멀티코어 컴퓨팅 HW #6

2010-11904
최재민

0. OpenMP

OpenMP 를 사용하기 위해서는 먼저 소스 파일에 #include <omp.h>를 추가하여야 하며, Makefile 의 컴파일 옵션(CFLAGS)에 -fopenmp 를 추가하고, OpenMP 라이브러리 함수를 사용하기 위해 LDFLAGS 에 -lgomp 를 추가하여야 한다.

사용되는 thread 의 개수를 explicit 하게 설정하기 위해서는 먼저 dynamic threading 을 해제하는 함수인 omp_set_dynamic(0) 을 호출해야 하며, omp_set_num_threads(TNUM) 함수를 사용하여 thread 의 개수를 TNUM 으로 설정할 수 있다. (TNUM 은 코드의 앞 부분에 define 되어 있다) 이렇게 설정하면 추후에 #pragma omp parallel 을 사용할 때 TNUM 개의 thread 가 실행되게 된다.

1. Matrix Multiplication

1.1. 병렬화하는 for loop 의 변화에 따른 성능 측정

먼저 여러 단계의 Nested loop 중 어떤 loop 를 #pragma omp for 로 병렬화해야 가장 좋은 성능이 나오는지 확인하기 위해 naive 한 행렬곱 코드를 사용하여 실험을 해 보았다.

```
#pragma omp parallel for private(j,k)
    for (i = 0; i < NDIM; i++) {
        for (j = 0; j < NDIM; j++) {
            for (k = 0; k < NDIM; k++) {
                c[i][j] += a[i][k] * b[k][j];
            }
        }
    }
```

<그림 1-1. Naive: i loop 병렬화>

```
    for (i = 0; i < NDIM; i++) {
#pragma omp parallel for private(k)
        for (j = 0; j < NDIM; j++) {
            for (k = 0; k < NDIM; k++) {
                c[i][j] += a[i][k] * b[k][j];
            }
        }
    }
```

<그림 1-2. Naive: j loop 병렬화>

```
    for (i = 0; i < NDIM; i++) {
        for (j = 0; j < NDIM; j++) {
#pragma omp parallel for
            for (k = 0; k < NDIM; k++) {
                c[i][j] += a[i][k] * b[k][j];
            }
        }
    }
```

<그림 1-3. Naive: k loop 병렬화>

Loop / Time (sec)	Test 1	Test 2	Test 3	Average
i loop	1.956215	1.993645	2.161665	2.037175
j loop	4.111727	4.460577	4.326716	4.299673

k loop	Over 100 seconds

〈표 1-1. Naive: k loop 병렬화〉

실험 방법은 현재는 comment 되어 있는 mat_mul.c 의 mat_mul 함수의 naive 부분을 사용하여 실험하였으며, 행렬 한 dimension 의 크기인 NDIM 은 2048, thread 의 개수인 TNUM 은 64 로 설정하였다. 이 상태에서 make run 명령을 통해 천둥 노드에서 실행한 결과를 cat task*를 통해 얻었다. 모든 경우 validation 은 성공적으로 수행되었다.

위에 표로 정리한 실험결과를 보면 알 수 있듯이, 점점 더 inner loop 를 병렬화 할 수록 성능은 급격히 나빠진다. 전체적인 일의 양은 세 경우 모두 NDIM * NDIM * NDIM / TNUM 으로 같은데 왜 성능이 나빠지는 걸까? 잘 생각해보면 parallel 하게 실행되는 loop 의 위치와 크기에 그 이유가 있음을 알 수 있다. k loop 의 경우를 보면, 가장 inner loop 인 k loop 를 여러 thread 들이 쪼개서 수행하게 되는데, 문제는 이런 상황이 i loop 의 횟수 (NDIM) * j loop 의 횟수 (NDIM) 만큼 반복된다는 것이다. NDIM * NDIM 횟수만큼 thread 를 fork 하고 join 하는 overhead 가 매우 크며, 반면에 이러한 overhead 가 없는 i loop 의 경우 j loop 나 j loop 보다 더 빠른 성능을 보여주게 되는 것이다. 즉, thread 들이 수행하는 loop 의 크기가 크면 클수록 병렬화가 효과적으로 이루어지고 더 좋은 성능을 보여주게 된다.

1.2. #pragma omp for collapse

다음은 #pragma omp for collapse 를 사용하는 것을 설명하도록 하겠다. 일단 collapse(CNUM)을 설정하게 되면 여러 for loop 이 중첩되어 있을 때 CNUM 만큼의 for loop 들을 합친 영역만큼을 나누어 thread 들에게 할당하게 된다. 예를 들어 다음과 같은 코드가 있고, 10 개의 thread 가 생성된다고 가정하자.

```
#pragma omp parallel for collapse(2)
    for (i = 0; i < 10; i++) {
        for (j = 0; j < 5; j++) {
            // do something
        }
    }
```

〈그림 1-4. Collapse 의 사용〉

위 코드에서 만약 collapse(2)를 쓰지 않고 대신 private(j)를 썼다면 10 개의 thread 가 i 를 나누어 하나씩 담당하고, 각 thread 가 j loop 를 수행할 것이다. 하지만 collapse(2)를 사용하면 2 개의 loop 를 합쳐주므로 총 50 개의 i, j 값들을 10 개의 thread 에 나누어 수행하게 된다. 이는 특히 tiling 을 사용할 때 유용한데, tiling 을 사용할 때에는 각 tile 이 하나의 ii 와 jj 쌍에 의해 결정되고 이러한 tile 1 개 이상을 thread 하나가 수행해야 가장 효과적이기 때문이다. 다음 코드를 보자.

```
#pragma omp parallel for collapse(2) private(kk,i,j,k)
    for(ii = 0; ii < NDIM; ii += BDIM) {
        for (jj = 0; jj < NDIM; jj += BDIM) {
            for (kk = 0; kk < NDIM; kk += BDIM) {
                iend = MIN(ii + BDIM, NDIM);
                for (i = ii; i < iend; i++) {
                    jend = MIN(jj + BDIM, NDIM);
                    for (j = jj; j < jend; j++) {
                        kend = MIN(kk + BDIM, NDIM);
                        for (k = kk; k < kend; k++) {
                            c[i][j] += a[i][k] * b[k][j];
                        }
                    }
                }
            }
        }
    }
}
```

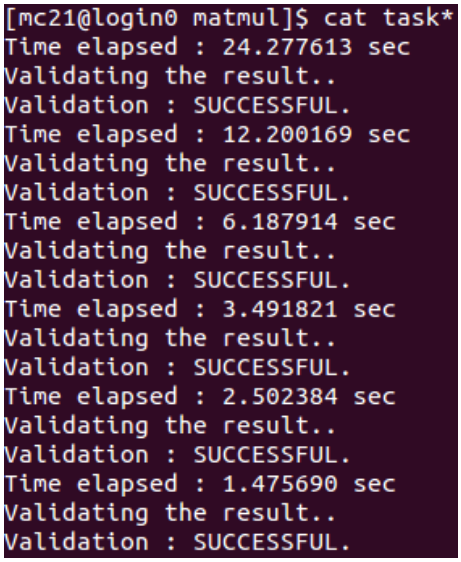
〈그림 1-5. Collapse 를 사용한 tiling 코드〉

위 코드에서는 collapse(2)를 사용하여 thread 들이 나눠가지는 iteration space 를 ii 와 jj 로 확장했으

며, 이로 인해 앞에서 설명한 tiling 이 가능해졌다. 즉, 하나의 thread 가 kk loop 이하를 실행하기 때문에 tiling 의 장점인 data reuse 를 exploit 할 수 있는 것이다. (물론 thread 의 개수가 부족할 경우 하나의 thread 가 하나 이상의 ii & jj pair 를 수행할 수는 있다)

1.3. Thread 개수 변화에 따른 성능 측정

Thread 개수 변화에 따른 성능 측정을 위해서 앞서 구현한 tiling 코드를 사용하였고, NDIM 은 2048, tile block dimension 의 크기를 의미하는 BDIM 은 256 로 설정하였다. 이와 같은 설정으로 thread 개수를 1 개에서 2 배씩 늘려가며 32 까지 실험한 결과는 다음과 같다.



<그림 1-6. Thread 개수에 따른 성능 측정 결과>

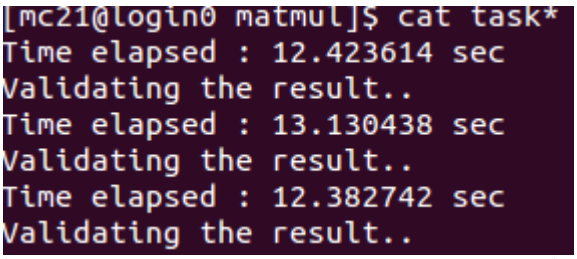
Thread 개수	실행시간 (sec)
1	24.277613
2	12.200169
4	6.187914
8	3.491821
16	2.502238
32	1.475690

<표 1-2. Thread 개수에 따른 성능 변화>

위 표를 보면 thread 의 개수가 2 배씩 증가함에 따라 실행 시간도 약 2 배씩 감소하는 것을 볼 수 있는데, 이는 strong scalability 가 잘 유지됨을 보여준다. 하지만 thread 의 개수가 증가할수록 실행시간이 감소하는 폭이 줄어드는데, 이는 병렬화 오버헤드 등의 이유로 thread 개수의 증가에 따른 성능 향상에 한계가 있기 때문에 나타나는 현상이다.

1.4. 주어진 문제 크기에서의 실험 결과

행렬곱의 경우 주어진 문제 크기는 4096 X 4096 이므로, NDIM 을 4096, tiling block 의 크기인 BDIM 은 512, thread 의 개수인 TNUM 은 32 로 설정하고 실험을 진행하였다. 3 번의 실험을 거친 결과는 다음과 같으며, 충분히 빠른 시간 내에 행렬곱을 수행한다는 것을 알 수 있다. Validation 작업은 시간이 너무 오래걸려 수행하지 않고 kill 하였다.



〈그림 1-7. 주어진 문제 크기에서의 실험 결과〉

2. Kmeans

2.1. 실행방법 및 코드 설명

OpenMP 를 사용하는 Kmeans 알고리즘은 kmeans_omp.cpp 에 정의되어 있으며, 이를 제대로 컴파일하고 실행하기 위해 Makefile 을 약간 변경하였다. Makefile 에서 kmeans_seq 는 그대로이지만 이제 pthreads 대신 OpenMP 를 사용하기 때문에 이름을 omp 로 변경하였고, 관련된 파일들의 이름도 suffix 에 omp 가 붙도록 하였다. 프로그램 실행을 위해서는 먼저 make 명령으로 executable 파일들을 생성해 주고, make run 명령을 실행하면 gen_data.py 를 실행하여 65536 개의 data point 와 64 개의 centroid point 를 가지는 데이터를 생성한 뒤 thor 에 enqueue 하게 되며, 생성되는 class 파일은 result_omp.class, point 파일은 final_centroid_omp.point 파일이 된다. 이들을 이용해 그래프를 생성하고자 할 때는 make graph_omp 명령을 실행하면 되고, 그 결과로 result_omp.png 파일이 생성된다. Thread 의 개수를 수정하고자 할 때는 보고서의 가장 처음 부분에서 설명했듯이 #define TNUM 부분을 수정해주면 된다.

새로 정의한 kmeans_omp.cpp 파일은 #pragma omp 를 사용한다는 점을 제외하고는 sequential 코드인 kmeans_seq.cpp 와 상당히 유사한데, 한 가지 큰 차이점은 기존의 centroids 와 count 대신에 kmeans 함수 내에서 새로 정의한 acc_centroids 와 acc_count 를 사용한다는 것이다. 이 두 변수는 malloc 을 이용하여 centroids 와 count 가 가리키는 메모리 공간 크기의 TNUM 배를 할당해주는데, 이는 update step 에서 partitioned[data_i]가 index 로 쓰이는 것으로 인한 문제를 방지하기 위한 것이다. 각 thread 는 이 부분에서 모든 thread 가 공유하는 centroids 와 count 를 update 하지 않으며, 대신 acc_centroids 와 acc_count 에서 자기 thread 에게 배정된 메모리 공간에 update 를 하게 된다. 이런 방법을 사용하면 mutex lock 등을 이용하지 않고 모든 thread 들이 병렬적으로 계산 작업을 수행할 수 있다. 이 작업이 끝나면 acc_centroids 와 acc_count 에 퍼져있는 값들을 가장 앞쪽으로 모아주는 accumulation 작업이 필요한데, 이 작업은 class_n * (TNUM - 1) 만큼의 loop iteration 이 필요하므로 1 개의 thread 만이 수행하도록 해도 큰 무리는 없다. 하지만 class 의 개수나 thread 의 개수가 증가하면 이 부분의 overhead 도 무시할 수 없게 된다고 생각하여 병렬화 방안을 곰곰히 생각해 본 결과, class_i 를 각 thread 에게 나누어 주면 acc_centroids 와 acc_count 의 index 가 동시에 사용될 일이 없어 병렬화가 가능하다는 것을 깨닫고 이 방식으로 구현하였다. 이후에는 앞 부분에 모아준 값들을 count 로 나누어 주고 다음 loop 를 실행하면 된다. #pragma omp for 의 끝에는 implicit 하게 barrier 가 있으므로 thread 들이 함께 실행되도록 하기 위해 따로 barrier 를 사용할 필요는 없다. 모든 iteration 이 끝나면 마지막으로 acc_centroids 에 저장된 centroid 데이터를 기존의 centroids 가 가리키는 메모리 공간에 옮겨주면 된다.

2.2 Thread 개수 변화에 따른 성능 측정

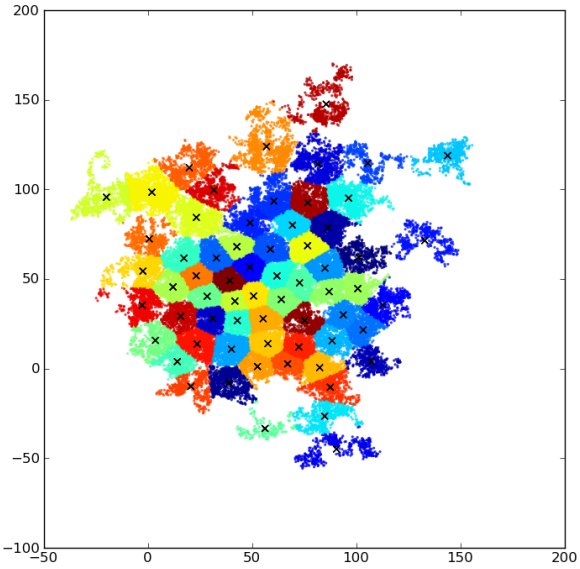
Thread 개수	실행시간 (sec)
1	9.539614945
2	4.836981084
4	2.473430637
8	1.406254950
16	1.068733690
32	0.784435558

〈표 2-1. Thread 개수에 따른 성능 변화〉

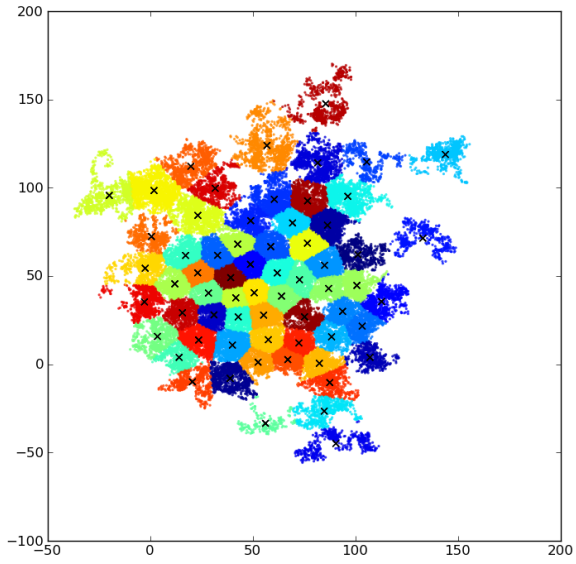
본 성능 측정은 65536 개의 data point, 64 개의 centroid 를 1024 번의 iteration 에 걸쳐 계산한 결과이다. 위 결과 표를 보면 알 수 있듯이 thread 의 개수가 2 배씩 증가할 수록 실행시간은 감소하나, 정확히 2 배씩 빨라지지는 못한다. 그 이유는 kmeans 알고리즘이 thread 의 개수에 대해 완전히 scalable 하지 못하기 때문이고, 중간중간 sequential 하게 수행되야 하는 부분들이 있기 때문이다.

2.3. 주어진 문제 크기에서의 실험 결과

2.2 번과는 다른 65536 개의 data point, 64 개의 centroid point, 1024 번의 iteration 에 대해 32 개의 thread 를 사용하여 kmeans_omp 를 실행한 결과 0.754872335 초가 걸렸으며, kmeans_seq 를 실행한 결과 9.562214595 초가 걸렸다. 이를 보면 OpenMP 를 이용하여 병렬화한 코드가 월등히 빨리 kmeans 알고리즘을 수행한다는 것을 알 수 있다. 각 경우 그래프를 그린 결과는 다음과 같으며 OpenMP 를 사용한 결과가 sequential 한 경우와 같다는 것을 알 수 있다.



〈그림 2-1. kmeans_seq 의 결과 그래프〉



〈그림 2-2. kmeans_omp 의 결과 그래프〉