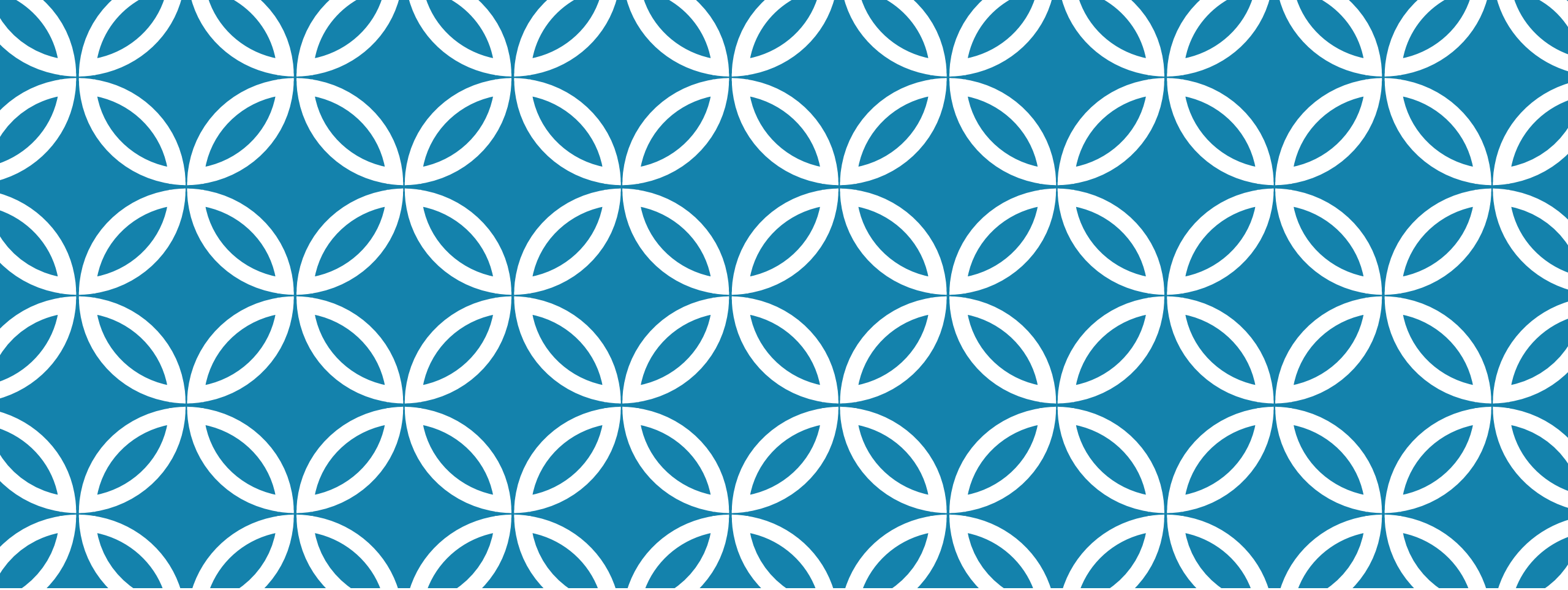


COORDINATION

Professor Ken Birman

CS4414 Lecture 17

IDEA MAP FOR TODAY

Reminder: Thread Concept

Lightweight vs. Heavyweight

Thread "context"

C++ mutex objects. Atomic data types.

The monitor pattern in C++

Problems monitors solve (and problems they don't solve)

Deadlocks and Livelocks

Additional Coordination Patterns

Today we focus on other patterns for coordinating threads or entire processes.

WITHOUT COORDINATION, MANY SYSTEMS MALFUNCTION

Performance can drop unexpectedly

Overheads may soar

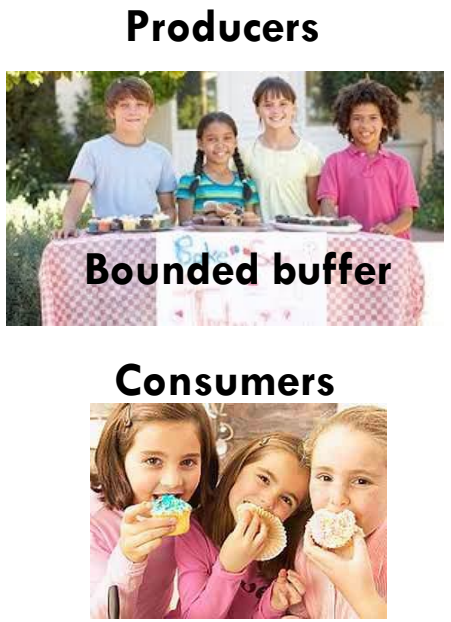
A coordination pattern is a visual or intellectual tool that we use when designing concurrent code, whether using threads or processes. It "inspires" a design that works well.

WHAT IS A COORDINATION PATTERN?

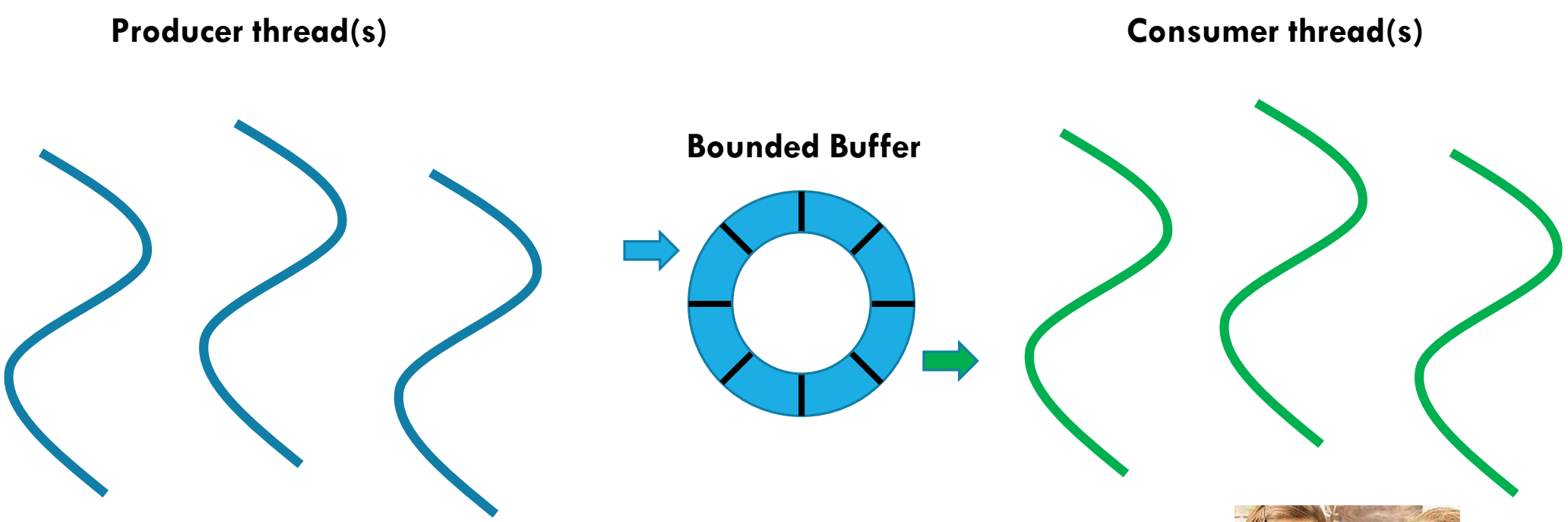
Think about producer-consumer (cupcakes and kids).

- The producer pauses if the display case is full
- The consumers wait if we run out while a batch is baking

This is an example of a coordination pattern.



PRODUCER – CONSUMER PATTERN



ANALOGY: SOFTWARE DESIGN PATTERNS

Motivation: Early object-oriented programming approaches had a very flat perspective on programs:

We had objects, including data structures.

Threads operated on those objects.

Developers felt that it was hard to capture higher-level system structures and behaviors by just designing some class.

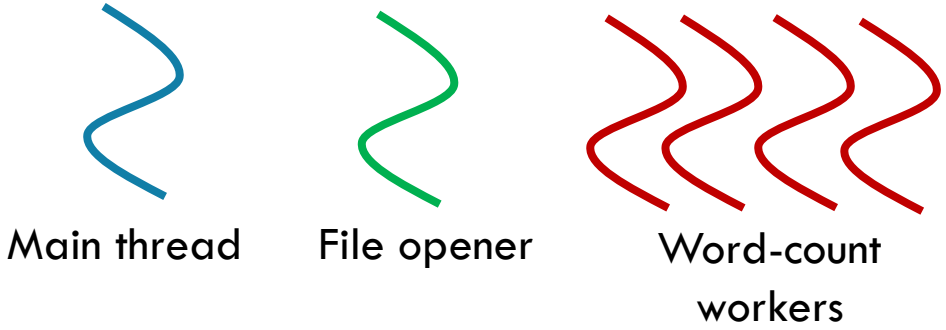
MODULARITY FOR COMPLEX, THREADED PROGRAMS

With larger programs, we invariably need to break the overall system up and think of it in terms of subsystems.

Each of these may have its own classes, its own threads, and its own internal patterns of coordination and behavior.

When a single system has many such “modules” side by side, the patterns used shape the quality of the resulting application

SOME EXAMPLES.



Fast-wc had a main thread, a thread for opening files (a form of module), a set of concurrent word counters, logic to merge the resulting std::map trees, and finally logic for sorting and printing the output.

We can think of this structure in a modular way. In fact, we *need* to think of it in a modular way to understand it!

WHAT EXACTLY DOES "MODULAR" MEAN?

A modular way of describing a system breaks it down into large chunks that may have complex implementations, but that offer simple abstraction barriers to one-another.

The operating system has many modules: the file system, the device drivers, the process management system, the clock system

Each involves a substantial but "separate" chunk of code.

MORE EXAMPLES

We touched on databases in Lecture 16

Databases often have a subsystem for file I/O, a subsystem to create quick index structures for fast item retrieval, subsystems to interact with users, subsystems to compile and execute queries

Each of these is like a module within a shared address space

MORE EXAMPLES

Web servers at companies like Amazon, Facebook, Netflix

The Linux kernel

The C++ compiler

C++ MODULARITY FEATURES

In fact, C++ has features to help with designing modular systems. C++ namespaces allow you to avoid accidental naming conflicts if two modular components happen to reuse names.

A C++ application can manage the mapping of threads to NUMA cores, and a parent thread can track or manage its children.

std::thread scheduling can be configured for these thread groups.

WHAT ABOUT MODULARITY FOR COORDINATION, LIKE IN HOMEWORK 3 PART II?

At present, these are not "baked into" std libraries, but you can easily implement your own classes using them.

Some are starting to show up in the boost:: libraries, which are "future ideas for C++ xx." Not all will make it!

Many companies are nervous about Boost (open source)

INSPIRATION: SOFTWARE ENGINEERING

There is some similarity between "synchronization" patterns and "software design patterns"

We learn about those in CS2110

Basic idea: *Problems that often arise in object oriented programs, and effective, standard ways of solving them.*

EXAMPLE: THE OBJECT VISITOR PATTERN

The visitor design pattern associates virtual functions with existing classes.

The class offers a static method that permits the caller to provide an object (a "functor") that implements this function interface. The base class keeps a list of visitors, and will call those functions when objects of the base-class type are created or modified.

With this you can build new logic that takes some action that was not already part of the design when the base class was created!

REMINDER: INTERFACES

In a C++ .hpp file, one normally puts the declarations of classes and templates, but the bodies are often in a .cpp file.

A “virtual” class is one that has a .hpp file defining it, but no implementations. An **interface** is a standardized virtual class.

A C++ class can “implement” an interface, and then you can pass class objects to any method that accepts the interface type.

EXAMPLE OF HOW YOU MIGHT USE VISITOR

Suppose that you wanted to "monitor" a collection of files.

We could build a base class that understands the file system and watches for changes. But we built that in 2020, and you might plan to use this logic as a library in 2025. In 2020 we can't guess at what you will be coding 5 years from now.

So our monitor class uses "visitor". In 2025 you will register a functor and it will receive "upcall events" each time a file of interest changes. And this works even if you have multiple visitors all using the file watcher class.

HOW TO THINK ABOUT THE VISITOR IDEA

When the binoculars were created, the company creating them didn't know who would use them and how.

This observer is a visitor. She knows how to use binoculars. The binoculars pass images to her. They "do upcalls to a virtual interface function".

The main difference is that with visitors several observers could share the one pair of binoculars. They get called one by one.

VISITOR PATTERN USE CASES

The visitor pattern is common with file systems: if an application is interested in a file or folder, this pattern allows one module to "refresh" when some other module makes a change.

It is also useful with GUI displays. If something changes, the GUI can refresh or even recompute its layout.

WHY IS IT HELPFUL TO GIVE THIS PATTERN A SPECIAL NAME AND A STANDARD API?

Visitor is a well known pattern and even taught in courses on software engineering.

So anyone who sees a comment about it, and then sees the Watch method, knows immediately what this is and how to use it.

In effect, it is a standard way to do “refresh notifications”

WHY IS THIS SUCH A BIG DEAL?

With patterns, we often find that we build one module now, and then some other module later (or separately), and eventually they need to be connected.

By agreeing on interfaces, a module is free to use any classes it needs and yet its objects can still "talk" to methods in the other modules. Those methods specify the interface it uses, and any object supporting the interface can be passed in.

FACTORY PATTERN

Another example from software engineering.

A “factory” is a method that will create some class of objects on behalf of a caller that doesn’t know anything about the class.

Basically, it does an allocation and calls a constructor, and then returns a pointer to the new object.

WHY A FACTORY IS USEFUL

If module A has code that explicitly creates an object of type Foo, C++ can type check the code at compile time.

But if module B wants to "register" class Foo so that A can create Foo objects, A might be compiled separately from B.

The factory pattern enables B to do this. A requires a factory interface (for any kind of object), and B registers a Foo factory

TEMPLATES ARE OFTEN USED TO IMPLEMENT MODERN C++ DESIGN PATTERNS

A template can instantiate standard logic using some new type that the user supplies. So this is a second and powerful option that doesn't require virtual functions and upcalls.

For example, we could do this for our bounded buffer. It would allow you to create a bounded buffer for any kind of object.

The bounded buffer *pattern* is valid no matter what objects it holds.

SUMMARY: WHY STANDARD SOFTWARE ENGINEERING PATTERNS HELP

They address the needs of larger, more modular systems

They are familiar and have standard structures. Developers who have never met still can quickly understand them.

They express functionality we often find valuable. If many systems use similar techniques to solve similar problems, we can create best-practice standards.

SYNCHRONIZATION PATTERNS

These are patterns that stretch across threads or even between processes. They can even be used in computer networks, where the processes are on different machines!

Producer consumer is a synchronization pattern.

SYNCHRONIZATION PATTERNS

Leader / workers is a second widely valuable synchronization pattern.

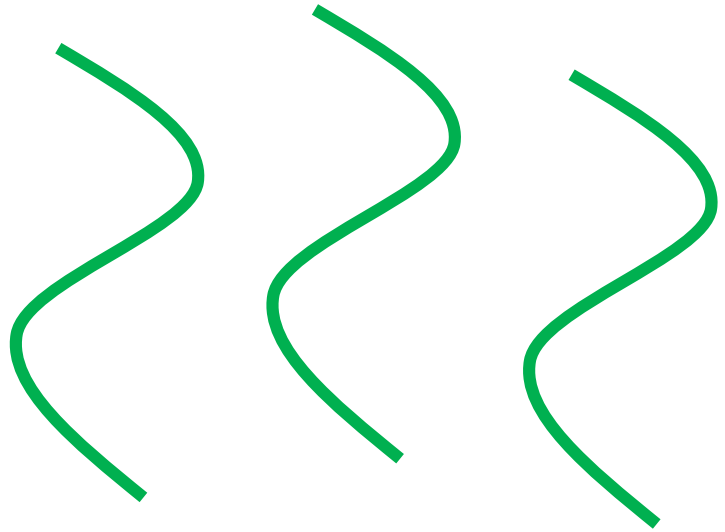
In this pattern, some thread is created to play the leader role. A set of workers will perform tasks on its behalf.

LEADER / WORKERS PATTERN

Leader thread

Tasks to be performed
(“peel these potatoes”)

Worker threads

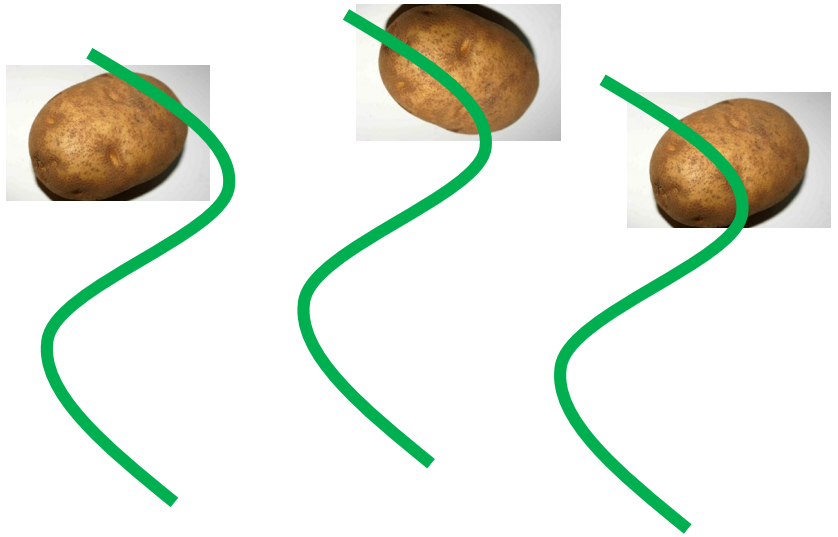


LEADER / WORKERS PATTERN

Leader thread

Tasks to be performed ("peel these potatoes")

Worker threads

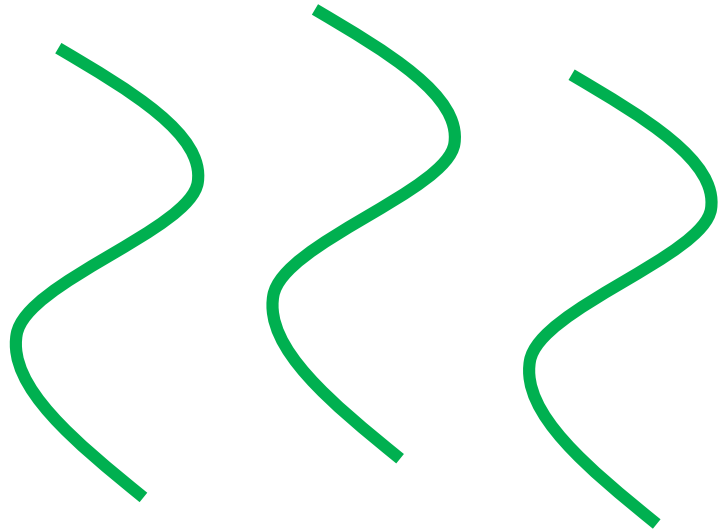


LEADER / WORKERS PATTERN

Leader thread

Bag is empty? Workers terminate (threads exit)

Worker threads



LEADER / WORKERS PATTERN

Word-to-do queue

We need a way to implement the bag of work.

One can pass arguments to the threads, but this is very rigid. If we have lots of tasks, it may be better to be flexible.

So the bag of work will be some form of queue. You'll need to protect it with locking! (*Why*?)

POOL OF TASKS

A std::list!

One option is to just fill a std::list with tasks to be performed, using a "task description object". Then launch threads.

The list has a front and a back, which can be useful if the task order matters. Some versions support priorities (a "priority queue").

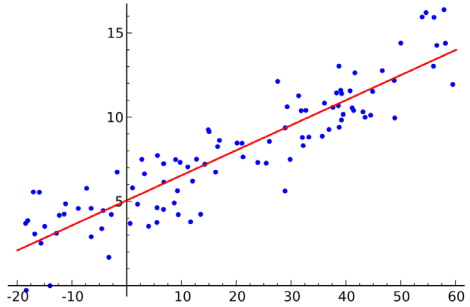
It is easy to test to see if the list is empty.

DYNAMIC TASK POOLS

Permits the leader to add tasks while the workers are running.

- The workers each remove a task from the pool, execute it, and then when finished, loop back and remove the next task.
- They may even use a second std::list to send results back to the leader! C++ calls this a *promise* pattern, supported by a std::promise library!
- But we can't use "empty" to signal that we are finished (*why*?). So, the leader explicitly pushes some form of special objects that say "job done" at the end of the task pool. As workers see these, they exit.

EXAMPLE: LOGISTIC REGRESSION



In AI, it is common to have a **parameter server** that creates a model, and a set of **workers** that work to train the model from examples. Later we will use the model as a classifer.

- Worker takes the current model plus some data files, computes a gradient, and passes this to the parameter server (the leader)
- Parameter server consumes the gradients, improves the model, then assigns a new task to the worker.
- Terminates when the model has converged.

BARRIER SYNCHRONIZATION

In this pattern, we have a set of threads (perhaps, the workers from our logistic regression example).

We use this pattern if we want all our threads to finish task A before any starts on task B.

For this, we use a barrier.

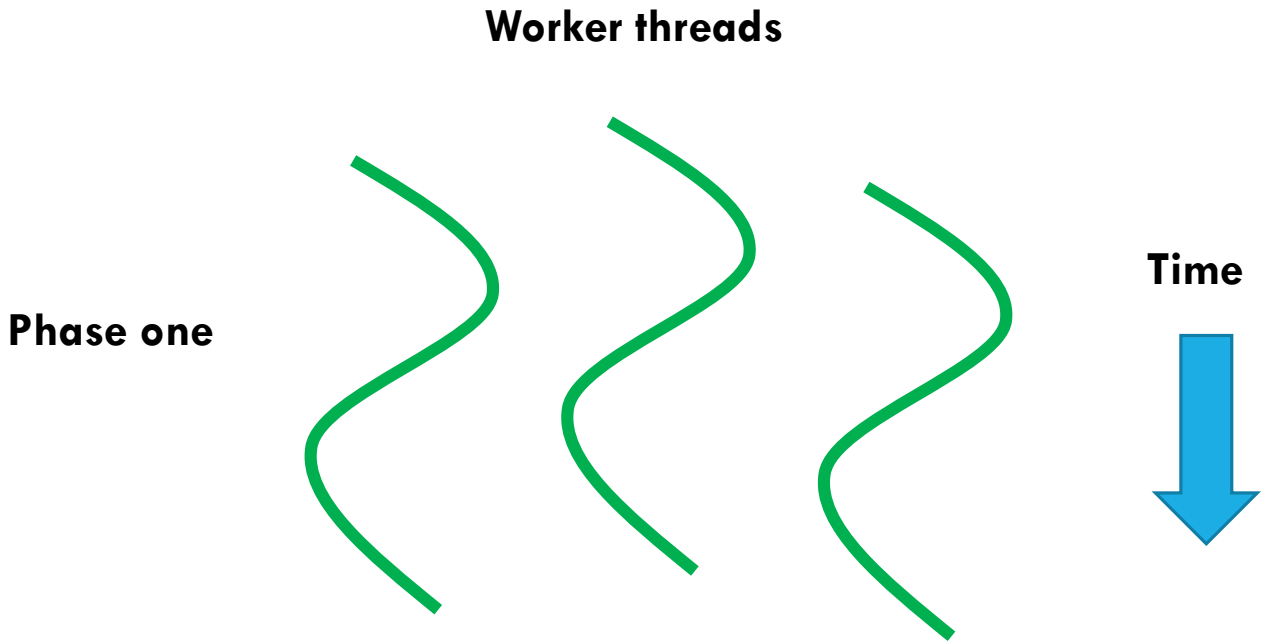
BUILDING A BARRIER

We normally use the monitor pattern.

The threads all call “barrier_wait”. This method uses a bool array to track which threads are ready, initialized to all false.

When all are ready, the thread that notices this issues notify_all to wake the others up. They wake up nearly simultaneously.

BUILDING A BARRIER



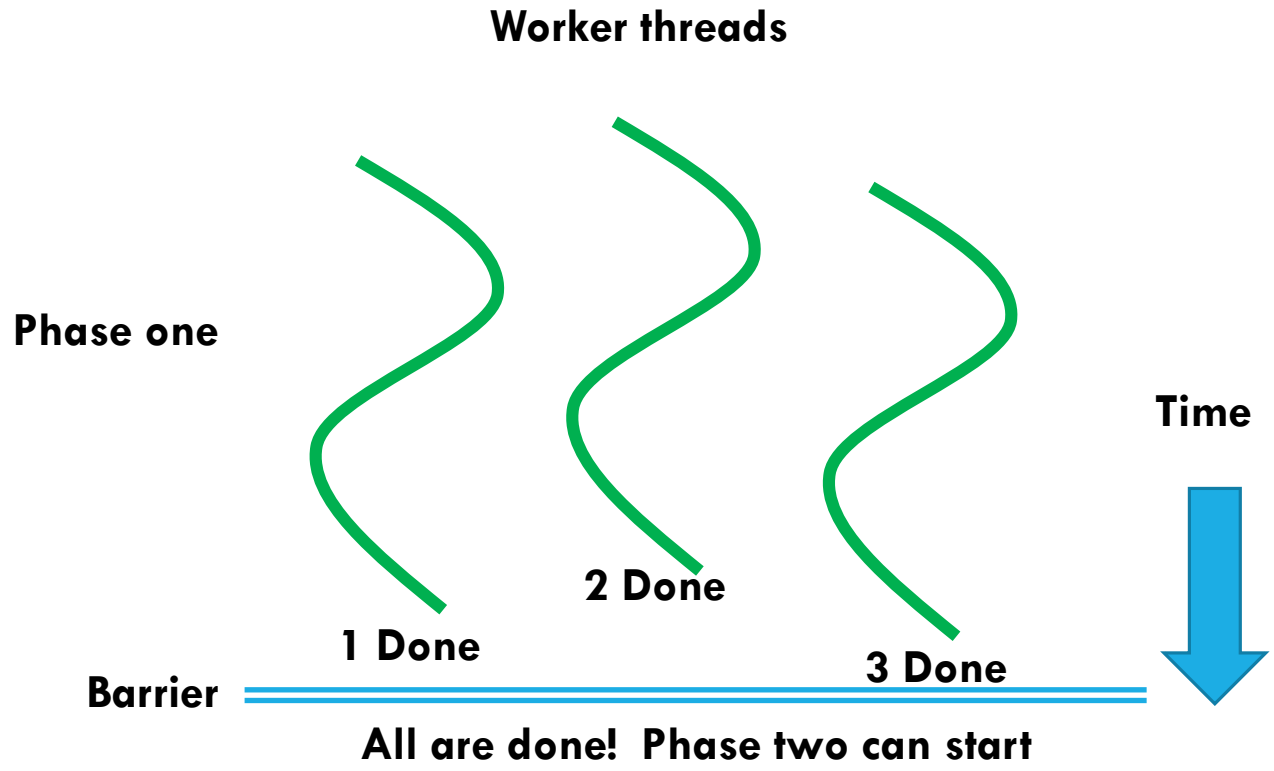
Example: A computation with distinct phases or epochs.

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

BUILDING A BARRIER

Example: A computation with distinct phases or epochs.

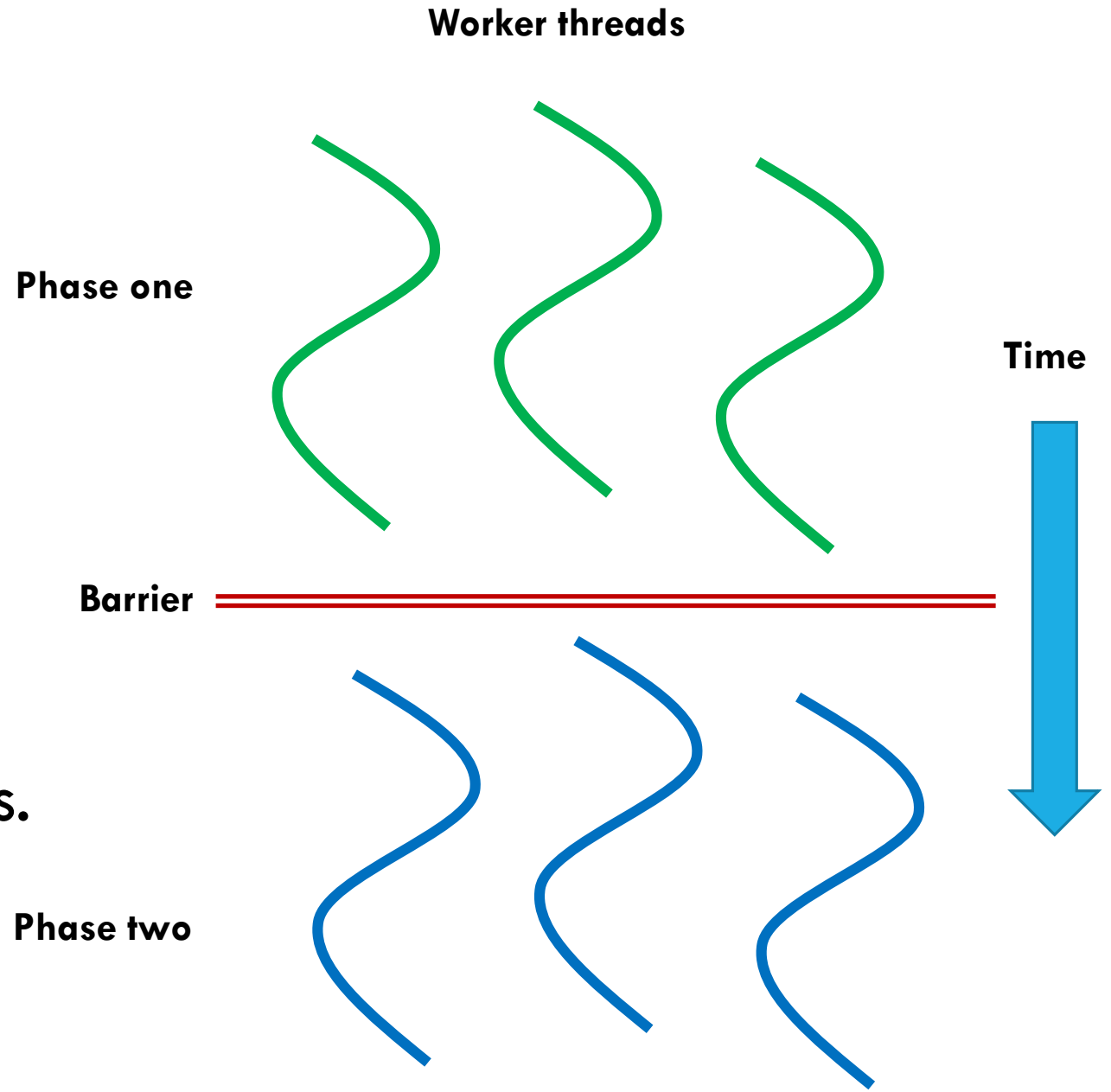
After phase one, all workers must wait until phase two starts.



BUILDING A BARRIER

Example: A computation with distinct phases or epochs.

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



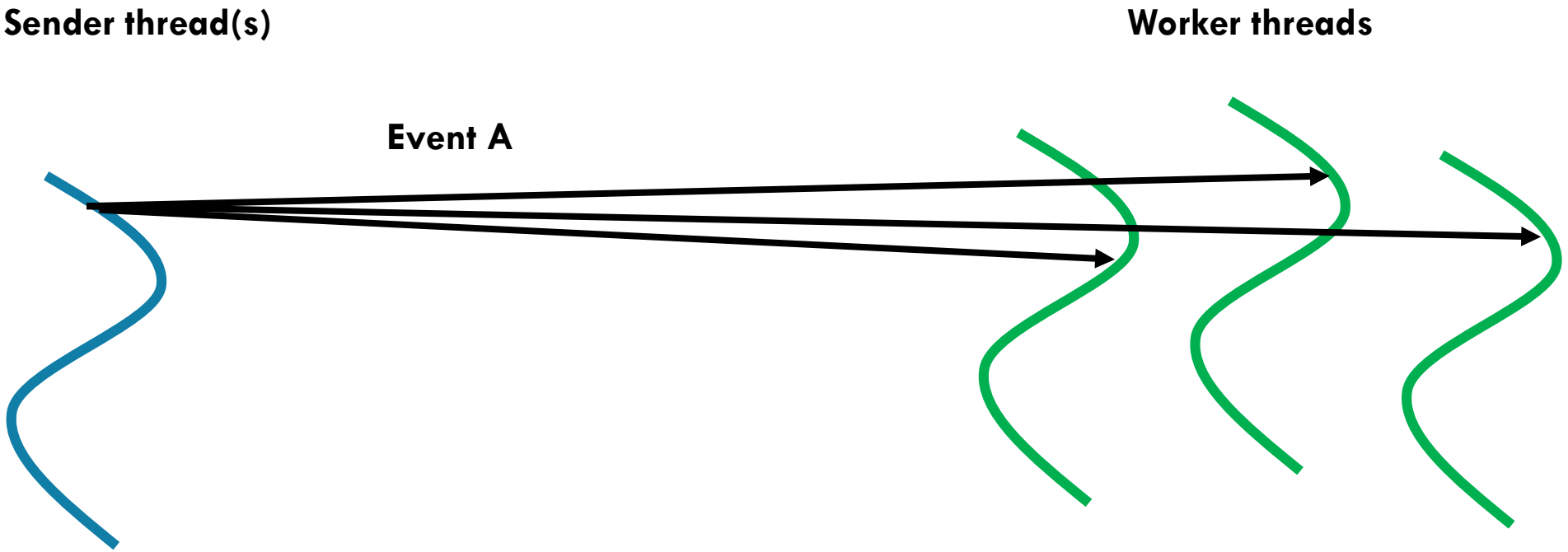
ORDERED MULTICAST PATTERN

This is a one-to-many pattern. Suppose some event occurs.

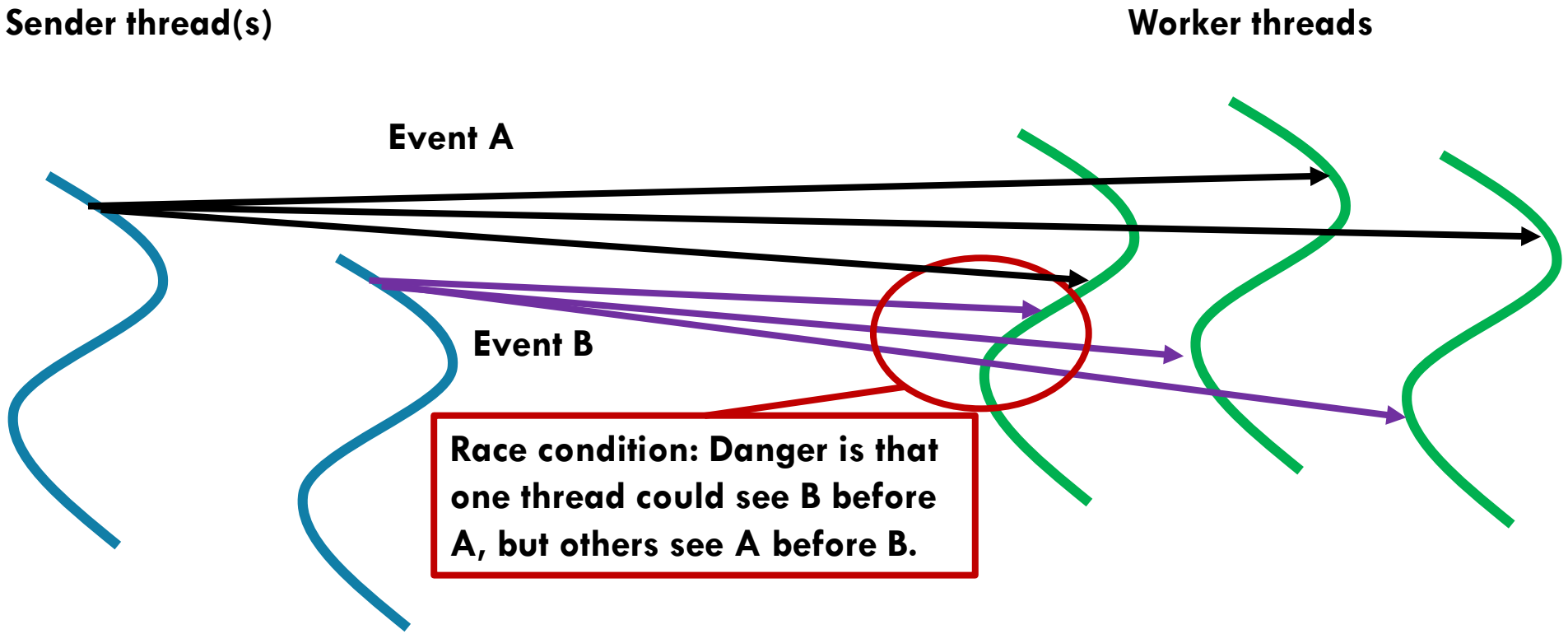
A sender thread needs every worker to see an object describing the event, so it puts that object on every worker's work queue.

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.

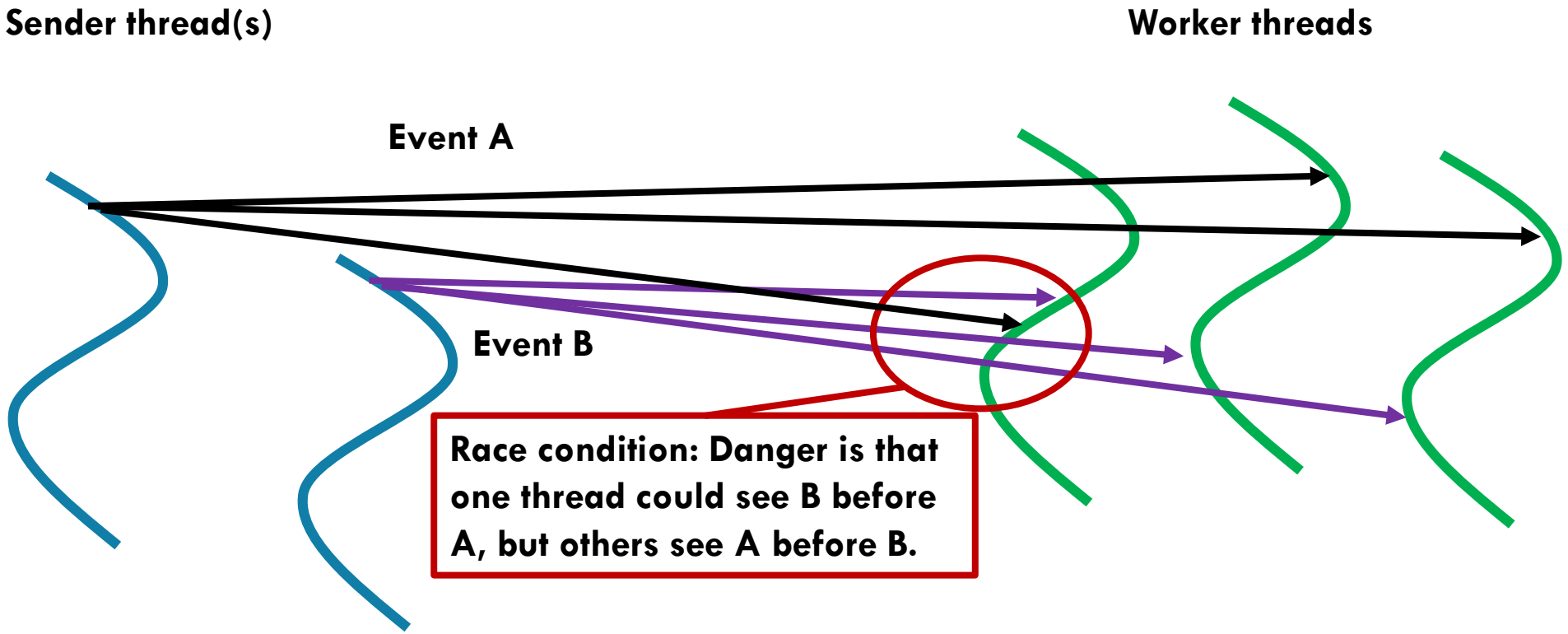
ORDERED MULTICAST PATTERN



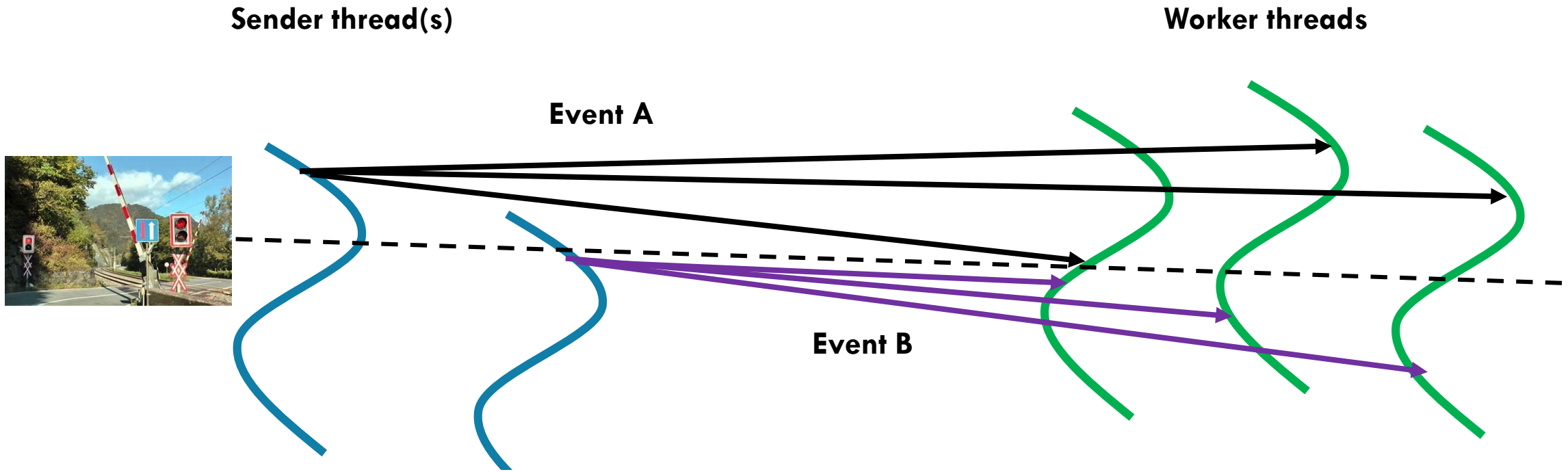
ORDERED MULTICAST PATTERN



ORDERED MULTICAST PATTERN



ORDERED MULTICAST PATTERN



An ordered multicast pattern implements a barrier that protects us against ordering inconsistencies. There are many ways to build the barrier. The *pattern* focuses on the behavior, not the implementation.

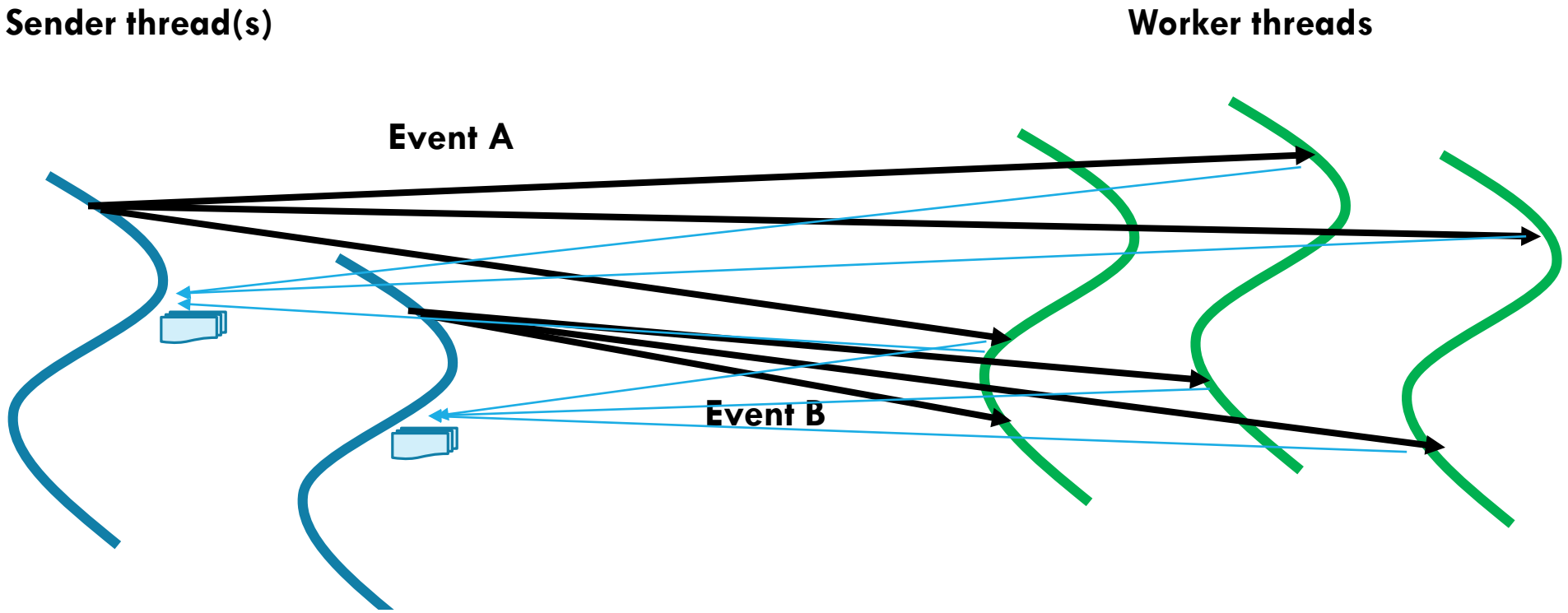
ORDERED MULTICAST WITH REPLIES

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.

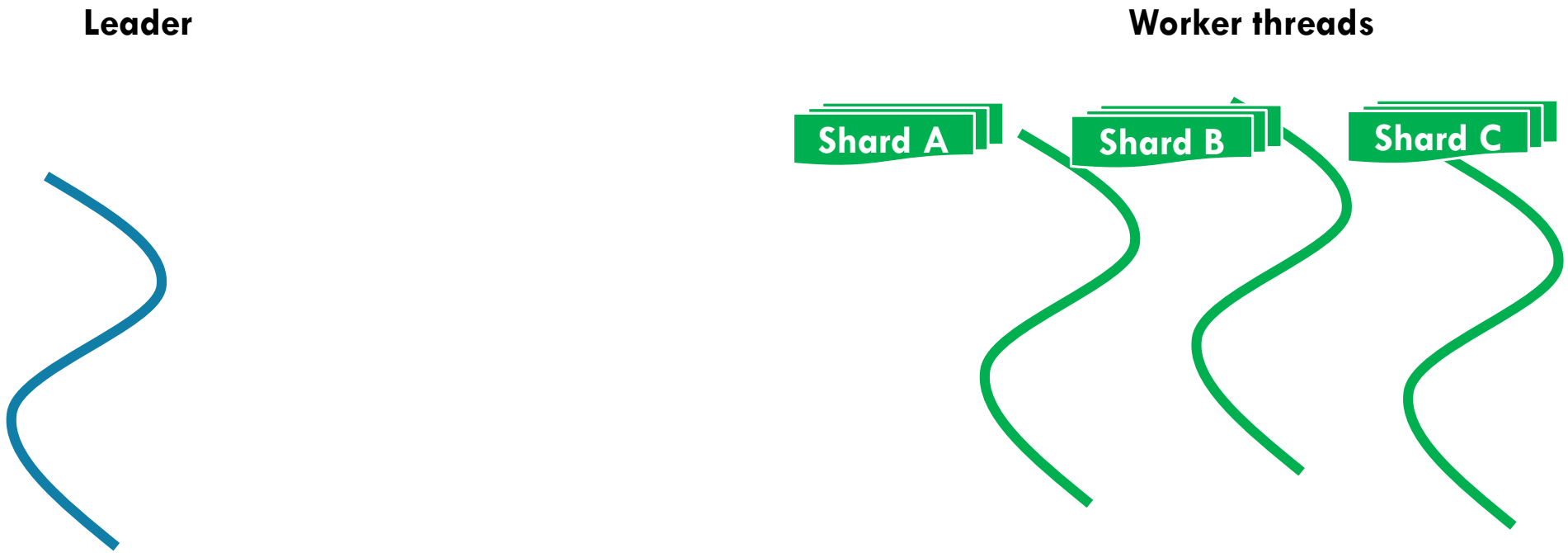
ALL-REDUCE PATTERN: IMPORTANT IN ML.

This pattern focuses on (key,value) pairs.

It assumes that there is a large (key,value) data set divided so that worker *k* has the *k'th shard* of the data set.

- For example, with integer keys, perhaps (key % n) == k
- With arbitrary objects, you can use the built-in C++ "hash" method.

ALL-REDUCE PATTERN: SHARDED DATA SET



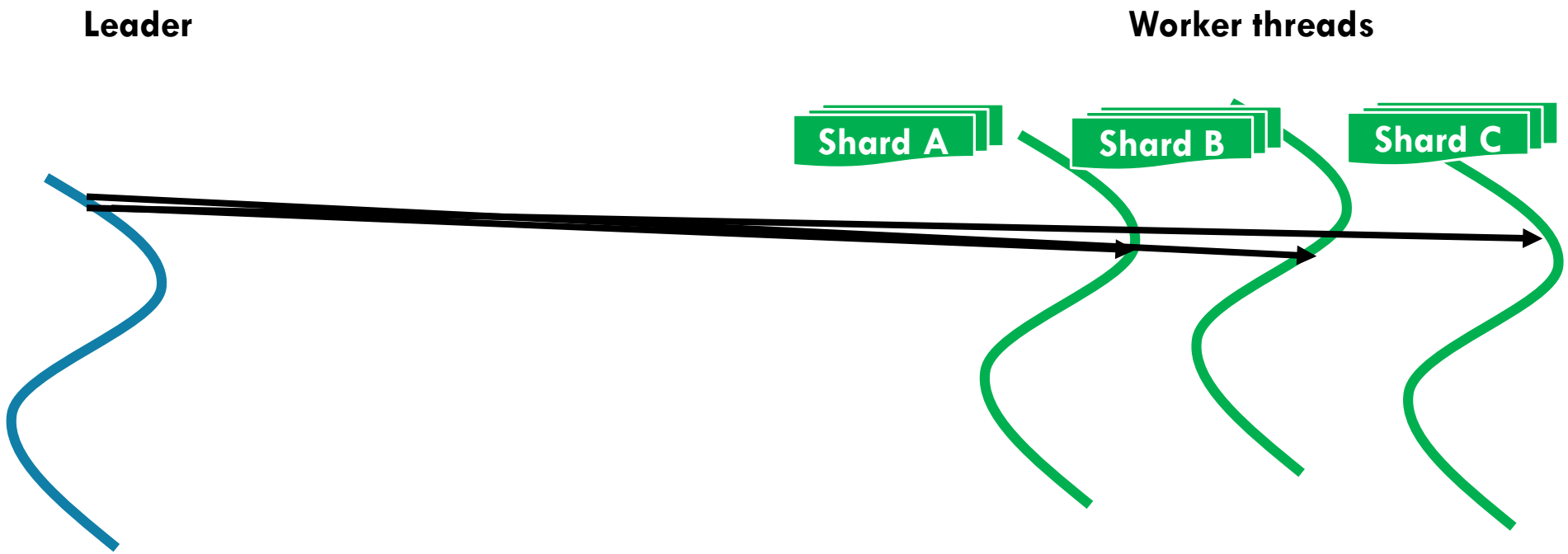
ALL-REDUCE: MAP STEP

The leader **maps** some task over the n workers. This can be done in any way that makes sense for the application.

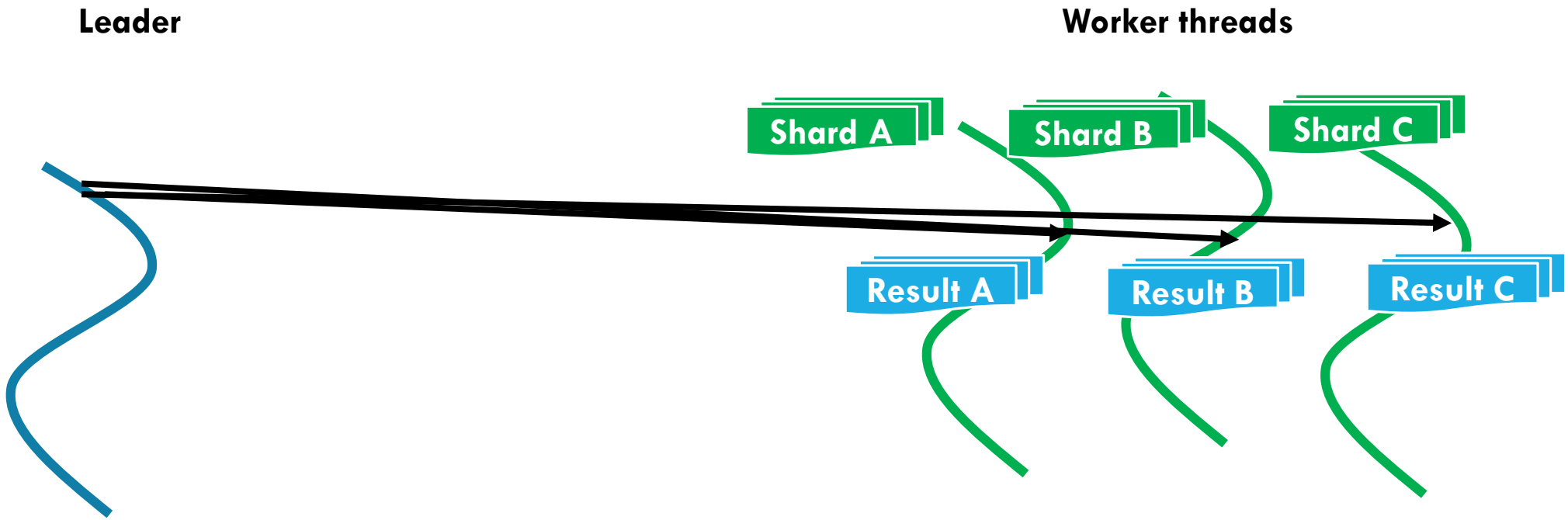
Each worker performs its share of the work by applying the requested function to the data in its shard.

When finished, each worker will have a list of new (key,value) pairs as its share of the result.

ALL-REDUCE PATTERN: MAP (FIRST STEP)



ALL-REDUCE PATTERN: MAP (FIRST STEP)



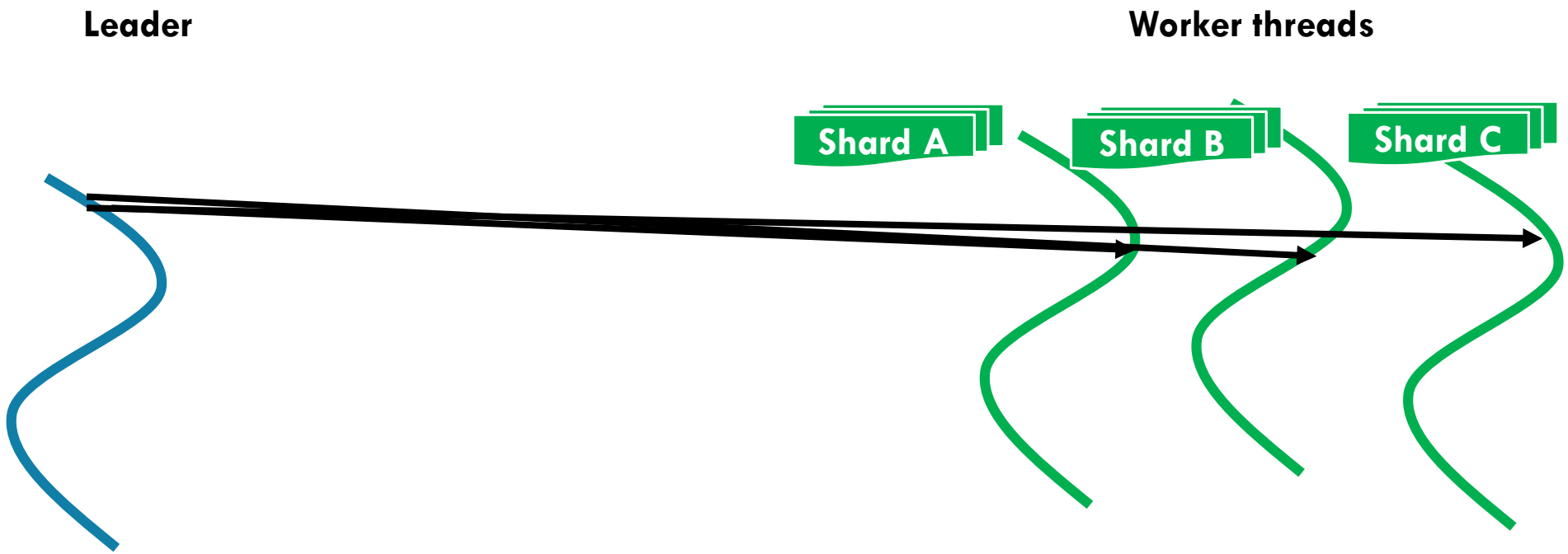
ALL-REDUCE: SHUFFLE EXCHANGE

Each worker breaks its key-value result set into n parts by applying the sharding rule to the keys.

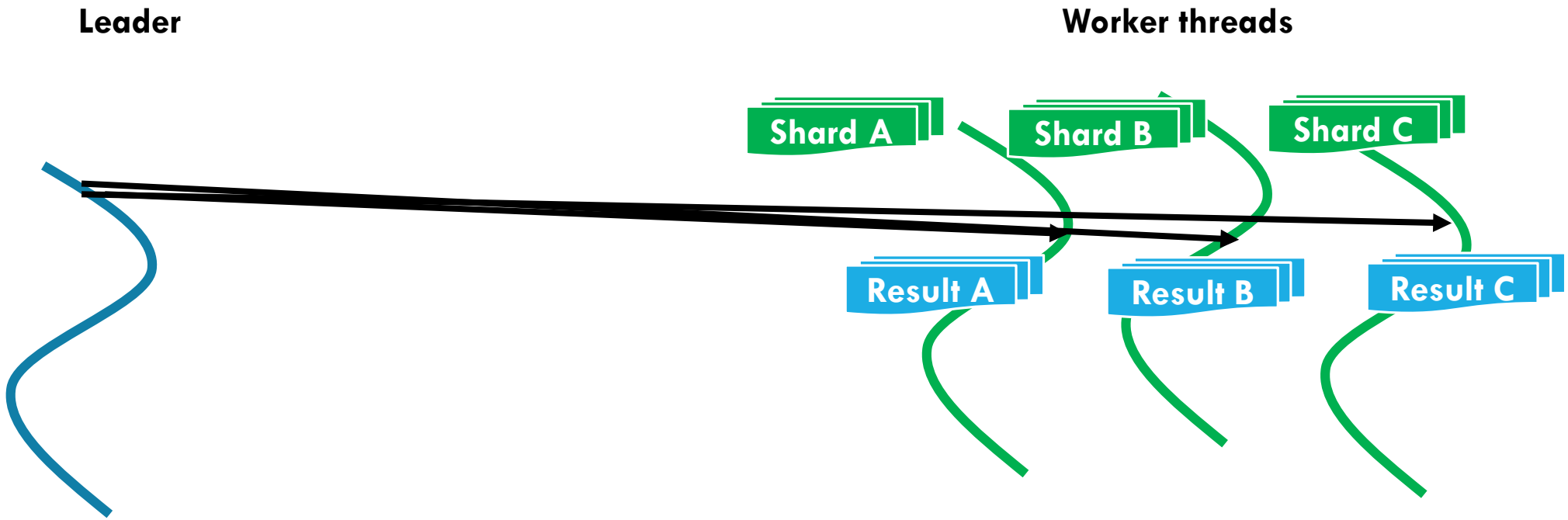
- Now it has one subset (perhaps empty) for each other worker.
- It hands that subset to corresponding worker.

Every worker waits until it has n subset, one from each worker.

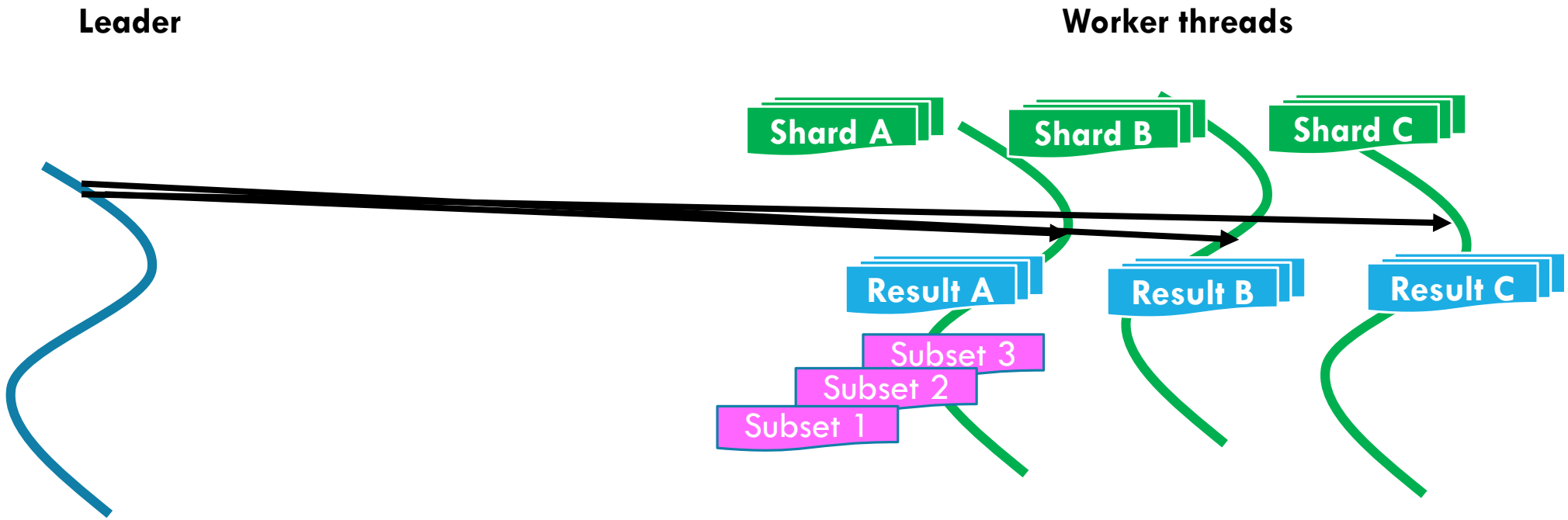
ALL-REDUCE PATTERN: MAP (FIRST STEP)



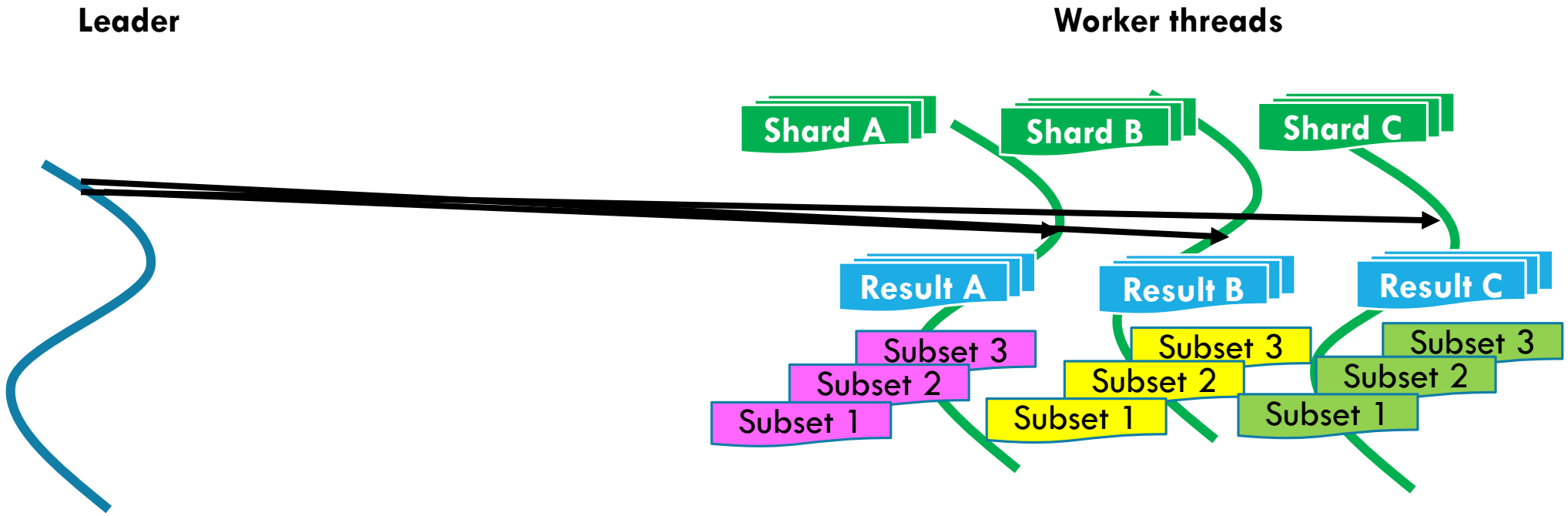
ALL-REDUCE PATTERN: MAP (FIRST STEP)



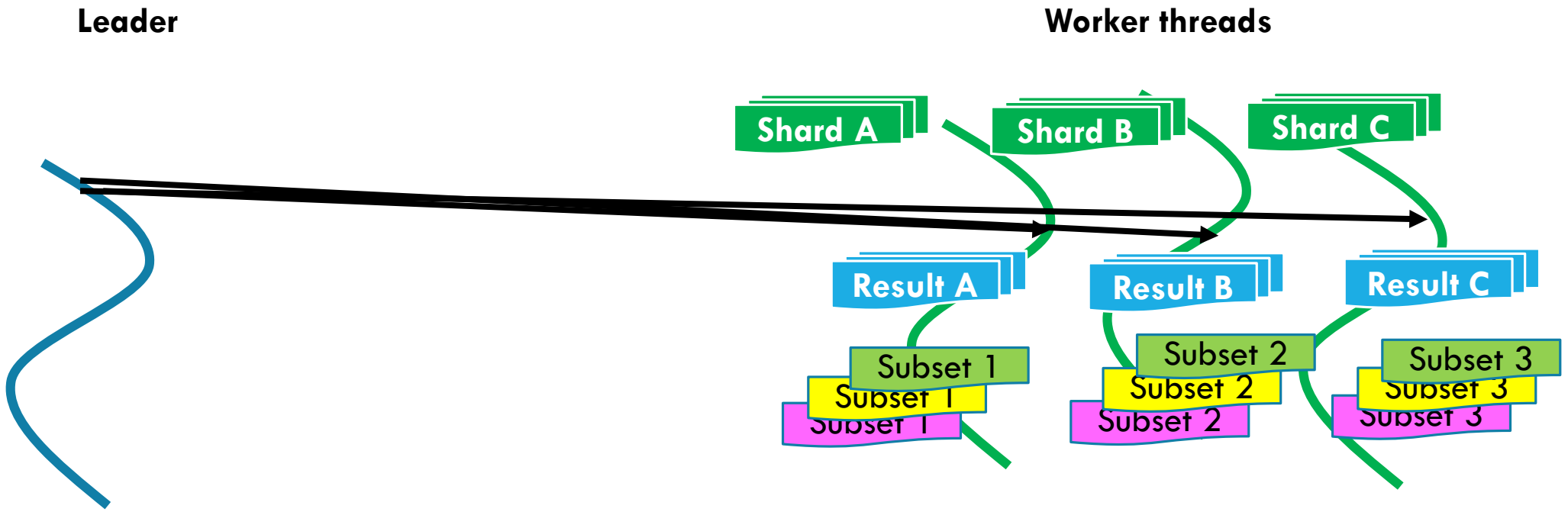
ALL-REDUCE PATTERN: MAP (FIRST STEP)



ALL-REDUCE PATTERN: SHUFFLE



ALL-REDUCE PATTERN: SORT



Not shown: There are messages being sent from A to B and C, from B to A and C, and from C to A and B. These "shuffle" the data

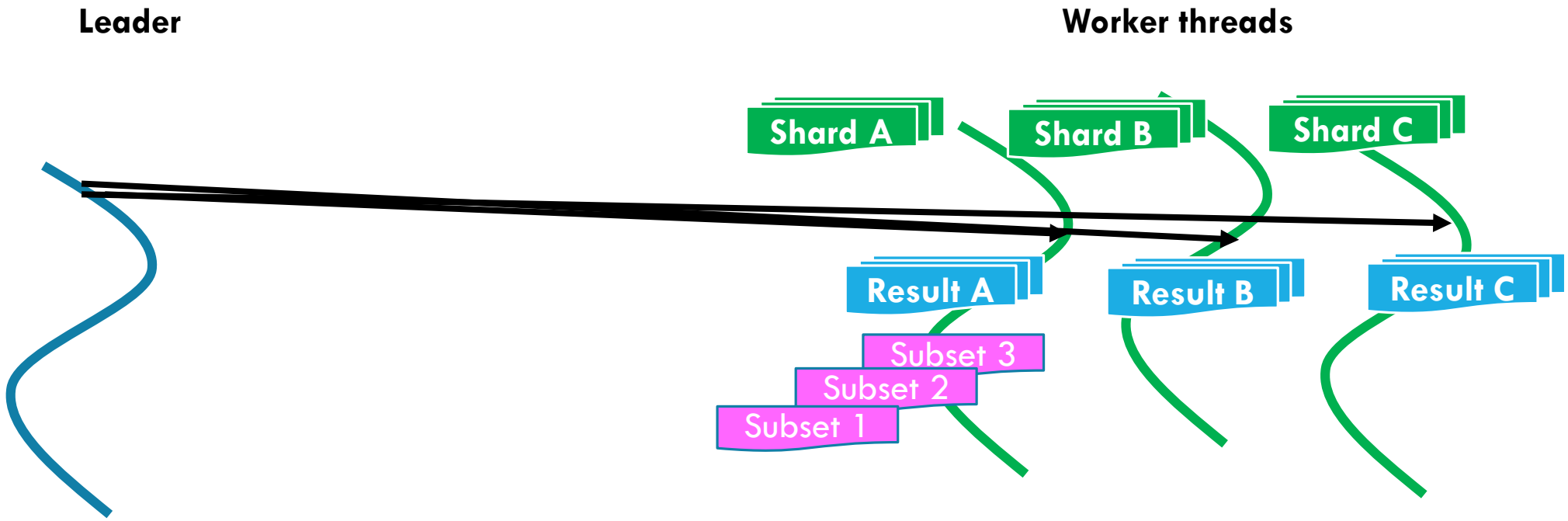
AFTER THE SHUFFLE STEP, WORKERS APPLY A REDUCE FUNCTION

Each worker combines the incoming data, then sorts by key.

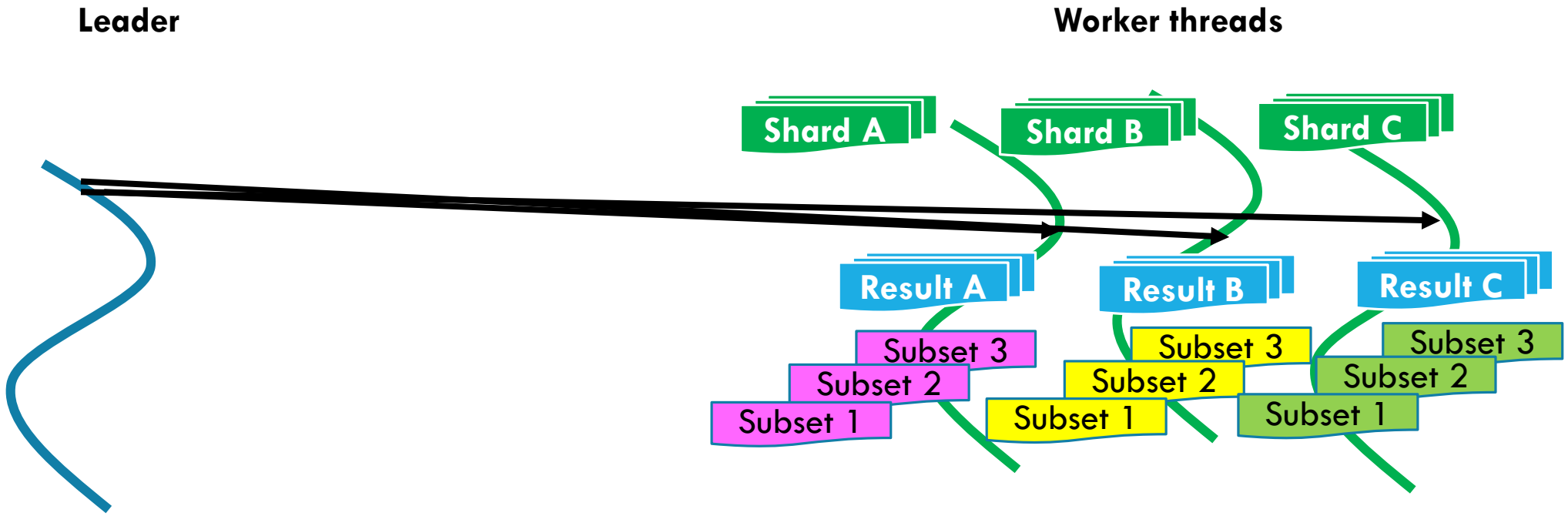
If it has multiple items with the same key, a **reducing function** is used to combine them. For example, **sum** might sum the values.

The new (key,value) pairs are the result of the all-reduce computation.

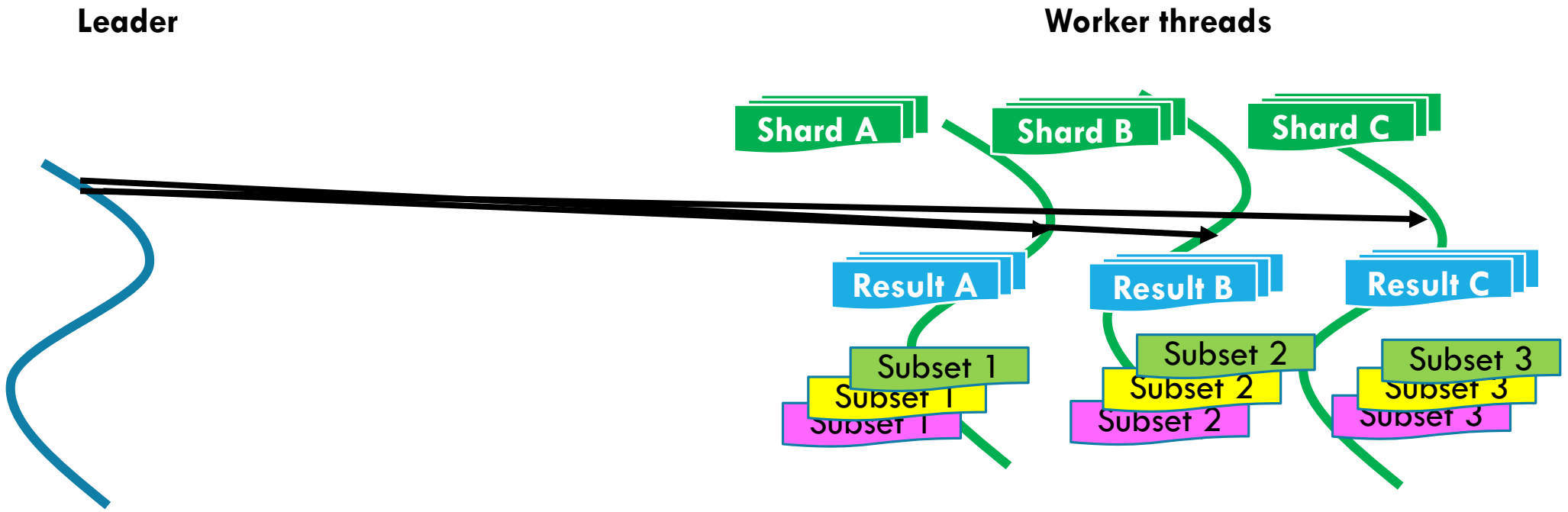
ALL-REDUCE PATTERN: MAP (FIRST STEP)



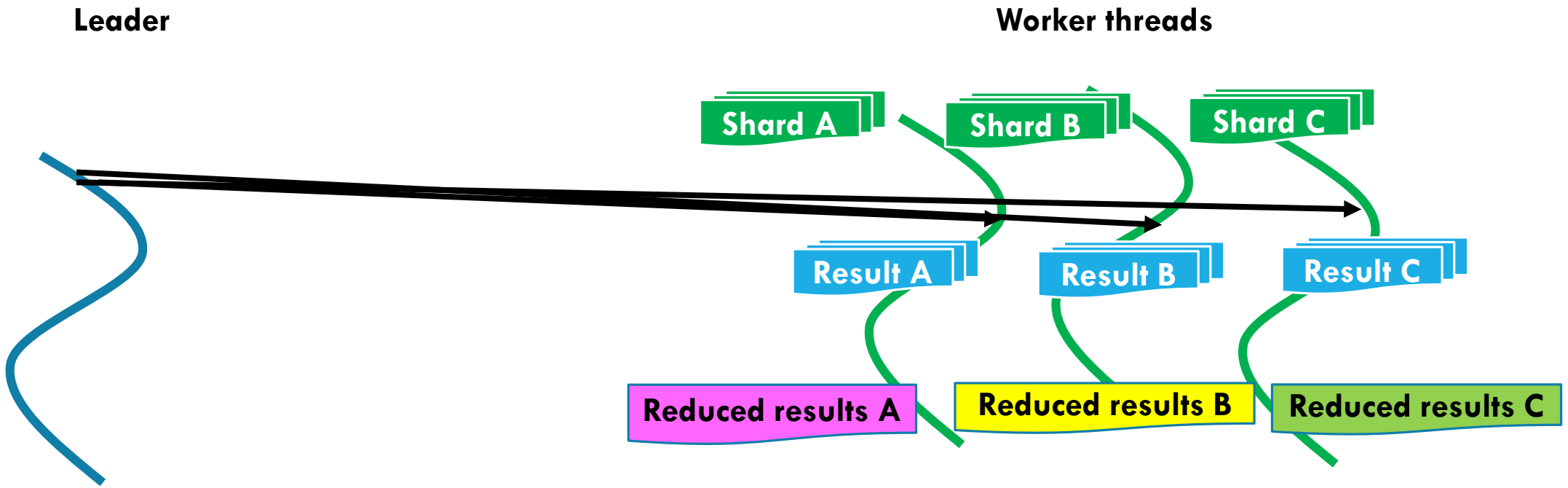
ALL-REDUCE PATTERN: SHUFFLE



ALL-REDUCE PATTERN: SORT



ALL-REDUCE PATTERN: REDUCE



MAP-REDUCE IS A COMPLEX PATTERN!

All-Reduce is hard to get "used to" but very powerful once you understand it and work with it.

Over the past ten years it has become the most widely used "tool" to create parallel systems for machine learning

Many algorithms can be expressed in terms of it

EXAMPLE: COUNT WORD FREQUENCIES

In the first step, each thread computes word frequencies in a subset (shard) of the input files.

In the shuffle step, each worker ends up responsible for part of the alphabet, based on the hash function.

In the reduce step, if a worker was sent multiple counts for the same word, it sums them to end up with one total per word.

EXAMPLE: MULTICORE SORTING

Map: Each worker scans its portion of the data, forming n "bins" (perhaps, using the hashing rule).

Shuffle: Each worker sends the k'th bin to the k'th worker.

Reduce: Each worker merges bins and sorts these intermediary results. We obtain sorted data spread over n workers.

GOALS OF THESE PATTERNS?

Use all the NUMA cores.

Keep workers busy on independent shares of some data set, or doing independent tasks. Ideally, there is no need for locking because they use distinct data, or only read shared data.

Tasks communicate through std::list or bounded buffers

SUMMARY

We are trying to work in stylized, familiar ways. Other developers who see your code will recognize the patterns.

These patterns aim for concurrent computing and sharing with as few locks as possible, to minimize overheads yet ensure correctness.

WE CAN NEVER ELIMINATE ALL THE LOCKS!

If we eliminate locks, NUMA memory consistency breaks.

This means: *Thread A might update X in memory, and then thread B might read X and see an old value.*

So… we can't *completely eliminate* the locks.

EVEN DISTRIBUTED SYSTEMS USE "LOCKS"

The ordered multicast pattern could arise inside a single C++ process that uses threads. We would implement it using locks.

But it could also arise between processes on different machines. Here, we would use a "distributed consensus protocol" to ensure fault-tolerant coordination for message order.

Same idea, but a different implementation

We use software design patterns to promote standard ways of building complex software systems.

We can also create standard coordination patterns, such as: *producer-consumer, leader-worker, ordered multicast, all-reduce.*

Each has a simple, elegant pattern. Implementations are complex… but we think about the pattern, not the way it was implemented!