# Parallel Task Executor in Java

Niravkumar Patel
Computer Science Department
San Jose State University
San Jose, CA 95192
425-772-2509
niravkumar.patel1989@gmail.com

## ABSTRACT
In software development there are many task and processes that can be executed in parallel which improves performance and shortens the response time. With the use of Blocking Queue and ThreadPoolExecutor, we can create a configurable class manager which can take the task that can be executed in parallel. Expected result would be a customizable class which will be initialized by configuration parameters like number of threads, type of task if known. After initialization, one can give any number of runnable task which can be processed in parallel.

## 1. INTRODUCTION
As per the "Moore's law", over the history of computing hardware, the number of transistors in a dense integrated circuit doubles approximately every two years. This low is proven to be true till date. Computational powers of systems are increasing exponentially. To take the benefits of the evolution of computing powers in software development, programming languages are providing frameworks for parallelism to increase the output and the performance of applications.

## 2. Parallelism and Concurrency
Parallelism refers to a procedure or a method to do work simultaneously. It means your work can be divided into smaller chunks and also those chunks can be executed parallel without affecting the final result. With introduction of parallelism in any problem solving or work executing service, will increase the throughput and also saves time. The time saved is purely depended upon the level of parallelism one can get by identifying how and when work are divided into independent chunks so that they are executed in parallel efficiently. To achieve parallelism in computing worlds, require hardware resources which can be multiple processor, multiple adder etc. After some extend there are limitation on hardware resource to fit into one machine. Concurrency is a way to achieve virtual parallelism while utilizing most of available hardware resources. Implementation of concurrent process are more complex than implementing parallelism as it involves communication between system while execution. In concurrency communication between systems can be a shared resources which will lead to issues such as deadlock. In parallel execution environment, communication are normally structured and predefined so there are less chances which makes it less prone to issues.

In perspective of a programmer, they can achieve parallelism or concurrency require knowledge on Process and Threads.

### 2.1 Process
Process is an instance of a computer program that is being executed. A new process require a copy of parent process and separate allocation of memory and resources. Process are independent of other processes. All the works that are computed in computers are set of instructions. Process is execution of these instructions. Each process has their own execution environment with allocated resources required like data, kernel context. Process runs in separate address space. Communication between processes happens using Inter process communication. Context switching between processes is costly compare to context switching between threads of the same process.

### 2.2 Thread
Thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler. Thread is a component of process. Multiple threads can be part of process and sharing resources such as memory allocated to process, same address space. Communication between threads are happened with the user of shared resources. Sharing common resources introduces complex failure scenario like deadlock. As threads are sharing same resources, context switching between threads by kernel is faster compare to context switching between processes. To manage concurrency between threads sharing same resources require handling complex problem which introduces mutex and semaphores. Due to the availability of multicore – multi processor system, use of thread can benefits in the throughput and the efficient use of resource.

Managing threads manually can be very complex. There are operations on Threads like putting thread to sleep, determining which thread is alive, thread notify, thread join, thread wait, suspending a thread. All the above operation if not properly handled then it can lead to many error condition which are hard to trace and also thread debugging also require much effort to identify the error.

## 3. Frameworks to Manage Threads
### 3.1 Fork/Join Framework
It is an ExecutorService for running ForkJoinTasks. It provides framework to manage and monitor thread operation. It contains a pool of threads in which based on requirement dynamically new threads are added and also suspended. It has many advantages over managing multiple threads manually. It is an implementation of Executor interface which ease the management of concurrent tasks.

Many algorithm in computing follows the “divide and conquer” paradigm.

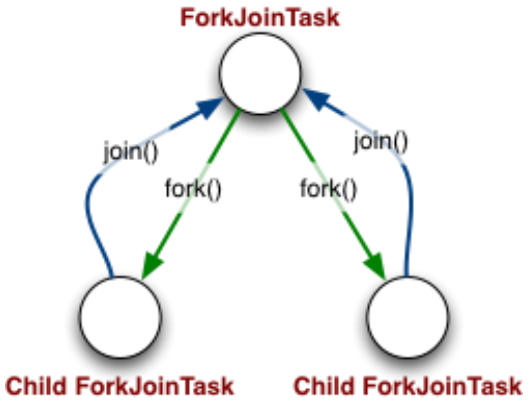


**Figure 1. ForkJoin Framework Bigger picture**

ForkJoinPool suits best for these kind of algorithms which are also referred as Map-Reduce technique [2]. To explain this divide and conquer task, take an example of huge array of integers. To compute the sum of each integer value from the array, one can think of dividing that array into smaller chunks and compute the some of those smaller chunks. Adding those value with other chunks will get the final sum. So having multiple threads working on different chunks can compute independently can improve performance.

**Divide and Conquer Example**

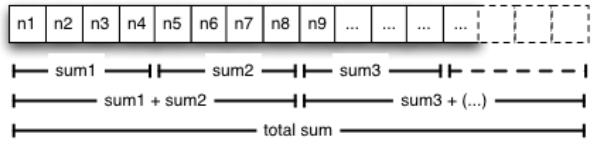


**Figure 2. ForkJoin Framework Application**

So with the use of ForkJoinPool, we can assign a specific chunk of element to a threads available in ForkJoinPool by submitting task.

## 3.2 ThreadPoolExecutor

It is an ExecutorService that execute each submitted task using one of possibly several pooled threads, normally configured using Executors factory methods. ThreadPoolExecutor is an implementation of java.util.Executor, java.util.ExecutorService interface [3]. ThreadPoolExecutor is best suitable for executing large number of asynchronous tasks. It provides a framework where burden of invoking new threads, keeping some threads alive, suspending thread, allocation of task to the available threads it taken away from a programmer. ThreadPoolExecutor manages all thread activity on submitted runnable task. One can also make changes to the default configuration by giving values either at the time of initialization or using configuration method supported by the class. Default constructor of ThreadPoolExecutor is as follows:

```
ThreadPoolExecutor(
        int corePoolSize,
        int maximumPoolSize,
        long keepAliveTime,
        TimeUnit unit,
        BlockingQueue<Runnable> workQueue
        )
```

Configuration parameter that one can manipulate to make most use of ThreadPoolExecutor to satisfy the requirements.

### *3.2.1 Core and Maximum Pool Size*

corePoolSize parameter is to defined minimum number of threads that needs to be keep alive even if there is no task pending to execute. There can be requirement when runnable task can be sometimes less and sometimes there is sudden increase of task, so one can make corePoolSize parameter such that in average case scenario, there are at least some number of threads that are kept alive. This number can be set by setting value of corePoolSize parameter. To manipulate the value of this corePoolSize, one can provide value at the time of constructor calling or at the run time one can also use the method setCorePoolSize(int).

maxPoolSize parameter is to defined the maximum number of threads that need to be created when there are so many tasks pending. This parameter is defined the upper limit of the number of threads which a ThreadPoolExecutor can have at any point of time. There can be a system limitation or resources limitation, one should keep number of threads up to the limit so that program execution should not halt because of limited resources. This number can be set by seeting value of maxPoolSize parameter. To manipulate the value of this maxPoolSize, one can provide value at the time of constructor calling or at the run time one can also use the method setMaximumPoolSize(int).

The main idea behind keeping active thread even if there is not a single task to execute is because it requires much more time to initialize new thread and assign task to it when task comes. Rather than creating new thread each time new task arrived, ThreadPoolExecutor assign task to active thread from the thread pool, saves a lot time when there are streaming of new task.

### *3.2.2 Keep-alive times*

This parameter is used to suspend threads that are idle and number of threads are more than corePoolSize. So considering case when number of threads are more than a corePoolSize and now there are at most corePoolSize number of threads or less threads are active. So the extra threads needs to be suspended. To suspend those thread after certain amount of time, this time can be configured by setting Keep-alive times in TimeUnit.NANOSECONDS. By doing this, resource consumption can be reduced. One can provide value at the time of constructor initialization or at the runtime using method setKeepAliveTime(long, java.util.concurrent.TimeUnit).

### *3.2.3 Queuing*

Number of task that are submitted can change dynamically. So when ThreadPoolExecutor is busy with all threads reaching active thread count maxPoolSize, pending task needs to be cached in some form of structure. ThreadPoolExecutor uses any blocking queue to submit these task and cache these task. There are three scenarios of task submission considering Thread count. One is when number of threads are less than the corePoolSize. In this

scenario, new task will be handed over to thread by creating new one. Second scenario in which number of threads are more than corePoolSize and less than maxPoolSize. In these case, new task will be pushed to blocking queue. Third scenario where number of threads are maxPoolSize and also the queue is full than in that case new task will be discarded.

To manage the task in the queue at the run time, one can make use of methods getQueue(), remove(java.lang.Runnable), purge().

### *3.2.4 Finalization*

After the work done, one can explicitly call shutdown() method on ThreadPoolExecutor. There is an another method for suspending core threads in the thread pool, that is allowCoreThreadTimeOut(boolean). With the use of this method, one set timer so that all the core thread can be terminated if the remain idle.

## 4. Role of Blocking Queue

Blocking Queue is Queue with extra thread safe operation handling wait condition on thread when queue is empty or full. So considering producer consumer problem and blocking queue as a task holder. Producer thread will start producing task and place it in blocking queue, and consumer thread will consume the task from the task holder. Now in case of queue is full, producer thread will go in wait state and come back to ready state when consumer consumes the task from the blocking queue. Considering the other scenario, where there is no task is produced by the producer and consumer thread has consumed all the threads from the queue till queue got empty. Now Consumer thread will go into wait state, and will get back in to ready state only after producer will add any task into the task holder blocking queue. So use of blocking queue in ThreadPoolExecutor will reduce the overhead on programmer to manage the upcoming tasks that need to be run in parallel. BlockingQueue interface has provided methods in different form to cover all different aspects. Main operation on blocking queue include adding an element, removing an element and checking the value of an element. All of these methods are provided by blocking queue with for different variation. One that throws an exception [add(e), remove(), element()]. For example adding an element to already full queue. One that returns special value which can be true/false depending upon the operation [offer(e),poll(),peek()]. One that blocks thread till operation got over [put(e), take()] and the last one that also allow to specify the time limit to wait for operation and give up if operation not succeeded [offer(e, time, unit), poll(time, unit)].

Blocking operation have many benefits if we consider producer-consumer problem.

**Table 1. Blocking Queue Operations**

| | **Insert** | **Remove** | **Examine** |
|---|---|---|---|
| Throws exception | add(e) | remove() | element() |
| Special Value | offer(e) | poll() | peek() |
| Blocks | put(e) | take() | NA |
| Time out | offer (e,time, unit) | poll(time,unit) | NA |

## 5. Practical Usage of ThreadPoolExecutor

Basic purpose of the ThreadPoolExecutor is to provide a pool of active threads, which can be run parallel and management of thread can be handled by the framework only. So usage of ThreadPoolExecutor covers wide range of application where multiple independent task that can be executed in parallel, are coming. After identification of the task, one can configured the ThreadPoolExecutor by setting corePoolSize, maxPoolSize, providing blocking queue and setting the keep-alive time.

For example, there is a requirement of large number of documents that needs to parse in real time. There can be basic three approach.

### 5.1 Single Threaded Execution

To complete the objective using single threaded environment. There will be only one thread that will take the file name, parse the file and write back the output. After output is written to the disk, it will take another file. So with single threaded environment, total time taken will be time taken to parse one file time number of files. So as the number of new file increases, the performance degrades and many number of files put down to wait state.

### 5.2 Multi-Threaded Execution

To complete the objective using multiple threaded environment. There will be bunch of threads that can be run in parallel for file parsing. So whenever new file arrives one has to initiate new threads and assign new file parsing task to that thread. So now that thread will be busy parsing allocated file, and meanwhile another parsing task arrived, one can again initiate new threads and assign new task to it. In this multi-threaded execution, the total time depend total upon the number of threads that are running in parallel, time taken to create new threads when task arrived. This seems to be a good solution but handling threads manually at run time can lead to many troublesome scenario. In case of where speed of new task coming is unpredictable than the handling of those task at the peak time becomes much more complex. Even creation of new thread when new task arrives takes time, rather one can have some set of threads already initialized and in idle state. So that whenever new task arrived, task can be allocated to one of the idle thread. There can be more scenario that needs to be handled, like if there are many task coming and all of the available threads are busy parsing files already allocated, then there needs to be a data structure to keep pending task in a cache or in a queue so that once any thread is don't with already allocated work, can start executing task pending in a queue. This data structure also need to be implemented in the Single Threaded environment for same scenario. Use of multi-threaded execution is definitely a boost in the processing but in programmer perspective it becomes complex as requirements changes.

### 5.3 Using ThreadPoolExecutors

All the case that are explained in Multi-Threaded execution are best fit only if they are implemented and handled correctly. Rather than handling manually, one can use ThreadPoolExecutors. One can identify the minimum number of threads that needs to be there that can be sufficient enough for the average case scenario. So that no need for creating new threads when new task arrived. One can identify the maximum number of thread as per the system limitation or the memory limitation, by doing this one can put an

upper limit on thread creation when there are many number of task in the pending state.

**ThreadPoolExecutor to parse documents**

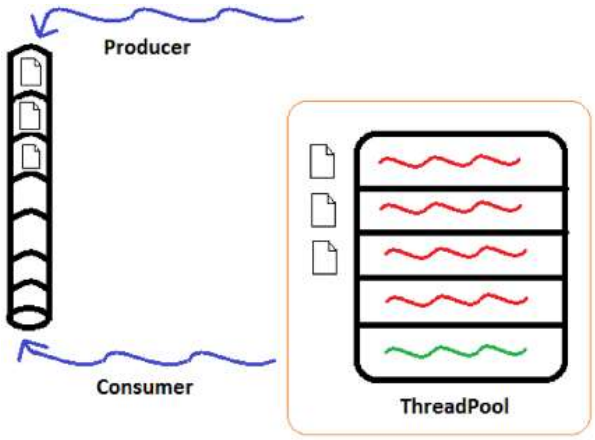


**Figure 3. ThreadPoolExecutor overview**

ThreadPoolExecutors provides a blocking queue to place the task for caching. There are mainly three strategies for queueing.

Direct Handoffs: SynchronousQueue that just pass the task to available thread without holding them. If there is no thread available immediately, task will fail.

Unbounded Queues: In this case, unbounded queue can be LinkedBlockingQueue, which has dynamic size. So all the task are cached to this queue and only corePoolSize number of thread are kept in the pool. This approach will be helpful when task arrives at nearly at average rate.

Bounded Queue: A blocking queue with defined number of slots are assigned to ThreadPoolExecutor. In this case we can limit the number of task that can be cached and also in this case we also need to take care of maxPoolSize parameter of ThreadPoolExecutor. There is a tradeoff between number of slots assigned to queue for task holder and number of maxPoolSize assigned. If we increase maxPoolSize than use of resources increases which may lead to resource exhaustion and if we decrease number maxPoolSize and increase the number of slots in the queue than it can slow down the throughput.

# 6. Implementation

## 6.1 Task

For a purpose to check effectiveness of ThreadPoolExecutor, I took a task of reading a single file (size: 84,611,072 bytes) and parse it. So considering a case where there are three task to read this file.

### *6.1.1 Task Sample*

```
File f = null;
FileReader fr = null;
f = new File("C:\\Users\\Niravkumar\\Desktop\\Temp.txt");
fr = new FileReader(f);
String line = null;
BufferedReader bf = new BufferedReader(fr);
while((line = bf.readLine())!=null){
        System.out.println(Thread.currentThread().getId()+"::"+
line);
}
bf.close();
fr.close();
```

## 6.2 Blocking Queue

For a purpose of caching task in a blocking queue. I have taken java.util.concurrent.ArrayBlockingQueue[3] as a task holder. Using singleton pattern [4], we can defined a queue by specifying queue size.

### *6.2.1 TaskHolder Sample*

```
private static BlockingQueue<Runnable> blockignQueue;
private static int queueSize = 10;

private TaskHolderQueue(){
}

public static BlockingQueue<Runnable> getTaskHolderQueue(){
if(blockignQueue==null){
        blockignQueue = new
        ArrayBlockingQueue<Runnable>(queueSize);
        return blockignQueue;
}else{
        return blockignQueue;
}
}
```

## 6.3 Task Producer

Role of task producer is to put the newly generated or received task into the task holder queue. For our performance testing we have taken 3 tasks that are being placed by the task producer into task holder queue. In practical scenario any process can take a role of producer as far as it is adding runnable task to the queue. It can be any streamed data packet that is converted to runnable task or in our case file name that is received to be parsed. TaskProducer can wait() on the queue, if it perform put(task) operation while queue is full.

### *6.3.1 TaskProducer Sample*

```
Task task = new Task();
TaskHolderQueue.getTaskHolderQueue().put(task);
```

## 6.4 Task Consumer

Role of task consumer is to fetch the task place by consumer from the task holder queue. Fetched task is given to the ThreadPoolExecutor for further processing. Task consumer can wait on blocking queue if it performs take() on queue when queue is empty.

### *6.4.1 ThreadPoolExecutor Sample*

```
ThreadPoolExecutor tpe = new ThreadPoolExecutor(5, 10, 100,
TimeUnit.SECONDS, TaskHolderQueue.getTaskHolderQueue());
```

### *6.4.2 Task Consumer Sample*

```
tpe.execute(TaskHolderQueue.getTaskHolderQueue().take());
```

## 7. Performance

To test the improvement on total time taken to parse 3 files using ThreadPoolExecutor, I have defined minimum number of 5 threads as corePoolSize and provided 3 task in java.util.concurrent.ArrayBlockingQueue. Time duration taken by whole process were 1.010 Seconds.

Same task are assigned to single threaded environment where parsing is done one by one. So there will be same number of task are assigned but they will be executed sequentially which means using single thread only. Time duration take by whole process where 2.307 Seconds.

With the use of ThreadPoolExecutor, there is a clear performance improvement. Here still the number of task is very less for the testing purpose, so as the number of task increases, better performance can be achieved.

## 8. Conclusion

To achieve parallelism, there are many frameworks available which are made to reduce the overhead of manual maintenance from a programmer and try to make it as much customizable as possible, so that one can configure it as per the requirements. As shown above, one can definitely make a use of ThreadPoolExecutor for processing independent task and can achieve great performance boost. On top of performance boost, ThreadPoolExecutor is highly configurable in terms of number of threads, time constraint on threads, variance on blocking queue as per the requirements.

## 9. REFERENCES


[1] Veldema, R., Hofman, R.F.H., Bhoedjang, R.A.F., Bal, H.E.: Runtime Optimizations for a Java DSM Implementation. ACM Concurrency: Practice and Experience 15, 299-316 (2003).

[2] "Fork and Join: Java Can Excel at Painless Parallel Programming Too!" by Julien Ponge, http://www.oracle.com/technetwork/articles/java/fork-join-422606.html.

[3] ThreadPoolExecutor class from Java 1.7 concurrent utilities homepage, https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ThreadPoolExecutor.html.

[4] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, "Design Patterns Elements of Reusable Object Oriented Software",Addison Wessley