Lecture 8: Memory Hierarchy and Cache

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

First hour of today’s class will be in the CS Seminar.
Dr. Kirk Sayre will present a seminar on Wednesday, March 8, 2006, at 1:30 p.m. in Claxton Complex 206.

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.
Standard Uniprocessor Memory Hierarchy

- **Intel Pentium 4 2 GHz processor**
- **P7 Prescott 478**
  - 8 Kbytes of 4 way assoc. L1 instruction cache with 32 byte lines.
  - 8 Kbytes of 4 way assoc. L1 data cache with 32 byte lines.
  - 256 Kbytes of 8 way assoc. L2 cache 32 byte lines.
  - 400 MB/s bus speed
  - SSE2 provide peak of 4 Gflop/s
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?

Latency in a Single System

- DRAM 9%/yr.
  - (2X/10 yrs)
- µProc 60%/yr.
  - (2X/1.5yr)

Processor-Memory Performance Gap:
- (grows 50% / year)
Commodity Processor Trends

Bandwidth/Latency is the Critical Issue, not FLOPS

<table>
<thead>
<tr>
<th></th>
<th>Annual increase</th>
<th>Typical value in 2005</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-chip floating-point performance</td>
<td>59%</td>
<td>4 GFLOP/s</td>
</tr>
<tr>
<td>Front-side bus bandwidth</td>
<td>23%</td>
<td>1 GWord/s</td>
</tr>
<tr>
<td>DRAM latency</td>
<td>(5.5%)</td>
<td>70 ns</td>
</tr>
<tr>
<td></td>
<td></td>
<td>= 280 FP ops</td>
</tr>
<tr>
<td></td>
<td></td>
<td>= 70 loads</td>
</tr>
</tbody>
</table>


Got Bandwidth?

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
- **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

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..
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)

Simplest Cache: Direct Mapped

- **Memory Address** → **Memory**

  ![Diagram of 4 Byte Direct Mapped Cache]

- 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?
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.

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)
<table>
<thead>
<tr>
<th>Address</th>
<th>tag</th>
<th>line</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>X0010</td>
<td>X001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X010</td>
<td>X011</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X100</td>
<td>X101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X110</td>
<td>X111</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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.

![Diagram of Direct-Mapped 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.

![Diagram of Fully Associative 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
Here assume cache has 8 blocks, while memory has 32

- **Fully associative**: 12 can go anywhere
- **Direct mapped**: 12 can go only into block 4 (12 mod 8)
- **Set associative**: 12 can go anywhere in Set 0 (12 mod 4)

---

Diagrams

**Serial:**
- CPU
- Registers
- Logic
- Cache
- Main Memory

**Parallel:**
- Shared Memory
- Network
- Cache 1
- Cache 2
- Cache p
- CPU 1
- CPU 2
- CPU p
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 subsystem.
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**

Principalps 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.**
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.

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.
Indirect Addressing

\[
d = 0
\]

\[
d = d + \sqrt{ x(j)^2 + y(j)^2 + z(j)^2 } 
\]

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

\* Change loop statement to \*

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

\* 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 cache lines at every instance of the loop and can cause cache thrashing.

\[
\text{Cache Thrashing by Memory Allocation}
\]

\[
\text{parameter ( } m = 1024^2 \text{ )}
\]

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

\* 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.

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

\* 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.
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.

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

- An alternate form is ...

```
```

Cache Blocking

\[
\begin{align*}
\text{do } & kk = 1, n, nblk \\
\text{do } & jj = 1, n, nblk \\
\text{do } & ii = 1, n, nblk \\
\text{do } & k = kk, kk+nblk-1 \\
\text{do } & j = jj, jj+nblk-1 \\
\text{do } & i = ii, ii+nblk-1 \\
& c(i,j) = c(i,j) + a(i,k) * b(k,j) \\
\text{end do} \\
\text{end do} \\
\text{end do} \\
\text{end do}
\end{align*}
\]
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 (I-cache, D-cache, TLB)
  - depends on technology / cost

- **Simplicity often wins**

![Diagram showing cache design space with dimensions cache size, associativity, and block size.]

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
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)?
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

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

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

{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
Multiply

\[ C = C + A \times B \]

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

Matrix Multiply

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

\[
\begin{align*}
  & \text{for } i = 1 \text{ to } n \\
  & \quad \{\text{read row } i \text{ of } A \text{ into fast memory}\} \\
  & \quad \text{for } j = 1 \text{ to } n \\
  & \quad \quad \{\text{read } C(i,j) \text{ into fast memory}\} \\
  & \quad \quad \{\text{read column } j \text{ of } B \text{ into fast memory}\} \\
  & \quad \quad \text{for } k = 1 \text{ to } n \\
  & \quad \quad \quad C(i,j) = C(i,j) + A(i,k) \times B(k,j) \\
  & \quad \quad \quad \{\text{write } C(i,j) \text{ back to slow memory}\}
\end{align*}
\]
Matrix Multiply
(unblocked, or untiled)

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

\[ \begin{array}{ccc}
C(i,j) & \quad + & A(i,:) \times B(:,j) \\
\end{array} \]

Matrix Multiply
(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 \( N \)
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\}
\[ C(i,j) = C(i,j) + A(i,k) \times B(k,j) \] (do a matrix multiply on blocks)
\{write block \( C(i,j) \) back to slow memory\}

\[ \begin{array}{ccc}
C(i,j) & \quad + & A(i,k) \times B(k,j) \\
\end{array} \]
Matrix Multiply (blocked or tiled)

Why is this algorithm correct?

Number of slow memory references on blocked matrix multiply

\[ m = N^2 n^2 \text{ read each block of } B \text{ } N^2 n^2 \text{ times (} N^2 n^2 \times n/N \times n/N \) \\
+ N^2 n^2 \text{ read each block of } A \text{ } N^2 n^2 \text{ times} \\
+ 2n^2 \text{ read and write each block of } C \text{ once} \\
= (2N + 2)n^2 \]

So \( q = f/m = 2n^2 / ((2N + 2)n^2) \)

\[ \approx n/N = b \text{ for large } n \]

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: \( 3b^2 \leq M \), so \( q \approx b \leq \sqrt{M/3} \)

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

---

More on BLAS (Basic Linear Algebra Subroutines)

- **Industry standard interface (evolving)**
- **Vendors, others supply optimized implementations**
- **History**
  - **BLAS1 (1970s):**
    - vector operations: dot product, saxpy \( (y = b \times x + y) \), etc
    - \( m=2^n, f=2^n, q \approx 1 \) or less
  - **BLAS2 (mid 1980s):**
    - matrix-vector operations: matrix vector multiply, etc
    - \( m=n^2, f=2n^2, q \approx 2, \) 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**
BLAS for Performance

Intel Pentium 4 w/SSE2 1.7 GHz

- Development of blocked algorithms important for performance

![Graph showing performance comparison between Level 1, Level 2, and Level 3 BLAS operations on an Intel Pentium 4 processor.](image)

BLAS for Performance

Alpha EV 5/6 500MHz (1Gflop/s peak)

- Development of blocked algorithms important for performance

![Graph showing performance comparison between Level 1, Level 2, and Level 3 BLAS operations on an Alpha EV 5/6 processor.](image)

BLAS 3 (n-by-n matrix matrix multiply) vs BLAS 2 (n-by-n matrix vector multiply) vs BLAS 1 (saxpy of n vectors)
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/phipac
  - www.cs.berkeley.edu/~iyer/asci_slides.ps
  - ATLAS: www.netlib.org/atlas/index.html

Strassen’s Matrix Multiply

- The traditional algorithm (with or without tiling) has $O(n^3)$ flops
- Strassen discovered an algorithm with asymptotically lower flops
  - $O(n^{\log_2 7})$ or $O(n^{2.81})$
- Consider a 2x2 matrix multiply, normally 8 multiplies

Let $M = \begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$

Let $p_1 = (a_{11} + a_{22}) \times (b_{11} + b_{22})$  
$p_2 = (a_{21} + a_{22}) \times b_{11}$
$p_3 = a_{11} \times (b_{12} - b_{22})$
$p_4 = a_{22} \times (b_{21} - b_{11})$

Then $m_{11} = p_1 + p_4 - p_5 + p_7$
$m_{12} = p_3 + p_5$
$m_{21} = p_2 + p_4$
$m_{22} = p_1 + p_3 - p_2 + p_6$

Extends to nxn by divide & conquer
Strassen
(continued)

\[ T(n) = \text{Cost of multiplying } n \times n \text{ matrices} \]
\[ = 7 \times T(n/2) + 18 \times (n/2)^2 \]
\[ = O(n^{\log_2 7}) \]
\[ = O(n^{2.81}) \]

° 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 \( O(n^{2.376..}) \)

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 Hierarchy

- 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

<table>
<thead>
<tr>
<th>BLAS</th>
<th>Memory Refs</th>
<th>Flops</th>
<th>Flops/ Memory Refs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 1</td>
<td>3n</td>
<td>2n</td>
<td>2/3</td>
</tr>
<tr>
<td>(y = y + \alpha x)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Level 2</td>
<td>(n^2)</td>
<td>2n^2</td>
<td>2</td>
</tr>
<tr>
<td>(y = y + Ax)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Level 3</td>
<td>(4n^3)</td>
<td>2n^5</td>
<td>(n/2)</td>
</tr>
<tr>
<td>(C = C + AB)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Development of blocked algorithms important for performance
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
```

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

---

c works correctly when n1,n2 are multiples of 4

dimension a(nd1,nd2), y(nd2), x(nd1)

```fortran
do 1=1,n1-3,4
t1=0.d0
t2=0.d0
t3=0.d0
t4=0.d0
do 1=1,n2-3,4
   t1=t1+a(j+0,i+0)*x(j+0)+a(j+1,i+0)*x(j+1)+
      a(j+2,i+0)*x(j+2)+a(j+3,i+0)*x(j+3)
   t2=t2+a(j+0,i+1)*x(j+0)+a(j+1,i+1)*x(j+1)+
      a(j+2,i+1)*x(j+2)+a(j+3,i+1)*x(j+3)
   t3=t3+a(j+0,i+2)*x(j+0)+a(j+1,i+2)*x(j+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)+
      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**
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:

\[ t_N = \left( \frac{f_p}{N} + f_s \right) t_1 \]  
Effect of multiple processors on run time

\[ S = \frac{1}{f_s + f_p/N} \]  
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 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 $f_{\text{processor}}$ and $f_{\text{memory}}$:
  - 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

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

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

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

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.

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

<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>125400</td>
<td>2.66E-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>303600</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>

TBON on ASCI Red

Performance Optimization of Numerically intensive Codes

Stefan Gödecker
Adolfy Hoisie