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
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

\[
\begin{pmatrix}
1 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 3 & 3 & 4 & 4 & 0 & 0 & 0 \\
0 & 0 & 5 & 5 & 6 & 6 & 7 & 7 \\
0 & 0 & 0 & 0 & 0 & 8 & 8 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 9
\end{pmatrix}
\]
HPL benchmark

- **HPL**: LINPACK-based dense linear algebra benchmark
  - Generates a linear system of equations “$Ax = b$”
    - Chooses $b$ such that $x$ (answer vector) values are known
  - Distributes dense matrix $A$ in block-cyclic pattern
  - LU factorization (similar to Gaussian elimination)
  - Backward substitution to solve system
  - Error calculation to verify correctness
  - Compare max sustained FLOPS (floating-point operations per section)
    - Usually significantly less than theoretical machine peak ($R_{max}$ vs $R_{peak}$)
  - 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]

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

Upper triangular system

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

Backward substitution

\[
\begin{bmatrix}
1.0 & 0.0 & 0.0 & | & 1.0 \\
0.0 & 1.0 & 0.0 & | & -2.0 \\
0.0 & 0.0 & 1.0 & | & -2.0
\end{bmatrix}
\]

Solved system
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 \(R[i,j]\)
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];
            }
        }
    }
}

\[ R = AB \]