데이터 집약적인 응용의 자원 사용 분석

*임수빈, 오지선, 김윤희

숙명여자대학교 컴퓨터과학과

{*roky5728, jsoh8088}@gmail.com, yulan@sookmyung.ac.kr

A resource usage analysis of data-intensive application

Su Bin Im, Ji-Sun Oh, Yoonhee Kim

Dept. of Computer Science, Sookmyung Women's University

요 약

자원을 효율적으로 활용하기 위해선 데이터 집약적인 HPC 응용 프로그램의 작업 특성을 파악하는 것이 필요하다. 본 논문에서는 Spark 에서 동작하는 Spark-Bench 의 응용인 Logistic Regression 과 SVD++를 사용하여 입력 데이터 양과 메모리 크기에 따라 Disk I/O, 메모리와 같은 자원 소비 패턴을 파악하여 분석한다. Glance 분석 도구를 사용하여 응용 프로그램의 입력 데이터 양과 메모리 크기에 따라 자원 소비 패턴이 달라짐을 분석하였다.

1. 서론

최근 데이터 집약적인 HPC(High Performance Computing) 응용 프로그램의 특성을 분석하여 실행 제어 연구 및 적응형 데이터 관리 연구가 활발한 추세이다. 자원을 효율적으로 활용하기 위해선 데이터 집약적인 응용 프로그램의 특성 분석에 따른 Disk I/O, 메모리 자원 소비 분석이 중요하다.

본 논문에서는 데이터 집약적인 HPC 응용 프로그램의 작업 특성을 분석한다. Spark-Bench 벤치마크인 Logistic Regression 과 SVD++로 실험한다. 이를 통해 응용의 작업 특성을 분석하고 자원 소비 특성을 파악한다. Glances 분석 도구를 사용하였으며 입력 데이터 양과 메모리 크기에 따라 I/O, 메모리 자원 사용 패턴을 분석하였다.

2. 관련 연구

2.1 Spark-Bench

Spark-Bench[1][3][4]은 Apache Spark 에 맞게 조성된 포괄적인 벤치마크이다. Spark-Bench 의 목적은 사용자가 다른 시스템 설계간의 균형을 이해하고 Apache Spark 를 위한 구성 최적화와 클러스터 프로비저닝을 이해할 수 있도록 돕는다.

실험에 선택한 워크로드는 다른 워크로드와 다른 특성을 보여주고, 서로 다른 시스템 병목현상을 나타낸다. 또한 Spark-Bench 는 Data Generator 를 가지며 이는 임의의 입력 데이터 크기를 생성하여 사용한다.

2.2 Logistic Regression

Logistic Regression[1][2]은 연속 또는 범주형 데이터를 예측하는데 사용한다. 알고리즘은 반복 분류 알고리즘으로 SGD(stochastic gradient descent)를 사용하여 분류모델을 학습한다. 두 세트의 점을 분리하는 초 평면 w 를 찾으려고 시도한다. 먼저 임의의 값에서 w 를 시작하고 반복적으로 w 를 데이터 위에 더하는 방식으로 w 값을 찾는다. 입력데이터 세트는 RDD 추상화를 통해 메모리에 보관되고, 매개 변수 벡터는 반복적으로 계산되고 업데이트된다. 따라서 데이터가 반복적으로 계산 되기 때문에 메모리에 있는 데이터를 캐싱하면 큰 이점이 있다.

2.3 SVD++

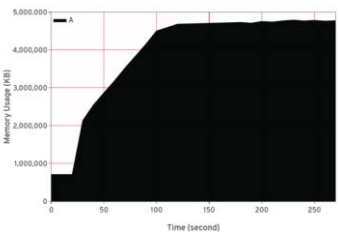
SVD++[1]는 사용자의 피드백을 고려하여 권장 품질을 향상시키는 협업 필터링(Collaborative Filtering Technique) 알고리즘이다. 이 알고리즘은 매 반복마다 모든 에지(Edge)의 바이어스 조건과 가중치 벡터를 계산하여 값을 업데이트한다.

3. 실험 및 분석

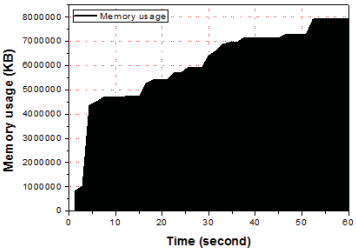
본 논문에서는 데이터를 반복적으로 재사용하는 특성을 가지고 있는 Logistic Regression 과 SVD++로 실험한다. Logistic Regression 과 SVD++ 응용을 분석한 실험 환경은 <표 1>과 같다.

	CPU	Core	RAM	Swap	OS
Name node	Intel(R)Core(TM) i7-4930K CPU @ 3.40GHz	12	3.8GB	3.8GB	Ubuntu 14.04.5 LTS
Worker Node	Intel(R) Core(TM) i7 CPU 950 @ 3.07GHz	8	23GB	23GB	Ubuntu 14.04.5 LTS

<표 1> HPC 응용 분석 실험 환경



<그림 1> Executor Memory 5GB, 입력 데이터 크기 7.5GB Logistic Regression 의 메모리 사용량

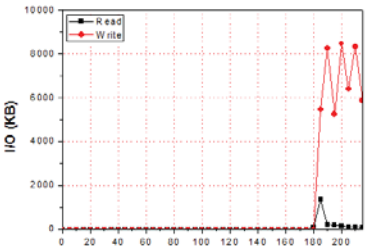


<그림 2> Executor Memory 20GB, 입력 데이터 크기 0.08GB SVD++의 메모리 사용량

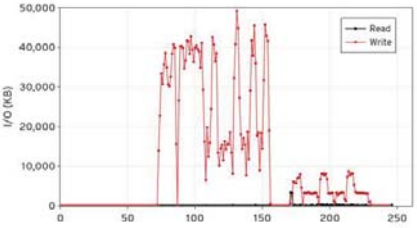
그림 1 은 Logistic Regression 을 실행했을 때 5s 마다 기록한 메모리 사용량이다. 메모리 사용량은 최소 704M, 최대 4773MB 이다. 그림 2 는 SVD++을 실행했을 때 1s 마다 기록한 메모리 사용량이다. 메모리 사용량은 최소 817MB, 최대 7,902MB 이다. 두 응용 다 실행 시간이 지남에 따라 메모리 사용량이 증가한다.

3.1 메모리 크기 조절

동일한 입력 데이터 크기 환경에서 Executor Memory 크기를 조절하여 실험하였다.

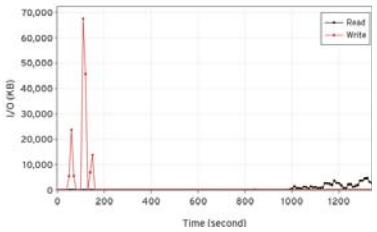


<그림 3> Executor Memory 10GB, 입력 데이터 크기 7.5GB Logistic Regression 의 I/O 패턴

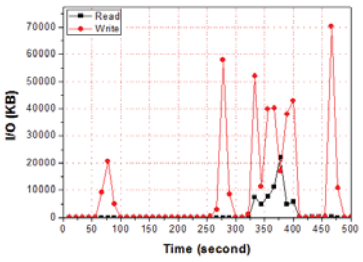


<그림 4> Executor Memory 5GB, 입력 데이터 크기 7.5GB Logistic Regression 의 I/O 패턴

Logistic Regression 의 Executor Memory 를 각각 10GB, 5GB 로 조절하여 실험하였다. 매 5s 마다 기록하였으며 실험의 Read, Write I/O 결과는 그림 3, 그림 4 와 같다. 그림 3 의 경우, Read 연산은 최대 1.3MB, Write 연산은 최대 8.4MB 이며 전체 실행시간은 3m 35s 이다. 실행 실행 180s 후에 Write I/O 양이 5.4MB 로 급격하게 증가하는 busty 한 패턴을 보인다. 그림 4 의 경우, Read 연산은 최대 3.404MB, Write 연산은 최대 49.216MB 이며 전체 실행 시간은 4m 6s 이다. 실행 70s 후 Write I/O 양이 급격하게 증가하며, 이는 Executor Memory 부족으로 디스크부터 RDD 저장을 수행함으로써 발생한다.



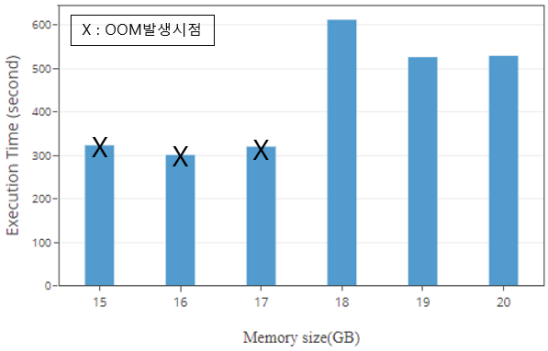
<그림 5> Executor Memory 10GB, 입력 데이터 크기 1.4GB SVD++의 I/O 패턴



<그림 6> Executor Memory 20GB, 입력 데이터 크기 1.4GB SVD++의 I/O 패턴

SVD++에서 Executor Memory 를 각각 10GB, 20GB 로 조절하여 실험하였다. 그림 5 와 그림 6 은 매 5s, 10s 마다 Read, Write I/O 를 기록한 결과이다. 그림 5 의 경우, Read 연산은 최대 12.20MB, Write 연산은 최대 67.554MB 이다. 이 때 6m 6s 에서 처음 OOM(Out of Memory)이 발생하며, 이는 graph data 생성 단계에서 Executor Memory 부족으로 발생한다. 그림 6 의 경우, Read 연산은 최대 22.08MB, Write 연산은 최대 70.42MB 이다. 전체 실행 시간은 8m 49s 으로 정상적으로 종료된다.

앞 선 실험을 통해 SVD++의 Executor Memory 의 경계점을 찾기 위해 실험을 진행하였다. 1.4GB 입력데이터와 메모리 크기를 조절하여 실행한다. 그림 7 은 메모리 크기별 실행 시간과 실행 성공여부를 나타낸다. 메모리 15GB, 16GB, 17GB 일 때 OOM 이 발생하며, 메모리 크기가 18GB 일 때 10m 12s 로 실행이 완료된다. 이에 따라 SVD++의 입력 데이터 1.4GB 일 때 메모리 크기가 18GB 을 경계선으로 실행이 성공됨을 확인할 수 있다.



<그림 7>SVD++의 메모리 크기 별 실행 시간

4. 결론

본 논문에서는 Spark 에서 동작하는 Spark-Bench 중 Logistic Regression 과 SVD++을 분석했다. Glance 도구를 사용하여 입력 데이터 양과 메모리 크기에 따라 Disk I/O, 메모리 자원 사용 패턴을 분석하였다. 입력 데이터 양과 메모리의 크기에 따라 Disk I/O, 메모리 자원 사용 패턴이 달라진다. 현재 그리고 향후에 데이터 집약적인 HPC 응용 프로그램을 위한 data-aware 스케줄링 변화를 예측하는 데 활용이 가능하다.

ACKNOWLEDGMENT

이 논문은 2017 년도 정부(미래창조과학부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임 (NRF-2017R1A2B4005681)

참 고 문 헌

[1] Li, Min, et al. "Sparkbench: a comprehensive benchmarking suite for in memory data analytic platform spark." Proceedings of the 12th ACM International Conference on Computing Frontiers. ACM, 2015.

[2] Zaharia, Matei, et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing." Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation. USENIX Association, 2012.

[3] Spark-bench Github Repository, https://github.com/gioenn/spark-bench

[4]Marco, Vicent Sanz, et al. "Improving spark application throughput via memory aware task co-location: a mixture of experts approach." Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference. ACM, 2017.