Multiple Regression Parallelize Project

OpenMP Project

2014136007 김기백, 2014136040 노유찬, 2017136069 유지훈

Multi-core Programming (CPH 351) 김덕수 교수님

1 Introduction

본 프로젝트에서는 강의를 통해 학습한 OpenMP를 활용하여 다중 회귀 프로그램을 병렬화 한다. 우리가 선정한 이 프로그램은 2개 이상의 독립 변수를 가진 전체 데이터의 경향을 나타낼 다변수 함수를 도출해내는 프로그램이다. 이는 기존의 다항 회귀 프로그램[1]을 기반으로, 데이터 정리 및 계산이 같은 부분을 활용하여 구축하였다. 또한 회귀에 사용할 입력 데이터로 League of Legends Diamond Ranked Games (10 min)[2] 을 사용한다. 게임 시작 10분간의 데이터를 담은 39개의 column, 9880개의 데이터를 이용하여 그 성능을 검증한다.

2 Solution

2.1 Multiple Regression

다변수 함수 도출 과정

n 개의 독립 변수를 가지는 N개의 데이터

$$(a_1, b_1, \dots, n_1, f_1), (a_2, b_2, \dots, n_2, f_2), \dots, (a_N, b_N, \dots, n_N, f_N)$$

에 대한 다중 회귀 방정식은 다음과 같이 표현된다.

$$z(a, b, \dots, n) = c_0 + c_1 a + c_2 b + \cdots + c_n n$$

[1] chrisengelsma/PolynomialRegression.h

[2] https://www.kaggle.com/bobbyscience/league-of-legends-diamond-ranked-games-10-min

각 데이터에서의 오차 $\mathrm{r_i}$ 는 측정치 f_i 와 추정치 $z_i(a_i, b_i, \dots, n_i)$ 의 차이로 표현한다.

$$\mathrm{r_i} = z_i(a_i, b_i, \dots, n_i) - f_i$$

오차 제곱의 합을 S라 하고, 이가 최소일 조건은 각 독립 변수에 대한 편미분 값이 0일 때이다.

$$\mathrm{S} = \sum_{i=1}^{N} (r_i)^2 = \sum_{i=1}^{N} (c_0 + c_1 a_i + c_2 b_i + \cdots + c_n n_i - f_i)^2$$

$$\frac{\partial \mathrm{S}}{\partial \mathrm{c_0}} = 0, \frac{\partial \mathrm{S}}{\partial \mathrm{c_1}} = 0, \dots \ , \frac{\partial \mathrm{S}}{\partial \mathrm{c_n}} = 0$$

이는 다음과 같이 정리할 수 있다.

$$\frac{\partial \mathrm{S}}{\partial \mathrm{c_0}} = 2 \sum_{i=1}^{N} (c_0 + c_1\, a_i + c_2\, b_i + \cdots + c_n\, n_i - f_i\,) = 0$$

$$\frac{\partial \mathrm{S}}{\partial \mathrm{c_1}} = 2 \sum_{i=1}^{N} (c_0 + c_1\, a_i + c_2\, b_i + \cdots + c_n\, n_i - f_i\,) a_i = 0$$

$$\dots$$

$$\frac{\partial \mathrm{S}}{\partial \mathrm{c_n}} = 2 \sum_{i=1}^{N} (c_0 + c_1\, a_i + c_2\, b_i + \cdots + c_n\, n_i - f_i\,) n_i = 0$$

위의 연립방정식을 행렬과 벡터로 표시하면 다음과 같다.

$$\begin{bmatrix} N & \sum a_i & \dots & \sum n_i \\ \sum a_i & \sum a_i^2 & \dots & \sum a_i n_i \\ \dots & \dots & \dots & \dots \\ \sum n_i & \sum a_i n_i & \dots & \sum n_i^2 \end{bmatrix} \begin{Bmatrix} c_0 \\ c_1 \\ \dots \\ c_n \end{Bmatrix} = \begin{Bmatrix} \sum f_i \\ \sum a_i f_i \\ \dots \\ \sum n_i f_i \end{Bmatrix}$$

$$\mathrm{Bc} = \mathrm{Y}$$

위의 연립방정식을 부분 피벗팅을 사용한 가우스 소거법을 활용하여 해를 구하여 각 계수들을 구한다.

구현 코드

MultipleRegression.h 내부의 코드는 크게 두 부분으로 나눌 수 있다. 첫 번째는 연립방정식을 표현하는 B행렬, Y벡터를 만드는 부분, 두 번째는 해당 행렬과 벡터를 붙여 가우스 소거법을 사용하여 계수들이 들어있는 c벡터를 완성시키는 부분이다.

MultipleRegression.h 첫 번째 부분

```
...
//0차, 1차 sigma (a_i, b_i, ...)
X[0][0] = (double)N;
for (int i = 1; i < np1; i++)
    for (int k = 0; k < N; ++k)
        X[0][i] += (TYPE)x[k][i - 1];

//2차 sigma (a_i * b_i ...)
for (int i = 0; i < n; ++i)
    for (int j = i; j < n; ++j)
        for (int k = 0; k < N; ++k)
            X[i+1][(j+1)] += (TYPE)(x[k][i] * x[k][j]);

//계산된 X값들로 B행렬 생성
for (int i = 0; i < np1; ++i) {
    for (int j = 0; j < np1; ++j) {
        B[i][j] = (i <= j) ? X[i][j]: X[j][i];

//Y 벡터 생성
for (int i = 0; i < np1; ++i) {
    for (int j = 0; j < N; ++j) {
        Y[i] += (TYPE)((i==0)?1:x[j][i-1])*y[j];
...
```

코드 설명

1. 입력 데이터의 각 값들을 순회하며 B행렬값의 기초가 될 각 시그마 값을 구해 X행렬을 채운다.
2. 대칭 행렬 형태의 B행렬을 X행렬의 값을 사용해 생성한다.
3. Y행렬 또한 x입력값, y입력값을 활용하여 시그마 값을 구한다.

MultipleRegression.h 두 번째 부분

```
...
// Load values of Y as last column of B
for (int i = 0; i <= n; ++i)
   B[i][np1] = Y[i];

n += 1;
int nm1 = n - 1;

// Pivotisation of the B matrix.
for (int i = 0; i < n; ++i)
   for (int k = i + 1; k < n; ++k)
      if (B[i][i] < B[k][i])
         B[i].swap(B[k]);

// Performs the Gaussian elimination.
// (1) Make all elements below the pivot equals to zero
//     or eliminate the variable.
for (int i = 0; i < nm1; ++i)
   for (int k = i + 1; k < n; ++k) {
      for (int j = 0; j <= n; ++j)
         B[k][j] -= (B[i][j] * B[k][i]) / B[i][i];          // (1)
   }

// Back substitution.
// (1) Set the variable as the rhs of last equation
// (2) Subtract all lhs values except the target coefficient.
```

```
// (3) Divide rhs by coefficient of variable being calculated.
for (int i = nm1; i >= 0; --i) {
    a[i] = B[i][n];                    // (1)
    for (int j = 0; j < n; ++j)
        if (j != i)
            a[i] -= B[i][j] * a[j];    // (2)
    a[i] /= B[i][i];                   // (3)
}

coeffs.resize(np1);                    //계수 출력
for (int i = 0; i < np1; ++i)
    coeffs[i] = a[i];
...
```

코드 설명

1. 가우스 소거법을 위하여 Y벡터를 B행렬의 마지막 새로운 열에 붙인다.
2. Pivotisation. Pivot이라 불리는 행과 열이 같은 원소(B[i][i])를 같은 열의 밑에 있는 (B[i + d][i])원소보다 크도록 첫 Pivot부터 행 전체를 바꾸며 작업을 수행한다.
3. Gaussian elimination. 가우스 소거법을 이용하여 피벗 아래의 모든 원소를 0으로 만든다.
4. Back substitution. 가우스 소거법 이후 자연스레 가장 아래의 행을 이용하면 계수c_n을 구할 수 있다. 이를 활용하여 후진대입을 진행해 모든 계수를 구한다.
5. 계수 벡터를 초기화하고 계산된 계수들을 내보낸다.

기타 설명

위의 과정은 원본 데이터와 계수가 저장될 행렬 및 벡터의 주소 값을 파라미터로 받기 때문에 이외의 데이터 복사, 이동 등의 과정은 존재하지 않는다. 따라서 병렬화를 진행할 때 소요시간을 줄이기 위하여 위의 코드에 OpenMP를 적용하면 될 것이다.

2.2 Parallelized Multiple Regression

병렬화 과정

앞서 살펴본 두 부분에는 데이터 종속성이 존재하기 때문에 두 부분 사이에 Barrier가 존재할 수 밖에 없다.

$$\begin{bmatrix} N & \sum a_i & \dots & \sum n_i \\ \sum a_i & \sum a_i^2 & \dots & \sum a_i n_i \\ \dots & \dots & \dots & \dots \\ \sum n_i & \sum a_i n_i & \dots & \sum n_i^2 \end{bmatrix} \begin{Bmatrix} c_0 \\ c_1 \\ \dots \\ c_n \end{Bmatrix} = \begin{Bmatrix} \sum f_i \\ \sum a_i f_i \\ \dots \\ \sum n_i f_i \end{Bmatrix}$$

$$Bc = Y$$

벡터 및 행렬을 생성하는 첫 번째 부분은 ∑연산의 특성 상 N개의 모든 데이터를 전부 순회하며 그 값을 연산하여 더해야 하기 때문에 이에 특화된 OpenMP for 구문에 사용할 수 있는 reduction clause를 사용하는 것이 좋을 것이다. ∑연산의 또다른 특성으로 그 일의 양이 동일하기 때문에 schedule clause는 사용하지 않아도 결과가 비슷하게 나올 것이다. 데이터 종속성이 없는 부분은 같은 for문 안에서 진행해서 최대한 동시에 진행하면 더욱 효율적으로 작업을 진행할 수 있을 것이다.

가우스 소거법 및 계수를 계산하는 두 번째 부분은 그 종속성이 뚜렷해 병렬화 할 수 있는 곳이 많지 않다. 특히 Pivotisation 연산의 경우 for문으로 행 전체를 반복적으로 교체하기 때문에 이 앞뒤로 Barrier가 필수적이며, 해당 부분 또한 병렬화가 불가능하다. 하지만 단순 비교 및 포인터 교체 연산만 하기 때문에 지연시간이 크지 않기 때문에 안심할 수 있다. 가우스 소거법 및 후진대입 또한 단순한 OpenPM for 구문 및 reduction 이외에 적용할 만한 부분이 없다.

위와 같은 논리로 병렬화를 진행하였으나 특정 구간에서 병목 현상이 심하게 발생하는 것을 각 구간에 timer를 설치하여 소요 시간을 측정해 보고 알게 되었다. 바로 3중 for문이 존재하는 2차 ∑연산이 있는 부분이다.

MultipleRegression.h 첫 번째 부분 2차 ∑연산

```
...
//2차 sigma (a_i * b_i ...)

for (int i = 0; i < n; ++i)
    for (int j = i; j < n; ++j) {
        t = 0;
        #pragma omp parallel for reduction(+: t)
        for (int k = 0; k < N; ++k) {
            t += (TYPE)(x[k][i] * x[k][j]);
        }
        X[i + 1][(j + 1)] = t;
    }
...
```

해당 부분에서 테스트 코드 기준 769ms가 소요되어 총 소요시간의 대부분을 차지하고 있는 것을 알게 되었다. 그 이유는 k for문을 순회하면서 하나의 데이터만 읽고 바로 다음 데이터를 읽어 cache miss 가 많이 발생하기 때문이다. 이 문제를 해결하기 위하여 k루프 안에서 여러 reduction 변수를 두어 한 번에 여러 데이터를 읽게 하여 소요시간 394ms까지 줄이는 것에 성공하였다.

구현 코드

MultipleRegressionParallelized.h 첫 번째 부분

```
std::vector<std::vector<TYPE> > X(np1, std::vector<TYPE>(np1*cache, 0));
// a = vector to store final coefficients.
std::vector<TYPE> a(np1*cache);
// Y = vector to store values of sigma(xi * yi)
std::vector<TYPE> Y(np1*cache, 0);
// B = normal augmented matrix that stores the equations.
std::vector<std::vector<TYPE> > B(np1, std::vector<TYPE>(np2*cache, 0));
...
```

```
for (int i = 0; i < np1; i++) {
    //0차, 1차 sigma, Y
    ta = tb = 0;
    #pragma omp parallel for reduction(+: ta, tb) num_threads(NUMTHREADS)
    for (int k = 0; k < N; ++k) {
        if (i != 0) ta += (TYPE)x[k][i - 1];
        tb += (TYPE)((i == 0) ? 1 : x[k][i - 1]) * y[k];
    }
    if (i != 0) X[0][i * cache] = ta;
    Y[i * cache] = tb;

    //2차 sigma
    if(i != 0)
    for (int j = i; j < np1; j+=8) {
        t = t2 = t3 = t4 = t5 = t6 = t7 = t8 = 0;
        #pragma omp parallel for reduction(+: t, t2, t3, t4, t5, t6, t7, t8)
num_threads(NUMTHREADS)
        for (int k = 0; k < N; k++) {
            t += (TYPE)(x[k][i - 1] * x[k][j - 1]);
            if (j + 1 < np1) t2 += (TYPE)(x[k][i - 1] * x[k][j]);
            if (j + 2 < np1) t3 += (TYPE)(x[k][i - 1] * x[k][j + 1]);
            ...
            if (j + 6 < np1) t7 += (TYPE)(x[k][i - 1] * x[k][j + 5]);
            if (j + 7 < np1) t8 += (TYPE)(x[k][i - 1] * x[k][j + 6]);
        }
        X[i][j * cache] = t;
        if (j + 1 < np1) X[i][(j + 1) * cache] = t2;
        ...
        if (j + 6 < np1) X[i][(j + 6) * cache] = t7;
        if (j + 7 < np1) X[i][(j + 7) * cache] = t8;
    }
}
```

```
#pragma omp parallel for num_threads(NUMTHREADS)
for (int i = 0; i < np1; ++i) {
    for (int j = 0; j < np1; ++j) {
        B[i][j * cache] = (i <= j) ? X[i][j * cache] : X[j][i * cache];
    }
    // Load values of Y as last column of B
    B[i][np1 * cache] = Y[i * cache];
}
...
```

코드 설명

1. Cache coherency 문제를 최소화 하기 위하여 cache값 만큼의 여유를 두어 선언한다.
2. 외곽의 i for 안에서 0차, 1차 시그마 연산 및 2차 시그마 연산을 모두 진행한다.
3. Reduction을 활용하여 X행렬 0, 1차와 Y벡터를 생성한다.
4. 2차 시그마 연산의 지연시간을 줄이기 위하여 하나의 데이터에서 8개의 reduction을 위한 접근을 한다.
5. X행렬이 완료된 후 B행렬을 생성하고, 가우스 소거법을 위한 Y벡터 병합도 함께 진행한다.

MultipleRegressionParallelized.h 두 번째 부분

```
//병렬화 불가
// Pivotisation of the B matrix.
for (int i = 0; i < n; ++i)
    for (int k = i + 1; k < n; ++k)
        if (B[i][i * cache] < B[k][i * cache]) {
            B[i].swap(B[k]);
        }
```

```
// Performs the Gaussian elimination.
for (int i = 0; i < nm1; ++i) {
   TYPE bii = B[i][i * cache];
   #pragma omp parallel for num_threads(NUMTHREADS)
   for (int k = i + 1; k < n; ++k) {
      for (int j = 0; j < np1; ++j) {
         B[k][j * cache] -= (B[i][j * cache] * B[k][i * cache]) / bii;
      }
   }
}

// Back substitution.
for (int i = nm1; i >= 0; --i) {
   TYPE reduc = B[i][n*cache];
   #pragma omp parallel for reduction(-:reduc) num_threads(NUMTHREADS)
   for (int j = 0; j < n; ++j)
      if (j != i)
         reduc -= B[i][j*cache] * a[j*cache];
   a[i*cache] = reduc / B[i][i*cache];
}

coeffs.resize(np1);    //계수 출력
#pragma omp parallel for num_threads(NUMTHREADS)
for (int i = 0; i < np1; ++i)
   coeffs[i] = a[i*cache];
```

코드 설명

1. 병렬화가 불가능한 Pivotisation은 그대로 진행한다.
2. 가우스 소거법을 진행한다. 연산이 겹치지 않는 두 번째 for문에서 병렬화한다.
3. 후진대입을 진행한다. 계수 계산 부분이 시그마 연산 및 나누기 연산으로 되어있어 reduction을 활용한다.
4. 계산된 계수들을 내보낸다.

3 Conclusion

결과 및 분석

39개의 Column 중 38개의 컬럼을 x에, 하나의 Column을 y에 넣고 Multiple Regression을 진행해보았다. 8개의 스레드를 지정하였다.

```
[Available game stats for calculation]
[ 1] blueWins
[ 2] blueWardsPlaced              [21] redWardsPlaced
[ 3] blueWardsDestroyed           [22] redWardsDestroyed
[ 4] blueFirstBlood               [23] redFirstBlood
[ 5] blueKills                    [24] redKills
[ 6] blueDeaths                   [25] redDeaths
[ 7] blueAssists                  [26] redAssists
[ 8] blueEliteMonsters            [27] redEliteMonsters
[ 9] blueDragons                  [28] redDragons
[10] blueHeralds                  [29] redHeralds
[11] blueTowersDestroyed          [30] redTowersDestroyed
[12] blueTotalGold                [31] redTotalGold
[13] blueAvgLevel                 [32] redAvgLevel
[14] blueTotalExperience          [33] redTotalExperience
[15] blueTotalMinionsKilled       [34] redTotalMinionsKilled
[16] blueTotalJungleMinionsKilled [35] redTotalJungleMinionsKilled
[17] blueGoldDiff                 [36] redGoldDiff
[18] blueExperienceDiff           [37] redExperienceDiff
[19] blueCSPerMin                 [38] redCSPerMin
[20] blueGoldPerMin               [39] redGoldPerMin
[100] test

Please Select stats by number to include on independent variables(=X) (input '0' to continue) :
 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35
36 37 38 39 0
Now, select ONE stats to be a range set(=Y) : 1

Press any key to continue...
```

Fig. 1. 모든 컬럼을 다 입력한 모습

```
  7.00   7.00   0.00   0.00   0.00   0.00 16
787.00   7.20 19647.00 233.00  47.00 435.00
1431.00  23.30 1678.70 }  | Y[ 99](    0.00)
| f(x)=       1.974 (p)f(x)=       1.974


Result Correct

*        DS_timer Report        *
* The number of timer = 2, counter = 2
**** Timer report ****
Serial : 18.11500 ms (18.11500 ms)
Parallel : 8.24700 ms (8.24700 ms)
**** Counter report ****
*        End of the report      *
       ×2.20
Press any key to continue...
```

Fig. 2. 위 데이터의 결과

Serial과 Parallel 코드의 결과가 같고 성능이 더 좋게 나오는 것을 볼 수 있었으나 그 효과가 체감되지 않았다. 그래서 기존의 데이터를 활용하여 별도의 Test Data로 병렬화의 효과를 알아보았다. 다음은 400개의 열을 가진 x데이터를 가지고 Multiple Regression을 진행한 결과이다. Y값으로는 임의의 데이터를 입력하였다. 동일하게 8개의 스레드를 지정하였다.

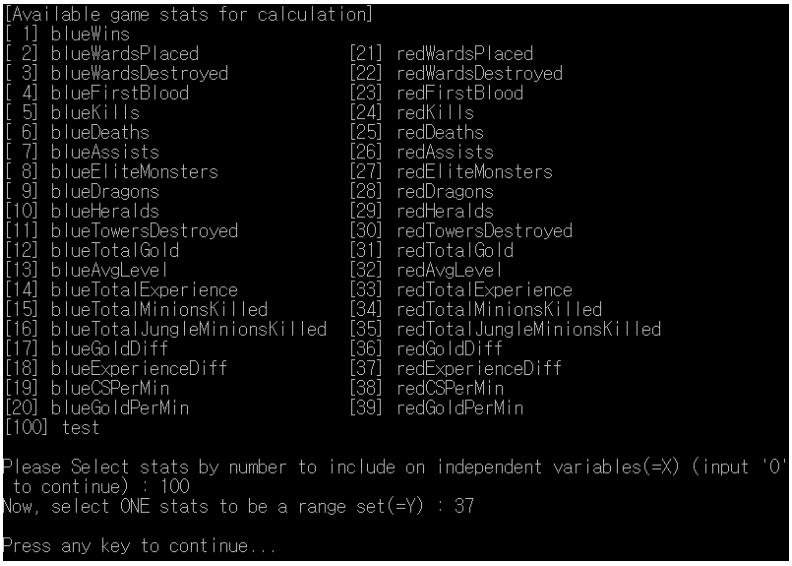

Fig. 3. 400 * 8000 Test data

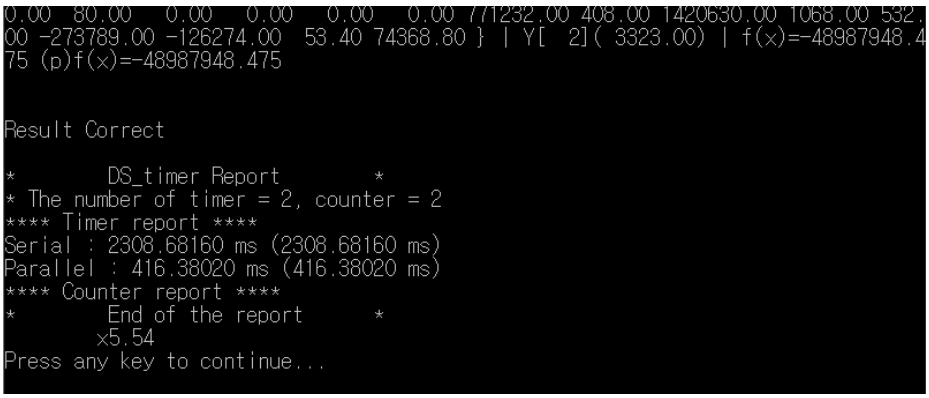

Fig. 4. Test data result

확실하게 연산이 많을 때 병렬화 효율이 올라가는 것을 볼 수 있었다. 하지만 2차 시그마 연산에서 딜레이가 많이 발생하기 때문에 Column 및 data length가 과도하게 커질 경우 그 효율이 떨어질 것으로 보인다.

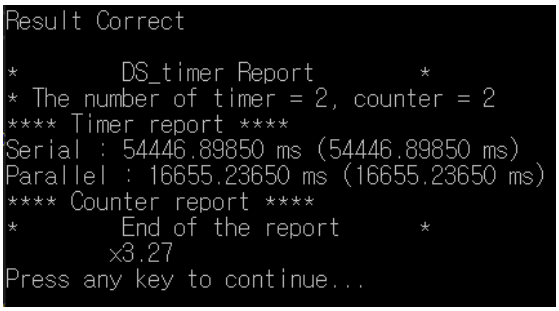

Fig. 5. 1400 * 8000 Test data Result

위와 같이 column 수를 1400개로 늘렸을 때 병렬처리 성능이 시리얼 보다 3.27배로 아까보다 낮은 효율이 나온다. 이를 해결하기 위해서는 2차 시그마 연산의 딜레이를 줄일 수 있는 근본적인 해결책을 제시해야 할 것이다.

그 외 다중회귀를 잘 나타내기 위하여 특정 Column을 조절하여 입력할 수 있게 하였고, 서로 다른 독립변수들 간 관계를 볼 수 있도록 OpenGL로 시각화를 하였다.

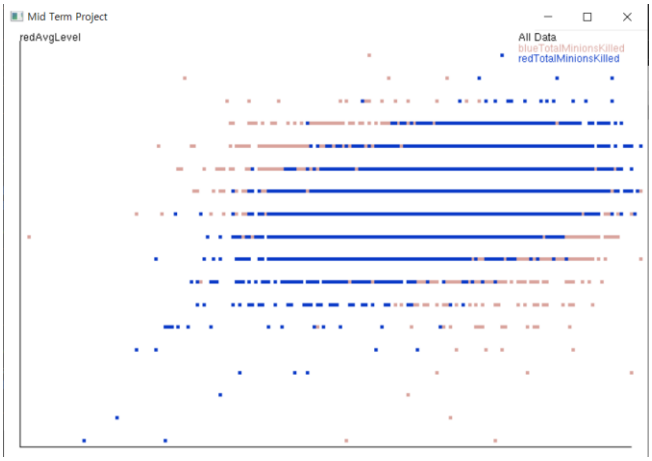

Fig. 6. League of Legends data 시각화 예시

느낀 점

[2014136007 김기백]

다중 회귀라는 생소하지만 매력적인 주제를 가지고 이번 프로젝트를 진행할 수 있어서 좋았다. 회귀 및 데이터 분석 관련 공부를 통해 이러한 데이터들을 처리하여 유의미한 자료를 만들 수 있게 되면 좋을 것 같다.

[2017136069 유지훈]

저명한 온라인 게임 '리그 오브 레전드'의 초반 10분 데이터에 다중 회귀를 적용하여 게임 결과나 유사 시간대 게임 지표들을 사용자가 지정한 독립 변수에 기반하여 예측하는 프로젝트를 진행하고 이를 병렬화하였다.

흥미를 끄는 프로젝트 시나리오에 다중 회귀라는 재미있는 이론을 직접 적용시켜 병렬화하는 것이기에 열의를 갖고 참여할 수 있었다. 가장 흥미로웠던 점은 최소한의 추가 공간을 이용하여 최대 10배의 효율을 낼 수 있었다는 점이다. 준비된 데이터셋의 크기가 조금 부족한 감이 있어서 수억배로 늘어났을 때의 결과는 확인하지 못해 아쉬웠으나 준비된 데이터로 측정한 성능 자체가 예상보다 더 큰 폭으로 향상되었기에 상당히 고무적이라 느꼈다.

[2014136040 노유찬]

복학해서 하는 첫 텀프였다.

처음에는 팀원들에게 민폐를 끼칠까 많은 걱정이 앞섰다. 생각보다 어려운 주제가 결정되었고 오랜만에 보는 복잡한 수식에 머리가 아찔했었다. 잘 이해를 못할 때 팀원들이 친절하게 알려주고 어려운 부분은 도맡아서 해결을 해주었다. 코로나로 인해 서로 대면하지 못하는 상황에서도 각자 모두 열심히 해서 책임감이 치솟는 느낌이다. CUDA 때는 팀이 어떻게 결정될 지 잘 모르지만, 가능하다면 또 다시 같이 하고 싶다. 감사합니다.

References

1. chrisengelsma, PolynomialRegression.h, http://gist.github.com/chrisengelsma/108f7ab0a746323beaaf7d6634cf4add, last accessed 2020/05/02
2. michel's fanboi, League of Legends Diamond Ranked Games (10 min), http://kaggle.com/bobbyscience/league-of-legends-diamond-ranked-games-10-min, last accessed 2020/05/06
3. Terry D. Johnson, Gaussian Elimination with Partial Pivoting (2000), https://web.mit.edu/10.001/Web/Course_Notes/GaussElimPivoting.html, last accessed 2020/05/02
4. Triangular_matrix / Forward and back substitution, https://en.wikipedia.org/wiki/Triangular_matrix#Forward_and_back_substitution, last accessed 2020/05/02
5. Koreatech E-Learning, https://el.koreatech.ac.kr/, last accessed 2020/05/08