파이프라인 기반의 병렬처리 라이브러리 구현

하승우*
*부경대학교 컴퓨터공학과
zukrukkg@gmail.com

Implementation of Pipeline-Based Parallel Processing Library

Seungu Ha*
*Dept. of Computer Engineering, Pukyong National University

요 약

본 논문에서는 fork-join과 work stealing을 이용하여 동적 병렬처리를 수행하는 라이브러리를 구현하였다. 이 라이브러리는 병렬처리를 직관적으로 할 수 있는 함수형 프로그래밍 스타일의 파이프라인 API를 제공한다. 이를 이용한 성능 테스트에서 멀티코어를 제대로 활용하는 결과를 얻을 수 있었다. 마지막으로 blocking 작업 실행 시 병렬성 유지를 위해 추가로 개선할 수 있는 방법을 제시한다.

1. 서론

최근의 컴퓨터 시스템은 개인용 PC에서도 수십 개의 논리코어(스레드)를 볼 수 있으며 서버 환경에서는 수백 개의 논리코어를 가진 CPU도 찾아볼 수 있다. 프로그램이 멀티코어를 제대로 활용하기 위해서는 전통적인 싱글스레드 프로그래밍이 아닌 멀티코어 프로그래밍이 요구된다.

병렬처리의 기본 접근법은 분할 정복이다. 하나의 작업을 여러 작은 작업으로 분할한 뒤, 분할한 작업들을 여러 스레드로 분배하여 처리한다. 프로그래머가 직접 미리 작업을 분할해서 스레드에 나누어 주는 정적인 방식의 병렬처리는 문제가 간단하다. 그러나 작업이 동적으로 분할되고 (즉 한 작업이 다시 여러 작업으로 나누어질 수 있고) 스레드에 분배되는 동적 병렬처리는 스케줄링 등 처리 방식에 대한 고민이 필요하다.

이에 대응하여 본 논문에서는 동적인 작업 분할을 위해 fork-join 방식을 사용하고, 분할된 작업들을 스케줄링하기 위해 work stealing 방식을 사용하는 병렬처리 시스템을 구현하였다. 그리고 마지막으로 blocking 작업 실행 시에도 병렬성을 유지할 수 있도록 추가적으로 개선할 수 있는 방법을 제시한다.

구현된 프로그램의 소스코드는 [1]에서 확인할 수 있다. 개발 언어로는 Vala를 사용하였다. 자바나 C# 등과 매우 유사하기에 단순 해석에 어려움은 없을 것이라고 생각한다.

2. 관련 연구

2.1 Fork-join

Fork-join 병렬화에서 제어 흐름은 여러 흐름으로 fork(분할)되고 이후 다시 하나의 흐름으로 join(결합)된다. 각 흐름은 독립적이고 각 흐름에서 다시 fork 및 join이 일어날 수 있다. 재귀적인 분할 정복을 구현하기 위해 사용된다. Fork-join을 통해 재귀적으로 분할된 작업들은 서로 독립적이므로 병렬로 처리를 할 수 있다.[2] 그림 1은 중첩된 fork-join 제어 흐름들을 나타낸다.

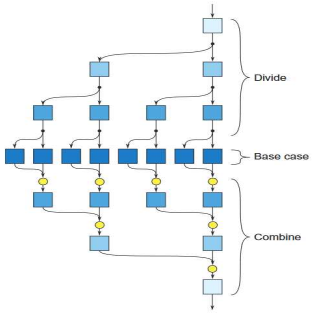

그림 1. 중첩된 fork-join 제어 흐름들[2]

2.2 Work stealing 및 work stealing dequeues

Work stealing에서 워커 스레드는 자신의 작업 큐를 가지고 있고 자신의 큐에서 작업을 꺼내 수행한다. 그리고 자신의 작업 큐가 비어 있을 경우 다

른 스레드의 작업 큐로부터 작업을 훔쳐와서 수행한다는 것이 기본 개념이다.[3]

다른 작업 스레드들의 작업 큐도 비어 있어서 가져올 작업이 없는 경우 공동으로 사용하는 서브미션 큐에서 가지고 온다. 워커 스레드가 처리 중인 작업 안에서 동적으로 또 다른 작업이 생성될 경우 그 작업은 해당 워커 스레드의 작업 큐에 등록된다. 스레드 풀의 워커 스레드가 아닌 외부 스레드에서 작업을 제출하는 경우에는 서브미션 큐에 우선 등록된다.

Work stealing dequeue는 work stealing을 효율적으로 구현하기 위하여 전용으로 최적화하여 설계된 Lock-free 큐(덱)이다. 각 워커 스레드가 이 큐를 하나씩 가지게 된다. 원형 버퍼와 CAS(compare and swap) 등의 원자적 연산에 기반한다. 이 큐에서 tail에의 offer 및 poll은 해당 큐를 가진 소유자 스레드가 수행하며, head에서의 poll은 비소유자 스레드에서 수행한다. 보다 세부적인 구현 내용은 WorkQueue.vala 소스코드[1]와 [3]을 참고하라.

3. 구현

3.1. 작업 분할 및 스케줄링

앞서 언급한 Work stealing 방식으로 작업을 스케줄링한다. 작업을 처리하는 워커 스레드들은 각자 작업 큐(work stealing dequeue)를 가지고 있다. 워커 스레드들은 워커 스레드 풀에서 관리된다. 기본적인 작업 분할 및 스케줄링 동작 과정은 2.1절과 2.2절에서 서술한 것과 동일하다.

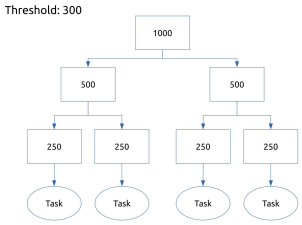

그림 2. Divide and conquer with threshold

다만 무분별한 fork로 인한 성능 하락을 방지하기 위해 각 작업은 해당 작업이 처리해야 하는 데이터 요소의 개수가 *threshold* 값보다 많을 때만 여러 작업으로 fork된다. 위의 그림 2는 *threshold*가 300일 때 1000개의 요소가 어떻게 나누어지는지를 보여준다. 본 논문의 구현에서 *threshold* 값은 아래의 그림 3과 같이 총 요소의 수와 스레드의 수에 기반하여 정해진다.

Default threshold (in DefaultTaskEnv.vala)

```
private const int64 MIN_THRESHOLD = 32768;
private const int64 THRESHOLD_UNKNOWN = 4194304;

public override int64 resolve_threshold (int64 elements, int threads) {
    if (threads == 1) return elements;
    if (elements < 0) return THRESHOLD_UNKNOWN;
    int64 t = threads;
    t = elements / t*2;
    return int64.max(t, MIN_THRESHOLD);
}
```

그림 3. threshold 값 정하기

3.2. Seq 파이프라인

함수형 스타일의 병렬처리 API를 제공하는 핵심 클래스이다. 메소드 체이닝을 통해 하나의 파이프라인을 구성한다. 데이터 소스 - 중간 작업들 - 최종 작업 순서로 구성된다. 그림 3은 Seq 파이프라인의 예시를 보여준다.

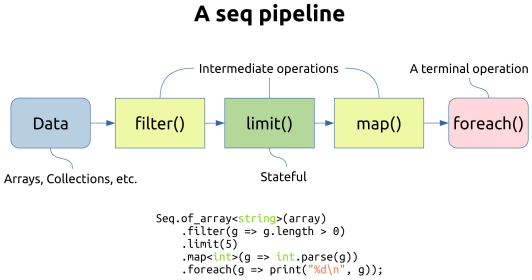

그림 4. Seq 파이프라인

중간 작업들은 stateless 작업과 stateful 작업으로 나뉜다. stateless 작업은 이전 데이터 요소의 처리로부터 어떤 상태도 가져오지 않고 각 요소를 독립적으로 처리한다. stateful 작업은 이전 요소의 상태를 알아야 할 수 있으며 심지어는 결과를 생성하기 위해 전체 요소를 처리해야 할 수도 있는 작업이다. 따라서 stateful 작업을 포함하지 않는 seq 파이프라인은 단일 패스로 처리될 수 있다. 반면 stateful 작업이 포함된 seq 파이프라인은 다중 패스로 처리하거나 중간에 데이터를 버퍼링해야 할 수 있다. 대표적으로 정렬 작업 *order_by()*가 stateful 작업이다. 파이프라인의 끝에 위치하는 최종 작업은 데이터 요소들을 읽어서 결과나 사이드 이펙트를 생성한다.

3.3. 병렬화된 정렬

4절에서 이 시스템을 사용한 정렬 벤치마크를 수

행하며 여기서는 그 알고리즘을 설명한다. 이 시스템에서 정렬을 병렬화하는 방법은 분할 합병 정렬을 응용한 것이다. 하나의 전체 정렬 작업은 여러 개의 부분 정렬 작업으로 나뉘어 처리되고 이후 다시 추가적인 합병(merge) 작업으로 합병된다. 잎 노드 작업의 정렬은 Timsort 알고리즘을 사용한다.

4. 평가

4.1. 벤치마크

이 시스템을 사용한 병렬화된 정렬과 순차적인 정렬의 처리 속도를 비교하는 벤치마크를 작성하여 실행하였다. 벤치마크의 소스코드는 [1]에서 확인할 수 있다. 벤치마크 실행 환경으로는 Google Compute Engine의 n1-highmem-8을 사용하였다. CPU는 4코어 8스레드이며 메모리는 52GB인 환경이다. 워커 스레드 풀의 워커 스레드 총 개수는 16개로 설정했다.

그림 5는 벤치마크 결과를 그래프로 표시한 것이다. Gpseq.parallel_sort (파란 선)가 3.3절에서 설명한 병렬화된 정렬이다. ArrayList.sort (녹색 선)는 순차적인 Timsort 정렬이다. GenericArray.sort_with_data (빨간 선)는 순차적인 퀵 정렬이다.

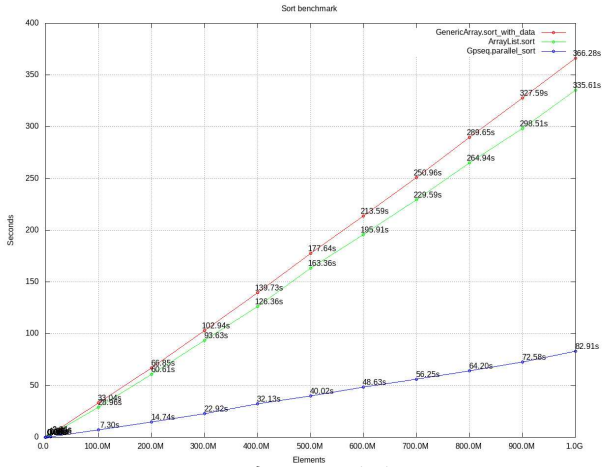

그림 5. 벤치마크

벤치마크 결과를 통해 이 시스템을 사용하여 병렬화된 정렬이 순차 정렬보다 약 4배 정도 속도가 빠른 것을 확인할 수 있다. 실행 환경의 CPU 코어 수가 4개임을 고려하면 멀티코어를 제대로 활용하고 있다고 볼 수 있다.

4.2. 문제점 및 개선 방향

이 시스템에는 앞서 미처 자세히 기술하지 못한 문제가 하나 있다. 워커 스레드 풀이 보유한 워커 스레드의 총 개수가 제한되어 있으므로 만약 워커 스레드가 I/O 작업 등의 blocking 작업을 수행하게 된다면 해당 스레드가 다른 작업을 수행하지 못 하고 놀게 되므로 시스템의 병렬성이 떨어지게 되는 문제가 생긴다. 이를 해결하기 위한 방법으로 Managed Blocking을 제안한다.

워커 스레드가 blocking 작업을 처리하게 될 시 새로운 워커 스레드를 생성하고 기존 스레드의 큐에 있는 남은 작업들을 새로 생성된 스레드로 옮긴다. 그러면 병렬성을 유지하면서 blocking 작업을 수행할 수 있다. 그리고 기존 스레드가 blocking 작업의 수행을 완료할 시 삭제하여 최종적으로는 스레드의 총 개수를 유지한다. 그림 6은 이를 설명한 그림이다.

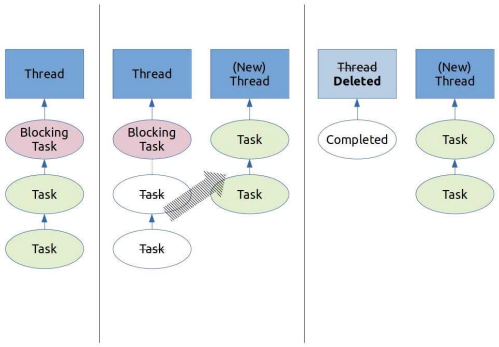

그림 6. Managed Blocking

또한 작업을 코루틴으로 만들고 cooperative 스케줄링 방식으로 처리되게 함으로써 병렬성 문제를 해결하는 것이 가능하다.[4]

5. 결론 및 향후 연구

병렬처리를 직관적으로 할 수 있는 함수형 스타일의 병렬처리 라이브러리를 Fork-join과 Work stealing을 이용하여 구현하였다. 이 시스템을 사용한 정렬 벤치마크에서 멀티코어를 제대로 활용하고 있음을 확인할 수 있었다.

4.2절에서는 본 논문에서 자세히 다루지 못한 시스템의 문제점과 추가적으로 개선할 수 있는 방향을 제시하였다. 향후 다른 시스템과 비교하고 시스템을 더 개선할 수 있는 방안을 연구하고자 한다.

참고문헌

[1] gpseq, https://gitlab.com/kosmospredanie/gpseq,

GitLab, 2019.
[2] M. McCool, J. Reinders, and A. Robison, "Structured Parallel Programming: Patterns for Efficient Computation". Morgan Kaufmann, 2012.
[3] M. Herlihy and N. Shavit, "The Art of Multiprocessor Programming", Morgan Kaufmann, 2008.
[4] gpseq issue - Cooperative scheduling, https://gitlab.com/kosmospredanie/gpseq/-/issues/21, GitLab, 2020.