Matrices in HPC

(Just enough for P2)
Matrices

- 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
  - Small, computation-intensive loop nests called **kernels**
  - 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**/normal matrix operations no longer work
  - It is a challenge to make sparse operations as efficient as dense operations

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

- 2D dense row-major matrices in C/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++) {
            int 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 row-major matrices in C/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];
            }
        }
    }
}
```

\[ R = AB \]
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;
            #pragma omp parallel for default(none) \ 
            shared(A,B,R,n,i,j) private(k)
            for (k = 0; k < n; k++) {
                R[i*n+j] += A[i*n+k] * B[k*n+j];
            }
        }
    }
}
```

Incorrect b/c of loop-carried dependencies

```
== SERIAL ==
Nthreads= 1  TEST=pass  MULT=  0.7825

== PARALLEL ==
Nthreads= 1  TEST=pass  MULT=  1.0443
Nthreads= 2  TEST=FAIL  MULT=  2.3410
Nthreads= 4  TEST=FAIL  MULT=  3.7809
Nthreads= 8  TEST=FAIL  MULT=  6.5935
Nthreads=16  TEST=FAIL  MULT=  9.2631
Nthreads=32  TEST=FAIL  MULT= 17.3672
```
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++) {
            int tmp = 0;
            #pragma omp parallel for default(none) reduction(+:tmp) shared(A,B,R,n,i,j) private(k)
            for (k = 0; k < n; k++) {
                tmp += A[i*n+k] * B[k*n+j];
            }
            R[i*n+j] = tmp;
        }
    }
}

== SERIAL ==
Nthreads= 1  TEST=pass  MULT=  0.7308

== PARALLEL ==
Nthreads= 1  TEST=pass  MULT=  1.0136
Nthreads= 2  TEST=pass  MULT=  2.2375
Nthreads= 4  TEST=pass  MULT=  4.1329
Nthreads= 8  TEST=pass  MULT=  7.6924
Nthreads=16  TEST=pass  MULT=  8.8584
Nthreads=32  TEST=pass  MULT= 16.4827

Correct w/ no speedup
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++) {
        #pragma omp parallel for default(none) shared(A,B,R,n,i) private(j,k)
        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];
            }
        }
    }
}
```

Correct w/ speedup

```
== SERIAL ==
Nthreads= 1  TEST=pass  MULT=  0.7669

== PARALLEL ==
Nthreads= 1  TEST=pass  MULT=  0.7816
Nthreads= 2  TEST=pass  MULT=  0.3983
Nthreads= 4  TEST=pass  MULT=  0.1991
Nthreads= 8  TEST=pass  MULT=  0.1032
Nthreads=16  TEST=pass  MULT=  0.1060
Nthreads=32  TEST=pass  MULT=  0.0785
```
Matrix access patterns

```c
void multiply_matrices(int *A, int *B, int *R, int n)
{
    int i, j, k;
    # pragma omp parallel for default(none) shared(A,B,R,n) private(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];
            }
        }
    }
}
```

== SERIAL ==
Nthreads= 1  TEST=pass  MULT=  0.7861

== PARALLEL ==
Nthreads= 1  TEST=pass  MULT=  0.7921
Nthreads= 2  TEST=pass  MULT=  0.3978
Nthreads= 4  TEST=pass  MULT=  0.1957
Nthreads= 8  TEST=pass  MULT=  0.1043
Nthreads=16  TEST=pass  MULT=  0.0908
Nthreads=32  TEST=pass  MULT=  0.0640

Correct w/ best speedup
(fewest fork/join operations)
HPL benchmark

- **HPL**: LINPACK-based dense linear algebra benchmark
  - Generation of a linear system of equations “Ax = b” - \( O(n^2) \)
    - Choose ‘b’ such that ‘x’ (answer vector) values are known
    - Distribute dense matrix ‘A’ in block-cyclic pattern
  - LU factorization - \( O(n^3) \)
  - Backward substitution to solve system - \( O(n^2) \)
  - Error calculation to verify correctness - \( O(n) \)
  - Calculate max sustained **FLOPS** (*floating-point operations per second*)
    - Usually significantly less than theoretical machine peak (\( R_{\text{max}} \) vs \( R_{\text{peak}} \))
    - 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`
• Similar to HPL benchmark

1) Random generation of linear system (x should be all 1’s)
2) Gaussian elimination (similar to LU factorization)
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 \([A | b]\)

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

Upper triangular system

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

Gaussian elimination

Solved system

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

Backward substitution