Concurrent Systems

$$3D4 \longleftrightarrow CS2016$$

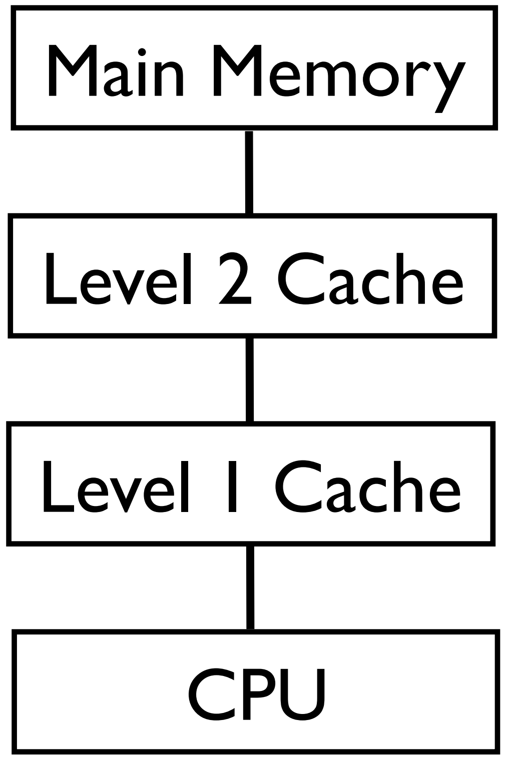
Operating Systems

Andrew Butterfield
ORI.G39, Andrew.Butterfield@scss.tcd.ie

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

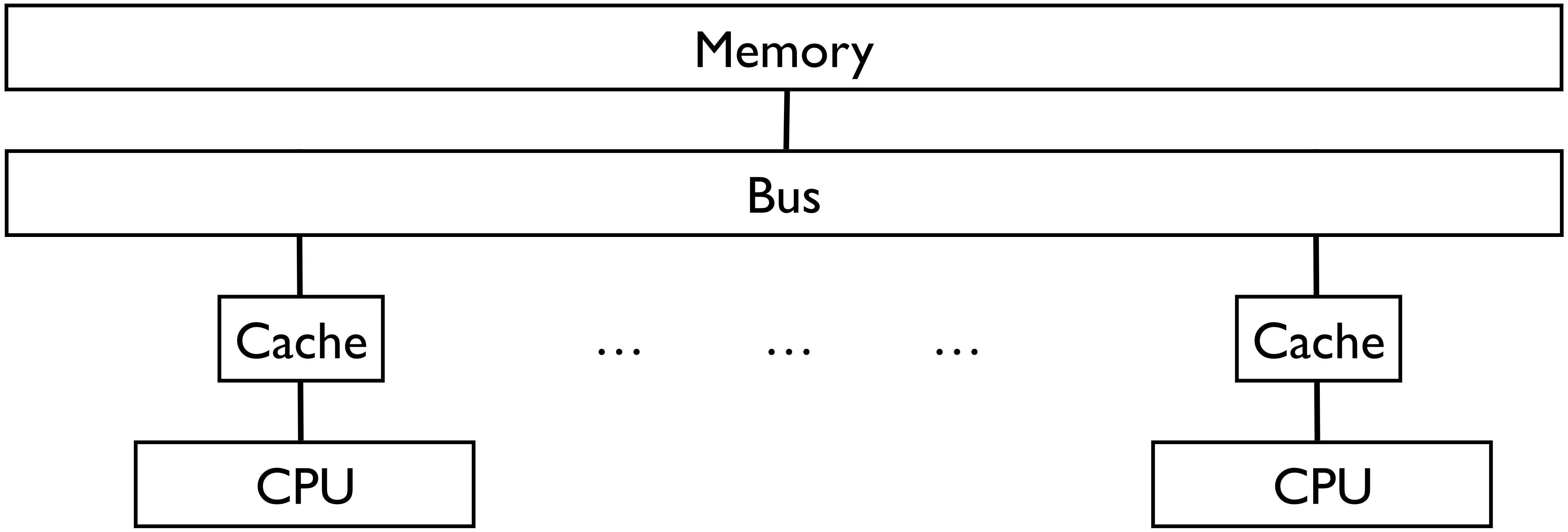
with thanks to Mike Brady

Hardware



Single Processor

Hardware



Multiple Processor

Shared Memory Machine

- Each processor has equal access to each main memory location – Uniform Memory Access (UMA).
  - Opposite: Non-Uniform Memory Access (NUMA)
- Each processor may have its own cache, or may share caches with others.
  - May have problems with cache issues, e.g. consistency, line-sharing, etc.

Sequential Program

- A *Sequential Program* has one site of program execution. At any time, there is only one site in the program where execution is under way.
- The *Context* of a sequential program is the set of all variables, processor registers (including hidden ones that can affect the program), stack values and, of course, the code, assumed to be fixed, that are current for the single site of execution.
- The combination of the context and the flow of program execution is often called [informally] a *thread*.

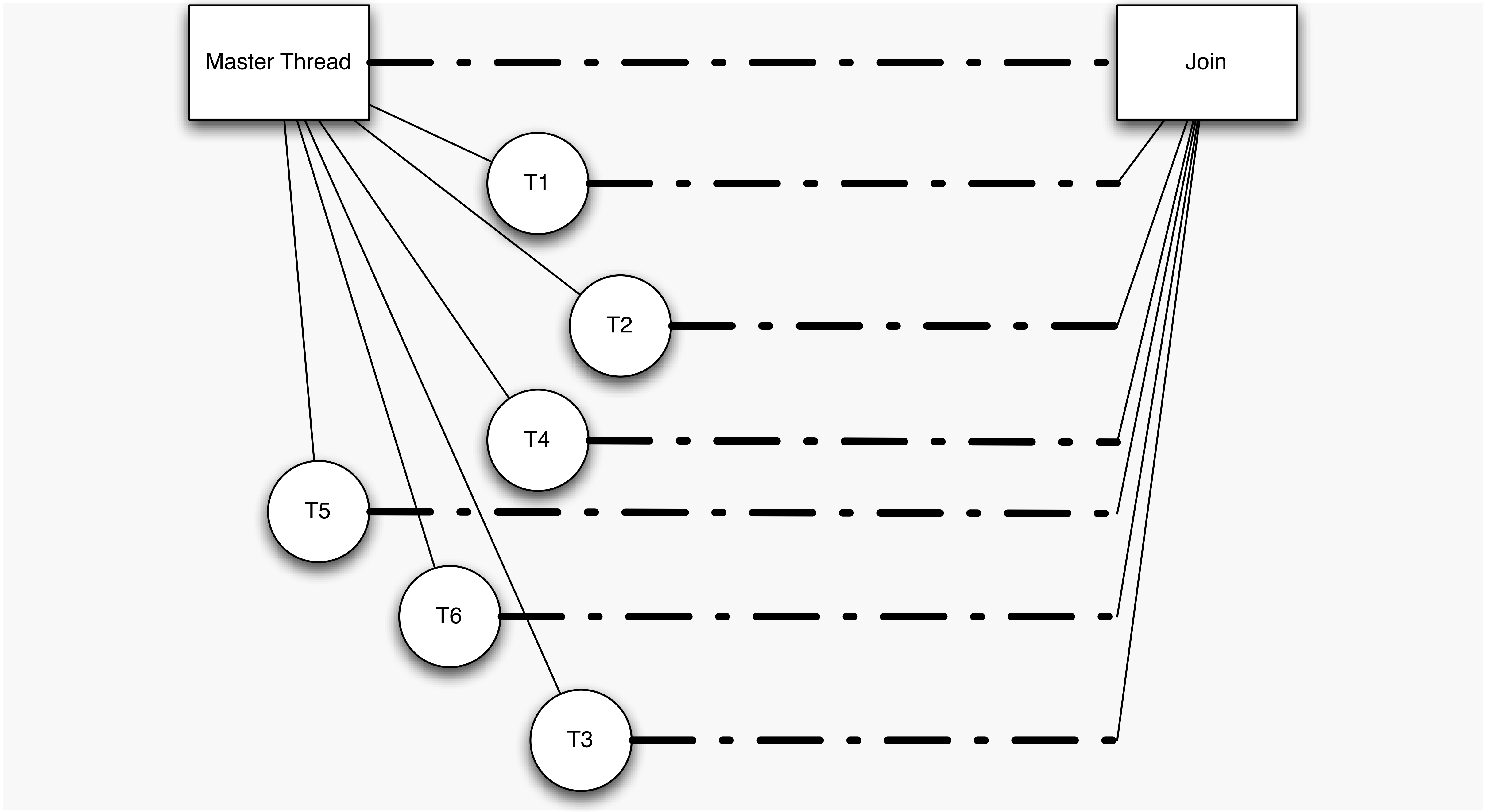
Concurrent Program

- A *Concurrent Program* has more than one site of execution. That is, if you took a snapshot of a concurrent program, you could understand it by examining the state of execution in a number of different sites in the program.
- Each site of execution has its own context—registers, stack values, etc.
- Thus, a concurrent program has *multiple threads* of execution.

Parallel Program

- A *Parallel Program*, like a concurrent program, has multiple threads of program execution.
- The key difference between *concurrent* and *parallel* is
  - In concurrent programs, only one execution agent is assumed to be active. Thus, while there are many execution sites, only one will be active at any time.
  - In parallel programs, multiple execution agents are assumed to be active simultaneously, so many execution sites will be active at any time.

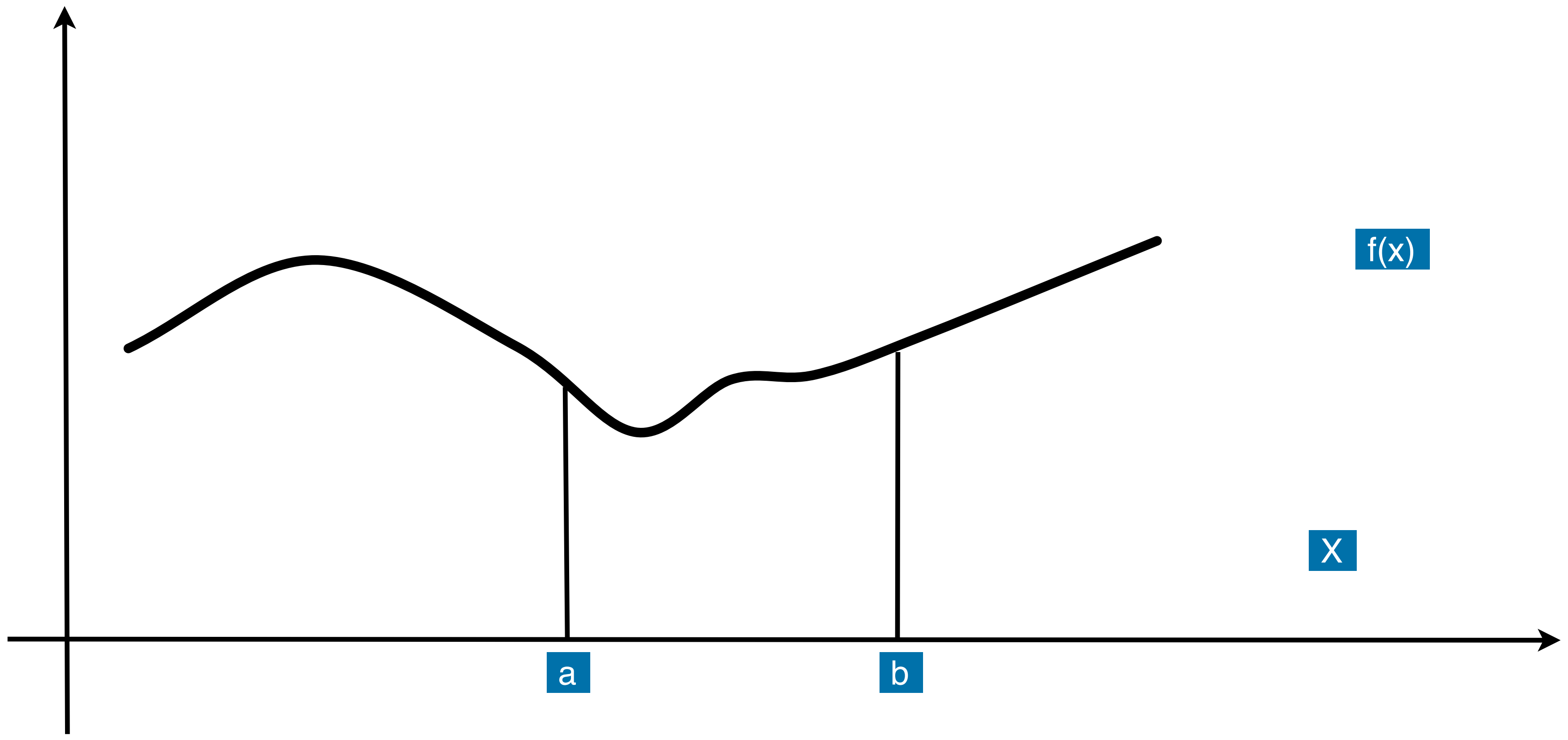
What's ~~wrong~~ deceptive about this?



“Massively” Parallel Problems

- Almost no interaction between parallel threads calculating independent parts of the solution.

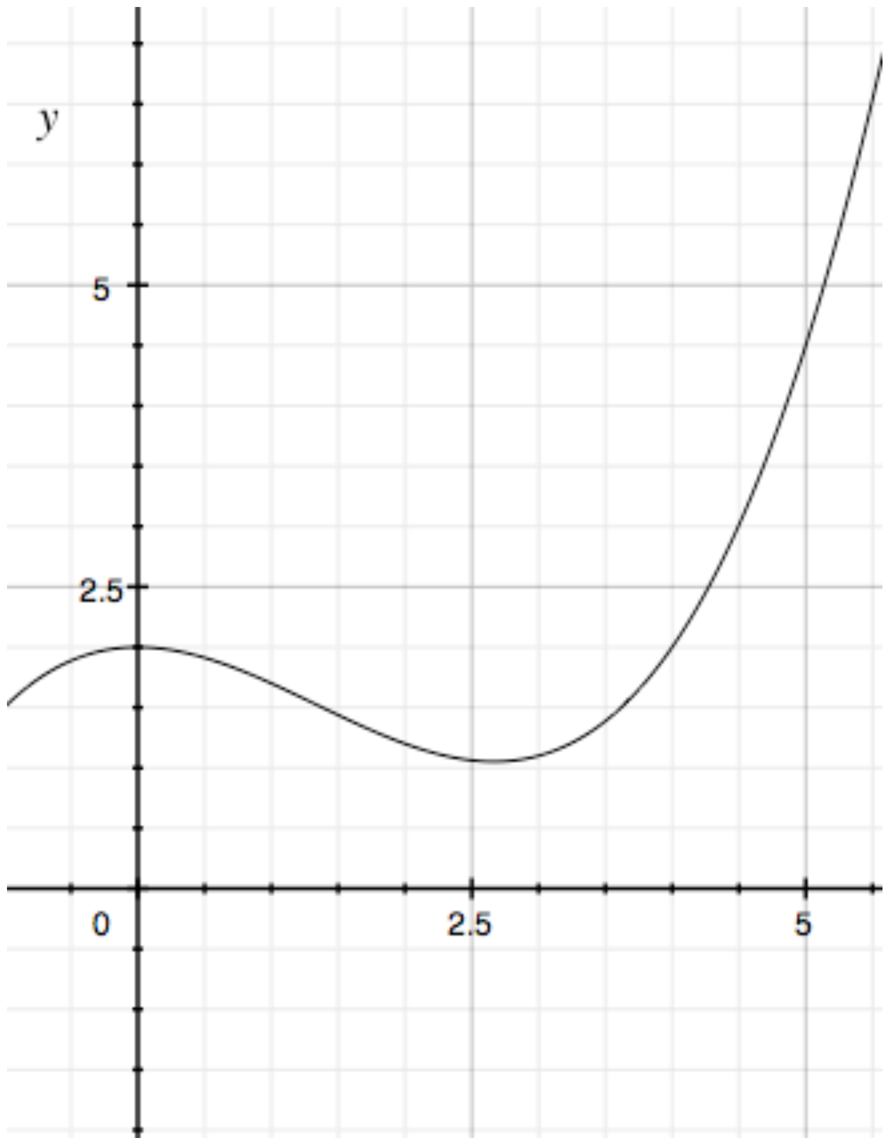
Example: Numerical Integration



Numerically integrate from a to b

Example

- Sample function: $y = 0.1x^3-0.4x^2+2$



Interaction

- No thread interaction shown.
- Most of the issues to do with threads (really, with any kind of parallel processing) arise over unforeseen interaction sequences.
    - For example, if two threads attempt to increment the same variable

Unsafe Interaction Example

Thread 1	Thread 2	G
L=G (4)	–	4
L=L+10 (14)	–	4
	L=G (4)	4
G=L	L++ (5)	14
	G=L	5

G = Global Variable; L = Separate Local Variables

Interactions & Dependencies

- Interactions occur because of *dependencies*.
- Interactions can be made safe using a variety of techniques.
    - E.g. semaphores, condition variables, etc.
        - We will look at them…
        - They tend to be expensive; sometimes *very* expensive.
- But, sometimes we can redesign the code to avoid dependencies.
- One of the biggest themes in this kind of programming is *dependency minimisation*.

We need to use our knowledge of…

- Sequential imperative programming, in C/C++ on a single processor.
- How programs could communicate.

…to do Parallel Programming

- High performance parallel programming
- We will confine ourselves to shared-memory machines.
- We will use dual-core (maybe quad-core) machines for practice.

Why is it this difficult?

- It's not clear whether it *really* is more difficult to think about using multiple agents, e.g. multiple processors, to solve a problem:
    - Maybe we are conditioned into thinking about just one agent;
    - Maybe it's natural;
    - Maybe it's our notations and toolsets;
    - Of course, maybe it really is trickier.