Fundamentals of Parallelism

Gordon Brown & Michael Wong

CppCon 2020 – Sep 2020

- Learning objectives:
  - Learn about communication patterns
  - Learn about reordering algorithms
  - Learn about handling dependencies
  - Learn about work distribution

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

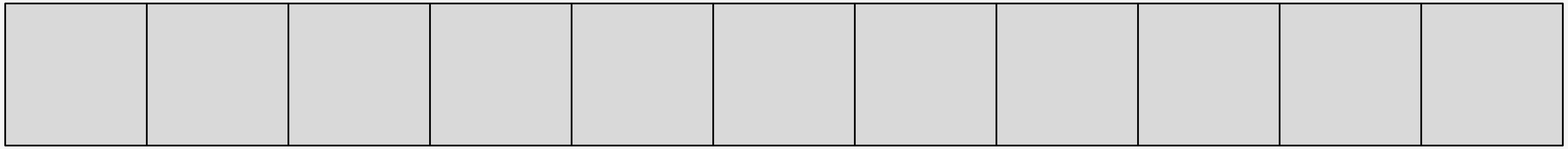
This requires communication, and in parallel computing this is done via memory

Communication patterns

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

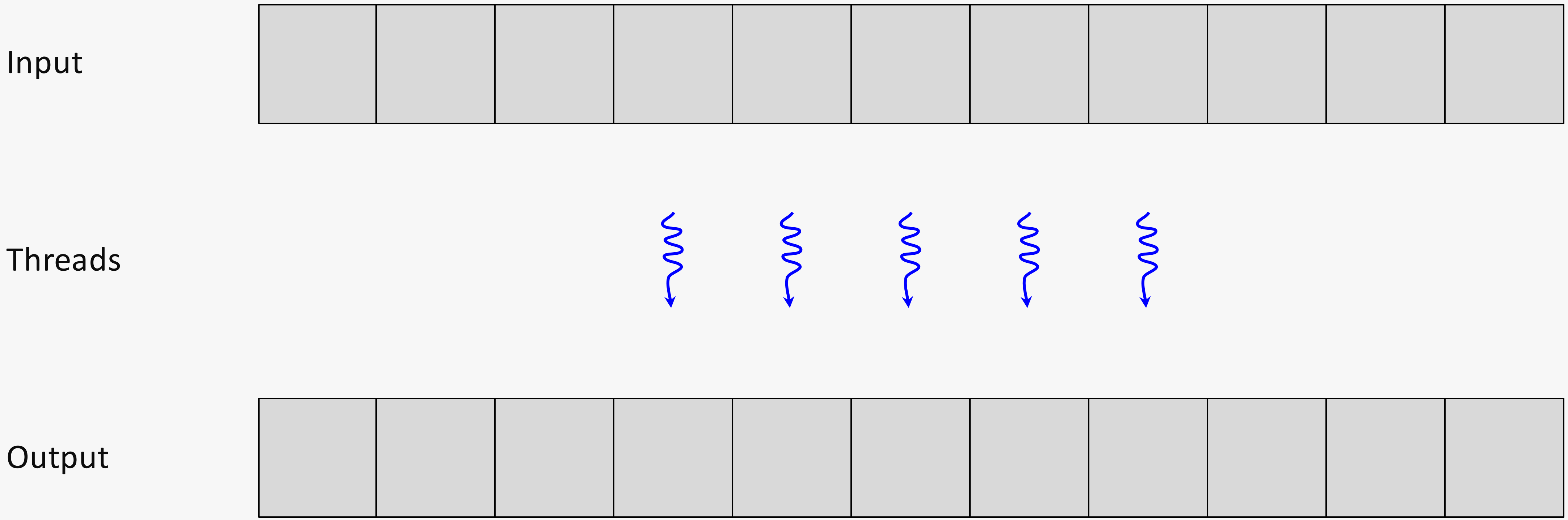
This requires communication, and in parallel computing this is done via memory

Memory



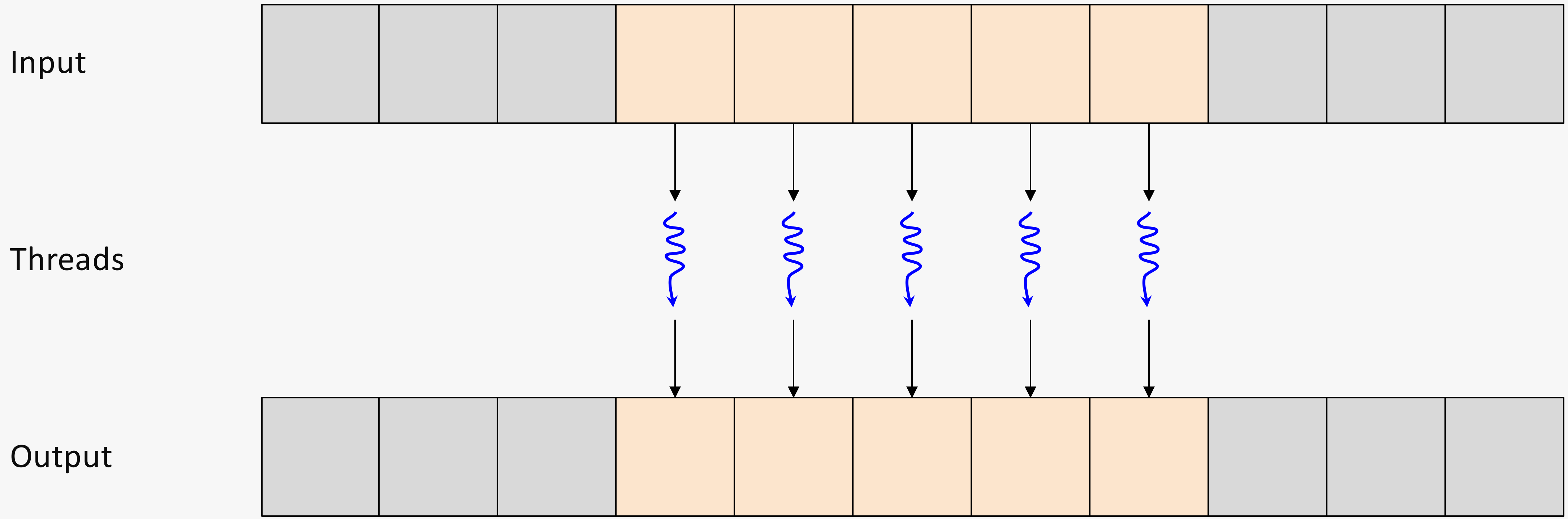
Communication patterns

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



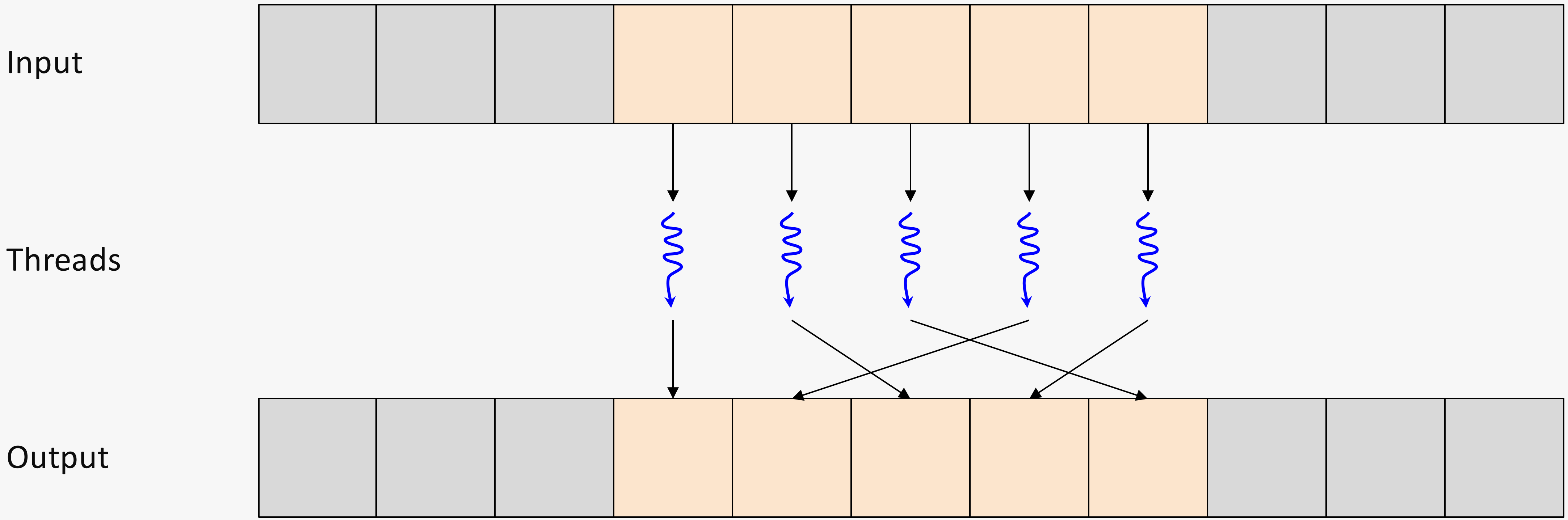
Map pattern

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



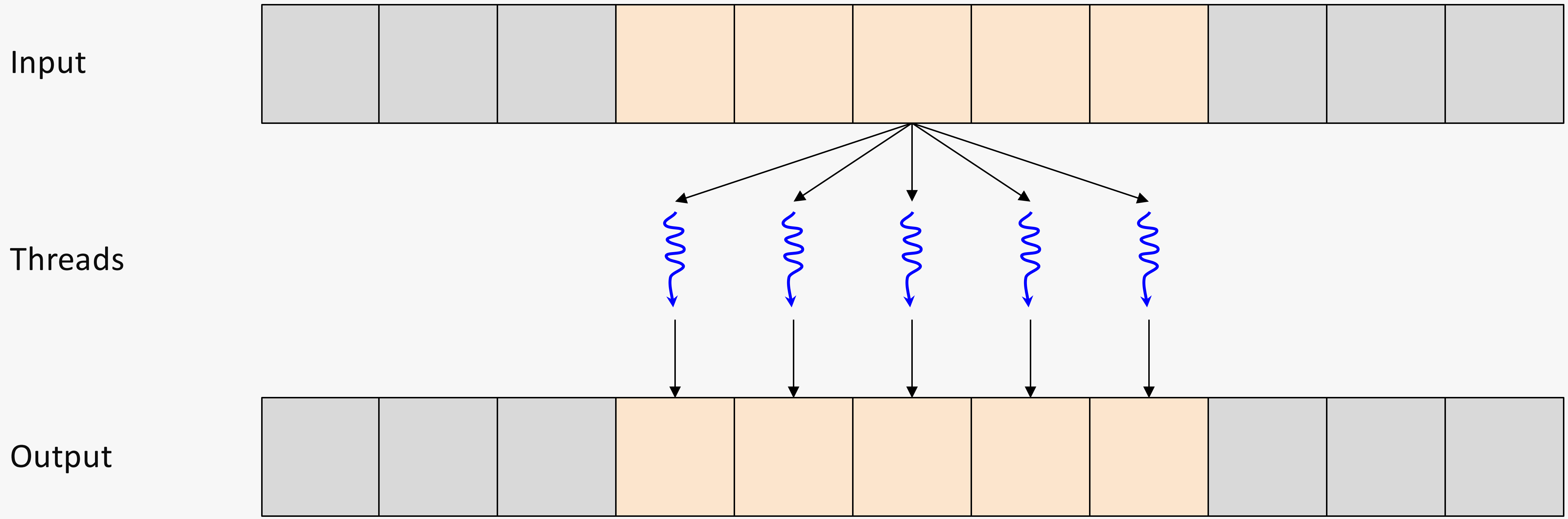
Transpose pattern

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



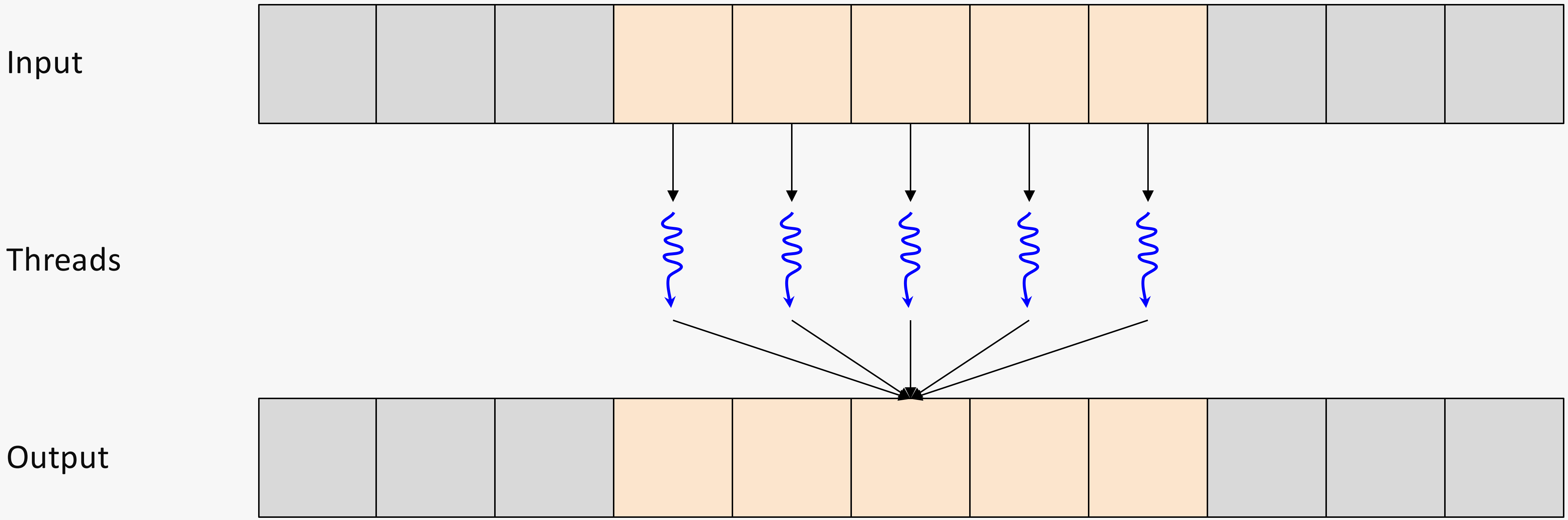
Scatter pattern

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



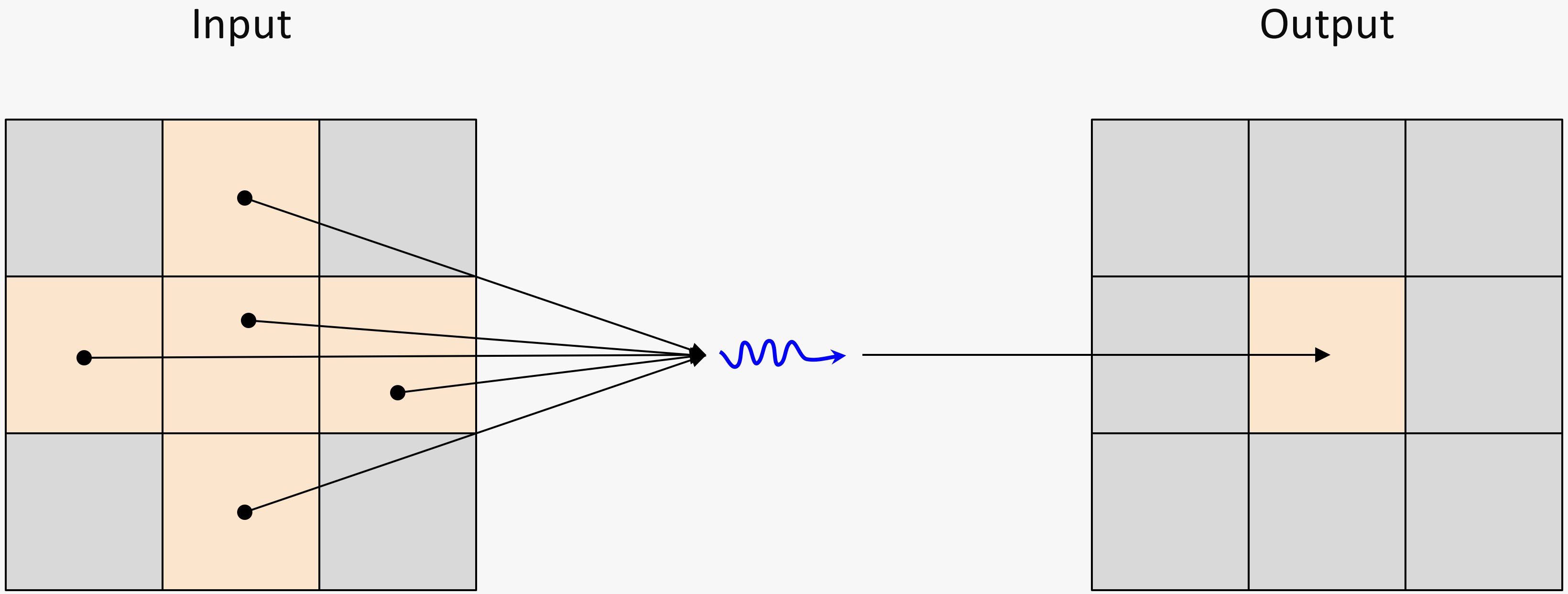
Gather pattern

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



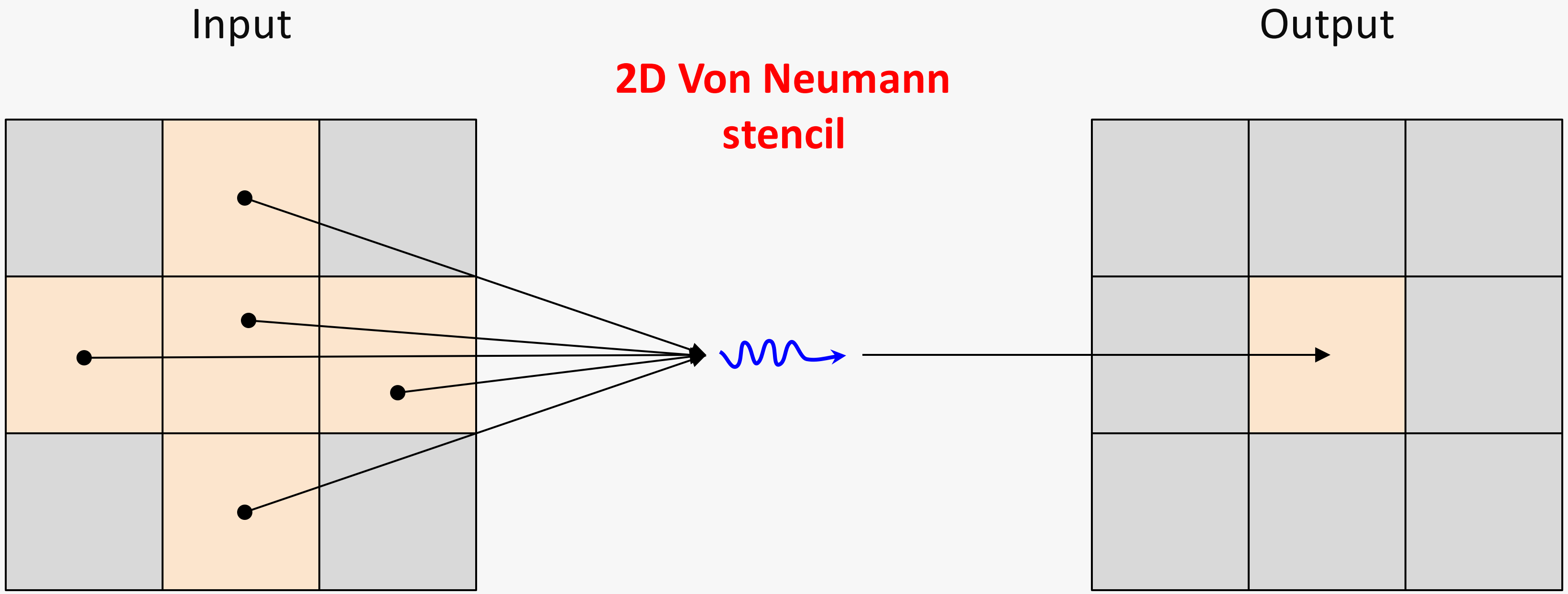
Stencil pattern

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?

```
void foo(int *in, int *out, int index) {
  out[index] = pi * in[128 - index];
}
```

Map Transpose Scatter Gather Stencil

What kind of communication pattern does this code embody?

```
void foo(int *in, int *out, int index) {
  out[index] = pi * in[128 - index];
}
```

Map

Transpose

Scatter

Gather

Stencil

What kind of communication pattern does this code embody?

```
void foo(int *in, int *out, int index) {
  out[index] = pi * in[index];
}
```

Map Transpose Scatter Gather Stencil

What kind of communication pattern does this code embody?

```
void foo(int *in, int *out, int index) {
  out[index] = pi * in[index];
}
```

Map Transpose Scatter Gather Stencil

What kind of communication pattern does this code embody?

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

Map Transpose Scatter Gather Stencil

What kind of communication pattern does this code embody?

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

Map   Transpose   Scatter   Gather   Stencil

What kind of communication pattern does this code embody?

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

Map Transpose Scatter Gather Stencil

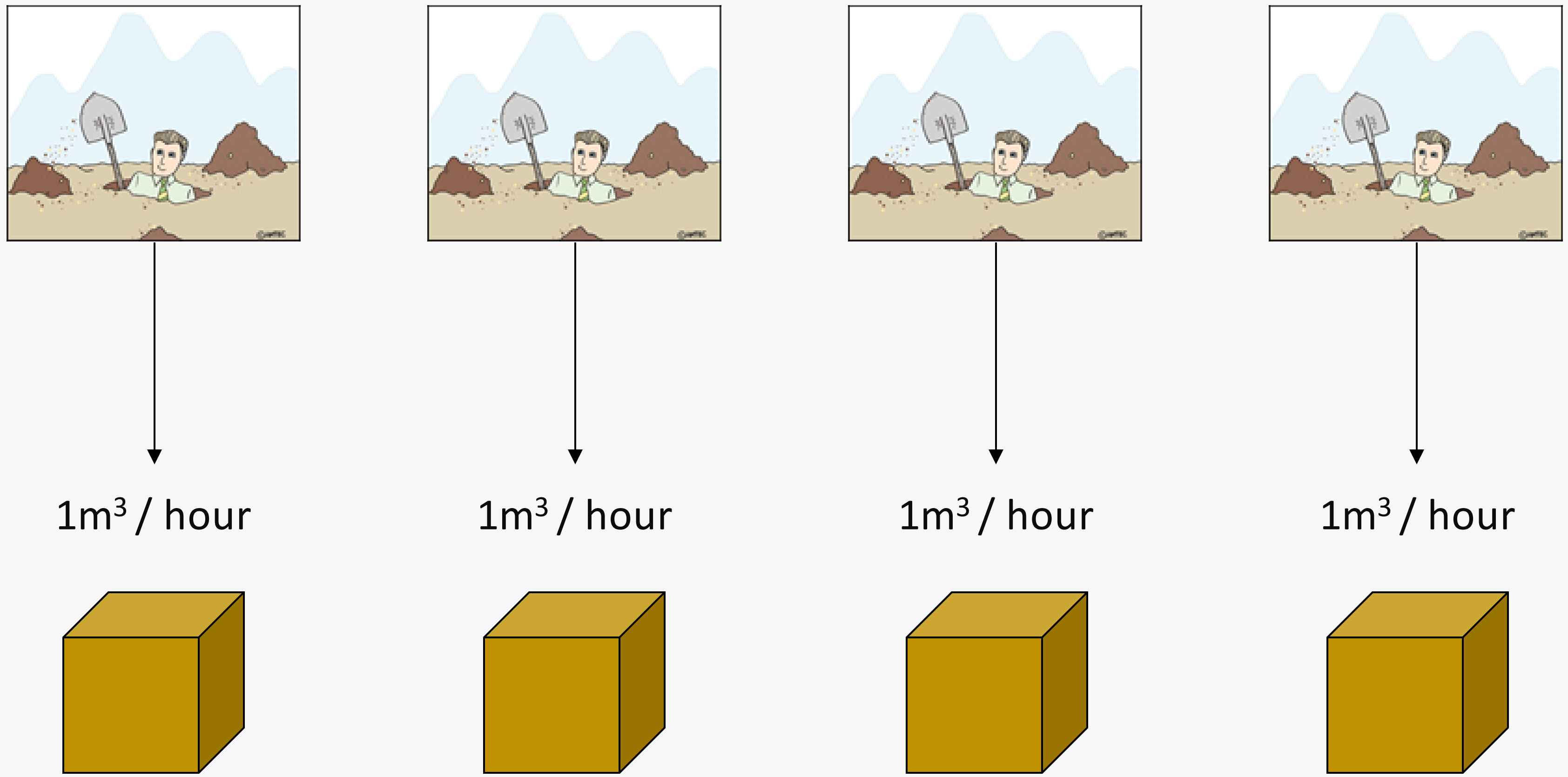
What kind of communication pattern does this code embody?

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

Map Transpose Scatter Gather Stencil

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

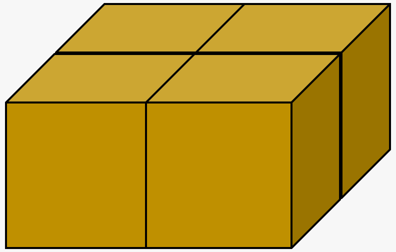
Say you now have four diggers...



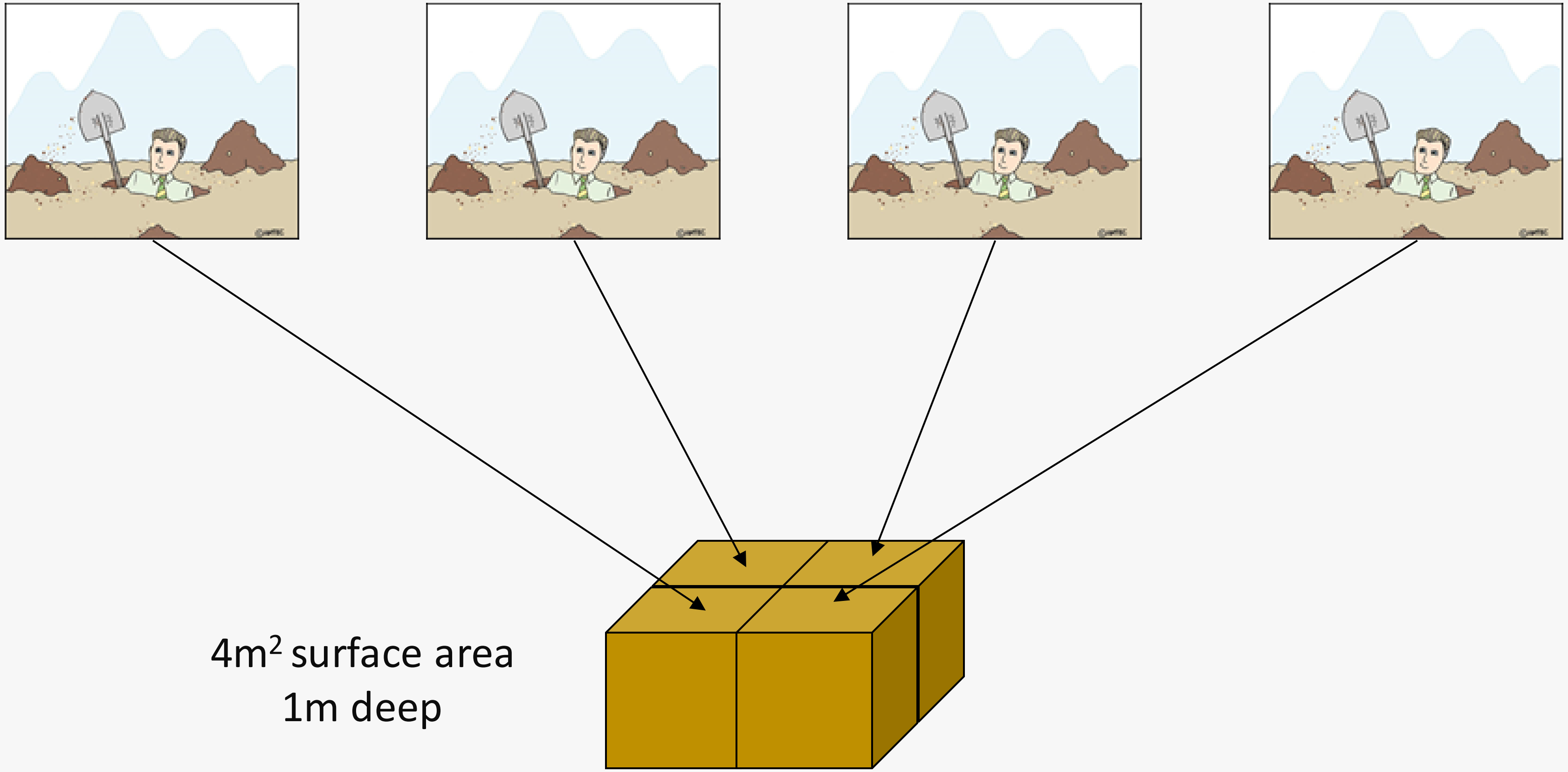
Say you want a hole with a $4m^2$ surface area




$4m^2$ surface area
1m deep

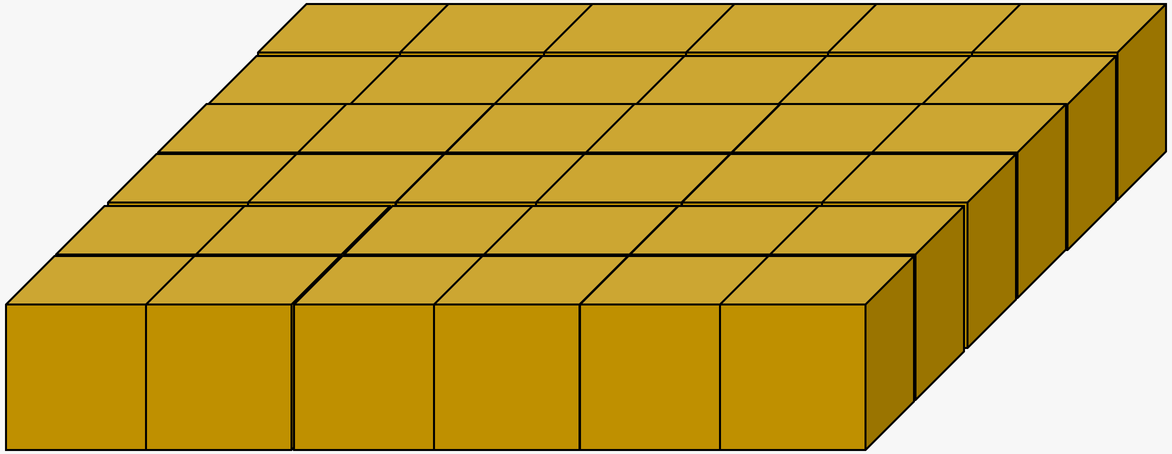


Each digger digs a part each

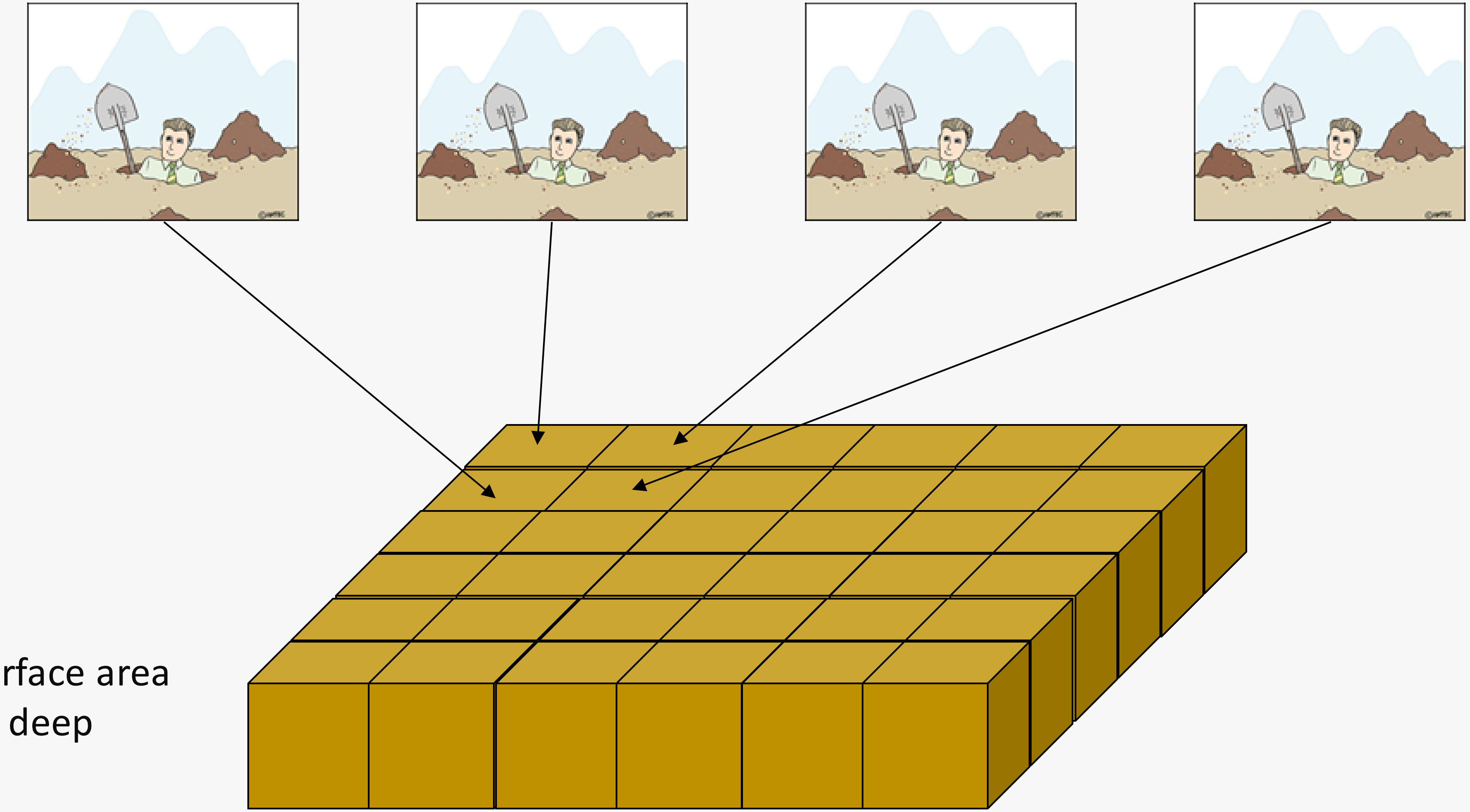


Say you want a hole with a $36m^2$ surface area

$36m^2$ surface area
1m deep

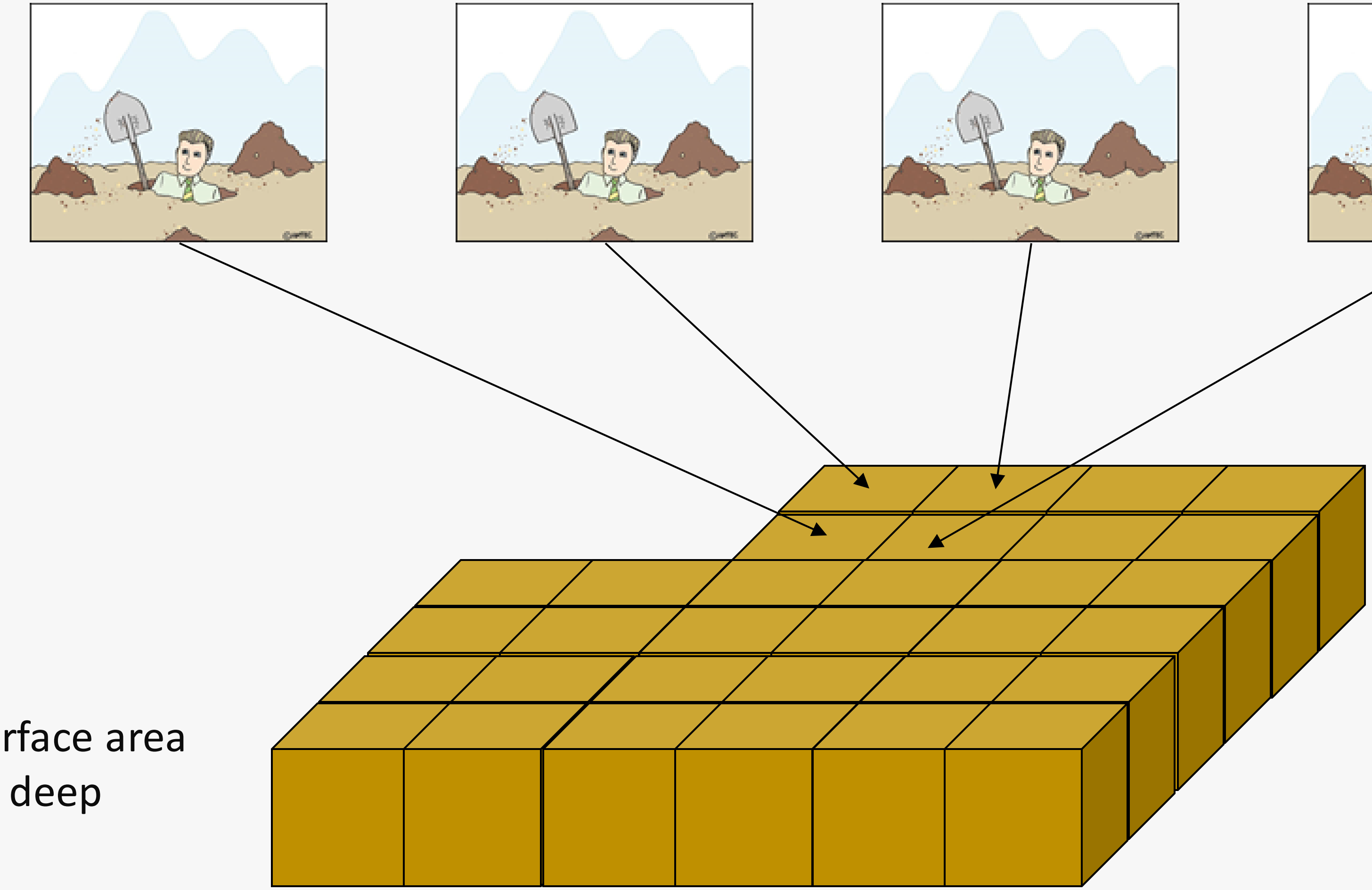


You can share the work between the diggers



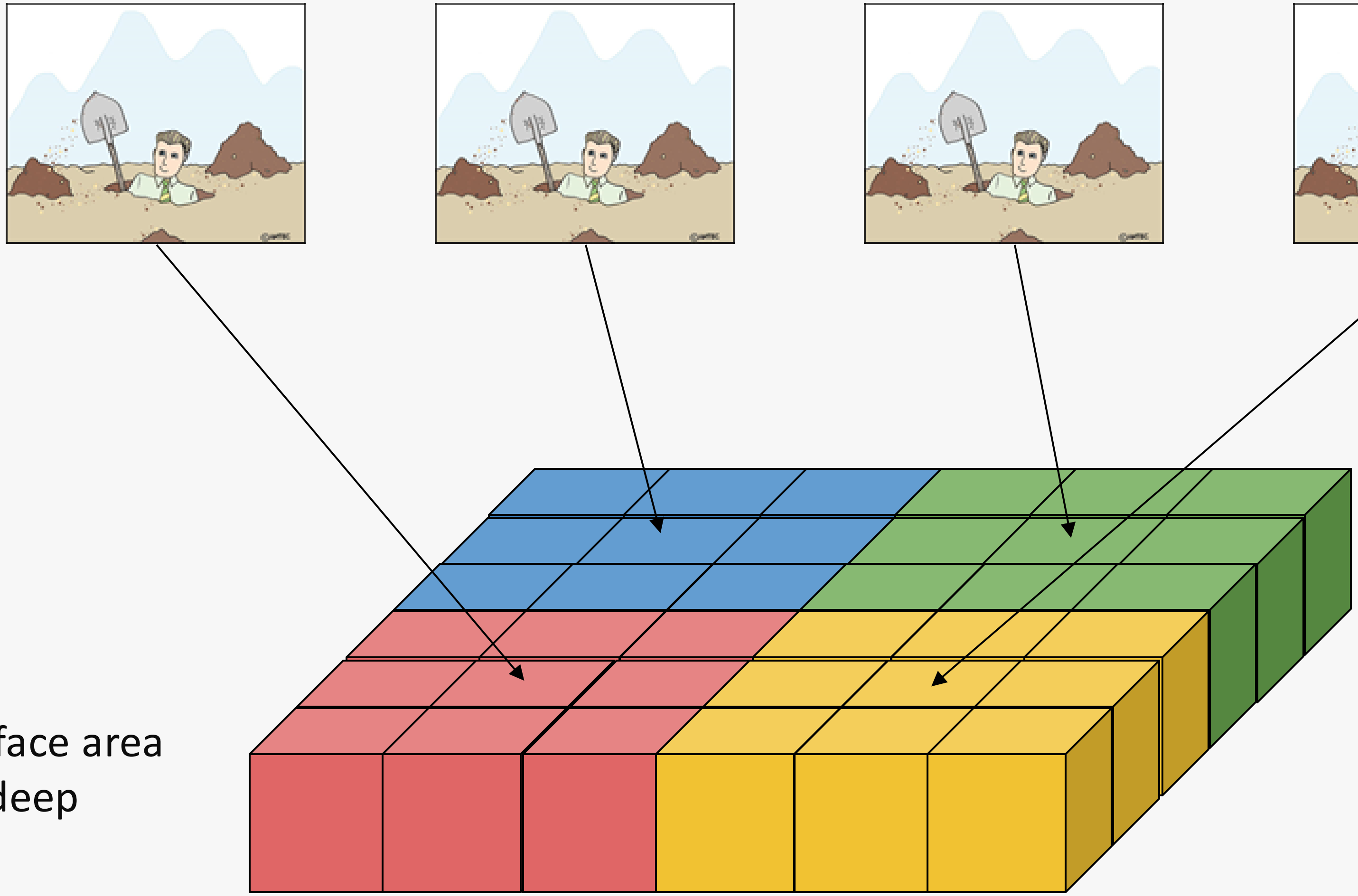
$36m^2$ surface area
1m deep

You can share the work between the diggers



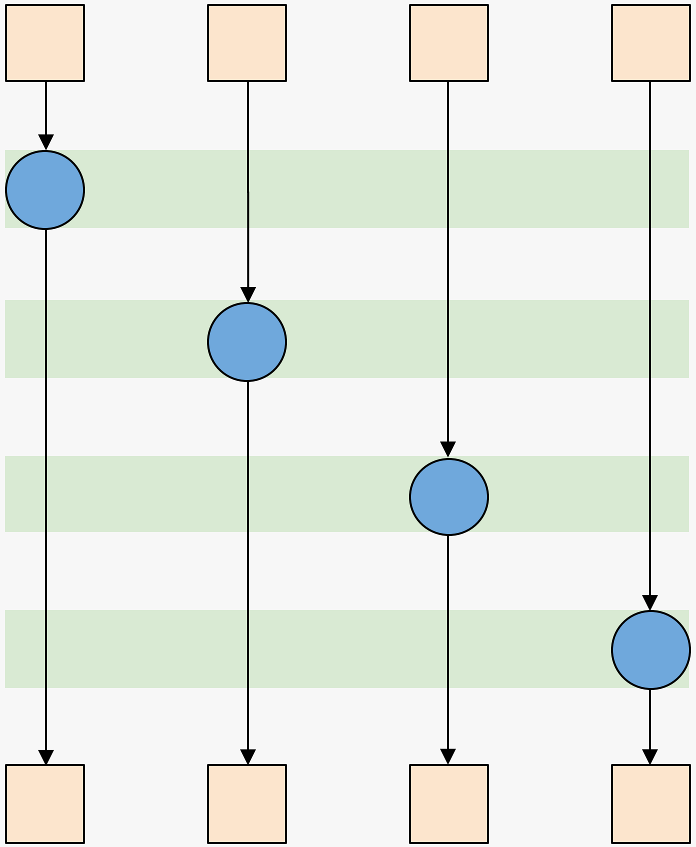
36m^2 surface area
1m deep

You can distribute work across the diggers

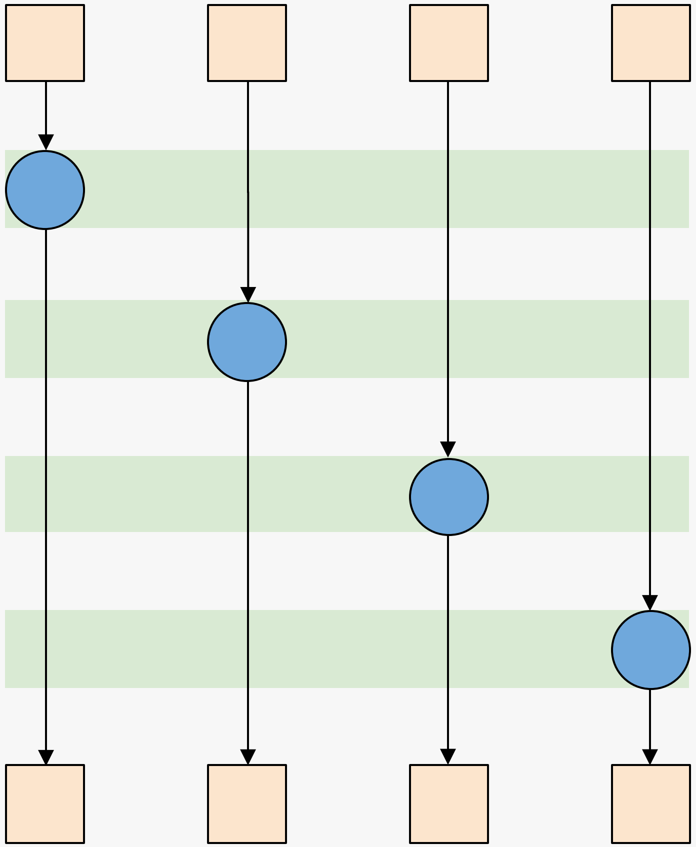


36m² surface area
1m deep

This applies to a transform algorithm

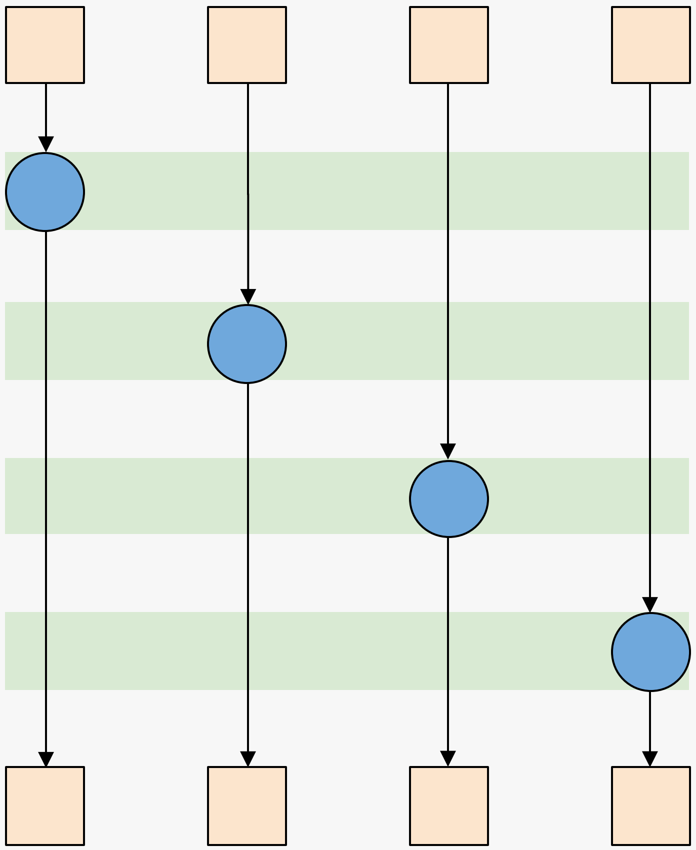


Let’s look at a serial transform...



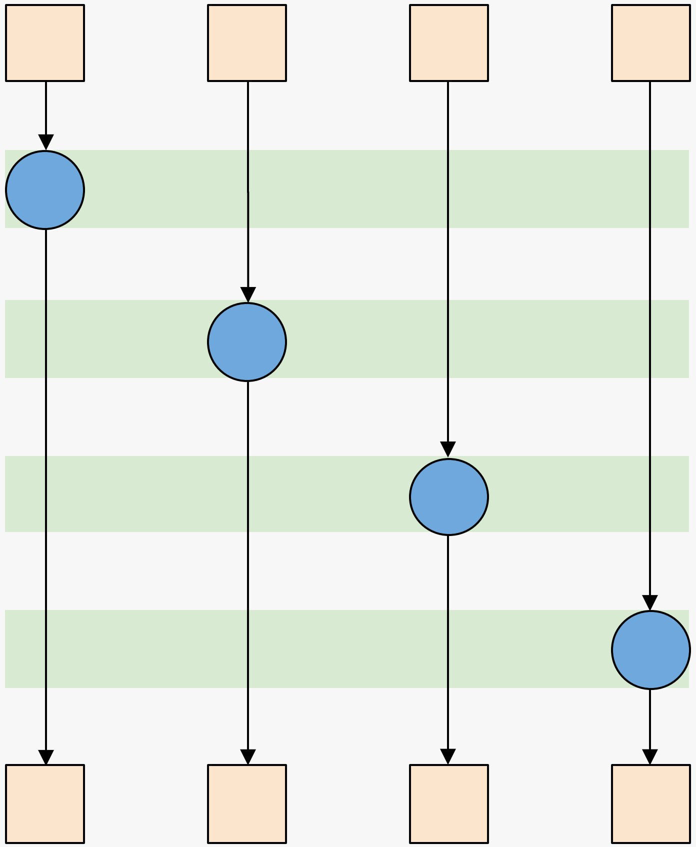
4 elements

Let’s look at a serial transform...



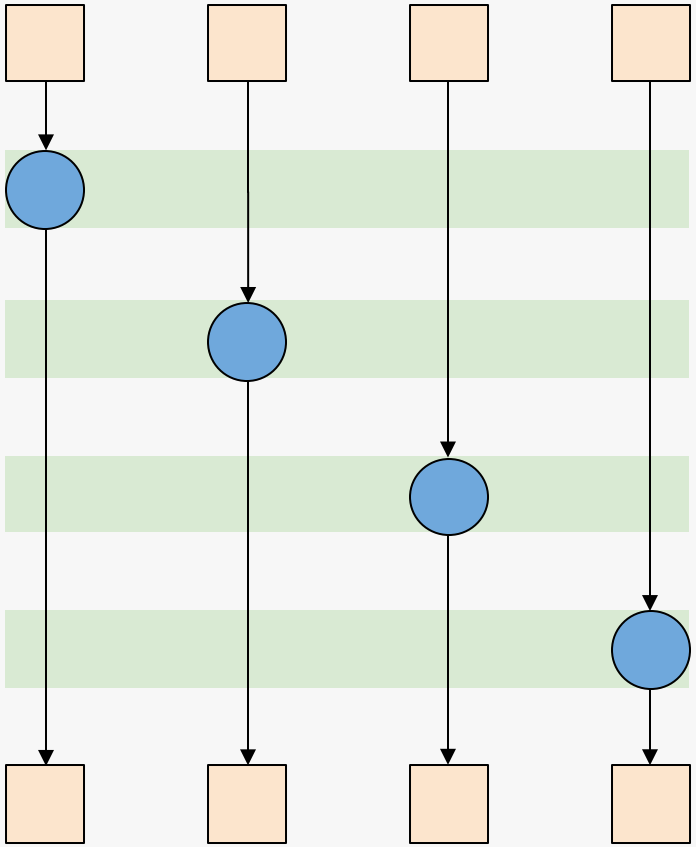
4 elements | 4 Operations

Let’s look at a serial transform...



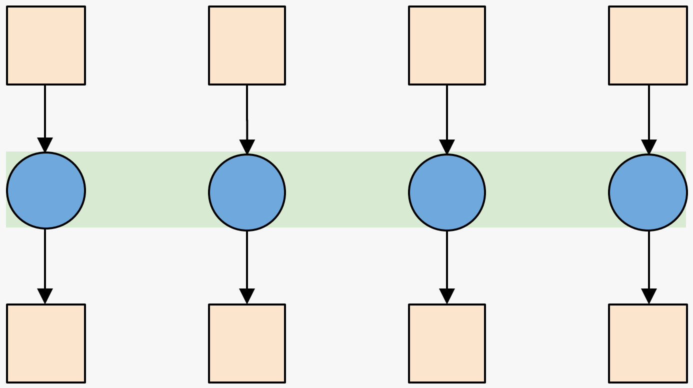
4 elements | 4 Operations| 4 steps

Let’s look at a serial transform...



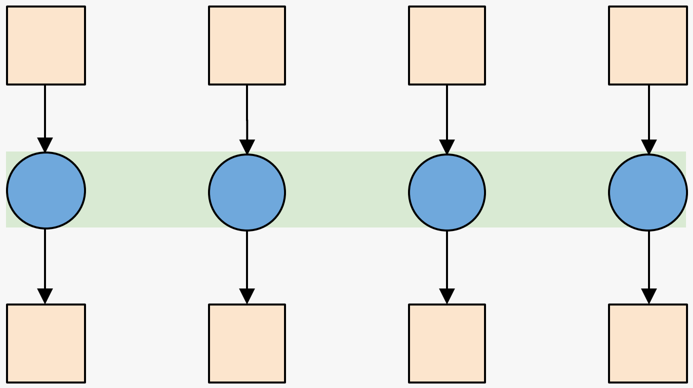
4 elements | 4 Operations | 4 steps | 1 operations / step

Now let's look at a parallel transform...



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

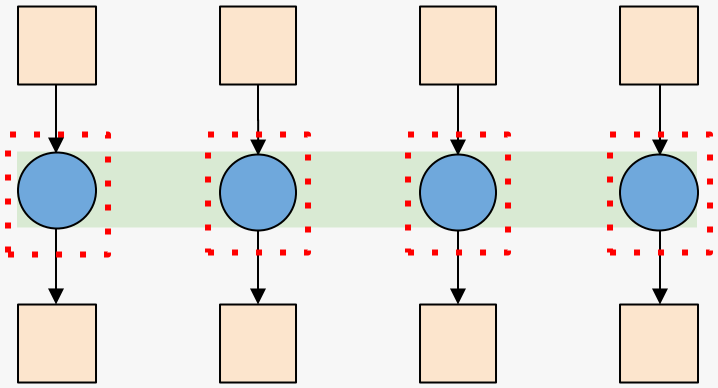
Now let's look at a parallel transform...



Brent's theorem

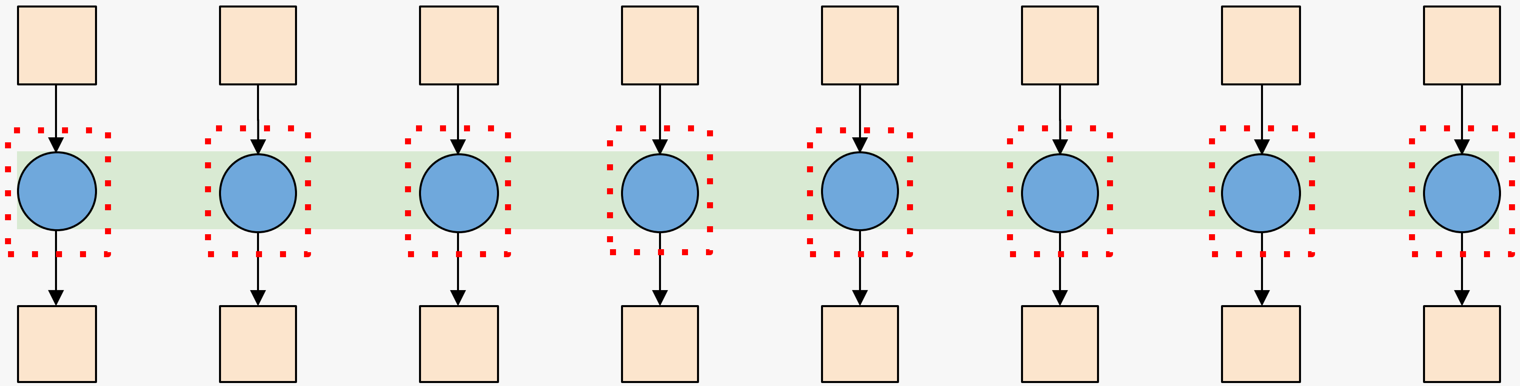
4 elements | 4 Operations | 1 step | 4 operations / step

In order to do this you need parallel workers



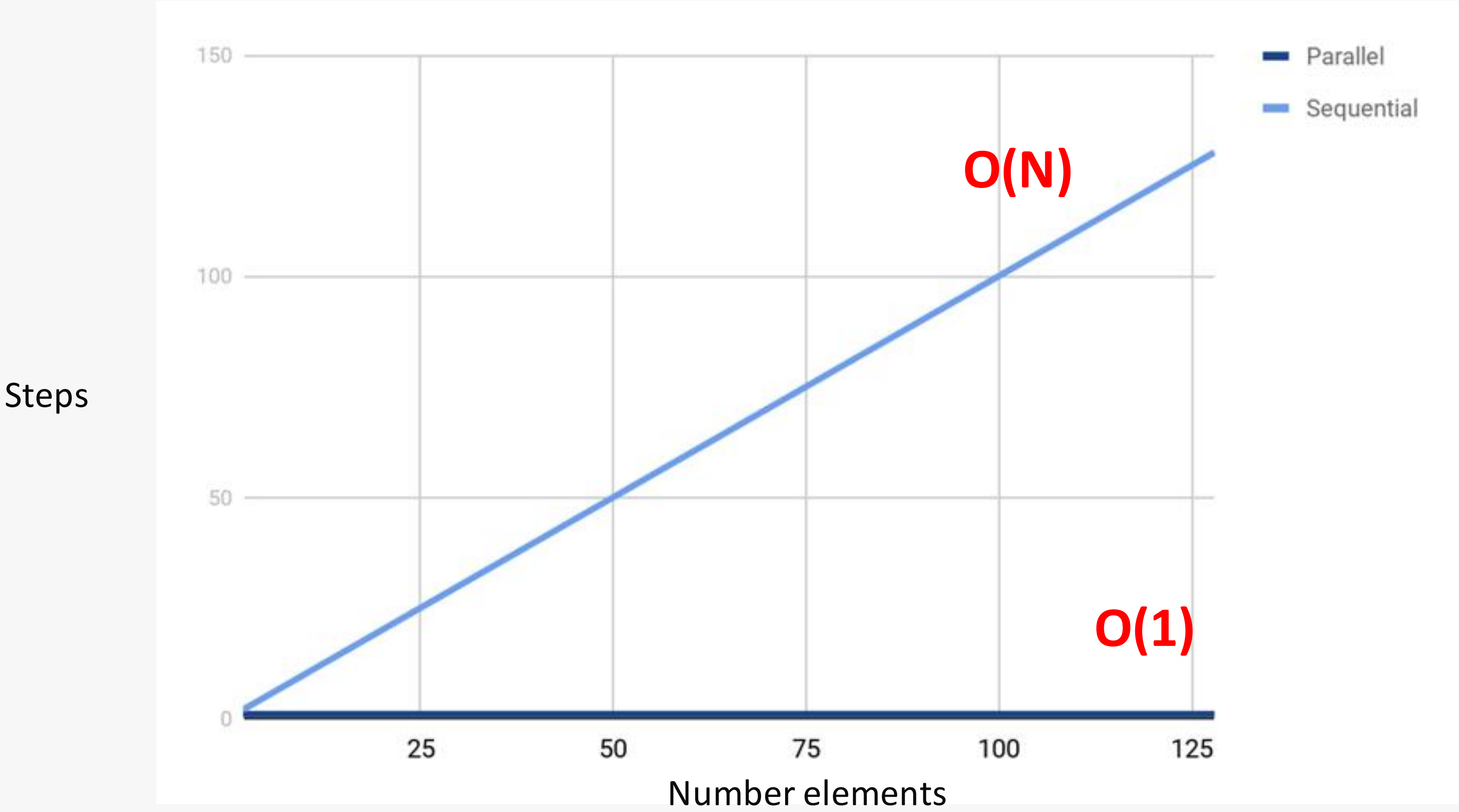
4 elements | 4 Operations | **4 workers** | 1 steps | 4 operations / step

Now let’s scale this up…

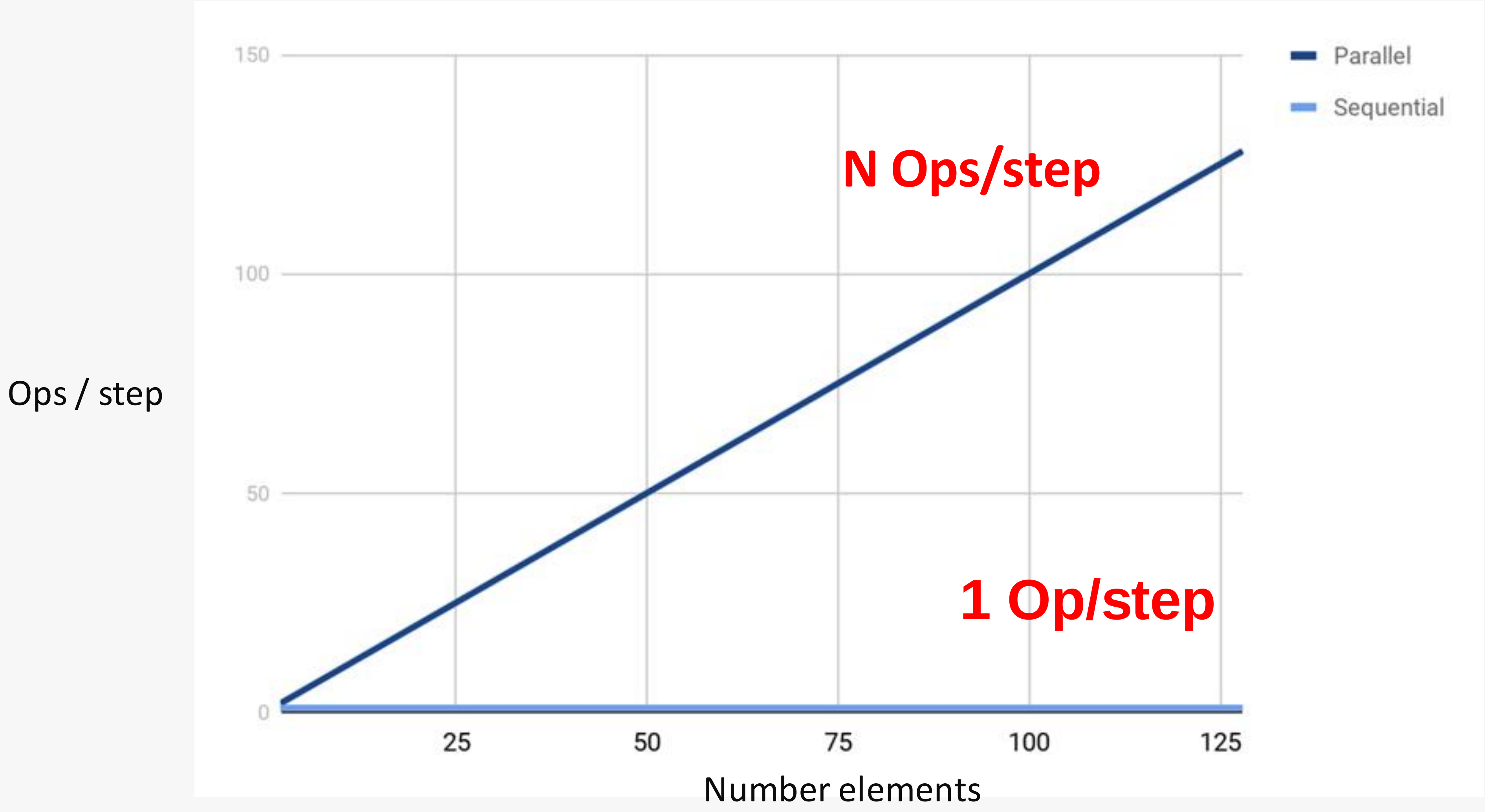


8 elements | 8 Operations | 8 workers | 1 step | 8 operations / step

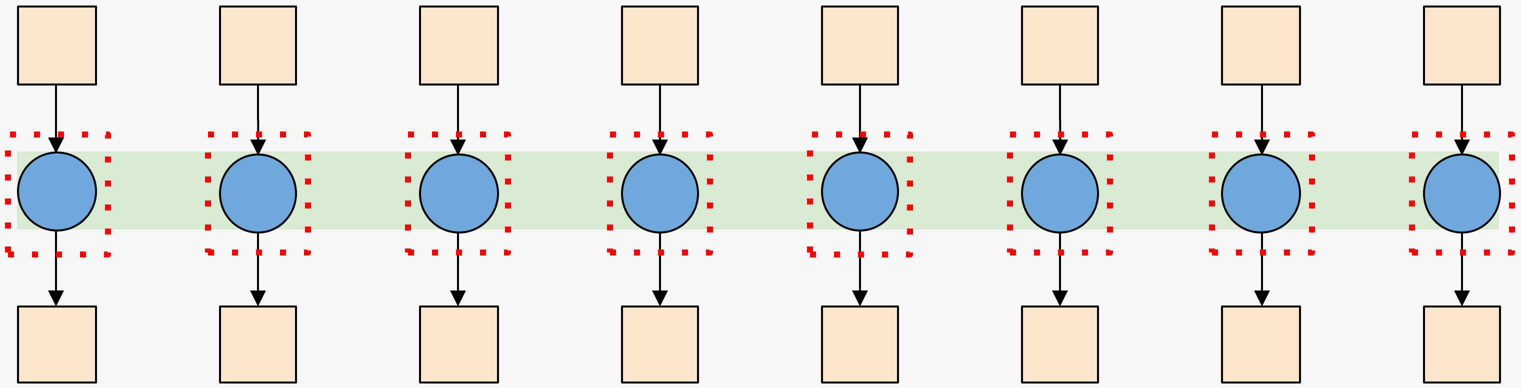
Step complexity of transform



Theoretical operations per step

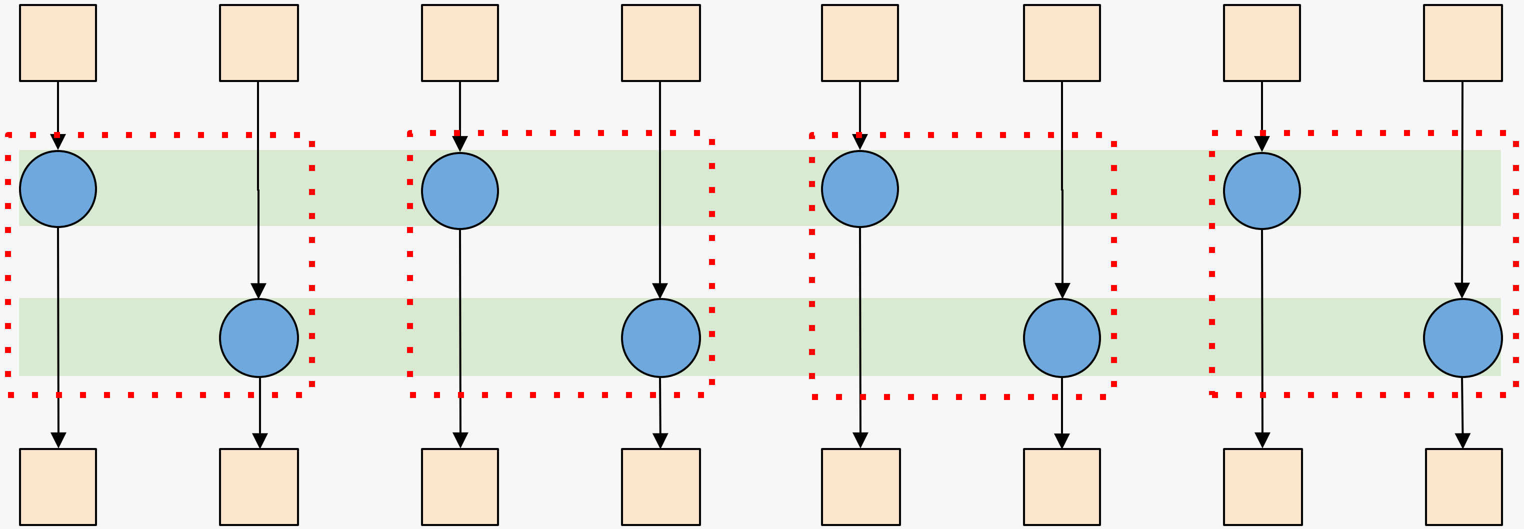


What happens if you only have 4 workers?



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

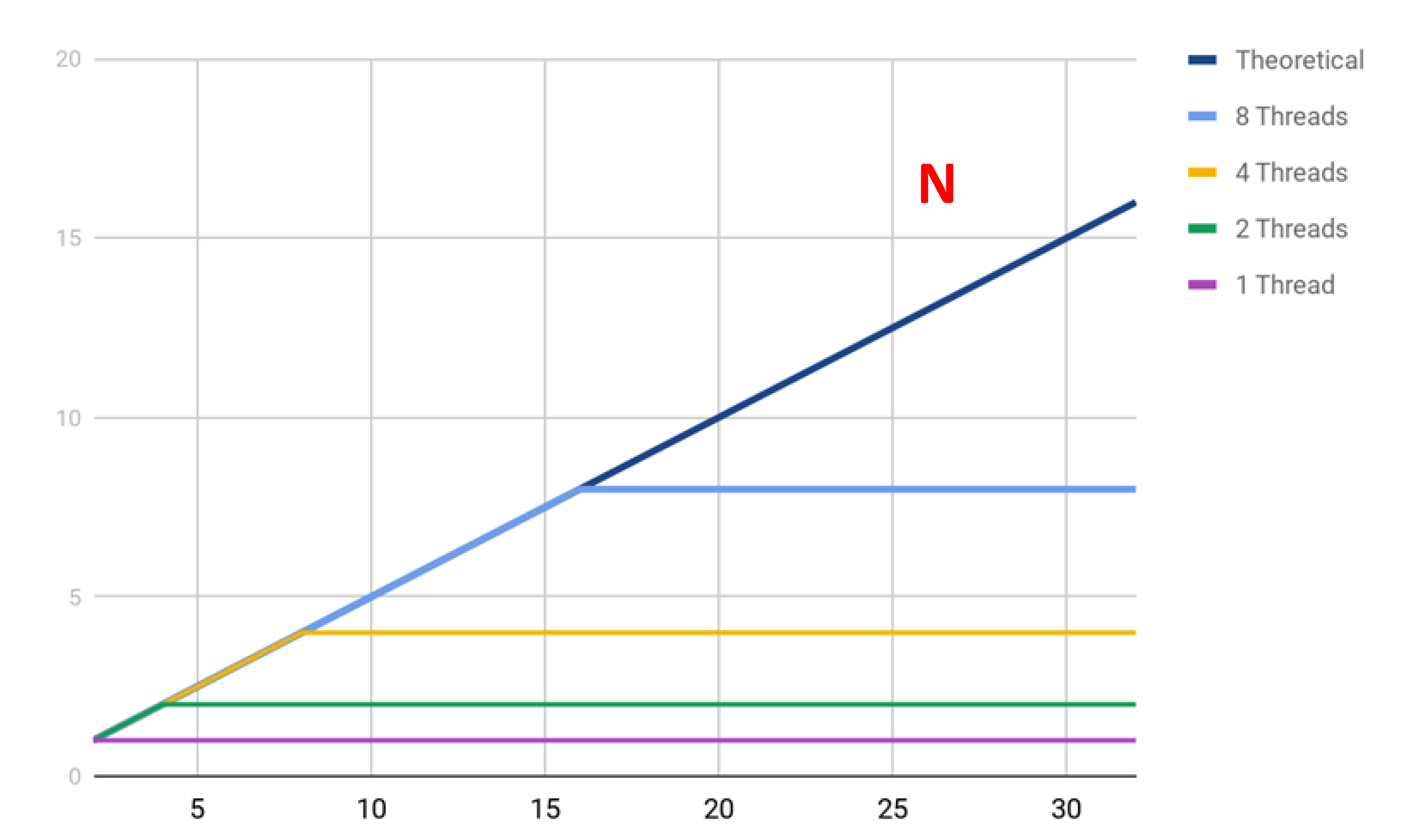
You have to batch work together



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

Actual operations per step

Ops / step



Number elements

Maximizing throughput

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

Maximising the actual operations / step will provide optimal throughout

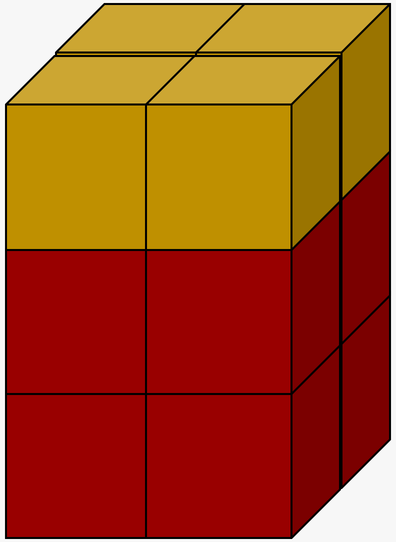
You will most often have a much larger number of operations to perform than available workers

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

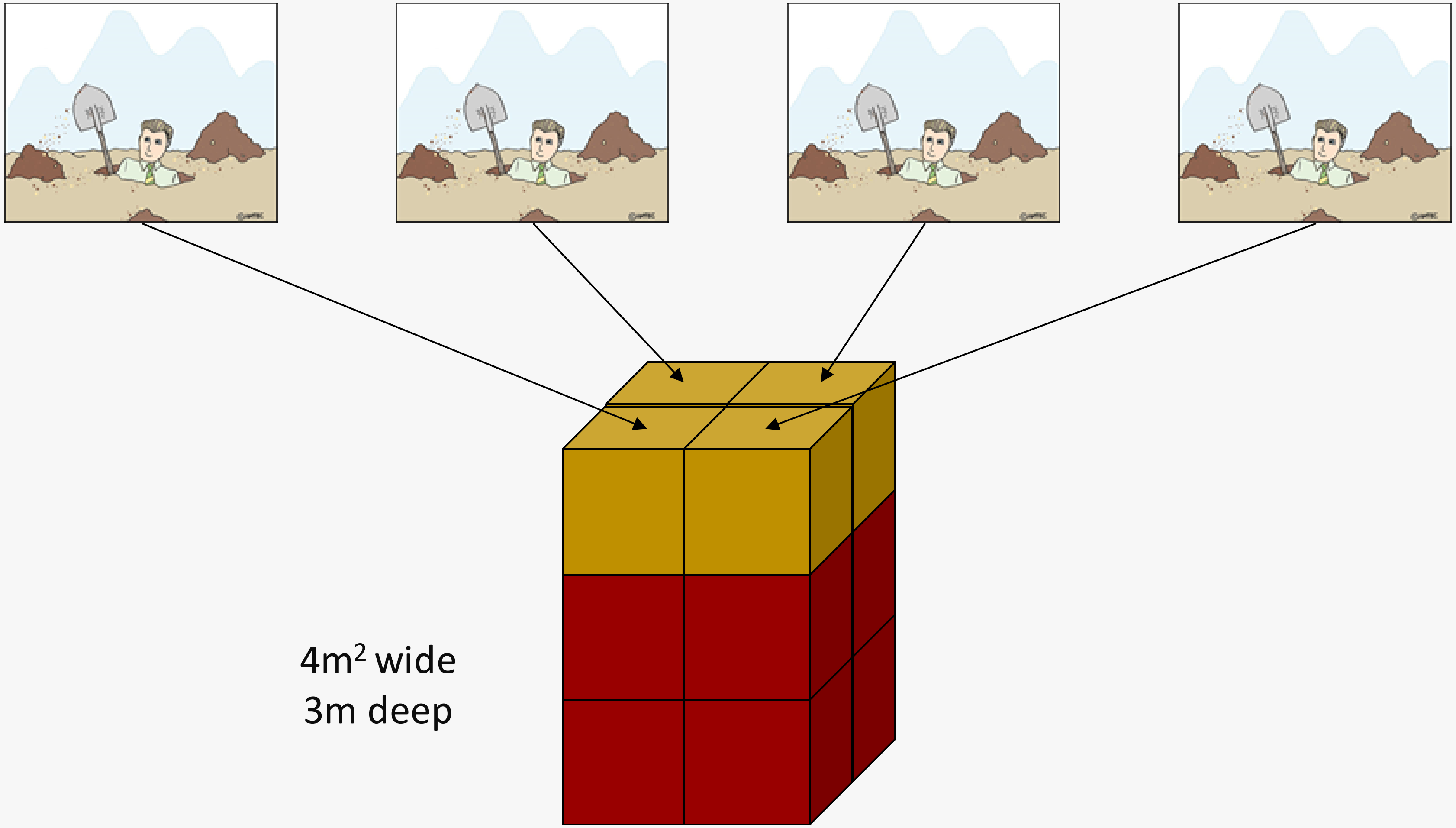
Say you want to dig a hole 3m deep




4m^2 wide
3m deep

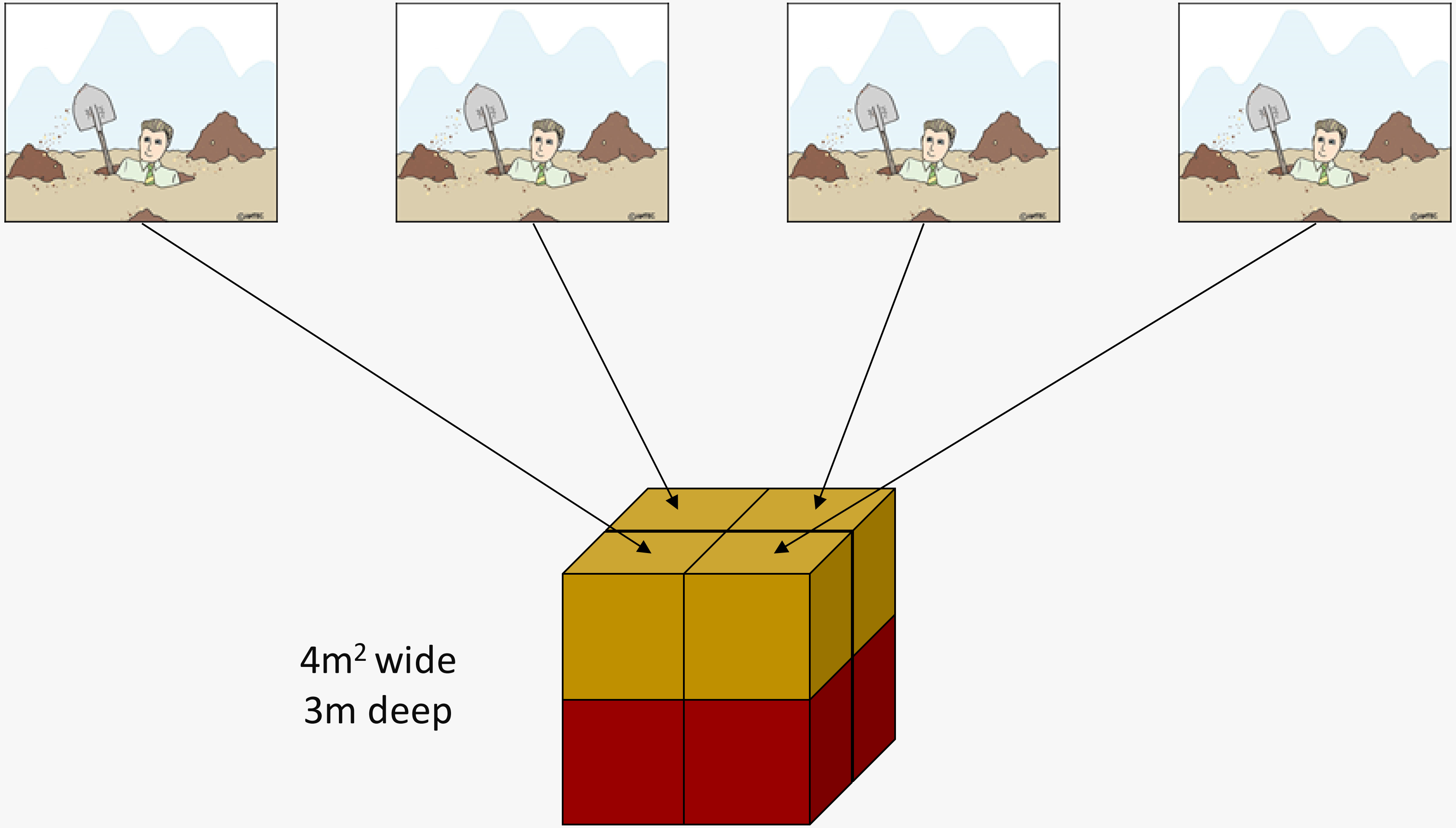


All four diggers have work to do



$4m^2$ wide
3m deep

All four diggers have work to do

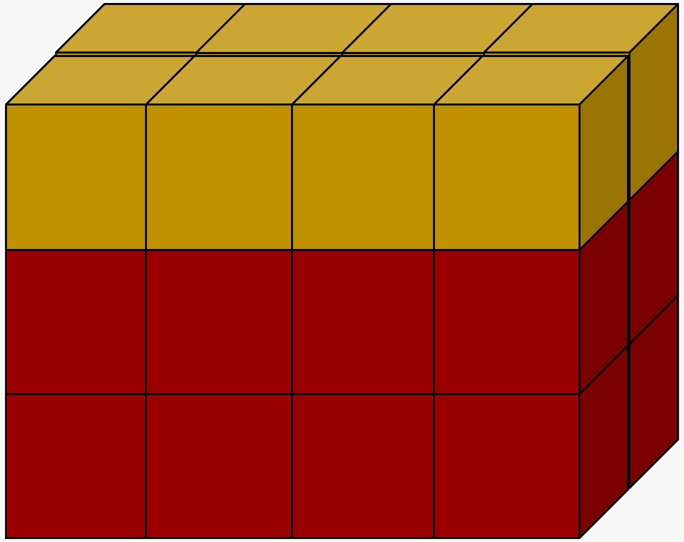


4m^2 wide
3m deep

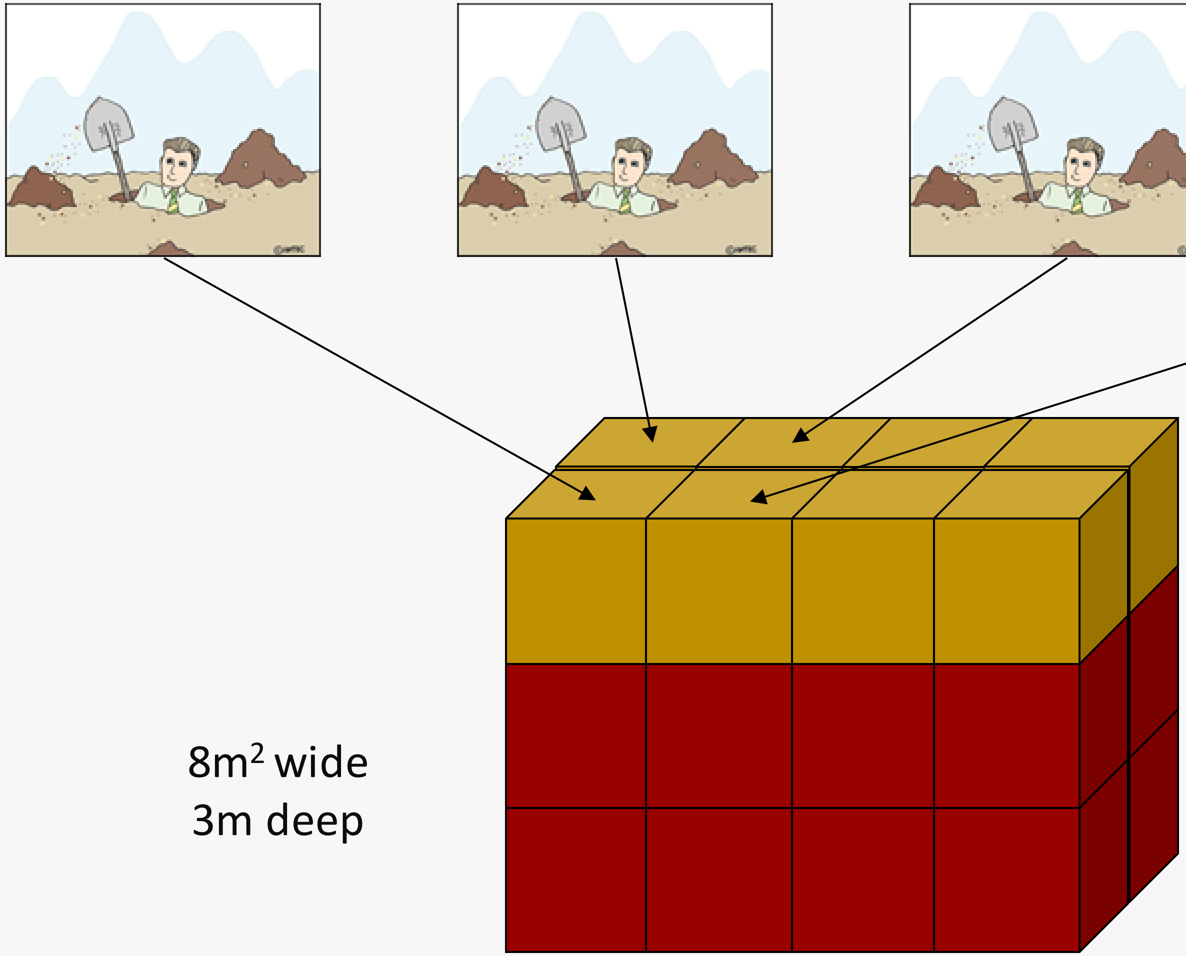
Now say the hole is $8m^2$ wide

$8m^2$ wide
3m deep

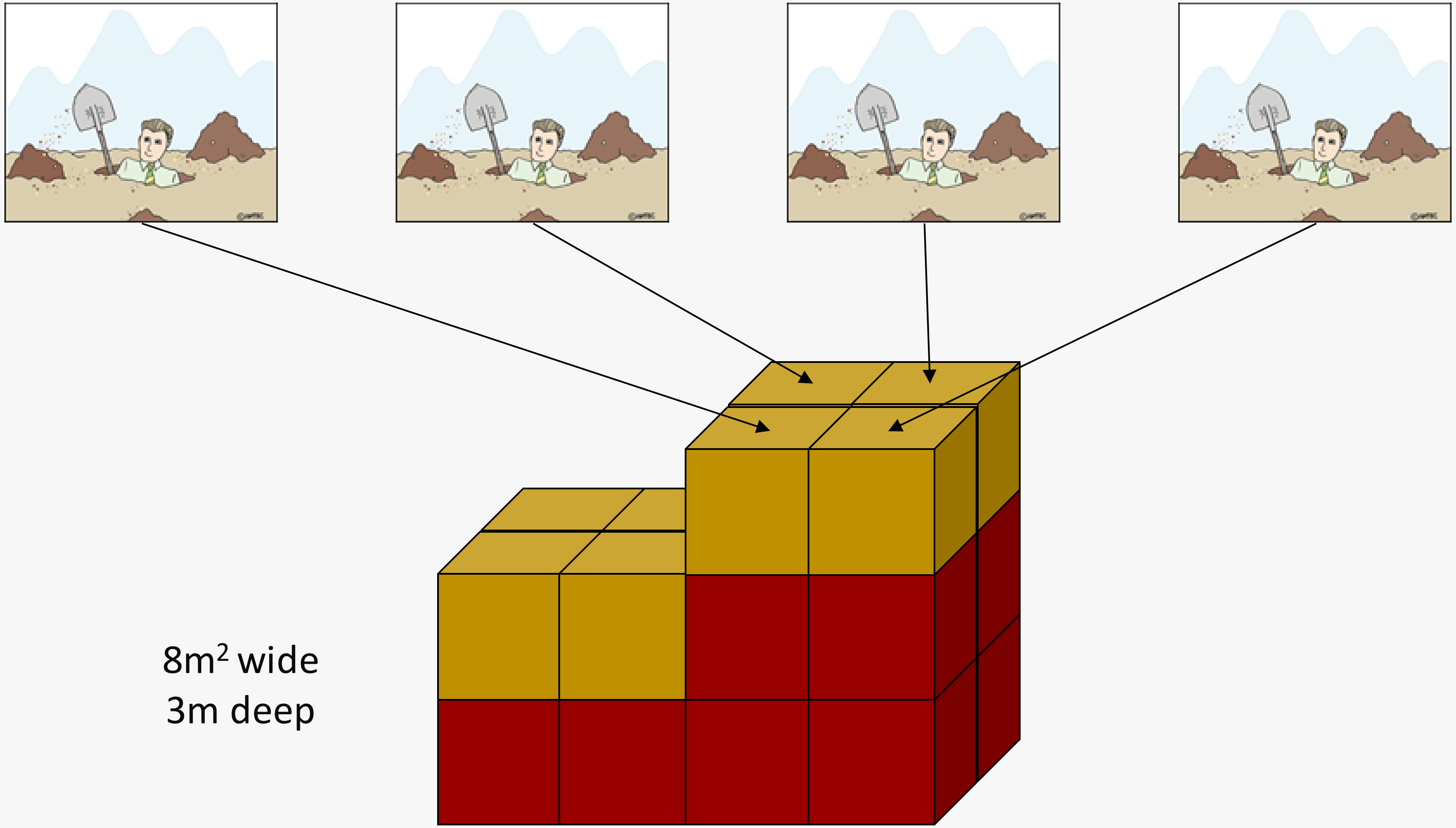


Again can share the work



$8m^2$ wide
3m deep

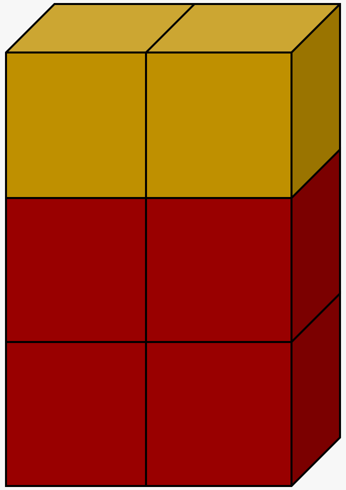
Again can share the work



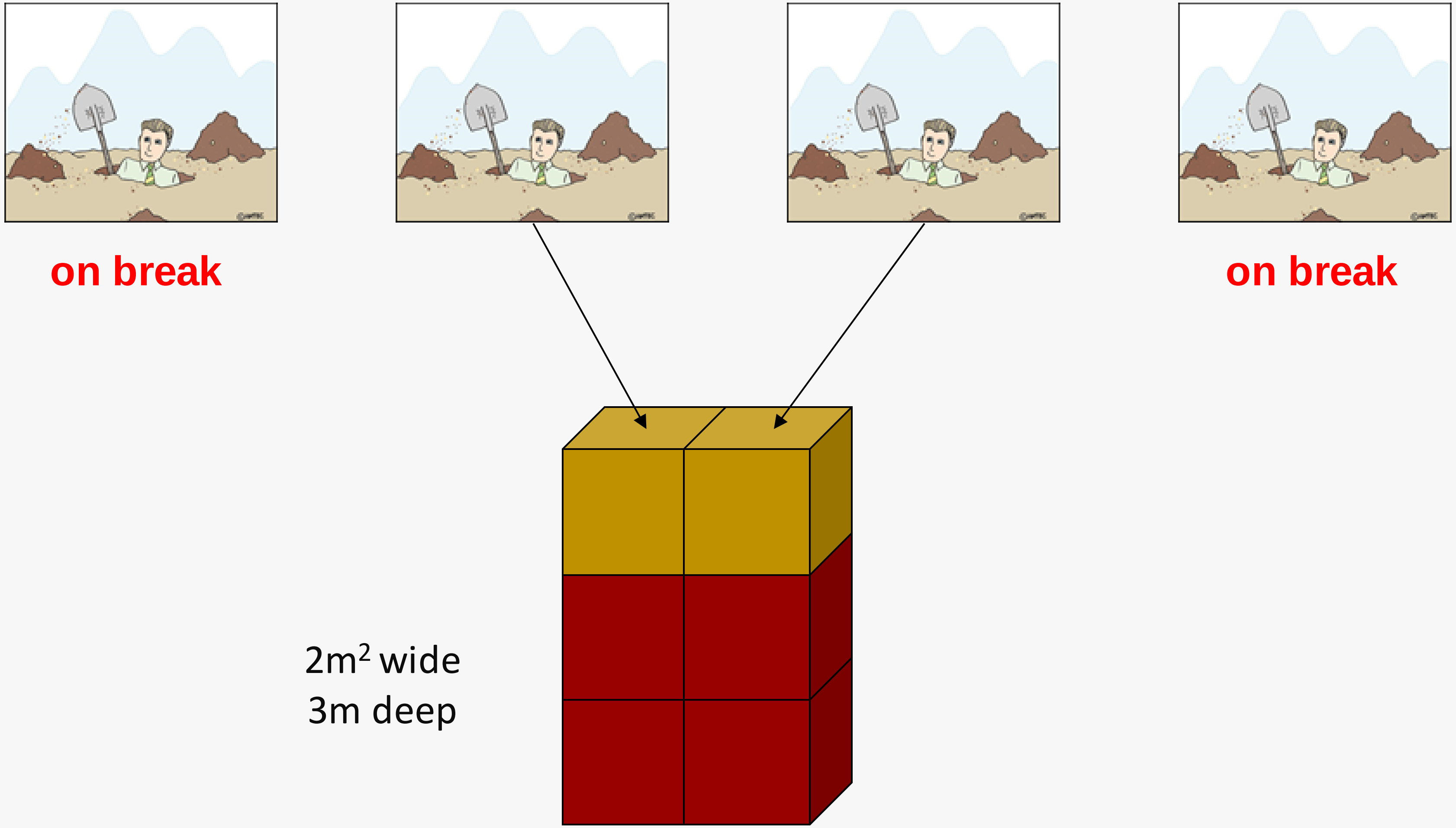
8m^2 wide
3m deep

Now say the hole $2m^2$ wide

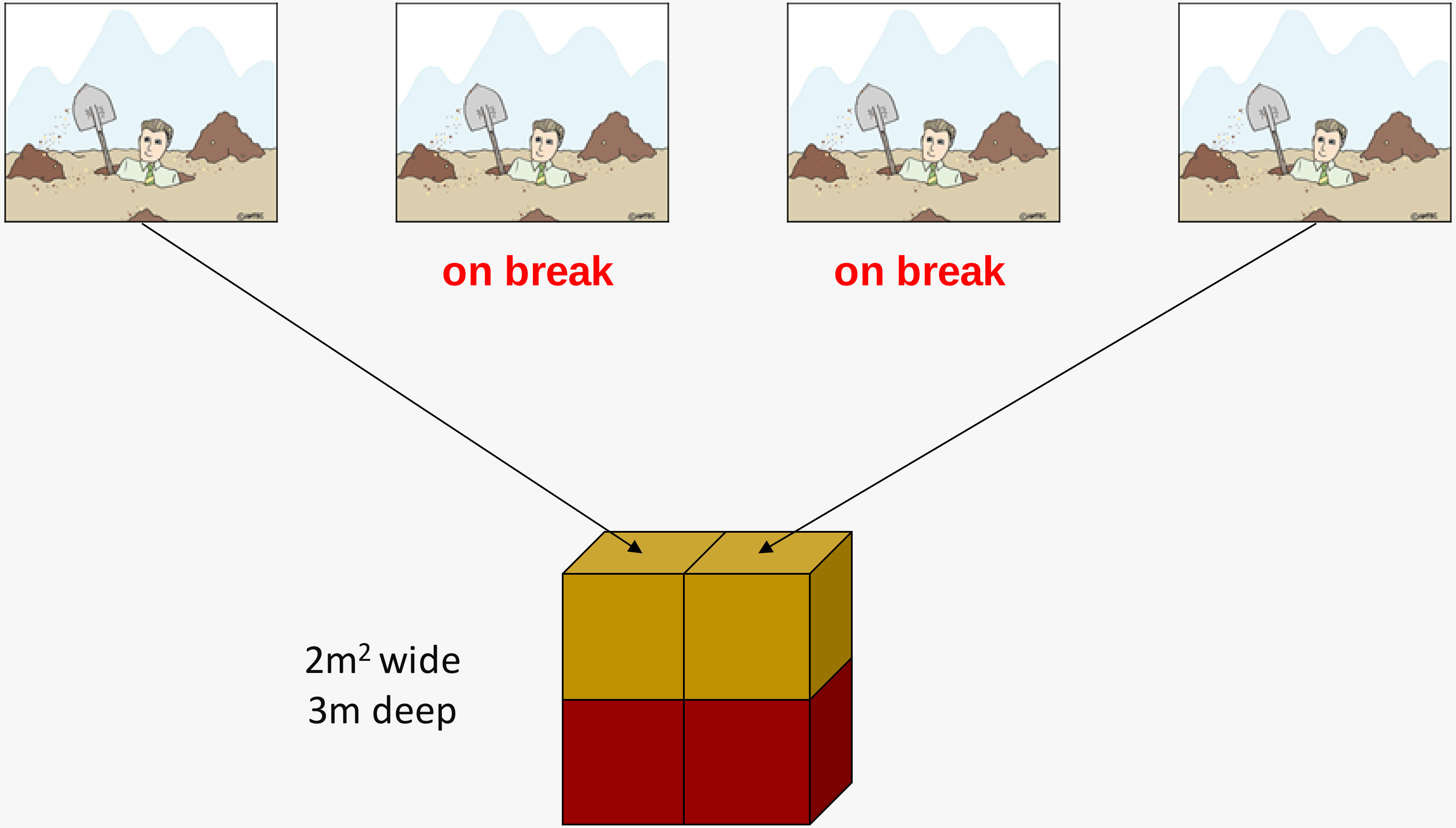
$2m^2$ wide
3m deep



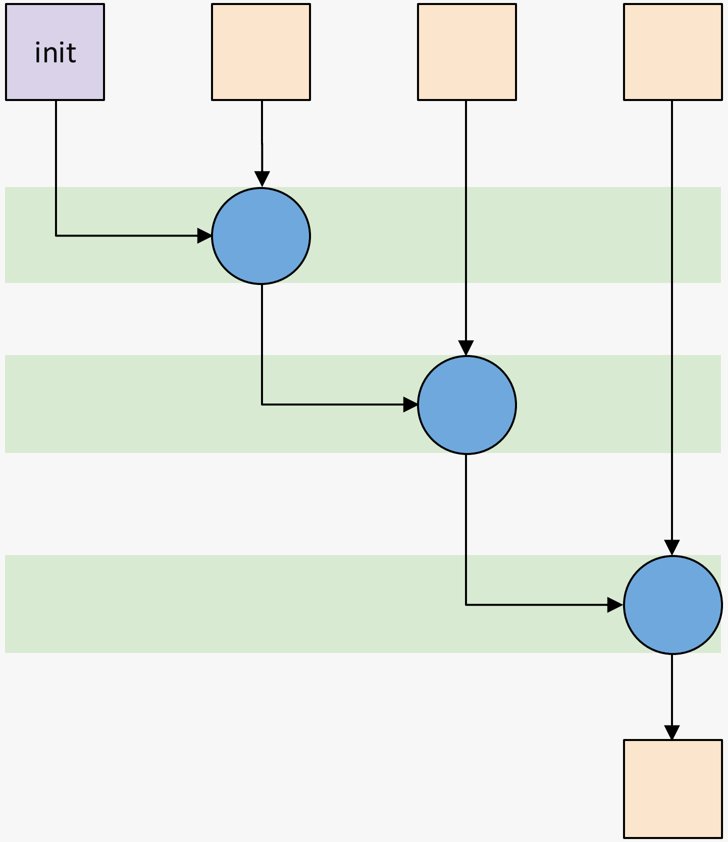
Now you have diggers with no work to do



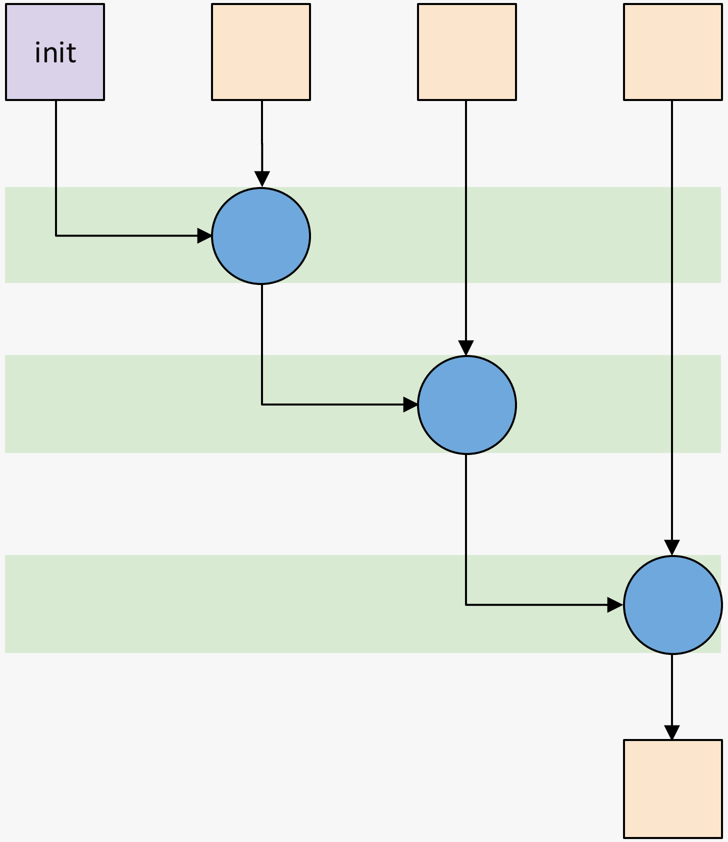
Now you have diggers with no work to do



This applies to a reduction algorithm

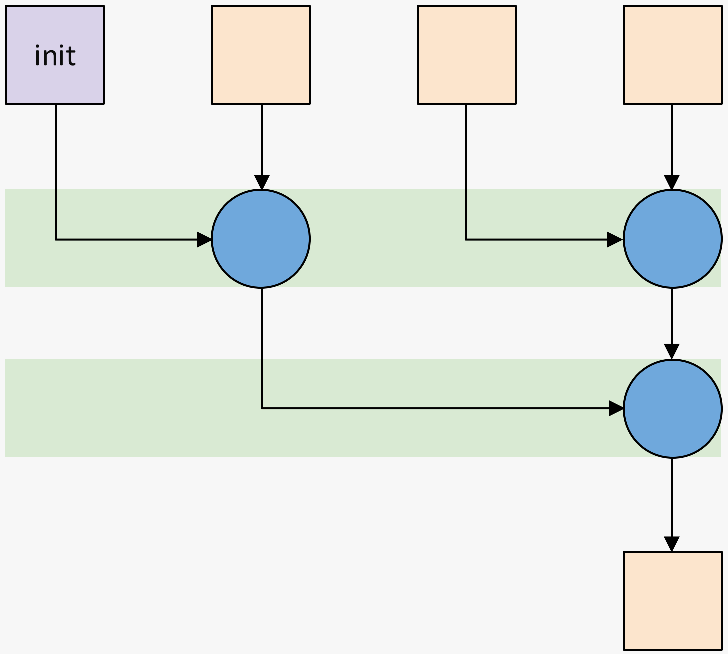


Let’s look at a serial reduction...



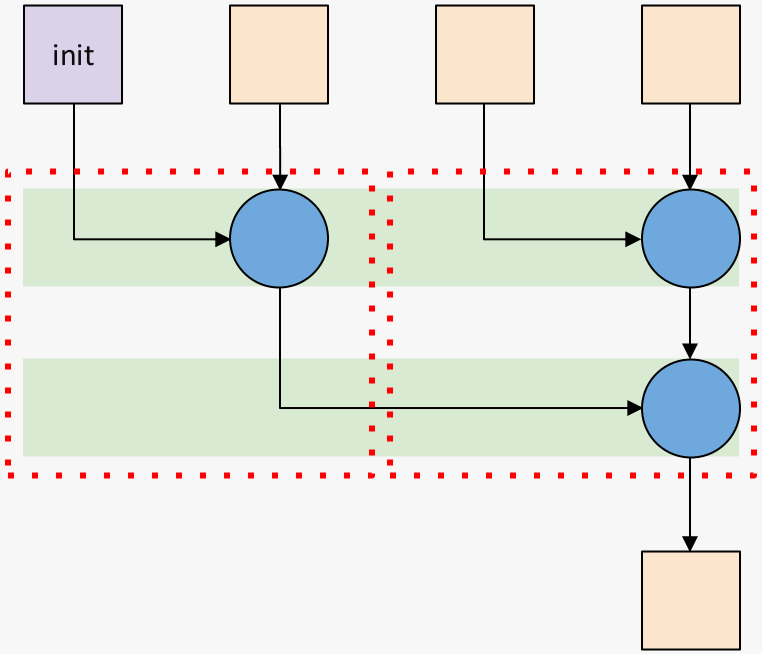
3 elements | 3 Operations| 3 steps | 1 operations / step

Now let’s look at a parallel reduction...



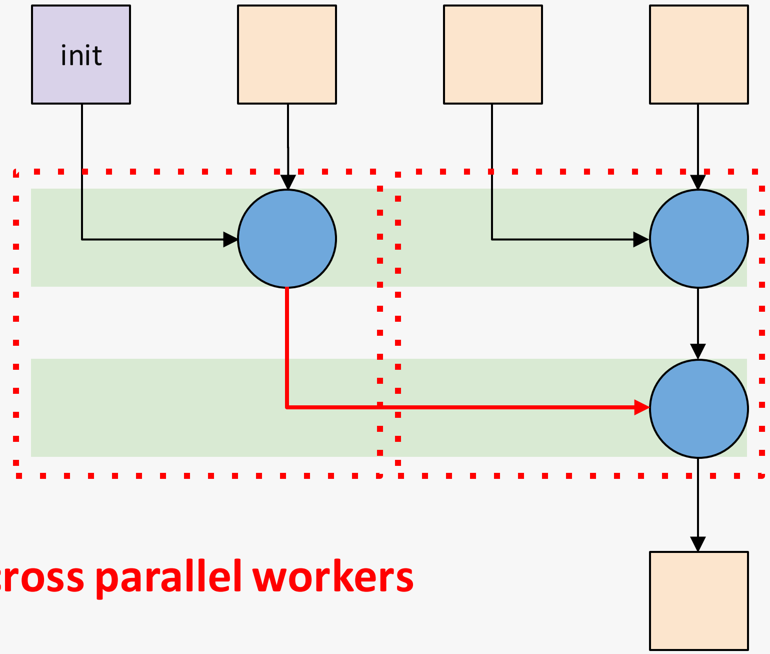
3 elements | 3 Operations| **2 steps** | **1.5 operations / step**

Let’s try to distribute this work



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

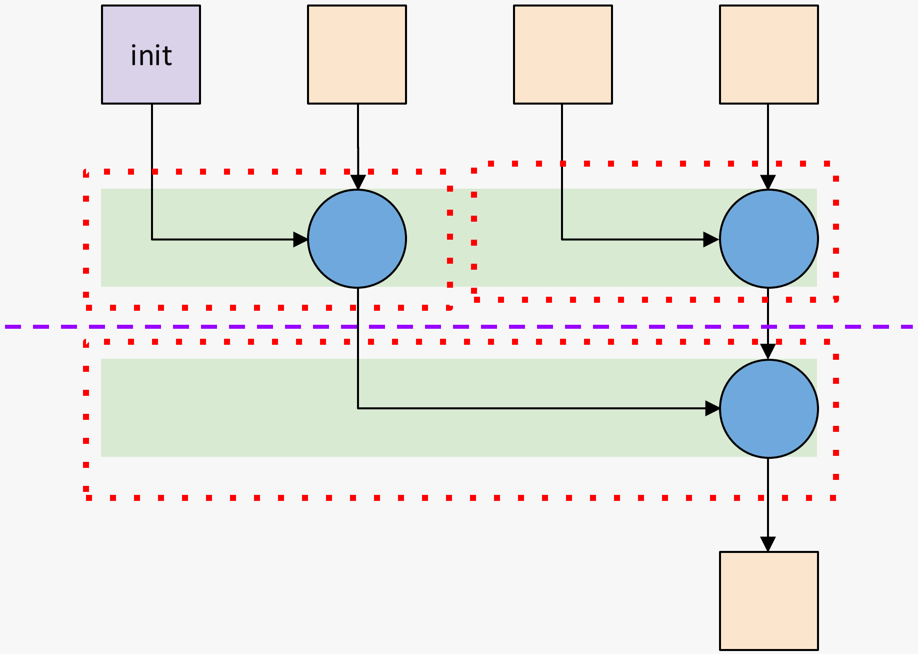
This creates a dependency between workers



Dependency across parallel workers

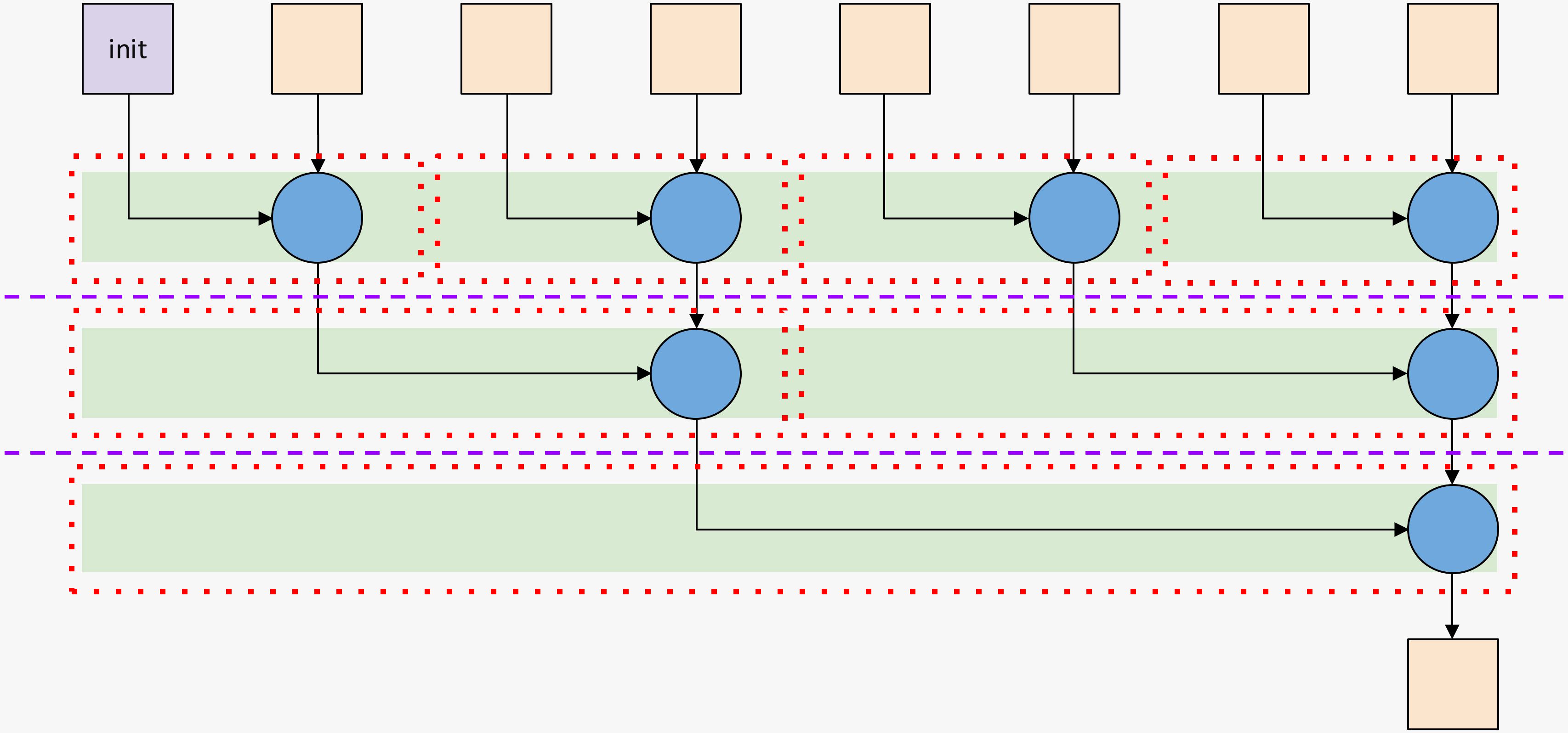
3 elements | 3 Operations | 2 workers | 2 steps | 1.5 operations / step

This creates a dependency between workers



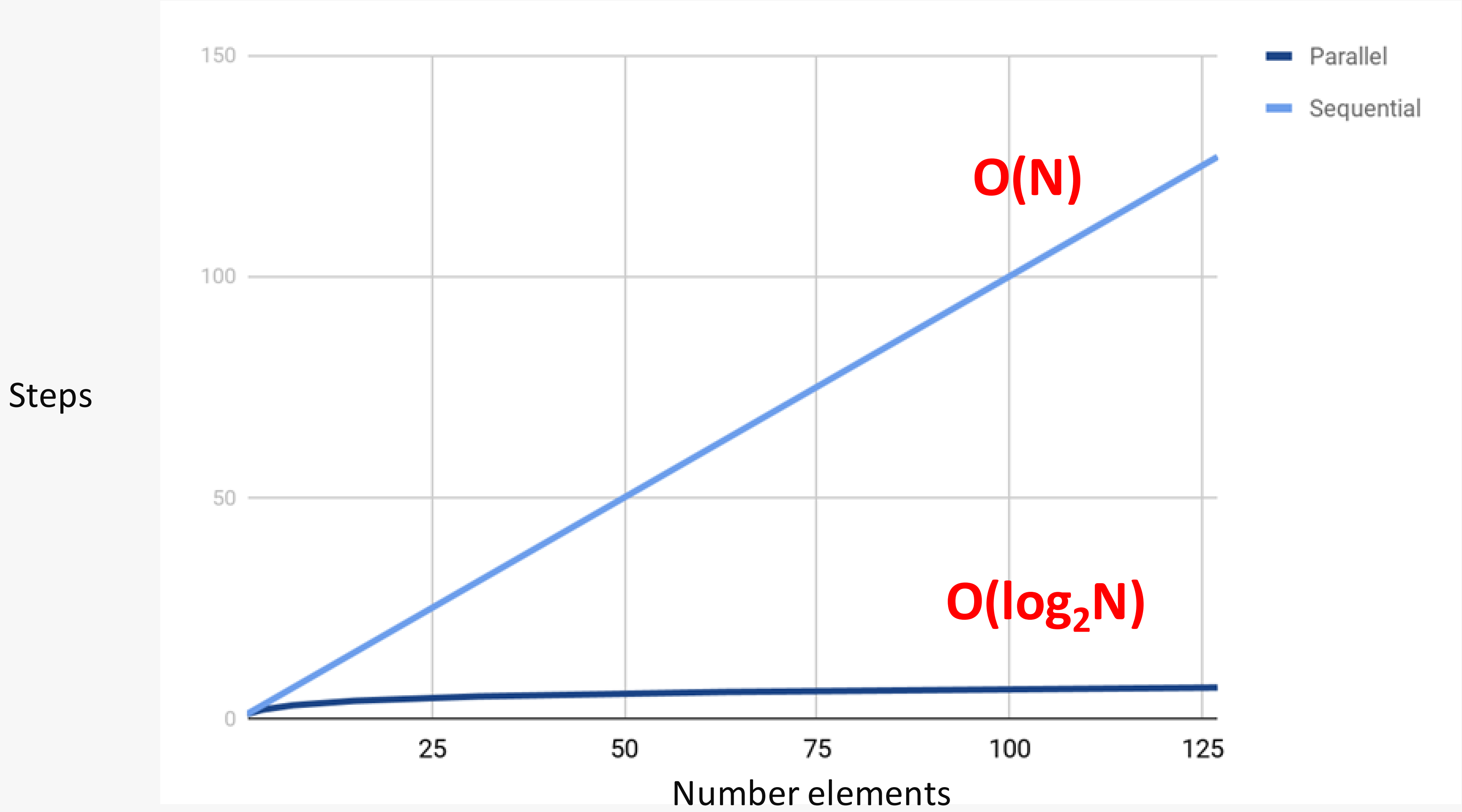
3 elements | 3 Operations| 2 workers | 2 steps | 1.5 operations / step

Now let's scale this up for 4 workers

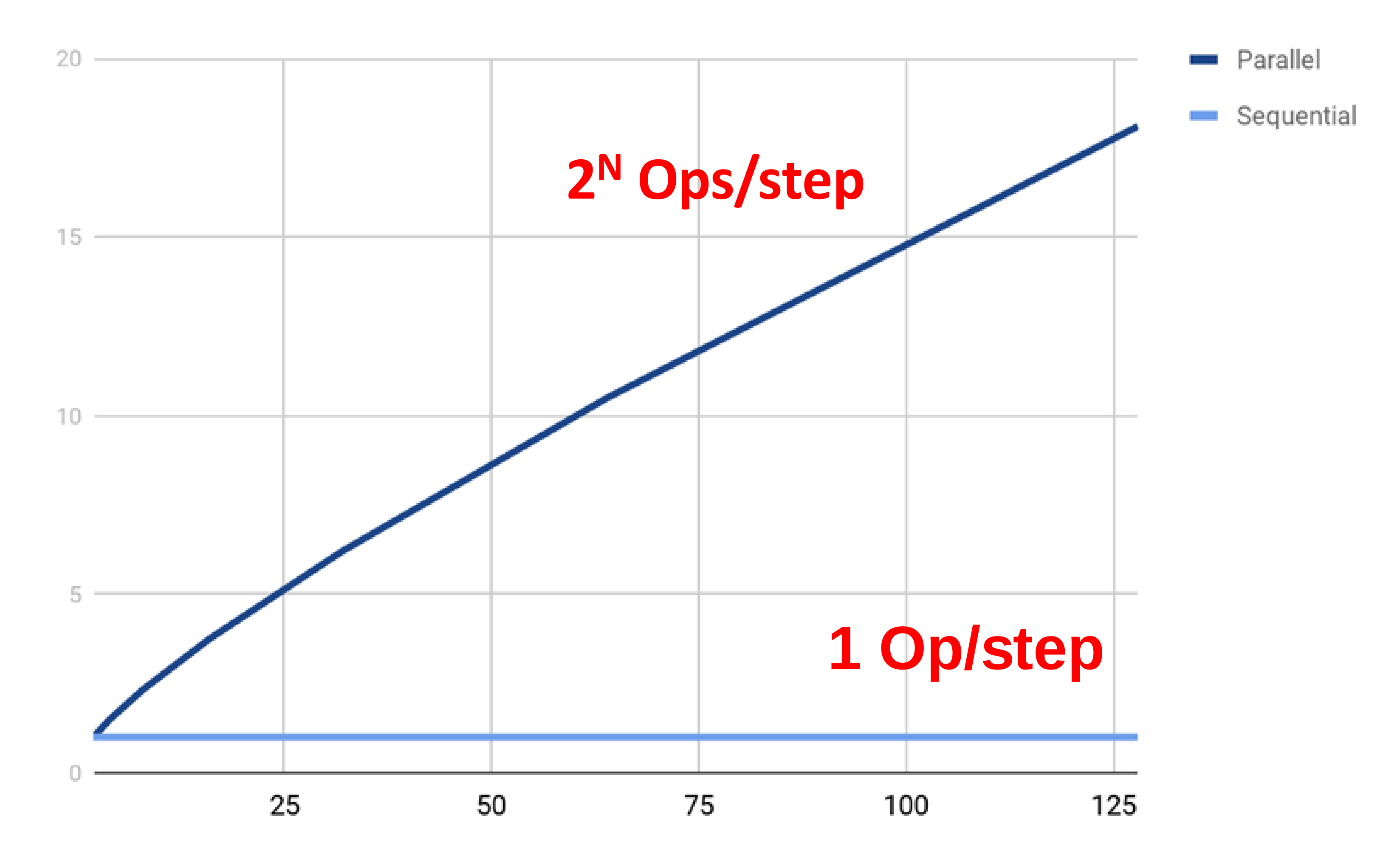


7 elements | 7 Operations| 4 workers | 3 steps | 2.3 operations / step

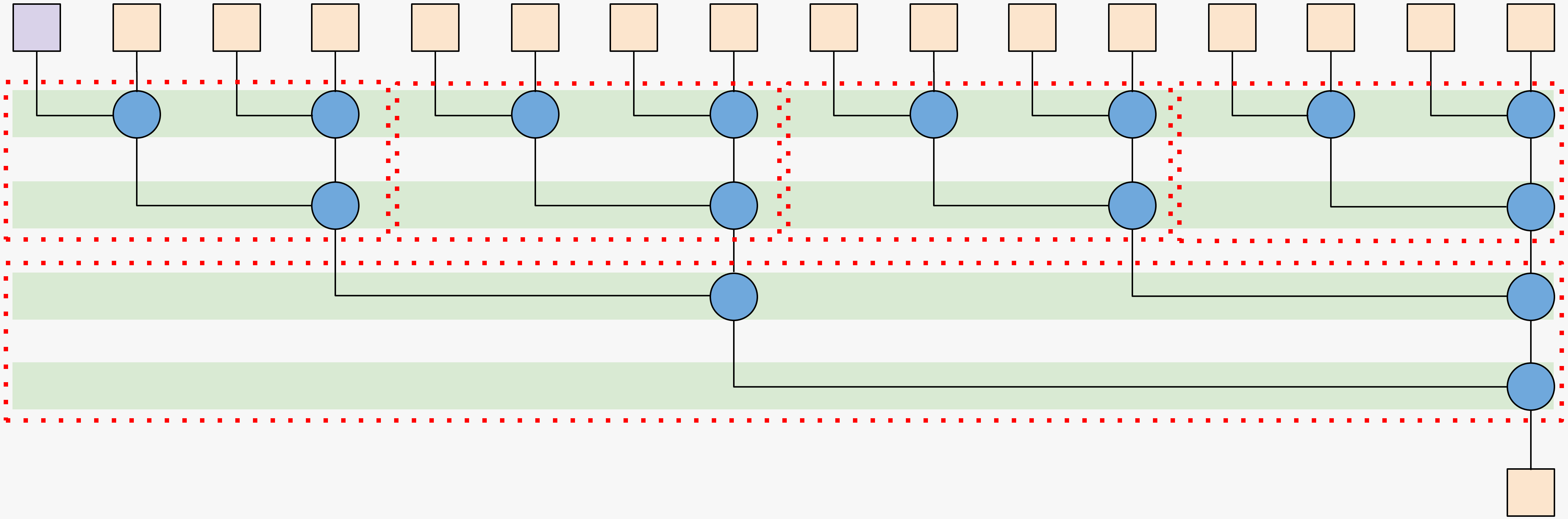
Step complexity of reduce



Theoretical operations per step

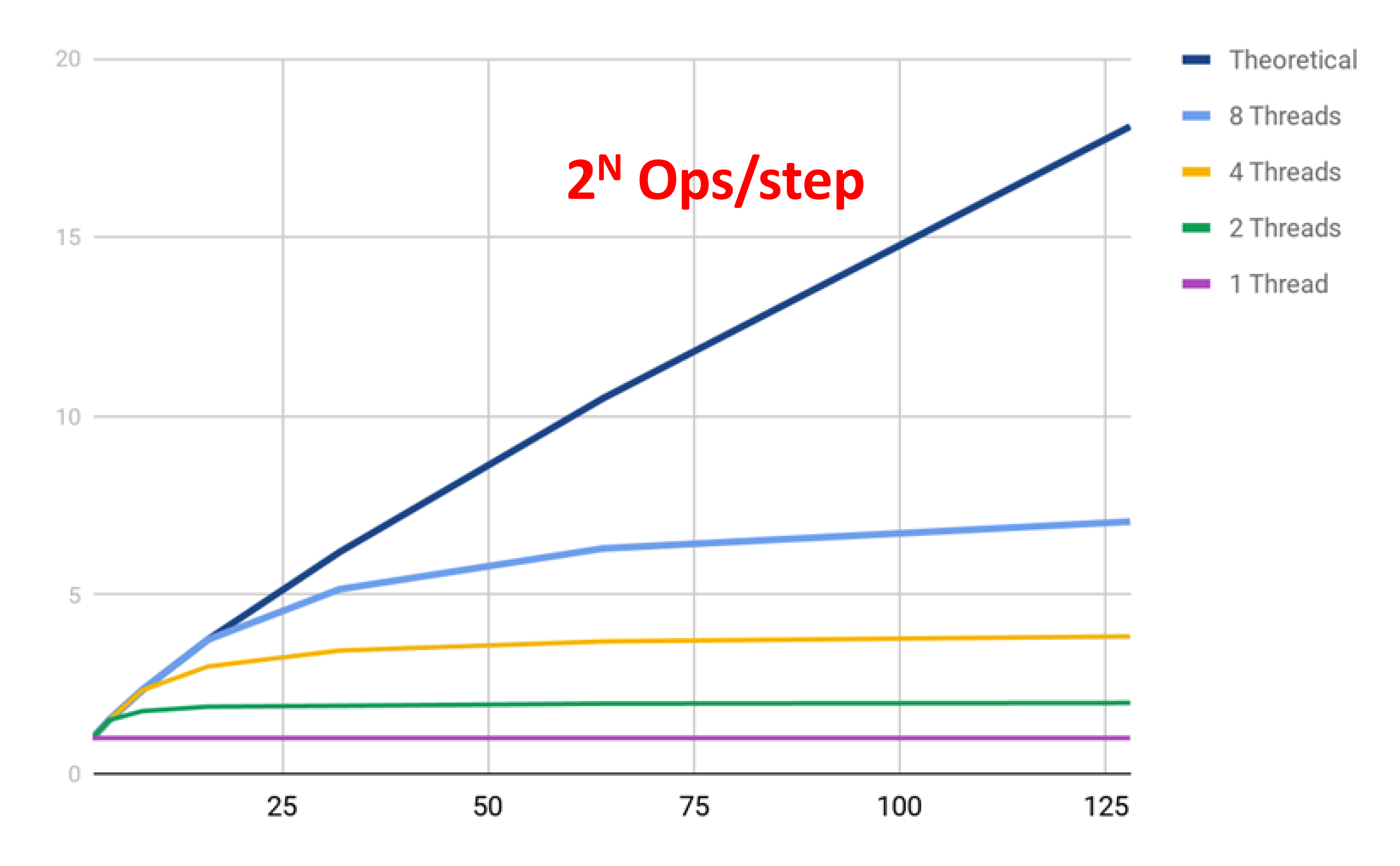


Now let’s scale this up for 4 workers



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

Key takeaways

Designing your algorithm to maximize the operations per step will improve throughput and utilization 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 codeplay®

THE HETEROGENEOUS SYSTEMS EXPERTS

Questions?