멀티코어 숙제6

김지현

모든 과제들은 디렉토리 안에서 **make run** 을 입력하면 컴파일부터 실행까지 모두 이뤄진다.

# 1. 행렬 곱셈

이하 모든 실험들은 **2048\*2048** 크기를 기준으로 수행되었다.

**i->k->j** 순서로 루프를 돌았을 때, 스레드 수에 따른 성능향상은 아래와 같았다. 이전 과제들과 크게 다르지 않게, 스레드 수가 많아질수록 성능이 향상되는것을 알 수 있다.

스레드 숫자를 32개로 놓고 실험하는경우, 간혹가다가 소요시간이 40초 이상과 같이 결과가 튀는 경우가 있는데, 이러한 결과는 걸러내었다.

|  |  |
| --- | --- |
| **스레드 수** | **성능** |
| 1 | 9.46048 |
| 2 | 2.43917 |
| 4 | 2.36918 |
| 8 | 1.33443 |
| 16 | 1.12647 |
| 31 | 0.97994 |
| 32 | 0.97474 |

omp\_set\_num\_threads(31);

#pragma omp parallel for

for (size\_t i = 0; i < width; ++i) {

for (size\_t k = 0; k < width; ++k) {

for (size\_t j = 0; j < width; ++j) {

result[i\*width + j] += lhs[i\*width + k] \* rhs[k\*width + j];

}

}

}

## 루프에 따른 병렬화 성능비교

|  |  |  |  |
| --- | --- | --- | --- |
| **스레드 수** | **i** | **j** | **k** |
| 1 | 13.3262 | 26.1869 | 19.5702 |
| 2 | 12.5457 | 13.0891 | 15.3761 |
| 4 | 1.95438 | 6.93723 | 15.7761 |
| 8 | 2.09495 | 3.76947 | 22.2628 |
| 16 | 2.14756 | 2.39504 | 35.0341 |
| 31 | 2.1448 | 2.15896 | 59.374 |
| 32 | 2.009 | 2.01432 |  |

**i->j->k** 순서로 루프를 돌았을 때, 어디서 병렬화를 하느냐에 따라 성능이 아래와 같이 차이가 났다. i, j 루프의 병렬화는 서로 성능이 비슷한 반면, k 루프는 병렬화시키면 오히려 성능이 나빠지는것을 알 수 있다. 이는 k 루프는 덧셈을 하는 루프이기때문에 제대로 병렬화시킬 수 없고, 루프문 안에서 수행하는 연산이 매우 단순하기때문에 시퀀셜로 실행하는것이 더 빠르기때문이다.

**i**루프와 **j**루프를 병렬화 시키는 코드는 각각 아래와 같다

omp\_set\_num\_threads(16);

#pragma omp parallel for

for (size\_t i = 0; i < width; ++i) {

for (size\_t j = 0; j < width; ++j) {

for (size\_t k = 0; k < width; ++k) {

result[i\*width + j] += lhs[i\*width + k] \* rhs[k\*width + j];

}

}

}

omp\_set\_num\_threads(16);

for (size\_t i = 0; i < width; ++i) {

#pragma omp parallel for

for (size\_t j = 0; j < width; ++j) {

for (size\_t k = 0; k < width; ++k) {

result[i\*width + j] += lhs[i\*width + k] \* rhs[k\*width + j];

}

}

}

**k**루프의 경우, 병렬로 계산한 결과를 모두 더해야하기때문에 특별한 조치가 필요하다. 나는 **OpenMP Reduction**을 사용하였다. 코드는 다음과 같다.

omp\_set\_num\_threads(16);

for (size\_t i = 0; i < width; ++i) {

for (size\_t j = 0; j < width; ++j) {

float sum = 0.0f;

#pragma omp parallel for reduction (+:sum)

for (size\_t k = 0; k < width; ++k) {

sum += lhs[i\*width + k] \* rhs[k\*width + j];

}

result[i\*width + j] = sum;

}

}

제일 안쪽 루프를 Reduction으로 병렬화시키면 계산 결과가 살짝 다르게 나온다. 부동소수점 표기법은 라운딩 오프 에러때문에 덧셈 순서에 따라 오차가 발생할 수 있기 때문이다.

## #pragma omp collapse

이 Directive는 자명한 형태의 중첩 for문을 하나로 합쳐서 병렬화시켜준다. 코드는 아래와 같다.

#pragma omp parallel for collapse(2)

for (size\_t i = 0; i < width; ++i) {

for (size\_t j = 0; j < width; ++j) {

for (size\_t k = 0; k < width; ++k) {

result[i\*width + j] += lhs[i\*width + k] \* rhs[k\*width + j];

}

}

}

32개 스레드로 2048\*2048 행렬에 대해 계산하였을때의 결과는 **2.071607**초로, collapse를 쓰지 않았을 경우에 비해 큰 차이는 나지 않았다. 이는 행렬의 크기가 충분히 크기때문에 2중 for문을 한줄로 편다고 하여 병렬화에 큰 이득이 없기때문이다.

# 2. K-means

16개의 센트로이드와, 1048576개의 데이터를 기준으로 실험하였을때, 스레드 수에 따른 성능변화는 아래와 같았다.

|  |  |
| --- | --- |
| **스레드 수** | **성능** |
| 1 | 20.2399 |
| 2 | 14.6675 |
| 3 | 12.0489 |
| 4 | 12.0863 |
| 8 | 8.3352 |
| 16 | 6.86469 |
| 31 | 6.87173 |
| 32 | 6.85892 |