Distributed Programming Assignment#1

120170220 변현기

1. (1) n개의 입력에 대하여 라우팅을 하기 위해서는 2x2의 스위치가 n/2개 필요하다. 따라서 각 스테이지는 n/2개의 스위치가 필요하다. 또한 하나의 입력에 대하여 2x2 스위치를 이용하여 n개의 결과로 routing하기 위해서는 개의 스테이지가 필요하다. 즉 총 필요한 스위치의 개수는 개이다.

(2) 출발지와 목적지의 비트를 xor연산을 한 결과를 이용하여 MSB부터 각 비트가 1이면 crossover, 0이면 straight through로 각 스테이지의 스위치를 세팅하면 된다.

(3) 1011과 0101을 xor하면 1110이 나오고, 0111과 1001을 xor하면 마찬가지로 1110이 나온다. 즉 각 스테이지별 스위치를 1110에 따라서 crossover, straight through로 설정을 하면 된다. 이 경우, 라우팅을 위한 스테이지 세팅이 같으며 충돌이 없기 때문에 blocking은 존재하지 않는다.

2. (1) hypercube에서 노드의 각 비트는 하나의 차원으로 볼 수 있다. hypercube에서 출발지와 목적지의 비트를 비교하여 LSB부터 왼쪽으로 한 비트씩 맞춰가면서, 차원을 맞춰가며, 출발지와 목적지의 비트를 같게 하는 방식으로 라우팅을 할 수 있다.

(2) 출발지 101101과 목적지 011010을 xor하면 110111이 나온다. 이 값과 출발지 101101의 각 비트를 LSB부터 차례대로 xor를 해주면 목적지인 011010이 나오게 된다. 따라서 이 경우의 라우팅 경로는 아래와 같다.

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

3. 주어진 문제에 대한 efficiency는 아래와 같다.

p가 k만큼 증가했을 때, efficiency를 유지하기 위해서 n을 m만큼 증가시킨다고 하면 아래와 같다.

이 때, efficiency를 유지하기 위해서는 위의 E와 같아야 하므로 다음과 같다.

이 식을 m에 대하여 정리하면 다음과 같다.

만약 프로세스를 8에서 16으로 증가시켰다면, 이므로 필요한 m의 값은 다음과 같다.

이 프로그램은 일 때 이므로 scalable하지 않다.

4. (1) prefix sum을 구현하는 가장 간단한 방법으로는 먼저 랭크가 낮은 프로세스가 다음 랭크의 프로세스로 자신의 sum을 send하는 방식이 있다. 이 방법의 장점으로는 각 노드가 계산해야 하는 덧셈의 수가 적다는 것이다. 그러나 이 방법은 순차적으로 랭크가 낮은 프로세스가 다음 프로세스에게 자신의 합을 보내고 받는 과정이 필요하다. 다른 방법으로는 브로드캐스팅을 통해 모든 노드가 배열 전체를 갖고 있으며, 각 노드가 자신의 prefix sum을 계산하는 방법이다. 이 방법은 각 노드가 수행해야하는 덧셈의 수가 많지만 프로세스 사이에 값을 보내고 받는 과정이 필요 없다.

(2)

위의 그래프는 prefix sum을 MPI\_SCAN과 send/receive 방식을 통해 구현한 implementation의 비교를 나타낸 그래프이다. x축은 np, number of process, 실행된 프로세스의 수를 나타내며 y축은 수행에 걸린 시간을 나타내며 단위는 ms이다. 각각의 결과는 n=1000에 대하여 5번씩 실행한 결과의 평균이다. 결과를 보면 프로세스의 수가 많아지면서 실행시간이 줄어드는듯 하였으나 8개의 프로세스 이후로 증가하는 것을 볼 수 있다. 이는 프로세스가 증가함에 따라 MPI를 이용하는 통신의 개수도 많아지므로 MPI로 인한 오버헤드라고 볼 수 있다. 또한 실제 물리적인 노드의 개수보다 프로세스의 개수가 많아지면서 context switching으로 인한 오버헤드도 존재할 것이다. 전체적인 성능은 MPI\_SCAN을 이용한 것이 더 느리게 나왔으며 이는 프로세스의 수가 증가함에 따라 더 큰 차이가 났다.

5. (1) quick sort는 divide and conquer 방식의 대표적인 정렬 알고리즘 중의 하나이다. pivot을 선택하여 pivot보다 작은 값은 pivot의 왼쪽으로, 큰 값은 오른쪽으로 나눈 다음 각각의 파티션에 대하여 재귀적으로 quick sort를 수행함으로써 배열을 정리한다. MPI를 이용하여 병렬적으로 quick sort를 하기 위해서 파티션을 나눈 후 뒤쪽의 파티션을 다른 프로세스로 보내서 처리하도록 구현하였다.

(2)

위 그래프는 구현한 parallel quick sort에 대하여 프로세스의 수와 수행시간 사이의 관계를 나타낸다. x축은 프로세스의 수이며 y축은 수행시간으로 단위는 ms이다. n=100000에 대하여 3번씩 실행한 결과의 평균이다. 프로세스가 증가함에 따라 수행시간이 줄어드는 것과 그 감소폭이 점점 줄어드는 것을 볼 수 있다.

(3) 프로세스가 증가함에 따라 수행시간이 줄어들었지만 선형적으로 감소하지는 않았다. 또한 감소폭도 점점 줄어들었다. 그 이유는 항상 모든 프로세스가 quick sort에 참여하지 않기 때문이다. 파티션을 나눈 후 뒤쪽의 파티션을 다른 프로세스로 넘겨서 처리하는 방식이므로 처음엔 하나의 프로세스만 quick sort에 참여하고 그 이후로 점차 참여하는 프로세스가 늘어나게 되므로 모든 프로세스를 완전히 활용하지 못하기 때문이다. 또한 MPI 통신으로 인한 오버헤드도 고려해야 한다.

6.