**parallel / distributed computing**

**assignment #1**

**20141602**

**황기덕**

1. 1. E cube routing routes a message from source to destination by traversing the differing dimensions in right - to - left order.

N = 2^m. N이 노드의 개수, m 이 dimension 이면. src, dest 모두 m개의 bit를 가진다. 이 때 src 와 dest 를 xor를 한 결과를 가지고, 그 결과가 1bit면 src를 한 비트씩 바꾸고, 0이면 바꾸지 않는다.

* 1. 64 node -> 64 = 2^6. 6 dimensions.

src = 101101 dest = 011010

src XOR dest = 110111

1. 101101 -> 101100
2. 101100 -> 101110
3. 101110 -> 101010
4. skip
5. 101010 -> 111010
6. 111010 -> 011010

5%의 non parallelizable part 와 95%의 parallelizable part가 있다.

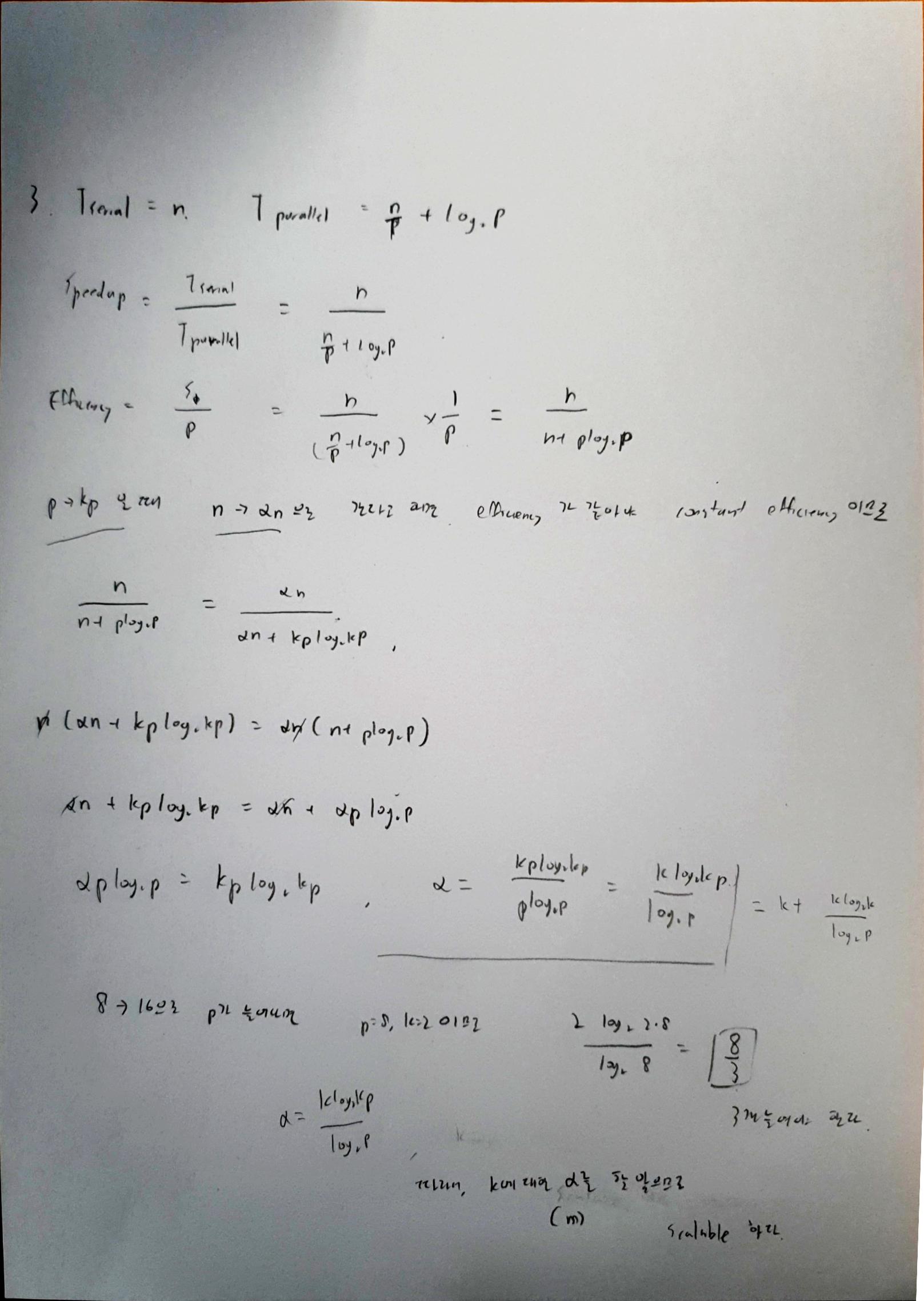
X-> 4cores, 1hour.

Y-> 16cores, 2hours.

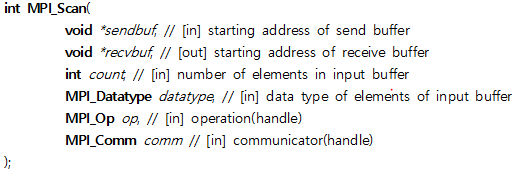
X-> 5/100 \* 1H + 95/100 \* 1H / 4 = 0.2875H

Y-> 5/100 \* 2H + 95/100 \* 2H / 16 = 0.21875H

이므로 Y를 사야한다.



**4 - 1)**



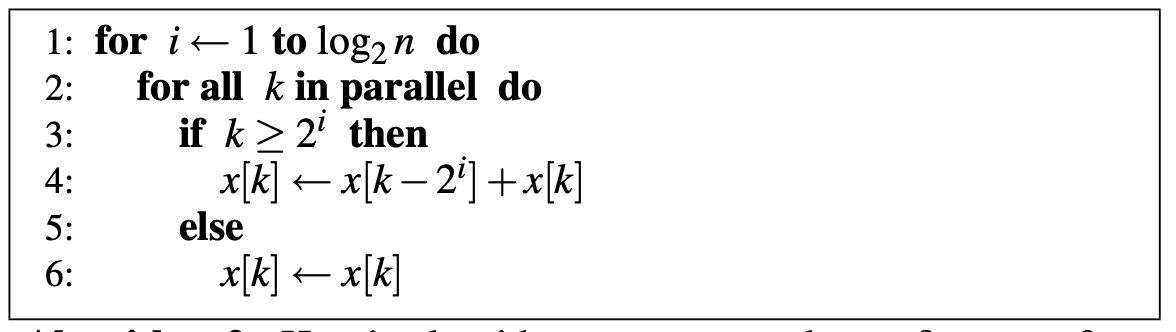
어떠한 데이터에 대해 그 전 processor로부터 받아 연산을 한 후 저장을 하고, 다음 processor에게 넘겨준다.

1. sequential implementation



가장 기본적인 알고리즘이다. x에 배열이 들어있고, 하나하나 돌면서 이전까지의 합의 값을 다음 값에 더해주는 식으로 prefix sum을 구해준다.

2. data-parallel algorithm

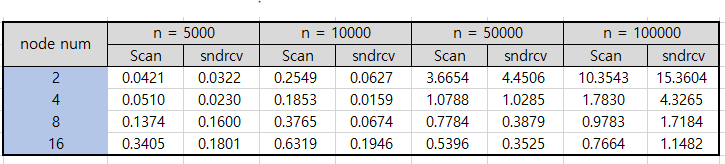


data parallel algorithm이다. 각각의 원소들이 parallel하게 계산을 하며, 하나하나 계산하지 않고 앞에서 더한 값을 그대로 가져와서 본인에게 더한다.

n = element 개수, k-> 각각의 원소들. x -> 배열

4- 2)

위 코드와 같이,



왼쪽 파란색은 node의 개수, 오른쪽 위의 회색에 n은 prefix sum 을 구할 때 배열의 길이, 밑의 scan 과 sndrcv는 각각 mpi\_scan을 사용했을 때와 send 와 recieve만을 사용하여 나온 결과이다.

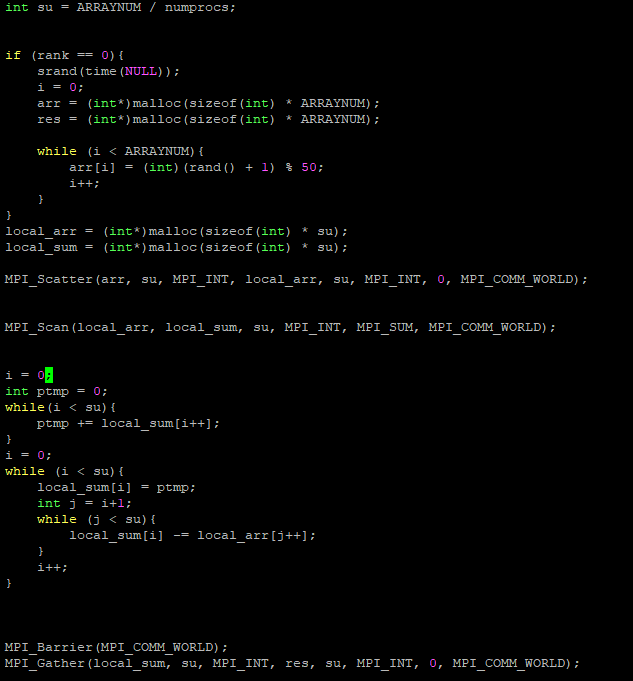
단위는 sec 이다.

n이 작을 때를 보면 node number가 작을수록 더 빠르다는 사실을 볼 수 있다. 이는 mpi를 통해 msg를 주고 받을 때, 정렬하고 더하는 overhead보다 send, recieve를 할 때 생기는 overhead가 더 커서 node number가 많아질수록 더 느려졌을거라는 사실을 추측할 수 있다.

n이 엄청 커진 n = 100000에서 보면 차이가 확연하게 드러난다. node number가 2일 때에는 초 단위로 걸리던 것들이 node number가 증가할수록 걸리는 시간이 확연하게 줄어든다. 이것은 정렬하고 prefix sum을 구하는 overhead가 snd/ rcv의 overhead보다 확연하게 커서 node를 많이 사용할 수록 시간이 줄어들었다는 결과를 확인할 수 있다.

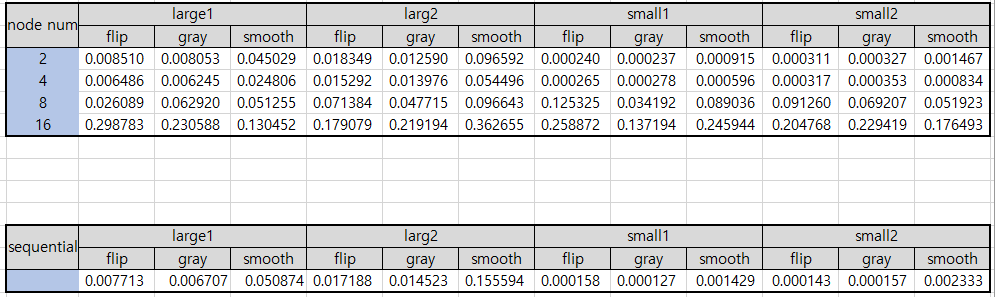
또, scan을 사용하였을 때가 snd/rcv로 했을 때가 결과가 더 좋았는데, 이는 scatter, gather, scan등의 만들어져있는 것을 쓰는 것이 사용자가 직접 snd/ rcv를 통해 handling 하는 것보다 효율이 더 좋다는 것을 의미한다.

-



다음과 같이, dynamic 하게 array의 원소의 수와 node의 수를 받아서 한 process에 들어갈 배열의 개수를 정해주고, 나머지는 직접 handling 해 주었으니 이 program은 number of processor의 개수에 independent하다.

**5.**



각각 실행한 결과 시간이다. 단위는 sec.

위 표가 parallel 하게 수행시켰을 때이고, 밑 표가 sequential 하게 실행했을때의 표이다.

gray, flip, smooth 세 가지를 비교할 경우엔 gray와 flip 이 비슷했고, smooth 연산이 항상 시간이 오래 걸렸다. 이는 단순 세개를 더해서 나누는 연산을 하는 gray와 두 위치를 swap하는 flip연산보다는 주변 8개의 pixel을 모두 모아서 더해 나누어주는 smooth연산이 더 오래 걸렸으므로 그랬을 것이다.

먼저 단순 sequential 과 parallel을 비교해 봤을 때에 sequential이 대체적으로 빨랐지만, smooth 와 node num이 2일 경우에는 parallel이 더 빨랐다. 이는 smooth 연산이 연산이 많아 연산 overhead보다 서로 msg를 주고받는 overhead가 더 적었기 때문이었을 것이다.

node num에 비교해서는 대체적으로 node num이 커질수록 msg 통신하는 overhead가 더 컸기 때문에 개수가 많아질수록 시간이 증가했다.

대체적으로 node 4일때까 가장 효율이 좋았는데, 이는 연산하는 overhead와 msg overhead가 존재하니 그 둘을 잘 계산해서 node를 결정해야 최고의 효율을 뽑아낼 수 있을 것이다.

img가 large일 때가 small 일 때보다 효율과 speedup이 더 좋았다. 이는 large일 때가 small일 때보다 연산할 것이 더많으므로 나온 결과일 것이다.

이 결과 자체에서는 scaliblity가 지켜지지 않았다. 아무래도 msg overhead가 크다 보니 연산의 양이 줄어드는 것이 그렇게 많이 느껴지지 않고, 오히려 msg overhead가 더 critical하게 작용해 시간이 오히려 더 늘어났다.

구현 방법:

1. flip

* ppm 파일을 읽어 들인 후 한줄 단위로 i번째 pixel과 width - i 번째 pixel을 서로 바꿔주었다.
* parallel을 사용할 때는 총 height 줄의 pixel이 있는데, 이것을 numprocs개로 분할하여서 data를 나누어 주어 연산하였다.

1. gray

* ppm 파일을 읽어 들인 후 한 픽셀마다 r,g, b,값을 모두 더해 3으로 나눠주었다.
* parallel을 사용할 때는 총 height 줄의 pixel을 numprocs 개로 분할하여 data를 나누어 주어서 똑같이 연산해 주었다.

1. smoothing

* ppm 파일을 읽어 들인 후 한 픽셀마다 그 주변의 픽셀의 값을 다 구해서 더해 준 다음 8로 나누어 주었다.
* parallel을 사용할 때는 height줄의 pixel에 대해 numprocs개로 분할하여 data를 나누어 주어 똑같이 연산해 주었다.