Parallel programming using MPI

Analysis and optimization

Bhupender Thakur, Jim Lupo, Le Yan, Alex Pacheco

Outline

- Parallel programming:
  Basic definitions
- Choosing right algorithms:
  Optimal serial and parallel
- Load Balancing
  Rank ordering, Domain decomposition
- Blocking vs Non blocking
  Overlap computation and communication
- MPI-IO and avoiding I/O bottlenecks
- Hybrid Programming model
  MPI + OpenMP
  MPI + Accelerators for GPU clusters

Choosing right algorithms:

How does your serial algorithm scale?

Notation	Name	Example
$O(1)$	constant	Determining if a number is even or odd; using a hash table
$O(\log\log n)$	double log	Finding an item using interpolation search in a sorted array
$O(\log n)$	log	Finding an item in a sorted array with a binary search
$O(n^c)$, $c<1$	fractional power	Searching in a kd-tree
$O(n)$	linear	Finding an item in an unsorted list or in an unsorted array
$O(n\log n)$	loglinear	Performing a Fast Fourier transform; heapsort, quicksort
$O(n^2)$	quadratic	Naïve bubble sort
$O(n^c)$, $c>1$	polynomial	Matrix multiplication, inversion
$O(c^n)$, $c>1$	exponential	Finding the (exact) solution to the travelling salesman problem
$O(n!)$	factorial	generating all unrestricted permutations of a poset

Parallel programming concepts:

Basic definitions

Symbol	Definition
n	problem size
p	number of processors
m	number of memory cells
T	number of steps for sequential algorithm
t	number of steps for parallel algorithm
w	work done by the parallel algorithm
S	speedup
E	efficiency
t_s	execution time for sequential algorithm
t_p	execution time for parallel algorithm

Parallel programming concepts:
Performance metrics

Speedup:

Ratio of parallel to serial execution time

$$S = \frac{t_s}{t_p}$$

Efficiency:

The ratio of speedup to the number of processors

$$E = \frac{S}{p}$$

Work

Product of parallel time and processors

$$W = tp$$

Parallel overhead

Idle time wasted in parallel execution

$$T_0 = pt_p - t_s$$

Parallel programming concepts

Analyzing serial vs parallel algorithms

Ask yourself

What fraction of the code can you completely parallelize ?

f ?

How does problem size scale? Processors scale as p. How does problem-size n scale with p?

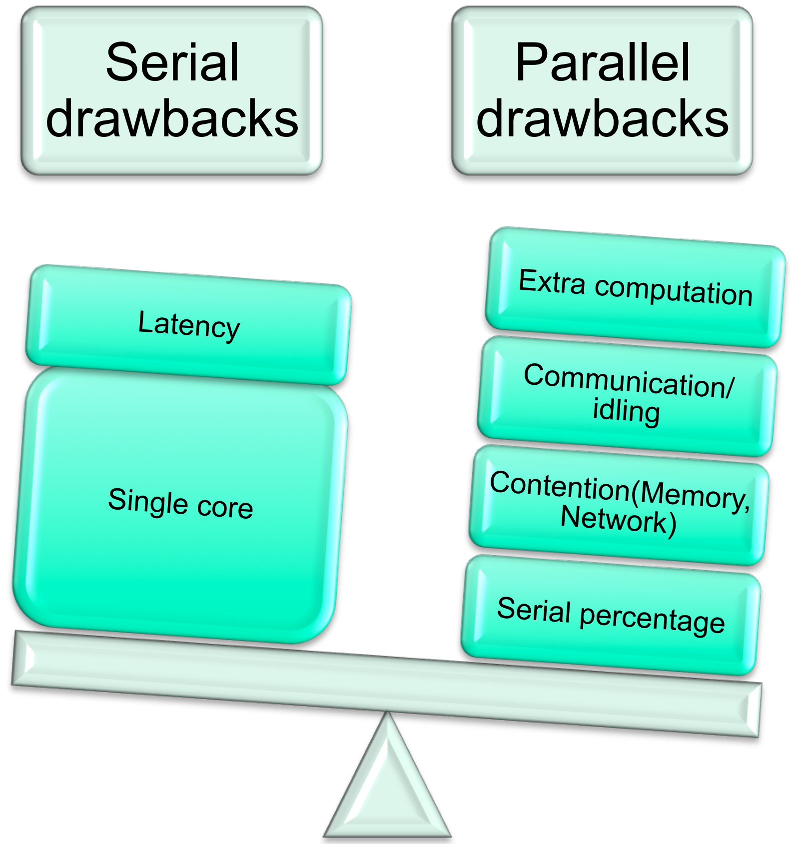
n(p) ?

How does parallel overhead grow?

$T_o(p)$?

Does the problem scale?

M(p)



LSU

HPC HIGH PERFORMANCE COMPUTING

Parallel programming concepts

Analyzing serial vs parallel algorithms

Amdahl's law

What fraction of the code can you completely parallelize ?

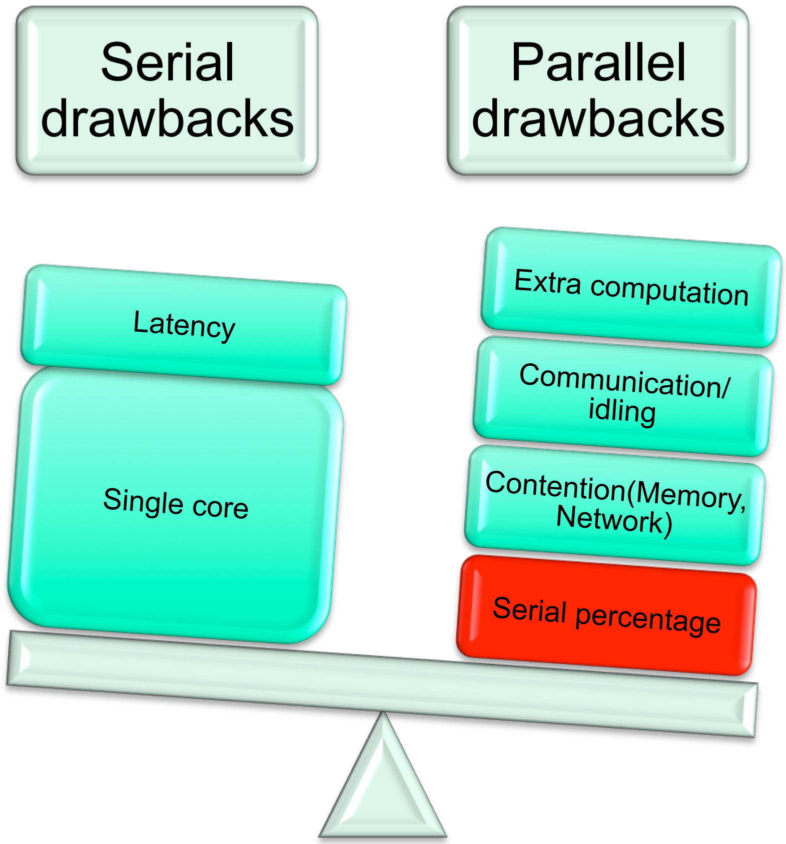
f ?

Serial time: t_s

Parallel time: $ft_s + (1-f)(t_s/p)$

$$S = \frac{t_s}{t_p} = \frac{t_s}{t_s f + \frac{(1-f)t_s}{p}}$$

$$= \frac{1}{f + \frac{(1-f)}{p}}$$



Parallel programming concepts

Analyzing serial vs parallel algorithms

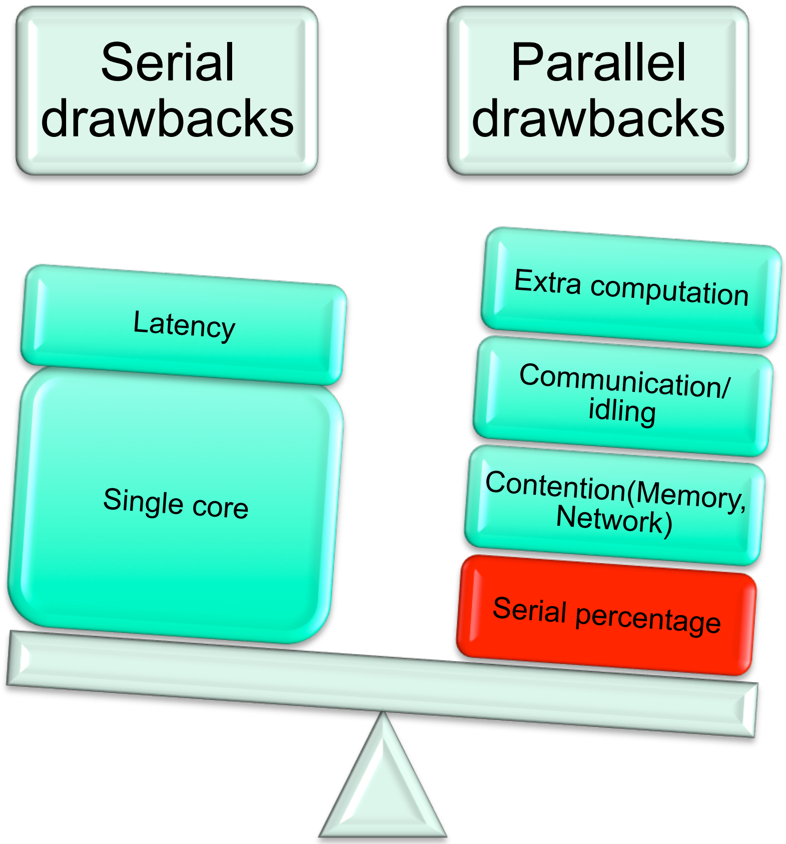
Quiz

if the serial fraction is 5%, what is the maximum speedup you can achieve ?

Serial time = 100 secs

Serial percentage = 5 %

Maximum speedup ?



Parallel programming concepts

Analyzing serial vs parallel algorithms

Amdahl's law

What fraction of the code can you completely parallelize ?

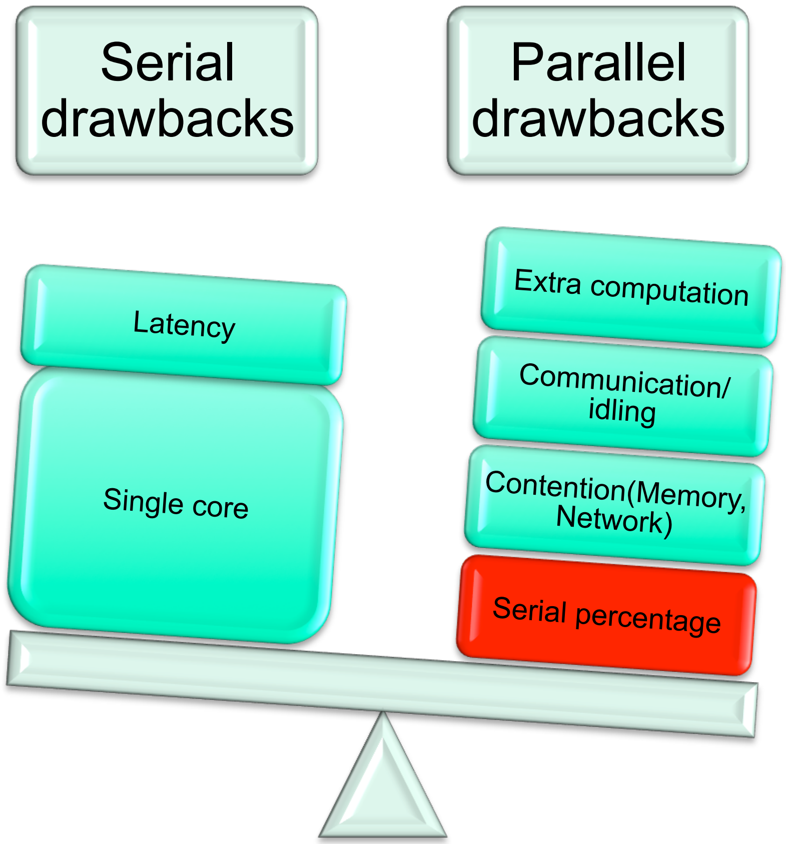
f ?

Serial time = 100 secs

Parallel percentage = 5 %

$$= \frac{1}{.05 + \frac{(1-f)}{p}}$$

$$= \frac{1}{.05 + 0} = 20$$



Parallel programming concepts

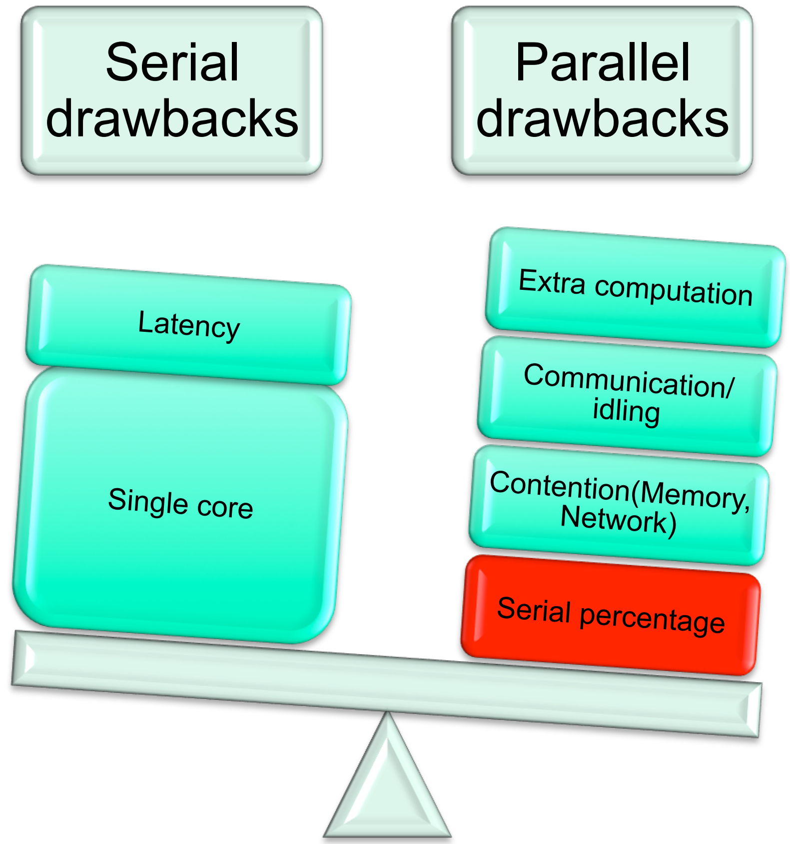
Analyzing serial vs parallel algorithms

Amdahl's Law approximately suggests:

" Suppose a car is traveling between two cities 60 miles apart, and has already spent one hour traveling half the distance at 30 mph. No matter how fast you drive the last half, it is impossible to achieve 90 mph average before reaching the second city "

Gustafson's Law approximately states:

" Suppose a car has already been traveling for some time at less than 90mph. Given enough time and distance to travel, the car's average speed can always eventually reach 90mph, no matter how long or how slowly it has already traveled."



Source: http://disney.go.com/cars/
http://en.wikipedia.org/wiki/Gustafson's_law

Parallel programming concepts
Analyzing serial vs parallel algorithms

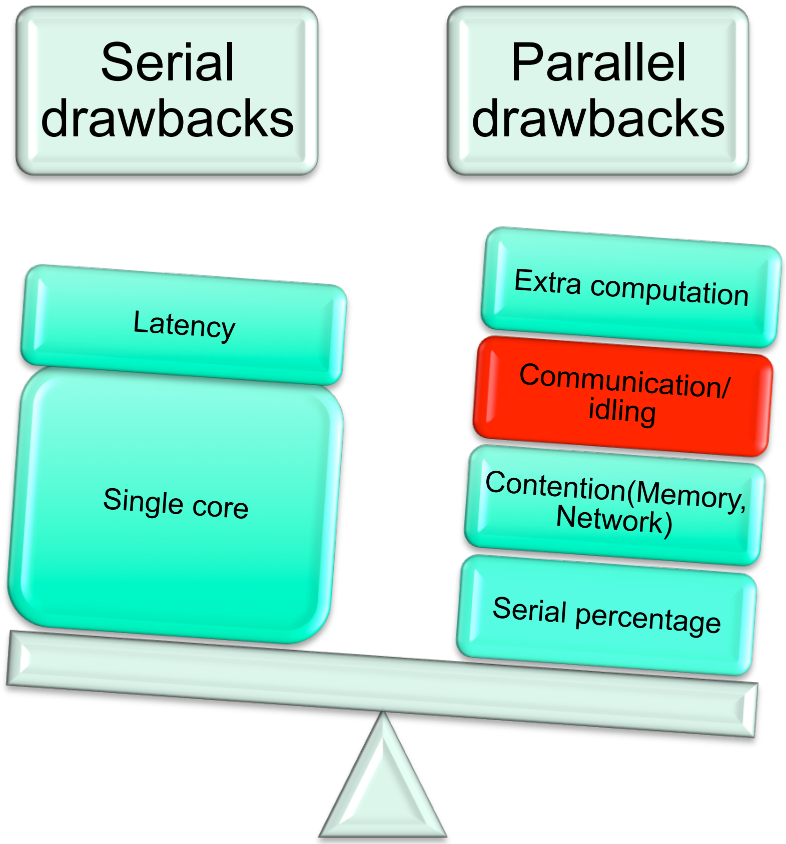
Communication Overhead

Simplest model:

Transfer time
= Startup time
+ Hop time(Node latency)
+ (Message length)/Bandwidth

$= t_s + t_h l + t_w l$

Send one big message instead of several small messages!
Reduce the total amount of bytes!
Bandwidth depends on protocol



Parallel programming concepts

Analyzing serial vs parallel algorithms

Point to point (MPI_Send)

$(t_s + t_w m)$

Collective overhead

All-to-all Broadcast (MPI Allgather):

$t_s \log_2 p + (p-1) \ t_w m$

All-reduce (MPI Allreduce) :

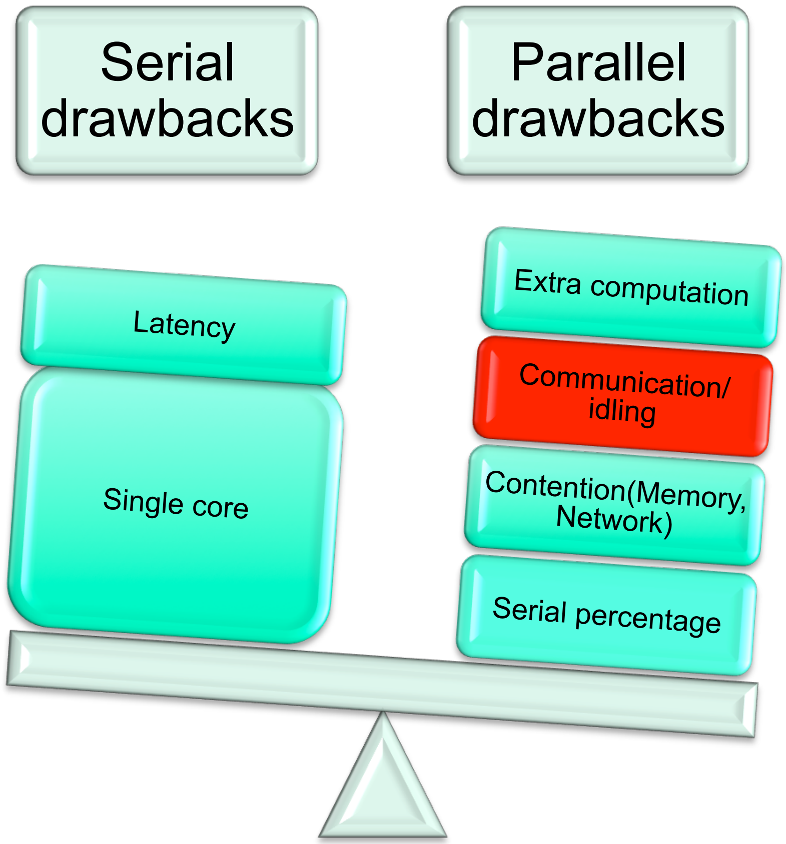
$(t_s + t_w m) \log_2 p$

Scatter and Gather (MPI Scatter) :

$(t_s + t_w m) \log_2 p$

All to all (personalized):

$(p-1) (t_s + t_w m)$



Parallel programming:
Basic definitions

Isoefficiency:

Can we maintain efficiency/speedup of the algorithm?

How should the problem size scale with p to keep efficiency constant?

$$E = \frac{1}{1 + (T_o / w)}$$

Maintain ratio *To(W,p) I W , overhead to parallel work constant*

Parallel programming:
Basic definitions

Isoefficiency relation: To keep efficiency constant you must increase problem size such that

$$T(n,1) \geq T_o(n,p)$$

Procedure:

1. Get the sequential time $T(n,1)$
2. Get the parallel time $pT(n,p)$
3. Calculate the overhead $T_o = pT(n,p) - T(n,1)$

How does the overhead compare to the useful work being done?

Parallel programming:
Basic definitions

Isoefficiency relation: To keep efficiency constant you must increase problem size such that

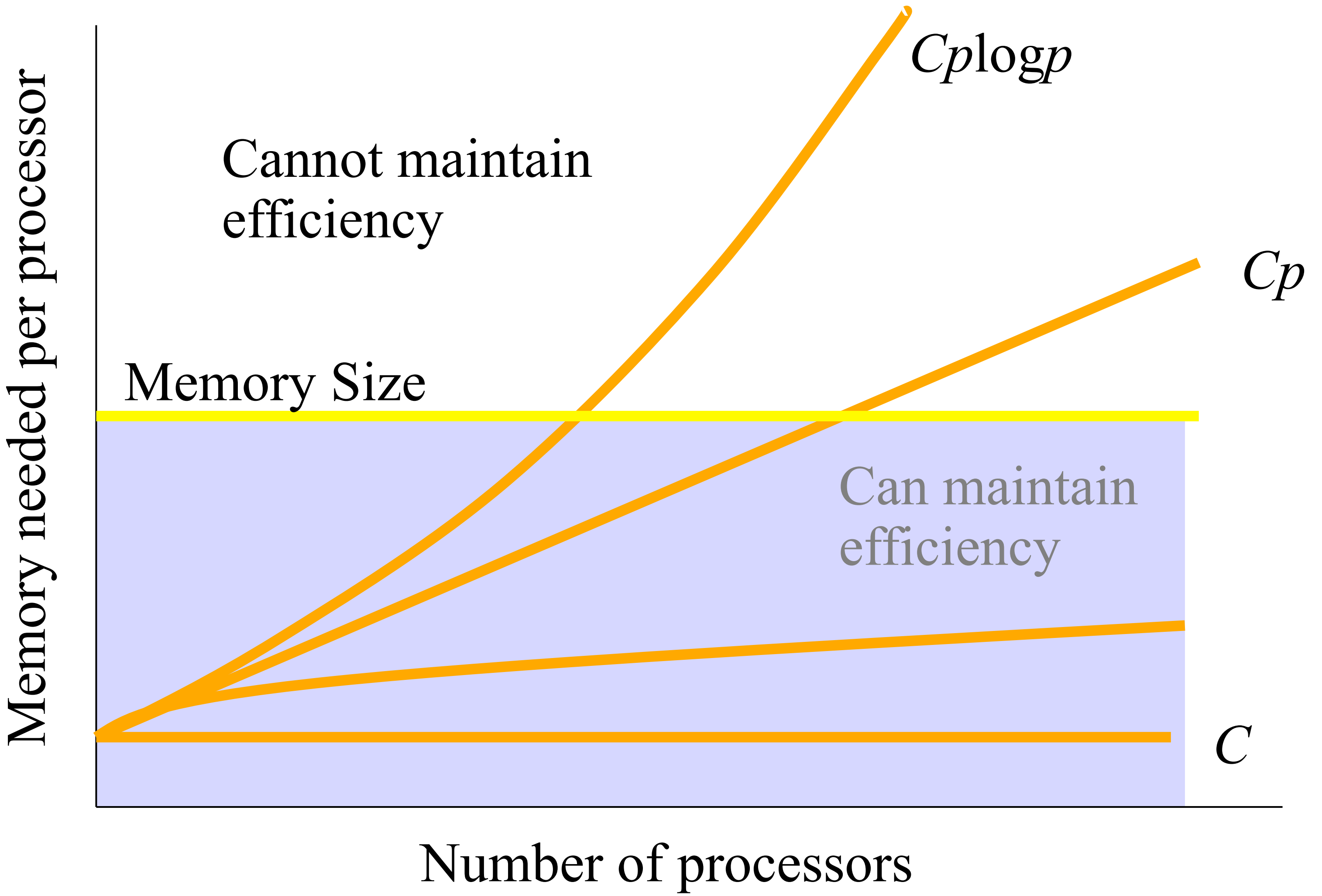
$$T(n,1) \geq T_o(n,p)$$

Scalability: Do you have enough resources(memory) to scale to that size

Maintain ratio *To(W,p) / W , overhead to parallel work constant*

Parallel programming:
Basic definitions

Scalaility: Do you have enough resources(memory) to scale to that size



Adding numbers

Each processor has n/p numbers.

4, 3, 9	1, 3, 7	7,5,8	5,2,9

Serial time	**Parallel time**	**Communicate and add**
n	n/p	$\log p + \log p$

Adding numbers

Each processor has n/p numbers.
Steps to communicate and add are 2 log p

Speedup:

$$S = \frac{n}{\left(\frac{n}{p} + 2\log p\right)}$$

$$E = \frac{n}{(n + 2p\log p)}$$

Isoefficiency

If you increase problem size n as O(p log p)
then efficiency can remain constant !

Adding numbers

Each processor has n/p numbers.
Steps to communicate and add are 2 log p

Speedup:

$$E = \frac{n}{\left(n + 2p\log p\right)}$$

Scalability

$$M(n) \geq (n/p) = p\log p/p$$
$$= \log p$$

Sorting

Each processor has n/p numbers.

4, 3, 9 | 1, 3, 7 | 7,5,8 | 5,2,9

Our plan

1. Split list into parts
2. Sort parts individually
3. Merge lists to get sorted list

Sorting

Each processor has n/p numbers.

4, 3, 9 | 1, 3, 7 | 7,5,8 | 5,2,9

Background

Bubble sort	$O(n^2)$	Stable	Exchanging
Selection sort	$O(n^2)$	Unstable	Selection
Insertion sort	$O(n^2)$	Stable	Insertion
Merge sort	$O(n\log n)$	Stable	Merging
Quick sort	$O(n\log n)$	Unstable	Partitioning

Choosing right algorithms:
Optimal serial and parallel

Case Study: Bubble sort

Main loop

For i : 1 to length_of(A) -1

Secondary loop

For j : i+1 to length_of(A)

Compare and swap
so smaller element is to left

if (A[j] < A[i]) swap(A[i], A[j])

[5 1 4 2]

Choosing right algorithms:
Optimal serial and parallel

Case Study: Bubble sort

Main loop

For i : 1 to length_of(A) -1

Secondary loop

For j : i+1 to length_of(A)

Compare and swap
so smaller element is to left

if (A[j] < A[i]) swap(A[i], A[j])

Compare		Swap
(i=1, j=2)	[5 1 4 2]	Y
(i=1, j=3)	[1 5 4 2]	N
(i=1, j=4)	[1 5 4 2]	N
(i=2, j=3)	[1 5 4 2]	Y
(i=2, j=4)	[1 4 5 2]	Y
(i=3, j=4)	[1 2 5 4]	Y

Choosing right algorithms:
Optimal serial and parallel

Case Study: Bubble sort

Main loop

For i : 1 to length_of(A) -1

Secondary loop

For j : i+1 to length_of(A)

Compare and swap so smaller element is to left

if (A[j] < A[i]) swap(A[i], A[j])

$N(N-1)/2 = O(N^2)$

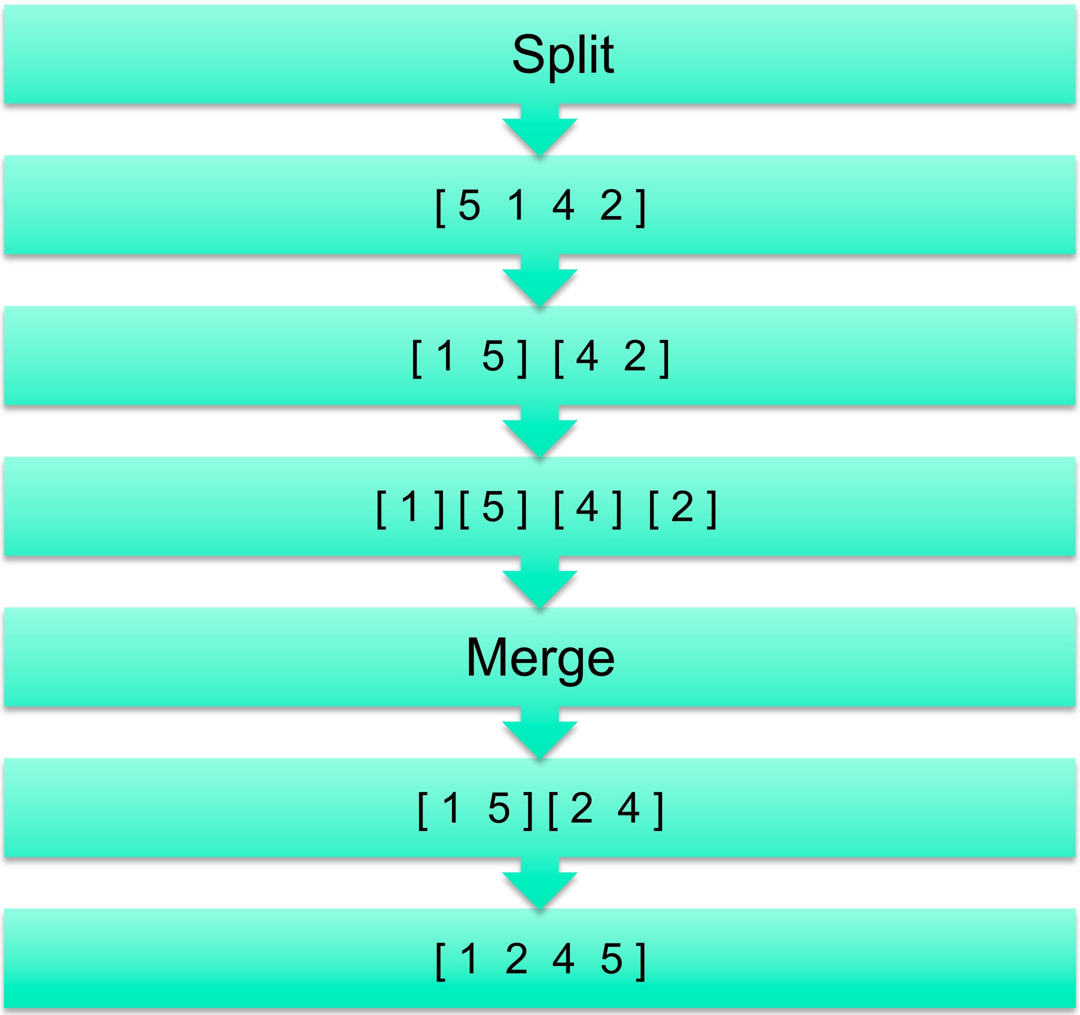
Comparisons

Compare		Swap
(i=1, j=2)	[5 1 4 2]	Y
(i=1, j=3)	[1 5 4 2]	N
(i=1, j=4)	[1 5 4 2]	N
(i=2, j=3)	[1 5 4 2]	Y
(i=2, j=4)	[1 4 5 2]	Y
(i=3, j=4)	[1 2 5 4]	Y

Choosing right algorithms:
Optimal serial and parallel

Case Study: Merge sort

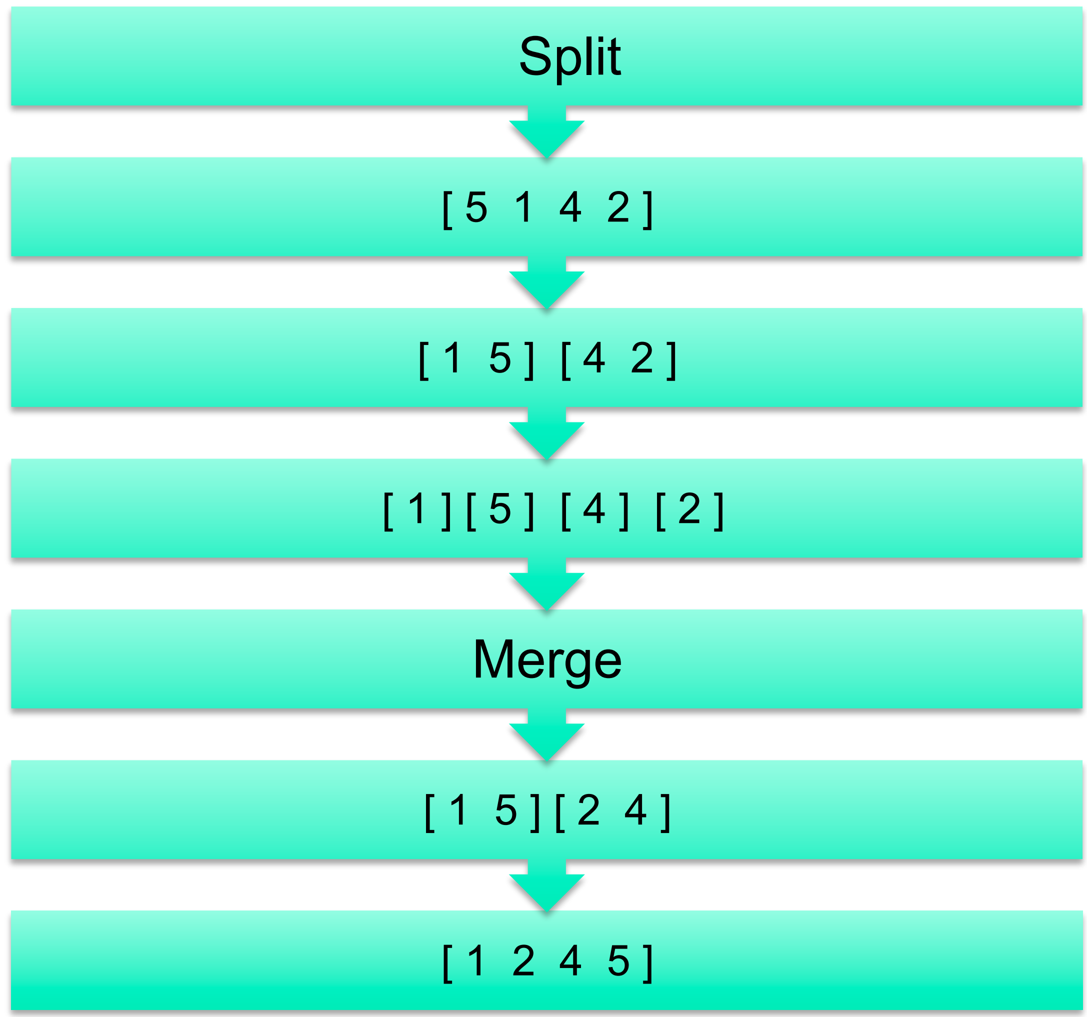
Recursively merge lists having one element each



Choosing right algorithms:
Optimal serial and parallel

Case Study: Merge sort

Recursively merge lists having one element each



Best sorting algorithms need

O(N log N)

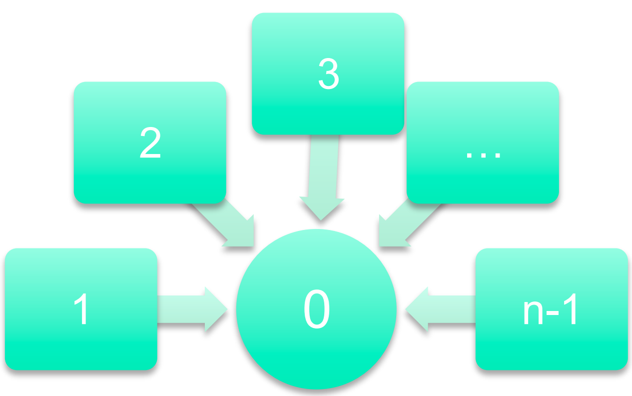
Choosing right algorithms:
Parallel sorting technique

Merging : Merge p lists having n/p elements each

Sub-optimally :

Pop-push merge on 1 processor

$$O(np)$$



[1 3 5 6] => [1]
[2 4 6 8]

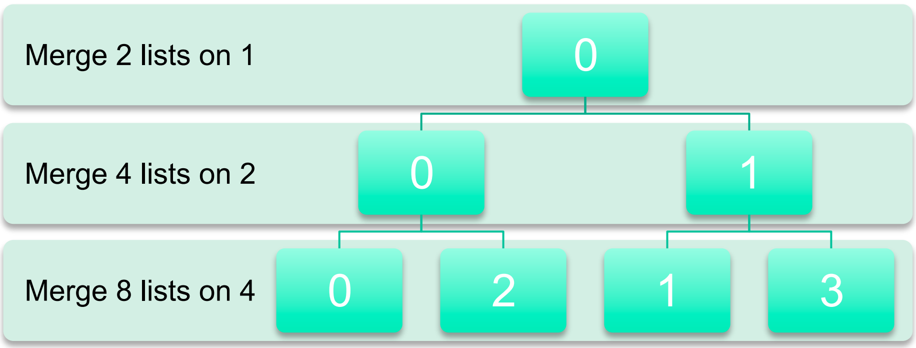
[3 5 6] => [1 2]
[2 4 6 8]

[3 5 6] => [1 2 3]
[4 6 8]

Choosing right algorithms:
Parallel sorting technique

Merging : Merge p lists having n/p elements each

Optimal: Recursively or tree based



Best merge algorithms need

$$O(n \log p)$$

Sorting

Each processor has n/p numbers.

4, 3, 9	1, 3, 7	7,5,8	5,2,9

Serial time

$$n \log n$$

Parallel time

$$\frac{n}{p} \log\left(\frac{n}{p}\right)$$

Merge time

$$(n-1).\frac{n}{p}$$

$$\frac{n}{p} + n \log p$$

Sorting

Each processor has n/p numbers.

4, 3, 9	1, 3, 7	7,5,8	5,2,9

Overhead

$$= n\log n - p\big((n/p)\log(n/p) + n\log p\big)$$

$$\sim p\log p + np\log p$$

$$\sim n\log p$$

Isoefficiency

$$n\log n \ \geq c(n\log p)$$

$$\Rightarrow n \geq p^c$$

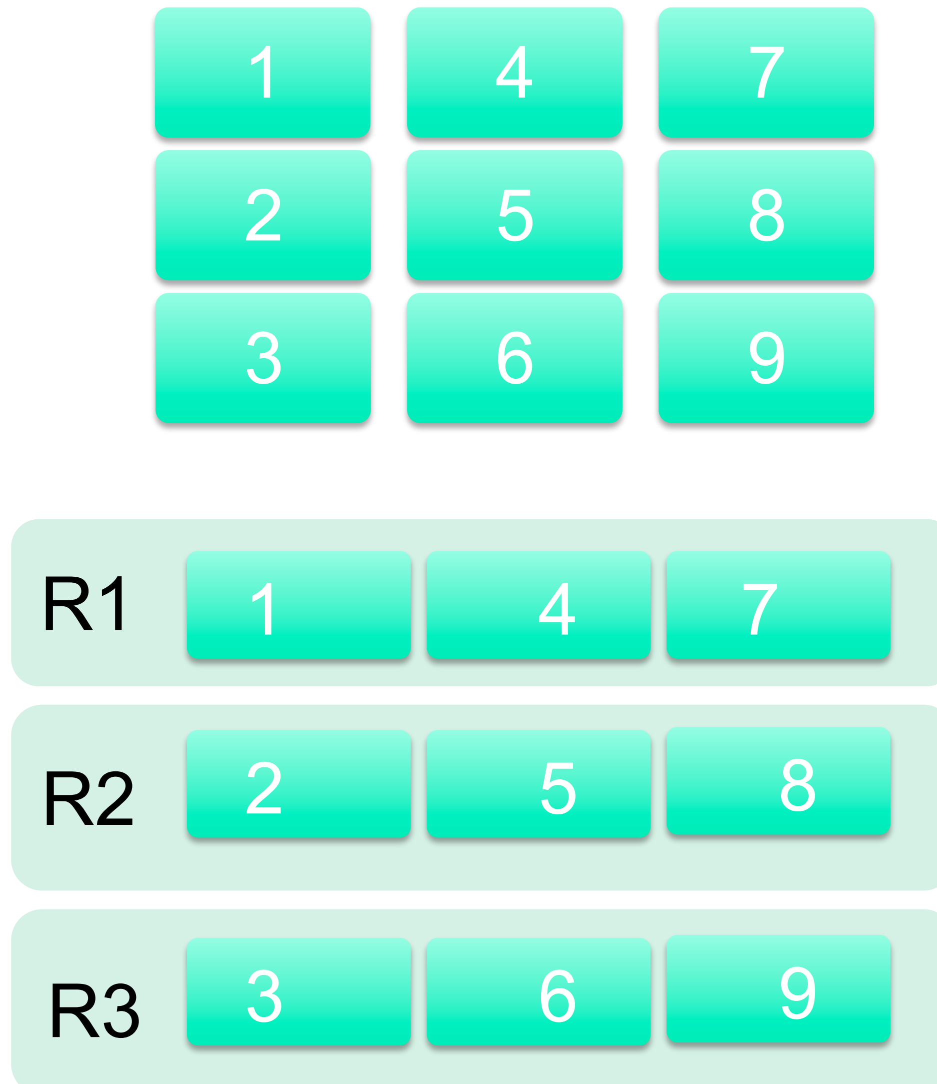
Scalability

$$= n/p = p^{c-1}$$

Low for c>2

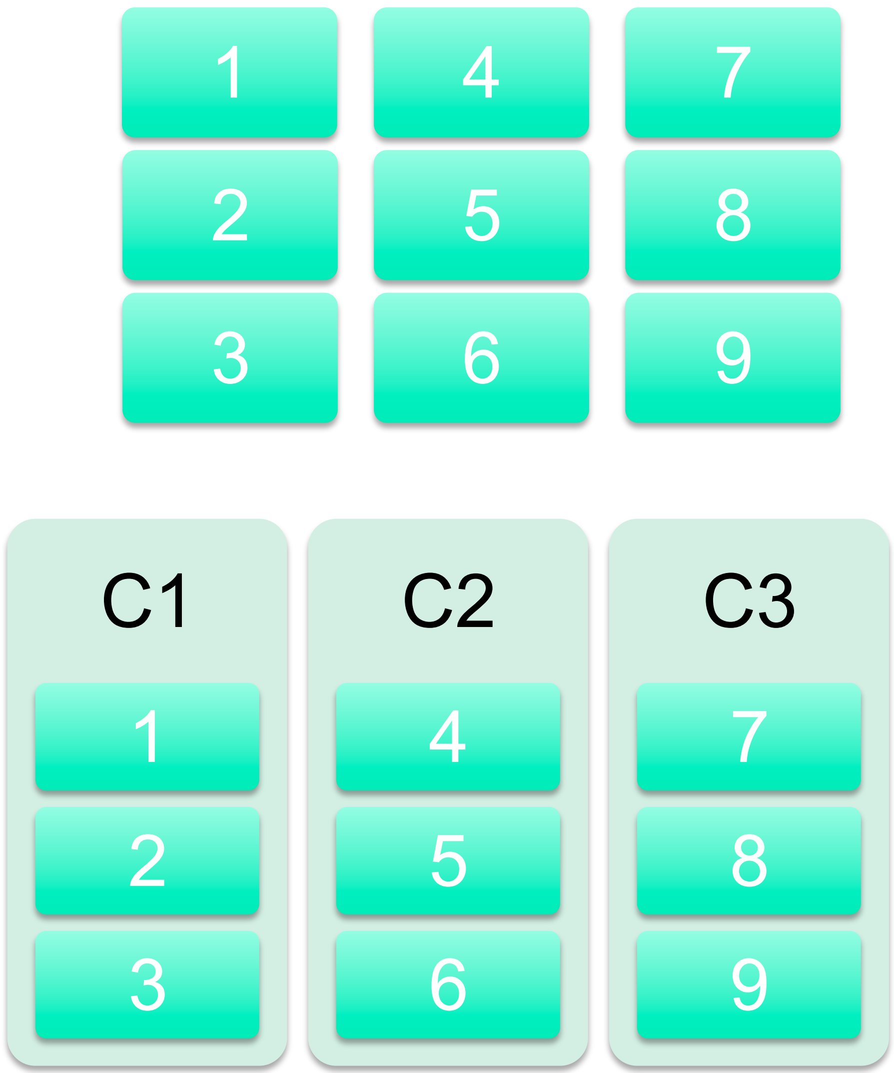
Data decomposition

1D : Row-wise



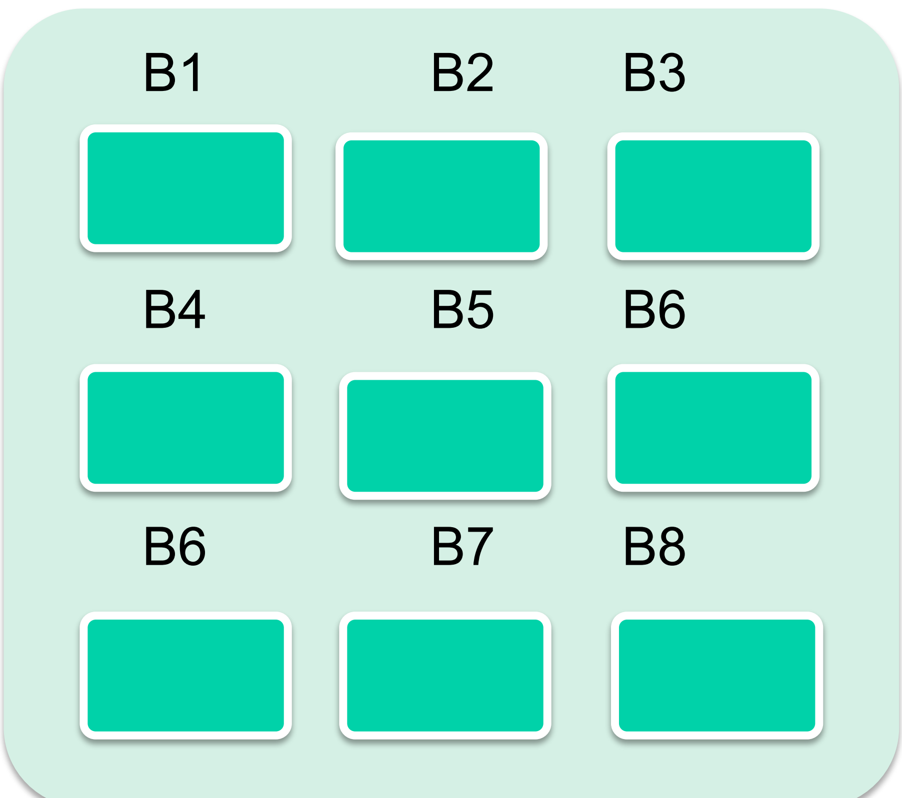
Data decomposition

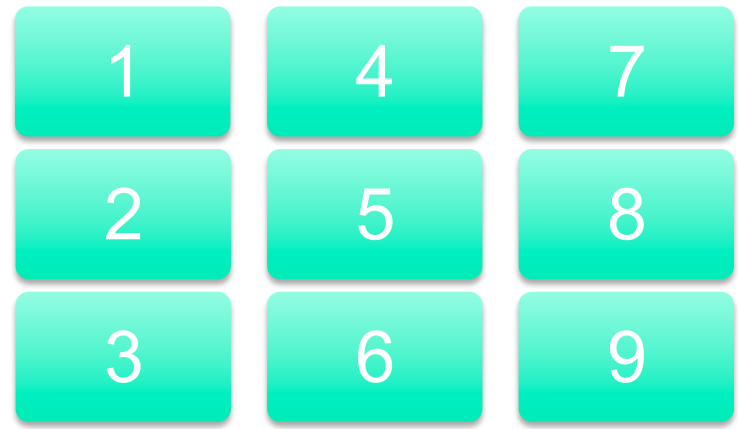
1D : Column-wise



Data decomposition

2D : Block-wise





Data decomposition

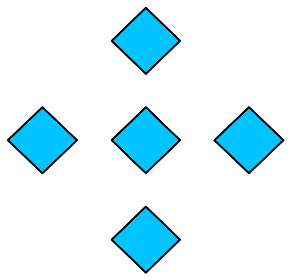
Laplace solver: (n x n) mesh with p processors

Time to communicate *1* cell:

$$t_{cell}^{comm} = \tau_s + t_w$$

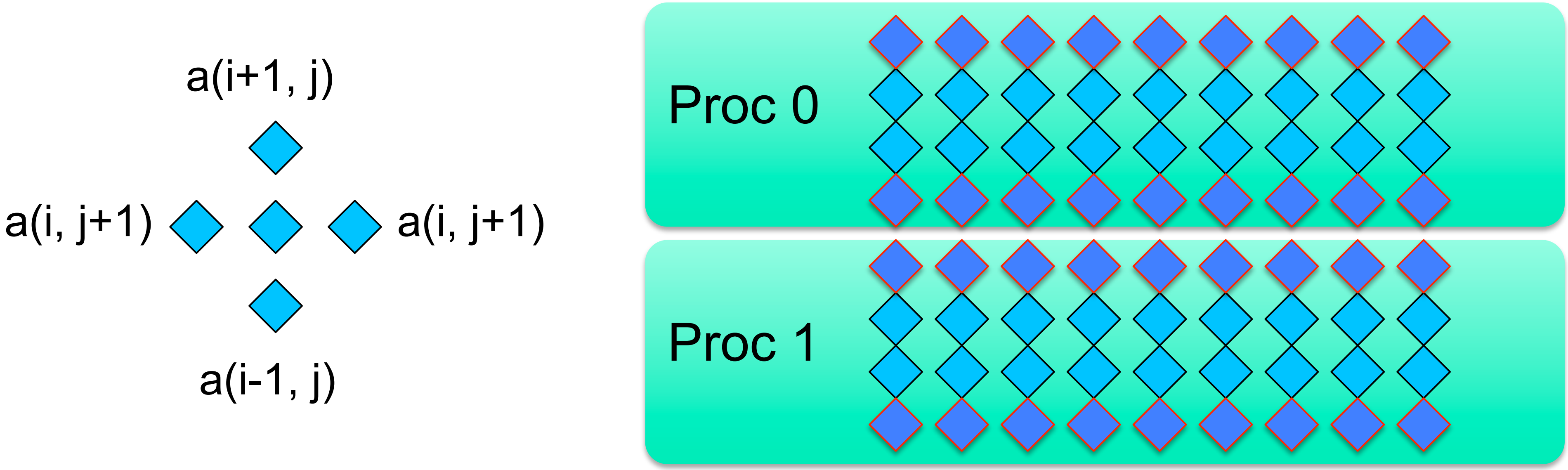
Time to evaluate stencil once:

$$t_{cell}^{comp} = 5 * (t_{float})$$



Data decomposition

Laplace solver: 1D Row-wise (n x n) with p processors



Data decomposition

Laplace solver: 1D Row-wise (n x n) with p processors

Serial time:

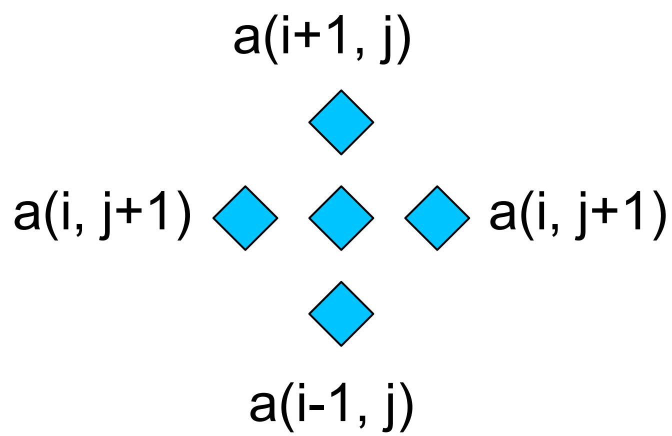
$$t_{seq}^{comp} = n^2 t_{cell}^{comm}$$

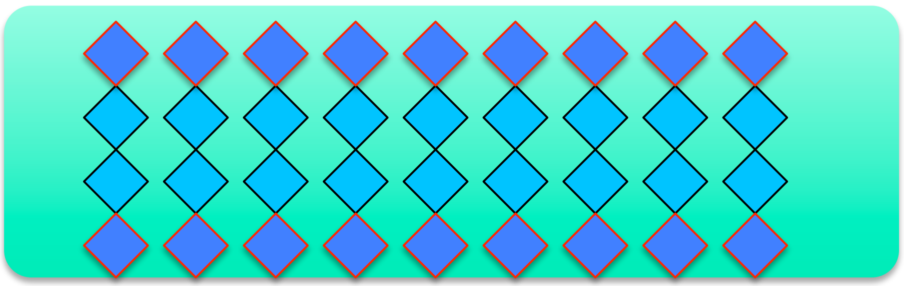
Parallel computation time

$$t_{process}^{comp} = \frac{n^2}{p} t_{cell}^{comp}$$

Ghost communication:

$$t^{comm} = 2n t_{cell}^{comm}$$





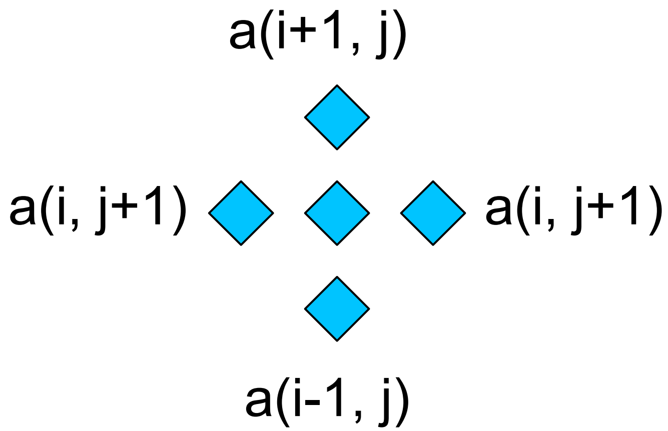
Data decomposition

Laplace solver: 1D Row-wise (n x n) with p processors

Overhead:

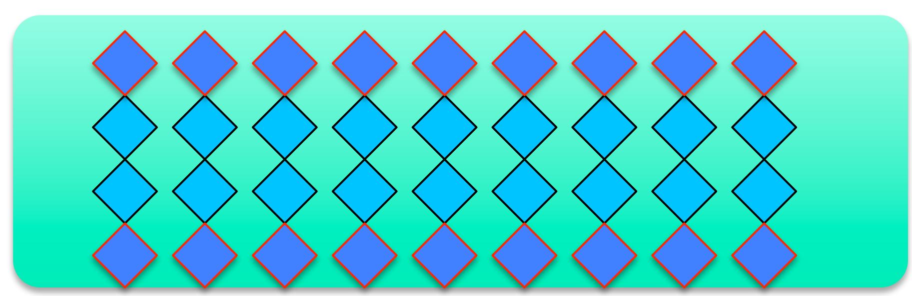
$$= t_{seq} - pt_p$$

$$= pn$$



Isoefficieny:

$$n^2 \geq cnp \Rightarrow n \geq cp$$



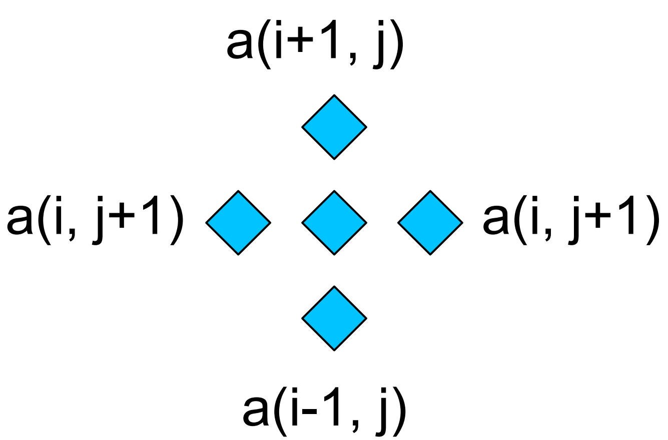
Scalability :

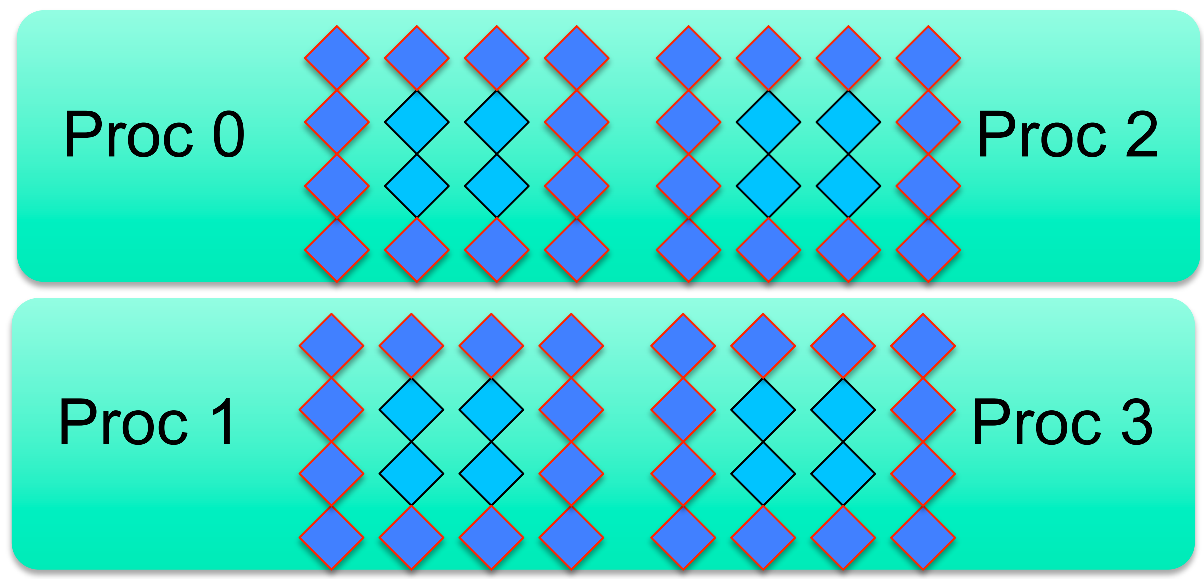
$$= c^2 p^2 / p = Cp$$

Poor Scaling

Data decomposition

Laplace solver: 2D Block-wise





Data decomposition

Laplace solver: 2D Row-wise (n x n) with p processors

Serial time:

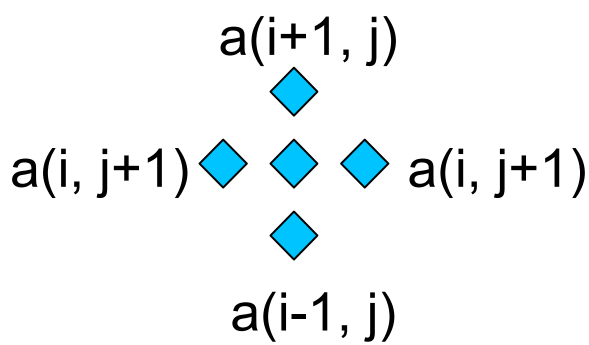
$$t_{seq}^{comp} = n^2 t_{cell}^{comm}$$

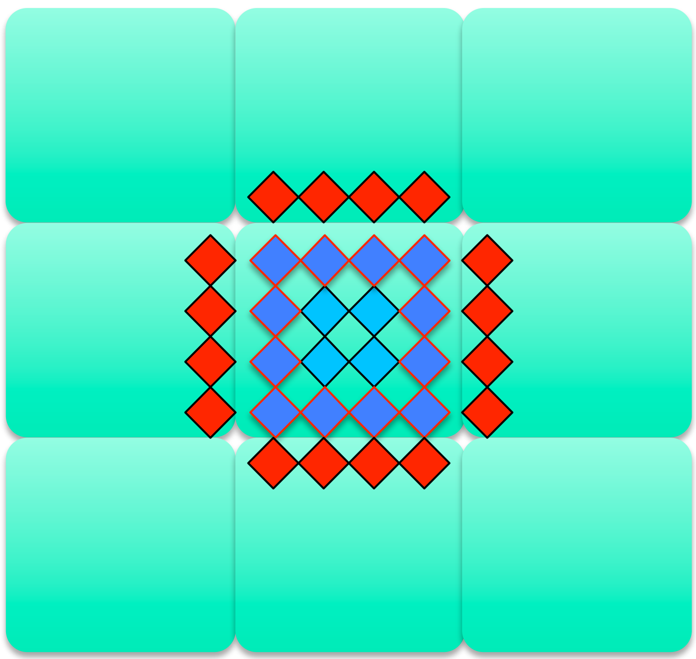
Parallel computation time:

$$t_{process}^{comp} = \frac{n^2}{p} t_{cell}^{comp}$$

Ghost communication:

$$t^{comm} = \frac{4n}{\sqrt{p}} t_{cell}^{comm}$$





Data decomposition

Laplace solver: 2D Row-wise (n x n) with p processors

Overhead:

$$= p.n / \sqrt{p}$$

$$= n\sqrt{p}$$

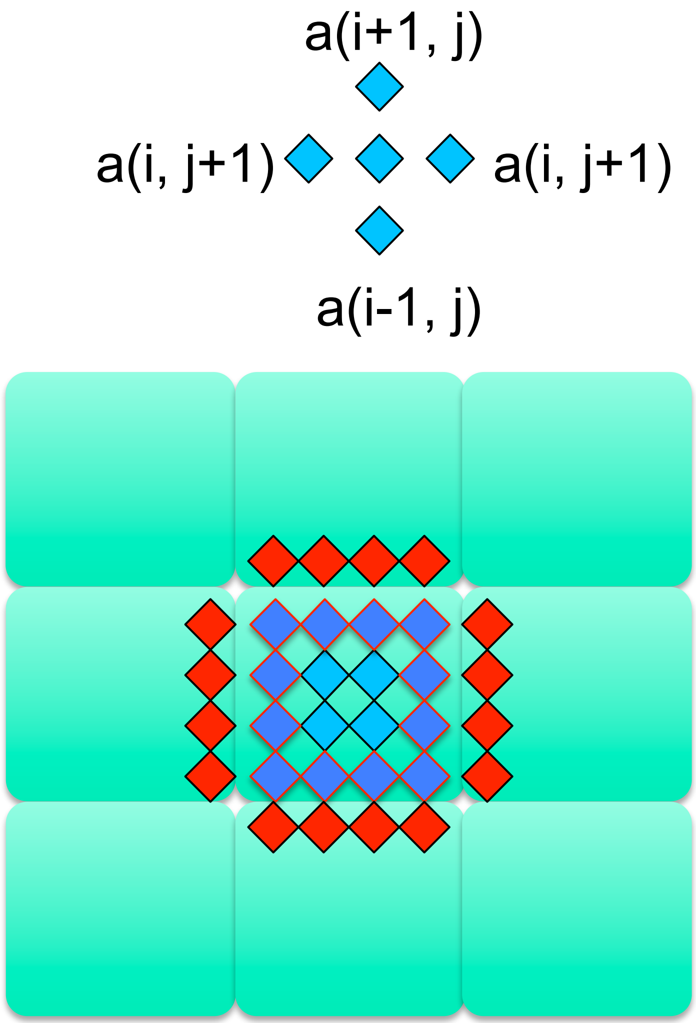
Isoefficiency:

$$n \sim \sqrt{p}$$

Scalability :

$$= (c\sqrt{p})^2 / p$$

$$= C$$ Perfect Scaling



Data decomposition

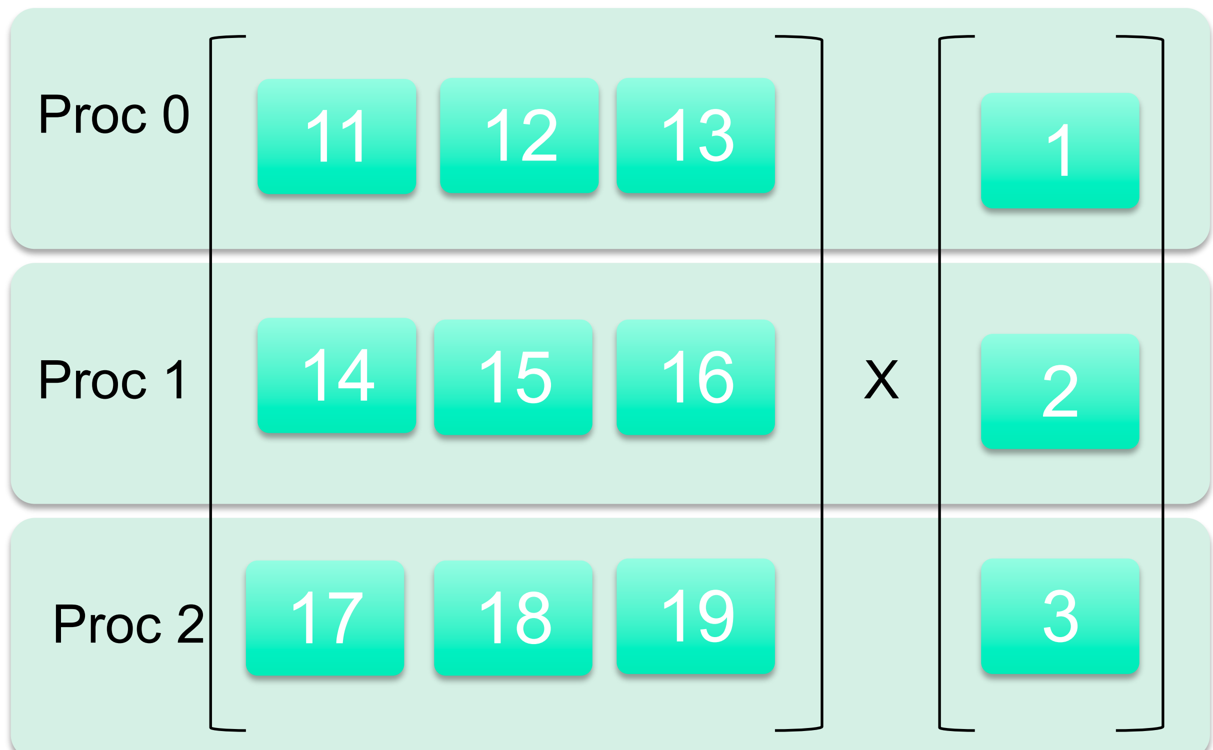
Matrix vector multiplication: 1D row-wise decomposition

Computation:
Each processor computes n/p elements,
n multiplies + (n-1) adds for each

$$O\left(\frac{n^2}{p}\right)$$

Communication:
All gather in the end so each processor has full copy of output vector

$$\log p + \sum_{i=1}^{\log p} 2^{i-1} . \frac{n}{p} = \log p + \frac{n(p-1)}{p}$$



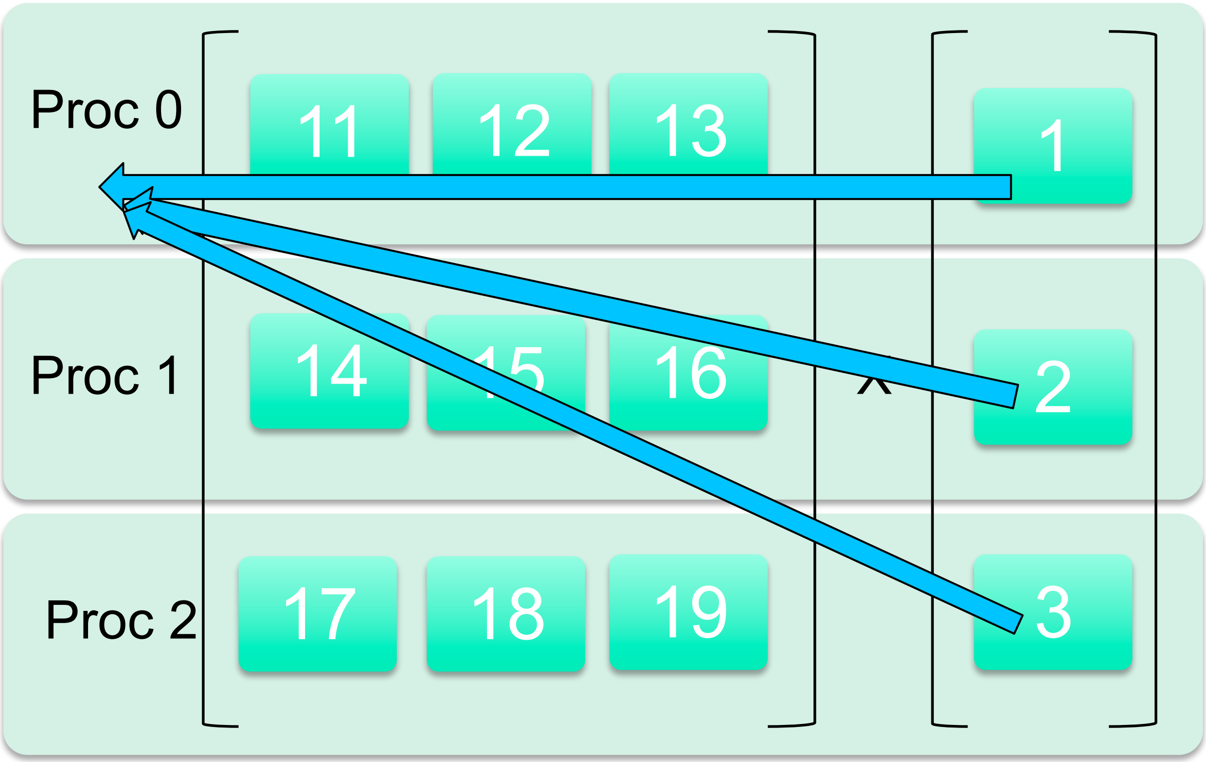
Data decomposition

Matrix vector multiplication: 1D row-wise decomposition

Algorithm:

1. Collect vector using MPI_Allgather
2. Local matrix multiplication to get output vector

Wastes much memory



Data decomposition

Matrix vector multiplication: 1D row-wise decomposition

Computation:
Each processor computes n/p elements,
n multiplies + (n-1) adds for each

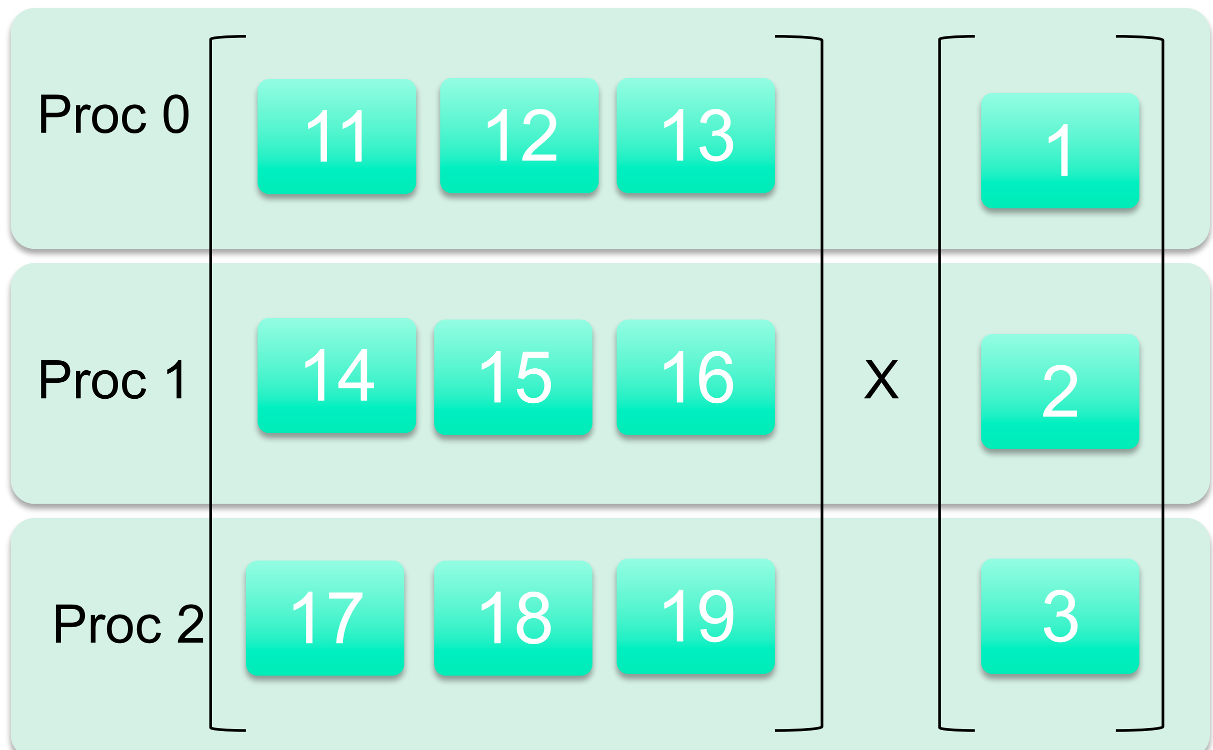
$$O\left(\frac{n^2}{p}\right)$$

Communication:
All gather in the end so each processor has full copy of output vector

$$\tau_w n + \tau_s \log p$$

Overhead:

$$\tau_s p \log p + \tau_w np$$



Data decomposition

Matrix vector multiplication: 1D row-wise decomposition

Speedup:

$$S = \frac{p}{1 + \left(\frac{p(\tau_s \log p + t_w n)}{t_c n^2} \right)}$$

Isoefficiency:

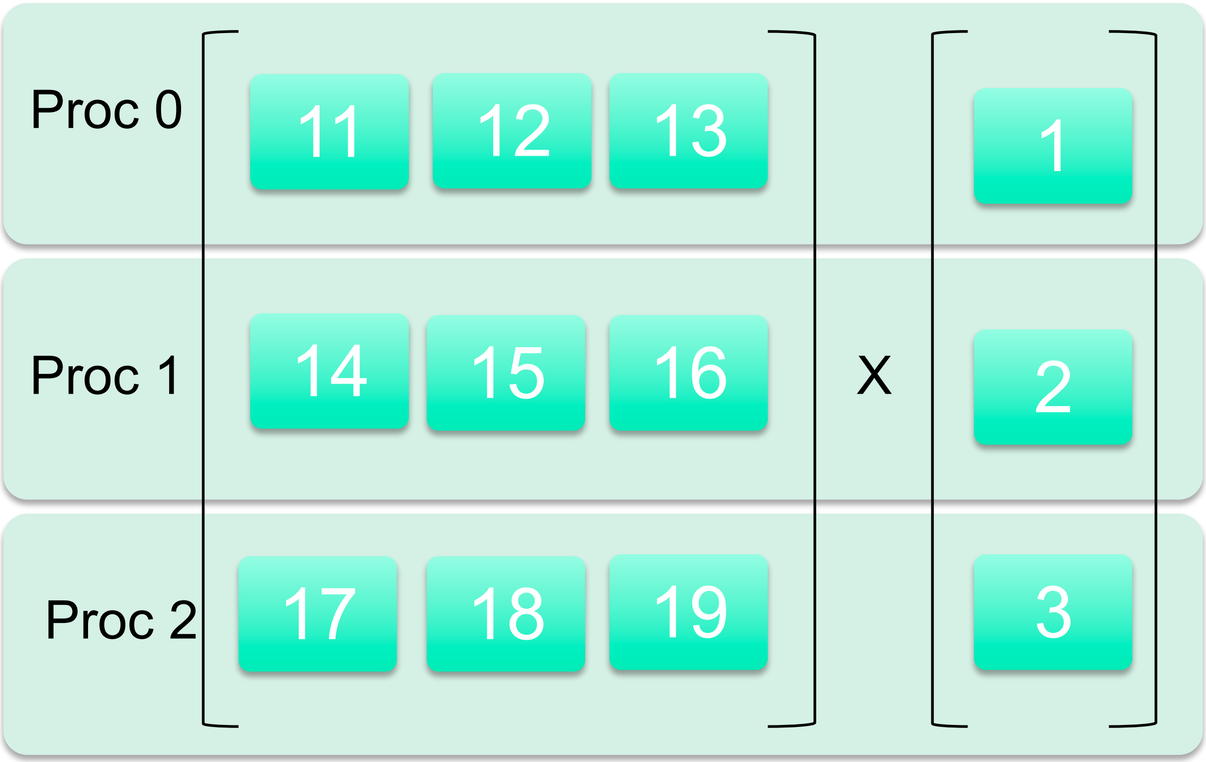
$$n^2 \sim p \log p + np$$

$$\Rightarrow n \geq cp$$

Scalability:

$$M(p) \geq n^2 / p = c^2 p$$

Not scalable !



Data decomposition

Matrix vector multiplication: 1D column-wise decomposition

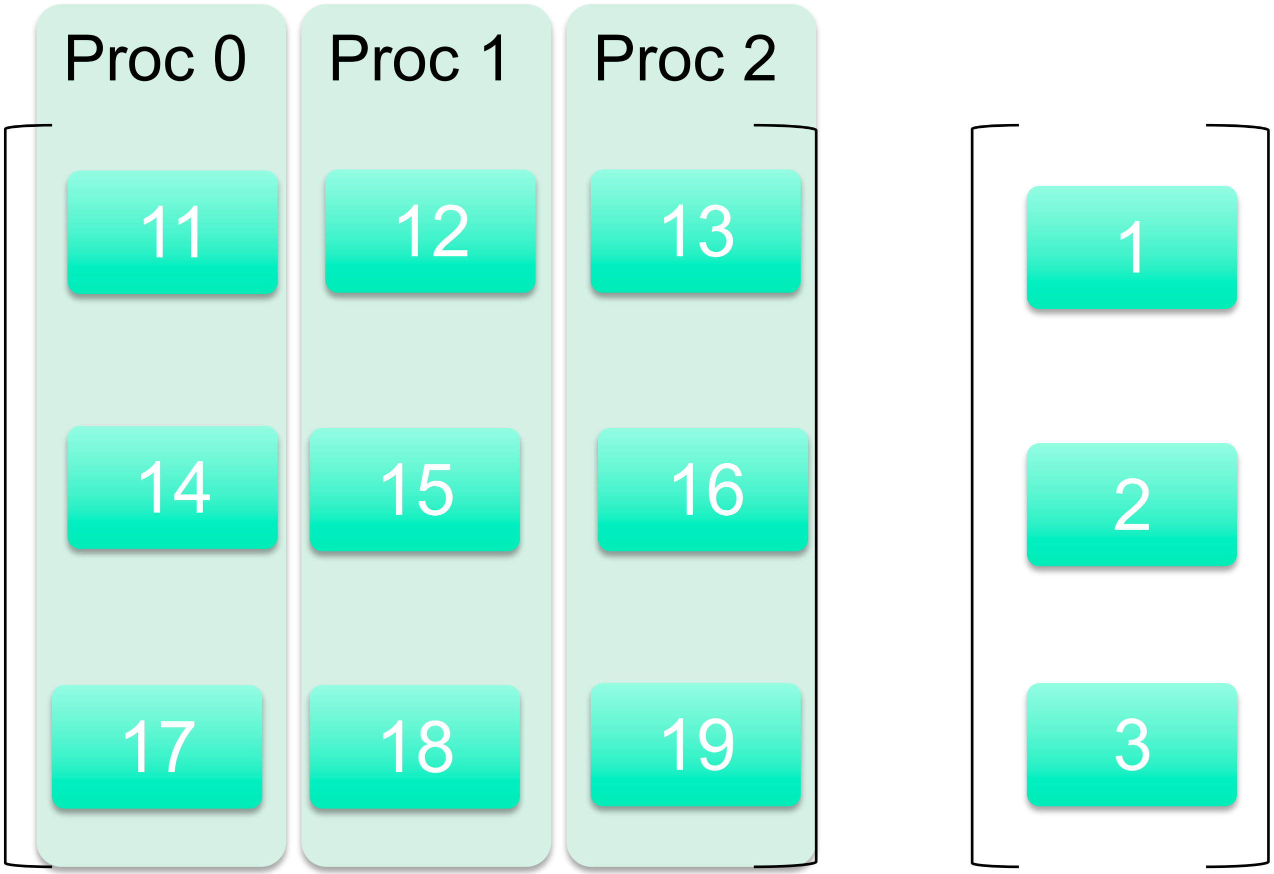
Serial Computation?

Parallel Computation?

Overhead?

Isoefficiency?

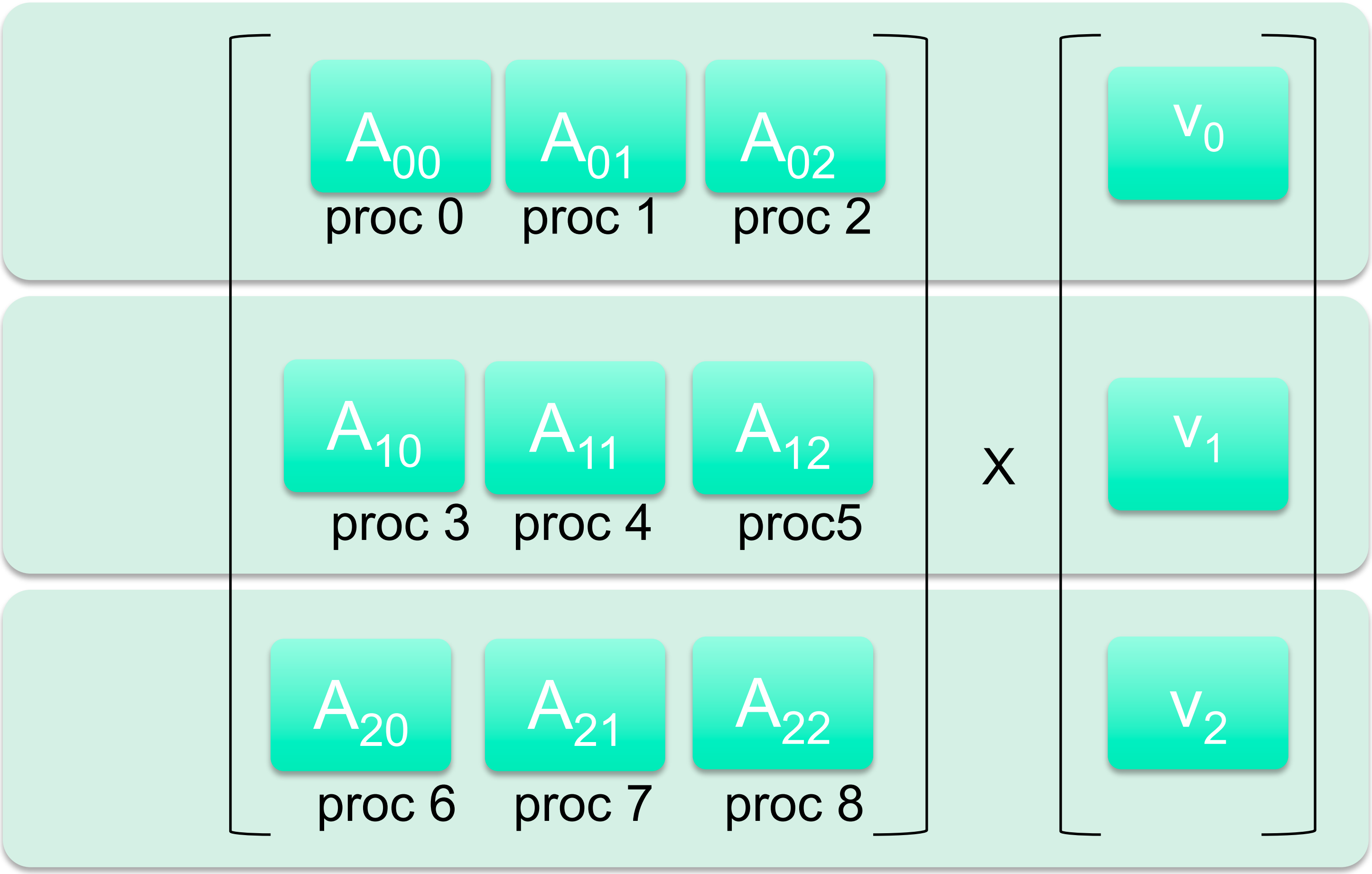
Scalability?



Data decomposition

Matrix vector multiplication: 2D decomposition

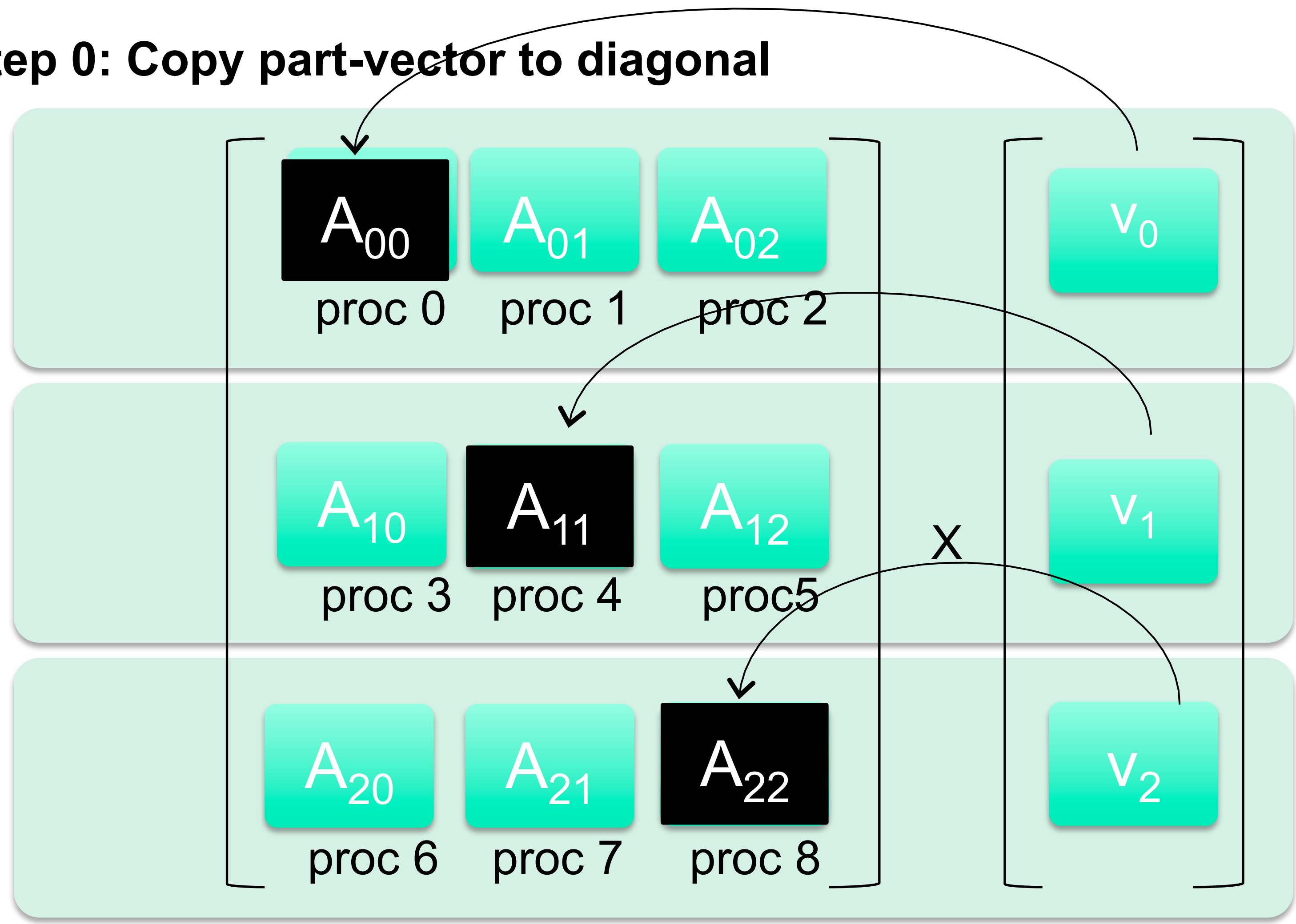
Algorithm:
Uses p Processors on a grid



Data decomposition

Matrix vector multiplication: 2D decomposition

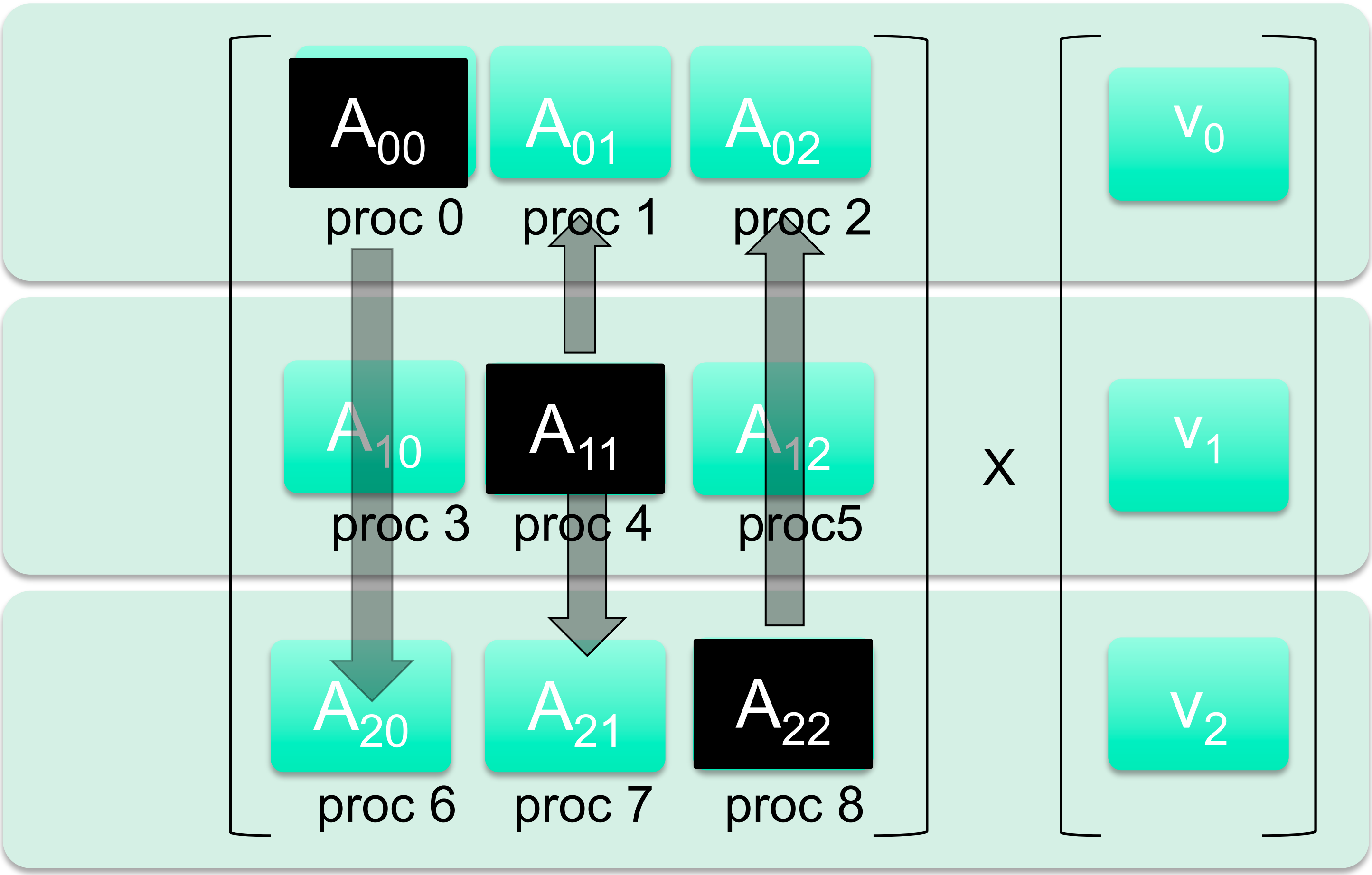
Algorithm Step 0: Copy part-vector to diagonal



Data decomposition

Matrix vector multiplication: 2D decomposition

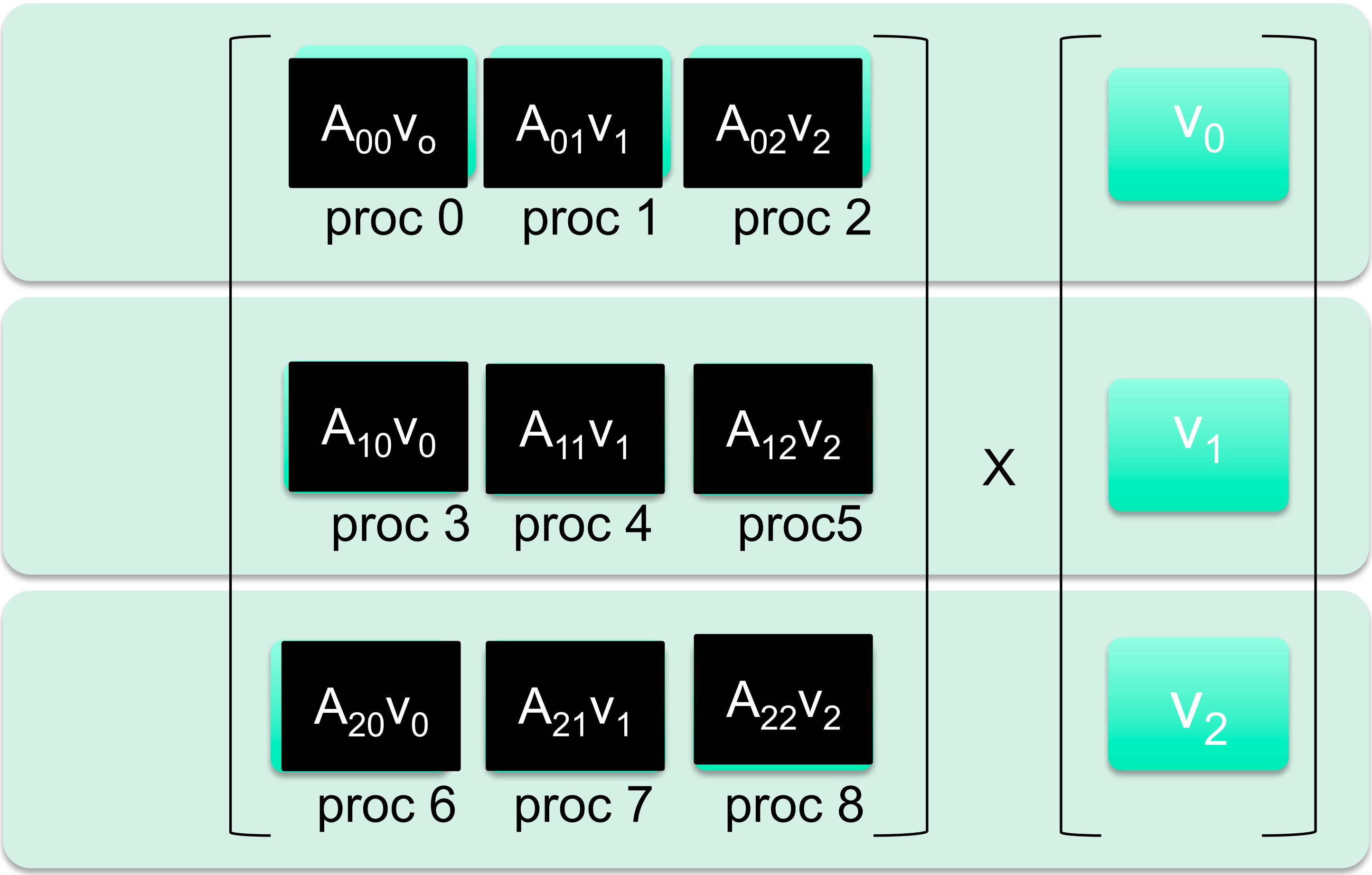
Algorithm Step 1: Broadcast vector along columns



Data decomposition

Matrix vector multiplication: 2D decomposition

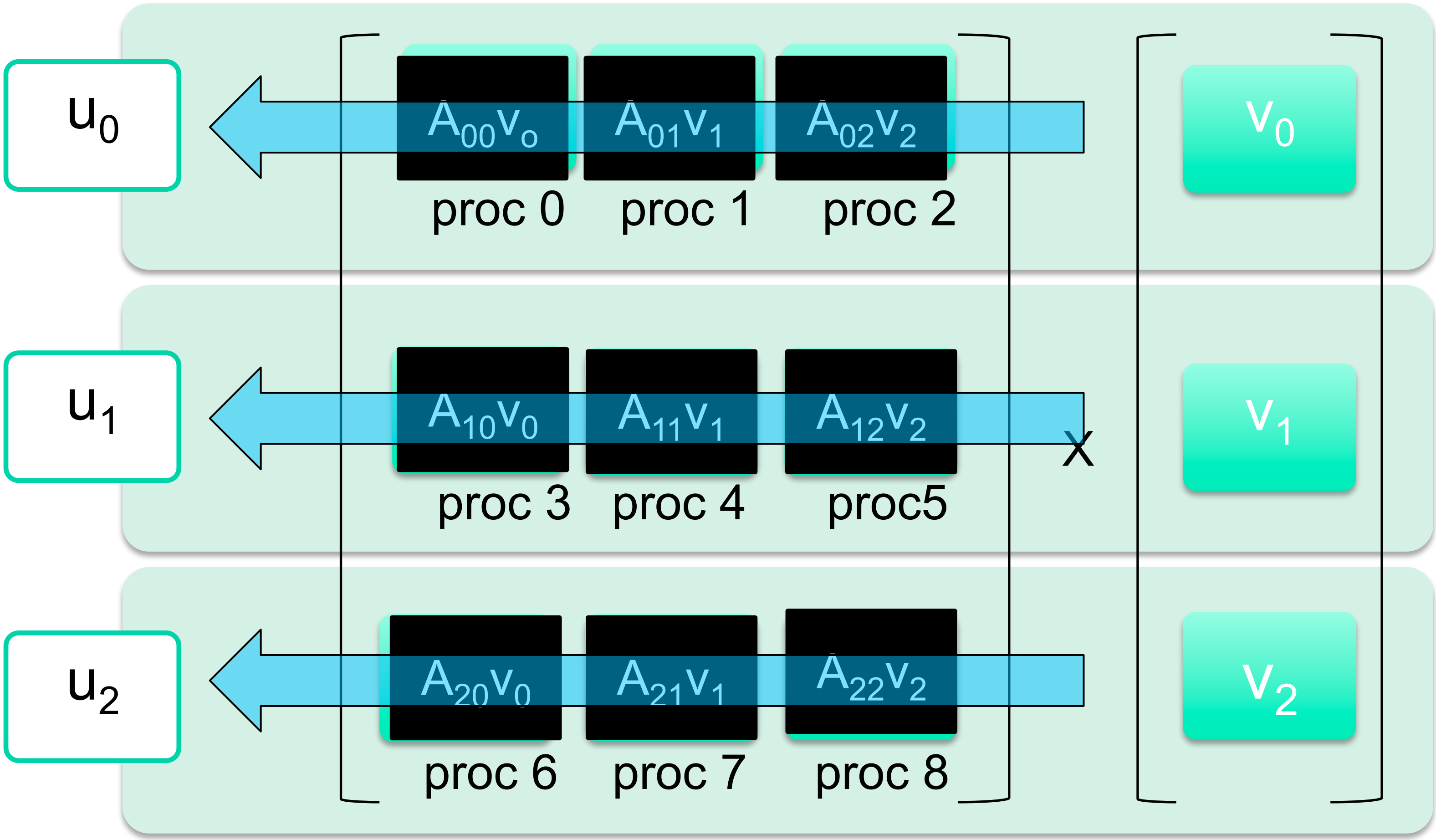
Algorithm Step 2: Local computation on each processor



Data decomposition

Matrix vector multiplication: 2D decomposition

Algorithm Step 3: Reduce across rows



Data decomposition

Matrix vector multiplication: 2D decomposition

Computation:
Each processor computes n^2/p elements,

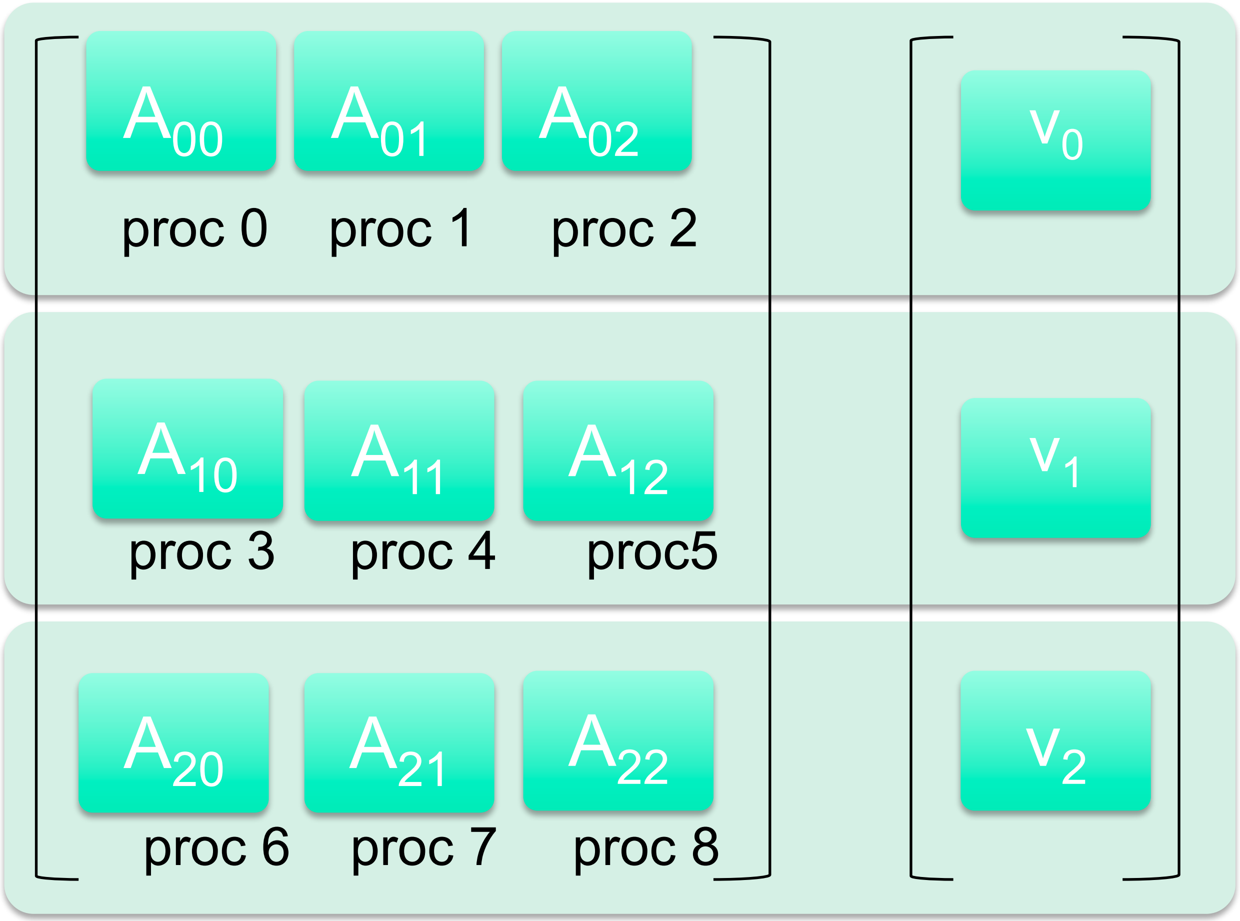
$$\tau_c\left(\frac{n^2}{p}\right)$$

Communication:

$$\frac{2n}{\sqrt{p}}\log(\sqrt{p})+\frac{n}{\sqrt{p}}\sim\frac{n\log p}{\sqrt{p}}$$

Overhead:

$$n\sqrt{p}\log(p)$$



Data decomposition

Matrix vector multiplication: 2D decomposition

Isoefficiency:

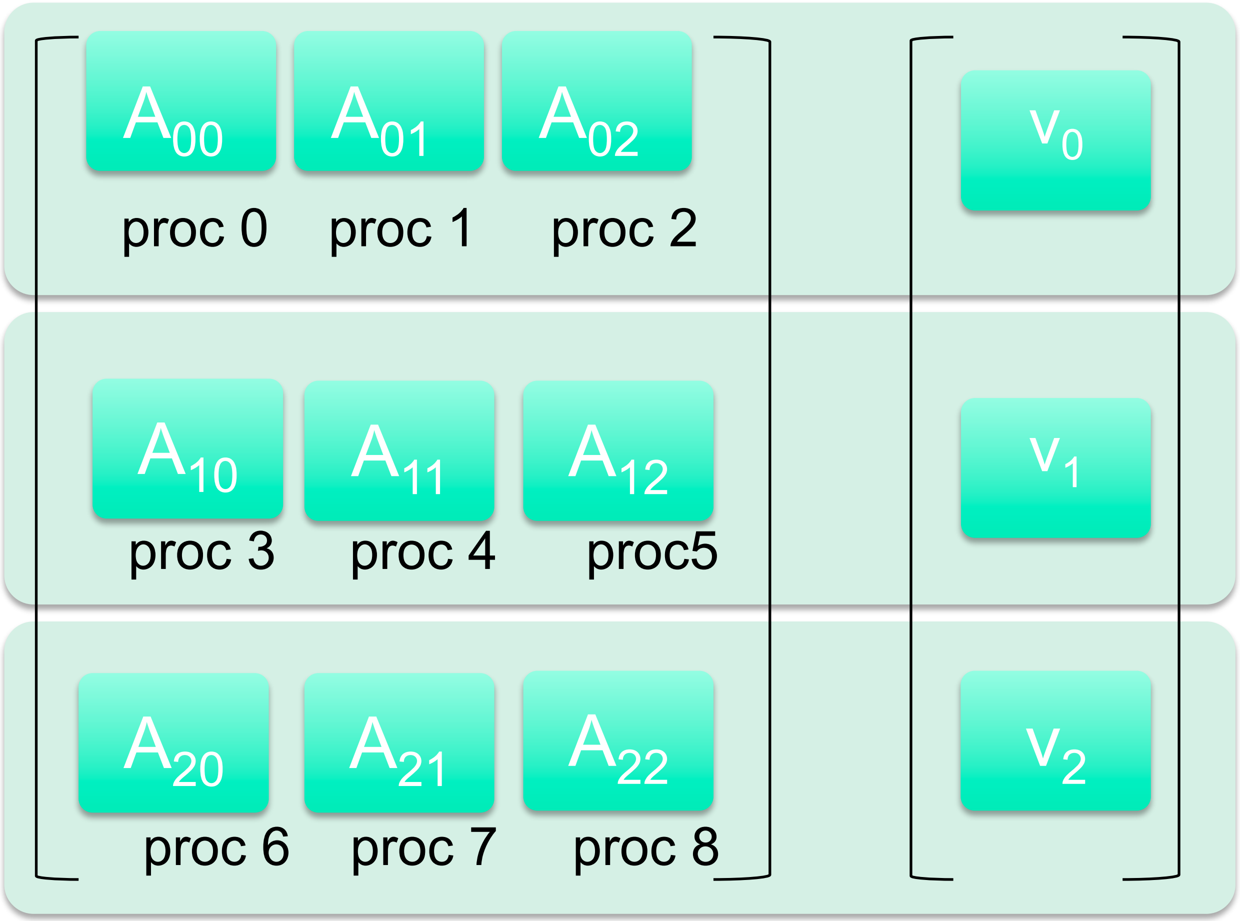
$$n^2 \sim n\sqrt{p}\log p$$

$$\Rightarrow n \geq c\sqrt{p}\log p$$

Scalability:

$$M(p) \geq \frac{n^2}{p} = (\log p)^2$$

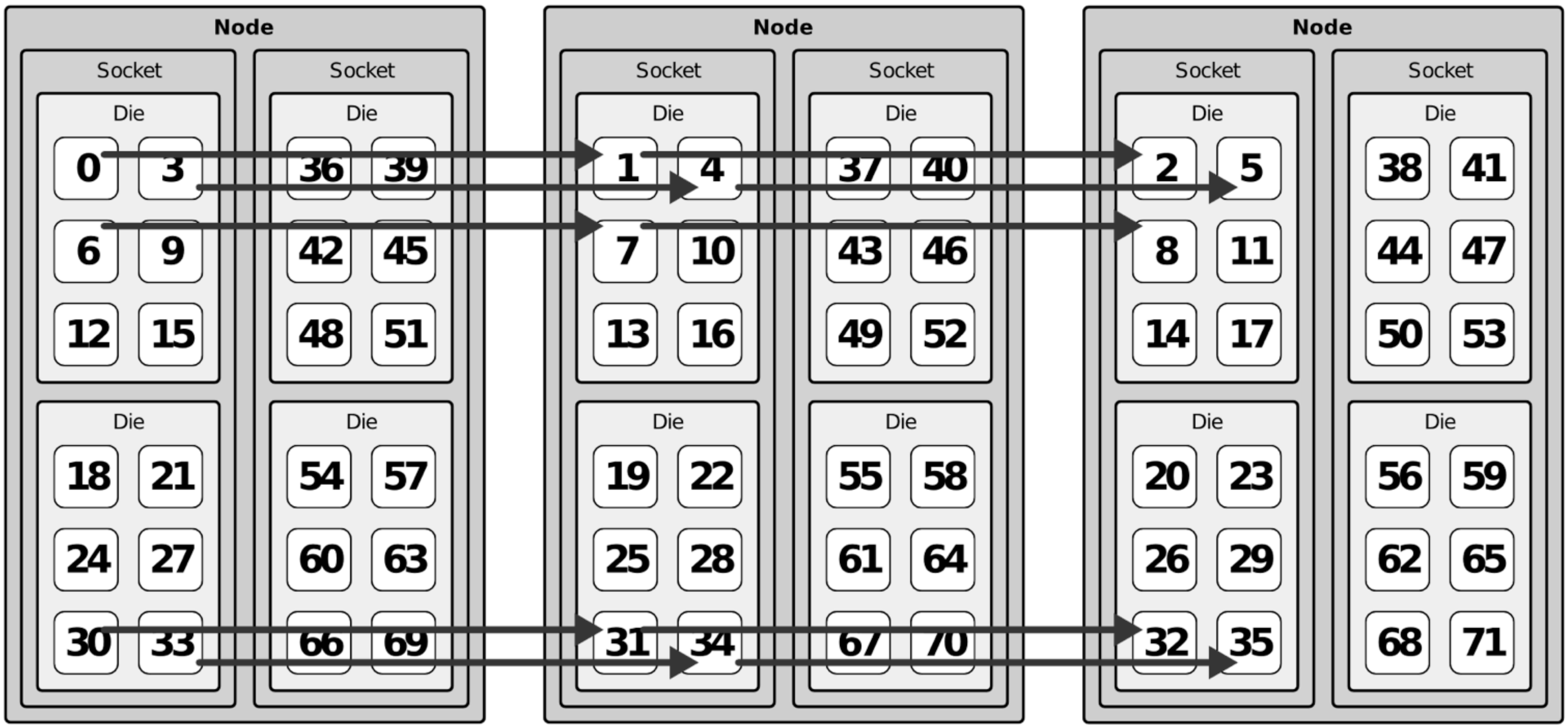
Scales better than 1D !

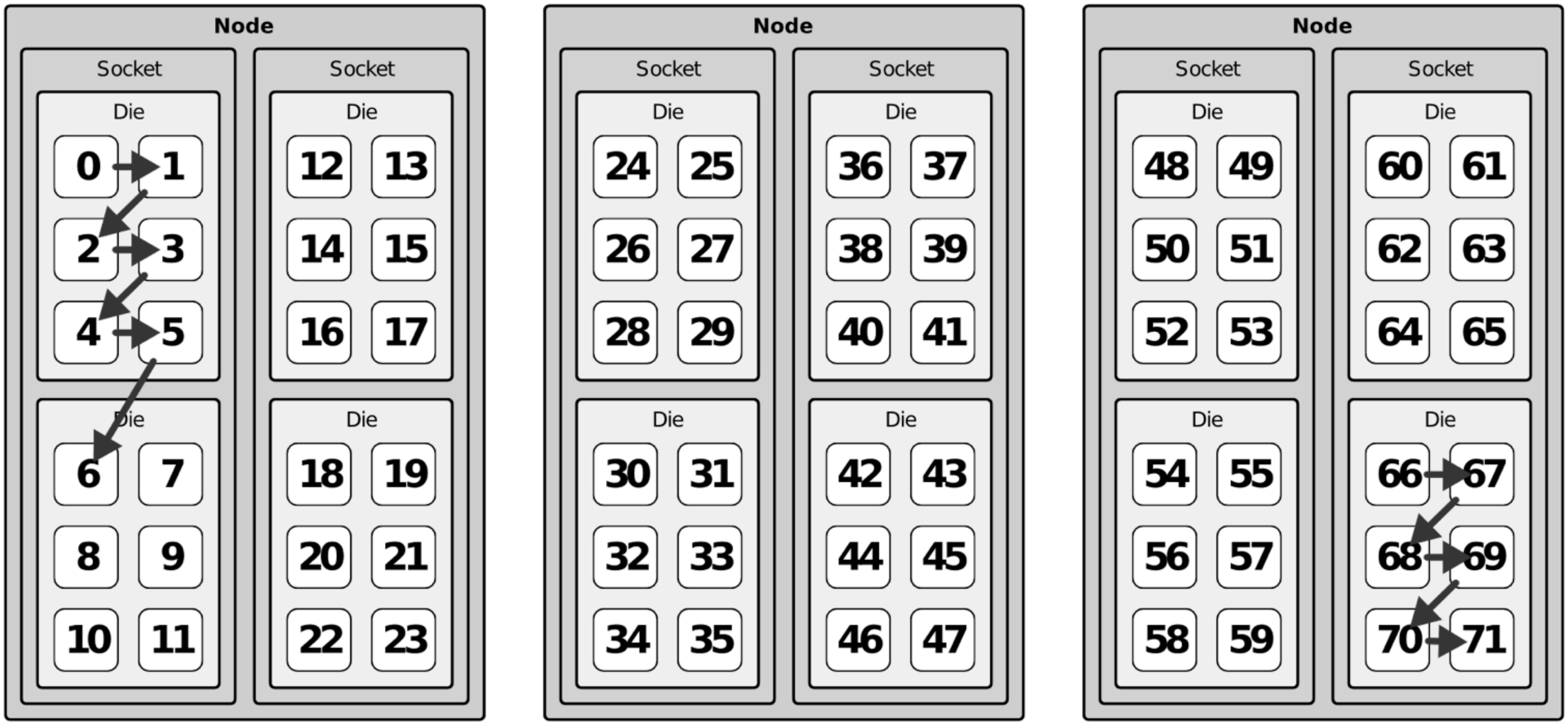


Lets look at the code

Summary

- Know your algorithm !
- Don’t expect the unexpected !
- Pay attention to parallel design and implementation right from the outset. It will save you lot of labor.


Node
Socket
Die
0
3
6
9
12
15
Die
18
21
24
27
30
33
Socket
Die
36
39
42
45
48
51
Die
54
57
60
63
66
69
Node
Socket
Die
1
4
7
10
13
16
Die
19
22
25
28
31
34
Socket
Die
37
40
43
46
49
52
Die
55
58
61
64
67
70
Node
Socket
Die
2
5
8
11
14
17
Die
20
23
26
29
32
35
Socket
Die
38
41
44
47
50
53
Die
56
59
62
65
68
71


Node
Socket
Die
0
1
2
3
4
5
Die
6
7
8
9
10
11
Socket
Die
12
13
14
15
16
17
Die
18
19
20
21
22
23
Node
Socket
Die
24
25
26
27
28
29
Die
30
31
32
33
34
35
Socket
Die
36
37
38
39
40
41
Die
42
43
44
45
46
47
Node
Socket
Die
48
49
50
51
52
53
Die
54
55
56
57
58
59
Socket
Die
60
61
62
63
64
65
Die
66
67
68
69
70
71