Towards Usable and Lean Parallel Linear Algebra Libraries

Robert van de Geijn

Department of Computer Sciences
and
Texas Institute for Computational and Applied Mathematics
The University of Texas at Austin
Austin, Texas 78712

April 22, 1996

CONTENTS

Meet the Team

Core Team

Almadena Chtchelkanova, Carter Edwards
John Gunnels, Ken Klimkowski
Greg Morrow, James Overfelt
Robert van de Geijn

Contributing Members

Philip Alpatov, Greg Baker
Joe Eaton, Murthy Guddati
Sam Guyer, Calvin Lin
Peter Nagel, Michelle Pal

Computer Science

Almadena Chtchelkanova
John Gunnels
Sam Guyer
Calvin Lin
Robert van de Geijn

Computational and Applied Mathematics

Joe Eaton
Carter Edwards
Murthy Guddati
Ken Klimkowski
James Overfelt
Michelle Pal
Robert van de Geijn

Physics

Philip Alpatov
Almadena Chtchelkanova
Greg Morrow

Aerospace Engineering

Greg Baker
Peter Nagel

Parallel Matrix Distributions: have we been doing it all wrong?

Traditional Matrix Distributions

- **One Dimensional**
  - ⋆ **Blocked**
  - ⋆ **Wrapped**
    Justification: Better load balance.
  - ⋆ **Block-Wrapped**
    Justification: Better granularity
  - ⋆ **Irregular Block-Wrapped?**
    Justification: Some applications need it

Generally suffices for “real” sparse applications solved iteratively.

Traditional Matrix Distributions (continued

- **Two Dimensional**
  - ⋆ **Blocked**
    Justification: Better scalability of the algorithm
  - ⋆ **Wrapped**
    Justification: Better load balance
  - ⋆ **Block-Wrapped**
    Justification: Better granularity
  - ⋆ **Irregular Block-Wrapped?**
    Justification: Some applications need it

Suffices for dense applications and sparse applications solved directly (and iteratively).

Application vs. Library

- What matrix distributions?
  Possible answers:
  - ⋆ Whatever an application needs.
    Issues: Filling the system of linear systems; Interface.
  - ⋆ Whatever makes the library scalable and efficient.
- Traditional view of cartesian matrix distributions:
  these solutions conflict.
  - ⋆ Edwards, Geng, Patra, and van de Geijn,
    "Parallel Matrix Distributions: have we been doing it all wrong?"

Example 1: Dense Linear Algebra

- Application: Boundary Element Methods
  - ⋆ arise in acoustics, electromagnetics, analysis of composites...
  - ⋆ can be solved by assembling large dense linear systems.
- Solution method: parallel dense linear solver (LU factorization).
- Library requirements: for scalability, a 2D matrix distribution is required.
- Problem: Building the matrix:
  - ⋆ Blocks on the diagonal are more expensive to generate. These are distributed to $O(\sqrt{p})$ processors. Hence load imbalance occurs.

Example 2: Sparse Iterative Methods

- Application: Computational Chemistry, NAS Parallel CG Benchmark:
  - ⋆ Unstructured sparse matrix.
- Solution method: parallel CG iteration.
- Library requirements: for scalability, a 2D matrix distribution is required.
- Problem: Standard 2D distributions lead to
  - ⋆ Diagonal distributed to $O(\sqrt{p})$ processors.
    - ◇ Load imbalance during matrix-vector product.
    - ◇ Problems during preconditioning:
      - ▷ Load imbalance.
      - ▷ Vectors and diagonal blocks are not matched on processors
  - ⋆ Extra communication requirements.

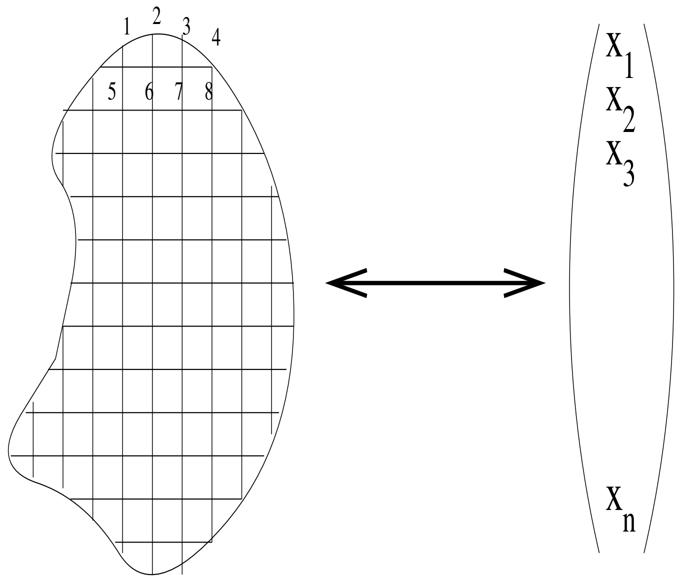
Example 3: Sparse Direct Methods

- Application: Problems in CFD, FEM,...
  - ⋆ Highly structured sparse matrix
- Solution method: Clever ordering followed by (multi)frontal method.
- Library requirements: for scalability, a 2D matrix distribution is required.
- Problem: Standard 2D distributions lead to
  - ⋆ Load imbalance during initial stages.
  - ⋆ Nested disection: Often don't know what to do with interface matrices.

Physically Based Matrix Distributions (PBMD)

Applications, vectors, and matrices

- To libraries, the matrix is the center of the universe
- To the application, vectors are more natural.



Vectors first, matrices second

- Typically, the elements of the vectors in linear systems have (same) physical meaning (locality)
  - ⋆ E.g., fixed point iteration:

$$x_{k+1} = A x_k$$

- Subdividing the vectors is equivalent to subdividing problem into domains.
- Claim: it is more natural to start by distributing vectors.
- New approach: physical problem (vectors) at the center of the universe.

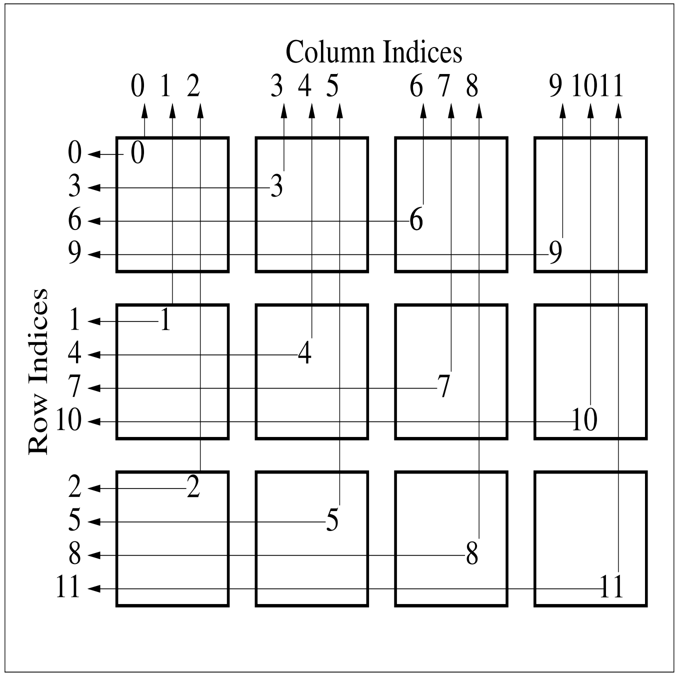
Partitioning Vectors and Matrices

- Assumption: p processors
- $Ax = y$.

$$P_x x = \left(\begin{array}{c} x_0 \\ \hline x_1 \\ \hline \vdots \\ \hline x_{p-1} \end{array} \right) \quad \text{and} \quad P_y y = \left(\begin{array}{c} y_0 \\ \hline y_1 \\ \hline \vdots \\ \hline y_{p-1} \end{array} \right)$$

$$P_y A P_x^T = \left(\begin{array}{c|c|c|c} A_{0,0} & A_{0,1} & \cdots & A_{0,p-1} \\ \hline A_{1,0} & A_{1,1} & \cdots & A_{1,p-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline A_{p-1,0} & A_{p-1,1} & \cdots & A_{p-1,p-1} \end{array} \right)$$

Inducing Matrix Distribution from a Inducing Vector Distribution



- Determine indices of rows of matrix from subvectors assigned to row of nodes
- Determine indices of columns of matrix from subvectors assigned to row of nodes

Resulting matrix distribution

0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,7	0,8	0,9	0,10	0,11
3,0	3,1	3,2	**3,3**	3,4	3,5	3,6	3,7	3,8	3,9	3,10	3,11
6,0	6,1	6,2	6,3	6,4	6,5	**6,6**	6,7	6,8	6,9	6,10	6,11
9,0	9,1	9,2	9,3	9,4	9,5	9,6	9,7	9,8	**9,9**	9,10	9,11
1,0	**1,1**	1,2	1,3	1,4	1,5	1.6	1,7	1,8	1,9	1,10	1,11
4,0	4,1	4,2	4,3	**4,4**	4,5	4,6	4,7	4,8	4,9	4,10	4,11
7,0	7,1	7,2	7,3	7,4	7,5	7,6	**7,7**	7,8	7,9	7,10	7,11
10,0	10,1	10,2	11,3	10,4	10,5	10,6	10,7	10,8	10,3	**10,10**	10,11
2,0	2,1	**2,2**	2,3	2,4	2,5	2,6	2,7	2,8	2,9	2,10	2,11
5,0	5,1	5,2	5,3	5,4	**5,5**	5,6	5,7	5,8	5,9	5,10	5,11
8,0	8,1	8,2	8,3	8,4	8,5	8,6	8,7	**8,8**	8,9	8,10	8,11
11,0	11,1	11,2	11,3	11,4	11,5	11,6	11,7	11,8	11,9	11,10	**11,11**

- Recall: $Ax = y$, the inducing vectors x and y were distributed identically.

Observations

- Traditional matrix distributions can be induced:
  - ⋆ One dimensional distributions:
    - ◇ Blocked by rows or columns
    - ◇ Wrapped by rows or columns
    - ◇ · · ·
  - ⋆ Two dimensional distributions:
    - ◇ Blocked
    - ◇ Wrapped
    - ◇ · · ·

Special Case: $P_x = P_y$

Vectors x and y are identically distributed.

Observations:

- Diagonal blocks are perfectly distributed.
- Blocks $A_{i,j}$ s.t. $i + j = constant$ are perfectly distributed.
- Benefits:
  - $\star$ Preconditioning.
  - $\star$ Load balance during matrix operation.
  - $\star$ Load balance during filling of matrix.
  - $\star \cdots$
- *Solves problems discussed in Example 1–3.*

Special Case: $P_x = P_y$

Benefits to the Chemistry community:

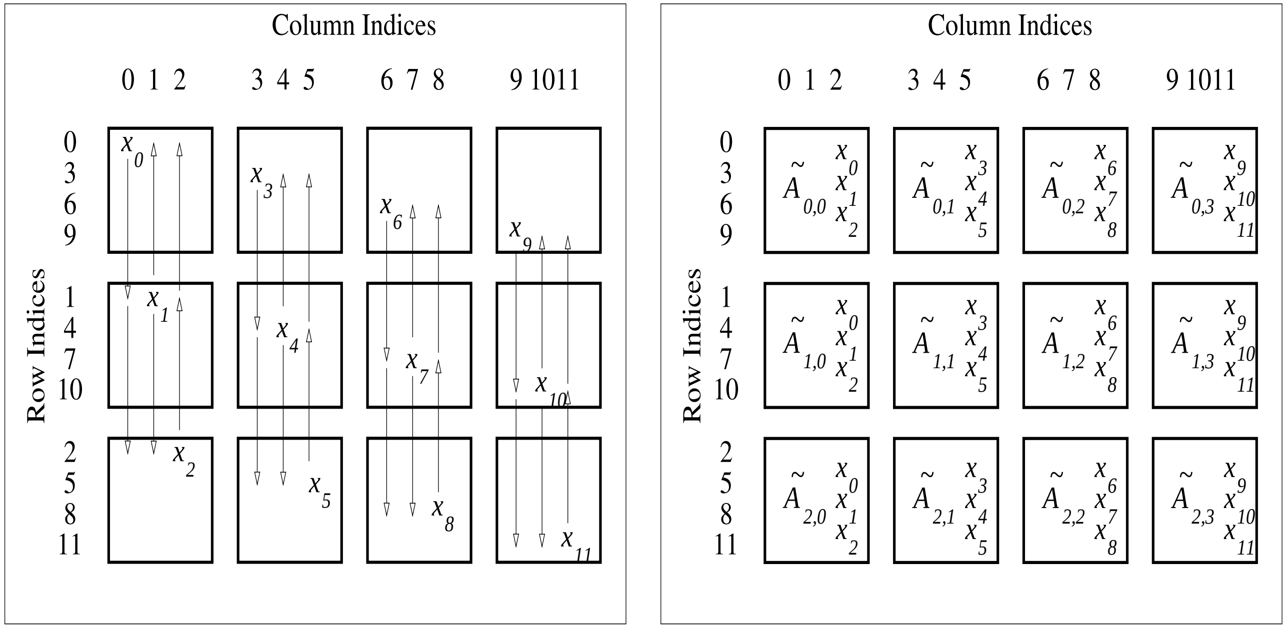
- Diagonal blocks are perfectly distributed.
  - ⋆ Tridiagonal perfectly distributed to all nodes after reduction to tridiagonal form.
  - ⋆ Eigenvalues perfectly distributed to all nodes after reduction to diagonal form.
  - ⋆ Computation of Jacobi rotations perfectly distributed when using a Jacobi method with odd-even ordering.
  - ⋆ Preconditioning in iterative methods.
- Blocks $A_{i,j}$ s.t. $i + j = constant$ are perfectly distributed.
  - ⋆ Symmetric matrix-vector multiply perfectly distributed.

Manipulating vectors, rows, and columns

Important Operations for BLAS

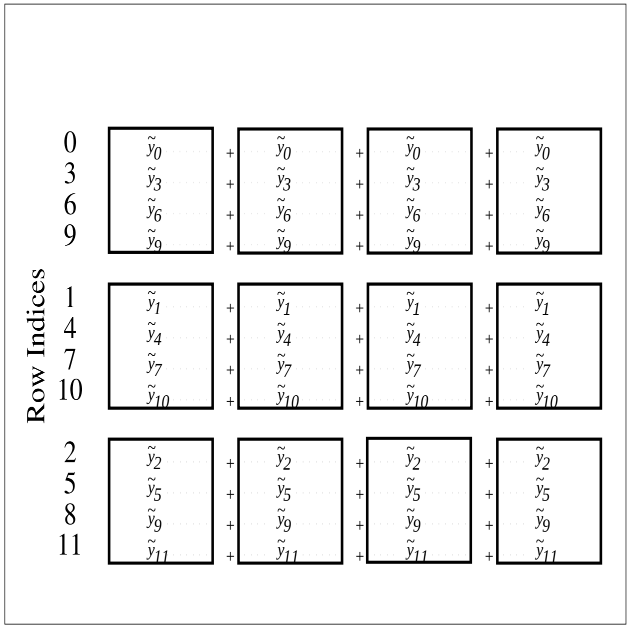
- The distribution of an inducing vector becomes an important intermediate step.
- Projecting and unprojecting:
  - ⋆ Matrix-row to vector, or visa versa.
  - ⋆ Matrix-column to vector, or visa versa.
  - ⋆ Matrix-column to matrix-row, or visa versa.
- “Spreading” a vector, matrix-row, or matrix-column is a frequent operation:
  - ⋆ Spread vector within node columns or node rows.
  - ⋆ Spread matrix-row within node columns; Spread matrix-column within node rows.
  - ⋆ Spread matrix-row within node rows; Spread matrix-column with node columns.

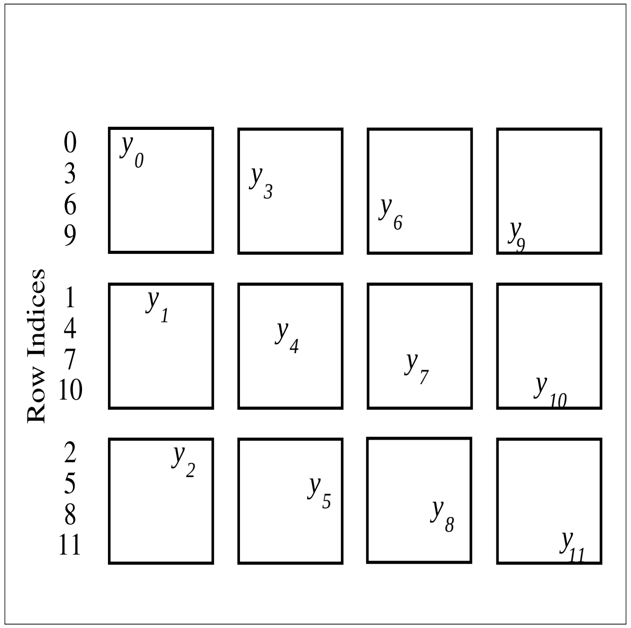
Motivation: Matrix-Vector Multiply



- Step 1: Spread x.
- Step 2: Local MV multiply.

Matrix-Vector Multiply: Algorithm (continued)





- Step 3: Distributed Sum within rows.

Matrix-Vector Multiply: Other Cases

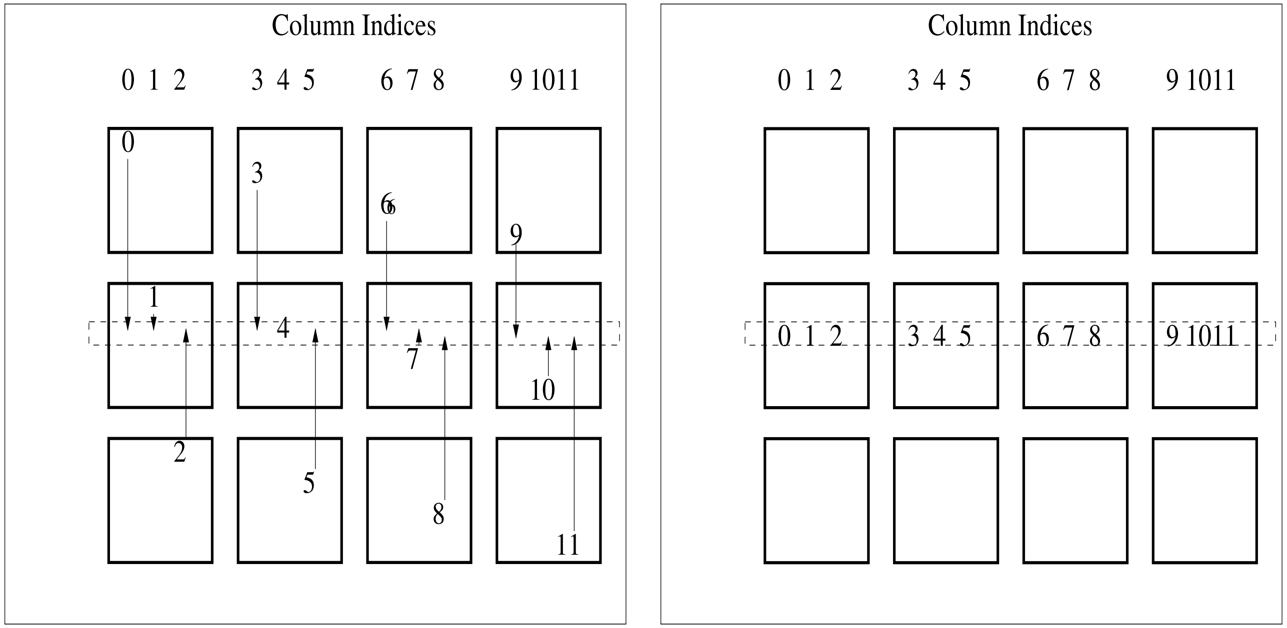
Consider $y = Ax$

- Case: x is a matrix-row, y is a matrix-column.
  - $\star$ Step 1: Redistribute matrix-row x to (inducing) vector (distribution).
  - $\star$ Step 2: Perform matrix-vector multiply.
  - $\star$ Step 3: Redistribute y from vector to matrix-column.
  - $\star$ **Shortcut: broadcast x within node columns, sum-to-one within node rows to yield y**

Matrix-Vector Multiply: Other Cases (continued)

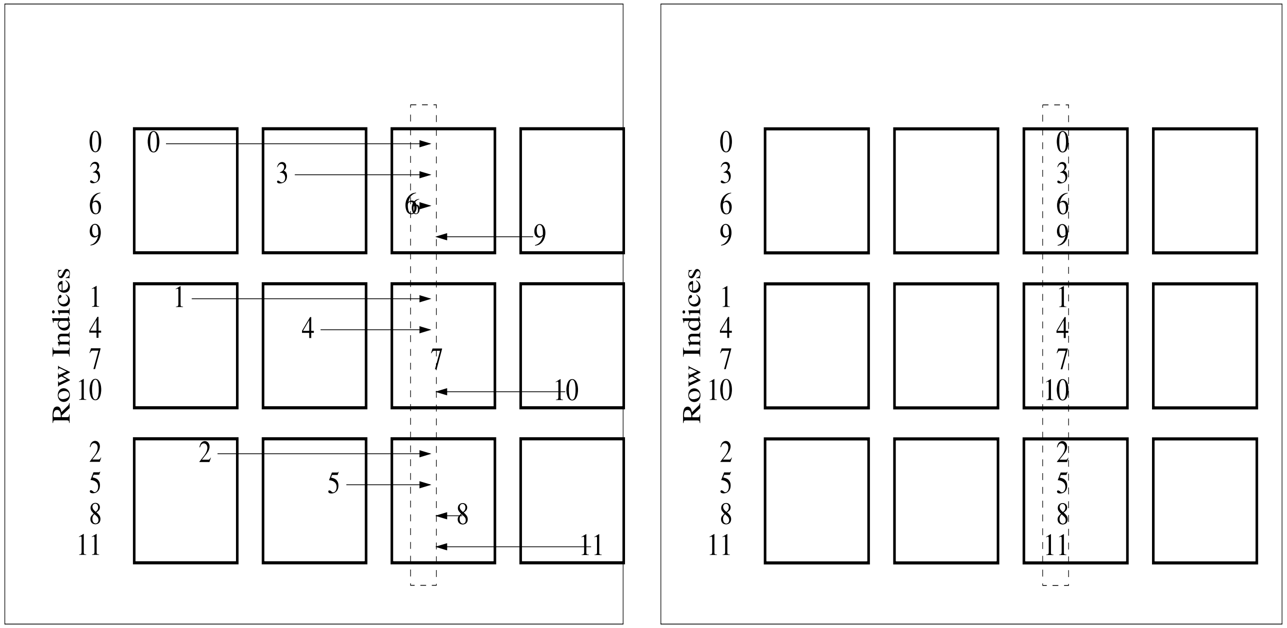
- Case: x is a matrix-column, y is a matrix-row.
  - $\star$ Step 1a: Redistribute matrix-row x to (inducing) vector distribution.
  - $\star$ Step 1b: Permute if necessary.
  - $\star$ Step 2: Perform matrix-vector multiply.
  - $\star$ Step 3a: Permute if necessary.
  - $\star$ Step 3b: Redistribute y from vector to matrix-column.

Vector to matrix-row



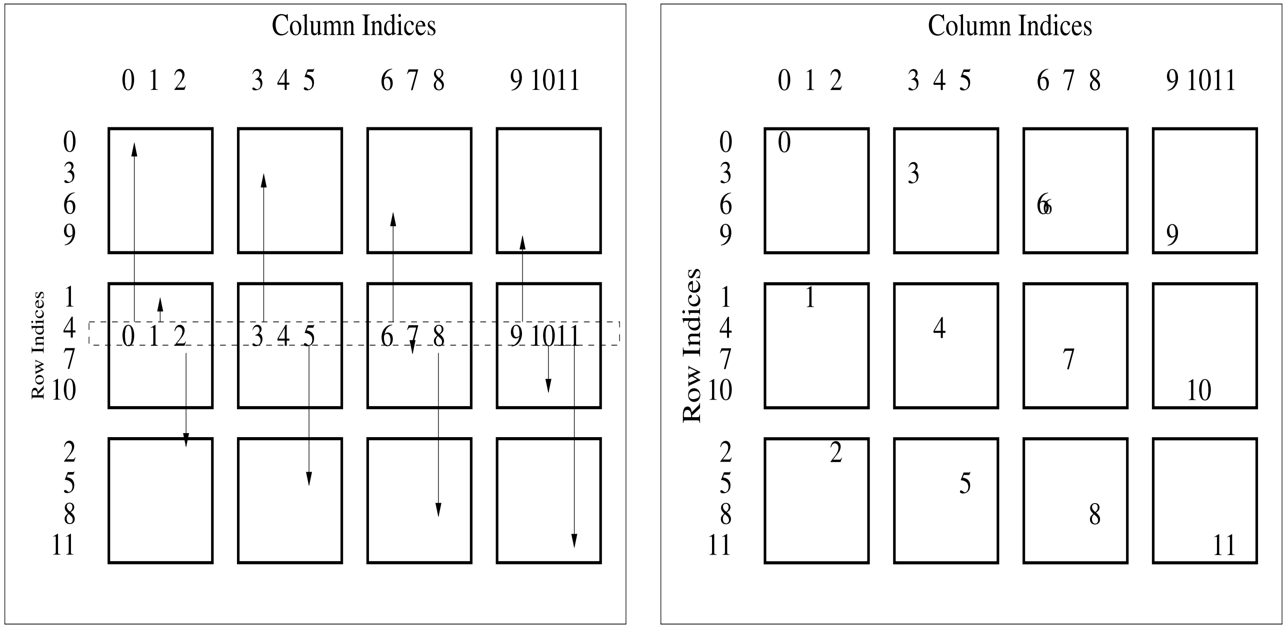
- Vector-to-row: project (gather).
- Row-to-vector: unproject (scatter).

Vector to matrix-column



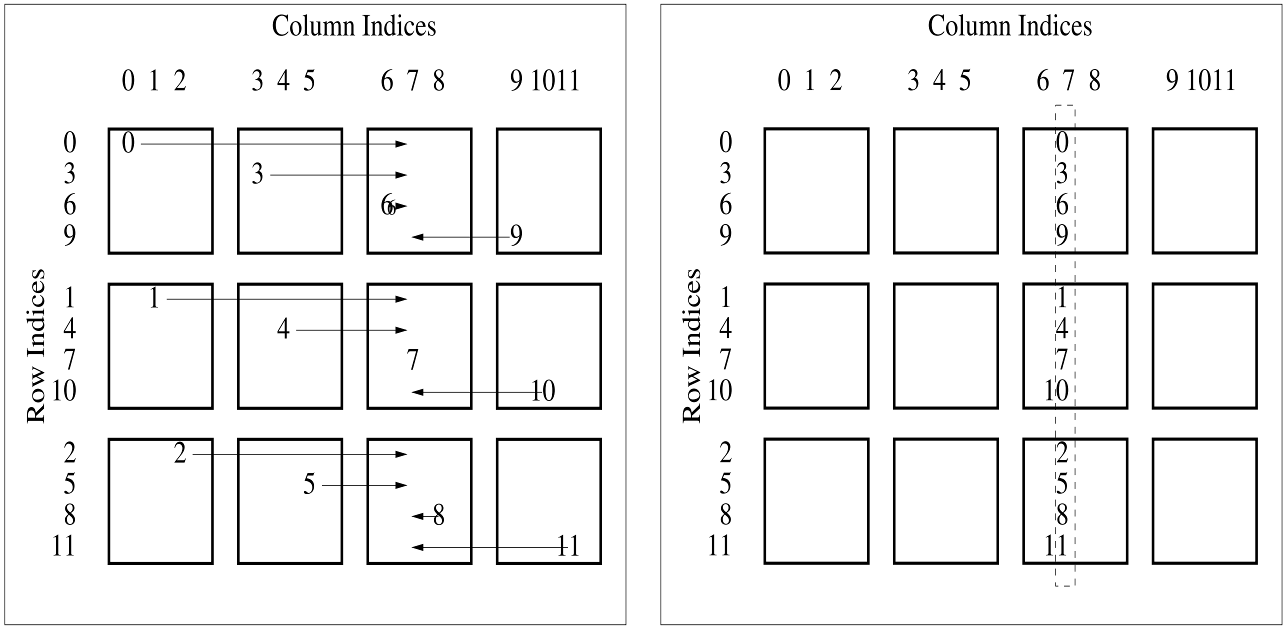
- Vector-to-column: project.
- Column-to-vector: unproject.

Matrix-row to matrix-column



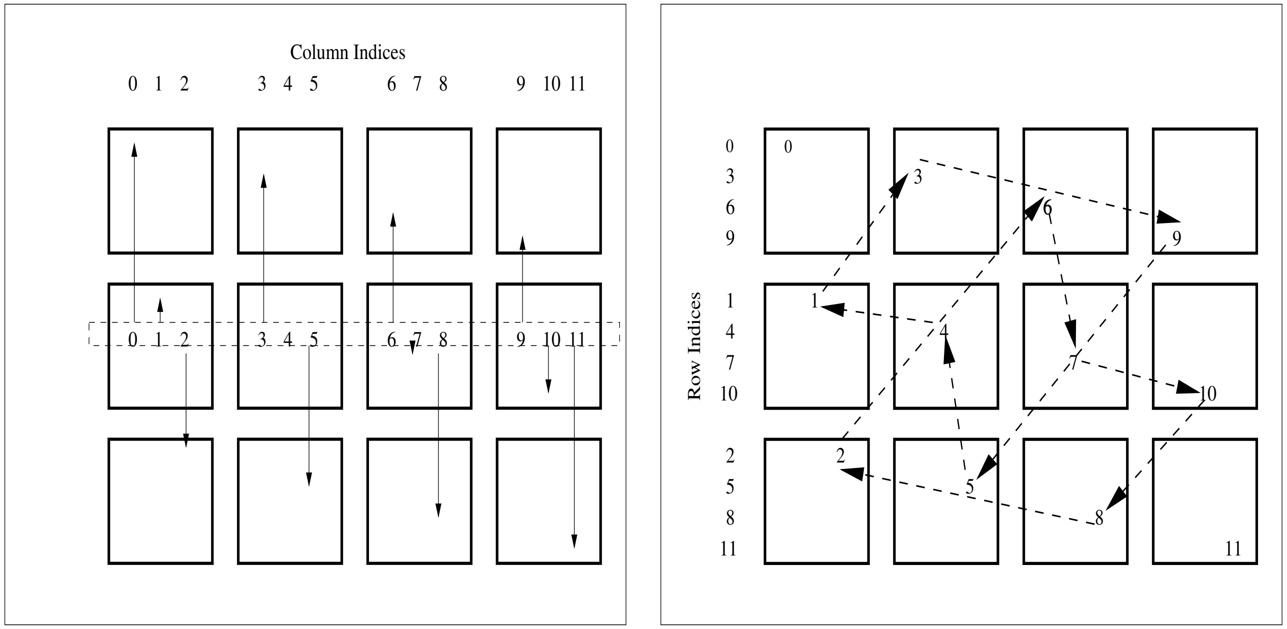
- Step 1: Matrix-row to vector.

Matrix-row to matrix-column (continued)



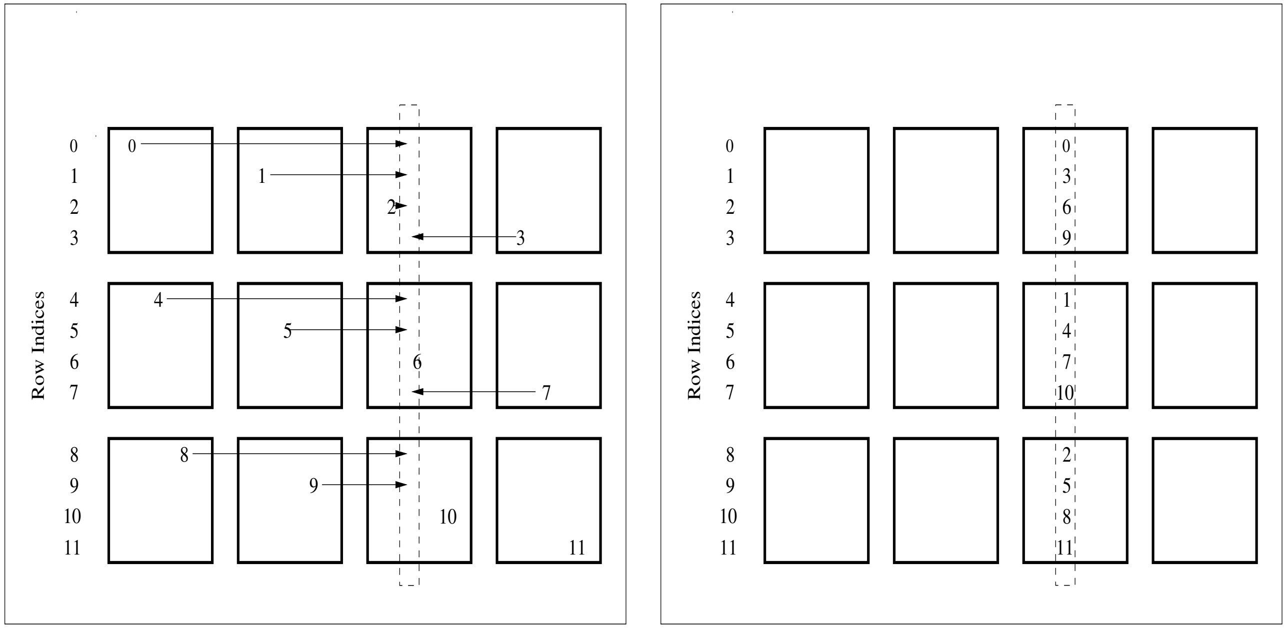
- Step 2: Vector to matrix-row.
- Notice: same inducing vector distribution for both matrix-row and -column distribution.

Row-to-column (Traditional Distribution)



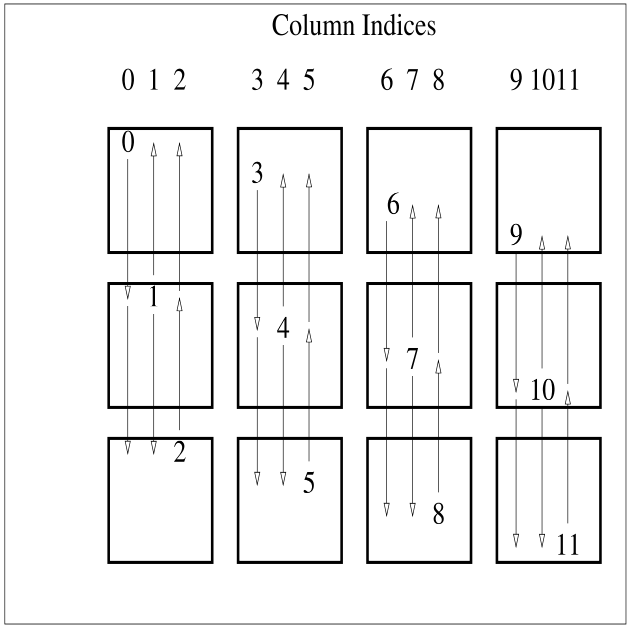
- Step 1: Matrix-row to vector.
- Step 2: Permute vector. Notice: different inducing vector distributions induce matrix-row and -column distributions.

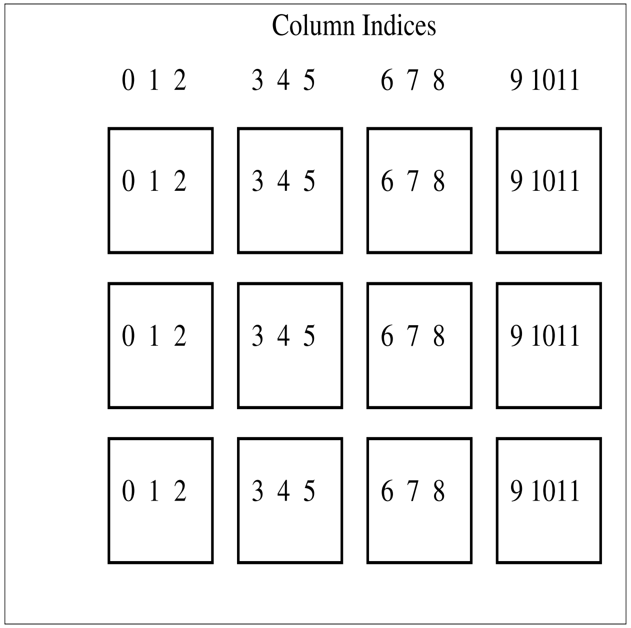
Matrix-row to matrix-column (Traditional Distribution) (continued)



- Step 3: Vector to matrix-column.

Spread Vector within Node Columns





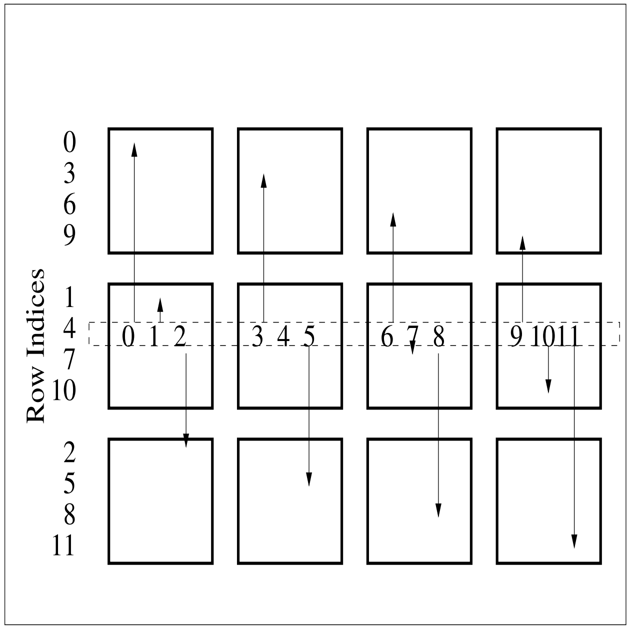
Spread Matrix-row within Node Columns

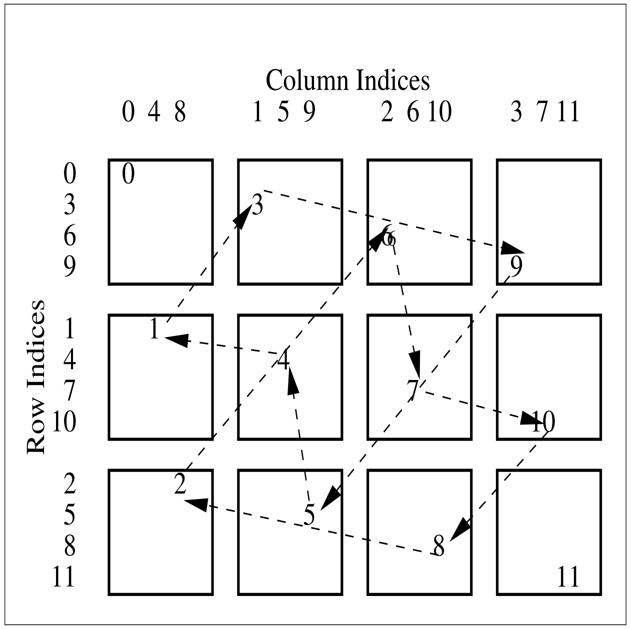
- Step 1: Matrix-row to vector.
- Step 2: Spread vector within node columns.
- **Shortcut: Broadcast matrix-row within node columns**

Spread Matrix-Column within Node Rows

- Step 1: Matrix-column to vector.
- Step 2: Spread vector within node rows.
- **Shortcut: Broadcast matrix column with node rows**

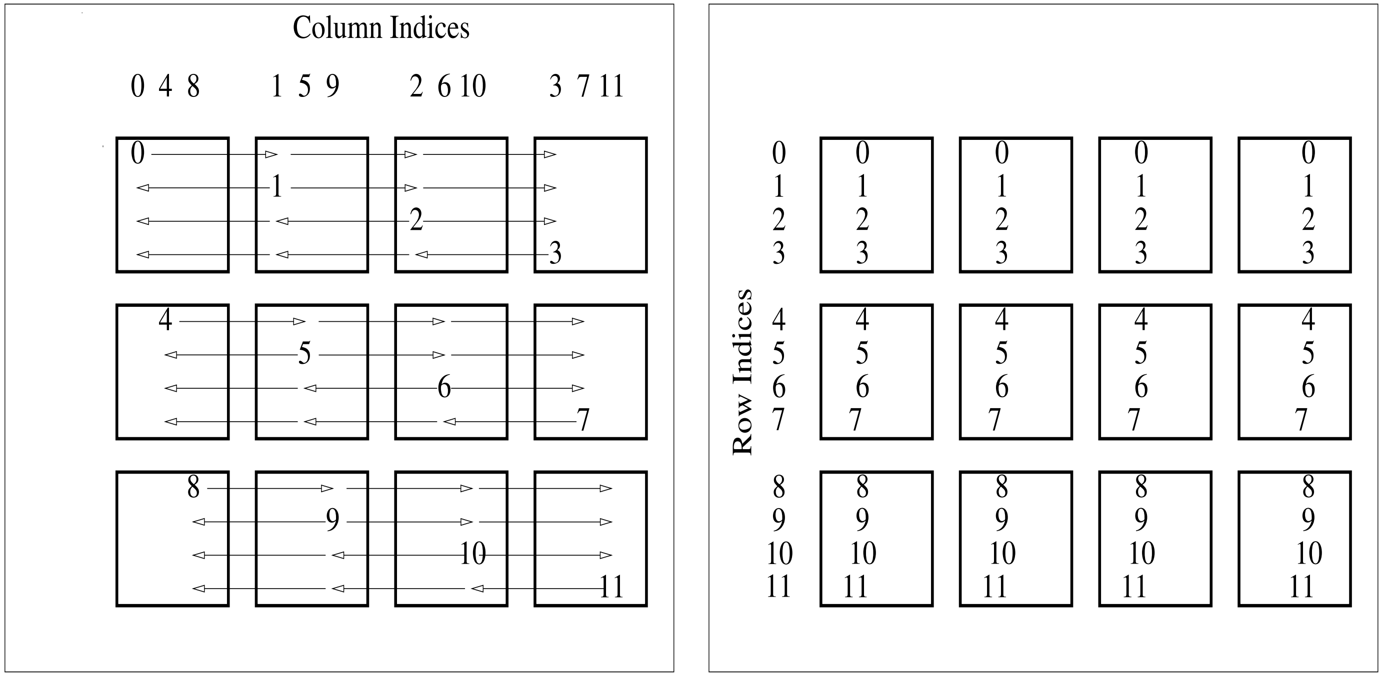
Spread Matrix-Row within Node Rows





- Step 1: Matrix-row to vector.
- Step 2: Permute.

Spread Matrix-Row within Node Rows (continued)



- Step 3: Spread vector within node rows.

Parallel Implementation of BLAS

We will first show:

- The presented vector manipulations can be used to parallelize the level 2 BLAS.
- The resulting level 2 BLAS can be used to parallelize the level 3 BLAS.
- The introduction of “level 2.5 BLAS” can be used to efficiently implement the level 3 BLAS.
- Performance results for level 3 BLAS using these techniques (in the simplified setting of “traditional” matrix distributions).

Preliminary

We will first show:

- The presented vector manipulations can be used to parallelize the level 2 BLAS.
- The resulting level 2 BLAS can be used to parallelize the level 3 BLAS.
- The introduction of "level 2.5 BLAS" can be used to efficiently implement the level 3 BLAS.
- Performance results for level 3 BLAS using these techniques (in the simplified setting of "traditional" matrix distributions).

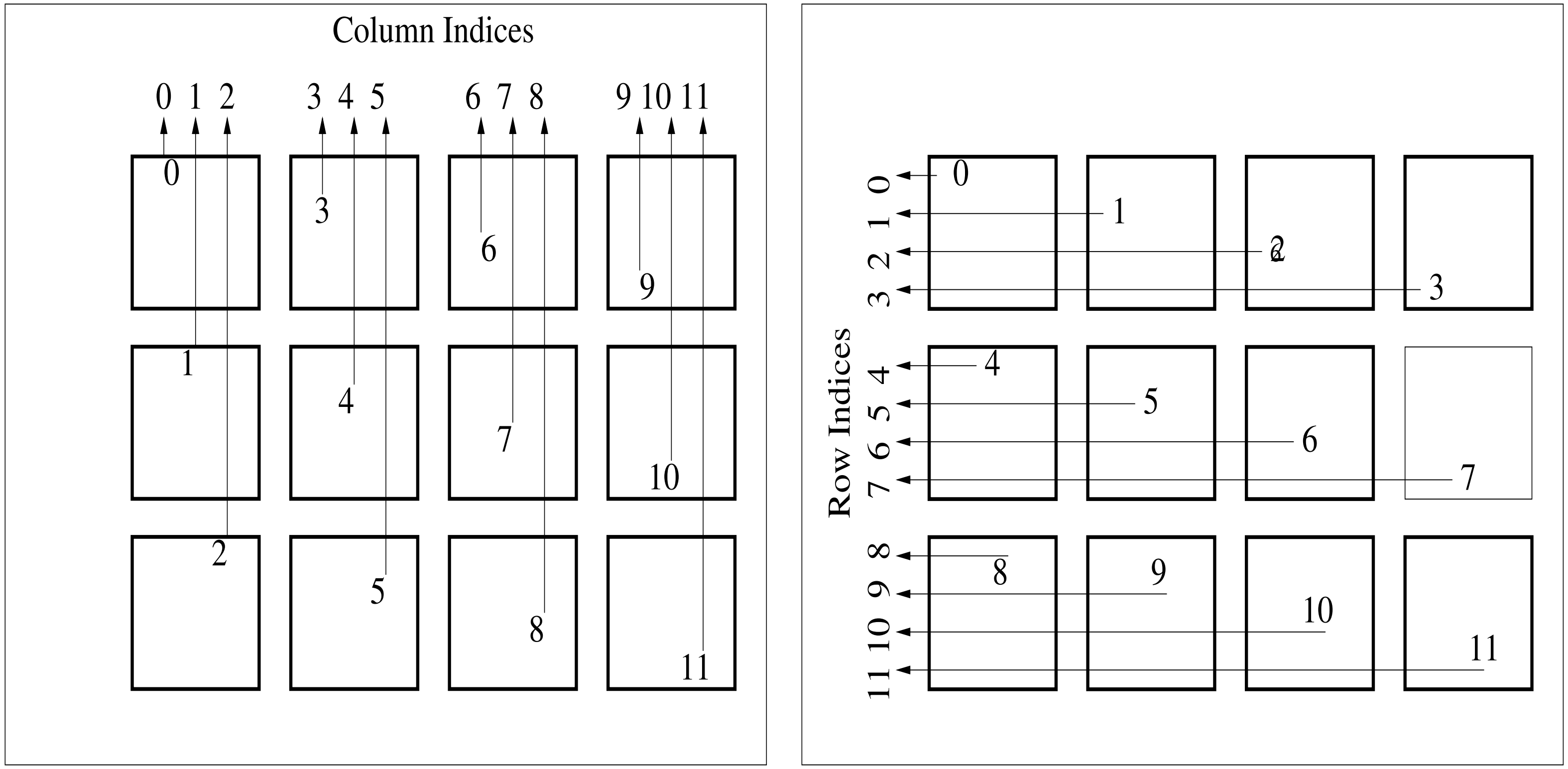
BLAS implementation: Partitioning

Partition as before:

$$P_x x = \left(\begin{array}{c} x_0 \\ \hline x_1 \\ \hline \vdots \\ \hline x_{p-1} \end{array} \right) \quad \text{and} \quad P_y y = \left(\begin{array}{c} y_0 \\ \hline y_1 \\ \hline \vdots \\ \hline y_{p-1} \end{array} \right)$$

$$P_y A P_x^T = \left(\begin{array}{c|c|c|c} A_{0,0} & A_{0,1} & \cdots & A_{0,p-1} \\ \hline A_{1,0} & A_{1,1} & \cdots & A_{1,p-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline A_{p-1,0} & A_{p-1,1} & \cdots & A_{p-1,p-1} \end{array} \right)$$

BLAS implementation: Distribution



BLAS implementation: Distribution (Continued)

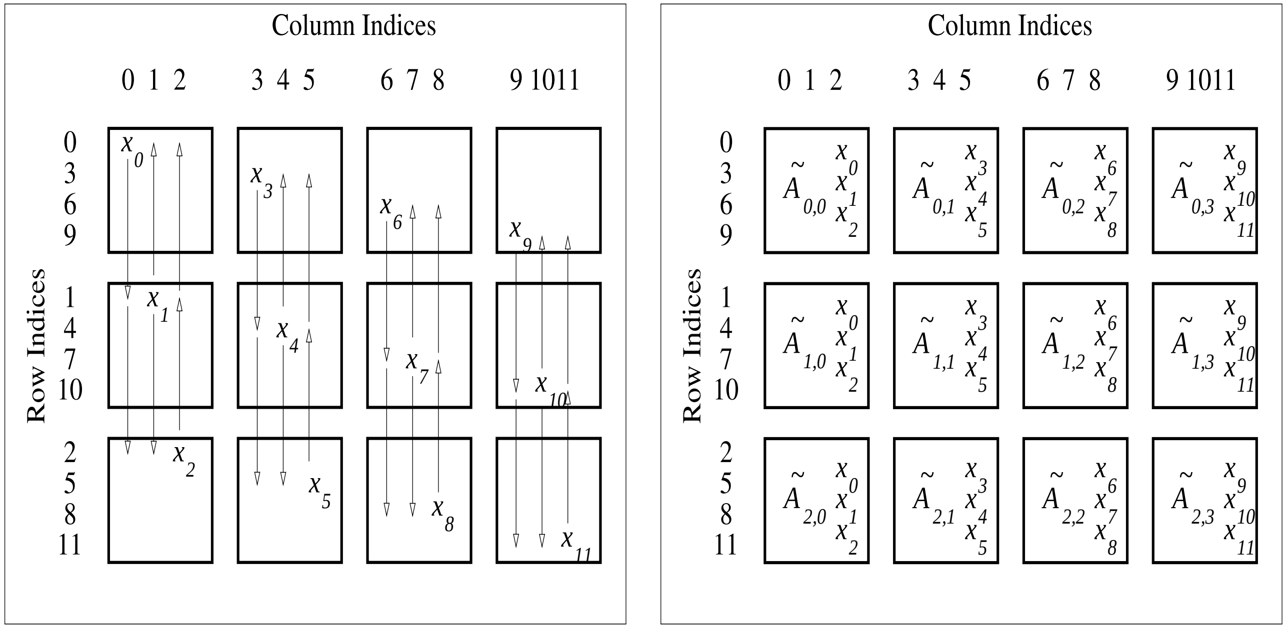
0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,7	0,8	0,9	0,10	0,11
1,0	1,1	1,2	1,3	1,4	1,5	1,6	1,7	1,8	1,9	1,10	1,11
2,0	2,1	2,2	2,3	2,4	2,5	2,6	2,7	2,8	2,9	2,10	2,11
3,0	3,1	3,2	3,3	3,4	3,5	3,6	3,7	3,8	3,9	3,10	3,11

4,0	4,1	4,2	4,3	4,4	4,5	4,6	4,7	4,8	4,9	4,10	4,11
5,0	5,1	5,2	5,3	5,4	5,5	5,6	5,7	5,8	5,9	5,10	5,11
6,0	6,1	6,2	6,3	6,4	6,5	6,6	6,7	6,8	6,9	6,10	6,11
7,0	7,1	7,2	7,3	7,4	7,5	7,6	7,7	7,8	7,9	7,10	7,11

8,0	8,1	8,2	8,3	8,4	8,5	8,6	8,7	8,8	8,9	8,10	8,11
9,0	9,1	9,2	9,3	9,4	9,5	9,6	9,7	9,8	9,9	9,10	9,11
10,0	10,1	10,2	10,3	10,4	10,5	10,6	10,7	10,8	10,9	10,10	10,11
11,0	11,1	11,2	11,3	11,4	11,5	11,6	11,7	11,8	11,9	11,10	11,11

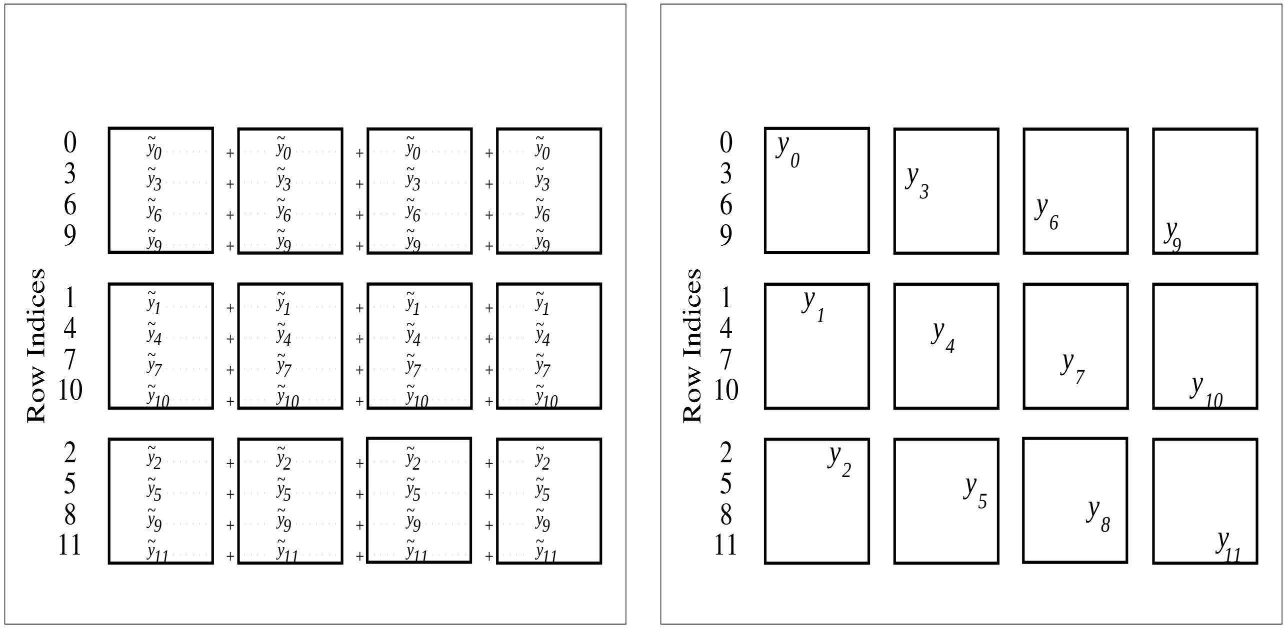
Parallelizing Level 2 BLAS

Matrix-Vector Multiply: Algorithm



- Step 1: Spread x.
- Step 2: Local MV multiply.

Matrix-Vector Multiply: Algorithm (continued)



- Step 3: Distributed Sum within rows.

Matrix-Vector Multiply: Other Cases

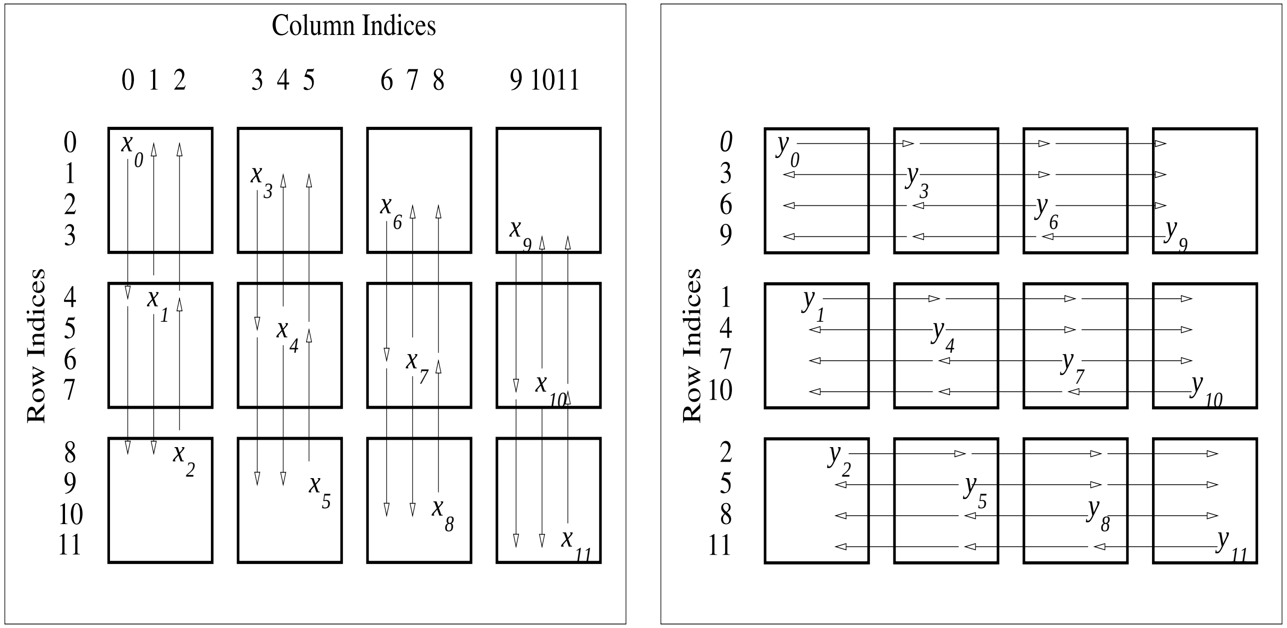
Consider $y = Ax$

- Case: x is a row, y is a column.
  - $\star$ Step 1: Redistribute row x to inducing vector distribution.
  - $\star$ Step 2: Perform matrix-vector multiply.
  - $\star$ Step 3: Redistribute y from inducing vector distribution.
  - $\star$ **Shortcut: broadcast x within columns, sum-to-one within rows to yield y**

Matrix-Vector Multiply: Other Cases (continued)

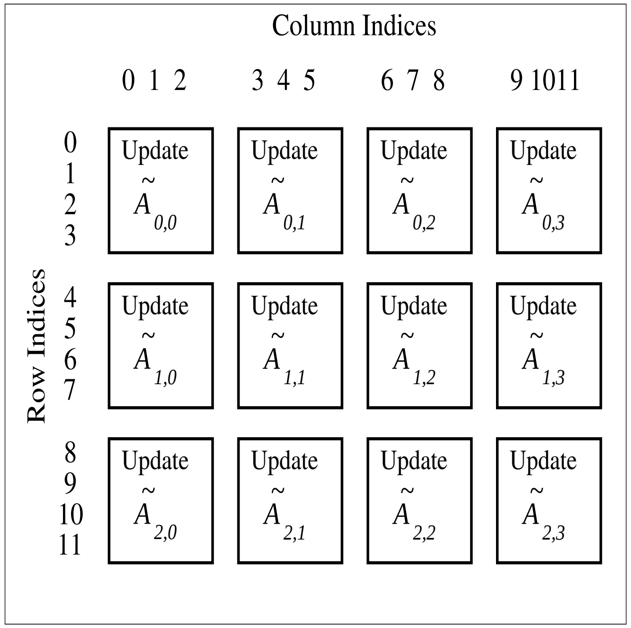
- Case: x is a column, y is a row.
  - ⋆ Step 1a: Redistribute row x to inducing vector distribution.
  - ⋆ Step 1b: Permute if necessary.
  - ⋆ Step 2: Perform matrix-vector multiply.
  - ⋆ Step 3a: Permute if necessary.
  - ⋆ Step 3b: Redistribute y from inducing vector distribution to column.

Rank-1 Update: Algorithm



- Step 1: Spread x.
- Step 2: Spread y.

Rank-1 Update: Algorithm (continued)



- Step 3: Local rank-1 update.

Parallelizing Level 3 BLAS

Building Level 3 BLAS on Level 2 BLAS

- All level 3 BLAS can be effectively parallelized using the Matrix-Vector multiply and Rank-1 update.
- Efficiency can be attained by introducing “Level 2.5” BLAS: Matrix-Panel and Rank-k update.
- Reference: Almadena Chtchelkanova, John Gunnels, Greg Morrow, James Overfelt, Robert A. van de Geijn, "Parallel Implementation of BLAS: General Techniques for Level 3 BLAS," *Concurrency: Practice and Experience,* to appear.

Partitioning the Matrices

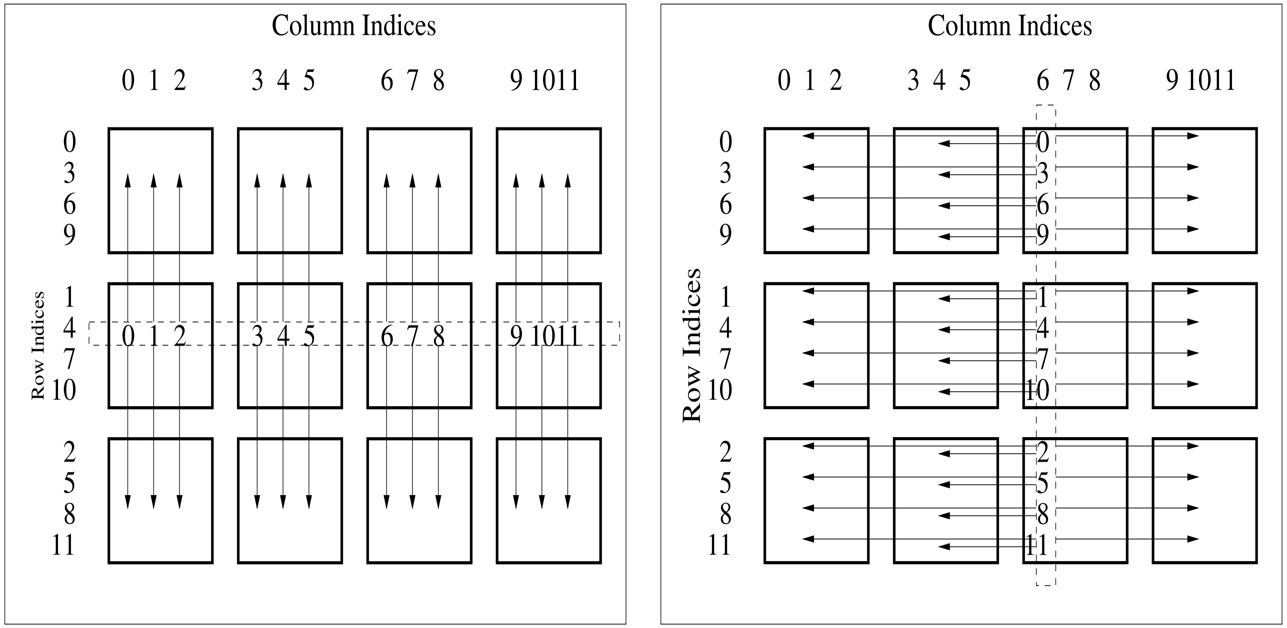
$$A = \left(\begin{array}{c|c|c|c} A_{0,0} & A_{0,1} & \cdots & A_{0,N-1} \\ \hline A_{1,0} & A_{1,1} & \cdots & A_{1,N-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline A_{N-1,0} & A_{N-1,1} & \cdots & A_{N-1,N-1} \end{array} \right)$$

- Same for B and C.

Matrix-Matrix Multiply: $C = AB$

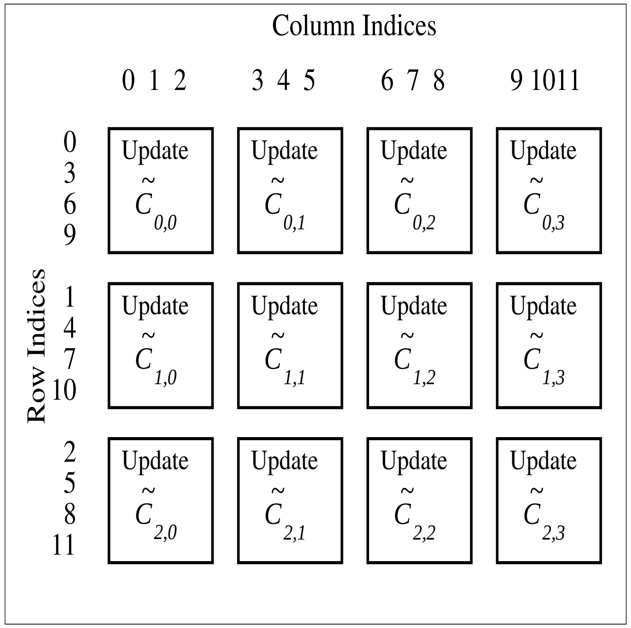
$$
\begin{aligned}
C \;=\; AB &= \left(\begin{array}{c} A_{00} \\ \hline A_{10} \\ \hline \vdots \\ \hline A_{(N-1)0} \end{array} \right) \left(\begin{array}{c|c|c|c} B_{00} & B_{01} & \cdots & B_{0(N-1)} \end{array} \right) && (1) \\
&+ \left(\begin{array}{c} A_{01} \\ \hline A_{11} \\ \hline \vdots \\ \hline A_{(N-1)1} \end{array} \right) \left(\begin{array}{c|c|c|c} B_{10} & B_{11} & \cdots & B_{1(N-1)} \end{array} \right) + \cdots && (2) \\
&+ \left(\begin{array}{c} A_{0(N-1)} \\ \hline A_{1(N-1)} \\ \hline \vdots \\ \hline A_{(N-1)(N-1)} \end{array} \right) \left(\begin{array}{c|c|c|c} B_{(N-1)0} & B_{(N-1)1} & \cdots & B_{(N-1)(N-1)} \end{array} \right) && (3)
\end{aligned}
$$

Matrix-Matrix Multiply: $C = AB$ (continued)



- Step 1: Spread current row of B
- Step 2: Spread current column of A.

Matrix-Matrix Multiply: $C = AB$ (continued)



- Step 3: Local rank-1 update.

Matrix-Matrix Multiply: $C = AB^T$

$$\left(\begin{array}{c} C_{0i} \\ \hline C_{1i} \\ \hline \vdots \\ \hline C_{(N-1)i} \end{array} \right) = A \left(\, B_{i0} \, \middle| \, B_{i1} \, \middle| \cdots \middle| \, B_{i(N-1)} \, \right)$$

- C is computed one column at a time.

Matrix-Matrix Multiply: $C = A^T B^T$

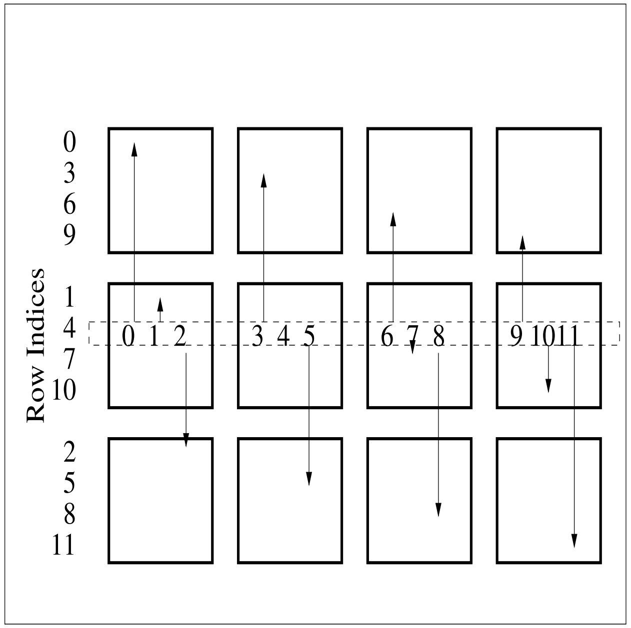
$$C \;=\; A^T B^T = \left(A_{00} \,\middle|\, A_{01} \,\middle|\, \cdots \,\middle|\, A_{0(N-1)} \right)^T \left(\begin{array}{c} B_{00} \\ \hline B_{10} \\ \hline \vdots \\ \hline B_{(N-1)0} \end{array} \right)^T \tag{4}$$

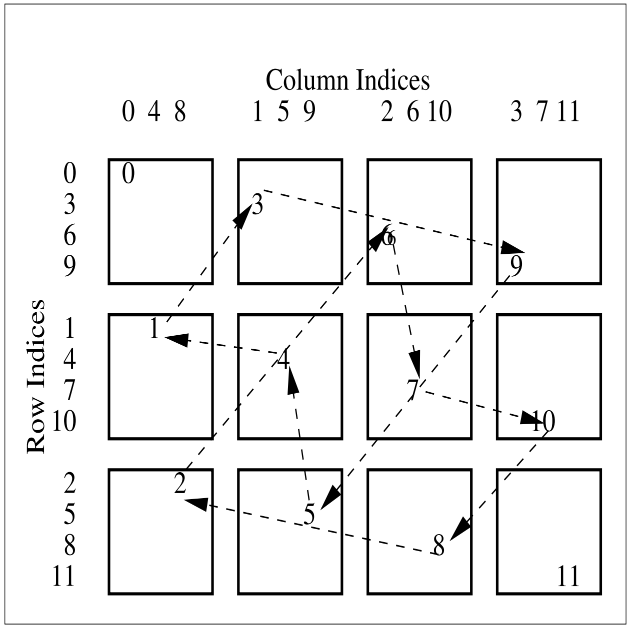
$$+ \;\; \left(A_{10} \,\middle|\, A_{11} \,\middle|\, \cdots \,\middle|\, A_{1(N-1)} \right)^T \left(\begin{array}{c} B_{01} \\ \hline B_{11} \\ \hline \vdots \\ \hline B_{(N-1)1} \end{array} \right)^T \;\; + \cdots \tag{5}$$

Matrix-Matrix Multiply: $C = A^T B^T$ (continued)

- Step 1: Row of A to inducing vector distribution.
- Step 2: Redistributed (if necessary) like vector that induces row distribution of C.
- Step 3: Spread within rows.
- Step 4: Column of B to inducing vector distribution.
- Step 5: Redistributed (if necessary) like vector that induces column distribution of C.
- Step 6: Spread within columnss.
- Step 7: Peform local rank-1 update.

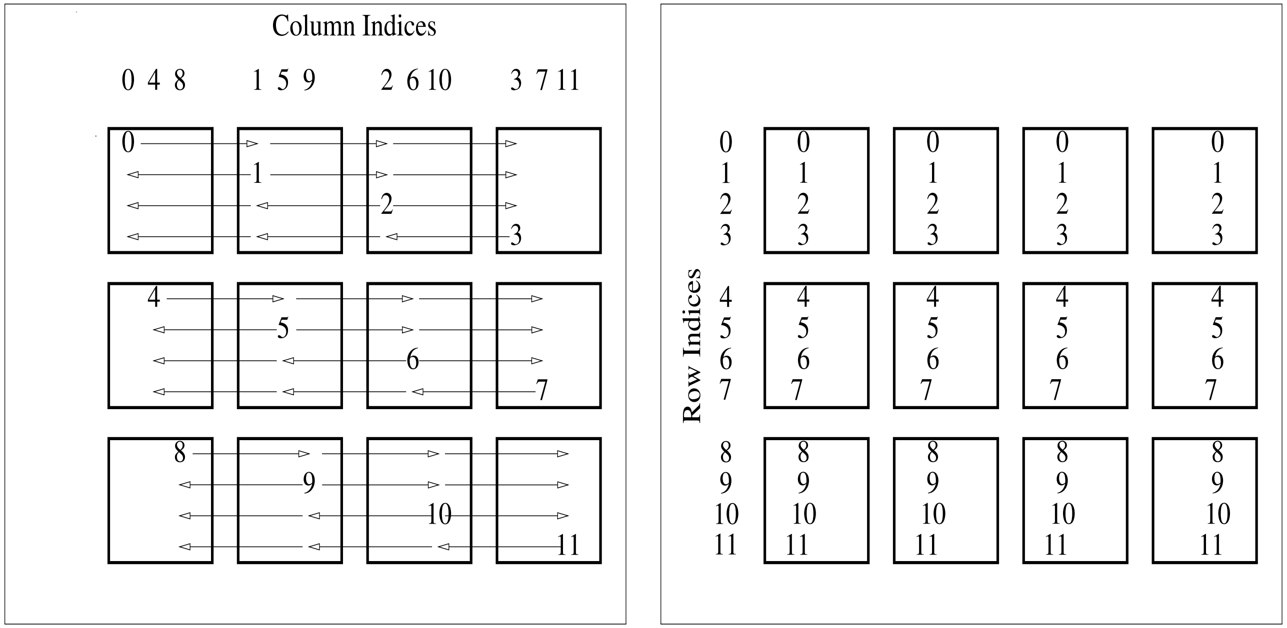
Matrix-Matrix Multiply: $C = A^T B^T$ (continued)





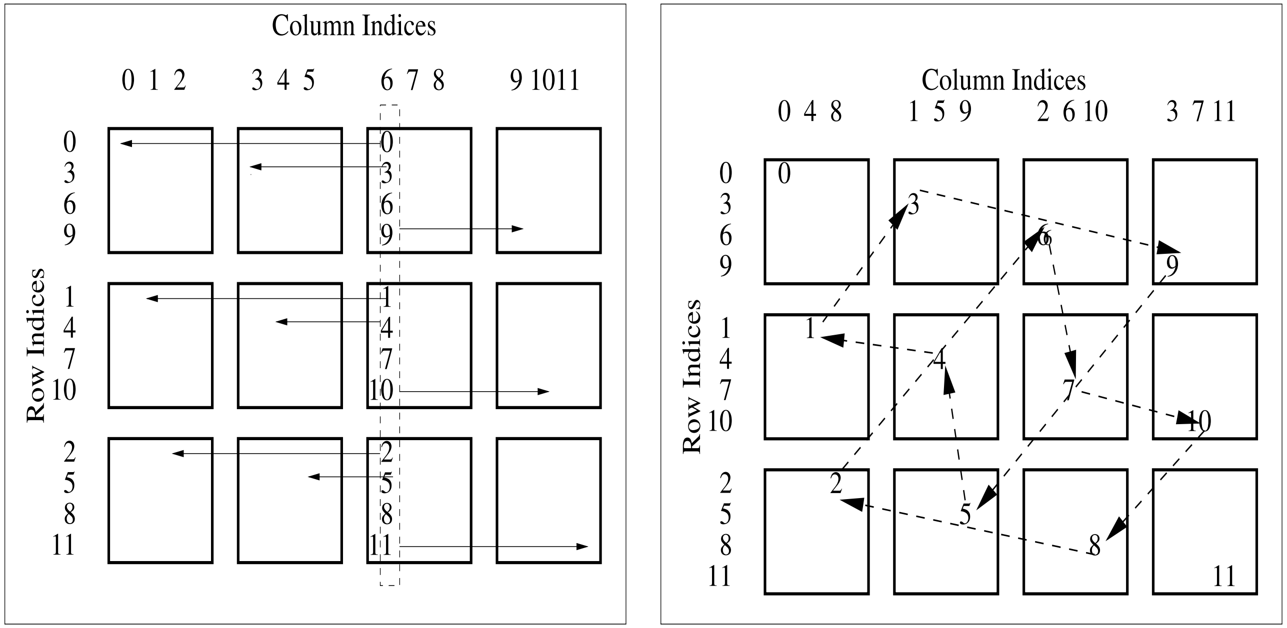
- Step 1: Spread current row of A within rows.

Matrix-Matrix Multiply: $C = A^T B^T$ (continued)

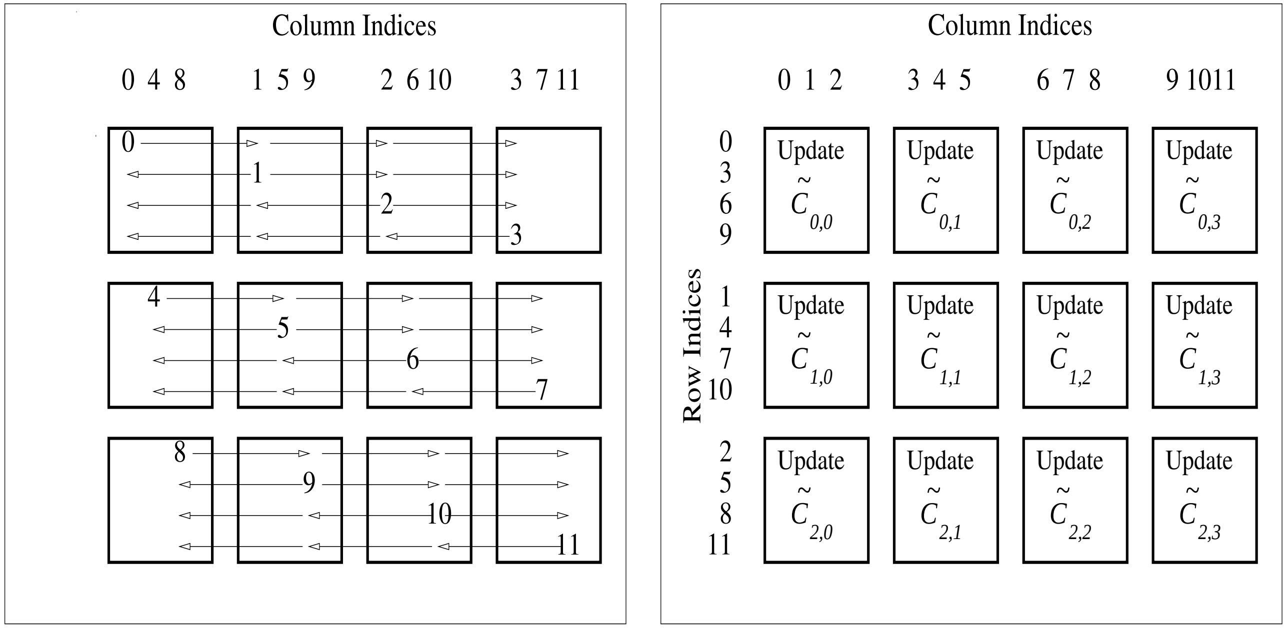


- Step 1: Spread current row of A within rows (continued).

Matrix-Matrix Multiply: $C = A^T B^T$ (continued)



- Step 2: Spread current column of B within columns.



- Step 2: Spread current column of B within columns (continued).
- Step 3: Local rank-1 updates.

Triangular Solve with Multiple RHS: $B = A^{-T} B$

- Assume: A lower triangular.
- $AX = B$ where X overwrites B.

$$\left(\begin{array}{c||c|c|c} A_{00} & 0 & \cdots & 0 \\ \hline\hline A_{10} & A_{11} & \cdots & 0 \\ \hline \vdots & \vdots & & \vdots \\ \hline A_{(N-1)0} & A_{(N-1)1} & \cdots & A_{(N-1)(N-1)} \end{array}\right) \left(\begin{array}{c} X_{0*} \\ \hline\hline X_{1*} \\ \hline \vdots \\ \hline X_{(N-1)*} \end{array}\right) = \left(\begin{array}{c} B_{0*} \\ \hline\hline B_{1*} \\ \hline \vdots \\ \hline B_{(N-1)*} \end{array}\right)$$

Implies

$$A_{00} \left(X_{00} \,\middle|\, X_{01} \,\middle|\, \cdots \,\middle|\, X_{0(N-1)} \right) = \left(B_{00} \,\middle|\, B_{01} \,\middle|\, \cdots \,\middle|\, B_{0(N-1)} \right) \tag{6}$$

and

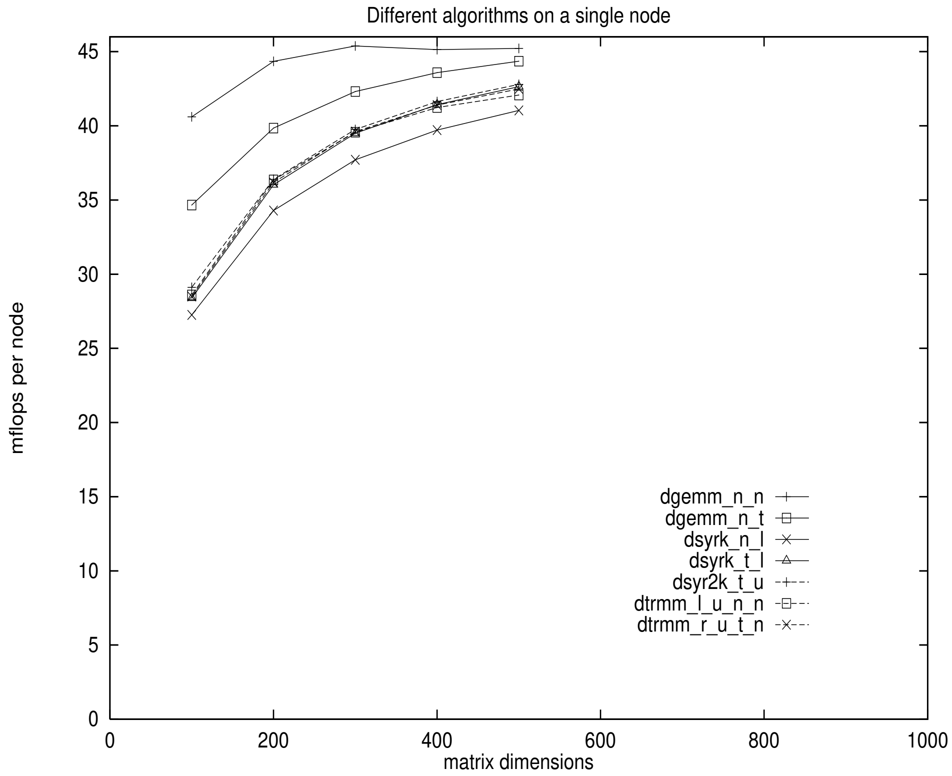
$$\left(\begin{array}{c|c|c} A_{11} & \cdots & 0 \\ \hline \vdots & & \vdots \\ \hline A_{(N-1)1} & \cdots & A_{(N-1)(N-1)} \end{array}\right) \left(\begin{array}{c} X_{1*} \\ \hline \vdots \\ \hline X_{(N-1)*} \end{array}\right) = \left(\begin{array}{c} B_{1*} \\ \hline \vdots \\ \hline B_{(N-1)*} \end{array}\right) - \left(\begin{array}{c} A_{10} \\ \hline \vdots \\ \hline A_{(N-1)0} \end{array}\right) \left(X_{0*} \right) \tag{7}$$

- Step 1: Compute X_{0*}.
- Step 2: Update B with a rank-1 update.
- Step 3: Recursively continue with smaller system.

Performance

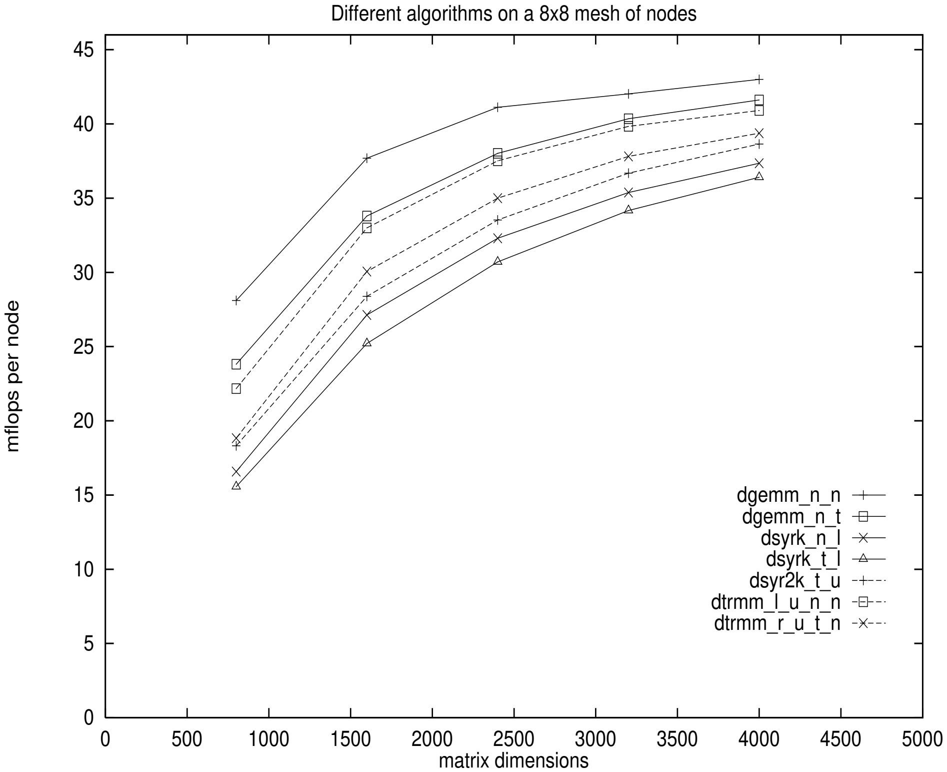
- Platform: 512 node Intel Paragon System at Caltech.
- OS: OSF R1.3.
- Communication library: MPI.
- Compute kernels: single node BLAS.

Performance (continued)



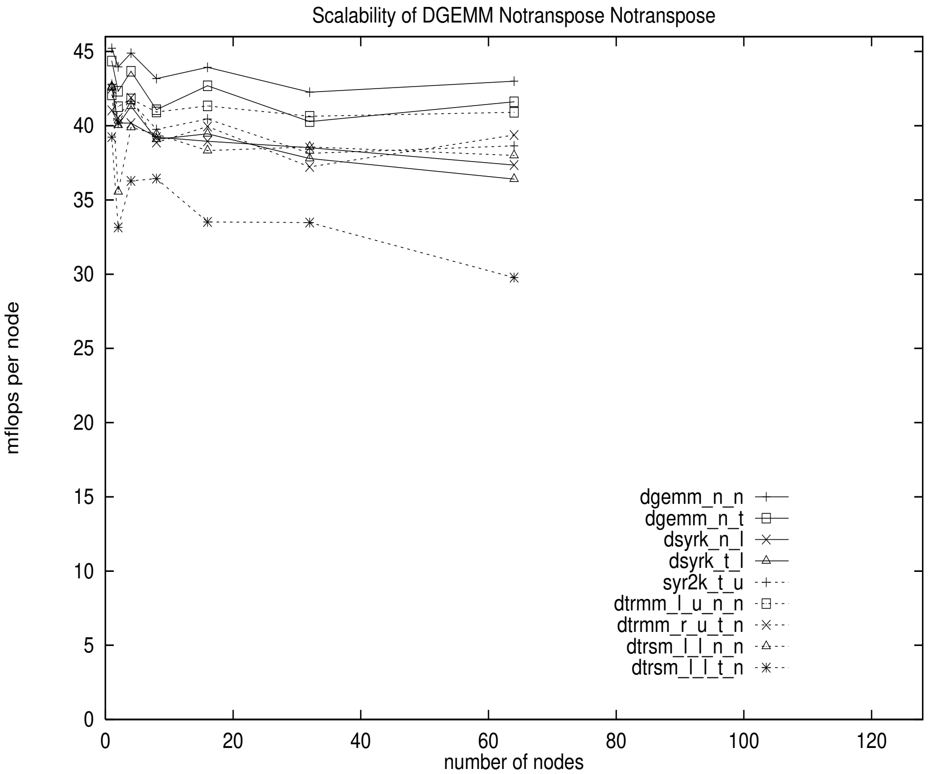
- Performance of various algorithms on a single node.

Performance (continued)



- Performance on 8×8 mesh.

Scalability



- Scalability memory utilization per node is equivalent to a 500×500 matrix.

$\boxed{\text{Building Libraries}}$

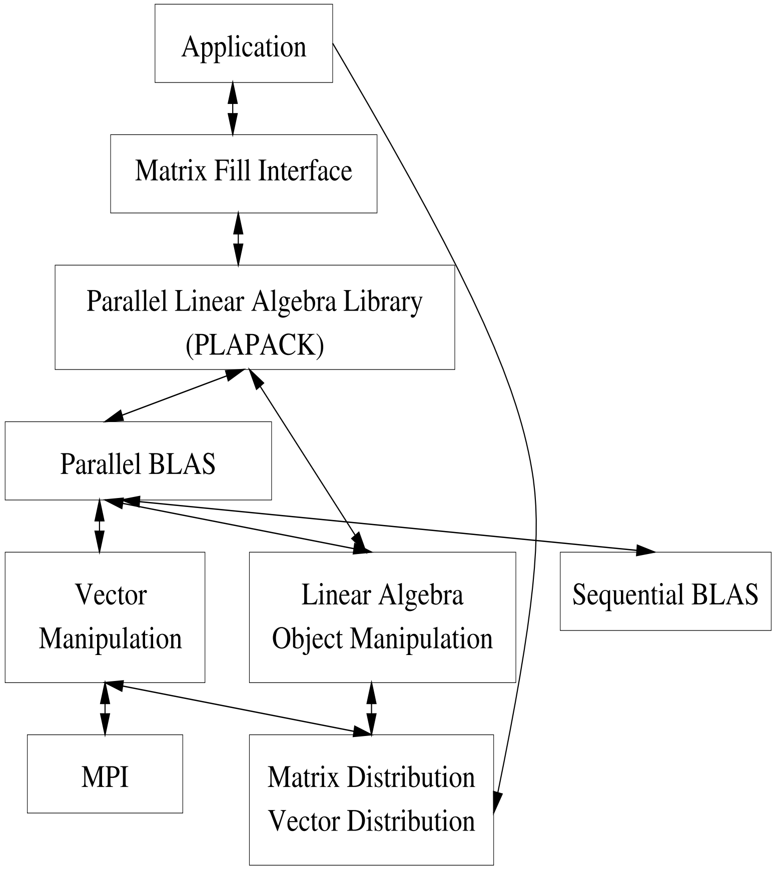
A Building Block Approach to Parallel Linear Algebra Libraries

- Implementation of `_dgemm` (matrix matrix multiplication):

  at least <u>32</u> cases!

- Solution:
  - $\star$ Building block approach (PLAPACK project).
  - $\star$ Object based (MPI-like) programming.

Parallel Linear Algebra Package (PLAPACK)



The SL_Library

- Prototype library.
- Uses Physically Based Matrix Distribution.
- Functionality similar to that of the current “Other”PACK.

Example implementation: $C = \alpha AB + \beta C$

(with the core plapack team)

- Form $C \leftarrow \alpha AB + \beta C$ using rank-nb updates.

```
int SL_Gemm_notransa_notransb ( int nb_alg, SL_LAObj alpha,
          SL_LAObj a, SL_LAObj b, SL_LAObj beta, SL_LAObj c )
{
  SL_LAObj acur = NULL, bcur = NULL, a1 = NULL, a2 = NULL,
              b1 = NULL, b2 = NULL,
  SL_Template Template = NULL;
  int currow, curcol, size_a, size_b, size;
```

- Scale $C \leftarrow \beta C$.

```
  SL_Scal( beta, c );
```

- Let $A_{\mathrm{cur}} = A$ and $B_{\mathrm{cur}} = B$

```
  SL_Matrix_submatrix( a, SL_DIM_ALL, SL_DIM_ALL,
                       SL_ALIGN_FIRST, SL_ALIGN_FIRST, &acur );
  SL_Matrix_submatrix( b, SL_DIM_ALL, SL_DIM_ALL,
                       SL_ALIGN_FIRST, SL_ALIGN_FIRST, &bcur );
```

- Loop until done

```
while ( TRUE ) {
```

  - $\star$ Determine width of next rank-k update.

```
SL_LAObj_left( acur, &size_a, &curcol );
SL_LAObj_top( bcur, &size_b,  &currow );
size = ( size_a < nb_alg ? size_a : nb_alg );
size = ( size_b < size   ? size_b : size   );
if ( size == 0 ) break;
```

  - $\star$ Let $A_{\text{cur}} = \left(A_1 \middle| A_2 \right)$ and $B_{\text{cur}} = \left(\frac{B_1}{B_2} \right)$.

```
SL_LAObj_vert_split_2( acur, size, &a1, &a2 );
SL_LAObj_horz_split_2( bcur, size, &b1, &b2 );
```

  - $\star$ Update $C \leftarrow C + \alpha A_1 B_1$.

```
SL_LAObj_matrix_annotate_set( a1, SL_COL_PANEL);
SL_LAObj_matrix_annotate_set( b1, SL_ROW_PANEL);
SL_Gemm_pan( alpha, a1, b1, SL_One, c);
```

  - $\star$ Update $A_{\text{cur}} = A_2$ and $B_{\text{cur}} = B_2$.

```
SL_LAObj_swap( &a2, &acur );   SL_LAObj_swap( &b2, &bcur );
```

- Continue loop.

```
    }

    SL_LAObj_free( &acur );    SL_LAObj_free( &bcur );
    SL_LAObj_free( &a1 );      SL_LAObj_free( &a2 );
    SL_LAObj_free( &b1 );      SL_LAObj_free( &b2 );
}
```

Performance

Example implementation: reduction to Hessenberg form

(with Philip Alpatov and Greg Morrow)

- Form $H \to QAQ^T$ where H is upperHessenberg.

```
int SL_dhrd2(SL_LAObj a)
{
   << declarations >>
```

- Extract global dimension and alignment information

```
SL_Matrix_global_all( a, &m, &n, &im, &in, &template);
```

- Create scalar objects, duplicated on all nodes, for α and β.

```
SL_Mscalar_create(datatype, 1, 1, template, &alpha);
SL_LAObj_buffer(alpha, &alpha_buffer);

SL_Mscalar_create(datatype, 1, 1, template, &min_beta);
SL_LAObj_buffer(min_beta, &min_beta_buffer);
```

- Create distributed vectors v, u, and y.

```
SL_Vector_create(datatype, n, template, im, &v);
SL_Vector_create(datatype, n, template, im, &u);
SL_Vector_create(datatype, n, template, im, &y);
```

- $A_{\mathrm{cur}} = A, u_{\mathrm{cur}} = u, v_{\mathrm{cur}} = v, y_{\mathrm{cur}} = y,$

```
SL_Matrix_submatrix( a, SL_DIM_ALL, SL_DIM_ALL,
    SL_ALIGN_FIRST, SL_ALIGN_FIRST, &acur );
SL_Vector_subvector( u, SL_DIM_ALL, SL_ALIGN_FIRST, &ucur );
SL_Vector_subvector( v, SL_DIM_ALL, SL_ALIGN_FIRST, &vcur );
SL_Vector_subvector( y, SL_DIM_ALL, SL_ALIGN_FIRST, &ycur );
```

- Loop until done

```
for(k=0;k<n-2;k++) {
```

- Let $A_{\text{cur}} = \left(\star \,\middle|\, A_2 \right) = \left(\begin{array}{c|c} \star & \star \\ \hline A_{21} & A_{22} \end{array} \right)$ where A_{21} is $n-k \times 1$ and A_{22} is $n-k \times n-k$

```
SL_LAObj_hoz_split_2( acur, 1, &star, &a2);
SL_LAObj_split_4( A_cur, k, 1,
                  &star, &star,
                  &a21 , &a22);
```

- Let $u_{\text{cur}} = \left(\begin{array}{c} \star \\ \hline u_{\text{cur}} \end{array} \right)$ reducing its length by one. Compute the Householder vector u_{cur} and scaling factor $-\beta$.

```
SL_LAObj_hoz_split_2( ucur, 1, &star.
                                &ucur);
SL_Copy( A_left, ucur );
compute_household_vector( ucur, min_beta );
```

- Let $v_{\mathrm{cur}} = \left(\frac{\star}{v_{\mathrm{cur}}} \right)$ reducing its length by one. Compute $v_{\mathrm{cur}} = A_{22}^T u_{\mathrm{cur}}$.

```
SL_LAObj_hoz_split_2( vcur, 1, &star.
                                &vcur);
SL_Gemv( SL_TRANSPOSE, SL_ONE, a22, ucur, SL_ZERO, vcur );
```

- Compute $y = A_2 u_{\mathrm{cur}}$.

```
SL_Gemv( SL_NOTRANSPOSE, SL_ONE, a2, ucur, SL_ZERO, y );
```

- Compute $\alpha = -\beta * v_{\mathrm{cur}}^T u_{\mathrm{cur}}$.

```
SL_dot( vcur, ucur, alpha );
*((double *)alpha_buffer) = *((double *)alpha_buffer) *
                    *((double *)min_beta_buffer);
```

- Let $y = \left(\frac{\star}{y_{\mathrm{cur}}} \right)$ where y_{cur} is of length $n - k$ (by updating view y_{cur}). Update $y_{\mathrm{cur}} = \alpha u_{\mathrm{cur}} + y_{\mathrm{cur}}$.

```
SL_LAObj_hoz_split_2( ycur, 1, &star, &ycur);
SL_Axpy( alpha, ucur, ycur );
```

- Update $A_{22} = A_{22} - \beta u_{\mathrm{cur}} v_{\mathrm{cur}}^T$.

```
SL_Ger( min_beta, ucur, vcur, a22 );
```

- Update $A_2 = A_2 - \beta y u_{\mathrm{cur}}^T$.

```
SL_Ger( min_beta, y, ucur, a2);
```

- Let $A_{\mathrm{cur}} = A_2$.

```
SL_LAObj_swap( &a2, &acur );
```

- End of loop

```
}
```

- Free all views used.

```
SL_LAObj_free(&vcur);
SL_LAObj_free(&v);

<< more SL_LAObj_free's >>
```

Conclusion

Conclusion

- Current matrix distributions, used by HPF and many packages like ScaLAPACK are inherently application unfriendly.
- Solution: Physically Based Matrix Distribution
- Surprisingly, specifying the matrix distribution in an application-friendly fashion *simplifies* library development, through the introduction of inducing vector distributions.
- Approach unifies parallel dense, sparse iterative and sparse direct.

- New approach has led to new parallel algorithms for
  - ⋆ Implementation of BLAS
  - ⋆ LU/Cholesky/QR factorization
  - ⋆ “Q” solve
  - ⋆ Reduction to condensed form
  - ⋆ Band reduction
  - ⋆ Jacobi’s method for the dense symmetric eigenproblem
  - ⋆ ...

Work in Progress (Long Term)

PLAPACK project:

- Physically Distributed Matrix Interface
- A comprehensive implementation of linear algebra libraries:
  - ⋆ Dense
  - ⋆ Sparse iterative
  - ⋆ Sparse direct
- Languages:
  - ⋆ Object oriented: C++
  - ⋆ Object based (like MPI)
  - ⋆ New languages: ZPL extensions (Larry Snyder/Calvin Lin)

Work in Progress (Short Term)

SL_library:

- Prototype library.
- Functionality similar to ScaLAPACK.
- One one distribution allowed.
  - ⋆ Identically distributed inducing vector distributions.
  - ⋆ Uniform block size.
- Parallel routines for dense linear algebra only.
- Languages:
  - ⋆ Object based (like MPI) C and Fortran

Alpha/Beta test sites

Alpha release expected July 15.

- The PLAPACK project
- The PRISM project
- The PEIGs library
- (Your name here)

More information

- Additional session tomorrow afternoon.
    - `http://www.cs.utexas.edu/users/rvdg`
        - ⋆ Click on **The PLAPACK Project** or **The SL_library**

References

- Almadena Chtchelkanova, Carter Edwards, John Gunnels, Greg Morrow, James Overfelt, Robert A. van de Geijn, "Towards Usable and Lean Parallel Linear Algebra Libraries," submitted to *Supercomputing 96*.
- C. Edwards, P. Geng, A. Patra, and R. van de Geijn, "Parallel Matrix Decompositions: have we been doing it all wrong?" PLAPACK Working Note #1, TR-95-39, Department of Computer Sciences, University of Texas, Oct. 1995.
- Robert van de Geijn and Jerrell Watts "SUMMA: Scalable Universal Matrix Multiplication Algorithm," PLAPACK Working Note #2, Department of Computer Sciences, The Unversity of Texas, TR-95-13, April 1995. Also: LAPACK Working Note #96 , May 1995. Journal version: Concurrency: Practice and Experience, to appear.
- Brian Grayson and Robert van de Geijn "A High Performance Parallel

Strassen Implementation," PLAPACK Working Note #3, Parallel Processing Letters, to appear.

- Almadena Chtchelkanova, John Gunnels, Greg Morrow, James Overfelt, Robert A. van de Geijn, "Parallel Implementation of BLAS: General Techniques for Level 3 BLAS," PLAPACK Working Note #4, TR-95-40, Department of Computer Sciences, University of Texas, Oct. 1995. Journal version: Concurrency: Practice and Experience, submitted .