

# Web and Cloud Computing - Parallel Computing & MapReduce

UBCO Master of Data Science – DATA 534





# This week slides

Parallel Computing  
map, filter, reduce  
MapReduce

# Summary of Internet, Web

---

Computer, MAC, IP, port, switch/router, firewall

UDP, TCP/IP, HTTP/HTTPS, DNS servers

Client-server, peer-to-peer

Web server: Apache, NCIX;

Web browser: Chrome, Firefox, Safari

HTML/CSS/JavaScript or API/JSON/XML



# Common well-known ports

- 1024-49151: internet community
- 49152-65535: private

Service	Port	Function
HTTP	80	Web
HTTPS	443	Web (secure)
FTP	20,21	File transfer
SFTP	22	File transfer (secure)
FTPS	989,990	File transfer (secure)
SIP	5060	VoIP (Internet phone)
DNS	53	Find IP address
SMTP	25	Internet mail
POP3	110	POP mailbox
IMAP	143	IMAP mailbox
Telnet	23	Remote login
SSH	22	Remote login (secure)
NNTP	119	Usenet newsgroups
NNTPS	563	Usenet (secure)
IRC	194	Chat
NTP	123	Network time of day
SNMP	161,162	Network management
CMIP	163,164	Network management
Syslog	514	Event logging
Kerberos	88	Authentication
NetBIOS	137-139	DOS/Windows naming

# Lecture Learning Goals

---

- Using examples of simple operations on a data collection (maybe examples from census data) and comparing iteration to operations like map/reduce, provide insight into which of these techniques is most appropriate for accessing a large, distributed data collection.
- Design and deploy a small program that uses map/reduce-like operations to access a large, distributed collection.

# Why parallel computing?

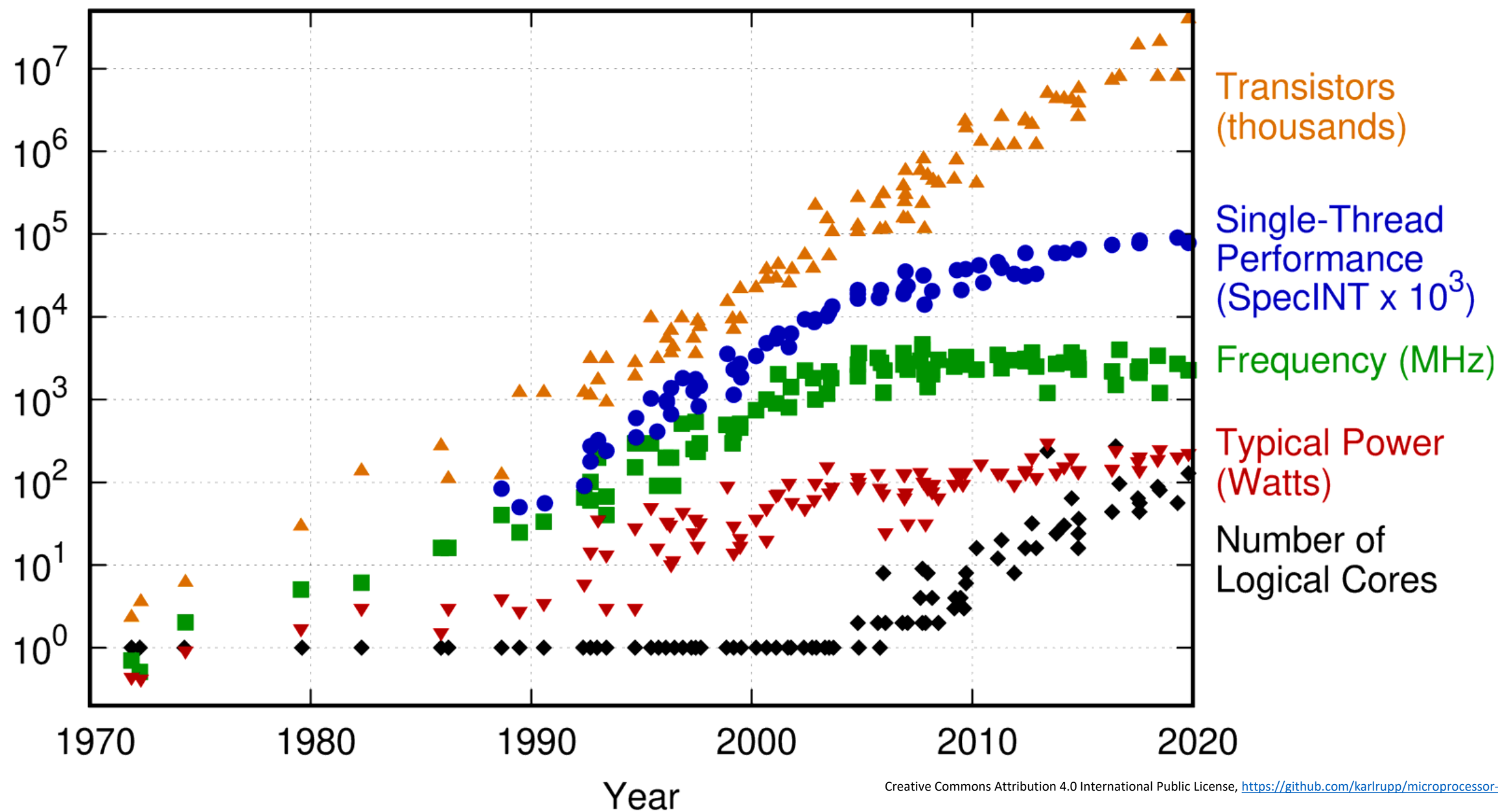
# Changing times

---

From 1986 – 2002, microprocessors were speeding like a rocket, increasing in performance an average of 50% per year.

Since then, it's dropped to about 20% increase per year.

# 48 Years of Microprocessor Trend Data



Creative Commons Attribution 4.0 International Public License, <https://github.com/karlrupp/microprocessor-trend-data>

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 2010-2019 by K. Rupp

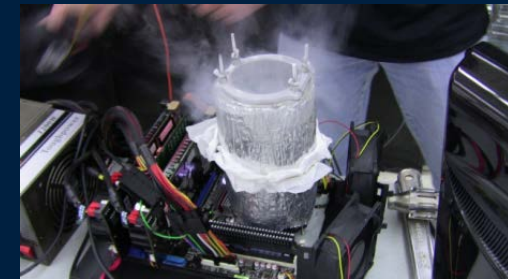


# Highest frequency

Since 2014, the Guinness World Record for the highest CPU clock rate is an overclocked, **8.723 GHz** AMD Piledriver-based FX-8370 chip. It surpassed the previous record achieved in 2011, an 8.429 GHz AMD FX-8150 Bulldozer-based chip. [[https://en.wikipedia.org/wiki/Clock\\_rate](https://en.wikipedia.org/wiki/Clock_rate)]

As of mid-2013, the highest clock rate on a production processor is the IBM zEC12, clocked at 5.5 GHz, which was released in August 2012.

Latest overclocking switched from liquid nitrogen (-196 Celsius) to liquid helium (-269 degrees Celsius)



[This Photo](#) by Unknown Author is licensed under [CC BY](#)

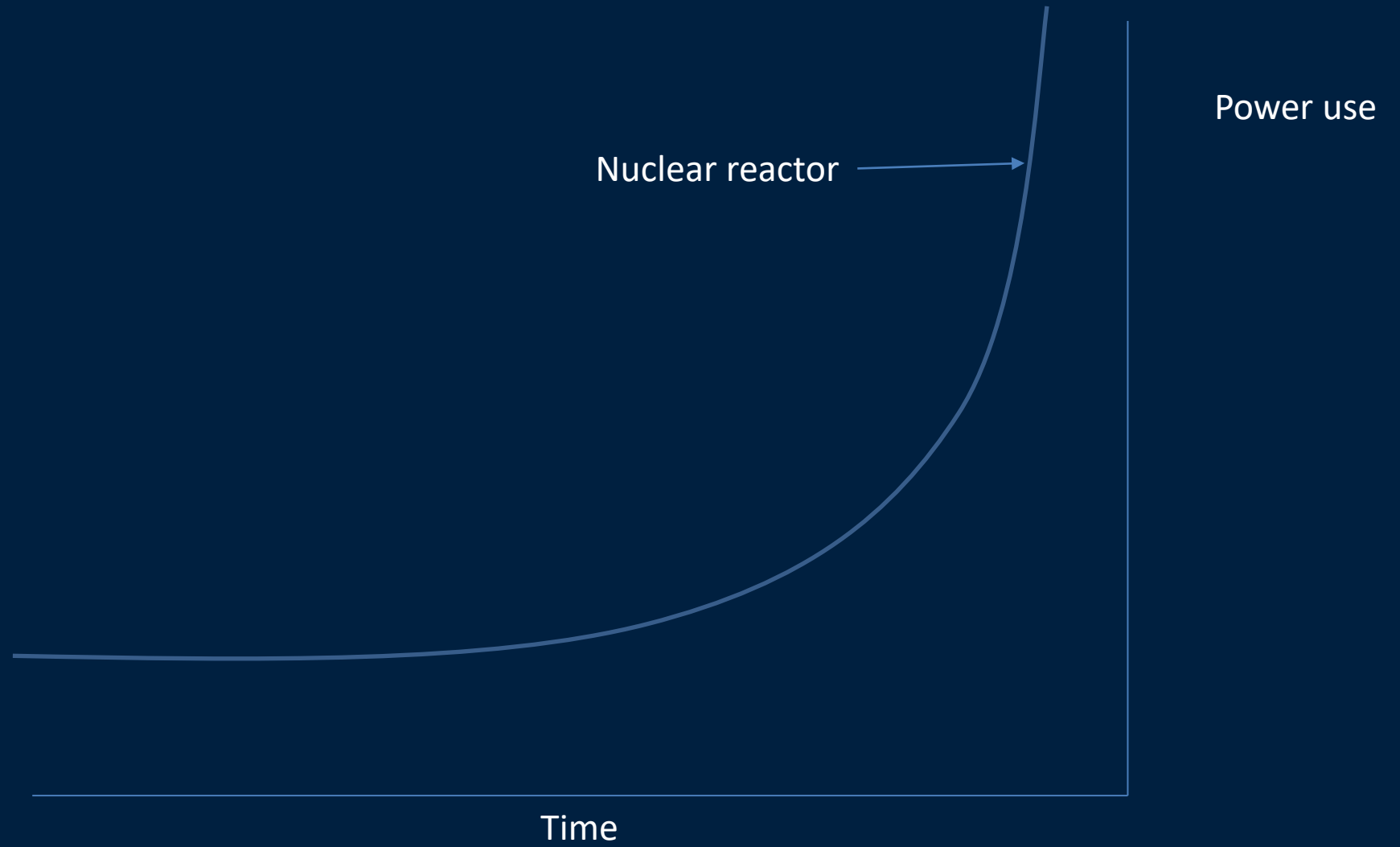
# Harder to increase frequency

---

heat

power consumption

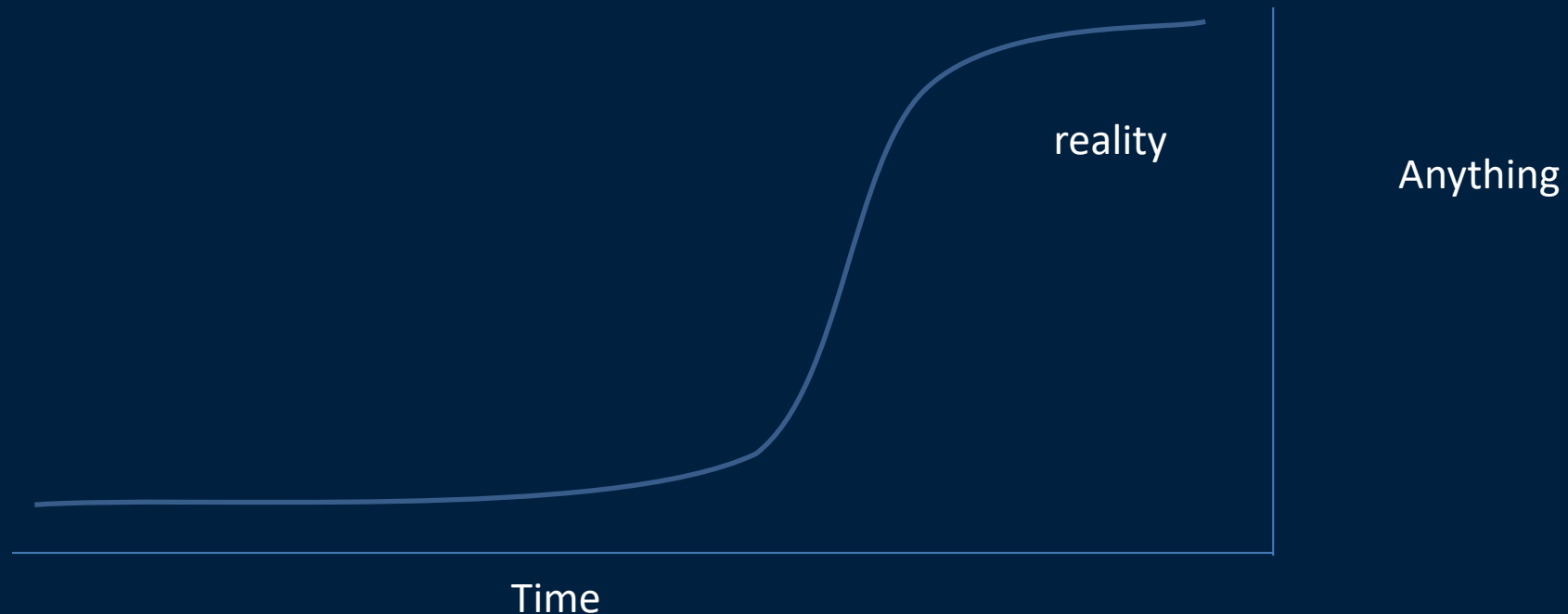
leakage problems



“If the automobile had followed the same development cycle as the computer, a Rolls-Royce would today cost \$100, get a million miles per gallon, and explode once a year, killing everyone inside.”

— **Robert X. Cringely**

<https://www.goodreads.com/quotes/100838-if-the-automobile-had-followed-the-same-development-cycle-as>



# Solution

Put multiple processors on a single chip

Up to the programmer to maximize utilization

- More processor are useless unless programmers know how to utilize them
- Serial programming does not benefit

Why do we need more computational power?

- Solve bigger and bigger problems
- Manipulate larger data sets
- Store more and more data
  - World data was estimated at 18 zettabytes in 2018 and is expected to reach 175 zettabytes by 2025.
- Data Analysis



This Photo by Unknown Author is licensed under [CC BY-SA](#)



# Parallel programming

---

Running 16 times the same game is useless

Need 1 game to run faster

# Approaches to the serial problem

---

**Rewrite** serial programs so that they're parallel.

Write translation programs that **automatically** convert serial programs into parallel programs.

- This is very difficult to do.
- Success has been limited.

# Example

---

Compute  $n$  values and add them together

```
def f(x):  
    return x * x;  
  
n = 100  
total = 0  
for i in range(1,n,1):  
    total = total + f(i)  
print("total=",total)
```

# Parallel

Have  $p$  cores numbered from 0 to  $p-1$


Each core  $i$  computes  $n/p$  values

Add results together: each core sends results to core 0

Core	0	1	2	3	4	5	6	7
subtotal	8	19	7	15	7	13	12	14

Core	0	1	2	3	4	5	6	7
total	95	19	7	15	7	13	12	14



---

**Better way**



# Better way

Core	0	1	2	3	4	5	6	7
subtotal	8	19	7	15	7	13	12	14

Core	0	1	2	3	4	5	6	7
subtotal	27		22		20		26	

Core	0	1	2	3	4	5	6	7
subtotal	49				46			

Core	0	1	2	3	4	5	6	7
subtotal	95							

# Compare

## First approach

- Master (core 0)
  - 7 receives
  - 7 additions

## Second approach

- Master (core 0)
  - 3 receives
  - 3 parallel additions
  - $3 = \text{Log}_2(8) = \text{Log}(8) / \text{Log}(2)$

With 1,000 cores, improvement of almost a factor of 100

# Writing parallel programs

---

## Task parallelism

- Split various tasks among the cores
- Each core performs different tasks on the same data

## Data parallelism

- Split data among the cores
- Each core performs the same tasks on different data

# Example: grading assignments

---

1000 assignments, 10 TAs, 100 questions

Task parallelism

- Each TA is assigned 10 questions and grade those on all assignments

Data parallelism

- Each TA is assigned 100 assignments and grade each one entirely

# Pitfalls

---

Decide how to split the work: by task, by data

Coordinate the work

- Communicate: some cores send partial results to another core
- Load balancing: share the load evenly to avoid any bottleneck, i.e. waiting for one core to finish its tasks
- Synchronization: make sure no core gets too far ahead of the others to avoid waiting a long time for one core



# Computing in parallel

Easiest: embarrassingly parallel tasks

- For each  $i$ , compute  $f(i)$

Hardest: depending tasks

- $F_0 = 1, F_1 = 1, F_n = F_{n-1} + F_{n-2}$
- [Fibonacci sequence can be computed faster using dynamic programming and memorization; this is not the point here]
- $x = x - t g$   
steepest descent:  $g$  gradient,  $t$  step size

In between: map-reduce

- $S = \sum_i f(i)$
- Computing  $s_i = f(i)$  is embarrassingly parallel; MAP
- Computing  $S = \sum_i s_i$  is a reduction; REDUCE

# Python map, reduce

```
import functools

# initializing list
lis = [ 1 , 3, 5, 6, 2 ]
def myfunc(a):
    return a*a

A = map(myfunc, lis)
s = functools.reduce(lambda a, b : a+b, A)

print("the sum of square of lis is: ", s)
```

the sum of square of lis is: 75

reduce and map functions in Python are sequential (need multiprocessing package to use multicores)

# Parallel code

Seq

Map [parallel]

Join/Synchronize

Seq

Map [parallel]

Reduce [parallel+join]

- Not all code is parallelizable
- Sequential code followed by parallel code

# Challenges aka bottlenecks

## Computation speed:

- how fast can you compute when everything is in memory?

## Memory

- How large is your memory?
- Memory hierarchy: RAM
- Moving data in memory: CPU-GPU

## Distributed computing

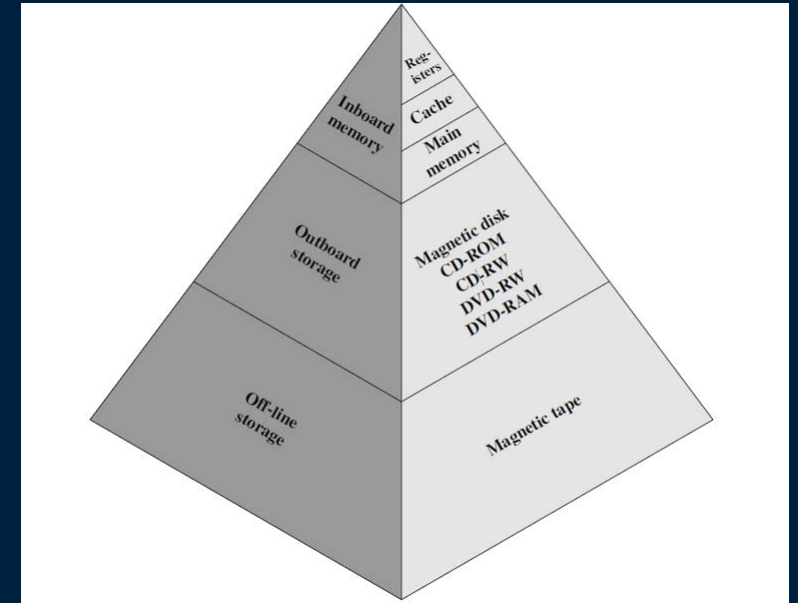
- Network speed: moving data between computers

## Reliability

- any error in the computation?

## Durability

- How long is the data stored without error?



<https://commons.wikimedia.org/wiki/File:Hierarkia.png>

# Amazon S3 FAQs <https://aws.amazon.com/s3/faqs/>

---

Q: How durable is Amazon S3?

- Amazon S3 Standard, S3 Standard-IA, S3 One Zone-IA, S3 Glacier, and S3 Glacier Deep Archive are all designed to provide 99.999999999% durability of objects over a given year. This durability level corresponds to an average annual expected loss of 0.000000001% of objects. For example, if you store 10,000,000 objects with Amazon S3, you can on average expect to incur a loss of a single object once every 10,000 years.

Q: How reliable is Amazon S3?

- Amazon S3 gives any developer access to the same highly scalable, highly available, fast, inexpensive data storage infrastructure that Amazon uses to run its own global network of web sites. The S3 Standard storage class is designed for 99.99% availability



# Data Scale and Availability

---

If your goals are to scale data

- to really big size, with very high-bandwidth access and
- to provide 11 9's of durability and 4 9's of availability (S3's claim)

Your data must be distributed and replicated so you can

- distribute load across many machines
- avoid single points of failure
- eliminate performance bottlenecks

But how do you manipulate distributed data at scale

- relational databases can't scale like this
- the underlying infrastructure is a distributed hash table
- but ... how do you search or transform a massive, distributed collection

# Scaling Data and Computation

Distributed Hash Tables scale data with simplified API

- each item has a globally unique key and a value (e.g., S3 object)
- values are stored on specific nodes, determined by their key
- a distributed index locates storing nodes based on their keys

But, what about manipulating this data

- virtual machine can scale computation load up to 1 physical machine
- beyond that, changes are required at the application / algorithm level
- we need ways to express the ability of computations to be parallel

Parallel computation

- divide work into chunks that can be computed independently on different nodes
- this means that there is no shared memory / communication during parallel phase
- not everything can be parallel; parallel phases are connected by sequential phases

# Iterating over a collection

---

Lets say you want to process census data to determine

- which city has the most people with PhD's with debt to income ratio  $> 2$

How do you do it?

- the census is a collection of personal **surveys** each with fields
  - highest **education** achieved
  - **debt**
  - **income**
  - **city** of residence
- do this in python

```
from functools import reduce
```

```
surveys = [  
    {  
        'education': 'HS',  
        'income': 40000,  
        'debt': 100000,  
        'city': 'Kelowna'  
    },  
    {  
        'education': 'PhD',  
        'income': 80000,  
        'debt': 200000,  
        'city': 'Kelowna'  
    },  
    {  
        'education': 'PhD',  
        'income': 100000,  
        'debt': 300000,  
        'city': 'Vancouver'  
    },  
]
```

```
{  
    'education': 'MS',  
    'income': 40000,  
    'debt': 100000,  
    'city': 'Vancouver'  
},  
{  
    'education': 'PhD',  
    'income': 100000,  
    'debt': 300000,  
    'city': 'Vancouver'  
},  
{  
    'education': 'PhD',  
    'income': 150000,  
    'debt': 350000,  
    'city': 'Vancouver'  
}  
]
```

```
from functools import reduce
from collections import Counter
```

```
surveys = [
    {
        'education': 'HS',
        'income': 40000,
        'debt': 100000,
        'city': 'Kelowna'
    },
    {
        'education': 'PhD',
        'income': 80000,
        'debt': 200000,
        'city': 'Kelowna'
    },
    {
        'education': 'PhD',
        'income': 100000,
        'debt': 300000,
        'city': 'Vancouver'
    },
    ...
]
```

```
Counter({'Vancouver': 3, 'Kelowna': 1})
Vancouver
```

```
from functools import reduce
from collections import Counter
```

```
surveys = [
    {
        'education': 'HS',
        'income': 40000,
        'debt': 100000,
        'city': 'Kelowna'
    },
    {
        'education': 'PhD',
        'income': 80000,
        'debt': 200000,
        'city': 'Kelowna'
    },
    {
        'education': 'PhD',
        'income': 100000,
        'debt': 300000,
        'city': 'Vancouver'
    },
    ...
]
```

```
A = [o['city'] for o in surveys if o['education']=='PhD' and o['debt'] / o['income']>2]
C = Counter(A)
print(C)
print(reduce(lambda a,b : max(a,b), C))
```

```
Counter({'Vancouver': 3, 'Kelowna': 1})
Vancouver
```

# Doing this at scale

---

If data is too big for one server

- put it in a distributed hash table like S3

If query load is too big for one server

- distribute it across multiple computation nodes like EC2's
- but how?

# Map Reduce *[Higher Order Functions]*

**d = map** (f, c)

- $d = \{f(c_0), f(c_1), \dots, f(c_{n-1})\}$  for all  $c_i$  in  $c$
- notice that map is massively parallel
- each  $f(c_i)$  is independent

**d = filter** (f, c)

- d contains all elements  $c_i$  of  $c$  where  $f(c_i)$  is true
- also massively parallel

**d = reduce** (f, c)

- $d = f(f(\dots f(f(i, c_0), c_1), \dots, c_{n-2}), c_{n-1})$
- notice that calls to  $f$  can not run in parallel
- each call to  $f(a_{i-1}, c_{i-1})$  depends on the previous one;  $a_i = f(a_{i-1}, c_{i-1})$



# Small Example

---

```
from functools import reduce
a = [1,2,3,4]
b = map(lambda i: i*2, a)
c = filter(lambda i: i>2, b)
d = reduce(lambda x, i: x+i, c)

print(a, b, c, d)
```

What prints?

# Small Example

```
from functools import reduce
a = [1,2,3,4]
b = map(lambda i: i*2, a)
c = filter(lambda i: i>2, b)
d = reduce(lambda x, i: x+i, c)

print(a, b, c, d)
```

What prints?

```
[1, 2, 3, 4] <map object at 0x0000024D89BC8A00> <filter object at 0x0000024D89BC89A0> 18
```



<terminated> DATA534-W3 SmallExampleMapFilterReduce.py [Python R...]  
<terminated> SmallExampleMapFilterReduce.py  
<terminated, exit value: 1> SmallExampleMapFilterReduce.py [debug]

```
mapreduce census SmallExempl... »  
1 from functools import reduce  
2 a = [1,2,3,4]  
3 b = map(lambda i: i*2, a)  
4 c = filter(lambda i: i>2, b)  
5 d = reduce(lambda x, i: x+i, c)  
6  
7 print(a, b, c, d)  
8
```

```
<terminated> SmallExampleMapFilterReduce.py [debug] [C:\Users\ylucet\anaconda3\python.exe]  
[1, 2, 3, 4] <map object at 0x00000218B024FEB0> <filter object at 0x00000218B024FCA0> 18
```

# Another Example

---

Consider

- a collection of student records
- each student has a name and a list of grades
- print the name of one of the students with the highest overall grade
  - among all of the individual grades achieved by each student
- using only map, filter and reduce

# Another Example

```
from functools import reduce
```

```
students = [  
    {"name": "bob", "grades": [1,2,3]},  
    {"name": "alice", "grades": [4,5,6]}  
]
```

```
studentsWithMaxGrade = map(lambda s: {  
    "name": s["name"],  
    "maxGrade": reduce(lambda m,g: max(m,g), s["grades"])  
}, students)
```

```
prizeWinner = reduce(lambda p,s: p if p["maxGrade"] > s["maxGrade"] else s,  
studentsWithMaxGrade)
```

```
print(prizeWinner["name"])
```

# Another Example

```
from functools import reduce
```

```
students = [
    {"name": "bob", "grades": [1,2,3]},
    {"name": "alice", "grades": [4,5,6]}
]
```

```
studentsWithMaxGrade = map(lambda s: {
    "name": s["name"],
    "maxGrade": reduce(lambda m,g: max(m,g), s["grades"])
}, students)
```

```
prizeWinner = reduce(lambda p,s: p if p["maxGrade"] > s["maxGrade"] else s,
studentsWithMaxGrade)
```

```
print(prizeWinner["name"])
```

Output

alice

# Map Reduce Census Data

---

Recall this problem ...

- the census is a collection of personal **surveys** each with fields
  - highest **education** achieved
  - **debt**
  - **income**
  - **city** of residence
- do this in python pseudocode using map reduce

```
from functools import reduce
```

```
surveys = [  
    {  
        'education': 'HS',  
        'income': 40000,  
        'debt': 100000,  
        'city': 'Kelowna'  
    },  
    {  
        'education': 'PhD',  
        'income': 80000,  
        'debt': 200000,  
        'city': 'Kelowna'  
    },  
    {  
        'education': 'PhD',  
        'income': 100000,  
        'debt': 300000,  
        'city': 'Vancouver'  
    },  
]
```

```
{  
    'education': 'MS',  
    'income': 40000,  
    'debt': 100000,  
    'city': 'Vancouver'  
},  
{  
    'education': 'PhD',  
    'income': 100000,  
    'debt': 300000,  
    'city': 'Vancouver'  
},  
{  
    'education': 'PhD',  
    'income': 150000,  
    'debt': 350000,  
    'city': 'Vancouver'  
}  
]
```





# Map Reduce Census Data

```
def aggregateByCity(d,c):  
    try:  
        d[c['city']].append(c['dirat'])  
    except KeyError:  
        d[c['city']] = [c['dirat']]  
    return d  
  
hasPhD = filter(lambda s: s['education']=='PhD', surveys)  
dirat = map(lambda r: {'city': r['city'],  
                       'dirat': float(r['debt']) / float(r['income'])},  
            hasPhD)  
match = filter(lambda r: r['dirat] > 2, dirat)  
byCity = reduce(aggregateByCity, match, {}).items()  
countByCity = map(lambda c: {'city': c[0], 'count': len(c[1])}, byCity)  
answer = reduce(lambda m,r: r if r['count'] > m['count'] else m, countByCity)
```

Output

```
{'city': 'Vancouver', 'count': 3}
```



# Distributed Map Reduce

---

## Distributed and in Parallel

- each map can run on a different (EC2) node
- reduce groups entries by key and reduces each group on single node

## Examples

- Hadoop is an apache API for distributed map reduce
- EMR is Amazon's encapsulation of Hadoop for EC2 clusters

# Next Class

---

Project Time

The lecture after:

Spark

NoSQL DB





THE UNIVERSITY OF BRITISH COLUMBIA

