Mathematical Software Programming (02635)

Lecture 7 — October 23, 2025

Instructor: Martin S. Andersen

Fall 2025

This week

Topics

- timing your programs
- complexity
- basic computer architecture and efficiency
- compiler optimization

Learning objectives

- analyze the runtime behavior and the time and space complexity of simple programs
- get a basic understanding of computer architecture and performance

Timing your programs

Basic timing routines available in header file `time.h`

Wall time

Prototype: `time_t time(time_t *tloc)`

- operating system time (not guaranteed to be *monotonic*)
- resolution: 1 second

CPU time

Prototype: `clock_t clock(void)`

- returns (approximation to) processor time used by process
- resolution is system-dependent
- `clock()` may *wrap around*
- divide by macro `CLOCKS_PER_SEC` to convert to seconds

Example: measuring CPU time

```c
#include <time.h>

int main(void) {
    double cpu_time;
    clock_t T1, T2;
    T1 = clock();
    /* ... code you want to time ... */
    T2 = clock();
    cpu_time = ((double)(T2-T1)) / CLOCKS_PER_SEC;
    printf("CPU time: %.2e\n",cpu_time);
    return 0;
}
```

What can you do if time resolution is poor?

Some platform-specific timing routines

GNU/Linux (macOS 10.6+)

- `clock_gettime()` (`#include <time.h>`)

POSIX (Portable Operating System Interface)

- `getrusage()` (`#include <sys/resource.h>`)

macOS

- `mach_absolute_time()`, `mach_timebase_info()` (`#include <mach/mach_time.h>`)

Windows

- `GetTickCount64()` (`#include <Windows.h>`)

Profiling

Purpose: get the bigger picture

- Dynamic program analysis
- Find the hotspots/bottlenecks in your code
- Analyze space (memory) and time complexity of programs

Tools

Modern (statistical) tools (no code change or re-compilation):

- Linux: Performance Tools (`perf`)
- macOS: Xcode / Instruments (`iprofiler`)
- Windows: Visual Studio Profiling Tools
- Google performance tools (gperftools)

Classical tools (instrumenting your code, needs re-compilation):

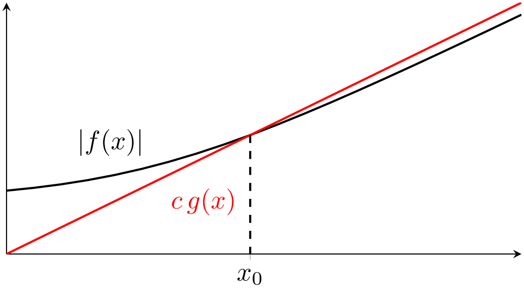
- Linux/Unix: `gprof` (see 'man gprof' for more information)

Big O notation

$f(x)$ is "big o" of $g(x)$ if for some $c > 0$ and x_0

$$|f(x)| \leq c\,g(x), \quad \forall x \geq x_0,$$

writen as $f(x) = O(g(x))$ as $x \to \infty$.



$g(x)$ provides an **asymptotic upper bound** on $f(x)$

Big O notation (cont.)

Consider the function $f(x) = 4x^3 - 2x^2 + 5x$ which satisfies

$$f(x) = 4x^3 - 2x^2 + 5x \leq |4x^3| + |2x^2| + |5x|.$$

Moreover, if $x \geq 1$,

$$f(x) \leq |4x^3| + |2x^2| + |5x| \leq |4x^3| + |2x^3| + |5x^3| \leq 11x^3 \implies f(x) = O(x^3).$$

- the statements $f(x) = O(x^4)$ and $f(x) = O(2^x)$ are also true
- we are often interested in the "best" asymptotic upper bound
- "big omega" yields **asymptotic lower bound** on f

$$f(x) = \Omega(g(x)) \iff g(x) = O(f(x))$$

- "big theta" yields **asymptotic tight bound** on f

$$f(x) = \Theta(g(x)) \iff f(x) = O(g(x)) \wedge f(x) = \Omega(g(x))$$

Time and space complexity

Time complexity

- accessing ith element of an array requires $O(1)$ time
- adding two vectors of length n requires $O(n)$ time
- finding the maximum element of a vector requires $O(n)$ time
- enumerating all possible permutations of n objects requires $O(n!)$ time

Space (memory) complexity

- a vector of length n requires $O(n)$ memory
- a matrix of size $m \times n$ requires $O(mn)$ memory

Some complexity classes (problem size n)

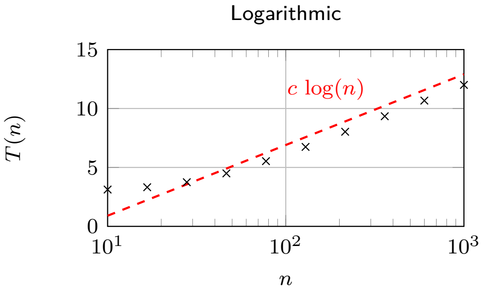
Complexity	Big-O notation
Constant	$O(1)$
Logarithmic	$O(\log n)$
Poly-logarithmic	$O((\log n)^k)$
Linear	$O(n)$
Quasi-linear/log-linear	$O(n(\log n)^k)$
Polynomial	$O(n^k)$
Exponential	$O(2^{\mathrm{poly}(n)})$
Factorial	$O(n!)$
Double exponential	$O(2^{2^{\mathrm{poly}(n)}})$

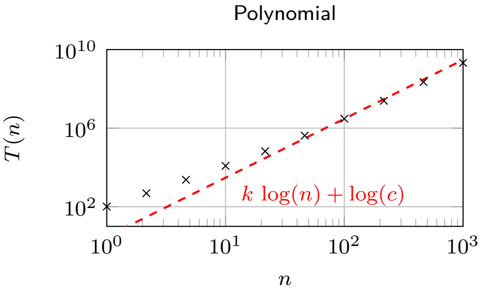
Comparison of time complexity

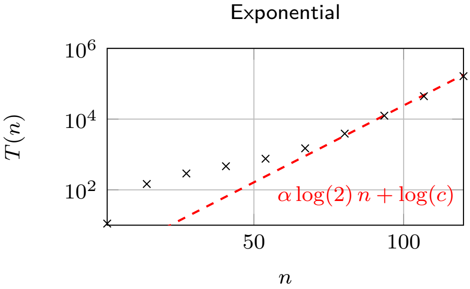
$T(n)$	$n = 10$	$n = 20$	$n = 30$	$n = 40$
n	0.00001 s	0.00002 s	0.00003 s	0.00004 s
n^2	0.0001 s	0.0004 s	0.0009 s	0.0016 s
n^3	0.001 s	0.008 s	0.027 s	0.064 s
n^5	0.1 s	3.2 s	24.3 s	1.7 min
2^n	0.001 s	1 s	17.9 min	12.7 days
3^n	0.059 s	58 min	6.5 years	3855 centuries

Hypothesis	Plot	Upper bound	Slope
$O(\log(n))$	Log n-axis	$T(n) \leq c \log(n)$	c
$O(n^k)$	Log-log plot	$\log(T(n)) \leq k \log(n) + \log(c)$	k
$O(2^{\alpha n})$	Log T-axis	$\log(T(n)) \leq \alpha \log(2)\, n + \log(c)$	$\alpha \log(2)$

Running time $T(n)$ versus input size n







Computer architecture

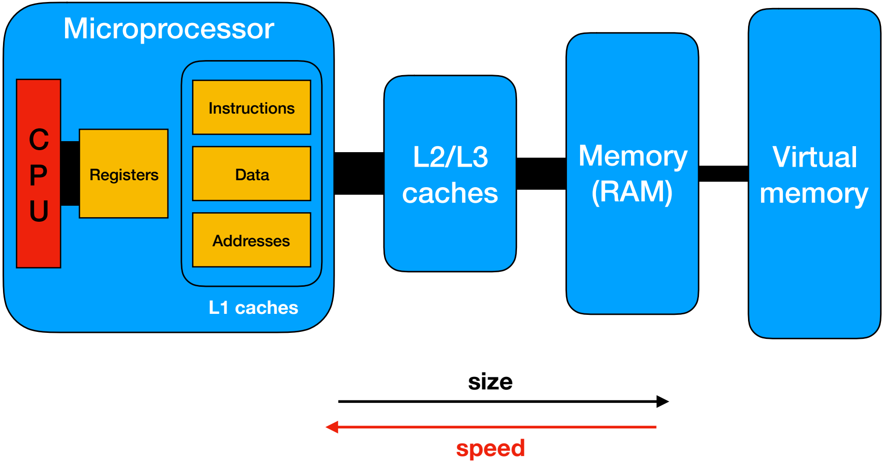
Central processing unit

- CPU cores, math coprocessor
- registers and cache memory
- instruction pipelining

Memory hierarchy

- L0: CPU registers
- L1: level 1 cache (SRAM)
- L2: level 2 cache (SRAM)
- L3: level 3 cache (SRAM)
- L4: random access memory (DRAM)
- L5: local secondary storage (local disks)
- L6: remote secondary storage (distributed file systems, servers)

Computer architecture (cont.)



Approximate latency comparison numbers (~2012)

Description	Time (ns)
CPU cycle	0.3
L1 cache reference	1
Branch mispredict	5
L2 cache reference	7
L3 cache reference	13
Main memory reference	100
Read 4K randomly from SSD	150,000
Read 1 MB sequentially from memory	250,000
Read 1 MB sequentially from SSD	1,000,000
Disk seek	10,000,000
Read 1 MB sequentially from disk	20,000,000
Send packet CA–Netherlands–CA	150,000,000

What are my system specs?

Windows Command Prompt

```
C:\>wmic cpu get L2CacheSize, L3CacheSize
```

Alternatively, download and use CPU-Z system profiler.

macOS

```
$ system_profiler SPHardwareDataType
$ sysctl -a | grep "hw.*cache"
```

Linux/Unix

```
$ lscpu | grep cache
$ cat /proc/cpuinfo
$ cat /proc/meminfo
```

Locality

Temporal locality

- reuse of instructions/data within a short window of time
- instructions may be reused in a *tight* loop
- repeatedly using/referencing the same variables

Spatial locality

- use of data elements within relatively close storage locations
- small stride access patterns (e.g. stride-1 access)
- execute instructions in sequence

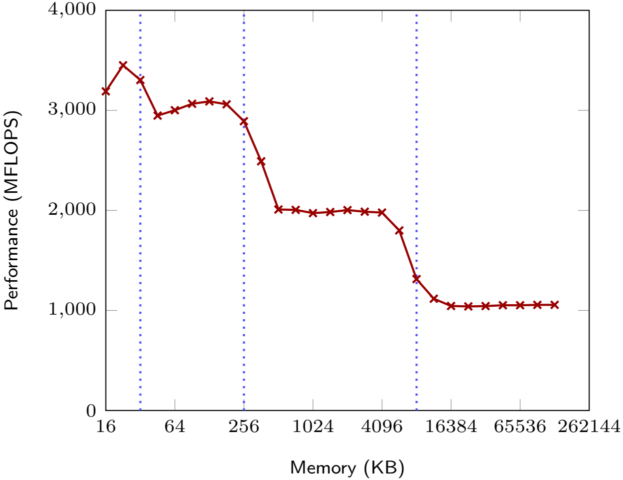
Example

```
sum = 0;
for (i=0; i<n; i++)
   sum += data[i];
```

Data size and cache size effects on performance

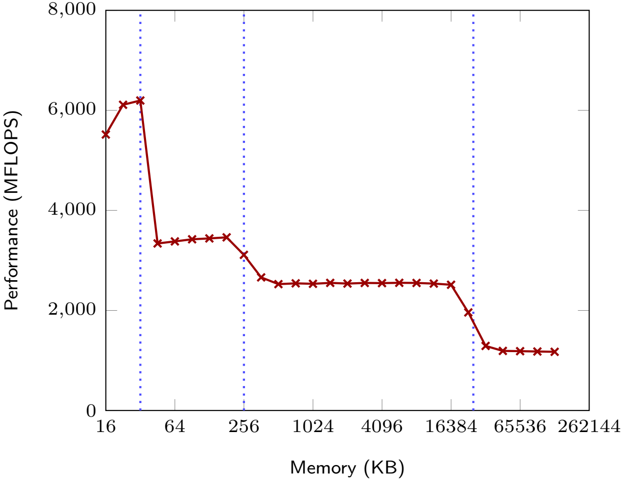
```
extern double arr[];  // global array defined in another source file

int datasize1(int elem) {

  for (int i=0; i<elem; i++) arr[i] *= 3;    // 1 FLOP

  /* return the number of memory accesses */
  return(elem);
}
```

Data size and cache size (cont.)



Specs: `Intel Xeon X5550 @ 2.67GHz, 32 kB L1, 256 kB L2, 8 MB L3 cache`

Data size and cache size (cont.)



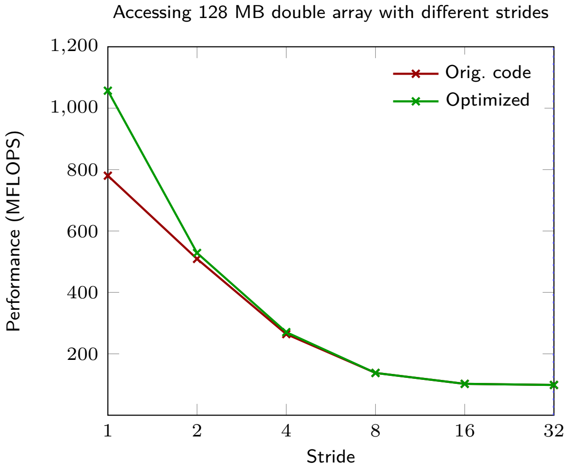
Specs: Intel Xeon E5-2660 v3 @ 2.60GHz, 32 kB L1, 256 kB L2, 25 MB L3 cache

Cache speed and spatial locality

```c
extern int N;                          // length or array
extern double arr[];

int stridetest(int incr) {

    for (int i = 0; i < N; i += incr) arr[i] *= 3;  // 1 FLOP

    /* return the number of memory accesses */
    return(N/incr);
}
```

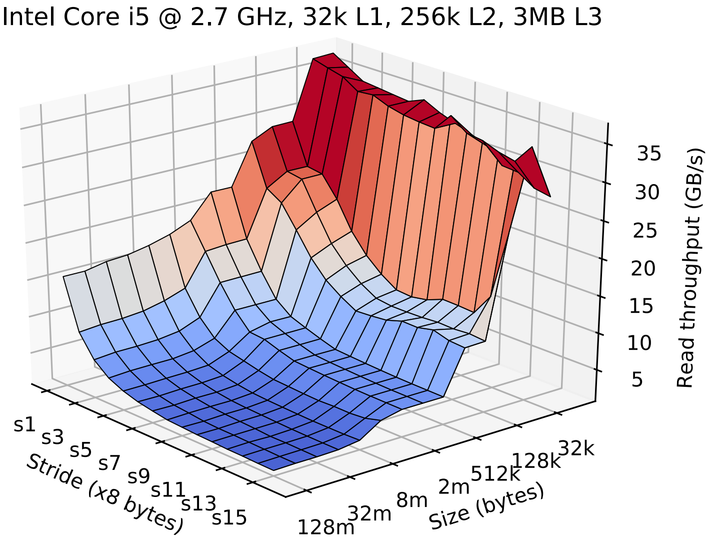
with `incr` equal to 1,2,4,8, . . .

Cache speed and spatial locality (cont.)



- Optimized code: uses a switch-statement on `incr` to allow extra compiler optimizations
- Random access: $\sim$ 108 MFLOPS

The memory mountain



Bryant & O'Hallaron (2003)

Instruction-level parallelism

```c
#define N 67108864          // 64*1024*1024
int arr[2] = {1,1};

for (i=0; i<N; i++) {  // Loop 1
   arr[0]++;           // S1
   arr[1]++;           // S2
}

for (i=0; i<N; i++) {  // Loop 2
   arr[0] += arr[1];   // S1
   arr[1] += arr[0];   // S2 (must wait for S1)
}
```

Optimizing compilers

Enable code optimization

```
$ gcc source.c -Wall -OX -o my_program
```

- `-O0` — no optimization (default, best option for debugging)
- `-O1` — most common forms of optimization
- `-O2` — additional code optimization
- `-O3` — most "expensive" code optimization (may increase size)
- `-Os` — code optimization that reduces size of executable
- `-ffast-math` — may violate IEEE standard

Makefile: add optimization flags to `CFLAGS` (e.g., `CFLAGS=-Wall -std=c99 -O2`)

Compile for a specific/generic CPU type

- `-march=native` (use `gcc -mcpu=help` to see supported CPU types)
- Intel CPUs: `-march=corei7`, `-march=x86-64-v4`, `-msse4.2`, `-mavx512f`, ...

Example: loop unrolling

```
for (i = 0; i < N; i++) y[i] = i;
```

can be re-written by compiler as

```
for(i = 0; i < (N - N%4); i+=4) {
    y[i]   = i;
    y[i+1] = i+1;
    y[i+2] = i+2;
    y[i+3] = i+3;
}
/* clean-up loop */
for(i = 4*(N/4); i < N; i++ ) y[i] = i;
```

- improves the "work to overhead" ratio
- GCC compiler flag `-funroll-loops`
- Clang: may be enabled with `-O1` (`#pragma clang loop unroll_count(N)`)

Today's exercises, part I: timing your code

Reproduce some of the plots from this lecture

- Write a simple timing framework to measure time and performance.
- Apply to the function `datasize1()` from the lecture.
- Get the performance characteristics of your computer.
- Compare with the specifications.

Today's excercises, part II: matrix-vector product

Looking ahead

- Next module: external linear algebra libraries (BLAS and LAPACK)
- BLAS matrix-vector multiplication function: **dgemv()** (mnemonic name for "**d**ouble precision **ge**neral **m**atrix **v**ector multiplication")
- After BLAS: **parallel** matrix-vector product with OpenMP (Bernd Dammann)
- Today's task: write your own versions of `dgemv()`, *i.e.*, `my_dgemv1()`, ...

Two-dimensional arrays and cache effects

- Matrices are "two-dimensional" but mapped to the one-dimensional memory address space
- This can lead to "good" or "bad" memory access with respect to performance
- Today's exercise: explore this for two different versions of the matrix-vector product

Today's exercises II: matrix times vector (cont.)

Computation

$$y \leftarrow \alpha A x + \beta y, \quad A \in \mathbb{R}^{m \times n},\ x \in \mathbb{R}^n,\ y \in \mathbb{R}^m$$

Method 1 (row-oriented)

$$y_i \leftarrow \alpha \sum_{j=1}^{n} A_{ij} x_j + \beta y_i, \quad i = 1, \ldots, m$$

Method 2 (column-oriented)

$$y \leftarrow \beta y + \sum_{j=1}^{n} \alpha x_j A_{:,j}$$

where $A_{:,j}$ is the jth column of A