



# Performance Engineering for Linear Solvers

Half-Day Tutorial

ISC High Performance 2024

Christie L. Alappat, Erlangen National High Performance Computing Center

Jonas Thies, TU Delft

Georg Hager, Erlangen National High Performance Computing Center

Hartwig Anzt, TU München

<https://go-nhr.de/PELS-ISC24>



# Tutorial Agenda

- Brief introduction to node-level computer architecture
- Performance modeling with the Roofline model
- Sparse matrix-vector multiplication (SpMV) performance and sparse-matrix data formats, and Roofline modeling of SpMV
- The Conjugate Gradient (CG) algorithm
- Preconditioning and preconditioned CG (PCG)
- Accelerating matrix power kernels (MPK) by cache blocking
- Optional: distributed-memory SpMV and MPK cache blocking



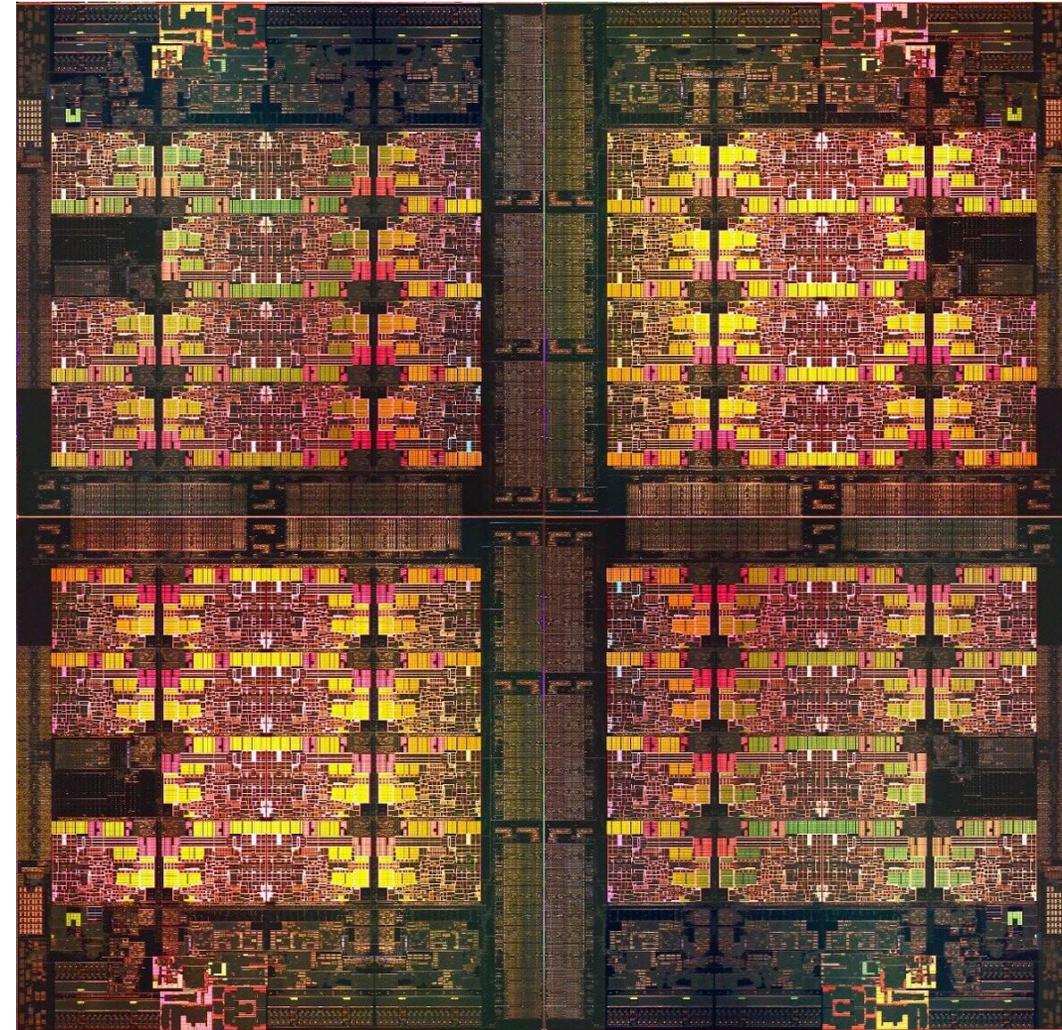
# HPC Node Architecture

CPU's

GPU's

# Multi-core today: Intel Xeon Sapphire Rapids (2023)

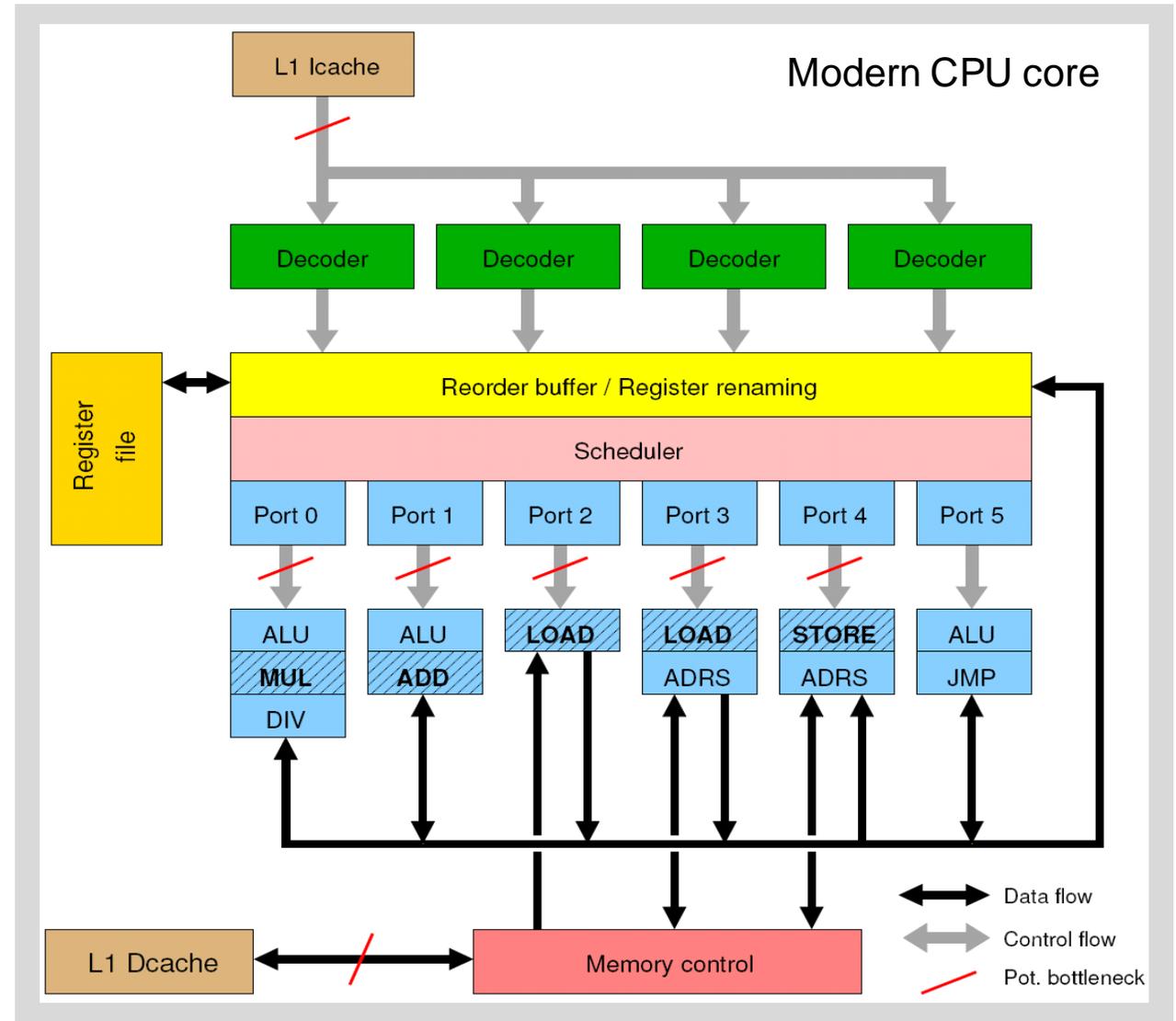
- Xeon “**Sapphire Rapids**” (Platinum/Gold/Silver/Bronze):  
Up to 60 cores running at 1.7+ GHz  
(+ “Turbo Mode” 4.8 GHz),
- “Intel 7” process / up to 350 W
- Multi-die package (4 chips)
- Clock frequency:  
**flexible** 😊



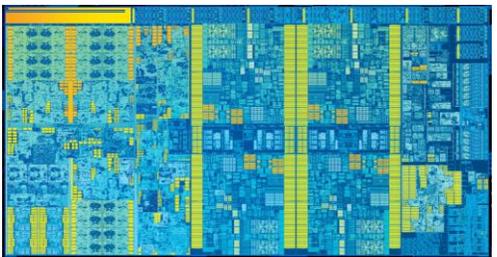
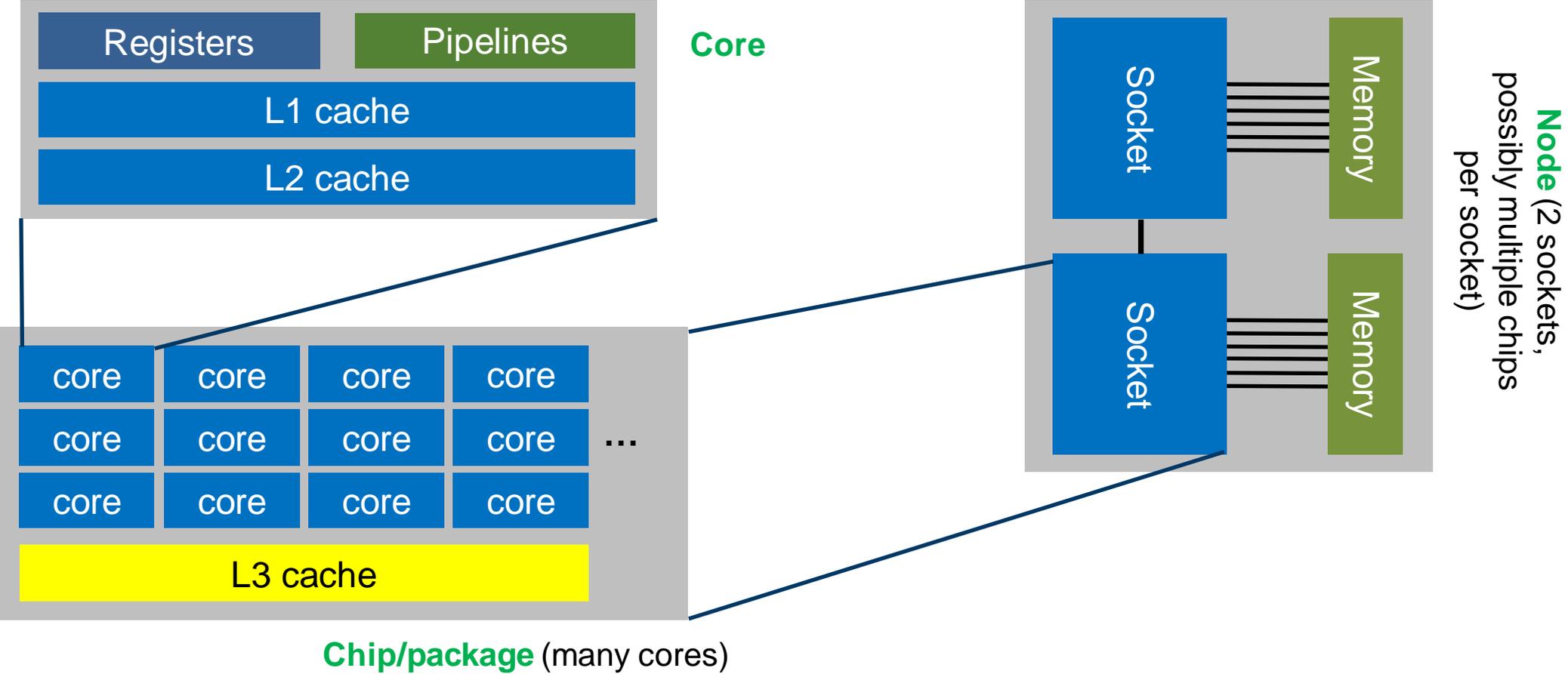
<https://www.techpowerup.com/292204/intel-sapphire-rapids-xe-on-4-tile-mcm-annotated>

# General-purpose cache-based microprocessor core

- Implements “Stored Program Computer” concept (Turing 1936)
- Similar designs on all modern systems
- (Still) multiple potential bottlenecks
- **Bottleneck mitigations**
  - Caches (instruction/data)
  - Instruction-level parallelism
  - Out-of-order and speculative execution

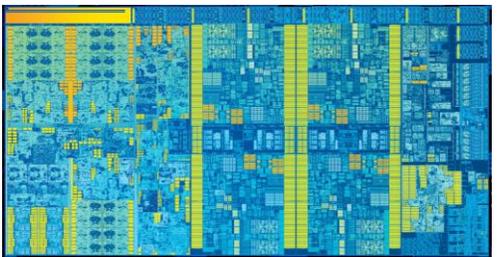
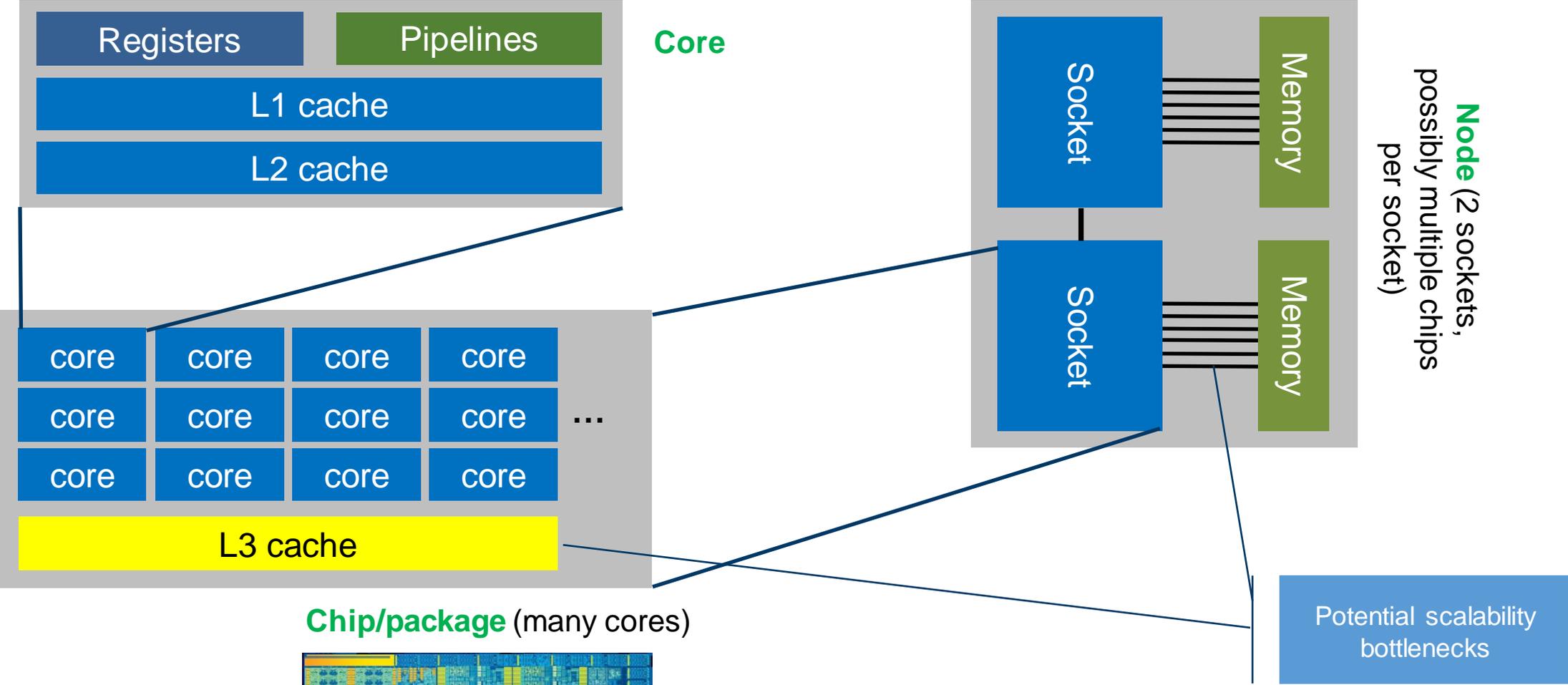


# Node topology of HPC systems



© Intel

# Node topology of HPC systems



© Intel

# Nvidia H100 “Hopper” SXM5 specs

## Architecture

- 80 B Transistors
- ~ 1.8 GHz clock speed
- ~ 144 “SM” units
  - 128 SP “cores” each (FMA)
  - 64 DP “cores” each (FMA)
  - 4 “Tensor Cores” each
  - 2:1 SP:DP performance
- ~ 34 TFlop/s DP peak (FP64)
- 50 MiB L2 Cache
- 80 GB HBM3
- MemBW ~ 3300 GB/s (theoretical)
- MemBW ~ 3000 GB/s (measured)

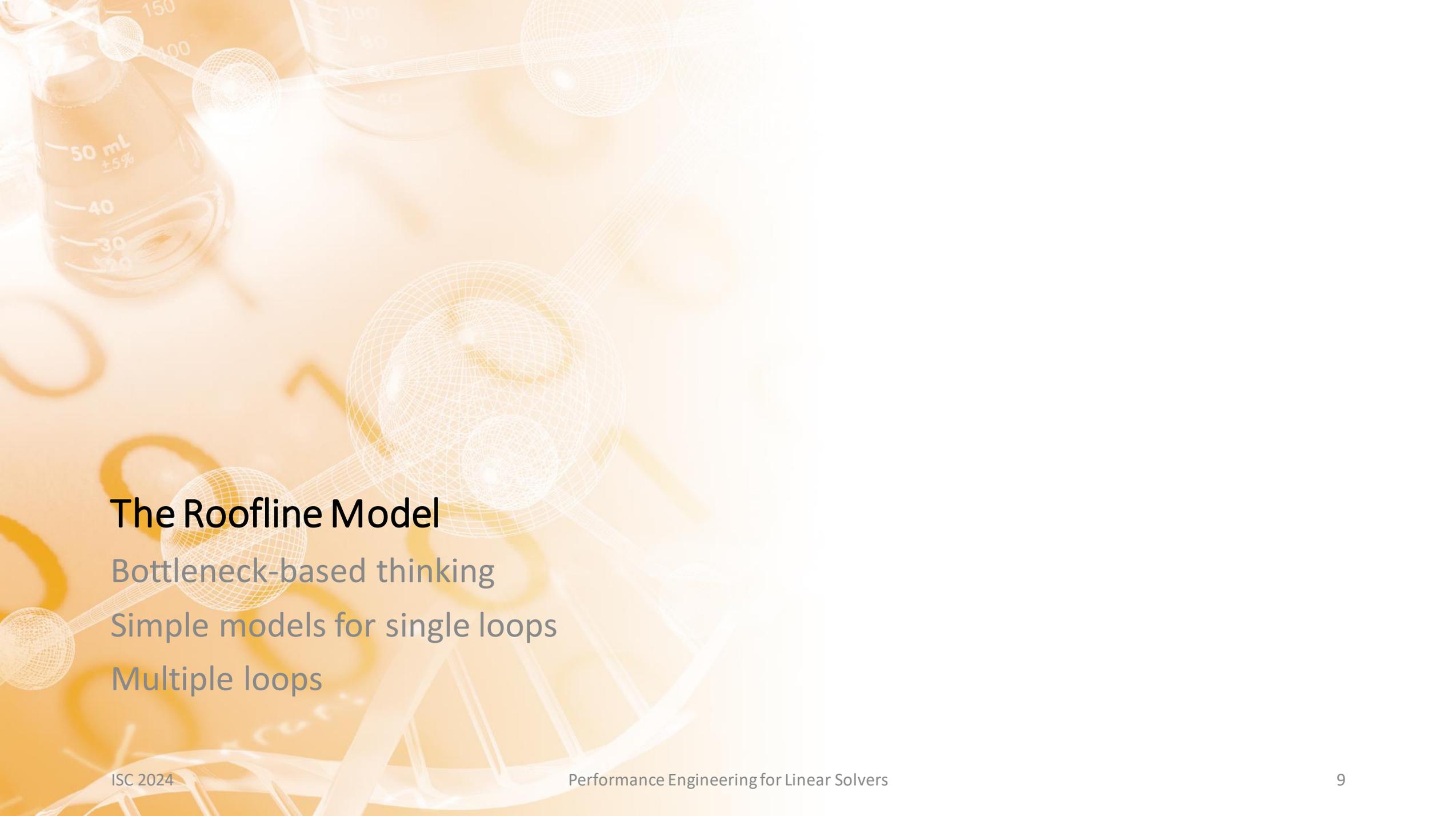


# Nvidia H100 “Hopper” SXM5 specs

## Architecture

- 80 B Transistors
- ~ 1.8 GHz clock speed
- ~ 144 “SM” units
  - 128 SP “cores” each (FMA)
  - 64 DP “cores” each (FMA)
  - 4 “Tensor Cores” each
  - 2:1 SP:DP performance
- ~ 34 TFlop/s DP peak (FP64)
- 50 MiB L2 Cache
- 80 GB HBM3
- MemBW ~ 3300 GB/s (theoretical)
- MemBW ~ 3000 GB/s (measured)





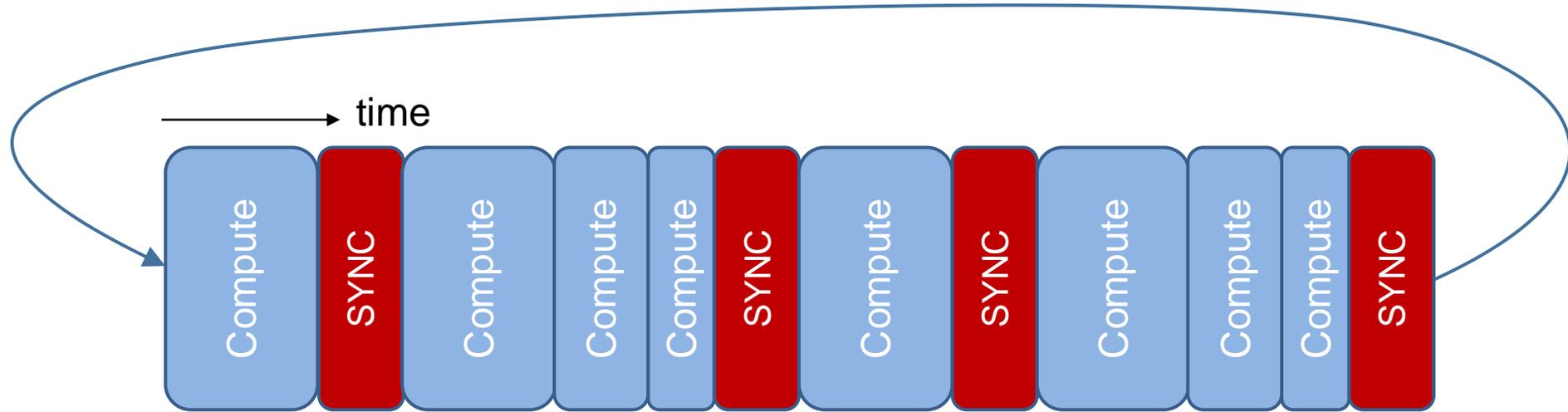
# The Roofline Model

Bottleneck-based thinking

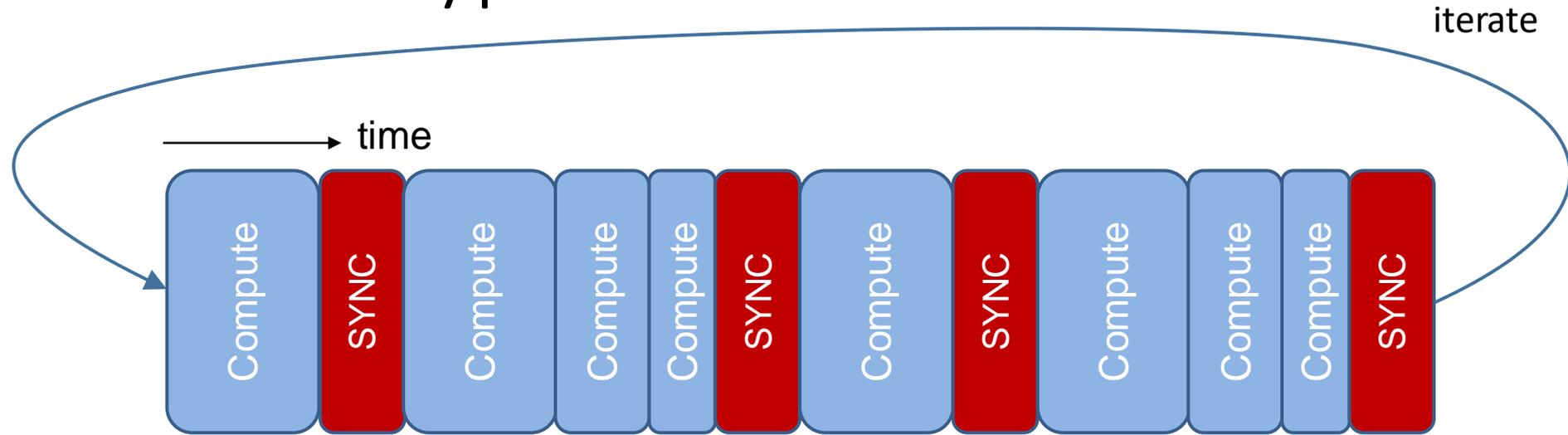
Simple models for single loops

Multiple loops

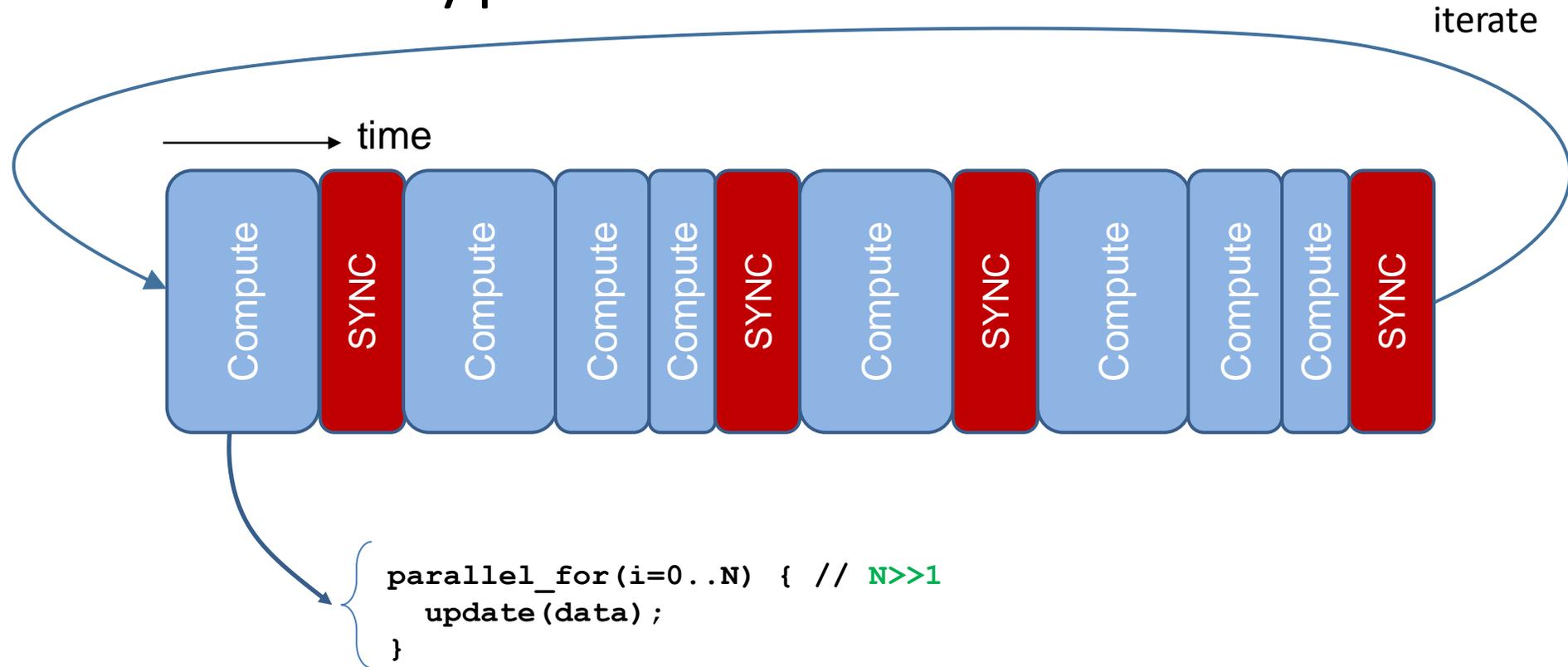
# Structure of typical solver code



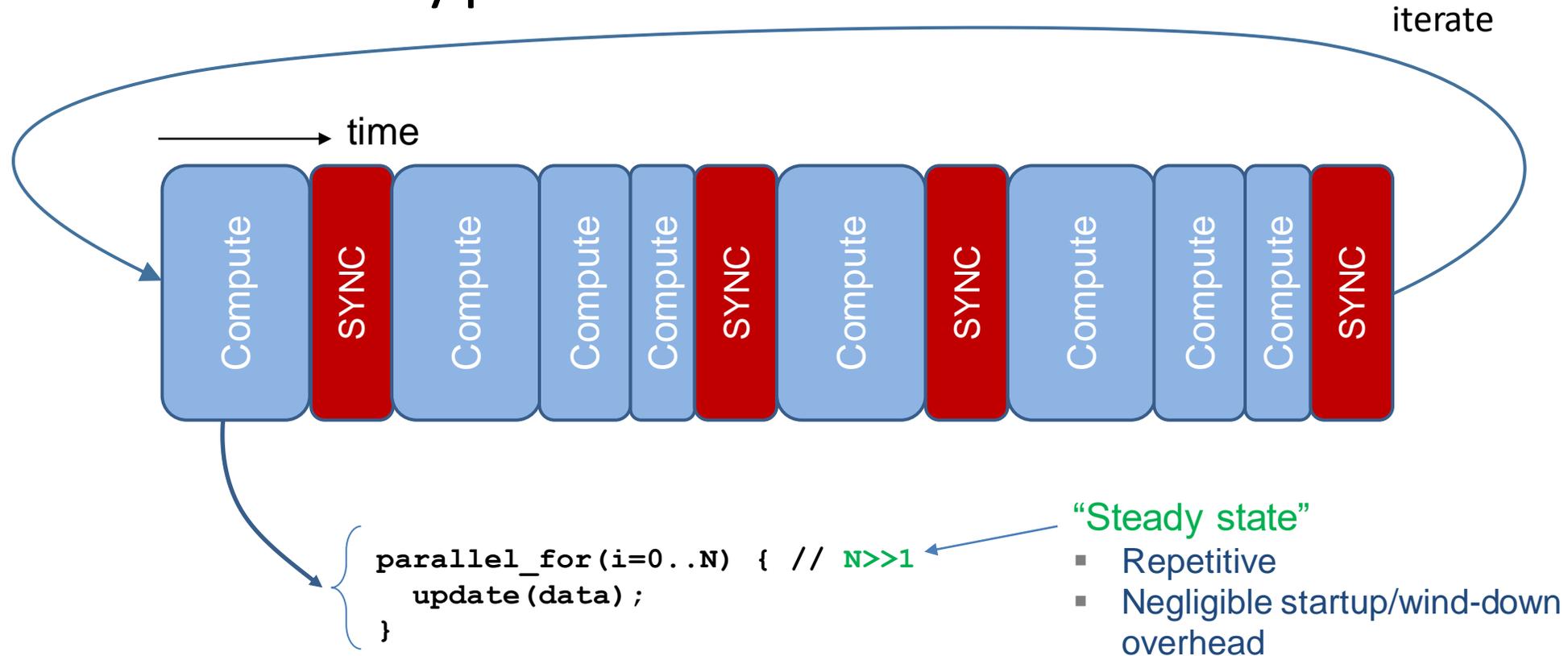
# Structure of typical solver code



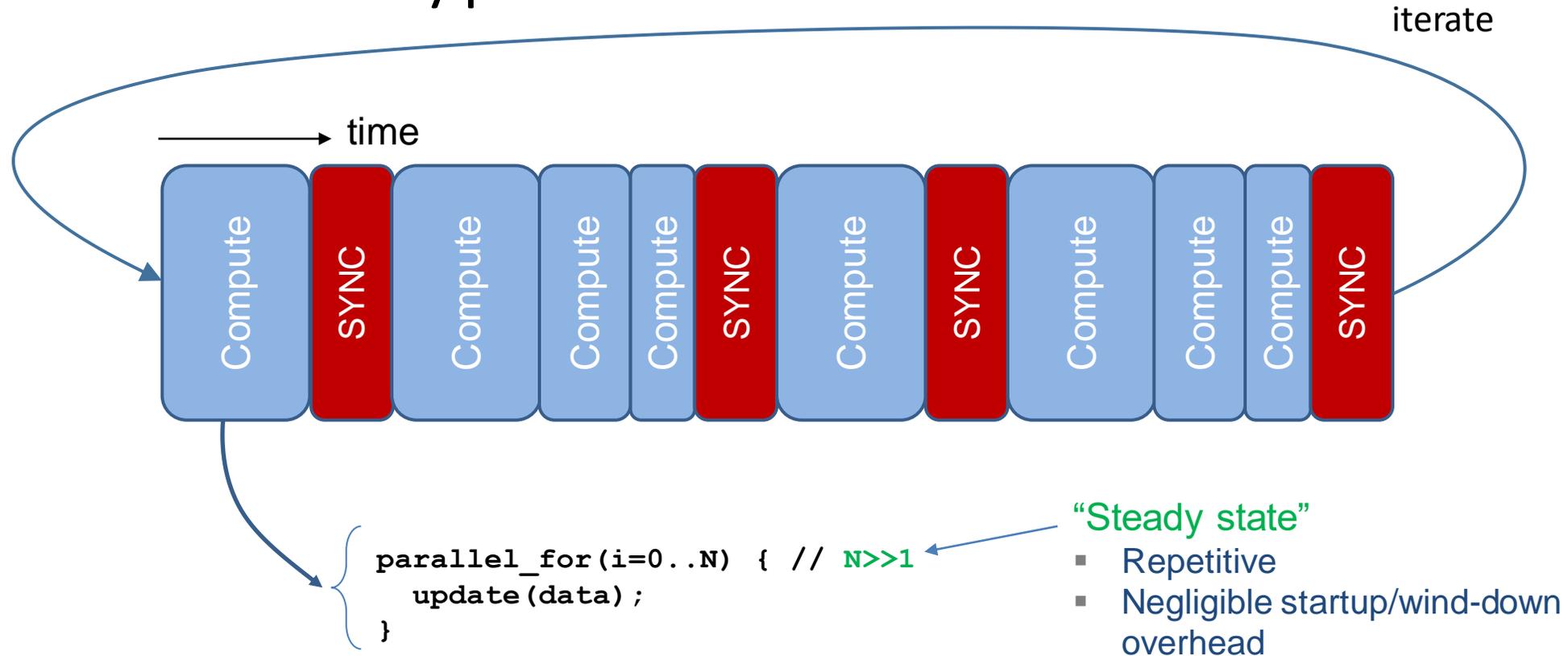
# Structure of typical solver code



# Structure of typical solver code



# Structure of typical solver code



Runtime model:  $T = f(\$STUFF, \$HARDWARE)$

# Resource bottlenecks

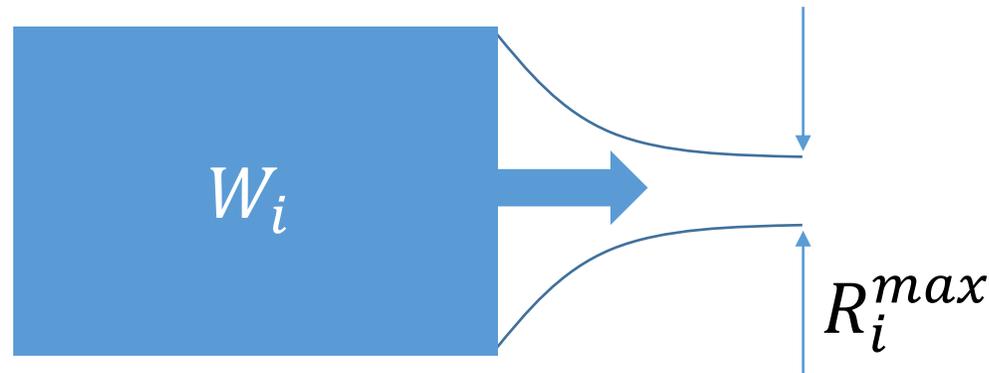
What is the **maximum performance** when limited by a **bottleneck**?

- Resource bottleneck  $i$  delivers resources at maximum rate  $R_i^{max}$
- $W_i$  = needed amount of resources

# Resource bottlenecks

What is the **maximum performance** when limited by a **bottleneck**?

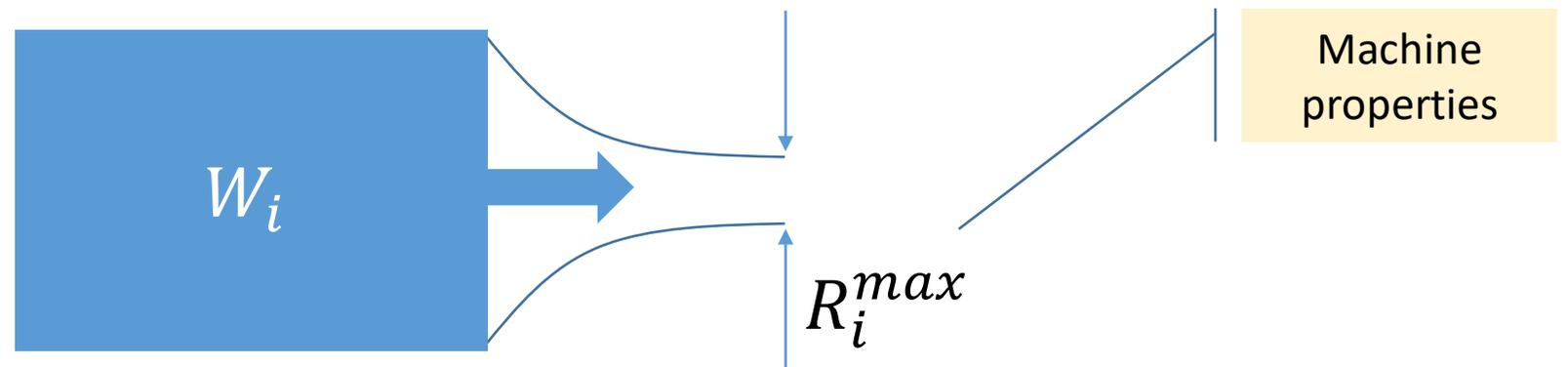
- Resource bottleneck  $i$  delivers resources at maximum rate  $R_i^{max}$
- $W_i$  = needed amount of resources



# Resource bottlenecks

What is the **maximum performance** when limited by a **bottleneck**?

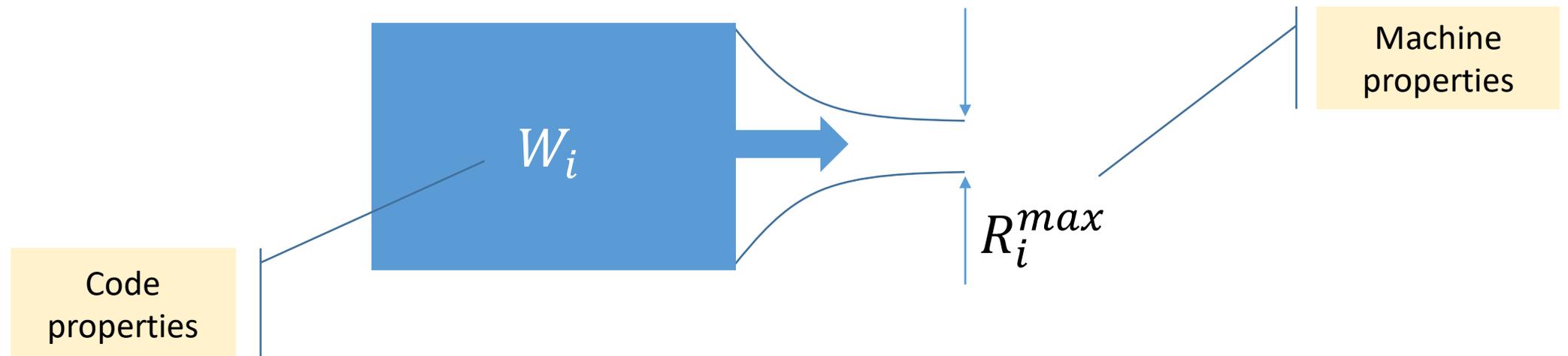
- Resource bottleneck  $i$  delivers resources at maximum rate  $R_i^{max}$
- $W_i$  = needed amount of resources



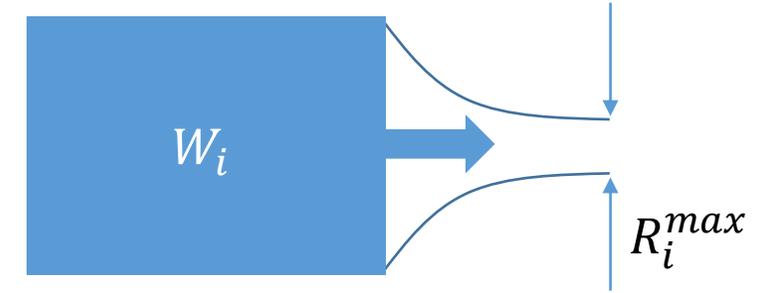
# Resource bottlenecks

What is the **maximum performance** when limited by a **bottleneck**?

- Resource bottleneck  $i$  delivers resources at maximum rate  $R_i^{max}$
- $W_i$  = needed amount of resources

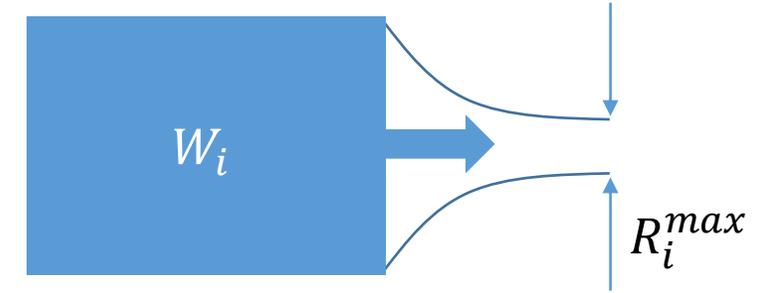


# Resource bottlenecks



Minimum runtime due to bottleneck  $i$ :  $T_i = \frac{W_i}{R_i^{max}} + \lambda_i$

# Resource bottlenecks

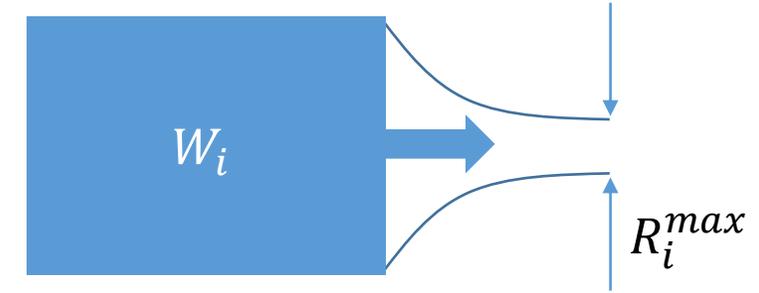


Minimum runtime due to bottleneck  $i$ :  $T_i = \frac{W_i}{R_i^{max}} + \lambda_i$

- Multiple bottlenecks?

→ multiple minimum runtimes:  $T_{\min} = f(T_1, \dots, T_n)$

# Resource bottlenecks



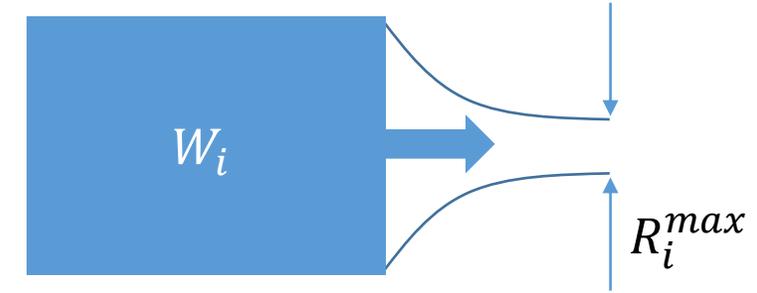
Minimum runtime due to bottleneck  $i$ :  $T_i = \frac{W_i}{R_i^{max}} + \lambda_i$

- Multiple bottlenecks?

→ multiple minimum runtimes:  $T_{\min} = f(T_1, \dots, T_n)$



# Resource bottlenecks



Minimum runtime due to bottleneck  $i$ :  $T_i = \frac{W_i}{R_i^{max}} + \lambda_i$

- Multiple bottlenecks?

→ multiple minimum runtimes:  $T_{\min} = f(T_1, \dots, T_n)$

- Overall performance:

$$P_{\max} = \frac{W}{T_{\min}}$$



# Simple bottleneck models for single loops

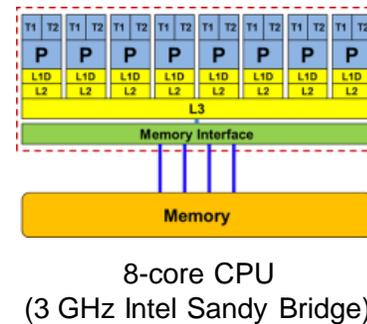
Two bottlenecks:

```
#pragma omp parallel for  
for(i=0; i<107; ++i)  
    a[i] = a[i] + s * c[i];
```

# Simple bottleneck models for single loops

Two bottlenecks:

```
#pragma omp parallel for
for(i=0; i<107; ++i)
    a[i] = a[i] + s * c[i];
```



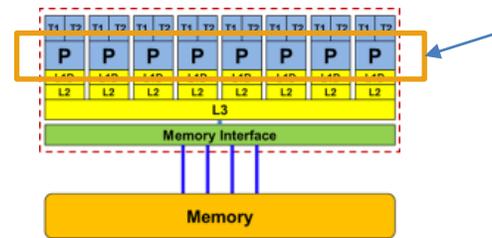
$$R_{flops}^{max} = 192 \frac{\text{Gflops}}{\text{s}}$$

$$R_{BW}^{max} = 40 \frac{\text{Gbyte}}{\text{s}}$$

# Simple bottleneck models for single loops

Two bottlenecks:

```
#pragma omp parallel for
for(i=0; i<107; ++i)
    a[i] = a[i] + s * c[i];
```



8-core CPU  
(3 GHz Intel Sandy Bridge)

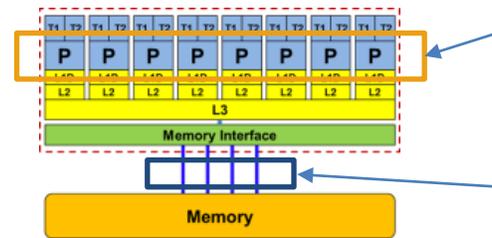
$$R_{flops}^{max} = 192 \frac{\text{Gflops}}{\text{s}}$$

$$R_{BW}^{max} = 40 \frac{\text{Gbyte}}{\text{s}}$$

# Simple bottleneck models for single loops

Two bottlenecks:

```
#pragma omp parallel for
for(i=0; i<107; ++i)
    a[i] = a[i] + s * c[i];
```



$$R_{flops}^{max} = 192 \frac{\text{Gflops}}{\text{s}}$$

$$R_{BW}^{max} = 40 \frac{\text{Gbyte}}{\text{s}}$$

8-core CPU  
(3 GHz Intel Sandy Bridge)

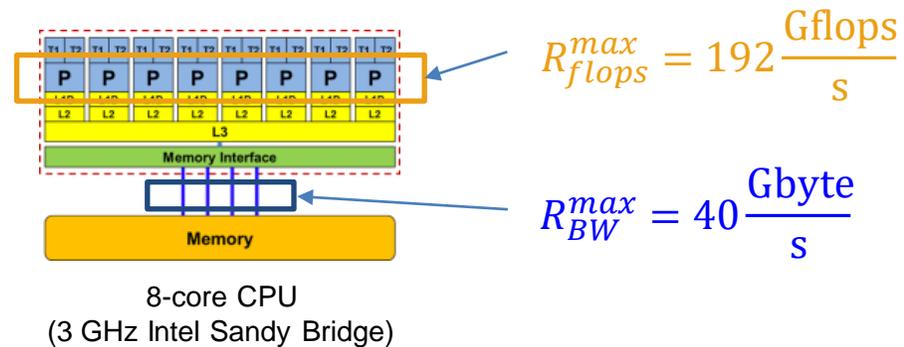
# Simple bottleneck models for single loops

Two bottlenecks:

```
#pragma omp parallel for  
for(i=0; i<107; ++i)  
    a[i] = a[i] + s * c[i];
```

$$W_{flops} = 2 \times 10^7 \text{ flops}$$

$$W_{BW} = 3 \times 8 \times 10^7 \text{ bytes}$$



# Simple bottleneck models for single loops

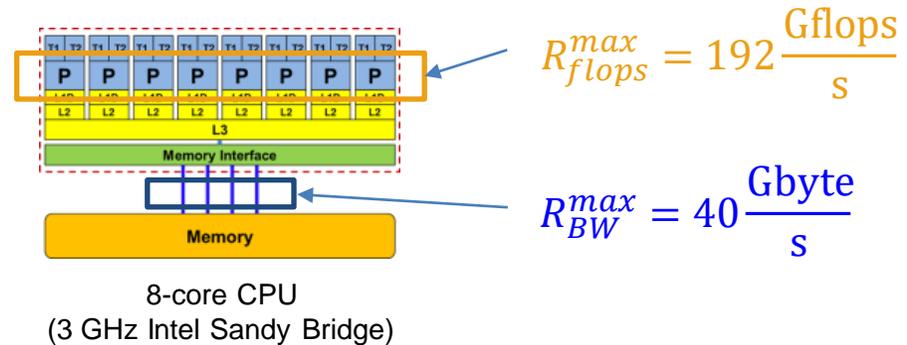
Two bottlenecks:

```
#pragma omp parallel for
for(i=0; i<107; ++i)
  a[i] = a[i] + s * c[i];
```

$$W_{flops} = 2 \times 10^7 \text{ flops}$$

$$W_{BW} = 3 \times 8 \times 10^7 \text{ bytes}$$

$$T_{flops} = \frac{2 \times 10^7 \text{ flops}}{192 \frac{\text{Gflops}}{\text{s}}} = 104 \mu\text{s}$$



$$T_{BW} = \frac{2.4 \times 10^8 \text{ bytes}}{40 \frac{\text{Gbyte}}{\text{s}}} = 6.0 \text{ ms}$$

# Bottleneck models for single loops

How do we reconcile the multiple bottlenecks?

I.e., what is the functional form of  $f(T_1, \dots, T_n)$ ?

→ **pessimistic** model (no overlap):  $f(T_1, \dots, T_n) = \sum_i T_i$

→ **optimistic** model (full overlap):  $f(T_1, \dots, T_n) = \max(T_1, \dots, T_n)$

Our example (two bottlenecks):  $T_{\min} = \max(T_{flops}, T_{BW}) = 6 \text{ ms}$

Maximum performance (“**light speed**”):  $P_{upper} = \frac{2 \times 10^7 \text{ flops}}{6.0 \times 10^{-3} \text{ s}} = 3.3 \text{ Gflop/s}$

# Bottleneck models for single loops

How do we reconcile the multiple bottlenecks?

I.e., what is the functional form of  $f(T_1, \dots, T_n)$ ?

→ **pessimistic** model (no overlap):  $f(T_1, \dots, T_n) = \sum_i T_i$

→ **optimistic** model (full overlap):  $f(T_1, \dots, T_n) = \max(T_1, \dots, T_n)$

Our example (two bottlenecks):  $T_{\min} = \max(T_{flops}, T_{BW}) = 6 \text{ ms}$

Maximum performance (“**light speed**”):  $P_{upper} = \frac{2 \times 10^7 \text{ flops}}{6.0 \times 10^{-3} \text{ s}} = 3.3 \text{ Gflop/s}$

Roofline model  
(Williams et al., 2008)

# From time to performance

$$P_{upper} = \frac{W_{flops}}{\max(T_{flops}, T_{BW})} = \frac{W_{flops}}{\max\left(\frac{W_{flops}}{R_{flops}}, \frac{W_{flops}}{R_{BW}}\right)} = \min\left(R_{flops}, R_{BW} \times \frac{W_{flops}}{W_{BW}}\right)$$

# From time to performance

$$P_{upper} = \frac{W_{flops}}{\max(T_{flops}, T_{BW})} = \frac{W_{flops}}{\max\left(\frac{W_{flops}}{R_{flops}}, \frac{W_{flops}}{R_{BW}}\right)} = \min\left(R_{flops}, R_{BW} \times \frac{W_{flops}}{W_{BW}}\right)$$

Machine model:  
Peak performance  
[flop/s]

# From time to performance

$$P_{upper} = \frac{W_{flops}}{\max(T_{flops}, T_{BW})} = \frac{W_{flops}}{\max\left(\frac{W_{flops}}{R_{flops}}, \frac{W_{flops}}{R_{BW}}\right)} = \min\left(R_{flops}, R_{BW} \times \frac{W_{flops}}{W_{BW}}\right)$$

Machine model:  
Peak performance  
[flop/s]

Machine model:  
Memory bandwidth  
[byte/s]

# From time to performance

$$P_{upper} = \frac{W_{flops}}{\max(T_{flops}, T_{BW})} = \frac{W_{flops}}{\max\left(\frac{W_{flops}}{R_{flops}}, \frac{W_{flops}}{R_{BW}}\right)} =$$

$$\min\left(R_{flops}, R_{BW} \times \frac{W_{flops}}{W_{BW}}\right)$$

Machine model:  
Peak performance  
[flop/s]

Application model:  
Computational  
intensity [flop/byte]

Machine model:  
Memory bandwidth  
[byte/s]

# “Roofline”!?

Common nomenclature:

$R_{flops}$  →  $P_{peak}$  peak performance

$R_{BW}$  →  $b_S$  memory bandwidth

$\frac{W_{flops}}{W_{BW}}$  →  $I$  computational intensity

# “Roofline”!?

Common nomenclature:

$R_{flops}$  →  $P_{peak}$  peak performance

$R_{BW}$  →  $b_S$  memory bandwidth

$\frac{W_{flops}}{W_{BW}}$  →  $I$  computational intensity

$$P_{upper} = \min(P_{peak}, I \times b_S)$$

# “Roofline”!?

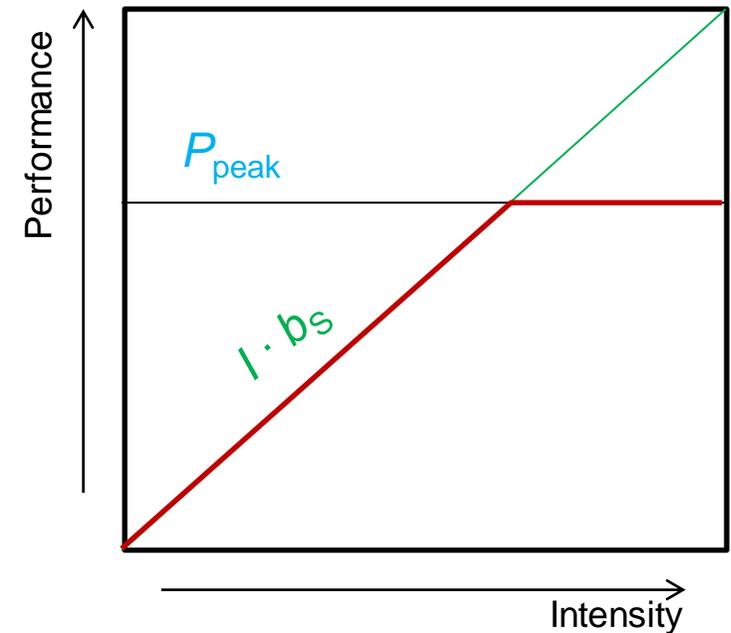
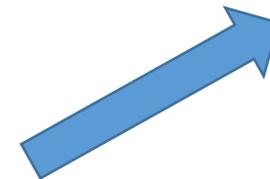
Common nomenclature:

$R_{flops}$  →  $P_{peak}$  peak performance

$R_{BW}$  →  $b_S$  memory bandwidth

$\frac{W_{flops}}{W_{BW}}$  →  $I$  computational intensity

$$P_{upper} = \min(P_{peak}, I \times b_S)$$



# “Roofline”!?

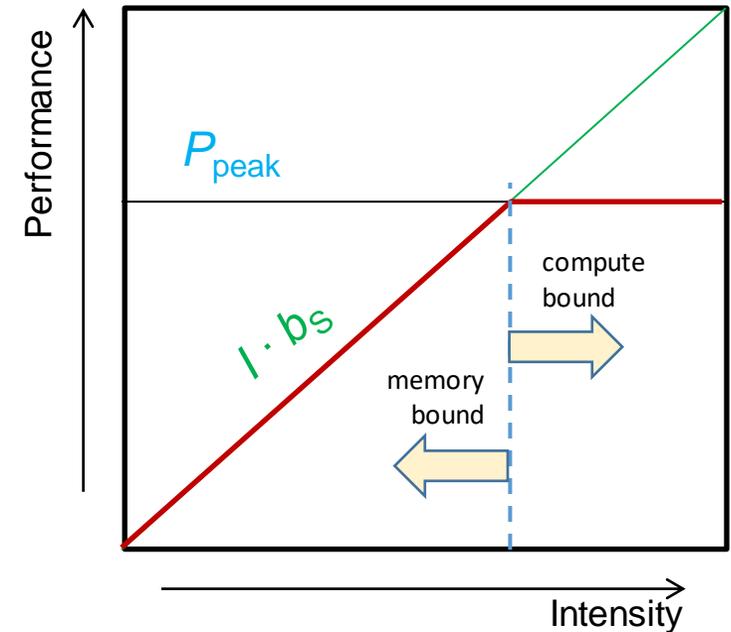
Common nomenclature:

$R_{flops}$  →  $P_{peak}$  peak performance

$R_{BW}$  →  $b_S$  memory bandwidth

$\frac{W_{flops}}{W_{BW}}$  →  $I$  computational intensity

$$P_{upper} = \min(P_{peak}, I \times b_S)$$



# “Roofline”!?

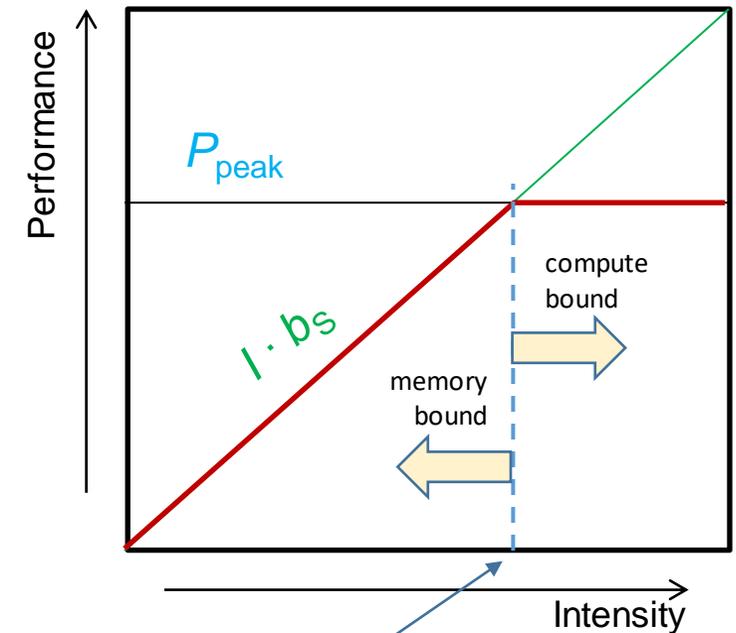
Common nomenclature:

$R_{flops}$  →  $P_{peak}$  peak performance

$R_{BW}$  →  $b_S$  memory bandwidth

$\frac{W_{flops}}{W_{BW}}$  →  $I$  computational intensity

$$P_{upper} = \min(P_{peak}, I \times b_S)$$



Threshold:  
 ≈ 10-15 F/B for current  
 Server CPUs/GPUs

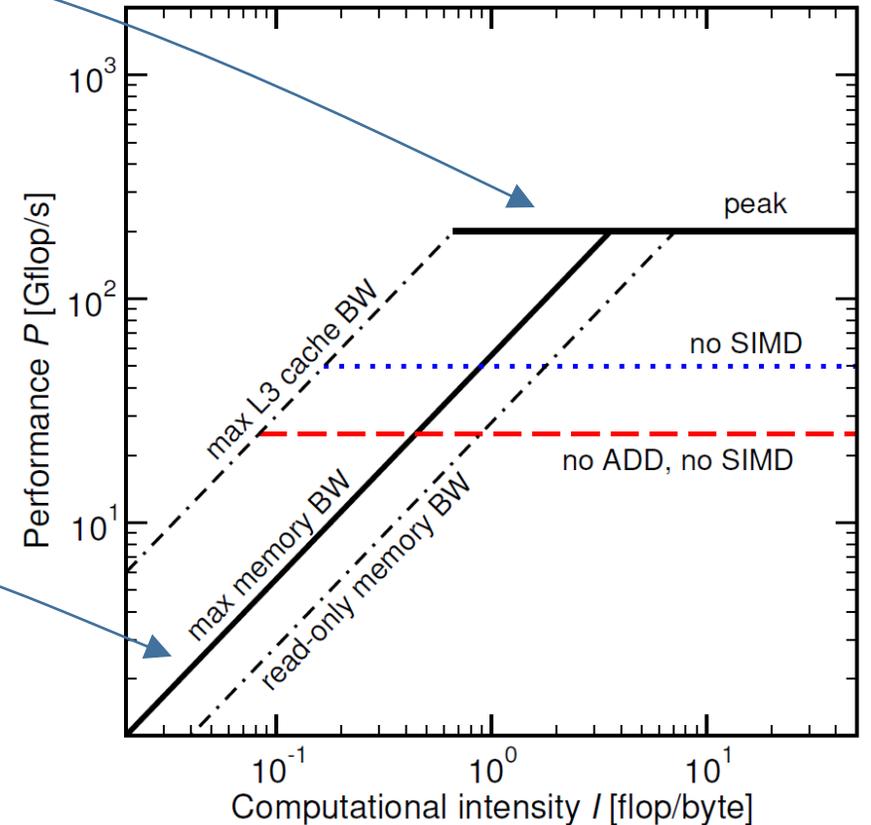
# Multiple bottlenecks?

Ceilings (flat)

- “Execution level” bottlenecks
- “Work” related
- Independent of intensity

Roofs (sloped)

- Data transfer bottlenecks
- “Traffic” related
- Linear in intensity



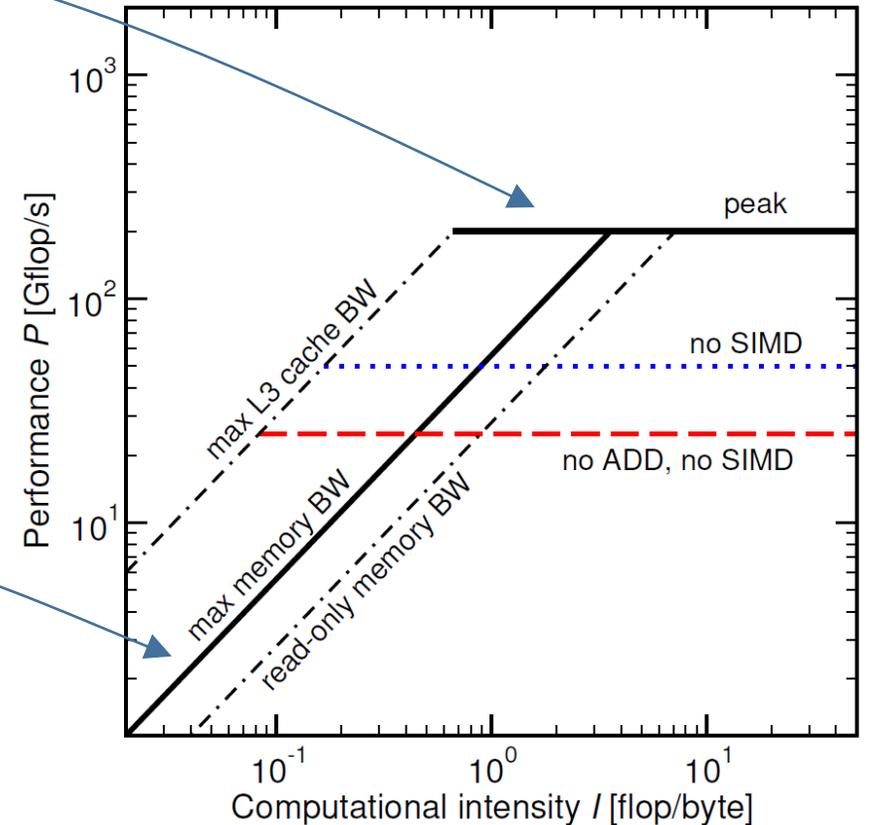
# Multiple bottlenecks?

Ceilings (flat)

- “Execution level” bottlenecks
- “Work” related
- Independent of intensity

Roofs (sloped)

- Data transfer bottlenecks
- “Traffic” related
- Linear in intensity



$$P_{upper} = \min_{i,j} (\{P_{max,i}\}, \{I_j \cdot b_j\})$$

# Two kinds of modeling

## Predictive

- Determine machine  $b_j$
- Calculate  $I_j, P_{\max,i}$
- Use  $P_{upper} = \min_{i,j}(\{P_{\max,i}\}, \{I_j \cdot b_j\})$
- Compare prediction(s) with measurement(s)
- Optimize, iterate

# Two kinds of modeling

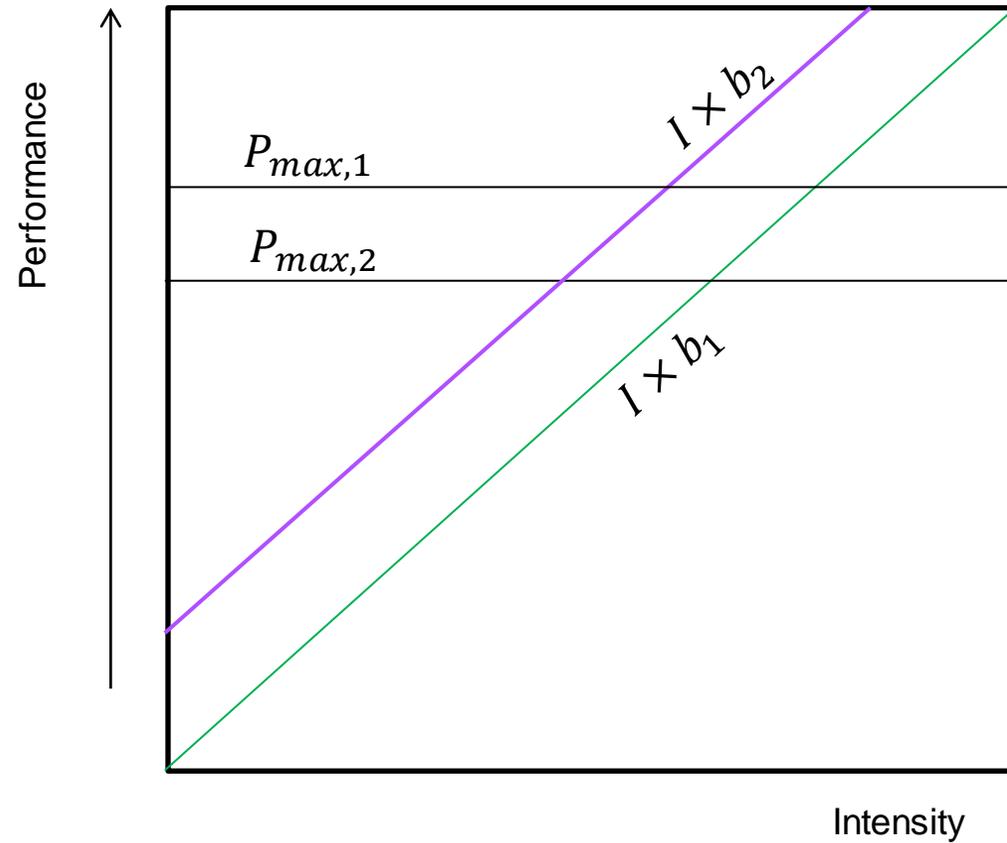
## Predictive

- Determine machine  $b_j$
- Calculate  $I_j, P_{\max,i}$
- Use  $P_{upper} = \min_{i,j}(\{P_{\max,i}\}, \{I_j \cdot b_j\})$
- Compare prediction(s) with measurement(s)
- Optimize, iterate

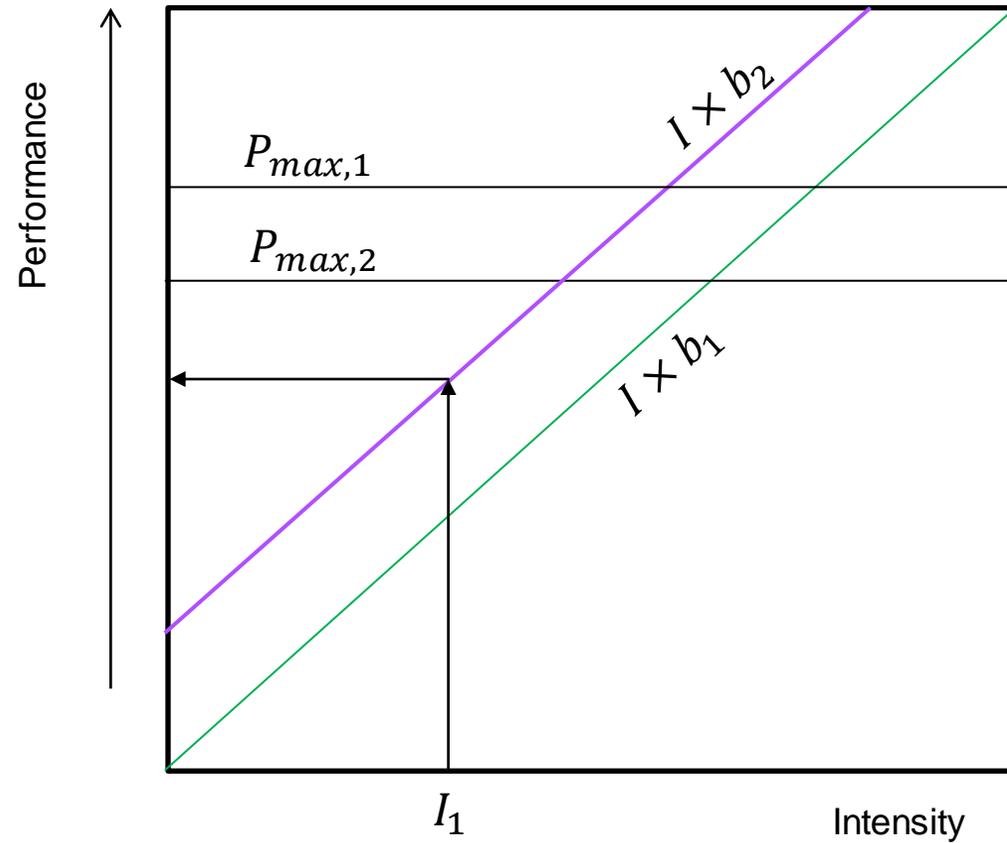
## Diagnostic/phenomenological

- Determine machine  $b_S, P_{\max,i}$
- Measure  $W_i$  (performance tools)
- Measure performance  $P$
- Compare with applicable roof/ceiling
- Optimize, iterate

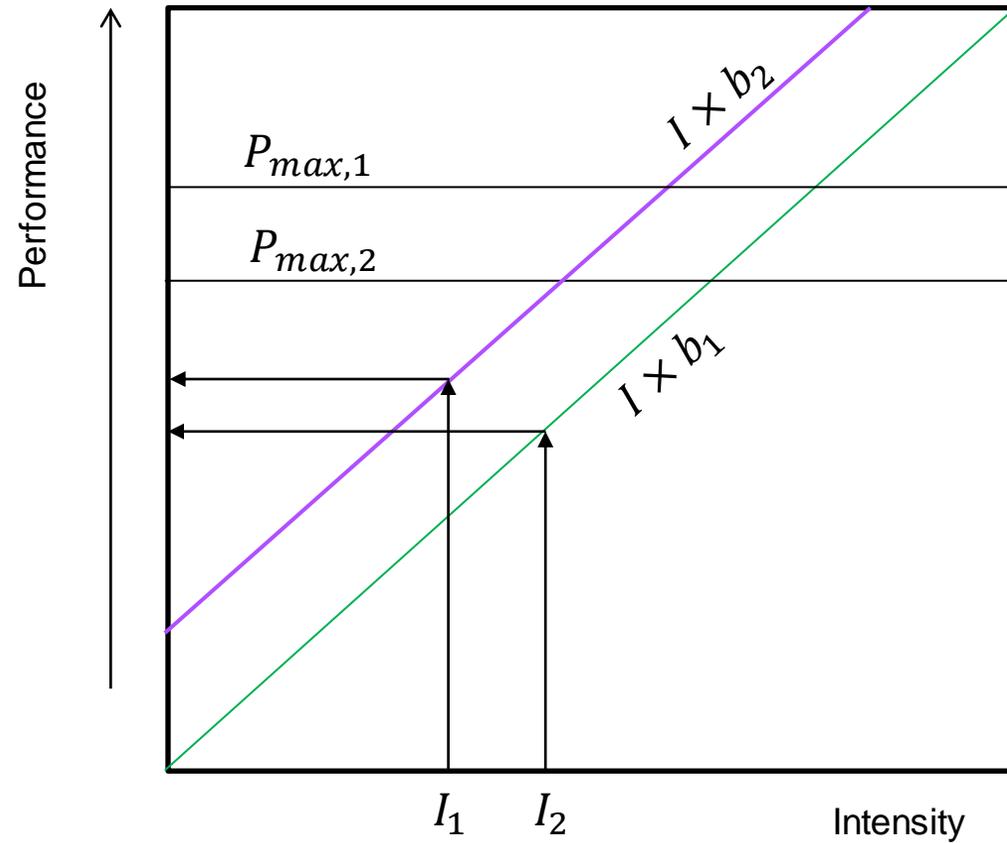
# Predictive modeling



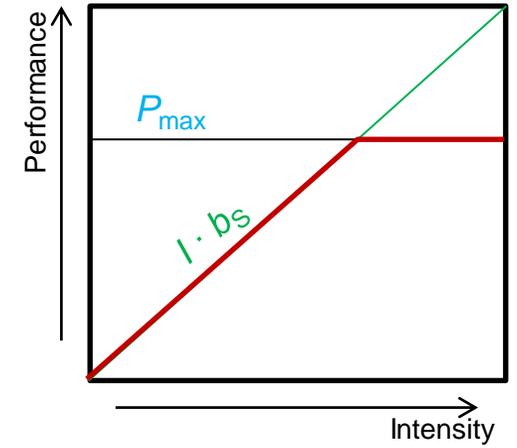
# Predictive modeling



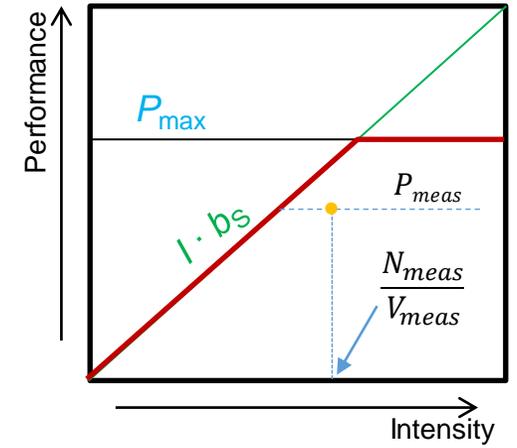
# Predictive modeling



# Diagnostic modeling

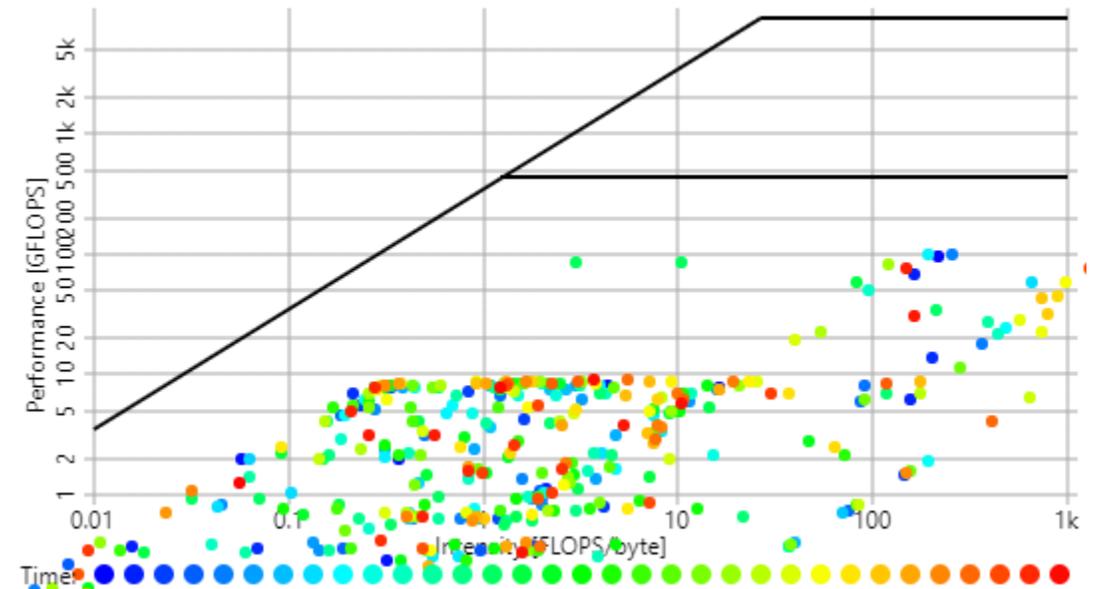
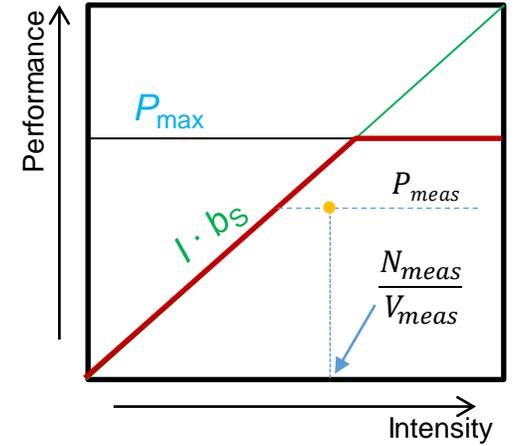
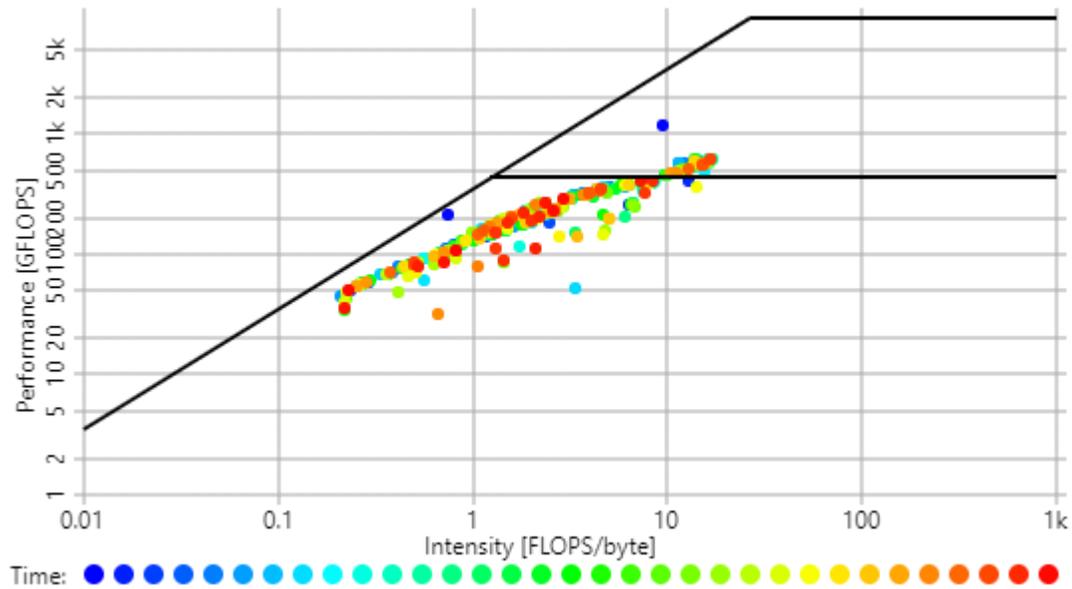


# Diagnostic modeling



# Diagnostic modeling

Two cluster jobs...



# What about multiple loops (i.e., solvers)?

Performance-based formulation is **inadequate** → go **back to time**

Solver:  $s$  components  $j = 1 \dots s$ ,  $t_j =$  model time for component  $j$

# What about multiple loops (i.e., solvers)?

Performance-based formulation is **inadequate** → go **back to time**

Solver:  $s$  components  $j = 1 \dots s$ ,  $t_j$  = model time for component  $j$

$$t_{\text{solver}} = \sum_{j=1}^s t_j = \sum_{j=1}^s f(T_{1,j}, \dots, T_{N,j})$$

# What about multiple loops (i.e., solvers)?

Performance-based formulation is **inadequate** → go **back to time**

Solver:  $s$  components  $j = 1 \dots s$ ,  $t_j$  = model time for component  $j$

$$t_{solver} = \sum_{j=1}^s t_j = \sum_{j=1}^s f(T_{1,j}, \dots, T_{N,j})$$

“Roofline”:

$$t_{solver} = \sum_{j=1}^s \max(T_{flops,j}, T_{BW,j})$$



## Roofline: Simple Examples

Dense linear algebra

Sparse linear algebra

Simple solvers: CG

# Dense linear algebra

```
for(i=0; i<N; ++i)
    a[i] = a[i]+s*x[i];
```

daxpy (BLAS-1)

```
for(i=0; i<N; ++i)
    s += a[i]*b[i];
```

dot product (BLAS-1)

## Roofline thinking:

## What is the computational intensity?

```
for(k=0; k<NK; ++k)
    for(l=0; l<NL; ++l)
        for(m=0; m<NM; ++m)
            y[k*NL+l] +=
                A[k*NM+m]*B[l*NM+m];
```

dense MMM (BLAS-3)

```
for(r=0; r<NR; ++r)
    for(c=0; c<NC; ++c)
        y[r] += A[r*NC+c]*x[c];
```

dense MVM (BLAS-2)

dot-product style

# Dot product

```
for(i=0; i<N; ++i)  
  s += a[i]*b[i];
```

- Two DP reads from memory ( $a[i]$ ,  $b[i]$ )  $\rightarrow$  16 byte/iteration
- 2 flops (\*,+) per iteration

Computational intensity  $I = \frac{2 \text{ flop}}{16 \text{ byte}} = 0.125 \frac{\text{flop}}{\text{byte}}$

# Daxpy

```
for(i=0; i<N; ++i)  
    a[i] = a[i]+s*x[i];
```

- Two DP reads, one DP write from/to memory → 24 byte/iteration
- 2 flops (+,\*) per iteration

Computational intensity  $I = \frac{2 \text{ flop}}{24 \text{ byte}} = 0.083 \frac{\text{flop}}{\text{byte}}$

# Dense MVM

```
for(r=0; r<NR; ++r)
  for(c=0; c<NC; ++c)
    y[r] += A[r*NC+c]*x[c];
```

- One DP read from memory for each matrix entry
- x[ ] and y[ ] are read and updated from cache after 1<sup>st</sup> read
- → 8 byte and 2 flops per iteration

Computational intensity  $I = \frac{2 \text{ flop}}{8 \text{ byte}} = 0.25 \frac{\text{flop}}{\text{byte}}$

# Dense MMM?

```
for(k=0; k<NK; ++k)
  for(l=0; l<NL; ++l)
    for(m=0; m<NM; ++m)
      y[k*NL+l] +=
        A[k*NM+m]*B[l*NM+m];
```

- Blocking/unrolling techniques can increase intensity beyond the Roofline knee

```
for(k=0; k<NK; k+=2)
  for(l=0; l<NL; l+=2)
    for(m=0; m<NM; ++m)
      y[k*NL+l]           += A[k*NM+m]*B[l*NM+m];
      y[(k+1)*NL+l]       += A[(k+1)*NM+m]*B[l*NM+m];
      y[k*NL+(l+1)]       += A[k*NM+m]*B[(l+1)*NM+m];
      y[(k+1)*NL+(l+1)]   += A[(k+1)*NM+m]*B[(l+1)*NM+m];
```

→ peak performance achievable



# Sparse Matrices and SpMV

Sparse Matrix Formats

Sparse Matrix Vector Product Parallelization

# Matrix Vector Multiplication

---

Input  $A, x, y$     Output  $y = A \cdot x$

- Central building block in many complex algorithms:
  - Orthogonalization, Krylov basis generation, preconditioner application, power iteration in Page Rank ...
- Before we turn to sparse matrices, we recall how we store & handle dense matrices on parallel processors (i.e. GPUs)

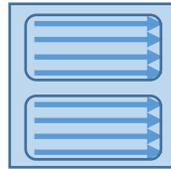
# Matrix Vector Multiplication

Input  $A, x, y$  Output  $y = A \cdot x$

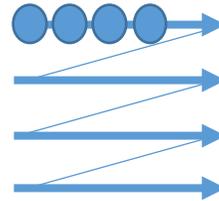
```
__global__ void sgemv_rowmajor( ...)\n{\n  int row = blockIdx.x*blockDim.x + threadIdx.x;\n  float sum = 0.0;\n  if (row < n){\n    for( int col=0; col<n; col++){\n      sum += m[ row*n + col ] * x[ col ];\n    }\n    y[ row ] = alpha * sum;\n  }\n}
```

```
__global__ void sgemv_colmajor( ...)\n{\n  int row = blockIdx.x*blockDim.x + threadIdx.x;\n  float sum = 0.0;\n  if (row < n){\n    for( int col=0; col<n; col++){\n      sum += m[ row + n*col ] * x[ col ];\n    }\n    y[ row ] = alpha * sum;\n  }\n}
```

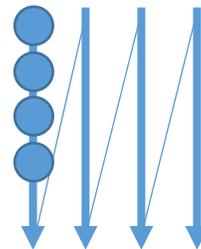
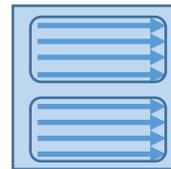
Parallel threads



Row-major



Col-major



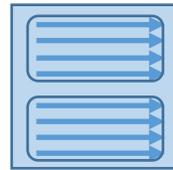
# Matrix Vector Multiplication

Input  $A, x, y$  Output  $y = A \cdot x$

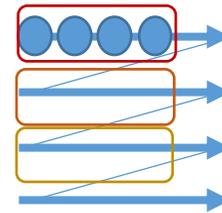
```
__global__ void sgemv_rowmajor( ...)\n{\n  int row = blockIdx.x*blockDim.x + threadIdx.x;\n  float sum = 0.0;\n  if (row < n){\n    for( int col=0; col<n; col++){\n      sum += m[ row*n + col ] * x[ col ];\n    }\n    y[ row ] = alpha * sum;\n  }\n}
```

```
__global__ void sgemv_colmajor( ...)\n{\n  int row = blockIdx.x*blockDim.x + threadIdx.x;\n  float sum = 0.0;\n  if (row < n){\n    for( int col=0; col<n; col++){\n      sum += m[ row + n*col ] * x[ col ];\n    }\n    y[ row ] = alpha * sum;\n  }\n}
```

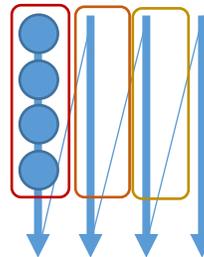
Parallel threads



Row-major



Col-major



First read

Second read

Third read

# Matrix Vector Multiplication

$$\text{Input } A, x, y \quad \text{Output } y = A \cdot x$$

```

__global__ void sgemv_rowmajor( ...)
{
  int row = blockIdx.x*blockDim.x + threadIdx.x;
  float sum = 0.0;
  if (row < n){
    for( int col=0; col<n; col++){
      sum += m[ row*n + col ] * x[ col ];
    }
    y[ row ] = alpha * sum;
  }
}

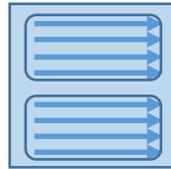
```

```

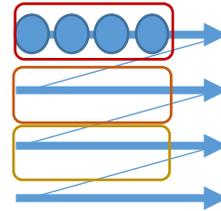
__global__ void sgemv_colmajor( ...)
{
  int row = blockIdx.x*blockDim.x + threadIdx.x;
  float sum = 0.0;
  if (row < n){
    for( int col=0; col<n; col++){
      sum += m[ row + n*col ] * x[ col ];
    }
    y[ row ] = alpha * sum;
  }
}

```

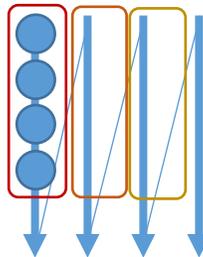
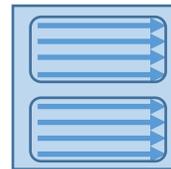
Parallel threads



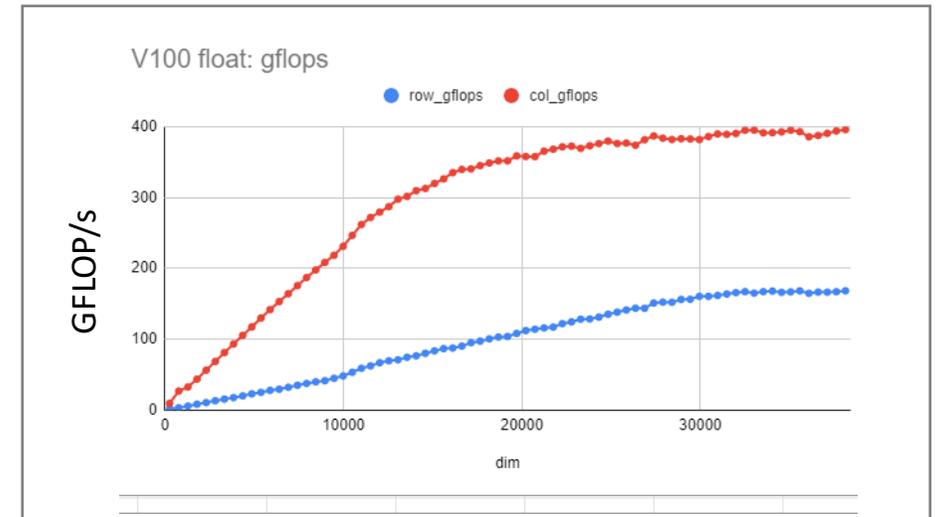
Row-major



Col-major



- First read
- Second read
- Third read



# Sparse Matrix Vector Multiplication

---

Input  $A, x, y$     Output  $y = A \cdot x$

- Matrix  $A$  contains only few nonzero elements.
- Storing all entries results in large overhead (memory & computation).

# Sparse Matrix Vector Multiplication

Input  $A, x, y$     Output  $y = A \cdot x$

- Matrix  $A$  contains only few nonzero elements.
- Storing all entries results in large overhead (memory & computation).
- **Idea:** Store only nonzero elements [nz] explicitly.

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & 0 & 0 & 9.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8.1 \end{pmatrix}$$

value = [ 5.4 1.1 2.2 8.3 3.7 1.3 3.8 4.2 5.4 9.2 1.1 8.1 ]    Value

# COO format

Input  $A, x, y$     Output  $y = A \cdot x$

- Matrix  $A$  contains only few nonzero elements.
- Storing all entries results in large overhead (memory & computation).
- **Idea:** Store only nonzero elements [nz] explicitly.

**Need to also store location of nonzero elements!**

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & 0 & 0 & 9.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8.1 \end{pmatrix}$$

Memory footprint of COO format:  
 $\text{nz}(\text{val}) + 2 * \text{nz}(\text{int})$

value = [ 5.4 1.1 2.2 8.3 3.7 1.3 3.8 4.2 5.4 9.2 1.1 8.1 ]

Value

colidx = [ 0 1 0 1 3 4 5 2 0 3 4 5 ]

Column-index

rowidx = [ 0 0 1 1 1 1 1 2 3 3 4 5 ]

Row-index

# COO format

Input  $A, x, y$  Output  $y = A \cdot x$

- Matrix  $A$  contains only few nonzero elements
  - Storing all entries is inefficient
  - **Idea:** Store only nonzero elements
- Need to**

Hands-on Exercise: Convert this matrix into COO format:

$$\begin{pmatrix} 0 & 0 & 0 & 4 & 2 & 0 \\ 0 & 2 & 3 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 2 & 3 & 4 & 3 & 1 \\ 0 & 2 & 0 & 0 & 0 & 1 \\ 1 & 2 & 3 & 0 & 0 & 1 \end{pmatrix}$$

format:

value = [ 5

colidx = [ 0

rowidx = [ 0 0 1 1 1 1 1 2 3 3 4 5 ] **Row-index**

Compute the memory requirement (# vals + # int)

# COO format

Input  $A, x, y$  Output  $y = A \cdot x$

- Matrix  $A$  contains only few nonzero elements
  - Storing all entries is inefficient
  - **Idea:** Store only nonzero elements
- Need to**

Hands-on Exercise: Convert this matrix into COO format:

$$\begin{pmatrix} 0 & 0 & 0 & 4 & 2 & 0 \\ 0 & 2 & 3 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 2 & 3 & 4 & 3 & 1 \\ 0 & 2 & 0 & 0 & 0 & 1 \\ 1 & 2 & 3 & 0 & 0 & 1 \end{pmatrix}$$

format:

value = [ 5

colidx = [ 0

rowidx = [ 0

Compute the memory requirement (# vals + # int)

17 vals + 34 int

Row-index

# COO SpMV

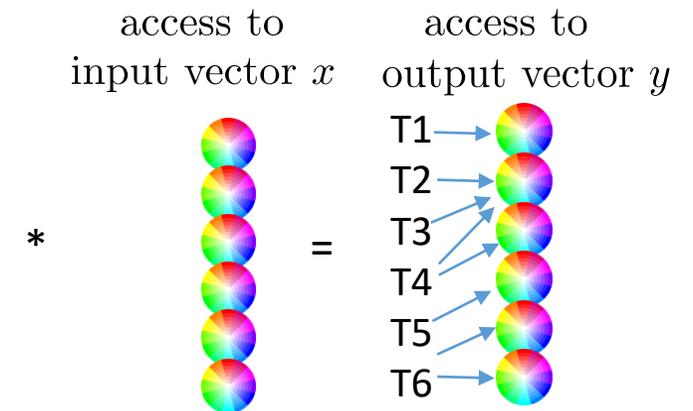
Input  $A, x, y$     Output  $y = A \cdot x$

Split nonzero elements into chunks and parallelize across chunks.

- Partial sums need synchronization / atomics to avoid write conflicts.
- Non-coalesced memory access (because row-major).

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & 0 & 0 & 9.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8.1 \end{pmatrix}$$

value = [	5.4	1.1	2.2	8.3	3.7	1.3	3.8	4.2	5.4	9.2	1.1	8.1	]
colidx = [	0	1	0	1	3	4	5	2	0	3	4	5	]
rowidx = [	0	0	1	1	1	1	1	2	3	3	4	5	]



Value  
Column-index  
Row-index

# CSR (==CRS) format

Input  $A, x, y$     Output  $y = A \cdot x$

- Matrix  $A$  contains only few nonzero elements.
- Storing all entries results in large overhead (memory & computation).
- **Idea:** Store only nonzero elements [nz] explicitly.

**Need to also store location of nonzero elements!**

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & 0 & 0 & 9.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8.1 \end{pmatrix}$$

Memory footprint of COO format:  
 $\text{nz}(\text{val}) + 2 * \text{nz}(\text{int})$

value = [ 5.4 1.1 2.2 8.3 3.7 1.3 3.8 4.2 5.4 9.2 1.1 8.1 ]

Value

colidx = [ 0 1 0 1 3 4 5 2 0 3 4 5 ]

Column-index

# CSR format

Input  $A, x, y$  Output  $y = A \cdot x$

- Matrix  $A$  contains only few nonzero elements.
- Storing all entries results in large overhead (memory & computation).
- **Idea:** Store only nonzero elements [nz] explicitly.

**Need to also store location of nonzero elements!**

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & 0 & 0 & 9.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8.1 \end{pmatrix}$$

Memory footprint of COO format:  
 $\text{nz}(\text{val}) + 2 * \text{nz}(\text{int})$

Memory footprint of CSR format:  
 $\text{nz}(\text{val}) + \text{nz}(\text{int}) + (n+1) (\text{int})$

value = [ 5.4 1.1 2.2 8.3 3.7 1.3 3.8 4.2 5.4 9.2 1.1 8.1 ]

Value

colidx = [ 0 1 0 1 3 4 5 2 0 3 4 5 ]

Column-index

rowptr = [ 0 2 7 8 10 11 12 ]

Points to the first element in each row

Number of nonzero elements

# CSR format

Input  $A, x, y$  Output  $y = A \cdot x$

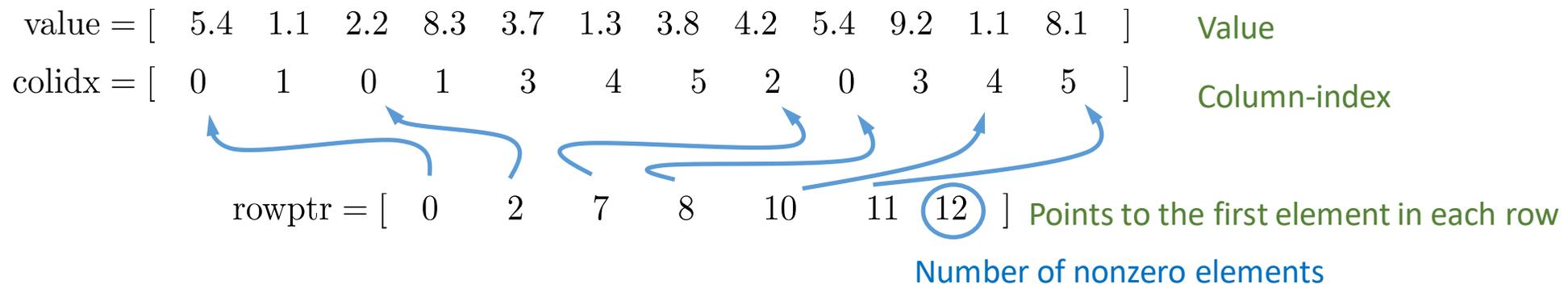
- Matrix  $A$  contains only few nonzero elements.
- Storing all entries results in large overhead (memory & computation).
- **Idea:** Store only nonzero elements [nz] explicitly.

**Need to also store location of nonzero elements!**

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & 0 & 0 & 9.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8.1 \end{pmatrix}$$

Memory footprint of COO format:  
 $\text{nz}(\text{val}) + 2 * \text{nz}(\text{int})$

Memory footprint of CSR format:  
 $\text{nz}(\text{val}) + \text{nz}(\text{int}) + (n+1) (\text{int})$



# CSR format

Input  $A, x, y$  Output  $y = A \cdot x$

- Matrix  $A$  contains only few nonzero elements
  - Storing all entries is inefficient
  - Idea: Store only nonzero elements
- Need to store:

Hands-on Exercise: Convert this matrix into CSR format:

$$\begin{pmatrix} 0 & 0 & 0 & 4 & 2 & 0 \\ 0 & 2 & 3 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 2 & 3 & 4 & 3 & 1 \\ 0 & 2 & 0 & 0 & 0 & 1 \\ 1 & 2 & 3 & 0 & 0 & 1 \end{pmatrix}$$

value = [ 5

colidx = [ 0

Compute the memory requirement (# vals + # int)

rowptr = [ 0 2 7 8 10 11 12 ] Points to the first element in each row

Number of nonzero elements

# CSR format

Input  $A, x, y$  Output  $y = A \cdot x$

- Matrix  $A$  contains only few nonzero elements
  - Storing all entries is inefficient
  - **Idea:** Store only nonzero elements
- Need to**

Hands-on Exercise: Convert this matrix into CSR format:

$$\begin{pmatrix} 0 & 0 & 0 & 4 & 2 & 0 \\ 0 & 2 & 3 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 2 & 3 & 4 & 3 & 1 \\ 0 & 2 & 0 & 0 & 0 & 1 \\ 1 & 2 & 3 & 0 & 0 & 1 \end{pmatrix}$$

format:

format:

t)

value = [ 5

colidx = [ 0

Compute the memory requirement (# vals + # int)

17 vals + 24 int

rowptr = [ 0 2 7 8 10 11 12 ] Points to the first element in each row

Number of nonzero elements

# CSR SpMV

Input  $A, x, y$  Output  $y = A \cdot x$

How to parallelize this?

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & 0 & 0 & 9.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8.1 \end{pmatrix}$$

value = [ 5.4 1.1 2.2 8.3 3.7 1.3 3.8 4.2 5.4 9.2 1.1 8.1 ]

Value

colidx = [ 0 1 0 1 3 4 5 2 0 3 4 5 ]

Column-index

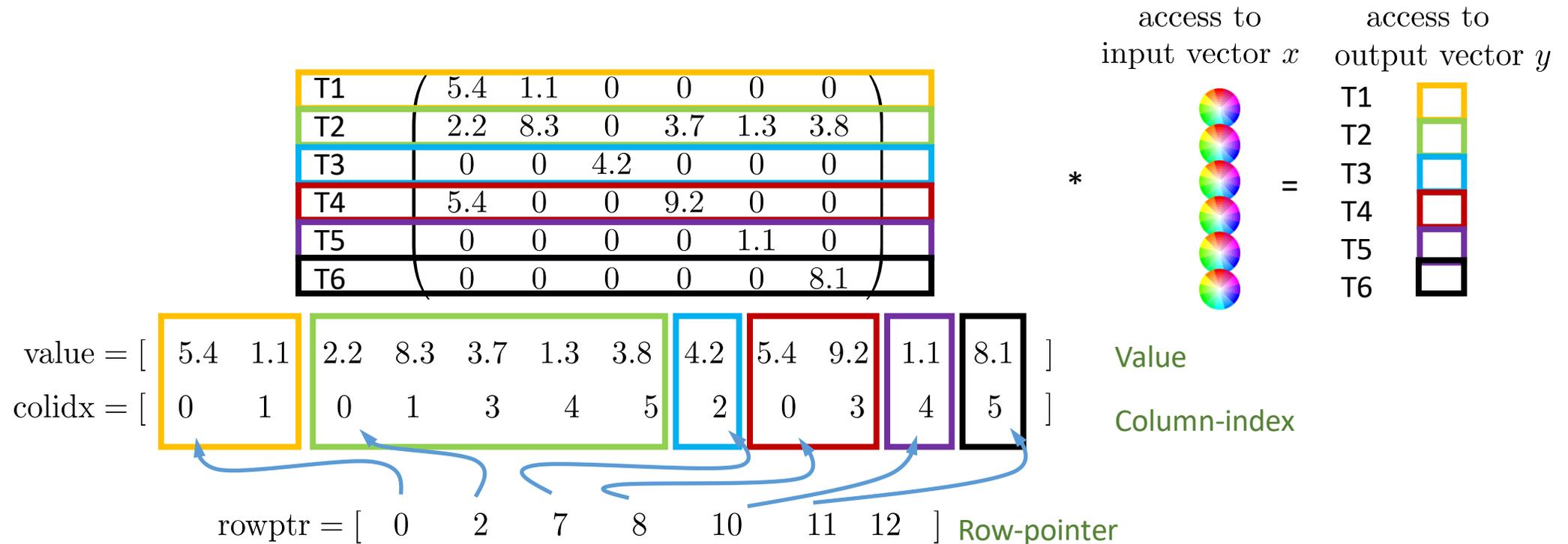
rowptr = [ 0 2 7 8 10 11 12 ]

Row-pointer

# CSR SpMV

Input  $A, x, y$     Output  $y = A \cdot x$

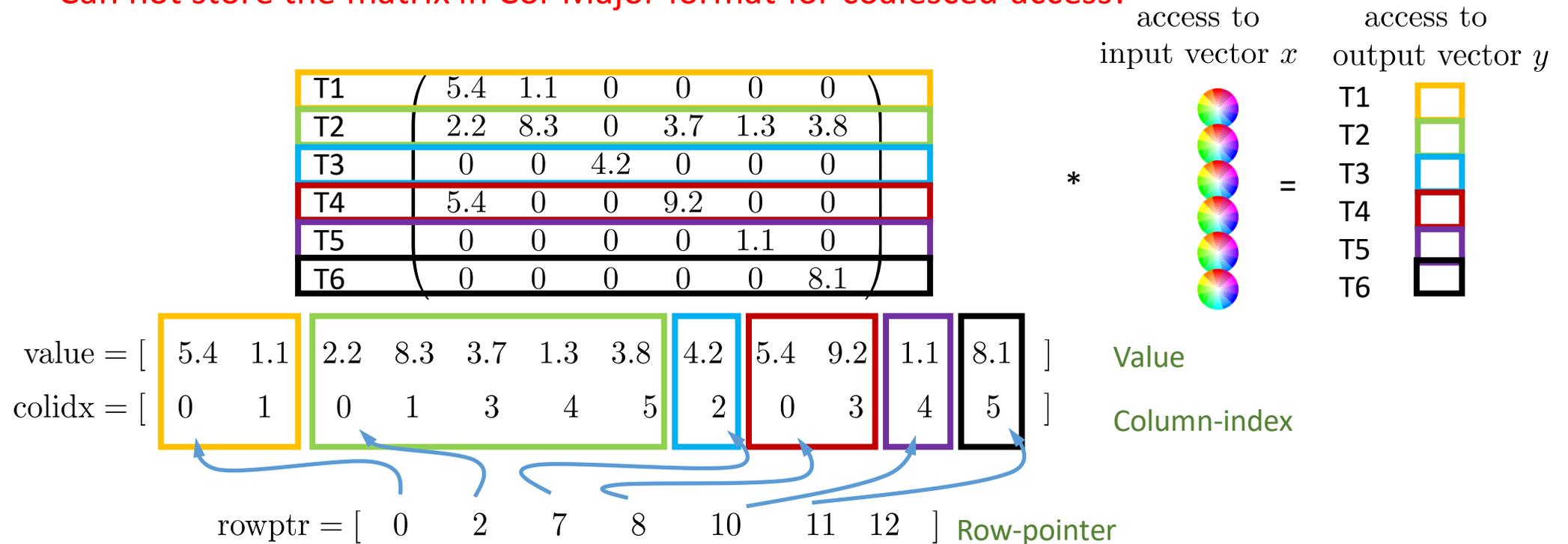
- Parallelize by rows:
  - Every “thread” handles the computation of one sum in local memory.



# CSR SpMV

Input  $A, x, y$     Output  $y = A \cdot x$

- Parallelize by rows:
  - Every “thread” handles the computation of one sum in local memory.
  - Significant workload imbalance!**
  - Can not store the matrix in Col-Major format for coalesced access!**





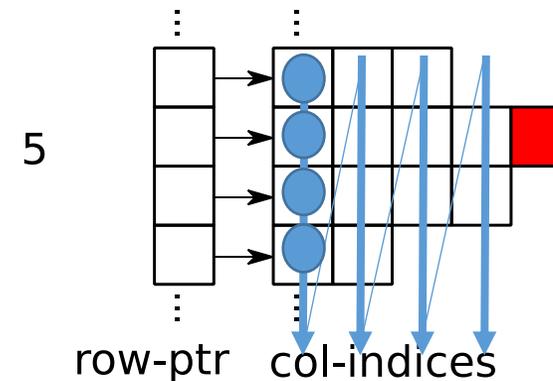
# CSR SpMV

Input  $A, x, y$  Output  $y = A \cdot x$

```
for( row=0; row<n; row++ )
{
    sum = 0.0;
    for( j=rowptr[row]; j<rowptr[row+1]; j++)
        sum += values[ j ] * x[ colind[j] ];
    y[ row ] = alpha * sum;
}
```

Storing values and columns in row-major.

-> On GPUs: non-coalesced memory access



Can we use column-major?

-> Only if all rows contain the same number of nonzero elements

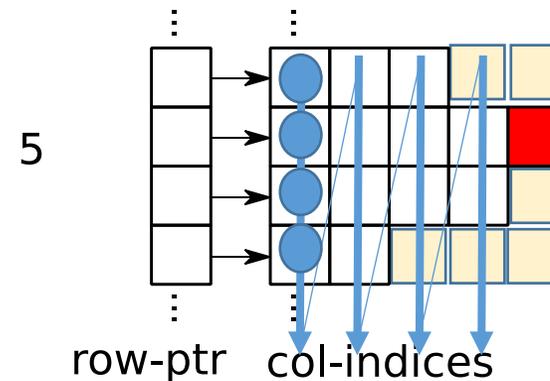
# CSR SpMV

Input  $A, x, y$  Output  $y = A \cdot x$

```
for( row=0; row<n; row++ )
{
    sum = 0.0;
    for( j=rowptr[row]; j<rowptr[row+1]; j++)
        sum += values[ j ] * x[ colind[j] ];
    y[ row ] = alpha * sum;
}
```

Storing values and columns in row-major.

-> On GPUs: non-coalesced memory access



Can we use column-major?

-> Only if all rows contain the same number of nonzero elements

# ELL Format

---

Input  $A, x, y$     Output  $y = A \cdot x$

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & 0 & 0 & 9.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8.1 \end{pmatrix}$$

# ELL Format

Input  $A, x, y$     Output  $y = A \cdot x$

*'Left-align nonzero elements'*

$$A = \begin{pmatrix} 5.4 & \leftarrow 1.1 & \leftarrow 0 & \leftarrow 0 & \leftarrow 0 & 0 \\ 2.2 & \leftarrow 8.3 & \leftarrow 0 & \leftarrow 3.7 & \leftarrow 1.3 & 3.8 \\ 0 & \leftarrow 0 & \leftarrow 4.2 & \leftarrow 0 & \leftarrow 0 & 0 \\ 5.4 & \leftarrow 0 & \leftarrow 0 & \leftarrow 9.2 & \leftarrow 0 & 0 \\ 0 & \leftarrow 0 & \leftarrow 0 & \leftarrow 0 & \leftarrow 1.1 & 0 \\ 0 & \leftarrow 0 & \leftarrow 0 & \leftarrow 0 & \leftarrow 0 & 8.1 \end{pmatrix}$$

$$\begin{bmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 3.7 & 1.3 & 3.8 & 0 \\ 4.2 & 0 & 0 & 0 & 0 & 0 \\ 5.4 & 9.2 & 0 & 0 & 0 & 0 \\ 1.1 & 0 & 0 & 0 & 0 & 0 \\ 8.1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 1 & - & - & - & - \\ 0 & 1 & 3 & 4 & 5 & - \\ 2 & - & - & - & - & - \\ 0 & 3 & - & - & - & - \\ 4 & - & - & - & - & - \\ 5 & - & - & - & - & - \end{bmatrix}$$

# ELL Format

Input  $A, x, y$  Output  $y = A \cdot x$

*'Left-align nonzero elements'*

$$A = \begin{pmatrix} 5.4 & \leftarrow 1.1 & \leftarrow 0 & \leftarrow 0 & \leftarrow 0 & 0 \\ 2.2 & \leftarrow 8.3 & \leftarrow 0 & \leftarrow 3.7 & \leftarrow 1.3 & 3.8 \\ 0 & \leftarrow 0 & \leftarrow 4.2 & \leftarrow 0 & \leftarrow 0 & 0 \\ 5.4 & \leftarrow 0 & \leftarrow 0 & \leftarrow 9.2 & \leftarrow 0 & 0 \\ 0 & \leftarrow 0 & \leftarrow 0 & \leftarrow 0 & \leftarrow 1.1 & 0 \\ 0 & \leftarrow 0 & \leftarrow 0 & \leftarrow 0 & \leftarrow 0 & 8.1 \end{pmatrix}$$

$$\begin{bmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 3.7 & 1.3 & 3.8 & 0 \\ 4.2 & 0 & 0 & 0 & 0 & 0 \\ 5.4 & 9.2 & 0 & 0 & 0 & 0 \\ 1.1 & 0 & 0 & 0 & 0 & 0 \\ 8.1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 1 & - & - & - & - \\ 0 & 1 & 3 & 4 & 5 & - \\ 2 & - & - & - & - & - \\ 0 & 3 & - & - & - & - \\ 4 & - & - & - & - & - \\ 5 & - & - & - & - & - \end{bmatrix}$$

*Pad rows to uniform length*

*Memory volume:*

*values:  $\max\_nnz\_row * \text{num\_rows}$*

*col-index:  $\max\_nnz\_row * \text{num\_rows}$*

*no row pointer*

# ELL Format

Input  $A, x, y$     Output  $y = A \cdot x$

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & 0 & 0 & 9.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8.1 \end{pmatrix}$$

5.4	1.1	0	0	0	0
2.2	8.3	3.7	1.3	3.8	0
4.2	0	0	0	0	0
5.4	9.2	0	0	0	0
1.1	0	0	0	0	0
8.1	0	0	0	0	0

0	1	—	—	—	—
0	1	3	4	5	—
2	—	—	—	—	—
0	3	—	—	—	—
4	—	—	—	—	—
5	—	—	—	—	—

*Pad rows to uniform length*

*Memory volume:*

*values:  $\max\_nnz\_row * \text{num\_rows}$*

*col-index:  $\max\_nnz\_row * \text{num\_rows}$*

*no row pointer*

# ELL SpMV

Input  $A, x, y$  Output  $y = A \cdot x$

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & 0 & 0 & 9.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8.1 \end{pmatrix}$$

T1	5.4	1.1	0	0	0	0		0	1	—	—	—	—
T2	2.2	8.3	3.7	1.3	3.8	0		0	1	3	4	5	—
T3	4.2	0	0	0	0	0		2	—	—	—	—	—
T4	5.4	9.2	0	0	0	0		0	3	—	—	—	—
T5	1.1	0	0	0	0	0		4	—	—	—	—	—
T6	8.1	0	0	0	0	0		5	—	—	—	—	—

*Pad rows to uniform length*

*Memory volume:*

*values:  $\max\_nnz\_row * \text{num\_rows}$*

*col-index:  $\max\_nnz\_row * \text{num\_rows}$*

*no row pointer*

# ELL SpMV

Input  $A, x, y$  Output  $y = A \cdot x$

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & 0 & 0 & 9.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8.1 \end{pmatrix}$$

T1	5.4	1.1	0	0	0	0		0	1	-	-	-	-	
T2	2.2	8.3	3.7	1.3	3.8	0		0	1	3	4	5	-	
T3	4.2	0	0	0	0	0		2	-	-	-	-	-	
T4	5.4	9.2	0	0	0	0		0	3	-	-	-	-	
T5	1.1	0	0	0	0	0		4	-	-	-	-	-	
T6	8.1	0	0	0	0	0		5	-	-	-	-	-	

Pad rows to uniform length

Memory volume:

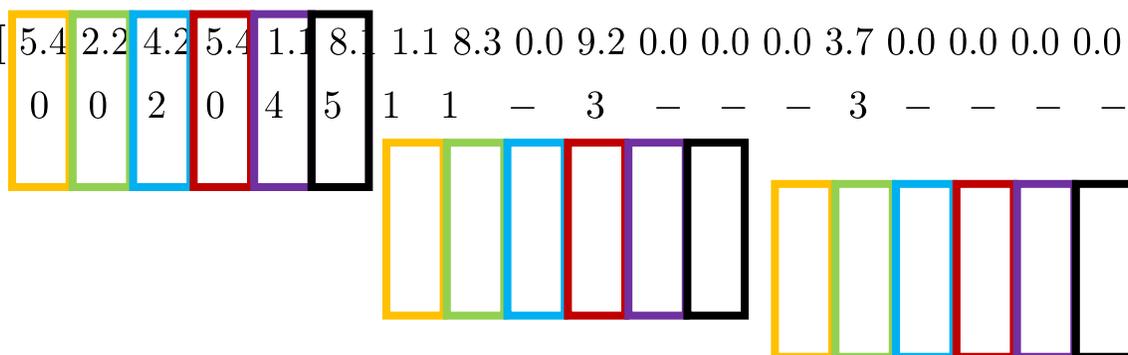
values:  $\max\_nnz\_row * \text{num\_rows}$

col-index:  $\max\_nnz\_row * \text{num\_rows}$

no row pointer

value = [ 5.4 2.2 4.2 5.4 1.1 8.1 1.1 8.3 0.0 9.2 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.3 0.0 0.0 0.0 0.0 0.0 3.8 0.0 0.0 0.0 0.0 ]

colidx = [ 0 0 2 0 4 5 1 1 - 3 - - - 3 - - - - - 4 - - - - - 5 - - - - ]

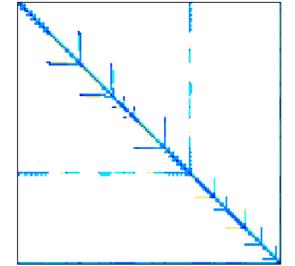


Coalesced access

# ELL SpMV

Input  $A, x, y$  Output  $y = A \cdot x$

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & 0 & 0 & 9.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8.1 \end{pmatrix}$$



T1	5.4	1.1	0	0	0	0		0	1	-	-	-	-	
T2	2.2	8.3	3.7	1.3	3.8	0		0	1	3	4	5	-	
T3	4.2	0	0	0	0	0		2	-	-	-	-	-	
T4	5.4	9.2	0	0	0	0		0	3	-	-	-	-	
T5	1.1	0	0	0	0	0		4	-	-	-	-	-	
T6	8.1	0	0	0	0	0		5	-	-	-	-	-	

Pad rows to uniform length

Memory volume:

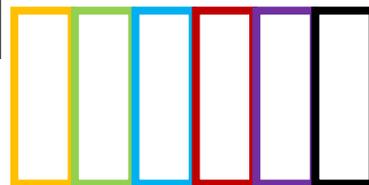
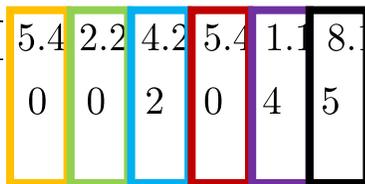
values:  $\max\_nnz\_row * \text{num\_rows}$

col-index:  $\max\_nnz\_row * \text{num\_rows}$

no row pointer

value = [ 5.4 2.2 4.2 5.4 1.1 8.1 1.1 8.3 0.0 9.2 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.3 0.0 0.0 0.0 0.0 0.0 3.8 0.0 0.0 0.0 0.0 ]

colidx = [ 0 0 2 0 4 5 1 1 - 3 - - - 3 - - - - - 4 - - - - - 5 - - - - ]



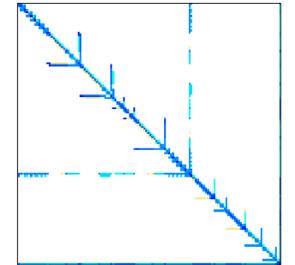
Coalesced access

Can be wasteful (overhead)

# ELL SpMV

Input  $A, x, y$  Output  $y = A \cdot x$

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & & & & & \\ 0 & & & & & \\ 0 & & & & & \end{pmatrix}$$



Hands-on Exercise: Convert this matrix into ELL format:

T1	5.4
T2	2.2
T3	4.2
T4	5.4
T5	1.1
T6	8.1

$$\begin{pmatrix} 0 & 0 & 0 & 4 & 2 & 0 \\ 0 & 2 & 3 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 2 & 3 & 4 & 3 & 1 \\ 0 & 2 & 0 & 0 & 0 & 1 \\ 1 & 2 & 3 & 0 & 0 & 1 \end{pmatrix}$$

value = [ 5.4 2.2 ]  
colidx = [ 0 0 ]

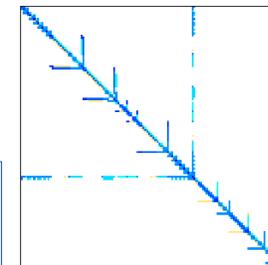
Compute the memory requirement (# vals + # int)

length  
row \* num\_rows  
\_row \* num\_rows  
0.0 0.0 ]  
- - ]  
erhead)

# ELL SpMV

Input  $A, x, y$  Output  $y = A \cdot x$

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & & & & & \\ 0 & & & & & \\ 0 & & & & & \end{pmatrix}$$



Hands-on Exercise: Convert this matrix into ELL format:

T1	5.4
T2	2.2
T3	4.2
T4	5.4
T5	1.1
T6	8.1

$$\begin{pmatrix} 0 & 0 & 0 & 4 & 2 & 0 \\ 0 & 2 & 3 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 2 & 3 & 4 & 3 & 1 \\ 0 & 2 & 0 & 0 & 0 & 1 \\ 1 & 2 & 3 & 0 & 0 & 1 \end{pmatrix}$$

value = [ 5.4 2.2 ]  
colidx = [ 0 0 ]

Compute the memory requirement (# vals + # int)

36 vals + 36 int

length  
row \* num\_rows  
\_row \* num\_rows  
0.0 0.0 ]  
- - ]  
erhead)

# Sliced-ELL Format

Input  $A, x, y$     Output  $y = A \cdot x$

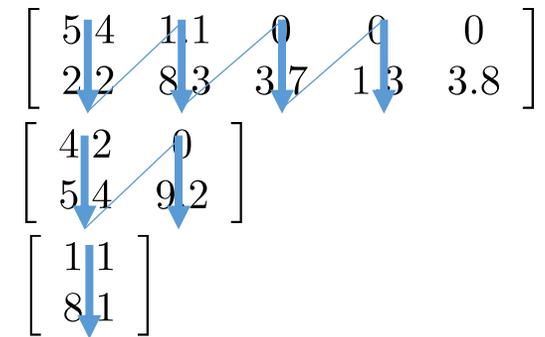
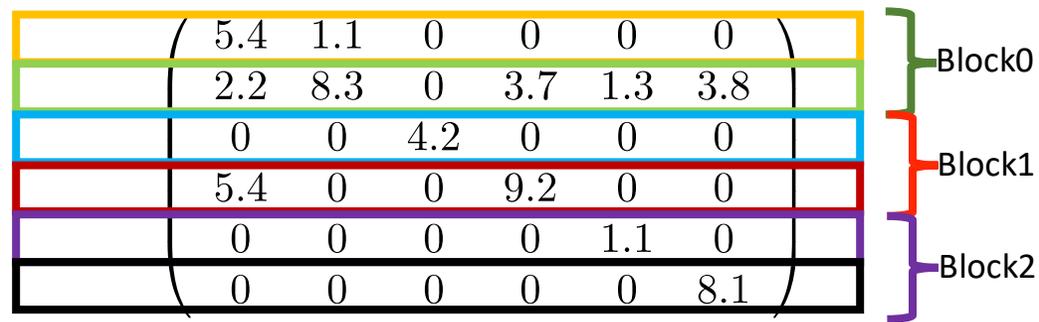
- Partition the matrix into blocks & use ELL for the distinct blocks.
  - Reduce overhead of ELL.
  - Can still store col-major.

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & 0 & 0 & 9.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8.1 \end{pmatrix}$$

# Sliced-ELL Format

Input  $A, x, y$     Output  $y = A \cdot x$

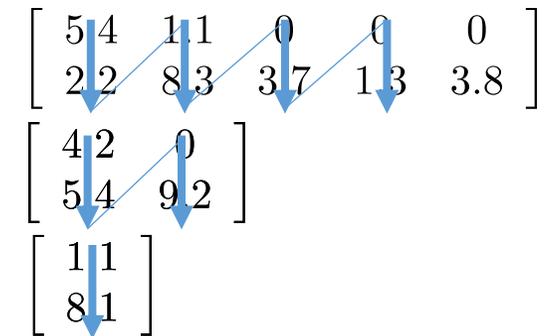
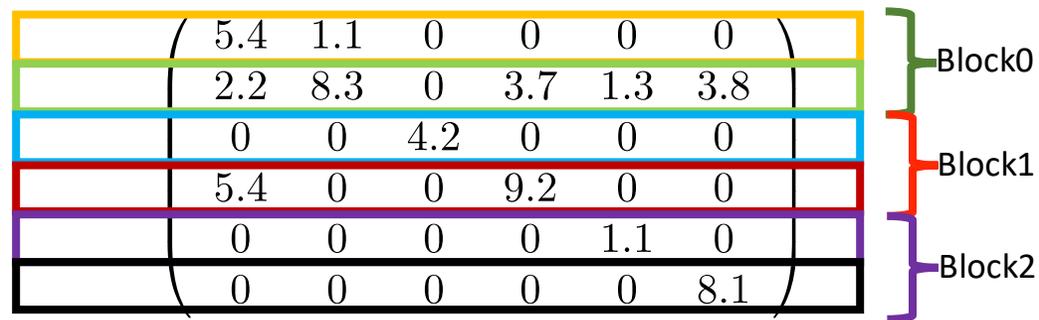
- Partition the matrix into blocks & use ELL for the distinct blocks.
  - Reduce overhead of ELL.
  - Can still store col-major.



# Sliced-ELL Format

Input  $A, x, y$     Output  $y = A \cdot x$

- Partition the matrix into blocks & use ELL for the distinct blocks.
  - Reduce overhead of ELL.
  - Can still store col-major.



value =	5.4	2.2	1.1	8.3	0.0	3.7	0.0	1.3	0.0	3.8	4.2	5.4	0.0	9.2	1.1	8.1
colidx =	0	0	1	1	-	3	-	4	-	5	2	0	-	3	4	5



Hands-on Exercise: Convert this matrix into Sliced-ELL format (SELL-2):

$$\begin{pmatrix} 0 & 0 & 0 & 4 & 2 & 0 \\ 0 & 2 & 3 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 2 & 3 & 4 & 3 & 1 \\ 0 & 2 & 0 & 0 & 0 & 1 \\ 1 & 2 & 3 & 0 & 0 & 1 \end{pmatrix}$$

Compute the memory requirement (# vals + # int)

Hands-on Exercise: Convert this matrix into Sliced-ELL format (SELL-2):

$$\begin{pmatrix} 0 & 0 & 0 & 4 & 2 & 0 \\ 0 & 2 & 3 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 2 & 3 & 4 & 3 & 1 \\ 0 & 2 & 0 & 0 & 0 & 1 \\ 1 & 2 & 3 & 0 & 0 & 1 \end{pmatrix}$$

Rowptr: 0 6 18 26

Compute the memory requirement (# vals + # int)

$$26 \text{ vals} + 26 + 4 \text{ int}$$

- How can we optimize this? Minimize the overhead? What is the overhead dependent on?
  - Bring rows with similar number of nonzero elements into the same block.
  - Sort rows by “length” and reorder the matrix, then convert to Sliced-ELL

Software and High-Performance Computing

## A Unified Sparse Matrix Data Format for Efficient General Sparse Matrix-Vector Multiplication on Modern Processors with Wide SIMD Units

Authors: Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, and Alan R. Bishop | [AUTHORS INFO & AFFILIATIONS](#)

<https://doi.org/10.1137/130930352>

- What happens for block-size 1?
- What happens for block size n (matrix size)?

# SpMV Formats and Kernels

$$\text{Input } A, x, y \quad \text{Output } y = A \cdot x$$

“Different kernels optimal for different problems”

## COO

- can compensate workload imbalance for irregular patterns
- Efficient for MIMD processing
- Strong support for atomics needed

## CSR

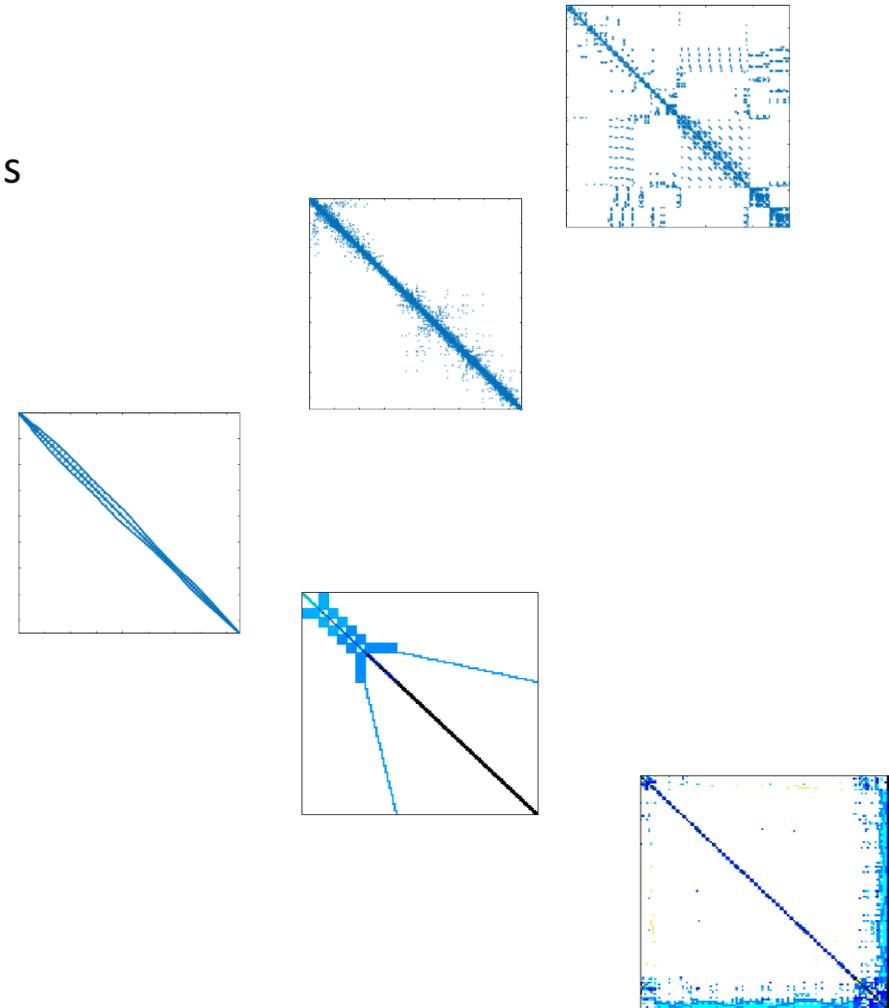
- small memory footprint
- Needs some logic for row-parallel processing
- Efficient for MIMD processing

## ELL

- Efficient for balanced matrices
- Enables col-major storage
- Efficient for SIMD processing

## SELL-c

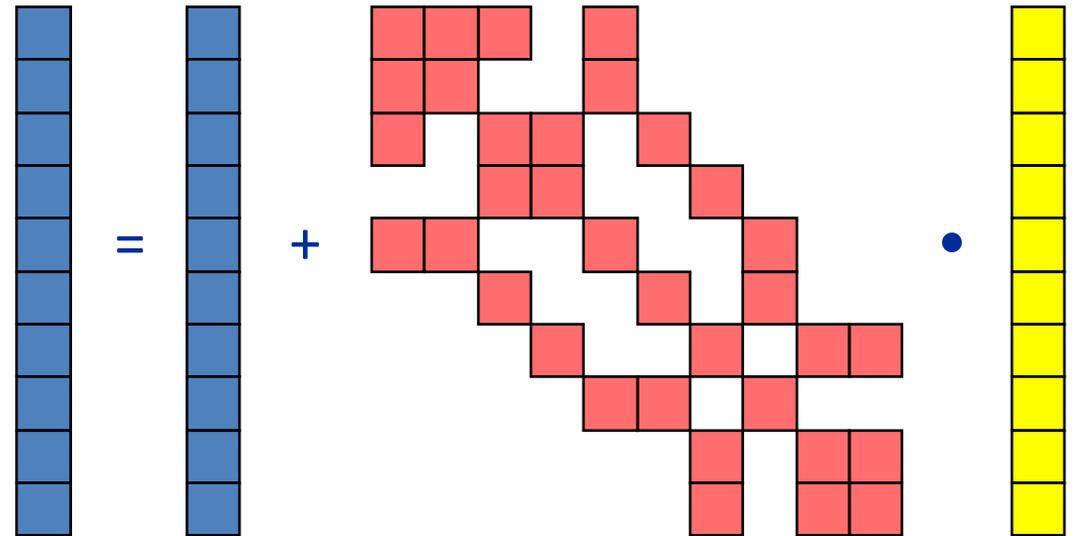
- Enables col-major storage
- Tunable between CSR and ELL



# SpMV (CSR) Roofline

Optimistic intensity:

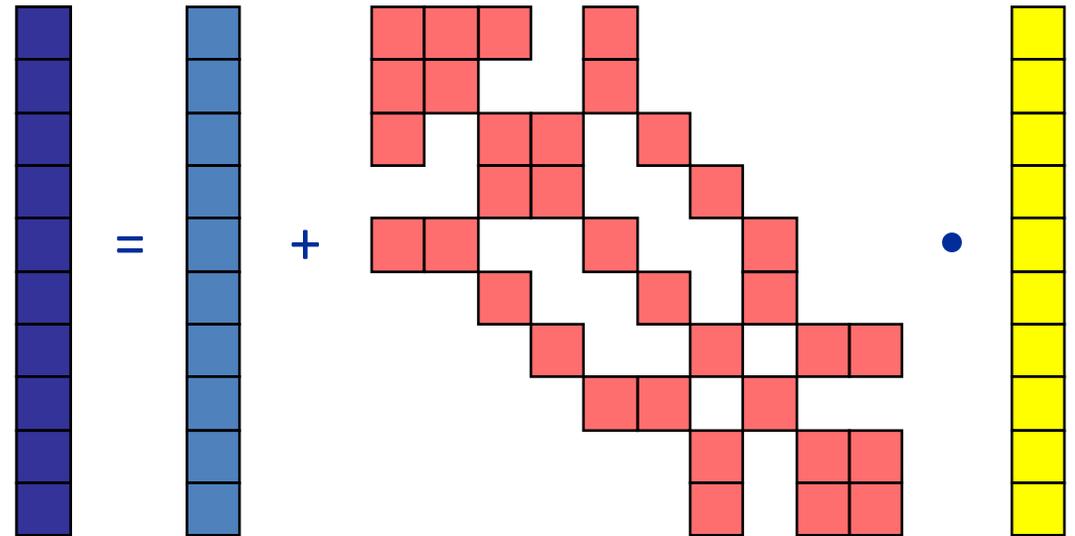
```
for (int row = 0; row < num_rows; ++row) {  
    double sum = 0.0;  
    for (int k = row_ptrs[row]; k < row_ptrs[row + 1]; ++k)  
        sum += mat_values[k] * b[col_idx[k]];  
    x[row] += sum;  
}
```



# SpMV (CSR) Roofline

Optimistic intensity:

```
for (int row = 0; row < num_rows; ++row) {  
    double sum = 0.0;  
    for (int k = row_ptrs[row]; k < row_ptrs[row + 1]; ++k)  
        sum += mat_values[k] * b[col_idx[k]];  
    x[row] += sum;  
}
```

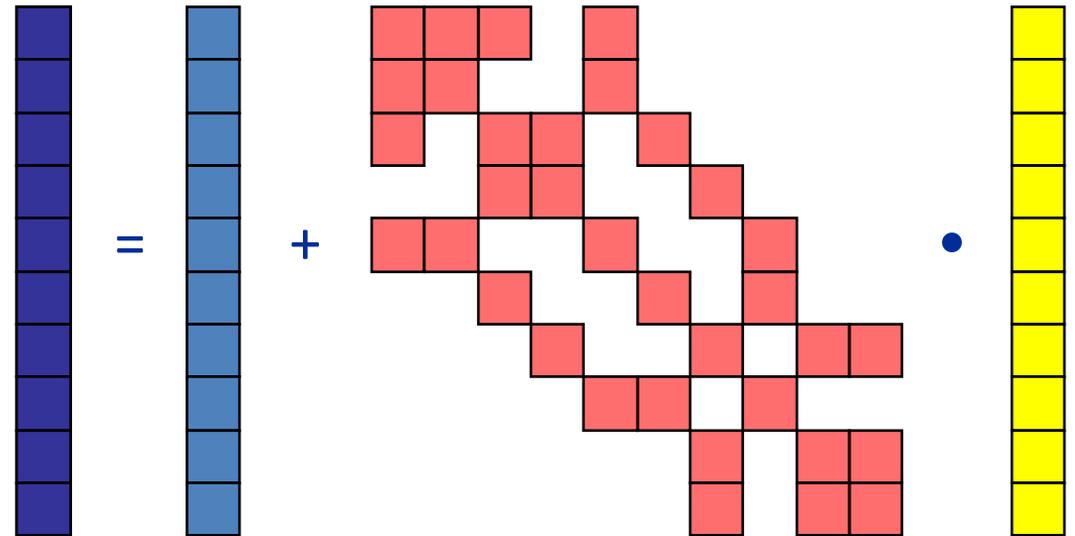


# SpMV (CSR) Roofline

Optimistic intensity:

$$I_{max} = \frac{2 N_{nz}}{12 N_{nz} + 20 N_r + 8 N_c} \frac{F}{B}$$

```
for (int row = 0; row < num_rows; ++row) {  
    double sum = 0.0;  
    for (int k = row_ptrs[row]; k < row_ptrs[row + 1]; ++k)  
        sum += mat_values[k] * b[col_idx[k]];  
    x[row] += sum;  
}
```

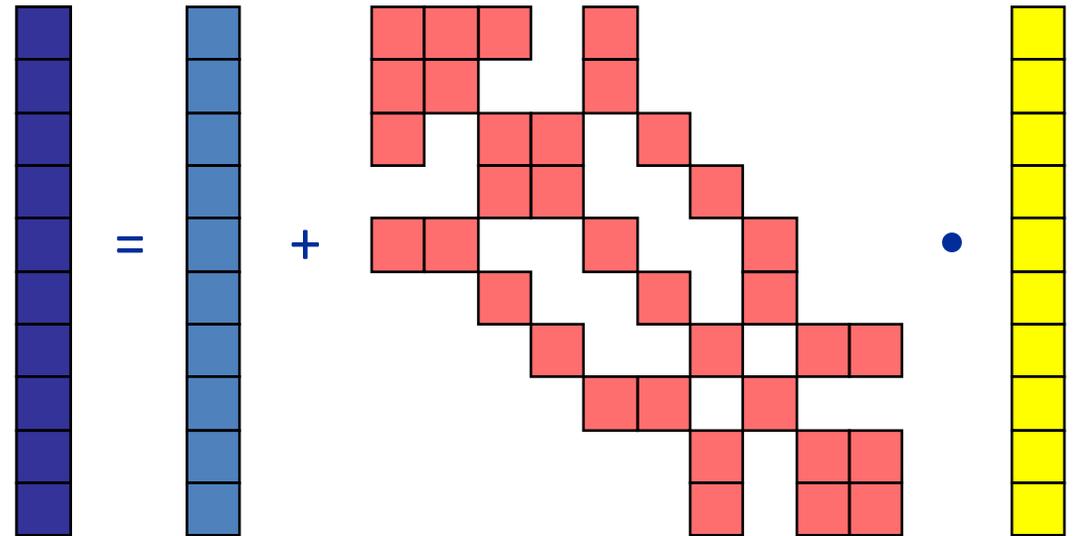


# SpMV (CSR) Roofline

```
for (int row = 0; row < num_rows; ++row) {  
    double sum = 0.0;  
    for (int k = row_ptrs[row]; k < row_ptrs[row + 1]; ++k)  
        sum += mat_values[k] * b[col_idx[k]];  
    x[row] += sum;  
}
```

Optimistic intensity:

$$I_{max} = \frac{2 N_{nz}}{12 N_{nz} + 20 N_r + 8 N_c} \frac{F}{B}$$
$$= \frac{1}{6 + 10/N_{nzc} + 4/N_{nzc}} \frac{F}{B}$$



# SpMV (CSR) Roofline

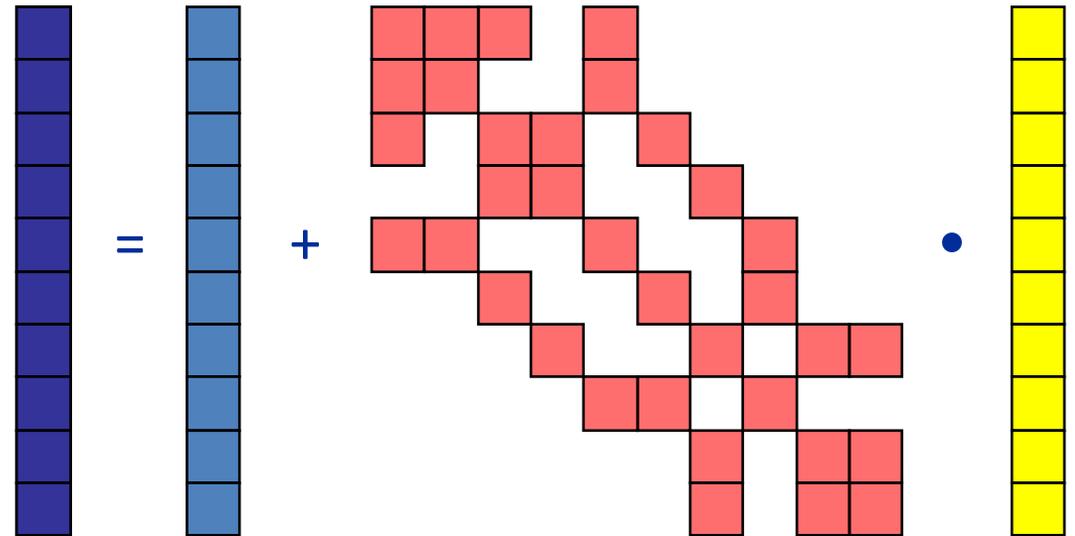
```
for (int row = 0; row < num_rows; ++row) {
    double sum = 0.0;
    for (int k = row_ptrs[row]; k < row_ptrs[row + 1]; ++k)
        sum += mat_values[k] * b[col_idx[k]];
    x[row] += sum;
}
```

Optimistic intensity:

$$I_{max} = \frac{2 N_{nz}}{12 N_{nz} + 20 N_r + 8 N_c} \frac{F}{B}$$

$$= \frac{1}{6 + 10/N_{nzc} + 4/N_{nzr}} \frac{F}{B}$$

square matrix



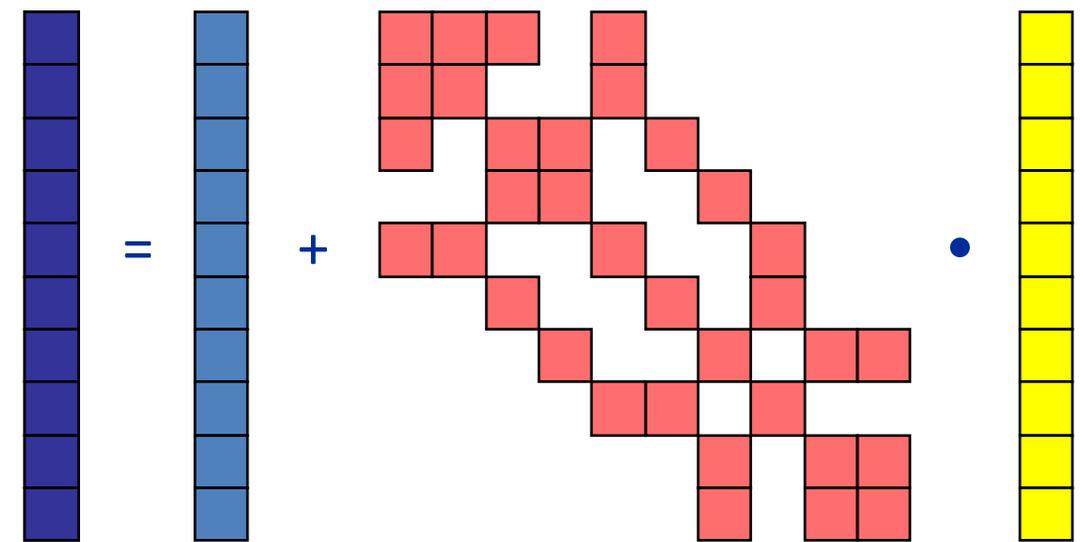
# SpMV (CSR) Roofline

```
for (int row = 0; row < num_rows; ++row) {
    double sum = 0.0;
    for (int k = row_ptrs[row]; k < row_ptrs[row + 1]; ++k)
        sum += mat_values[k] * b[col_idx[k]];
    x[row] += sum;
}
```

Optimistic intensity:

$$\begin{aligned}
 I_{max} &= \frac{2 N_{nz}}{12 N_{nz} + 20 N_r + 8 N_c} \frac{F}{B} \\
 &= \frac{1}{6 + 10/N_{nzc} + 4/N_{nzc}} \frac{F}{B} \\
 &= \frac{1}{6 + 10/N_{nzc} + 4/N_{nzc}} \frac{F}{B}
 \end{aligned}$$

square matrix



# SpMV (CSR) Roofline

Optimistic intensity:

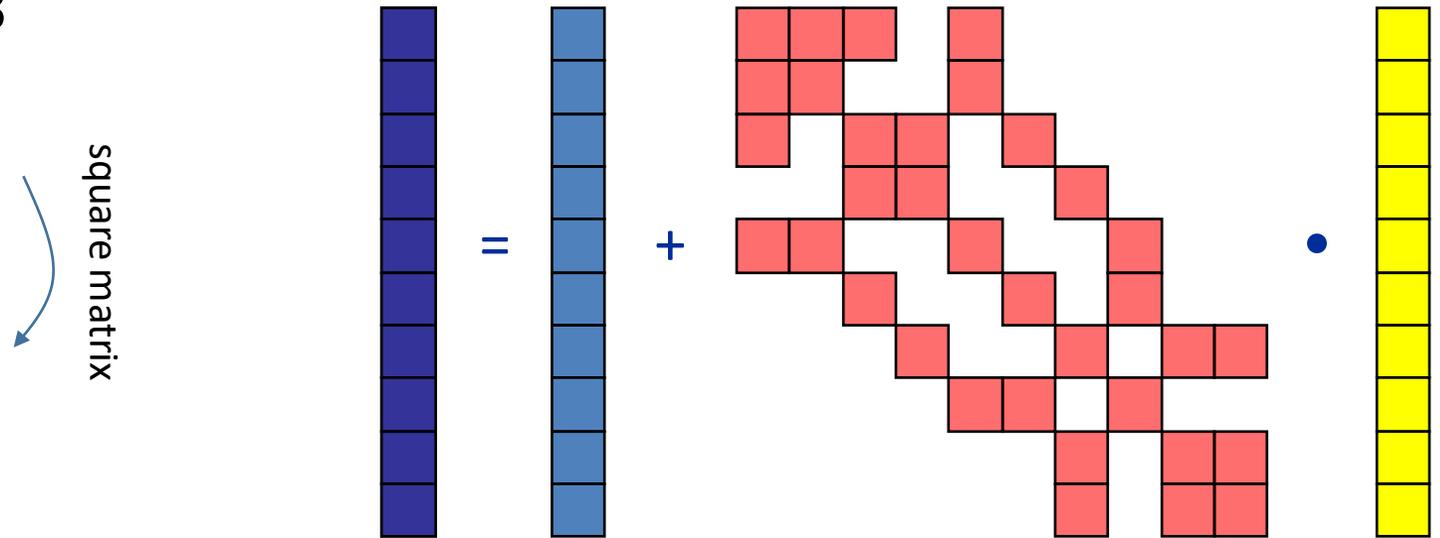
```
for (int row = 0; row < num_rows; ++row) {
    double sum = 0.0;
    for (int k = row_ptrs[row]; k < row_ptrs[row + 1]; ++k)
        sum += mat_values[k] * b[col_idxes[k]];
    x[row] += sum;
}
```

$$I_{max} = \frac{2 N_{nz}}{12 N_{nz} + 20 N_r + 8 N_c} \frac{F}{B}$$

$$= \frac{1}{6 + 10/N_{nzc} + 4/N_{nzc}} \frac{F}{B}$$

$$= \frac{1}{6 + 10/N_{nzc} + 4/N_{nzc}} \frac{F}{B}$$

$$\xrightarrow{N_{nzc} \gg 10} \frac{1}{6} \frac{F}{B}$$



# Roofline “failure” with SpMV

Reasons for performance not attaining the limit

1. Intensity lower than the minimum

- More RHS traffic than the optimistic limit ( $\frac{4}{N_{nzs}} B/F$ )

2. “Slow code”

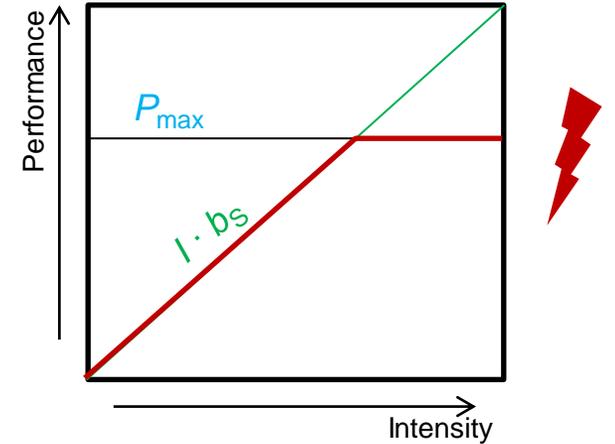
- “invisible” performance ceiling due to inefficient instructions or inefficient execution

3. Load imbalance

- A single process/thread cannot saturate the memory bandwidth

4. Erratic memory access patterns for RHS

- Latency dominates



# Roofline “failure” with SpMV

Reasons for performance not attaining the limit

1. Intensity lower than the minimum

- More RHS traffic than the optimistic limit ( $\frac{4}{N_{nzs}} B/F$ )

2. “Slow code”

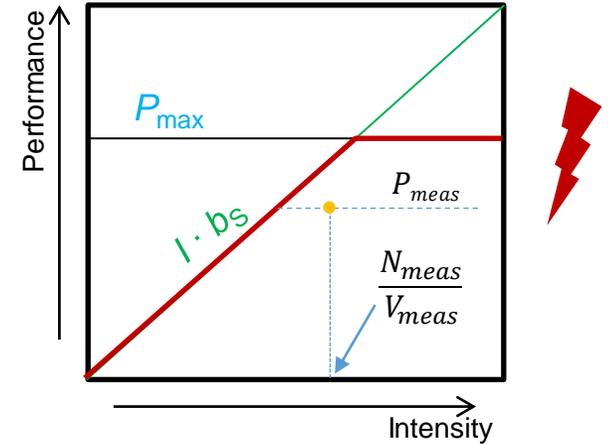
- “invisible” performance ceiling due to inefficient instructions or inefficient execution

3. Load imbalance

- A single process/thread cannot saturate the memory bandwidth

4. Erratic memory access patterns for RHS

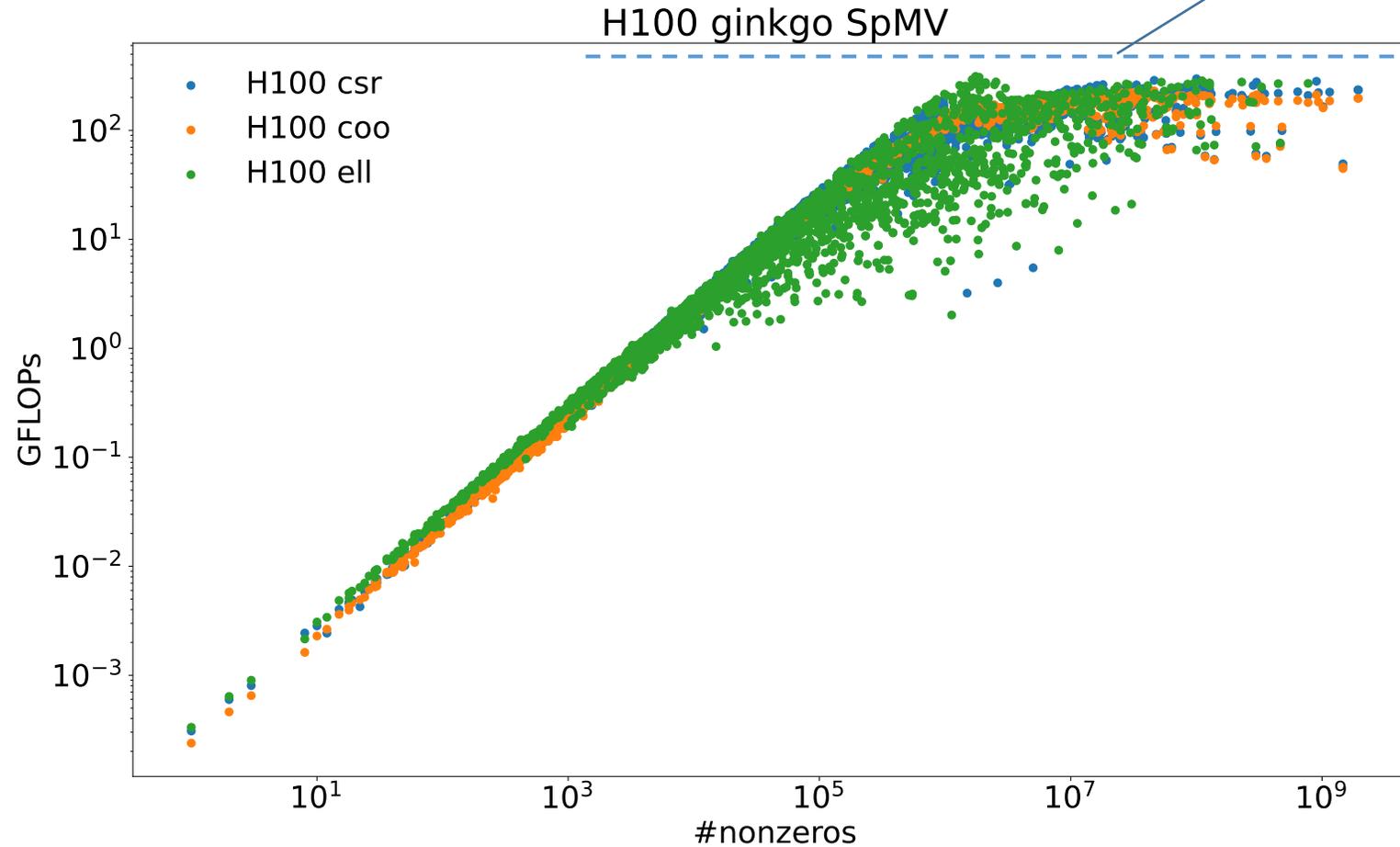
- Latency dominates



# Experiences with SpMV on GPUs

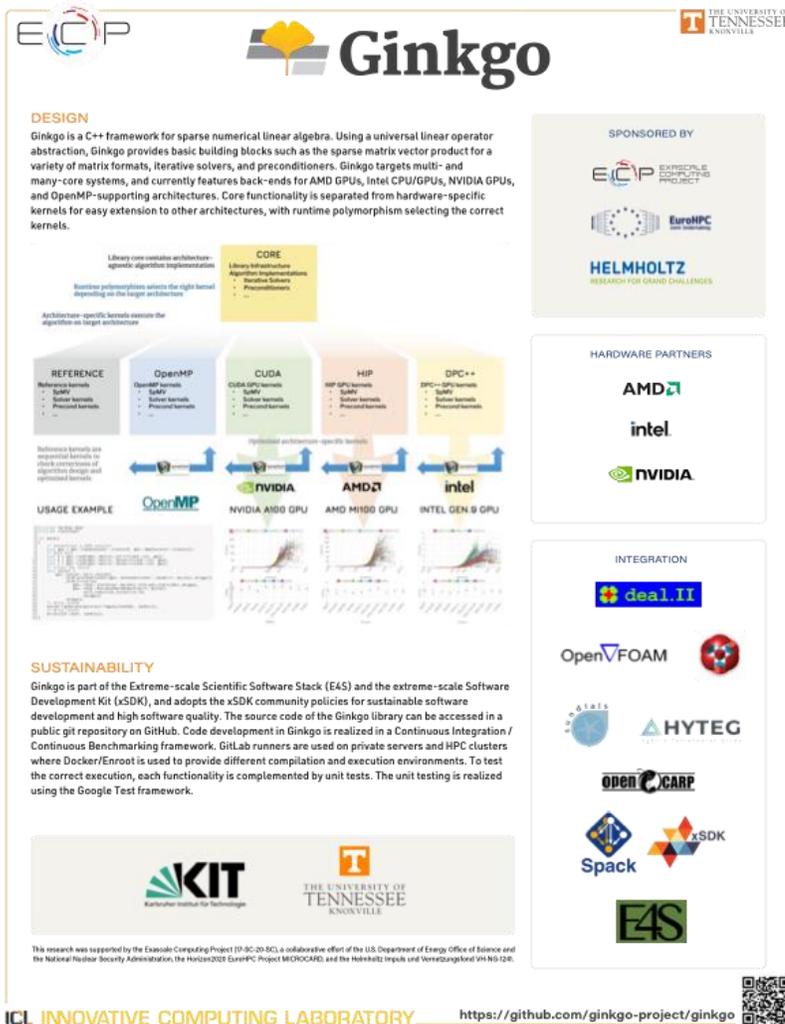
Looking at ~3,000 test matrices from Suite Sparse Matrix Collection

Absolute CSR limit for  
 $b_S = 3 \text{ TB/s}$



# Ginkgo Sparse Linear Algebra Library

<https://github.com/ginkgo-project/ginkgo>



**DESIGN**  
Ginkgo is a C++ framework for sparse numerical linear algebra. Using a universal linear operator abstraction, Ginkgo provides basic building blocks such as the sparse matrix vector product for a variety of matrix formats, iterative solvers, and preconditioners. Ginkgo targets multi- and many-core systems, and currently features back-ends for AMD GPUs, Intel CPU/GPUs, NVIDIA GPUs, and OpenMP-supporting architectures. Core functionality is separated from hardware-specific kernels for easy extension to other architectures, with runtime polymorphism selecting the correct kernels.

**SPONSORED BY**  
EICIP, EUROHPC, HELMHOLTZ RESEARCH FOR GRAND CHALLENGES

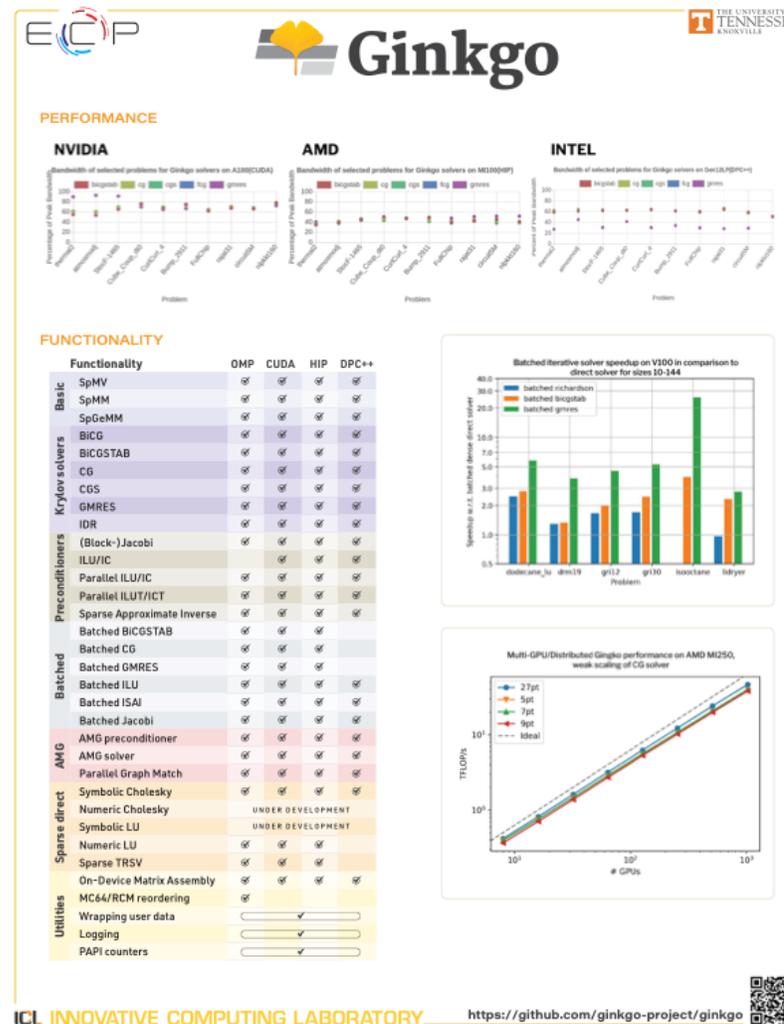
**HARDWARE PARTNERS**  
AMD, intel, NVIDIA

**INTEGRATION**  
deal.II, OpenFOAM, HYTEG, CARP, Spack, ASDK, E4S

**SUSTAINABILITY**  
Ginkgo is part of the Extreme-scale Scientific Software Stack (E4S) and the extreme-scale Software Development Kit (xSDK), and adopts the xSDK community policies for sustainable software development and high software quality. The source code of the Ginkgo library can be accessed in a public git repository on GitHub. Code development in Ginkgo is realized in a Continuous Integration / Continuous Benchmarking framework. GitLab runners are used on private servers and HPC clusters where Docker/Enroot is used to provide different compilation and execution environments. To test the correct execution, each functionality is complemented by unit tests. The unit testing is realized using the Google Test framework.

**KIT** Karlsruhe Institute of Technology  
**THE UNIVERSITY OF TENNESSEE** KNOXVILLE

This research was supported by the Exascale Computing Project (17-SC-20-RC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, the Horizon2020 Exascale Project MCR04RC, and the Helmholtz impulse and Verwendungsforum IMW-DZ.



**PERFORMANCE**

**NVIDIA** **AMD** **INTEL**

**FUNCTIONALITY**

Functionality	OMP	CUDA	HIP	DPC++
<b>Basic</b>				
SpMV	☑	☑	☑	☑
SpMM	☑	☑	☑	☑
SpGeMM	☑	☑	☑	☑
<b>Krylov solvers</b>				
BICG	☑	☑	☑	☑
BICGSTAB	☑	☑	☑	☑
CG	☑	☑	☑	☑
CGS	☑	☑	☑	☑
GMRES	☑	☑	☑	☑
IDR	☑	☑	☑	☑
(Block-)Jacobi	☑	☑	☑	☑
<b>Preconditioners</b>				
ILU/IC	☑	☑	☑	☑
Parallel ILU/IC	☑	☑	☑	☑
Parallel ILUT/ICT	☑	☑	☑	☑
Sparse Approximate Inverse	☑	☑	☑	☑
Batched BICGSTAB	☑	☑	☑	☑
Batched CG	☑	☑	☑	☑
Batched GMRES	☑	☑	☑	☑
Batched ILU	☑	☑	☑	☑
Batched ISAI	☑	☑	☑	☑
Batched Jacobi	☑	☑	☑	☑
<b>AMG</b>				
AMG preconditioner	☑	☑	☑	☑
AMG solver	☑	☑	☑	☑
Parallel Graph Match	☑	☑	☑	☑
<b>Sparse direct</b>				
Symbolic Cholesky	☑	☑	☑	☑
Numeric Cholesky	☑	☑	☑	☑
Symbolic LU	☑	☑	☑	☑
Numeric LU	☑	☑	☑	☑
Sparse TRSV	☑	☑	☑	☑
On-Device Matrix Assembly	☑	☑	☑	☑
<b>Utilities</b>				
MC64/RCM reordering	☑	☑	☑	☑
Wrapping user data	☑	☑	☑	☑
Logging	☑	☑	☑	☑
PAPI counters	☑	☑	☑	☑

**Batched iterative solver speedup on V100 in comparison to direct solver for sizes 3D-144**

**Multi-GPU/Distributed Ginkgo performance on AMD MI250, weak scaling of CG solver**

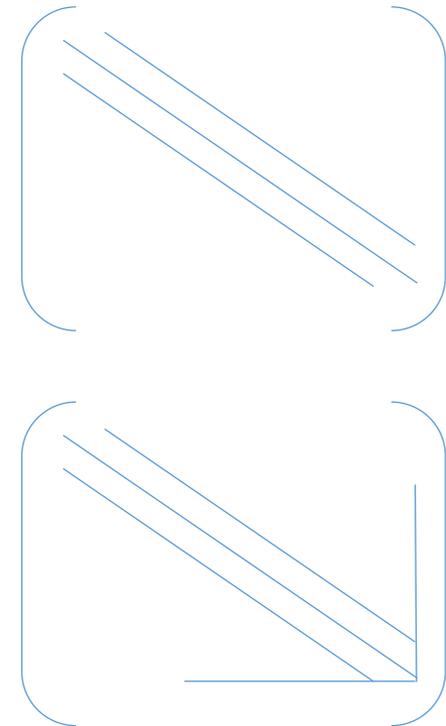
**ICL INNOVATIVE COMPUTING LABORATORY** <https://github.com/ginkgo-project/ginkgo>

# Ginkgo SpMV on different architectures

<https://github.com/ginkgo-project/ginkgo/tree/develop/benchmark/spmv>

benchmark/spmv/spmv -input '[{"stencil":"27pt","size":5000000}]' -executor cuda -formats csrc,coo,ell -repetitions auto

```
bash-5.1$ benchmark/spmv/spmv -input '[{"stencil":"27pt","size":5000000}]' -executor cuda -formats csrc,coo,ell -repetitions auto
This is Ginkgo 1.8.0 (develop)
  running with core module 1.8.0 (develop)
  the reference module is 1.8.0 (develop)
  the OpenMP module is 1.8.0 (develop)
  the CUDA module is 1.8.0 (develop)
  the HIP module is not compiled
  the DPCPP module is not compiled
Running on cuda(0)
Running with 2 warm iterations and adaptively determined repetitions with 10 <= rep <= 4294967295 and a minimal runtime of 0.05s
The random seed for right hand sides is 42
The formats are csrc,coo,ell
The number of right hand sides is 1
Running test case stencil(5000000, 27pt)
Matrix is of size (5000211, 5000211), 133432831
  Running spmv: csrc
  Running spmv: coo
  Running spmv: ell
{
  {
    "stencil": "27pt",
    "size": 5000000,
    "spmv": {
      "csrc": {
        "storage": 1621194820,
        "max_relative_norm2": 1.2278834839731986e-16,
        "time": 0.001414877317073181,
        "repetitions": 41,
        "completed": true
      },
      "coo": {
        "storage": 2134925296,
        "max_relative_norm2": 0.0,
        "time": 0.0013904711463414642,
        "repetitions": 41,
        "completed": true
      },
      "ell": {
        "storage": 1620068364,
        "max_relative_norm2": 2.0844231402369755e-16,
        "time": 0.0009703884838709675,
        "repetitions": 62,
        "completed": true
      }
    }
  },
  "rows": 5000211,
  "cols": 5000211,
  "nonzeros": 133432831,
  "optimal": {
    "spmv": "ell"
  }
}
}
ISC 2024
```



# Matrix Power Kernel (MPK)

- Calculate:  $y = A^p x$
- Repeatedly perform back-to-back SpMVs:

```
for k=1:p; do  
  y[k] = SpMV(A, y[k-1])  
done
```

- Matrix  $A$  loaded  $p$  times from main memory
- Potential for caching the matrix entries?
- → See last part of tutorial

C. L. Alappat et al.: *Level-based Blocking for Sparse Matrices: Sparse Matrix-Power-Vector Multiplication*. IEEE Transactions on Parallel and Distributed Systems **34**(2), 581-597 (2023), DOI: [10.1109/TPDS.2022.3223512](https://doi.org/10.1109/TPDS.2022.3223512)

# Matrix Power Kernel (MPK)

- Calculate:  $y = A^p x$
- Repeatedly perform back-to-back SpMV:

```
for k=1:p; do
  y[k] = SpMV(A, y[k-1])
done
```

- Matrix  $A$  loaded  $p$  times from main memory
- Potential for caching the matrix entries?
- → See last part of tutorial

y[0]



x

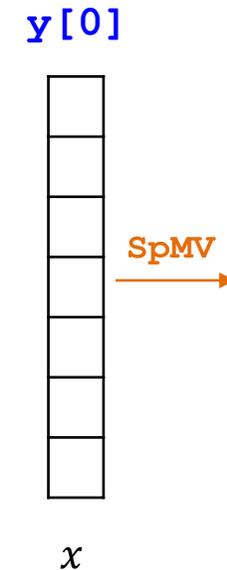
C. L. Alappat et al.: *Level-based Blocking for Sparse Matrices: Sparse Matrix-Power-Vector Multiplication*. IEEE Transactions on Parallel and Distributed Systems **34**(2), 581-597 (2023), DOI: [10.1109/TPDS.2022.3223512](https://doi.org/10.1109/TPDS.2022.3223512)

# Matrix Power Kernel (MPK)

- Calculate:  $y = A^p x$
- Repeatedly perform back-to-back SpMVs:

```
for k=1:p; do
  y[k] = SpMV(A, y[k-1])
done
```

- Matrix  $A$  loaded  $p$  times from main memory
- Potential for caching the matrix entries?
- → See last part of tutorial



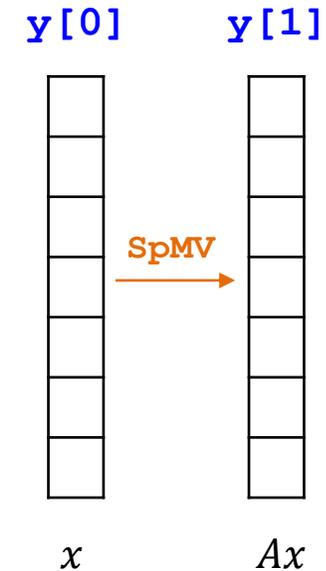
C. L. Alappat et al.: *Level-based Blocking for Sparse Matrices: Sparse Matrix-Power-Vector Multiplication*. IEEE Transactions on Parallel and Distributed Systems **34**(2), 581-597 (2023), DOI: [10.1109/TPDS.2022.3223512](https://doi.org/10.1109/TPDS.2022.3223512)

# Matrix Power Kernel (MPK)

- Calculate:  $y = A^p x$
- Repeatedly perform back-to-back SpMV:

```
for k=1:p; do  
  y[k] = SpMV(A, y[k-1])  
done
```

- Matrix  $A$  loaded  $p$  times from main memory
- Potential for caching the matrix entries?
- → See last part of tutorial



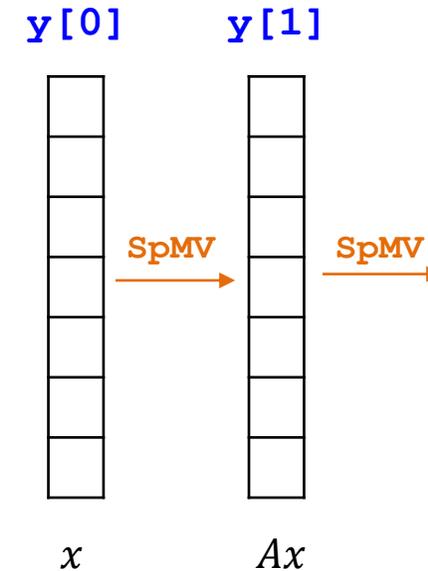
C. L. Alappat et al.: *Level-based Blocking for Sparse Matrices: Sparse Matrix-Power-Vector Multiplication*. IEEE Transactions on Parallel and Distributed Systems **34**(2), 581-597 (2023), DOI: [10.1109/TPDS.2022.3223512](https://doi.org/10.1109/TPDS.2022.3223512)

# Matrix Power Kernel (MPK)

- Calculate:  $y = A^p x$
- Repeatedly perform back-to-back SpMV:

```
for k=1:p; do  
  y[k] = SpMV(A, y[k-1])  
done
```

- Matrix  $A$  loaded  $p$  times from main memory
- Potential for caching the matrix entries?
- → See last part of tutorial



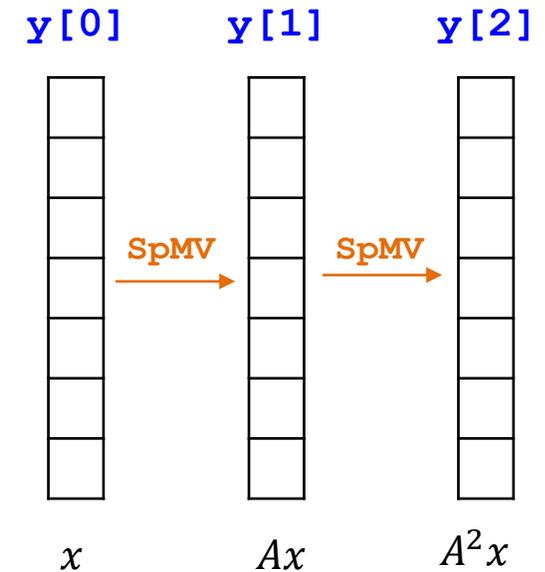
C. L. Alappat et al.: *Level-based Blocking for Sparse Matrices: Sparse Matrix-Power-Vector Multiplication*. IEEE Transactions on Parallel and Distributed Systems **34**(2), 581-597 (2023), DOI: [10.1109/TPDS.2022.3223512](https://doi.org/10.1109/TPDS.2022.3223512)

# Matrix Power Kernel (MPK)

- Calculate:  $y = A^p x$
- Repeatedly perform back-to-back SpMV:

```
for k=1:p; do  
  y[k] = SpMV(A, y[k-1])  
done
```

- Matrix  $A$  loaded  $p$  times from main memory
- Potential for caching the matrix entries?
- → See last part of tutorial



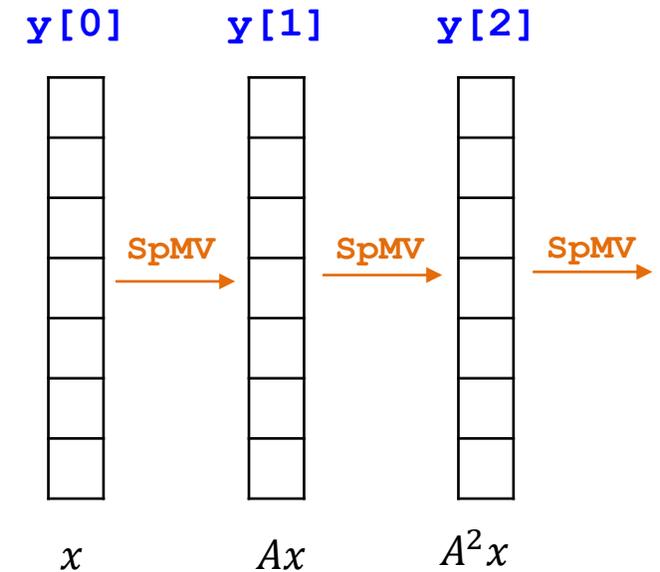
C. L. Alappat et al.: *Level-based Blocking for Sparse Matrices: Sparse Matrix-Power-Vector Multiplication*. IEEE Transactions on Parallel and Distributed Systems **34**(2), 581-597 (2023), DOI: [10.1109/TPDS.2022.3223512](https://doi.org/10.1109/TPDS.2022.3223512)

# Matrix Power Kernel (MPK)

- Calculate:  $y = A^p x$
- Repeatedly perform back-to-back SpMVs:

```
for k=1:p; do
  y[k] = SpMV(A, y[k-1])
done
```

- Matrix  $A$  loaded  $p$  times from main memory
- Potential for caching the matrix entries?
- → See last part of tutorial



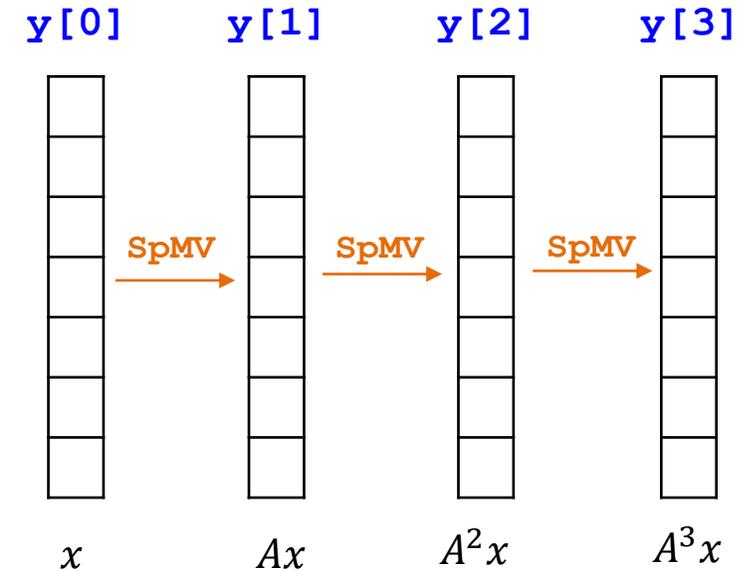
C. L. Alappat et al.: *Level-based Blocking for Sparse Matrices: Sparse Matrix-Power-Vector Multiplication*. IEEE Transactions on Parallel and Distributed Systems **34**(2), 581-597 (2023), DOI: [10.1109/TPDS.2022.3223512](https://doi.org/10.1109/TPDS.2022.3223512)

# Matrix Power Kernel (MPK)

- Calculate:  $y = A^p x$
- Repeatedly perform back-to-back SpMVs:

```
for k=1:p; do  
  y[k] = SpMV(A, y[k-1])  
done
```

- Matrix  $A$  loaded  $p$  times from main memory
- Potential for caching the matrix entries?
- → See last part of tutorial



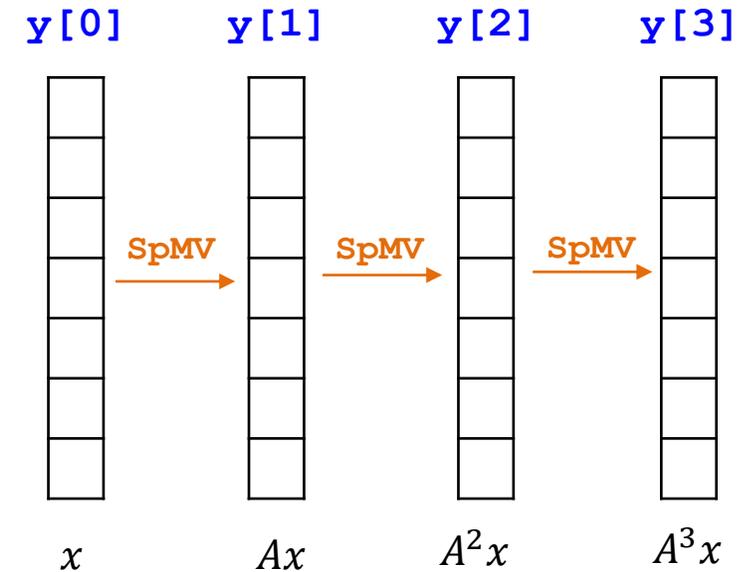
C. L. Alappat et al.: *Level-based Blocking for Sparse Matrices: Sparse Matrix-Power-Vector Multiplication*. IEEE Transactions on Parallel and Distributed Systems **34**(2), 581-597 (2023), DOI: [10.1109/TPDS.2022.3223512](https://doi.org/10.1109/TPDS.2022.3223512)

# Matrix Power Kernel (MPK)

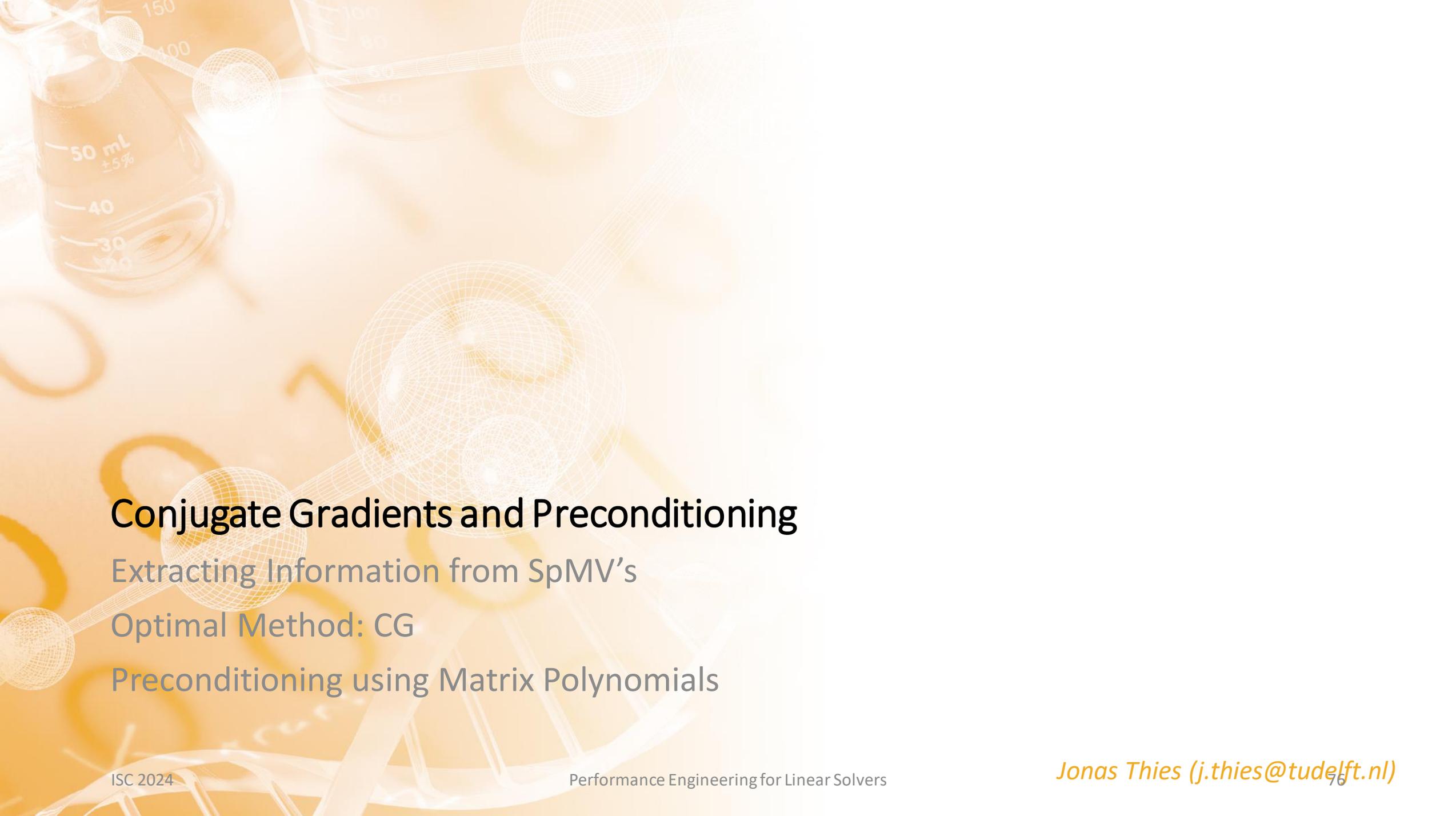
- Calculate:  $y = A^p x$
- Repeatedly perform back-to-back SpMV:

```
for k=1:p; do  
   $y[k] = \text{SpMV}(A, y[k-1])$   
done
```

- Matrix  $A$  loaded  $p$  times from main memory
- Potential for caching the matrix entries?
- → See last part of tutorial



C. L. Alappat et al.: *Level-based Blocking for Sparse Matrices: Sparse Matrix-Power-Vector Multiplication*. IEEE Transactions on Parallel and Distributed Systems **34**(2), 581-597 (2023), DOI: [10.1109/TPDS.2022.3223512](https://doi.org/10.1109/TPDS.2022.3223512)



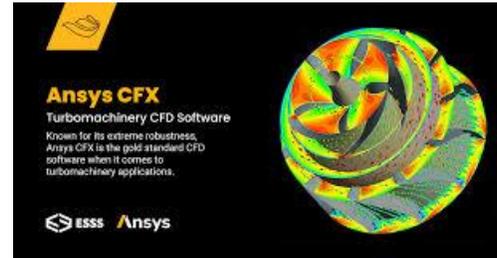
# Conjugate Gradients and Preconditioning

Extracting Information from SpMV's

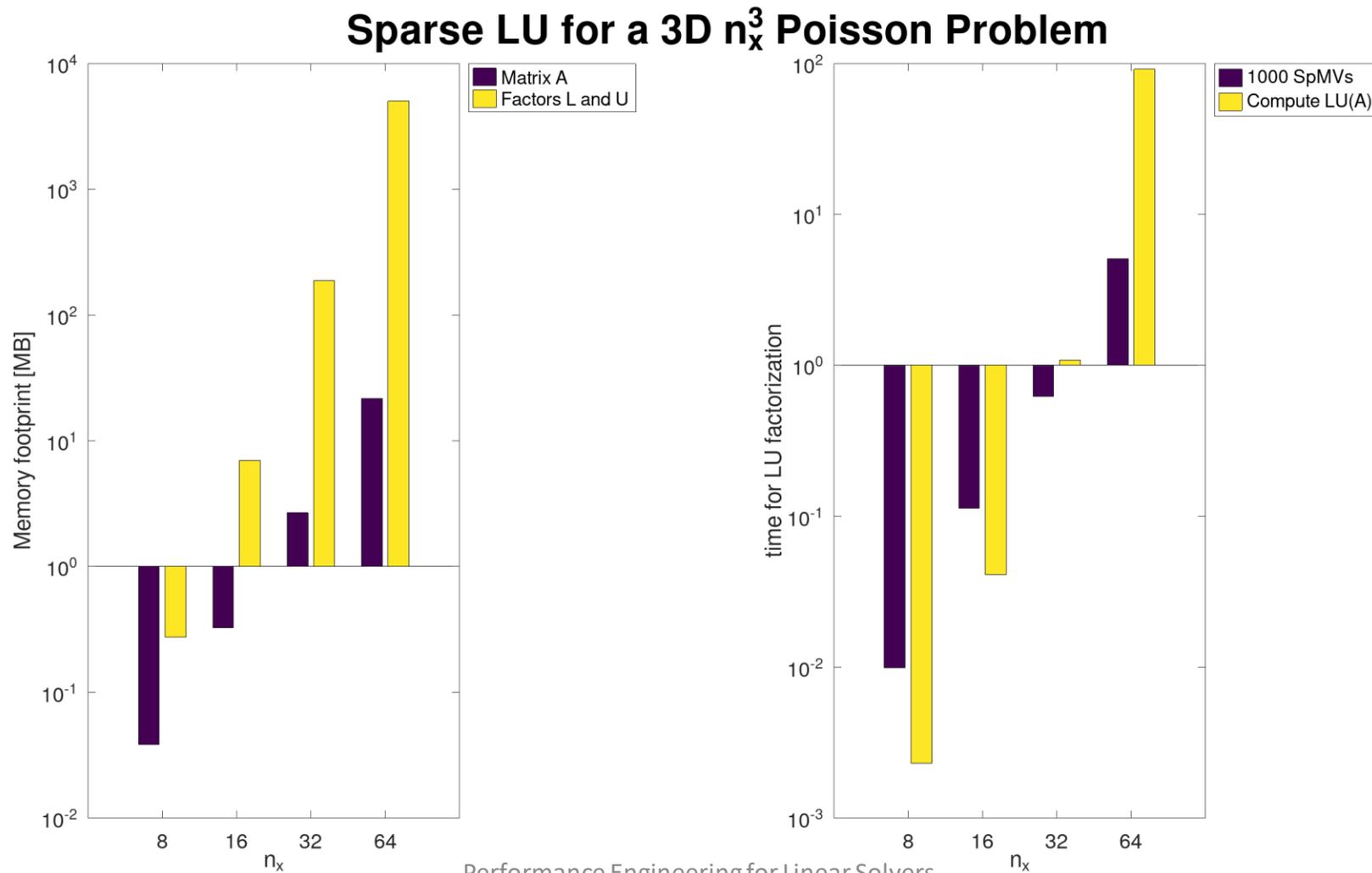
Optimal Method: CG

Preconditioning using Matrix Polynomials

# Motivating Iterative Linear Solvers



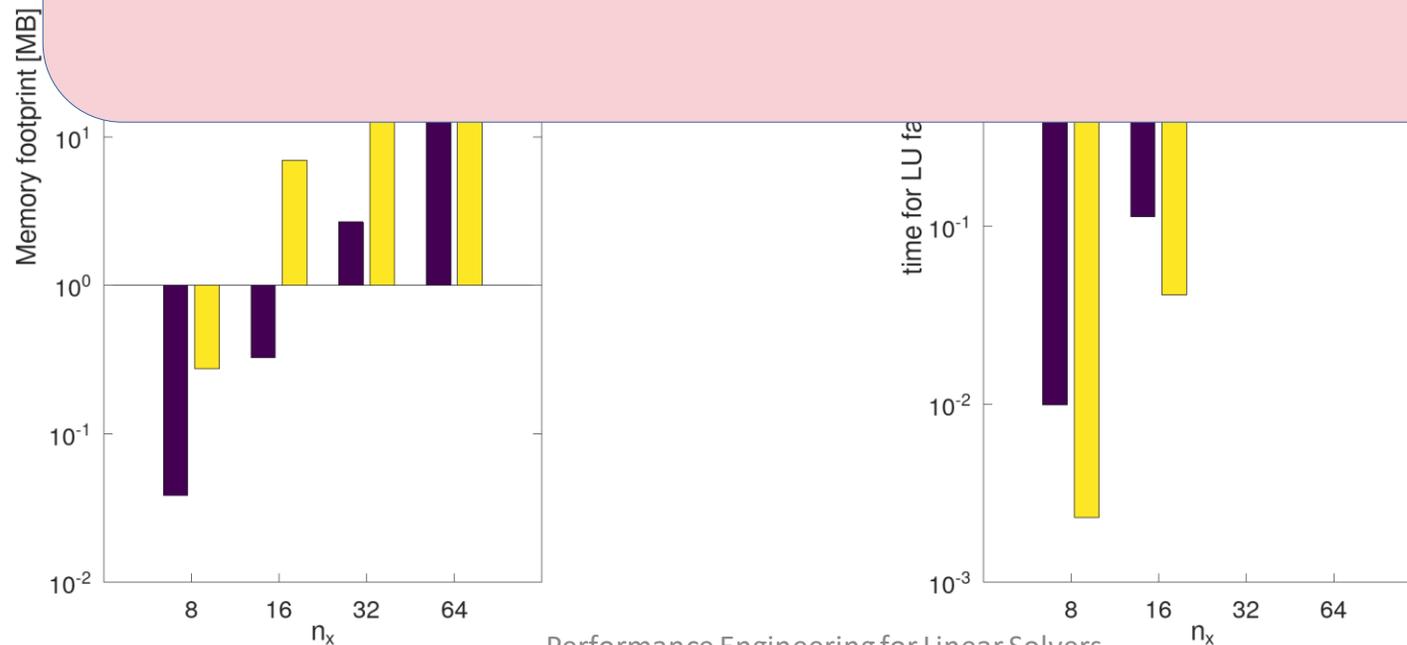
# Motivating Iterative Linear Solvers



# Motivating Iterative Linear Solvers

Sparse LU for a 2D  $n^3$  Poisson Problem

**As Direct Solvers are costly and full of Data Dependencies...**



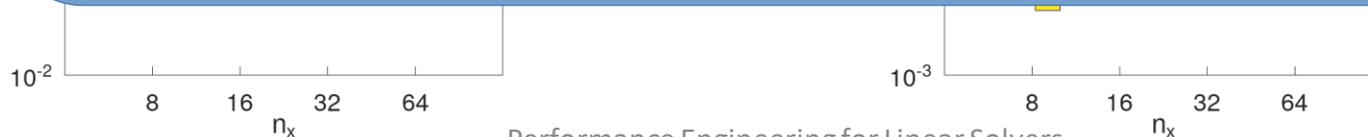
# Motivating Iterative Linear Solvers

Sparse LU for a 2D  $n^3$  Poisson Problem

**As Direct Solvers are costly and full of Data Dependencies...**

**How can we solve a linear system using Sparse Matrix-Vector (SpMV) products ?**

Memory footprint [MB]



# Richardson Iteration

# Richardson Iteration

The Geometric Series  $\frac{1}{1-x} = \sum_{k=0}^{\infty} x^k$  converges for  $|x| < 1$

# Richardson Iteration

The Geometric Series  $\frac{1}{1-x} = \sum_{k=0}^{\infty} x^k$  converges for  $|x| < 1$

The Neumann Series  $(I - A)^{-1} = \sum_{k=0}^{\infty} A^k$  converges for  $\|A\| < 1$

# Richardson Iteration

The Geometric Series  $\frac{1}{1-x} = \sum_{k=0}^{\infty} x^k$  converges for  $|x| < 1$

The Neumann Series  $(I - A)^{-1} = \sum_{k=0}^{\infty} A^k$  converges for  $\|A\| < 1$

The most simple iterative method one can imagine:  $x^{(k+1)} = x^{(k)} + (b - Ax^{(k)}) = b + (I - A)x^{(k)}$

# Richardson Iteration

The Geometric Series  $\frac{1}{1-x} = \sum_{k=0}^{\infty} x^k$  converges for  $|x| < 1$

The Neumann Series  $(I - A)^{-1} = \sum_{k=0}^{\infty} A^k$  converges for  $\|A\| < 1$

The most simple iterative method one can imagine:  
can also be written as  $x^{(k+1)} = x^{(k)} + (b - Ax^{(k)}) = b + (I - A)x^{(k)}$

$$r^{(0)} = b - Ax^{(0)}$$

$$x^{(k+1)} = x^{(k)} + r^{(k)}$$

$$r^{(k+1)} = (I - A)r^{(k)}$$

# Better Approximations

# Better Approximations

Richardson's method approximates  $x$  in the  $k$ 'th Krylov subspace

$$x^{(k)} \in x^{(0)} \cup \underbrace{\left\{ r^{(0)}, Ar^{(0)}, A^2r^{(0)}, \dots, A^{k-1}r^{(0)} \right\}}_{\mathcal{K}^{(k)}(A, r^{(0)})}$$

# Better Approximations

Richardson's method approximates  $x$  in the  $k$ 'th Krylov subspace

$$x^{(k)} \in x^{(0)} \cup \underbrace{\left\{ r^{(0)}, Ar^{(0)}, A^2r^{(0)}, \dots, A^{k-1}r^{(0)} \right\}}_{\mathcal{K}^{(k)}(A, r^{(0)})}$$

We can choose **better coefficients** by realizing that Krylov vectors tend to point more and more in the direction of the largest Eigenvector of  $A$

$$Av = \lambda v$$

# Better Approximations

Richardson's method approximates  $x$  in the  $k$ 'th Krylov subspace

$$x^{(k)} \in x^{(0)} \cup \underbrace{\left\{ r^{(0)}, Ar^{(0)}, A^2r^{(0)}, \dots, A^{k-1}r^{(0)} \right\}}_{\mathcal{K}^{(k)}(A, r^{(0)})}$$

We can choose **better coefficients** by realizing that Krylov vectors tend to point more and more in the direction of the largest Eigenvector of  $A$

$$Av = \lambda v$$

**Idea:** always add a direction orthogonal to the previous ones

# Arnoldi's Method

input:  $A, v_0$  with  $\|v_0\|_2 = 1$

$$V = v_0 / \|v_0\|_2$$

**for**  $k = 0, 1, \dots$  **do**

$$v_{k+1} = Av_k$$

**for**  $j = 0, 1, \dots, k$  **do**

$$h_{j,k} = v_j^T v_{k+1}$$

$$v_{k+1} = v_{k+1} - h_{j,k} v_j$$

**end for**

$$h_{k+1,k} = 1 / \|v_{k+1}\|_2$$

$$V = [V, v_{k+1} \cdot h_{k+1,k}]$$

**end for**

# Arnoldi's Method

```
input:  $A, v_0$  with  $\|v_0\|_2 = 1$   
 $V = v_0 / \|v_0\|_2$   
for  $k = 0, 1, \dots$  do  
     $v_{k+1} = Av_k$   
    for  $j = 0, 1, \dots, k$  do  
         $h_{j,k} = v_j^T v_{k+1}$   
         $v_{k+1} = v_{k+1} - h_{j,k} v_j$   
    end for  
     $h_{k+1,k} = 1 / \|v_{k+1}\|_2$   
     $V = [V, v_{k+1} \cdot h_{k+1,k}]$   
end for
```

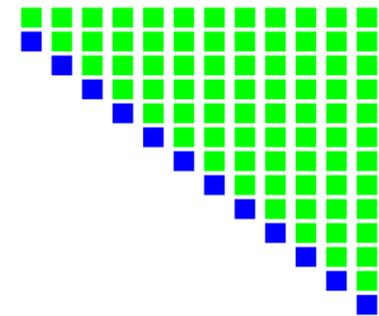
Where  $V$  is orthonormal:  $V_k^T V_k = I,$

# Arnoldi's Method

```
input:  $A, v_0$  with  $\|v_0\|_2 = 1$   
 $V = v_0 / \|v_0\|_2$   
for  $k = 0, 1, \dots$  do  
   $v_{k+1} = Av_k$   
  for  $j = 0, 1, \dots, k$  do  
     $h_{j,k} = v_j^T v_{k+1}$   
     $v_{k+1} = v_{k+1} - h_{j,k} v_j$   
  end for  
   $h_{k+1,k} = 1 / \|v_{k+1}\|_2$   
   $V = [V, v_{k+1} \cdot h_{k+1,k}]$   
end for
```

Where  $V$  is orthonormal:  $V_k^T V_k = I$ ,

and  $H_{k,k} = h_{0:k,0:k}$  is “upper Hessenberg”

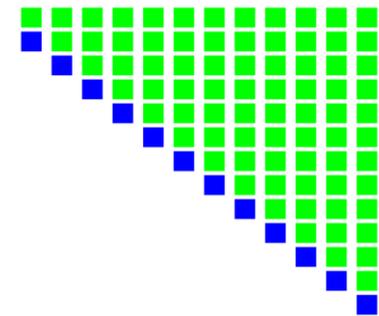


# Arnoldi's Method

```
input:  $A, v_0$  with  $\|v_0\|_2 = 1$   
 $V = v_0 / \|v_0\|_2$   
for  $k = 0, 1, \dots$  do  
   $v_{k+1} = Av_k$   
  for  $j = 0, 1, \dots, k$  do  
     $h_{j,k} = v_j^T v_{k+1}$   
     $v_{k+1} = v_{k+1} - h_{j,k} v_j$   
  end for  
   $h_{k+1,k} = 1 / \|v_{k+1}\|_2$   
   $V = [V, v_{k+1} \cdot h_{k+1,k}]$   
end for
```

Where  $V$  is orthonormal:  $V_k^T V_k = I$ ,

and  $H_{k,k} = h_{0:k,0:k}$  is “upper Hessenberg”



They satisfy the **Arnoldi-relation**

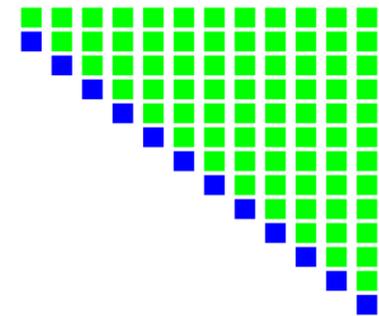
$$AV_k = V_{k+1} H_{k+1,k}$$

# Arnoldi's Method

```
input:  $A, v_0$  with  $\|v_0\|_2 = 1$   
 $V = v_0 / \|v_0\|_2$   
for  $k = 0, 1, \dots$  do  
   $v_{k+1} = Av_k$   
  for  $j = 0, 1, \dots, k$  do  
     $h_{j,k} = v_j^T v_{k+1}$   
     $v_{k+1} = v_{k+1} - h_{j,k} v_j$   
  end for  
   $h_{k+1,k} = 1 / \|v_{k+1}\|_2$   
   $V = [V, v_{k+1} \cdot h_{k+1,k}]$   
end for
```

Where  $V$  is orthonormal:  $V_k^T V_k = I$ ,

and  $H_{k,k} = h_{0:k,0:k}$  is “upper Hessenberg”



They satisfy the **Arnoldi-relation**

$$AV_k = V_{k+1} H_{k+1,k}$$

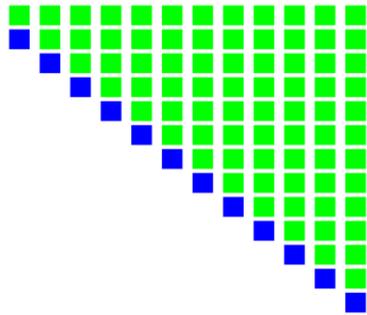
By construction, the eigenvalues of  $H_{k,k}$  (“*Ritz Values*”) approximate those of  $A$  as  $k \rightarrow N$

If  $A$  is Symmetric...

# If $A$ is Symmetric...

$$H_{k,k} = V_k^T A V_k$$

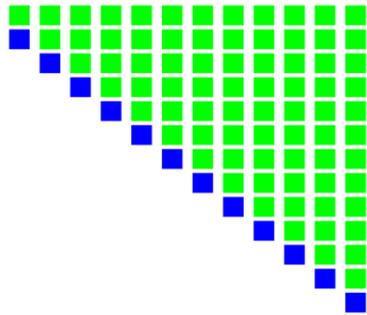
is (still) upper Hessenberg



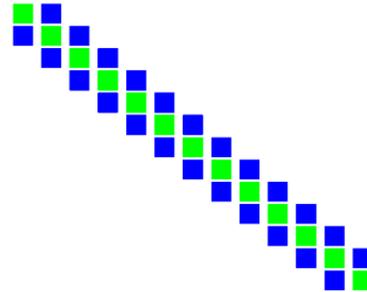
# If $A$ is Symmetric...

$$H_{k,k} = V_k^T A V_k$$

is (still) upper Hessenberg



...and symmetric!

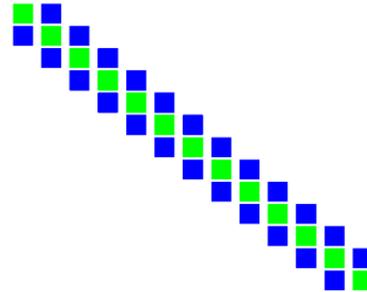
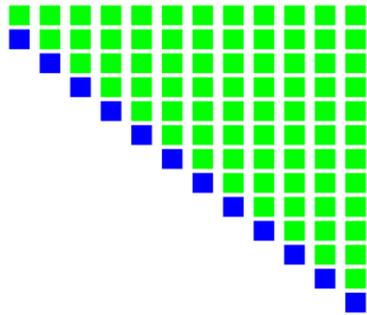


# If A is Symmetric...

$$H_{k,k} = V_k^T A V_k$$

is (still) upper Hessenberg

...and symmetric!

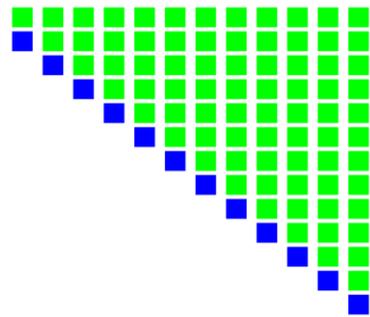


**Only need to orthogonalize against two vectors!  
("short recurrence")**

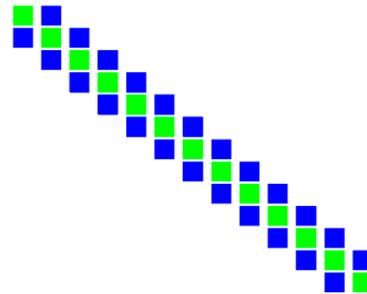
# If A is Symmetric...

$$H_{k,k} = V_k^T A V_k$$

is (still) upper Hessenberg



...and symmetric!



**Only need to orthogonalize against two vectors!**  
(“short recurrence”)

## Lanczos algorithm:

input:  $A, v_0$

$$V = v_0 / \|v_0\|_2$$

$$\beta_0 = 0$$

**for**  $k = 0, 1, \dots$  **do**

$$v_{k+1} = A v_k$$

$$\alpha_k = v_k^T v_{k+1}$$

$$v_{k+1} = v_{k+1} - \alpha_k v_k - \beta_k v_{k-1}$$

$$\beta_{k+1} = \|v_{k+1}\|_2$$

$$V = [V, v_{k+1}/\beta_{k+1}]$$

**end for**

# Conjugate Gradient Algorithm

Based on the Lanczos method, one can derive an iterative linear solver

For **symmetric and positive definite (spd)** matrices

That **minimizes the error** in every iteration

And only requires storing **three additional vectors**

```
input:  $A, b, x_0, it_{max}$   
 $r_0 = b - Ax_0$   
 $p_0 = r_0$   
for  $k = 0, \dots, it_{max}$  do  
     $\alpha_k = \frac{r_k^T r_k}{p_k^T A p_k}$   
     $x_{k+1} = x_k + \alpha_k p_k$   
     $r_{k+1} = r_k - \alpha_k A p_k$   
     $\beta_k = \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$   
     $p_{k+1} = r_{k+1} + \beta_k p_k$   
end for
```

# CG Implementation

```
def cg_solve(A, b, x0, tol, maxit):
    #<create x, p, q and r>
    axpby(1.0,x0,0.0,x)           # x = x0
    spmv(A, x, r); axpby(1.0, b, -1.0, r) # r = b - Ax
    axpby(1.0, r, 0.0, p)         # p = r
    rho = dot(r,r); rho_old = 1.0
    for it in range(maxit+1):
        if rho < tol*tol:
            break;                # check for convergence
        spmv(A, p, q)             # q = A*p
        alpha = rho/dot(p,q)
        axpby(alpha, p, 1.0, x)   # x = x+alpha*p
        axpby(-alpha, q, 1.0, r) # r = r - alpha*q
        rho_old = rho; rho = dot(r, r)
        beta = rho / rho_old
        axpby (1.0, r, beta, p)   # p = r+beta*p
    return x
```

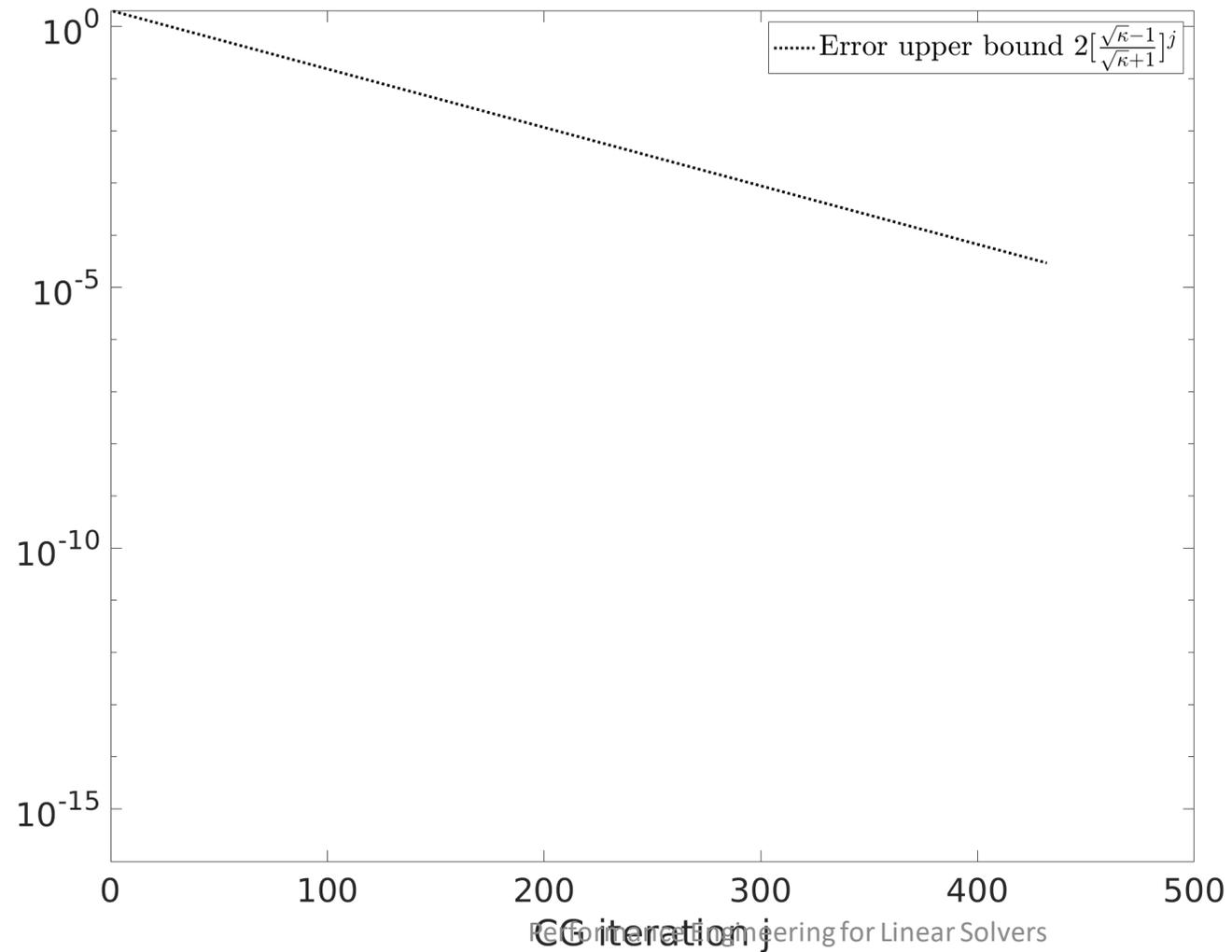
# Convergence & Spectrum

Lower bound on convergence rate:  $\|x_k - x\|_2 \leq 2 \left[ \frac{\sqrt{\kappa_2(A)} - 1}{\sqrt{\kappa_2(A)} + 1} \right]^k \|x_0 - x\|_2$

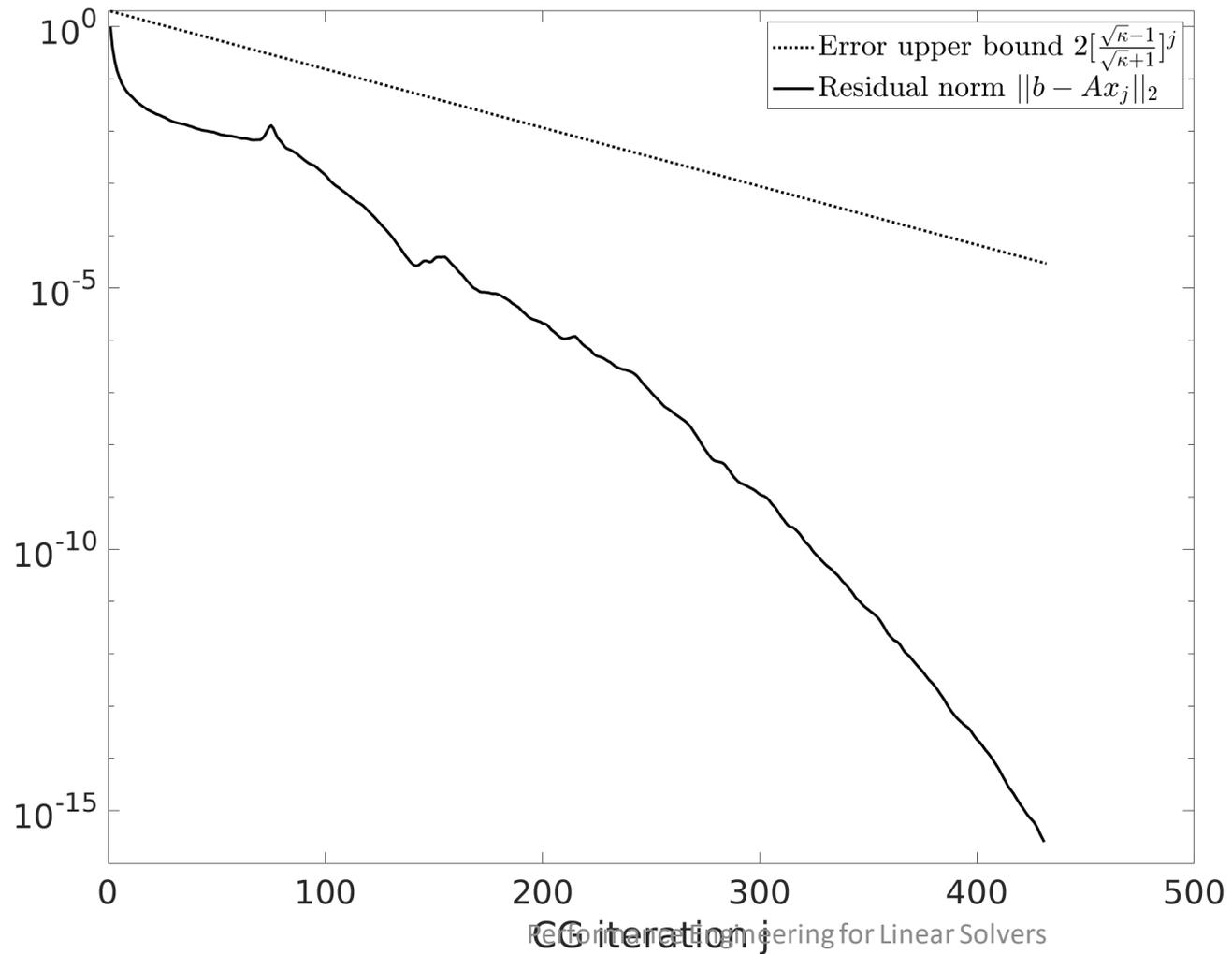
Where  $\kappa_2(A) = \frac{\lambda_{\min}(A)}{\lambda_{\max}(A)}$  is called the *spectral condition number*.

**In practice, this bound is very pessimistic:** Whenever an extreme Ritz value converges, it no longer affects convergence

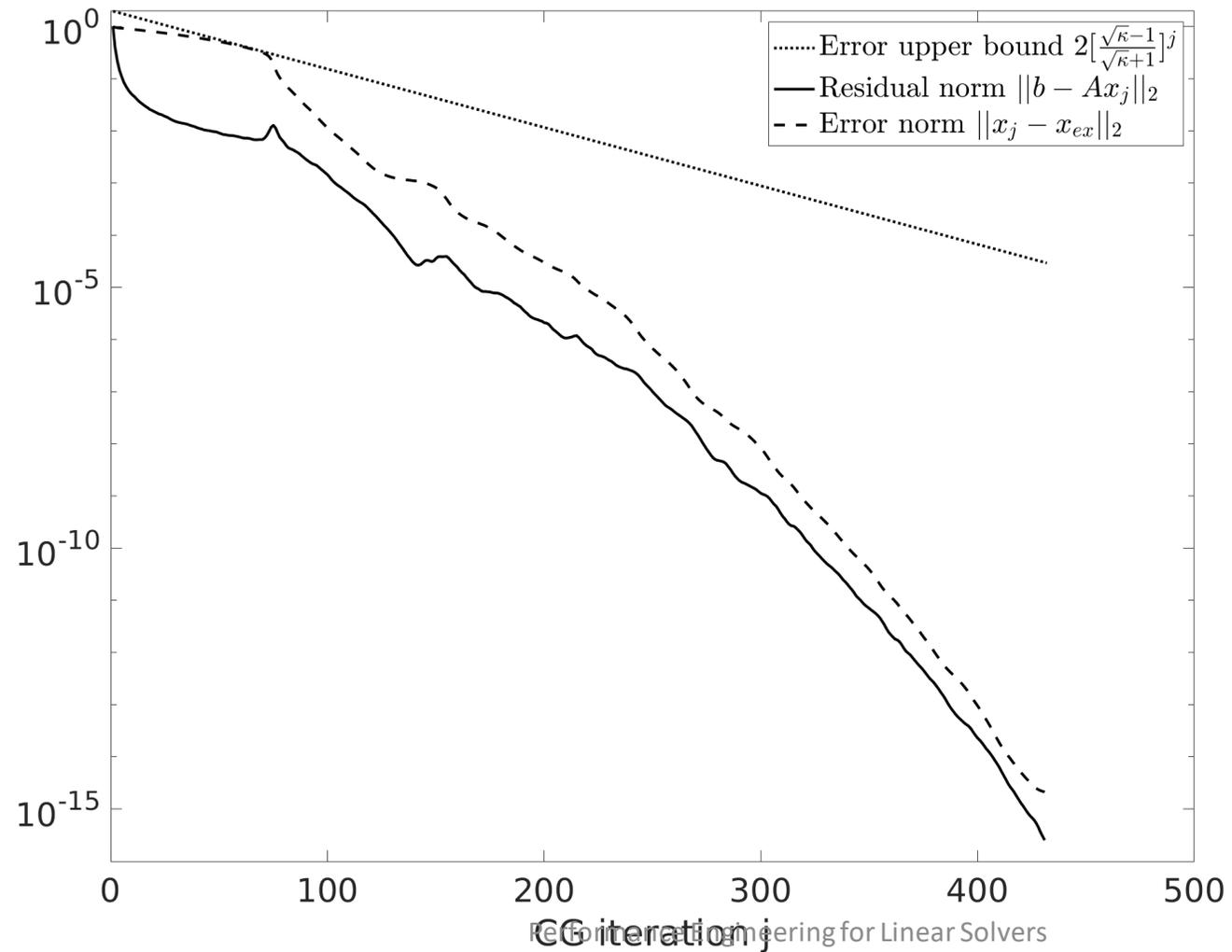
# Convergence & Spectrum



# Convergence & Spectrum



# Convergence & Spectrum



# Preconditioning, General Idea

We can solve one of the related linear systems

$$M_1^{-1}Ax = M_1^{-1}b$$

(left preconditioning)

$$AM_2^{-1}y = b$$

$$x = M_2^{-1}y$$

(right preconditioning)

$$M_1^{-1}AM_2^{-1}y = M_1^{-1}b$$

$$x = M_2^{-1}y$$

(two-sided preconditioning)

# Preconditioning, General Idea

We can solve one of the related linear systems

$$M_1^{-1}Ax = M_1^{-1}b$$

(left preconditioning)

$$AM_2^{-1}y = b$$

$$x = M_2^{-1}y$$

(right preconditioning)

$$M_1^{-1}AM_2^{-1}y = M_1^{-1}b$$

$$x = M_2^{-1}y$$

(two-sided preconditioning)

Construction  
and application  
of M-operators  
should be cheap

# Preconditioning, General Idea

We can solve one of the related linear systems

$$M_1^{-1}Ax = M_1^{-1}b$$

(left preconditioning)

$$AM_2^{-1}y = b$$

$$x = M_2^{-1}y$$

(right preconditioning)

$$M_1^{-1}AM_2^{-1}y = M_1^{-1}b$$

$$x = M_2^{-1}y$$

(two-sided preconditioning)

Construction  
and application  
of M-operators  
should be cheap

Resulting  
operator  
should have  
more favorable  
spectrum

# Preconditioning, General Idea

We can solve one of the related linear systems

$$M_1^{-1}Ax = M_1^{-1}b$$

(left preconditioning)

Construction and application of M-operators should be cheap

$$AM_2^{-1}y = b$$

$$x = M_2^{-1}y$$

(right preconditioning)

Resulting operator should have more favorable spectrum

$$M_1^{-1}AM_2^{-1}y = M_1^{-1}b$$

$$x = M_2^{-1}y$$

(two-sided preconditioning)

Additional work should be amortized by faster convergence

# Preconditioning, General Idea

We can solve one of the related linear systems

$$M_1^{-1}Ax = M_1^{-1}b$$

(left preconditioning)

Construction and application of M-operators should be cheap

$$AM_2^{-1}y = b$$

$$x = M_2^{-1}y$$

(right preconditioning)

Resulting operator should have more favorable spectrum

$$M_1^{-1}AM_2^{-1}y = M_1^{-1}b$$

$$x = M_2^{-1}y$$

(two-sided preconditioning)

Additional work should be amortized by faster convergence

Required operations should run well on target hardware

# A Simple Polynomial Preconditioner

Let

$$D := \text{diag}(A), \quad A_1 = \sqrt{D^{-1}} A \sqrt{D^{-1}}$$
$$L := I - \text{tril}(A_1), \quad \text{s.t. } A_1 = I - L - L^T$$

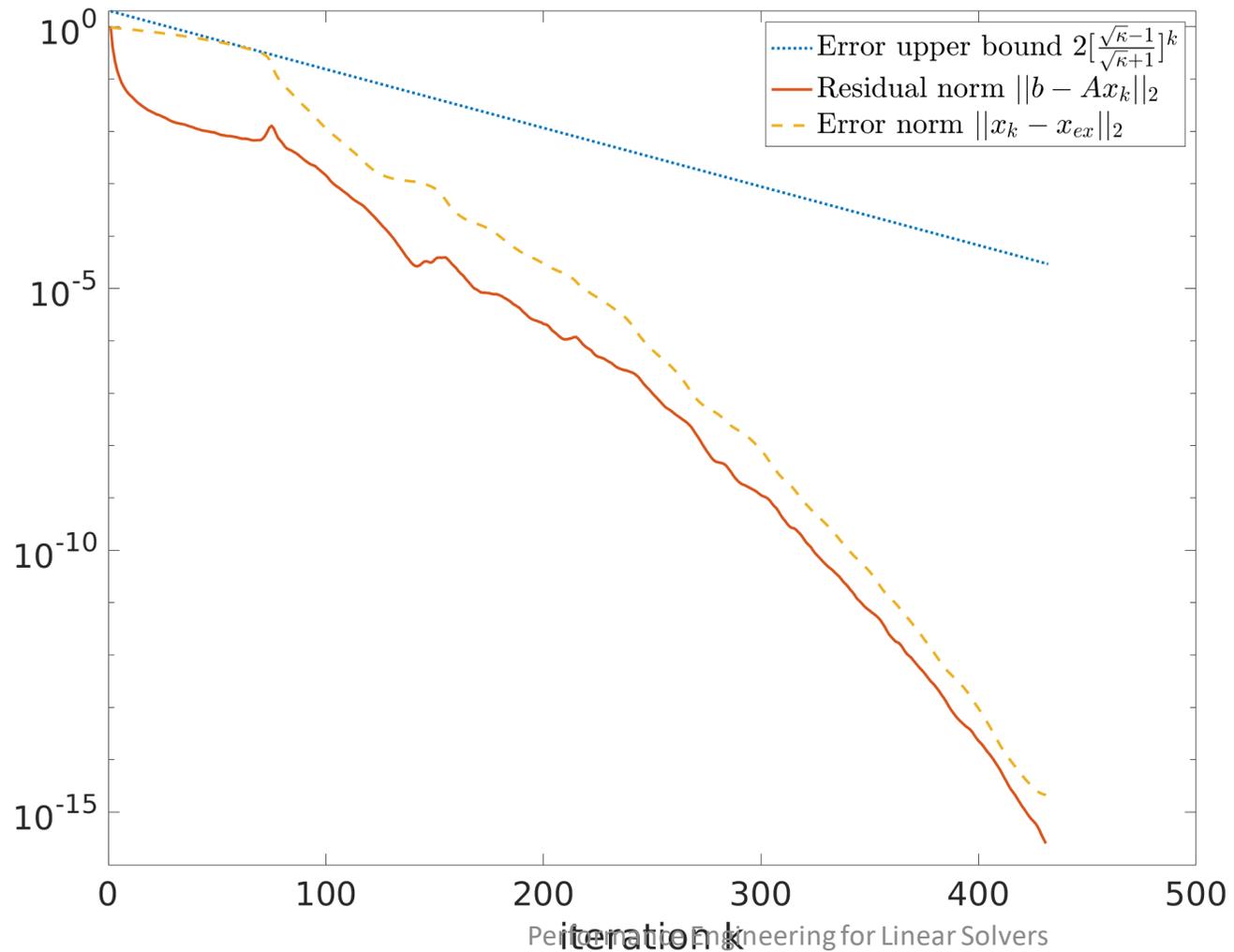
And solve the preconditioned problem

$$M_k^{-1} v := v + Lv + \dots L^k v \approx (I - L)^{-1} v$$
$$M_k^{-1} A_1 M_k^{-T} y_1 = M_k^{-1} \sqrt{D^{-1}} b$$
$$x_1 = M_k^T y_1,$$
$$x = \sqrt{D^{-1}} x_1$$

# A Simple Polynomial Preconditioner

Let

And



$$L^{-1}A\sqrt{D}^{-1}$$

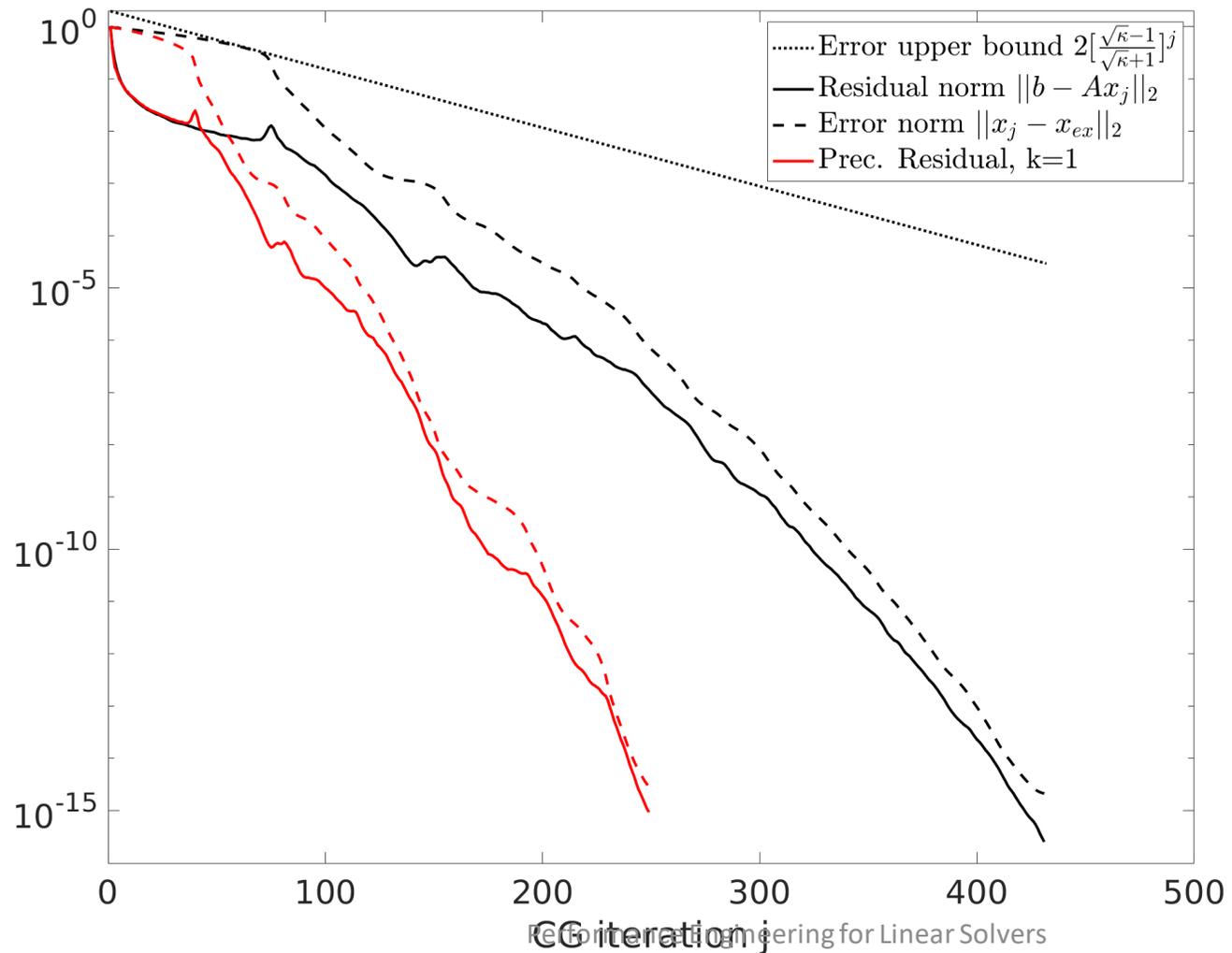
$$L - L^T$$

$$L)^{-1}v$$

# A Simple Polynomial Preconditioner

Let

And



$$\overline{-1} A \sqrt{D}^{-1}$$

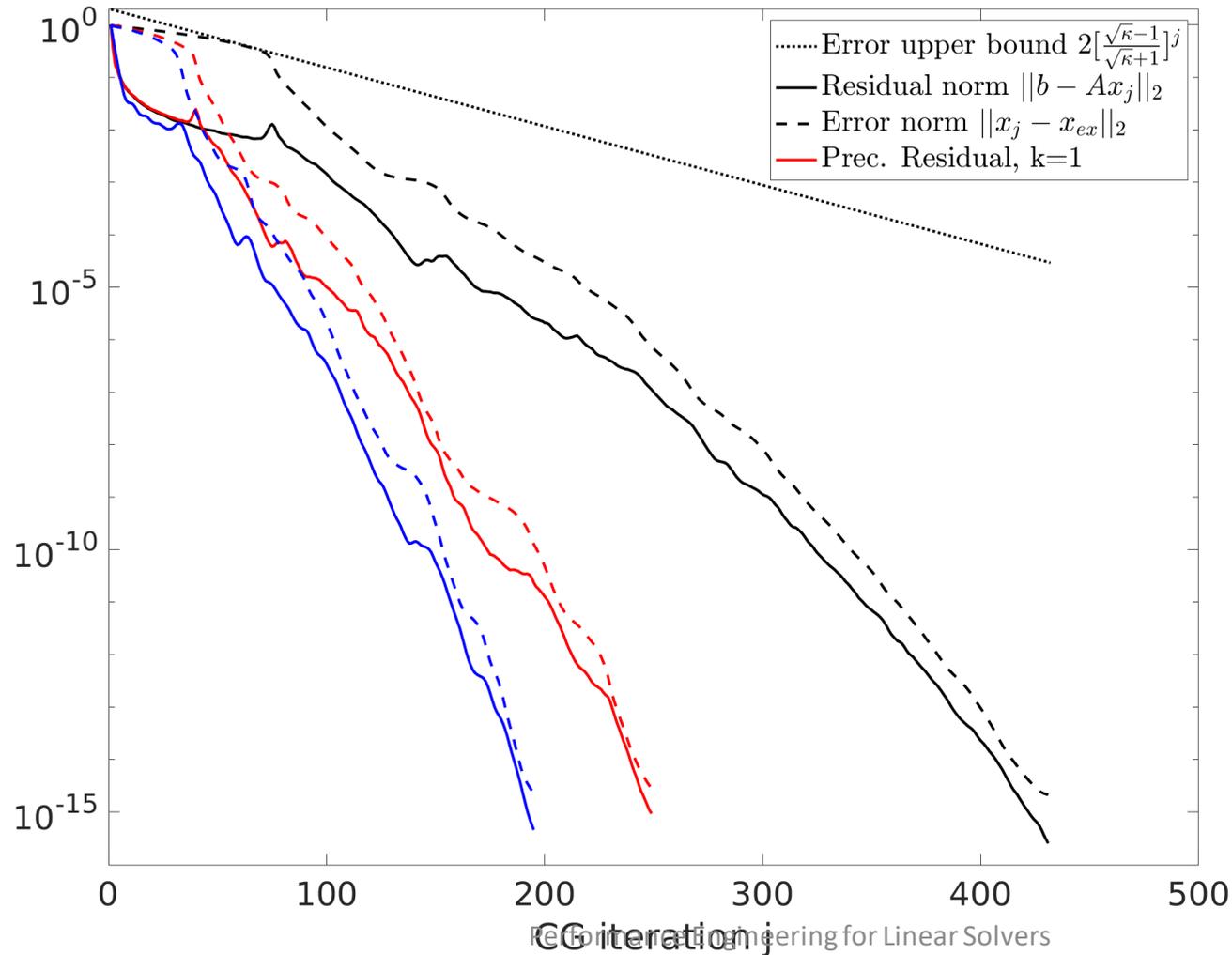
$$L - L^T$$

$$L)^{-1} v$$

# A Simple Polynomial Preconditioner

Let

And



$$\overline{L}^{-1} A \sqrt{D}^{-1}$$

$$L - L^T$$

$$L)^{-1} v$$

# Better Preconditioners

- Incomplete Cholesky:  $A \approx LL^T$
- IChol, approx. triangular solves by polynomial
- Domain Decomposition (mostly distrib. memory)
- Multigrid methods: use SpMV-based smoothers

Alappat et al: *Algebraic Temporal Blocking for Sparse Iterative Solvers on Multi-Core CPUs.*

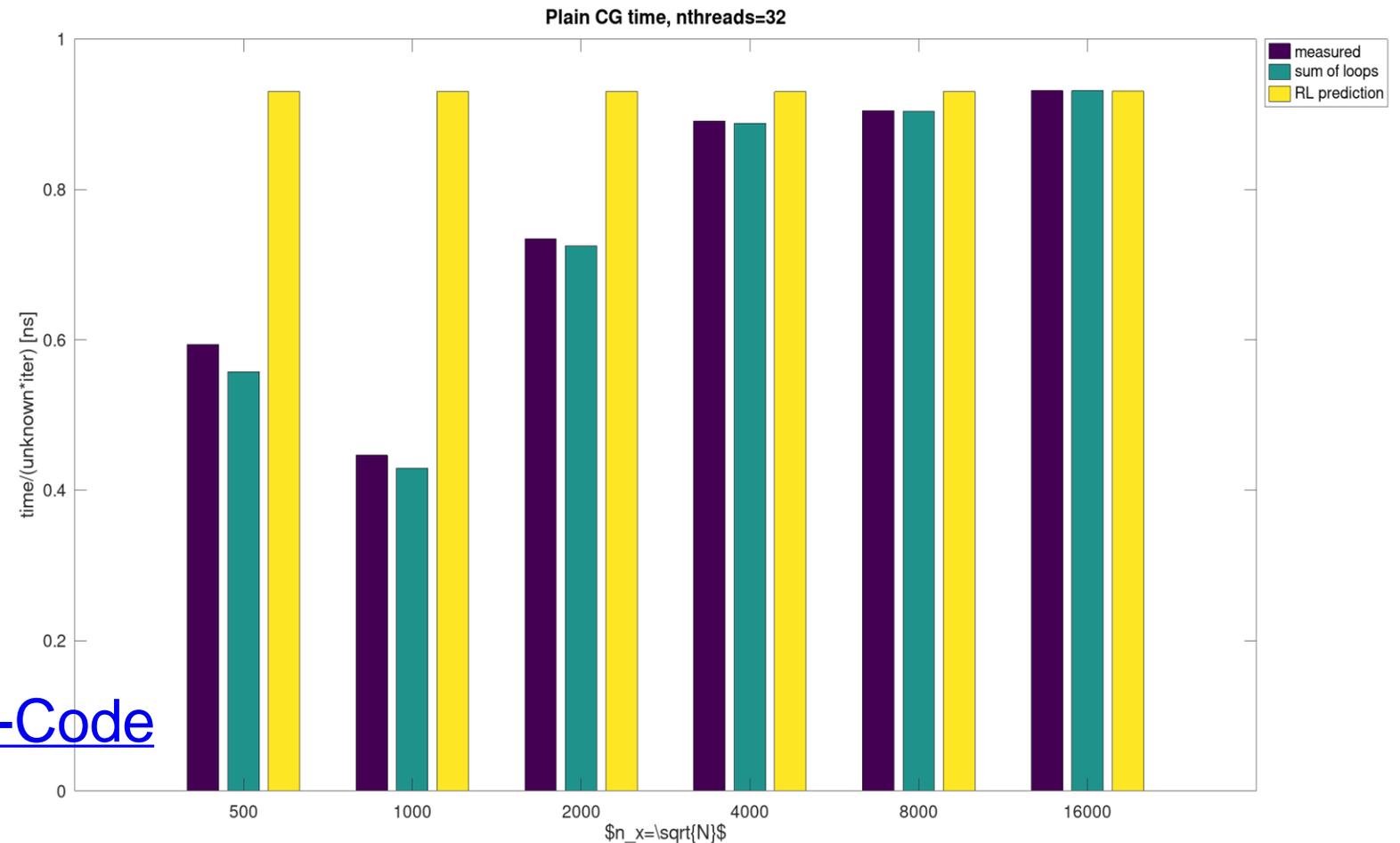
(submitted, <https://arxiv.org/abs/2309.02228>)

# Python/Numba Implementation (Live Demo)

Code available at

<https://go-nhr.de/PELS-Code>

# Python/Numba Implementation (Live Demo)



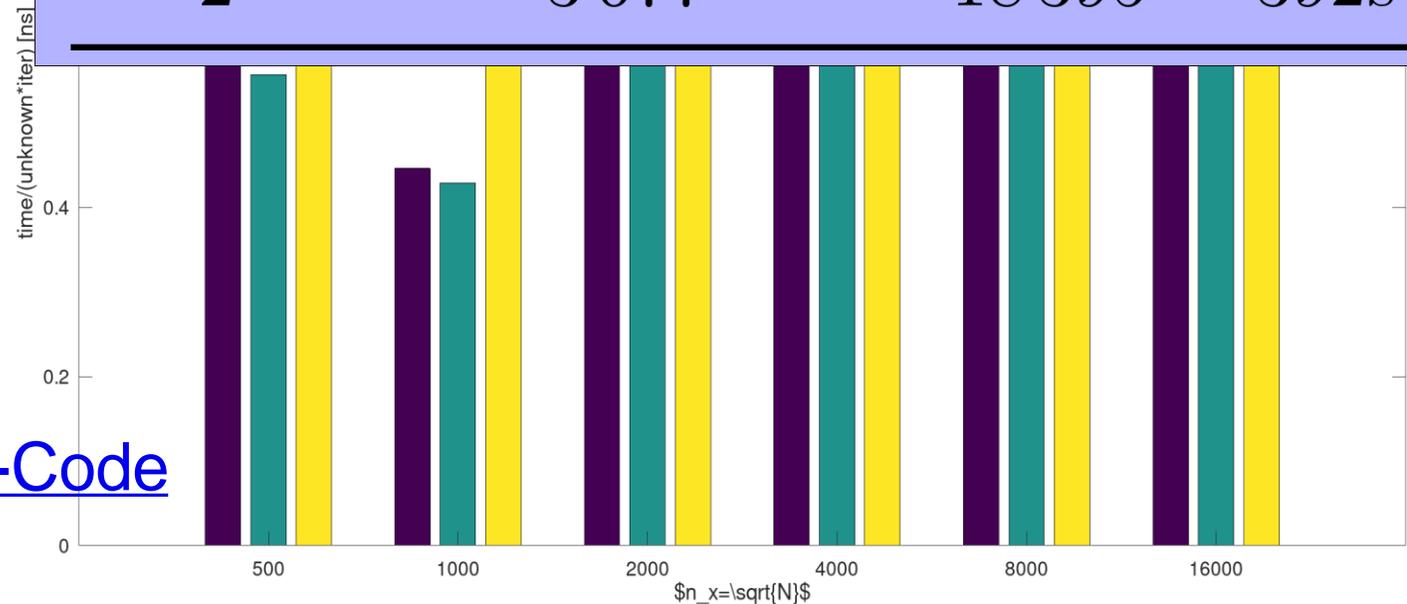
Code available at  
<https://go-nhr.de/PELS-Code>

# Python/Numba Implementation

**2D Laplace, N=25M, 32-Core Intel CPU**

poly $k$	CG iterations	SpMV's	time
0	9 194	9 196	310s
1	4 770	14 316	352s
2	3 677	18 395	392s

measured  
of loops  
prediction



Code available at  
<https://go-nhr.de/PELS-Code>

# Python/Numba Implementation

## 2D Laplace, N=25M, 32-Core Intel CPU

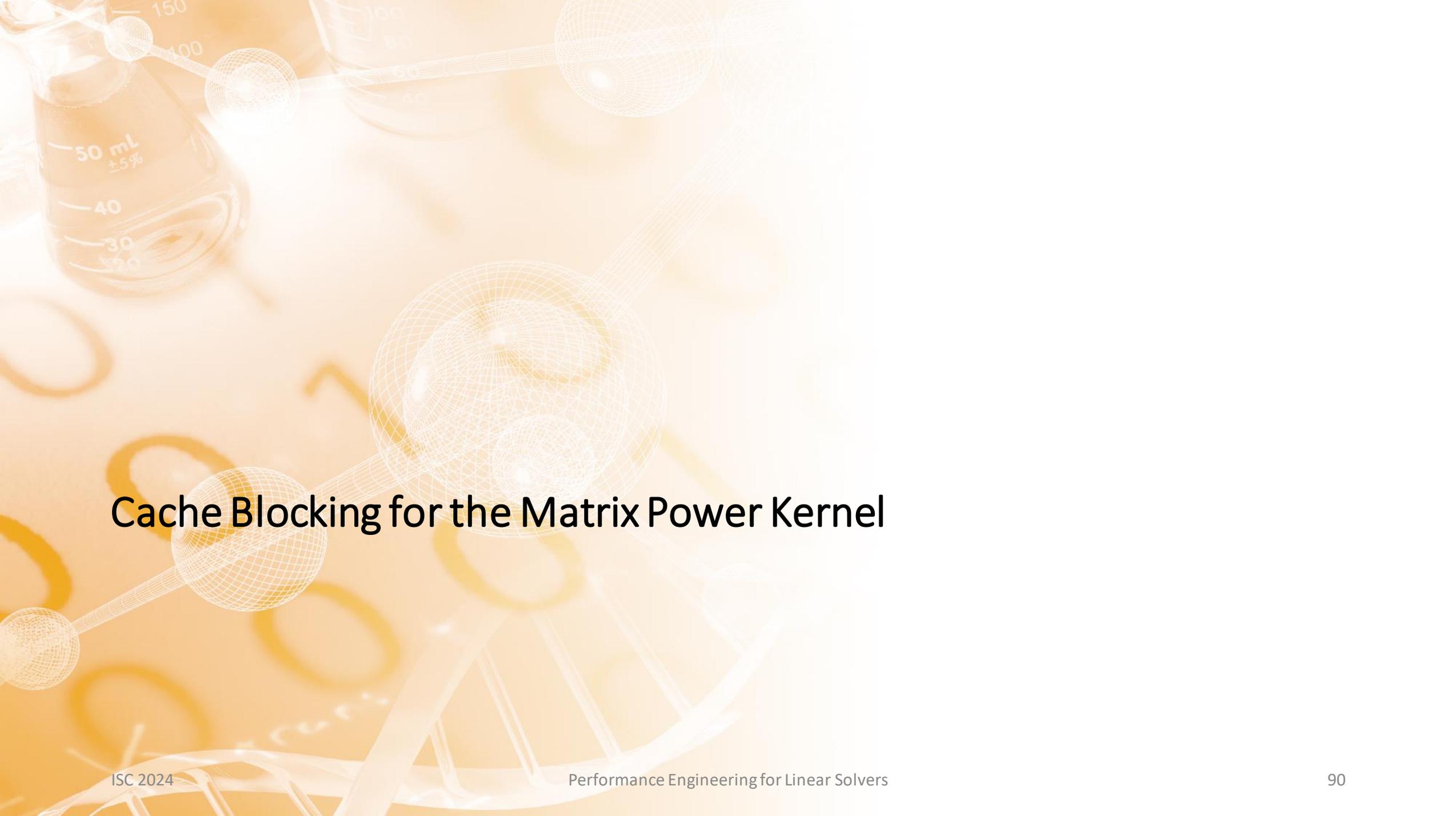
poly $k$	CG iterations	SpMV's	time
0	9 194	9 196	310s
1	4 770	14 316	352s
2	3 677	18 395	392s

## NVidia A100 GPU

poly $k$	CG iterations	SpMV's	time
0	9 148	9 150	32.6s
1	4 228	12 690	44.2s
2	3 695	18 485	48.9s

Code available at

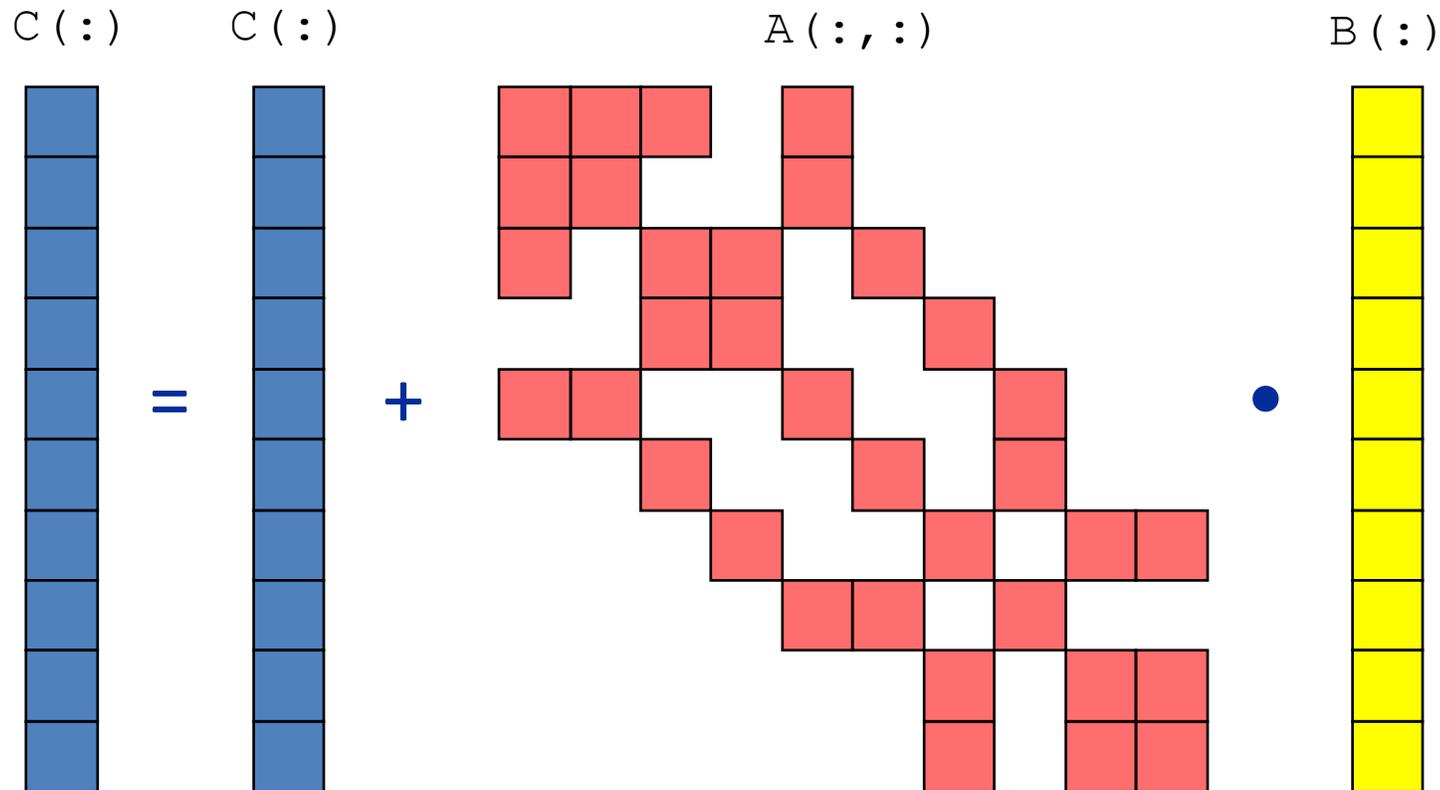
<https://go-nhr.de/PELS-C>



# Cache Blocking for the Matrix Power Kernel

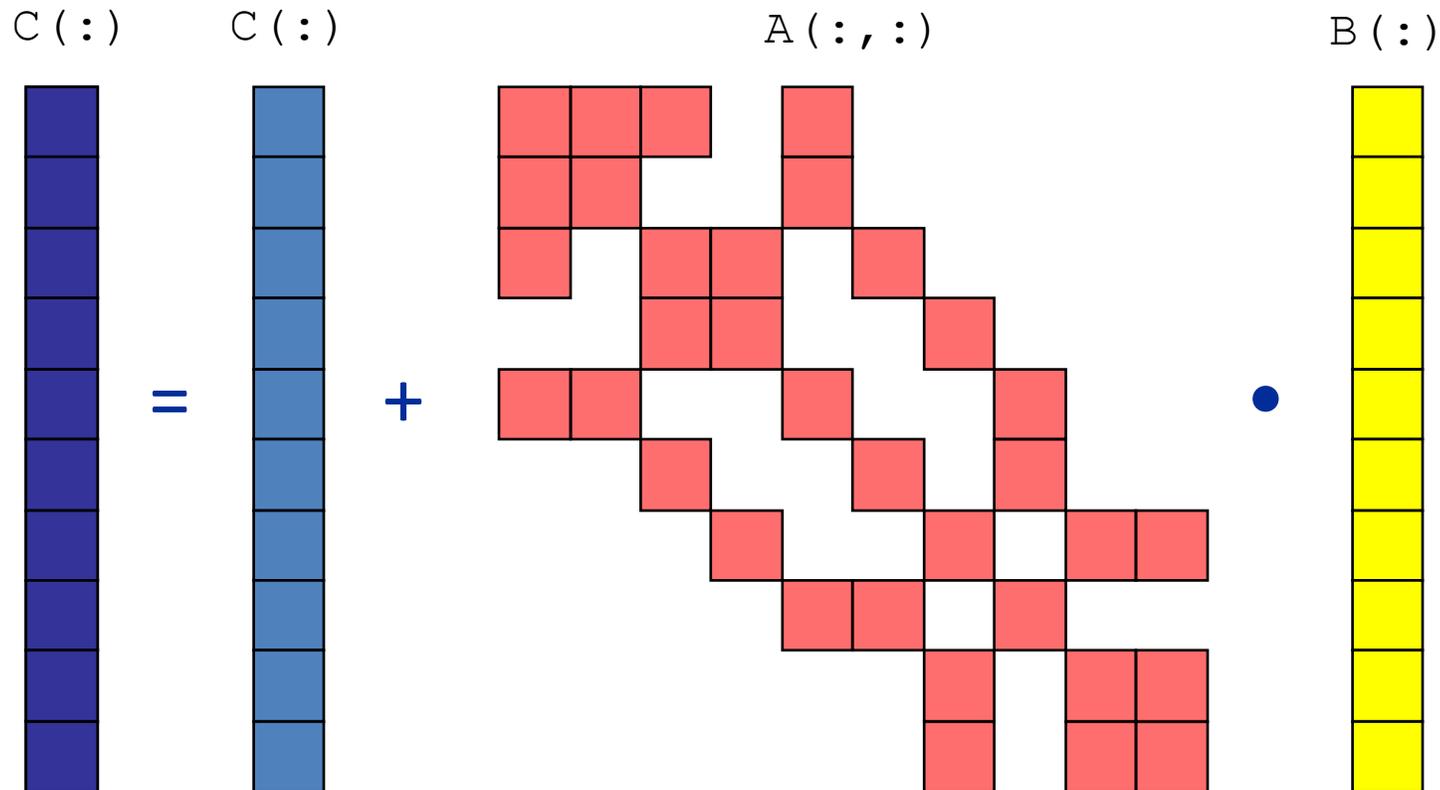
# Motivation – Sparse Matrix Vector Multiplication

- Easy to parallelize but sparse irregular data structures / accesses
- SpMV Performance  $\leftrightarrow$  Strongly Memory Bound (high code balance)



# Motivation – Sparse Matrix Vector Multiplication

- Easy to parallelize but sparse irregular data structures / accesses
- SpMV Performance  $\leftrightarrow$  Strongly Memory Bound (high code balance)



## RACE & Cache Blocking for MPK

C. L. Alappat, G. Hager, O. Schenk, and G. Wellein: *Level-based Blocking for Sparse Matrices: Sparse Matrix-Power-Vector Multiplication*. IEEE Transactions on Parallel and Distributed Systems 34(2), 581-597 (2023), DOI: [10.1109/TPDS.2022.3223512](https://doi.org/10.1109/TPDS.2022.3223512)

# Motivation – Matrix power kernel (MPK)

- Calculate:  $y = A^p x$
- Repeatedly perform back to back SpMVs

```
for k=1:p; do  
  y[k] = SpMV(A, y[k-1])  
done
```

# Motivation – Matrix power kernel (MPK)

- Calculate:  $y = A^p x$
- Repeatedly perform back to back SpMV

```
for k=1:p; do
  y[k] = SpMV(A, y[k-1])
done
```

$y[0]$



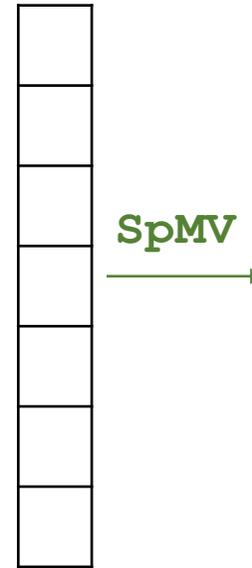
$x$

# Motivation – Matrix power kernel (MPK)

- Calculate:  $y = A^p x$
- Repeatedly perform back to back SpMVs

```
for k=1:p; do
  y[k] = SpMV(A, y[k-1])
done
```

$y[0]$

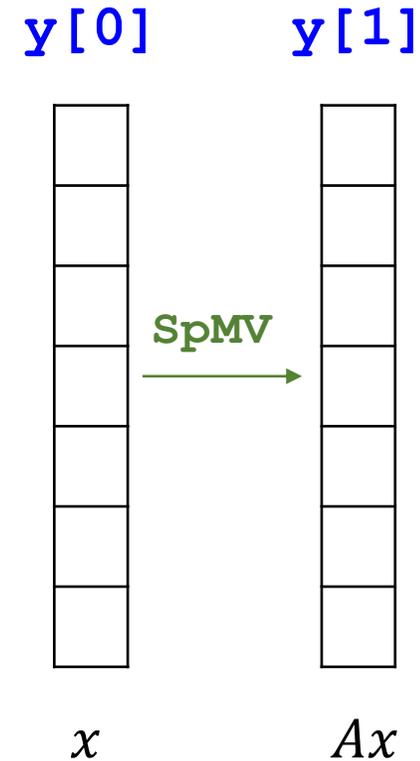


$x$

# Motivation – Matrix power kernel (MPK)

- Calculate:  $y = A^p x$
- Repeatedly perform back to back SpMVs

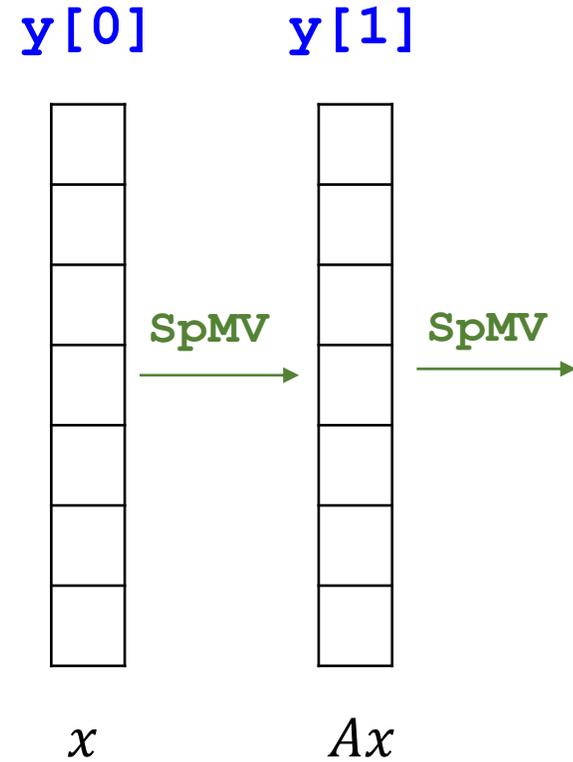
```
for k=1:p; do
  y[k] = SpMV(A, y[k-1])
done
```



# Motivation – Matrix power kernel (MPK)

- Calculate:  $y = A^p x$
- Repeatedly perform back to back SpMVs

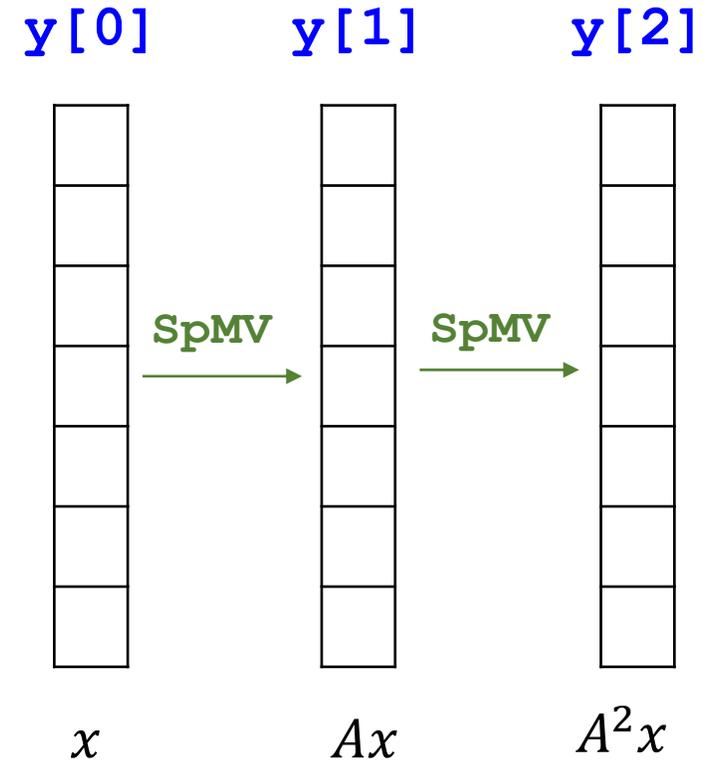
```
for k=1:p; do
  y[k] = SpMV(A, y[k-1])
done
```



# Motivation – Matrix power kernel (MPK)

- Calculate:  $y = A^p x$
- Repeatedly perform back to back SpMV

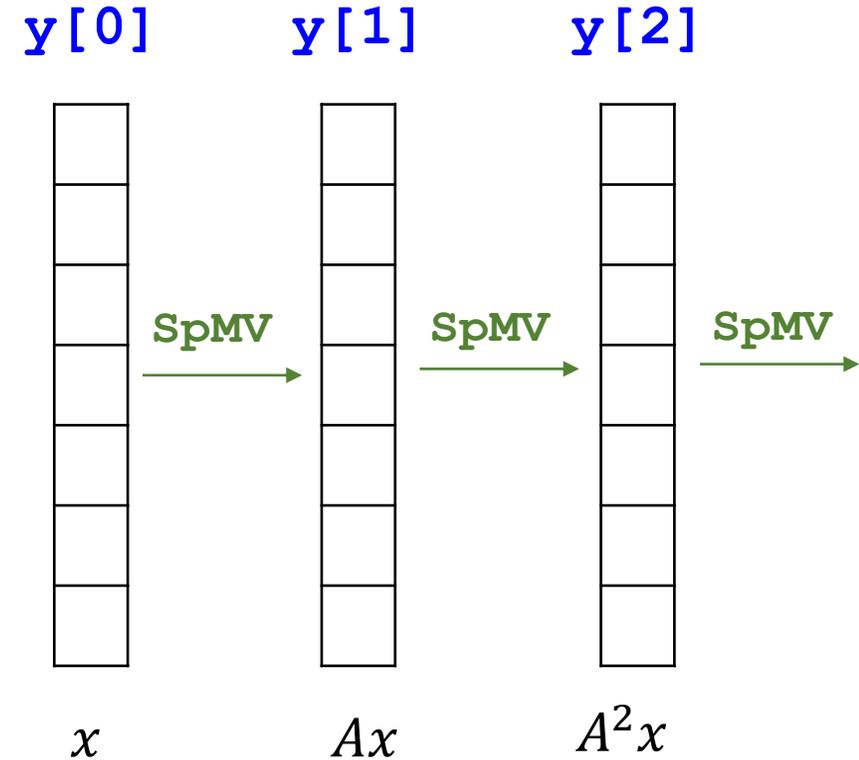
```
for k=1:p; do
  y[k] = SpMV(A, y[k-1])
done
```



# Motivation – Matrix power kernel (MPK)

- Calculate:  $y = A^p x$
- Repeatedly perform back to back SpMVs

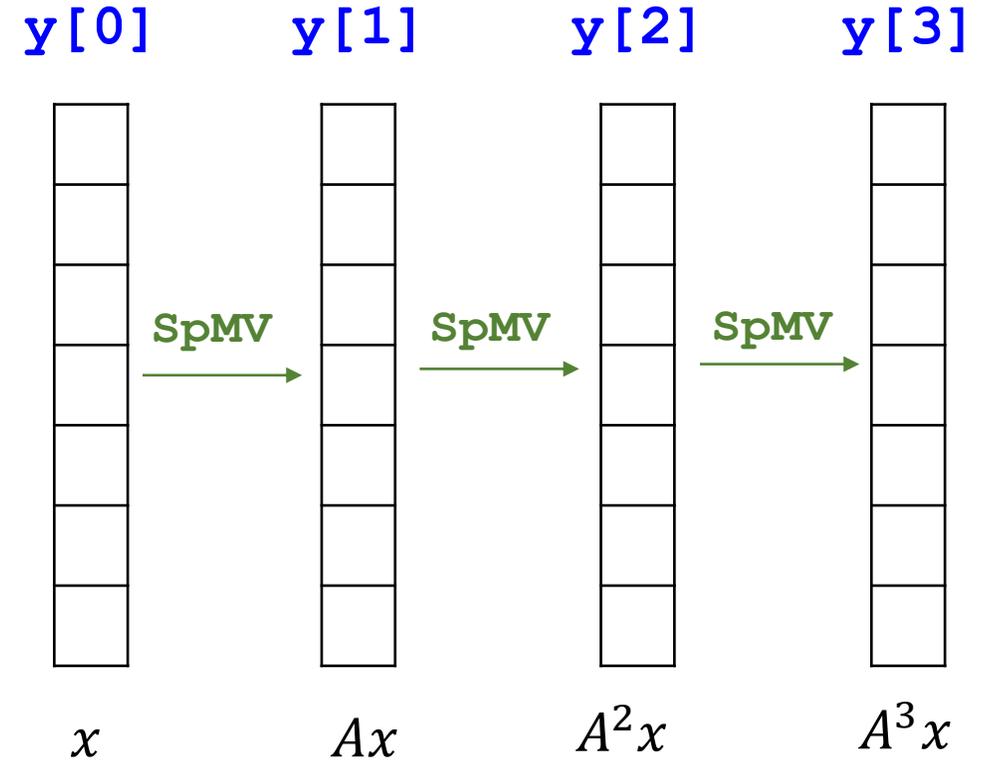
```
for k=1:p; do
  y[k] = SpMV(A, y[k-1])
done
```



# Motivation – Matrix power kernel (MPK)

- Calculate:  $y = A^p x$
- Repeatedly perform back to back SpMV

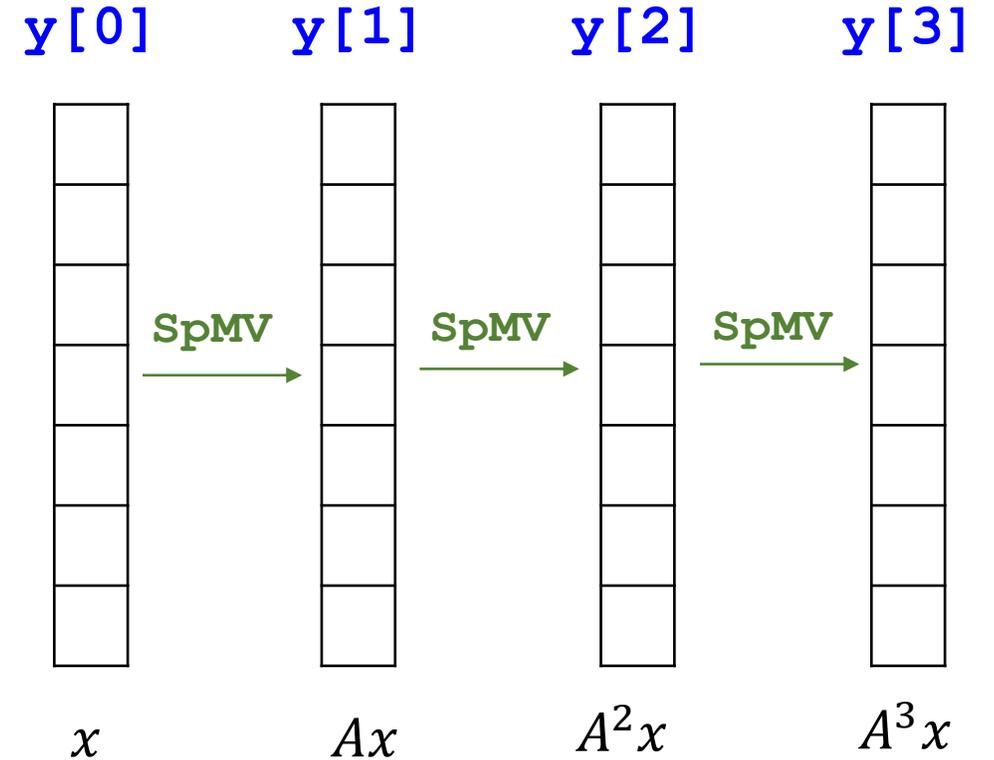
```
for k=1:p; do
  y[k] = SpMV(A, y[k-1])
done
```



# Motivation – Matrix power kernel (MPK)

- Calculate:  $y = A^p x$
- Repeatedly perform back to back SpMV

```
for k=1:p; do
  y[k] = SpMV(A, y[k-1])
done
```

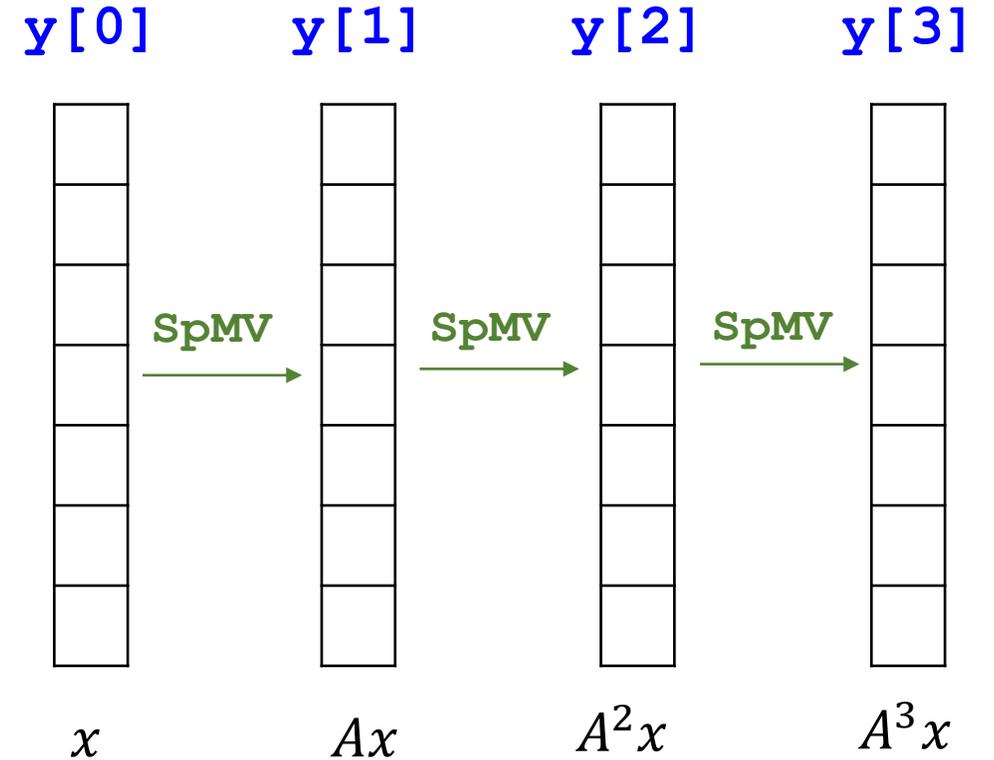


Same matrix  $A$  loaded  $p$  times from main memory!!!

# Motivation – Matrix power kernel (MPK)

- Calculate:  $y = A^p x$
- Repeatedly perform back to back SpMV

```
for k=1:p; do
  y[k] = SpMV(A, y[k-1])
done
```



Same matrix  $A$  loaded  $p$  times from main memory!!!

How to cache the matrix  $A$  across the matrix power calculation?

# MPK – existing caching approaches

- Huber et al.: Graph-based higher-order time integration of PDEs<sup>1</sup>
  - “Geometrical approach” based on matrix bandwidth
  - Works for 2D stencil matrices → Runs into problem for 3D and/or unstructured matrices
- Mohiyuddin et al.: Minimizing communication in sparse matrix solvers<sup>2</sup>
  - “Domain decomposition” of underlying graph
  - Requires “ghosting” → Indirect accesses or redundant copies of the matrix entries → Scalability!!

→ Exploit level structure in RACE for cache blocking!

The logo for RACE, with the letters R, A, C, and E in red, blue, green, and yellow respectively.

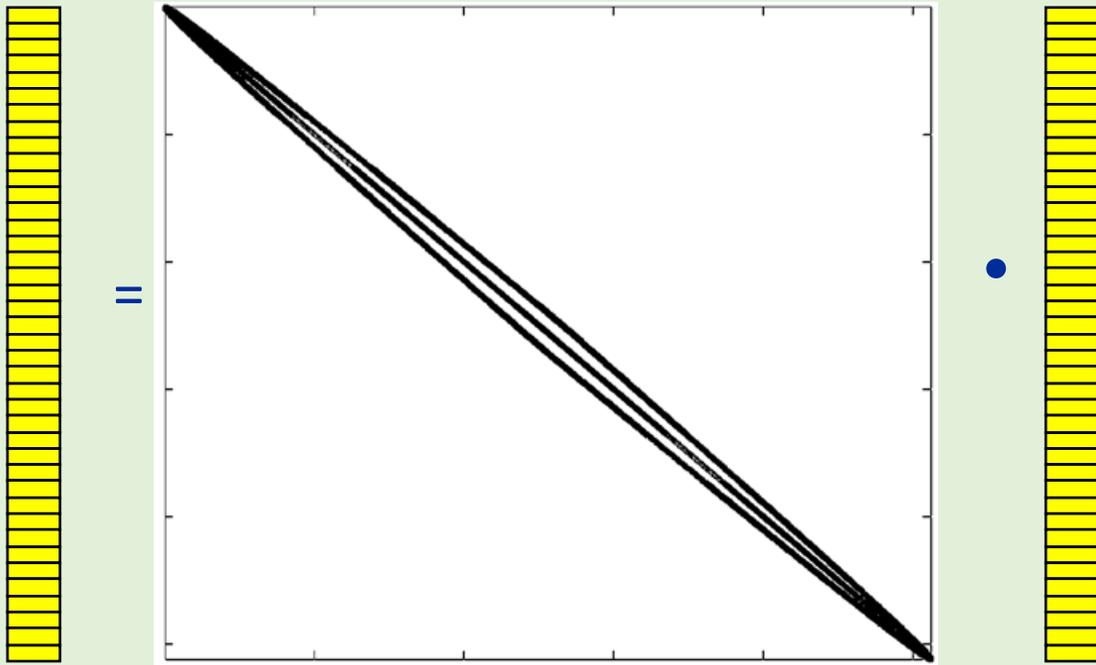
<sup>1</sup>Huber et al., 2021. Graph-based multi-core higher-order time integration of linear autonomous partial differential equations. J. Comput. Sci. [DOI:10.1016/j.jocs.2021.101349](https://doi.org/10.1016/j.jocs.2021.101349)

<sup>2</sup>Mohiyuddin et al., 2009. Minimizing communication in sparse matrix solvers. In Proceedings of the SC'09. [DOI:10.1145/1654059.1654096](https://doi.org/10.1145/1654059.1654096)

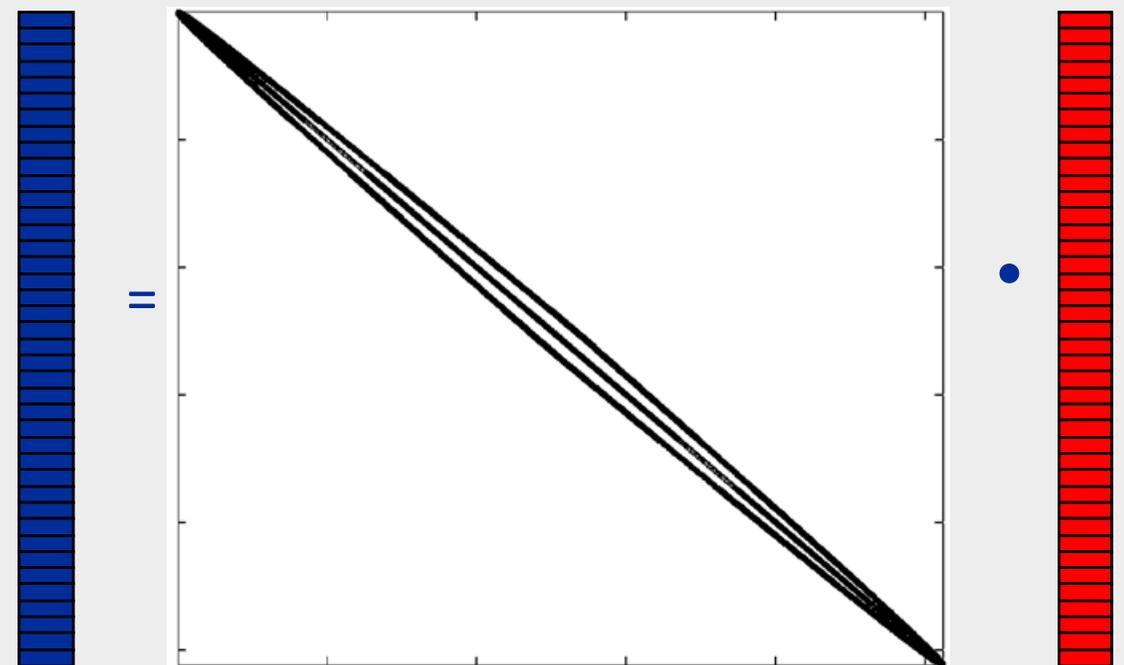
# Matrix power – Traditional approach vs. Cache Blocking

Calculate  $y = A^3 x$

**TRAD approach**



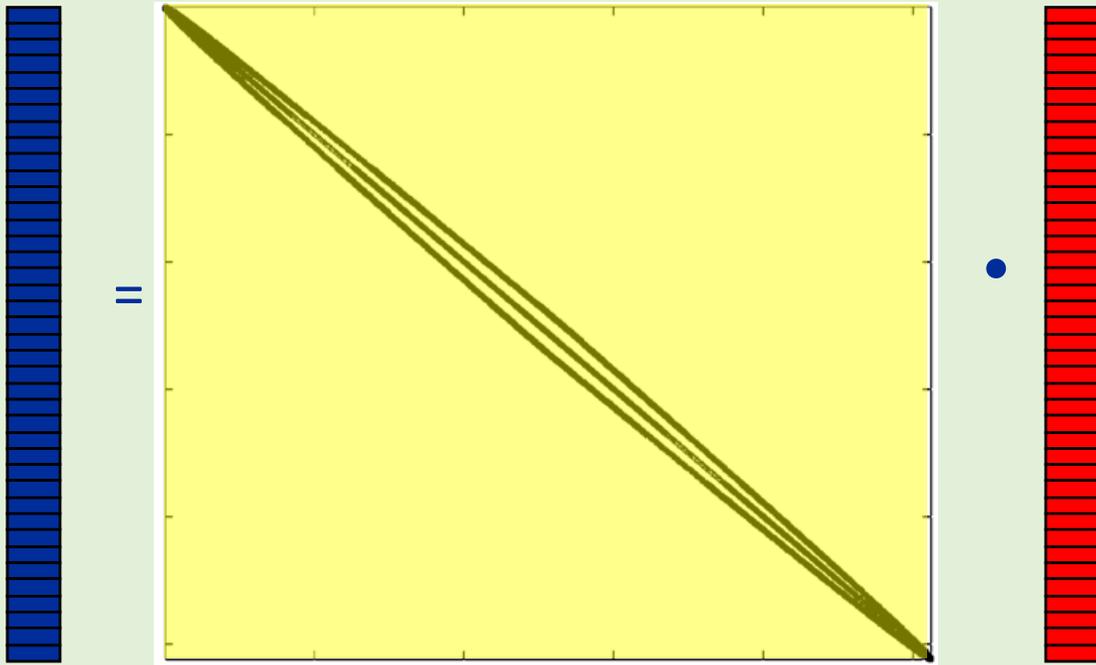
**RACE approach**



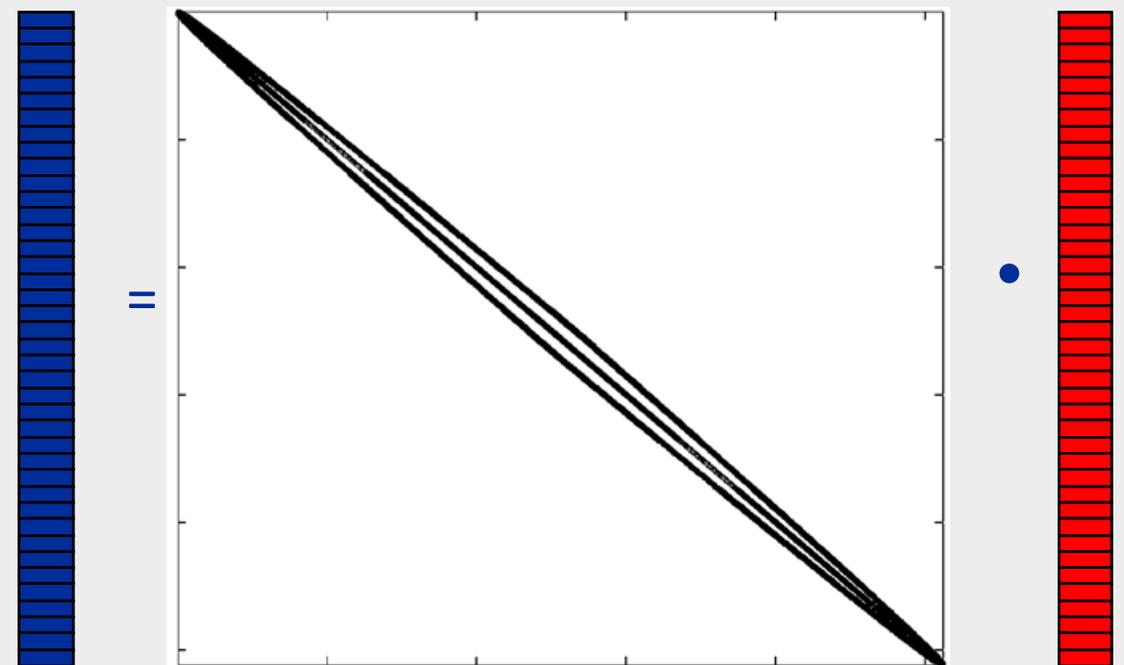
# Matrix power – Traditional approach vs. Cache Blocking

Calculate  $y = A^3 x$

**TRAD approach**



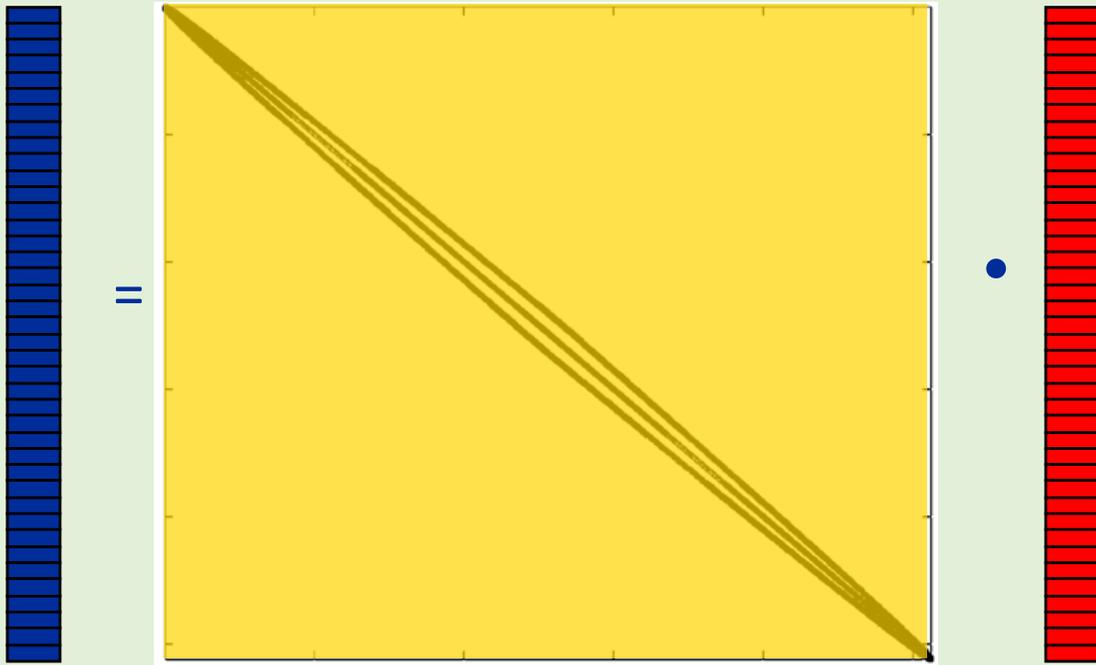
**RACE approach**



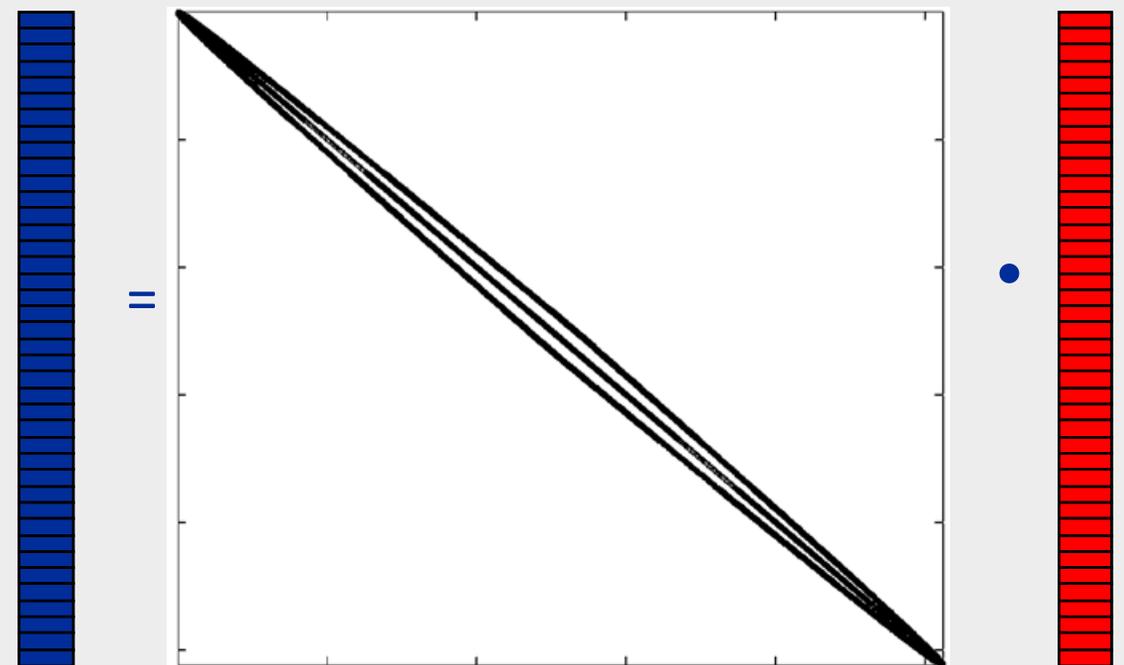
# Matrix power – Traditional approach vs. Cache Blocking

Calculate  $y = A^3 x$

**TRAD approach**



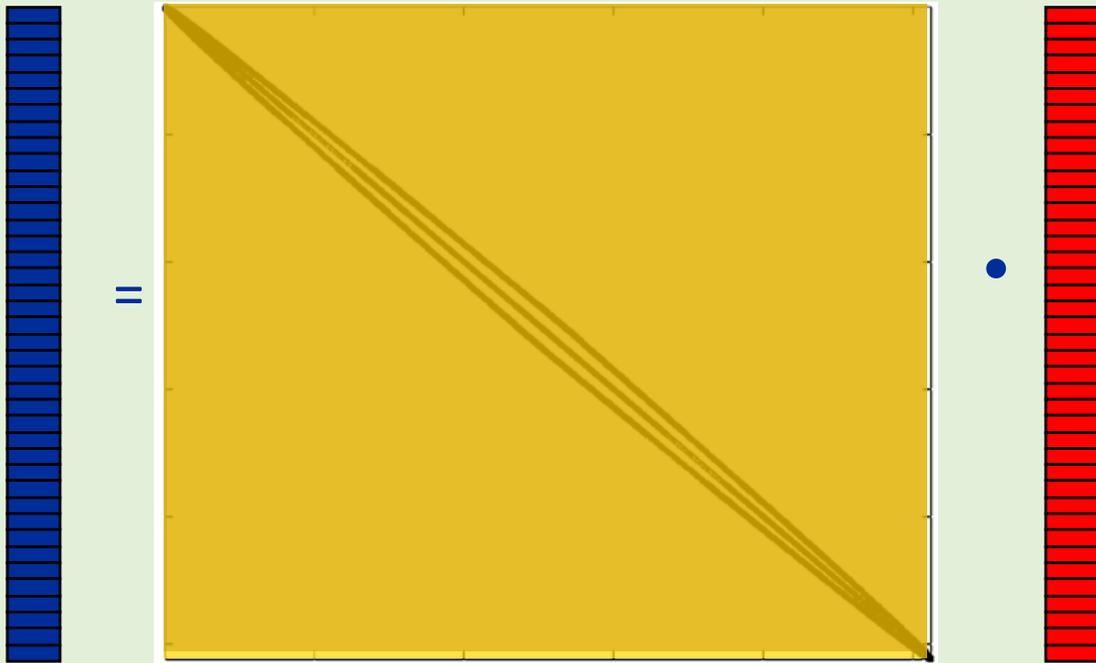
**RACE approach**



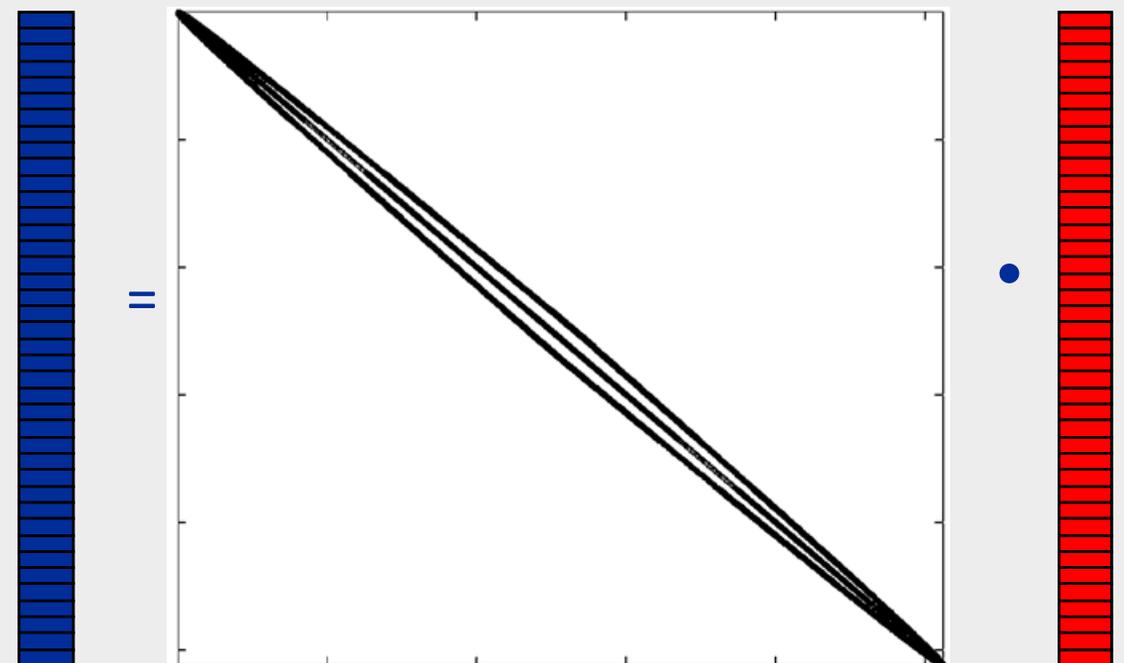
# Matrix power – Traditional approach vs. Cache Blocking

Calculate  $y = A^3 x$

**TRAD approach**



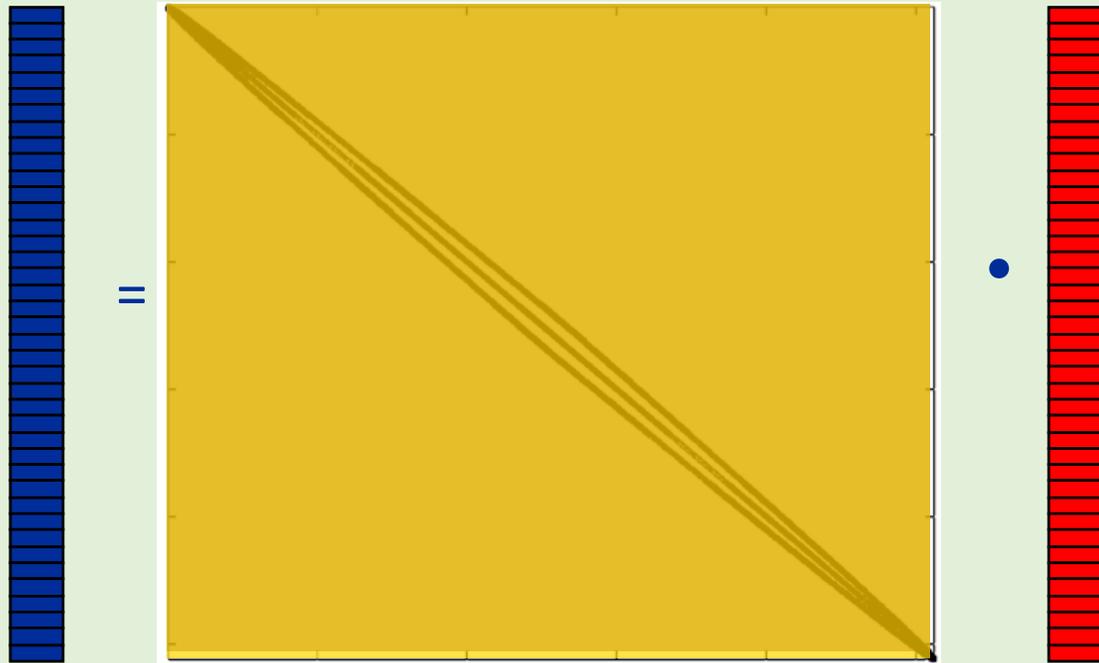
**RACE approach**



# Matrix power – Traditional approach vs. Cache Blocking

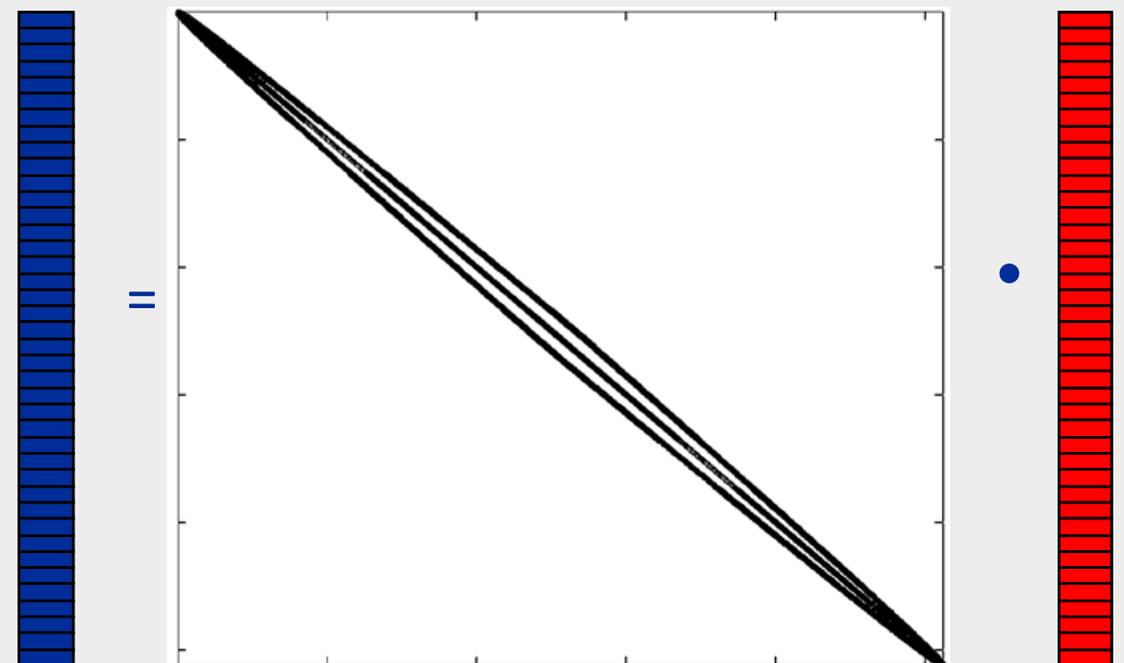
$$\text{Calculate } y = A^3 x$$

**TRAD approach**



Matrix accessed 3 times from memory

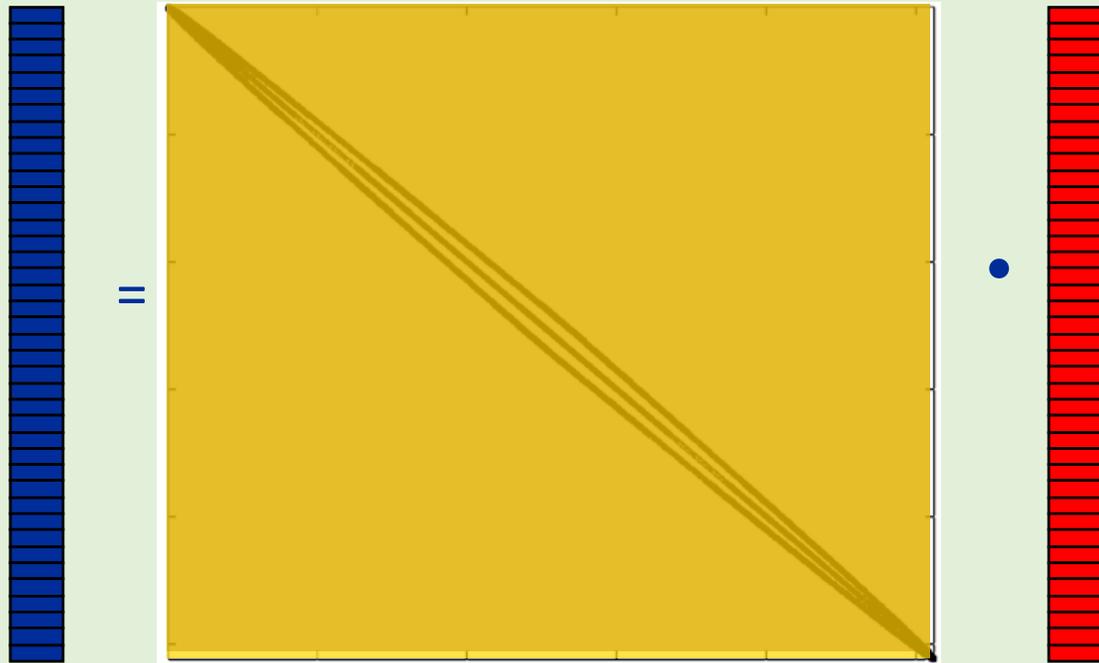
**RACE approach**



# Matrix power – Traditional approach vs. Cache Blocking

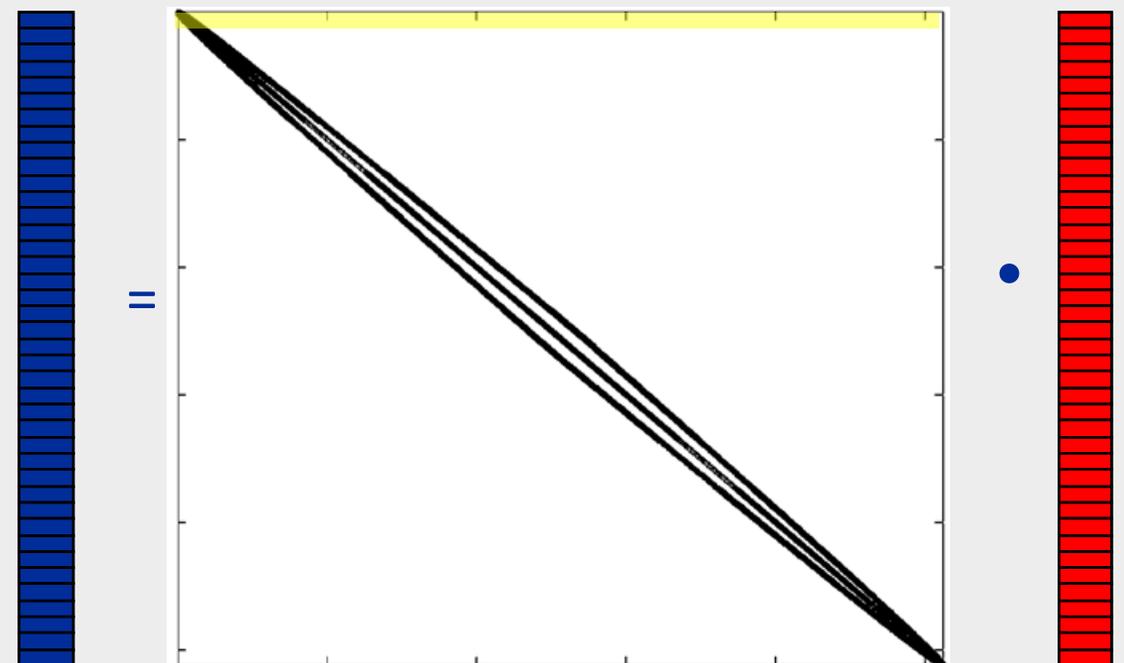
$$\text{Calculate } y = A^3 x$$

**TRAD approach**



Matrix accessed 3 times from memory

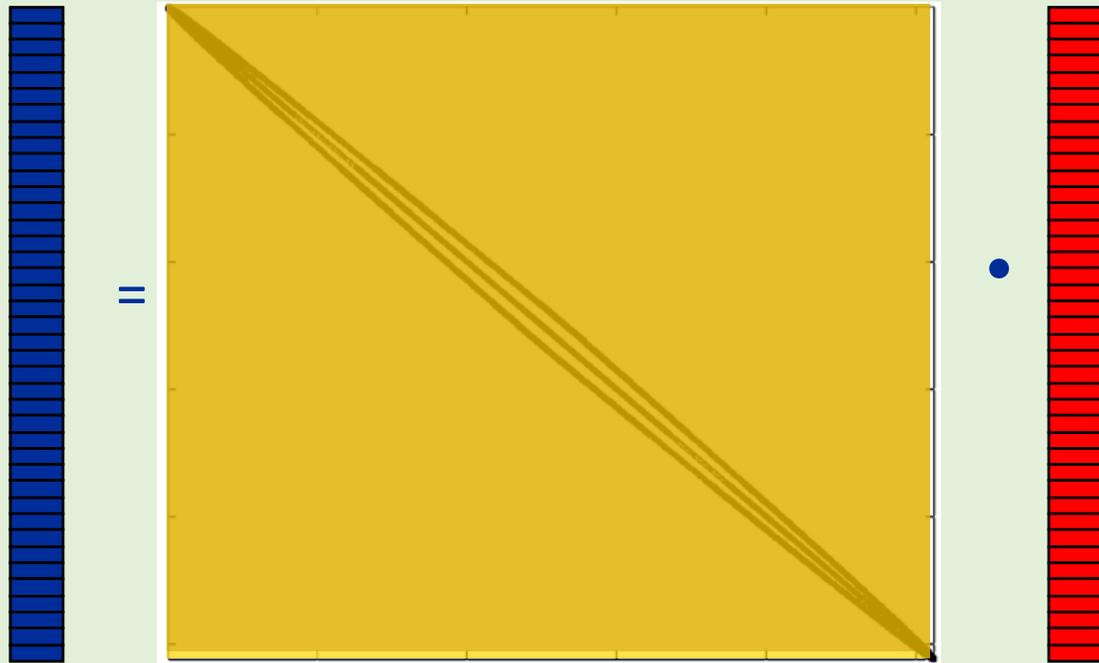
**RACE approach**



# Matrix power – Traditional approach vs. Cache Blocking

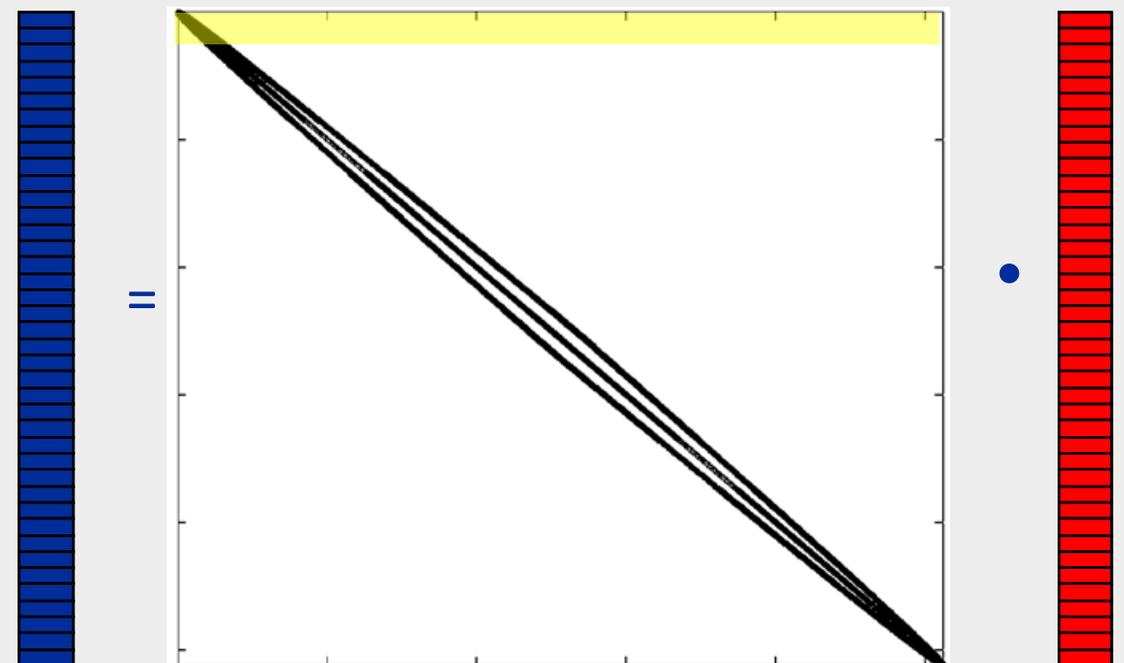
$$\text{Calculate } y = A^3 x$$

## TRAD approach



Matrix accessed 3 times from memory

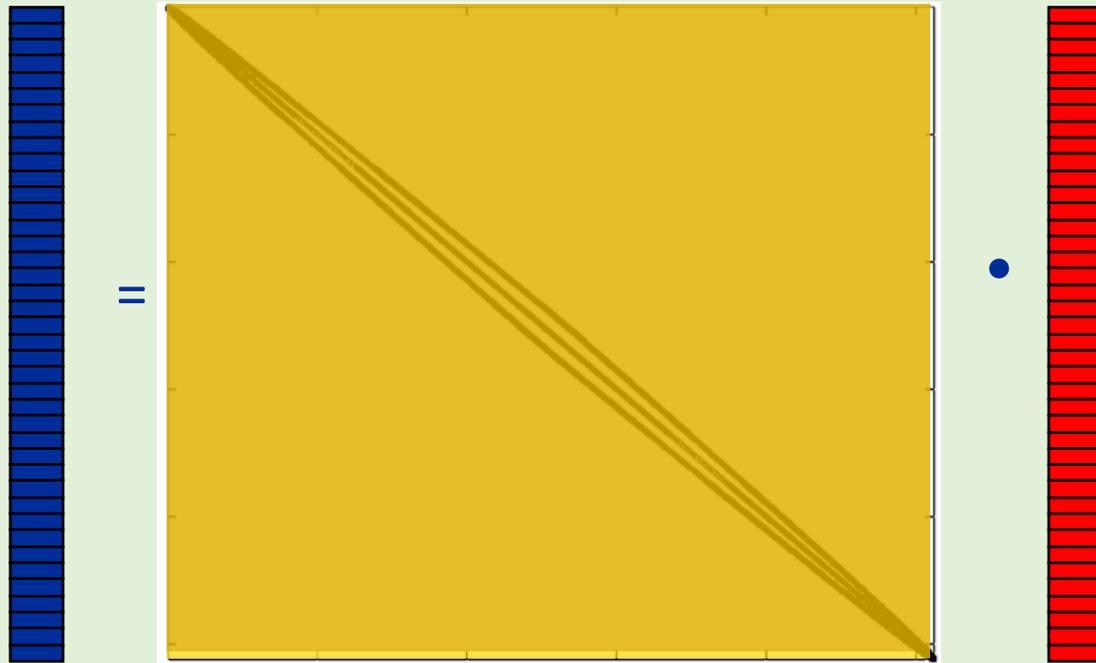
## RACE approach



# Matrix power – Traditional approach vs. Cache Blocking

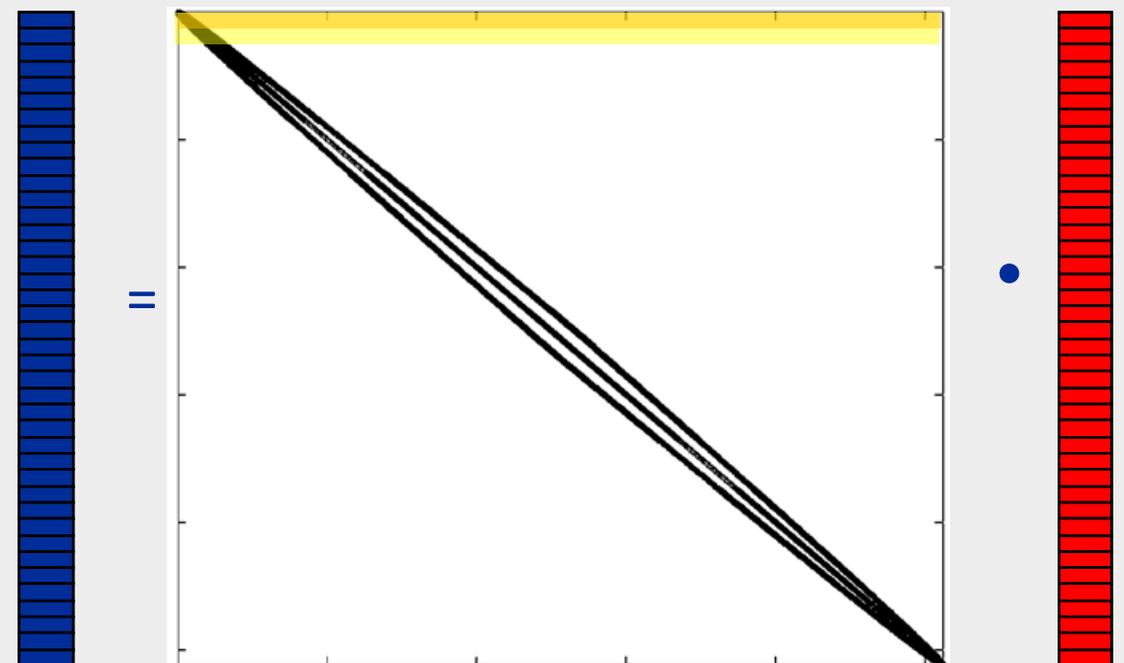
$$\text{Calculate } y = A^3 x$$

**TRAD approach**



Matrix accessed 3 times from memory

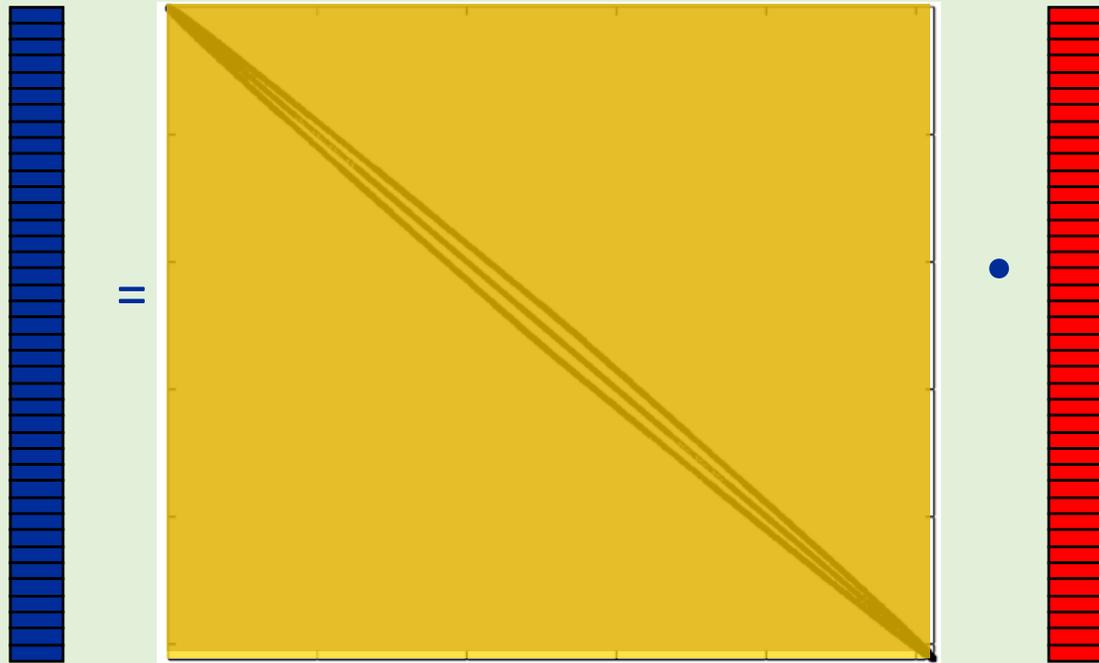
**RACE approach**



# Matrix power – Traditional approach vs. Cache Blocking

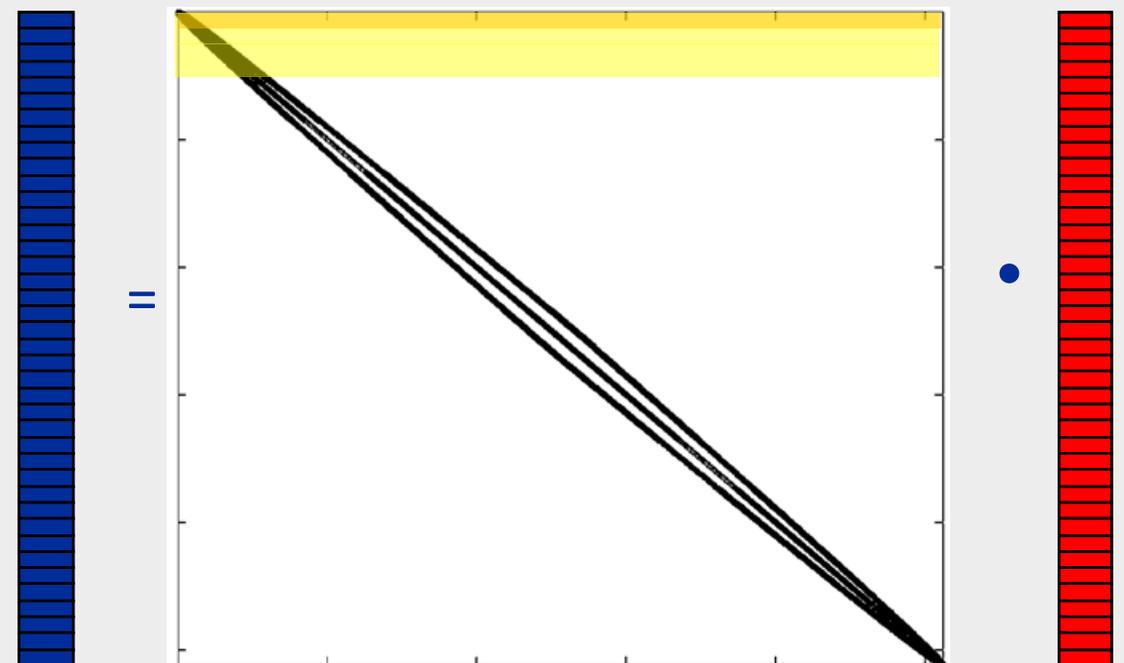
$$\text{Calculate } y = A^3 x$$

## TRAD approach



Matrix accessed 3 times from memory

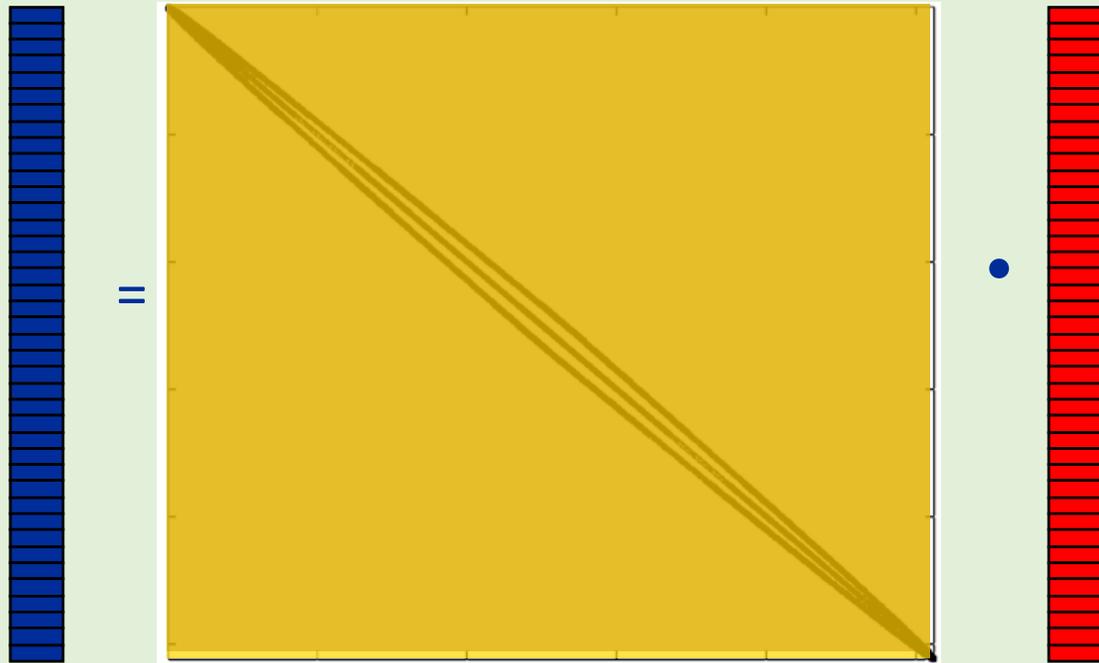
## RACE approach



# Matrix power – Traditional approach vs. Cache Blocking

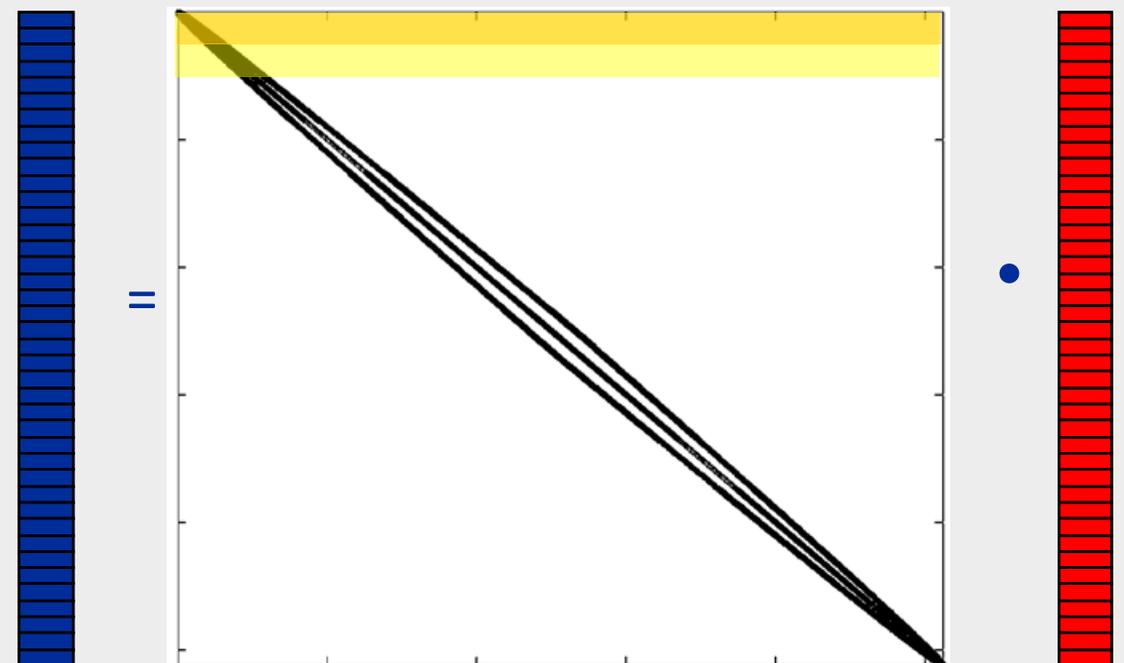
$$\text{Calculate } y = A^3 x$$

## TRAD approach



Matrix accessed 3 times from memory

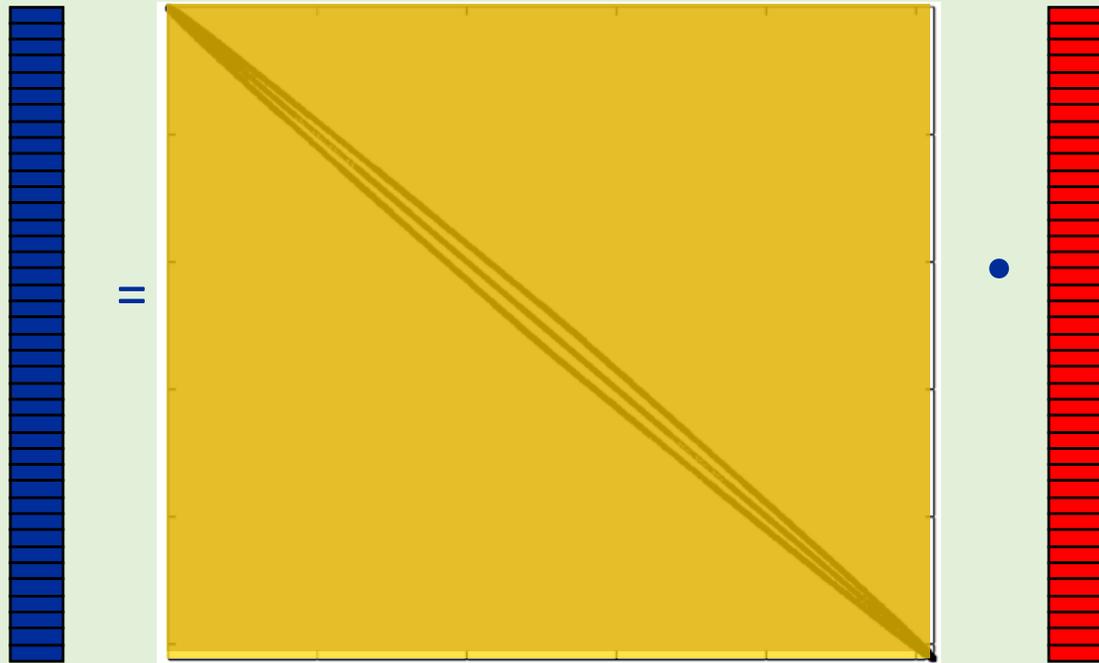
## RACE approach



# Matrix power – Traditional approach vs. Cache Blocking

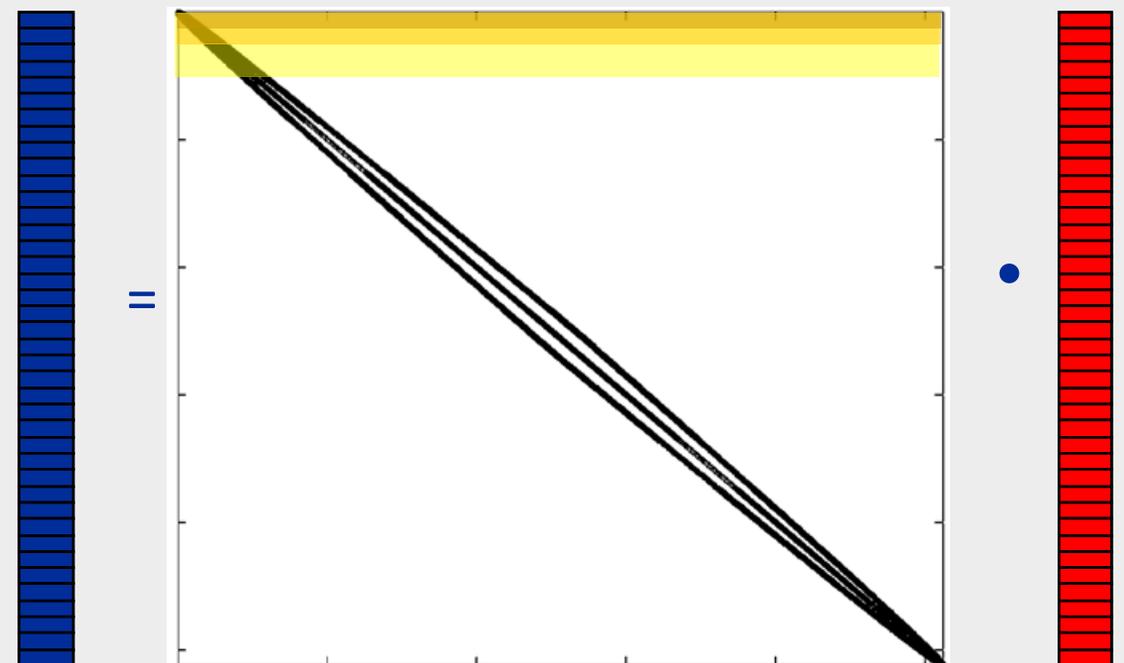
$$\text{Calculate } y = A^3 x$$

**TRAD approach**



Matrix accessed 3 times from memory

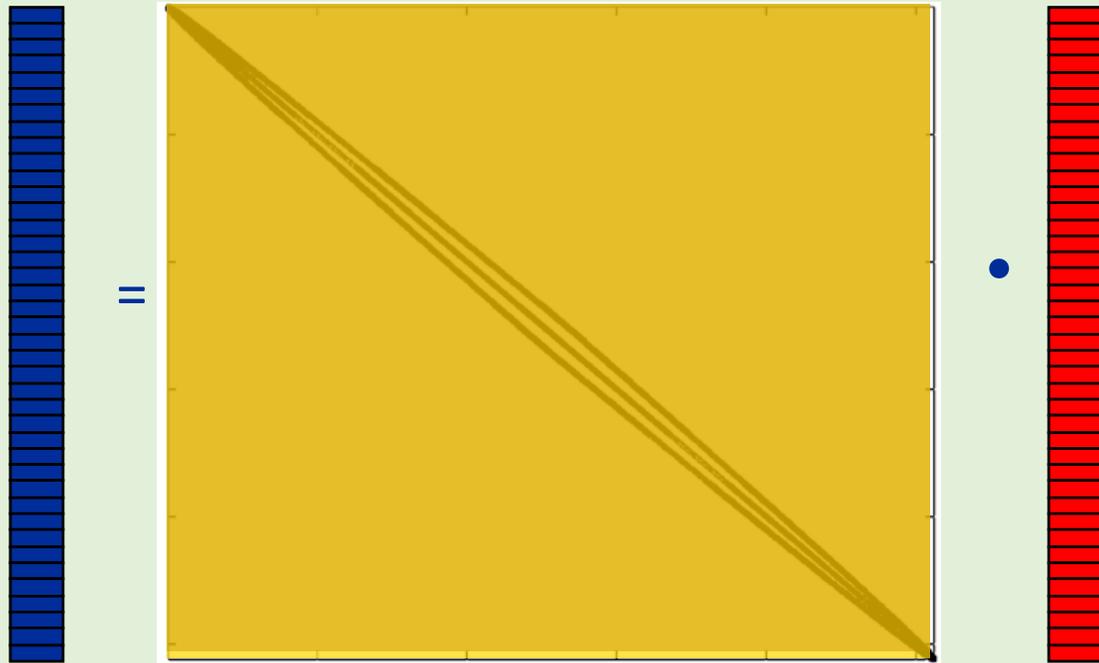
**RACE approach**



# Matrix power – Traditional approach vs. Cache Blocking

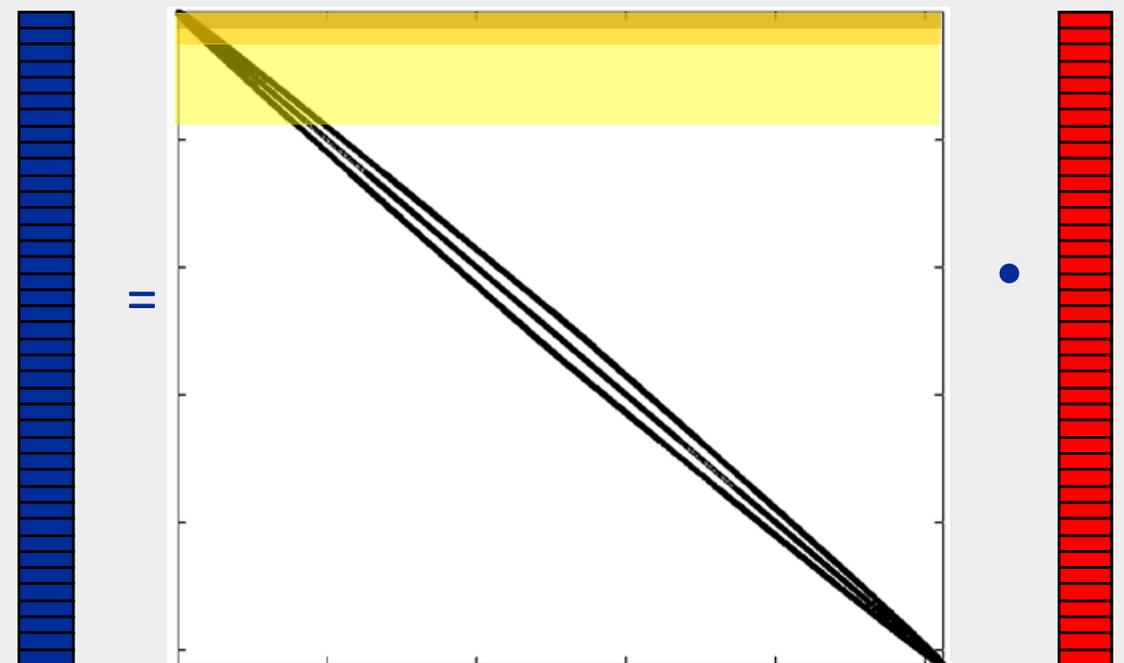
$$\text{Calculate } y = A^3 x$$

**TRAD approach**



Matrix accessed 3 times from memory

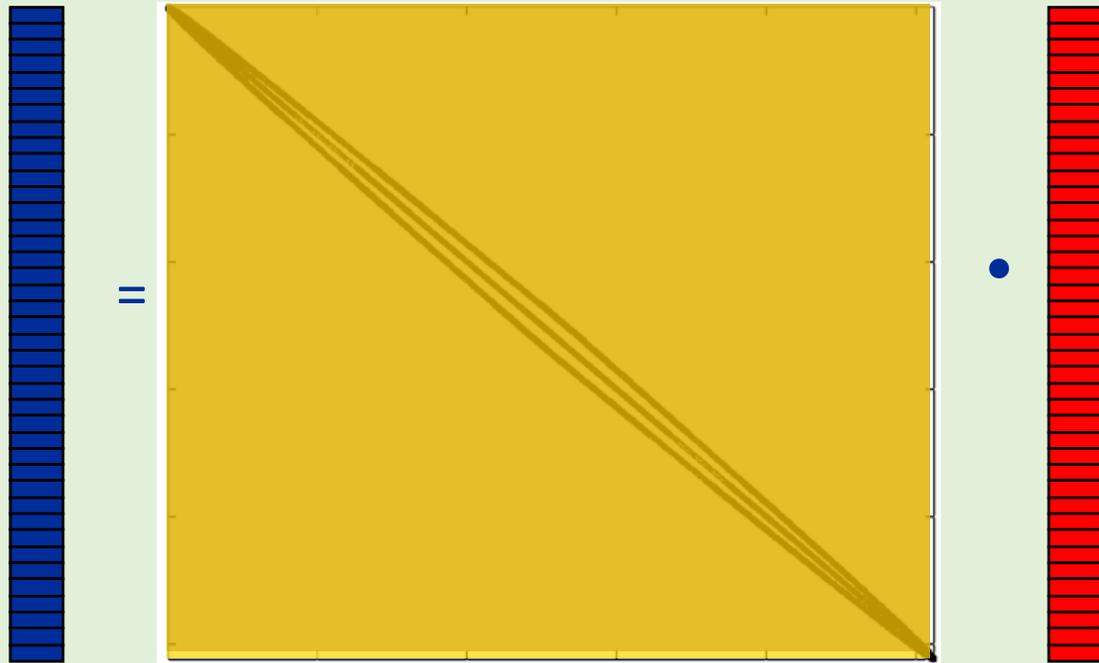
**RACE approach**



# Matrix power – Traditional approach vs. Cache Blocking

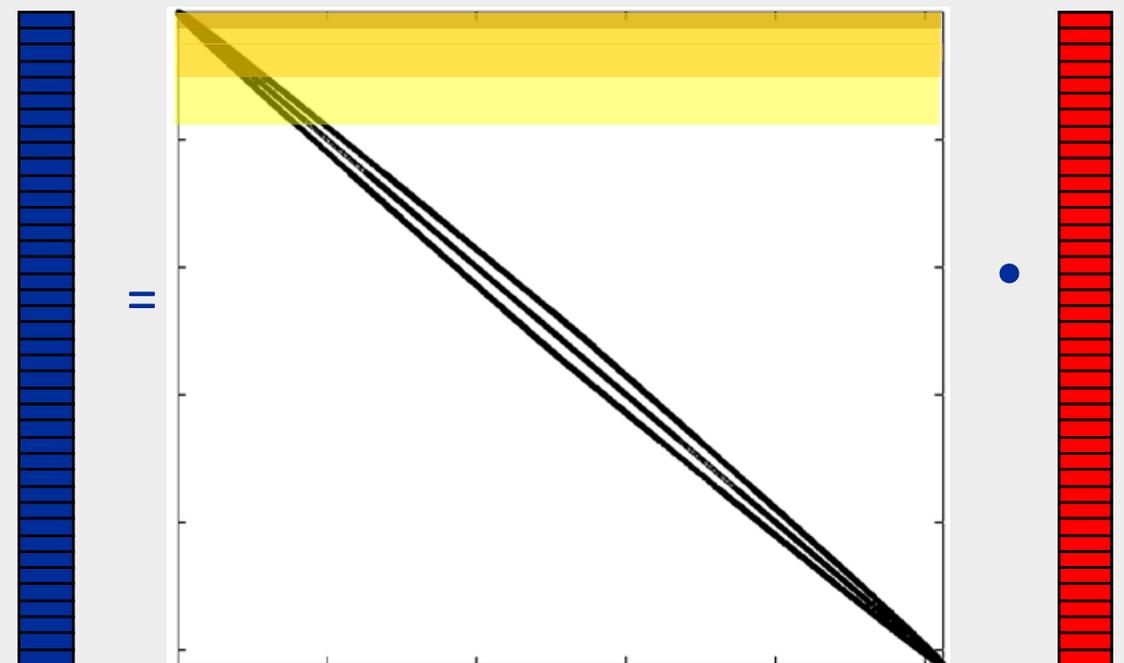
$$\text{Calculate } y = A^3 x$$

## TRAD approach



Matrix accessed 3 times from memory

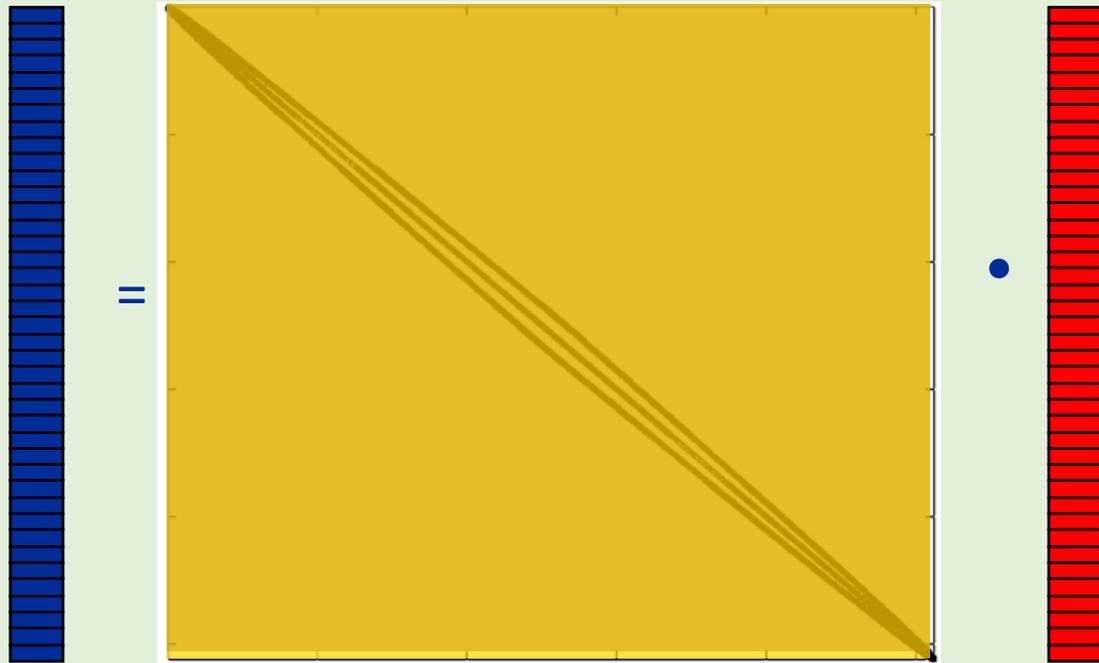
## RACE approach



# Matrix power – Traditional approach vs. Cache Blocking

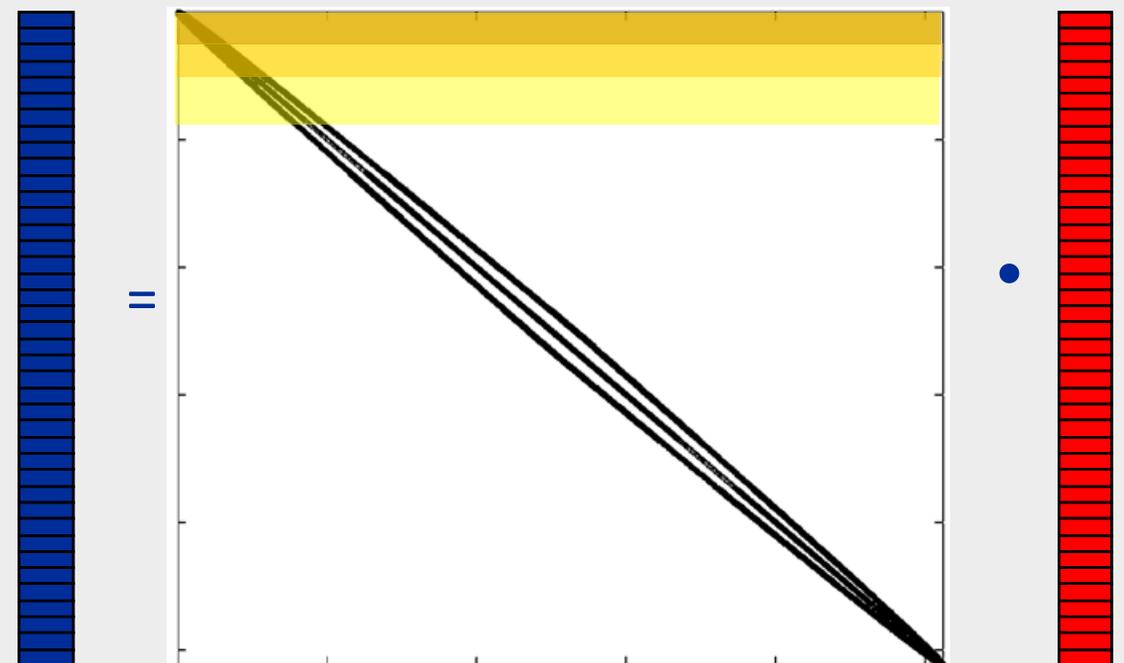
$$\text{Calculate } y = A^3 x$$

**TRAD approach**



Matrix accessed 3 times from memory

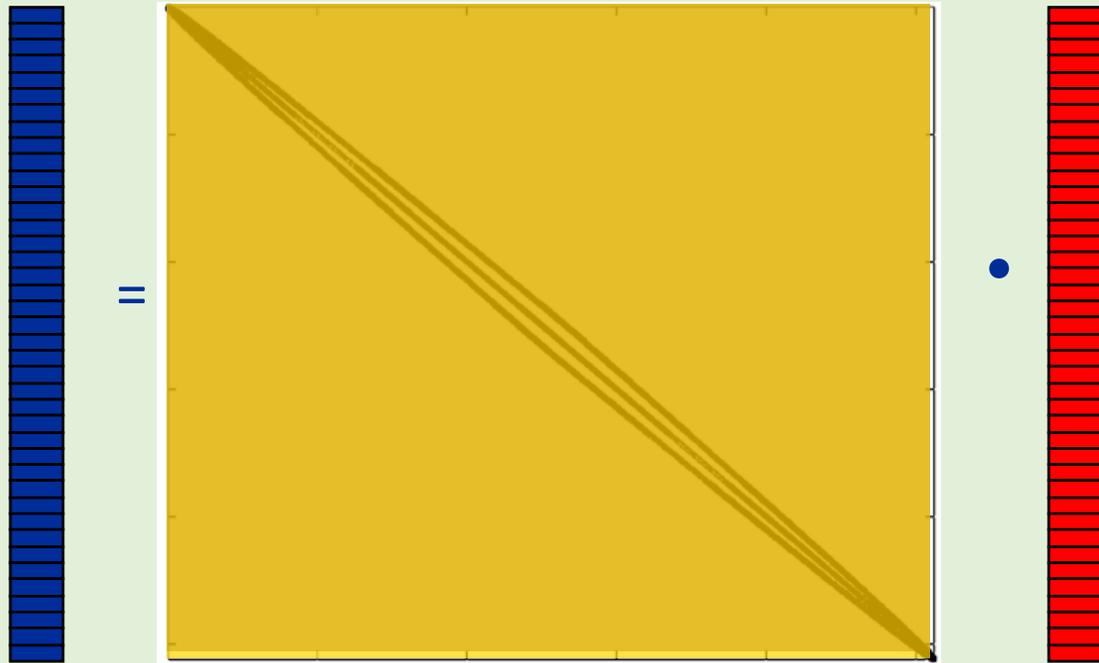
**RACE approach**



# Matrix power – Traditional approach vs. Cache Blocking

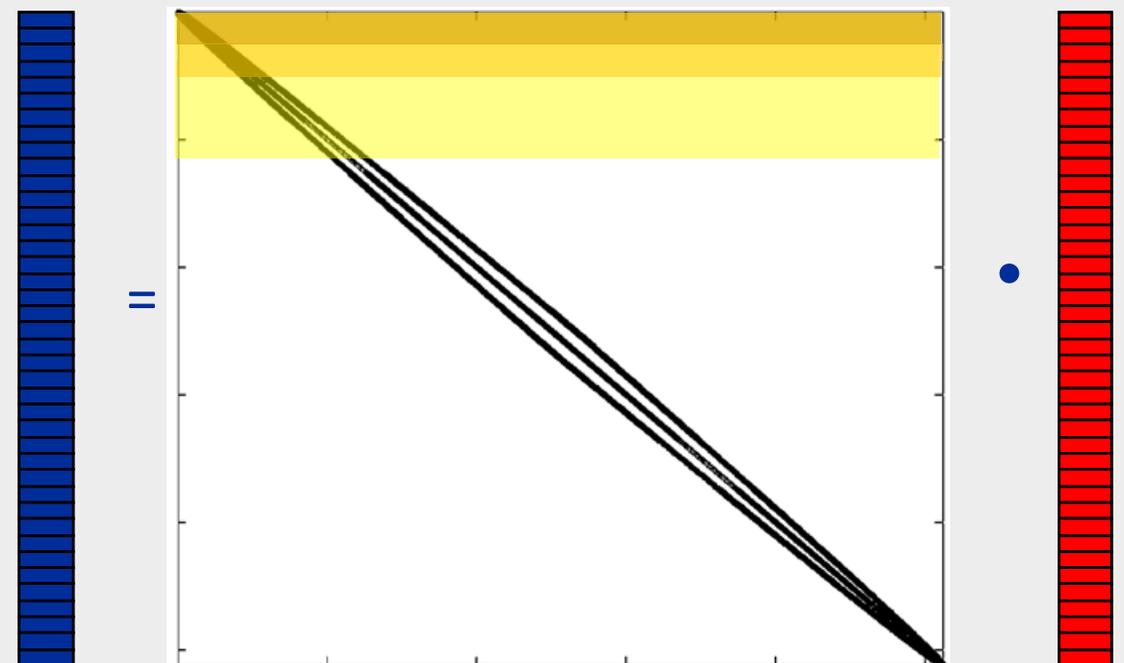
$$\text{Calculate } y = A^3 x$$

## TRAD approach



Matrix accessed 3 times from memory

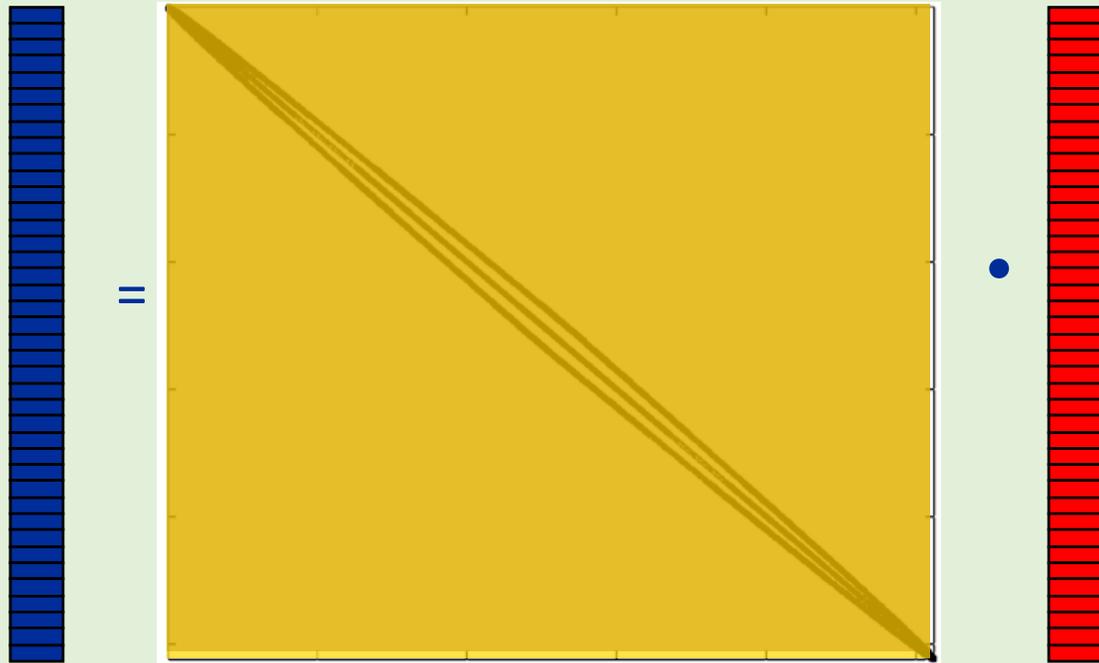
## RACE approach



# Matrix power – Traditional approach vs. Cache Blocking

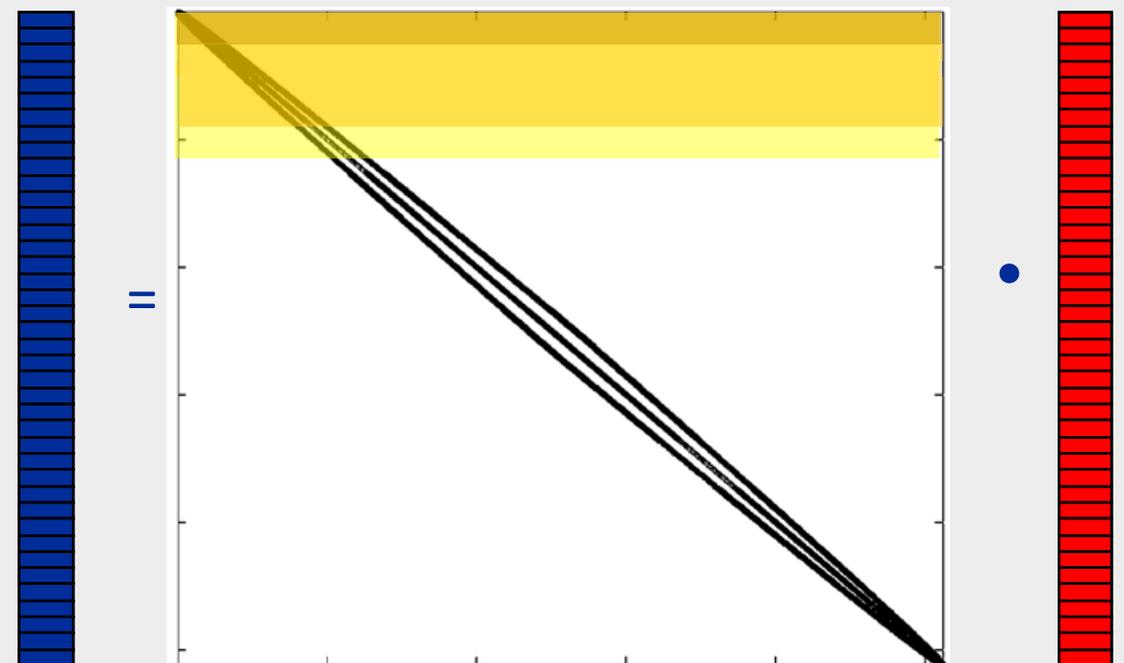
$$\text{Calculate } y = A^3 x$$

## TRAD approach



Matrix accessed 3 times from memory

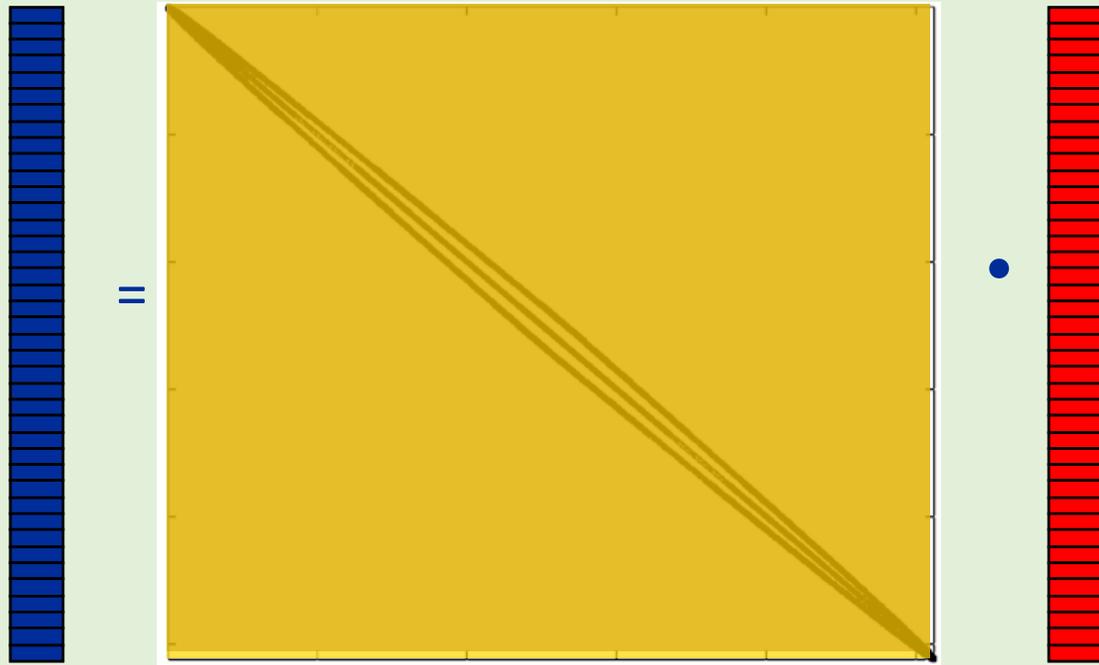
## RACE approach



# Matrix power – Traditional approach vs. Cache Blocking

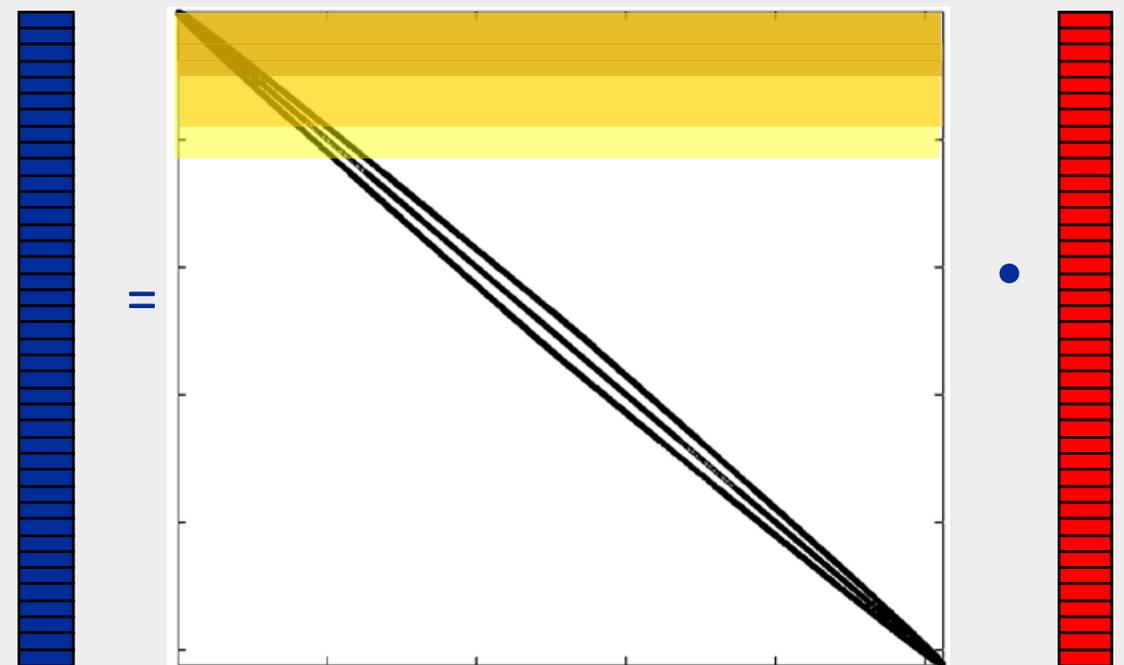
$$\text{Calculate } y = A^3 x$$

**TRAD approach**



Matrix accessed 3 times from memory

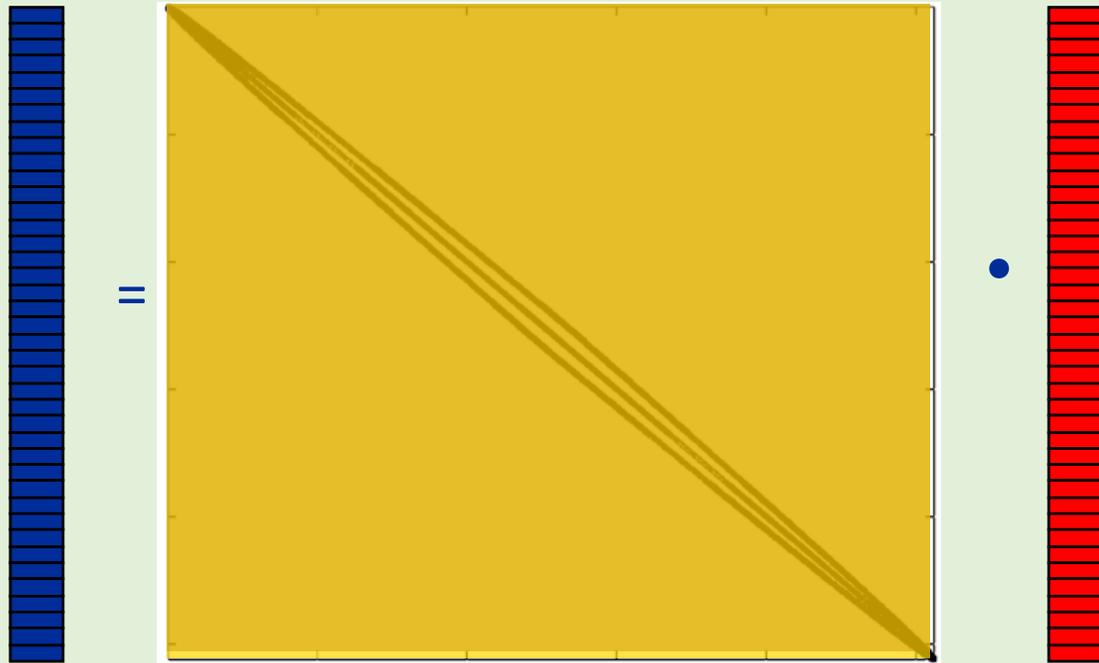
**RACE approach**



# Matrix power – Traditional approach vs. Cache Blocking

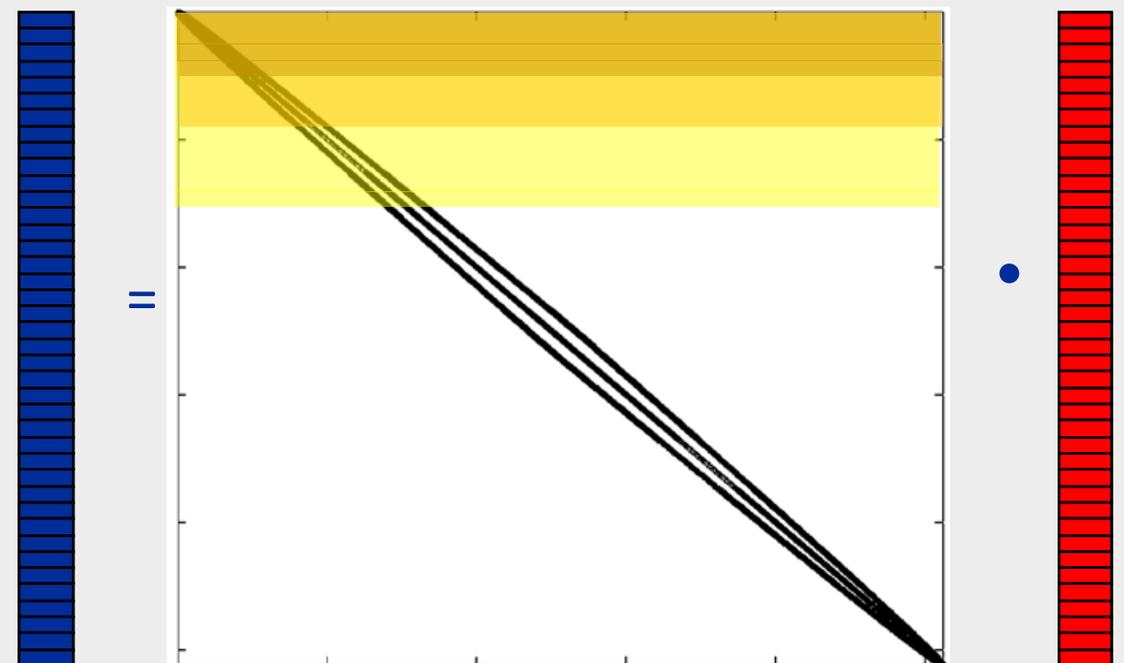
$$\text{Calculate } y = A^3 x$$

## TRAD approach



Matrix accessed 3 times from memory

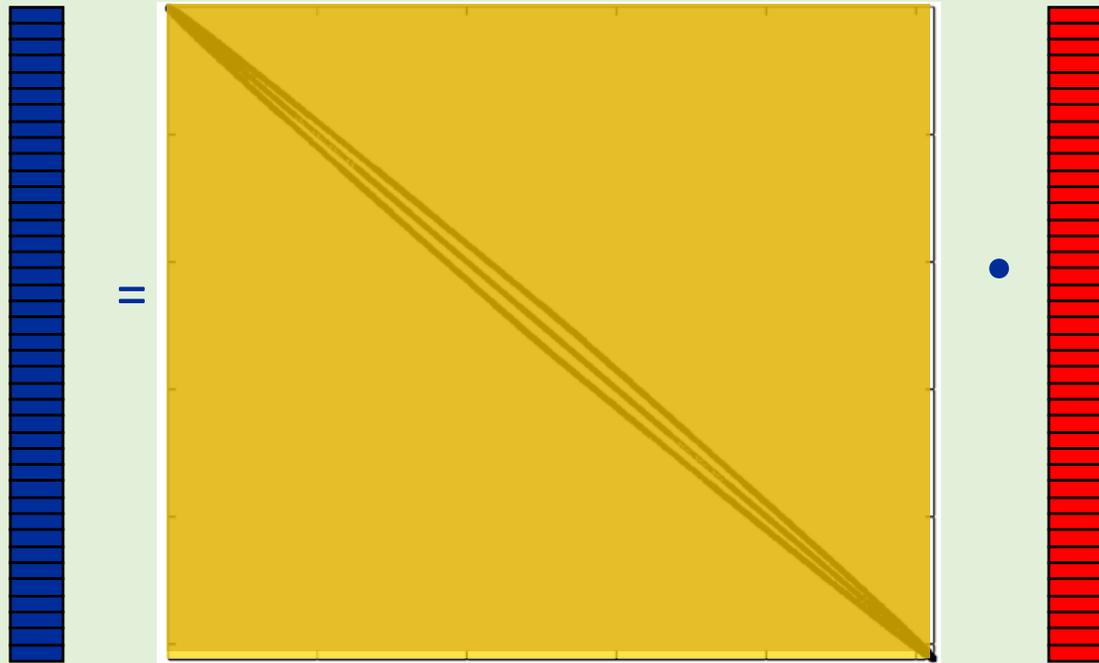
## RACE approach



# Matrix power – Traditional approach vs. Cache Blocking

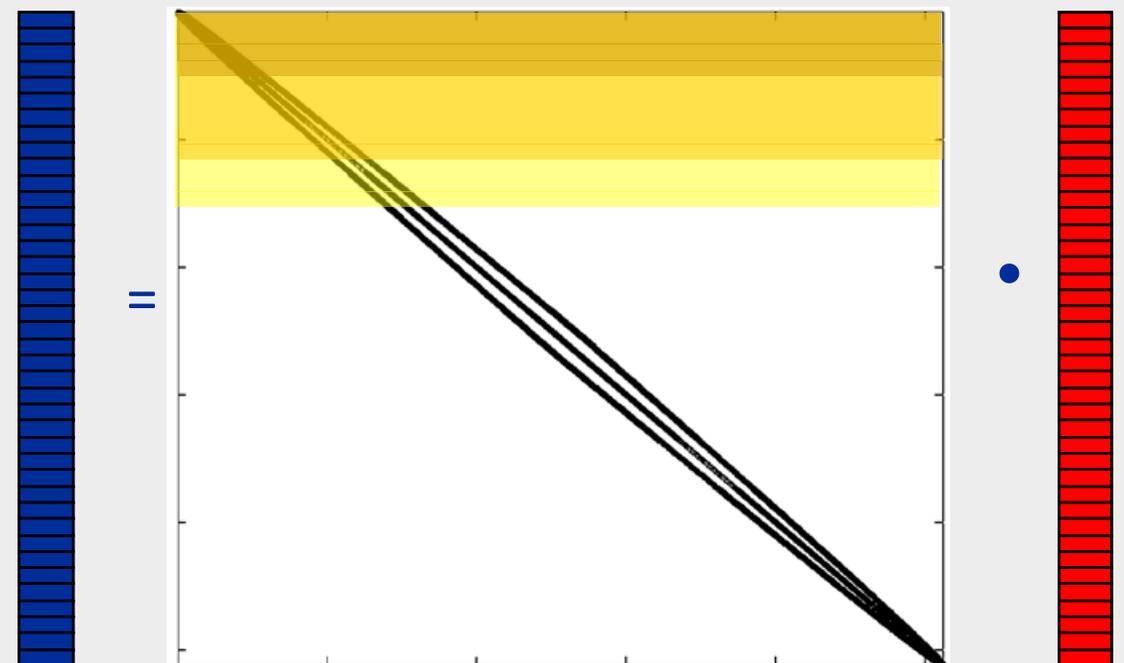
$$\text{Calculate } y = A^3 x$$

**TRAD approach**



Matrix accessed 3 times from memory

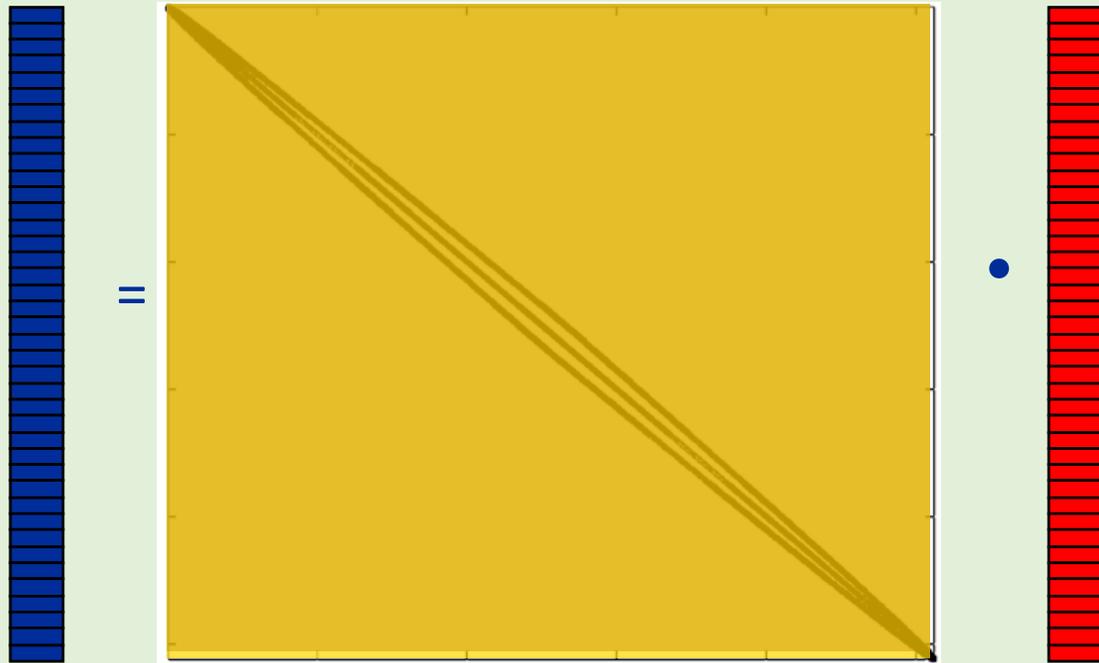
**RACE approach**



# Matrix power – Traditional approach vs. Cache Blocking

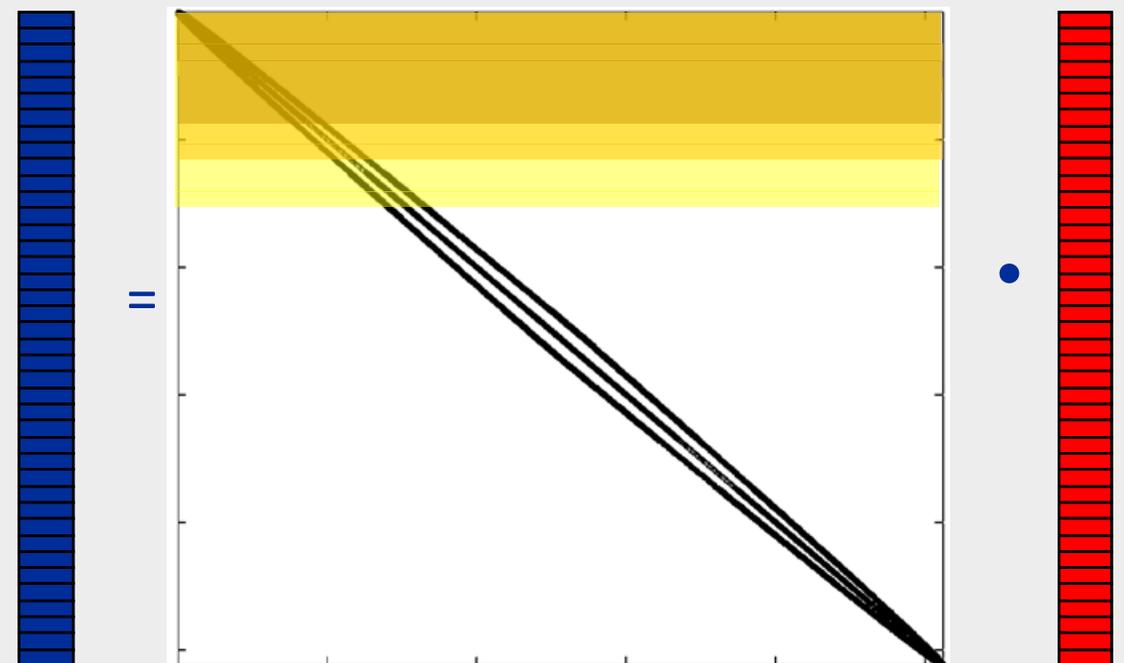
$$\text{Calculate } y = A^3 x$$

## TRAD approach



Matrix accessed 3 times from memory

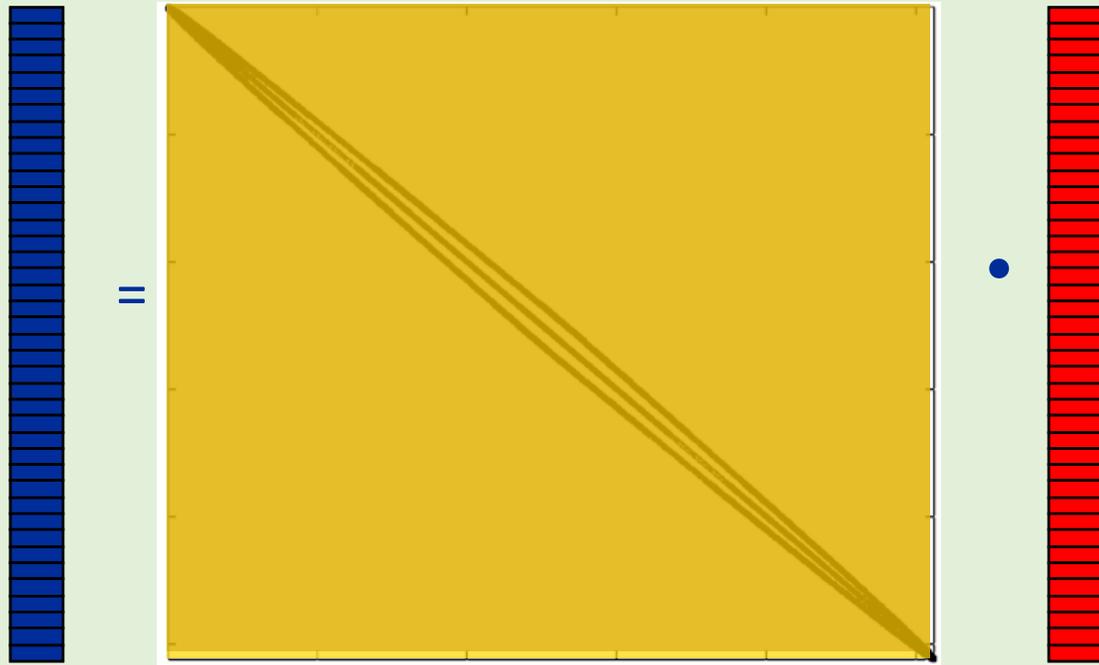
## RACE approach



# Matrix power – Traditional approach vs. Cache Blocking

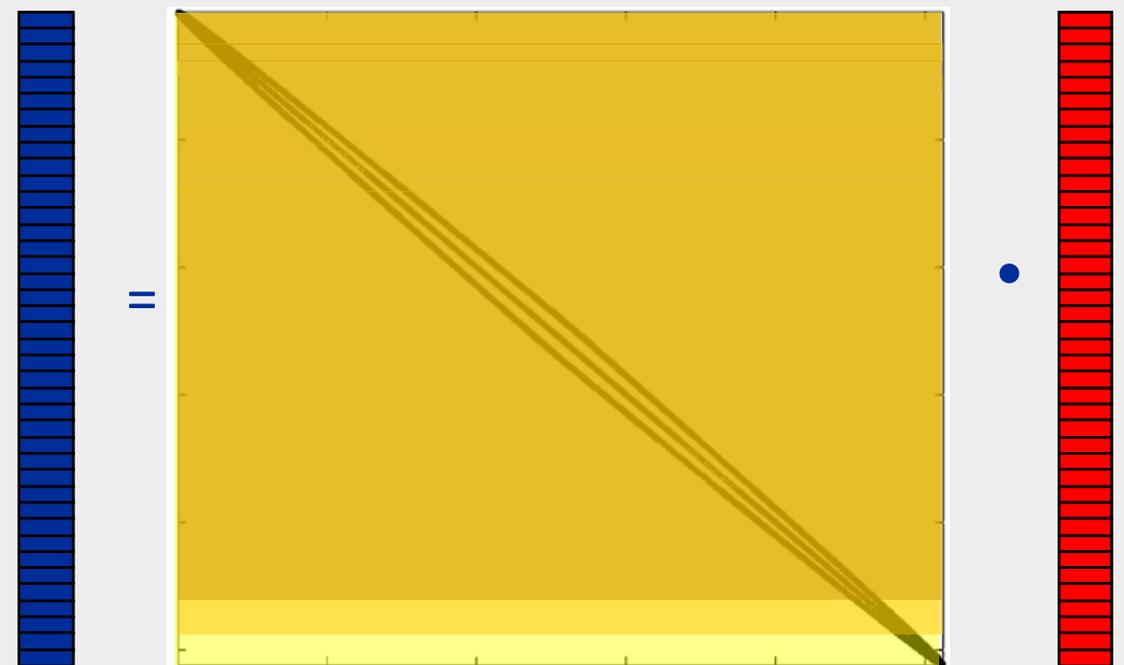
$$\text{Calculate } y = A^3 x$$

**TRAD approach**



Matrix accessed 3 times from memory

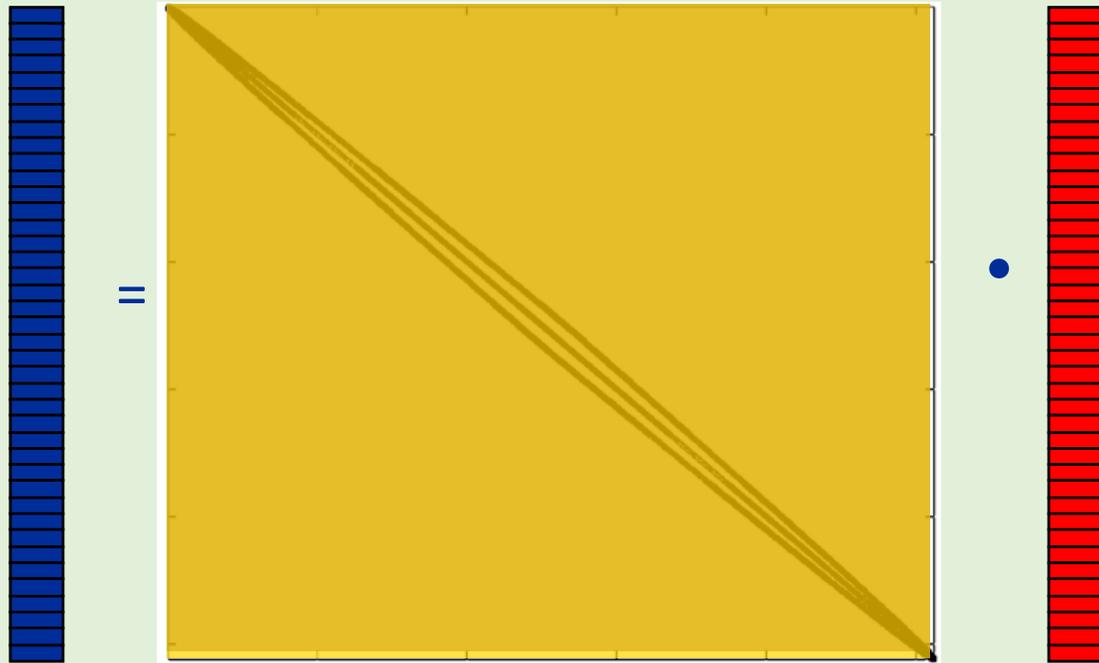
**RACE approach**



# Matrix power – Traditional approach vs. Cache Blocking

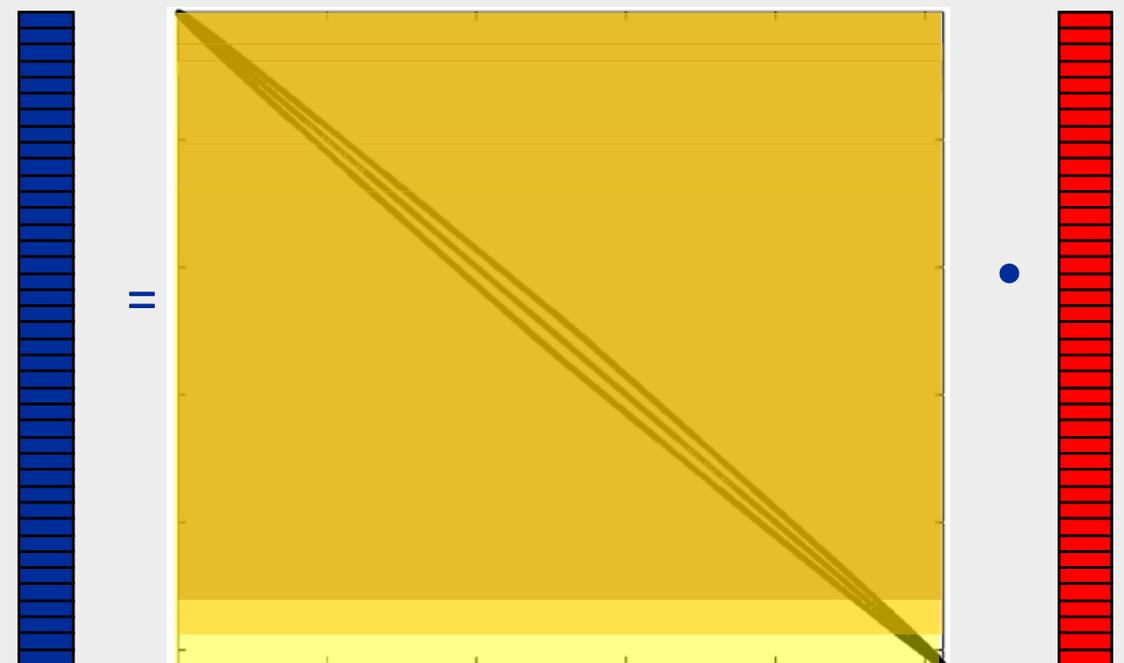
$$\text{Calculate } y = A^3 x$$

**TRAD approach**



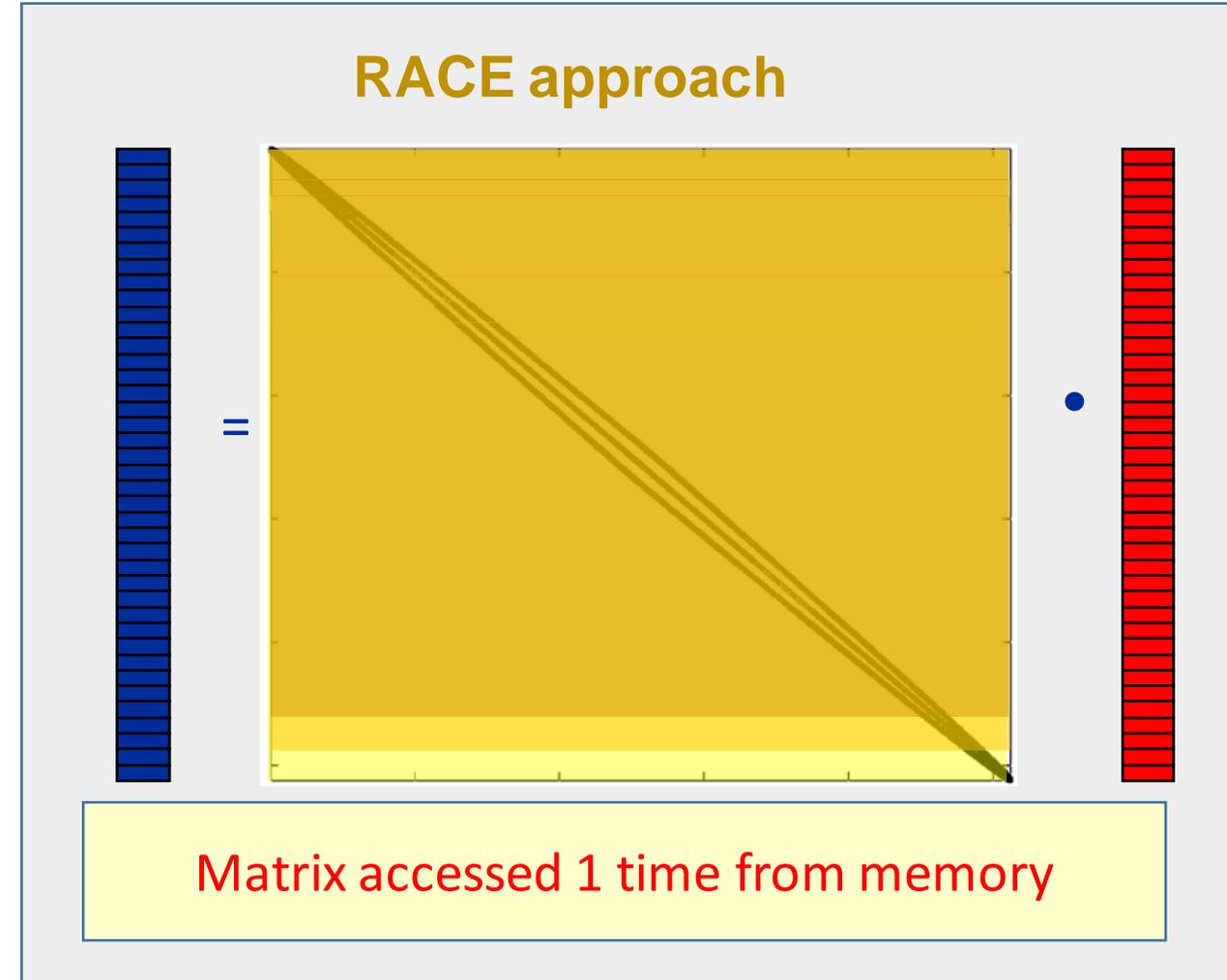
Matrix accessed 3 times from memory

**RACE approach**



Matrix accessed 1 time from memory

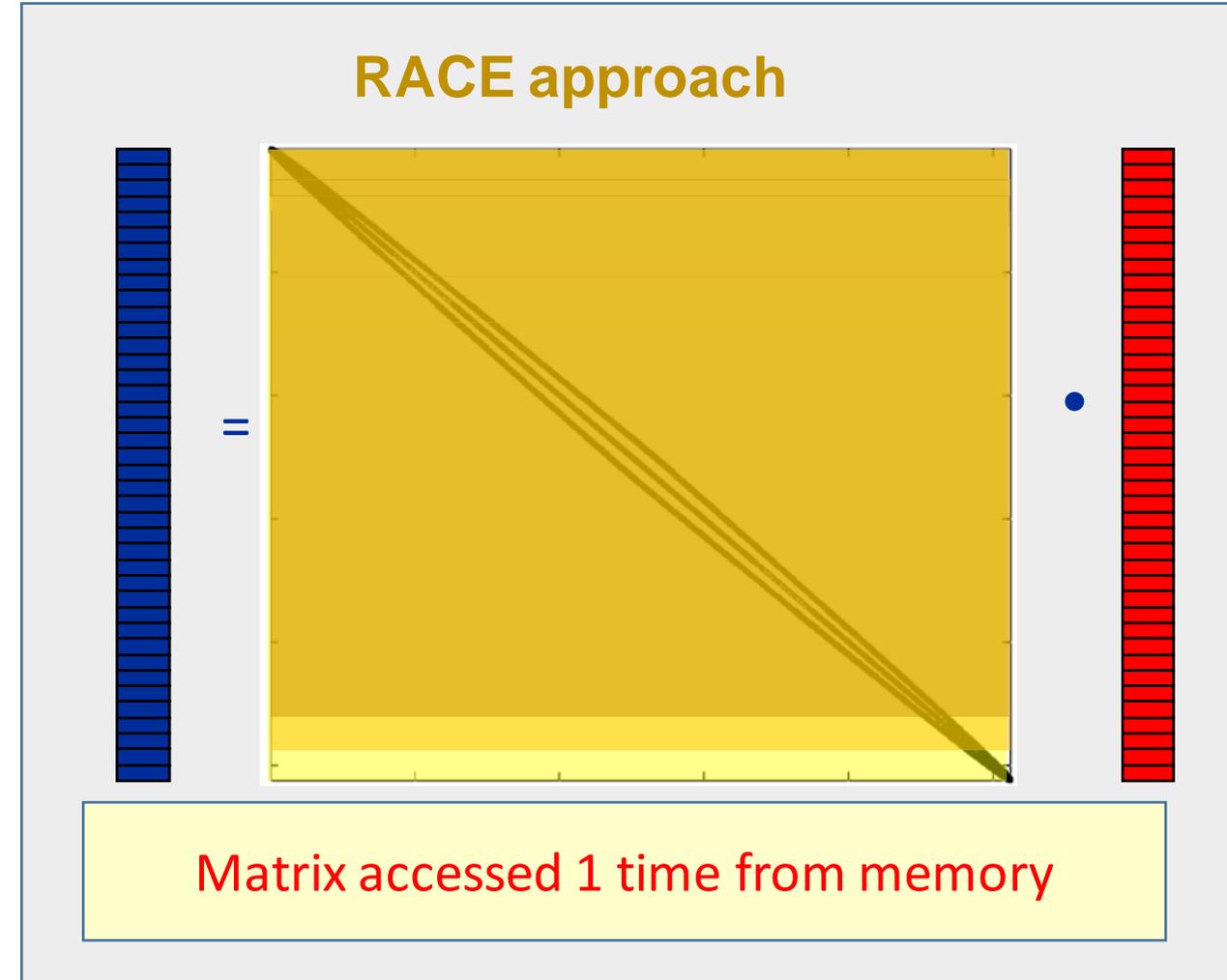
# Matrix power – Traditional approach vs. Cache Blocking



# Matrix power – Traditional approach vs. Cache Blocking



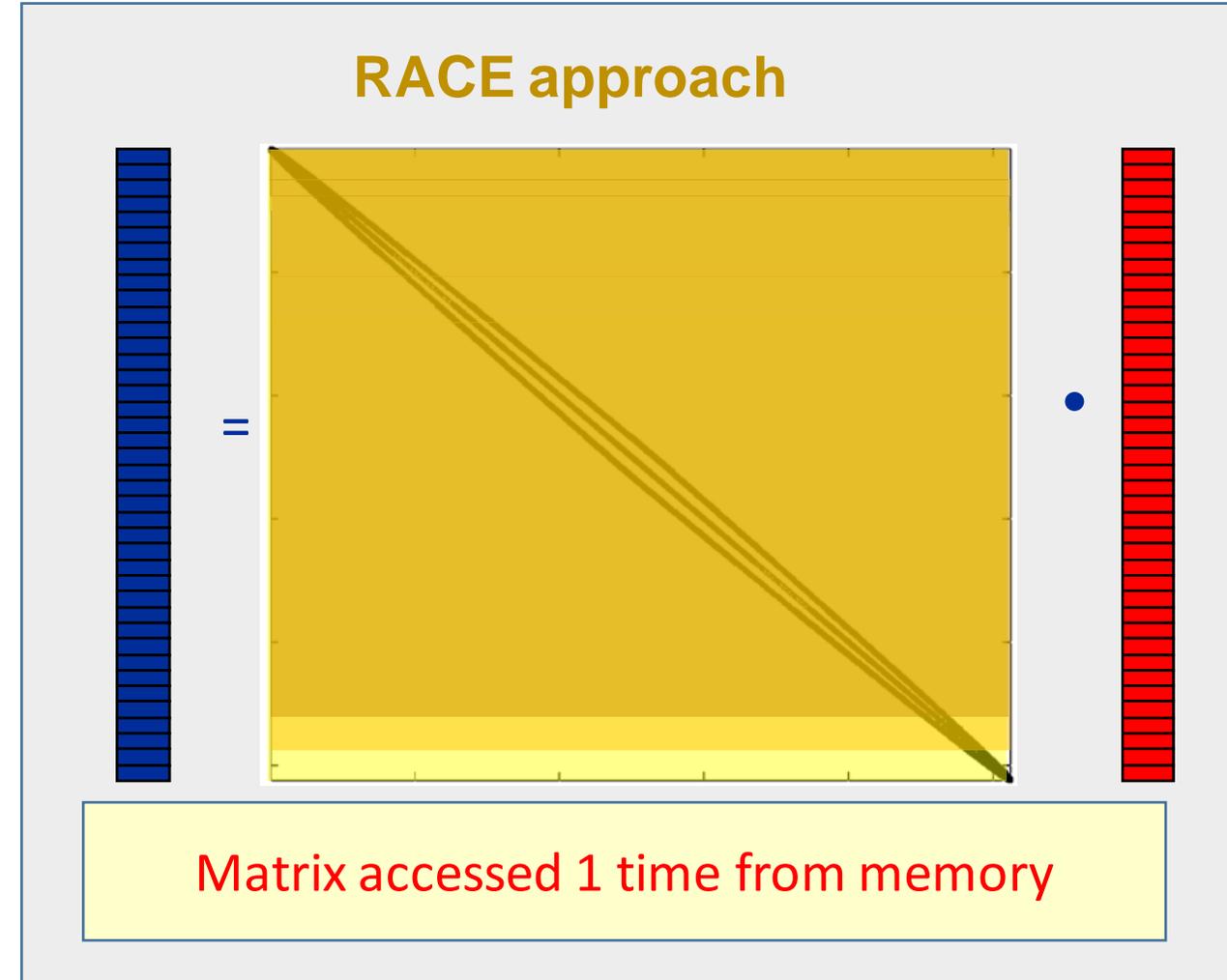
How to do that in general for sparse matrices?



# Matrix power – Traditional approach vs. Cache Blocking



How to do that in general for sparse matrices?

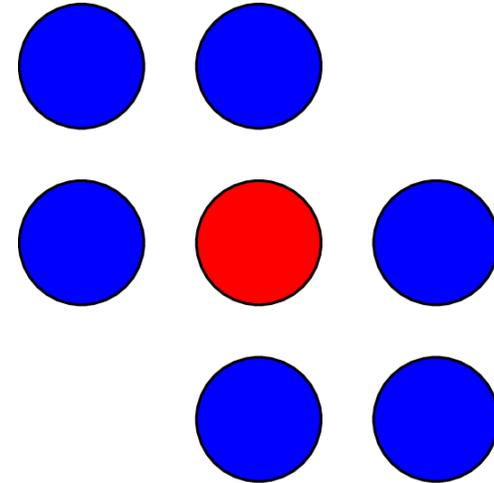
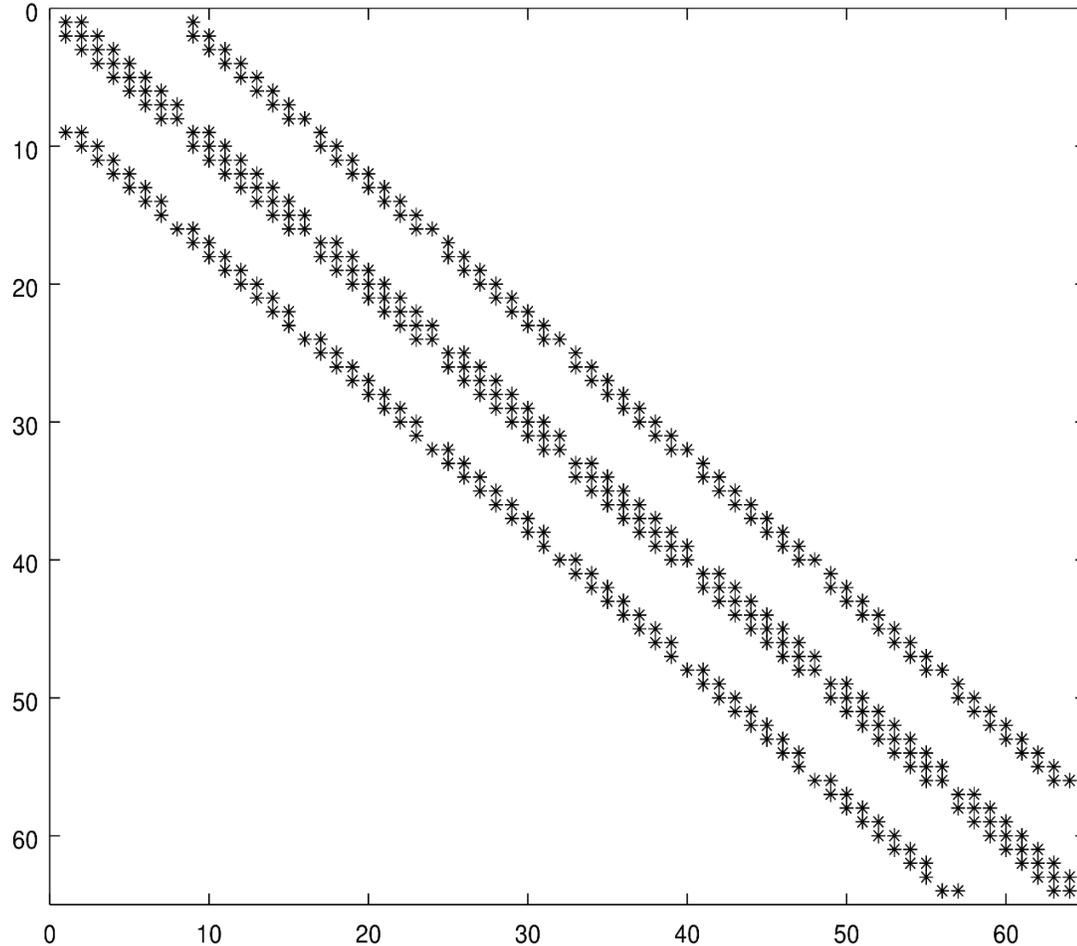


# SpMV – Graph Traversal – RACE



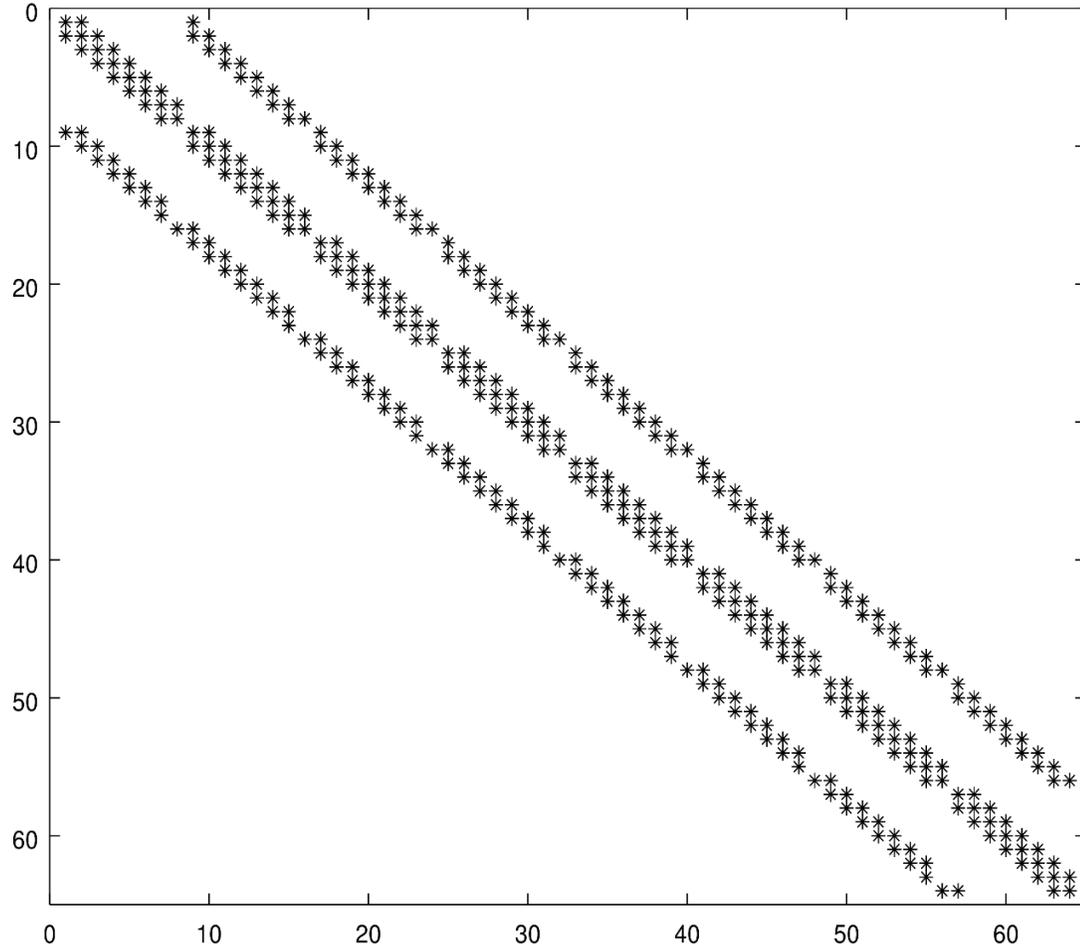
# Sample Stencil Matrix and its graph representation

## Symmetric Matrix (Stencil)

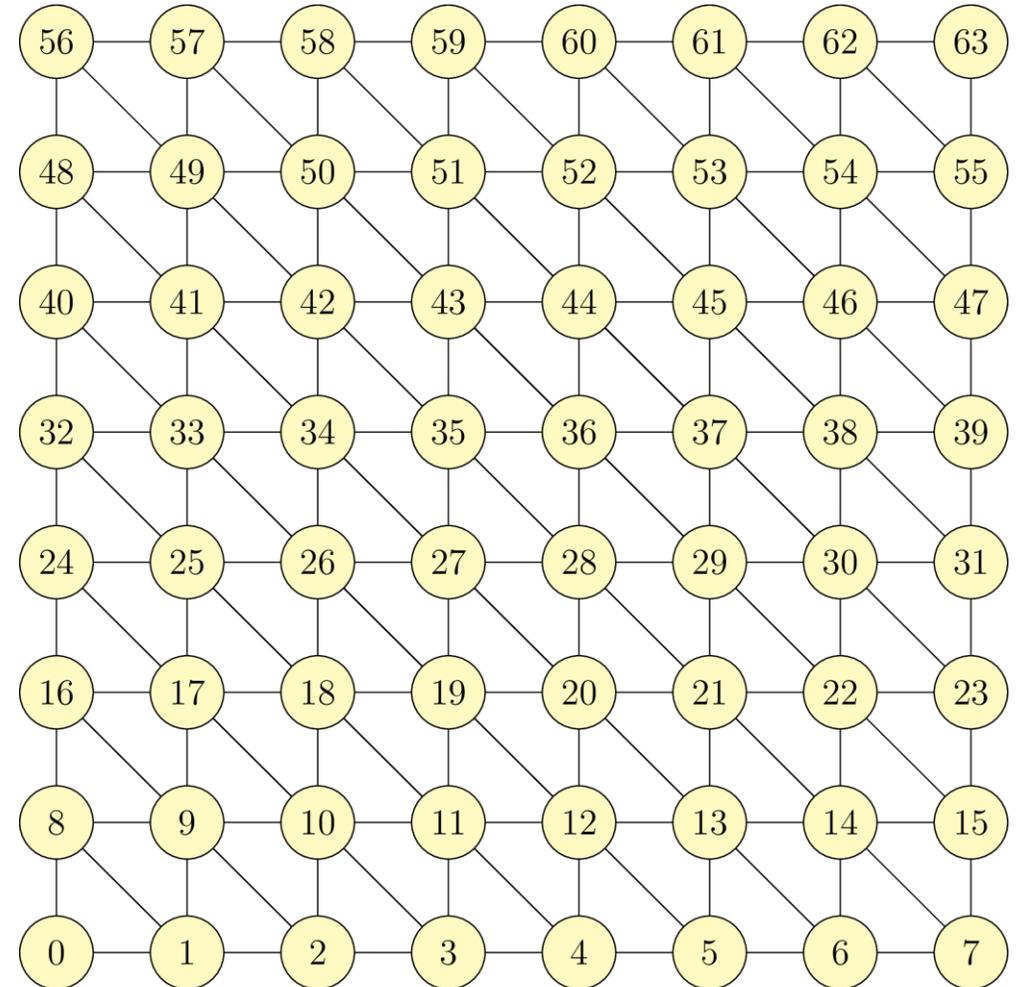


# Sample Stencil Matrix and its graph representation

## Symmetric Matrix (Stencil)

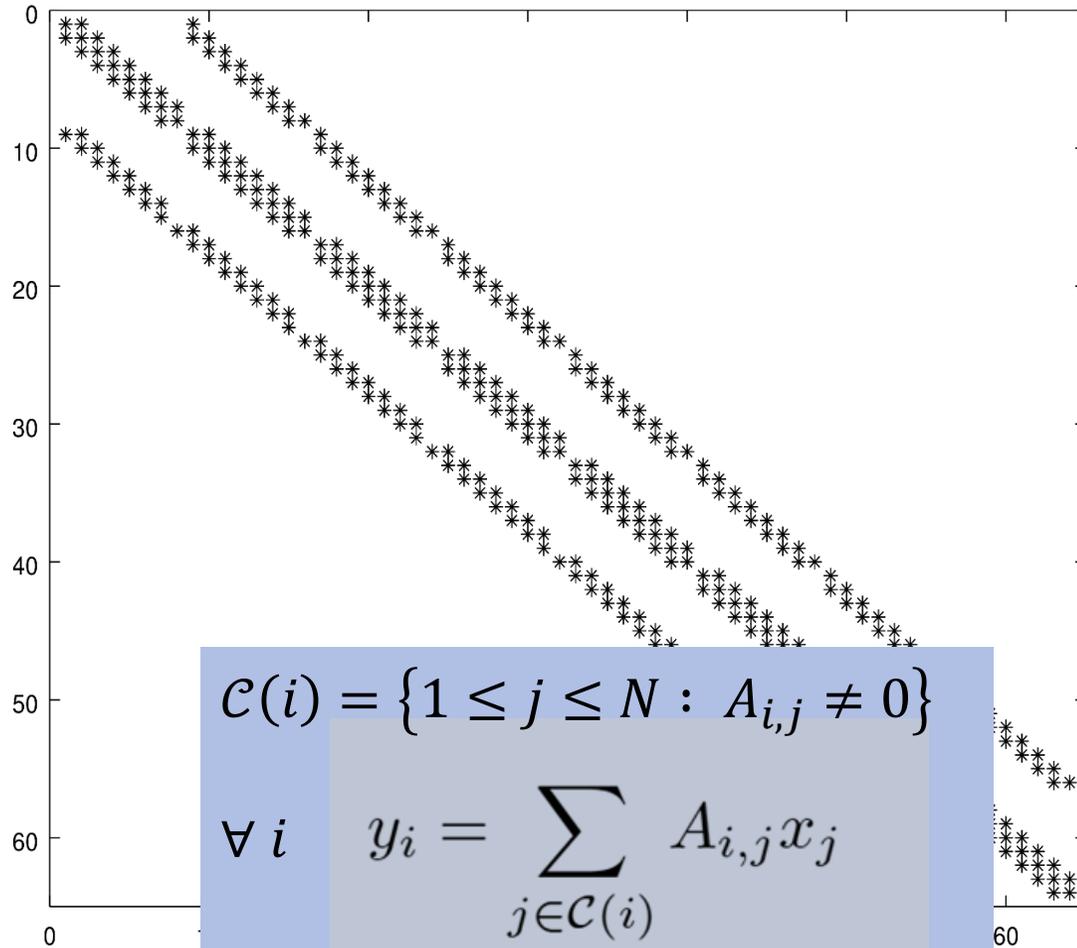


## Undirected Graph

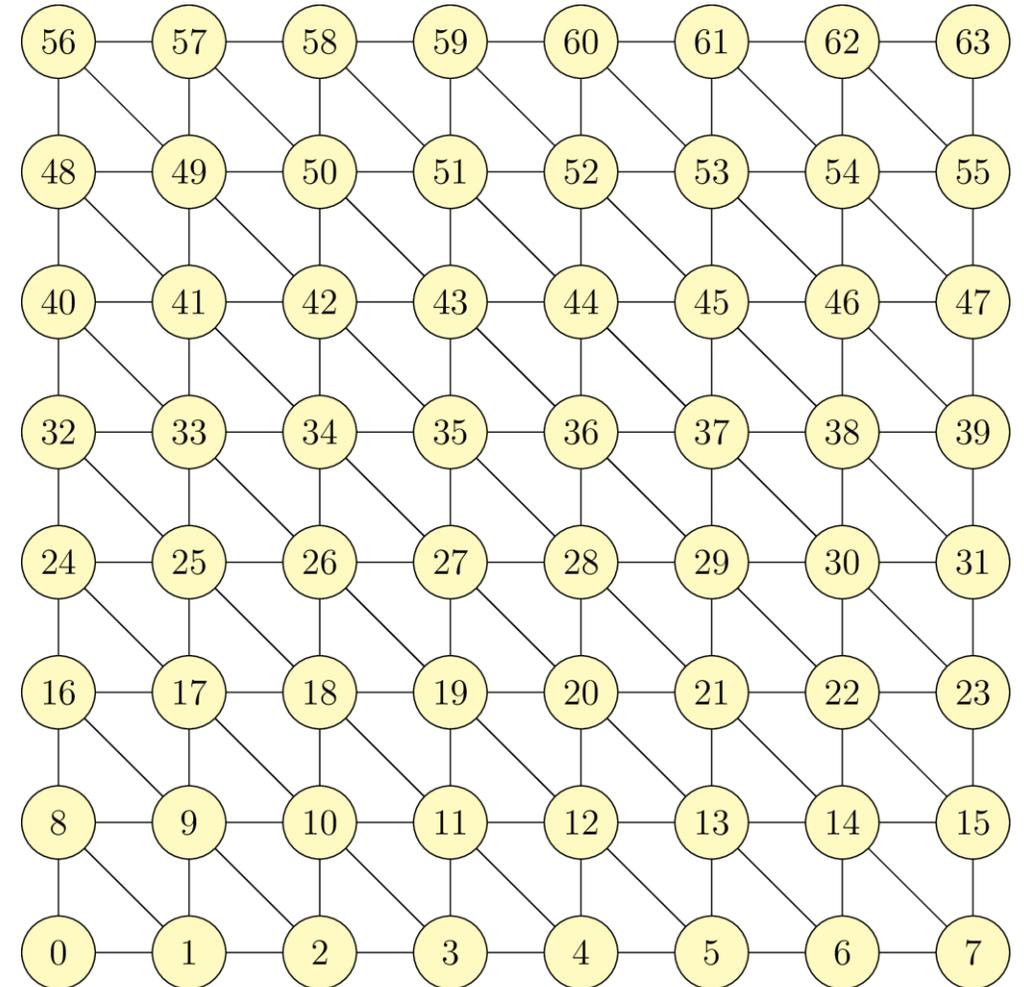


# Sample Stencil Matrix and its graph representation

## Symmetric Matrix (Stencil)

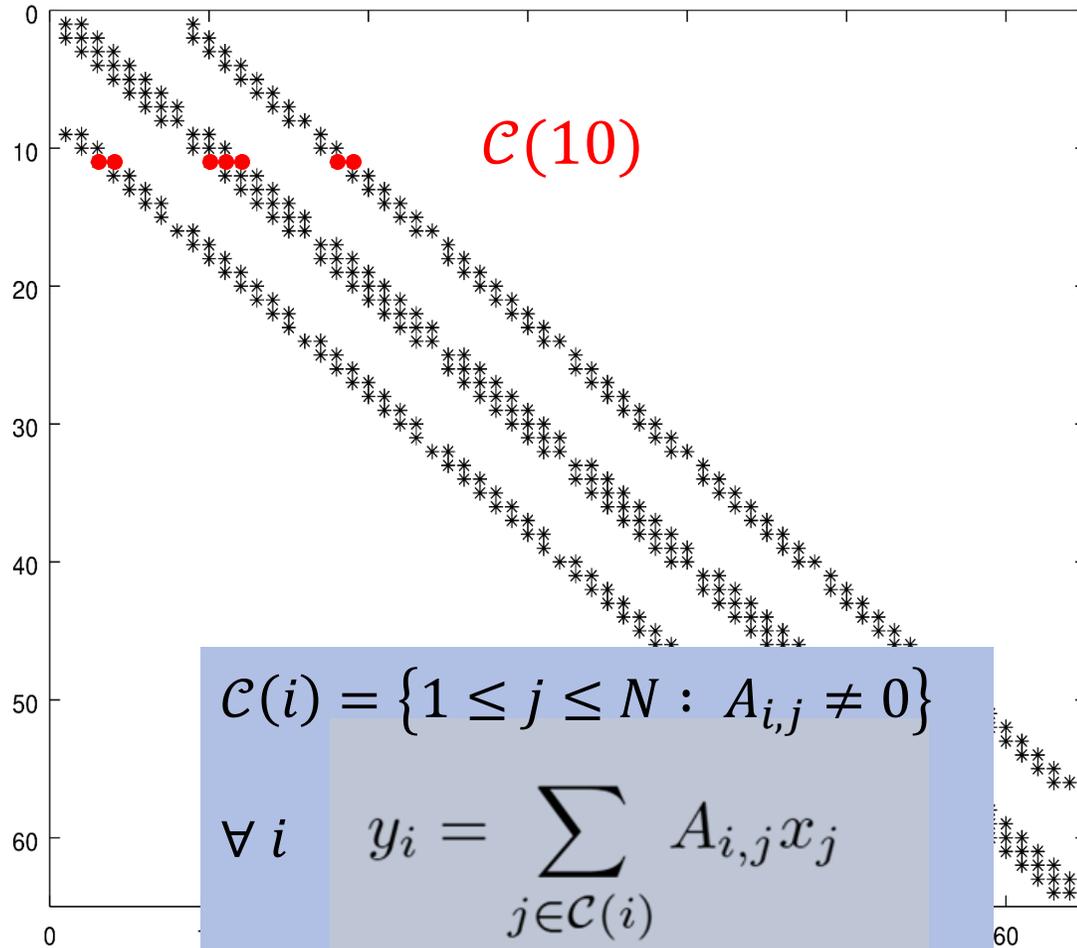


## Undirected Graph

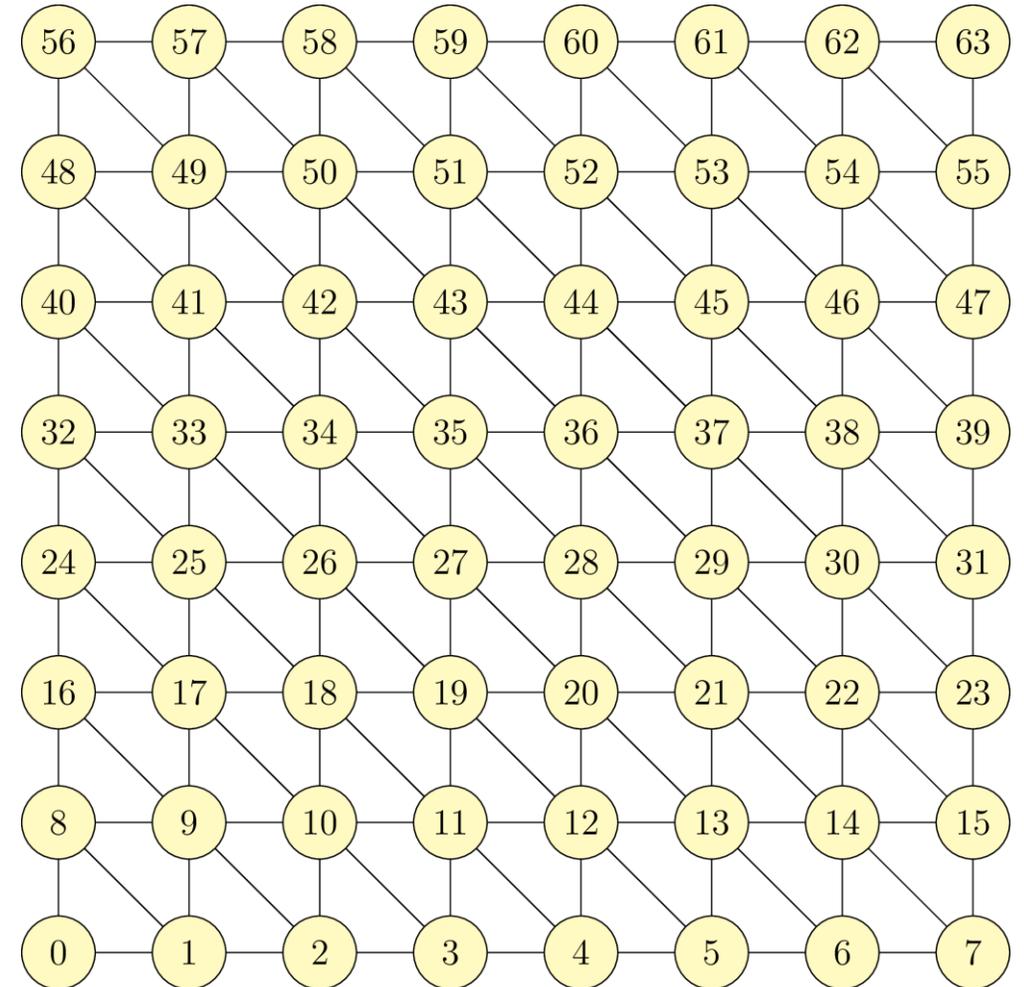


# Sample Stencil Matrix and its graph representation

## Symmetric Matrix (Stencil)

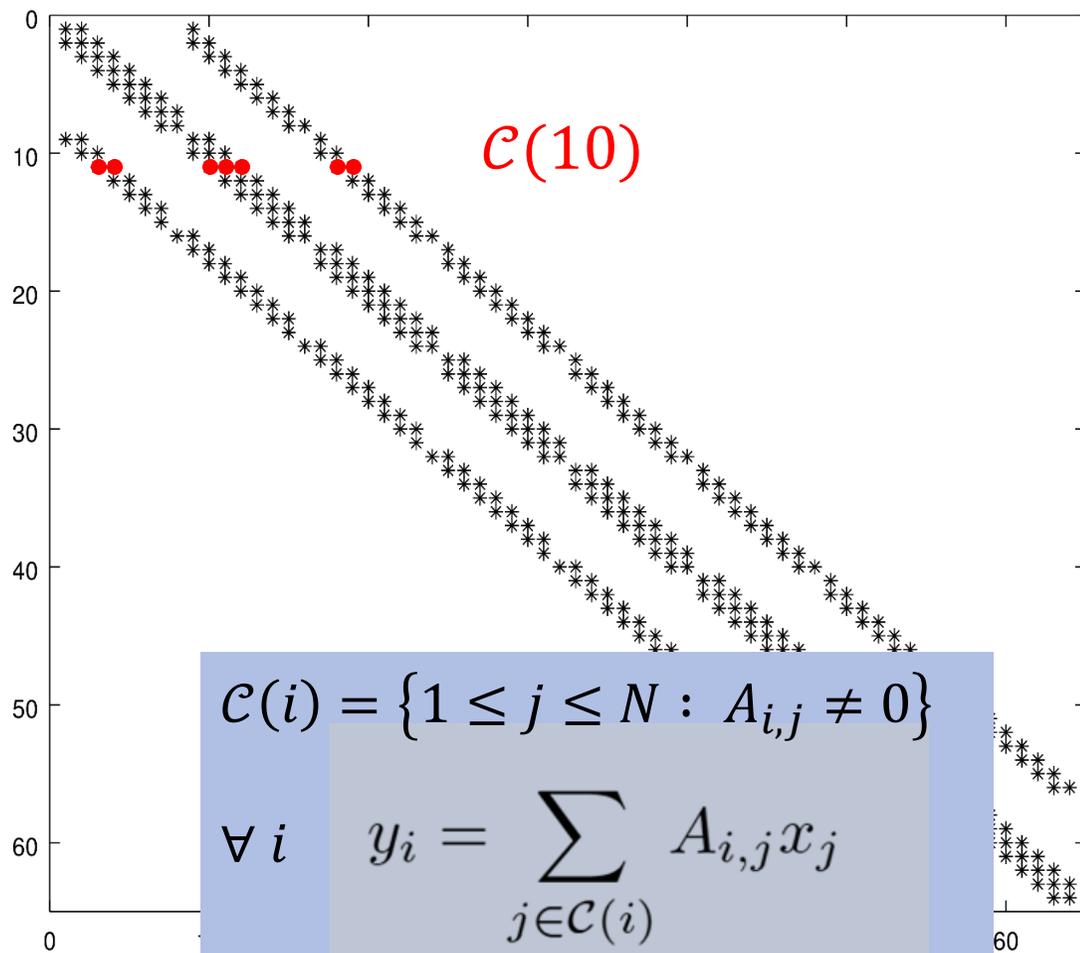


## Undirected Graph

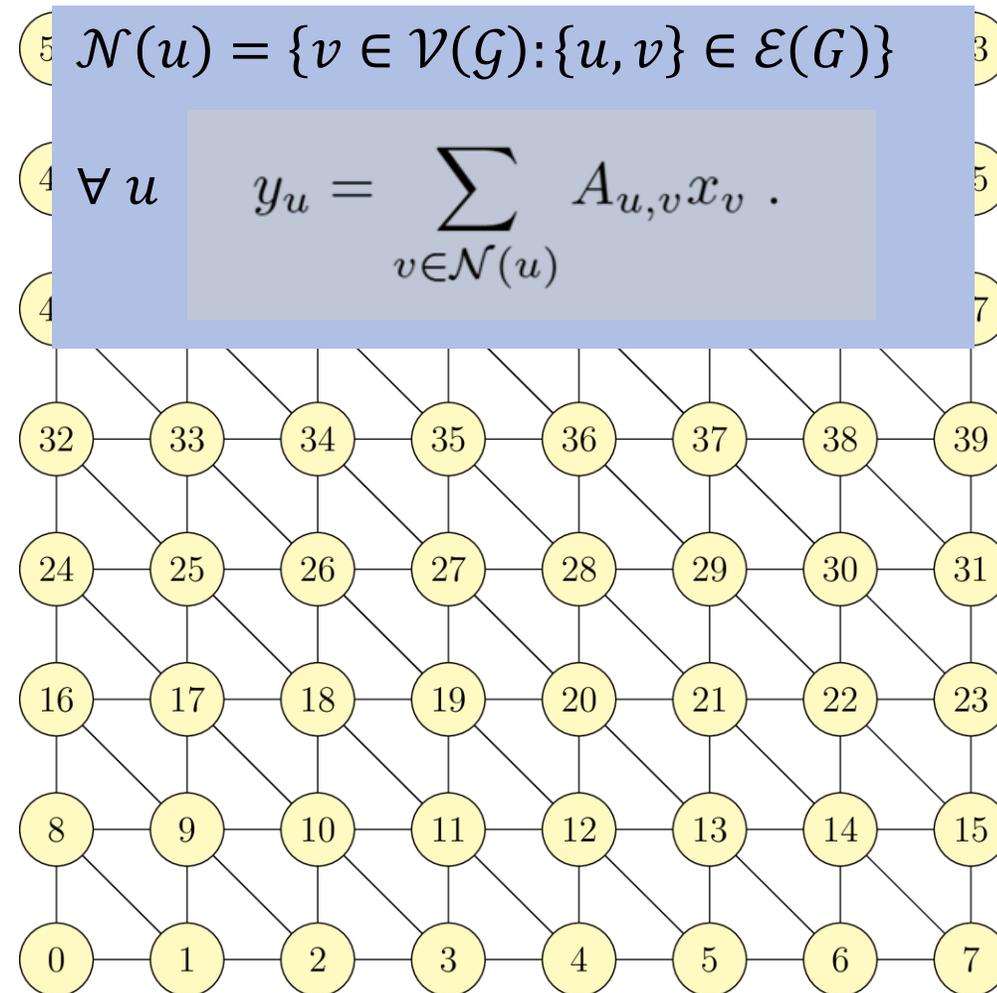


# Sample Stencil Matrix and its graph representation

## Symmetric Matrix (Stencil)

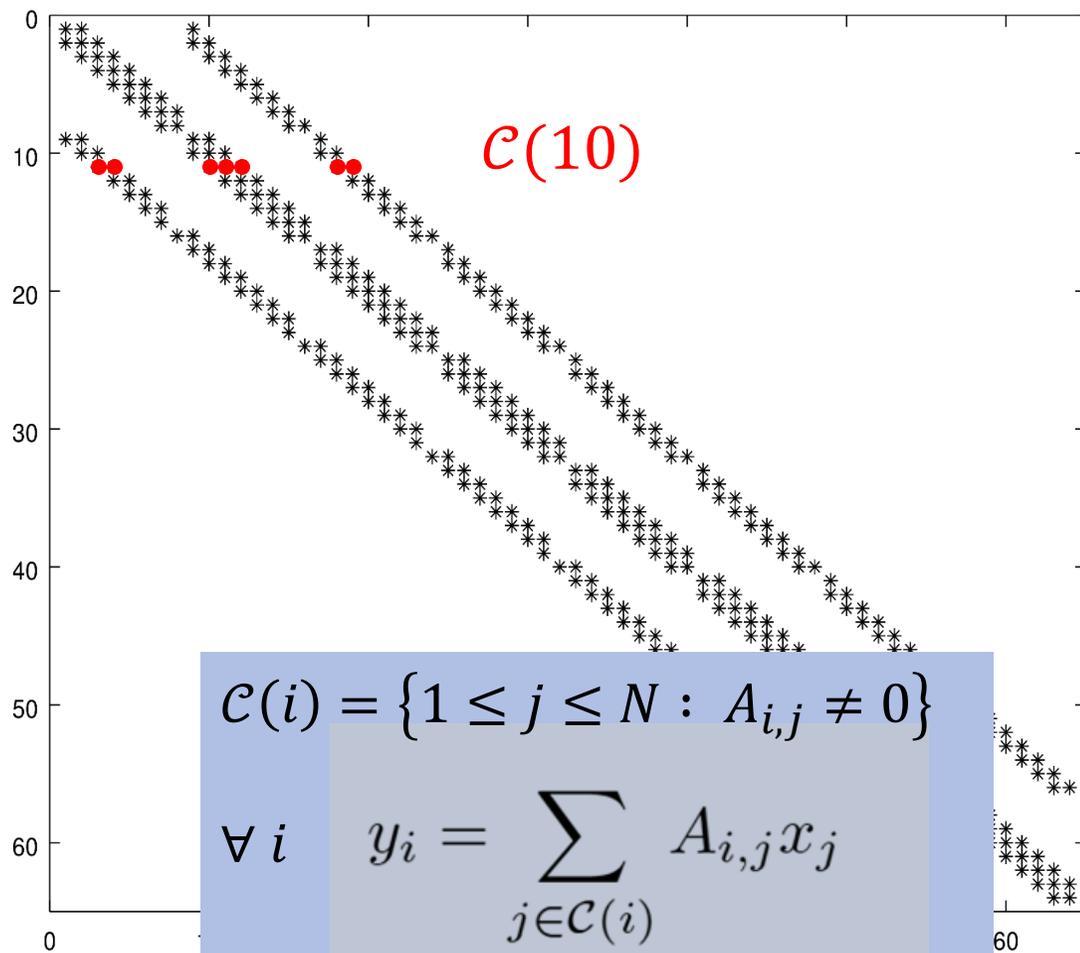


## Undirected Graph

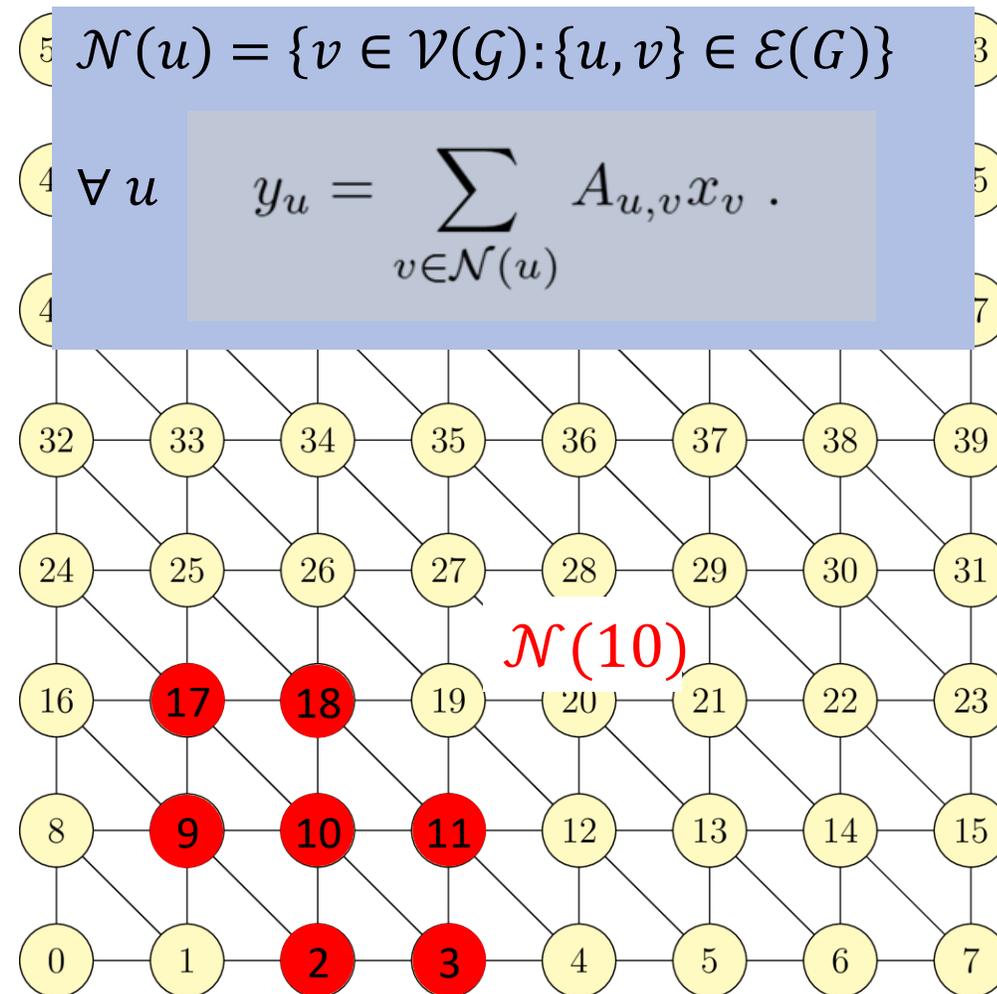


# Sample Stencil Matrix and its graph representation

## Symmetric Matrix (Stencil)



## Undirected Graph



# Consider SpMV as „graph traversal“

- Standard view: Run over all rows  $i=1, \dots, n_r$  and calculate

$$y_i = \sum_{j \in COL(i)} A_{i,j} x_j$$

where  $COL(i)$  contains the column indices of the non-zeros in  $i$ -th row

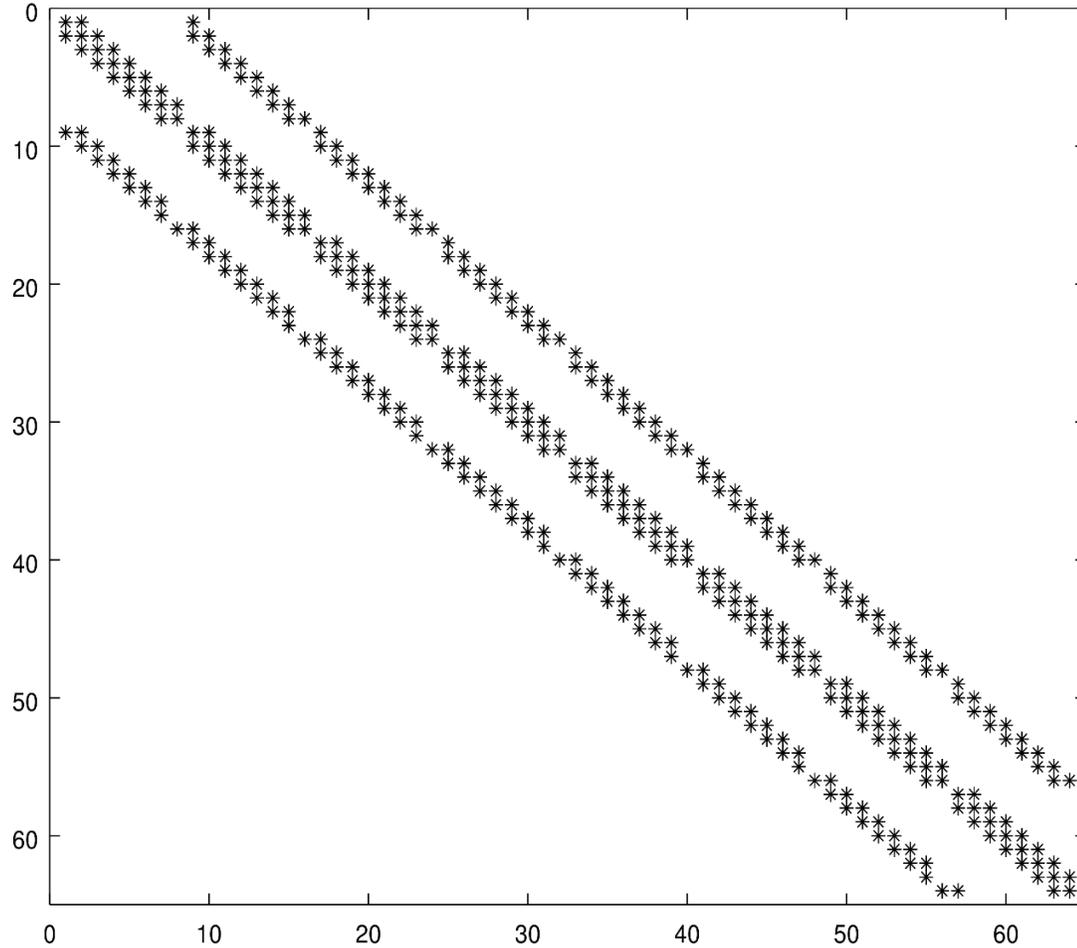
- RACE – **graph-based view**: row  $\leftrightarrow$  vertex & non-zero entry  $\leftrightarrow$  edge

In the graph terminology a SpMV operation ( $y = Ax$ ) can be formulated as follows: If  $G = (V, E)$  is the graph representation of the sparse matrix  $A$  then for every vertex  $u \in V(G)$  calculate

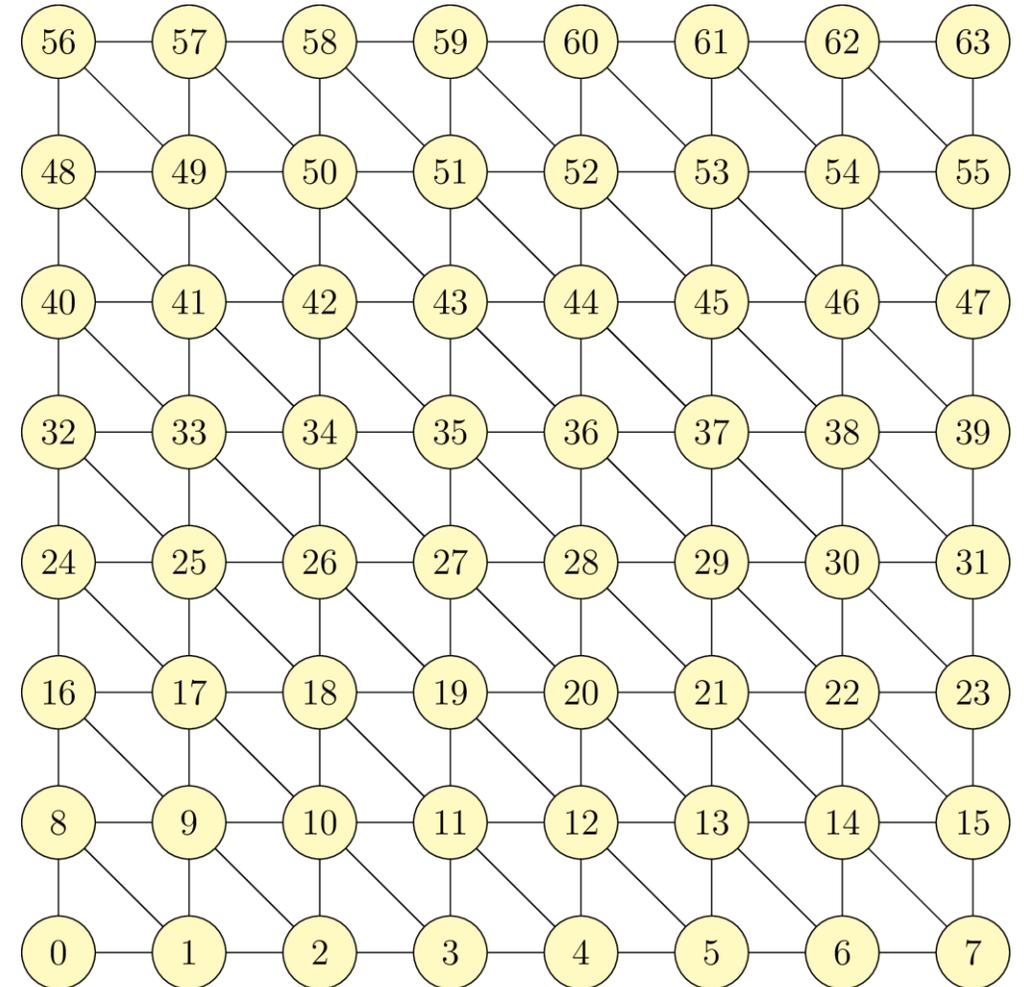
$$y_u = \sum_{v \in N(u)} A_{u,v} x_v . \quad (2)$$

# Sample Stencil Matrix and its graph representation

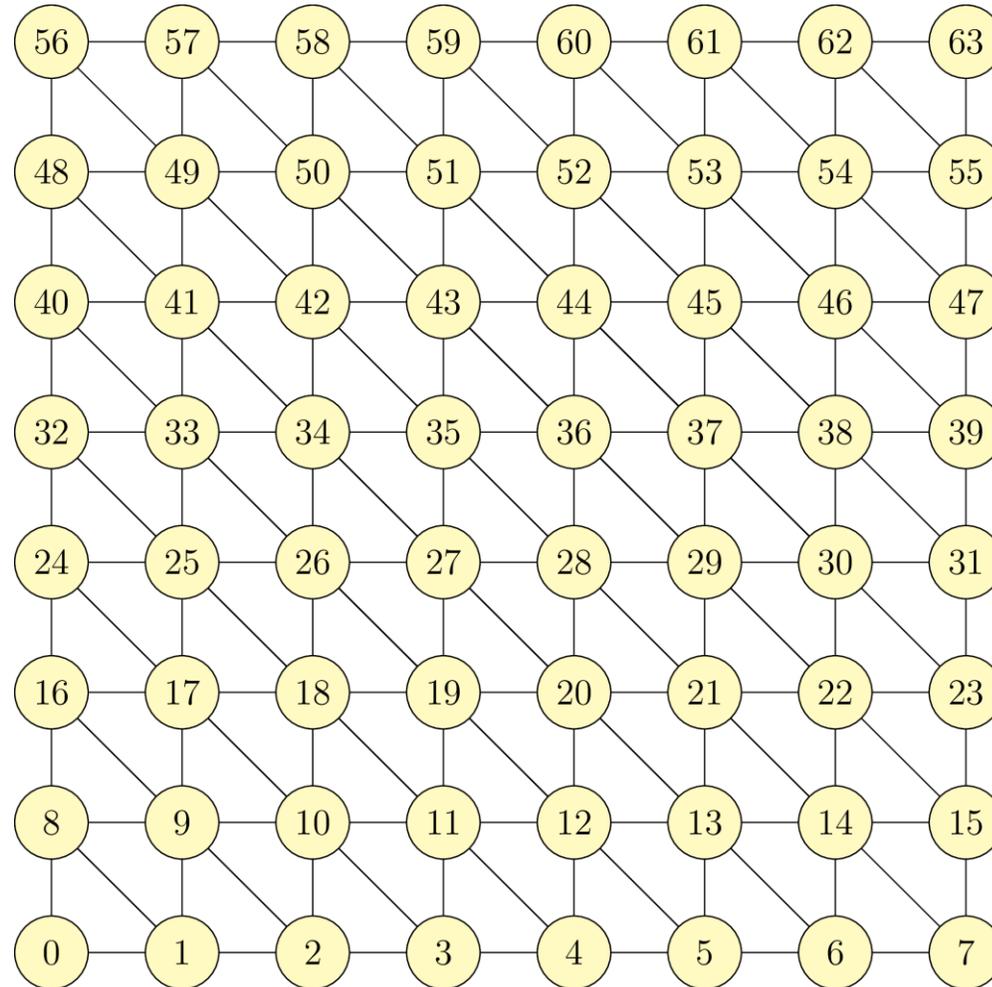
## Symmetric Matrix (Stencil)



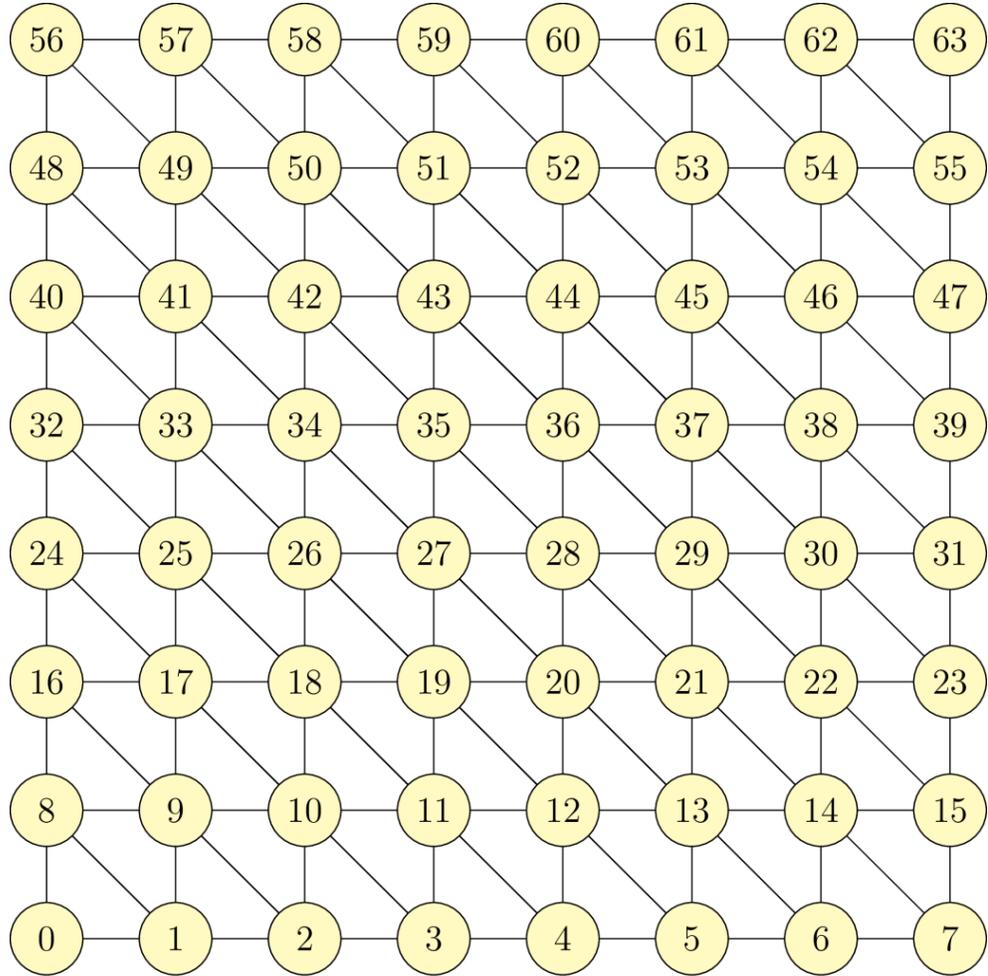
## Undirected Graph

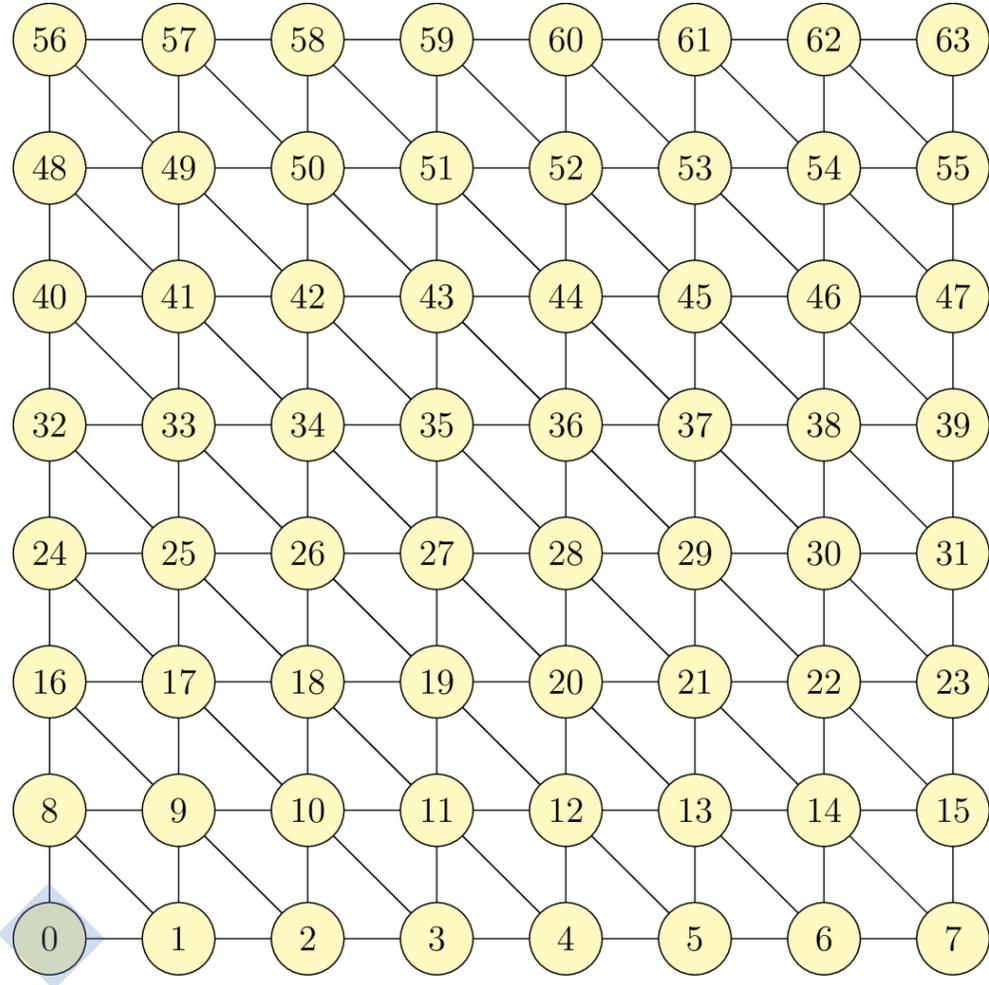


# Sample Stencil Matrix and its graph representation



# RACE

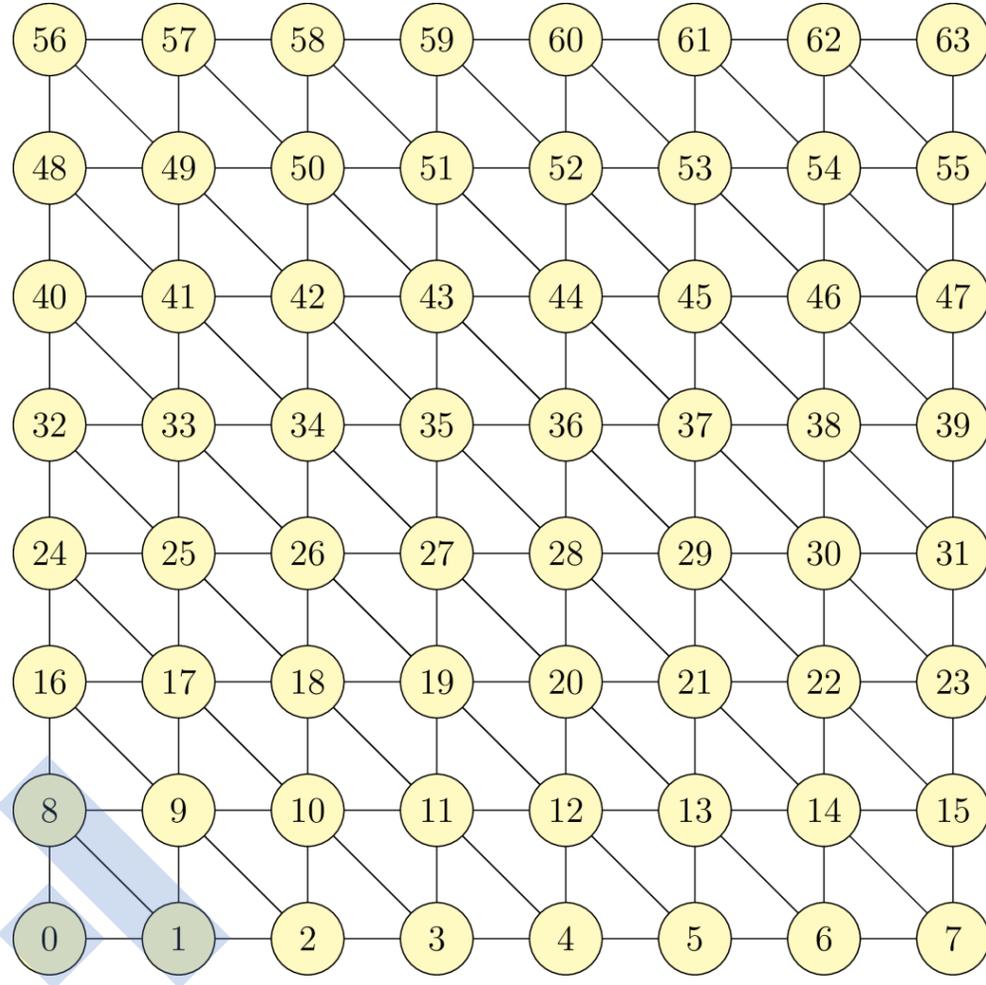




1

levels

# RACE

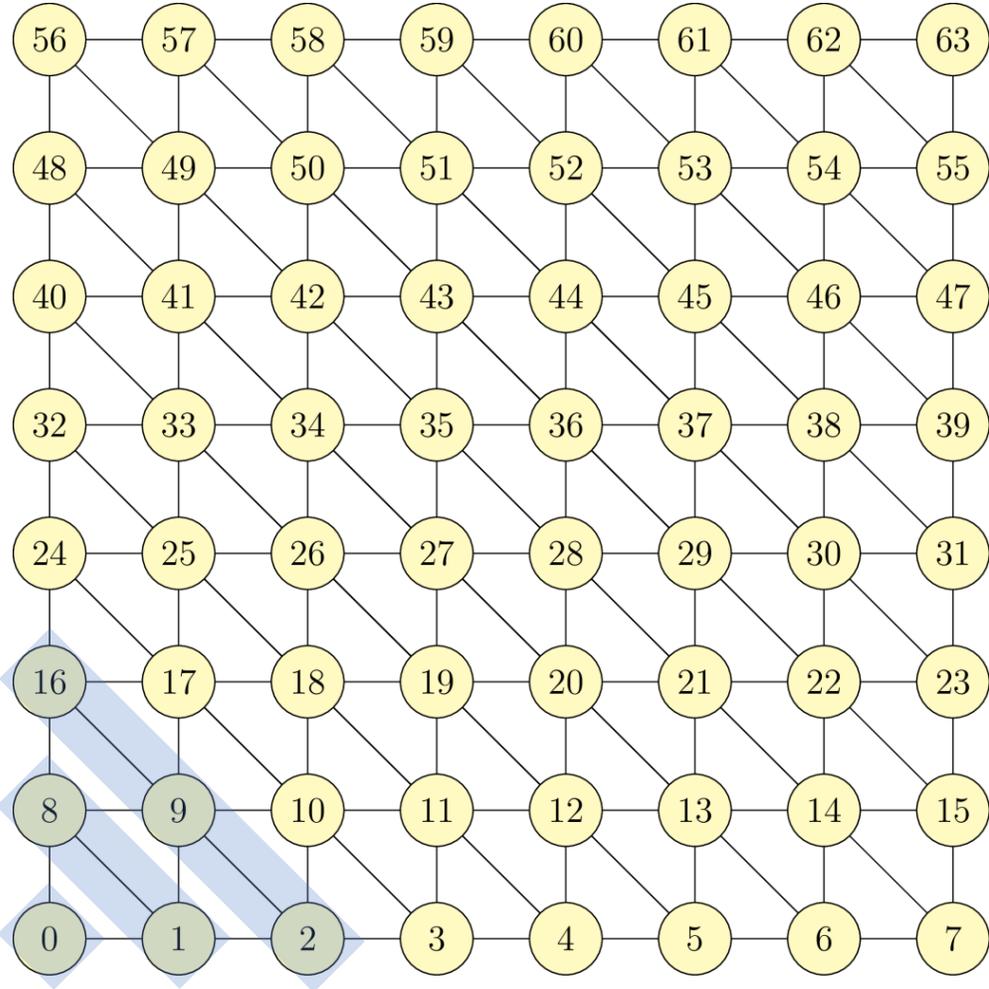


1

2

levels

# RACE



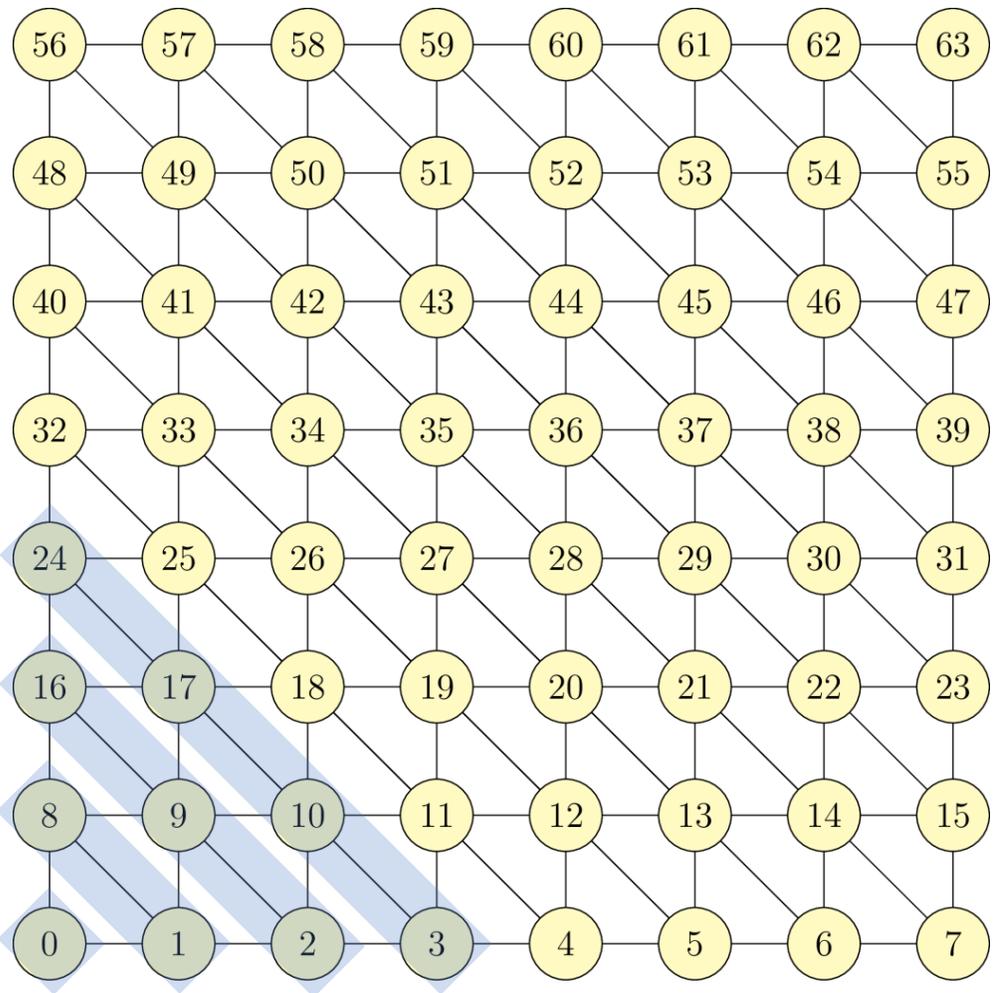
1

2

3

levels

# RACE



1

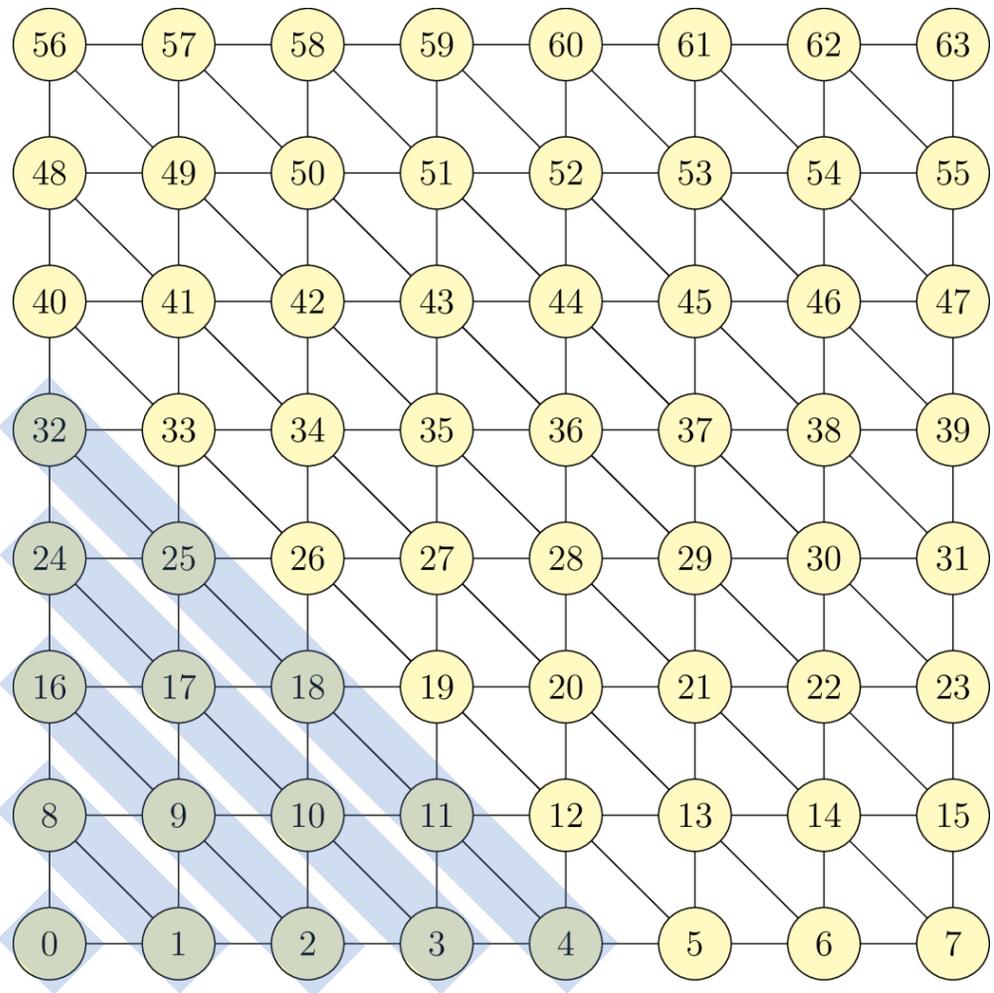
2

3

4

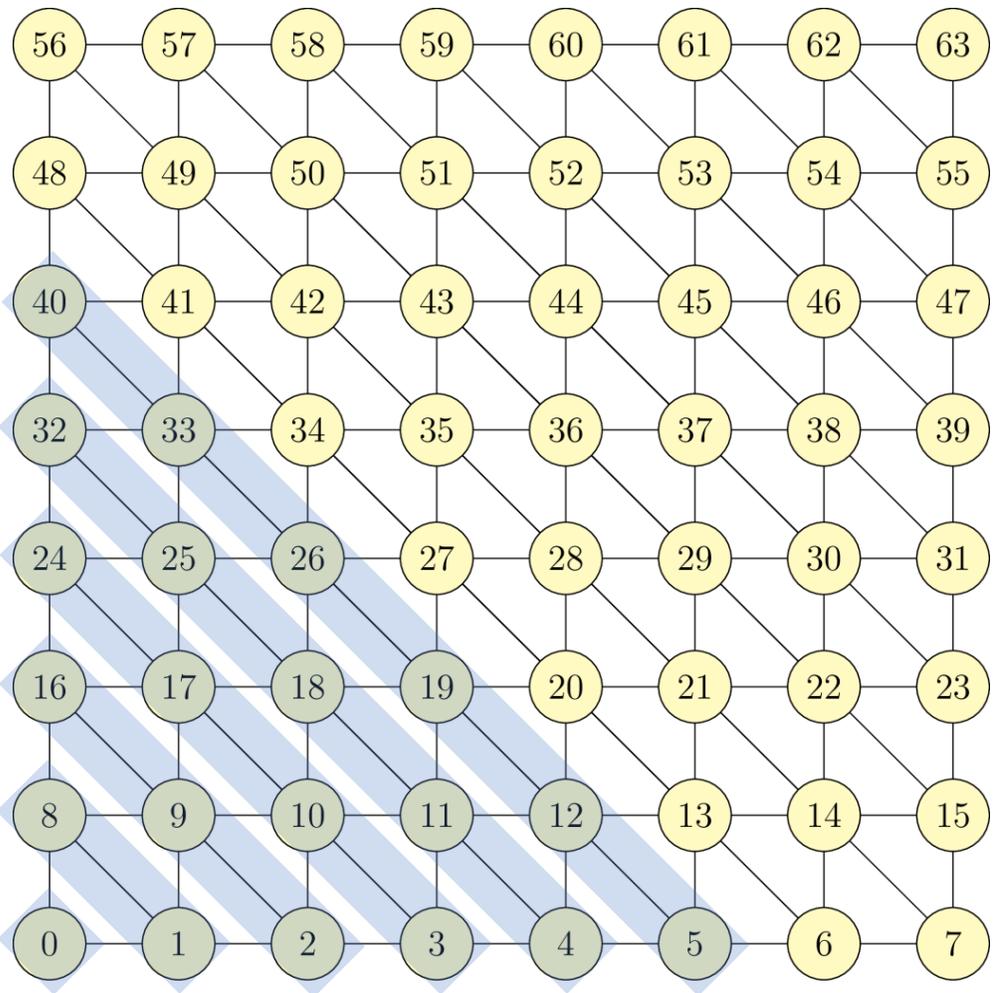
levels

# RACE



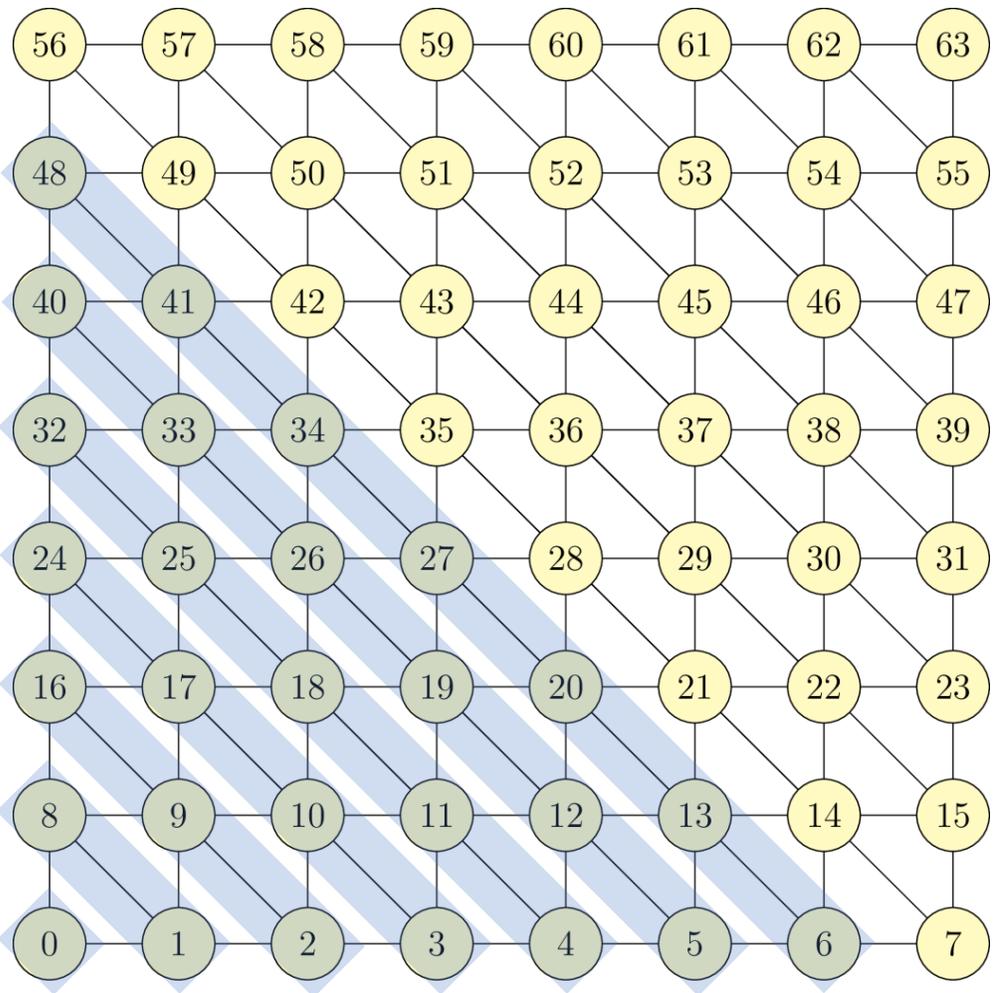
levels

# RACE



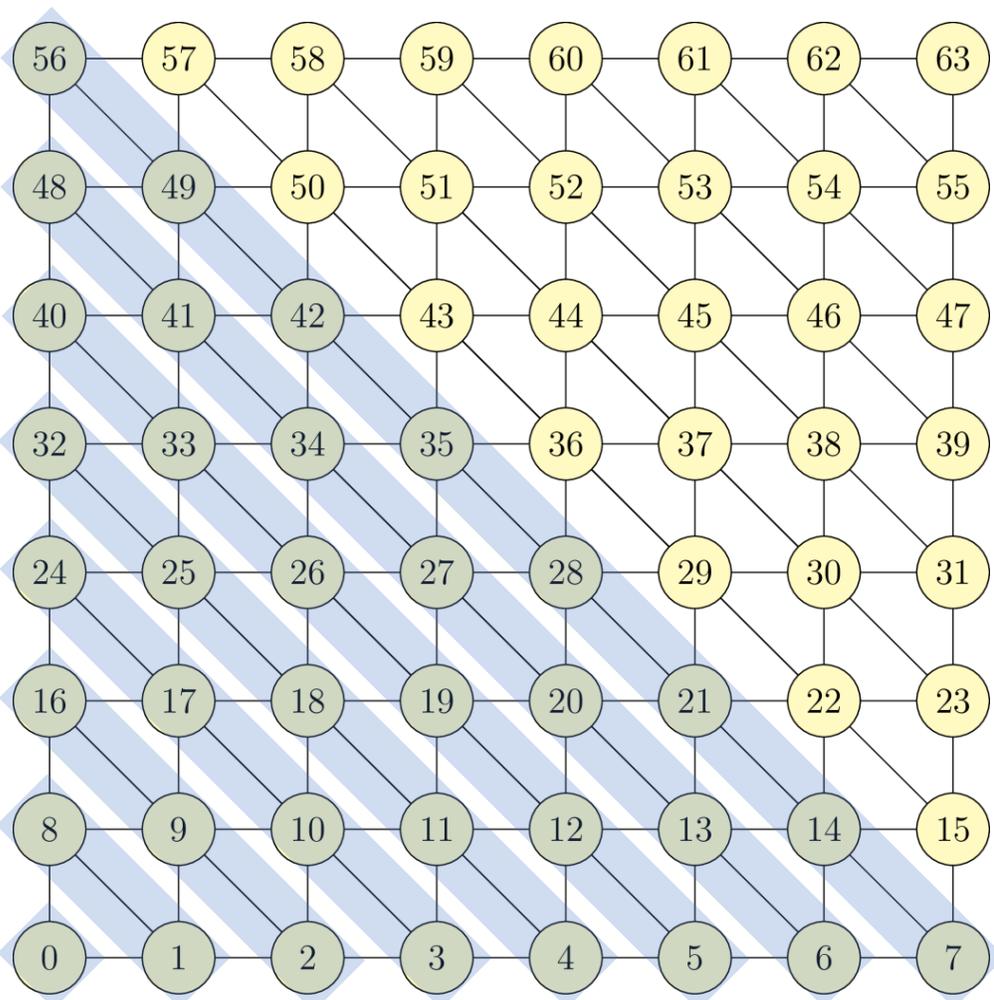
levels

# RACE



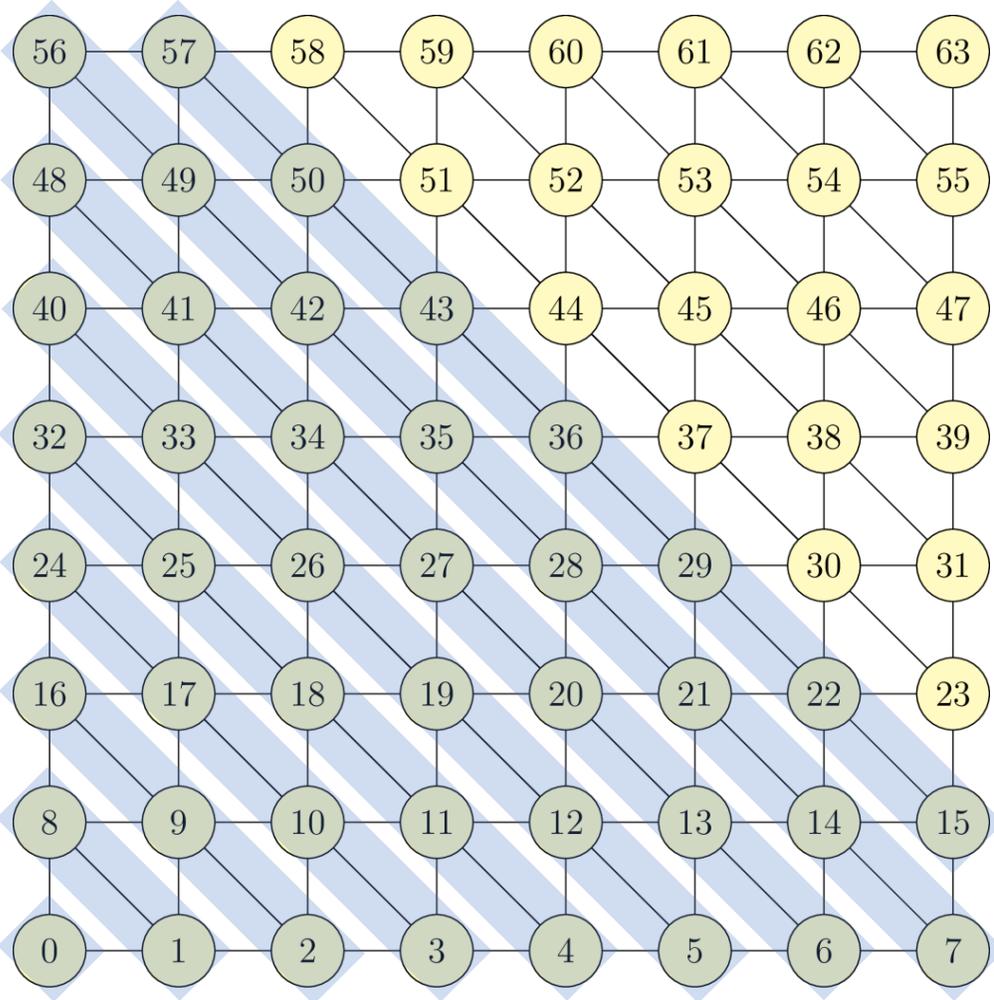
levels

# RACE



levels

# RACE



1

2

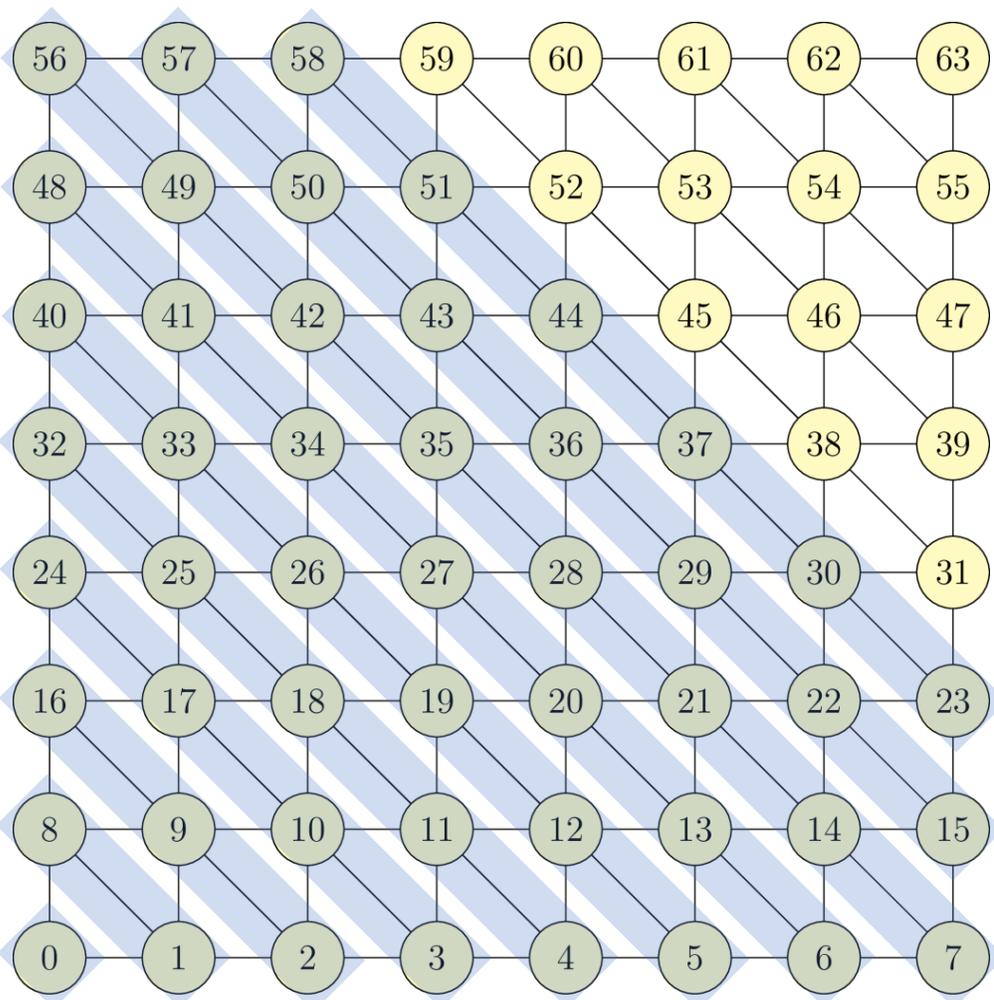
3

4

...

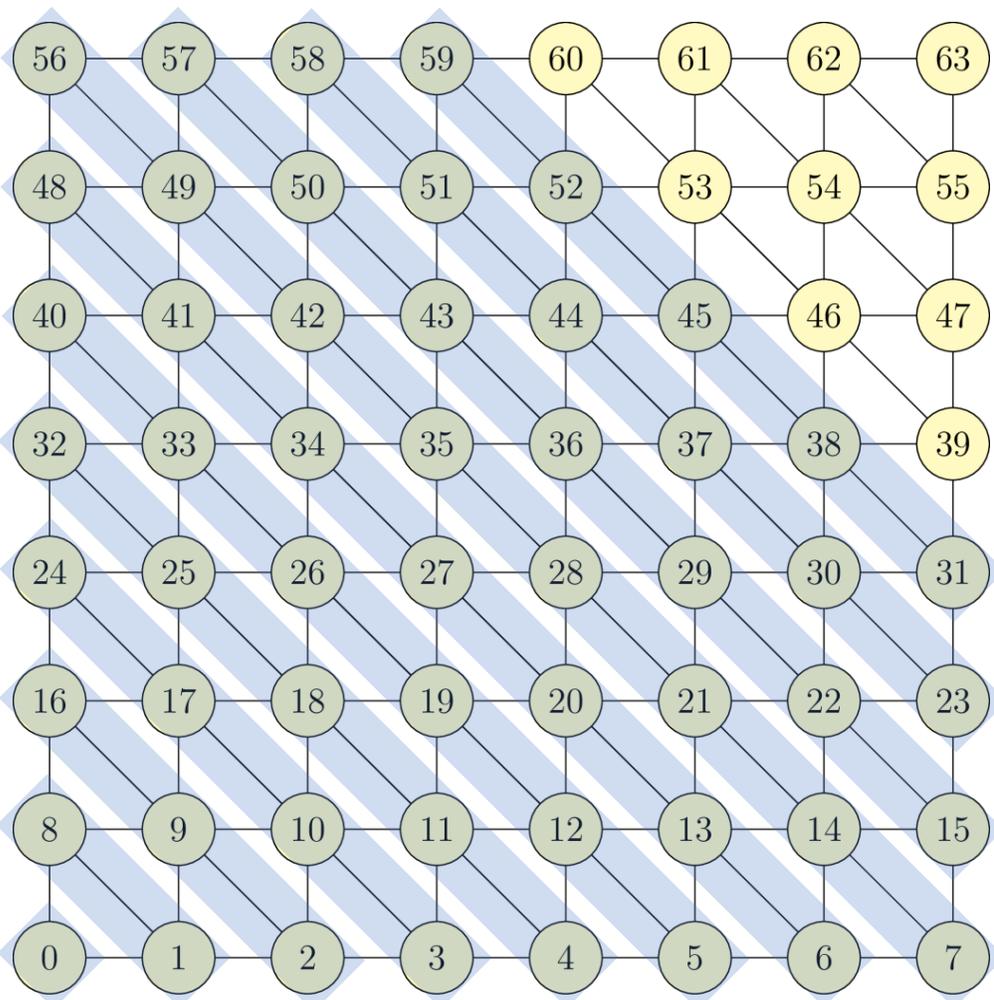
levels

# RACE



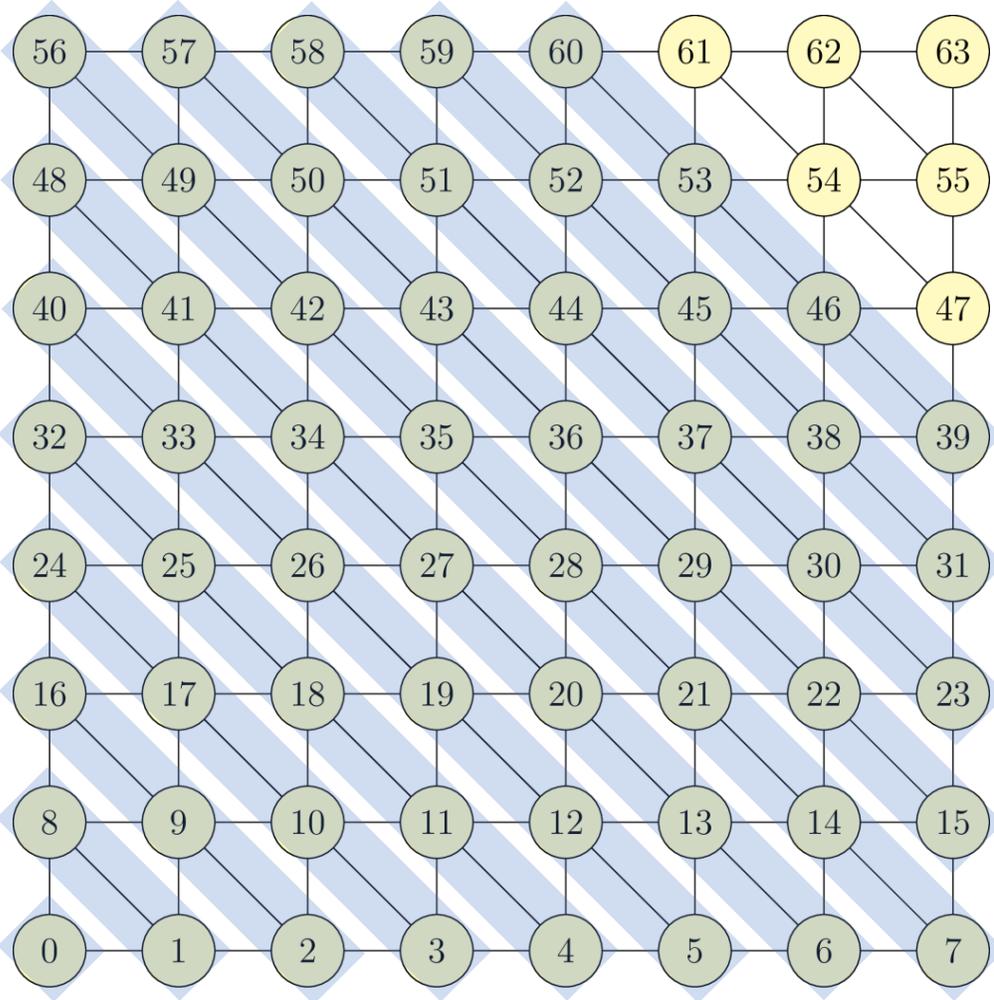
levels

# RACE



levels

# RACE



1

2

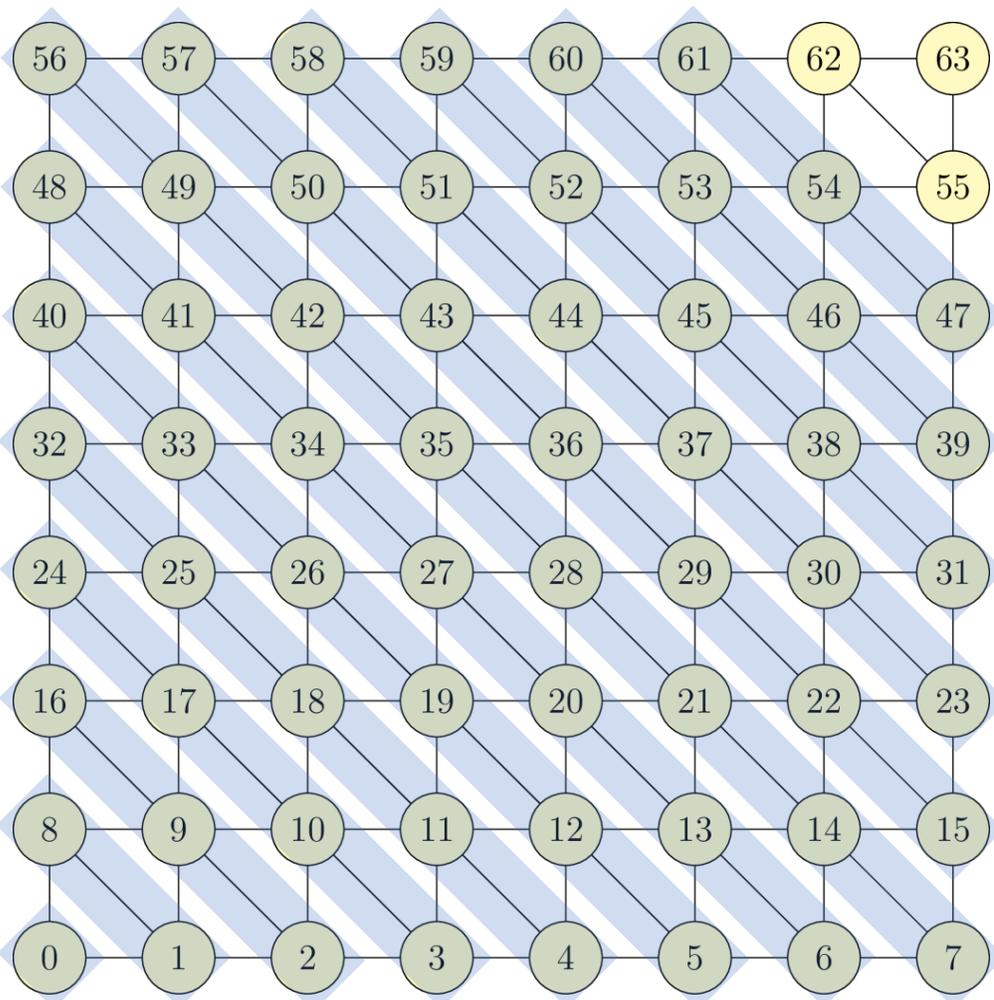
3

4

...

levels

# RACE



1

2

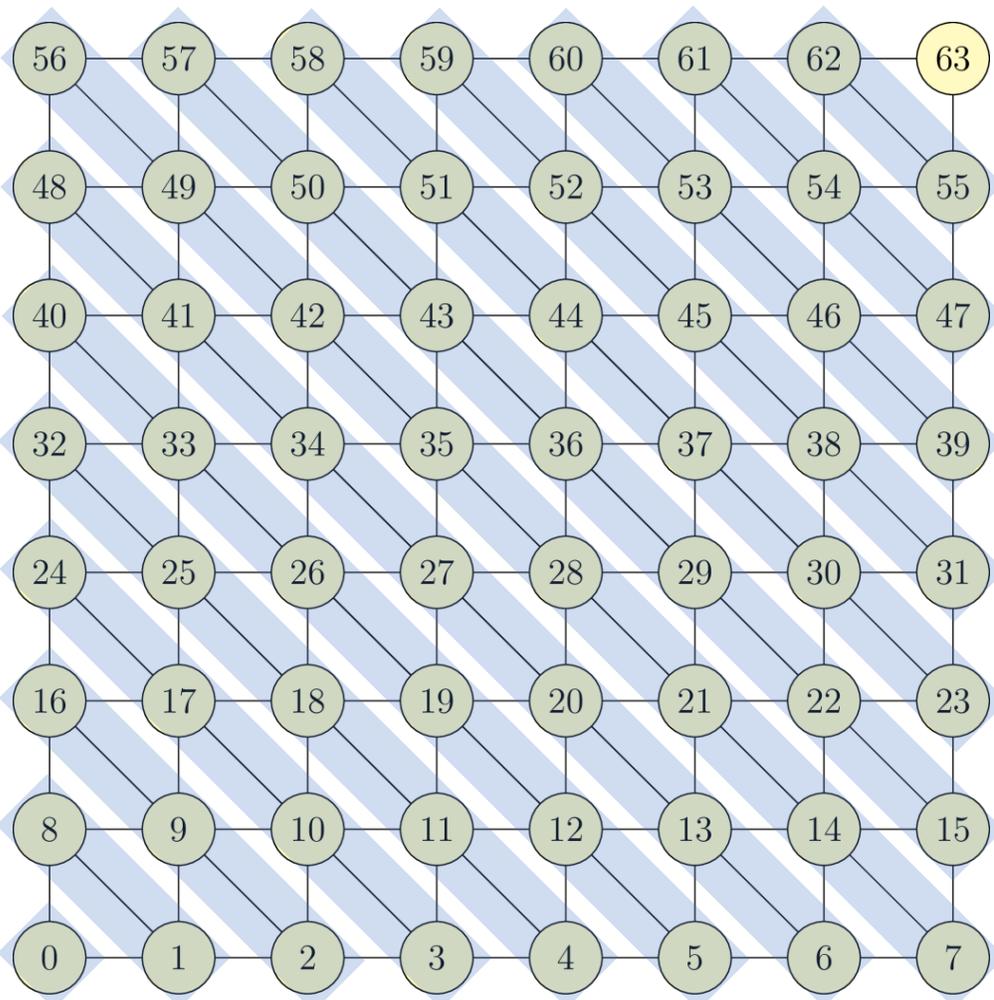
3

4

...

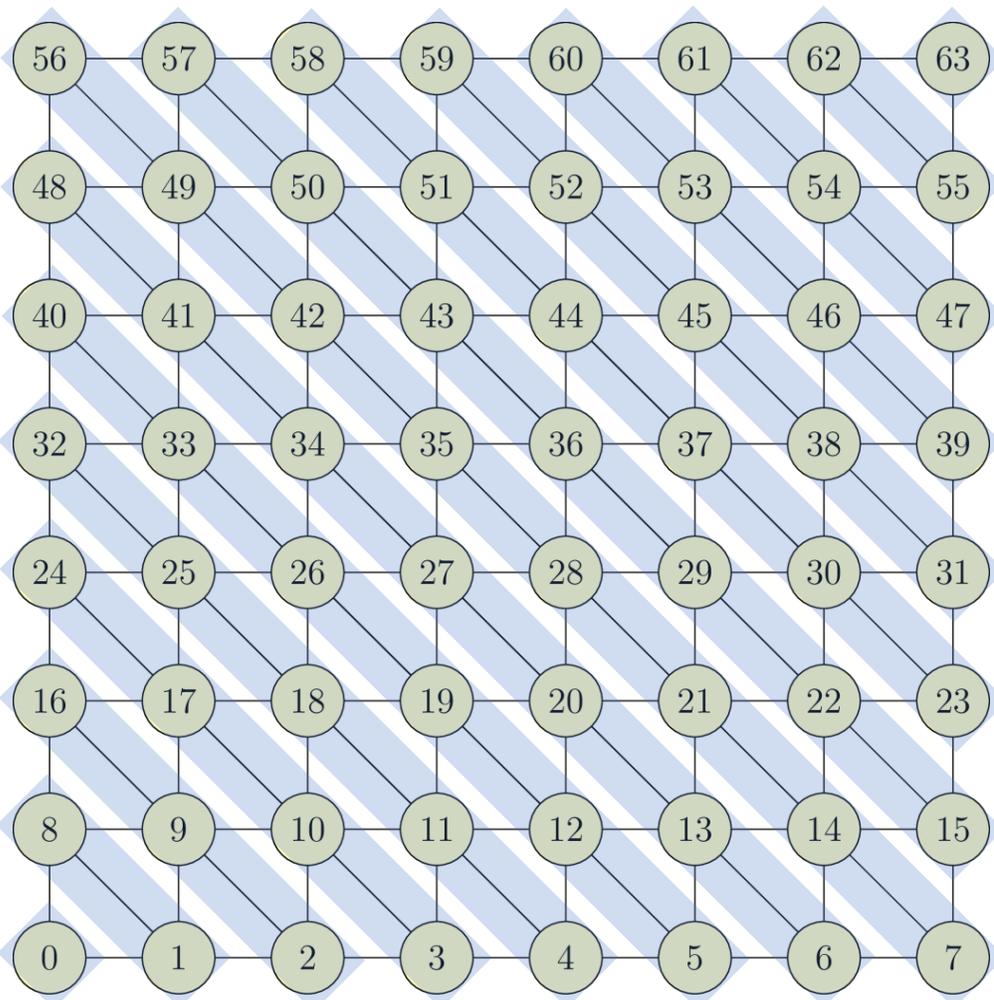
levels

# RACE



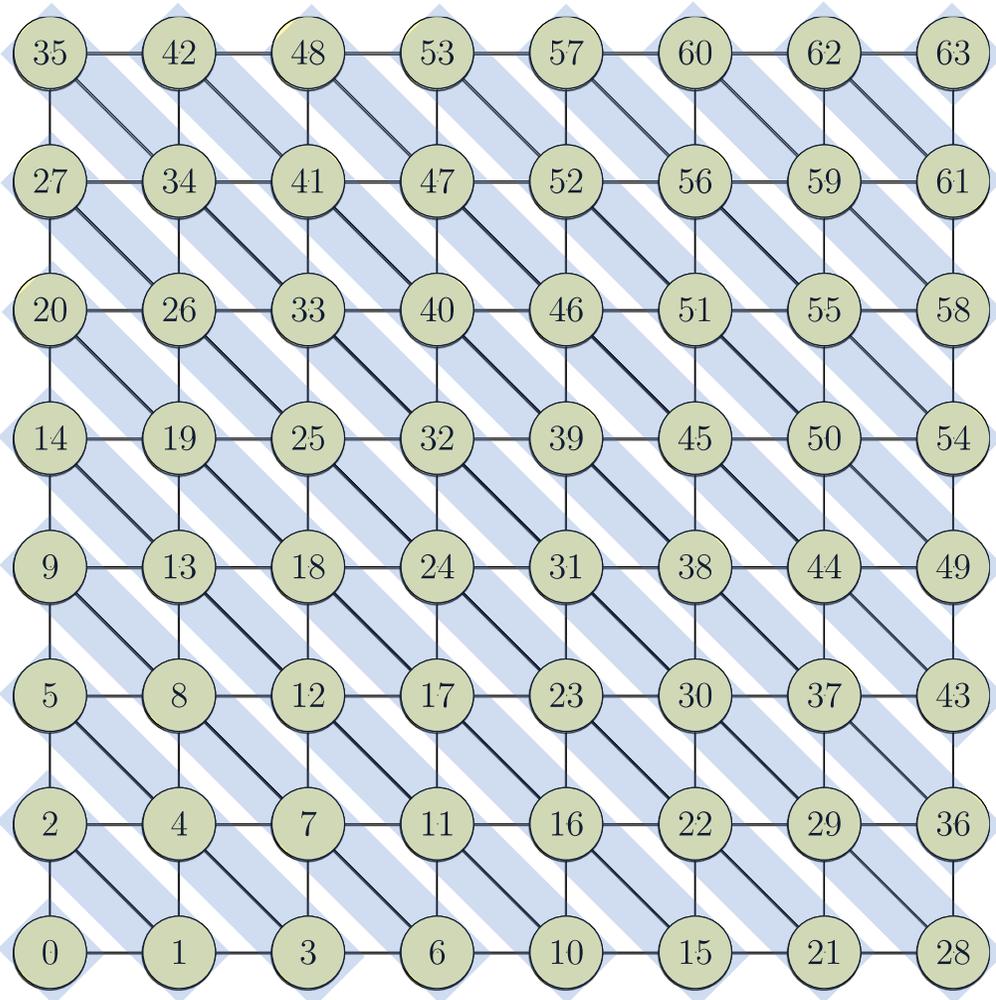
levels

# RACE



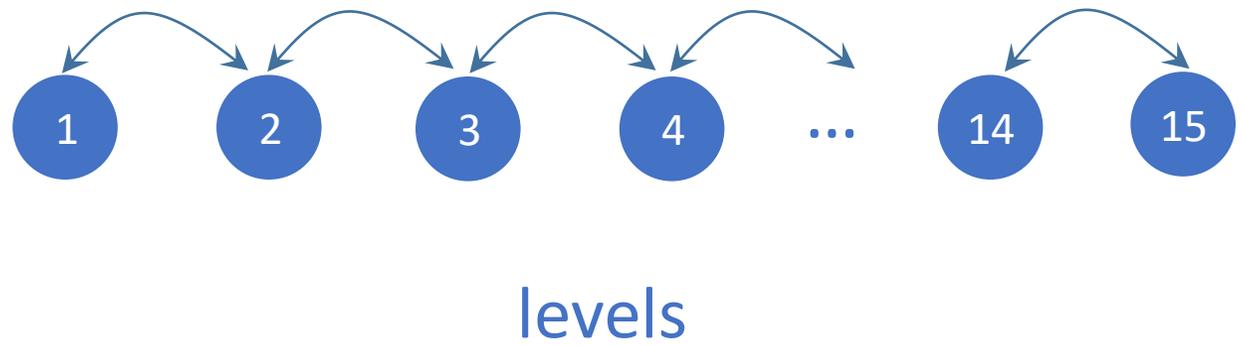
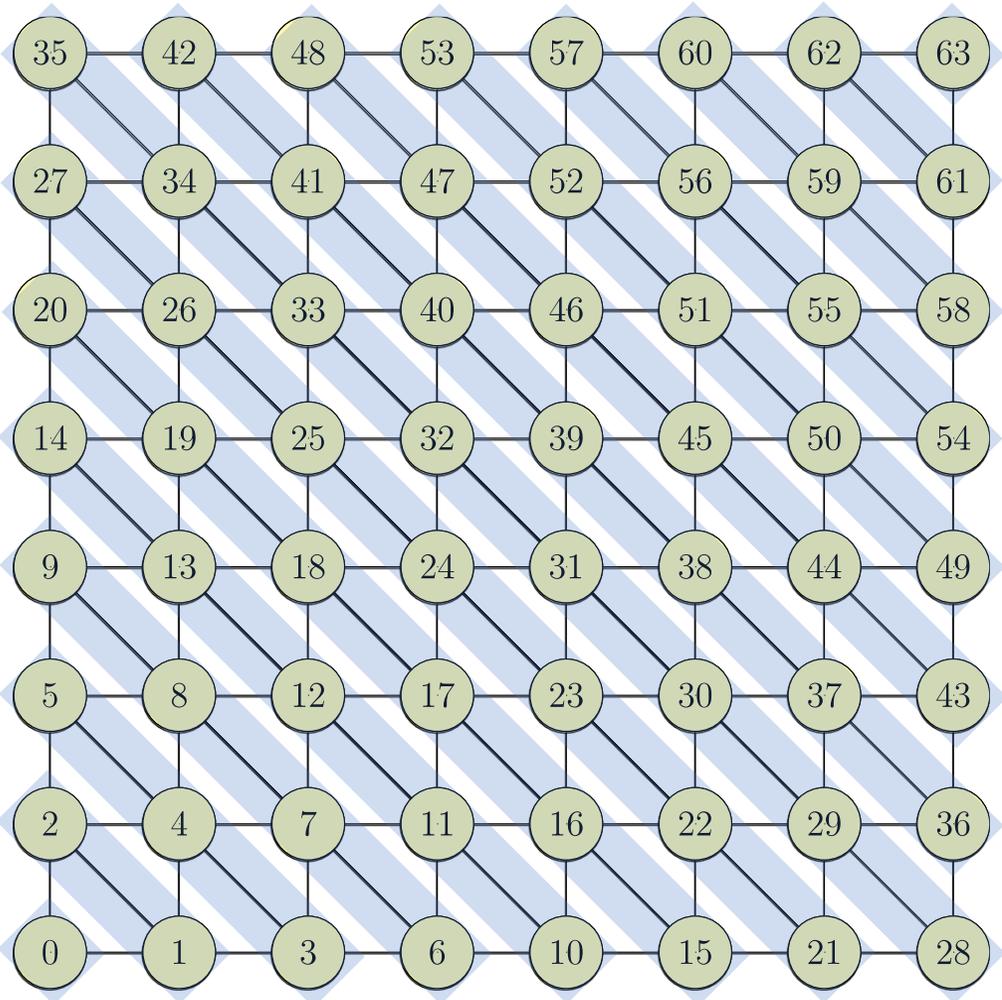
levels

# RACE

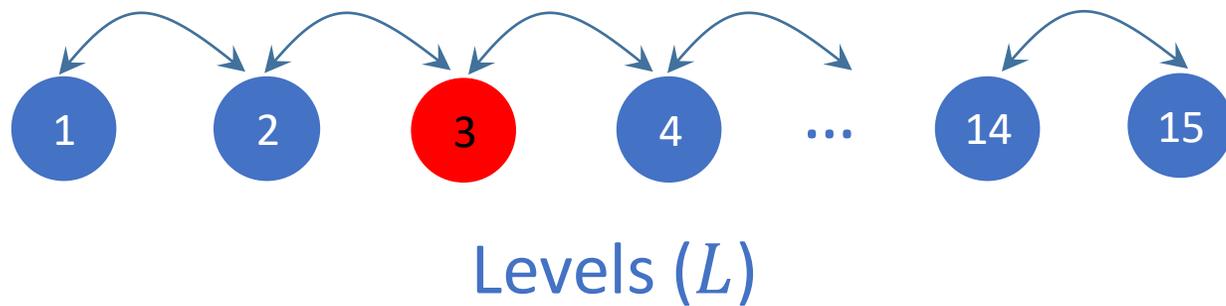
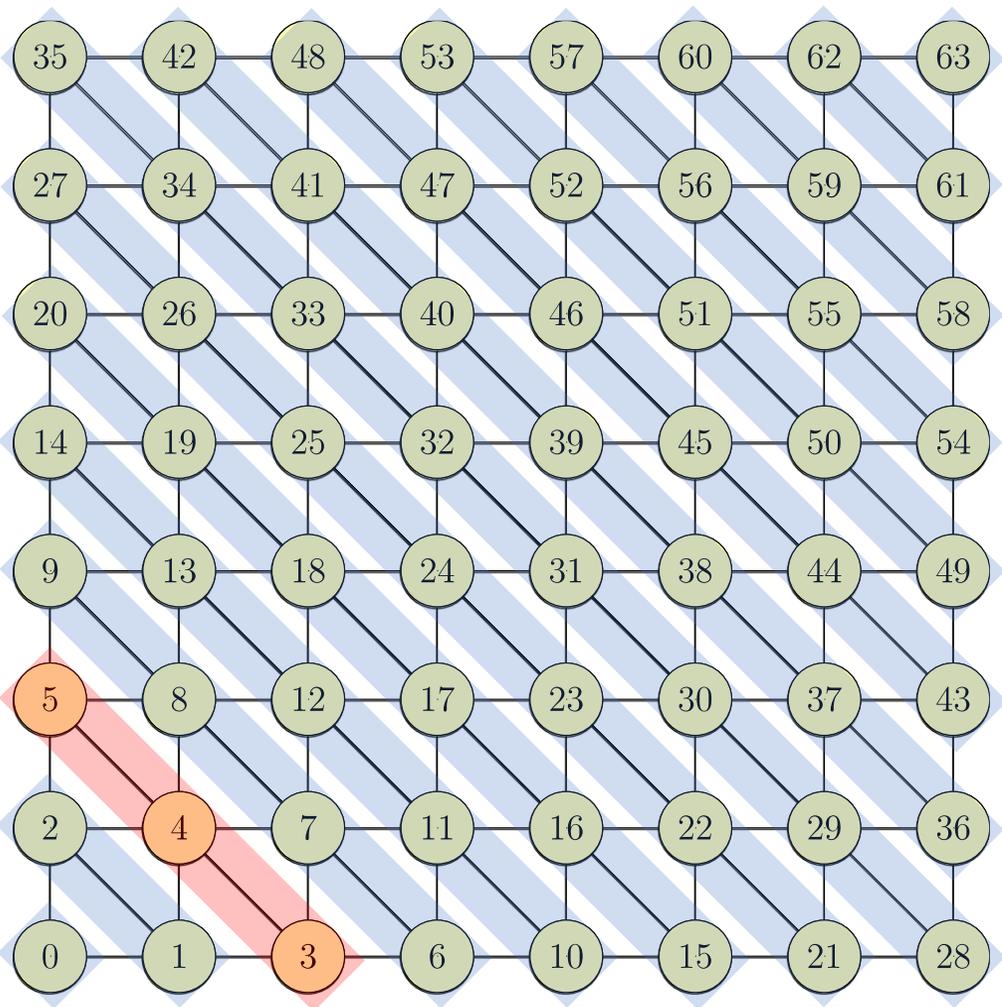


levels

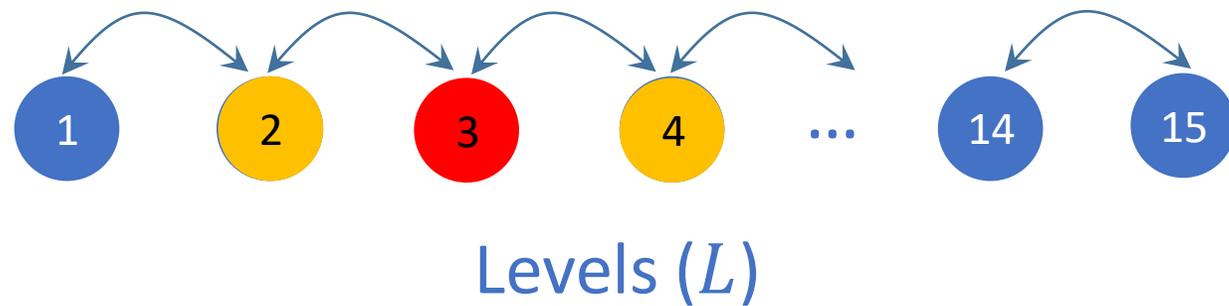
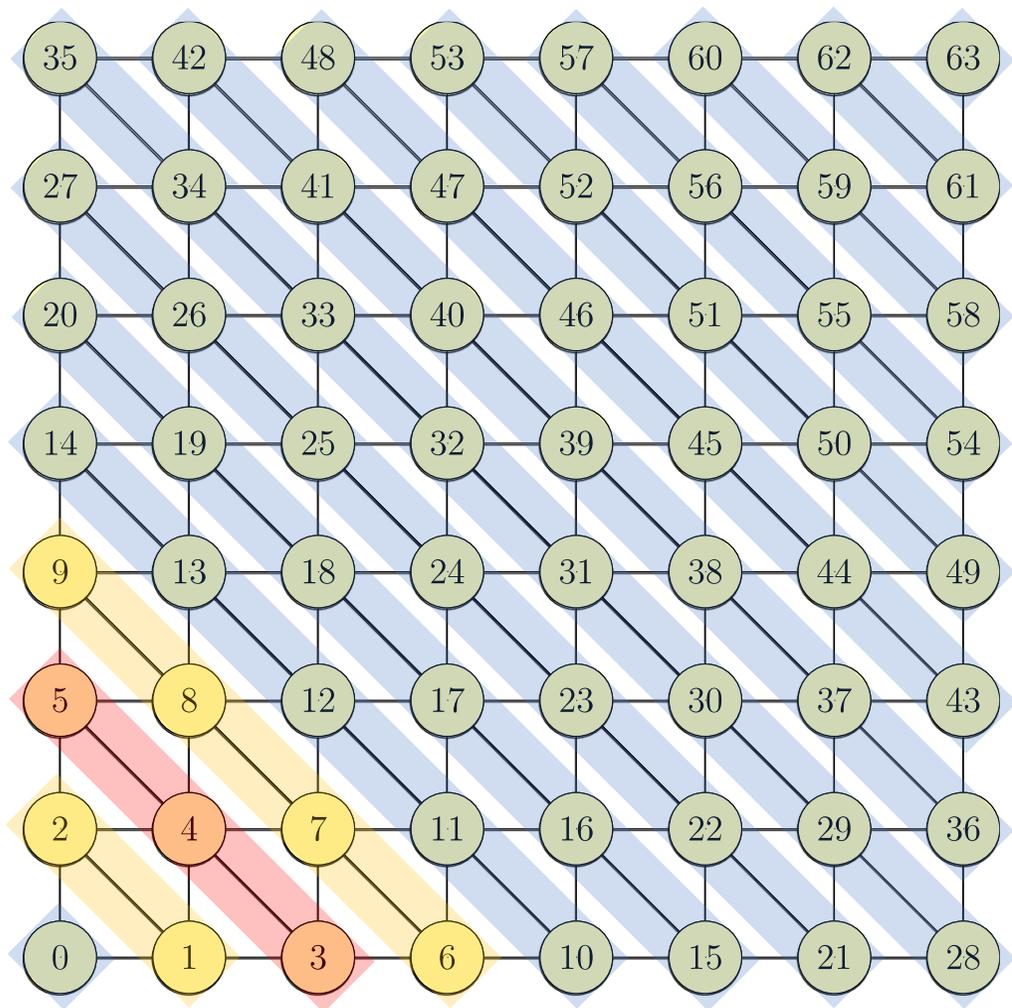
# RACE



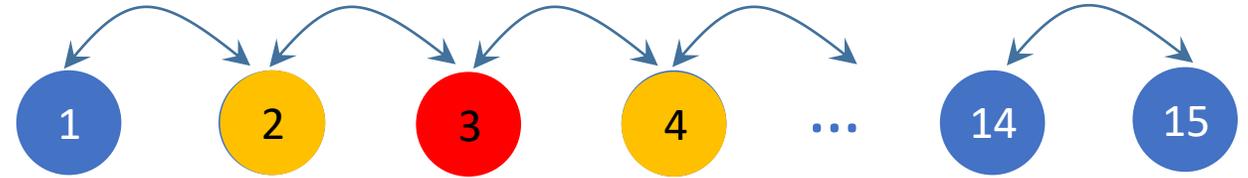
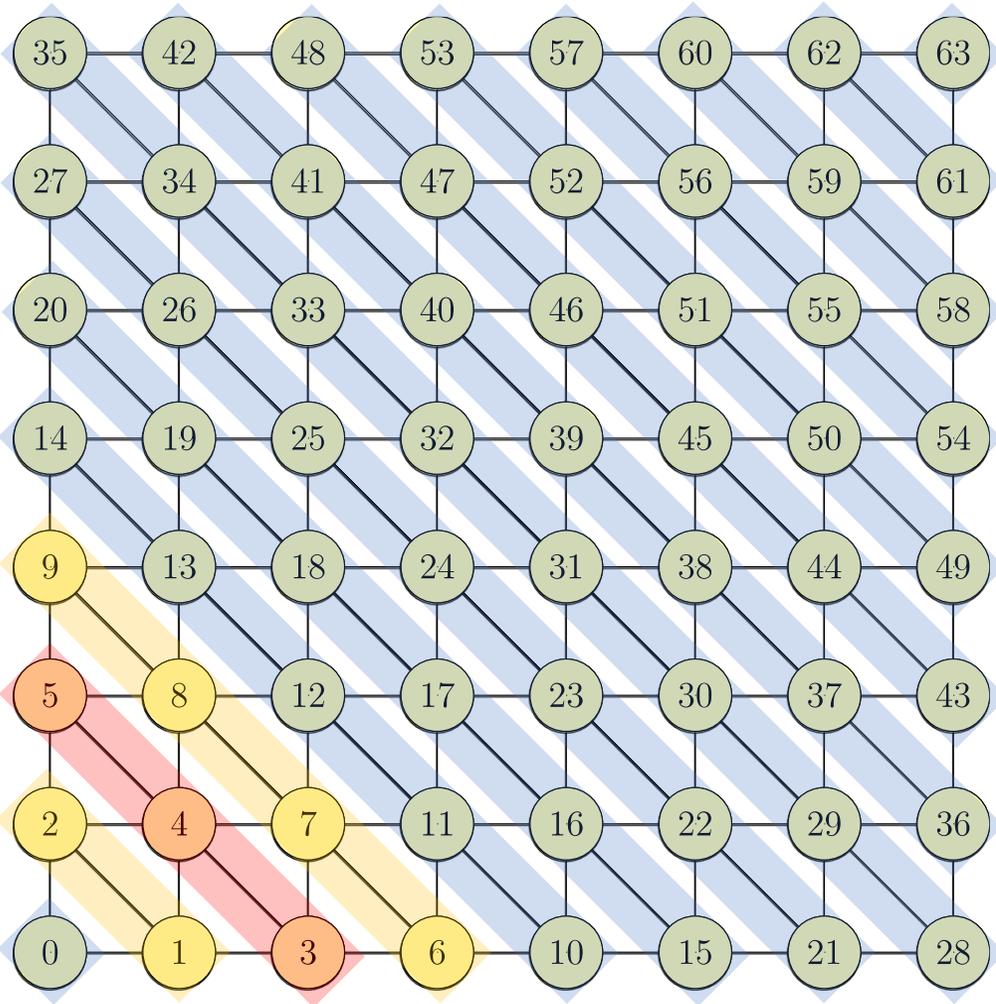
# RACE



# RACE



# RACE

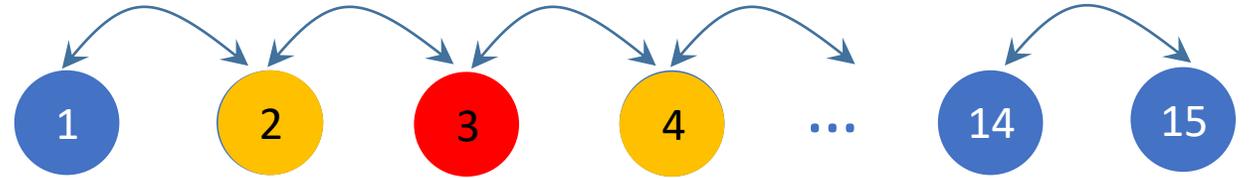
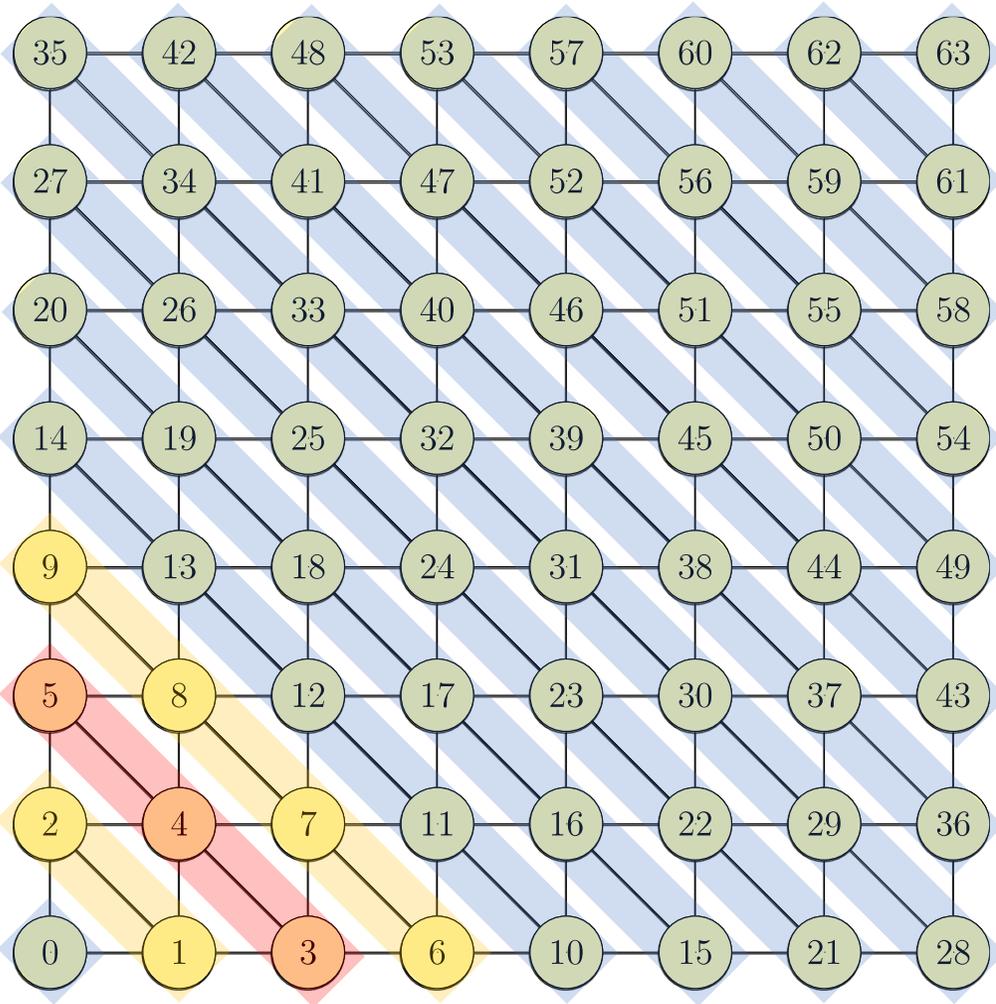


Levels ( $L$ )

**Key property**

$$\mathcal{N}(L(i)) = \{L(i-1), L(i), L(i+1)\}$$

$A^p x$  computations on  $L(i)$  will require  $A^{p-1} x$  to be complete on  $L(i-1), L(i), L(i+1)$



Levels ( $L$ )

**Key property**

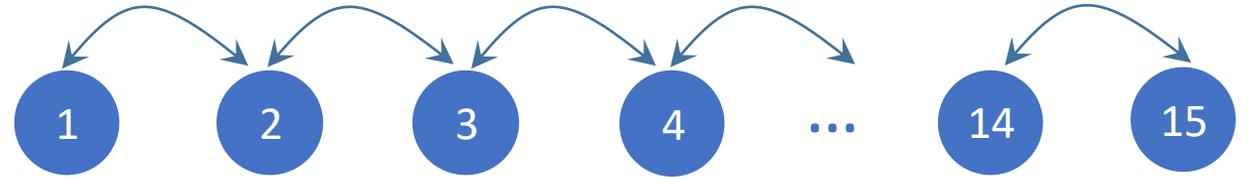
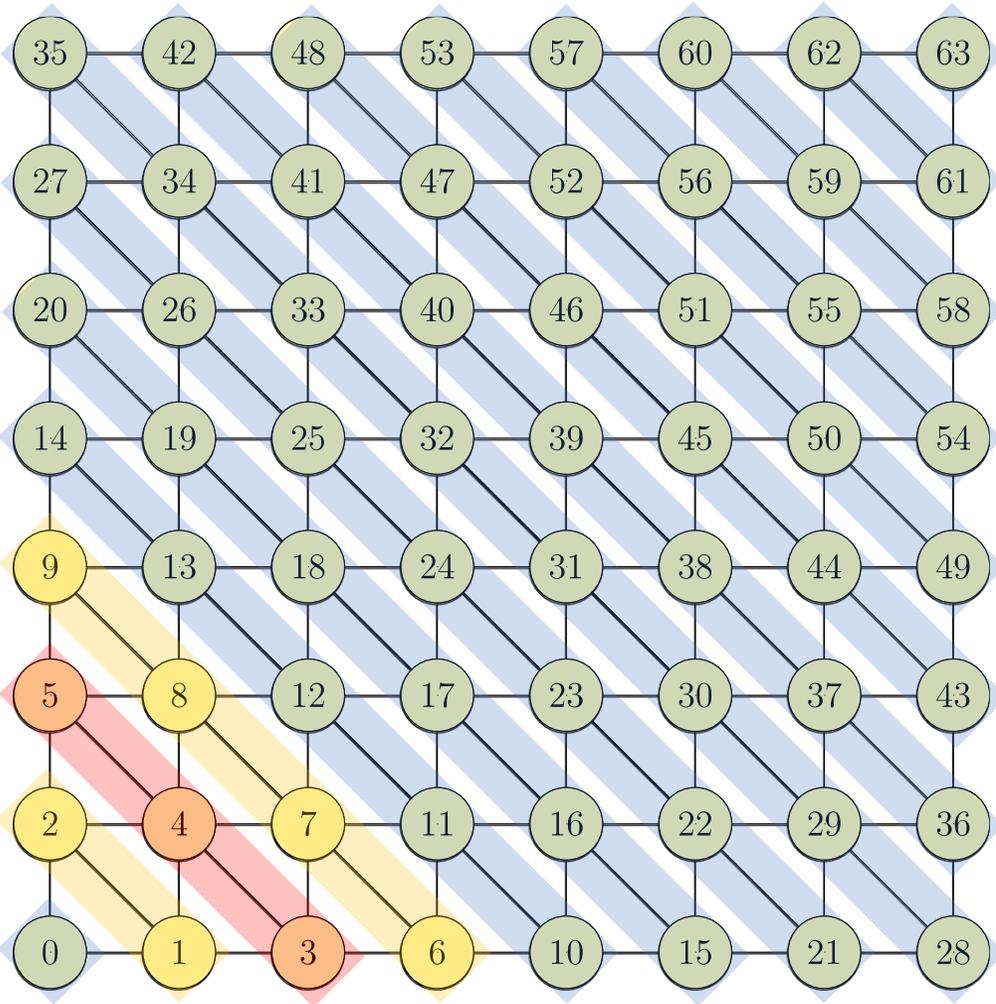
$$\mathcal{N}(L(i)) = \{L(i-1), L(i), L(i+1)\}$$

$A^p x$  computations on  $L(i)$  will require  $A^{p-1} x$  to be complete on  $L(i-1), L(i), L(i+1)$



Neighbors localized → dependencies localized

# RACE



Levels ( $L$ )

**Key property**

$$\mathcal{N}(L(i)) = \{L(i-1), L(i), L(i+1)\}$$

$A^p x$  computations on  $L(i)$  will require  $A^{p-1} x$  to be complete on  $L(i-1), L(i), L(i+1)$



Neighbors localized → dependencies localized

# RACE – Level traversal and matrix powers

```
do k = 1, p
  y(:, k) = SpMV(A, y(:, k-1))
enddo
```

No cache blocking!

Levels

Matrix Powers

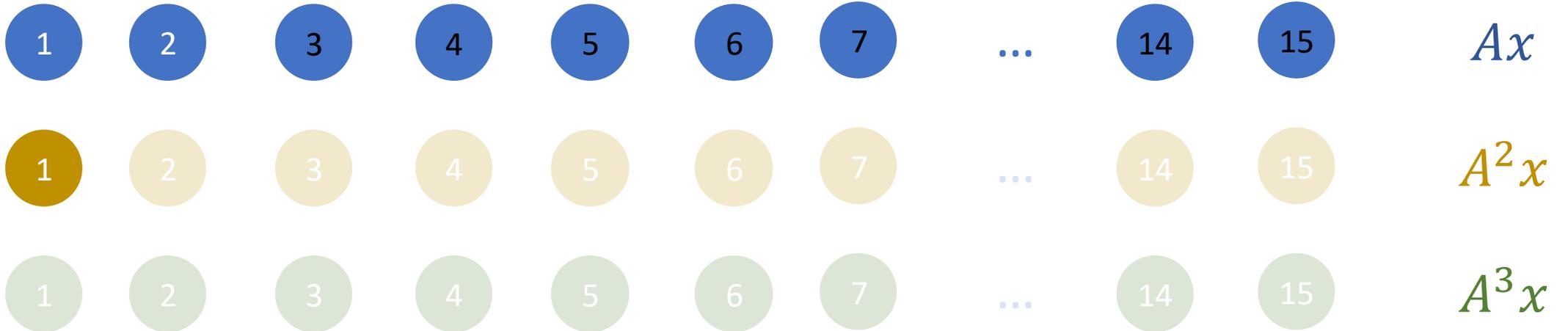


# RACE – Level traversal and matrix powers

```
do k = 1, p
  y(:, k) = SpMV(A, y(:, k-1))
enddo
```

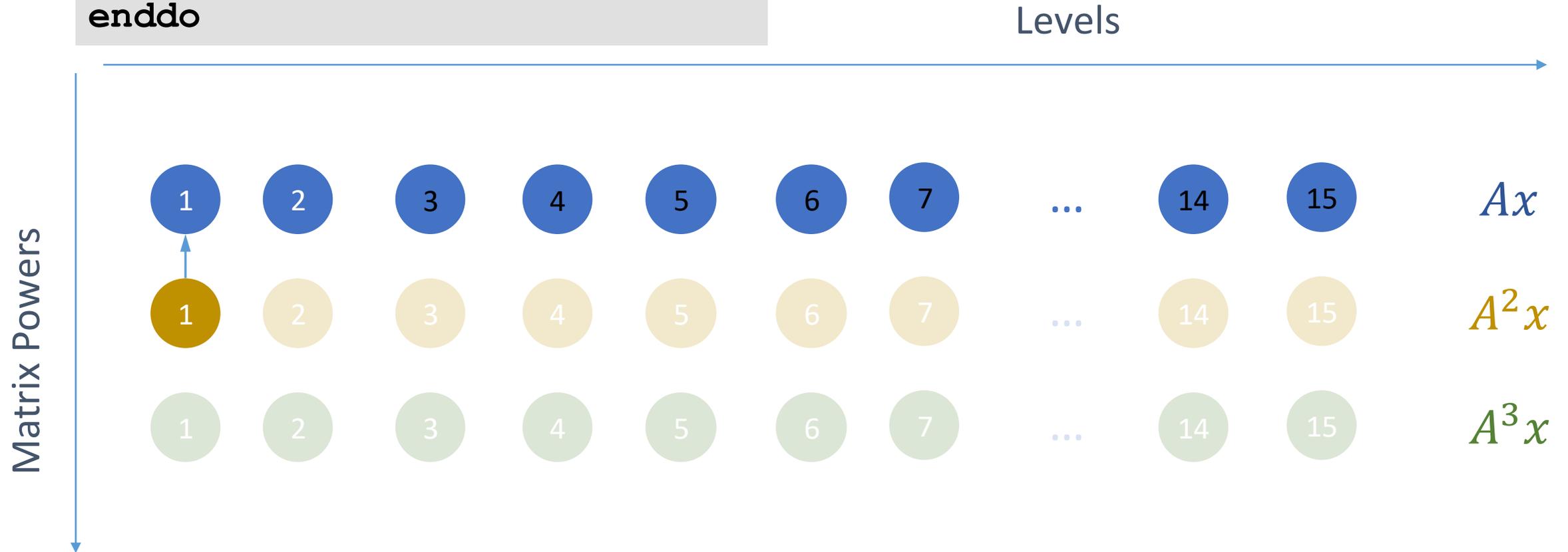
Levels

Matrix Powers



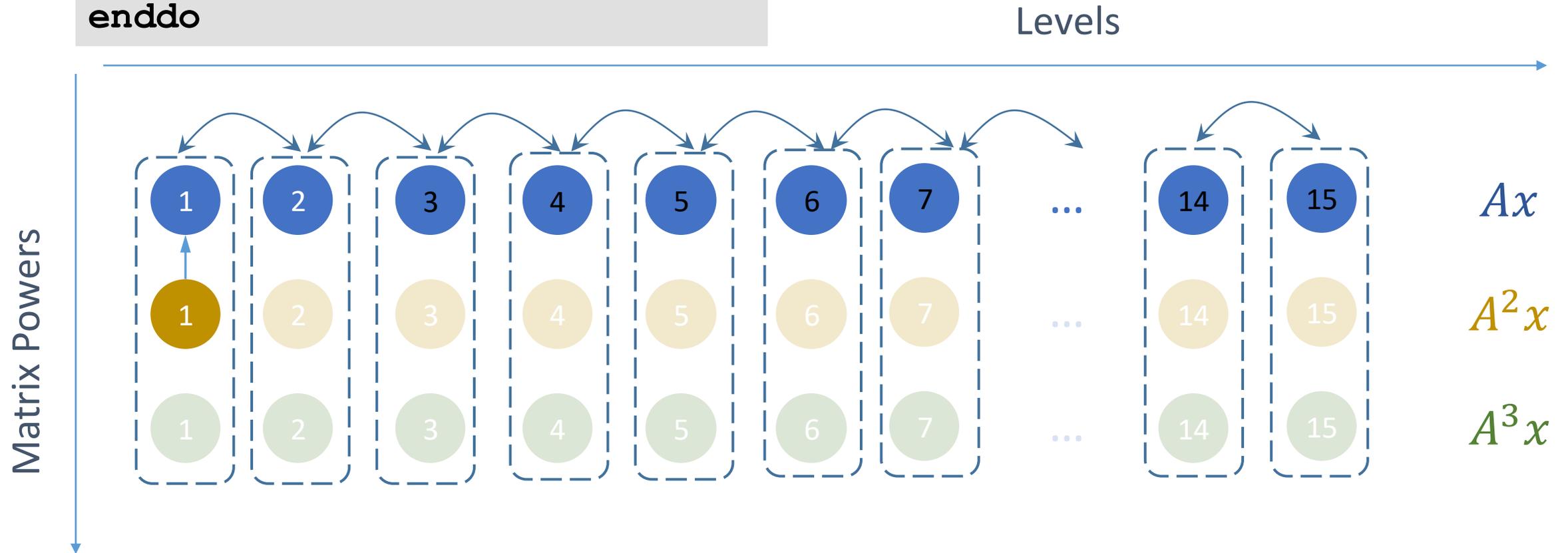
# RACE – Level traversal and matrix powers

```
do k = 1, p
  y(:, k) = SpMV(A, y(:, k-1))
enddo
```



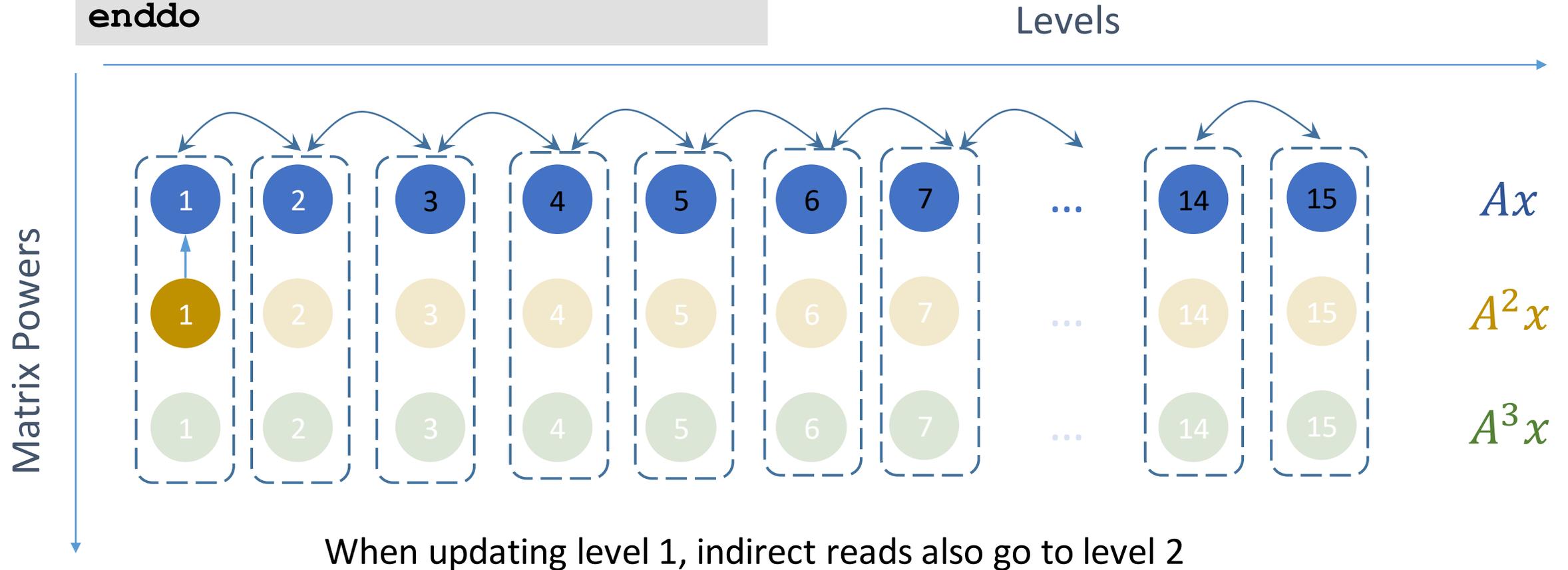
# RACE – Level traversal and matrix powers

```
do k = 1, p
  y(:, k) = SpMV(A, y(:, k-1))
enddo
```



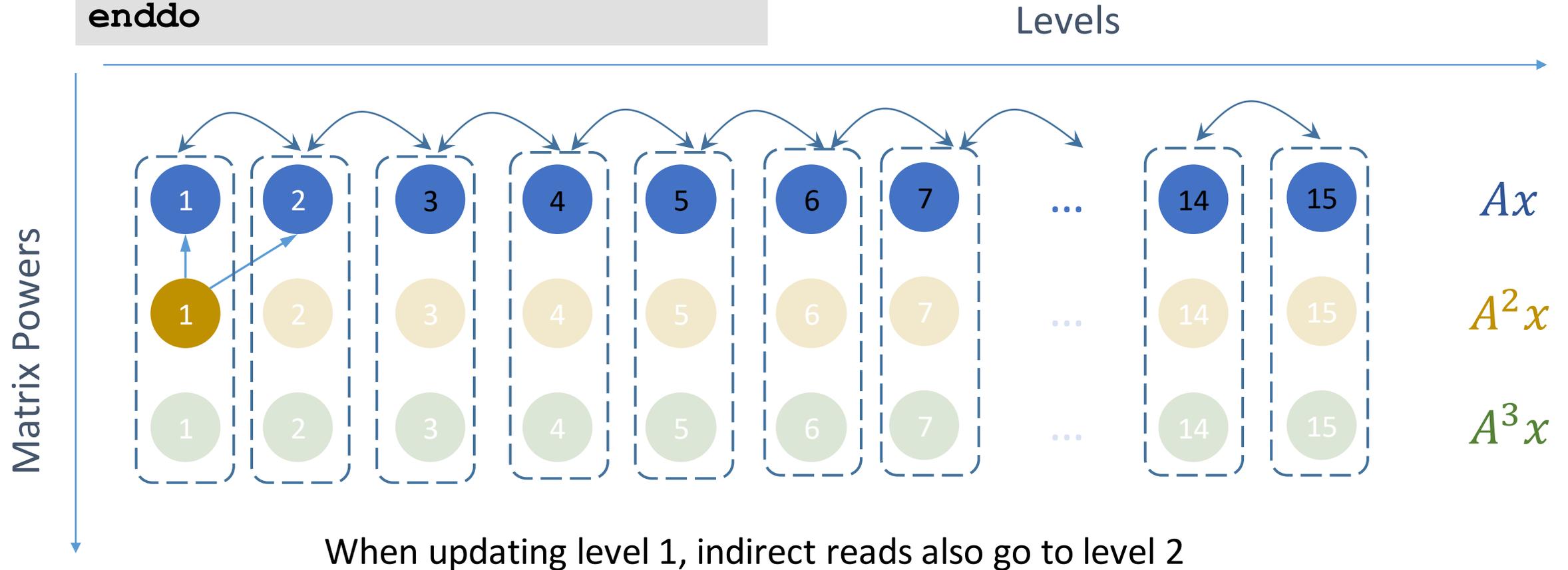
# RACE – Level traversal and matrix powers

```
do k = 1, p
  y(:, k) = SpMV(A, y(:, k-1))
enddo
```



# RACE – Level traversal and matrix powers

```
do k = 1, p
  y(:, k) = SpMV(A, y(:, k-1))
enddo
```

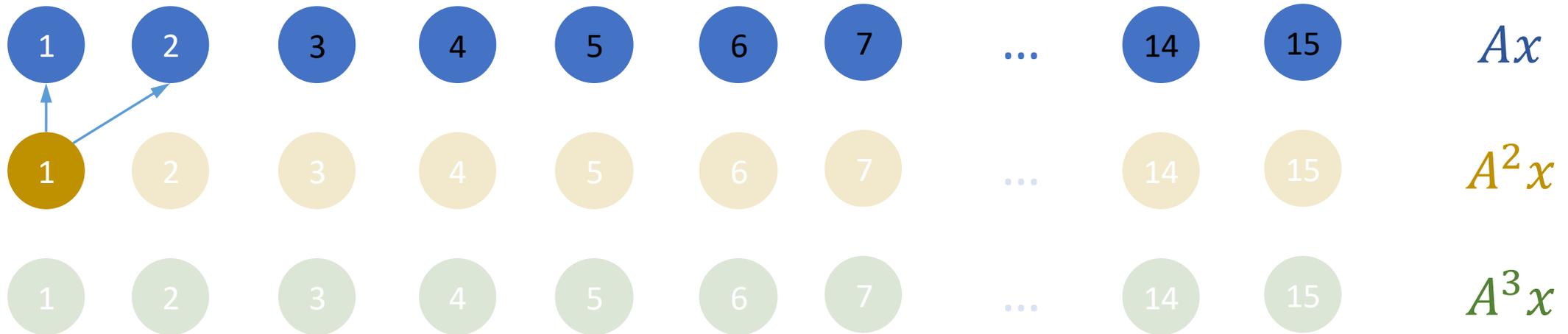


# RACE – Level traversal and matrix powers

```
do k = 1, p
  y(:, k) = SpMV(A, y(:, k-1))
enddo
```

Levels

Matrix Powers



When updating level 1, indirect reads also go to level 2

# RACE – Level traversal and matrix powers

```
do k = 1, p
  y(:, k) = SpMV(A, y(:, k-1))
enddo
```

Levels

Matrix Powers



When updating level 1, indirect reads also go to level 2

# RACE – Level traversal and matrix powers

```
do k = 1, p
  y(:, k) = SpMV(A, y(:, k-1))
enddo
```

Levels

Do not pollute the cache → reuse all loaded elements



When updating level 1, indirect reads also go to level 2

# RACE – Level traversal and matrix powers

```
do k = 1, p
  y(:, k) = SpMV(A, y(:, k-1))
enddo
```

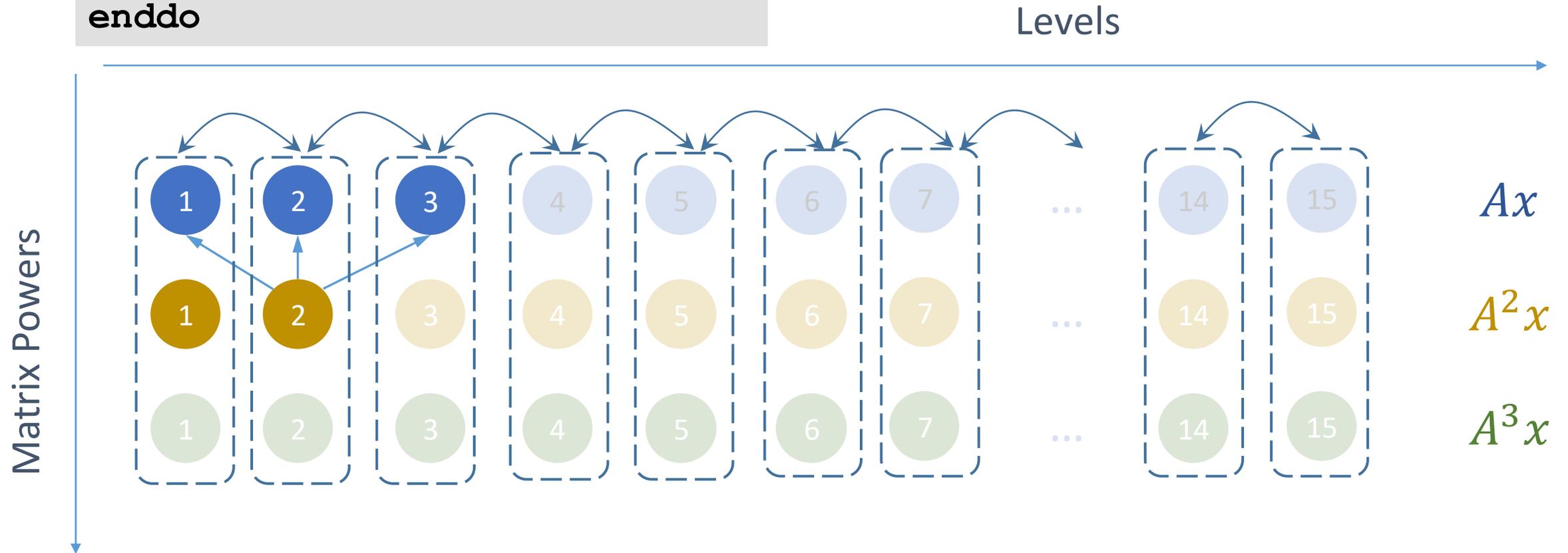
Levels

Matrix Powers



# RACE – Level traversal and matrix powers

```
do k = 1, p
  y(:, k) = SpMV(A, y(:, k-1))
enddo
```

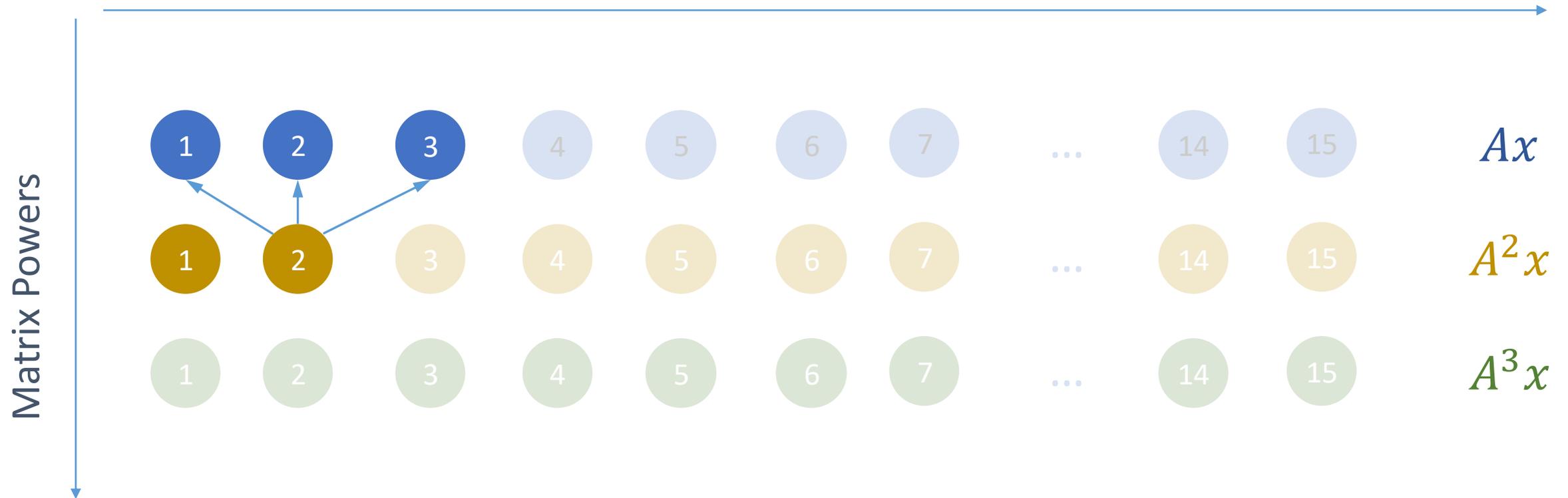


When updating level 2, indirect reads also go to levels 1 and 3

# RACE – Level traversal and matrix powers

```
do k = 1, p
  y(:, k) = SpMV(A, y(:, k-1))
enddo
```

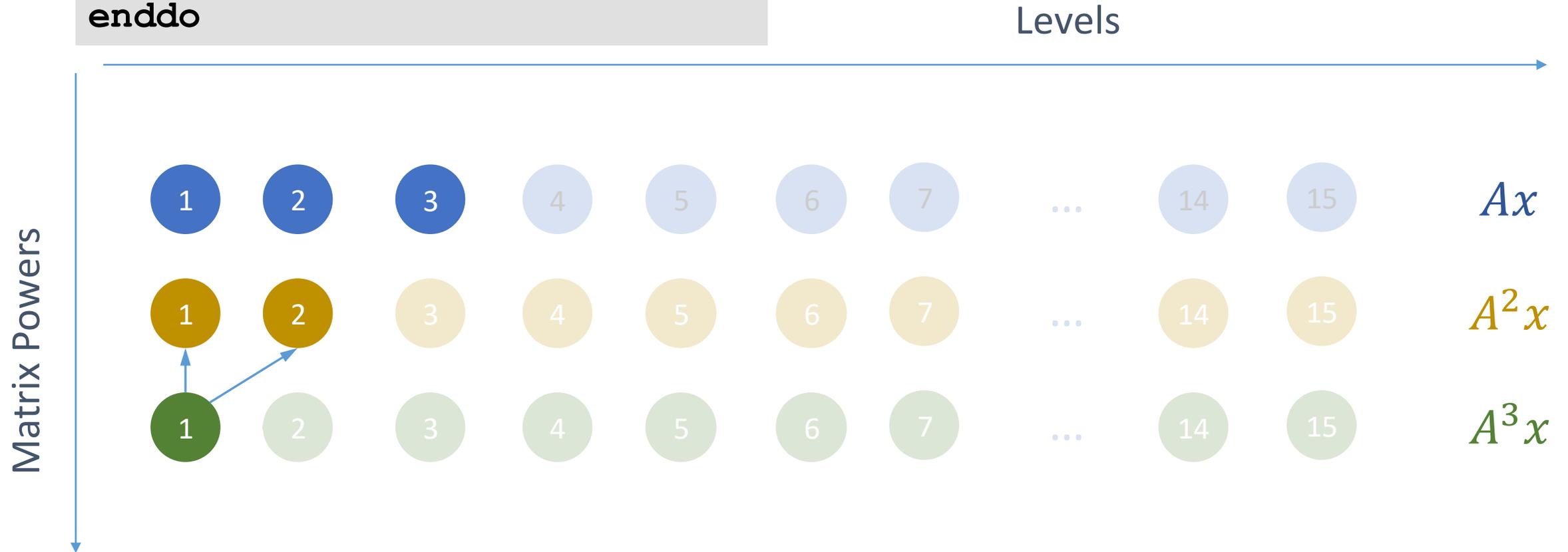
Levels



When updating level 2, indirect reads also go to levels 1 and 3

# RACE – Level traversal and matrix powers

```
do k = 1, p
  y(:, k) = SpMV(A, y(:, k-1))
enddo
```

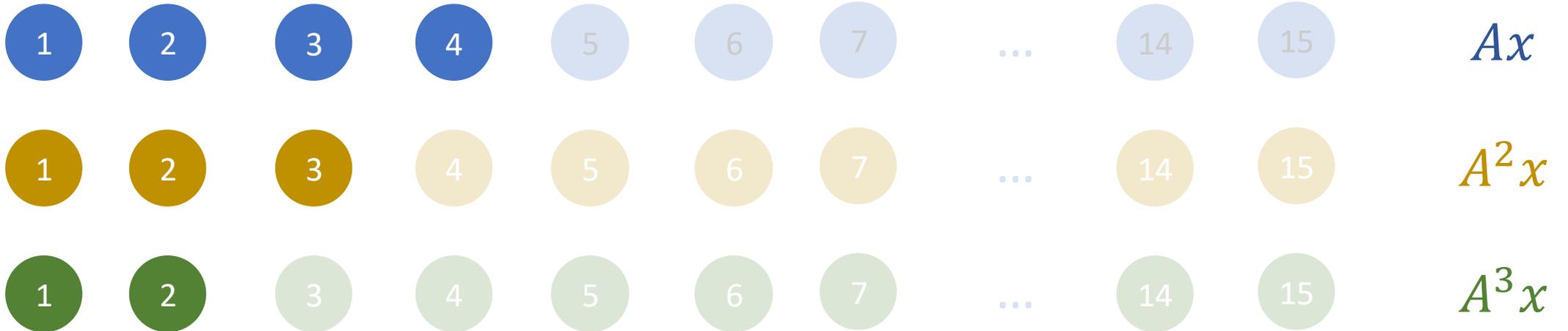


# RACE – Level traversal and matrix powers

```
do k = 1, p
  y(:, k) = SpMV(A, y(:, k-1))
enddo
```

Levels

Matrix Powers



# RACE – Level traversal and matrix powers

```
do k = 1, p
  y(:, k) = SpMV(A, y(:, k-1))
enddo
```

Levels

Matrix Powers



# RACE: MPK Pseudocode

```
do d in 1:L+p-1
  p_start = max(1, d-(L-1))
  p_end = min(d, p)
  do k in p_start:p_end
    l=(d-k+1)
     $y(:, k) = \text{SpMV}(A(j, :), y(:, k-1), \text{level\_ptr}[l]:\text{level\_ptr}[l+1])$ 
  enddo
enddo
```



# RACE: MPK Pseudocode

```
do d in 1:L+p-1
  p_start = max(1, d-(L-1))
  p_end = min(d, p)
  do k in p_start:p_end
    l=(d-k+1)
    y(:, k) = SpMV(A(j,:), y(:,k-1), level_ptr[l]:level_ptr[l+1])
  enddo
enddo
```

← Traverse along diagonal



# RACE: MPK Pseudocode

```
do d in 1:L+p-1
  p_start = max(1, d-(L-1))
  p_end = min(d, p)
  do k in p_start:p_end
    l=(d-k+1)
    y(:, k) = SpMV(A(j,:), y(:,k-1), level_ptr[l]:level_ptr[l+1])
  enddo
enddo
```

← Traverse along diagonal



# RACE: MPK Pseudocode

```
do d in 1:L+p-1
  p_start = max(1, d-(L-1))
  p_end = min(d, p)
  do k in p_start:p_end
    l=(d-k+1)
    y(:, k) = SpMV(A(j,:), y(:,k-1), level_ptr[l]:level_ptr[l+1])
  enddo
enddo
```

← Traverse along diagonal



# RACE: MPK Pseudocode

```
do d in 1:L+p-1
  p_start = max(1, d-(L-1))
  p_end = min(d, p)
  do k in p_start:p_end
    l=(d-k+1)
    y(:, k) = SpMV(A(j,:), y(:,k-1), level_ptr[l]:level_ptr[l+1])
  enddo
enddo
```

← Traverse along diagonal



# RACE: MPK Pseudocode

```
do d in 1:L+p-1
  p_start = max(1, d-(L-1))
  p_end = min(d, p)
  do k in p_start:p_end
    l=(d-k+1)
    y(:, k) = SpMV(A(j,:), y(:,k-1), level_ptr[l]:level_ptr[l+1])
  enddo
enddo
```

← Traverse along diagonal



# RACE: MPK Pseudocode

```
do d in 1:L+p-1
  p_start = max(1, d-(L-1))
  p_end = min(d, p)
  do k in p_start:p_end
    l=(d-k+1)
    y(:, k) = SpMV(A(j,:), y(:,k-1), level_ptr[l]:level_ptr[l+1])
  enddo
enddo
```

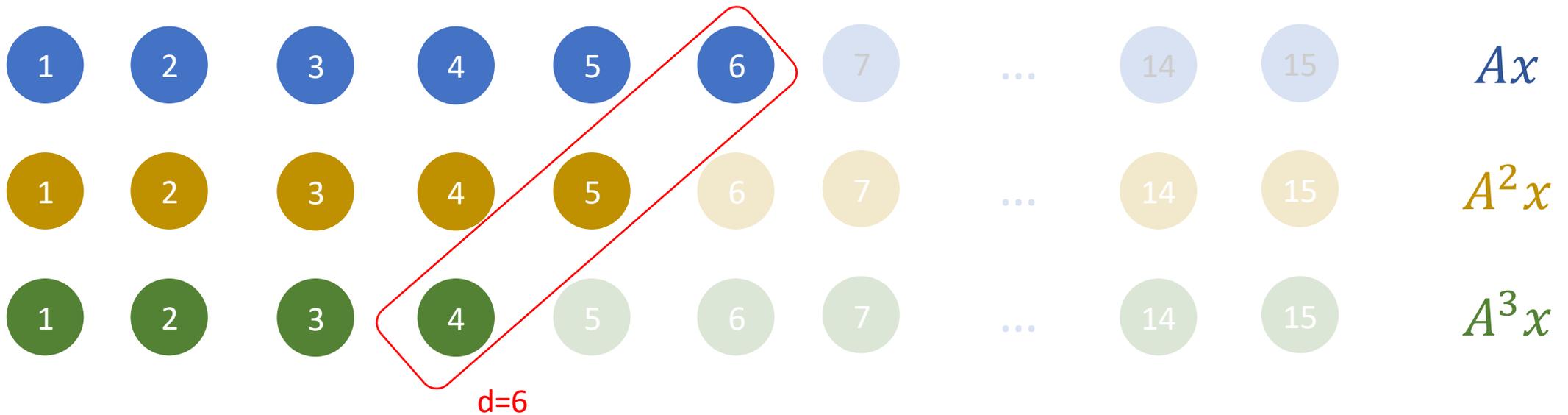
← Traverse along diagonal



# RACE: MPK Pseudocode

```
do d in 1:L+p-1
  p_start = max(1, d-(L-1))
  p_end = min(d, p)
  do k in p_start:p_end
    l=(d-k+1)
    y(:, k) = SpMV(A(j,:), y(:,k-1), level_ptr[l]:level_ptr[l+1])
  enddo
enddo
```

← Traverse along diagonal

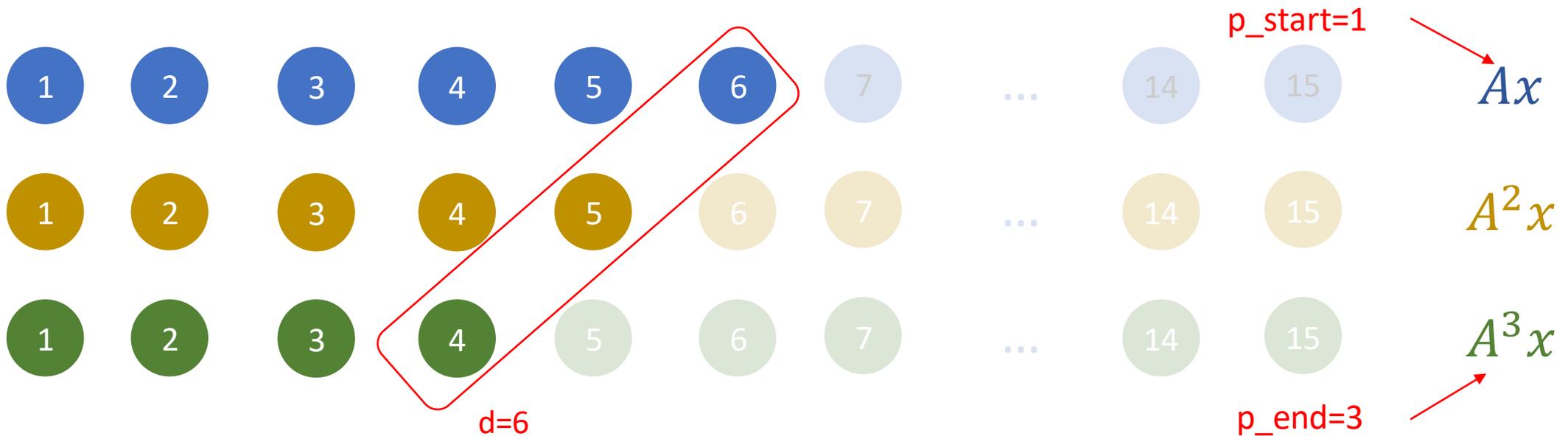


# RACE: MPK Pseudocode

```
do d in 1:L+p-1
  p_start = max(1, d-(L-1))
  p_end = min(d, p)
  do k in p_start:p_end
    l=(d-k+1)
    y(:, k) = SpMV(A(j,:), y(:,k-1), level_ptr[l]:level_ptr[l+1])
  enddo
enddo
```

← Traverse along diagonal

← All powers in diagonal



# RACE: MPK Pseudocode

```
do d in 1:L+p-1
  p_start = max(1, d-(L-1))
  p_end = min(d, p)
  do k in p_start:p_end
    l=(d-k+1)
    y(:, k) = SpMV(A(j,:), y(:,k-1), level_ptr[l]:level_ptr[l+1])
  enddo
enddo
```

← Traverse along diagonal

← All powers in diagonal

← Power k computation on level l



# RACE: MPK Pseudocode

```
do d in 1:L+p-1
  p_start = max(1, d-(L-1))
  p_end = min(d, p)
  do k in p_start:p_end
    l=(d-k+1)
    y(:, k) = SpMV(A(j, :), y(:, k-1), level_ptr[l]:level_ptr[l+1])
  enddo
enddo
```

← Traverse along diagonal

← All powers in diagonal

← Power k computation on level l

← OpenMP parallel SpMV on all vertices in level l



# RACE – Level traversal and matrix powers

```
do k = 1, p
  y(:, k) = SpMV(A, y(:, k-1))
enddo
```

Levels

Matrix Powers



A is loaded only once if  $(p + 1) \times N_{nz}(L) \times 12 \text{ bytes} < C$

$N_{nz}(L)$  – avg. non-zeros in a level  
 $C$  – cache size

# RACE – Level traversal and matrix powers

```
do k = 1, p
  y(:, k) = SpMV(A, y(:, k-1))
enddo
```

Levels

Matrix Powers



A is loaded only once if  $(p + 1) \times N_{nz}(L) \times 12 \text{ bytes} < C$

$N_{nz}(L)$  – avg. non-zeros in a level  
 $C$  – cache size

# RACE – Level traversal and matrix powers

```
do k = 1, p
  y(:, k) = SpMV(A, y(:, k-1))
enddo
```

Levels

Matrix Powers



A is loaded only once if  $(p + 1) \times N_{nz}(L) \times 12 \text{ bytes} < C$

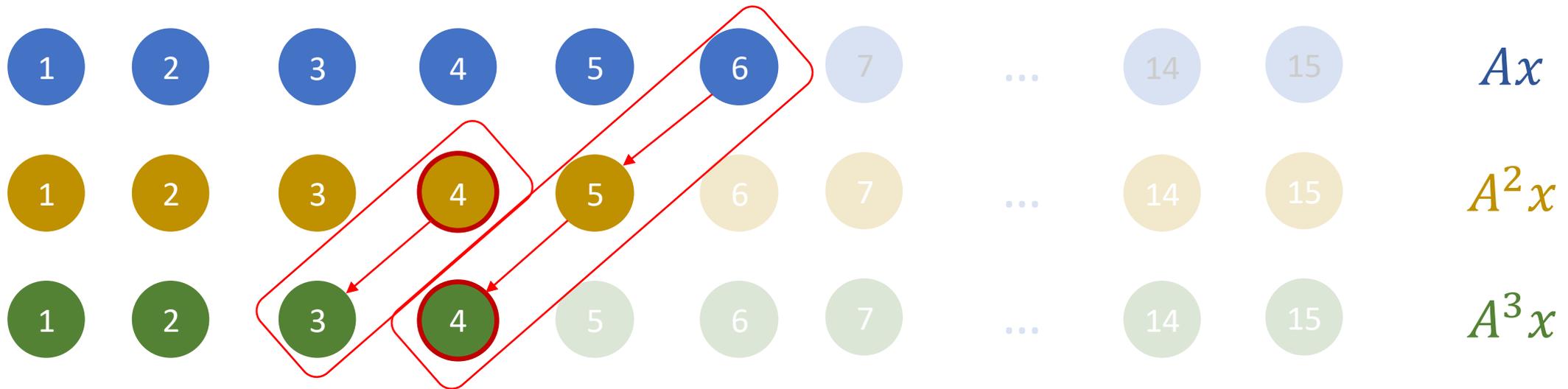
$N_{nz}(L)$  – avg. non-zeros in a level  
 $C$  – cache size

# RACE – Level traversal and matrix powers

```
do k = 1, p
  y(:, k) = SpMV(A, y(:, k-1))
enddo
```

Levels

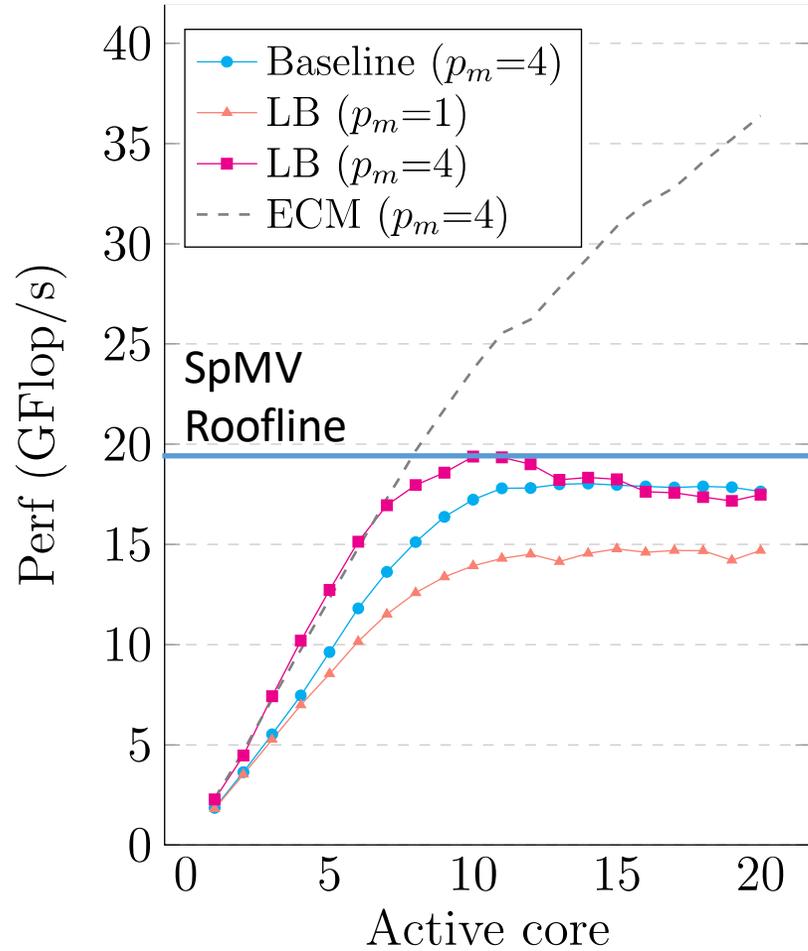
Matrix Powers



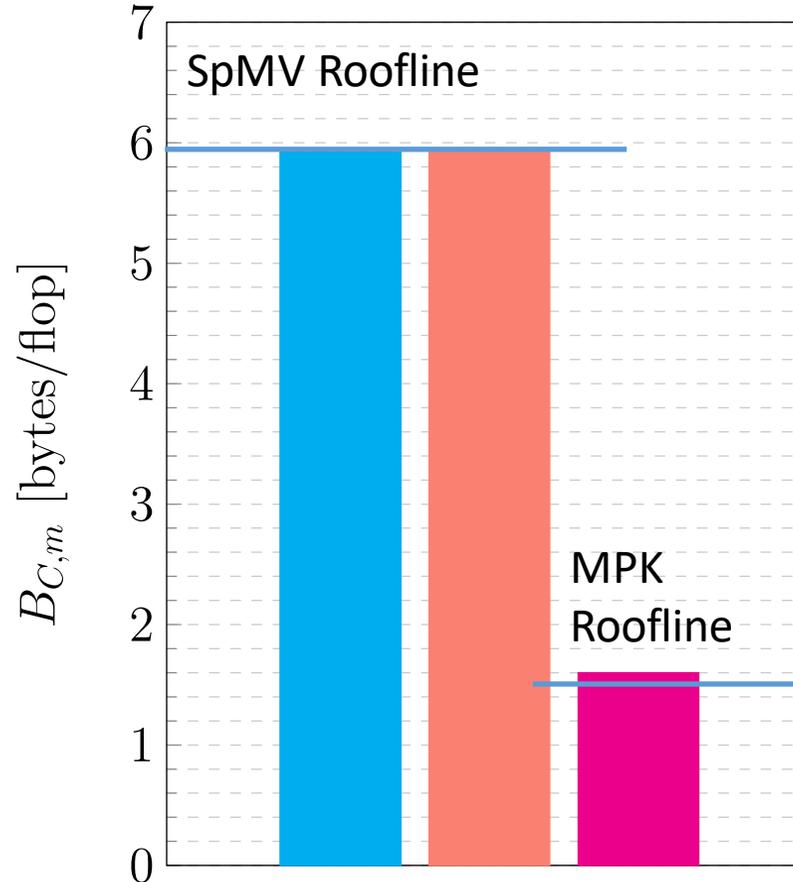
A is loaded only once if  
 $(p + 1) \times N_{nz}(L) \times 12 \text{ bytes} < C$

$N_{nz}(L)$  – avg. non-zeros in a level  
 $C$  – cache size

# RACE MPK – First Implementation



Performance



Memory traffic

Intel Xeon Gold 6248

- 1 Socket (20c)

**pwtk** matrix

- $N_r = 217,918$
- $N_{nz} = 11,634,424$

# RACE MPK – Performance Problem Identified

- Scheme seems to work (reduces data traffic) – at least for **pwtk**
- But: **Performance** 😞 !!!!
- Analysis of hardware performance counters (LIKWID) for **pwtk** matrix:  
`INSTR_RETIRED_ANY` up 2x for level based SpMV!

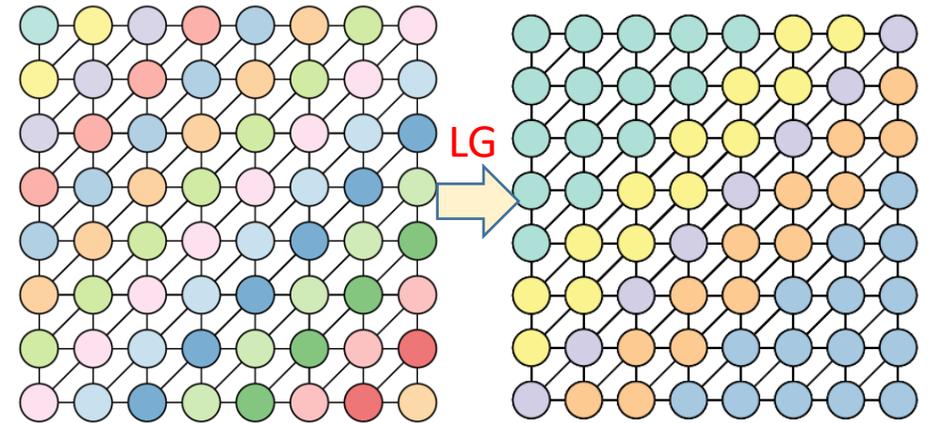
→ Frequent thread **synchronisations!**

Reason: After each level threads sync!

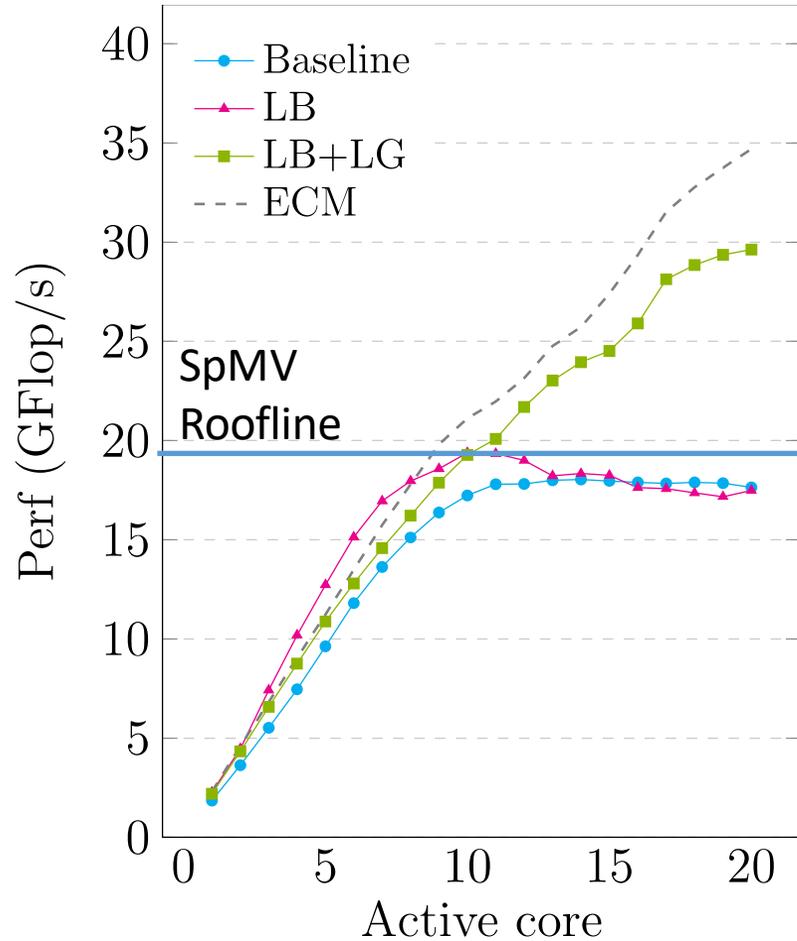
Measures:

→ Reduce #levels by level aggregation („**LG**“)

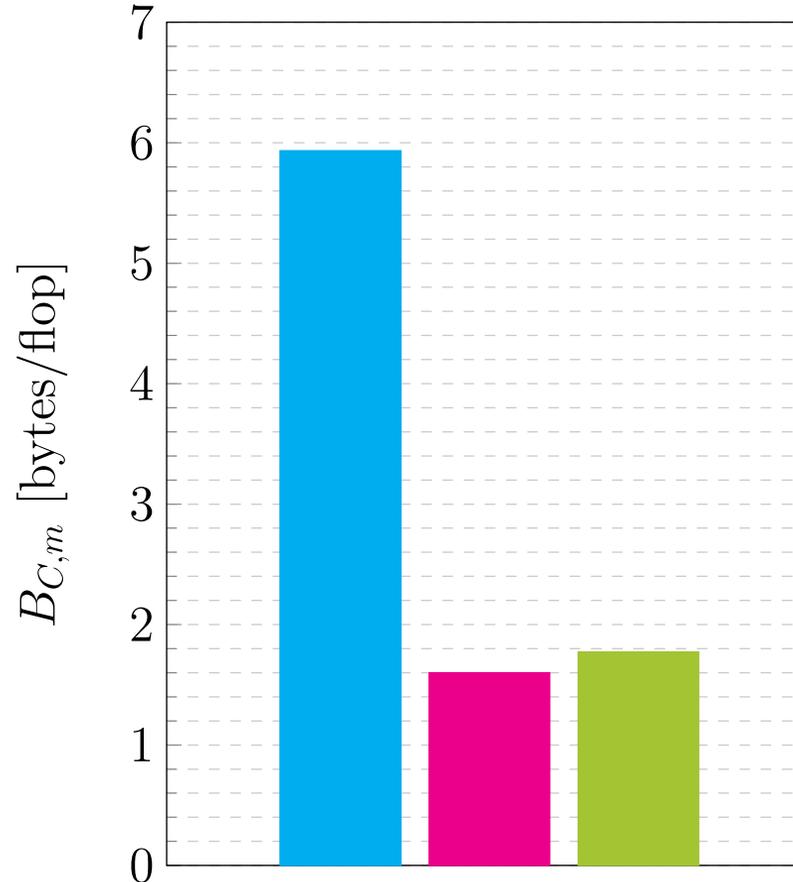
→ Global sync. replaced by point-to-point sync. („**p2p**“)



# RACE MPK – LG optimization



Performance



Memory traffic

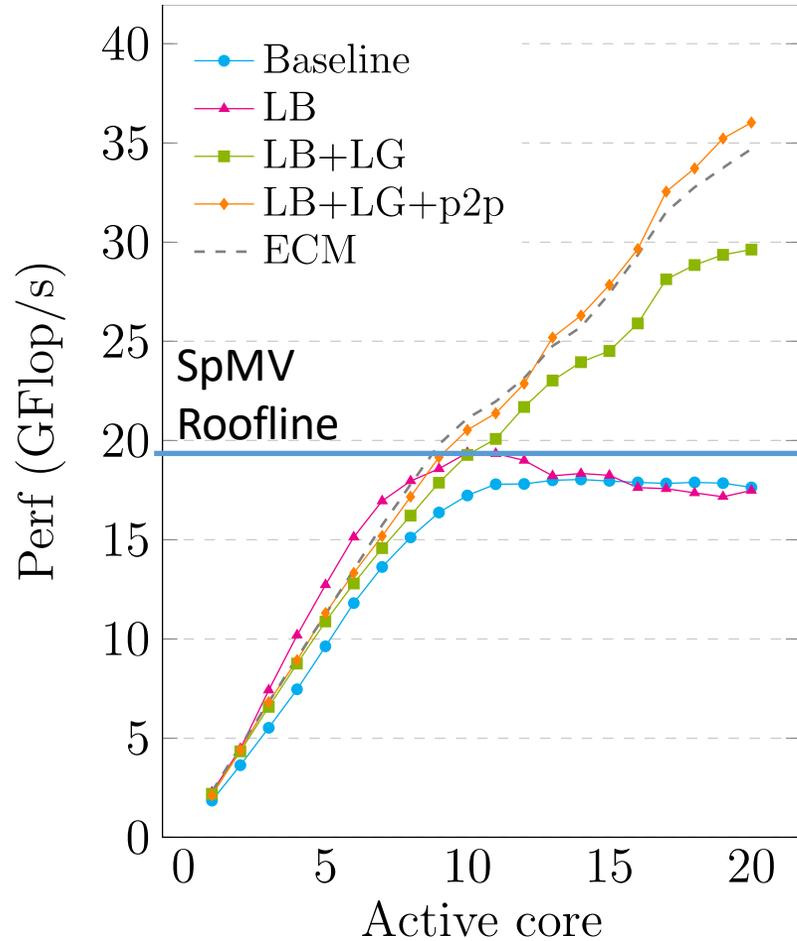
Intel Xeon Gold 6248

- 1 Socket (20c)

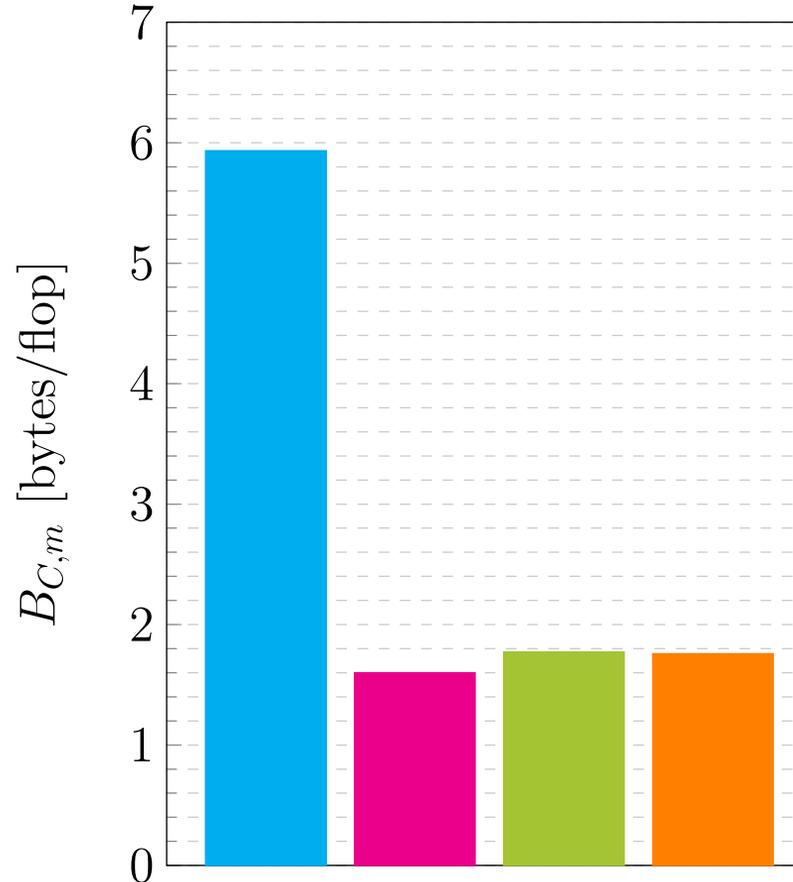
**pwtk** matrix

- $N_r = 217,918$
- $N_{nz} = 11,634,424$

# RACE MPK – LG+p2p optimization



Performance



Memory traffic

Intel Xeon Gold 6248

- 1 Socket (20c)

**pwtk** matrix

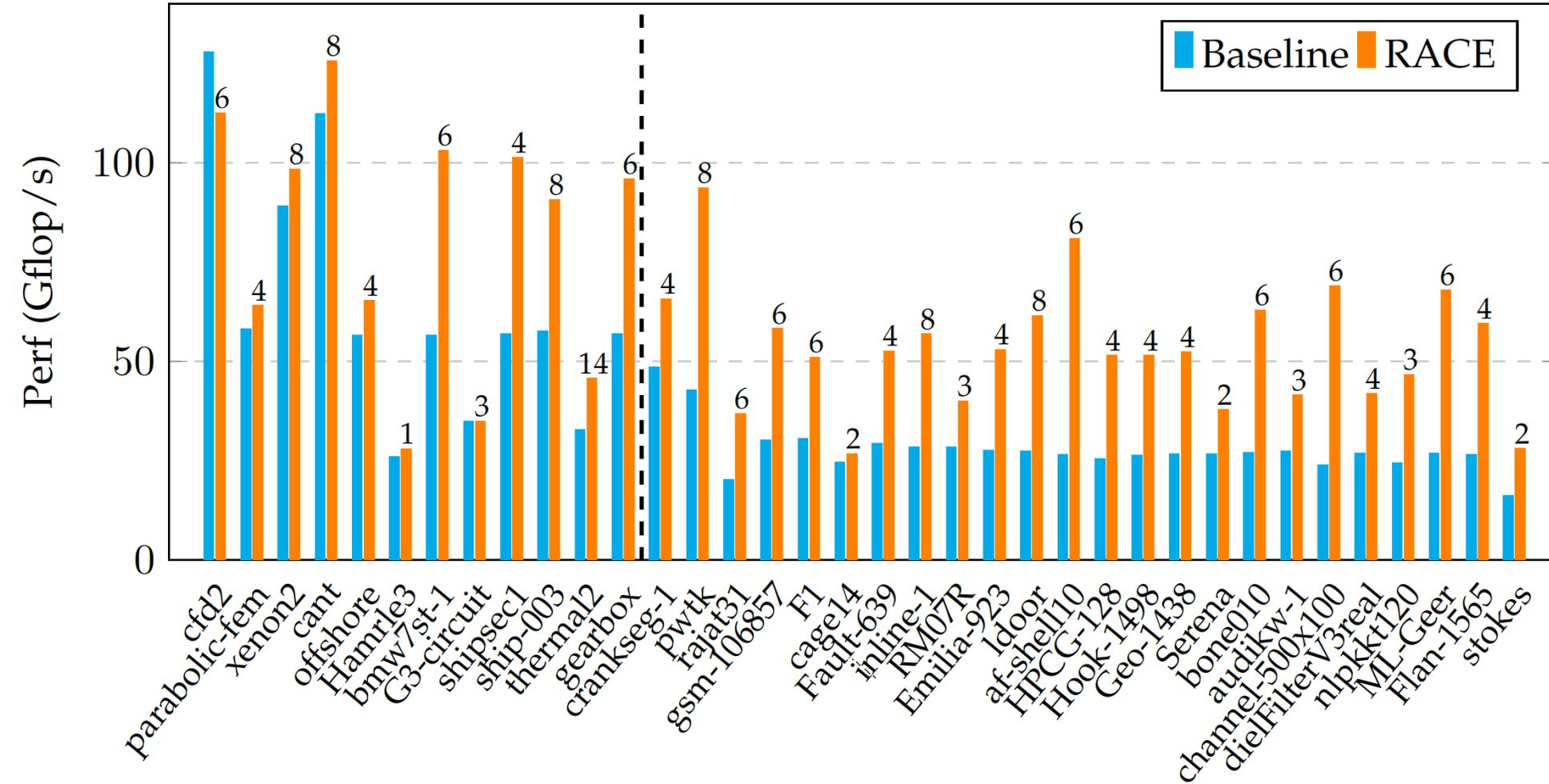
- $N_r = 217,918$
- $N_{nz} = 11,634,424$



# Matrix power kernel: Performance – Intel Ice Lake

Intel Xeon Platinum  
8368 (Ice Lake)

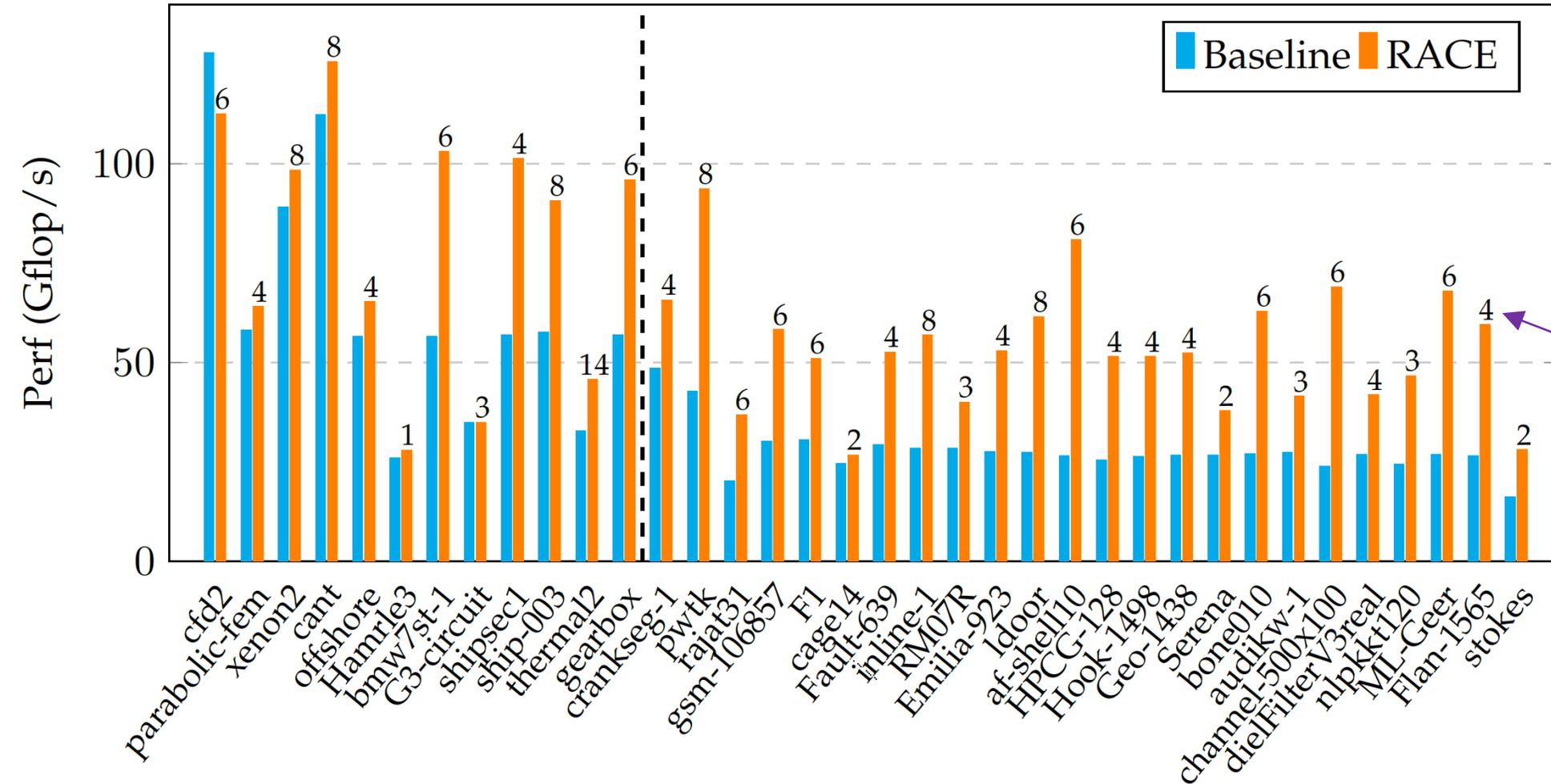
- 38 cores
- 104 MB cache (L2+L3)



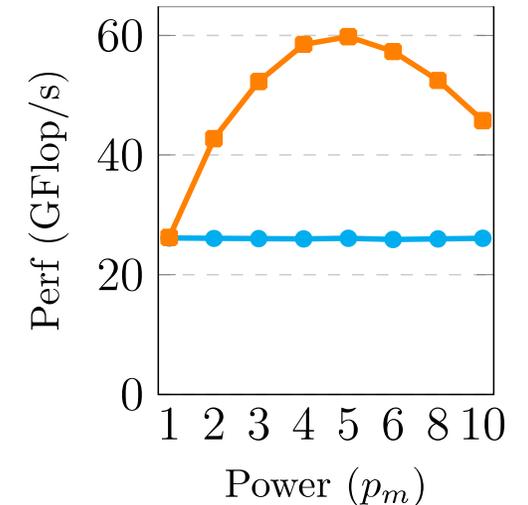
# Matrix power kernel: Performance – Intel Ice Lake

Intel Xeon Platinum 8368 (Ice Lake)

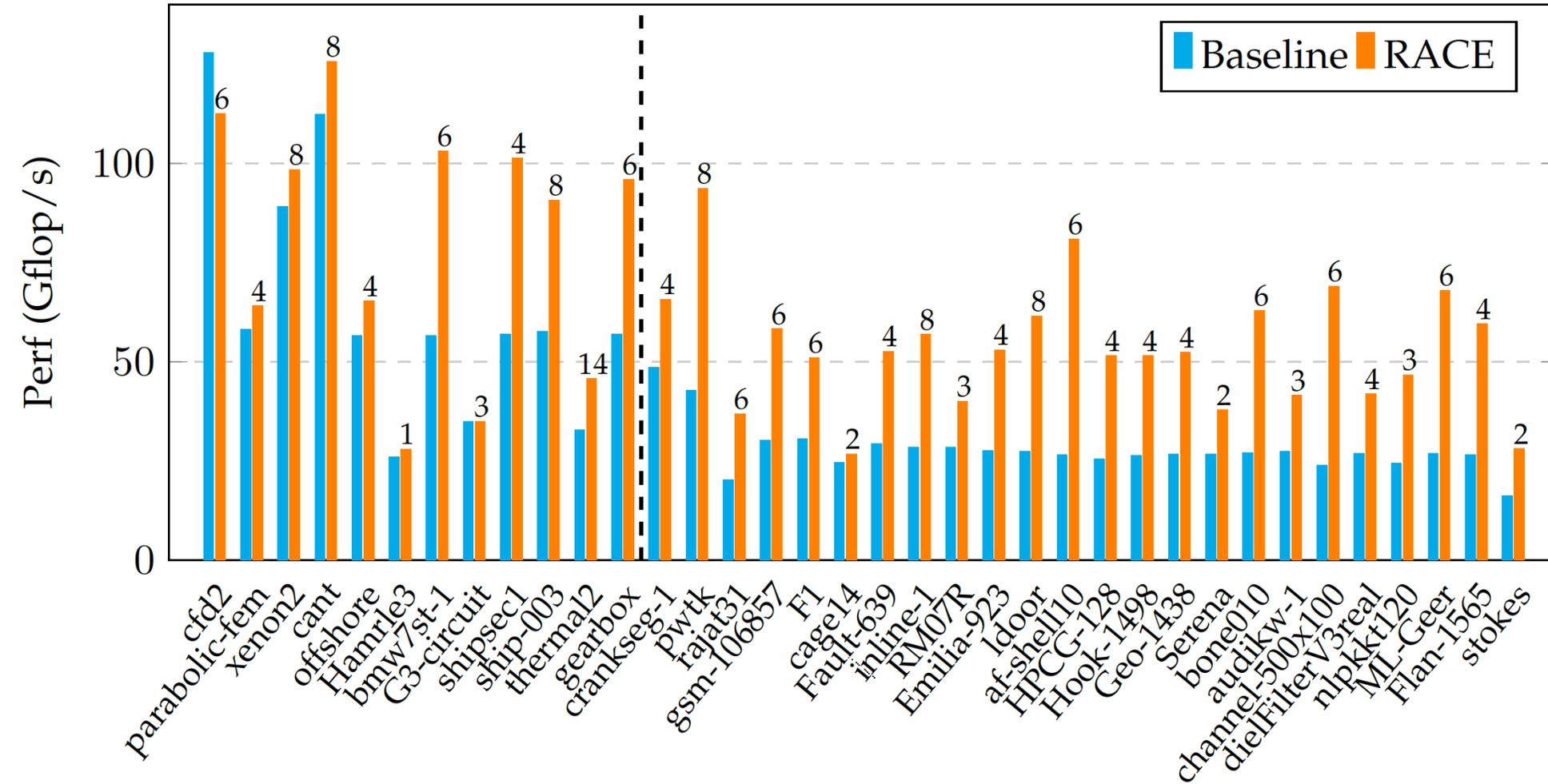
- 38 cores
- 104 MB cache (L2+L3)



Power value with maximum performance.

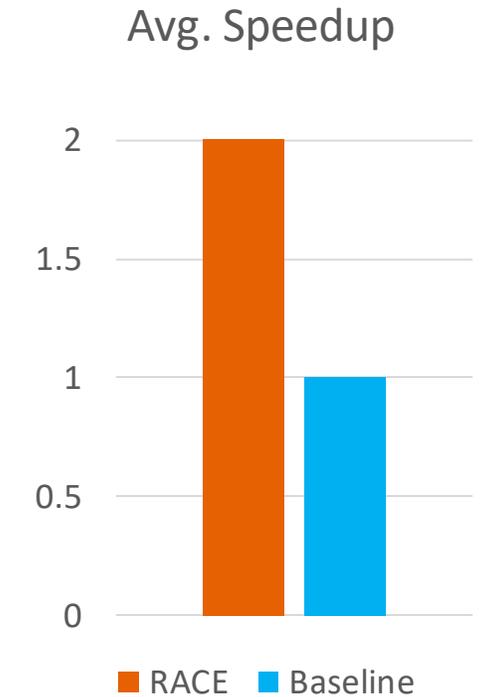


# Matrix power kernel: Performance – Intel Ice Lake

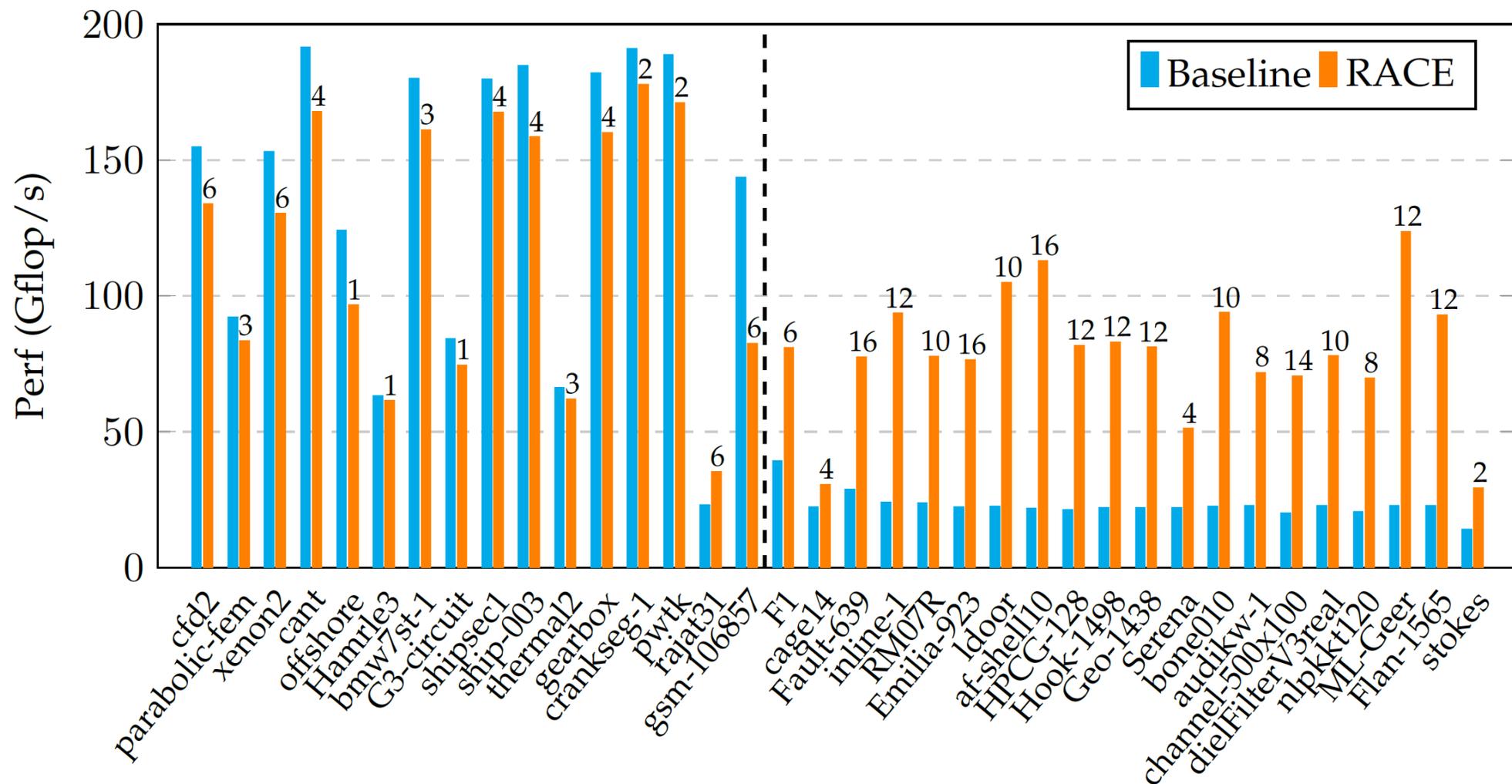


Intel Xeon  
Platinum 8368

- 38 cores
- 104 MB cache (L2+L3)

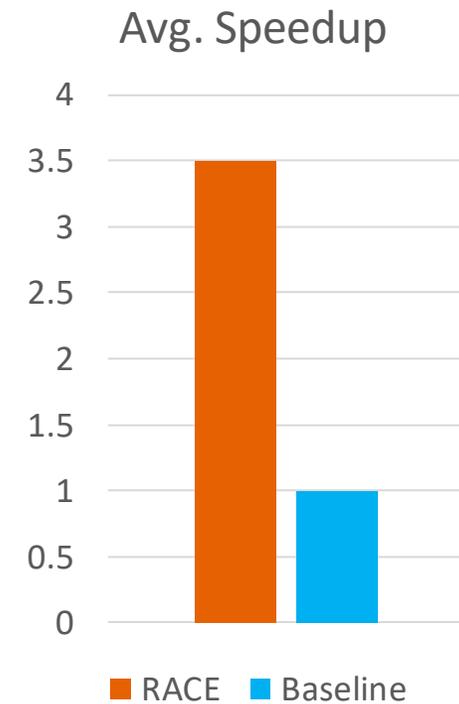


# Matrix power kernel: Performance – AMD Rome



AMD EPYC 7662

- 64 cores
- 288 MB cache (L2+L3)



# RACE - summary

- Inner kernel: OpenMP parallel standard SpMV routine
- Overhead: BFS & Set up of data structures (approx.  $\leq 50$  SpMVs)
- Parameters: Power ( $p_m$ ), Available Cache Size, Max. recursion depth
- Cache size  $\leftrightarrow$  max. polynomial degree ( $p_m$ )
  - Larger caches  $\rightarrow$  larger  $p_m \rightarrow$  better performance
  - Polynomial degree higher than  $p_m \rightarrow$  Computation in chunks of  $p_m$
- No loss of accuracy!

# RACE – MPK applications

- Exponential Integrators → Polynomial approximations
- $s$ -step Krylov methods (CA-GMRES)
- Polynomial preconditioning
- Algebraic Multigrid smoothers

# RACE – MPK applications

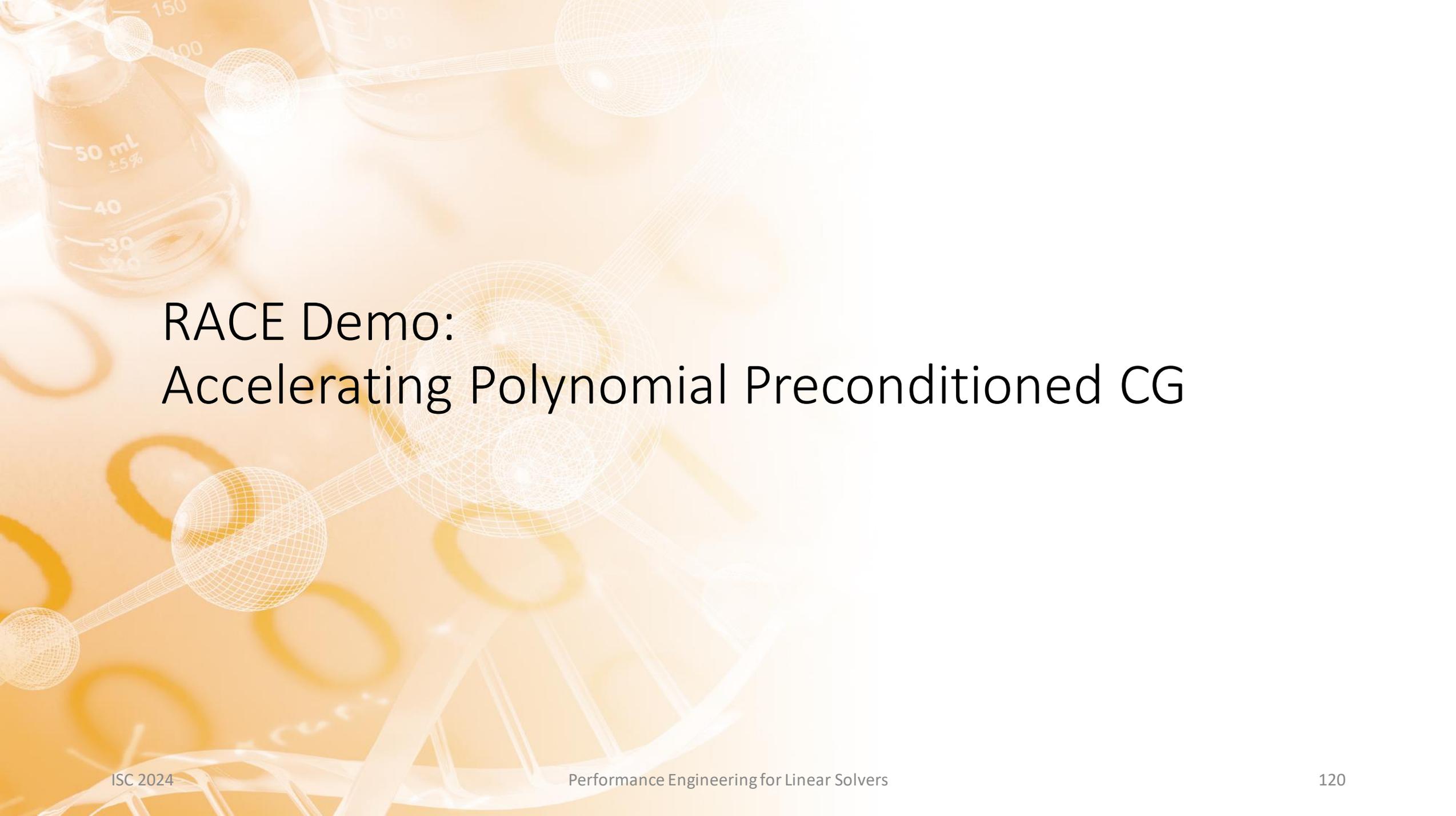
- Exponential Integrators → Polynomial approximations
- $s$ -step Krylov methods (CA-GMRES)
- Polynomial preconditioning
- Algebraic Multigrid smoothers

## ALGEBRAIC TEMPORAL BLOCKING FOR SPARSE ITERATIVE SOLVERS ON MULTI-CORE CPUS\*

CHRISTIE ALAPPAT<sup>†</sup>, JONAS THIES<sup>‡</sup>, GEORG HAGER<sup>†</sup>, HOLGER FEHSKE<sup>†</sup>, AND GERHARD WELLEIN<sup>† §</sup>

**Abstract.** Sparse linear iterative solvers are essential for many large-scale simulations. Much of the runtime of these solvers is often spent in the implicit evaluation of matrix polynomials via a sequence of sparse matrix-vector products. A variety of approaches has been proposed to make these polynomial evaluations explicit (i.e., fix the coefficients), e.g., polynomial preconditioners or  $s$ -step Krylov methods. Furthermore, it is nowadays a popular practice to approximate triangular solves by a matrix polynomial to increase parallelism. Such algorithms allow to evaluate the polynomial

<https://doi.org/10.48550/arXiv.2309.02228>



# RACE Demo: Accelerating Polynomial Preconditioned CG

# Using RACE

```
# include <RACE/interface.h>

RACE::dist k = RACE::POWER;
Nt = omp_get_num_threads();
RACE::Interface race (Nr, Nt, k, rowPtr, col );

//power value; here 4
int pm = 4 ;
//cache size in bytes; here 30 MB
double C = 30*1024*1024;
//perform pre-processing, find levels
race.RACEColor(pm, C);

int *perm, *invPerm , permLen=Nr;
race.getPerm(&perm, &permLen) ;
race.getInvPerm(&invPerm, &permLen) ;
//permute matrix and vector data structures
permute (perm, invPerm) ;
```

Pre-processing

```
struct functionArg
{
    //user-defined struct for input and output
    //arguments of the call-back function
    int Nr;
    . . .
};

//user-defined call-back function
void foo(int row_s, int row_e, int pow, void * voidArg)
{
    functionArg * arg = (functionArg *) voidArg;
    . . .
}

functionArg* args = new functionArg;
//fill args
args->Nr = 1000;
. . .

void* voidArgs = (void*) args;
int foo_id = race.registerFunction(&foo, voidArgs, pm);
race.executeFunction(foo_id);
```

Processing

# Neumann polynomial apply

$$w = (I - L)^k A (I - U)^k v$$

# Neumann polynomial apply

$$w = (I - L)^k A (I - U)^k v$$

$$t_1 = (I - U)^k v$$

$$t_2 = A t_1$$

$$w = (I - L)^k t_2$$

# Neumann polynomial apply

$$w = (I - L)^k A (I - U)^k v$$

$$t_1 = (I - U)^k v$$

Cache blocking

$$t_2 = At_1$$

$$w = (I - L)^k t_2$$

Cache blocking

# Neumann polynomial apply

$$w = (I - L)^k A (I - U)^k v$$

$$t_1 = (I - U)^k v$$

Cache blocking

$$t_2 = At_1$$

$$w = (I - L)^k t_2$$

Cache blocking

Can we do better?

# Neumann polynomial apply

$$w = (I - L)^k A (I - U)^k v$$

$$t_1 = (I - U)^k v$$

$$t_2 = At_1 = (L + U)t_1$$

$$w = (I - L)^k t_2$$

# Neumann polynomial apply

$$w = (I - L)^k A (I - U)^k v$$



$$t_1 = (I - U)^k v$$

$$t_2 = At_1 = (L + U)t_1$$

$$w = (I - L)^k t_2$$

Cache blocking

# Neumann polynomial apply

$$w = (I - L)^k A (I - U)^k v$$



$$t_1 = (I - U)^k v$$

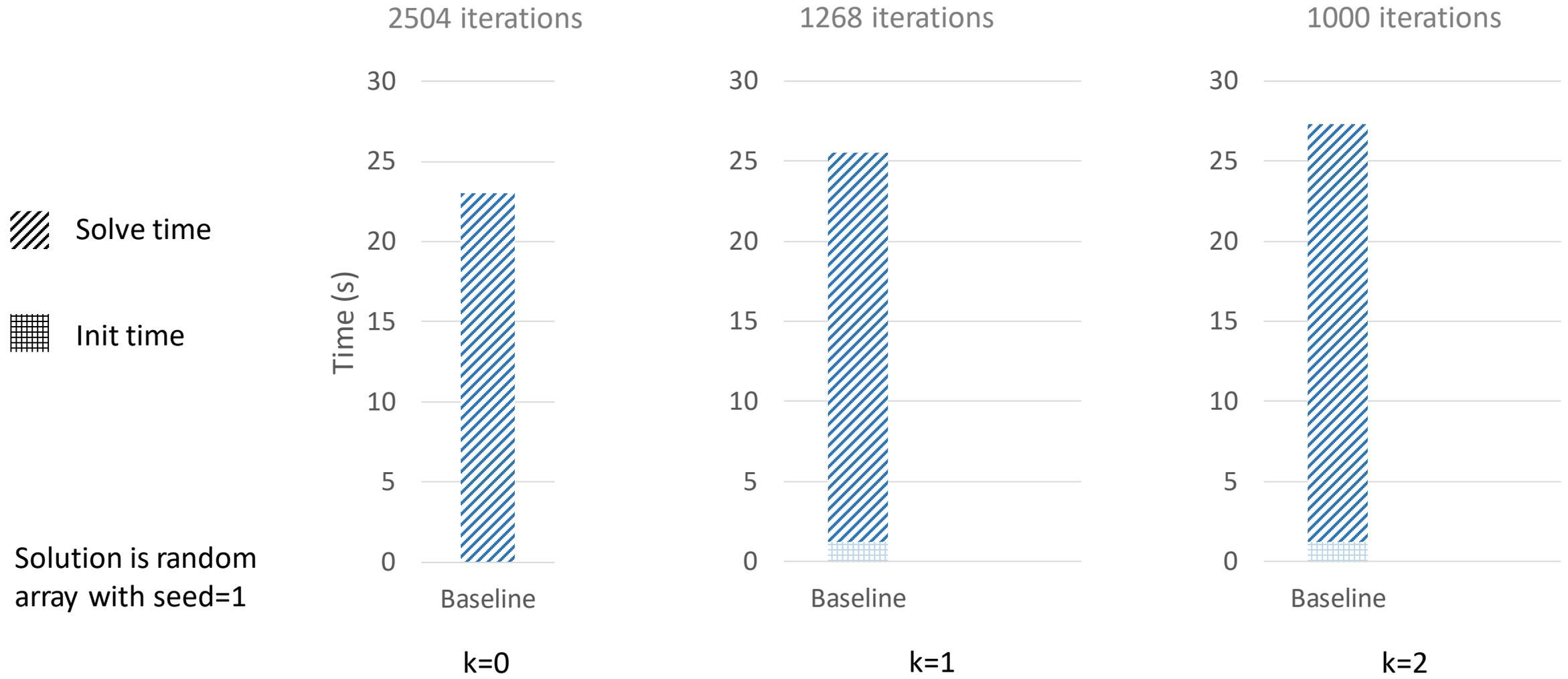
$$t_2 = A t_1 = (L + U) t_1$$

$$w = (I - L)^k t_2$$

Cache blocking

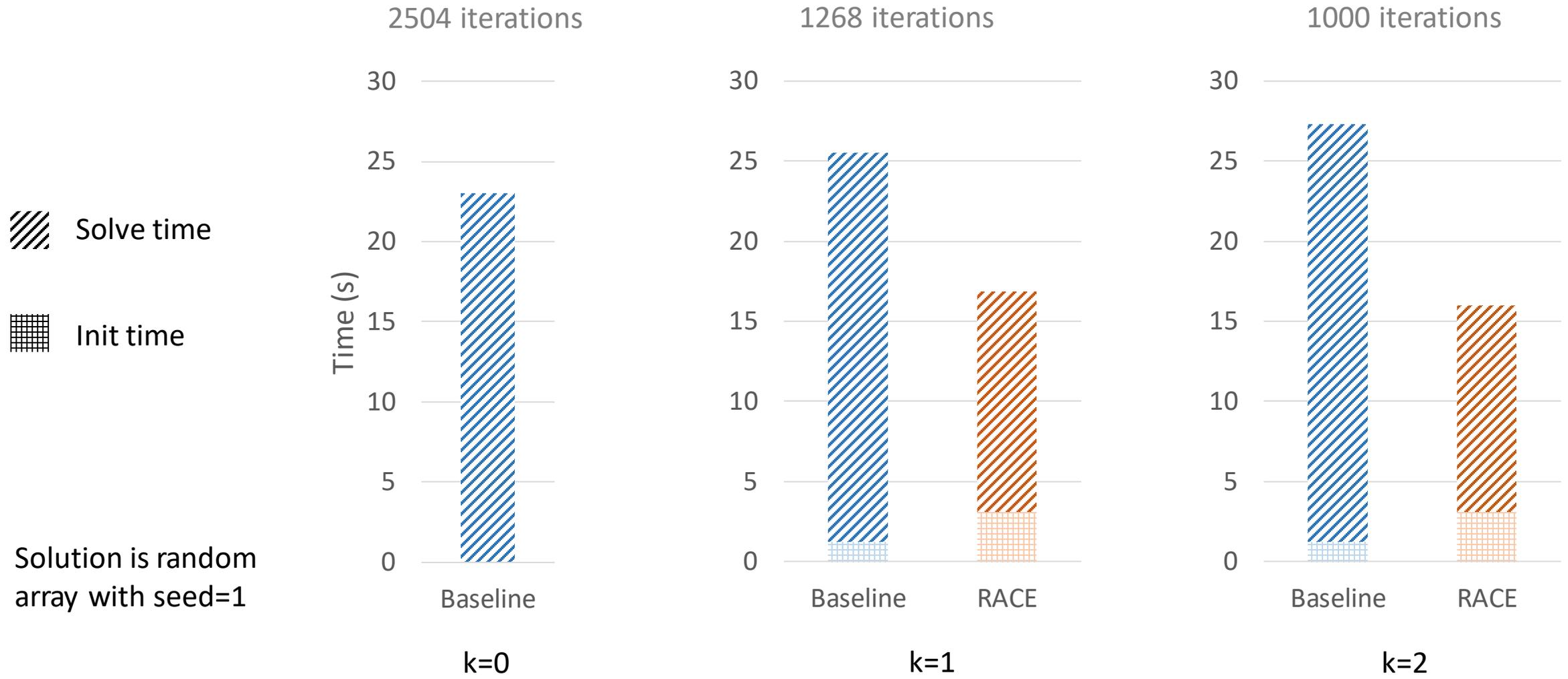
Total power =  $2k + 1$

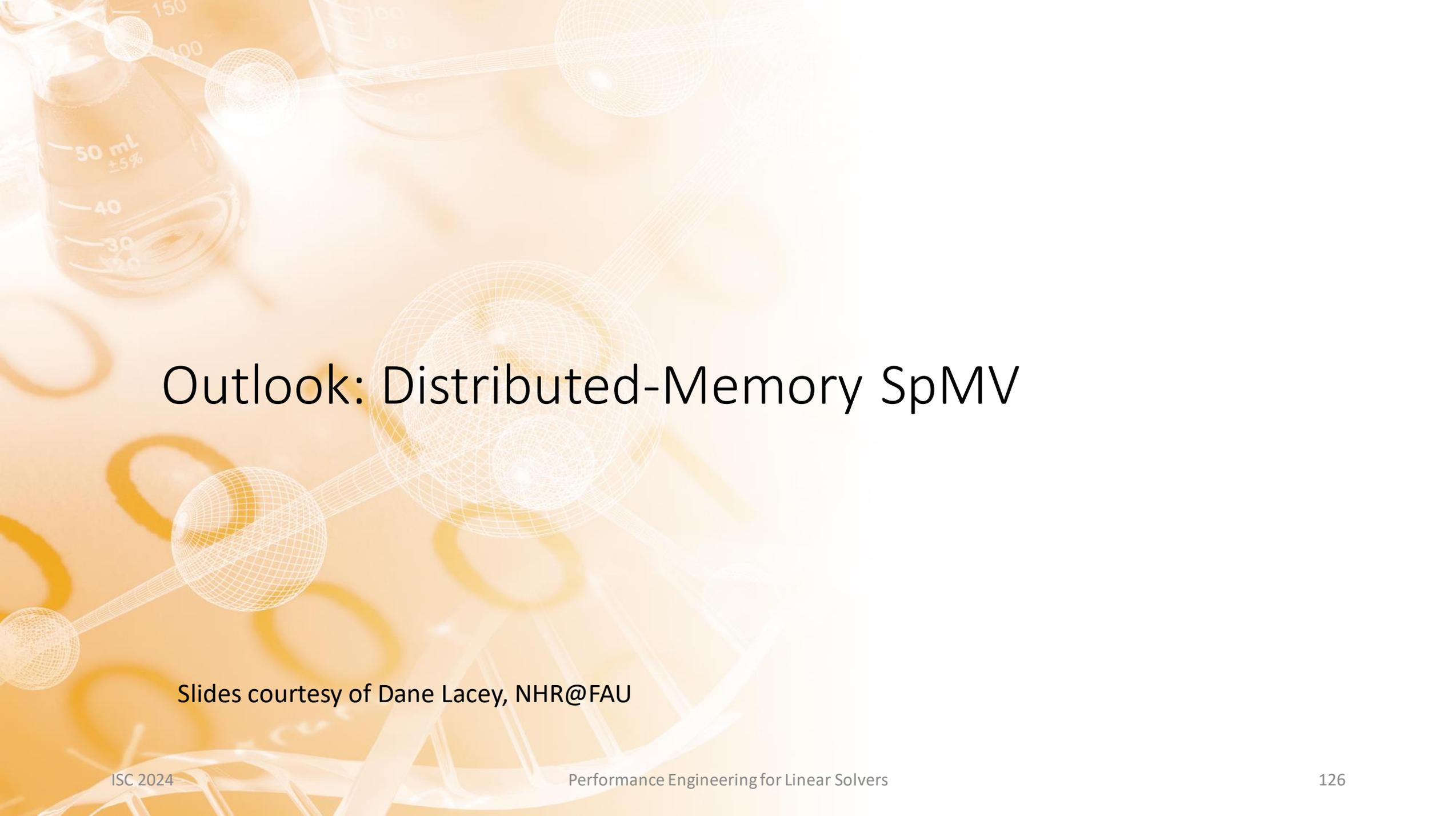
# Time to solve Laplace2000x2000 to 1e-3 tolerance on 1 NUMA domain (18c) of Intel Ice lake (Fritz)



Solution is random array with seed=1

# Time to solve Laplace2000x2000 to 1e-3 tolerance on 1 NUMA domain (18c) of Intel Ice lake (Fritz)





# Outlook: Distributed-Memory SpMV

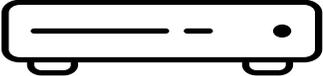
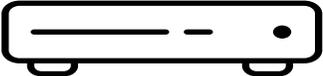
Slides courtesy of Dane Lacey, NHR@FAU



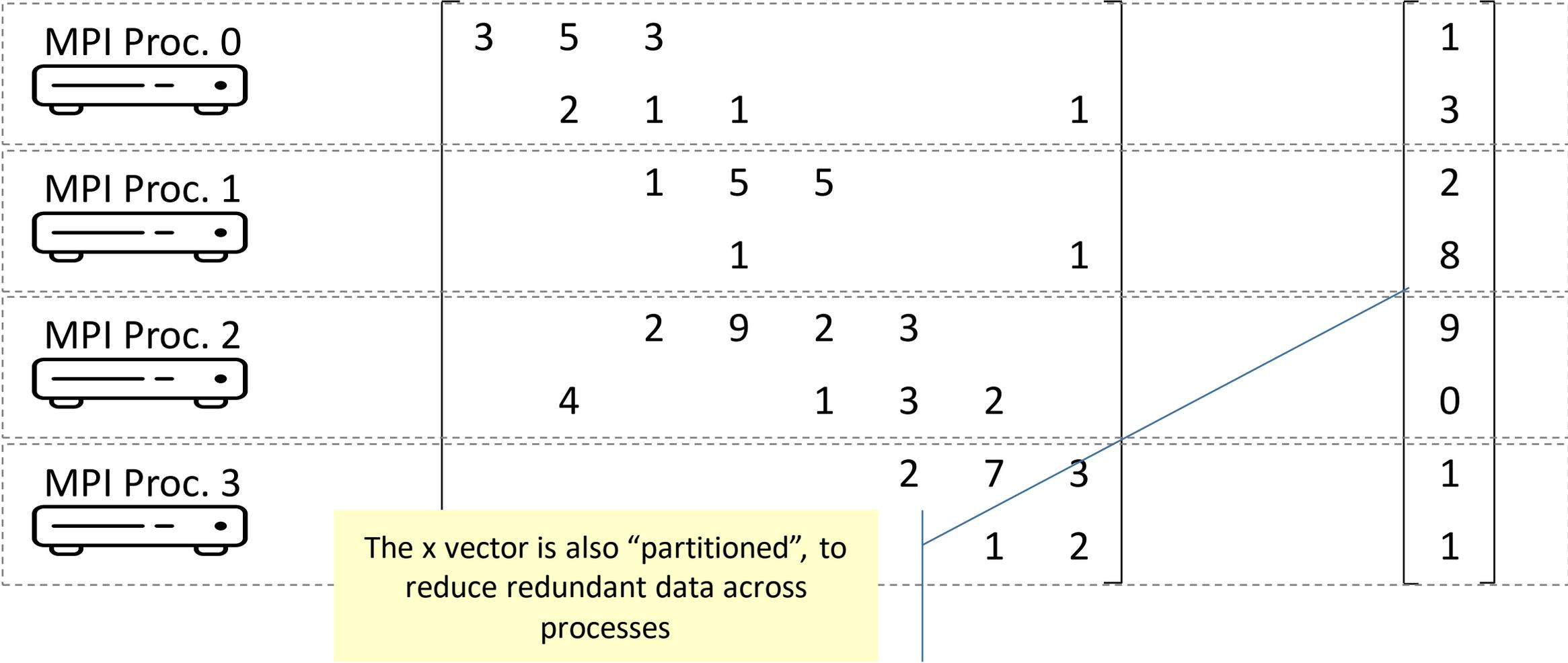




# Distributed SpMV Example

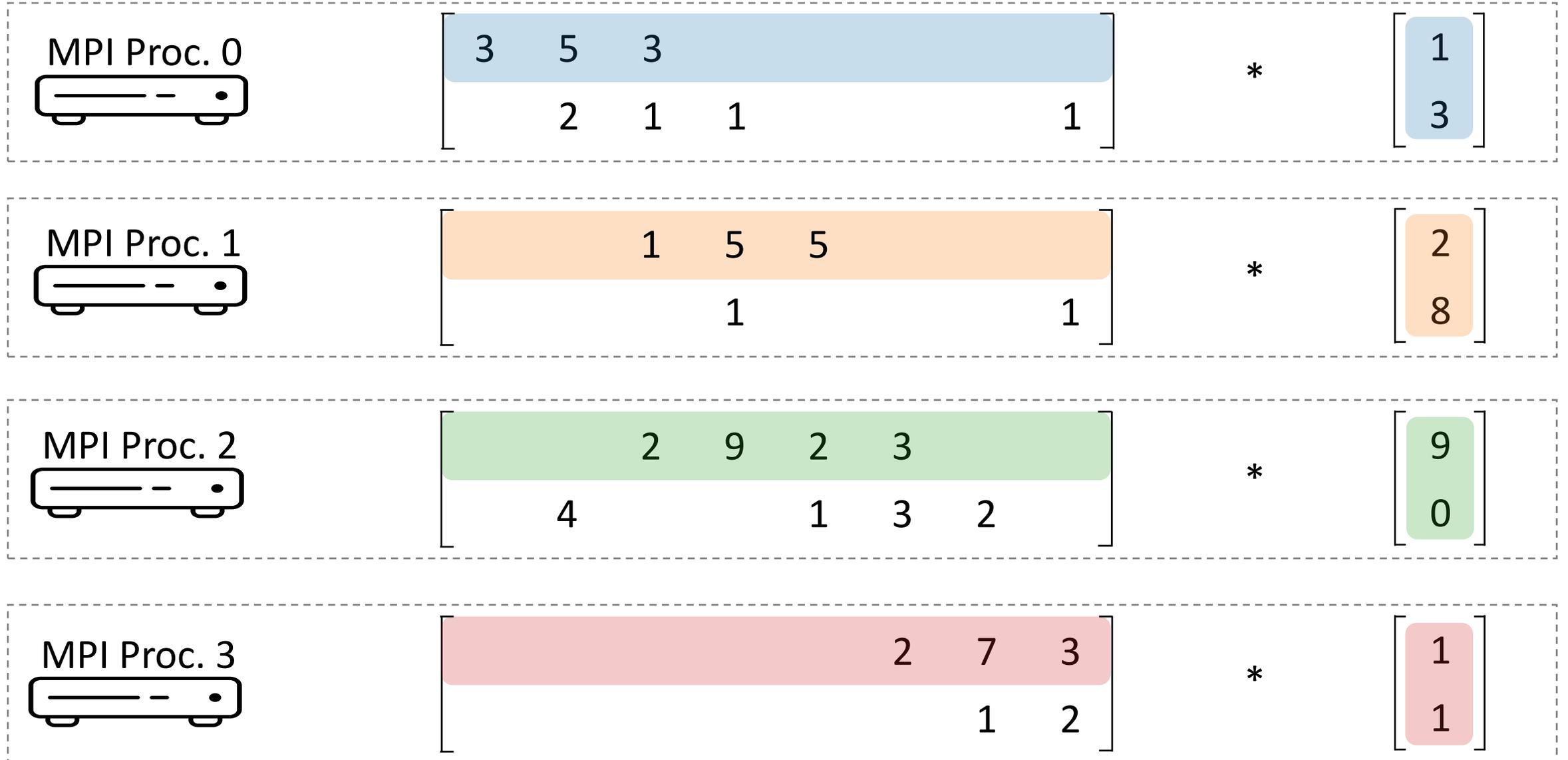
<p>MPI Proc. 0</p> 	3	5	3							1
		2	1	1					1	3
<p>MPI Proc. 1</p> 			1	5	5					2
				1					1	8
<p>MPI Proc. 2</p> 			2	9	2	3				9
		4			1	3	2			0
<p>MPI Proc. 3</p> 						2	7	3		1
							1	2		1

# Distributed SpMV Example

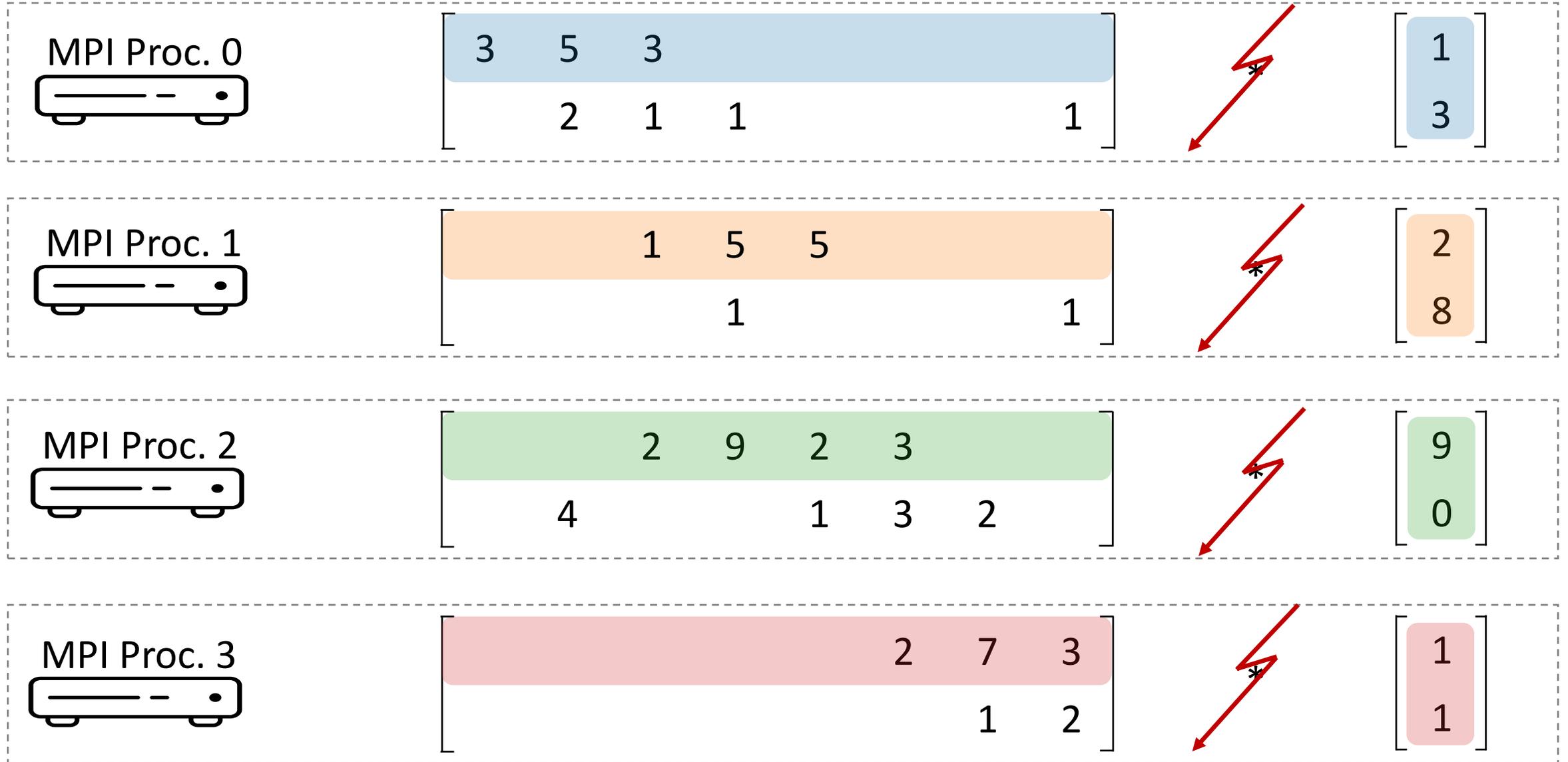




# Distributed SpMV Example



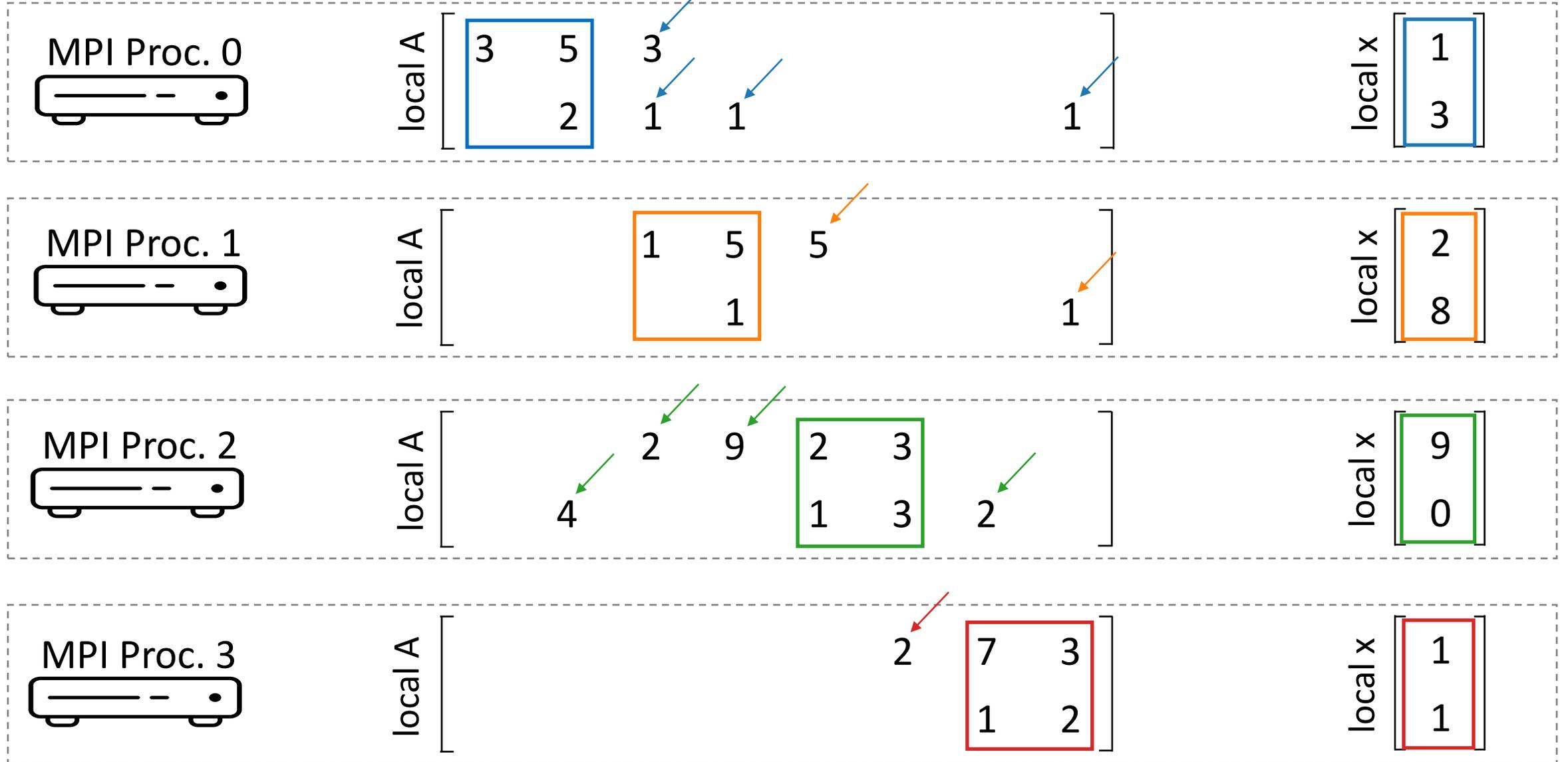
# Distributed SpMV Example



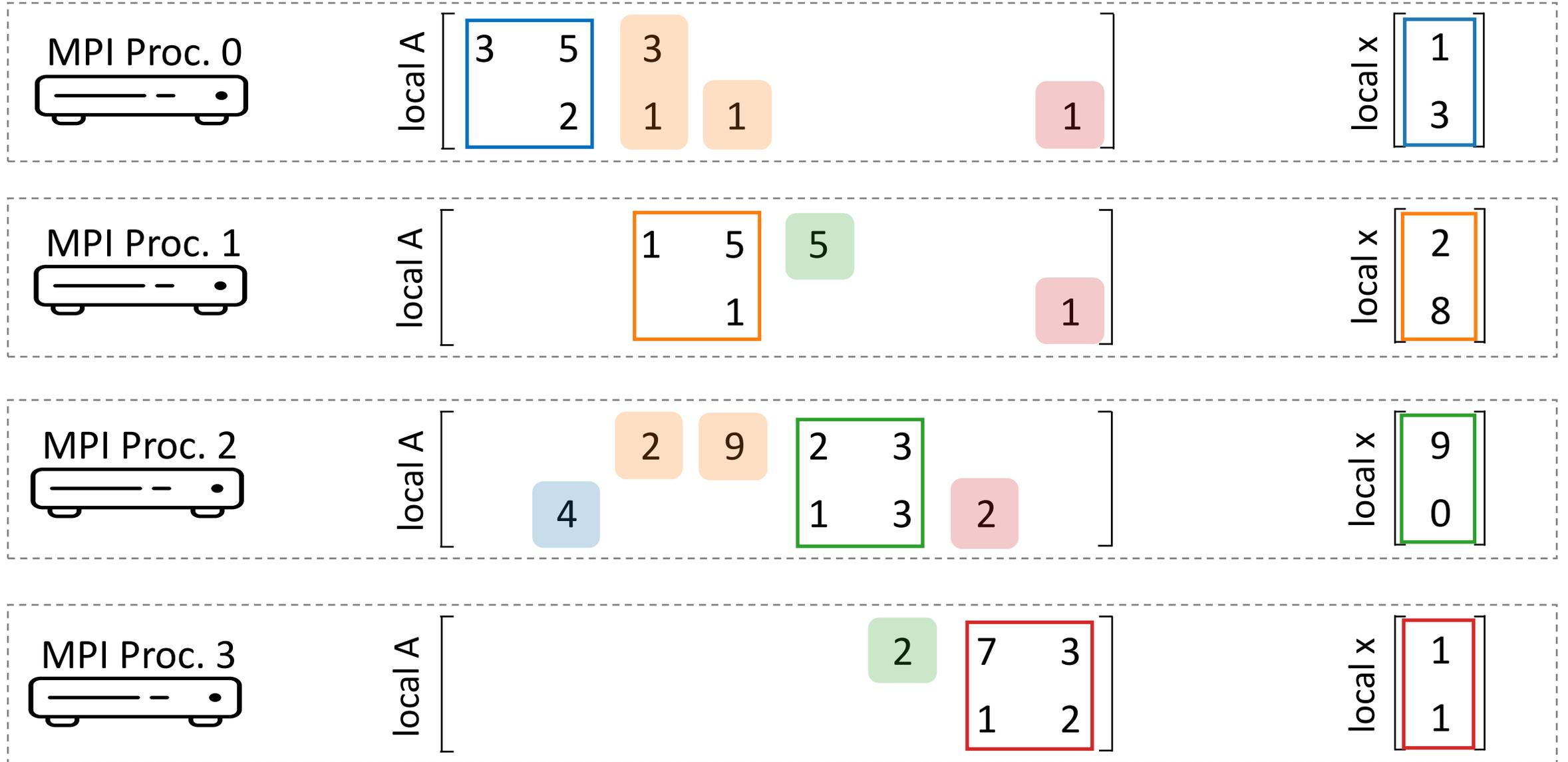




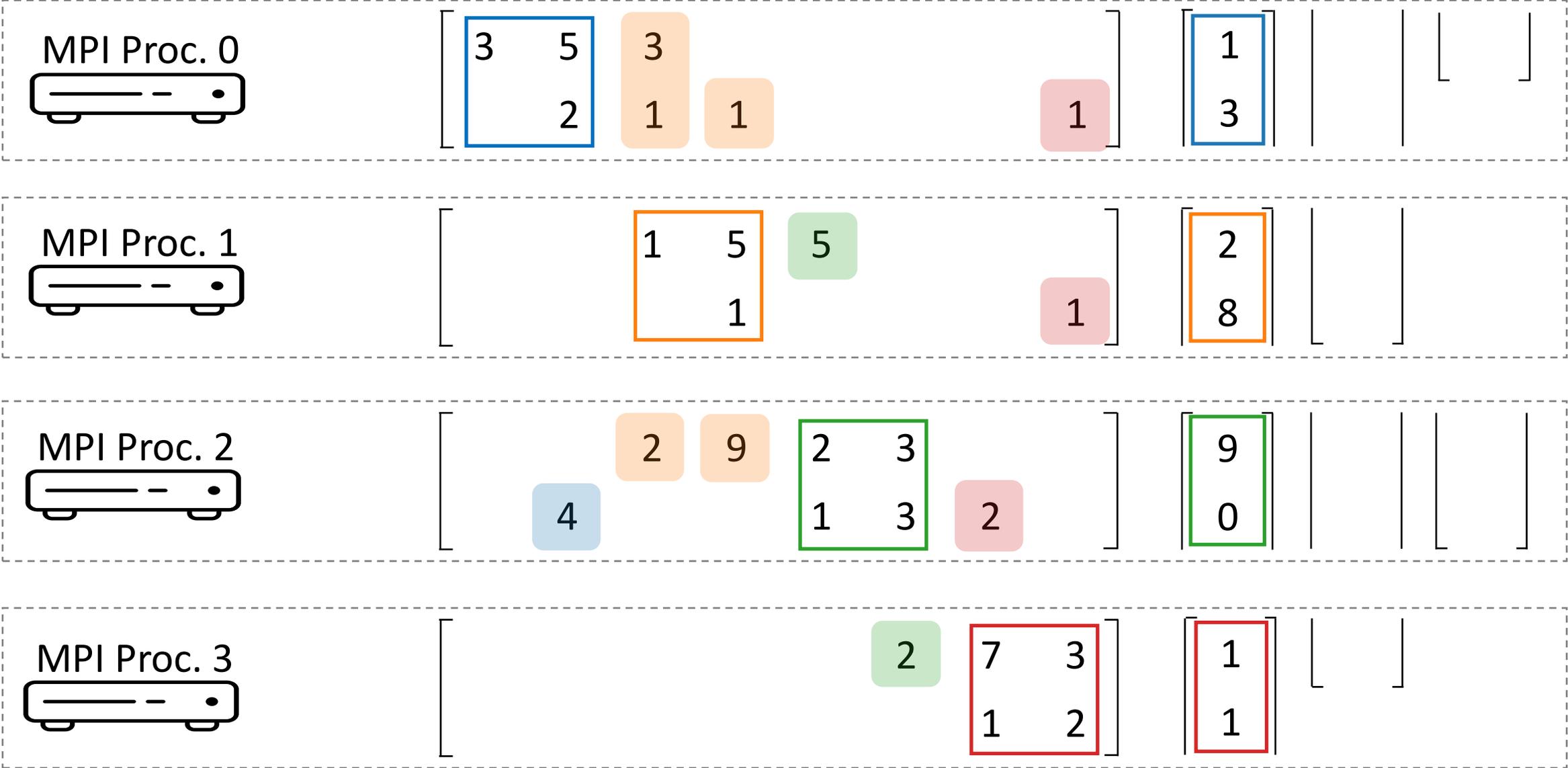
# Distributed SpMV Example



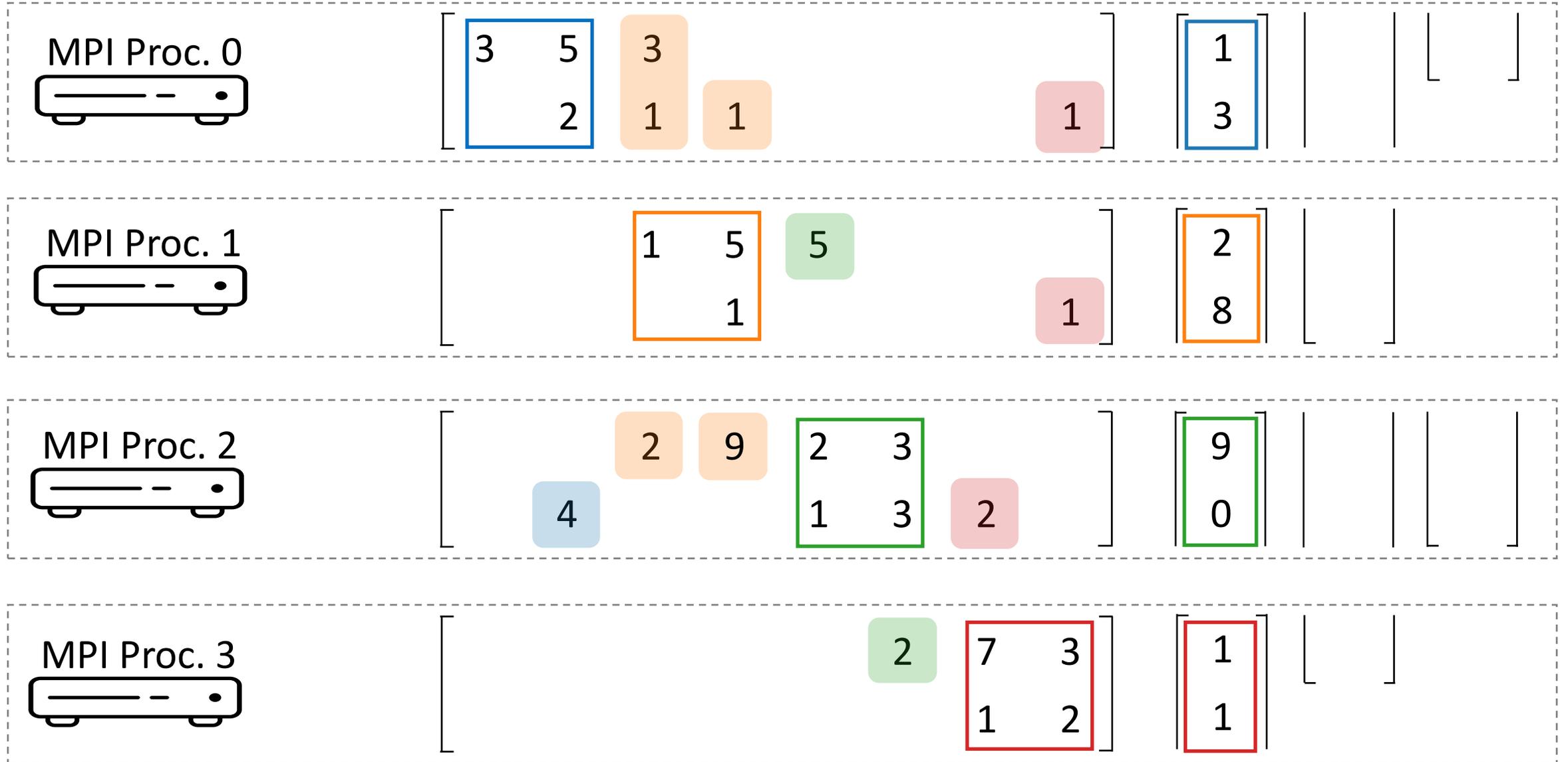
# Distributed SpMV Example



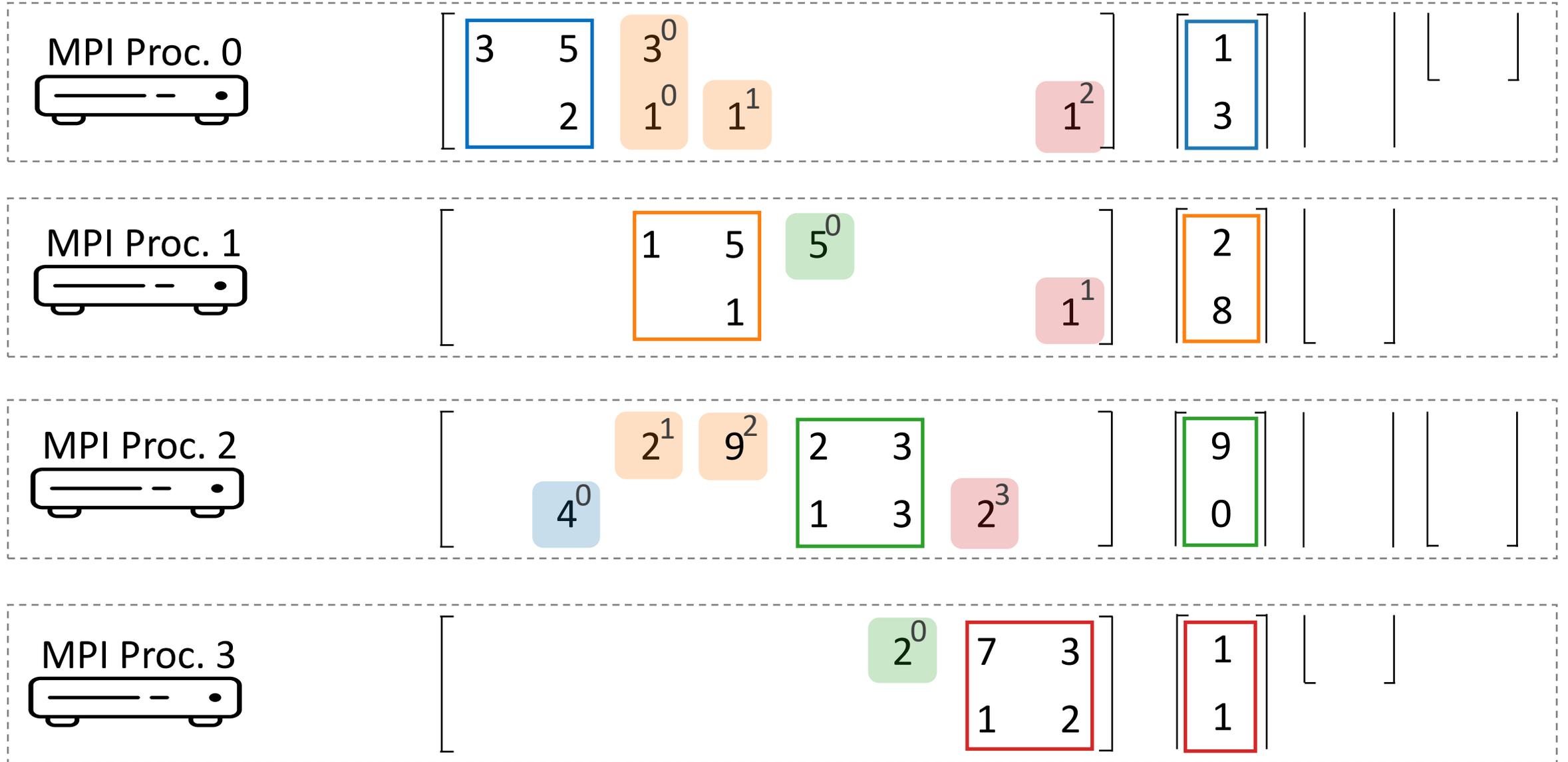
# Distributed SpMV Example



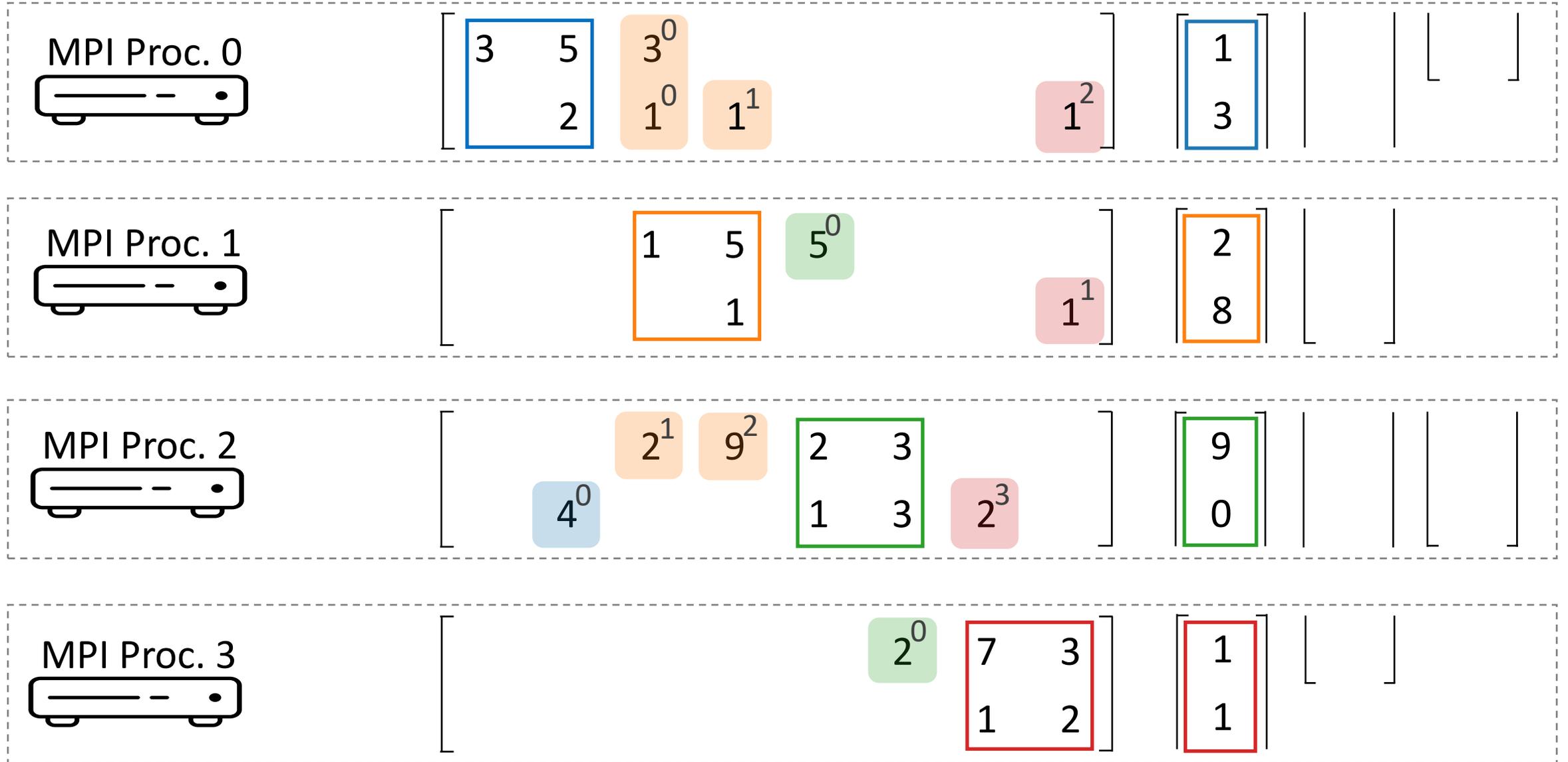
# Distributed SpMV Example



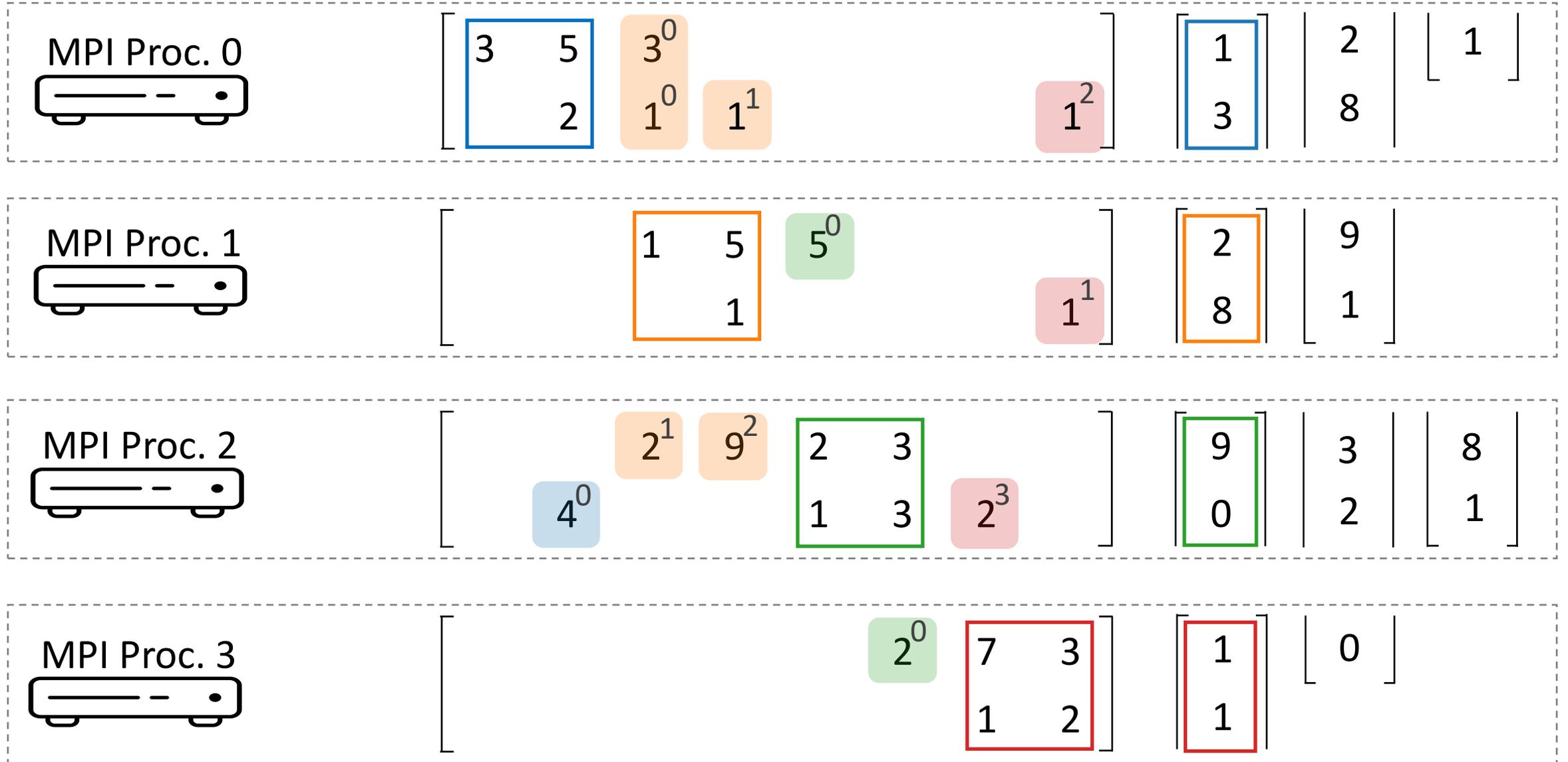
# Distributed SpMV Example



# Distributed SpMV Example

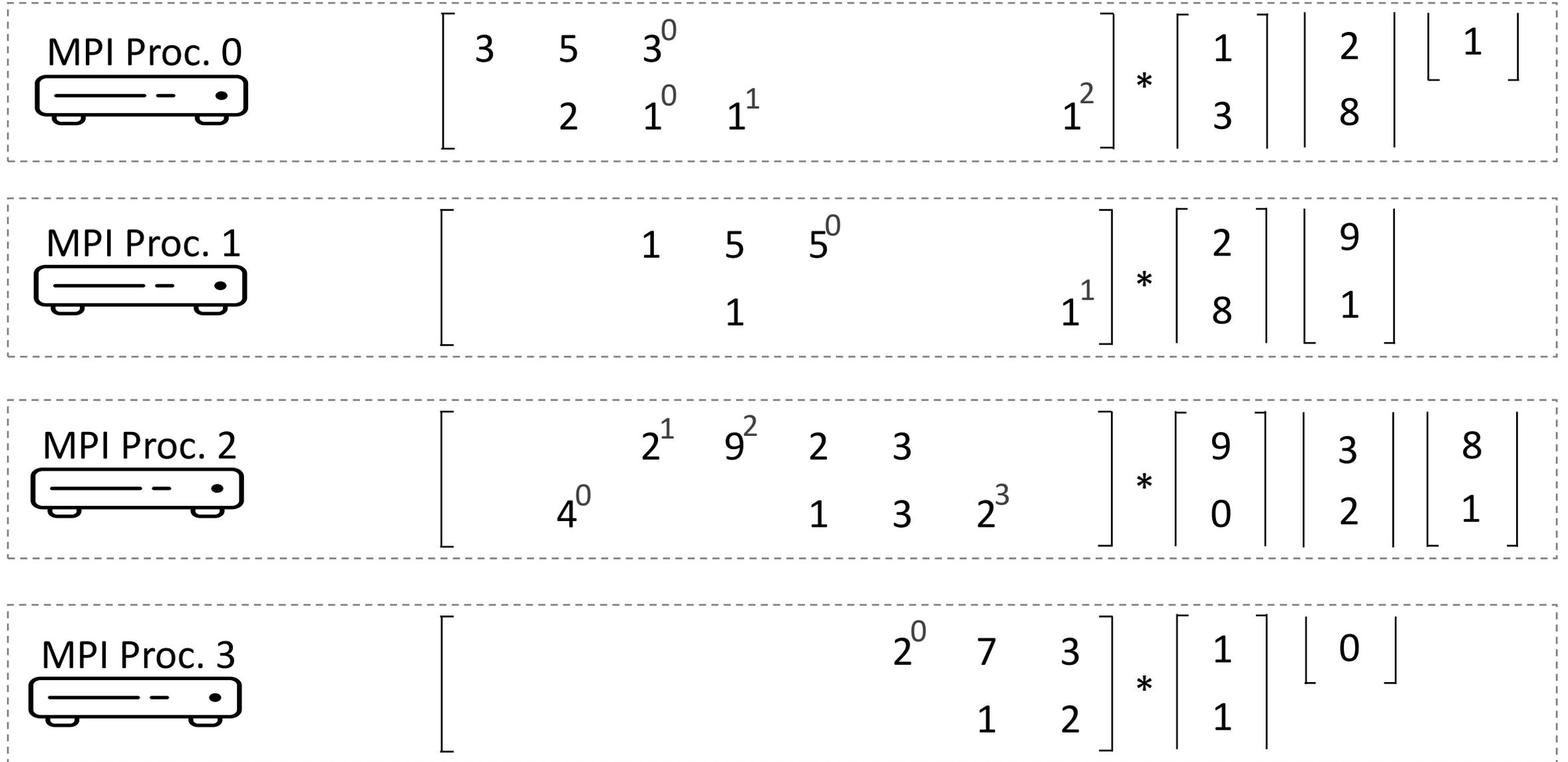


# Distributed SpMV Example

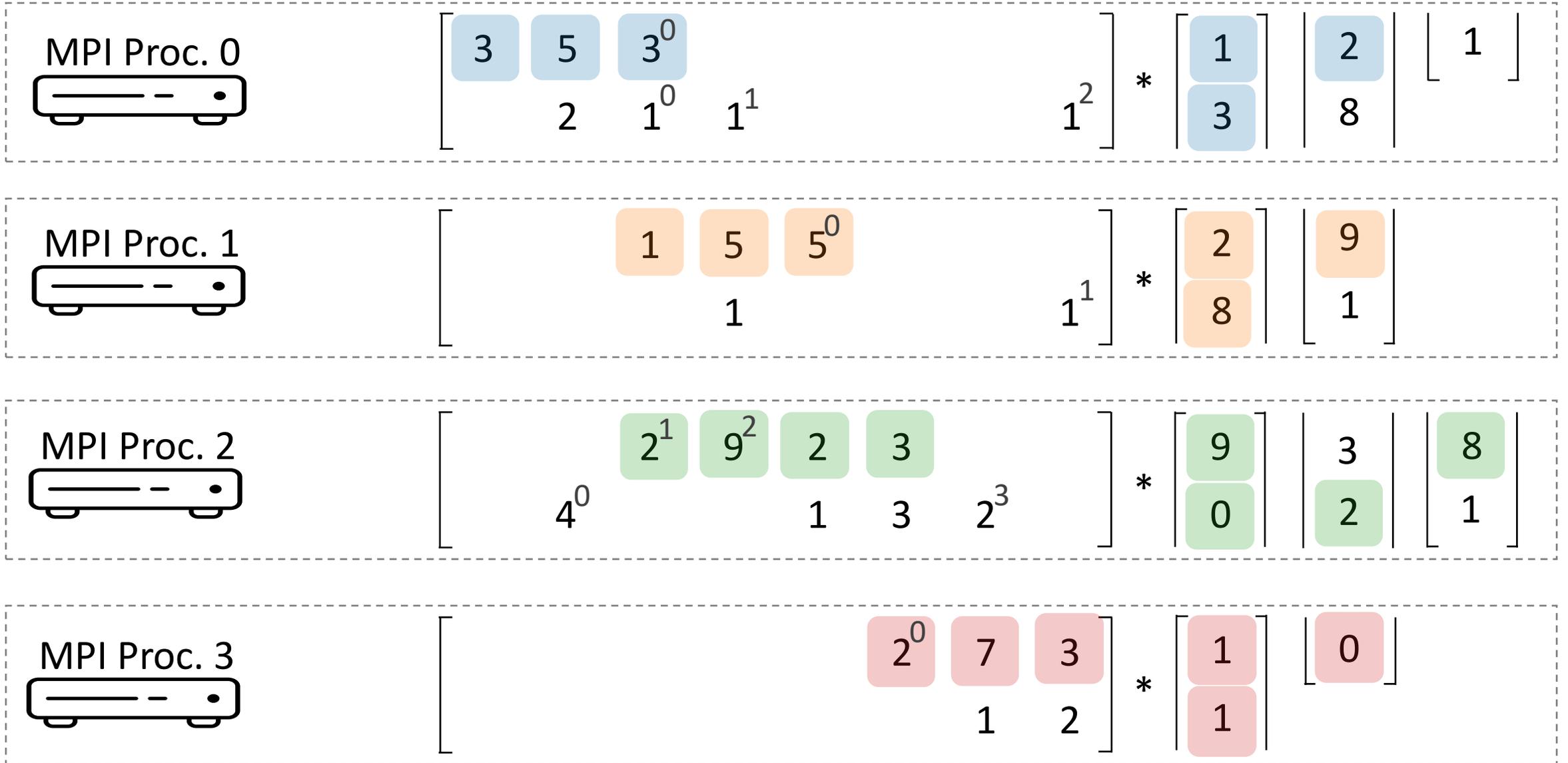




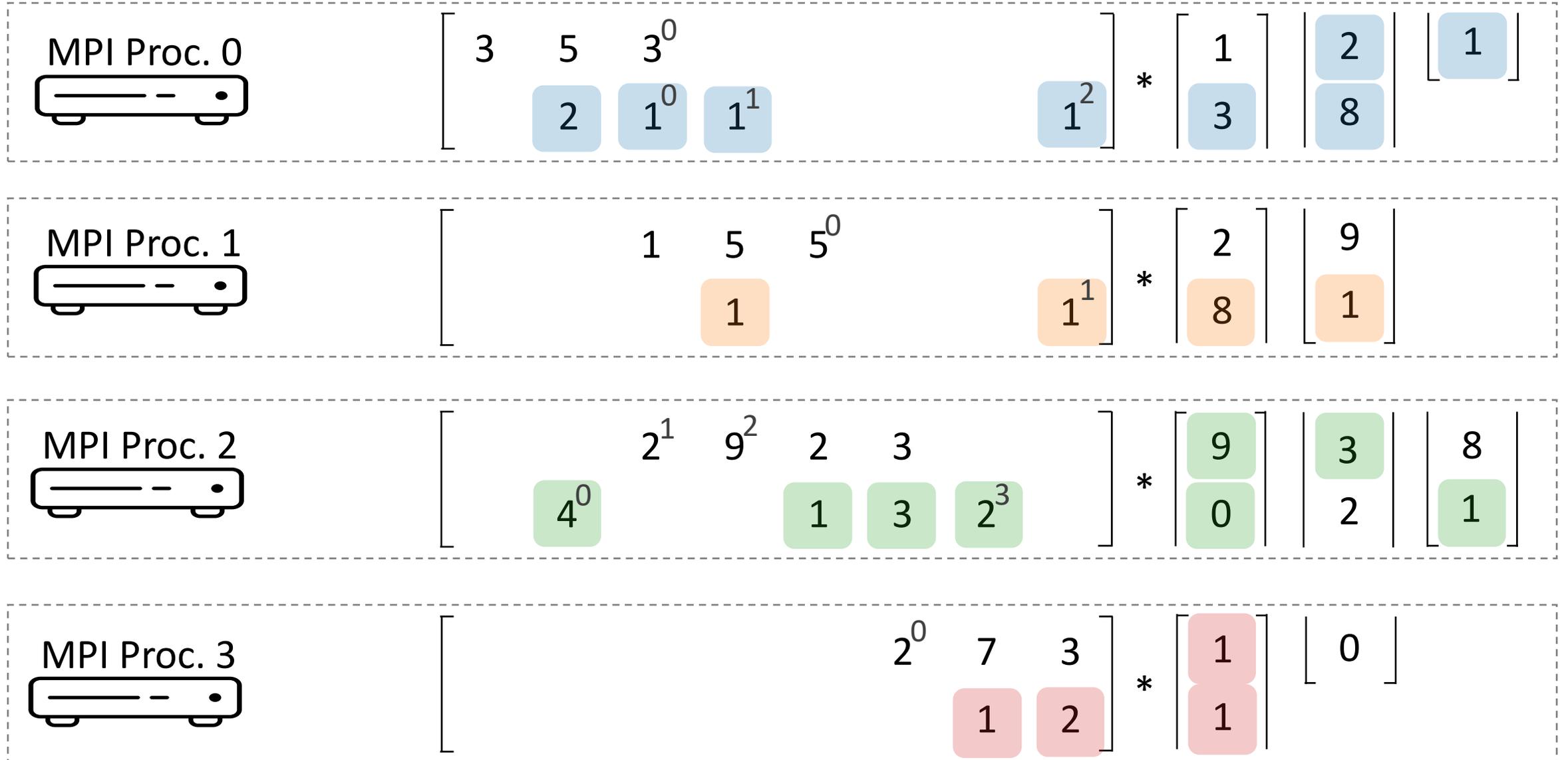
# Distributed SpMV Example



# Distributed SpMV Example

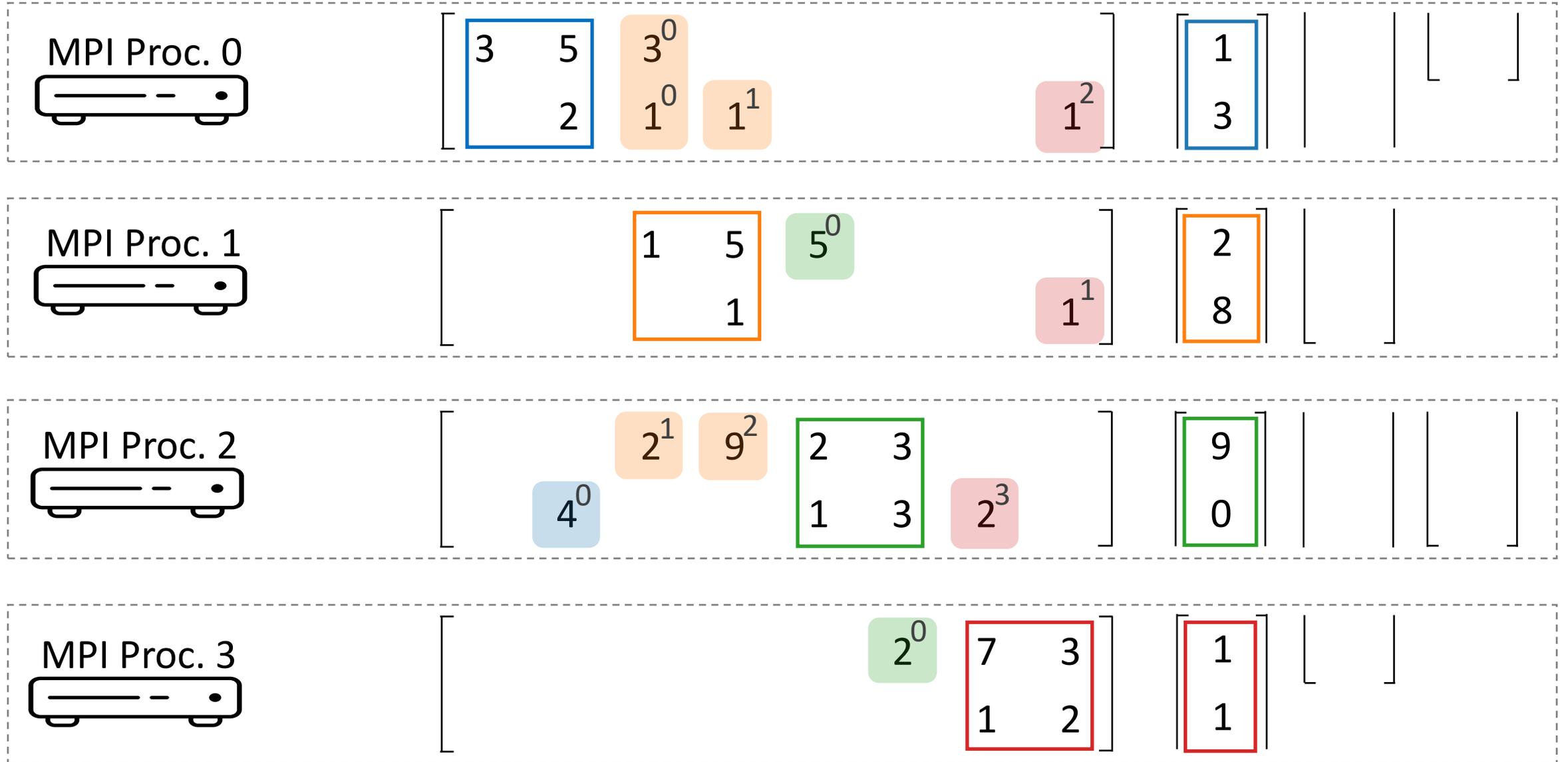


# Distributed SpMV Example

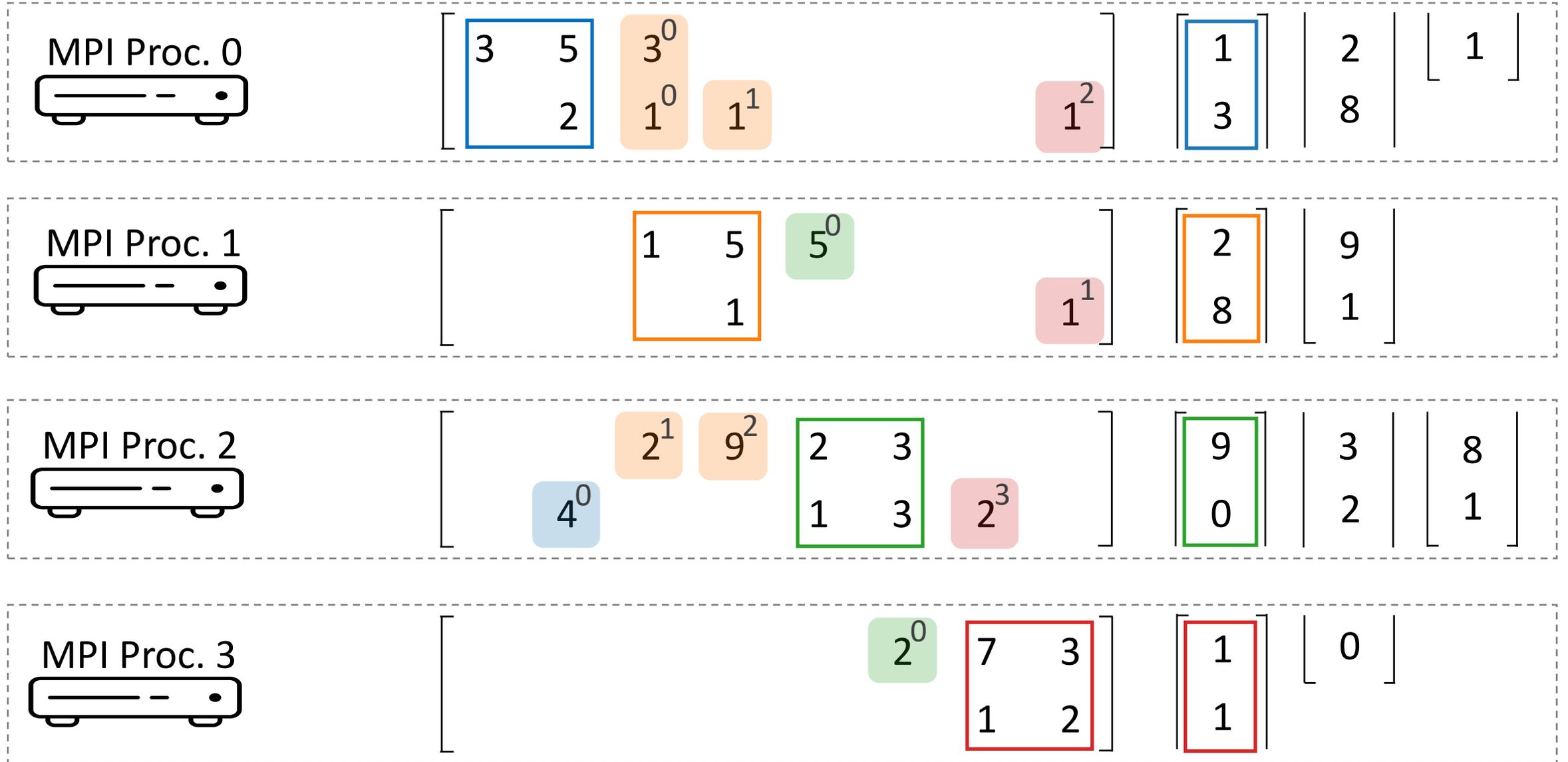


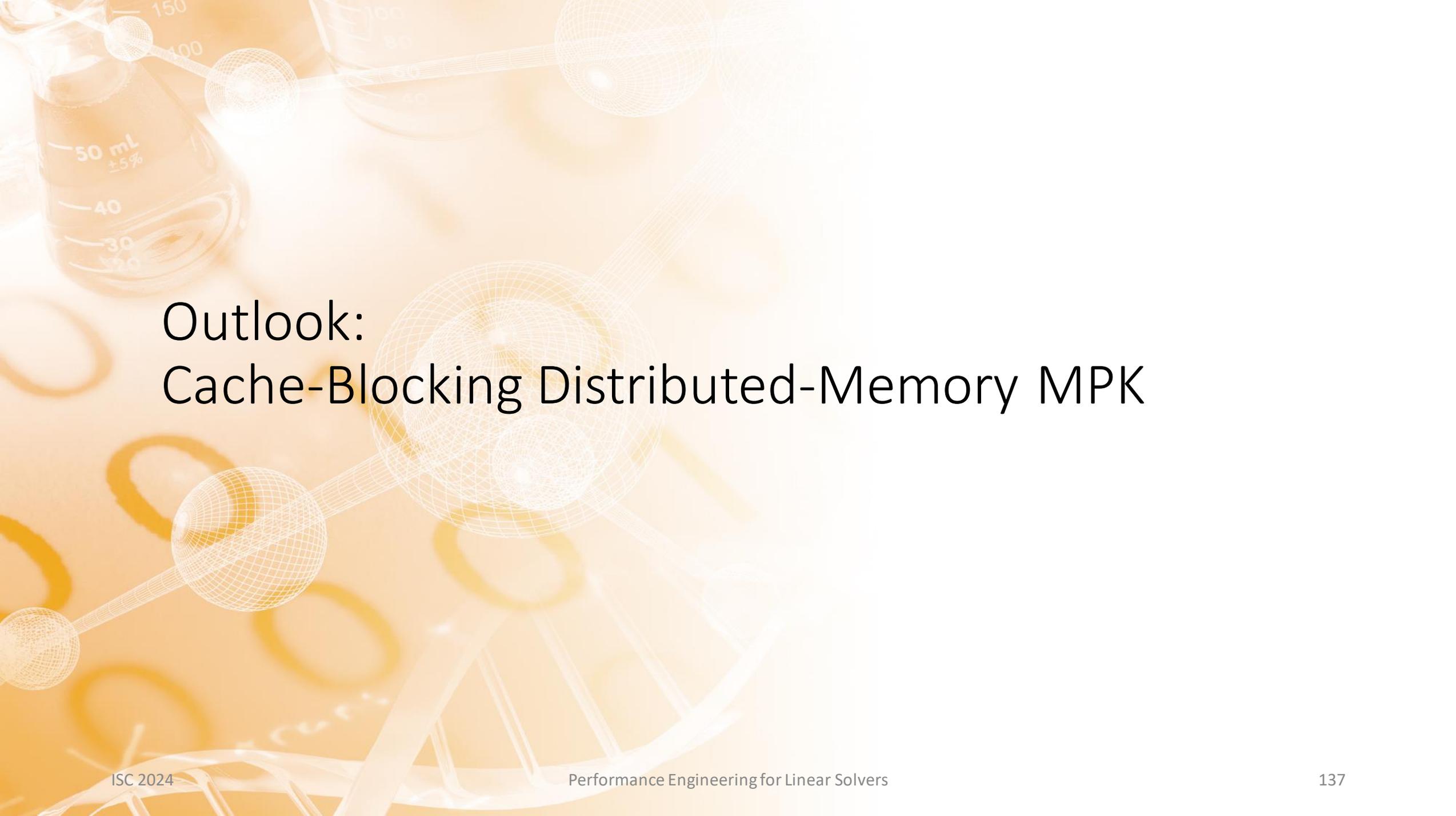


# Distributed SpMV Example



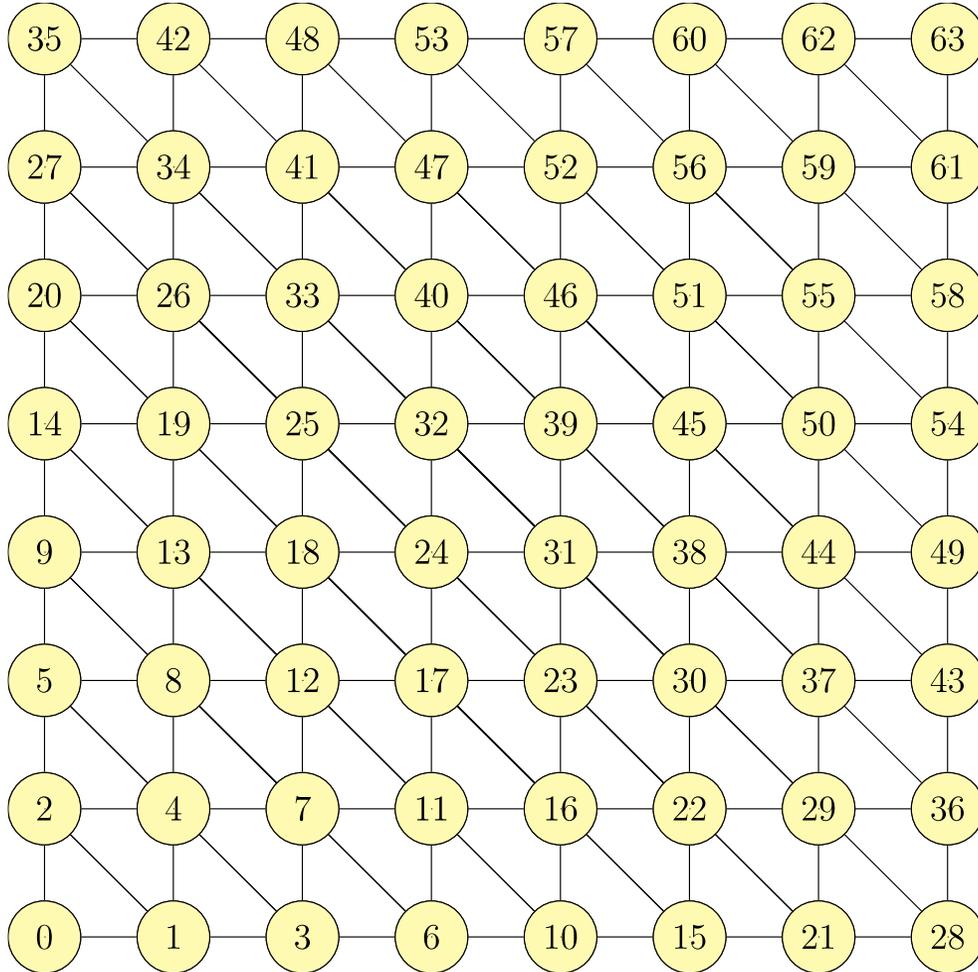
# Distributed SpMV Example



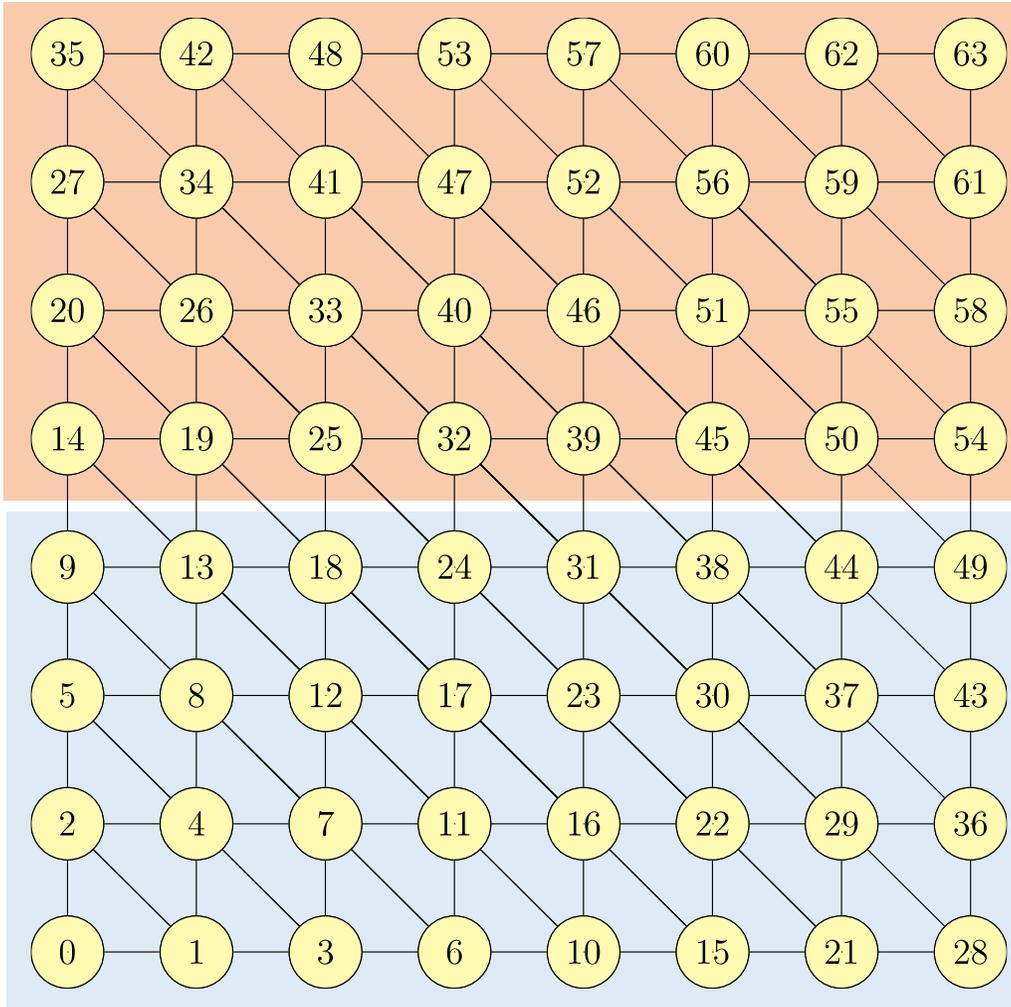


# Outlook: Cache-Blocking Distributed-Memory MPK

# Distributed MPK



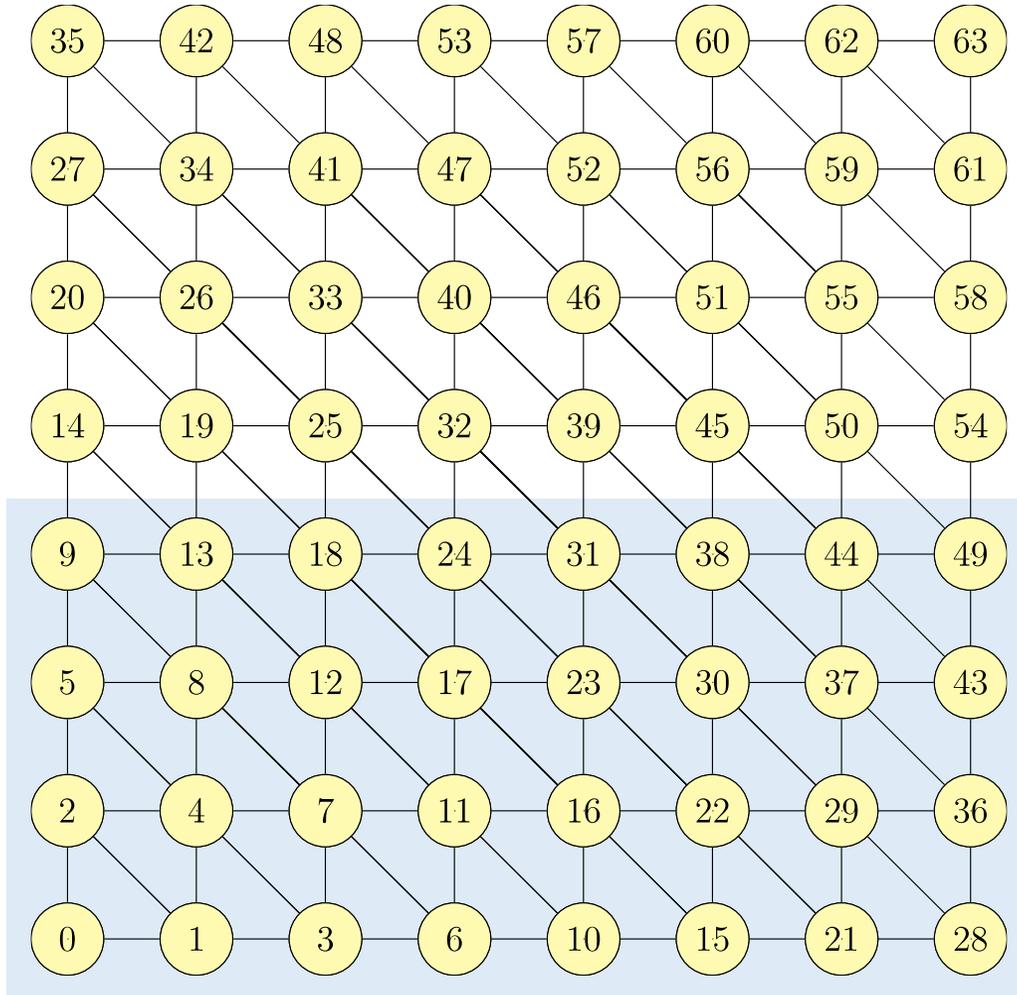
# Distributed MPK



Proc 1

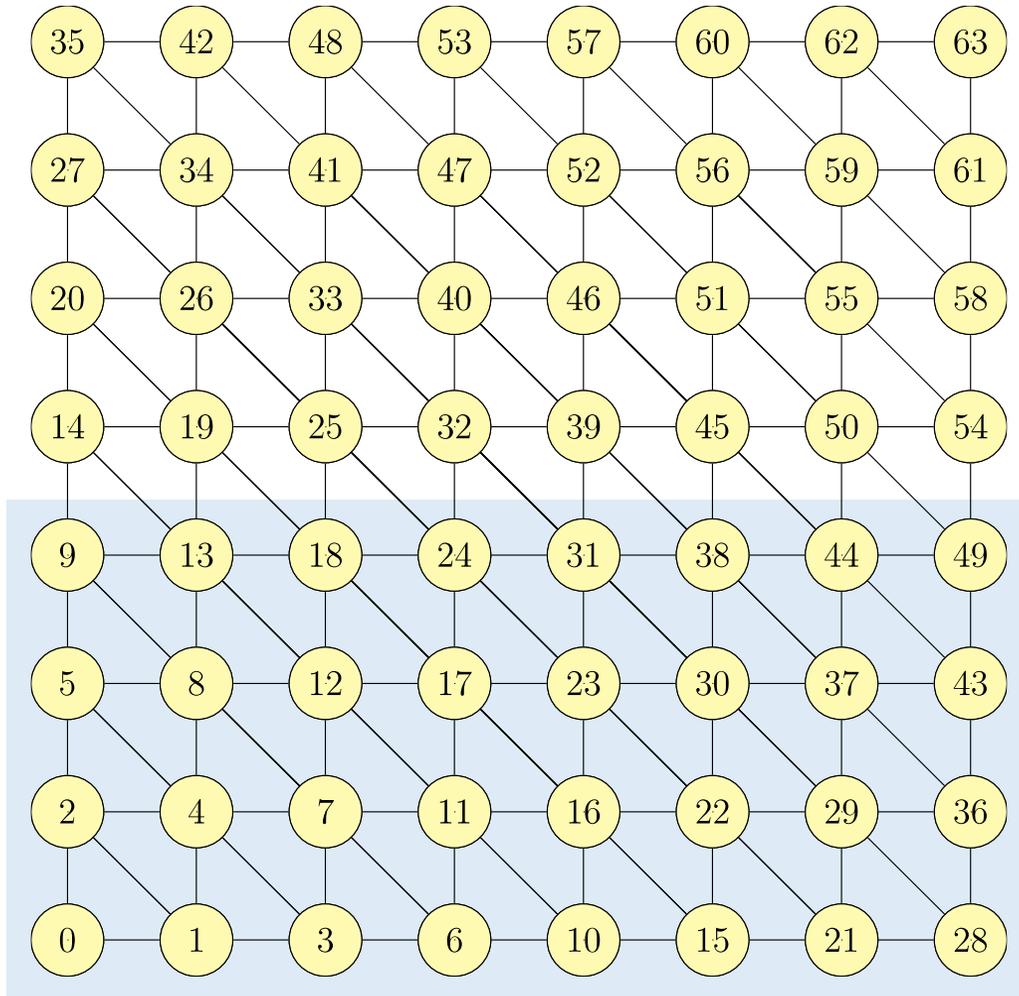
Proc 0

# Distributed MPK



Proc 0

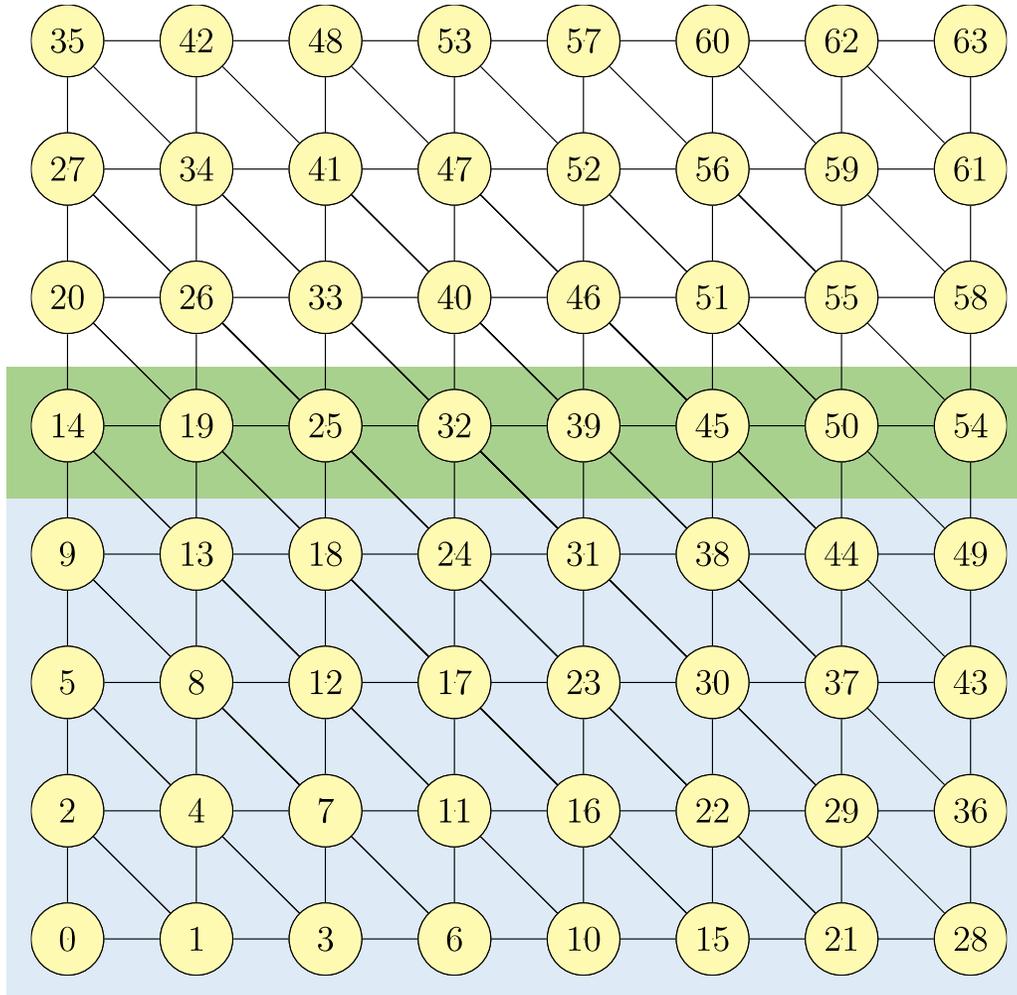
# Distributed MPK



Computing  $Ax$  on Proc 0  
requires neighbors of Proc 0.

Proc 0

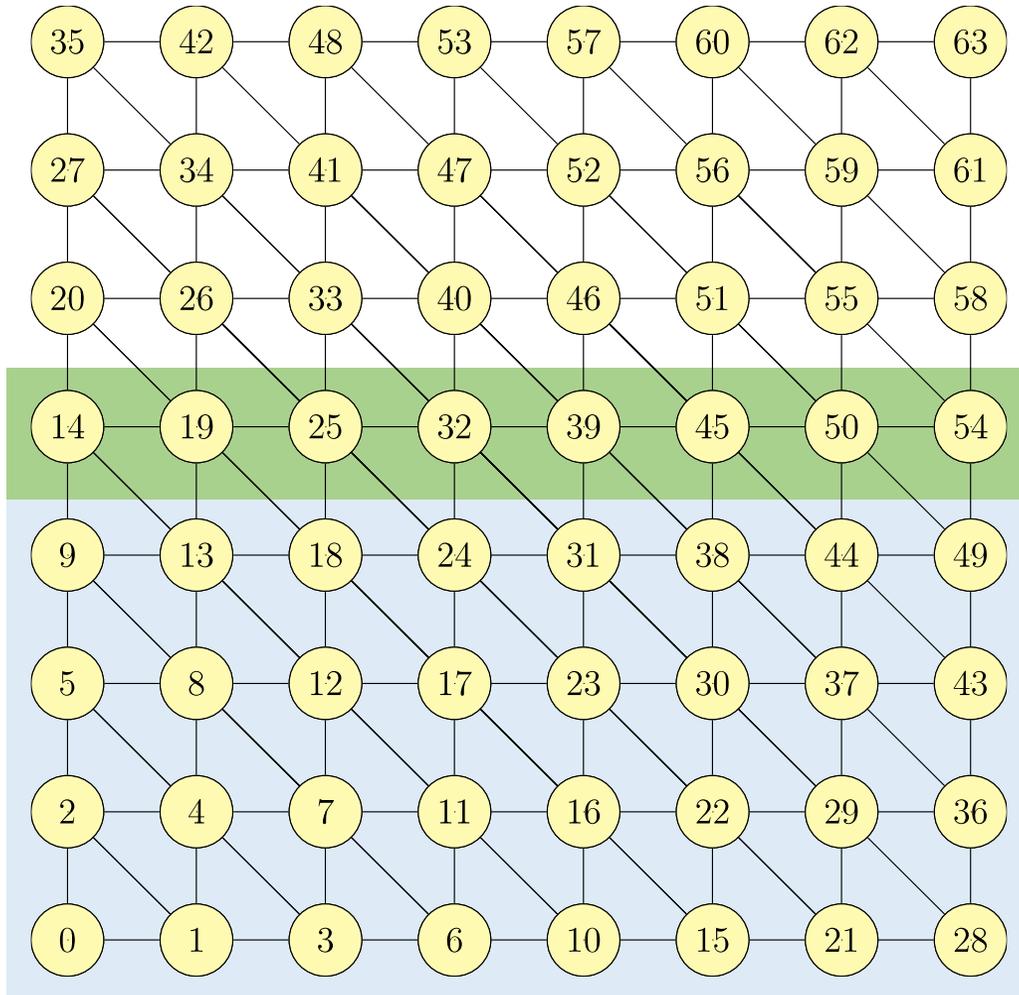
# Distributed MPK



Computing  $Ax$  on Proc 0  
requires neighbors of Proc 0.

Proc 0

# Distributed MPK

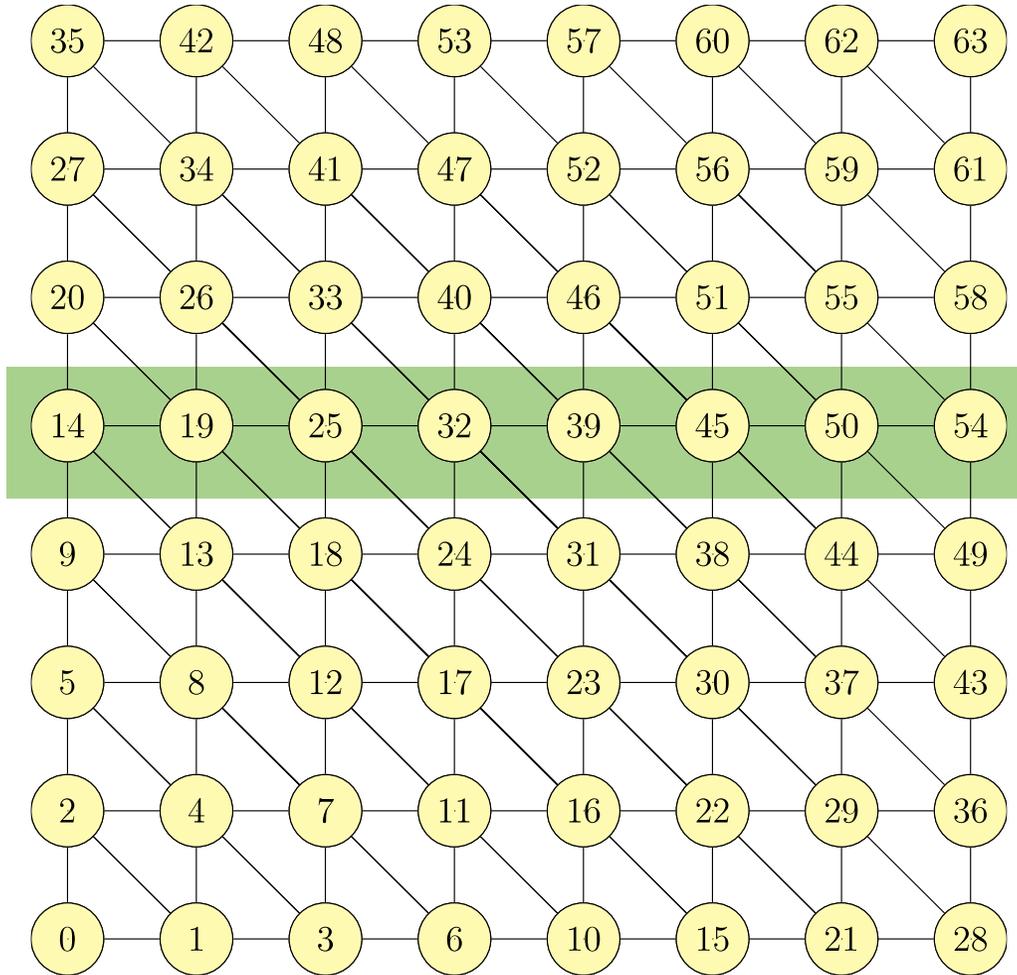


Computing  $Ax$  on Proc 0  
requires neighbors of Proc 0.

How about computing  $A^p x$ ?

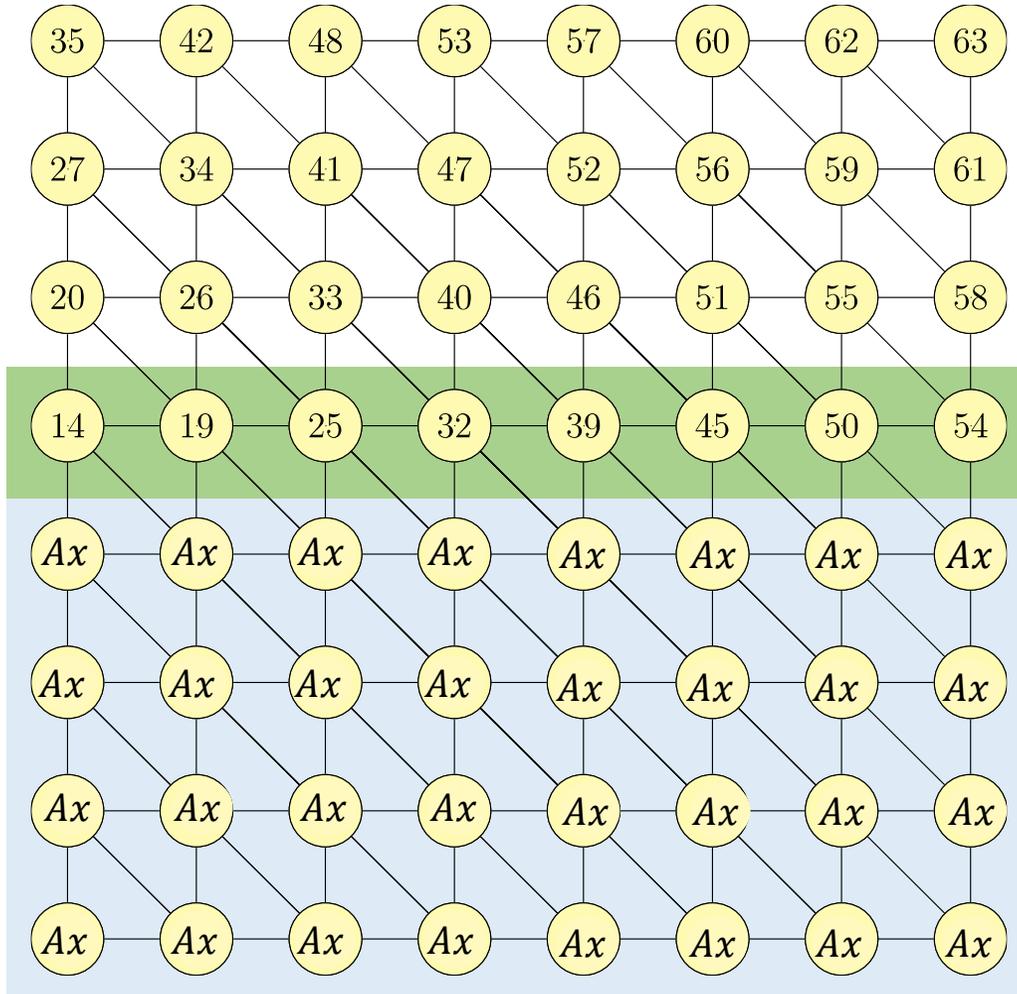
Proc 0

# Distributed MPK



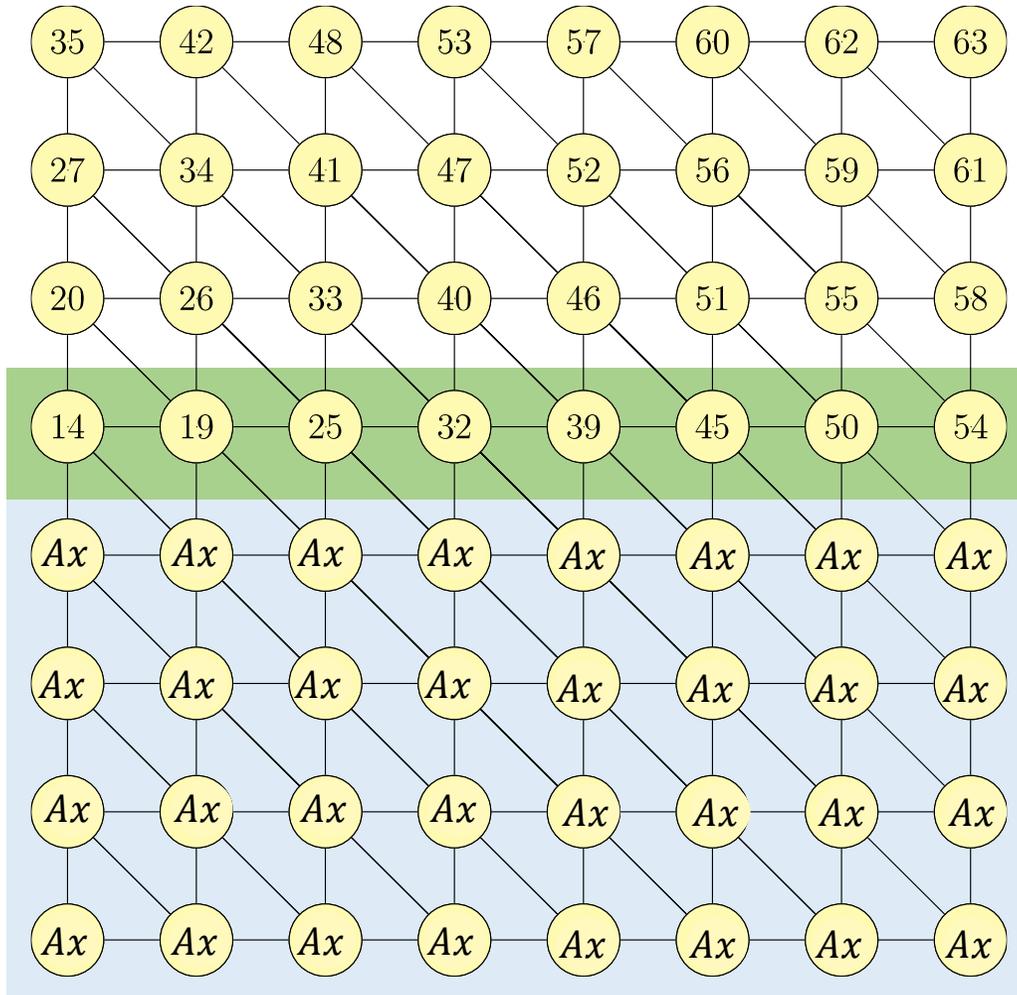
Proc 0

# Distributed MPK



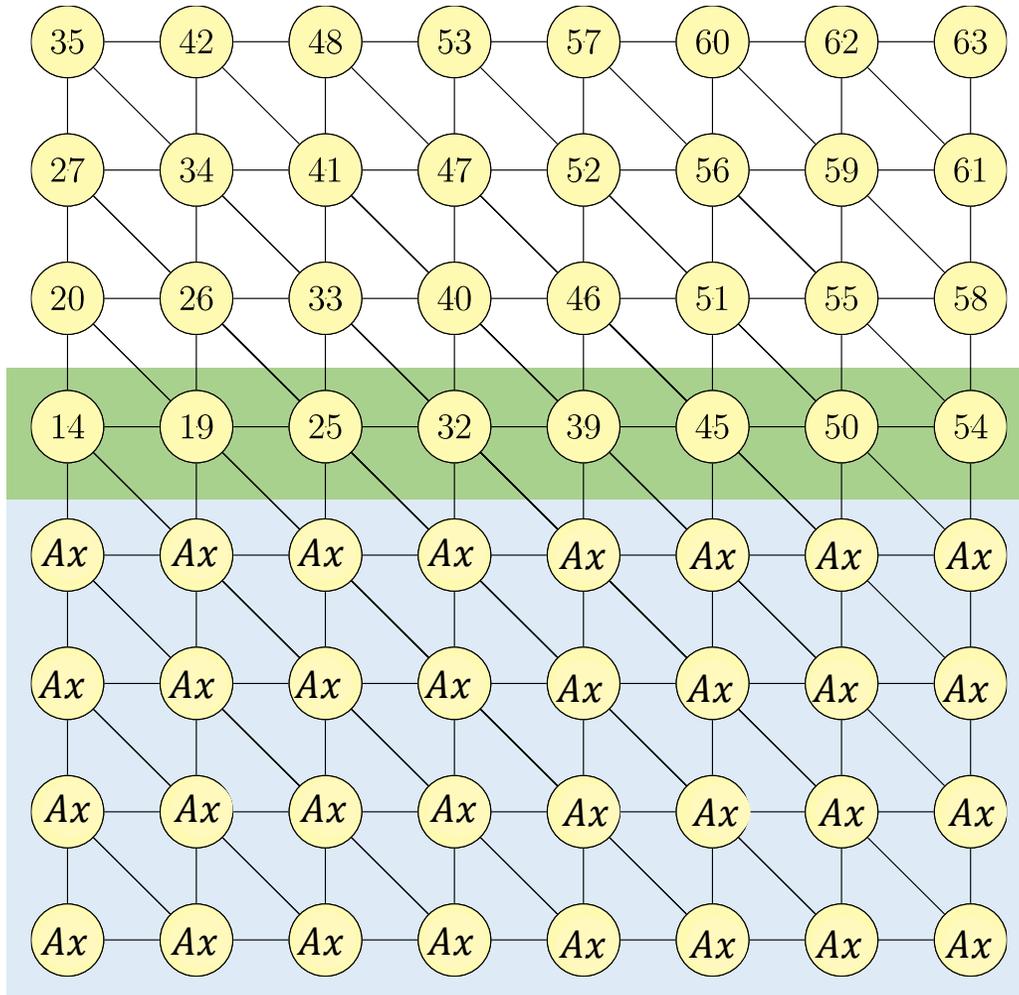
Proc 0

# Distributed MPK



Proc 0

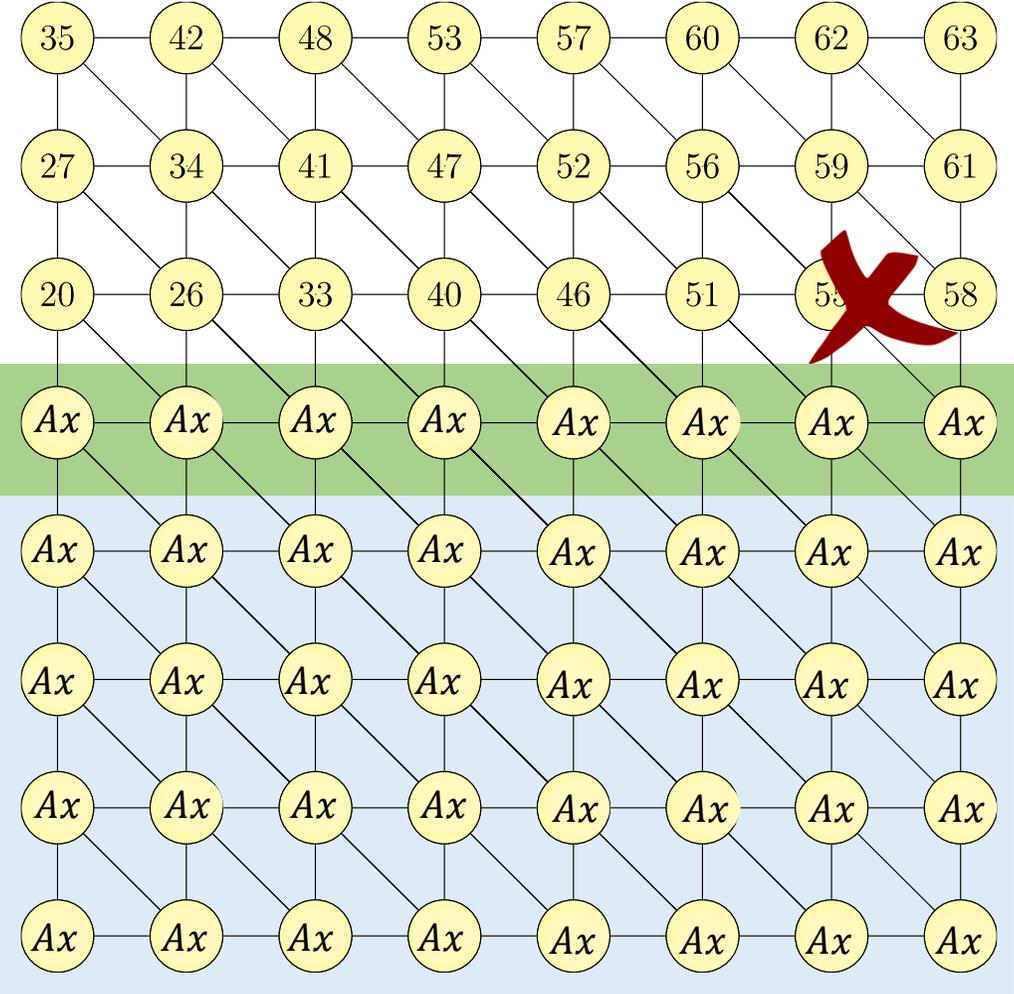
# Distributed MPK



To compute  $A^2 x$  we need  $Ax$  on the neighbors.

Proc 0

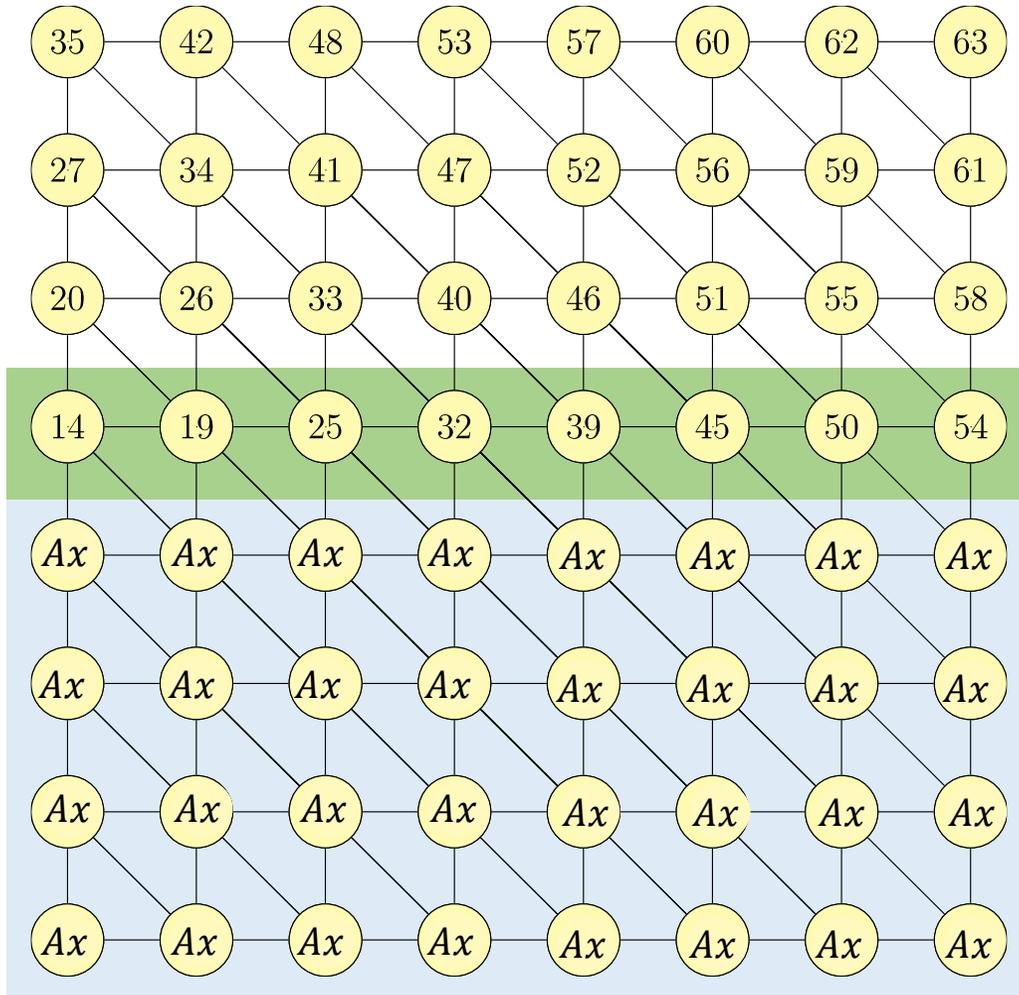
# Distributed MPK



To compute  $A^2x$  we need  $Ax$  on the neighbors.

Proc 0

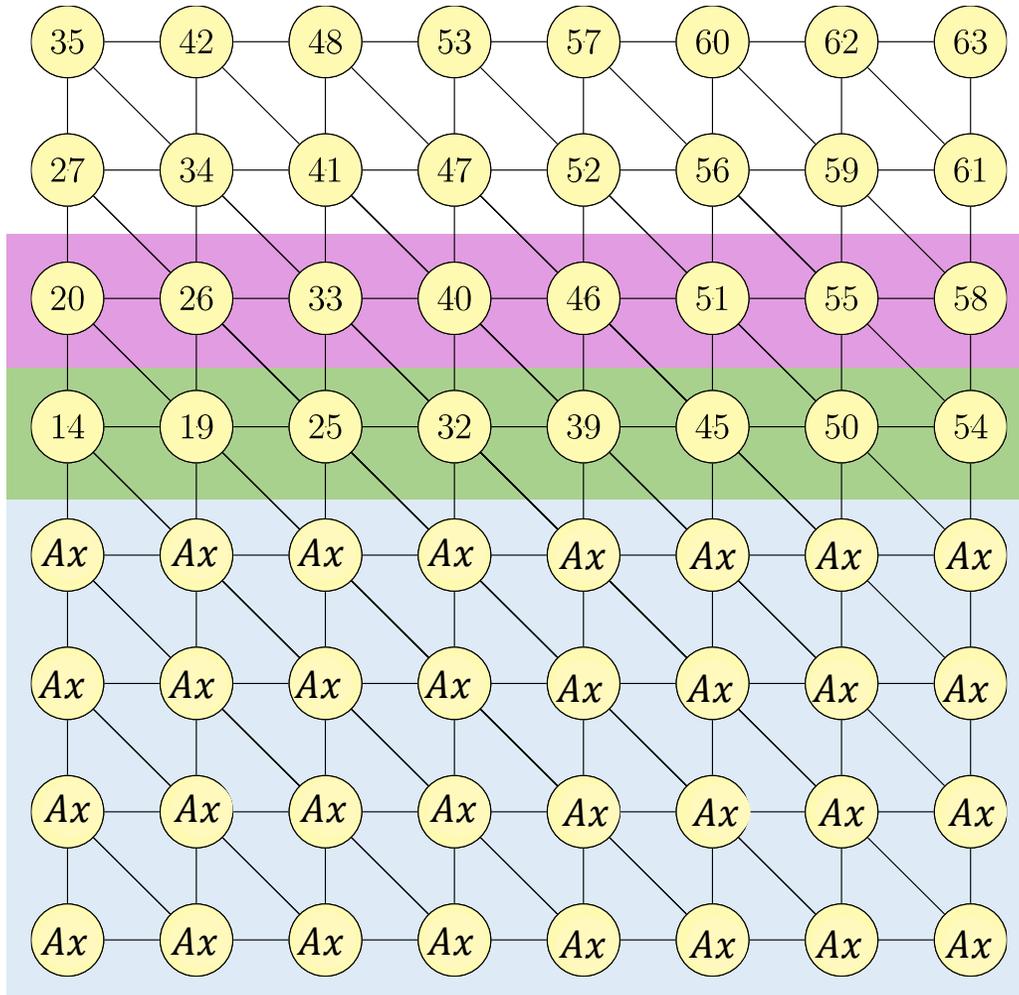
# Distributed MPK



To compute  $A^2 x$  we need  $Ax$  on the neighbors.

Proc 0

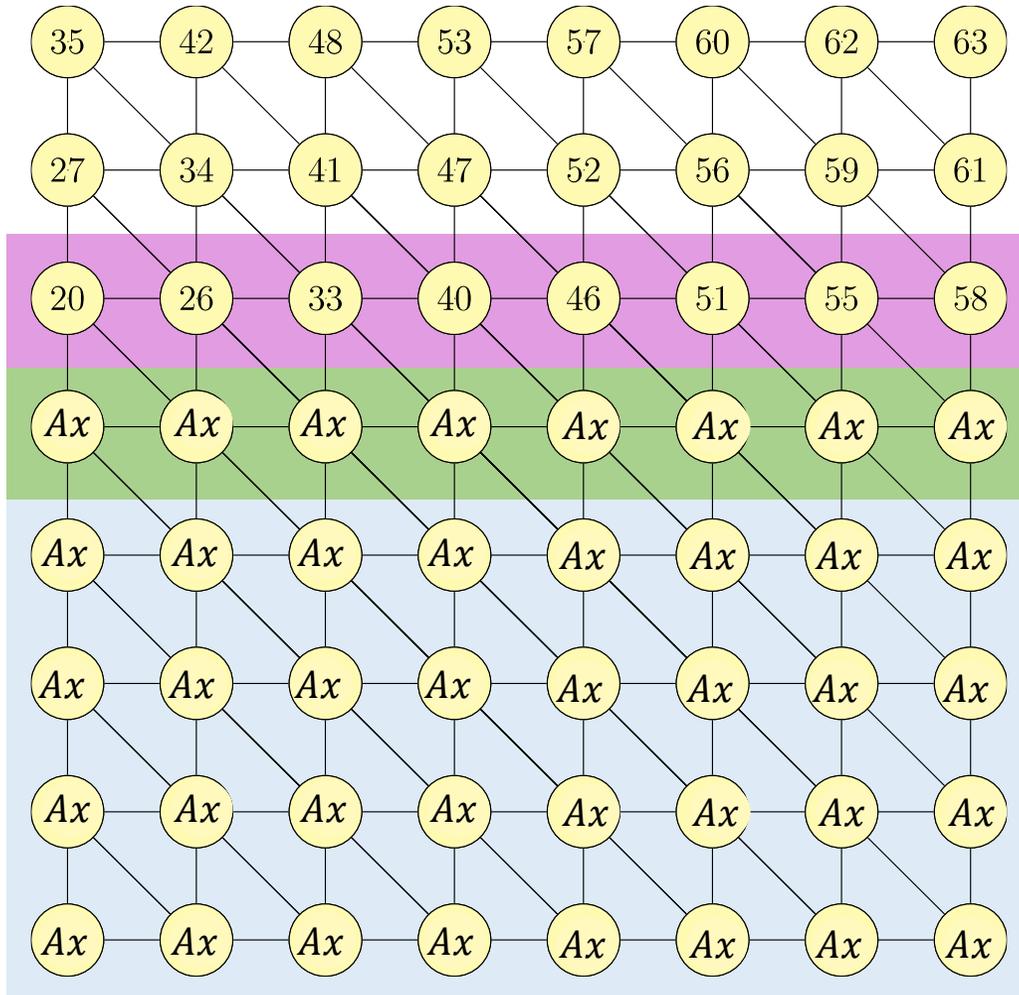
# Distributed MPK



To compute  $A^2 x$  we need  $Ax$  on the neighbors.

Proc 0

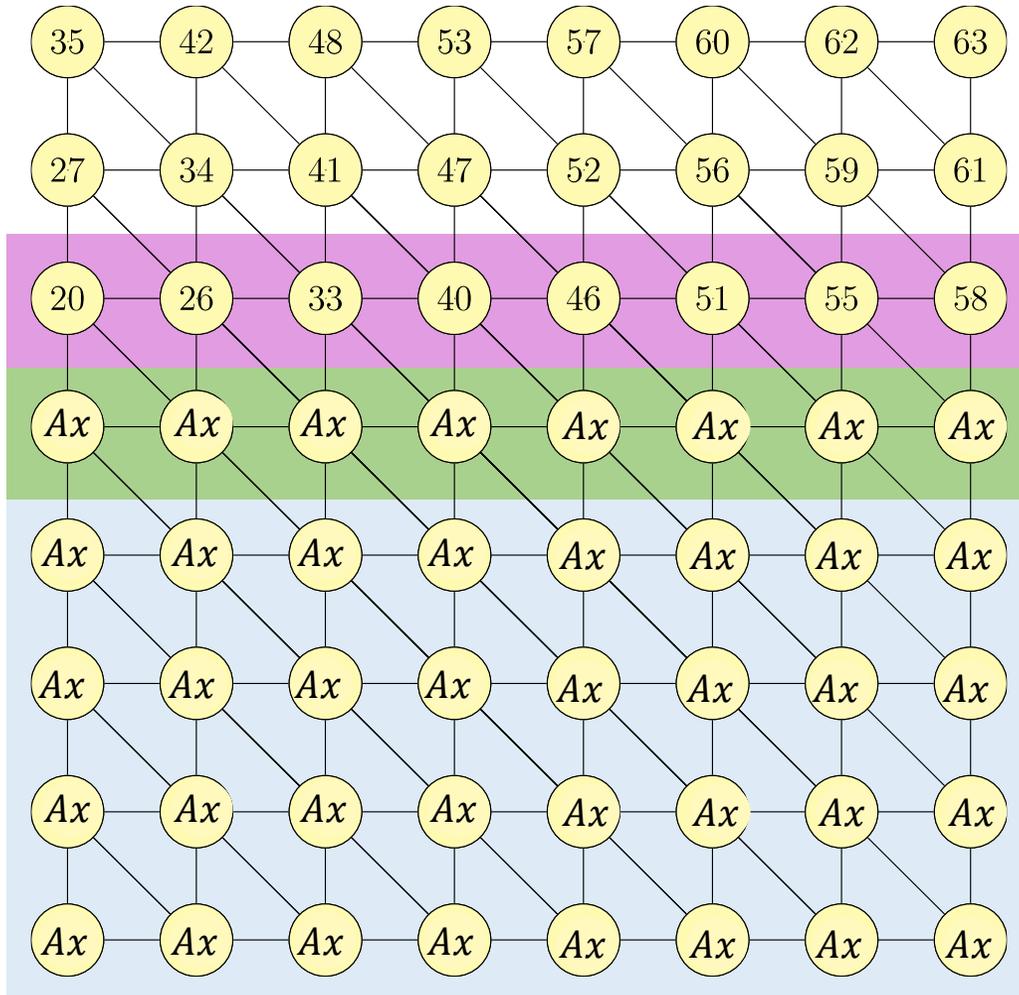
# Distributed MPK



To compute  $A^2x$  we need  $Ax$  on the neighbors.

Proc 0

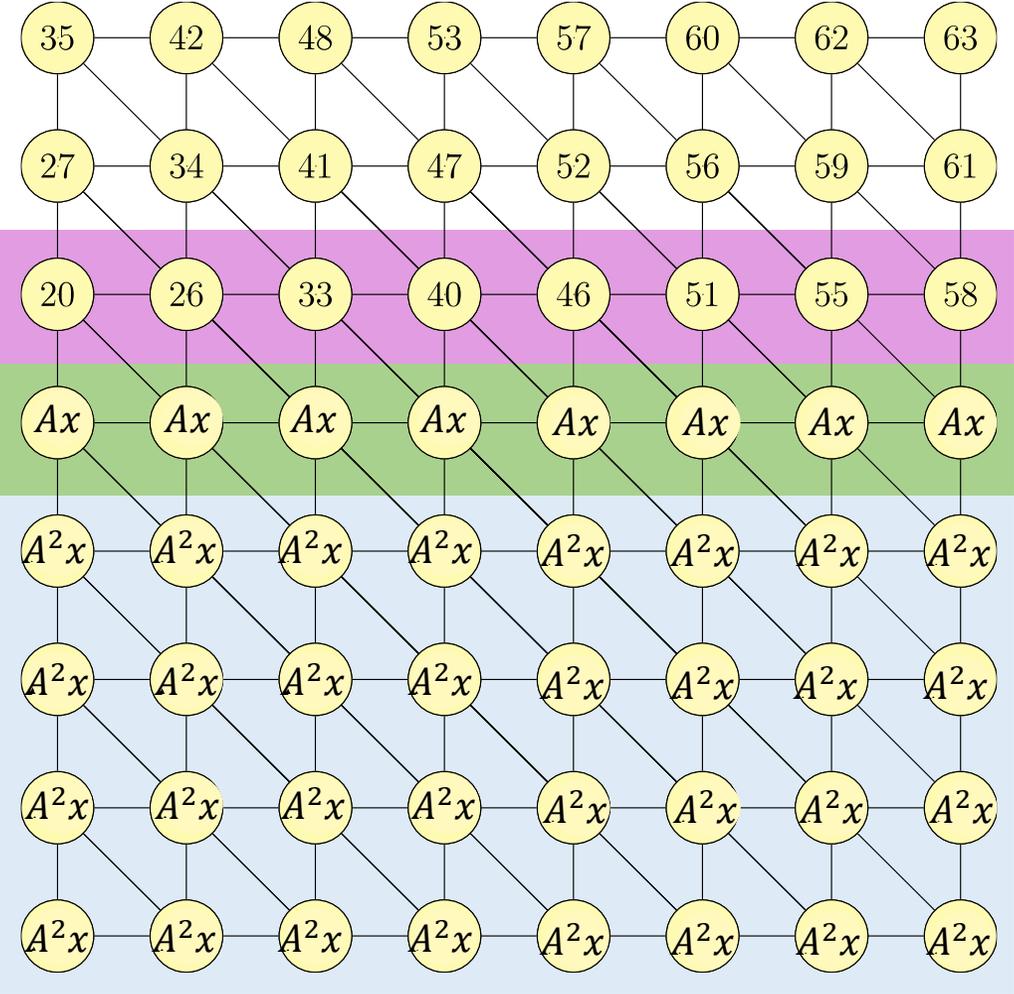
# Distributed MPK



To compute  $A^2 x$  we need  $Ax$  on the neighbors.

Proc 0

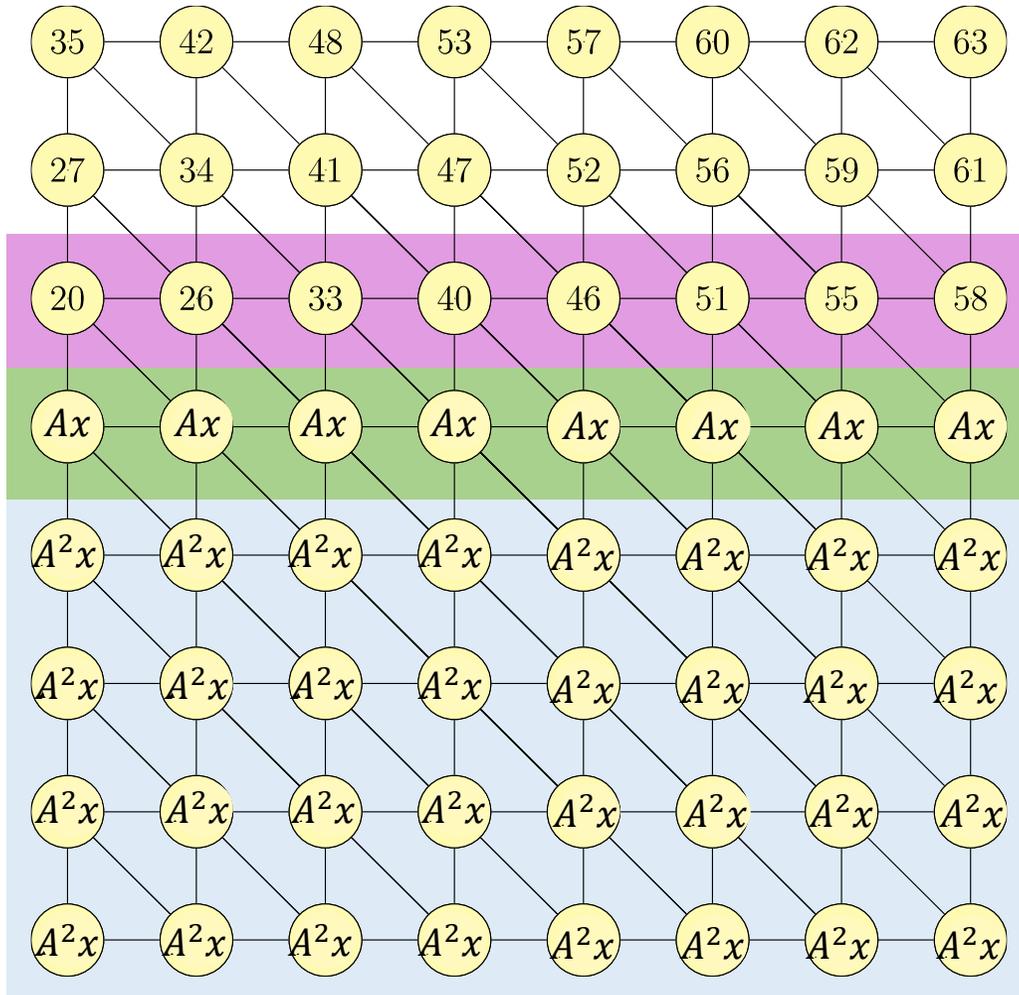
# Distributed MPK



To compute  $A^2x$  we need  $Ax$  on the neighbors.

Proc 0

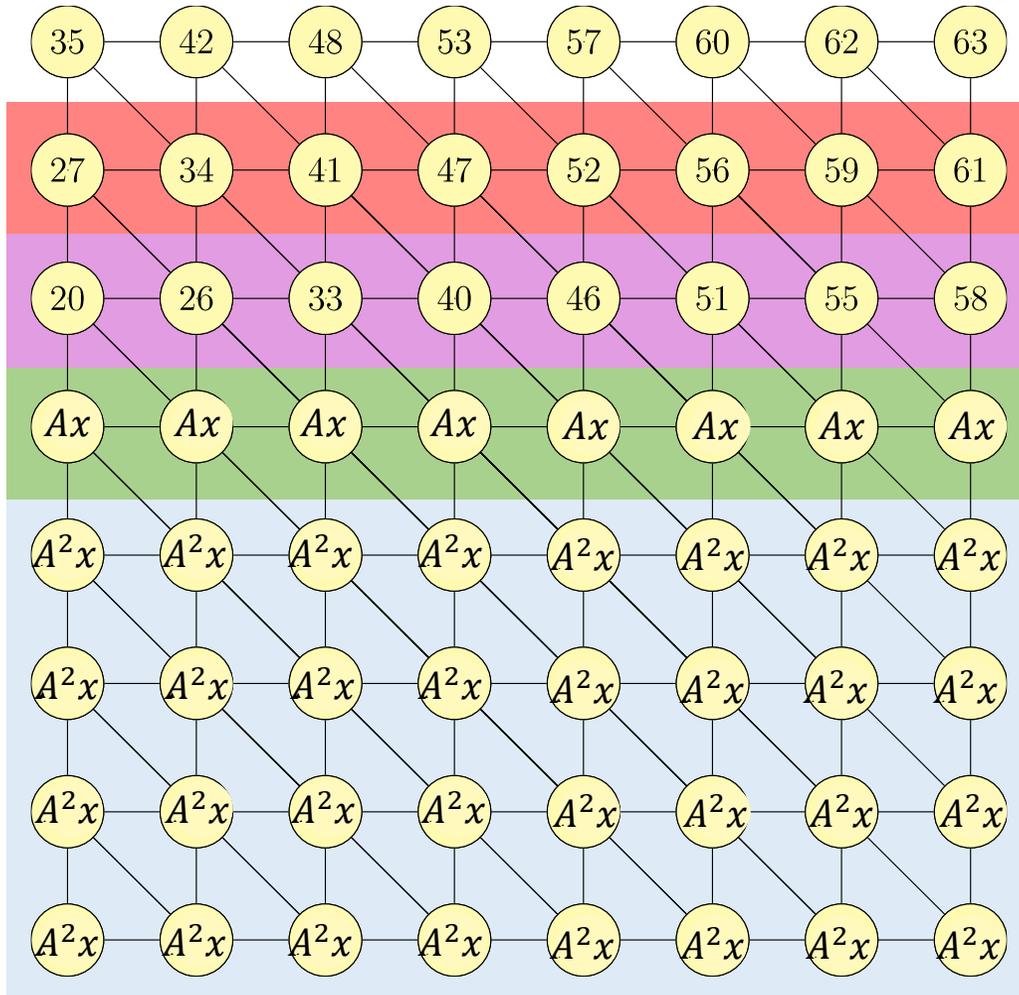
# Distributed MPK



To compute  $A^3 x$  we need  $A^2 x$  on the neighbors.

Proc 0

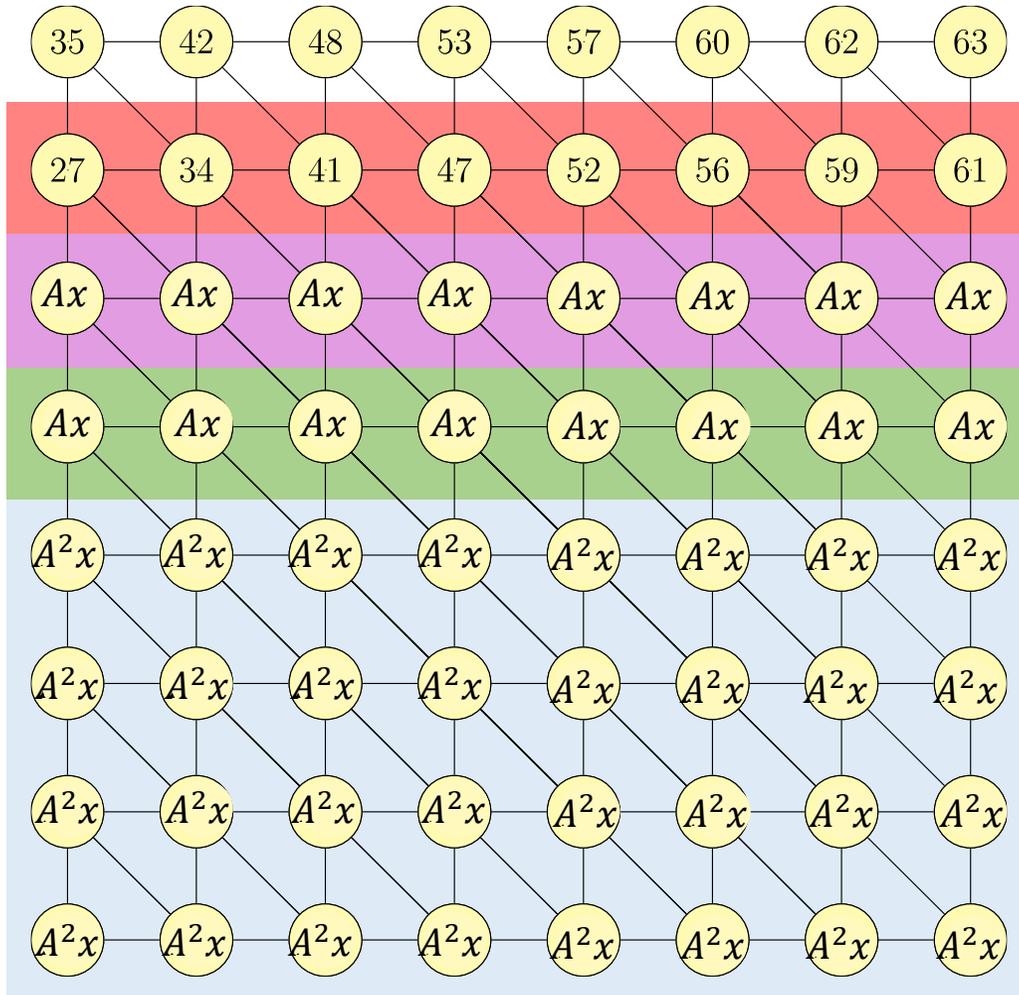
# Distributed MPK



To compute  $A^3 x$  we need  $A^2 x$  on the neighbors.

Proc 0

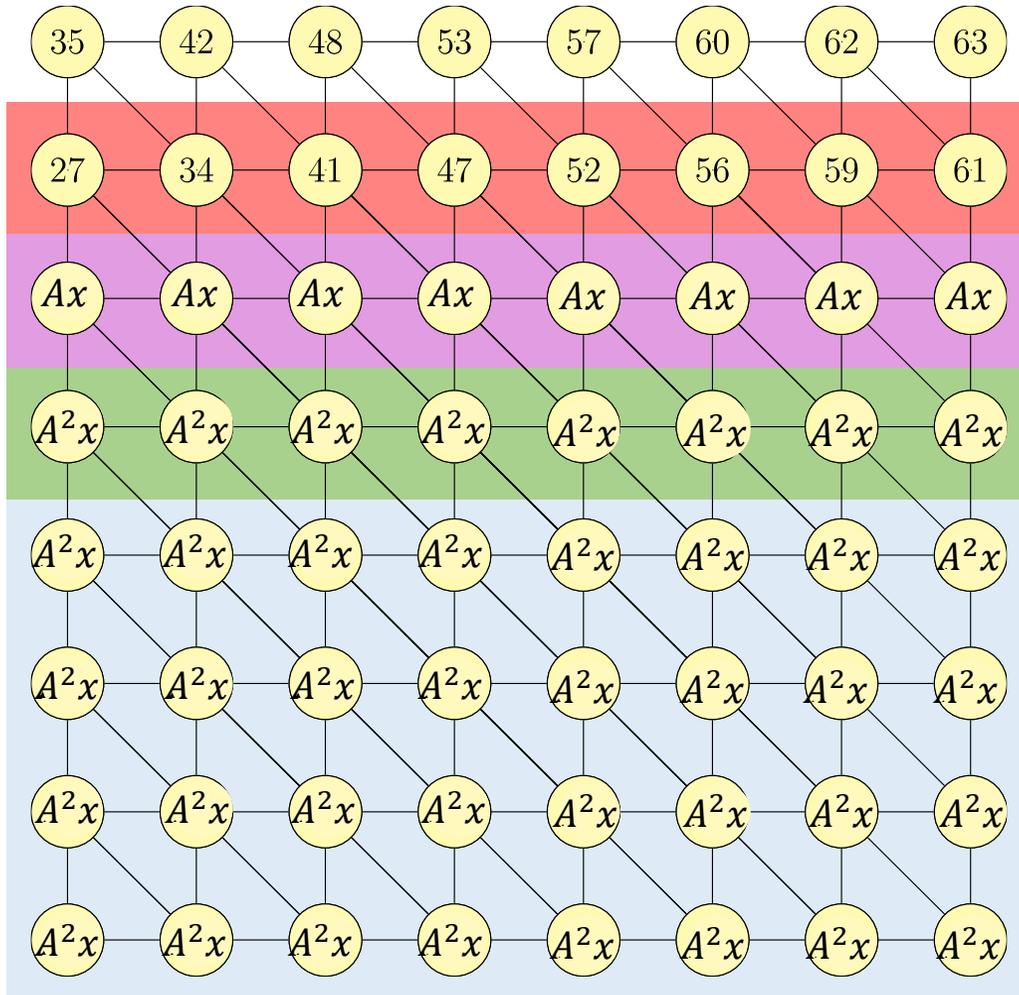
# Distributed MPK



To compute  $A^3 x$  we need  $A^2 x$  on the neighbors.

Proc 0

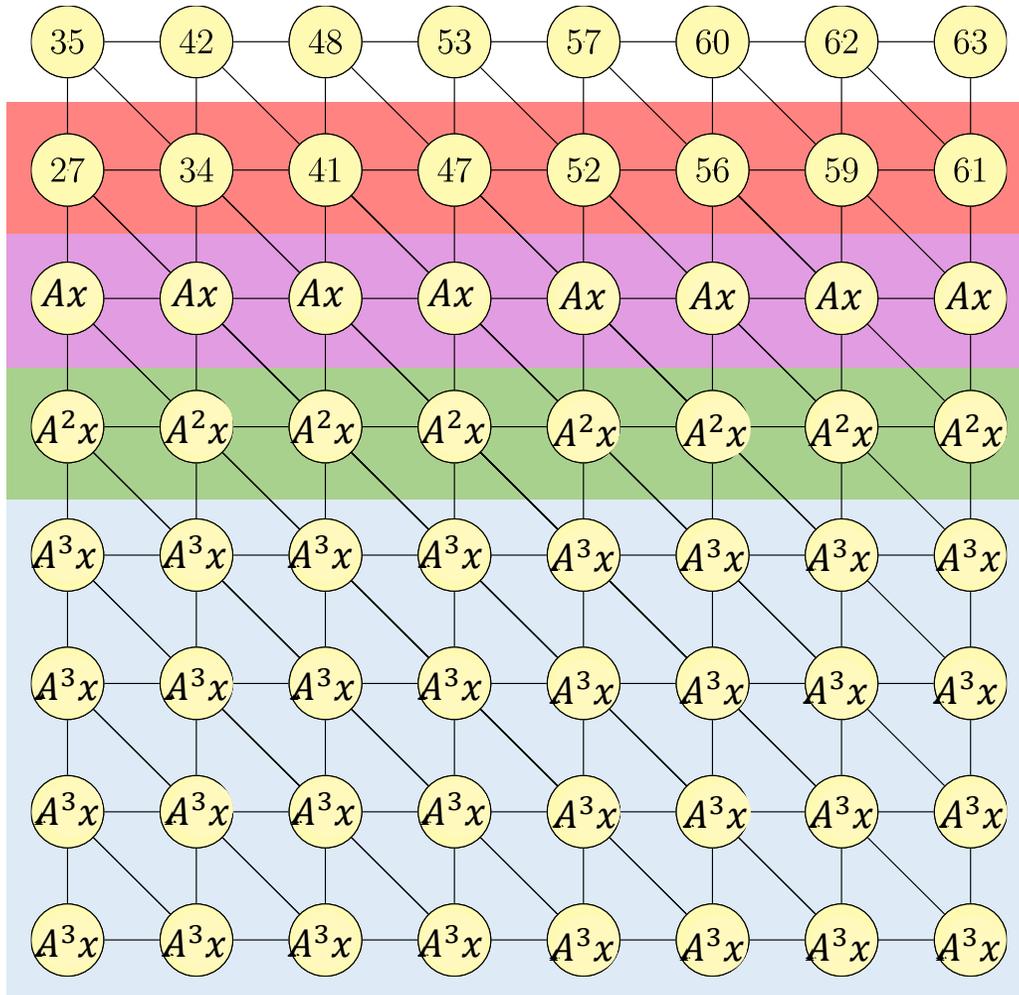
# Distributed MPK



To compute  $A^3 x$  we need  $A^2 x$  on the neighbors.

Proc 0

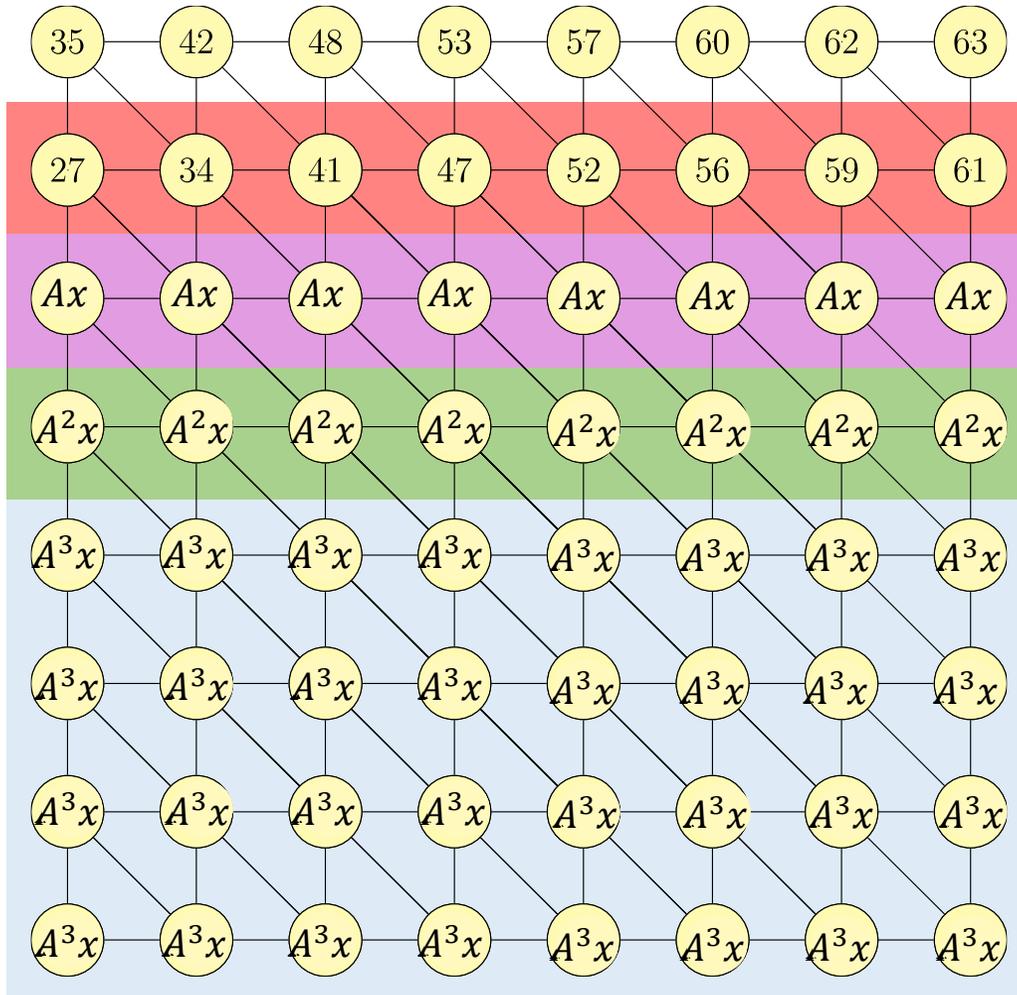
# Distributed MPK



To compute  $A^3x$  we need  $A^2x$  on the neighbors.

Proc 0

# Distributed MPK

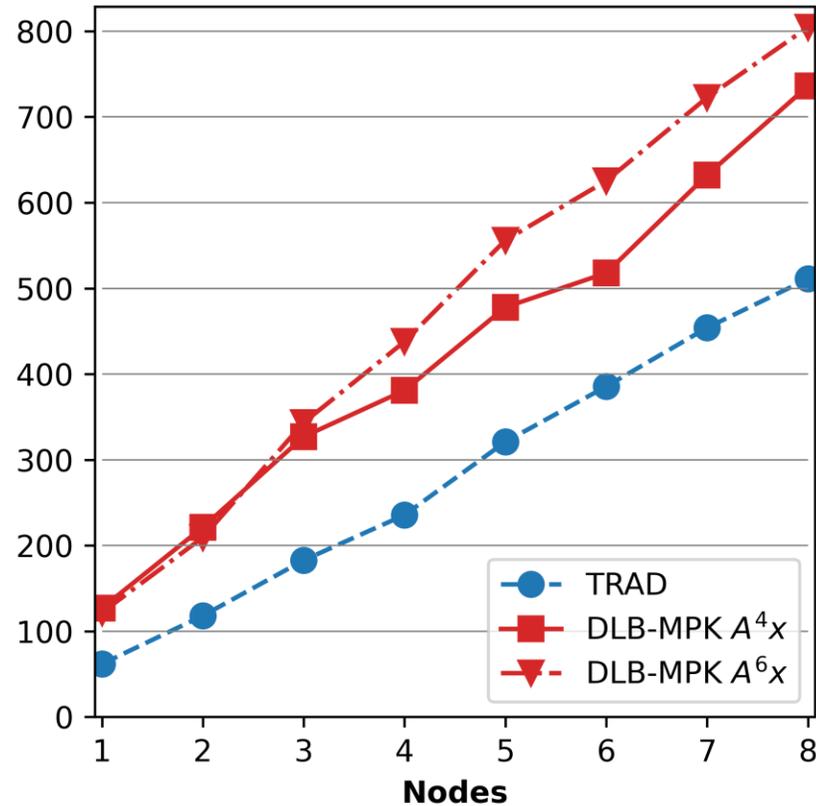
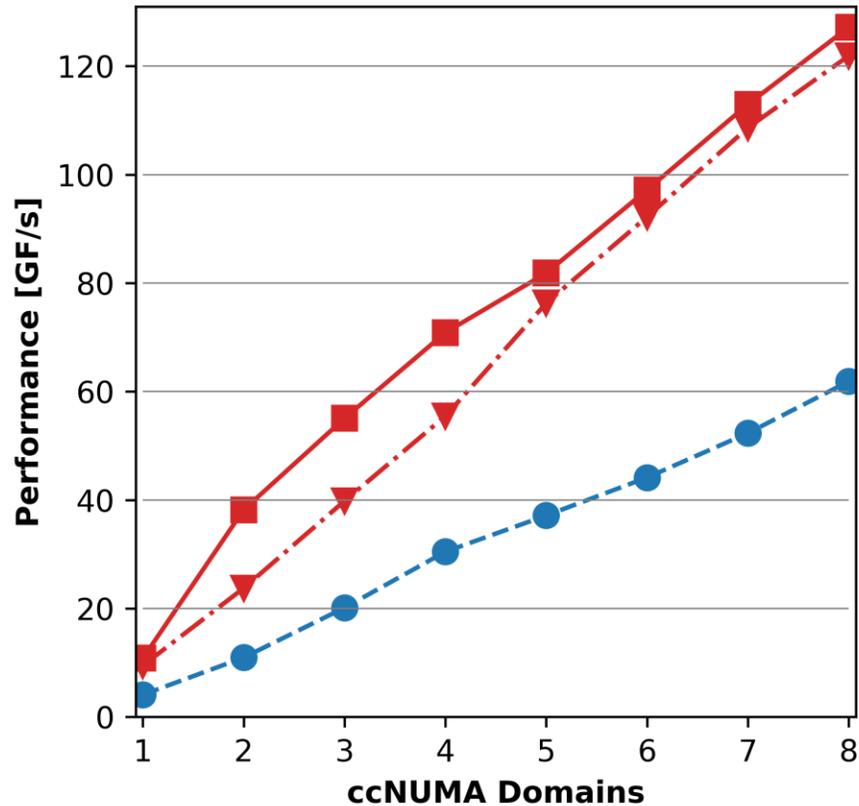


To compute  $A^3x$  we need  $A^2x$  on the neighbors.

In general, to compute  $A^p x$  we need  $p$  neighbors.

Proc 0

# Distributed MPK



It works!

No redundant work  
and/or extra  
communication  
required, see upcoming  
paper.\*

\* D. Lacey, C. Alappat, F. Lange, G. Hager, and G. Wellein: *Cache Blocking of Distributed-Memory Parallel Matrix Power Kernels*, to be submitted.

# Tutorial conclusions

- Memory bandwidth limitations are ubiquitous in sparse linear solvers
- SpMV performance depends on the storage format
- Roofline is an indispensable tool for performance analysis
- Time to solution is a fusion of flop/s performance and fast convergence
- Matrix powers can be optimized for better cache reuse

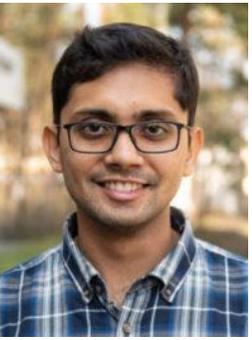


# Appendix

# Performance Engineering for Linear Solvers

This tutorial covers code analysis, performance modeling, and optimization for linear solvers on CPU and GPU nodes. Performance Engineering is often taught using simple loops as instructive examples for performance models and how they can guide optimization; however, full, preconditioned linear solvers comprise multiple back-to-back loops enclosed in an iteration scheme that is executed until convergence is achieved. Consequently, the concept of “optimal performance” has to account for both hardware resource efficiency and iterative solver convergence. We convey a performance engineering process that is geared towards linear iterative solvers. After introducing basic notions of hardware organization and storage for dense and sparse data structures, we show how the Roofline performance model can be applied to such solvers in predictive and diagnostic ways and how it can be used to assess the hardware efficiency of a solver, covering important corner cases such as pure memory boundedness. Then we advance to the structure of preconditioned solvers, using the Conjugate Gradient Method (CG) algorithm as a leading example. Hotspots and bottlenecks of the complete solver are identified followed by the introduction of advanced performance optimization techniques like preconditioning and cache blocking.

# Christie L. Alappat



Christie Alappat received his master's degree with honors from the Bavarian Graduate School of Computational Engineering, Friedrich-Alexander-Universität Erlangen-Nürnberg. He is in the final stages of completing his doctoral studies under the guidance of Prof. Gerhard Wellein. At the same time, he is currently working at Intel as a math algorithm engineer. His research interests include performance engineering, sparse matrix and graph algorithms, iterative linear solvers, and eigenvalue computation. He is the author of the RACE open-source software framework, which is used to accelerate challenging computations in sparse linear algebra on modern compute devices. He is also the lead author of a paper that received the SIAM Activity Group on Supercomputing (SIAG/SC) Best Paper Prize in 2024.

<https://hpc.fau.de/person/christie-alappat/>

# Jonas Thies



Jonas has more than 20 years of experience in HPC and scientific computing with applications in CFD, climate research and quantum physics. Specifically, he has worked on domain decomposition methods for sparse linear systems, implicit ocean models, sparse eigenvalue problems on heterogeneous supercomputers, code optimization for multi-core CPUs and vector processors, and software and performance engineering for scientific applications.

Jonas has a PhD in applied mathematics (Groningen 2011). He spent two years at the Center for Interdisciplinary Mathematics in Uppsala, after which he moved to Cologne as a Scientific Employee of the German Aerospace Center (DLR) Institute for Software Technology. There he led a research group on parallel numerics from 2017 to 2021. Since June 2021 he is an Assistant Professor at the Delft High Performance Computing Center DHPC, where he coordinates the center's training activities.

<https://www.tudelft.nl/en/eemcs/the-faculty/departments/applied-mathematics/people/dr-j-jonas-thies>

# Hartwig Anzt



Hartwig Anzt is the Chair of Computational Mathematics at the TUM School of Computation, Information and Technology of the Technical University of Munich (TUM) Campus Heilbronn. He also holds a Research Associate Professor position at the Innovative Computing Lab (ICL) at the University of Tennessee (UTK). Hartwig Anzt received a PhD in applied mathematics from the Karlsruhe Institute of Technology (KIT) and specializes in iterative methods and preconditioning techniques for the next generation hardware architectures. He also has a long track record of high-quality development. He is author of the MAGMA-sparse open-source software package and managing lead of the Ginkgo math software library. Hartwig Anzt had served as a PI in the Software Technology (ST) pillar of the US Exascale Computing Project (ECP), including a coordinated effort aiming at integrating low-precision functionality into high-accuracy simulation codes. He also is a PI in the EuroHPC project MICROCARD.

Hartwig Anzt is the main author of more than 100 peer-reviewed publications, part of the scientific committee of international conferences, Associate Editor of the SIAM Journal on Scientific Computing (SISC), Associate Editor of ACM Transactions on Parallel Computing, workshop chair for ISC High Performance 2022, and has been elected as SIAM Activity Group on Supercomputing program manager.

<https://hartwiganzt.github.io/>

# Georg Hager



Georg Hager holds a PhD and a habilitation degree in Computational Physics from the University of Greifswald. Since 2021 he heads the Training and Support Division of the newly founded “Erlangen National High Performance Computing Center (NHR@FAU).” Previously he was a senior researcher in the HPC Services group at Erlangen Regional Computing Center (RRZE), which is part of the University of Erlangen-Nürnberg. Recent research includes architecture-specific optimization strategies for current microprocessors, performance engineering of scientific codes, and analytic modeling of massively parallel programs. His textbook “Introduction to High Performance Computing for Scientists and Engineers” is recommended or required reading in many HPC-related lectures and courses worldwide. He has more than two decades of experience in teaching high performance computing and performance engineering to students and scientists. Together with colleagues from NHR@FAU and other centers, he conducts long-standing series of tutorials on Performance Engineering and Hybrid Programming.

<https://blogs.fau.de/hager>