**Distributed Programming Assignment #1**

20141549 우상연

1. **Omega Network**

1-1

**X = log2(n)**

**Y = n/2**

**Z = n/2 \* log2(n)**

Source가 0 부터 n-1 까지 n 개라고 하면, stage를 1개 지나면 destination 0 ~ 1,

2개를 지나면 0~3 즉, x개의 stage를 지나면 2^x개의 destination에 도달할 수 있다. 따라서 x = **log2(n)이다.**

2x2 switch 한개에는 두개의 input이 들어오는데 stage당 n개의 input이 들어오므로 switch는 n/2개가 필요하다. 따라서 **Y = n/2**

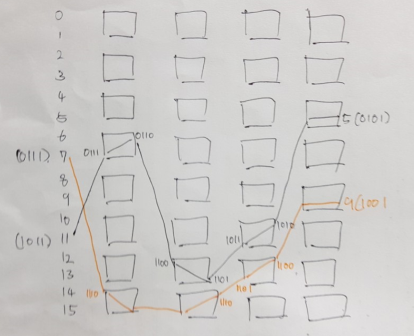
**1 – 2**

Omega Network에서 routing 방법에 대해 설명하겠다. 출발 input 과 원하는 output을 xor연산을 하면 각 스위치에서 cross 될지 아니면 straight할지 결정된다. 예시를 들면

Input : 1001 ,output : 1110 이라하면 둘의 XOR 연산 결과는 0111이다. N이 16이므로 Stage는 4개가 되고 각 스위치에서 처음에는 0 이므로 straight 나머지는 1이므로 cross되게 된다. 각 stage 에서는 input이 left shift되어 shuffle된다. 이런식으로 원하는 output으로 조작할 수 있다.

**1 – 3**

**Blocking 이 존재하지않는다. 흐름도를 그려보았다.**



**2 .**

**2 - 1**

Hypercube network에서 각 node의 주소는 좌표로서 표현될 수 있다. 예를 들어 3차원 hyper cube에서는 x, y, z축이 있다고 할 수 있고, 이때 특정 node의 주소를 좌표로 표현하면 (0,1,0)과 같이 표현할 수 있다. 이를 단순하게 비트열로서 010과 같이 표현할 수 있다. 예를 들어 source node의 address가 010이고, destination node의 address가 100이라고해보자. 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. 2-1에서 설명한 방법처럼 101101 -> 001101로 message가 이동될 것이다. 그리고 001101 -> 011010으로 메시지가 전송되는 것도 위와같이 수행하면 된다.

따라서 총 라우팅 경로는 101101 -> 001101 -> 011101 -> 011001 -> 011011 -> 011010 이된다.

**3 .**

**이 문제는 scalability 를 유지시키는 데 관한 문제이다.**

E = S / P , E = n / ( n+ plog2(p)) 인데 여기서 p가 8 -> 16 이되었으므로, plog2(p)부분이 2\*4/3만큼 증가하였다. 따라서 n이 8/3만큼 증가해야한다. 이 프로그램의 경우 process개수를 증가시켰을 때 일정 비율로 problem size인 n을 증가시키면 변하지 않는 efficiency를 얻을 수 있으므로 weak scalable하다고 할 수 있다.

**4.**

**4 - 1**

MPI\_Scan의 사용법을 익히고 이를 활용하여 prefix sum을 수행하는 코드를 작성해보았다.

Parallel prefix sum을 수행하는 알고리즘 2가지를 생각해보았다.

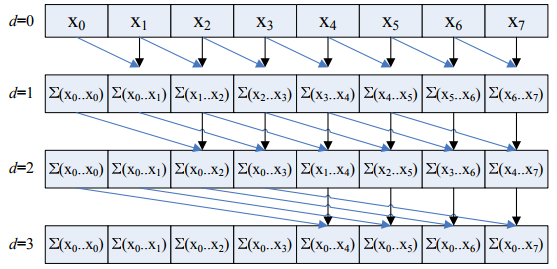
1. 단순한 방법. Time은 O(N).

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.

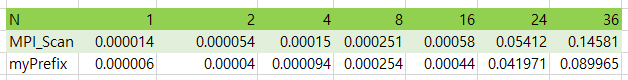
1. 효율적인 방법 Time은 log(N)

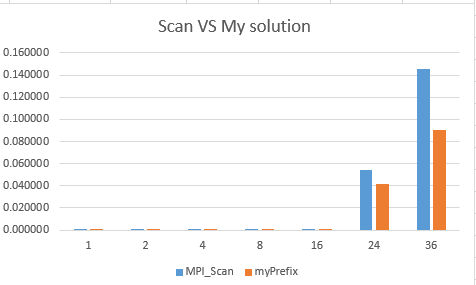


위 그림과 같은 방법으로 노드가 8개일 경우에 수행한 과정이다. 각 스탭에서 prefix sum이 진행되는 jump 수가 2배씩 증가하며 수행하게 되므로 결국 logN의 time안에 끝낼수 있게된다.

4 – 2

4 - 1에서 구한 mpi\_scan.c의 결과와 내가 제안한 알고리즘을 이용한 결과의 시간을 분석해 보았다. 각 processor를 다양하게 증가시키며 수행해본 결과 대체로 my solution이 시간은 더 적게 걸렸다. 그러나 시간이 N이 16보다 큰 경우에는 급격히 증가하였는데, 이는 N이 늘어나면 또 다른 컴퓨터에서 수행되며 context switching 같은 비용이 발생하게 된 것으로 보인다. 또 한 mpi\_scan에 비해 logN배 빠르지 않은 점 또한 위에서 말한 비용에 의한것으로 보인다. 그러나 점차 노드가 증가할수록 제시한 방법이 효율을 잘 내고있음을 볼 수 있다.





**5. parallel한 quick sort 알고리즘에 대하여 설명하겠다.**

**Step 1. 각 데이터를 프로세스의 수로 나누어 각 프로세스로 보낸다.**

**Step 2. 첫번째 프로세스에서 임의의 value로 pivot을 선택한다. (이때 이 pivot을 median value로 택하면 특정 케이스에 대해불필요한 연산을 줄일 수 있다.)**

**Step 3. 프로세스 끼리 짝을 묶고 pivot 보다 작은값을 앞 으로, 큰 값을 뒤로 모은다.(여기서 앞 뒤는 프로세스의 짝을 말하는데 custom하게 설정해도 될것 같다.)**

**Step 4. 작은쪽과 큰쪽에서 또다시 pivot value를 구하고 이를 다른 process에 알려준다.**

**Step 5. 이런식으로 방법을 반복하면 각 프로세스 별로 결과가 정렬된다.**

**Step 6. 정렬된 결과를 다시 하나로 합치면 완료된다.**

**P5/par\_qsort.c에 구현하였다.**

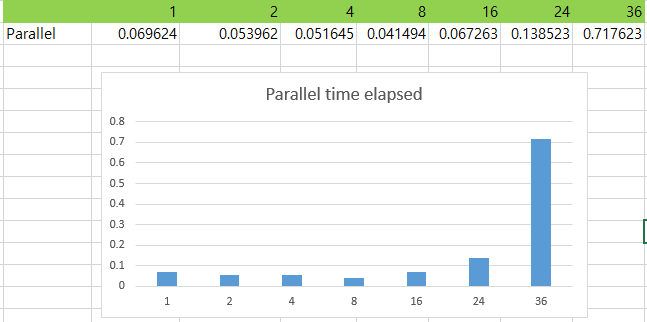
**5 -2**

**이 알고리즘이 나는 NlogN/2의 시간복잡도를 가진다고 생각한다. 프로세스를 여럿으로 나누지만 결국 pivot을 기준으로 두 곳으로 나누어지기 때문이다. 따라서 프로세스가 2이상일때 scale 되기시작한다.**

**5 -3**

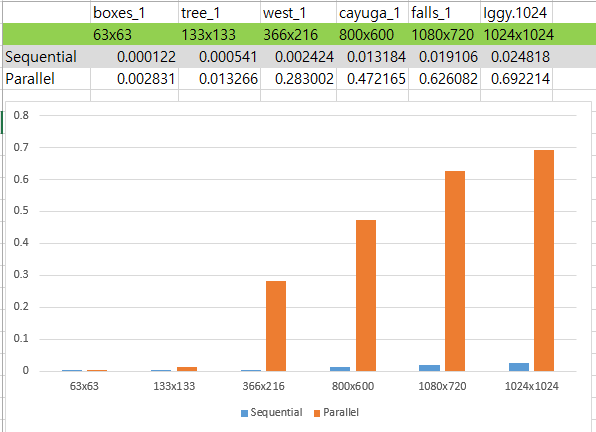
**예상보다 speedup이 느린이유는 일단 pivot에 대하여 나누어지는 결과들을 계속 process들에게 send되고 receive 되는것이 기대되는 시간보다 더 걸릴뿐더러, 이를 여러 컴퓨터에서 실행하게되면 single thread의 퀵소트보다 느려지게 될 수도 있다.**

**6 . Image Processing**



**위는 프로세스의 수에 따른 수행시간의 변화이다. 1024x1024 ppm파일로 수행하였다.**

**프로세스의 수가 8일때 성능이 가장 좋게 나왔지만 1,2,4,8 의 시간이 거의 비슷한정도이고 오차가 어느정도 있는 편이므로 거의 없다고 보는게 무방할 것이다. 이는 Pixel을 프로세스의 수만큼 나누어 분할하여 flip and greyscale을 처리한 후 다시 merge하는 방식을 사용하였는데 프로세스가 16개 이상으로 많아질 경우에는 이미 처리속도가 빠른 작업을들 분산처리했기때문에 오히려 각 프로세스로 나누고 merge하는데에 overhead가 더 커졌기 때문이라고 볼 수 있다.**



**다음은 ppm파일의 크기별로 sequential 과 parallel하게 구현한 결과에 대한 비교이다. Parallel한 경우 노드의 갯수는 36개를 사용하였다. 대체적으로 sequential한 방법이 월등하게 빨랐으며, 이는 이 문제와 같은 경우 분산처리가 매우 비효율 적임을 말해준다. 위에서 말했듯이 프로세스의 수가 많아져봤자 어차피 flip 과 greyscale은 연산속도가 매우 빠른 linear time 이기때문에 오히려 분산처리하는데에 걸리는 overhead가 더욱 크게 된다. 따라서 이같은경우에는 scalability가 전혀 없다고 할 수 있다.**