University of Illinois at Urbana-Champaign
Dept. of Electrical and Computer Engineering

ECE 408 / CS 483 / CSE 408: Applied Parallel Programming

Parallelism Then and Now

What Will You Learn Today?

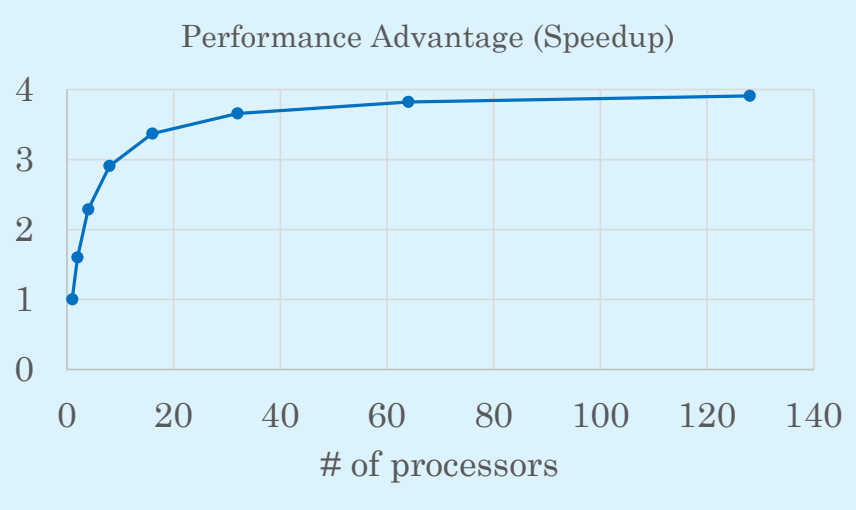
- to gain perspective on the past, present, and future of parallel programming
- to understand some of the remaining challenges

Parallelism Does Not Affect Algorithmic Complexity

Where is parallel programming headed?

Let's **start with** some historical **perspective**.

In practice, **parallelism provides a fixed gain**.

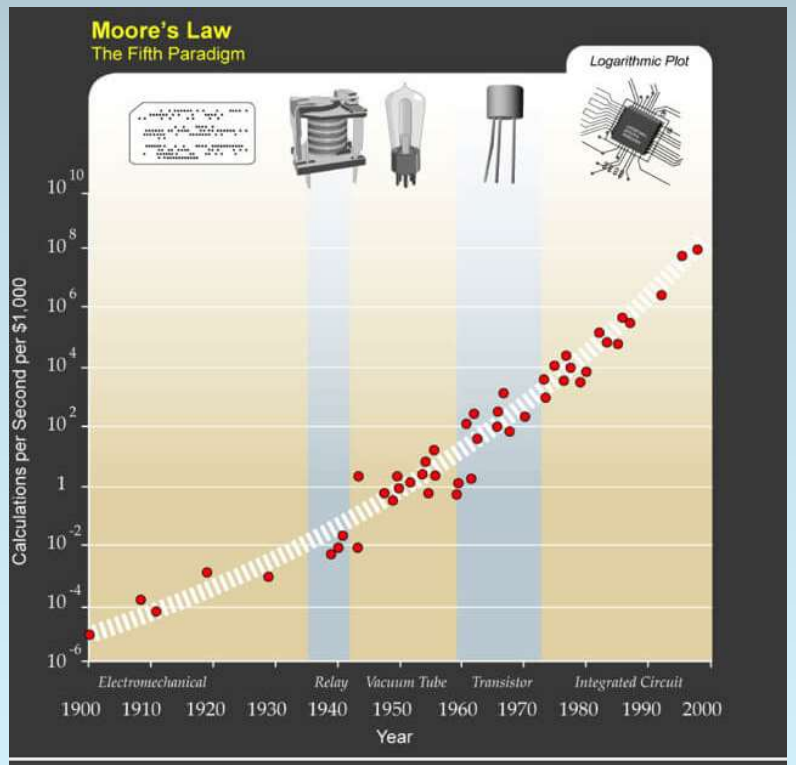


No code/input scales to infinite resources).

An Easy Alternative: Do Nothing for a While

Want a fixed gain?

- **Wait** a couple years!
- (Performance of CPUs grows exponentially over time!)
- **Or build in hardware**.



This chart, from Kurzweil's *The Singularity Is Near* (2005), shows the exponential progress of computing across five technologies. Integrated circuits have continued to follow the trend since the book was published but may soon encounter new physical limits. Will they be succeeded by a sixth paradigm?

Suffering is Only Popular in Movies (and Novels)

Before the **late 2000s**,

- almost **no one cared** about parallelism:
- <5% of research community
- and a miniscule fraction of industry.

Not many are motivated by

"No gain, just pain!"

Good Idea! Why Not Do Your Thesis on That?

An old joke:

"Only graduate students write parallel programs."

Explanation for undergraduates from Greg Pfister's *In Search of Clusters*:

"…parallelism is the wave of the future, and graduate students are inexpensive, intelligent, and motivated."

In 2004, Hardware Underwent a Sea Change

Then came the power wall.

Intel cancelled a product.

Hardware went parallel instead.

Multicore chips became ubiquitous.

Today, it is **difficult to buy** …

- a desktop/laptop with one core, or even
- a smartphone with **one core**!

Meanwhile, in Another Part of the Industry…

Parallel graphics hardware

- developed in late 1990s / early 2000s,
- with **more compute** than CPUs, and
- **more memory bandwidth**
- (both resources had motivated parallel programming).

In 2007, NVIDIA introduced CUDA, a reasonably easy way to program them.

Low-Hanging Fruit: Ubiquitous Multicore

Parallel computing was demanding:

- Use 16× the resources?
- Deliver at least 15× performance.

But by 2010, parallel hardware was everywhere!

Why not exploit it?

- 2× or 4× is better than 1×, and
- not so hard to achieve.

Today, Millions Can Write Parallel Code

Today, millions of people can write GPU code.

And **GPUs are ubiquitous**,

- in every platform
- from $100 cell phones
- through supercomputers.

What will happen next?

Let's review some of the hurdles.

Not Everything Can Fit on a Chip

Historically,
why did people write parallel code?

Often for applications that

- **needed more resources**
- (compute power, memory, memory bandwidth, I/O bandwidth).

Important caveat: **one chip still** has **one chip** of memory and I/O bandwidth, no matter how much computation it can do.

Parallelism Under the Hood

Another reason for parallelism:

- **raw performance / scalability**
- NOT supercomputing—that market never relevant,

In databases! For decades,

- **SQL queries** have been **executed in parallel**
- (courtesy of the $10B database industry).

My hypothesis from grad school: for enough $, any application can be made embarrassingly parallel!

(Entertainment $ drove GPU development, of course.)

Many More Cannot Write Parallel Code

Although many programmers can write GPU code, **many more cannot**.

And **parallelism** is **not always embraced**…

For example,

- when PlayStation 3 incorporated IBM's parallel Cell processor in 2006,
- many game designers didn't want to write code for it,
- since that code could not run on other consoles.

A Thriving Business from the 1980s

Old model: **buy ~1,000 PCs** combining

- various motherboards,
- various processors,
- various disk drives, and
- various graphics cards.

Sell time on your **machines**

- to help companies
- **debug** their **sequential programs**!

A New Business for the 21st Century

New

Old model: **buy ~1,000 PCs** combining

- various motherboards,
- various processors,
- various disk drives, and
- various graphics cards.

Sell time on your **machines**

- to help companies
- **debug** their **sequential** parallel **programs**!

Lots of Microarchitecture to Worry About!

You should recognize the issue from class.

Which bottleneck matters to your code?

- Number of registers
- Number of threads/block
- Number of threads/SM
- Number of blocks/SM
- Shared memory/block
- Shared memory/SM
- Constant memory size
- Number of FLOPs
- Number of threads/warp
- Latency of DRAM
- Bandwidth to DRAM
- Latency to shared mem.
- Bandwidth to shared mem.
- … or something else?

Be sure to consider future GPU designs, too!

Portable Performance is Still a Hard Problem

How will we handle cross-platform issues?

How will we handle generational issues?

A lot of **progress made** in the last decade
by … graduate students!

And mostly by **UIUC graduate students**,

because they're the best, of course!

(But there is still work to be done!)

Is Better Performance a Competitive Advantage?

Another question for the future:

Will performance become a competitive advantage in products?

Maybe. Historically, performance was not an issue in most software companies.

But performance is power (finish faster and turn the processor off).

Lots of room for apps, languages, and runtime systems in the datacenter and on mobile devices.

There are Several Other Challenges

Now let's ask a slightly different question:

What's [still] hard about parallelism?

Some of the **challenges** you already know…

- **atomicity** (a technical issue)
- **performance portability** (system details vs. algorithm)

Others we'll talk about now

- [in]compatibility with good **software engineering** practices
- expressing **precedence** and dependence
- **determinism** and reproducibility

Parallelism is Easy to Find in Specifications

Let's start with software engineering.

To parallelize a code, we (usually)

- need to **find** a source of **parallelism**.
- Almost always **trivial** to identify opportunities **in a task specification**.

In fact, I spent a fair bit of time

- finding a problem with no obvious parallelism
- (which in fact has a closed-form solution).

Harder to Find / Implement in Existing Code

But programmers rarely start from scratch.

Not always easy to see good opportunities for parallelism **in existing code**.

Often **challenging to** implement and **integrate** with existing code.

Focus is Different: Are the Two Compatible?

Is the difficulty fundamental? Maybe.

An observation made based on a talk by Matt Frank (credit to him for the idea, and blame to me for making it overly trite):

Parallelism is a **control** abstraction.

Software engineering focuses on **data** abstractions.

That difference should worry you.

Yes, People Have Tried

Yes, people have tried myriad ways of expressing parallelism in terms of data

- dataflow (hardware and software architectures)
- Communicating Sequential Processes / actor languages
- pipeline parallelism
- stream processing
- processor in memory

They work sometimes, for some applications.

Information Hiding is Key to Software Engineering

Another way of looking at the same issue…

In 1971,

- **David Parnas** defined **information hiding**.*
- Build **modules** to **expose only** the **interface**.
- Implementation **details must be hidden**.

*D. L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM,* 15(12):1053-1058, 1972. (Based on his 1971 CMU CS tech report.)

Information Hiding is Challenging for Parallel Code

In 1990,

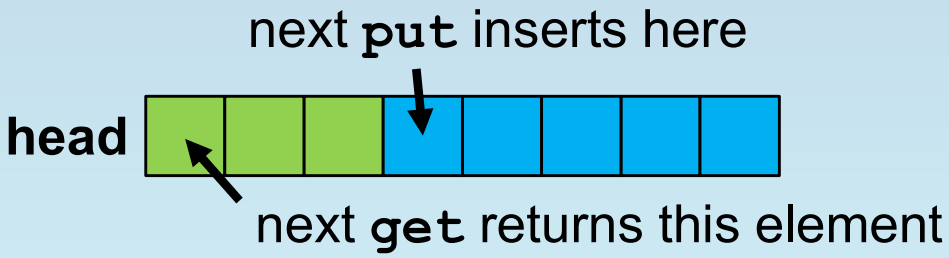
- **Satoshi Matsuoka** defined the **synchronization anomaly**.*
- Synchronization (atomicity, precedence, dependence) is at odds with inheritance.
- **Hard to encapsulate synchronization**
  - such that authors of derived classes need not
  - know about the details and/or rewrite them!

*S. Matsuoka, A. Yonezawa, OOPSLA/ECOOP'90 Workshop on Reflections and Metalevel Architectures in Object-Oriented Languages, ACM, Aug. 1990. Usually, the 1993 book chapter is cited instead.

A Simple Example Illustrates the Problem

A simple example:

a finite queue with atomic get/put.



Extend with a new method:

- get first two elements atomically
- must synchronize with get/put

Solutions, Anyone?

Since 1990,

- hundreds (thousands) of papers have tried to provide a solution.
- Many papers / languages claim success.
- Oddly, no one claims success about anyone else's language…

Solution is **absent from** most common **parallel interfaces** (POSIX, Java, CUDA, …).

(Another fun one: get first element from each of two queues.)

Useful to Express Precedence and Dependence

Another common problem:
expressing precedence and dependence.

Atomicity does not imply order.

Sometimes …

- need to finish one task before another starts,
- a task may need to create other tasks, or
- a task may want to terminate other tasks (for example, a parallel search thread that finds a solution can terminate other searchers).

CUDA Extensions Support Fork-Join Parallelism

In **basic CUDA**,
- **dependence** can **only** be expressed
- **within thread blocks and**
- **across kernels**.

In 2012, NVIDIA introduced an **extension** to CUDA based on fork-join parallelism:
- **thread block can launch** "child" **kernels**,
- and can wait for all kernels to complete.
- Nested: kernel is complete only when all children are complete.

Compiler Technology Makes it More Viable

Unfortunately, the **overheads are large**.

So, in his Ph.D., Izzat El Hajj
(Ph.D. UIUC, 2018)

- used **compiler techniques**
- to aggregate dynamic launches and
- **amortize the overhead**.

What's the Hard Part of Dependence?

Why is precedence challenging?

As you've seen,

- even cross-thread dependences can be subtle,
- so **programmer may forget** to enforce them,
- which may or may not cause immediate problems.

In some cases, programmers

- may **choose to speculate** using old values,
- but software speculation **can lead to unexpected behavior**.

Make Sure that You Know the Rules

Finally, the **exact meaning** of synchronization operations are sometimes **subtle**.

For example, **which of the following are guaranteed complete by `__syncthreads`?**

- reads/writes to shared memory
- reads/writes to global memory
- atomic operations on global memory
- execution of the same **`__syncthreads`** in thread blocks with lower linear order
- writes back to DRAM (from chip cache)
- **`printf`** statements

One Last Problem: Determinism and Reproducibility

Imagine working on a new kernel.

You execute the code, and it crashes.

Then you execute it again, and it doesn't.

What happened?

Non-determinism!

Not a Big Issue Today in CUDA

Compilation happens in the driver, and

- ISAs may not vary much, so
- variations in code are few
- (less true if you also execute on other systems).

Thread block execution strategies probably don't vary much.

And thread blocks interact weakly.

Warp scheduling is perhaps the most common possible source.

But CUDA is Evolving

But **advanced CUDA**
- **allows parallel kernels**, and
- CUDA has evolved to become **more flexible**.

Why?
- Without that flexibility,
- some applications are hard to express
- in a way that enables performance improvement.

And the Problem May Get Worse…

As CUDA and GPUs evolve,

- **problems may be more common**
- (as they are in most parallel systems).

One thing some of you have already seen:

- Add a `printf` to make the code work.
- Remove it and the code crashes…oops.

Data Races Depend on Numerous Factors

If a code has data races, the results depend on

- number of parallel thread blocks,
- number of parallel and child kernels,
- thread block scheduling (depends on number of SMs, relative speed of thread blocks from different kernels, relative resource requirements…), and
- warp scheduling
  (affected by all latencies and warp size).

Debugging in the Future (or Past)

So now you're using a more sophisticated version of CUDA …

… and working on a new kernel.

You execute the code, and it crashes.

You launch the debugger
(there is one, you know…)
and set a breakpoint.

What happens at the breakpoint?

Debugging is Not Deterministic, Either

One warp in one SM hits the breakpoint

- Do the other warps in the thread block stop?
- What about other thread blocks on that SM?
- What about other SMs?
- What about child kernels? Parent kernels? Unrelated kernels?

Cross-chip communication (to stop the other warps) is not instantaneous.

How close are your races?

Applications Considered on a Rolling Basis

Researchers have come up with many good answers over the last 50 years.

But there's still work for you to do!

Maybe for your Ph.D.?