Considerations in Parallel Algorithm Design

Louis J. M. Aslett

i-like Reading Group

10th February 2014

Goal of this talk

Provide a little insight into what considerations there are in parallelising algorithms beyond the trivial ‘100% independent tasks’ scenario.

Actual code & specific technology details will be ignored today but happy to discuss!

Overview

I. Computer architecture background

II. Parallel programming background

III. Parallel programming design with toy statistical examples

IV. Final comments

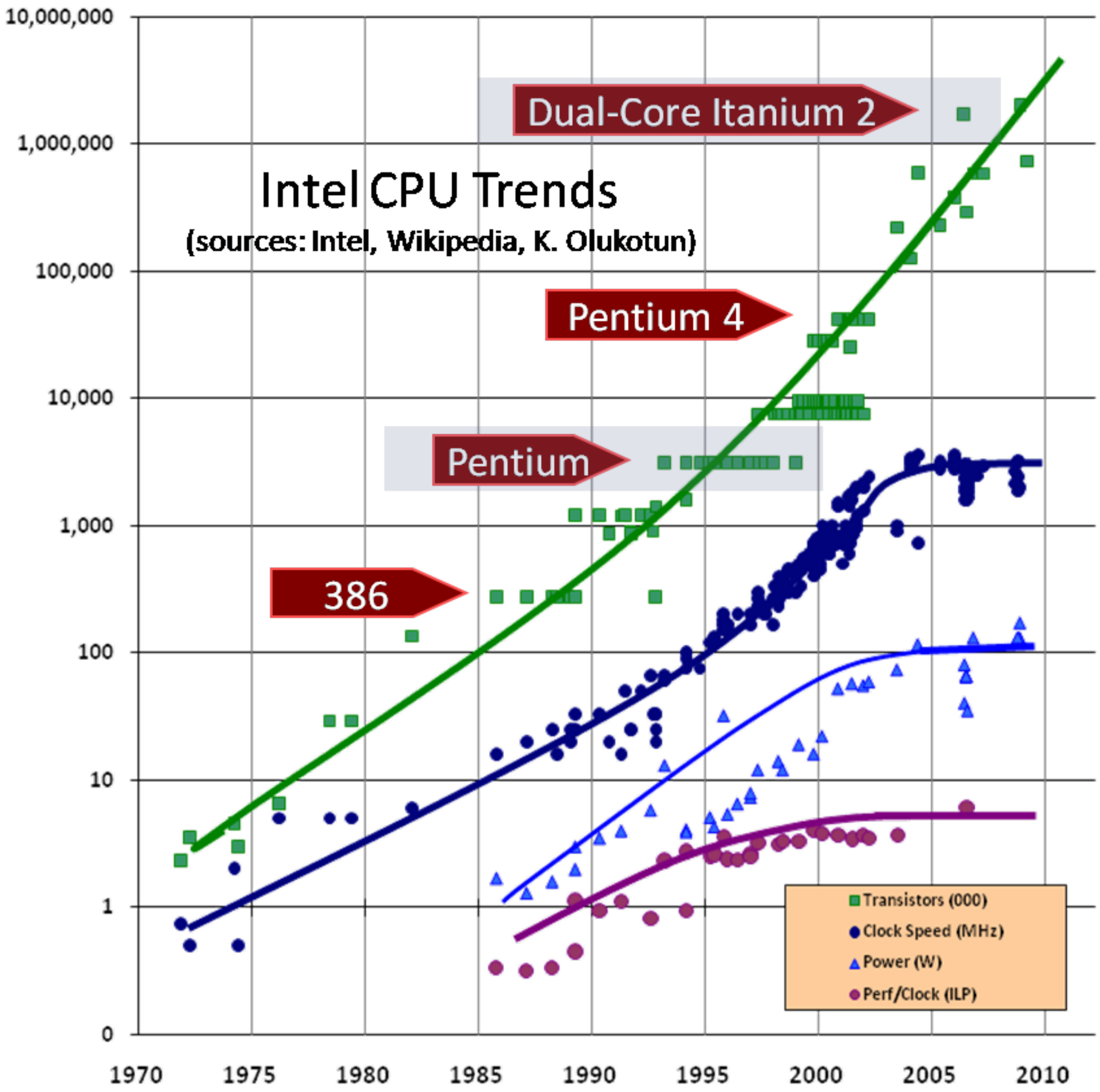
I.

Computer architecture background

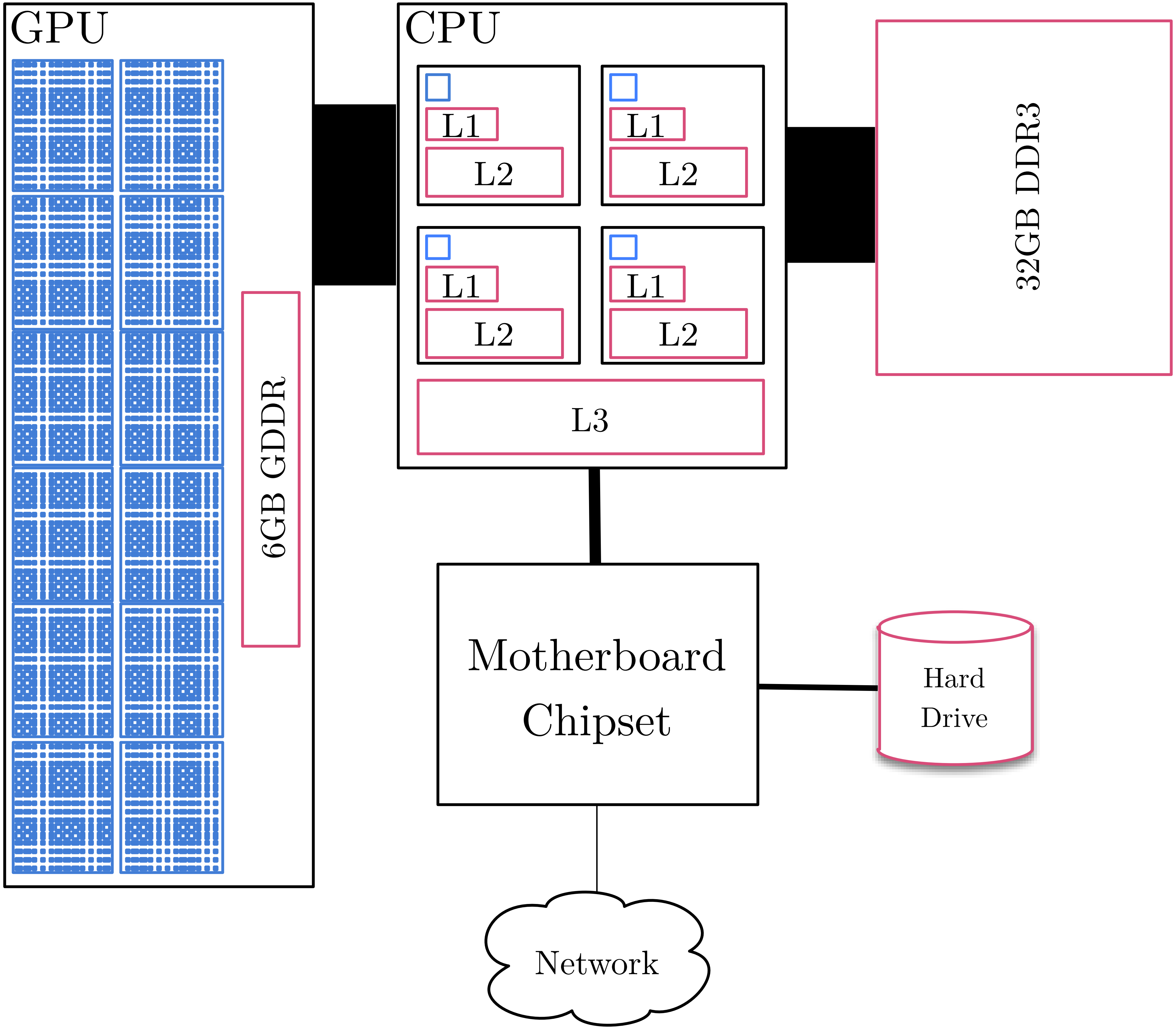
Background Reading

- ‘The free lunch is over: A fundamental turn toward concurrency in software’ http://www.gotw.ca/publications/concurrency-ddj.htm

- ‘Welcome to the jungle’ http://herbsutter.com/welcome-to-the-jungle/



‘The free lunch is over’ — Herb Sutter



Simplified computer architecture

Memory Access	Size	Latency
Registers (per core)	168 physical 16 named (x86-64)	0 clocks
L1 Cache (per core)	0.03MB	~ 4 clocks
L2 Cache (per core)	0.25MB	~ 12 clocks
L3 Cache (shared)	2 - 8 MB	~ 36 clocks
Main memory	up to 32,768MB	~ 212 clocks
Hard drive	Terabytes	can be $>10^6$ clocks

Cache line: 64 bytes

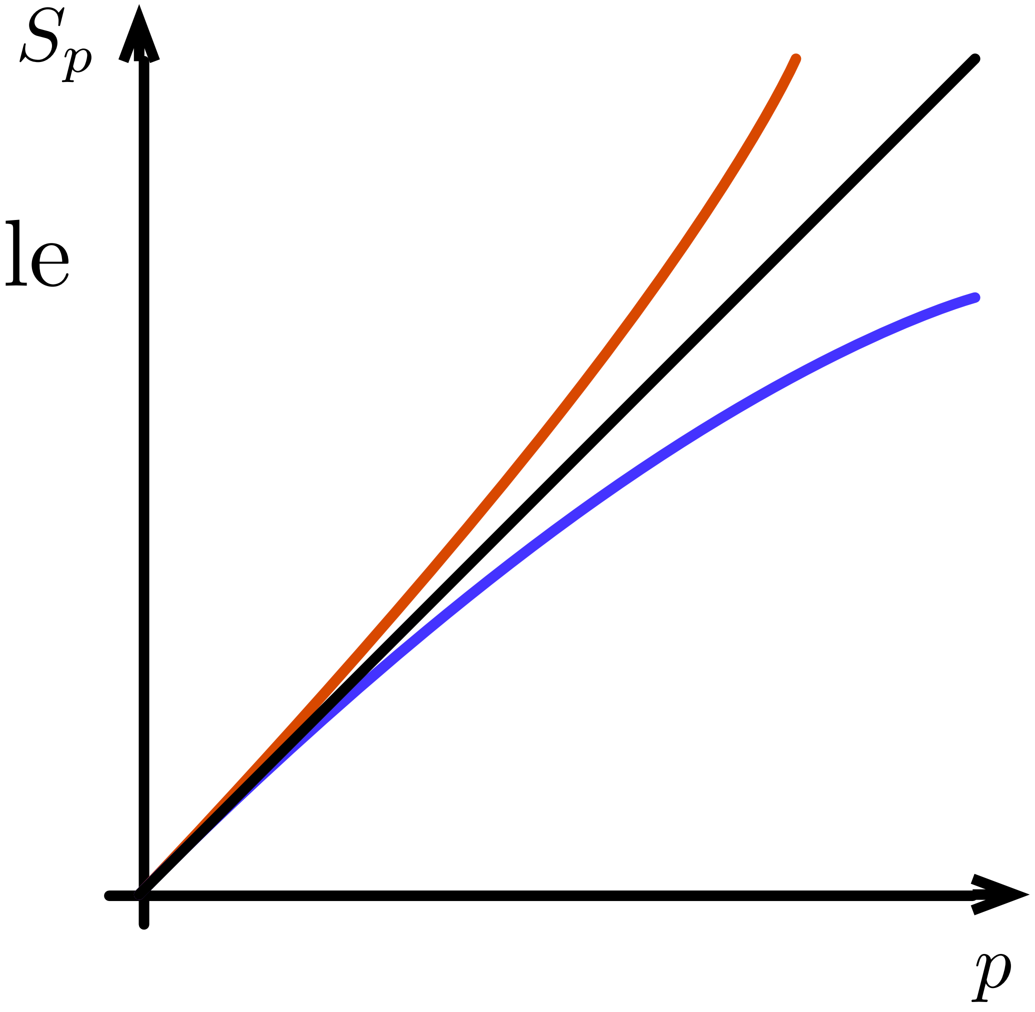
Approximations based on Intel Haswell

II.

Parallel programming background

Parallel speedup

$$S_p = \frac{T_s}{T_p}$$



Superlinear speedup not impossible in embarrassingly parallel setting due to memory access.

Sublinear speedup most common.

Linear should be the goal.

See Amdahl's Law and Gustafson's Law.

Types of parallelism

	Single Instruction	Multiple Instruction
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

SISD: no parallelism

MISD: not a common setting

SIMD: classic GPU setting

MIMD: classic CPU setting

Some common tools

A. GPUs

B. CPUs

C. Clusters

A. GPUs in a nutshell

Extraordinarily parallel devices (upto 2688 cores at present).

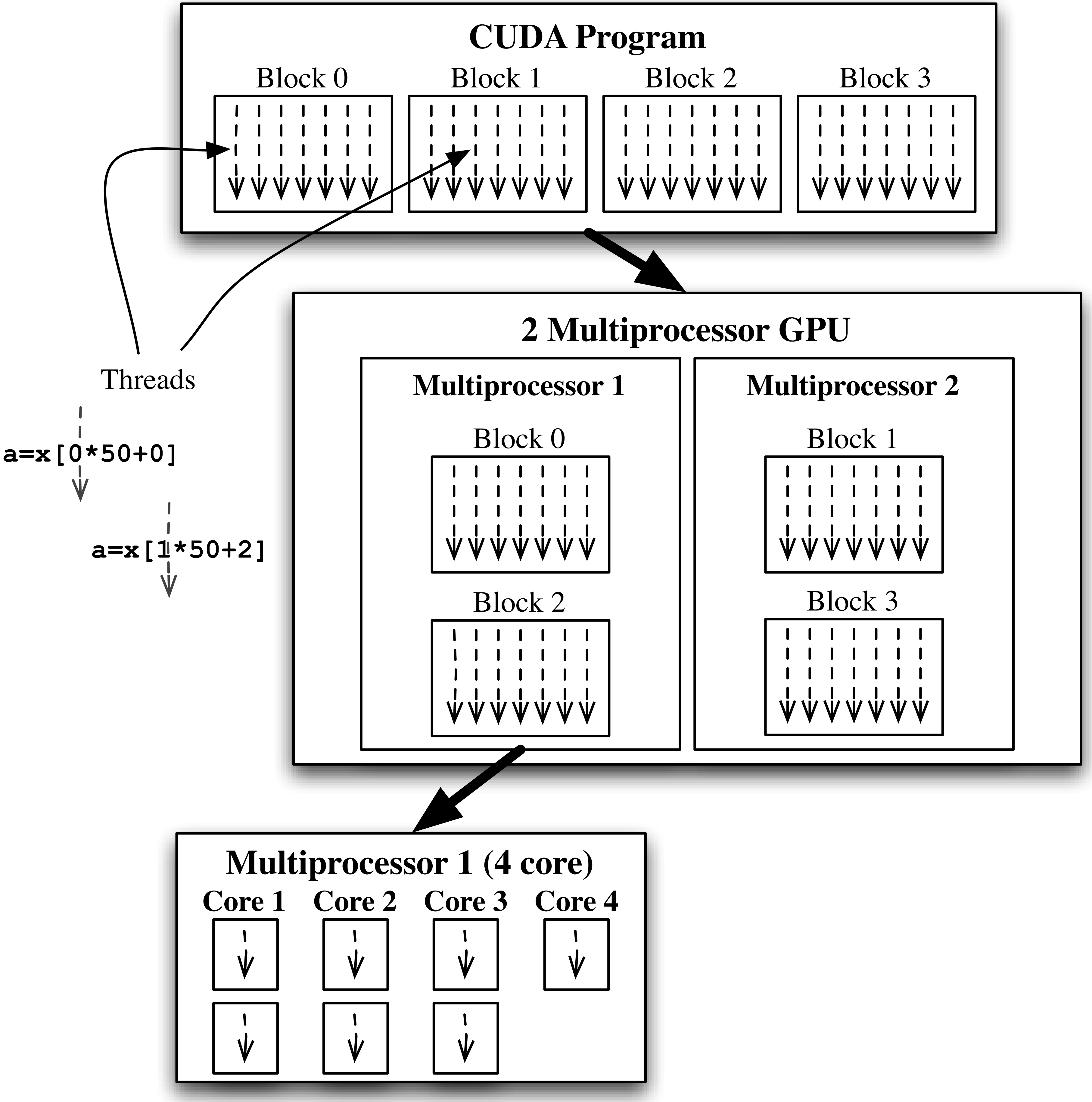
Single instruction multiple data (threads) is the *only* mode of operation.

Note that GPUs cannot access the system memory: any data must be copied to, and any results from, the GPU. This can be costly for large data sets.

A mental model for GPUs

- ***Kernel***: a C function which is flagged to be run on a GPU.
- A kernel is executed on the core of a multiprocessor inside a ***thread***. A thread can be thought of as just an index $j \in \mathbb{N}$
- At any given time, a block of threads is executed on a multiprocessor. A block can be thought of as just an index $i \in \mathbb{N}$. Very loosely: an index of multiprocessors in devices.
- Together (i, j) corresponds to exactly one kernel running on a core of a single multiprocessor.

Very simplistically speaking, think of how to parallelise your problem by how to split it into identical chunks indexed by a pair $(i, j) \in \mathbb{N} \times \mathbb{N}$


CUDA Program
Block 0
Block 1
Block 2
Block 3
Threads
a=x[0*50+0]
a=x[1*50+2]
2 Multiprocessor GPU
Multiprocessor 1
Block 0
Block 2
Multiprocessor 2
Block 1
Block 3
Multiprocessor 1 (4 core)
Core 1
Core 2
Core 3
Core 4

3 golden GPU concepts

i) Memory accesses are *slow* compared to the cores. Always have many more total threads than cores to mask this.

ii) Conditional sections of an algorithm can quickly kill performance.

iii) Random or disorganised memory accesses will make a GPU under perform a CPU!

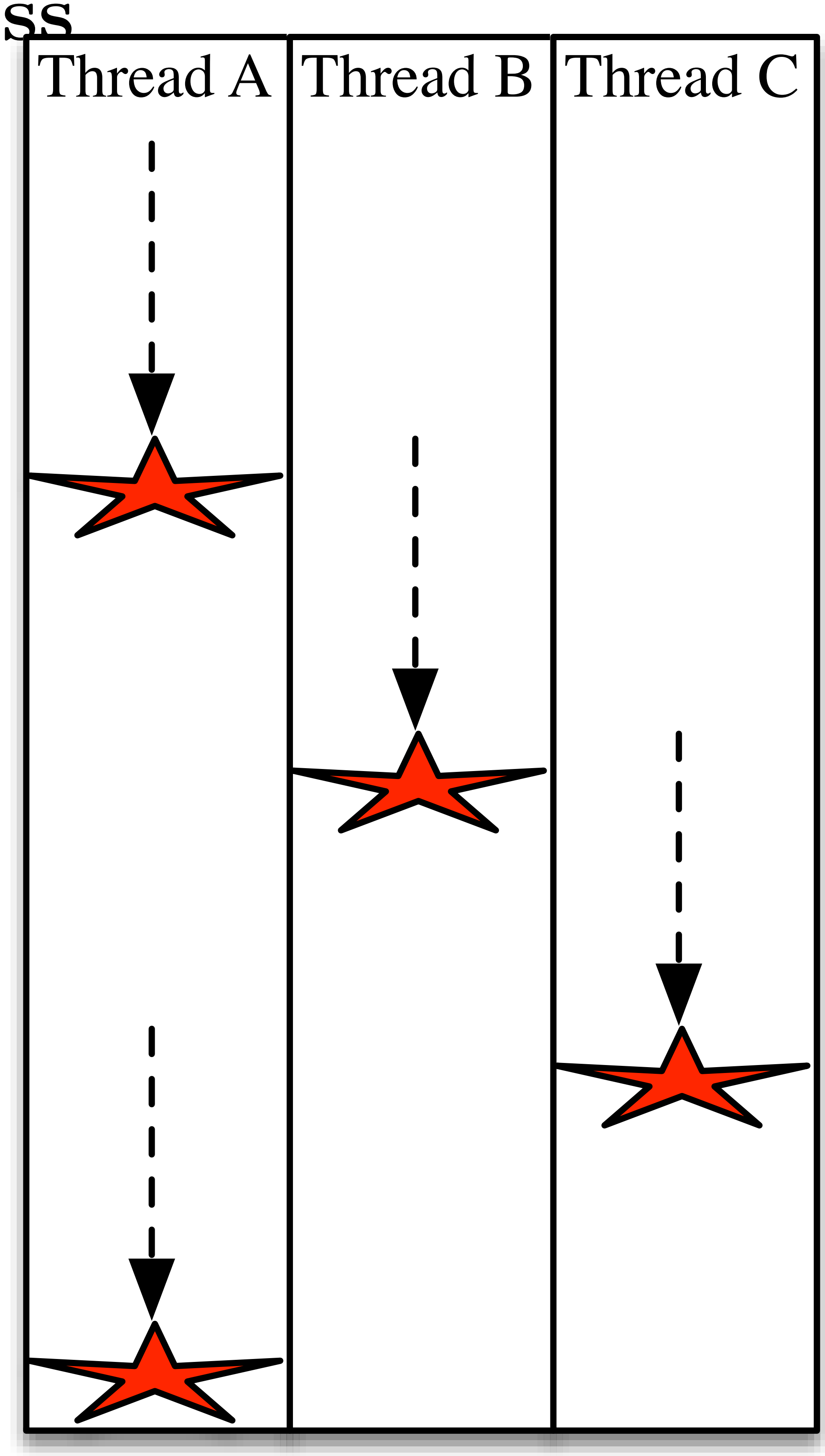
GPU i) Slow memory access

★ = Global memory access
=> Execution stall!

Global memory accesses take 400-600 cycles, so a core will stall when a request is made.

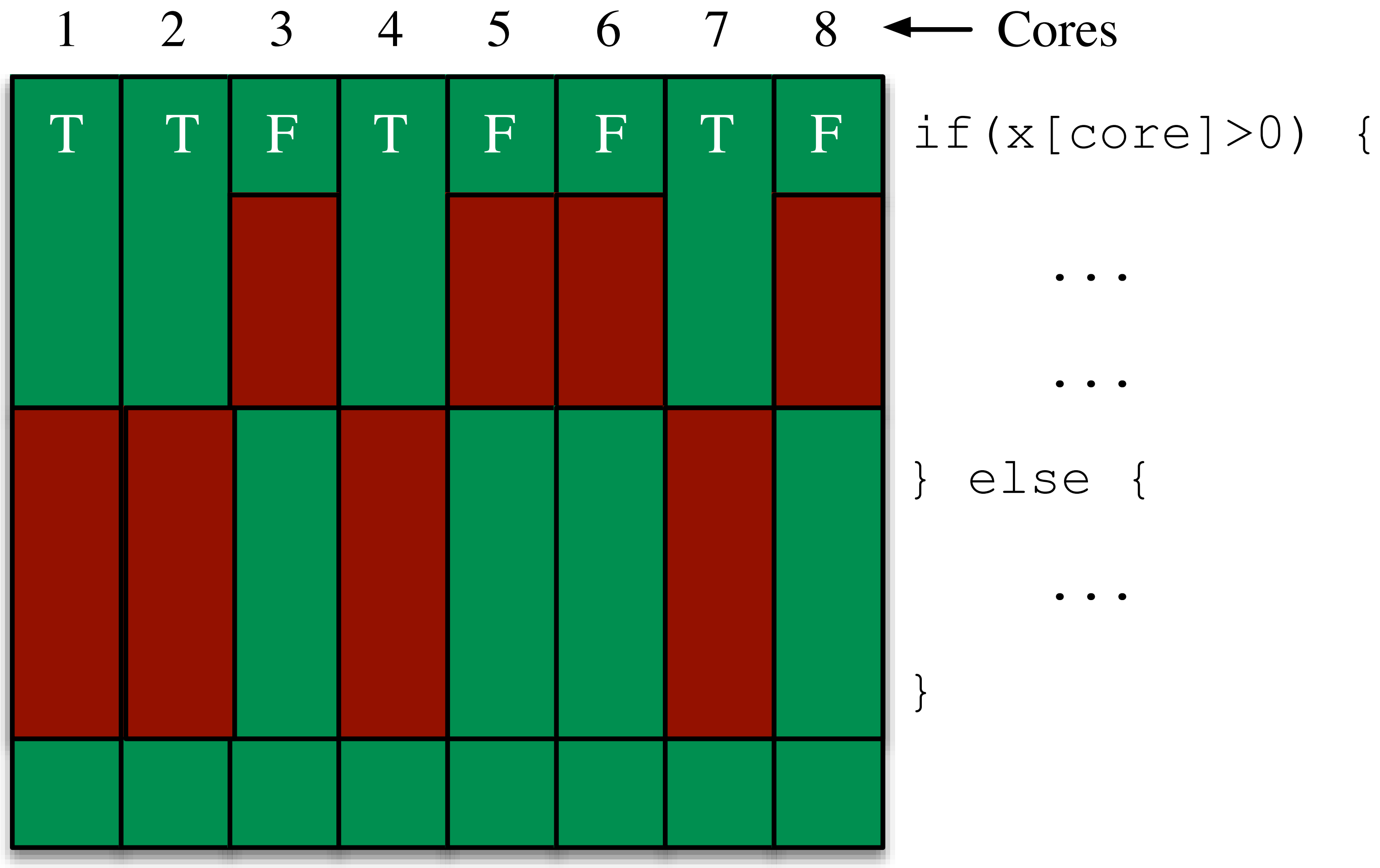
But, if # threads > # cores then **CUDA will interleave another thread** and run until the memory request is fulfilled and the first thread can run again.

Also, consider carefully **store -vs- recompute**. Might be quicker to recompute simple values than incur memory accesses!



GPU ii) Conditional execution

The exact same code is run on multiple items of data, so **conditional statements can kill performance.** The total run time is the sum of all branch run times.



```
if(x[core]>0) {

     ...

     ...
} else {

     ...

}
```

Unless you know in advance the conditional result.

GPU iii) Coalesced memory access

When a floating point number is requested from memory, that number *and* the following 3 are loaded (128-bit memory bus).

Thus, if consecutive threads require consecutive regions of memory, there are a quarter the number of memory transactions required.

If an algorithm requires random or disorganised memory access then this can reduce performance at least 4 fold compared to the intended GPU programming model.

B. CPU parallelism

CPUs are not nearly as parallel as GPUs, but have certain advantages, including not being limited to single instruction multiple data algorithms.

Often there is ‘free’ parallelism you never even see happening.

The tools are very easy. If you know C already, OpenMP can be learned in a day.

CPUs cores much more powerful than GPU cores.

CPU: Low level 'freebies'

Some things come 'for free' on the CPU:

- It will (usually) execute serial code where there is no dependency out-of-order automatically.

- Good compilers will often identify places where CPU vector arithmetic can be used (MMX/SSE/AVX).

- For smaller data problems, you can forget about memory accesses due to automatic caching.

- Integer arithmetic is *fast* compared to GPU.

CPU: How parallel to go?

Unlike the GPU, because of caching you will most often want to match the level of parallelism to the number of physical cores (but profile to be sure!)

Context switching is moving from one thread of execution to another and is *expensive* on a CPU (v fast on GPU). It can also destroy caching efficiency.

Care required not to overload the CPU with threads.

CPU: Biggest factor

The biggest factor in parallel performance using a tool such as OpenMP is shared memory access.

If more than one thread of execution needs to access the same memory location to update a result, then there is expensive coordination of cores involved.

Because CPUs tend to have less parallelism can try to design around this.

CPU: Random number generation

As of Ivy Bridge, Intel CPUs include ‘true’ hardware random number generation. If the algorithm involves heavy generation of random numbers this could outperform a pseudo random number generator.

Upto 500MB of random data per second (vs 3.5MB per second using GSL).

C. Cluster parallelism

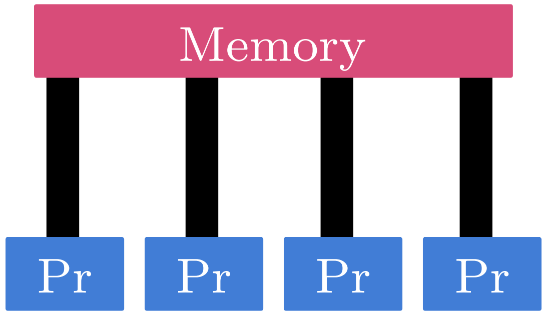
The step up in complexity for parallelism over a cluster is potentially significant.

No longer a shared memory system: no unified view of the data set visible to everyone unless it is copied to every machine. Changes in one machine not automatically visible to others.

Network communication is the slowest possible link!

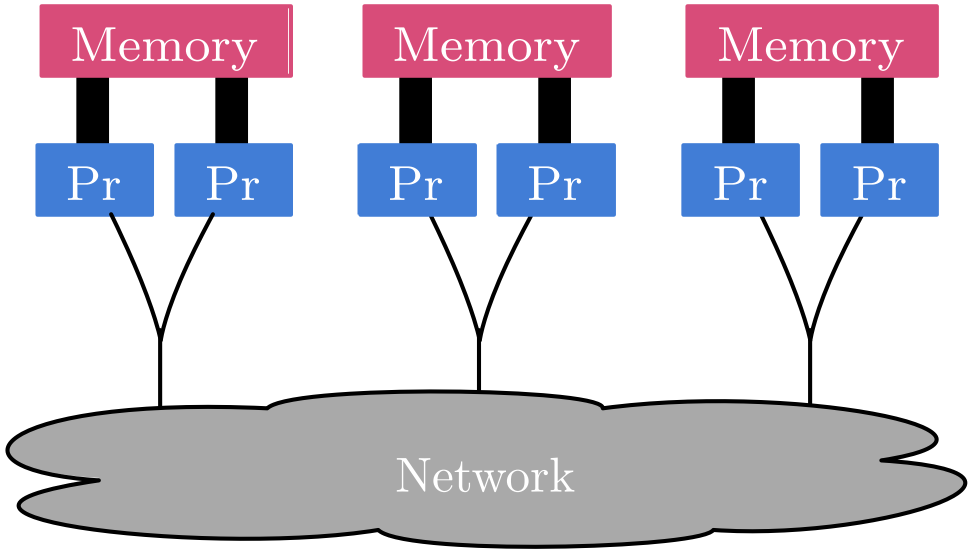
Clusters: The complexity of memory

CPU/GPU model



‘shared memory’

Cluster model



‘distributed memory’

- Might be the only choice if data set too large for one computer.
- Adds significant complexity.

Single common enemy

- Access of shared memory.
- Strategies are slightly different to deal with this on CPU, GPU and cluster.
- Doing additional (modest) computational work may be preferable to sharing memory if the choice exists.

III.

Parallel programming design with toy statistical examples

Common strategy

A. Partition

B. Communication

C. Agglomeration

D. Mapping

See ‘Designing and Building Parallel Programs’ by Ian Foster. Old but still relevant.

A. Partition

- Divide into small pieces both the *computation* related with the algorithm and the *data* on which the computation takes place.

- **Domain decomposition:** decompose data first, then computation.

- **Functional decomposition:** decompose computation first, then data.

- Just this step required => ‘100% independent tasks’

Partition: Objectives

- At least an order of magnitude more parts of the partition than available cores.
- Minimise redundant computation/storage.
- Roughly equal sized parts.
- Partition scales up as problem size increases.

Partition: Toy example 1 — KDE

$$\hat{f}_h(x) = \frac{1}{nh} \sum_{i=1}^{n} K\left(\frac{x - x_i}{h}\right)$$

Domain decomposition:

Compute K on full grid for each data point in parallel.

Good when data size large, grid coarse.

Functional decomposition:

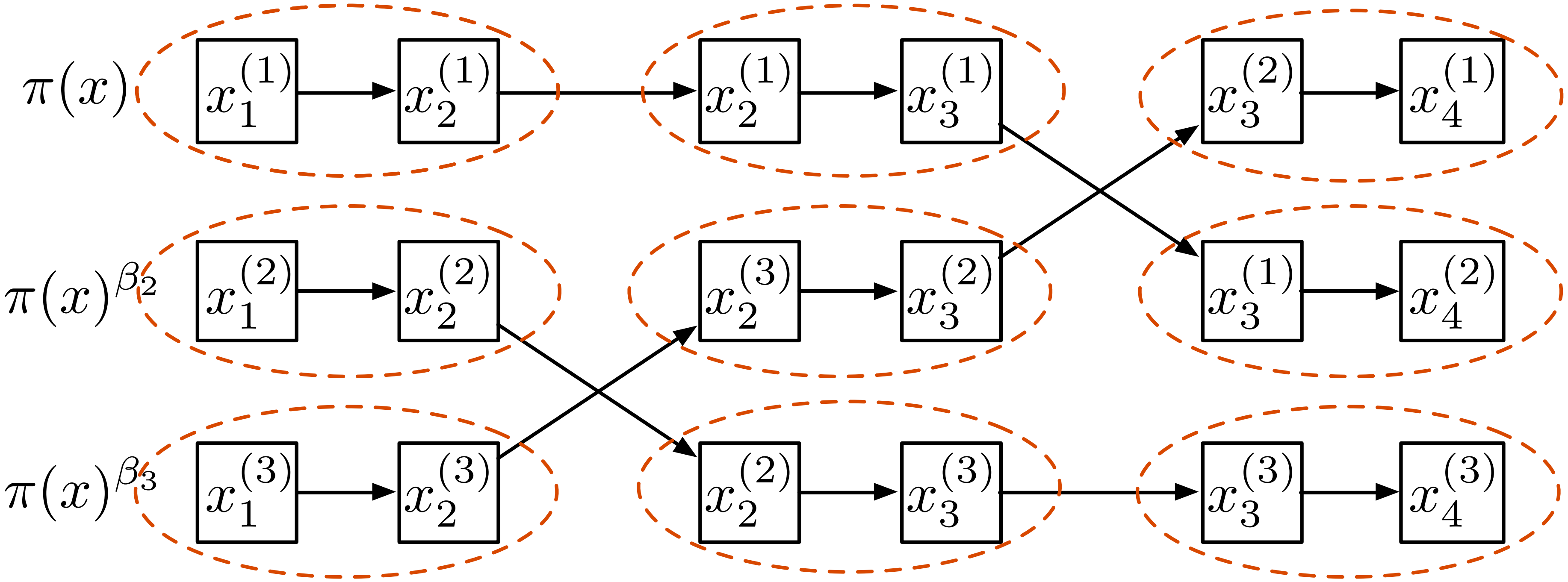
Compute estimate $\hat{f}_h(\cdot)$ for all data points parallelising over the grid.

Good when data size small, grid fine.

Partition: Toy example 2 — Parallel Tempering

$$p(\mathbf{x}_1) = \prod_{i=1}^{m} p_{\beta_i}(x_1^{(i)}) \text{ where } p_\beta(x) = \pi(x)^\beta$$

For a collection of RWMH chains with uniform swap proposals, the natural decomposition is **functional**: each RWMH should be performed in parallel.



Partition: Toy example 3 — Gibbs IGMRF

Model for each pixel of an image defined intrinsically, dependent only on four nearest neighbours (mean is sample mean of the four neighbouring pixels)

$$x_i \mid x_{-i} \sim \mathrm{N}\left(\sum_{j \in \mathcal{N}(i)} x_j / |\mathcal{N}(i)|, \mathbb{I} \right)$$

Gibbs sampling this non-stationary, unconditioned GMRF is then straight-forward.

Clear functional parallelism sampling a full sweep over the image.

B. Communication

- Partitions are planned to execute in parallel but cannot, in general, execute completely independently.

- Computation in one task requires data associated with another task => communication between tasks.

Communication: The challenge

- No communication => ‘embarrassingly parallel’
- Communication means that issues around memory efficiency (CPU/GPU) and communication (cluster) come to the forefront.
- In highly non-local or asynchronous settings, communication can end up dominating computation.

Communication: Types

- Local -vs- global
  Local task just needs data from ‘neighbours’. Nice for a GPU and cluster.
  Global will have high communication with ‘distant’ data. Might be ok for CPU if cached.

- Structured -vs- unstructured
  Determines ability to target a particular method.

- Synchronous -vs asynchronous
  Asynchronous means point of communication unknown and one task must request data from another. (Uncommon in stats?)

Communication: Objectives

- Roughly equal communication for all tasks.
- Small amounts of interaction with neighbours.
- Computation able to proceed concurrently (else waiting on previous results).
- Communication able to proceed concurrently (synchronised).

Communication: Toy example 1, KDE

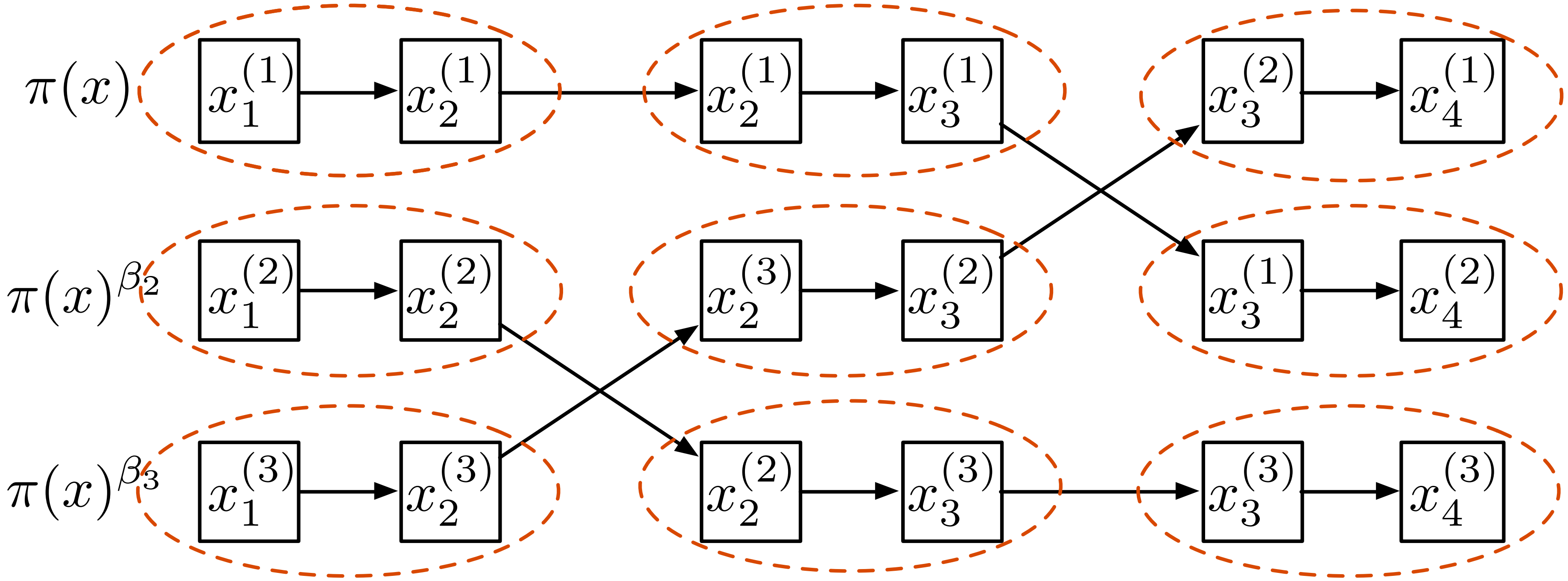
With the KDE domain decomposition, there is a potential communication issue for large grid problems.

If grid too large to store n copies to later sum, then memory storing result for the grid must be updated by every task.

i.e. There will be n tasks wanting to add their contribution to the memory locations holding the grid values.

=> if data can be held on single machine, might prefer functional decomposition here.

Communication: Toy example 2 — Parallel Tempering



Zero communication in the RWMH sections.

Local but random memory accesses in the swap section. If more than one swap this section is potentially highly serial.

Redesignable to enable continued parallel execution?

Communication: Toy example 3 — Gibbs IGMRF

Memory accesses requires real care when computing on GPU or cluster.

GPU: Boundary conditions hinder SIMD. Hard to coalesce memory accesses. Still faster than CPU.

Cluster: if different blocks of pixels on different machines then asynchronous pixel requests involved.

	$x_{i(j-1)}$	
$x_{(i-1)j}$	x_{ij}	$x_{(i+1)j}$
	$x_{i(j+1)}$	

C. Agglomeration

- Now start to think about the target technology (CPU/GPU/cluster).
- Combine tasks into a single thread to get the right balance of concurrency and communication.
- i.e. broadly speaking: CPU will want heavy agglomeration, GPU will want light agglomeration (as long as communication under control).

Agglomeration step: Objectives

- To reduce communication.
- To identify places where duplication of computation may be preferable to communication or storage.
- To identify data which can perhaps be replicated at small cost to reduce communication.
- This can be highly problem specific: *auto-tuning strongly recommended where possible!*

Agglomeration: Toy examples

Implicitly agglomerated already for speed of presentation!

Each of the examples would have been fully partitioned, whereas we only did first level domain/ functional partition as it was appropriate here.

e.g. Could further parallelise KDE sum for massive scalability on very large clusters and huge data.

D. Mapping

- How do the agglomerated tasks map to the technology.
- Not relevant to CPU.
- GPU => block/thread division
- Cluster => Careful distribution because no shared memory.

II.

Final comments & odds and ends

Parallel libraries

Often problems can be expressed in a way that allows use of already optimised general purpose parallel libraries

- cuBLAS + Magma
- scaLAPACK
- Thrust
- MapReduce/Hadoop
- Storm

e.g. Silverman (1982): KDE using FFT

Mining computer science parallel algorithms

CS has a head start researching this for decades (the Cray-1 in 1976 was a vector machine!) Some algorithms will map to existing solutions.

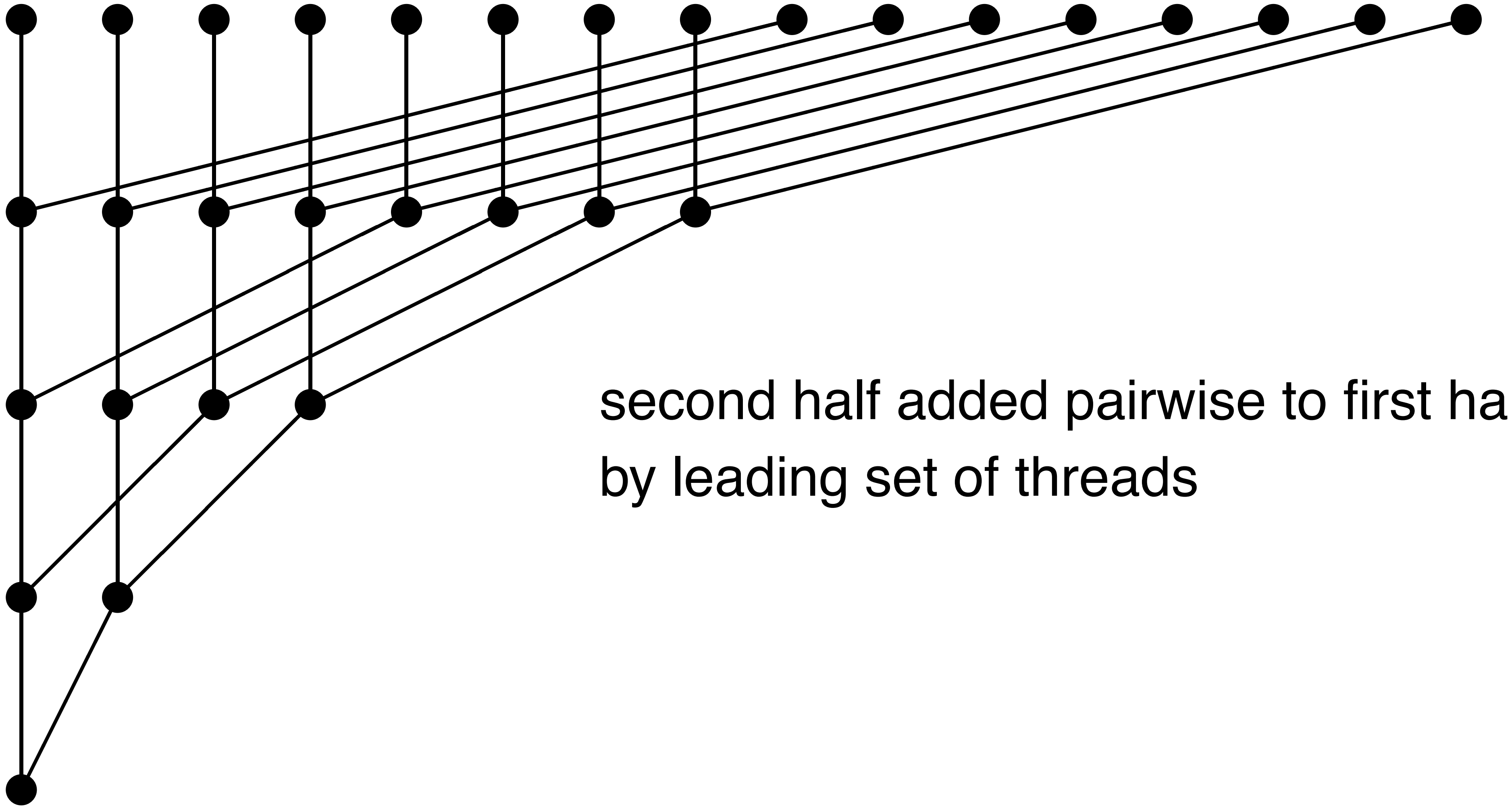
For example,

Definition: An operator, $\circ$, is a reduction operator if it is commutative and associative.

$$x \circ y = y \circ x \quad , \quad x \circ (y \circ z) = (x \circ y) \circ z$$

If your algorithm is a reduction operator then there are established parallel techniques. Moreover, different techniques optimised for GPU/cluster/...

Reduction example (Mike Giles)

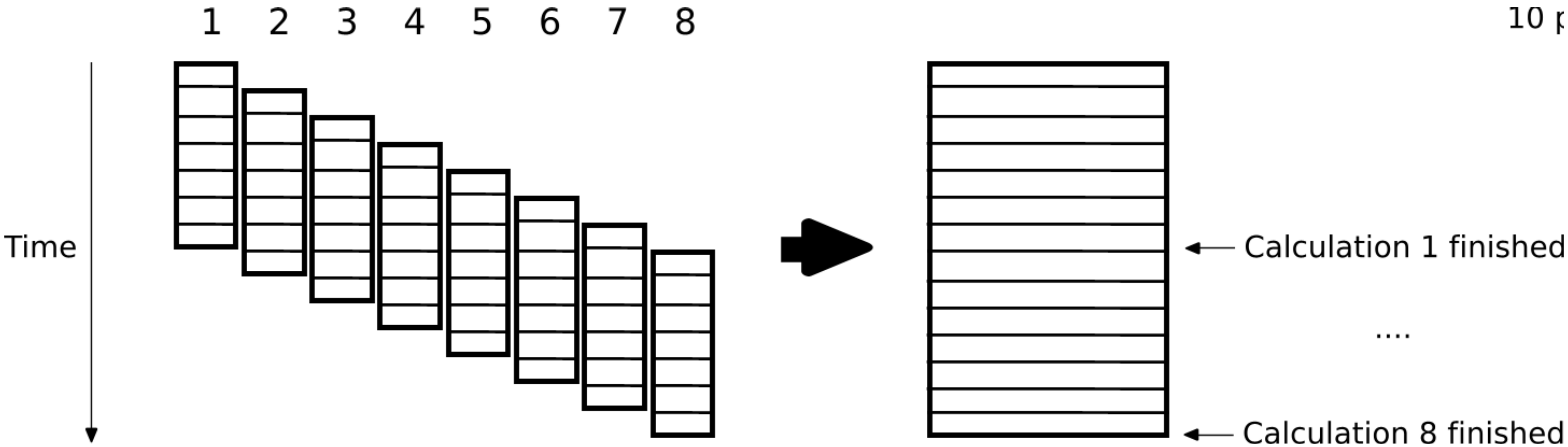


second half added pairwise to first half
by leading set of threads

‘Pseudo-parallel’: Pipelining

For sequential/streaming problems. Say 1 data point takes t seconds to compute.

If you can decompose the sequential algorithm then you can pipeline so that total execution time for n data items is less than nt.


1
2
3
4
5
6
7
8
10 p
Time
Calculation 1 finished
....
Calculation 8 finished