

CS 594 Spring 2003

Lecture 3:

Overview of High-Performance Computing

Jack Dongarra
 Computer Science Department
 University of Tennessee

A Little History

- ◆ Von Neumann and Goldstine - 1947
 - "Can't expect to solve most big [$n > 15$] linear systems without carrying many decimal digits [$d > 8$], otherwise the computed answer would be completely inaccurate." - WRONG!
- ◆ Turing - 1949
 - "Carrying d digits is equivalent to changing the input data in the d -th place and then solving $Ax=b$. So if A is only known to d digits, the answer is as accurate as the data deserves."
 - Backward Error Analysis
- ◆ Rediscovered in 1961 by Wilkinson and publicized
- ◆ Starting in the 1960s- many papers doing backward error analysis of various algorithms
- ◆ Many years where each machine did FP arithmetic slightly differently
 - Both rounding and exception handling differed
 - Hard to write portable and reliable software
 - Motivated search for industry-wide standard, beginning late 1970s
 - First implementation: Intel 8087
- ◆ ACM Turing Award 1989 to W. Kahan for design of the IEEE Floating Point Standards 754 (binary) and 854 (decimal)
 - Nearly universally implemented in general purpose machines

Defining Floating Point Arithmetic

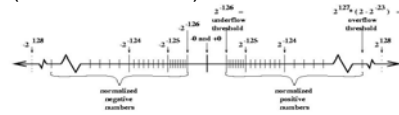
- ◆ Representable numbers
 - Scientific notation: $\pm/- d.d\dots d \times r^{exp}$
 - sign bit $\pm/-$
 - radix r (usually 2 or 10, sometimes 16)
 - significant $d.d\dots d$ (how many base- r digits d ?)
 - exponent exp (range?)
 - others?
- ◆ Operations:
 - arithmetic: $+, -, \times, /, \dots$
 - how 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, $2^{\text{largest_number}}$, 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: $\pm/- 1.d\dots d \times 2^{exp}$
 - Macheps = Machine epsilon = $2^{-\# \text{significant bits}}$ = relative error in each operation
 - OV = overflow threshold = largest number
 - UN = underflow threshold = smallest number

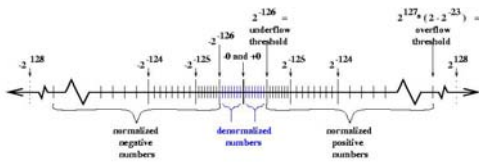
Format	# bits	#significant bits	macheps	#exponent bits	exponent range
Single	32	23+1	2^{-24} ($\sim 10^{-7}$)	8	$2^{-126} - 2^{127}$ ($\sim 10^{-38}$)
Double	64	52+1	2^{-53} ($\sim 10^{-16}$)	11	$2^{-1022} - 2^{1023}$ ($\sim 10^{-308}$)
Double	≥ 80	≥ 64	$\leq 2^{-64}$ ($\sim 10^{-19}$)	≥ 15	$2^{-16382} - 2^{16383}$ ($\sim 10^{-4932}$)

Extended (80 bits on all Intel machines)



IEEE Floating Point Arithmetic Standard 754 - "Denorms"

- ◆ Denormalized Numbers: $\pm/- 0.d\dots d \times 2^{\text{min_exp}}$
 - 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 $(a \neq b)$ 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}
 - return +/- infinity
- ◆ Divide by zero Exception
 - return +/- infinity = $1/\pm 0$
 - sign of zero important!
- ◆ Also return +/- infinity for
 - $3 \pm \text{infinity}$, 2^{infinity} , $\text{infinity}^{\text{infinity}}$
 - 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
 - > 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

7

Error Analysis

- ◆ **Basic error formula**
 - > $f(a \text{ op } b) = (a \text{ op } b)^*(1 + d)$ where
 - » op one of +, -, *, /
 - » $|d| \leq \text{macheps}$
 - » assuming no overflow, underflow, or divide by zero
- ◆ **Example: adding 4 numbers**
 - > $f(x_1 + x_2 + x_3 + x_4) = \{((x_1 + x_2)^*(1 + d_1) + x_3)^*(1 + d_2) + x_4\}^*(1 + d_3)$

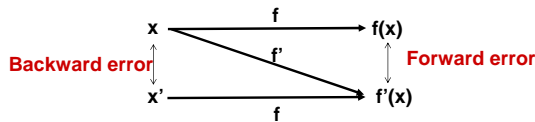
$$= x_1^*(1 + d_1)^*(1 + d_2)^*(1 + d_3) + x_2^*(1 + d_1)^*(1 + d_2)^*(1 + d_3) + x_3^*(1 + d_2)^*(1 + d_3) + x_4^*(1 + d_3)$$

$$= x_1^*(1 + e_1) + x_2^*(1 + e_2) + x_3^*(1 + e_3) + x_4^*(1 + e_4)$$
 where each $|e_i| \ll 3^* \text{macheps}$
 - > get exact sum of slightly changed summands $x_i^*(1 + e_i)$
 - > **Backward Error Analysis** - algorithm called numerically stable if it gives the exact result for slightly changed inputs
 - > Numerical Stability is an algorithm design goal

8

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.



9

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.

$$\text{Cond} = \frac{|\text{Relative change in solution}|}{|\text{Relative change in input data}|} = \frac{|f(x) - f(x')/f(x)|}{|(x' - x)/x|}$$

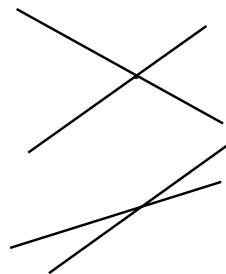
- ◆ Problem is sensitive, or ill-conditioned, if $\text{cond} \gg 1$.
- ◆ When function f is evaluated for approximate input $x' = x + h$ instead of true input value of x .
- ◆ Absolute error = $f(x + h) - f(x) \approx h f'(x)$
- ◆ Relative error = $[f(x + h) - f(x)] / f(x) \approx h f'(x) / f(x)$

10

Sensitivity: 2 Examples

cos($\pi/2$) and 2-d System of Equations

- ◆ Consider problem of computing cosine function for arguments near $\pi/2$.
- ◆ Let $x \approx \pi/2$ and let h be small perturbation to x . Then
 - absolute error = $\cos(x+h) - \cos(x) \approx -h \sin(x) \approx -h$,
 - relative error $\approx -h \tan(x) \approx \infty$
- ◆ 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.

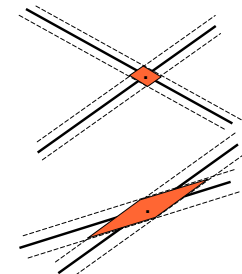


11

Sensitivity: 2 Examples

cos($\pi/2$) and 2-d System of Equations

- ◆ Consider problem of computing cosine function for arguments near $\pi/2$.
- ◆ Let $x \approx \pi/2$ and let h be small perturbation to x . Then
 - absolute error = $\cos(x+h) - \cos(x) \approx -h \sin(x) \approx -h$,
 - relative error $\approx -h \tan(x) \approx \infty$
- ◆ 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.



12

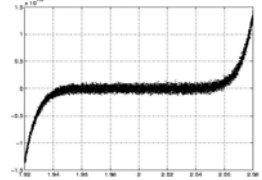
Example: Polynomial Evaluation Using Horner's Rule

- ◆ **Horner's rule to evaluate** $p = \sum_{k=0}^{n-1} c_k * x^k$
 - $p = c_n$, for $k=n-1$ down to 0, $p = x*p + c_k$
- ◆ **Numerically Stable**
- ◆ **Apply to** $(x-2)^9 = x^9 - 18*x^8 + \dots - 512$
 $-2^9 + x*(2^8 - x*(2^7 + \dots))$
- ◆ **Evaluated around 2**

```

begin
  p := c[n];
  for k := n-1 to 0 by -1 do
    p := p*x + c[k]
  end { for }
  HornerPoly := p;
end { HornerPoly }

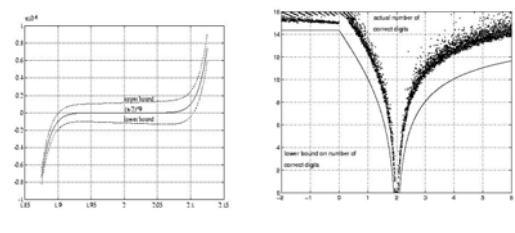
```



13

Example: polynomial evaluation (continued)

- ◆ $(x-2)^9 = x^9 - 18*x^8 + \dots - 512$
- ◆ **We can compute error bounds using**
 - $f(a \text{ op } b) = (a \text{ op } b) * (1+d)$



14

What happens when the "exact value" is not a real number, or is too small or too large to represent accurately?

You get an "exception"

15

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?)

16

Summary of Values Representable in IEEE FP

- ◆ **+- Zero**

+	0...0	0.....0
---	-------	---------
- ◆ **Normalized nonzero numbers**

+	Not 0 or all-1s	anything
---	-----------------	----------
- ◆ **Denormalized numbers**

+	0...0	nonzero
---	-------	---------
- ◆ **+-Infinity**

+	1...1	0.....0
---	-------	---------
- ◆ **NANs**

+	1...1	nonzero
---	-------	---------

 - Signaling and quiet
 - Many systems have only quiet

17

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
- ◆ **IBM RS6K and Java**

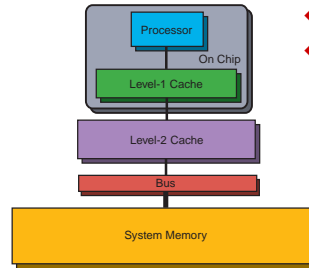
18

Types of Parallel Computers

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

19

Standard Uniprocessor Memory Hierarchy



- ◆ Intel Pentium 4 2 GHz 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

20

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

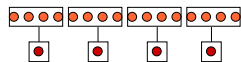
21

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

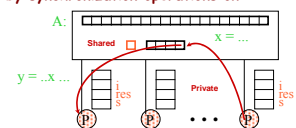


=> Classes of Data

- ◆ Logically Shared
 - the original n numbers, the global sum
- ◆ Logically Private
 - the individual function evaluations
 - what about the individual partial sums?

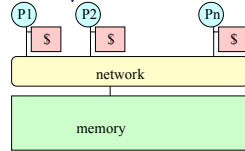
Programming Model 1

- ◆ 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



Model 1

- ◆ A shared memory machine
- ◆ Processors all connected to a large shared memory
- ◆ "Local" memory is not (usually) part of the hardware
 - Sun, DEC, Intel "SMPs" (Symmetric multiprocessors) in Millennium: SGZ 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 -- repeatedly accessed data should be copied
 - Cray T3E

25

Shared Memory code for computing a sum

Thread 1

```
[s = 0 initially]
local_s1 = 0
for i = 0, n/2-1
    local_s1 = local_s1 + f(A[i])
s = s + local_s1
```

Thread 2

```
[s = 0 initially]
local_s2 = 0
for i = n/2, n-1
    local_s2 = local_s2 + f(A[i])
s = s + local_s2
```

What could go wrong?

26

Pitfall and solution via synchronization

- Pitfall in computing a global sum $s = \text{local_s1} + \text{local_s2}$

<p>Thread 1 (initially $s=0$)</p> <pre>load s [from mem to reg] s = s + local_s1 [=local_s1, in reg] store s [from reg to mem]</pre>	<p>Thread 2 (initially $s=0$)</p> <pre>load s [from mem to reg; initially 0] s = s + local_s2 [=local_s2, in reg] store s [from reg to mem]</pre>
---	--

Time

- 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

<p>Thread 1</p> <pre>lock load s s = s + local_s1 store s unlock</pre>	<p>Thread 2</p> <pre>lock load s s = s + local_s2 store s unlock</pre>
--	--

- Locks must be atomic (execute completely without interruption)

27

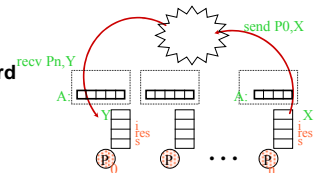
Programming Model 2

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

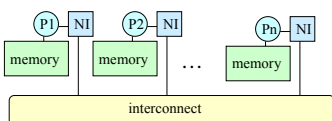
- ◆ Like distributed programming

- 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



29

Computing $s = x(1) + x(2)$ on each processor

- First possible solution

Processor 1
 send xlocal, proc2
 [xlocal = x(1)]
 receive xremote, proc2
 s = xlocal + xremote

Processor 2
 receive xremote, proc1
 send xlocal, proc1
 [xlocal = x(2)]
 s = xlocal + xremote

- Second possible solution - what could go wrong?

Processor 1
 send xlocal, proc2
 [xlocal = x(1)]
 receive xremote, proc2
 s = xlocal + xremote

Processor 2
 send xlocal, proc1
 [xlocal = x(2)]
 receive xremote, proc1
 s = xlocal + xremote

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

30

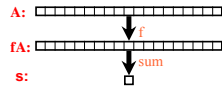
Programming Model 3

◆ 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

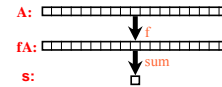
Model 3

◆ 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

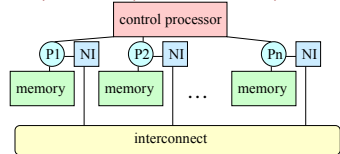
A = array of all data
 $fA = f(A)$
 $s = \text{sum}(fA)$



◦ 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?

- Treat machine as "flat", always use message passing, even within SMP (simple, but ignore important part of memory hierarchy)
- Expose two layers: shared memory (OpenMP) and message passing (MPI) higher performance, but ugly to program

34

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

36

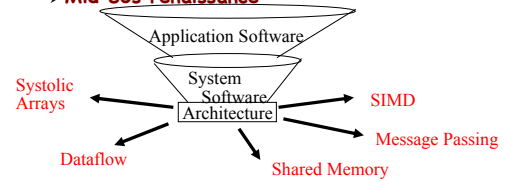
Recap

- ◆ **Parallel Comp. Architecture driven by familiar technological and economic forces**
 - application/platform cycle, but focused on the most demanding applications
 - hardware/software learning curve
 - ◆ **More attractive than ever because 'best' building block - the microprocessor - is also the fastest BB.**
 - ◆ **History of microprocessor architecture is parallelism**
 - translates area and density into performance
 - ◆ **The Future is higher levels of parallelism**
 - Parallel Architecture concepts apply at many levels
 - Communication also on exponential curve
- => **Quantitative Engineering approach**

37

History

- ◆ **Parallel architectures tied closely to programming models**
 - Divergent architectures, with no predictable pattern of growth.
 - **Mid 80s renaissance**



38

Programming Model

- ◆ **Conceptualization of the machine that programmer uses in coding applications**
 - How parts cooperate and coordinate their activities
 - Specifies communication and synchronization operations
- ◆ **Multiprogramming**
 - Independent jobs, no communication or synch. at program level
- ◆ **Shared address space**
 - like bulletin board
- ◆ **Message passing**
 - like letters or phone calls, explicit point to point
- ◆ **Data parallel:**
 - more regimented, global actions on data
 - Implemented with shared address space or message passing

39

Economics

- ◆ **Commodity microprocessors not only fast but CHEAP**
 - Development costs tens of millions of dollars
 - BUT, many more are sold compared to supercomputers
 - Crucial to take advantage of the investment, and use the commodity building block
- ◆ **Multiprocessors being pushed by software vendors (e.g. database) as well as hardware vendors**
- ◆ **Standardization makes small, bus-based SMPs commodity**
- ◆ **Desktop: few smaller processors versus one larger one?**
- ◆ **Multiprocessor on a chip?**

40