



# Parallelism in Modern C++; from CPU to GPU

Gordon Brown, Michael Wong  
Codeplay C++ Std and SYCL team

CppCon 2018

# Legal Disclaimer

This work represents the view of the author and does not necessarily represent the view of Codeplay.

Other company, product, and service names may be trademarks or service marks of others.

# Acknowledgement Disclaimer

Numerous people internal and external to the original C++/Khronos group, in industry and academia, have made contributions, influenced ideas, written part of this presentations, and offered feedback to form part of this talk.

We even lifted this acknowledgement and disclaimer from some of them.

But we claim all credit for errors, and stupid mistakes. **These are ours, all ours!**

# Codeplay - Connecting AI to Silicon

## Products

### ComputeCpp™

C++ platform via the SYCL™ open standard, enabling vision & machine learning execution on GPU, CPU and NPU

### ComputeAorta™

The heart of Codeplay's compute technology enabling OpenCL™, SPIR™, HSA™ and Vulkan™

## Company

High-performance software solutions for custom heterogeneous systems

Enabling the toughest processor systems with tools and middleware based on open standards

Established 2002 in Scotland

~70 employees



## Addressable Markets

Automotive (ISO 26262)

IoT, Smartphones & Tablets

High Performance Compute (HPC)

Medical & Industrial

**Technologies:** Vision Processing  
Machine Learning  
Artificial Intelligence  
Big Data Compute

## Customers

**RENESAS**

**BROADCOM**

**Imagination**

**QUALCOMM**

**Movidius**

**Partners**  
**INTEL** **AMD**

# Instructors



**Michael Wong**



**Gordon Brown**

# Who are we?

Michael Wong

Current Khronos SYCL chair,  
WG21 SG14, SG5 chair, Directions  
Group, Director of ISO C++  
Head of Canada for Programming  
Languages  
Lead ISO C++ future Heterogeneous  
VP of R & D  
Past Compiler Technical Team lead  
for IBM's C++ 11/14, clang update  
for OpenMP, OpenMP CEO leading  
to accelerators

Gordon Brown

Developer with Codeplay Software  
for 6 years, background in C++  
programming models for  
heterogeneous systems

Worked on ComputeCpp (SYCL)  
since its inception and contributor  
to the Khronos SYCL standard for 6  
years and to C++ executors and  
heterogeneity for 2 years

# But more importantly who are you?

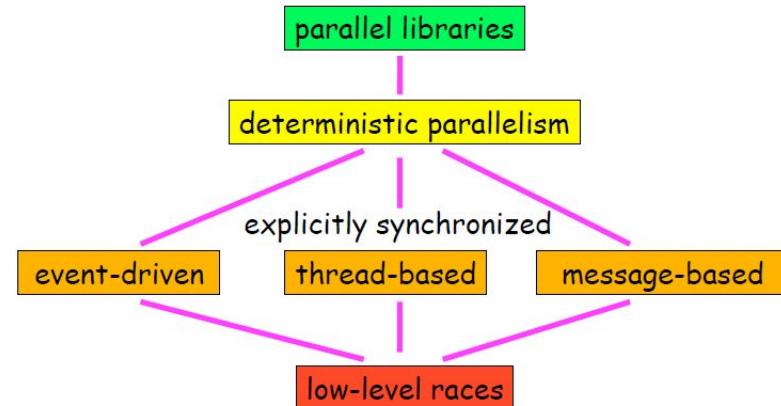
- Hobbyist
  - Programmer
  - Manager
  - Designer
  - Architect
  - Teacher
- What do you want to get out of the class?
- Maybe we can tailor future content to what you need.

# What will you learn in this class?

This class will cover the following topics:

- Fundamentals of parallelism
- Parallel algorithm design
- C++ threading library
- Synchronisation
- CPU & GPU architecture
- SYCL programming model

## Top-Down Concurrency



# This class is interactive

If you have a question just put your hand up

- You can stop us at any time to ask a question or clarify something we've said

We have a number of exercises throughout the class

- Opportunities to what you have learnt into practice
- These are optional, do as much or as little as you want
- We encourage you to experiment, break the code and just see what happens

# Exercises

The exercises will take the form of implementing parallel versions of a number of standard algorithms

The exercises can be found here:

<https://github.com/AerialMantis/cppcon2018-parallelism-class>

The **master** branch of this repo has the exercises and the **solutions** branch has the solutions to the exercises

# Exercises

Most exercises only require a C++17 standard compiler, you can also use <https://godbolt.org/> or <https://wandbox.org/> or <https://coliru.stacked-crooked.com/>

Each of the exercises will provide a stub for the algorithm and some instructions to help you implement it

They will also have a benchmark which will verify your implementation against the standard library

# Setting up ComputeCpp SYCL

Some of the exercises will require you to have ComputeCpp SYCL configured

- You can do this by installing ComputeCpp and the OpenCL drivers on your machine
- Or you can do this by using the docker image we are providing

All of the instructions for this are in the Github README

Wifi

CppCon  
Password is: stepanov

# Schedule

Day 1	Day 2
Welcome	
<b>Chapter 1: Importance of Parallelism</b> <b>Chapter 2: Standard Algorithms</b>	<b>Chapter 8: Data parallelism</b> <b>Chapter 9: Understanding CPU &amp; GPU Architecture</b>
Break	
<b>Exercise 1: Sequential Algorithms</b> <b>Chapter 3: Fundamentals of Parallelism</b>	<b>Chapter 10: C++ for GPUs (part 1)</b> <b>Exercise 3: GPU Algorithms</b>
Lunch	
<b>Chapter 4: Performance Portability</b> <b>Chapter 5: C++ Multi-threaded programming</b>	<b>Chapter 11: C++ for GPUs (part 2)</b>
Break	
<b>Exercise 2: Thread Parallel Algorithms</b> <b>Chapter 6: Synchronization &amp; Advanced Abstraction</b> <b>Chapter 7: Atomics &amp; Memory Model</b>	<b>Exercise 3: GPU Algorithms cont.</b> <b>Chapter 12: Guidelines for Parallel Computing</b> <b>Question &amp; Answer</b>



# Chapter 1: Importance of Parallelism

Prerequisites:

1. Interest in Parallelism and concurrency
2. Nothing else
3. Have fun but want to be challenged

In this chapter:

1. The current landscape of computer architectures and their limitations
2. The performance benefits of parallelism

# Imagine you need to dig a hole...



# But you want to get it done faster...

Let's say you have three options:



# But you want to get it done faster...

Let's say you have three options:

- Simply dig faster with the one shovel you have



# But you want to get it done faster...

Let's say you have three options:

- Simply dig faster with the one shovel you have
- Buy a better shovel that moves more dirt



# But you want to get it done faster...

Let's say you have three options:

- Simply dig faster with the one shovel you have
- Buy a better shovel that moves more dirt
- Hire additional people to help you dig

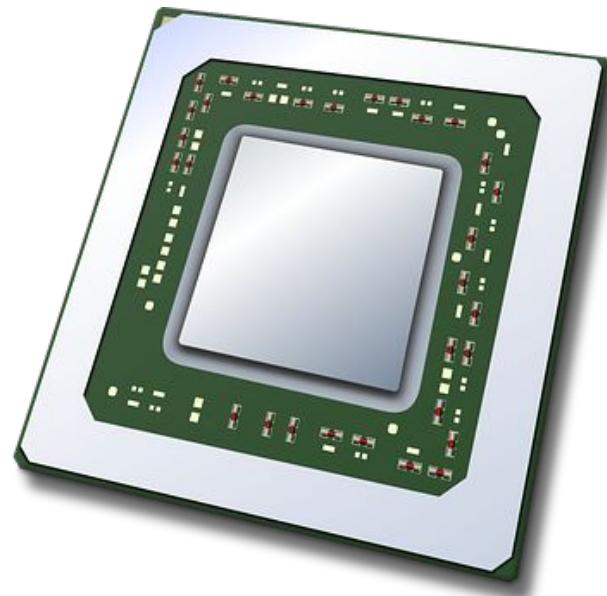


©AMTEC

# This is analogous to processor design



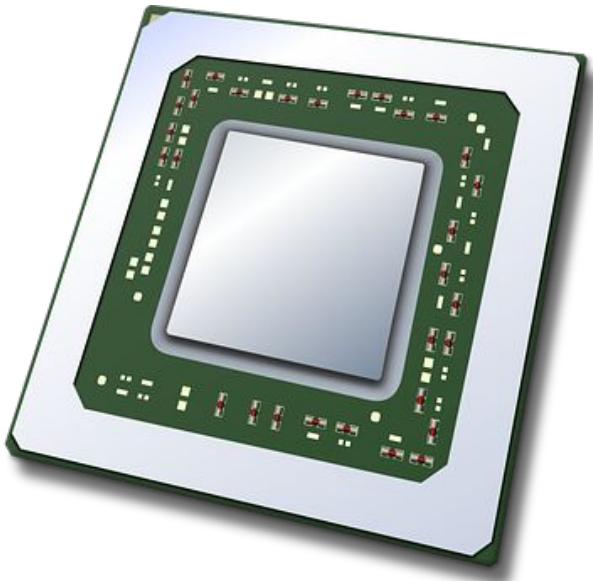
=



# When you need a processor to run faster...

You have three options:

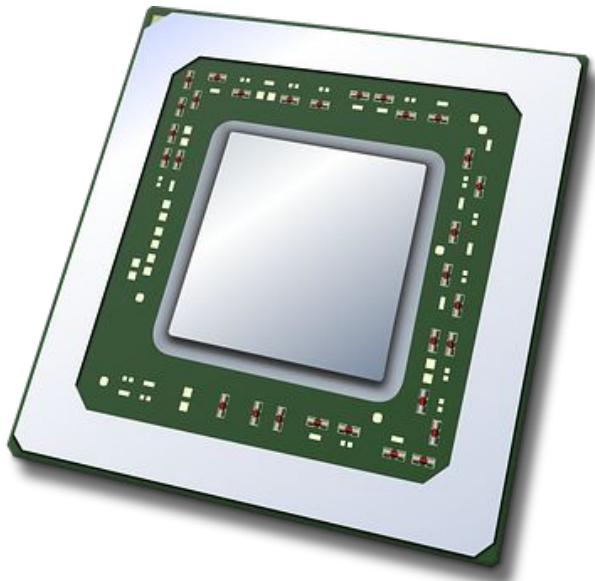
- Run at a higher clock speed



# When you need a processor to run faster...

You have three options:

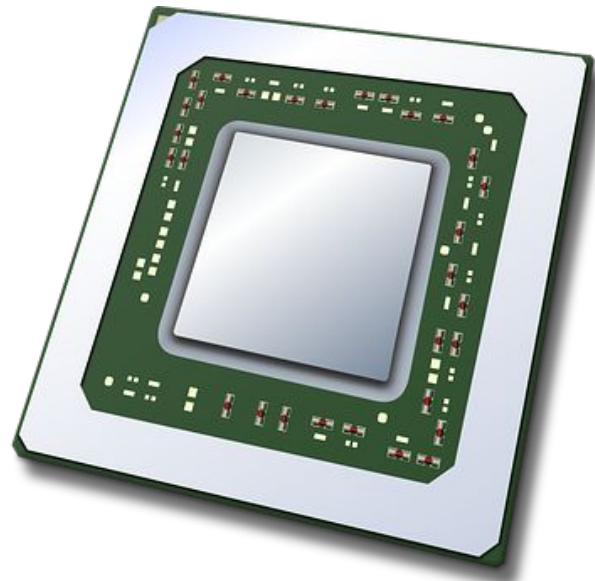
- Run at a higher clock speed
- Do more work on each clock cycle



# When you need a processor to run faster...

You have three options:

- Run at a higher clock speed
- Do more work on each clock cycle
- Have multiple processors work in parallel

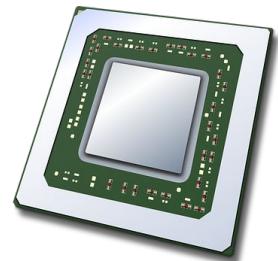


# When you need a processor to run faster...

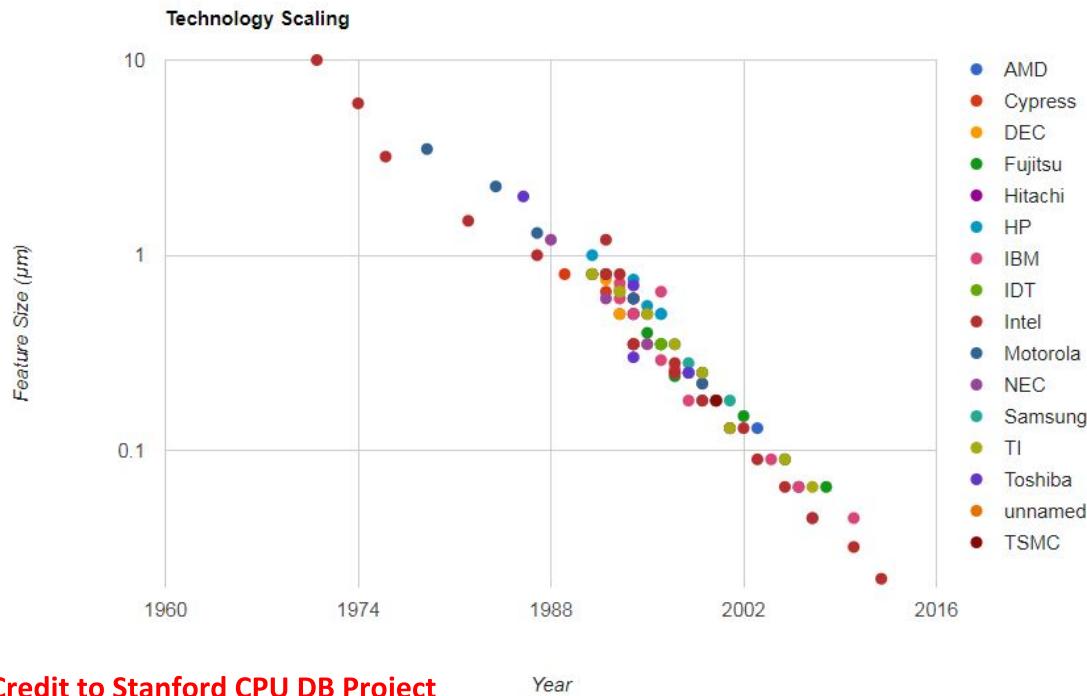
You have three options:

- Run at a higher clock speed
- Do more work on each clock cycle
- Have multiple processors work in parallel

Increasing the clock speed of modern processors increases power consumption

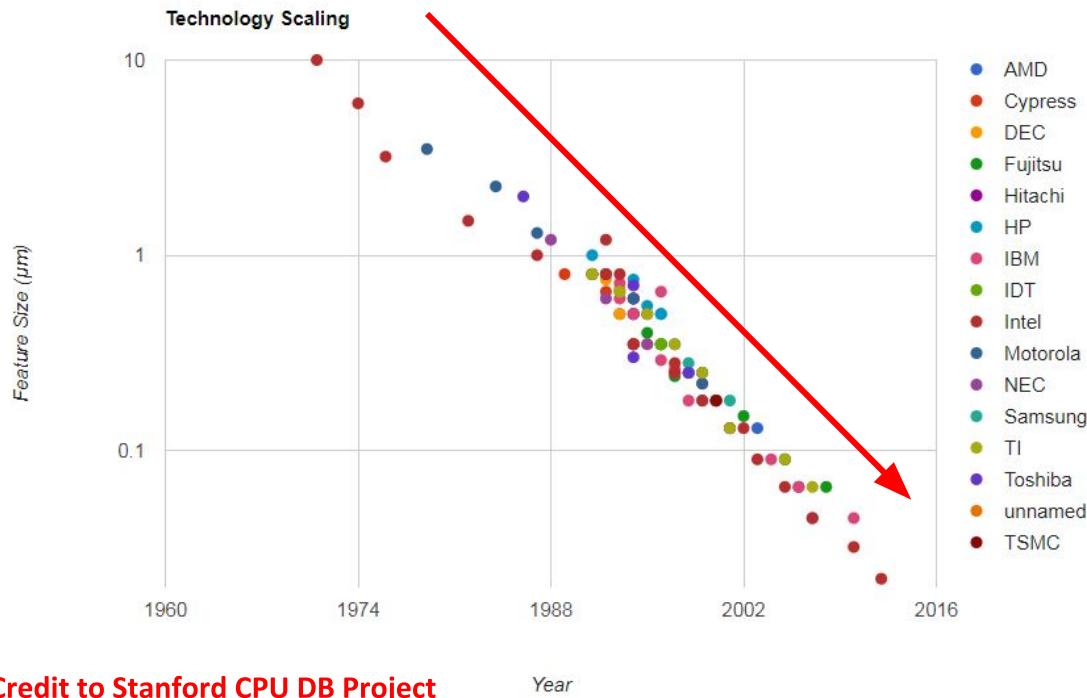


# The problem...



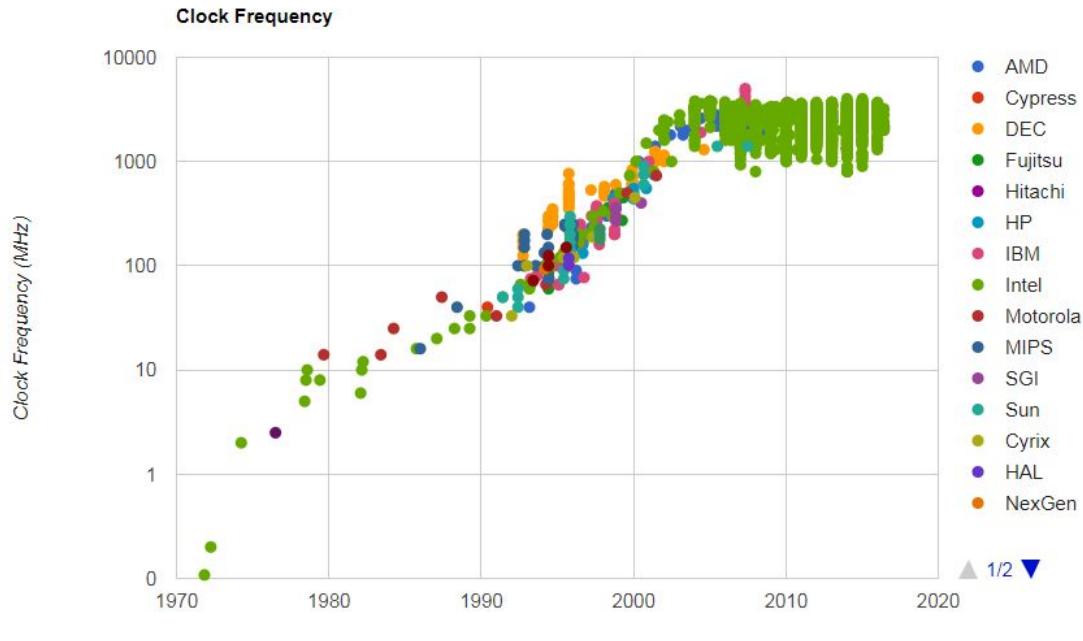
Credit to Stanford CPU DB Project

# The problem...



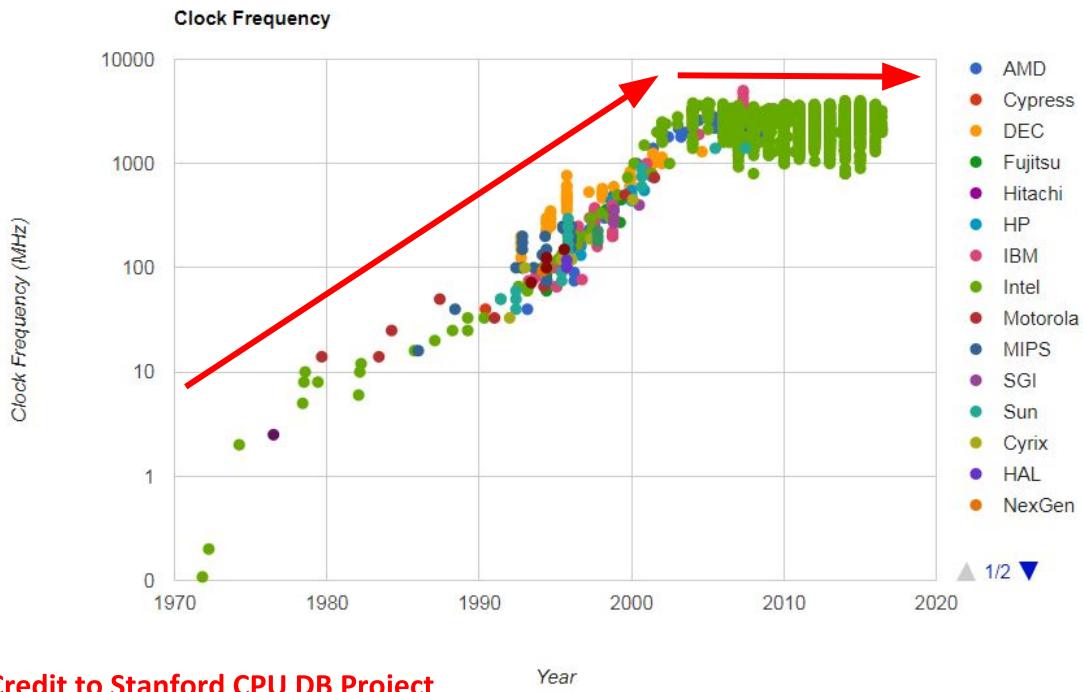
The size of  
transistors are  
continuing to  
decrease

# The problem...



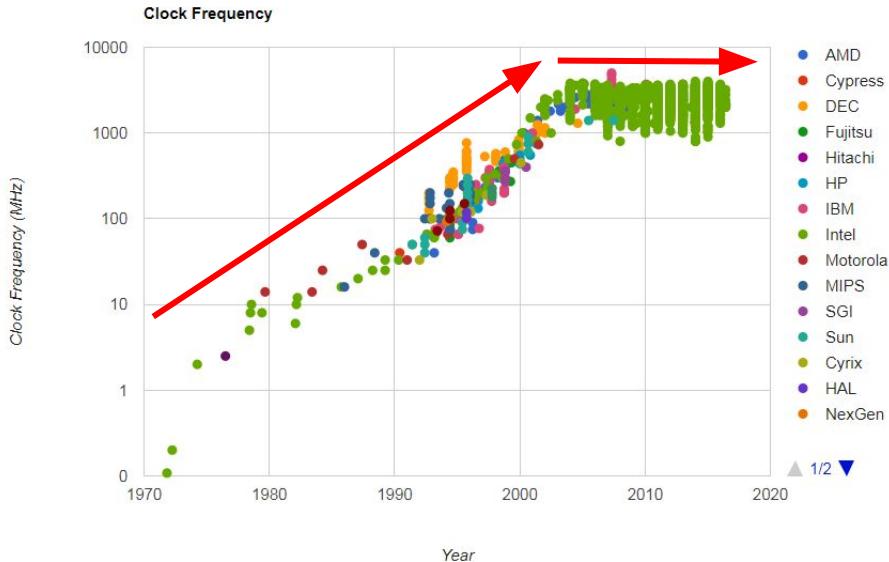
Credit to Stanford CPU DB Project

# The problem...



However in recent years CPU clock speeds have stopped increasing

# CPU clock speeds have stopped increasing despite transistor sizes continuing to decrease, why is this?



# The problem is power

- Fitting more transistors on a single processor
  - Requires more power
  - Which generates more heat
- Cooling these devices becomes the limiting factor in processor design
  - Applies to everything from HPC to mobile devices



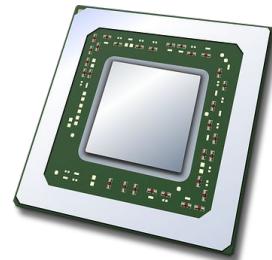
# When you need a processor to run faster...

You have three options:

- Run at a higher clock speed
- Do more work on each clock cycle
- Have multiple processors work in parallel

Making use of more powerful instructions can provide a performance gain

However there's a limit to how much instruction level parallelism can be achieved

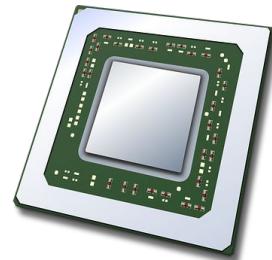


# When you need a processor to run faster...

You have three options:

- Run at a higher clock speed
- Do more work on each clock cycle
- Have multiple processors work in parallel

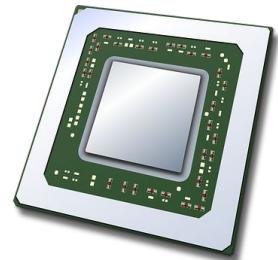
Having a large number of smaller processors execute in parallel can provide additional performance gain



# When you need a processor to run faster...

You have three options:

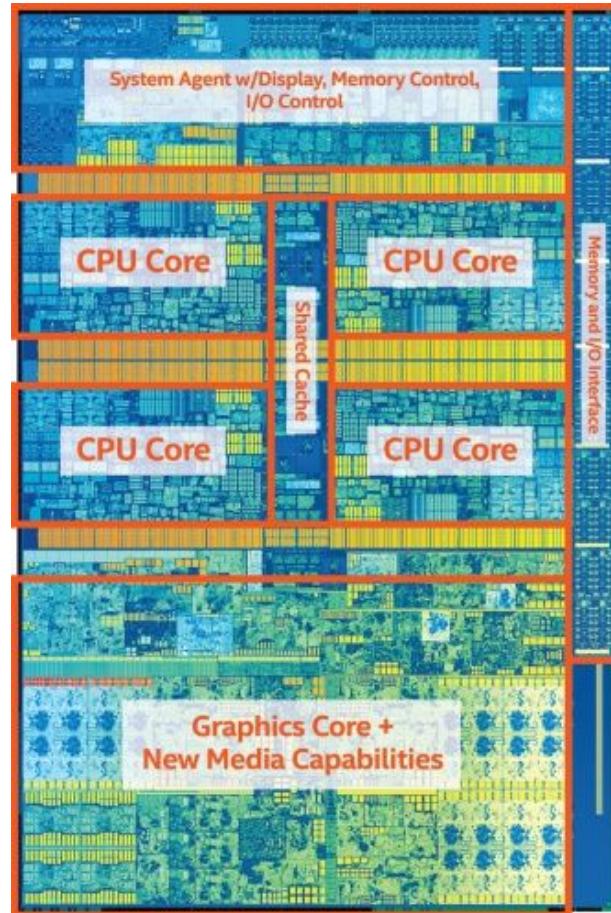
- Run at a higher clock speed
- Do more work on each clock cycle
- Have multiple processors work in parallel



# Take a typical Intel chip...

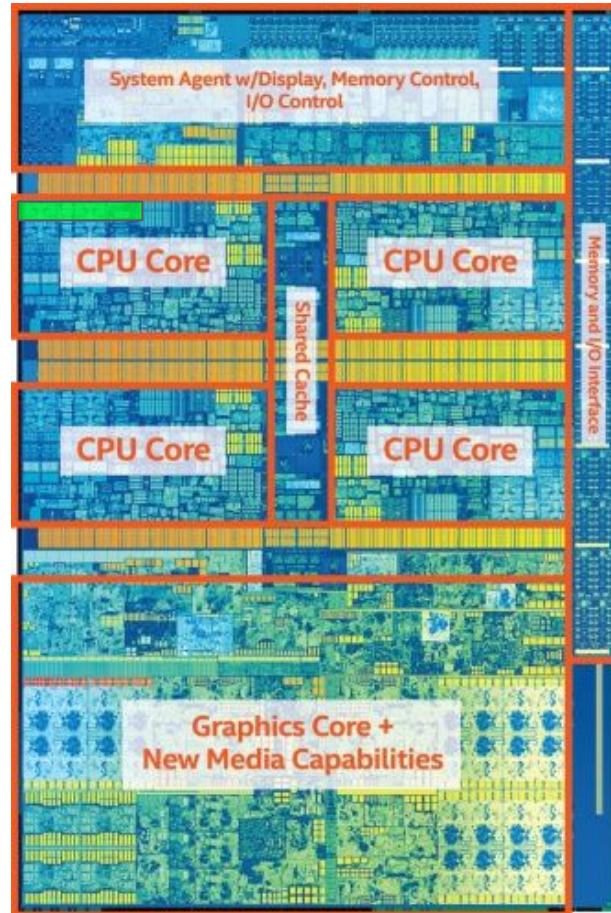
Intel Core i7 7th Gen

- 4x CPU cores
  - Each with hyperthreading
  - Each with 8-wide AVX instructions
- GPU
  - With 1280 processing elements



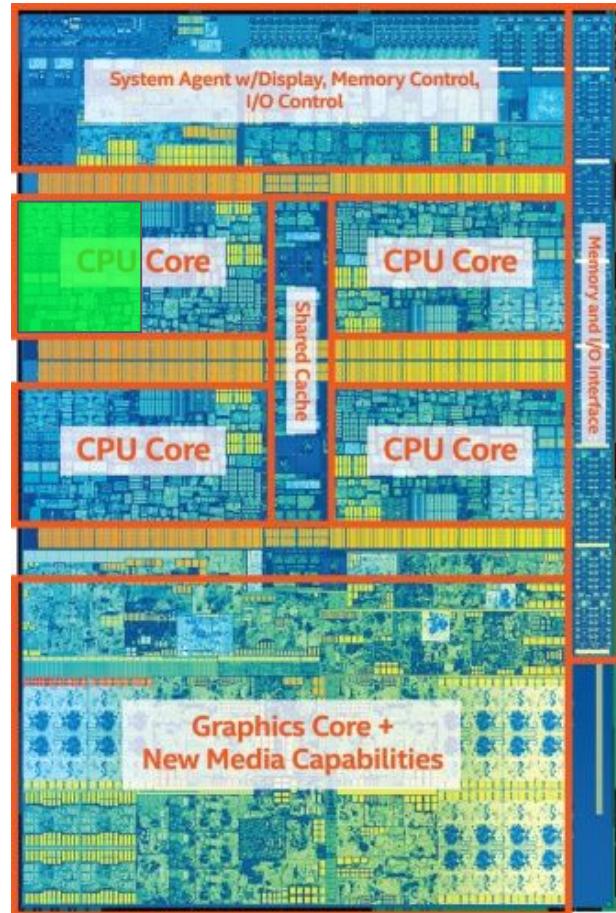
# Take a typical Intel chip...

- Serial C++ code alone only takes advantage of a very small amount of the available resources of the chip



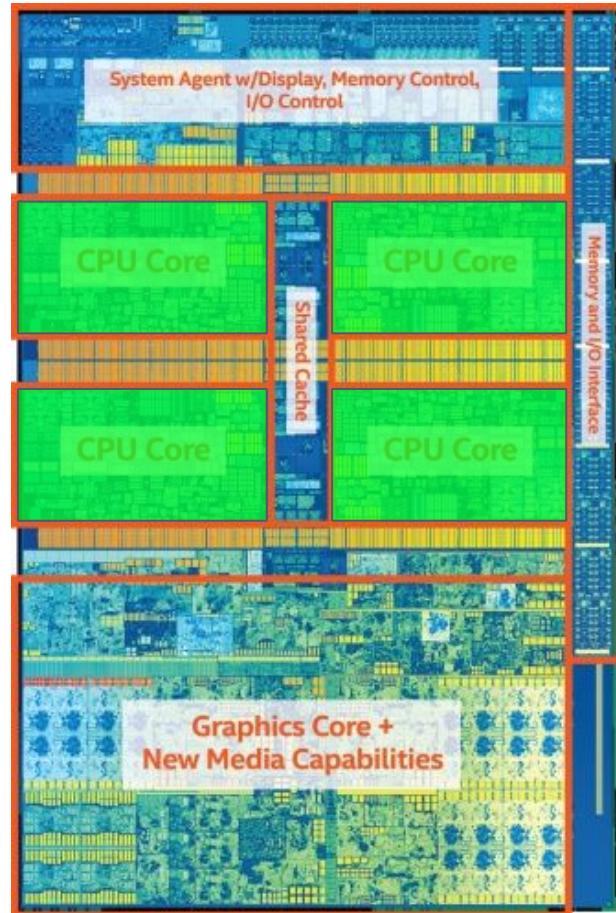
# Take a typical Intel chip...

- Serial C++ code alone only takes advantage of a very small amount of the available resources of the chip
- Using vectorisation allows you to fully utilise the resources of a single hyperthread



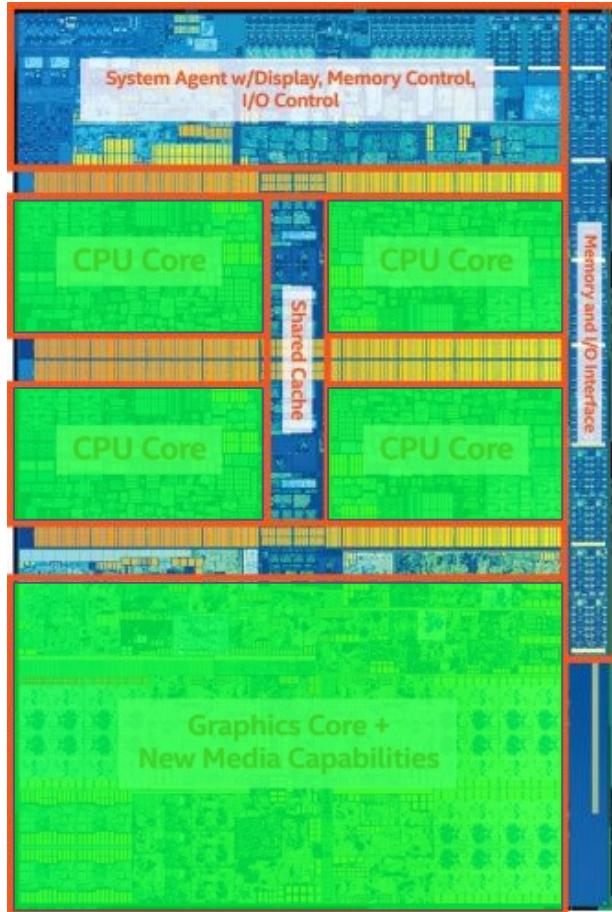
# Take a typical Intel chip...

- Serial C++ code alone only takes advantage of a very small amount of the available resources of the chip
- Using vectorisation allows you to fully utilise the resources of a single hyperthread
- Using multi-threading allows you to fully utilise all CPU cores



# Take a typical Intel chip...

- Serial C++ code alone only takes advantage of a very small amount of the available resources of the chip
- Using vectorisation allows you to fully utilise the resources of a single hyperthread
- Using multi-threading allows you to fully utilise all CPU cores
- **Using heterogeneous dispatch allows you to fully utilise the entire chip**



# So now you have multiple diggers...

How do you manage them?



©AMTEC

# So now you have multiple diggers...

How do you manage them?

- Make sure they all have work to do



# So now you have multiple diggers...

How do you manage them?

- Make sure they all have work to do
- Make sure they are all working efficiently



# So now you have multiple diggers...

How do you manage them?

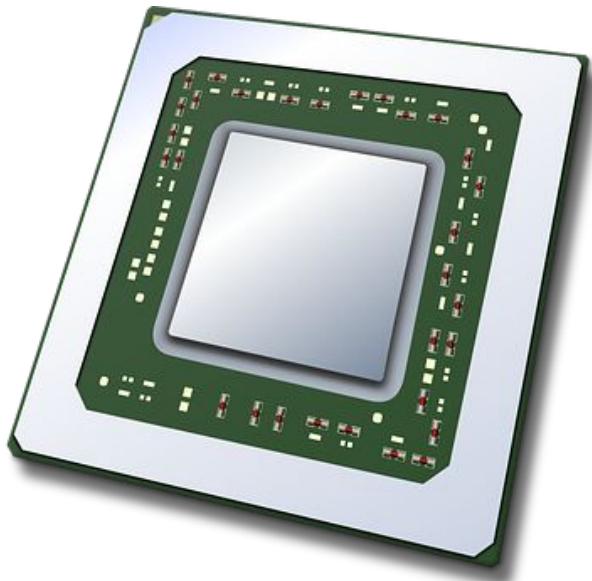
- Make sure they all have work to do
- Make sure they are all working efficiently
- Make sure they don't get in each other's way



# This also applies to parallel processors

How do you manage them?

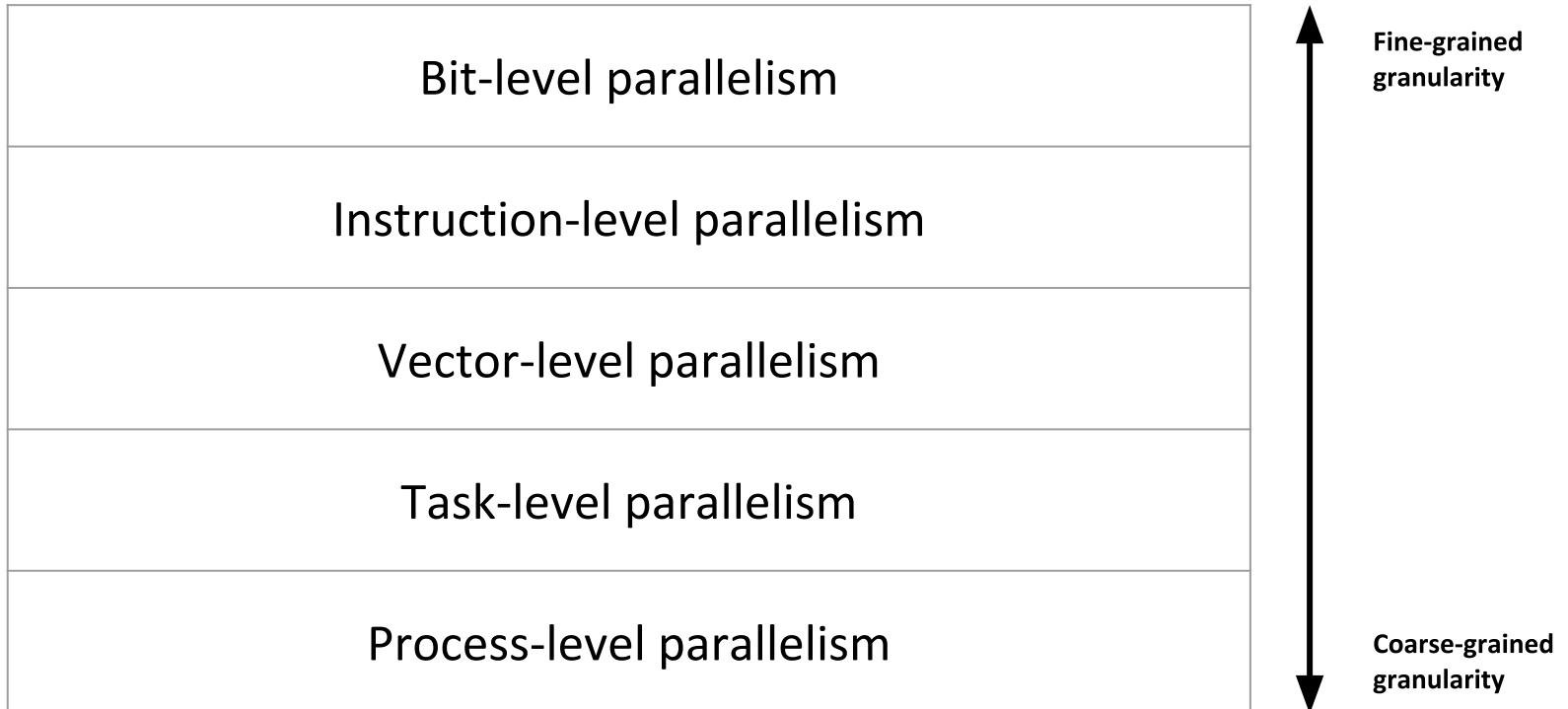
- Make sure they all have work to do
- Make sure they are all working efficiently
- Make sure they don't get in each other's way



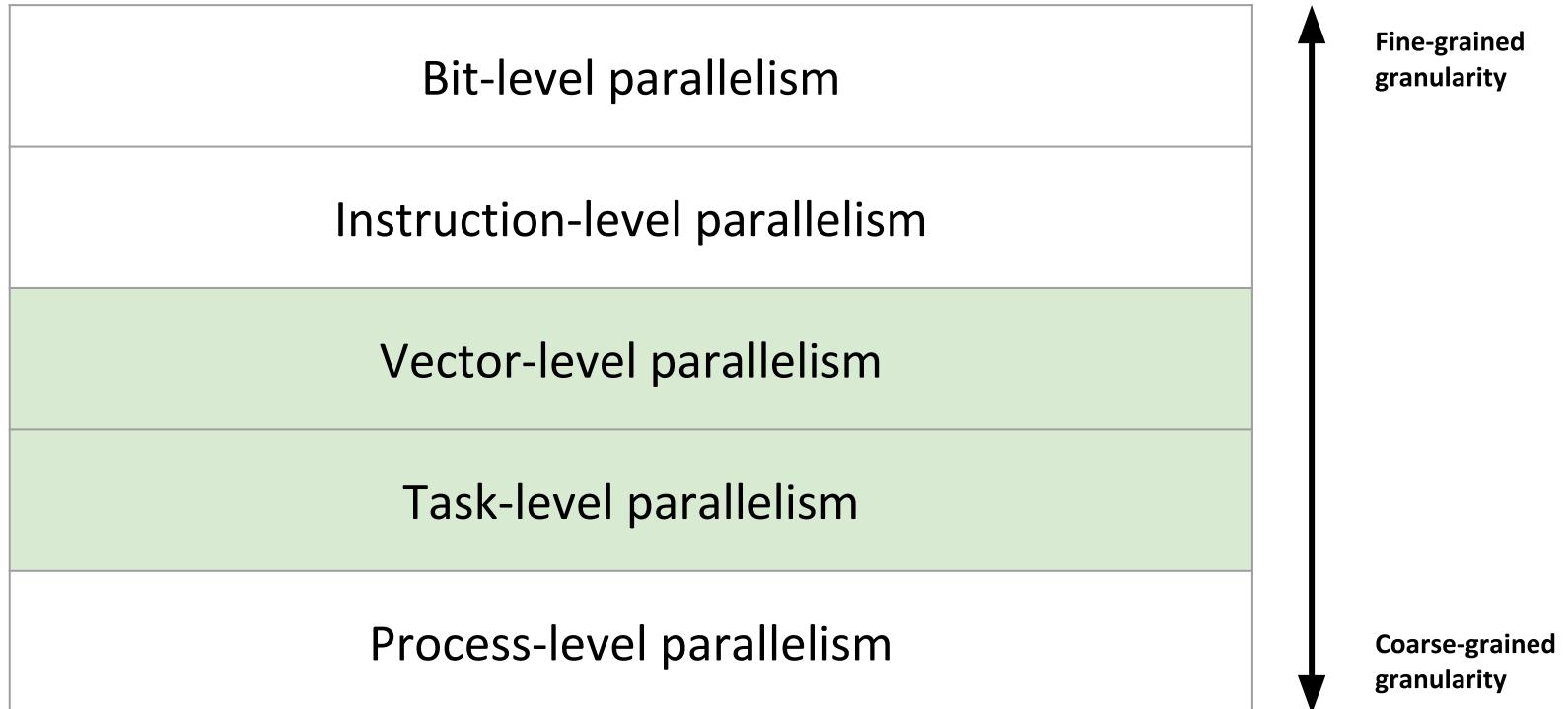
# Parallelism vs concurrency



# The different kinds of parallelism



# The different kinds of parallelism



# The challenge of parallel computing

Parallel computing is about breaking up a problem into smaller tasks and having multiple processors working together to solve a problem

# The challenge of parallel computing

Parallel computing is about breaking up a problem into smaller tasks and having multiple processors working together to solve a problem

The challenge comes in constructing those tasks in such a way that they:

- Make efficient use of the available hardware
- Adhere to the benefits and limitations of the hardware
- Coordinate effectively to complete the work
- Maintaining the correctness of your application

# The challenge of parallel computing

Parallel computing is about breaking up a problem into smaller tasks and having multiple processors working together to solve a problem

The challenge comes in constructing those tasks in such a way that they:

- Make efficient use of the available hardware
- Adhere to the benefits and limitations of the hardware
- Coordinate effectively to complete the work
- Maintaining the correctness of your application

This will be the focus of the class...

# Key takeaways

The clock speeds of CPUs are not getting any faster, so gaining further performance must come from parallelism

Sequential code alone uses a very small fraction of the available resources of a typical chip

The challenge of parallel computing is efficiently coordinating tasks across multiple processes in order to solve a problem



# Chapter 2: Standard Algorithms

## Prerequisites:

1. Previous chapters
2. Familiarity with the C++ STL  
(algorithms & iterators)

## You will learn:

1. Standard algorithms
2. C++17 parallel algorithms
3. Execution policies
4. Evaluating performance of algorithms
5. Fastest way of getting parallel performance

# Patterns

Many of the challenges of parallel computing and their solutions are best represented by common computational patterns

This class will focus on the patterns present in the C++ 11/14 standard algorithms, how those patterns must be adapted to provide opportunities for parallelism, and how to apply those patterns to CPU and GPU hardware

# Standard algorithms

The C++ standard defines algorithms as:

- *“Describes components that C++ programs may use to perform algorithmic operations on containers and other sequences”*

We're going to look at three of these

transform

accumulate

partial\_sum

# transform

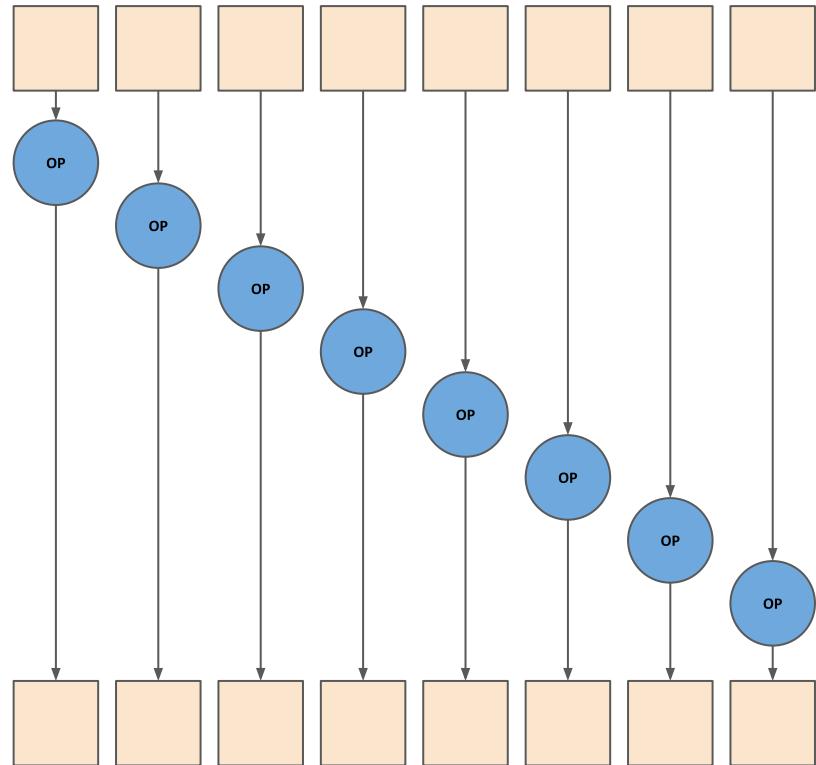
```
d_first transform(first, last,  
                  d_first,  
                  unary_op)
```

```
for each pair of it in [first, last) and d_it in [d_first, last)  
    *d_it = unary_op(*it)  
return d_first
```

# transform

```
d_first transform(first, last,  
                 d_first,  
                 unary_op)
```

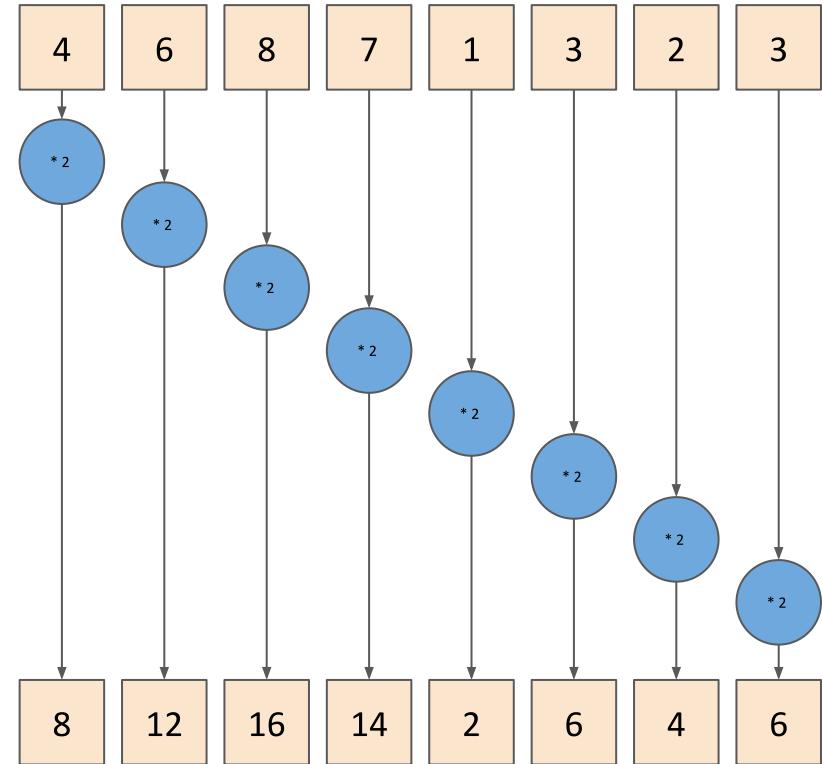
```
for each pair of it in [first, last) and  
d_it in [d_first, last)  
    *d_it = unary_op(*it)  
return d_first
```



# transform

```
d_first transform(first, last,  
                  d_first,  
                  unary_op)
```

```
for each pair of it in [first, last) and  
d_it in [d_first, last)  
    *d_it = unary_op(*it)  
return d_first
```



# accumulate

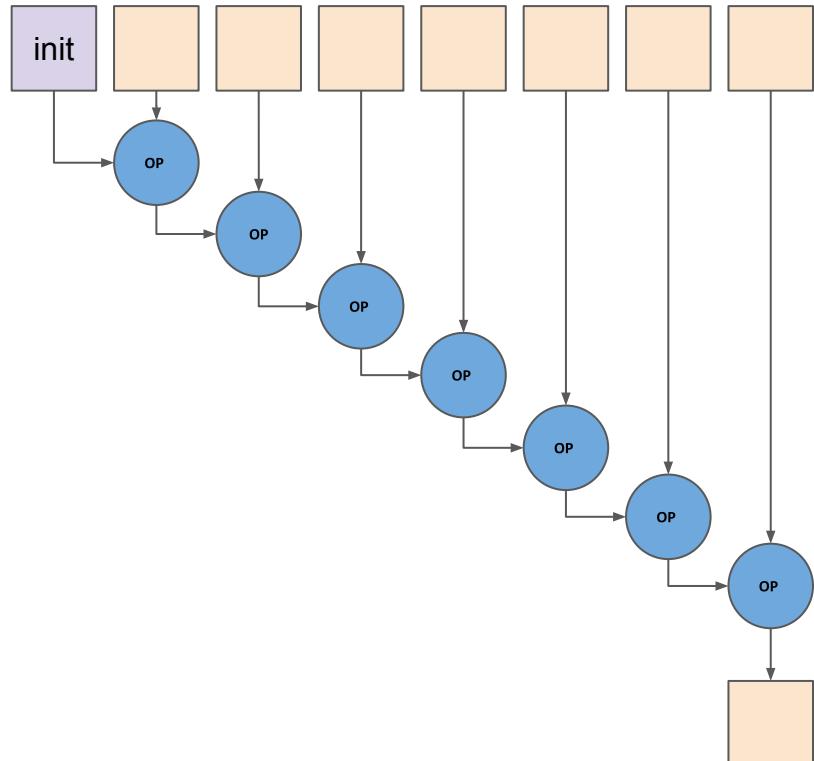
```
result accumulate(first, last,  
                 init,  
                 [binary_op])
```

```
first acc = init  
then for each it in [first, last) in order  
    acc = binary_op(acc, *it)  
return acc
```

# accumulate

```
result accumulate(first, last,  
                 init,  
                 [binary_op])
```

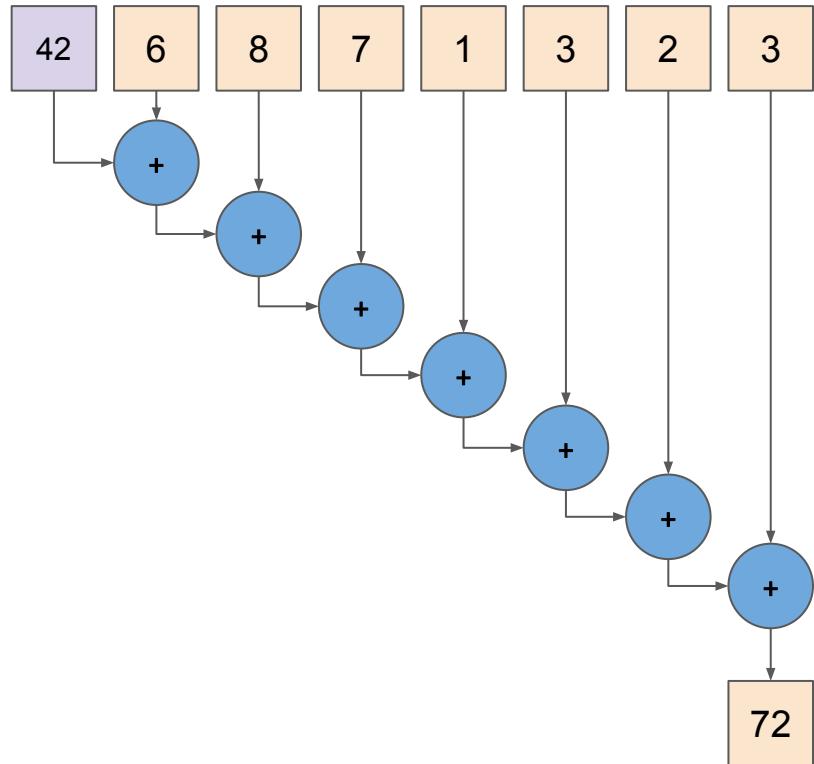
```
first acc = init  
then for each it in [first, last) in order  
    acc = binary_op(acc, *it)  
return acc
```



# accumulate

```
result accumulate(first, last,  
                 init,  
                 [binary_op])
```

```
first acc = init  
then for each it in [first, last) in order  
    acc = binary_op(acc, *it)  
return acc
```



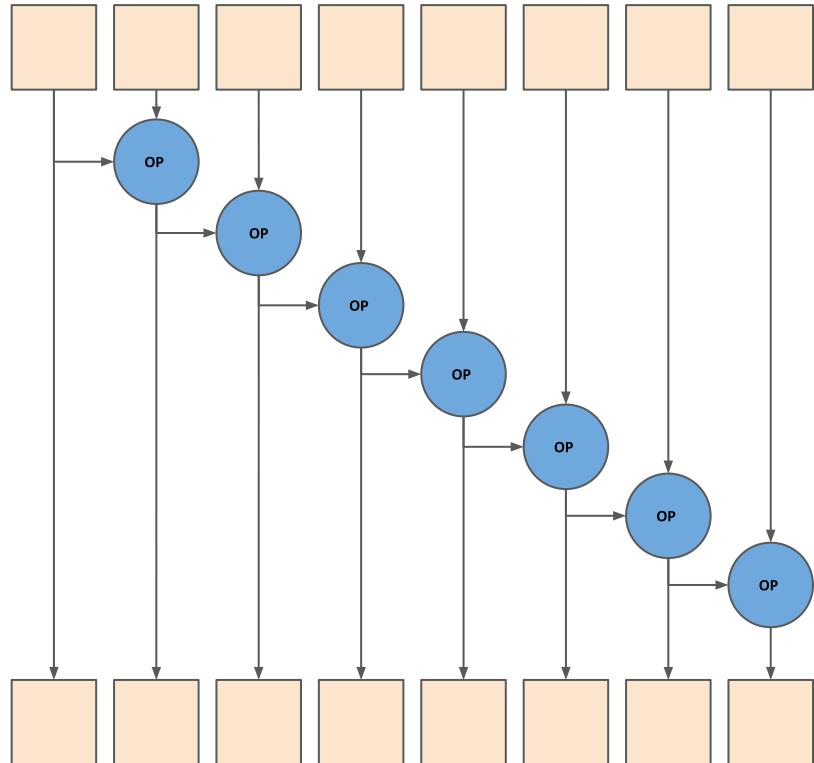
# partial\_sum

```
d_first partial_sum(first, last,
                     d_first,
                     [binary_op])
first sum = *first++
*d_first++ = sum
for each pair of it in [first, last) and d_it in [d_first, last) in order
    sum = binary_op(sum, *it)
    *d_it = sum
return d_first
```

# partial\_sum

```
d_first partial_sum(first, last,
                    d_first,
                    [binary_op])

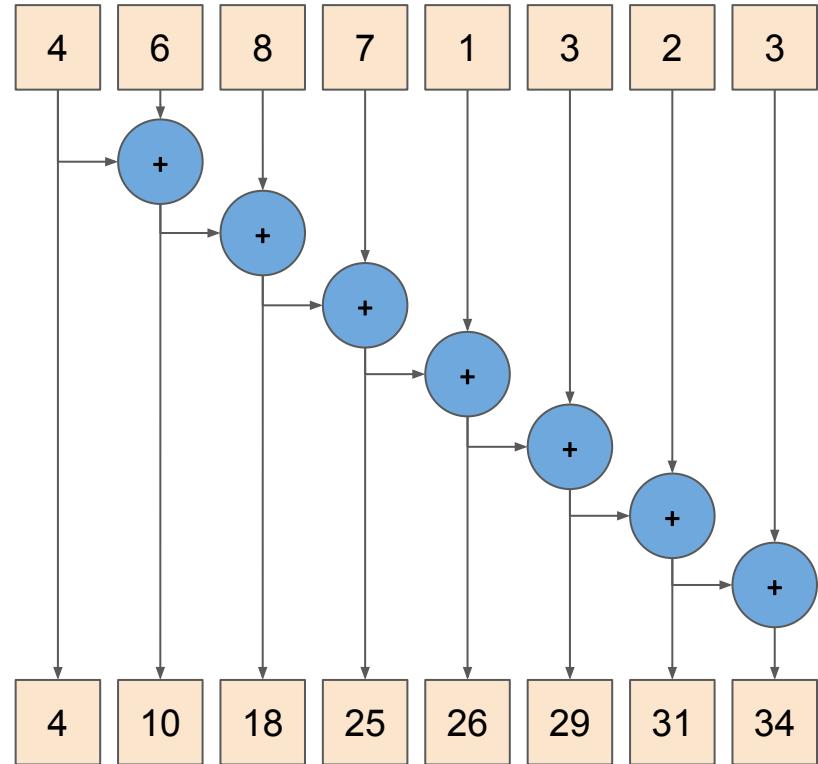
first sum = *first++
*d_first++ = sum
for each pair of it in [first, last) and
d_it in [d_first, last) in order
    sum = binary_op(sum, *it)
    *d_it = sum
return d_first
```



# partial\_sum

```
d_first partial_sum(first, last,
                    d_first,
                    [binary_op])

first sum = *first++
*d_first++ = sum
for each pair of it in [first, last) and
d_it in [d_first, last) in order
    sum = binary_op(sum, *it)
    *d_it = sum
return d_first
```



# Moving from sequential to parallel algorithm

- **Guideline:**
  - Thread Programming  $\Leftrightarrow$  assembly language of Parallel programming
  - Algorithm & Pattern  $\Leftrightarrow$  High level Parallel Programming
- The fastest way to get good speedup in C++17
  - Use parallel algorithms

# C++17 parallel algorithms

C++17 introduces a number of parallel algorithms and new execution policies which dictate how they are parallelized

There are three kinds of new algorithms introduced:

- Overloads for most algorithms taking an execution policy
- New unordered versions of existing ordered algorithms
- New fused algorithms

# Execution policies

An execution policy describes how a generic algorithm can be parallelized

They allow programmers to request parallelism with a particular set of constraints

# Standard execution policies

- `sequenced_execution_policy` (`seq`)
  - Operations are indeterminately sequenced in the calling thread
  - Do not parallelize
- `parallel_execution_policy` (`par`)
  - Operations are indeterminately sequenced with respect to each other within the same thread
  - Parallelize but don't vectorise
- `parallel_unsequenced_execution_policy` (`par_unseq`)
  - Operations are unsequenced with respect to each other and possibly interleaved
  - Parallelize and may vectorise

# Parallel versions of existing algorithms

adjacent_difference	for_each[_n]	merge	remove[_copy _copy_if _if]	stable_partition
adjacent_find	generate[_n]	min_element	replace[_copy _copy_if _if]	stable_sort
all_of	includes	minmax_element	reverse[_copy]	swap_ranges
any_of	implace_merge	mismatch	rotate[_copy]	transform
copy[_if _n]		move	search[_n]	
count[_if]	is_heap[_until]	none_of	set_difference	
equal	is_partitioned	nth_element	set_intersection	
	is_sorted[_until]	partial_sort[_copy]	set_symmetric_difference	uninitialized_copy[_n]
fill[_n]	lexicographical_compare	partition[_copy]	set_union	uninitialized_fill[_n]
find[_end _first_of _if _if_not]	max_element		sort	unique[_copy]

# Parallel versions of existing algorithms

adjacent_difference	<code>for_each[_n]</code>	merge	<code>remove[_copy _copy_if _if]</code>	stable_partition
adjacent_find	<code>generate[_n]</code>	min_element	<code>replace[_copy _copy_if _if]</code>	stable_sort
all_of	<code>includes</code>	minmax_element	<code>reverse[_copy]</code>	swap_ranges
any_of	<code>implace_merge</code>	mismatch	<code>rotate[_copy]</code>	transform
copy[_if _n]		move	<code>search[_n]</code>	
count[_if]	<code>is_heap[_until]</code>	none_of	<code>set_difference</code>	
equal	<code>is_partitioned</code>	nth_element	<code>set_intersection</code>	
	<code>is_sorted[_until]</code>	<code>partial_sort[_copy]</code>	<code>set_symmetric_difference</code>	<code>uninitialized_copy[_n]</code>
fill[_n]	<code>lexicographical_compare</code>	<code>partition[_copy]</code>	<code>set_union</code>	<code>uninitialized_fill[_n]</code>
<code>find[_end _first_of _if _if_not]</code>	<code>max_element</code>		<code>sort</code>	<code>unique[_copy]</code>

# New unordered algorithms

adjacent_difference	for_each[_n]	merge	remove[_copy _copy_if _if]	stable_partition
adjacent_find	generate[_n]	min_element	replace[_copy _copy_if _if]	stable_sort
all_of	includes	minmax_element	reverse[_copy]	swap_ranges
any_of	implace_merge	mismatch	rotate[_copy]	transform
copy[_if _n]	inclusive_scan	move	search[_n]	
count[_if]	is_heap[_until]	none_of	set_difference	
equal	is_partitioned	nth_element	set_intersection	
exclusive_scan	is_sorted[_until]	partial_sort[_copy]	set_symmetric_difference	uninitialized_copy[_n]
fill[_n]	lexicographical_compare	partition[_copy]	set_union	uninitialized_fill[_n]
find[_end _first_of _if _if_not]	max_element	reduce	sort	unique[_copy]

# New fused algorithms

adjacent_difference	<code>for_each[_n]</code>	merge	<code>remove[_copy _copy_if _if]</code>	<code>stable_partition</code>
adjacent_find	<code>generate[_n]</code>	<code>min_element</code>	<code>replace[_copy _copy_if _if]</code>	<code>stable_sort</code>
all_of	<code>includes</code>	<code>minmax_element</code>	<code>reverse[_copy]</code>	<code>swap_ranges</code>
any_of	<code>implace_merge</code>	<code>mismatch</code>	<code>rotate[_copy]</code>	<code>transform</code>
<code>copy[_if _n]</code>	<code>inclusive_scan</code>	<code>move</code>	<code>search[_n]</code>	<code>transform_exclusive_scan</code>
<code>count[_if]</code>	<code>is_heap[_until]</code>	<code>none_of</code>	<code>set_difference</code>	<code>transform_inclusive_scan</code>
<code>equal</code>	<code>is_partitioned</code>	<code>nth_element</code>	<code>set_intersection</code>	<code>transform_reduce</code>
<code>exclusive_scan</code>	<code>is_sorted[_until]</code>	<code>partial_sort[_copy]</code>	<code>set_symmetric_difference</code>	<code>uninitialized_copy[_n]</code>
<code>fill[_n]</code>	<code>lexicographical_compare</code>	<code>partition[_copy]</code>	<code>set_union</code>	<code>uninitialized_fill[_n]</code>
<code>find[_end _first_of _if _if_not]</code>	<code>max_element</code>	<code>reduce</code>	<code>sort</code>	<code>unique[_copy]</code>

# In this class we will focus on...

`transform` (non-serial) -> new overload of `transform` (serial)

`reduce` -> `unordered accumulate`

`inclusive_scan` -> `unordered partial_sum (inclusive)`

`transform_reduce` -> `fused transform (unordered) & reduce`

# Why do we need unordered algorithms?

Having algorithms be unordered means the some algorithms can be implemented to perform the operations in parallel where they couldn't before

# Why do we need unordered algorithms?

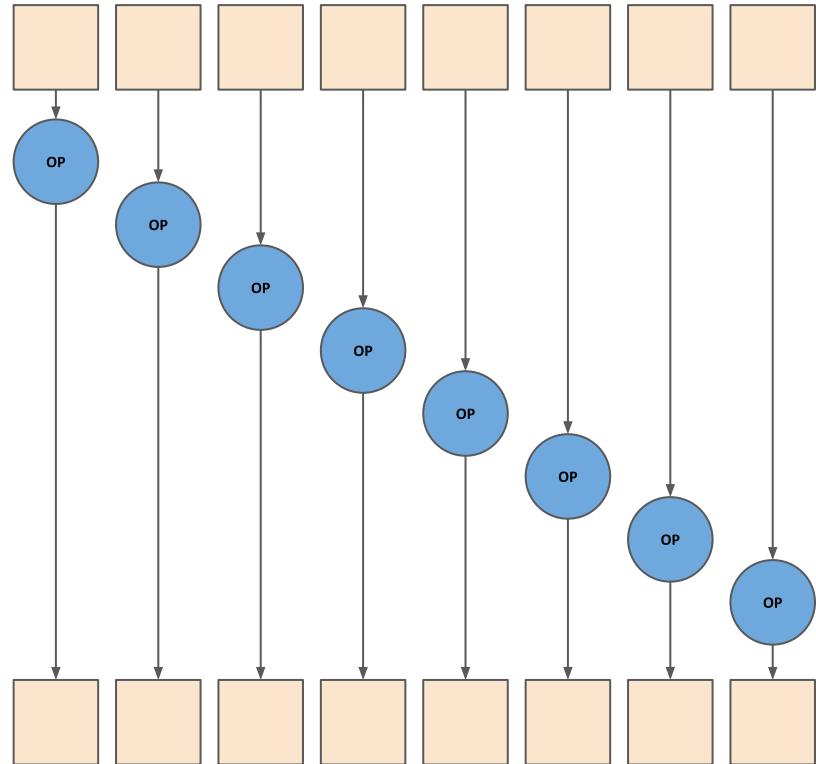
Having algorithms be unordered means the some algorithms can be implemented to perform the operations in parallel where they couldn't before

However allowing algorithms to be unordered comes with some caveats

# transform (serial)

```
d_first transform(first, last,  
                 d_first,  
                 unary_op)
```

```
for each pair of it in [first, last) and  
d_it in [d_first, last) in order  
    *d_it = unary_op(*it)  
return d_first
```



# transform (non-serial)

```
d_first transform([execution_policy,]  
                  first, last,  
                  d_first,  
                  unary_op)
```

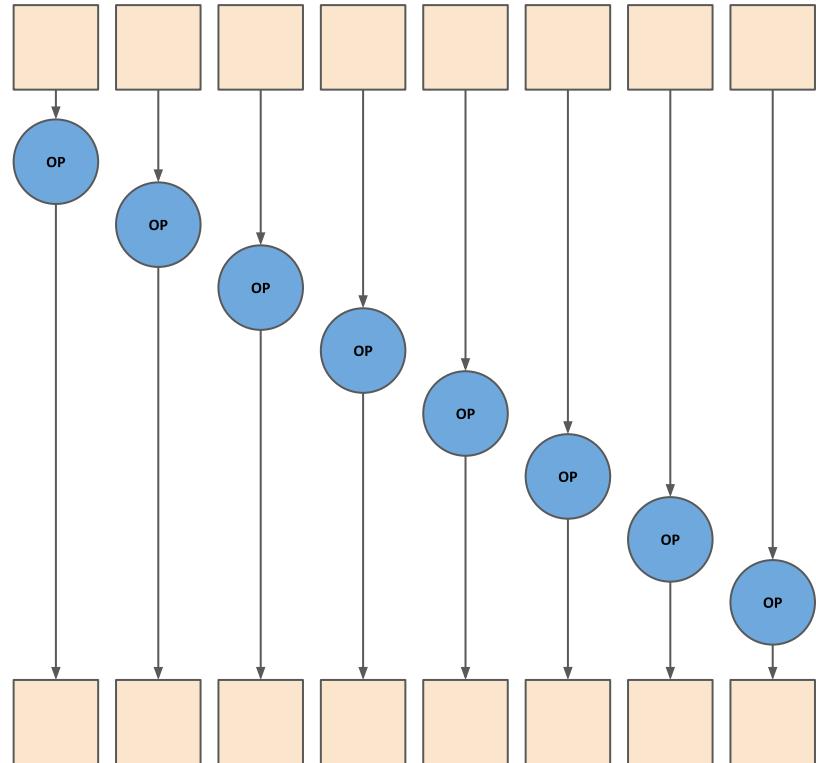
```
for each pair of it in [first, last) and d_it in [d_first, last)  
    *d_it = unary_op(*it)  
return d_first
```

# transform (non-serial)

```
d_first  
transform([execution_policy],
```

```
    first, last,  
    d_first,  
    unary_op)
```

```
for each pair of it in [first, last) and  
d_it in [d_first, last)  
    *d_it = unary_op(*it)  
return d_first
```



# transform (non-serial)

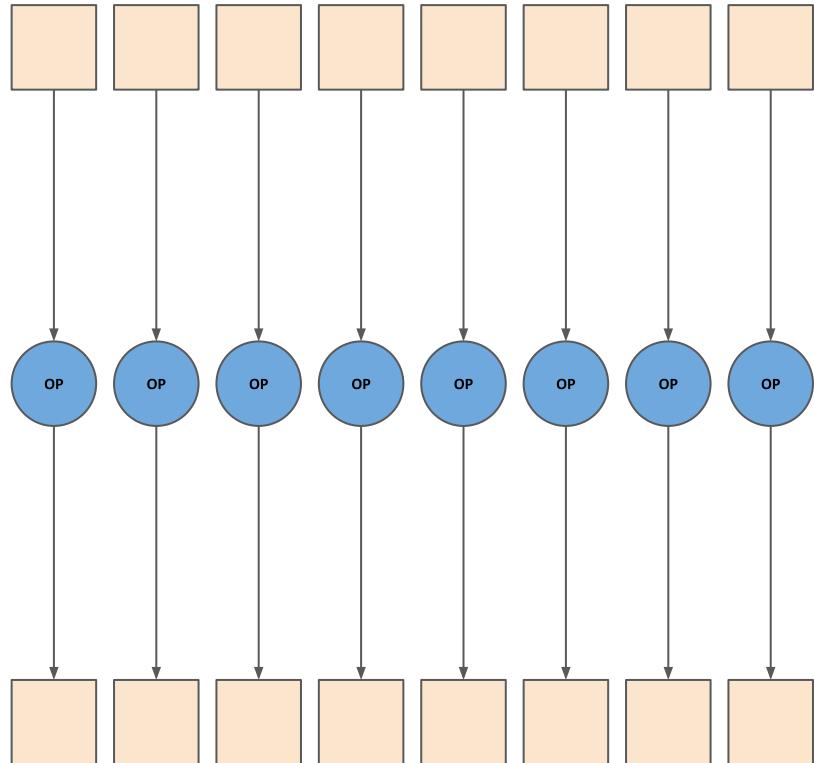
```
d_first  
transform([execution_policy],
```

```
    first, last,  
    d_first,  
    unary_op)
```

```
for each pair of it in [first, last) and  
d_it in [d_first, last)
```

```
    *d_it = unary_op(*it)
```

```
return d_first
```



# transform (non-serial)

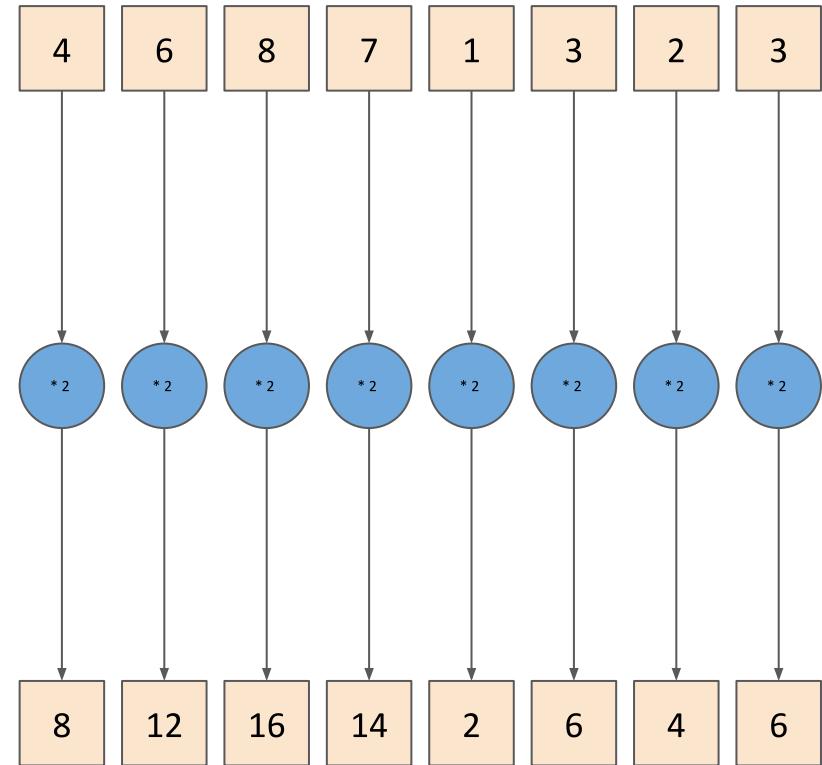
```
d_first  
transform([execution_policy],
```

```
    first, last,  
    d_first,  
    unary_op)
```

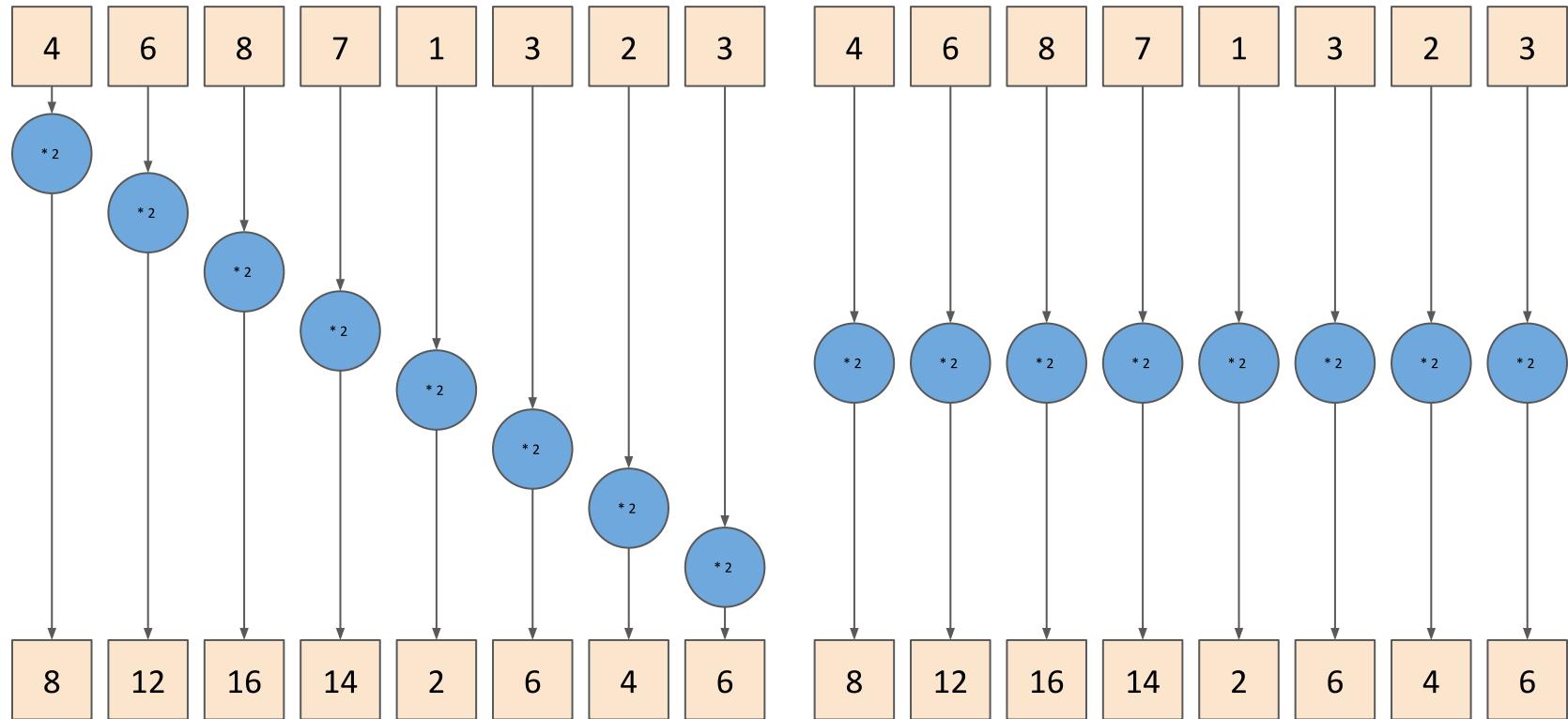
```
for each pair of it in [first, last) and  
d_it in [d_first, last)
```

```
    *d_it = unary_op(*it)
```

```
return d_first
```



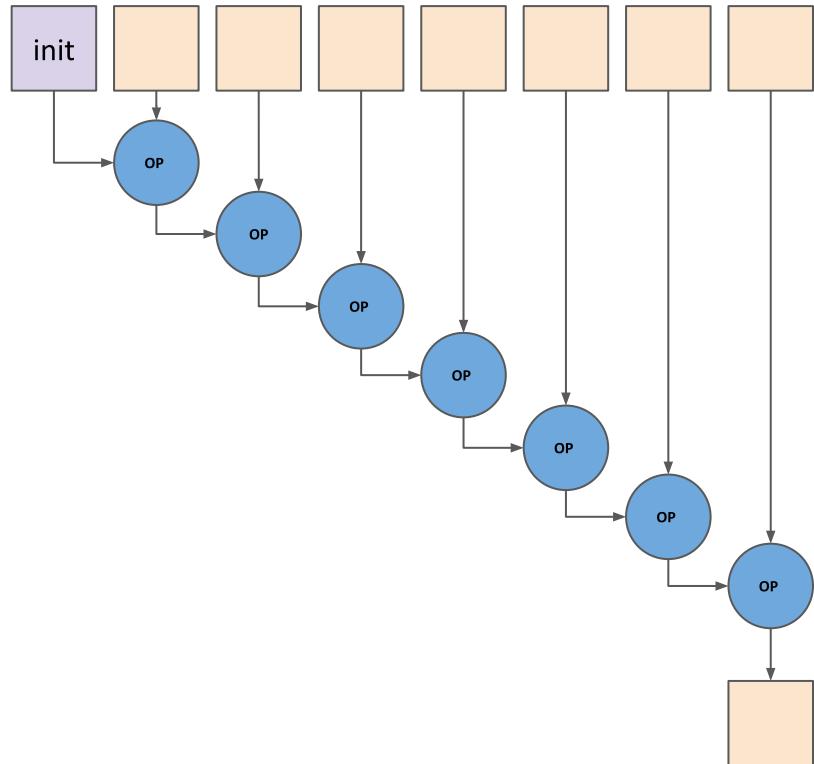
# transform (non-serial)



# accumulate

```
result accumulate(first, last,  
                 init,  
                 [binary_op])
```

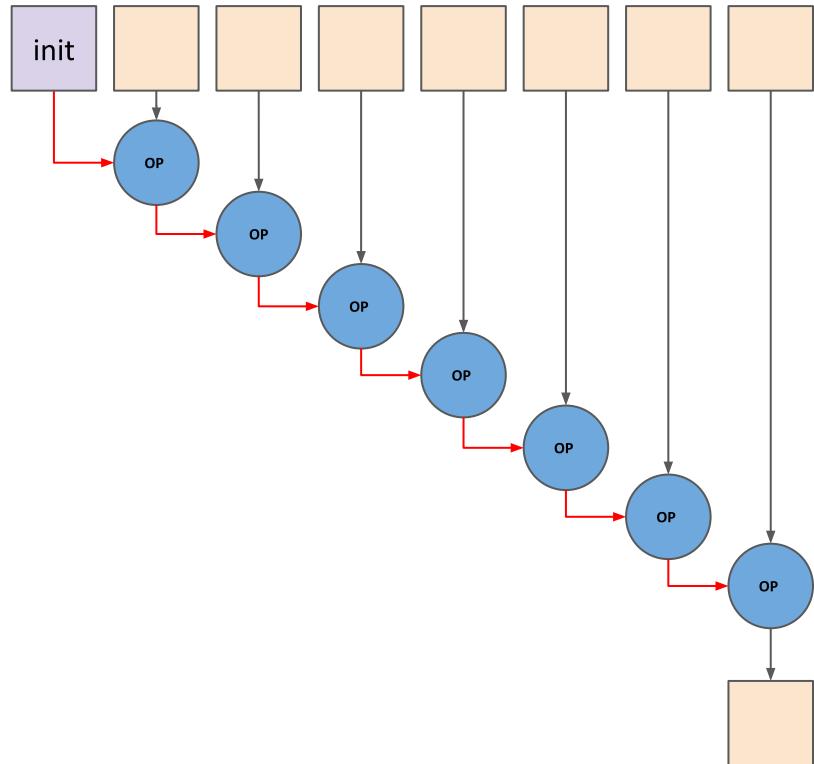
```
first acc = init  
then for each it in [first, last) in order  
    acc = binary_op(acc, *it)  
return acc
```



# accumulate

```
result accumulate(first, last,  
                 init,  
                 [binary_op])
```

```
first acc = init  
then for each it in [first, last) in order  
    acc = binary_op(acc, *it)  
return acc
```



# reduce

```
result reduce([execution_policy,]  
             first, last,  
             init,  
             [binary_op])
```

```
acc = GSUM(binary_op, init, *first, ..., *(last-1))
```

# What is this GSUM thing?

$\text{GSUM}(\text{op}, a^1, \dots, a^N) == \text{GNSUM}(\text{op}, b^1, \dots, b^N)$   
where  $b^1, \dots, b^N$  may be any permutation of  $a^1, \dots, a^N$

$\text{GNSUM}(\text{op}, a^1, \dots, a^N) == a^1$ , if  $N == 1$

$\text{GNSUM}(\text{op}, a^1, \dots, a^N) == \text{op}(\text{GNSUM}(\text{op}, a^1, \dots, a^K),$   
 $\text{GNSUM}(\text{op}, a^{K+1}, \dots, a^N))$ , otherwise  
where  $a^K$  is an arbitrary point between  $a^1$  and  $a^N$

# reduce

```
result reduce([execution_policy, ]
             first, last,
             init,
             [binary_op])

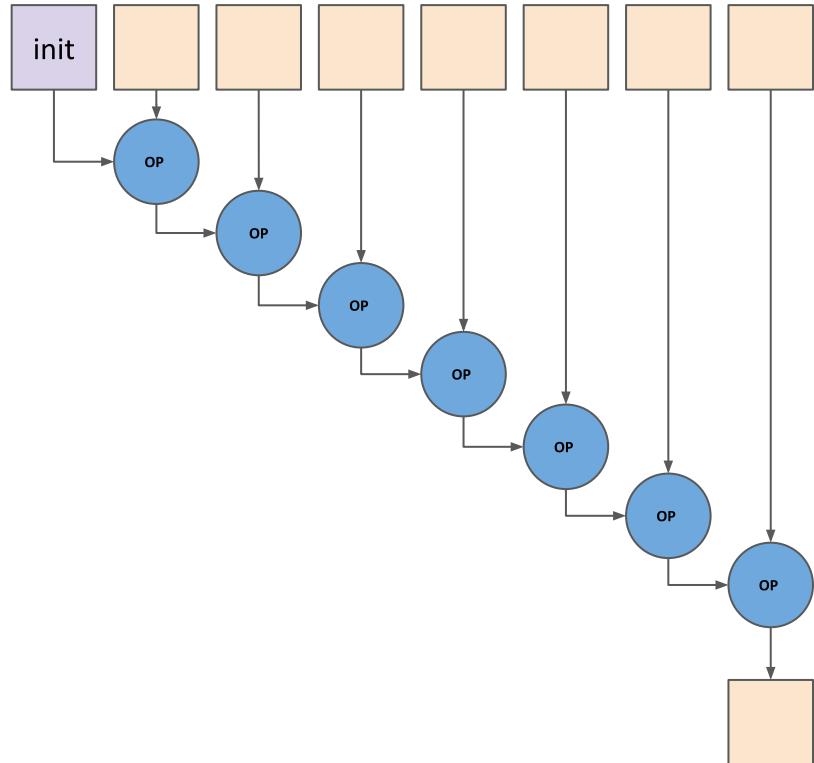
acc = GSUM(binary_op, init, *first, ..., *(last-1))
return acc
```

# reduce

```
result reduce([execution_policy,]  
            first, last,  
            init,  
            [binary_op])
```

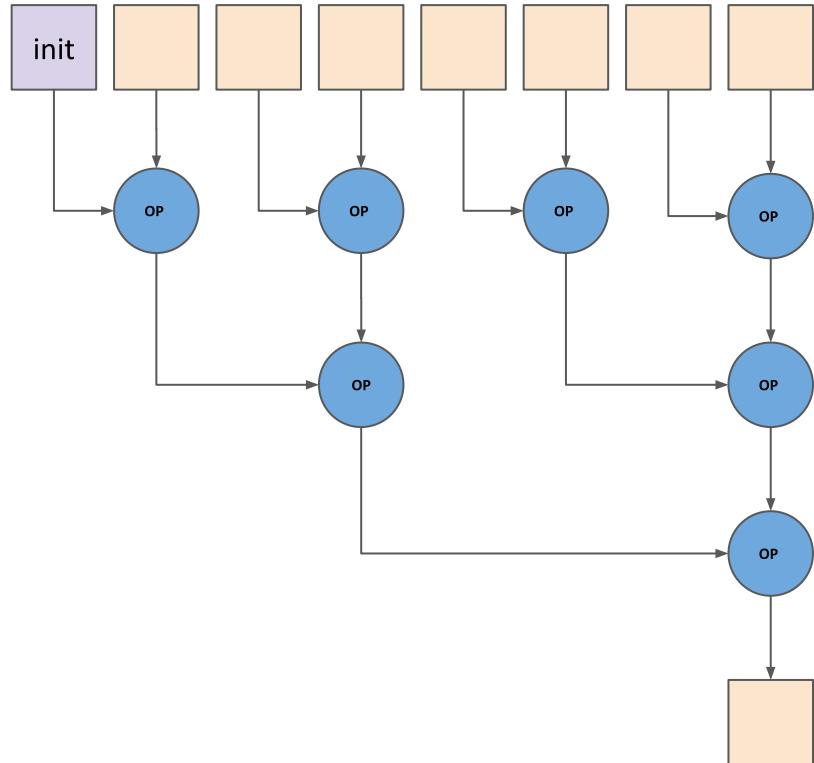
```
acc = GSUM(binary_op, init, *first,  
           ... , *(last-1))
```

```
return acc
```



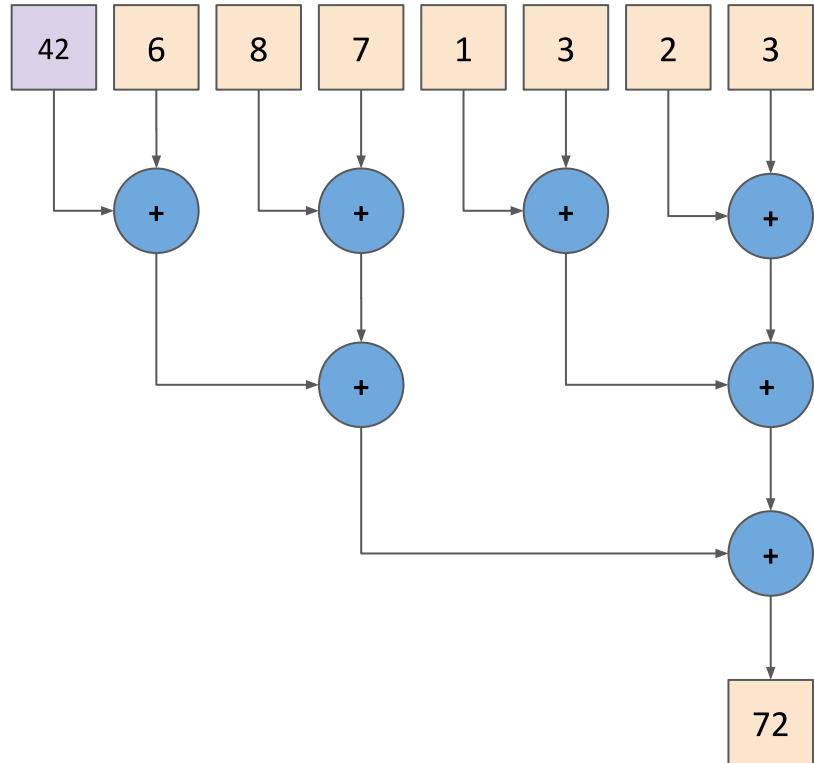
# reduce

```
result reduce([execution_policy,]  
            first, last,  
            init,  
            [binary_op])  
  
acc = GSUM(binary_op, init, *first,  
           ... , *(last-1))  
  
return acc
```

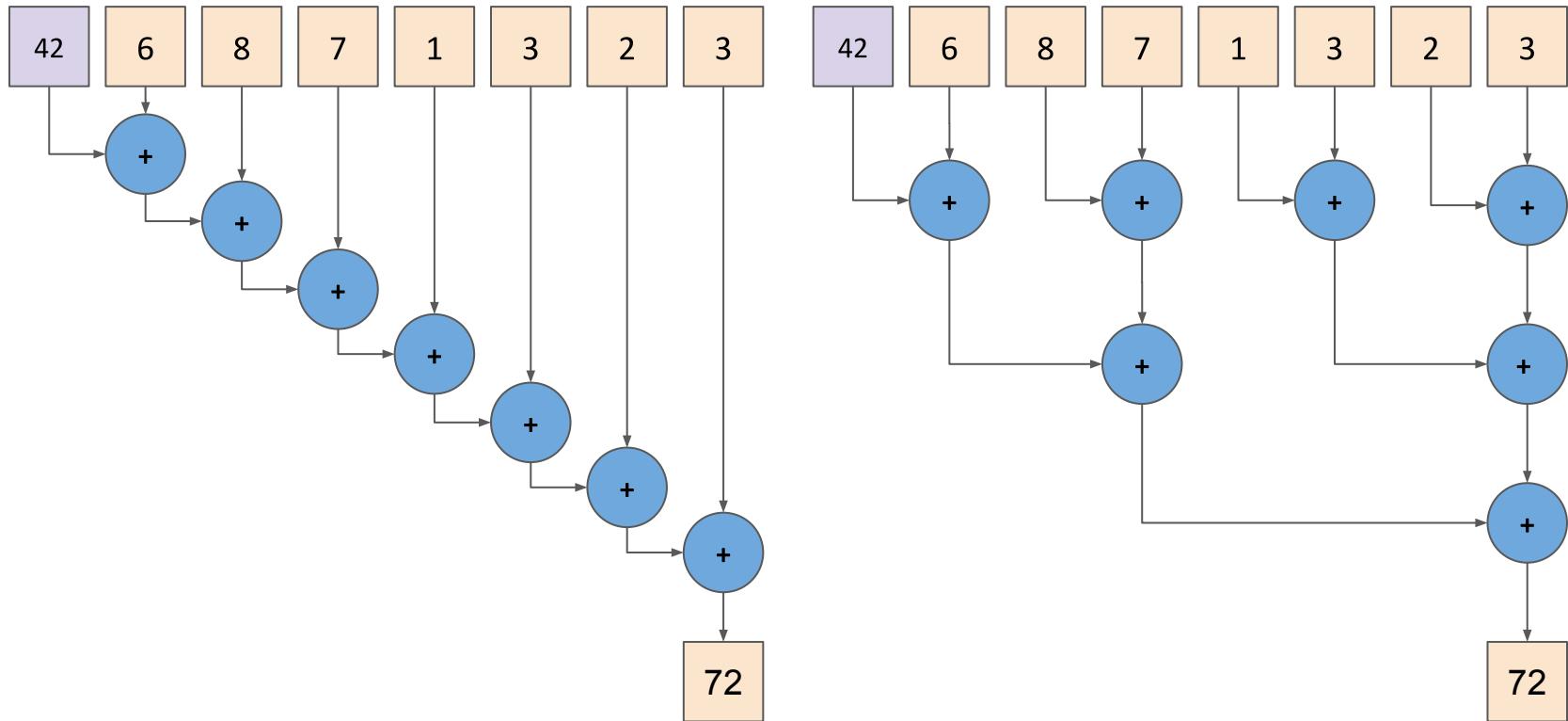


# reduce

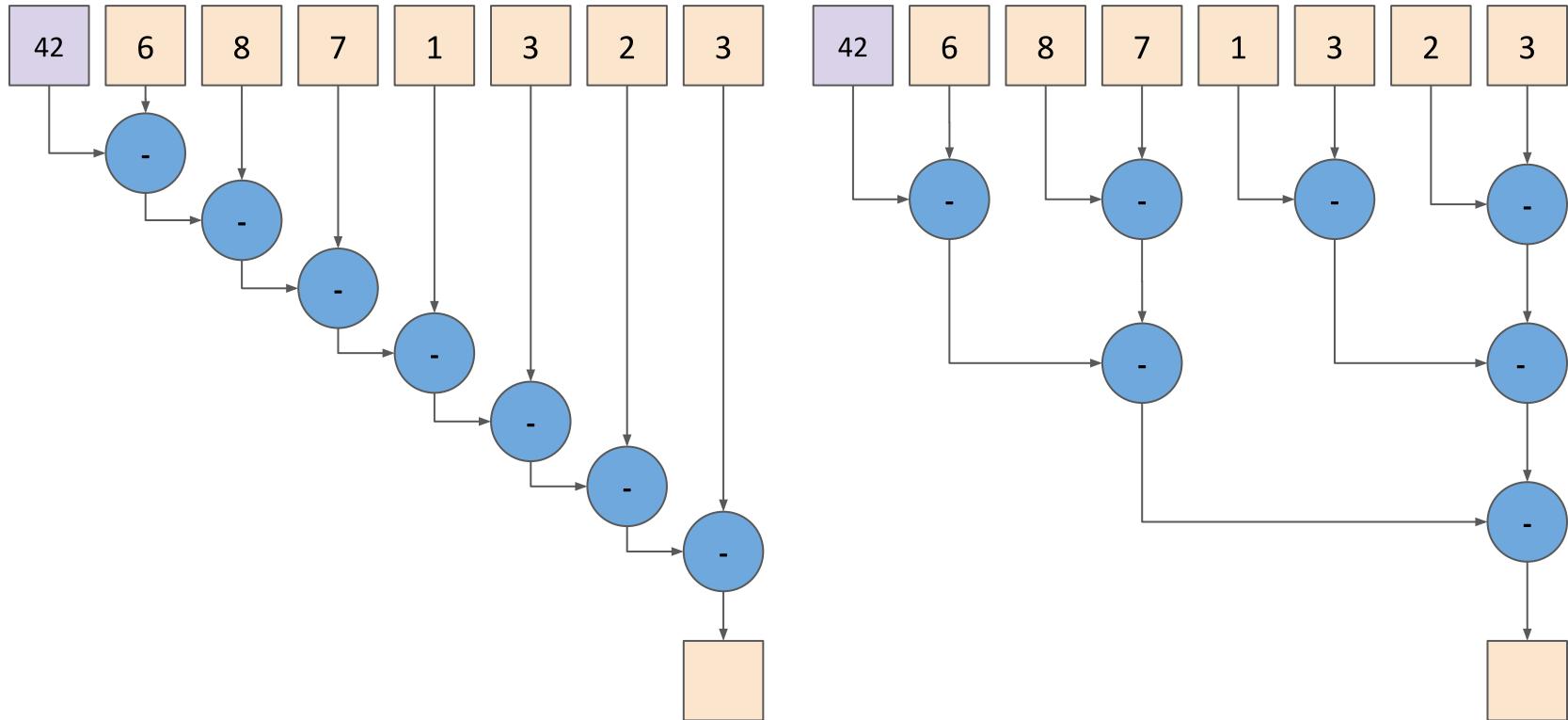
```
result reduce([execution_policy,]  
            first, last,  
            init,  
            [binary_op])  
  
acc = GSUM(binary_op, init, *first,  
           ...,*last-1))  
  
return acc
```



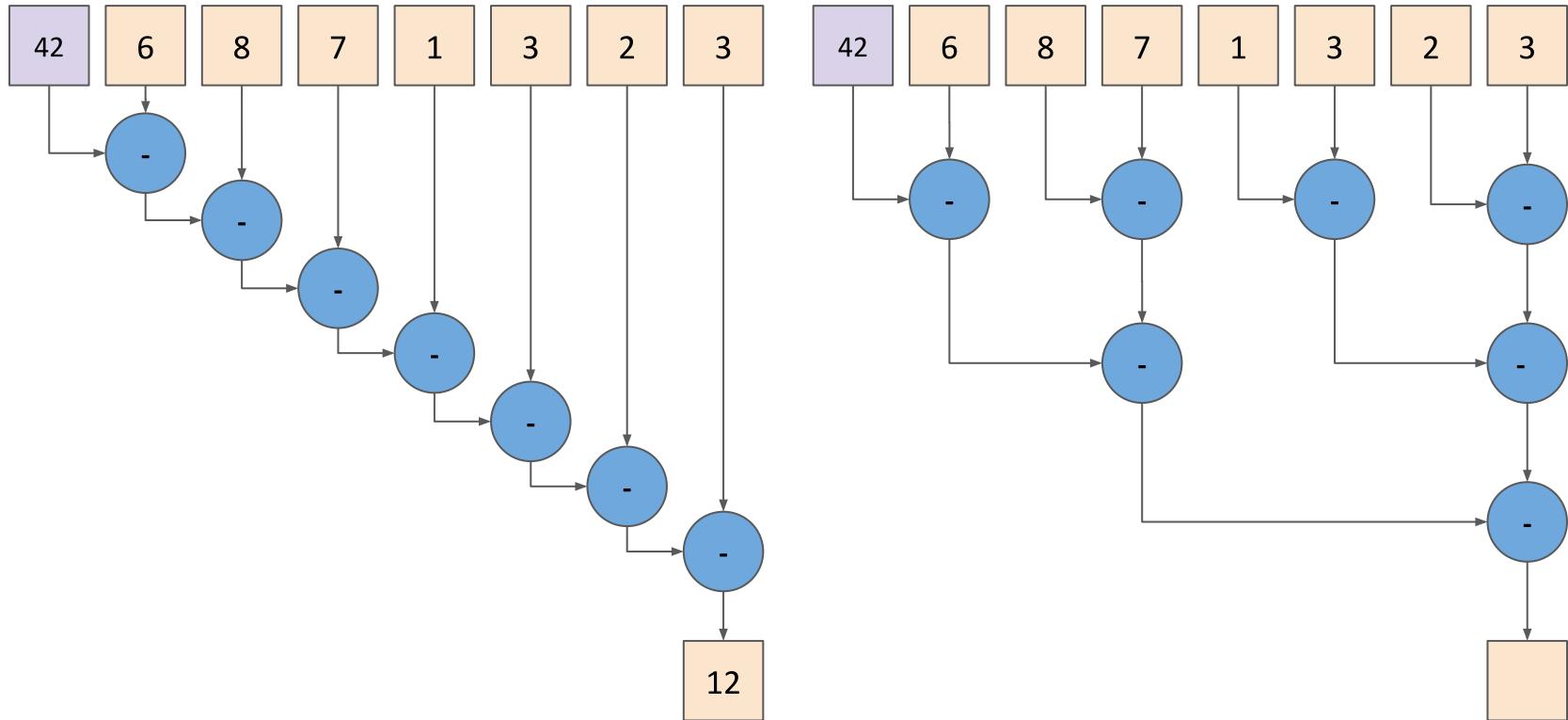
# reduce



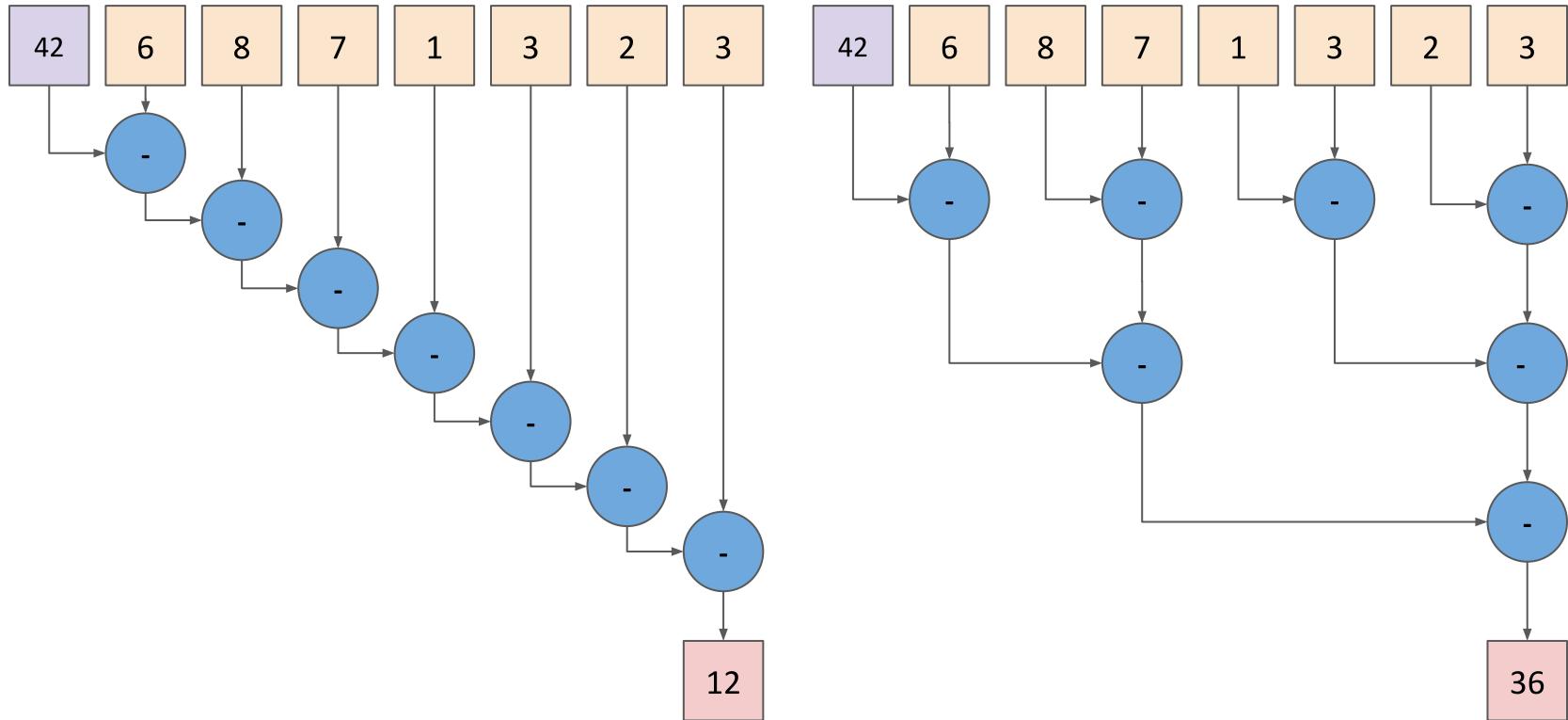
# reduce



# reduce



# reduce



# Quick maths revision

Due to the requirements of GSUM and GNSUM, reduce is allowed to be unordered

However this means the `binary_op` is required to be **commutative** and **associative**

# Commutativity

Commutativity means changing the order of operations does not change the result

## Integer operations

$$x + y == y + x$$

$$x * y == y * x$$

$$x - y != y - x$$

$$x / y != y / x$$

## Floating-point operations

$$x + y == y + x$$

$$x * y == y * x$$

$$x - y != y - x$$

$$x / y != y / x$$

# Associativity

Associativity means changing the grouping of operations does not change the result

Integer operations

$$(x + y) + z == x + (y + z)$$

$$(x * y) * z == x * (y * z)$$

$$(x - y) - z == x - (y - z)$$

$$(x / y) / z == x / (y / z)$$

Floating-point operations

$$(x + y) + z == x + (y + z)$$

$$(x * y) * z == x * (y * z)$$

$$(x - y) - z == x - (y - z)$$

$$(x / y) / z == x / (y / z)$$

# transform\_reduce

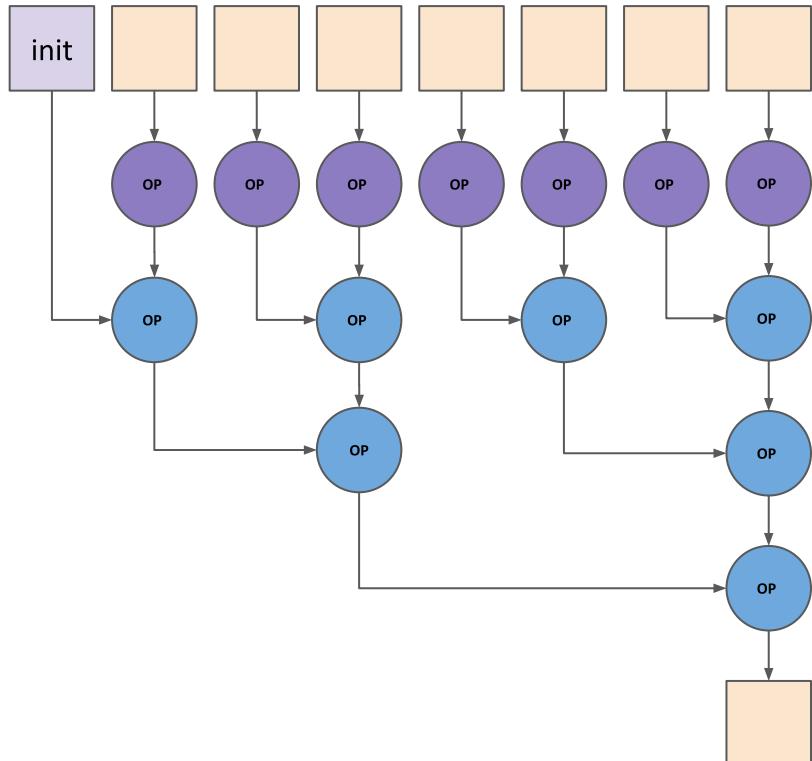
```
result reduce([execution_policy,]
             first, last,
             init,
             [binary_op,]
             [unary_op])
```

```
acc = GSUM(binary_op, init, unary_op(*first),
            ..., unary_op(*(last-1)))
return acc
```

# transform\_reduce

```
result reduce([execution_policy,]
             first, last,
             init,
             [binary_op,]
             [unary_op])
```

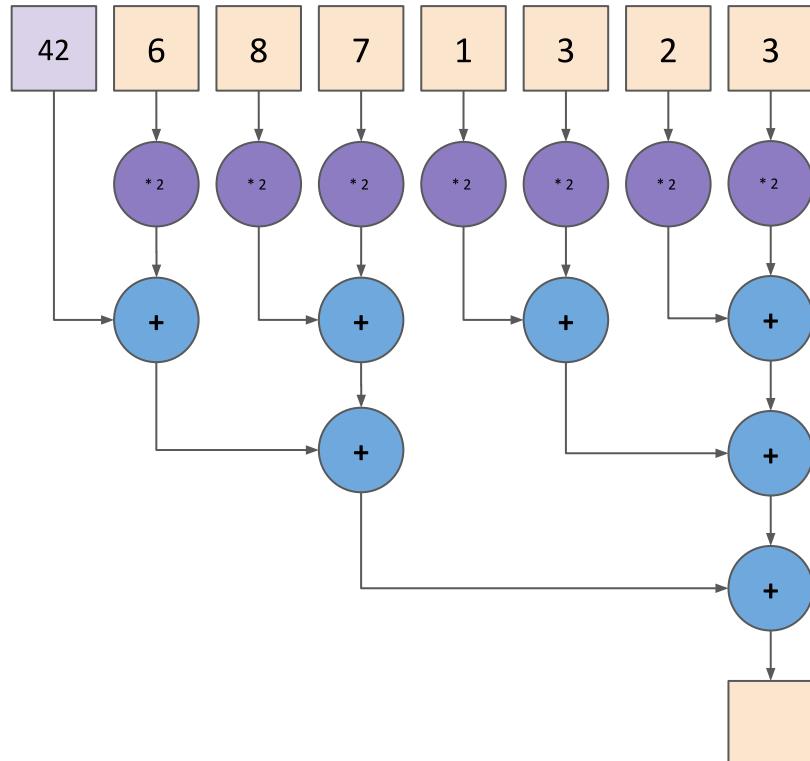
```
acc = GSUM(binary_op, init,
            unary_op(*first),
            unary_op(*(last-1))..,
return acc
```



# transform\_reduce

```
result reduce([execution_policy,]
             first, last,
             init,
             [binary_op,]
             [unary_op])
```

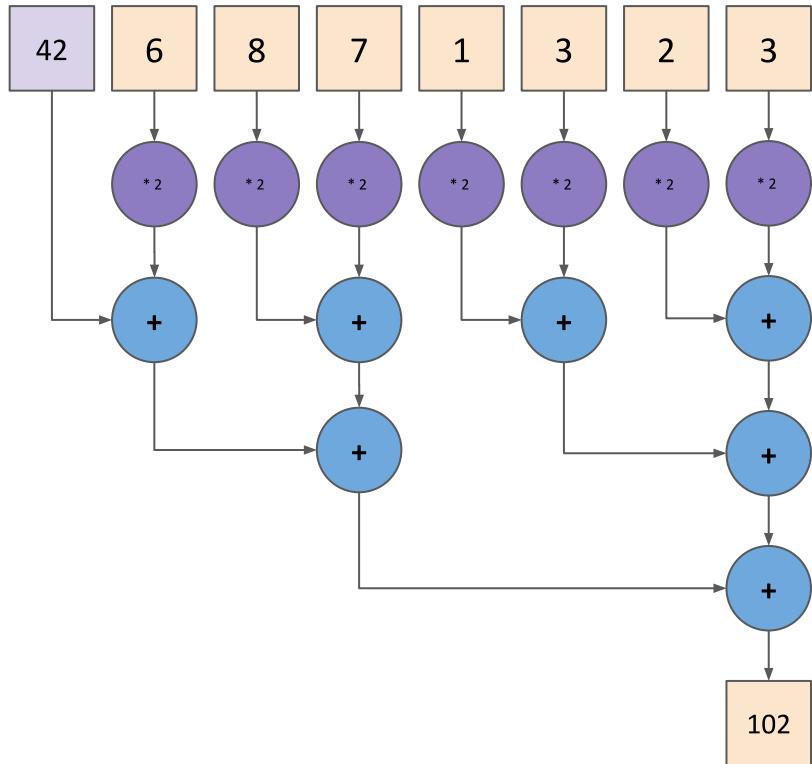
```
acc = GSUM(binary_op, init,  
           unary_ōp(*first),  
           unary_ōp(*last-1)))  
  
return acc
```



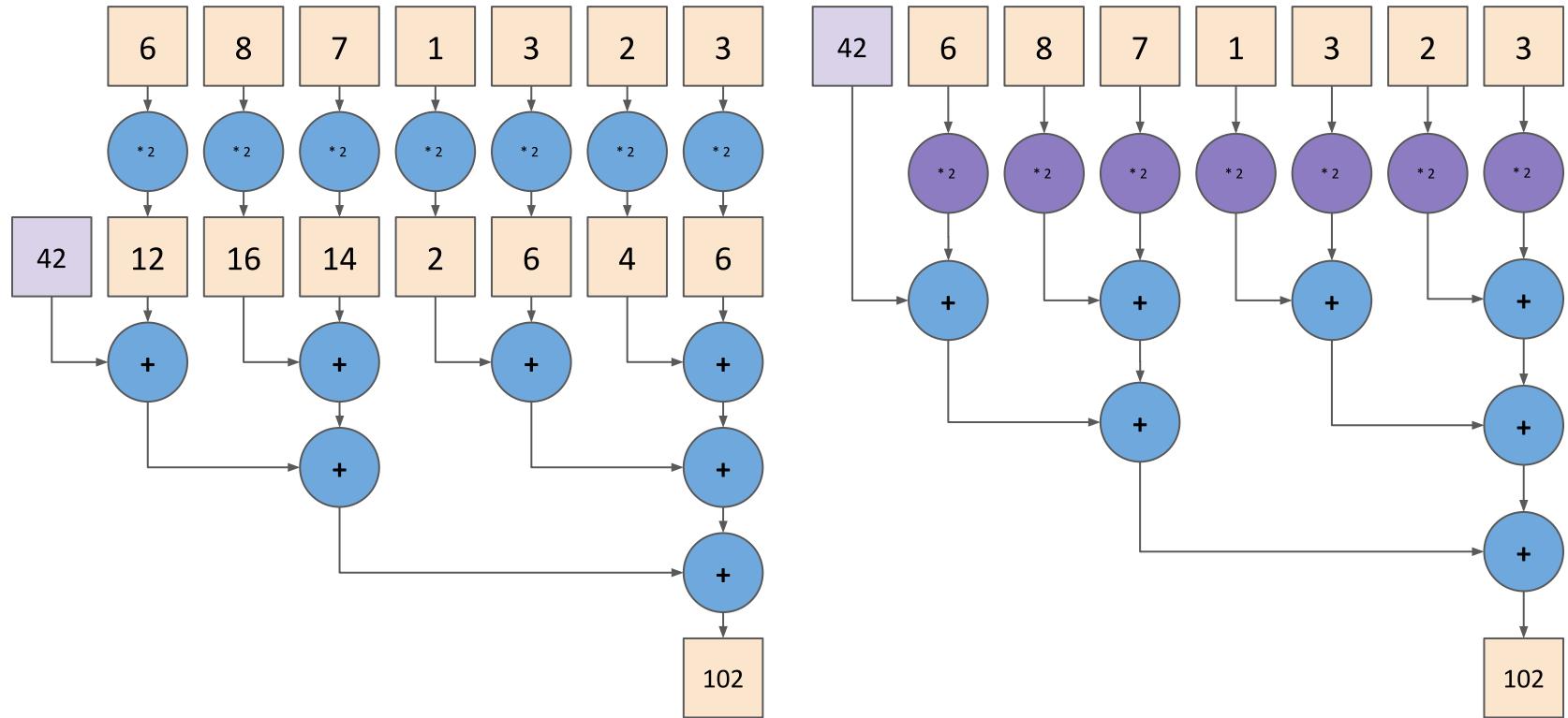
# transform\_reduce

```
result reduce([execution_policy,]
             first, last,
             init,
             [binary_op,]
             [unary_op])
```

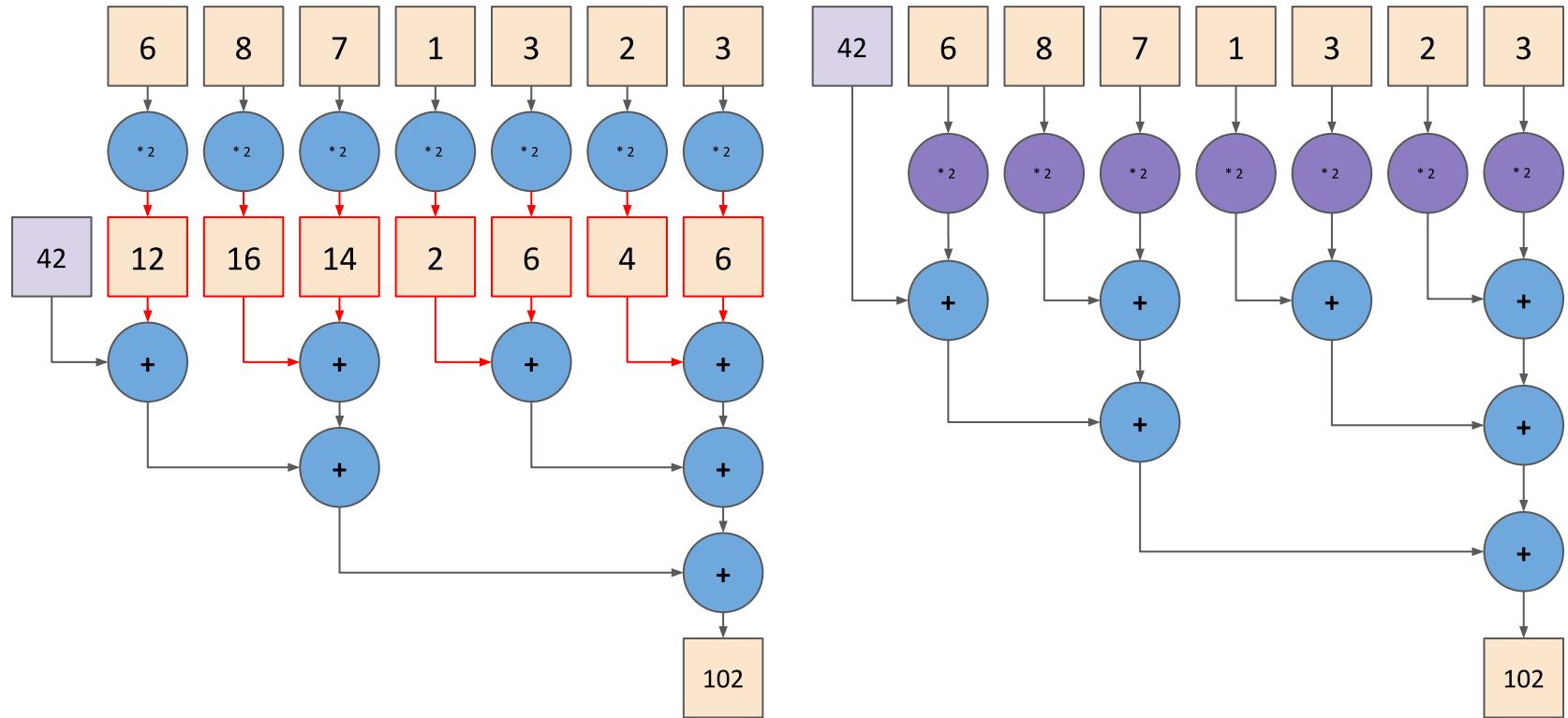
```
acc = GSUM(binary_op, init,
           unary_op(*first),
           unary_op(*(last-1))..,
return acc
```



# Benefit of fused transform & reduce



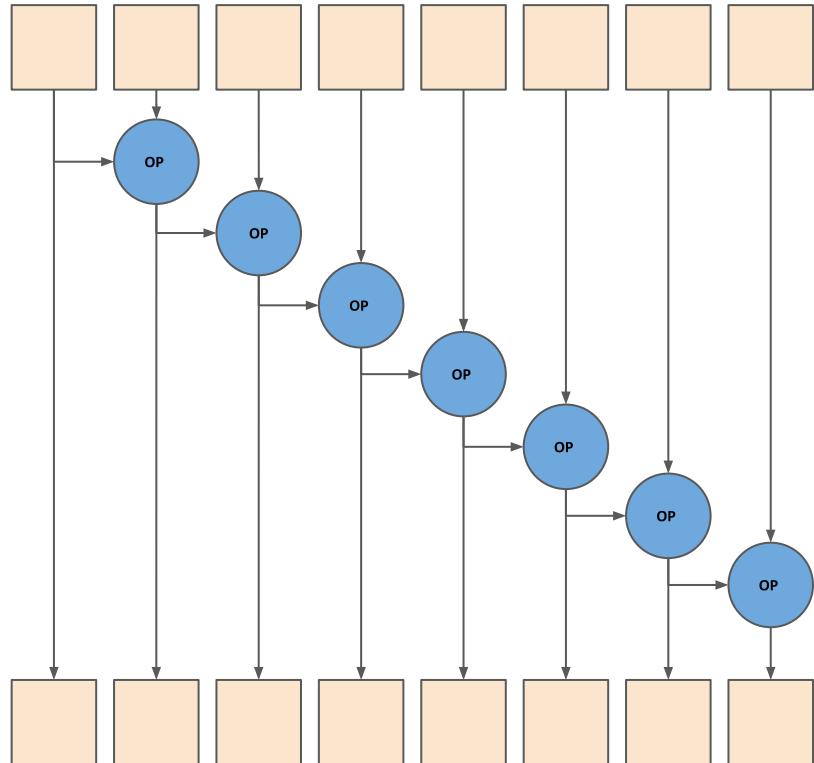
# Benefit of fused transform & reduce



# partial\_sum

```
d_first partial_sum(first, last,
                    d_first,
                    [binary_op])

first sum = *first++
*d_first++ = sum
for each pair of it in [first, last) and
d_it in [d_first, last) in order
    sum = binary_op(sum, *it)
    *d_it = sum
return d_first
```



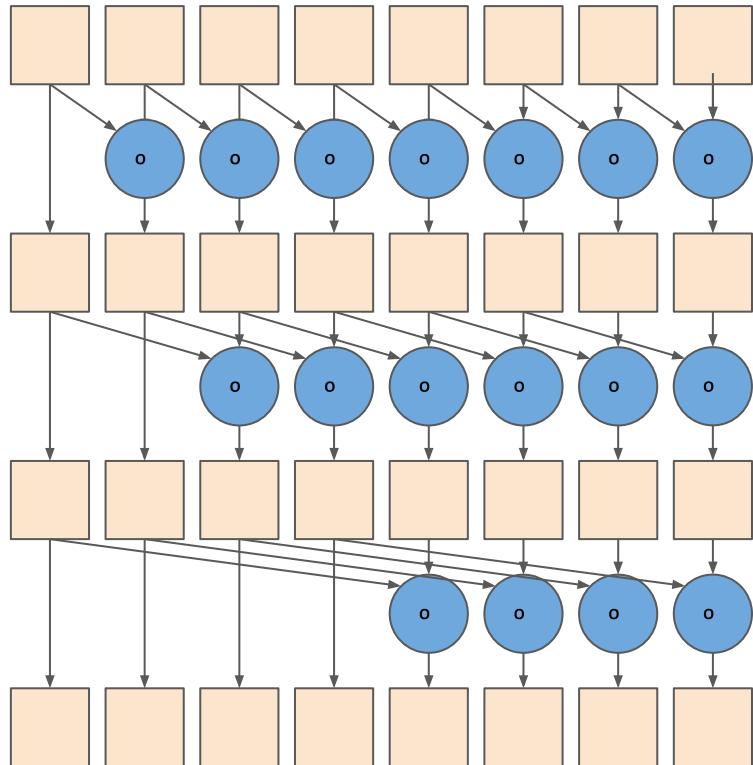
# inclusive\_scan

```
d_first partial_sum([execution_policy],  
                     first, last,  
                     d_first,  
                     [binary_op],  
                     [init])
```

```
/* very complicated :( */  
return d_first
```

# inclusive\_scan

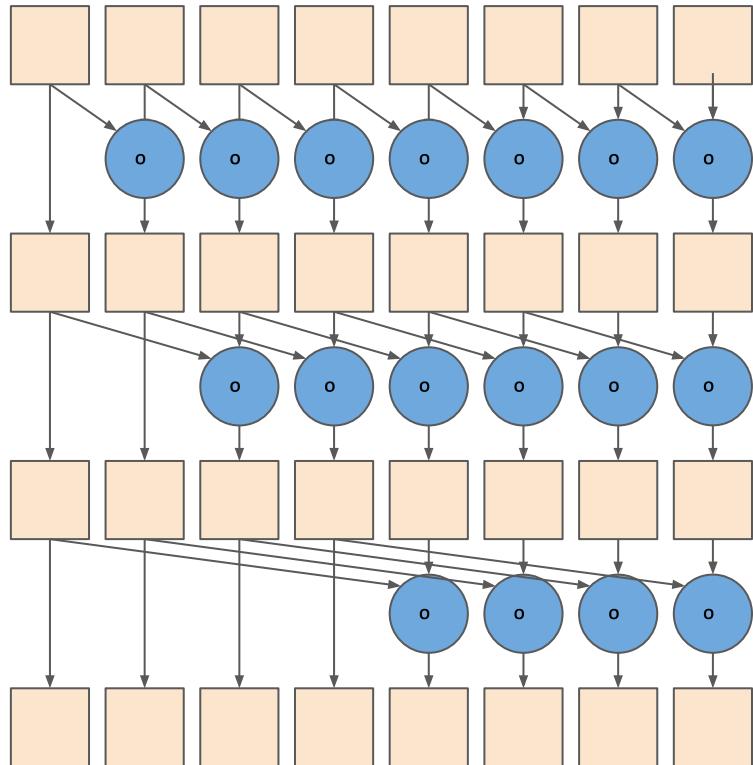
```
d_first  
partial_sum([execution_policy,]  
           first, last,  
           d_first,  
           [binary_op],  
           [init])  
  
/* very complicated :( */  
return d_first
```



# inclusive\_scan

```
d_first  
partial_sum([execution_policy,  
            first, last,  
            d_first,  
            [binary_op],  
            [init])  
  
/* very complicated :( */  
return d_first
```

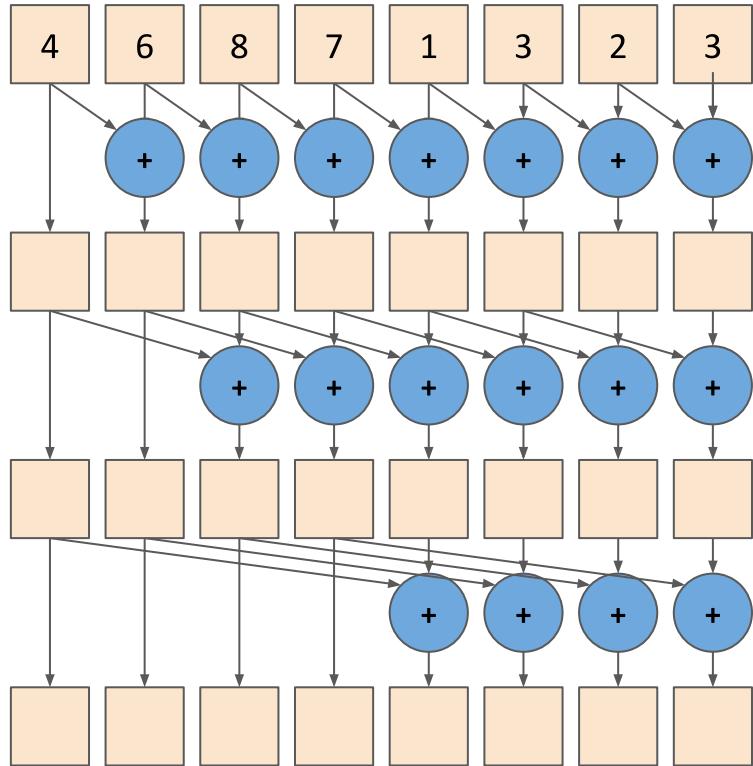
Hillis & Steele Scan



# inclusive\_scan

```
d_first  
partial_sum([execution_policy,  
            first, last,  
            d_first,  
            [binary_op],  
            [init])  
  
/* very complicated :( */  
return d_first
```

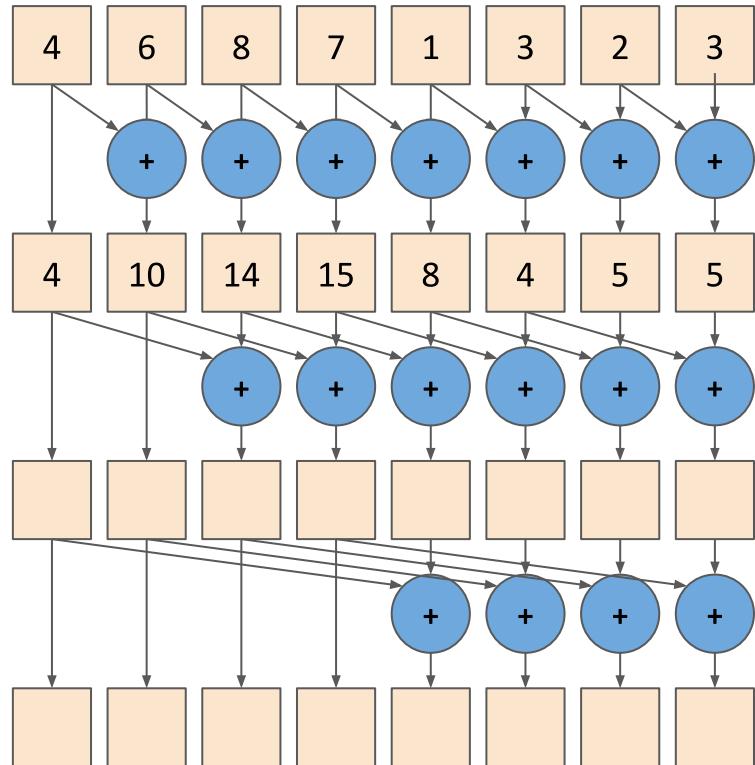
Hillis & Steele Scan



# inclusive\_scan

```
d_first  
partial_sum([execution_policy,  
            first, last,  
            d_first,  
            [binary_op],  
            [init])  
  
/* very complicated :( */  
return d_first
```

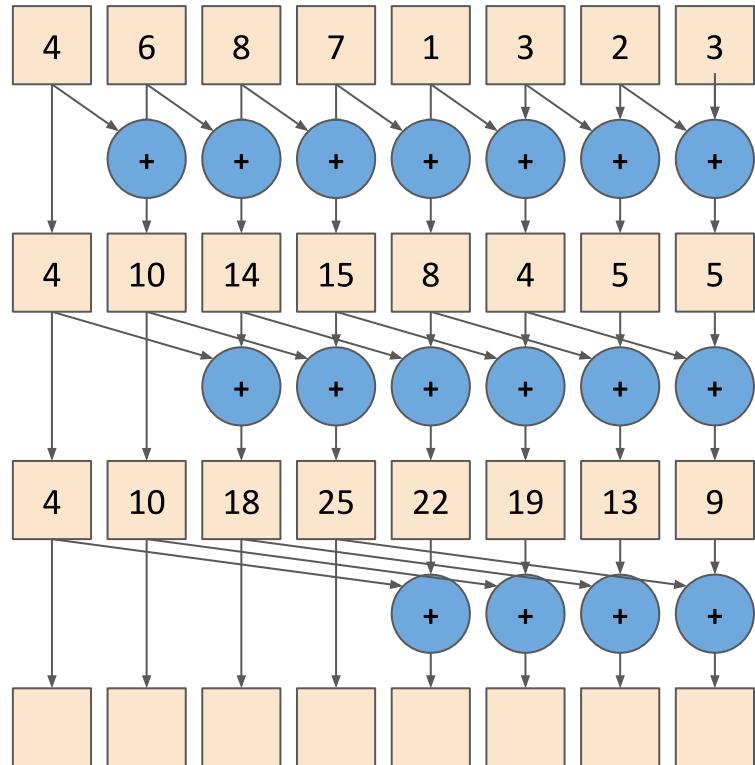
Hillis & Steele Scan



# inclusive\_scan

```
d_first  
partial_sum([execution_policy,  
            first, last,  
            d_first,  
            [binary_op],  
            [init])  
  
/* very complicated :( */  
return d_first
```

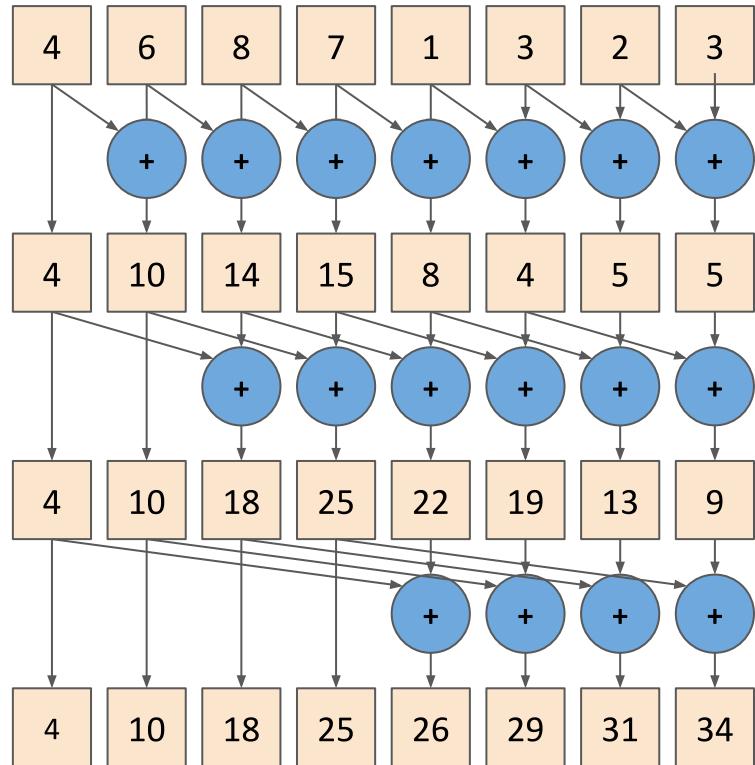
Hillis & Steele Scan



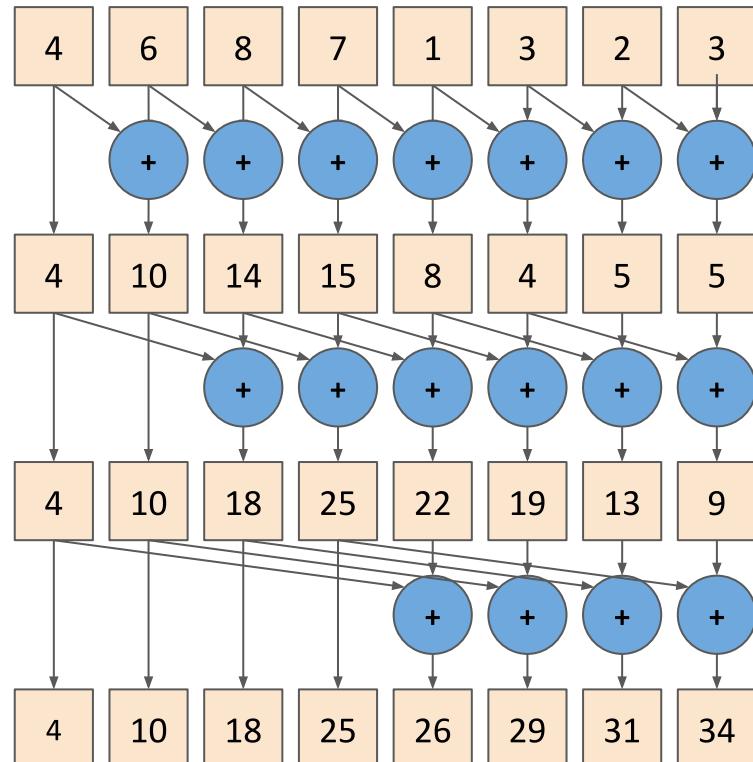
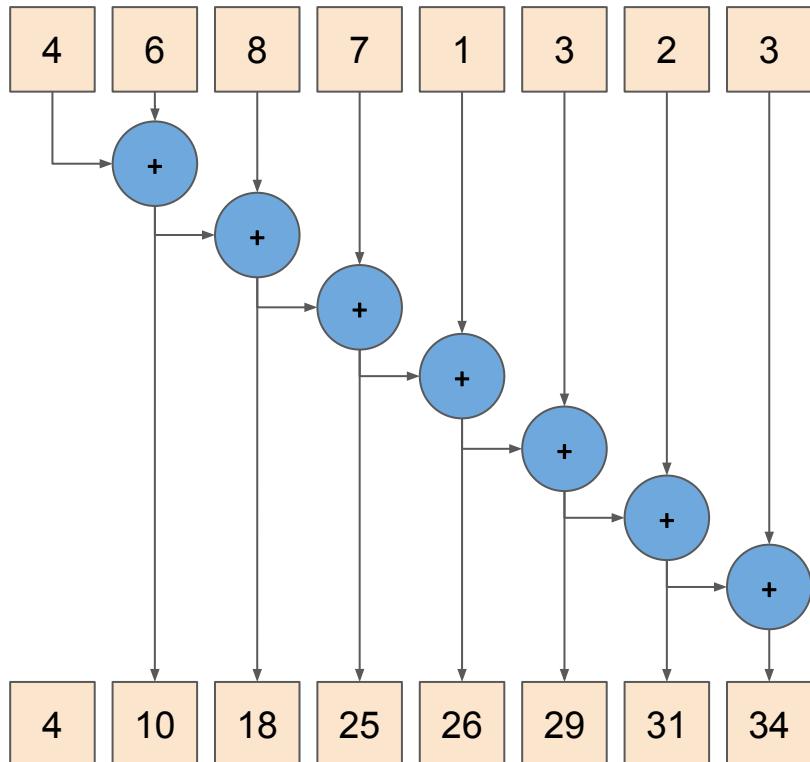
# inclusive\_scan

```
d_first  
partial_sum([execution_policy,]  
           first, last,  
           d_first,  
           [binary_op],  
           [init])  
  
/* very complicated :( */  
return d_first
```

Hillis & Steele Scan



# inclusive\_scan



# inclusive\_scan

The `binary_op` for `inclusive_scan` is also required to be associative

Integer operations

$$(x + y) + z == x + (y + z)$$

$$(x * y) * z == x * (y * z)$$

$$(x - y) - z == x - (y - z)$$

$$(x / y) / z == x / (y / z)$$

Floating-point operations

$$(x + y) + z == x + (y + z)$$

$$(x * y) * z == x * (y * z)$$

$$(x - y) - z == x - (y - z)$$

$$(x / y) / z == x / (y / z)$$

# A note about iterators

`transform` (serial), `accumulate`, `partial_sum` use **Input/Output** iterators:

- Increment (without multiple passes)

# A note about iterators

`transform` (serial), `accumulate`, `partial_sum` use **Input/Output** iterators:

- Increment (without multiple passes)

`transform` (unordered), `reduce`, `transform_reduce`, `partial_sum` use **Forward** iterators:

- Increment (with multiple passes)

# A note about iterators

`transform` (serial), `accumulate`, `partial_sum` use **Input/Output** iterators:

- Increment (without multiple passes)

`transform` (unordered), `reduce`, `transform_reduce`, `partial_sum` use **Forward** iterators:

- Increment (with multiple passes)

When moving data (important for GPUs) use **Contiguous** iterators:

- All elements in the range are laid out contiguously in memory

# Measuring algorithm performance

When parallelizing a piece of code it's important to measure any performance benefit

# Measuring algorithm performance

When parallelizing a piece of code it's important to measure any performance benefit

You want to isolate the part of the code that you are interested in measuring to ensure that unrelated parts of the code do not influence the measurement

# Measuring algorithm performance

When parallelizing a piece of code it's important to measure any performance benefit

You want to isolate the part of the code that you are interested in measuring to ensure that unrelated parts of the code do not influence the measurement

You also want to run the code multiple times and calculate the average to ensure you get a more accurate representation of the execution time

# Measuring algorithm performance

Ensure that there are no background processes

- If the OS switches out the hardware threads that are running your benchmark then it can affect the measurement
- You can often ensure this using command line options to bind your process to particular threads

# Measuring algorithm performance

Ensure that there are no background processes

- If the OS switches out the hardware threads that are running your benchmark then it can affect the measurement
- You can often ensure this using command line options to bind your process to particular threads

Ensure that you run the benchmark with a range of different input sizes and operation complexities

- There is always overhead involved in parallelism, so sometimes the overhead can affect the performance

# Measuring algorithm performance

Be aware of the hardware that you are running on

- The performance can vary depending on the hardware
- Different hardware will have varying overhead in launching work and copying data to where the work is being done

# Measuring algorithm performance

Be aware of the hardware that you are running on

- The performance can vary depending on the hardware
- Different hardware will have varying overhead in launching work and copying data to where the work is being done

Always run in release mode

- Debug mode will not give accurate performance evaluation
- This is due to additional code that is generated for debug builds

# Measuring algorithms

```
template <typename Func>
auto benchmark(Func &&func, int iterations, std::string
caption)
    std::chrono::duration<double, std::milli>
        totalTime{0};
    for (int i = 0; i < iterations; i++) {
        std::chrono::steady_clock::time_point start =
            std::chrono::steady_clock::now();
        func();
        std::chrono::steady_clock::time_point end =
            std::chrono::steady_clock::now();
        totalTime += (end - start);
    }
    return averageTime;
}
```

This is a function which uses std::chrono to time a number of iterations of a function and calculate the average

# Key takeaways

C++17 introduces new unordered and fused algorithms and execution policies to provide the opportunity for parallelism

Unordered algorithms introduce new considerations to how they are implemented and used, and furthermore how they can be parallelised

There are many important things to consider when benchmarking the performance of an algorithm



# Exercise 1:

## Implementing Sequential Algorithms

- Implement the sequential variant of transform
- Implement the sequential variant of reduce
- Implement the sequential variant of transform\_reduce
- Evaluate the performance of the algorithms

Exercise document: <https://goo.gl/o688Z9>

# Exercise 1:

## Implementing Sequential Algorithms

```
1. template <class ForwardIt1, class ForwardIt2, class UnaryOperation>
2. ForwardIt2 transform(sequential_execution_policy seq, ForwardIt1 first, ForwardIt1 last,
3.                      ForwardIt2 d_first, UnaryOperation unary_op) {
4.
5.     /* implement me */
6.
7. }
```

[https://github.com/AerialMantis/cppcon2018-parallelism-class/blob/master/include/bits/sequential\\_transform.h](https://github.com/AerialMantis/cppcon2018-parallelism-class/blob/master/include/bits/sequential_transform.h)

# Exercise 1:

## Implementing Sequential Algorithms

```
1. template <class ForwardIt, class T, class BinaryOperation, class UnaryOperation>
2. T reduce(sequential_execution_policy seq, ForwardIt first, ForwardIt last,
3.           T init, BinaryOperation binary_op) {
4.
5.     /* implement me */
6.
7. }
```

[https://github.com/AerialMantis/cppcon2018-parallelism-class/blob/master/include/bits/sequential\\_reduce.h](https://github.com/AerialMantis/cppcon2018-parallelism-class/blob/master/include/bits/sequential_reduce.h)

# Exercise 1:

## Implementing Sequential Algorithms

```
1. template <class ForwardIt, class T, class BinaryOperation>
2. T transform_reduce(sequential_execution_policy seq, ForwardIt first, ForwardIt last,
3.                     T init, BinaryOperation binary_op, UnaryOperation unary_op) {
4.
5.     /* implement me */
6.
7. }
```

[https://github.com/AerialMantis/cppcon2018-parallelism-class/blob/master/include/bits/sequential\\_transform\\_reduce.h](https://github.com/AerialMantis/cppcon2018-parallelism-class/blob/master/include/bits/sequential_transform_reduce.h)



# Chapter 3: Fundamentals of Parallelism

## Prerequisites:

1. Previous chapters
2. Patterns and parallel algorithms

## You will learn:

1. Communication patterns
2. Reordering algorithms
3. Work distribution
4. Handling dependencies

# Communication patterns

Parallel computing is about breaking up a problem into smaller tasks and having multiple processors working together to solve a problem

# Communication patterns

Parallel computing is about breaking up a problem into smaller tasks and having multiple processors **working together** to solve a problem

This requires communication, and in parallel computing this is done via memory

# Communication patterns

Parallel computing is about breaking up a problem into smaller tasks and having multiple processors **working together** to solve a problem

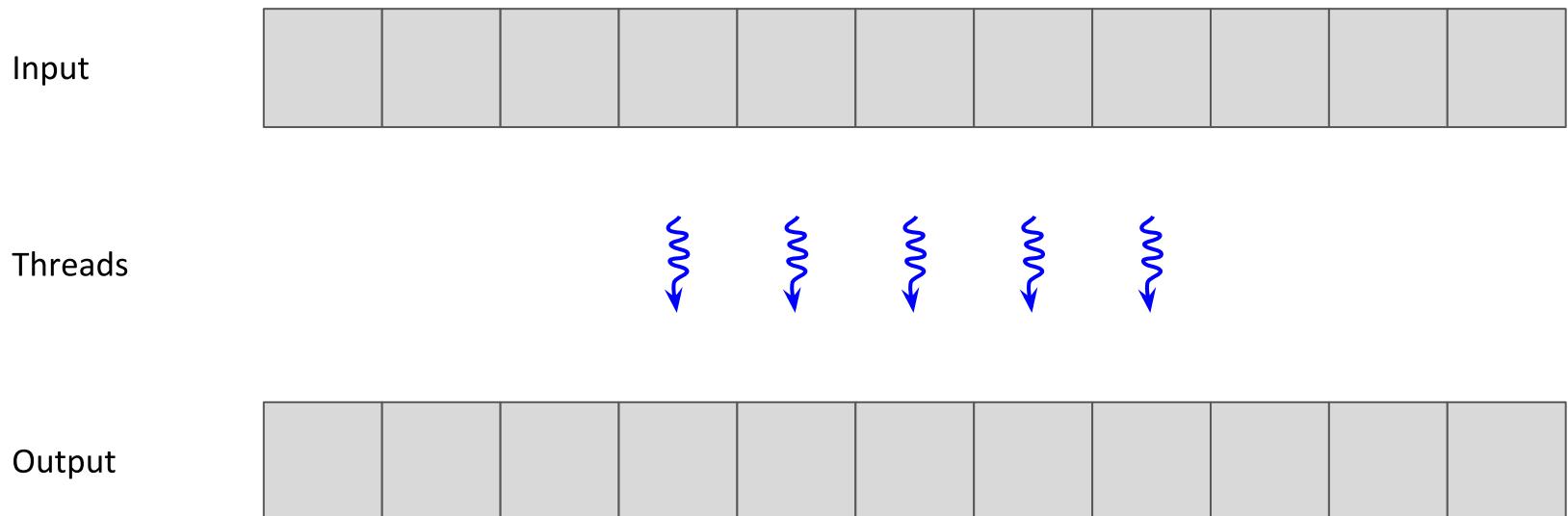
This requires communication, and in parallel computing this is done via memory

Memory



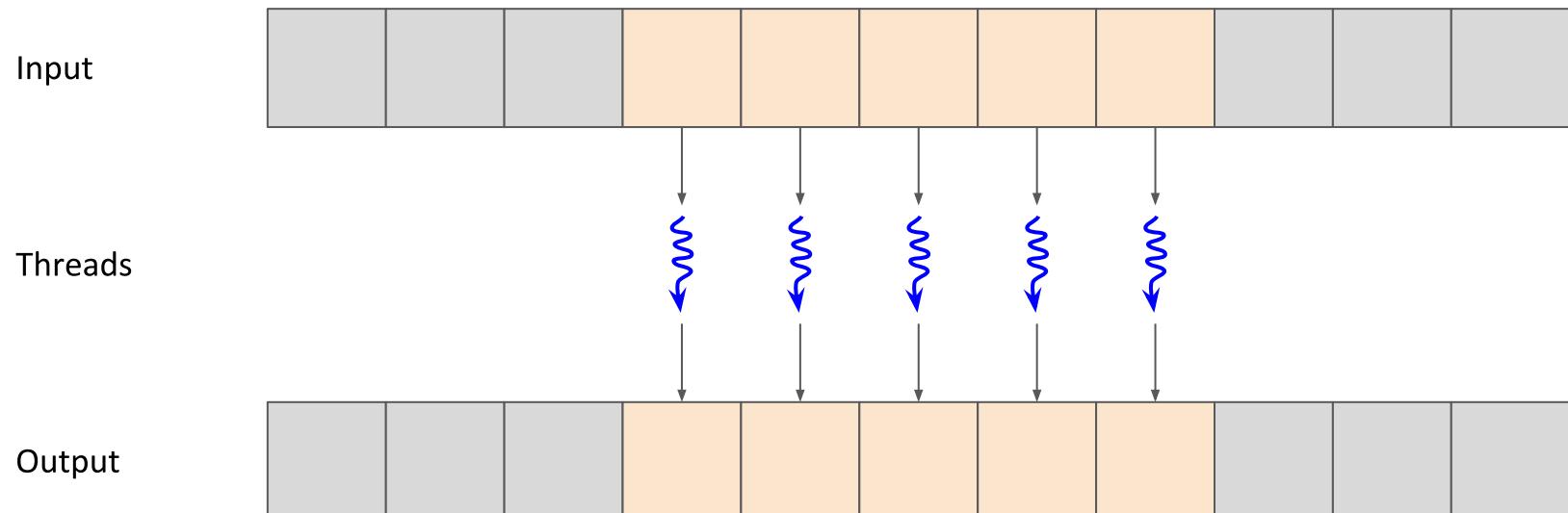
# Communication patterns

Communication patterns are used to describe the relationship between threads and the data they read from and write to



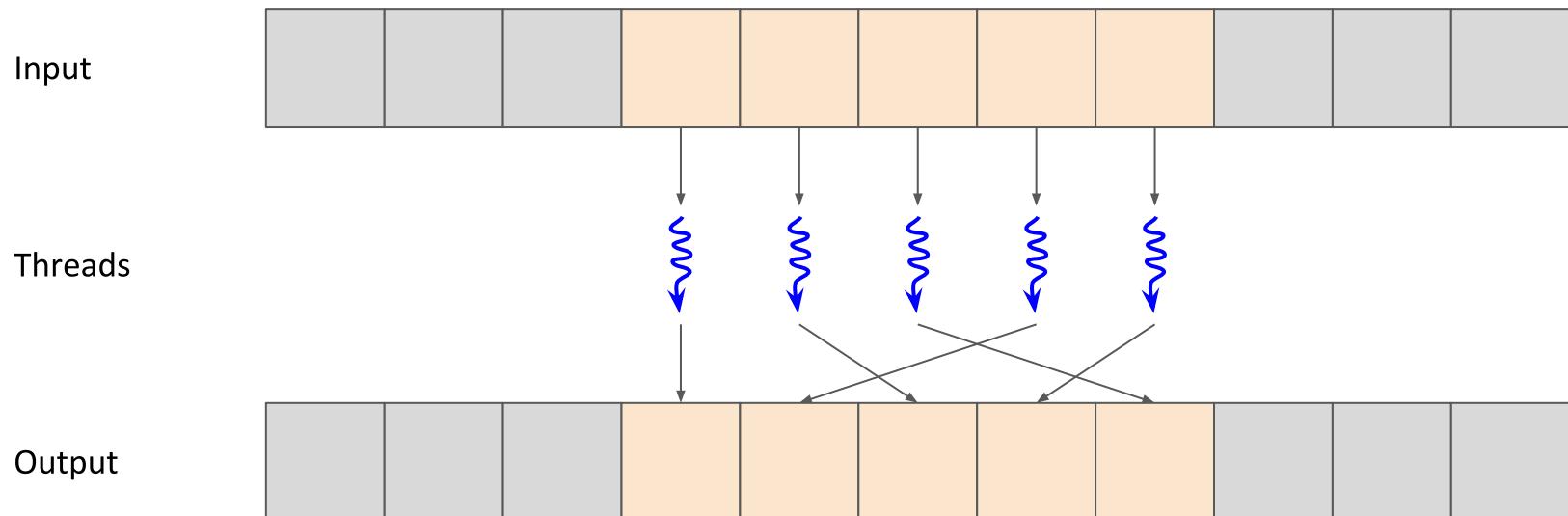
# Map pattern

A map pattern is any operation in which each element of the input range maps to the same element of the output range.



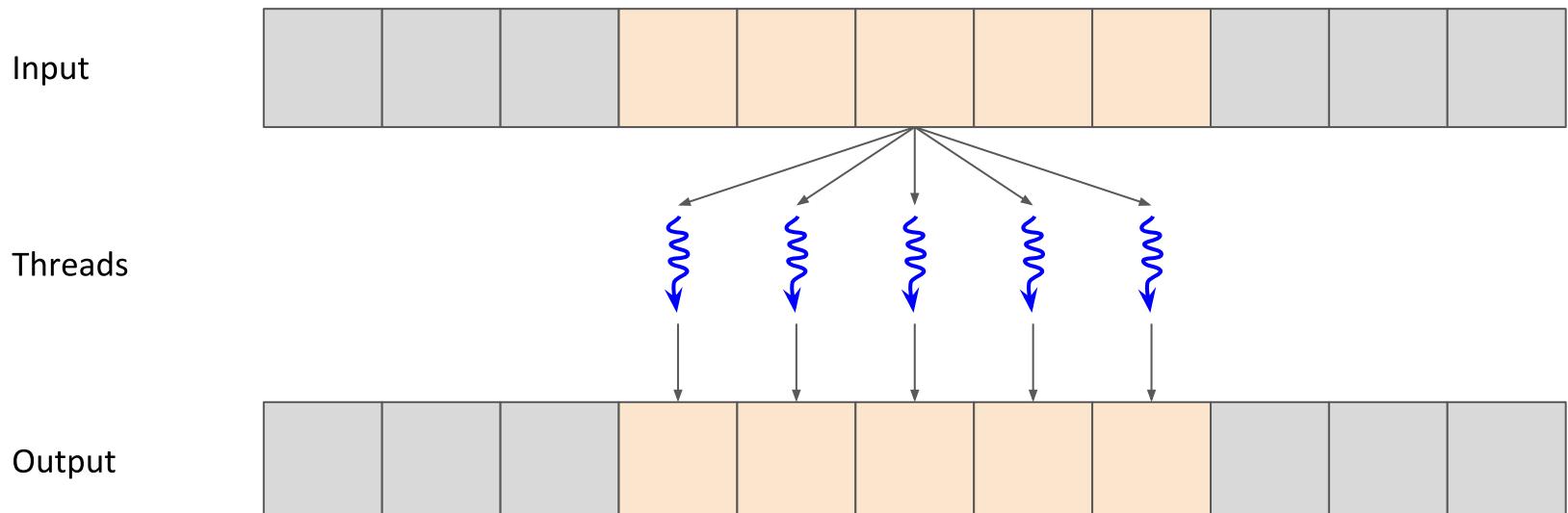
# Transpose pattern

A transpose pattern is any operation in which each element of the input range maps to a different element of the output range.



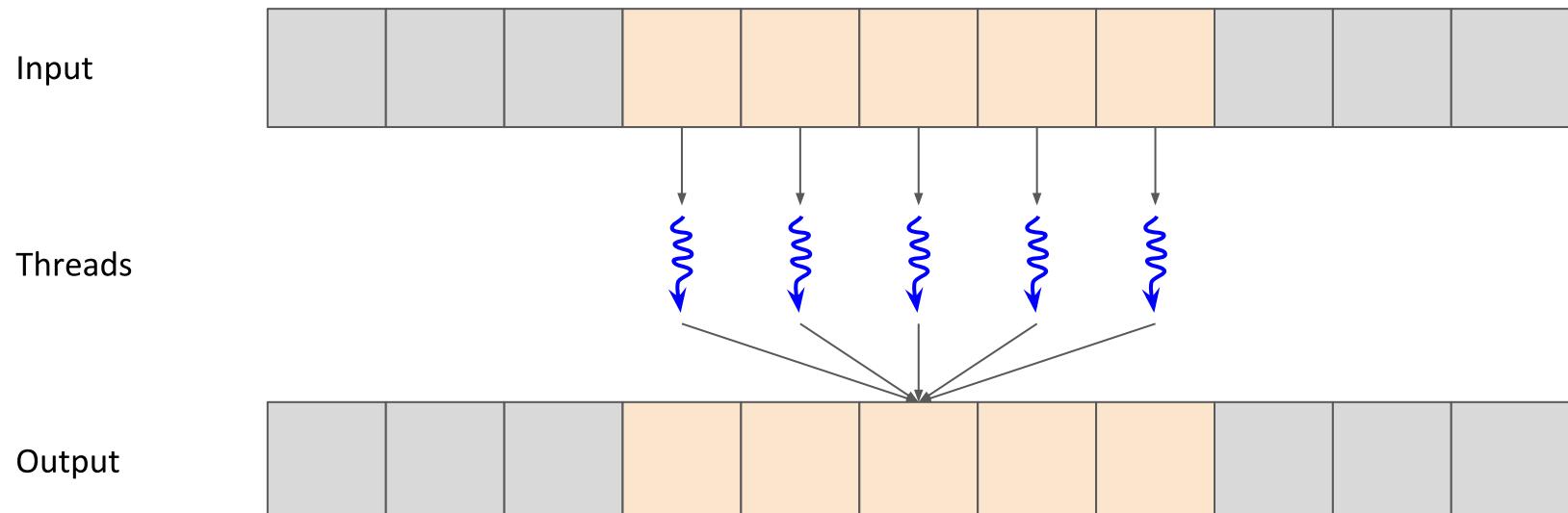
# Scatter pattern

A scatter pattern is any operation in which a single element of the input range maps to multiple elements of the output range.



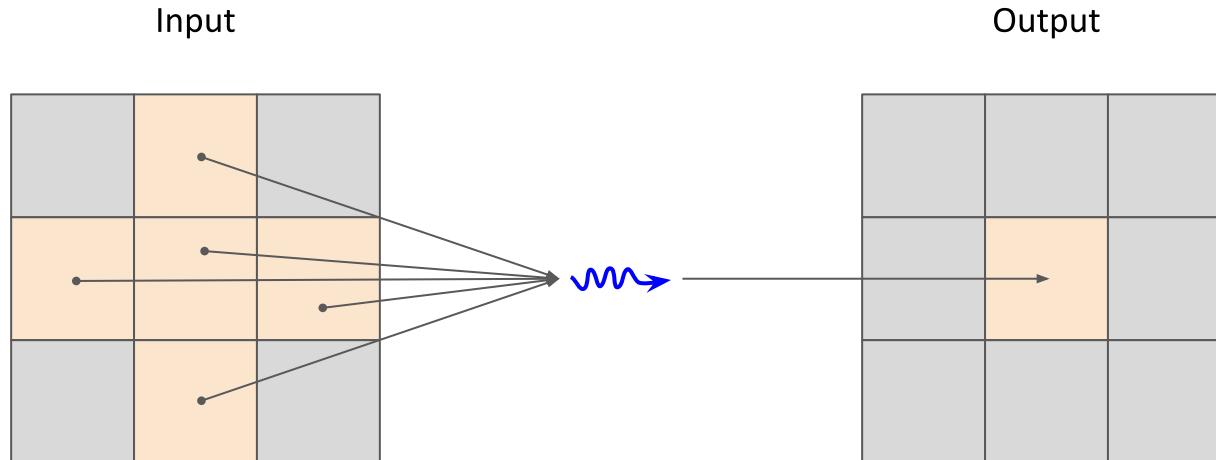
# Gather pattern

A gather pattern is any operation in which multiple elements of the input range maps to a single element of the output range.



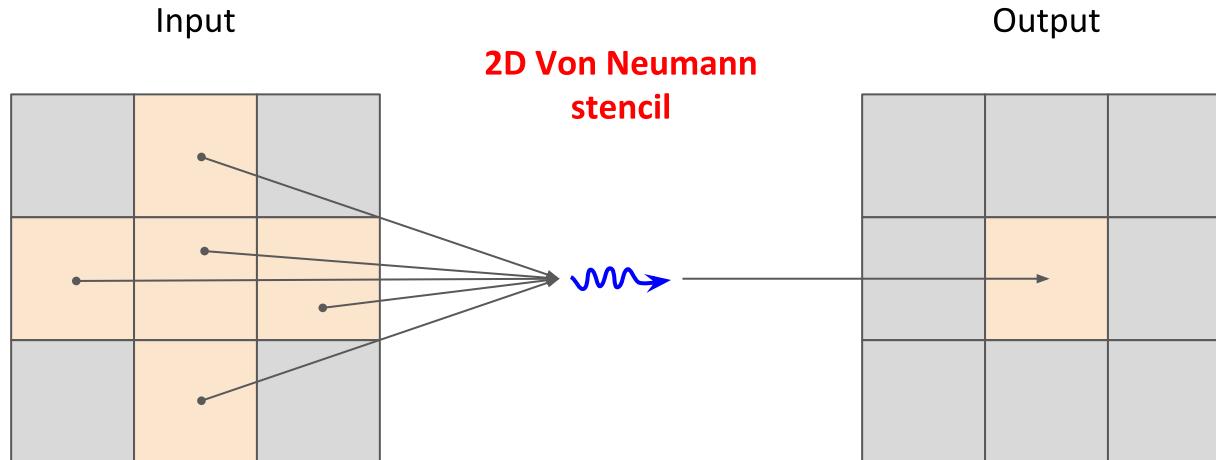
# Stencil pattern

A stencil pattern is a special case of the gather pattern where elements are arranged a multi-dimensional space in which a grouping of elements of the input range maps to a single element of the output range.



# Stencil pattern

A stencil pattern is a special case of the gather pattern where elements are arranged a multi-dimensional space in which a grouping of elements of the input range maps to a single element of the output range.



# What kind of communication pattern does this code embody?

```
1. void foo(int *in, int *out, int index) {  
2.     out[index] = pi * in[128 - index];  
3. }  
4.  
5.  
6.  
7.  
8.
```

Map

Transpose

Scatter

Gather

Stencil

# What kind of communication pattern does this code embody?

```
1. void foo(int *in, int *out, int index) {  
2.     out[index] = pi * in[128 - index];  
3. }  
4.  
5.  
6.  
7.  
8.
```

Map

Transpose

Scatter

Gather

Stencil

# What kind of communication pattern does this code embody?

```
1. void foo(int *in, int *out, int index) {  
2.     out[index] = pi * in[index];  
3. }  
4.  
5.  
6.  
7.  
8.
```

Map

Transpose

Scatter

Gather

Stencil

# What kind of communication pattern does this code embody?

```
1. void foo(int *in, int *out, int index) {  
2.     out[index] = pi * in[index];  
3. }  
4.  
5.  
6.  
7.  
8.
```

Map

Transpose

Scatter

Gather

Stencil

# What kind of communication pattern does this code embody?

```
1. void foo(int *in, int *out, int index) {  
2.     if(index % 2) {  
3.         out[index] = (in[index] + in[index - 1] + in[index + 1]) / 3;  
4.     }  
5. }  
6.  
7.  
8.
```

Map

Transpose

Scatter

Gather

Stencil

# What kind of communication pattern does this code embody?

```
1. void foo(int *in, int *out, int index) {  
2.     if(index % 2) {  
3.         out[index] = (in[index] + in[index - 1] + in[index + 1]) / 3;  
4.     }  
5. }
```

Map

Transpose

Scatter

Gather

Stencil

# What kind of communication pattern does this code embody?

```
1. void foo(int *in, int *out, int index) {  
2.     if(index % 2) {  
3.         out[index - 1] = in[index] / 2;  
4.         out[index + 1] = in[index] / 2;  
5.     }  
6. }  
7.  
8.
```

Map

Transpose

Scatter

Gather

Stencil

# What kind of communication pattern does this code embody?

```
1. void foo(int *in, int *out, int index) {  
2.     if(index % 2) {  
3.         out[index - 1] = in[index] / 2;  
4.         out[index + 1] = in[index] / 2;  
5.     }  
6. }  
7.  
8.
```

Map

Transpose

Scatter

Gather

Stencil

# Let's go back to the holes analogy...



# Say you now have four diggers...

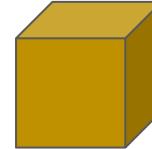
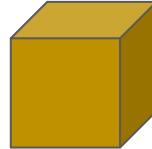
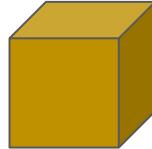
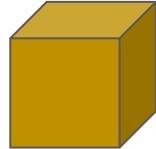


$1m^3 / \text{hour}$

$1m^3 / \text{hour}$

$1m^3 / \text{hour}$

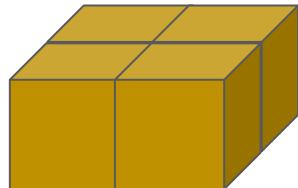
$1m^3 / \text{hour}$



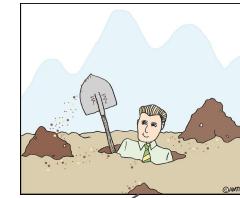
# Say you want a hole with a $4\text{m}^2$ surface area



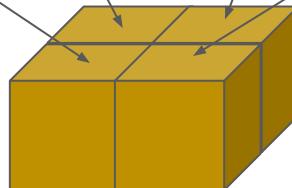
$4\text{m}^2$  surface area  
1m deep



# Each digger digs a part each



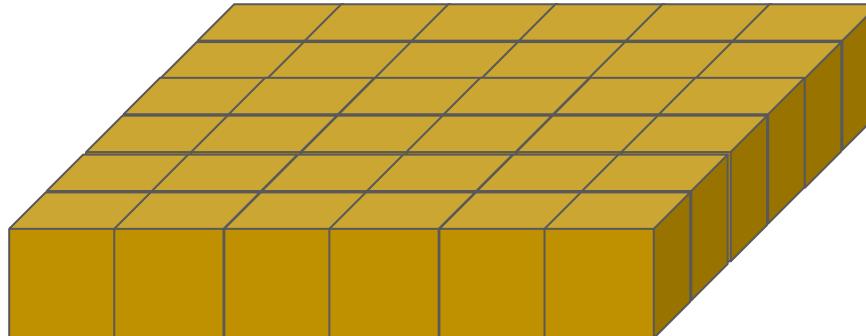
$4m^2$  surface area  
1m deep



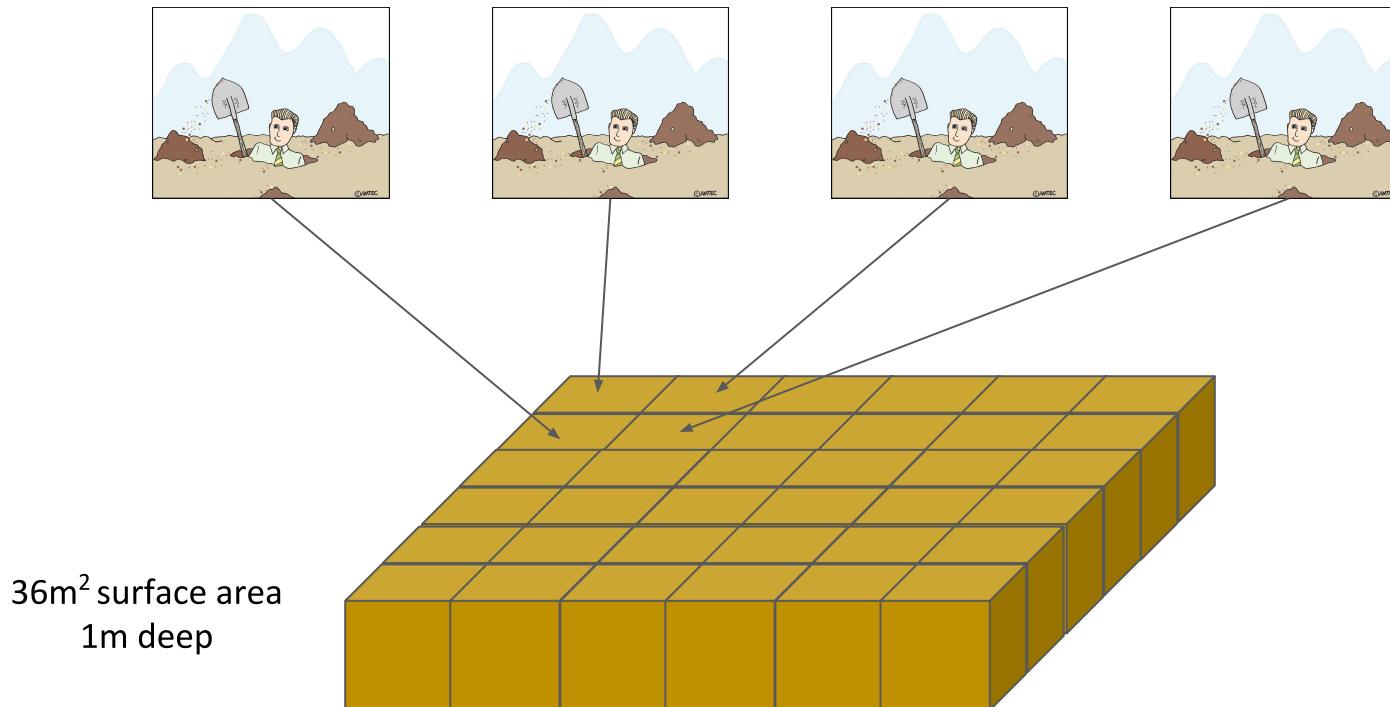
# Say you want a hole with a $36\text{m}^2$ surface area



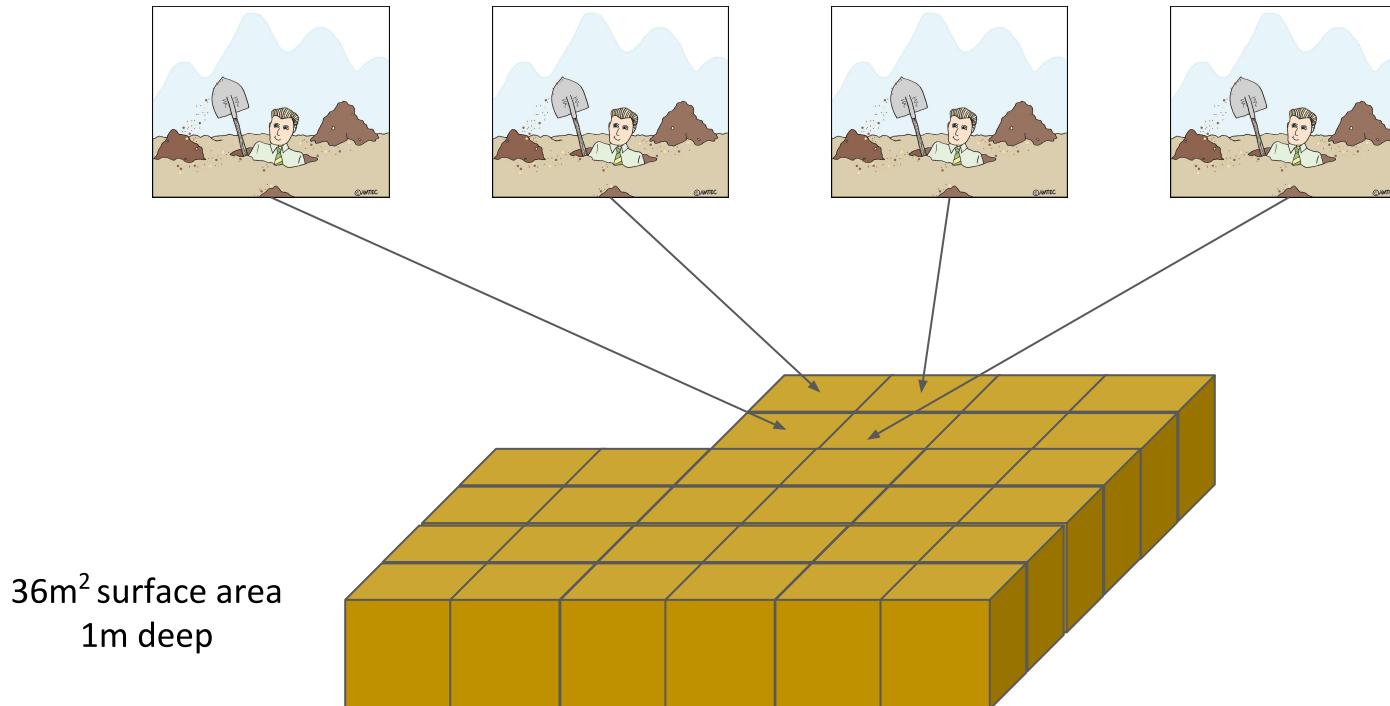
$36\text{m}^2$  surface area  
1m deep



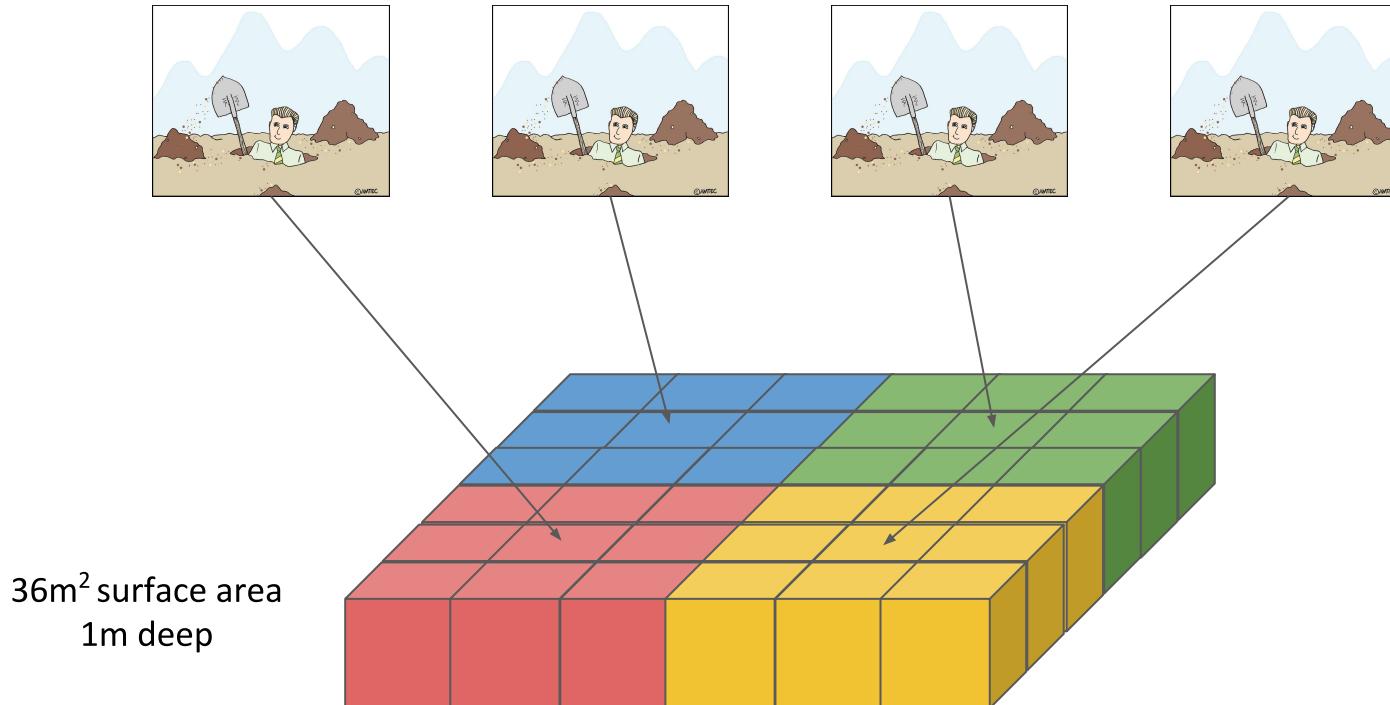
# You can share the work between the diggers



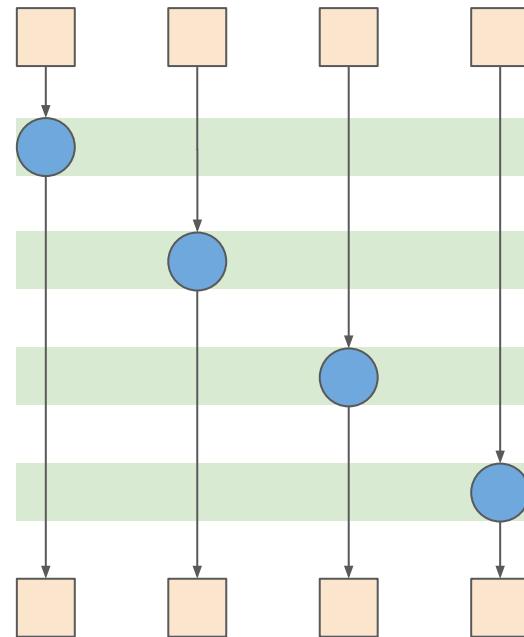
# You can share the work between the diggers



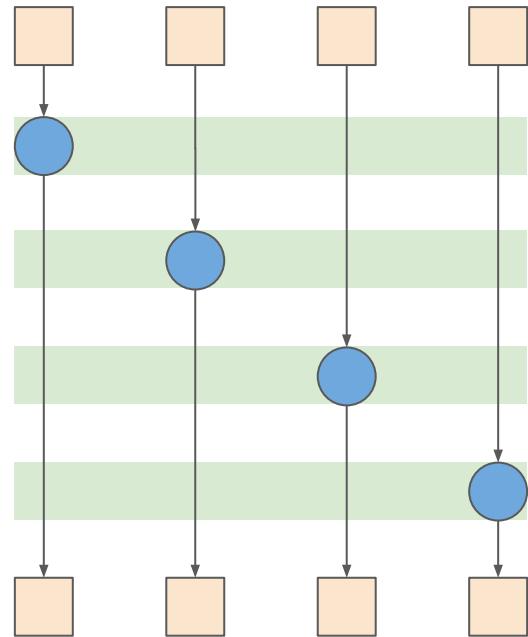
# You can distribute work across the diggers



# This applies to the transform algorithm

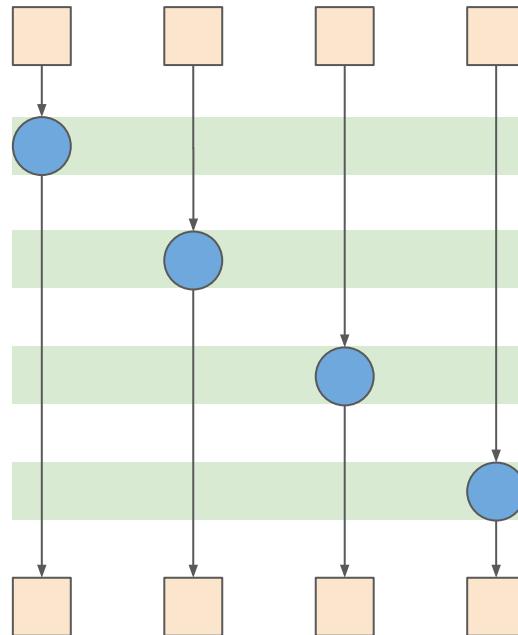


# Let's look at the serial transform...



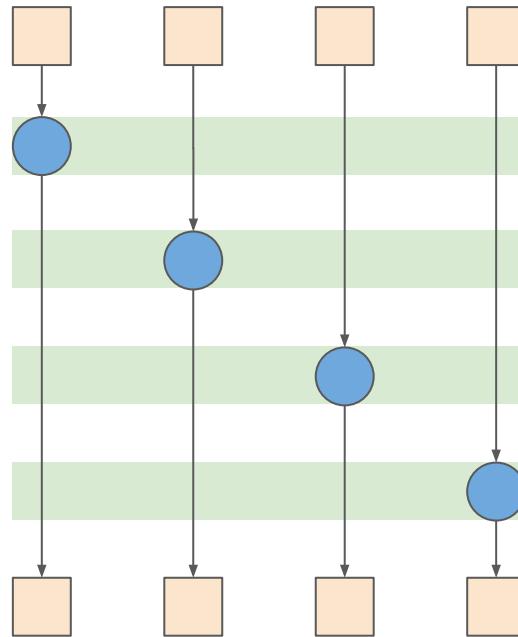
4 elements

# Let's look at the serial transform...



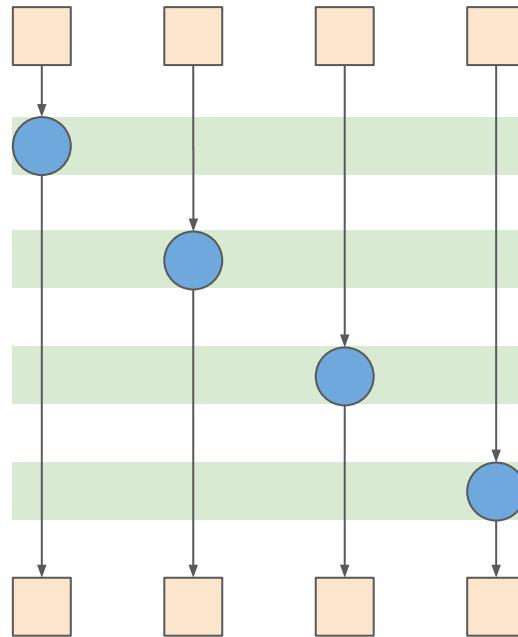
4 elements | 4 Operations

# Let's look at the serial transform...



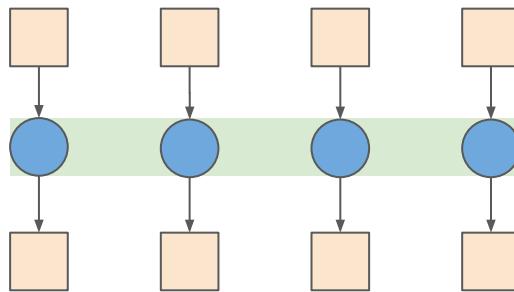
4 elements | 4 Operations | 4 steps

# Let's look at the serial transform...



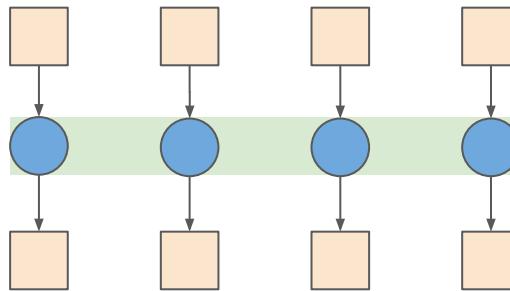
4 elements | 4 Operations| 4 steps | 1 operations / step

# Now let's look at the parallel transform...



4 elements | 4 Operations | **1 step** | **4 operations / step**

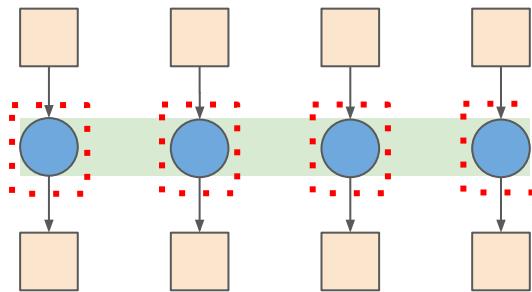
# Now let's look at the parallel transform...



Brent's theorem

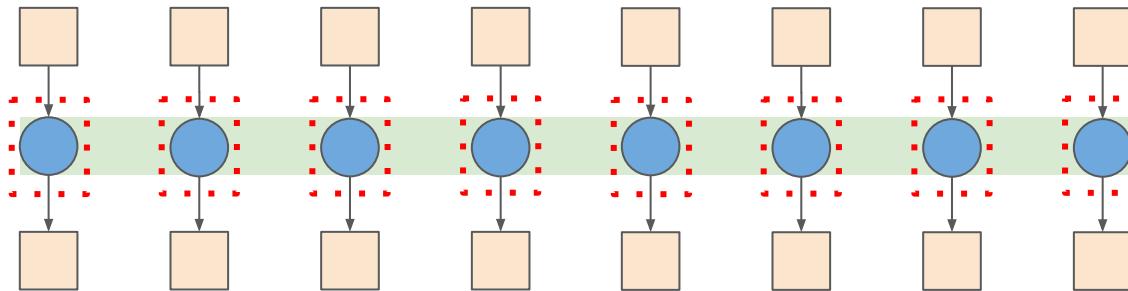
4 elements | 4 Operations | 1 step | 4 operations / step

# In order to do this you need parallel workers



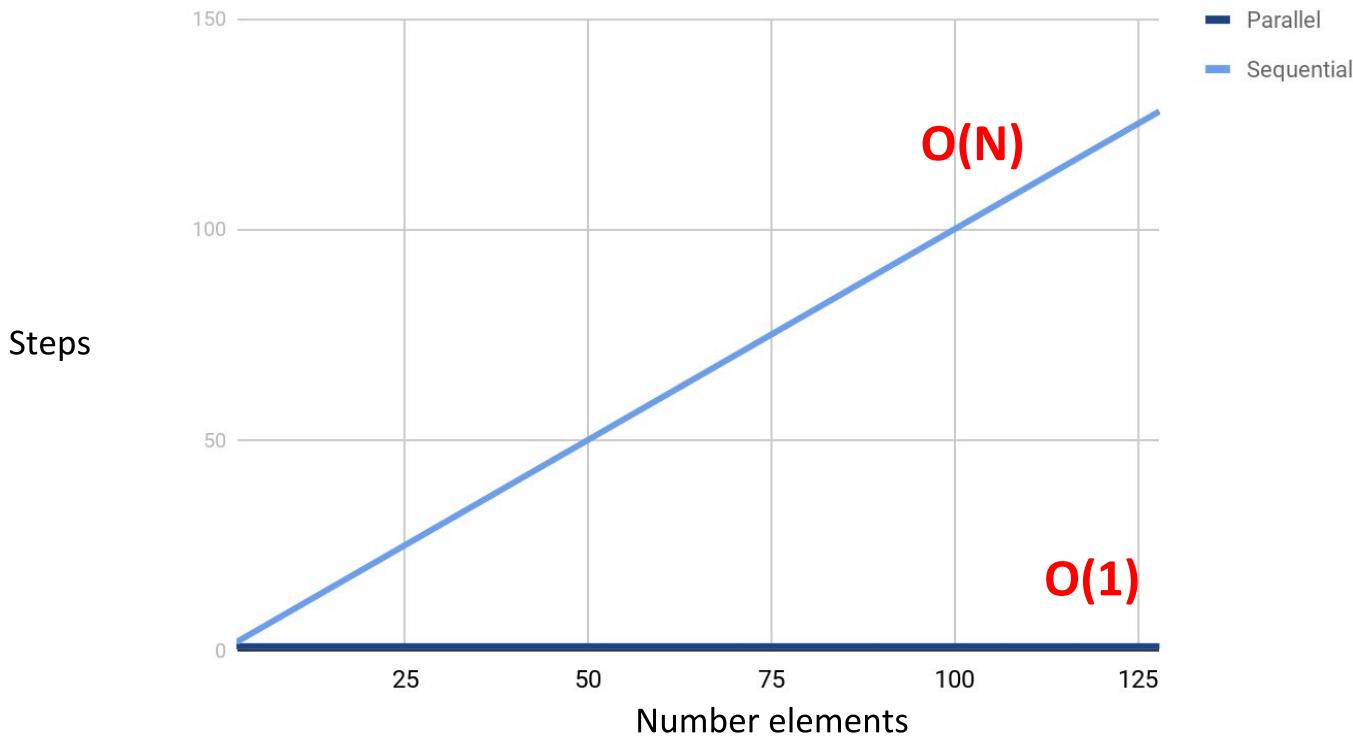
4 elements | 4 Operations | **4 workers** | 1 steps | 4 operations / step

# Now let's scale this up...

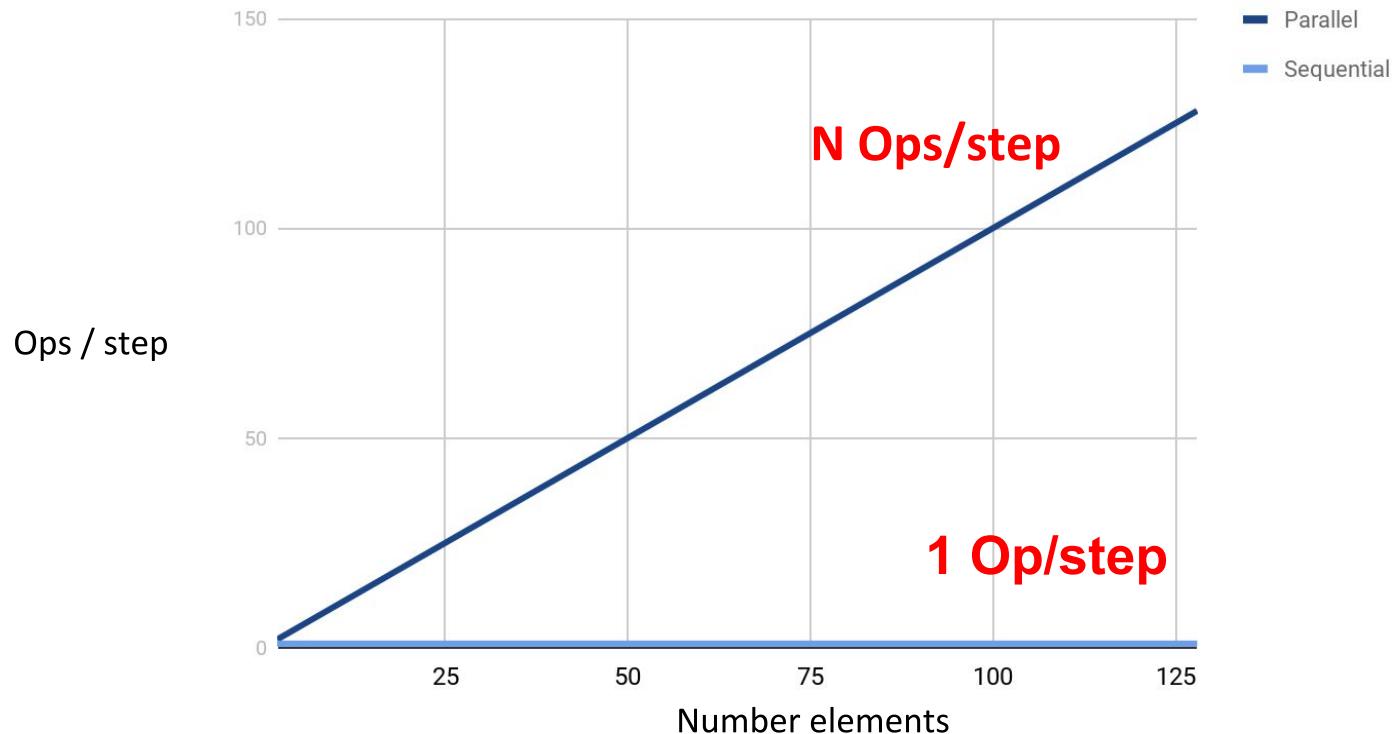


**8 elements | 8 Operations | 8 workers | 1 step | 8 operations / step**

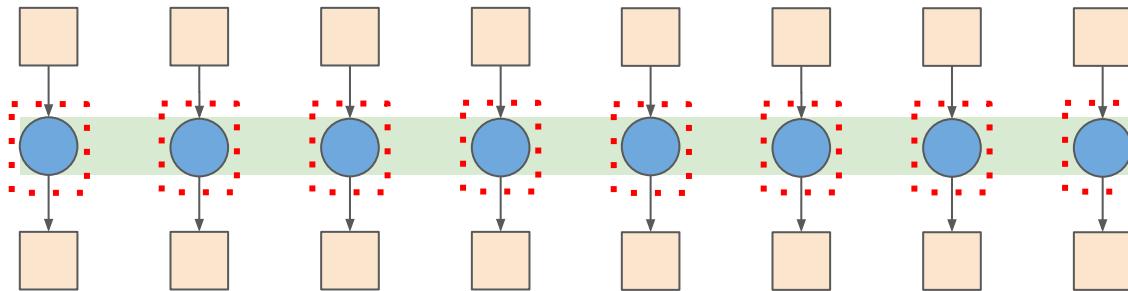
# Step complexity of transform



# Theoretical operations per step

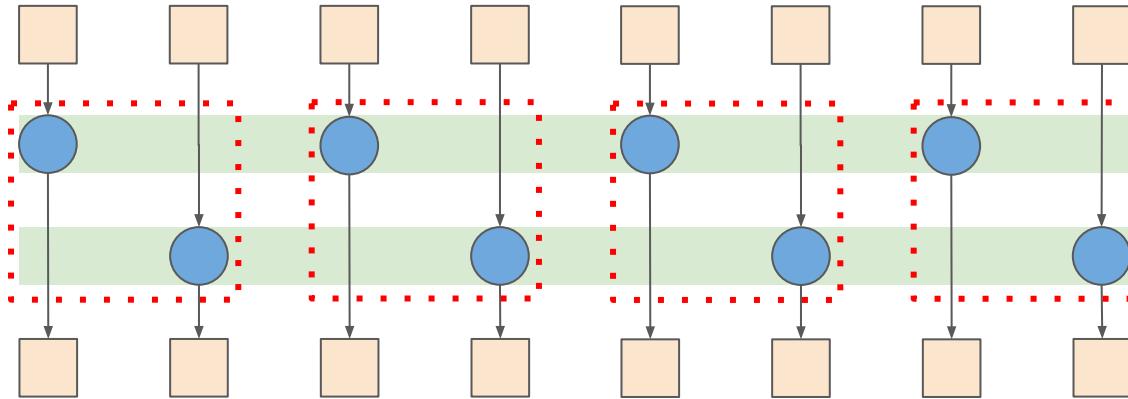


# What happens if you only have 4 workers?



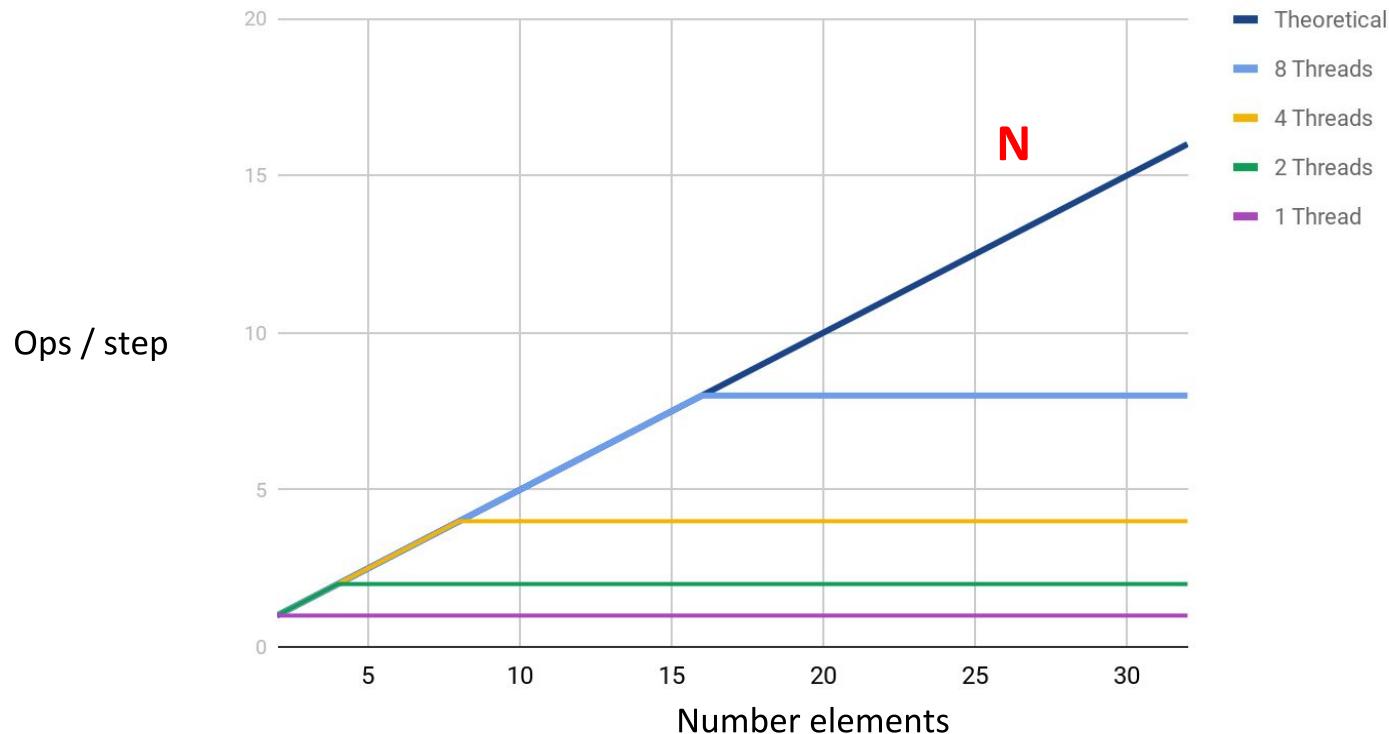
8 elements | 8 Operations | 8 workers | 1 step | 8 operations / step

# You have to batch work together



8 elements | 8 Operations | **4 workers** | 2 steps | **4 operations / step**

# Actual operations per step



# Maximising throughput

The theoretical operations / step is always limited by the available workers

Maximising the actual operations / step will provide optimal throughout

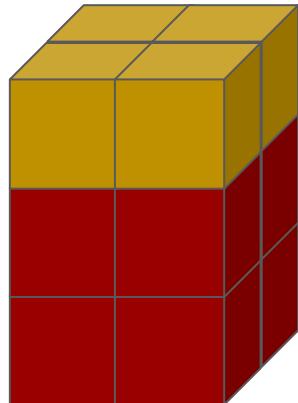
You will most often have a much larger number of operations to perform than available workers

How you perform this batching may differ depending on the architecture you are executing on

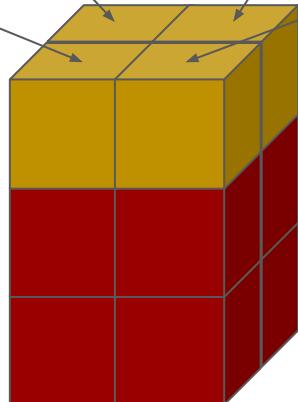
# Say you want to dig a hole 3m deep



4m<sup>2</sup> wide  
3m deep



# All four diggers have work to do

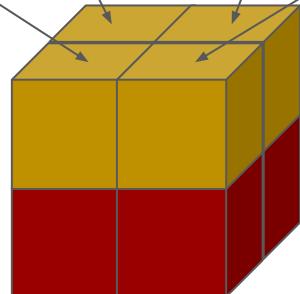


4m<sup>2</sup> wide  
3m deep

# All four diggers have work to do



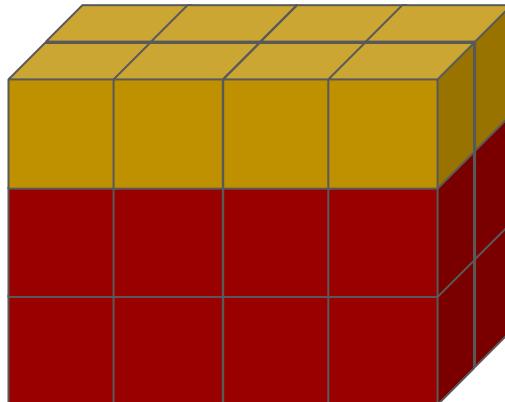
4m<sup>2</sup> wide  
3m deep



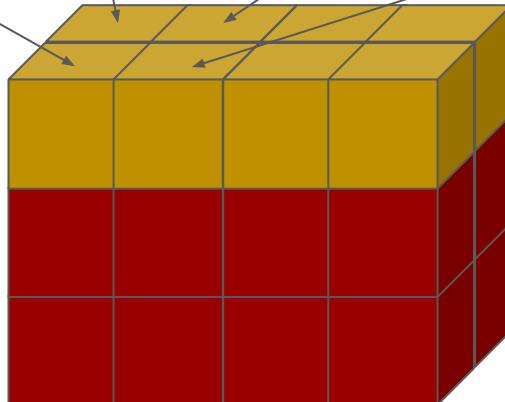
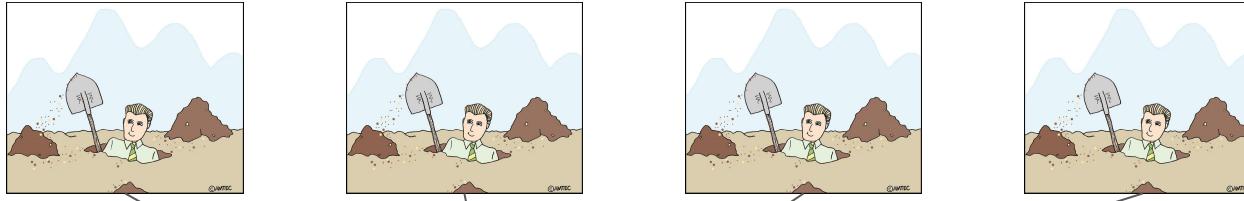
# Now say the hole is $8\text{m}^2$ wide



$8\text{m}^2$  wide  
3m deep

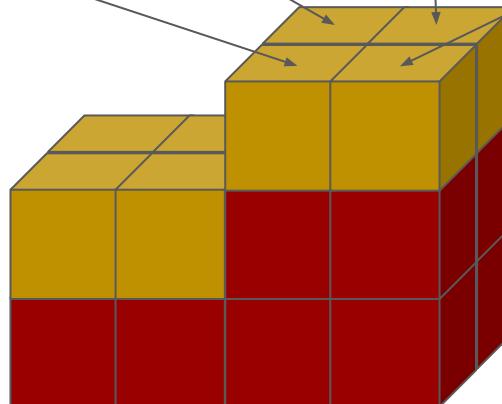
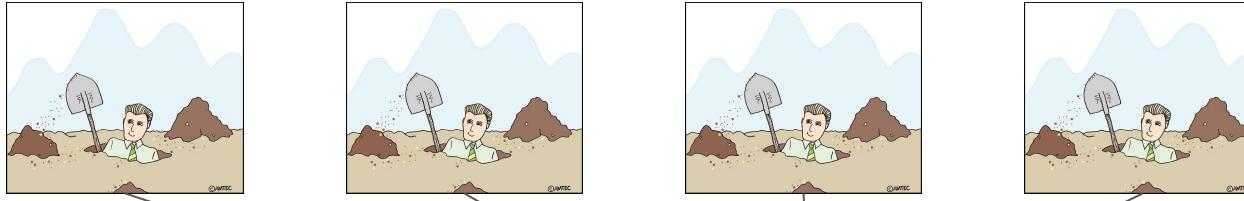


# Again can share the work



8m<sup>2</sup> wide  
3m deep

# Again can share the work

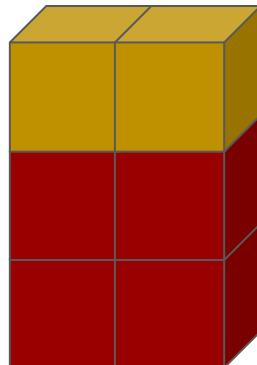


8m<sup>2</sup> wide  
3m deep

# Now say the hole 2m<sup>2</sup> wide



2m<sup>2</sup> wide  
3m deep



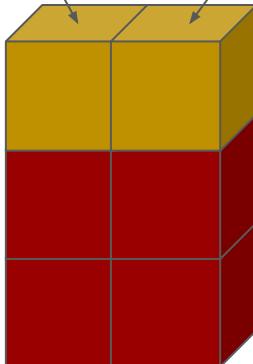
# Now you have diggers with no work to do



on break



on break



2m<sup>2</sup> wide  
3m deep

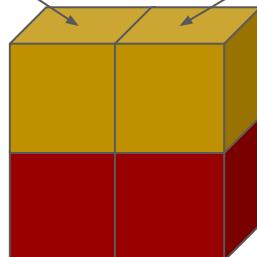
# Now you have diggers with no work to do



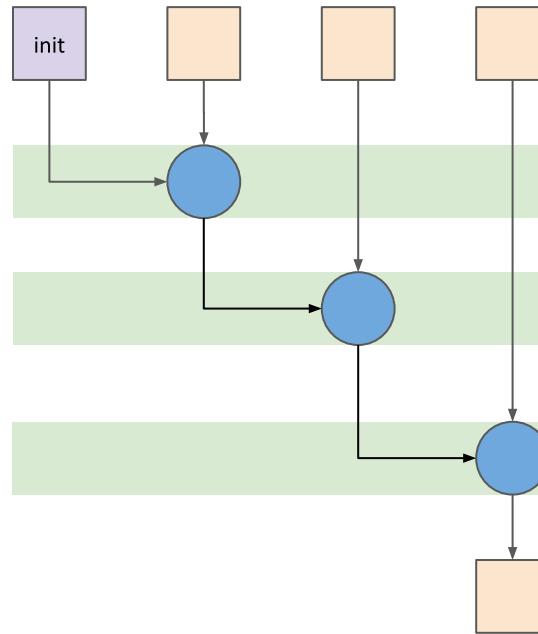
on break

on break

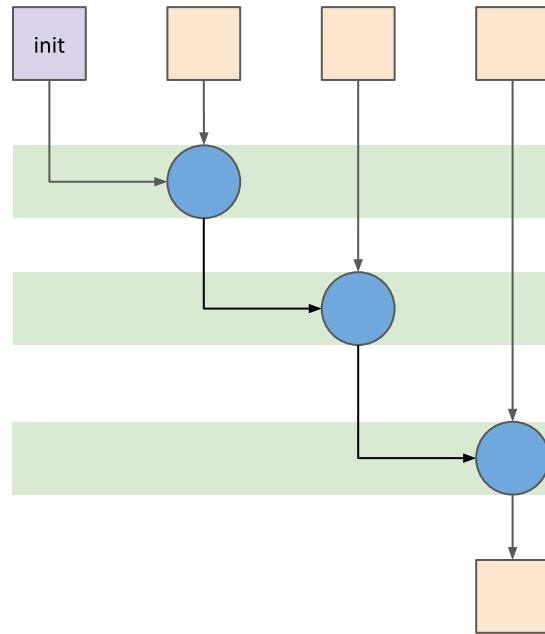
2m<sup>2</sup> wide  
3m deep



# This applies to the reduce algorithm

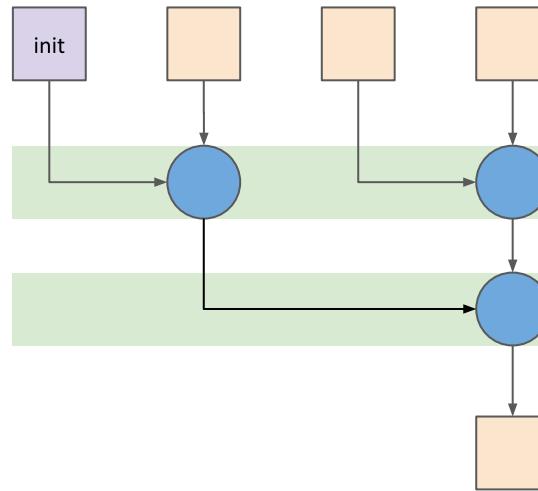


# Let's look at the serial reduce...



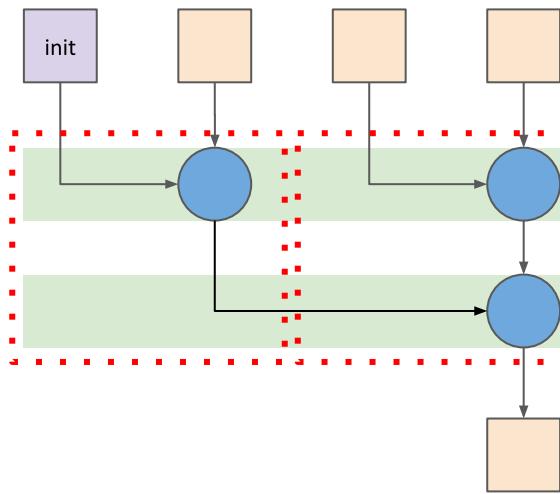
3 elements | 3 Operations| 3 steps | 1 operations / step

# Now let's look at the parallel reduce...



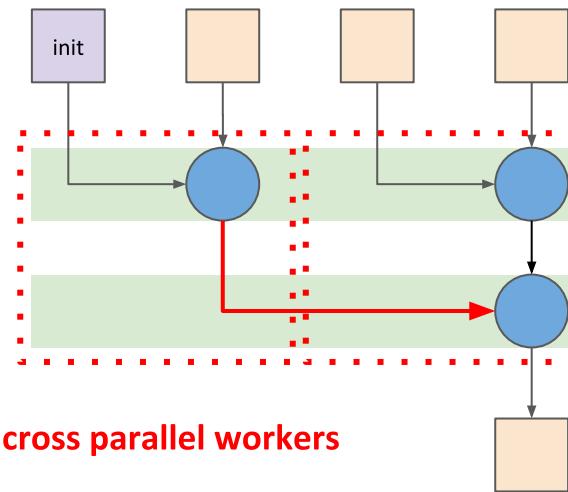
3 elements | 3 Operations | **2 steps** | **1.5 operations / step**

# Let's try to distribute this work



3 elements | 3 Operations | 2 workers | 2 steps | 1.5 operations / step

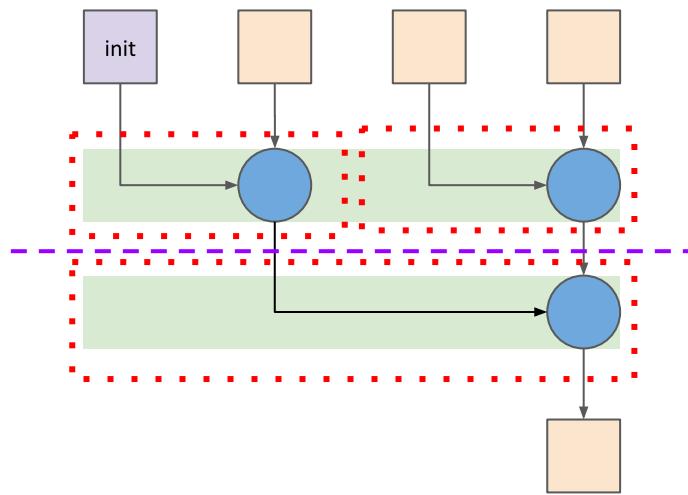
# This creates a dependency between workers



**Dependency across parallel workers**

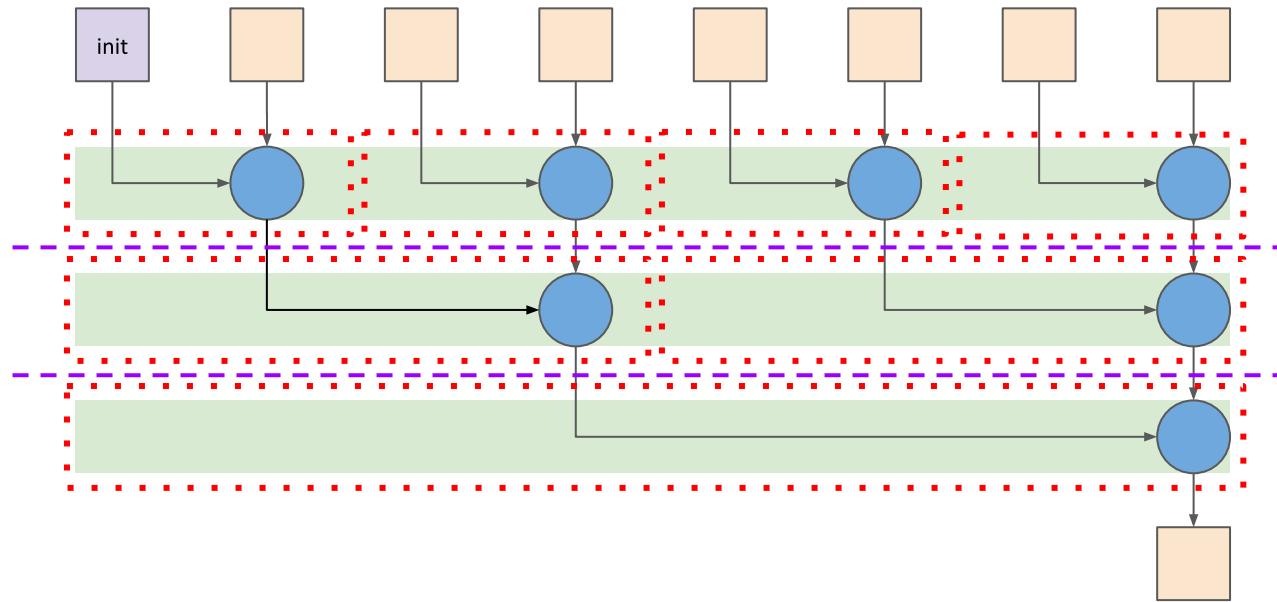
3 elements | 3 Operations | 2 workers | 2 steps | 1.5 operations / step

# This creates a dependency between workers



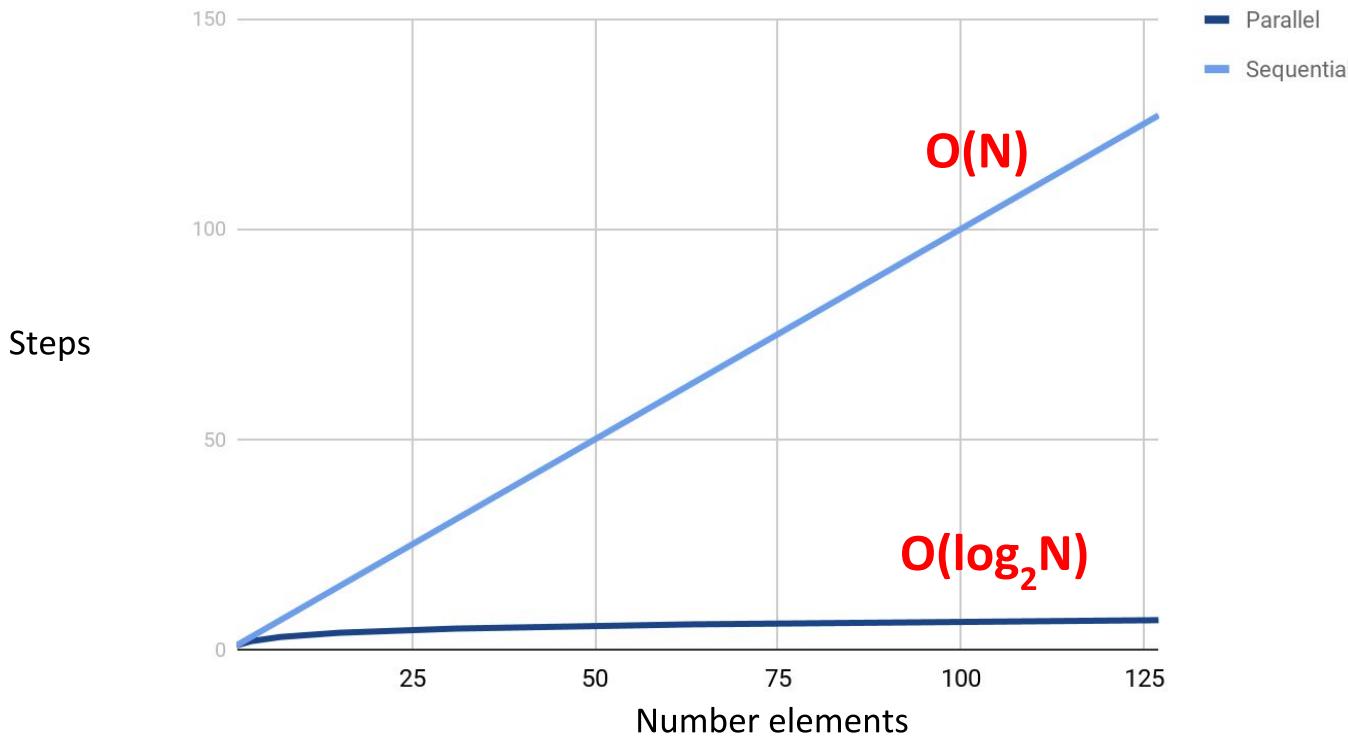
3 elements | 3 Operations | 2 workers | 2 steps | 1.5 operations / step

# Now let's scale this up for 4 workers

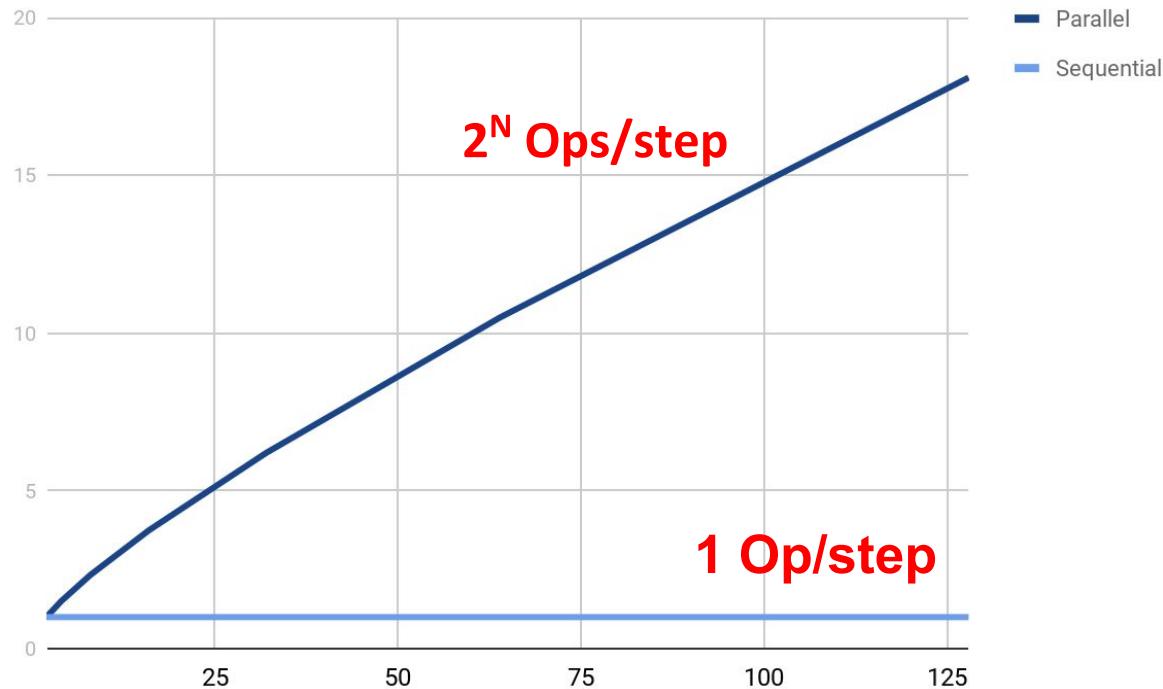


7 elements | 7 Operations | 4 workers | 3 steps | 2.3 operations / step

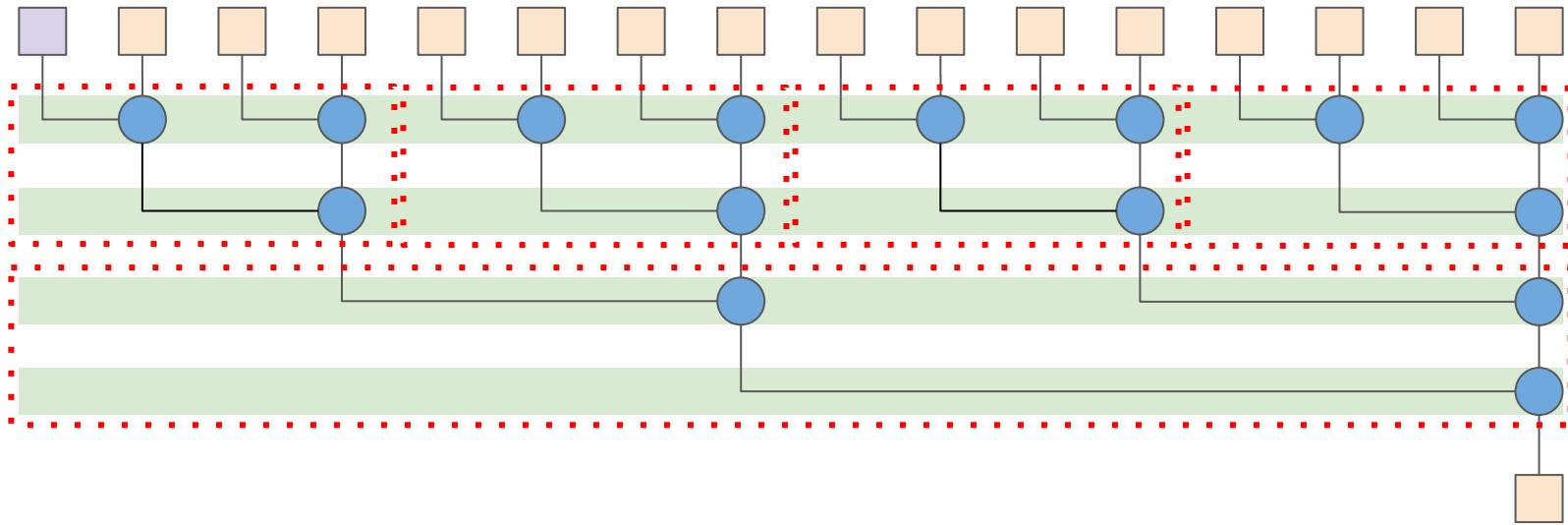
# Step complexity of reduce



# Theoretical operations per step

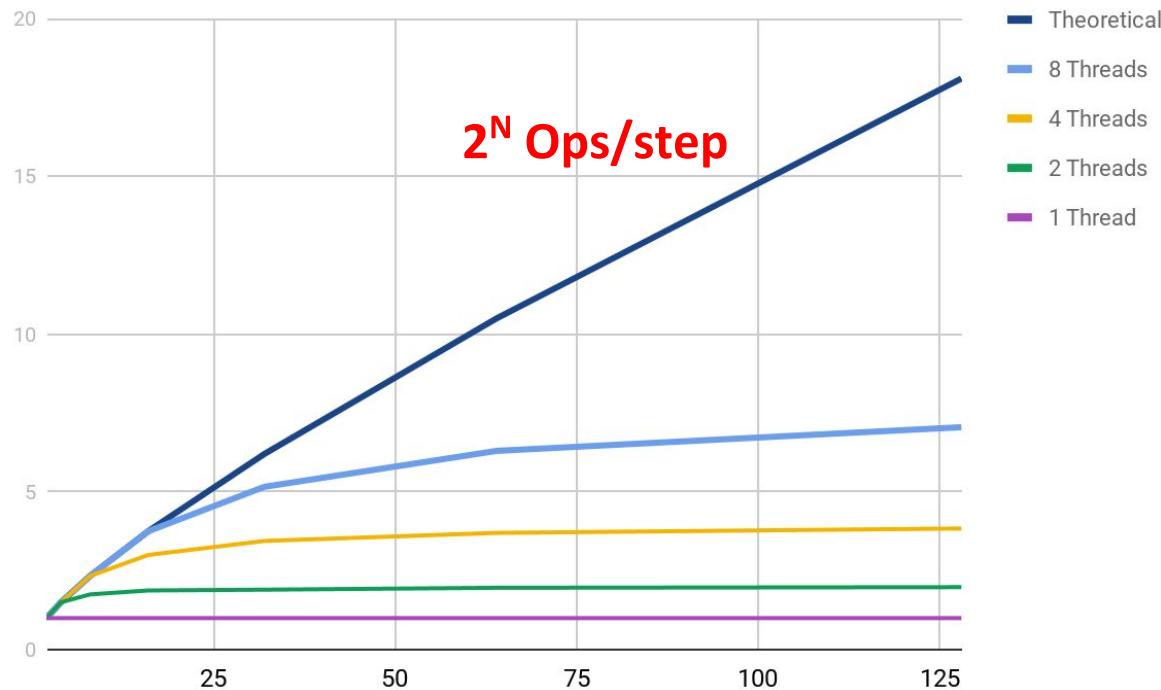


# Now let's scale this up for 4 workers



**15 elements | 15 operations | 4 workers | 4 steps | 3.8 operations / step**

# Actual operations per step



# Handling dependencies

You should structure your algorithm to distribute dependencies

You should avoid dependencies across workers in the same step

When you have dependencies between operations this creates dependencies between workers

These dependencies often have different implications depending on the architecture you are executing on

# Task vs data parallelism



Task parallelism:

- Few large tasks with different operations / control flow
- Optimized for latency

Data parallelism:

- Many small tasks with same operations on multiple data
- Optimized for throughput

# Key takeaways

Designing your algorithm to maximise the operations per step will improve throughput and utilisation of the hardware

Designing your algorithm to distribute dependencies between operations will reduce increase operations per step and improve throughput

How you batch work together on workers and how you handle dependencies will vary depending on the architecture you are executing on



# Chapter 4: Performance Portability

## Prerequisites:

1. Previous chapters
2. Want performance
3. Also need to be portable
4. Want to be productive

## You will learn:

1. Golden Triangle of Parallel programming
2. What are the goals of parallel programming
3. How to iterate through parallel programming tasks

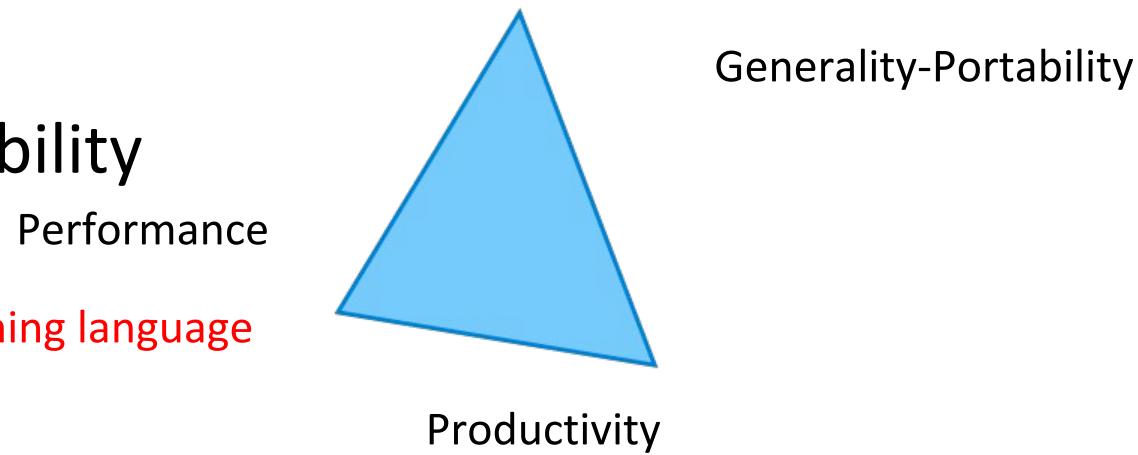
# Step Back

So what are the goals of multithreaded programming

And how do you do it in general?

# So What are the Goals?

- Goals of Parallel Programming over and above sequential programming
  1. Performance
  2. Productivity
  3. Generality-Portability



- Oh, really??? What about correctness, maintainability, robustness, and so on?

- And if correctness, maintainability, and robustness don't make the list, why do productivity and generality?

- Given that parallel programs are much harder to prove correct than are sequential programs, again, shouldn't correctness really be on the list?

- What about just having fun?

# Performance

- Broadly includes scalability and efficiency
- If not for performance why not just write sequential program?
- parallel programming is primarily a performance optimization, and, as such, it is one potential optimization of many.

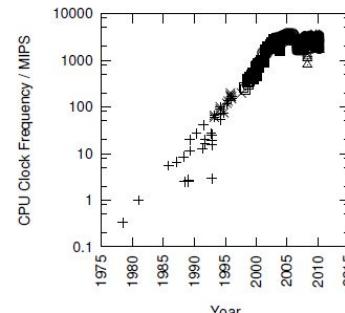
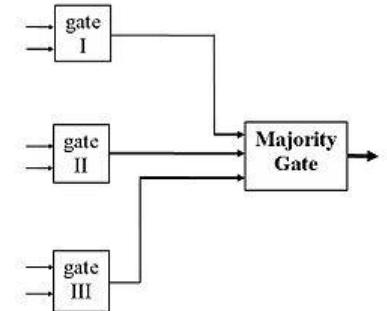


Diagram thanks to Paul McKenney

- Are there no cases where parallel programming is about something other than performance?



# Productivity

- Perhaps at one time, the sole purpose of parallel software was performance. Now, however, productivity is gaining the spotlight.

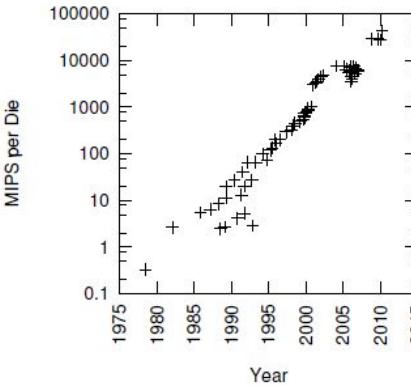
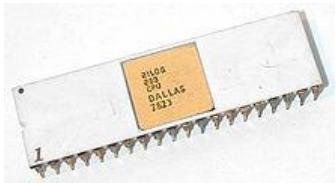


Diagram thanks to Paul McKenney

- Why all this prattling on about non-technical issues??? And not just any non-technical issue, but productivity of all things? Who cares?

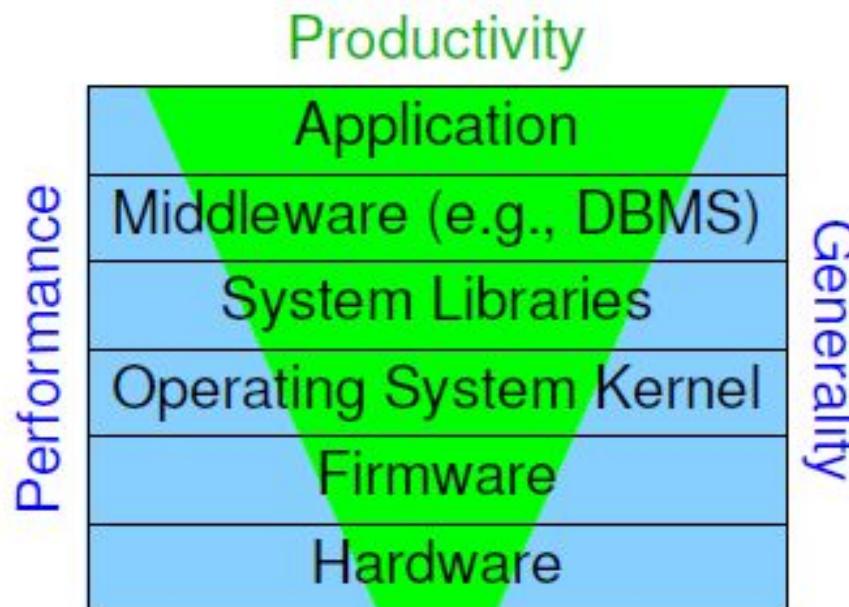


- Given how cheap parallel systems have become, how can anyone afford to pay people to program them?



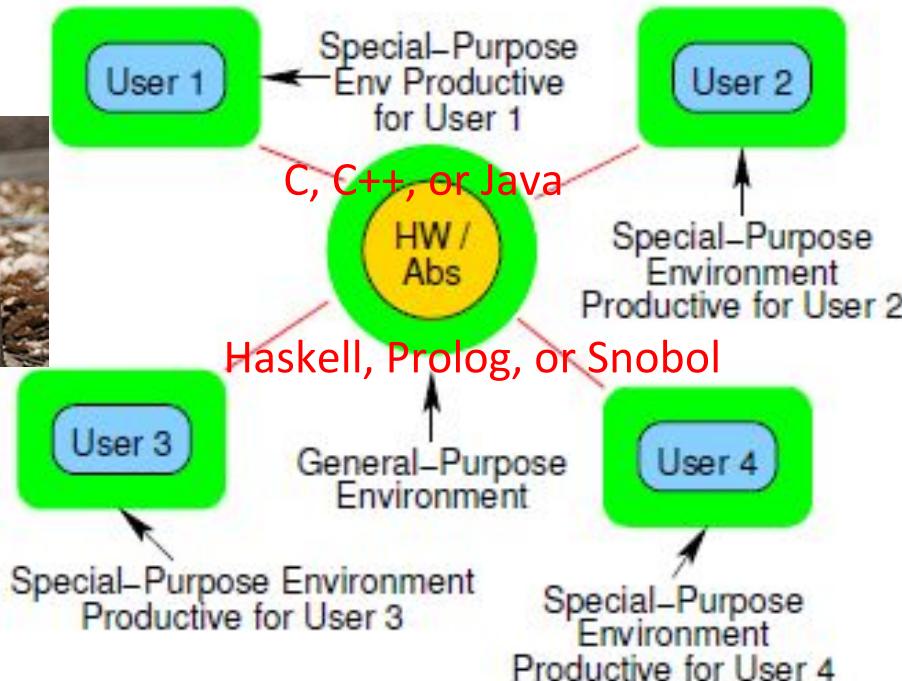
# Generality (Portability)

- C/C++ locking+threads
- Java
- MPI
- OpenMP
- SQL



Until such a nirvana appears, it will be necessary to make engineering tradeoffs

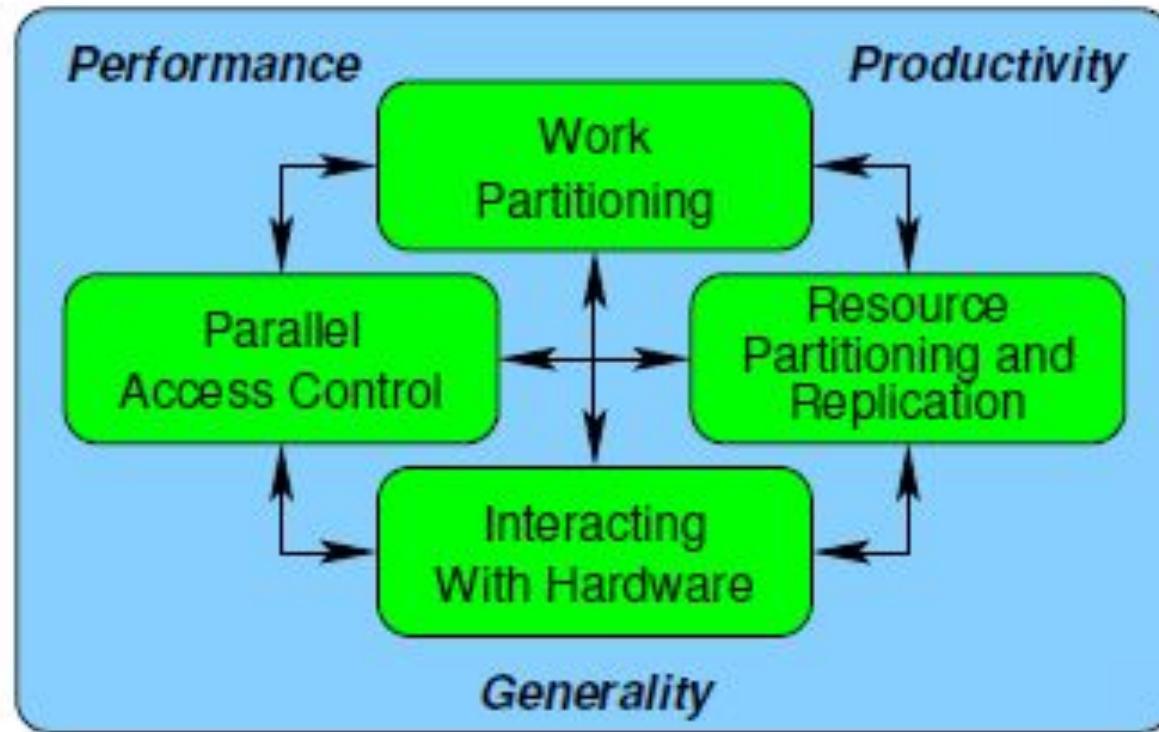
# Tradeoff Generality and Performance





- This is a ridiculously unachievable ideal! Why not focus on something that is achievable in practice?

# Parallel programming tasks



# Work Partitioning

- Greatly increase performance, scalability but can greatly increase complexity
- permitting threads to execute concurrently greatly increases the program's state space, which can make the program difficult to understand and debug
  - Can decrease productivity

- Other than CPU cache capacity, what might require limiting the number of concurrent threads?

# Parallel Access Control

- Does access depend on resource location?
- How does thread coordinate access to the resource?

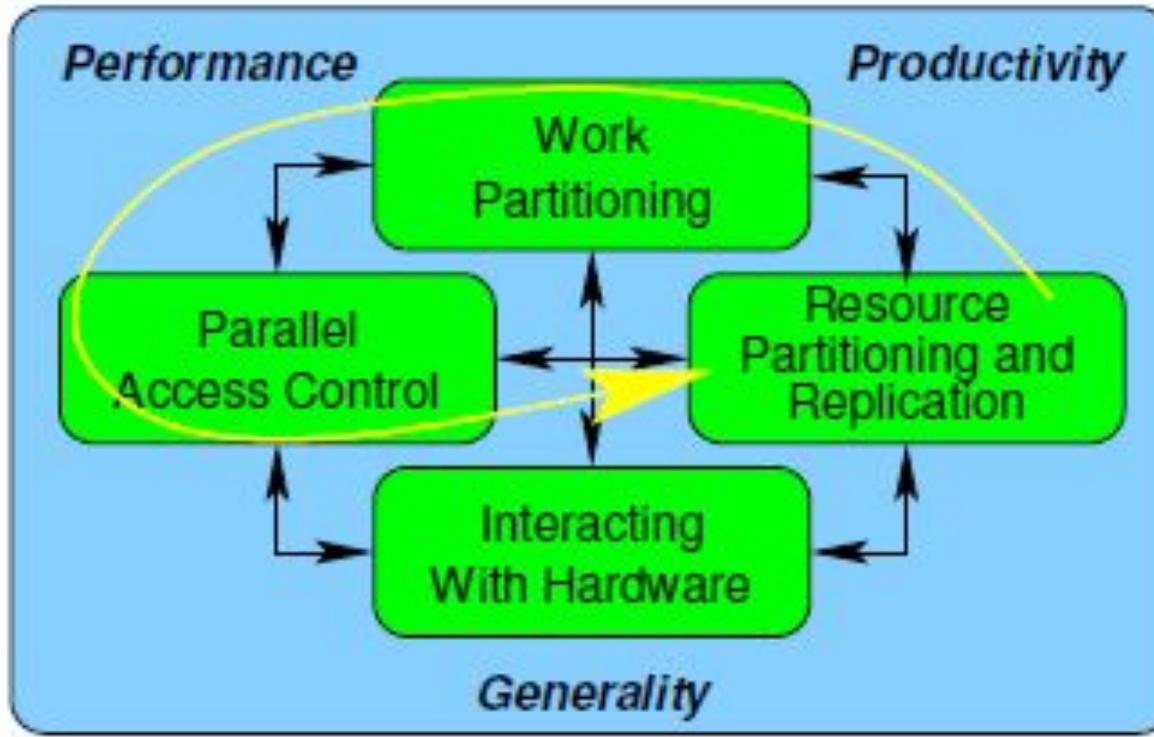
# Resource partitioning and replication

- Data may be partitioned over computers, disks, NUMA nodes, CPU cores, pages, cache lines, instances of synchronization primitives, or critical section of code
- Resource partitioning is frequently application dependent

# Interacting with Hardware

- developers working with novel hardware features and components will often need to work directly with such hardware
- direct access to the hardware can be required when squeezing the last drop of performance out of a given system

# Composite Capabilities



# Key takeaways

Iron Triangle of Parallel programming language

Remember how to iterate through parallel programming tasks



# Chapter 5: C++ Multi-threading

## Prerequisites:

1. Need to know intermediate level C++
2. Understand constructors, destructors, move constructor/assignment, copy constructor/assignment
3. Understand variadic templates, variadic parameter, parameter packs

## You will learn:

1. Understand how to create threads and wait on threads.
  - a. Launch, join, detach,
2. Understand how to share data between threads.
  - a. Pass parameters to thread
3. Invariants, Lifetime issues, deadlock
4. Mutexes and condition variables
5. Async and futures
6. Parallel reduction using threads

# Parallelism and Concurrency in C11/C++11/C++14/C++17

- C99/C++98: does not have parallelism or concurrency support
- C++11 have multithreading support
  - Memory model, atomics API
  - Language support: TLS, static init, termination, lambda function
  - Library support: thread start, join, terminate, mutex, condition variable
  - Advanced abstractions: basic futures, thread pools
- C11 will have similar memory model, atomics API. TLS, static init/termination
  - Some minor differences like `__Atomic` qualifier
- C++14 enhanced the memory model with better definition of lock-free vs obstruction-free
  - Clarified `atomic_signal_fence`
  - Improved on out-of-thin-air results
  - Realized consume ordering as described was problematic
- C++17 we added
  - Define strength ordering of memory models
  - **ParallelSTL**, progress guarantees

# So Why do we need to standardize concurrency?

- Reflects the real world
  - Multi-core processors
  - Solutions for very large problems
  - Internet programming
- Standardize existing practice
  - C++ threads=OS threads
  - shared memory
  - Loosely based on POSIX, Boost thread
  - Does not replace other specifications
    - MPI, OpenMP, UPC, autoparallelization, many others
- Can help existing advanced abstractions
  - TBB, PPL, Cilk,

# Concurrency Language and Library

- Core Language: what does it mean to share memory and how it affects variables
  - TLS
  - Static duration variable initialization/destruction
  - **Memory model**
  - **Atomic types and operations**
  - **Lock-free programing**
  - Fences
- Library
  - **How to create/synchronize/terminate threads,**
  - **Thread , mutex , locks**
  - **RAII for locking, type safe**
  - **propagate exceptions**
  - few advanced abstraction
  - **Async() , promises and futures**
  - **parallel STL**

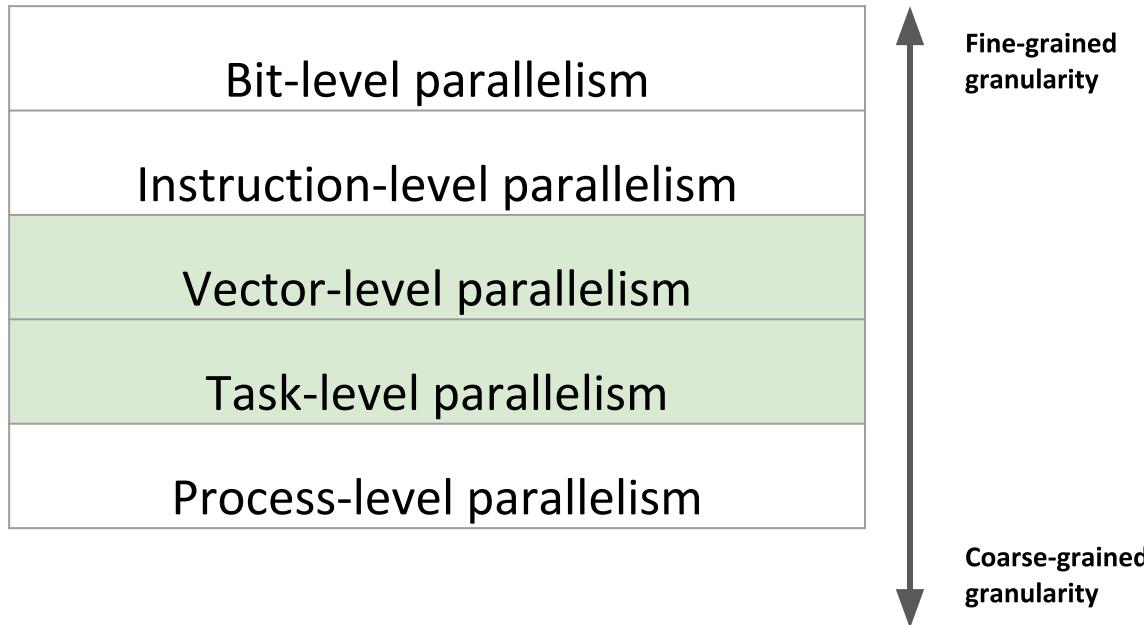
# What we got in C++

- Low level support to enable higher abstractions
  - Elementary Thread pools in asynch
- Ease of programming
  - Writing correct concurrent code is hard
  - Lots of concurrency in modern HW
  - Portability with the same natural syntax
  - Not achievable before
  - Uncompromising Performance
  - Stable memory model
  - System level interoperability
- C++ shares threads with other languages

# What we are still trying to get

- higher parallel abstractions
  - Transactional memory (TM), executors, latches and barriers, atomic<shared<T>>, queues and counters
  - SIMD, Task Blocks, coroutines, networking
- Distributed and Heterogeneous programming
- Complete Compatibility between C and C++
- Total isolation from programmer mistakes

# What is Parallelization?

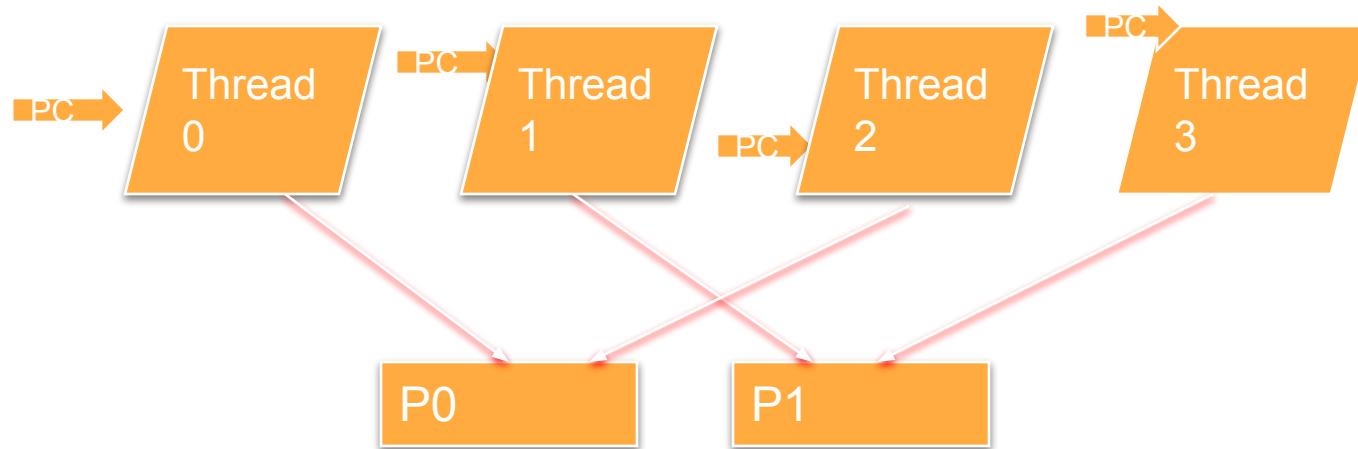


Something is parallel if there is a certain level of independence in the order of operations

- If there is notion of ordering, I must do this, then do that, then it is not parallel

# What is a thread?

- **This** is where the parallelism is.
- What is a thread?
  - A series of instructions with its own program counter and state
- What is a process?
  - An instance of running program, has at least one thread



# Parallel Overhead

- Total CPU > serial CPU
  - Newly introduced parallel portions in your program now needs to be executed, not a big deal
  - Threads need to communicate data to each other and synchronize
    - this is usually the problem, and why parallel code may not scale
  - Most memory models will give you some elegant way to strip out that overhead
  - But it will get worse when increasing number of threads
  - Efficient parallelization is about minimizing the overhead

# Fork-Join Pattern

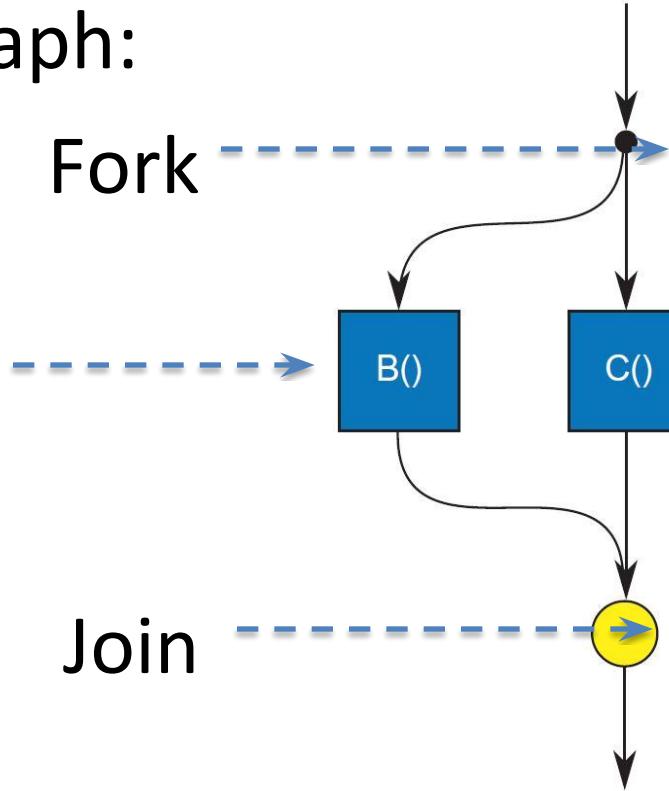
- Control flow **divides** (forks) into multiple flows, then **combines** (joins) later
- During a fork, one flow of control becomes two
- Separate flows are “independent”
  - Does “independent” mean “not dependent” ?
  - No, it just means that the 2 flows of control “are not constrained to do similar computation”
- During a join, two flows become one, and only this one flow continues

# Fork-Join Pattern

- Fork-Join directed graph:

Independent work

Is it possible for B() and C() to have dependencies between them?



# Fork-Join Pattern

- Typical **divide-and-conquer** algorithm implemented with fork-join:

```
void DivideAndConquer( Problem  $P$  ) {  
    if(  $P$  is base case ) {  
        Solve  $P$ ;  
    } else {  
        Divide  $P$  into  $K$  subproblems;  
        Fork to conquer each subproblem in parallel;  
        Join;  
        Combine subsolutions into final solution;  
    }  
}
```

# Thread launching

- Basic thread class
  - a. Fork a function execution, join operation, fork-join pattern
  - b. std::thread takes any “callable object” and runs it asynchronously:
- Three different ways of launching a thread
  1. Ordinary function f in t1
  2. Class w with operator()() in t2
  3. Lambda expression in t3

```
void f();  
class W {  
    void operator()()const;  
    void normalize(long double, int,  
std::vector<float>);  
};  
void bar()  
{  
    std::thread t1(f); //f() executes in separate thread  
    W w;  
    t1.join(); //wait for thread t1 to end  
    std::thread t2(w); // run function object  
    w.operator()() asynchronously  
    std::thread t3([]{std::cout << "lambda\n"; });  
}
```

## cppreference.com

Create account

Search



Page Discussion

View Edit History

C++ Thread support library std::thread

## std::thread

Defined in header &lt;thread&gt;

class thread; (since C++11)

The class `thread` represents a single thread of execution. Threads allow multiple functions to execute concurrently.

Threads begin execution immediately upon construction of the associated thread object (pending any OS scheduling delays), starting at the top-level function provided as a constructor argument. The return value of the top-level function is ignored and if it terminates by throwing an exception, `std::terminate` is called. The top-level function may communicate its return value or an exception to the caller via `std::promise` or by modifying shared variables (which may require synchronization, see `std::mutex` and `std::atomic`)

`std::thread` objects may also be in the state that does not represent any thread (after default construction, move from, `detach`, or `join`), and a thread of execution may be not associated with any `thread` objects (after `detach`).

No two `std::thread` objects may represent the same thread of execution; `std::thread` is not `CopyConstructible` or `CopyAssignable`, although it is `MoveConstructible` and `MoveAssignable`.

### Member types

Member type	Definition
<code>native_handle_type</code>	<i>implementation-defined</i>

### Member classes

<code>id</code>	represents the <i>id</i> of a thread (public member class)
-----------------	---

### Member functions

<code>[constructor]</code>	constructs new thread object (public member function)
<code>(destructor)</code>	destroys the thread object, underlying thread must be joined or detached (public member function)
<code>operator=</code>	moves the thread object (public member function)

### Observers

<code>joinable</code>	checks whether the thread is joinable, i.e. potentially running in parallel context (public member function)
<code>get_id</code>	returns the <i>id</i> of the thread (public member function)

Page

Discussion

View

Edit

History

C++ / Thread support library / std::thread /



Take the stress out of  
job searching. Let  
companies come to  
you on Indeed Prime!

ADS VIA CARBON

## std::thread::thread

<code>thread() noexcept;</code>	(1) (since C++11)
<code>thread( thread&amp;&amp; other ) noexcept;</code>	(2) (since C++11)
<code>template&lt; class Function, class... Args &gt;</code> <code>explicit thread( Function&amp; f, Args&amp;... args );</code>	(3) (since C++11)
<code>thread(const thread&amp;) = delete;</code>	(4) (since C++11)

Constructs new thread object.

- 1) Creates new thread object which does not represent a thread.
- 2) Move constructor. Constructs the thread object to represent the thread of execution that was represented by other. After this call other no longer represents a thread of execution.
- 3) Creates new std::thread object and associates it with a thread of execution. The new thread of execution starts executing

```
std::invoke(decay_copy(std::forward<Function>(f)), decay_copy(std::forward<Args>(args))...);
```

where `decay_copy` is defined as

```
template <class T>
std::decay_t<T> decay_copy(T&& v) { return std::forward<T>(v); }
```

Except that the calls to `decay_copy` are evaluated in the context of the caller, so that any exceptions thrown during evaluation and copying/moving of the arguments are thrown in the current thread, without starting the new thread.

The completion of the invocation of the constructor *synchronizes-with* (as defined in `std::memory_order`) the beginning of the invocation of the copy of `f` on the new thread of execution.

This constructor does not participate in overload resolution if `std::decay_t<Function>` is the same type as `std::thread`. (since C++14)

- 4) The copy constructor is deleted; threads are not copyable. No two `std::thread` objects may represent the same thread of execution.

## Parameters

# Thread joining

- An initialized thread object represents an active thread of execution, and can be joined, or detached
- Once called, the thread object is non-joinable
  - Calling either join or detach on non-joinable object will result in runtime exception
  - If you have not called detach or join for joinable threads, then upon calling destructor, it will call std::terminate
    - This program is not well defined
- **Guideline: must always call join or detach**

To join or not to join, that is the question

thread\_joinable.cpp

# Joining

- Sets a synchronization point between child thread and the parent thread
- Blocks execution of the thread that calls join function, until child thread's execution finish
- After call, the child thread' object can safety be destroyed

# Detach

- Separates the child thread from the parent thread
- Allows execution to continue independently
- Allocated resources will be freed once the child thread exits
- No synchronization with main thread
  - Child thread can outlive parent thread
  - Parent thread cannot hold any references to child thread

# Joining thread

thread\_join

# Detach thread

thread\_detach

# Need a Better joining mechanism

- For join, introduces a synchronization point, and blocks
- We would like to do some other tasks in between waiting for the child thread
- In real world, you will do many tasks between spinning off child threads

# What happens if I have an exception?

## thread\_exception

# A more reliable join: first try

thread\_exception

# RAII

- Resource acquisition is initialization
- When we execution a program, objects will be constructed from top to bottom (in execution order), then destructed in reverse order
  - First created object will be destroyed last

# Thread Guard

Thread\_guard (make sure other operation is caught)

# Passing parameters by values

thread\_param\_by\_val

# Pass parameters by reference

thread\_param\_by\_ref

# Guideline

- passing by references with detach is dangerous

# Data Life time in multithreaded mode

- Data can change during async calls

```
int x, y, z;  
Widget *pw;  
  
...
```

*call `f(x, y)` asynchronously (i.e., on a new thread);*

- *As `f` execute*
  - *x,y,z, pw could be out of scope*
  - *pw could be deleted*
  - *Values might change*
- *Exact details will depend on parameters (unlike single threaded mode)*

# Life time and parameters

```
void f(int xParam, int yParam); // pass by value:  
    // f unaffected by  
    // changes to x, y  
void f(int& xParam, int& yParam); // pass by ref:  
void f(const int& xParam, const int& yParam); // f affected by  
                                                // changes to x, y
```

int x, y, z;

Widget \*pw;

...

*call f(x, y) asynchronously;*

- No declaration insulates f from changes to z, pw, and \*pw.

# Data Lifetime

- Data lifetime issues critical in multi-threading (MT) design.
  - A special aspect of synchronization/race issues.
  - Even shared immutable data subject to lifetime issues.

# When thread2 outlives thread1

## thread\_lifetime

# Guideline for std::thread arguments

- By-reference/by-pointer parameters in async calls always risky.
  - Prefer pass-by-value.
  - Including via lambdas!
- 2 strategies to avoid lifetime issues
  - Copy data for use by the asynchronous call.
  - Ensure referenced objects live long enough.

```
void f(int xParam); // function to call asynchronously
{
    int x;
    ...
    std::thread t1([&]{ f(x); }); // risky! closure holds a ref to x
    std::thread t2([=]{ f(x); }); // okay, closure holds a copy of x
    ...
} // x destroyed
```

# Copying arguments for Async Calls

- std::thread's variadic constructor (conceptually) copies everything:

```
void f(int xVal, const Widget& wVal);
```

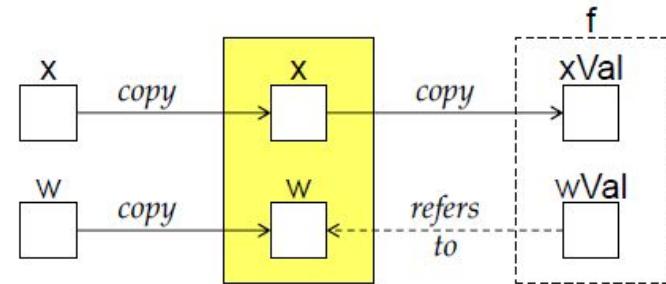
```
int x;
```

```
Widget w;
```

```
...
```

```
std::thread t(f, x, w); // invoke copy of f on co
```

- Copies of f, x, w, guaranteed to exist until asynch call returns.
- Inside f, wVal refers to a *copy of w, not w itself.*
- Copying optimized to moving whenever possible.



# Copying Arguments

- Using by-value captures in closures works, too:

```
void f(int xVal, const Widget& wVal);
```

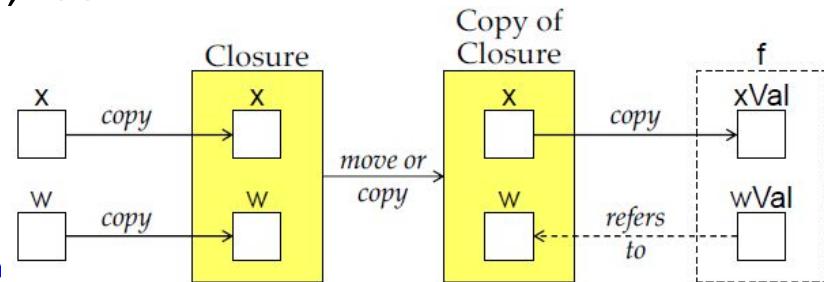
```
int x;
```

```
Widget w;
```

```
...
```

```
std::thread t([=]{ f(x, w); }); // invoke copy of f
```

- Closure contains copies of x and w.
- Closure copied by thread ctor; copy exists until f returns.
  - Copying optimized to moving whenever possible.
- Inside f, wVal refers to a *copy of w, not w itself*.



# Copying Arguments with Bind: another way

- Another approach is based on std::bind:

```
void f(int xVal, const Widget& wVal);
```

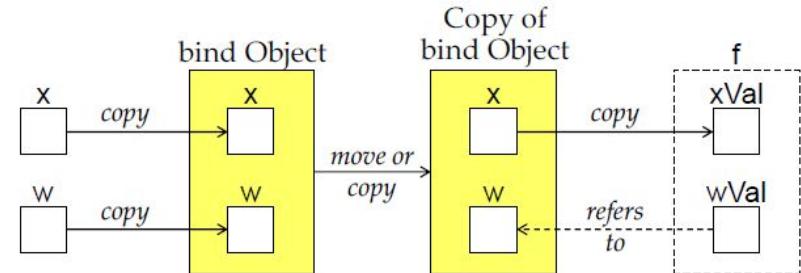
```
int x;
```

```
Widget w;
```

```
...
```

```
std::thread t(std::bind(f, x, w)); // invoke f with copies of x, w, ...
```

- Object returned by bind contains copies of x and w.
- That object copied by thread ctor; copy exists until f returns.
- Inside f, wVal refers to a *copy of w, not w itself.*
- Lambdas are usually a better choice than bind.
- Easier for readers to understand.
- More efficient.



# Guideline for Copying Arguments

- Options for creating argument copies with sufficient lifetimes:
  - Use variadic thread constructor.
  - Use lambda with by-value capture.
  - Use bind.
- **Prefer** to:
  - Use lambda, more elegant, and copying is explicit

# Ensure References live long enough

- One way is to delay locals' destruction until asynch call is complete:

```
void f(int xVal, const Widget& wVal); // as before
{
    int x;
    Widget w;
    ...
    std::thread t([&]{ f(x, w); });
    // wVal really refers to w
    ...
    t.join(); // destroy w only after t finishes
}
```

# What if you really want to pass by ref or need to mix ? Method 1

- Use lambdas:
- Given

```
void f(int xVal, int yVal, int zVal, Widget& wVal);
```

- **Lambdas: use by-reference capture:**

```
{  
    Widget w;  
    int x, y, z;  
    ...  
    std::thread t([=, &w]{ f(x, y, z, w); }); // pass copies of x, y, z;  
    ... // pass w by reference  
}
```

- You're responsible for avoiding data races on w.

# What if you really want to pass by ref or need to mix ? Method 2 and 3

- Use **Variadic thread constructor or bind**: Use C++11's std::ref:
  - Creates objects that act like references.
    - Copies of a std::ref-generated object refer to the same object.

```
void f(int xVal, int yVal, int zVal, Widget& wVal); // as before
{
    static Widget w;
    int x, y, z;
    ...
    std::thread t1(f, x, y, z, std::ref(w)); // pass copies of
    std::thread t2(std::bind(f, x, y, z, std::ref(w))); // x, y, z; pass w
    ... // by reference
}
```

# To move or not to move

thread\_move

# Guideline

- Don't move one thread into another without calling thread constructors or thread assignment operator especially if the destination thread already has a thread

# What is your id?

thread\_id

# Use these tools to do parallel reduction

- Divide and conquer: divide our data set into several blocks, and use multiple threads to accumulate the values in each block
- Reduction: take the sum from each thread and accumulate that, leading to a reduction
- Main problems of finding how many threads to run
  - Oversubscription: when we run too many threads, then the processor have, each thread will have a switch context which harms performance
  - Overload: if we run small amount of data in each thread and we spawn millions of threads leading to excessive overhead (work is smaller then the cost of new threads), then that will kill performance

# How many threads should we run

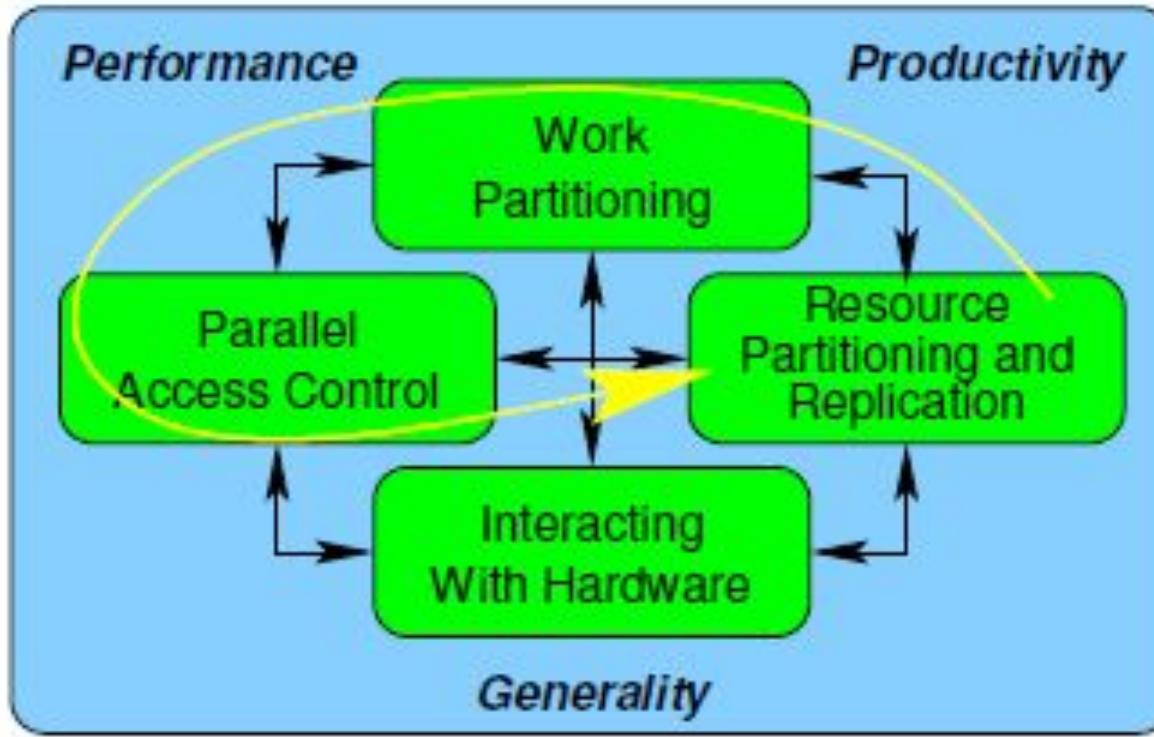
- We need to set minimum number of elements to process on a particular thread
- Find maximum number of threads to avoid overload
- Next find maximum number of threads that processors can run parallel using hardware\_concurrency
- Now select minimum of these 2 values as # of running threads

# Example

Want to process 6000 elements

1. Set minimum number elements to run per thread as 600, so we need 10 threads to avoid overload
2. Ask `hardware_concurrency` and it returns 12, this is the number of threads you can run in parallel
3. This means we should run  $\min(10,12)$  and run 10 threads

# Composite Capabilities



# Key takeaways

When launching thread, you **must always call join or detach**

Passing argument by references with detach is dangerous

Don't move one thread into another without calling thread constructors or thread assignment operator especially if the destination thread already has a thread

You can build higher abstractions with threads, but wait you can do better!



# Exercise 2:

## Implementing Parallel Algorithms

- Implement the parallel variant of transform
- Implement the parallel variant of reduce
- Implement the parallel variant of transform\_reduce
- Evaluate the performance of the algorithms

Exercise document: <https://goo.gl/o6EzdQ>

# Exercise 2:

## Implementing Parallel Algorithms

```
1. template <class ForwardIt1, class ForwardIt2, class UnaryOperation>
2. ForwardIt2 transform(parallel_execution_policy seq, ForwardIt1 first, ForwardIt1 last,
3.                      ForwardIt2 d_first, UnaryOperation unary_op) {
4.
5.     /* implement me */
6.
7. }
```

[https://github.com/AerialMantis/cppcon2018-parallelism-class/blob/master/include/bits/parallel\\_transform.h](https://github.com/AerialMantis/cppcon2018-parallelism-class/blob/master/include/bits/parallel_transform.h)

# Exercise 2:

## Implementing Parallel Algorithms

```
1. template <class ForwardIt, class T, class BinaryOperation, class UnaryOperation>
2. T reduce(parallel_execution_policy seq, ForwardIt first, ForwardIt last,
3.           T init, BinaryOperation binary_op) {
4.
5.     /* implement me */
6.
7. }
```

[https://github.com/AerialMantis/cppcon2018-parallelism-class/blob/master/include/bits/parallel\\_reduce.h](https://github.com/AerialMantis/cppcon2018-parallelism-class/blob/master/include/bits/parallel_reduce.h)

# Exercise 2:

## Implementing Parallel Algorithms

```
1. template <class ForwardIt, class T, class BinaryOperation>
2. T transform_reduce(parallel_execution_policy seq, ForwardIt first, ForwardIt last,
3.                     T init, BinaryOperation binary_op, UnaryOperation unary_op) {
4.
5.     /* implement me */
6.
7. }
```

[https://github.com/AerialMantis/cppcon2018-parallelism-class/blob/master/include/bits/parallel\\_transform\\_reduce.h](https://github.com/AerialMantis/cppcon2018-parallelism-class/blob/master/include/bits/parallel_transform_reduce.h)



# Chapter 6: Synchronization and advanced abstraction

Prerequisites:

1. Previous chapters
2. Resource Acquisition is initialization
3. Threads
4. Parallelism vs concurrency

You will learn:

1. locks
2. mutexes
3. RAII: Lock\_guard
4. Deadlocks
5. Unique locks
6. Async
7. Futures
8. Lock based programming
9. Maybe: condition variables and wait
10. Maybe: promises, packaged tasks

# Locks and Invariants

Most common problems in multithreaded programming are due to invariants being broken while updating

Invariants: statements that are always true for a data structure

E.g. for list, size variable always contains number of elements

# Race condition

Whenever there are two concurrent accesses to data, and one of them is a write, and the two accesses are not ordered by a happens before relationship

Practically: in concurrency, anytime when the outcome depends on timing of order of execution of operations of two or more threads

Most likely it will cause a broken invariant

But most of the time, it will work, especially if there are very few threads

# Mutex/Critical region

- Prevent simultaneous update of shared variables
- Can cause race conditions
- Force only one thread at a time through

```
for (i=0; i<n;++i){
```

...

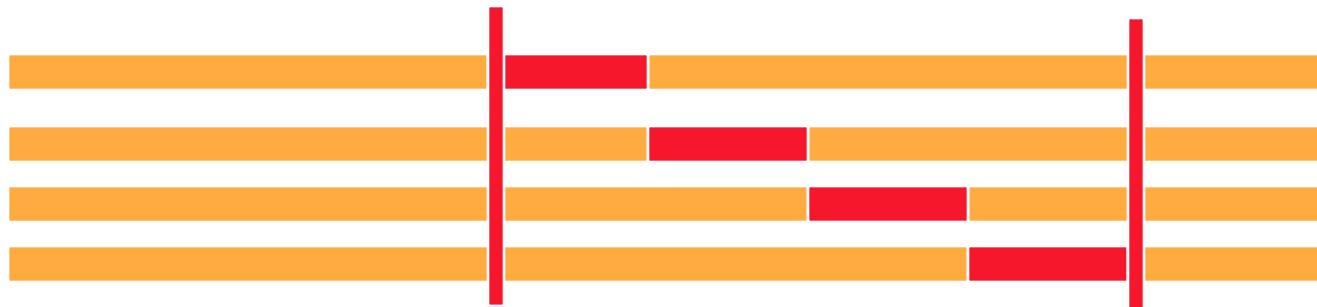
```
    sum+=a[i];
```

...

```
}
```

One at a  
time  
please

If sum is a shared  
variable, this loop can not  
run parallel



# Mutex

Used to protect shared data

Most of STL is not thread safe

Pushing an element on a list:

1. Create a new node
2. Setting the nodes next to current head node
3. Changing head pointer to point to new node

# Mutexes

C++11 has four types:

- `std::mutex`: non-recursive, no timeout support
- `std::timed_mutex`: non-recursive, timeout support
- `std::recursive_mutex`: recursive, no timeout support
- `std::recursive_timed_mutex`: recursive, timeout support

Recursively locking non-recursive mutexes ⇒ undefined behavior.

std::mutex - cppreference.com X Coliru

https://en.cppreference.com/w/cpp/thread/mutex

Create account Search

Page Discussion

C++ Thread support library std::mutex

View Edit History

## std::mutex

Defined in header <mutex>

`class mutex;` (since C++11)

The `mutex` class is a synchronization primitive that can be used to protect shared data from being simultaneously accessed by multiple threads.

`mutex` offers exclusive, non-recursive ownership semantics:

- A calling thread *owns* a `mutex` from the time that it successfully calls either `lock` or `try_lock` until it calls `unlock`.
- When a thread owns a `mutex`, all other threads will block (for calls to `lock`) or receive a `false` return value (for `try_lock`) if they attempt to claim ownership of the `mutex`.
- A calling thread must not own the `mutex` prior to calling `lock` or `try_lock`.

The behavior of a program is undefined if a `mutex` is destroyed while still owned by any threads, or a thread terminates while owning a `mutex`. The `mutex` class satisfies all requirements of [Mutex](#) and [StandardLayoutType](#).

`std::mutex` is neither copyable nor movable.

### Member types

Member type	Definition
<code>native_handle_type</code> (optional)	implementation-defined

### Member functions

(constructor)	constructs the <code>mutex</code> (public member function)
(destructor)	destroys the <code>mutex</code> (public member function)
<code>operator=</code> [deleted]	not copy-assignable (public member function)

### Locking

<code>lock</code>	locks the <code>mutex</code> , blocks if the <code>mutex</code> is not available (public member function)
<code>try_lock</code>	tries to lock the <code>mutex</code> , returns if the <code>mutex</code> is not available (public member function)
<code>unlock</code>	unlocks the <code>mutex</code> (public member function)

### Native handle

<code>native_handle</code>	returns the underlying implementation-defined native handle object (public member function)
----------------------------	--

SendGrid  
Transactional Email Delivery. Start sending for Free with a 5-min Integration.  
ADS VIA CARBON

# Mutual Exclusion

- A mutex is a primitive object used for controlling access in a multi-threaded system.
- A mutex is a shared object (a resource)
- Simplest use:

```
std::mutex m;  
int sh; // shared data  
// ...  
m.lock();  
// manipulate shared data:  
sh+=1;  
m.unlock();
```

# Mutex – try\_lock()

- Don't wait unnecessarily

```
std::mutex m;  
int sh; // shared data  
// ...  
if (m.try_lock()) { // manipulate shared data:  
    sh+=1;  
    m.unlock();  
}  
else {  
    // maybe do something else  
}
```

# Mutex – try\_lock\_for()

- Don't wait for too long:

```
std::timed_mutex m;
int sh; // shared data
// ...
if (m.try_lock_for(std::chrono::seconds(10))) { // Note: time
    // manipulate shared data:
    sh+=1;
    m.unlock();
}
else {
    // we didn't get the mutex; do something else
}
```

# Mutex – try\_lock\_until()

- We can wait until a fixed time in the future:

```
std::timed_mutex m;  
int sh; // shared data  
// ...  
if (m.try_lock_until(midnight)) { // manipulate shared data:  
    sh+=1;  
    m.unlock();  
}  
else {  
    // we didn't get the mutex; do something else  
}
```

# Recursive Mutex

- In some important use cases it is hard to avoid recursion

```
std::recursive_mutex m;  
int sh; // shared data  
// ...  
void f(int i)  
{  
    // ...  
    m.lock();  
    // manipulate shared data:  
    sh+=1;  
    if (--i>0) f(i);  
    m.unlock();  
    // ...  
}
```

# C++14 additional Mutexes

C++14 adds:

- `std::shared_timed_mutex`:
- Non-recursive reader/writer lock w/timeout support.
- Adds `lock_shared/try_lock_shared` to `std::timed_mutex` API.

Based on Boost's `shared_mutex`, but with fewer capabilities:

- Can't upgrade read lock to exclusive lock.

# Locking with mutex

mutex\_lock\_unlock

# Guideline

Always remember to unlock

# RAlI classes for Mutexes: lock\_guard

Mutexes typically managed by RAlI classes:

`std::lock_guard`: lock mutex in ctor, unlock it in dtor.

```
std::mutex m; // mutex object
```

```
{
```

```
std::lock_guard<std::mutex> L(m); // lock m
```

```
... // critical section
```

```
} // unlock m
```

No other operations.

No copying/moving, no assignment, no manual unlock, etc.

Locks `std::shared_timed_mutexes` in exclusive (write) mode.

std::lock\_guard - cppreference.com Coliru

https://en.cppreference.com/w/cpp/thread/lock\_guard

Create account Search

Page Discussion

C++ Thread support library std::lock\_guard

## std::lock\_guard

Defined in header <mutex>

```
template< class Mutex >
class lock_guard;
```

The class `lock_guard` is a mutex wrapper that provides a convenient RAII-style mechanism for owning a mutex for the duration of a scoped block.

When a `lock_guard` object is created, it attempts to take ownership of the mutex it is given. When control leaves the scope in which the `lock_guard` object was created, the `lock_guard` is destructed and the mutex is released.

The `lock_guard` class is non-copyable.

### Template parameters

**Mutex** - the type of the mutex to lock. The type must meet the *BasicLockable* requirements

### Member types

Member type	Definition
<code>mutex_type</code>	<code>Mutex</code>

### Member functions

(constructor)	constructs a <code>lock_guard</code> , optionally locking the given mutex <small>(public member function)</small>
(destructor)	destroys the <code>lock_guard</code> object, unlocks the underlying mutex <small>(public member function)</small>
<code>operator=</code> [deleted]	not copy assignable <small>(public member function)</small>

### Example

Run this code

```
#include <thread>
#include <mutex>
#include <iostream>

int g_i = 0;
std::mutex g_i_mutex; // protects g_i

void safe_increment()
```

Now change lock to RAII lock guard

`mutex_lock_guards`

Returning pointer or reference to unprotected data

Mutex Unprotected Data

Passing code to the protected data structure which you don't have control

Mutex unprotected data access

# RAII classes for Mutex: unique\_lock

std::unique\_lock: much more flexible.

May lock mutex after construction, unlock before destruction.

Moveable, but not copyable.

Supports timed mutex operations:

Try locking, timeouts, etc.

Typically the best choice for unshared timed mutexes.

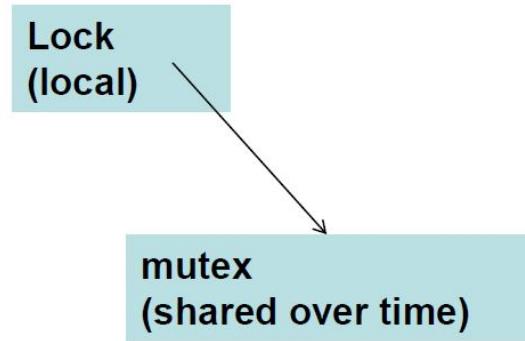
Locks std::shared\_timed\_mutexes in exclusive (write) mode.

```
using RTM = std::recursive_timed_mutex; // typedef  
  
RTM m; // mutex object  
  
{  
  
    std::unique_lock<RTM> L(m); // lock m  
  
    ... // critical section  
  
    L.unlock(); // unlock m  
  
    ...  
  
} // nothing happens
```

# RAII for mutexes: std::lock

- A lock represents local ownership of a non-local resource(the mutex)

```
Std::mutex m;  
Int sh; // shared data  
void f()  
{  
    // ...  
    Std::unique_lock lck(m); // grab (acquire) the mutex  
    // manipulate shared data:  
    sh+=1;  
} // implicitly release the mutex
```



# Guideline for using mutex locks

Villain:

Returning pointer or reference  
to unprotected data

Passing code to the protected  
data structure which you don't  
have control with

Race condition in the interface

Hero:

Standalone application

-e.g. Linux kernel

Parallel library

# std::shared\_lock(C++14)

C++14 adds:

`std::shared_lock`: like `std::unique_lock`, but locks  
`std::shared_timed_mutexes` for shared (read) access.

```
using STM = std::shared_timed_mutex; // typedef
STM m; // mutex object
{
    std::shared_lock<STM> L(m); // lock m in read mode
    ... // critical section
    // for reading
    L.unlock(); // unlock m
    ...
}
```

// nothing happens

# Deadlock

Single mutex with lock\_guard

Multiple mutex

No mutex

# Multiple Mutex acquisition and Transactional Memory

Acquiring mutexes in different orders leads to deadlock:

```
int weight, value;  
  
std::mutex wt_mux, val_mux;  
  
{ // Thread 1  
  
    std::lock_guard<std::mutex> wt_lock(wt_mux); // wt 1st  
  
    std::lock_guard<std::mutex> val_lock(val_mux); // val 2nd  
  
    work with weight and value // critical section  
  
}  
  
{ // Thread 2  
  
    std::lock_guard<std::mutex> val_lock(val_mux); // val 1st  
  
    std::lock_guard<std::mutex> wt_lock(wt_mux); // wt 2nd  
  
    work with weight and value // critical section  
  
}
```

# Lock ordering deadlocks

mutex\_lock\_ordering\_deadlock

# Single mutex with lock\_guard

# Potential Deadlock

- Unstructured use of multiple locks is hazardous:

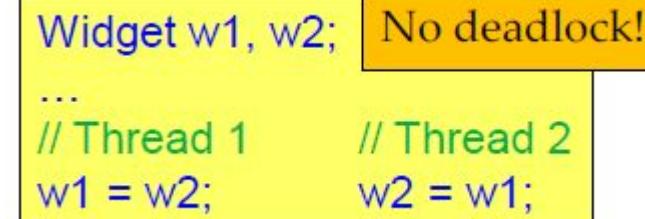
```
std::mutex m1;
std::mutex m2;
int sh1; // shared data
int sh2;
// ...
void f()
{
    // ...
    std::unique_lock lck1(m1);
    std::unique_lock lck2(m2);
    // manipulate shared data:
    sh1+=sh2;
}
```

# Multiple Mutex Acquisition: shared\_lock

Works with std::shared\_lock as well as std::unique\_lock.

E.g., for write access to assignment target, read access to source:

```
class Widget {  
  
public:  
  
Widget& operator=(const Widget& rhs)  
{  
  
if (this != &rhs) {  
  
std::unique_lock<std::shared_timed_mutex> dest(m, std::defer_lock);  
  
std::shared_lock<std::shared_timed_mutex> src(rhs.m, std::defer_lock);  
  
std::lock(dest, src); // lock dest in write mode, src in read mode  
  
... // assign data  
  
} // unlock mutexes  
  
return *this;  
}  
  
private:  
  
mutable std::shared_timed_mutex m;  
  
...  
};
```



# RAII for mutexes: std::lock

- We can safely use several locks

```
void f()
{
    // ...
    std::unique_lock lck1(m1,std::defer_lock); // make locks but don't yet
    //try to acquire the mutexes
    std::unique_lock lck2(m2,std::defer_lock);
    std::unique_lock lck3(m3,std::defer_lock);
    // ...
    lock(lck1,lck2,lck3);
    // manipulate shared data
} // implicitly release the mutexes
```

# How to solve Multiple mutex acquisition

std::lock solves this problem:

```
{ // Thread 1  
    std::unique_lock<std::mutex> wt_lock(wt_mux, std::defer_lock);  
    std::unique_lock<std::mutex> val_lock(val_mux, std::defer_lock);  
    std::lock(wt_lock, val_lock); // get mutexes w/o  
    // deadlock  
    work with weight and value // critical section  
}  
  
{ // Thread 2  
    std::unique_lock<std::mutex> val_lock(val_mux, std::defer_lock);  
    std::unique_lock<std::mutex> wt_lock(wt_mux, std::defer_lock);  
    std::lock(val_lock, wt_lock); // get mutexes w/o  
    // deadlock  
    work with weight and value // critical section  
}
```

# Use unique lock to solve deadlock

Mutex deadlock unique lock

# Multiple mutex

## Locks are Impractical for Generic Programming=callback

Thread 1:  
m1.lock();  
m2.lock();  
... + Thread 2:  
m2.lock();  
m1.lock();  
... = deadlock

Easy. Order Locks.

Now let's get slightly more real:

What about Thread 1 + A thread running f():  
template <class T>  
void f(T &x, T y) {  
 unique\_lock<mutex> \_(m2);  
 x = y;  
}

What locks does  $x = y$  acquire?

## What locks does $x = y$ acquire?

- Depends on the type T of x and y.
  - The author of f() shouldn't need to know.
    - That would violate modularity.
  - But lets say it's shared\_ptr<TT>.
    - Depends on locks acquired by TT's destructor.
    - Which probably depends on its member destructors.
    - Which I definitely shouldn't need to know.
    - But which might include a shared\_ptr<TTT>.
      - Which acquires locks depending on TTT's destructor.
      - Whose internals I definitely have no business knowing.
      - ...- And this was for an unrealistically simple f()!
- We have no realistic rules for avoiding deadlock!

In practice: Test & fix?

```
template <class T>
void f(T &x, T y) {
    unique_lock<mutex> _(m2);
    x = y;
}
```

# Transactions Naturally Fit Generic Programming Model



- Composable, no ordering constraints

```
f() implementation:  
template <class T>  
void f(T &x, T y) {  
    transaction {  
        x = y;  
    }  
}
```

```
Class implementation:  
class ImpT  
{  
    ImpT& operator=(ImpT T&  
rhs)  
    {  
        transaction {  
            // handle assignment  
        }  
    }  
};
```

Impossible to deadlock

# No mutex

Thread 1:

```
t2.join();
```

Thread 2:

```
t1.join();
```

# What can you do with locks and mutexes?

Can implement thread  
safe stack

LIFO data structure

Implement using singly  
linked list

Push and pop from the  
head

Problems:

Race condition in the  
interface

# Condition Variables

Allow threads to communicate about changes to shared data.

- Consumers `wait` until producers `notify` about changed state.

Rules:

- Call `wait` while holding locked mutex.
- `wait` unlocks mutex, blocks thread, enqueues it for notification.
- At notification, thread is unblocked and moved to mutex queue.
- “Notified threads awake and run with the mutex locked.”

Condition variable types:

- `condition_variable`: wait on `std::unique_lock<std::mutex>`.
- Most efficient, appropriate in most cases.
- `condition_variable_any`: wait on any lock type.
- Possibly less efficient, more flexible.
- E.g., works with
  - `std::shared_lock<std::shared_timed_mutex>`

# Condition Variables: Wait

[wait parameters:](#)

- Mutex for shared data (required).
- Timeout (optional).
- Predicate that must be true for thread to continue (optional).
  - Allows library to handle spurious wakeups.
  - Often specified via lambda.

[Notification options:](#)

- [notify\\_one](#) waiting thread.
  - When all waiting threads will do and only one needed.
  - No guarantee that only one will be awakened.
- [notify\\_all](#) waiting threads.

# Wait Examples

```
std::atomic<bool> readyFlag(false);

std::mutex m;

std::condition_variable cv;

{

    std::unique_lock<std::mutex> lock(m);

    while (!readyFlag)                                // loop for spurious wakeups
        cv.wait(lock);                               // wait for notification

    cv.wait(lock, []{ return readyFlag == true; }); // ditto, but library loops

    if (cv.wait_for(lock,                         // if (notification rcv'd
                    std::chrono::seconds(1),           // or timeout) and
                    []{ return readyFlag == true; })) { // predicate's true...
        ...
        // critical section
    }

    else {
        ...
        // timed out w/o getting
    }
}

}
```

# Notification Example

```
std::atomic<bool> readyFlag(false);    // as before  
  
std::condition_variable cv;  
  
{  
...                                // make things “ready”  
  
readyFlag = true;  
  
cv.notify_one();                  // wake ~1 thread  
}  
                                // blocked on cv  
  
{  
...                                // make things “ready”  
  
readyFlag = true;  
  
cv.notify_all();                  // wake all threads  
}  
                                // blocked on cv (all but  
                                // 1 will then block on m)
```

`notify_all` moves all blocked threads from the condition variable queue to the corresponding mutex queue.

# What can you do with condition variable and wait

Build a thread safe queue      Push

FIFO data structure      Pop

Implemented using singly  
linked list      Front  
Empty

Push from tail, and pop  
from head      Size

(same as stack interface)

# Synchronous vs asynchronous

Synchronous operations

Blocks the caller process  
until its operation  
completes

Asynchronous operations

Non-blocking

Initiates the operation, but the  
caller continues, and can discover  
completion in future

Good for when computing long  
running task in a separate thread

# How it works

## Caller thread

1. Asynchronous task caller get a future associated with asynchronous task and creates shared state
2. Dispatch the async task
3. When caller need the result, it calls get method on the future
4. If get method blocks, then async task has not finished yet

## Async thread

1. Async thread initiated
2. When ready, modifies shared state that is linked to std::future
3. Completes execution

std::async - cppreference.com X Coliru X https://en.cppreference.com/w/cpp/thread/async 67% ⌂ ⌃ ⌄ ⌅ ⌆ ⌇

**c++ Thread support library**

**std::async**

Defined in header `<future>`

```
template< class Function, class... Args>
std::future<std::result_of_t<std::decay_t<Function>(std::decay_t<Args>...)>>
    async( Function&& f, Args&&... args );
```

(since C++11)  
(until C++17)

```
template< class Function, class... Args>
std::future<std::invoke_result_t<std::decay_t<Function>, std::decay_t<Args>...>>
    async( Function&& f, Args&&... args );
```

(since C++17)  
(until C++20)

```
template< class Function, class... Args>
[[nodiscard]] std::future<std::invoke_result_t<std::decay_t<Function>, std::decay_t<Args>...>>
    async( Function&& f, Args&&... args );
```

(since C++20)

```
template< class Function, class... Args >
std::future<std::result_of_t<std::decay_t<Function>(std::decay_t<Args>...)>>
    async( std::launch::policy, Function&& f, Args&&... args );
```

(since C++11)  
(until C++17)

```
template< class Function, class... Args >
std::future<std::invoke_result_t<std::decay_t<Function>, std::decay_t<Args>...>>
    async( std::launch::policy, Function&& f, Args&&... args );
```

(since C++17)  
(until C++20)

```
template< class Function, class... Args >
[[nodiscard]] std::future<std::invoke_result_t<std::decay_t<Function>, std::decay_t<Args>...>>
    async( std::launch::policy, Function&& f, Args&&... args );
```

(since C++20)

The template function `async` runs the function `f` asynchronously (potentially in a separate thread which may be part of a thread pool) and returns a `std::future` that will eventually hold the result of that function call.

- Behaves as if (2) is called with `policy` being `std::launch::async | std::launch::deferred`. In other words, `f` may be executed in another thread or it may be run synchronously when the resulting `std::future` is queried for a value.
- Calls a function `f` with arguments `args` according to a specific launch policy:
  - If the `async` flag is set (i.e. `[policy & std::launch::async] != 0`), then `async` executes the callable object `f` on a new thread of execution (with all thread-locals initialized) as if spawned by `std::thread(std::forward<f>(f), std::forward<Args>(args)...)`, except that if the function `f` returns a value or throws an exception, it is stored in the shared state accessible through the `std::future` that `async` returns to the caller.
  - If the `deferred` flag is set (i.e. `[policy & std::launch::deferred] != 0`), then `async` converts `f` and `args...` to a copyable way as by `std::move` (i.e. `f` does not get a new thread of execution). Instead, `lazy_evaluation` is performed: the first call to a nonempty member function on the `std::future` that `async` returns to the caller will cause the copy of `f` to be invoked (as an `eval`) with the copies of `args...` (also passed as `evalvalues`) in the current thread (which does not have to be the thread that originally called `std::async`). The result or exception is placed in the shared state associated with the `future` and only then it is made ready. All further accesses to the same `std::future` will return the result immediately.
  - If both the `std::launch::async` and `std::launch::deferred` flags are set in `policy`, it is up to the implementation whether to perform asynchronous execution or lazy evaluation.

If neither `std::launch::async` nor `std::launch::deferred`, nor any implementation-defined policy flag is set in `policy`, the behavior is undefined.

In any case, the call to `std::async` synchronizes-with (as defined in `std::memory_order`) the call to `f`, and the completion of `f` is sequenced-before making the shared state ready. If the `async` policy is chosen, the associated thread completion synchronizes-with the successful return from the first function that is waiting on the shared state, or with the return of the last function that releases the shared state, whichever comes first.

### Parameters

`f` - *Callable* object to call  
`args...` - parameters to pass to `f`  
`policy` - bitmask value, where individual bits control the allowed methods of execution

Bit	Explanation
<code>std::launch::async</code>	enable asynchronous evaluation
<code>std::launch::deferred</code>	enable lazy evaluation

Type requirements

# Parallel reduction with async async

# Basic thread replacement but adds exception and value

Building blocks:

- `std::async`: Request asynchronous execution of a function.
- `Future`: token representing function's result.

Unlike raw use of `std::thread` objects:

- Allows values or exceptions to be returned.
- Just like “normal” function calls.

# async

```
double bestValue(int x, int y);           // something callable  
std::future<double> f =                 // in concept, run λ  
    std::async( []{ return bestValue(10, 20); } ); // asynchronously;  
                                                // get future for it  
...                                              // do other work  
double val = f.get();                     // get result (or  
                                                // exception) from λ
```

As usual, auto reduces verbiage:

```
auto f = std::async( []{ return bestValue(10, 20); } );  
...  
auto val = f.get();
```

# Async Launch Policies

`std::launch::async`: function runs on a new thread.

- Maintains calling thread's responsiveness (e.g., GUI threads).

```
auto f = std::async(std::launch::async, doBackgroundWork);
```

`std::launch::deferred`: function runs on thread invoking `get` or

- wait on `std::async`'s future:

```
auto f = std::async(std::launch::deferred,
```

```
[]{ return bestValue(10, 20); });
```

...

```
auto val = f.get(); // run  $\lambda$  synchronously here
```

- Useful for debugging, performance tuning.

By default, implementation chooses, presumably with goals:

- Take advantage of all hardware concurrency, i.e., scale.
- Avoid oversubscription.

# Futures

Two kinds:

- `std::future<T>`: result may be accessed only once.
  - Suitable for most use cases.
  - Moveable, not copyable.
  - Exactly one future has right to access result.
- `std::shared_future<T>`: result may be accessed multiple times.
  - Appropriate when multiple threads access a single result.
  - Both copyable and moveable.
  - Multiple `std::shared_futures` for the same result may exist.
  - Creatable from `std::future`.
  - Such creation transfers ownership.

# Futures results

Result retrieval via get:

- Blocks until a return is available, then grabs it.
- For `future<T>`, “grabs”  $\equiv$  “moves (if possible) or copies.”
- For `shared_future<T>` or `anyKindOfFuture<T&>`, “grabs”  $\equiv$  “gets reference to.”
- “Return” may be an exception (which is then propagated).

# Futures Alternatives

An alternative is `wait`:

Blocks until a return is available.

```
std::future<double> f = std::async([]{ return bestValue(10, 20); });
```

...

```
f.wait(); // block until λ is done
```

A timeout may be specified.

Most useful when `std::launch::async` specified.

```
std::future<double> f =  
std::async(std::launch::async, []{ return bestValue(10, 20); });  
  
...  
  
while ( f.wait_for(std::chrono::seconds(0)) != // if result of λ  
std::future_status::ready) { // isn't ready,  
... // do more work  
}  
  
double val = f.get(); // grab result
```

# When you don't care about the return from future

Useful when callers want to know only when a callee finishes.

Callable objects returning void.

Callers uninterested in return value.

But possibly interested in exceptions.

```
void initDataStructs(int defValue);  
void initGUI();  
  
std::future<void> f1 = std::async( []{ initDataStructs(-1); } );  
std::future<void> f2 = std::async( []{ initGUI(); } );  
  
... // init everything else  
  
f1.get(); // wait for asynch. inits. to  
f2.get(); // finish (and get exceptions,  
// if any)  
... // proceed with the program
```

# Packaged tasks

Wraps any callable target so it can be invoked asynchronously

Return value or exception is stored in a shared state

Unlike `async`, packaged task is not called automatically after construction, but has to be called explicitly.

To run synchronously, do nothing and it will be executed sequentially

To run asynchronously, have to detach the task

Have to specify template parameter in a special way

```
template< class R, class ...Args >class packaged_task<R(Args...)>;
```

1. packaged\_task<int(int,int)> task(callable objectf)
2. packaged\_task<int(int,int)> task([](int a, int b) { return std::pow(a, b); })
3. packaged\_task<int()> task(std::bind(f, 2, 11))

std::packaged\_task - cppreference x Coliru https://en.cppreference.com/w/cpp/thread/packaged\_task

Create account Search

Page Discussion View Edit History

C++ Thread support library std::packaged\_task

## std::packaged\_task

Defined in header `<future>`

`template< class > class packaged_task; //not defined (1) (since C++11)`

`template< class R, class ...Args > class packaged_task<R(Args...)>; (2) (since C++11)`

The class template `std::packaged_task` wraps any *Callable* target (function, lambda expression, bind expression, or another function object) so that it can be invoked asynchronously. Its return value or exception thrown is stored in a shared state which can be accessed through `std::future` objects.

Just like `std::function`, `std::packaged_task` is a polymorphic, allocator-aware container: the stored callable target may be allocated on heap or with a provided allocator. (until C++17)

### Member functions

(constructor)	constructs the task object (public member function)
(destructor)	destructs the task object (public member function)
<code>operator=</code>	moves the task object (public member function)
<code>valid</code>	checks if the task object has a valid function (public member function)
<code>swap</code>	swaps two task objects (public member function)

### Getting the result

<code>get_future</code>	returns a <code>std::future</code> associated with the promised result (public member function)
-------------------------	--

### Execution

<code>operator()</code>	executes the function (public member function)
<code>make_ready_at_thread_exit</code>	executes the function ensuring that the result is ready only once the current thread exits (public member function)
<code>reset</code>	resets the state abandoning any stored results of previous executions (public member function)

### Non-member functions

<code>std::swap</code> ( <code>std::packaged_task</code> ) (C++11) (function template)	specializes the <code>std::swap</code> algorithm
---	--

All the tools your team needs in one place.  
Slack: Where work happens.  
ADS VIA CARBON

slack

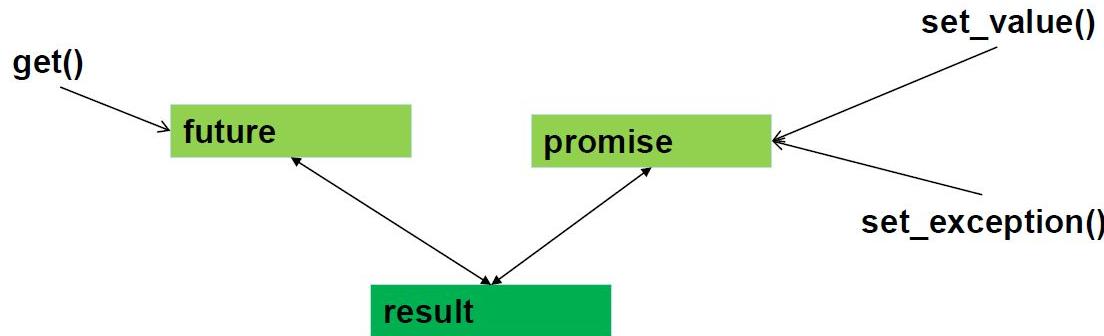
# Packaged task

packaged\_task

# promises

- Promise is paired with a future
- A thread with access to the future object can wait for the result to be set while another thread that has access to the associated promise object can set the value to store it and make the future ready

# Future and Promise



- `future+promise` provides a simple way of passing a value from one thread to another
  - No explicit synchronization
  - Exceptions can be transmitted between threads

# Future and Promise

- Get from a **future<X>called f:**  
**X v = f.get();// if necessary wait for the value to get**
- Put to a **promise<X>called p(attached to f):**

```
try {  
    X res;  
    // compute a value for res  
    p.set_value(res);  
}  
catch (...) {  
    // oops: couldn't compute res  
    p.set_exception(std::current_exception());  
}
```

https://en.cppreference.com/w/cpp/thread/promise

Create account Search

Page Discussion

C++ Thread support library std::promise

## std::promise

Defined in header <future>

```
template< class R > class promise;           (1) (since C++11)
template< class R > class promise<R&>;      (2) (since C++11)
template<>     class promise<void>;          (3) (since C++11)
```

1) base template  
2) non-void specialization, used to communicate objects between threads  
3) void specialization, used to communicate stateless events

The class template std::promise provides a facility to store a value or an exception that is later acquired asynchronously via a std::future object created by the std::promise object. Note that the std::promise object is meant to be used only once.

Each promise is associated with a *shared state*, which contains some state information and a *result* which may be not yet evaluated, evaluated to a value (possibly void) or evaluated to an exception. A promise may do three things with the shared state:

- *make ready*: the promise stores the result or the exception in the shared state. Marks the state ready and unblocks any thread waiting on a future associated with the shared state.
- *release*: the promise gives up its reference to the shared state. If this was the last such reference, the shared state is destroyed. Unless this was a shared state created by std::async which is not yet ready, this operation does not block.
- *abandon*: the promise stores the exception of type std::future\_error with error code std::future\_errc::broken\_promised, makes the shared state *ready*, and then *releases* it.

The promise is the "push" end of the promise-future communication channel: the operation that stores a value in the shared state *synchronizes-with* (as defined in std::memory\_order) the successful return from any function that is waiting on the shared state (such as std::future::get). Concurrent access to the same shared state may conflict otherwise: for example multiple callers of std::shared\_future::get must either all be read-only or provide external synchronization.

### Member functions

(constructor)	constructs the promise object (public member function)
(destructor)	destroys the promise object (public member function)
<b>operator=</b>	assigns the shared state (public member function)
<b>swap</b>	swaps two promise objects (public member function)

Getting the result

All the tools your team needs in one place.  
Slack: Where work happens.  
ADS VIA CARBON

slack

# Promises

promises

# Promises and exception

```
1. #include <iostream>
2. #include <thread>
3. #include <future>
4. #include <stdexcept>
5. void throw_exception(){
6.     throw std::invalid_argument("input cannot be
negative");
7. }
8. void calculate_square_root(std::promise<int>& prom){
9.     int x=1;
10.    std::cout << "Please, enter an integer value: ";
11.    try {
12.        std::cin >> x;
13.        if(x < 0)
14.            {
15.                throw_exception();
16.            }
17.        prom.set_value(std::sqrt(x));
18.    }
19.    catch (std::exception&)
20.    {
21.        prom.set_exception(std::current_exception());
22.    }
}
```

```
1. void print_result(std::future<int>& fut) {
2.     try {
3.         int x = fut.get();
4.         std::cout << "value: " << x << '\n';
5.     }
6.     catch (std::exception& e) {
7.         std::cout << "[exception caught: " << e.what()
8.             << "]\n";
9.     }
10.    int main(){
11.        std::promise<int> prom;
12.        std::future<int> fut = prom.get_future();
13.        std::thread printing_thread(print_result, std::ref(fut));
14.        std::thread calculation_thread(calculate_square_root,
15.                                         std::ref(prom));
16.        printing_thread.join();
17.        calculation_thread.join();
18.        return 0;
19.    }
```

# Key takeaways

Supports Lock-based data structures

Always Maintain invariants

No race conditions and deadlocks

Handle exceptions



# Chapter 7: Atomic and Memory Model

## Prerequisites:

1. Previous chapters
2. Threads
3. Locked based programming
4. Deadlocks
5. Race conditions

## You will learn:

1. What is a memory model
2. Why do we need it
3. What atomics facilities are provided
4. When do you need it

# Memory Model and Consistency model, a quick tutorial

- Sequential Consistency (SC)

**Sequential consistency was originally defined in 1979 by Leslie Lamport as follows:**

- “... the result of any execution is the same as if the reads and writes occurred in some order, and the operations of each individual processor appear in this sequence in the order specified by its program”
- But chip/compiler designers can be annoyingly helpful:
- It can be more expensive to do exactly what you wrote.
- Often they'd rather do something else, that could run faster.

# Sequential Consistency: a tutorial

- The semantics of the **single threaded** program is defined by the program order of the statements. This is the strict sequential order. For example:

x = 1;

r1 = z;

y = 1;

r2 = w;

# Sequential Consistency for program understanding

- Suppose we have two threads.  
Thread 1 is the sequence of statement above. Thread 2 is:

Thread 1:      Thread2:

x = 1;      w=1;

r1 = z;      r3=y;

y = 1;      z=1;

r2 = w;      r4=x;

- 2 of 4! Possible interleavings:

x = 1;      x=1;

w = 1;      w=1;

r1 = z;      r3=y;

r3 = y;      z=1;

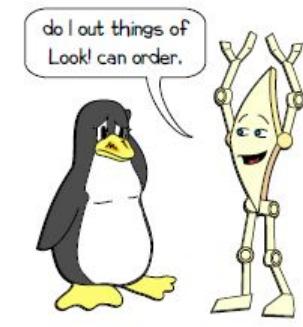
y = 1;      r4=x;

z = 1;      r1=z;

r2 = w;      y=1;

r4 = x;      r2=w;

(All variables are initialized to zero.)



# Now add fences to control reordering

Thread 1:

$x = 1;$

$r1 = z;$

`fence();`

$y = 1;$

$r2 = w;$

Is  $r3==1$  and  $r4==1$  possible?

Is  $r1==1$  and  $r2==1$  possible?

Thread2:

$w=1;$

$r3=y;$

`fence();`

$z=1;$

$r4=x;$



# Memory Model and instruction reordering

- Definitions:
- Instruction reordering: When a program executes instructions, especially memory reads and writes, in an order that is different than the order specified in the program's source code.
- Memory model: Describes how memory reads and writes may appear to be executed relative to their program order.
- Affects the valid optimizations that can be performed by compilers, physical processors, and caches.

# What is a memory model?

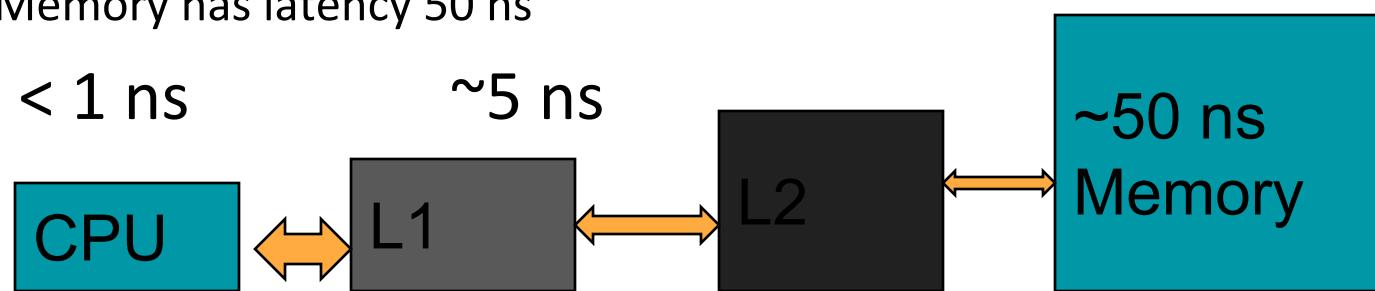
- A set of rules that describe allowable semantics for memory accesses on a computer program
  - Defines the expected value or values returned by any loads in the program
  - Determines when a program produces undefined behavior (e.g load from uninitialized variables)
- Stroustrup: represents a contract between the implementers and the programmers to ensure that most programmers do not have to think about the details of modern computer hardware
- critical component of concurrent programming

# Thread switching and you

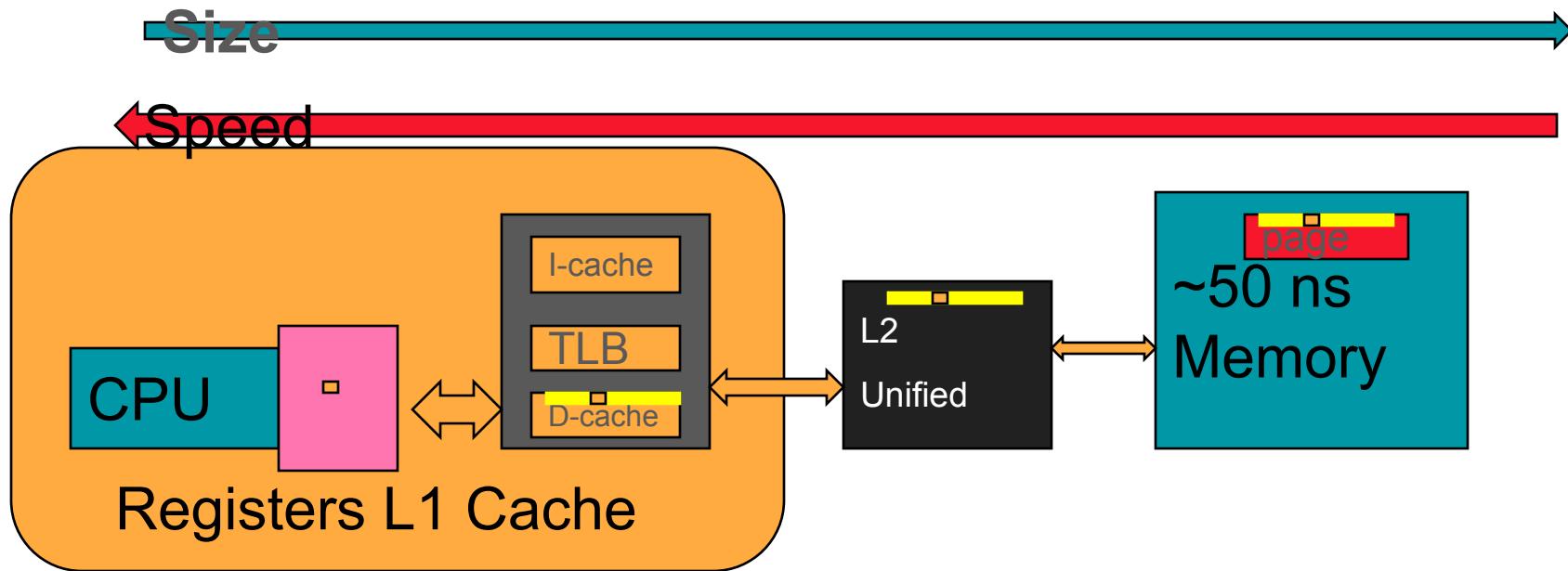
- Execution of a sequential program
- Software, not hardware
- A processor can run a thread
- Put it aside
  - Thread does I/O
  - Thread runs out of time
- Run another thread
- You work in an office
- When you leave for lunch, someone else takes over your office.
- If you don't take a break, a security guard shows up and escorts you to the cafeteria.
- When you return, you may get a different office

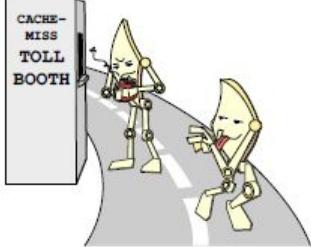
# Memory vs CPU

- Memory latency is crucial to getting right performance, and affects parallelization
- Knowing the memory characteristics will help to write faster code, design language
- 1 GHz means one result every nanosecond
- Memory has latency 50 ns

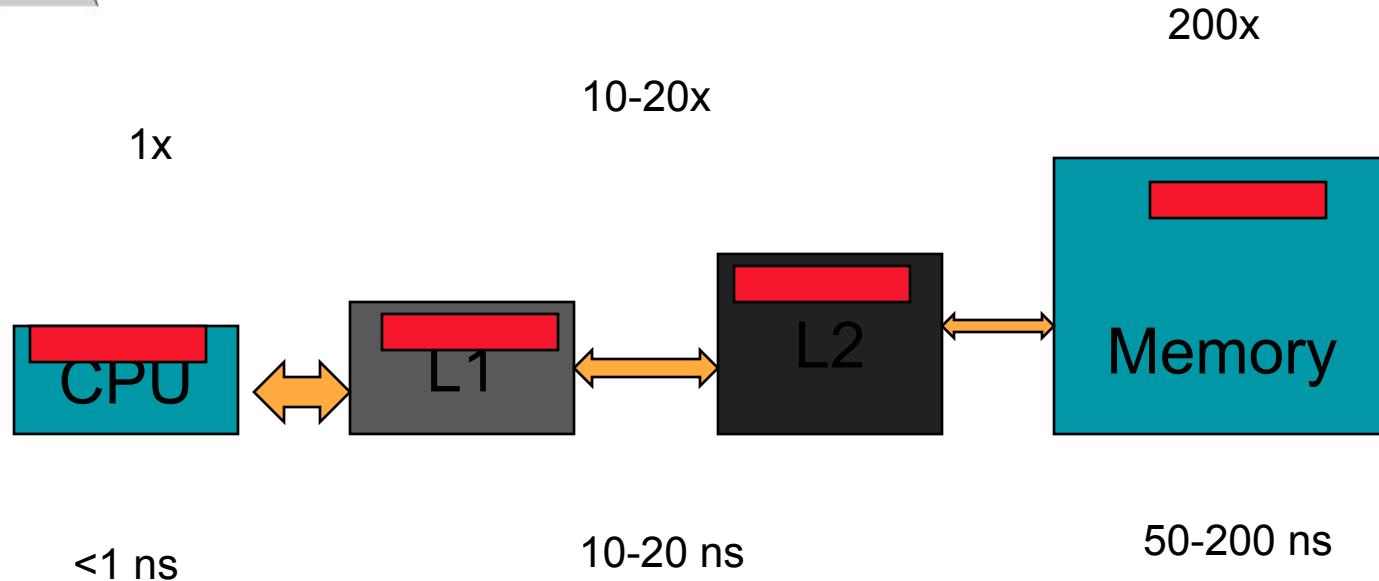


# The memory hierarchy

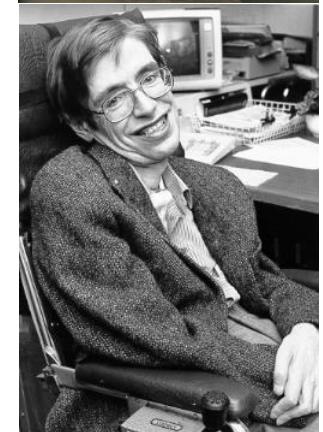
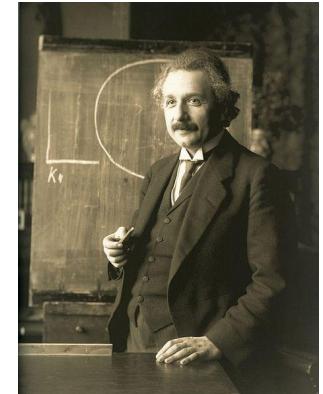
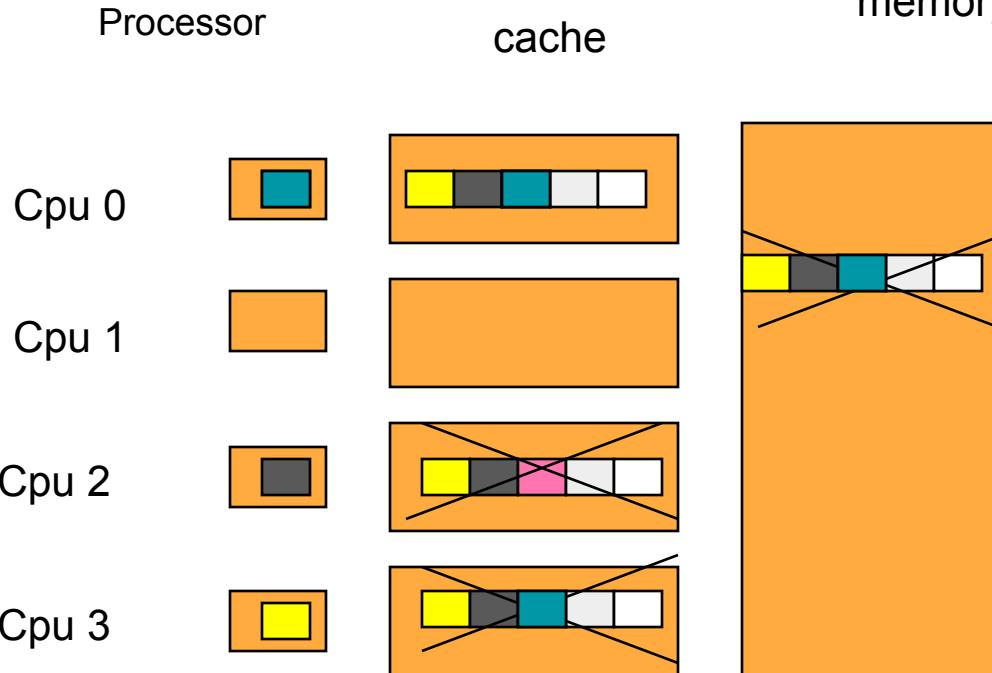




# Caches and Memory

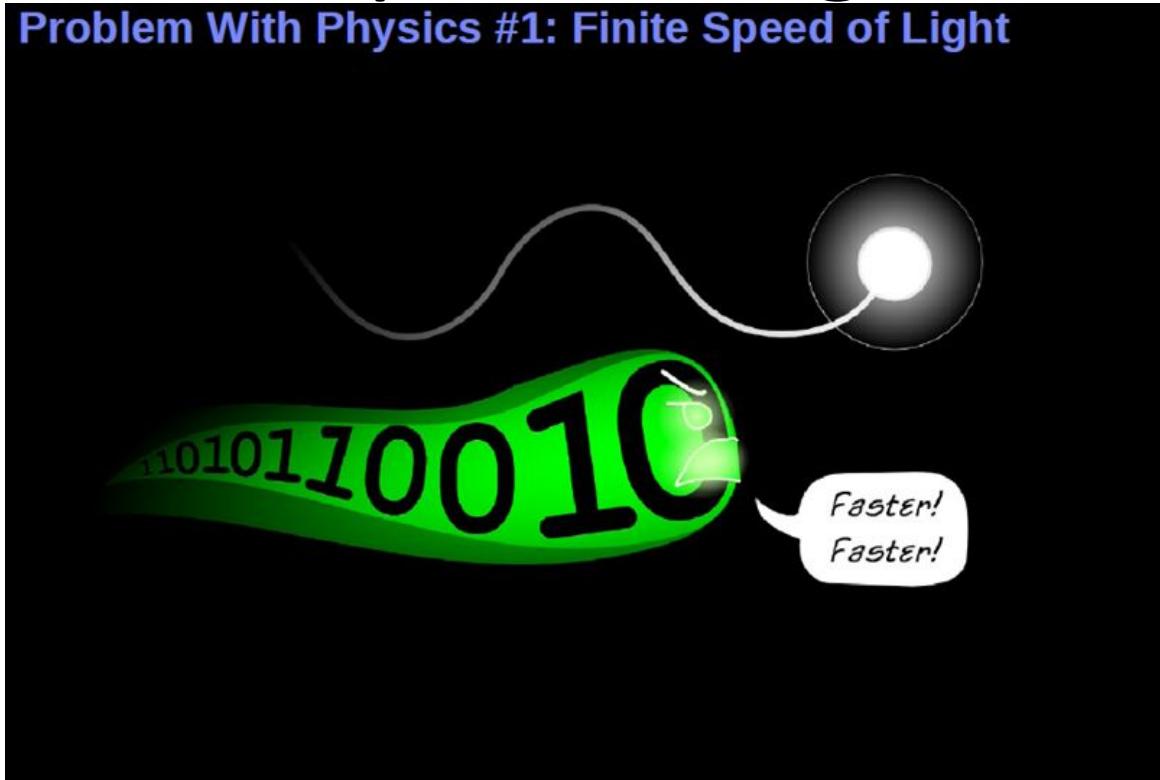


# Caches in an MP system and why do these guys hate us?



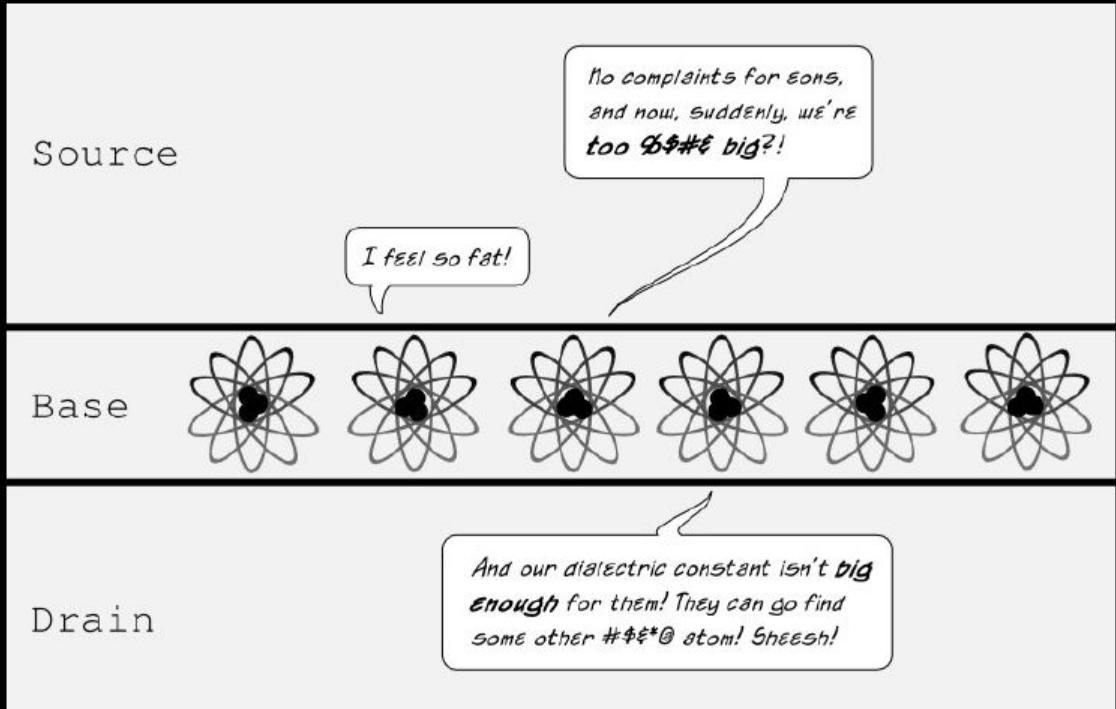
# Problem With Physics #1:finite speed of light

## Problem With Physics #1: Finite Speed of Light



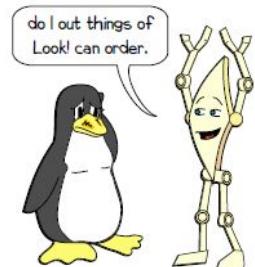
# Problem with Physics #2: atomic nature of matter

## Problem With Physics #2: Atomic Nature of Matter



# Memory Model

- One of the most important aspect of C++0x /C1x is almost invisible to most programmers
  - memory model
    - How threads interact through memory
    - What assumptions the compiler is allowed to make when generating code
    - 2 aspects
      - How things are laid out in memory
      - What happens when two threads access the same memory location and one of them is a modify
        - » Data race
        - » Modification order
        - » **Atomicity/isolation**
        - » **Visibility**
        - » **Ordering**



# Memory Model

- Locks and atomic operations communicate non-atomic writes between two threads
- **Volatile is not atomic (this is not Java)**
- Data races cause undefined behavior
- Some optimizations are no longer legal

# Message shared memory

- Writes are explicitly communicated
  - Between pairs of threads
  - Through a lock or an atomic variable
- The mechanism is acquire and release
  - One thread releases its memory writes
    - `V=32; atomic_store_explicit(&a,3, memory_order_release );`
  - Another thread acquires those writes
    - `i=atomic_load_explicit(&a, memory_order_acquire );`

# What is a memory location

- A non-bitfield primitive data object
- A sequence of adjacent bitfields
  - Not separated by a structure boundary
  - Not interrupted by the null bitfield
  - Avoid expensive atomic read-modify-write operations on bitfields

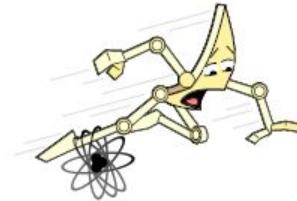
# Data race condition

1. A non-atomic write to a memory location in one thread AND
2. A non-atomic read from or write to that same location in another thread AND
3. With no happens-before relations between them
  - Result in **Undefined Behaviour**
  - Also called “**catch fire semantics**”

# Atomics: To Volatile or Not Volatile

- Too much history in volatile to change its meaning
- It is not used to indicate atomicity like Java
- Volatile atomic
  - means something from the environment may also change this in addition to another thread OR
  - Real time requirements the compiler does not know about, so it further restricts compiler optimizations

# Atomic Design



- Want shared variables
  - **that can be concurrently updated without introducing data race,**
  - **that are atomically updated and read**
    - half updated states are not visible,
- that are implemented without lock overhead whenever the hardware allows,
- that provide access to hardware atomic read-modify write (fetch-and-add, xchg, cmpxchg) instructions whenever possible.

# Race Free semantics and Atomic Memory operations

- If a program has a race, it has undefined behavior
  - This is sometimes known as “catch fire” semantics
  - No compiler transformation is allowed to introduce a race
    - More restrictions on invented writes
    - Possibly fewer speculative stores **and (potentially) loads**
- There are atomic memory operations that don’t cause races (or they race but are well-defined)
  - Can be used to implement locks/mutexes
  - Also useful for lock-free algorithms
- Atomic memory operations are expressed as library function calls
  - Reduces need for new language syntax

# Atomic Operations and Type

- Data race: if there is no enforced ordering between two accesses to a single memory location from separate threads, one or both of those accesses is not atomic, and one or both is a write, this is a data race, and causes undefined behavior.
- These types avoid undefined behavior and provide an ordering of operations between threads

# Lock-free atomics

- Large atomics have no hardware support
  - Implemented with locks
- Locks and signals don't mix
  - Test for lock-free
- Compile-time macros for basic types
  - Always lock-free
  - Never lock-free
  - May be lock-free
- RTTI for each type

# Memory Ordering Operations

```
enum memory_order {  
    memory_order_relaxed, // just atomic, no constraint  
    memory_order_release,  
    memory_order_acquire,  
    memory_order_consume,  
    memory_order_acq_rel, // both acquire and release  
    memory_order_seq_cst}; // sequentially consistent
```

- Every atomic operation has a default form, implicitly using seq\_cst, and a form with an explicit order argument
- When specified, argument is expected to be just an enum constant

# 3 Memory Models, but 6 Memory Ordering Constraints

- Sequential Consistency
  - Single total order for all SC ops on all variables
  - default
- Acquire/Release/consume
  - Pairwise ordering rather than total order
  - Independent Reads of Independent Writes don't require synchronization between CPUs
- Relaxed Atomics
  - Read or write data without ordering
  - Still obeys happens-before
- Operations on variable have attributes, which can be explicit

# Operations available on atomic types

	atomic_flag	bool/other-type	T*	integral
test_and_set, clear	Y			
is_lock_free		Y	Y	Y
load, store, exchange, compare_exchange_weak +strong		Y	Y	Y
fetch_add (+=), fetch_sub (-=), ++, --			Y	Y
fetch_or ( =), fetch_and (&=), fetch_xor (^=)				Y

# Examples

```
int x = 0;  
atomic<int> y = 0;
```

*Thread 1:*

```
x = 17;  
y.store(1,  
memory_order_release);  
// or: y.store(1);
```

*Thread 2:*

```
while  
(y.load(memory_order_acquire)  
!= 1)  
// or: while (y.load() != 1)  
  
assert(x == 17);
```

```
int x = 0;  
atomic<int> y = 0;
```

*Thread 1:*

```
x = 17;  
y = 1;
```

*Thread 2:*

```
while (y != 1)  
continue;  
assert(x == 17);
```

# Key takeaways

Atomics for experts only

Supports Lock free data structures

Use sequential consistency for understanding

Use more relaxed models for performance

