COSC 420: High Perf. Computing

Dr. Joe Anderson
Course Policies

● See web page for most info: resources, schedule, due dates, etc.

● Highlights of course structure
  – Largely project/program based
  – Two midterms, one final
  • May not be standard exam format, TBD
Strategies for success

- University policy: 1 credit = 4 hours work outside lecture
- Keep documentation open and nearby
- Talk to friends, others in class. Hang out in the lab
- Care for your physical being! Eat, sleep, and rest appropriately.
- Spaced repetition is shown to be vastly superior to “cramming”
Bonus point opportunities

- You may receive up to 5% total bonus on your final grade by participating in voluntary professional development and community service. Examples include:
  - Gull Week (Sept 9 – 19)
  - GullCode (MATH/COSC club)
  - Game Jam (SU Indies club)
  - Volunteer work
  - See me for further approval...
Course Goals

- Recap of the focus of COSC 320
  - Design advanced algorithms and structures
  - Prove asymptotic complexity, ignore “constant” and “small” overhead
- Contrast with this course
  - Still working with advanced DS and Algorithms
  - Now, be concerned with reducing the extra computational overhead, possibly with specialized hardware
Course structure

• Assignments will largely consist of 5-7 short term projects
  – ... but there will still be some math to do and turn in
  – There will be some hardware components, system configuration, etc.

• Minimal programming guidance given in class, mostly theory and problem solving techniques

• Presentations are likely, TBD
Programming Tools

- With a focus on implementation, focus on using specific software environment/toolchain
  - C language (yay!) with some Python later
- Still working on a GNU/Linux environment
  - Be familiar with command-line tools!
- Will eventually use multi-node setups to run software
High-Performance Tools

- Distributed memory
  - Multiple independent processes on multiple compute nodes (but which commonly communicate or use shared memory storage)
  - MPI Library – message passing interface
    - Allows one to easily run programs in parallel with communication constructs
      - Not always free/cheap!
    - Also has bindings for Fortran and Python (C++ bindings are deprecated)
High-Performance Tools

- Shared memory
  - Multiple programs that share the same memory pool during execution
    - No communication overhead!
    - Need to manage race conditions (ouch)
    - Executable can be run in a standard way
  - OpenMP (multiprocessing) library
    - Provided through compiler tools, not separate library routines
  - MPI can also support shared memory (later)
Getting Started: Using MPI

- Install the “openmpi” and “openmpi-devel” packages
- You may need to add the paths to mpicc and mpiexec to your “PATH” environment variable
int main() {
    ....
    printf("Hello world\n");
    ....
}

int main() {
    ....
    printf("Hello world\n");
    ....
}

int main() {
    ....
    printf("Hello world\n");
    ....
}

int main() {
    ....
    printf("Hello world\n");
    ....
}
“Hello, world!”

```c
#include<stdio.h>

int main(){
    printf(“Hello, world!
”);
    return 0;
}

# Run program with: mpiexec -n 5 ./a.out
What happened?

• **Single Program Multiple Data (SPMD)**
  - One executable gets copied between multiple nodes (-n 5)
  - Each node runs the *exact same* program!
    • This means, if you want different nodes to behave differently, we will need some tools for introspection
  
• We won’t worry about the analogous MPMD structure
Using MPI F’real

• Compile with MPI library
  – #include<mpi.h>
  – Use `mpicc` or `gcc -I/path/to/mpi.h`

• MUST CALL
  – int MPI_Init(int *argc, int ***argv)
  – int MPI_Finalize()
Other Important Tools

- **Type:** MPI_Comm
  - Represents the “MPI communicator”
- **int MPI_Abort(MPI_Comm, int)**
  - Aborts and returns an error code
- **Object:** MPI_INFO_ENV
  - key/value pairs for `mpiexec` options
More tools

- int MPI_Get_processor_name(char *name, int *resultlen)
  - Places name in the provided buffer
  - Length of name in second param
  - “name” param must be length at least MPI_MAX_PROCESSOR_NAME
Managing Processes

• The group of processes is the “communicator”
  – MPI_COMM_WORLD is a predefined communicator, but you can have more

• MPI_Comm_size() – total size of the comm
• MPI_Comm_rank() – the rank of the current process within the communicator
• Both above take MPI_Comm and int*
  – Result stored in the second param
More MPI Interface

- MPI_Get_processor_name – retrieves the node name
  - Takes char* to store and int* for the length
  - MPI_MAX_PROCESSOR_NAME is a constant defining the largest name possible
Using these tools for parallelism

- We want to “distribute” the labor of a task
- But, according to the above, every process is identical!
- ... not really, they each know their own rank
- So we use the rank to determine who does which work
Lab Task 1

• (See posted instructions for more detail)
• Given a (probably large) number, N, determine if it is prime or composite
  – Brute force, for now
  – How? Discuss!
• Once we know how many tasks must be done, we can use MPI to distribute them evenly across the nodes
Hardware Parallelism

- Take into account various hardware architectures
  - CPU
  - FPU
    - Takes heavy advantage of “pipeline” methodology
  - GPU
Pipelines

- Focus on floating point operations (FLOPS)
- Stages of a floating point operation
  - Decode instruction, find data locations
  - Fetch data into registers
  - Align exponents
  - Do the operation
  - Normalize result
  - Store
Pipelines

- Supposing each stage has dedicated hardware
- The second instruction can begin decoding once the first one has begun fetching
- The third can be decoded while the second is fetched and first is aligned
- Etc...
Pipeline speedup

- Total time for \( n \) operations without pipeline is
  - \( t(n) = n \times l \times t \)
  - Number of instructions is \( n \)
  - Number of stages is \( l \)
  - Cycle time is \( t \)
- We say the “rate” is \( n / t(n) \)
  - So without pipeline, \( 1 / (l \times t) \)
- With pipeline, \( t(n) = ? \)
Pipeline speedup

- With pipeline, \( t(n) = (s + l + n-1)*t \)
  - \( S \) is the “startup time” of the pipeline to distribute data as necessary
  - Written sometimes as \( t(n) = [n + n_{1/2}]\cdot t \)

- Consider limit of the rate for each as \( n \) goes to infinity
  - With pipeline, has a dependence on \( n \)
  - Becomes “\( l \) times” faster!

- To get to this case, consider the rate with \( n^{1/2} \) instructions...
Pipeline in action

- Consider vectors/arrays $a$, $b$, $c$ and the two following loops:
  - For(i)
    $$a[i] = b[i] + c[i]$$
  - For(i)
    $$a[i+1] = a[i]*b[i] + c[i]$$
- Can we speed up the second, despite dependence?
Other hardware parallelism

- Multiple-issue: independent instructions that can happen at the same time
- Branch prediction: compiler can “guess” which branch of a conditional will happen
- Out-of-order execution: instructions rearranged by compiler
- Prefetching: speculatively request data before it is explicitly used
Memory Hierarchy

• Main considerations
  – Location, distance from CPU/FPU/GPU
  – Buses: wires that transfer data between different memories
  – Latency
  – Bandwidth
Memory Hierarchy

Latency from next level (cycles)

- 230-360
- 26
- 12
- 4

Size (bytes)

- 2G
- 2M
- 256k
- 32k
- 192

Registers
L1 cache
L2 cache
L3 cache
Main memory