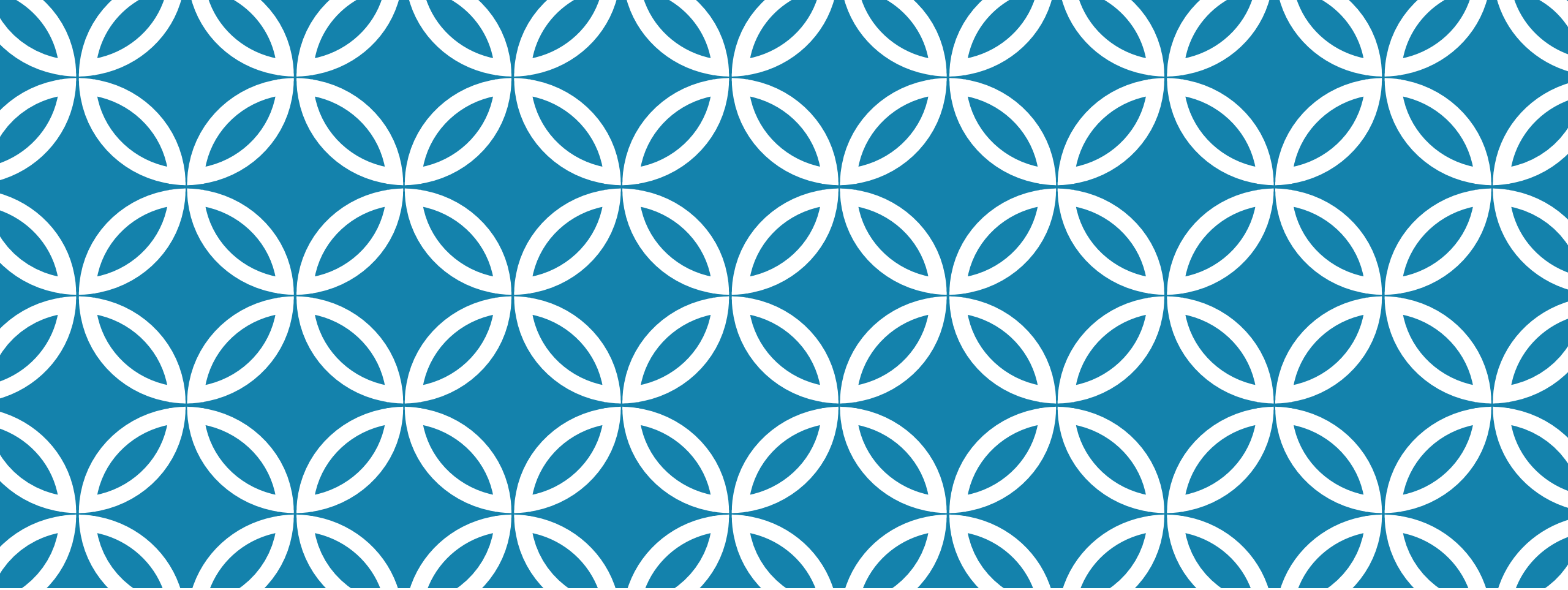


COMPUTING ON COLLECTIONS OF OBJECTS

Ken Birman

CS4414/5416 Lecture 11

KEY IDEAS FOR TODAY

Cloud-hosted applications often end up managing huge numbers of objects or files.

But Linux emerged as an O/S focused on systems with relatively smallish numbers of files

Our focus today is on where objects come from, how storage servers host large object collections, and how applications query large object collections. We will even touch on one case that is often used in distributed settings, although it makes sense even in a single process using threads

WE USED TO ALWAYS KEEP "COLLECTIONS" OF DATA IN DATABASES

But today, MLs train on real world data that can include every conceivable media and data type. Small files might feel more natural, for example if we have a million emails, or fifty thousand examples of properly filled-out W4 forms.

Sometimes, the very first task is to just figure out ("classify") the input!

Yet ML training and RAG ML knowledge retrieval systems need a more regular and predictable way to deal with data

STRUCTURED AND UNSTRUCTURED DATA

It is common to say that cloud data is **structured** or **unstructured**.

Unstructured data means web pages, photos, or other kinds of content that isn't organized into some kind of table.

Structured data means "a table" with a regular structure, or a list of items in a similar format such as (key,value) tuples in a collection

A TABLE IS A STRUCTURED FORM OF DATA

Cow Name	Weight	Age	Sex	Milking?
Bessie	375kg	4	F	Y
Sally	480kg	3	M	Y
Clover		2	F	N
Daisy	411kg	5	F	Y
…				

Even so, notice that this table has an error: Sally isn't a male cow. "Milking" should be N. And we are missing weight data for Clover.

Often the first step is to clean up missing data, visibly incorrect data, etc.

STRUCTURED AND UNSTRUCTURED DATA

There are many tools to convert unstructured data to structured data.

For example, we can take a photo and extract the photo meta-data. This would initially be a list of (key,value) pairs. The values would be byte arrays

If we deserialize the values, we obtain some form of structure, and the fields in the structure become the “columns” in our row

STRUCTURED AND UNSTRUCTURED DATA

Another example with a photo collection.

We could take a set of photos and *segment* them to outline the objects in the image: fences, plants, cows, dogs, etc.

Then we can tag the objects: this is Bessie the cow, that is Scruffy the dog, over there is the milking barn. And finally, we could make one table per photo with a row for each of the tagged objects within the photo.

AUTOMATED PIPELINES

In fact the big cloud companies have huge automated pipelines that do exactly this task.

Photos are uploaded into, say, Facebook. Then in big batches they are auto-segmented, tagged to identify the people, and this in turn allows them to repost to the feeds of friends of those people.

AUTOMATED PIPELINES

Notice that the sequence would have a database query in it: first, the people in a photo are often friends of the person who uploaded it.

… so the autotagger would want a list of those friends as an input.

Then the autotagger would probably want to find prior photos of those individuals: a second query that returns a list of photos.

A PHOTO AND ITS META-DATA

TAG	VALUE	ADDITIONAL_VALUE
GPS	42°26'26.27" N -76°29'47.80"	DMS
Cow	Bessie	Object #3
Cow	Daisy	Object #4
Dog	Scruffy	Object #5
DATETIME	Jan 15, 2020	10:18.25.821
Bldg	Milking shed	Object #8
Man	Farmer Jim	Object #71
Bldg	Farm House	Object #2
Vehicle	Tractor	Object #33

NOTICE THAT THE META DATA HAD MORE THAN ONE SOURCE

Some meta-data fields were put there by the camera, but other applications could add more tags

Here, some were added by photo analysis applications. The extra meta-data includes information about a series of objects identified by some sort of computer vision software.

Each type of meta-data would have its own columns.

JSON FILES

The cloud has a standard way of representing things like tags, in a file format called JSON.

```
{"widget": {
    "debug": "on",
    "window": {
        "title": "Sample Konfabulator Widget",
        "name": "main_window",
        "width": 500,
        "height": 500
    },
    "image": {
        "src": "Images/Sun.png",
        "name": "sun1",
        "hOffset": 250,
        "vOffset": 250,
        "alignment": "center"
    },
    "text": {
        "data": "Click Here",
        "size": 36,
        "style": "bold",
        "name": "text1",
        "hOffset": 250,
        "vOffset": 100,
        "alignment": "center",
        "onMouseUp": "sun1.opacity = (sun1.opacity / 100) * 90;"
    }
}}
```

It looks like a web page, with text fields, “field” : “value”

These can nest, using a simple bracket notation.

STRUCTURED AND UNSTRUCTURED DATA

Now, imagine that we actually had many photos

We could do this same process photo by photo.

We end up with one row per photo. The photo name or id is just one more column!

WHERE SHOULD THE PIPELINE PUT THE EXTRACTED INFORMATION?

In some systems we extract into a database (a good option).

Small files can feel natural but can overload Linux (the directory structure is inefficient if one directory might have a really huge number of files in it)

A modern and popular option is to use a **key-value storage layer.**

Common products of this kind include DynamoDB (AWS), CosmosDB (Azure), Reddis, and there are many more.

KEY-VALUE STORAGE

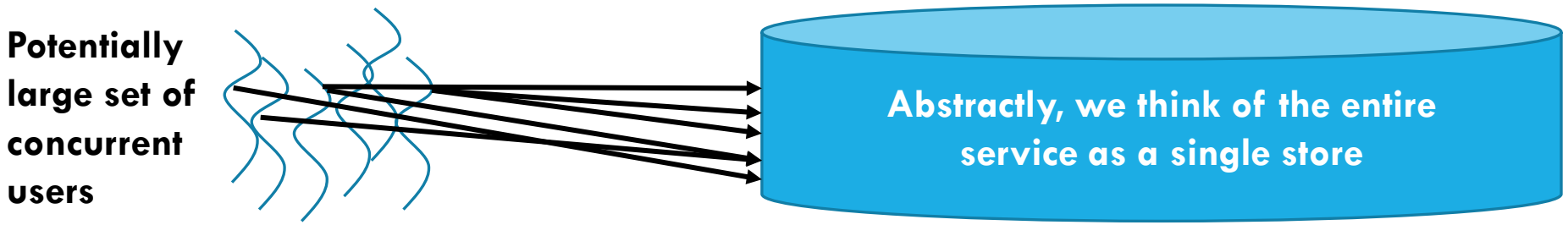
These have a very simple interface:

- **put(KT key, VT value)**
- **VT v = get(KT key)**

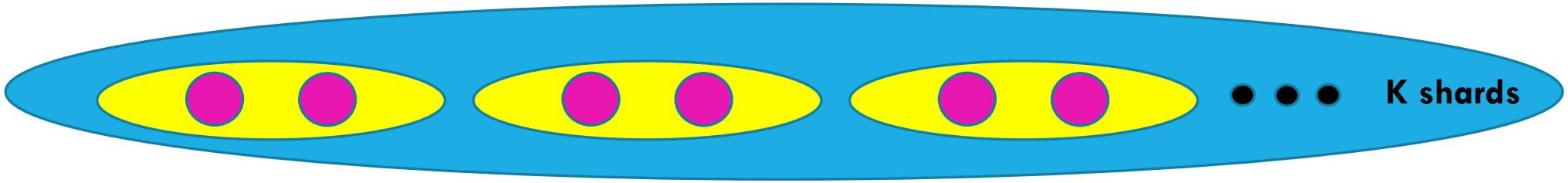
They are automatically sharded for scalability. Often the KT and VT are just byte vectors or strings, and "hashing" is used to convert to a pseudo-random integer. Then, by taking it modulo the number of shards, we know which shard will handle this (key, value) tuple.

SHARDED K/V STORE: MAIN IDEAS

Our data consists of files or K/V objects

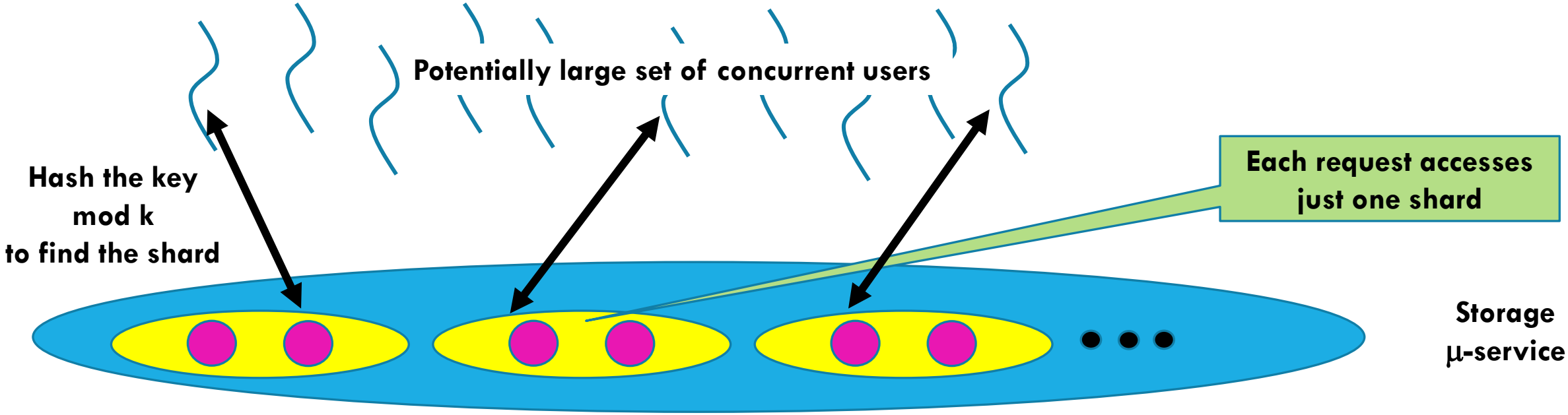


The K/V service is just one of the many vendor-supplied microservices in the ecosystem



Sharding breaks the single K/V store into chunks, each with 1/k'th of the data

SHARDS HAVE 2 (OR MORE) NODES!



The pairs of nodes here represent the idea of **replication**. Both of the purple nodes are individual servers, on different computers.

We use ***state machine replication*** to keep them in sync.

WHAT'S IN A KEY?

Are K/V stores file systems?

Often a string that "looks like" a full file system path

- The pathnames can include directory names, offering an easy way to "organize" large K/V data collections
  - ◊ This works even though the directories don't "exist"!
- Using string patterns it is easy to select only the objects in some "directory", or just the ones with names (keys) matching a pattern

PROGRAMMING LANGUAGE CONCEPTS

A collection is a list of elements with some shared type that your code is focused upon. You **bind** to a key-value store (like a form of file open) and if successful, end up with a **connector.**

Given a connector, your code can now **iterate** over the collection.

No connector will show you aren't allowed to access, but fancy connectors can additionally select only objects with keys matching a pattern you supply.

ITERATOR OBJECTS

An iterator object represents some portion of the collection.

It has a **begin** point, a **next** operator, and an **end** point.

You can iterate a range within some list, or iterate in reverse order…

EXAMPLES

My photo meta-data was a table, but I can think of it as a collection of rows, one row per meta-data item.

Every row always has a unique row key. The value is the row contents: a struct or array or an object with one field per column.

To scan the full row, a for loop will begin with the first row and scan to the last row: **begin… next… next…. End.**

EXAMPLES

You would see code like this in C++:

```
for(auto row = table.begin(); row = row.next(); row != table.end())
{
    do something with this row
}
```

EXAMPLES

You would see code like this in C++:

```
for(auto row: table)
{
        do something with this row
}
```

CONNECTORS TO K/V STORES HAVE BUILT-IN ITERATORS FOR MOST LANGUAGES

This lets you

- Bind, at which point permissions are checked
- Select some collection of K/V data you are allowed to access (many connectors include a way to filter at this stage)

- But then can do more filtering in your code, in any language!

EXAMPLES

You would see code like this in C++:

```
for(auto row: table)
{
    if(row.cow_id = 1471)
    {
        do something with this row
    }
}
```

BUT WE CAN DO EVEN BETTER!

Many languages have built-in layers that look like database query languages!

C++ has one too, but it is awkward because of compile-time type rules. So Python, Java, and other languages with runtime types are actually more powerful for this style of coding.

In Microsoft Azure there is a language like C++ or Java called C#, and it has a layer called LINQ for this purpose.

LINQ EXAMPLES

Notice that cpplink has methods much like SQL operators

But you supply lambda methods that shape the way the operators behave

utilForever/**CppLinq**

```
#include <iostream>
#include <vector>
#include "cpplinq.hpp"
int main() {
   std::vector<int> numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
   auto result = cpplinq::from(numbers)
                 .where([](int n) { return n % 2 == 0; })
                 .select([](int n) { return n * n; })
                 .to_vector();
   for (auto n : result) {
      std::cout << n << " ";
   }
   return 0;
}
```

Visit Microsoft LINQ for .NET documentation to learn more

LINQ EXAMPLES

Things to notice:

- Code is very "succinct"

- Lots of use of lambdas

- On Microsoft platforms, LINQ is available in 44 languages!

- C# version even allows use of reflection to optimize the queries

Double the odd numbers, then keep those in the range [3,11]:

```
int src[] = {1, 2, 3, 4, 5, 6, 7, 8};
auto dst = from(src)
        .where( [](int a) { return a % 2 == 1; })  // 1, 3, 5, 7
        .select([](int a) { return a * 2; })              // 2, 6, 10, 14
        .where( [](int a) { return a > 2 && a < 12; }) // 6, 10
        .toStdVector();     // dst will be a std::vector with 6, 10
```

Order descending all the distinct numbers from an array of integers, transform them into strings and print the result.

```
int numbers[] = {3, 1, 4, 1, 5, 9, 2, 6};
auto result = from(numbers)
          . distinct()
          . orderby_descending([](int i) {return i;})
          . select([](int i){std::stringstream s; s<<i; return s.str();})
          . toStdVector();
for(auto i : result)
    std::cout << i << std::endl;
```

Visit Microsoft LINQ for .NET documentation to learn more

EXAMPLE WITH STRUCTS

In a list of friends, find the subset who are under age 18, order them by age, then return their names.

```
struct Friends { std::string name; int age; };

Friends src[] = {
   {“Kevin”, 14}, {“Anton”, 18}, {“Agata”, 17}, “Saman”, 20}, {“Alice”, 15},
};

auto dst = from(src).where([](const Friends & who) { return who.age < 18; })
                    .orderBy([](const Friends & who) { return who.age; })
                    .select( [](const Friends & who) { return who.name; })
                    .toStdVector();

// dst type: std::vector<:string>…  items: “Kevin”, “Alice”, “Agata”
```

Visit Microsoft LINQ for .NET documentation to learn more

EXAMPLE WITH STRINGS

In a list of text messages, count the number of messages to Dennis by sender:

```
struct Message {  std::string PhoneA; std::string PhoneB;  std::string Text;  };

Message messages[] = {
    {“Anton”,  “Troll”, “Hello, friend!”},
    {“Denis”,  “Mark”,  “Join us to watch the game?"},
    {“Anton”,  “Sarah”, “OMG! ”},
    {“Denis”,  “Jimmy", “How r u?”},
    {“Denis”,  “Mark",  “The night is young!”},
};

int DenisUniqueContactCount =
    from(messages)
        .where([](const Message & msg) { return msg.PhoneA == “Denis”; })
        .distinct([](const Message & msg) { return msg.PhoneB; })
        .count();
```

Visit Microsoft LINQ for .NET documentation to learn more

WORD EXTRACTION EXAMPLE

This is a case you might use when scanning emails or similar documents to extract fields from them or phrasing

Then could use that data to train an LLM to automatically suggest replies

```
string sentence = "the quick brown fox jumps over the lazy dog";
// Split the string into individual words to create a collection.
string[] words = sentence.Split(' ');

// Using query expression syntax.
var query = from word in words
            group word.ToUpper() by word.Length into gr
            orderby gr.Key
            select new { Length = gr.Key, Words = gr };

// Using method-based query syntax.
var query2 = words.
    GroupBy(w => w.Length, w => w.ToUpper()).
    Select(g => new { Length = g.Key, Words = g }).
    OrderBy(o => o.Length);

foreach (var obj in query)
{
    Console.WriteLine("Words of length {0}:", obj.Length);
    foreach (string word in obj.Words)
        Console.WriteLine(word);
}
```

SOME LINQ OPERATORS. THE FULL LIST HAS MORE!

Filters and reorders:

- `where(predicate)`, `where_i(predicate)`
- `take(count)`, `takeWhile(predicate)`, `takeWhile_i(predicate)`
- `skip(count)`, `skipWhile(predicate)`, `skipWhile_i(predicate)`
- `orderBy()`, `orderBy(transform)`
- `distinct()`, `distinct(transform)`
- `append(items)`, `prepend(items)`
- `concat(linq)`
- `reverse()`
- `cast()`

Transformers:

- `select(transform)`, `select_i(transform)`
- `groupBy(transform)`
- `selectMany(transfom)`

Bits and Bytes:

- `bytes(ByteDirection?)`
- `unbytes(ByteDirection?)`
- `bits(BitsDirection?, BytesDirection?)`
- `unbits(BitsDirection?, BytesDirection?)`

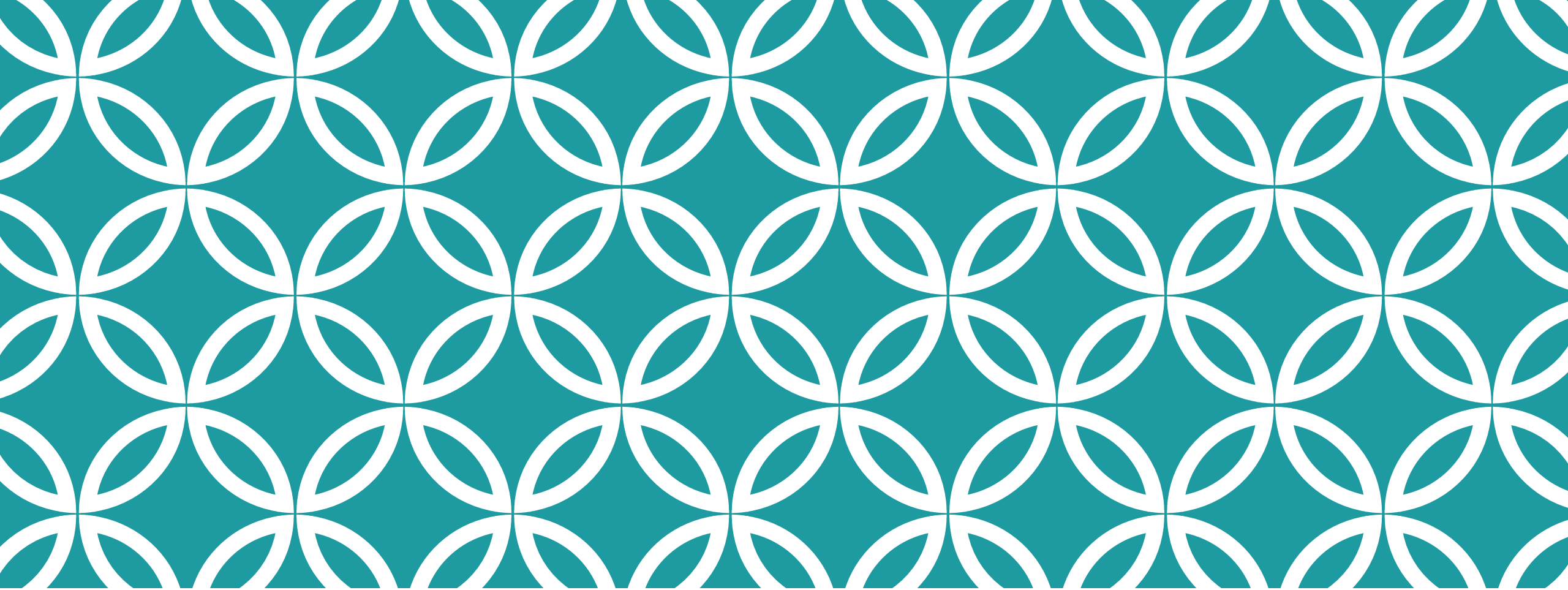
Aggregators:

- `all()`, `all(predicate)`
- `any()`, `any(lambda)`
- `sum()`, `sum()`, `sum(lambda)`
- `avg()`, `avg()`, `avg(lambda)`
- `min()`, `min(lambda)`
- `max()`, `max(lambda)`
- `count()`, `count(value)`, `count(predicate)`
- `contains(value)`
- `elementAt(index)`
- `first()`, `first(filter)`, `firstOrDefault()`, `firstOrDefault(filter)`
- `last()`, `last(filter)`, `lastOrDefault()`, `lastOrDefault(filter)`
- `toStdSet()`, `toStdList()`, `toStdDeque()`, `toStdVector()`

Fancy stuff:

- gz(), ungz(), leftJoin, rightJoin, crossJoin, fullJoin

Visit Microsoft LINQ for .NET documentation to learn more



NOW WE KNOW HOW TO EXTRACT OUR DATA…

How to compute on it at massive scale?

BIG DATA SETS CAN BE REALLY BIG

Many ML systems train or fine tune on huge data sets sharded in a massive K/V store

In such cases they will often assign one worker per shard (or even k per shard) and carry out their ML task in a parallel way

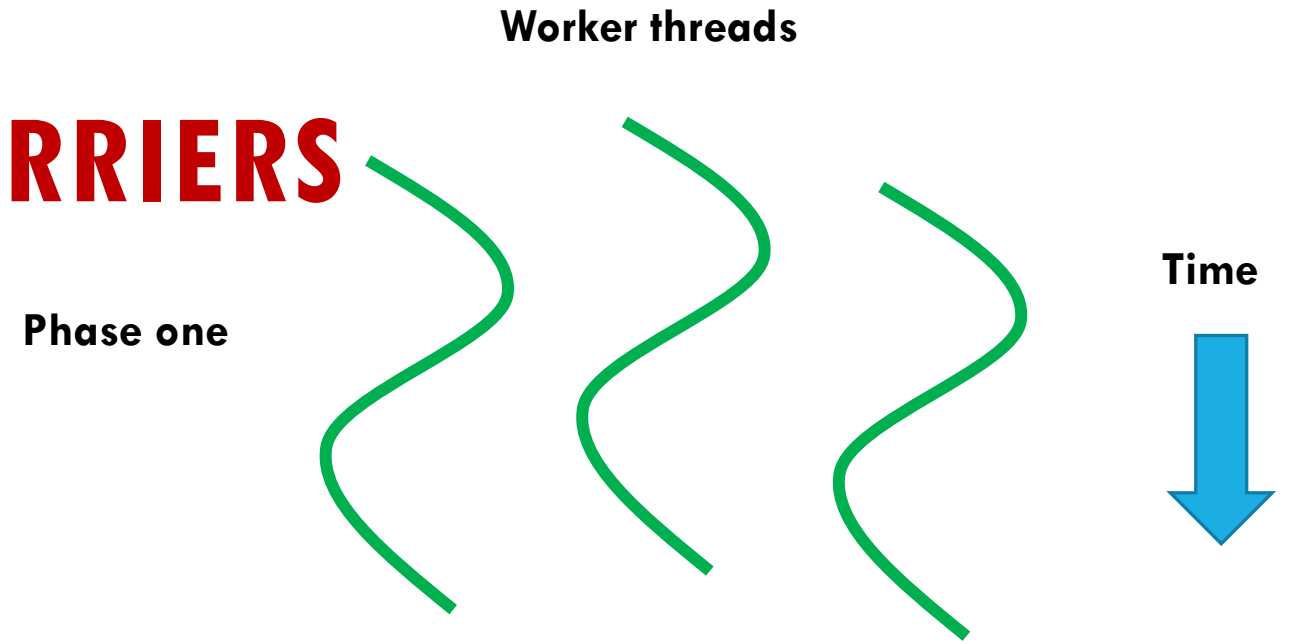
For example, in stochastic gradient descent a leader assigns workers to improve DNN parameters, then merges the gradients, then repeats

… PARALLELISM "PATTERNS"!

Arises in distributed computing of this kind

The "collective communication libraries" such as Open MPI CCL and NVIDIA's NCCL emerged from ML compute on sharded collections

THEY START WITH BARRIERS



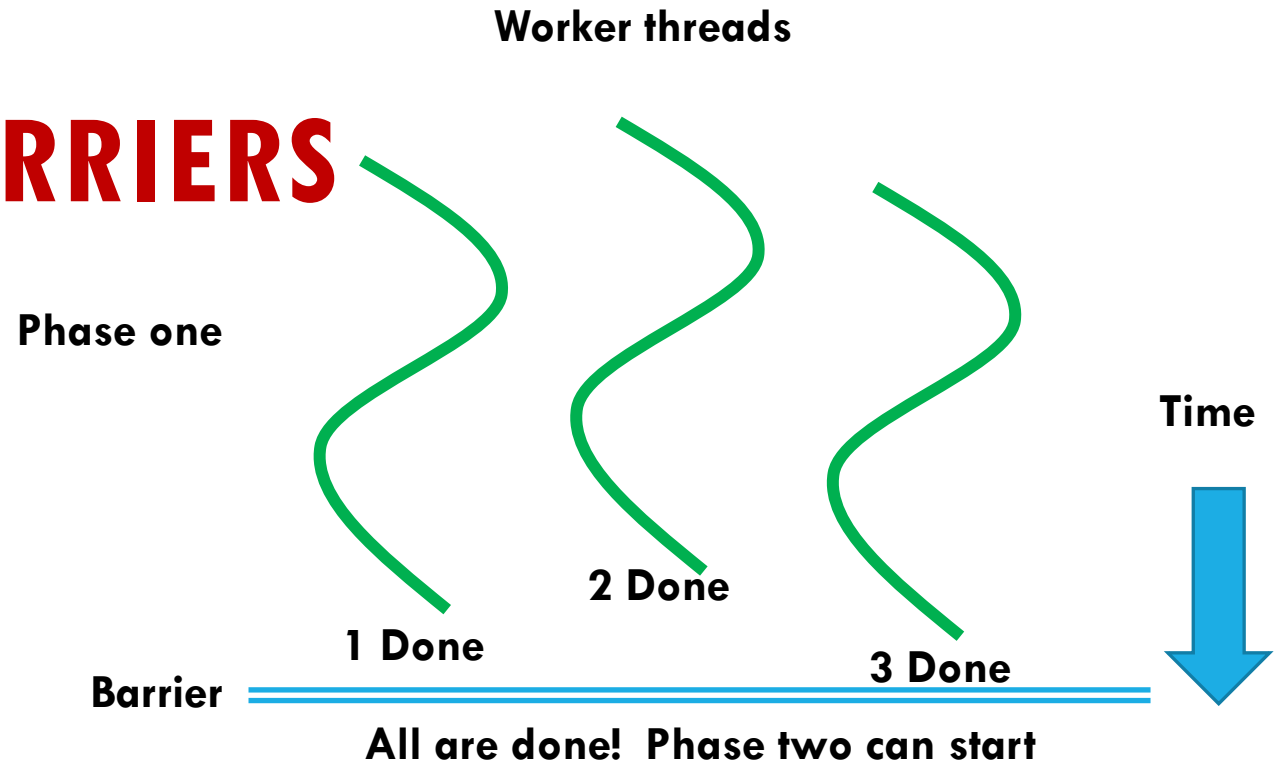
Example: A computation with distinct phases or epochs.

After phase one, all workers must wait until phase two starts.

THEY START WITH BARRIERS

Example: A computation with distinct phases or epochs.

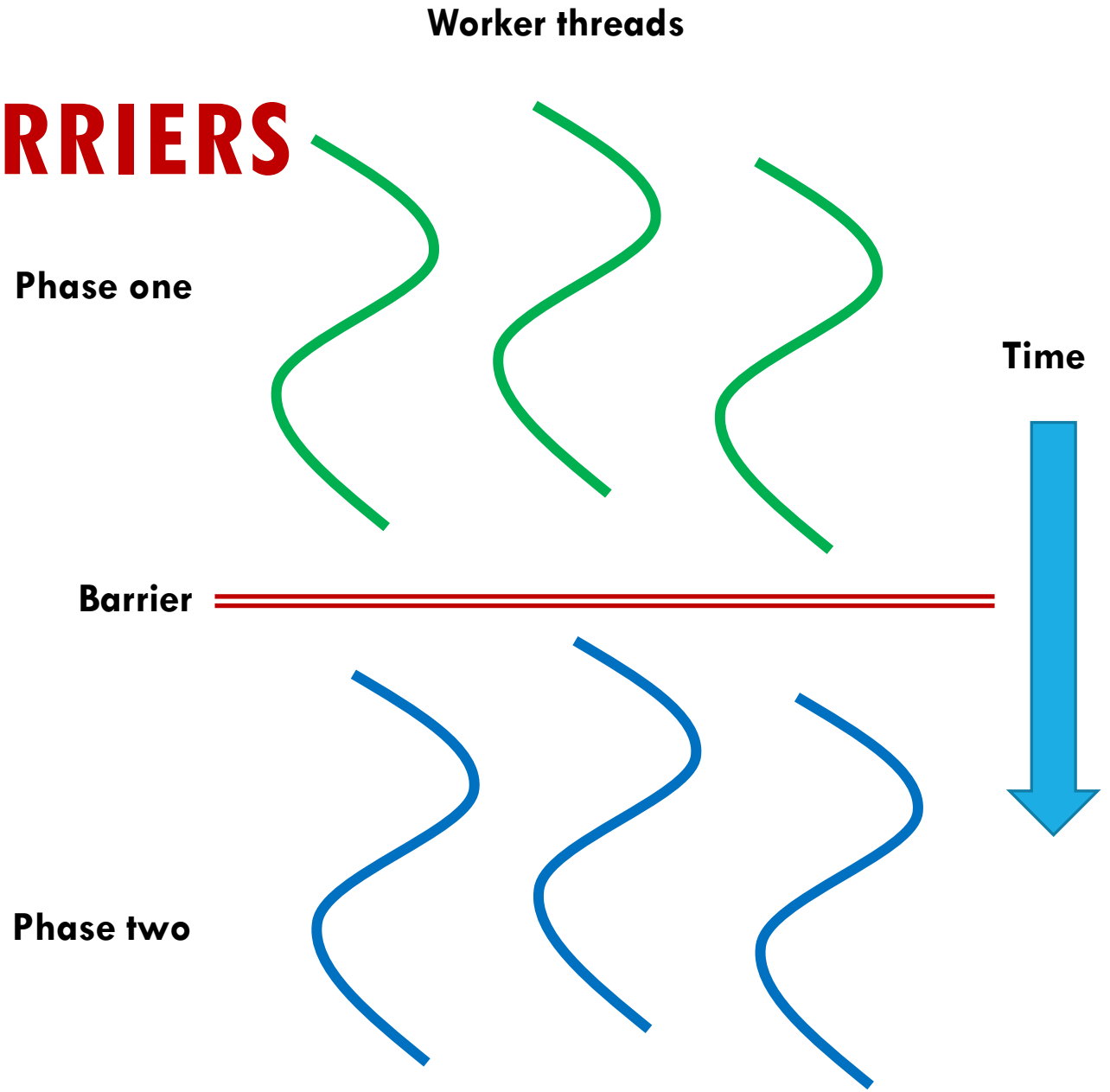
After phase one, all workers must wait until phase two starts.



THEY START WITH BARRIERS

Example: A computation with distinct phases or epochs.

After phase one, all workers must wait until phase two starts.



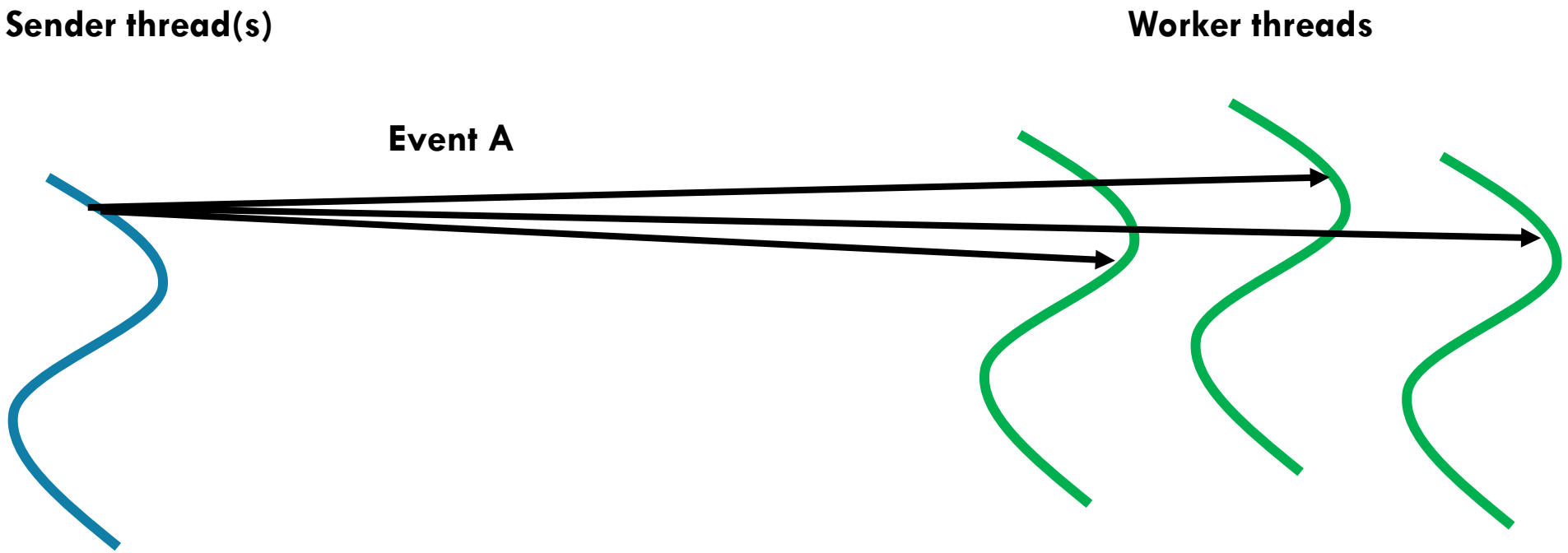
UNORDERED MULTICAST PATTERN

This is a one-to-many pattern. Suppose some event occurs. We sometimes call it broadcast but the meaning is the same.

A sender thread needs every worker to see an object describing the event, so it puts that object on every worker's work queue.

The pattern permits multiple senders: A sender locks all of the work queues, then emplaces the request, then unlocks. Thus all workers see the same ordering of requests.

UNORDERED MULTICAST PATTERN



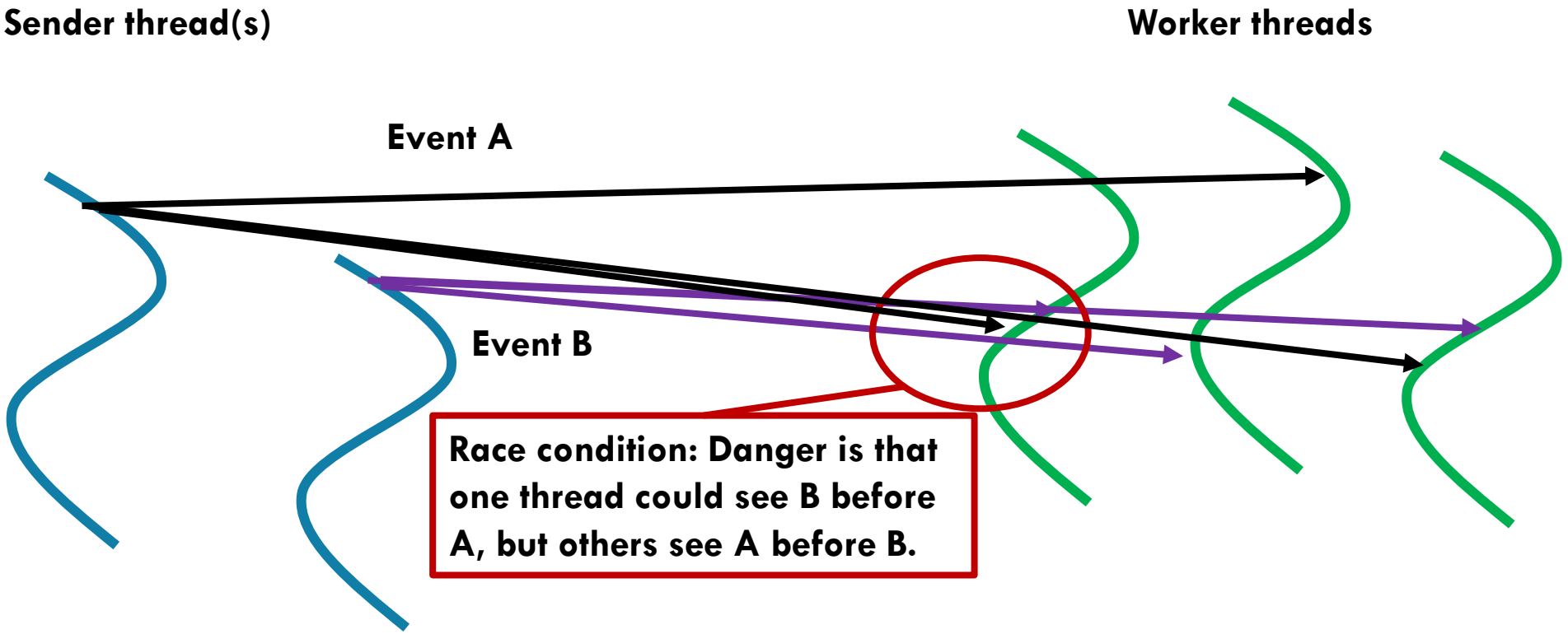
THE NEED FOR "ORDERED" MULTICAST

An unordered multicast is fine if there is only one initiator

But what if a pool of workers might be supporting multiple leaders?

With unordered multicast, they could end up working on different tasks and deadlock!

UNORDERED MULTICAST RISK



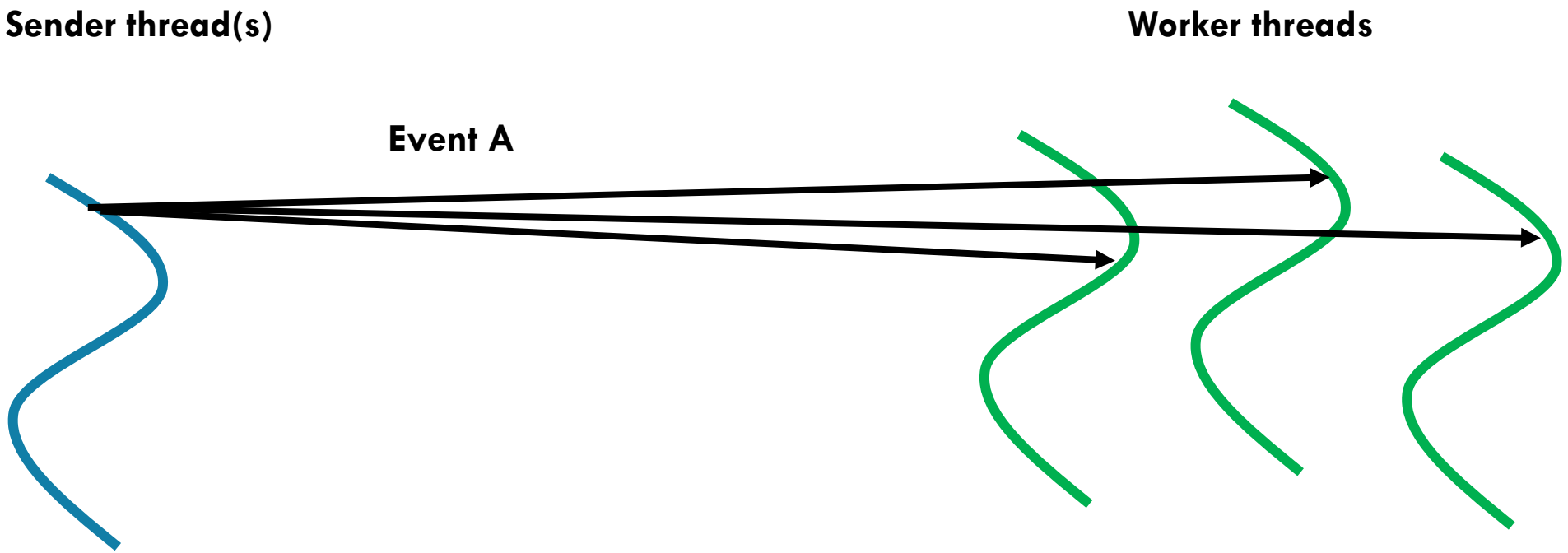
ORDERED MULTICAST PATTERN

This is a one-to-many pattern. Suppose some event occurs. We sometimes call it broadcast but the meaning is the same.

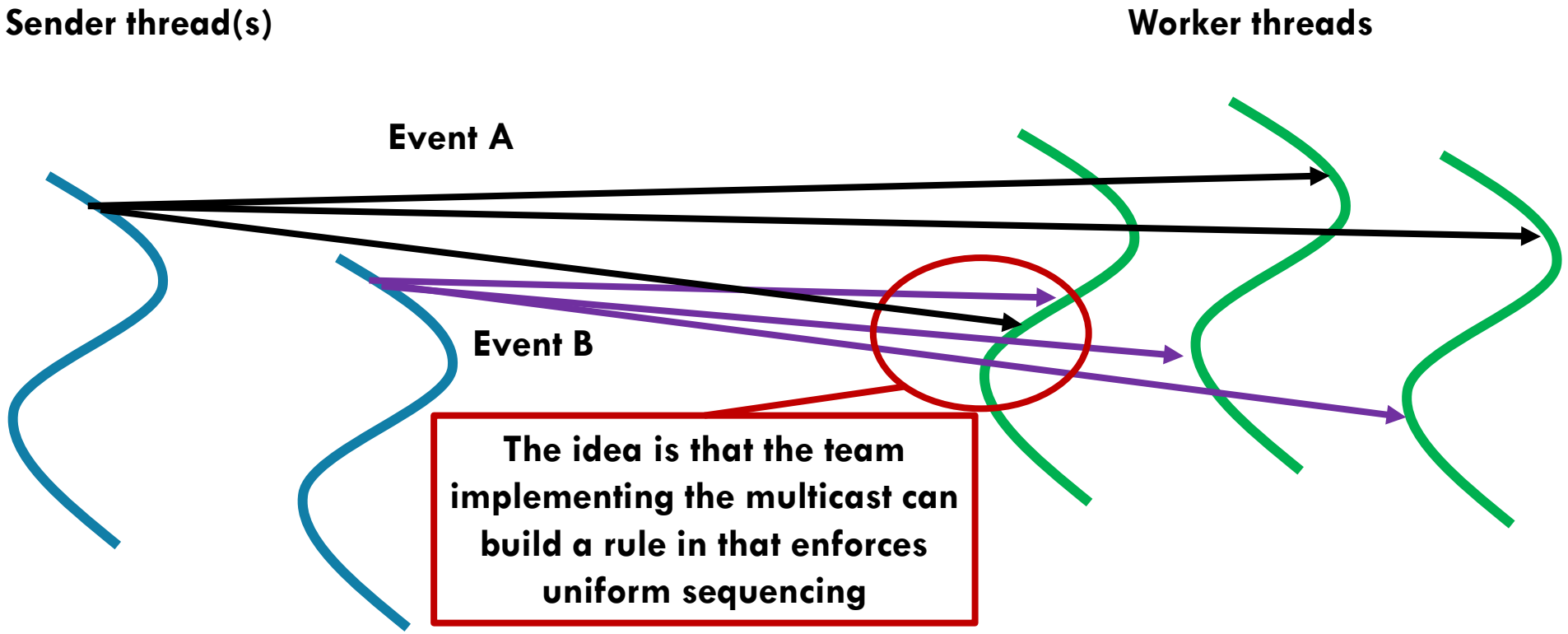
A sender thread needs every worker to see an object describing the event, so it puts that object on every worker's work queue.

The pattern permits multiple senders: A sender locks all of the work queues, then emplaces the request, then unlocks. Thus all workers see the same ordering of requests.

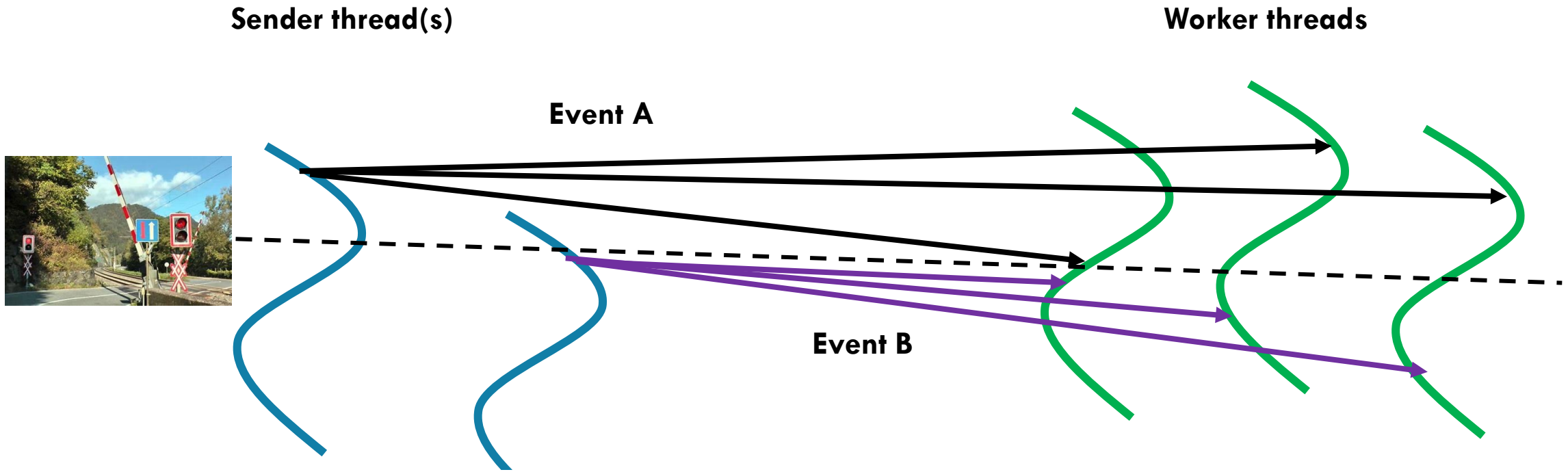
ORDERED MULTICAST PATTERN



ORDERED MULTICAST PATTERN



ORDERED MULTICAST PATTERN



This is often done using a form of barrier. There are many ways to build the barrier. In this example, A gets ordered first, so B has to wait

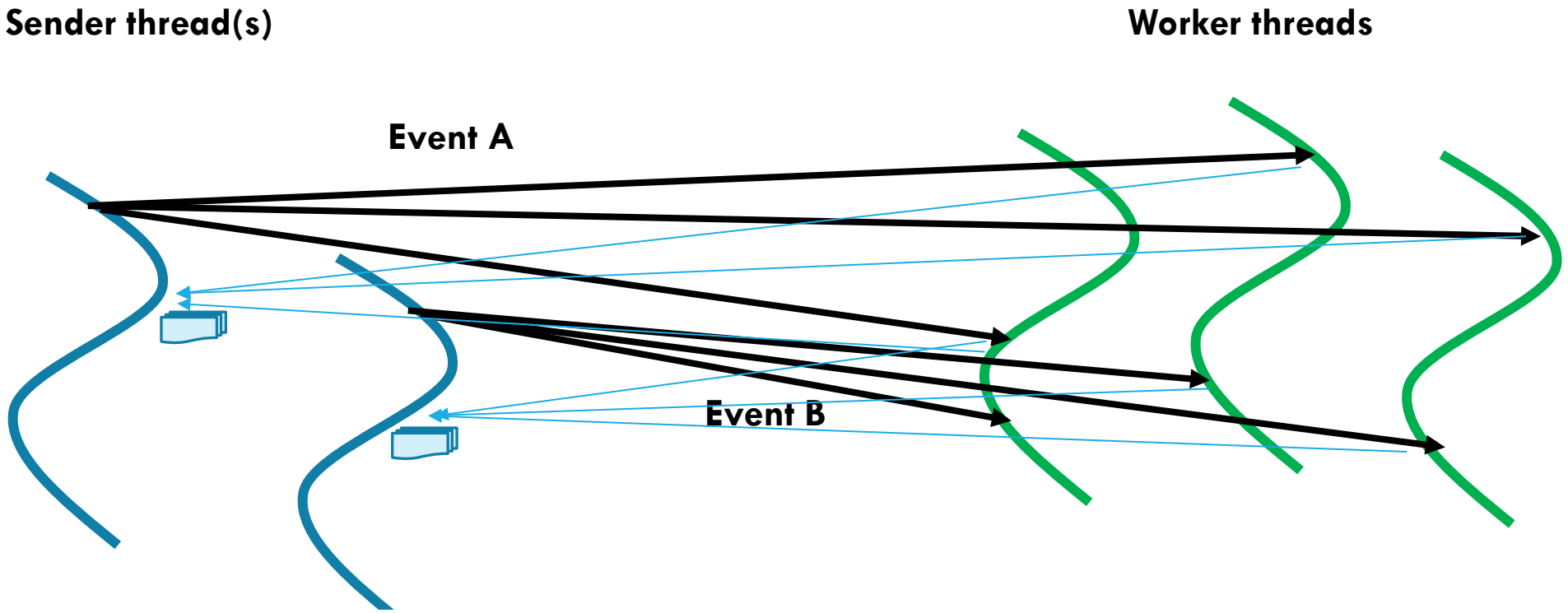
ORDERED MULTICAST WITH REPLIES

In this model, we start with an ordered multicast, but then the leader for a given request awaits replies by supplying a reply queue.

Often, this uses a std::future in C++: a kind of object that will have its value filled in "later".

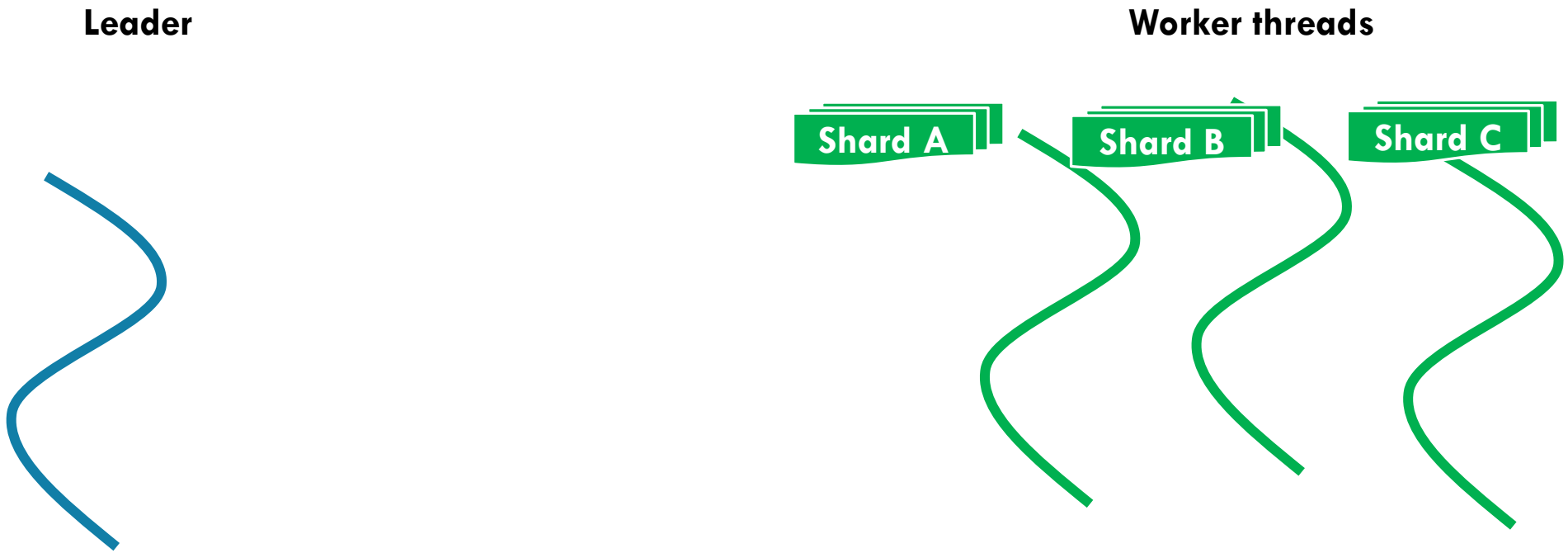
The leader makes n requests, then collects n corresponding replies.

ORDERED MULTICAST PATTERN



With replies, workers can send results back to the sender threads.

ALL-REDUCE PATTERN: SHARDED DATA SET



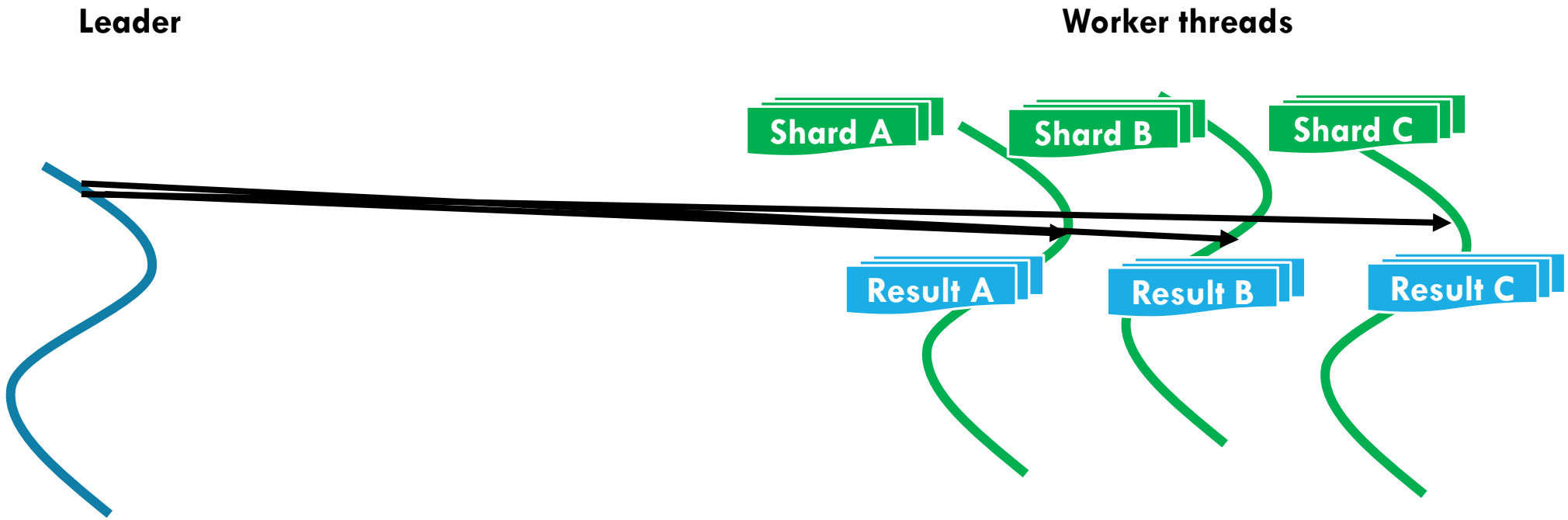
ALL-REDUCE: BROADCAST STEP

The leader sends a request to all workers in some set. They are scattered over different shards, hence have distinct local data.

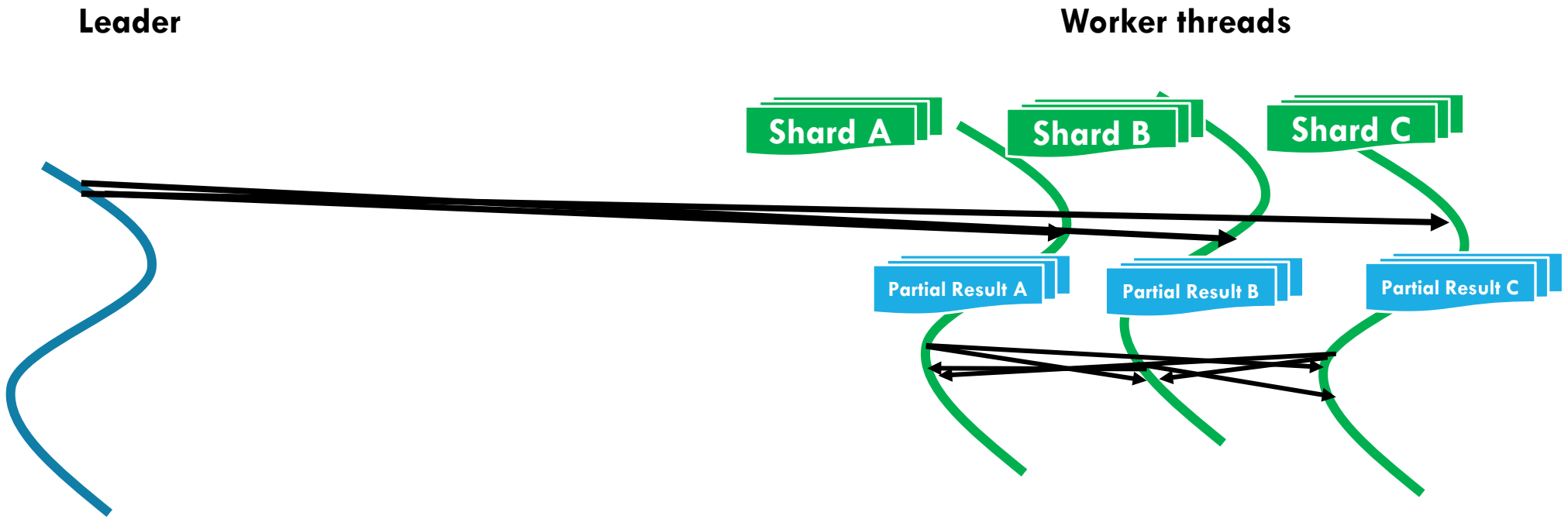
Each worker performs its share of the work by applying the requested function to the data in its shard.

When finished, each worker will have a share of the result.

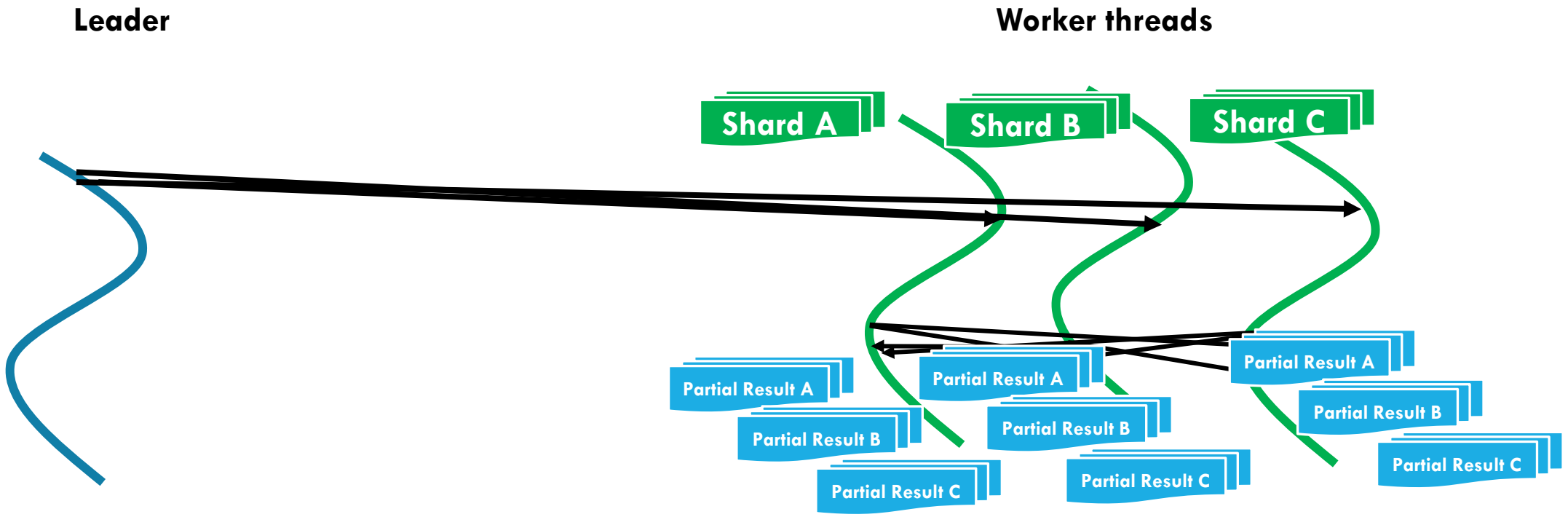
ALL-REDUCE PATTERN: BROADCAST



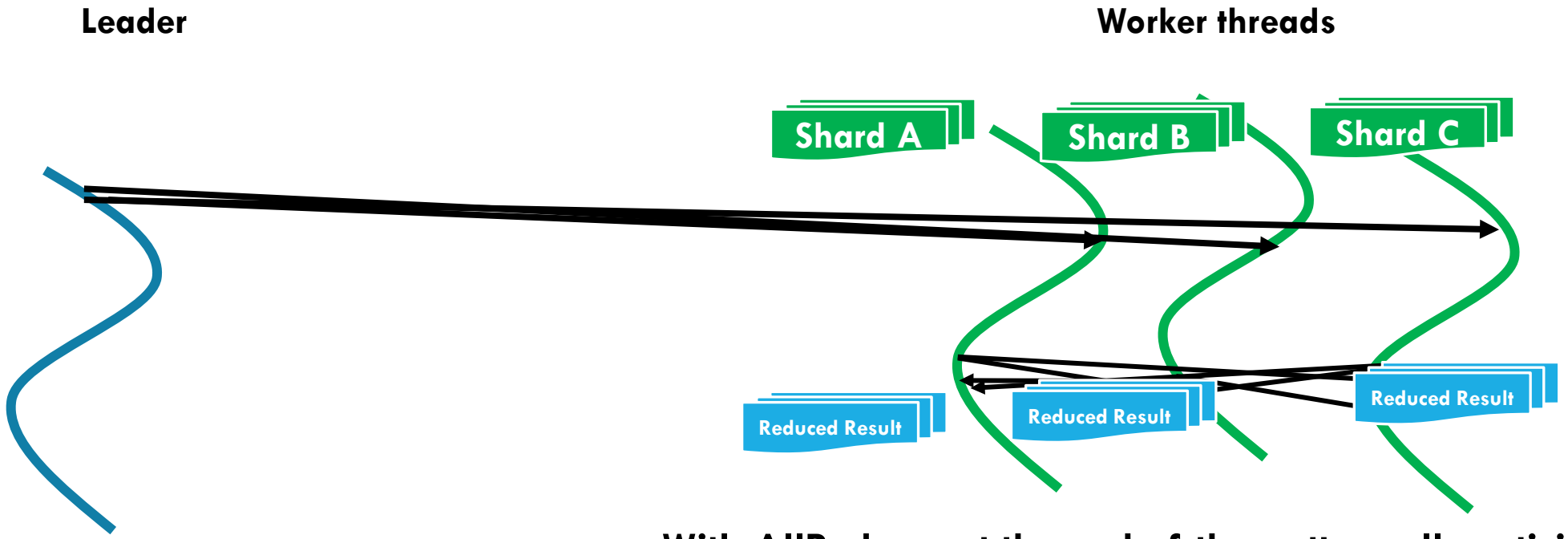
ALL-REDUCE PATTERN: ALL-TO-ALL



ALL-REDUCE PATTERN: ALL-TO-ALL

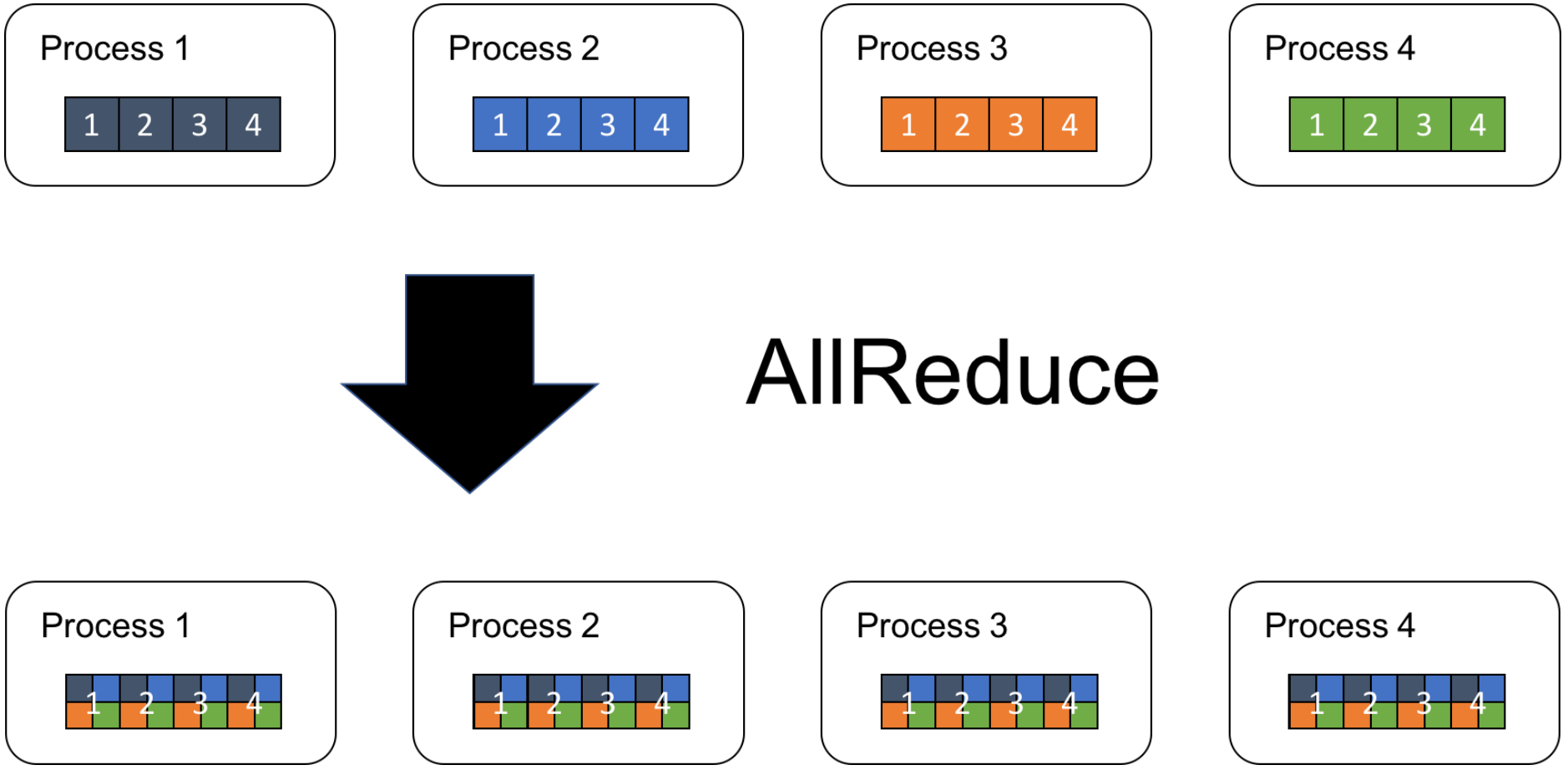


ALL-REDUCE PATTERN: ALL-TO-ALL



With AllReduce, at the end of the pattern all participants have identical "replicas" of the reduced result. The map step is usually the slow one, and reducing is usually fast

EXAMPLE: START WITH VECTORS OF INTS (DIFFERENT FOR EACH WORKER). REDUCE SUMS THE INTEGERS



MAP-REDUCE

Same idea, but we shard the partial results (these are assumed to be (K,V) tuples, and the keys are used to map to the shards)

The worker on shard s ends up with the reduced result for keys that map to shard s, rather than all workers having identical outcome results.

Very useful in big-data settings (All-Reduce runs out of memory)

COLLECTIVE COMMUNICATION LIBRARY PRIMITIVES

API	One-Sentence Description
All-Reduce	Combines values from all processes and distributes the result back to all.
Reduce-Scatter	Reduces values across processes and scatters the result to all processes.
All-Gather	Gathers data from all processes and distributes the combined data to all.
Reduce	Combines values from all processes and sends the result to a single process.
Broadcast	Sends data from one process to all other processes.
All-to-All	Each process sends distinct data to every other process and receives distinct data in return.
Map-Reduce	Processes map the request and their local shard content to create intermediate key-value pairs. These are shuffled to shard owners, which reduce them into a final result.

Many programmers rely on All-Reduce for ML compute. Big-data mining systems favor Map-Reduce, because it keeps the computation and results sharded.

COULD WE USE CCLS IN A MULTITHREADED PROGRAM, TOO?

Modern servers can have 100 NUMA CPUs per computer

At this scale it makes sense to use CCL styles of computing even among your threads!

The CCL libraries would not work in such cases but you can easily implement this pattern of interactions by hand

SUMMARY

Modern machine learning applications often compute on big collections of input files, which they transform into objects or databases

Object-oriented software engineers approached this using design patterns. Today we see similar patterns in coordination, synchronization and distributed computing.

ML systems make especially heavy use of All-Reduce, and Big Data systems love Map-Reduce

SELF-TEST

It is move-in day on campus!

How does the coordination of the arriving families and students resemble the coordination patterns we discussed?

Thinking of All-Reduce and Map-Reduce, which is more similar to a move-in coordination pattern?

THOUGHT QUESTION

Consider doubles tennis

Does this fit any of the coordination patterns? Why not?

What does this tell us about the "coverage" of software design patterns? Lorenzo Alvisi teaches a course, cs5414, that looks at a wide variety of distributed coordination and agreement issues beyond the CCLs.

Doubles tennis matches a computing style Alicia uses in her work that was studied by Amy Ousterhout in a paper called "The Power of Two." It appeared at SOSP around 2013.

THOUGHT QUESTION (3 SLIDES)

You work at a major insurance company, and have been hired to implement a supervised fine tuning LLM solution for the online chat bot on the web page.

For each category (vehicles, property, life, annuities,etc) your company has a huge number of specialized products. Many of those involve choices of investment mixtures.

As a result, the company has tens of thousands of documents and articles. Different client questions should retrieve the right ones. Your job is to reduce this to a job an LLM could do?

SLIDE 2 OF 3: EXAMPLES OF HOW IT MIGHT BE USED

"I am purchasing a life annuity with a seven year roll-up. Should I delay in the hope that the guaranteed rate will rise?"

"I own a life insurance product. I'm thinking about shifting the underlying death benefit to a market index more focused on AI. Which options should I consider?"

"I just purchased a motor cycle and am curious about your insurance pricing. I have a few minor citations on my car license."

SLIDE 3 OF 3: THE ACTUAL QUESTION!

You've decided to start by using an LLM to figure out which documents could be useful for which categories of client and product.

Then you will run a supervised fine tuning reinforcement learning process that takes a foundation LLM and produces a specialist LLM for each client/product pairing.

Which coordination patterns are likely to be useful at each step?

SELF-TEST QUESTIONS

Think back to what we learned about how the Linux file system is implemented (this was covered in Lecture 4)

What aspects would probably perform badly if a Linux file system needed to hold billions (or trillions) of files, one per object?

If you really had to host that many objects on Linux how would you do it? (Hint: read about “Linux archives”).

SELF-TEST QUESTIONS

Suppose that a task involves scanning so many objects that no single server can do it sufficiently quickly.

Think of some task that fits this description. Now invent a way to do the computation in a sharded manner (you may need a final step that combines the results).

Can you identify a scanning task that simply cannot be sharded?

SELF-TEST QUESTIONS

Suppose that you have gained access to a very large amount of data from an IT ticketing system, like the one for the COECIS helpdesk.

In addition to IT ticket records (in a format like JSON), the data includes logs from all the servers used by your company. The log contents and formats vary widely.

How would you approach a task such as creating weekly reports highlighting the top ten issues customers encounter, and assessing overall system stability and reliability over time?

SELF-TEST QUESTIONS

Consider some form of unstructured data that you might wish to transform into structured data for an AI system, such as a medical information system.

What kinds of missing data might be seen in the form of unstructured data you are thinking about?

Can these missing items be "filled in"? How might doing so run the risk of ML hallucinations later? Conversely, how might ignoring incomplete data records cause problems? How do humans deal with such problems?

SELF-TEST QUESTIONS

Still assuming you work for a medical records company, suppose your job involves creating agentic ML services for other companies to access records

Identify examples of privacy obligations this kind of agent would need to respect (these are called *HIPPA obligations* in the US medical system).

Now brainstorm a big-data object-oriented approach (with as many microservices as needed) to support agentic queries without violating HIPPA.