Outline

• Time Scaling
• Examples
• General performance strategies
• Computer architectures
• Measuring code speed
  – Reduce strength
  – Minimize array writes
• Profiling
## Time Scaling for Algorithms

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>Log(N)</td>
<td>Search binary tree</td>
</tr>
<tr>
<td>N</td>
<td>Iterate over N elements</td>
</tr>
<tr>
<td>N \log(N)</td>
<td>Quicksort; mergesort; FFT</td>
</tr>
<tr>
<td>N^2</td>
<td>Allocate N x N (2D) array</td>
</tr>
<tr>
<td>N^3</td>
<td>Matrix operations (Multiply / invert matrices)</td>
</tr>
<tr>
<td>2^N</td>
<td>Towers of Hanoi</td>
</tr>
<tr>
<td>N!</td>
<td>Traveling salesman problem</td>
</tr>
</tbody>
</table>
Binary Tree Search

Successful (left) and unsuccessful (right) search in a BST

- **Successful search for R**
  - Black nodes could match the search key
  - R is greater than E, so look to the right
  - Found R (search hit) so return value

- **Unsuccessful search for T**
  - Gray nodes cannot match the search key
  - T is greater than S, so look to the right
  - T is less than X, so look to the left
  - Link is null, so T is not in tree (search miss)
Quicksort

Starting array

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>23</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>77</th>
<th>33</th>
</tr>
</thead>
<tbody>
<tr>
<td>44</td>
<td>75</td>
<td>23</td>
<td>43</td>
<td>55</td>
<td>12</td>
<td>64</td>
<td>77</td>
<td>33</td>
<td></td>
</tr>
</tbody>
</table>

Partition

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th>43</th>
<th>55</th>
<th>44</th>
<th>64</th>
<th>77</th>
<th>75</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>23</td>
<td>33</td>
<td>43</td>
<td>55</td>
<td>44</td>
<td>64</td>
<td>77</td>
<td>75</td>
<td></td>
</tr>
</tbody>
</table>

Quicksort-left, Partition

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>23</td>
</tr>
</tbody>
</table>

Quicksort-right, Partition

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>43</td>
<td>55</td>
<td>44</td>
<td>64</td>
<td>77</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>55</td>
<td>44</td>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Resulting array

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>23</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>77</th>
<th>33</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>23</td>
<td>33</td>
<td>43</td>
<td>44</td>
<td>55</td>
<td>64</td>
<td>75</td>
<td>77</td>
<td></td>
</tr>
</tbody>
</table>
Matrix Multiplication

\[ C[i][j] = \sum_{k=1}^{n} A[i][k] \times B[k][j] \]
Towers of Hanoi
General Performance Strategies

• Choose the best algorithm for each job

• Consider Ahmdal’s law:

\[
\text{Speedup} = \frac{t_{\text{old}}}{t_{\text{new}}} = \frac{1}{(1 - f) + f/N}
\]

  – Make common case fastest (e.g., eliminate unnecessary steps at inner loops)
  – Make rare case correct (e.g., avoid mistakes in less frequent functions)
Ahmdal’s law

\[ f = 0.01 \]

\[ f = 0.05 \]

\[ f = 0.2 \]

\[ f = 0.5 \]

San Diego Supercomputer Center, Summer Institute Tutorial
Gustafson’s Law

Instead of running the same size problem for all $N$, we can also consider running larger problems with better code or greater resources, which leads to Gustafson’s law.
Testing Gustafson’s and Amdahl’s Law

Garrison Prinslow (Washington University-St. Louis), Overview of Performance Measurement for Multicore Processors
General Performance Strategies

• Minimize jumps/branches
• Consider array index order and operation choices
• Reduce local variables/parameters (especially arrays)
• Minimize dynamic memory allocation
General Performance Strategies

• A few good, but less known, things to do for fast code:
  – Prefer static linking and position-dependent code (as opposed to PIC, position-independent code).
  – Prefer 64-bit code and 32-bit data.
  – Prefer array indexing to pointers
  – Prefer regular memory access patterns.
  – Minimize control flow.
  – Avoid data dependencies.
Computer Architectures

• Old paradigm: predictable performance (clock frequency, each operation takes a fixed number of cycles)

• New paradigm: statistically optimal performance (deep cache hierarchies, pipelines, speculative execution)

• Implication: must now **measure** all important operations
Computer Architectures

Chip Level Architecture

- **Ring Interconnect:**
  - Dedicated “stops”: each CPU Core, Graphics, & System Agent
  - Bi directional, 32 Bytes wide

- **Shared Last Level Cache (LLC):**
  - Both Graphics & CPU cores
  - Cache slices act like a single monolithic albeit distributed cache
  - 2-8MB, depending on product
Measuring Code Speed

Accurate benchmarking requires:

– Not measuring the speed of debug builds.
– Using the baseline and the benchmarked code under the same conditions.
– Not including ancillary work in measurement, particularly in an imbalanced fashion.
– Focusing on common cases instead of rare ones.
Reduce Strength

• When implementing an algorithm, use operations of the minimum strength possible.
• The speed hierarchy of operations is (fast to slow):
  – comparisons
  – \((u)\text{int}\) add, subtract, bitops, shift
  – floating point add, sub (separate unit!)
  – indexed array access (caveat: cache effects)
  – \((u)\text{int32}\) mul
  – FP mul
  – FP division, remainder
  – \((u)\text{int}\) division, remainder
Reduce Strength: digits\textsubscript{10}

Basic algorithm using division

```c
uint32_t digits10(uint64_t v) {
    uint32_t result = 0;
    do {
        ++result;
        v /= 10;
    } while (v);
    return result;
}
```

Reduced strength algorithm using comparison with division fallback

```c
uint32_t digits10(uint64_t v) {
    uint32_t result = 1;
    for (;;) {
        if (v < 10) return result;
        if (v < 100) return result + 1;
        if (v < 1000) return result + 2;
        if (v < 10000) return result + 3;
        v /= 10000U;
        result += 4;
    }
}
```
Reduce Strength: digits10 relative speedup

Data from Andrei Alexandrescu (Facebook)
Minimize Array Writes

• To be faster, code should reduce the number of array writes, and more generally, writes through pointers.
• On modern machines having large register files and ample register renaming hardware, most named individual variables (numbers, pointers) end up sitting in registers – which are fast
• Avoid array writes wherever possible: array operations (and other indirect accesses) are less natural across the entire compiler-processor-cache hierarchy because:
  – Array accesses are not registered
  – Whenever pointers are involved, the compiler must assume the pointers could point to global data, meaning any function call may change pointed-to data arbitrarily.
  – Array writes are the worst: writing one word to memory is essentially a cache line read followed by a cache line write.
Minimize Array Writes: Integer to String Conversion

**Standard algorithm using reversal**

```c
uint32_t u64ToAsciiClassic(uint64_t value, char* dst) {
    // Write backwards.
    auto start = dst;
    do {
        *dst++ = '0' + (value % 10);
        value /= 10;
    } while (value != 0);
    const uint32_t result = dst - start;
    // Reverse in place.
    for (dst--; dst > start; start++, dst--) {
        std::iter_swap(dst, start);
    }
    return result;
}
```

**Revised algorithm with less array writes**

```c
uint32_t uint64ToAscii(uint64_t v, char *const buffer) {
    auto const result = digits10(v);
    uint32_t pos = result - 1;
    while (v >= 10) {
        auto const q = v / 10;
        auto const r = static_cast<uint32_t>(v % 10);
        buffer[pos--] = '0' + r;
        v = q;
    }
    assert(pos == 0); // Last digit is trivial to handle
    *buffer = static_cast<uint32_t>(v) + '0';
    return result;
}
```
Minimize Array Writes: Integer to String Conversion Relative Speedup

Data from Andrei Alexandrescu (Facebook)
Code Optimization Links

• Basic concepts for C/C++ code optimization:
  http://www.eventhelix.com/realtimemantra/basics/optimizingcandcppcode.htm

• Vectorization on modern Intel Core processors:

• Optimized subroutines in C/C++ (advanced):
  http://www.agner.org/optimize/
Profiling with gprof

• gprof is a profiling tool for UNIX/Linux applications. First developed in 1982, it is still extremely popular and widely used.

• Universally supported by all major C/C++ and Fortran compilers

• Extremely easy to use
  – Compile code with -pg option: adds instrumentation to executable
  – Run application: file named gmon.out will be created.
  – Run gprof to generate profile: gprof a.out gmon.out

• Introduces virtually no overhead

• Output is easy to interpret
## Flat Profile View

<table>
<thead>
<tr>
<th>%time</th>
<th>cumulative seconds</th>
<th>self seconds</th>
<th>calls</th>
<th>total ms/call</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>84.2</td>
<td>36.48</td>
<td>36.48</td>
<td>1</td>
<td>36480.00</td>
<td>tuned.c</td>
</tr>
<tr>
<td>8.7</td>
<td>40.24</td>
<td>3.76</td>
<td>1</td>
<td>3760.00</td>
<td>tuned.c</td>
</tr>
<tr>
<td>4.6</td>
<td>42.23</td>
<td>1.99</td>
<td>1</td>
<td>1990.00</td>
<td>tuned.c</td>
</tr>
<tr>
<td>2.1</td>
<td>43.14</td>
<td>0.91</td>
<td>1</td>
<td>70.00</td>
<td>tuned.c</td>
</tr>
<tr>
<td>0.2</td>
<td>43.21</td>
<td>0.07</td>
<td>1</td>
<td>70.00</td>
<td>tuned.c</td>
</tr>
<tr>
<td>0.2</td>
<td>43.28</td>
<td>0.07</td>
<td></td>
<td></td>
<td>dtrsa.f</td>
</tr>
<tr>
<td>0.1</td>
<td>43.31</td>
<td>0.03</td>
<td></td>
<td></td>
<td>dtrca [9]</td>
</tr>
<tr>
<td>0.0</td>
<td>43.31</td>
<td>0.00</td>
<td>50000</td>
<td>0.00</td>
<td>sin [10]</td>
</tr>
<tr>
<td>0.0</td>
<td>43.31</td>
<td>0.00</td>
<td>64</td>
<td>0.00</td>
<td>pthread_self [11]</td>
</tr>
<tr>
<td>0.0</td>
<td>43.31</td>
<td>0.00</td>
<td>1</td>
<td>0.00</td>
<td>main [1]</td>
</tr>
<tr>
<td>0.0</td>
<td>43.31</td>
<td>0.00</td>
<td></td>
<td></td>
<td>_start [2]</td>
</tr>
</tbody>
</table>

Search Engine: (regular expressions supported)

|      |
### Call Graph

#### Call Graph Profile

<table>
<thead>
<tr>
<th>index</th>
<th>$%$time</th>
<th>self descendants</th>
<th>called/total called/self called/total parents</th>
<th>name index</th>
<th>children</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>99.0</td>
<td>0.01</td>
<td>17.54 1/1</td>
<td>__start [2]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.01</td>
<td>17.54</td>
<td>1</td>
<td>qmf2 [1]</td>
<td>qmf2 [1]</td>
</tr>
<tr>
<td></td>
<td>6.07</td>
<td>0.00</td>
<td>101/101 (.xsolve [3])</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>5.72</td>
<td>0.00</td>
<td>101/101 (.ysolve [4])</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.81</td>
<td>0.00</td>
<td>101/101 (.yrhs [5])</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.42</td>
<td>0.00</td>
<td>101/101 (.xrhs [6])</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.36</td>
<td>0.05</td>
<td>1/1</td>
<td>(.setup\textunderscore matrix [7])</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.03</td>
<td>0.08</td>
<td>1/1</td>
<td>(.init_psi [9])</td>
<td></td>
</tr>
</tbody>
</table>

---

| [2]   | 99.0     | 0.00             | 17.55 1/1         | __start [2]|          |
|       | 0.01     | 17.54            | 1                  | qmf2 [1]    | qmf2 [1] |

---

| [3]   | 34.2     | 6.07             | 0.00 101/101      | qmf2 [1]    | \(<\text{spontaneous}>\) |
|       | 6.07     | 0.00             | 101 \(.xsolve [3]\) |

---

| [4]   | 32.3     | 5.72             | 0.00 101/101      | qmf2 [1]    | \(<\text{spontaneous}>\) |
|       | 5.72     | 0.00             | 101 \(.ysolve [4]\) |

---

| [5]   | 15.8     | 2.81             | 0.00 101/101      | qmf2 [1]    | \(<\text{spontaneous}>\) |
|       | 2.81     | 0.00             | 101 \(.yrhs [5]\) |

---

| [6]   | 13.6     | 2.42             | 0.00 101/101      | qmf2 [1]    | \(<\text{spontaneous}>\) |
|       | 2.42     | 0.00             | 101 \(.xrhs [6]\) |
Limitations of gprof

• gprof only measures time spent in user-space code; does not account for system calls or time waiting for CPU or I/O
• gprof has limited utility for threaded applications (e.g. parallelized using OpenMP or Pthreads) and will normally only report usage for thread 0
• gprof can be used for MPI applications and will generate a gmon.out.id file for each MPI process. But it will not give an accurate picture of the time spent waiting for communications
• gprof will not report usage for un-instrumented library routines
• By default, gprof only gives function level rather than statement level profile information (requires debug mode + gprof –l). But once a function has been identified as a hotspot, it’s usually obvious where the time is being spent (e.g. statements in innermost loop)
MATLAB profiler

profile on
profile clear
calcRate(Y(1:75000),N(1:75000));
profreport('calcRate')

Self time is the time spent in a function excluding the time spent in its child functions. Self time also includes overhead resulting from the process of profiling.
MATLAB Optimization

• MATLAB’s Techniques for Improving Performance
  – http://www.mathworks.com/help/techdoc/matlab prog/f8-784135.html#f8-793781

• MATLAB’s What things can I do to increase the speed and memory performance of my MATLAB code?

• Improving the Speed of MATLAB Calculations
Next Class

- Formulating eigenproblems for electro-optic systems
- See Joannopoulos, Chapter 2 and Appendix D