Assignment 2: LU Decomposition

Aniket Gattani

Program Overview:

The LU Decomposition is the representation of the matrix ***A*** into lower triangular matrix ***L*** and upper triangular matrix ***U***. The program performs the following steps sequentially:

1. Initializes a random whole number matrix ***A*** and ***A_ORG*** which is the copy of ***A***.
2. Performs pivoting to reduce the rounding-off errors. In this step, we modify the matrix ***A*** but keep the matrix ***A_ORG*** unchanged. It also uses the permutation matrix P which is a 1-D matrix to save the pivoting state.
3. It also checks the L2,1 Norm of the residual matrix (***A*** - ***LU*** where ***LU*** is the cartesian product of ***L*** and ***U***).

Distribution of Data:

The distribution of data is one of the most important aspects of this program. Since the program is run on processors with NUMA architecture, the different cores on the machine have to use their local caches and DRAMs effectively and reduce the usage of remote DRAMs which takes significant time to load. There are a few caveats that we have to consider when distributing the data:

- Structure of the matrices
- Initializing the matrices
- False sharing
- Accessing the matrices

We will discuss more about how these are handled below.

Structure of the matrices

The matrices are of the type:

```
typedef struct {double** mat; int rows; int cols;} matrix;
```

- The above structure helps us to distribute different rows of the ***matrix.mat*** to different threads. Since we are only dealing with square matrices lets just assume ***n*** as the size of the matrix which means ***rows = cols = n***

- Row pivoting becomes cheaper since all we need to do is swap the pointers to different rows and reduce the extra work to swap the entire data of the rows. This can incur a bit of overhead because of remote fetches but the tradeoff is very low.

- One can argue that the matrices can be stored in a 1-D manner with size n*n. However, that would increase false sharing which is described below.

Initializing the matrices

- We allot the memory dynamically to the struct matrix. We first just allot memory to **matrix.mat** as follows:

```
a.mat = (double**)malloc(sizeof(double*)*n);
a.mat[i] = (double*)malloc(sizeof(double)*n);
```

- Since Linux uses a first touch policy, the workers should touch the rows which they are about to compute upon to make effective use of caches. Now ***malloc*** does not allot memory unless the memory locations are accessed. So I iterated over all the rows for a particular worker and tried to initialize all elements inside every row. Not only does this allot memory but also touch the rows for individual threads.

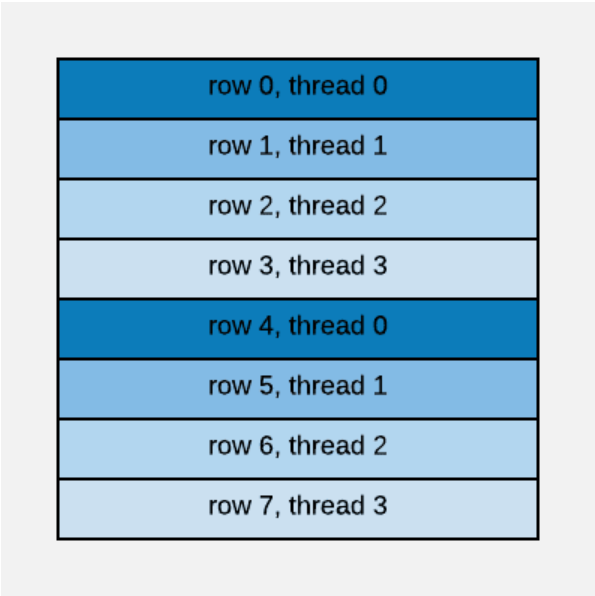
- The matrix ***A*** is initialized using a reentrant pseudo random generator which is parallelized. For every thread, we use a different seed which is the number of that thread. I have used rand_r() which is thread safe instead of rand().

False sharing:

- Since we have arranged the matrix in different rows and allocating data dynamically to each row, the rows have contiguous allocations but consecutive rows might not.

- The cache lines are about 64 bytes on NOTS. Since each thread is operating on different rows, the only way false sharing can occur is when we update the end of one row and the beginning of the consecutive row. Since the individual rows might not be continuous, it means that we have reduced false sharing.

Accessing the matrices:



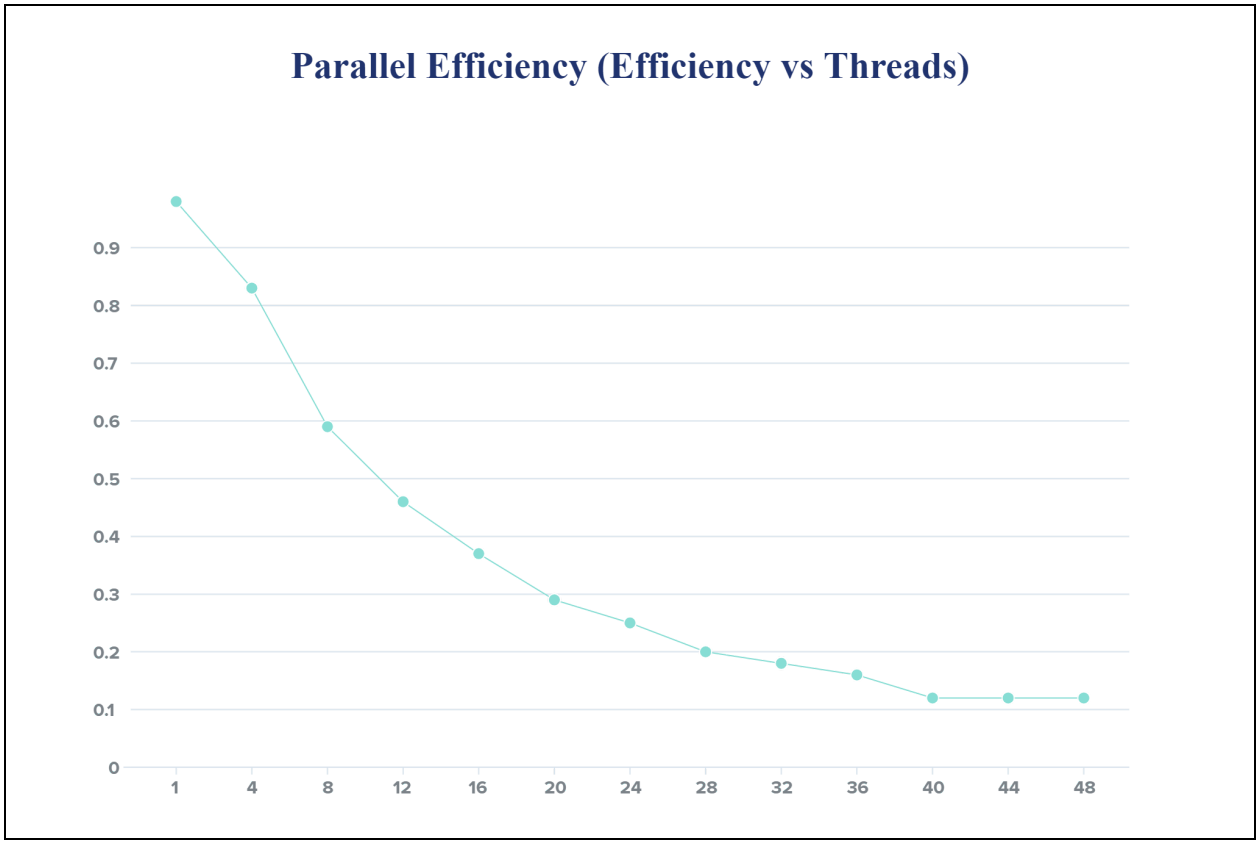
- We want the data distribution as illustrated above. The rows represent the rows of the square matrices A, L. Let us say that the number of threads specified for the run is 4. Then thread 0 should operate on rows 0, 4, 8, … , thread 1 on rows 1, 5, 9, …, thread 2 on rows 2, 6, 10, … and thread 3 on rows 3, 7, 11, …

- I avoided giving consecutive rows to each worker because LU Decomposition works on a submatrix for a given pivot row. Let's say the pivot row number is ***i***, then the submatrix for computation is ***((i,i), (n,i), (n,n), (i,n))***. If I would have allotted consecutive rows, then the workers will have an uneven distribution of data for every pivot row and I would have forfeited the benefit obtained during the first touch.

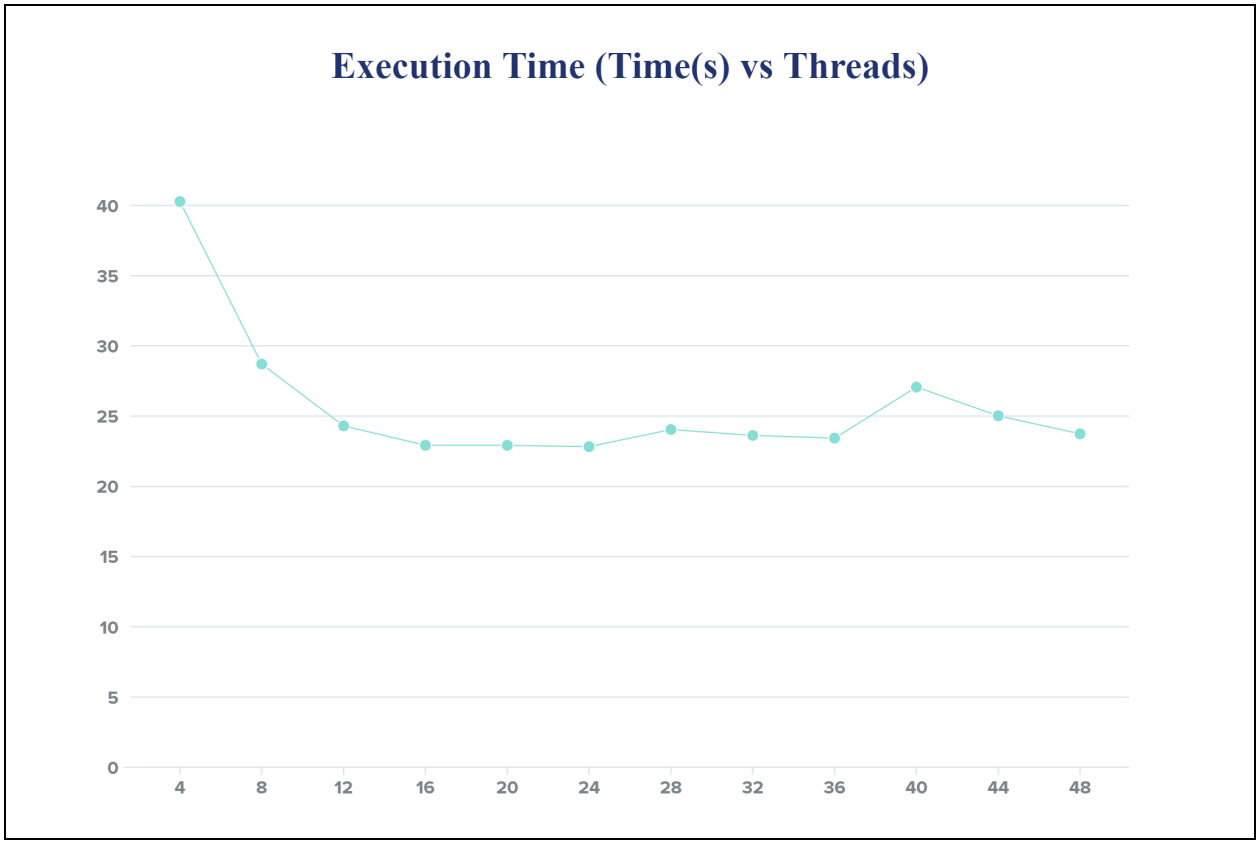
- All for loops are scheduled statically. This is because OpenMP allotts the tasks in a sequential manner to every row and the allocation becomes deterministic. We want to tightly bind our hardware threads to a particular openMp thread to make the best use of caches.

- Since openmp has an implicit barrier at the end of every parallel for, the threads are synchronized implicitly after updating the matrix ***L*** and ***A.***

Efficiency Graph:



Computed on node with 48 hardware threads with matrix size = 7000



Computed on node with 48 hardware threads with matrix size = 7000

Matrix Size	**Workers**	**Hardware threads**	**Remote Cache Fetches**	**Local Cache Fetches**	**Percentage (Remote%Local)**
7000	8	32	3.76e+05	2.75e+08	0.1%
7000	16	32	4.94e+05	7.40e+07	0.67%
7000	32	32	1.37e+05	7.63e+07	0.18%

HPCToolkit remote vs local DRAM Fetches

From the efficiency graphs, we see the following:

- The threads are bound to hardware threads using ***OMP_PROC_BIND=spread.*** Spread helps the threads to sparsely distribute the threads on the hardware threads and prevents a skewed distribution.

- **Threads <= ¼ hardware threads, Total Hardware Threads = 48** :
    - The efficiency is high in this phase because we benefit from threads performing parallelly. The overall work is divided into multiple cores.
    - There are a few remote and local cache fetches because the threads are allotted sparsely on both Numa sockets and the pivot row fetch requires a remote fetch for threads on the other socket and local DRAM fetch for threads on the same socket but different nodes. However, the overall work split is sufficient to overcome the remote fetch effects.

- **Threads > ¼ of Hardware Threads, Total Hardware Threads = 48** :
    - The speedup increases upto 12 threads but then decreases abruptly. From the time graph, we see that the execution time stays the same. In this phase all the cores have at least one thread working on them.
    - From looking at the hpctoolkit stats it looks like the remote DRAM fetches remain the same but local DRAM fetches decrease in this phase. This is because now we have more cores running in parallel and they all have a lesser number of rows divided amongst them. This means that their L2 and L3 caches handle more proportion of the data and consequently the fetches to local DRAM are decreased. (Hpctoolkit stats are obtained from core with h/w threads = 32)
    - This overcompensates for the benefit of parallelism because the ratio of remote DRAM fetches to local DRAM fetches is increased. Remote cache fetches will remain the same because pivot rows still have to be fetched by all processors.
    - The remote fetches are less than 1% which implies that the data organization and accesses are optimal.
    - For threads = number of h/w threads, the percentage for remote vs local DRAM almost stays the same because hyperthreading doesn't add much to the cache efficiency. The hyperthreads will work on the same core and will not add extra cache to the cores. Neither will they reduce local to remote fetches. The data organization and distributions remain almost identical.

- As an experiment, I tried finding efficiency with matrix size = 6000. The efficiency is slightly better than with n=7000 at threads = 48 (efficiency = 0.2). This is because the overall memory allocation is decreased and more proportion of data can be obtained from L3 and L2 caches. So accesses to local DRAM decrease and we see a small benefit.

Note: For some reason, NOTS always allocates a processor with 48 hardware threads when running a batch job. The interactive session requests on the other hand allocates a processor with 32 threads. Hence, the hpc viewer stats are for nodes with h/w threads = 32 and the efficiency as well as time graphs are for h/w threads = 48. The comparisons are made not considering the thread numbers but instead at architecture level.

Correctness:

- The program was found to **not** contain any data races by running the command ***make check***
- On observing the L2,1 norm for matrix size = 7000, the L2,1 norm was found to be 3.62451e-07 which is pretty small considering the matrix of this size. All values inside the matrix are within the range 1-100.
- Furthermore, the difference stays the same even with different threads. This implies that the matrix is deterministic and is independent of threads and the above comparisons are valid.