Defining Floating Point Arithmetic

- Representable numbers
  - Scientific notation: \( r \cdot d.d \ldots d \times e^{\text{exp}} \)
  - sign bit +/-
  - radix \( r \) (usually 2 or 10, sometimes 16)
  - significand \( d.d.d \) (how many base-\( r \) digits \( d \)?)
  - exponent \( \text{exp} \) (range?)
  - others?

- Operations:
  - arithmetic: +, -, *, /...
  - how to round result to fit in format
  - comparison (<, =, >)
  - conversion between different formats
  - short to long FP numbers, FP to integer
  - exception handling
    - what to do for 0/0, 2*largest_number, etc.
    - binary/decimal conversion
      - for I/O, when radix not 10

Language/library support for these operations

---

IEEE Floating Point Arithmetic Standard 754 (1985) - Normalized Numbers

- Normalized Nonzero Representable Numbers: +-1.d.d.d x 2**exp
  - sign bit, nonzero significand, maximum exponent
  - fills in gap between zero and underflow threshold UN
  - Overflow Exception
    - occurs when exact nonzero result is too large to represent accurately
    - Ex: 2**OV
  - Divide by zero Exception
    - return +- infinity
    - sign of zero important!
  - Also return +- infinity for
    - 0/0, 2**-infinitly, infinity/infinity, infinity*infinity
    - Result is exact, not an exception!

- Zero: +-0.d.d.d x 2**min_exp
  - Why bother with -0 later

- Why bother with -0 later

IEEE Floating Point Arithmetic Standard 754 - “Denorms”

- Denormalized Numbers: +-0.d.d.d x 2**(-inf)
  - sign bit, nonzero significand, minimum exponent
  - Fills in gap between UN and 0
  - Underflow Exception
    - occurs when exact nonzero result is less than underflow threshold UN
    - Ex: UN/3
    - return a denorm, or zero
  - Why bother?
    - Necessary so that following code never divides by zero
    - if (a != b) then x = a/(a-b)

IEEE Floating Point Arithmetic Standard 754 - +- Infinity

- +- Infinity: Sign bit, zero significand, maximum exponent
  - Overflow Exception
    - occurs when exact finite result too large to represent accurately
    - Ex: 2**OV
  - Divide by zero Exception
    - return +- infinity
    - sign of zero important!
  - Also return +- infinity for
    - 0/0, 2**-infinity, infinity/infinity, infinity*infinity
    - Result is exact, not an exception!

IEEE Floating Point Arithmetic Standard 754 - NAN (Not A Number)

- NAN: Sign bit, nonzero significand, maximum exponent
  - Invalid Exception
    - occurs when exact result not a well-defined real number
    - 0/0
    - sqrt(-1)
    - infinity/infinity, infinity/infinity, 0*infinity
  - NAN + 3
  - Return a NAN in all these cases
  - Two kinds of NANs
    - Quiet - propagates without raising an exception
    - Signaling - generate an exception when touched
    - good for detecting uninitialized data
Error Analysis

- **Basic error formula**
  \[ f(a \text{ op } b) = (a \text{ op } b) \times (1 + d) \]
  where op one of +,-,\*,/
  \[ |d| \ll \text{macheps} \]
  assuming no overflow, underflow, or divide by zero
- **Example:** adding 4 numbers
  \[ f(a_1 + a_2 + a_3 + a_4) = \left( (a_1 + a_2) \times (1 + d_1) + a_3 \right) \times (1 + d_2) + a_4 \]
  where each \(|e_i| \ll 3 \times \text{macheps} \)
  
  - Backward Error Analysis - algorithm called numerically stable if it gives the exact result for slightly changed inputs
  - Numerical Stability is an algorithm design goal

Backward error

- Approximate solution is exact solution to modified problem.
- How large a modification to original problem is required to give result actually obtained?
- How much data error in initial input would be required to explain all the error in computed results?
- Approximate solution is good if it is exact solution to "nearby" problem.

Sensitivity and Conditioning

- Problem is insensitive or well conditioned if relative change in input causes commensurate relative change in solution.
- Problem is sensitive or ill-conditioned, if relative change in solution can be much larger than that in input data.

\[
\text{Cond} = \frac{\text{Relative change in solution}}{\text{Relative change in input data}}
= \frac{\left| (f(x) - f(x + h)) / f(x) \right|}{\left| (x - x + h) / x \right|}
\]
- Problem is sensitive, or ill-conditioned, if cond \gg 1.
- When function f is evaluated for approximate input \(x' = x + h\) instead of true input value of x.
- Absolute error \( = f(x + h) - f(x) \approx h \cdot f'(x) \approx \infty \)
- Relative error \( = (f(x + h) - f(x)) / f(x) \approx h \cdot f'(x) / f(x) \)

Sensitivity: 2 Examples

- **cos(\pi/2) and 2-d System of Equations**
- Consider problem of computing cosine function for arguments near \(\pi/2\).
- Let \(x = \pi/2\) and let \(h\) be small perturbation to \(x\). Then

\[
\text{Abs: } |f(x + h) - f(x)| = |h \cdot \sin(x)| \approx |h| \sin(x)
\]

\[
\text{Rel: } \left| \frac{f(x + h) - f(x)}{f(x)} \right| = \left| \frac{h \cdot \sin(x)}{f(x)} \right| \approx \left| \frac{|h|}{f(x)} \cdot \sin(x) \right| \approx \frac{|h|}{f(x)} \cdot 1
\]

So small change in \(x\) near \(\pi/2\) causes large change in output (relative change in \(\cos(x)\) regardless of method used.
- \(\text{cos}(1.57079) = 0.63267949 \times 10^{-5}\)
- \(\text{cos}(1.57078) = 1.64267949 \times 10^{-5}\)
- Relative change in output is a quarter million times greater than relative change in input.

Sensitivity: 2 Examples

- Consider problem of computing cosine function for arguments near \(\pi/2\).
- Let \(x = \pi/2\) and let \(h\) be small perturbation to \(x\). Then

\[
\text{absolute error} = \cos(x + h) - \cos(x) = -h \cdot \sin(x) \approx -h \cdot 1
\]

\[
\text{relative error} = \left| \frac{\cos(x + h) - \cos(x)}{\cos(x)} \right| \approx \left| \frac{-h \cdot 1}{\cos(x)} \right| \approx \frac{-h}{\cos(x)} \approx \frac{h}{\cos(x)}
\]

So small change in \(x\) near \(\pi/2\) causes large relative change in \(\cos(x)\) regardless of method used.
- \(\cos(1.57079) = 0.63267949 \times 10^{-5}\)
- \(\cos(1.57078) = 1.64267949 \times 10^{-5}\)
- Relative change in output is a quarter million times greater than relative change in input.

Example: Polynomial Evaluation

Using Horner’s Rule

- **Horner’s rule to evaluate** \(p = \sum_{k=0}^{n} c_k \cdot x^k\)
  \(p = c_n\), for \(k=n-1\) down to \(0\), \(p = x \cdot p + c_k\)

- **Numerically Stable**
- **Apply to** \((x-2)^9 = x^9 - 18x^8 + ... - 512\)
- **Evaluated around 2**

```
begin
p = c_0;
for k = n-1 to 0 by -1 do
    p = p \times x + c_k;
end { for }
HonerPoly \leftarrow p;
end { HonerPoly }
```
Example: polynomial evaluation (continued)

\[(x-2)^9 = x^9 - 18x^8 + \ldots - 512\]

We can compute error bounds using

\[f(a \oplus b) = (a \oplus b) \times (1 + d)\]

### Exception Handling

- What happens when the "exact value" is not a real number, or too small or too large to represent accurately?
- 5 Exceptions:
  - Overflow - exact result > OV, too large to represent
  - Underflow - exact result nonzero and < UN, too small to represent
  - Divide-by-zero - nonzero/0
  - Invalid - 0/0, sqrt(-1), ...
  - Inexact - you made a rounding error (very common!)
- Possible responses
  - Stop with error message (unfriendly, not default)
  - Keep computing (default, but how?)

### Summary of Values Representable in IEEE FP

- +/- Zero
- Normalized nonzero numbers
- Denormalized numbers
- +/-Infinity
- NaNs
  - Signaling and quiet
  - Many systems have only quiet

### Hazards of Parallel and Heterogeneous Computing

- What new bugs arise in parallel floating point programs?
- Ex 1: Nonrepeatability
  - Makes debugging hard!
- Ex 2: Different exception handling
  - Can cause programs to hang
- Ex 3: Different rounding (even on IEEE FP machines)
  - Can cause hanging, or wrong results with no warning
- See [www.netlib.org/lapack/lawns/lawn112.ps](http://www.netlib.org/lapack/lawns/lawn112.ps)
- IBM RS6K and Java

### Types of Parallel Computers

- The simplest and most useful way to classify modern parallel computers is by their memory model:
  - shared memory
  - distributed memory

### Standard Uniprocessor Memory Hierarchy

- Intel Pentium 4
  - 2.8GHz processor
- P7 Prescott 478
  - 8 Kbytes of 4 way assoc. L1 instruction cache with 32 byte lines.
  - 8 Kbytes of 4 way assoc. L1 data cache with 32 byte lines.
  - 256 Kbytes of 8 way assoc. L2 cache, 32 byte lines.
  - 400 MB/s bus speed.
  - SSE2 provide peak of 4 Gflop/s.
Shared Memory / Local Memory

- Usually think in terms of the hardware
- What about a software model?
- How about something that works like cache?
- Logically shared memory

Parallel Programming Models

- Control
  - how is parallelism created
  - what orderings exist between operations
  - how do different threads of control synchronize?
- Naming
  - what data is private vs. shared
  - how logically shared data is accessed or communicated
- Set of operations
  - what are the basic operations
  - what operations are considered to be atomic
- Cost
  - how do we account for the cost of each of the above

Trivial Example \[ \sum_{i=0}^{n-1} f(A[i]) \]

- parallel decomposition:
  - each evaluation and each partial sum is a task
  - assign n/p numbers to each of p procs
    - each computes independent “private” results and partial sum
    - one (or all) collects the p partial sums and computes the global sum

- classes of data
  - logically shared
    - the original n numbers, the global sum
  - logically private
    - the individual function evaluations
    - what about the individual partial sums?

Model 1

- A shared memory machine
- Processors all connected to a large shared memory
- “Local” memory is (usually) not part of the hardware
  - see DEC slide “SMP” (symmetric multiprocessors) in Millennium, SGI Origin
- Cost: much cheaper to cache than main memory
  - machine model 1a: A shared address space machine
    - replace caches by local memories (in abstract machine model)
    - this affects the cost model as repeatedly accessed data should be copied
      - Cray T3E

Programming Model 1

- shared address space
  - program consists of a collection of threads of control
  - each with a set of private variables
    - e.g., local variables on the stack
    - collectively with a set of shared variables
      - e.g., static variables, shared common blocks, global heap
  - threads communicate implicitly by writing and reading shared variables
  - threads coordinate explicitly by synchronization operations on shared variables
    - writing and reading flags
    - locks, semaphores
  - like concurrent programming on uniprocessor

Shared Memory code for computing a sum

Thread 1

\[ [s = 0 \text{ initially}] \]
\[ \text{for } i = 0, n/2-1 \]
\[ \text{local}\_s1 = 0 \]
\[ \text{local}\_s2 = 0 \]
\[ \text{local}\_s1 = \text{local}\_s1 + f(A[i]) \]
\[ s = s + \text{local}\_s1 \]

Thread 2

\[ [s = 0 \text{ initially}] \]
\[ \text{for } i = n/2, n-1 \]
\[ \text{local}\_s2 = 0 \]
\[ \text{local}\_s2 = \text{local}\_s2 + f(A[i]) \]
\[ s = s + \text{local}\_s2 \]

What could go wrong?
Pitfall and solution via synchronization

° Pitfall in computing a global sum $s = local_s1 + local_s2$

Thread 1 (initially $s=0$)
- load $s$ [from mem to reg]
- $s = s + local_s1$ [in reg]
- store $s$ [from reg to mem]

Thread 2 (initially $s=0$)
- load $s$ [from mem to reg; initially 0]
- $s = s + local_s2$ [in reg]
- store $s$ [from reg to mem]

° Instructions from different threads can be interleaved arbitrarily

° What can final result $s$ stored in memory be?

° Race Condition

° Possible solution: Mutual Exclusion with Locks

Thread 1
- lock
- load $s$ [from mem to reg]
- $s = s + local_s1$ [in reg]
- store $s$ [from reg to mem]
- unlock

Thread 2
- lock
- load $s$ [from mem to reg]
- $s = s + local_s2$ [in reg]
- store $s$ [from reg to mem]
- unlock

° Locks must be atomic (execute completely without interruption)

Programming Model 2

° Message Passing
- program consists of a collection of named processes
  - thread of control plus local address space
  - local variables, static variables, common blocks, heap
  - processes communicate by explicit data transfers
  - matching pair of send & receive by source and dest. proc.
  - coordination is implicit in every communication event
  - logically shared data is partitioned over local processes

° Like distributed programming

° Program with standard libraries: MPI, PVM

Model 2

° A distributed memory machine
  - Cray T3E, IBM SP2, Clusters
  - Processors all connected to own memory (and caches)
  - cannot directly access another processor's memory

° Each "node" has a network interface (NI)
- all communication and synchronization done through this

° First possible solution

Processor 1
- send $x_{local}$, proc2
  - $x_{local} = x[1]$
- receive $x_{remote}$, proc2
  - $s = x_{local} + x_{remote}$

Processor 2
- send $x_{local}$, proc1
  - $x_{local} = x[2]$
- receive $x_{remote}$, proc1
  - $s = x_{local} + x_{remote}$

° Second possible solution - what could go wrong?

Processor 1
- send $x_{local}$, proc2
  - $x_{local} = x[1]$
- receive $x_{remote}$, proc2
  - $s = x_{local} + x_{remote}$

Processor 2
- send $x_{local}$, proc1
  - $x_{local} = x[2]$
- receive $x_{remote}$, proc1
  - $s = x_{local} + x_{remote}$

° What if send/receive act like the telephone system? The post office?

Programming Model 3

° Data Parallel
  - Single sequential thread of control consisting of parallel operations
  - Parallel operations applied to all (or defined subset) of a data structure
  - Communication is implicit in parallel operators and "shifted" data structures
  - Elegant and easy to understand and reason about
  - Not all problems fit this model

° Like marching in a regiment

A = array of all data
$fa = f(A)$
$s = \text{sum}(fa)$

° Think of Matlab

Model 3

° Vector Computing
  - One instruction executed across all the data in a pipelined fashion
  - Parallel operations applied to all (or defined subset) of a data structure
  - Communication is implicit in parallel operators and "shifted" data structures
  - Elegant and easy to understand and reason about
  - Not all problems fit this model

° Like marching in a regiment

A = array of all data
$fa = f(A)$
$s = \text{sum}(fa)$

° Think of Matlab
Model 3

- An SIMD (Single Instruction Multiple Data) machine
- A large number of small processors
  - A single "control processor" issues each instruction
  - Each processor executes the same instruction
  - Some processors may be turned off on any instruction

Machines not popular (CM2), but programming model is
  - Implemented by mapping n-fold parallelism to p processors
  - Mostly done in the compilers (HPF = High Performance Fortran)

Model 4

- Since small shared memory machines (SMPs) are
  the fastest commodity machine, why not build a
  larger machine by connecting many of them with a
  network?
- CLUMP = Cluster of SMPs
- Shared memory within one SMP, message passing
  outside
- Clusters, ASCI Red (Intel), ...

Programming Model 5

- Bulk Synchronous Processing (BSP) – L. Valiant
- Used within the message passing or shared memory
  models as a programming convention
- Phases separated by global barriers
  - Compute phases: all operate on local data (in distributed
    memory)
    - or read access to global data (in shared memory)
  - Communication phases: all participate in rearrangement or
    reduction of global data
- Generally all doing the "same thing" in a phase
  - All do f, but may all do different things within f
- Simplicity of data parallelism without restrictions

Summary so far

- Historically, each parallel machine was unique,
  along with its programming model and programming
  language
- You had to throw away your software and start
  over with each new kind of machine - ugh
- Now we distinguish the programming model from
  the underlying machine, so we can write portably
  correct code, that runs on many machines
- MPI now the most portable option, but can be tedious
- Writing portably fast code requires tuning for the
  architecture
  - Algorithm design challenge is to make this process easy
  - Example: picking a blocksize, not rewriting whole
    algorithm