병렬분산컴퓨팅 (CSE5414)

Fall 2020

Assignment #1

이름: 김인호

학번: 20161577

담당 교수: 박성용 교수님

20161577
김인호

병렬분산컴퓨팅 (CSE5414)

Assignment #1

1. Scalability

$$E = \frac{Speedup}{p} = \frac{T_{serial}}{p \cdot T_{parallel}} = \frac{n}{p(\frac{n}{p} + \log_2 p)} = \frac{n}{n + p \cdot \log_2 p}$$

Process 개수 p를 k배 증가시켰을 때 동일한 efficiency를 유지하기 위해 n은 다음 A배 만큼 증가시켜야 한다.

$$E = \frac{n}{n + p \cdot \log_2(p)} = \frac{A \cdot n}{A \cdot n + (k \cdot p) \cdot \log_2(k \cdot p)}$$

$$\Longrightarrow A = \frac{(k \cdot p) \cdot \log_2(k \cdot p)}{p \cdot \log_2(p)} = \frac{k \cdot \log_2(k \cdot p)}{\log_2(p)} = \frac{k \cdot (\log_2(k) + \log_2(p))}{\log_2(p)}$$

$$= \frac{k \cdot \log_2(k)}{\log_2(p)} + k$$

따라서 p를 8에서 16으로 2배 증가시킨다면 p=8, k=2를 대입하여 A=8/3인 것을 알 수 있다.

$$A = \frac{2 \cdot \log_2(2)}{\log_2(8)} + 2 = \frac{8}{3}$$

결론적으로 해당 문제는 process의 개수를 증가시켰을 때 problem size도 일정량 증가시켜서 동일한 efficiency를 유지할 수 있다. 즉 어떤 k값에 따라 efficiency를 유지시키는 A값을 찾을 수 있다는 것이다. 따라서 이론적으로 동일한 시간 내에 더 큰 problem size를 처리할 수 있기 때문에 weakly scalable 하다고 판단할 수 있다.

2. [MPI] Prefix sums

(1)

먼저 MPI_Scan()은 다음과 같은 parameter를 받는다.

```
int MPI_Scan(const void *sendbuf, void *recvbuf, int count,
    MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

입력 parameter:

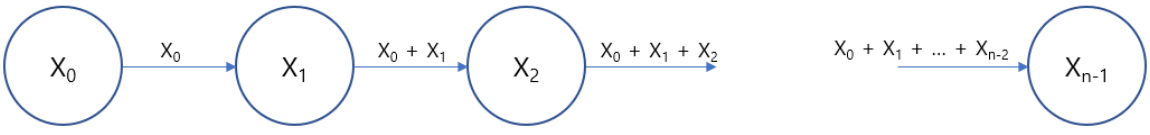
- sendbuf: 입력 buffer
- count: 입력 buffer에 있는 원소 개수
- datatype: 입력 buffer 원소의 type
- op: Scan 시 실행되는 operation
- comm: 통신 대상 communicator

출력 parameter:

- recvbuf: 출력 buffer

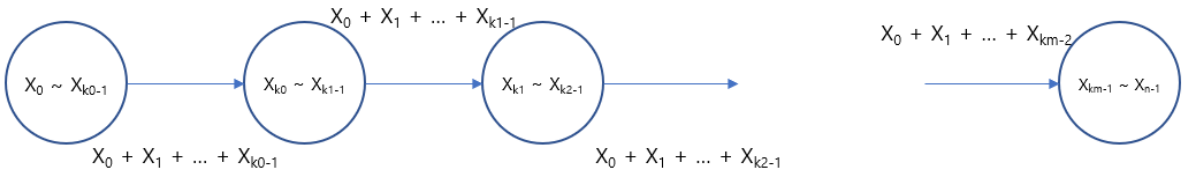
MPI_Scan()은 prefix reduction을 수행한다. 즉 MPI_Reduce()나 MPI_Alllreduce()처럼 모든 node에 대해서 한꺼번에 연산을 수행하는 것이 아니라 각 i 번째 node에 대해서 0 부터 i 번째 node까지에 대한 연산을 수행한다. 이를 이용해서 prefix sums, 즉 X_0 부터 X_{n-1} 까지의 n개의 숫자에 대한 X_0, $X_0 + X_1$, $X_0 + X_1 + X_2$, 등의 n개의 부분 합을 쉽게 구할 수 있다.

Prefix sums를 구하기 위해서 다음과 같은 병렬 방법을 구현할 수 있다.



n개의 node가 있고 각 i번째 node가 i번째 숫자를 갖고 있는 상태에서 i번째 node는 (i-1)번째 node로부터 0~(i-1)번째 숫자까지의 부분 합을 받고 이 부분 합에다가 i번째 숫자를 더하여 0~i번째 숫자까지의 부분 합을 구한다. 그리고 이를 i+1번째 node에 전송한다. 모든 전송이 끝나면 각 i번째 node는 0~i번째 숫자까지의 prefix sum을 저장하고 있다.

또 다른 방법은 다음과 같다.

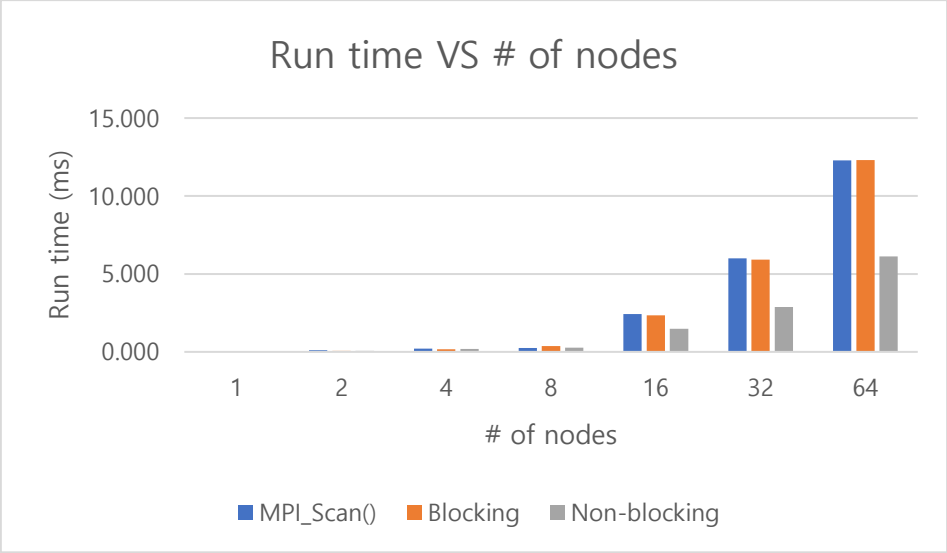


n개의 숫자에 대해 각각에 해당하는 n개의 node를 활용하는 것이 아닌 m개의 node에 n개의 숫자를 조각별로 나누어 처리하는 것이다. 따라서 각 node는 n/m개의 숫자를 저장하고 있고 n/m개의 부분 합을 구하게 되는 것이다. 위 방법과 비슷하게 i번째 node는 n/m개의 숫자 중 가장 마지막 숫자에 대한 부분 합을 구하고 이를 (i+1)번째 node에 보내어 진행한다.

(2)

위에서 언급한 방법 중 첫번째 방법을 blocking과 non-blocking send/recv를 이용하여 구현하였다. 아래는 N에 따른 각 구현 방법의 평균 실행 시간(ms)이다.

N	1	2	4	8	16	32	64
MPI_Scan()	0.013	0.098	0.200	0.253	2.420	6.002	12.286
Blocking	0.007	0.063	0.168	0.368	2.336	5.914	12.306
Non-blocking	0.012	0.068	0.187	0.270	1.489	2.883	6.123



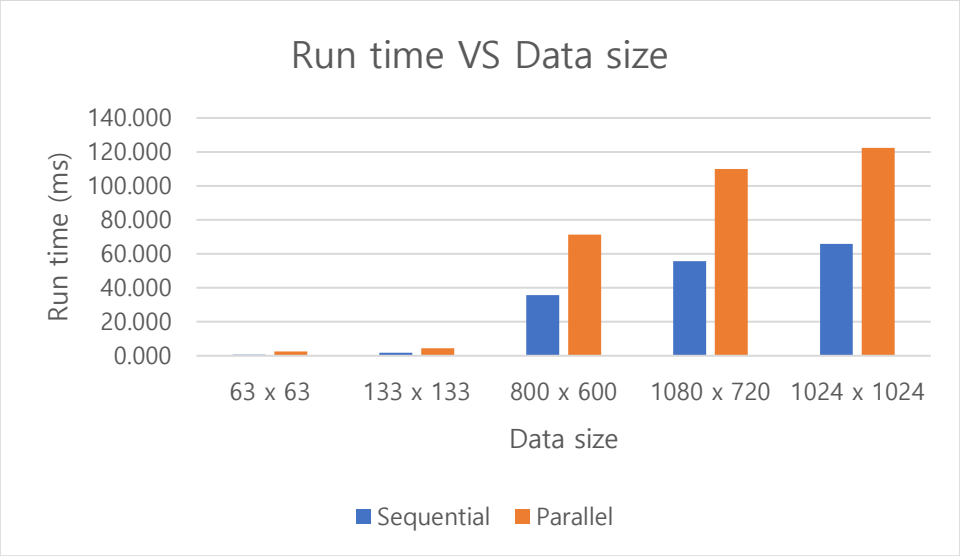
여기서 눈에 띄는 점은 MPI_Scan()과 blocking을 활용한 방법의 성능이 N과 무관하게 비슷하고 non-blocking 방법이 N=16부터 큰 차이로 더 빠르다는 것이다. MPI_Scan()도 blocking 통신을 이용하고 해당 구현 방법에서는 N이 곧 node의 개수이기 때문에 node가 많아질수록 blocking의 경우에는 기다려야 하는 경우가 더 많아질 수 있다는 것과 관련된 것을 알 수 있다.

3. [MPI] PPM processing

(1) Flip

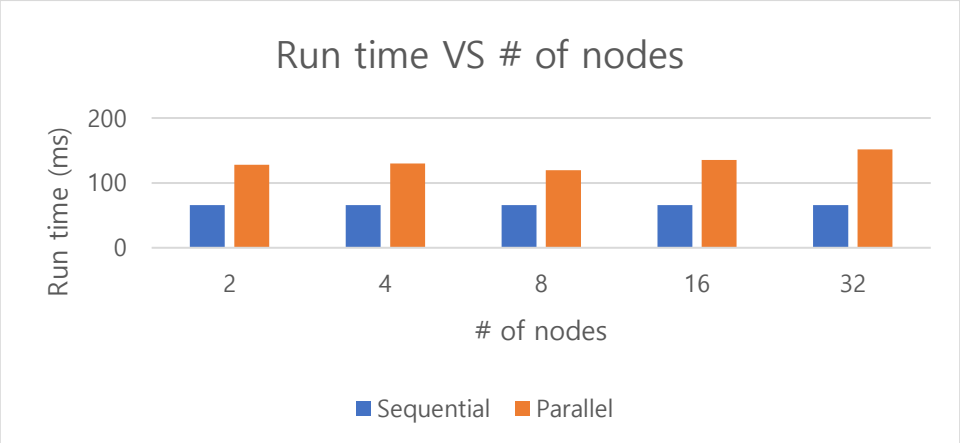
먼저 병렬 처리를 node 4개로 고정시키고 처리하는 PPM 파일의 크기를 바꿨을 때의 실행 시간(ms) 비교 결과이다.

	boxes_1	tree_1	cayuga_1	falls_1	Iggy
Size	**63 x 63**	**133 x 133**	**800 x 600**	**1080 x 720**	**1024 x 1024**
Sequential	0.611	1.855	35.647	55.720	65.801
Parallel	2.570	4.520	71.382	109.984	122.464



그리고 아래는 위의 1024 x 1024 크기의 파일을 node 개수를 바꾸며 flip 처리를 한 실행 시간의 비교 결과이다.

Nodes	**2**	**4**	**8**	**16**	**32**
Parallel	127.71	129.731	119.7565	135.572	151.506
Sequential	65.801	65.801	65.801	65.801	65.801

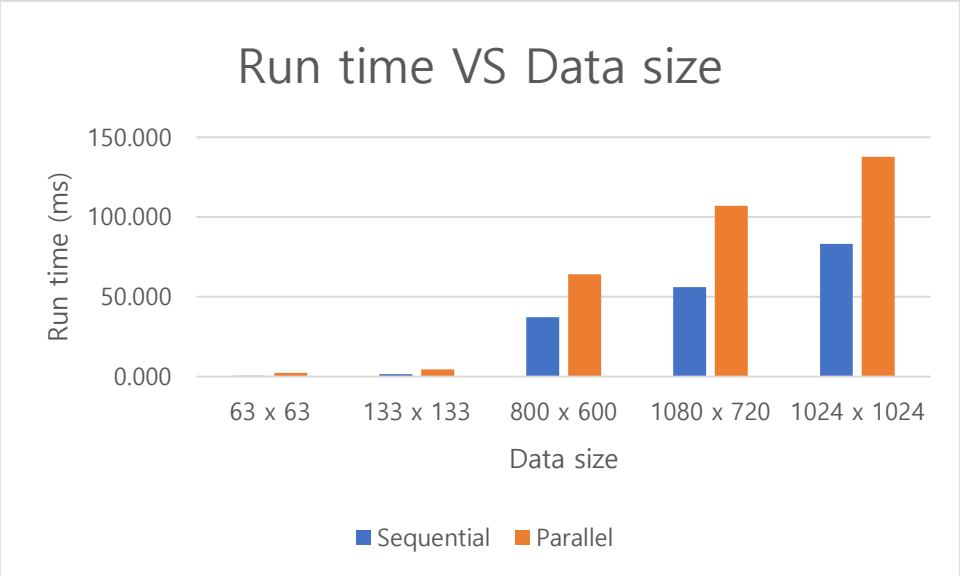


Flip 처리의 병렬 구현은 이미지를 row 단위로 여러 node에 뿌린 다음에 각 node가 각 row 별로 flip을 진행하는 방식이다. 하지만 어느 경우에도 parallel하게 처리하는 방법이 sequential하게 처리하는 방법보다 빠르지 않았다. 전반적으로 실행시간이 약 2배 이상으로 더 걸리는 것을 볼 수 있다. 이는 병렬로 처리하기 위한 overhead가 적지 않아서 오히려 sequential한 방법이 더 빠른 것이다. 따라서 이미지의 flip 처리는 병렬 방식으로 전혀 scalable하지 않다고 판단할 수 있다.

(2) Grayscale

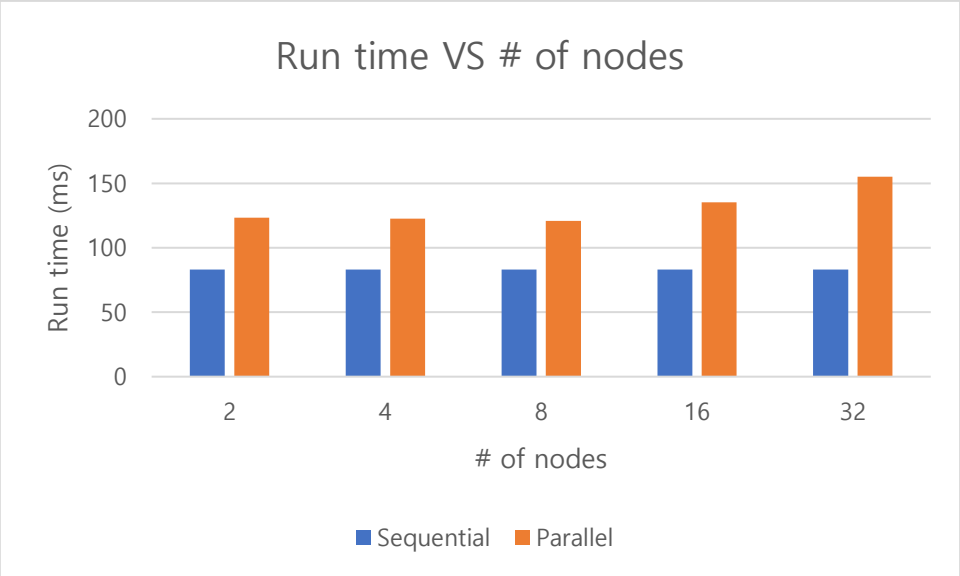
먼저 병렬 처리를 node 4개로 고정시키고 처리하는 PPM 파일의 크기를 바꿨을 때의 실행 시간(ms) 비교 결과이다.

	boxes_1	tree_1	cayuga_1	falls_1	Iggy
Size	**63 x 63**	**133 x 133**	**800 x 600**	**1080 x 720**	**1024 x 1024**
Sequential	0.591	1.462	37.211	56.126	83.198
Parallel	2.319	4.591	64.010	106.987	137.753



그리고 아래는 위의 1024 x 1024 크기의 파일을 node 개수를 바꾸며 흑백 처리를 한 실행 시간의 비교 결과이다.

Nodes	**2**	**4**	**8**	**16**	**32**
Sequential	83.198	83.198	83.198	83.198	83.198
Parallel	123.241	122.626	120.798	135.175	155.049

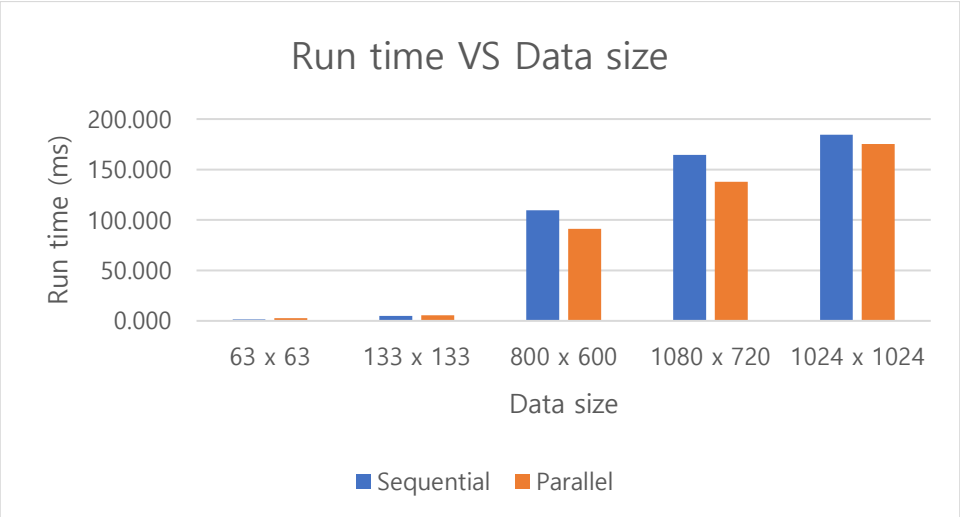


Grayscale 처리의 병렬 구현은 이미지를 row 단위로 여러 node에 뿌린 다음에 각 node가 각 pixel에 대하여 R, G, B의 평균을 구한 후 흑백 값을 저장하는 방식이다. Flip 경우와 마찬가지로 grayscale 처리도 parallel하게 처리하는 방법이 sequential하게 처리하는 방법보다 빠르지 않았다. 따라서 이미지의 grayscale 처리는 병렬 방식으로 전혀 scalable하지 않다고 판단할 수 있다.

(3) Smooth

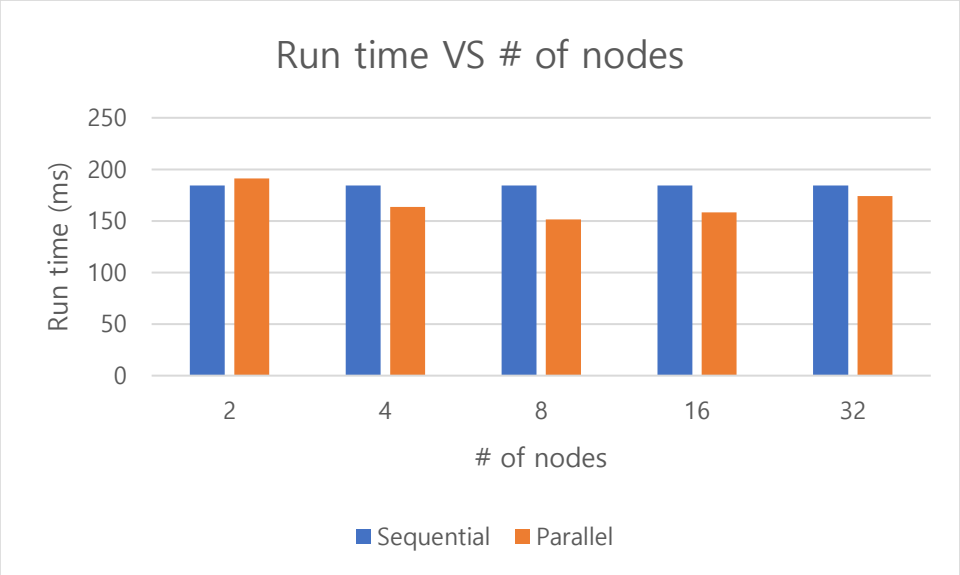
먼저 병렬 처리를 node 4개로 고정시키고 처리하는 PPM 파일의 크기를 바꿨을 때의 실행 시간 (ms) 비교 결과이다.

	boxes_1	tree_1	cayuga_1	falls_1	Iggy
Size	**63 x 63**	**133 x 133**	**800 x 600**	**1080 x 720**	**1024 x 1024**
Sequential	1.452	4.853	109.704	164.376	184.476
Parallel	2.669	5.404	91.215	137.792	175.409



그리고 아래는 위의 1024 x 1024 크기의 파일을 node 개수를 바꾸며 smooth 처리를 한 실행 시간의 비교 결과이다.

Nodes	2	4	8	16	32
Sequential	184.476	184.476	184.476	184.476	184.476
Parallel	191.324	163.426	151.3695	158.258	173.992



Smooth의 구현은 각 pixel마다 상, 하, 좌, 우 그리고 4개의 대각선 방향으로의 평균값을 저장하는 것이고 주변 pixel이 없을 경우 (이미지 가장자리에 위치한 pixel) 평균 계산에 포함하지 않았다. 병렬의 경우 이미지를 연속된 row 단위로 각 node에 나눈 뒤 각 node는 해당 연속된 row의 가장자리에 위치한 pixel을 제외한 나머지 pixel의 smooth 처리를 한다. 이를 다 합친 후 root node가 각 연속된 row의 가장자리에 있던 pixel들의 smooth 처리를 진행한다.

먼저 눈에 띄는 점은 data의 크기를 늘렸을 때 sequential과 parallel의 처리 시간의 격차가 줄어든다는 것이다. 즉 data의 크기가 어느 정도 크다면 parallel하게 처리하는 방식이 더 빨라질 수 있다는 것을 예측할 수 있다. 또한 data 크기를 고정시키고 node의 개수를 증가시켰을 때도 격차가 점점 좁아지는 것을 볼 수 있고 심지어 node의 개수가 2개일 때는 병렬 처리를 위한 overhead가 가장 적기 때문에 sequential한 처리보다 빨랐다는 것을 볼 수 있다. 결론적으로 data의 크기를 계속 늘리거나 node의 개수를 계속 늘리게 되면 parallel 방식의 처리가 더 빠를 것으로 판단할 수 있고 어느 정도의 data 크기나 node 개수를 달성했을 때 scalable한 효과를 보일 수 있을 것으로 예상된다. PPM 파일을 확인하기 위해서는 default 이미지 뷰어를 활용하였다.

4. [RPC] Calculator

Server 프로그램은 주어진 두개의 피연산자로 덧셈 (그리고 뺄셈), 곱셈, 나눗셈, n제곱의 연산을 수행하는 역할을 한다. 따라서 각 연산에 해당하는 함수를 contract에 작성하여 rpcgen으로 client와 server 각각에 해당하는 program과 stub을 생성하였다.

Client 프로그램은 사용자에게로부터 입력을 받고 주어진 expression에 대해 valid한지 체크를 하면서 연산자 우선 순위("**" > "*", "/" > "+", "-")에 따라 알맞는 RPC 호출을 수행한다. 구현의 복잡도를 줄이기 위해 모든 동일한 연산자 우선 순위에 대하여 left associativity(왼쪽에 먼저 등장하는 연산자 우선)를 사용하였다. 즉 6/3*2=4이고 2**3**2=64 이다. 이를 구현하기 위해서 연산자 stack과 피연산자 stack을 활용하였고 내부적으로 "**" 연산자의 경우 "^" 글자로 바꿔서 처리하였다.