# LU Decomposition

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \Rightarrow \begin{bmatrix} l_{11} & l_{12} & l_{13} \\ l_{21} & l_{22} & l_{23} \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ u_{21} & u_{22} & u_{23} \\ u_{31} & u_{32} & u_{33} \end{bmatrix}$$

Group 2
Devin Piner, John Pace, Kevin Macfarlane, Scott Clay

# Goals

-Implement algorithms for LU decomposition for 3 different computer platforms

- Shared Memory SIMD:
  - openMP
- Distributed-Memory MIMD:
  - MPI
- SIMT using GPU:
  - CUDA

-Sequential code used as a baseline

-Analyze performance differences, overhead, scalability, etc.

-Use discovered optimization methods to develop an improved program

# LU Decomposition

```
for k=0 to n-1:  #iterate through n rows
        max=0
        maxIndex=k
        # find the pivot row by the maximum leading element in column k
        for i=k to n:
                temp=abs(a[i][k])
                if (temp>max):
                        max=temp
                        maxIndex=i
        if (maxIndex != k):
                swap(a[k], a[i])   # swap rows
        # find the multiplicative inverse of the leading element in row k
        pivot = -1/a[k][k]
        for i=k+1 to n:
                temp = pivot * a[i][k]  # multiplier coefficient for row i
                L[i][k]=(-(temp))       # store multiplier coefficient for row i in L
                // perform row reduction, Ri = Ri + temp(Rk)
                for j=k to n:
                        a[i][j]=a[i][j] + temp*a[k][j]
```

$$a = \begin{bmatrix} 0.5 & 1.5 & 2 \\ 1.5 & 2 & 2.5 \\ 2 & 2.5 & 4.5 \end{bmatrix} \quad swap(R3, R1) \rightarrow \begin{bmatrix} 2 & 2.5 & 4.5 \\ 1.5 & 2 & 2.5 \\ 0.5 & 1.5 & 2 \end{bmatrix}$$

Iteration k=0:

$$pivot = \frac{-1}{a[0][0]} = -.5$$

$c = pivot * a[1][0] = -.75$ # Store in L[1][0]

$R_2 \rightarrow R_2 + c(R_1)$

$c = pivot * a[2][0] = -.25$ # Store in L[2][0]

$R_3 \rightarrow R_3 + c(R_1)$

swap(L[2][0], L[1][0])

$$a = \begin{bmatrix} 2 & 2.5 & 4.5 \\ 0 & 0.125 & -0.875 \\ 0 & 0.875 & 0.875 \end{bmatrix} \quad L = \begin{bmatrix} 1 & 0 & 0 \\ 0.25 & 1 & 0 \\ 0.75 & \square & 1 \end{bmatrix}$$

Iteration k=1:

$$swap(R2, R1) \rightarrow \begin{bmatrix} 2 & 2.5 & 4.5 \\ 0 & 0.875 & 0.875 \\ 0 & 0.125 & -0.875 \end{bmatrix}$$

$$pivot = \frac{-1}{a[1][1]} = -1.14$$

$c = pivot * a[2][1] = -.14$ # Store in L[2][1]

$R_3 \rightarrow R_3 + c(R_2)$

$$U = \begin{bmatrix} 2 & 2.5 & 4.5 \\ 0 & 0.875 & 0.875 \\ 0 & 0 & -1 \end{bmatrix} \quad L = \begin{bmatrix} 1 & 0 & 0 \\ 0.25 & 1 & 0 \\ 0.75 & -0.14 & 1 \end{bmatrix}$$

# LU Decomposition

-U is the result of the row reduction from Gaussian Elimination

-The elements of L are the coefficient multipliers computed during the Gaussian Elimination

-Big O complexity is ($n^3$)

-To reconstruct matrix *a*, multiply LU

- This gives the correct values but the rows will not be ordered
- Requires a one-dimensional pivot array or permutation matrix to recompose

# Sequential

-C++ implementation of the previous algorithm

-Receives:

- + Matrix *a*, containing the input matrix
- + Matrix *L*, initialized to 0's with 1's on the diagonal

- Changes matrix *a* through each iteration, turning it into upper matrix *U*

-Stores coefficient multipliers in matrix *L* as they are found

-L[1][0] and L[n-1][0] must be swapped to produce the correct output matrix. Reason is unknown. Will require more testing with different input matrices.

```
LUdecomp(float **a, float **L, int n){
        float pivot,max,temp;
        int indmax,i,j,k,lk,master;
        float *tmp = new float[n];
        for (k = 0 ; k < n-1 ; k++) {
                max = 0.0;
                indmax = k;
                for (i = k ; i < n ; i++) {
                        temp = abs(a[k][i]);
                        if (temp > max) {
                                max = temp;
                                indmax = i;
                        }
                }
                if (indmax != k) {
                        for (j = k; j < n; j++) {
                                temp = a[j][indmax];
                                a[j][indmax] = a[j][k];
                                a[j][k] = temp;
                        }
                }
                pivot = -1.0/a[k][k];
                for (i = k+1 ; i < n ; i++){
                        tmp[i]= pivot*a[k][i];
                        L[k][i]=((-1.0)*tmp[i]);
                        for(j=k; j<n; j++)
                                a[j][i] = a[j][i] + tmp[i]*a[j][k];
        }
                if (indmax != k && k==0){
                temp=L[k][indmax];
        L[k][indmax]=L[k][k+1];
                        L[k][k+1]=temp;
                }
        }
}
```

# Sequential Program Output

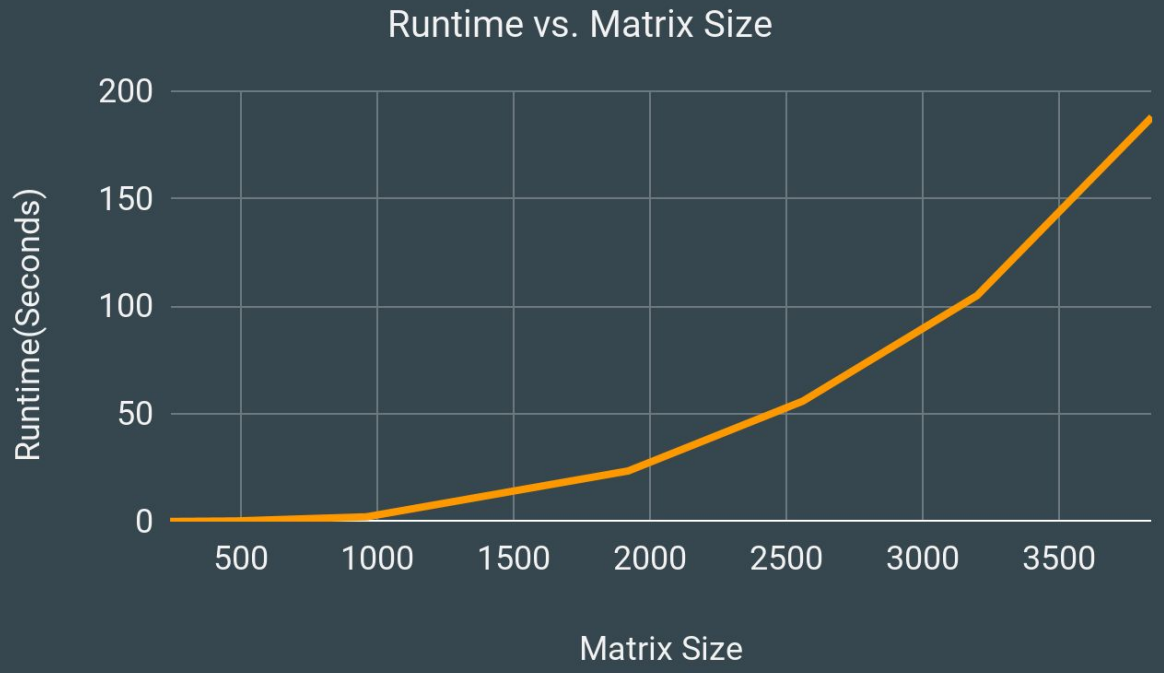
```
[john.pace@heracles lab4]$ g++ ludecom-seq.cpp -o ludecomp-seq
[john.pace@heracles lab4]$ sbatch c_slurm.sh 3 1
Submitted batch job 22954
[john.pace@heracles lab4]$ cat slurm_output.22954
Input matrix:
Row 1:  0.50    1.50    2.00
Row 2:  1.50    2.00    2.50
Row 3:  2.00    2.50    4.50
Upper matrix:
Row 1:  2.00    2.50    4.50
Row 2:  0.00    0.88    0.88
Row 3:  0.00    0.00    -1.00
Lower matrix:
Row 1:  1.00    0.00    0.00
Row 2:  0.25    1.00    0.00
Row 3:  0.75    0.14    1.00
LU Decomposition runs in 0.00 seconds
```

# Sequential Program Output

```
[john.pace@heracles lab4]$ sbatch c_slurm.sh 9 1
Submitted batch job 22955
[john.pace@heracles lab4]$ cat slurm_output.22955
Input matrix:
Row 1:  0.50    1.50    2.00    2.50    3.00    3.50    4.00    4.50    5.00
Row 2:  1.50    2.00    2.50    3.00    3.50    4.00    4.50    5.00    5.50
Row 3:  2.00    2.50    4.50    3.50    4.00    4.50    5.00    5.50    6.00
Row 4:  2.50    3.00    3.50    8.00    4.50    5.00    5.50    6.00    6.50
Row 5:  3.00    3.50    4.00    4.50    12.50   5.50    6.00    6.50    7.00
Row 6:  3.50    4.00    4.50    5.00    5.50    18.00   6.50    7.00    7.50
Row 7:  4.00    4.50    5.00    5.50    6.00    6.50    24.50   7.50    8.00
Row 8:  4.50    5.00    5.50    6.00    6.50    7.00    7.50    32.00   8.50
Row 9:  5.00    5.50    6.00    6.50    7.00    7.50    8.00    8.50    40.50
Upper matrix:
Row 1:  5.00    5.50    6.00    6.50    7.00    7.50    8.00    8.50    40.50
Row 2:  0.00    0.95    1.40    1.85    2.30    2.75    3.20    3.65    0.95
Row 3:  0.00    0.00    1.66    0.32    0.47    0.63    0.79    0.95    -10.50
Row 4:  0.00    0.00    0.00    4.24    0.36    0.48    0.60    0.71    -13.17
Row 5:  0.00    0.00    0.00    0.00    7.77    0.36    0.45    0.54    -16.24
Row 6:  0.00    0.00    0.00    0.00    0.00    12.26   0.33    0.39    -19.63
Row 7:  0.00    0.00    0.00    0.00    0.00    0.00    17.71   0.25    -23.31
Row 8:  0.00    0.00    0.00    0.00    0.00    0.00    0.00    24.13   -27.27
Row 9:  0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    -0.86
Lower matrix:
Row 1:  1.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
Row 2:  0.10    1.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
Row 3:  0.40    0.32    1.00    0.00    0.00    0.00    0.00    0.00    0.00
Row 4:  0.50    0.26    0.08    1.00    0.00    0.00    0.00    0.00    0.00
Row 5:  0.60    0.21    0.06    0.04    1.00    0.00    0.00    0.00    0.00
Row 6:  0.70    0.16    0.05    0.03    0.03    1.00    0.00    0.00    0.00
Row 7:  0.80    0.11    0.03    0.02    0.02    0.01    1.00    0.00    0.00
Row 8:  0.90    0.05    0.02    0.01    0.01    0.01    0.01    1.00    0.00
Row 9:  0.30    0.37    0.11    0.08    0.06    0.05    0.04    0.04    1.00
LU Decomposition runs in 0.00 seconds
```

# Sequential Runtimes

| Matrix Size | Runtime |
|---|---|
| 240 | 0.04 |
| 480 | 0.27 |
| 960 | 2.19 |
| 1920 | 23.53 |
| 2560 | 55.97 |
| 3200 | 105.14 |
| 3840 | 188.04 |



# OpenMP V1

- Sequential outer-loop
  - Looping over the rows
  - Parallel body

- Attempting Manual Self-scheduling
  - Variables to keep track of which element to process next
  - While-loops instead of static- or dynamic-scheduled for-loops
  - Critical sections to assign elements

https://github.com/devpin95/ParallelSystems/blob/master/Final/LUdecom_omp.cpp#L127

# Variables

```
float pivot;                // the value to multiply the row by
float pivots[n-1];          // array of pivots for each row
int k;                      // the column we are looking at

float globalmax;            // the global max for pivoting
int globalmaxindex;         // the row index of the global max
float privatemax;           // the max that the thread has encountered
int private_row_index;      // the row index being looked at
int private_column_index;   // the column index being looked at
float privatetemp;          // the value in the column that needs to be checked as the max

int row_pointer;            // The pointer to the next row that needs to be processed
int column_pointer;         // The pointer to the next column that needs to be processed
int tid;                    // thread ID
```

# Sequential Outer For-Loop

```
for ( k = 0; k < n; ++k ) {
    #pragma omp parallel
        shared(L, U, globalmax, globalmaxindex, row_pointer, column_pointer, pivots)
        firstprivate(n, k)
        private(pivot, privatemax, private_row_index, tid)
    {
```

# Initializing Swapping Variables

```
#pragma omp single
{
    // Set the row pointer to the first column that needs to be looked at
    // this will always be the same as the column variable k.
    // set the column pointer to k if we need to pivot. Set it to k because all columns before k will
    // be 0 and we dont need to swap them.
    // reset the global max to 0.0 as well.
    row_pointer = k;
    column_pointer = k;
    globalmax = 0.0;
}
```

# Finding the Column Max

```
while ( row_pointer < n ) {
    #pragma omp critical
    {
        private_row_index = row_pointer;
        ++row_pointer;
    }

    // check if all of the rows have already been checked
    // if they have then the thread needs to skip checking the index
    if ( private_row_index >= n ) break;

    // otherwise, we need to get the absolute value of the elements at U[privateindex][k]
    privatemax = abs(U[private_row_index][k]);

    // now we need to compare it to the global value
    #pragma omp critical
    {
        if ( globalmax < privatemax ) {
            globalmax = privatemax;
            globalmaxindex = private_row_index;
        }
    }

// check the row pointer at the end so that the thread doesnt go back through
// we dont need to worry about a critical section because we also check it at the beginning
}
```

# Swapping Rows

```
// the index is not the row we are on so we need to swap it to the current row
while (column_pointer < n) {
    #pragma omp critical
    {
        private_column_index = column_pointer;
        ++column_pointer;
    }

    // check if all of the rows have already been checked
    // if they have then the thread needs to skip checking the index
    if (private_column_index >= n) break;

    // swap the current row with the row with the max value
    float top = U[k][private_column_index];
    float bottom = U[globalmaxindex][private_column_index];
    U[k][private_column_index] = bottom;
    U[globalmaxindex][private_column_index] = top;
}
```

# Placing pivots in L

```
#pragma omp single
    row_pointer = k + 1;
}

while ( row_pointer < n ) {
    #pragma omp critical
    {
        private_row_index = row_pointer;
        ++row_pointer;
    }

    if ( private_row_index >= n ) break;

    float sign = 1.0;
    if ( U[private_row_index][k] < 0.0 ) {
        sign = -sign;
    }

    L[private_row_index][k] = sign * (U[private_row_index][k] / U[k][k]);
}
```

# Row Reductions

```
#pragma omp single
{
    column_pointer = k;
    row_pointer = k + 1;
}
```

```
while ( row_pointer < n ) {
    #pragma omp critical
    {
        private_column_index = column_pointer;
        private_row_index = row_pointer;

        ++column_pointer;

        if ( column_pointer >= n ) {
            column_pointer = k;
            ++row_pointer;
        }
    }

    if ( private_row_index >= n ) break;

    pivot = L[private_row_index][k];
    U[private_row_index][private_column_index] -= (pivot * U[k][private_column_index]);
}
```

# Problems

- BAD!
  - Sequential outer loop
  - Critical section overhead
    - More work done in critical section than in parallel
    - Bottleneck

- Improvements
  - Parallelize outer for-loop
  - For-loops instead of while-loops
    - Spread tasks across threads

| Matrix Size | Runtime (seconds) |
|---|---|
| 240 | 2.22 |
| 480 | 16.56 |
| 960 | 132.65 |
| 1920 | 1200+ (20 min) |

# OpenMP V2

-Modified version of cge-omp1.cpp

-Uses shared matrices *a* and *L*

-Dynamic Scheduling

-After finding the pivot and swapping rows in the k loop, L and U are found with:

```
#pragma omp parallel for shared(a,L) firstprivate(pivot,n,k) private(i,j,temp) schedule(dynamic)
for (i = k+1 ; i < n; i++)
{
        temp = pivot*a[i][k];
        L[i][k]=((-1.0)*temp);
        for (j = k ; j < n ; j++)
                a[i][j] = a[i][j] + temp*a[k][j];
}
```

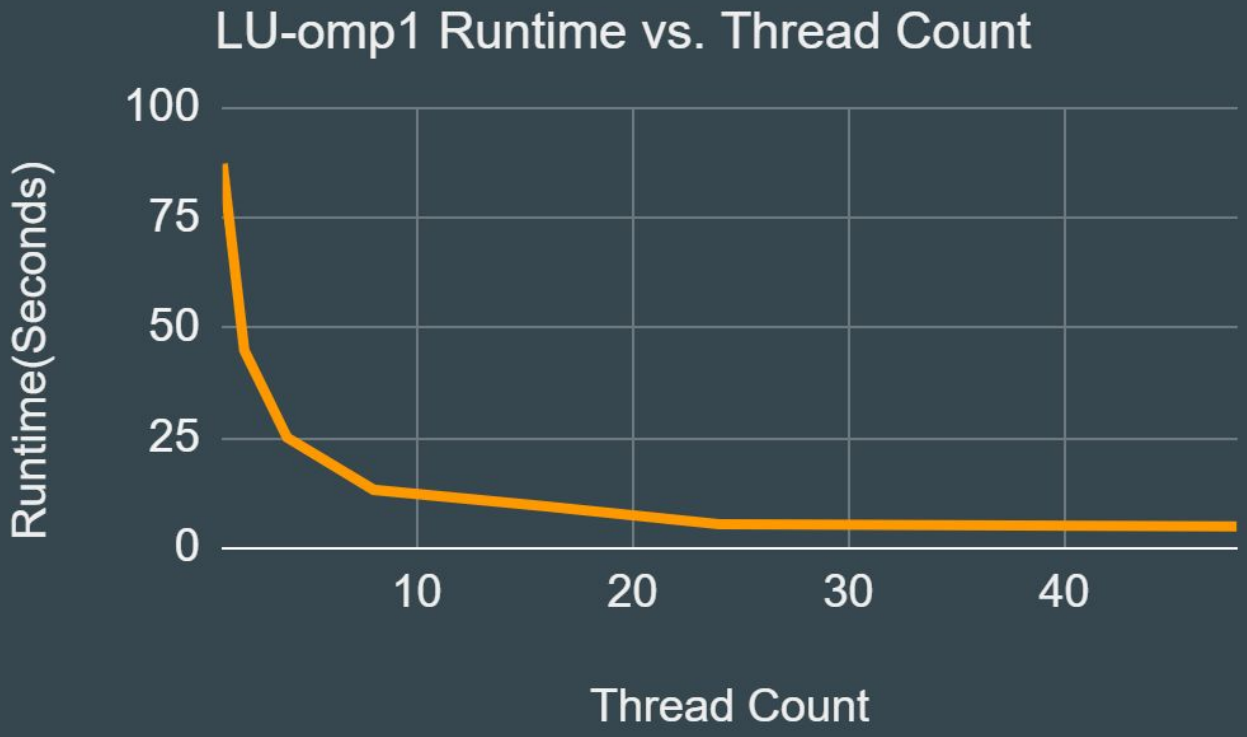
# OpenMP V2

```
[john.pace@heracles lab4]$ g++ -fopenmp lu-omp1.cpp -o lu-omp
You have new mail in /var/spool/mail/john.pace
[john.pace@heracles lab4]$ sbatch openmp_slurm.sh 9 1 1
Submitted batch job 22851
[john.pace@heracles lab4]$ cat slurm_output.22851
 thread affinity/proc_bind =
close
The input matrix
Row 1:  0.50    1.50    2.00    2.50    3.00    3.50    4.00    4.50    5.00
Row 2:  1.50    2.00    2.50    3.00    3.50    4.00    4.50    5.00    5.50
Row 3:  2.00    2.50    4.50    3.50    4.00    4.50    5.00    5.50    6.00
Row 4:  2.50    3.00    3.50    8.00    4.50    5.00    5.50    6.00    6.50
Row 5:  3.00    3.50    4.00    4.50    12.50   5.50    6.00    6.50    7.00
Row 6:  3.50    4.00    4.50    5.00    5.50    18.00   6.50    7.00    7.50
Row 7:  4.00    4.50    5.00    5.50    6.00    6.50    24.50   7.50    8.00
Row 8:  4.50    5.00    5.50    6.00    6.50    7.00    7.50    32.00   8.50
Row 9:  5.00    5.50    6.00    6.50    7.00    7.50    8.00    8.50    40.50
Upper matrix:
Row 1:  5.00    5.50    6.00    6.50    7.00    7.50    8.00    8.50    40.50
Row 2:  0.00    0.95    1.40    1.85    2.30    2.75    3.20    3.65    0.95
Row 3:  0.00    0.00    1.66    0.32    0.47    0.63    0.79    0.95    -10.50
Row 4:  0.00    0.00    0.00    4.24    0.36    0.48    0.60    0.71    -13.17
Row 5:  0.00    0.00    0.00    0.00    7.77    0.36    0.45    0.54    -16.24
Row 6:  0.00    0.00    0.00    0.00    0.00    12.26   0.33    0.39    -19.63
Row 7:  0.00    0.00    0.00    0.00    0.00    0.00    17.71   0.25    -23.31
Row 8:  0.00    0.00    0.00    0.00    0.00    0.00    0.00    24.13   -27.27
Row 9:  0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    -0.86
Lower matrix:
Row 1:  1.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
Row 2:  0.30    1.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
Row 3:  0.40    0.32    1.00    0.00    0.00    0.00    0.00    0.00    0.00
Row 4:  0.50    0.26    0.08    1.00    0.00    0.00    0.00    0.00    0.00
Row 5:  0.60    0.21    0.06    0.04    1.00    0.00    0.00    0.00    0.00
Row 6:  0.70    0.16    0.05    0.03    0.03    1.00    0.00    0.00    0.00
Row 7:  0.80    0.11    0.03    0.02    0.02    0.01    1.00    0.00    0.00
Row 8:  0.90    0.05    0.02    0.01    0.01    0.01    0.01    1.00    0.00
Row 9:  0.10    0.37    0.11    0.08    0.06    0.05    0.04    0.04    1.00
Gaussian Elimination runs in 0.00 seconds
Matrix multiplication is computed using max of threads = 1 threads or cores
 Matrix size   = 9
```

# OpenMP V2

Matrix Size = 3840

| threads | LU-omp1 |
|---|---|
| 1 | 87.27 |
| 2 | 44.93 |
| 4 | 25.01 |
| 8 | 13.15 |
| 16 | 9.36 |
| 24 | 5.36 |
| 48 | 4.84 |



# OpenMP V2

-Best single node performance

-Best single thread performance

# MPI

-Modified version of cge-mpi1.cpp

-MPI_Send, MPI_Bcast, MPI_Recv

-After finding the pivot and swapping rows in the k loop, L and U are found with:

```
MPI_Bcast(tmp + k + 1 ,n - k - 1,MPI_FLOAT,master,MPI_COMM_WORLD);
// after tmp is broadcast to all processes, add it to L only on pid 0

if(myProcessID==0)
{
        for(i=k+1; i<n; i++)
        {
                L[k][i]=((-1.0)*tmp[i]);
        }
}
```

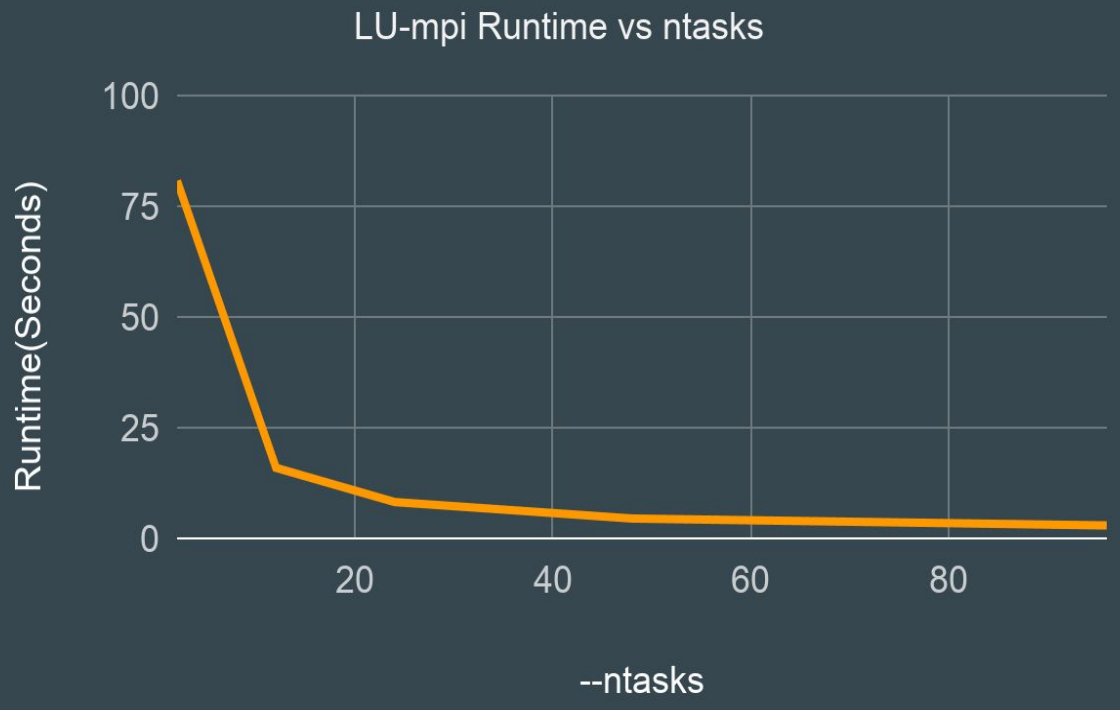
```
//Perform row reductions
j = lk ;
if (myProcessID < master) j++;
for ( ; j < nCols ; j++)
{
        for (i = k+1; i < n; i++)
        {
                U[j][i] = U[j][i] + tmp[i]*U[j][k];

        }
}
```

# MPI

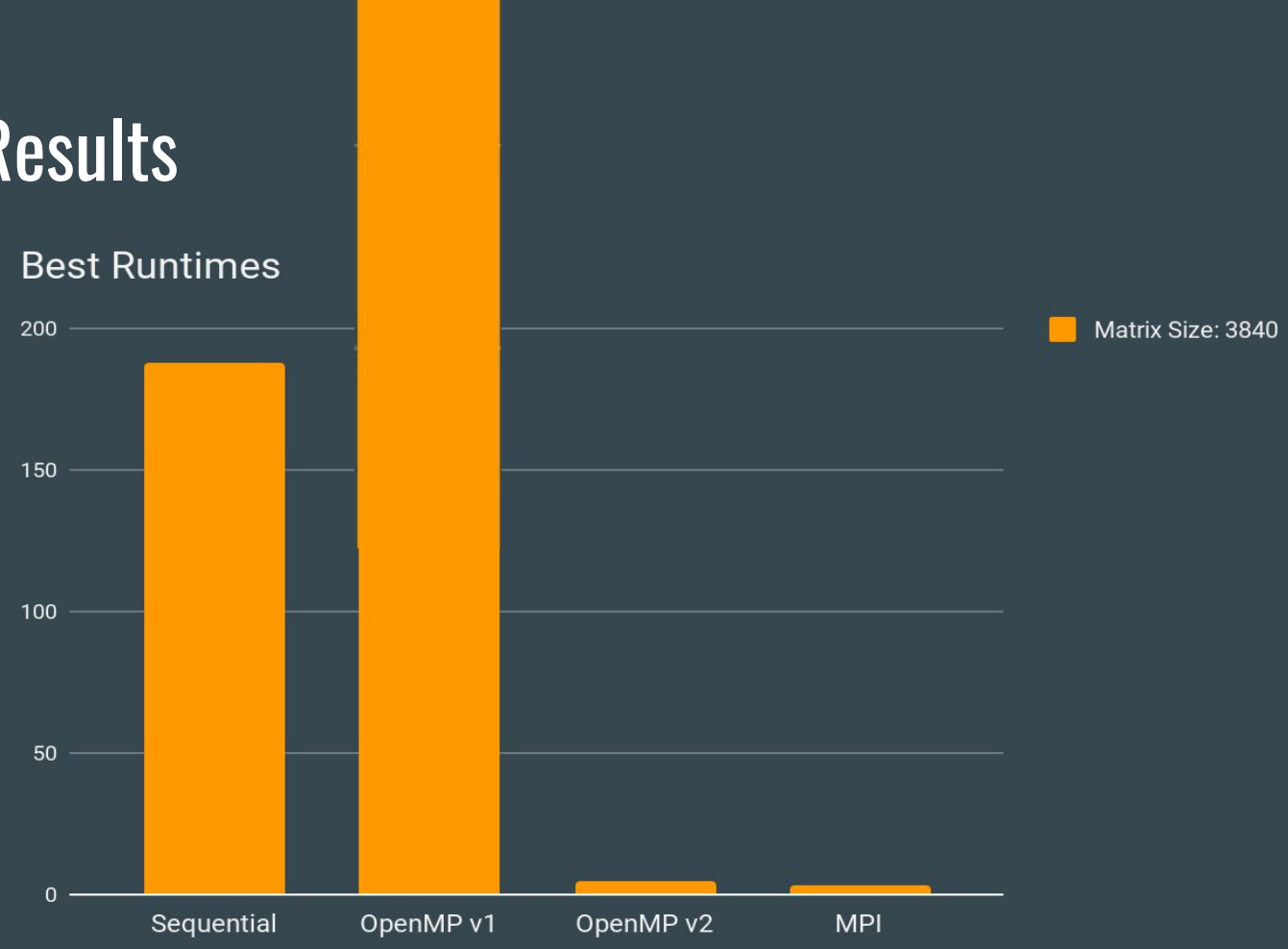
```
[john.pace@heracles lab4]$ sbatch mpi_slurm.sh 9 1
Submitted batch job 22802
[john.pace@heracles lab4]$ cat slurm_output.22802
(node2:0,1,2)

Input matrix:
Row 1:  0.500   1.500   2.000   2.500   3.000   3.500   4.000   4.500   5.000
Row 2:  1.500   2.000   2.500   3.000   3.500   4.000   4.500   5.000   5.500
Row 3:  2.000   2.500   4.500   3.500   4.000   4.500   5.000   5.500   6.000
Row 4:  2.500   3.000   3.500   8.000   4.500   5.000   5.500   6.000   6.500
Row 5:  3.000   3.500   4.000   4.500   12.500  5.500   6.000   6.500   7.000
Row 6:  3.500   4.000   4.500   5.000   5.500   18.000  6.500   7.000   7.500
Row 7:  4.000   4.500   5.000   5.500   6.000   6.500   24.500  7.500   8.000
Row 8:  4.500   5.000   5.500   6.000   6.500   7.000   7.500   32.000  8.500
Row 9:  5.000   5.500   6.000   6.500   7.000   7.500   8.000   8.500   40.500
Upper matrix:
Row 1:  5.000   5.500   6.000   6.500   7.000   7.500   8.000   8.500   40.500
Row 2:  0.000   0.950   1.400   1.850   2.300   2.750   3.200   3.650   0.950
Row 3:  0.000   0.000   1.658   0.316   0.474   0.632   0.789   0.947   -10.500
Row 4:  0.000   0.000   0.000   4.238   0.357   0.476   0.595   0.714   -13.167
Row 5:  0.000   0.000   0.000   0.000   7.770   0.360   0.449   0.539   -16.242
Row 6:  0.000   0.000   0.000   0.000   0.000   12.260  0.325   0.390   -19.633
Row 7:  0.000   0.000   0.000   0.000   0.000   0.000   17.712  0.255   -23.311
Row 8:  0.000   0.000   0.000   0.000   0.000   0.000   0.000   24.126  -27.266
Row 9:  0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   -0.865
Lower matrix:
Row 1:  1.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
Row 2:  0.300   1.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
Row 3:  0.400   0.316   1.000   0.000   0.000   0.000   0.000   0.000   0.000
Row 4:  0.500   0.263   0.079   1.000   0.000   0.000   0.000   0.000   0.000
Row 5:  0.600   0.211   0.063   0.045   1.000   0.000   0.000   0.000   0.000
Row 6:  0.700   0.158   0.048   0.034   0.026   1.000   0.000   0.000   0.000
Row 7:  0.800   0.105   0.032   0.022   0.017   0.014   1.000   0.000   0.000
Row 8:  0.900   0.053   0.016   0.011   0.009   0.007   0.006   1.000   0.000
Row 9:  0.100   0.368   0.111   0.079   0.061   0.050   0.042   0.037   1.000
LU Decomposition runs in 0.00 seconds
```

# Runtimes

| --ntasks | --ntasks-per-node | Matrix Size | LU-mpi |
|---|---|---|---|
| 2 | 2 | 3840 | 80.73 |
| 12 | 12 | 3840 | 15.93 |
| 24 | 12 | 3840 | 8.47 |
| 24 | 24 | 3840 | 8.20 |
| 48 | 24 | 3840 | 4.48 |
| 96 | 24 | 3840 | 2.92 |



# Overall Results



# Future Plans

- Improve performance on existing algorithms
- Critical section improvements of openMP program
- Implement static scheduling and compare to dynamic scheduling performance
- Add one-dimensional pivot array or permutation matrix for original matrix reconstruction
- CUDA
    a. Mentally prepare (CUDA By Example)
    b. Modify our previous algorithms
    c. Grab a beer (celebrate or commiserate?)
    d. Possibly repeat (from a or c)
- Develop a better parallel version applying different optimization methods

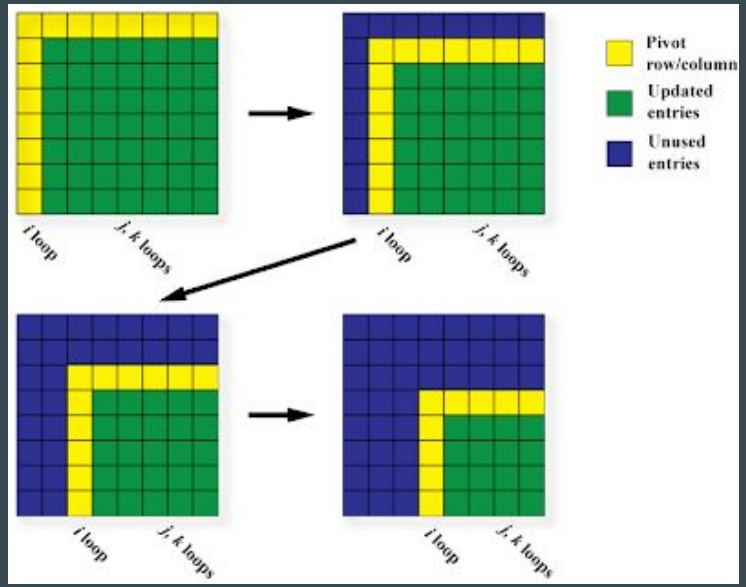
# Potential Improvement

```
for k := 1 step 1 until N
        begin p := 1/a[k, k];
            a[k, k] := p;
            for i := k+1 step 1 until N
                    begin q := -a[i, k]×p;
                            a[i, k] := q;
                            for j := k+1 step 1 until N
                                    a[i, j] := a[i, j] + q×a[k, j];
                    end;
        end;
```

**Program 3-4** Row wise form of Gaussian elimination without pivoting.

```
for k := 1 step 1 until N
        begin p := 1/a[k, k];
            a[k, k] := p;
            for i := k+1 step 1 until N
                    a[i, k] := -a[i, k]×p;
            for j := k+1 step 1 until N
                    begin q := a[k, j];
                            for i := k+1 step 1 until N
                                    a[i, j] := a[i, j] + q×a[i, k];
                    end;
        end;
```

**Program 3-5** Column-wise form of Gaussian elimination without pivoting.



# SIMD Pseudocode

```
for k := 1 step 1 until N
     begin
          m := idamax(a, k, N);
          piv[k] := m;
          swap(a, k, m, N);
          p := 1/a[k, k]
          a[k, k] := p;
          a[i, k] := -a[i, k] * p, (k+1 ≤ i ≤ N);
          for j := k+1 step 1 until N
               begin q := a[k, j];
                    a[i, j] := a[i, j] + q * a[i, k], (k+1 ≤ i ≤ N);
               end;
          end;
```

* Computation on columns of matrices vectorize well, computations on rows do not.

# Sources

https://www.cs.princeton.edu/courses/archive/fall11/cos323/notes/cos323_f11_lecture05_linsys.pdf

http://www.personal.psu.edu/jhm/f90/lectures/lu.html

https://www.cs.rutgers.edu/~venugopa/parallel_summer2012/ge.html#algo

# Questions?