Matrices in HPC

(just enough for P3)
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
**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
  - 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
**P3 (OpenMP)**

- 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) | Augmented matrix [A | b] |
|--------------------------|-------------------------|
| $3x + 2y - z = 1$        | $3.0 \quad 2.0 \quad -1.0 \quad 1.0$ |
| $2x - 2y + 4z = -2$     | $2.0 \quad -2.0 \quad 4.0 \quad -2.0$ |
| $-x + \frac{1}{2}y - z = 0$ | $-1.0 \quad 0.5 \quad -1.0 \quad 0.0$ |

<table>
<thead>
<tr>
<th>Upper triangular system</th>
<th>Solved system</th>
</tr>
</thead>
<tbody>
<tr>
<td>$3.0 \quad 2.0 -1.0 \quad 1.0$</td>
<td>$1.0 \quad 0.0 \quad 0.0 \quad 1.0$</td>
</tr>
<tr>
<td>$0.0 \quad -3.3 \quad 4.7 \quad -2.7$</td>
<td>$0.0 \quad 1.0 \quad 0.0 \quad -2.0$</td>
</tr>
<tr>
<td>$0.0 \quad 0.0 \quad 0.3 \quad -0.6$</td>
<td>$0.0 \quad 0.0 \quad 1.0 \quad -2.0$</td>
</tr>
</tbody>
</table>
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];
            }
        }
    }
}
```
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];
            }
        }
    }
}
```

\[ R = A \times B \]
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=  0.1289

**== PARALLEL ==**

Nthreads= 1  TEST=pass  MULT=  0.4256
Nthreads= 2  TEST=FAIL  MULT=  0.5004
Nthreads= 4  TEST=FAIL  MULT=  0.5088
Nthreads= 8  TEST=FAIL  MULT=  0.6106
Nthreads=16  TEST=FAIL  MULT=  0.9350
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++) {
            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;
        }
    }
}
```

Correct w/ no speedup

<table>
<thead>
<tr>
<th>Nthreads</th>
<th>TEST</th>
<th>MULT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>pass</td>
<td>0.1078</td>
</tr>
<tr>
<td>2</td>
<td>pass</td>
<td>0.2820</td>
</tr>
<tr>
<td>4</td>
<td>pass</td>
<td>0.4502</td>
</tr>
<tr>
<td>8</td>
<td>pass</td>
<td>0.6040</td>
</tr>
<tr>
<td>16</td>
<td>pass</td>
<td>0.8390</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Nthreads</th>
<th>TEST</th>
<th>MULT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>pass</td>
<td>0.1290</td>
</tr>
<tr>
<td>2</td>
<td>pass</td>
<td>0.1348</td>
</tr>
<tr>
<td>4</td>
<td>pass</td>
<td>0.0735</td>
</tr>
<tr>
<td>8</td>
<td>pass</td>
<td>0.0410</td>
</tr>
<tr>
<td>16</td>
<td>pass</td>
<td>0.0380</td>
</tr>
</tbody>
</table>
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];
            }
        }
    }
}
```

---

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

== SERIAL ==
Nthreads= 1  TEST=pass  MULT=  0.1279

== PARALLEL ==
Nthreads= 1  TEST=pass  MULT=  0.2530
Nthreads= 2  TEST=pass  MULT=  0.1274
Nthreads= 4  TEST=pass  MULT=  0.0698
Nthreads= 8  TEST=pass  MULT=  0.0390
Nthreads=16  TEST=pass  MULT=  0.0337