

Lecture 5: Memory Hierarchy and Cache

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

1

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)

2

Cache-Related Terms

- ◆ **ICACHE** : Instruction cache
- ◆ **DCACHE (L1)** : Data cache closest to registers
- ◆ **SCACHE (L2)** : Secondary data cache
- ◆ **TCACHE (L3)** : Third level data cache
 - Data from SCACHE has to go through DCACHE to registers
 - TCACHE is larger than SCACHE, and SCACHE is larger than DCACHE
 - Not all processors have TCACHE

3

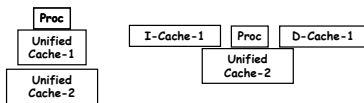
Unified versus Split Caches

- ◆ 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..

4

Unified vs Split Caches

◆ Unified vs Separate I&D



◆ Example:

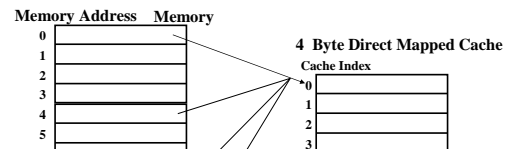
- 16KB I&D: Inst miss rate=0.64%, Data miss rate=6.47%
- 32KB unified: Aggregate miss rate=1.99%

◆ Which is better (ignore L2 cache)?

- Assume 33% data ops ⇒ 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)

5

Simplest Cache: Direct Mapped

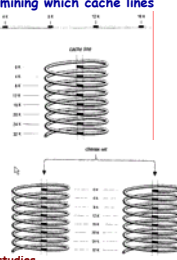


- ◆ Location 0 can be occupied by data from:
 - Memory location 0, 4, 8, ... etc.
 - In general: any memory location whose 2 LSBs of the address are 0s
 - Address<1:0> => cache index
- ◆ Which one should we place in the cache?
- ◆ How can we tell which one is in the cache?

6

Cache Mapping Strategies

- There are two common sets of methods in use for determining which cache lines are used to hold copies of memory lines.
- 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 to replace:
 - Random:** Choose any line using an analog random number 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.



7

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)

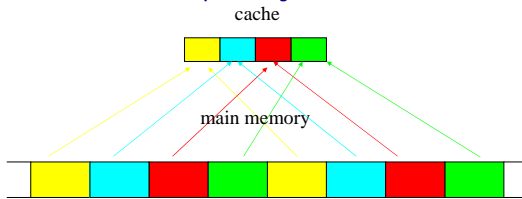


- Cache line length:** number of bytes loaded together in one entry
- Direct mapped:** only one address (line) in a given range in cache
- Associative:** 2 or more lines with different addresses exist

8

Direct-Mapped Cache

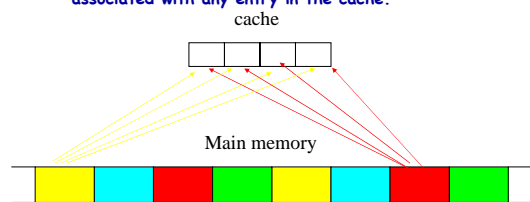
- Direct mapped cache:** A block from main memory can go in exactly one place in the cache. This is called direct mapped because there is direct mapping from any block address in memory to a single location in the cache.



9

Fully Associative Cache

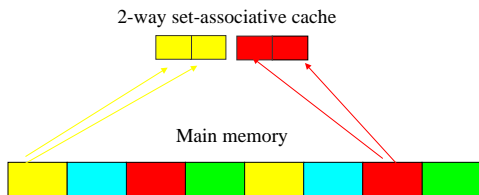
- Fully Associative Cache:** A block from main memory can be placed in any location in the cache. This is called fully associative because a block in main memory may be associated with any entry in the cache.



10

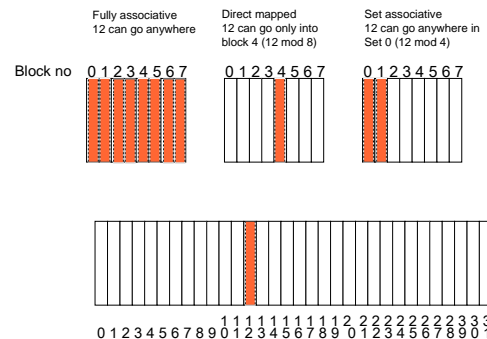
Set Associative Cache

- 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.



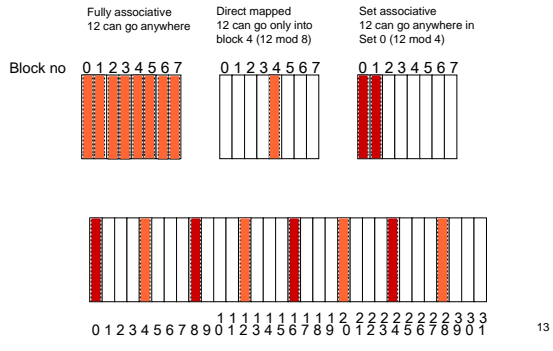
11

Here assume cache has 8 blocks, while memory has 32

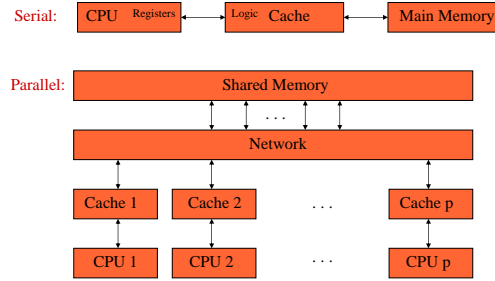


12

Here assume cache has 8 blocks, while memory has 32



Diagrams



Tuning for Caches

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

15

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 subsystem.

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 (e.g., straightline code, array access)
- ◆ Last 15 years, HW relied on locality for speed

17

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.

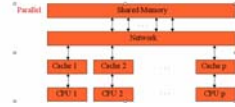
18

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 matrices.
- ◆ Machine architects can add sets to the associativity. Users can buy another vendor's machine. However, neither solution is realistic.

19

Cache Coherence for Multiprocessors



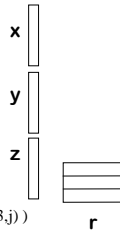
- ◆ All data must be coherent between memory levels. Multiple processors with separate caches must inform the other processors quickly about data modifications (by the cache line). Only hardware is fast enough to do this.
- ◆ Standard protocols on multiprocessors:
 - > Snoopy: all processors monitor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

20

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.

21

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, which is longer than a cache line on all but one memory subsystem that is available today.

22

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
    
```

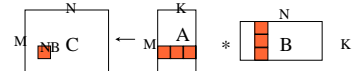
- ◆ An alternate form is ...

23

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
      end do
    end do
  end do
end do
    
```



24

Summary :

The Cache Design Space

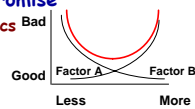
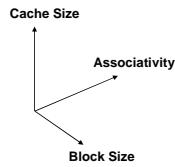
- ◆ Several interacting dimensions

- 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



25

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**

26

Optimizing Matrix 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)?

27

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

28

Optimizing Matrix Multiply for Caches

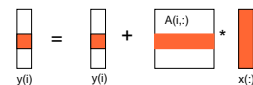
- ◆ Several techniques for making this faster on modern processors
 - heavily studied
- ◆ Some optimizations done automatically by compiler, but can do much better
- ◆ In general, you should use optimized libraries (often supplied by vendor) for this and other very common linear algebra operations
 -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

29

Warm up: Matrix-vector multiplication $y = y + A*x$

```

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



30

Warm up: Matrix-vector 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 = $2*n^2$
- ° $q = f/m \approx 2$
- ° Matrix-vector multiplication limited by slow memory speed

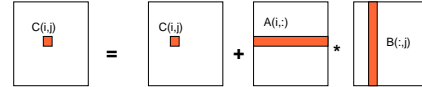
31

Multiply $C=C+A*B$

```

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)

```



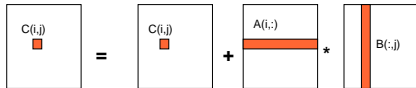
32

Matrix Multiply $C=C+A*B$ (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}

```



33

Matrix Multiply (unblocked, or untiled)

$q = \text{ops/slow mem ref}$

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



34

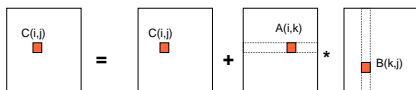
Matrix Multiply (blocked, or tiled)

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}

```



35

Matrix Multiply (blocked or tiled)

$q = \text{ops/slow mem ref}$

Why is this algorithm correct?

n size of matrix
 b blocksize
 N number of blocks

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 $q = f/m = 2*n^3 / ((2*N + 2)*n^2)$
 $\approx n/N = b$ for large 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: $3*b^2 \leq M$, so $q \approx b \leq \sqrt{M/3}$

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

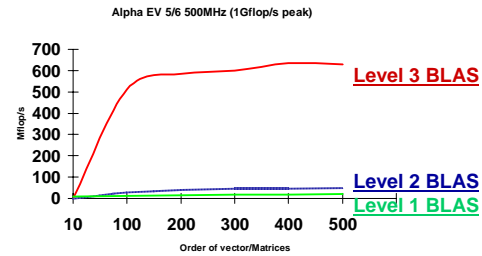
36

More on BLAS (Basic Linear Algebra Subroutines)

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

37

BLAS for Performance



- ◆ **Development of blocked algorithms important 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)

38

Optimizing in 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/hipac
 - www.cs.berkeley.edu/~iyer/ascii_slides.ps
 - ATLAS: www.netlib.org/atlas/index.html

39

Strassen's 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 $M = \begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$

Let $p_1 = (a_{11} + a_{22}) * (b_{11} + b_{22})$ $p_5 = (a_{11} + a_{12}) * b_{22}$
 $p_2 = (a_{21} + a_{22}) * b_{11}$ $p_6 = (a_{21} - a_{11}) * (b_{11} + b_{12})$
 $p_3 = a_{11} * (b_{12} - b_{22})$ $p_7 = (a_{12} - a_{22}) * (b_{21} + b_{22})$
 $p_4 = a_{22} * (b_{21} - b_{11})$

Then $m_{11} = p_1 + p_4 - p_5 + p_7$ Extends to nxn by divide&conquer
 $m_{12} = p_3 + p_5$
 $m_{21} = p_2 + p_4$
 $m_{22} = p_1 + p_3 - p_2 + p_6$

40

Strassen (continued)

$$\begin{aligned}
 T(n) &= \text{Cost of multiplying } nxn \text{ matrices} \\
 &= 7 * T(n/2) + 18 * (n/2)^2 \\
 &= O(n^{\log_2 7}) \\
 &= O(n^{2.81})
 \end{aligned}$$

- 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 $O(n^{2.376..})$

41

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)**

42

Summary: Memory Hierachy

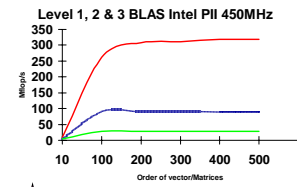
- Virtual memory was controversial at the time: can SW automatically manage 64KB across many programs?
 - > 1000X DRAM growth removed the controversy
- Today VM allows many processes to share single memory without having to swap all processes to disk; **today VM protection is more important than memory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

43

Performance = Effective Use of Memory Hierachy

- 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 $y=y+ax$	$3n$	$2n$	$2/3$
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

44

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

45

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

46

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:

$$t_N = (f_p/N + f_s)t_1 \quad \text{Effect of multiple processors on run time}$$

$$S = 1/(f_s + f_p/N) \quad \text{Effect of multiple processors on speedup}$$

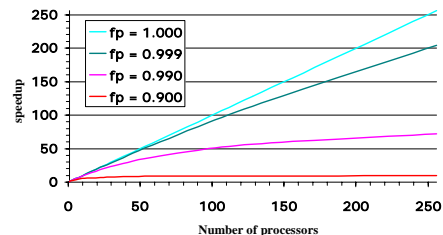
Where:

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

47

Illustration of Amdahl's Law

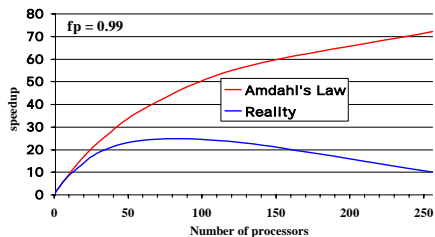
It takes only a small fraction of serial content in a code to degrade the parallel performance. It is essential to determine the scaling behavior of your code before doing production runs using large numbers of processors



48

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.



49

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_{memory} :
 - The growing processor-memory performance gap will undermine our efforts at achieving maximum possible speedup!

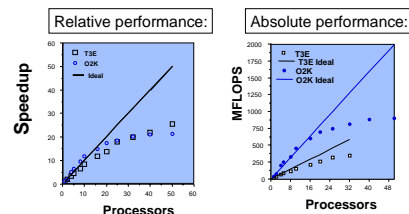
50

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

51

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.

52

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
- $S_A \rightarrow 1 / (1-f)$ as $n \rightarrow \infty$

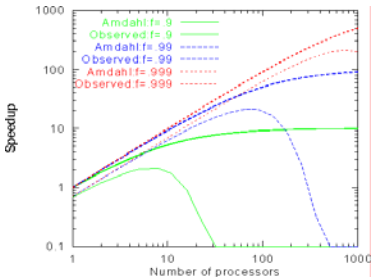
53

Fixed-Problem Size Scaling (Cont'd)

- Efficiency $(\eta) = T(1) / [T(n) * n]$
- 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

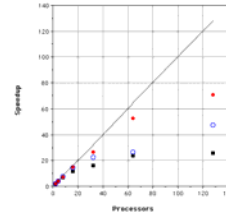
54

Fixed-Problem Size Scaling: Examples



S. Goedecker and Adolfo 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.*

Fixed-Problem Size Scaling Examples



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.

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

Example of a Scaled Speedup Experiment

Processors	NChains	Time	Natoms	Time per PE 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

