 Lecture 4: Overview of High-Performance Computing

Jack Dongarra
Computer Science Department
University of Tennessee

High Performance Computers

- ~ 20 years ago
  - \(1 \times 10^6\) Floating Point Ops/sec (MFlop/s)
  - Scalar based
- ~ 10 years ago
  - \(1 \times 10^9\) Floating Point Ops/sec (GFlop/s)
  - Vector & Shared memory computing, bandwidth aware
  - Black box: need, bit very tolerant
- Today
  - \(1 \times 10^{12}\) Floating Point Ops/sec (TFlop/s)
  - Highly parallel, distributed processing, message passing, network based
  - Data decomposition, communication/computation
- ~ 10 years away
  - \(1 \times 10^{15}\) Floating Point Ops/sec (PFlop/s)
  - Many more levels, HPC, combinatorial graphs & HPC
  - More adaptable, LT and bandwidth aware, fault tolerant, extended precision, all end 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 \(\Rightarrow\) factor of \(\sim 7500\) (Doubling in less than a year, for 12 years)
- How
  - Clock = \(126x\)
  - External Bus Width & Caching -
    - \(36\) 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

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

Plan for Today

- Finish up last week’s slides
- Talk about computer arithmetic
- Begin Cache discussion
- Go over homework
20th List: The TOP10

<table>
<thead>
<tr>
<th>Rank</th>
<th>Manufacturer</th>
<th>Computer</th>
<th>GFlop/s</th>
<th>Installation Site</th>
<th>Country</th>
<th>Year</th>
<th>Area of Installation</th>
<th>F Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NEC</td>
<td>Cray Y-MP</td>
<td>35.86</td>
<td>Earth Simulator</td>
<td>Japan</td>
<td>1990</td>
<td>Research</td>
<td>~10000</td>
</tr>
<tr>
<td>2</td>
<td>HP</td>
<td>AlphaServer SC</td>
<td>3.16</td>
<td>Los Alamos</td>
<td>USA</td>
<td>2002</td>
<td>Research</td>
<td>0.4 GF/s</td>
</tr>
<tr>
<td>3</td>
<td>NEC</td>
<td>Cray Y-MP</td>
<td>3.34</td>
<td>National Laboratory</td>
<td>USA</td>
<td>1990</td>
<td>Research</td>
<td>0.4 GF/s</td>
</tr>
<tr>
<td>4</td>
<td>NEC</td>
<td>VPP-500</td>
<td>3.34</td>
<td>National Laboratory</td>
<td>USA</td>
<td>2002</td>
<td>Research</td>
<td>0.4 GF/s</td>
</tr>
<tr>
<td>5</td>
<td>SGI</td>
<td>CM-2</td>
<td>293 TF/s</td>
<td>National Laboratory</td>
<td>USA</td>
<td>2002</td>
<td>Research</td>
<td>0.4 GF/s</td>
</tr>
<tr>
<td>6</td>
<td>IBM</td>
<td>ES45 1 GHz</td>
<td>182 TF/s</td>
<td>Los Alamos</td>
<td>USA</td>
<td>2002</td>
<td>Research</td>
<td>0.4 GF/s</td>
</tr>
<tr>
<td>7</td>
<td>IBM</td>
<td>ASCI White</td>
<td>35.8 TF/s</td>
<td>National Laboratory</td>
<td>USA</td>
<td>2002</td>
<td>Research</td>
<td>0.4 GF/s</td>
</tr>
<tr>
<td>8</td>
<td>Fujitsu</td>
<td>Bluesky Pacific</td>
<td>35.8 TF/s</td>
<td>National Laboratory</td>
<td>USA</td>
<td>2002</td>
<td>Research</td>
<td>0.4 GF/s</td>
</tr>
<tr>
<td>9</td>
<td>IBM</td>
<td>Xeon Cluster</td>
<td>35.8 TF/s</td>
<td>National Laboratory</td>
<td>USA</td>
<td>2002</td>
<td>Research</td>
<td>0.4 GF/s</td>
</tr>
<tr>
<td>10</td>
<td>IBM</td>
<td>ASCI Red</td>
<td>35.8 TF/s</td>
<td>National Laboratory</td>
<td>USA</td>
<td>2002</td>
<td>Research</td>
<td>0.4 GF/s</td>
</tr>
</tbody>
</table>

182 fell off; 500 was 318 in June
Cluster on the Top500

Top500 Conclusions
- Microprocessor based supercomputers have brought a major change in accessibility and affordability.
- MPPs continue to account for 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, Gigabit, ATM
- Open Source Unix
  - Linux, BSD
- Message Passing Computing
  - MPI, PVM
  - HPF

Advantages:
- Best price-performance
- Low entry-level
- 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.

High-Performance Computing Directions:
- COTS PC Nodes
  - Pentium, Alpha, PowerPC, SMP
- COTS LAN/SAN Interconnect
  - Ethernet, Myrinet, Gigabit, ATM
- Open Source Unix
  - Linux, BSD
- Message Passing Computing
  - MPI, PVM
  - HPF

Distributed and Parallel Systems

Distributed systems
- hetero-
- geneous
- 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-sharing

Massively parallel systems
- homo-
- geneous
- 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-sharing

Virtual Environments

Do they make any sense?
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 scientists.
- **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.

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
  - ... encourages growth of a “third party” parallel software industry & parallel versions of widely used codes

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
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.

Achieving TeraFlops

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

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

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

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

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

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 different
- 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: +/- d.d. d + r
  - Sign bit: +/-
  - Significant d.d. (how many base-r digits of?)
  - Exponent exp (range?)
  - Others?

- Operations:
  - Arithmetic: +/-, x, /,...
  - Comparison (>, <, =)
  - Conversion between different formats
  - Short to long float numbers, FPU to integer
  - Exception handling
  - What to do for 0.0, largest number, etc.
  - Binary/decimal conversion
  - Arithmetic: +, -, *, /,...

- Language/library support for these operations

IEEE Floating Point Arithmetic Standard 754 - Normalized Numbers

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

- Format
  - Single: 32 bits, 23+1 significand bits, 2^-24
  - Double: 64 bits, 52+1 significand bits, 2^-107
  - Extended (80 bits on all Intel machines)

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

- Invalid Exception
  - Occurs when exact result not a well-defined real number
  - sqrt(-1)
  - Returns a NaN in all these cases

- Floating Point Arithmetic
  - Sign bit, nonzero significand, minimum exponent
  - Fills in gap between 0 and underflow threshold UN
  - Return a denorm, or zero

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
  - Return: 2^OV
  - Divide by zero Exception
  - Return: +- infinity

- Numerical Stability

Error Analysis

- Basic Error Formula
  - If a op b = (a op b)*(1+d)
  - Where d <= macheps
    - Assuming no overflow, underflow, or divide by zero

- Example: Adding 4 Numbers
  - x1, x2, x3, x4
  - Return: x1*(1+d) + x2*(1+d) + x3*(1+d) + x4*(1+d)
  - Where each |d| <= 3*macheps

- Get exact sum of slightly changed summands x^*1+a^*

- 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.

\[
\text{Cond} = \frac{\text{Relative change in solution}}{\text{Relative change in input data}} = \left| \frac{f(x + h) - f(x)}{f(x)} \right| / \left| \frac{(x' - x)}{x} \right|
\]
- 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 = \( f(x + h) - f(x) \)
- Relative error = \( \left| \frac{f(x + h) - f(x)}{f(x)} \right| \)

**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 \( k = n \) down to 0, \( p = x \cdot p + c_k \)
- Numerically Stable
- Apply to \( (x-2)^9 = \sum_{k=0}^{9} \binom{9}{k} (-2)^{9-k} x^k \)
- Evaluated around 2

**Sensitivity: 2 Examples**
- \( \cos(p/2) \) and 2-d System of Equations
  - Consider problem of computing cosine function for arguments near \( p/2 \).
  - Let \( x \approx p/2 \) and let \( h \) be small perturbation to \( x \). Then
    - absolute error \( = \cos(x + h) - \cos(x) \approx -h \cdot \sin(x) \approx -h \)
    - relative error \( = \cos(x + h) - \cos(x) \approx -h \cdot \sin(x) \approx -h \tan(x) \approx -h \)
  - So small change in \( x \) near \( p/2 \) 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 (continued)
- \( (x-2)^9 = x^9 - 18 \times x^8 + \ldots - 512 \)
- We can compute error bounds using
  - \( f(a \text{ op } b) = (a \text{ op } b)^*(1+d) \)
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

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

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

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/

Cache and Its Importance in Performance

- Motivation:
  - Time to run code = clock cycles running code + clock cycles waiting for memory
  - For many years, CPUs 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.

What is a cache?

- Small, fast storage used to improve average access time to slow memory
- Exploits spatial 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?

Who Cares About the Memory Hierarchy?

Processor-DRAM Memory Gap (latency)

- "Moore’s Law" (grows 50% / year)
- Processor-Memory Performance Gap:
  - µProc 60%/yr. (2X/1.5yr)
  - DRAM 9%/yr. (2X/10 yrs)
Matrix-multiply, optimized several ways

![Graph showing Mflop/s vs Order of vector/Matrices for IBM RS/6000 Power 3 (200 MHz, 800 MFlop/s Peak).]

**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 CPUs do, too.

**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

**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.

**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.

**Levels of the Memory Hierarchy**
- **Upper Level**
  - CPU/Registers: 100s bytes
  - Cache: 1-2 K bytes
  - Secondary Storage: 1-2 M bytes
- **Lower Level**
  - Main Memory: 10 M bytes
  - Tertiary Storage: 10 G bytes

**Capacity**
- **CPU Registers**: 100s bytes
- **Cache**: 1-2 K bytes
- **Main Memory**: 10 M bytes
- **Tertiary Storage**: 10 G bytes

**Access Time**
- **CPU Registers**: <100 ns
- **Cache**: 1-2 K bytes
- **Main Memory**: 10-20 ns
- **Tertiary Storage**: 10-100 ms

**Cost**
- **CPU Registers**: $0.01/bit
- **Cache**: $0.001/bit
- **Main Memory**: $0.0001/bit
- **Tertiary Storage**: $0.00001/bit
Modern processors use a variety of techniques for performance:

- **caches**
  - Small amount of fast memory where values are "cached" in hope of reuse by 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.

---

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)

---

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

---

Unified vs Split Caches

- **Unified vs Separate I&D**

  - 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.

  - 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 ⇒ 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?)
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.

Simplest Cache: Direct Mapped

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

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.

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.
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.

2-way set-associative cache

Main memory

Here assume cache has 8 blocks, while memory has 32

<table>
<thead>
<tr>
<th>Fully associative</th>
<th>Direct mapped</th>
<th>Set associative (in set 0) ((12 \mod 4))</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7</td>
<td>0 1 2 3 4 5 6 7</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
</tbody>
</table>

Here assume cache has 8 blocks, while memory has 32

Diagrams

Serial:

CPU

Cache

Main Memory

Parallel:

Shared Memory

Network

Cache1

Cache2

... CacheP

CPU1

CPU2

... CPUP

Tuning for Caches

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

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
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.

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.

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 largest currently.

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.

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

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.
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

\[ d = 0 \]
\[ \text{do } i = 1, n \]
\[ j = \text{ind}(i) \]
\[ d = d + \sqrt{2} \left( x(j)^2 + y(j)^2 + z(j)^2 \right) \]
\[ \text{end do} \]

- Change loop statement to

\[ d = d + r(1,j)^2 + r(2,j)^2 + r(3,j)^2 \]

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
different types of memory in one loop iteration.

Cache Thrashing by Memory Allocation

- 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.

\[ \text{real } a(m), \text{extra}(32), b(m) \]

- block is at least 128 bytes in length

Cache Blocking

- We want blocks to fit into cache. On parallel computers we have \( p \times \text{cache} \) so that data may fit into cache on \( p \) processors, but not one.

This leads to superlinear speed up! Consider matrix-matrix multiply.

\[ \begin{align*}
\text{do } & j = 1, n \\
\text{do } & i = 1, n \\
c(i,j) &= c(i,j) + a(i,k) * b(k,j) \\
\text{end do} \\
\text{end do}
\end{align*} \]

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 (L-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

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)?

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 )

Talk about Assignment


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)
    d[i][j] = a[i][j] * c[i][j];

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

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
Multiplication \( y = y + A^*x \)

For \( i = 1:n \)
\[
\begin{align*}
&\text{for } j = 1:n \\
&\quad y(i) = y(i) + A(i,j)*x(j)
\end{align*}
\]

\[ y(i) = y(i) + A(i,:)*x(:) \]

Multiplication \( y = y + A^*x \)

(Read \( x(1:n) \) into fast memory)

(Read \( y(1:n) \) into fast memory)

For \( i = 1:n \)
\[
\begin{align*}
&\text{(Read row } i \text{ of } A \text{ into fast memory)} \\
&\quad \text{for } j = 1:n \\
&\quad \quad y(i) = y(i) + A(i,j)*x(j)
\end{align*}
\]

(Write \( y(1:n) \) back to slow memory)

\* \( m = \) number of slow memory refs = \( 3n^2 + n^2 \)
\* \( f = \) number of arithmetic operations = \( 2n^2 \)
\* \( q = f/m \approx 2 \)

Matrix-vector multiplication limited by slow memory speed

\[ C = C + A*B \]

For \( i = 1 \) to \( n \)
\[
\begin{align*}
&\text{for } j = 1 \text{ to } n \\
&\quad \text{for } k = 1 \text{ to } n \\
&\quad \quad C(i,j) = C(i,j) + A(i,k) * B(k,j)
\end{align*}
\]

\[ C(i,j) = C(i,j) + A(i,:)*B(:,j) \]

\[ C = C + A*B \text{(unblocked, or untiled)} \]

(Read \( row i \) of \( A \) into fast memory)

(Read \( C(i,j) \) into fast memory)

(Read column \( j \) of \( B \) into fast memory)

For \( k = 1 \) to \( n \)
\[
\begin{align*}
&\quad C(i,j) = C(i,j) + A(i,k) * B(k,j)
\end{align*}
\]

(Write \( C(i,j) \) back to slow memory)

\[ C(i,j) = C(i,j) + A(i,:)*B(:,j) \]

\[ C = C + A*B \text{(blocked, or tiled)} \]

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

For \( i = 1 \) to \( N \)
\[
\begin{align*}
&\text{for } j = 1 \text{ to } N \\
&\quad \text{for } k = 1 \text{ to } N \\
&\quad \quad \text{(Read block } C(i,j) \text{ into fast memory)} \\
&\quad \quad \text{(Read block } A(i,k) \text{ into fast memory)} \\
&\quad \quad \text{(Read block } B(k,j) \text{ into fast memory)}
\end{align*}
\]

\[ C(i,j) = C(i,j) + A(i,k) * B(k,j) \text{(do a matrix multiply on blocks)} \]

(Write block \( C(i,j) \) back to slow memory)

\[ C(i,j) = C(i,j) + A(i,:)*B(:,j) \]
Matrix Multiply (blocked or tiled)  

Why is this algorithm correct?

Number of slow memory references on blocked matrix multiply:

\[ m = n^2 \times \frac{N}{n} \times 3 \times \frac{n}{N} \times \frac{n}{N} \]
\[ = n^2 \times \frac{N}{n} \times 3 \times \frac{n}{N} \times \frac{n}{N} \]
\[ = \left(\frac{N}{n}\right) \times \left(\frac{n}{N}\right) \times \left(\frac{n}{N}\right) \]
\[ = \left(\frac{N}{n}\right) \times \left(\frac{n}{N}\right) \times \left(\frac{n}{N}\right) \]
\[ \approx n \times \frac{n}{N} \]

So we can improve performance by increasing the blocksize \( b \)

Can be much faster than matrix-vector multiply (\( q = 2 \))

Limit: All three blocks from A, B, C must fit in fast memory (cache), so we cannot make these blocks arbitrarily large:

\[ 3b^2 \leq M \]
\[ \Rightarrow q \approx b \leq \sqrt{\frac{M}{3}} \]

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

BLAS for Performance

<table>
<thead>
<tr>
<th>Order of vectors/matrices</th>
<th>Mflop/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 1 BLAS</td>
<td></td>
</tr>
<tr>
<td>Level 2 BLAS</td>
<td></td>
</tr>
<tr>
<td>Level 3 BLAS</td>
<td></td>
</tr>
<tr>
<td>Intel 1863</td>
<td></td>
</tr>
<tr>
<td>Alpha EV 5/6 500MHz (GHz)peak)</td>
<td></td>
</tr>
</tbody>
</table>

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

Matrix Multiply (continued)

\[ \text{Consider a } 2 \times 2 \text{ matrix multiply, normally } 8 \text{ multiplies} \]
\[ \begin{align*}
    a_{11} & = (a_{11} + a_{22}) \times (b_{11} + b_{22}) \\
    a_{12} & = (a_{12} + a_{21}) \times (b_{11} + b_{22}) \\
    a_{21} & = (a_{11} + a_{22}) \times (b_{11} + b_{22}) \\
    a_{22} & = (a_{12} + a_{21}) \times (b_{11} + b_{22})
\end{align*} \]

Then:

\[ m_{11} = p_1 + p_2 - p_5 + p_7 \]
\[ m_{12} = p_4 + p_5 \]
\[ m_{21} = p_1 + p_4 \]
\[ m_{22} = p_2 + p_3 + p_5 - p_7 \]

Extends known by divide & conquer

Algebra Subroutines

- Industry standard interface (evolving)
- Vendors, others supply optimized implementations
- History
  - BLAS1 (1970s):
    - Vector operations: dot product, saxpy, etc
  - BLAS2 (mid 1980s):
    - Matrix vector operations: matrix vector multiply, etc
  - BLAS3 (late 1980s):
    - Matrix operations: matrix matrix multiply, etc

Good algorithms used BLAS3 when possible

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

Available in several libraries

Up to several times 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 \( O(n^{2.376}) \)
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 Hierarch

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 \(f(\text{ops})\):
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
Flops/Memory Refs
Level 1
\(y = y + ax\)
\(3n^2 \quad 2n^{2/3}\)
Level 2
\(y = y + Ax\)
\(n^2 \quad 2n\)
Level 3
\(C = C + AB\)
\(4n^{2/3} \quad 2n^{3/2}\)
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

Main memory: arbitrary size
Data packet size: 32 doublewords
Access time: 2.292 cycles
Cache: 32,768 doublewords
Data packet size: 1 doubleword
Access time: 0 cycles
32 FP registers
Data packet size: 1 doubleword
Access time: 0 cycles
CPU

Design your program for optimal spatial and
temporal data locality!
Effect of Stride and Array Size on Access Time

Optimal Data Locality: Data Structures

dimension r(n), f(n), t(n)
do 100 j=1,n
   r(j) = (r(j-1)-r(j-2))**2 + (t(j)-t(j-2))**2 +... 
   if (dist(i,e.cut-off)) 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 j=1,n
   r(1,j) = (r(1,j-1)-r(1,j-2))**2 + (t(1,j)-t(1,j-2))**2 +... 
   if (dist(i,e.cut-off)) 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

Instruction Level Parallelism: Floating Point

Instruction Level Parallelism: Floating Point

Code Restructuring for On-Chip Parallelism: Original Code

Code Restructuring for On-Chip Parallelism: Original Code

Modified Code for On-Chip Parallelism

Software Pipelining

Software Pipelining
Improving Ratio of Floating Point Operations to Memory Accesses

```fortran
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
    t=t+a(j,i)*x(j)
  10    y(i)=t
  return
end
```

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.

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:

\[
\begin{align*}
t_s &= \frac{t}{t_s + t_p} \quad \text{Effect of multiple processors on run time} \\
S &= \frac{1}{\frac{t}{t_s} + \frac{t_p}{N}} \quad \text{Effect of multiple processors on speedup}
\end{align*}
\]

Where:

- \( t_s \) = serial fraction of code
- \( t_p \) = parallel fraction of code = 1 - \( t_s \)
- \( N \) = number of processors
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.

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

More on Amdahl’s Law

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

Parallel Performance Metrics: Speedup

- 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: \( S_A(n) = \frac{T(1)}{T(n)} = \frac{1}{n \times (1 - f)} \)
- This bounds the speedup based only on the fraction of the code that cannot use parallelism (1 - f); it ignores all other factors
- \( S_A \rightarrow 1 / (1 - f) \) as \( n \rightarrow \infty \)
Fixed-Problem Size Scaling: Examples

Scaled Speedup Experiments

- 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.

Example of a Scaled Speedup Experiment

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

Suggested alternate approach (Gustafson): Constant time benchmarks
- run code for a fixed time and measure work done

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

TBON on ASCI Red

Performance Optimization for Intensive Codes

Status Quo/Parallel Trends