Program Optimization

Textbook: Section 5

Today

- **Overview**
- **Generally Useful Optimizations**
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions (in text)
  - Removing unnecessary procedure calls (in text)
- **Optimization Blockers**
  - Procedure calls
  - Memory aliasing
- **Profiling**

Performance Realities

- ***There's more to performance than asymptotic complexity (Big O)***

- **Constant factors matter too!**
  - Easily see 10:1 performance range depending on how code is written
  - Must optimize at multiple levels:
    - algorithm, data representations, procedures, and loops
- **Must understand system to optimize performance**
  - How programs are compiled and executed
  - How modern processors + memory systems operate
  - How to measure program performance and identify bottlenecks
    - This is what we will focus on at the lab
  - How to improve performance without destroying code modularity and generality

Compiler optimizers

- **Provide efficient mapping of program to machine**
  - register allocation
  - code selection and ordering (scheduling)
  - dead code elimination
  - eliminating minor inefficiencies
- **Don’t (usually) improve asymptotic efficiency**
  - up to programmer to select best overall algorithm
  - big-O savings are (often) more important than constant factors
    - but constant factors also matter
- **Have difficulty overcoming “optimization blockers”**
  - potential memory aliasing
  - potential procedure side-effects
  - programmer needs to manually optimize if there is an optimization blocker

Limitations of Optimizing Compilers

- **Operate under fundamental constraint**
  - Must not cause any change in program behavior

- **Most analysis is performed only within procedures**
  - Whole-program analysis is too expensive in most cases
  - Newer versions of GCC do interprocedural analysis within individual files
    - But, not between code in different files
- **Most analysis is based only on *static* information**
  - Compiler has difficulty anticipating run-time inputs

- **When in doubt, the compiler must be conservative**

Today

- **Overview**
- **Generally Useful Optimizations**
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions (in text)
  - Removing unnecessary procedure calls (in text)
- **Optimization Blockers**
  - Procedure calls
  - Memory aliasing
- **Dealing with Conditionals**

Generally Useful Optimizations

- **Optimizations that you or the compiler should do regardless of processor / compiler**

- **Code Motion**
  - Reduce frequency with which computation performed
    - If it will always produce same result
    - Especially moving code out of loop

```
void set_row(double *a, double *b,
   long i, long n)
{
    long j;
    for (j = 0; j < n; j++)
         a[n*i+j] = b[j];
}
```

$\longrightarrow$

```
    long j;
    int ni = n*i;
    for (j = 0; j < n; j++)
         a[ni+j] = b[j];
```

Reduction in Strength

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide

  ```
  16*x   -->    x << 4
  ```

  - Utility machine dependent
  - Depends on cost of multiply or divide instruction
    - On Intel Nehalem, integer multiply requires 3 CPU cycles
- Recognize sequence of products

```
for (i = 0; i < n; i++) {
  int ni = n*i;
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
}
```

$\longrightarrow$

```
int ni = 0;
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
  ni += n;
}
```

Multiplication is replaced by addition

Today

- Overview
- Generally Useful Optimizations
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions (in text)
  - Removing unnecessary procedure calls (in text)
- **Optimization Blockers**
  - Procedure calls
  - Memory aliasing
- Profiling

Optimization Blocker #1: Procedure Calls

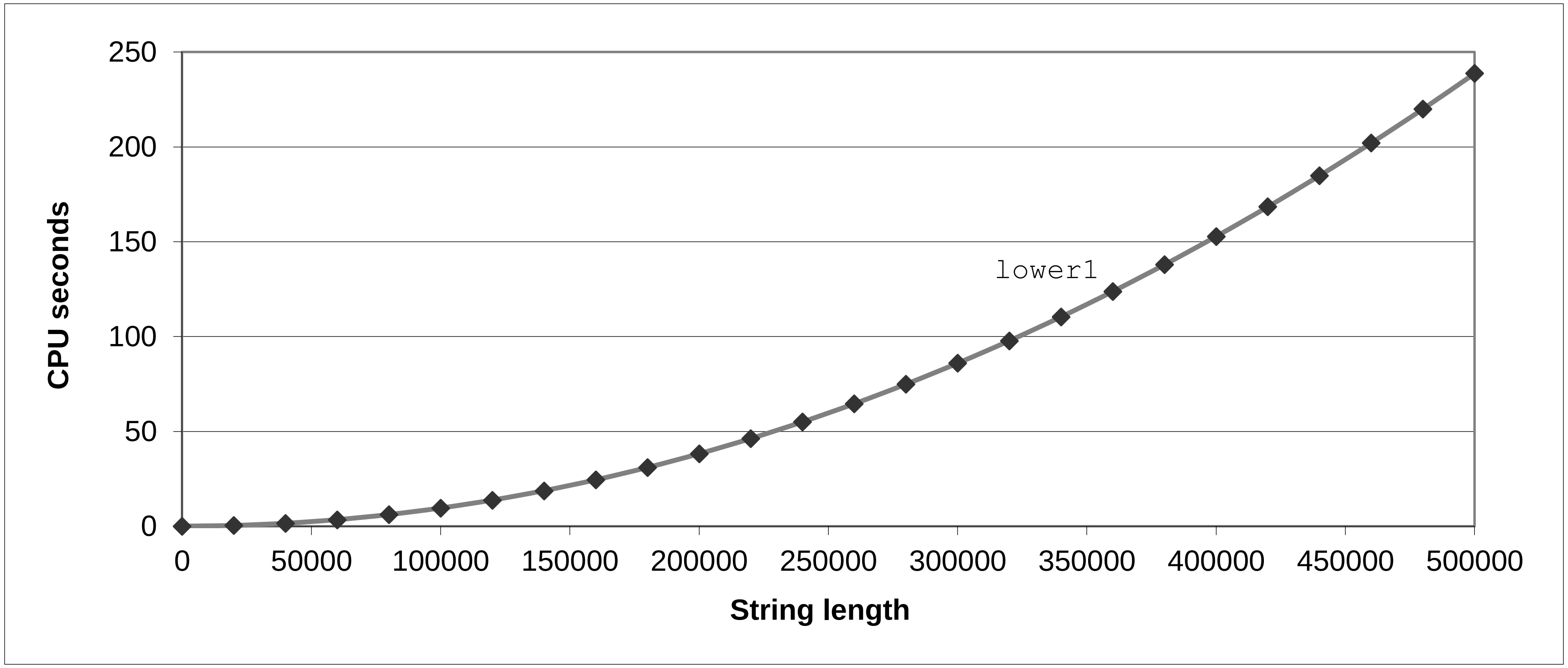
- **Limited optimization near function calls**
  - Procedure to Convert String to Lower Case

```
void lower(char *s)
{
  size_t i;
  for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

strlen needs to be executed at every iteration

Lower Case Conversion Performance

- Time quadruples when double string length
- Quadratic performance



Calling Strlen

```
/* My version of strlen */
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0') {
       s++;
       length++;
    }
    return length;
}
```

- **Strlen performance**
  - Only way to determine length of string is to scan its entire length, looking for null character.
- **Overall performance, string of length N**
  - N calls to strlen
  - Require times N, N-1, N-2, …, 1
  - Overall O(N^2) performance

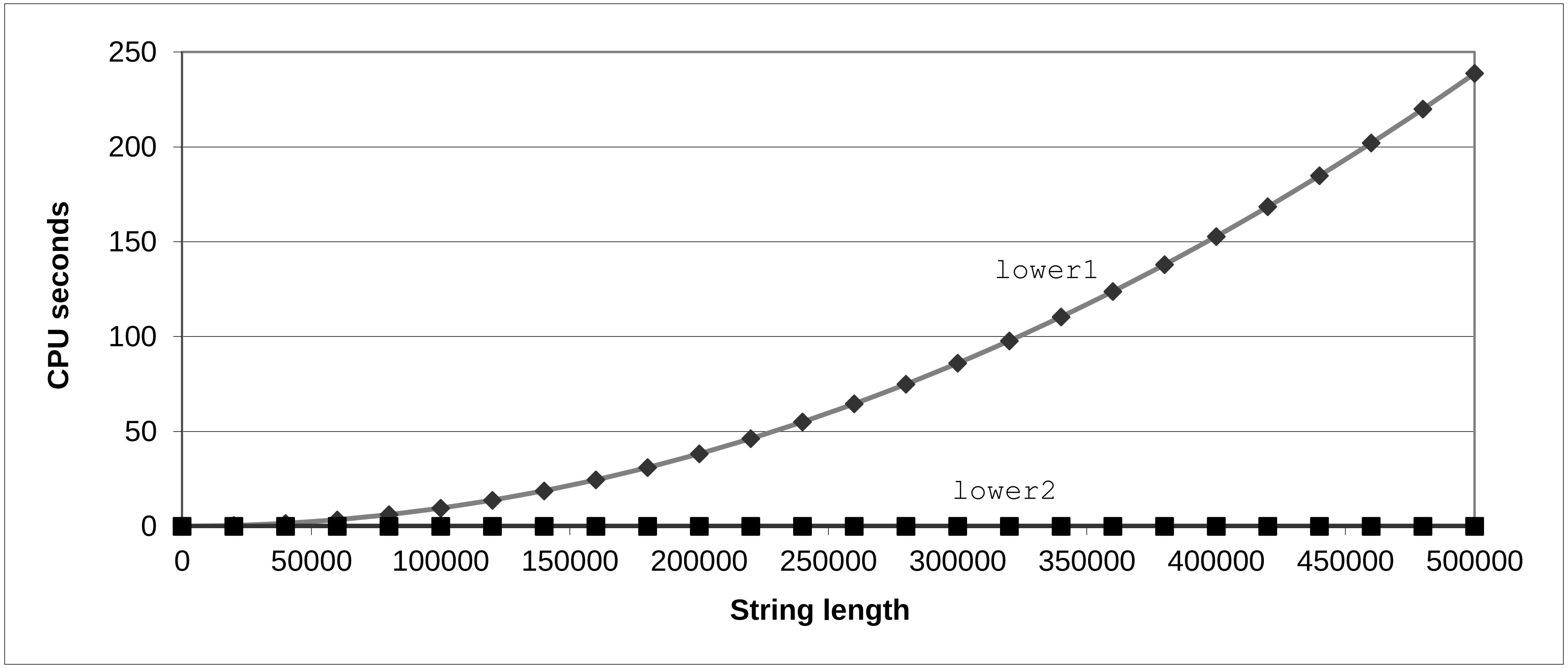
Improving Performance

```
void lower(char *s)
{
  size_t i;
  size_t len = strlen(s);
  for (i = 0; i < len; i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

- Move call to `strlen` outside of loop
- Since result does not change from one iteration to another
- Form of code motion

Lower Case Conversion Performance

- Time doubles when double string length
- Linear performance of lower2



Optimization Blocker#1: Procedure Calls

- ***Why couldn't compiler move `strlen` out of inner loop?***
  - Procedure may have side effects
    - Alters global state each time called
  - Function may not return same value for given arguments

- **Warning:**
  - Compiler treats procedure call as a black box
  - Weak optimizations near them

- **Remedies:**
  - Do your own code motion

Optimization Blocker#2: Memory Aliasing

- **Aliasing**
  - Two different memory references specify single location
  - Easy to have happen in C
    - Since allowed to do address arithmetic
    - Direct access to storage structure

```
void twiddle1(int *xp, int *yp)
{
    *xp += *yp;
    *xp += *yp;
}

void twiddle2(int *xp, int *yp)
{
    *xp += 2* *yp;
}
```

function twiddle2 is more efficient.

- twiddle2 requires only three memory references (read *xp, read *yp, write *xp),

whereas

- twiddle1 requires six (two reads of *xp, two reads of *yp, and two writes of *xp).

Optimization Blocker#2: Memory Aliasing

- Consider, however, the case in which xp and yp are equal. Then function
- twiddle1 will perform the following computations:

```
*xp += *xp;   /* Double value at xp */
*xp += *xp;   /* Double value at xp */
```

- The result will be that the value at xp will be increased by a factor of 4. On the other hand, function twiddle2 will perform the following computation

```
*xp += 2* *xp;   /* Triple value at xp */
```

Getting High Performance

- **Good compiler and flags**
- **Don’t do anything stupid**
  - Watch out for hidden algorithmic inefficiencies
  - Write compiler-friendly code
    - Watch out for optimization blockers:
      procedure calls & memory references
  - Look carefully at innermost loops (where most work is done)

- **Tune code for machine (will not be covered in this course)**
  - Exploit instruction-level parallelism
  - Avoid unpredictable branches
  - Make code cache friendly

Today

- **Overview**
- **Generally Useful Optimizations**
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions (in text)
  - Removing unnecessary procedure calls (in text)
- **Optimization Blockers**
  - Procedure calls
  - Memory aliasing
- **Profiling**

Profiling

- Debugging helps the programmer find and fix bugs …
- *Profiling* helps the programmer find inefficiencies

- Profiling involves running a version of the program instrumented with code to measure:
  - How much time is spent in certain areas of the code.

Gprof

- gprof is a utility that measures a program's performance and behavior.
- This produces *non-interactive* profile output

- The output provides detail:
- The time that the program ran, and time for each function
  - Statistics and detail on "*performance*"

- Which functions called other function and how many times
  - Statistics and detail on the "*call graph*"

Running gprof

- 1. First compile program using the “-pg” flag:

```
$ gcc -pg profiling.c –o profiling
```

- 2. Run the program (will generate file gmon.out):

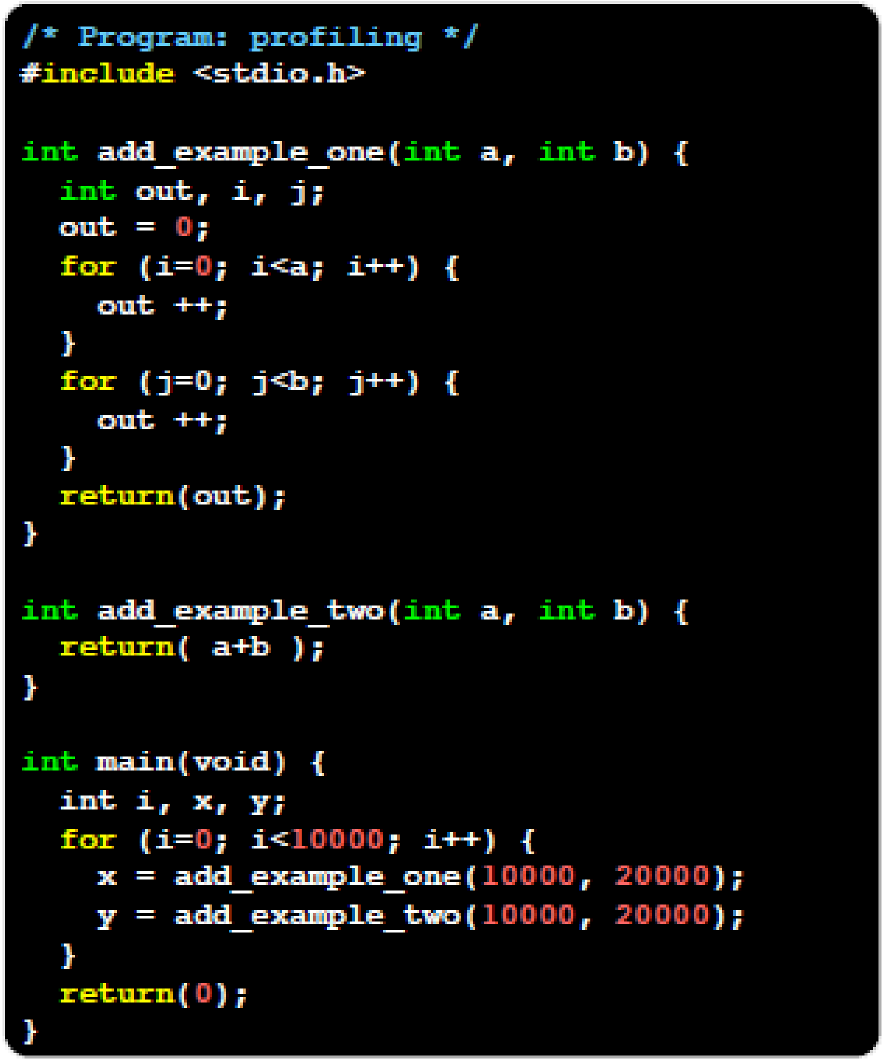
```
$ ./profiling
```

- 3. Run gprof with the named program

```
$ gprof profiling | less
```

- 4. Review the output
- 5. Optimize the program, re-profile
- 6. GOTO step#1

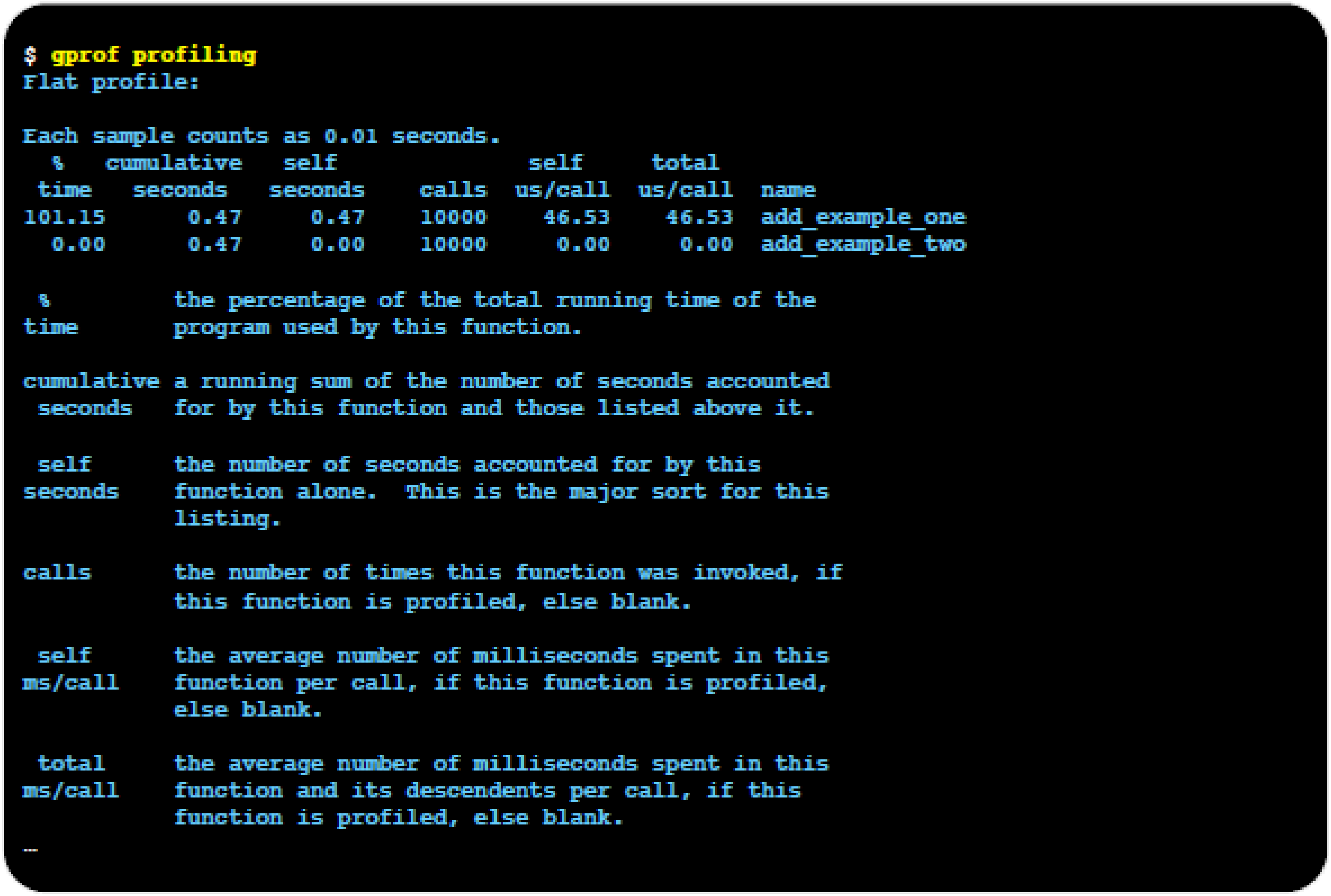
Example Inefficient Program



```c
/* Program: profiling */
#include <stdio.h>

int add_example_one(int a, int b) {
  int out, i, j;
  out = 0;
  for (i=0; i<a; i++) {
    out ++;
  }
  for (j=0; j<b; j++) {
    out ++;
  }
  return(out);
}

int add_example_two(int a, int b) {
  return( a+b );
}

int main(void) {
  int i, x, y;
  for (i=0; i<10000; i++) {
    x = add_example_one(10000, 20000);
    y = add_example_two(10000, 20000);
  }
  return(0);
}
```

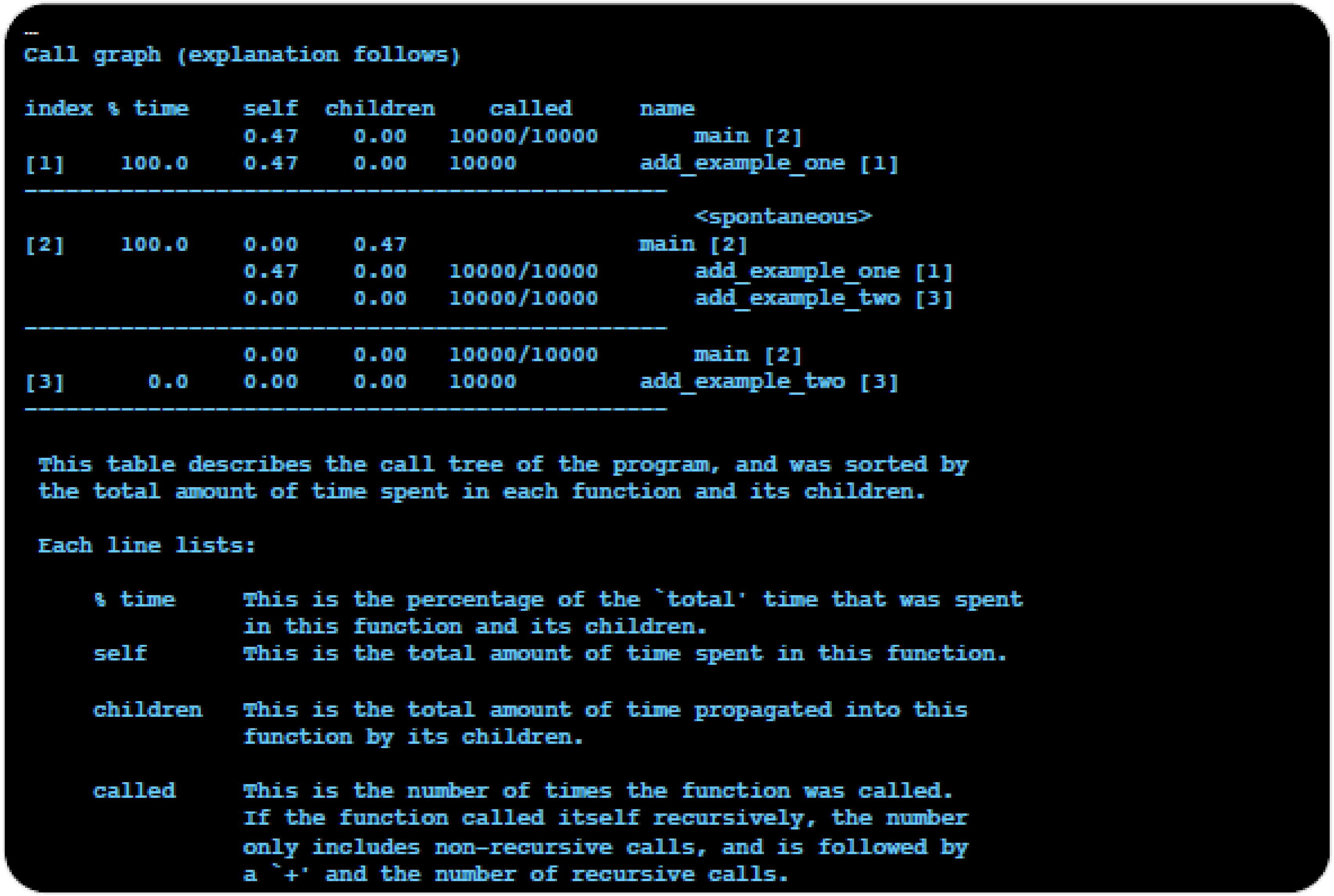
- Which function is going to run slow?

Gprof (flat profile)



```
$ gprof profiling
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
101.15      0.47     0.47    10000    46.53    46.53  add_example_one
  0.00      0.47     0.00    10000     0.00     0.00  add_example_two

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
           else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
           function is profiled, else blank.
…
```

Gprof (Call Graph)

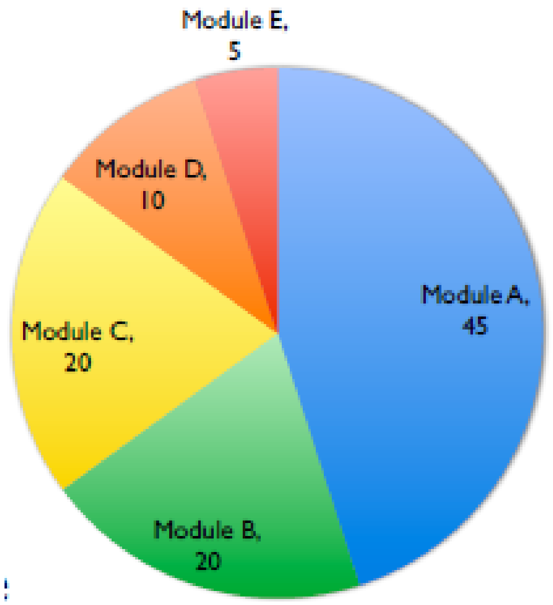


```
…
Call graph (explanation follows)

index % time    self  children    called     name
                0.47    0.00   10000/10000       main [2]
[1]    100.0    0.47    0.00   10000         add_example_one [1]
-----------------------------------------------
                                                 <spontaneous>
[2]    100.0    0.00    0.47                 main [2]
                0.47    0.00   10000/10000       add_example_one [1]
                0.00    0.00   10000/10000       add_example_two [3]
-----------------------------------------------
                0.00    0.00   10000/10000       main [2]
[3]      0.0    0.00    0.00   10000         add_example_two [3]
-----------------------------------------------

 This table describes the call tree of the program, and was sorted by
 the total amount of time spent in each function and its children.

 Each line lists:

     % time     This is the percentage of the `total' time that was spent
                in this function and its children.
     self       This is the total amount of time spent in this function.

     children   This is the total amount of time propagated into this
                function by its children.

     called     This is the number of times the function was called.
                If the function called itself recursively, the number
                only includes non-recursive calls, and is followed by
                a `+' and the number of recursive calls.
```

Optimization revisited…

- When optimizing, you focus on modules of the program which implement the features and processing of the program.
- Which parts of the program you select depends on what parts are running the most.
- Then, focus on those parts which take up the most time.

- Profiling tells us where to spend our time.



How to measure speedup?

- **We use Amdahl's Law:**
  - The overall speedup T of the program is expressed

$$T = \frac{1}{(1-k) + \frac{k}{s}}$$

- k is the percentage of total execution time spent in the optimized module(s).
- s is the execution time expressed in terms of a n-factor speedup

Example

- Assume that a module A of a program is optimized.
  - A represents 45% of the run time of the program.
  - The optimization reduces the runtime of module from 750 ms to 50ms

s = 750/50 = 15

k = .45

$$T = \frac{1}{(1-k)+\frac{k}{s}}$$

$$T = \frac{1}{(1-.45)+\frac{.45}{15}} = \frac{1}{.55+.03} = 1.724$$