**Fork/Join**

The fork/join framework is an implementation of the ExecutorService interface that helps you take advantage of multiple processors. It is designed for work that can be broken into smaller pieces recursively. The goal is to use all the available processing power to enhance the performance of your application.

As with any ExecutorService implementation, the fork/join framework distributes tasks to worker threads in a thread pool. The fork/join framework is distinct because it uses a *work-stealing* algorithm. Worker threads that run out of things to do can steal tasks from other threads that are still busy.

The center of the fork/join framework is the [ForkJoinPool](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html" \t "_blank) class, an extension of the AbstractExecutorService class. ForkJoinPool implements the core work-stealing algorithm and can execute [ForkJoinTask](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinTask.html" \t "_blank) processes.

**Basic Use**

The first step for using the fork/join framework is to write code that performs a segment of the work. Your code should look similar to the following pseudocode:

**if (my portion of the work is small enough)**

**do the work directly**

**else**

**split my work into two pieces**

**invoke the two pieces and wait for the results**

Wrap this code in a ForkJoinTask subclass, typically using one of its more specialized types, either [RecursiveTask](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/RecursiveTask.html" \t "_blank) (which can return a result) or [RecursiveAction](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/RecursiveAction.html" \t "_blank).

After your ForkJoinTask subclass is ready, create the object that represents all the work to be done and pass it to the invoke() method of a ForkJoinPool instance.

**Blurring for Clarity**

To help you understand how the fork/join framework works, consider the following example. Suppose that you want to blur an image. The original *source* image is represented by an array of integers, where each integer contains the color values for a single pixel. The blurred *destination* image is also represented by an integer array with the same size as the source.

Performing the blur is accomplished by working through the source array one pixel at a time. Each pixel is averaged with its surrounding pixels (the red, green, and blue components are averaged), and the result is placed in the destination array. Since an image is a large array, this process can take a long time. You can take advantage of concurrent processing on multiprocessor systems by implementing the algorithm using the fork/join framework.

Work stealing

In [parallel computing](https://en.wikipedia.org/wiki/Parallel_computing), **work stealing** is a [scheduling](https://en.wikipedia.org/wiki/Scheduling_(computing)) strategy for [multithreaded](https://en.wikipedia.org/wiki/Multithreading_(software)) computer programs. It solves the problem of executing a *dynamically multithreaded* computation, one that can "spawn" new threads of execution, on a *statically multithreaded* computer, with a fixed number of processors (or [cores](https://en.wikipedia.org/wiki/Processor_core)). It does so efficiently in terms of execution time, memory usage, and inter-processor communication.

As an example, consider the following trivial fork–join program in [Cilk](https://en.wikipedia.org/wiki/Cilk" \o "Cilk)-like syntax:

**function** f(a, b):

c ← **fork** g(a)

d ← h(b)

**join**

return c + d

**function** g(a):

return a × 2

**function** h(a):

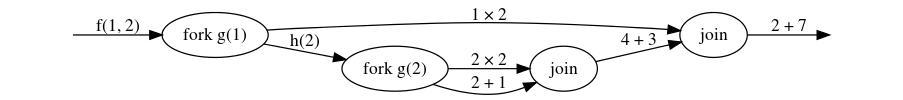
b ← **fork** g(a)

c ← a + 1

**join**

return b + c

The function call f(1, 2) gives rise to the following computation graph:



In the graph, when two edges leave a node, the computations represented by the edge labels are logically parallel: they may be performed either in parallel, or sequentially. The computation may only proceed past a *join* node when the computations represented by its incoming edges are complete. The work of a scheduler, now, is to assign the computations (edges) to processors in a way that makes the entire computation run to completion in the correct order (as constrained by the join nodes), preferably as fast as possible.