Multicore and GPU Programming: Locked Hash Table

Giwon, Seo
Dept. of Computer Science. Yonsei University
2015147531 / mgp10029

1. Self Introduction

kiwon0502@gmail.com

현재 4학년 재학 중이고 머신러닝/AI 분야보다는 CA/OS에 더 큰 관심이 있습니다. 주변 사람보다 전공 지식은 떨어질 수 있지만, 열정만큼은 뒤지지 않습니다. 열심히 하겠습니다. 잘 부탁드립니다!

단순히 딥러닝 모델을 통해 사용만 해보았기 때문에 GPU의 구조를 비롯하여 GPU를 동작하는 방식과 GPU 프로그래밍에 대해 자세히 알고 싶습니다.

2. HW Introduction

Linked List를 활용한 Chaining 방식의 Hash Table[1]에서 contains( ), insert( ), remove( ) 함수를 구현하였다. 위 함수들을 thread를 활용하여 성능을 높이고자 할 때 mutex를 사용해야 한다. 기본적으로 구현된 locked_hash_table의 경우 global mutex를 설정하여 table에 접근할 때 lock을 설정하고 작업이 끝난 이후 lock을 해제한다. 이 방법은 구현하기 매우 간단하지만, 프로그램의 실행 속도가 느리다는 단점이 있다. 이후 내용에서 향상된 실행 속도를 가지는 lock 방식에 대해 살펴보고자 한다.

3. My Approach

Hash Table의 Key를 모아놓은 bucket list에서 각각의 bucket에 접근할 때 lock을 설정하는 fine-grained lock 방식은 이전보다 속도는 빨라지지만, bucket의 수만큼 mutex를 설정하여 공간적 부담이 크다는 단점이 있다. 따라서 bucket의 수만큼 mutex를 설정하지 않고 일정한 상수 N개의 mutex를 설정하여 각 mutex가 (bucket_Number%N)번째 bucket을 관리하는 형태의 Lock Striping 방식을 사용하였다.

4. Experiment

Bucket의 개수는 상수 TABLE_SIZE 값인 1000으로 통제 변인으로써 두었다. 가시성을 위해 걸린 시간(초)은 소수점 셋째 자리에서 반올림하여 계산하였다. 향상 수치의 경우 (1 - 실험값/초기값)*100 (%)로 계산하였다.

4-1. Mutex_num

Thread의 개수는 64개로 고정하고 Mutex 개수인 N값을 조작 변인으로 사용하였다. 1000개의 Bucket을 균등하게 분배하기 위해 1000의 약수를 N의 후보로 지정하였다. 또한, request_cpus를 16으로 설정하였기 때문에 16을 N의 후보로 지정하였다. Bucket 개수와 item 개수가 각각 1000, 5000으로 매우 컸기 때문에 충분히 작은 수와 큰 수를 N으로 설정하여 속도를 비교해보았다.

N	시간(s)	Imp.(%)
Base	1.52	0
10	0.34	77.63
16	0.25	83.55
100	0.09	94.08

4-2. Threads_num

위 결과를 통해 N값이 커질수록 속도가 빨라지지만, 그 차이가 작아지므로 원활한 결과 비교를 위해 N = 10으로 고정하였다. request_cpus 값인 16에 맞게 thread 개수를 설정하여 결과를 비교하였다.

Thread #	시간(s)	Imp.(%)
64 (Base)	0.33	0
32	0.33	0
16	0.18	45.45
8	0.24	27.27

4-3. Items_num

위 실험에서 가장 높은 속도를 보여준 N = 10, Thread 개수 = 16으로 고정한 후 Item 개수를 다르게 하여 better model이 base model보다 어느 정도의 성능 향상을 이루어냈는지 알아보았다.

Item #	Base(s)	Better(s)	Imp.(%)
5000	1.22	0.27	77.87
10000	1.71	0.38	77.78
15000	1.70	0.38	77.65
20000	2.28	0.46	79.82

4-4. Test_num

N = 10, Thread 개수 = 16, Item 개수 = 5000으로 고정한 후 Test 개수를 다르게 하여 better model이 base model보다 어느 정도의 성능 향상을 이루어냈는지 알아보았다.

Test #	Base(s)	Better(s)	Imp.(%)
500000	1.22	0.30	75.41
700000	1.70	0.41	75.88
900000	2.21	0.52	76.47
1000000	2.46	0.60	75.61

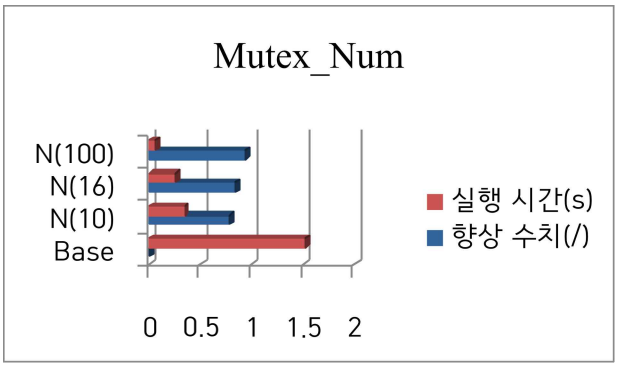
4-5. extra_reads

N = 10, Thread 개수 = 16, Item 개수 = 5000, Test 개수 = 500000으로 고정한 후 extra_reads를 다르게 하여 better model이 base model보다 어느 정도의 성능 향상을 이루어냈는지 알아보았다.

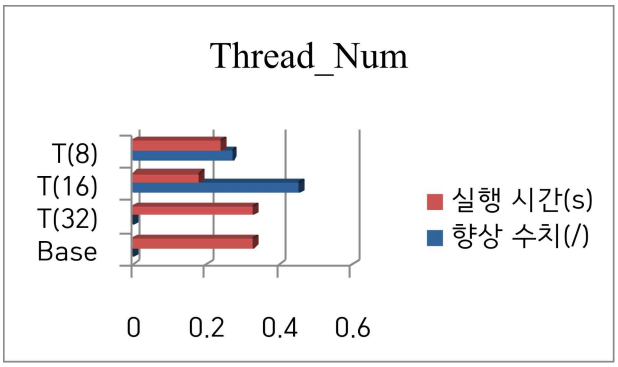
extra_reads	Base(s)	Better(s)	Imp.(%)
10	1.22	0.30	75.41
20	2.04	0.54	73.53
50	4.58	1.16	74.67
100	8.77	2.31	73.66

5. Discussion

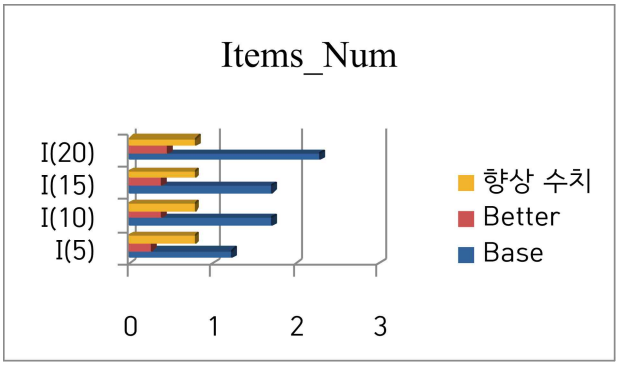
실행 시간과 향상 수치의 가시성을 위해 향상 수치를 백분율이 아닌 비율로 나타내었다.



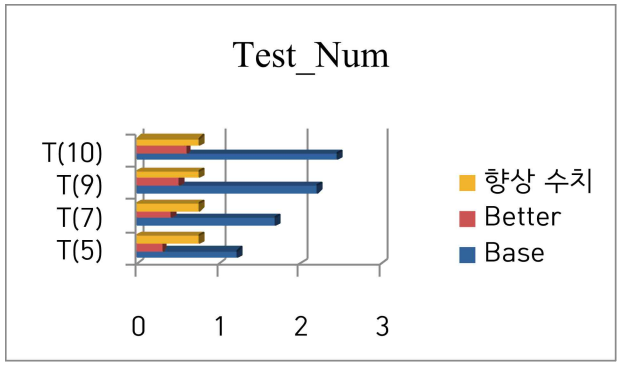
Mutex 배열의 크기를 늘릴수록 동시에 Lock을 처리할 수 있게 되어 눈에 띄게 실행 시간이 줄어드는 것을 확인할 수 있었다. 하지만 N의 크기가 테이블 크기에 가까워질수록 fine-grained lock 방식과 같이 공간적 부담이 커질 것이다.



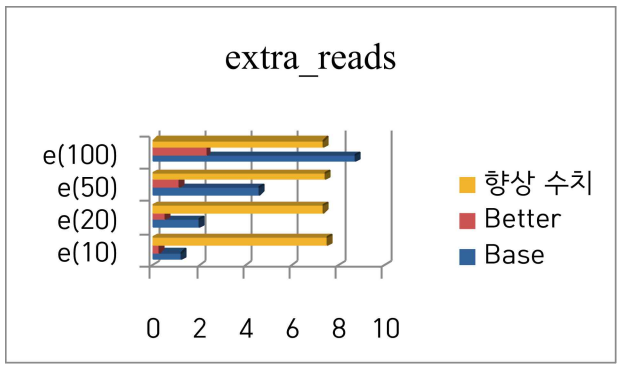
Thread 개수의 경우 request_cpu를 16으로 설정했기 때문에 16개까지는 향상되는 모습을 보이지만 이후 오히려 성능이 하락하는 모습을 보여주었다.



전체적으로 77% 정도의 향상 수치를 보여주고 있다. 이를 통해 Item 개수는 실행 시간의 향상 정도에는 영향을 주지 않는 것을 알 수 있다.



Test 개수에 따른 향상 수치는 전체적으로 75~76%를 유지하고 있지만, Test 개수가 증가할수록 실행 시간이 늘어나는 모습을 알 수 있었다.



extra_reads 또한 값의 변화와 상관없이 전체적으로 73~75%의 향상 수치를 비슷하게 기록하였고 값이 증가함에 따라 실행 시간 또한 증가하였다.

6. Conclusion

Hash Table 전체에 Lock을 거는 “global lock” 방식보다는 mutex 개수를 지정하여 mod 연산을 통해 얻어지는 버킷들에 Lock을 거는 “lock striping” 방식이 효율적이라는 것을 알 수 있다. 이 과정에서 설정한 request_cpu에 따라 thread와 mutex 개수를 적절히 조정하여 더 효율적인 결과를 얻을 수 있다.

7. Reference

[1] Robert Lafore, Data Structures and Algorithms in Java. Sams Publishing, 2015.