Matrix-Vector Multiplication

Multiplying a square matrix by a vector

Sequential algorithm

- Simply a series of dot products

```
Input: Matrix mat[m][n]
       Vector vec[n]
Output: out[m]

for ( i = 0; i < m; i++ )
{
    out[i] = 0;
    for ( j = 0; j < n; j++ )
        out[i] += mat[i][j] * vec[j];
}
```

  - Inner loop requires n multiplications and $n - 1$ additions
  - Complexity of inner loop is $\Theta(n)$
  - There are a total of m dot products
  - Overall complexity: $\Theta(mn)$; $\Theta(n^2)$ for a square matrix

Data decomposition options

- Domain decomposition strategy
- Three options for decomposition
  1. Rowwise block striping
     - Divide matrix elements into group of rows (same as Floyd's algorithm)
     - Each process responsible for a contiguous group of either $\lfloor m/p \rfloor$ or $\lceil m/p \rceil$ rows
  2. Columnwise block striping
     - Divide matrix elements into group of columns
     - Each process responsible for a contiguous group of either $\lfloor n/p \rfloor$ or $\lceil n/p \rceil$ columns
  3. Checkerboard block decomposition
     - Form a virtual grid
     - Matrix is divided into 2D blocks aligning with the grid
     - Let the grid have r rows and c columns
     - Each process responsible for a block of matrix containing at most $\lceil m/r \rceil$ rows and $\lceil n/c \rceil$ columns
- Storing vectors `vec` and `out`
  - Divide vector elements among processes
    - Each process is responsible for a contiguous group of either $\lfloor n/p \rfloor$ or $\lceil n/p \rceil$ elements
  - Replicate vector elements
    - Vector replication acceptable because vectors have only n elements compared to n^2 elements in matrices
    - A task storing a row or column of matrix and single element from either vector is responsible for $\Theta(n)$ elements
    - And a task storing rows and columns of the matrix and all elements of the vectors is still responsible for $\Theta(n)$

- Six possible combinations: three ways to store matrices and two ways to store vectors

Rowwise block-striped decomposition

- Partitioning through domain decomposition
- Primitive task associated with
  - Row of matrix
  - Entire vector
- Task i has row i of `mat` and the entire vector `vec`
- Each primitive task computes the inner product of an entire row with `vec` and assigns the result to one element of `out`
- Vector `out` replicated with an `all-gather` communication
- Mapping strategy decision tree
  - Agglomerate primitive tasks associated with contiguous groups of rows and assign each of these combined tasks to a single PE
  - Each PE computes a single element (or a block of elements in case of rowwise block striped matrix) of the result vector
- Assume $m = n$
  - Sequential matrix-vectormultiplication time complexity is $\Theta(n^2)$
  - For parallel algorithm, each process multiplies its portion of the matrix by the vector
  - No process is responsible for more than $\lceil n/p \rceil$ rows
  - Complexity of multiplication portion is $\Theta(n^2/p)$
  - In an efficient all-gather communication, each PE sends $\lceil \log p \rceil$ messages, total number of elements passed is $n(p-1)/p$ when p is a power of 2
  - Communication complexity: $\Theta(\log p + n)$
  - Overall complexity of parallel matrix-vector multiplication algorithm

$$\Theta(n^2/p + n + \log p)$$

  - Isoefficiency of the parallel algorithm
    - Time complexity of sequential algorithm: $\Theta(n^2)$
    - Only overhead in parallel algorithm due to `all-gather`
    - For reasonably large n, message transmission time is greater than message latency
    - Simplify communication complexity to $\Theta(n)$
    - Isoefficiency function given by

$$n^2 \geq Cpn \Rightarrow n \geq Cp$$

    - Memory utilization function: $M(n) = n^2$
    - Scalability function of parallel algorithm

$$\begin{aligned} M(Cp)/p &= C^2p^2/p \\ &= C^2p \end{aligned}$$

    - Memory utilization must grow linearly with the number of PEs
    - Algorithm is not highly scalable

- Processes must concatenate their pieces of the vector into a complete vector
- Function `MPI_Allgatherv`
  - Gather data from all tasks and deliver the combined data to all tasks
  - If the number of items to be gathered from each process is the same, a simpler function `MPI_Allgather` may be used

    ```
    int MPI_Allgatherv ( void * send_bufr, int send_count, MPI_Datatype send_type,
        void * recv_bufr, int * recv_count, int * recv_disp,
        MPI_Datatype recv_type, MPI_Comm comm );
    ```

    `send_bufr` Starting address of send buffer

    `send_count` Number of elements in send buffer

    `send_type` Data type of send buffer elements

    `recv_bufr` The only output parameter; Starting address of receive buffer to store the gathered elements

    `recv_count` Integer array containing the number of elements to be received from each process

    `recv_disp` Integer array to specify the displacement relative to `recv_bufr` at which to place the incoming data from processes

    `recv_type` Data type of receive buffer elements

    `comm` Communicator
  - The block of data sent by ith process is received at every process and placed in the ith block of `recv_bufr`
- Replicated vector I/O
  - Assume binary file for input
  - Process $p-1$ opens the file to read; read n and transmits it to other processes
  - Each process allocates memory to store vector
  - Process $p-1$ reads the vector and broadcasts it to other processes
  - Replicated vector printed by a single process; each process has a copy of the vector
- Program code
- Benchmarking
  - Sequantial algorithm complexity: $\Theta(n^2)$
  - Time needed to compute a single iteration of the loop performing inner product given by χ
  - χ determined by dividing execution time of sequential algorithm by n^2
  - Expected time for computational portion of parallel program: $\chi n\lceil n/p\rceil$
  - All-gather reduction requires each process to send $\lceil \log p\rceil$ messages
    - Latency of each message: λ
    - Total number of vector elements transmitted: $n(2^{\lceil \log p\rceil}-1)/2^{\lceil \log p\rceil}$
    - Each vector element a `double` occupying 8 bytes
    - Expected execution time for all-gather step

$$\lambda\lceil \log p\rceil + 8n\frac{2^{\lceil \log p\rceil}-1}{2^{\lceil \log p\rceil}\beta}$$

Columnwise block-striped decomposition

- Design and analysis

- Assume that each primitive task i has column i of matrix and element i of input and output vectors
- Each task i multiplies its matrix column by corresponding element of vector, giving a vector of partial results

$c_0 =$	$a_{0,0}b_0 +$	$a_{0,1}b_1 +$	$a_{0,2}b_2 +$	$a_{0,3}b_3 +$	$a_{0,4}b_4$
$c_1 =$	$a_{1,0}b_0 +$	$a_{1,1}b_1 +$	$a_{1,2}b_2 +$	$a_{1,3}b_3 +$	$a_{1,4}b_4$
$c_2 =$	$a_{2,0}b_0 +$	$a_{2,1}b_1 +$	$a_{2,2}b_2 +$	$a_{2,3}b_3 +$	$a_{2,4}b_4$
$c_3 =$	$a_{3,0}b_0 +$	$a_{3,1}b_1 +$	$a_{3,2}b_2 +$	$a_{3,3}b_3 +$	$a_{3,4}b_4$
$c_4 =$	$a_{4,0}b_0 +$	$a_{4,1}b_1 +$	$a_{4,2}b_2 +$	$a_{4,3}b_3 +$	$a_{4,4}b_4$
	Proc 0	Proc 1	Proc 2	Proc 3	Proc 4

- At the end of computation, each task needs only a single element of resultant vector c_i
- Need an *all-to-all* communication
  - Every task has the n partial results needed to add to produce final result
  - Partial result element j on task i must be transferred to task j
- Agglomeration and mapping
  - Static number of tasks
  - Regular communication pattern (all-to-all)
    - Every primitive task has identical computation and communication requirements
    - Agglomerating them into larger task with the same number of columns will help us balance the workload
  - Computation time per task is constant
  - Strategy
    - Agglomerate groups of columns (primitive tasks) into metatasks
    - Create one task per MPI process
- Complexity analysis
  - Sequential algorithm complexity: $\Theta(n^2)$
  - Assume square matrix $(m = n)$
  - Each process multiplies its portion of matrix by its block of vector
    - No process responsbile for more than $\lceil n/p \rceil$ columns of matrix or elements of vector
    - Time complexity of initial multiplication phase: $\Theta(n(n/p)) = \Theta(n^2/p)$
  - p partial vectors, each of length at most $\lceil n/p \rceil$
  - Time complexity to sum the partial vectors: $\Theta(n)$
  - Overall computational complexity of parallel algorithm: $\Theta(n^2/p)$
  - Number of steps for all-to-all exchange: $\lceil \log p \rceil$, using hypercube communication pattern
    - In each step, a process sends $n/2$ values and receives $n/2$ values
    - Total number of elements sent and received: $n\lceil \log p \rceil$
    - Communication complexity of all-gather: $\Theta(n \log p)$
  - Overall complexity: $\Theta(n^2/p + n \log p)$
- Isoefficiency analysis
  - Sequential time complexity: $\Theta(n^2)$
  - Parallel overhead limited to all-to-all operation
    - When n is large, message transmission time dominates message latency
    - Use an approach where a process sends a message to each of the other $p - 1$ processes
    - Each message contains just the elements the destination is supposed to receive from the source
    - Total number of messages is $p - 1$ but each process passes $\leq n$ elements
    - Parallel communication time: $\Theta(n)$
  - Isoefficiency function: $n^2 \geq Cpn \Rightarrow n \geq Cp$
  - Scalability function: C^2p

  - Not highly scalabale because in order to maintain a constant efficiency, memory used per processor must increase linearly with the number of processors

- Reading a columnwise block-striped matrix
  - Read a matrix stored in row-major order and distribute it among processes in columnwise block-striped fashion
  - Matrix elements not stored as a contiguous group in the file
    - Each row of matrix must be scattered among all processes
  - Single process responsible for I/O
    - Read a row of matrix into temporary buffer
    - Scatter the elements of buffer among processes

- Function `MPI_Scatterv`
  - Scatter a buffer in parts to all processes in a communicator

```
int MPI_Scatterv ( void * send_bufr, int * send_count, int * send_disp,
                   MPI_Datatype send_type, void * recv_bufr, int recv_count,
                   MPI_Datatype recv_type, int root, MPI_COMM comm );
```

    `send_bufr` Starting address of send buffer; significant only at `root`
    `send_count` Number of elements in send buffer
    `send_disp` Integer array to specify the displacement; relative to `send_bufr` from which to take the outgoing data; entry i in array corresponds to data going to process i
    `send_type` Data type of send buffer elements
    `recv_bufr` Starting address of receive buffer to store the process's portion of elements received
    `recv_count` Integer array containing the number of elements to be received
    `recv_type` Data type of receive buffer elements
    `root` Rank of sending process
    `comm` Communicator
  - Collective communication function
    - All processes in communicator participate in its execution
    - Function requires each process to have previously initialized two arrays
      1. Array to indicate the number to send to each process
      2. Array to indicate displacement of block of elements being sent

- Printing a columnwise block-striped matrix
  - Data motion opposite to reading the matrix
  - Replace scatter with gather
  - Use `MPI_Gatherv` because different processes contribute different number of elements

- Function `MPI_Gatherv`
  - Gathers into specific locations from all processes in a group

```
int MPI_Gatherv ( void * send_bufr, int send_count, MPI_Datatype send_type,
                  void * recv_bufr, int * recv_count, int * recv_disp,
                  MPI_Datatype recv_type, int root, MPI_COMM comm );
```

    `send_bufr` Starting address of send buffer
    `send_count` Number of elements in send buffer
    `send_type` Data type of send buffer elements
    `recv_bufr` Starting address of receive buffer; significant only at root

`recv_count` Integer array containing the number of elements to be received from each process; significant only at root

`recv_disp` Integer array to specify the displacement; relative to `recv_bufr` at which to place the incoming data; entry i in array corresponds to data coming from process i

`recv_type` Data type of receive buffer elements

`root` Rank of sending process

`comm` Communicator

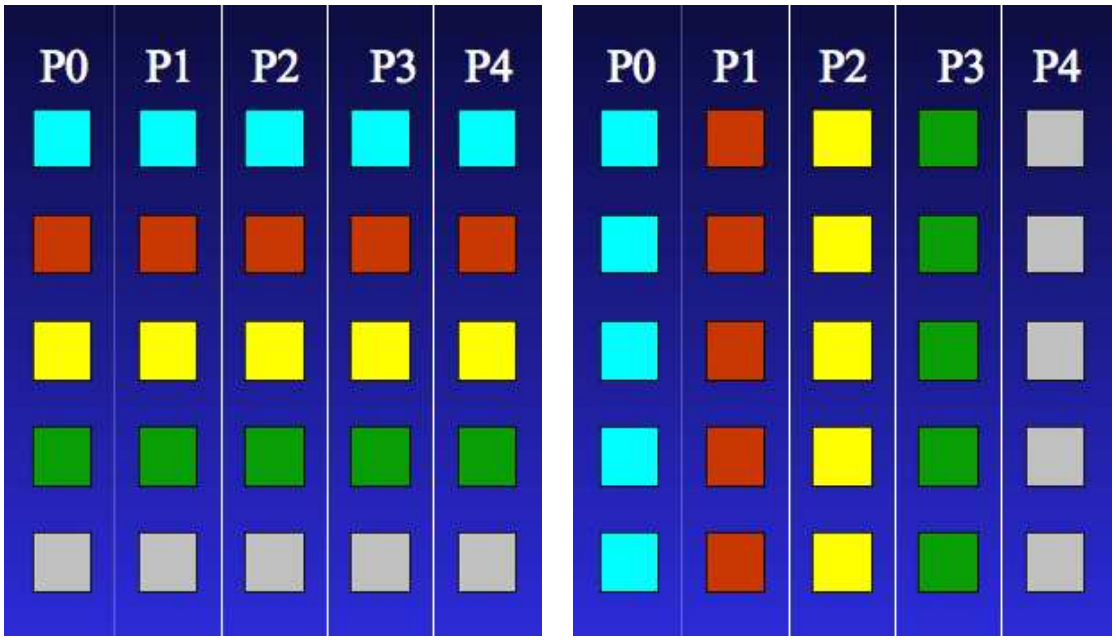
- Distributing partial results
  - A series of m inner product operations create the m element result vector c

$$\begin{aligned} c_0 &= a_{0,0}b_0 + a_{0,1}b_1 + \cdots + a_{0,n-1}b_{n-1} \\ c_1 &= a_{1,0}b_0 + a_{1,1}b_1 + \cdots + a_{1,n-1}b_{n-1} \\ &\vdots \\ c_{m-1} &= a_{m-1,0}b_0 + a_{m-1,1}b_1 + \cdots + a_{m-1,n-1}b_{n-1} \end{aligned}$$

  - Associate with each primitive task i column i of matrix A and element i of vector b
  - Multiplying each element of column by b_i yields

$$(a_{0,i}b_i), (a_{1,i}b_i), \ldots, (a_{m-1,i}b_i)$$

  - Product $a_{i,j}b_i$ is the ith term of the inner product for c_j, $0 \leq j < n$
  - After multiplication, each task needs to distribute $n-1$ result terms to other processors and collect $n-1$ terms from them
  - *all-to-all* exchange
  - After the exchange, add the m elements to produce c_i
- Function `MPI_Alltoallv`
  - Send data from all to all processors



Before After

- Each process may send different amount of data and provide displacements for the input and output data

```
int MPI_Alltoallv ( void * send_bufr, int * send_count, int * send_disp,
                    MPI_Datatype send_type, void * recv_bufr, int * recv_count,
                    int * recv_disp, MPI_Datatype recv_type, int root,
                    MPI_COMM comm );
```

  send_bufr Starting address of send buffer

  send_count Number of elements in send buffer

  send_disp Integer array to specify the displacement; relative to `send_bufr` from which to take the outgoing data; entry i in array corresponds to data going to process i

  send_type Data type of send buffer elements

  recv_bufr Starting address of receive buffer; significant only at root

  recv_count Integer array containing the number of elements to be received from each process; significant only at root

  recv_disp Integer array to specify the displacement; relative to `recv_bufr` at which to place the incoming data; entry i in array corresponds to data coming from process i

  recv_type Data type of receive buffer elements

  root Rank of sending process

  comm Communicator

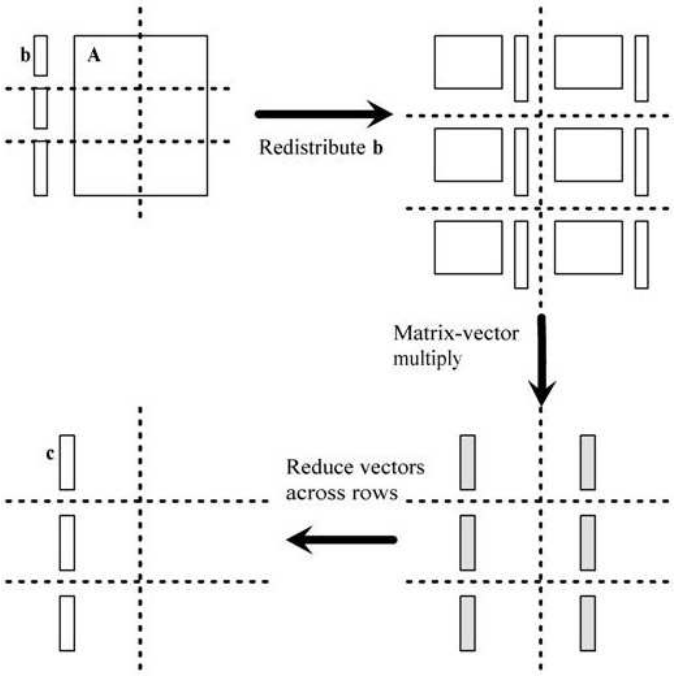
- Requires two pairs of count/displacement arrays
  1. First pair for values being sent
     - `send_count`: Number of elements
     - `send_disp`: Index of first element in the array
  2. Second pair for values being received
     - `recv_count`: Number of elements
     - `recv_disp`: Index of first element in the array

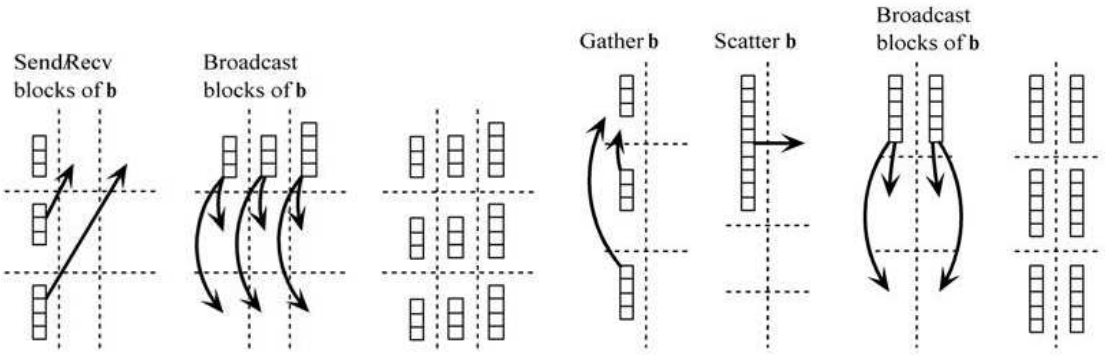
- The code
- Benchmarking
  - Time needed to compute a single iteration of the loop performing inner product given by χ
  - Expected time for computational portion of parallel program: $\chi n\lceil n/p\rceil$
  - Algorithm performs all-to-all exchange of partially computed portions of vector
  - Two methods to perform all-to-all exchange
    1. Each process sends $\lceil \log p\rceil$ messages of length $n/2$
       - Total data elements transmitted: $\lceil \log p\rceil n/2$
    2. Each process sends directly to each of the other processes the elements meant for the process
       - Each process sends $p-1$ messages
       - Total data elements transmitted: $n(p-1)/p$
  - For large n, transmission time dominates latency; second approach will be better
    - Time needed for transmission of single byte: $1/\beta$
    - Time to perform all-gather of `doubles`: $(p-1)/(\lambda + 8n/(p\beta))$
  - Total execution time

$$\chi n \left\lceil \frac{n}{p} \right\rceil + (p-1)\left(\lambda + \frac{8n}{p\beta}\right)$$

Checkerboard block decomposition

- Design and analysis
  - Associate a primitive task with each element of matrix
  - Each primitive task performs one multiply: $d_{ij} = a_{ij} \times b_j$
  - Each element of the resulting vector $c_i = \sum_{j=0}^{n-1} d_{ij}$
    - For each row i, add all the d_{ij} terms to produce c_i
  - Agglomerate primitive tasks into rectangular blocks
    - Associate a task with each block
    - Processes form a 2D grid
    - Vector distributed by blocks among processes in first column of grid
  - Three principal tasks of parallel algorithm
    1. Redistribute vector so that each task has its correct portion
       - Move vector from processes in first row to processes in first column
       (a) p is a square
          - First column/first row processes send/receive portions of vector
       (b) p is not a square
          - Gather vector on process (0,0)
          - Process (0,0) broadcasts to processes in first row
       - First row processes scatter vector within columns
    2. Perform matrix-vector multiplication with the portion assigned to task
    3. Perform sum-reduction on the task's portion of resulting vector





p is a square number p is not a square number

- Analyzing complexity
  - p is a square number
    - If grid is $1 \times p$, columnwise block-striped
    - If grid is $p \times 1$, rowwise block-striped
  - Each process does its share of computation: block of size $\lceil n/\sqrt{p} \rceil \times \lceil n/\sqrt{p} \rceil \Rightarrow \Theta(n^2 p)$
  - Redistribute vectors: $\Theta(n/\sqrt{p} + (\log p)(n/\sqrt{p})) = \Theta(n \log p/\sqrt{p})$
  - Reduction of partial results vectors: $\Theta(n \log p/\sqrt{p})$
  - Overall parallel complexity
$$\Theta(n^2/p + n \log p/\sqrt{p})$$
- Isoefficiency analysis
  - Sequential complexity: $\Theta(n^2)$
  - Parallel communication complexity: $\Theta(n \log p/\sqrt{p})$
  - Isoefficiency function
$$n^2 \geq Cn\sqrt{p} \log p \Rightarrow n \geq C\sqrt{p} \log p$$
  - Since $M(n) = n^2$, scalability function is:
$$\begin{aligned} \frac{M(C\sqrt{p}\log p)}{p} &= \frac{C^2 p \log^2 p}{p} \\ &= C^2 \log^2 p \end{aligned}$$
  - Checkerboard is more scalable than the other two implementations

- Creating communicators
  - Communicator provides the environment for message-passing among processes
    - Default communicator is `MPI_COMM_WORLD`
  - Four collective communication operations involving subsets of processes in matrix-vector multiplication with checkerboard block decomposition
    1. Processes in first column of grid gather vector b when p is not square
    2. Processes in first row of grid scatter vector b when p is not square
    3. Each first row process broadcasts its block of vector b to other processes in the same column of grid
    4. Each row of processes in grid performs an independent sum-reduction, yielding result vector in the first column
  - Want processes in a virtual 2D grid
    - Create a custom communicator
    - Collective communications involve all processes in a communicator
    - Need to do broadcasts and reductions among subsets of processes

    - Create communicators for processes in the same row or same column
  - Communicator contains
    - Process group
    - Context
    - Attributes
      - Topology – allows us to address processes in another way compared to the rank
      - MPI supports two topologies – Cartesian/grid and graph

- Function `MPI_Dims_create`
  - Create a division of processors in a Cartesian grid
  - Prefer to create a virtual mesh of processes that is as close to square as possible for scalability

```
int MPI_Dims_create ( int num_nodes, int num_dims, int * dims );
```

IN num_nodes Number of nodes/processes in the grid

IN num_dims Number of Cartesian dimensions

INOUT dims Integer array of size `num_dims` specifying the number of nodes in each dimension; a value of 0 indicates that the function should fill in a suitable value

  - Usage example

```
int p;
int sz[2];
...
MPI_Comm_size ( MPI_COMM_WORLD, &p );
...
sz[0] = sz[1] = 0;
MPI_Dims_create ( p, 2, sz );
```

- Function `MPI_Cart_create`
  - Make a new communicator to which topology information has been attached
  - Done after determining the size of each dimension of virtual grid of processes

```
int MPI_Cart_create ( MPI_Comm comm_old, int num_dims, int * dims,
        int * periods, int reorder, MPI_Comm * comm_cart );
```

IN comm_old Input communicator

IN num_dims Number of Cartesian dimensions

IN dims Integer array of size `num_dims` specifying the number of processes in each dimension

IN periods Logical array of size `num_dims` specifying whether the grid is periodic (true) or not (false) in each dimension

    - Grid is periodic if communication wraps around the edges

IN reorder Ranking may be reordered (true) or not (false); currently being ignored by MPI functions

OUT comm_cart Communicator with new Cartesian topology

  - Usage example
    - Making new communicator from `MPI_COMM_WORLD`

```
MPI_Comm  cart_comm;      // Cartesian topology communicator
int       periodic[2];    // Message wraparound flag
int       sz[2];          // Size of each dimension
...
periodic[0] = periodic[1] = 0;      // Grid is not periodic
sz[0] = sz[1] = 0;                  // Initialize each dimension size to 0
MPI_Dims_create ( p, 2, sz );
MPI_Cart_create ( MPI_COMM_WORLD, 2, sz, periodic, 1, &cart_comm );
```

- Reading a checkerboard matrix
  - Single process (process 0) responsible to read the matrix and distribute contents to appropriate processes
  - Distribution pattern similar to the method for columnwise striping
  - Each row scattered among a subset of processes, within the same row of virtual grid
  - Matrix read by process 0
    - Each time a row is read, send it to appropriate process in the appropriate row of process grid
    - Receiving process gets the matrix row, and scatters it among the processes in its row of the process grid
    - Achieved by `MPI_Cart_rank`, `MPI_Cart_coords`, and `MPI_Comm_split`
- Function `MPI_Cart_rank`
  - Process 0 needs to know its rank to send a matrix row to the first process in appropriate row in the process grid
  - Determine process rank in communicator given Cartesian location

```
int MPI_Cart_rank (
    IN  MPI_Comm   comm,    // Communicator with Cartesian structure
    IN  int      * coords,  // Process' grid location
    OUT int      * rank     // Output rank of the process
    );
```

    `coords` Integer arrays of size `num_dims` of Cartesian topology associated with `comm` specifying the Cartesian coordinates of the process

  - Usage example
    - Assume r rows in virtual process grid and m rows in matrix
    - Row i of the input matrix mapped to row of process grid specified by `BLOCK_OWNER(i, r, m)`

```
int     dest_coord[2];     // Coordinates of process receiving row
int     dest_rank;         // Rank of process receiving row
int     grid_rank;         // Rank of process in virtual grid
int     i;
...
for ( i = 0; i < m; i++ )
{
    dest_coord[0] = BLOCK_OWNER(i, r, m);
    dest_coord[1] = 0;
    MPI_Cart_rank ( grid_comm, dest_coord, &dest_rank );

    if ( grid_rank == 0 )
    {
        // Read matrix row i
        ...
        // Send matrix row i to process dest_rank
        ...
    }
    else
        if ( grid-rank == dest_rank )
        {
            // Receive matrix row i from process 0 in virtual grid
        }
}
```

- Function `MPI_Cart_coords`
  - Determine process coordinates in Cartesian topology given its rank in the group

- Allows process to allocate correct amount of memory for its portion of matrix and vector
- Allows process 0 (the one reading the matrix) to find out its coordinates
  - Knowing its coordinates enables process 0 to avoid sending itself a message if it happens to be the first process in a row of process grid

```
int MPI_Cart_coords (
    IN  MPI_Comm comm,     // Communicator with Cartesian structure
    IN  int      rank,     // Rank of process within group of comm
    IN  int      max_dims, // Dimensions in virtual grid
    OUT int *    coords    // Coordinates of specified process in virtual grid
    );
```

`max_dims` Length of vector `coords` in the calling program

`coords` Integer array of size `max_dims` to return the Cartesian coordinates of specified process

- Usage example

```
int        grid_id;
MPI_Comm   grid_comm;
int        grid_coords[2];
...
MPI_Cart_coords ( grid_comm, grid_id, 2, grid_coords );
```

- Function `MPI_Comm_split`
  - Partitions the processes of a communicator into one or more subgroups
    - Scatter is a collective operation
    - The input row is scattered amond only the processes in a single row of the process grid by dividing the Cartesian communicator into separate communicators for every row in process grid
  - Constructs a communicator for each subgroup, based on colors and keys
  - Allows processes in each subgroup to perform their own collective communications
  - Needed for columnwise scatter and rowwise reduce

```
int MPI_Comm_split (
    IN  MPI_Comm   comm,    // Old communicator
    IN  int        color,   // Partition number
    IN  int        key,     // Control of rank assignment
    OUT MPI_Comm * new_comm // New communicator
    );
```

`color` Control of subset assignment as a nonnegative integer; processes with the same color are in the same new communicator

`key` Ranking order of processes in new communicator

`new_comm` New communicator shared by processes in same partition

  - We determined the process coordinates by using `MPI_Cart_coords`
  - Usage example
    - Group together processes in the same row by using the value of `grid_coords[0]` as `color`
    - Rank processes according to their column by using `grid_coords[1]` as ranking order determinant

```
MPI_Comm       grid_comm;         // 2D process grid
MPI_Comm       row_comm;          // Processes in same row
int            grid_coords[2];    // Location of process in grid
...
MPI_Comm_split ( grid_comm, grid_coords[0], grid_coords[1], &row_comm );
```

  - This function can also be used to divide the Cartesian communicator into separate communicators for every column of the grid

Benchmarking

- Analytical model, considering p as a square number
- Time to perform a single iteration of the loop to compute inner product: χ
- Each process responsible for a block of matrix with size $\left\lceil \frac{n}{\sqrt{p}} \right\rceil \times \left\lceil \frac{n}{\sqrt{p}} \right\rceil$
- Estimated computation time of parallel program: $\chi \left\lceil \frac{n}{\sqrt{p}} \right\rceil \times \left\lceil \frac{n}{\sqrt{p}} \right\rceil$
- Redistribute vector b
  - Processes in first column of grid pass their blocks of vector to processes in the first row of grid
  - Each process responsible for at most $\left\lceil \frac{n}{\sqrt{p}} \right\rceil$ elements of vector
  - Time to send/receive: $\lambda + 8 \left\lceil \frac{n}{\sqrt{p}} \right\rceil / \beta$
  - Each process then broadcasts its block to other processes in the column
  - Time to broadcast: $\log \sqrt{p} \left(\lambda + 8 \left\lceil \frac{n}{\sqrt{p}} \right\rceil / \beta \right)$
- Reduce partial results
  - Ignore the time for addition as the communications time dominates
  - Communications time same as that needed for broadcast: $\log \sqrt{p} \left(\lambda + 8 \left\lceil \frac{n}{\sqrt{p}} \right\rceil / \beta \right)$