Karlsruhe Institute of Technology

Concurrent Programming in C++

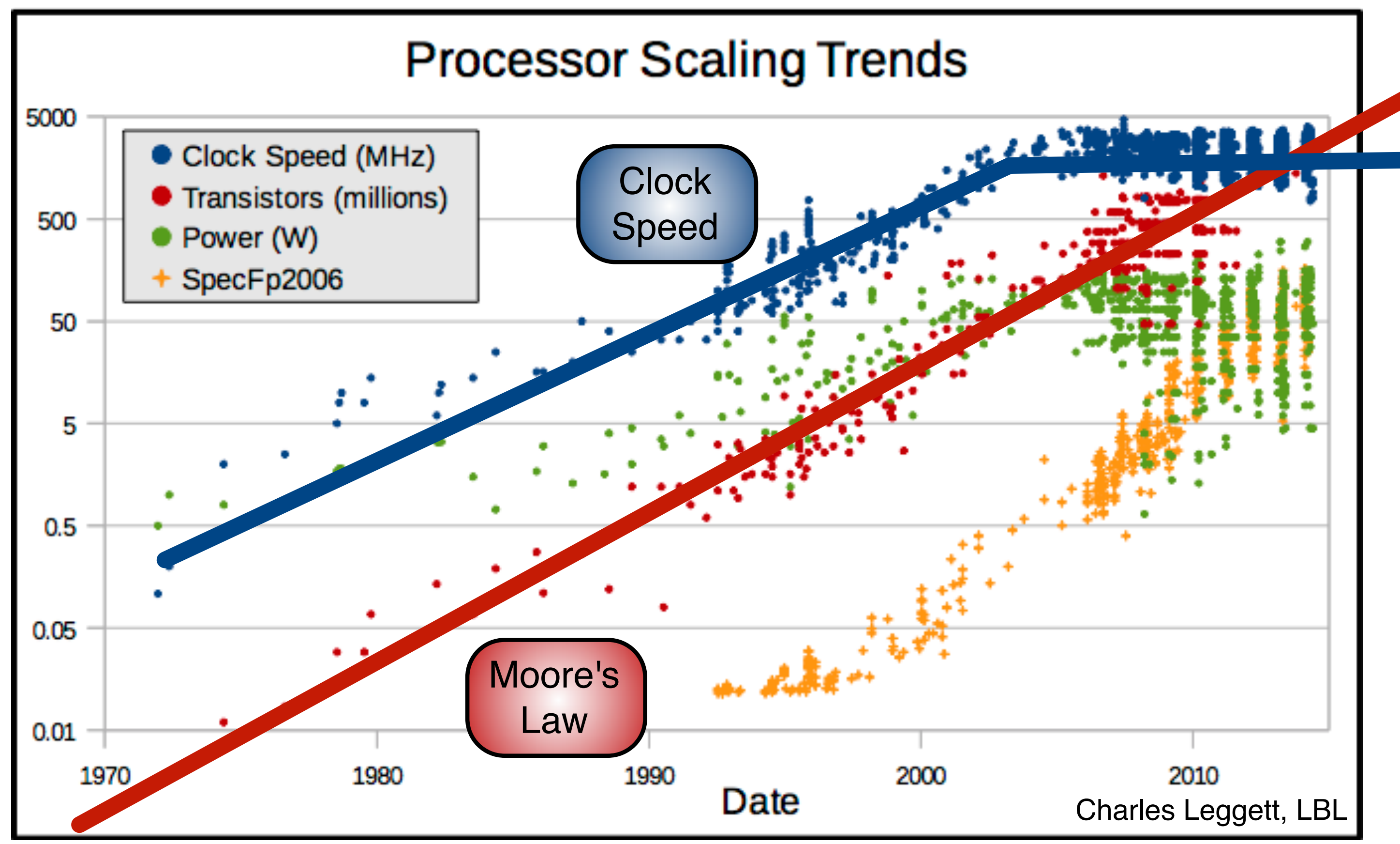
Graeme Stewart and William Breaden-Madden

University of Glasgow | School of Physics & Astronomy

2015-09-06

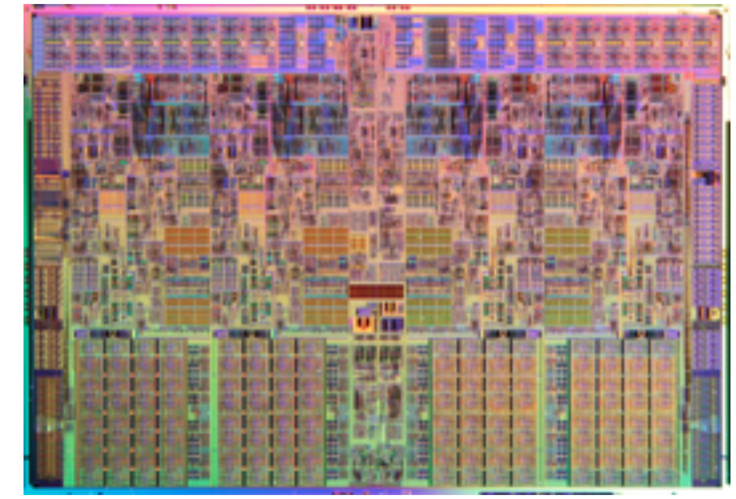
Modern CPU Evolution

- Moore's law continues for now*
  - So transistor density doubles approximately every 24 months
- This used to mean that computers were about x2 faster every 2 years
- But not anymore - hardly any increases now in clock speed
  - So little increase in single threaded performance



*With some signs of slowing, however!

CPU Real Estate



- Increasing numbers of transistors are looking for something useful to do:
  - Vector registers
  - Out of order execution
  - Multiple Cores
  - Hyperthreading
- However, although these things are good at increasing the theoretical throughput of the CPU, exploiting these techniques can be far from easy

Do we need to bother with concurrency?

- For a lot of problems trivial parallelisation is sufficient to exploit multiple cores
  - This can even be exploited via slots in a batch system and we don't care what runs where
  - High energy physics has been able to adopt this technique for many years, very successfully
- But sometimes this isn't enough
  - Overheads of trivial parallelisation can be non-trivial (file merging, message passing, batch and grid workload managers)
    - And are sysadmins really going to want to have 63 job slots on a Xeon Phi…? Unlikely.
  - Memory consumption can be considerable
  - May not make best use of hardware
    - Cores might go idle, because we run out of memory before we run out of cores (lightweight cores, many core machines)

Types of Parallelism

- Data Parallelism
  - When we do the same thing to many independent data objects

- Task Parallelism
  - When we use the same data as input to various different tasks

- Mixed Parallelism
  - But often we should mix and match these approaches to get the best results

Going Parallel

- Programming in parallel requires thinking about a problem to identify the best way forward
  - Decomposition - how can I break the task down?
  - Scaling - will things work as the task or the resource gets bigger?
  - Patterns - does this task fit a model other people solved already?
  - Correctness - am I getting the right answer, or just a faster wrong one?
- As you discover more about parallelism you'll develop better intuition about how to approach all of these questions
- Adding parallelism is a fundamental design choice - you are choosing a *different* path from a serial code design

The parallel punch-up: Ahmdal vs. Gustavson

- Just about the first thing everyone learns about parallelism is *Ahmdal's Law*
    - The observation that as the parallel portion of an applications run time goes down with increasing concurrency, the total run time becomes dominated by the serial portion
    - $t = s + p/n$
    - As $n \rightarrow \infty$, $t \rightarrow s$
- So, for a fixed problem size, serial overheads may kill your ability to exploit parallelism

The parallel punch-up: Ahmdal vs. Gustavson

- However, Gustavson observed that this is only true for a *fixed problem size*
- If you can scale up the problem, as your parallel resources increase, then your scaling will be maintained
    - $t=s+p/n$
    - As n↑ try to increase p as well
- i.e., Scale the problem to the number of processors, don't fix it in size

What we look at today

- Today’s tutorial is divided into two parts
  - Multi-threading in C++
  - Intel Threaded Building Blocks

C++ Threads

- With C++11, language support was added for concurrency
  - This was a great step forwards, as it properly defined the behaviour of C++ in concurrent environment
    - e.g., the C++ memory model
  - Although support might be considered quite basic (compared to other toolkits), it’s a great step forwards from the low level threading libraries used before, like pthreads
  - As this is *language* support, it’s not going away and will probably improve
- Using C++ threads is a good way to learn some of the basic advantages and problems that come up when programming concurrently

What we do with C++ Threads today

- How to spawn threads, wait or detach them and identify them
- Passing arguments to the thread's invocation function and what can go wrong when we try to do that
- What can happen if threads access data at the same time: data races
- How to protect against this using a mutex and how to avoid deadlock
- Avoiding locks with atomic variables
- How to launch from asynchronous functions and read their return values from a future

Intel Threaded Building Blocks

- High level toolkit for managing concurrency in C++
- Not oriented at threads at all, but at parallel tasks
  - Much more intuitive way to think about our scientific problems
- High level pattens supported directly
- Uses *task stealing* to keep working threads busy when the problem is inhomogeneous
- Very flexible task scheduler that can be interacted with directly, if needed
- Support for flow graphs, which can be used to build complex workflows
- Support utilities for concurrency: thread safe containers, fast threaded malloc, timing functions

What we do with TBB today

- Using parallel_for to execute data parallel tasks concurrently
- Using parallel_reduce to parallelise calculations that have dependencies across data objects
- Looking at how concurrent_vector can be used
- How TBB pipelines work, as a way of chaining different tasks together in an ‘assembly line’

Final Note

- We'll look at two threading toolkits today
  - However, there are many toolkits available
- Only you are going to know which one fits best with your problem and the resources you have at your disposal
- *Threading is hard, so choose your weapon with care!*