Week 2 Discussion: Parallel Architectures and Software

- Use of Intel SIMD SSE/AVX intrinsics for PA1 SIMD
- False sharing in shared memory architectures
- Parallel software

UCSB CS140

Use of Intel SIMD Intrinsics on CSIL

- **Code to optimize in Programming Assignment 1 SIMD:**

```
for (i=0; i<n; i++)
        sum = sum+ a[i];
```

- **Transform this loop with unrolling**

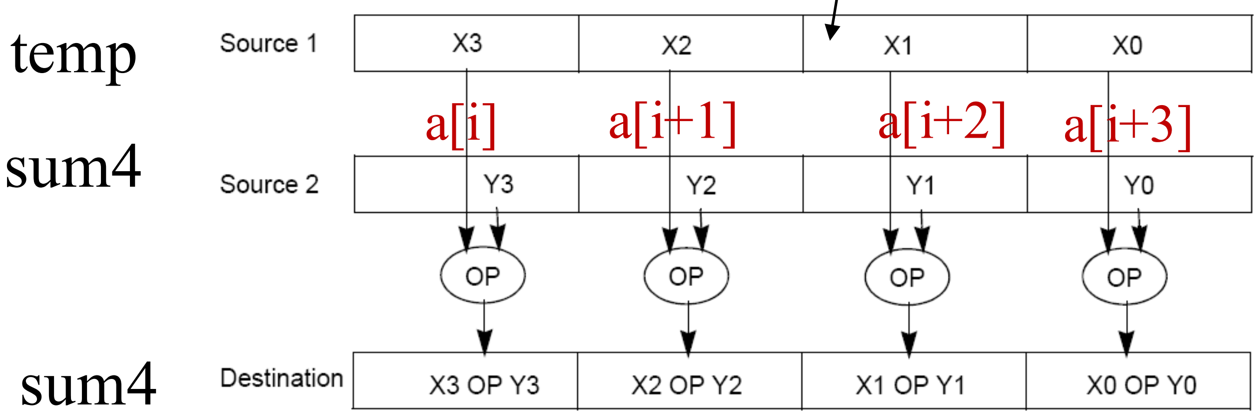
```
for (i = 0;  i<n/4*4; i=i+4){
    sum = sum + a[i];
    sum=  sum + a[i+1];
    sum=  sum + a[i+2];
    sum=  sum + a[i+3];
}
for(i=n/4*4; i<n; i++) sum   += a[i];
```

Use of 128-bit SIMD instruction

For each 4 members in array {
  Load 4 members to the SSE register
  Accumulate with 4 additions in the register
}
Fetch 4 results from the register and add together

Related SSE Intrinsics

__m128i _mm_setzero_si128( )	returns 128-bit zero vector
__m128i _mm_loadu_si128(__m128i *p)	Load data stored at pointer p of memory to a 128bit vector, returns this vector.
__m128i _mm_add_epi32(__m128i a, __m128i b)	returns vector (a_0+b_0, a_1+b_1, a_2+b_2, a_3+b_3) with 4 32-bit integers
void _mm_storeu_si128(__m128i *p, __m128i a)	stores content of 128-bit vector ”a” to memory starting at pointer p



Use of Intel SIMD SSE Intrinsics

```
for (i = 0;  n/4*4; i=i+4){
        sum = sum + a[i];
        sum=  sum + a[i+1];
        sum=  sum + a[i+2];
        sum=  sum + a[i+3]}
```

Load data from memory to a 128-bit register

Add 4 numbers in parallel

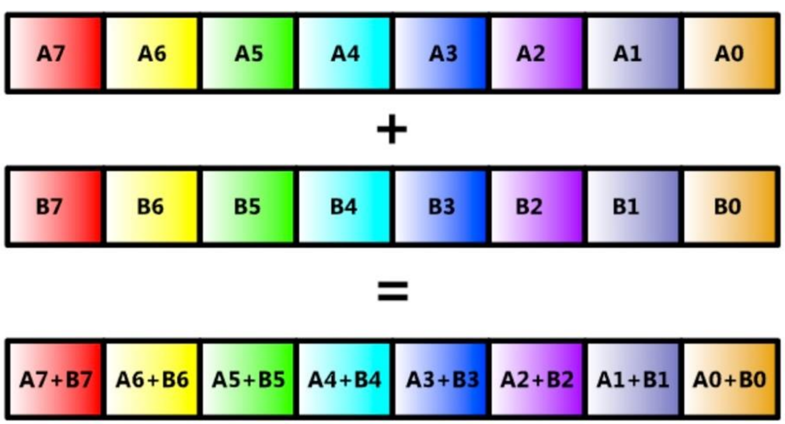
```
__m128i sum4=__mm_setzero_si128();
 for (i = 0;  n/4*4; i=i+4){
       __m128i temp=_mm_loadu_si128 (& a[i]) ;
       sum4=_mm_add_epi32(sum4, temp); }
```

- Next, copy out 4 integers from sum4 and add them to sum.

```
int s[4] __attribute__((aligned(16)));
_mm_storeu_si128(__m128i *)s, sum4);
sum=s[0]+s[1]+s[2]+s[3];
```

Related AVX2 (256 Bits) for PA1 SIMD

__m256i _mm_setzero_si256( )	returns 256-bit zero vector
__m256i _mm_loadu_si256(__m256i *p)	Load data stored at pointer p of memory to a 256bit vector, returns this vector.
__m156i _mm_add_epi32(__m256i a, __m256i b)	returns vector (a_0+b_0, a_1+b_1,…, a_7+b_7) with 8 32-bit integers
void _mm_storeu_si256(__m256i *p, __m256i a)	stores content of 256-bit vector ”a” to memory starting at pointer p



https://www.cs.virginia.edu/~cr4bd/3330/F2018/simdref.html

Cache coherence in a shared memory machine

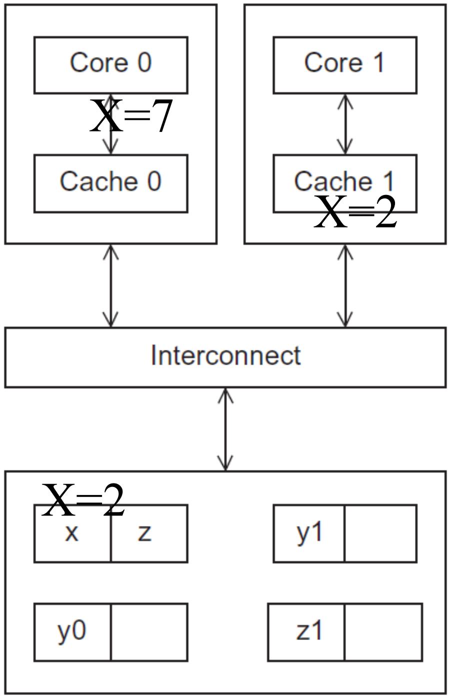
x = 2; /* shared variable */

Time	Core 0	Core 1
0	`y0 = x;`	`y1 = 3*x;`
1	`x = 7;`	Statement(s) not involving x
2	Statement(s) not involving x	`z1 = 4*x;`

y0 eventually ends up = 2

y1 eventually ends up = 6

Statement z1 is executed in Core 1 after x=7 in Core 0



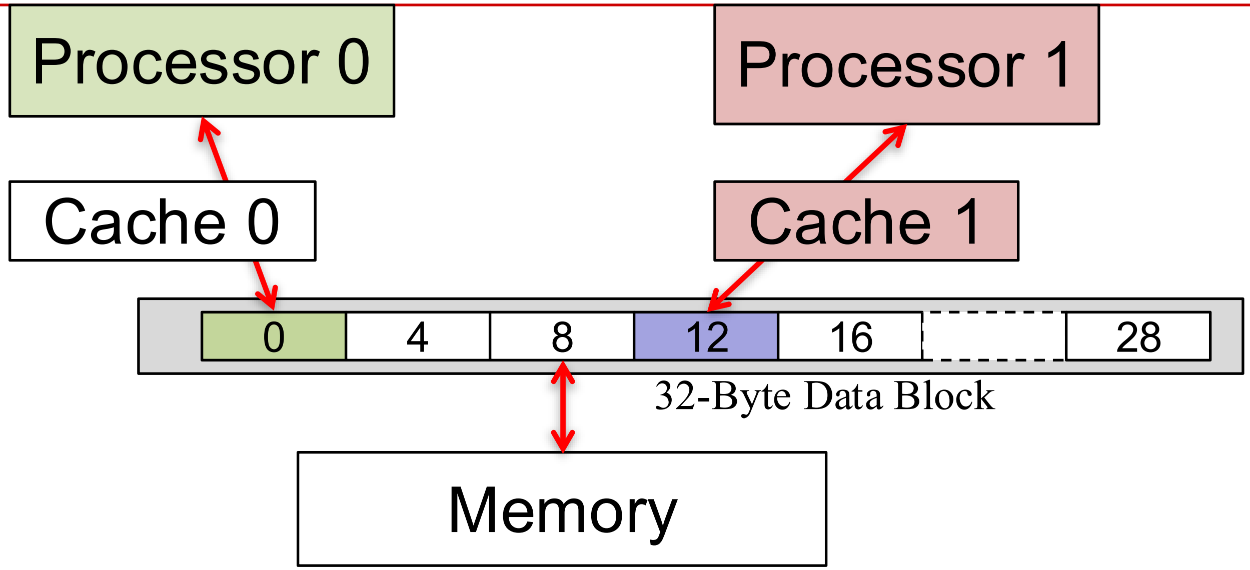
Should z1 = 4*7 or 4*2?

z1=4*7

Cache coherence: an update to a variable cached in one processor should be seen in other processors.

Hardware ensures cache coherence.

False Sharing in Shared Memory Machines

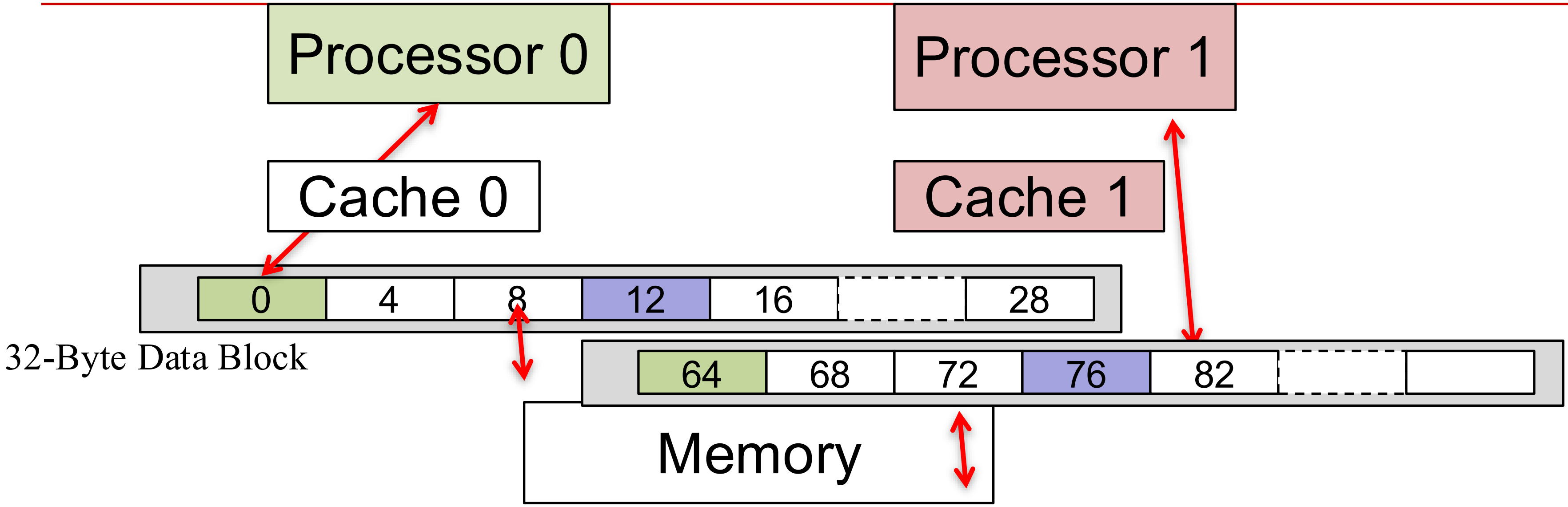


- Cache block size is 32 bytes

Time	Proc 0	Proc 1
0	Write data #0 Invalidate cache block of Proc 1	
1		Write data #12 Invalidate cache block at Proc 0

Local cache is not effectively used due frequent invalidation

No False Sharing



- Cache block size is 32 bytes

Time	Proc 0	Proc 1
0	Write data #0 This block is not cached in other proc.	
1		Write data #82 Invalidating cache block does not affect others

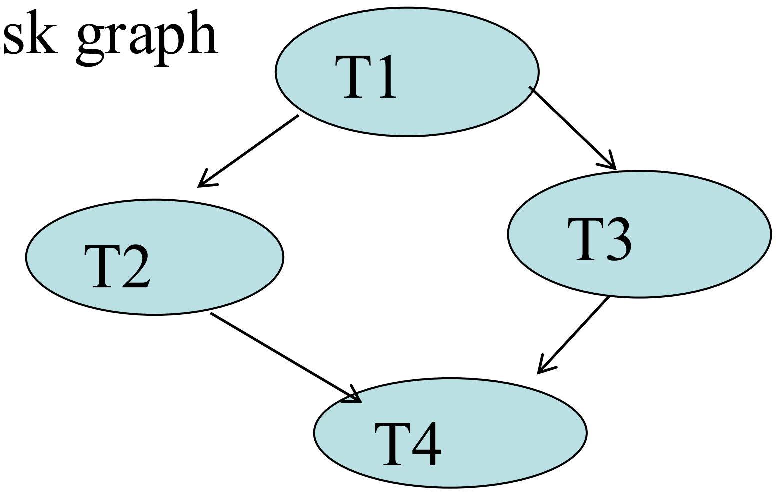
Local cache can be effectively used for each processor

Discussion on Parallel Software

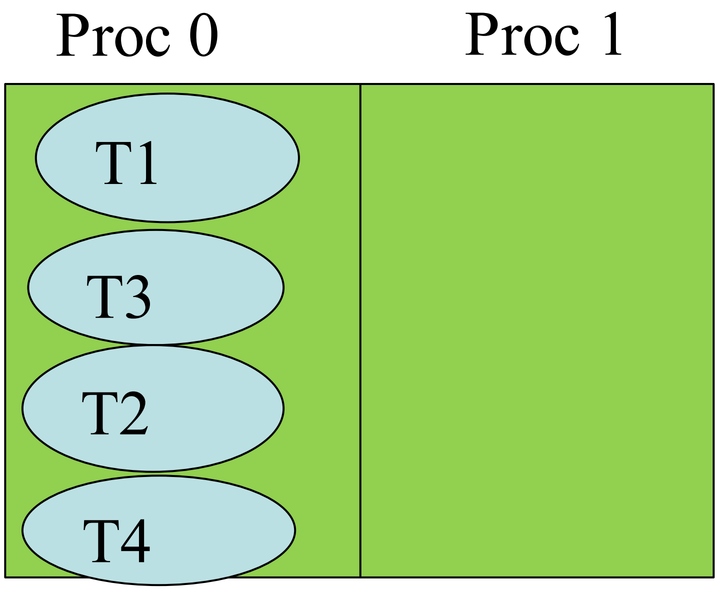
- Example of valid and invalid scheduling
- Parallel tree sum
- Parallelizing matrix vector multiplication
- Running MPI on CSIL
- Expanse cluster usage
  - If time does not permit, next week

Task Scheduling: Map and execute tasks on multiprocessors

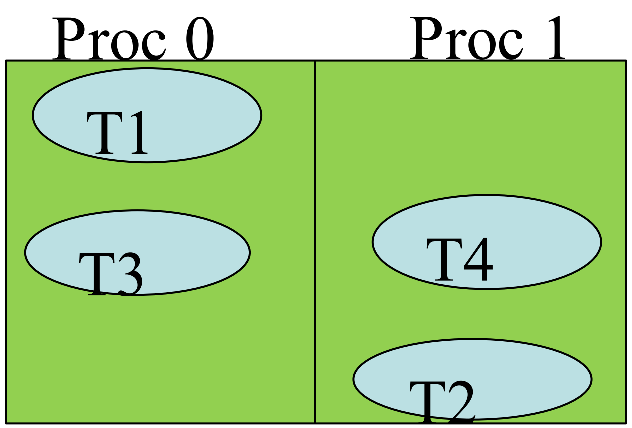
Task graph

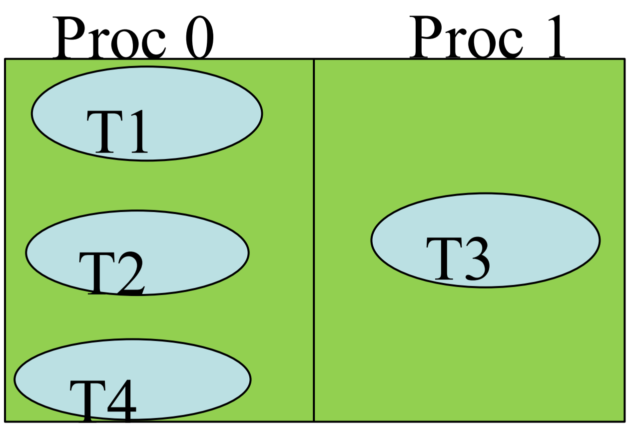


Which schedule is valid?



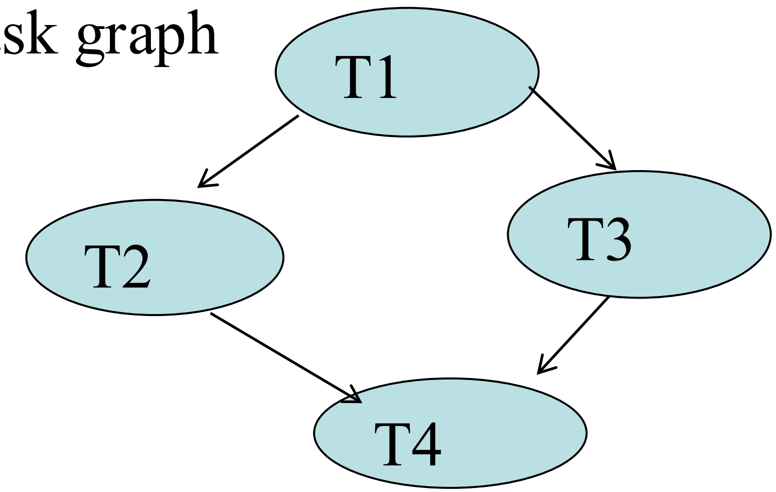
Each processor executes assigned tasks sequentially



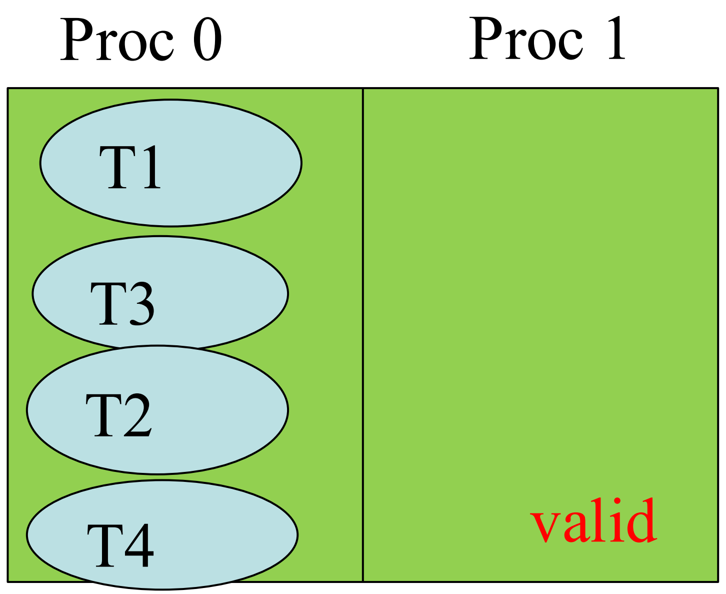


Task Scheduling: Map and execute tasks on multiprocessors

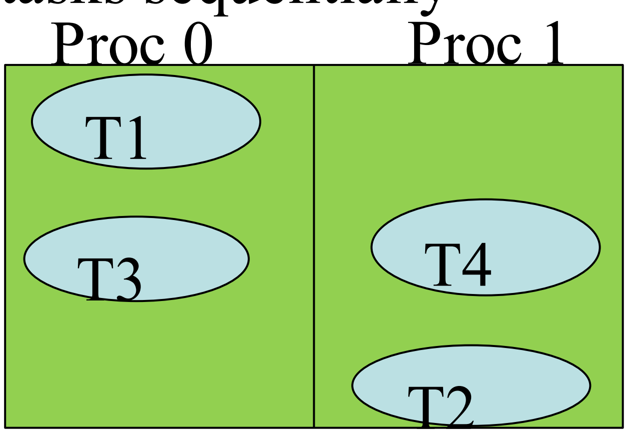
Task graph



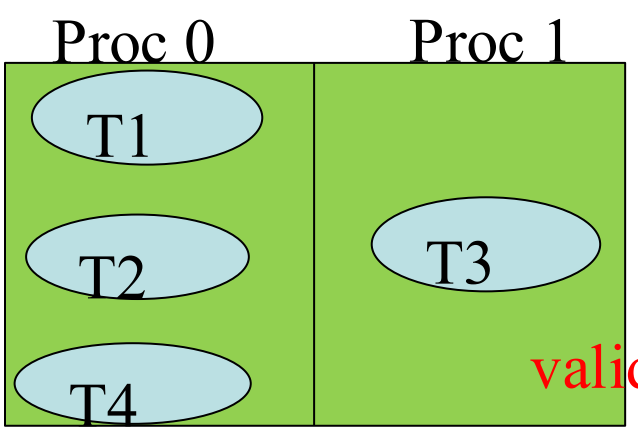
Which schedule is valid?



Each processor executes assigned tasks sequentially



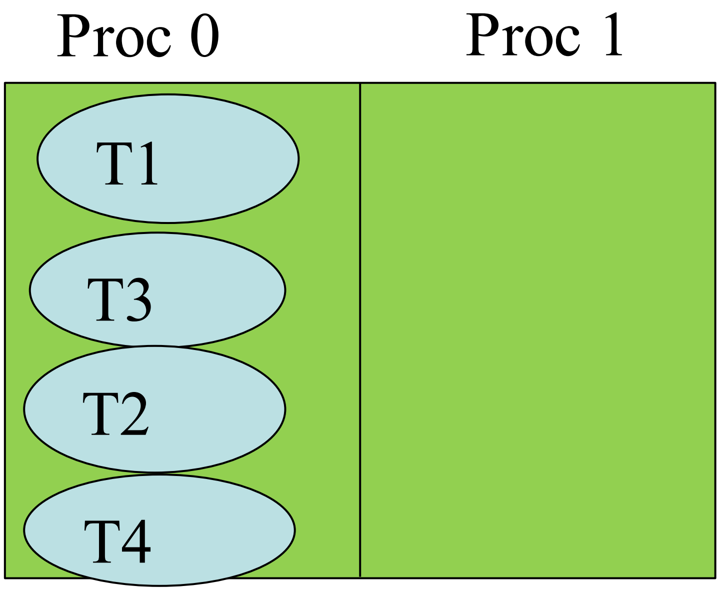
Not valid as T2->T4 dependence is violated.



Example: Estimation of Parallel Time from a Schedule

Assume each task takes 1 time unit

Parallel time=4

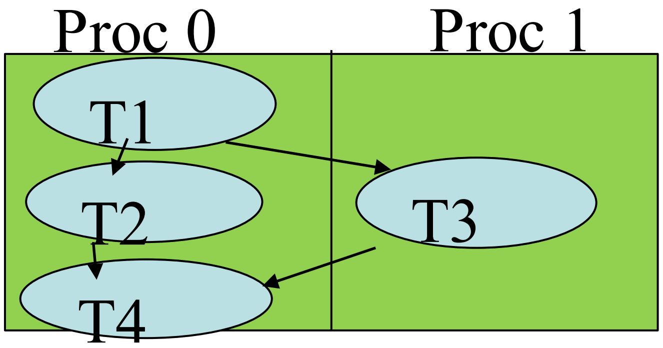


Shared memory machines
with 0 synchronization cost

Parallel time=3

Distributed memory machines
Communication costs 0 time unit

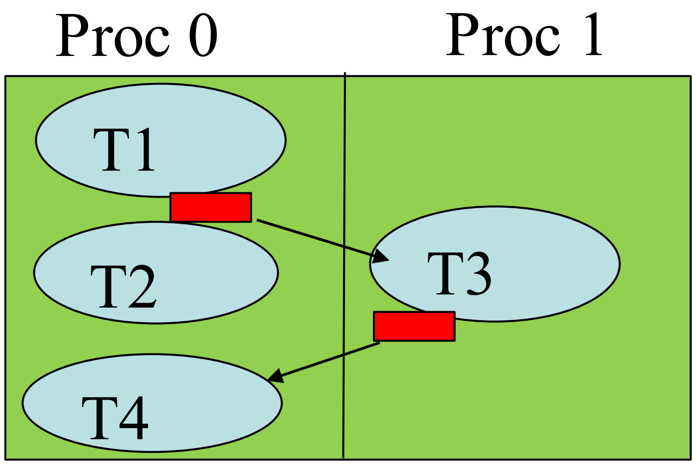
Parallel time=3



Example: Estimation of Parallel Time on Distributed Memory Machines

Assume each task takes 1 time unit

Sending startup costs 0.5
Receiving costs 0
Message travel costs 0.5, which can overlap with computation



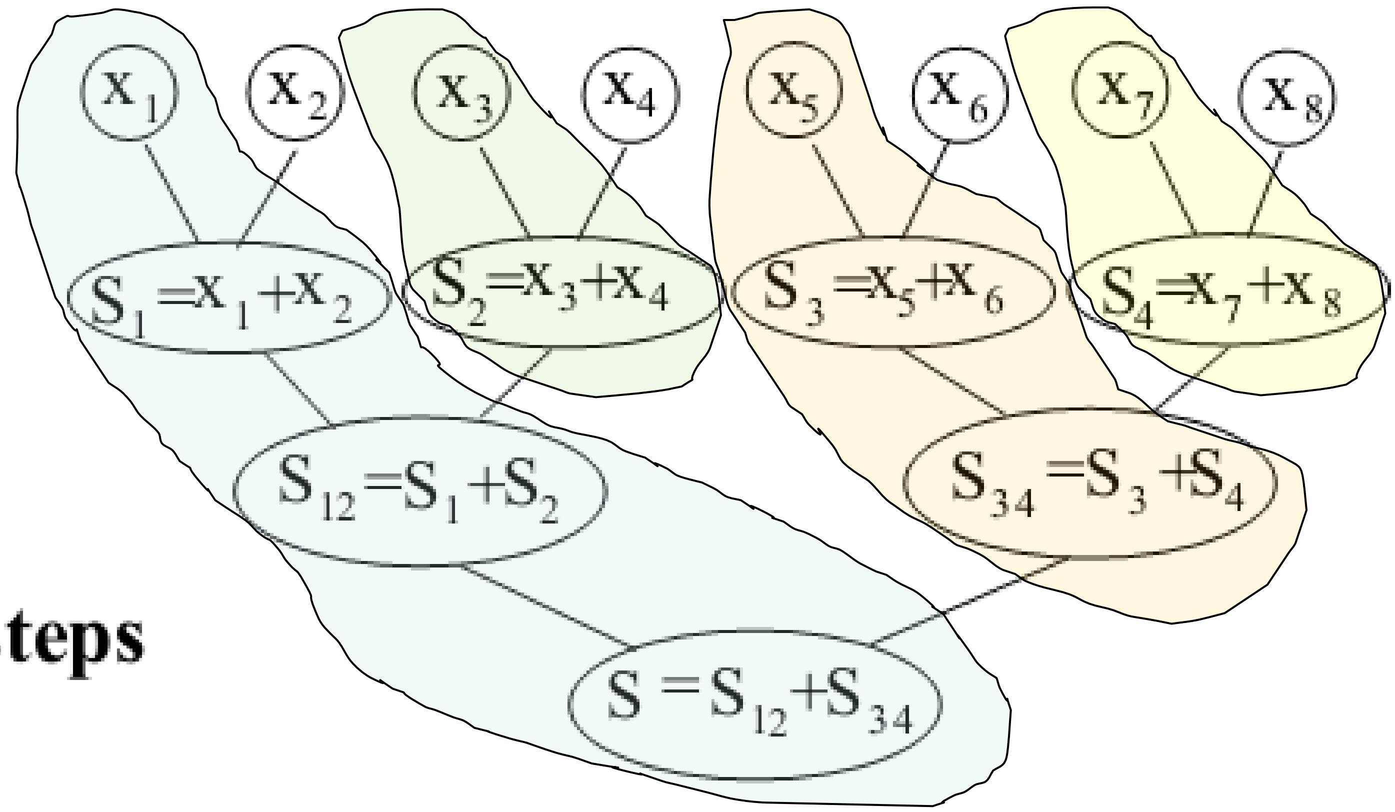
The path T1, T3, T4 costs 5

- T1 costs 1
- Message T1→T3 costs: 0.5 +0.5
- T3 costs 1
- Message T3→T4 costs: 0.5 +0.5
- T4 costs 1

Parallel time= completion time of T3 = 5

How to write SPMD code for tree summation?

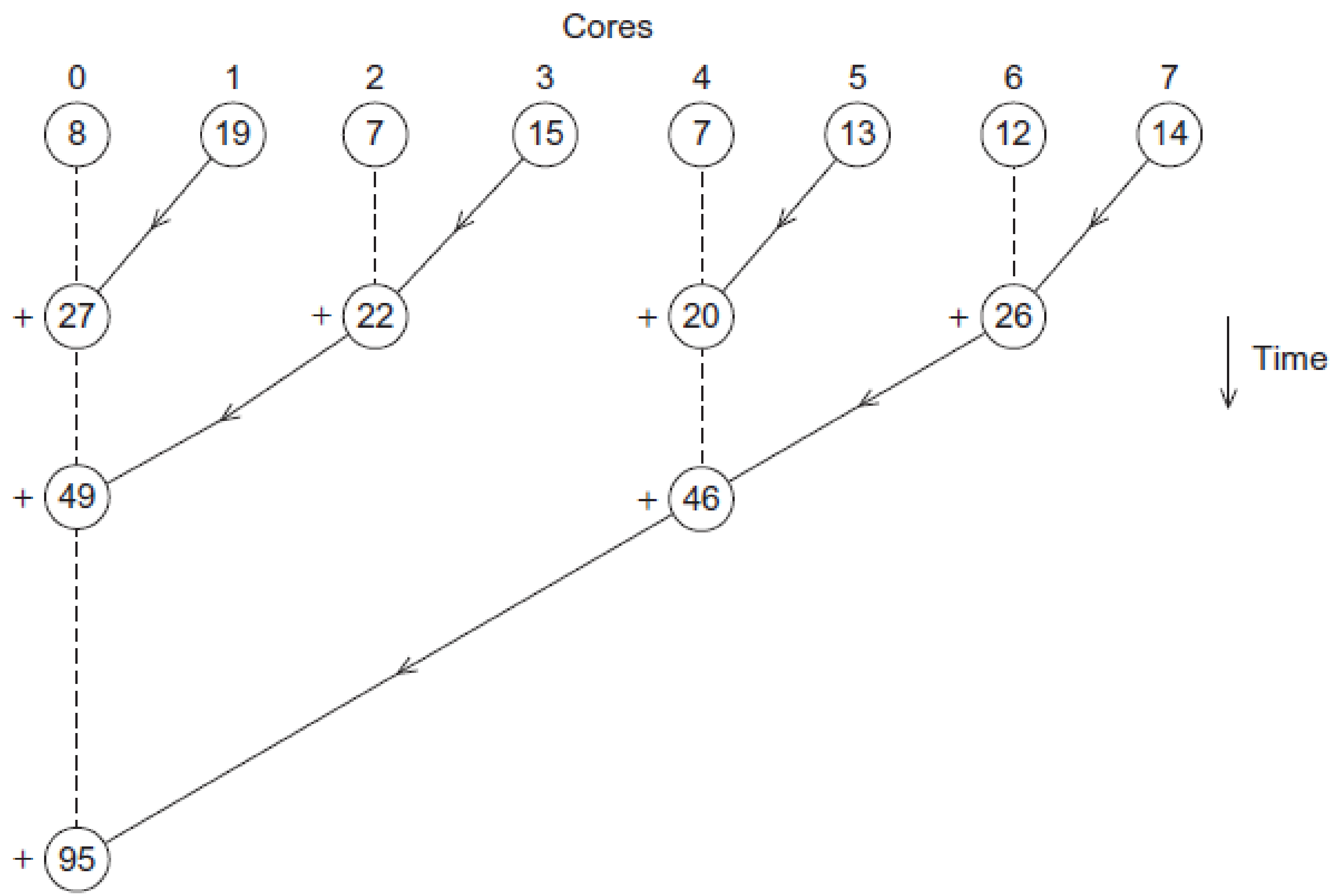
Hints for Programming Assignment 1



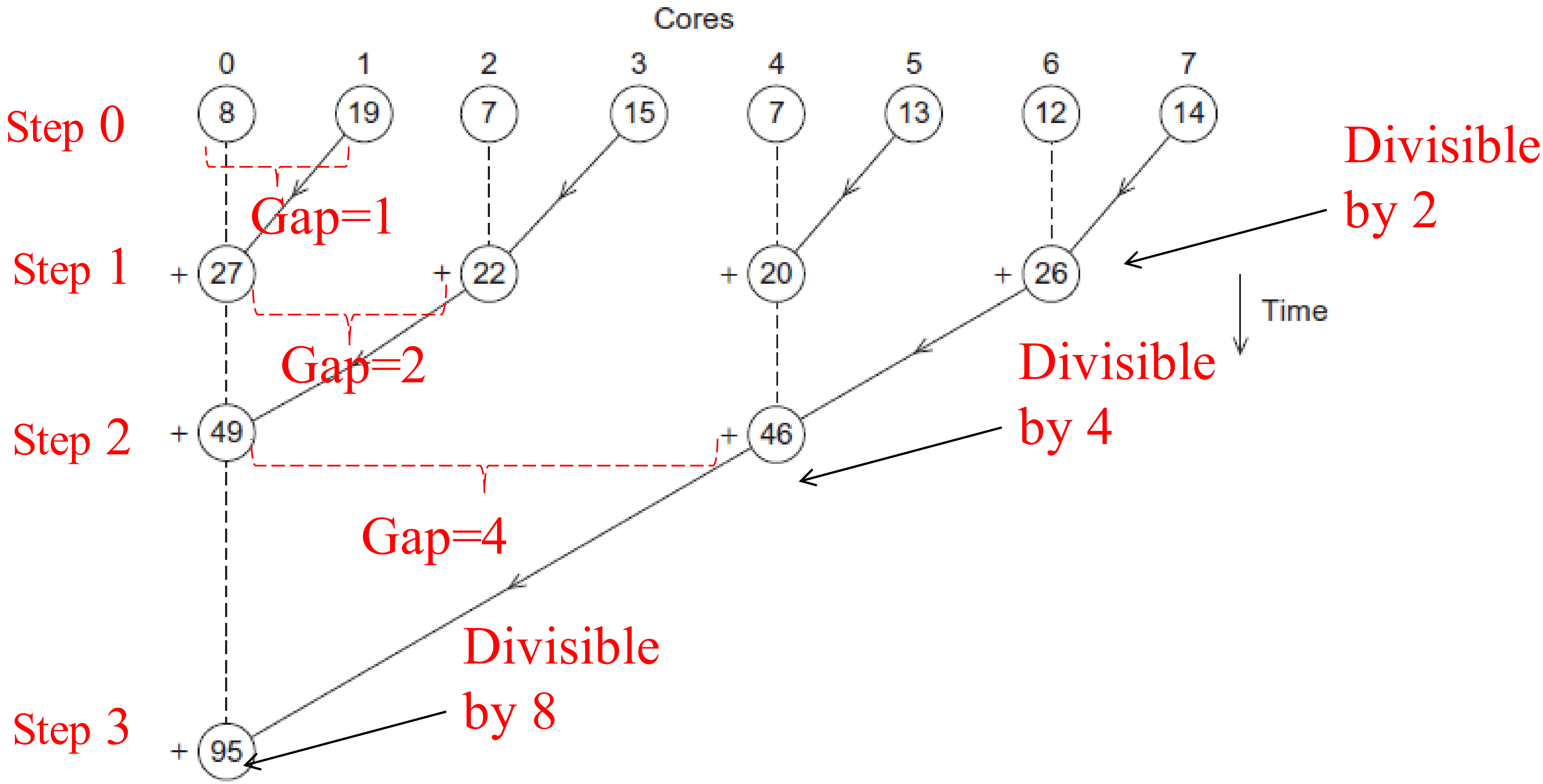
3 steps

Parallel summation: Textbook Figure 1.1

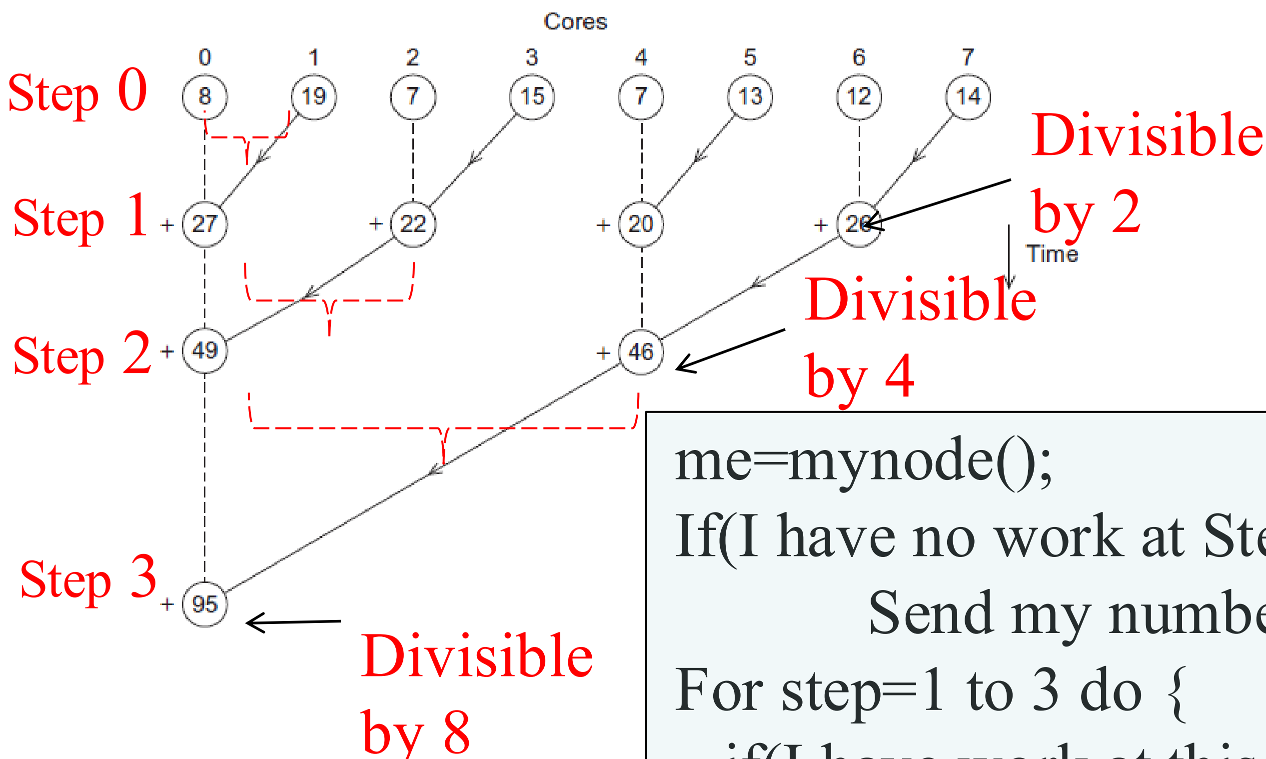
- **Skew the previous graph**



Patterns of parallel computation:
Who needs to receive a number and add it? Who needs to send it?

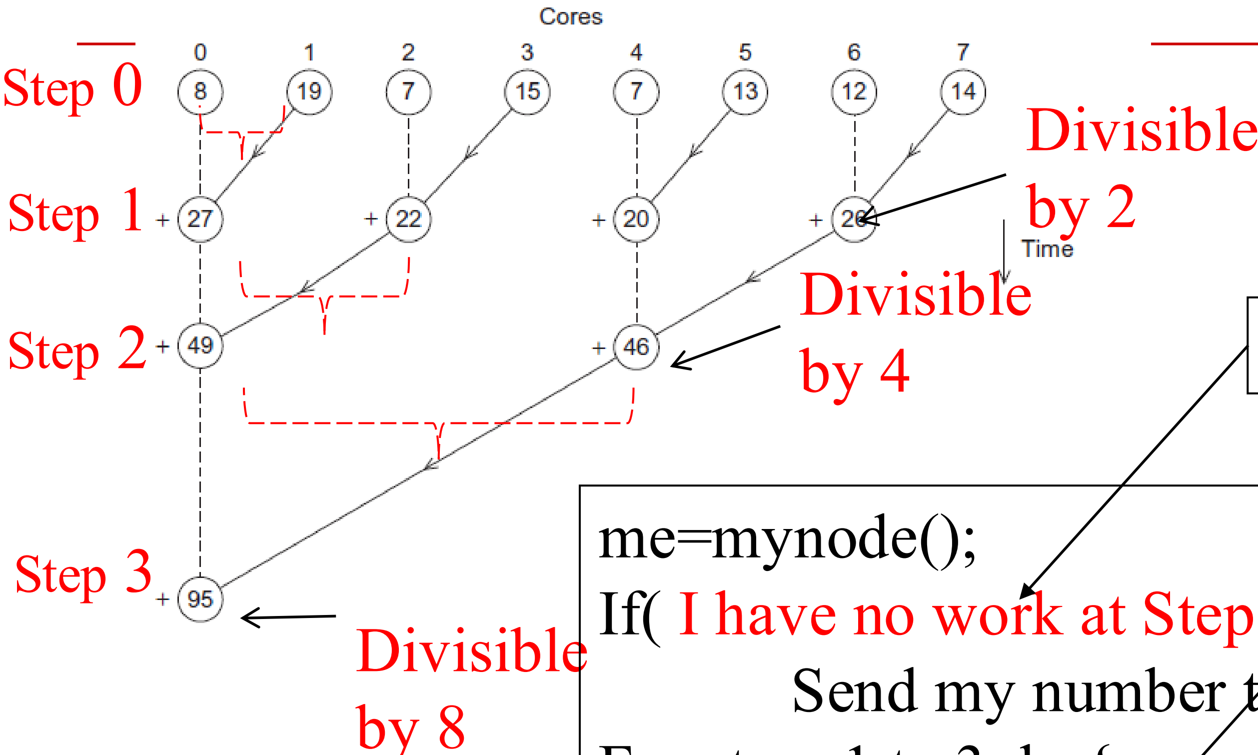


Patterns of parallel computation & SPMD code




```
me=mynode();
If(I have no work at Step 1)
        Send my number to a neighbor;
For step=1 to 3 do {
  if(I have work at this step) {
       Receive a number and accumulate
       If (I have no work at next step)
           Send my sum to somebody who needs;
  }
}
```

Patterns of parallel computation & SPMD code



I am divisible by 2

I am divisible by step

```
me=mynode();
If( I have no work at Step 1)
        Send my number to a neighbor;
For step=1 to 3 do {
   if( I have work at this step) {
        Receive a number and accumulate
        If (I have no work at next step)
            Send my sum to somebody who needs;
   }
}
```

Matrix-vector multiplication: y= A * x

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} * \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} = \begin{pmatrix} 1*1+2*2+3*3 \\ 4*1+5*2+6*3 \\ 7*1+8*2+9*3 \end{pmatrix} = \begin{pmatrix} 14 \\ 32 \\ 50 \end{pmatrix}$$

Problem: $y = A * x$ where A is a $n \times n$ matrix and x is a column vector of dimension n.

Sequential code:

for $i = 1$ **to** n **do**
  $y_i = 0;$
  for $j = 1$ to n **do**
    $y_i = y_i + a_{i,j} * x_j;$
  endfor
endfor

Textbook 113-114
Exercise 0

Partitioning and Task graph for matrix-vector multiplication

Partitioned code:

for $i = 1$ **to** n **do**

S_i : $y_i = 0;$

for $j = 1$ to n **do**

$y_i = y_i + a_{i,j} * x_j;$

endfor

endfor

S_i : Read row A_i and vector x.

Write element y_i

yi= Row Ai multiplies x

Task graph:

Execution Schedule and Task Mapping

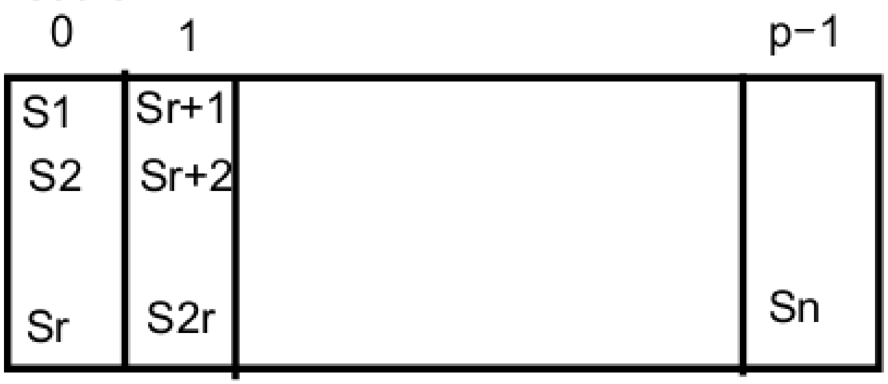
S_i : Read row A_i and vector x.

Write element y_i

yi= Row Ai multiplies x

Task graph:

Schedule:



Example,
n=10, p=3, what is r?

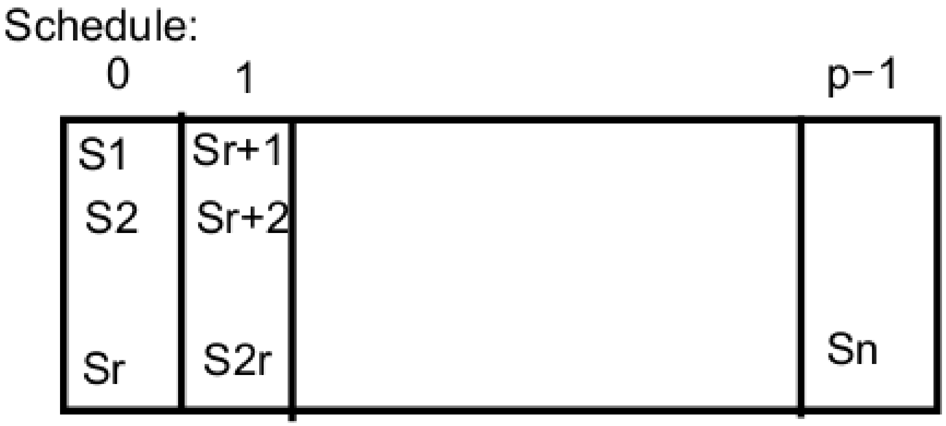
r=4 with 4, 4, and 2 distribution.

Mapping function of tasks S_i:

$$proc\_map(i) = \lfloor \frac{i-1}{r} \rfloor \text{ where } r = \lceil \frac{n}{p} \rceil.$$

r=3 is wrong with 3, 3, 3 distribution
Thus r=n/p is wrong

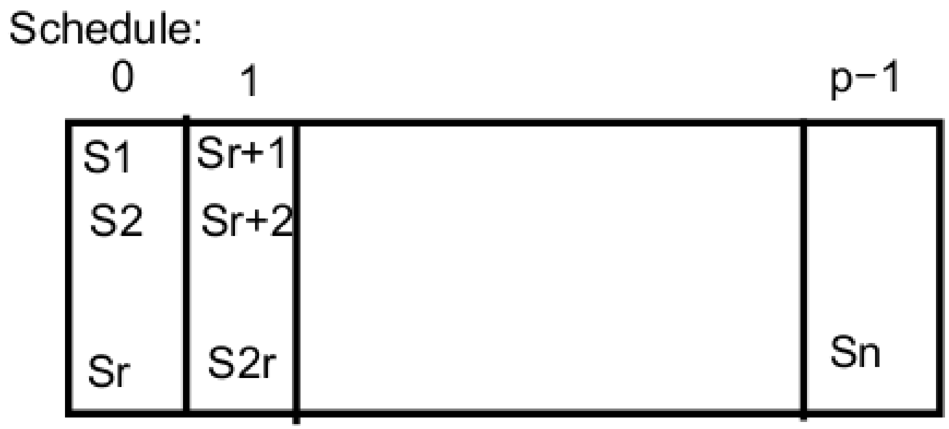
Estimation of Parallel Time from the Schedule



- Each task performs *n* additions and *n* multiplications.
  - Assume Each addition/multiplication costs ω
- The parallel time is approximately $\frac{n}{p} \times 2n\omega$

- Only count main arithmetic costs
  - Ignore low-level implementation cost such as local address calculation, and loop iteration overhead.
  - They are less significant

Unoptimized SPMD Code for y= A*x



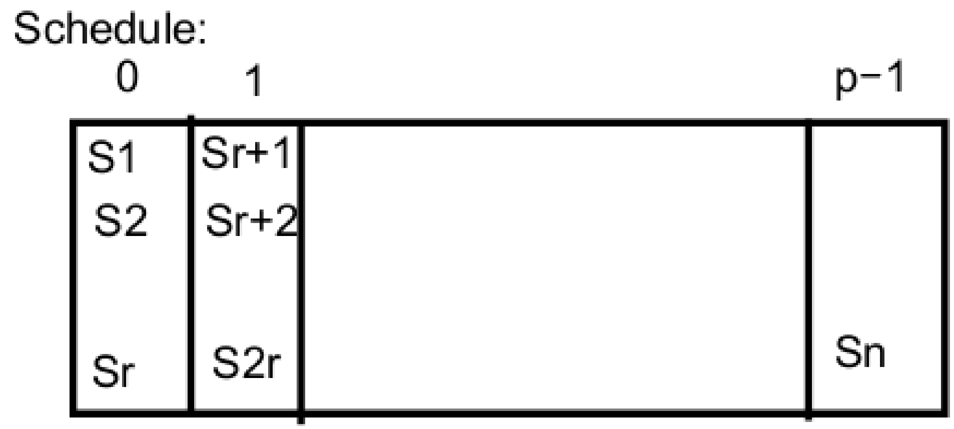

```
me=mynode();
for i = 1 to n do
   if proc_map(i)== me, then
         do Si
```

Si

```
y[i]=0
for j= 1 to n do
   y[i]=y[i] + a[i][j]*x[j]
```

- Easy to understand.
- Extra overhead to iterate through all n checkups

SPMD Code for y= A*x with loop checkup overhead removed




```
me=mynode();
p= noproc();
r = ceiling(n/p);
first = me*r +1;
last =  first +r;
for i = first to last do
   do Si
```

Si

```
y[i]=0
for j= 1 to n do
   y[i]=y[i] + a[i][j]*x[j]
```

- More difficult to read code
- No overhead for checkup