Lecture 4: Common Algorithmic Themes

October 18, 2023

How to Write Efficient Parallel Programs?

General principles

- **Data locality**: minimize communications by placing data near the CPUs that need them
- **load balancing**: minimize periods of inactivity
- **Overlap communication with computation**: avoid processors sitting idle while transferring data

Load balancing

"Load balancing" = who does what? = affecting tasks to CPU

Predictable workload

$\Rightarrow$ **Static** job affectation

  = Can be determined in advance, fixed over time

- Typical scenario:
  - All data requiring the same amount of computation time

  $\rightarrow$ Distributed by block, cyclic, ...

Unpredictable workload

$\Rightarrow$ **Dynamic** load balancing

  - Affectation of tasks to processes *during* the computation

- ~~Master-slave~~ Boss-worker paradigm
- "Work stealing" paradigm

~~Master-Slave~~ Boss-Worker Model

- The boss knows the data and the work to be done
- Available workers ask for work
- The boss sends tasks (or orders the workers to stop)

Limitations

- The boss needs a lot of RAM if they have to load all the data
- 2 message exchanges per task (A/R) $\longrightarrow$ high granularity
- Too many workers $\rightarrow$ the boss becomes a bottleneck

Advantages :

- Good load balancing, even with heterogeneous resources (or availability/speed that varies with time)
- *Checkpointing* is very easy (checkpoint the boss)

Work-Stealing Model

Principle

- Each processor manages their own work list
  - *A priori* fair initial distribution
- If task list is empty :
  - Choose a *victim* (randomly?)
  - "Steal" a fraction (50% ?) of the victim's remaining work

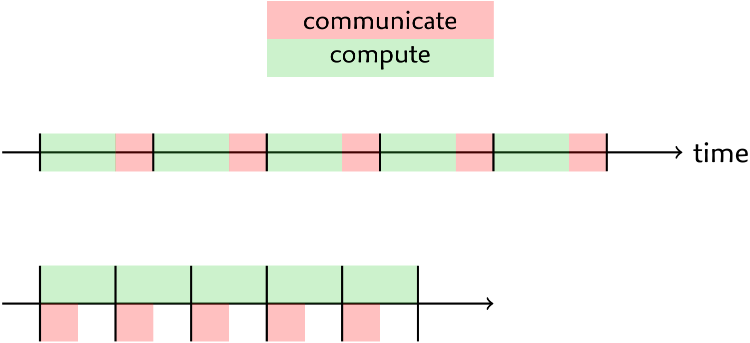
+ Completely symmetrical
  - *"No gods, no masters"* Ⓐ
+ Every process participates in the calculation
  - No ~~parasite~~ bosses twiddling their thumbs

– not easy to detect when the computation is terminated

– difficult to program

– difficult to *checkpoint*

Overlapping Communication and Computation

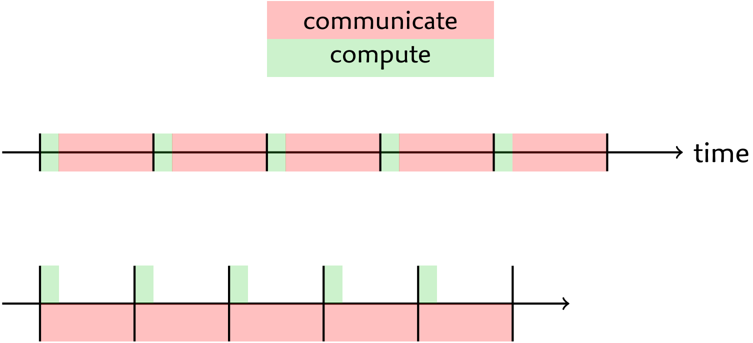


$$T = T_{comp} + T_{comm} \qquad \text{(before)}$$

$$T' = \max(T_{comp}, T_{comm}) \qquad \text{(after)}$$

$$\geq T/2$$

Overlapping Communication and Computation



$$
\begin{aligned}
T &= T_{comp} + T_{comm} && \text{(before)} \\
T' &= \max(T_{comp}, T_{comm}) && \text{(after)} \\
&\geq T/2
\end{aligned}
$$

Data Parallelism

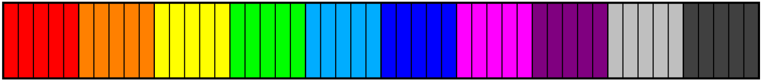
Classic example: *map*

```
for (int i = 0; i < n; i++)
        B[i] = f(A[i], i)
```

- No need for communication / synchronization between processes!
- Data distribution?
- Load balancing?

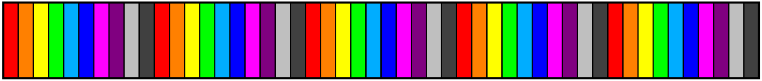
1D Distribution

By blocks:



- Easiest !
- Favored by MPI

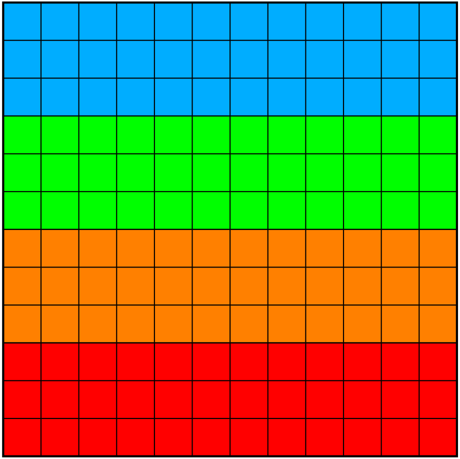
Cyclic



- May improve load balancing
- Also possible with MPI
  - Must create “types” $\rightsquigarrow$ `MPI_Type_vector` ...

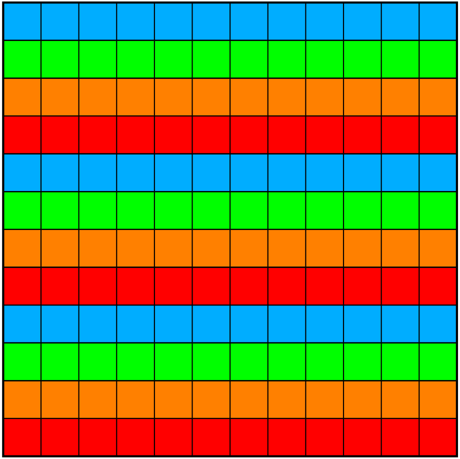
1D Distribution of 2D Data

By blocks:



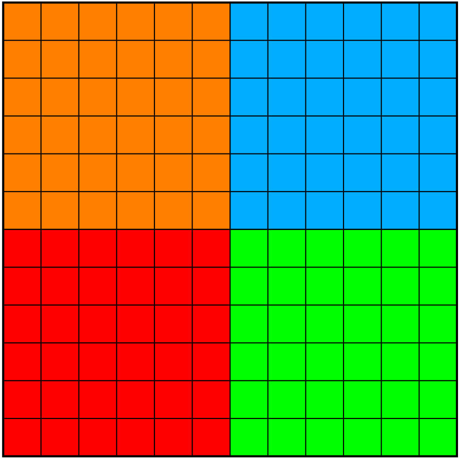
1D Distribution of 2D Data

Cyclic:



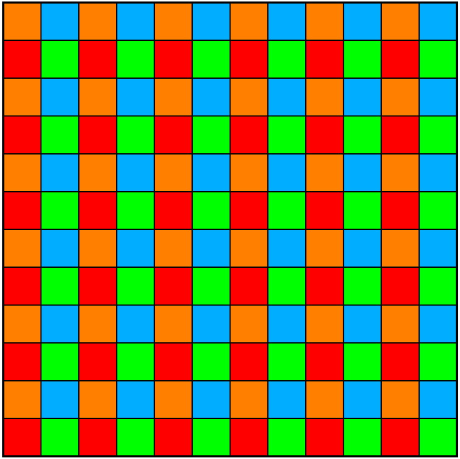
2D Distribution of 2D Data

By blocks:



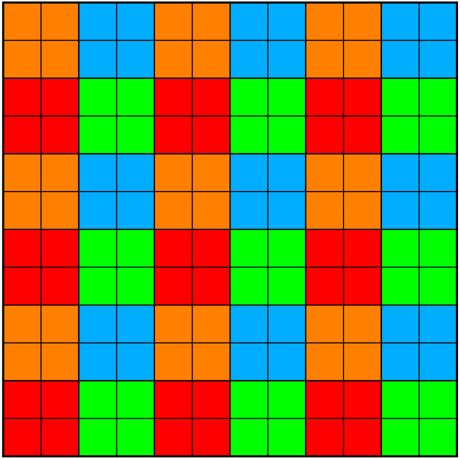
2D Distribution of 2D Data

Cyclic:

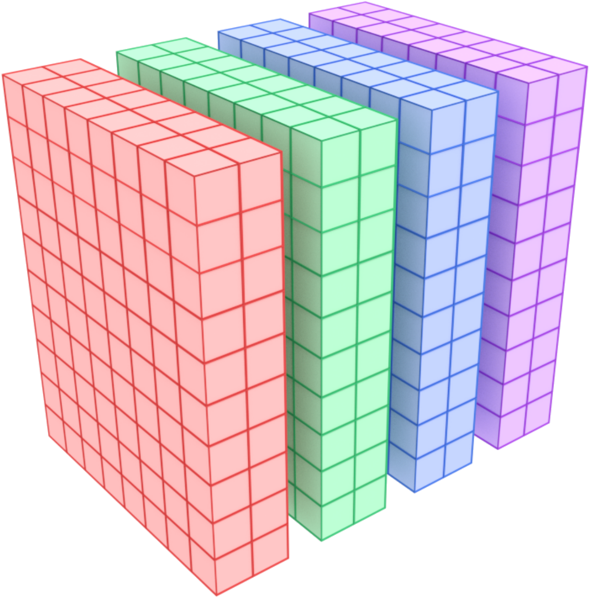


2D Distribution of 2D Data

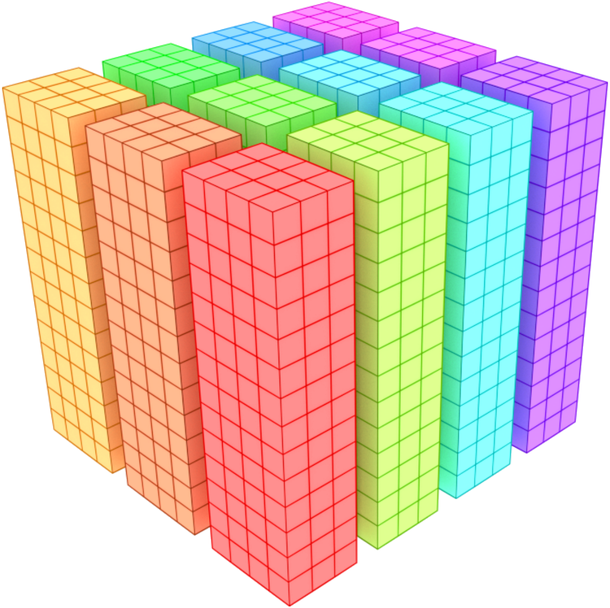
Block-Cyclic:



1D Distribution of 3D Data



2D Distribution of 3D Data



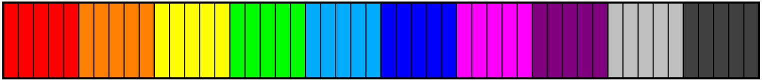
Data Parallelism (continued)

Classic example: *reduce*

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

- Data dependency on `sum`? Easy to bypass
- Distributed memory $\rightarrow$ communications
- Binomial tree algorithm

MPI in action: `reduce`



```c
// Global array of size n
// p processes
int root = 0;
double sum = 0;
for (int i = rank * n / p; (rank + 1) * n / p; i++)
        sum += A[i];
MPI_Reduce(MPI_IN_PLACE, &sum, 1, MPI_DOUBLE, MPI_SUM,
           root, MPI_COMM_WORLD);
```

$$T = \frac{n}{p} + \lceil \log_2 p \rceil (\alpha + \beta)$$

Approximating Solutions of PDEs

Classic example: heat equation

$$\frac{\partial T}{\partial t} = \alpha \nabla^2 T = \alpha \left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{\partial^2 T}{\partial z^2} \right)$$

- Heat diffusion in homogeneous material
- $T(x, y, z, t)$ = temperature in point (x, y, z) at time t

Goal :

- Compute $T(x, y, z, t)$
- Over a finite domain
- $T(x, y, z, 0)$ known (initial conditions)
- Eventual boundary conditions (e.g. $T(0, y, z, t) = cst$)

Approximating Solutions of PDEs

Euler's Method

$$\frac{\partial T}{\partial t} = \alpha \nabla^2 T = \alpha \left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{\partial^2 T}{\partial z^2} \right)$$

Approximation

- Divide time in small intervals

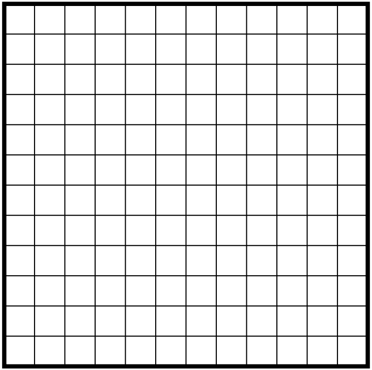
$$\frac{\partial T}{\partial t} \approx \frac{T(x, y, t + \Delta t) - T(x, y, t)}{\Delta t}$$

- Divide space in small "cells"

$$\frac{\partial^2 T}{\partial x^2} \approx \frac{T(x - \Delta x, y, t) - 2T(x, y, t) + T(x + \Delta x, y, t)}{\Delta x^2}$$

Approximating Solutions of PDEs

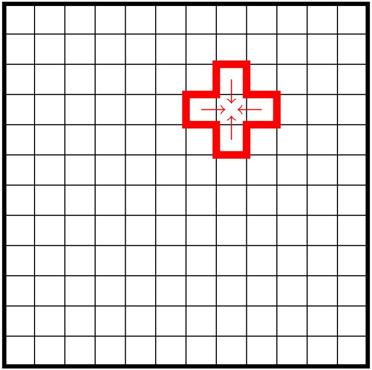
Euler's Method



"Stencil" method

Approximating Solutions of PDEs

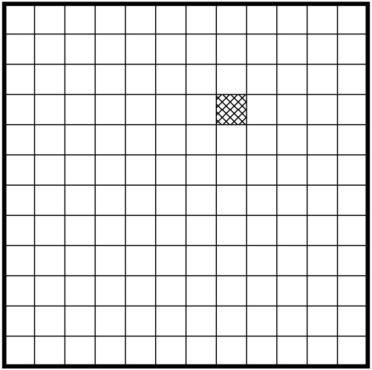
Euler's Method



"Stencil" method

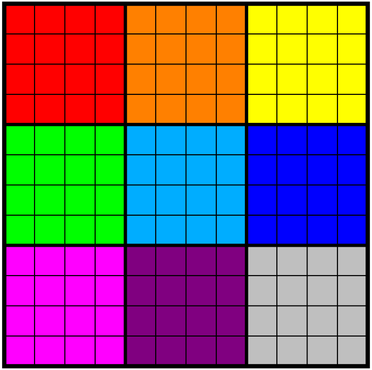
Approximating Solutions of PDEs

Euler's Method

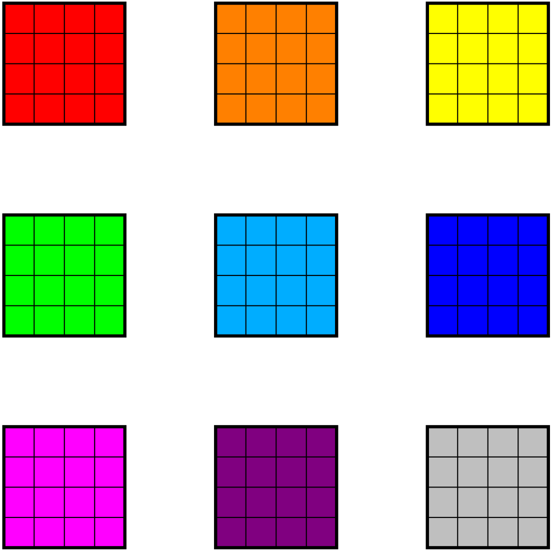


"Stencil" method

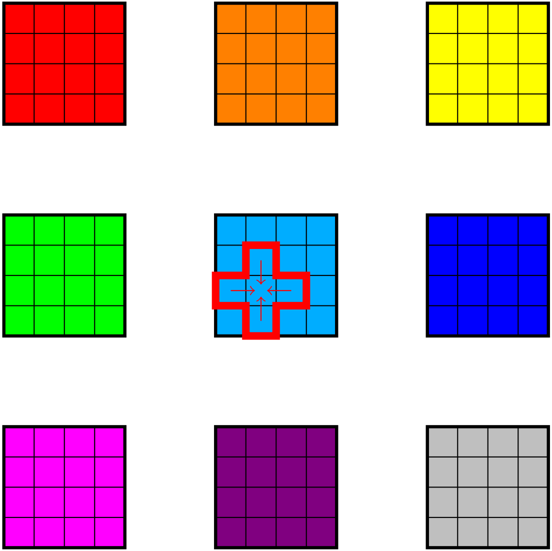
"Domain Decomposition" (a.k.a. Data Parallelism)



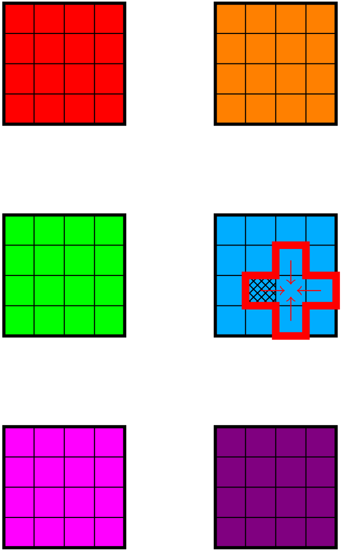
“Domain Decomposition” (a.k.a. Data Parallelism)



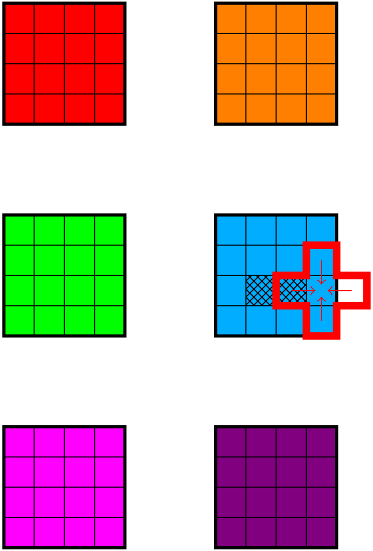
“Domain Decomposition” (a.k.a. Data Parallelism)



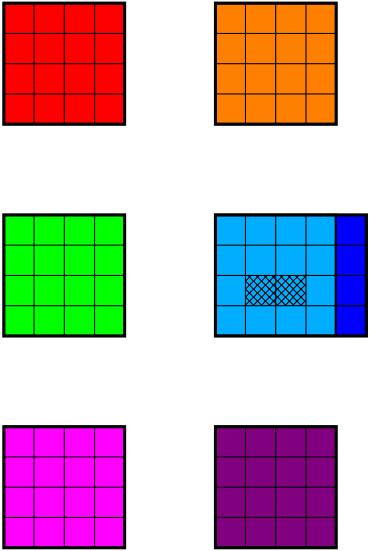
"Domain Decomposition" (a.k.a. Data Parallelism)



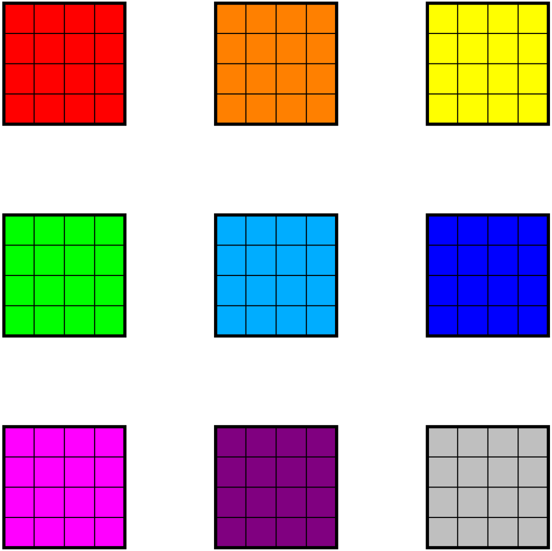
“Domain Decomposition” (a.k.a. Data Parallelism)



"Domain Decomposition" (a.k.a. Data Parallelism)



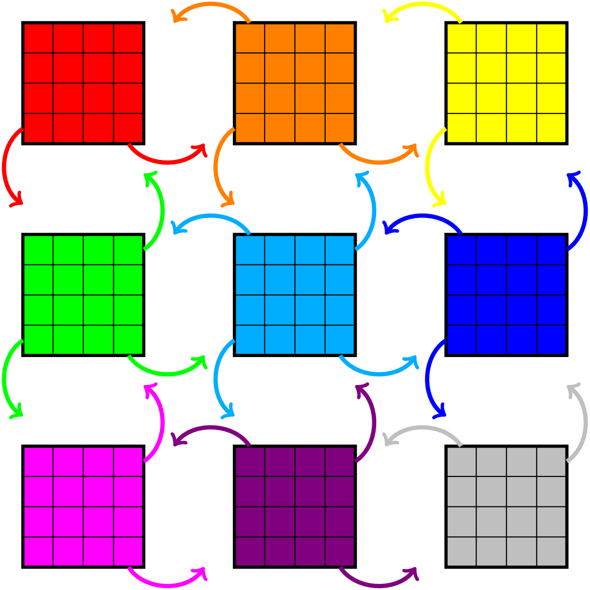
"Domain Decomposition" (a.k.a. Data Parallelism)



Each processor:

- Knows T at time t

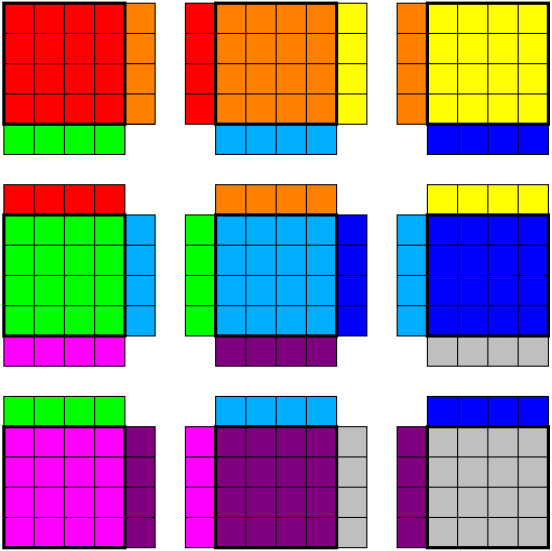
"Domain Decomposition" (a.k.a. Data Parallelism)



Each processor:

- Knows T at time t
- Sends/Receives the halo from its neighbors

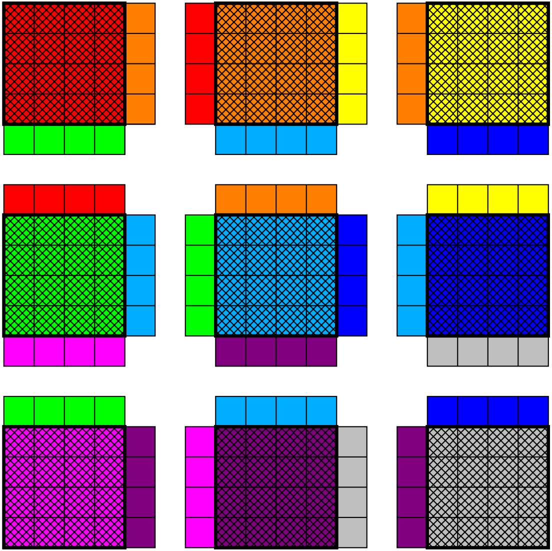
"Domain Decomposition" (a.k.a. Data Parallelism)



Each processor:

- Knows T at time t
- Sends/Receives the halo from its neighbors

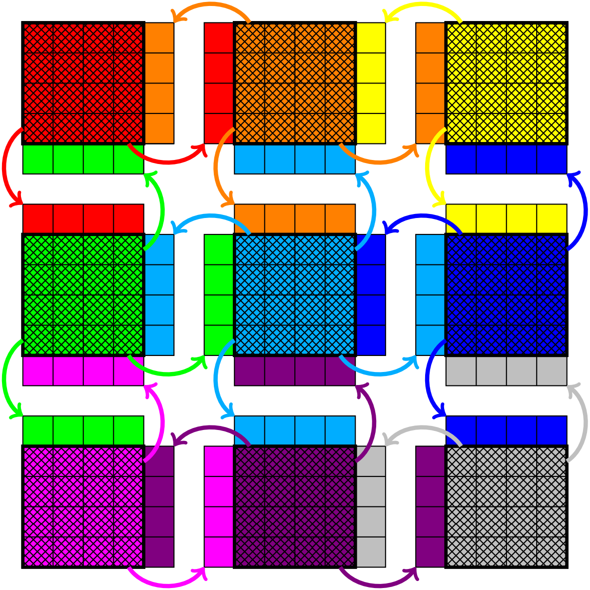
"Domain Decomposition" (a.k.a. Data Parallelism)



Each processor:

- Knows T at time t
- Sends/Receives the halo from its neighbors
- Compute T at time $t + \Delta t$

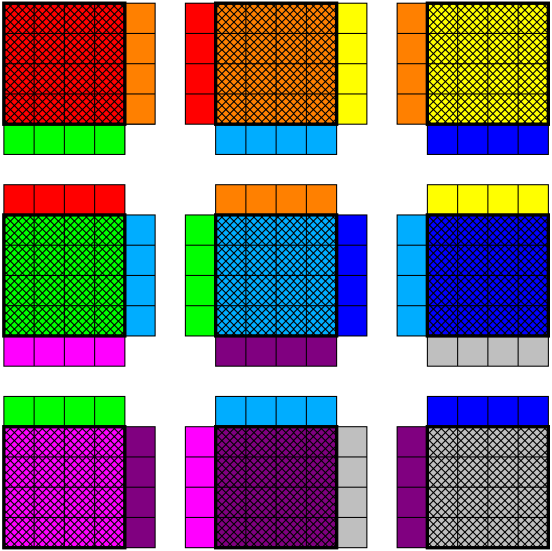
“Domain Decomposition” (a.k.a. Data Parallelism)



Each processor:

- Knows T at time t
- Sends/Receives the halo from its neighbors
- Compute T at time $t + \Delta t$
- Rinse, repeat

“Domain Decomposition” (a.k.a. Data Parallelism)



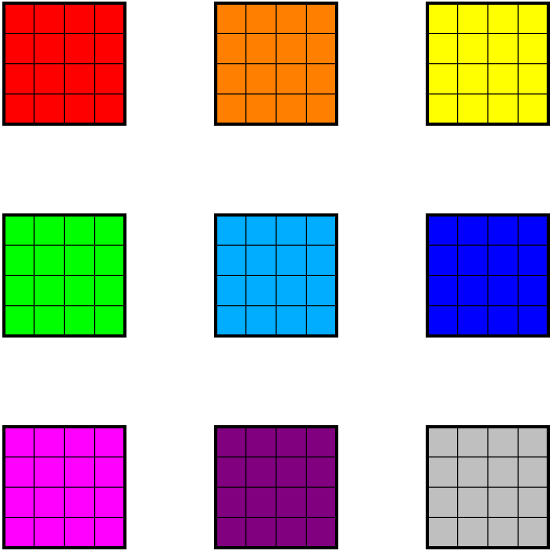
Each processor:

- Knows T at time t
- Sends/Receives the halo from its neighbors
- Compute T at time $t + \Delta t$
- Rinse, repeat

Problem

Progress blocked by communications

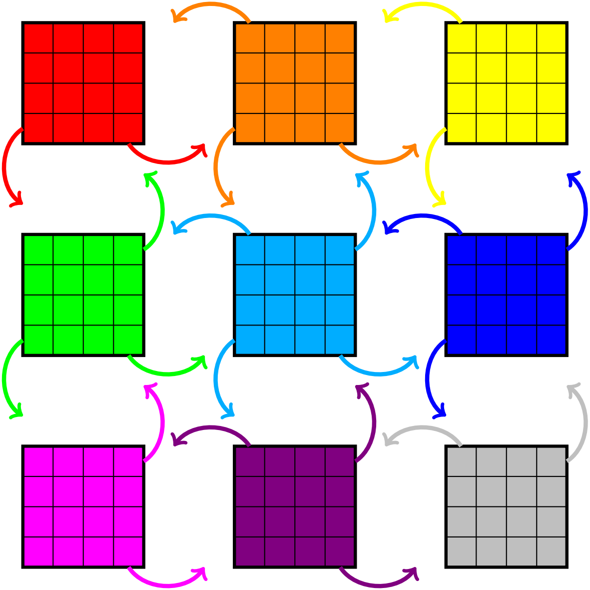
“Domain Decomposition” (a.k.a. Data Parallelism)



Each processor:

- Knows T at time t

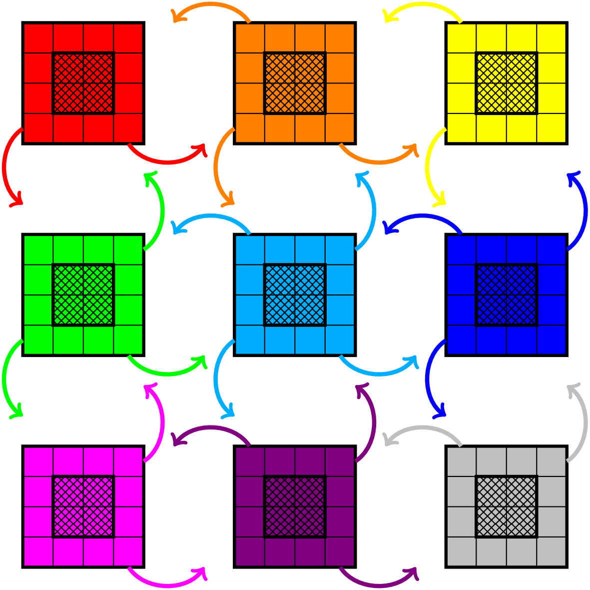
"Domain Decomposition" (a.k.a. Data Parallelism)



Each processor:

- Knows T at time t
- Sends the halo to its neighbors (Isend)

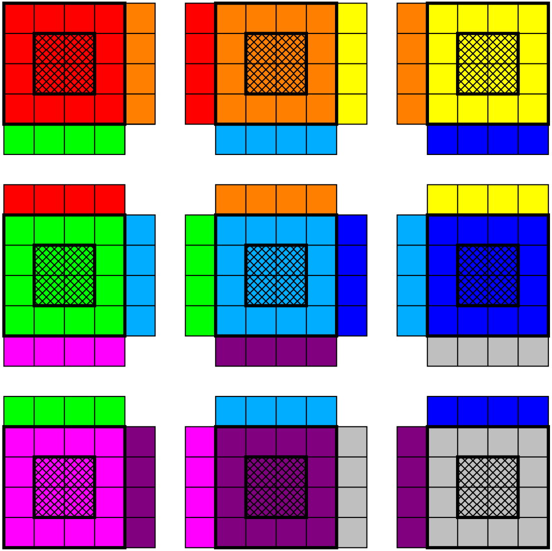
"Domain Decomposition" (a.k.a. Data Parallelism)



Each processor:

- Knows T at time t
- Sends the halo to its neighbors (Isend)
- Compute T at time $t + \Delta t$ (interior)

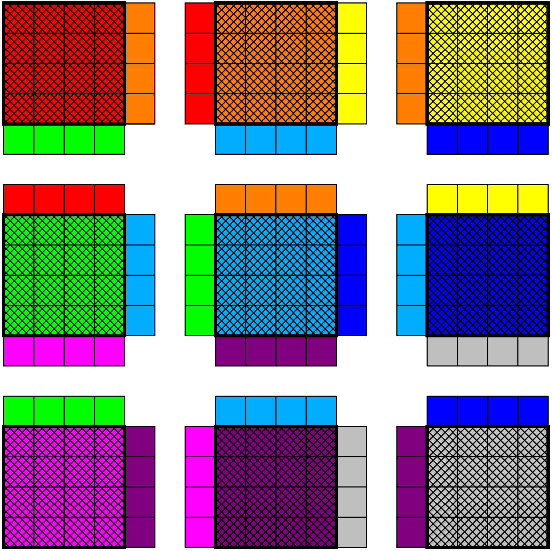
"Domain Decomposition" (a.k.a. Data Parallelism)



Each processor:

- Knows T at time t
- Sends the halo to its neighbors (Isend)
- Compute T at time $t + \Delta t$ (interior)
- Waits end of comms

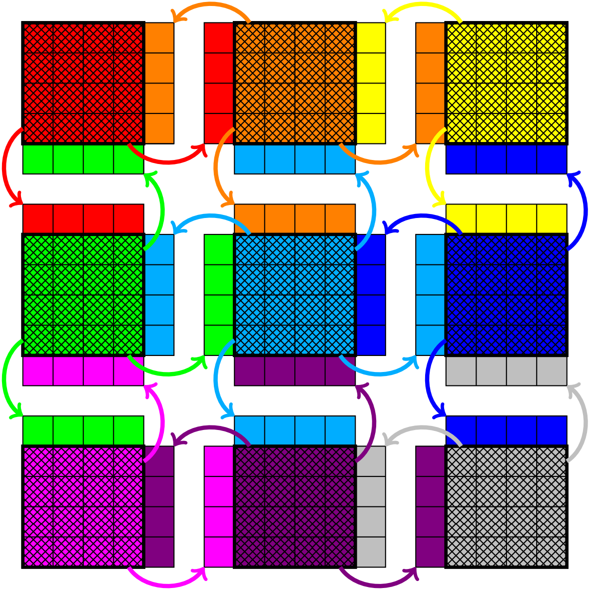
"Domain Decomposition" (a.k.a. Data Parallelism)



Each processor:

- Knows T at time t
- Sends the halo to its neighbors (Isend)
- Compute T at time $t + \Delta t$ (interior)
- Waits end of comms
- Compute T at time $t + \Delta t$ (border)

"Domain Decomposition" (a.k.a. Data Parallelism)



Each processor:

- Knows T at time t
- Sends the halo to its neighbors (Isend)
- Compute T at time $t + \Delta t$ (interior)
- Waits end of comms
- Compute T at time $t + \Delta t$ (border)
- Rinse, repeat

```c
int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype,
                 int dest, int sendtag,
                 void *recvbuf, int recvcount, MPI_Datatype recvtype,
                 int source, int recvtag,
                 MPI_Comm comm, MPI_Status *status);
```

```c
int MPI_Isend(void *buf, int count, MPI_Datatype datatype,
              int dest, int tag,
              MPI_Comm comm, MPI_Request *request);
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
              int source, int tag,
              MPI_Comm comm, MPI_Request *request);
int MPI_Waitall(int count,
                MPI_Request requests[], MPI_Status statuses[]);
```

warning!

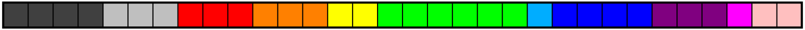
- East / West border: non-contiguous data elements

$\Rightarrow$ Must create derived MPI types (`MPI_Type_vector(...)`)

Data Parallelism (redux)

Classic example: *prefix-sum* (a.k.a. “scan”)

```c
void exclusive_scan(double acc, int n, double * A)
{
    for (int i = 0; i < n; i++) {
        double tmp = A[i];
        A[i] = acc;
        acc += tmp;                               /* n FLOP */
    }
}
```



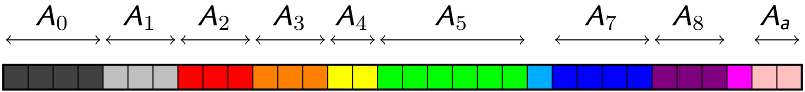
- $A_i \leftarrow \mathrm{acc} + A_0 + A_1 + \cdots + A_{i-1}$
- Common operation when dealing with irregular sizes
- (Apparently unescapable) **data dependency**
  - Each iteration needs the result of the previous one

⇝ Change the algorithm

Data Parallelism (redux)

Classic example: *prefix-sum* (a.k.a. "scan")

```c
void exclusive_scan(double acc, int n, double * A)
{
    for (int i = 0; i < n; i++) {
        double tmp = A[i];
        A[i] = acc;
        acc += tmp;                         /* n FLOP */
    }
}
```



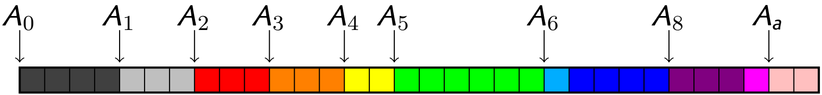
- $A_i \leftarrow \mathtt{acc} + A_0 + A_1 + \cdots + A_{i-1}$
- Common operation when dealing with irregular sizes
- (Apparently unescapable) **data dependency**
  - Each iteration needs the result of the previous one

⇝ Change the algorithm

Data Parallelism (redux)

Classic example: *prefix-sum* (a.k.a. "scan")

```
void exclusive_scan(double acc, int n, double * A)
{
    for (int i = 0; i < n; i++) {
        double tmp = A[i];
        A[i] = acc;
        acc += tmp;                              /* n FLOP */
    }
}
```

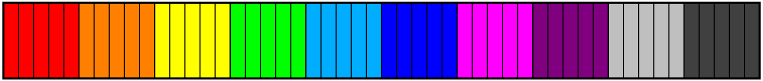


- $A_i \leftarrow \text{acc} + A_0 + A_1 + \cdots + A_{i-1}$
- Common operation when dealing with irregular sizes
- (Apparently unescapable) **data dependency**
  - Each iteration needs the result of the previous one

$\rightsquigarrow$ Change the algorithm

Distributed `Prefix-Sum`

- `exclusive_scan(0, ...)` a block-distributed array



1. P_i computes the sum S_i of its own slice [local]
2. Distributed algorithm for $T \leftarrow$ `exclusive_scan(0, p, S)`
   - `MPI_Exscan`

$\rightarrow$ P_i gets T_i = sum of *all previous* slices

3. P_i `exclusive_scan(T[i], ...)` its own slice [local]

Analysis

$$T = [\text{communication}] + 2\frac{n}{p}\ \text{FLOP}$$

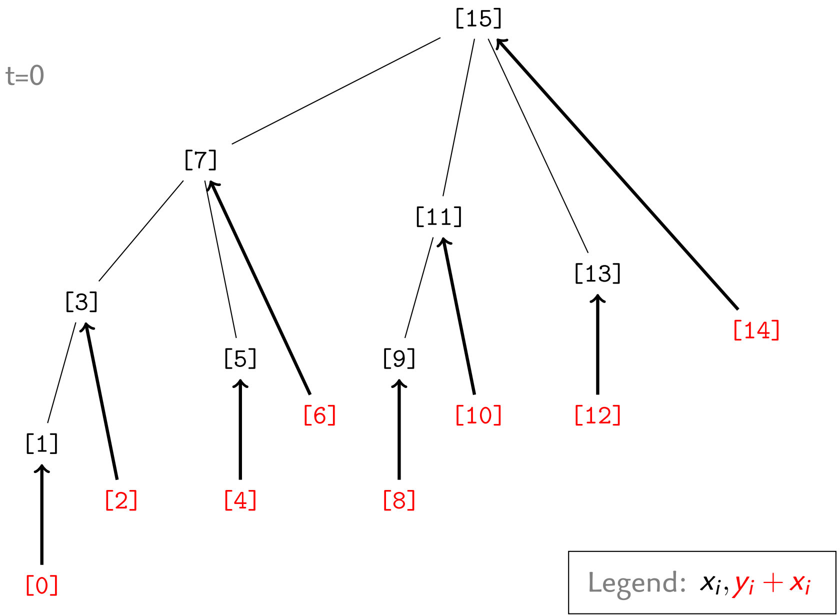
MPI_Exscan

Phase 1 : reduce

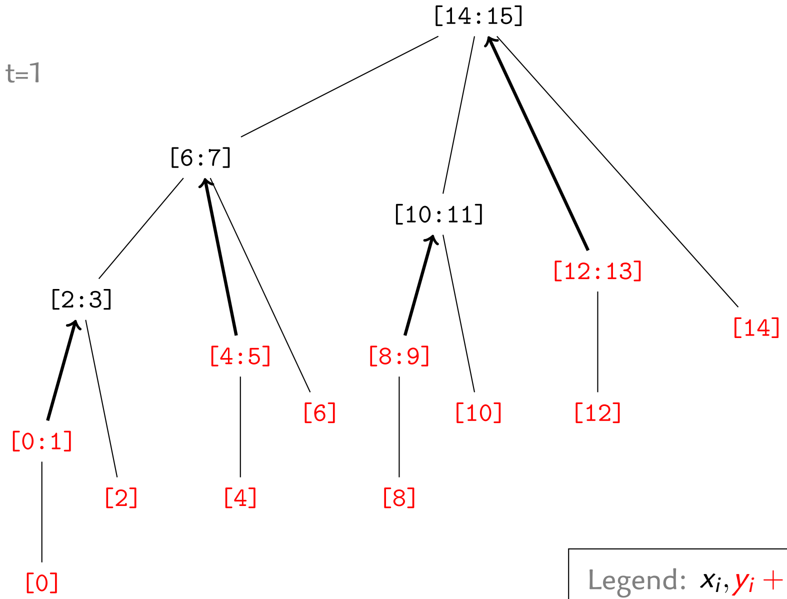
- Input x_i
- Receive $u_0, u_1, \dots$ from children
- Set $y_i \leftarrow \sum u_j$. Send $x_i + y_i$ to father

$\rightarrow \lceil \log_2 p \rceil$ successive messages of size 1

MPI_Reduce Using the Binomial Tree Algorithm

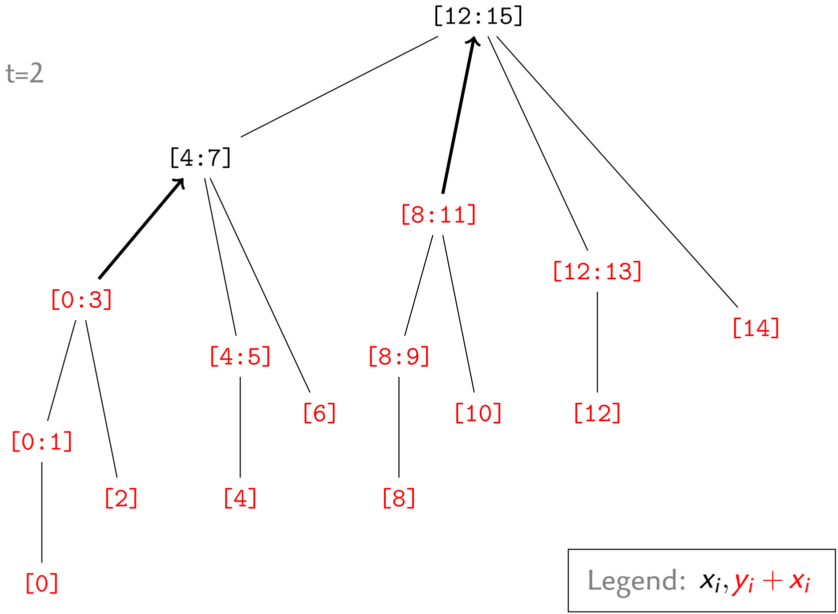


`MPI_Reduce` Using the Binomial Tree Algorithm

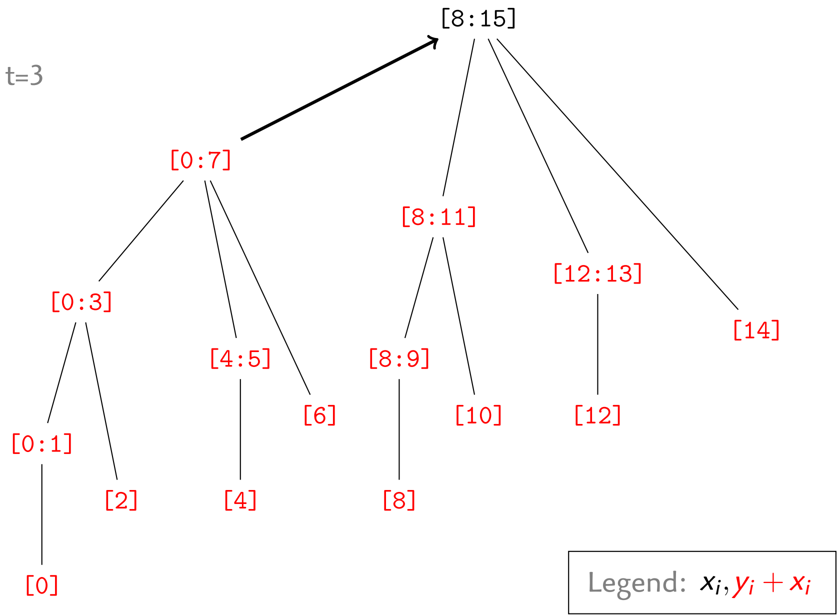


Legend: $x_i, y_i + x_i$

MPI_Reduce Using the Binomial Tree Algorithm

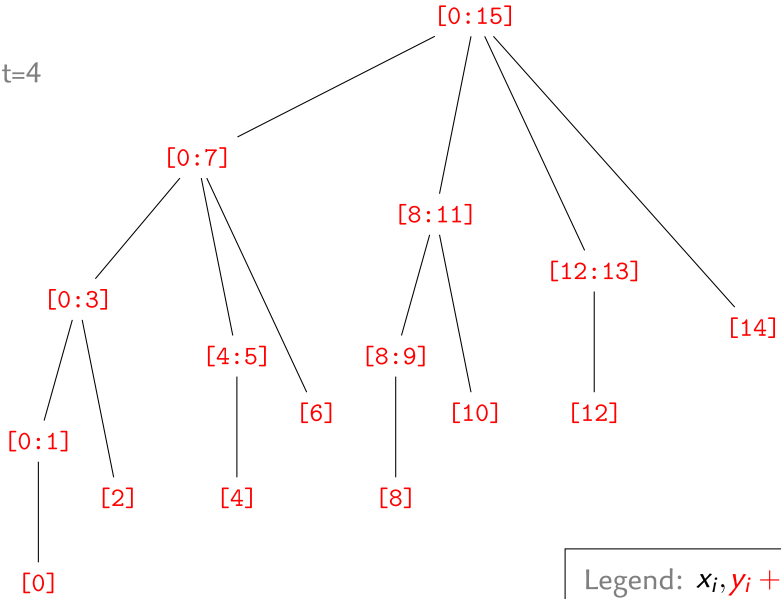


MPI_Reduce Using the Binomial Tree Algorithm



`MPI_Reduce` Using the Binomial Tree Algorithm

t=4



Legend: $x_i, y_i + x_i$

`MPI_Exscan`

Phase 1 : `reduce`

- Input x_i
- Receive $u_0, u_1, \dots$ from children
- Set $y_i \leftarrow \sum u_j$. Send $x_i + y_i$ to father

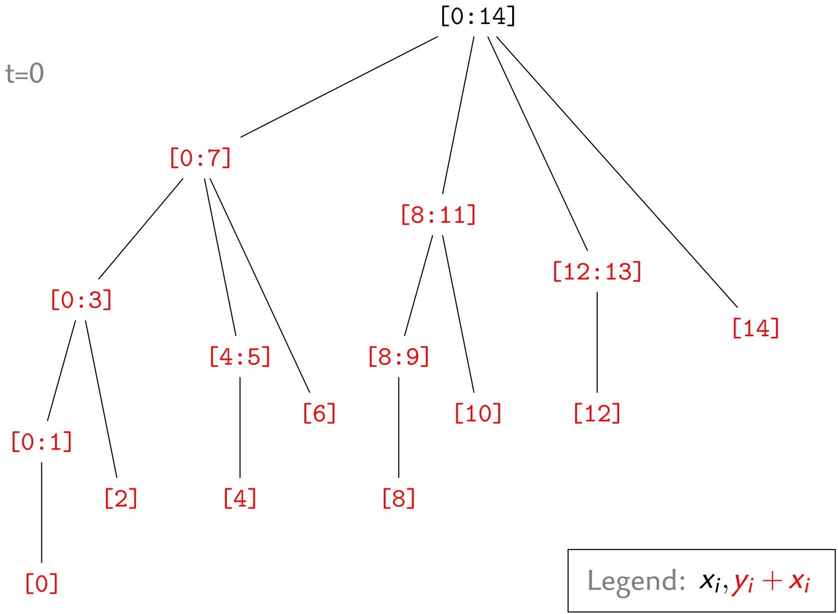
$\rightarrow \lceil \log_2 p \rceil$ successive messages of size 1

Phase 2 : `exclusive scan`

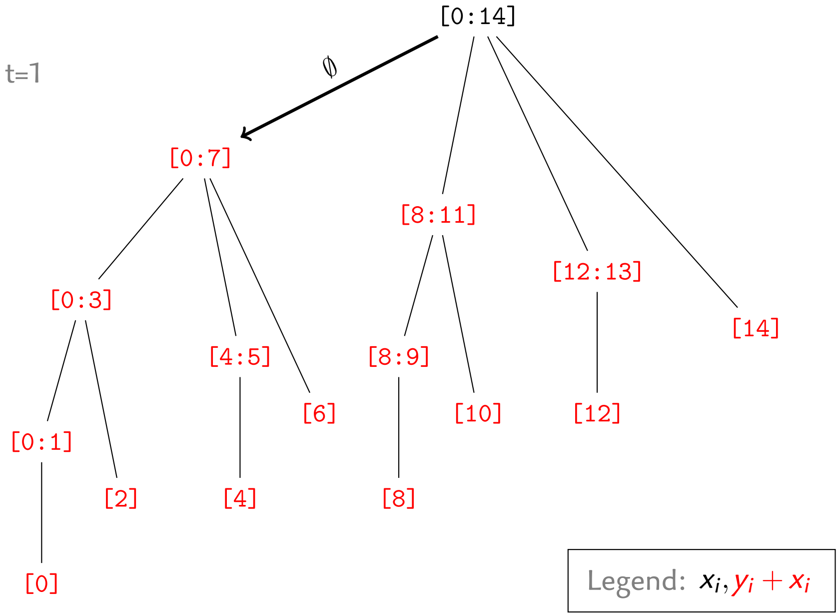
1. Root: $w_i \leftarrow 0$, otherwise receive w_i from father
   $=$ [sum of values of *left* siblings]
2. Set $x_i \leftarrow y_i + w_i$
3. Sends $w_i + u_0 + \cdots + u_{j-1}$ to the j-th children
   $\rightsquigarrow$ Prefix-sum the u_j's starting from w_i

$\rightarrow \lceil \log_2 p \rceil$ successive messages of size 1

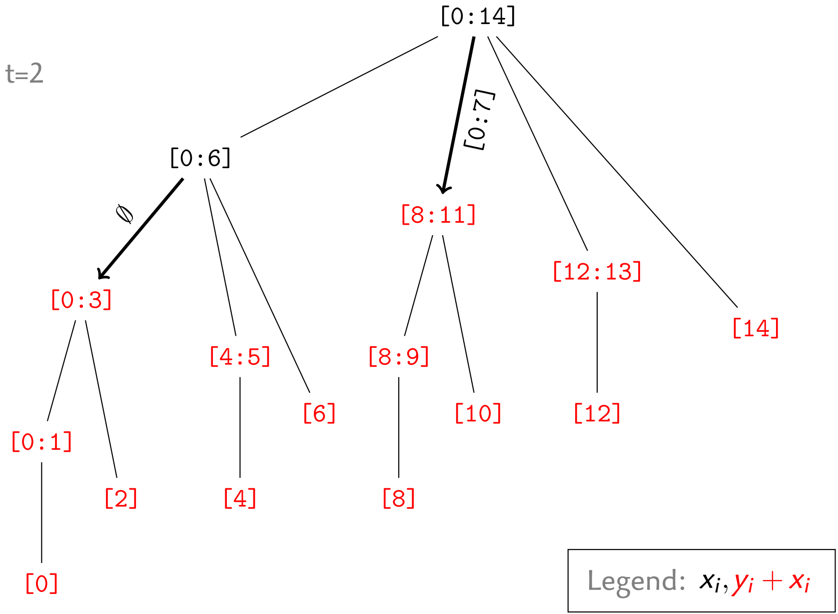
MPI_Exscan Using the Binomial Tree Algorithm



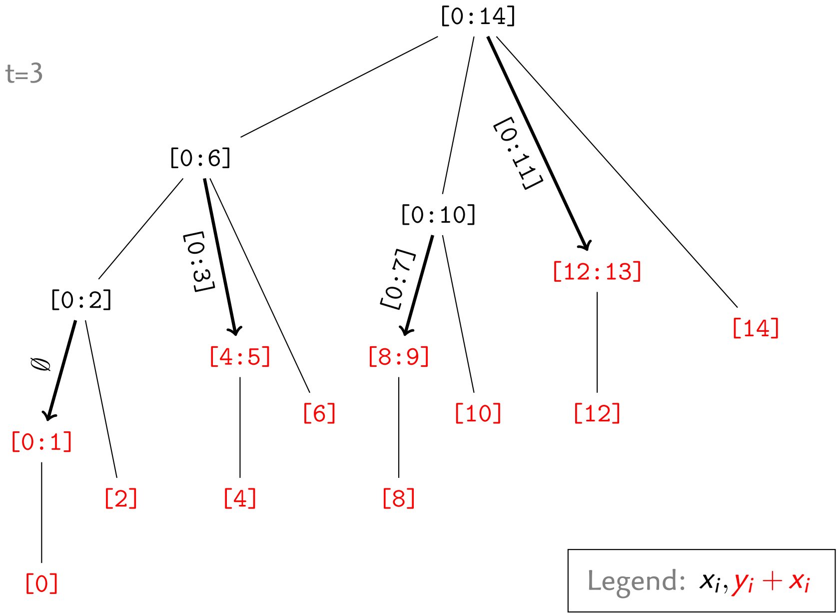
`MPI_Exscan` Using the Binomial Tree Algorithm



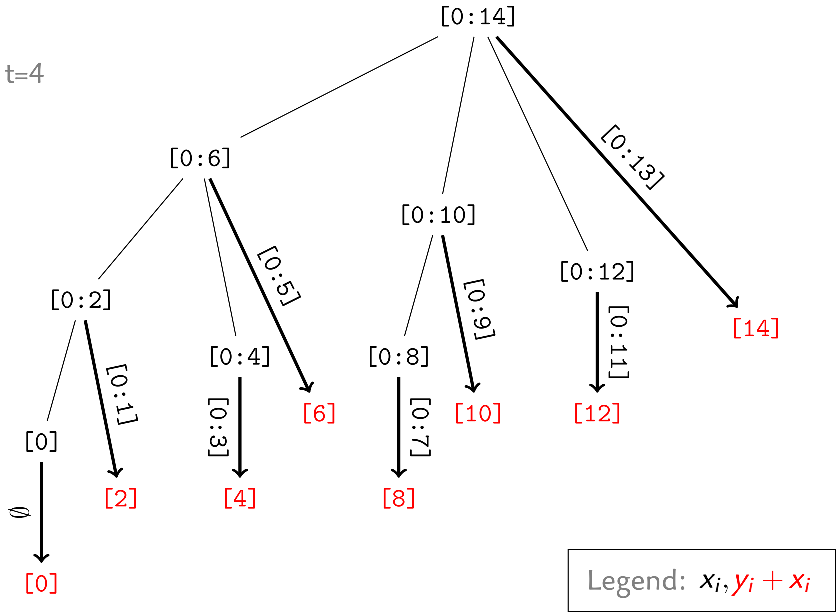
MPI_Exscan Using the Binomial Tree Algorithm



`MPI_Exscan` Using the Binomial Tree Algorithm

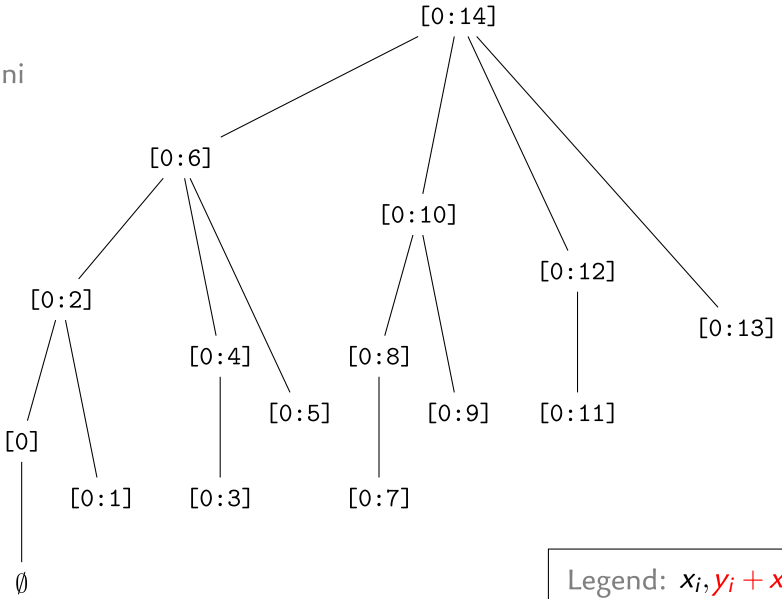


MPI_Exscan Using the Binomial Tree Algorithm



`MPI_Exscan` Using the Binomial Tree Algorithm

fini



Legend: $x_i, y_i + x_i$

Linear Algebra

Importance of Linear Algebra

Reason

Systems of **linear** equations are the only ones we can solve

- (efficiently, at least)

- $Ax = b$
- $AX = B$ (multiple right-hand sides)
- $\min_x \|Ax - b\|_2$ (overdetermined linear least squares)
- $\min_x \|x\|_2$ s.t. $Ax = b$ (underdetermined linear least squares)
- $Av = \lambda v$ (eigenvalues/eigenvectors)

Basic building block of nearly all scientific computation

BLAS

Basic Linear Algebra Subroutines

- Software libraries developed in the 1980's (Fortran-77...)
- Simple and common operations
- heavily optimized

  $\Rightarrow$ you will never do better

Common HPC Design Strategy

- More complex linear algebra use the BLAS
- Simplifies development
  - Just need to remember the interface
- High-speed BLAS $\rightsquigarrow$ high-speed software
- Ugly low-level optimizations *confined* inside the BLAS

BLAS Levels

$$\mathtt{x} \in \{\mathtt{S}, \mathtt{D}, \mathtt{C}, \mathtt{Z}\}$$

Level 1 Routines: vector operations

- $\mathtt{xSCAL}: x \leftarrow \alpha x$
- $\mathtt{xCOPY}: x \leftarrow y$
- $\mathtt{xAXPY}: y \leftarrow \alpha x + y$
- $\mathtt{xDOT}: \alpha \leftarrow x \cdot y$
- $\mathtt{xNORM}: \alpha \leftarrow \|x\|_2$
- $\mathtt{xSUM}: k \leftarrow \sum_i |x_i|$
- $\mathtt{IxAMAX}: k \leftarrow \arg\max_i |x_i|$

BLAS Levels

$$\mathtt{x} \in \{\mathtt{S}, \mathtt{D}, \mathtt{C}, \mathtt{Z}\}$$

Level 2 Routines: matrix-vector operations

- Matrix-vector product (`xGEMV`)
  - $y \leftarrow \alpha Ax + \beta y$
  - Options to multiply by A^t or A^h
  - Special cases: symmetric A (`xSYMV`), triangular A (`xTRMV`)
- Triangular solve (`xTRSV`)
  - Solve $Lx = b$ or $Ux = b$
- Rank-1 update (`xGER`)
  - $A \leftarrow A + \alpha xy^t$

BLAS Levels

$$\mathtt{x} \in \{\mathtt{S}, \mathtt{D}, \mathtt{C}, \mathtt{Z}\}$$

Level 3 Routines: matrix-matrix operations

- Matrix-Matrix product (`xGEMM`)
  - $C \leftarrow \alpha AB + \beta C$
  - Options to use A^t or B^t
  - Special cases: symmetric A, B (`xSYMM`), triangular A (`xTRMM`)
- Triangular solve (`xTRSM`) with multiple right-hand sides
  - Solve $LX = B$ or $UX = B$
- Rank-k symmetric update (`xSYRK`)
  - $C \leftarrow \alpha AA^t + \beta C$
  - Symmetric C (otherwise it is just `xGEMM`)

Data Representation

Matrices represented as **1D arrays**

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & a & b \end{pmatrix}$$

- Memory: `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0xa, 0xb]`
- **Never ever** as array of pointers to arrays

`double **A` is **banished**

- Can write `A[i][j]`
- Accessing `A[i + 1][j]` is **costly** (indirection)
- Pointers take space
- `malloc()` takes time (how much?)
- Allocation complex and error-prone

Data Representation (continued)

Flat arrays: two possible orderings

- Memory: `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0xa, 0xb]`
- Row-major order (C, Pascal)

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & a & b \end{pmatrix}$$

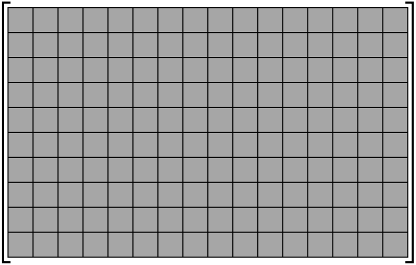
- Column-major order (Fortran, matlab, julia, R)

$$\begin{pmatrix} 0 & 3 & 6 & 9 \\ 1 & 4 & 7 & a \\ 2 & 5 & 8 & b \end{pmatrix}$$

Data Representation (continued)

Matrix given by

- Memory address of first coefficient
- #Rows (m)
- #Columns (n)
- Leading dimension ("stride")
  - Sub-matrices for free

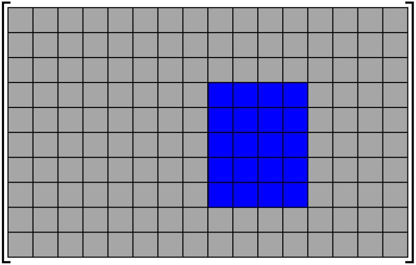


- Row-major (`ld >= n`)
  - $M_{ij} \rightsquigarrow$ `A[i*ld + j]`
  - $A, m = 10, n = 16, ld = 16$
- Column-major (`ld >= m`)
  - $M_{ij} \rightsquigarrow$ `A[j*ld + i]`
  - $A, m = 10, n = 16, ld = 10$

Data Representation (continued)

Matrix given by

- Memory address of first coefficient
- #Rows (m)
- #Columns (n)
- Leading dimension ("stride")
  - Sub-matrices for free

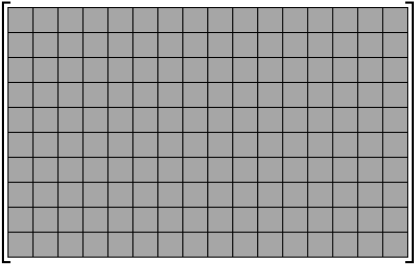


- Row-major (`ld >= n`)
  - $M_{ij} \rightsquigarrow$ `A[i*ld + j]`
  - $A, m = 10, n = 16, ld = 16$
  - $A + 57, m = 5, n = 4, ld = 16$
- Column-major (`ld >= m`)
  - $M_{ij} \rightsquigarrow$ `A[j*ld + i]`
  - $A, m = 10, n = 16, ld = 10$
  - $A + 84, m = 5, n = 4, ld = 10$

Data Representation (continued)

Vector given by

- Memory address of first coefficient
- Size (n)
- Increment ("stride")

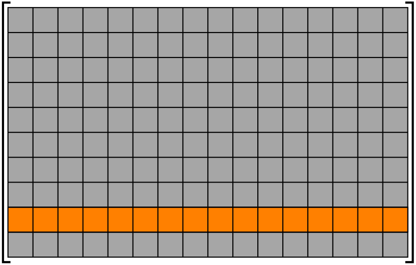


- Row-major ($M_{ij} \rightsquigarrow$ `A[i*ld + j]`)

Data Representation (continued)

Vector given by

- Memory address of first coefficient
- Size (n)
- Increment ("stride")

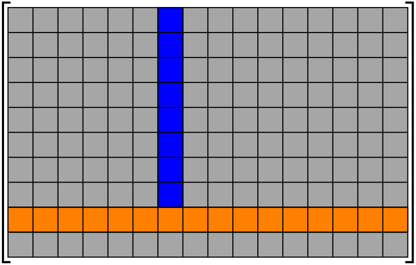


- Row-major ($M_{ij} \rightsquigarrow$ `A[i*ld + j]`)
  - $A + 128, n = 16, i = 1$

Data Representation (continued)

Vector given by

- Memory address of first coefficient
- Size (n)
- Increment ("stride")

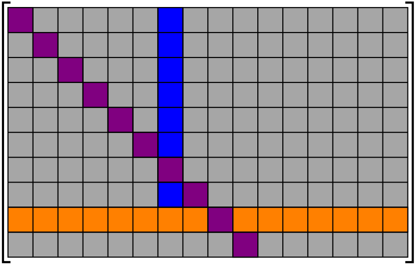


- Row-major ($M_{ij} \rightsquigarrow$ `A[i*ld + j]`)
  - $A + 128, n = 16, i = 1$
  - $A + 6, n = 8, i = 16$

Data Representation (continued)

Vector given by

- Memory address of first coefficient
- Size (n)
- Increment (“stride”)

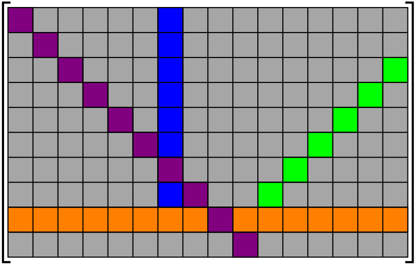


- Row-major ($M_{ij} \rightsquigarrow$ `A[i*ld + j]`)
  - $A + 128, n = 16, i = 1$
  - $A + 6, n = 8, i = 16$
  - $A, n = 10, i = 17$

Data Representation (continued)

Vector given by

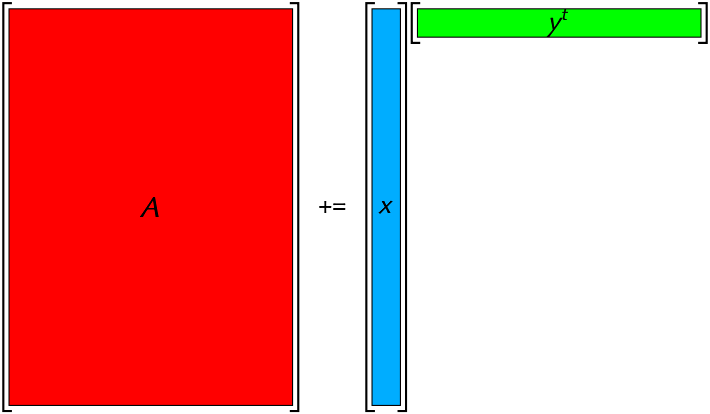
- Memory address of first coefficient
- Size (n)
- Increment (“stride”)



- Row-major ($M_{ij} \rightsquigarrow$ `A[i*ld + j]`)
  - $A + 128, n = 16, i = 1$
  - $A + 6, n = 8, i = 16$
  - $A, n = 10, i = 17$
  - $A + 31, n = 6, i = 15$

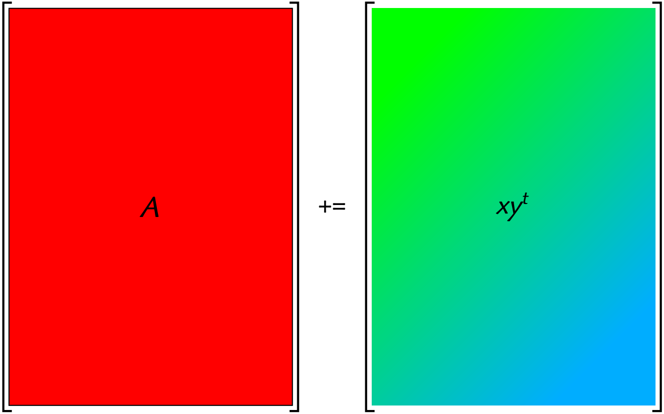
Rank-1 update (`xGER`)

$A \leftarrow A + \alpha x y^t$



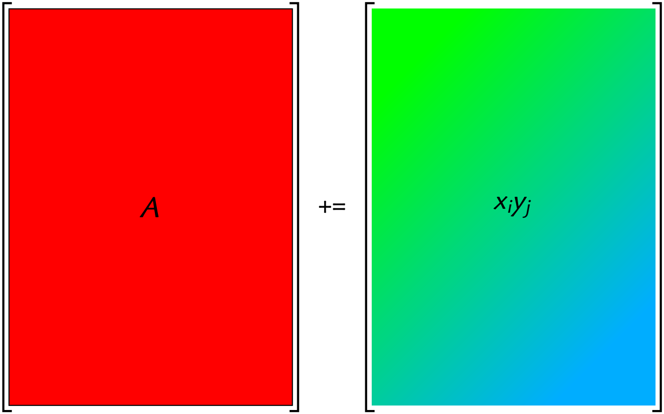
Rank-1 update (`xGER`)

$A \leftarrow A + \alpha x y^t$



Rank-1 update (`xGER`)

$A \leftarrow A + \alpha x y^t$



Double-Precision Rank-1 update (`DGER`)

$A \leftarrow A + \alpha x y^t$

```c
void dGER(int m, int n, double alpha, double *x, int incx,
          double *y, int incy, double *A, int ldA)
{
    for (int i = 0; i < m; i++)
        for (int j = 0; j < n; j++)
            A[i * ldA + j] += alpha * x[i * incx] * y[j * incy];
}
```

- $3mn$ integer multiplications, $2mn + m$ integer additions
- $3mn$ FLOP
- $4mn$ memory accesses

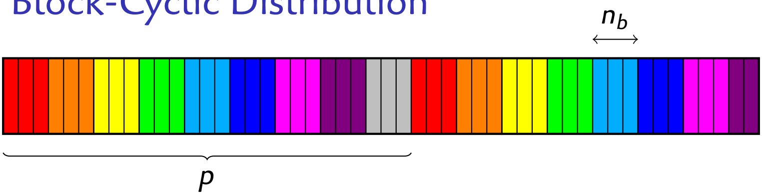
Double-Precision Rank-1 update (`DGER`)

$A \leftarrow A + \alpha x y^t$

```c
void dGER(int m, int n, double alpha, double *x, int incx,
          double *y, int incy, double *A, int ldA)
{
    int ix = 0, iA = 0;
    for (int i = 0; i < m; i++) {
        double tmp = alpha * x[ix];
        int jy = 0;
        for (int j = 0; j < n; j++) {
            A[iA + j] += tmp * y[jy];
            jy += incy;
        }
        ix += incx;
        iA += ldA;
    }
}
```

- 0 integer multiplications, $2mn + 2m$ integer additions
- $2nm + m$ FLOP
- $3mn + m$ memory accesses

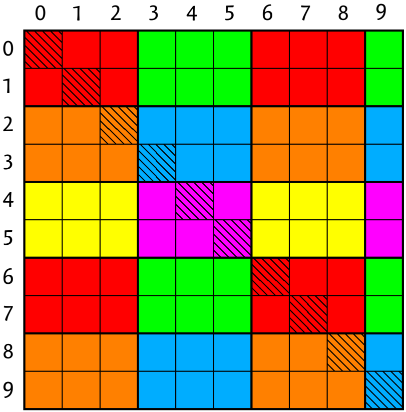
1D Block-Cyclic Distribution



- 3 parameters: (n, n_b, p)
- Element i belongs to process $\lfloor i/n_b \rfloor \bmod p$

```c
/* returns the number of items I have */
int my_size(int n, int nb, int p, int rank)
{
    int nblocks = n / nb;                          /* #full blocks */
    int res = (nblocks / p) * nb;                   /* lower-bound */
    int extrablocks = nblocks % p;
    if (rank < extrablocks)                /* I have an extra block */
        res += nb;
    if (rank == extrablocks)               /* I have the last block */
          res += n % nb;
    return res;
}
```

2D Block-Cyclic Distribution

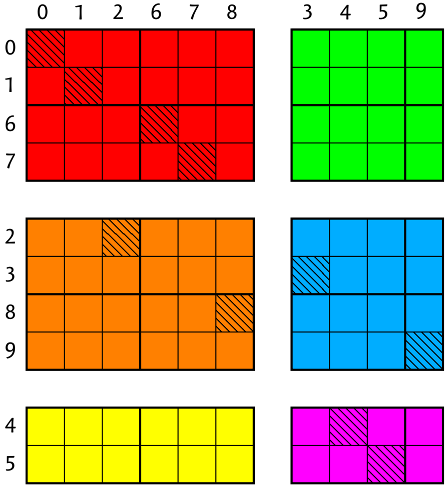


- Matrix: $m \times n$
- Process grid: $P \times Q$
- Block size: $v \times h$

Special cases

- By block
  - $v = m/P$
  - $h = n/Q$
- Purely Cyclic
  - $v = 1$
  - $h = 1$
- 1D
  - $h = n$ or $v = m$

2D Block-Cyclic Distribution



- Rows distributed block-cyclically
- Cols distributed block-cyclically

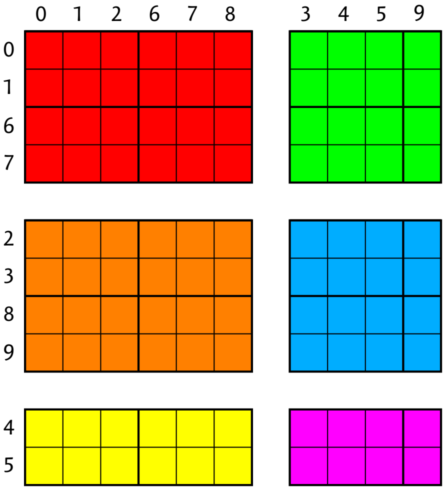
⇝ 6 parameters
  - (m, n, v, h, P, Q)

- Each process has a “local” matrix
  - Its blocks stacked together
  - With a stride!
- Not the same size everywhere!
  - `my_size(m, v, p, rank)` rows
  - `my_size(n, h, q, rank)` cols

Exercise for next week

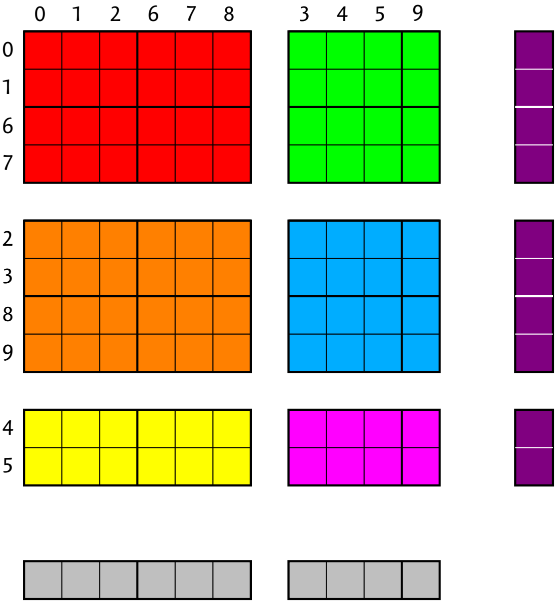
Write a function that compute the **trace** (sum of diagonal elements) of a block-cyclically distributed matrix.

`xGER` in Block-Cyclic Distribution



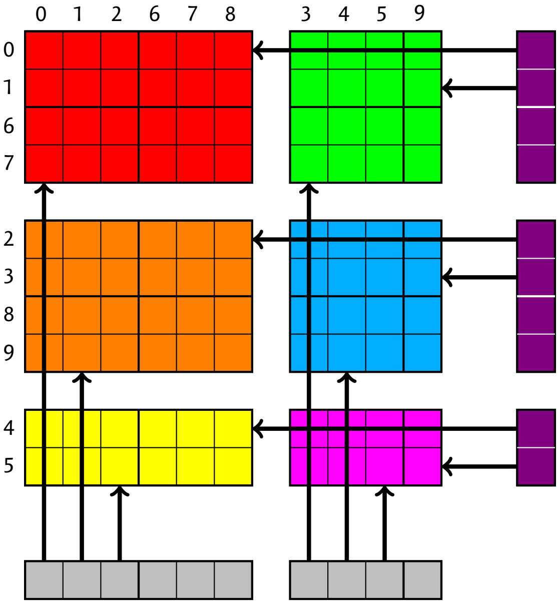
- $A \leftarrow A + \alpha x y^t$
- x and y block-cyclically distributed
- replicate x / y on process rows / cols
    - Question: where are x and y at the beginning?
- Then local `xGER`
- "Easy" 

`xGER` in Block-Cyclic Distribution



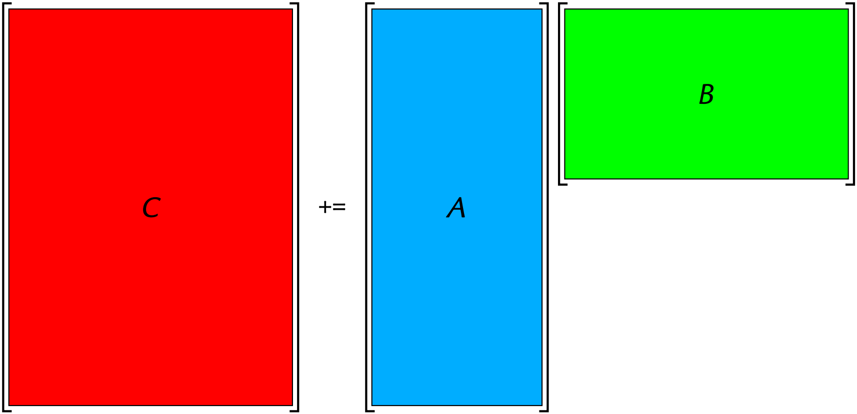
- $A \leftarrow A + \alpha x y^t$
- x and y block-cyclically distributed
- replicate x / y on process rows / cols
  - Question: where are x and y at the beginning?
- Then local `xGER`
- "Easy"

`xGER` in Block-Cyclic Distribution



- $A \leftarrow A + \alpha x y^t$
- x and y block-cyclically distributed
- replicate x / y on process rows / cols
  - Question: where are x and y at the beginning?
- Then local `xGER`
- "Easy"

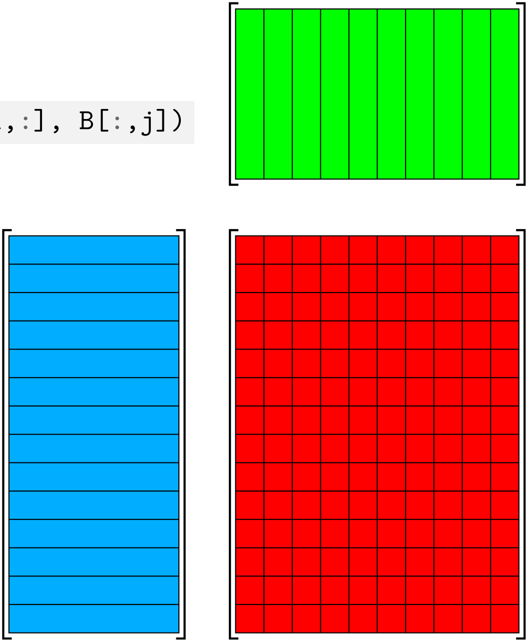
Level-3 General Matrix-Matrix Product (`xGEMM`)



Level-3 General Matrix-Matrix Product (`xGEMM`)

$$C_{ij} \leftarrow C_{ij} + \sum_k A_{ik} B_{kj}$$

$C_{ij} \leftarrow C_{ij} +$ `xDOT(A[i,:], B[:,j])`



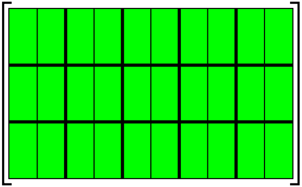
Algorithm for P_{ij}

- Allgather $A[i,:]$ on process row i
- Allgather $B[:,j]$ on process col j
- `xDOT(...)`
- Store full row/col

Level-3 General Matrix-Matrix Product (`xGEMM`)

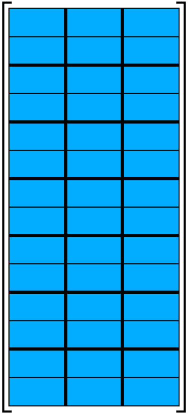
$$C_{ij} \leftarrow C_{ij} + \sum_k A_{ik} B_{kj}$$

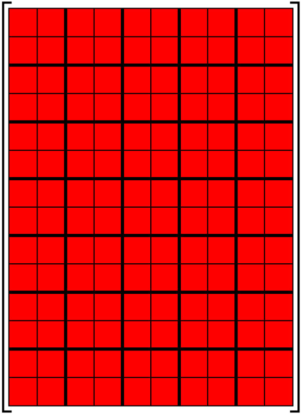
$C_{ij} \leftarrow C_{ij} +$ `xDOT(A[i,:], B[:,j])`



Algorithm for P_{ij}

- Allgather $A[i,:]$ on process row i
- Allgather $B[:,j]$ on process col j
- `xGEMM(...)`
- Store 3 blocks
- Pipeline
  - complex

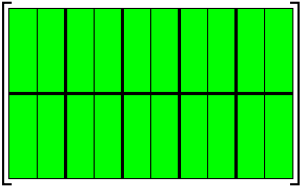




Level-3 General Matrix-Matrix Product (`xGEMM`)

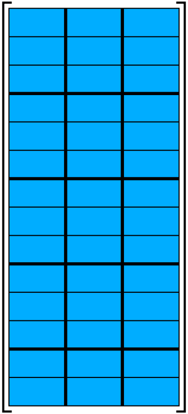
$$C_{ij} \leftarrow C_{ij} + \sum_k A_{ik} B_{kj}$$

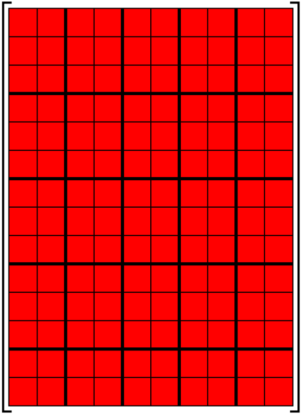
$$C_{ij} \leftarrow C_{ij} + \texttt{xDOT(A[i,:], B[:,j])}$$



Algorithm for P_{ij}

- Allgather $A[i,:]$ on process row i
- Allgather $B[:,j]$ on process col j
- ~~`xGEMM(...)`~~
- ~~pipeline~~
- Store full row/col

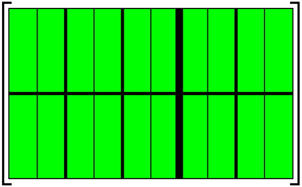




Level-3 General Matrix-Matrix Product (`xGEMM`)

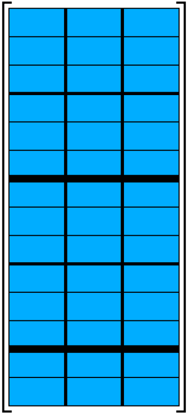
$$C_{ij} \leftarrow C_{ij} + \sum_k A_{ik} B_{kj}$$

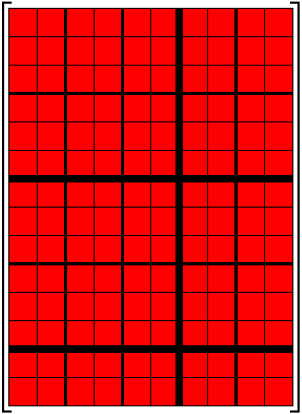
$$C_{ij} \leftarrow C_{ij} + \texttt{xDOT(A[i,:], B[:,j])}$$



Algorithm for P_{ij}

- Allgather $A[i,:]$ on process row i
- Allgather $B[:,j]$ on process col j
- `xGEMM(...)`
- Store 6 blocks
- Pipeline
  - complex





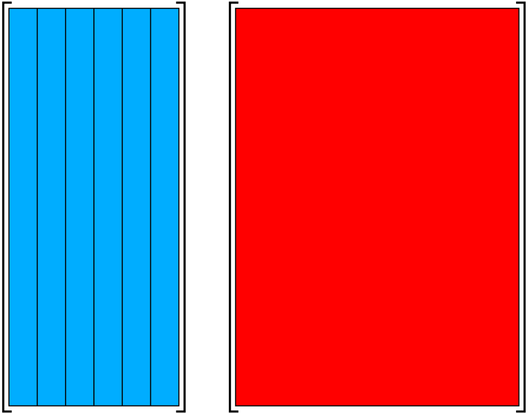
Level-3 General Matrix-Matrix Product (`xGEMM`)

$$C \leftarrow C + \sum_k A_k B_k$$

$$C \leftarrow C + \texttt{xGER(A[:,k], B[k:])}$$

Algorithm for P_{ij}

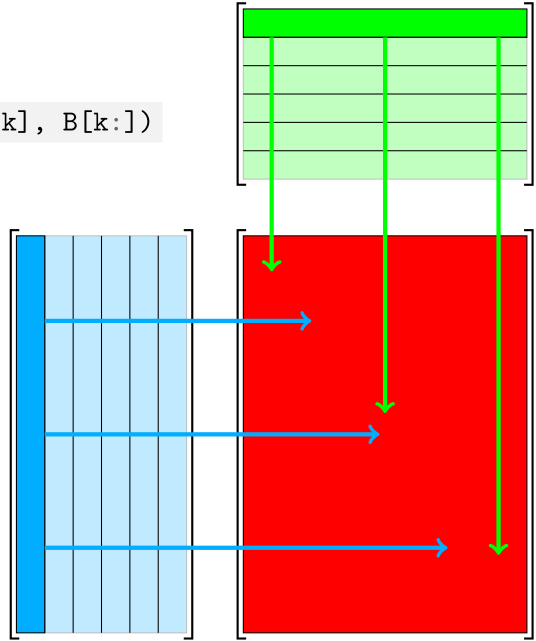
- Repeat k times:
- Broadcast $A[:,k]$
- Broadcast $B[k,:]$
- `xGER(...)`
- Pipeline
  - Simpler



Level-3 General Matrix-Matrix Product (`xGEMM`)

$$C \leftarrow C + \sum_k A_k B_k$$

$C \leftarrow C +$ `xGER(A[:,k], B[k:])`



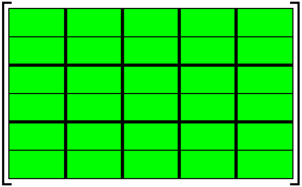
Algorithm for P_{ij}

- Repeat k times:
- Broadcast $A[:, k]$
- Broadcast $B[k, :]$
- `xGER(...)`
- Pipeline
  - Simpler

Level-3 General Matrix-Matrix Product (`xGEMM`)

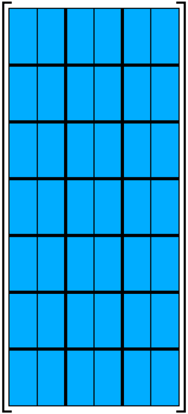
$$C \leftarrow C + \sum_k A_k B_k$$

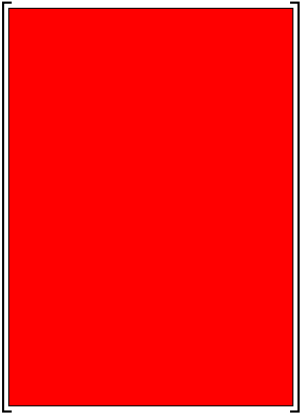
$$C \leftarrow C + \texttt{xGER(A[:,k], B[k:])}$$



Algorithm for P_{ij}

- Repeat k/b times:
- Broadcast $A[:,k]$
- Broadcast $B[k,:]$
- `xGEMM(...)`
- Pipeline
  - Simpler

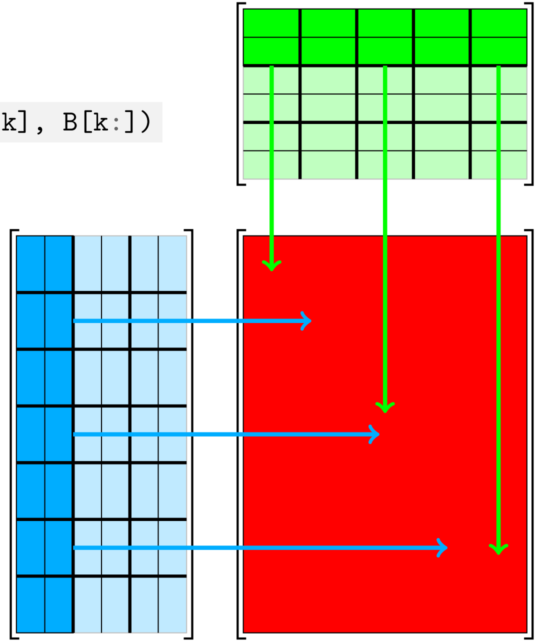




Level-3 General Matrix-Matrix Product (`xGEMM`)

$$C \leftarrow C + \sum_k A_k B_k$$

$C \leftarrow C+$ `xGER(A[:,k], B[k:])`



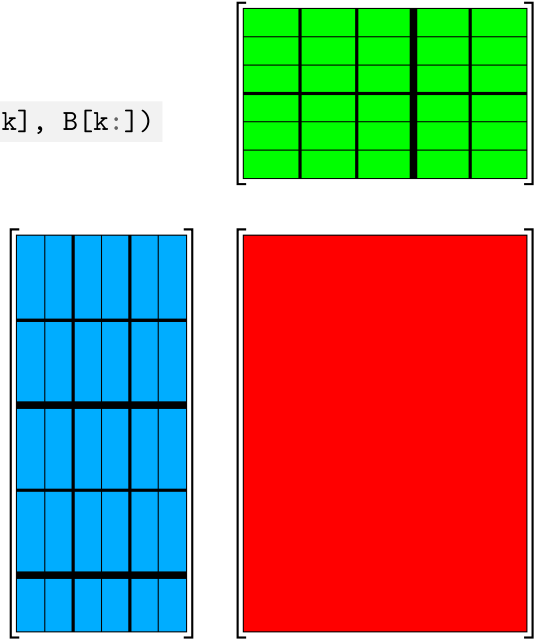
Algorithm for P_{ij}

- Repeat k/b times:
- Broadcast $A[:, k]$
- Broadcast $B[k, :]$
- `xGEMM(...)`
- Pipeline
  - Simpler

Level-3 General Matrix-Matrix Product (`xGEMM`)

$$C \leftarrow C + \sum_k A_k B_k$$

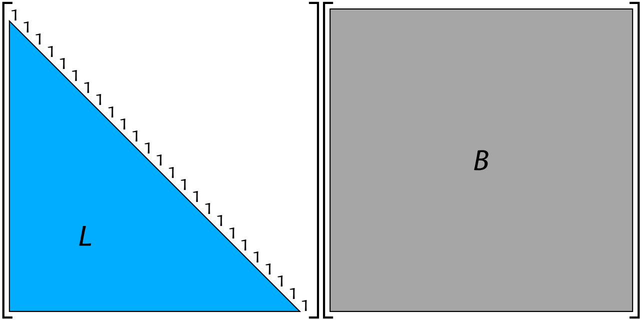
$C \leftarrow C+$ `xGER(A[:,k], B[k:])`



Algorithm for P_{ij}

- Repeat k/b times:
- Broadcast $A[:,k]$
- Broadcast $B[k,:]$
- `xGEMM(...)`
- Pipeline
  - Simpler

Level-3 Triangular Solve (`xTRSM`)

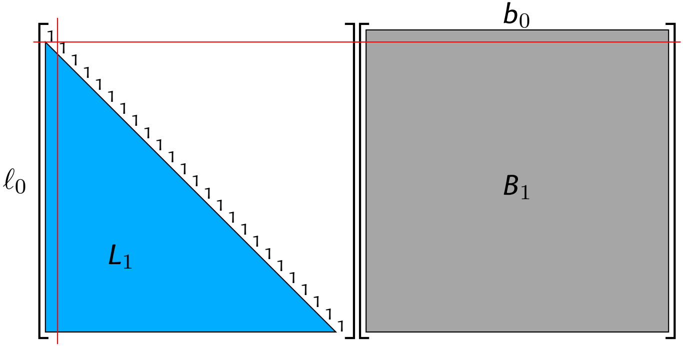


$$\begin{pmatrix} 1 & \\ \ell_0 & L_1 \end{pmatrix} \begin{pmatrix} x_0 \\ X_1 \end{pmatrix} = \begin{pmatrix} b_0 \\ B_1 \end{pmatrix}$$

$$x_0 = b_0$$
$$L_1 X_1 = B_1 - \ell_0 x_0$$

Iterative algorithm

Level-3 Triangular Solve (`xTRSM`)



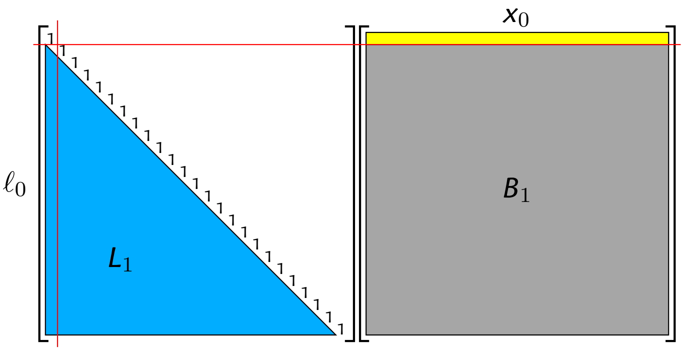
$$\begin{pmatrix} 1 & \\ \ell_0 & L_1 \end{pmatrix} \begin{pmatrix} x_0 \\ X_1 \end{pmatrix} = \begin{pmatrix} b_0 \\ B_1 \end{pmatrix}$$

$$x_0 = b_0$$

$$L_1 X_1 = B_1 - \ell_0 x_0$$

Iterative algorithm

Level-3 Triangular Solve (`xTRSM`)

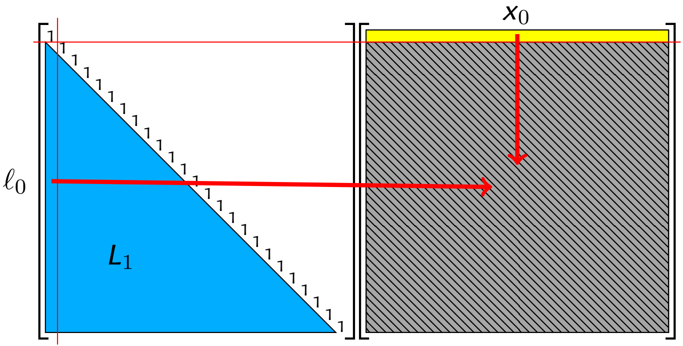


$$\begin{pmatrix} 1 & \\ \ell_0 & L_1 \end{pmatrix} \begin{pmatrix} x_0 \\ X_1 \end{pmatrix} = \begin{pmatrix} b_0 \\ B_1 \end{pmatrix}$$

$$x_0 = b_0$$
$$L_1 X_1 = B_1 - \ell_0 x_0$$

Iterative algorithm

Level-3 Triangular Solve (`xTRSM`)



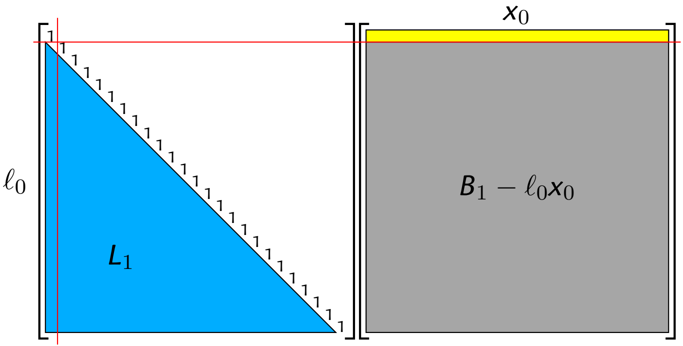
$$\begin{pmatrix} 1 & \\ \ell_0 & L_1 \end{pmatrix} \begin{pmatrix} x_0 \\ X_1 \end{pmatrix} = \begin{pmatrix} b_0 \\ B_1 \end{pmatrix}$$

$$x_0 = b_0$$
$$L_1 X_1 = B_1 - \ell_0 x_0$$

Iterative algorithm

1. $B_1 \leftarrow B_1 - \ell_0 x_0$
   - level-2 `xGER`

Level-3 Triangular Solve (`xTRSM`)



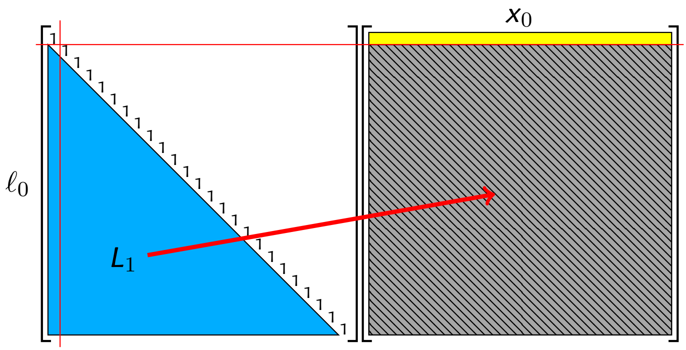
$$\begin{pmatrix} 1 & \\ \ell_0 & L_1 \end{pmatrix} \begin{pmatrix} x_0 \\ X_1 \end{pmatrix} = \begin{pmatrix} b_0 \\ B_1 \end{pmatrix}$$

$$x_0 = b_0$$
$$L_1 X_1 = B_1 - \ell_0 x_0$$

Iterative algorithm

1. $B_1 \leftarrow B_1 - \ell_0 x_0$
   - level-2 `xGER`

Level-3 Triangular Solve (`xTRSM`)



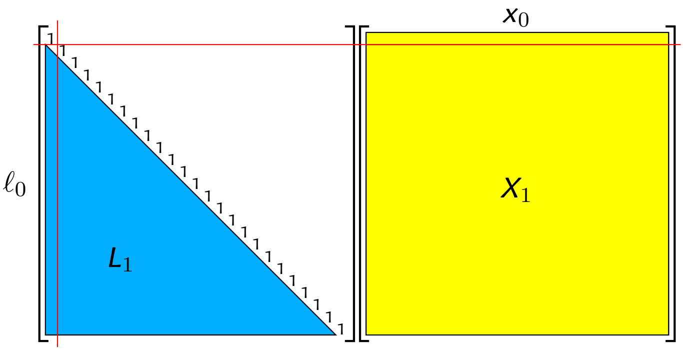
$$\begin{pmatrix} 1 & \\ \ell_0 & L_1 \end{pmatrix} \begin{pmatrix} x_0 \\ X_1 \end{pmatrix} = \begin{pmatrix} b_0 \\ B_1 \end{pmatrix}$$

$$x_0 = b_0$$
$$L_1 X_1 = B_1 - \ell_0 x_0$$

Iterative algorithm

1. $B_1 \leftarrow B_1 - \ell_0 x_0$
   - level-2 `xGER`
2. Solve $L_1 X_1 = B_1 - \ell_0 x_0$
   - level-3 `xTRSM`

Level-3 Triangular Solve (`xTRSM`)



$$\begin{pmatrix} 1 & \\ \ell_0 & L_1 \end{pmatrix} \begin{pmatrix} x_0 \\ X_1 \end{pmatrix} = \begin{pmatrix} b_0 \\ B_1 \end{pmatrix}$$

$$x_0 = b_0$$
$$L_1 X_1 = B_1 - \ell_0 x_0$$

Iterative algorithm

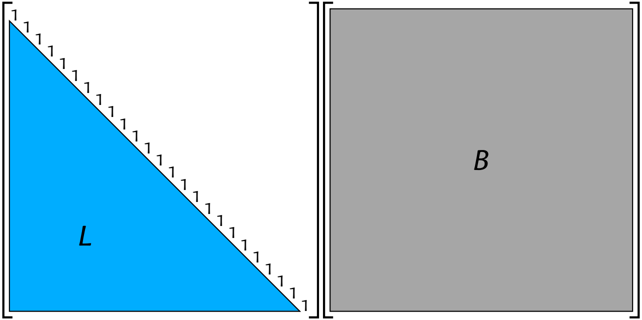
1. $B_1 \leftarrow B_1 - \ell_0 x_0$
   - level-2 `xGER`
2. Solve $L_1 X_1 = B_1 - \ell_0 x_0$
   - level-3 `xTRSM`

Level-3 Triangular Solve (`xTRSM`)

Complexity Analysis

- `xGER` runs in time $T_{ger} \leq \alpha mn$.
- `xTRSM(m, n) == xGER(m, n) + ... + xGER(1, n)`
- $T_{trsm} \leq \frac{\alpha}{2} m^2 n$

Blocked Level-3 Triangular Solve (`xTRSM`)



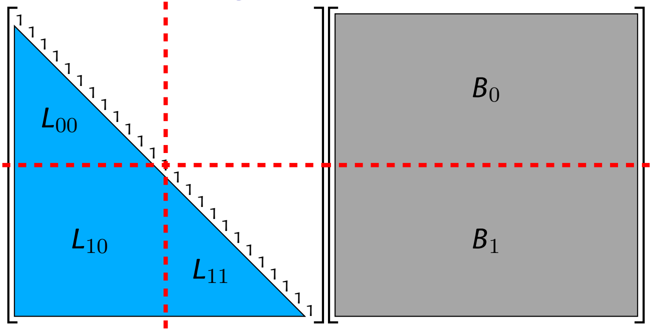
$$\begin{pmatrix} L_{00} & \\ L_{10} & L_{11} \end{pmatrix} \begin{pmatrix} X_0 \\ X_1 \end{pmatrix} = \begin{pmatrix} B_0 \\ B_1 \end{pmatrix}$$

$$L_{00} X_0 = B_0$$

$$L_{11} X_1 = B_1 - L_{10} X_0$$

Blocked algorithm

Blocked Level-3 Triangular Solve (`xTRSM`)



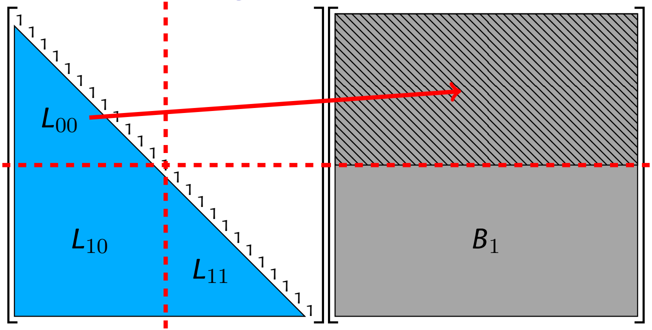
$$\begin{pmatrix} L_{00} & \\ L_{10} & L_{11} \end{pmatrix} \begin{pmatrix} X_0 \\ X_1 \end{pmatrix} = \begin{pmatrix} B_0 \\ B_1 \end{pmatrix}$$

$$L_{00} X_0 = B_0$$

$$L_{11} X_1 = B_1 - L_{10} X_0$$

Blocked algorithm

Blocked Level-3 Triangular Solve (`xTRSM`)



$$\begin{pmatrix} L_{00} & \\ L_{10} & L_{11} \end{pmatrix} \begin{pmatrix} X_0 \\ X_1 \end{pmatrix} = \begin{pmatrix} B_0 \\ B_1 \end{pmatrix}$$

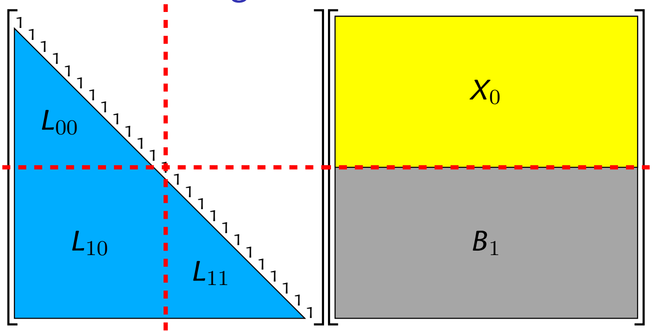
$$L_{00}X_0 = B_0$$

$$L_{11}X_1 = B_1 - L_{10}X_0$$

Blocked algorithm

1. Solve $L_{00}X_0 = B_0$
   - level-3 `TRSM`

Blocked Level-3 Triangular Solve (`xTRSM`)



$$\begin{pmatrix} L_{00} & \\ L_{10} & L_{11} \end{pmatrix} \begin{pmatrix} X_0 \\ X_1 \end{pmatrix} = \begin{pmatrix} B_0 \\ B_1 \end{pmatrix}$$

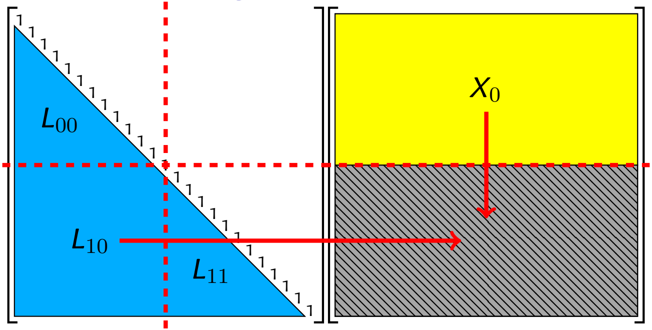
$$L_{00} X_0 = B_0$$

$$L_{11} X_1 = B_1 - L_{10} X_0$$

Blocked algorithm

1. Solve $L_{00} X_0 = B_0$
   - level-3 `TRSM`

Blocked Level-3 Triangular Solve (`xTRSM`)



$$\begin{pmatrix} L_{00} & \\ L_{10} & L_{11} \end{pmatrix} \begin{pmatrix} X_0 \\ X_1 \end{pmatrix} = \begin{pmatrix} B_0 \\ B_1 \end{pmatrix}$$

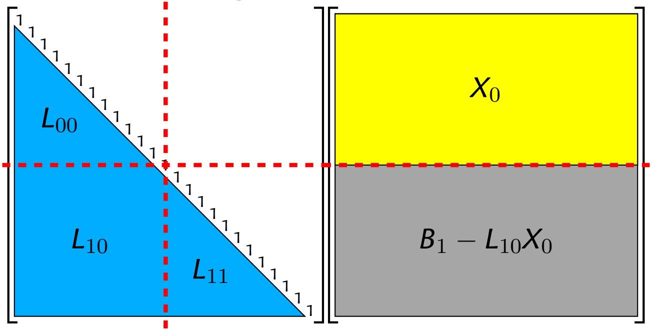
$$L_{00}X_0 = B_0$$

$$L_{11}X_1 = B_1 - L_{10}X_0$$

Blocked algorithm

1. Solve $L_{00}X_0 = B_0$
   - level-3 `TRSM`
2. $B_1 \leftarrow B_1 - L_{10}X_0$
   - level-3 `GEMM`

Blocked Level-3 Triangular Solve (`xTRSM`)



$$\begin{pmatrix} L_{00} & \\ L_{10} & L_{11} \end{pmatrix} \begin{pmatrix} X_0 \\ X_1 \end{pmatrix} = \begin{pmatrix} B_0 \\ B_1 \end{pmatrix}$$

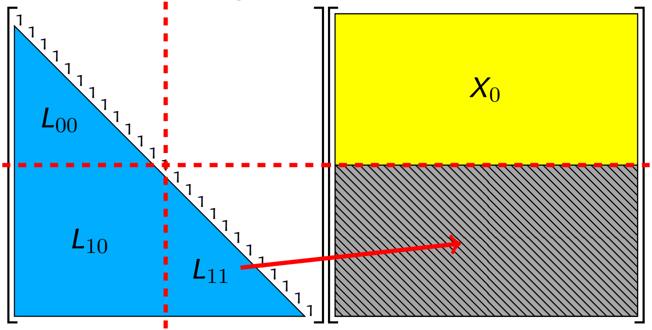
$$L_{00}X_0 = B_0$$

$$L_{11}X_1 = B_1 - L_{10}X_0$$

Blocked algorithm

1. Solve $L_{00}X_0 = B_0$
   - level-3 `TRSM`
2. $B_1 \leftarrow B_1 - L_{10}X_0$
   - level-3 `GEMM`

Blocked Level-3 Triangular Solve (`xTRSM`)



$$\begin{pmatrix} L_{00} & \\ L_{10} & L_{11} \end{pmatrix} \begin{pmatrix} X_0 \\ X_1 \end{pmatrix} = \begin{pmatrix} B_0 \\ B_1 \end{pmatrix}$$

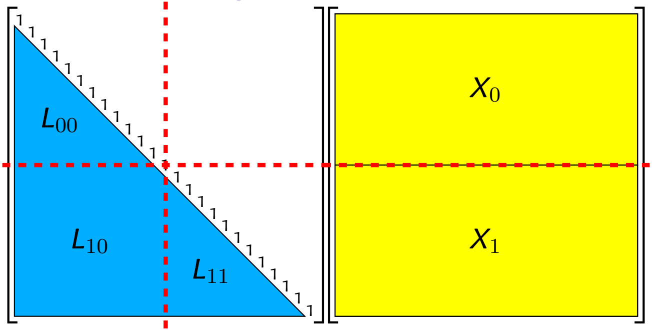
$$L_{00} X_0 = B_0$$

$$L_{11} X_1 = B_1 - L_{10} X_0$$

Blocked algorithm

1. Solve $L_{00} X_0 = B_0$
   - level-3 `TRSM`
2. $B_1 \leftarrow B_1 - L_{10} X_0$
   - level-3 `GEMM`
3. Solve $L_{11} X_1 = B_1 - L_{10} X_0$
   - level-3 `TRSM`

Blocked Level-3 Triangular Solve (`xTRSM`)



$$\begin{pmatrix} L_{00} & \\ L_{10} & L_{11} \end{pmatrix} \begin{pmatrix} X_0 \\ X_1 \end{pmatrix} = \begin{pmatrix} B_0 \\ B_1 \end{pmatrix}$$

$$L_{00}X_0 = B_0$$

$$L_{11}X_1 = B_1 - L_{10}X_0$$

Blocked algorithm

1. Solve $L_{00}X_0 = B_0$
   - level-3 `TRSM`
2. $B_1 \leftarrow B_1 - L_{10}X_0$
   - level-3 `GEMM`
3. Solve $L_{11}X_1 = B_1 - L_{10}X_0$
   - level-3 `TRSM`

Blocked Level-3 Triangular Solve (`xTRSM`)

Complexity Analysis

- `xGER` runs in time $T_{ger} \leq \alpha mn$
- `xGEMM` runs in time $T_{gemm} \leq \beta mnk$
- (unblocked) `xTRSM(m, n)` runs in time $T \leq \alpha m^2 n/2$
- If $m \leq m_0$, use unblocked algorithm (too small)
  - $T_m = \frac{\alpha}{2} m^2 n$
- Otherwise, use recursive blocked algorithm
  - $T_m = 2T_{m/2} + \beta \left(\frac{m}{2}\right)^2 n + \mathcal{O}(1)$

Asymptotic performance

- Assume $T_m = \gamma m^2 n$ (as in unblocked case)
- $\gamma m^2 n = 2\gamma \left(\frac{m}{2}\right)^2 n + \beta \left(\frac{m}{2}\right)^2 n + \mathcal{O}(1)$
- $\gamma = \beta/2$ (for large m)

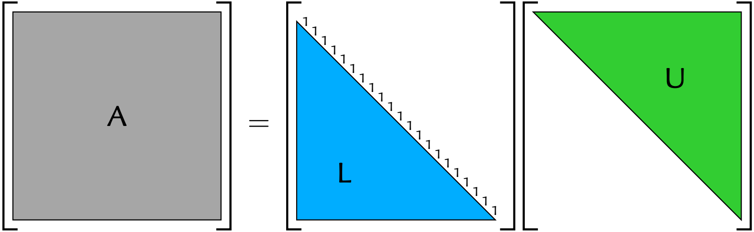
Solving Actual Problems

L	A	P	A	C	K
L	-A	P	-A	C	-K
L	A	P	A	-C	-K
L	-A	P	-A	-C	K
L	A	-P	A	C	K
L	-A	-P	-A	C	-K

LAPACK: Linear Algebra PACKage

- Development started in the 1990's (still active)
- Built upon the BLAS (Fortran: column-major)
- Solve linear systems, least-squares, eigenvalues, etc.
- **Main algorithmic idea**: use level-3 BLAS
  - High arithmetic intensity
  - Significant performance gain over naive code

LU Factorization



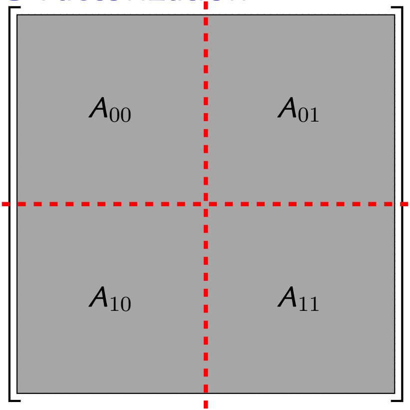
- Main tool to solve $Ax = b$ when A is invertible
- **DISCLAIMER**:
  - This presentation ignores pivoting
  - Pivoting is required for numerical stability
  - In parallel, pivoting is a pain in the `[---REDACTED---]`

LU Factorization

$$\begin{bmatrix} A \end{bmatrix}$$

Recursive algorithm

LU Factorization

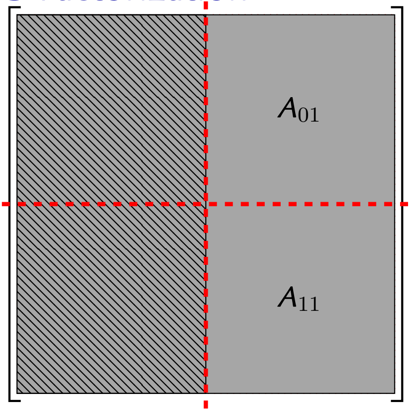


Recursive algorithm

$$\begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} = \begin{pmatrix} L_{00} & \\ L_{10} & L_{11} \end{pmatrix} \begin{pmatrix} U_{00} & U_{01} \\ & U_{11} \end{pmatrix}$$

$$L_{00}U_{01} = A_{01}$$
$$L_{11}U_{11} = A_{11} - L_{10}U_{01}$$

LU Factorization



Recursive algorithm

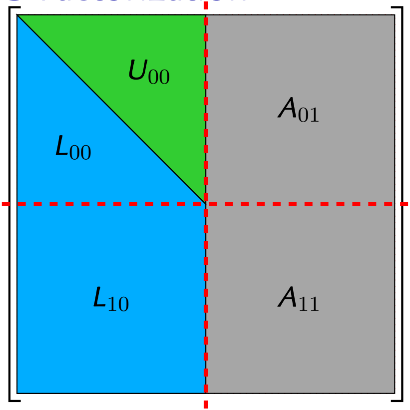
1. Factorize left half

$$\begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} = \begin{pmatrix} L_{00} & \\ L_{10} & L_{11} \end{pmatrix} \begin{pmatrix} U_{00} & U_{01} \\ & U_{11} \end{pmatrix}$$

$$L_{00}U_{01} = A_{01}$$

$$L_{11}U_{11} = A_{11} - L_{10}U_{01}$$

LU Factorization



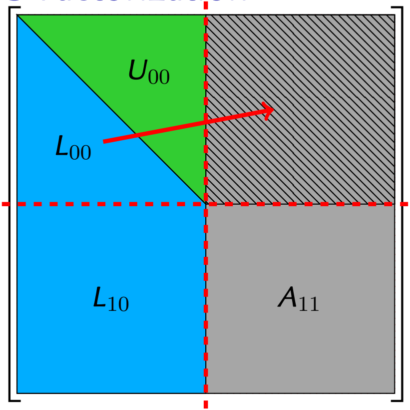
Recursive algorithm

1. Factorize left half

$$\begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} = \begin{pmatrix} L_{00} & \\ L_{10} & L_{11} \end{pmatrix} \begin{pmatrix} U_{00} & U_{01} \\ & U_{11} \end{pmatrix}$$

$$L_{00}U_{01} = A_{01}$$
$$L_{11}U_{11} = A_{11} - L_{10}U_{01}$$

LU Factorization



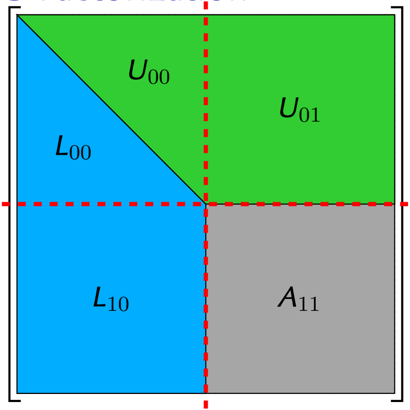
Recursive algorithm

1. Factorize left half
2. Solve $L_{00}U_{01} = A_{01}$
   - level-3 `TRSM`

$$\begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} = \begin{pmatrix} L_{00} & \\ L_{10} & L_{11} \end{pmatrix} \begin{pmatrix} U_{00} & U_{01} \\ & U_{11} \end{pmatrix}$$

$$L_{00}U_{01} = A_{01}$$
$$L_{11}U_{11} = A_{11} - L_{10}U_{01}$$

LU Factorization



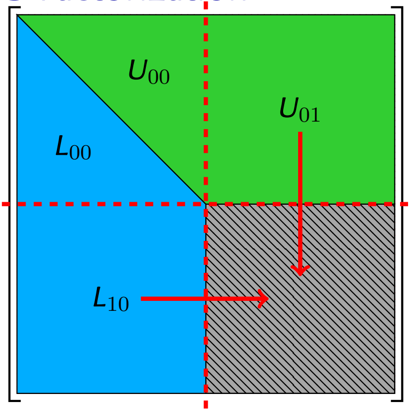
Recursive algorithm

1. Factorize left half
2. Solve $L_{00}U_{01} = A_{01}$
   - level-3 `TRSM`

$$\begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} = \begin{pmatrix} L_{00} & \\ L_{10} & L_{11} \end{pmatrix} \begin{pmatrix} U_{00} & U_{01} \\ & U_{11} \end{pmatrix}$$

$$L_{00}U_{01} = A_{01}$$
$$L_{11}U_{11} = A_{11} - L_{10}U_{01}$$

LU Factorization



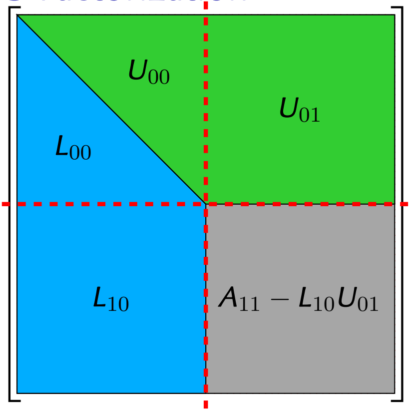
Recursive algorithm

1. Factorize left half
2. Solve $L_{00}U_{01} = A_{01}$
   - level-3 TRSM
3. $A_{11} \leftarrow A_{11} - L_{10}U_{01}$
   - level-3 GEMM

$$\begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} = \begin{pmatrix} L_{00} & \\ L_{10} & L_{11} \end{pmatrix} \begin{pmatrix} U_{00} & U_{01} \\ & U_{11} \end{pmatrix}$$

$$L_{00}U_{01} = A_{01}$$
$$L_{11}U_{11} = A_{11} - L_{10}U_{01}$$

LU Factorization



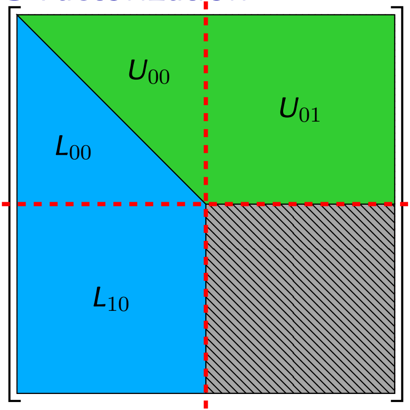
Recursive algorithm

1. Factorize left half
2. Solve $L_{00}U_{01} = A_{01}$
   - level-3 `TRSM`
3. $A_{11} \leftarrow A_{11} - L_{10}U_{01}$
   - level-3 `GEMM`

$$\begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} = \begin{pmatrix} L_{00} & \\ L_{10} & L_{11} \end{pmatrix} \begin{pmatrix} U_{00} & U_{01} \\ & U_{11} \end{pmatrix}$$

$$L_{00}U_{01} = A_{01}$$
$$L_{11}U_{11} = A_{11} - L_{10}U_{01}$$

LU Factorization



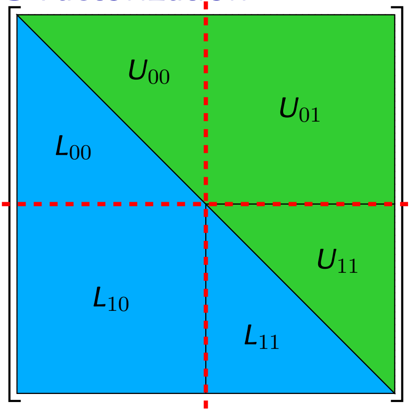
Recursive algorithm

1. Factorize left half
2. Solve $L_{00}U_{01} = A_{01}$
   - level-3 `TRSM`
3. $A_{11} \leftarrow A_{11} - L_{10}U_{01}$
   - level-3 `GEMM`
4. Factorize $A_{11} - L_{10}U_{01}$

$$\begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} = \begin{pmatrix} L_{00} & \\ L_{10} & L_{11} \end{pmatrix} \begin{pmatrix} U_{00} & U_{01} \\ & U_{11} \end{pmatrix}$$

$$L_{00}U_{01} = A_{01}$$
$$L_{11}U_{11} = A_{11} - L_{10}U_{01}$$

LU Factorization



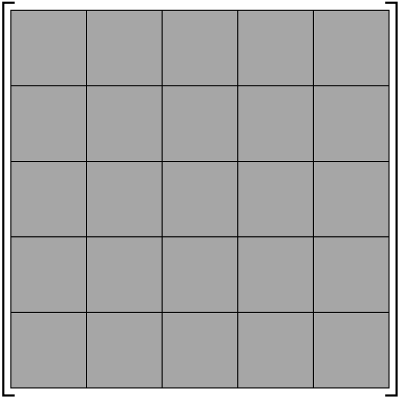
Recursive algorithm

1. Factorize left half
2. Solve $L_{00}U_{01} = A_{01}$
   - level-3 `TRSM`
3. $A_{11} \leftarrow A_{11} - L_{10}U_{01}$
   - level-3 `GEMM`
4. Factorize $A_{11} - L_{10}U_{01}$

$$\begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} = \begin{pmatrix} L_{00} & \\ L_{10} & L_{11} \end{pmatrix} \begin{pmatrix} U_{00} & U_{01} \\ & U_{11} \end{pmatrix}$$

$$\begin{aligned} L_{00}U_{01} &= A_{01} \\ L_{11}U_{11} &= A_{11} - L_{10}U_{01} \end{aligned}$$

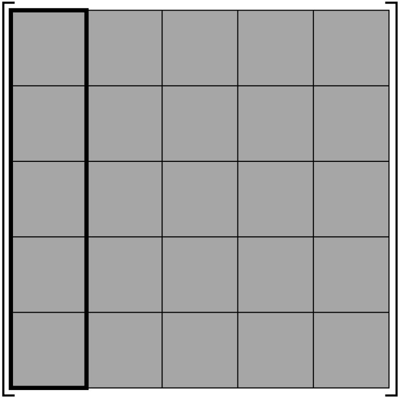
Distributed LU Factorization



Remarks

- 2D distribution of the matrix

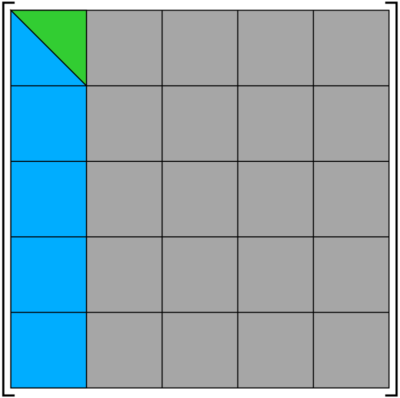
Distributed LU Factorization



Remarks

- 2D distribution of the matrix
- Proc. on a column cooperate: panel factorization

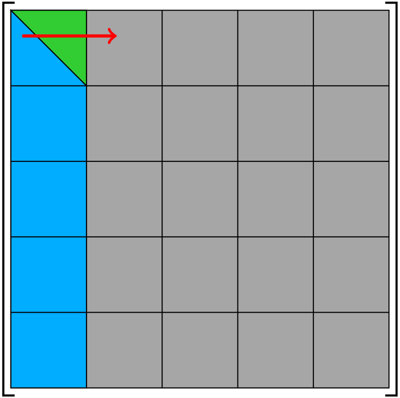
Distributed LU Factorization



Remarks

- 2D distribution of the matrix
- Proc. on a column cooperate: panel factorization

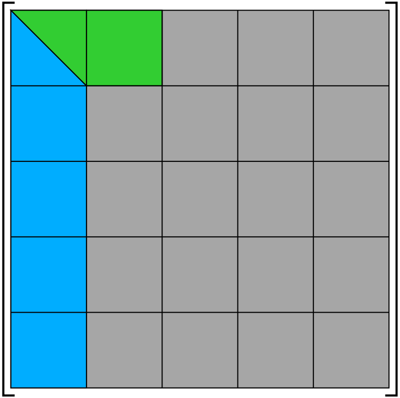
Distributed LU Factorization



Remarks

- 2D distribution of the matrix
- Proc. on a column cooperate: panel factorization
- Data flow (pipeline)

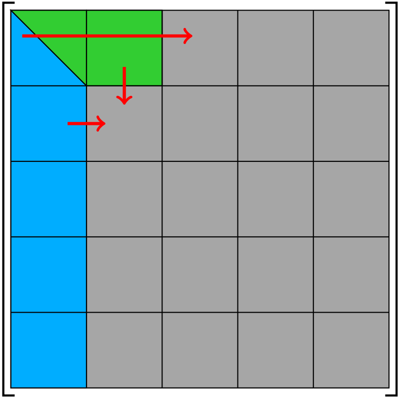
Distributed LU Factorization



Remarks

- 2D distribution of the matrix
- Proc. on a column cooperate: panel factorization
- Data flow (pipeline)

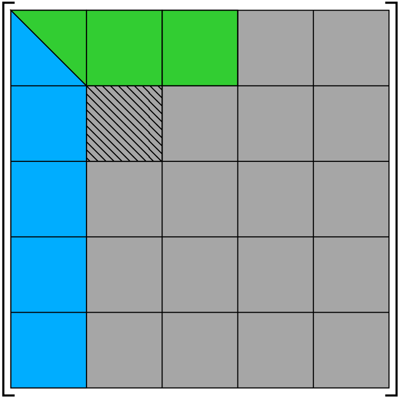
Distributed LU Factorization



Remarks

- 2D distribution of the matrix
- Proc. on a column cooperate: panel factorization
- Data flow (pipeline)

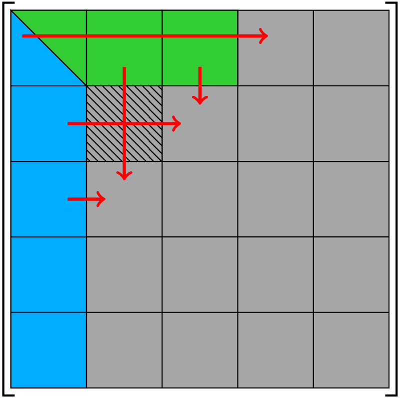
Distributed LU Factorization



Remarks

- 2D distribution of the matrix
- Proc. on a column cooperate: panel factorization
- Data flow (pipeline)

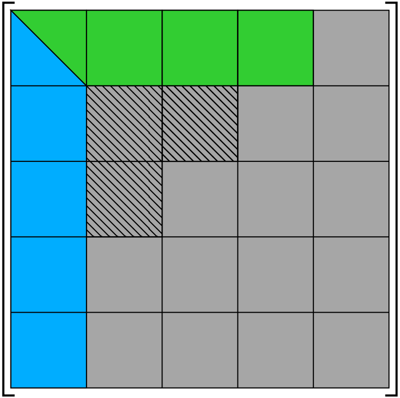
Distributed LU Factorization



Remarks

- 2D distribution of the matrix
- Proc. on a column cooperate: panel factorization
- Data flow (pipeline)

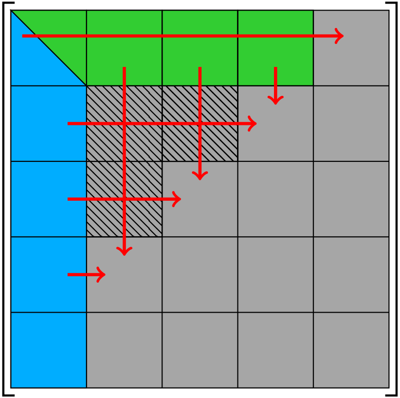
Distributed LU Factorization



Remarks

- 2D distribution of the matrix
- Proc. on a column cooperate: panel factorization
- Data flow (pipeline)

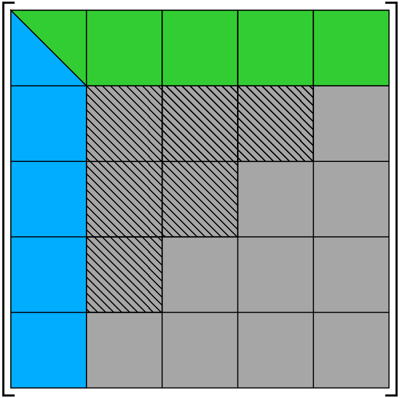
Distributed LU Factorization



Remarks

- 2D distribution of the matrix
- Proc. on a column cooperate: panel factorization
- Data flow (pipeline)

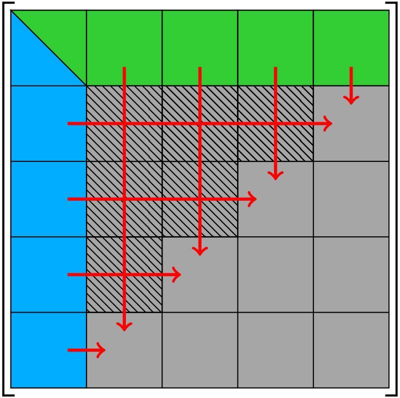
Distributed LU Factorization



Remarks

- 2D distribution of the matrix
- Proc. on a column cooperate: panel factorization
- Data flow (pipeline)

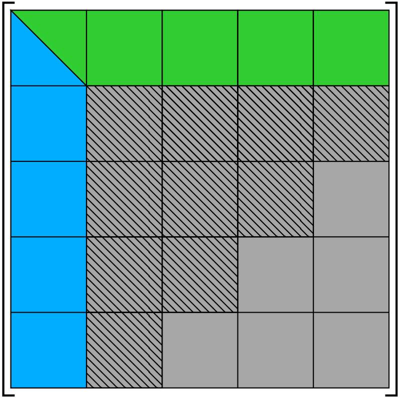
Distributed LU Factorization



Remarks

- 2D distribution of the matrix
- Proc. on a column cooperate: panel factorization
- Data flow (pipeline)

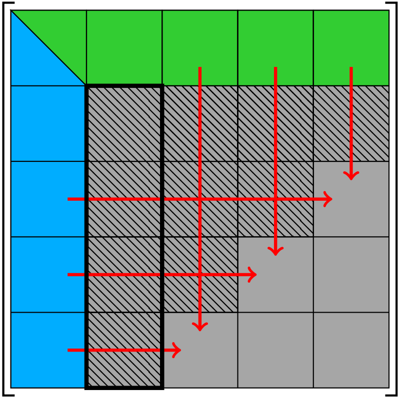
Distributed LU Factorization



Remarks

- 2D distribution of the matrix
- Proc. on a column cooperate: panel factorization
- Data flow (pipeline)

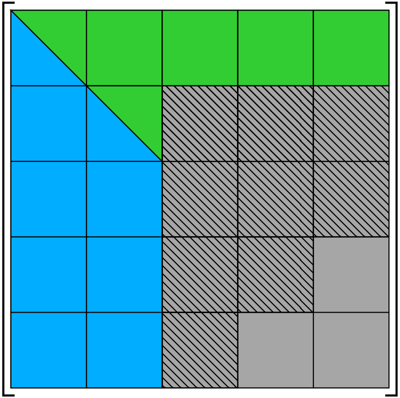
Distributed LU Factorization



Remarks

- 2D distribution of the matrix
- Proc. on a column cooperate: panel factorization
- Data flow (pipeline)
- 2nd panel factorization

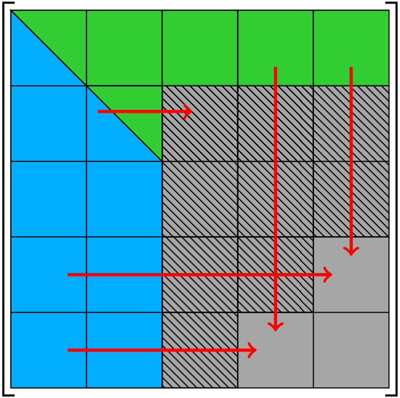
Distributed LU Factorization



Remarks

- 2D distribution of the matrix
- Proc. on a column cooperate: panel factorization
- Data flow (pipeline)
- 2nd panel factorization

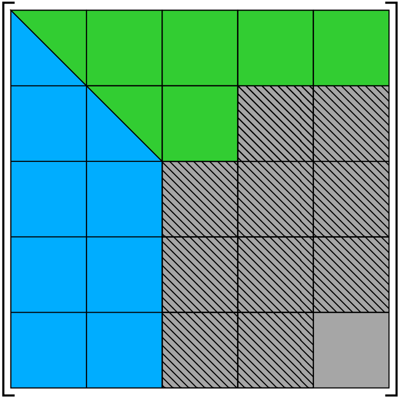
Distributed LU Factorization



Remarks

- 2D distribution of the matrix
- Proc. on a column cooperate: panel factorization
- Data flow (pipeline)
- 2nd panel factorization

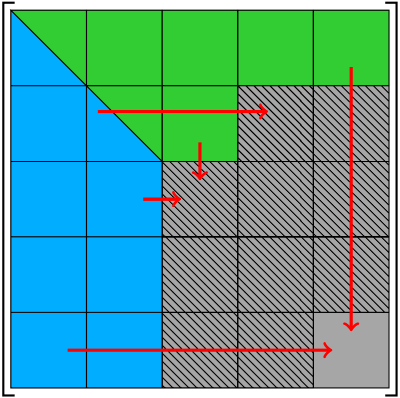
Distributed LU Factorization



Remarks

- 2D distribution of the matrix
- Proc. on a column cooperate: panel factorization
- Data flow (pipeline)
- 2nd panel factorization

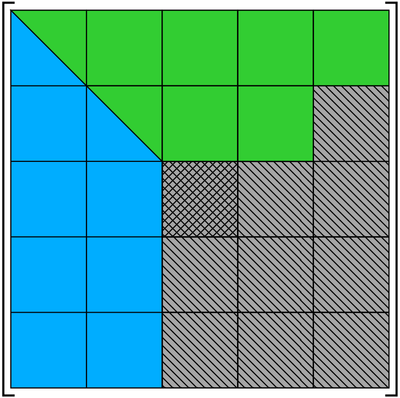
Distributed LU Factorization



Remarks

- 2D distribution of the matrix
- Proc. on a column cooperate: panel factorization
- Data flow (pipeline)
- 2nd panel factorization

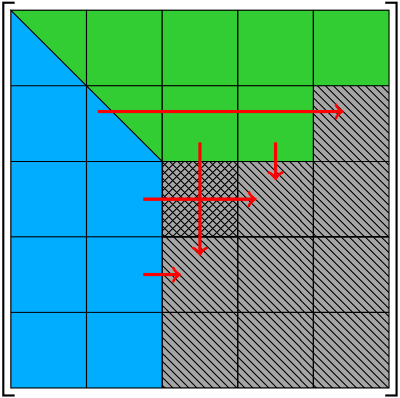
Distributed LU Factorization



Remarks

- 2D distribution of the matrix
- Proc. on a column cooperate: panel factorization
- Data flow (pipeline)
- 2nd panel factorization
- 1st row/col now inactive

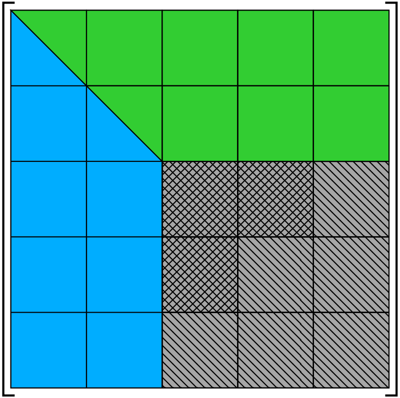
Distributed LU Factorization



Remarks

- 2D distribution of the matrix
- Proc. on a column cooperate: panel factorization
- Data flow (pipeline)
- 2nd panel factorization
- 1st row/col now inactive

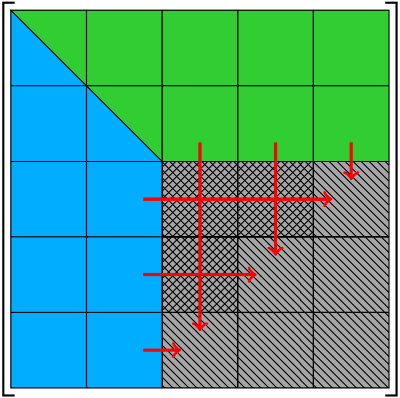
Distributed LU Factorization



Remarks

- 2D distribution of the matrix
- Proc. on a column cooperate: panel factorization
- Data flow (pipeline)
- 2nd panel factorization
- 1st row/col now inactive

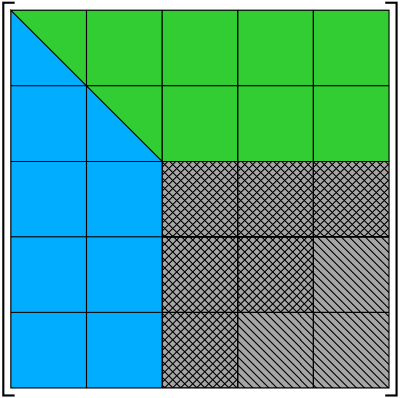
Distributed LU Factorization



Remarks

- 2D distribution of the matrix
- Proc. on a column cooperate: panel factorization
- Data flow (pipeline)
- 2nd panel factorization
- 1st row/col now inactive

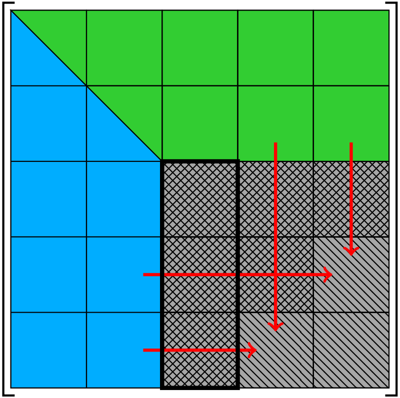
Distributed LU Factorization



Remarks

- 2D distribution of the matrix
- Proc. on a column cooperate: panel factorization
- Data flow (pipeline)
- 2nd panel factorization
- 1st row/col now inactive

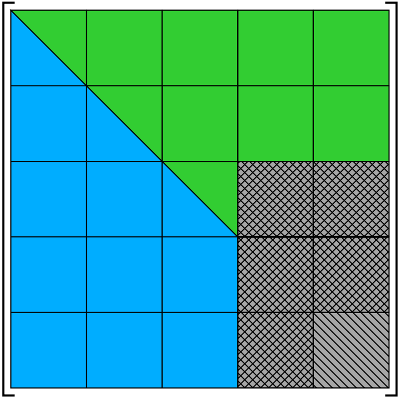
Distributed LU Factorization



Remarks

- 2D distribution of the matrix
- Proc. on a column cooperate: panel factorization
- Data flow (pipeline)
- 2nd panel factorization
- 1st row/col now inactive

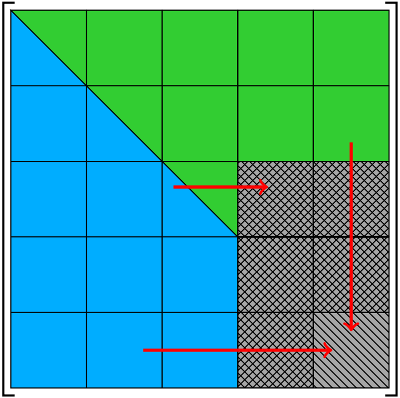
Distributed LU Factorization



Remarks

- 2D distribution of the matrix
- Proc. on a column cooperate: panel factorization
- Data flow (pipeline)
- 2nd panel factorization
- 1st row/col now inactive

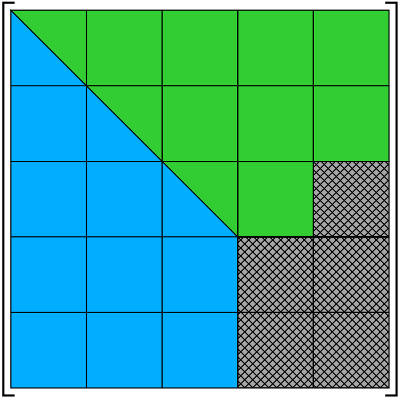
Distributed LU Factorization



Remarks

- 2D distribution of the matrix
- Proc. on a column cooperate: panel factorization
- Data flow (pipeline)
- 2nd panel factorization
- 1st row/col now inactive

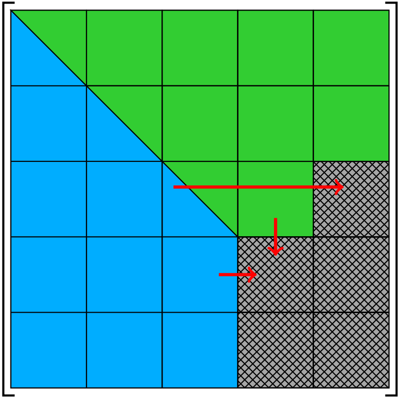
Distributed LU Factorization



Remarks

- 2D distribution of the matrix
- Proc. on a column cooperate: panel factorization
- Data flow (pipeline)
- 2nd panel factorization
- 1st row/col now inactive
- 2nd row/col now inactive

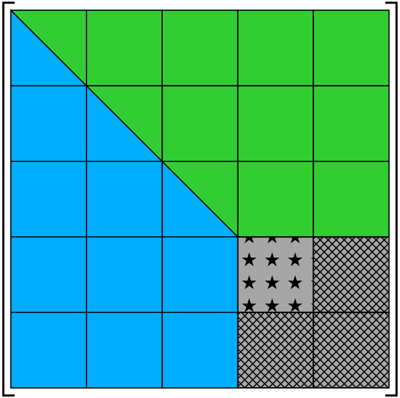
Distributed LU Factorization



Remarks

- 2D distribution of the matrix
- Proc. on a column cooperate: panel factorization
- Data flow (pipeline)
- 2nd panel factorization
- 1st row/col now inactive
- 2nd row/col now inactive

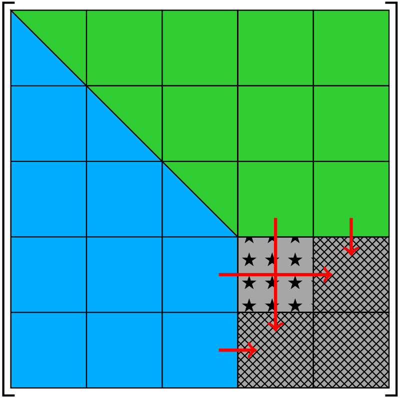
Distributed LU Factorization



Remarks

- 2D distribution of the matrix
- Proc. on a column cooperate: panel factorization
- Data flow (pipeline)
- 2nd panel factorization
- 1st row/col now inactive
- 2nd row/col now inactive

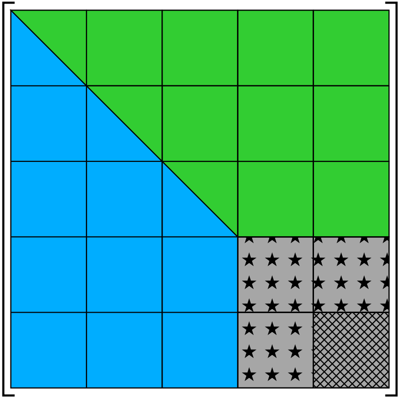
Distributed LU Factorization



Remarks

- 2D distribution of the matrix
- Proc. on a column cooperate: panel factorization
- Data flow (pipeline)
- 2nd panel factorization
- 1st row/col now inactive
- 2nd row/col now inactive

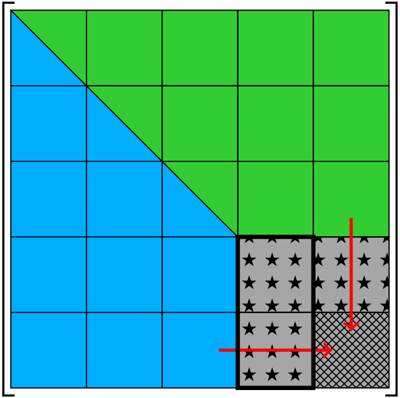
Distributed LU Factorization



Remarks

- 2D distribution of the matrix
- Proc. on a column cooperate: panel factorization
- Data flow (pipeline)
- 2nd panel factorization
- 1st row/col now inactive
- 2nd row/col now inactive

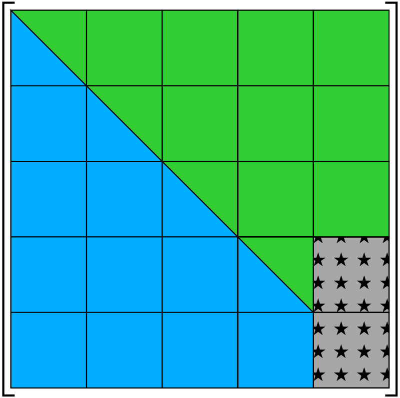
Distributed LU Factorization



Remarks

- 2D distribution of the matrix
- Proc. on a column cooperate: panel factorization
- Data flow (pipeline)
- 2nd panel factorization
- 1st row/col now inactive
- 2nd row/col now inactive

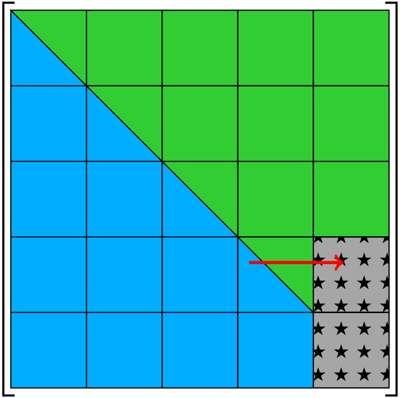
Distributed LU Factorization



Remarks

- 2D distribution of the matrix
- Proc. on a column cooperate: panel factorization
- Data flow (pipeline)
- 2nd panel factorization
- 1st row/col now inactive
- 2nd row/col now inactive

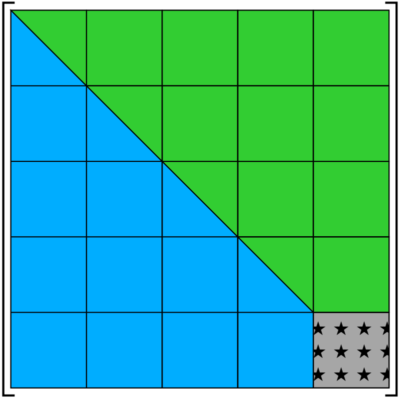
Distributed LU Factorization



Remarks

- 2D distribution of the matrix
- Proc. on a column cooperate: panel factorization
- Data flow (pipeline)
- 2nd panel factorization
- 1st row/col now inactive
- 2nd row/col now inactive

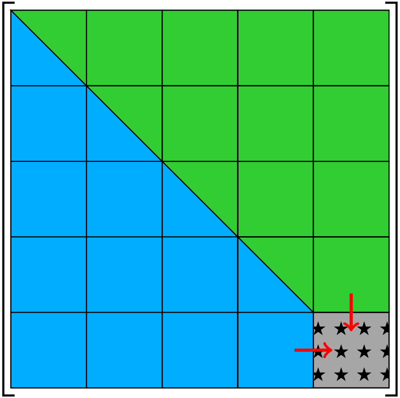
Distributed LU Factorization



Remarks

- 2D distribution of the matrix
- Proc. on a column cooperate: panel factorization
- Data flow (pipeline)
- 2nd panel factorization
- 1st row/col now inactive
- 2nd row/col now inactive

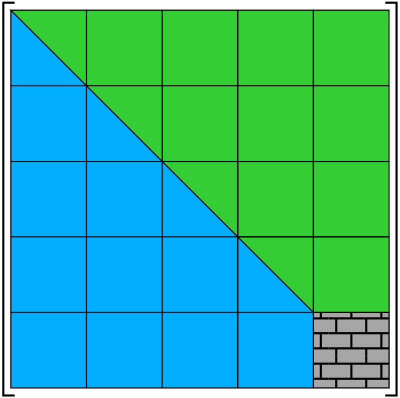
Distributed LU Factorization



Remarks

- 2D distribution of the matrix
- Proc. on a column cooperate: panel factorization
- Data flow (pipeline)
- 2nd panel factorization
- 1st row/col now inactive
- 2nd row/col now inactive

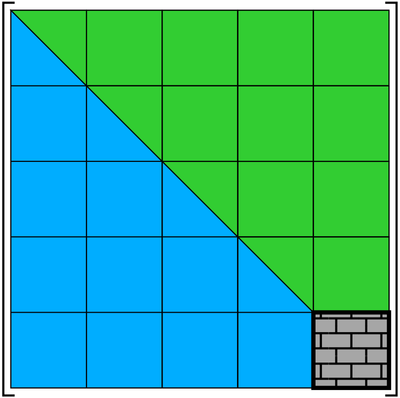
Distributed LU Factorization



Remarks

- 2D distribution of the matrix
- Proc. on a column cooperate: panel factorization
- Data flow (pipeline)
- 2nd panel factorization
- 1st row/col now inactive
- 2nd row/col now inactive

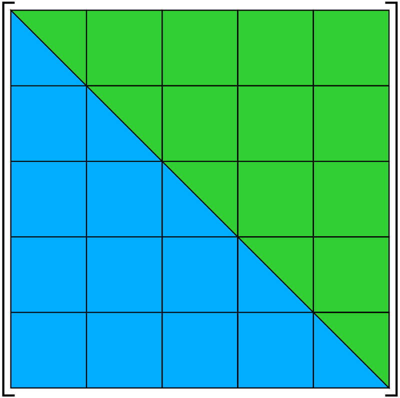
Distributed LU Factorization



Remarks

- 2D distribution of the matrix
- Proc. on a column cooperate: panel factorization
- Data flow (pipeline)
- 2nd panel factorization
- 1st row/col now inactive
- 2nd row/col now inactive

Distributed LU Factorization



Remarks

- 2D distribution of the matrix
- Proc. on a column cooperate: panel factorization
- Data flow (pipeline)
- 2nd panel factorization
- 1st row/col now inactive
- 2nd row/col now inactive

Distributed-memory LU

- Uses 2D cyclic block distribution to keep everyone busy
- Implented in `HPL` , `ScaLAPACK`
- Major benchmark of HPC machines ("LINPACK")

Iterative Methods in Linear Algebra

Important family of algorithms

- Solving $Ax = b$ or $Ax = \lambda x$
- Conjugate gradient, Lanczos, GMRES, etc.

Principle

- Choose an initial vector x_0
- Iterate the sequence $x_{i+1} = Ax$
- At each iteration, build an approximation of the solution
- Stop when the process has converged
  - If it ever does...

Main point

- Bulk of workload: $y \leftarrow Ax$
  - Matrix-vector product (`GEMV`)
  - Fast when A is a **sparse matrix**

Sparse Linear Algebra

- Many physical situations yields sparse linear systems
- E.g. Finite Element Method

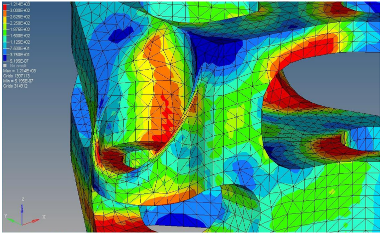


Image: (c) PyCAE

- One variable per cell
- A single equation only relates variables of neighboring cells
- $Ax = b$ with sparse A (mostly zero coefficients)

Sparse Matrices

Goals when dealing with sparse matrices

- Don't store 0
- Don't compute $0 + x$ and $0 \times x$

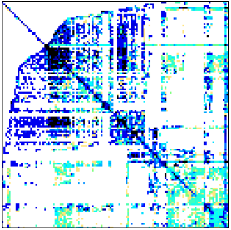


Image: (c) T. davis

Most important parameter: **density**

- Proportion of non-zero coefficients
- density < 0.01 (say) $\rightsquigarrow$ sparse

Sparse Matrix Storage, Triplet Form

Simplest Possible Idea

List of triplets (i, j, x) for each non-zero entry $A[i, j] = x$.

```
struct triplet {
    int n, m, nz;
    int nzmax;      /* allocated size for Ai, Aj, Ax */
    int *Ai;
    int *Aj;
    double *Ax;
};                  /* note: order is irrelevant */
```

Assuming `int` indices (i, j) and `double` coefficients (x),

`sizeof(A)` $= 16|A| + 16$ bytes (regardless of n, m)

Sparse Matrix Storage, Triplet Form (continued)

Pros

- Friendly format for I/O
- Repeated i, j can be allowed (coefficients are summed)

Possible file format :

```
n m
i_0 j_0 x_0
i_1 j_1 x_1
.....
```

Cons

- Limited operations. $x \leftarrow A[i, j]$ is impossible!

Storage, Triplet Form, Operations

- $A \leftarrow A^t$

```
int *tmp = Ai;        /* constant-time! */
Ai = Aj;
Aj = tmp;
```

Storage, Triplet Form, Operations

- $A \leftarrow A^t$
- $A \leftarrow \lambda A$

```
for (int p = 0; p < nz; p++)
    Ax[p] *= lambda;
```

Storage, Triplet Form, Operations

- $A \leftarrow A^t$
- $A \leftarrow \lambda A$
- $y \leftarrow y + xA$

```
for (int p = 0; p < nz; p++)
    y[Aj[p]] += x[Ai[p]] * Ax[p];
```

Storage, Triplet Form, Operations

- $A \leftarrow A^t$
- $A \leftarrow \lambda A$
- $y \leftarrow y + xA$
- $y \leftarrow y + Ax$

```
for (int p = 0; p < nz; p++)
    y[Ai[p]] += x[Aj[p]] * Ax[p];
```

Storage, Triplet Form, Operations

- $A \leftarrow A^t$
- $A \leftarrow \lambda A$
- $y \leftarrow y + xA$
- $y \leftarrow y + Ax$
- $A \leftarrow PAQ$

```
for (int p = 0; p < nz; p++) {
    Ai[p] = Pinv[Ai[p]];
    Aj[p] = Qinv[Aj[p]];
}
```

Storage, Triplet Form, Operations

- $A \leftarrow A^t$
- $A \leftarrow \lambda A$
- $y \leftarrow y + xA$
- $y \leftarrow y + Ax$
- $A \leftarrow PAQ$
- $A \leftarrow B + C$

Essentially concatenate B and C.

Storage, Triplet Form, Operations

- $A \leftarrow A^t$
- $A \leftarrow \lambda A$
- $y \leftarrow y + xA$
- $y \leftarrow y + Ax$
- $A \leftarrow PAQ$
- $A \leftarrow B + C$

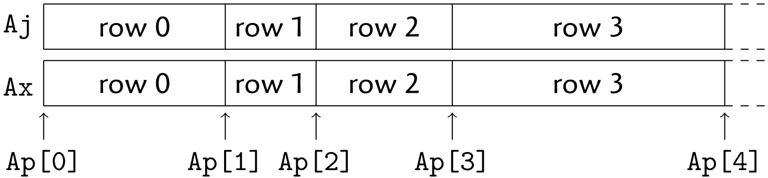
Main Limitations

- Everything else is impossible!
- Wasteful storage

Storage, Compressed Sparse Rows (CSR) Form

Pack Rows Together

```c
struct csr {
    int n, m;
    int *Ap;      // row pointers    (size n+1)
    int *Aj;      // column indices (size nz)
    double *Ax;  // coefficients   (size nz)
};
```



`sizeof(A)` $= 12|A| + 4n + 12$ bytes (regardless of m)

Storage, CSR Form, Operations

- $x \leftarrow |A|$

```
x = Ap[n + 1];
```

Storage, CSR Form, Operations

- $x \leftarrow |A|$
- $y \leftarrow y + xA$

```
for (int i = 0; i < n; i++)
    for (int p = Ap[i]; p < Ap[i + 1]; p++)
        y[Aj[p]] += x[i] * Ax[p];
```

Storage, CSR Form, Operations

- $x \leftarrow |A|$
- $y \leftarrow y + xA$
- $y \leftarrow y + Ax$

```
for (int i = 0; i < n; i++)
    for (int p = Ap[i]; p < Ap[i + 1]; p++)
        y[i] += x[Aj[p]] * Ax[p];
```

Storage, CSR Form, Operations

- $x \leftarrow |A|$
- $y \leftarrow y + xA$
- $y \leftarrow y + Ax$
- $A \leftarrow AQ$

```
for (int p = 0; p < Ap[n + 1]; p++)
    Aj[p] = Qinv[Aj[p]];
```

Storage, CSR Form, Operations

- $x \leftarrow |A|$
- $y \leftarrow y + xA$
- $y \leftarrow y + Ax$
- $A \leftarrow AQ$

Remarks

- Order of entries in row irrelevant
- Duplicates are allowed (but wasteful)
- Explicit 0 entries are allowed (but wasteful)

Storage, CSR Form, Operations

- $x \leftarrow |A|$
- $y \leftarrow y + xA$
- $y \leftarrow y + Ax$
- $A \leftarrow AQ$

Main Limitation

- Transpose, row permutation not in place.
  - Lazy solution: convert to triplets, work, convert back to CSR
- $x \leftarrow A[i,j]$ still impossible...
  - Direct access to rows/iteration over rows is possible.

Matrix-Vector product (GEMV)

$y \leftarrow y + \alpha Ax$

Dense GEMV

- All matrix coeffs are contiguous in memory

```
for (int i = 0; i < n; i++)                        // 3nm FLOP
    for (int j = 0; j < m; j++)
        y[i] += alpha * A[i*m + j] * x[j];
```

Sparse GEMV

```
for (int k = 0; k < nz; k++)                       // 3nz FLOP
    y[Ai[k]] += alpha * Ax[k] * x[Aj[k]];
```

Recurring problem in HPC

Iterative methods with sparse matrices

- compute $x_{i+1} = Ax_i$
- For $i = 1, 2, 3, 4, 5, \ldots$
- With large, sparse matrix A
- In parallel
  - Need x_{i+1} to start x_{i+2}
  - Iterations are necessarily sequential
  - $\Rightarrow$ Parallelize the **matrix-vector product**

Data parallelism

- A is **distributed** between processes (how?)
- x_i is **distributed** between processes (how?)
- x_{i+1} must be distributed identically to iterate

Distributed Matrix-Vector Product

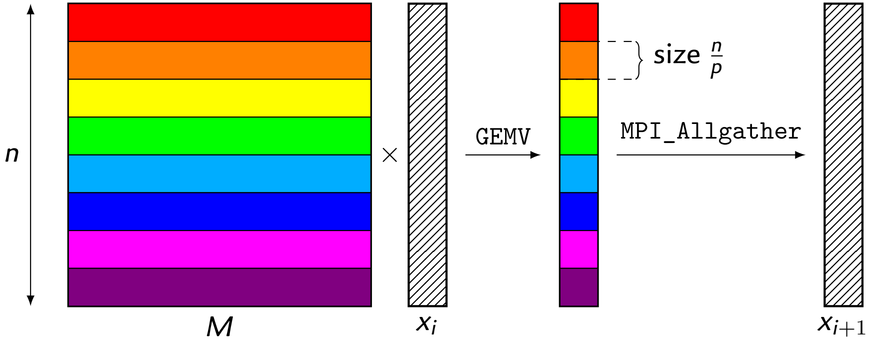
1D block distribution (per rows)

Data distribution

M : 1D block distribution (blocks of rows)

x : owned by all processes

y : owned by all processes



Machine parameters

- C = processor FLOP/s
- D = network bandwidth (float / s)

Matrix characteristics

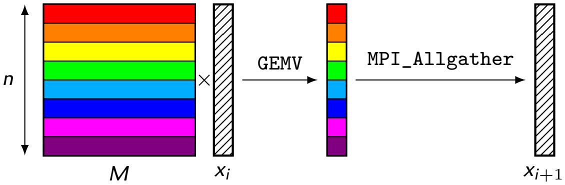
- n = size
- d = *density* (dn^2 non-zero coeffs)

	`GEMV`	`MPI_Allgather`
Sequential	$3dn^2/C$	0
Distributed	$3dn^2/(pC)$	$\geq n/D$

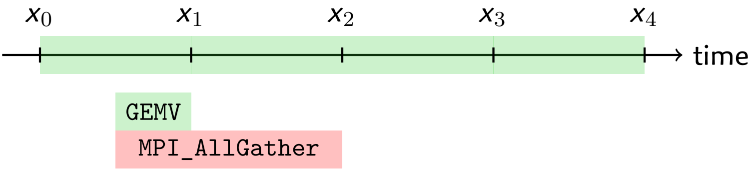
$$\text{Speedup} \leq \frac{dn^2}{C}/(\frac{dn^2}{pC} + \frac{n}{D}) \leq dn\frac{D}{C}$$

- D/C = "machine balance" = very important

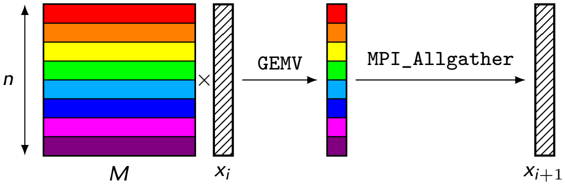
Communication Limits Acceleration



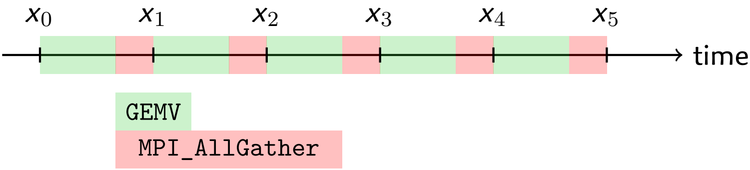
$p = 1$



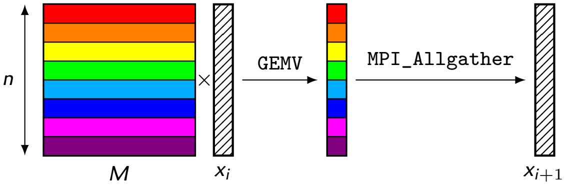
Communication Limits Acceleration



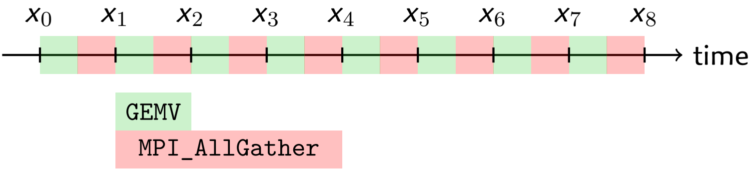
$$p = 2$$



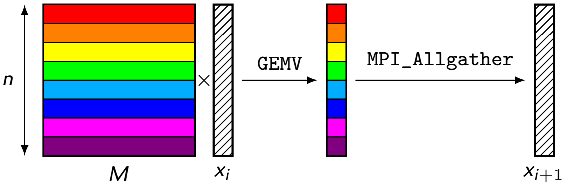
Communication Limits Acceleration



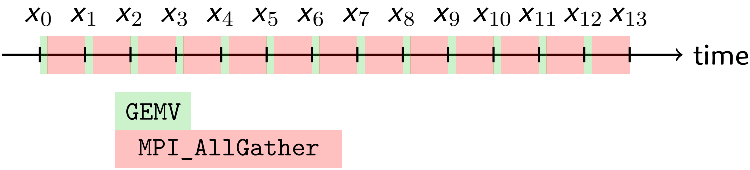
$p = 4$



Communication Limits Acceleration



$p = 20$



Distributed Matrix-Vector Product

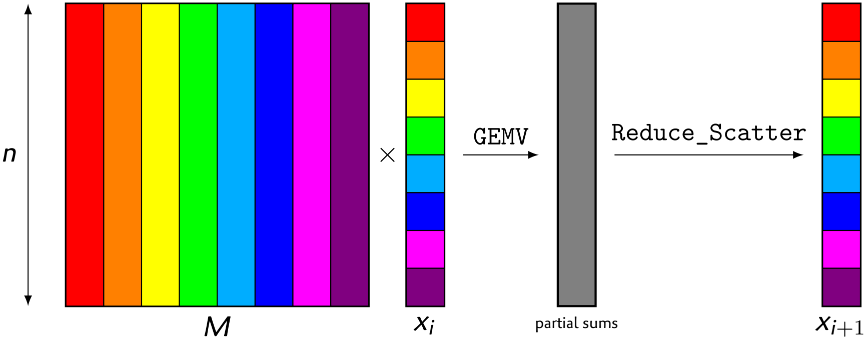
1D block distribution (per columns)

Data distribution

M : 1D block distribution (blocks of columns)

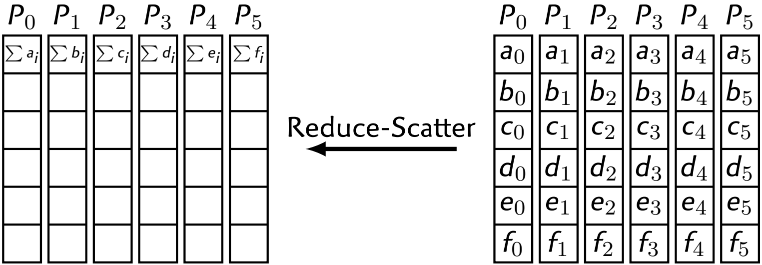
x : 1D block distribution

y : 1D block distribution



MPI : Reduce-Scatter

```c
int MPI_Reduce_scatter_block(void* sendbuf, void* recvbuf,
                             int recvcount, MPI_Datatype datatype,
                             MPI_Op op, MPI_Comm comm)
```



- Lower bound: $T \geq \lceil \log_2 p \rceil \alpha + (p-1)\frac{n}{p}\beta$
- Ring algorithm: $T = (p-1)(\alpha + \frac{n}{p}\beta)$

Distributed Matrix-Vector Product

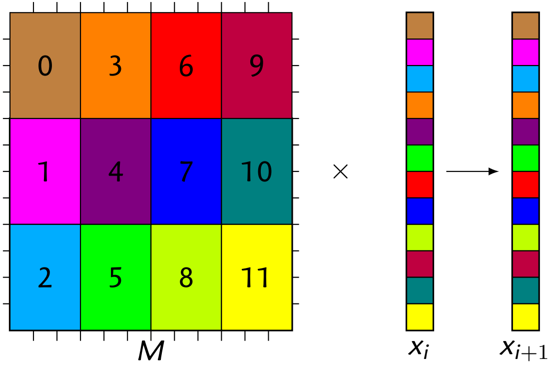
2D block distribution

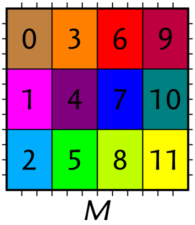
Data distribution

M : 2D block distribution ($v \times h$ blocks)

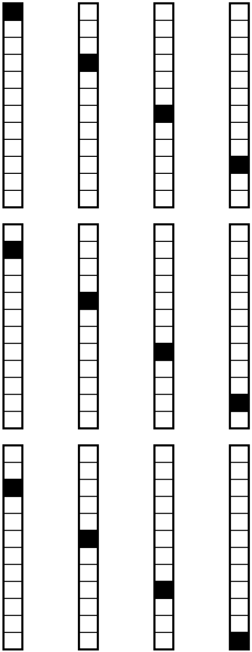
x : 1D block distribution

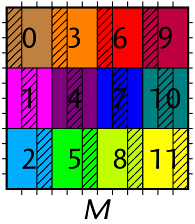
y : 1D block distribution

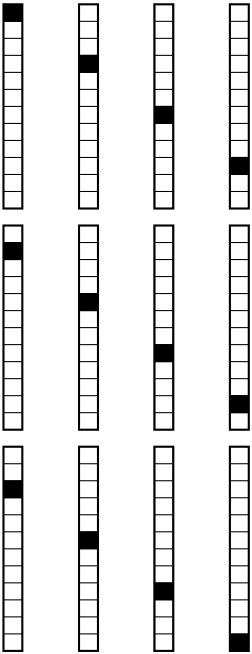




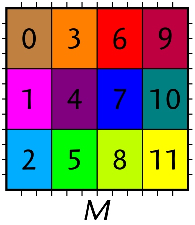
Algorithm





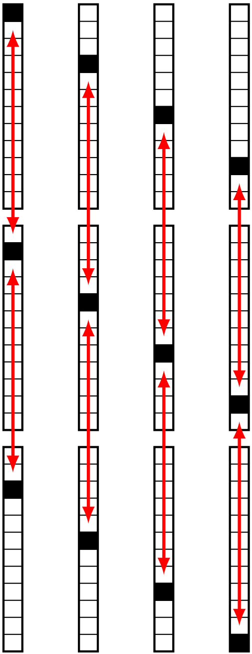


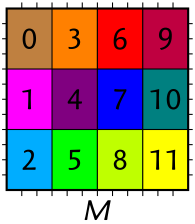
Algorithm

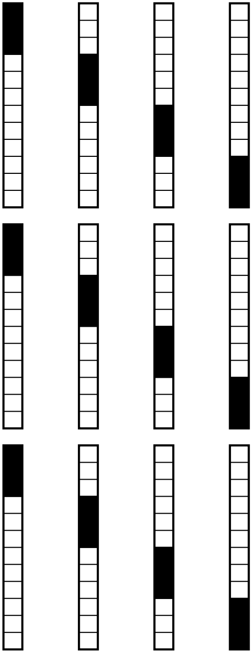


Algorithm

- `MPI_Allgather`

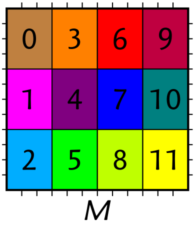






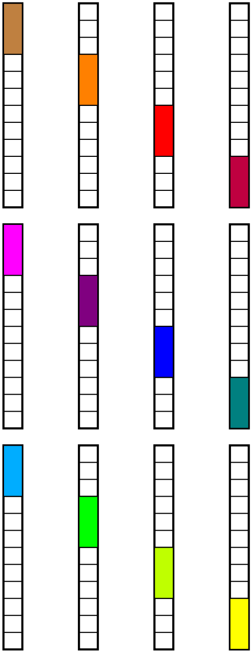
Algorithm

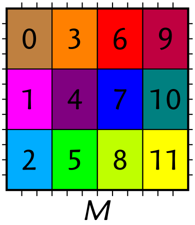
- `MPI_Allgather`



Algorithm

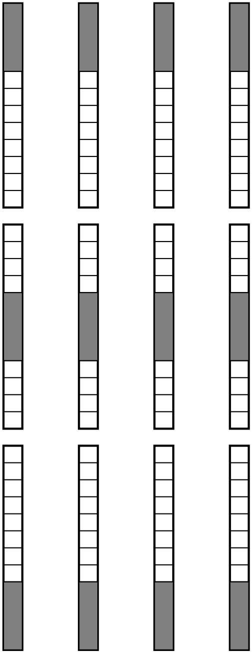
- `MPI_Allgather`
- `GEMV` (partial sums)

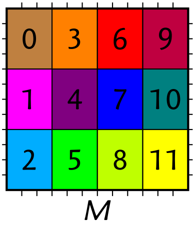




Algorithm

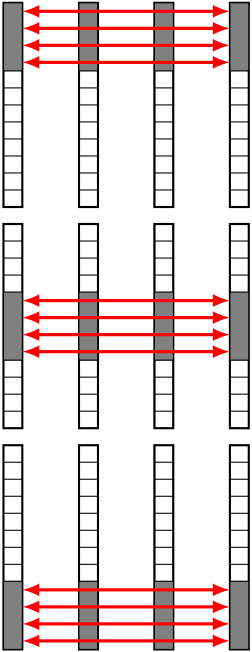
- `MPI_Allgather`
- `GEMV` (partial sums)

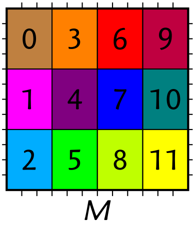




Algorithm

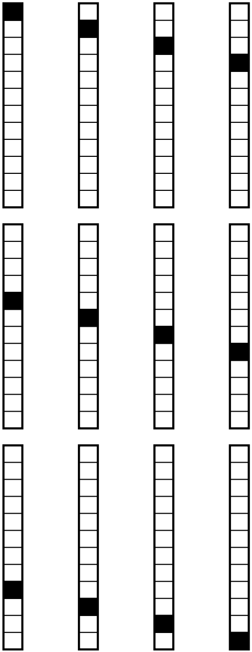
- `MPI_Allgather`
- `GEMV` (partial sums)
- `MPI_reduce_scatter`

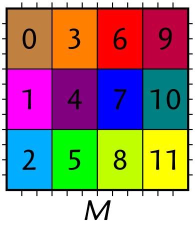


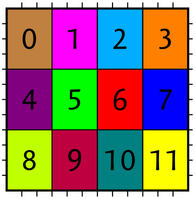


Algorithm

- `MPI_Allgather`
- `GEMV` (partial sums)
- `MPI_reduce_scatter`

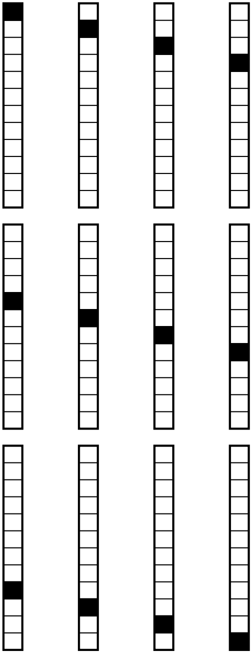


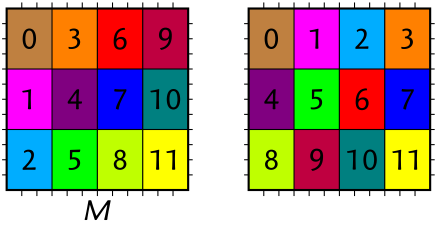




Algorithm

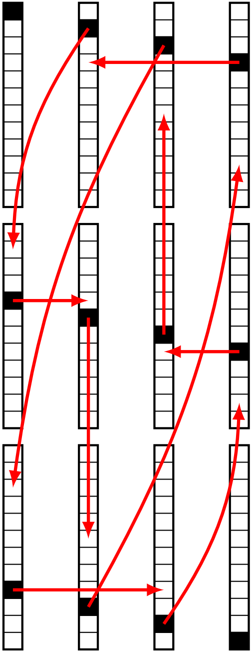
- `MPI_Allgather`
- `GEMV` (partial sums)
- `MPI_reduce_scatter`

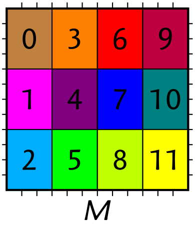




Algorithm

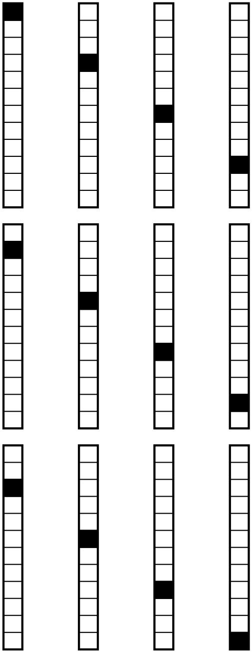
- `MPI_Allgather`
- `GEMV` (partial sums)
- `MPI_reduce_scatter`
- `MPI_sendrecv` (transpose)





Algorithm

- `MPI_Allgather`
- `GEMV` (partial sums)
- `MPI_reduce_scatter`
- `MPI_sendrecv` (transpose)



Analysis

Process grid of size $p = P \times Q$

	Sequential	Distributed
`GEMV`	$3dn^2/C$	$3dn^2/(pC)$
`MPI_Allgather`	0	$n/(PD)$
`MPI_reduce_scatter`	0	$n/(QD)$
`MPI_sendrecv`	0	$n/(pD)$

Assume $P = Q = \sqrt{p}$ and ignore latencies:

$$\text{Speedup} = \frac{dn^2}{C} / \left(\frac{dn^2}{pC} + \frac{n}{D\sqrt{p}} \right)$$

Progress

- Communication time **also decreases** when p grows
- Speed-up **no longer bounded** when p grows

Discrete Fourier Transform

Case Study

Discrete Fourier Transform (DFT)

- The DFT of an array X of n (complex) numbers is

$$Y[k] = \sum_{j=0}^{n-1} X[j] \left(\omega_n^k\right)^j, \qquad \text{with} \qquad \omega_n = e^{-\frac{2i\pi}{n}} \qquad (0 \leq k < n)$$

Rewriting the definition

When $n = n_1 \times n_2$, we set $j = j_1 n_2 + j_2$ and $k = k_2 n_1 + k_1$

$$Y[k_2 n_1 + k_1] = \sum_{j_2=0}^{n_2-1} \left[\left(\sum_{j_1=0}^{n_1-1} X[j_1 n_2 + j_2] {\omega_{n_1}}^{j_1 k_1} \right) {\omega_n}^{j_2 k_1} \right] {\omega_{n_2}}^{j_2 k_2}$$

DFT: Recursive Algorithm

$n = n_1 \times n_2$, we set $j = j_1 n_2 + j_2$ and $k = k_2 n_1 + k_1$

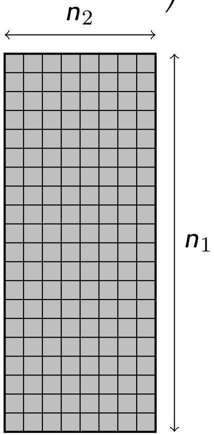
$$Y[k_2 n_1 + k_1] = \sum_{j_2=0}^{n_2-1} \left[\left(\sum_{j_1=0}^{n_1-1} X[j_1 n_2 + j_2] {\omega_{n_1}}^{j_1 k_1} \right) {\omega_n}^{j_2 k_1} \right] {\omega_{n_2}}^{j_2 k_2}$$

Algorithm to compute the DFT of size $n_1 \times n_2$

1. Do n_2 DFTs of size n_1 (internal sum)
2. Multiply by the *twiddle factors* ${\omega_n}^{j_2 k_1}$
3. Do n_1 DFTs of size n_2 (external sum)

DFT: Recursive Algorithm (cont'd)

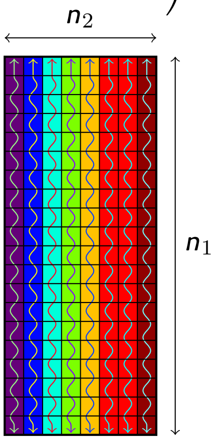
$$Y[k_2 n_1 + k_1] = \sum_{j_2=0}^{n_2-1} \left[\left(\sum_{j_1=0}^{n_1-1} X[j_1 n_2 + j_2] {\omega_{n_1}}^{j_1 k_1} \right) {\omega_n}^{j_2 k_1} \right] {\omega_{n_2}}^{j_2 k_2}$$



DFT: Recursive Algorithm (cont'd)

$$Y[k_2 n_1 + k_1] = \sum_{j_2=0}^{n_2-1} \left[\left(\sum_{j_1=0}^{n_1-1} X[j_1 n_2 + j_2] {\omega_{n_1}}^{j_1 k_1} \right) {\omega_n}^{j_2 k_1} \right] {\omega_{n_2}}^{j_2 k_2}$$

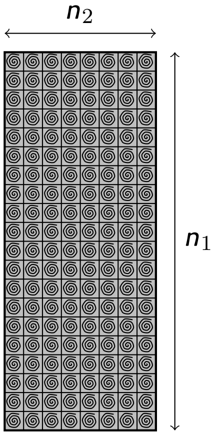
- $U[\star, j_2] \leftarrow \mathrm{DFT}(X[\star, j_2])$
  - $0 \leq j_2 < n_2$



DFT: Recursive Algorithm (cont'd)

$$Y[k_2 n_1 + k_1] = \sum_{j_2=0}^{n_2-1} \left(U[k_1 n_2 + j_2] {\omega_n}^{j_2 k_1} \right) {\omega_{n_2}}^{j_2 k_2}$$

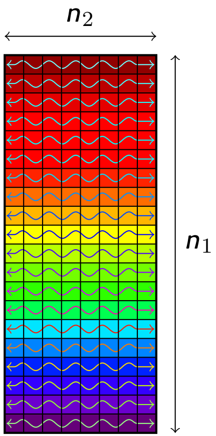
- $U[\star, j_2] \leftarrow \mathsf{DFT}(X[\star, j_2])$
  - $0 \leq j_2 < n_2$
- $V[k_1, j_2] \leftarrow U[k_1, j_2] \cdot {\omega_n}^{j_2 k_1}$
  - $0 \leq j_2 < n_2$
  - $0 \leq k_1 < n_1$



DFT: Recursive Algorithm (cont'd)

$$Y[k_2 n_1 + k_1] = \sum_{j_2=0}^{n_2-1} V[k_1 n_2 + j_2]{\omega_{n_2}}^{j_2 k_2}$$

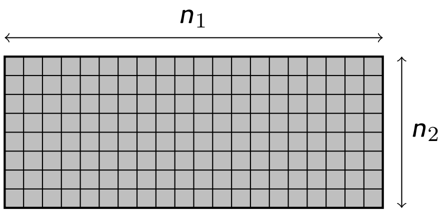
- $U[\star, j_2] \leftarrow \mathsf{DFT}(X[\star, j_2])$
  - $0 \leq j_2 < n_2$
- $V[k_1, j_2] \leftarrow U[k_1, j_2] \cdot {\omega_n}^{j_2 k_1}$
  - $0 \leq j_2 < n_2$
  - $0 \leq k_1 < n_1$
- $W[k_1, \star] \leftarrow \mathsf{DFT}(V[k_1, \star])$
  - $0 \leq k_1 < n_1$



DFT: Recursive Algorithm (cont'd)

$$Y[k_2 n_1 + k_1] = W[k_1 n_2 + k_2]$$

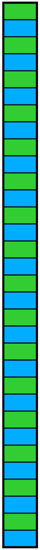
- $U[\star, j_2] \leftarrow \text{DFT}(X[\star, j_2])$
  - $0 \le j_2 < n_2$
- $V[k_1, j_2] \leftarrow U[k_1, j_2] \cdot {\omega_n}^{j_2 k_1}$
  - $0 \le j_2 < n_2$
  - $0 \le k_1 < n_1$
- $W[k_1, \star] \leftarrow \text{DFT}(V[k_1, \star])$
  - $0 \le k_1 < n_1$
- $Y \leftarrow \textsc{Transpose}(W)$



FFT: Classic (Sequential) Recursive Algorithm

- Common choice : $n_2 = 2$
  - *"Radix-2 Decimation in Time"*
- FFT of size 2 : $(x, y) \to (x + y, x - y)$

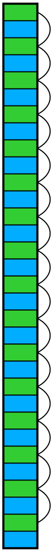
```
void FFT(const double * X, double *Y, int n, int s)
{
    if (n == 1) {
       Y[0] = X[0];
       return;
    }
    double omega_n = exp(-2*I*pi / n);
    double omega = 1;   // twiddle factor
    FFT(X,     Y       , n/2, 2*s);
    FFT(X + s, Y + n/2, n/2, 2*s);
    for (int i = 0; i < n/2; i++) {
        double p = Y[i];
        double q = Y[i + n/2] * omega;
        Y[i] = p + q;
        Y[i + n/2] = p - q;
        omega *= omega_n;
    }
}     // T(n) == 2 * T(n/2) + O(n) == O(n * log n)
```



FFT: Classic (Sequential) Recursive Algorithm

- Common choice : $n_2 = 2$
  - *"Radix-2 Decimation in Time"*
- FFT of size 2 : $(x, y) \to (x + y, x - y)$

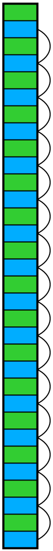
```
void FFT(const double * X, double *Y, int n, int s)
{
    if (n == 1) {
        Y[0] = X[0];
        return;
    }
    double omega_n = exp(-2*I*pi / n);
    double omega = 1;    // twiddle factor
    FFT(X,     Y       , n/2, 2*s);
    FFT(X + s, Y + n/2, n/2, 2*s);
    for (int i = 0; i < n/2; i++) {
        double p = Y[i];
        double q = Y[i + n/2] * omega;
        Y[i] = p + q;
        Y[i + n/2] = p - q;
        omega *= omega_n;
    }
}    // T(n) == 2 * T(n/2) + O(n) == O(n * log n)
```



FFT: Classic (Sequential) Recursive Algorithm

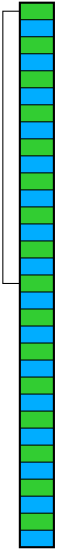
- Common choice : $n_2 = 2$
  - *"Radix-2 Decimation in Time"*
- FFT of size 2 : $(x, y) \to (x + y, x - y)$

```
void FFT(const double * X, double *Y, int n, int s)
{
    if (n == 1) {
       Y[0] = X[0];
       return;
    }
    double omega_n = exp(-2*I*pi / n);
    double omega = 1;   // twiddle factor
    FFT(X,     Y       , n/2, 2*s);
    FFT(X + s, Y + n/2, n/2, 2*s);
    for (int i = 0; i < n/2; i++) {
        double p = Y[i];
        double q = Y[i + n/2] * omega;
        Y[i] = p + q;
        Y[i + n/2] = p - q;
        omega *= omega_n;
    }
}     // T(n) == 2 * T(n/2) + O(n) == O(n * log n)
```



FFT: Classic (Sequential) Recursive Algorithm

- Common choice : $n_2 = 2$
  - *"Radix-2 Decimation in Time"*
- FFT of size 2 : $(x, y) \rightarrow (x + y, x - y)$

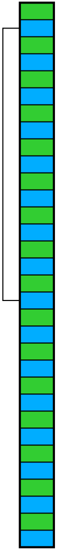


```
void FFT(const double * X, double *Y, int n, int s)
{
    if (n == 1) {
       Y[0] = X[0];
       return;
    }
    double omega_n = exp(-2*I*pi / n);
    double omega = 1;    // twiddle factor
    FFT(X,      Y      , n/2, 2*s);
    FFT(X + s, Y + n/2, n/2, 2*s);
    for (int i = 0; i < n/2; i++) {
        double p = Y[i];
        double q = Y[i + n/2] * omega;
        Y[i] = p + q;
        Y[i + n/2] = p - q;
        omega *= omega_n;
    }
}     // T(n) == 2 * T(n/2) + O(n) == O(n * log n)
```

FFT: Classic (Sequential) Recursive Algorithm

- Common choice : $n_2 = 2$
  - *"Radix-2 Decimation in Time"*
- FFT of size 2 : $(x,y) \to (x+y, x-y)$

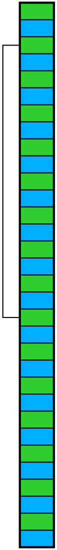
```
void FFT(const double * X, double *Y, int n, int s)
{
    if (n == 1) {
       Y[0] = X[0];
       return;
    }
    double omega_n = exp(-2*I*pi / n);
    double omega = 1;   // twiddle factor
    FFT(X,      Y       , n/2, 2*s);
    FFT(X + s, Y + n/2, n/2, 2*s);
    for (int i = 0; i < n/2; i++) {
        double p = Y[i];
        double q = Y[i + n/2] * omega;
        Y[i] = p + q;
        Y[i + n/2] = p - q;
        omega *= omega_n;
    }
}     // T(n) == 2 * T(n/2) + O(n) == O(n * log n)
```



FFT: Classic (Sequential) Recursive Algorithm

- Common choice : $n_2 = 2$
  - *"Radix-2 Decimation in Time"*
- FFT of size 2 : $(x, y) \to (x + y, x - y)$

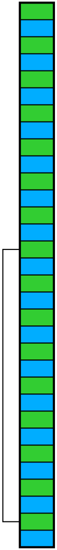
```c
void FFT(const double * X, double *Y, int n, int s)
{
    if (n == 1) {
       Y[0] = X[0];
       return;
    }
    double omega_n = exp(-2*I*pi / n);
    double omega = 1;   // twiddle factor
    FFT(X,      Y       , n/2, 2*s);
    FFT(X + s, Y + n/2, n/2, 2*s);
    for (int i = 0; i < n/2; i++) {
        double p = Y[i];
        double q = Y[i + n/2] * omega;
        Y[i] = p + q;
        Y[i + n/2] = p - q;
        omega *= omega_n;
    }
}     // T(n) == 2 * T(n/2) + O(n) == O(n * log n)
```



FFT: Classic (Sequential) Recursive Algorithm

- Common choice : $n_2 = 2$
  - *"Radix-2 Decimation in Time"*
- FFT of size 2 : $(x, y) \rightarrow (x + y, x - y)$

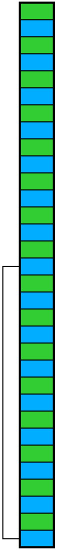
```c
void FFT(const double * X, double *Y, int n, int s)
{
    if (n == 1) {
       Y[0] = X[0];
       return;
    }
    double omega_n = exp(-2*I*pi / n);
    double omega = 1;    // twiddle factor
    FFT(X,      Y       , n/2, 2*s);
    FFT(X + s, Y + n/2, n/2, 2*s);
    for (int i = 0; i < n/2; i++) {
        double p = Y[i];
        double q = Y[i + n/2] * omega;
        Y[i] = p + q;
        Y[i + n/2] = p - q;
        omega *= omega_n;
    }
}     // T(n) == 2 * T(n/2) + O(n) == O(n * log n)
```



FFT: Classic (Sequential) Recursive Algorithm

- Common choice : $n_2 = 2$
  - *"Radix-2 Decimation in Time"*
- FFT of size 2 : $(x, y) \to (x + y, x - y)$

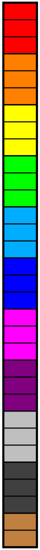
```
void FFT(const double * X, double *Y, int n, int s)
{
    if (n == 1) {
       Y[0] = X[0];
       return;
    }
    double omega_n = exp(-2*I*pi / n);
    double omega = 1;   // twiddle factor
    FFT(X,     Y      , n/2, 2*s);
    FFT(X + s, Y + n/2, n/2, 2*s);
    for (int i = 0; i < n/2; i++) {
        double p = Y[i];
        double q = Y[i + n/2] * omega;
        Y[i] = p + q;
        Y[i + n/2] = p - q;
        omega *= omega_n;
    }
}     // T(n) == 2 * T(n/2) + O(n) == O(n * log n)
```



FFT: Classic (Sequential) Recursive Algorithm

- Common choice : $n_2 = 2$
  - *"Radix-2 Decimation in Time"*
- FFT of size 2 : $(x, y) \to (x + y, x - y)$

```
void FFT(const double * X, double *Y, int n, int s)
{
    if (n == 1) {
       Y[0] = X[0];
       return;
    }
    double omega_n = exp(-2*I*pi / n);
    double omega = 1;    // twiddle factor
    FFT(X,     Y      , n/2, 2*s);
    FFT(X + s, Y + n/2, n/2, 2*s);
    for (int i = 0; i < n/2; i++) {
        double p = Y[i];
        double q = Y[i + n/2] * omega;
        Y[i] = p + q;
        Y[i + n/2] = p - q;
        omega *= omega_n;
    }
}     // T(n) == 2 * T(n/2) + O(n) == O(n * log n)
```



???

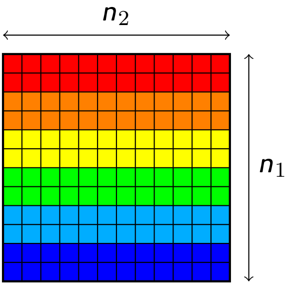
The "Six-Step" DFT Algorithm

- $N = p^2k^2$ data items
- Assumptions: nodes have $\sqrt{N}$ memory, block distribution

Main strategy: use $n_1 = n_2 = \sqrt{N}$

- Recursive DFTs confined **inside nodes**

⇝ No communication needed in recursive calls



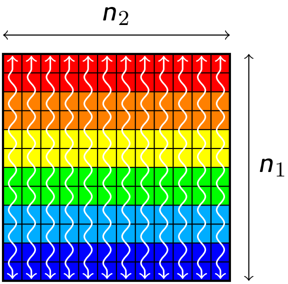
The "Six-Step" DFT Algorithm

- $N = p^2k^2$ data items
- Assumptions: nodes have $\sqrt{N}$ memory, block distribution

Main strategy: use $n_1 = n_2 = \sqrt{N}$

- Recursive DFTs confined **inside nodes**

$\rightsquigarrow$ No communication needed in recursive calls



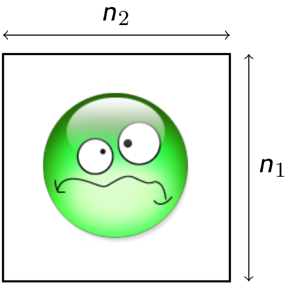
The "Six-Step" DFT Algorithm

- $N = p^2k^2$ data items
- Assumptions: nodes have $\sqrt{N}$ memory, block distribution

Main strategy: use $n_1 = n_2 = \sqrt{N}$

- Recursive DFTs confined **inside nodes**

⇝ No communication needed in recursive calls



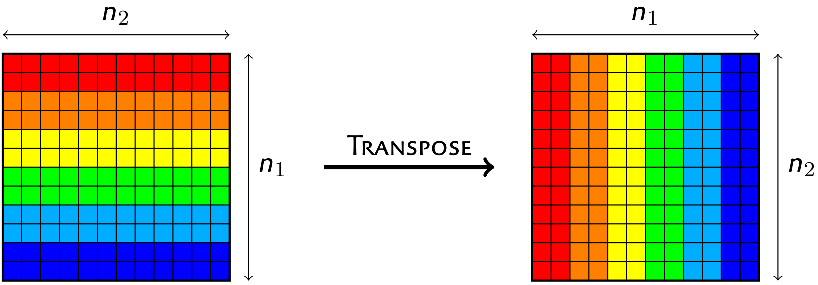
The “Six-Step” DFT Algorithm

- $N = p^2k^2$ data items
- Assumptions: nodes have $\sqrt{N}$ memory, block distribution

Main strategy: use $n_1 = n_2 = \sqrt{N}$

- Recursive DFTs confined **inside nodes**

⇝ No communication needed in recursive calls



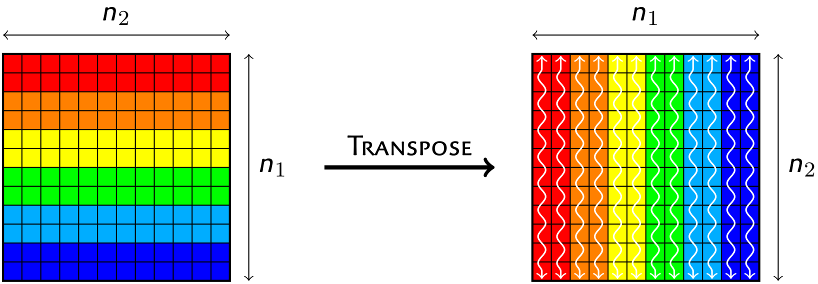
The "Six-Step" DFT Algorithm

- $N = p^2k^2$ data items
- Assumptions: nodes have $\sqrt{N}$ memory, block distribution

Main strategy: use $n_1 = n_2 = \sqrt{N}$

- Recursive DFTs confined **inside nodes**

⇝ No communication needed in recursive calls



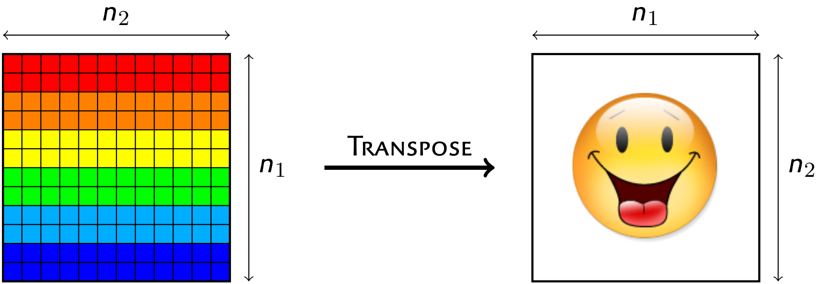
The "Six-Step" DFT Algorithm

- $N = p^2 k^2$ data items
- Assumptions: nodes have $\sqrt{N}$ memory, block distribution

Main strategy: use $n_1 = n_2 = \sqrt{N}$

- Recursive DFTs confined **inside nodes**

⇝ No communication needed in recursive calls



The "Six-Step" DFT Algorithm

- $N = p^2 k^2$ data items
- Assumptions: nodes have $\sqrt{N}$ memory, block distribution

Main strategy: use $n_1 = n_2 = \sqrt{N}$

- Recursive DFTs confined **inside nodes**

$\rightsquigarrow$ No communication needed in recursive calls

Plan

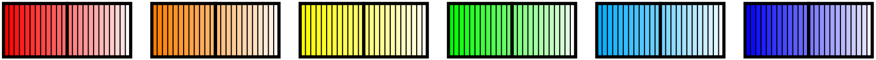
1. Transpose
2. First pass of recursive DFTs ("columns") [local]
3. Multiplication by twiddle factors [local]
4. Transpose
5. Second pass of recursive DFTs ("rows") [local]
6. Transpose

Transposing a Square Block-Distributed Matrix

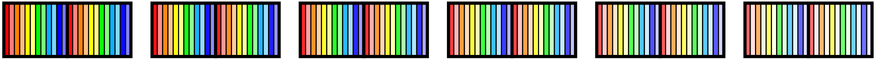
Pure Data Movement Problem

- Dimension $pk \times pk$
- Block-distributed: k (consecutive) rows / process

Initial block distribution:



Goal:

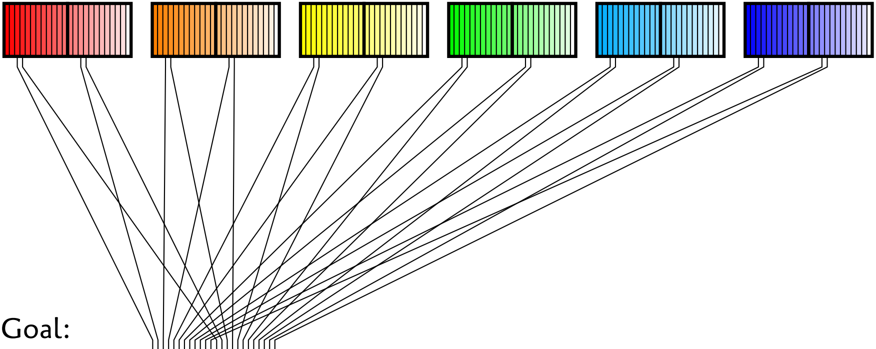


Transposing a Square Block-Distributed Matrix

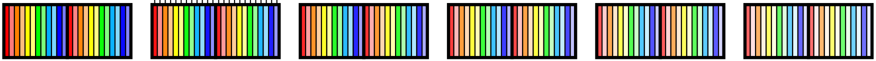
Pure Data Movement Problem

- Dimension $pk \times pk$
- Block-distributed: k (consecutive) rows / process

Initial block distribution:



Goal:

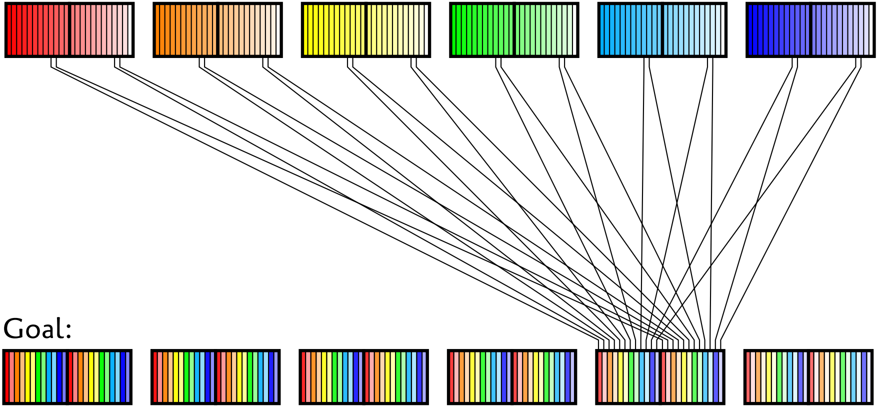


Transposing a Square Block-Distributed Matrix

Pure Data Movement Problem

- Dimension $pk \times pk$
- Block-distributed: k (consecutive) rows / process

Initial block distribution:



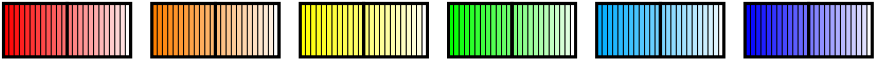
Goal:

Transposing a Square Block-Distributed Matrix

Pure Data Movement Problem

- Dimension $pk \times pk$
- Block-distributed: k (consecutive) rows / process

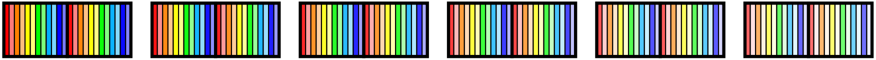
Initial block distribution:



First DFT pass ("columns")

- j-th column = `A[j::p*k]`
- Each column needs k items from each process
- Each process needs k^2 items from each other process

Goal:



How To Do This?

Not completely trivial problem

- (solution **not found** on google / StackOverflow / ...)

Somewhat Simpler: **Scatter** to Cyclic Distribution

- Rank 0 has an array of size n (multiple of p)
- Goal: process i gets `A[i::p]`
  - *i.e.* $B[k] = A[i + kp]$ for $0 \leq k < n/p$

Roadmap

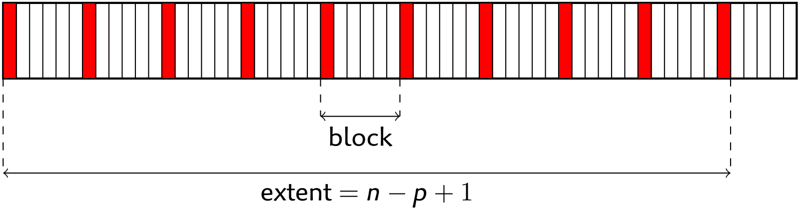
1. Create a custom MPI Type (N/p items with stride of p)
   - `MPI_Type_vector`
2. "Cheat" by changing its *extent*
   - `MPI_Type_create_resized`
3. Then `MPI_Scatter` with the custom type

Scatter to Cyclic Distribution

```
MPI_Datatype strided;

MPI_Type_vector(n/p, 1, p, MPI_DOUBLE, &strided);
```

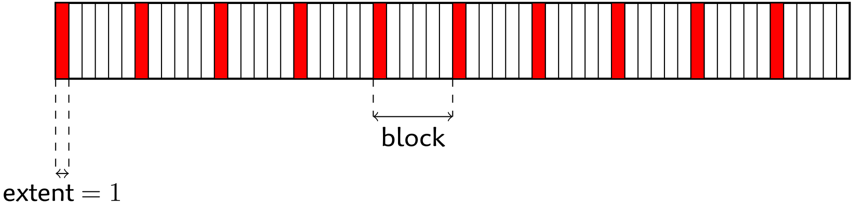
- n/p blocks
- Each block = 1 $\times$ `double`
- p `double` between the start of two blocks
- $(n-p+1)\times$ `double` between two `strided` types



Scatter to Cyclic Distribution

```
MPI_Datatype strided, cyclic;
MPI_Aint extent, lb;
MPI_Type_vector(n/p, 1, p, MPI_DOUBLE, &strided);
MPI_Type_get_extent(MPI_DOUBLE, &lb, &extent);
MPI_Type_create_resized(strided, 0, extent, &cyclic);
```

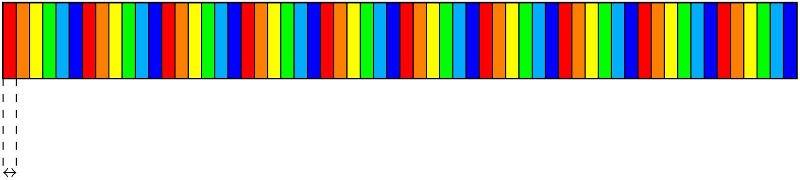
- n/p blocks
- Each block = 1 $\times$ `double`
- p `double` between the start of two blocks
- Only $1\times$ `double` between two `cyclic` types



Scatter to Cyclic Distribution

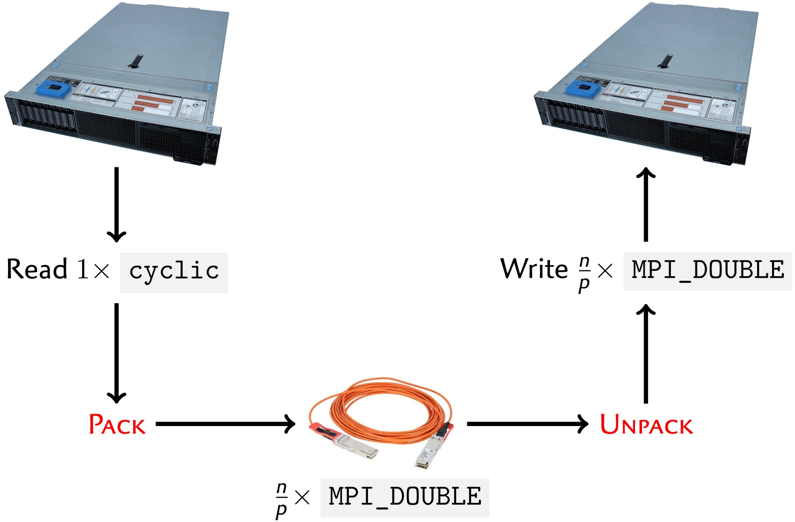
```
MPI_Datatype strided, cyclic;
MPI_Aint extent, lb;
MPI_Type_vector(n/p, 1, p, MPI_DOUBLE, &strided);
MPI_Type_get_extent(MPI_DOUBLE, &lb, &extent);
MPI_Type_create_resized(strided, 0, extent, &cyclic);
MPI_Type_commit(&cyclic);
MPI_Scatter(X,1,cyclic, Y,n/p,MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

- n/p blocks
- Each block = 1 × `double`
- p `double` between the start of two blocks
- Only $1\times$ `double` between two `cyclic` types

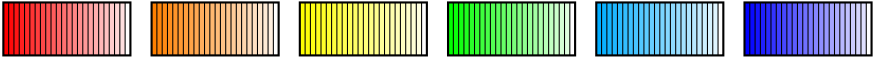


extent $= 1$

Packing and Unpacking



Next: Redistribute Block $\longrightarrow$ Cyclic



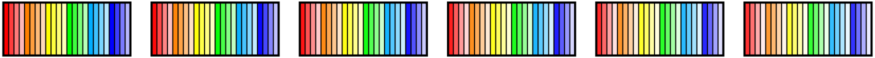
Communication pattern

- $n = p^2\ell$
- i-th rank gets `A[i::p]`
- Each process sends ℓ items to each other process

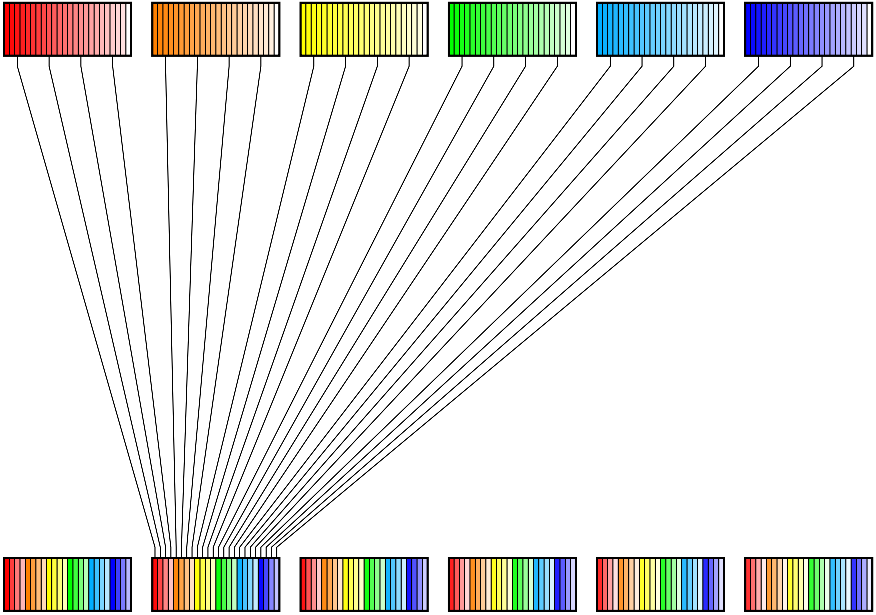
  $\Rightarrow$ `MPI_Alltoall`

Data location

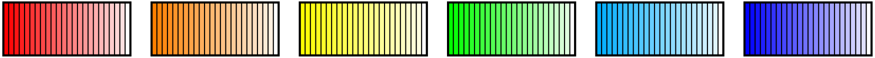
- ℓ **non-contiguous** items in send buffer
- ℓ **contiguous** items in reception buffer
  - Just like before (scatter to cyclic)



Next: Redistribute Block $\longrightarrow$ Cyclic

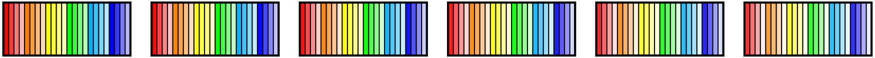


Next: Redistribute Block ⟶ Cyclic

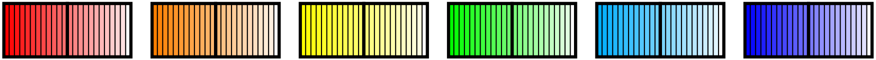


```
MPI_Datatype strided, cyclic;
MPI_Aint extent, lb;
MPI_Type_vector(l, 1, p, MPI_DOUBLE, &strided);
MPI_Type_get_extent(MPI_DOUBLE, &lb, &extent);
MPI_Type_create_resized(strided, 0, extent, &cyclic);
MPI_Type_commit(&cyclic);

MPI_Alltoall(Y,1,cyclic, Z,l,MPI_DOUBLE, MPI_COMM_WORLD);
```



Finally: Transpose Square Block-Distributed Matrix

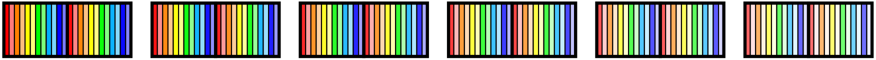


Communication pattern

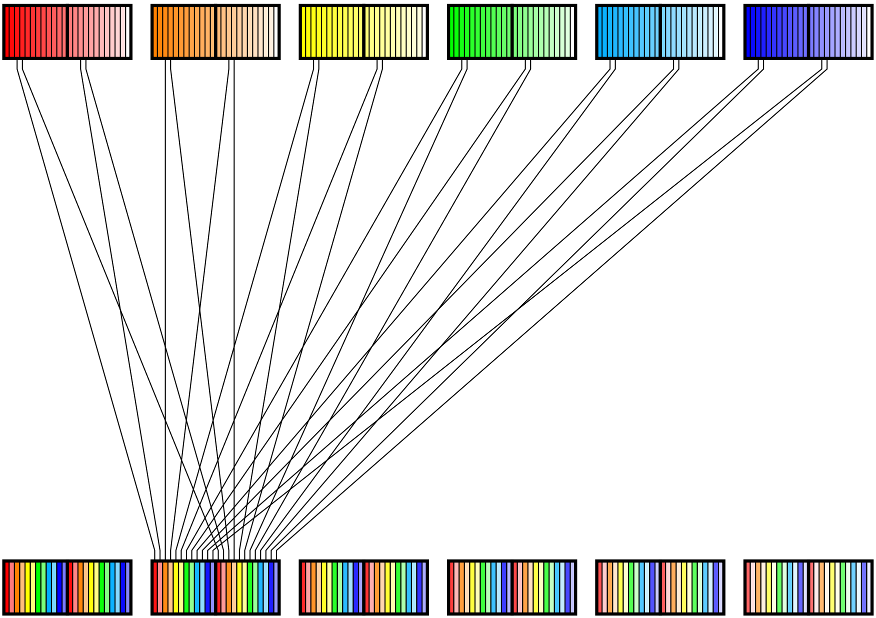
- $n = p^2 k^2$
- j-th column: `A[j::k*p]`, length kp, k columns / process
- Each process sends k^2 items to each other process
  $\Rightarrow$ `MPI_Alltoall`

Data location

- k **non-contiguous** **blocks** of k items in send buffer
- k^2 **non-contiguous** items in reception buffer



Finally: Transpose Square Block-Distributed Matrix

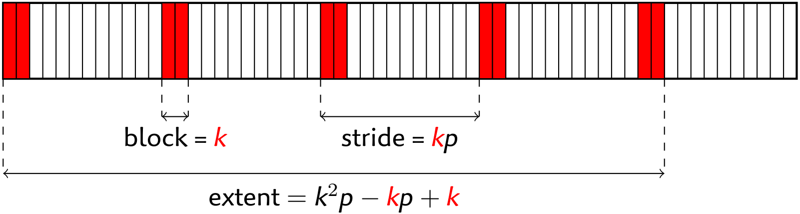


More MPI Data Types

```
MPI_Aint extent, lb;
MPI_Type_get_extent(MPI_DOUBLE, &lb, &extent);

MPI_Datatype strided_in;
MPI_Type_vector(k, k, k*p, MPI_DOUBLE, &strided_in);
```

- k blocks
- Each block = k $\times$ `double`
- kp `double` between the start of two blocks
- $(k^2p - kp + k) \times$ `double` between two `strided_in` types

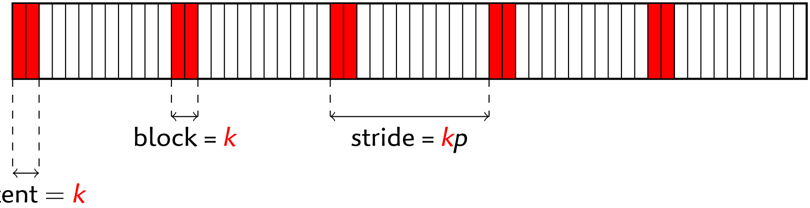


More MPI Data Types

```
MPI_Aint extent, lb;
MPI_Type_get_extent(MPI_DOUBLE, &lb, &extent);

MPI_Datatype strided_in, cyclic_in;
MPI_Type_vector(k, k, k*p, MPI_DOUBLE, &strided_in);
MPI_Type_create_resized(strided_in, 0, k*extent, &cyclic_in);
MPI_Type_commit(&cyclic_in);
```

- k blocks
- Each block = k $\times$ `double`
- kp `double` between the start of two blocks
- Only $k\times$ `double` between two `cyclic_in` types

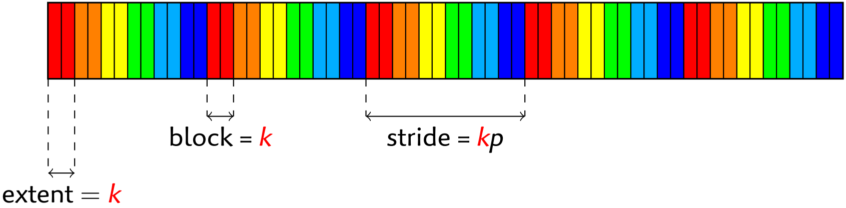


More MPI Data Types

```
MPI_Aint extent, lb;
MPI_Type_get_extent(MPI_DOUBLE, &lb, &extent);

MPI_Datatype strided_in, cyclic_in;
MPI_Type_vector(k, k, k*p, MPI_DOUBLE, &strided_in);
MPI_Type_create_resized(strided_in, 0, k*extent, &cyclic_in);
MPI_Type_commit(&cyclic_in);
```

- k blocks
- Each block = k $\times$ `double`
- kp `double` between the start of two blocks
- Only $k \times$ `double` between two `cyclic_in` types

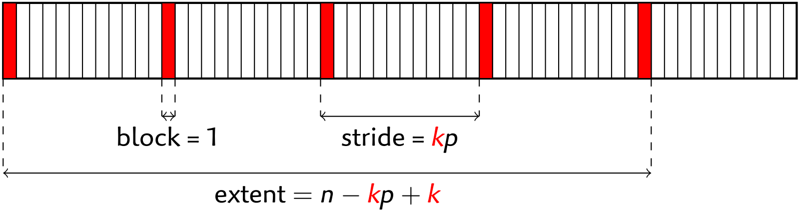


Even More MPI Data Types

```
MPI_Aint extent, lb;
MPI_Type_get_extent(MPI_DOUBLE, &lb, &extent);

MPI_Datatype strided_out;
MPI_Type_vector(k, 1, k*p, MPI_DOUBLE, &strided_out);
```

- k blocks
- Each block = 1 $\times$ `double`
- kp `double` between the start of two blocks
- $(k^2p - kp + 1)\times$ `double` between two `strided_out` types

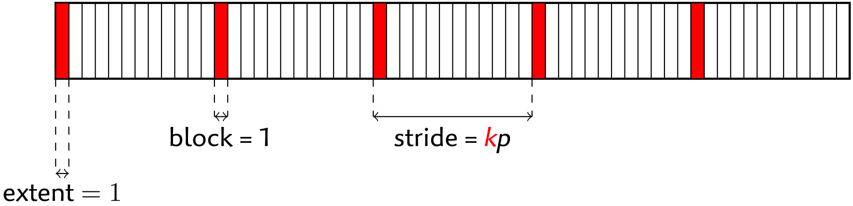


Even More MPI Data Types

```
MPI_Aint extent, lb;
MPI_Type_get_extent(MPI_DOUBLE, &lb, &extent);

MPI_Datatype strided_out, cyclic_out;
MPI_Type_vector(k, 1, k*p, MPI_DOUBLE, &strided_out);
MPI_Type_create_resized(strided_out, 0, extent, &cyclic_out);
MPI_Type_commit(&cyclic_out);
```

- k blocks
- Each block = 1 $\times$ `double`
- kp `double` between the start of two blocks
- Only $1\times$ `double` between two `cyclic_out` types

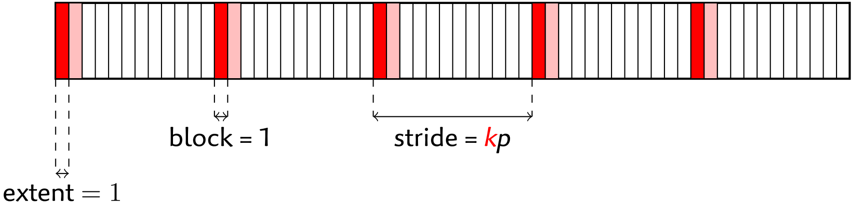


Even More MPI Data Types

```
MPI_Aint extent, lb;
MPI_Type_get_extent(MPI_DOUBLE, &lb, &extent);

MPI_Datatype strided_out, cyclic_out;
MPI_Type_vector(k, 1, k*p, MPI_DOUBLE, &strided_out);
MPI_Type_create_resized(strided_out, 0, extent, &cyclic_out);
MPI_Type_commit(&cyclic_out);
```

- k blocks
- Each block = 1 $\times$ `double`
- kp `double` between the start of two blocks
- Only $1\times$ `double` between two `cyclic_out` types

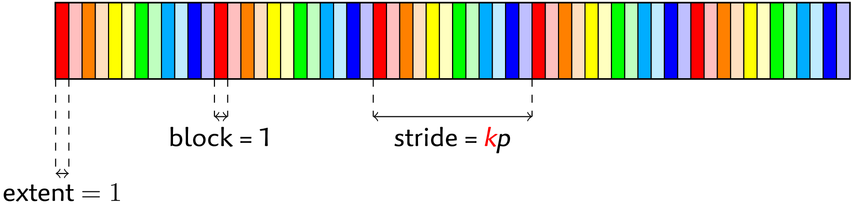


Even More MPI Data Types

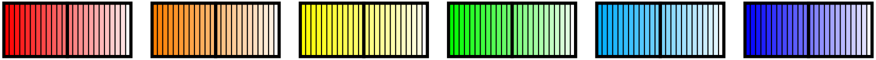
```
MPI_Aint extent, lb;
MPI_Type_get_extent(MPI_DOUBLE, &lb, &extent);

MPI_Datatype strided_out, cyclic_out;
MPI_Type_vector(k, 1, k*p, MPI_DOUBLE, &strided_out);
MPI_Type_create_resized(strided_out, 0, extent, &cyclic_out);
MPI_Type_commit(&cyclic_out);
```

- k blocks
- Each block = 1 × `double`
- kp `double` between the start of two blocks
- Only $1\times$ `double` between two `cyclic_out` types



Finally: Transpose Square Block-Distributed Matrix

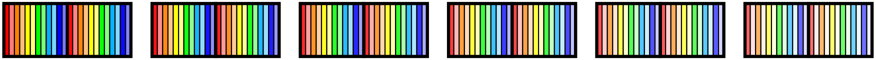


```
MPI_Aint extent, lb;
MPI_Type_get_extent(MPI_DOUBLE, &lb, &extent);

MPI_Datatype cyclic_in, strided_in;
MPI_Type_vector(k, k, k*p, MPI_DOUBLE, &strided_in);
MPI_Type_create_resized(strided_in, 0, k*extent, &cyclic_in);
MPI_Type_commit(&cyclic_in);

MPI_Datatype cyclic_out, strided_out;
MPI_Type_vector(k, 1, k*p, MPI_DOUBLE, &strided_out);
MPI_Type_create_resized(strided_out, 0, extent, &cyclic_out);
MPI_Type_commit(&cyclic_out);

MPI_Alltoall(Y,1,cyclic_in, Z,k,cyclic_out, MPI_COMM_WORLD);
```



Packing and Unpacking Redux

Read $1\times$ `cyclic_in`

Write $k\times$ `cyclic_out`

PACK

UNPACK

$k^2\times$ `MPI_DOUBLE`