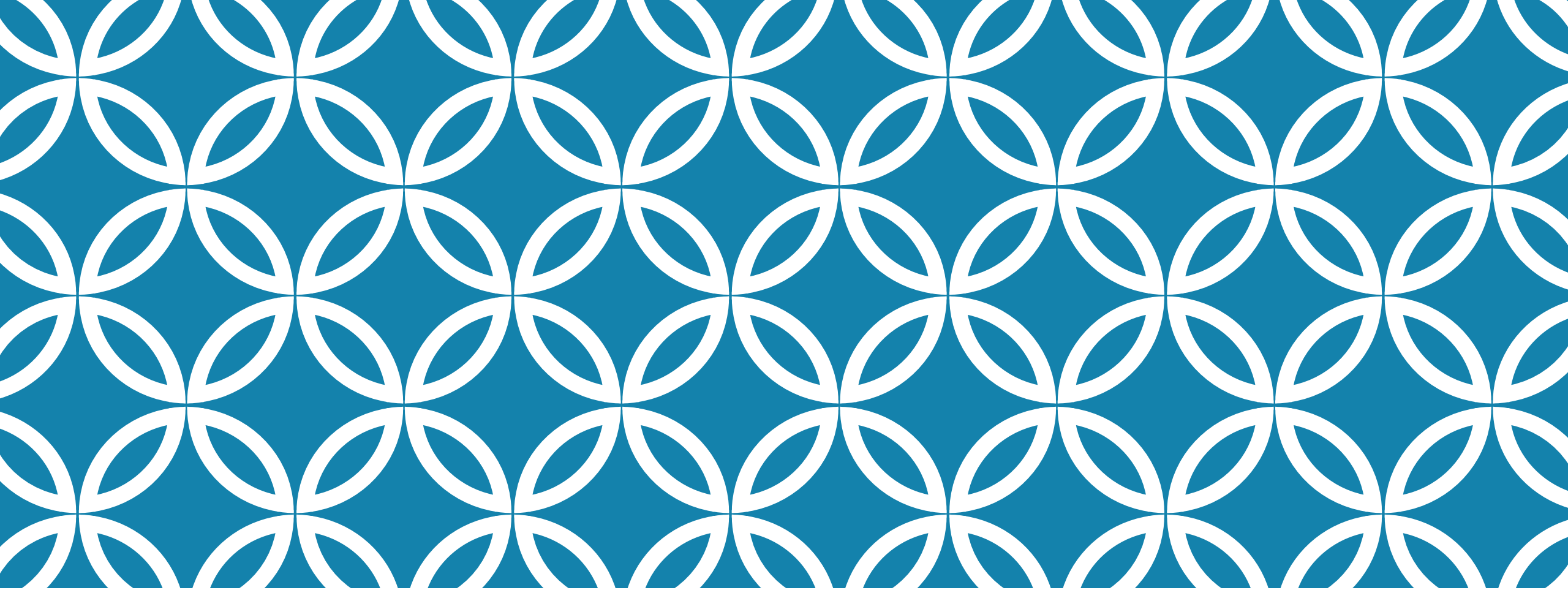


PERFORMANCE: BIG PICTURE

Professor Ken Birman

CS4414 Lecture 11

IDEA MAP FOR TODAY

With so much to keep in mind, how can we possibly understand performance?

Today will be a "big picture" lecture talking about the challenge of visualizing all those different elements

YOUR JOB? BE A DETECTIVE!

Nancy Drew

You suspect that your program isn't the fastest it could be.

You need to be the sleuth and track down the bottleneck!

- This centers on developing a mental image of your code as it executes
- You'll need to have a theory of how fast it "could be", than search for evidence that something is slowing it down

A GOOD DETECTIVE HAS AN OPEN MIND

You do need a mental image… but your theory could be wrong.

Sometimes the most obvious "issue" isn't the root cause– it may be a symptom of the real cause, but "downstream" from it.

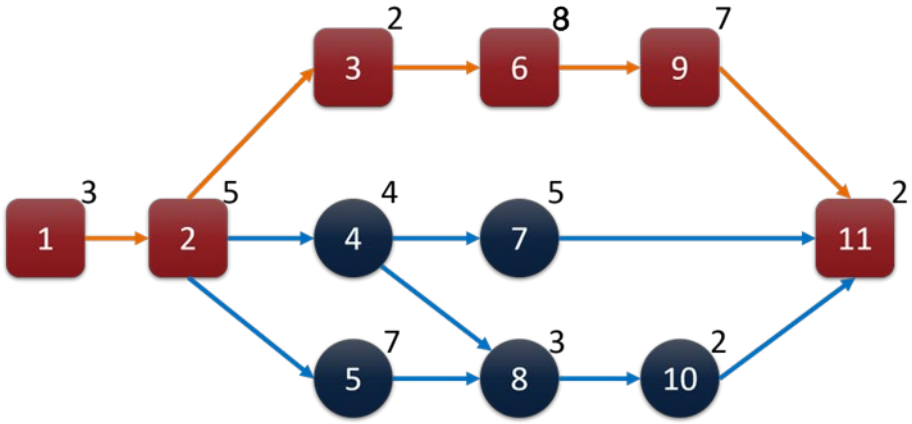
- Example: your code isn't scanning files quickly. *Is it the algorithm?*
- Perhaps, the issue isn't the scanning logic itself. Maybe something else is causing this slowdown, and the scanner is just "waiting"

DEFINITION: CRITICAL PATH

A critical path is the longest end-to-end sequence of sequentially dependent activities in an application.

This example shows 11 subtasks in some program (node numbers) each annotated by the expected delay.

The application has parallelism, yet the steps shown in red determine the critical path



GOOD PERFORMANCE VERSUS BUSY WORK

One huge challenge for performance tuning is that a busy machine often isn't an optimized machine!

We can be busy for a good reason, like training a machine-learning model

But often a busy computer is "thrashing" – doing work pointlessly

IDEAL VERSUS REALITY…

Ideally, we want all the "moving parts" seamlessly interacting to provide a smooth, efficient workflow

In practice we often find that most parts of the system are bottlenecked behind some very busy but ineffective component

PREMATURE… OPTIMIZATION

It can be very tempting to rush to optimize some part of your program where you've just come up with an idea to speed it up

Recall the "drive to Niagara Falls" example from Lecture 3 – sure, a fast car can go faster, but if this means that you catch up to the next bottleneck sooner, you don't really arrive any earlier!

BIG PICTURE PROCESS

It is important to approach a systems programming challenge by really visualizing the whole task – all aspects of the solution

This includes the tasks that the operating system or network will be responsible for, and perhaps even things that other services are providing (in larger settings your programs often talk to services that run on other machines or in the cloud)

DOMAIN CROSSINGS CAN BE COSTLY

A domain crossing occurs when we move data from the storage device to the Linux kernel or from kernel to user memory

They also occur when the user process issues a system call, requesting that Linux do something (like open a file, read data)

And they even occur if threads share a resource and must take turns accessing it (we'll talk about this case a lot in future classes)

MODERN SYSTEMS HIDE THESE COSTS

Your code can access data without considering costs, and Linux will conceal the overheads

But this means that the same logic might be faster or slower depending on factors you aren't controlling.

Gaining control involves intentionally designing code to ensure that data will be in the most efficient place at the right time

TOOLS OF THE TRADE

When you approach a performance question, pause and think about this big picture, and try to visualize *all aspects*

- Is your program reading files? How many? How big?
- Overall, are you working with a really large amount of data, like gigabytes, or smaller things?
- How fast is the hardware you'll run on?
- Complexity of the algorithms you'll be using.

ISOLATION TESTING

Used to study some component of your application. You create a dedicated specialized test to measure its speed or hunt for bugs.

You often can do this by "breaking" your application – with some special argument, main just calls the test logic, then exits.

This allows you to understand the speed of that element and to tune it, without worrying about the rest of your program.

HOW FAST "SHOULD" YOUR CODE BE?

With a whiteboarding process you can often arrive at very crude estimates – rough but still very useful!

- Time needed to do the file I/O
- Computational time per "data item", and "how many items"?
- Will there be a great deal of copying needed?
- What aspects look very sequential to you?

HIERARCHY OF DELAY

Think of each part of your application in terms of

- *Bandwidth:* How fast data can be moved through it.
- *Latency:* How long it takes.

Keep in mind that the disk and Linux and the network are all parts of your application even if you didn't code those

A BUSY THING CAUSES DELAY. BUT SO DOES AN IDLE THING!

We tend to think that delay is always caused by heavy loads

This is sometimes true. If you put a storage device under heavy load, it bogs down. But this might not consume CPU time.

But often, being "overloaded" shows up as "100% idle". That component is spending all its time *waiting,* not *computing*.

CPU IS NOT ALWAYS THE ISSUE!

Sometimes we see components that are waiting for other things.

Each type of device has a minimal delay. This can grow if a backlog occurs due to overload.

- Reading from a storage device? Normally < 1 ms
- Reading over a network? Similar, but also depends on
  - …. Where the service resides
  - …. *How* you talk to it

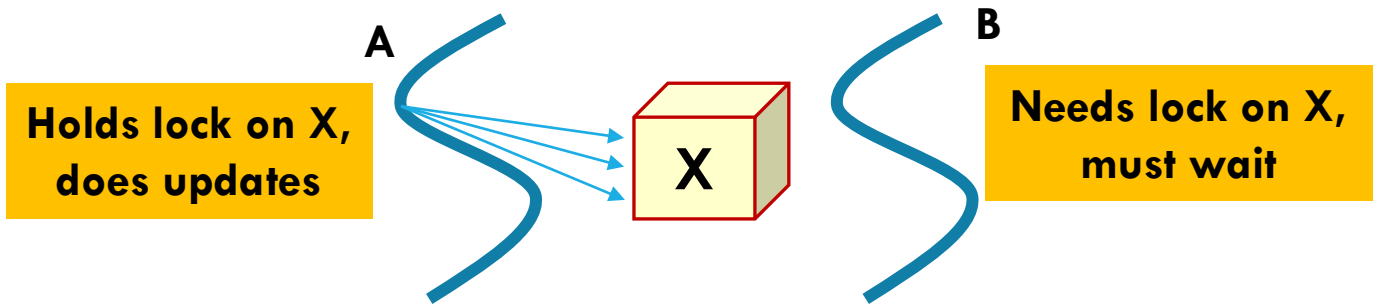
NETWORK TYPES

The fastest networks are used in high speed clusters or data centers. Some use hardware accelerators called RDMA (remote DMA transfer over the network – ultra high bandwidth)

TCP/IP is fast in a cluster or inside a data center, but can be much slower with a wide-area link.

Terms: LAN means "local area network". WAN: "wide area".

LOCKING DELAYS



Later in the course we will be focused on multithreaded code.

In that sort of program, we use "locking" to prevent threads from interfering with one-another and breaking the logic.

Waiting for a lock could be a cause of delay in such cases… and threads with locking are very common these days!

PIPELINING

A *huge* tool is the idea of creating a steady flow via a pipeline

We say that we have a pipeline if there is some "sender" and some "receiver", and they can both run simultaneously

Like a bucket brigade, a pipeline buffers some data (a cost), freeing sender and receiver to run in parallel

PIPELINES HIDE DELAY!

They let us request something "long before" we need it. A producer task can run faster than the consumer task.

If the data shows up when we aren't yet ready to process it, that data just waits in the pipeline

YOU SHOULDN'T LET PIPELINES GET "TOO DEEP"

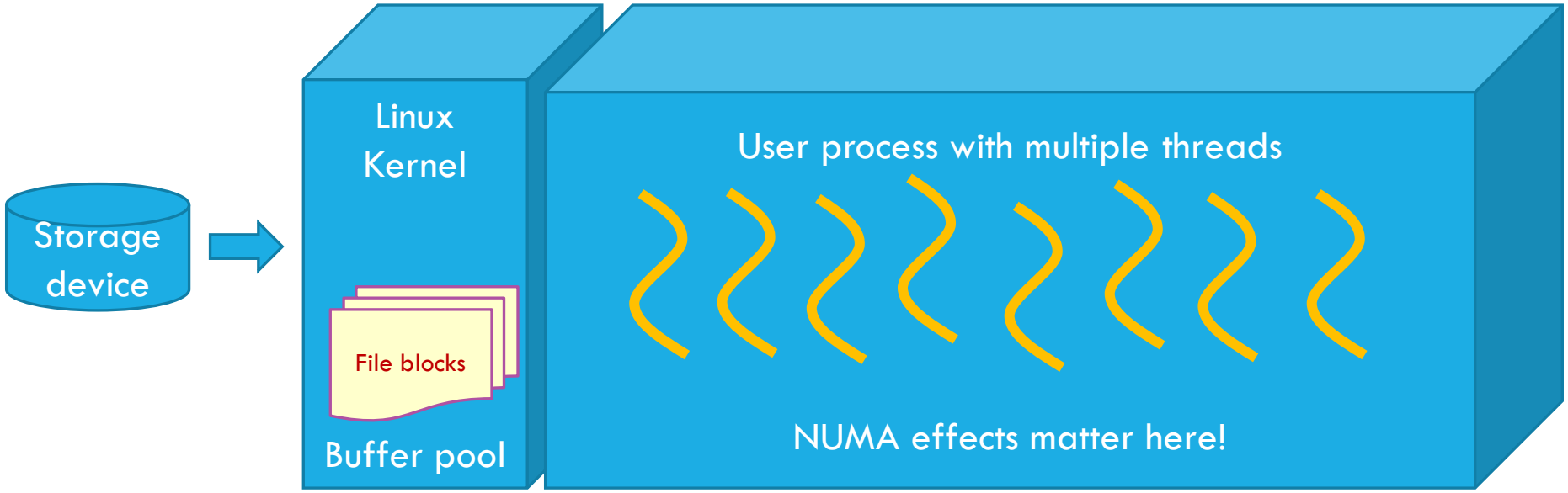
If a pipeline is holding huge amounts of data, or huge amounts of some other resources, costs accumulate

- That memory could have been useful elsewhere
- Linux limits how many files can be open all at once
- Data might even become stale, if the underlying files change

Use your analysis to select a smart pipeline size – "depth"

OFTEN WE HAVE ADEQUATE MEMORY AND PROCESSORS TO SHIFT LOGIC THIS WAY

A pipeline is just one way to use memory to speed things up. A machine has many resources… what is the best *use for* that space?



OUR APPLICATION DESIGN SHAPES PERFORMANCE

It does so explicitly when we launch multiple threads, or decide to have each thread use its own std::map to avoid locking and improve data locality.

It also does so implicitly, when our code includes hints that can lead the C++ compiler to discover the best compilation strategy, or behaves in a way that helps Linux prefetch file blocks.

EXAMPLES OF COMPILATION HINTS

- C++ will optimize inner loops with integer loop variables and simple termination conditions, putting loop variables in registers
- The compiler will put pointers into registers in “tight loops”
- Values reused close to one-another will be held in registers
- Methods with modest numbers of “native type” arguments will be called using registers to pass the parameters

WRITE YOUR CODE AS IF YOU WERE DESCRIBING THE DESIRED MACHINE CODE

The cleaner the mapping to efficient machine code, the easier it will be for C++ to discover your intent and generate great code

In contrast, very complex logic may be harder for it to optimize

This matters for performance-critical code, but not for "general" logic. When tuning a critical path component, aim for simple, ultra-efficient C++ code that the compiler can easily optimize

WRITE YOUR CODE AS IF YOU WERE DESCRIBING THE DESIRED MACHINE CODE

Compilers do best with loops that have “simple” termination conditions, not complicated expressions that call functions.

They are very good at sequentially scanning data structures or arrays in memory.

Any form of straightline code will compile well.

AVOID CODING CHOICES THAT CAN OBSTRUCT COMPILER ANALYSIS

Loop conditions that involve lots of function calls to functions that can't be "pre-evaluated" at compile time.

Lots of inline if statements with unpredictable test conditions

Complicated array indexing with expressions that can only be evaluated "at runtime"

REMEMBER THAT CLASSES AND TEMPLATES ARE AUTOMATICALLY ELIMINATED!

C++ eliminates these at compile time, and once you understand how it does this, you can "visualize" the resulting code.

It does end up with very long, messy, variable *names*. But names are just compile time information and won't change the machine code.

This is in contrast to Java or Python, and one reason C++ performs so well!

Once templates are eliminated, *constant expression evaluation* eliminates most remaining overheads due to classes and generics!

THE CPU PLAYS A BIG ROLE, TOO!

After C++ maps your code to machine instructions we aren't even finished!

The CPU itself will look ahead at many instructions and try to pre-load operands and might even reorder instructions! It also predicts which way branches (ifs, loops) will go.

The rule is to "**preserve the semantics, not the rigid ordering.**"

EVERYTHING IS PROGRAMMABLE. BUT NOT ALWAYS DIRECTLY USING C++ CODE

We have many ways to "take control" of the operating system, or the devices, or choices the compiler will make.

They are not always the identical mechanism. Our C++ code is our way of talking to the compiler, and through it, ending up with machine code matched to the hardware.

But the *pattern of system calls* we issue is our way of talking to Linux

BUFFERED PRINTING

When a program is being debugged we often send output to the console.

This is very helpful for debugging. (Useful features: ^Z to pause the program, fg to restart it, ^S/^Q to pause/resume printing)

But for this to work, your program will do one I/O system call either per line, or per character. (Default: per line)

IMPROVING CRITICAL PATH PERFORMANCE

Often a program generating an output file turns out to have a critical path in which the actual output operations are costly.

Core issue? Linux I/O system calls can be expensive and slow. Writing one character or one line at a time "maximizes" this cost.

Remedy? Write into a buffer… configure it to output each time 4096 bytes accumulate.

… WHY 4096 BYTES?

Linux is optimized for 4K I/O operations

So if you write 15 characters, then 7, then 24… this is slow! You are doing multiple system calls when perhaps one would suffice

As a result, the streaming I/O library can **buffer.** It switches to 4K mode if the output target is another program (via a pipe) or a file (via I/O redirect). This improves efficiency dramatically!

EVERYTHING HAS A PRICE…

The downside of 4K writes is that if a program dies (or terminates) while buffering data that has not yet been written, the last lines won't be written out.

It becomes important for you to "flush" those last buffered lines

So here we see a form of active control, yet it isn't purely in the form of writing C++ code that takes control of something

WHAT WAS THE PRICE?

You gained performance, but accepted that I/O will be buffered and hence that your program might run for a while before each new write occurs.

This creates a mental cost: if the program stops unexpectedly or crashes, some I/O might not have been done. You need to be sure to flush that I/O – and this is the cost of buffering.

"**Simpler, but slower**" versus "**faster, but a little harder to understand**"

DON'T SWEAT THE SMALL STUFF

Start by trying to understand whether something is 10x slower than it should be.

Finding the major bottlenecks, or the very inefficient pieces of a solution, can pay off: fixing those first gives dramatic improvements… After that, you can focus on smaller things

ALGORITHMS (SOMETIMES) MATTER…

As a student you've learned a lot about algorithms

If the complexity genuinely reflects the costly resource, and we are in a situation where asymptotic costs are the bottleneck, picking the right algorithm is key.

But those two "ifs" are not minor points!

EFFICIENT ALGORITHMS DON'T ALWAYS FOCUS ON THE COSTLY RESOURCE

Many algorithms were created using standard metrics like compute time for one thread, or space consumed

In a parallel setting with a lot of memory, we might be fine with spending memory to save time – we saw examples earlier today.

And computing may actually be "cheap" too!

WHAT WOULD BE A COSTLY RESOURCE OTHER THAN MEMORY OR CPU TIME?

Think about disk access

If an algorithm is designed to focus on, say, balancing a tree for constant depth, **but the tree is on a disk**, the tree nodes might not really match one-to-one with 4K disk blocks.

The algorithm might do a lot of disk block reads and writes that the complexity metric doesn't count. Those I/Os are costly!

… ALGORITHMS ARE CONCEPTUAL TOOLS

When we work with algorithms we are working in a very conceptual way, highly abstracted from concrete resources. An algorithm is a **design pattern**

Our challenge as systems builders – engineers – is to map our understanding of the application into "relevant" algorithmic questions where the metrics we optimize are the costly aspects of the overall application pipeline.

SUMMARY: BIG PICTURE

The big picture is central to performance-oriented systems programming. Concurrency can hide pipeline delays.

We "gain control" over mechanisms in many ways – sometimes with our direct C++ code, but sometimes by arranging our program in clever ways, or by giving useful hints to the C++ compiler or Linux knowing they will make smart choices

SUMMARY: BOTTLENECKS

Start by understanding bottlenecks and the critical path, and visualizing the desired flow of your computation.

You won't be able to improve performance unless you understand goals, and understand where you started.

Random changes just make code messy, add bugs, and might not help – we want to only make *the right changes*.

SUMMARY: BOTTLENECKS

They really come in two forms

- Unavoidable work being done as efficiently as possible
- Accidental work (or delays, perhaps even idle time) arising from some form of mismatch between our code and the system

Once you identify a bottleneck, you can often intervene to improve exactly the slow step

NEXT LECTURE: PERFORMANCE MONITORING

Linux is full of tools we can use to measure performance and even understand overhead sources

These tools enable us to compare actual behavior of a program with our conceptual expectations

We'll see how they can let us find bottlenecks