package com.twitter.search.earlybird.util;

import java.util.List;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.ThreadFactory;

import java.util.stream.Collectors;

import com.google.common.util.concurrent.ThreadFactoryBuilder;

import com.twitter.util.Await;

import com.twitter.util.Future;

import com.twitter.util.Future$;

import com.twitter.util.FuturePool;

import com.twitter.util.FuturePool$;

public final class ParallelUtil {

private ParallelUtil() {

}

public static <T, R> List<R> parmap(String threadName, CheckedFunction<T, R> fn, List<T> input)

throws Exception {

return parmap(threadName, input.size(), fn, input);

}

/\*\*

\* Runs a function in parallel across the elements of the list, and throws an exception if any

\* of the functions throws, or returns the results.

\*

\* Uses as many threads as there are elements in the input, so only use this for tasks that

\* require significant CPU for each element, and have less elements than the number of cores.

\*/

public static <T, R> List<R> parmap(

String threadName, int threadPoolSize, CheckedFunction<T, R> fn, List<T> input)

throws Exception {

ExecutorService executor = Executors.newFixedThreadPool(threadPoolSize,

buildThreadFactory(threadName));

FuturePool futurePool = FuturePool$.MODULE$.apply(executor);

List<Future<R>> futures = input

.stream()

.map(in -> futurePool.apply(() -> {

try {

return fn.apply(in);

} catch (Exception e) {

throw new RuntimeException(e);

}

})).collect(Collectors.toList());

try {

return Await.result(Future$.MODULE$.collect(futures));

} finally {

executor.shutdownNow();

}

}

private static ThreadFactory buildThreadFactory(String threadNameFormat) {

return new ThreadFactoryBuilder()

.setNameFormat(threadNameFormat)

.setDaemon(false)

.build();

}

@FunctionalInterface

public interface CheckedFunction<T, R> {

/\*\*

\* A function from T to R that throws checked Exceptions.

\*/

R apply(T t) throws Exception;

}

}