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
Aside: 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

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

- 2D dense row-major matrices in C/C++
  - Often stored in 1D arrays w/ access via array index arithmetic
  - “Row-major” order: outer dimension specified first
  - Address of \((i,j)\)th element is \((\text{base} + \text{size}(i \times \text{cols} + j))\)
Matrix representation

- Parallelism w/ 2D matrices
  - 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

- Parallelism w/ 2D matrices
  - 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

<table>
<thead>
<tr>
<th>Nthreads</th>
<th>TEST</th>
<th>MULT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>pass</td>
<td>0.7825</td>
</tr>
<tr>
<td>2</td>
<td>FAIL</td>
<td>1.0443</td>
</tr>
<tr>
<td>4</td>
<td>FAIL</td>
<td>2.3410</td>
</tr>
<tr>
<td>8</td>
<td>FAIL</td>
<td>3.7809</td>
</tr>
<tr>
<td>16</td>
<td>FAIL</td>
<td>6.5935</td>
</tr>
<tr>
<td>32</td>
<td>FAIL</td>
<td>9.2631</td>
</tr>
<tr>
<td>64</td>
<td>FAIL</td>
<td>17.3672</td>
</tr>
</tbody>
</table>

== SERIAL ==

== PARALLEL ==
```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

== 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
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>0.7669</td>
</tr>
<tr>
<td>2</td>
<td>pass</td>
<td>0.3983</td>
</tr>
<tr>
<td>4</td>
<td>pass</td>
<td>0.1991</td>
</tr>
<tr>
<td>8</td>
<td>pass</td>
<td>0.1032</td>
</tr>
<tr>
<td>16</td>
<td>pass</td>
<td>0.1060</td>
</tr>
<tr>
<td>32</td>
<td>pass</td>
<td>0.0785</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)

<table>
<thead>
<tr>
<th>Nthreads</th>
<th>TEST</th>
<th>MULT</th>
</tr>
</thead>
<tbody>
<tr>
<td>SERIAL</td>
<td></td>
<td>0.7861</td>
</tr>
<tr>
<td>1</td>
<td>pass</td>
<td></td>
</tr>
<tr>
<td>PARALLEL</td>
<td></td>
<td>0.7921</td>
</tr>
<tr>
<td>1</td>
<td>pass</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>pass</td>
<td>0.3978</td>
</tr>
<tr>
<td>4</td>
<td>pass</td>
<td>0.1957</td>
</tr>
<tr>
<td>8</td>
<td>pass</td>
<td>0.1043</td>
</tr>
<tr>
<td>16</td>
<td>pass</td>
<td>0.0908</td>
</tr>
<tr>
<td>32</td>
<td>pass</td>
<td>0.0640</td>
</tr>
</tbody>
</table>
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]

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

Upper triangular system

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

Backward substitution

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

Solved system