COORDINATION PATTERNS

Professor Ken Birman

CS4414 Lecture 18

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?

Producers

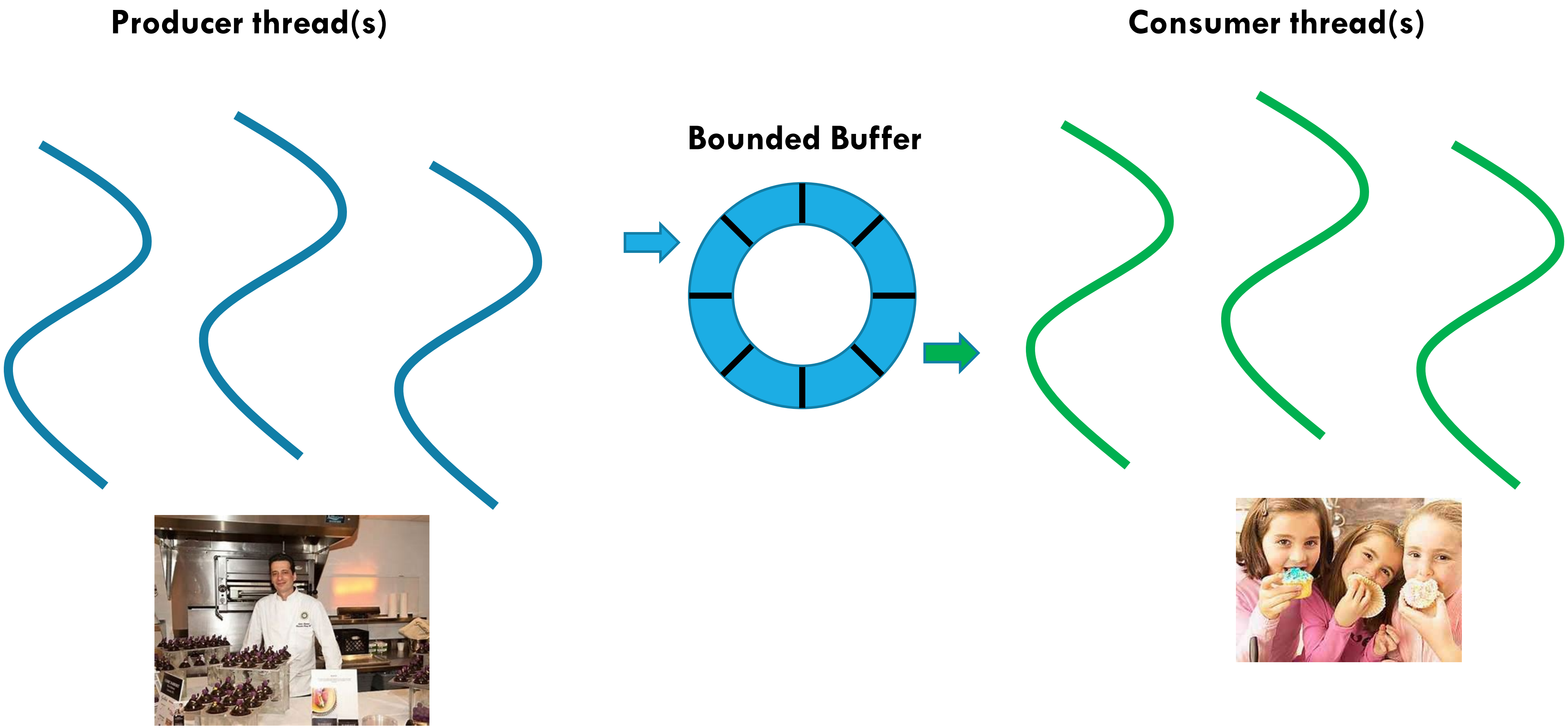
Consumers

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.

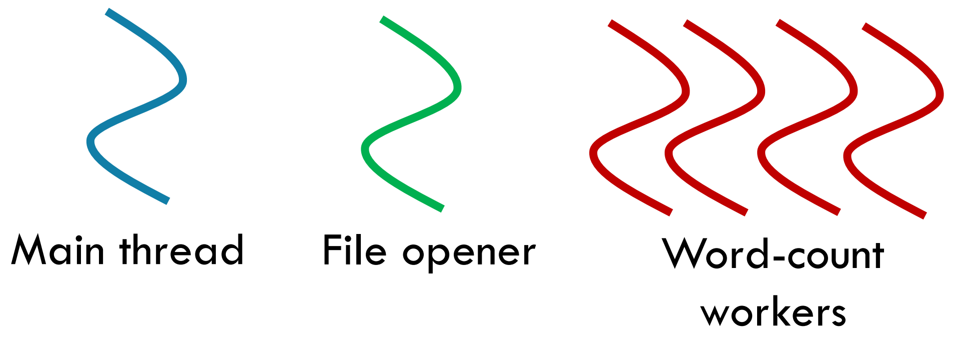
MODULARITY FOR COMPLEX, THREADED PROGRAMS

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.

CHOICES JUMP OUT AT US…

Should we share data, and protect it with locks, or monitors? Or even use atomic data types?

Should we avoid sharing data even at the cost of using more memory (for example, the extra copies of std::map objects used in Ken's word counter, one per thread)? This brings costs, too (in that example, the need to merge the sub-count trees at the end)

MORE EXAMPLES

Databases are integrated into modern systems (in PyTorch or TensorFlow you can just do SQL queries inline, and there is a way to do that in C++ too!)

… these bind to databases, and each one of those is a complex service that deals with concurrent users, scaling, prefetching, fault-tolerance

Yet the user doesn't see any of this complexity and is even unaware of the other concurrent users.

MORE EXAMPLES

Web servers at companies like Amazon, Facebook, Netflix

The Linux kernel

The C++ compiler

SOFTWARE DESIGN PATTERNS

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

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

EXAMPLE: THE OBJECT VISITOR PATTERN

The visitor 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.

With this visitor pattern, you attach a notifier to the directory those files live in. The file system supports this form of visitation but doesn't know who will use it in the future.

Each time a file changes, your program receives a notification.

HOW TO THINK ABOUT THE VISITOR IDEA

Consider a restaurant pager.

You place a take-out order but then can go outside or wander around.

When your food is ready the pager wakes up and this notifies you. No need for the food preparer to do anything more.

VISITOR PATTERN USE CASES

The visitor pattern is common with file systems: it allows an application with files open to "refresh" if a change occurs.

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

KEY ELEMENTS OF THIS PATTERN

The visitor is an object in some application written *after* the service was created. The application object derives from (implements) the notification interface.

The visitor "interface" is a fully virtual class with an event notification method. The service will treat application objects as instances of visitor objects

The service doesn't know what objects will be using it, but can use the interface class to notify "future customers".

KEY ELEMENTS OF THIS PATTERN

Visitor Interface is a base class

The application class derives from (implements) the base class, but extends it with other application functionality

WHY GIVE THIS PATTERN A SPECIAL NAME AND THINK OF IT AS A STANDARD?

Visitor is a well known pattern and even taught in courses on software engineering. We sometimes teach it in CS2110

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?

By allowing modules to standardize the way that they coordinate and interact, patterns bring a uniform way to create bigger systems from modular components.

The monitored module doesn't know who will monitor it, but does know how to notify those future watchers.

The watchers simply need to implement the required interface

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

WHY A FACTORY IS USEFUL

… without the factory, this same coordination is quite hard!

By allowing B to offer a factory that creates "widget objects" A has a way to ask B to create new B objects (derived from widget) and yet A doesn't even know the definition of class B.

B simply needs to implement the factory interface

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.

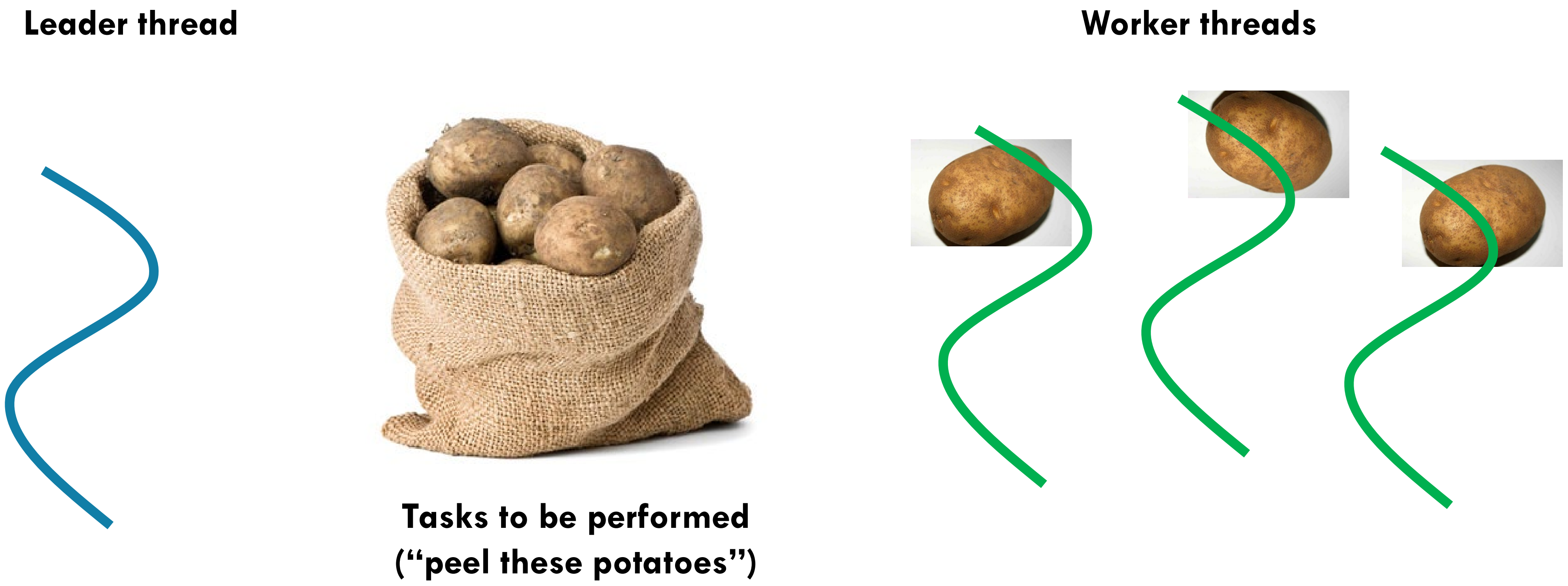
SYNCHRONIZATION PATTERNS

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 / WORKERS PATTERN

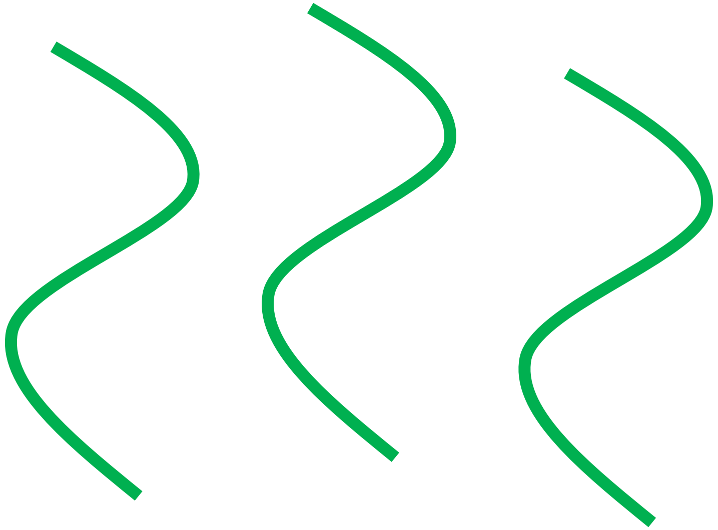


LEADER / WORKERS PATTERN

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”).

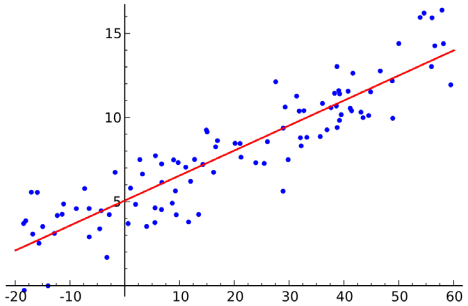
It is easy to test to see if the list is empty.

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.

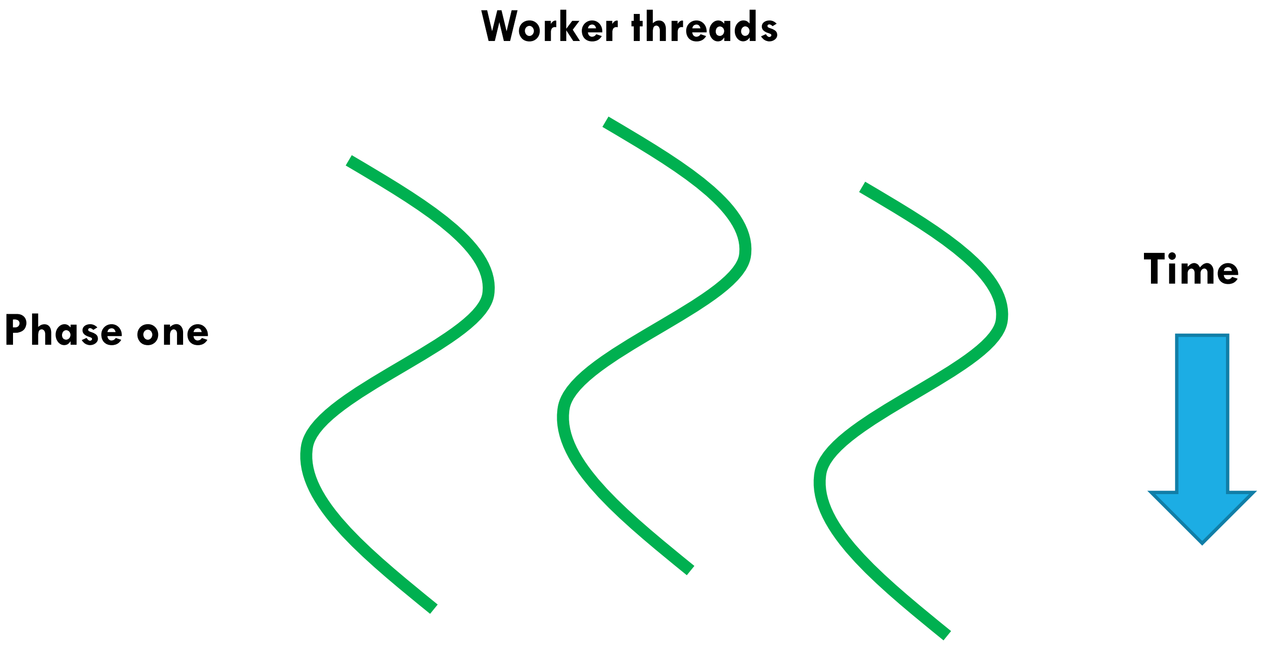
The threads all call "barrier_wait". This method uses a bool array to track which threads are ready, initialized to all false.

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.

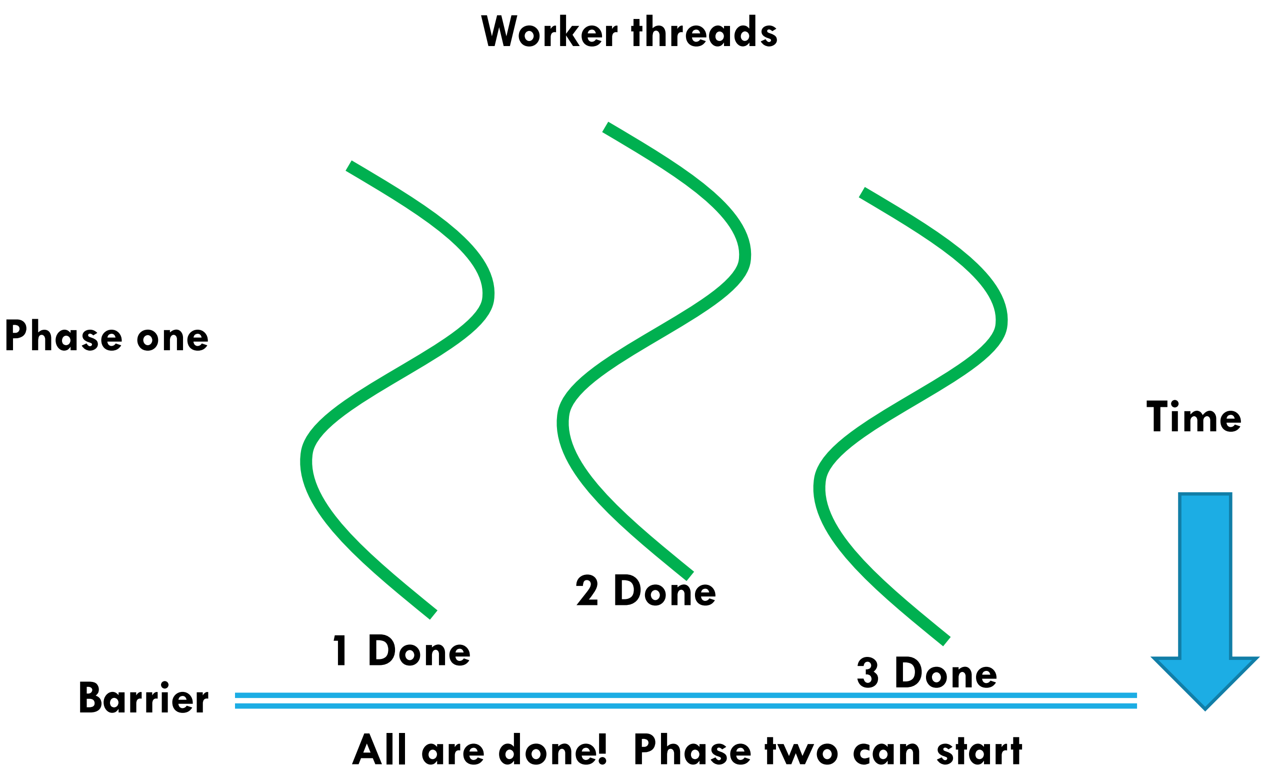
After phase one, all workers must wait until phase two starts.



BUILDING A BARRIER

Example: A computation with distinct phases or epochs.

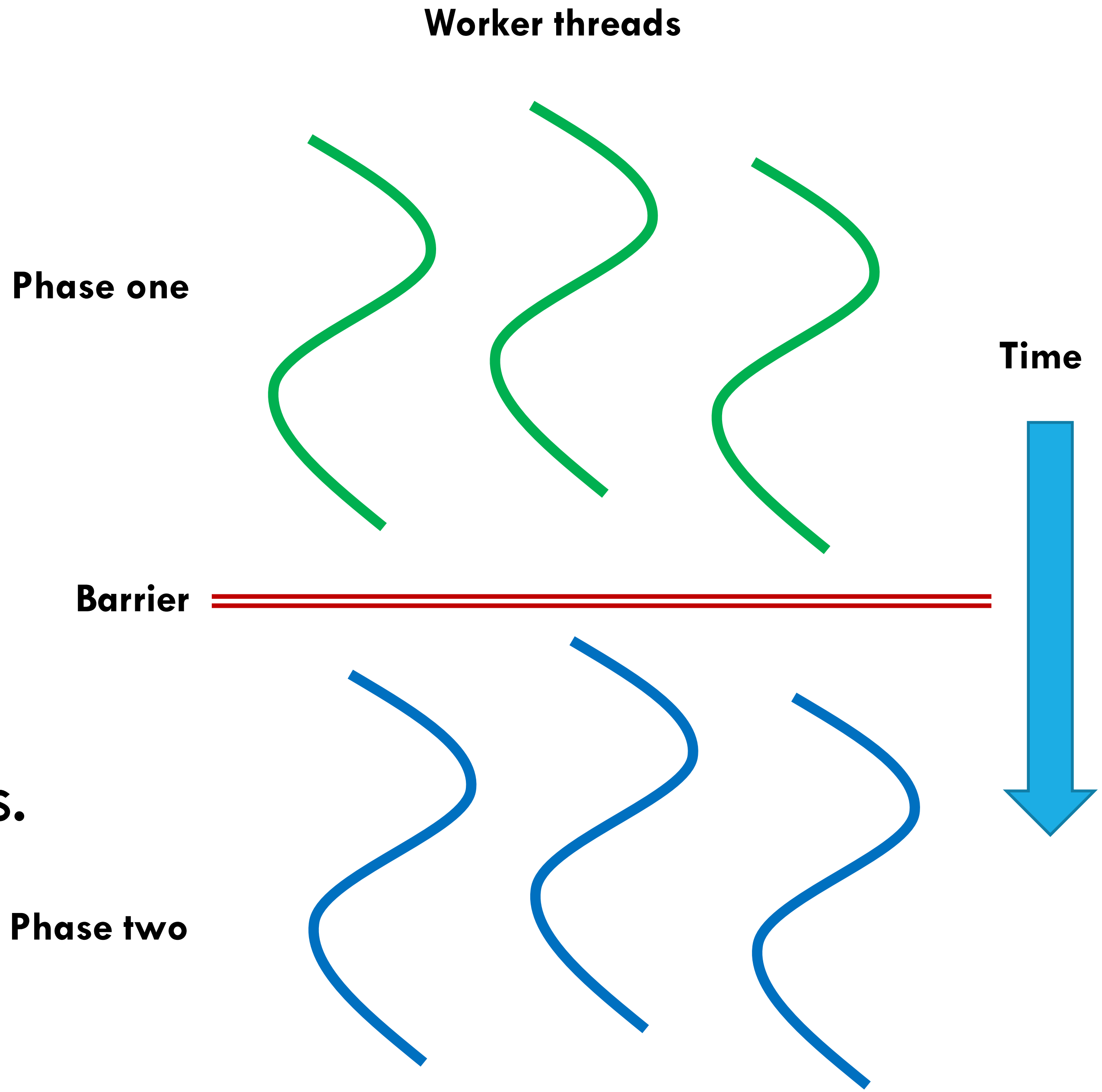
After phase one, all workers must wait until phase two starts.



BUILDING A BARRIER

Example: A computation with distinct phases or epochs.

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



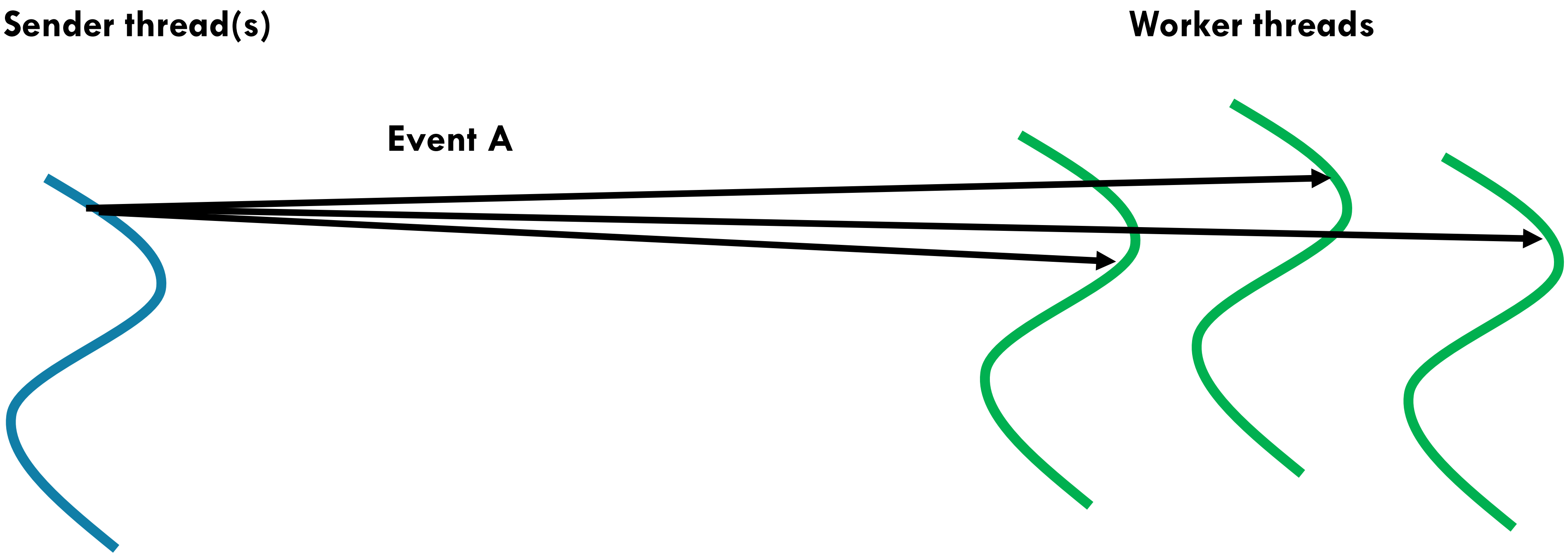
ORDERED MULTICAST PATTERN

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

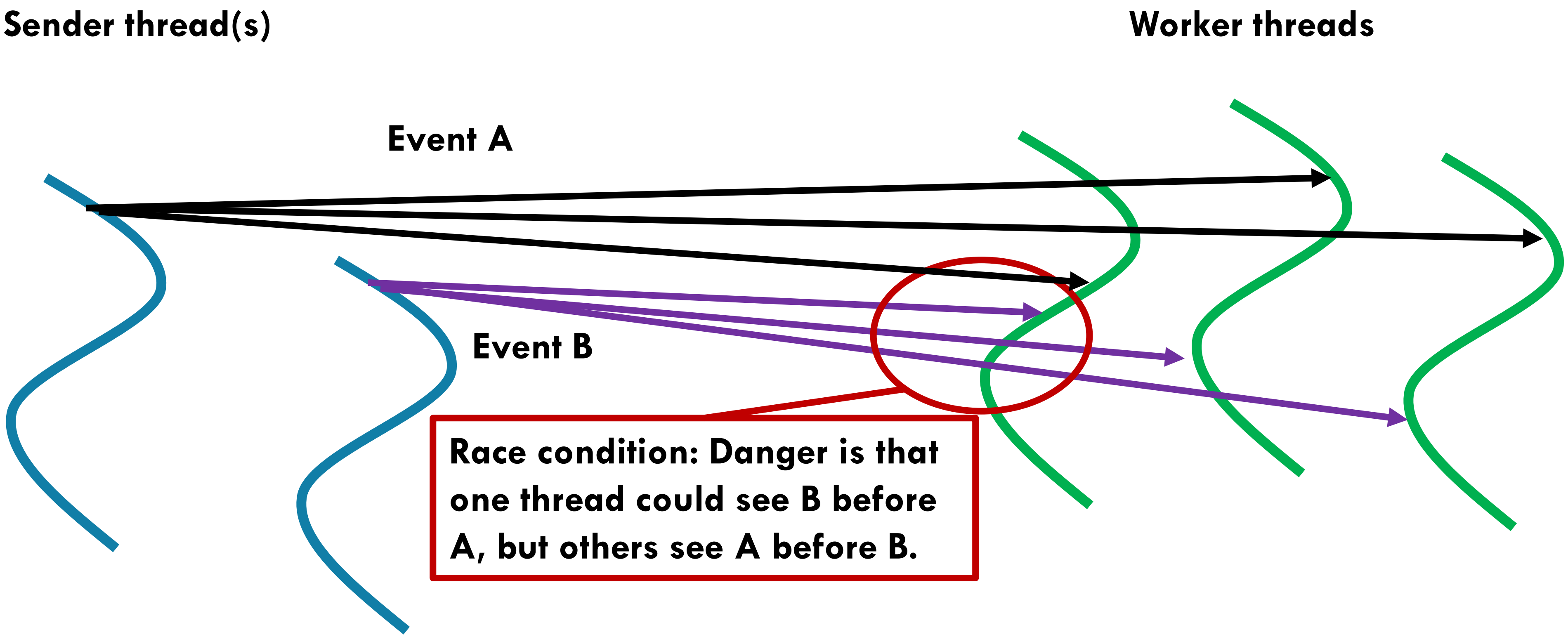
A sender thread needs every worker to see an object describing the event, so it puts that object on every worker's work queue.

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.

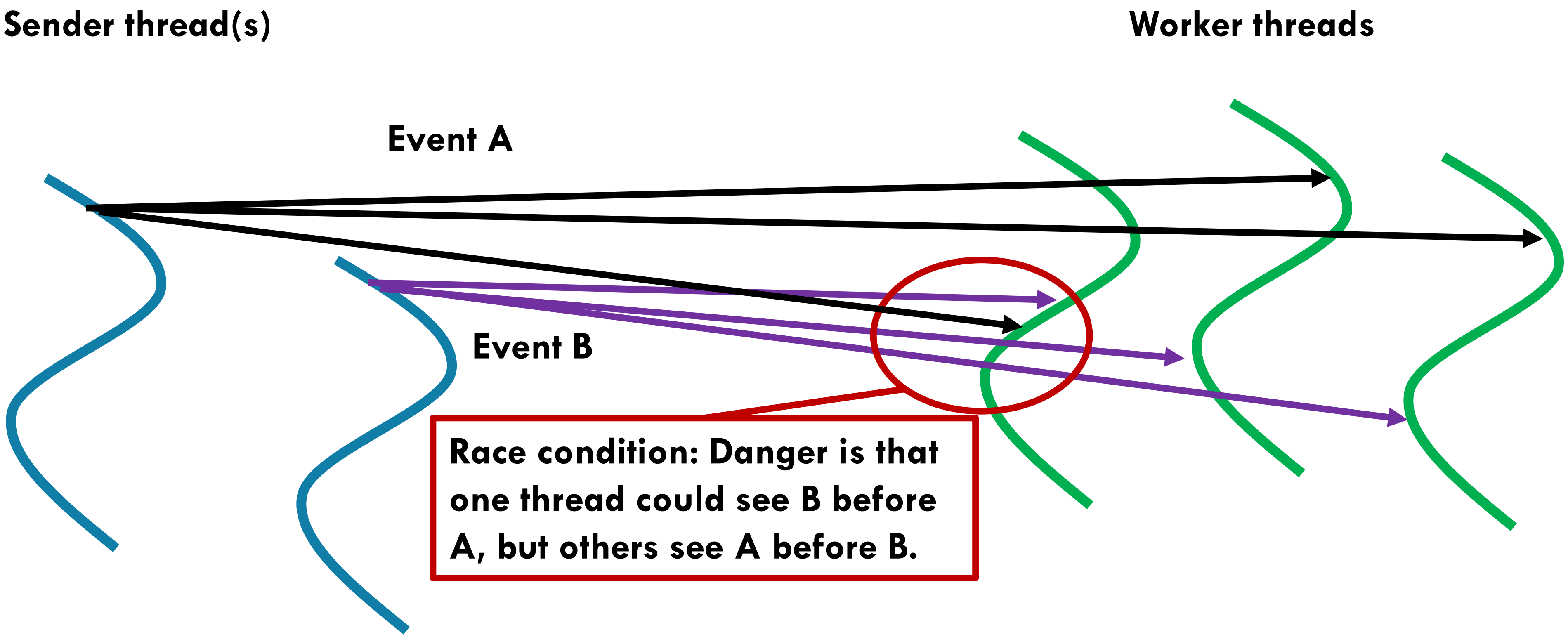
ORDERED MULTICAST PATTERN



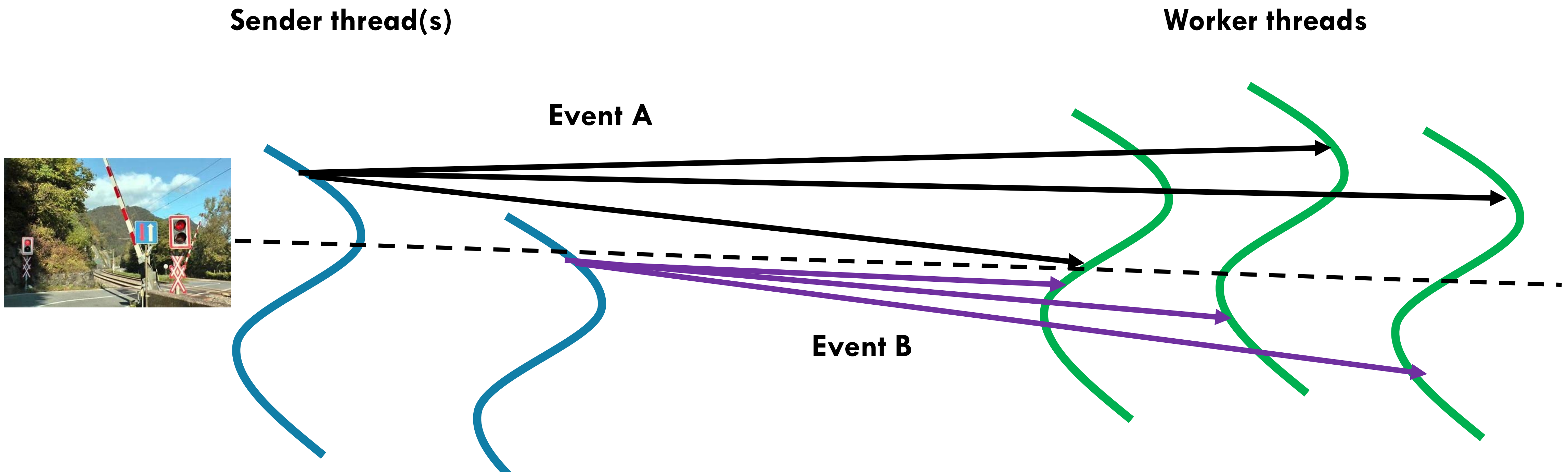
ORDERED MULTICAST PATTERN



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.

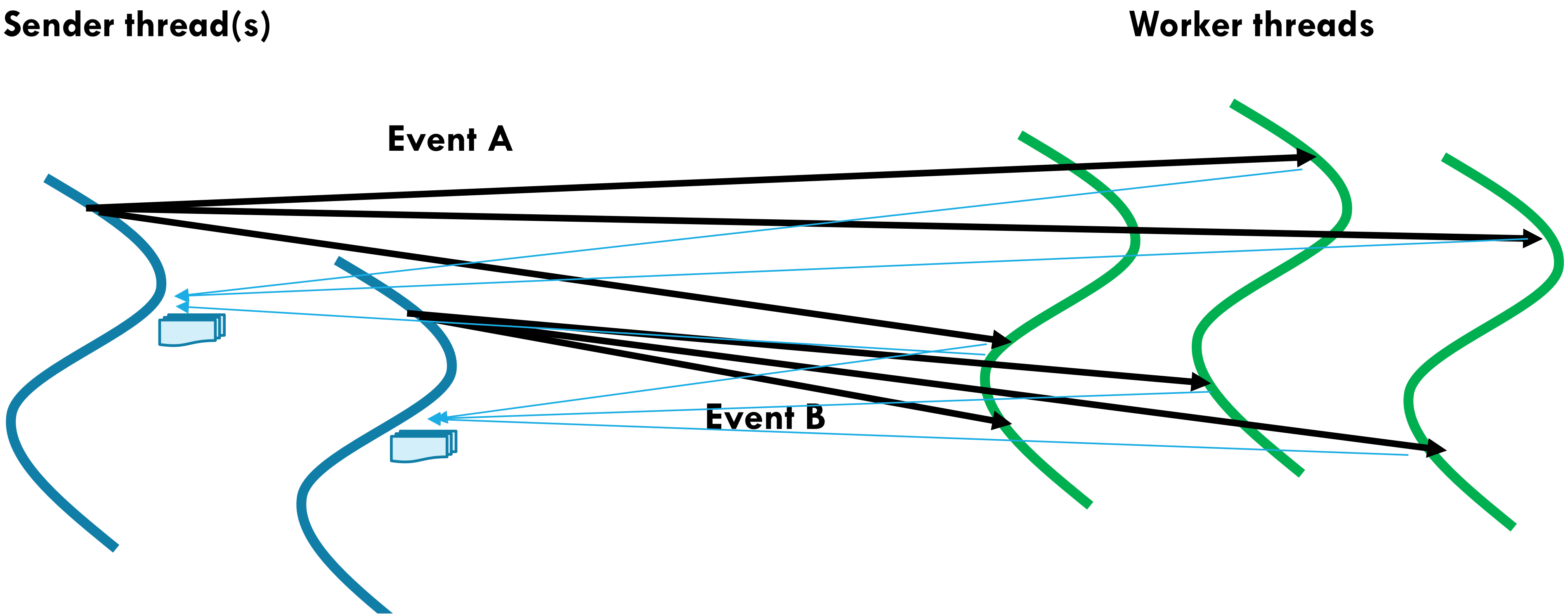
ORDERED MULTICAST WITH REPLIES

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.

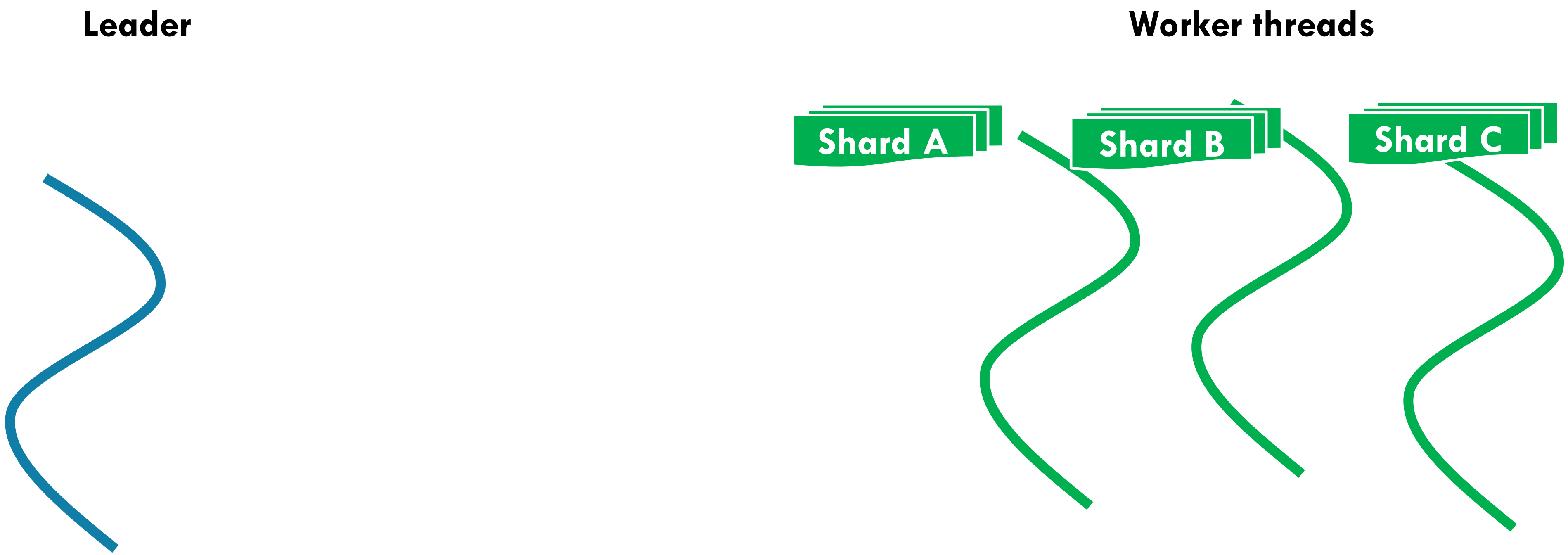
ALL-REDUCE PATTERN: IMPORTANT IN ML.

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



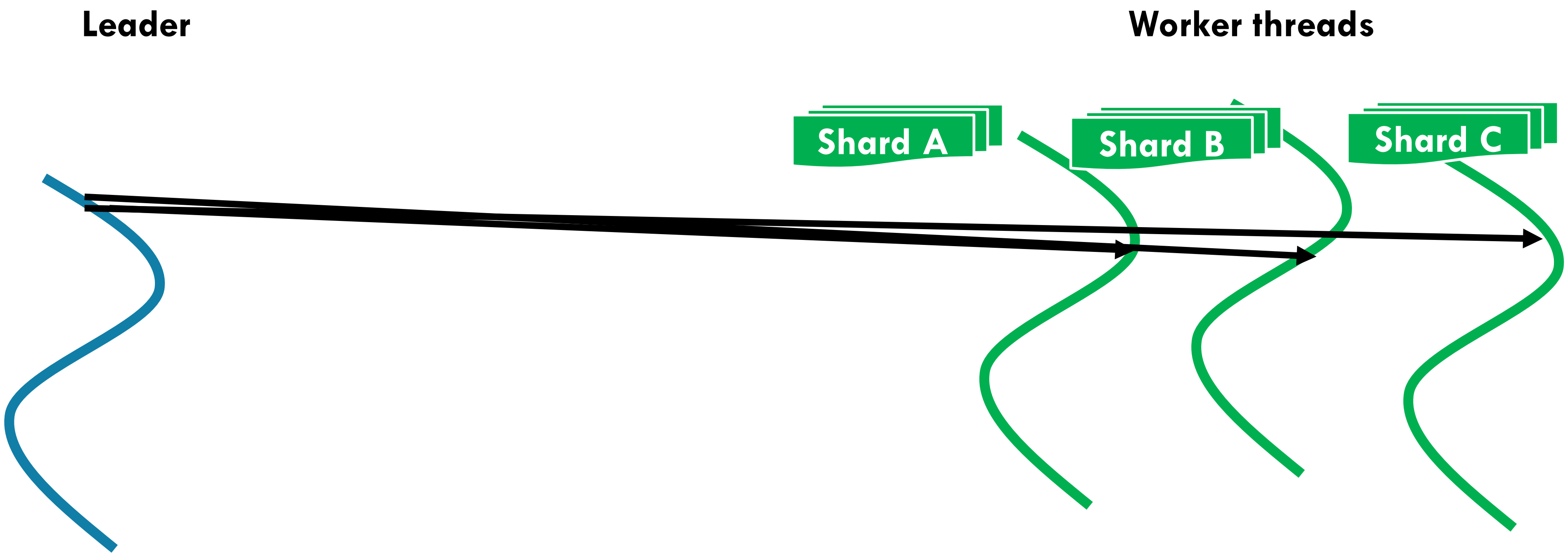
ALL-REDUCE: MAP STEP

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

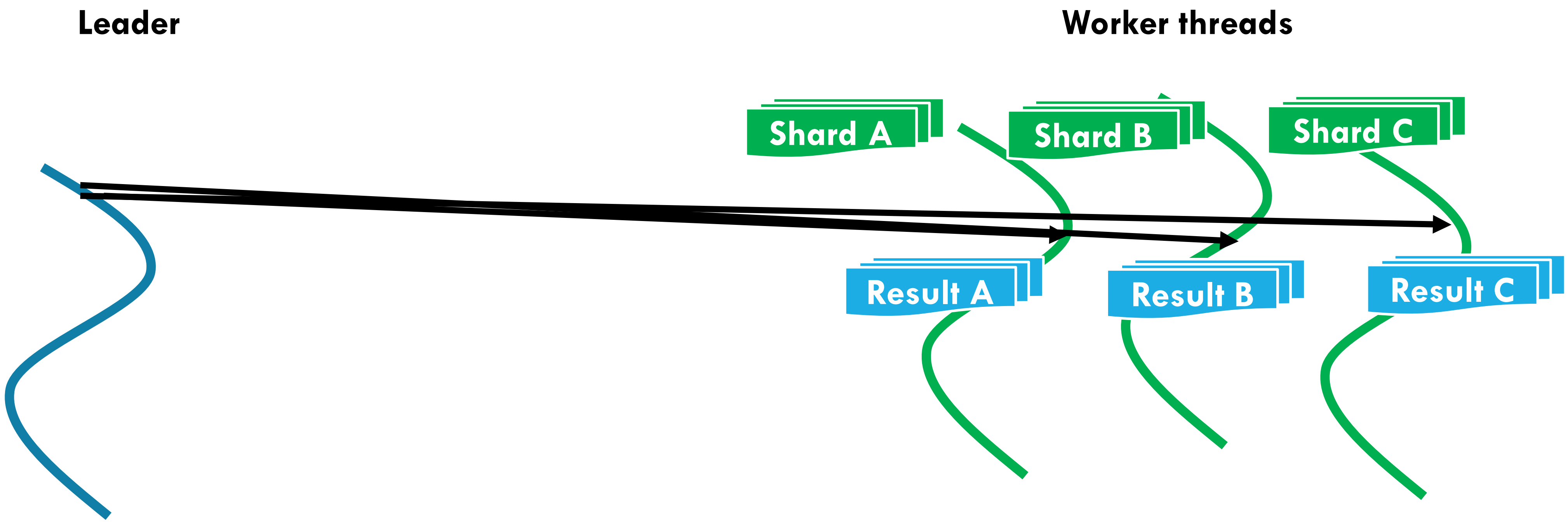
Each worker performs its share of the work by applying the requested function to the data in its shard.

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

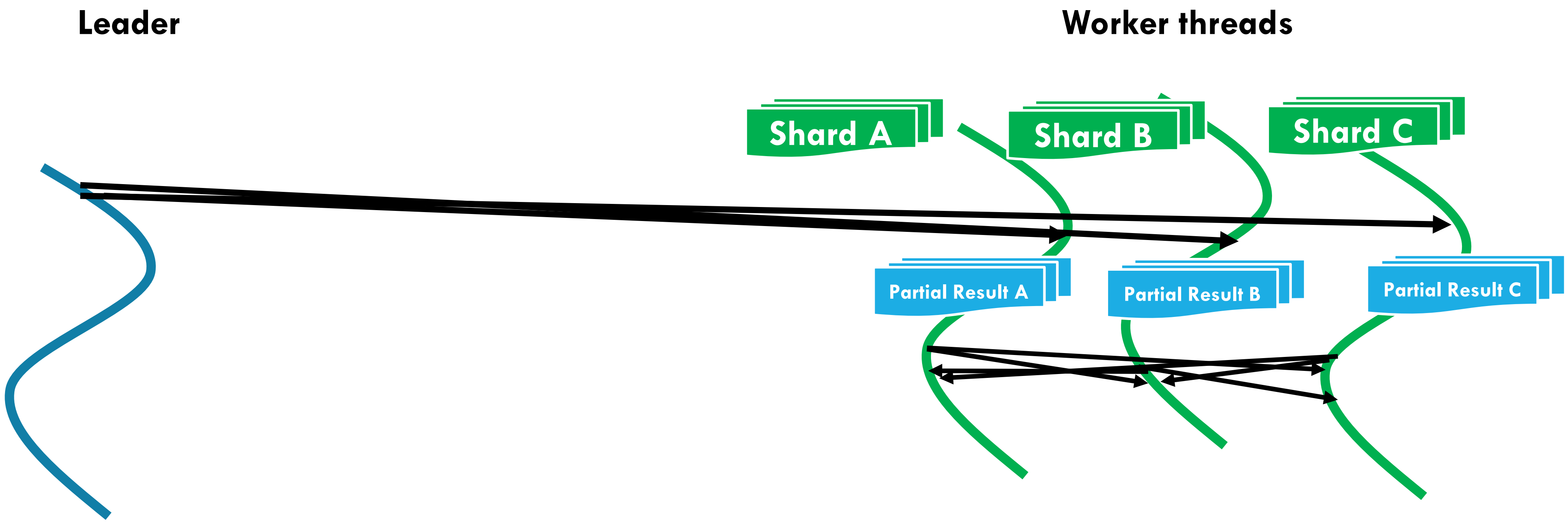
ALL-REDUCE PATTERN: MAP (FIRST STEP)



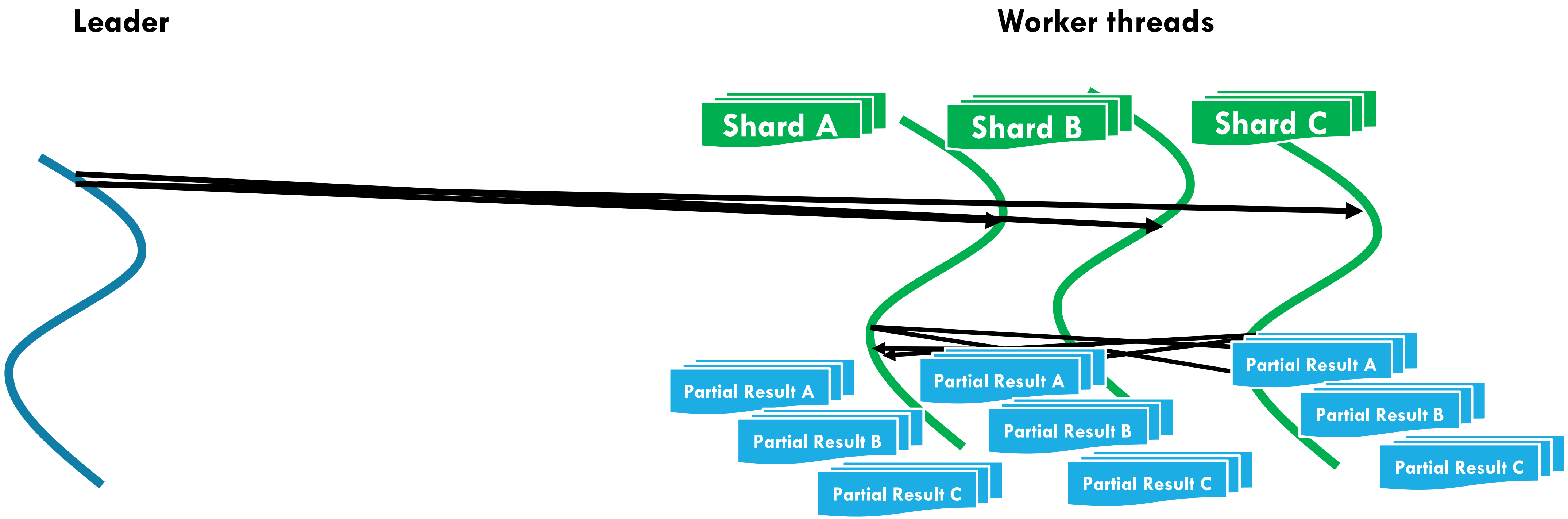
ALL-REDUCE PATTERN: MAP (FIRST STEP)



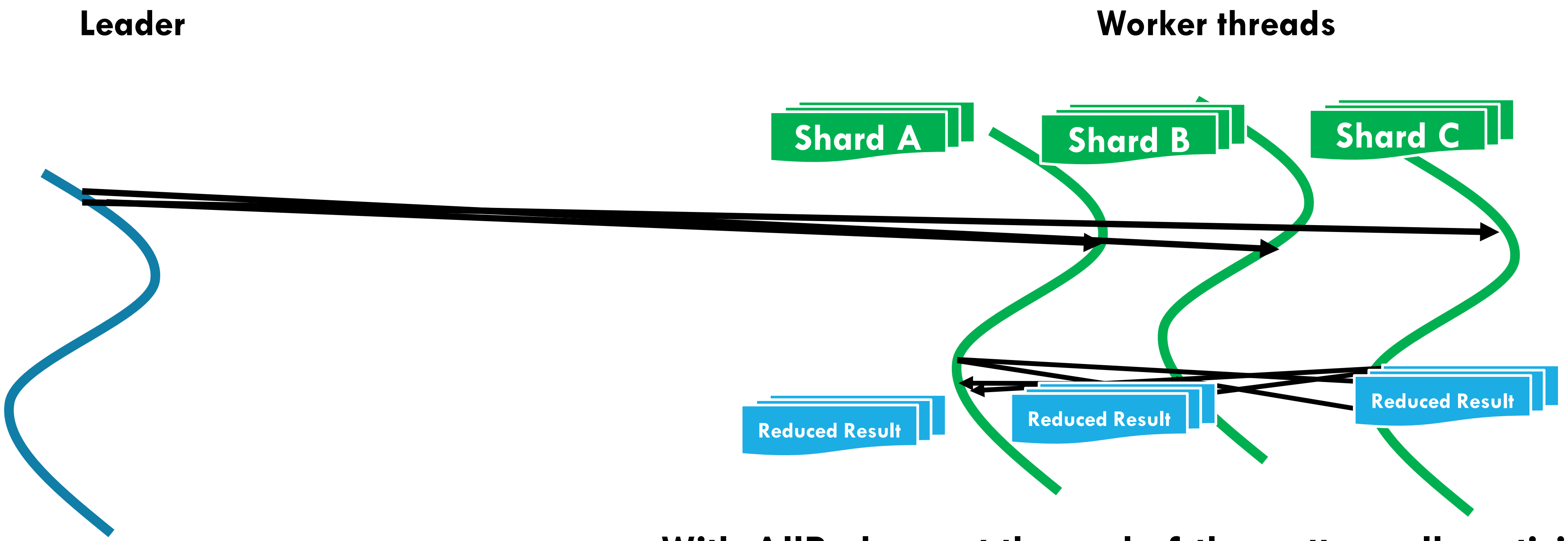
ALL-REDUCE PATTERN: SHUFFLE



ALL-REDUCE PATTERN: SHUFFLE

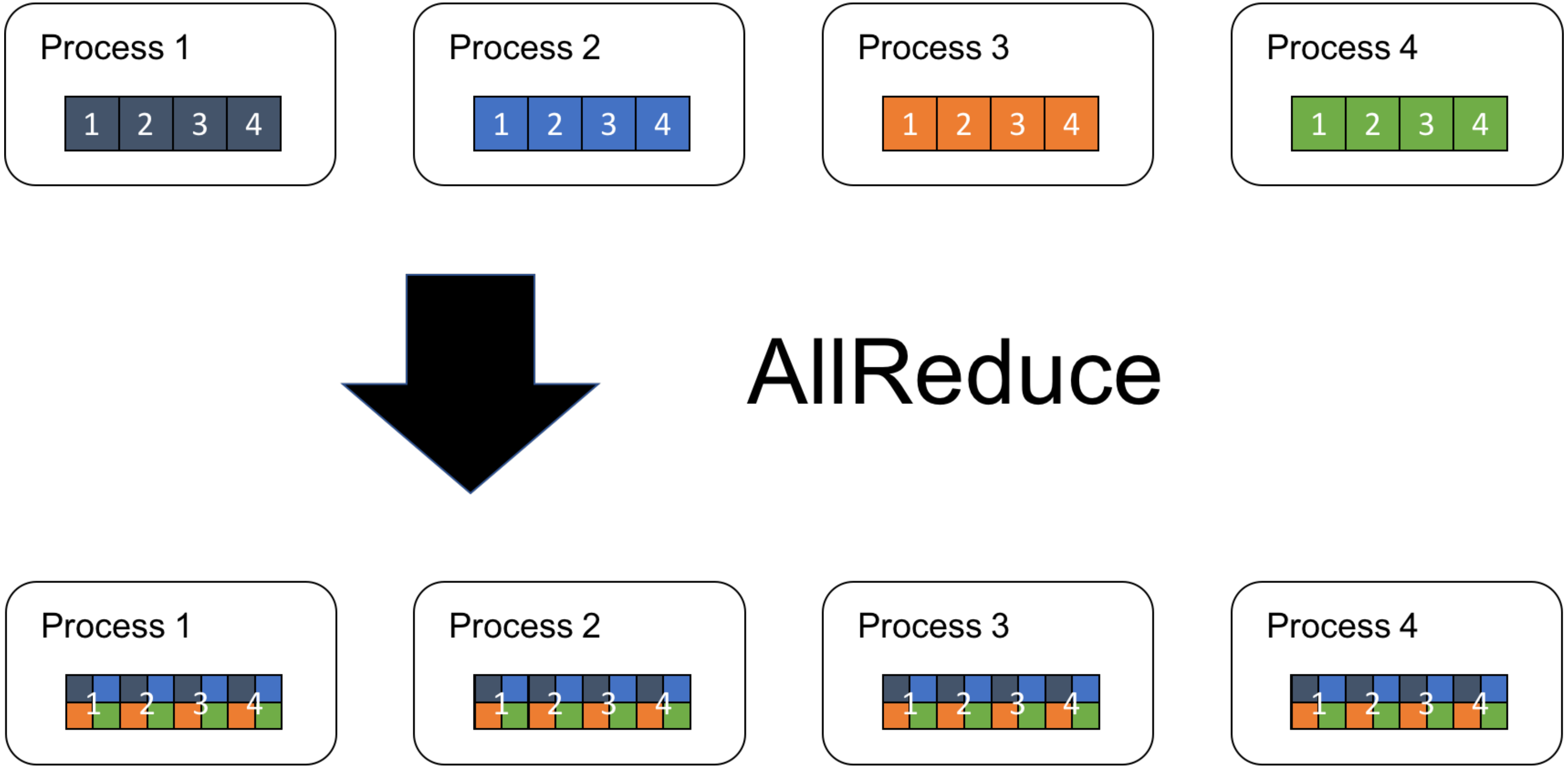


ALL-REDUCE PATTERN: SHUFFLE



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: MAP STEP RESULTS ARE VECTORS OF INTS. REDUCE MIGHT SUM THE INTEGERS



MAP-REDUCE IS AN EVEN MORE COMPLEX PATTERN!

With Map-Reduce, each worker ends up with a *distinct share* of the results. Data is "spread out" at the start and at the end. Useful if the final result would be too big to hold on a single computer.

Instead of a set of all-to-all broadcasts, MapReduce uses point-to-point messages: worker1 sends data intended for worker2 only to worker2, etc.

We won't show this on a slide, but hopefully you get the idea

GOALS OF THESE “COLLECTIVE COMMUNICATION” (CCL) 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 (1)

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.

SUMMARY (2)

We illustrated use of software design patterns as conceptual tools (abstractions) to promote standard ways of building complex software systems.

Common coordination patterns include: *producer-consumer, leader-worker, ordered multicast, all-reduce, other CCL library packages.*

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