University of Southampton

University of Southampton

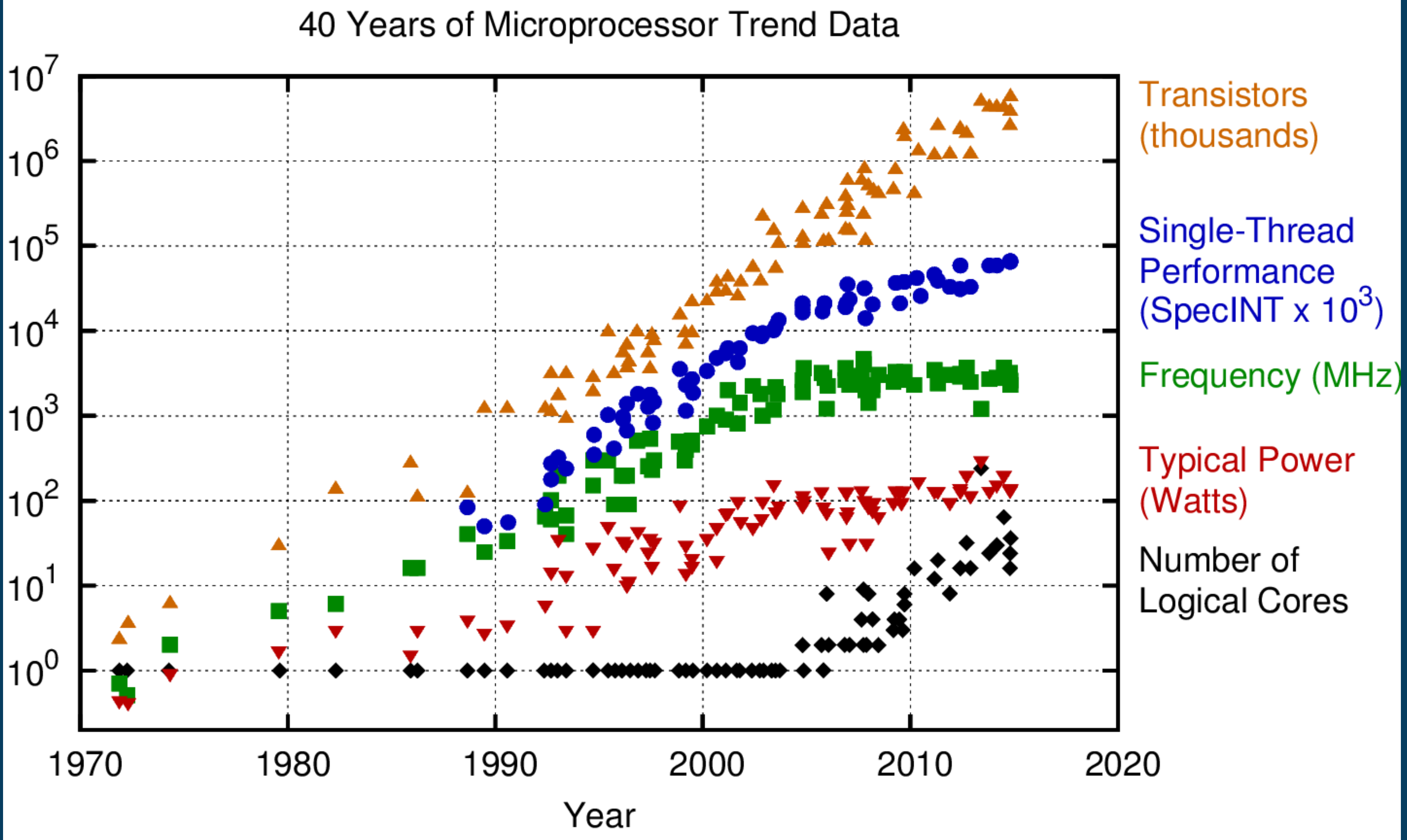
COMP2212 Programming Language Concepts

Introduction to Concurrency

Dr Julian Rathke

Concurrency vs Parallel Computing

- What is Parallel Computing?
  - Multiple computations are created by dividing a task in to smaller, **independent**, pieces of work. These are then executed simultaneously.
  - Can be multi-core, symmetric multiprocessing, distributed, massively parallel
- What is Concurrency?
  - Several computations executing simultaneously, potentially **interacting** with each other - this is typically done on a single (multi-core) machine
- “Multiprogramming”first became a subject in the 1960s with important contributions by Dijkstra, Hoare and Brinch-Hansen
  – invented foundational concepts of *critical regions*, *mutual exclusion*, *locks*, *monitors*, etc. that are still being used today



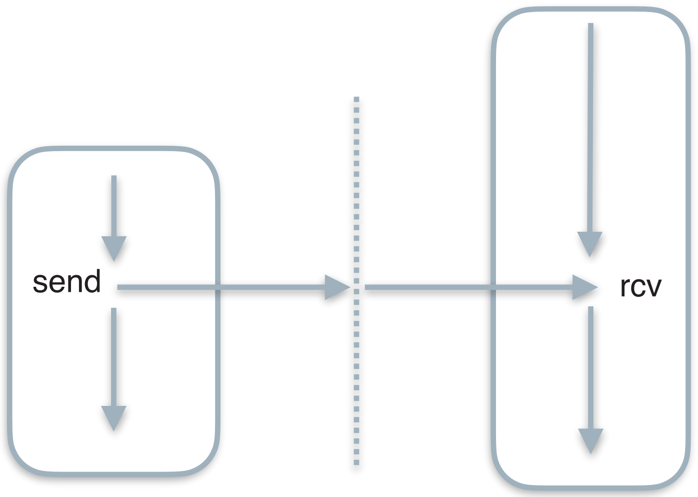
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

Concurrency from first principles

- What is clear from the definition of concurrency is that in this computational model, the simultaneously executing tasks will interact.
- We call each executing sequence of control a **thread**
- For threads to interact they must be able to synchronise
- This means that threads must be able to send signals to other threads and threads must be able to recognise when signals have been sent from other threads.
- This is the basis of all inter-thread communication however how the signals are sent, what information is sent with them and how the threads synchronise on signals varies greatly across systems.

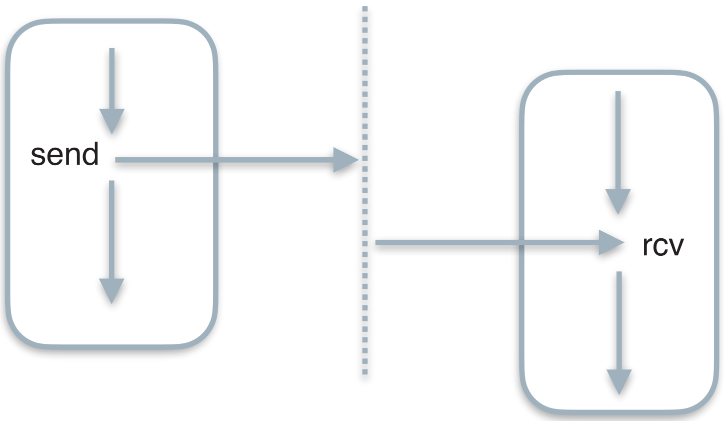
Signal and Wait

- One paradigm of synchronisation is that of "signal and wait".
- A thread that signals another thread sends its signal and then enters a waiting state so that it cannot be scheduled further until the signal is received
- This is also called **synchronous communication** - threads that are communicating rendezvous at a given point in their execution.



Signal and Continue

- An alternative paradigm of synchronisation is that of "signal and continue"
- A thread that signals to other threads does not need to wait until the signal is received. It may continue its execution immediately after signalling.
- This is also called **asynchronous communication** - threads that are sending do not need to rendezvous. Threads that are receiving signals do still need to rendezvous on a signal.



Semantics First

- If we were to introduce concurrency support in to a programming language a reasonable way of starting would be to consider the semantics
- We would need to choose the synchronisation mechanism and what operators the language will feature
- But how do we write down a small step semantics for multiple threads?
- We could imagine a relation of the form

$$T1,T2 \ldots ,Tn \longrightarrow T1',T2' \ldots , Tn'$$

- where each thread Ti may execute in a single step
- But where is the interaction there?

No Thread Is An Island

- The semantics of each individual thread cannot be considered in isolation.
- The semantics of each thread must consider the interactions it makes with the environment in which it executes
- This can be interactions with memory, with other threads etc
- The semantics of a thread could therefore be defined as a small step operational semantics where the relation also includes an indication of any interaction the thread makes as well as computation steps internal to the thread.

$$T \xrightarrow{\text{action}} T'$$

Here, 'action' refers to whether the thread is sending a signal, receiving a signal or reducing without interacting.

Multiple threads

$$\begin{tikzcd}
T1 \arrow[r, "\text{action-1-1}"] & T1' \arrow[r, "\text{action-1-2}"] & T1'' & \ldots \\
T2 \arrow[r, "\text{action-2-1}"] & T2' \arrow[r, "\text{action-2-2}"] & T2'' & \ldots \\
 & \ldots & & \\
Tn \arrow[r, "\text{action-n-1}"] & Tn' \arrow[r, "\text{action-n-2}"] & Tn'' & \ldots
\end{tikzcd}$$

Where actions may synchronise - i.e. action-i-k and action-j-k may be complementary send and receive signal actions.

Or action-i-k and action-j-k' may be write to memory and read memory actions.

A simple concurrency calculus

- Just like the lambda calculus is a language used for the study of pure functions, we can define small calculi for studying concurrency.
- A particularly well-known example of such a calculus is called CCS, or the Calculus of Communicating Systems
- This was introduced by Robin Milner around 1980 - he was the same chap who developed Type Inference and Polymorphism in ML
- This calculus distills the essence of concurrent threads of computation that may execute both independently and by communication with other threads.
- The semantics of the calculus is given in the style described above by operational semantics with extra annotations indicating the kind of action being performed.
- CCS is one among many many examples of what is called a Process Calculus.

The Syntax of CCS

- Grammar of CCS :
  - P ::= nil I a ! P I a ? P I τ P I P II P I P \ a I X I rec X . P
- nil is an inert process - it has terminated
- a!P sends a signal on a channel named 'a' and then continues execution as P
- a?P receives a signal on a channel named 'a' and then continues execution as P
- τ P takes an internal computation step
- P II P represents two concurrent processes, this models thread spawning also
- P \ a is a restriction operation - no communication on channel 'a' originating inside of P can be seen
- X and rec X . P are used for recursive behaviour e.g. rec X . a! b? X is a process that repeatedly sends a signal on 'a' and receives a signal on 'b'

Example CCS Process

- An example CCS process is given below:

P1	a ! b ? rec X . (z ? nil ll s ? (e ! ll X))
	ll
P2	a? b! s ! s ! s ! z !
	ll
P3	a? b! z !
	ll
P4	rec X . e ? X

Server thread P1 publishes an invitation to client threads P2 and P3 to respond on channel 'b'. The first to do so gets to interact with P1, the other is blocked.

Client thread P2 or P3 calls channel 's' some number of times and calls 'z' to finish. Process P4, counts the number of signals sent on channel 'e'.

Variations of synchronisation

- I won't present the semantics of CCS here but it is typically presented as synchronous communication
- The rule for communication looks like this

$$\frac{P \xrightarrow{a!} P' \qquad Q \xrightarrow{a?} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'}$$

- This says that, if P is ready to send a signal on channel 'a' and Q is ready to receive a signal on channel 'a' then when both P and Q are executing in parallel then they may communicate to produce a computation step internal to P || Q.
- In this case, if P is ready to send a signal it must wait until there is a corresponding process Q to receive the signal.
- This is also what is known as a **handshake** communication as it only involves a single sender and receiver.

Asynchrony in CCS

- It is straightforward to consider an asynchronous variation of CCS
- We simply adjust the grammar to forbid any continuation of the process after a send signal
- That is
  - P ::= nil | **a ! nil** | a ? P | τ P | P || P | P \ a | X | rec X . P
- In this case, a communication would take the form e.g.
  - a ? (b ! nil || P) || a! nil || c ? b ? Q
- In this example, the leftmost process can receive a signal on channel 'a'. It then spawns a new thread that does nothing but send a signal on 'b' and can immediately proceed as P without waiting.
- The rightmost process may eventually receive the signal on channel 'b' if it receives a signal on 'c' first.

Broadcast communication

- Another common communication mechanism that one sees is that of broadcast (as opposed to handshake) communication.
- In that model, a signal is sent to all concurrent processes that are ready to receive that signal, not just a single process.
- The process calculus CSP - Communicating Sequential Processes developed by Tony Hoare (of Quicksort fame) uses such a mechanism.
- This calculus is similar to CSP except for the broadcast communication and a denotationally presented semantics rather than the small step operational semantics of CCS.

Non-determinism

- You will notice from the previous example that running the overall system does not have a determined behaviour.
- The number of signals ultimately sent on channel 'e' depends on whether P2 or P3 first responds to the initial signal.
- In practice this choice is made by operator system scheduling
- In this calculus, either interaction can be taken
- What is clear is that the semantics of a process does not take the form of a single sequence of labelled reductions

$$\begin{tikzcd} T \arrow[r, "\text{act1}"] & T' \arrow[r, "\text{act2}"] & T'' \arrow[r, "\text{act3}"] & T''' & \cdots & {} \arrow[r, "\text{act-n}"] & T'''''' \end{tikzcd}$$

- Rather, the semantics of a process forms a **tree** of labelled reductions as choices are made in the overall system behaviour.
- Versions of CCS sometimes also include an explicit choice operation - P + Q that means, behave like P next or behave like Q next.

Complex Semantics

- Therefore, when it comes to understanding a concurrent program we must consider each of its threads and the semantics of each thread can be represented as a tree of behaviours.
- When considering whether an implementation of an particular thread meets its semantics we must have a means of considering
  - the actual behaviour of a thread (as a tree of actions) and
  - the specified semantics (as a tree of actions)
- We need a way of comparing trees of actions
- This is the topic of the next few lectures. The particular forms of trees we use are actually represented as graphs of labelled actions and are referred to as **labelled transition systems.**
- The notions of equality between concurrent systems is a rich field and forms the basis of many reasoning and verification techniques for concurrency.

Next Lecture

Labelled Transition Systems