item 48

스트림 병렬화는 주의해서 적용하라

유현우

CONTENTS

CONTENTS

자바의 동시성 프로그래밍

1996 - 2003 2003 - now

- 처음 릴리스된 1996년부터 스레드, 동기화, wait/notify를 지원
- 자바 5부터 동시성 컬렉션인 java.util.concurrent 라이브러리, Executor 프레임워크를 지원
- 자바 7부터 고성능 병렬 분해 프레임워크인 fork/join 패키지를 추가
- 자바 8부터 병렬 스트림을 지원 (parallel 메서드)

CONTENTS

잘못된 병렬 스트림의 문제점

Item45 메르센 소수(2^p - 1) 출력 예제

```
public static void main(String[] args) {
    long startTime1 = System.currentTimeMillis();
    primes()
        .map(p -> TWO.pow(p.intValueExact()).subtract(ONE))
        .filter(mersenne -> mersenne.isProbablePrime(50))
        .limit(20)
        .forEach(mp -> System.out.println(mp.bitLength() + ": " + mp));
    System.out.println(System.currentTimeMillis() - startTime1);
}

static Stream<BigInteger> primes() {
    return Stream.iterate(TWO, BigInteger::nextProbablePrime);
}
```

잘못된 병렬 스트림의 문제점

```
public static void main(String[] args) {
    long startTime1 = System.currentTimeMillis();
    primes()
        .parallel()
        .map(p -> TWO.pow(p.intValueExact()).subtract(ONE))
        .filter(mersenne -> mersenne.isProbablePrime(50))
        .limit(20)
        .forEach(mp -> System.out.println(mp.bitLength() + ": " + mp));
    System.out.println(System.currentTimeMillis() - startTime1);
}
```

동시성 프로그래밍을 할 때는 안전성과 응답 가능 상태를 유지하기 위해 애써야한다.

잘못된 병렬 스트림의 문제점

```
public static void main(String[] args) {
    long startTime1 = System.currentTimeMillis();
    primes()
        .parallel()
        .map(p -> TWO.pow(p.intValueExact()).subtract(ONE))
        .filter(mersenne -> mersenne.isProbablePrime(50))
        .limit(20)
        .forEach(mp -> System.out.println(mp.bitLength() + ": " + mp));
    System.out.println(System.currentTimeMillis() - startTime1);
}
```

연산이 끝나지 않으면서 CPU는 90%나 잡아먹는 상태가 무한히 계속된다.

(응답 불가: liveness failure)

성능 개선을 위해서 병렬 스트림을 사용했는데 자바에서 파이프라인을 병렬화할 방법을 찾지 못해 문제가 발생한 것이다.

잘못된 병렬 스트림의 문제점

```
public static void main(String[] args) {
    long startTime1 = System.currentTimeMillis();
    primes()
        .parallel()
        .map(p -> TWO.pow(p.intValueExact()).subtract(ONE))
        .filter(mersenne -> mersenne.isProbablePrime(50))
        .limit(20)
        .forEach(mp -> System.out.println(mp.bitLength() + ": " + mp));
    System.out.println(System.currentTimeMillis() - startTime1);
}

static Stream<BigInteger> primes() {
    return Stream.iterate(TWO, BigInteger::nextProbablePrime);
}
```

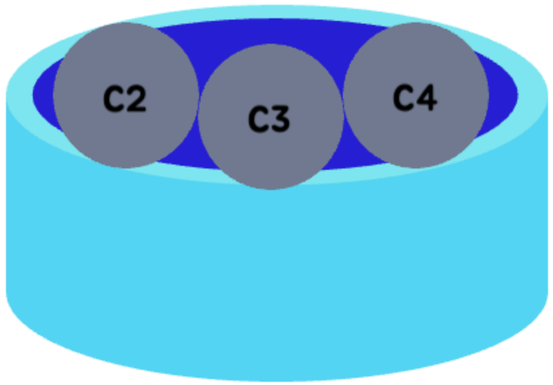
데이터 소스가 Stream.iterate 거나 중간 연산으로 limit 을 쓰면 병렬화로 성능 개선을 기대할 수 없다.

파이프라인 병렬화는 limit 이 있을 때, CPU 코어가 남는다면 윈소를 몇개 더 처리한 후 제한된 개수 이후의 결과를 버려도 아무런 해가 없다고 가정한다.

- 윈소 하나를 계산하는 비용이 대략 그 이전까지의 윈소 전부를 계산한 비용을 합친 것만큼 든다.
- 계속해서 버려지면서 자동 병렬화 알고리즘이 제 기능을 못하게 마비되게 된다.

쿼드 코어 시스템에서 19번째 계산이 마치고 마지막 20번째 계산이 수행되는 시점에

CPU 코어가 3개가 한가한 경우

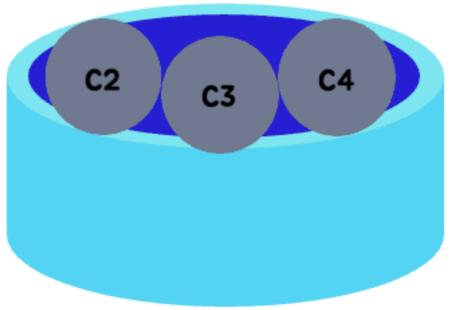


C1

```
public static void main(String[] args) {
    long startTime1 = System.currentTimeMillis();
    primes()
        .parallel()
        .map(p -> TWO.pow(p.intValueExact()).subtract(ONE))
        .filter(mersenne -> mersenne.isProbablePrime(50))
        .limit(20)
        .forEach(mp -> System.out.println(mp.bitLength() + ": " + mp));
    System.out.println(System.currentTimeMillis() - startTime1);
}
```

잘못된 병렬 스트림의 문제점

CPU 코어가 남는다면 원소를 몇개 더 처리한 후 제한된 개수 이후의 결과를 버려도 아무런 해가 없다고 가정한다.

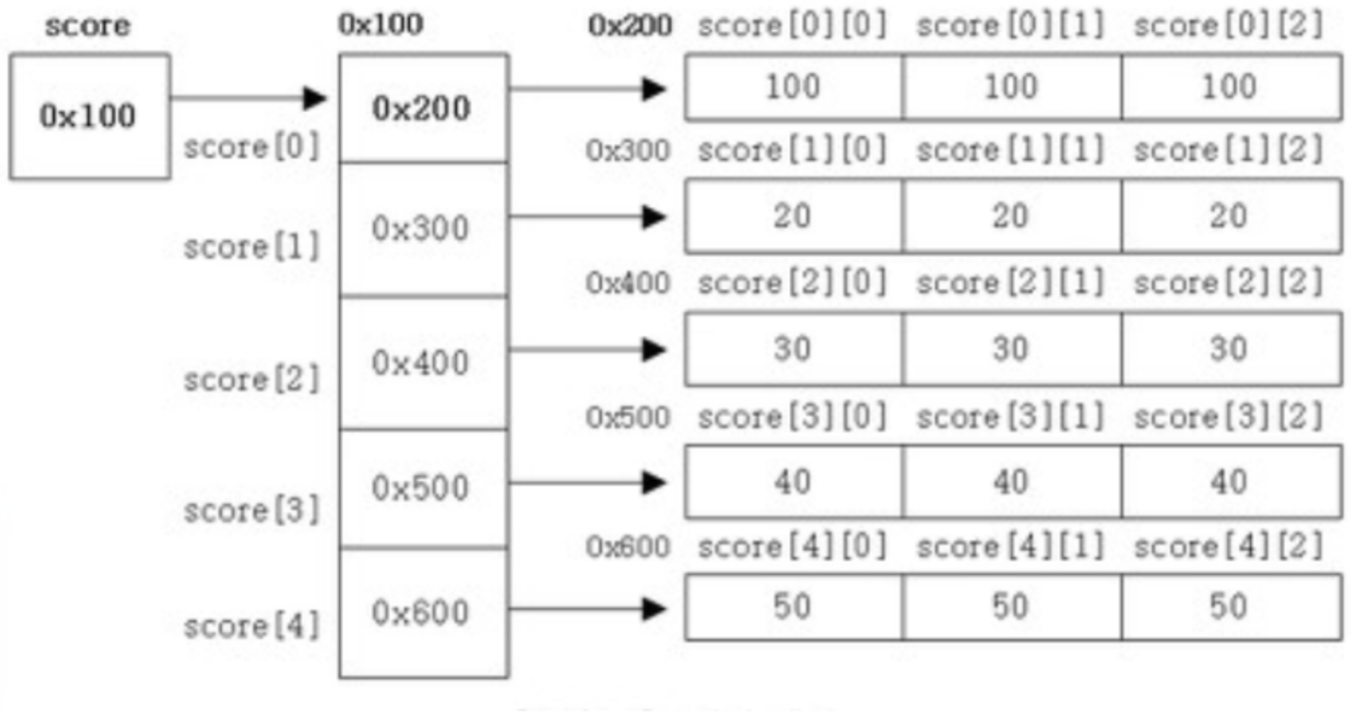


20번째 계산이 끝나더라도 이 계산은 끝나지 않으며 각각 20번째 계산보다 2배, 4배, 8배의 시간이 더필요해진다.

- (이전 원소 전부를 계산한 비용 * 2)

CONTENTS

1. 스트림의 소스



스트림의 소스가 ArrayList, HashMap, HashSet, ConcurrentHashMap의 인스턴스거나, 배열, int 범위, long 범위 등 쪼개기 쉬울 때 병렬화의 효과가 가장 좋다.

1. 스트림의 소스

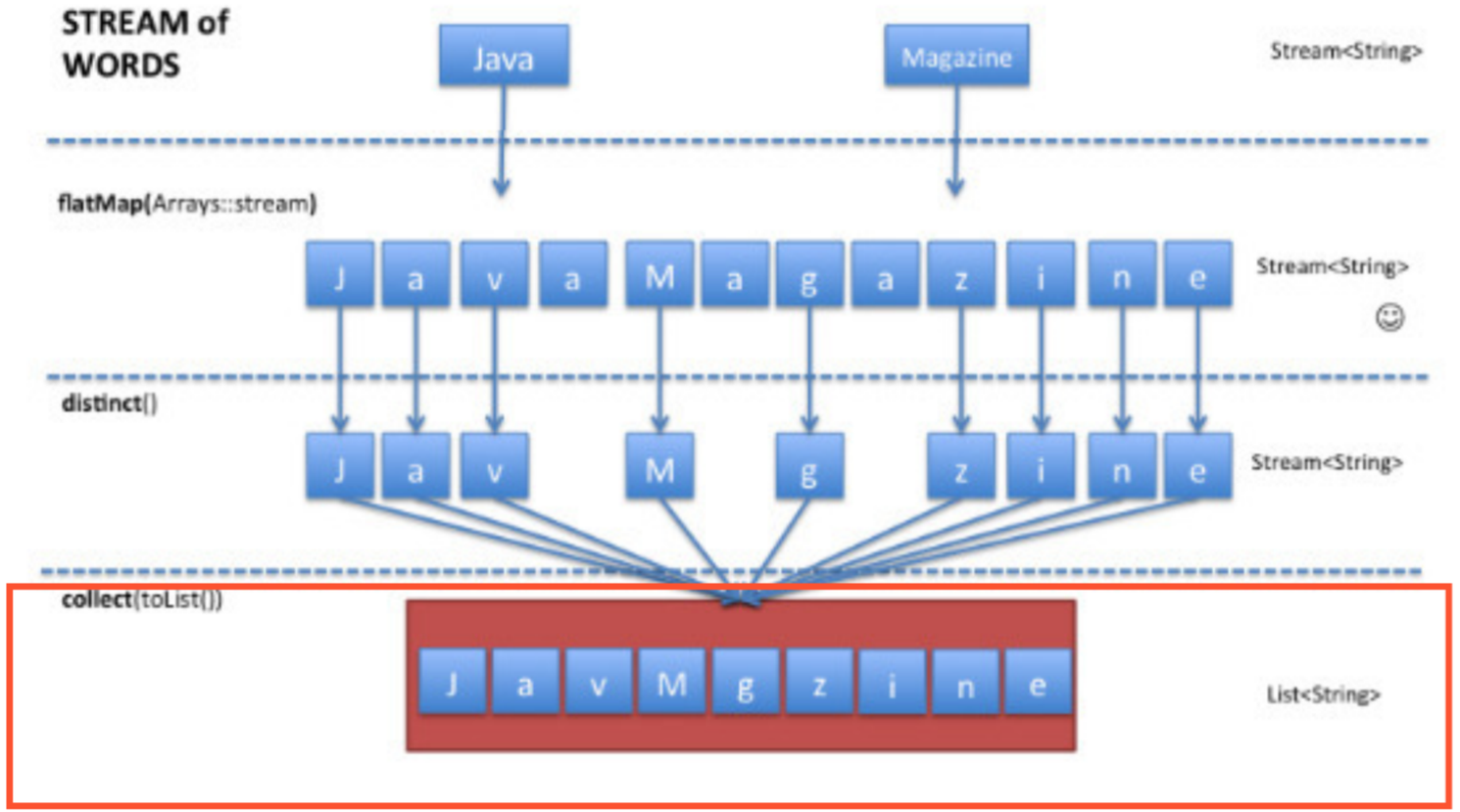
이웃한 원소의 참조들이 메모리에 연속해서 저장되어 참조 지역성이 높기 때문이다.

- 참조 지역성이 낮으면, 스레드는 데이터가 주 메모리에서 캐시 메모리로 전송되어 오기를 기다리며 시간을 보내게 된다.
- 기본 타입 배열인 경우 참조가 아닌 데이터 자체가 메모리에 연속해서 저장되기 때문에 참조 지역성이 제일 좋다.

데이터를 원하는 크기로 정확하고 손쉽게 나눌 수 있어서 스레드에 분배하기 좋다.

- 나누는 작업은 Spliterator 가 담당하며 Stream이나 Iterable의 spliterator 메서드로 얻을 수 있다.

2. 스트림 파이프라인의 종단 연산



종단 연산에서 수행하는 작업량이 **파이프라인 전체 작업에서 상당 비중**을 차지하면서
순차적인 연산인 경우 병렬 수행의 효과가 제한된다.

2. 스트림 파이프라인의 종단 연산

종단 연산 중 병렬화에 가장 적합한 것은 축소(reduction)다.

- 완성된 형태로 제공되는 메서드 (min, max, count, sum)
- 조건에 맞으면 바로 반환하는 메서드들 (anyMatch, allMatch, noneMatch)
- Stream의 collect 메서드는 컬렉션들을 합치는 부담이 크기 때문에 병렬화에 적합하지 않다.

2. 스트림 파이프라인의 종단 연산

```
public long pi(long n) {
  return LongStream.range(2, n)
      .parallel() // 3~5배  정도 성능 개선
      .mapToObj(BigInteger::valueOf)
      .filter(i -> i.isProbablePrime(50))
      .count();
}
```

일정 부분씩 쪼개서 isProbablePrime()의 결과에 따라 나누면 되기 때문에 병렬화 가능하다.

CONTENTS

주의 사항

spliterator()를 반드시 재정의하고 스트림의 병렬화 성능을 강도높게 테스트 후에 병렬화를 적용해야 한다.

Stream의 reduce 연산에 건네지는 accumulator 와 combiner 함수는 반드시 결합 법칙을 지켜야 하며, 간섭받지 않아야 하고, 상태를 갖지 않아야 한다.

파이프라인이 수행하는 진짜 작업이 병렬화에 드는 추가 비용을 상쇄하지 못한다면 성능 향상은 미미하다.

- 스트림 안의 원소 수 * 수행되는 코드 줄이 최소 수십만은 되어야 성능 향상이 가능하다.

정리

- 확신 없이는 스트림 파이프라인 병렬화는 시도조차 하지 말자
- 스트림을 잘못 병렬화하면 프로그램을 오동작하게 하거나 성능을 급격하게 떨어뜨린다.
- 운영 환경과 유사한 조건에서 수행해보며 성능지표를 유심히 관찰하고 확실하면 사용하자