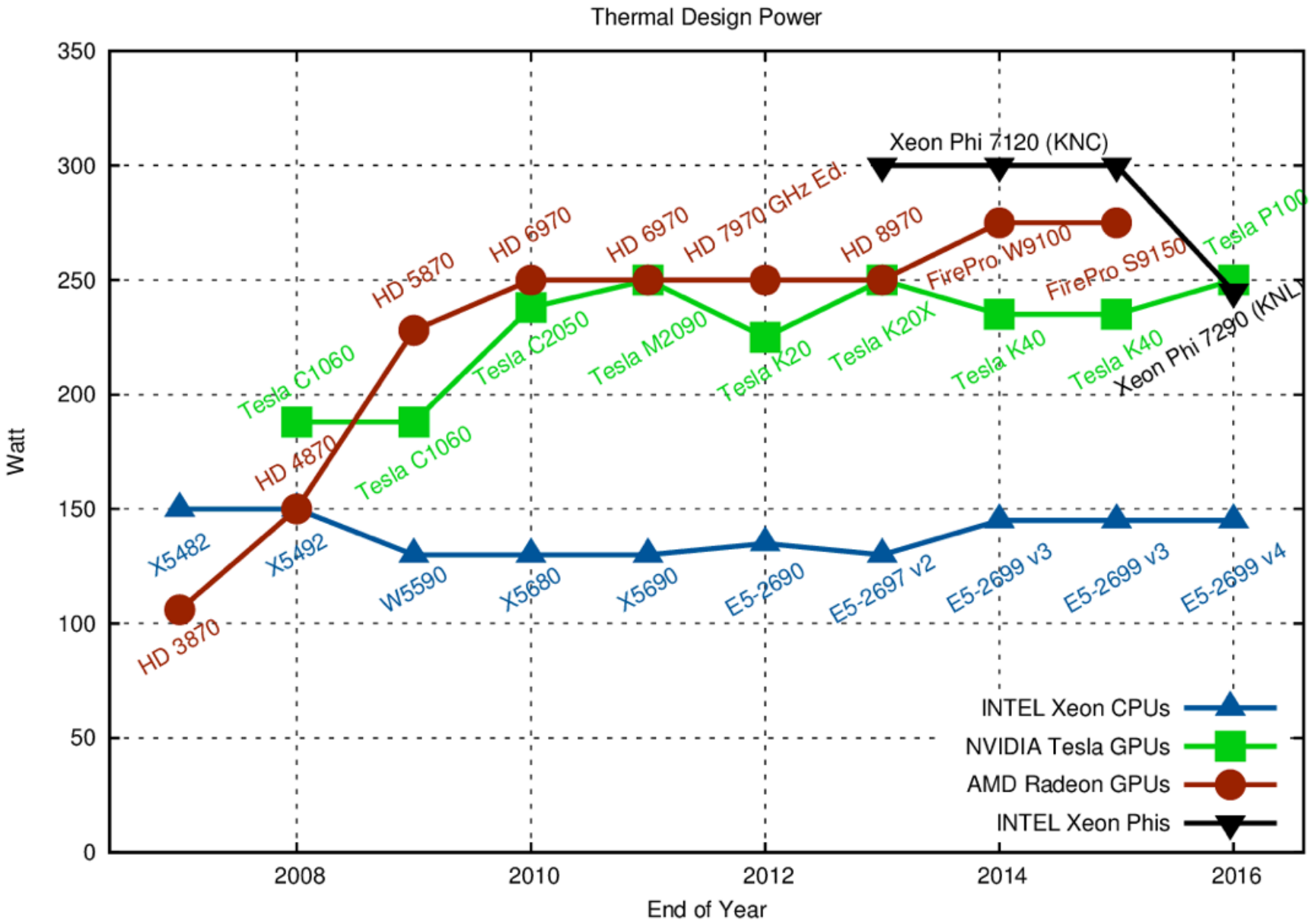
DIT 873 & DAT346
Techniques for Large-Scale Data

Lecture 3

Recap

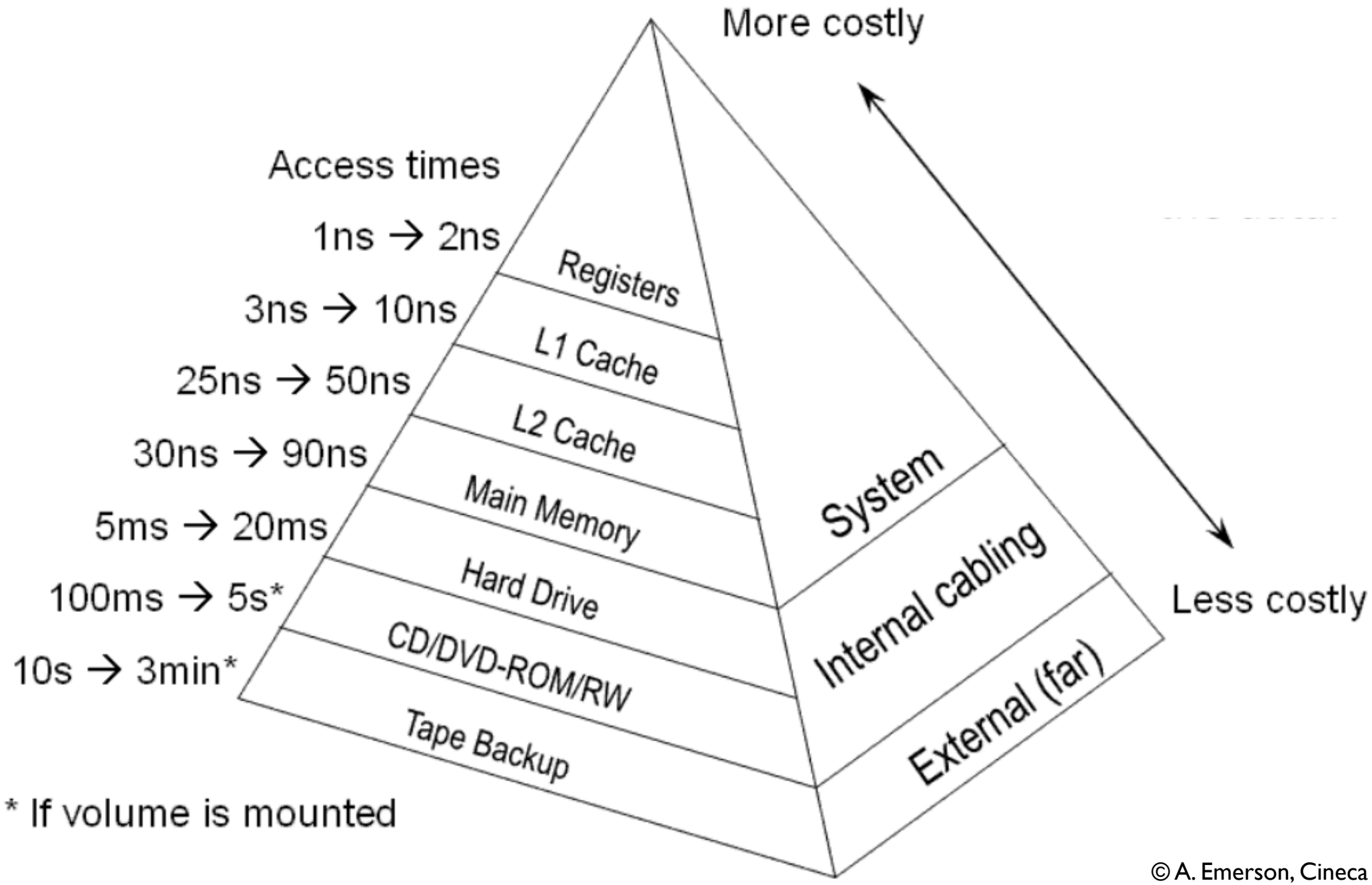


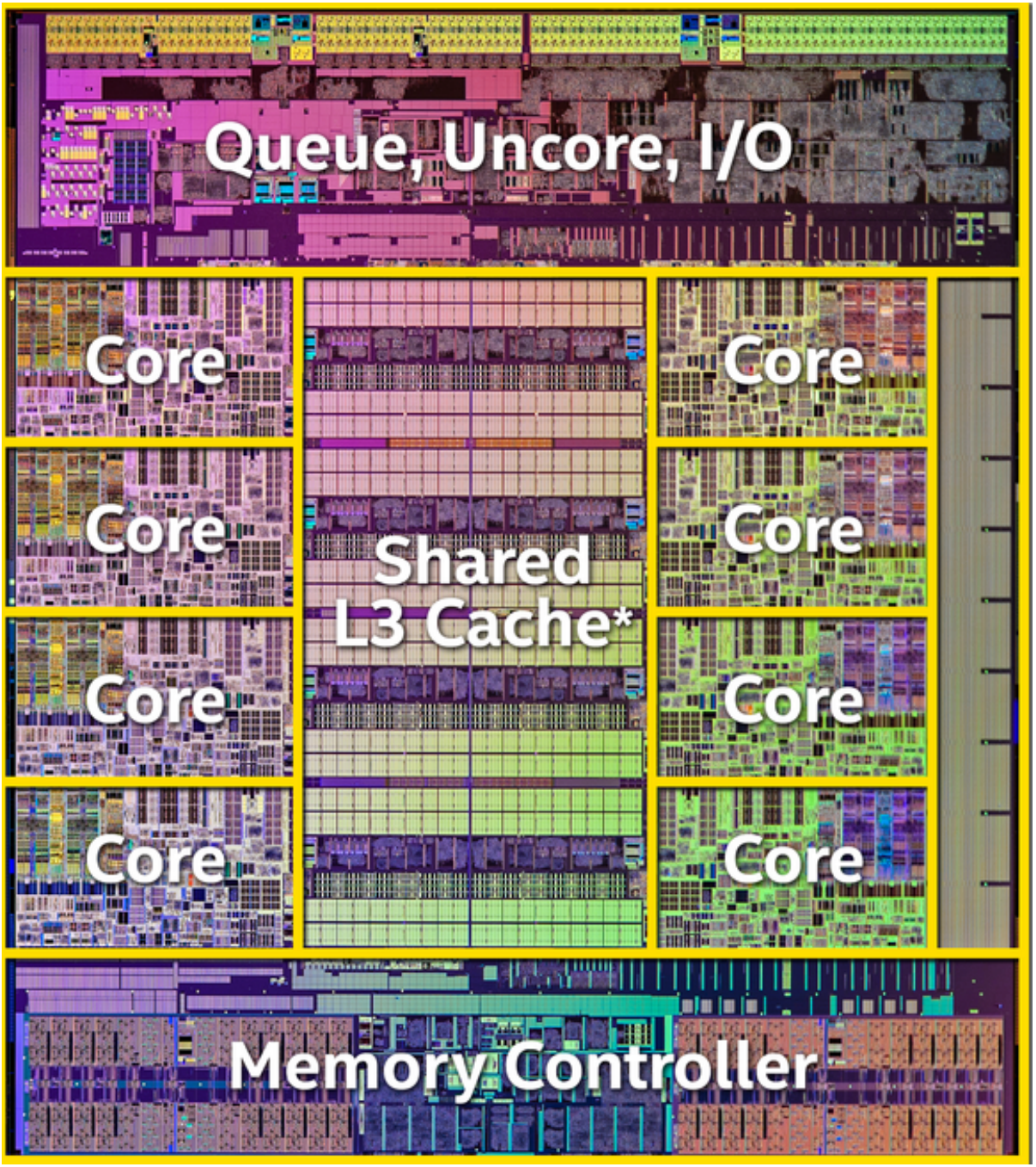
- Power consumption approx. cubic in clock rate
- Bottleneck: Cooling

See Sec. 2.1 Rauber & Rünger (2013)

https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/#comment-364

Memory Hierarchy

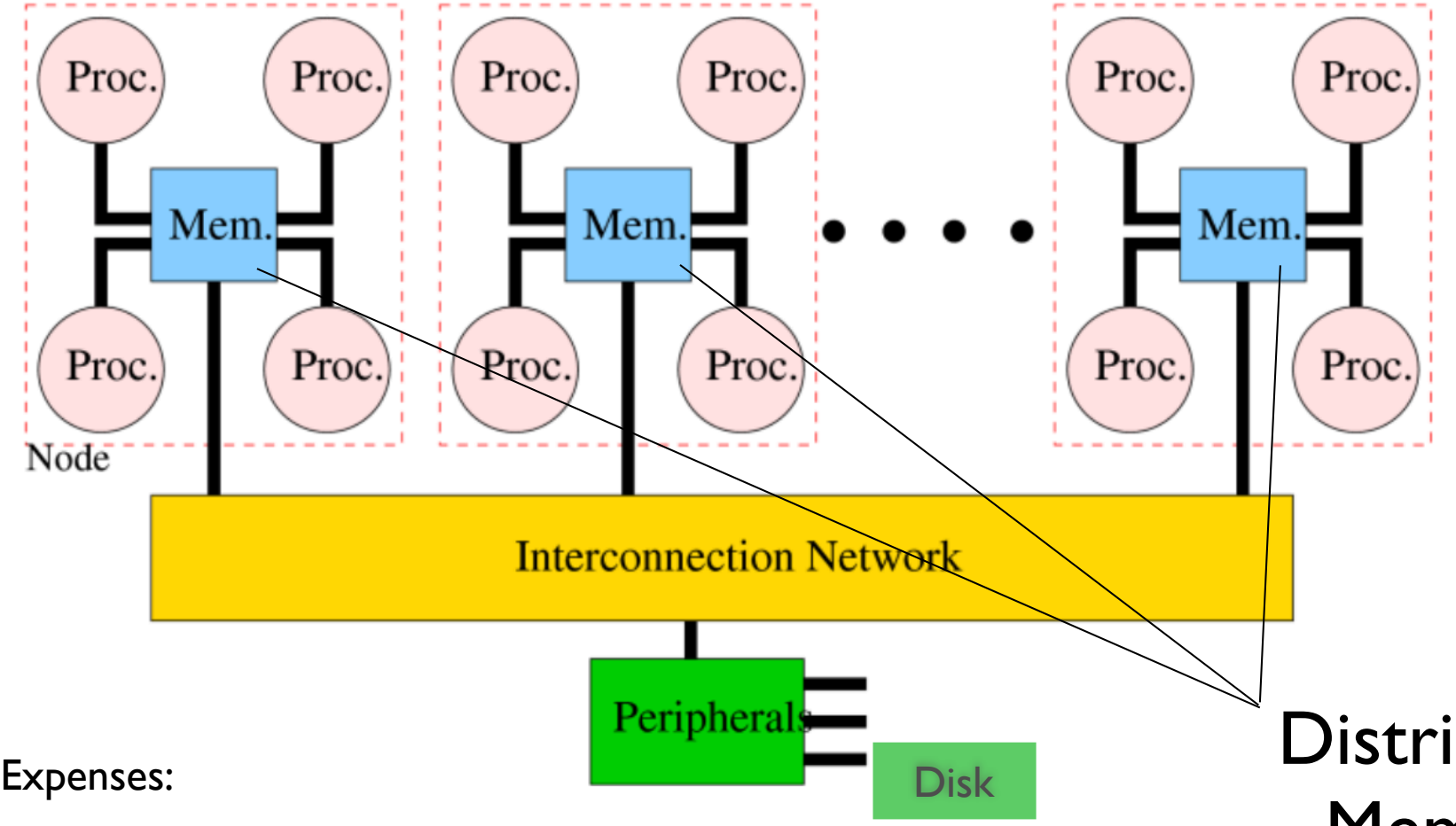




Intel Core i7

See also Sec. 2.4.5 Rauber & Rünger (2013)

Classical HPC Compute Cluster



- Expenses:
  - "Large" SMP computers expensive (bus systems, motherboards)
  - Interconnects like Infiniband, Myrinet, ...
  - Less effective economy of scale

Performance comparison: HPC vs Commodity at 2M SEK

- If one were to build a 2M SEK Beowulf cluster of commodity PCs
  - How many PCs?
  - How many cores (of 2.2GHz or better) in total?
  - How much total memory (2666MHz DDR4 or better)?
  - How much total disk space?

- Let's ignore networking for now

Please look at PC vendors and give your informed estimates for the questions above. Please share via Chat with link to individual PC

Image & Text: Silicon Mechanics, Inc.

- If one were to build a 2M SEK Beowulf cluster of commodity PCs
  - How many PCs?

    Average 191

  - How many cores (of 2.2GHz or better) in total?

    Average 1162 (10x)

  - How much total memory (2666MHz DDR4 or better)?

    Average 3361 GB (~55%)

  - How much total disk space?

    Average 293 TB (5x)

Best system substantially more powerful than HPC server: 175 PCs, 750 (~6.7x) cores, 11200 GB (~2x) RAM and 2000 TB (~34x) disk

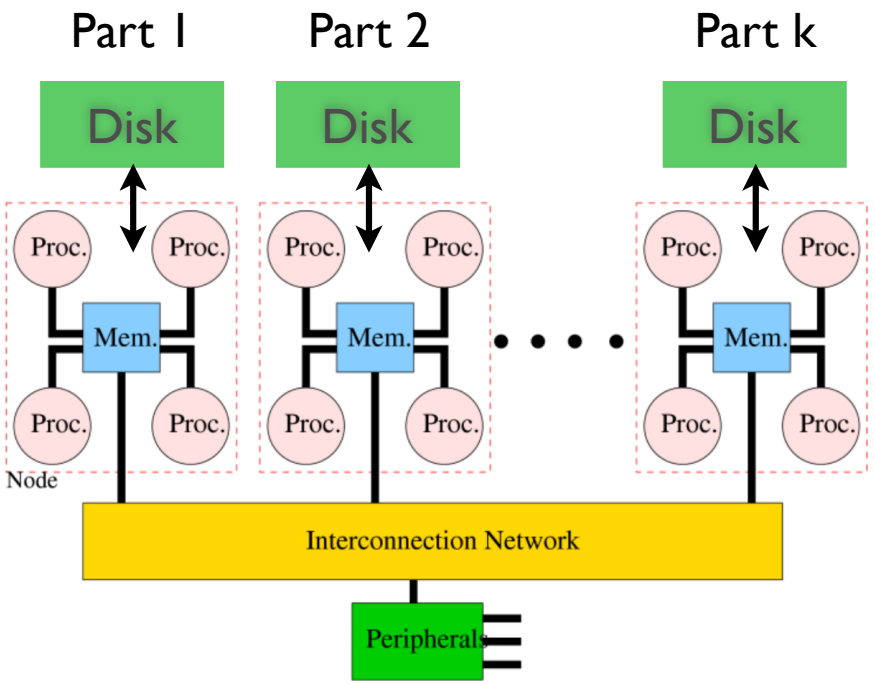
Continue …

Workload Comparison

	HPC	Data Science
CPU	Lots of CPU Floating Point (FP) Little memory	Medium Mixed integer/FP Large memory
Data/File IO	Often very little IO	Massive IO Computations IO-bound
Tasks	Standard Tools (e.g. matrix solvers)	Custom tools and frequently new, changing problems

Data-intensive Compute Cluster

Assuming analysis can be performed on separate subsets of data:



- Clusters allow to keep more data in main memory: 32 SMP nodes with 32GB RAM hold 1 TB RAM
- Increased File IO throughput if disks are locally attached to nodes (speedup ~ # nodes)

Designing large data software

Consequences of memory hierarchy

Analysis of Algorithms

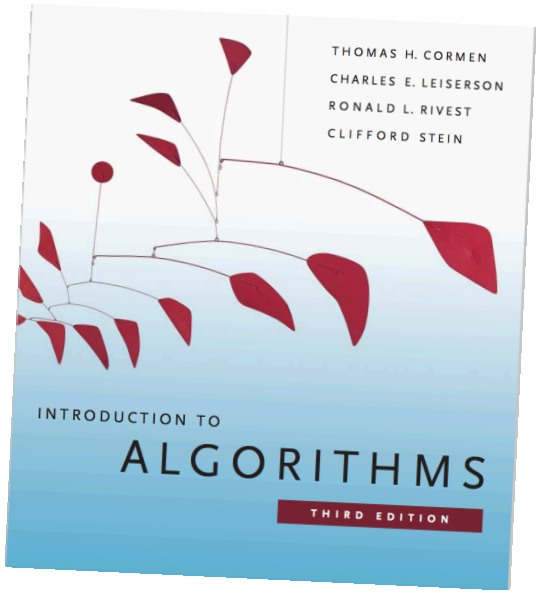
2.2 Analyzing algorithms

Analyzing an algorithm has come to mean predicting the resources that the algorithm requires. Occasionally, resources such as memory, communication bandwidth, or computer hardware are of primary concern, but most often it is computational time that we want to measure. Generally, by analyzing several candidate algorithms for a problem, we can identify a most efficient one. Such analysis may indicate more than one viable candidate, but we can often discard several inferior algorithms in the process.

Before we can analyze an algorithm, we must have a model of the implementation technology that we will use, including a model for the resources of that technology and their costs. For most of this book, we shall assume a generic one-processor, ***random-access machine (RAM)*** model of computation as our implementation technology and understand that our algorithms will be implemented as computer programs. In the RAM model, instructions are executed one after another, with no concurrent operations.

Strictly speaking, we should precisely define the instructions of the RAM model and their costs. To do so, however, would be tedious and would yield little insight into algorithm design and analysis. Yet we must be careful not to abuse the RAM model. For example, what if a RAM had an instruction that sorts? Then we could sort in just one instruction. Such a RAM would be unrealistic, since real computers do not have such instructions. Our guide, therefore, is how real computers are designed. The RAM model contains instructions commonly found in real computers: arithmetic (such as add, subtract, multiply, divide, remainder, floor, ceiling), data movement (load, store, copy), and control (conditional and unconditional branch, subroutine call and return). Each such instruction takes a constant amount of time.

From:

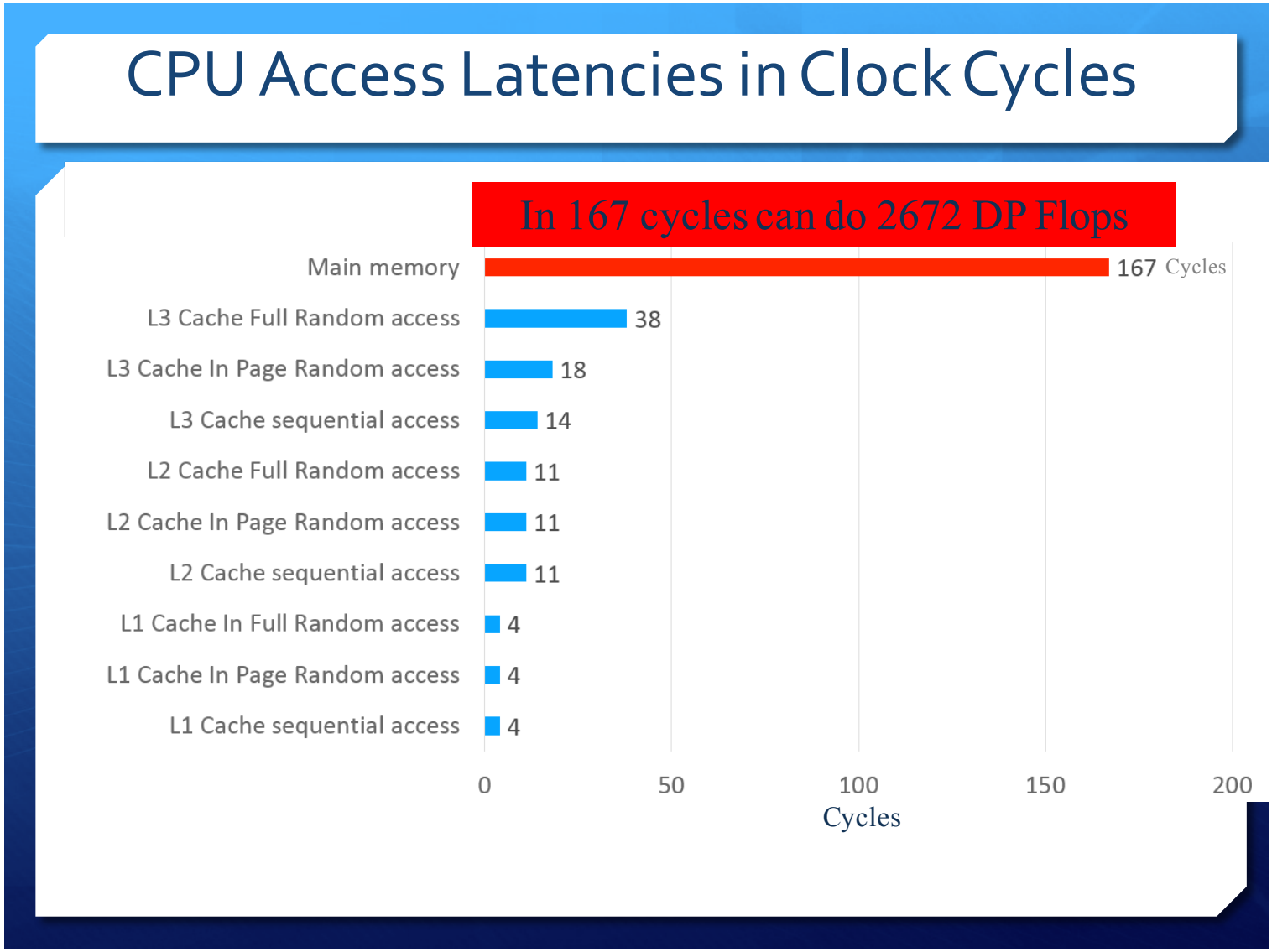


Latency

```
          0.5 ns - CPU L1 dCACHE reference
          1   ns - speed-of-light (a photon) travel a 1 ft (30.5cm) distance
          5   ns - CPU L1 iCACHE Branch mispredict
          7   ns - CPU L2  CACHE reference
         71   ns - CPU cross-QPI/NUMA best  case on XEON E5-46*
        100   ns - MUTEX lock/unlock
        100   ns - own DDR MEMORY reference
        135   ns - CPU cross-QPI/NUMA best  case on XEON E7-*
        202   ns - CPU cross-QPI/NUMA worst case on XEON E7-*
        325   ns - CPU cross-QPI/NUMA worst case on XEON E5-46*
     10,000   ns - Compress 1K bytes with Zippy PROCESS
     20,000   ns - Send 2K bytes over 1 Gbps NETWORK
    250,000   ns - Read 1 MB sequentially from MEMORY
    500,000   ns - Round trip within a same DataCenter
 10,000,000   ns - DISK seek
 10,000,000   ns - Read 1 MB sequentially from NETWORK
 30,000,000   ns - Read 1 MB sequentially from DISK
150,000,000   ns - Send a NETWORK packet CA -> Netherlands
|    |    |    |
|    |    | ns|
|    | us|
| ms|
```

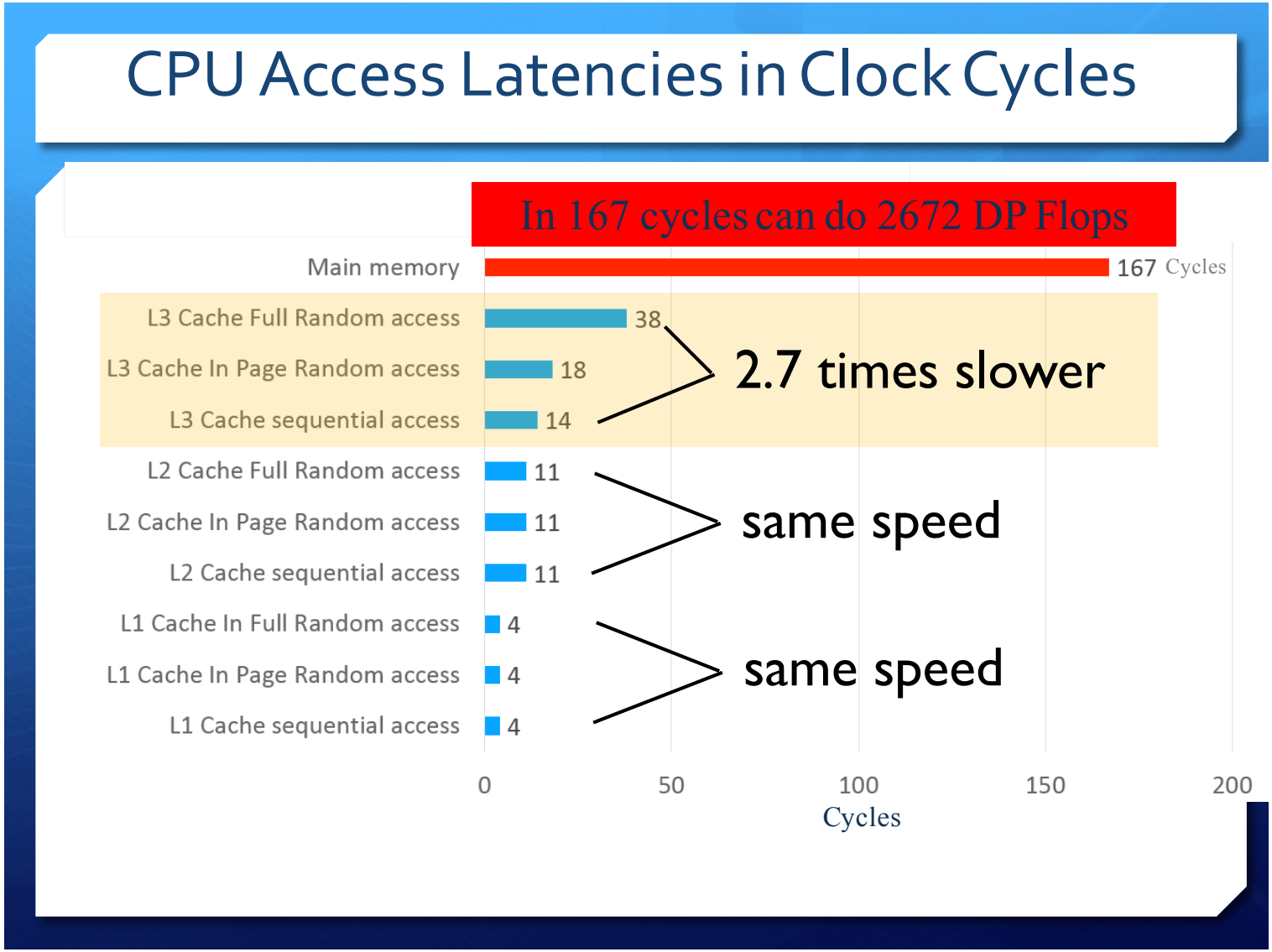
Stackoverflow & http://norvig.com/21-days.html#answers

Latency vs. computation



From:An Overview of High Performance Computing and Benchmark Changes for the Future. Jack Dongarra. Oak Ridge Natl Lab

Latency vs. computation



From:An Overview of High Performance Computing and Benchmark Changes for the Future. Jack Dongarra. Oak Ridge Natl Lab

Consequences ...

- Computation is cheap, data movement is very expensive (compute data colocation problem at higher level)
- Unit cost assumption in complexity analysis very wrong for large data
- Which algorithm to choose not trivial for big data ...

Interested in algorithms and data structures for big data on real machines?
Check out: Demaine, Cache-oblivious Algorithms and Data Structures (2002).
http://erikdemaine.org/papers/BRICS2002/paper.pdf

Case study: random access to array

- Application: Looking up or aggregating values stored in an array
  - ids come from separate array/table/ dataframe sorted according to another criterion
  - ids come as a stream from customers/ processes

Case study: random access to array

```
def out_of_order(data, queries):
    sum = 0.0
    for i in queries:
        sum += data[i]
    return sum
```

data: array of floating point numbers

queries: array of integers

Case study: random access to array

```
def out_of_order(data, queries):
    sum = 0.0
    for i in queries:
        sum += data[i]
    return sum
```

O(n) complexity

n = #elements in queries

```
def in_order(data, queries):
    sum = 0.0
    queries.sort()
    for i in queries:
        sum += data[i]
    return sum
```

O(n log n) complexity

Case study: random access to array

array size	# queries	speedup*
1000	10	1.25
10000	100	1.5
100K	1000	1.5
1M	100K	2.23
10M	1M	1.87
100M	10K	1.56
100M	1M	1.71

Typical: depends on sampled queries
(note results not with Python code)

speedup = original running time / improved running time

$O(n)$ $O(n \log(n))$

Parallel programming models

Multithreading

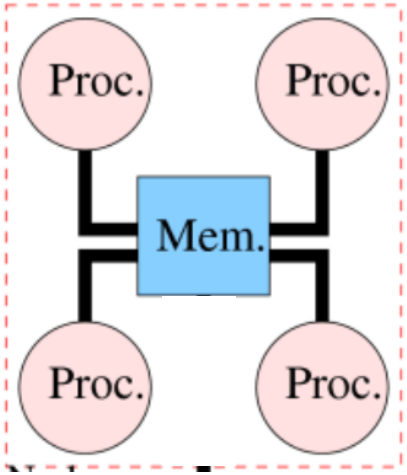
https://www.scipy.org/topical-software.html#parallel-and-distributed-programming

In multi-threaded programming all threads can simultaneously

In this exact formulation of question, without mentioning of other technical means (e.g. locking)

A	B	C	D	E
read data	write data	read and write data	read and write, but not the same data	

Parallel programming models: multi-threading

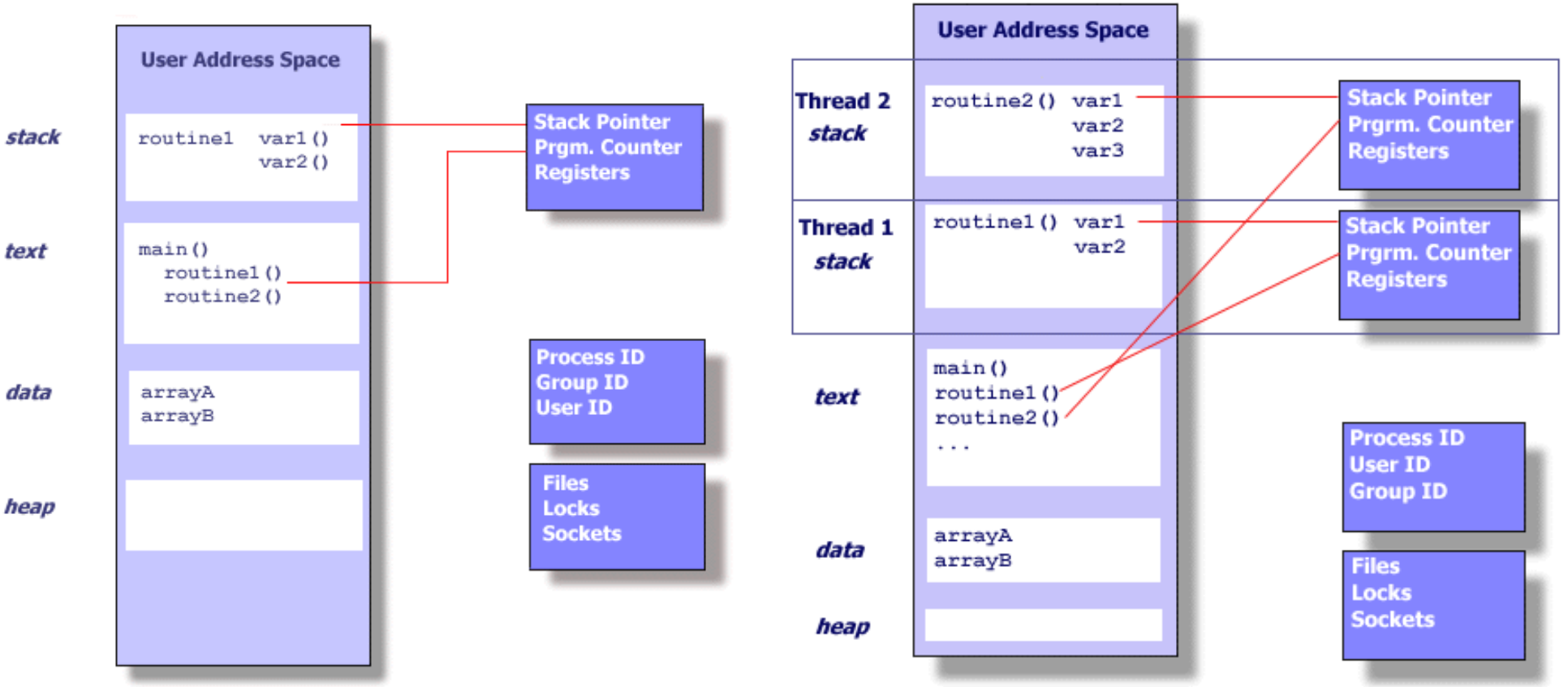


- Threads communicate via variables in **shared memory**.
- Simultaneous read access to data
- Write access to same data requires blocking (lock).
- Automated multi-threading in some libraries (BLAS) and languages (C++).
- Explicit tools and libraries available

In multi-threaded programming the time needed to communicate between two threads is typically on the order of

A	B	C	D	E
2ns	200ns	2000ns	20,000ns	

Threads



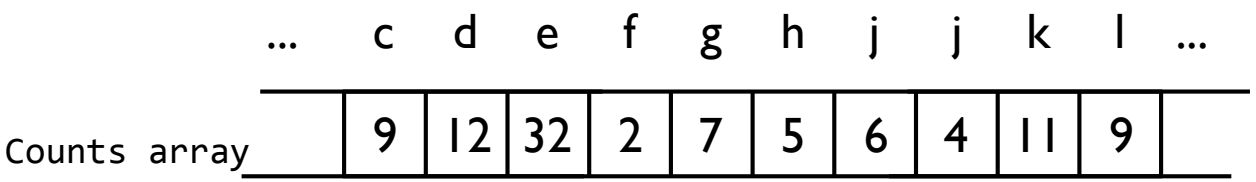
From https://computing.llnl.gov/tutorials/pthreads/

Threads & Shared Memory

- Scoping and access rules of programming language still apply (e.g. local or private variables)
- Since all threads share address space can pass pointer to (address of) a large array and *read concurrently*

Count character frequencies

```
import numpy as np
counts = np.zeros(256, dtype=int)

def countChars(line):
    for c in line:
        counts[ord(c)] += 1

with open('sample.txt') as file:
    for line in file:
        countChars(line)

print(counts)
```



Count character frequencies in parallel

```
import numpy as np
counts = np.zeros(256, dtype=int)
```

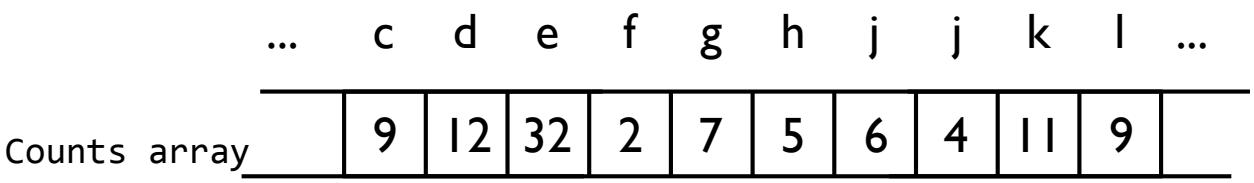
```
def countChars(line):
    for c in line:
        counts[ord(c)] += 1
```

Consumer:
worker consumes lines and counts chars

```
with open('sample.txt') as file:
    for line in file:
        countChars(line)
```

Producer:
produces lines of text for workers (consumers) to process

```
print(counts)
```



Count character frequencies in parallel

- Questions:
  - How do producer/consumer communicate?
  - How are the counts communicated?

multiprocessing.Queue

```
def countCharWorker(queue):
    while True:
        line = queue.get()
        for c in line:
            # increase c count

def producer(…)
    queue = multiprocessing.Queue()

    # create countCharWorker …

    with open('sample.txt') as file:
        for line in file:
            queue.put(line)
```

Queue organizes access

Multiple consumer can get items off the queue in parallel

get() blocks — returns only when *line* available

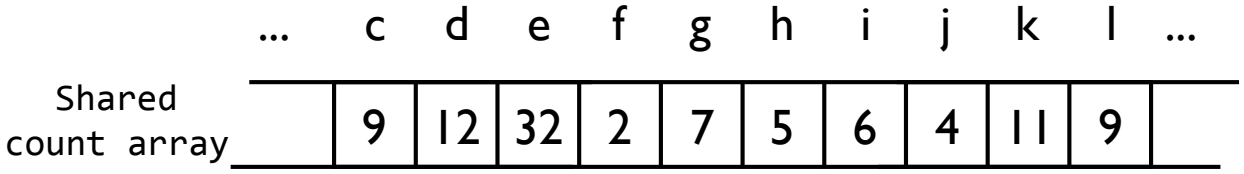
Threads writing to memory

Multi-threaded character counting:

Thread #1

the quick brown fox

```
count['i'] += 1
```




```
count['i'] += 1
```

Thread #2

multi-threading

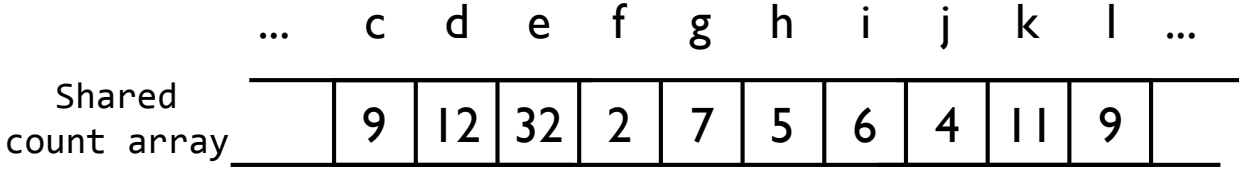
Threads writing to memory

Multi-threaded character counting:

Thread #1

the quick brown fox

```
count['i'] += 1
```




```
count['i'] += 1
```

Thread #2

multi-threading

Possible Execution

1. Thread 1 reads current value 6 from count array into register
2. Thread 1 increments register
3. Thread 2 reads current value 6 from count array
4. Thread 1 writes incremented value 7
5. Thread 2 increments register
6. Thread 2 writes incremented value 7

⇒ count['i'] incorrect

Locking

- Protects from errors due to parallel writes
- Lock
  - is acquired before reading/writing data
  - is released when done
  - assures serial access to shared mem
- Incurs additional overhead (space + time)

Assume you implemented the counting of character frequencies in parallel using 16 parallel threads. There is one lock on the whole array of counts.

The speedup will be about

Single lock makes this a serial program with the overhead of locking

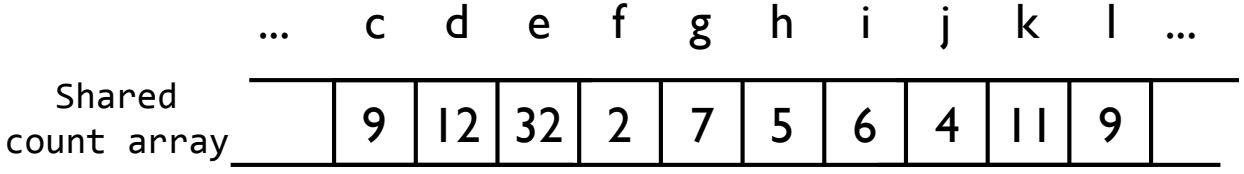
A	B	C	D	E
≤1	2	4	8	16

Threads writing to memory

Multi-threaded character counting:

Thread #1

```
acquire lock count['i']
count['i'] += 1
release lock count['i']
```



Thread #2

```
acquire lock count['i']
count['i'] += 1
release lock count['i']
```

Possible Execution

1. Thread 1 locks count[i]
2. Thread 2 tries to lock count[i] and waits
3. Thread 1 reads current value 6 from count array into register
4. Thread 1 increments register
5. Thread 1 writes incremented value 7
6. Thread 1 releases lock
7. Thread 2 locks count[i]
8. Thread 2 reads current value 7 from count array
9. Thread 2 increments register
10. Thread 2 writes incremented value 8
11. Thread 2 releases lock

⇒ count['i'] correct

Deadlocks

- Execution stops because 2 or more threads wait for each other
  - Two threads need to write variables a & b
  - Thread 1 locks a and waits for b
  - Thread 2 locks b and waits for a

Alternatives to locking

- Modern CPUs and compilers offer *lock-free* or *atomic* operations
- E.g., increment is atomic internally in the CPU and thus is thread-safe (simultaneous increments from multiple threads possible)
- Consequently there are lock-free data structures e.g. bloom filter, hash tables, …

See https://gcc.gnu.org/onlinedocs/gcc/_005f_005fatomic-Builtins.html

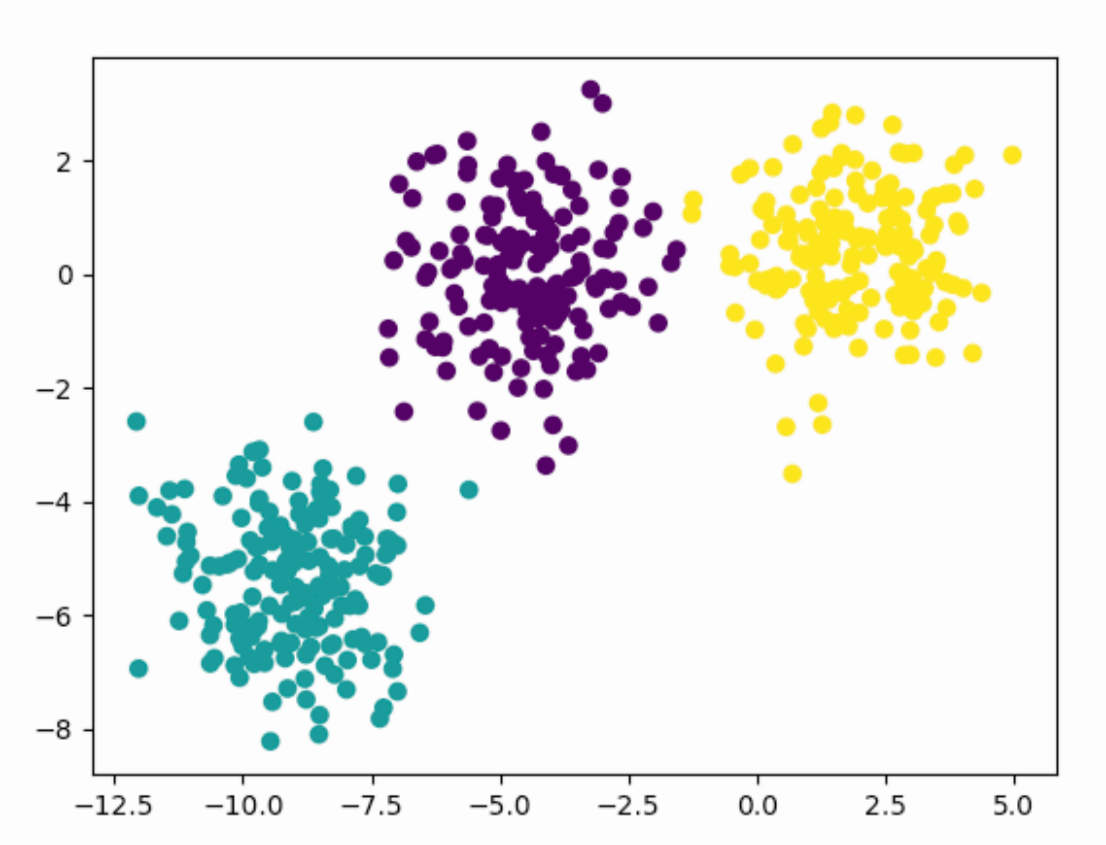
Parallel programming models

Multithreading

https://www.scipy.org/topical-software.html#parallel-and-distributed-programming

Parallel programming models

Multithreading



Case study: k-means clustering

k-means clustering

- Partitioning of the data into k cluster C_1, C_2, …, C_k.
  - $C_i \cap C_j = \emptyset$ for $i \neq j$
  - $C_1 \cup C_2 \cup \ldots \cup C_k = \{1,\ldots,n\}$
- k needs to be fixed in advance

k-means clustering

$$min_{C_1,C_2,\ldots,C_k}\{\sum_{k=1}^{K} W(C_k)\}$$

$$W(C_k) = \frac{1}{|C_k|}\sum_{i,j\in C_k} \| x_i - x_j \|^2$$

$\| x - y \|^2$ Euclidean distance

k-means clustering

- Chose k centroids among data points
- Iterate:
  - assign data points to closest centroid
  - recompute centroids

```
def nearestCentroid(datum, centroids):
    # norm(a-b) is Euclidean distance, matrix - vector computes difference
    # for all rows of matrix
    dist = np.linalg.norm(centroids - datum, axis=1)
    return np.argmin(dist), np.min(dist)


def kmeans(k, data, nr_iter = 100):
    N = len(data)

    # Choose k random data points as centroids
    centroids = data[np.random.choice(np.array(range(N)),size=k,replace=False)]
    print "Initial centroids\n", centroids

    N = len(data)

    # The cluster index: c[i] = j indicates that i-th datum is in j-th cluster
    c = np.zeros(N, dtype=int)

    print "Iteration\tVariation\tDelta Variation"
    total_variation = 0.0
    for j in range(nr_iter):
        #print "=== Iteration %d ===" % (j+1)

        # Assign data points to nearest centroid
        variation = np.zeros(k)
        cluster_sizes = np.zeros(k, dtype=int)
        for i in range(N):
            cluster, dist = nearestCentroid(data[i],centroids)
            c[i] = cluster
            cluster_sizes[cluster] += 1
            variation[cluster] += dist**2
        delta_variation = -total_variation
        total_variation = sum(variation)
        delta_variation += total_variation
        print "%3d\t\t%f\t%f" % (j, total_variation, delta_variation)

        # Recompute centroids
        centroids = np.zeros((k,2))
        for i in range(N):
            centroids[c[i]] += data[i]
        centroids = centroids / cluster_sizes.reshape(-1,1)
```

k-means clustering

- Questions to ask:
  - Which sections can be parallelized?
  - What needs to be serial?
  - When is communication necessary?
  - How much data needs to be communicated?

Parallelizing multiple k-means

Note: k is usually determined empirically. E.g. run for k = 2, 3, 4, 5, …

```
def kmeans(k, data, nr_iter):
    ...

def compute_kmeans(args):
    ...

    p = multiprocessing.Pool(args.workers)
    s = p.starmap(kmeans,
                  [(k, data, 100) for k in range(2,max_K)])
```

There is no communication between parallel threads!

starmap allows multiple arguments in contrast to map

When running k-means in parallel for k = 128, 256, 512, 1024 on a 4-core machine the expected speedup is ~4 …

A	B	C	D	E
True	False			

Parallel programming problems

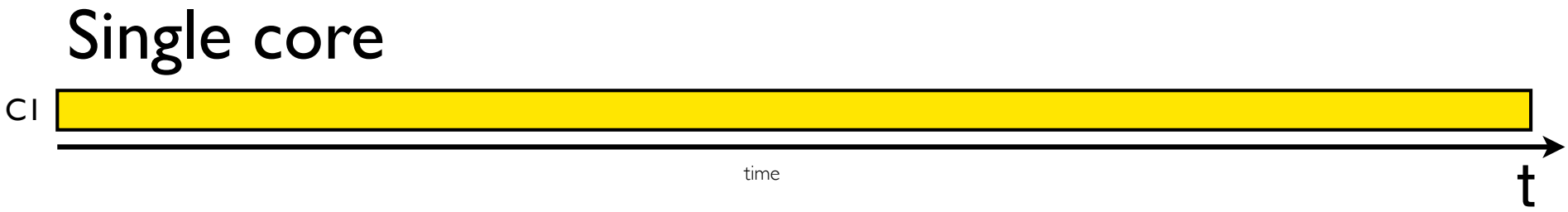
Load balancing

```
def nearestCentroid(datum, centroids):
    # norm(a-b) is Euclidean distance, matrix - vector computes difference
    # for all rows of matrix
    dist = np.linalg.norm(centroids - datum, axis=1)
    return np.argmin(dist), np.min(dist)


def kmeans(k, data, nr_iter = 100):
    N = len(data)

    # Choose k random data points as centroids
    centroids = data[np.random.choice(np.array(range(N)),size=k,replace=False)]
    print "Initial centroids\n", centroids

    N = len(data)

    # The cluster index: c[i] = j indicates that i-th datum is in j-th cluster
    c = np.zeros(N, dtype=int)

    print "Iteration\tVariation\tDelta Variation"
    total_variation = 0.0
    for j in range(nr_iter):
        #print "=== Iteration %d ===" % (j+1)

        # Assign data points to nearest centroid
        variation = np.zeros(k)
        cluster_sizes = np.zeros(k, dtype=int)
        for i in range(N):
            cluster, dist = nearestCentroid(data[i],centroids)
            c[i] = cluster
            cluster_sizes[cluster] += 1
            variation[cluster] += dist**2
        delta_variation = -total_variation
        total_variation = sum(variation)
        delta_variation += total_variation
        print "%3d\t\t%f\t%f" % (j, total_variation, delta_variation)

        # Recompute centroids
        centroids = np.zeros((k,2))
        for i in range(N):
            centroids[c[i]] += data[i]
        centroids = centroids / cluster_sizes.reshape(-1,1)
```

- Complexity is O(k · N · iterations)
- Measured 22s, 26s, 33s, 48s = 129s total for k =128, 256,512,1024
- Parallel implementation needs 48s ⇒speedup of 2.6

Q: Why the large k?

Load Balancing: k-means



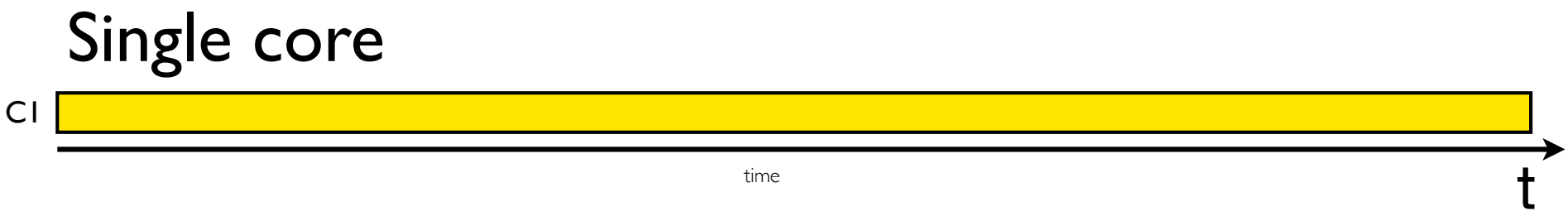
Serial and parallel sections on 4 cores

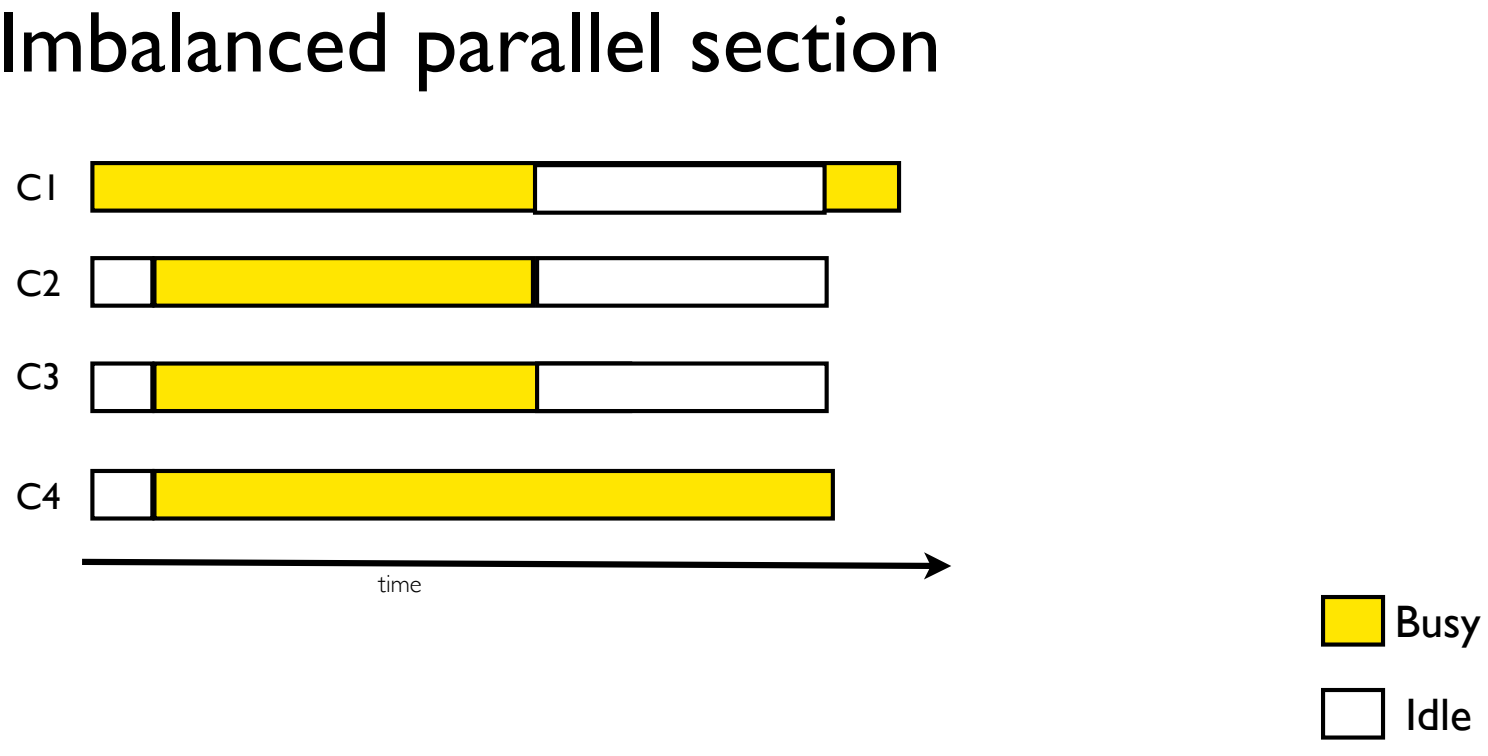
Student Q: How to control load balancing?

A: This is controlled by the design of the problem. For example on 3 cores, one could core could have handled the k=128 and k=256 clustering in 48s, the second core k=512 in 33s, and the third k=1024 in 48s, for the same speedup 2.6 on 3 instead of 4 cores.

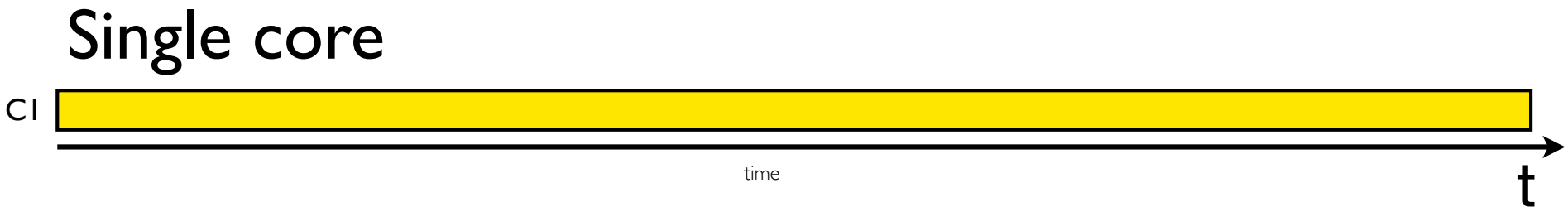
There are more complicated libraries for assigning a large number of threads to cores. Simulation codes, for weather or traffic for example, might do load balancing dynamically changing (making smaller or larger) the part of the world for which each node computes the simulation if it takes longer.

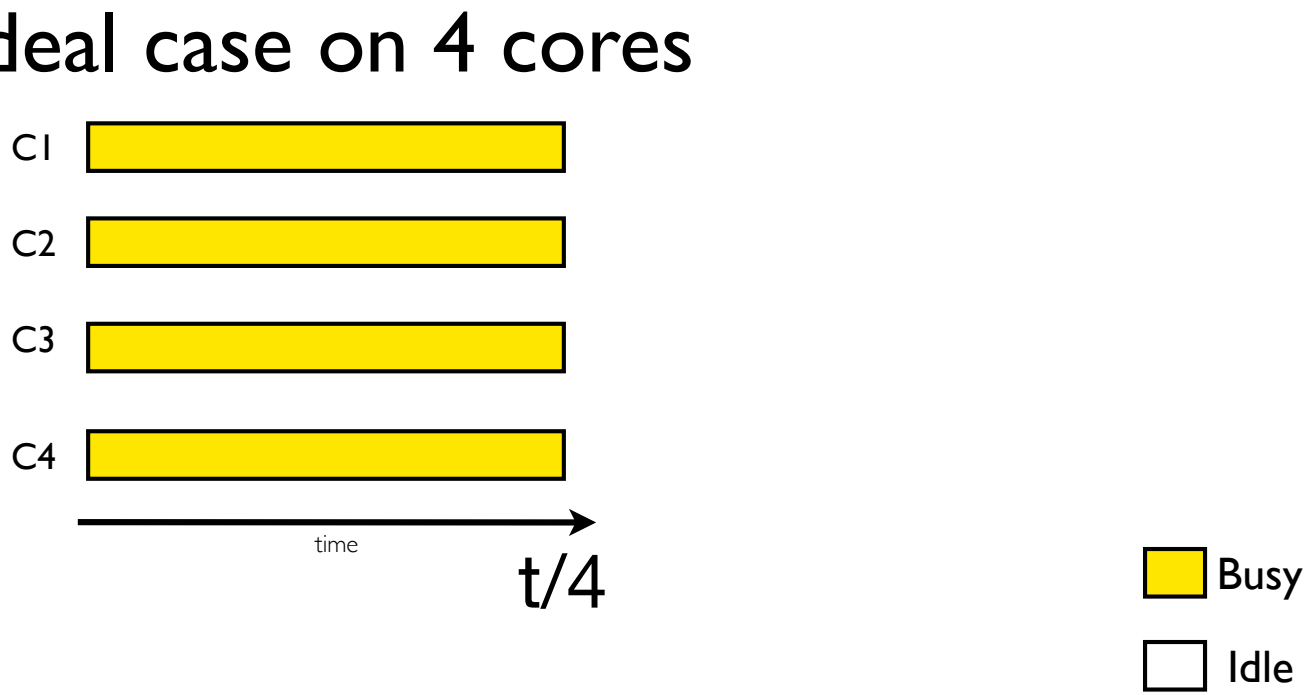
Load Balancing: k-means





Load Balancing: Ideal case





Load Balancing:
Even distribution of computational load among cores s.t. no idling occurs

Load Balancing

Parallel programming models

Multithreading — sharing data

https://www.scipy.org/topical-software.html#parallel-and-distributed-programming

Under the hood: Python arrays

```
>>> import random
>>> def square(x):
...     return x*x
>>> l = [1,3.14, "Hello World", {}, random, square]
>>> l.insert(0,[])
```

Lists are not arrays

Under the hood: Python arrays

- Typically an array is:
  - A contiguous fixed-size block of memory
  - Interpreted as a fixed number of cells of the same data type (char, int, float, double)
  - a[i] convenient notation for reading from/writing to memory at a specific location
  - No overhead (but no flexibility or safety)

address

address+8

address+16

Assuming 64bit floating point numbers

Under the hood: Python arrays

```
>>> a= np.zeros(10)
>>> a
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
>>> a.dtype
dtype('float64')
>>> help(a.ctypes)
>>> a.ctypes.get_data()
140314792769984
```

Address of memory block

address

address+8

address+16

Assuming 64bit floating point numbers

Under the hood: Python arrays

- Numpy, Python array module expose the underlying C-arrays
- This allows sharing them in memory
- See `multiprocessing.RawArray`

https://docs.python.org/2/library/array.html