

|  |  |
| --- | --- |
| Employee Name: | MANOJ KUMAR YEKOLLU |
| Training Supervised by: | **KRISHNA BODDULURI** |
| Training Task #: | **Assignment 05162019** |
| Training Resource Materials: | **Implement Faster Parallel Processing in Java 8 using streams and fork/join** |
| Training Task Date: | **05/16/2019** |
| Task Due Date: | **05/30/2019** |
| Task Submitted Date: | **05/30/2019** |
| Github link: | <https://github.com/NightFury546/Egiants-Assignments/tree/master/JavaParallelProcessingForkJoinDemo> |
| Technologies used for Training | **Java 8 , Steams , ForkJoin framework** |

**Task Description/Requirement:**

**Implement Faster Parallel Processing in Java 8 using streams and fork/join**

**High Level Synopsis:**

While dealing with big number and huge amount of data Current generation computer are not giving optimum performance. We are facing a bottleneck situation with the performance of present computers. Even though there are some researches in progress on Quantum Computing. But those won’t be available in coming 30 years. To solve the present performance bottlenecks and to improve the performance of the computer, Java 8 introduced Streams concept.

Here we can splice the bigger task into multiple smaller tasks using fork() and assign to different threads available in the thread pool. And again, combine the output of the each thread into one using the join().

By using this we can utilize the all cores of the processor and achieve the concurrency which leads the lesser processing times.

Here are few examples:

Machine generated alternative text:
WHAT 
ForkJoinTask 
RecursiveAction 
Fork/Join 
Framework 
ForkJoinPool 
RecursiveTask 

Machine generated alternative text:
HOW DOES IT WORK? 
Big Task 
Fork() 
Smaller Task 
Fork() 
Even Smaller 
Task 
Join() 
Partial 
Result 
Fork() 
Even Smaller 
Task 
Join() 
Jotn() 
Fork() 
Smaller Task 
Fork() 
Even Smaller 
Task 
Join() 
Partial 
Result 
Jotn() 
Fork() 
Even Smaller 
Task 
Join() 
Final Result 

**Sample Algorithm:**

Machine generated alternative text:
public Result solve(Task t) { 
split t into smaller tasks 
for each of these tasks 
solve(ti) 
wait for all tasks to complete 
join all individual results 
return result • 
Simple algorithm 

Using this stream concept, I have written a program which demonstrated the CPU performances and compare those values with the sequential processing. The results are attached below.

Parallel Programming Example:

**public** **static** **long** parallelSum(**long** n) {

/\* return Stream.iterate(1L, i -> i + 1)

.limit(n)

.parallel()

.reduce(0L, Long::sum);\*/

**return** LongStream.*rangeClosed*(1, n)

.parallel() //intermediate operation.

.reduce(0L, Long::*sum*); //terminal operation

}

// commented part is other way of doing it.

**There are other examples which are added in GitHub repository.**

**I acknowledge that this document can be supplied to USCIS in compliance with CPT/OPT/STEM OPT audit:**

MANOJ KUMAR YEKOLLU

**Output:**

**Parallel Programming Output:**

Form the below output we can clearly see the performance scalability using the parallel processing using streams.

In Sequential CPU of 4 cores took :108 msec

In Iterator Processing it took: 4msecs

If we use Parallel processing it only takes: 2 msec

CPU core: 8

Concurrent parallelism: 4

Result: 50000005000000

Result: 50000005000000

Result: 50000005000000

Result: 50000005000000

Result: 50000005000000

Result: 50000005000000

Result: 50000005000000

Result: 50000005000000

Result: 50000005000000

Result: 50000005000000

Parallel sum done in: 2 msecs\n

Result: 50000005000000

Result: 50000005000000

Result: 50000005000000

Result: 50000005000000

Result: 50000005000000

Result: 50000005000000

Result: 50000005000000

Result: 50000005000000

Result: 50000005000000

Result: 50000005000000

Sequential sum done in: 108 msecs\n

Result: 50000005000000

Result: 50000005000000

Result: 50000005000000

Result: 50000005000000

Result: 50000005000000

Result: 50000005000000

Result: 50000005000000

Result: 50000005000000

Result: 50000005000000

Result: 50000005000000

Iterative sum done in: 4 msecs