Mathematical Software Programming (02635)

Module 6 — Fall 2016

Instructor: Bernd Dammann

Checklist — what you should know by now

- How to write a simple program in C (`int main(void) {}`)
- Basic data types (`int`, `long`, `float`, `double`, ...)
- Basic input/output (`printf`, `scanf`)
- Implicit/explicit typecasting
- How to compile and run a program from terminal / command prompt
- Control structures and loops (`if`, `else if`, `switch`, `for`, `do`, `while`)
- Pitfalls with integer and floating point arithmetic
- Arrays and multidimensional arrays
- Pointers: *"dereferencing"* and *"address of"* operators
- Use of functions to structure programs
- Dynamic memory allocation (`malloc`, `calloc`, `realloc`, `free`)
- Basic error checking (check return values, etc.)
- Data structures and types (`struct`, `typedef`, lists, stacks, queues)

This week

Topics

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

Learning objectives

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

Timing your programs

Basic timing routines available 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;

    return 0;
}
```

What can you do if time resolution is poor?

Some platform-specific timing routines

GNU/Linux

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

POSIX (Portable Operating System Interface)

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

Mac OS X

- `mach_absolute_time()` (`#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`)
- Mac OS X: Xcode / Instruments (`iprofiler`)
- Windows: Visual Studio Profiling Tools
- Google performance tools

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

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

Big-O notation and complexity

$$f(x) = O(g(x)) \text{ as } x \to \infty$$

- Big-O notation captures "growth rate"
- means that there exists a positive constant c and x_0 such that

$$|f(x)| \leq c|g(x)| \text{ for all } x \geq x_0$$

Space (memory) complexity

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

Time complexity

- accessing ith element of an array requires $O(1)$ time
- adding to vectors of length n requires $O(n)$ time
- finding the maximum element of a vector requires $O(n)$ time

Computer architecture

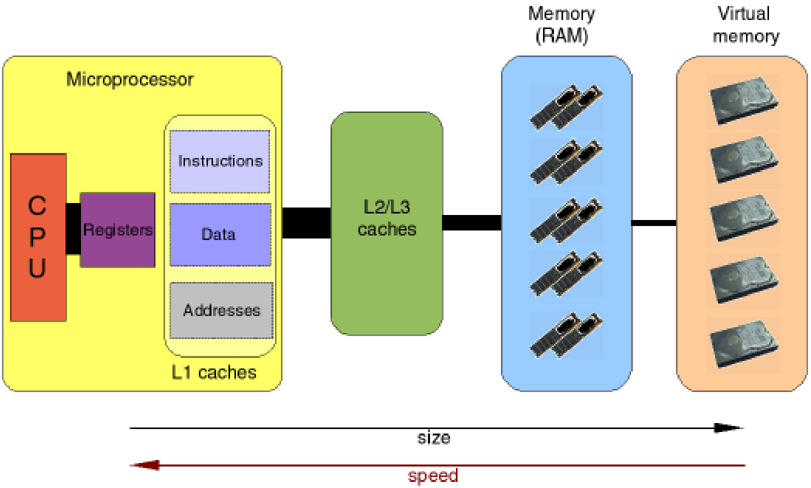
Central processing unit

- 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.)



The thickness of the bars connecting the levels illustrates the differences in bandwith.

What are my system specs?

Windows

```
C:\Users\user>msinfo32
```

Mac OS X

```
$ system_profiler SPHardwareDataType
```

Linux / Unix

```
$ lscpu
$ cat /proc/cpuinfo
$ cat /proc/meminfo
```

Locality

Temporal locality

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

Spatial locality

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

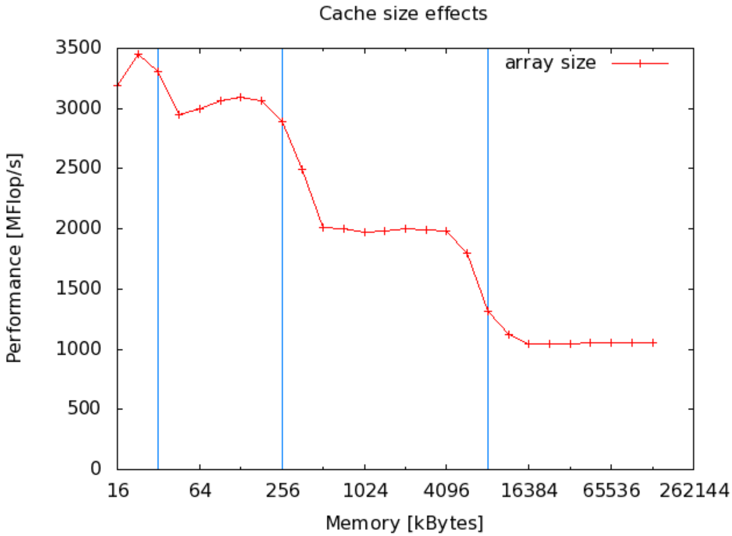
Example

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

Data size and cache size effects on performance

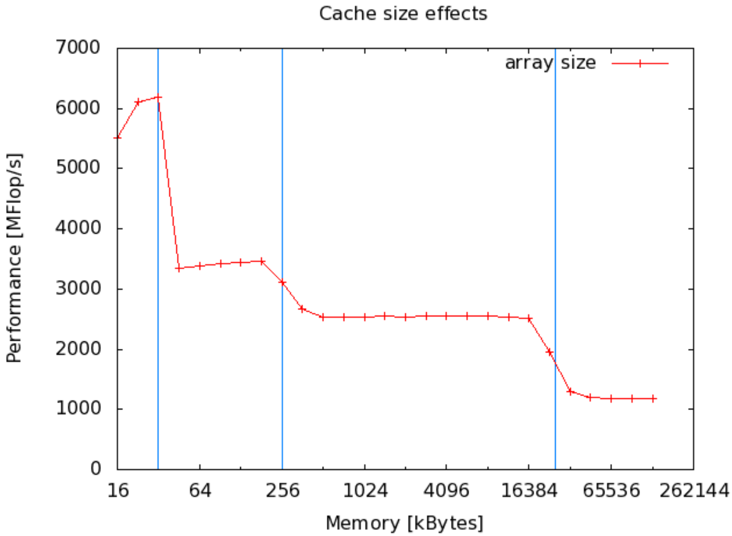
```
int
datasize1(int elem) {

    double arr[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, 32kB L1, 256 kB L2, 8 MB L3 cache size

Data size and cache size . . . (cont.)



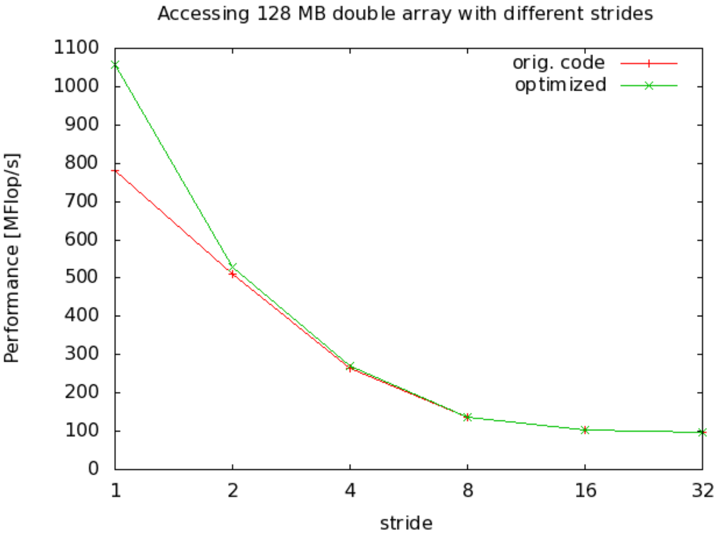
Specs: Intel Xeon E5-2660 v3 @ 2.60GHz, 32 kB L1, 256 kB L2, 25 MB L3 cache size

Cache speed and spatial locality

```c
#define N 16777216          // 64*1024*1024
double arr[N];              // 8*N = 128 MB

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.)



The optimized version uses a switch-statement on the `incr` variable, to allow the compiler to apply extra optimizations. Random access: $\sim$ 108 MFlop/s!

Instruction-level parallelism

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

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

for (i=0; i<N; i++) {  // Loop 2
   arr[0]++;
   arr[1]++;
}

for (i=0; i<N; i++) {  // Loop 3
   arr[0] += arr[1];
   arr[1] += arr[0];
}
```

Which loop(s) benefits from instruction-level parallelism?

Optimizing compilers

Enable code optimization

```
$ cc 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 reduce size of executable

Compile for a specific/generic CPU type

- `-mtune=native`, `-mtune=core2`
- `-mtune=generic`

Example: loop unrolling

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

can be rewritten 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`
- may be enabled with `-O1` (e.g., with Clang)

Today's exercises I: timing your code

Reproduce some of the lecture's plots

- 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
- . . . and compare with the specifications

Today's excercises II: matrix times vector

Time-travel: looking a few weeks ahead

- In one of the lectures after the fall break, you will learn how to use numerical libraries, especially LAPACK and BLAS
- One of BLAS functions that is often used is the function that can do matrix-vector multiplications: **dgemv()**
- The name comes from the BLAS naming convention (more in a later lecture), and means ''**d**ouble precision **ge**neral **m**atrix **v**ector multiplication''.
- Today's task: write your own version(s) of `dgemv()`, i.e. `my_dgemv1()`, `...`

two dimensional arrays and cache effects

- unlike vectors, matrices are two dimensional objects, that are mapped to the one-dimensional memory address space
- this can lead to good or bad access, when it comes to performance
- in today's exercise, you should explore this for two different versions of the matrix times vector example

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