Java byte by byte

Java features explained with examples

Java phasers made simple

By Francisco Alvarez | 3rd February 2018 5 Comments

I recently came across Java phasers while reviewing different constructs for inter-thread communication. Phasers are associated to the concepts of **fuzzy barriers** and **point to point synchronisation** which are all well explained on this parallel programming course.

As to this post, I will borrow the example of the one-dimensional stencil to explain very intuitively the motivation behind the concept of phaser.

The code corresponding to all the examples in this post can be found on this repository

One-Dimensional stencil

In order to justify the need for phasers, we will start by demonstrating the use of other synchronisation constructs to solve the computational problem of the one-dimensional stencil.

Given a one dimensional array A with N elements, the value of each element is calculated according to this function:

$$A_n = \begin{cases} k & \text{if n=0 or n=N-1} \\ (A_{n-1} + A_{n+1})/2 & \text{in any other case} \end{cases}$$

In other words, the value of the first and last elements is fixed (these values do not need to be the same though), whereas the value of each of the remaining N-2 elements is the average of the value of its neighbours. This is a recursive process: as the value of its neighbours change, each element needs to re-compute its own value. Ideally, after a few iterations, the value of all the elements should converge and the recursive iterations stop.

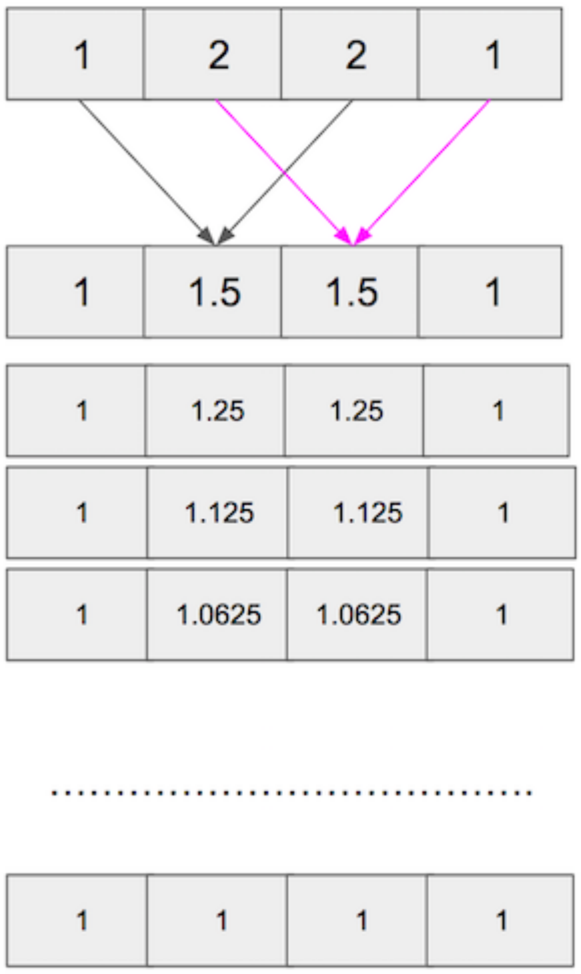
Just a brief digression for those curious about Physics: the aforementioned Wikipedia link says t

"stencil codes are most commonly found in the codes of computer simulations". No wonder! Yo

Manage

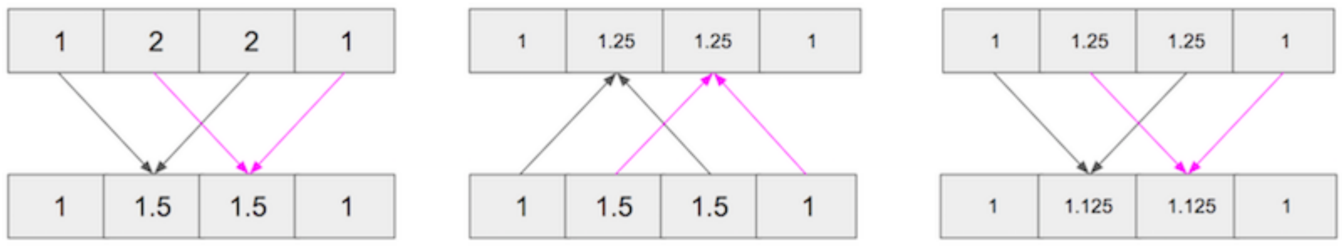
> *picture the one-dimensional stencil as a metal bar connected to a heat source by its sides. The temperature of the sides remains constant whereas the temperature along the bar changes over time until reaching thermal equilibrium.*

To get some intuition about the stencil problem, let's work through an example. To make it simple, consider an array of length 4 whose first and last elements have a fixed value 1. The remaining elements have an initial value 2. Therefore, the initial array is [1,2,2,1]. After the first iteration, the array becomes [1,1.5,1.5,1]. The following diagram shows how the values change on each iteration and how, eventually, the values converge to 1.



stencil algorithm

Implementation detail: in order to avoid creating a new array on each iteration, the implementation of the algorithm can use just 2 arrays: one containing the values at the beginning of the iteration and other containing the resulting values at the end. Then, for the next iteration, the arrays are flipped. This diagram illustrates the process:



stencil algorithm simplified

And the below snippet is the sequential implementation of the algorithm (the code snippets presented in this post are written in Scala, although any Java developer can easily read them):

```
/**
  * Sequential version
  * @param iter Number of iterations
  * @param arr Array with fixed boundary values. The other values are calculated as the av
  * @return Array with the final values after completing all iterations
  */
def seq(iter: Int, arr: Array[Double]): Array[Double] = {
  //defensive copy to keep arr from being mutated outside this method
  var myVal = arr.clone()
  var myNew = arr.clone()

  for(j <- (1 to iter)) {
    for (i <- (1 until arr.length-1)) {
      myNew(i) = (myVal(i - 1) + myVal(i + 1)) / 2.0
    }
    val temp = myNew
    myNew = myVal
    myVal = temp
  }
  myVal
}
```

Parallel implementations

Working on top of the sequential solution, this section describes other approaches to solve the problem using different parallel-programming constructs: **Latch, Cyclic Barrier and Phasers**.

Latch

According to CountDownLatch, a latch is *"A synchronization aid that allows one or more threads to wai*
set of operations being performed in other threads completes."

The solution to the stencil problem with latches involves the following steps:

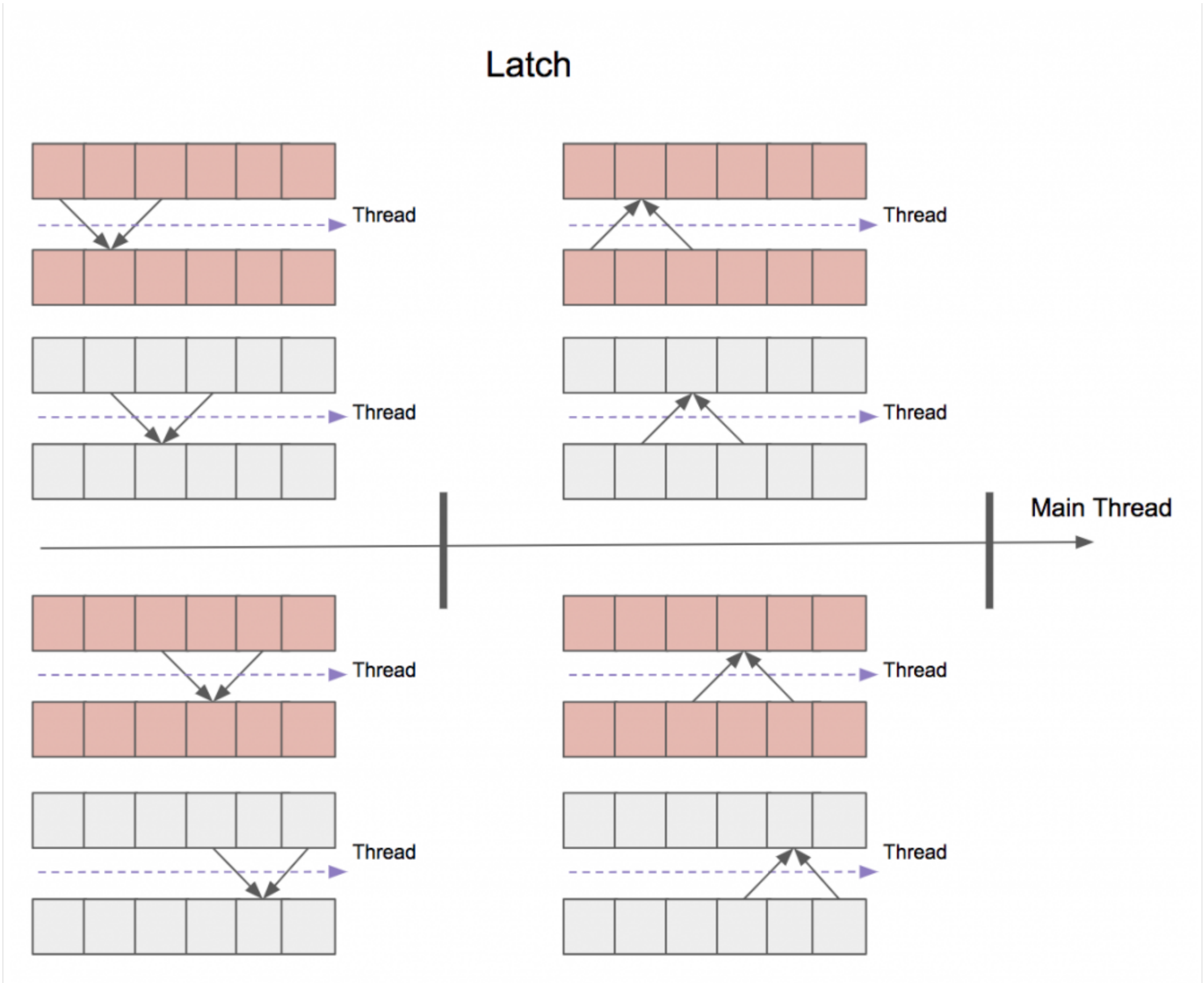
- create a new latch for each iteration with a count equals to N-2
- create N-2 tasks to be executed in parallel by threads of a dedicated execution context while the main thread awaits until all tasks are completed; each task calculates the value of one element of the array
- once a task is finished, the last action of the executing thread is to let the main thread know that the the task is completed
- when all tasks are terminated, the main thread resumes its activity by flipping the old and new arrays and starting a new iteration

It is worth noting that:

- the communication through the latch happens between each of the threads of the dedicated execution context and the main thread
- once a task is terminated, the executing thread becomes available to execute another task; so it is fine to have a thread pool with less threads than tasks
- the latch represents a barrier to the execution of the main thread only

The following diagram represents 2 iterations of the process just described:

Manage



latch

Cyclic Barrier

From the definition of CyclicBarrier, a cyclic barrier is *"A synchronization aid that allows a set of threads to all wait for each other to reach a common barrier point"*.

Here are the main characteristsics of this solution:

- each of the N-2 tasks calculates the value of one element of the array from beginning to end, that is, through all the iterations
- these tasks are not independent as the values of the neighbours that each task needs to calculate its own value are calculated by other tasks
- therefore, before moving on to a new iteration, each task needs to wait for all others to complet

This problem seems a good match for a cyclic barrier and comprises the following steps:

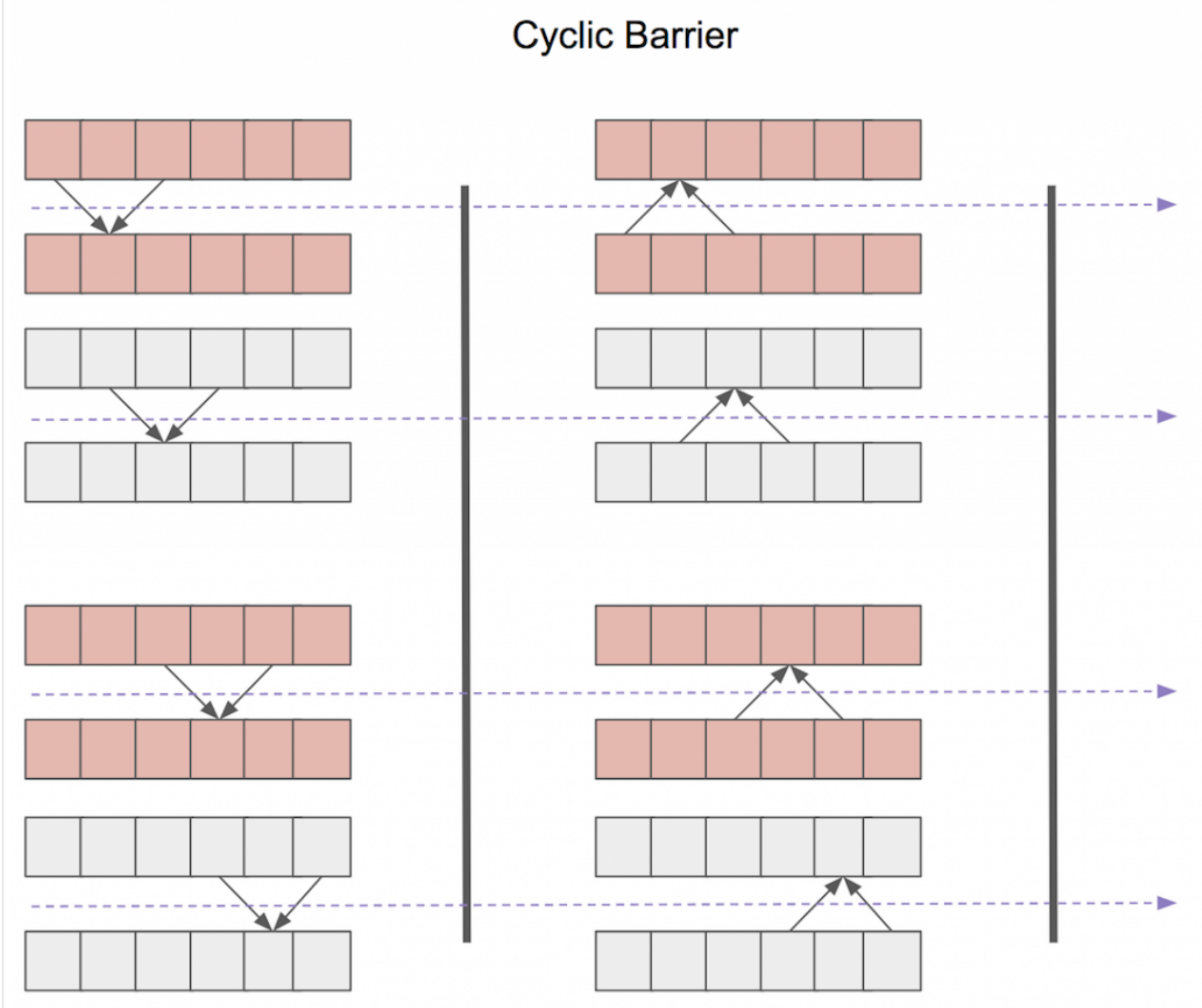
Manage

- create a cyclic barrier with N-2 parties
- create N-2 tasks to be executed in parallel by threads of a dedicated execution context, each task calculates the value of one element of the array through all iterations
- at the end of each iteration, each thread awaits at the barrier until all other threads reach it.
- finally, the arrays are flipped and a new iteration begins

The main features of this algorithm are:

- the communication through the barrier happens among all threads of the dedicated execution context
- the cyclic barrier represents a common barrier for all threads
- no thread is released after each iteration, so the thread pool must have at least as many threads as tasks (N-2)

The below diagram represents 2 iterations of the process:



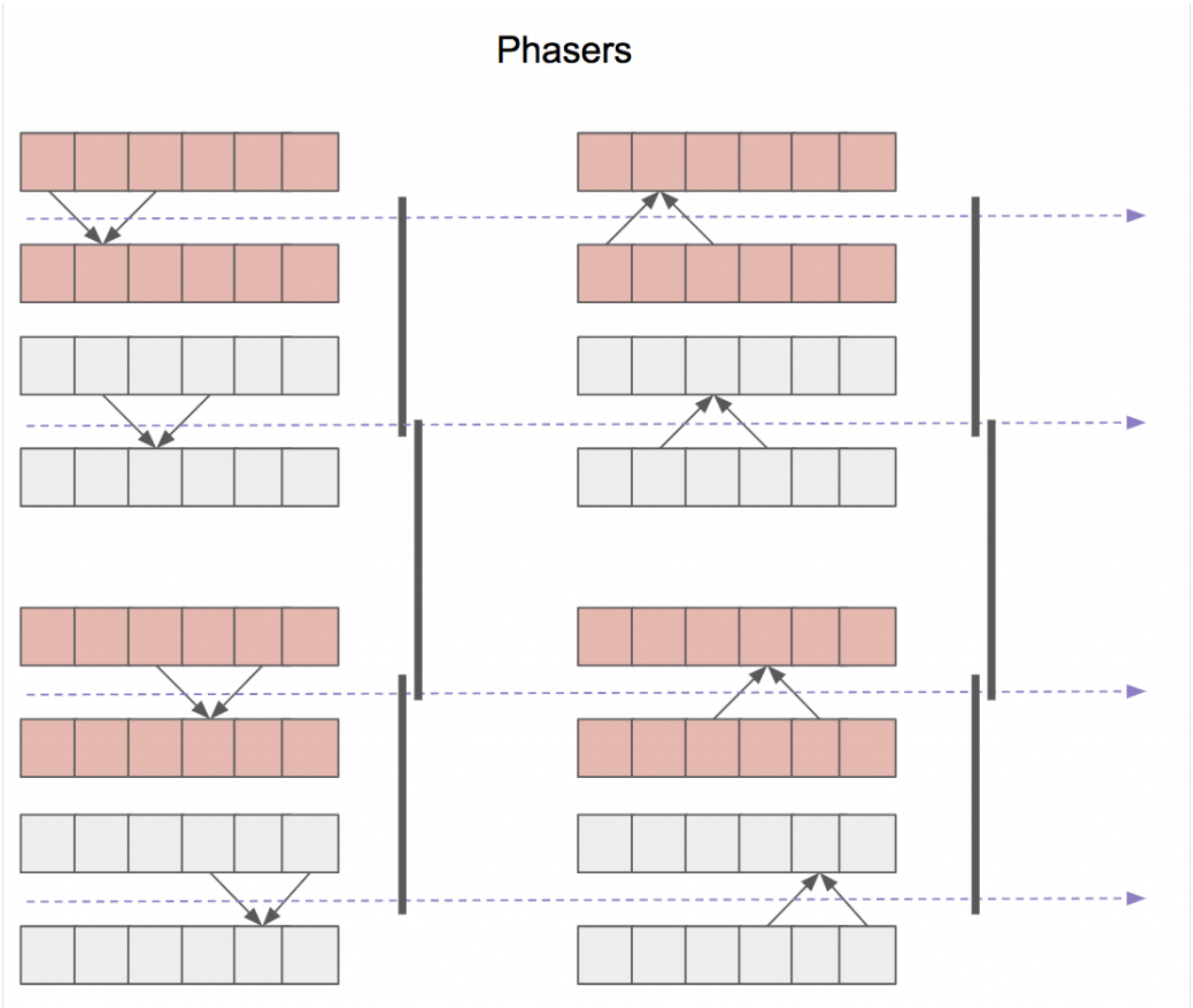
cyclic barrier

Phasers

Phasers allow to reduce the synchronisation contention cost when there is a large number of parties. In order to understand this, let's revisit the cyclic barrier example: we said that *"before moving on to a new iteration, each task needs to wait for all others to complete"* and as a consequence there is a common barrier for all threads.

Well, the above statement is not completely true. Strictly speaking, each task needs to wait only for the adjacent tasks that compute the value of its neighbours. Therefore, the situation is better described by the following diagram, where each thread shares a barrier with the adjacent ones. This fine-grained synchronisati achieved through the use of phasers. The implementation details can be found on the source code.

Manage



phasers

A final twist: fuzzy barriers

When dealing with cycling barriers and phasers, it was mentioned that the number of threads needed is N-2. For large arrays (think of millions of elements), this is not feasible. You will get the dreaded

```
An exception or error caused a run to abort: unable to create new native thread
java.lang.OutOfMemoryError: unable to create new native thread
 at java.lang.Thread.start0(Native Method)
 at java.lang.Thread.start(Thread.java:714)
 at java.util.concurrent.ThreadPoolExecutor.addWorker(ThreadPoolExecutor.java:950)
 at java.util.concurrent.ThreadPoolExecutor.execute(ThreadPoolExecutor.java:1357)
```

And even if it were possible to create so many threads, the operating system would struggle to handle them all with a limited amount of CPUs.

In order to get around this issue, the elements of the array can be grouped and then create a task for each group. Each task will calculate the values of all the elements of that group in a sequential fashion and the creation and synchronisation of barriers will happen at group level.

As discussed in the previous section, the different approaches could be:

- **CyclicBarrier**: in order to progress to the next phase, all the elements of each group must be calculated
- **Phasers**: to move to the next phase, a thread only needs to wait for the elements of the adjacent groups to be calculated

Let’s take a closer look at the last point. In reality, a thread only needs to wait for the **first/last element** of the adjacent groups to be calculated. Therefore, each thread can:

- firstly, calculate the first and last elements of its group
- signal to the other threads that the first/last elements are calculated (this is interpreted like the thread reaching the first point of the synchronisation barrier) and therefore those other threads can progress to the next phase (if they are ready to do so)
- calculate the internal elements of its group (which are not depended upon by the other threads)
- await, at the second point of the barrier, for the threads of the adjacent groups to reach the first point of the barrier

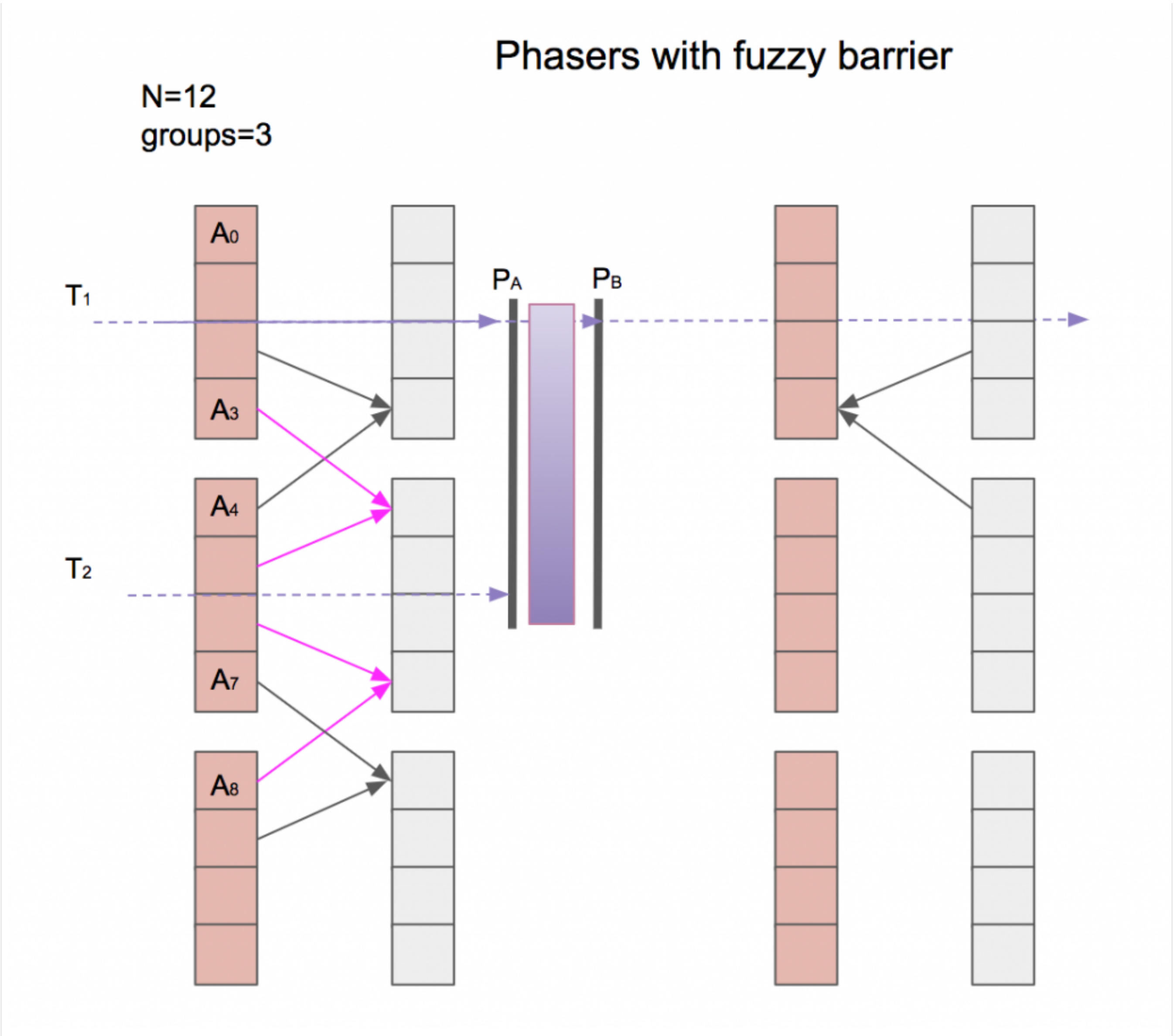
This is called Phasers with **split-phase barrier** or **fuzzy barrier** . This capability to perform local work within the barrier is another of the main characteristic of the phasers together with the capability to make fine-grained synchronisation.

The following diagram illustrates the idea of fuzzy barrier. The diagram represents a 12-element array divided in 3 groups of 4 elements. To keep things simple, we will focus on the barrier shared by the first and second groups.

Each thread T_x calculates the value of all elements in that group, starting with the values on the sides.

For instance, T_1 calculates A_3, reaches P_A and signals it out to the other threads. Then it proceeds to calculate the rest of the elements in the group and, when it is done, waits on P_B until T_2 reaches P_A.

When T_2 has calculated A_4 and A_7, it reaches P_A and emits the corresponding signal. At this point, T_1 can progress to phase 2 while T_2 completes phase 1 by calculating the rest of the elements of its group, and so on.



fuzzy barrier

Conclusion

The journey from the latch solution to the phasers one has led us to reduce the synchronisation contention costs by means of:

1. changing from CountDownLatch to CyclicBarrier to swap the iterative actions of the algorithm: `for each iteration { for each array element { ... } } -> for each array element { for each iteration { ... } }`
2. changing from CyclicBarrier to Phasers to break up the large common synchronisation barrier int ll individual ones
3. grouping elements into groups that are processed in parallel (while the content of each group i cessed sequentially): `for each group {for each iteration { for each array element {`

4. introducing fuzzy barriers to perform local work inside the barrier and therefore further reducing the contention

The code for all the examples discussed on this post can be found on this repository

Category: Uncategorised Tags: concurrency, cyclic barrier, fuzzy barrier, latch, parallel programming, phaser

5 thoughts on "Java phasers made simple"

Barry S
19th July 2020

Your explanations were quite helpful. I was able to take your scala code, translate it to Java 8 and produce the correct results. Great job.

I computed the run time for the group methods (iter = 1000000 and g = 25). The fuzzy barrier was just a bit more performant than the phasers method, but was significantly faster than the cyclic barrier method.

parWithCyclicBarrierAndGroups: 96773
parWithPhasersAndGroups: 15370
parWithFuzzyBarrier: 15072

Timi A
8th January 2020

This has been really helpful to read, but the diagrams are not loading on the page. I think they'll make it even clearer.
Is this something that you can look into please? Especially for the phaser part.

Francisco Alvarez Post author
8th March 2020

Manage

Sorry for that. It’s fixed

Dave Stibrany
19th August 2018

The section “A final twist: fuzzy barriers” really helped me understand the point of separating arrive() and awaitAdvance(). Thanks for that!

Francisco Alvarez Post author
19th August 2018

I’m glad it was helpful

This site uses Akismet to reduce spam. Learn how your comment data is processed.