# Project 2B

CS111

05/10/19

# General Comments on Multithreading

- You noticed that time per operation increases with the number of threads, more so for lists.
  - Does this mean that multithreading is inferior ?
  - Why should we bother with multithreading in that case?

# General Comments on Multithreading

- You noticed that time per operation increases with the number of threads, more so for lists.
  - Does this mean that multithreading is inferior ?
  - Why should we bother with multithreading in that case?
- Maybe time per operation is not a great metric
  - It doesn’t take parallelism into account
  - What would be better?

# General Comments on Multithreading

- You noticed that time per operation increases with the number of threads, more so for lists.
  - Does this mean that multithreading is inferior ?
  - Why should we bother with multithreading at all in that case?

- Maybe time per operation is not a great metric
  - It doesn't take parallelism into account
  - What would be better?

- Throughput : the number of operations per second
  - This allows to take parallelism into account
  - This makes more sense when multithreading is involved

# On Contention

- Locks on the shared variable force serialization
  - All threads can run in parallel until they reach the part of code where they acquire the lock
  - At that point they have to wait in line : this is contention
- Why is contention worse for list operations ?

# On Contention

- Locks on the shared variable force serialization
  - All threads can run in parallel until they reach the part of code where they acquire the lock
  - At that point they have to wait in line : this is contention
- Why is contention worse for list operations ?
  - Inserting in a list requires more work than addition
  - The **proportion** of time spent in the critical section is higher for list operations
- How could we decrease that contention for lists?

# On Contention

- Locks on the shared variable force serialization
  - All threads can run in parallel until they reach the part of code where they acquire the lock
  - At that point they have to wait in line : this is contention
- Why is contention worse for list operations ?
  - Inserting in a list requires more work than addition
  - The **proportion** of time spent in the critical section is higher for list operations
- How could we decrease that contention for lists?
  - We could create several ‘queues’ to decrease serialization
  - This would be done by splitting up the list in sublists
  - We would lock each sublist independently

# Synchronization overhead

- You used 2 types of locks to protect critical sections
  - Spin-locks
  - Mutexes
- Both prevented race conditions
  - Do they add any overhead?
  - Is one better than the other?
- What is the overhead for spin-locks?

# Synchronization overhead

- You used 2 types of locks to protect critical sections
  - Spin-locks
  - Mutexes
- Both prevented race conditions
  - Do they add any overhead?
  - Is one better than the other?
- What is the overhead for spin-locks?
  - CPU cycles are used as the thread tries to acquire the lock
  - How can we measure that overhead?

# Synchronization overhead

- You used 2 types of locks to protect critical sections
  - Spin-locks
  - Mutexes
- Both prevented race conditions
  - Do they add any overhead?
  - Is one better than the other?
- What is the overhead for spin-locks?
  - CPU cycles are used as the thread tries to acquire the lock
  - How can we measure that overhead?
- What is the overhead for mutexes?
  - Context switches when the thread can’t acquire the lock
  - How can we measure that overhead?

# Some additions in this project

- lab2_list.c -> --lists option
  - Accept ‘number of lists’ as a parameter
  - Initialize several shared lists, each locked independently
  - Decreases contention !
- Execution profiling report. Evaluate:
  - How much time was spent acquiring mutexes?
  - How many cycles were spent acquiring spin-locks?
  - Add an 8th column of data to your .csv
- New graphs
  - Using throughput as a metric
- gperftools -> multi-threaded application friendly
  - Heap checker
  - Heap profiler
  - Cpu-profiler -> what we’ll use to measure spin-lock overhead

# How does the cpu profiler work?

- It takes a snapshot of the CPU every millisecond
  - The interval time between snapshots is user controllable
- At each snapshot, it records what function is running
  - This is accumulated over the process runtime
- After the process has run, we can look at the log:
  - If 100 snapshots were taken in function X, we can assume that function ran for 100 ms (which is an approximation ... )
  - In the case of spin locks, what function do we want to monitor?

# How does the cpu profiler work?

- It takes a snapshot of the CPU every millisecond
  - The interval time between snapshots is user controllable
- At each snapshot, it records what function is running
  - This is accumulated over the process runtime
- After the process has run, we can look at the log:
  - If 100 snapshots were taken in function X, we can assume that function ran for 100 ms (which is an approximation ... )
  - In the case of spin locks, what function do we want to monitor?
    - Test_and_Set()
- The profiler will tell us how many CPU cycles we used to acquire locks
  - This is a good evaluation metric of the overhead

# gperftools - Install

- Releases found : https://github.com/gperftools/gperftools/releases
- Unpack and cd to directory
- ./configure --prefix=<path>
- make
- make install
- (make clean)

Rmk: You do not need root permission to do this

Rmk: You also need to install gv to access graphical output, this is optional

# gperftools - Use

The pprof command will allow you to analyse the profiling result.

To turn on CPU profiling, you have two options:

1. Define environment variable CPUPROFILE to the filename to dump the profile to (in your makefile)

   Rmk: delete that file before each run

2. Bracket the code you want profiled using ProfilerStart() and ProfilerStop(). ProfilerStart() takes the profile filename as argument

# Updating your makefile : profile option

1. Link the library: path_to_lib = absolute path where you installed

LD_PRELOAD = <path_to_lib> (should be in /usr/lib(64))

2. Link file to record dump

CPUPROFILE = ./raw.gperf

3. pprof with --text

pprof --text <executable> <dump file> profile.out

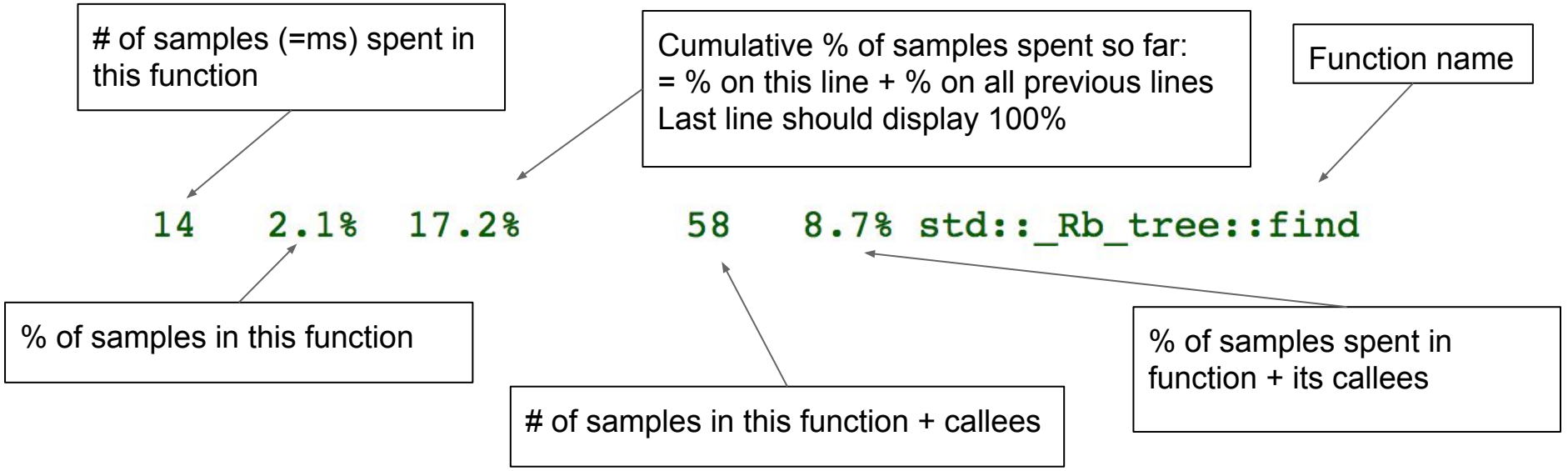
4. pprof with --list

pprof --list=<function> <exec> <dump> >> profile.out

Rmk: You also have to compile with the -lprofile flag

# Analyzing the output

- For more information
- You **need** to use the text output, but if you first want to use the graphical output to make things clearer you can
- Text output will produce lines that look like:



# Gperftools and mutexes

- How can we use gperftools to measure the overhead incurred by mutexes?

# Gperftools and mutexes

- How can we use gperftools to measure the overhead incurred by mutexes?
  - We can't!
  - When a mutex can't be acquired, the thread goes to sleep
  - The overhead is a context switch, and time spent sleeping
  - This can't be measured by counting CPU cycles

- What can we do instead?

# Gperftools and mutexes

- How can we use gperftools to measure the overhead incurred by mutexes?
  - We can't!
  - When a mutex can't be acquired, the thread goes to sleep
  - The overhead is a context switch, and time spent sleeping
  - This can't be measured by counting CPU cycles

- What can we do instead?
  - We can directly time how long it takes for us to acquire a mutex

# Timing Mutex Waits

- Computing wait-for-lock time:
  - Measure time before and after getting the lock
  - Add up all wait time for all threads
  - Divide by number of operations
  - Output Statistics for the run

- If you are not locking -> should get 0
- For example:
  - Allocate an array of timers (# threads)
  - clock_gettime, pthread_mutex_lock([sublist]), clock_gettime
  - add time to timer[threadnum]

# Addressing the Underlying Problem

- These steps will allow you to prove the original intuition: degradation is a result of increased contention

- To decrease contention, we can split the list
  - Add a lists option to specify the number of chunks
  - Select which list to insert node in
  - Get list length
  - Delete inserted nodes
  - exit

# Inserting into different sublists

- These is no requirement of total ordering
  - sublist 1 must be ordered, sublist 2 must be ordered
  - But there is no order requirement **between** sublists
- You are still generating the same number of keys
  - They will be assigned to a single sublist

- To select which list : use a hash function
  - It doesn't matter what hash function you use
  - The goal is to map threads*iterations -> #lists

# Inserting into different sublists

- These is no requirement of total ordering
  - sublist 1 must be ordered, sublist 2 must be ordered
  - But there is no order requirement **between** sublists
- You are still generating the same number of keys
  - They will be assigned to a single sublist

- To select which list : use a hash function
  - It doesn’t matter what hash function you use
  - The goal is to map threads*iterations -> #lists
  - Could be as simple as a modulo operation (key_value % lists)

# Two sources of efficiency

- Splitting lists increases efficiency by decreasing contention: serialization isn't as bad
- What is the second effect?

# Two sources of efficiency

- Splitting lists increases efficiency by decreasing contention: serialization isn't as bad
- What is the second effect?
  - The sublists will be shorter!
  - Therefore insertion operations will be faster
  - The **proportion** of time spent in the critical section will decrease
  - All threads will, on average, wait a shorter amount of time to acquire a lock
  - Fewer cycles/time wasted to acquire a spin-lock/mutex

- These cycles/time can be spent doing useful operations in the critical section