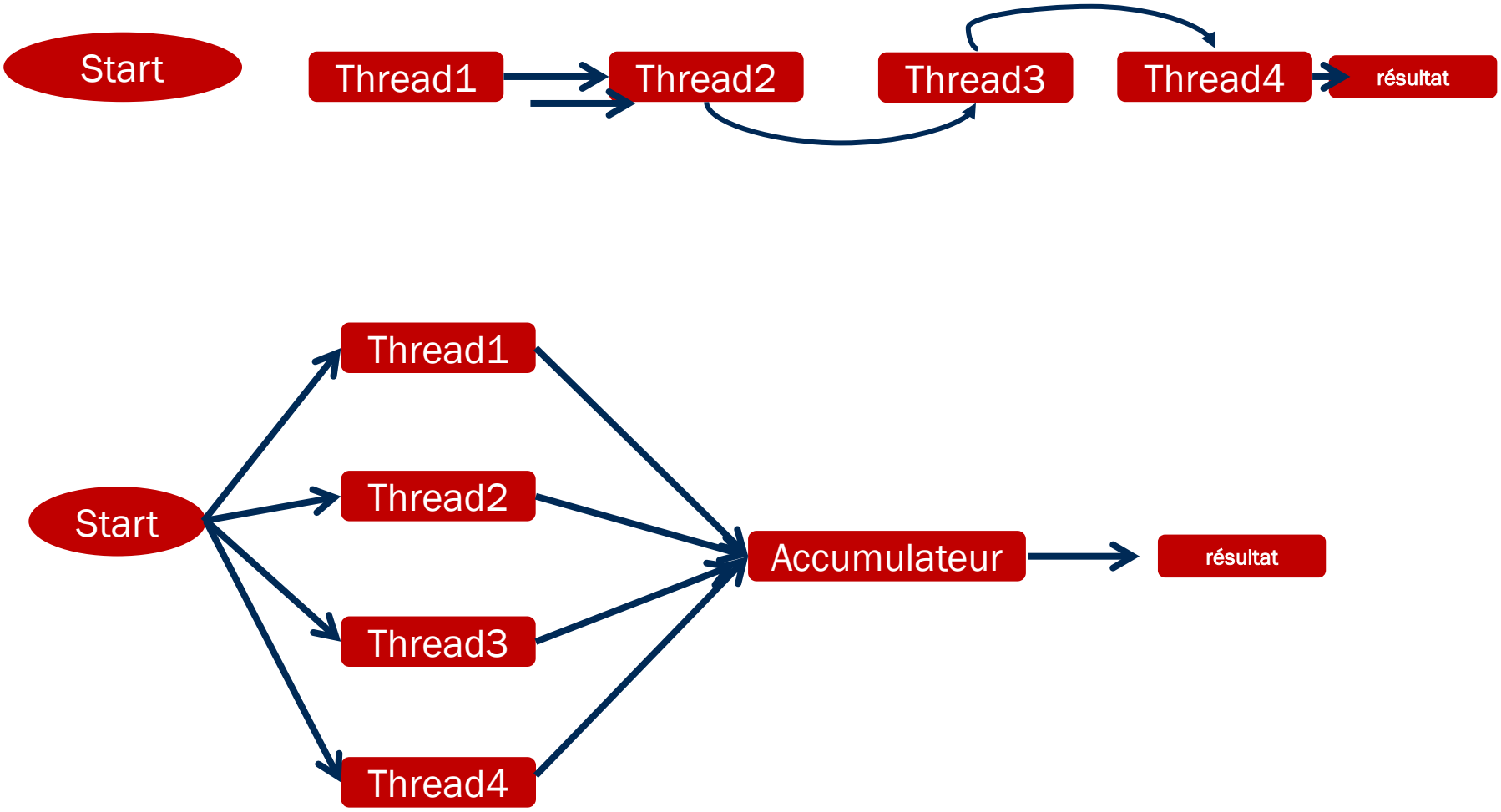
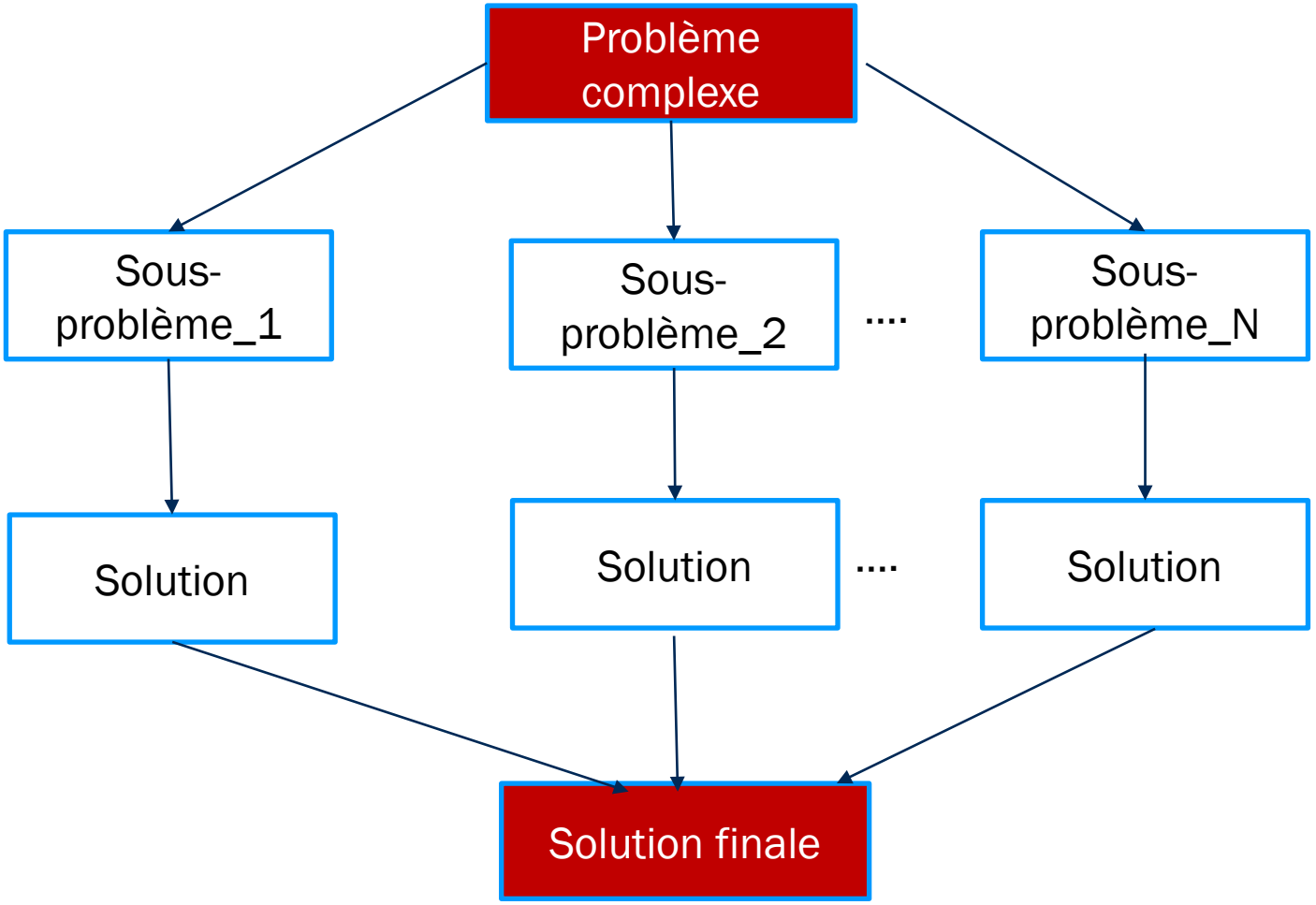
Paradigmes de programmation avancés II

Cours-02: Fork&Join

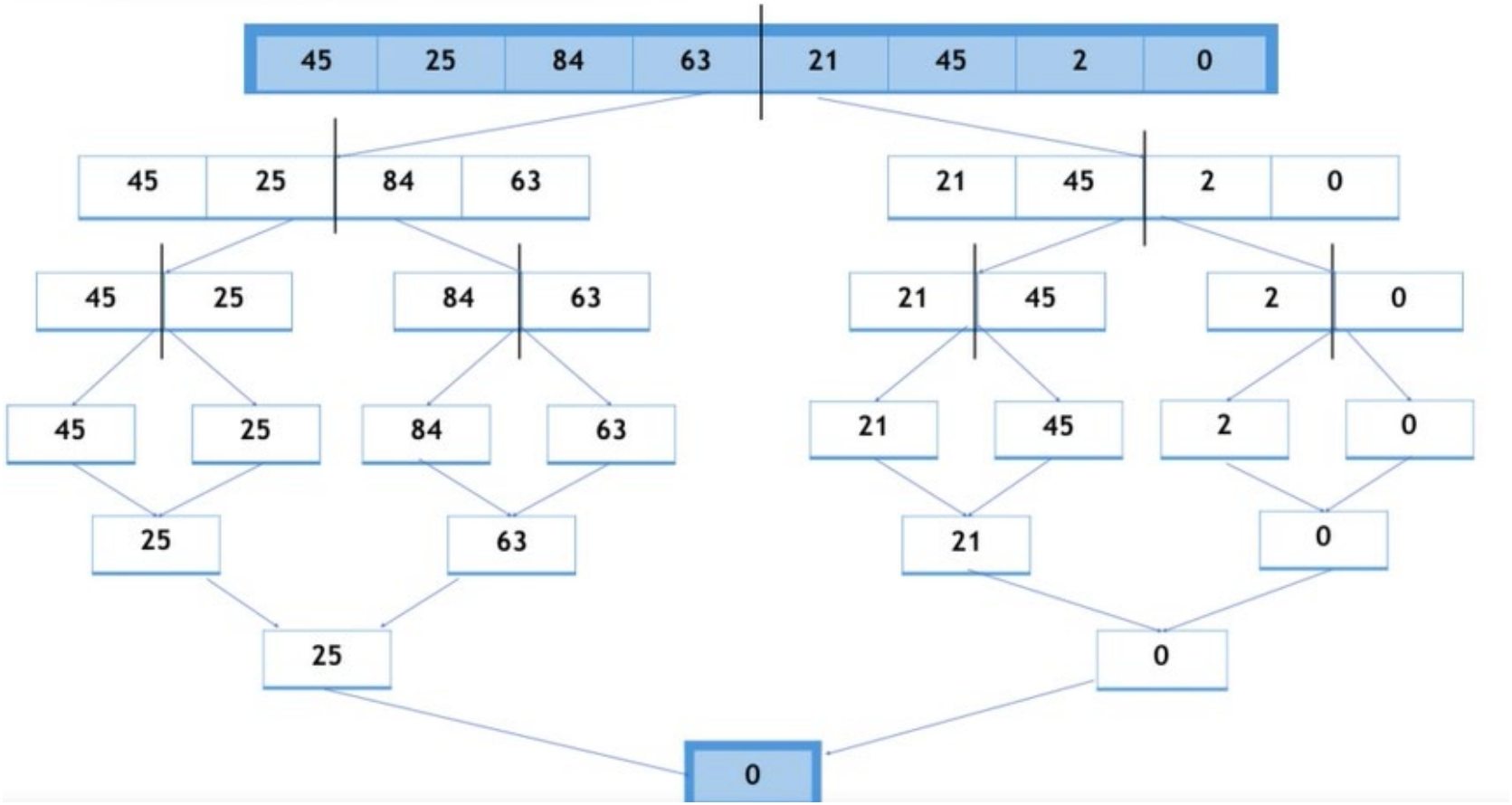
Séquentiel VS algorithme parallèle



Deviser pour régner



Deviser pour régner



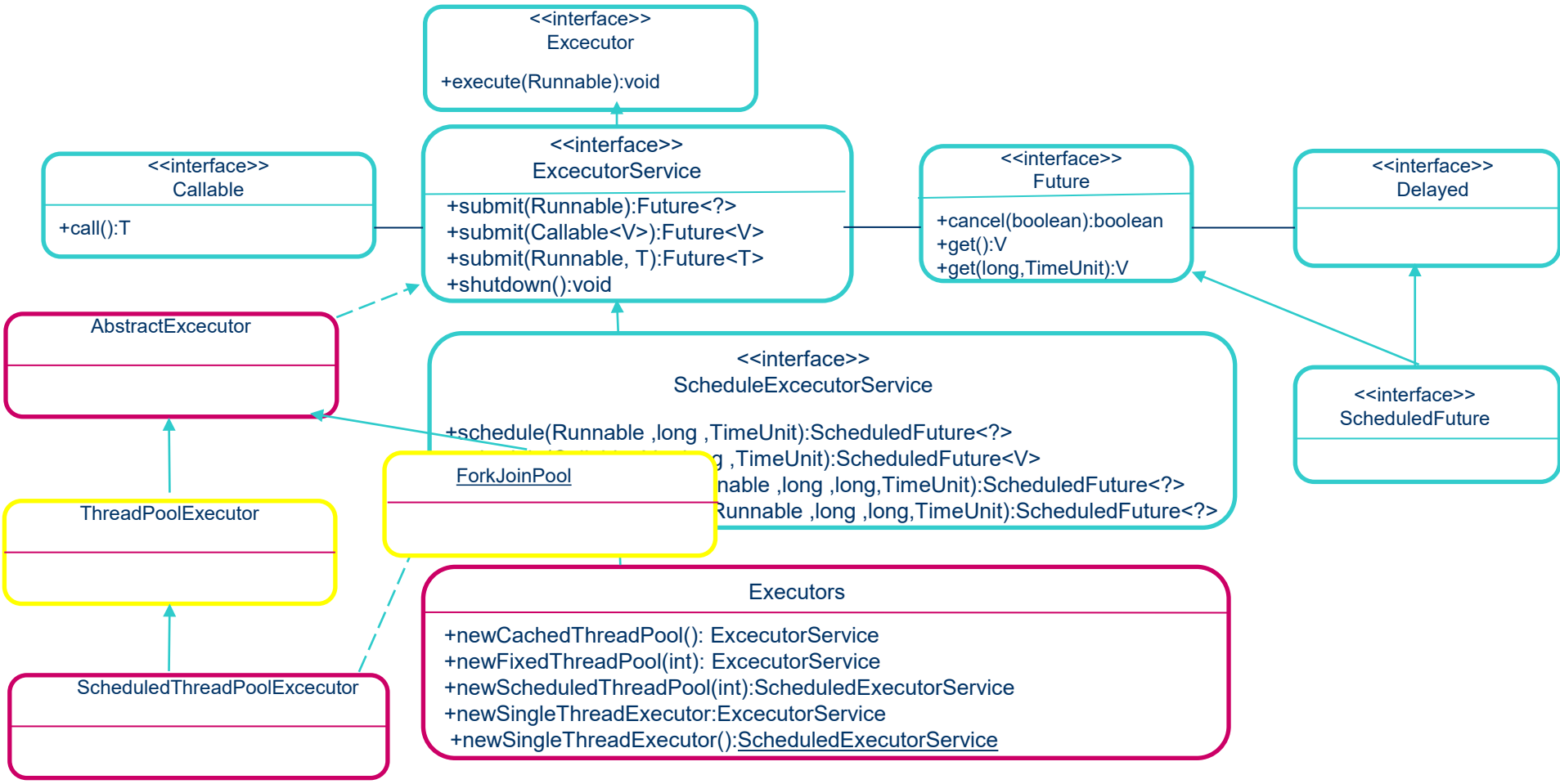
Exemple séquentiel

```
public class FindMin{
 public static void main(String[] args) {

     int[] array = {99,9,14,5,7,33,6,8,21,29,33,44,55,66,77,88,2, 3, 1};
     int min=FindMinimun(array, 0, array.length-1);
     System.out.println(min);
}

private static int FindMinimun(int[] array, int i, int j){
      int mid, min1, min2;

      if(i<j){
           mid= (i+j)/2;
           min1= FindMinimun(array, i, mid);
           min2=FindMinimun(array, mid +1, j);

           if(min1 <min2) return min1;
           else return min2;

      }else{ return array[i];


 } }
}
```

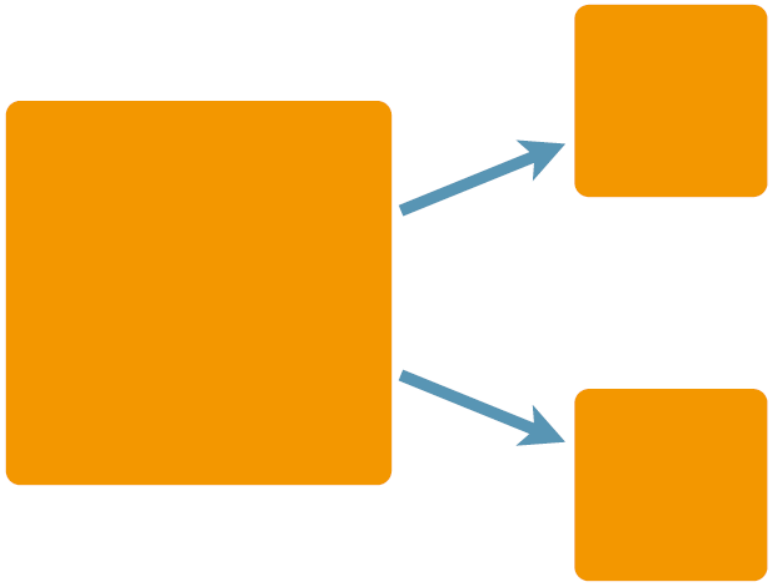
Exemple Multi-threadé

```
private static int FindMinimun (int[] array, int i, int j) throws InterruptedException ,ExecutionException {

  int mid, min1, min2;

  if(i<j){

      mid= (i+j)/2;

      ExecutorService service = Executors.newFixedThreadPool(5);

      Future<Integer>  future1 = service.submit(new Callable<Integer>(){
          @Override
          public Integer call() throws Exception{
               System.out.println(Thread.currentThread()+"for min index: " +i+"and max index: "+mid);
               return FindMinimun(array, i, mid);
          }
      });

      Future<Integer>  future2 = service.submit(new Callable<Integer>(){
          @Override
          public Integer call() throws Exception{
                System.out.println(Thread.currentThread()+"for min index: " +(mid+1)+"and max index: "+j);
               return FindMinimun(array, mid +1, j);
          }
      });

      min1 = future1.get();
      min2= future2.get();

     service.shutdown();

      if(min1 <min2) return min1;
      else return min2;
  }else{

      return array[i];
  }

}
```

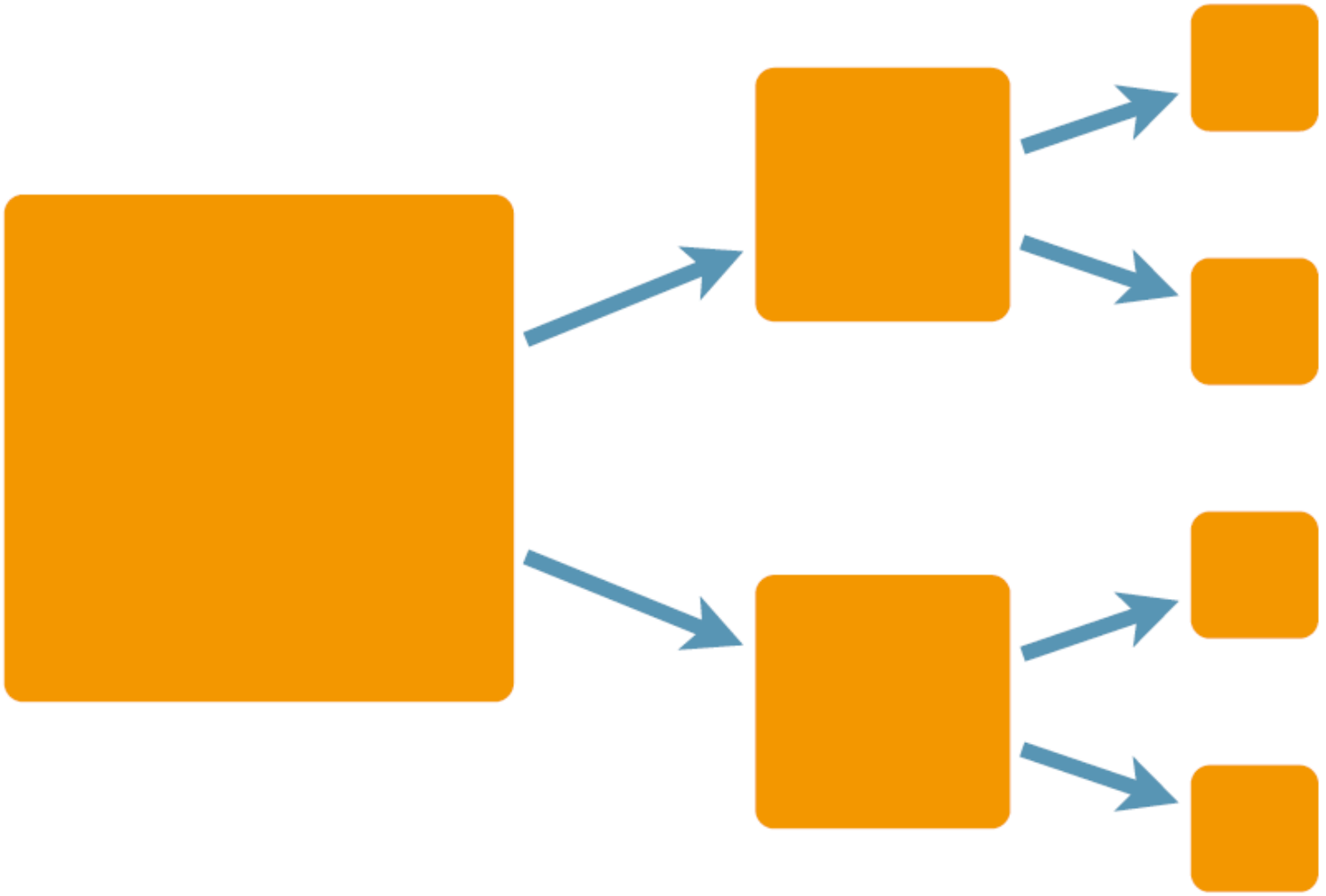
Framework-Executor(Rappel)



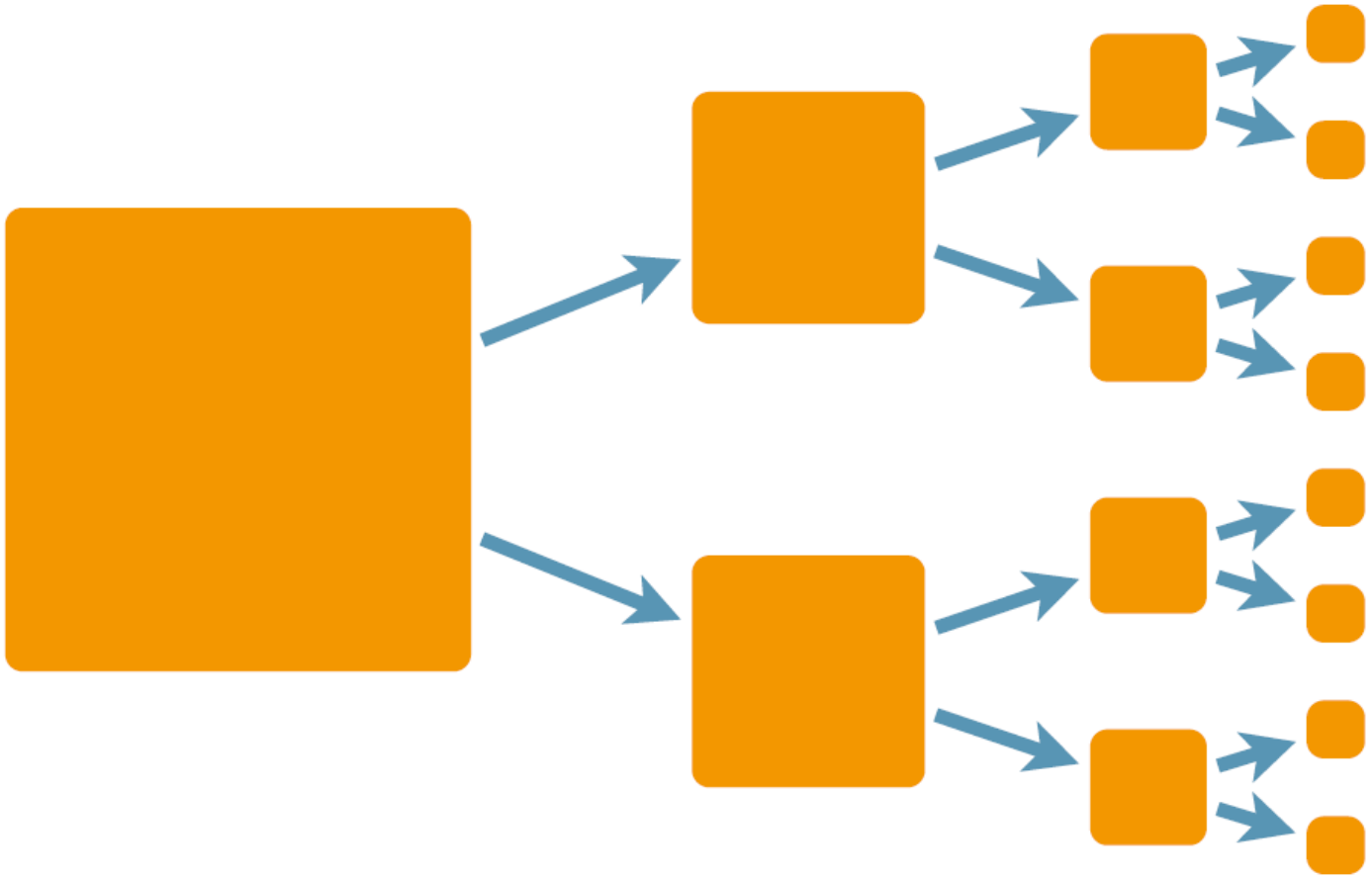
Fork/Join



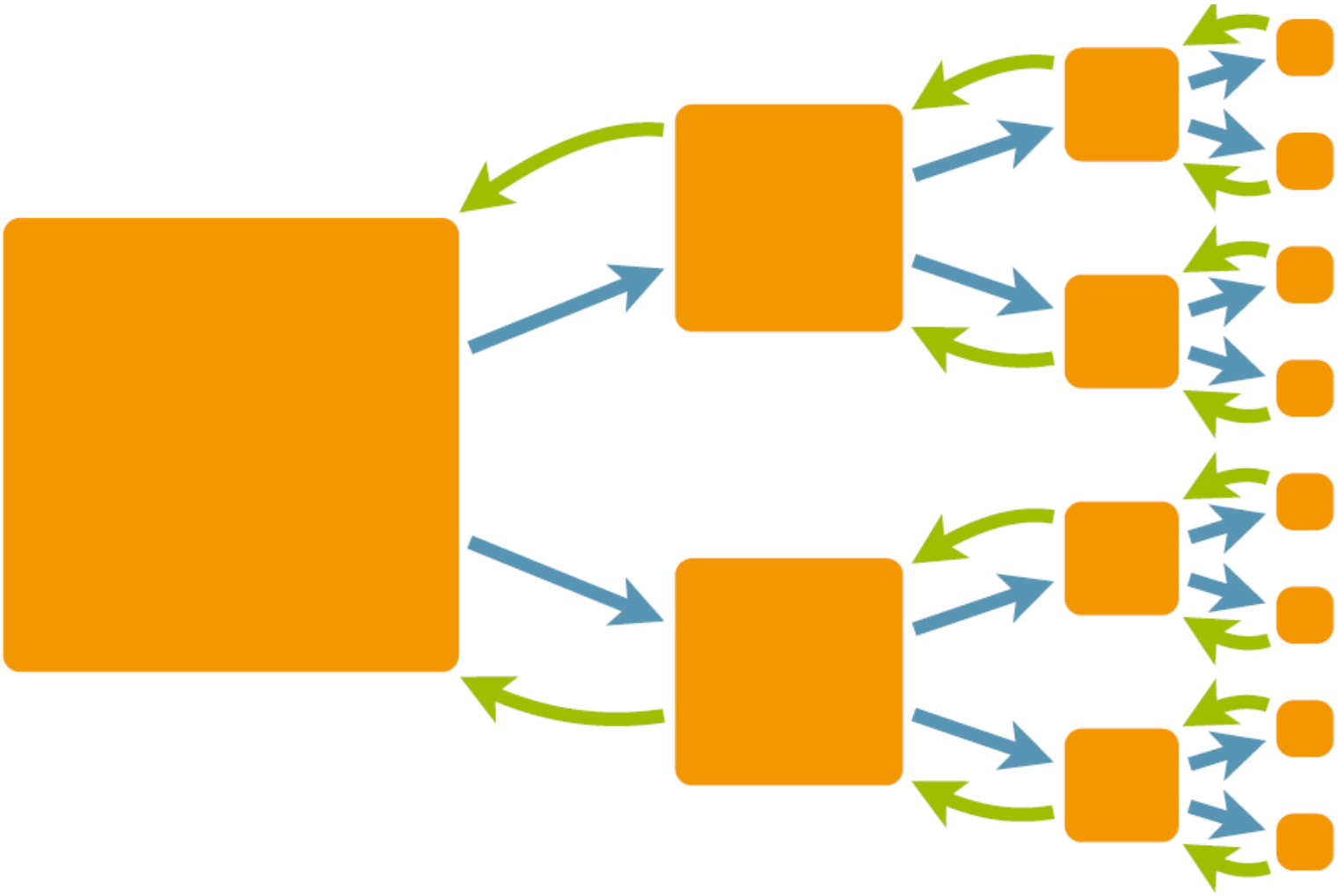
Fork/Join



Fork/Join



Fork/Join



Exemple FORK/JOIN

```
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveAction;
public class RecursiveActionDemo {
    public static void main(String[] args) {
        ForkJoinPool pool = new ForkJoinPool();
        int [] data = {1,2,3,4,5,6,7,8,9,10};
        Square app = new Square(data, 0, data.length);
        pool.invoke(app);
        System.out.println(app.result);
    }
}
class Square extends RecursiveAction {
        final int LIMIT = 3;
        //keep static
        static int result;
        int start, end;
        int[] data;
        Square(int[] data, int start, int end) {
            this.start = start;
            this.end = end;
            this.data = data;
            System.out.println(" Objet de plus avec debut:   "+this.start+ "  et fin :  "+this.end );
        }
        @Override
        protected void compute() {
            System.out.println(" compute");
            if((end - start)< LIMIT){
                for(int i= start;i<end;i++){
                    result+= data[i]*data[i];
                }
            }else {
              int mid = (start + end)/2;
              Square left = new Square(data, start, mid);
              Square right = new Square(data, mid, end);
              left.fork();
              right.fork();
              left.join();
              right.join();
            }
        }
}
```

Fork/Join

- Traiter un grand nombre de données
- La donnée est divisée en paquet
- Chaque paquet est traitée indépendamment, et fournit un résultat partiel
- Ces résultats intermédiaires sont regroupés, pour fournir le résultat global
- Modélisation très fréquente en parallélisme