COMPSCI 589
Lecture 12: Introduction to Data Parallel Computing

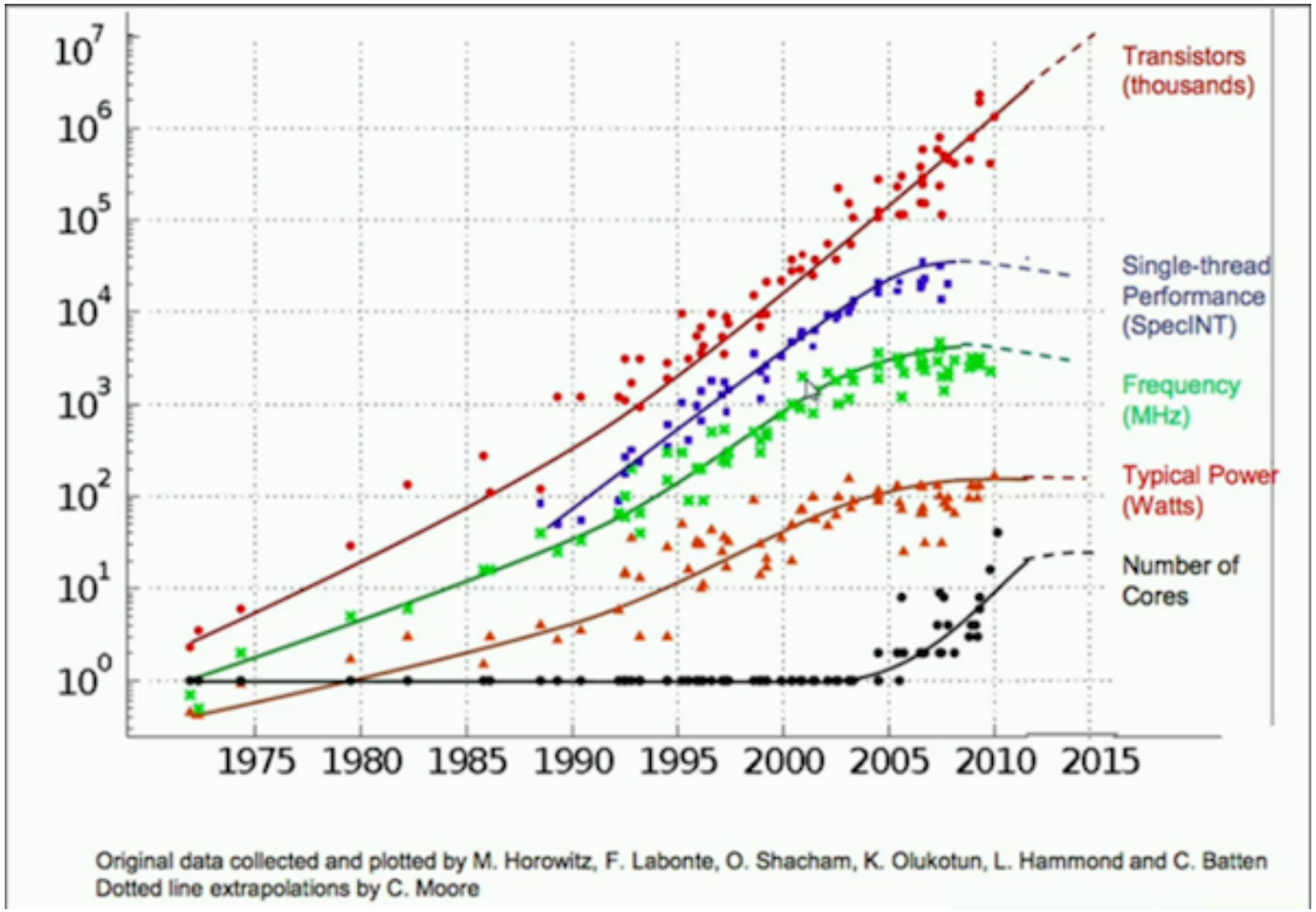
Benjamin M. Marlin

College of Information and Computer Sciences
University of Massachusetts Amherst

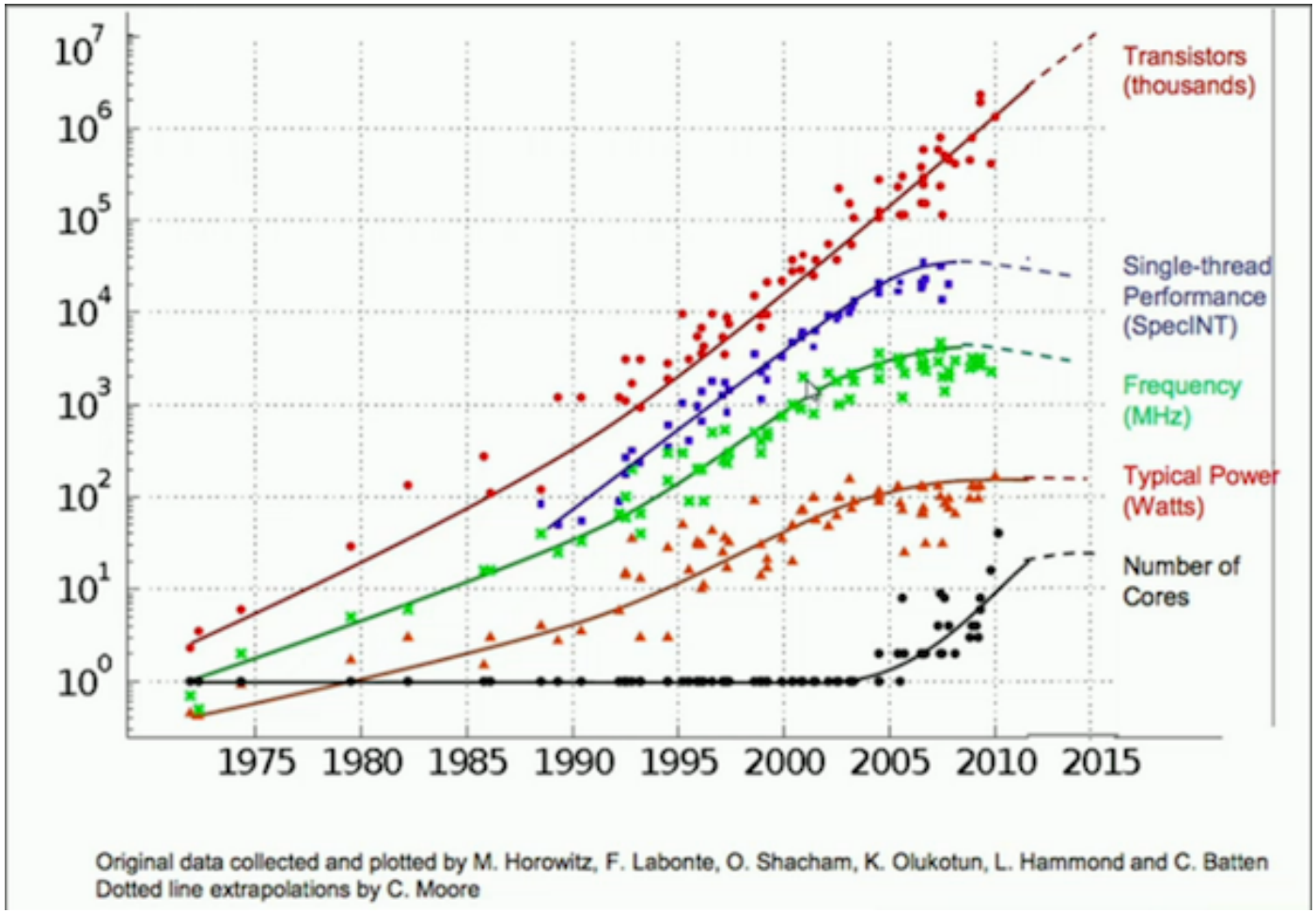
Slides by Benjamin M. Marlin (marlin@cs.umass.edu).
Created with support from National Science Foundation Award# IIS-1350522.

Outline

Moore's Law

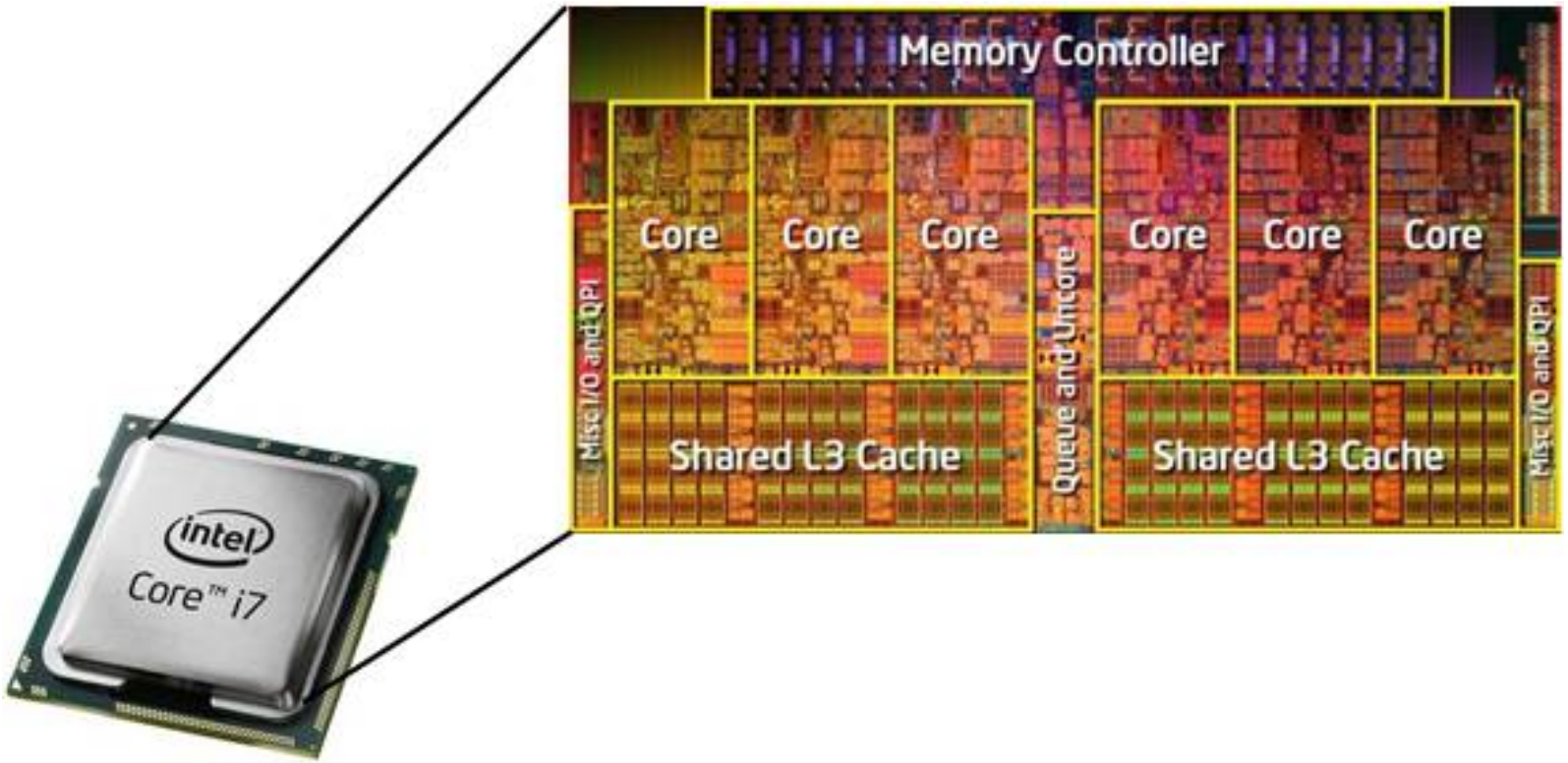

10^7
10^6
10^5
10^4
10^3
10^2
10^1
10^0
1975 1980 1985 1990 1995 2000 2005 2010 2015
Transistors (thousands)
Single-thread Performance (SpecINT)
Frequency (MHz)
Typical Power (Watts)
Number of Cores
Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore

Moore's Law

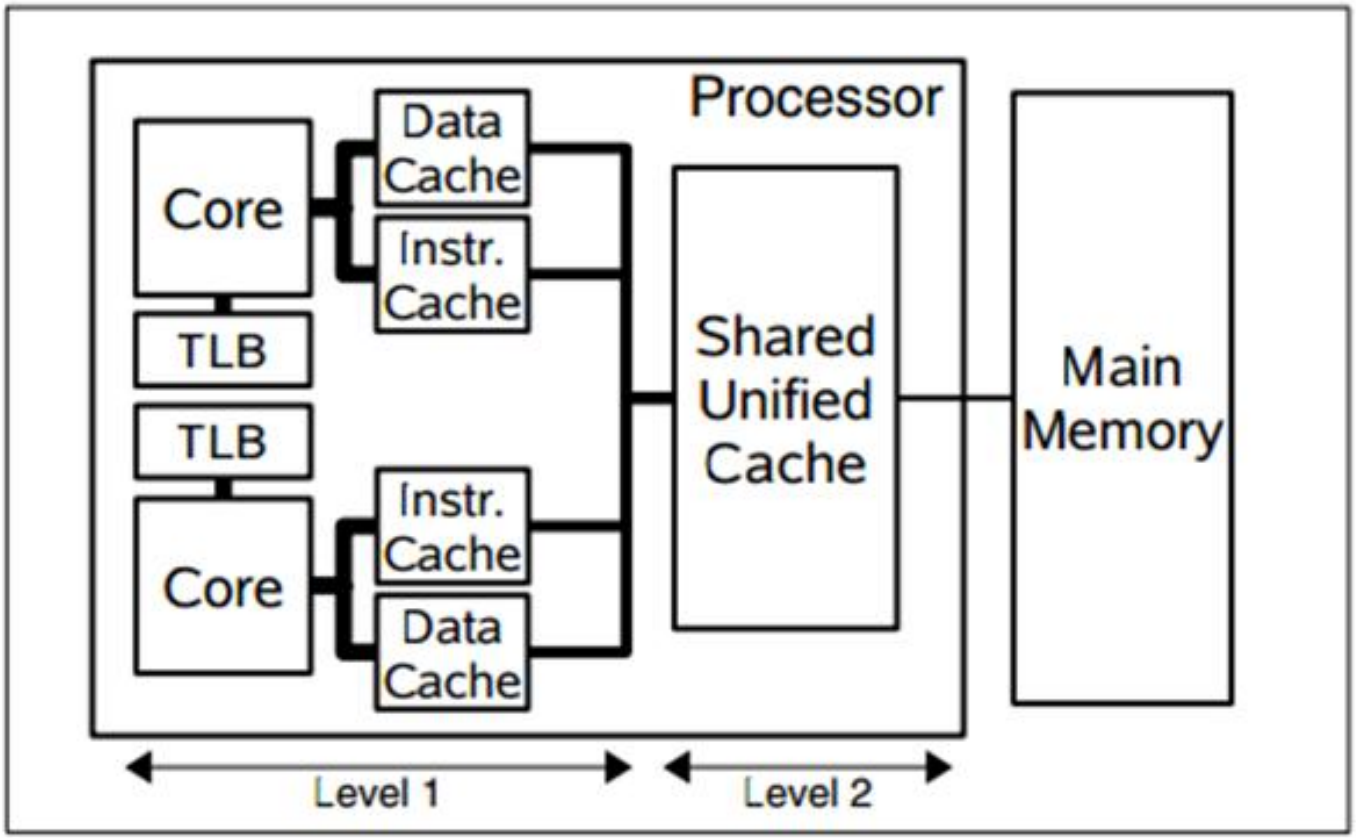


Machine Learning's free ride ended in about 2005.

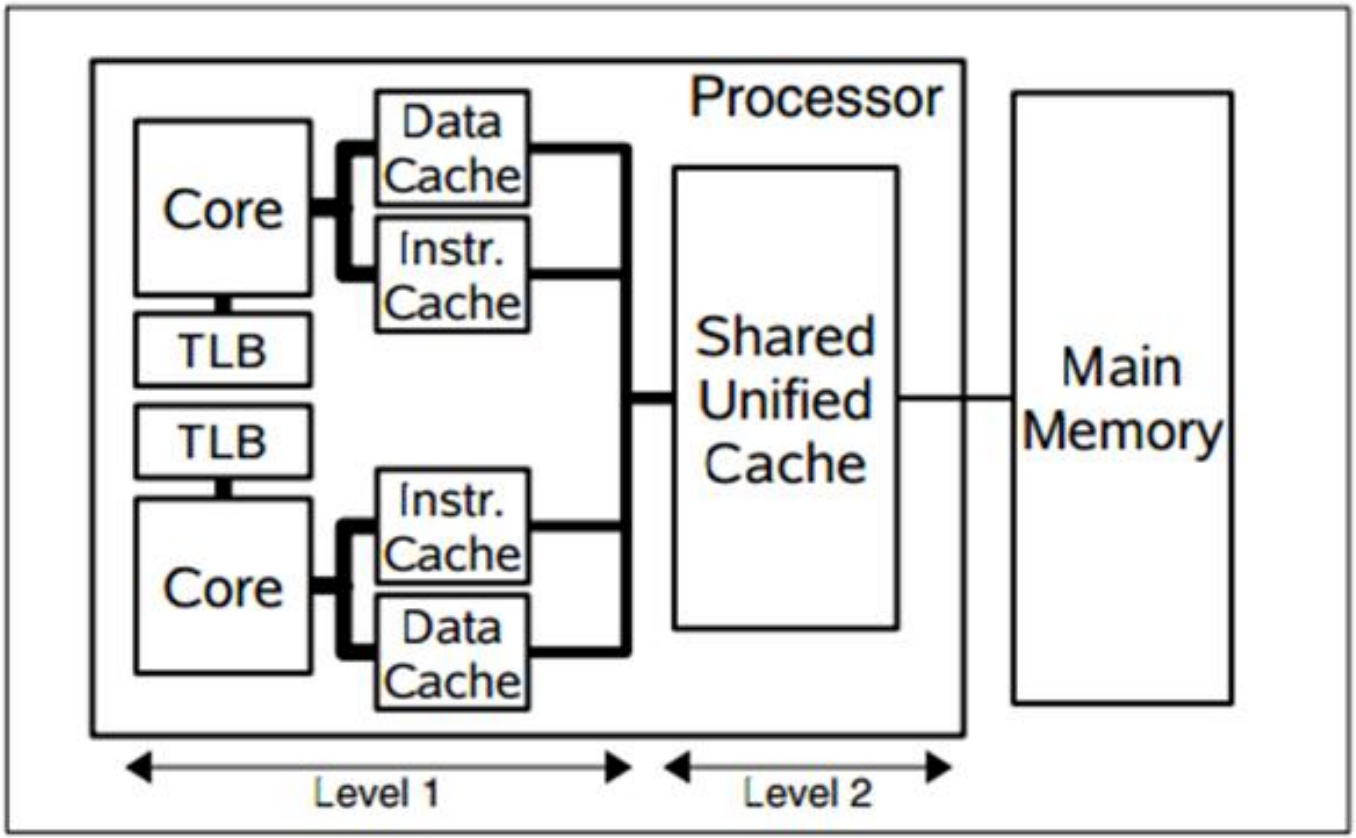
Multi-Core CPUs



Multi-Core CPU Memory Architecture



Multi-Core CPU Memory Architecture

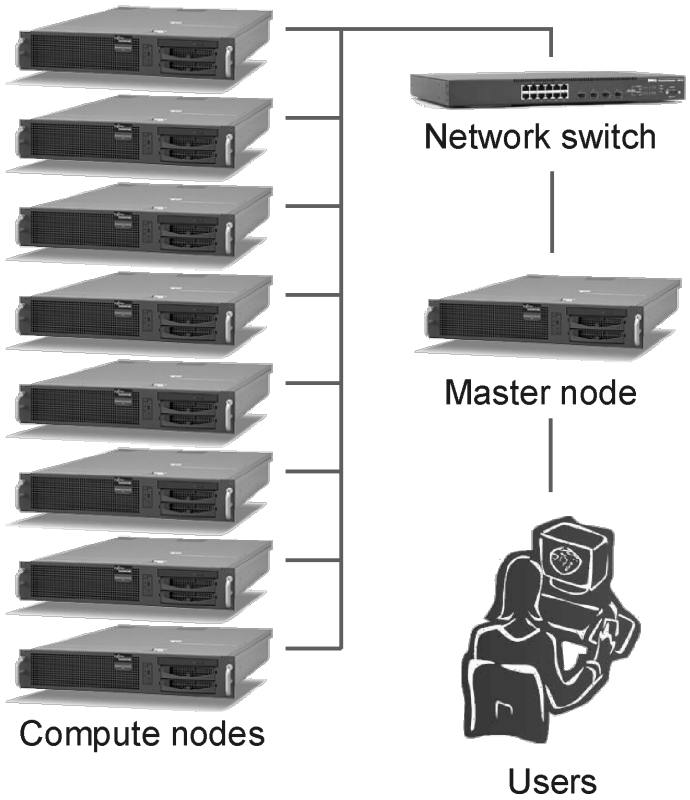


Question: What if one multi-core CPU isn't enough?

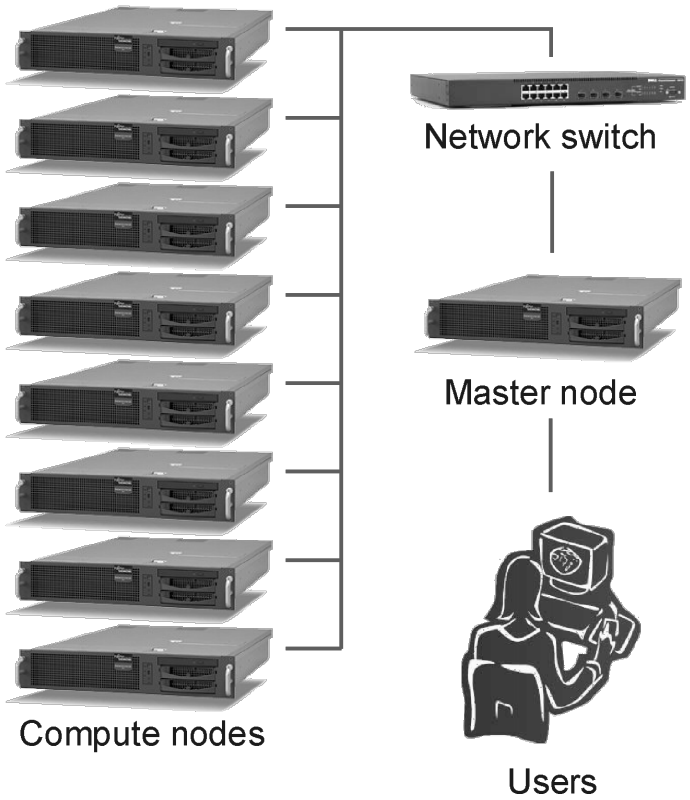
Distributed Computer (Cluster)

Facebook custom-built server rack.

Distributed Computer (Cluster) Architecture

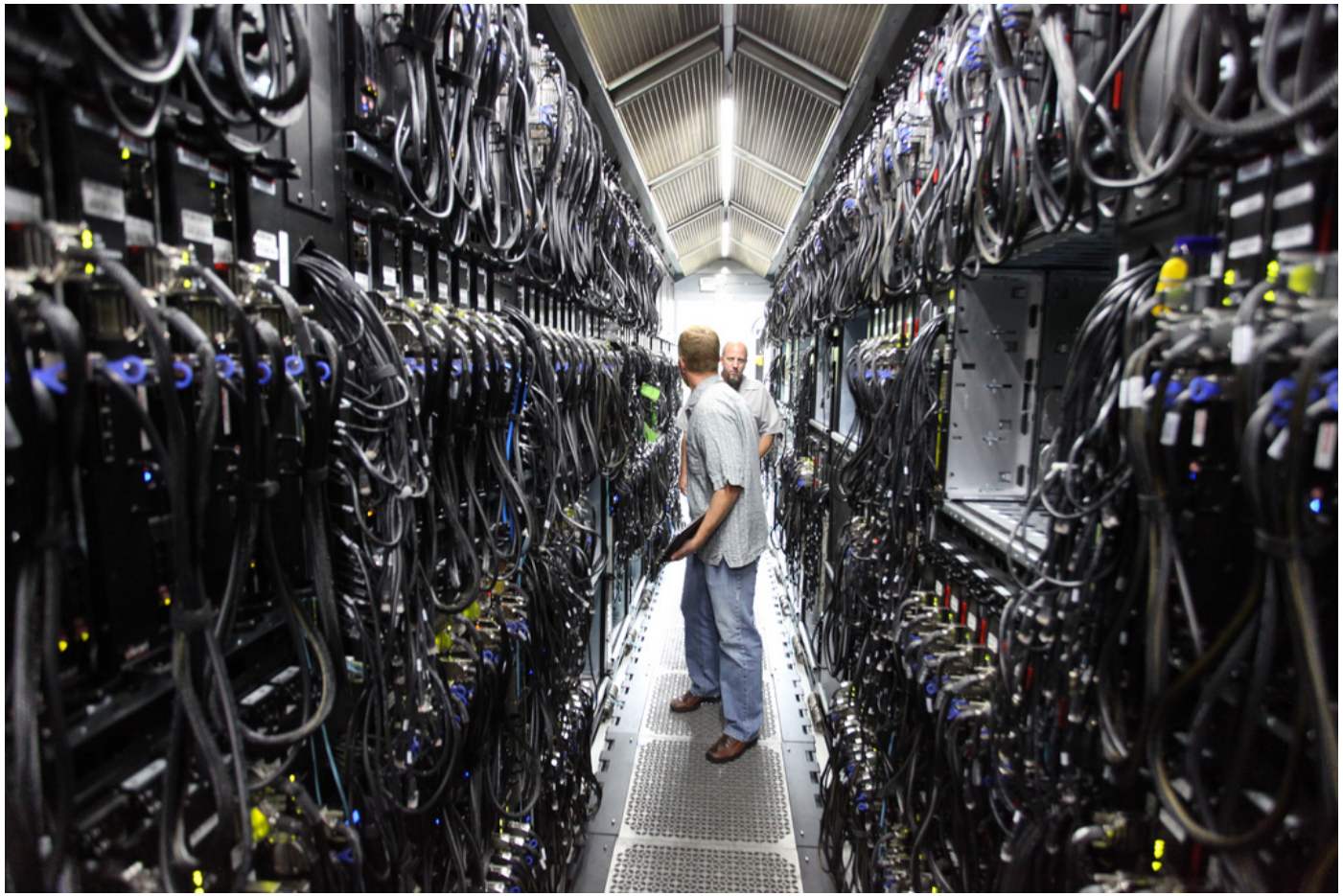


Distributed Computer (Cluster) Architecture



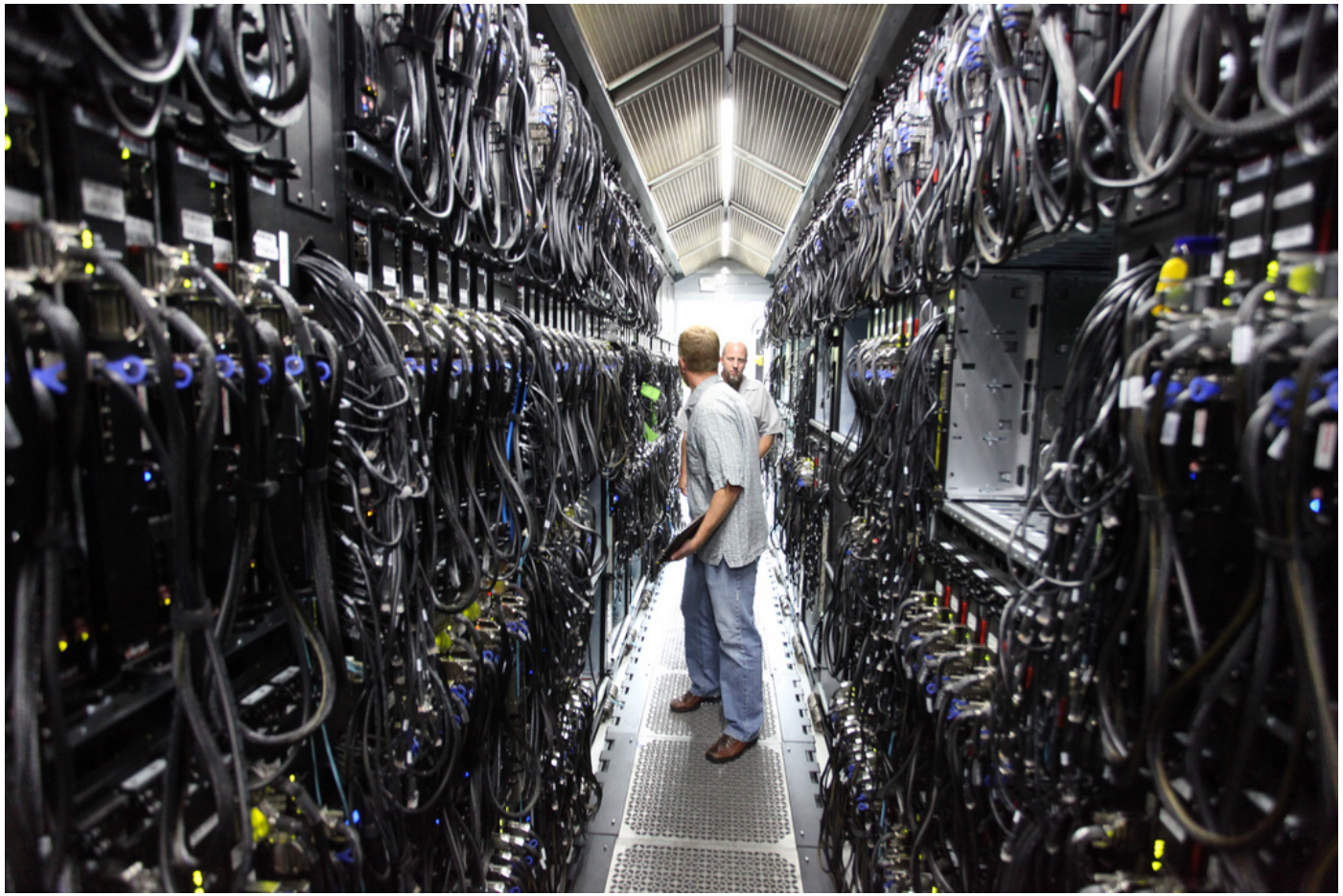
Question: What if a one-rack cluster isn't enough?

Server Container



Server container used by Bing Search.

Server Container



Server container used by Bing Search.

Question: What if a one-container cluster isn't enough?

Container Hanger

Container hanger at Google data center.

https://www.youtube.com/watch?v=zRwPSFpLX8I

Container Hanger

Container hanger at Google data center.

`https://www.youtube.com/watch?v=zRwPSFpLX8I`

Question: What if you don't want to manage a compute center?

Cloud Computing

Infrastructure as a service (IAAS) + Hardware Virtualization

`http://aws.amazon.com/ec2/pricing/`

Outline

Types of Parallelism

- **Shared-Memory Parallel Computing:** Running multiple threads concurrently with access to a single shared memory. Low communication overhead. Limited scalability. Typical of multi-core setting.

Types of Parallelism

- **Shared-Memory Parallel Computing:** Running multiple threads concurrently with access to a single shared memory. Low communication overhead. Limited scalability. Typical of multi-core setting.
- **Distributed Computing:** Running multiple processes on different machines that are networked together. High communication overhead. High scalability. Typical of cluster setting.

Types of Parallel Computing Tasks

- **Embarrassingly Parallel:** Tasks need no communication between them at all. Near linear speedups. Examples:

Types of Parallel Computing Tasks

- **Embarrassingly Parallel:** Tasks need no communication between them at all. Near linear speedups. Examples: cross validation workflows.

Types of Parallel Computing Tasks

- **Embarrassingly Parallel:** Tasks need no communication between them at all. Near linear speedups. Examples: cross validation workflows.
- **Data Parallel:** Each task performs the same operations over a distinct set of data elements. Tasks typically communicate only with a "master" task for synchronization and not with each other. Sub-linear speedups. Examples:

Types of Parallel Computing Tasks

- **Embarrassingly Parallel:** Tasks need no communication between them at all. Near linear speedups. Examples: cross validation workflows.
- **Data Parallel:** Each task performs the same operations over a distinct set of data elements. Tasks typically communicate only with a "master" task for synchronization and not with each other. Sub-linear speedups. Examples: data parallel implementations of any algorithms we've seen to date)

Types of Parallel Computing Tasks

- **Embarrassingly Parallel:** Tasks need no communication between them at all. Near linear speedups. Examples: cross validation workflows.
- **Data Parallel:** Each task performs the same operations over a distinct set of data elements. Tasks typically communicate only with a "master" task for synchronization and not with each other. Sub-linear speedups. Examples: data parallel implementations of any algorithms we've seen to date)
- **Fine-grained Parallel:** Each CPU performs local computation but needs to frequently communicate results with other CPUs. Sub-linear speedups. Examples:

Types of Parallel Computing Tasks

- **Embarrassingly Parallel:** Tasks need no communication between them at all. Near linear speedups. Examples: cross validation workflows.
- **Data Parallel:** Each task performs the same operations over a distinct set of data elements. Tasks typically communicate only with a "master" task for synchronization and not with each other. Sub-linear speedups. Examples: data parallel implementations of any algorithms we've seen to date)
- **Fine-grained Parallel:** Each CPU performs local computation but needs to frequently communicate results with other CPUs. Sub-linear speedups. Examples: physics simulations.

Why the Sub-Linear Scaling?

- The parallel speedup is limited by the amount of time a non-embarrassingly parallel process spends either in single threaded code or communicating with other processes.

Why the Sub-Linear Scaling?

- The parallel speedup is limited by the amount of time a non-embarrassingly parallel process spends either in single threaded code or communicating with other processes.
- Amdahl's Law: Let α be the fraction of code that must run single threaded and P be the number of processors. Then the maximum parallel speedup is approximated by:

$$\frac{1}{\frac{1-\alpha}{P} + \alpha}$$

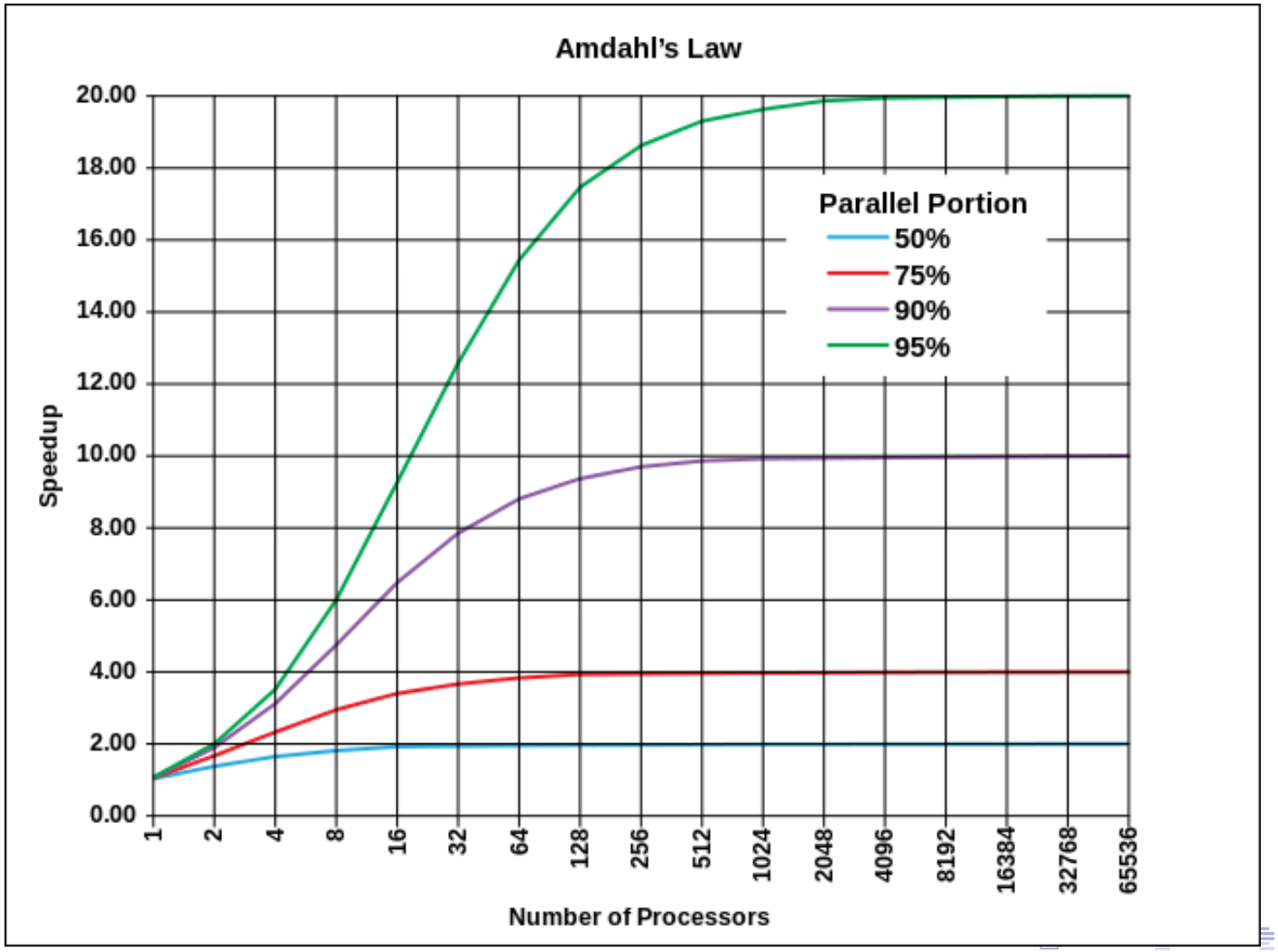
Why the Sub-Linear Scaling?

- The parallel speedup is limited by the amount of time a non-embarrassingly parallel process spends either in single threaded code or communicating with other processes.
- Amdahl's Law: Let α be the fraction of code that must run single threaded and P be the number of processors. Then the maximum parallel speedup is approximated by:

$$\frac{1}{\frac{1-\alpha}{P} + \alpha}$$

- In the limit as P goes to infinity, this ratio converges to $1/\alpha$.

Amdahl's Law


Amdahl's Law
Speedup
20.00
18.00
16.00
14.00
12.00
10.00
8.00
6.00
4.00
2.00
0.00
Parallel Portion
50%
75%
90%
95%
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
Number of Processors

Outline

Imperative Programming

- Imperative programming languages (C, C++, Java, etc.) have multiple problems with concurrency when parallelism is introduced through multi-threading with shared memory.

Imperative Programming

- Imperative programming languages (C, C++, Java, etc.) have multiple problems with concurrency when parallelism is introduced through multi-threading with shared memory.
- Examples: Order-Violation, Atomicity-Violation, Deadlock, ...

Imperative Programming

- Imperative programming languages (C, C++, Java, etc.) have multiple problems with concurrency when parallelism is introduced through multi-threading with shared memory.
- Examples: Order-Violation, Atomicity-Violation, Deadlock, ...
- Writing correct multi-threaded code is highly non-trivial. Since the order of execution of code in threads is basically random, detecting and replicating bugs is extremely difficult.

Imperative Programming

- Imperative programming languages (C, C++, Java, etc.) have multiple problems with concurrency when parallelism is introduced through multi-threading with shared memory.
- Examples: Order-Violation, Atomicity-Violation, Deadlock, ...
- Writing correct multi-threaded code is highly non-trivial. Since the order of execution of code in threads is basically random, detecting and replicating bugs is extremely difficult.
- A key problem in PL is automatically compiling a single-threaded imperative program into a multi-threaded program. No such general purpose compiler exists today.

Functional Programming

- The main problems in multi-threaded imperative programs result from allowing mutable state through global variables accessible to all threads.

Functional Programming

- The main problems in multi-threaded imperative programs result from allowing mutable state through global variables accessible to all threads.
- This is often referred to as the "side effects" problem. In other words, a function can depend on variables other than its explicit inputs, and modify state variables as a result of producing it's output.

Functional Programming

- The main problems in multi-threaded imperative programs result from allowing mutable state through global variables accessible to all threads.
- This is often referred to as the "side effects" problem. In other words, a function can depend on variables other than its explicit inputs, and modify state variables as a result of producing it's output.
- Functional programming eliminates all of these problems by prohibiting mutable state variables and building computations only by nesting function evaluations.

Functional Programming

- The main problems in multi-threaded imperative programs result from allowing mutable state through global variables accessible to all threads.
- This is often referred to as the "side effects" problem. In other words, a function can depend on variables other than its explicit inputs, and modify state variables as a result of producing it's output.
- Functional programming eliminates all of these problems by prohibiting mutable state variables and building computations only by nesting function evaluations.
- There is no explicit looping, only recursion. The component functions are required to always return the same output when run on the same input.

Anonymous Functions

- In functional programming, functions are first-class types that can be passed to other functions as input.

Anonymous Functions

- In functional programming, functions are first-class types that can be passed to other functions as input.
- Some mostly imperative languages like Python also support this.

Anonymous Functions

- In functional programming, functions are first-class types that can be passed to other functions as input.
- Some mostly imperative languages like Python also support this.
- Since functional programs are often specified using many short functions, naming all the functions becomes tedious.

Anonymous Functions

- In functional programming, functions are first-class types that can be passed to other functions as input.
- Some mostly imperative languages like Python also support this.
- Since functional programs are often specified using many short functions, naming all the functions becomes tedious.
- Anonymous functions are simply unnamed functions that can be written as inline expressions.

Anonymous Functions

- In functional programming, functions are first-class types that can be passed to other functions as input.
- Some mostly imperative languages like Python also support this.
- Since functional programs are often specified using many short functions, naming all the functions becomes tedious.
- Anonymous functions are simply unnamed functions that can be written as inline expressions.
- Python supports this using the Lambda Function syntax:

Anonymous Functions

- In functional programming, functions are first-class types that can be passed to other functions as input.
- Some mostly imperative languages like Python also support this.
- Since functional programs are often specified using many short functions, naming all the functions becomes tedious.
- Anonymous functions are simply unnamed functions that can be written as inline expressions.
- Python supports this using the Lambda Function syntax:

```
lambda x,y:  x+y
```

Functional Programming and Parallel Computing

- The advantage of functional programming is that the resulting programs are trivial to parallelize automatically since different arguments to the same function can always be computed in parallel.

Functional Programming and Parallel Computing

- The advantage of functional programming is that the resulting programs are trivial to parallelize automatically since different arguments to the same function can always be computed in parallel.

$$f(g(x), h(x), i(x), j(k(x), l(x)))$$

Functional Programming and Data Parallel Computing

- Functional programming is a natural match for data parallel computing where we want to do things like:

Functional Programming and Data Parallel Computing

- Functional programming is a natural match for data parallel computing where we want to do things like:
  - Apply the same function to all elements in a data set (Map)

Functional Programming and Data Parallel Computing

- Functional programming is a natural match for data parallel computing where we want to do things like:
  - Apply the same function to all elements in a data set (Map)
  - Apply a Boolean filter to select only certain data elements (Filter)

Functional Programming and Data Parallel Computing

- Functional programming is a natural match for data parallel computing where we want to do things like:
  - Apply the same function to all elements in a data set (Map)
  - Apply a Boolean filter to select only certain data elements (Filter)
  - Aggregate a number of data elements by summing, maxing, etc. (Reduce or Fold).

Functional Programming and Data Parallel Computing

- Functional programming is a natural match for data parallel computing where we want to do things like:
  - Apply the same function to all elements in a data set (Map)
  - Apply a Boolean filter to select only certain data elements (Filter)
  - Aggregate a number of data elements by summing, maxing, etc. (Reduce or Fold).
- It turns out that a small number of such easily parallelizable functional programming primitives are sufficient for creating data-parallel implementations of machine learning algorithms.

Data Parallel Computing: Map

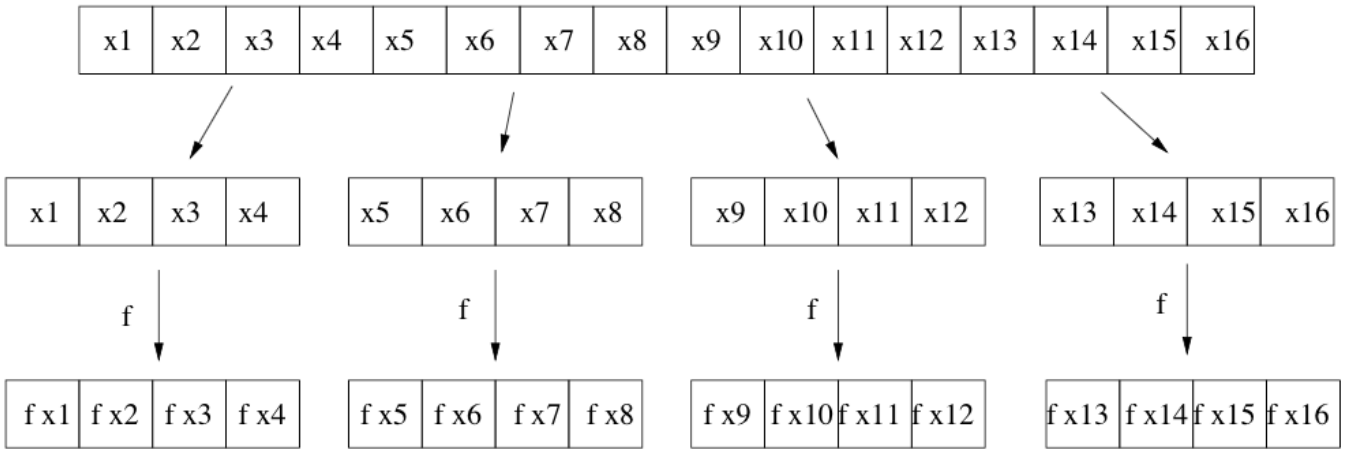
- **Map:** Applies the function f to each value of the array x. This is an embarrassingly parallel operation.

$$Map(f, x) = [f(x_1), ..., f(x_n)]$$

Data Parallel Computing: Map

- **Map:** Applies the function f to each value of the array x. This is an embarrassingly parallel operation.

$$Map(f,x) = [f(x_1),...,f(x_n)]$$



Data Parallel Computing: Reduce

- **Reduce:** Applies the function g to the first two elements of the array x recursively until the input contains a single value, which it then returns.

$$Reduce(g,x) = \begin{cases} Reduce(g,[g(x_1,x_2),x_3,\ldots,x_n]) & \ldots n > 1 \\ x_1 & \text{Otherwise} \end{cases}$$

Data Parallel Computing: Reduce

- **Reduce:** Applies the function g to the first two elements of the array x recursively until the input contains a single value, which it then returns.

$$Reduce(g,x) = \begin{cases} Reduce(g,[g(x_1,x_2),x_3,\ldots,x_n]) & \ldots n > 1 \\ x_1 & \text{Otherwise} \end{cases}$$

- Ex:
  $Reduce(+,[3,10,20,15]) =$

Data Parallel Computing: Reduce

- **Reduce:** Applies the function g to the first two elements of the array x recursively until the input contains a single value, which it then returns.

$$Reduce(g,x) = \begin{cases} Reduce(g,[g(x_1,x_2),x_3,...,x_n]) & ...n > 1 \\ x_1 & \text{Otherwise} \end{cases}$$

- Ex:
$Reduce(+,[3,10,20,15]) = Reduce(+,[(3+10),20,15]) =$

Data Parallel Computing: Reduce

- **Reduce:** Applies the function g to the first two elements of the array x recursively until the input contains a single value, which it then returns.

$$Reduce(g,x) = \begin{cases} Reduce(g,[g(x_1,x_2),x_3,...,x_n]) & ...n > 1 \\ x_1 & \text{Otherwise} \end{cases}$$

- Ex:
$Reduce(+,[3,10,20,15]) = Reduce(+,[(3+10),20,15]) =$
$Reduce(+,[13,20,15]) =$

Data Parallel Computing: Reduce

- **Reduce:** Applies the function g to the first two elements of the array x recursively until the input contains a single value, which it then returns.

$$Reduce(g,x) = \begin{cases} Reduce(g,[g(x_1,x_2),x_3,...,x_n]) & ...n > 1 \\ x_1 & \text{Otherwise} \end{cases}$$

- Ex:
$Reduce(+,[3,10,20,15]) = Reduce(+,[(3+10),20,15]) =$
$Reduce(+,[13,20,15]) = Reduce(+,[(13+20),15]) =$

Data Parallel Computing: Reduce

- **Reduce:** Applies the function g to the first two elements of the array x recursively until the input contains a single value, which it then returns.

$$Reduce(g,x) = \begin{cases} Reduce(g,[g(x_1,x_2),x_3,...,x_n]) & ...n > 1 \\ x_1 & \text{Otherwise} \end{cases}$$

- Ex:
$Reduce(+,[3,10,20,15]) = Reduce(+,[(3+10),20,15]) =$
$Reduce(+,[13,20,15]) = Reduce(+,[(13+20),15]) =$
$Reduce(+,[33,15]) =$

Data Parallel Computing: Reduce

- **Reduce:** Applies the function g to the first two elements of the array x recursively until the input contains a single value, which it then returns.

$$Reduce(g,x) = \begin{cases} Reduce(g,[g(x_1,x_2),x_3,...,x_n]) & ...n > 1 \\ x_1 & \text{Otherwise} \end{cases}$$

- Ex:
$Reduce(+,[3,10,20,15]) = Reduce(+,[(3+10),20,15]) =$
$Reduce(+,[13,20,15]) = Reduce(+,[(13+20),15]) =$
$Reduce(+,[33,15]) = Reduce(+,[(33+15)]) =$

Data Parallel Computing: Reduce

- **Reduce:** Applies the function g to the first two elements of the array x recursively until the input contains a single value, which it then returns.

$$Reduce(g,x) = \begin{cases} Reduce(g,[g(x_1,x_2),x_3,...,x_n]) & ...n > 1 \\ x_1 & \text{Otherwise} \end{cases}$$

- Ex:
$Reduce(+,[3,10,20,15]) = Reduce(+,[(3+10),20,15]) =$
$Reduce(+,[13,20,15]) = Reduce(+,[(13+20),15]) =$
$Reduce(+,[33,15]) = Reduce(+,[(33+15)]) = 48$

Data Parallel Computing: Reduce

- **Reduce:** Applies the function g to the first two elements of the array x recursively until the input contains a single value, which it then returns.

$$Reduce(g,x) = \begin{cases} Reduce(g,[g(x_1,x_2),x_3,...,x_n]) & ...n > 1 \\ x_1 & \text{Otherwise} \end{cases}$$

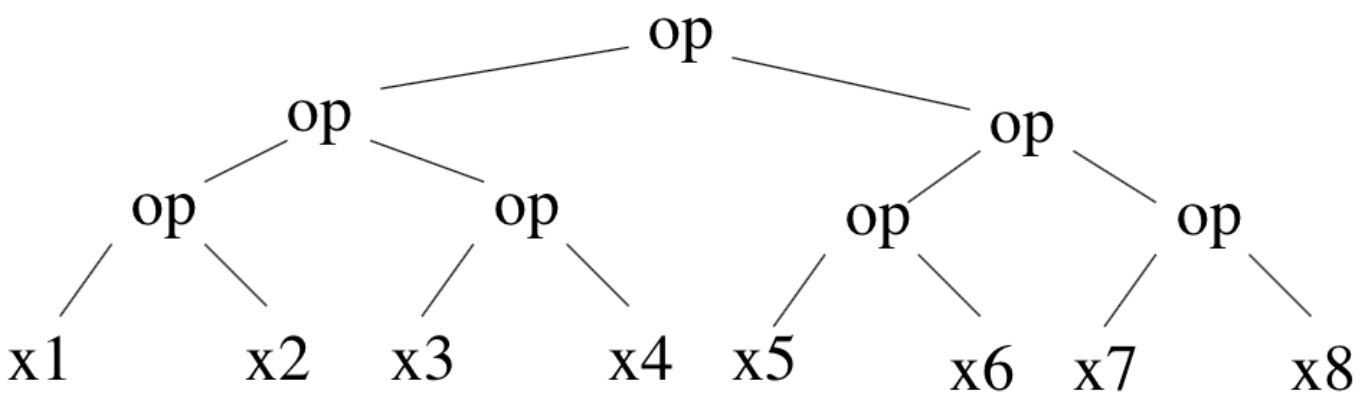
- Ex:
  $Reduce(+,[3,10,20,15]) = Reduce(+,[(3+10),20,15]) =$
  $Reduce(+,[13,20,15]) = Reduce(+,[(13+20),15]) =$
  $Reduce(+,[33,15]) = Reduce(+,[(33+15)]) = 48$
- **Question:** Can we parallelize this computation?

Data Parallel Computing: Reduce

- If the function is associative, this computation can be parallelized using a balanced binary tree.

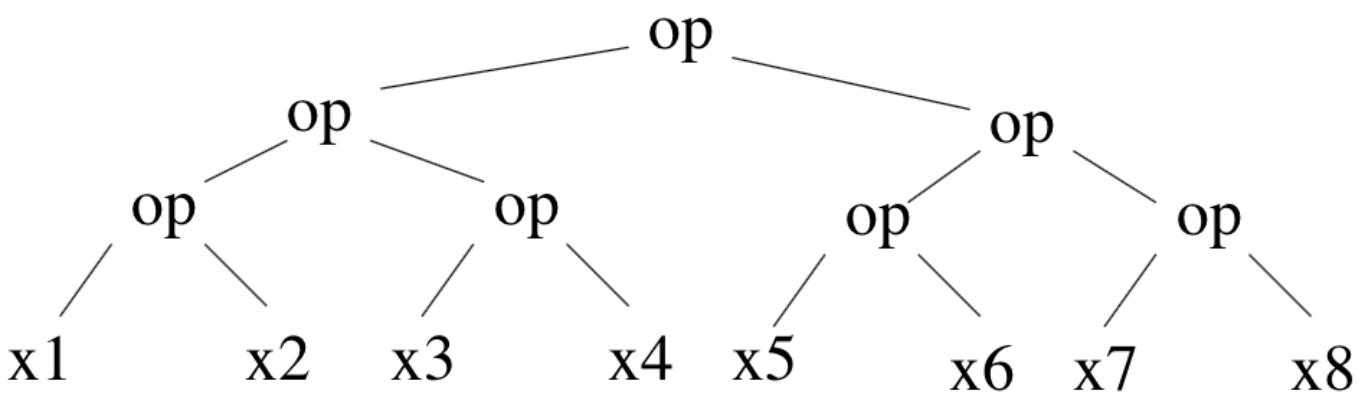
Data Parallel Computing: Reduce

- If the function is associative, this computation can be parallelized using a balanced binary tree.



Data Parallel Computing: Reduce

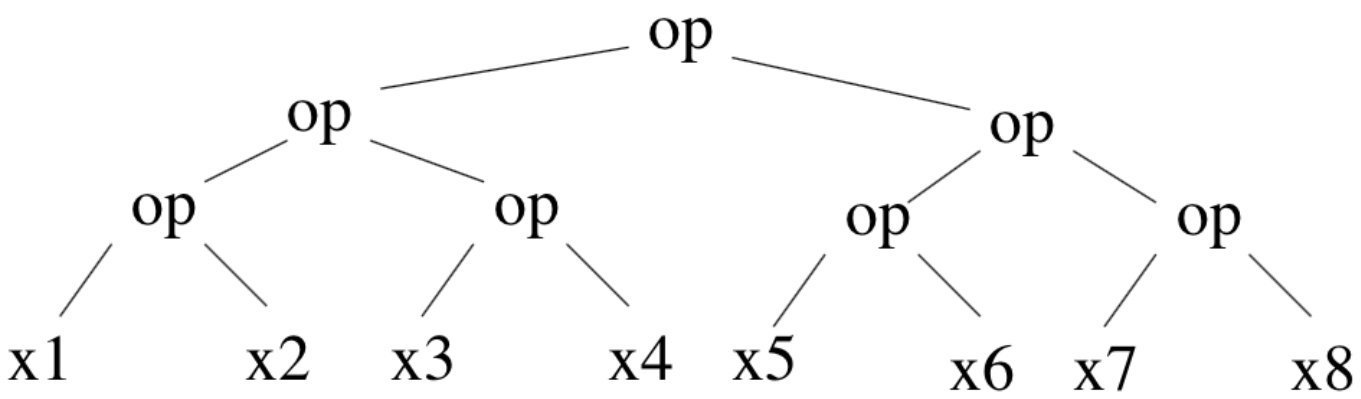
- If the function is associative, this computation can be parallelized using a balanced binary tree.



- Ex: $Reduce(+, [3, 10, 20, 15]) =$

Data Parallel Computing: Reduce

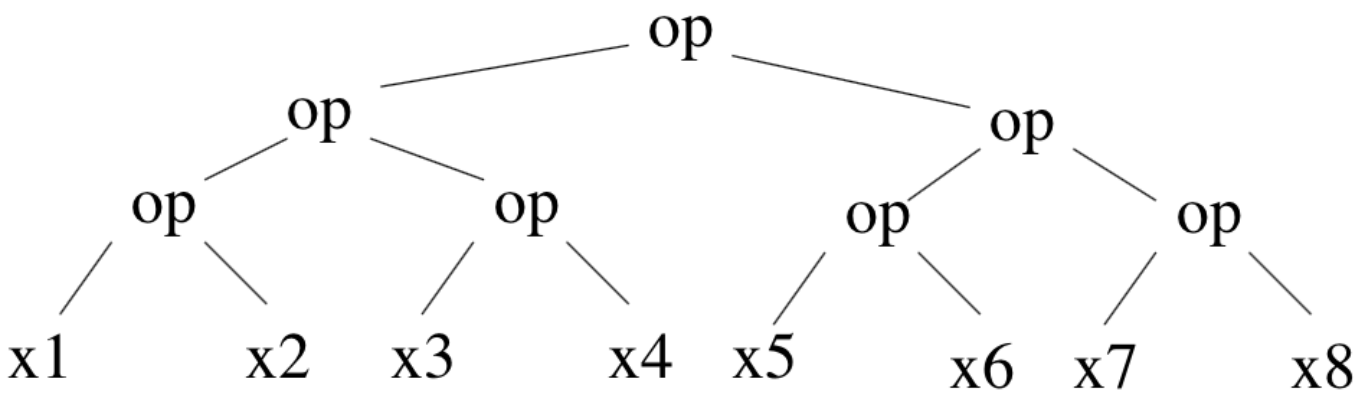
- If the function is associative, this computation can be parallelized using a balanced binary tree.



- Ex: $Reduce(+, [3, 10, 20, 15]) =$
  $Reduce(+, [Reduce(+, [3, 10]), Reduce(+, [20, 15])]) =$

Data Parallel Computing: Reduce

- If the function is associative, this computation can be parallelized using a balanced binary tree.



- Ex: $Reduce(+, [3, 10, 20, 15]) =$
  $Reduce(+, [Reduce(+, [3, 10]), Reduce(+, [20, 15])]) =$
  $Reduce(+, [13, 35]) = 48$