WHY IT IS IMPORTANT (BUT HARD) TO LEVERAGE MODERN HARDWARE

Professor Ken Birman

CS4414 Lecture 3

IDEA MAP FOR TODAY

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

REMINDER FROM LECTURE 1

We had a "word-count shootout" and C++ was much faster!

But what was the C++ program doing that yielded such a speedup, and why didn't the standard Linux approach using existing commands do as well?

And why were Python and Java so much slower?

CORE IDEA

Our task was to compute word frequencies, then output them in a specific sorted order (descending by count, but alphabetic for ties).

The Linux kernel source code has about 26M lines of code in 74,000 files. It contains 4M distinct words, as defined above.

One option is to treat this as a big file and only use Linux commands.

FINDING THE WORDS

Scan the files, breaking out each word and discarding garbage. This is called "splitting".

Build a lookup tree… you'll insert each "new" word into it with a count of 1. If the word is found in the tree, just increment counter

At the end you'll need to output the data sorted in descending order by frequency of each word: a second sorting task.

HOW DID THE PROGRAMS WORK?

The pure Linux version was 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?

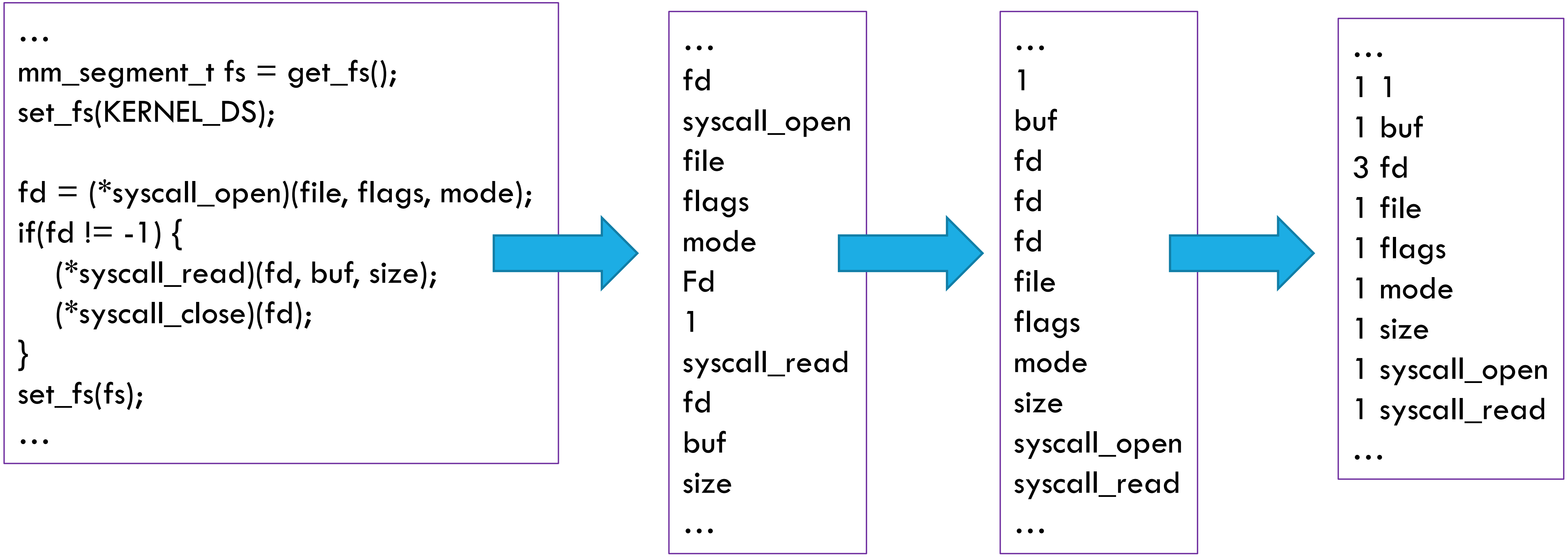
The pure Linux version was easy to write but looks horrible:

```
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 THIS APPLICATION

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



WHERE DID WORD COUNTING OCCUR?

We did it in two steps. First, we sorted the file.

Uniq reads the sorted file and (–c flag) counts identical lines.

The final sort was not shown on that slide: “sort –r –n”. This outputs in descending order by number… **which isn’t quite right!**

sort –r –n will be in <u>reversed</u> alphabetical order for ties!

LINUX SUMMARY

It involved running a chain of 6 processes linked by pipes.

It was quite slow.

#4: Pure Linux (buggy sort order)

real	2m38.965s
user	2m43.999s
sys	27.084s

A "hack" to fix the output order: <u>Negate</u> the counts, sort with –n but not –r, then strip the "-" signs. Ugly, but it would work.

WRITING A PROGRAM TO DO THIS

Same idea, but now we need to “take control”

We will need programming tools to do the sorting and counting.

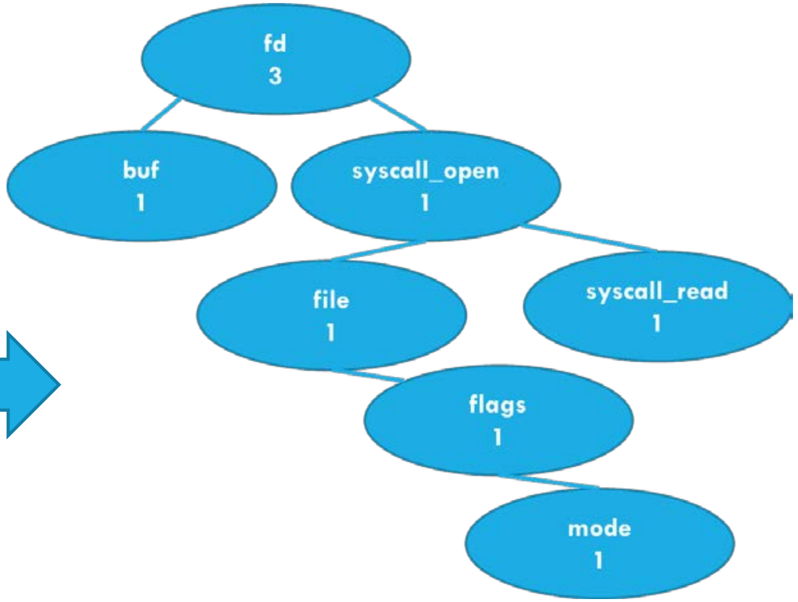
This lets us fix the issue of wanting our output to be sorted by (count,word) with descending count, but alphabetic word

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);
…
```

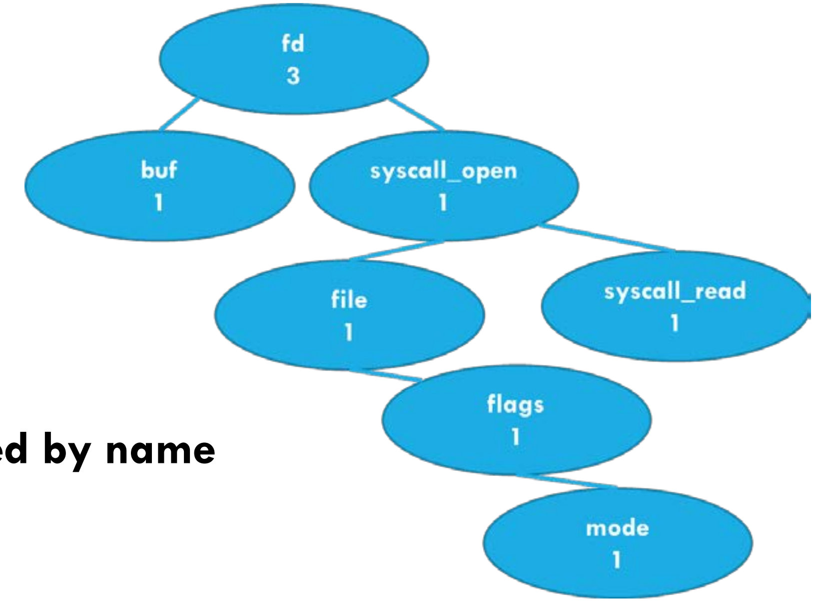
…
fd
syscall_open
file
flags
mode
fd
1
syscall_read
fd
buf
size
…



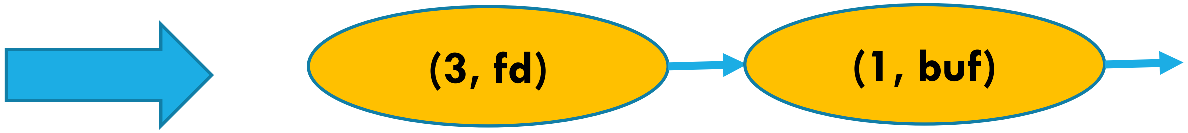
Sorted by name

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

VISUALIZING THIS APPLICATION



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 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 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?

#1: C++ using 24 parallel threads on 24 cores
real 4.645s
user 14.779s
sys 1.983s

We created two C++ versions.

Sagar's was pure and quite fast; you saw it in recitation Monday.

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

C++ DISADVANTAGE

C++ is syntactically different from Java or Python, which can take a little time to adjust to.

A purist, like Sagar, wouldn't like Ken's code: Sagar thinks I could have gotten the identical speed in pure C++ if I had a deeper perspective on some of its costs.

This is why Sagar is teaching you C++, rather than me!

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 then 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.

DATA STRUCTURES AND COMPILATION QUALITY ARE JUST THE START.

Our server had 28 cores, and each core had a way to pretend to be two CPUs (hyperthreading), making 56 CPUs.

At first it seemed as if we should use two threads per core. But Linux needed some cores, and hyperthreading turned out to slow things down. We got the best numbers with 24 application threads.

After they finish, we could then merge the trees into one big count tree, combining the sub-results.

IN FACT, WE SHOULD THINK OF THE APPLICATION AS A SERIES OF TASKS

I'm just using this term to mean "some part of a bigger job". Stages would be another common term for this idea.

Overall, we want to scan all 74,000 files. But it might make sense to subdivide this into a set of tasks.

A single task might do the work of scanning files 1 to 1600

WE WANT TO KEEP ALL 26 CORES BUSY

This form of parallelism forces us to make a choice.

We do this by creating “threads” that each perform a task

THREAD: A BIG TOPIC FOR CS4414

Think about a method that has no return value:

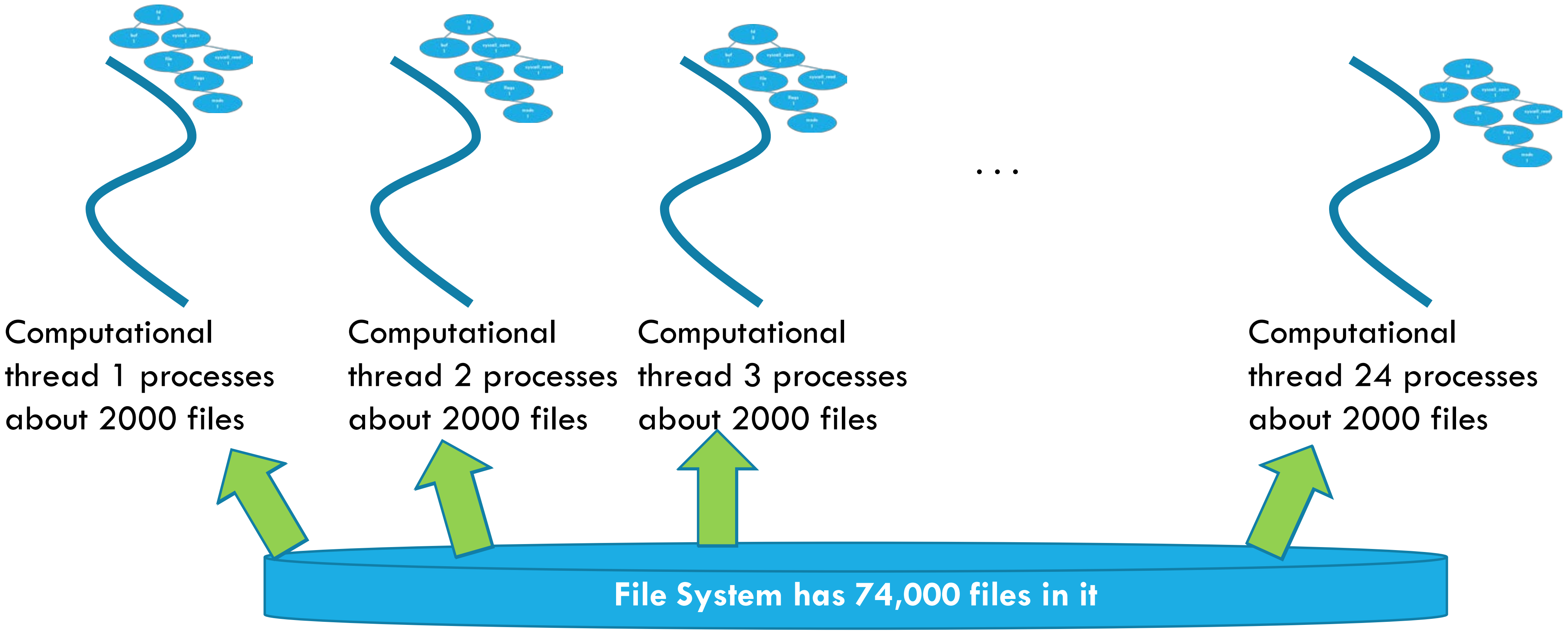
do_something(args….);

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

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

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

VISUALIZING TASK-LEVEL PARALLELISM



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.

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

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

This is how long we waited for it to finish

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.

Later we will see that sharing objects involves adding locking and that this brings costs.

DOWN SIDE OF HAVING 56 TREES…

… we end up with 56x more memory in use!

Linux had 4M “unique” words, but with 56 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 56 times will require 16MB x 56 = 896MB. We have enough memory!

DOWNSIDE OF HAVING 56 TREES…

At the end, we have 56 trees containing sub-counts.

Before sorting, we'll need to merge them.

TREE MERGE IS "EASY"

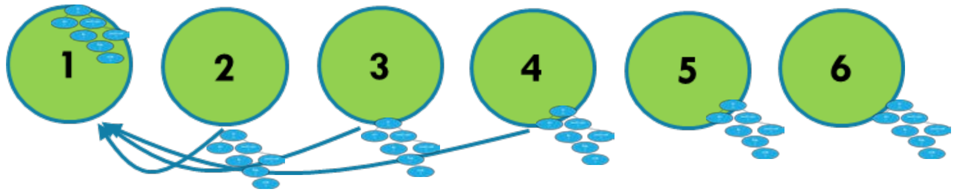
For each node in tree B, look up that word in tree A, sum the counts.

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

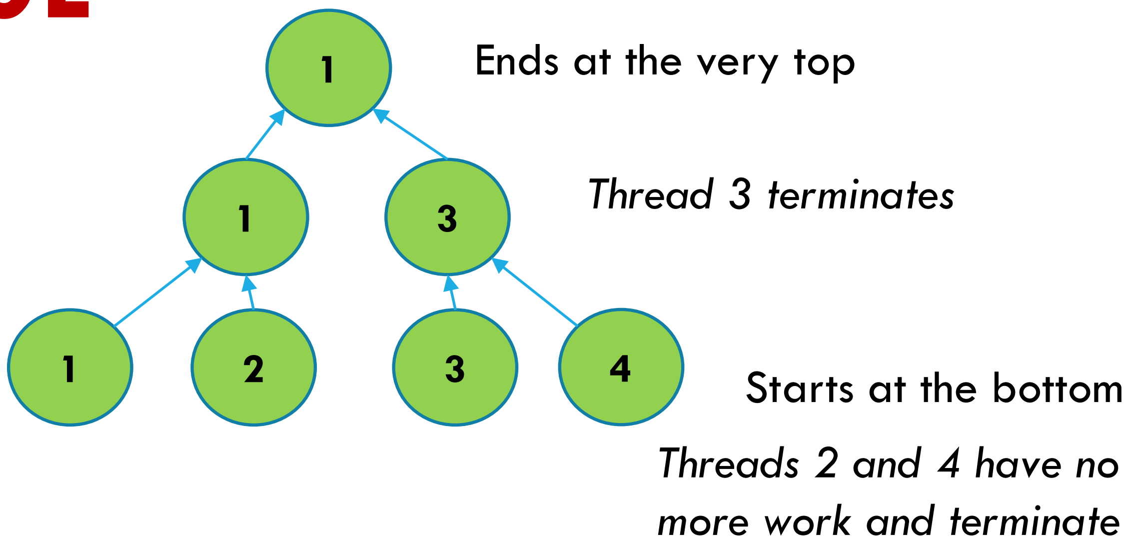
Visualization: Thread 1 runs the merge step.



… this is linear: 55 merge operations. Can we do better?

PARALLEL BINARY MERGE

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



For 56 threads:

Merge [1,2] and [3,4] and … [55,56]

Then [1,3] ……………… [53, 55]

…

Finally: [1, 29]

PARALLEL BINARY MERGE

Ends at the very top

In th
the k

ates

t the bottom

nd 4 have no
nd terminate

For :
Merc
Then

Finally: [1, 29]

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!

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)?

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 do a safe, socially distanced trip to Niagara falls.

There are six of you. One option is to rent a mini-bus and sit far apart, but the mini-bus is slow

A DAY TRIP TO NIAGARA FALLS

Better plan: You rent three convertible sports cars.
With roofs open, each can safely hold two people
Best of all, the cars are "insanely fast".

Gene Amdahl's Tractor in Norway

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.

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.

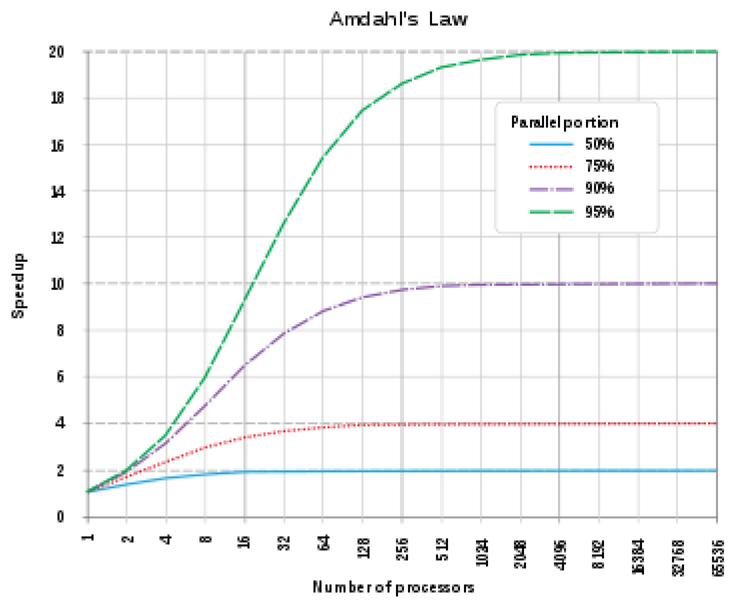
But the computation will also have sequential tasks, which could include sequential logic buried in the operating system or the hardware.

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



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 it 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

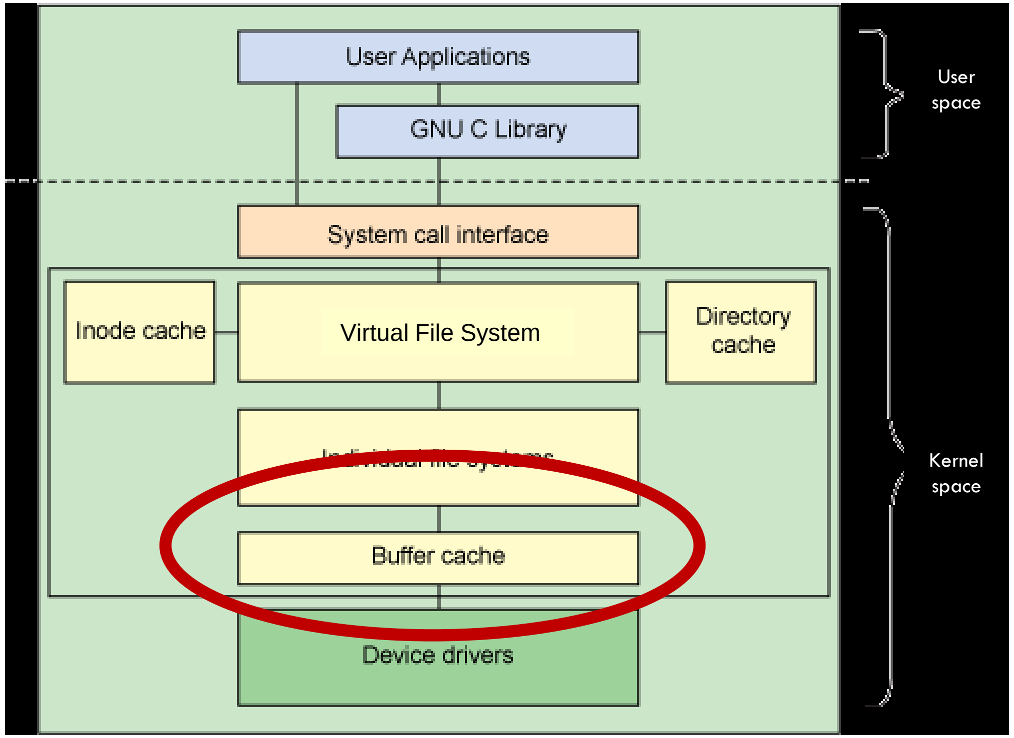
With 56 threads doing concurrent reads, the file system is doing a lot of data fetches from the disk (in "blocks" of 4096 bytes)

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:

- L1: the registers. C++ might "cache" data in them
- L2 instruction and data cache: much larger, slightly slower pair of caches used by the the CPU.
- L3 data cache: shared by the entire NUMA computer (all the CPU cores).
- 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 28 cores are busy (two threads 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.

ADD IT ALL UP?

What limits the peak speed of this motor?

It can be quite hard to document each incremental step that speeds up a complex parallel program such as word-count.

… There are just too many moving parts

But it is easy to gain factors of 10, and when we consider tasks with a lot of data parallelism, this can grow to 10,000x

WOULD GOOGLE OR MICROSOFT CARE?

In fact they are kind of "meh" about 10% speedups. But 10,000 would be a different story.

Broadly, these companies don't get excited about algorithms unless they are sure your code will run in an asymptotic case

But big "real" speedups can add up to real money and they care *a lot about money.* Our 30x example would probably matter.

PERFORMANCE-CRITICAL TASKS

The speedups that count involve scenarios that impact profit.

Examples might include responsiveness of a web page or a Facebook feed, or how fast an ML algorithm trains.

Companies don’t care much about speed for “rare” things.

OUR AGENDA IN THE NEXT 24 LECTURES

In the recitation, learn C++-17 and Linux.

Meanwhile, in class, learn to think in terms of performance-aware program design that considers memory hierarchies, parallelism, prefetching and caching.

By the end of the semester, be able to write amazingly good code that is correct, secure (we'll get to that), and performant!