Assignment 2 Report

Yi Zhou (yz166)

1. Project Description

The program implements a LU decomposition with row pivoting on a square matrix A to get its upper-triangular and lower-triangular matrixes U and L as well as a compact representation of its permutation matrix P. The program uses OpenMP to first allocate memory on all threads available for matrix A with uniform random numbers, U, and L in double precision and matrix P as a one-dimension array. Then, it runs a parallel implementation of LU decomposition on these matrixes. The pseudocodes for matrix initialization and LU decomposition with parallelization elements are shown in the figure below.

```
initialization():
inputs: A(n,n), P(n), U(n,n), L(n,n)

omp parallel region:
    initialize A as an n x n matrix with uniform random number
    initialize P as a vector of length n with values from 1 to n
    initialize U as an n x n matrix with values same as A
    initialize L as an n x n matrix with 1s on the diagonal and 0s above the diagonal
```

Figure 1 pseudocode for parallel matrix initialization

```
max = 0.0
k' = 0
omp parallel region:
  for k = 1 to n
    local_max = 0.0
    local_k' = 0
    for i = startrow to n with a step of nworkers increase
      if local_max < |U(i,k)|
        local_max = |U(i,k)|
        local_k' = i
    max = local_max
    k' = local_k'
    if max == 0
      error (singular matrix)
    swap P[k] and P[k']
    swap U(k,:) and U(k',:)
    swap L(k,1:k-1) and L(k',1:k-1)
    for i = startrow' to n with a step of nworkers increase
      L(i,k) = U(i,k) / U(k,k)
      U(i,k) = 0
      for j = k+1 to n
          U(i,j) = U(i,j) - L(i,k) * U(k,j)
```

Figure 2 pseudocode for parallel LU decomposition

2. Data Partition

As seen from the LU decomposition pseudocode above, all the value fetching and updating is on the row base, except the row pivoting. Thus, choose 1D block distribution to allocate the data in memory is a better choice for the program. In addition, only a limited number of processors can be used for the program and to make it easier to get the index of the first row of memory allocation that each thread has in each iteration, 1D block-cyclic distribution is applied in the parallel program.

The matrix data is allocated as rows and evenly distributed to threads available. Besides, since numa_alloc_local is used in the initialization, each thread allocates its responsible rows data in memory attached to the socket, which potentially decrease the memory bandwidth among these rows data.

3. Program working logic

In the matrix memory allocation process, the program evokes a parallel region and each thread allocates all matrixes' data that has row index equals to its thread number, tid, plus a multiple of the number of workers.

During the LU decomposition, the program evokes a parallel region again. For each iteration, threads first get their own pivot row value concurrently and update a global maximum variable sequentially. Then, the parallel region is blocked for two threads to finish the data swapping. Lastly, threads update their allocated data for matrix L and U concurrently.

4. Parallelism Exploitation

The program exploits parallelism in two parts. First, it is parallelized for threads to allocate their own data in memory. Second, threads are parallelized to run the same implementation in matrix_size iterations. Rather than using one single thread to allocate data memory, it is better to have all threads allocate the same matrixes' row indexes data in memory, which greatly decreases the memory bandwidth for reading and writing data from other threads to the master thread in one socket and the remote access between sockets. This is also the reason why the program applies parallelism in the decomposition.

5. Parallel Synchronized

There are two positions in the program where the parallel work is synchronized.

In the process of row pivoting, threads first get their own maximum value for column k into private variables, local max and local k'. Then, the parallel work is synchronized for each thread to update the shared variable, global max and global k' sequentially.

The second one is when performing the swap operations, the program blocks the parallel region to wait for two threads that are responsible for matrix row k and global k' to complete their data swapping.

6. Program Performance

To evaluate the performance of the program implementation, the problem size n = 8000 is used for all measurements. The table1 shows all timing measurements:

- S: wall clock time of a sequential execution, measured using command: make runs
- P: the number of processors used
- T(P): the wall clock time of an execution on p processors, measured using command: make runp / ./lu-omp 8000 p
- Parallel efficiency: is computed as S/(p * T(p))

Table 1. Timing measurements for the LU decomposition phase

S	P	T(P)	parallel efficiency
179s	1	178s	1.005618
179s	2	91s	0.983516
179s	4	51s	0.877451
179s	8	41s	0.545732
179s	16	40s	0.279688
179s	32	43s	0.130087

The figure3 provides a graph representation for the parallel efficiencies using 1, 2, 4, 8, 16, and 32 threads.

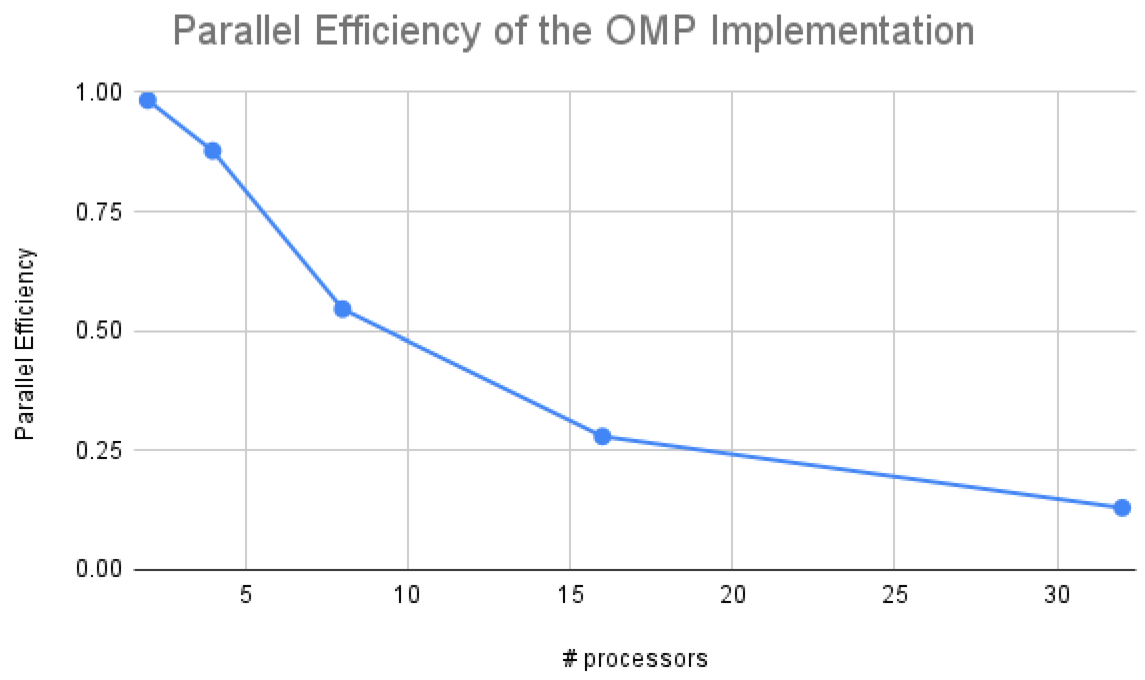
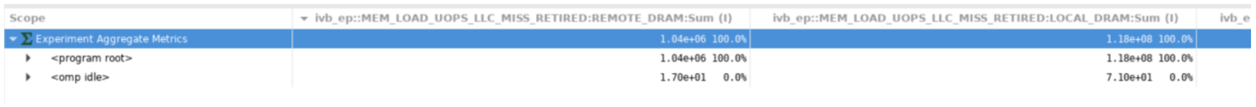


Figure 3 Parallel Efficiency vs. # processors[1]

As shown in the table and figure above, my implementation gets a parallel efficiency around 28% with 16 processors and 13% with 32 processors, which I think is a reasonable result for the LU decomposition with partial pivoting algorithm (achieves about half of the parallel efficiency compared with the block LU decomposition algorithm one that is given).

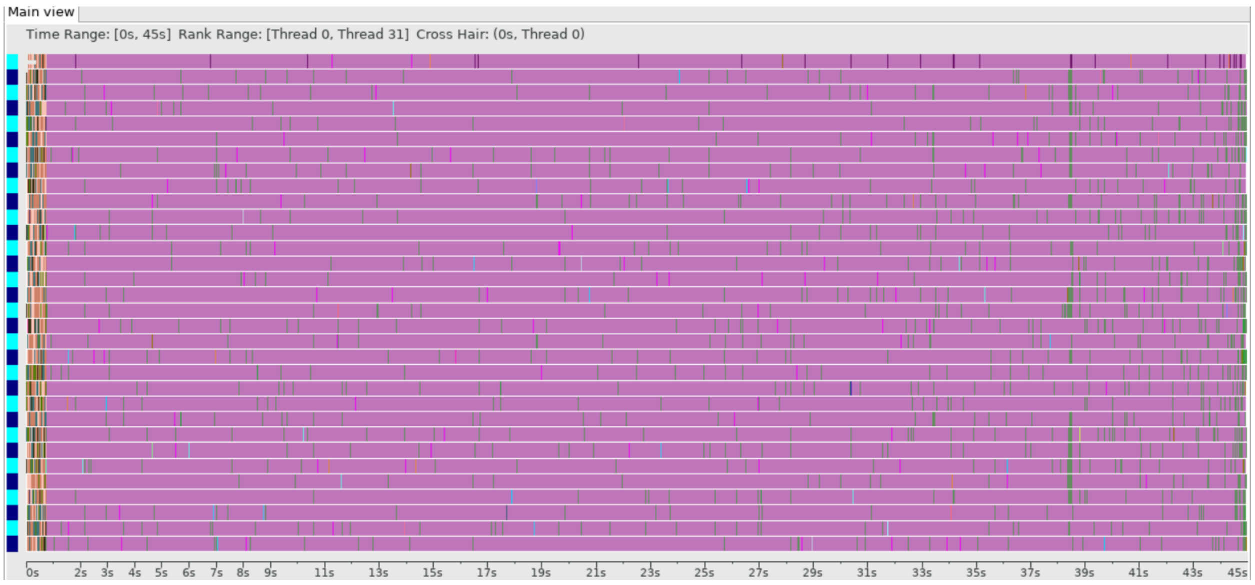
The figure4 shows the REMOTE_DRAM and LOCAL_DRAM measurements using HPCToolkit. It is measured using variables OMP_PROC_BIND=spread, problem size n = 8000 and 32 processors. The remote access is computed as 1.04e+06/(1.04e+06 + 1.18e+08) = 0.87%, which is a percentage that is expected to this parallel program. This is the best remote access I got using HPCToolkit measurements. Other remote accesses computed are around 2%. Thus, my program implementation spends most time on the local memory, greatly decreases the time-consuming remote memory transactions.



Scope	ivb_ep::MEM_LOAD_UOPS_LLC_MISS_RETIRED:REMOTE_DRAM:Sum (I)	ivb_ep::MEM_LOAD_UOPS_LLC_MISS_RETIRED:LOCAL_DRAM:Sum (I)	ivb_e
Experiment Aggregate Metrics	1.04e+06 100.0%	1.18e+08 100.0%	
<program root>	1.04e+06 100.0%	1.18e+08 100.0%	
<omp idle>	1.70e+01 0.0%	7.10e+01 0.0%	

Figure 4 remote access

Moreover, shown by the trace view (for LU decomposition function) in figure5, the most time costing parts of the program is where threads allocate matrixes data in memory at the beginning and parallel work is being synchronized closed to the end of the program. The paralleled LU decomposition computation has evenly distributed work for all threads. Besides, even though the row pivoting requires all threads to be synchronized to update the global max variable, the number of processors used is not large and the effect on the time is negligible.



Time Range: [0s, 45s] Rank Range: [Thread 0, Thread 31] Cross Hair: (0s, Thread 0)

Figure 5 HPCToolkit - trace view