit multithreading

In general, we use explicit multithreading when we want absolute control and fine tuning of our program.

* + Advantages/disadvantages of explicit threading
    - Advantages:
      * Maximize Efficiency
      * Better control on synchronization (Using Locks for control)
      * Scheduling and load balancing (Improve control on balancing num of instructions per thread)
      * Enable use of special hardware (Dedicate threads to run on GPU)
    - Disadvantages:
      * Requires rewriting programs to effectively use multiple threads
      * Programmer must handle race conditions
      * Programmer must use threads appropriate to hardware
      * Programming language constructs for thread vary
      * These days, threads are considered low-level primitives

## **OpenMP**

* Requirements and limitations of OpenMP

I couldn’t find anything listed as a concrete requirement. The installation of the OpenMP Library or a compatible compiler (g++) is essential

Limitations:

* + - Not meant for distributed memory parallel systems
    - Not guaranteed to make the most efficient use of shared memory
    - Requires application program to handle certain tasks (Data Dependencies, Data Conflicts, Deadlocks, Synchronization of I/O)
    - Geared for C Programs (Can easily be mixed with C++ programs)

Components of OpenMP API

* + Compiler directives

Ex: #pragma omp (and any other clauses)

* + Runtime library routines

Ex: omp\_get\_num\_threads();

      omp\_get\_thread\_num();

* + Environment variables

Ex: NUM\_OF\_THREADS=1

* Syntax of OpenMP directives

#pragma omp (all other clauses)

* OpenMP parallel directive
  + When used

Following #pragma omp directive in an OpenMP program, parallelizes anything in the body among the specified number of threads

* + Definitions and uses for parallel clauses
    - if – syntax: #pragma omp parallel if (boolean\_expression)

Will only execute the pragma directive if the Boolean is met in runtime. Can be used to run a check prior to parallelization

* + - num\_threads – syntax: #pragma omp parallel num\_threads(8)
    - Specifies explicitly the number of threads to be used by the parallel body loop.
    - private, shared, default, firstprivate
      * private – syntax: #pragma omp parallel private(x)

If the value(s) in private came in defined, they will be undefined as soon as each copy enters the thread, when they exit, they will have the same value as when they entered.

* + - * shared – syntax: #pragma omp parallel shared(x)

This clause is used to share the same variable between multiple threads. We have to be careful when using shared variables to avoid race conditions

* + - * default – syntax: #pragma omp parallel default(shared)  
        The default clause specifies the default handling for variables. default(none) forces declaration of all variables statuses
      * firstprivate – syntax: #pragma omp parallel firstprivate(x)   
        Each thread gets it’s own copy of a variable, initialized to the starting value outside the scope of the parallel. When they exit, values don’t modify outside the scope
    - reduction
      * syntax – #pragma omp parallel for reduction(+:result)

Applies the operation prior to the colon to everything matching the variables.

* threadprivate variables and the parallel copyin clause

threadprivate variables will retain their values at the end of a section and there fore can be used in a subsequent parallel section (stored in heap memory, which holds the thread local storage)

copyin clause initializes threadprivate data upon entry to a parallel region.

* Using OpenMP
  + Compiling for OpenMP

export OMP\_NUM\_OF\_THREADS=4

g++ file.cpp -o file -fopenmp

* + Race conditions and how to avoid them

Any writes to variables (especially those that are shared) are not thread-safe and will create race conditions. Reads are generally safe, and reductions or atomics should be used in place of most standard conventions

* + Identifying errors in OpenMP code

#include “omp.h” is necessary for any operation like num\_threads();

* Measures of parallelism:  Definitions and how to compute
  + Parallel speedup
    - A measure of how much speed we gain by increasing the number of tasks we are using compared to the base speed.
    - Sn = Elapsed Time for 1 Thread / Elapsed Time for N threads
  + Parallel efficiency
    - A measure of the average OVERALL gain in speed per thread we add. Perfect efficiency is 100% gain per thread, which = 1.
    - En = Sn / Number of Threads (N)
  + Parallel cost
    - A measure of how much CPU we consume by adding N number of threads compared to the base case of using exactly one thread.
    - Pn = CPU Time for N threads/ CPU Time for 1 thread

## MPI

* Requirements and limitations of MPI

MPI makes use of many libraries to be compiled correctly. In order to prevent having to manually call these libraries, compiling with  
mpicxx or mpic++

will get the job done.

MPI is not limited by the shared memory space that OpenMP is. However, it is, like OpenMP, limited by the number of processors that you have access to (across many computers)

* Components of MPI API
  + Runtime library routines (e.g. MPI\_Send, MPI\_Recv, etc.)
    - MPI\_Init(int \* argc, char \*\* argv) -> Establishes links to the other MPI processes, all of which are initiated with another program (mpiexec), it also provides a acopy of the command line arguments to the other processes. It must be called only once
    - MPI\_Finalize() -> Shuts down the MPI library’s runtime infrastructure. Do not call it before Init.
    - MPI\_Comm\_size(MPI\_COMM comm, int \* size) -> where comm is likely MPI\_COMM\_WORLD, automatically created by MPI, determines number of eligible total MPI processes
    - MPI\_Comm\_rank(MPI\_COMM comm, int \* rank) -> determines the rank of the current process
    - MPI\_Send(void \*buf, int count, MPI\_Datatype datatype, int dest, int tag, MPI\_Comm comm) -> sends {count} number of items stored in {buf} of type {datatype} to the process with rank {dest}
    - MPI\_Recv(void \*buf, int count, MPI\_Datatype datatype, int source, int tag, MPI\_Comm comm, MPI\_Status \* status) -> receives {count} number of items stored in {buf} of type {datatype} to the process with rank {dest}  
      For either of these, MPI\_ANY\_TAG and for Recv MPI\_ANY\_SOURCE will work
    - MPI\_Status -> object contains MPI\_SOURCE, MPI\_TAG, MPI\_ERROR
    - MPI\_Bcast(void \*buf, int count, MPI\_Datatype datatype, int source, MPI\_Comm comm) sends/receives {count} number of items stored in {buf} of type {datatype} from the process with rank {source}
    - MPI\_Reduce(void \*sendbuf, void \*recvbuf, int count, MPI\_Datatype datatype, MPI\_Op op, int target, MPI\_Comm comm) -> SendBuf represents individual contribution from a task, recvbuf holds the total of everyone’s contribution, target is the rank of where the payload is being delivered, op is any of MPI’s operations (MPI\_SUM, MPI\_PROD)
    - MPI\_Allreduce(void \*sendbuf, void \*recvbuf, int count, MPI\_Datatype datatype, MPI\_Op op, MPI\_Comm comm) Same as before, but recvbuf contains the final value delivered to every rank
    - MPI\_Gather(const void \*sendbuf, int sendcount, MPI\_Datatype sendtype, void \*recvbuf, int recvcount, MPI\_Datatype recvtype, int root, MPI\_Comm comm) -> Used when different ranks have components of an array which need to be assembled into a single array on a single array. root is the rank of the process that gets the final result. send count should be equal to receive count
    - MPI\_Allgather(const void \*sendbuf, int sendcount, MPI\_Datatype sendtype, void \*recvbuf, int recvcount, MPI\_Datatype recvtype, MPI\_Comm comm) -> Same as before, but everyone gets a copy of the recvbuf
    - MPI\_Probe(int source, int tag, MPI\_Comm comm, MPI\_Status \*status) -> Gives information but doesn’t read the message
    - MPI\_Scatter(const void \*sendbuf, int sendcount, MPI\_Datatype sendtype, void \*recvbuf, int recvcount, MPI\_Datatype recvtype, int root, MPI\_Comm comm) -> Sends data from one task to all tasks in a group, where root is the one task
    - MPI\_Comm\_split(MPI\_Comm comm, int color, int key, MPI\_Comm \*newcomm) -> Create a new communicator based on colors and keys. comm is usually MPI\_COMM\_WORLD, color is a nonnegative int that controls the assignment of rank in the new communicator, key controls rank assignment, newcomm allows us to save the new communicator
    - MPI\_Comm\_free(MPI\_Comm &comm) -> Marks the object for deallocation
    - MPI\_Comm\_group(MPI\_Comm comm, MPI\_Group \*group) -> returns the group associated with a communicator
    - MPI\_Group\_incl(MPI\_Group group, int n, const int ranks[], MPI\_Group \*newgroup) -> produces a group by reordering an existing group and taking only listed members:

N is the number of members of the new group, ranks is an array listing the ranks of the members of a new group, &newgroup is a pointer to the new group

* + - MPI\_Comm\_create\_group(MPI\_Comm comm, MPI\_Group group, int tag, MPI\_Comm \*newcomm) -> creates a new communicator based on a group:  
      comm is the communicator of the larger group (typically MPI\_COMM\_WORLD), group is the subgroup used to build the new communicator, tag is used to distinguish between multiple calls, &newcomm pointer to a new communicator
    - MPI\_Group\_free(MPI\_Group &group) -> Deallocate a group
  + Communications in MPI
    - Latency and bandwidth
      * Latency – Amount of time it takes for initial communication to get from its source to its destination
      * Bandwidth – Once a path is established, the amount of data per second that can be transmitted
    - Blocking versus non-blocking
      * Blocking calls like MPI\_Ssend or MPI\_Srecv prevent further execution until communication is finished, they ensure it is safe to overwrite sent data, and ensure that data has arrived and is ready for use
      * Nonblocking calls, like MPI\_Isend and MPI\_Irecv only start the communication, they do not wait for it to complete. MPI\_Wait and MPI\_Test can be used to aid in this process (Wait waits for an MPI send/receive to complete, MPI\_Test tests the completion of a specific send or receive)
    - Buffered versus synchronous versus ready
      * Buffered communications – The data to be communicated is written to a local system buffer, transmitted and received in a local system buffer, then read
      * Synchronous communications – The send does not complete until a matching receive has begun
      * Ready communications – The user asserts that a matching receive has been posted, problems can arise if this is not the case
    - Point-to-point versus collective
      * Point to point communication – Rank of destination process is fixed, receiving process receives messages from the one process at a time, messages may be sent and received using blocking or non-blocking operations, each message has a tag. Only a pair of processes is involved in any one operation, logic involved in processing is performed by user’s program
      * Collective communication – One or more processes are simultaneously involved in a single send or receive . BCast and Reduce are examples of this. All collective processes use a communicator, the same function with the same parameters, aand often do not use a barrier.
  + Utility programs (e.g. mpirun, mpiexec, mpicxx)  
    mpicxx -> MPI compiler

mpirun -> Run an MPI compiled program

mpiexec -> Used to initiate programs on all non-master processes

* Using MPI
  + Compiling for MPI and running MPI programs

mpicxx or mpic++

then

mprun -np {number of processes} ./executable

* + Timing MPI programs
    - For elapsed time: Manager does  
      double start = MPI\_Wtime()

……

double elapsed = MPI\_Wtime() – start;

* + - For CPU time, on each thread

clock\_t t1 = clock();

…worker worker worker

t1 = clock() – t1;

CPU = static\_cast<double> (t1) / static\_cast<double> (CLOCKS\_PER\_SEC);

MPI\_Reduce(&CPU, &totalCPU, 1, MPI\_DOUBLE, MPI\_SUM, 0, MPI\_COMM\_WORLD);

* + Deadlocks and how to avoid them

Deadlocks occur when a blocking receiving is waiting on a send that will never occur, or when a blocking send is waiting on a receive that will never occur. These can be avoided by the use of non-blocking calls, or more easily, the programmer being diligent about when they are sending and receive.

* + Identifying errors in MPI code
  + Requirements for running MPI programs on multiple systems
    - Executable Code has to exist on every system
    - Every pair of processors can talk to each other
    - Specify a host file on the MPI command line
  + Questions on the demos performed during Session 26
    - Black magic using all the computer in BEN002 lab, Hello World, Fourier, etc. all done across every single computer in the lab
* GPU Computing
  + What is a GPU?
    - Graphics Processing Unit, primary purpose is for graphical output on a computer. However , they can be used to offload tasks if we already have something running on our CPU/want to run on our CPU and GPU at the same time
  + What is CUDA?
    - Used to be Compute Unified Device Architecture, but NVIDIA kept the tag and dropped the meaning. It provides compilers and utility programs to build code for GPUs. It also provides an API that enables communication between a host CPU and a device (GPU), occurring natively without function/method calls
  + Blocks and threads
    - NVIDIA GPUs can run up to 1024 threads In parallel. Threads are organized into 1, 2, or 3 dimensional grids of blocks. Each group of threads in a block is organized into warps.   
      Threads in a block run code in parallel (a warp) and each warp is typically a multiple of 32 threads
  + Anatomy of a CUDA program
    - C and C++ programs are ocmpiled with nvcc, these programs typically use the .cu file extensions. All Blocks are presented with the same code, but they can differentiate work by thread number. CUDA also makes use of kernels, which are methods run on the GPU device. Kernels are defined as the \_\_global\_\_ void type
    - cudaMalloc() -> Host allocates memory on device
    - cudaMemcpy() -> Host copies data to device
    - \_\_syncthreads() -> Do a computation using blocks of tasks with threads allocated to each block
    - kernelName<<blocks, threads>>(args); -> activate the kernel call with blocks number of blocks and threads number of threads
    - cudaMemcpy() -> Host copies data back from device
    - cudaFree() -> Host frees up allocated memory on the device
    - basic program  
      #include <iostream>

// CUDA "hello world" kernel

\_\_global\_\_ void sayHello() {

printf("Hello from block %d, thread %d\n",blockIdx.x, threadIdx.x);

}

// Main method for this program

int main(int argc, char \*argv[]) {

// Invoke the sayHello kernel with 1 block and 6 threads

sayHello<<<1, 6>>>();

// Synchronization is required to get printf output

cudaError\_t cudaerr = cudaDeviceSynchronize();

// Exit!

exit(0);

* + - cudaGetDeviceProperties() -> Provides > 70 properties of a device
    - threadIdx.x -> thread index
    - blockIdx.x -> block index
  + Questions on the demos performed during Session 27
    - Hello World and Fourier again, but with a warp size of >32 threads, Alan is not aware of why it levels off in performance.