

# Profilers and Bottlenecks

## ICS432 - Fall 2008 Concurrent and High-Performance Programming

Henri Casanova (henric@hawaii.edu)

### Good Performance?

- You have a code that was given to you or that you wrote
- You compile it with your favorite optimizing compiler, you have removed obvious optimization blockers
- And then, performance is “poor”
  - Not sufficient for the code to be used to meet deadlines
  - The code could still be usable but lead to long waits, and you can tell that the performance is way below the peak performance
- What do you do?

### Why is Performance Poor?

- Performance is poor because the code suffers from a **performance bottleneck**
- Definition
  - An application runs on a platform that has many components
    - CPU, Memory, Operating System, Network, Hard Drive, Video Card, etc.
  - Pick a component and make it faster
  - If the application performance increases, that component was the bottleneck!

### Identifying a Bottleneck

- It can be difficult
  - You're not going to change the memory bus just to see what happens to the application
  - But you can run the code on a different machine and see what happens
- Typical Approach
  - Know/discover the characteristics of the machine
  - Know/discover the characteristics of the application
  - Observe the application execution on the machine
  - Reason about what the bottleneck is
- Luckily there are well-known bottlenecks that are likely candidates when performance is poor

### Removing a Bottleneck

- Brute force: Hardware Upgrade
  - Sometimes necessary, but can only get you so far and may be very costly
    - e.g., memory technology
- Instead, modify the code
  - The bottleneck is there because the code uses a “resource” heavily or in non-intelligent manner
  - This is, unfortunately, what we have to do often after the fact
    - You wrote a beautifully structured/modular code
    - It's slow and you have to decrease readability, modularity to increase performance

### The Memory Bottleneck

- The memory is a very common bottleneck that beginning programmers often don't think about
  - When you look at code, you often pay more attention to computation
  - $a[i] = b[j] + c[k]$ 
    - The access to the 3 arrays take more time than doing an addition
    - For the code above, the memory is the bottleneck for many machines!

## Why the Memory Bottleneck?

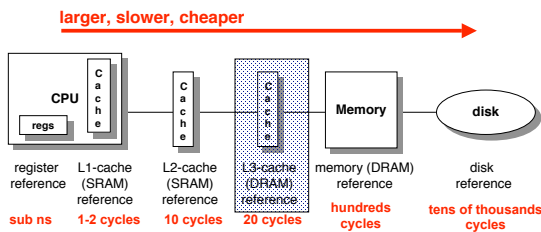
- In the 70's, everything was balanced
  - The memory kept pace with the CPU
    - n cycles to execute an instruction, n cycles to bring in a word from memory
- No longer true
  - CPUs have gotten 1,000x faster
  - Memory have gotten 10x faster and 1,000,000x larger
- ➔ Flops are free and bandwidth is expensive and processors are **STARVED** for data

## Memory Bottleneck: Example

- Fragment of code:  $a[i] = b[j] + c[k]$ 
  - Three memory references: 2 reads, 1 write
  - One addition: can be done in one cycle
- If the memory bandwidth is 12.8GB/sec, then the rate at which the processor can access integers (4 bytes) is:  $12.8 * 1024 * 1024 * 1024 / 4 = 3.4\text{GHz}$
- The above code needs to access 3 integers
- Therefore, the rate at which the code gets its data is  $\sim 1.1\text{GHz}$
- But the CPU could perform additions at 4GHz!
- Therefore: **The memory is the bottleneck**
  - And we assumed memory worked at the peak!!!
  - We ignored other possible overheads on the bus
  - In practice the gap can be around a factor 15 or higher

## Reducing the Memory Bottleneck

- The way in which computer architects have dealt with the memory bottleneck is via the memory **hierarchy**



## Locality

- The memory hierarchy is useful because of “locality”
- **Temporal locality**: a memory location that was referenced in the past is likely to be referenced again
- **Spatial locality**: a memory location next to one that was referenced in the past is likely to be referenced in the near future
- This is great, but what we write our code for performance we want our code to have the **maximum** amount of locality
  - The compiler can do some work for us regarding locality
  - But unfortunately not everything

## Programming for Locality

- Essentially, a programmer should keep a mental picture of the memory layout of the application, and reason about locality
  - When writing concurrent code on a multi-core architecture, one should also think of which caches are shared/private
- This can be extremely complex, but there are a few well-known techniques
- The typical example is with 2-D arrays

## Example: 2-D Array Initialization

```

int a[200][200];
for (i=0; i<200; i++) {
    for (j=0; j<200; j++) {
        a[i][j] = 2;
    }
}

int a[200][200];
for (j=0; j<200; j++) {
    for (i=0; i<200; i++) {
        a[i][j] = 2;
    }
}
    
```

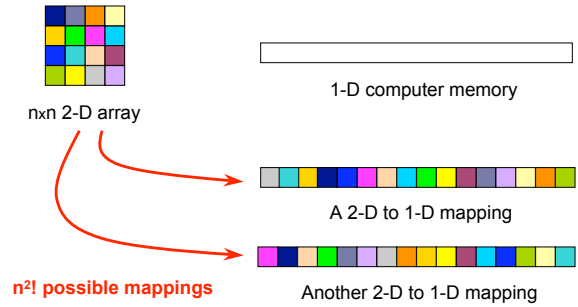
- Which alternative is best?
  - i,j?
  - j,i?
- To answer this, one must understand the **memory layout** of a 2-D array

## 2-D Arrays in Memory

- A static 2-D array is one declared as  

```
<type> <name>[<size>][<size>]
int myarray[10][30];
```
- The elements of a 2-D array are stored in **contiguous** memory cells
- The problem is that:
  - The array is 2-D, conceptually
  - Computer memory is 1-D
- 1-D computer memory: a memory location is described by a single number, its address
  - Just like a single axis
- Therefore, there must be a mapping from 2-D to 1-D
  - From a 2-D abstraction to a 1-D implementation

## Mapping from 2-D to 1-D?



## Row-Major, Column-Major

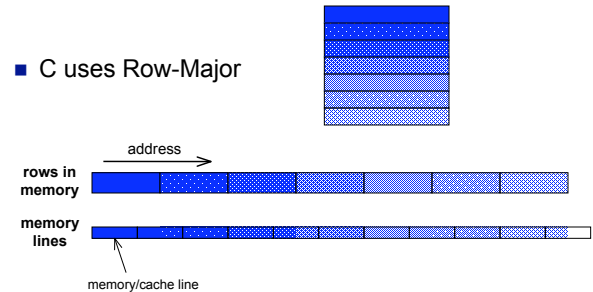
- Luckily, only 2 of the  $n^2!$  mappings are ever implemented in a language



- Row-Major:
  - Rows are stored contiguously
- Column-Major:
  - Columns are stored contiguously

## Row-Major

- C uses Row-Major



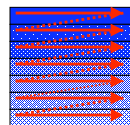
- Matrix elements are stored in contiguous memory lines

## Row-Major

- C uses Row-Major

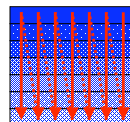
### First option

```
int a[200][200];
for (i=0; i<200; i++)
    for (j=0; j<200; j++)
        a[i][j]=2;
```



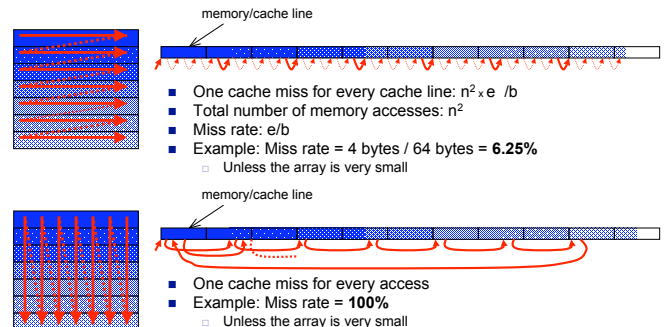
### Second option

```
int a[200][200];
for (j=0; j<200; j++)
    for (i=0; i<200; i++)
        a[i][j]=2;
```



## Counting cache misses

- $n \times n$  2-D array, element size =  $e$  bytes, cache line size =  $b$  bytes



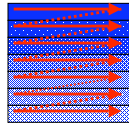
- One cache miss for every cache line:  $n^2 \times e / b$
- Total number of memory accesses:  $n^2$
- Miss rate:  $e/b$
- Example: Miss rate = 4 bytes / 64 bytes = **6.25%**
  - Unless the array is very small

- One cache miss for every access
- Example: Miss rate = **100%**
  - Unless the array is very small

## Array Initialization in C

### First option

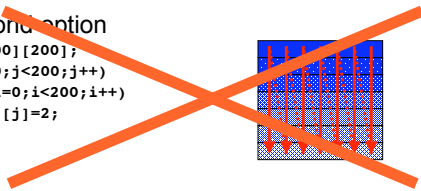
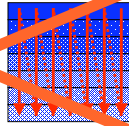
```
int a[200][200];
for (i=0; i<200; i++)
  for (j=0; j<200; j++)
    a[i][j]=2;
```



Good Locality

### Second option

```
int a[200][200];
for (j=0; j<200; j++)
  for (i=0; i<200; i++)
    a[i][j]=2;
```



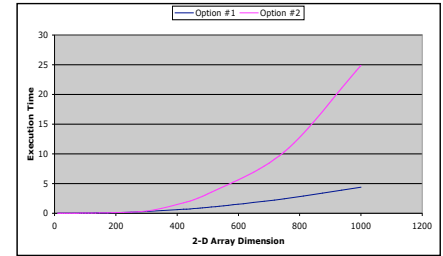
## Performance Measurements

### Option #1

```
int a[X][X];
for (i=0; i<200; i++)
  for (j=0; j<200; j++)
    a[i][j]=2;
```

### Option #2

```
int a[X][X];
for (j=0; j<200; j++)
  for (i=0; i<200; i++)
    a[i][j]=2;
```



Experiments on my laptop

- Note that other languages use column major
  - e.g., FORTRAN

## Let's try it..

- I wrote a simple program that initializes a two-dimensional array either long rows or along columns
- It comes with a Makefile that compiles to two version of the code with different compiler optimization flags
- It's on the Web site (locality\_example.zip)
- Let's run it on my laptop and on my server
  - And perhaps see some odd compiler things..

## Matrix Multiplication

- A classic example for locality-aware programming is matrix multiplication

```
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    for (k=0; k<N; k++)
      c[i][j] += a[i][k] * b[k][j];
```

- There are 6 possible orders for the three loops
  - i-j-k, i-k-j, j-i-k, j-k-i, k-i-j, k-j-i
- Each order corresponds to a different access patterns of the matrices
- Let's focus on the inner loop, as it is the one that's executed most often

## Inner Loop Memory Accesses

- Each matrix element can be accessed in three modes in the inner loop
  - Constant: doesn't depend on the inner loop's index
  - Sequential: contiguous addresses
  - Stride: non-contiguous addresses (N elements apart)

$c[i][j] += a[i][k] * b[k][j];$

- | Loop Order        | a[i][k]    | b[k][j]    |
|-------------------|------------|------------|
| i-j-k: Constant   | Sequential | Strided    |
| i-k-j: Sequential | Constant   | Sequential |
| j-i-k: Constant   | Sequential | Strided    |
| j-k-i: Strided    | Strided    | Constant   |
| k-i-j: Sequential | Constant   | Sequential |
| k-j-i: Strided    | Strided    | Constant   |

## Loop order and Performance

- Constant access is better than sequential access
  - it's always good to have constants in loops because they can be put in registers (as we've seen in our very first optimization)
- Sequential access is better than strided access
  - sequential access is better than strided because it utilizes the cache better
- Let's go back to the previous slides

## Best Loop Ordering?

$c[i][j]$	+=	$a[i][k]$	*	$b[k][j]$ ;
i-j-k: Constant		Sequential		Strided
i-k-j: Sequential		Constant		Sequential
j-i-k: Constant		Sequential		Strided
j-k-i: Strided		Strided		Constant
k-i-j: Sequential		Constant		Sequential
k-j-i: Strided		Strided		Constant

- k-i-j and i-k-j should have the *best* performance
- i-j-k and j-i-k should be *worse*
- j-k-i and k-j-i should be the *worst*

## Optimizing Further?

- There are many other things we could do to the code
  - “blocking”
  - loop unrolling
  - instruction reordering
  - ...
- There are many things the compiler can do to the code and there are many compiler flags we could use
- In the end, how do we determine the best implementation for a given architecture?

## Automatic Program Generation

- It is difficult to optimize code because
  - There are many possible options for tuning/modifying the code
  - These options interact in complex ways with the compiler and the hardware
- This is really an “optimization problem”
  - The objective function is the code’s performance
  - The feasible solutions are all possible ways to implement the software
  - Typically a finite number of implementation decisions are to be made
  - Each decision can take a range of values
    - e.g., the 7th loop in the 3rd function can be unrolled 1, 2, ..., 20 times
    - e.g., the “block size” could be 2x2, 4x4, ..., 400x400
    - e.g., function could be recursive or iterative
- And one needs to do it again and again for different platforms

## Automatic Program Generation

- What is good at solving hard optimization problems?
  - computers
- Therefore, a computer program could generate the computer program with the best performance
  - Could use a brute force approach: try all possible solutions
    - but there is an exponential number of them
  - Could use genetic algorithms
  - Could use some ad-hoc optimization technique

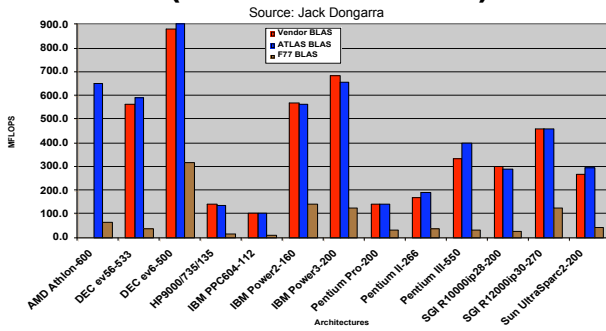
## Matrix Multiplication

- We have seen that for matrix multiplication there are several possible ways to optimize the code
  - block size
  - optimization flag to the compiler
  - order of loops
  - ...
- It is difficult to find the best one
- People have written automatic matrix multiplication program generators!

## The ATLAS Project

- ATLAS is a software that you can download and run on most platforms.
- It runs for a while (perhaps a couple of hours) and generates a .c file that implements matrix multiplication!
- ATLAS optimizes for
  - Instruction cache reuse
  - Floating point instruction ordering
    - pipeline functional units
  - Reducing loop overhead
  - Exposing parallelism
    - multiple functional units
  - Cache reuse

## ATLAS (500x500 matrices)



- ATLAS is faster than all other portable BLAS implementations and it is comparable with machine-specific libraries provided by the vendor.

## Improving an Application

- So, we have seen ways in which to improve pieces of code
- The problem is that one typically doesn't have an application that just performance an array initialization, or a matrix multiplication
- In fact, there are many parts of the application that one could think of optimizing for memory, etc.

## Speedup

- We need a metric to quantify the impact of your performance enhancement
- Speedup: ratio of "old" time to "new" time
  - old time = 2h
  - new time = 1h
  - speedup = 2h / 1h = 2
- Sometimes one talks about a "slowdown" in case the "enhancement" is not beneficial
  - Happens more often than one thinks

## Bad News: Amdahl's Law

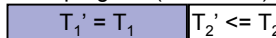
- Consider a program whose execution consists of two phases
- You have an idea to speed up phase #2

Old program (unenhanced)



Old time:  $T = T_1 + T_2$

New program (enhanced)



New time:  $T' = T_1' + T_2'$

$T_1$  = time that is NOT enhanced.

$T_2$  = time that can be enhanced.

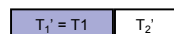
$T_2'$  = time after the enhancement.

## Back to Amdahl's Law

Old program (unenhanced)

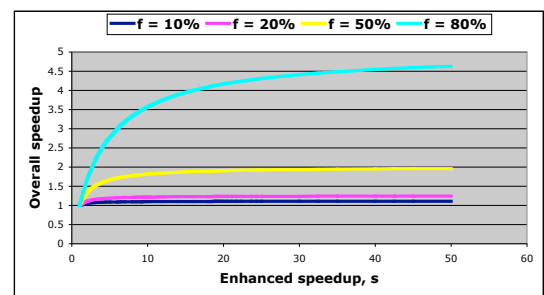


New program (enhanced)



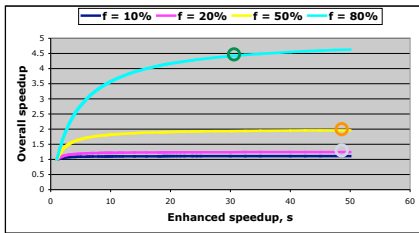
- $f$ : fraction of code enhanced =  $T_2 / T$
- $s$ : speedup of enhanced fraction:  $T_2 / T_2'$
- $T' = T_1 + T_2'$   
 $= T - T_2 + T_2 / s$   
 $= T - f * T + f * T / s$
- Speedup =  $T / T'$
- **Amdahl's Law:** Speedup =  $1 / (1 - f + f/s)$

## Amdahl's Law: Example



Plot of  $1 / (1 - f + f/s)$  for 4 values of  $f$  and for increasing values of  $s$

## Amdahl's Law: Example



- Speeding up 20% of the code by a factor 50 only results in an overall speedup of ~1.25
- Speeding up 50% of the code by a factor 50 only results in an overall speedup of 2
- Speeding up 80% of the code by a factor 30 only results in an overall speedup of 4.5
- Speeding up 10% of the code is useless!

## Lessons from Amdahl's Law

- It's a law of diminishing return
- One must first focus on optimizing part of a program that are responsible for the largest fraction of the execution time
  - Spending a lot of time on a portion of code that represents 10% of the execution time is a waste of time unless one is after a tiny improvement
- It sounds obvious, but people new to high performance computing often forget how bad Amdahl's law is
  - People lose track of how bad the graph on the previous slide is

## Now What?

- We have defined **measures of performance**
- We know that the portion is slower than expected (e.g., slower than the peak) because of a **bottleneck**
- We know we should focus on a portion of the code that is a large fraction of the overall execution time (**Amdahl's law**)
- Question: how do we know which part of the code is the most "expensive"?
  - If you've not written the code you may not know
  - If you've written the code you may have some idea (although experience shows that many programmers don't)
  - The most expensive part may be in some library function you haven't written
- You could put `gettimeofday()` calls everywhere, but that gets really cumbersome for large projects
- The standard way: use a **profiler**

## What is a Profiler?

- A profiler is a tool that monitors the execution of a program and that reports the amount of time spent in different functions
- Useful to identify the expensive functions
- Profiling cycle
  - Compile the code with the profiler
  - Run the code
  - Identify the most expensive function
  - Optimize that function
    - call it less often if possible
    - make it faster
  - Repeat until you can't think of any ways to further optimize the most expensive function
- UNIX has a good, free profiler called **gprof**

## Using gprof

- Compile your code using `gcc` with the **-pg** option
- Run your code until completion
- Then run `gprof` with your program's name as single command-line argument
- Example

```
% gcc -pg prog.c -o prog
% ./prog
% gprof prog > profile_file
```
- The output file contains all profiling information

## Profiling output

- The content of the file is explained in detail in the file itself
- At the beginning of the file is a summary of which fraction of the code is spent in which function
- In the middle section is a detailed entry for each function
- At the end of the file is a "function index", in which each function is assigned a number in brackets, e.g., [3]

## Profiling Output

- “Flat” profiling summary

in the function itself    in the function and its children

% time	cumulative seconds	self seconds	name
30.9	0.77	0.77	__multadd_D2A [1]
16.9	1.19	0.42	_scheduler <cycle 1> [3]
15.3	1.57	0.38	_scandir [5]
9.2	1.80	0.23	_NSLookupAndBindSymbolHint [6]
6.4	1.96	0.16	_job <cycle 1> [8]
4.4	2.07	0.11	_NSIsSymbolNameDefinedHint [9]
1.6	2.11	0.04	_hash_nkey [10]
1.6	2.15	0.04	_pthread_key_create [11]
1.2	2.18	0.03	_quorem_D2A [12]
1.2	2.21	0.03	_mh_dylib_header [13]
1.2	2.24	0.03	_probe_submitter [14]
1.2	2.27	0.03	_request_submitter [15]

## Profiling output

- The middle section of the file provides detailed information for each function
- Entry format:

index	% time	self	children	called	name
		1.21	3.10	80/132	f1 [111]
		0.69	1.13	52/132	f2 [123]
[1]	23.1	2.12	4.23	132	func [1]
		4.23	0.00	32/5231	c [39]

- Can vary depending on the version of gprof
  - You should really read the explanations in the file to be sure

## Profiling output

index	% time	self	children	called	name
		1.21	3.10	80/132	f1 [111]
		0.69	1.13	52/132	f2 [123]
[1]	23.1	2.12	4.23	132	func [1]
Function func [1]		4.23	0.00	32/5231	c [39]

## Profiling output

index	% time	self	children	called	name
		1.21	3.10	80/132	f1 [111]
		0.69	1.13	52/132	f2 [123]
[1]	23.1	2.12	4.23	132	func [1]
Function func [1]		4.23	0.00	32/5231	c [39]

Parents: f1 [111], f2 [123]

## Profiling output

index	% time	self	children	called	name
		1.21	3.10	80/132	f1 [111]
		0.69	1.13	52/132	f2 [123]
[1]	23.1	2.12	4.23	132	func [1]
Function func [1]		4.23	0.00	32/5231	c [39]

Parents: f1 [111], f2 [123]

Children: c [39]

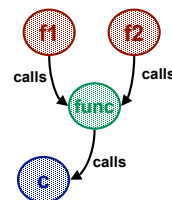
## Profiling output

index	% time	self	children	called	name
		1.21	3.10	80/132	f1 [111]
		0.69	1.13	52/132	f2 [123]
[1]	23.1	2.12	4.23	132	func [1]
Function func [1]		4.23	0.00	32/5231	c [39]

Parents: f1 [111], f2 [123]

Children: c [39]

### Call graph

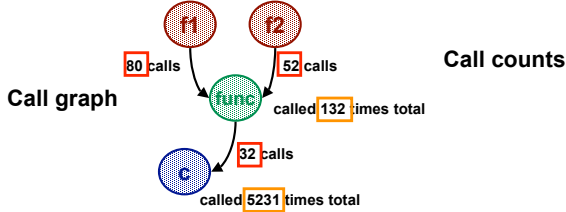


## Profiling output

Parents: f1 [111], f2 [123]

index % time	self	children	called	name
	1.21	3.10	80/132	f1 [111]
	0.69	1.13	52/132	f2 [123]
[1]	23.1	2.12	132	func [1]
Function func [1]	4.23	0.00	32/5231	c [39]

Children: c [39]

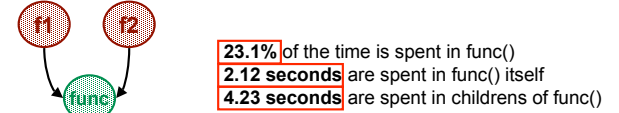


## Profiling output

Parents: f1 [111], f2 [123]

index % time	self	children	called	name
	1.21	3.10	80/132	f1 [111]
	0.69	1.13	52/132	f2 [123]
[1]	23.1	2.12	132	func [1]
Function func [1]	4.23	0.00	32/5231	c [39]

Children: c [39]

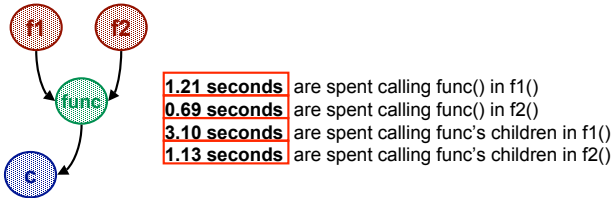


## Profiling output

Parents: f1 [111], f2 [123]

index % time	self	children	called	name
	1.21	3.10	80/132	f1 [111]
	0.69	1.13	52/132	f2 [123]
[1]	23.1	2.12	132	func [1]
Function func [1]	4.23	0.00	32/5231	c [39]

Children: c [39]



## Using gprof

- Get the gprof output
- Understand the output
- Identify the function that has the highest self time
  - make it faster by removing bottlenecks an optimization blockers
  - or call it less often (typical example: malloc)
- Repeat until there is no improvement
- Go on to the next function

## Conclusion

- When performance becomes an issue the first thing to do is to use a profiler
  - There are many Java profilers as well
- Then one must hope that a few functions are responsible for most of the execution time
  - A terrible situation is to have 100 functions that each account for 1% of the execution time (luckily it's rare)
- One can then invest time improving the costly functions and see where that gets us