ECE 695 Numerical Simulations Lecture 3: Practical Assessment of Code Performance

Prof. Peter Bermel

January 13, 2017

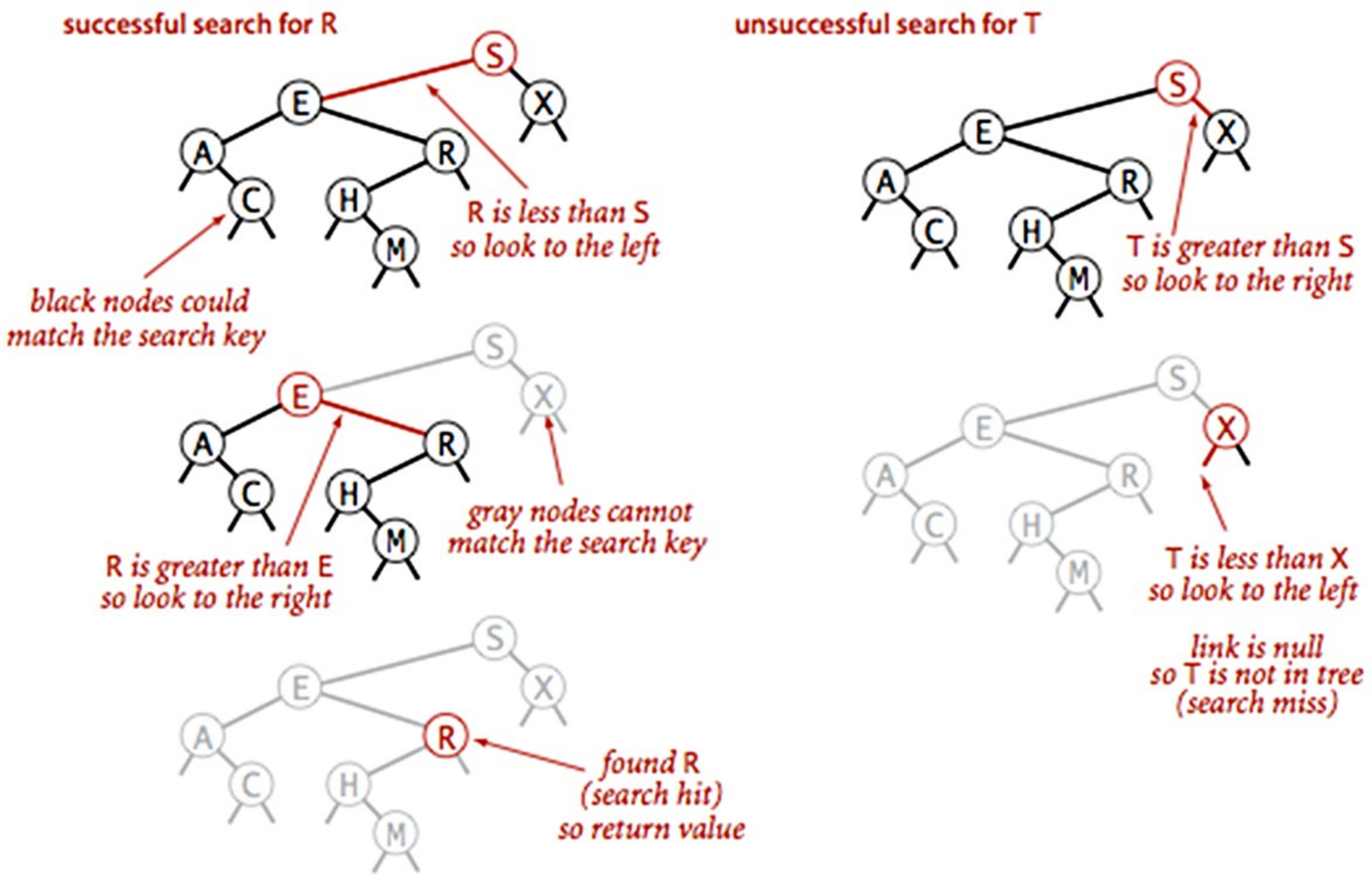
Outline

- Time Scaling
- Examples
- General performance strategies
- Computer architectures
- Measuring code speed
  - Reduce strength
  - Minimize array writes
- Profiling

Time Scaling for Algorithms

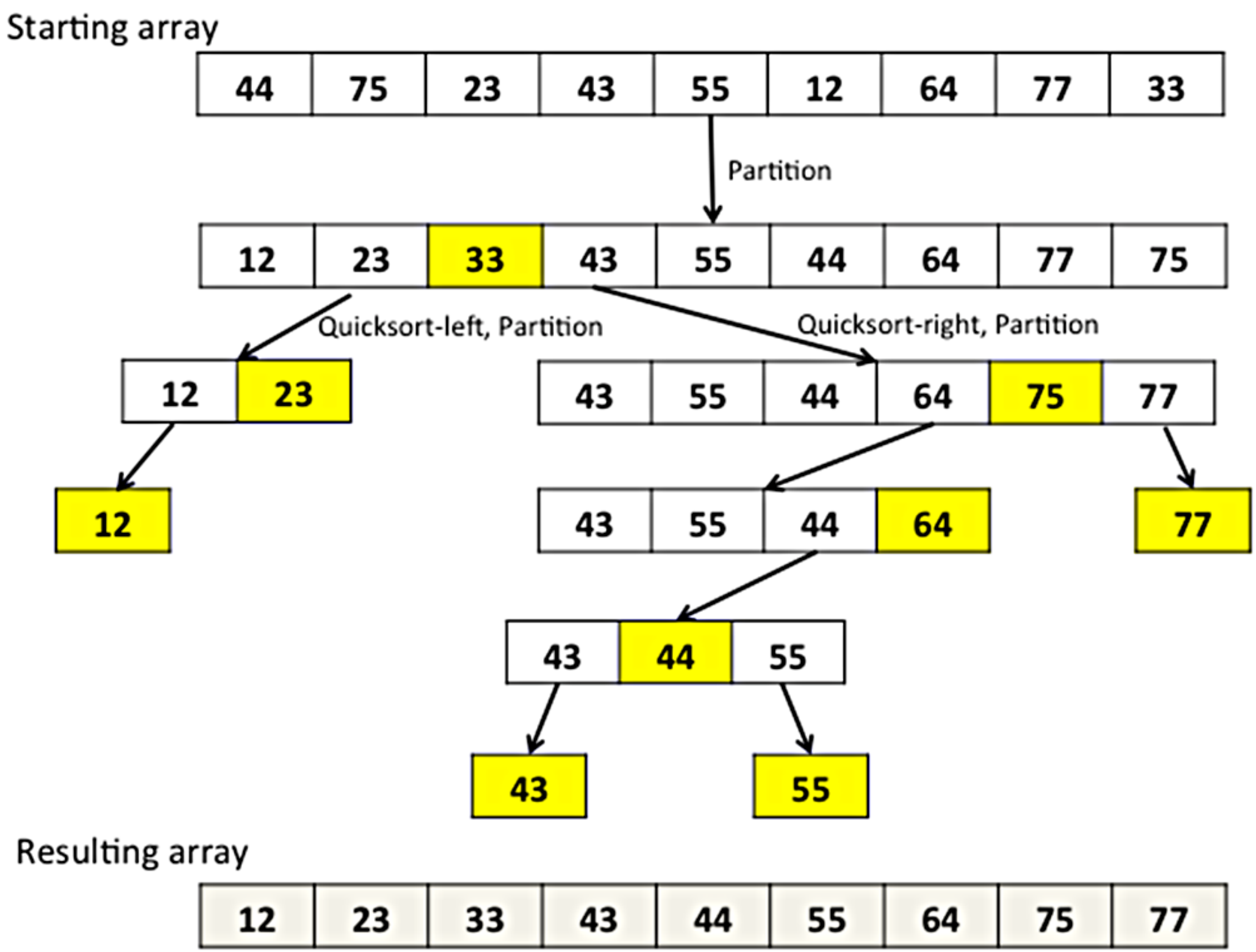
Complexity	Examples
1	Arithmetic
Log(N)	Search binary tree
N	Iterate over N elements
N log(N)	Quicksort; mergesort; FFT
N^2	Allocate N x N (2D) array
N^3	Matrix operations (Multiply / invert matrices)
2^N	Towers of Hanoi
$N!$	Traveling salesman problem

Binary Tree Search

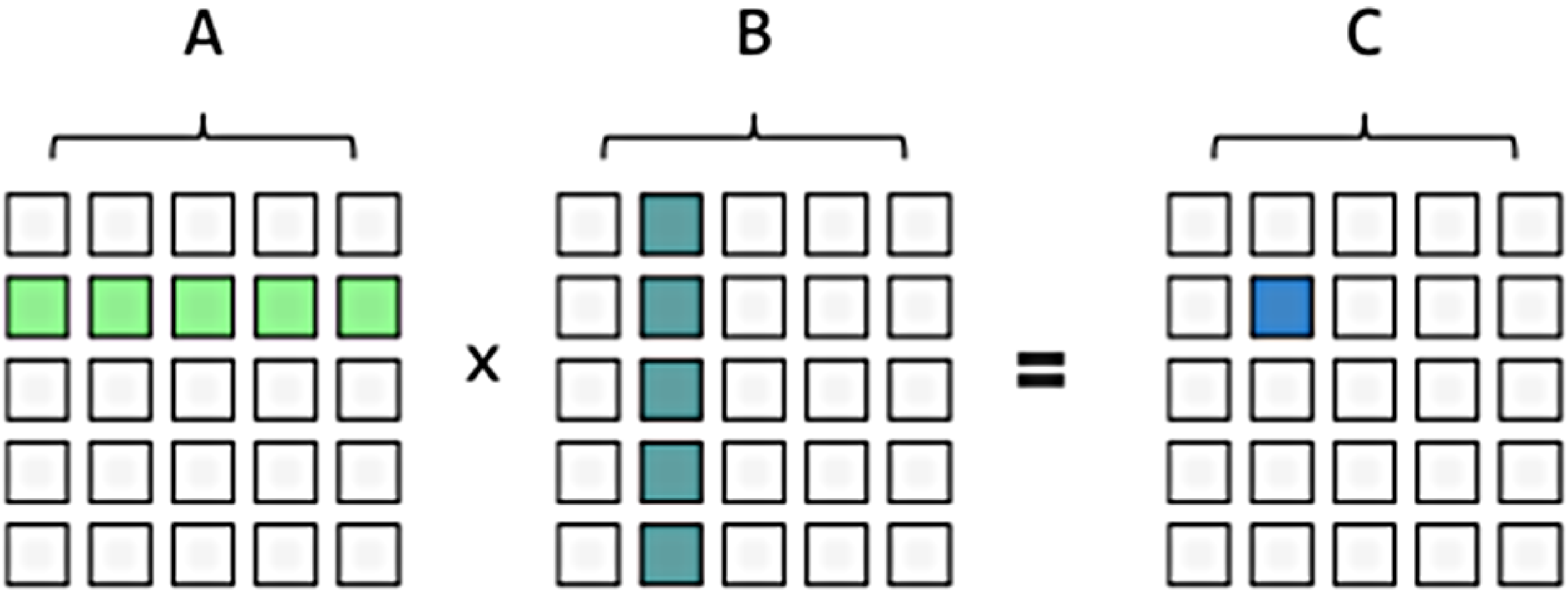


Successful (left) and unsuccessful (right) search in a BST

Quicksort

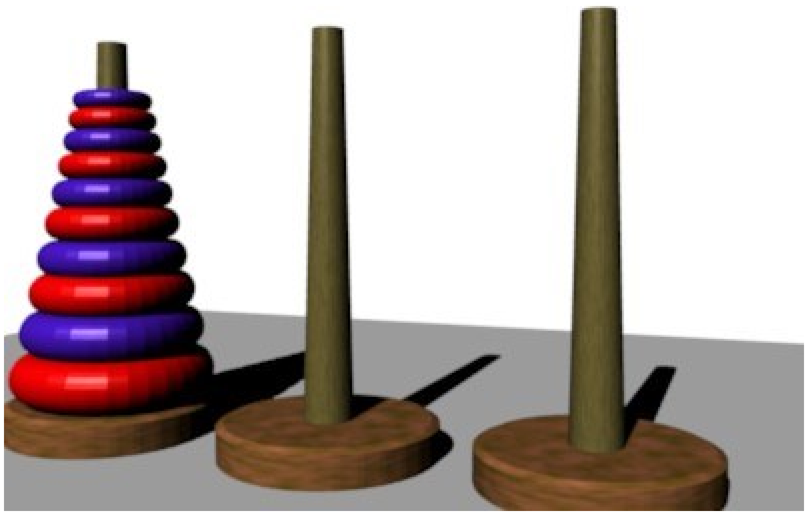

Starting array
44
75
23
43
55
12
64
77
33
Partition
12
23
33
43
55
44
64
77
75
Quicksort-left, Partition
Quicksort-right, Partition
12
23
43
55
44
64
75
77
12
43
55
44
64
77
43
44
55
43
55
Resulting array
12
23
33
43
44
55
64
75
77

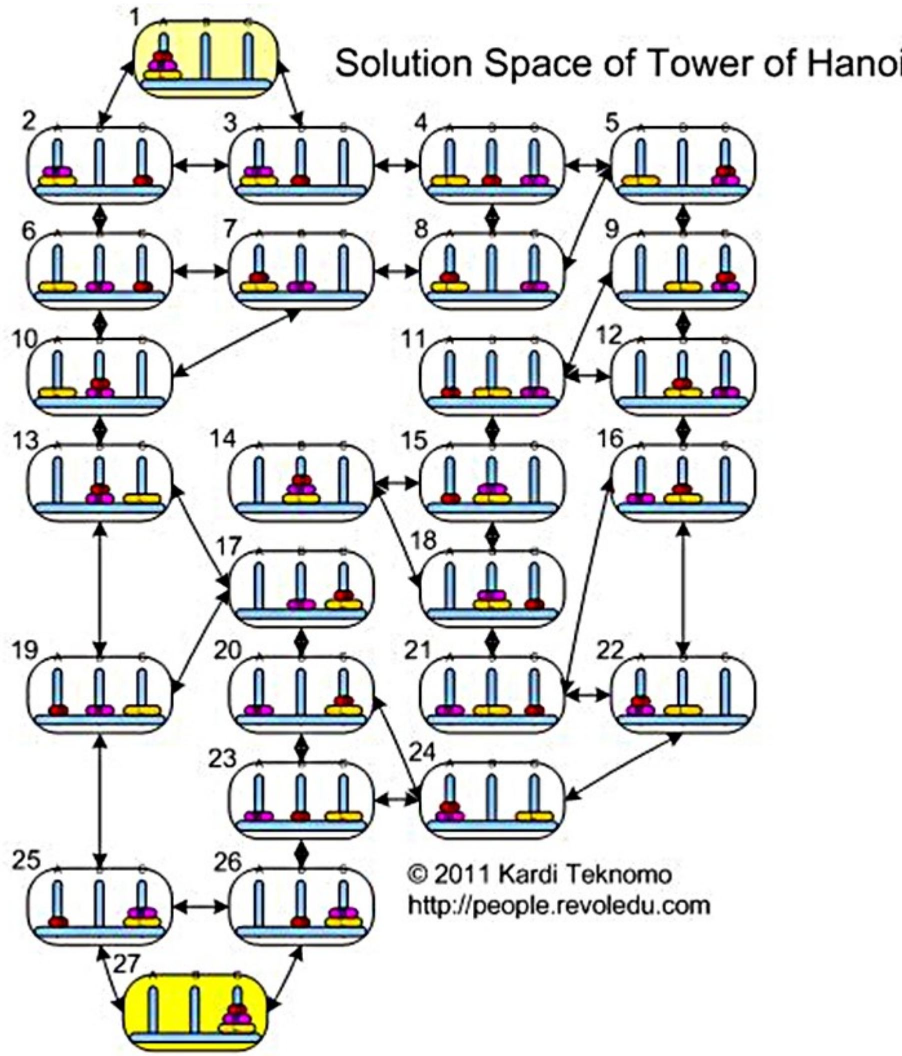
Matrix Multiplication



$$C[i][j] = \sum_{k=1}^{n} A[i][k] * B[k][j]$$

Towers of Hanoi





General Performance Strategies

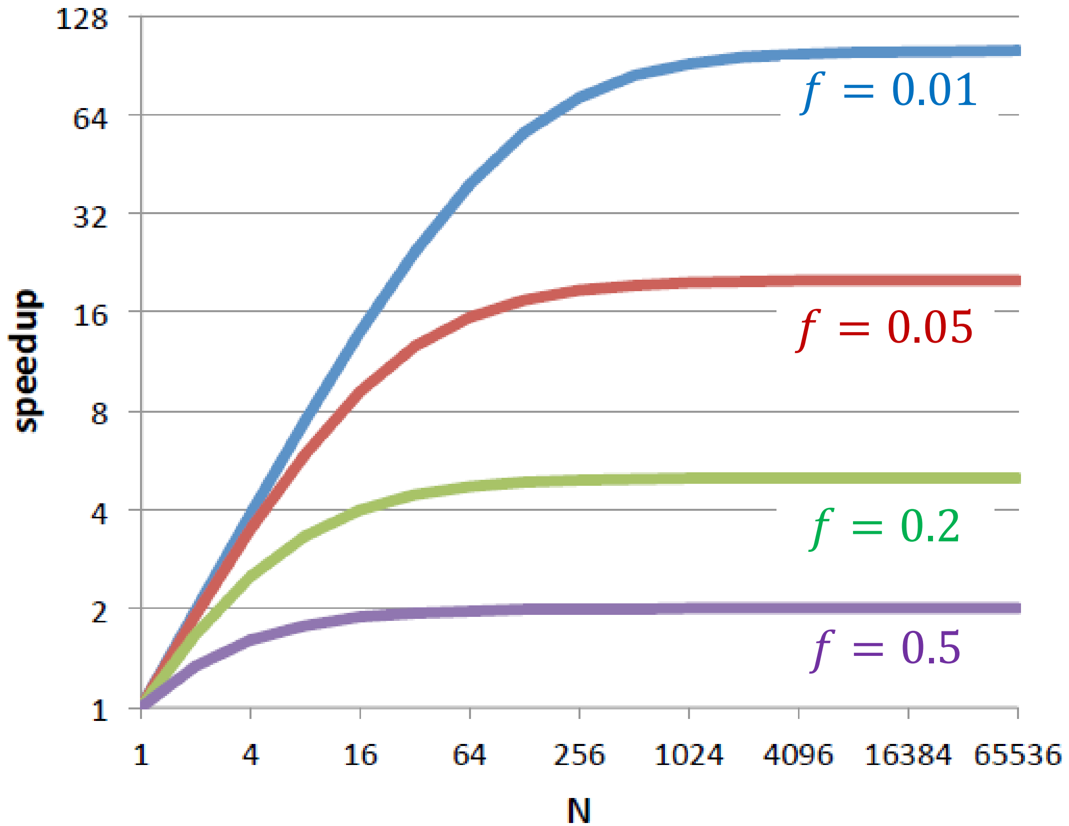
- Choose the best algorithm for each job
- Consider Ahmdal's law:

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

  – Make common case fastest (e.g., eliminate unnecessary steps at inner loops)

  – Make rare case correct (e.g., avoid mistakes in less frequent functions)

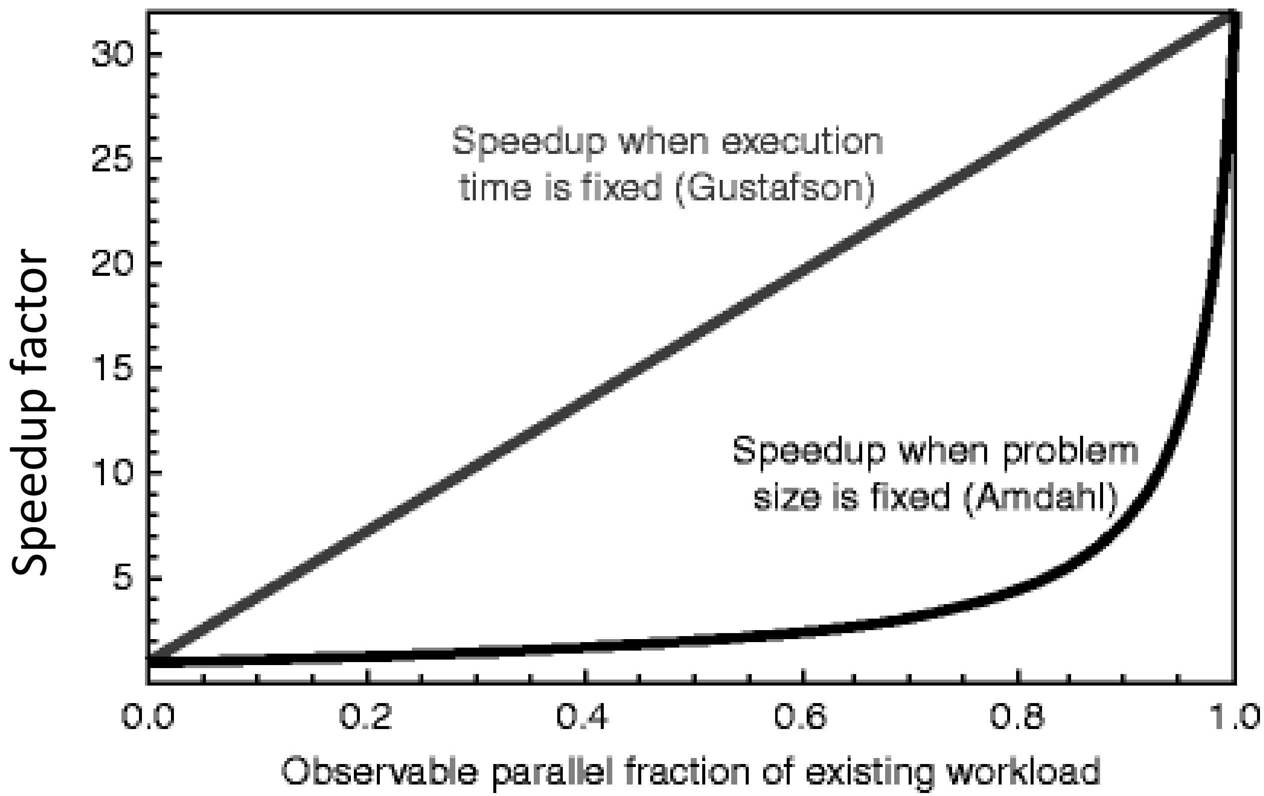
Ahmdal’s law



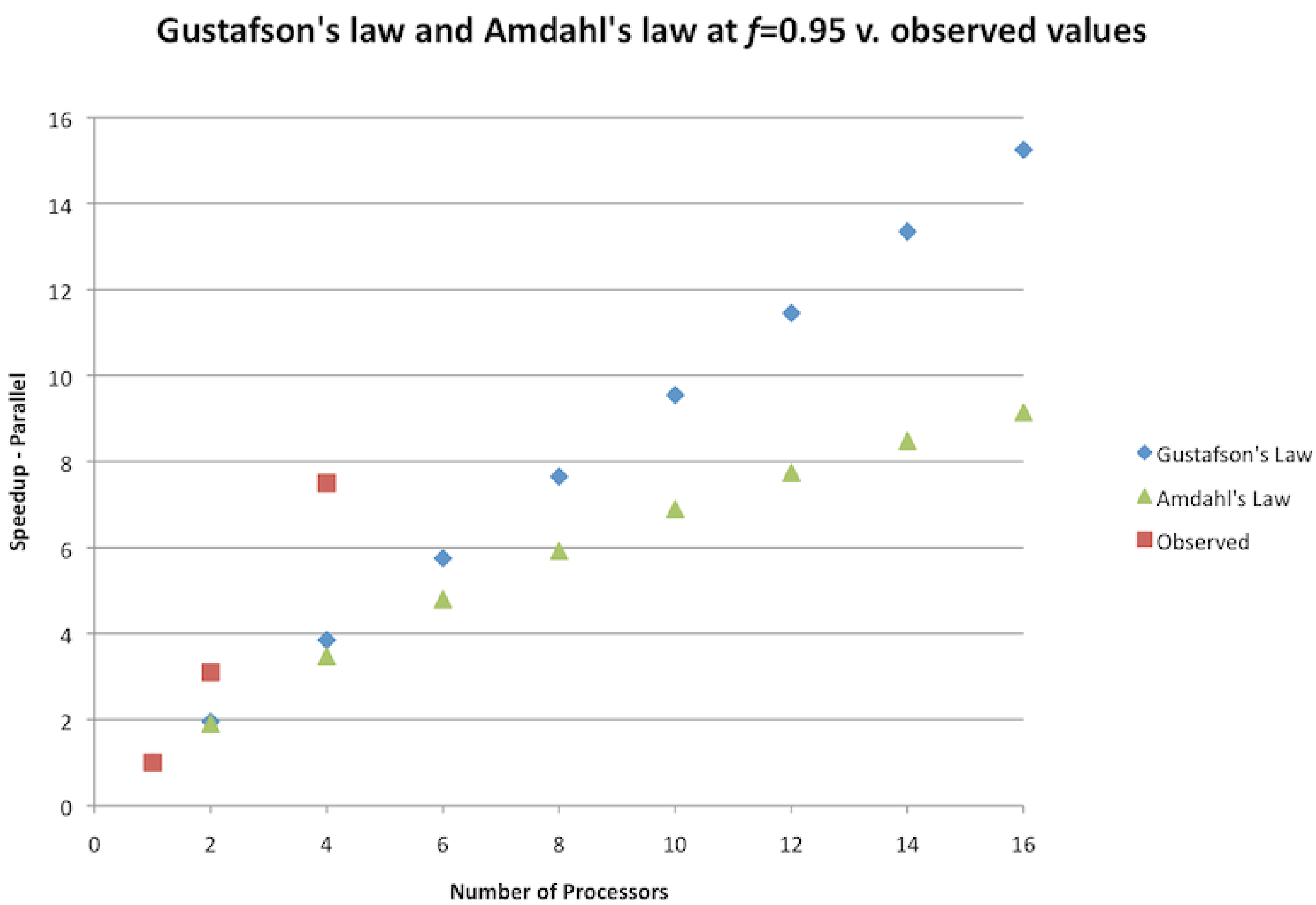
San Diego Supercomputer Center, Summer Institute Tutorial

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

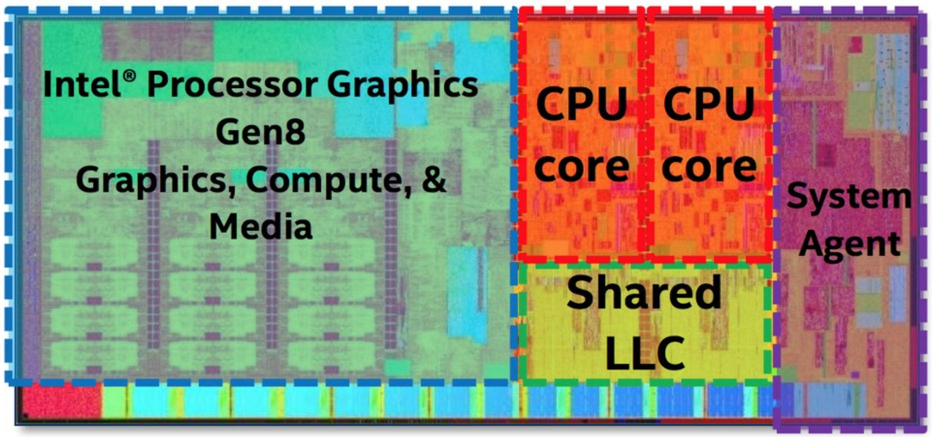
- A few good, but less known, things to do for fast code:
  - Prefer static linking and position-dependent code (as opposed to PIC, position-independent code).
  - Prefer 64-bit code and 32-bit data.
  - Prefer array indexing to pointers
  - Prefer regular memory access patterns.
  - Minimize control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

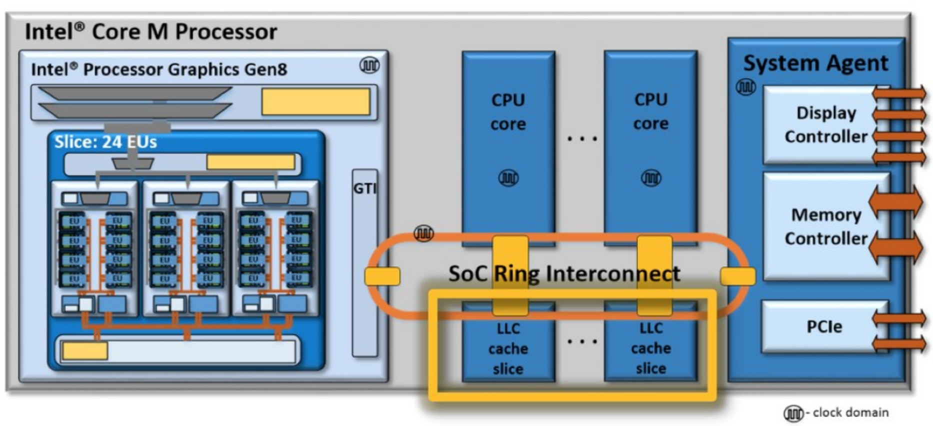
Computer Architectures

- 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 <u>measure</u> all important operations

Computer Architectures

Chip Level Architecture





- Ring Interconnect:
  - Dedicated "stops": each CPU Core, Graphics, & System Agent
  - Bi directional, 32 Bytes wide
- <u>Shared</u> Last Level Cache (LLC)
  - Both Graphics & CPU cores
  - Cache slices act like a single monolithic albeit distributed cache
  - 2-8MB, depending on product

IDF14 ComputerBase.de

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)int add, subtract, bitops, shift
  - floating point add, sub (separate unit!)
  - indexed array access (caveat: cache effects)
  - (u)int32 mul
  - FP mul
  - FP division, remainder
  - (u)int division, remainder

Reduce Strength: digits10

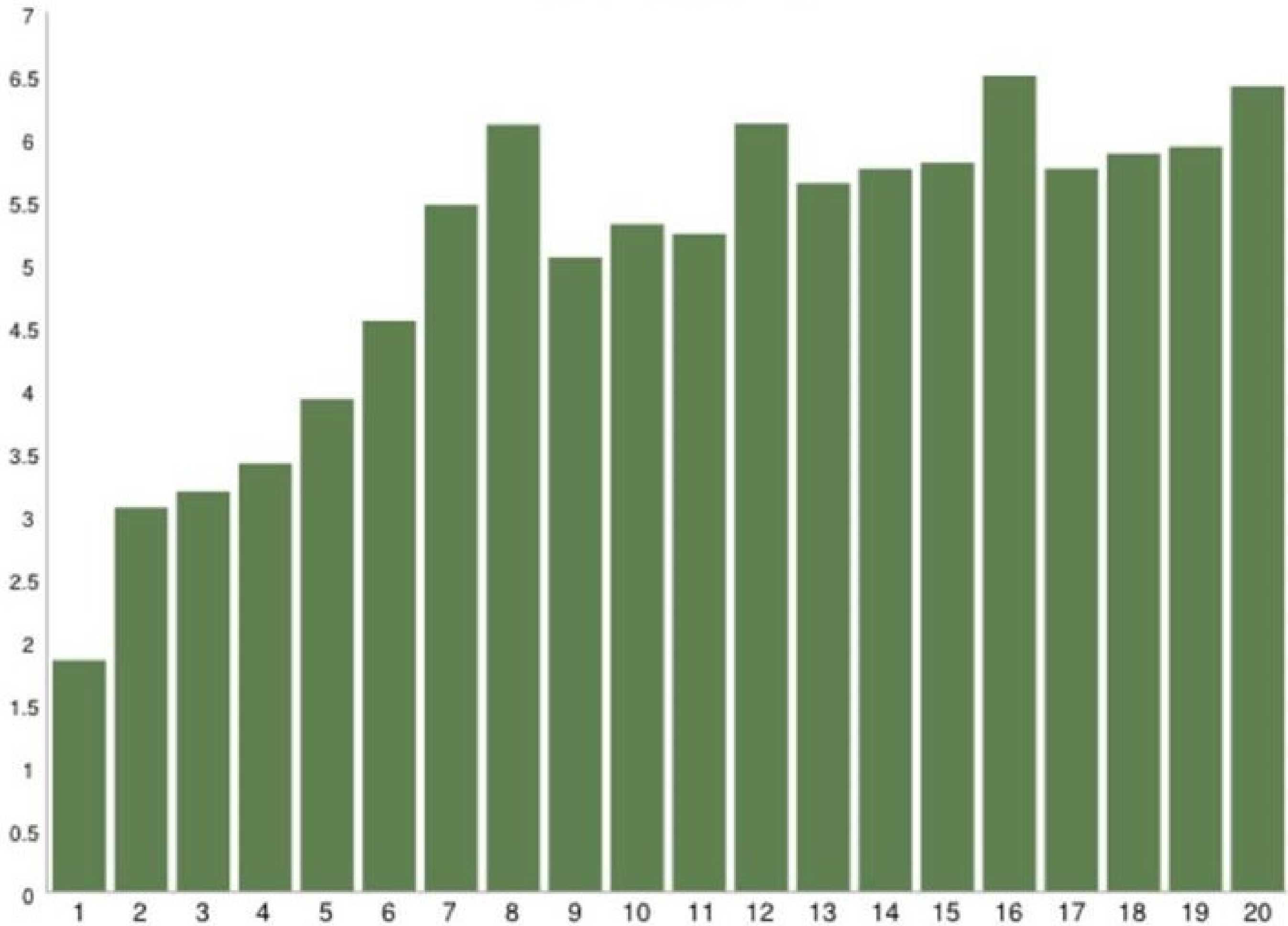
```
uint32_t digits10(uint64_t v) {
    uint32_t result = 0;
    do {
        ++result;
         v /= 10;
    } while (v);
     return result;
}
```

Basic algorithm using division

```
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;
  }
}
```

Reduced strength algorithm using comparison with division fallback

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

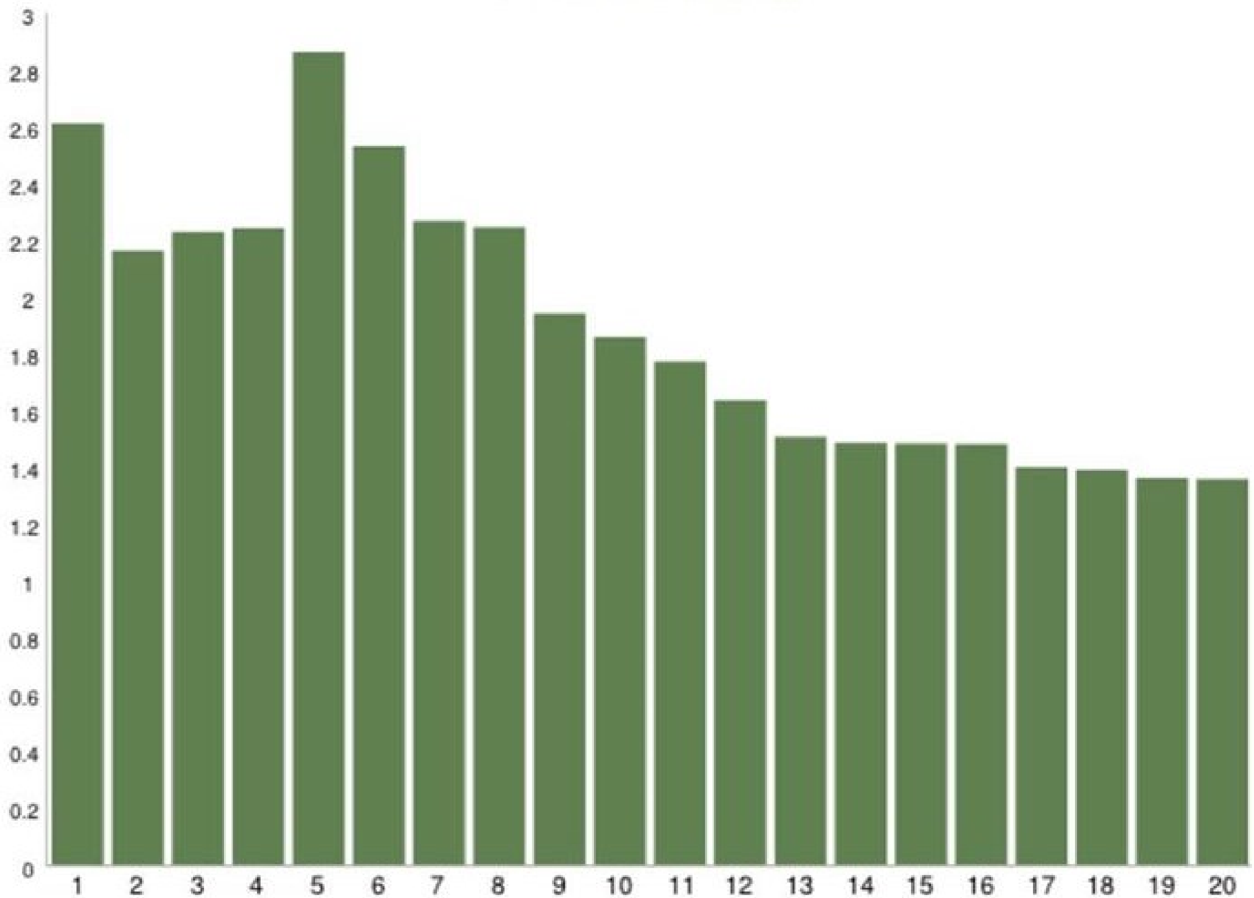
```
uint32_t u64ToAsciiClassic(uint64_t value,
char* dst) {
   // Write backwards.
   auto start = dst;
   do {
      *dst++ = ’0’ + (value % 10);
      value /= 10;
   } while (value != 0);
   const uint32_t result = dst - start;
   // Reverse in place.
   for (dst--; dst > start; start++, dst--) {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

Standard algorithm using reversal

```
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);</uint32_t>
      buffer[pos--] = ’0’ + r;
      v = q;
   }   assert(pos == 0); // Last digit is trivial to
handle
   *buffer = static_cast<uint32_t>(v) +
’0’;</uint32_t>
   return result;
}
```

Revised algorithm with less array writes

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:

https://software.intel.com/en-us/articles/practical-intel-avx-optimization-on-2nd-generation-intel-core-processors

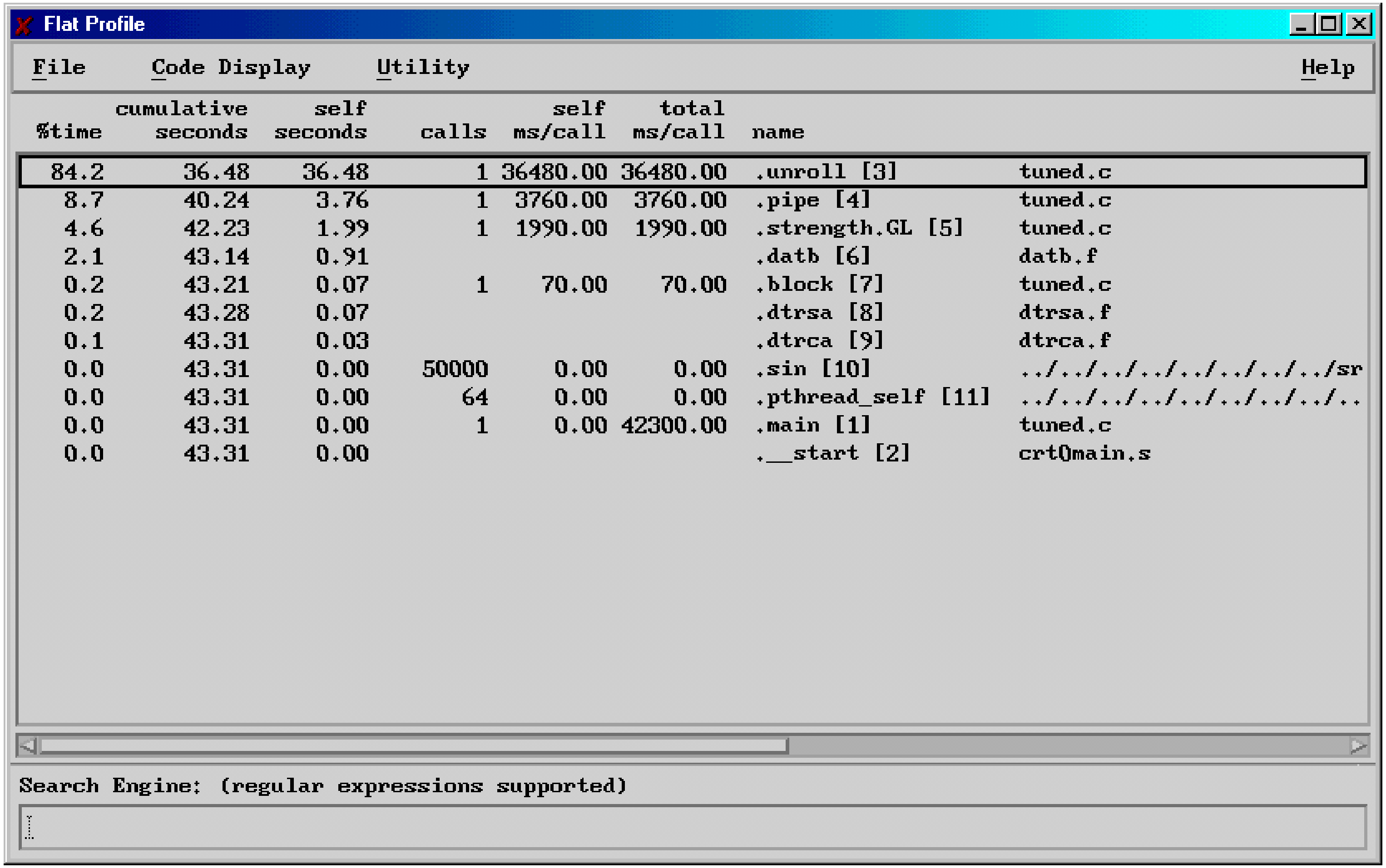
- Optimized subroutines in C/C++ (advanced):

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



%time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name	
84.2	36.48	36.48	1	36480.00	36480.00	.unroll [3]	tuned.c
8.7	40.24	3.76	1	3760.00	3760.00	.pipe [4]	tuned.c
4.6	42.23	1.99	1	1990.00	1990.00	.strength.GL [5]	tuned.c
2.1	43.14	0.91				.datb [6]	datb.f
0.2	43.21	0.07	1	70.00	70.00	.block [7]	tuned.c
0.2	43.28	0.07				.dtrsa [8]	dtrsa.f
0.1	43.31	0.03				.dtrca [9]	dtrca.f
0.0	43.31	0.00	50000	0.00	0.00	.sin [10]	../../../../../../../../sr
0.0	43.31	0.00	64	0.00	0.00	.pthread_self [11]	../../../../../../../../..
0.0	43.31	0.00	1	0.00	42300.00	.main [1]	tuned.c
0.0	43.31	0.00				.__start [2]	crt0main.s

Call Graph

Call Graph Profile

File Help

```
                              called/total       parents
index  %time    self descendents  called+self    name           index
                              called/total       children

                0.01       17.54     1/1            .__start [2]
[1]     99.0    0.01       17.54     1          .qmfd2 [1]
                6.07        0.00   101/101          .xsolve [3]
                5.72        0.00   101/101          .ysolve [4]
                2.81        0.00   101/101          .yrhs [5]
                2.42        0.00   101/101          .xrhs [6]
                0.36        0.05     1/1            .setup_matrix [7]
                0.03        0.08     1/1            .init_psi [9]

-----------------------------------------------
                                                  <spontaneous>
[2]     99.0    0.00       17.55                .__start [2]
                0.01       17.54     1/1            .qmfd2 [1]

-----------------------------------------------
                6.07        0.00   101/101          .qmfd2 [1]
[3]     34.2    6.07        0.00   101          .xsolve [3]

-----------------------------------------------
                5.72        0.00   101/101          .qmfd2 [1]
[4]     32.3    5.72        0.00   101          .ysolve [4]

-----------------------------------------------
                2.81        0.00   101/101          .qmfd2 [1]
[5]     15.8    2.81        0.00   101          .yrhs [5]

-----------------------------------------------
                2.42        0.00   101/101          .qmfd2 [1]
[6]     13.6    2.42        0.00   101          .xrhs [6]

-----------------------------------------------
```

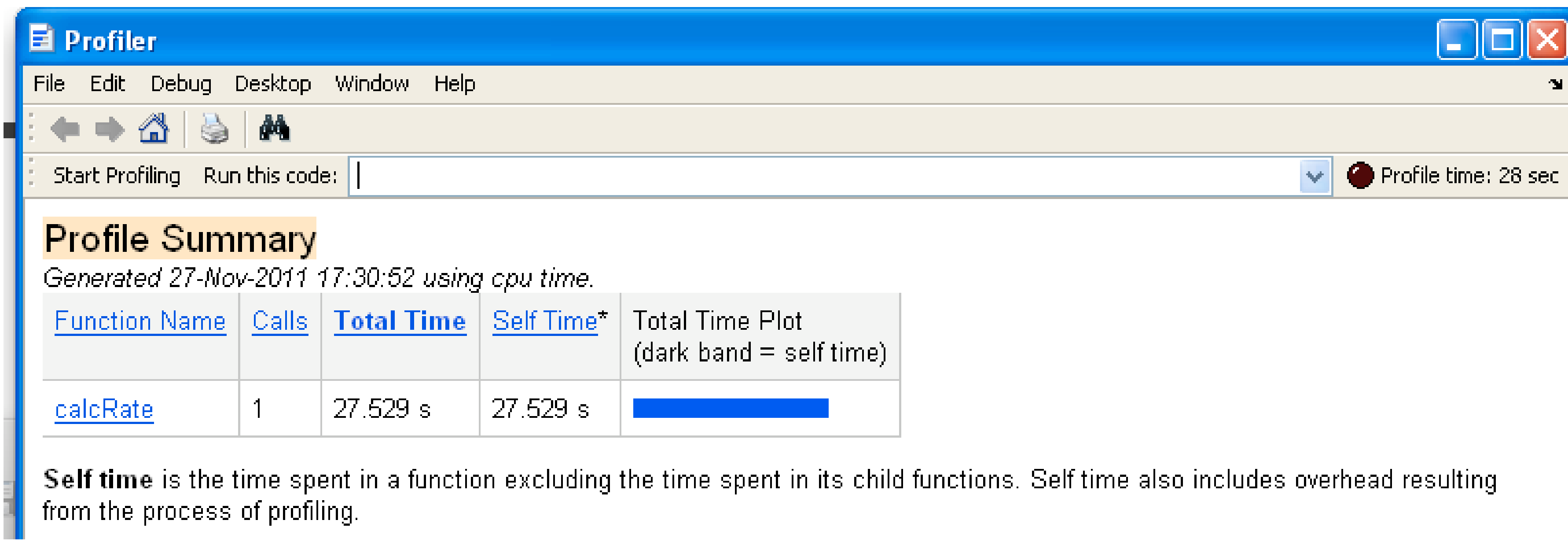
Search Engine: (regular expressions supported)

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')
```



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?
  - http://www.mathworks.com/support/solutions/en/data/1-15NM7/?solution=1-15NM7
- Improving the Speed of MATLAB Calculations
  - http://web.cecs.pdx.edu/~gerry/MATLAB/programming/performance.html

Next Class

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