Lecture 9: Dense Matrices and Decomposition

CSCE 569 Parallel Computing

Department of Computer Science and Engineering
Yonghong Yan
yanyh@cse.sc.edu
http://cse.sc.edu/~yanyh
Review: Parallel Algorithm Design and Decomposition

• Introduction to Parallel Algorithms
  – Tasks and Decomposition
  – Processes and Mapping

• Decomposition Techniques
  – Recursive Decomposition
  – Data Decomposition
  – Exploratory Decomposition
  – Hybrid Decomposition

• Characteristics of Tasks and Interactions
  – Task Generation, Granularity, and Context
  – Characteristics of Task Interactions.
Decomposition, Tasks, and Dependency Graphs

- Decompose work into tasks that can be executed concurrently
- Decomposition could be in many different ways.
- Tasks may be of same, different, or even indeterminate sizes.
- Task dependency graph:
  - node = task
  - edge = control dependence, output-input dependency
  - No dependency == parallelism
Degree of Concurrency

• Definition: the number of tasks that can be executed in parallel
• May change over program execution

• Metrics
  – Maximum degree of concurrency
    • Maximum number of concurrent tasks at any point during execution.
  – Average degree of concurrency
    • The average number of tasks that can be processed in parallel over the execution of the program
    • Speedup: \( \frac{\text{serial\_execution\_time}}{\text{parallel\_execution\_time}} \)

• Inverse relationship of degree of concurrency and task granularity
  – Task granularity \(\uparrow\) (less tasks), degree of concurrency \(\downarrow\)
  – Task granularity \(\downarrow\) (more tasks), degree of concurrency \(\uparrow\)
Critical Path Length

- A directed path: a sequence of tasks that must be serialized
  - Executed one after another

- Critical path:
  - The longest weighted path throughout the graph

- Critical path length: shortest time in which the program can be finished
  - Lower bound on parallel execution time
Critical Path Length and Degree of Concurrency

Database query task dependency graph

Questions:
What are the tasks on the critical path for each dependency graph?
What is the shortest parallel execution time?
How many processors are needed to achieve the minimum time?
What is the maximum degree of concurrency?
What is the average parallelism (average degree of concurrency)?
Total amount of work/(critical path length)
2.33 (63/27) and 1.88 (64/34)
Finer task granularity $\Rightarrow$ more overhead of task interactions
- Overhead as a ratio of useful work of a task
- Example: sparse matrix-vector product interaction graph

Assumptions:
- each dot $(A[i][j]*b[j])$ takes unit time to process
- each communication (edge) causes an overhead of a unit time
- If node 0 is a task: communication = 3; computation = 4
- If nodes 0, 4, and 5 are a task: communication = 5; computation = 15
- coarser-grain decomposition $\Rightarrow$ smaller communication/computation ratio (3/4 vs 5/15)
Processes and Mapping

A good mapping must minimize parallel execution time by:

• Mapping independent tasks to different processes
  – Maximize concurrency

• Tasks on critical path have high priority of being assigned to processes

• Minimizing interaction between processes
  – mapping tasks with dense interactions to the same process.

• Difficulty: these criteria often conflict with each other
  – E.g. No decomposition, i.e. one task, minimizes interaction but no speedup at all!
Recursive Decomposition: Min

Finding the minimum in a vector using divide-and-conquer

**procedure** SERIAL_MIN (A, n)

\[
\text{min} = A[0];
\]

\[
\text{for } i := 1 \text{ to } n - 1 \text{ do }
\]

\[
\text{if } (A[i] < \text{min}) \text{ min } := A[i];
\]

\[
\text{return } \text{min};
\]

**procedure** RECURSIVE_MIN (A, n)

\[
\text{if } (n = 1) \text{ then } \text{min } := A[0];
\]

\[
\text{else }
\]

\[
\text{lmin } := \text{RECURSIVE_MIN } (A, \text{n/2});
\]

\[
\text{rmin } := \text{RECURSIVE_MIN } (&(A[\text{n/2}]), \text{n} - \text{n/2});
\]

\[
\text{if } (\text{lmin} < \text{rmin}) \text{ then } \text{min } := \text{lmin};
\]

\[
\text{else } \text{min } := \text{rmin};
\]

\[
\text{return } \text{min};
\]

Applicable to other associative operations, e.g. sum, AND ... Known as reduction operation
Data Decomposition
-- The most commonly used approach

• **Steps:**
  1. **Identify the data on which computations are performed.**
  2. **Partition this data across various tasks.**
     • **Partitioning induces a decomposition of the problem, i.e. computation is partitioned**

• **Data can be partitioned in various ways**
  – **Critical for parallel performance**

• **Decomposition based on**
  – **output data**
  – **input data**
  – **input + output data**
  – **intermediate data**
Output Data Decomposition: Example

Count the frequency of item sets in database transactions

<table>
<thead>
<tr>
<th>Database Transactions</th>
<th>Itemsets</th>
<th>Itemset Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>A, B, C, E, G, H</td>
<td>A, B, C</td>
<td>1</td>
</tr>
<tr>
<td>B, D, E, F, K, L</td>
<td>D, E</td>
<td>3</td>
</tr>
<tr>
<td>A, B, F, H, L</td>
<td>C, F, G</td>
<td>0</td>
</tr>
<tr>
<td>D, E, F, H</td>
<td>A, B</td>
<td>2</td>
</tr>
<tr>
<td>F, G, H, K</td>
<td>C, D</td>
<td>0</td>
</tr>
<tr>
<td>A, B, F, K, L</td>
<td>D, K</td>
<td>1</td>
</tr>
<tr>
<td>B, C, D, G, H, L</td>
<td>D, K</td>
<td>2</td>
</tr>
<tr>
<td>G, H, L</td>
<td>C, D, K</td>
<td>0</td>
</tr>
<tr>
<td>D, E, F, K, L</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F, G, H, L</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Decompose the item sets to count
  - each task computes total count for each of its item sets
  - append total counts for item sets to produce total count result
Input Data Partitioning: Example

Count the frequency of item sets in database transactions

- Partition computation by partitioning the set of transactions
  - a task computes a local count for each item set for its transactions

- Sum local count vectors for item sets to produce total count vector
Dense matrix algorithms

- Dense linear algebra and BLAS
- Image processing/stencil
- Iterative methods
The Motifs (formerly “Dwarfs”) from “The Berkeley View” (Asanovic et al.) form key computational patterns.
Dense linear algebra

- Software library solving linear system

- BLAS (Basic Linear Algebra Subprogram)
  - Vector, matrix vector, matrix matrix

- Linear Systems: Ax=b

- Least Squares: choose x to minimize ||Ax-b||^2
  - Overdetermined or underdetermined
  - Unconstrained, constrained, weighted

- Eigenvalues and vectors of Symmetric Matrices
  - Standard (Ax = λx), Generalized (Ax=λBx)

- Eigenvalues and vectors of Unsymmetric matrices
  - Eigenvalues, Schur form, eigenvectors, invariant subspaces
  - Standard, Generalized

- Singular Values and vectors (SVD)
  - Standard, Generalized

- Different matrix structures
  - Real, complex; Symmetric, Hermitian, positive definite; dense, triangular, banded ...

- Level of detail
  - Simple Driver
  - Expert Drivers with error bounds, extra-precision, other options
  - Lower level routines (“apply certain kind of orthogonal transformation”, matmul...)
BLAS (Basic Linear Algebra Subprogram)

- **BLAS 1, 1973-1977**
  - 15 operations (mostly) on vectors (1-d array)
  - “AXPY” (y = α·x + y), dot product, scale (x = α·x)
  - Up to 4 versions of each (S/D/C/Z), 46 routines, 3300 LOC
  - **Why BLAS 1?** They do O(n^1) ops on O(n^1) data: AXPY
    - 2n flops on 3n read/writes
    - Computational intensity = (2n)/(3n) = 2/3

```c
void axpy_base(int N, REAL Y[], REAL X[], REAL a) {
    int i;
    for (i = 0; i < N; ++i)
        Y[i] += a * X[i];
}
```
BLAS 2

- **BLAS 2, 1984-1986**
  - 25 operations (mostly) on matrix/vector pairs
  - "GEMV": \( y = \alpha \cdot A \cdot x + \beta \cdot x \), "GER": \( A = A + \alpha \cdot x \cdot y^T \), \( x = T^{-1} \cdot x \)
  - Up to 4 versions of each (S/D/C/Z), 66 routines, 18K LOC

- **Why BLAS 2?** They do \( O(n^2) \) ops on \( O(n^2) \) data
  - Computational intensity still just \(~(2n^2)/(n^2) = 2\)

\[
A \cdot x = \begin{bmatrix}
  a_{11} & a_{12} & \cdots & a_{1n} \\
  a_{21} & a_{22} & \cdots & a_{2n} \\
  \vdots & \vdots & \ddots & \vdots \\
  a_{m1} & a_{m2} & \cdots & a_{mn}
\end{bmatrix} \begin{bmatrix}
  x_1 \\
  x_2 \\
  \vdots \\
  x_n
\end{bmatrix} = \begin{bmatrix}
  a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \\
  a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \\
  \vdots \\
  a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n
\end{bmatrix}
\]

```c
void matvec_base(int M, int N, REAL Y[], REAL A[][N], REAL B[]) {
    int i, j;
    for (i = 0; i < M; i++) {
        REAL temp = 0.0;
        for (j = 0; j < N; j++) {
            temp += A[i][j] * B[j];
        }
        Y[i] = temp;
    }
}
```
**BLAS 3**

- **BLAS 3, 1987-1988**
  - 9 operations (mostly) on matrix/matrix pairs
  - "GEMM": \( C = \alpha \cdot A \cdot B + \beta \cdot C, \ C = \alpha \cdot A \cdot AT + \beta \cdot C, \ B = T^{-1} \cdot B \)
  - Up to 4 versions of each (S/D/C/Z), 30 routines, 10K LOC
  - Why BLAS 3? They do \( O(n^3) \) ops on \( O(n^2) \) data
    - Computational intensity \( (2n^3)/(4n^2) = n/2 \) = big at last!
    - Good for machines with caches, deep mem hierarchy

\[
A[M][K] \times B[K][N] = C[M][N]
\]
Decomposition for AXPY, Matrix Vector, and Matrix Multiplication
BLAS 1: AXPY

- \( y = \alpha \cdot x + y \)
  - \( x \) and \( y \) are vectors of size \( N \)
  - In C, \( x[N], y[N] \)
  - \( \alpha \) is scalar

- Decomposition is simple
  - \( N \) iterations (\( N \) elements of \( X \) and \( Y \)) are distributed among threads
  - 1:1 mapping between iteration and element of \( X \) and \( Y \)
  - \( X \) and \( Y \) are shared

```c
void axpy_openmp(int N, REAL Y[], REAL X[], REAL a) {
    int i;
    #pragma omp parallel for
    for (i = 0; i < N; ++i)
        Y[i] += a * X[i];
}
```
BLAS 2: Matrix Vector Multiplication

- \( y = A \cdot x \)
  - \( A[M][N], x[N], y[N] \)
- Row-wise decomposition

```c
void matvec_omp_parallel(int M, int N, REAL Y[], REAL A[][N], REAL B[], int num_tasks) {
    #pragma omp parallel num_threads(num_tasks)
    {
        int tid = omp_get_thread_num();
        int Mt = N/num_tasks;
        int i_start = tid*Mt;
        int i, j;
        for (i = i_start; i < i_start + Mt; i++) {
            REAL temp = 0.0;
            for (j = 0; j < N; j++) {
                temp += A[i][j] * B[j];
            }
            Y[i] = temp;
        }
    }
}

void matvec_omp_parallel_for(int M, int N, REAL Y[], REAL A[][N], REAL B[], int num_tasks) {
    int i, j;
    #pragma omp parallel for private(i,j) num_threads(num_tasks)
    for (i = 0; i < M; i++) {
        REAL temp = 0.0;
        for (j = 0; j < N; j++) {
            temp += A[i][j] * B[j];
        }
        Y[i] = temp;
    }
}
BLAS 3: Dense Matrix Multiplication


- Base
- Base_1: column major order of access
- row1D_dist
- column1D_dist
- rowcol2D_dist

- Decomposition is to calculate Mt and Nt
void mm_omp_parallel_for_row1D(int N, int K, int M, REAL * A, REAL * B, REAL * C) {
    int i, j, w;
    #pragma omp parallel private (i, j, w) num_threads(4)
    #pragma omp for
    for (i=0; i<N; i++)
        for (j=0; j<M; j++) {
            REAL temp = 0.0;
            for (w=0; w<K; w++)
                temp += A[i*K+w]*B[w*M+j];
            C[i*M+j] = temp;
        }
}
void mm_omp_parallel_for_col1D(int N, int K, int M, REAL * A, REAL * B, REAL * C) {
    int i, j, w;
    #pragma omp parallel private (i, j, w) num_threads(4)
    for (i=0; i<N; i++)
        #pragma omp for
        for (j=0; j<M; j++) {
            REAL temp = 0.0;
            for (w=0; w<K; w++)
                temp += A[i*K+w] * B[w*M+j];
            C[i*M+j] = temp;
        }
}
BLAS 3: Dense Matrix Multiplication

- Row/Column-based 2-D

```c
void mm_omp_parallel_for_rowcol2D(int N, int K, int M, REAL * A, REAL * B, int i, j, w;
#pragma omp parallel for private (i, j, w) num_threads(4)
for (i=0; i<N; i++)
#pragma omp parallel for shared(i) private (j, w) num_threads(4)
for (j=0; j<M; j++) {
    REAL temp = 0.0;
    for (w=0; w<K; w++)
        temp += A[i*K+w]*B[w*M+j];
    C[i*M+j] = temp;
}
```

Need nested parallelism

```c
export OMP_NESTED=true
```
Dense matrix algorithms

• Dense linear algebra and BLAS
• Image processing/stencil
• Iterative methods
**What is Multimedia**

- Multimedia is a combination of text, graphic, sound, animation, and video that is delivered interactively to the user by electronic or digitally manipulated means.

<table>
<thead>
<tr>
<th>Medium</th>
<th>Elements</th>
<th>Time-dependence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text</td>
<td>Printable characters</td>
<td>No</td>
</tr>
<tr>
<td>Graphic</td>
<td>Vectors, regions</td>
<td>No</td>
</tr>
<tr>
<td>Image</td>
<td>Pixels</td>
<td>No</td>
</tr>
<tr>
<td>Audio</td>
<td>Sound, Volume</td>
<td>Yes</td>
</tr>
<tr>
<td>Video</td>
<td>Raster images, graphics</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Videos contains frame (images)

---

Examples of individual content forms combined in multimedia:

- Text
- Audio
- Still Images
- Animation
- Video Footage
- Interactivity

https://en.wikipedia.org/wiki/Multimedia
Image Format and Processing

• Pixels
  – Images are matrix of pixels

• Binary images
  – Each pixel is either 0 or 1
Image Format and Processing

- **Pixels**
  - Images are matrix of pixels

- **Grayscale images**
  - Each pixel value normally range from 0 (black) to 255 (white)
  - 8 bits per pixel
Image Format and Processing

- **Pixels**
  - Images are matrix of pixels

- **Color images**
  - Each pixel has three/four values (4 bits or 8 bits each) each representing a color scale

<table>
<thead>
<tr>
<th>Sample Length:</th>
<th>4</th>
<th>4</th>
<th>4</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel Membership:</td>
<td>Alpha</td>
<td>Red</td>
<td>Green</td>
<td>Blue</td>
</tr>
<tr>
<td>Bit Number:</td>
<td>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sample Length:</th>
<th>8</th>
<th>8</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel Membership:</td>
<td>Blue</td>
<td>Green</td>
<td>Red</td>
</tr>
<tr>
<td>Bit Number:</td>
<td>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sample Length:</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel Membership:</td>
<td>Alpha</td>
</tr>
<tr>
<td>Bit Number:</td>
<td>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
</tr>
</tbody>
</table>
An image histogram is a graph of pixel intensity (on the $x$-axis) versus number of pixels (on the $y$-axis). The $x$-axis has all available gray levels, and the $y$-axis indicates the number of pixels that have a particular gray-level value.

https://www.allaboutcircuits.com/technical-articles/image-histogram-characteristics-machine-learning-image-processing/
Histograms of Monochrome Image

```c
void calculate_histogram(image, histogram, length, width)
{
    int    length, width;
    short **image;
    unsigned long histogram[];

    for(i=0; i<length; i++)
        for(j=0; j<width; j++)
            histogram[k] = histogram[k] + 1;
}

Figure 5. Histogram of a dark image. Image by Sneha H.L.

Figure 6. Histogram of a bright image. Image by Sneha H.L.
```

http://homepages.inf.ed.ac.uk/rbf/BOOKS/PHILLIPS/cips2edsr/HIST.C
Histogram of Color Images

- Image density

```
for( int y = 0; y < image.rows; y++ ) {
    for( int x = 0; x < image.cols; x++ ) {
        for( int c = 0; c < 3; c++ ) {
            new_image.at<Vec3b>(y,x)[c] =
                saturate_cast<uchar>( alpha*( image.at<Vec3b>(y,x)[c] ) + beta );
        }
    }
}
```

https://docs.opencv.org/3.4.0/d3/dc1/tutorial_basic_linear_transform.html
OpenMP Parallelization of Histogram

• Decomposition based on output (pixel values, 0 - 255)
  – Each thread searches the whole image to only count those pixels that have the value it should count for
  • E.g. with 4 threads: 0-63 for thread 0, 64-127 for thread 1, ...

• Decomposition based on the input (image)
  – Each thread search part of the image to count all the pixels and store the partial histogram locally
  – Add up all the partial histogram
Image Filtering

- Changing pixel values by doing a convolution between a kernel (filter) and an image.

```c
for(i=1; i<rows-1; i++){
    if( (i%10) == 0 ) printf("%d ", i);
    for(j=1; j<cols-1; j++){
        sum = 0;
        for(a=-1; a<2; a++)
            for(b=-1; b<2; b++){
                sum = sum +
                    the_image[i+a][j+b] *
                    filter[a+1][b+1];
            }
        sum = sum/d;
        if(sum < 0) sum = 0;
        if(sum > max) sum = max;
        out_image[i][j] = sum;
    }
} /* ends loop over j */
} /* ends loop over i */
```
Image Filtering: The magic of the filter matrix

- [http://lodev.org/cgtutor/filtering.html](http://lodev.org/cgtutor/filtering.html)

It is the basic of convolution neural network
Convolution Neural Network for Object Detection

- Pooling: sample-based discretization process

http://cs231n.github.io/convolutional-networks/
OpenMP Parallelization of Image Filtering

- Decomposition according to the input image
- Since input and output images are separate, it is straightforward
  - Could be row1D, col1D, rowcol2D
- False-sharing for writing boundary of output images

```c
for(i=1; i<rows-1; i++){
    if( (i%10) == 0 ) printf("%d ", i);
    for(j=1; j<cols-1; j++){  
        sum = 0;
        for(a=-1; a<2; a++){
            for(b=-1; b<2; b++){
                sum = sum +
                the_image[i+a][j+b] *
                filter[a+1][b+1];
            }
        }
        sum = sum/d;
        if(sum < 0) sum = 0;
        if(sum > max) sum = max;
        out_image[i][j] = sum;
    }
}  /* ends loop over j */
}  /* ends loop over i */
```
Dense matrix algorithms

- Dense linear algebra and BLAS
- Image processing/stencil
- Iterative methods
Iterative Methods

• Iterative methods can be expressed in the general form:

\[ x^{(k)} = F(x^{(k-1)}) \]

Hopefully: \( x^{(k)} \to s \) (solution of my problem)

• Wide variety of computational science problem
  – CFD, molecular dynamics, weather/climate forecast, cosmology,

• Will it converge? How rapidly?
Iterative Stencil Applications

Loop until some condition is true

Perform computation which involves communicating with N,E,W,S neighbors of a point (5 point stencil)

[Convergence test?]

Stencil is similar as image filtering/convolution
• Assignment 2 and 3:

```c
while ((k <= mits) && (error > tol)) {
    error = 0.0;

    /* Copy new solution into old */
    for (i = 0; i < n; i++)
        for (j = 0; j < m; j++)
            uold[i][j] = u[i][j];

    for (i = 1; i < (n - 1); i++)
        for (j = 1; j < (m - 1); j++) {
            resid = (ax * (uold[i - 1][j] + uold[i + 1][j]) +
                    ay * (uold[i][j - 1] + uold[i][j + 1]) +
                    b * uold[i][j] - f[i][j]) / b;
            //printf("i: %d, j: %d, resid: %f\n", i, j, resid);

            u[i][j] = uold[i][j] - omega * resid;
            error = error + resid * resid;
        }

    /* Error check */
    if (k % 500 == 0)
        printf("Finished %ld iteration with error: %g\n", k, error);
    error = sqrt(error) / (n * m);

    k = k + 1;
} /* End iteration loop */
printf("Total Number of Iterations: %ld\n", k);
printf("Residual: %.15g\n", error);
```
Jacobi

- An iterative method for approximating the solution to a system of linear equations.
- $Ax=b$ where the $i$th equation is

$$a_{i,1}x_1 + a_{i,1}x_1 + \cdots + a_{i,n}x_n = b_i$$

$$x_i = \frac{1}{a_{i,i}} \left[ b_i - \sum_{j \neq i} a_{i,j}x_j \right]$$

- $a$’s and $b$’s are known, **want to solve for $x$’s**
OpenMP Parallelization of Jacobi

- Similar as image filtering
  - Enclosed by the `while` to be iterative
- `omp parallel` for outer `while` loop
- `omp for` for inner `for` loops
- `single and reduction` are needed

```c
while ((k <= mits) && (error > tol)) {
  error = 0.0;

  /* Copy new solution into old */
  for (i = 0; i < n; i++)
    for (j = 0; j < m; j++)
      uold[i][j] = u[i][j];

  for (i = 1; i < (n - 1); i++)
    for (j = 1; j < (m - 1); j++)
      {
        resid = (ax * (uold[i - 1][j] + uold[i + 1][j]) +
                 ay * (uold[i][j - 1] + uold[i][j + 1]) +
                 b * uold[i][j] - f[i][j]) / b;
        //printf("i: %d, j: %d, resid: %f\n", i, j, resid);
        u[i][j] = uold[i][j] - omega * resid;
        error = error + resid * resid;
      }

  /* Error check */
  if (k % 500 == 0)
    printf("Finished %ld iteration with error: %g\n", k, error);
  error = sqrt(error) / (n * m);

  k = k + 1;
}
/* End iteration loop */
printf("Total Number of Iterations: %ld\n", k);
printf("Residual: %.15g\n", error);
```
Ghost Cell Exchange

• For assignment 3:
Background:

C multidimensional array
Vector/Matrix and Array in C

• C has row-major storage for multiple dimensional array
  – $A[2,2]$ is followed by $A[2,3]$

• 3-dimensional array
  – $B[3][100][100]$

• Think it as recursive definition
  – $A[4][10][32]$
Column Major

Fortran is column major

Row-major order

Column-major order
Array Layout: Why We Care?

1. Makes a big difference for access speed

- For performance, set up code to go in row major order in C
  - Caching: each read from memory will bring other adjacent elements to the cache line

- (Bad) Example: 4 vs 16 accesses
  - `matmul_base_1`

```c
for i = 1 to n
    for j = 1 to n
        A[j][i] = value
```
2. Affect decomposition and data movement

- Decomposition may create submatrices that are in non-contiguous memory locations, e.g. A3 and B1
- Submatrices in contiguous memory location of 2-D row major matrix
  - A single-row submatrix, e.g. A2
  - A submatrix formed with adjacent rows with full column length, e.g. A1
Array Layout: Why We Care?

2. Affect decomposition and submatrix

- Row or column wise distribution of 2-D row-major array
- # of data movement to exchange data between T0 and T1
  - Row-wise: one memory copy by each
  - Column-wise: 16 copies each
Array and pointers in C

• In C, an array is a pointer + dimensionality
  – They are literally the same in binary, i.e. pointer to the first element, referenced as base address

• Cast and assignment from array to pointe, int A[M][N]
  • A, &A[0][0], and A[0] have the same value, i.e. the pointer to the first element of the array

• Cast a pointer to an array
  – int *ap; int (*A)[N] = (int(*))[N]ap; A[i][j] ....

• Address calculation for array references
  – Address of A[i][j] = A + (i*N+j)*sizeof (int)