# Exam 2: CSE 443/543 Spring 2020

Exam 2 takes place on Monday, April 13 during normal class time. It will be administered via Canvas and will use Proctorio for proctoring the exam. A quiz to test your Proctorio setup will be administered in class on Wednesday, April 8.

This exam is closed book and no notes are permitted. You cannot look up information online during the exam. You will need to perform simple mathematics and you can use the built-in calculator from Proctorio for this.

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.

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

## 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

* + Running parallel jobs

Not really sure what he means here, #pragma omp parallel covers this

* 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