CS492 Distributed Systems & Algorithms
Final Report: **Mr.CL**

20030767 최종욱, 20050145 김준기, 20060080 김민국

2009 년 12 월 24 일

1 Introduction

우리는 지난 학기 동안 분산 시스템의 기본적인 구조와 MapReduce, 그리고 분산 시스템 상에서 이용되는 여러 알고리즘에 대해 공부해왔다. 특히나 3 가지 프로젝트를 통해 Hadoop 을 이용한 MapReduce 프레임워크 기반으로 실제 프로그램을 구현해 보면서 이에 관해 더욱 깊은 이해를 할 수 있었다. 이를 통해서 MapReduce 프레임워크가 임의의 규모로 클러스터를 확장시키기 편리하여 대용량의 데이터를 처리는 데 무척 용이함을 알게 되었다. 우리는 기존에 해결하기 어려웠지만 Hadoop 을 통해 새로운 접근 방법을 시도해볼 수 있는 문제를 찾다가 matrix multiplication 이라는 주제를 선택하였다.

Matrix multiplication 은 과학 연산과 데이터 마이닝 등 여러 분야에서 많이 이용되는 기본적인 연산이며 매우 방대한 크기의 matrix 를 이용해 계산하는 경우가 자주 발생한다. 이 때문에 이를 최적화 시키기 위한 많은 시도가 이루어져 왔는데, 우리는 MapReduce 를 통해서 임의의 대용량의 matrix 를 가지고 multiplication 을 수행하는 알고리즘을 개발하고자 한다.

이렇게 방대한 크기의 matrix 를 다룰 수 있게 하는 것도 중요하지만, 연산 수행에 필요한 시간을 단축키는 노력 또한 함께 이루어져야 한다. 따라서 더 빠르게 계산을 수행할 수 있도록 matrix multiplication 과정에서 GPU 에 의한 가속 기능을 도입하였다. GPU 는 그 성격상 실수와 matrix 연산에 특화되어 일반 CPU 보다 훨씬 더 나은 성능을 보여주기 때문이다.

이러한 아이디어를 바탕으로 하여, 우리는 GPU 를 이용한 분산 시스템 상에서의 Hadoop 을 이용한 matrix multiplication 알고리즘을 설계하였다. 분산 시스템을 통해서 대용량의 데이터에 대한 확장성 (scalability) 을 얻고, GPU 를 통해 연산의 속도 향상을 이룸으로써 이전보다 개선된 matrix multiplication 을 수행하고자 한다.

2 Design Overview

2.1 Assumptions

우리가 목표로 하는 시스템의 효과적인 설계와 구현을 위해 약간의 가정이 필요하다.

- Matrix 의 각 원소들은 single precision floating point format 을 통해서 표현된다. CUDA 라이브러리에서 double precision 을 지원하기는 하지만 JCublas 와 연동할 때 내부적인 변환에 따른 성능 저하 및 심각한 계산 오류가 발생하였다.[1]

- 각 matrix 는 dense matrix 이다. Sparse matrix 의 경우 모든 원소를 연속적으로 저장하지 않고 개별적인 쌍으로 저장하기 때문에 원소를 접근할 때 index 를 key 로 하는 탐색 작업이

[1] 파일럿 테스트 과정에서 무작위로 생성한 100×100 행렬을 이용해 곱셈을 했을 때 single precision 으로 입력을 주어 계산하면 한 원소가 250 정도의 값을 가지는데 double precision 의 경우 0 에 매우 가까운 실수가 되어 correctness 판정을 통과하지 못했다.

필요하다. 이것은 GPU 처럼 연속적인 메모리를 할당하여 한꺼번에 pipelining 하는 구조에 적합하지 않으므로 우리는 dense matrix 인 경우만 고려하였다.

2.2 Block Approach

Dense matrix multiplication 을 수행할 때 I/O 병목에 의해 계산 속도가 제약되는 이유는 $O(n^3)$ 의 곱셈 횟수와 $O(n^2)$ 의 element 를 불러오는 횟수가 matrix 가 커질수록 큰 차이가 나기 때문이다. 특히 이러한 I/O 병목 문제는 multi-processor 나 네트워크 기반 클러스터로 규모를 확대할수록 더욱 심각해지는데, 이는 프로세서끼리 혹은 클러스터 상의 컴퓨터끼리의 통신 대역폭이 제한되어 있기 때문이다. 따라서 우리는 프로세서의 성능을 최대한으로 활용하기 위해 프로세서가 처리할 데이터가 가능한 한 지속적으로 공급될 수 있도록, 혹은 데이터를 한 번 불러왔을 때 가능한 한 많이 반복적으로 연산에 활용하는 알고리즘과 코드를 작성해야 한다.

이 문제를 해결하기 위해 우리는 matrix 를 block 단위로 쪼개어 적절히 이를 분산시켜 multiplication 을 수행하는 방법을 선택하였다. 만약 각 block 들이 적당히 큰 상황이라면, 우리는 각 노드 상에서 이러한 block 들의 각 element 들을 한 번 불러와 적당한 횟수만큼 multiplication 에 다시 이용할 수 있을 것이다. 이때 각 block 들은 일반적으로 square matrix 지만 아닌 경우도 있을 수 있으므로 알고리즘에 따라서 적절한 block decomposition 을 구현하는 것이 중요하다.

2.3 NVIDIA CUDA

실험에 사용된 클러스터 상의 모든 노드에는 NVIDIA 의 그래픽 카드가 장착되어 있다. NVIDIA 에서는 CUDA 라는 기술을 통해 개발자들이 C 언어로 직접 GPU 에서 동작하는 프로그램을 만들 수 있도록 하고 있다. 이것을 기반으로 BLAS API[2]를 구현한 CUBLAS 를 이용하면 matrix multiplication 을 GPU 를 이용하여 수행할 수 있다.

2.4 Apache Hama Project

프로젝트 초기에는 matrix multiplication 구현 자체에 대한 부담을 줄이고 GPU 를 이용한 가속에 집중하기 위해 이미 Hadoop 기반으로 다양한 matrix 연산을 구현하고 있는 Apache 재단의 Hama 프로젝트를 이용하였다. 하지만 다음과 같은 이유로 Hama 프로젝트를 사용하지 않고 직접 Hadoop MapReduce 로 구현하게 되었다.

- Hama 가 Hadoop 기반 데이터베이스인 HBase 를 이용하고 있었고, 아직 HBase 가 성능 측면이나 프로젝트 자체의 성숙도가 떨어져 실제 우리가 원하는 규모의 연산을 돌리기에는 적합치 않았다.

- 또한 Hama 에서는 matrix 의 원소들을 기본으로 double precision 으로 처리하고 있는데 여기에 CUDA 를 연동시키려면 single precision 으로 변환하는 과정이 필요하여 불필요한 성능 저하를 발생시켰고 이렇게 동작하도록 Hama 의 코드 전체를 바꾸는 것이 용이하지 않았다.

3 Implementation

TODO: 최종적으로 Hadoop 위에 바로 코딩해서 올린 구조 설명

[2]Basic Linear Algebra Subprograms. Vector-Vector, Vector-Matrix, Matrix-Matrix 연산들을 단계별로 정의하고 있다.

3.1 Data Model

3.2 MapReduce Execution

4 Performance Analysis

4.1 Pure Java

4.2 JCublas with NVIDIA CUDA

5 Conclusion

TODO: 부딪힌 문제점들 설명 (Hama와 HBase의 설치 중 삽질한 것, CUDA 드라이버 관련 삽질한 것, JCublas에서 double/float 삽질한 것 등)

TODO: 앞으로 개선할 점과 계속해서 연구한다면 어떤 것들을 해보고 싶다 등등