# Distributed Programming (CSE5414) Assignment #1

*By 20141500 권 태국*

## 1번 문제) Omega Network

### 1-1)

x = log2(n)

y = n/2

z = x * y = n/2 * log2(n)

Source의 번호가 0 부터 n-1까지라고 하자. Stage 1개를 지나면 source 0은 destination 0~1까지에 접근할 수 있다. 여기서 stage 1개를 더 지나면 source 0은 destination 0~3까지에 접근할 수 있다. 여기서 stage 1개를 더 지나면 source 0은 destination 0~7까지 접근 할 수 있다. 이렇듯 stage 1개를 추가로 지날 때마다 도달할 수 있는 destination의 범위는 2배로 늘어난다. 즉 input이 n개 일 때, 모든 input이 모든 destination에 도달 가능하려면 총 x개의 stage를 지나야 하고, 이 때 n = 2^x 이다. 따라서 x = log2(n) 이다.

2 x 2 switch 한 개에는 2개의 input이 들어 온다. 따라서 각 stage당 총 n개의 input이 들어옴으로 2 x 2 switch는 n/2 개가 필요하다. 따라서 y = n/2 이다.

z = x * y 임으로 n/2 * log2(n) 이다.

### 1-2)

Omega Network에서 routing 방법에 대해 설명해보겠다. 이전 stage의 output이 다음 stage의 input으로 연결될 때 perfect shuffle을 한다. 왼쪽으로 1bit를 shuffle하고 carry bit는 rotate하는 것이다. 예를 들어 1101 -> 1011과 같은 식이다. 그리고 2x2 switch 내에서는 input의 가장 하위 bit를 조작하여 output으로 낼 수 있다. 예를 들면 switch의 input중에 하나가 1101이라고 했을 때 이것은 1100 혹은 1101이 되어 output될 수 있다.

자, 이제 Omega Network에서 routing이 어떻게 가능한지 설명하겠다. 예를 들어 source 가 1011이고, destination이 1001인 경우를 생각해보자. 4bit짜리 input인 경우 stage는 총 4개가 사용된다. 따라서 이는 곧 비트열의 회전이 정확하게 1바퀴하고서 1bit 더 일어난다는 것을 의미한다. 그리고 우리는 매 stage당 가장 하위 비트를 0 혹은 1로 세팅할 수 있다는 점을 알 고 있다. Stage는 총 4개임으로 우리는 routing과정에서 비트

열의 각 비트에 대해서 0혹은 1로 세팅할 기회를 정확하게 1회 가지고 있다는 뜻이다. 즉 이 말은 임의의 source에서 임의의 destination으로의 routing이 가능함을 의미한다.

자, 그러면 어떻게 routing이 구현될 수 있을 지 구체적인 알고리즘에 대해서 예시를 들어 설명해보겠다. 1011에서 1001로의 routing을 고려해보자. 정확하게 bit열의 rotate가 1바퀴하고 1bit 더 일어난다. 따라서 일단 1011을 1회 rotate하여 0111을 만들자. 자 이제 0111에서 1001로 비트열을 변경시키는 문제를 생각해보자. 이 것이 곧 routing의 알고리즘을 의미한다. 이 비트열은 1바퀴 rotate되며 매 과정에서 가장 하위 비트를 0혹은 1로 세팅할 수 있다. 즉 모든 비트 각각에 대해서 조작의 기회를 1회 가지고 있다는 것이다. 0111 -> 1001의 변경이 가능 하려면

4번째 비트를 1로 세팅 (그대로)

3번째 비트를 0으로 세팅 (변경)

2번째 비트를 0으로 세팅 (변경)

1번재 비트를 1로 세팅 (변경)

을 수행해야한다.

자 맨 앞부터 각각 stage 1, stage 2, stage 3, stage 4를 의미한다. Stage 1에서는 straight through를 하고, stage 2,3,4에서는 cross over를 해야한다.

위와 같은 알고리즘을 통해 routing을 구현할 수 있다. Source bit와 destination bit만 알면 아래와 같은 스텝으로 단순하게 routing을 수행할 수 있다.

스텝 1) source bit를 왼쪽으로 1회 rotate (1011 - > 0111)

스텝 2) 스텝 1의 결과와 destination bit를 xor. (0111^1001 = 1110)

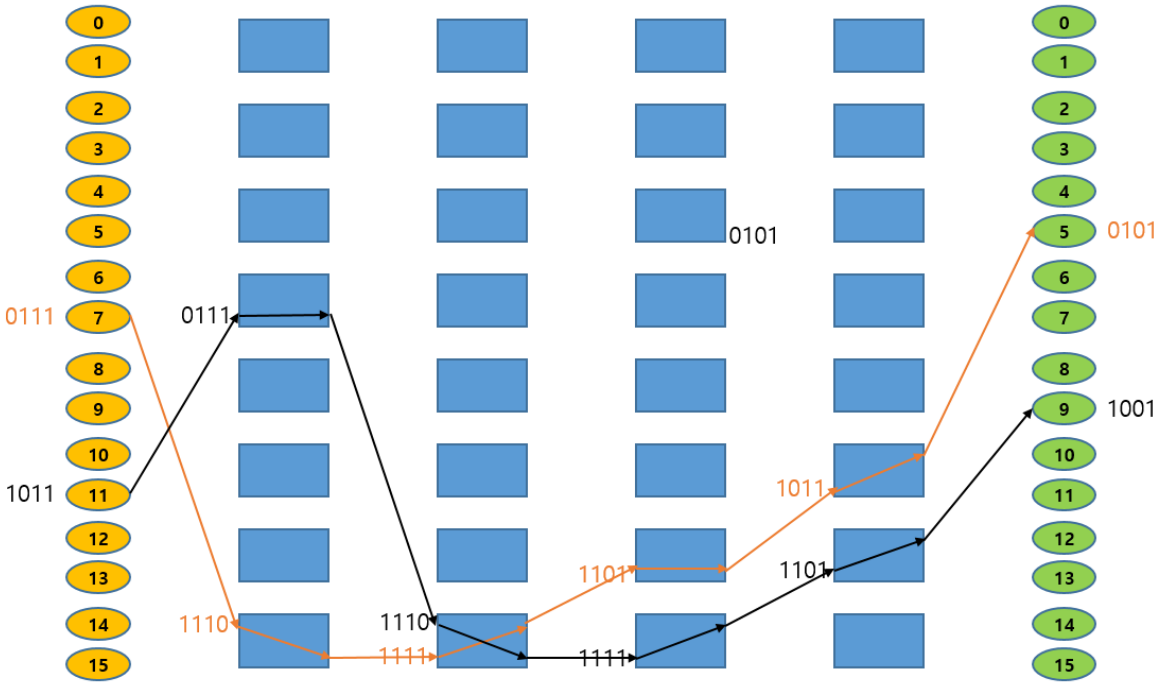
스텝 3) 스텝 2의 결과를 reverse (1110 -> 0111)

스텝 3의 결과 비트열을 맨 앞부터 순회하면 그것이 각 stage에서 해야할 것을 의미한다.

(0은 straight through, 1은 crossover)

## 1-3)

Blocking은 존재하지 않는다.
아래에 흐름을 그려보았다.




## 2번 문제) Hypercube Network

### 2-1)

Hypercube network에서 각 node의 주소는 좌표로서 표현될 수 있다. 예를 들어 3차원인hyper cube에서는 x축,y축,z축이 있다고 할 수 있고 이때 특정 node의 주소를 좌표로 표현하면 (0,1,0)과 같이 표현할 수 있다. 그리고 이를 단순하게 비트열로서 010과 같이 표현할 수 있다.

예를 들어 source node의 address가 010이고, destination node의 address가 100이라고 해보자. 두 address를 xor한다. 010 ^ 100 = 110. 이 것은 source부터 destination까지의 거쳐야하는 경로를 의미한다. 일단 당장 어디로 message를 건네야 하는지 판단하려면 가장 맨 앞에 등장하는 1을 찾아서 그 1에 해당하는 link로 message를 전송하면 된다.

010 -> 110

110 -> 100

과 같이 routing이 이루어질 것이다.

위와 같은 routing mechanism은 일반적으로 모든 hypercube에 대해서 똑같이 적용할 수 있다.

2-2)

101101 -> 011010으로의 메시지가 어떻게 routing 되는 지 설명해보겠다.

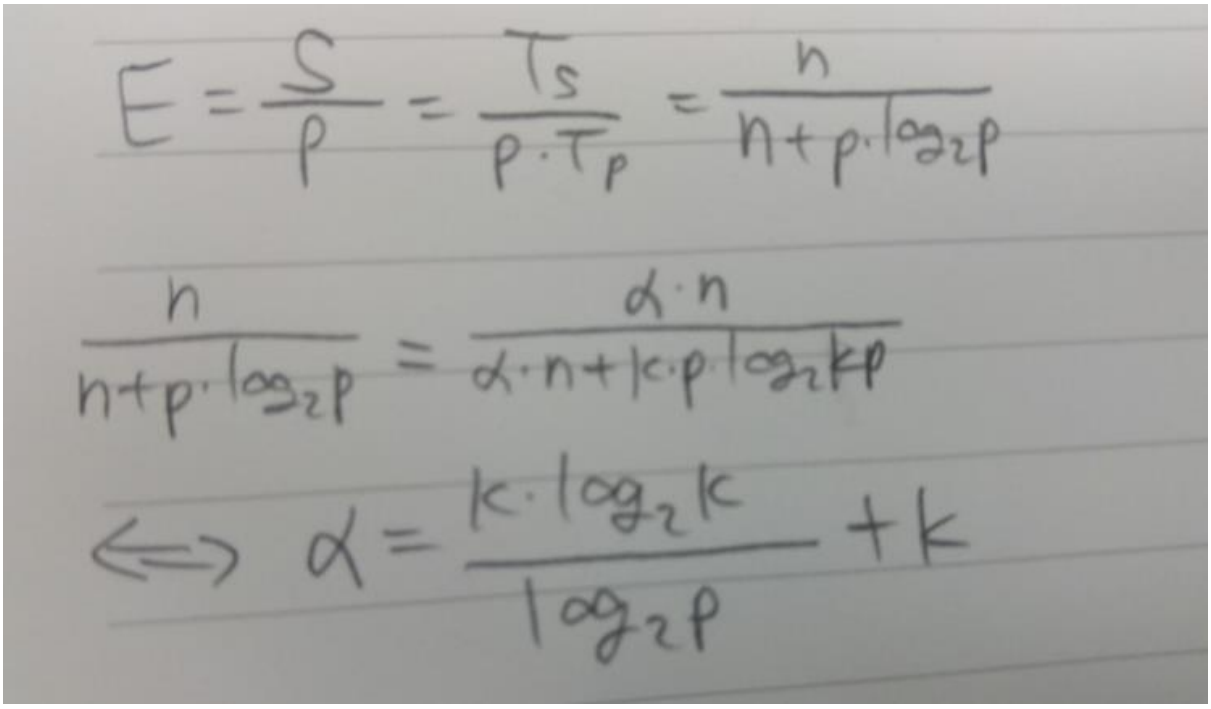
101101 ^ 011010 = 110111이다.

즉 2-1에서 설명한 방법에 따라서 101101 -> 001101로 message가 이동될 것이다. 그리고

001101 -> 011010으로 메시지가 전송되는 것도 위에서 했던 것과 마찬가지로 수행하면 된다.

그러면 총 라우팅 경로는 101101 -> 001101 -> 011101 -> 011001 -> 011011 -> 011010 이다.

## 3번 문제) Scalability




위는 P가 k배로 증가할 때 n은 몇 배로 증가해야 상수 efficiency를 유지할 수 있는 지에 대한 식이다.

Process 개수가 8개에서 16개로 2배 늘어날 때 n을 몇 배 증가시켜야 하는지 구하기 위해서 위 식에 p = 8, k = 2를 대입하면 알파 = 8/3. 즉 n을 8/3배 증가시켜야 한다는 답을 구할 수 있다.

이 프로그램의 경우 process개수를 증가시켰을 때 일정 비율로 problem size인 n을 증가시키면 변하지 않는 efficiency를 얻을 수 있으므로 weak scalable하다고 할 수 있다.

# 4번 문제) MPI - Primitives

## 4-1)

MPI_Scan의 사용법을 익히고 이를 활용하여 prefix sum을 수행하는 코드를 작성해보았다. (mpi_scan.c)

Parallel prefix sum을 수행하는 알고리즘 2가지를 생각해보았다.

1. 단순한 방법. Time은 O(N).

   첫 번째로는 가장 단순한 방법이다. Time 1에 Node 0은 Node 1에게 X0를 전송한다. 그리고 Time 2에 Node 1은 Node 2에게 X0 + X1 을 전송한다.

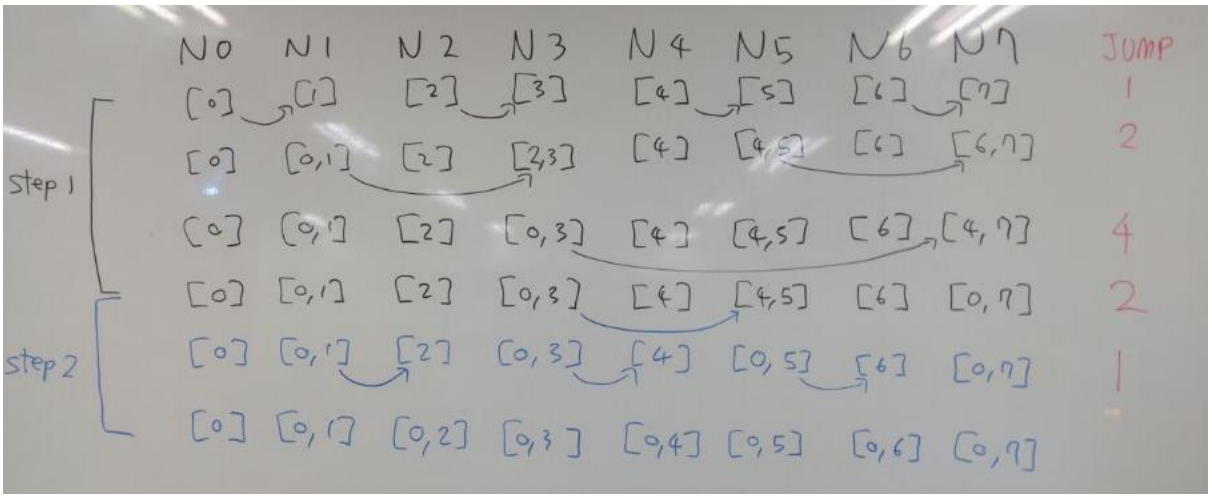
   이런 식으로 반복하면, Time N-1에 Node N-2는 Node N-1에게 X0 + … + X(N-2)를 전송하고, 모든 Node들이 각자의 Partial sum들을 가지게 된다.

   이 방법은 단순하고 구현하기 쉽지만 Time이 N과 비례하여 O(N) 만큼 드므로 비효율적이다.

2. 효율적인 방법. Time은 O(logN).

   두 번째로는 효율적이고, log타임에 prefix sum을 수행할 수 있는 알고리즘이다.

   말로 설명하기가 애매함으로 노드가 8개일 때의 경우에 대해서 예시를 들어설명하겠다.



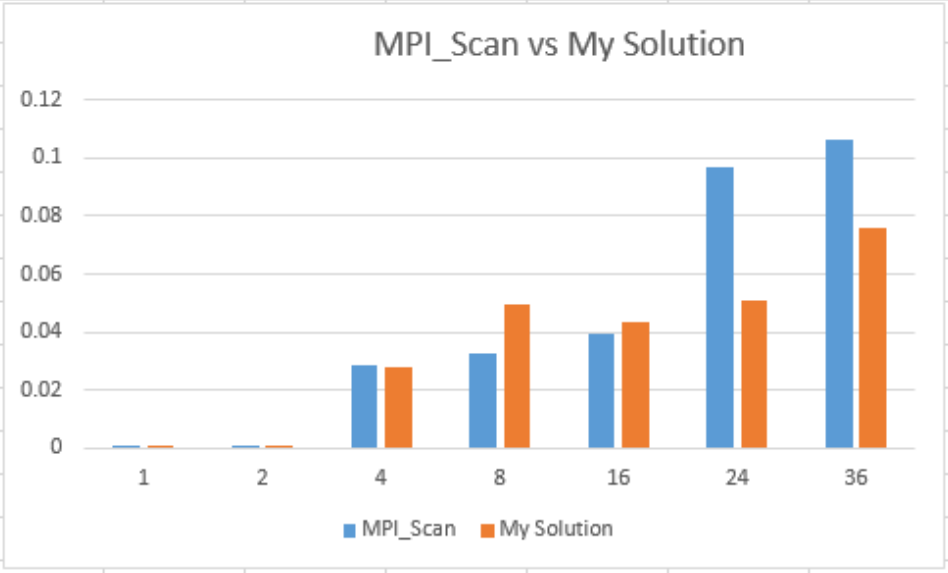

   보면 크게 2가지 Step으로 나눌 수 있다. 1번 째 Step에서는 jump크기가 1에서 2배씩 N/2까지 증가 하며 합을 구하고 있다. 그리고 2번 째 Step에서는 jump크기가 N/4에서 1까지 감소 하며 합을 구하고 있다. 위 그림을 보면 직관적으로 어떤 방법인지 이해가 될 것 이다.

4-2)

나는 4-1에서 2번 알고리즘을 사용하여 prefix sum을 직접 구현하였다. (my_solution.c)

그리고 다양한 N크기에 대해서 MPI_Scan을 사용한 방법과 나의 솔루션을 성능측면에서 비교해 보았다.

| N | 1 | 2 | 4 | 8 | 16 | 24 | 36 |
|---|---|---|---|---|---|---|---|
| MPI_Scan | 0.00054 | 0.000186 | 0.028297 | 0.032294 | 0.039131 | 0.097174 | 0.10622 |
| My Solution | 0.00033 | 0.000187 | 0.027807 | 0.049644 | 0.043652 | 0.050548 | 0.07573 |




N을 1부터 36까지 증가시켜가며 수행시간을 측정해보았다.

일단 눈에 띄는 점 중에 하나가 process 개수가 1,2에서 4가 되면서 수행시간이 급격히 증가한 것이다. 이 이유는 process 개수가 2개 일 때는 모든 process가 같은 컴퓨터내에서 실행이 되고, 또한 process 2개가 전부 각각 cpu core를 할당 받아서 빠르게 실행되지만 process 개수가 4개가 넘어가면 서로 다른 컴퓨터에서 process가 실행되고, context switching등의 비용이 추가로 발생하기 시작하여 급격하게 수행시간이 늘어나는 것으로 추측된다.

그리고 MPI_Scan과 My Solution을 비교해보면, 초반에는 거의 비슷하거나 MPI_Scan이 아주 약간 우세한 그런 경향을 보이지만 N의 크기가 커질수록 점점 더 My Solution이 MPI_Scan에 비해 빨라지는 경향을 띄고 있다.

## 5번 문제) MPI – Simple Example

5-1) p5_1.c에 구현하였다.

5-2) p5_2.c에 구현하였다.

5-3) p5_3.c에 구현하였다.

5-4) 5-1,2,3 중에 하나를 골라서 np를 1로 두고 실행시키면 된다.

5-1,2,3의 성능 비교)

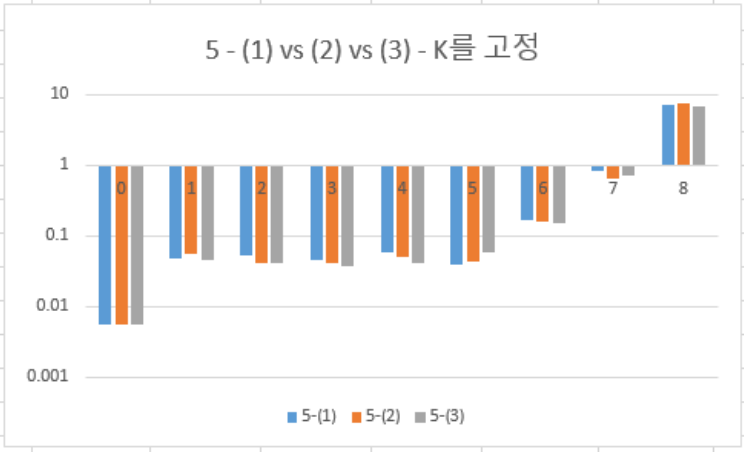
유의미한 비교를 위해서 수행시간의 측정에 있어서 integer들을 파일에서 읽는 부분은 측정에서 제외하였다.

a. K를 고정

overflow

process갯수 = K = 18 100만

| 10^N | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| 5-(1) | 0.005635 | 0.049255 | 0.05297 | 0.044986 | 0.058962 | 0.039205 | 0.168342 | 0.848672 | 7.063216 |
| 5-(2) | 0.005543 | 0.054626 | 0.041875 | 0.041657 | 0.050347 | 0.042828 | 0.158409 | 0.657589 | 7.733846 |
| 5-(3) | 0.005608 | 0.046288 | 0.040845 | 0.037261 | 0.041147 | 0.058534 | 0.150492 | 0.734925 | 6.77667 |

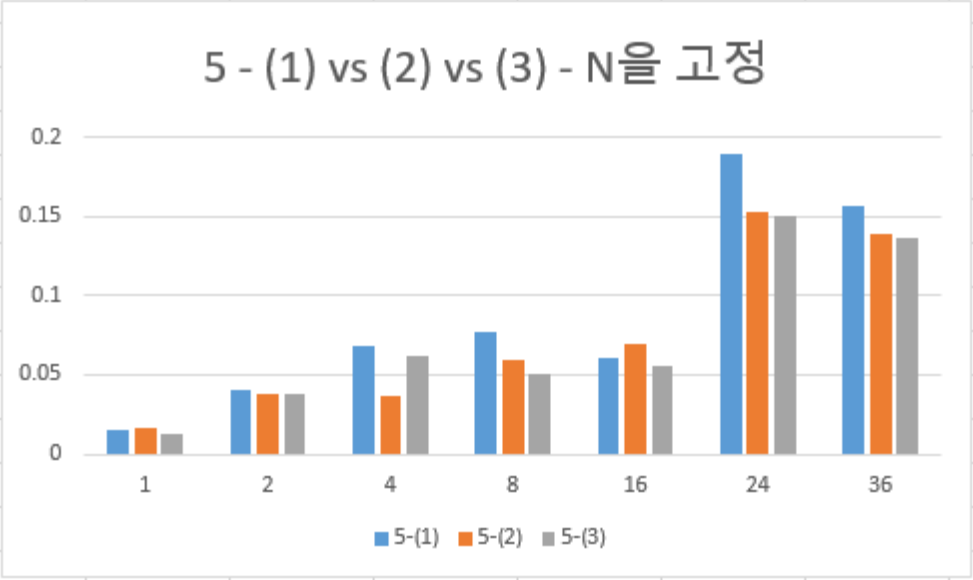



K = 18으로 고정해두고 실험해보았다. 위에서 볼 수 있듯 (1), (2), (3)은 서로 비슷한 성능을 가지고 있음을 알 수 있다. 그리고 N이 너무 작으면 수행 시간에 끼치는 영향이 매우 적다. 따라서 N이 100만 가까이는 되야 전체 수행시간에 유의미한 영향을 미친다. 그리고 N = 1일때와 10일 때 차이가 많이 나는 이유는 N=1이면, process 18개중 17개는 아무것도 하지않고, 오직 master 하나만 1개의 숫자를 처리하기 때문에 수행시간이 극도로 작게 나오는 것이다.

b. N을 고정

N=500000

| K | 1 | 2 | 4 | 8 | 16 | 24 | 36 |
|---|---|---|---|---|---|---|---|
| 5-(1) | 0.015314 | 0.0409 | 0.068296 | 0.076535 | 0.060101 | 0.18984 | 0.15712 |
| 5-(2) | 0.016525 | 0.03772 | 0.036887 | 0.05882 | 0.069416 | 0.152454 | 0.13923 |
| 5-(3) | 0.012579 | 0.038528 | 0.061503 | 0.050906 | 0.055085 | 0.150622 | 0.135834 |




N을 50만으로 고정하고 K를 변화시키며 실험을 진행하였다. 역시 크게 유의미한 차이를 관찰 할 수 없었다. 주목할 점은 K=16에서 24로 커지면서 수행시간이 2배 이상 커지는 것이다. 실험 환경이 컴퓨터 9대, 컴퓨터당 물리적 코어 2개 (논리적 코어 4개)이다. 이 때, process 개수가 16에서 24로 커지면, 컴퓨터당 2개가 넘는 process들이 할당되면서 context switching와 network 통신 등의 overhead가 커지고 분산 효과는 미미해지는 결과로 생각된다.

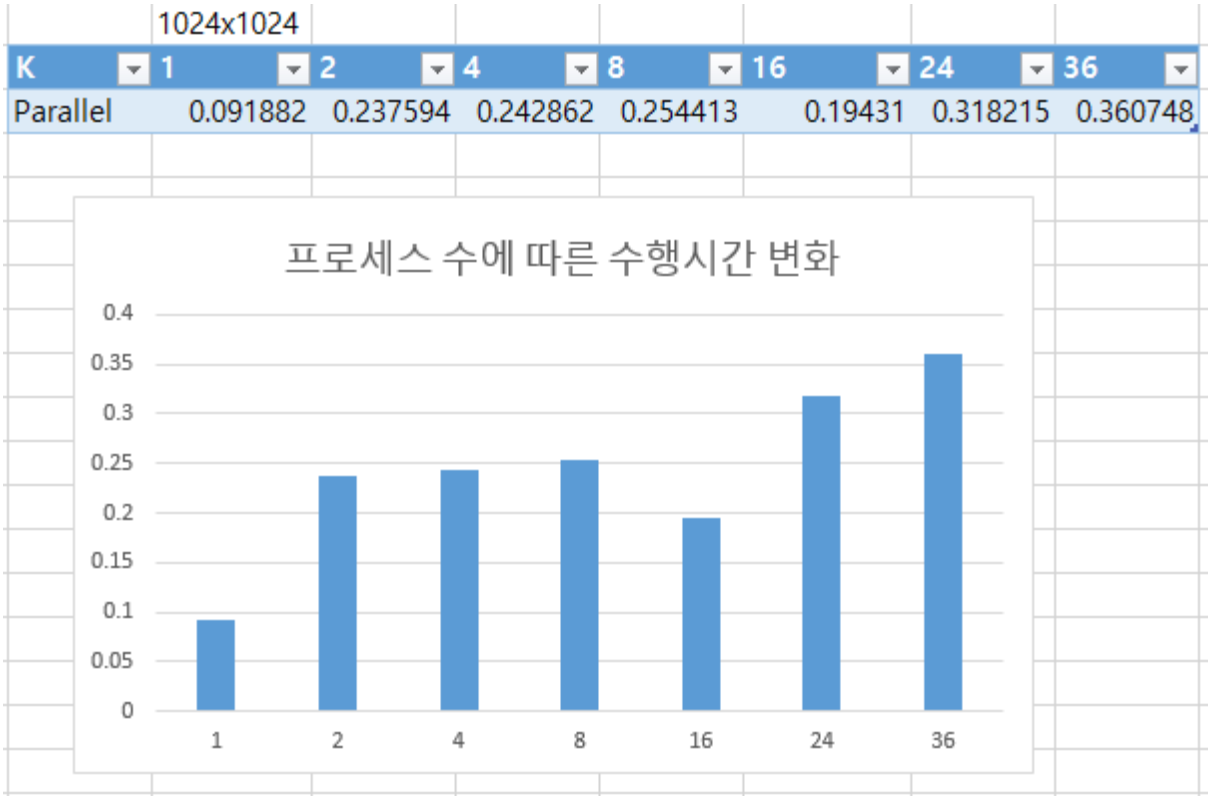
그리고 위 결과를 보면 K=1일 경우 (요구사항 5-(4)에 해당)가 가장 성능이 좋은 것을 알 수 있다. 그 이유는 N이 충분히 크지 못하기 때문이다. 5번 문제의 경우 N 1 당 처리해야하는 연산양이 매우 적다. 즉, 분산 효과를 보기 어려운 경우이다. 만약 이론적으로 생각한다면 N이 정말로 충분하게 크다면 분산을 하는 경우가 훨씬 성능이 좋을 것이다. 그러나 이 것은 실제로 구현하기 쉽지 않은데 그 이유는 N이 극도로 클 경우 메모리 할당 문제등의 이슈가 있기 때문이다.

## 5-1,2,3의 안정성 비교)

결론적으로는 (3) > (2) > (1) 순으로 안정적이다. 왜냐하면 일단 (1)의 경우 각각의

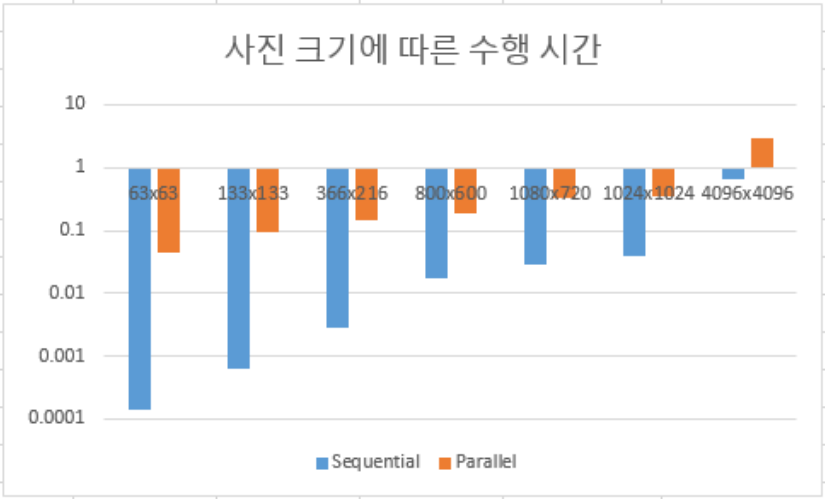
process가 local sum을 구한 후 이를 master가 total한다. 따라서 이는 overflow가 발생할 가능성이 매우 크다. 따라서 안정성이 가장 떨어진다. 그리고 (2)의 경우에는 collapse를 구한 후 master로 보내기 때문에 (1)에 비해 overflow가 발생할 가능성이 현저히 작다. 마지막으로 (3)의 경우에는 ultimate collapse를 구한 후 master를 보냄으로 (2)보다 더 안전하다고 할 수 있다.

## 6번 문제) MPI – Image Processing




위는 프로세스 수에 따른 수행시간의 변화이다. (1024 x 1024 크기의 ppm을 바탕으로 실험을 수행하였다) 5번에서와 같이 프로세스 수가 가장 적은 1일 때 최고 성능을 발휘하는 데 이는 6번 문제 같은 경우에는 분산 처리를 하는 것이 옳지 않음을 의미한다. 왜냐하면 애초에 처리 속도가 매우 따른 작업들을 굳이 분산해서 하는 것은 오히려 overhead만 커지고 효율적이지 않는 방법이기 때문이다.

| K=36 | | | | | | | |
|---|---|---|---|---|---|---|---|
| | boxes_1 | tree_1 | west_1 | cayuga_1 | falls_1 | Iggy.1024 | X4 |
| Size | 63x63 | 133x133 | 366x216 | 800x600 | 1080x720 | 1024x1024 | 4096x4096 |
| Sequentia | 0.000142 | 0.000642 | 0.002853 | 0.017761 | 0.028672 | 0.038665 | 0.659706 |
| Parallel | 0.043516 | 0.091967 | 0.145134 | 0.187319 | 0.32941 | 0.357924 | 2.877206 |




위는 PPM파일의 크기에 따른 수행 시간을 나타낸 것이다. 역시, Sequential이 가장 빠르다. 아까 에도 말했다시피 6번 문제의 경우에는 분산처리하지 않는 것이 가장 효율적인 방법이다. 예를 들어 사진이 1024 * 1024 일 경우에 (N = 1024, M = 1024) sequential 프로그램의 시간 복잡도는 O(N*M*pixel당 처리 복잡도) 이다. 근데 단순한 grayscale과 flip만 수행함으로 pixel 당 처리 시간 복잡도는 1이다. 따라서 O(N*M)이 된다. 즉 linear time이라고 할 수 있다. 위 결과에서도 볼 수 있듯이 1024x1024 와 4096x4096의 수행시간차이는 약 64배. 사진의 크기도 64배이다. 즉, sequential하게 처리하면 매우 빠르게 처리 가능하다. 근데 이것 작업을 분산처리하면 위에서 보다시피 매우 긴 시간이 걸린다. 분산처리에 따른 overhead가 크기 때문이다. 이 문제의 경우에는 overhead를 감내할 만큼의 처리양이 많은 작업이 없기 때문에 분산처리를 사용하지 않는 편이 옳다. 따라서 scalability가 전혀 없다고 말해야 할 것이다.