Why is performance important?

Acceptable response time
Ability to add more functionality
Ability to scale
Use less power / resources
Acceptable response times

Many systems have stringent requirements

- Anti-lock break system → response time ≤ hydraulic system
- Mpeg decoder → > 20 frames a second or will get a jittery movie
- Google Search → results should be available within a second

If the response times are not met, system is not usable
Ability to add more functionality

If the necessary work can be done faster, more room within the required response time can lead to:

→ added features
→ higher-quality processing
→ bigger data sets
→ competitive advantage!
Ability to scale

Successful programs will get pushed hard

- From hundred to millions of users/documents/data
  - Scale the system to handle the increased workload
  - Gracefully deal with unexpected issues due to scaling
Use less power and resources

More instruction executed \( \rightarrow \) more power used

- In 2005 approx. 1.2% US total power went to computer servers.
  - Many supercomputer centers are now power limited.
- The viability of cell phones is dictated by battery life.
  - An iPhone lasts less than a day on one charge.
    More functionality cannot be added without improving the battery life
    or making the existing applications more efficient.

The cost of scaling an inefficient system is high

- Adding servers is expensive.
Improving performance is hard

Knowing that there is a performance problem
Identifying the performance bottlenecks
Establishing the main causes of the problem
Eliminating them
Knowing that there is a problem

We know how to find incorrect programs
→ testing, verification, and validation

But how close is your program to its maximum achievable performance?

➤ Sometimes hard to know whether program performance can improve much.

How do you know if your program is performing well?

➤ Back of the envelope calculations
➤ Performance debugging
➤ Scalability testing
➤ Comparisons to similar programs
➤ Experience!
Identifying performance bottlenecks

Profile the programs
- Figure out in what lines of code the program spends most of its time.
  - Is that expected? Or is there a problem?
- Look at machine characteristics
  - Are instruction, cache, memory, and IO behavior normal?
  - A complex system with a small window in which to look.

Scalability testing
- Push the program to the limit
- Often must do with limited resources
- Need to understand what will scale and what will not

Measure without Perturbing
- If profiling change the performance too much, results are not valid
Establish the leading cause

Study the algorithm

- Is the algorithm too costly?
- Can any frequently executing subcomputation be eliminated?
  - Preprocessing, caching, etc.

Study data structures and data layout

- Is how the data is laid out in memory affecting program behavior?

Study the program structure

- Are their known inefficiencies in the way the application is coded?

Trial-and-error

- Many hunches may not work out at first, even if the intuition is correct!
- What you think is a performance bottleneck may be hidden by a larger effect you weren’t aware of.
- Persistence and detective work.
Eliminating performance problems

**Performance cuts through abstraction boundaries**
- Performance engineering must be performed holistically ("end to end").

**Need to understand the abstraction layers and their impact on overall performance**
- All the application layers
- The compiler
- The processor
- The operating system
- The network
- Etc.

**Adhere to good software-engineering principles**
- Simplicity, modularity, portability, etc.
- Do not compromise correctness!
Matrix Multiplication

PERFORMANCE ENGINEERING: CASE STUDY
Matrix multiplication is a fundamental operation in many computations.

Examples: video encoding, weather simulation, computer graphics

\[
\begin{pmatrix}
c_{11} & c_{12} & \cdots & c_{1n} \\
c_{21} & c_{22} & \cdots & c_{2n} \\
\vdots & \vdots & \ddots & \vdots \\
c_{n1} & c_{n2} & \cdots & c_{nn}
\end{pmatrix} = \begin{pmatrix}
a_{11} & a_{12} & \cdots & a_{1n} \\
a_{21} & a_{22} & \cdots & a_{2n} \\
\vdots & \vdots & \ddots & \vdots \\
a_{n1} & a_{n2} & \cdots & a_{nn}
\end{pmatrix} \cdot \begin{pmatrix}
b_{11} & b_{12} & \cdots & b_{1n} \\
b_{21} & b_{22} & \cdots & b_{2n} \\
\vdots & \vdots & \ddots & \vdots \\
b_{n1} & b_{n2} & \cdots & b_{nn}
\end{pmatrix}
\]

\[c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}\]

Assume for simplicity that \(n = 2^k\).
import sys, random
from time import *

n = 4096
A = [[1.0*random.randrange(0,2**64-1)
     for row in xrange(n)]
     for col in xrange(n)]
B = [[1.0*random.randrange(0,2**64-1)
     for row in xrange(n)]
     for col in xrange(n)]
C = [[0
     for row in xrange(n)]
     for col in xrange(n)]

start = time()
for i in xrange(n):
    for j in xrange(n):
        for k in xrange(n):
            C[i][j] += A[i][k] * B[k][j]
end = time()

print '%0.6f' % (end-start)

Running time
26,825 seconds on a
12-core Intel Xeon
X5650 system

Is that slow or fast?
Performance of Python code

**Back-of-the-envelope calculation**

It took 26,825 seconds = almost 7.5 hours to multiply two $4096 \times 4096$ matrices

$2 \times 4096^3 = 137,438,953,472$ floating-point operations

Operations per second = $137,438,953,472 / 26,825 = 5.12 \times 10^6$

**Intel Xeon X5650**

12 cores clocked at 2.67GHz with 4 floating-point operations per cycle

$= 128.4 \times 10^9$ FLOPS

The code achieves $5.12 \times 10^6 / 128.4 \times 10^9 \approx 1/25,000$ peak FP performance

**How can we improve performance?**
Python is slow. Code it in Java!

```java
import java.util.Random;

public class mm {
    static int n = 1024;
    static double[][] A = new double[n][n];
    static double[][] B = new double[n][n];
    static double[][] C = new double[n][n];

    public static void main(String[] args) {
        Random r = new Random();

        for (int i=0; i<n; i++) {
            for (int j=0; j<n; j++) {
                A[i][j] = (double)r.nextLong();
                B[i][j] = (double)r.nextLong();
                C[i][j] = 0;
            }
        }

        long start = System.nanoTime();
        for (int i=0; i<n; i++) {
            for (int j=0; j<n; j++) {
                for (int k=0; k<n; k++) {
                    C[i][j] += A[i][k]*B[k][j];
                }
            }
        }
        long stop = System.nanoTime();
        System.out.println((double)(stop-start) / 1000*1000*1000);
    }
}
```

Running time 2, 257 seconds
Performance of Java code

<table>
<thead>
<tr>
<th>Version</th>
<th>Time (s)</th>
<th>Speedup</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>26,825</td>
<td>1.00</td>
<td>103, 499, 540, 900, 324</td>
</tr>
<tr>
<td>Java</td>
<td>2,257</td>
<td>11.88</td>
<td>761, 916, 516, 742</td>
</tr>
</tbody>
</table>

Why is Java faster?

Python is *interpreted* by a virtual machine.
Java is *compiled* to byte-code,
    which is then interpreted by a virtual machine at the start of execution
    but will be eventually compiled by a Just-In-Time (JIT) compiler

How can we improve performance?

Use a language that compiles directly to machine code.
Java also provides sophisticated memory management (i.e., a garbage collector), which is expensive and can result in a loss of data locality.
```c
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>

typedef unsigned long long uint64_t;

static const int n = 4096;
double A[n][n], B[n][n], C[n][n];

// Simple pseudorandom-number generator
static uint64_t X = 0xce3f12500545b241ul;
static const uint64_t a = 0xd31cd625d63ba689ul;
static const uint64_t b = 0x0a58ec3022b3b941ul;

inline uint64_t prandnum() {
    X = a*X + b;
    return X;
}

float tdiff (struct timeval *start, struct timeval *end) {
    return (end->tv_sec-start->tv_sec) + 1e-6*(end->tv_usec-start->tv_usec);
}

int main() {
    for (int i=0; i<n; ++i) {
        for (int j=0; j<n; ++j) {
            A[i][j] = prandnum();
            B[i][j] = prandnum();
            C[i][j] = 0;
        }
    }

    struct timeval start, end;
    gettimeofday(&start, NULL);
    for (int i=0; i<n; ++i) {
        for (int j=0; j<n; ++j) {
            for (int k=0; k<n; ++k) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    gettimeofday(&end, NULL);
    printf("%0.6f\n", tdiff(&start, &end));
    return 0;
}

// Running time
54 seconds
```
Performance of C code

<table>
<thead>
<tr>
<th>Version</th>
<th>Time (s)</th>
<th>Speedup</th>
<th>Relative</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>26,825</td>
<td>1.00</td>
<td>1.00</td>
<td>103, 499, 540, 900, 324</td>
</tr>
<tr>
<td>Java</td>
<td>2,257</td>
<td>11.88</td>
<td>11.88</td>
<td>761, 916, 516, 742</td>
</tr>
<tr>
<td>C</td>
<td>54</td>
<td>515.87</td>
<td>43.42</td>
<td>164, 483, 442, 767</td>
</tr>
</tbody>
</table>

Compilation is *much* faster than interpretation!

Moveover, Java and Python do array-bounds checking, whereas C does not. But why is there a 43× relative speedup when the number of instructions executed decreased by a factor of less than 5?

Memory references are much more expensive than processor operations.

The C compiler is much better than Java at keeping data values in processor registers rather than in the memory system.

<table>
<thead>
<tr>
<th>Version</th>
<th>Memory References</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>235, 360, 602, 232</td>
</tr>
<tr>
<td>C</td>
<td>3, 650, 713, 016</td>
</tr>
</tbody>
</table>
Impact of the Programming Language

Programs can get Optimized

- Algebraic simplification
- Dead code elimination
- Loop invariant code motion
- Register allocation
- Instruction selection and scheduling

Programs written in some languages are easier to analyze than others

- Strongly typed 😊
- Dynamic class modification 😞

When optimizations are done

- At compile-time
  - Can do a lot, time is no barrier
  - But less information about the program
- At runtime
  - Need to amortize the cost
  - JIT compilation of commonly used

© Robert Miller 2009
Parallelism

Let's use all the processing cores!
Impact of Parallelization

The Multicore Menace

- Old days: every 18 months performance doubles
- Now: We only get more cores
- Now: Performance IS parallelism
Impact of Parallelization

The Multicore Menace

- Old days: every 18 months performance doubles
- Now: We only get more cores
- Now: Performance IS parallelism

Amdhal’s Law

- Any computation can be analyzed in terms of a portion that must be executed sequentially, $T_s$, and a portion that can be executed in parallel, $T_p$. Then for $n$ processors:
  - $T(n) = T_s + T_p/n$
  - $T(\infty) = T_s$, thus maximum speedup $(T_s + T_p) / T_s$

Load Balancing

The work is distributed among processors so that all processors are kept busy all of the time.

Granularity

- The size of the parallel regions between synchronizations or the ratio of computation (useful work) to communication (overhead)
Parallelizing loops using Cilk

```c
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>

typedef unsigned long long uint64_t;

static const int n = 4096;
double A[n][n], B[n][n], C[n][n];

// Simple pseudorandom-number generator
static uint64_t X = 0xce3f12500545b241ul;
static const uint64_t a = 0xd31cd625d63ba689ul;
static const uint64_t b = 0x0a58ec3022b3b941ul;

inline uint64_t prandnum() {
    X = a*X + b;
    return X;
}

float tdiff (struct timeval *start, struct timeval *end) {
    return (end->tv_sec-start->tv_sec)
        +1e-6*(end->tv_usec-start->tv_usec);
}

int main() {
    for (int i=0; i<n; ++i) {
        for (int j=0; j<n; ++j) {
            A[i][j] = prandnum();
            B[i][j] = prandnum();
            C[i][j] = 0;
        }
    }
    struct timeval start, end;
    gettimeofday(&start, NULL);
    cilk_for (int i=0; i<n; ++i) {
        cilk_for (int j=0; j<n; ++j) {
            for (int k=0; k<n; ++k) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    gettimeofday(&end, NULL);
    printf("%0.6f\n", tdiff(&start, &end));
    return 0;
}
```

Running time on 12 cores
120 seconds
Performance of Parallel Loops

<table>
<thead>
<tr>
<th>Version</th>
<th>Time (s)</th>
<th>Speedup</th>
<th>Relative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>26,825</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Java</td>
<td>2,257</td>
<td>11.88</td>
<td>11.88</td>
</tr>
<tr>
<td>C</td>
<td>54</td>
<td>515.87</td>
<td>43.42</td>
</tr>
<tr>
<td>Parallel loops</td>
<td>120</td>
<td>223.54</td>
<td>0.43</td>
</tr>
</tbody>
</table>

We lost ground!

1. Parallel loops induce overhead.
2. The compiler was vectorizing the inner loop, and the introduction of `cilk_for` disabled the vectorization.

Vector instructions

Instructions that perform a uniform operation over several data elements simultaneously.

The ability of the compiler to vectorize a loop is limited, and specific heuristic conditions must be met.
Architectural Impact

Modern processors are extremely complex

- Has many components, each can be problematic!
  - Instructions
  - Memory System
  - Processor Bus and IO Subsystem
  - Disk System
  - GPU and Graphics System
  - Network

- Made to work well for the common cases

Issues with Instructions

- Pipeline stalls and pipeline flush
  - Ex: cache miss on a memory fetch
- Interlocking (interference between instructions)
- Instruction level parallelism
- Use of short vector instructions (SSE)

© Robert Miller 2009
Blocking the output matrix

**Compute each block submatrix of C separately**

- Two parallel outer loops compute the blocks of C concurrently.
- Three inner loops perform the multiplication on strips of A and B, which the compiler can vectorize.

$$\begin{align*}
C[i][j] &+ A[i][k] \times B[k][j];
\end{align*}$$
const int BLOCKSIZE = 64;
// Macro for indexing location (x,y) in matrix A with d columns
#define IND(A, x, y, d)  A[(x)*(d)+(y)]

// Inner loops
inline void
mm_base(double *C, const double *A, const double *B, int size, int n)
{
    // reorder inner loops
    for (int i = 0; i < size; ++i) {
        for (int k = 0; k < n; ++k) {
            for (int j = 0; j < size; ++j) {
                IND(C,i,j,n) += IND(A,i,k,n) * IND(B,k,j,n);
            }
        }
    }
}

// Outer loops
inline void
mm_loop(double *C, const double *A, const double *B, int n)
{
    const int blocksize = n < BLOCKSIZE ? n : BLOCKSIZE;
    
cilk_for (int bi = 0; bi < n; bi += blocksize) {
        cilk_for (int bj = 0; bj < n; bj += blocksize) {
            mm_base(C+(bi*n)+bj, A+(bi*n), B+bj, blocksize, n);
        }
    }
}
## Performance of blocking the output

<table>
<thead>
<tr>
<th>Version</th>
<th>Time (s)</th>
<th>Speedup</th>
<th>Relative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>26,825.3</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Java</td>
<td>2,257.0</td>
<td>11.88</td>
<td>11.88</td>
</tr>
<tr>
<td>C</td>
<td>54.0</td>
<td>515.87</td>
<td>43.42</td>
</tr>
<tr>
<td>Parallel loops</td>
<td>120.2</td>
<td>223.54</td>
<td>0.43</td>
</tr>
<tr>
<td>Blocking output</td>
<td>15.6</td>
<td>1,719.55</td>
<td>7.69</td>
</tr>
</tbody>
</table>

The compiler can now vectorize the inner loop. But we’re still more than $15 \times$ away from peak floating-point performance. Where is the time going?

<table>
<thead>
<tr>
<th>Version</th>
<th>Memory References</th>
<th>L3-Cache Misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocking output</td>
<td>7,031,795,350</td>
<td>3,633,630,174</td>
</tr>
</tbody>
</table>

Over half the memory references are missing in the cache!
Impact of the Memory System

Memory Hierarchy

➢ The dilemma: How do you have a lot of memory and access them very fast
  • Small amount of memory $\rightarrow$ fast access
  • Large amount of memory $\rightarrow$ slow access

➢ Cache Hierarchy
  • Store most probable accesses in small amount of memory with fast access
  • Hardware heuristics determine what will be in each cache and when

➢ The temperamental cache
  • If your access pattern matches heuristics of the hardware $\rightarrow$ blazingly fast
  • Otherwise $\rightarrow$ dog slow

© Robert Miller 2009
Tiling

Block all the matrices

• Three outer loops, the outermost two of which are parallel, compute a matrix product where each element is a square submatrix tile.
• Three inner loops perform the multiplication on submatrix tiles, which the compiler can vectorize.
• Choose the tile size to minimize running time (ensure the active tiles fit into cache).

\[ C = A \times B \]
Data Reuse

Data reuse

- Change of computation order can reduce the # of loads to cache
- Calculating a row (1024 values of A)
  - A: $1024 \times 1 = 1024$ + B: $384 \times 1 = 394$ + C: $1024 \times 384 = 393,216 = 394,524$
- Blocked Matrix Multiply ($32^2 = 1024$ values of A)
  - A: $32 \times 32 = 1024$ + B: $384 \times 32 = 12,284$ + C: $32 \times 384 = 12,284 = 25,600$
Tiling code

const int BLOCKSIZE = 64;
// Macro for indexing location (x,y) in matrix A with d columns
#define IND(A, x, y, d) A[(x)*(d)+(y)]

// Inner loops
inline void mm_base(double *C, const double *A, const double *B, int size, int n)
{
    for (int i = 0; i < size; ++i) {
        for (int k = 0; k < size; ++k) {
            for (int j = 0; j < size; ++j) {
                IND(C, i, j, n) += IND(A, i, k, n) * IND(B, k, j, n);[@\lilabel{base-multiply}@]
            }
        }
    }
}

// Outer loops
inline void mm_loop(double *C, const double *A, const double *B, int n)
{
    const int blocksize = n < BLOCKSIZE ? n : BLOCKSIZE;

    cilk_for (int bi = 0; bi < n; bi += blocksize) {
        cilk_for (int bj = 0; bj < n; bj += blocksize) {
            for (int bk = 0; bk < n; bk += blocksize) {
                mm_base(C + (bi*n)+bj, A + (bi*n)+bk, B + (bk*n)+bj, blocksize, n);
            }
        }
    }

Running time
4.7 seconds
### Tiling

<table>
<thead>
<tr>
<th>Version</th>
<th>Time (s)</th>
<th>Speedup</th>
<th>Relative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>26, 825.3</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Java</td>
<td>2, 257.0</td>
<td>11.88</td>
<td>11.88</td>
</tr>
<tr>
<td>C</td>
<td>54.0</td>
<td>515.87</td>
<td>43.42</td>
</tr>
<tr>
<td>Parallel loops</td>
<td>120.2</td>
<td>223.54</td>
<td>0.43</td>
</tr>
<tr>
<td>Blocking output</td>
<td>15.6</td>
<td>1, 719.55</td>
<td>7.69</td>
</tr>
<tr>
<td>Tiling</td>
<td>4.7</td>
<td>5, 707.45</td>
<td>3.32</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Version</th>
<th>Memory References</th>
<th>L3-Cache Misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocking output</td>
<td>7, 031, 795, 350</td>
<td>3, 633, 630, 174</td>
</tr>
<tr>
<td>Tiling</td>
<td>10, 675, 744, 835</td>
<td>227, 526, 545</td>
</tr>
</tbody>
</table>

### Problem

The tuning parameter for tile size renders the code nonportable.
“Cache-oblivious” tiling

Divide and conquer

8 recursive multiplications of $n/2 \times n/2$ submatrices
1 addition of $n \times n$ matrices

\[
\begin{pmatrix}
  C_{11} & C_{12} \\
  C_{21} & C_{22}
\end{pmatrix}
= \begin{pmatrix}
  A_{11} & A_{12} \\
  A_{21} & A_{22}
\end{pmatrix} \cdot \begin{pmatrix}
  B_{11} & B_{12} \\
  B_{21} & B_{22}
\end{pmatrix}
\]

\[
= \begin{pmatrix}
  A_{11}B_{11} & A_{11}B_{12} \\
  A_{21}B_{11} & A_{21}B_{12}
\end{pmatrix} + \begin{pmatrix}
  A_{12}B_{21} & A_{12}B_{22} \\
  A_{22}B_{21} & A_{22}B_{22}
\end{pmatrix}
\]
D&C Cilk code

```c
void mm_dac(double *C, const double *A, const double *B, int size, int n)
{
    // Base case
    if (size == 1) {
        IND(C,0,0,n) += IND(A,0,0,n)*IND(B,0,0,n);
        return;
    }
    // Initialize submatrices
    cilk_spawn mm_dac(C11, A11, B11, size/2, n);
    cilk_spawn mm_dac(C12, A11, B12, size/2, n);
    cilk_spawn mm_dac(C21, A21, B11, size/2, n);
    mm_dac(C22, A21, B12, size/2, n);
    cilk_sync;
    cilk_spawn mm_dac(C11, A12, B21, size/2, n);
    cilk_spawn mm_dac(C12, A12, B22, size/2, n);
    cilk_spawn mm_dac(C21, A22, B21, size/2, n);
    mm_dac(C22, A22, B22, size/2, n);
    cilk_sync;
}
```

The named `child` function may execute in parallel with the `parent` caller.

Control cannot pass this `sync` point until all spawned children have returned.

Running time
216 seconds
## D&C performance

<table>
<thead>
<tr>
<th>Version</th>
<th>Time (s)</th>
<th>Speedup</th>
<th>Relative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>26,825.3</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Java</td>
<td>2,257.0</td>
<td>11.88</td>
<td>11.88</td>
</tr>
<tr>
<td>C</td>
<td>54.0</td>
<td>515.87</td>
<td>43.42</td>
</tr>
<tr>
<td>Parallel loops</td>
<td>120.2</td>
<td>223.54</td>
<td>0.43</td>
</tr>
<tr>
<td>Blocking output</td>
<td>15.6</td>
<td>1,719.55</td>
<td>7.69</td>
</tr>
<tr>
<td>Tiling</td>
<td>4.7</td>
<td>5,707.45</td>
<td>3.32</td>
</tr>
<tr>
<td>D&amp;C</td>
<td>215.9</td>
<td>124.25</td>
<td>0.02</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Version</th>
<th>Instructions</th>
<th>Memory References</th>
<th>L3-Cache Misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocking output</td>
<td>188,324,001,381</td>
<td>7,031,795,350</td>
<td>3,633,630,174</td>
</tr>
<tr>
<td>Tiling</td>
<td>220,512,757,143</td>
<td>10,675,744,835</td>
<td>227,526,545</td>
</tr>
<tr>
<td>D&amp;C</td>
<td>14,286,772,228,562</td>
<td>5,010,121,137</td>
<td>155,581,724</td>
</tr>
</tbody>
</table>

Need to coarsen the base case!
inline void
transpose(int n,
    const double *src, int srcSep,
    double *dst, int dstSep)
{
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            IND(dst, j, i, dstSep) =
                IND(src, i, j, srcSep);
        }
    }
}

void mm_base(double *C, const double *A, const double *B, int size, int n)
{
    // Base case
    if (size <= THRESHOLD) {
        mm_base(C, A, B, size, n);
        return;
    }
    // Initialize submatrices
    cilk_spawn mm_dac(C11, A11, B11, size/2, n);
    cilk_spawn mm_dac(C12, A11, B12, size/2, n);
    cilk_spawn mm_dac(C21, A21, B11, size/2, n);
    cilk_spawn mm_dac(C22, A21, B12, size/2, n);
    cilk_sync;
    cilk_spawn mm_dac(C11, A12, B21, size/2, n);
    cilk_spawn mm_dac(C12, A12, B22, size/2, n);
    cilk_spawn mm_dac(C21, A22, B21, size/2, n);
    cilk_spawn mm_dac(C22, A22, B22, size/2, n);
    cilk_sync;
}

Running time
3.1 seconds
Row-major matrix layout

Contiguous accesses are better
Data is fetched as 64-byte cache blocks.
Contiguous data (unit stride) ⇒ single fetch provides 8 doubles.
## D&C + coarsening + transpose perf.

<table>
<thead>
<tr>
<th>Version</th>
<th>Time (s)</th>
<th>Speedup</th>
<th>Relative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>26,825.3</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Java</td>
<td>2,257.0</td>
<td>11.88</td>
<td>11.88</td>
</tr>
<tr>
<td>C</td>
<td>54.0</td>
<td>515.87</td>
<td>43.42</td>
</tr>
<tr>
<td>Parallel loops</td>
<td>120.2</td>
<td>223.54</td>
<td>0.43</td>
</tr>
<tr>
<td>Blocking output</td>
<td>15.6</td>
<td>1,719.55</td>
<td>7.69</td>
</tr>
<tr>
<td>Tiling</td>
<td>4.7</td>
<td>5,707.45</td>
<td>3.32</td>
</tr>
<tr>
<td>D&amp;C</td>
<td>215.9</td>
<td>124.25</td>
<td>0.02</td>
</tr>
<tr>
<td>D&amp;C+coarsening+transpose</td>
<td>3.1</td>
<td>8,667.21</td>
<td>69.65</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Version</th>
<th>Instructions</th>
<th>Memory References</th>
<th>L3-Cache Misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocking output</td>
<td>188,324,001,381</td>
<td>7,031,795,350</td>
<td>3,633,630,174</td>
</tr>
<tr>
<td>Tiling</td>
<td>220,512,757,143</td>
<td>10,675,744,835</td>
<td>227,526,545</td>
</tr>
<tr>
<td>D&amp;C</td>
<td>14,286,772,228,562</td>
<td>5,010,121,137</td>
<td>155,581,724</td>
</tr>
<tr>
<td>D&amp;C+</td>
<td>175,818,127,063</td>
<td>665,888,803</td>
<td>107,905,973</td>
</tr>
</tbody>
</table>
## Final results

<table>
<thead>
<tr>
<th>Version</th>
<th>Time (s)</th>
<th>Speedup</th>
<th>Relative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>26,825.27</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Java</td>
<td>2,256.96</td>
<td>11.88</td>
<td>11.88</td>
</tr>
<tr>
<td>C</td>
<td>53.69</td>
<td>515.87</td>
<td>43.42</td>
</tr>
<tr>
<td>Parallel loops</td>
<td>120.15</td>
<td>223.54</td>
<td>0.43</td>
</tr>
<tr>
<td>Blocking output</td>
<td>15.63</td>
<td>1,719.55</td>
<td>7.69</td>
</tr>
<tr>
<td>Tiling</td>
<td>4.74</td>
<td>5,707.45</td>
<td>3.32</td>
</tr>
<tr>
<td>D&amp;C</td>
<td>215.93</td>
<td>124.25</td>
<td>0.02</td>
</tr>
<tr>
<td>D&amp;C+coarsening+transpose</td>
<td>3.10</td>
<td>8,667.21</td>
<td>69.65</td>
</tr>
<tr>
<td>Parallel Strassen+</td>
<td>1.97</td>
<td>13,616.75</td>
<td>1.97</td>
</tr>
<tr>
<td>Intel MKL (the pros)</td>
<td>1.07</td>
<td>25,070.09</td>
<td>1.84</td>
</tr>
</tbody>
</table>
Summary

There can be a lot of room for performance improvements!

- Matrix multiplication is exceptional. Other programs may not yield gains this large
- That said, matrix multiplication from Python to MKL yields a 25,070× improvement!
- In comparison miles per gallon improvement

Many factors impact performance

- Algorithm selection
- Language construct selection
- Memory management
- Architectural details
- Programming language used
- Parallelism achieved

Performance engineering requires

- Knowing that there is a performance problem
- Identifying the performance bottlenecks
- Establishing the leading cause of the problem
- Eliminating the performance problem
6.172 Performance Engineering of Software Systems
taught in the fall
prerequisites 6.004, 6.005, and 6.006