GETTING SPEED IN APPLICATIONS WITH LOTS OF MOVING PARTS

Professor Ken Birman

CS4414 Lecture 3

IDEA MAP FOR TODAY

Our word count scenario had a lot of stuff happening

➢ Reading files from the Linux-managed file system

➢ Scanning them line by line, counting word occurrences

➢ Sorting and printing the output

Revisit the example from lecture 1. C++ was faster because it allowed Ken to leverage parallelism using threads.

Parallelism is a powerful tool, but only gives a speedup if the program itself is parallelizable. Sequential bottlenecks limit achievable speed

There are many "hidden" opportunities for parallelism that can benefit even a sequential program. A good example is prefetching in a cache

WHAT WE WILL (AND WON'T) COVER

This isn't a programing course, or an intro to operating systems, so I'm going to assume you do basically understand how to code things, and data structures (hash-maps, trees…), file systems…

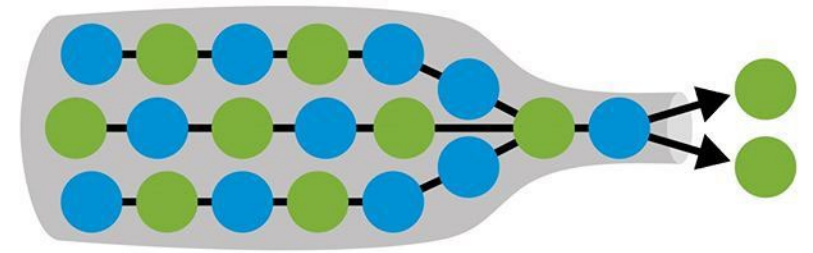But I am interested in situations tasks with "lots of moving parts" and multiple options to pick from. What really shapes performance?

A PHILOSOPHY OF PERFORMANCE

Make your code sleek and elegant and "standard"

Then experiment to _find the bottlenecks_

Fix them (even if your changes are less elegant). But don't mess with the rest of your program. If a change wouldn't help in the larger picture, why destroy that really beautiful code?

GETTING THE LAY OF THE LAND…

Linux kernel has about 26M lines of code in 50,000 files.

Start by *asking how fast this can possibly be done*

➢ Small experiment: **time cat {bunch of files} > /dev/null**
   Slightly fancier: in N parallel tasks, read 50,000/N of them

➢ Fastest with 10-15 tasks. Linux needs 4.25s to read all the files!

➢ With N=1 will be sequential. Less "sys time" but takes 2m19s!

INSIGHT

We won't (can't) outperform 4.5s on this machine.

New goal: Code should "keep up" with the incoming data.

Count in parallel as Linux file I/O loads files from disk

SPLITTING LINES INTO "WORDS"

Define a word to be a sequence of A-Z, a-z, 0-9, _

Examples of words:

➤ for, which, if, else, define, int, float, …

➤ n, inode_table, fd

➤ 0x61AFF00

PART OF A SOLUTION IN "PURE LINUX"

1) Find all source files (they end in .c or .h)

2) Print ("cat") each file

3) Line by line, map non-word-chars to blank, then map all blanks to newline (\012). Long list of words… and blank lines

4) sort, count unique, sort again, print output

HOW DID THE PROGRAMS WORK?

This version is easy to write but looks horrible:

```
find . -type f \( -name '*.c' -o –name '*.h'\) -exec cat {} \; |
 tr -c '[A-Za-z0-9_ \012]' ' ' | tr -s '[ ]' '\012' | sort | uniq –c | sort –r –n
```

HOW DID THE PROGRAMS WORK?

This version is easy to write but looks horrible:

```
find . -type f \( -name '*.c' -o –name '*.h'\) -exec cat {} \;  |
 tr -c '[A-Za-z0-9_ \012]' ' '  | tr -s '[ ]' '\012'  | sort  | uniq –c  | sort –r –n
```

It uses what Linux calls a "pipe". A process prints output to stdout (normally, the console) but we "redirect" it to become stdin (input) to another process. This uses 5 pipe operations: |

VISUALIZING THE PURE LINUX SOLUTION

| find . -type f \(-name '*.c' -o –name '*.h'\) -exec cat {} \; | | tr -c '[A-Za-z0-9_ \012]' ' ' | | tr -s '[ ]' '\012' | | sort | | uniq –c | sort –r -n |

```
…
mm_segment_t fs = get_fs();
set_fs(KERNEL_DS);

fd = (*syscall_open)(file, flags, mode);
if(fd != -1) {
    (*syscall_read)(fd, buf, size);
    (*syscall_close)(fd);
}
set_fs(fs);
…
```

```
…
fd
syscall_open
        file
flags
        mode
Fd
1
syscall_read
        fd
buf
  size
…
```

```
…
fd
syscall_open
File

flags
mode
fd

1
syscall_read
fd
buf

size
…
```

```
…
1
buf
fd
fd
fd
file
flags
mode
size
syscall_open
syscall_read
…
```

```
…
1 1
1 buf
3 fd
1 file
1 flags
1 mode
1 size
1 syscall_open
1 syscall_read
…
```

LINUX COMMANDS ARE WAY TOO SLOW

It involves a chain of 6 processes linked by pipes.

Even so, quite slow.

#4: Pure Linux (buggy sort order)
     real   2m38.965s
     user   2m43.999s
     sys   27.084s

… and didn't even have the output ordering we desire! Sort lacks a "descending, but ascending by name for ties" option

WRITING A **PROGRAM** TO DO THIS

Same idea, but with a program, we "take control"

Now we can code for speed... to keep up with Linux file system

A program can fix the issue of wanting our output to be sorted by (count,word) with descending count, but alphabetic for ties

VISUALIZING THIS APPLICATION

```
…
mm_segment_t fs = get_fs();
set_fs(KERNEL_DS);

fd = (*syscall_open)(file, flags, mode);
if(fd != -1) {
    (*syscall_read)(fd, buf, size);
    (*syscall_close)(fd);
}
set_fs(fs);
…
```

```
…
fd
syscall_open
file
flags
mode
fd
1
syscall_read
fd
buf
size
…
```
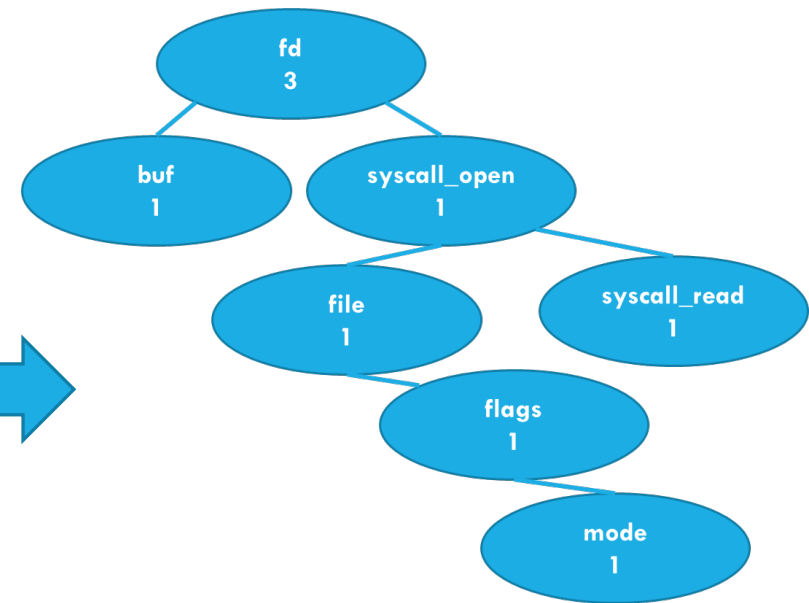

Sorted by name

Phase one: Count words in the file *using a tree sorted by name*

WHY A TREE? WHAT ABOUT A HASH TABLE?
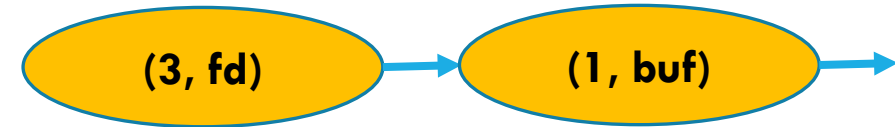
A tree is an O(nLogn) structure. Hash table is O(1).

But for a hash table we need to estimate the approximate size required. With 24M words… maybe 10-20 threads… hash tables (one per thread) could be 2-4 GB of memory. Each.

Trees are more compact, so Ken picked std::map, a tree

SORTING FOR DESIRED OUTPUT ORDER
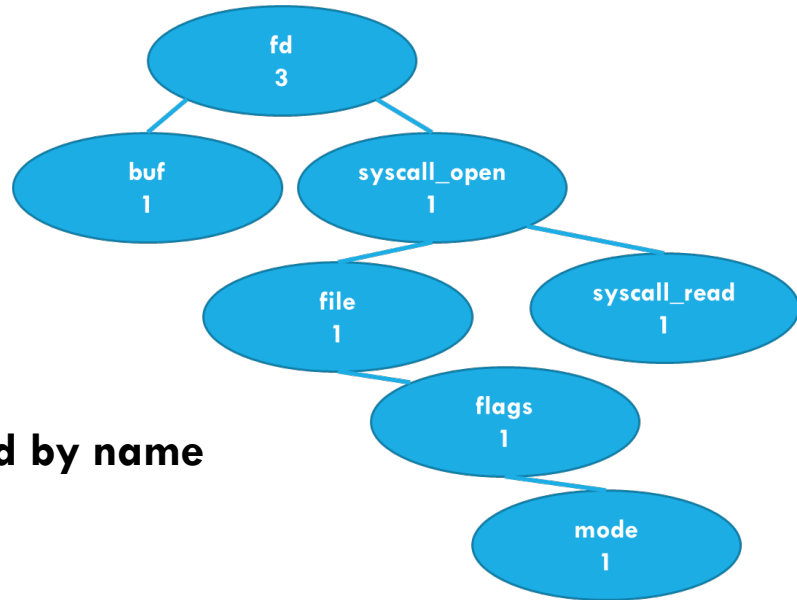
Sorted by name

Re-sorted by (count, name)

Output

Word	Count
fd	3
buf	1

Phase two: Sort by (count,word), then print output

PYTHON, JAVA AND C++ ALL TURN OUT TO HAVE PREBUILT TOOLS FOR EACH STEP

Every one of these steps can just use a standard library.

We end up with very elegant, concise code.

It looks pretty similar for all three languages

LET'S START WITH PYTHON

Python has a built-in file scanner like find, string splitter, built in vectors, and a vector sort. It doesn't leverage hardware parallelism.

One of our course staff members (Lucy) coded this up…

#3 Lucy's Python version
       real   1m30.857s
       user   1m30.276s
       sys   0.572s

WHAT ABOUT JAVA VERSUS C++?

Lucy also created a Java version. It compiles in two stages:

➤ First to Java byte code

➤ Then to machine code (JIT)

> **#2 Lucy's Java version (no threads)**
> real   1m49.373s
> user   3m16.950s
> sys   8.742s

Both compilation steps are highly efficient, but there are some situations in which Java can only know the type of an object at *runtime*. This "runtime polymorphism" slows some libraries down.

C++ VERSION?

We created two C++ versions.

Sagar's was pure and quite fast; we posted the code on the "extra materials" on our schedule web page.

Ken's dropped into C for file I/O steps and went further than Sagar in leveraging parallelism. This was fastest of all.

QUALITY OF MACHINE CODE

Whether we use Python or Java or C++, at the end of the day the computer executes machine code. We saw some last week.

Python itself is implemented in Java or C++ and compiled.

But Python *interprets* your code. This causes slowdown.

RUNTIME TYPES VERSUS STATIC TYPES

With Java "interesting" things (like tree nodes, or strings) are *objects.*

Java object types are learned at runtime… this is called "reflection". Reflection has a cost, paid at runtime – programs run slower. There are ways to speed reflection up, but overheads remain an issue.

C++ types are always fully known at compile time (statically). This lets the compiler use type information to do code optimization.

THREADS: A BIG TOPIC FOR CS4414
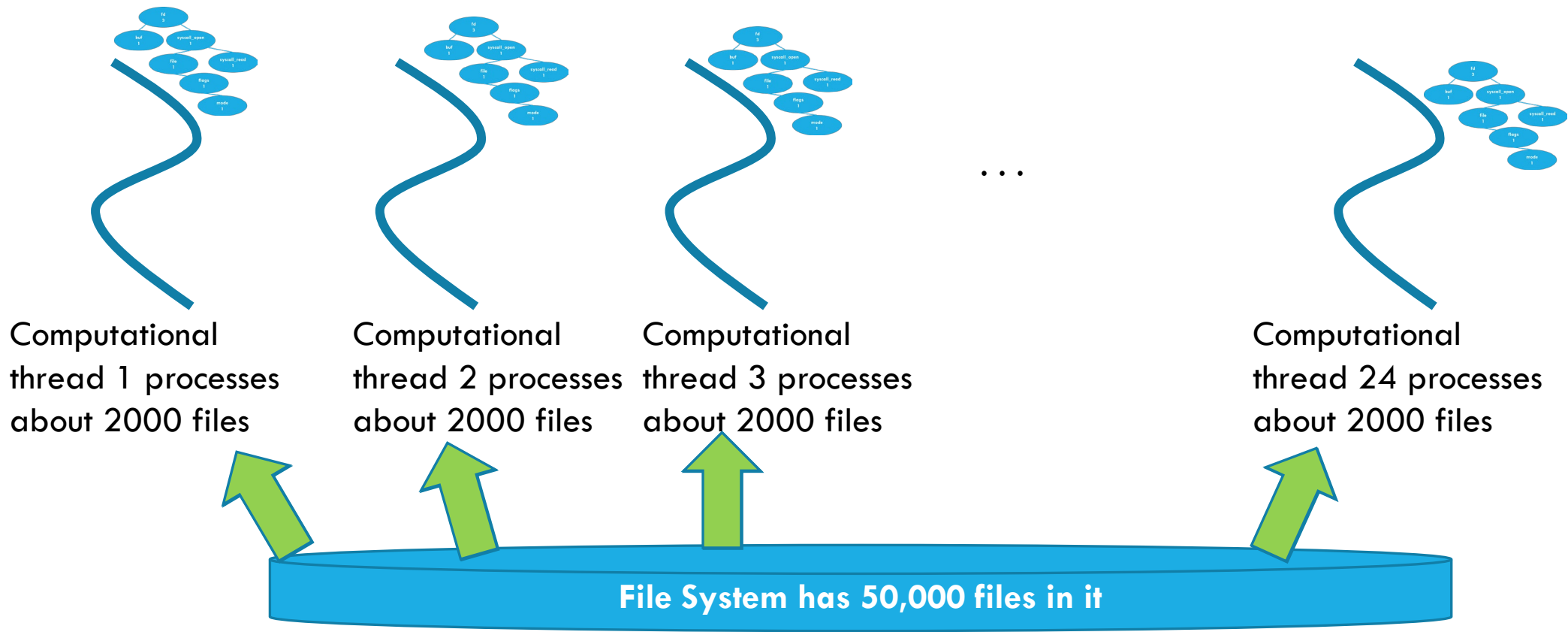
Think about a method that has no return value:

do_something(args….);

A **thread** runs some method in parallel with its parent.

"You set the table… I'll get some chips and salsa"

"You scan files 1…1000" … "I'll scan 1001…2000"

VISUALIZING TASK-LEVEL PARALLELISM

Computational thread 1 processes about 2000 files

Computational thread 2 processes about 2000 files

Computational thread 3 processes about 2000 files

. . .

Computational thread 24 processes about 2000 files

File System has 50,000 files in it

UNDERSTANDING THE TIMER OUTPUT

In this example, my program will run silently on 8 cores using 16 threads

The "real" (wall clock) time
was 18.469 seconds.

This is how long we waited for it to finish

```
% time taskset 0xFF ./fast-wc -n16  -s

real    0m18.469s
user    0m43.406s
sys     0m18.203s
```

UNDERSTANDING THE TIMER OUTPUT

The "user" time measures compute in my 16 threads. It can be as much as 16*real time!

```
% time taskset 0xFF ./fast-wc -n16  -s

real    0m18.469s
user    0m43.406s
sys     0m18.203s
```

WHY DID I SHOW EACH THREAD WITH ITS OWN WORD-COUNT TREE?

A tree node needs to live somewhere in physical memory.

If each core builds its own word-count tree for files it scans, that tree will be entirely in its local memory, and not shared.

We will see that <u>sharing</u> objects involves adding locking that brings costs. Objects used by a single thread don't need locking.

DOWN SIDE OF HAVING 24 TREES…

… we end up with 24x more memory in use!

Linux had 4M "unique" words, but with 24 threads, only 27,000 words are seen by half or more! 3.2M are seen by just 1 or 2 threads.

Suppose an average word-count node requires 64 bytes, and that we end up with 250,000 nodes per thread. With 24 times will require 16MB x 24 = ~400MB. We have enough memory!

DOWNSIDE OF HAVING 24 TREES…

At the end, we have 24 trees containing sub-counts.

Before sorting, we'll need to merge them.
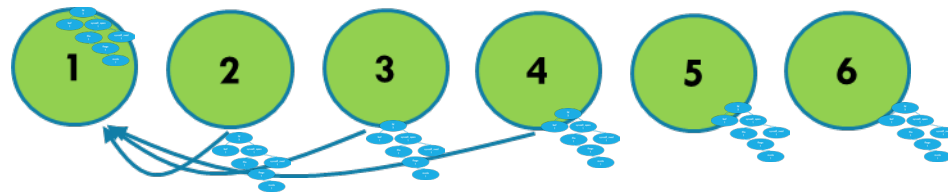
WITH SORTED TREES, TREE MERGE IS "EASY"

For each node in tree B, look up that word in tree A, sum the counts.

C++ is like Java or Python: it has "iterators" for data structures

Just a tiny **for** loop. But who should run it?

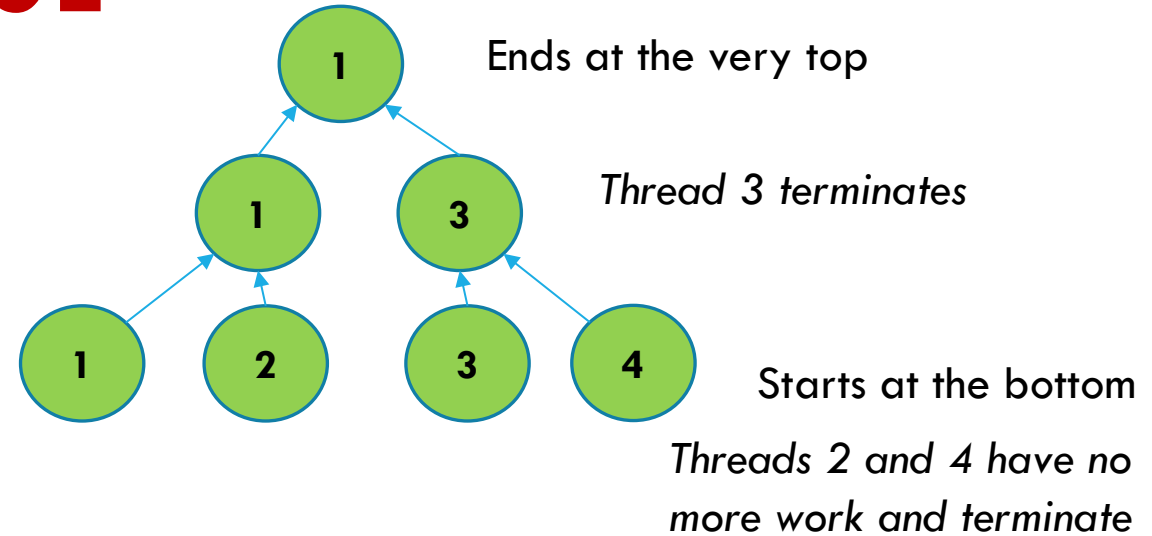# EACH THREAD COMPUTES A PARTIAL WORD COUNT ON A PORTION OF OUR DATA

Visualization: Thread 1 runs the merge step.

… this is linear: 23 merge operations. Can we do better?

PARALLEL BINARY MERGE

In this picture, we merge from the bottom to the top

Ends at the very top

Thread 3 terminates

Starts at the bottom

Threads 2 and 4 have no more work and terminate

For 24 threads:

Merge   [1,2] and [3,4] and … [23,24]

Then   [1,3] ……………..   [23, 24]

        …

Finally: [1, 13]

PARALLEL BINARY MERGE

Ends at the very top

1

In th...
the b...

For ...

Mer...

Ther...

Finally: [1, 13]

...ates

...t the bottom

...d 4 have no
...nd terminate

Rule: Each thread t has a variable k and initializes k = t (its own thread-id in [1…n]). Initialize s to 1.

do {

If (k is even) { thread t terminates. }

else { merge tree t + s into tree t; }

k >>= 1; s <<= 1;

} until (all trees merged into tree 1);

WORTH IMPLEMENTING?

```
% time taskset 0xFF ./fast-wc -n16  -s

real    0m18.469s
user    0m43.406s
sys     0m18.203s
```

Time spent in Linux: File I/O

My program offers parallel merge (-p). It helps… a tiny bit.

Issue: my C++ version was really bottlenecked by file I/O. No matter how fast the threads run, the "sys 18.203s" remains!

C++ tricks can't reduce runtime below 18.203s without some way of improving the efficiency of parallel file I/O!

REMINDER: TREE OR HASH LIST?

Now we can answer the question from slides 12/13/14

In fact replacing our tree with a hash list probably wouldn't help: the file I/O bottleneck wouldn't be impacted.

The real-time speed limit is ultimately the need to read all the files (unless Linux already had them in the file system cache)

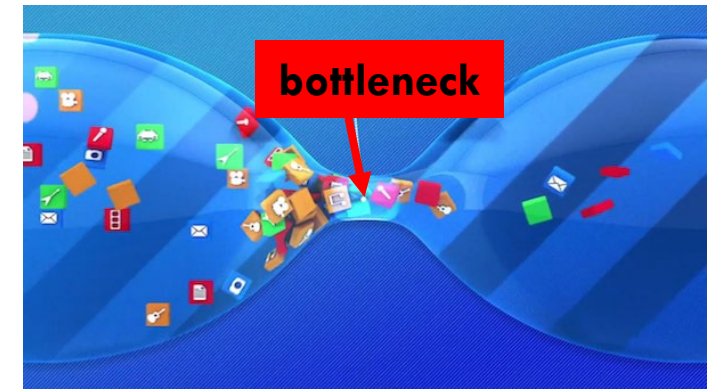# NEW CHALLENGE: KEEP EVERYTHING RUNNING SIMULTANEOUSLY!
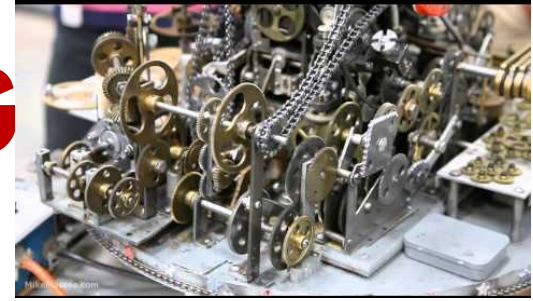
Finding the bottleneck can be difficult

Even our little merge program has many moving parts

➢ All those threads, building trees

➢ But also the work Linux is doing when the threads open files and read them.

Which is the limiting stage of our complete "system" (fast-wc + Linux)?

bottleneck

TERMINOLOGY

A *bottleneck*: "the limiting factor" for some task… we don't really use the term for a "balanced" task that has no limiting spot.

Compute-bound: The task is bottlenecked (limited) by the speed of calculations on some kind of in-memory data.

I/O bound: The task is bottlenecked on fetching data from some kind of storage device, or over the network.

OUR CHALLENGE: NOT JUST DATA STRUCTURES AND PARALLELISM, BUT BOTTLENECKS

How can we identify the bottlenecks that limit performance?

Can we even measure the degree of parallelism we are achieving?

➢ In fact Linux has tools we can use for that.

➢ We'll be learning about them!

AMDAHL'S LAW

Gene on the family farm in Norway

Gene Amdahl was a leading research on parallelism and supercomputing in IBM's HPC division.

He became interested in a basic question. *How fast can computations be performed, with infinite parallelism?*

A DAY TRIP TO NIAGARA FALLS

You and your friends want to check out Niagara falls.

There are six of you. One option is to rent a plus-sized car (but those big vehicles are slow)

A DAY TRIP TO NIAGARA FALLS

Better plan: You rent three convertible sports cars.

Each holds two people, and these are "insanely fast".

But as you head north, the narrow road has a **_bottleneck!_** Until you <u>all</u> pass this slow tractor, the group will have to wait.

Gene Amdahl's Tractor in Norway

HOW AMDAHL THOUGHT ABOUT PARALLELISM

In any computation, we have some parts that are highly parallel, such as scanning our 74,000 different files. Parallelism can speed those up.

But the computation will also have sequential tasks, which could include sequential logic buried in the operating system or the hardware.
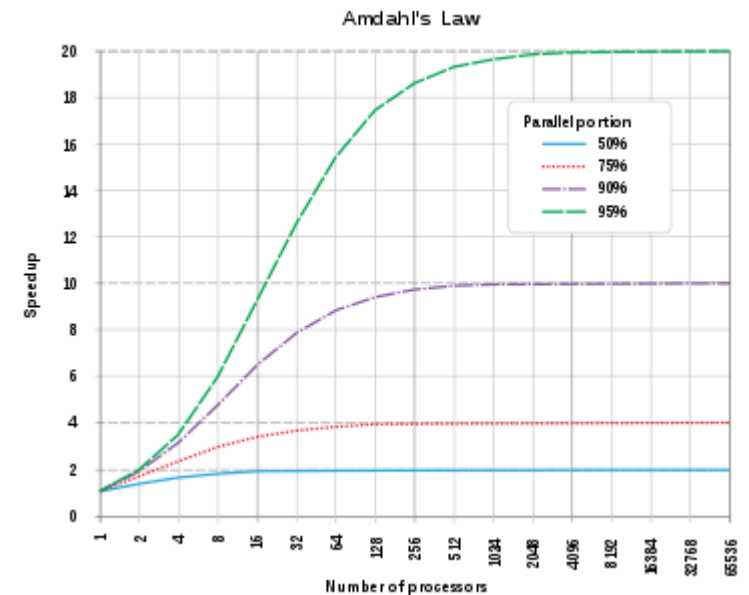
The sequential work will limit the speedup due to parallelism!

HOW AMDAHL EXPRESSED HIS LAW

Suppose that *p* represents the percentage of the task that can be parallelized.

Then *1/(1-p)* is the maximum possible speedup

Insight: The parallel tasks could all be done simultaneously, yet we would still have to do the sequential parts step by step.

WHAT WOULD BE <u>SEQUENTIAL</u> IN OUR WORD-FREQUENCY APPLICATION?

Each distinct word-count tree is managed by code that does "find or insert" and "increase the count" operations.

Those individual operations will be sequential.

WHAT ELSE WOULD BE <u>SEQUENTIAL</u> IN OUR WORD COUNT APPLICATION?

Once we have our single tree, we have to re-sort it, because we wanted our printed output to have common words at the top.

Ken and Sagar both needed a second sorted tree for this.

In fact, counting and the final sort both have identical cost!

THE FILE SYSTEM ENDS UP VERY BUSY!

These threads are opening and reading a *lot* of files

Can the file system keep up?

If not, our threads won't be active…

THE FILE SYSTEM ENDS UP VERY BUSY!

This is a famous issue with Linux. For example, Google and Facebook have Linux servers holding huge collections of web pages or photos.

They ended up putting images into "strips" to reduce the load on the file system.

FILE ACCESS COSTS: TWO ASPECTS

Each file has to be opened, which is a moment when Linux checks that the user has permission to access the file.

Once the file is open, it takes time to read and process data.

When reading, the fast-wc application does many "system calls"

FILE ACCESS COSTS: TWO ASPECTS

With 24 threads doing concurrent reads, the file system is doing a lot of data fetches from the disk (in "blocks" of 4096 bytes)

If those reads become a bottleneck, our threads will pause and we lose parallelism.

HOW THE LINUX FILE SYSTEM IS STRUCTURED

User level programs can't access files "directly".   They use Linux.

The implementation is modular with multiple layers, but notice the various caches: inodes, buffers and directories.
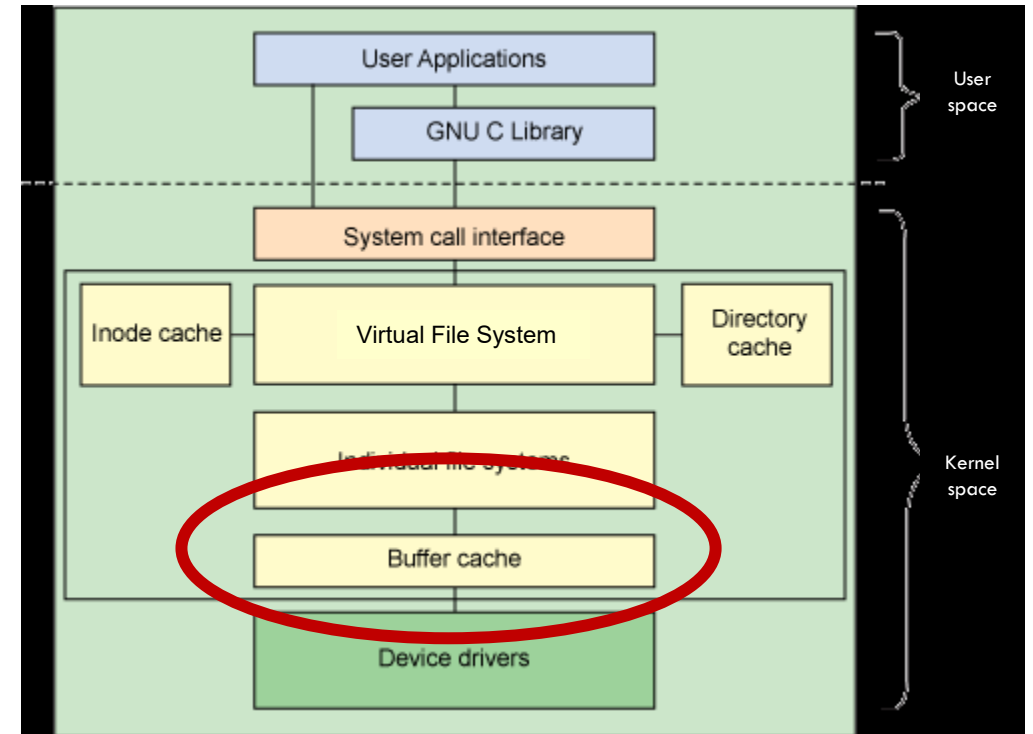
CACHE: A CONCEPT USED THROUGHOUT COMPUTING

A pool of memory holding copies of data that "lives" elsewhere.

➤ The Linux buffer pool is a cache of data read from files. Each time data is read, Linux keeps a copy (for a while). If it fills up, Linux will "evict" something else to make room.

➤ If the application re-reads that same data Linux can avoid the need to fetch it from the storage device again.

PREFETCHING INTO A CACHE

Linux also watches for sequential read patterns: you read the first 4096 bytes from a file, then the next 4096, then the next…

Linux will bet that you plan to continue doing this and issues one or two reads ahead of time, saving the data in cache.

Question: In what way is this a form of "parallelism"?

PREFETCHING INTO A CACHE

Linux also watches for sequential read patterns: you read the first

Why is prefetching a form of parallelism?

Answer: It lets us overlap the work of finding and reading the next block of the file (the next 4096 bytes) with the word-counting logic for the current block.

Linux
or tw

Question: In what way is this a form of "parallelism"?

WHY PREFETCHING AND CACHING HELP

Modern disks (SSD and rotating disks!) have large delays compared to memory access. 0.1ms or more delay.

Without a high rate of cache hits, we would spend 1 second (or longer) waiting for disk read requests to complete

With prefetching into the Linux buffer pool, we don't experience those 0.1ms delays. Our threads keep running

IN FACT THE CPU ITSELF USES CACHES AND PREFETCHING, TOO!

A modern CPU has multiple caches:

➤ L0: the registers. C++ might "cache" data in them

➤ L1 instruction and data cache: much larger, slightly slower pair of caches used by the the CPU.

➤ L2,L3 data caches: shared by CPUs in one core (L2) or the entire NUMA computer (L3)

➤ Main memory: In modules; largest, but slowest to access

PERFORMANCE DIFFERS ENORMOUSLY!

Accessing L1 cache on the Dell server I used as an example last time: 2 or 3 clock cycles. The clock runs at 3GHz.

Accessing the L2 cache takes 12 or 13 clock cycles

L3 access jumps to perhaps 40-75 clock cycles. The actual delay depends on how heavily loaded the memory bus is.

If we need to go to the memory module, there are two cases: the closest memory module will require125 clock cycles to access. A remote memory module takes 250 clock cycles.

IDEAL CASE

All of our 24 cores are busy (one thread each). But maybe some are busy in the kernel, not in my user code.

Each word-count thread is hard at work counting on the current block

At every level of the memory hierarchy, prefetching is anticipating the next instruction needed, next data needed, next block of file data needed, and already loading it.

HYPERTHREADING?

No simple, universal answer

In Ken's word-count, hyperthreading suggests that we should try with 48 cores (2 x 24).   But in fact this doesn't help

In fact hyperthreading helps only for very specific cases, detailed in the CPU user manual.

SUMMARY?

Plenty of opportunities for parallelism....

... yet it didn't feel embarrassingly easy! In fact it was hard.

This is not an unusual experience.