**분산 프로그래밍 과제 1**

김동환 20141508

1-1)

E-cube routing은 두 노드의 각 바이너리 라벨을 이용해 두 노드사이의 거리를 Xor 연산인 Hamming weight로 구할 수 있다.

src address를 S , dst address를 D라 하면 둘 사이 거리는 R = S xor R 해서 구한 R을 이진수로 표현했을때 1의 갯수이다.

S에서 출발하고 이제 R을 이진수로 표현했을때 least significant 부터 1인 지점으로 이동한다.

참고:

1-2)

S = 101101

D = 011010

R = S xor D = 110111

경로 : 101101 출발 -> 111101 -> 111111 도착

2)

machine X : 4 CPUs , each executing the app in 1 hour on its own

machine Y : 16 CPUs, each executing the app in 2 hour on its own

어플리케이션 수행해야 되는 양을 W라 하면 0.95W를 병렬화해서 수행해야 한다.

X머신의 각 cpu에 대해서 보면 1W = 1hour \* capacity / 1cpu 이다.

즉 x머신의 각 cpu의 capacity = 1W이다. (단위 시간당 하는 W의 양)

Y머신의 각 cpu에 대해서 보면 0.95W = 2hour \* capacity/ 1cpu

capacity = 0.5W이다.

오버헤드를 무시하고 병렬화를 하면

어플리케이션을 수행하는데 걸리는 시간은 4 cpus \* 1W \* hours + 0.05W = 1W이다.

머신 x에서의 수행 시간은 hours = 0.25(hour)이다.

반면 y에서의 수행시간은

16 cpus \* 0.475W \* hours + 0.05W = 1W

hours = 0.125(hour)이다.

즉 y머신이 더 빨리 수행하므로 사야한다.

3번 )

비례 인자를 k’이라하자

k ×

4-1) (1) Understand the semantics of MP I\_Scan operation and devise at least two parallel prefix sum algorithms (i.e., explain the algorithms without MPI notation)

첫번째 알고리즘은 아래와 같이 구현하려 합니다.

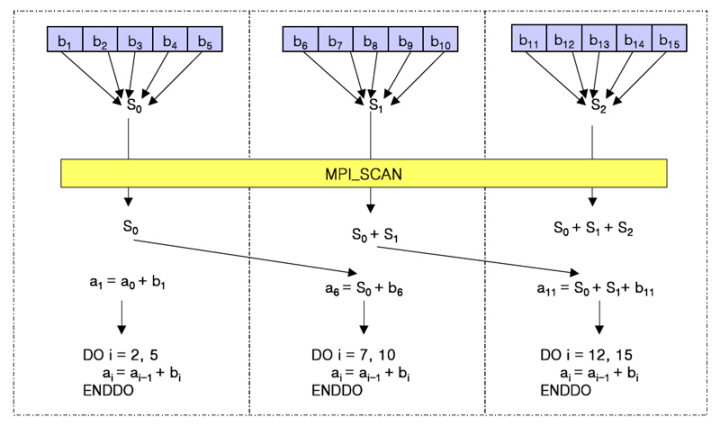


그림 출처 <<http://k-atoms.ksc.re.kr/mpi/3_3_7.html>>

위의 그림에서 a(i)는 0~i까지 원소의 prefix\_sum이다.

해당 알고리즘은 1~n까지의 수를 더하고, 8개의 프로시저가 있다면

0번 프로시저는 1~n/8, 1번은 n/8+1 ~ n/4까지 ..... , 각 프로시저는 맡은 구간의 합을 더하고

각 구간의 합을 S(index)로 표시한다.

그후 mpi\_scan을 이용하여 각 프로시져로 구간의 합을 넘김으로서, 프로시저 5번이 st번째 인덱스부터 dt번째 인덱스까지의 prefix\_sum을 구한다면 반복문을 통해서 mpi\_scan에 의해 전달 받은 값인 S0+S1+S2+S3+S4에 st번째 원소를 더 함으로서 a[st]를 구한다.

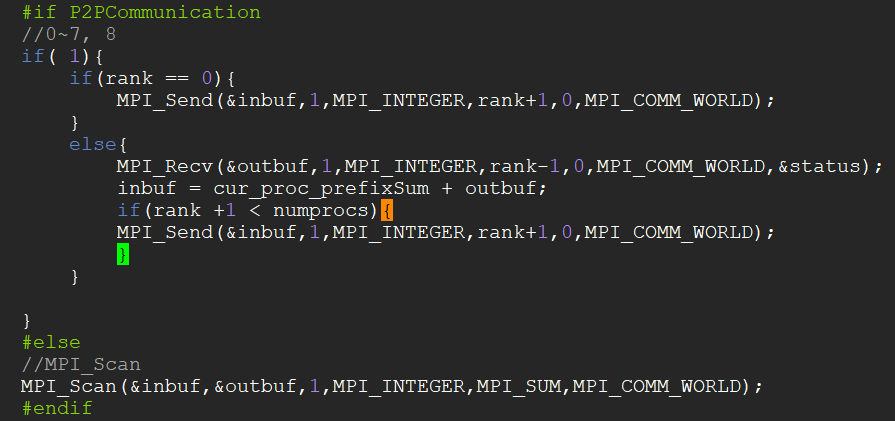
고안해본 두번째 알고리즘은

1. 마스터 노드가 다른 노드들이 각자의 구간에서 구한 합을 전달 받고(위와 같습니다). 즉 n번 노드는 자신이 연산을 맡은 st ~ dt까지의 부분합을 구합니다. 그리고 마스터 노드로 배열을 넘깁니다.
2. 마스터 노드는 종합적으로 prefix\_sum을구하는 알고리즘입니다. 즉 마스터 노드에서의 연산은 n번 노드의 배열을 전달 받으면 st-1의 값을 n번 노드의 st~dt의 모든 배열에 더해줍니다. 0~마지막 노드까지 연산을 수행하면 prefix sum이 완성됩니다.

4-2번)

1. 구현,실행

두가지 mpi 프로그램을 하나의 파일안에 작성하였습니다. pb4\_distributed\_MPI\_Scan.c 프로그램에서 #define P2PCommunication을 1로 하면 MPI send,receive만으로 구현된 프로그램이 작동하고 0으로 하면 MPI\_Scan을 사용해서 합니다.



1. 성능 비교 결과

두개의 프로그램 모두 100000개(십만)의 랜덤 숫자에 대한 prefix sum을 구했습니다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 병렬 프로그램, mpisend,recv | | |  |  |
| Proc 0 > Elapsed time = 3.453559e-02 seconds | | | | |
| Proc 1 > Elapsed time = 3.323357e-02 seconds | | | | |
| Proc 2 > Elapsed time = 3.165801e-02 seconds | | | | |
| Proc 3 > Elapsed time = 2.680215e-02 seconds | | | | |
| Proc 4 > Elapsed time = 2.393832e-02 seconds | | | | |
| Proc 5 > Elapsed time = 2.448449e-02 seconds | | | | |
| Proc 6 > Elapsed time = 2.316351e-02 seconds | | | | |
| Proc 7 > Elapsed time = 2.108268e-02 seconds | | | | |
|  |  |  |  |  |
| 병렬 프로그램,mpi\_scan | | |  |  |
|  |  |  |  |  |
| Proc 0 > Elapsed time = 3.226133e-02 seconds | | | | |
| Proc 1 > Elapsed time = 3.052774e-02 seconds | | | | |
| Proc 2 > Elapsed time = 2.909413e-02 seconds | | | | |
| Proc 3 > Elapsed time = 2.764894e-02 seconds | | | | |
| Proc 4 > Elapsed time = 2.612802e-02 seconds | | | | |
| Proc 5 > Elapsed time = 2.464941e-02 seconds | | | | |
| Proc 6 > Elapsed time = 2.073900e-02 seconds | | | | |
| Proc 7 > Elapsed time = 1.919670e-02 seconds | | | | |

1. 성능 비교 분석

제가 send,recv로 구현한 방법은 n-1번째 노드와 n번째 노드가 통신이 수행되어야 n에서 n+1번째 노드로 데이터를 전송합니다. 그러므로 약 7번의 통신 오버헤드가 발생합니다.

하지만 Mpi\_scan은 트리 형태로 전파되는 구조를 가지므로 마스터노드가 3번만 통신하면 종료 되므로 속도가 더 빠를 것입니다.

기능은 동일하지만 알고리즘이 달라서 발생하는 차이입니다.

5-a번)

병렬, 직렬 성능 비교

1. 모든 large,small 파일을 hori하게 변환 했을때 병렬, 직렬 각각의 경우에 대한 평균 시간을 구했습니다. 이떄 병렬 프로그램의 노드의 갯수는 8개 입니다.
2. 각 병렬, 직렬에 대한 수행 시간 입니다.

|  |  |  |  |
| --- | --- | --- | --- |
| **열1** | **병렬** | **직렬** | **단위(s)** |
| large 합 | 86.9 | 1.78 |  |
| large 평균 | 7.9 | 0.161 |  |
| small 합 | 7.5 | 0.12 |  |
| small 평균 | 0.469 | 0.000748 |  |

병렬,직렬 성능 차이에 대한 논의

1. 병렬프로그램의 성능이 large프로그램에 있어서는 약 49배까지 성능이 차이납니다.

이는 communication시에 발생하는 오버헤드가 실제 분산 프로그래밍 계산에 비해 매우 크기 때문이라 추측할 수 있습니다.

병렬 상황에서 scailabilty에 따른 성능 차이결과

1. 노드가 2개인 경우 large 폴더를 수행하는데 총 시간은 10.3108초가 평균적으로 한개 프로그램당 1.03108초가 소요됬습니다. 이는 8개의 노드를 사용할때에 비해 1/8가량 입니다. 그리고 직렬 프로그램 보다는 9배가량 큽니다.

병렬 상황에서 scailabilty에 따른 성능 차이분석 (노드를 증가시킴에 따른 0~8)

1. 노드의 갯수가 적을수록 성능이 더 뛰어날 것으로 예상됩니다. 해당 프로그램에서의 병렬 프로그램의 오버헤드가 현저히 높기 때문입니다.

개인적으로 어려웠던점과 해결

1. open\_mpi 프로그래밍 자체가 처음이라 에러가 많이 발생 하였습니다. 가장 대표 적인 에러는 배열을 정적할당으로 시도한 점입니다. 정확한 이유는 모르지만 동적 할당을 해야만 open mpi 프로그램이 정상적으로 작동되었습니다.
2. file을 한개의 노드만이 한번에 접근 해야 했습니다. 공유 자원이라 동기화를 사용하는 방법도 있지만 구현시 rank =0 이 읽고 scatterv하는 방법을 사용했습니다.
3. ppm file 형식이 처음이라 이해하는데 시간이 걸렸습니다. 이후에도 a번을 구현하면서 대칭으로 이동시키려면 pixel 단위로 대칭 이동시키지 않고 byte단위로 이동시켜서 실수를 했습니다.
4. ppm file의 width의 길이가 width \* 3 만큼의 바이트라는 점 때문에 collective open mpi 명령어 사용에서 혼동이 있었습니다. 일단은 BYTE 단위로 맞춰서 전송하고 읽었습니다. 하지만 픽셀단위로 유저 mpi 타입 contiguous하게 생성했더라면 세로 대칭을 구현하는데 쉬웠을거 같습니다.(직렬 프로그래밍은 픽셀 단위로 이동시켰습니다.)

추가로 더 해보고 싶으나 시간상 못한 부분

1. mpi 타입을 생성해서 하기 (문제에 derived data type을 쓰라고 하긴함)
2. b,c번 추가로 구현
3. 마스터 노드가 scatterv하는 방식과, 각각의 노드들이 필요한 부분 만큼 파일에서 읽는 경우와 성능 비교.
4. 쉘 스크립트로 한번에 파일들을 실행해보고 싶다. 쉘 스크립트 작성 해보려다 시간이 모자라서 일일이 했습니다.
5. 커뮤니케이션 오버헤드랑, 병렬 프로그램 부분, 시리얼 프로그램 부분 각각에 대한 시간을 측정하면 더 정확한 분석이 가능했을 것 같습니다.

제출시 샘플 ppm파일들 제출,

사용한 linux ppm view : ImageMagick