Data Parallel Haskell을 이용한 희소 행렬의 성능 분석

송지원[O1], 변석우[2], 우균[34*]
[1]부산대학교 전기전자컴퓨터공학과, [2]경성대학교 컴퓨터공학과,
[3]부산대학교 전기컴퓨터공학부, [4]LG전자 스마트제어센터
jiwon@pusan.ac.kr, swbyun@ks.ac.kr, woogyun@pusan.ac.kr

Performance Analysis of Sparse Matrix Using Data Parallel Haskell

Jiwon Song[O1], Sugwoo Byun[2], Gyun Woo[34*]
[1]Department of Electrical and Computer Engineering, Pusan National University,
[2]School of Computer Science & Engineering, Kyungsung University,
[3]School of Electrical and Computer Engineering, Pusan National University,
[4]Smart Control Center of LG Electronics

요 약

큰 규모의 복잡한 계산을 처리하는 시스템의 필요성이 증가함에 따라 복잡한 계산을 빠르게 처리하는 방법에 대한 연구가 진행되고 있다. CPU를 포함한 하드웨어의 성능이 발전한 결과 멀티코어가 널리 쓰이고 있지만, 소프트웨어는 멀티코어를 제대로 활용하고 있지 못하다. 따라서 이러한 문제를 극복하는 방법으로 병렬 컴퓨팅에 대한 연구가 진행되고 있으며 그중 하나로 병렬 처리를 지원하는 프로그래밍 언어를 사용하는 방법이 있다. 이 논문에서는 Haskell 프로그래밍 언어에서 병렬 프로그래밍 라이브러리를 사용하여 구현한 희소 행렬과 리스트 제시법으로 구현한 희소 행렬을 구현하고 성능을 비교하였다. 그 결과 배열의 크기가 작을 경우 병렬 프로그래밍을 사용한 희소 행렬보다 리스트 제시법을 사용한 희소 행렬의 실행 시간이 더 빠른 것으로 나타났으나, 배열의 크기가 일정이상 커질수록 병렬 프로그래밍을 사용한 희소 행렬의 실행 시간이 더 빠른 것으로 나타났다. 이를 통해 한 배열의 크기가 1000 이상인 희소 행렬을 사용하여 프로그램을 구현할 경우 병렬 프로그래밍을 사용하는 것이 효율적임을 알 수 있다.

1. 서 론

큰 규모의 복잡한 계산을 처리하는 시스템의 필요성이 증가하면서 복잡한 계산을 빠르게 처리하는 방법에 대한 연구가 다양하게 진행되고 있다. 단순히 시스템의 속도를 올리는 방법으로는 성능 향상에 한계가 있으므로 최근 하드웨어의 성능을 향상하는 방법으로 멀티코어를 사용하고 있다. 하지만 현재의 소프트웨어로는 멀티코어를 제대로 활용하지 못하여 기존 소프트웨어의 속도 향상에 한계가 있다. 이를 해결하기 위해 최근 들어 병렬 컴퓨팅에 대한 연구가 진행되고 있다. 병렬 컴퓨팅이란 동시에 많은 계산을 하는 연산의 한 방법으로 크고 복잡한 문제를 작게 나눠 병렬로 처리하는 데 주로 사용한다[1]. 이는 멀티코어 하드웨어를 최대한 활용하는 데 도움을 준다. 또한, 병렬 컴퓨팅은 소프트웨어의 개발에도 큰 영향을 끼쳤다[2].

병렬 컴퓨팅을 해결하는 대표적인 방법은 병렬 프로그래밍을 지원하는 프로그래밍 언어를 사용하는 것이다. 병렬 프로그래밍을 지원하는 프로그래밍 언어는 연산 순서에 제약이 덜한 경우가 많아 계산을 병렬적으로 처리하는 데 적합하기 때문이다. 최근 병렬 프로그래밍에 적합한 언어로 Haskell이 떠오르고 있다. Haskell은 순수 함수형 프로그래밍 언어[3]로 병렬로 처리하여도 항상 같은 결과를 보장하기 때문에 병렬 컴퓨팅에 적합한 언어가 될 수 있다.

이 논문에서는 Haskell로 병렬 프로그래밍을 사용하는 희소 행렬(sparse matrix)을 구현하고 성능을 분석한다. 희소 행렬은 0이 아닌 원소의 수가 매우 적은 행렬이다[4]. 희소 행렬의 연산을 병렬 프로그래밍을 사용하여 원소의 연산을 동시에 수행하도록 한다.

이 논문에서는 배열 데이터를 병렬로 처리할 수 있도록 최적화된 Haskell 라이브러리인 DPH(Data Parallel Haskell) 라이브러리[5]를 사용한 희소 행렬을 구현하였다. 그리고 Haskell의 리스트 제시법(list comprehension)을 사용한 희소 행렬을 각각 구현하였다. 두 희소 행렬을 구현한 후 성능을 측정하여 성능 측정 대상 프로그램의 수행 시간과 메모리 할당량을 비교한다.

이 논문의 구성은 다음과 같다. 2장에서는 관련 연구로서 희소 행렬과 DPH를 설명한다. 3장은 DPH과 리스트 제시법을 사용하여 병렬 프로그래밍을 사용하는 희소 행렬을 구현한다. 4장에는 병렬 프로그래밍을 사용하는 희소 행렬과 리스트 제시법을 사용하는 희소 행렬의 비교 실험을 통하여 프로그램 수행 시간과 메모리 할당량을 비교한 후 5장에서 결론을 맺는다.

2. 관련 연구

2.1 희소 행렬

희소 행렬이란 행렬의 원소 대부분이 0인 행렬을 의미한다. 서로 다른 두 행렬을 연산할 경우 계산 과정에서 행렬의 모든 행과 열을 탐색하여야 한다. 곱셈 과정에서 원소 중 하나가 0인 경우 그 값은 0이 되므로 모든 행과 열을 사용하는 것은 불필요하다. 희소 행렬 중 0이 아닌 원소의 행과 열, 해당 원솟값을 배열에 저장하고 0인 원소를 저장하지 않는다. 이를 이용하면 행렬의 모든 행과 열을 탐색하지 않아도 두 행렬 간 연산이 가능하다. 희소 행렬은 주로 병렬 처리가 필요한 시스템의 개발 및 마이크로 커널의 제작, 이미지 처리와 선형 변환을 이용한 동영상 및 소리의 샘플링에 사용할 수 있다.

2.2 Data Parallel Haskell

Data Parallel Haskell은 배열과 같은 연속된 데이터를 병렬 프로그래밍을 사용하여 처리할 수 있는 Haskell 라이브러리이다. DPH는 다중코어 CPU를 사용하여 중첩 데이터 병렬처리를 지원한다[6]. DPH 라이브러리에는 병렬 행렬(parallel array) 데이터 타입을 제공하여 배열 내 중첩된 데이터의 연산을 수행할 수 있다. 따라서 DPH는 분할 정복 알고리즘[7], 희소 행렬 및 트리와 같은 규칙적이지 않은 데이터를 처리하는 데 적합하다.

3. Data Parallel Haskell을 이용한 희소 행렬 구현

3장에서는 이 논문에서 제안한 Haskell 언어와 DPH 라이브러리를 사용하여 병렬 프로그래밍을 사용하는 희소 행렬을 설계한다. DPH 라이브러리는 배열을 병렬로 처리할 수 있는 병렬 배열 데이터 타입을 제공한다. 일반적인 Haskell의 배열과 다른 DPH 병렬 배열의 특징은 아래와 같다[8].

1) 임의의 타입 a에 대하여 병렬 배열의 타입은 [:a:]로 표기한다.
2) 서로 다른 두 병렬 배열 간의 연산이 가능하다. DPH는 Haskell의 배열 연산자를 지원하며, 대부분의 DPH의 명령어는 Haskell의 명령어와 대응한다. 예를 들어 Haskell 언어에서 배열을 입력받는 map, length 등의 명령어는 DPH 라이브러리에서 mapP, lengthP와 대응한다.
3) 병렬 배열은 Haskell 언어에서 지원하는 리스트 제시법을 지원한다. 리스트 제시법은 Haskell의 배열 데이터를 다루는 방법으로, 수학의 조건 제시법을 사용한 방법이다. DPH 라이브러리는 병렬 배열에서 Haskell의 배열과 같이 리스트 제시법을 사용할 수 있도록 하였으나 이를 병렬로 처리하게 하였다.

DPH 배열의 전체적인 사용법은 다른 Haskell 배열과 큰 차이가 없다. 다만, DPH 배열의 가장 큰 특징은 기존의 Haskell 배열과 달리 병렬 프로그래밍을 지원한다는 점이다. 그리고 DPH 배열에서 원소 중 하나를 삭제할 경우 해당 행렬이 삭제되는데, 이 점은 기존의 Haskell 배열과 다른 점이다[9].

희소 행렬은 0이 아닌 원소와 그 해당 원소의 행과 열의 번호를 저장하여야 한다. Haskell의 경우 리스트의 원소로 리스트를 사용할 수 있는데, 이를 사용하여 0이 아닌 원소와 그 원소의 행과 열의 번호를 리스트로 저장할 수 있다. 희소 행렬 내의 0이 아닌 원소와 그 원소의 열의 원소를 저장하는 변수 mySparser를 기존 Haskell의 타입(type)으로 정의하면 아래와 같다.

```
type mySparser = [(Int, Double)]
```

mySparser는 특정 행의 희소 행렬 원소 중 0이 아닌 원소의 열의 위치와 원소를 각각 정수(Int)와 실수(Double)로 저장한다. mySparser를 사용할 경우 특정 행의 위치를 따로 저장할 필요 없어 희소 행렬 배열을 사용하여 함수를 구현할 경우 희소 행렬 내 mySparser 타입을 제어하는 함수를 구현하기만 하면 된다. mySparser를 저장하는 희소 행렬 mySparseMat을 기존 Haskell의 타입으로 정의하면 다음과 같다.

```
type mySparseMat = [mySparser]
```

DPH 라이브러리는 기존의 Haskell의 리스트와 사용 방법은 비슷하지만, 실행 시 병렬 프로그래밍을 지원한다. 병렬 배열을 사용하여 희소 행렬을 구현하면 행렬의 원소를 사용하여 계산할 시 여러 개의 스레드(thread)로 나누어 계산할 수 있다. 기존의 Haskell로 구현한 mySparser와 mySparseMat을 각각 DPH로 구현한 mySparserP와 mySparseMatP는 다음과 같다.

```
type mySparserP = [:(Int, Double):]
type mySparseMatP = [:mySparserP:]
```

또한, 원소 모두가 실수인 희소 행렬의 곱셈 하는 연산을 구현하면 다음과 같다. 각 행의 모든 열을 곱한 후 그 결과를 합하는 기존 행렬의 곱셈과 달리 희소 행렬의 경우 행렬의 원소인 열과 해당 열의 원소가 저장된 리스트를 내적 연산과 같은 방식으로 계산한다. 희소 행렬의 곱셈 연산은 희소 행렬 sm의 특정한 행 r에서 실수로 되어있는 1차원 배열 v를 곱하여 그 결과를 저장한다. 이를 리스트 제시법을 사용하여 구현한 함수 smMul은 아래와 같이 구현할 수 있다.

```
smMul :: mySparseMat -> [Double] -> [Double]
smMul sm v =
  [sum [ x * (v !! c) | (c, x) <-r] | r <-sm]
```

함수 smMul에서는 희소 행렬 sm의 각 행 r로부터 열의 정보 c와 그 값 x를 가져온다. 그 후 1차원 실수 배열 v의 c번째 위치를 구하여 특정 리스트에 저장한다. r의 모든 원소에 대하여 연산을 마친 다음 계산한 값을 sum 함수를 사용하여 합을 내고 그 결과를 리스트의 원소로 저장한다. smMul은 r의 원소에 대하여 곱셈 과정을 수행한 후 해당 리스트를 반환한다. smMul을 DPH 병렬 배열로 구현한 smMulP은 다음과 같이 구현할 수 있다.

```
smMulP :: mySparseMatP -> [:Double:] -> [:Double:]
smMulP sm v =
  [:sum [ x * (v !: c) | (c, x) <-r] | r <-sm:]
```

4. 실험 및 성능 분석

4장에서는 3장에서 설계한 병렬 프로그래밍을 지원하는 희소 행렬과 리스트 제시법을 사용한 두 희소 행렬을 직접 구현하고 두 프로그램 간의 성능을 비교하는 실험을 하였다. DPH 라이브러리를 사용하여 구현한 희소 행렬과 Haskell의 리스트 제시법을 사용한 희소 행렬의 곱셈 계산 프로그램을 구현한 후, 프로그램 실행 시간, 메모리 할당량을 비교하였다. 성능 비교 시 사용한 운영체제는 Ubuntu 14.04.1이며 4GB Ram 환경에서 실험하였다.

한 열의 개수가 100개, 500개, 1,000개, 2,000개, 3,000개, 5,000개, 10,000개, 100,000개의 2차원 행렬의 곱셈을 각 프로그램을 입력받아 해당 프로그램을 실행한다. 이 논문에서 구현한 프로그램(P1)과 병렬 프로그래밍을 사용하지 않고 리스트 제시법을 사용하여 구현한 비교 프로그램(P2)의 실행 시간과 메모리 할당량을 각각 표 1, 표 2에 제시하였다. 그리고 표 1과 표 2의 에서는 P2 프로그램 대비 P1 프로그램의 비율을 기록하였다.

표 1 P1과 P2의 실행시간(ms) 비교

크기	100	500	1000	2000	3000	5000	10000	100000
P1	3.1	4.6	7.4	14.9	22.4	37.1	81.5	754.1
P2	1.2	4.3	7.8	19.7	38.5	85.2	283.1	38010.3
비율	253.7	107.2	94.9	75.6	58.2	43.6	28.8	2.0

표 2 P1과 P2의 메모리 할당량(mb) 비교

크기	100	500	1000	2000	3000	5000	10000	100000
P1	1.3	3.4	5.9	11.3	16.7	27.4	52.2	512.8
P2	0.6	2.5	4.7	9.6	14.3	23.8	46.5	463.4
비율	216.6	136.0	125.5	117.7	116.7	115.1	112.3	110.7

표 1에서 배열의 크기가 각각 100, 500인 희소 행렬의 경우 P1 프로그램보다 P2 프로그램의 수행 속도가 더 짧다. 하지만 크기가 1,000인 희소 행렬의 경우 P1의 수행속도는 P2보다 약 6% 더 빠르다. 그리고 배열의 크기가 커질수록 P2 대비 P2의

수행 속도 비율은 최대 약 98% 더 빠르다. 배열의 크기가 작을 경우 리스트 제시법을 사용한 희소 행렬 프로그램의 실행 속도가 더 빠르지만, 배열의 크기가 일정 크기 이상이 되는 경우 병렬 프로그래밍을 사용한 희소 행렬 프로그램이 더 빠른 것을 알 수 있다. 메모리 할당량은 P1이 P2보다 더 많은 메모리를 할당받지만, 배열의 크기가 커질수록 P2 프로그램에 대한 P2의 메모리 할당량 비율은 점차 낮아진다.

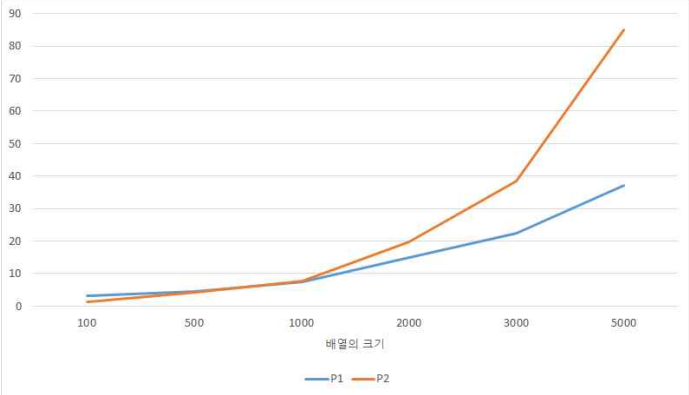

그림 1 P1과 P2의 실행 시간 비교

그림 1에서 볼 수 있듯이, 배열의 크기가 500 이하인 희소 행렬인 경우 병렬 프로그래밍을 사용하는 P1보다 리스트 제시법을 사용하는 P2의 실행 시간이 더 짧다. 하지만 배열의 크기가 500과 1000 사이에 두 프로그램의 수행 시간이 같아지는 시점을 전후로 배열의 크기가 같아진 이후 배열의 크기가 커질수록 병렬 프로그래밍을 사용한 희소 행렬 프로그램의 실행 시간이 더 짧아 결과를 빨리 출력할 수 있다. 즉, 배열의 크기가 작은 경우 리스트 제시법을 사용하는 것이 더 유리하고, 일정 크기 이상 큰 경우 병렬 프로그래밍을 사용하는 것이 유리하다.

5. 결론 및 향후 연구

이 논문에서는 DPH 라이브러리로 병렬 프로그래밍을 사용하는 희소 행렬 프로그램과 리스트 제시법을 사용하여 구현한 희소 행렬 프로그램을 각각 구현하였다. 그리고 구현한 두 프로그램의 실행시간, 메모리 할당량을 비교하였다. 그 결과 수행시간을 비교하였을 때 배열의 크기가 작은 경우 리스트 제시법을 사용한 희소 행렬 프로그램이 더 빠른 것으로 나타났다. 하지만 배열의 크기가 1000 이상일 경우에는 DPH를 사용한 희소 행렬 프로그램이 리스트 제시법을 사용한 희소 행렬보다 6% 더 빠르고, 배열의 크기가 100,000인 경우 최대 98% 더 빠른 것으로 나타났다. 메모리 할당량을 비교하였을 때는 DPH로 구현한 희소 행렬 프로그램이 리스트 제시법으로 구현한 희소 행렬 프로그램보다 더 많은 메모리 할당량을 차지하는 것으로 나타났다. 하지만 배열의 크기가 클수록 두 프로그램의 메모리 할당량의 차이는 점차 작아지는 것으로 나타났다. 이 실험 결과를 바탕으로 한 배열의 크기가 1000 이상인 희소 행렬을 사용하여 프로그램을 구현할 경우, DPH 라이브러리를 사용하는 것이 실행 속도 면에서 좋은 성능을 기대할 수 있다.

향후 연구 과제로 병렬 프로그래밍을 지원하는 다른 언어로 병렬 프로그래밍을 사용한 희소 행렬을 비교하는 연구를 하여 DPH로 구현한 희소 행렬의 성능을 평가하고자 한다. 서로 다른 프로그래밍 언어로 개발한 희소 행렬 프로그램을 비교하여 Haskell의 병렬 프로그래밍 라이브러리의 성능을 객관적으로 평가할 계획이다.

ACKNOWLEDGMENT

본 연구는 미래창조과학부의 SW컴퓨팅산업원천기술개발 사업의 일환으로 수행하였음(B0101-15-0644, 매니코어 기반 초고성능 스케일러블 OS 기초연구).

*교신 저자 : 우균(부산대학교, woogyun@pusan.ac.kr).

참 고 문 헌

[1] A. Grama, A. Gupta, and G. Karypis, *Introduction to parallel computing: design and analysis of algorithms*, Redwood City, CA: Benjamin/Cummings Publishing Company, 1994.

[2] M. Herlihy and N. Shavit, The art of multiprocessor programming, PODC, 2006.

[3] M. Lipovaca, Learn you a haskell for great good!: a beginner's guide, no starch press, 2011.

[4] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," ACM Transactions on Mathematical Software (TOMS), Vol. 38, No. 1, pp. 1-25, 2011.

[5] M. M. Chakravarty, R. Leshchinskiy, S. P. Jones, G. Keller and S. Marlow, "Data Parallel Haskell: a status report," Proceedings of the 2007 workshop on Declarative aspects of multicore programming, ACM. pp. 10-18, 2007.

[6] Manuel M. T. Chakravarty, Gabriele Keller, Roman Lechtchinsky, Wolf Pfannenstiel, "Nepal-nested data parallelism in Haskell." Euro-Par 2001 Parallel Processing, Springer Berlin Heidelberg. pp. 524-534. 2001.

[7] J. L. Bentley, "Multidimensional divide-and-conquer," Communications of the ACM, Vol. 23, No. 4, pp. 214-229, 1980.

[8] K. Jens, L. Dmytro and B. Baldurl, "Data Parallel Haskell: A Tutorial For the Curious Haskeller", 2013.

[9] S. P. Jones, "Harnessing the multicores: Nested data parallelism in haskell," In Proceeding s of the 6th Asian Symposium on Programming Languages and Systems, pp. 138-138, 2008.