University of California, Berkeley
Masters in Molecular Simulation and Software Engineering
Online Degree

Chemistry 281, 3 Units Software Engineering for Scientific Computing
Fall 2021.

Prof. Teresa Head-Gordon
274 Stanley Hall
thg@berkeley.edu.

Course description: The course covers computer architecture and software features that have the greatest impact on performance. It addresses debugging and performance tunning, detecting memory and stack overwrites, malloc corruption, hotspot, paging, cache misses. A toolbox with common algorithms: sorting, searching, hashing, trees, graph traversing, is followed by common patterns used in object-oriented design. It describes programming paradigms, dynamic libraries, distributed architectures, and services. Lectures on linear algebra and performance libraries are provided as background for future courses. HPC paradigms and GPU programming are introduced. Software packaging, extensibility, and interactivity is followed by team development, testing and hardening.

Contributions of this course to the broader curricular objectives: The objective of this recurrent course is to equip students with the skills and tools every software engineer must master for a successful professional career.

Course format: Three 50-minute lectures of faculty led, web-based asynchronous instruction per week plus 1 hour of web-based, synchronous discussion and 1 hour of GSI-led, web-based, synchronous lab to complete the course in 15 weeks. There will be five reading assignments that will expand on the lecture material. All students will be required to participate in the discussions. GSIs will go over homework assignments and practice exercises that are quantitative and provide guidance as necessary. The students are expected to have four hours of outside work for a total of nine hours per week.

Resources
Introduction to Algorithms, third Edition, Cormen, Leiserson, Rivest, Stein
Software Engineering: Theory and Practice, Shari Lawrence Pfleeger
Computer Architecture Fourth Edition, Hennesey and Patterson
Numerical Linear Algebra, Trefethen and Bau
CUDA Application Design and Development, Rob Farber
Debugging with GDB: The GNU Source-Level Debugger, Stallman, Pesch, Shebs

Reading List
Weeks 1 and 2
Intro to Linux
https://ryanstutorials.net/linuxtutorial/
https://tldp.org/LDP/intro-linux/intro-linux.pdf Chapters 2-6

https://tldp.org/HOWTO/pdf/Emacs-Beginner-HOWTO.pdf Optional
http://www.jesshamrick.com/2012/09/10/absolute-beginners-guide-to-emacs/ Optional

Week 3

GDB
http://www.unknownroad.com/rtfm/gdbtut/gdbtoc.html
https://www.techbeamers.com/how-to-use-gdb-top-debugging-tips/
https://www.youtube.com/watch?v=bWH-nL7v5F4 Debugging - GDB Tutorial (Optional)

Weeks 4 and 5
What Every Programmer Should Know About Memory
https://people.freebsd.org/~lstweart/articles/cpumemory.pdf
Chapter 3 CPU Caches
Chapter 4 Virtual Memory

Week 6
Essence of Linear Algebra
https://www.3blue1brown.com/essence-of-linear-algebra-page
Vector and Matrix Operations
https://inst.eecs.berkeley.edu/~ee16a/fa19/lecture/Notes2A.pdf
https://inst.eecs.berkeley.edu/~ee16a/fa19/lecture/Notes2B.pdf

Week 7
Eigenvalues and Eigenvectors
https://inst.eecs.berkeley.edu/~ee16a/fa19/lecture/Notes9.pdf

Week 8
Intro to parallel programming
Sections A-D
https://hpc.llnl.gov/training/tutorials/introduction-to-parallel-computing-tutorial

Week 9
What Every Programmer Should Know About Memory
https://people.freebsd.org/~lstweart/articles/cpumemory.pdf
Chapter 6.5 Numa Programming

Grading: There will be 6 programming assignments due every other week and a final project plus weekly discussions on the provided reading material and homework assignments and weekly quizzes. Students will receive a letter grade based on the following breakup: quizzes: 10%, discussions: 20% , programming assignments: 30%, final project 40%.

Quizzes are given during the video lectures. They are intended to test the students' understanding. The quizzes will be in the form of multiple-choice questions. They will be automatically graded.

The discussion part will be graded according to the degree of participation of the students.

Programming assignments are given on a weekly or bi-weekly basis depending on the complexity. Each assignment has multiple parts. Each part has a different point value depending on the complexity. The students will upload their program. They will be tested for correctness by running those programs on data not seen by the students.

In the final project the students will implement an application to solve a practical problem. The students will have to design the input and data model, choose the algorithm and decide how to present the results. They will also design a testing protocol. The grade is based on the level of completeness of the project and the quality of the design.

Prerequisites: the students will have had MSSE courses (1) C275A Intro to Programming, (2) C275B Software Best Practices. Students are expected to be familiar with programming in C++ and have a basic understanding of LINUX. Additional materials will be provided for students to peruse as necessary.

Course Requirements
Each student is required to view all of the online lectures, do all the online quizzes and submit all homework assignments. There will be a suggested reading lists and a mandatory reading list to complement the lectures. The students are expected to peruse the mandatory reading lists. Discussions among students and GSIs are encouraged. A discussion board will be available. A laptop/workstation is required. The instructor will provide a Linux-based VM with all the required software preinstalled. The student will have to install the free version of VMWare client or VirtualBox. Access to NERSC computers will be provided as necessary. Installation of the free Moba Xterm or Xming utilities on the user's computer is highly recommended.

Office hours: The instructor will be available 1 hour per week for one on one consultation by appointment. The instructor will also be available for synchronous open class discussion one hour per week. These synchronous office hours will be posted on the course website. The GSIs will be available 5 hours per week.

Learning Objectives for this course
Upon completion of this course the students will have the basic skills necessary for developing high performance computing software.

Detailed Course Syllabus

Week 1: Introduction to Computers I - Basic Concepts.
Binary representation. Understanding Finite Precision, Floating Point Operations. Roundoff. Example: Loss of orthogonality in the Conjugate Gradient Operation. Brief description of the architecture of modern processors. Registers, basic operations, branch (miss) predictions, loads and stores.

Week 2: Introduction to Computers II - Memory - SIMD parallelism (Vectorization)
Memory hierarchies, Caches. The importance of memory locality. Memory Bandwidth. Virtual Memory, the TLB cache. Paging. Cache Coherence, Strong Coherence, Weak Coherence, coherence protocols: MESI. Modern Processors Architecture, MultiSocket, NUMA, Buses, Network Cards, Communication Fabrics, I/O. Main architectures: x86-64: Intel, AMD, RISC: ARM

Week 3: Software structure - Compilers, Linkers, Libraries
Overview of Programming models, execution models, procedural, object oriented, and functional. Componetization. Compilation and linking. Creating and using dynamic shared libraries. Dynamic linking and loading. The GNU C library, Interacting with the Operating System. Signals.

Week 4: Introduction to Debugging and Machine Level Concepts

Debugging demonstration. Basic debugging: examining variables, code correctness. Advanced debugging: finding memory overwrites, setting watch points. Debugging optimized and deoptimized code. Examining memory. Understanding memory faults. Using memory fencing. Stepping. Low level debugging. Developing intuition. Memory Leaks.
Live demo: Debugging with GDB and Emacs

Week 5: Algorithms and data structures - The Software Engineer toolbox
Data structures, searching, sorting, trees, graph algorithms, hashing, divide and conquer, recursion, etc. Introduction to asymptotic analysis, algorithmic complexity, NP hard, NP complete.

Week 6: Common Patterns - Patterns in Python and STL
Overview of common object-oriented patterns: Singleton, Factory, Decorator, Module, Proxy, Iterator, Mediator, Compute kernel, Event-based, Join, Monitor, Thread Pool, Thread-specific storage, Lazy initialization, etc.

Week 7: Advanced Topics. Interprocess Communication
Client-server architecture. Daemons. Services. Sockets.

Week 8: Intro to Scientific Computing - Numerically Intensive kernels
Dense Linear Algebra, Sparse Linear Algebra. Common sparse storage formats. Linear Solvers, Condition Number, Numerical Issues. Why pivoting is needed. Iterative Solvers, Convergence. Brief introduction to Finite Difference and Finite Elements.
High performance libraries, OpenBlas: dgemm, dgemv, dtrsm, and others.

Week 9: Parallel programming
Models: Shared Memory, NUMA, multi-threading, openmp, distributed memory, message passing, MPI. Race conditions, starvation, deadlocks, partitioning, load balancing, scalability. The Map-Reduce paradigm. Tips for debugging parallel programs.

Week 10: GPUs
Understanding GPU architecture. Similarities and differences with CPUs. SIMD programming model. Writing a simple CUDA kernel. Allocating Device Memory. Transferring memory. --Libraries: cuBlas, cuSparse. Compiling cuda kernels. Debugging CUDA code.
Other Options, OpenCL, HIP, oneAPI.

Week 11: Performance Tuning
Simple Performance tuning: finding Hotspots. Advanced Performance analysis: Roofline performance Models. Microarchitecture Utilization. Memory Analysis, cache misses, page faults.
Live demo: profiling with vtune.
-
Week 12: Containers - Software Deployment
Packaging your software. Deploying your software. Advantages of containers.
Increased portability, consistency. Disadvantages. Live demo: creating a Docker container.

Week 13: Mixed-language programming
Mixing Fortran, C, and C++. Using the best of both worlds: connecting Python to C/Fortran. An application of dynamic shared libraries.

Week 14: The Design-Testing-Regression Cycle. Software Security
Team Development. Code Repositories. Branching and versioning. Writing Specifications. Defining Interfaces. Encapsulation. Component testing Code Invariants. Correctness. Updating

changes. Regression testing. Performance regression Testing. Coverage testing. Detecting code that is never executed by your tests. Negative Testing. Developing software with security in mind.