Design and Implementations of Parallel OpenMP and MPI LU factorization : Application to Finite Difference Method.

PUNIT K. JHA[*], HEEHO PARK[**], ZIYU ZHOU[!], AND CONG LIN[!*]

[1] *The University of Illinois at Urbana-Champaign*

Compiled January 16, 2018

Many scientific applications have linear systems $Ax = b$ which need to be solved for different vectors b. LU decomposition, which is a variant of Gaussian Elimination, is an efficient technique to solve a linear system. The main idea of the LU decomposition is to factorize the give matrix A into an upper (U) triangular and a lower (L) triangular matrix such that $A = LU$. In this report we present different algorithms for the LU factorization of dense matrices. Finite-difference methods are numerical methods that are used to find solutions to differential equations using approximate spatial and temporal derivatives. These derivatives are based on discrete values at spatial grid points and discrete time levels. The advection equation is the partial differential equation that governs the motion of a conserved scalar field as it is advected by a known velocity field, which is a vector. Discretization of the 2D advection equation leads to a system of linear equations that we solve by the LU decomposition. We implement and analyze the following algorithms in this report:

1. Serial LU Decomposition .
2. Serial Block LU Decomposition with partial pivoting.
3. Row Block OpenMP LU Decomposition .
4. OpenMP Task based Block LU Decomposition with partial pivoting.
5. MPI LU Decomposition.
6. 2D-Cartesian Topology with OpenMP implementation of LU decomposition.

INTRODUCTION

Finding the solutions of linear systems is an important problems in many areas of science and engineering like density functional theory [1], quantum transport [2], dynamical mean field theory [3], and uncertainty quantification, for example. There are many methods for solving a linear system with direct or iterative methods

LU Decomposition

The most commonly used direct method for solving general linear system is Gaussian elimination with partial pivoting. LU decomposition decomposes a square matrix A into a lower triangular matrix L and a upper triangular matrix U, such that

$$A = LU.$$

There are many algorithms available to solve LU decomposition, such as the Doolittle algorithm and the Crout algorithm. In this report, we adopt the idea of the Doolittle algorithm to implement our serial algorithms.

The Doolittle algorithm requires L to be a unit matrix, where all the elements in the main diagonal are ones, so that the system of equations to solve the LU decomposition can be determined.

It traverses the original matrix by column, divides the elements below the diagonal by the diagonal elements to obtain entries in L, and subtracts values from each row to obtain U.

SOLVING LINEAR EQUATIONS

Solving linear equations after apply LU decomposition to the original matrix is much easier than solving them directly. Denote the linear system as

$$Ax = b.$$

Given that the original matrix A has been decompose as $A = LU$, we can solve the above system by first solving the equations of $Ly = b$ to obtain y, and then solving $Ux = y$ to obtain x which is the solution to $Ax = b$.

The reason why this process is much easier is that there is no need to apply Gaussian elimination since A has been decomposed into two triangular matrices so that we only need to apply forward and backward substitution to solve $Ly = b$ and $Ux = y$.

OPEN MP

OpenMP (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior. OpenMP uses a portable, scalable model that gives programmers a simple and flexible interface for developing parallel applications for platforms ranging from the standard desktop computer to the supercomputer. [4]

The `parallel` Construct

It is when the parallel construct is encountered that new threads are started. These threads will continue in existence until the parallel region for which they were started comes to an end. A parallel region is delimited by a block of code enclosed in in C/C++ and by an end parallel construct in Fortran.

The `master` Construct

The MASTER directive specifies a region that is to be executed only by the master thread of the team. All other threads on the team skip this section of code There is no implied barrier associated with this directive.

The `task` Construct

Tasks in OpenMP are code blocks that the compiler wraps up and makes available to be executed in parallel. The task directive defines the code associated with the task and its data environment. The task construct can be placed anywhere in the program; whenever a thread encounters a task construct, a new task is generated.

The Depend Clause

The depend clause takes a type followed by a variable or list of variables.

```
pragma omp task depend(in: x) depend( out: y)
depend(inout: z)
```

The (address of) variables passed to the depend clause are used to correctly order the tasks. These constraints are determined by the type of dependency specified:

IN– dependencies will make a task dependent on the last task that used the same variable .

OUT– dependencies will make a task dependent on the last task that used the same variable .

INOUT– dependencues are the same as an out dependency, they are only used for readability.

These constraints establish an order of tasks and only between sibling tasks. There is no data movement or synchronization with respect to external accesses of the data. Only a synchronization of memory accesses between dependent tasks is established.[7].

Open MP Data sharing attribute clauses

The following data sharing attribute clauses are defined in OpenMP. **shared:** the data within a parallel region is shared,i.e, visible and accessible by all threads simultaneously. By default, all variables in the work sharing region are shared except the loop iteration counter.

private: the data within a parallel region is private to each thread, which means each thread will have a local copy and use it as a temporary variable. A private variable is not initialized and the value is not maintained for use outside the parallel region. By default, the loop iteration counters in the OpenMP loop constructs are private.

firstprivate: like private except initialized to original value.

lastprivate: like private except original value is updated after construct. **reduction:** used to join work from all threads after construct.

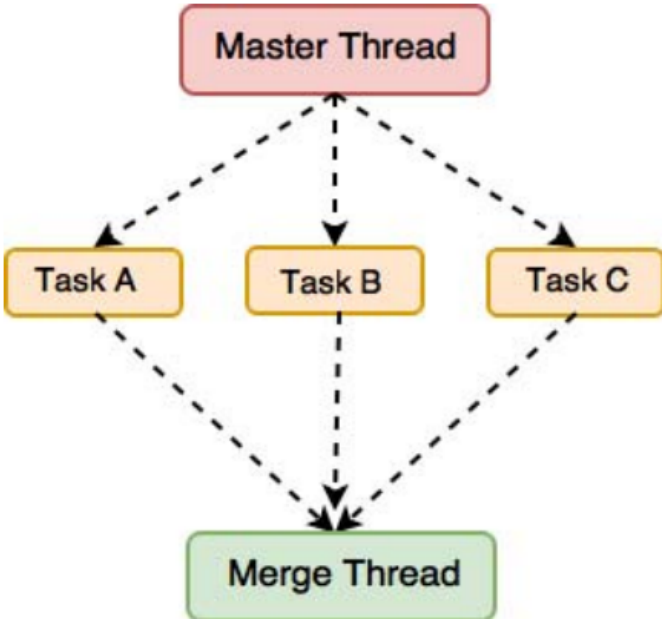

Fig. 1. Master thread creates a team of parallel worker threads; the tasks are executed in parallel by the worker threads; at the end of the parallel region, the tasks are merged.

OpenMP Scheduling

The various scheduling options that are available with OpenMp are:

- **Static Schedules** - By default, OpenMP statically assigns loop iterations to threads. When the parallel for block is entered, it assigns each thread the set of loop iterations it is to execute.
- **Dynamic Schedules**- OpenMP assigns one iteration to each thread. When the thread finishes, it will be assigned the next iteration that hasn't been executed yet. However, there is some overhead to dynamic scheduling. After each iteration,

the threads must stop and receive a new value of the loop variable to use for its next iteration.

- **Guided Schedules** - Instead of static, or dynamic, we can specify guided as the schedule. This scheduling policy is similar to a dynamic schedule, except that the chunk size changes as the program runs. It begins with big chunks, but then adjusts to smaller chunk sizes if the workload is imbalanced.
- **Auto** - The auto scheduling type delegates the decision of the scheduling to the compiler and/or runtime system.

MPI

Message Passing Interface (MPI) is a standardized and portable message-passing standard. The standard defines the syntax and semantics of a core of library routines useful to a wide range of users writing portable message-passing programs in C, C++, and Fortran. We used the following MPI functions in our codes :

- **MPI_Send**- Performs a blocking send.
- **MPI_Recv**- Blocking receive for a message
- **MPI_Bcast**- Broadcasts a message from the process with rank "root" to all other processes of the communicator
- **MPI_Cart_create**- Makes a new communicator to which topology information has been attached
- **MPI_Cart_shift**- Returns the shifted source and destination ranks, given a shift direction and amount.
- **MPI_Cart_coords**-Determines process coords in cartesian topology given rank in group.

BLUE WATERS

Brief Intro to Blue Waters Super Computing System

Blue Waters is a Cray XE6/XK7 system consisting of more than 22,500 XE6 compute nodes. Each XE6 node contains two AMD Interlagos processors. I also has more than 4200 XK7 compute nodes. Each XK7 has one AMD Interlagos processor and one NVIDIA GK110 *Kepler* accelerator in a single Gemini interconnection fabric.

Blue Waters Nodes

The Blue Waters has three different types of nodes which are listed below:

Traditional Compute Nodes (XE6)

The XE6 dual-socket nodes have 2 AMD Interlagos model 6276 CPU processors (one per socket) with a clock speed of at least 2.3 GHz and 64 GB of physical memory. The Interlagos architecture employs the AMD Bulldozer core design in which two integer cores share a single floating point unit. The Bulldozer core has 16KB/64KB data/instruction L1 caches and 2 MB shared L2 . Each core is able to complete up to 8 floating point operations per cycle. The architecture supports 8 cores per socket with two die, each die containing 4 cores forming a NUMA domain. The 4 cores of a NUMA (Non-Uniform Memory Access) domain share an 8 MB L3 cache.

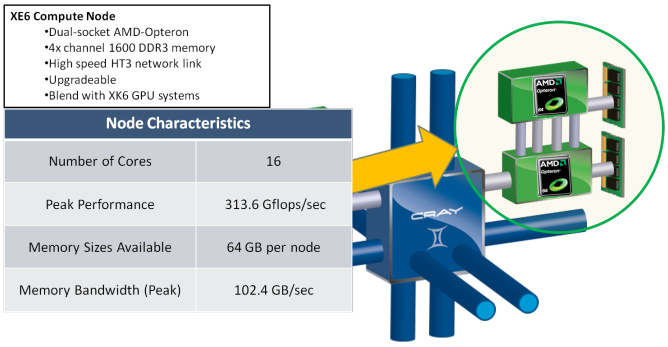

Fig. 2. Traditional Compute Nodes (XE6)

GPU-enabled Compute Nodes (XK7)

The accelerator nodes are equipped with one Interlagos model 6276 CPU processor and one NVIDIA GK110 "Kepler" accelerator K20X. The CPU acts as a host processor to the accelerator. In the current design, the NVIDIA accelerator does not directly interact with the Gemini interconnect so the date has to be moved to a node containing an accelerator. Each XK7 node has 32 GB of system memory while the accelarator has 6 GB of memory. The Kepler GK110 implementation includes 14 Streaming Multiprocessor (SMX) units and six 64 bit memory controllers. Each of the SMX units feature 192 single precision CUDA cores.

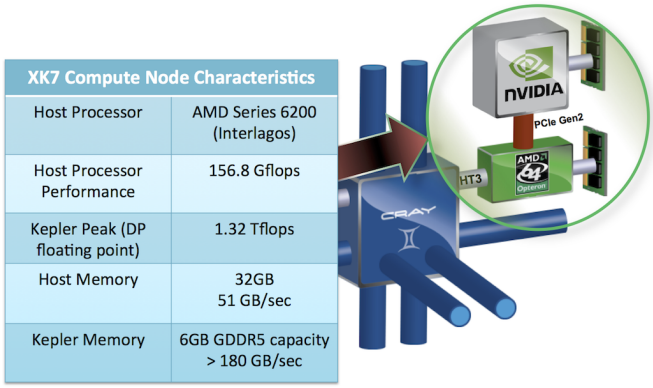

Fig. 3. GPU-enabled Compute Nodes (XK7)

I/O and Service Nodes (XIO)

Each Cray XE6/XK6 XIO blade has four service nodes. These service nodes use AMD Opteron "Istanbul" six-core processors, with 16 GB of DDR2 memory, and the same Gemini interconnect processors as compute nodes. These XIO nodes perform on of the following functions:

- **esLogin Nodes**: provide user access and log in services for the system.
- **PBS MOM Nodes**: these are nodes which run the jobs (where the PBS script gets executed and aprun is launched) and where interactive batch jobs (qsub -I) place a user.
- **Network Service Node**: used to connect to external network setting.
- **LNET Router Nodes**: Manage file system metadata and transfer data to and from storage devices and applications.

Running Scripts on Blue Waters

The aprun command is used to specify the resources and placement parameters needed for the scripts at launch. At a high level, aprun is similar to mpiexec or mpirun. The following are the most commonly used options for aprun.

- -n: Number of processing elements(PEs) required for the application.
- -N: Number of PEs to place per node.
- -S: Number of PEs to place per NUMA node.
- -d: Number of CPU cores required for each PE and its threads.
- -j: Number of CPUs to use per compute unit .
- -cc: Bind PEs to CPU cores.

ALGORITHMS IMPLEMENTED

The algorithms that were implemented by us are explained in the sections below. These algorithms apply OpenMP and MPI.

Serial LU Decomposition with Partial Pivoting

First of all we look at the serial LU decomposition algorithm which is shown below. It consists of three nested loops, which can be adopted for parallel algorithms. For each iteration of the outer loop, there is a division step and an elimination step. With each step in computation, only the lower right section of the matrix becomes active, this leads to an increase in the amount of computation in the direction of the lower right corner of the matrix –thus causing a non-uniform computational load. We also have to consider the pivoting operation just inside the inner loop that can lead to additional computational expenses.

Algorithm 1. Algorithm of Serial LU Decomposition with Partial Pivoting

```
/* Initialize permutation matrix */
for int i ← 0 to n − 1 do
    p[i] ← i
/* LU decomposition */
for int i ← 0 to n − 1 do
    maxvalue ← 0
    maxidx ← i
    for int k ← i to n − 1 do
        if abs(a[k][i]) > maxvalue then
            maxvalue ← abs(a[k][i])
            maxidx ← k
    /* Pivoting */
    if maxidx ! = i then
        Exchange p[i] and p[maxidx]
        Exchange a[i] and a[maxidx]
    /* Decomposing L and U */
    for int j ← i + 1 to n − 1 do
        a[j][i] ← a[j][i]/a[i][i]
        for int k ← i + 1 to n − 1 do
            a[j][k] ← a[j][k] − a[j][i] × a[i][k]
Return a
```

As the serial LU decomposition with partial pivoting shows, starting from the diagonal element $a[i][i]$, pivoting occurs by first finding the largest element (in absolute value) $maxvalue$ and its index $maxidx$ in column i below the diagonal. It is then ascertained that this largest element is in fact the diagonal element itself , i.e, if $maxidx$ is equal to i. If not, pivoting needs to occur by exchanging row i and row $maxidx$. In other words, if the diagonal elements are the largest (in absolute value) among elements in the corresponding columns, then no pivoting is needed.

In our case, the elements in the diagonal satisfy the above condition, since the courant number α is always smaller than 1 (Figure 22) due to numerical stability and accuracy reasons - this is the CFL condition and in fact it happens often so that in scientific computing the system matrix is dominated by the diagonal. Therefore, pivoting can be disregarded for our application.

Serial LU Decomposition without Pivoting

In the serial LU decomposition without pivoting (algorithm shown below), during the k-th iteration:

- division takes (n-k-1) arithmetic operations and elimination takes $(n-k-1)^2 \times 2$ arithmetic operations.
- If we assume that division, multiplication and subtraction each takes a unit of time, the total sequential execution time becomes:

$$T = \sum_{k=0}^{n-1}(n-k-1) + 2\sum_{k=0}^{n-1}(n-k-1)^2 = \frac{2}{3}n^3 - \frac{2}{3} - \frac{n^2}{2} - \frac{n}{6} \quad \textbf{(1)}$$

- This leads to an asymptotic runtime of $O(n^3)$
- There are six possible permutations of the indices i,j, and k which give different loop orders of LU decomposition. These can be called as the "ijk" forms.
- The "kij" and the "kji" form are immediate update algorithms in that the elements of the matrix are updated when the necessary multipliers are known. This is in contrast to other forms where update is delayed.
- The "kij" form has the best performance due to the fact that it uses the C programming language with high spatial locality since it store the matrix in a row major order. [8]

Algorithm 2. Algorithm of Serial LU factorization

```
for int k ← 0 to n − 1 rows of the matrix do
    /* Division */
    for int i ← k + 1 to n − 1 do
        a[i][k] ← a[i][k]/a[k][k]
    /* Elimination */
    for int i ← k + 1 to n − 1 do
        for int j ← k + 1 to n − 1 do
            a[i][j] ← a[i][j] − a[i][k] ∗ a[k][j]
```

Analysis And Results

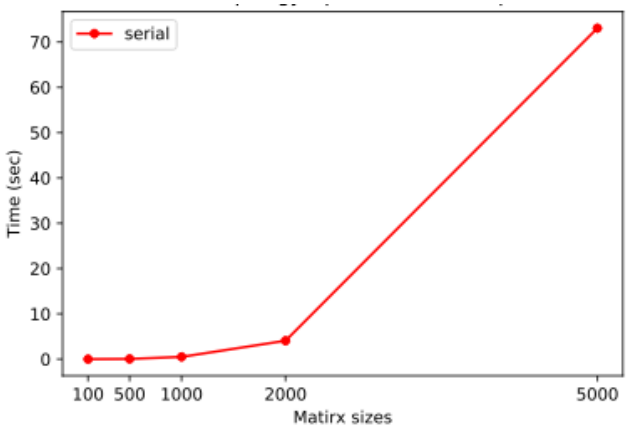

Fig. 4. Time taken by the serial version of LU decomposition.

Row Block OpenMP LU Decomposition

In the block row date distribution algorithm as show in this section, the $n \times n$ coefficient matrix is block striped among p threads or cores such that each core has n/p contiguous rows of matrix. We also use the OpenMP scheduling construct to specify the way to distribute the chunks of the matrix.

Algorithm 3. Algorithm of LU factorization assigned in row block for parallization using OpenMP

```
for int k ← 0 to n − 1 in contiguous rows of the matrix do
    /* Division */
    #pragma omp parallel for
    for int i ← k + 1 to n − 1 do
        a[i][k] ← a[i][k]/a[k][k]
    /* Elimination */
    #pragma omp parallel for
    for int i ← k + 1 to n − 1 do
        for int j ← k + 1 to n − 1 do
            a[i][j] ← a[i][j] − a[i][k] ∗ a[k][j]
```

Analysis And Results

Table 1. The time taken by different matrix sizes for 4 threads

Matirx Size	100	500	1000	2000
Time in sec	0.00126	0.0493	0.331	2.52
L1 DCA (MM)	1.95	111	701	4700
L1 DCM (MM)	0.37	0.665	3.05	31
L2 DCA (MM)	0.94	3.21	20.7	203
L2 DCM (MM)	0.0003	0.007	1.67	10.6

- We observe that OpenMP implementations of LU decomposition are generally faster than the serial versions. As the number of threads increases the time taken by the codes decreases.
- However, we observe for a 100×100 matrix as the number of threads increase the time taken by the algorithm also increases. This might be due to the overheads associated with the multi-threaded implementation, which are:
  - overhead incurred in creating the threads
  - overhead incurred in terminating/merging threads
  - overhead incurred in switching the CPU between threads (load sharing).
  - overhead in saving register state of a thread when suspending it, and restoring the state when resuming it.
  - overhead incurred in saving and restoring a thread's cache state.

Table 2. The time taken by different matrix sizes for 16 threads

Matirx Size	100	500	1000	2000
Time in sec	0.00179	0.0503	0.266	1.49
L1 DCA (MM)	1.97	109	521	2320
L1 DCM (MM)	0.04	0.607	2.27	19.6
L2 DCA (MM)	0.097	3.1	15.7	107
L2 DCM (MM)	0.0004	0.0107	1.21	3.71

In many cases, the LU factorization algorithm is simple enough that there isn't much difference in calculation speed between the RowCyclic (where the rows are distributed in a cyclic manner to each threads) and RowBlock (presented in this section) methods. Using more threads will certainly speed-up the calculation but the speed-up is not greater than 30% even if more than 4 threads are applied to the calculation.

Table 3. The time taken by different matrix sizes for 8 threads

Matirx Size	100	500	1000	2000
Time in sec	0.00144	0.0501	0.272	1.88
L1 DCA (MM)	1.96	110	534	3050
L1 DCM (MM)	0.038	0.612	2.37	230
L2 DCA (MM)	0.095	3.11	1.61	134
L2 DCM (MM)	0.0003	0.008	1.22	5.95

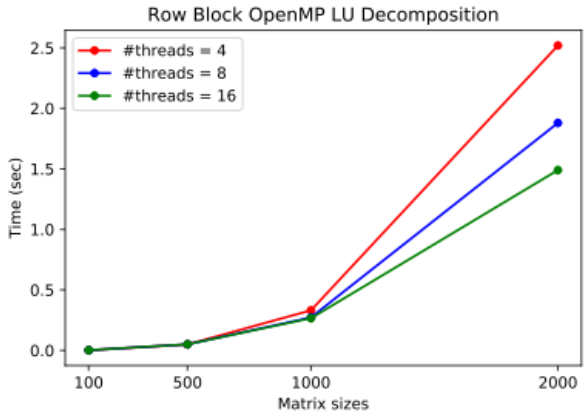

Fig. 5. OMP time taken as a function of matrix sizes.

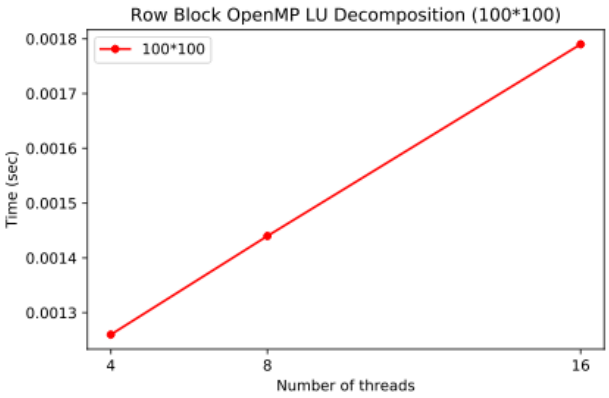

Fig. 6. OMP time taken as a function of matrix sizes and number of threads.

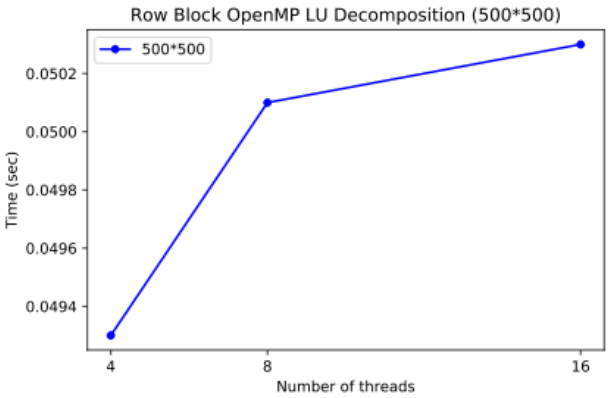

Fig. 7. OMP time taken as a function of number of threads.

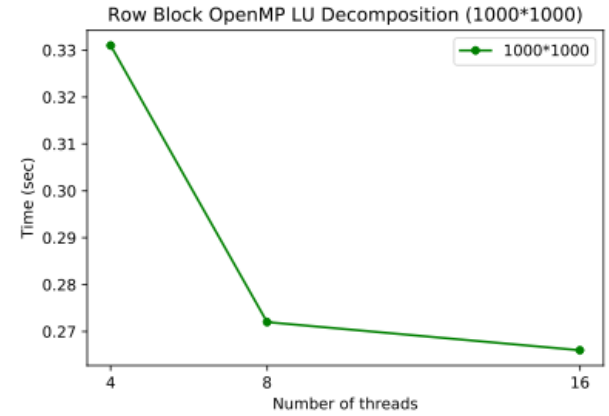

Fig. 8. OMP time taken as a function of number of threads.

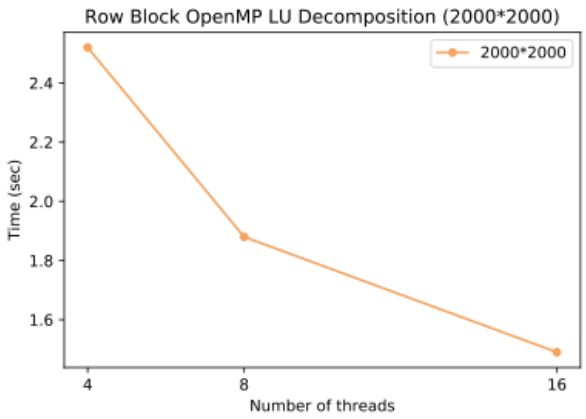

Fig. 9. OMP time taken as a function of number of threads.

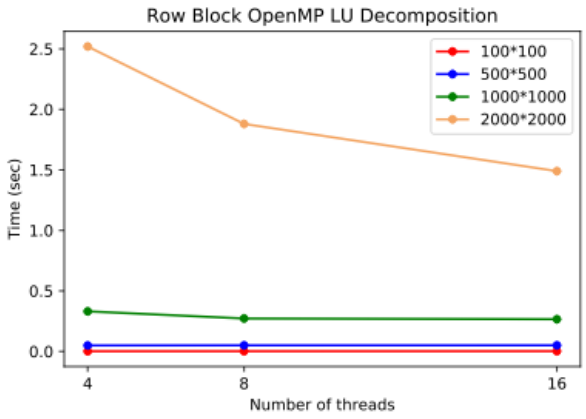

Fig. 10. OMP time taken as a function of number of threads.

OpenMP Task based Block LU Decomposition .

We implemented the right looking variant of block LU decomposition using LAPACK libraries. Suppose we have an $M * N$ matrix A that is partitioned as shown in the figure below. We aim to factorize $A = LU$, we proceed as: [5]

$$L_{00}U_{00} = A_{00} \quad (2)$$
$$L_{10}U_{00} = A_{10} \quad (3)$$
$$L_{00}U_{01} = A_{01} \quad (4)$$
$$L_{10}U_{01} + L_{11}U_{11} = A_{11} \quad (5)$$

$$\begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} = \begin{bmatrix} L_{00} & 0 \\ L_{10} & L_{11} \end{bmatrix} * \begin{bmatrix} U_{00} & U_{01} \\ 0 & U_{11} \end{bmatrix}$$

Fig. 11. Block LU decomposition.

Here A_{00} is an $r \times r$ matrix , A_{01} is an $(M-r) \times r$, and A_{11} is an $(M-r) \times (N-r)$ matrix. L_{00} and L_{11} are the lower triangular

matrices which have 1s in the main-diagonal, and U_{00} and U_{11} are two upper triangular blocks of matrices.

Equations (1) and (2) taken together perform LU factorization on the first $M \times r$ panel of A. Once this is completed , the matrices L_{00}, L_{11} and U_{00} are known. Then, the lower triangular system in Eq(3) can be solved to give U_{01}. Finally, we rearrange Eq. (4) as Schur's complement

$$A'_{11} = A_{11} - L_{10}U_{01} = L_{11}U_{11} \quad \textbf{(6)}$$

From the above equation, we learn that the problem of finding L_{11} and U_{11} reduces to finding the LU decomposition of the $(M-r) \times (N-r)$ matrix A'_{11}. This can be done by applying the steps outlined above to A'_{11} instead of A. This can be solved as shown in the algorithm below iteratively to obtain the LU decomposition of the whole matrix A. We develop an inplace algorithm where L and U are written on the same input matrix A. We further assume that our input is in column major layout and we then convert them to tile column major layout. To create multiple threads and generate tasks, we enclose the main for loop within OpenMP parallel and master pragma. This way only the master thread will generate the tasks and multiple threads can pick up the tasks for execution. The LAPACK/BLAS functions that were used are:

- **DGETRF** -computes an LU factorization of a general M-by-N matrix A using partial pivoting with row interchanges.
- **DGEMM** - performs one of the matrix-matrix operations
  $C := alpha * op(A) * op(B) + beta * C$
  where $op(X)$ is one of
  $op(X) = X$ or $op(X) = X^T$,
  alpha and beta are scalars, and A, B and C are matrices
- **DTRSM** solves one of the matrix equations
  $op(A) * X = alpha * B$ or $X * op(A) = alpha * B$ where alpha is a scalar, X and B are m by n matrices, A is a unit, or non-unit, upper or lower triangular matrix and $op(A)$ is one of
  $op(A) = A$ or $op(A) = A^T$.
- **cblas_dswap**-Exchanges the elements of two vectors (double precision)
- **cblas_daxpy** Computes a constant times a vector plus a vector (double-precision).

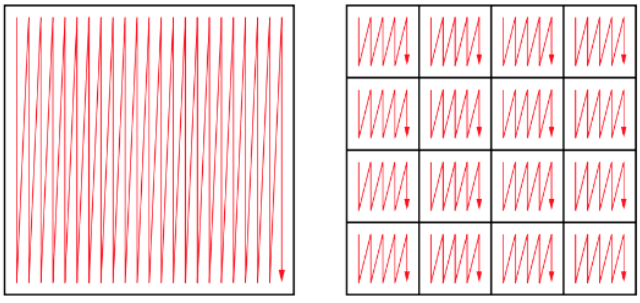

Fig. 12. Column major data layout and tile layout.

Algorithm 4. Algorithm of OpenMP Task based Block LU Decomposition with partial pivoting

```
Lapack convert column layout to tile layout
#pragma omp parallel
#pragma omp master
  for k ← 0 to numtiles do
    #pragma omp task depend(inout: A(0,k)[M × NB]) \
      depend(out: piv[k × NB : NB])
    Lapack LU decompostion
    for j ← k + 1 to numtiles do
      #pragma omp task depend(in: A(0,k)[M × NB]) \
        depend(in: piv[k × NB : NB]) \
        depend(inout: A(0,j)[M × NB]
      Lapack line swaps
      Lapack triangular solve
      Lapack Schur's complement
  for t ← 1 to numtiles do
    #pragma omp task depend(in: piv[(nt − 1) × NB : NB]) \
      depend(inout: A(0,k)[M × NB])
    Lapack line swaps
Lapack convert tile layout to column layout
```

MPI LU Decomposition.

This MPI implementation works without "master" and "slave" processes. The matrix is generated outside the MPI region, so that each process has a copy of the matrix available. Each process computes their respective bundle of rows and afterwards, broadcasts using MPI_Bcast those updated rows to all other processes. For successful MPI computations, the matrix must be stored in a memory contiguous fashion.
Our MPI implementation first generates the following matrix:

$$\begin{bmatrix} 1 & 1 & 1 & 1 & \dots & 1 \\ 1 & 2 & 2 & 2 & \dots & 2 \\ 1 & 2 & 3 & 3 & \dots & 3 \\ 1 & 2 & 3 & 4 & \dots & 4 \\ 1 & 2 & 3 & 4 & \dots & 5 \\ & & \cdots\cdots\cdots & & & \end{bmatrix}$$

This is a matrix whose decomposition is unique and the L and U portions are given as:
L=

$$\begin{bmatrix} 1 & 0 & 0 & 0 & \dots & 0 \\ 1 & 1 & 0 & 0 & \dots & 0 \\ 1 & 1 & 1 & 0 & \dots & 0 \\ 1 & 1 & 1 & 1 & \dots & 0 \\ 1 & 1 & 1 & 1 & \dots & 1 \end{bmatrix}$$

U=

$$\begin{bmatrix} 1 & 1 & 1 & 1 & \dots & 1 \\ 0 & 1 & 1 & 1 & \dots & 1 \\ 0 & 0 & 1 & 1 & \dots & 1 \\ 0 & 0 & 0 & 1 & \dots & 1 \\ 0 & 0 & 0 & 0 & \dots & 1 \end{bmatrix}$$

Algorithm 5. Algorithm of MPI LU Decomposition

```
for int i ← 0 to n − 2 in do
    for int j ← i + 1 to n − 1 do
        if j%nprocs == rank then
            if a[j][i] ≠ 0 then
                a[j][i] ← a[j][i] / a[i][i]
                for int k ← i + 1 to n − 1 do
                    a[j][k] ← a[j][k] − a[j][i] × a[i][k]
    for j ← i + 1 to n − 1 do
        Bcast(a[j][i], n − i, type, j%nprocs, comm)
```

Analysis And Results

Table 4. The time taken by different matrix sizes for 4 processes

Matirx Size	100	500	1000	2000
Time in sec	0.0371	0.586	04.32	28.2
L1 DCA (MM)	13.3	589	4408	27914
L1 DCM (MM)	0.538	20.6	158	655
L2 DCA (MM)	0.58	34.8	289	1170
L2 DCM (MM)	0.035	7.66	789	455

Table 5. The time taken by different matrix sizes for 8 processes

Matirx Size	100	500	1000	2000
Time in sec	0.0511	0.906	6.83	42.5
L1 DCA (MM)	18.8	824	6414	36619
L1 DCM (MM)	0.63	26.4	205	969
L2 DCA (MM)	0.676	46	377	1527
L2 DCM (MM)	0.056	10.2	96.3	515

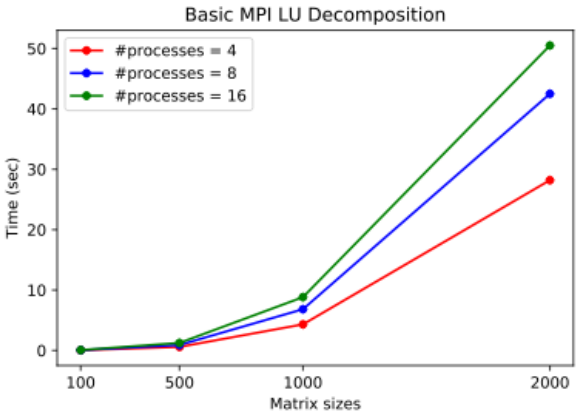

Fig. 13. MPI time taken as a function of matrix sizes

Table 6. The time taken by different matrix sizes for 16 processes

Matirx Size	100	500	1000	2000
Time in sec	0.0674	1.25	8.83	50.5
L1 DCA (MM)	30.1	1052	7986	43700
L1 DCM (MM)	0.761	34.1	262	1169
L2 DCA (MM)	0.815	58.7	472	1749
L2 DCM (MM)	0.077	11.4	109	560

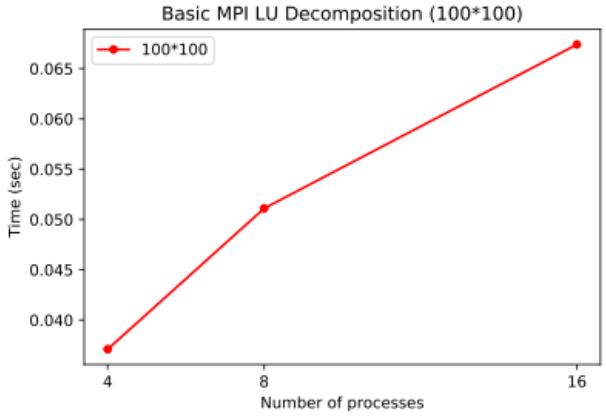

Fig. 14. OMP time taken as a function of number of processes

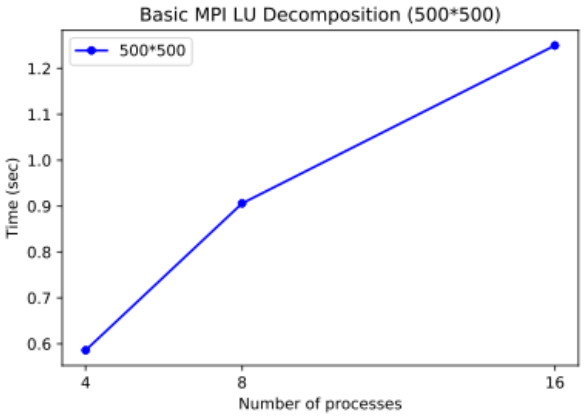

Fig. 15. OMP time taken as a function of number of processes

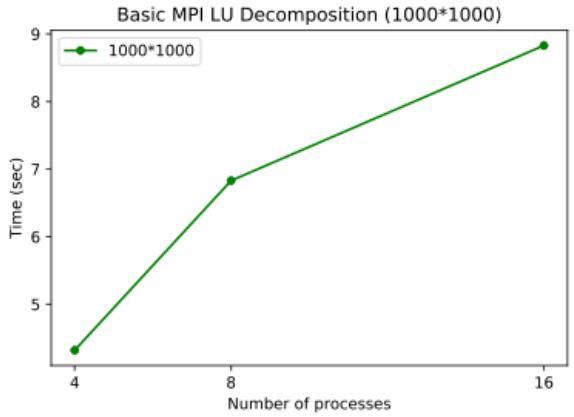

Fig. 16. OMP time taken as a function of number of processes

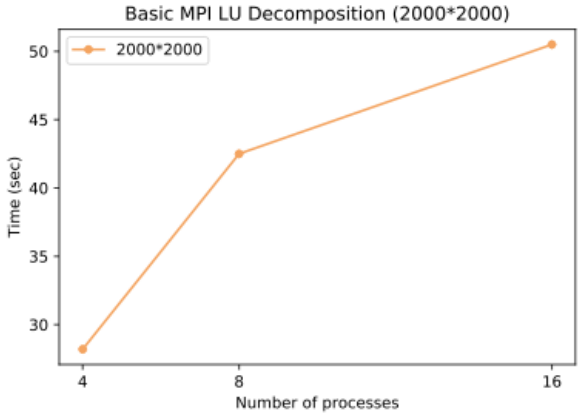

Fig. 17. OMP time taken as a function of number of processes

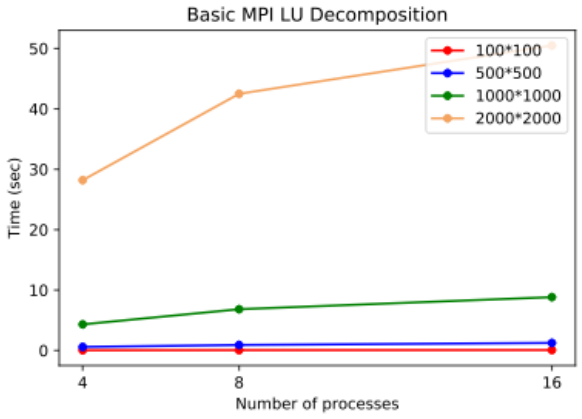

Fig. 18. OMP time taken as a function of matrix sizes

The results show that both computation time and L1 or L2 cache misses and cache accesses increase with increasing the number of processes, for all tested matrix sizes. This seems unexpected at first, since intuitively at least computation time is expected to decrease as the number of processes increase that split the computation work among each other. However, looking at the logic of this naive implementation, it becomes apparent that the problem lies within the excessive communication through the MPI_Bcast calls. MPI_Bcast starting at any element $a[j][i]$sends all remaining (updated) elements within the given row j to all other processes; this call itself is again embedded in the j loop that iterates through the row indexes and the outer i-loop. Therefore, the dominating term of this communication has complexity $O(n^3)$, which is as complex as a serial LU decomposition itself. On top of the communication, each process performs its respective computation workload too. The overall computation gets completely consumed by the communication cost and this grows even more as the number of processes increase (more processes to broadcast to and from), therefore the performance exacerbates.

2D Cartesian Topology with OpenMP LU decomposition

We went on to implement the LU decomposition on 2D Cartesian grid. The algorithm used is presented below [6] . This algorithm works only if we have a square matrix and the number of processes or ranks is a perfect square. So the 2D grid that is created has an equal number of rows and columns.

- The square matrix is generated and initialized on all the processes, then boundaries or tiles are created. Each process then works on its own portion of the tile.
- The updated matrix rows from each processes is sent to the process that needs these matrix rows to perform elimination for LU. This is done by MPI_Send and MPI_Recv invoked by each of the process.
- Elimination is performed on each processes using OpenMP.

Analysis And Results

Table 7. The time taken by different matrix sizes (NxN) for the Cartesian Implementation

	Time(sec)				
Matirx Size	100	500	1000	2000	5000
4 cores	0.006	0.075	3.014	28.06	726.85
16 cores	0.005	0.036	0.466	8.966	215.936
25 cores	0.318	0.403	0.746	2.631	500.34
64 cores	N.A	0.2	0.531	1.46	60.9

- From the Tables and the plots shows in this section we can conclude that for smaller matrix sizes (upto 500×500) our Cartsian topology implementation using MPI and OpenMP is slower than than the serial version. This might be due to the communications overhead associated with MPI processes.
- As we increase the number of processes (which must be a perfect square) the time taken for lhe LU decomposition decreases almost exponentially. However, only the largest matrix sizes show the maximum decrease in time.
- The experiment with 64 cores and a 100x100 matrix did not run on the campus cluster.

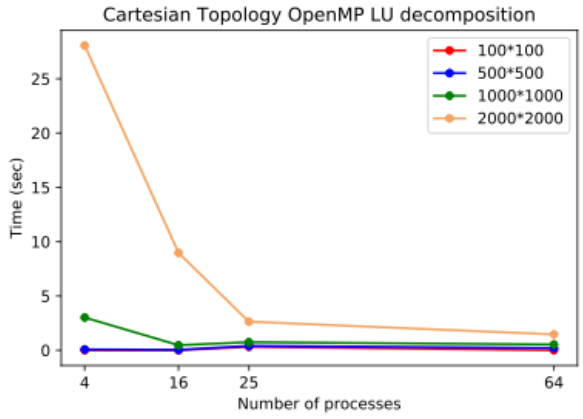

Fig. 19. The time taken by the Cartesian topology OpenMP implementation as a function of number of processes

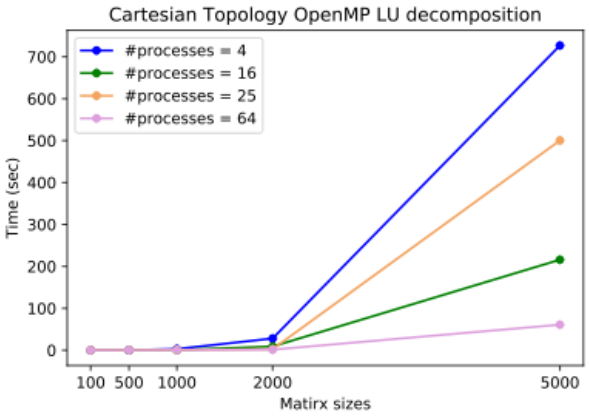

Fig. 20. The time taken by the Cartesian topology OpenMp implementaion as a function of matrix sizes

Applications of LU Factorization

In this section we apply the LU factorization algorithms developed above to solve a system of system matrix of a discretized passive scalar transport equation.The motivation of this study is to solve an advection problem of a passive scalar distribution over a 2D square domain using a finite difference discretization of the governing equations. The partial differential equation that governs the physics is as follows:

PDE Scalar Transport:

$$\frac{\partial C}{\partial t} + u_x \frac{\partial C}{\partial x} + u_y \frac{\partial C}{\partial y} = 0 \tag{7}$$

This discretized using second order central difference schemes for the convective terms and a first order implicit time stepping scheme for the time derivative results in:

Finite Difference Discretized Equation:

$$\frac{C_{i,k}^{(n)} - C_{i,k}^{(n+1)}}{\Delta t} + u_x \frac{C_{i-1,k}^{(n+1)} - C_{i+1,k}^{(n+1)}}{2\Delta x} + u_y \frac{C_{i,k-1}^{(n+1)} - C_{i,k+1}^{(n+1)}}{2\Delta y} = 0 \tag{8}$$

Assuming a homogeneous grid spacing and the same constant transport velocity in both directions, this can be rewritten into an implicitly solvable form:

Implicit form:

$$C_{i,k}^{n} = C_{i,k}^{n+1} + \alpha(C_{i+1,k}^{n+1} - C_{i-1,k}^{n+1}) + \alpha(C_{i,k+1}^{n+1} - C_{i,k-1}^{n+1}) \tag{9}$$

where: $\alpha = C\frac{\Delta t}{2\Delta x}$. The application of this implicit difference equation demonstrated on a small 4-by-4 grid point square domain with periodic boundary conditions, as depicted below:

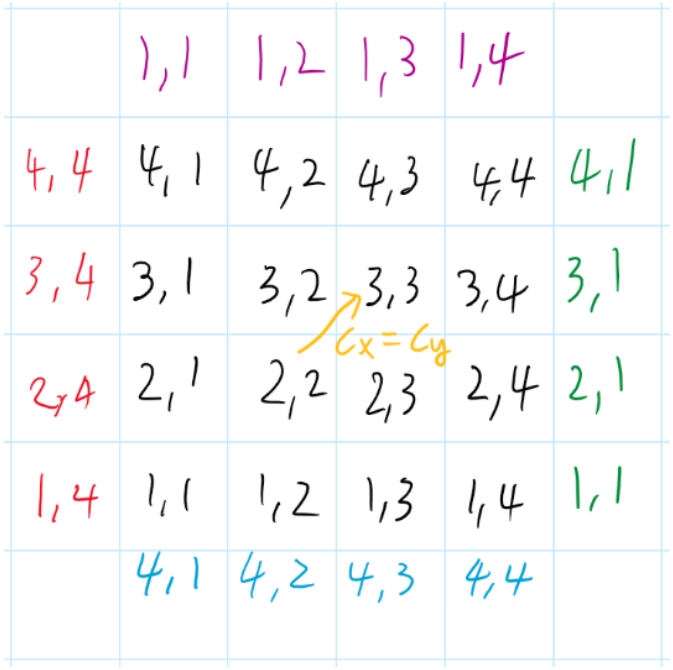

Fig. 21. 4-by-4 domain with periodic boundary conditions.

Eq. 9 results in the following linear time invariant system:

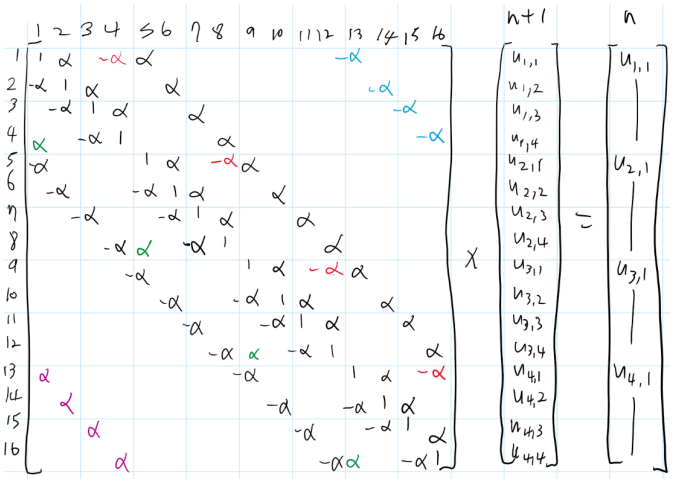

Fig. 22. Linear time invariant system of equations

To get the updated solution vector at time (n+1), one must in principle invert the system matrix A. The finer the discretization of the domain, the larger the system matrix will become

and increases the computational cost of solving the matrix. The LU decomposition is the most common way to avoid the costly explicit computation of the inverse and the parallelized LU decomposition has the potential to reduce the computational cost even further.
We used the above mentioned algorithms to decompose the finite difference matrix that was generated. Then we went on to solve the linear system of equations using a serial code whose algorithm is presented below.

Algorithm 6. Algorithm of Solving Linear Equations Sequentially

/* Given decomposed matrix a, where $a = (L - E) + U$, solve $ax = b$ */
/* Solve $Ly = b$ */
Declare a vector y with length n
$y[0] \leftarrow b[0]$
for *int* $i \leftarrow 1$ to $n - 1$ **do**
  $temp \leftarrow 0$
  for *int* $j \leftarrow 0$ to $i - 1$ **do**
    $temp \leftarrow temp + a[i][j] \times y[j]$
  $y[i] = b[i] - temp$
/* Solve $Ux = y$ */
Declare a vector x with length n
$x[n-1] \leftarrow y[n-1]/a[n-1][n-1]$
for *int* $i \leftarrow n - 2$ to 0 **do**
  $temp \leftarrow 0$
  for *int* $j \leftarrow i + 1$ to $n - 1$ **do**
    $temp \leftarrow temp + a[i][j] \times x[j]$
  $x[i] \leftarrow (y[i] - temp)/a[i][i]$
Return x

This is a 2D corner transport advection simulation applying parallel OpenMP LU decomposition and an OpenMP LU solver. The following figure is a 95x95 domain, which requires a LU decomposition for a matrix with approximately 81.5 million (95**4) elements (diagonally dominant and mostly zero). This simulation had 95 timesteps to simulate a full period, and the calculation took 387.8 seconds. The a calculation required LU decomposition of the matrix 1 time and LU solver 95 times with 8 threads. A 25% improvement compared to the serial code took 561.4 seconds.

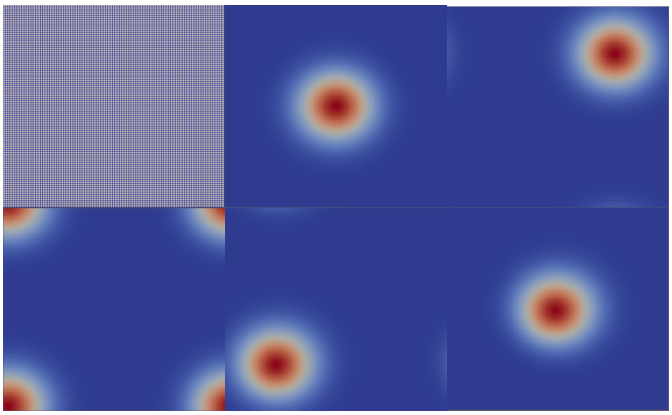

Fig. 23. Implicit difference equation demonstrated a small 4-by-4 grid.

Algorithm 7. Algorithm of Solving Linear Equations using OpenMP

/* Given decomposed matrix a, where $a = (L - E) + U$, solve $ax = b$ */
Declare vectors y and x with length n
$y[0] \leftarrow b[0]$
/* Solve $Ly = b$ */
for *int* $i \leftarrow 1$ to $n - 1$ **do**
  $temp \leftarrow 0$
  #pragma omp parallel if($i > 400$) shared($y, a, temp$)
    #pragma omp for reduction($+ : temp$) schedule(static)
    for *int* $j \leftarrow 0$ to $i - 1$ **do**
      $temp \leftarrow temp + a[i][j] \times y[j]$
  $y[i] = b[i] - temp$
$x[n-1] \leftarrow y[n-1]/a[n-1][n-1]$
/* Solve $Ux = y$ */
for *int* $i \leftarrow n - 2$ to 0 **do**
  $temp \leftarrow 0$
  #pragma omp parallel if($N - i + 1 > 400$) shared($x, a, temp$)
    #pragma omp for reduction($+ : temp$) schedule(static)
    for *int* $j \leftarrow i + 1$ to $n - 1$ **do**
      $temp \leftarrow temp + a[i][j] \times x[j]$
  $x[i] \leftarrow (y[i] - temp)/a[i][i]$
Return x

Analysis And Results

- The 55x55 domain was simulated for 55 timesteps and was performed from single to 16 threads applying parallel LU decomposition. The speed-up was seen gradually up to 8 threads and then dropped significantly when it ran on 16 threads. The problem size may have been too small for 16 threads to overcome overhead cost over computation time.

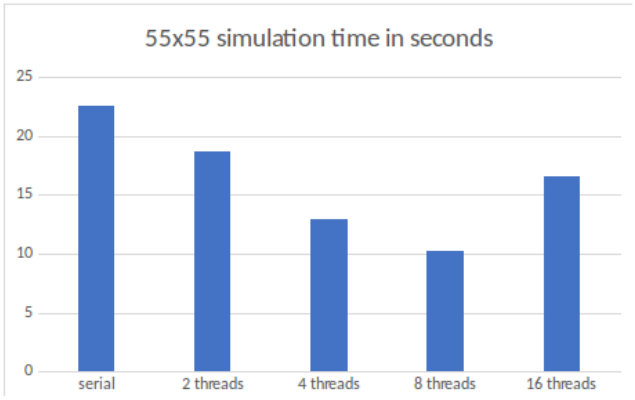

Fig. 24. 55x55 simulation time in seconds.

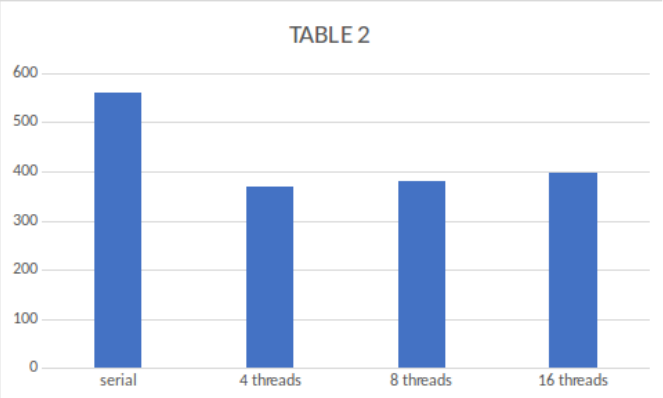

Fig. 25. 95x95 simulation time in seconds.

- So 95x95 domain was tested for multiple threads. In the larger problem, the increase in performance was not seen in each of parallel case, but similar performance improvement of about 25% compared to the serial code. The row block parallelization method is known to improve about 30% compared to serial even if there are large number of threads available. [8].
- To further improve the performance, parallel LU solver was implemented. However, no improvement in performance was noticed. This can be explained by the fact that for 55x55 domain, LU decomposition costs 20.9 seconds and LU solver takes 3.3 seconds; and for 75x75 domain, they take 133.8 seconds and 14.6 seconds, respectively. So, LU decomposition takes 7 10 times more than LU solver; therefore, we should not expect much improvement in performance by parallelizing the inner loops of the LU solver (cannot parallelize the outer loop due to dependency).

ACKNOWLEDGMENTS

Many thanks to Prof. Mark Snir for his guidance and patience while teaching this course

REFERENCES

1. P. Hohenberg and W. Kohn, “Inhomogeneous electron gas,” Phys. Rev., vol. 136, pp. B 864-B871, Nov 1964.
2. S. Li, W. Wu, and E. Darve, “A fast algorithm for sparse matrix computations related to inversion,” Journal of Computational Physics, vol. 242, pp. 915–945, 2013.
3. J. M. Tang and Y. Saad, “A probing method for computing the diagonal of a matrix inverse,” Numerical Linear Algebra with Applications, vol. 19, no. 3, 2012
4. https://en.wikipedia.org/wiki/OpenMP accessed on January 14th, 2018.
5. https://github.com/hitchpy accessed on January 10th, 2018.
6. https://github.com/vraja2/parallel-LU-decomposition
7. http://www.nersc.gov/users/software/programming-models/openmp/openmp-tasking/ accessed on accessed on January 14th, 2018 .
8. P. D. Michailidis, K. G. Margaritis, "Implementing Paralle LU Factorizationwith Pipeliningon a multicore using OpenMP", 13th IEEE International Conference on Computational Science and Engineering, 2010.