**Parallel/Distributed Computing (CSEG414/CSE5414) Assignment #1**

**120190211 한장훈**

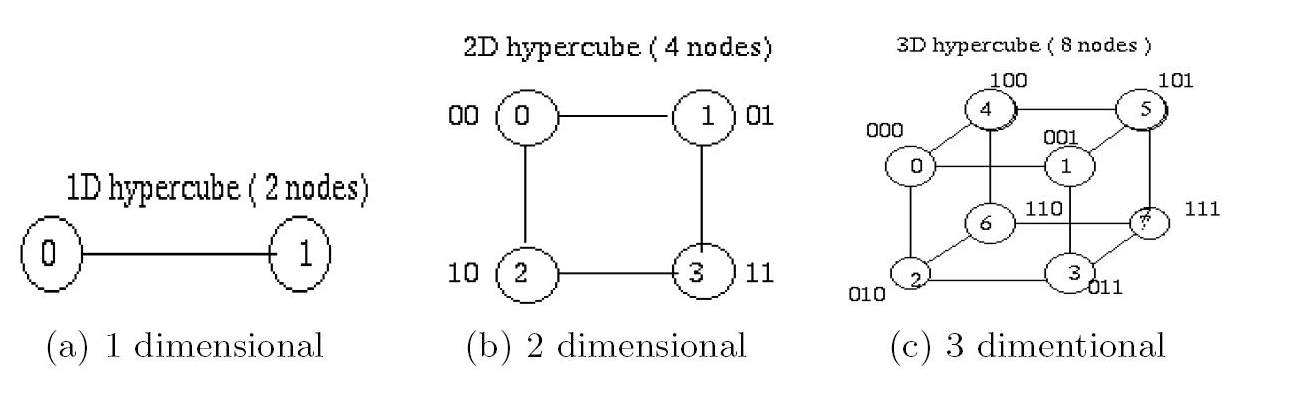
**1. In Hypercube network,**

**(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 해밍 코드를 사용하여 하이퍼 큐브 네트워크의 노드에 주소를 부여하여 라우팅 한다. 이때 해밍코드는 인접한 두 숫자가 1bit만 다르게 된다. 하이퍼 큐브의 dimension이 늘어날 때 MSB가 1bit가 차이나도록 노드를 설정 함으로써 확장이 가능하다. 다음은 dim이 1 ,2 3일 때 예시이다.



이렇게 하이퍼큐브 네트워크의 노드에 주소를 할당하고 E-cube routing을 진행하는데 순서는 다음과 같다.

1. 목적지와 현재 노드를 XOR하여 가장 가까운 1로 설정된 bit를 LSB로부터 몇 번째 위치 인지 확인 후 현재 노드의 주소의 대응되는 위치를 토글한다.

2. 1번의 과정을 목적지에 도착할때까지 반복한다.

2) 출발노드를 101101 도착노드를 011010 이라고 하면 경로는 다음과 같다.

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

**2. An application with a 5% non-parallelizable part, is to be modified for parallel execution. Currently on the market there are two parallel machines available: machine X with 4 CPUs, each CPU capable of executing the application in 1 hour on its own, and, machine Y with 16 CPUs, with each CPU capable of executing the application in 2 hours on its own. Which is the machine you should buy, if the minimum execution time is required?**

Machine X의 CPU는 1시간이 걸리는데 5%가 non-parallelizable part 라고 하였으므로 는 0.05시간이다. 4개의 CPU를 사용 할 경우 X의 총 시간은 + = 0.05+0.95/4 = 0.2875 시간이 걸린다. Machine Y의 CPU는 2시간이 걸리므로 마찬가지로 계산하면 16개의 CPU를 사용할 때 걸리는 시간은 + = 0.1+1.9/16 = 0.21875 시간이다. 따라서 Machine Y를 사는데 Reasonable한 선택이다.

**3. Suppose Tserial = n and Tparallel = n/ p + log2(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 ? Hint: Suppose we increase the number of processors, p. If we can find a corresponding rate of increase in the problem size so that the program always has the same efficiency E, then the program is (weakly) scalable.**

Processor의 개수를 p, efficiency를 E라고 하면 다음의 식이 성립한다.

p를 k배 증가 시킨다고 했을 때 동일한 E를 얻기위한 을 구하자.

에 관해서 정리하면 이 된다. 즉 n을 배 증가시키면 같은 E가 유지된다. P를 8에서 16으로 증가시키면 n은 즉 배 증가시켜야 E가 유지된다. 이때 n을 증가시켜 같은 E를 가지게 할 수 있으므로 weakly scalable 하다고 할 수 있다.

**4. (MPI – Primitives) Finding prefix sums is a generalization of global sum. Rather than simply finding the sum of n values, X0 + X1 + X2 + ... + Xn-1, the prefix sums are the n partial sums X0, X0 + X1, X0 + X1 + X2, ..., X0 + X1 + ... + Xn-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.**

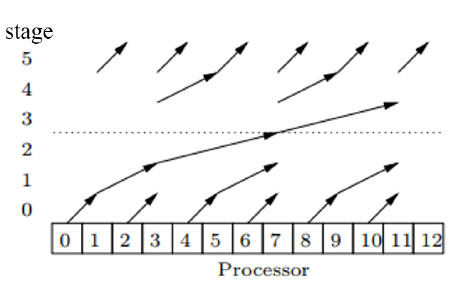
1)

**Understand the semantics of MPI\_Scan operation**

[j…k] 까지의 prfixsum을 라고 하자. 모든 노드를 fully connected 이고 single port로 통신하며 synchronized round 이고 같은 길이의 패킷을 교환한다. 크게 3가지의 모델이 있다.

Binomial tree, Simultaneous binomial tree , Pipelined binary tree이다.

이중 Binomial tree 는 라고 했을 때 n개의 round로 이루어져있고 각각의 round는 up-phase와 downphase로 이루어져있다. k번째 라운드에서 (k=0, … ,n-1) up-phrase시 각각의 PE(processing element) j가 을 만족할 때 PE 로부터 정보를 받게 된다. 받은 PE들은 partial result를 더하게 되고 k개의 round 후에 ⊕[j – + 1..j] 를 가지게 된다. Down phase에선 round를 다시 n에서 1로 줄이게 되는데 j ∧ () = 만족시키는 PE j 들은 PE j + 로 partial result를 보내게 된다. 최종적으로 ⊕[0..j + ] 를 얻게 된다.



**two parallel prefix sum algorithms**

가) 제일 간단한 prefixsum 알고리즘은 순차적으로 prefix sum을 진행하는 것이다. 당연히 O()의 시간복잡도를 가지게된다. 모든 네트워크와 연결될 필요가 없고 인접한 노드끼리만 연결이 되어있으면 된다. K= 0,1,2,…,n-2 일때 rank k와 rank k+1만 연결되어 있어도 구현이 가능하다.

알고리즘은 다음과 같다.

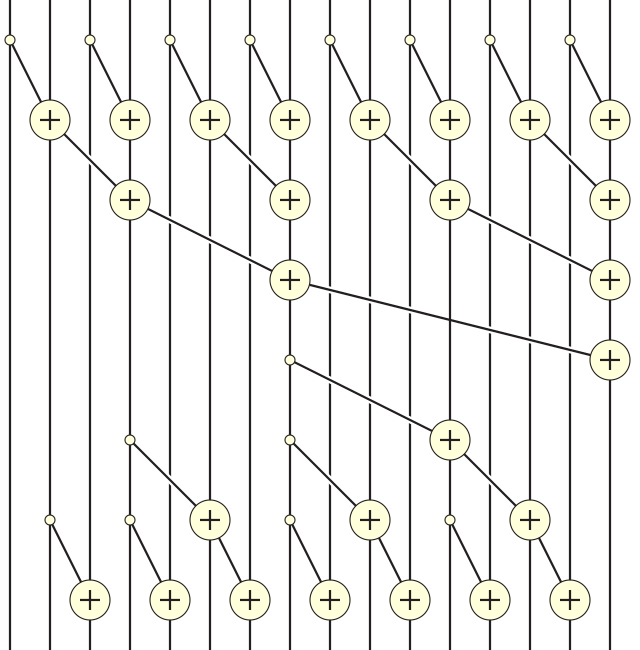
1. rank 0인 노드에서 random number()를 생성한다.

2. n개의 노드로 각각 randomnumber 를 전송한다.

3. rank 0인 노드는 prefix\_sum을 으로 설정한다.

4. K= 0,1,2,…,n-2 에서 rank k의 노드는 자신의 prefix\_sum을 rank k+1로 전송하고 rank k+1은 받은 값에 를 더하여 prefix\_sum으로 저장한다. 반복하여 최종 prefix sum을 구한다.

나) 다른 방법은 트리를 사용하는 것이다. 앞에서 언급한 Binomial tree가 그 예이다. 트리를 사용하기 때문에 O() 의 시간 복잡도를 가진다. 실제 MPI SCAN에서 사용하는 방법과 동일하다.



트리를 사용한 parallel prefix sum 알고리즘

2)

1에서 설명한 관점에서 2가지 알고리즘을 비교하고자 하였다.( sequential한 방법과 tree를 사용한 방법) 따라서 한 코드는 blocking send/ recv를 통해서 sequential하게 구현하였으며 tree의 알고리즘의 경우 MPISCAN의 알고리즘과 동일하기 때문에 다른 한 코드는 MPI\_Scan을 사용하여 구현하였다. 문제의 수를 n개라고 했을 때 프로세스의 수가 더 부족한 경우가 많으므로 당연히 각각의 프로세스가 문제 n개를 나눠서 알고리즘에 맞게 prefixsum을 구하도록 하였다. 문제의 수가 프로세스의 수와 다를수도 있기 때문에 같은 포션으로 나누되 나머지가 있는경우에는 rank0의 프로세스에 나머지까지 처리하도록 하였다. 따라서 프로세스의 수와 상관없이 작동한다. 알고리즘이 잘돌아가는지 확인하기 위해 마지막 rank의 최종 prefixsum과 마스터 노드에서 계산한 total sum이 같은지를 확인하였다. 다음은 실험 결과이다.

1. 문제의 크기(n)가 십만일 경우



(x축 프로세스의 수 , y축 걸린시간)

2. 문제의 크기가 백만 일 경우



(x축 프로세스의 수 , y축 걸린시간)

결과분석.

N이 십만일 때 Blocking의 경우 프로세스 수가 1개일 때는 MPISCAN보다 빠르지만 점점 프로세스의 수가 증가할수록 MPISCAN보다 속도가 느려지는 것을 알 수 있다. 이는 sequential 한 알고리즘보다 트리구조의 알고리즘이 시간복잡도가 작고 효율적이기 때문이다. 1의 경우 blocking 방식은 프로세스의 수가 4일 때부터 속도가 다시 늘어나기 시작하는데 이는 분할해서 얻는 시간적 이점보다 통신 간의 오버헤드가 크기 때문이다. Blocking 방식에 비해 MPI\_SCAN은 그나마 소요시간이 지속적으로 감소하긴 하나 그 비율이 점차 줄어든다.

N이 백만일 경우에도 N이 십만일때와 비슷한 양상을 보인다. 차이점이라고 하면 N이 십만일 때 비해 통신오버헤드 대비 분산해서 얻는 이득이 크다는 것이다. 자세히 보면 앞에서 N이 십만일 때 blocking의 경우 프로세스가 4개부터 이점이 없었지만 N이 백만일 경우 프로세스가 5일때까지 지속적으로 분산프로그래밍의 효과가 나타난다. 이는 parallel portion이 늘어났기 때문이다. 하지만 프로세스수가 10개일때는 역시 통신 오버헤드가 커져서 이점이 없다.

MPI SCAN의 경우 N이 십만에 비해 N이 백만일 때 프로세스가 증가하면 더 시간이 감소하는 경향이 있었고 문제 사이즈가 커졌기 때문에 프로세스 개수가 10일 때도 여전히 소요 시간이 감소하는 경향을 보였다.

**5. (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**

**(a) flip an image horizontally (mirroring) and**

**(b) reduce the image to grayscale by taking the average of the red, green, and blue values for each pixel and**

**(c) smooth the image by calculating the mean of each pixel’s value and its eight neighbours (some algorithms consider only the values from the diagonal neighbours or the horizontal and vertical neighbours).**

**When implementing your MPI programs, try to use MPI derived data types as much as you can. 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.**

**1. implement**

프로그램은 다음과 같은 과정을 거친다.

1. 이미지를 읽는다.

2. 이미지를 좌우로 미러링한다.

3. RGB를 평균하여 grayscale로 바꾼다.

4. 3\*3 평균 smooth matix를 곱하여 스무딩한다.

5. 이미지를 저장한다.

구현시 각 단계의 함수를 구현하였다. 이미지를 읽는 함수는 imread, 미러링 하는 함수는 horizontal\_flip\_RGB ,grayscale로 바꾸는 함수는 RGB2gray 스무딩하는 함수는 graysmooth 저장하는 함수는 imwritegray이다.

Sequential 의 경우 하나의 프로세서가 처리하기 때문에 앞써 말한 흐름대로 코드를 짜서 gcc로 컴파일 후 실행하면 된다. 다음은 squential로 한 결과이다.

**Sequential 프로그램을 통한 이미지생성**



왼쪽: 원본 오른쪽: 처리후

Parallel의 경우 이미지 처리를 분산적으로 하기위해 일을 나눠야 한다. 이때 이미지 처리는 픽셀 마다 처리하기 때문에 이미지를 일정하게 나눠서 각자 프로세스에서 처리하고 나중에 합쳐서 저장 해주면 된다. 이때 가로로 나눌것인지 세로로 나눌것인지 생각해봐야 하는데 우리는 미러링을 해야하기 때문에 가로로 나누는 것이 타당하다. 왜냐하면 미러링을 하려면 반대편 픽셀정보를 알아야하는데 세로로 할경우 반대편 픽셀정보가 없어서 분산 처리가 불가 하기 때문이다.

한편 스무딩의 경우 주면 자신을 합해서 총 9개의 픽셀에 대해서 평균을 구해서 자신 픽셀값으로 스무딩하게 되는데 이때 완전히 나눠서 프로세싱할경우 worker 에서 잘린 맨 윗줄과 밑줄을 스무딩할 때 그 픽셀값을 모르는 경우가 생긴다. 이렇게 되면 해결책은 2가지이다. 6개로만 스무딩을 하는경우 아니면 중복해서 맨 윗줄과 밑줄까지 전달해주는 경우이다. 본 프로그램에서는 후자의 방법을 통해 스무딩을 동일하게 해주었다.

**Worker 프로세스에서 처리한 이미지**



**Parallel 프로그램을 통한 이미지생성**



왼쪽: 원본 오른쪽: 처리후

위 그림에서 볼수 있듯이 각각의 프로세스에서 처리한 후 합쳐서 최종 이미지를 저장하였다.

사진을 좌우반전한 후 grayscale로 바꾸고 스무딩한 결과이다. 리눅스의 ImagemagicK를 통해서 출력하려고 했으나 cspro 권한이 없어서 설치가 안됐으므로 honeyview를 사용하였다.

**2. try to use MPI derived data types as much as you can**

parallel 하게 프로그램시 마지막에 master node로 각자 worker들이 처리한 부분을 전달해 줘야한다. 이때 단순하게 픽셀값을 특정 개수만큼 send하는 경우도 있지만 derived data type을 사용해서 보낼 수도 있다. 따라서 프로그램에서 partition만큼의 길이를 가지는 contiguous한 derive type을 create 후 사용하였다. (스무딩후 최종 값은 1차원 배열 이다.)

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

|  |  |
| --- | --- |
|  | House(111\*132) |
| 1 | 6.99E-03 |
| 2 | 5.49E-03 |
| 3 | 4.89E-03 |
| 4 | 3.14E-03 |
| 5 | 4.89E-03 |
| sequential | 5.63E-03 |

|  |  |
| --- | --- |
|  | mcFaddin(800\*600 |
| 1 | 1.39E-01 |
| 5 | 1.06E-01 |
| 10 | 9.60E-02 |
| sequentail | 1.73E-01 |

실험은 2가지에 관해서 진행하였다. 작은 이미지인 house와 큰 이미지인 mcfaddin에 관하여 진행하였는데 작은 이미지의 경우 4까지는 분산프로그래밍의 효과가 있다가 5부터 오버헤드로 인해 증가하기 시작한다. 두번째 큰이미지의 경우에는 10까지도 분산프로그래밍의 효과가 있으며 프로세스의 수가 20일 때 정도부터 감소하기 시작한다. 이는 역시 통신시 발생한 오버헤드로 인하여 분산프로그래밍의 이점을 감쇄시켰기 때문이다.

Parallel 한 방법이 sequential 한 방법에 비해 두가지 모두 최대 약 1. 7배정도 효과가 있었다. 이론상 프로세스가 증가하면 그만큼 시간도 줄어야 하지만 통신에 드는 오버 헤드 때문에 그렇게 나타나지 않은 것 같고 문제 사이즈를 충분히 키워도 역시 통신 간 오버헤드가 어느 순간 분산 프로그래밍의 이점을 뛰어넘게 된다.

**Mcfaddin의 결과**

위의 경우는 일반적인 스피드와 효율의 양상이다. 프로세스가 늘어날 때 속도는 증가하고 효율은 점차 감소하게 된다. 여기 까지 보면 어느 구간(2<p<10) 에서의 scalabity 관점에서 괜찮아 보이지만 어느 순간 프로세스 수가 너무 많아지면(p>20) 속도와 효율이 감소하게 된다. 따라서 전체적으로 보면 scalablity 관점에서 매우 poor 하다.

실험에 영향을 끼친 요인

이론과 다르게 실험결과가 나온이유는 첫째로 통신 오버헤드가 있다는 점이다. 또한 분산프로그램시 코드를 추가하는데 코드가 깔끔하지 않아서 시간이 오래 소요됐을 수도 있다. 또한 실험시 cspro4와 5를 사용하였는데 아무래도 같이 사용하다보니 분산컴퓨팅에서 코어 할당시 어떤 문제로 인해 딜레이가 생겼을 수도 있다고 생각한다.