

Chapter 4

Data-Level Parallelism in Vector, SIMD, and GPU Architectures.

We will cover sections 4.1, 4.2, 4.3,
and 4.5 and GPUs (section 4.5)

Flynn's Taxonomy



<i>classic von Neumann</i> SISD Single instruction stream Single data stream	(SIMD) Single instruction stream Multiple data stream
MISD Multiple instruction stream Single data stream <i>Does it make sense? Yes, systolic array.</i>	(MIMD) Multiple instruction stream Multiple data stream

Data Parallelism



- Concurrency arises from performing the **same operations on different pieces of data**
 - Single instruction multiple data (SIMD)
 - E.g., dot product of two vectors
- Contrast with data flow
 - Concurrency arises from executing different operations in parallel (in a data driven manner)
- Contrast with thread (“control”) parallelism
 - Concurrency arises from executing different threads of control in parallel

3

Introduction



- SIMD architectures can exploit significant data-level parallelism for:
 - matrix-oriented scientific computing
 - media-oriented image and sound processors
- SIMD is more energy efficient than MIMD
 - Only needs to fetch one instruction per data operation
 - Makes SIMD attractive for personal mobile devices
- SIMD allows programmer to continue to think sequentially

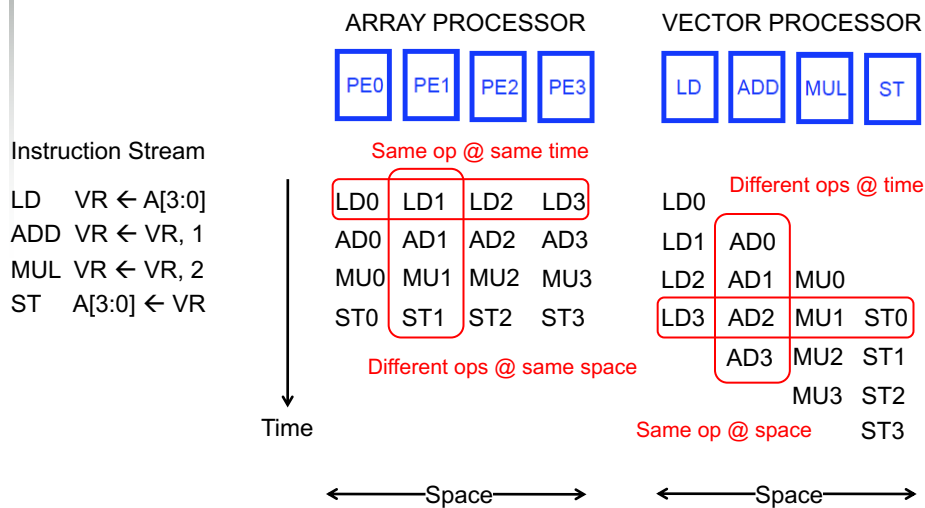
4

SIMD Processing

- Single instruction operates on multiple data elements
 - In time or in space
- Multiple processing elements
- Time-space duality
 - Array processor:** Instruction operates on multiple data elements at the same time
 - Vector processor:** Instruction operates on multiple data elements in consecutive time steps

5

Array vs. Vector Processors



6

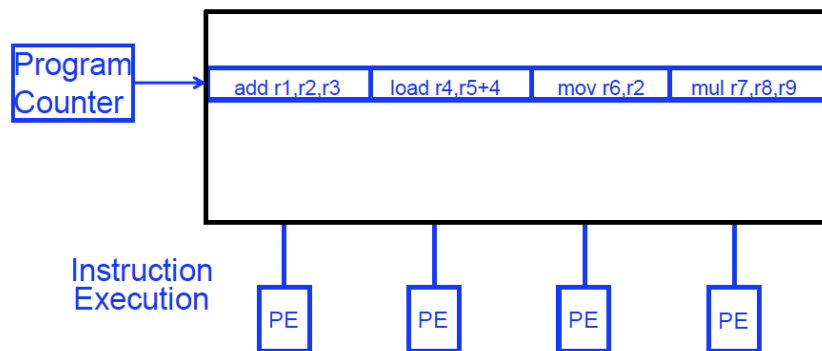
SIMD Parallelism

- Array and vector architectures
- SIMD extensions
- Graphics Processor Units (GPUs)
- For x86 processors:
 - Expect two additional cores per chip per year
 - SIMD width to double every four years
 - Potential speedup from SIMD to be twice that from MIMD!

7

SIMD Array Processing vs. VLIW

- VLIW (Very Large Instruction Word)

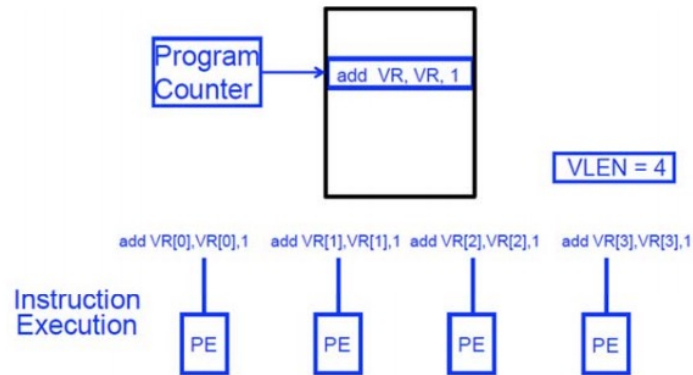


8

SIMD Array Processing vs. VLIW



- VLIW (Very Large Instruction Word)



9

Vector Processors



- A vector is a one-dimensional array of numbers
- Many scientific/commercial programs use vectors
for (i = 0; i <= 49; i++)
 C[i] = (A[i] + B[i]) / 2
- A vector processor is one whose instructions operate on vectors rather than scalar (single data) values
- Basic requirements
 - Need to load/store vectors → vector registers (contain vectors)
 - Need to operate on vectors of different lengths → vector length register (VLEN)
 - Elements of a vector might be stored apart from each other in memory → vector stride register (VSTR)
 - Stride: distance between two elements of a vector

10

Vector Processors (II)



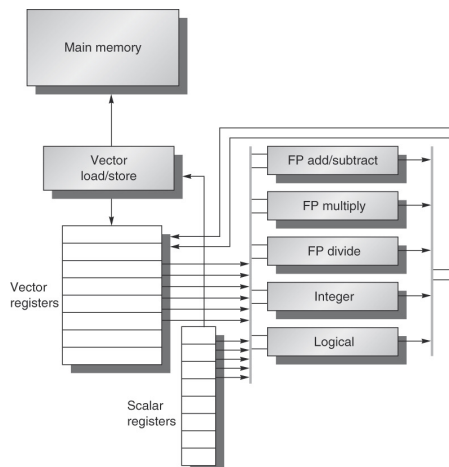
- A vector instruction performs an operation on each element in consecutive cycles
 - Vector functional units are pipelined
 - Each pipeline stage operates on a different data element
- Vector instructions allow deeper pipelines
 - No intra-vector dependencies → no hardware interlocking within a vector
 - No control flow within a vector
 - Known stride allows prefetching of vectors into cache/memory

11

Vector Architectures

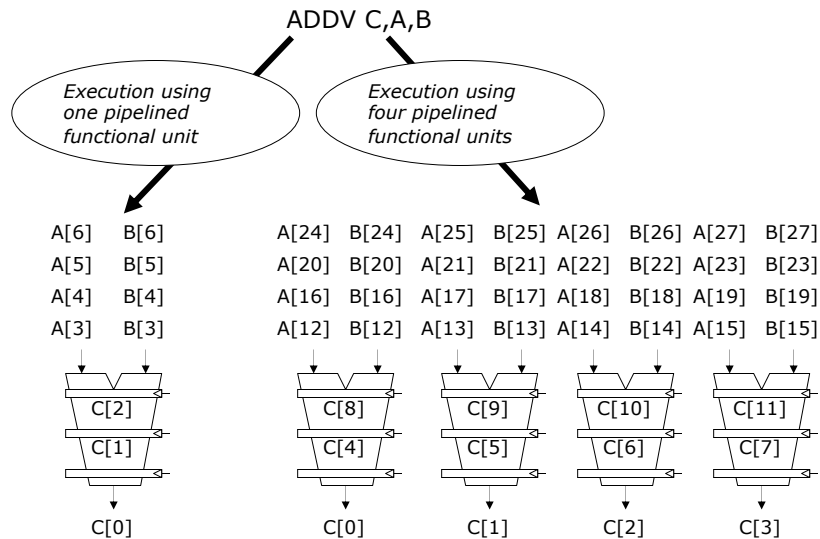


- Basic idea:
 - Read sets of data elements into “vector registers”
 - Operate on those registers
 - Disperse the results back into memory
- Registers are controlled by compiler
 - Used to hide memory latency
 - Leverage memory bandwidth



12

Vector Instruction Execution



Slide credit: Krste Asanovic

13

Vector Processor Advantages



- + No dependencies within a vector
 - Pipelining, parallelization work well
 - Can have very deep pipelines, no dependencies!
- + Each instruction generates a lot of work
 - Reduces instruction fetch bandwidth
- + Highly regular memory access pattern
 - Interleaving multiple banks for higher memory bandwidth
 - Prefetching
- + No need to explicitly code loops
 - Fewer branches in the instruction sequence

14

Vector Processor Disadvantages



- Works (only) if parallelism is regular (data/SIMD parallelism)

- ++ Vector operations

- Very inefficient if parallelism is irregular

- How about searching for a key in a linked list?

To program a vector machine, the compiler or hand coder must make the data structures in the code fit nearly exactly the regular structure built into the hardware. That's hard to do in first place, and just as hard to change. One tweak, and the low-level code has to be rewritten by a very smart and dedicated programmer who knows the hardware and often the subtleties of the application area. Often the rewriting is

Fisher, "Very Long Instruction Word architectures and the ELI-512," ISCA 1983. 15

Vector Processor Limitations



- Memory (bandwidth) can easily become a bottleneck, especially if

1. compute/memory operation balance is not maintained
2. data is not mapped appropriately to memory banks

RV64V



- Example architecture: RV64V
 - Loosely based on Cray-1
 - Vector registers
 - Each register holds a vector (32 elements, each 64 bits long)
 - Register file has 16 read ports and 8 write ports
 - Vector functional units
 - Fully pipelined
 - Data and control hazards are detected
 - Vector load-store unit
 - Fully pipelined
 - One word per clock cycle after initial latency
 - Scalar registers
 - 32 general-purpose registers
 - 32 floating-point registers

17

RV64V Instructions



- vadd.vv: add two vectors
- vadd.sv: add a scalar to a vector
- vld, vst: vector load and vector store from address
- Example: DAXPY

fld	F0,a	; load scalar a
vld	V1,Rx	; load vector X
vmul.vs	V2,V1,F0	; vector-scalar multiply
vld	V3,Ry	; load vector Y
vadd.vv	V4,V2,V3	; add
vst	V4,Ry	; store the result
- Requires 6 instructions vs. almost 250 for RISC V

18

Vector Execution Time



- Execution time depends on three factors:
 - Length of operand vectors
 - Structural hazards
 - Data dependencies
- RV64V functional units consume one element per clock cycle
 - Execution time is approximately the vector length
- *Convoy*
 - Set of vector instructions that could potentially execute together

19

Chaining



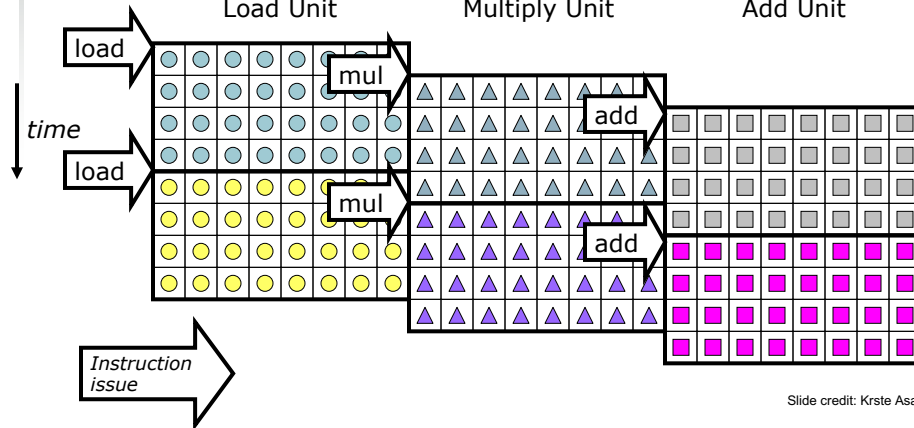
- Sequences with read-after-write dependency hazards can be in the same convey via *chaining*
- *Chaining (within a convoy)*
 - Allows a vector operation to start as soon as the individual elements of its vector source operand become available

20

Vector Instruction Level Parallelism

Can overlap execution of multiple vector instructions

- example machine has 32 elements per vector register and 8 lanes
- Complete 24 operations/cycle while issuing 1 short instruction/cycle



Slide credit: Krste Asanovic

23

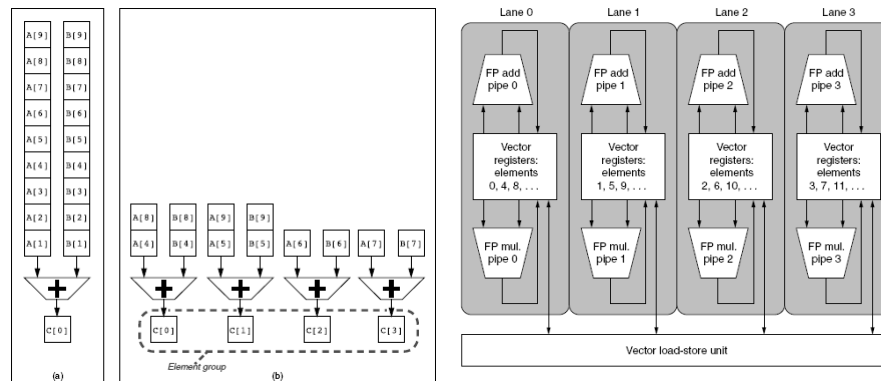
Challenges

- Start up time
 - Latency of vector functional unit
 - Assume the same as Cray-1
 - Floating-point add => 6 clock cycles
 - Floating-point multiply => 7 clock cycles
 - Floating-point divide => 20 clock cycles
 - Vector load => 12 clock cycles
- Improvements:
 - > 1 element per clock cycle
 - Non-32 long vectors
 - IF statements in vector code
 - Memory system optimizations to support vector processors
 - Multiple dimensional matrices
 - Sparse matrices
 - Programming a vector computer

24

Multiple Lanes

- Element n of vector register A is “hardwired” to element n of vector register B
 - Allows for multiple hardware lanes



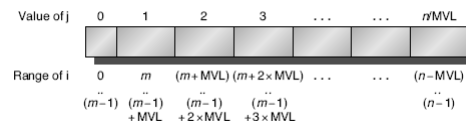
25

Vector Length Register

- Vector length not known at compile time?
- Use Vector Length Register (VLR)
- Use strip mining for vectors over the maximum vector length (MVL):

```
low = 0;
VL = (n % MVL); /*find odd-size piece using modulo op % */
for (j = 0; j <= (n/MVL); j=j+1) { /*outer loop*/
    for (i = low; i < (low+VL); i=i+1) /*runs for length VL*/
        Y[i] = a * X[i] + Y[i]; /*main operation*/
    low = low + VL; /*start of next vector*/
    VL = MVL; /*reset the length to maximum vector length*/
}
```

Vector operation



26

Vector Mask Registers



- Consider the loop:
for (i = 0; i < 32; i=i+1)
 if (X[i] != 0)
 X[i] = X[i] – Y[i];
- Use vector mask register, VM, to “disable” elements:

vsetpcfgi	1	;Enable 1 predicate register
vld	v0,Rx	;load vector X into V0
vld	v1,Ry	;load vector Y
fld	F0,#0	;load FP zero into F0
vpne	p0, v0, F0	;sets p0(i) to 1 if v0(i) != F0
vsub.vv	v0, v0, v1	;subtract under vector mask
vst	v0,Rx	;store the result in X
vdisable		;disable predicate registers
- GFLOPS rate decreases!

27

Memory Banks



- Memory system must be designed to support high bandwidth for vector loads and stores
- Spread accesses across multiple banks
 - Control bank addresses independently
 - Load or store non sequential words
 - Support multiple vector processors sharing the same memory
- Bank-level parallelism helps.

28

Stride

- Consider:


```

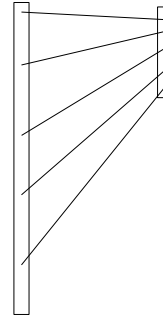
for (i = 0; i < 32; i=i+1)
  for (j = 0; j < 32; j=j+1) {
    A[i][j] = 0.0;
    for (k = 0; k < 32; k=k+1)
      A[i][j] = A[i][j] + B[i][k] * D[k][j];
  }
      
```

The matrix D



D is stored row-wise

Loading with stride



- Must vectorize multiplication of rows of B with columns of D
- Use *vlds* (vector load with stride) to load the columns of a matrix into a vector.
 - $\text{vlds } v, (R1, R2) \rightarrow v[i] = \text{address}(R1 + i \cdot R2)$
- Bank conflict (stall) occurs when the same bank is hit faster than bank busy time:
 - $\# \text{banks} / \text{LCM}(\text{stride}, \# \text{banks}) < \text{bank busy time}$

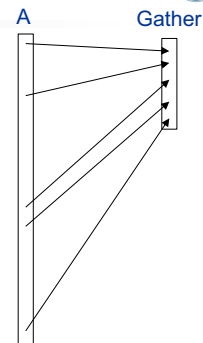
30

Scatter-Gather

- Consider:


```

for (i = 0; i < n; i=i+1)
  A[K[i]] = A[K[i]] + C[M[i]];
      
```
- Use *vldx* (load using index vector), and *vstx*:
 - $\text{vldx } v1, (R1, v2) \rightarrow v1[i] = \text{address}(R1 + v2[i])$



vld	Vk, Rk	;load K
vldx	Va, (Ra,Vk)	;load A[Ra+K[]] -- Gather
vld	Vm, Rm	;load M
vldx	Vc, (Rc,Vm)	;load C[Rc+M[]]
vadd.vv	Va, Va, Vc	;add them
vstx	Va, (Ra,Vk)	;store A[Ra+K[]] -- Scatter

31

SIMD Extensions (section 4.3)



- Limitations, compared to vector instructions:
 - Number of data operands encoded into op code
 - No sophisticated addressing modes (strided, scatter-gather)

32

SIMD Implementations



- Implementations:
 - Intel MMX (1996)
 - Eight 8-bit integer ops or four 16-bit integer ops
 - Streaming SIMD Extensions (SSE) (1999)
 - Eight 16-bit integer ops
 - Four 32-bit integer/fp ops or
 - two 64-bit integer/fp ops
 - Advanced Vector Extensions (AVX) (2010)
 - Four 64-bit integer/fp ops
 - AVX-512 (2013)
 - Operands must be consecutive and aligned memory locations

33

Example SIMD Code

- Assume 64-bit architecture (8 bytes) and 4-SIMD operations
- Example $Y[i] = Y[i] + a \cdot x[i]$, $i=1, \dots, 32$:

fld	F0,a	;load scalar a
splat.4D	F0, F0	;make 4 copies of F0 in F0, F1, F2 and F3
addi	R1,Rx,#256	;last address to load (32 * 8)
L: fld.4D	F4,0[Rx]	;load X[i], X[i+1], X[i+2], X[i+3] to F4, F5, F6 and F7
fmul.4D	F4,F4,F0	;a×X[i],a×X[i+1],a×X[i+2],a×X[i+3]
fld.4D	F8,0[Ry]	;load Y[i], Y[i+1], Y[i+2], Y[i+3] to F8, F9, F10 and F11
fadd.4D	F8,F8,F4	;a×X[i]+Y[i], ..., a×X[i+3]+Y[i+3]
fst.4D	0[Ry],F8	;store into Y[i], Y[i+1], Y[i+2], Y[i+3]
DADDIU	Rx,Rx,#32	;increment index to X (by 32 bytes – 4 words)
DADDIU	Ry,Ry,#32	;increment index to Y (by 32 bytes – 4 words)
bne	Rx, R1, L	;will iterate 8 times, 4 Y's computed in each iteration

34

Loop-Level Parallelism (Sec. 4.5)

- Focuses on determining whether data accesses in later iterations are dependent on data values produced in earlier iterations
 - Loop-carried dependence

Example 1:

for (i=0 ; i<1000 ; i++)

$x[i] = x[i] + s$; // No loop-carried dependence

for (i=2 ; i<1000 ; i++)

$x[i] = x[i-1] + x[i-2]$; // loop-carried dependence

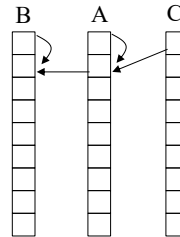
35

Loop-Level Parallelism



■ Example 2:

```
for (i=0; i<100; i=i+1) {
    A[i+1] = A[i] + C[i];    /* S1 */
    B[i+1] = B[i] + A[i+1]; /* S2 */
}
```



- S1 and S2 use values computed in previous iteration (bad for parallelism)
- S2 uses value computed by S1 in same iteration
- Iterations i , $i+1$, $i+2$, ... of the loop cannot be executed in parallel (non-parallelisable loop).

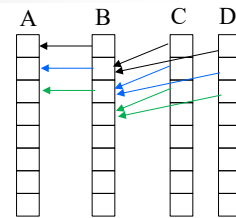
36

Loop-Level Parallelism



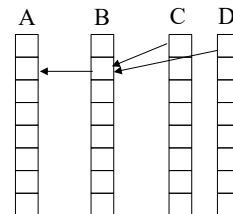
■ Example 3:

```
for (i=0; i<100; i=i+1) {
    A[i] = A[i] + B[i];    /* S1 */
    B[i+1] = C[i] + D[i]; /* S2 */
}
```



- S1 uses value computed by S2 in previous iteration but dependence is not circular so loop is parallel
- May transform to a parallel loop:

```
A[0] = A[0] + B[0];
for (i=0; i<99; i=i+1) do in parallel {
    B[i+1] = C[i] + D[i];
    A[i+1] = A[i+1] + B[i+1];
}
B[100] = C[99] + D[99];
```



37

Loop-Level Parallelism



- Example 4:

```
for (i=0; i<100; i=i+1) {  
    A[i] = B[i] + C[i];  
    D[i] = A[i] * E[i];  
}
```

No loop carried dependencies

- Example 5:

```
for (i=1; i<100; i=i+1) {  
    Y[i] = Y[i-1] + Y[i];  
}
```

Has loop carried dependencies

38

Loop-Level Parallelism



- Example 6:

```
for (i=0; i<100; i=i+1) {  
    A[2 * i + 1] = B[i] + C[i];  
    D[2 * i] = A[2 * i] + E[i];  
}
```

No loop carried dependencies

- Example 6 is a special case of the following:

```
for (i=1; i<100; i=i+1) {  
    A[a * i + b] = ....  
    D[i] = A[c * i + d] + ....  
}
```

loop carried dependencies??

39

Finding dependencies



- Assume indices are affine:
 - $a * i + b$ (i is loop index)
- Assume:
 - Store to $a * i + b$ and load from $c * i + d$
 - i is in the range from m to n
 - Dependence exists if:
 - There exists j, k such that $m \leq j \leq n, m \leq k \leq n$
 - Store to $a * j + b$, load from $c * k + d$, and $a * j + b = c * k + d$
 - GCD test (sufficient but not necessary):
 - A dependence does not exist if $\text{GCD}(c, a)$ does not evenly divide $(d - b)$

40

Finding dependencies



- Example 2:

```
for (i=0; i<100; i=i+1) {  
    Y[i] = X[i] / c; /* S1 */  
    X[i] = X[i] + c; /* S2 */  
    Z[i] = Y[i] + c; /* S3 */  
    Y[i] = c - Y[i]; /* S4 */  
}
```
- Watch for anti-dependencies and output dependencies

41

Reductions

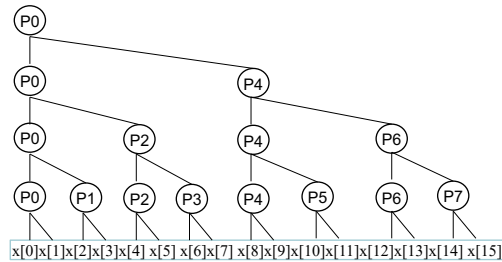
Assume 8 processors, how can you speed up the following loops?:

```
sum = 0 ;
for (i=0; i<16; i=i+1)
    sum += x[i];
```

```
sum = 0 ;
for (i=0; i<400; i=i+1)
    sum += x[i];
```

```
sum = 0 ;
for (i=0; i<16; i=i+1)
    x[i] = x[i-1] + x[i];
```

```
for (i=1; i<400; i=i+1)
    x[i] = x[i-1] + x[i];
```

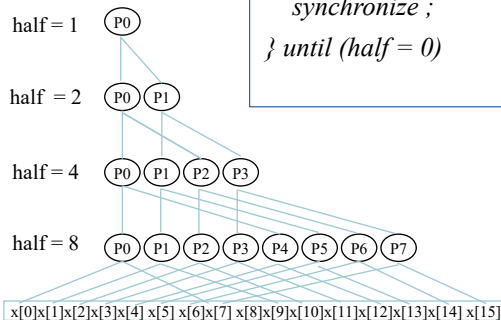


Prefix computation

42

Reductions

```
half = 8 ;                                /*half = n/2*/
Repeat
{  if(Pid < half) x[Pid] = x[Pid] + x[Pid + half] ;
  half = half / 2;
  synchronize ;                          /* a barrier */
} until (half = 0)
```



43