# Assignment 2 LU Decomposition

**Chengyuan Li**

## Algorithm Design

### Data Initialization

In this LU decomposition algorithm, we have to initialize four matrix `A`, `L`, `U` and `pi`. So before running our algorithm, we need first to assign random values to `A`. And set the diagonal elements of `U` to 1. So I used following codes to randomize these matrix

```
#pragma omp parallel for schedule(static, 1)  shared(A, L, U, pi, n, nworkers)
    for (int i = 0; i < n; i++) {
        srand48_r(i, &buffer);
        A[i] = (double*)numa_alloc_local(n * sizeof(double));
        for (int j = 0; j < n; j++) {
            double rand_val;
            drand48_r(&buffer, &rand_val);
            A[i][j] = rand_val;
        }
        L[i] = (double*)numa_alloc_local(n * sizeof(double));
        U[i] = (double*) numa_alloc_local(n * sizeof (double));
        pi[i] = i;
        memset(L[i], 0.0, n);
        L[i][i] = 1;
        memset(U[i], 0.0, n);
```

In the row data allocation and initialization phase, use `omp for` to iterate over rows. Set the schedule clause of `schedule static(1)`, and each thread allocates its row with `numa_alloc_local`.Under the `static(1)` clause, the program assigns each iteration from beginning to end in a round-robin way to each thread.

### LU Decomposition

LU decomposition is a matrix decomposition technique that factors a matrix as the product of a lower triangular matrix (L) and an upper triangular matrix (U). This decomposition is fundamental in numerical analysis for solving systems of linear equations, matrix inversion, and determining the determinant of a matrix. The following is the pseudocode of the LU decomposition:

```
 inputs: a(n,n)
   outputs: π(n), l(n,n), and u(n,n)
   initialize(a,l,u,π)
   for i = 1 to n
     π[i] = i
   for k = 1 to n
 7        max = 0
 8        for i = k to n
 9          if max < |a(i,k)|
10            max = |a(i,k)|
11            k' = i                      //step1 find the max row index
12        if max == 0
13          error (singular matrix)
14        swap π[k] and π[k']
15        swap a(k,:) and a(k',:)
16        swap l(k,1:k-1) and l(k',1:k-1)  //step2 swap in L
17        u(k,k) = a(k,k)                 //step3 assign A[k,k:] to U[k,k:]
18        for i = k+1 to n
19          l(i,k) = a(i,k)/u(k,k)        //step4 update A and L
20          u(k,i) = a(k,i)               //step3 assign A[k,k:] to U[k,k:]
21        for i = k+1 to n
22          for j = k+1 to n
23            a(i,j) = a(i,j) - l(i,k)*u(k,j) //step4 update A and L
```

For my implementation, I will develop the parallel strategy by analyze the relationship between the four steps and find out what the best strategy is.

## Algorithm Implementation

### Step 1

For the step1, our purpose is to find the maximum column `k'` under `kth` row. This is the very first step before other steps begin, so here are two option to parallel here: one is add no parallelism here and make it serial, the other one is use `omp reduction` here to get the max value concurrently. I choose the first one because the benefit of using the parallelism is so trivial here that I can barely see the improvement.

For the step2, step3 and step4 we can use `omp for` to accomplish the parallelism.

### Step 2

As we can see in the line `16`, step2 is independent from other parts and the only thing we need to do is update `L` by swaping so we can do it by using `omp for nowait` here. Here could be kind of tricky because the `nowait` clause, which means that other parts could be done concurrently regardless of the swap in `L`. This is because the swaps just happend in `L[k, 1:k-1]` and the other parts which use the `L` just in the range of column from `k + 1` to the end. So this could be done concurrently. The following is my code:

```
#pragma omp for nowait
        for (int j = 0; j < k; j++) {
            std::swap(L[k][j], L[k_p][j]);
        }
```

### Step 3

In the pseudocode of the algorithm, we first assign `a(k, k)` to `u(k, k)` and after that we travel from column `k + 1` to `n` and assign `a(i, k + 1: n)` to `u(i, k + 1: n)`, so these two steps could be done together by assign `a(i, k:n)` to `u(i, k:n)`. And for the same reason as step2, this assignment is independent from other parts, so we can use `omp for nowait` to parallel. The following is my code:

```
#pragma omp for nowait
        for (int i = k; i < n; i++) {
            U[k][i] = A[k][i];
        }
```

### Step 4

With the similar reason of step2 and step3, we can update the `L` and `A` independently, so we also can use the `omp for nowait` clause. The following is my code:

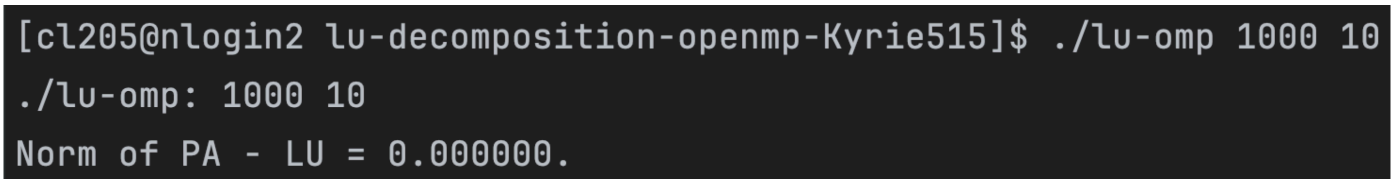
```
#pragma omp for nowait
        for (int i = n - 1; i > k; i--) {
            double t = A[i][k] / A[k][k];
            L[i][k] = t;
            for (int j = k + 1; j < n; j++) {
                A[i][j] -= t * A[k][j];
                }
        }
```

## Experiments

## Implementation Correctness And Data Race

### Correctness

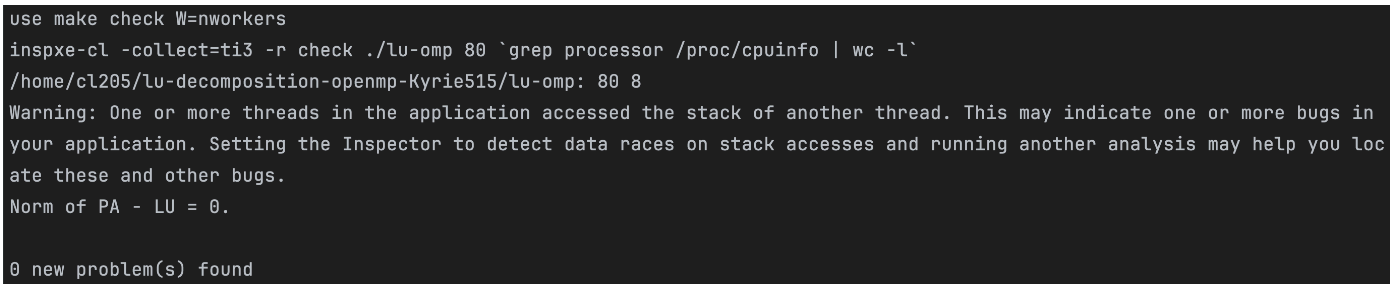
In the `utils.h` head file I define some functions to check the results of the decomposition. At first we check the results of norm by using `n = 1000` and `nworkers = 10`:



```
[cl205@nlogin2 lu-decomposition-openmp-Kyrie515]$ ./lu-omp 1000 10
./lu-omp: 1000 10
Norm of PA - LU = 0.000000.
```

### Data Race

We can use `make check` to chekc the data race of the program:



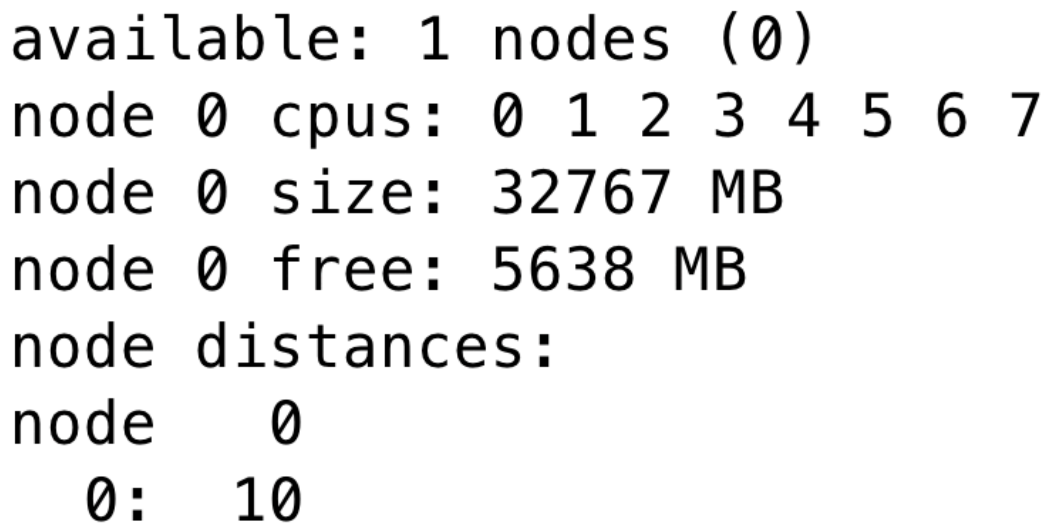
```
use make check W=nworkers
inspxe-cl -collect=ti3 -r check ./lu-omp 80 `grep processor /proc/cpuinfo | wc -l`
/home/cl205/lu-decomposition-openmp-Kyrie515/lu-omp: 80 8
Warning: One or more threads in the application accessed the stack of another thread. This may indicate one or more bugs in
your application. Setting the Inspector to detect data races on stack accesses and running another analysis may help you loc
ate these and other bugs.
Norm of PA - LU = 0.

0 new problem(s) found
```

As we can see, there is no data race in my program.

## Managing Threads and Data on NOTS

At first we can use `numactl --hardware` to see the configuration of the node:

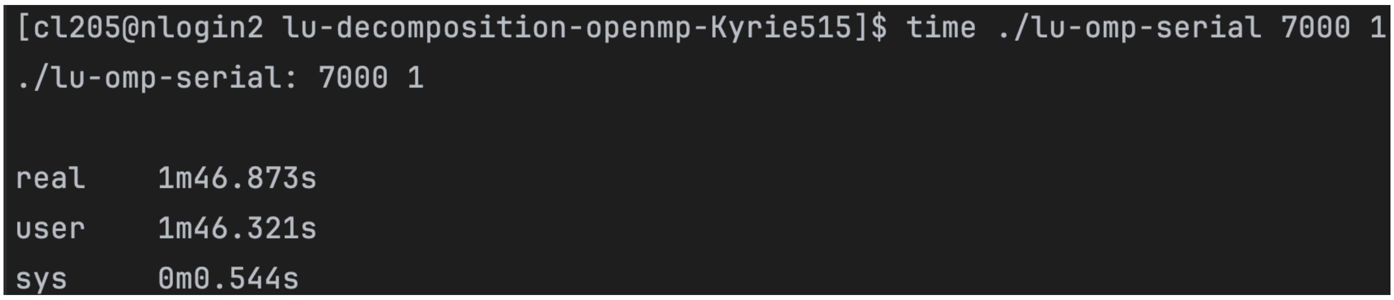


```
available: 1 nodes (0)
node 0 cpus: 0 1 2 3 4 5 6 7
node 0 size: 32767 MB
node 0 free: 5638 MB
node distances:
node   0
  0:  10
```

- **available: 1 nodes (0)**: This indicates that system has 1 NUMA node available, identified as node 0. NUMA systems are designed to optimize memory access by dividing memory and CPUs into distinct nodes. In systems with multiple nodes, each CPU/core has faster access to some parts of the memory (the memory local to its node) compared to other parts (the memory local to other nodes).
- **node 0 cpus: 0 1 2 3 4 5 6 7**: This lists the CPUs (or cores) that are part of node 0. In this case, CPUs 0 through 7 are present, indicating an 8-core processor or an 8-core segment of a larger processor that's allocated to this particular NUMA node.
- **node 0 size: 32767 MB**: This specifies the total size of memory available in node 0, which is 32,767 MB (or approximately 32 GB). This is the total physical memory allocated to this NUMA node.
- **node 0 free: 5638 MB**: This shows the amount of free memory available in node 0 at the time of the query, which is 5,638 MB (or approximately 5.6 GB). This indicates the unused portion of memory that applications and processes can still allocate.
- **node distances: node 0 0: 10**: This part of the output describes the relative memory access costs between nodes. The matrix shows the "distance" to access memory across nodes. In systems with multiple NUMA nodes, this matrix would indicate the relative cost of accessing memory from one node to another. A lower number indicates lower latency and higher bandwidth availability. In your output, since there's only one node, it shows the self-distance as 10, which is a standard reference value indicating the baseline cost of local memory access within the same node.

## Parallel Efficiency

All the following data I get is based on matrix size `n = 7000` to calculate. At first we need to get the serial time: use `./lu-omp-serial 7000 1` to get the serial time:



```
[cl205@nlogin2 lu-decomposition-openmp-Kyrie515]$ time ./lu-omp-serial 7000 1
./lu-omp-serial: 7000 1

real    1m46.873s
user    1m46.321s
sys     0m0.544s
```

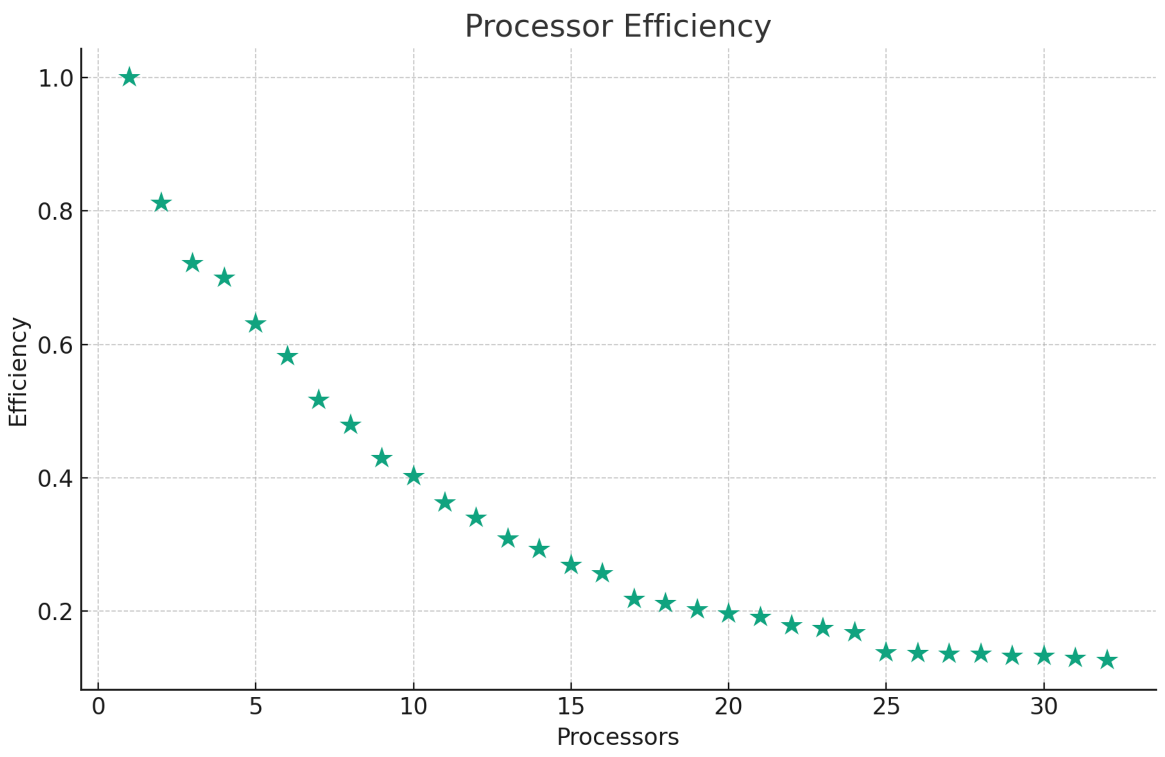
As we can see, the real time is `106.873`s. And then we run the `./submit.sbatch` to get the run time from `nworkers = 1` to `nworkers = 32`. And we can use the following formula:

$$\eta = \frac{S}{p * T(p)} \tag{1}$$

So the following are the time data of different `nworkers`:

| Processors | Time/s | Efficiency | Processors | Time/s | Efficiency |
|---|---|---|---|---|---|
| 1 | 106.87 | 1.00 | 17 | 28.90 | 0.22 |
| 2 | 65.85 | 0.81 | 18 | 27.99 | 0.21 |
| 3 | 49.38 | 0.72 | 19 | 27.72 | 0.20 |
| 4 | 38.21 | 0.69 | 20 | 27.22 | 0.20 |
| 5 | 33.86 | 0.63 | 21 | 26.70 | 0.19 |
| 6 | 30.61 | 0.58 | 22 | 27.22 | 0.18 |
| 7 | 29.56 | 0.51 | 23 | 26.61 | 0.17 |
| 8 | 27.91 | 0.48 | 24 | 26.41 | 0.17 |
| 9 | 27.68 | 0.43 | 25 | 31.01 | 0.14 |
| 10 | 26.60 | 0.40 | 26 | 29.96 | 0.14 |
| 11 | 26.83 | 0.36 | 27 | 29.16 | 0.14 |
| 12 | 26.25 | 0.34 | 28 | 28.19 | 0.14 |
| 13 | 26.69 | 0.31 | 29 | 27.78 | 0.13 |
| 14 | 26.08 | 0.29 | 30 | 26.87 | 0.13 |
| 15 | 26.45 | 0.27 | 31 | 26.63 | 0.13 |
| 16 | 26.04 | 0.26 | 32 | 26.40 | 0.13 |

As we can see the parallel efficiency for `processors = 16` is `26%`. And the following is the figure of efficiency:


Processor Efficiency
Efficiency
Processors
1.0
0.8
0.6
0.4
0.2
0
5
10
15
20
25
30