Outline

- Announcements:
  - HW II Idue Friday!
- Validating Model Problem
- Software performance
- Measuring performance
- Improving performance

Validating Model Problem

- Our solution C is only an approximation of the true solution
- The accuracy of the approximation will depend on
  - dt
  - dx
  - "smoothness" of the initial conditions
- Two things to watch for:
  - lambda <1→solution will "blow up" if lambda>1
  - decreasing dx will make solutions more accurate
    - Try at coarse (m=10) and fine resolution (m=100)

Software Performance

- Factors influencing speed
  - Hardware
    - Clock speed, memory size/speed, bus speed
  - Algorithmic scaling
    - What happens as n gets big?
  - Interaction between algorithm and hardware
    - Compiler optimizations vs. hand tuning
Computer Architecture

- Von Neumann described a memory hierarchy
  - Fast→slow
  - $$$→cheap
  - Register L1 L2 RAM Disk

Architecture and Performance

- Avoid using the disk!
  - Minimize reading and writing
  - Buy more RAM so you don’t use virtual memory
- Buy a system with lots of fast cache
- Buy a system with lots of RAM
- Then, if you have money left, buy a fast chip

Algorithm Performance

- There are often several algorithms for solving the same problem, each with its own performance characteristics
- Much of computer science is concerned with finding the fastest algorithm for a given problem
- Typically, we’re interested in how an algorithm scales
  - How it performs as the problem size increases
  - To determine this, we need an accounting system (or model) of how a program runs
  - The simplest model assumes all commands take the same amount of time, so we just need to count commands
  - We could use more complicated models that weight commands
Algorithmic Performance of Linear Search

- Linear Search
  - Inputs: integer array x of length n, value k
  - Output: j s.t. x[j] = k, or j = -9 if k not in x
  ```
  int LinSearch(int x[], int n, int k){
    j=0;
    while(x[j] != k & j < n){
      j=j+1;
    }
    if(j == n){
      return(-9);
    }
    return(j);
  }
  ```

Algorithmic Performance of Binary Search

- Binary Search
  - Inputs: SORTED integer array x of length n, value k, integers st, en
  - Output: j s.t. x[j] = k, or j = -9 if k not in x[st:en]
  ```
  int BinSearch(int x[], int st, int en, int k){
    int mid, ans;
    mid = (en-st)/2 + mod(en-st, 2); // middle of array
    if(mid == st) {
      if(x[mid] == k) {
        return(mid);
      }
      else {
        ans = BinSearch(x, st, (st+mid-1), k); // search on right
      }
    }
    else {
      ans = BinSearch(x, st, (st+mid), k); // search on left
    }
    return(ans);
  }
  ```

Comparing Linear and Binary Search

- Linear Search
  - max n iterations through loop
  - will work on any array
- Binary Search
  - max log₂(n) recursions (log₂(n) < n)
  - faster if x is already sorted
  - but sorting takes approx. n² steps
- So, if you have to perform < n searches, use linear search
- If you have to perform > n searches, sort first, then use BinSearch
Interation between algorithm and hardware

- There are several ways to implement the same algorithm
- Their performance could be very different, if they get compiled in different ways.
- The differences are due to interactions with the memory hierarchy

Rules of thumb

- Minimize computation (precomputing)
  \[ x1 = \frac{-b + \sqrt{b^2 - 2ac}}{2a}; \]
  \[ x2 = \frac{-b - \sqrt{b^2 - 2ac}}{2a}; \]
  - better than \[ x1 = \frac{-b + \sqrt{b^2 - 2ac}}{2a}; \] etc.
- Minimize division
  \[ \text{overdx} = \frac{1}{\text{dx}}; \]
  \[ \text{overdx}^2 = \frac{1}{\text{dx}^2}; \]
- Minimize function/subroutine calls (inlining)
  - There is overhead associated with calling functions
  - This goes against good programming which encourages modularity

Rules of Thumb

- Avoid big jumps in memory
  - Data is moved from RAM to cache in chunks
  - Accessing arrays sequentially maximizes cache-reuse
  - Special implication for 2 (or higher) D arrays
  - Memory is sequential:
    - Row Major: C, C++, Java, Matlab
      ```
      for(j=0;j<m;j++){
        for(k=0;k<n;k++){
          A[j][k]=…
        }
      }
      ```
    - Column Major: FORTRAN
      ```
      do k=1,3 do j=1,2      A[j,k]=…  
      enddo enddo
      ```
Improving performance

- To improve algorithmic performance
  - Take more CS!
- To improve interaction with hardware
  - Check out compiler optimizations
  - Then, start hand-tuning the code

Compiler Optimization

- A good compiler can take care of lots of things automatically
  - Some precomputing
  - Some inlining (for small functions)
  - Other things like loop unrolling:

```c
for(j=0;j<100;j++)
    for(k=0;k<20;k++)
        A[j][k]=...
```

```c
for(j=0;j<100;j++)
    for(k=0;k<20;k+=4)
        A[j][k]=...
        A[j][k+1]=...
        A[j][k+2]=...
        A[j][k+3]=...
```

Measuring Performance

- Before we start messing with working code, we should identify where code is slow
- This is called profiling
  - Goal is to identify bottlenecks—places in the code that limit performance
- We can use profiling tools like prof (gprof) or insert timing calls
- Important to check performance on problems of different sizes
My Advice

• Before you do anything, make sure your code works
  • well-tuned incorrect code is still incorrect
  • It is better to solve your problem slowly than not at all!
  • Look for algorithmic improvements
  • Try compiler options
    • Read your compiler's manual to learn about what they do
  • Last but not least, try hand tuning