

Lecture 10

Loop Carried Dependences

이재진

서울대학교 컴퓨터공학부

<http://aces.snu.ac.kr>



THUNDER Research Group
Seoul National University
서울대학교 천둥 연구실



Anti Dependence and Output Dependence

- False dependence로 불림
 - 실제 연산의 실행 순서와 관련되어 있는 것이 아니라 메모리 위치의 재사용에 의해 일어나는 디펜던스
- 변수 재명명(variable renaming)으로 제거할 수 있음
 - Flow dependence: $S1 \rightarrow S2, S3 \rightarrow S4$
 - Anti dependence: $S2 \rightarrow S3$

```
S1: t = a + b
S2: sum1 = t + s1
S3: t = c + d
S4: sum2 = t + s2
```

```
S1: t1 = a + b
S2: sum1 = t1 + s1
S3: t2 = c + d
S4: sum2 = t2 + s2
```



Loop-Independent Dependences

- S1 and S2 both reference the same location on the same loop iteration, but with S1 preceding S2 during execution of the loop iteration

```
for (i=0; i<N; i++) {  
    A[i] = B[i];  
    F[i+1] = A[i];  
}
```



Loop-carried Dependences

- S1 references a location on one iteration of a loop
- On subsequent iteration, S2 references the same location

```
sum = 0;
for (i=0; i<N; i++) {
    sum = sum + A[i];
}
```

```
sum = sum + A[0];
sum = sum + A[1];
sum = sum + A[2];
sum = sum + A[3];
...
```

```
for (i=0; i<N; i++) {
    A[i+1] = F[i];
    F[i+1] = A[i];
}
```

```
A[1] = F[0];
F[1] = A[0];
A[2] = F[1];
F[2] = A[1];
...
```



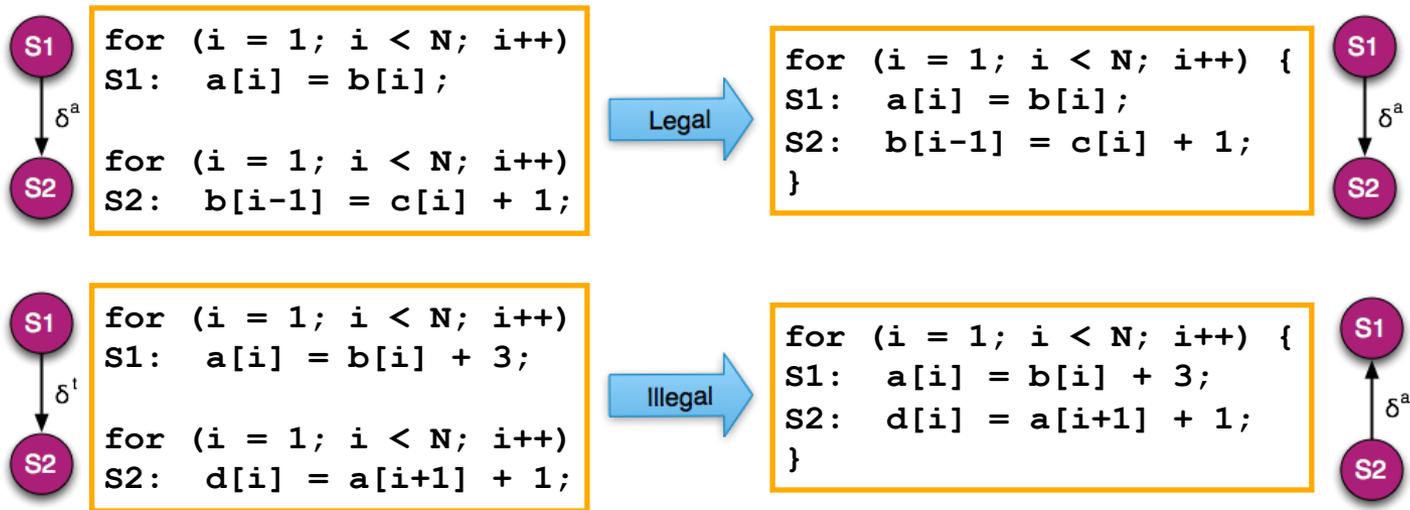
Fundamental Theorem of Dependence

- Any reordering transformation that preserves every dependence in a program preserves the meaning of that program



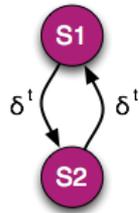
Loop Fusion

- Takes two adjacent loops that have the same iteration space traversal and combines their bodies into a single loop
- Fusion is illegal if fusing two loops causes the dependence direction to be changed

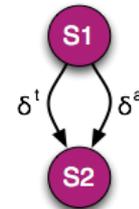


Loop Distribution (Fission)

- Takes a loop that contains multiple statements and splits it into two loops with the same iteration-space traversal
- Can be used to convert a sequential loop to multiple parallel loops
 - Converts loop-carried dependences to loop-independent dependences
- Legal when it does not result in breaking any cycles in the dependence graph of the original loop



```
for (i = 1; i < N; i++) {
S1:  a[i] = b[i] + 3;
S2:  b[i-1] = a[i+1] + 1;
}
```

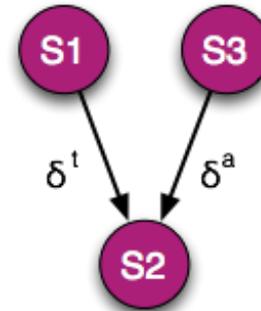


```
for (i = 1; i < N; i++)
S1:  a[i] = b[i] + 3;

for (i = 1; i < N; i++)
S2:  b[i-1] = a[i+1] + 1;
```



Loop Distribution (cont'd)



```
for (i = 1; i < N; i++) {  
S1:  a[i] = b[i] + 3;  
S2:  c[i] = a[i] + 1;  
S3:  d[i] = c[i+1];  
}
```



```
for (i = 1; i < N; i++)  
S1:  a[i] = b[i] + 3;  
  
for (i = 1; i < N; i++)  
S3:  d[i] = c[i+1];  
  
for (i = 1; i < N; i++)  
S2:  c[i] = a[i] + 1;
```



Reduction

- 어떤 연산을 이용해 여러 개의 값을 모아서 하나의 값을 생성하는 과정
- 대표적인 병렬 컴퓨팅 패턴
- $+$, $*$, \min , \max 등의 연산에 대하여 정의됨
 - 교환 법칙, 결합 법칙이 성립하고 항등원을 가지는 연산
 - 항등원 : 연산이 정의된 집합에 대하여 임의의 원소 a 와 항등원을 연산한 결과는 항상 a

```
sum = 0;  
for(i = 0; i < 8; i = i+1){  
    sum = sum + x[i];  
}
```



Reduction (cont'd)

- 루프 캐리드 디펜던스 때문에 for 루프를 순차적으로 실행할 수 밖에 없음
- N 개의 데이터 아이템이 있다면 모두 N 번의 덧셈 연산이 필요함
 - N 번의 순차적인 덧셈 스텝이 필요

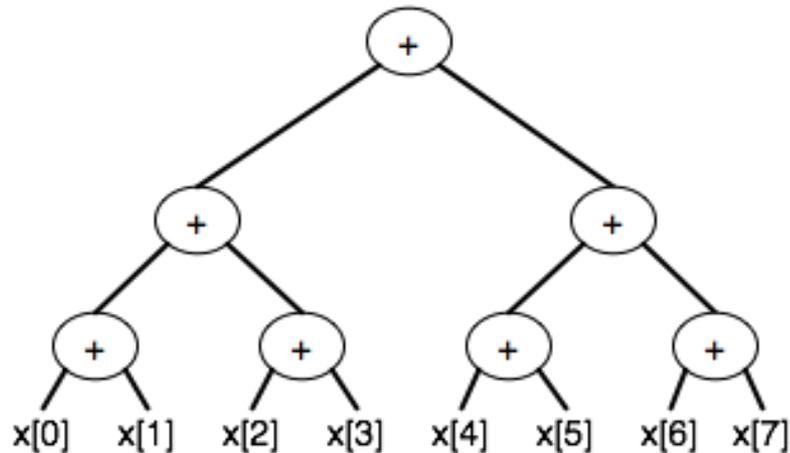
```
sum = 0;  
for(i = 0; i < 8; i = i+1){  
    sum = sum + x[i];  
}
```



Parallel Reduction

- Parallel reduction을 적용하면 $\log N$ 번의 덧셈 스텝만 필요

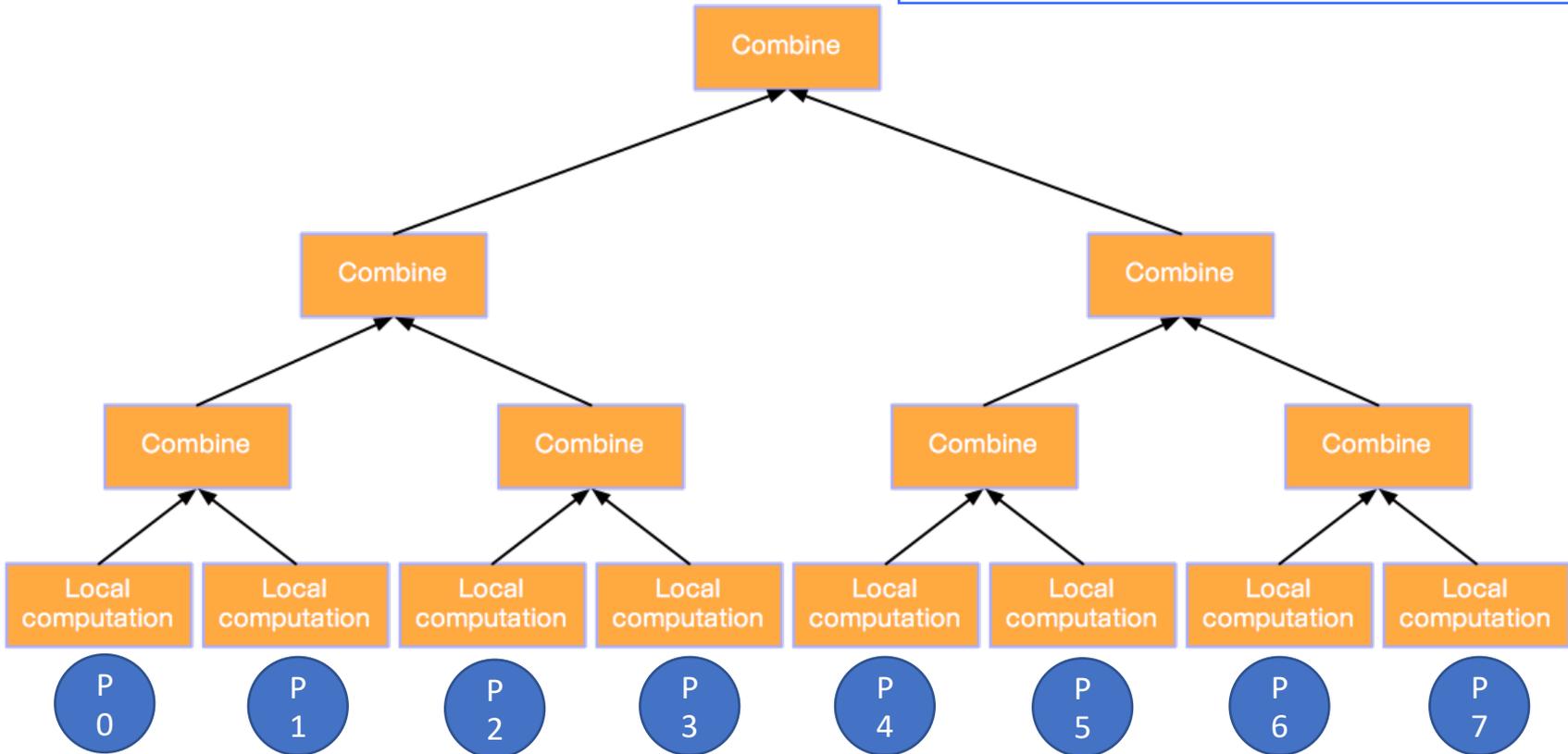
$$\text{sum} = ((x[0] + x[1]) + (x[2] + x[3])) \\ + ((x[4] + x[5]) + (x[6] + x[7]))$$



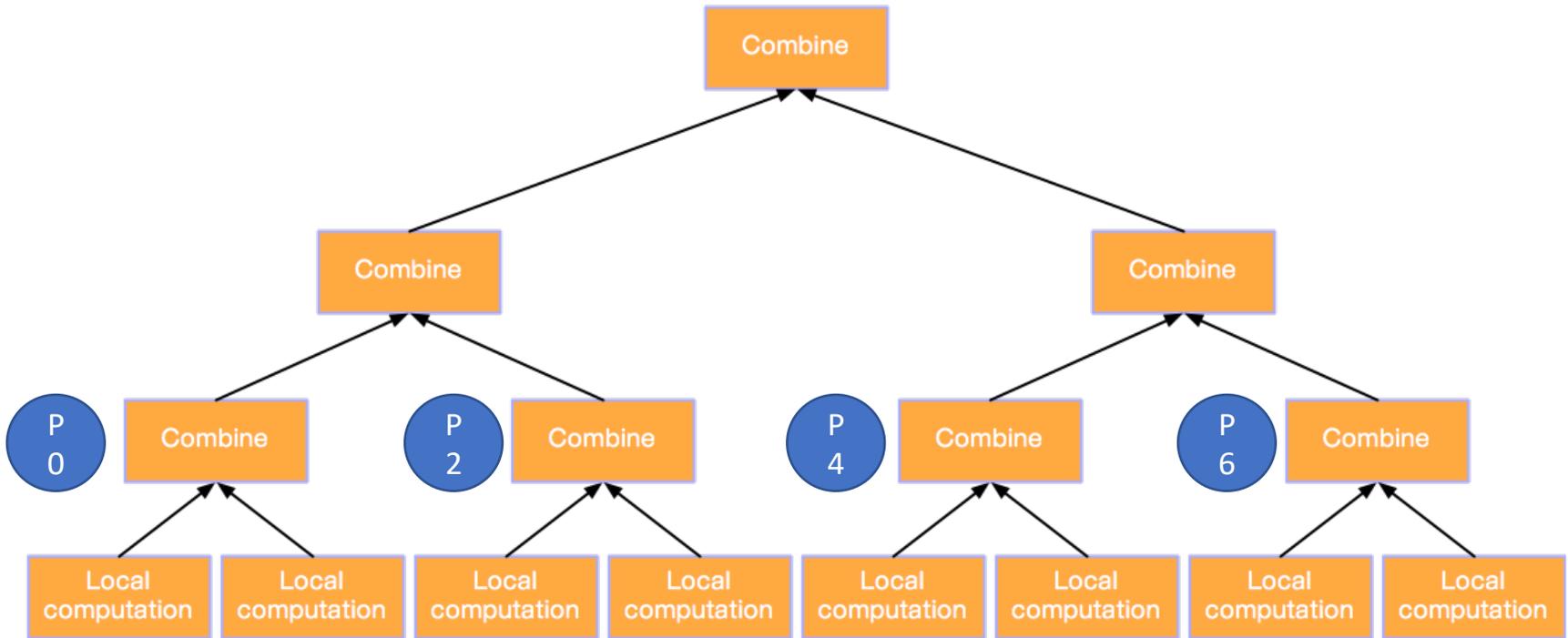
Parallel Reduction의 구조

- Tree 구조

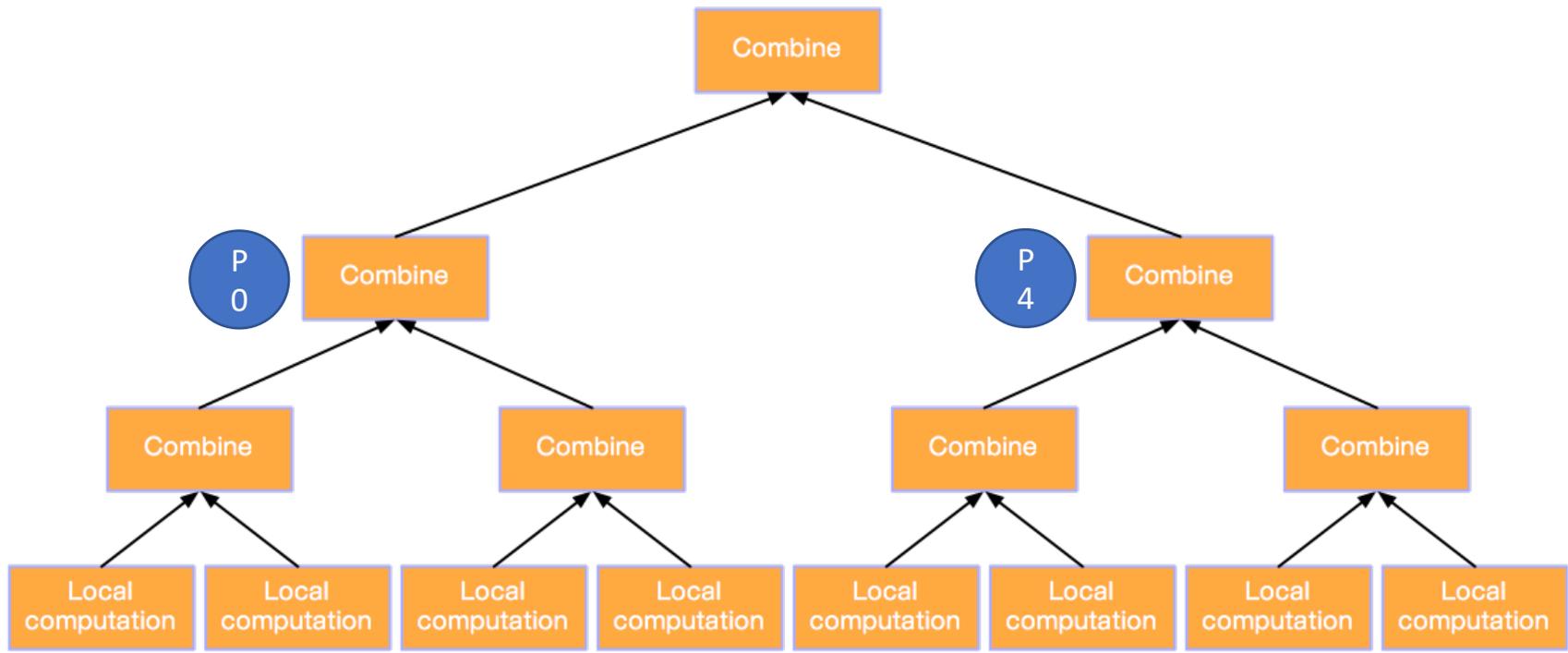
```
sum = 0;  
for(i = 0; i < 8; i = i+1){  
    sum = sum + f(x[i]);  
}
```



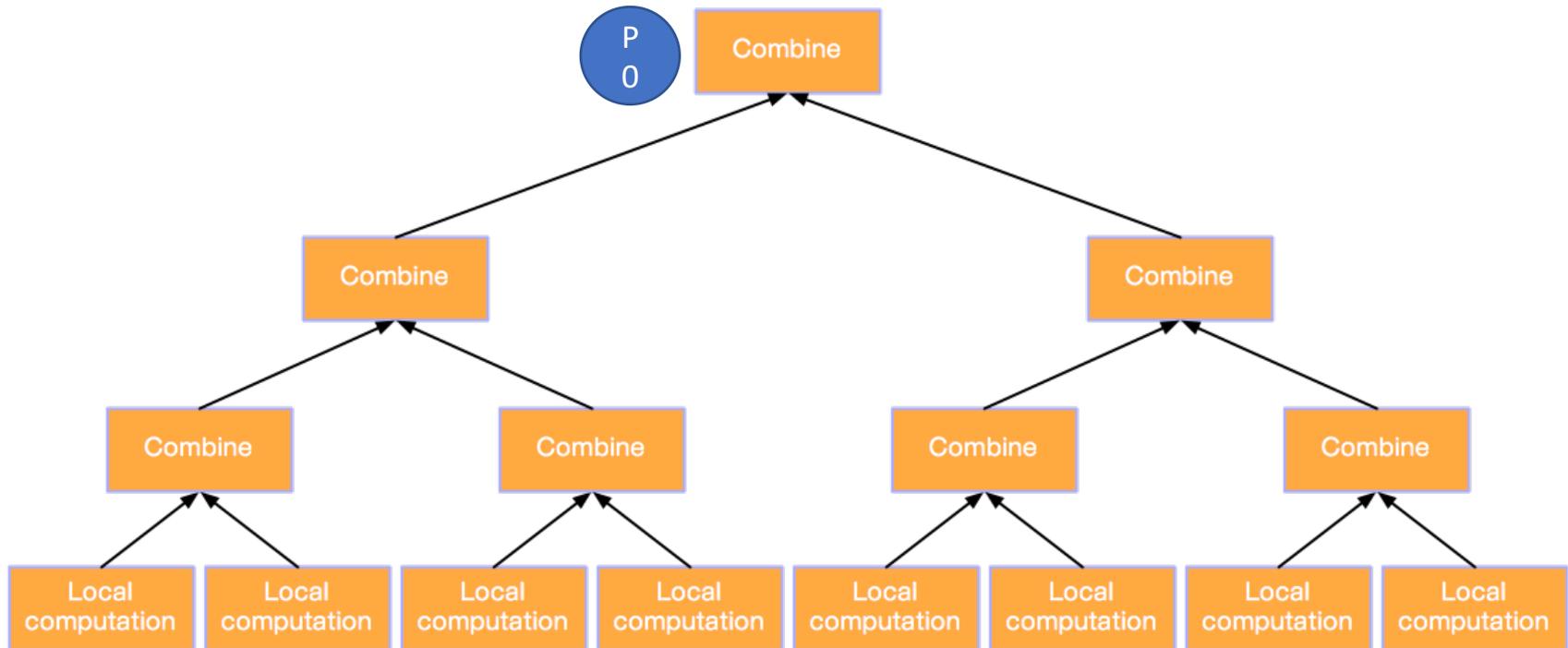
Parallel Reduction의 구조 (cont'd)



Parallel Reduction의 구조 (cont'd)



Parallel Reduction의 구조 (cont'd)



Scan

- 주어진 연산의 결과로 얻는 열의 위치 i 에 있는 원소는, 원래 열의 위치 0 부터 위치 i 까지 위치한 원소들에 주어진 연산을 적용하여 얻음
 - 연산에 대하여 결합법칙이 성립해야 함
- 대표적인 병렬 컴퓨팅 패턴
- 예) prefix sum

$$a_0, a_1, a_2, \dots, a_{n-1}$$

$$a_0, a_0+a_1, a_0+a_1+a_2, \dots, a_0+a_1+\dots+a_{n-1}$$



Prefix Sum

- 루프 캐리드 디펜던스 때문에 for 루프를 순차적으로 실행할 수 밖에 없음
- N 개의 데이터 아이템이 있다면 모두 $N - 1$ 번의 덧셈 연산이 필요함

$a_0, a_1, a_2, \dots, a_{n-1}$

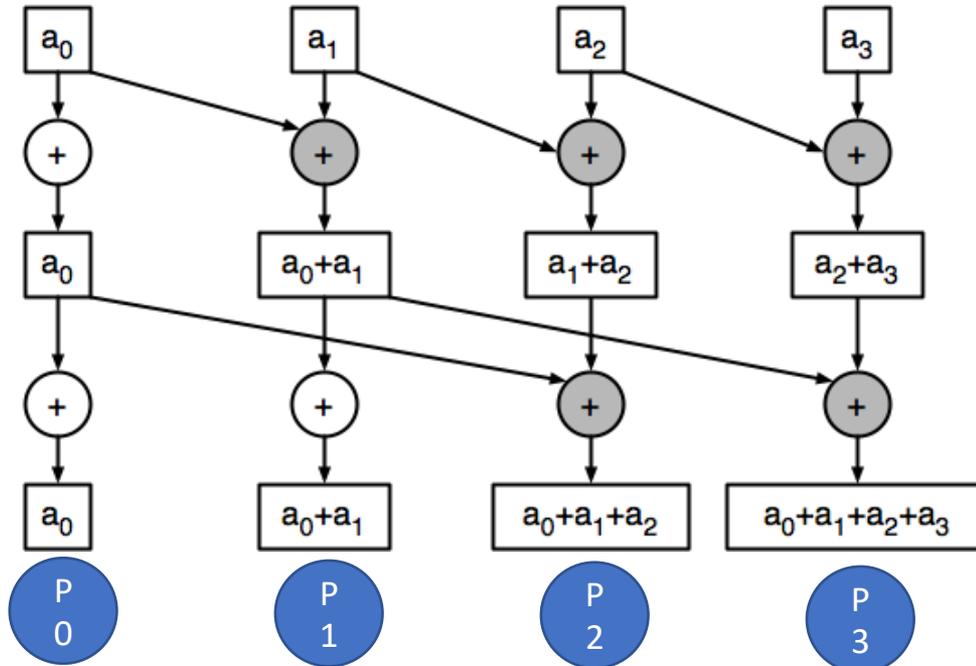
$a_0, a_0+a_1, a_0+a_1+a_2, \dots, a_0+a_1+\dots+a_{n-1}$

```
prefix_sum[0] = a[0];  
for(i = 1; i < 8; i = i+1){  
    prefix_sum[i] = prefix_sum[i-1] + a[i];  
}
```



Parallel Prefix Sum

- Parallel scan을 적용하면 $\log N$ 번의 덧셈 스텝만 필요
 - 연산의 순서를 바꾸는 것



Parallel Scan의 응용

- 다항식의 계산
- 점화식의 계산
- Sorting
- Histogram
- ...

