A Tiny Bit of Linear Algebra

(i.e., just enough for P3)
Linear algebra

- Many scientific phenomena can be modeled as matrix operations
  - Differential equations, mesh simulations, view transforms, etc.
  - Many of these phenomena involve solving linear equations
  - Doing this requires linear algebra and the manipulation of large matrices
- Very efficient on vector processors (including GPUs)
  - Data decomposition and SIMD parallelism
  - Popular packages: BLAS, LINPACK, LAPACK, ATLAS

\[
\begin{bmatrix}
1 & 0 & 1 & 1 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 1 \\
\end{bmatrix}
\cdot
\begin{bmatrix}
\ln(l_1) \\
\ln(l_2) \\
\ln(l_3) \\
\ln(l_4) \\
\ln(l_5) \\
\ln(l_6) \\
\ln(l_7) \\
\end{bmatrix}
=
\begin{bmatrix}
\ln(r_{1,3,4}) \\
\ln(r_{1,3,5}) \\
\ln(r_{2,6}) \\
\ln(r_{2,7}) \\
\end{bmatrix}
\]
Dense vs. sparse matrices

- A **sparse** matrix is one in which most elements are zero
  - Could lead to more load imbalances
  - Can be stored more efficiently, allowing for larger matrices
  - **Dense** matrix operations no longer work
  - It is a challenge to make sparse operations as efficient as dense operations

\[
\begin{pmatrix}
11 & 22 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 33 & 44 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 55 & 66 & 77 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 88 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 99 & 0
\end{pmatrix}
\]
HPL benchmark

- **HPL**: LINPACK-based dense linear algebra benchmark
  - Generates a linear system of equations “Ax = b” - O(n^2)
    - Chooses b such that x (answer vector) values are known
  - Distributes dense matrix A in block-cyclic pattern - O(n)
  - LU factorization - O(n^3) (similar to Gaussian elimination)
  - Backward substitution to solve system - O(n^2)
  - Error calculation to verify correctness - O(n)
  - Compare max sustained FLOPS (floating-point operations per section)
    - Usually significantly less than theoretical machine peak (Rmax vs Rpeak)
  - Serves as proxy app for target workloads (similar characteristics)
    - Used to rank world's fastest systems on the Top500 list twice each year
  - Compiled on cluster
    - Located in /shared/apps/hpl-2.1/bin/Linux_PII_CBLAS
P3 (OpenMP)

- Similar to HPL benchmark
  1) Random generation of linear system (x should be all 1’s)
  2) Gaussian elimination
  3) Backwards substitution (row- or column-oriented)

Non-random example

Original system (Ax = b)

\[
\begin{align*}
3x + 2y - z &= 1 \\
2x - 2y + 4z &= -2 \\
-x + \frac{1}{2}y - z &= 0
\end{align*}
\]

Augmented matrix [A | b]

\[
\begin{pmatrix}
3.0 & 2.0 & -1.0 & 1.0 \\
2.0 & -2.0 & 4.0 & -2.0 \\
-1.0 & 0.5 & -1.0 & 0.0
\end{pmatrix}
\]

Upper triangular system

\[
\begin{pmatrix}
3.0 & 2.0 & -1.0 & 1.0 \\
0.0 & -3.3 & 4.7 & -2.7 \\
0.0 & 0.0 & 0.3 & -0.6
\end{pmatrix}
\]

Solved system

\[
\begin{pmatrix}
1.0 & 0.0 & 0.0 & 1.0 \\
0.0 & 1.0 & 0.0 & -2.0 \\
0.0 & 0.0 & 1.0 & -2.0
\end{pmatrix}
\]
Matrix representation

• 2D dense matrices in C
  - Often stored in 1D arrays w/ access via array index arithmetic
  - Trace data access patterns to determine dependencies
  - Your goals: 1) **analyze**, 2) **parallelize** (w/ OpenMP), and 3) **evaluate**
  - Example (matrix multiplication):

```c
void multiply_matrices(int *A, int *B, int *R, int n)
{
    int i, j, k;
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            R[i*n+j] = 0;
            for (k = 0; k < n; k++) {
                R[i*n+j] += A[i*n+k] * B[k*n+j];
            }
        }
    }
}
```
Matrix representation

- **2D dense matrices in C**
  - Often stored in 1D arrays w/ access via array index arithmetic
  - Trace data access patterns to determine dependencies
  - Your goals: 1) **analyze**, 2) **parallelize** (w/ OpenMP), and 3) **evaluate**
  - Example (matrix multiplication):

```c
void multiply_matrices(int *A, int *B, int *R, int n)
{
    int i, j, k;
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            R[i*n+j] = 0;
            for (k = 0; k < n; k++) {
                R[i*n+j] += A[i*n+k] * B[k*n+j];
            }
        }
    }
}
```

read as \( R[i,j] \)
Matrix access patterns

```c
void multiply_matrices(int *A, int *B, int *R, int n) {
    int i, j, k;
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            R[i*n+j] = 0;
            for (k = 0; k < n; k++) {
                R[i*n+j] += A[i*n+k] * B[k*n+j];
            }
        }
    }
}
```