CS 594 Spring 2003
Lecture 4:
Overview of High-Performance Computing

Jack Dongarra
Computer Science Department
University of Tennessee

Plan for Today

- Finish up last weeks slides
- Talk about computer arithmetic
- Begin Cache discussion
- Go over homework

High Performance Computers

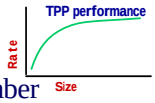
- ~ 20 years ago
  - $1x10^6$ Floating Point Ops/sec (Mflop/s)
    - » Scalar based
- ~ 10 years ago
  - $1x10^9$ Floating Point Ops/sec (Gflop/s)
    - » Vector & Shared memory computing, bandwidth aware
    - » Block partitioned, latency tolerant
- ~ Today
  - $1x10^{12}$ Floating Point Ops/sec (Tflop/s)
    - » Highly parallel, distributed processing, message passing, network based
    - » data decomposition, communication/computation
- ~ 10 years away
  - $1x10^{15}$ Floating Point Ops/sec (Pflop/s)
    - » Many more levels MH, combination/grids&HPC
    - » More adaptive, LT and bandwidth aware, fault tolerant, extended precision, attention to SMP nodes

Linpack (100x100) Analysis

- Compaq 386/SX20 SX with FPA - .16 Mflop/s
- Pentium IV – 2.53 GHz – 1190 Mflop/s
- 12 years → factor of ~ 7500 (Doubling in less than a year, for 12 years)
- How
  - Clock = 126x
  - External Bus Width & Caching –
    - » 16 vs. 64 bits = 4x
  - Floating Point -
    - » 4/8 bits multi vs. 64 bits (1 clock) = 8x
  - Compiler Technology = 2x
- Potential for the 2.53 GHz Pentium 4 is 5.6 Gflop/s!

Top 500 Computers

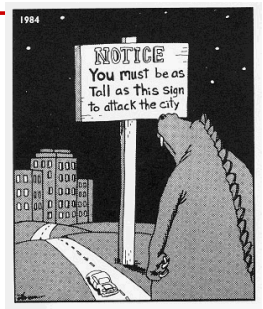
- Listing of the 500 most powerful Computers in the World
- Yardstick: Rmax from LINPACK MPP
  $Ax=b$, *dense problem*
  Updated twice a year
  SC'xy in the States in November
  Meeting in Mannheim, Germany in June

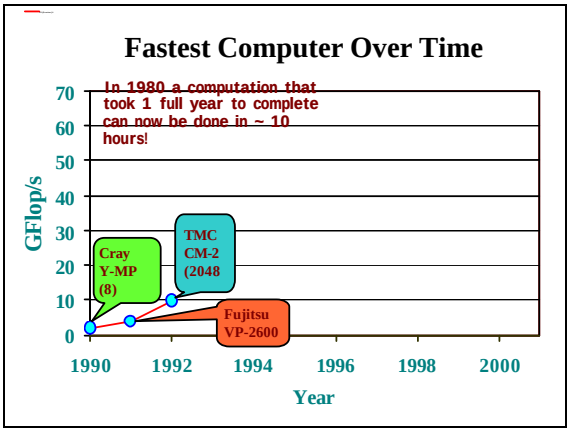


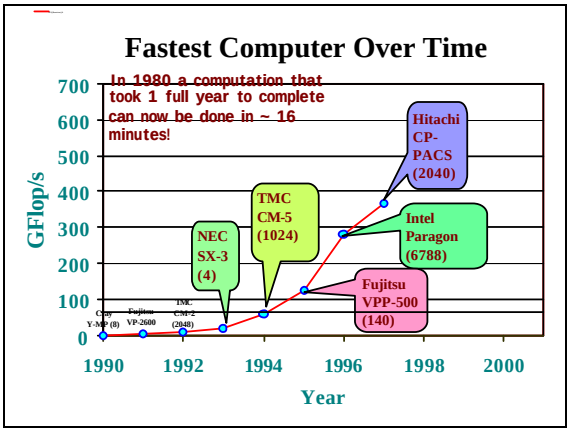
10 Year for Top500 and 25 Year for Linpack Benchmark

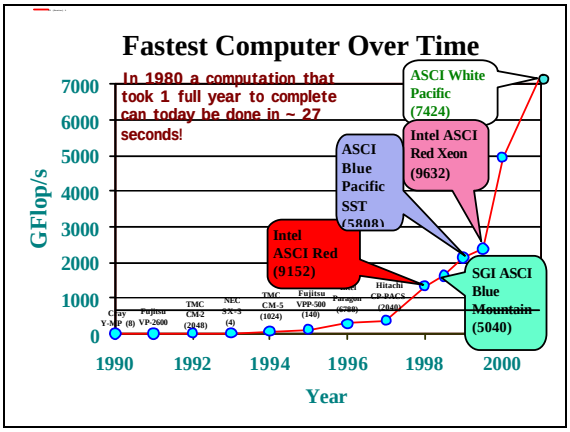
Big Means What?

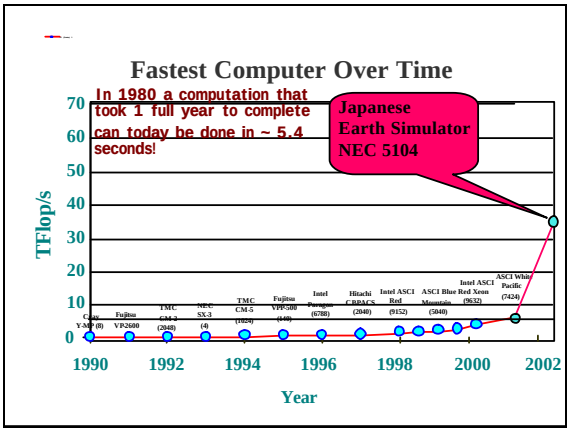
- Over the last 10 years the range for the Top500 has increased greater than Moore's Law
- 1993:
  - #1 = 59.7 GFlop/s
  - #500 = 422 MFlop/s
- 2003:
  - #1 = 35.8 TFlop/s
  - #500 = 196 GFlop/s











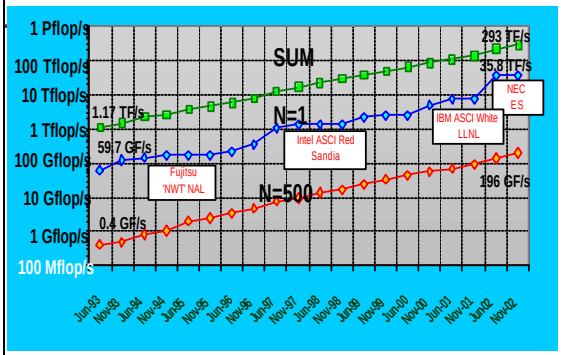
20th List: The TOP10

Rank	Manufacturer	Computer	R_{max} [TF/s]	Installation Site	Country	Year	Area of Installation	# Proc
1	NEC	Earth-Simulator	35.86	Earth Simulator Center	Japan	2002	Research	5120
2	HP	ASCI Q, AlphaServer SC	7.73	Los Alamos National Laboratory	USA	2002	Research	4096
2	HP	ASCI Q, AlphaServer SC	7.73	Los Alamos National Laboratory	USA	2002	Research	4096
4	IBM	ASCI White SP Power3	7.23	Lawrence Livermore National Laboratory	USA	2000	Research	8192
5	Linux NetworX	MCR Cluster	5.69	Lawrence Livermore National Laboratory	USA	2002	Research	8192
6	HP	AlphaServer SC ES45 1 GHz	4.46	Pittsburgh Supercomputing Center	USA	2001	Academic	3016
7	HP	AlphaServer SC ES45 1 GHz	3.98	Commissariat a l'Energie Atomique (CEA)	France	2001	Research	2560
8	HPTi	Xeon Cluster - Myrinet2000	3.34	Forecast Systems Laboratory - NOAA	USA	2002	Research	1536
9	IBM	pSeries 690 Turbo	3.16	HPCx	UK	2002	Academic	1280
10	IBM	pSeries 690 Turbo	3.16	NCAR (National Center for Atmospheric Research)	USA	2002	Research	1216

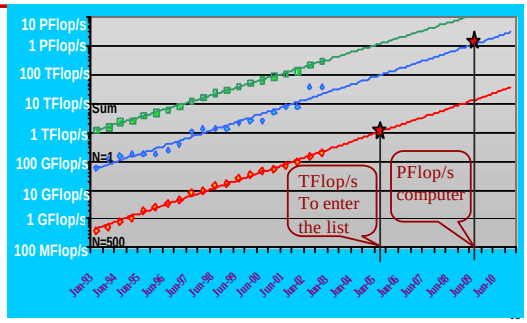
11

182 fell off; 500 was 318 in June

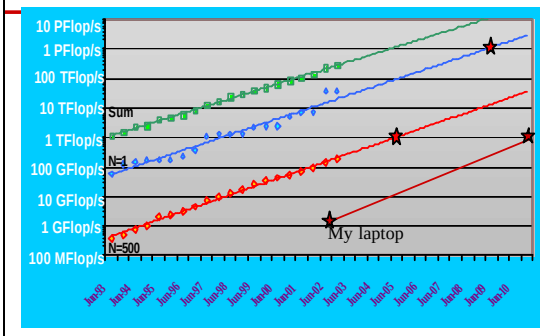
TOP500 - Performance



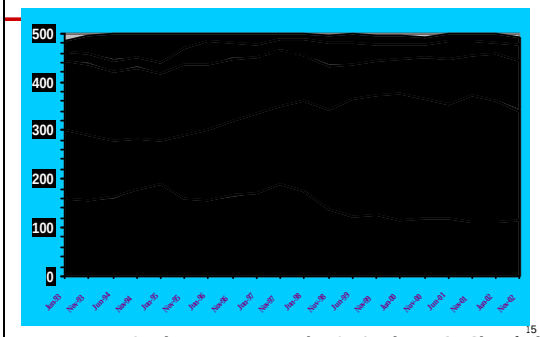
Performance Extrapolation



Performance Extrapolation



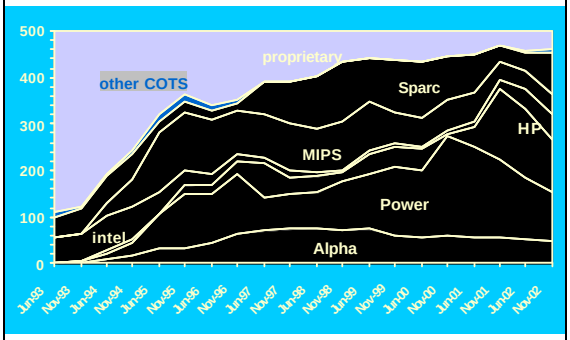
Customer Type



15

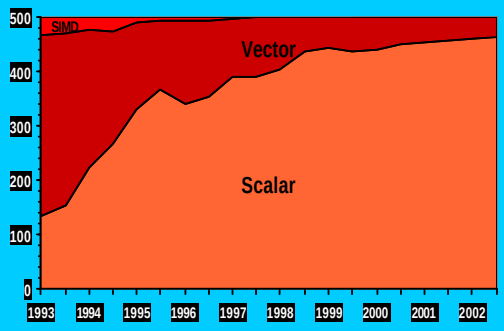
226 Industry, 115 Research, 102 Academic, 32 Classified

Chip Technology

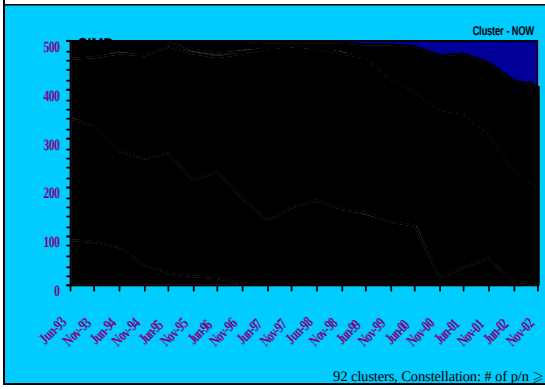


16

Processor Type

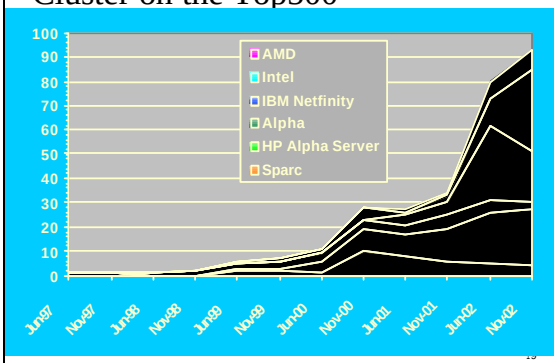


Architectures



92 clusters, Constellation: # of p/n ≥ n

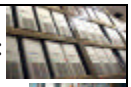
Cluster on the Top500



Top500 Conclusions

- **Microprocessor based supercomputers have brought a major change in accessibility and affordability.**
- **MPPs continue to account of more than half of all installed high-performance computers worldwide.**

High-Performance Computing Directions: Beowulf-class PC Clusters



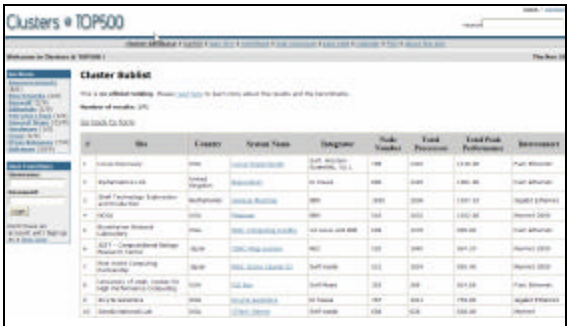
Definition:

- **COTS PC Nodes**
  - **Pentium, Alpha, PowerPC, SMP**
- **COTS LAN/SAN Interconnect**
  - **Ethernet, Myrinet, Giganet, ATM**
- **Open Source Unix**
  - **Linux, BSD**
- **Message Passing Computing**
  - **MPI, PVM**
  - **HPF**

Advantages:

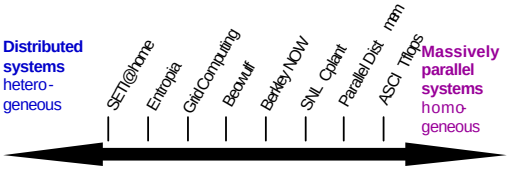
- **Best price-perfo**
- **Low entry-level cost**
- **Just-in-place configuration**
- **Vendor invulnerable**
- **Scalable**
- **Rapid technology tracking**

Enabled by PC hardware, networks and operating system achieving capabilities of scientific workstations at a fraction of the cost and availability of industry standard message passing libraries. However, much more of a contact sport.



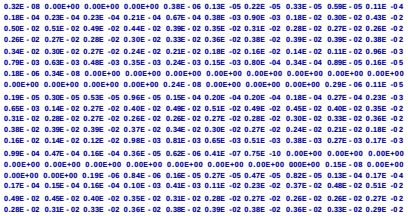
- **Peak performance**
- **Interconnection**
- **http://clusters.top500.org**
- **Benchmark results to follow in the coming months**

Distributed and Parallel Systems

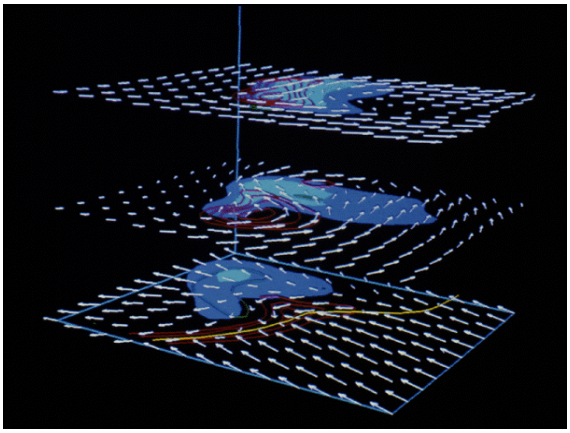


- **Gather (unused) resources**
- **Steal cycles**
- **System SW manages resources**
- **System SW adds value**
- **10% - 20% overhead is OK**
- **Resources drive applications**
- **Time to completion is not critical**
- **Time-shared**
- **Bounded set of resources**
- **Apps grow to consume all cycles**
- **Application manages resources**
- **System SW gets in the way**
- **5% overhead is maximum**
- **Apps drive purchase of equipment**
- **Real-time constraints**
- **Space-shared**

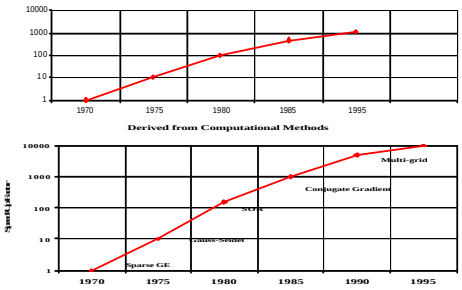
Virtual Environments



Do they make any sense?



Performance Improvements for Scientific Computing Problems



26

Highly Parallel Supercomputing: Where Are We?

- **Performance:**
  - Sustained performance has dramatically increased during the last year.
  - On most applications, sustained performance per dollar now exceeds that of conventional supercomputers. But...
  - Conventional systems are still faster on some applications.
- **Languages and compilers:**
  - Standardized, portable, high-level languages such as HPF, PVM and MPI are available. But ...
  - Initial HPF releases are not very efficient.
  - Message passing programming is tedious and hard to debug.
  - Programming difficulty remains a major obstacle to usage by mainstream scientist.

27

Highly Parallel Supercomputing: Where Are We?

- **Operating systems:**
  - Robustness and reliability are improving.
  - New system management tools improve system utilization. But...
  - Reliability still not as good as conventional systems.
- **I/O subsystems:**
  - New RAID disks, HiPPI interfaces, etc. provide substantially improved I/O performance. But...
  - I/O remains a bottleneck on some systems.

28

The Importance of Standards - Software

- Writing programs for MPP is hard ...
- But ... one-off efforts if written in a standard language
- Past lack of parallel programming standards ...
  - ... has restricted uptake of technology (to "enthusiasts")
  - ... reduced portability (over a range of current architectures and between future generations)
- Now standards exist: (PVM, MPI & HPF), which ...
  - ... allows users & manufacturers to protect software investment
  - ... encourage growth of a "third party" parallel software industry & parallel versions of widely used codes

29

The Importance of Standards - Hardware

- **Processors**
  - commodity RISC processors
- **Interconnects**
  - high bandwidth, low latency communications protocol
  - no de-facto standard yet (ATM, Fibre Channel, HPPI, FDDI)
- **Growing demand for total solution:**
  - robust hardware + usable software
- **HPC systems containing all the programming tools / environments / languages / libraries / applications packages found on desktops**

30

The Future of HPC

- The expense of being different is being replaced by the economics of being the same
- HPC needs to lose its "special purpose" tag
- Still has to bring about the promise of scalable general purpose computing ...
- ... but it is dangerous to ignore this technology
- Final success when MPP technology is embedded in desktop computing
- Yesterday's HPC is today's mainframe is tomorrow's workstation

31

Achieving TeraFlops

- In 1991, 1 Gflop/s
- 1000 fold increase
  - Architecture
    - » exploiting parallelism
  - Processor, communication, memory
    - » Moore's Law
  - Algorithm improvements
    - » block-partitioned algorithms

32

Future: Petaflops (10^{15} fl pt ops/s)

Today $\approx \sqrt{10^{15}}$ flops for our workstations

- A Pflop for 1 second ≈ a typical workstation computing for 1 year.
- From an algorithmic standpoint
  - concurrency
  - data locality
  - latency & sync
  - floating point accuracy
  - dynamic redistribution of workload
  - new language and constructs
  - role of numerical libraries
  - algorithm adaptation to hardware failure

33

Petaflop (10^{15} flop/s) Computers Within the Next Decade

- Five basis design points:
  - Conventional technologies
    - » 4.8 GHz processor, 8000 nodes, each w/16 processors
  - Processing-in-memory (PIM) designs
    - » Reduce memory access bottleneck
  - Superconducting processor technologies
    - » Digital superconductor technology, Rapid Single-Flux-Quantum (RSFQ) logic & hybrid technology multi-threaded (HTMT)
  - Special-purpose hardware designs
    - » Specific applications e.g. GRAPE Project in Japan for gravitational force computations
  - Schemes utilizing the aggregate computing power of processors distributed on the web
    - » SETI@home ~26 Tflop/s

34

Petaflops (10^{15} flop/s) Computer Today?

- 1 GHz processor (O(10^9) ops/s)
  - 1 Million PCs
  - $1B ($1K each)
  - 100 Mwatts
  - 5 acres
  - 1 Million Windows licenses!!
  - PC failure every second

35

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

36

Defining Floating Point Arithmetic

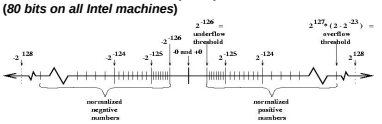
- Representable numbers
  - Scientific notation: +/ - d.d…d x r^exp
  - sign bit +/ -
  - radix r (usually 2 or 10, sometimes 16)
  - significand d.d…d (how many base-r digits d?)
  - exponent exp (range?)
  - others?
- Operations:
  - arithmetic: +,-,x,/,...
    - » how to round result to fit in format
  - 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 - Normalized Numbers

- Normalized Nonzero Representable Numbers: +- 1.d…d x 2^exp
  - Macheps = Machine epsilon = 2^-#significand bits = relative error in each operation
  - OV = overflow threshold = largest number
  - UN = underflow threshold = smallest number

Format	# bits	#significand bits	macheps	#exponent bits	exponent range
Single	32	23+1	2^{-24} (~10^{-7})	8	2^{-126} - 2^{127} (~10^{+-38})
Double	64	52+1	2^{-53} (~10^{-16})	11	2^{1022} - 2^{1023} (~10^{+308})
Double	>=80	>=64	<=2^{-64}(~10^{-19})	>=15	2^{16382} - 2^{16383} (~10^{+4932})

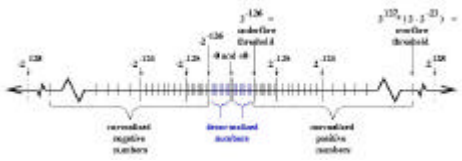
Extended (*80 bits on all Intel machines*)



IEEE Floating Point Arithmetic Standard 754 - "Denorms"

- Denormalized Numbers: +-0.d…d x 2^{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 != 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/+-0
  - sign of zero important!
- Also return +- infinity for
  - 3+infinity, 2*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
  - 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
  - fl(a op b) = (a op b)*(1 + d) where
    - » op one of +,-,*,/
    - » |d| <= macheps
    - » assuming no overflow, underflow, or divide by zero
- Example: adding 4 numbers
  - $fl(x_1+x_2+x_3+x_4) = \{[(x_1+x_2)*(1+d_1) + x_3]*(1+d_2) + x_4\}*(1+d_3)$

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

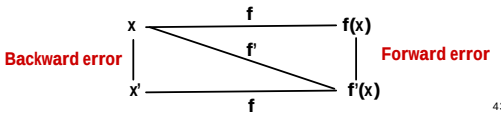
$$= x_1*(1+e_1) + x_2*(1+e_2) + x_3*(1+e_3) + x_4*(1+e_4)$$

where each $|e_i| <\sim 3*macheps$

  - 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

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.



43

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.**

Cond = |Relative change in solution| / |Relative change in input data|
= |[f(x) – f(x)]/f(x)| / |(x' – x)/x|

- Problem is sensitive, or ill-conditioned, if cond >> 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) ≈ h f'(x)
- Relative error =[f(x + h) – f(x)] / f(x) ≈ h f'(x) / f(x)

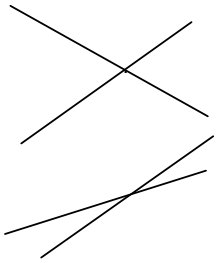
44

Sensitivity: 2 Examples cos(p/2) and 2-d System of Equations

- Consider problem of computing cosine function for arguments near p/2.
- Let x ≈ p/2 and let h be small perturbation to x. Then

absolute error = cos(x+h) – cos(x)
≈ -h sin(x) ≈ -h,
relative error ≈ -h tan(x) ≈ 8

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



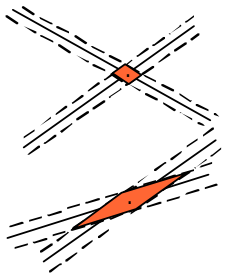
45

Sensitivity: 2 Examples cos(p/2) and 2-d System of Equations

- Consider problem of computing cosine function for arguments near p/2.
- Let x ≈ p/2 and let h be small perturbation to x. Then

absolute error = cos(x+h) – cos(x)
≈ -h sin(x) ≈ -h,
relative error ≈ -h tan(x) ≈ 8

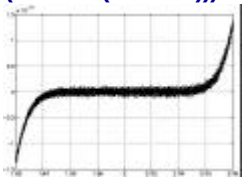
- So small change in x near p/2 causes large relative change in cos(x) regardless of method used.
- cos(1.57079) = 0.63267949 x 10^{-5}
- cos(1.57078) = 1.64267949 x 10^{-5}
- Relative change in output is a quarter million times greater than relative change in input.



46

Example: Polynomial Evaluation Using Horner's Rule

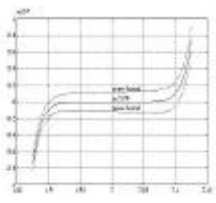
- **Horner's rule to evaluate** $p = \sum_{k=0}^{n-1} c_k * x^k$
  - $p = c_n$, for k=n-1 down to 0, $p = x*p + c_k$
- **Numerically Stable**
- **Apply to** $(x-2)^9 = x^9 - 18*x^8 + \ldots - 512$
  $-2^9 + x*(2^8 - x*(2^7 + \ldots)))$
- **Evaluated around 2**

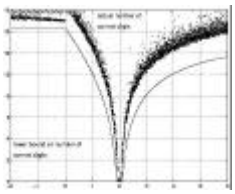


47

Example: polynomial evaluation (continued)

- $(x-2)^9 = x^9 - 18*x^8 + \ldots - 512$
- **We can compute error bounds using**
  - fl(a op b)=(a op b)*(1+d)





48

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”

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

+-	0…0	0……0
+-	Not 0 or all 1s	anything
+-	0…0	nonzero
+-	1….1	0……0
+-	1….1	nonzero

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

Lecture 5: Memory Hierarchy and Cache

Cache: A safe place for hiding and storing things.
Webster's New World Dictionary (1976)

Tools for Performance Evaluation

- Timing and performance evaluation has been an art
  - Resolution of the clock
  - Issues about cache effects
  - Different systems
- Situation about to change
  - Today's processors have counters

Performance Counters

- Almost all high performance processors include hardware performance counters.
- On most platforms the APIs, if they exist, are not appropriate for a common user, functional or well documented.
- Existing performance counter APIs
  - Intel Pentium
  - SGI MIPS R10000
  - IBM Power series
  - DEC Alpha pfm pseudo - device interface
  - Via Windows 95, NT and Linux on these systems

55

Performance Data (cont.)

- Cycle count
- Floating point instruction count
- Integer instruction count
- Instruction count
- Load/store count
- Branch taken / not taken count
- Branch
- Pipeline stalls due to memory subsystem
- Pipeline stalls due to resource conflicts
- I/D cache misses for different levels
- Cache invalidations

56

PAPI Usage

- Application is instrumented with PAPI
- Will be layered over the best existing vendor-specific APIs for these platforms
- call PAPIf_flops(real_time, proc_time, flpins, mflops, check)
- PAPI_flops(&real_time, &proc_time, &flpins, &mflops);
- Show example http://www.cs.utk.edu/~terpstra/using_papi/

57

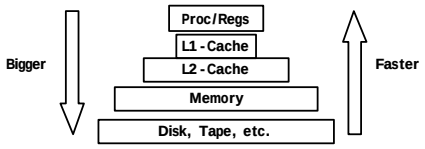
Cache and Its Importance in Performance

- Motivation:
  - Time to run code = clock cycles running code + clock cycles waiting for memory
  - For many years, CPU's have sped up an average of 50% per year over memory chip speed ups.
- Hence, memory access is the bottleneck to computing fast.
- Definition of a cache:
  - Dictionary: a safe place to hide or store things.
  - Computer: a level in a memory hierarchy.

58

What is a cache?

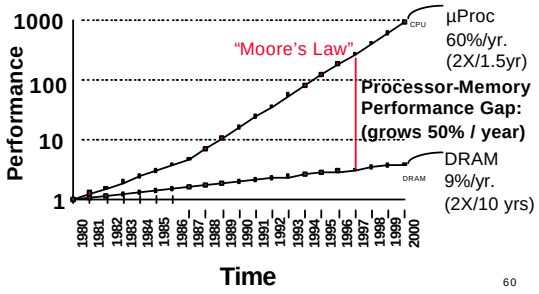
- Small, fast storage used to improve average access time to slow memory.
- Exploits spacial and temporal locality
- In computer architecture, almost everything is a cache!
  - Registers "a cache" on variables – software managed
  - First-level cache a cache on second-level cache
  - Second-level cache a cache on memory
  - Memory a cache on disk (virtual memory)
  - TLB a cache on page table
  - Branch-prediction a cache on prediction information?



59

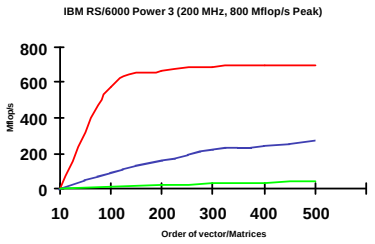
Who Cares About the Memory Hierarchy?

Processor-DRAM Memory Gap (latency)



60

Matrix-multiply, optimized several ways



61

Cache Sporting Terms

- **Cache Hit:** The CPU requests data that is already in the cache. We want to maximize this. The hit rate is the percentage of cache hits.
- **Cache Miss:** The CPU requests data that is not in cache. We want to minimize this. The miss time is how long it takes to get data, which can be variable and is highly architecture dependent.
- Two level caches are common. The L1 cache is on the CPU chip and the L2 cache is separate. The L1 misses are handled faster than the L2 misses in most designs.
- Upstream caches are closer to the CPU than downstream caches. A typical Alpha CPU has L1-L3 caches. Some MIPS CPU's do, too.

62

Cache Benefits

- Data cache was designed with two key concepts in mind
  - Spatial Locality
    - » When an element is referenced its neighbors will be referenced too
    - » Cache lines are fetched together
    - » Work on consecutive data elements in the same cache line
  - Temporal Locality
    - » When an element is referenced, it might be referenced again soon
    - » Arrange code so that data in cache is reused often

63

Cache-Related Terms

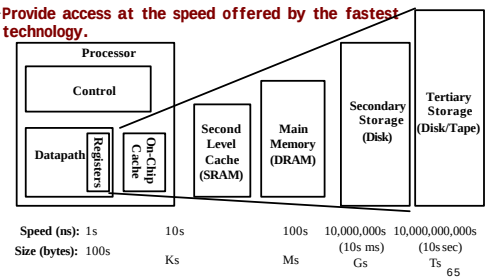
Least Recently Used (LRU): Cache replacement strategy for set associative caches. The cache block that is least recently used is replaced with a new block.

Random Replace: Cache replacement strategy for set associative caches. A cache block is randomly replaced.

64

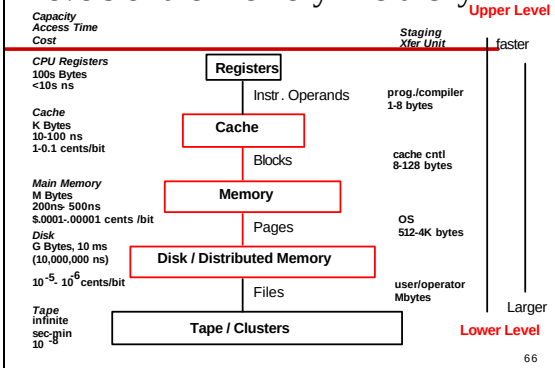
A Modern Memory Hierarchy

- By taking advantage of the principle of locality:
  - Present the user with as much memory as is available in the cheapest technology.
  - Provide access at the speed offered by the fastest technology.



65

Levels of the Memory Hierarchy



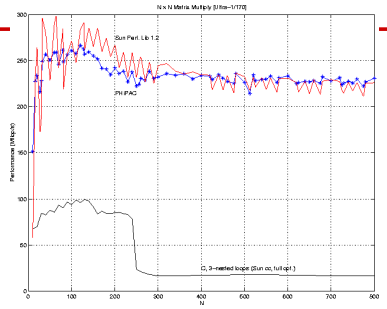
66

Reality

- Modern processors use a variety of techniques for performance
  - caches
    - » small amount of fast memory where values are "cached" in hope of reusing recently used or nearby data
    - » different memory ops can have very different costs
  - parallelism
    - » superscalar processors have multiple "functional units" that can run in parallel
    - » different orders, instruction mixes have different costs
  - pipelining
    - » a form of parallelism, like an assembly line in a factory
- Why is this your problem?
  - » In theory, compilers understand all of this and can optimize your program; in practice they don't.

67

optimized several ways



Speed of n-by-n matrix multiply on Sun Ultra-1/170, peak = 330 MFlops

68

Traditional Four Questions for Memory Hierarchy Designers

- Q1: Where can a block be placed in the upper level? *(Block placement)*
  - Fully Associative, Set Associative, Direct Mapped
- Q2: How is a block found if it is in the upper level? *(Block identification)*
  - Tag/Block
- Q3: Which block should be replaced on a miss? *(Block replacement)*
  - Random, LRU
- Q4: What happens on a write? *(Write strategy)*
  - Write Back or Write Through (with Write Buffer)

69

Cache-Related Terms

- ICACHE : Instruction cache
- DCACHE (L1) : Data cache closest to registers
- SCACHE (L2) : Secondary data cache
  - Data from SCACHE has to go through DCACHE to registers
  - SCACHE is larger than DCACHE
  - Not all processors have SCACHE

70

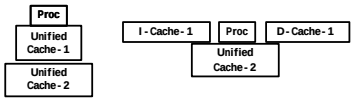
Unified versus Split Caches

- This refers to having a single or separate caches for data and machine instructions.
- Split is obviously superior. It reduces thrashing, which we will come to shortly..

71

Unified vs Split Caches

- Unified vs Separate I&D



- Example:
  - 16KB I&D: Inst miss rate=0.64%, Data miss rate=6.47%
  - 32KB unified: Aggregate miss rate=1.99%
- Which is better (ignore L2 cache)?
  - Assume 33% data ops Þ 75% accesses from instructions (1.0/1.33)
  - hit time=1, miss time=50
  - Note that *data* hit has 1 stall for unified cache (only one port)

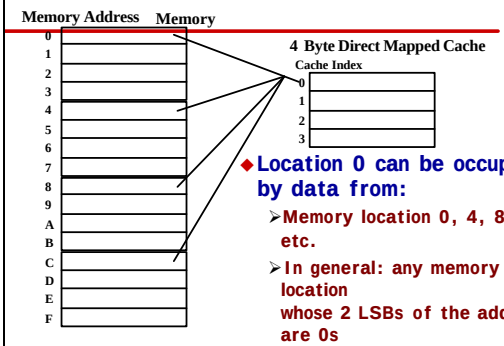
72

Where to misses come from?

- Classifying Misses: 3 Cs
  - *Compulsory*—The first access to a block is not in the cache, so the block must be brought into the cache. Also called *cold start misses* or *first reference misses*.
    (Misses in even an Infinite Cache)
  - *Capacity*—If the cache cannot contain all the blocks needed during execution of a program, capacity misses will occur due to blocks being discarded and later retrieved.
    (Misses in Fully Associative Size X Cache)
  - *Conflict*—If block-placement strategy is set associative or direct mapped, conflict misses (in addition to compulsory & capacity misses) will occur because a block can be discarded and later retrieved if too many blocks map to its set. Also called *collision misses* or *interference misses*.
    (Misses in N-way Associative, Size X Cache)
- 4th "C": (for parallel)
  - *Coherence* - Misses caused by cache coherence.

73

Simplest Cache: Direct Mapped

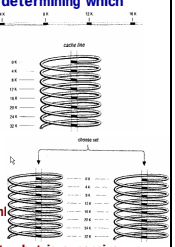


- Location 0 can be occupied by data from:
  - Memory location 0, 4, 8, ... etc.
  - In general: any memory location whose 2 LSBs of the address are 0s
  - Address<1:0> => cache

74

Cache Mapping Strategies

- There are two common sets of methods in use for determining which cache lines are used to hold copies of mem
- Direct: Cache address = memory address MODULO cache size.
- Set associative: There are N cache banks and memory is assigned to just one of the banks. There are three algorithmic choices for which line replace:
  - Random: Choose any line using an analog random num generator. This is cheap and simple to make.
  - LRU (least recently used): Preserves temporal locality, but is expensive. This is not much better than random according to (biased) studies.
  - FIFO (first in, first out): Random is far superior.



75

Cache Basics

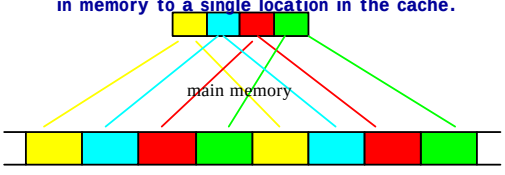
- Cache hit: a memory access that is found in the cache -- cheap
- Cache miss: a memory access that is not in the cache - expensive, because we need to get the data from elsewhere
- Consider a tiny cache (for illustration only)

- Cache line length: number of bytes loaded together in one entry
- Direct mapped: only one address (line) in a given range in cache
- Associative: 2 or more lines with different addresses exist

76

Direct-Mapped Cache

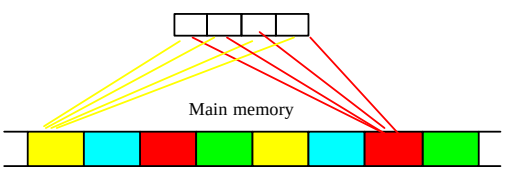
- Direct mapped cache: A block from main memory can go in exactly one place in the cache. This is called direct mapped because there is direct mapping from any block address in memory to a single location in the cache.



77

Fully Associative Cache

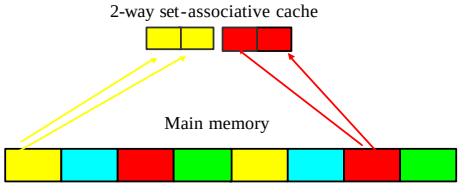
- Fully Associative Cache : A block from main memory can be placed in any location in the cache. This is called fully associative because a block in main memory may be associated with any entry in the cache.



78

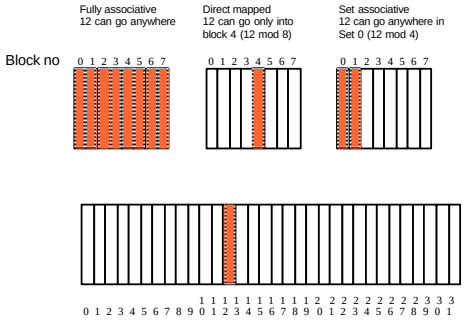
Set Associative Cache

- **Set associative cache : The middle range of designs between direct mapped cache and fully associative cache is called set-associative cache. In a n-way set-associative cache a block from main memory can go into N (N > 1) locations in the cache.**



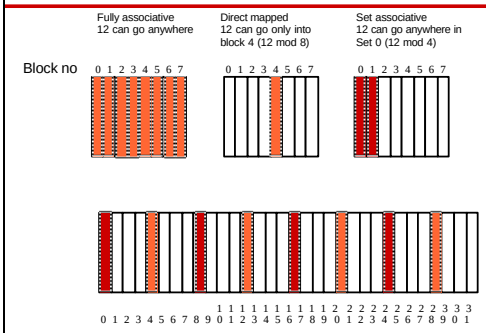
79

Here assume cache has 8 blocks, while memory has 32



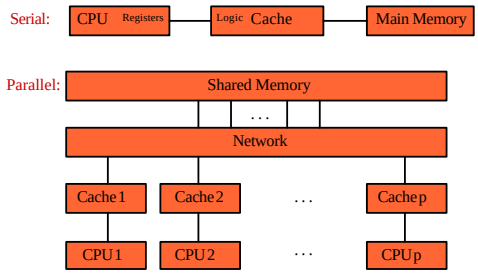
80

Here assume cache has 8 blocks, while memory has 32



81

Diagrams



82

Tuning for Caches

1. **Preserve locality.**
2. **Reduce cache thrashing.**
3. **Loop blocking when out of cache.**
4. **Software pipelining.**

83

Registers

- **Registers are the source and destination of most CPU data operations.**
- **They hold one element each.**
- **They are made of static RAM (SRAM), which is very expensive.**
- **The access time is usually 1-1.5 CPU clock cycles.**
- **Registers are at the top of the**

84

Memory Banking

- This started in the 1960's with both 2 and 4 way interleaved memory banks. Each bank can produce one unit of memory per bank cycle. Multiple reads and writes are possible in parallel.
  - Memory chips must internally recover from an access before it is reaccessed
- The bank cycle time is currently 4-8 times the CPU clock time and getting worse every year.
- Very fast memory (e.g., SRAM) is unaffordable in large quantities.
- This is not perfect. Consider a 4 wa[illegible] memory and a stride 4 algorithm. T[illegible] non-interleaved memory systems.

Bank 1	Bank 2	Bank 3	Bank 4
A(1)	A(2)	A(3)	A(4)
A(5)	A(6)	A(7)	A(8)
A(9)	A(10)	A(11)	A(12)
A(13)	A(14)	A(15)	A(16)
.	.	.	.
.	.	.	.
.	.	.	.

The Principle of Locality

- **The Principle of Locality:**
  - Program access a relatively small portion of the address space at any instant of time.
- **Two Different Types of Locality:**
  - Temporal Locality (Locality in Time): If an item is referenced, it will tend to be referenced again soon (e.g., loops, reuse)
  - Spatial Locality (Locality in Space): If an item is referenced, items whose addresses are close by tend to be referenced soon

86

Principals of Locality

- **Temporal:** an item referenced now will be again soon.
- **Spatial:** an item referenced now causes neighbors to be referenced soon.
- Lines, not words, are moved between memory levels. Both principals are satisfied. There is an optimal line size based on the properties of the data bus and the memory subsystem designs.
- Cache lines are typically 32-128 bytes with 1024 being the longest currently

87

What happens on a write?

- ***Write through*—The information is written to both the block in the cache and to the block in the lower-level memory.**
- ***Write back*—The information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced in cache.**
  - is block clean or dirty?
- Pros and Cons of each?
  - WT: read misses cannot result in writes

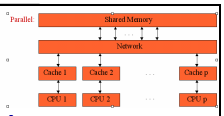
88

Cache Thrashing

- **Thrashing occurs when frequently used cache lines replace each other. There are three primary causes for thrashing:**
  - Instructions and data can conflict, particularly in unified caches.
  - Too many variables or too large of arrays are accessed that do not fit into cache.
  - Indirect addressing, e.g., sparse

89

Cache Coherence for Multiprocessors



- **All data must be coherent between memory levels. Multiple processors with separate caches must inform the other processors quickly about data modifications (by the cache line). Only hardware is fast enough to do this.**
- **Standard protocols on multiprocessors:**
  - Snoopy: all processors monitor the memory bus.
  - Directory based: Cache lines maintain an extra 2 bits per processor to maintain clean/dirty status bits.

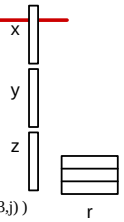
90

Processor Stall

- **Processor stall** is the condition where a cache miss occurs and the processor waits on the data.
- A better design allows any instruction in the instruction queue to execute that is ready. You see this in the design of some RISC CPU's, e.g., the RS6000 line.
- Memory subsystems with hardware data prefetch allow scheduling of data movement to cache.
- Software pipelining can be done when loops

Indirect Addressing

```
d = 0
do i = 1,n
  j = ind(i)
  d = d + sqrt( x(j)*x(j) + y(j)*y(j) + z(j)*z(j) )
end do
```



- Change loop statement to

```
d = d + sqrt( r(1,j)*r(1,j) + r(2,j)*r(2,j) + r(3,j)*r(3,j) )
```

- Note that r(1,j)-r(3,j) are in contiguous memory and probably are in the same cache line (d is probably in a register and is irrelevant). The original form uses 3 cache lines at every instance of the loop and can cause

Cache Thrashing by Memory Allocation

```
parameter ( m = 1024*1024 )
real a(m), b(m)
```

- For a 4 Mb direct mapped cache, a(i) and b(i) are always mapped to the same cache line. This is trivially avoided using padding.

```
real a(m), extra(32), b(m)
```


- extra is at least 128 bytes in length

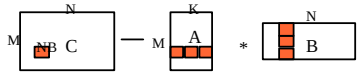
Cache Blocking

- We want blocks to fit into cache. On parallel computers we have p x cache so that data may fit into cache on p processors, but not one. This leads to superlinear speed up! Consider matrix-matrix multiply.

```
do k = 1,n
  do j = 1,n
    do i = 1,n
      c(i,j) = c(i,j) + a(i,k)*b(k,j)
    end do
  end do
end do
```

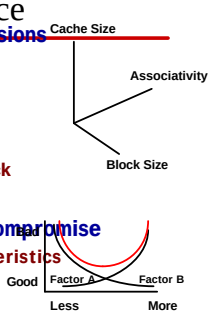

Cache Blocking

```
do kk = 1,n,nblk
  do jj = 1,n,nblk
    do ii = 1,n,nblk
      do k = kk,kk+nblk-1
        do j = jj,jj+nblk-1
          do i = ii,ii+nblk-1
            c(i,j) = c(i,j) + a(i,k) * b(k,j)
          end do
      . . .
end do
```



Summary : The Cache Design Space

- **Several interacting dimensions**
  - cache size
  - block size
  - associativity
  - replacement policy
  - write-through vs write-back
  - write allocation
- **The optimal choice is a compromise**
  - depends on access characteristics
    - » workload
    - » use (I-cache, D-cache, TLB)
  - depends on technology / cost
- **Simplicity often wins**



Lessons

- The actual performance of a simple program can be a complicated function of the architecture
- Slight changes in the architecture or program change the performance significantly
- Since we want to write fast programs, we must take the architecture into account, even on uniprocessors
- Since the actual performance is so complicated, we need simple models to help us design efficient algorithms
- We will illustrate with a common technique for improving cache performance, called blocking

97

Addition for Caches

- Dimension A(n,n), B(n,n), C(n,n)
- A, B, C stored by column (as in Fortran)
- Algorithm 1:
  - for i=1:n, for j=1:n, A(i,j) = B(i,j) + C(i,j)
- Algorithm 2:
  - for j=1:n, for i=1:n, A(i,j) = B(i,j) + C(i,j)
- What is "memory access pattern" for Algs 1 and 2?
- Which is faster?
- What if A, B, C stored by row (as in C)?

98

Homework Assignment

- Implement, in Fortran or C, the six different ways to perform matrix multiplication by interchanging the loops. (Use 64-bit arithmetic.) Make each implementation a subroutine, like:
- subroutine ijk (a, m, n, lda, b, k, ldb, c, ldc)
- subroutine ikj (a, m, n, lda, b, k, ldb, c, ldc)

99

Talk about Assignment

- http://www.cs.utk.edu/~dongarra/WEB-PAGES/SPRING-2002/homework05.html

100

Loop Fusion Example

```
/* Before */
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
      a[i][j] = 1/b[i][j] * c[i][j];
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
      d[i][j] = a[i][j] + c[i][j];

/* After */
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
  {   a[i][j] = 1/b[i][j] * c[i][j];
      d[i][j] = a[i][j] + c[i][j];}
```

2 misses per access to a & c vs. one miss per access; improve spatial locality

101

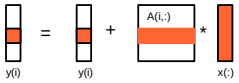
Multiply for Caches

- Several techniques for making this faster on modern processors
  - heavily studied
- Some optimizations done automatically by compiler, but can do much better
- In general, you should use optimized libraries (often supplied by vendor) for this and other very common linear algebra operations
  - BLAS = Basic Linear Algebra Subroutines
- Other algorithms you may want are not going to be supplied by vendor, so need to know these techniques

102

multiplication y = y + A*x

```
for i = 1:n
        for j = 1:n
                y(i) = y(i) + A(i,j)*x(j)
```



103

multiplication y = y + A*x

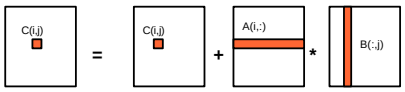
```
{read x(1:n) into fast memory}
{read y(1:n) into fast memory}
for i = 1:n
        {read row i of A into fast memory}
        for j = 1:n
                y(i) = y(i) + A(i,j)*x(j)
{write y(1:n) back to slow memory}
```

° m = number of slow memory refs = 3*n + n^2
° f = number of arithmetic operations = 2*n^2
° q = f/m ~= 2
° Matrix-vector multiplication limited by slow memory speed

104

C=C+A*B

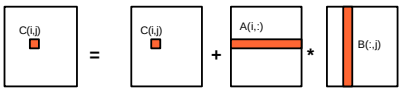
```
for i = 1 to n
        for j = 1 to n
        for k = 1 to n
                C(i,j) = C(i,j) + A(i,k) * B(k,j)
```



105

C=C+A*B(unblocked, or untiled)

```
for i = 1 to n
  {read row i of A into fast memory}
   for j = 1 to n
      {read C(i,j) into fast memory}
      {read column j of B into fast memory}
       for k = 1 to n
          C(i,j) = C(i,j) + A(i,k) * B(k,j)
      {write C(i,j) back to slow memory}
```

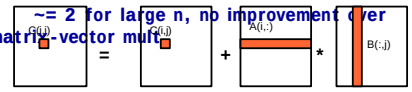


106

(unblocked, or untiled)

q=ops/slow mem ref

Number of slow memory references on unblocked matrix multiply
m = n^3 read each column of B n times
+ n^2 read each column of A once for each i
+ 2*n^2 read and write each element of C once
= n^3 + 3*n^2
So q = f/m = (2*n^3)/(n^3 + 3*n^2)
~= 2 for large n, no improvement over matrix-vector mult

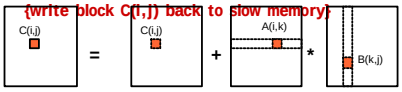


107

(blocked, or tiled)

Consider A,B,C to be N by N matrices of b by b subblocks where b=n/N is called the blocksize

```
for i = 1 to N
    for j = 1 to N
      {read block C(i,j) into fast memory}
      for k = 1 to N
                {read block A(i,k) into fast memory}
                {read block B(k,j) into fast memory}
                C(i,j) = C(i,j) + A(i,k) * B(k,j) {do a
  matrix multiply on blocks}
      {write block C(i,j) back to slow memory}
```



108

(blocked or tiled)

q=ops/slow mem ref

Why is this algorithm correct?

Number of slow memory references on blocked matrix multiply
m = N*n^2 read each block of B N^3 times (N^3 * n/N * n/N)
+ N*n^2 read each block of A N^3 times
+ 2*n^2 read and write each block of C once
= (2*N + 2)*n^2
So q = f/m = 2*n^3 / ((2*N + 2)*n^2)
~= n/N = b for large n

So we can improve performance by increasing the blocksize b
Can be much faster than matrix-vector multiplty (q=2)

Limit: All three blocks from A,B,C must fit in fast memory (cache), so we
cannot make these blocks arbitrarily large: 3*b^2 <= M, so q ~= b <= sqrt(M/3)

Theorem (Hong, Kung, 1981): Any reorganization of this algorithm
(that uses only associativity) is limited to q =O(sqrt(M))

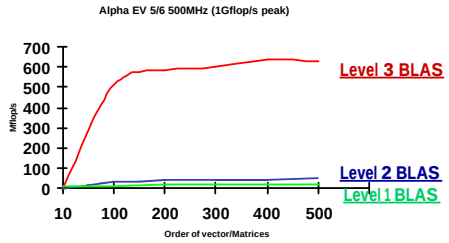
109

Algebra Subroutines)

- Industry standard interface(evolving)
- Vendors, others supply optimized implementations
- History
  - BLAS1 (1970s):
    - » vector operations: dot product, saxpy (y= *x+y), etc
    - » m=2*n, f=2*n, q ~1 or less
  - BLAS2 (mid 1980s)
    - » matrix-vector operations: matrix vector multiply, etc
    - » m=n^2, f=2*n^2, q~2, less overhead
    - » somewhat faster than BLAS1
  - BLAS3 (late 1980s)
    - » matrix-matrix operations: matrix matrix multiply, etc
    - » m >= 4n^2, f=O(n^3), so q can possibly be as large as n, so BLAS3 is potentially much faster than BLAS2
- Good algorithms used BLAS3 when possible (LAPACK)

110

BLAS for Performance



BLAS 3 (n-by-n matrix matrix multiply) vs
BLAS 2 (n-by-n matrix vector multiply) vs
BLAS 1 (saxpy of n vectors)

- Development of blocked algorithms

111

practice

- Tiling for registers
  - loop unrolling, use of named "register" variables
- Tiling for multiple levels of cache
- Exploiting fine-grained parallelism within the processor
  - super scalar
  - pipelining
- Complicated compiler interactions
- Hard to do by hand (but you'll try)
- Automatic optimization an active research area
  - PHIPAC: www.icsi.berkeley.edu/~bilmes/phipac

112

Matrix Multiply

- The traditional algorithm (with or without tiling) has O(n^3) flops
- Strassen discovered an algorithm with asymptotically lower flops
  - O(n^2.81)
- Consider a 2x2 matrix multiply, normally 8 multiplies

Let M = [m11 m12] = [a11 a12] [b11 b12]
[m21 m22] [a21 a22] [b21 b22]

Let p1 = (a12 - 122) * (b21 + b22)
p2 = (a11 + a22) * (b11 + b22)
p3 = (a11 - a21) * (b11 + b12)
p4 = (a11 + a12) * b22
p5 = a11 * (b12 - b22)
p6 = a22 * (b21 - b11)
p7 = (a21 + a22) * b11

Then m11 = p1 + p2 - p4 + p6
m12 = p4 + p5
m21 = p6 + p7
m22 = p2 - p3 + p5 - p7

Extends tonxn by divide&conquer

113

(continued)

T(n) = Cost of multiplying nxn matrices
= 7*T(n/2) + 18*(n/2)^2
= O(n^log_2 7)
= O(n^2.81)

- Available in several libraries
- Up to several time faster if n large enough (100s)
- Needs more memory than standard algorithm
- Can be less accurate because of roundoff error
- Current world's record is O(n^2.376..)

114

Summary

- Performance programming on uniprocessors requires
  - understanding of memory system
    - » levels, costs, sizes
  - understanding of fine-grained parallelism in processor to produce good instruction mix
- Blocking (tiling) is a basic approach that can be applied to many matrix algorithms
- Applies to uniprocessors and parallel processors
  - The technique works for any architecture, but choosing the blocksize b and other details depends on the architecture
- Similar techniques are possible on other data structures
- You will get to try this in Assignment 2 (see the class homepage)

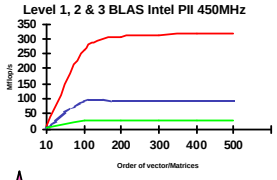
Summary: Memory Hierachy

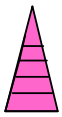
- Virtual memory was controversial at the time:
  can SW automatically manage 64KB across many programs?
  - 1000X DRAM growth removed the controversy
- Today VM allows many processes to share single memory without having to swap all processes to disk; today VM protection is more important than memory hierarchy
- Today CPU time is a function of (ops, cache misses) vs. just f(ops):
  What does this mean to Compilers, Data structures, Algorithms?

Performance = Effective Use of Memory Hierarchy

- Can only do arithmetic on data at the top of the hierarchy
- Higher level BLAS lets us do this

BLAS	Memory Refs	Flops	Flops/ Memory Refs
Level 1 $y=y+_a x$	3n	2n	2/3
Level 2 $y=y+Ax$	n^2	$2n^2$	2
Level 3 $C=C+AB$	$4n^2$	$2n^3$	n/2





- Development of blocked algorithms important for performance

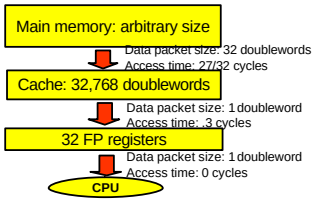
Homework Assignment

- Implement, in Fortran or C, the six different ways to perform matrix multiplication by interchanging the loops. (Use 64-bit arithmetic.) Make each implementation a subroutine, like:
- subroutine ijk (a, m, n, lda, b, k, ldb, c, ldc)
- subroutine ikj (a, m, n, lda, b, k, ldb, c, ldc)

Thanks

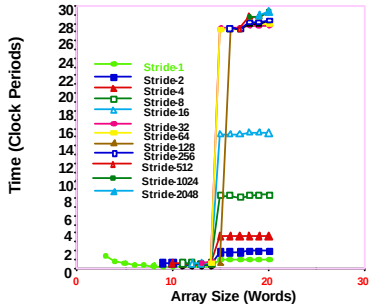
- These slides came in part from courses taught by the following people:
  - Kathy Yelick, UC, Berkeley
  - Dave Patterson, UC, Berkeley
  - Randy Katz, UC, Berkeley
  - Craig Douglas, U of Kentucky
- Computer Architecture A Quantitative Approach,

Schematic View of a Typical Memory Hierarchy: IBM 590



Design your program for optimal spatial and temporal data locality !

Effect of Stride and Array Size on Access Time



121

Optimal Data Locality: Data Structures

```
   dimension rx(n), ry(n),
&  rz(n), fx(n), fy(n), fz(n)

    do 100 i=1,n
      do 100 j=1,i-1
        dist=(rx(i)-rx(j))**2 +
&     (ry(i)-ry(j))**2 +…
        if (dist.le.cutoff) then
c calculate interaction
            dfx=…
            dfy=…
            dfz=…
c accumulate force
            fx(j)=fx(j)+dfx
            fy(j)=fy(j)+dfy
            fz(j)=fz(j)+dfz
          endif
100 continue
```

```
    dimension r(3,n), f(3,n)


    do 100 i=1,n
      do 100 j=1,i-1
        dist=(rx(i)-rx(j))**2 +
&     (ry(i)- ry(j))**2 +…
        if (dist.le.cutoff) then
c calculate interaction
            dfx=…
            dfy=…
            dfz=…
c accumulate force
            fx(1,j)=fx(1,j)+dfx
            fy(2,j)=fy(2,j)+dfy
            fz(3,j)=fz(3,j)+dfz
          endif
100       continue
```

122

Instruction Level Parallelism: Floating Point

- IBM RS/6000 Power2 (130 MHz)
  - 2 FP units, each capable of
    - 1 fused multiply-add (1;2) or
    - 1 add (1;1) or
    - 1 multiply (1;2)
    - 1 quad load/store (1;1)
  - leading to (up to)
    - 4 FP Ops per CP
    - 4 mem access Ops per CP
- DEC Alpha EV5 (350 MHz)
  - 1 FP unit, capable of
    - 1 floating point add pipeline (1;4)
    - 1 floating point mult. pipeline (1;4)
    - 1 load/store (1;3)
  - leading to (up to)
    - 2 FP Ops per CP
    - 1 mem access Ops per CP
- SGI R10000 (200 MHz)
  - 1 FP unit capable of
    - 1 floating point add pipeline (1;2)
    - 1 floating point multiply pipeline (1;2)
    - 1 load/store (1;3)
  - leading to (up to)
    - 2 FP Ops per CP
    - 1 memory access Ops per CP

123

Code Restructuring for On-Chip Parallelism: Original Code

```
      program length
      parameter (n=2**14)
      dimension a(n)
      subroutine length1(n,aa,tt)
      implicit real*8 (a-h,o-z)
      dimension a(n)
      tt=0.d0
      do 100, j=1,n
          tt=tt+a(j)*a(j)
  100   continue
      return
      end
```

124

Modified Code for On-Chip Parallelism

```
      subroutine length4 (n,a,tt)
c works correctly only if n is multiple of 4
      implicit real*8 (a-h,o-z)
      dimension a(n)
      t1=0.d0
      t2=0.d0
      t3=0.d0
      t4=0.d0
      do 100, j=1,n-3,4
c first floating point instruction unit, all even cycles
            t1=t1+a(j+0)*a(j+0)
c first floating point instruction unit, all odd cycles
            t2=t2+a(j+1)*a(j+1)
c second floating point instruction unit, all even cycles
            t3=t3+a(j+2)*a(j+2)
c second floating point instruction unit, all odd cycles
            t4=t4+a(j+3)*a(j+3)
100   continue
      tt= t1+t2+t3+t4
      return
      end
```

125

Software Pipelining

```
c first FP unit, first cycle:
c do one MADD (with t1 and a0 available in registers) and load a1:
    t1=t1+a0* a0
    a1=a(j+1)
c first floating point unit, second cycle:
c do one MADD (with t2 and t1 available in registers) and load a0 for next
    iteration:
    t2=t2+a1*a1
    a0=a(j+0+4)
c second FP unit, first cycle:
c do one MADD (with t3 and a2 available in registers) and load a3:
    t3=t3+a2*a2
    a3=a(j+2)
c second FP unit, second cycle:
c do one MADD (with t4 and a3 available in registers) and load a2 for next
    iteration:
    t4=t4+a3*a3
    a2=a(j+1+4)
```

126

Improving Ratio of Floating Point Operations to Memory Accesses

```
      subroutine mult(n1,nd1,n2,nd2,y,a,x)
      implicit real*8 (a-h,o-z)
      dimension a(nd1,nd2),y(nd2),x(nd1)

      do 10, i=1,n1
      t=0.d0
      do 20, j=1,n2
20       t=t+a(j,i)*x(j)
10       y(i)=t
      return
      end
```

****** 2 FLOPS**
****** 2 LOADS**

Improving Ratio of Floating Point Operations to Memory Accesses

```
c works correctly when n1,n2 are multiples of 4
      dimension a(nd1,nd2), y(nd2), x(nd1)
      do i=1,n1-3,4
        t1=0.d0
        t2=0.d0
        t3=0.d0
        t4=0.d0
        do j=1,n2-3,4
         t1=t1+a(j+0,i+0)*x(j+0)+a(j+1,i+0)*x(j+1)+
     1     a(j+2,i+0)*x(j+2)+a(j+3,i+1)*x(j+3)
         t2=t2+a(j+0,i+1)*x(j+0)+a(j+1,i+1)*x(j+1)+
     1     a(j+2,i+1)*x(j+2)+a(j+3,i+0)*x(j+3)
         t3=t3+a(j+0,i+2)*x(j+0)+a(j+1,i+2)*x(j+1)+
     1     a(j+2,i+2)*x(j+2)+a(j+3,i+2)*x(j+3)
         t4=t4+a(j+0,i+3)*x(j+0)+a(j+1,i+3)*x(j+1)+
     1     a(j+2,i+3)*x(j+2)+a(j+3,i+3)*x(j+3)
        enddo
        y(i+0)=t1
        y(i+1)=t2
        y(i+2)=t3
        y(i+3)=t4
      enddo
```

32 FLOPS
20 LOADS

Summary of Single-Processor Optimization Techniques (I)

- **Spatial and temporal data locality**
- **Loop unrolling**
- **Blocking**
- **Software pipelining**
- **Optimization of data structures**
- **Special functions, library subroutines**

Summary of Optimization Techniques (II)

- **Achieving high-performance requires code restructuring. Minimization of memory traffic is the single most important goal.**
- **Compilers are getting better: good at software pipelining. But they are not there yet: can do loop transformations only in simple cases, usually fail to produce optimal blocking, heuristics for unrolling may not match your code well, etc.**
- **The optimization process is machine-specific and requires detailed architectural knowledge.**

Amdahl's Law

Amdahl's Law places a strict limit on the speedup that can be realized by using multiple processors. Two equivalent expressions for Amdahl's Law are given below:

$t_N = (f_p/N + f_s)t_1$ Effect of multiple processors on run time

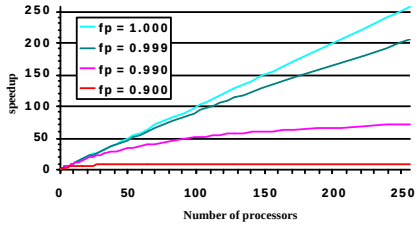
$S = 1/(f_s + f_p/N)$ Effect of multiple processors on speedup

Where:

f_s = serial fraction of code
f_p = parallel fraction of code = 1 - f_s
N = number of processors

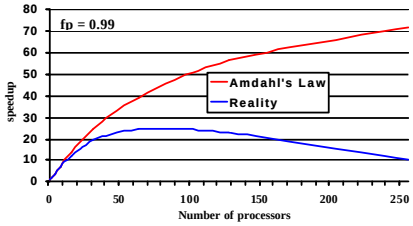
Illustration of Amdahl's Law

It takes only a small fraction of serial content in a code to degrade the parallel performance. It is essential to determine the scaling behavior of your code before doing production runs using large numbers of processors



Amdahl's Law Vs. Reality

Amdahl's Law provides a theoretical upper limit on parallel speedup *assuming that there are no costs for communications*. In reality, communications (and I/O) will result in a further degradation of performance.



More on Amdahl's Law

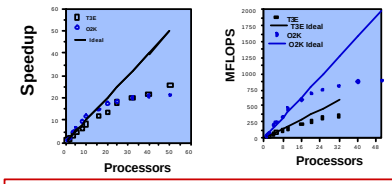
- **Amdahl's Law can be generalized to any two processes of with different speeds**
- **Ex.: Apply to $f_{processor}$ and f_{memory}:**
  - **The growing processor-memory performance gap will undermine our efforts at achieving maximum possible speedup!**

Gustafson's Law

- **Thus, Amdahl's Law predicts that there is a maximum scalability for an application, determined by its parallel fraction, and this limit is generally not large.**
- **There is a way around this: *increase the problem size***
  - ***bigger problems mean bigger grids or more particles: bigger arrays***
  - **number of serial operations generally remains constant; number of parallel operations increases: *parallel fraction increases***

Parallel Performance Metrics: Speedup

Relative performance: Absolute performance:



Speedup is only one characteristic of a program - it is not synonymous with performance. In this comparison of two machines the code achieves comparable speedups but one of the machines is faster.

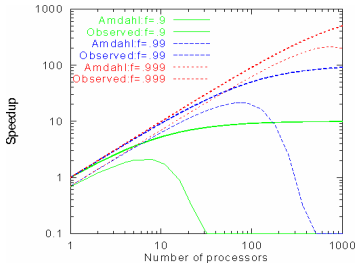
Fixed-Problem Size Scaling

- a.k.a. Fixed-load, Fixed-Problem Size, Strong Scaling, Problem-Constrained, constant-problem size (CPS), variable subgrid
- Amdahl Limit: $S_A(n) = T(1) / T(n) = \frac{1}{f / n + (1 - f)}$
- This bounds the speedup based only on the fraction of the code that cannot use parallelism (1-*f*); it ignores all other factors
- $S_A \dashrightarrow 1 / (1-f)$ as $n \dashrightarrow \infty$

Fixed-Problem Size Scaling (Cont'd)

- Efficiency (*n*) = T(1) / [T(*n*) * n]
- Memory requirements decrease with *n*
- Surface-to-volume ratio increases with *n*
- Superlinear speedup possible from cache effects
- Motivation: what is the largest # of procs I can use effectively and what is the fastest time that I can solve a given problem?
- Problems:
  - Sequential runs often not possible (large problems)
  - Speedup (and efficiency) is misleading if processors are slow

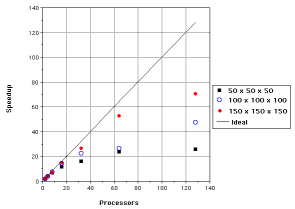
Fixed-Problem Size Scaling: Examples



S. Goedecker and Adolfy Hoisie, Achieving High Performance in Numerical Computations on RISC Workstations and Parallel Systems, *International Conference on Computational Physics: PC'97 Santa Cruz, August 25-28 1997.*

139

Fixed-Problem Size Scaling Examples



140

Scaled Speedup Experiments

- a.k.a. Fixed Subgrid-Size, Weak Scaling, Gustafson scaling.
- Motivation: Want to use a larger machine to solve a larger global problem *in the same amount of time*.
- Memory and surface-to-volume effects remain constant.

141

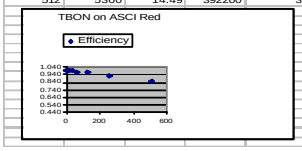
Scaled Speedup Experiments

- Be wary of benchmarks that scale problems to unreasonably-large sizes
  - scale the problem to fill the machine when a smaller size will do;
  - simplify the science in order to add computation
    -> "World's largest MD simulation - 10 gazillion particles!"
  - run grid sizes for only a few cycles because the full run won't finish during this lifetime or because the resolution makes no sense compared with resolution of input data
- Suggested alternate approach (Gustafson): Constant time benchmarks
  - run code for a fixed time and measure work done

142

Example of a Scaled Speedup Experiment

Processor	NChains	Time	Natoms	Time per Atom per PE	Time per Atom	Efficiency
1	32	38.4	2368	1.62E-02	1.62E-02	1.000
2	64	38.4	4736	8.11E-03	1.62E-02	1.000
4	128	38.5	9472	4.06E-03	1.63E-02	0.997
8	256	38.6	18944	2.04E-03	1.63E-02	0.995
16	512	38.7	37888	1.02E-03	1.63E-02	0.992
32	940	35.7	69560	5.13E-04	1.64E-02	0.987
64	1700	32.7	125800	2.60E-04	1.66E-02	0.975
128	2800	27.4	207200	1.32E-04	1.69E-02	0.958
256	4100	20.75	303400	6.84E-05	1.75E-02	0.926
512	5300	14.49	392200	3.69E-05	1.89E-02	0.857



143

144