D P - #1

학 번: 20111599

작성자: 김 세 영

1. In Omega network,

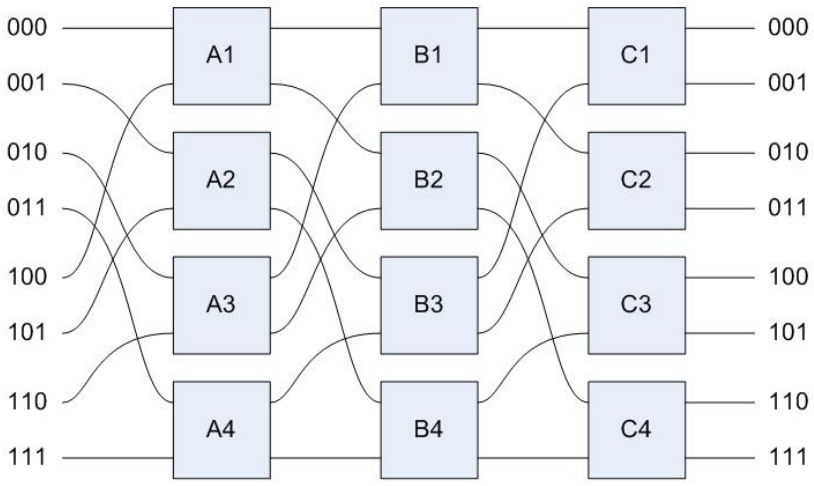
(1) In general, an n-input Omega network requires x stages of 2 x 2 switches. Each stage requires y switch modules. In total, the network uses z switches. Get the values of x, y, and z and discuss why.

(2) Describe how routing can be implemented in Omega network.

(3) Show the switch settings for routing a message from node (1011) to node (0101) and from node (0111) to node (1001) simultaneously. Does blocking exist in this case ?

(1)

Omega Network는 multistage interconnection network로 이는 processing element(PE)들이 스위치들이 배치되어 있는 여러 개의 stage들을 통해 연결을 구성한다. 각 Input과 Output엔 주소가 지정되어 있고, 각 stage의 output들은 perfect shuffle을 통해 다음 stage의 input들로 연결된다. 각 PE가 이진 수로 표시된다면 Perfect shuffle의 매 stage는 one bit cyclic logical left shift가 이루어진다고 생각하면 된다. 예를 들어, 8x8 Omega Network를 먼저 생각해보면 다음과 같은 그림이다.



NxN Omega Network를 구성하려면 N개의 input/output processing element들이 필요하며, input/output 별로 각 PE를 표현하기 위해 각 PE는 n-bit 이진수로 표현된다. Omega Network는 input과 output PE들간의 연결을 위해 필요한 것이기 때문에 만약 2x2 switch를 사용하고 perfect shuffle을 사용한다면, PE와 스위치간의 연결 혹은 두 스위치간의 연결을 구성하려면 각 stage별로 N/2개의 스위치가 구성되어야 한다. 예를 들어, N개의 input PE들에 대해 perfect shuffle을 수행하면 이들에게 연결될 N개의 input이 필요하며 input 개수가 2개인 2x2 스위치를 N/2개만큼 사용하여 연결을 구성할 수 있다. Perfect shuffle은 기본적으로 매 stage마다 one bit cyclic logical left shift를 수행하여 다음 단계의 input들로 연결되기 때문에 PE를 표현하는 N-bit의 수가 perfect shuffle을 통해 원래 자기 자신의 수로 되돌아오려면 N번의 shift가 이루어져야 한다. 따라서, N개의 stage가 필요하다.

결론적으로, Omega Network는 각 stage 별로 N/2개의 스위치들을 갖게 되고, log2(N)개의 stage들을 갖게 된다. 따라서, 총 N/2 * log2(N) 개의 스위치들이 필요하다.

(2)

Switch들이 어떻게 설정되느냐에 따라서 주어진 시간에 네트워크에서 어떤 connection path가 만들어질지 결정된다. 이러한 방법들로 두 가지가 있는데, 하나는 **destination-routing**이고 다른 하나는 **XOR-tag routing**이다. Omega Network는 blocking될 가능성이 크다. 그럼에도 불구하고 free network 상황에서는 어느 하나의 input에서 다른 하나의 output으로 반드시 경로가 만들어 질 수 있다.

Destination routing

스위치 세팅이 전적으로 message destination 값에 따라 변하게 된다. Destination address의 MSB가 첫 stage의 스위치에 대한 output을 결정한다. 만약에 MSB가 0이라면 upper output이 선택되고, MSB가 1이라면 lower output이 선택된다. 이 후, 다음 MSB가 다음 stage의 스위치에 대한 output을 결정하고, 마지막 output이 선택될 때까지 진행된다.

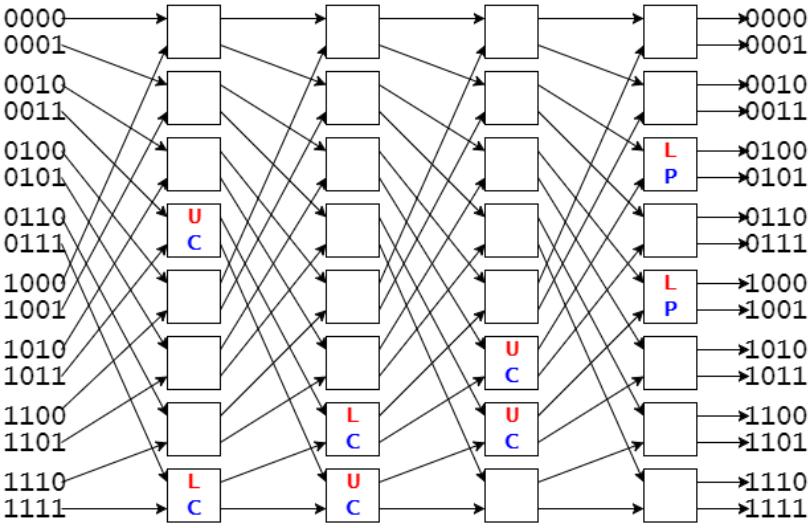
XOR-tag routing

스위치 세팅이 (source PE)^(destination PE) 값에 따라 결정된다. XOR-tag의 MSB가 첫 stage의 switch에 대한 세팅을 한다. 만약 MSB가 0이면, switch가 pass-through로 설정되고, 1이면 crossed되게 설정된다. 다음 MSB가 다음 stage의 스위치에 대한 설정을 하게 되고, 마지막 output이 선택될 때까지 진행된다.

예를 들어, PE 001이 PE 010에게 메시지를 보내고 싶어한다. XOR-tag는 011이므로 적합한 스위치 세팅은 A2 straight, B3 crossed, C2 crossed가 된다.

(3)

Node(1011)에서 Node(0101)로, Node(0111)에서 Node(1001)로 메시지를 동시에 보내려고 한다. 4-bit로 각 노드를 표현하였기 때문에 16x16 omega network가 필요하고, **Destination-tag routing**을 사용하는 경우는 각 switch에 빨간색으로, **XOR-tag routing**을 사용하는 경우 파란색으로 각 switch의 설정을 도시하였다. 각 switch의 설정이 서로 모순인 경우가 없으므로 blocking은 존재하지 않는다 (U: upper output / L: lower output, P: pass-through / C: cross).



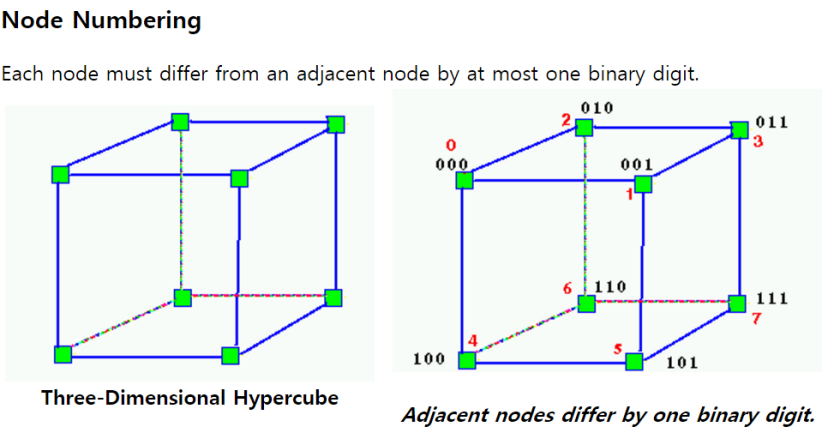
2. In Hypercube network,

(1) Discuss the routing mechanism used for hypercube (*E-cube routing*).

(2) Consider a 64-node hypercube network. Based on the E-cube routing algorithm, show how to route a message from node (101101) to node (011010). All intermediate nodes must be identified on the routing path.

(1)

E-cube routing은 hypercube network의 각 노드가 hamming code 방식으로 주소를 할당받는 경우에 사용될 수 있다 (인접한 두 노드는 오직 1bit만 다름). 이와 같은 주소 할당은 hypercube의 dimension이 늘어날 때 대응되는 두 노드를 오직 MSB 1bit만이 다르도록 설정하는 것으로 간단히 구현할 수 있다. 이와 같은 주소 할당 방식을 [그림2]에 간략히 보였다.



Hamming code 방식의 hypercube 노드 주소 할당

Hyper network 노드의 주소가 할당되었을 때 E-cube routing은 다음과 같은 방식으로 이루어진다.

1) 메시지를 보낼 현재 노드 주소와 목적지의 주소를 XOR한다. XOR된 값을 이진수로 변환하면 1이 설정된 offset이 LSB로부터 몇 번 bit를 toggle해야할지 결정해준다. 예를 들어, XOR 된 값이 101이면 현재 노드 주소의 1번 째 비트와 3번째 비트를 toggle 시켜준 두 값이 다음 destination 후보로 설정될 수 있다.
2) toggle된 값이 목적지의 주소와 다르면 1번을 다시 수행하고, 같아지면 3번으로 넘어간다.
3) 목적지에 도착했으므로 종료.

(2)

현재 노드 주소 101101부터 목적지 노드 011010까지 (1)에 기술된 E-cube routing을 사용하면 routing path는 아래와 같다.

101101 -> 101100 -> 101110 -> 101010 -> 111010 -> 011010

3. Suppose $T_{serial} = n$ and $T_{parallel} = n/p + log_2(p)$, where times are in microseconds. If we increase p by a factor of k, find a formula for how much we'll need to increase n in order to maintain constant efficiency. How much should we increase n by if we double the number of processes from 8 to 16 ? Is the parallel program scalable ?

p가 k배 증가할 때, efficiency가 변하지 않기 위해선 n은 다음과 같이 증가되어야 한다.

Efficiency = T(serial) / (p * T(parallel)

따라서, 기존 Efficiency는 n / (n + p * log(p))이다.

여기서 p를 k배 증가시키게 되면 efficiency는 n′ / (n′ + k * p * log(k * p))이다.

n을 얼마나 증가시켜야 하는지

= n′ / n

= k * log(k * p) / log(p)

= k + k * log(k) / log(p)

따라서, n을 ***k + k * log(k) / log(p)***배 증가 시키면 같은 efficiency를 얻을 수 있다.

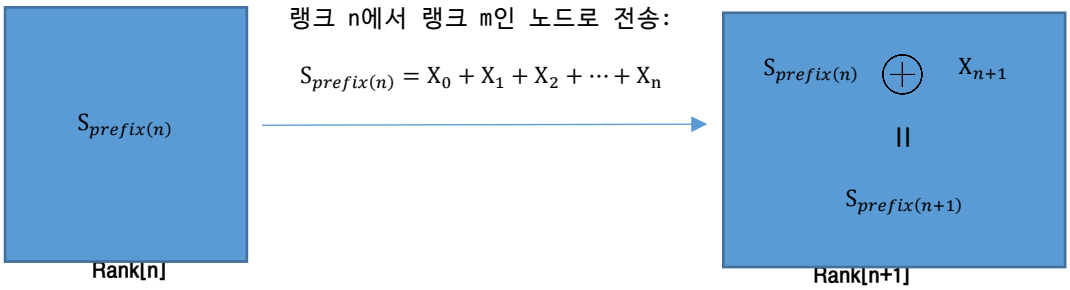
p = 8개에서 16개로 k = 2배가 되면, 위의 공식으로부터 얻을 수 있는 n의 증가는 **k + k * log (k) / log (p) = 2 + 2 * log(2) / log(8) = 8 / 3** 배 이다. 한편, 주어진 parallel program은 **scalable**한데, 이는 weak scalability test에 의해 p의 증가율 k에 대해 항상 대응되는 n의 증가율 (k + k * log(k) / log(p)) > k 를 찾을 수 있기 때문이다.

4. (MPI - Primitives) Finding **prefix sums** is a generalization of global sum. Rather than simply finding the sum of n values, $X_0 + X_1 + X_2 + \ldots + X_{n-1}$, the prefix sums are the n partial sums X_0, $X_0 + X_1$, $X_0 + X_1 + X_2$, ..., $X_0 + X_1 + \ldots + X_{n-1}$. MPI provides a collective communication function, *MPI_Scan*, that can be used to compute prefix sums.

(1) Understand the semantics of *MPI_Scan* operation and devise at least **two** parallel prefix sum algorithms (i.e., explain the algorithms without MPI notation).
(2) Implement this operation using only MPI send and receive (blocking and non blocking) calls. When implementing your solutions, make sure that your implementation is not dependent on the number of processors used. Verify your results by generating n random integers and compare the performance with that of **original** *MPI_Scan* as you increase the number of nodes involved. Discuss the results.

(1) 두 개의 parallel prefix sum 알고리즘을 고안하라.

첫 번째 parallel prefix sum 알고리즘



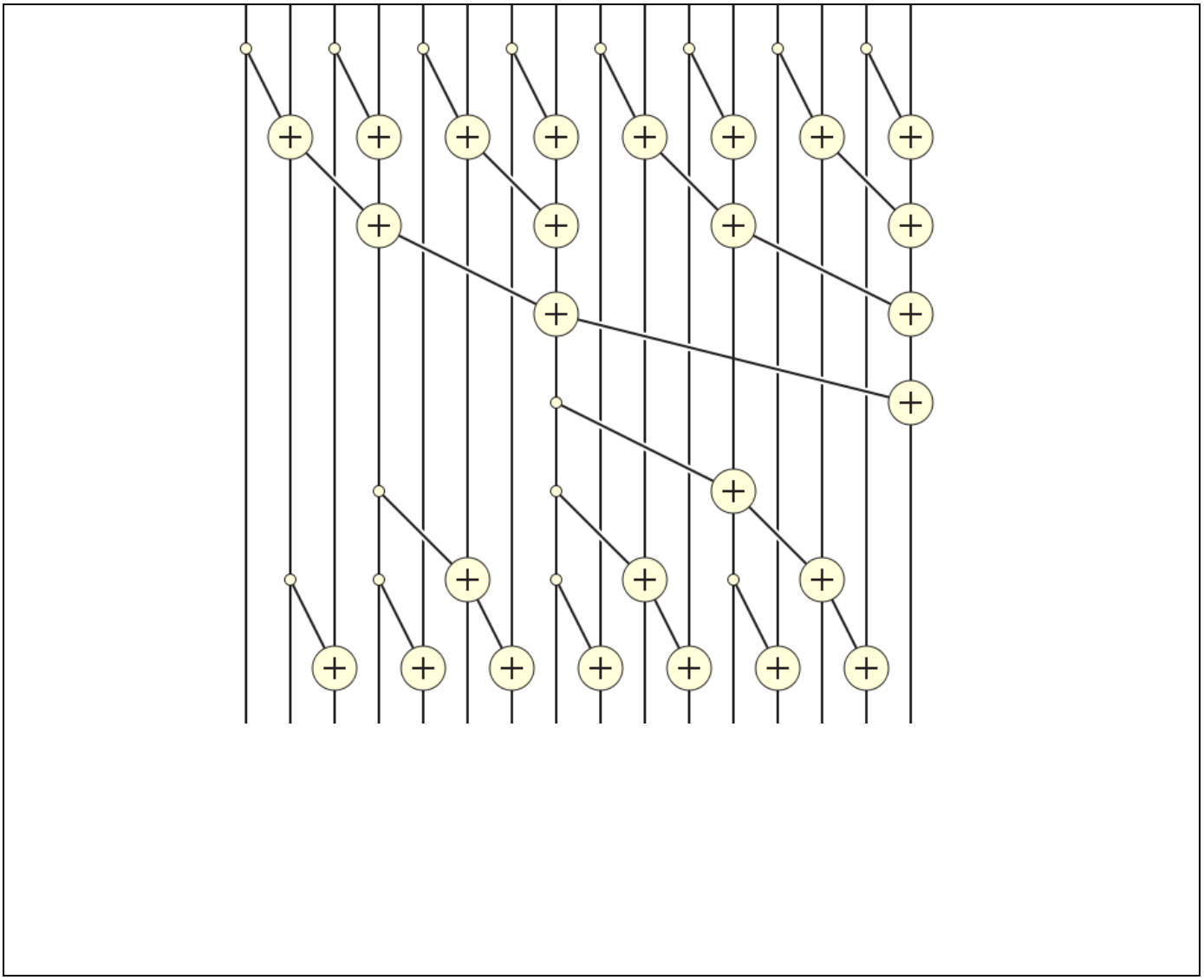
설명:

1. 랭크 0인 노드에서 n개의 랜덤숫자 생성. ($X_0, X_1, X_2, \ldots, X_{n-1}$)
2. 랭크 0인 노드를 제외한 나머지 노드들 각각에게 $X_1, X_2, \ldots, X_{n-1}$ 값 하나씩을 순차적으로 전송한다. (2를 수행하면, 랭크 0인 노드는 X_0, 랭크 1인 노드는 X_1, …, 랭크 n-1인 노드는 X_{n-1} 값을 하나씩 갖는 상태가 된다)
3. 랭크 0이 랭크 1에게 $S_{prefix0} = X_0$을 전송한다. 랭크 1이 $S_{prefix0}$을 수신하면 자신이 갖고 있던 X_1과 더하여 $S_{prefix1} = S_{prefix0} + X_1$를 생성한다. (초기화)
4. 랭크 n-2이 랭크 n-1에게 $S_{prefix(n-2)}$을 전송한다. 랭크 n-1이 $S_{prefix(n-2)}$을 수신하면 자신이 갖고 있던 X_{n-1}과 더하여 $S_{prefix(n-1)} = S_{prefix(n-2)} + X_{n-1}$를 생성한다. (점화식)
5. 3, 4번을 모두 거치면 각 노드의 랭크에 순서대로 $S_{prefix0}, S_{prefix1}, \ldots, S_{prefix(n-1)}$가 저장된다.

첫 번째 parallel prefix sum 알고리즘의 특징:

- O(N) 알고리즘
  (rank N인 노드가 prefix sum을 구해야 rank N+1인 노드가 prefix sum을 계산할 수 있기 때문)

두 번째 parallel prefix sum 알고리즘



두 번째 parallel prefix sum 알고리즘의 특징:

- O(NlogN) 알고리즘
  (rank N인 노드가 prefix sum을 구해야 rank N+1인 노드가 prefix sum을 계산할 수 있기 때문)

(1) MPI_Scan을 이용하지 않고 MPI_Send, MPI_Recv (blocking version) / MPI_ISend, MPI_IRecv (non-blocking version) 구현하고, 기존의 MPI_Scan 버전과 성능을 비교하라.

Number of prefix sums to calculate (number of Processors)	**2**	**5**	**10**	**15**	**20**	**25**
Original MPI_Scan Execution time (second)	0.00009	0.04	0.16	0.02	0.18	0.3
Blocking Scan Execution time (second)	0.00007	0.08	0.2	0.02	0.316	0.44
Non-blocking Scan Execution time (second)	0.00008	0.048	0.11	0.018	0.13	0.23

분석 결과:

1) Non-blocking scan 방식이 다른 프로그램들보다 성능이 좋았다.
2) 계산해야 할 prefix sum 개수가 늘어남에 따라 non-blocking scan 방식이 타 프로그램에 비해 더 일찍 끝난다는 것을 확연히 느낄 수 있었다.
3) 처리 속도: Blocking scan < Original MPI_Scan < Non-blocking scan

5. (MPI – Simple Example) The *collapse* of a set of integers is defined as the sum of the integers in the set. Similarly, the collapse of a single integer is defined as the integer that is the sum of its digits. For example, the collapse of the integer 134957 is 29. This can clearly be carried out recursively, until a single digit results: the collapse of 29 is 11, and its collapse is the single digit 2. The ultimate collapse of a set of integers is just their collapse followed by the result being collapsed recursively until only a single digit {0, 1, ..., 9} remains. Your task is to write a MPI program that will find the ultimate collapse of a one-dimensional array of N integers using the approaches given below and compare them. Check also whether you have same results or not.

(1) Use K computers in parallel, each adding up approximately N/K of the integers and passing its local sum to the master, which then totals the partial sums and forms the ultimate collapse of that integer.
(2) Use K computers in parallel, each doing a collapse of its local set of N/K integers and passing the partial result to a master, which then forms the ultimate collapse of the partial collapses.
(3) Use K computers in parallel, each doing an ultimate collapse on each one of its local set of N/K integers individually, then adding the local collapsed integers and collapsing the result recursively to obtain a single digit. Each of the K then sends its digit on to the master for final summing and ultimate collapse.
(4) Use 1 computer to process all N integers using any of the three approaches above.

(1) – (교수님의 지시로 생략)

(2) – dp1_20111599.tar의 mpi_problem5 폴더의 5_1.c

(3) – dp1_20111599.tar의 mpi_problem5 폴더의 5_1.c

(4) – run.sh에서 np를 1로 설정하고 (2)와 (3)을 실행시킨다.

(2)와 (3)의 비교:

1) (2)는 각 노드의 N/K 정수들을 모두 더하고 collapse 결과를 master에게 반환하여 master가 수신받은 K개 정수들의 ultimate_collapse를 구한다.

2) (3)는 각 노드의 N/K 정수들을 모두 더하고 ultimate_collapse 결과를 master에게 반환하여 master가 수신 받은 K개 ultimate_collapse들의 최종 ultimate_collapse를 구한다.

3) (2)와 (3)의 결과는 동일하다.

6. (MPI – Image Processing) It is generally agreed that topics in image processing have the high potential for significant parallelism. In this question, you are to read in a PPM(Portable Pix Map) file in P6 format(full color), and write sequential and parallel program written in MPI to *flip* a image horizontally (mirroring) and reduce the image to *grayscale* by taking the average of the red, green, and blue values for each pixel. Compare the performance of sequential and parallel versions of the program and discuss the results over a cluster of workstations. Use different PPM files with various data sizes and discuss the scalability aspects of your code as you increase the number of nodes. When submitting your code, include the short report on the questions above, the sample PPM files used and your programs (sequential and parallel versions). Also include the name of the PPM viewer(Linux version) in a readme file.

PPM이란?

Portable Pixmap Format의 약자. 플랫폼 간에 쉽게 교환될 수 있도록 만들어진 이미지 파일 포멧.

File format description [edit]

Each file starts with a two-byte magic number (in ASCII) that identifies the type of file it is (Pl

Type		Magic number		Magic number		Extension	Colors
Portable BitMap	[1]	P1	ASCII	P4	binary	.pbm	0–1 (black & white)
Portable GrayMap	[2]	P2	ASCII	P5	binary	.pgm	0–255 (gray scale)
Portable PixMap	[3]	P3	ASCII	P6	binary	.ppm	0–255 (RGB)

그림 1.

[그림 1]에서 볼 수 있듯이 변환할 컬러 이미지는 PPM 형식으로 되어 있고, 결과 이미지는 grayscale 이미지인 PGM 형식으로 변환되어야 한다. 즉, .ppm 파일이 작성한 프로그램의 input이 되고, .pgm 파일이 그것의 output으로 생성되어야 한다.

PPM 파일 예제 (ASCII 형태 - magic number가 P3인 경우)

```
P3
3 2
255
# The part above is the header
# "P3" means this is a RGB color image in ASCII
# "3 2" is the width and height of the image in pixels
# "255" is the maximum value for each color
# The part below is image data: RGB triplets
255   0   0     0 255   0     0   0 255
255 255   0   255 255 255     0   0   0
```

Image (magnified)

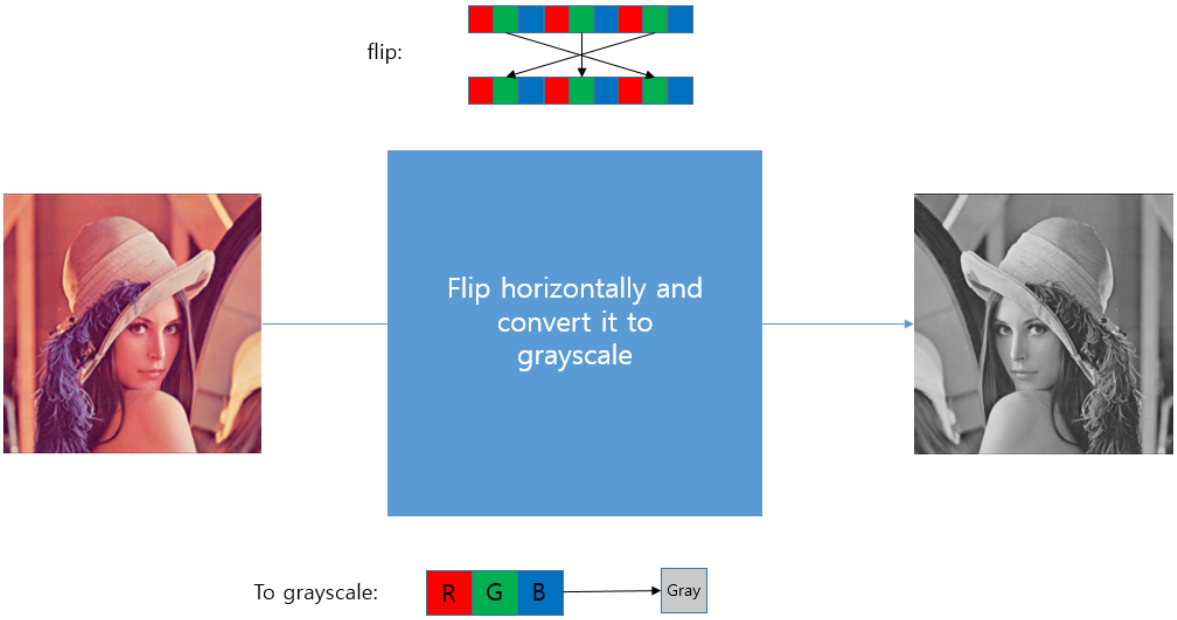
PGM 파일 예제 (ASCII 형태 - magic number가 P2인 경우)

```
P2
# Shows the word "FEEP" (example from Netpbm man page on PGM)
24 7
15
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  3  3  3  3  0  0  7  7  7  7  0  0 11 11 11 11  0  0 15 15 15 15  0
0  3  0  0  0  0  0  7  0  0  0  0  0 11  0  0  0  0  0 15  0  0 15  0
0  3  3  3  0  0  0  7  7  7  0  0  0 11 11 11  0  0  0 15 15 15 15  0
0  3  0  0  0  0  0  7  0  0  0  0  0 11  0  0  0  0  0 15  0  0  0  0
0  3  0  0  0  0  0  7  7  7  7  0  0 11 11 11 11  0  0 15  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
```

Example (magnified)

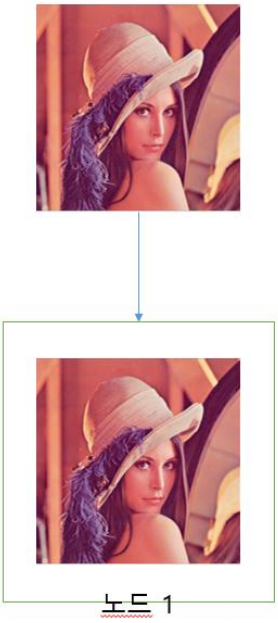
PPM의 magic number는 반드시 P3 혹은 P6.
PGM의 magic number는 반드시 P2 혹은 P5.

구현해야 하는 프로그램 내용:



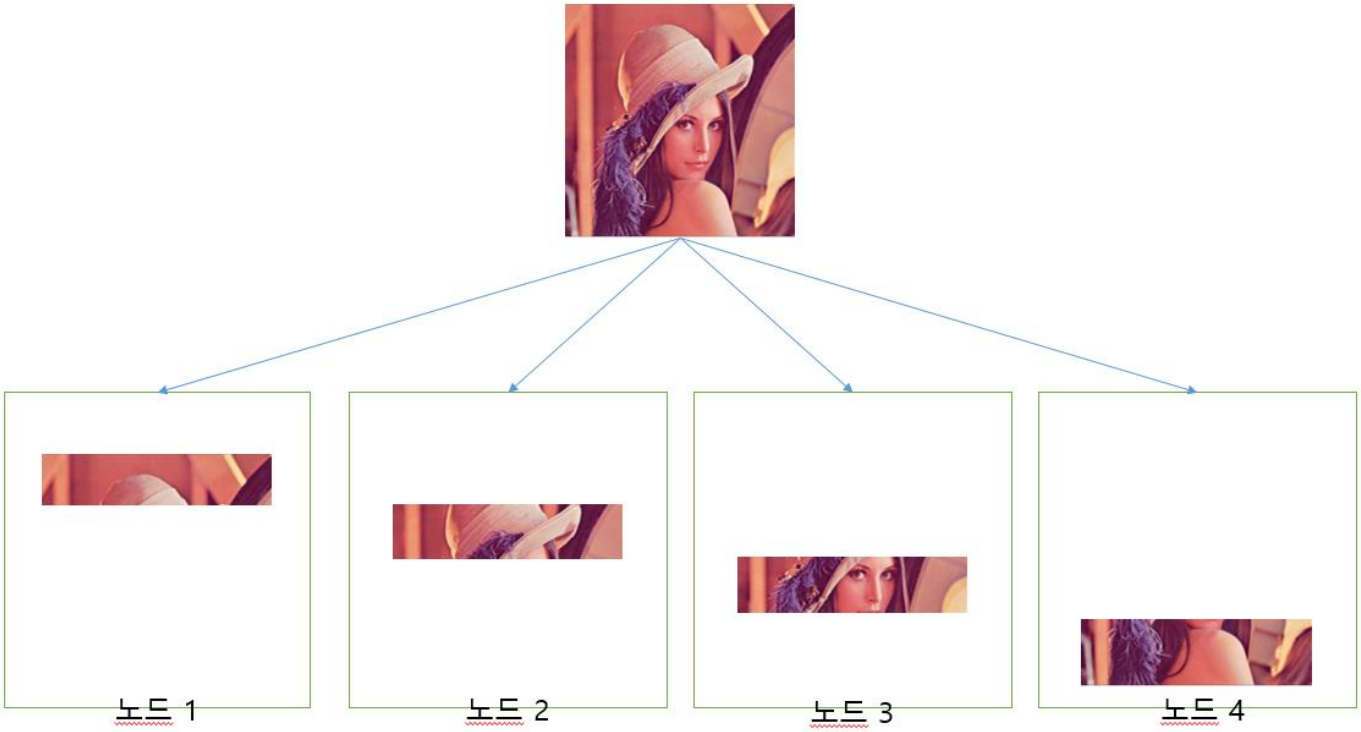
위 그림을 처리하는 serial program과 parallel program을 작성해야 한다.

Serial Execution 모델:



* 한 개의 노드가 이미지 프로세싱 수행 (flip_and_rgb2gray_serial.c)

Parallel Execution 모델:



* 여러 개의 노드가 분산처리 작업 수행 (flip_and_rgb2gray_parallel.c)

Compare the performance of sequential and parallel versions of the program and discuss the results over a cluster of workstations.

1) 이미지 데이터 크기 변화 (노드의 개수 고정 – 10개)

Serial vs Parallel version

Data size	128 x 128 (CatLogo.128.ppm)	256 x 256 (Clown.256.ppm)	512 x 512 (Embryos.512.ppm)	1024 x 1024 (Iggy.1024.ppm)
Serial execution time (second)	0.068	0.2	0.6	2.2
Parallel execution time (second)	0.009	0.106	0.100	0.364

수행 결과: 이미지 데이터 크기가 커짐에 따라 수행시간이 늘어남을 알 수 있다.

2) 노드의 개수 변화 (이미지 데이터 크기 고정: Iggy.1024.ppm - 1024 x 1024, 용량: 3MB)

Serial vs Parallel version

Number of processors	1 (serial)	5 (parallel)	10 (parallel)	15 (parallel)	20 (parallel)	25 (parallel)
Execution time (second)	2.8	2.7s	2.63	2.82	2.68	2.61
Received Image block sizes from slaves (KB)	0	839	943	977	995	1006

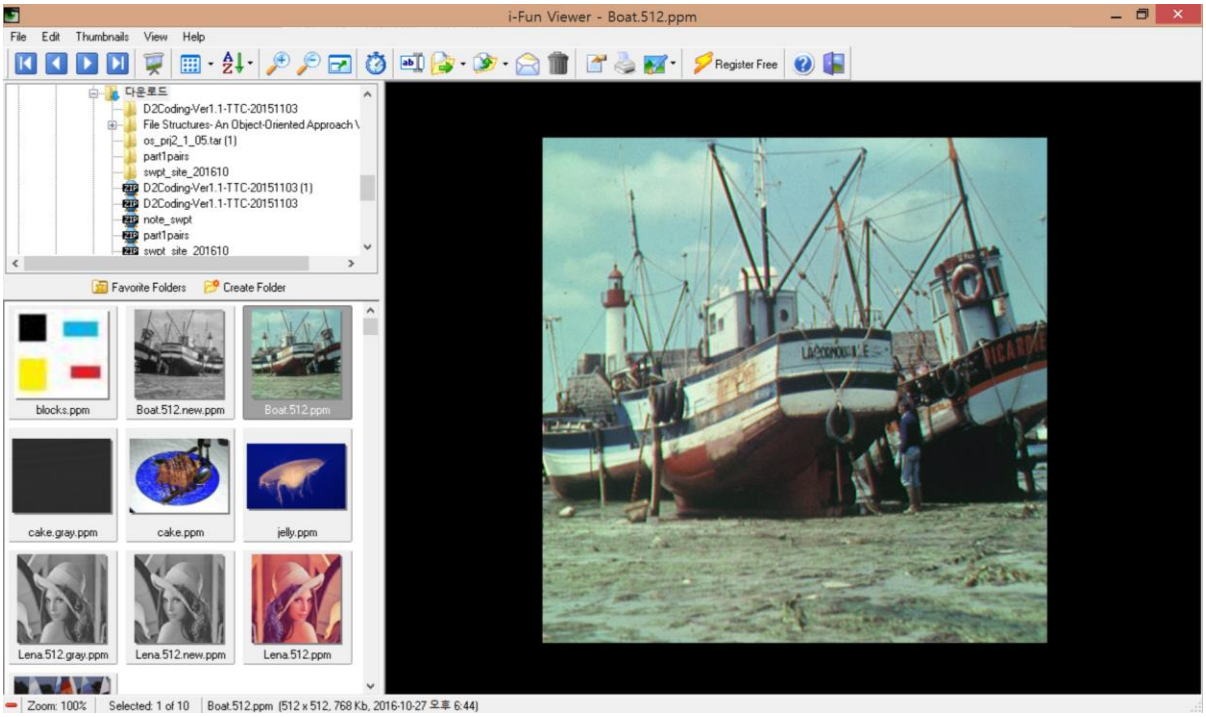
수행 결과: 노드 개수의 크기를 늘려도 수행시간은 크게 줄어들지 않는다.

원인 분석:

1) 각 노드에서 분산 처리된 이미지 block들을 master 노드에게 전송하는 부분에서 latency 발생.
2) 노드의 개수가 증가함에 따라 master 노드가 각 노드로부터 수신해야 되는 이미지 block 크기는 줄어드나 전체 수신 크기는 점점 늘어남. 자신을 제외한 모든 노드들로부터 수신하는데 걸리는 시간이 증가할 수 밖에 없다. (master 노드 자신이 처리해야 하는 이미지 크기가 줄어들기 때문에 그 block을 제외한 나머지 이미지 데이터는 증가하기 때문)

flip & convert to grayscale 수행 결과:

Before: Bot.512.ppm



After: Boat.512.new.ppm