멀티코어 프로그래밍

과제 보고서



교 수 : 김 덕 수

학 번 : 2015136021

성 명 : 김 건 용

과 제 : Lab2-1(Matrix-Vector) / Lab2-2(Trapezoidal Rule)

Lab 2-1. Matrix-Vector

과제 : 행렬과 벡터의 곱을 구하는 프로그램 작성

과제 목표

* Parallel Construct, Work-sharing Construct 활용
* 병렬처리에서 Read / Write Operation의 차이를 알기
* 병렬처리의 힘을 느껴 보기

과제 개요

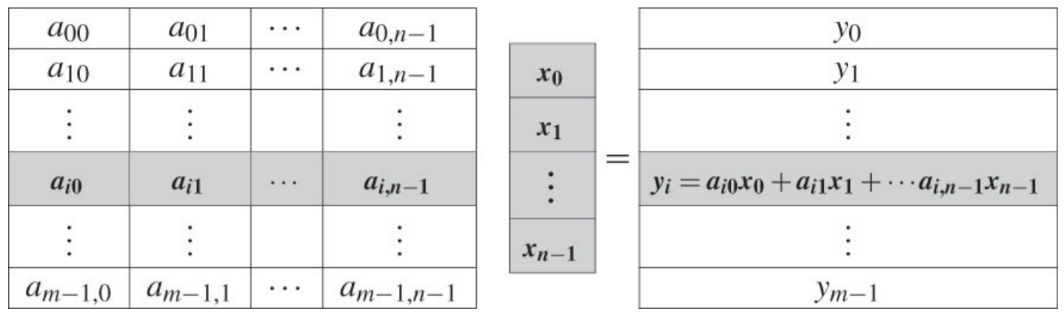
* Input : m, n for Matrix A(m, n), Vector b(n, 1)
* Input에 맞춰 A, b가 자동 생성(genRandomInput => float (){})
* Output : Serial Algorithm과 비교해서 결과 비교(bool isNotCorrect)

알고리즘 구상

* Serial Algorithm

: 단순한 loop를 통해 Matrix / Vector 값 연산

: 2중 loop를 통해 값을 하나하나 구해서 result에 저장



for i in range(0, m), sum = 0 :

for j in range(0, n):

sum = sum + A[i][j] \* b[j]

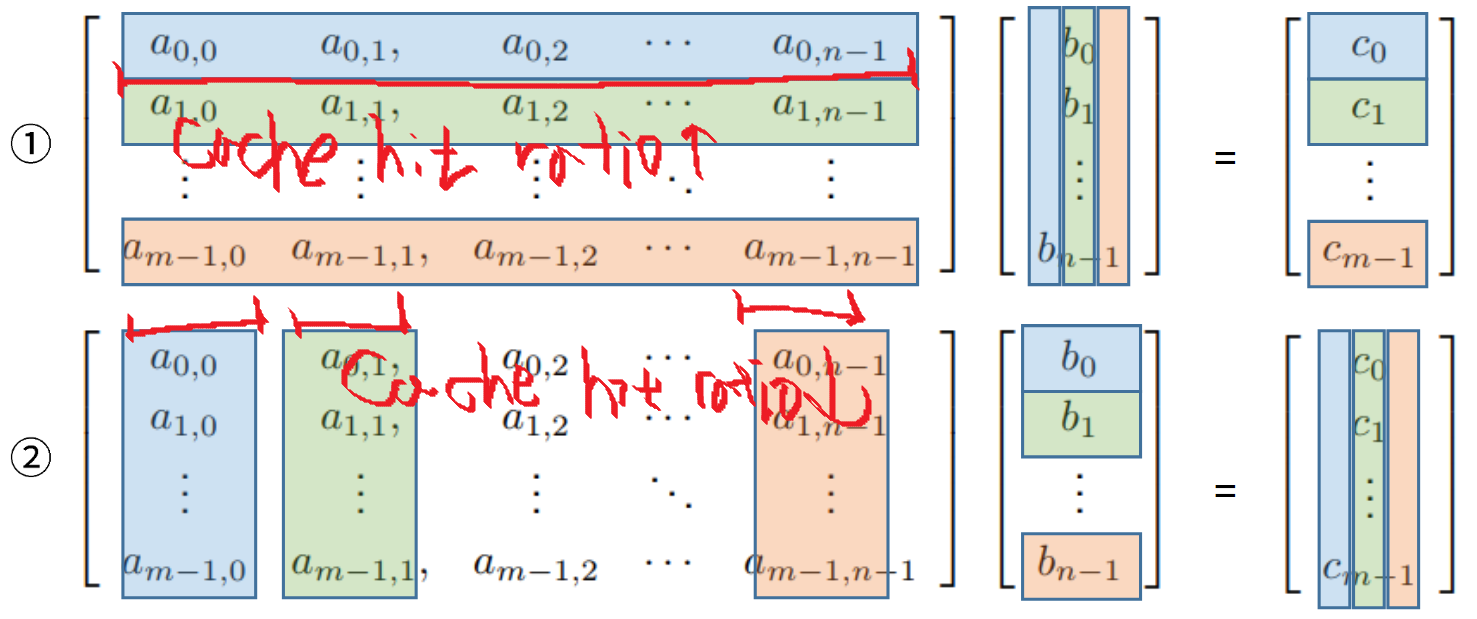
result[i] = sum

* Parallel Algorithm

: 2가지의 알고리즘 구상 가능 – 작업 분배의 방식에 따른 알고리즘

: 행 단위로 작업을 분배하는 경우 / 열 단위로 작업을 분배하는 경우  
 -> 어떤 알고리즘이 더 빠를까?

: 알고리즘 자체는 둘 모두 O(n^2/m)으로 시간복잡도는 동일할 것이다. 다만, 우리가 사용하는 C/C++의 경우, 하나의 일차원 배열을 포인터의 연속된 구간으로 할당하고 이를 캐싱하기 때문에 행 단위의 작업을 할 시 캐시 히트율이 높을 것이다.



: 행 단위의 작업 분배

for i in range(0, m):

for j in range(0, n):

result\_parallel[i] = A[i][j] \* b[j]

: 열 단위의 작업 분배

for j in range(0, n):

for i in range(0, m):

result\_parallel[i] = A[i][j] \* b[j]

구현

* Serial Algorithm

for (int i = 0; i < m; i++)

{

float sum = 0;

for (int j = 0; j < n; j++)

{

sum += A[i][j] \* X[j];

}

Y\_serial[i] = sum;

}

* Parallel Algorithm : Row Wise

#pragma omp for

for (int i = 0; i < m; i++)

{

for (int j = 0; j < n; j++)

{

Y\_parallel[i] += A[i][j] \* X[j];

}

}

* Parallel Algorithm : Col Wise

#pragma omp for

for (int i = 0; i < n; i++)

{

for (int j = 0; j < m; j++)

{

Y\_parallel[j] += A[j][i] \* X[i];

}

}

* Correction Checking Routine

bool isNotCorrect = false;

for (int i = 0; i < m; i++)

{

if (Y\_serial[i] != Y\_parallel[i])

{

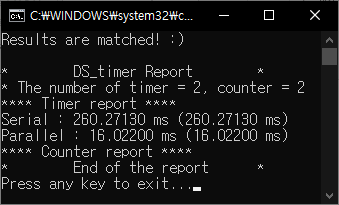
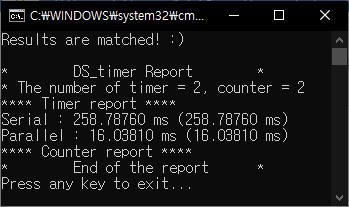
isNotCorrect = true;

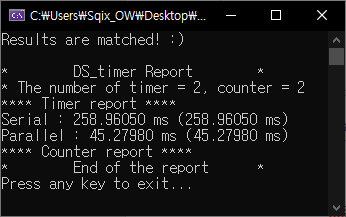
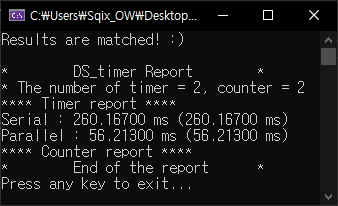
}

}

결과 (24 threads Used, m,n = 10000)

* Parallel Algorithm : Row Wise



*  Parallel Algorithm : Col Wise

앞서 예측했던 대로 Row Wise가 Col Wise보다 더욱 속도가 빠름을 결과를 통해 알 수 있다.

느낀 점

* 두 알고리즘 모두 O(n^2 / m) 형태의 시간복잡도를 띔에도 불구하고 캐시 히트율을 고려하여 작성된 Row-wise의 알고리즘이 2배 이상 빠르다는 점이 놀라웠다.
* 앞선 학년에 배운 운영체제 및 컴퓨터 구조 이론을 활용하여 알고리즘을 작성하여야 더욱 빠른 코드를 작성할 수 있음을 느낄 수 있었던 과제였다.

Lab 2-2. Trapezoidal Rule

과제 : 사다리꼴 법칙을 이용해 그래프상의 특정 구간의 근사값을 구하는 프로그램 작성

과제 개요

* Input : a(start point of x), b(end point of x), n(div num), f(x)=x^2
* Serial Algorithm 구현
* Parallel Algorithm 구현
* a, b, n을 조절해가며 성능 비교 / 분석

알고리즘 구상

* 사다리꼴 법칙 = 윗변 a, 아랫변 b, 높이 h인 사다리꼴의 넓이 S는

S = (a + b) \* h / 2

* 길이가 r인 특정 구간을 n개로 나눈 뒤 각 구간의 넓이를 사다리꼴의 넓이로 근사하고 이의 합을 구하려면
* Serial Algorithm

**:** 단순 for loop를 통해 위 수식을 사용하면 될 것.

* Parallel Algorithm

: 구간별 넓이 합을 구하도록 작업을 분배한 뒤, 병합 과정을 통한 넓이 합 도출

//Parallel Part

#pragma omp for  
 for i in range(0, div\_index):

result\_array += ;

//Merge with serial

for I in range(0, div\_index):

sum += result\_array[i]

구현

* Serial Algorithm

// Serial Algorithm

for (int i = 0; i < div\_index; i++)

{

double x1 = init\_x\_axis + i \* x\_range;

double x2 = init\_x\_axis + (i + 1) \* x\_range;

double area = x\_range \* 0.5 \* (x1 \* x1 + x2 \* x2);

sum\_serial += area;

}

* Parallel Algorithm

// Parallel Algorithm

#pragma omp parallel

{

#pragma omp for

for (int i = 0; i < div\_index; i++)

{

double x1 = init\_x\_axis + i \* x\_range;

double x2 = init\_x\_axis + (i + 1) \* x\_range;

double area = x\_range \* 0.5 \* (x1 \* x1 + x2 \* x2);

result\_arr[i] += area; // bottleneck을 피하기 위해 배열 사용

}

}

//Merge with Serial

for (int i = 0; i < div\_index; i++)

{

sum\_parallel += result\_arr[i];

}

* Checking Code

if (sum\_serial == sum\_parallel) printf("Correct!");

else printf("Not Correct!");

결과

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Task 1 | Task 2 | Task 3 | Task 4 |
| a | 0 | -1024 | -1 | 0 |
| b | 1024 | 1024 | 1 | 1024 |
| n | 1073741824 | 293817231 | 100000000 | 100 |
| Area(Cal) | 357913941.33 | 715827882.66 | 0.6666667 | 357931837.03 |
| Area(Real) | 1073741824/3 | 2147483648/3 | 0.6666667 | 1073741824/3 |
| T(serial) | 3020.79(ms) | 891.8306(ms) | 309.1105(ms) | 0.0005(ms) |
| T(parallel) | 771.7718(ms) | 209.1935(ms) | 75.8105(ms) | 0.9331(ms) |

* 분할 N의 개수가 높을수록 Parallel 알고리즘이 뛰어난 모습을 보여주었다.
* Task 4의 n이 작은 숫자인 경우, overhead가 efficiency를 이기지 못했다.

느낀 점

* 알고리즘을 수식으로 표현하고 코드를 작성하니까 코드 작성이 한결 쉬웠다. 괜히 종이와 펜을 들고 먼저 작업하라고 하신 게 아닌 듯 하다.
* 항상 예제를 통해서 알고리즘 혹은 점화식을 작성할 때 반드시 여러 예제를 통해서 검증해봐야 할 것 같다. Init\_x\_axis 부분을 처음에 넣지 않았더니 0이 아닌 start index를 가진 모든 input에 대한 에러가 발생했었다.(이후에 고침)
* 알고리즘은 간결하지만 동시다발적으로 여러 쓰레드로 분할할 수 있는 인풋이 많은 경우나, 동작 하나가 과도하게 큰 특정 작업에 대해서 병렬처리를 하는 것을 고려해야 할 것 같다. Task 4와 같은 경우 Thread state context를 교환하는 과정에서 overhead가 발생하는데 이 overhead때문에 늦춰지는 시간이 병렬화의 이점보다 더 커서 저렇게 결과가 나온 것 같아 보인다.