# Chapter 4. ForkJoin Pool

The **ForkJoinPool** in Java is a specialized implementation of the **ExecutorService** interface, designed to efficiently execute a large number of small tasks, often in a recursive and divide-and-conquer manner. It is particularly well-suited for tasks that can be broken down into smaller subtasks, which can then be executed in parallel, potentially leading to significant performance improvements on multi-core processors.

**Divide and Conquer strategy**

The term "divide and conquer" is derived from the Latin phrase "divide et impera" and is often associated with the military strategy of breaking the enemy's power into smaller, manageable chunks, and then taking control of those parts one by one. The idea is to divide larger forces into smaller groups that can be defeated in detail, rather than facing a unified, larger force.

Imagine a large territory controlled by a powerful enemy. If Caesar's army tried to conquer this territory as a whole, the battle would be challenging due to the enemy's unified and concentrated force. Instead, Caesar could divide the territory into smaller regions and conquer each region individually. Each conquered region would then serve as a base for attacking the next, until the entire territory is under control. This strategy reduces the complexity of the problem (the large enemy territory) into simpler sub-problems (smaller regions).

In the context of algorithms, divide and conquer involves three steps applied recursively to solve a problem:

A diagram of problems with purple circles and white text

Description automatically generated

1. **Divide:** Break down a large problem into smaller sub-problems. These sub-problems are similar to the original but smaller in scale, just like breaking down a large territory into smaller regions.
2. **Conquer:** Solve the smaller sub-problems. If they are still too large, continue to divide them further. In military terms, this is akin to defeating each smaller region one at a time. When the sub-problems are small enough, they can be tackled directly and solved easily, just as a smaller region would be easier to conquer than a large territory.
3. **Combine:** Merge the solutions of the sub-problems to form the solution to the original problem. Once all regions are conquered, Caesar would consolidate control over the territory, combining the strength of the conquered regions to establish rule over the entire territory.

In both cases, the power of the strategy lies in the systematic breakdown of a large, complex problem into smaller, more manageable parts that can be handled individually and then integrated to form a comprehensive solution.

**Divide and Conquer application example**

Now let’s apply this approach to something algorithmic. Assume we have an array with 20 numbers and need to calculate its sum.

A diagram of numbers and a circle

Description automatically generated

We will use a divide-and-conquer strategy to compute the sum of all elements.

The process is described as follows:

1. **Initial Task (sum(arr)):** The initial task is to sum all elements in the array. The array has 20 elements, and the task assesses whether the current problem size is "big".
2. **Division (Splitting):** If the array is too large to process efficiently in a single task, it is split into two halves, creating subtasks to sum the left and right halves of the array (referred to as **left\_half\_arr** and **right\_half\_arr**).
3. **Recursive Splitting:** Each of these subtasks is then evaluated to see if they are "big". If so, they are further split into halves. This process continues recursively until the subarrays are considered "not so big" and manageable for direct computation. In the image, the array is split into quarters.
4. **Direct Calculation:** Once the subarrays are small enough (the "not so big" point), the algorithm calculates the sum directly for these segments. In this case, the calculations are **sum(0…length/4)**, **sum(length/4…length/2)**, and so on.
5. **Conquering (Joining):** After the sums of the smaller segments are calculated, the results are combined (joined) to produce the sum of larger segments, ultimately leading to the sum of all elements in the original array.
6. **Final Result:** The final step is to join all the partial sums to get the total sum of the array.

The visual representation showcases how the fork/join framework efficiently breaks down a large task into smaller, more manageable tasks, processes them in parallel (potentially on different threads or processors), and then combines the results to obtain the final outcome. This approach is particularly effective for multi-core systems, where such parallel computations can be performed simultaneously, leading to faster processing times.

**ForkJoin algorithm**

Now let’s describe a more generic idea of the ForkJoin algorithm.

A diagram of a network

Description automatically generated

Initial task

Here's an explanation of the algorithm:

1. **Initial Task (Root Node):** The algorithm starts with the initial task, the starting point of the process where the task is at its whole, unbroken state.
2. **Forking Phase:**
   * The initial task begins to "fork," which means it divides itself into smaller subtasks. These are represented by the arrows labeled "fork" pointing to new nodes.
   * Each of these new nodes represents a subtask that can be processed in parallel. The process of forking continues recursively, with subtasks creating further subtasks until a level of granularity is reached where the tasks cannot or should not be divided further.
3. **Parallel Execution:**
   * These subtasks are executed in parallel, potentially by different threads or processors. In a real system, this would be where the **ForkJoinPool** employs its threads to run these tasks concurrently.
4. **Joining Phase:**
   * After a subtask completes its work, it reaches a "join" point. The join operation is where the results of the subtasks are combined. This could mean aggregating results, such as summing values, or simply waiting for all subtasks to complete before proceeding to the next step.
   * The arrows labeled "join" indicate the points in the process where the tasks are being synchronized or their results are being combined.
   * This joining continues until all subtasks have been rejoined back into a single result, which completes the original task.
5. **Completion:**
   * The process ends when all subtasks have been joined back together, and the original task is now complete with all results from the subtasks combined.

The fork-join model is well-suited for problems that can be broken down into smaller, independent tasks that can be executed concurrently, which is why it's often used for recursive algorithms, parallel processing, and complex data processing tasks. The model takes full advantage of multi-core processors, where each core can handle different subtasks simultaneously, leading to significant improvements in performance for suitable problems.

**ForkJoin Use Cases**

Many algorithms can take advantage of using ForkJoin Pool for processing:

* **Parallel Computing:** When you have tasks that can be divided into independent, smaller tasks, **ForkJoinPool** can help leverage multi-core processors to execute these tasks in parallel, often resulting in significant performance gains.
* **Recursive Algorithms:** Algorithms that naturally divide their work recursively, such as merge sort, quicksort, or tasks involving tree traversals, can benefit from being implemented using a fork-join approach.
* **Data Processing:** Processing large datasets can be expedited by breaking down the dataset into smaller chunks, processing each chunk in parallel, and then combining the results. This is particularly effective for operations that can be performed independently on segments of data, such as filtering, mapping, or reducing.
* **Image Processing:** Tasks like applying filters, resizing, or transforming images can be divided into operations on smaller image segments. Each segment can be processed in parallel, leading to faster processing times for large images.
* **Parallel Streams:** Introduced in Java 8, parallel streams utilize **ForkJoinPool** to provide a high-level, easy-to-use framework for processing collections of data in parallel, abstracting away the details of task decomposition and thread management.

**Benefits**

* **Efficiency:** By enabling tasks to be processed in parallel, **ForkJoinPool** can significantly reduce the time required to complete large computational tasks on multi-core processors.
* **Scalability:** It scales well with the number of processors, as the work-stealing mechanism ensures that all cores are kept busy with minimal overhead.
* **Flexibility:** Programmers can easily implement complex parallelism and concurrency patterns without having to manually manage thread creation, synchronization, and communication.

### Work-stealing algorithm

The work-stealing algorithm is a scheduling strategy used to balance the workload across multiple processors or threads in a concurrent computing environment. It is particularly used in fork-join pools like the one in Java's **ForkJoinPool**.

A diagram of a steal

Description automatically generated

**How Work-Stealing Works:**

1. **Task Distribution:** Initially, tasks are distributed among all available workers (which can be threads or processors) that each have their own double-ended queue (deque) of tasks.
2. **Local Task Execution:** Workers take tasks from the front (head) of their own deque and execute them. Tasks may generate more tasks (subtasks), which are added to the front of the deque, allowing for depth-first execution of related tasks, which can improve cache performance.
3. **Task Theft:** When a worker finishes all the tasks in its deque, it becomes idle. To avoid idleness, it looks for tasks in other workers' deques. It steals tasks from the bottom (tail) of another worker's deque, as this is less likely to cause contention than stealing from the head, where the deque's owner operates.
4. **Continued Stealing:** The process of stealing tasks continues until there are no more tasks left to steal, meaning all tasks have been completed.

**Why It's Used for Fork-Join Pools:**

1. **Task Granularity:** Fork-join pools often deal with tasks that can be broken down into smaller tasks recursively. The granularity of these tasks can vary greatly, and work-stealing helps in efficiently managing a mix of large and small tasks.
2. **Load Balancing:** Work-stealing is designed to dynamically balance the workload across all workers without central coordination, which is a common case in fork-join pools. It reduces idle time and makes sure that all workers are equally busy.
3. **Reduced Contention:** In a fork-join pool, since each thread has its own task queue, the contention for shared resources is minimized. Workers mostly operate on their own deques, interacting with others only when stealing tasks.
4. **Non-blocking Synchronization:** Work-stealing typically uses non-blocking synchronization mechanisms, which is suitable for fork-join tasks where blocking operations can lead to underutilization of CPU resources.
5. **Recursive Task Dependency:** Fork-join tasks often have dependencies (one task waiting for the completion of its subtasks), and work-stealing allows workers to execute subtasks deeply before moving to others, which can help in respecting these dependencies.