Journal of University of Duhok, Vol. 20, No.1 (Pure and Eng. Sciences), Pp 46-53, 2017
DOI: https://doi.org/10.26682/sjuod.2018.20.1.5

PARALLELIZE AND ANALYSIS LU FACTORIZATION AND QUADRANT INTERLOCKING FACTORIZATION ALGORITHM IN OPENMP

DELBRIN H. AHMED and NAEEM A. ASKAR
Dept. Of Mathematics, College Of Basic Education, University of Duhok, Kurdistan Region-Iraq

(Received: June 13, 2017; Accepted for publication: August 29, 2017)

ABSTRACT

Recent developments in high performance computer architecture have a significant effect on all fields of scientific computing. the solution of linear systems of equations lies at the heart of many applications in scientific computing. This paper describes, compare and analyzes the parallel LU factorization and QIF Factorization methods that are used in linear system solving on a multicore using OpenMP interface. In our work, illustrate that the QIF Algorithm performs better in performance compared to LU Factorization algorithm as it takes a small step to solve the problem.

1. INTRODUCTION

Lower Upper factorization (LUF) and Quadrant Interlocking Factorization (QIF) are methods used to solve systems of linear equations. LU algorithm involves a back substitution process while QIF involve bidirectional substitution. The study of both methods using parallel algorithm is to gain the performance of parallelism.

We are requested to solve linear equations for x given *A* and *B* where *A* is a $(n \times n)$ nonsingular matrix, x is the unknown vector and *B* is the right-hand side vector.

1.1 LU Factorization

Generally, LU Factorization can be described as follows:

Step1: LU Decomposition

Let *A* be a square matrix. An LU decomposition converts A into two matrices L and U where A = LU, and where L and U are lower and upper triangular matrices (of the same size), respectively. This means that L has only zeros above the diagonal and U has only zeros below the diagonal. For a 3X3 matrix, this becomes:

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

AX=B then becomes (LU)x = B, this equation can be rewritten as L(Ux) = B.

Step2: Forward Substitution

Next, solve lower triangular matrix Ly= B where y = Ux, L and B are known using forward substitution to obtain vector y.

$$\begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

This process is so called because for lower triangular matrix, one first computes y_1, then substitutes that forward into the next equation to solve for y_2, and repeats through y_n.

Step3: Backward Substitution

Last, solve upper triangular matrix Ux = y where U and y are known using backward substitution
to obtain solution *x*.

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

In an upper triangular matrix, one works *backwards*, first computing x_n , then substituting that that *back* into the *previous* equation to solve for x_{n-1}, and repeating through x_1.

1.2 Theorem QIF Factorization

Generally, QIF Factorization can be described as follows:

Step1: WZ Factorization

Let A be a square matrix. W and Z is decomposes by matrix A into interlocking quadrant factors of butterfly
form. For a $3X3$ matrix, this becomes:

$$\begin{bmatrix} a_{11} & a_{1i}^T & a_{1n} \\ a_{i1} & A_{n-2} & a_{in} \\ a_{n1} & a_{ni}^T & a_{nn} \end{bmatrix} = \begin{bmatrix} 1 & & 0 \\ w_{i1} & W_{n-2} & w_{in} \\ 0 & & 1 \end{bmatrix}\begin{bmatrix} z_{11} & z_{1i}^T & z_{1n} \\ & Z_{n-2} & \\ z_{n1} & z_{ni}^T & z_{nn} \end{bmatrix}$$

This process will solve for the elements that do not contain value 0 or 1. It begin by solving which row $z_1$and row $z_n$in matrix Z in order to solve column W_1 and column W_1 in matrix W, then continues substitutes that inwards Z and inwards W until reach to the center.

This solution represents a series of $(2\ x\ 2)$ linear systems and we used Cramers Rule(For more information on Cramers Rule see [6]) to solve it in order to reduce matrix A_{n-2}.

$Ax = B$ then becomes (WZ)x = B, this equation can be rewritten as W(Zx) = B.

Step2: Bi-directional Substitution

Next, solve lower triangular matrix Wy = B where y = Zx, W and B are known using forward
substitution to obtain vector y.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ w_{2,1} & 1 & 0 & 0 & 0 & 0 & w_{2,n} \\ \vdots & w_{3,2} & 1 & & 0 & w_{3,n-1} & \vdots \\ & \vdots & & \ddots & & \vdots & \\ \vdots & w_{n-2,2} & 0 & & 1 & w_{n-2,n-1} & \vdots \\ w_{n-1,1} & 0 & 0 & 0 & 0 & 1 & w_{n-1,n} \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ \\ \vdots \\ y_{n-1} \\ y_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ \\ \vdots \\ b_{n-1} \\ b_n \end{bmatrix}$$

This process begins from the top and bottom moving inwards b_i-directionally and each time substituting
for two values simultaneously.

Step3: Bi-directional Solution

Last, solve upper triangular matrix Z_x = y where Z and y are known using backward substitution
to obtain solution *x*.

$$\begin{bmatrix} z_{1,1} & z_{1,2} & \cdots & \cdots & \cdots & z_{1,n-1} & z_{1,n} \\ 0 & z_{2,2} & \cdots & \cdots & \cdots & z_{2,n-1} & 0 \\ 0 & 0 & \ddots & & & 0 & 0 \\ 0 & 0 & & \ddots & & 0 & 0 \\ 0 & 0 & & & \ddots & 0 & 0 \\ 0 & z_{n-1,2} & \cdots & \cdots & \cdots & z_{n-1,n-1} & 0 \\ z_{n,1} & z_{n,2} & 0 & 0 & 0 & z_{n,n-1} & z_{n,n} \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ \\ \vdots \\ y_{n-1} \\ y_n \end{bmatrix}$$

This procedure continues outwards from center forwards and backwards

1.3 Parallel Design

In the LU, we only parallel some part of the equation that consist lots of loop. From the algorithm, the most loops will appear in calculation of the lower triangle and upper

DOI: https://doi.org/10.26682/sjuod.2018.20.1.5

triangle. The loops consist of three nested for loops as shown below (the program language used here is C++).

```
for () //--most outer loops cannot be parallel
{
//--parallel this section
for()
{
for()
{ }
}
//--parallel this section
for()
{
for()
{ }
}
}
```

Fig. (1): -Loops fraction in sequential LU Factorization program

From the figure 1 above, we can see two of the second for loops can be parallel without producing a incorrect answer (data are not dependable at this level), while for the most outer for loops cannot be parallel because the data becoming dependable with each other. There are others for loops in the algorithm but most of it is only one of two nested loops which do not give very much effect to the execution time. Furthermore, parallel the less nested loops can cost an overhead to the algorithm without given sufficient speed up.

For the case of QIF method, we also parallel the same part of the algorithm that is the calculation to determine the W matrices and Z matrices and can be visual as figure above. The implantation is somewhat almost the same but only has less iteration of a factor 0.5 due to the algorithm use that can calculate two values at one loops.

2. DESIGNING USING OPENMP

Designing with OpenMP API (application program interface) in the algorithm for LU and QIF will use the same implementation as discusses before. In this section we are going to see how the OpenMP is use to parallel LU and QIF algorithm. the coding of the algorithm are done using Microsoft Visual Studio 2010 Professional with personal laptop.

2.1 LU

Implement OpenMP in LU are easier than MPI (MPI is a message-passing application programmer interface) and Pthread due to well develop API that ease the parallel programming. By using the #pragmaomp parallel for at the parallel part of the programming, the portioning of the work are done automatically. Below shows how the OpenMP is written in the algorithm for LU.

```
for (iterate for n times)
{
#pragmaomp parallel forprivate (sum1)
for()
{
for()
{ //--do summation sum1 here }
//--update Upper Triangle here
}
#pragmaomp parallel forprivate (sum1)
for()
{
for()
{//--so summation sum1 here }
//-- update Lower triangle here
}
}
```

Fig.(2):- LU algorithm with OpenMP

As we can see from figure 2, there are two #pragmaomp parallel for used with the private attribute of sum1.The sum1 is variable declare globally that some of the thread created may refer to the same address. By set the sum1 to become private, each thread created will have its own sum1 address. The total iteration at most outer loops will be n times where n is the dimension of the matrix.

2.2 QIF

For QIF method, the coding of the algorithm will be look more than the same as LU which implement the #pragmaomp parallel for at the calculation to determine the matrix Z and matrix W.

```
for (iterates for n/2 times)
{
#pragmaomp parallel forprivate (sum1,sum2)
for()
{
for()
{ //--do summation sum1,sum2 here }
//--update Matrix Z here
}
#pragmaomp parallel forprivate (sum1,sum2)
for()
{
for()
{//--so summation sum1,sum2 here }
//-- update Matrix W here
}
}
```

Fi.(3):- QIF with OpenMP

Figure 3 above shows that the each #pragmahad aprivate (sum1,sum2)which due the algorithm of QIF that can calculate the two matrix values at the same time. This lowers the iteration process for each calculation and produces more throughputs. The total iterations for the most outer loops will be only n/2 where n is the dimension of the matrix.

3. Parallel Performance

The purpose of parallelizing the algorithm is to enhance its performance. Methods of analysis for our parallel design are based on Amdahl's law and Gustafson's law.

Amdahl's Law: The existence of non-parallelizable (sequential fraction) computations limits the potential benefit of parallelization.

Gustafson-Barsis Law: Problems with large, repetitive data sets can be efficiently parallelized. It showed that Amdahl's law is invalid for cases where the problem size could be increase and the regularity of the problem could be used to deploy as many processors as the problem needed.

Speedup

Speedup S_p is defined with formula:

$$S_p = \frac{T_s}{T_p}$$

Where T_s the execution time for sequential algorithm is, T_p is the execution time of the parallel
algorithm with p processors.

Efficiency

Efficiency $E_p$is defined with:

$$E_p = \frac{S_p}{p}$$

Where S_p is the speedup with two processors.

Total Parallel Overhead

Total parallel overhead T_o is defined with formula:

$$T_o = pT_p - T_s$$

Where T_s the execution time for sequential algorithm is, $T_p$is the execution time of the parallel
algorithm with p processors.

4. RESULT

We performed testing for sequential program of both LU Factorization and QIF. The objective of this testing is to measure the execution time for sequential implementation. Theoretically, QIF can solve the problem faster than LU Factorization because it takes less computational steps for solving the linear equation. In testing sequential programs, we decided to measure the execution time versus the size of the linear system (n). The actual size of data set is nxn, a 2D matrix. The range of the linear system size is starting from 100 to 1000 as we focus on the low bound of linear system size. We started with 100 size of linear system and then later increase the size in time and record the execution time was measured in second.

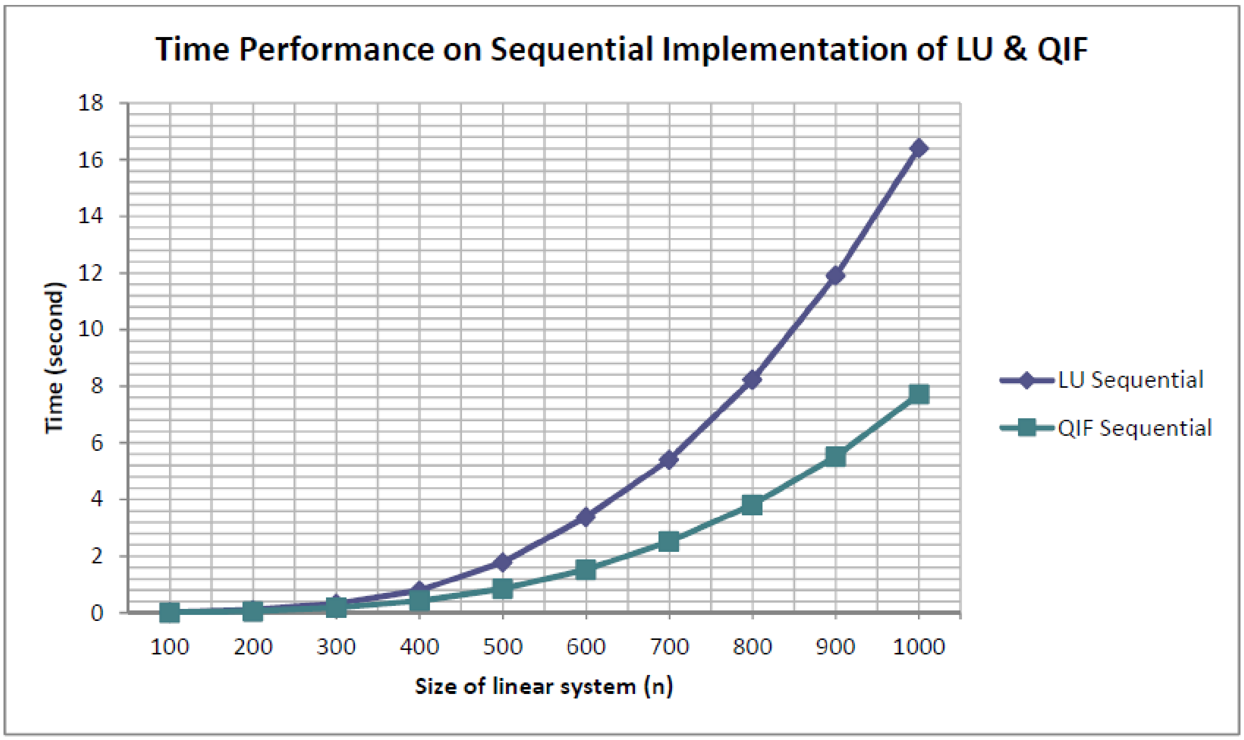


Figure 4: Time Performance on Sequential Implementation of LU Factorization & QIF

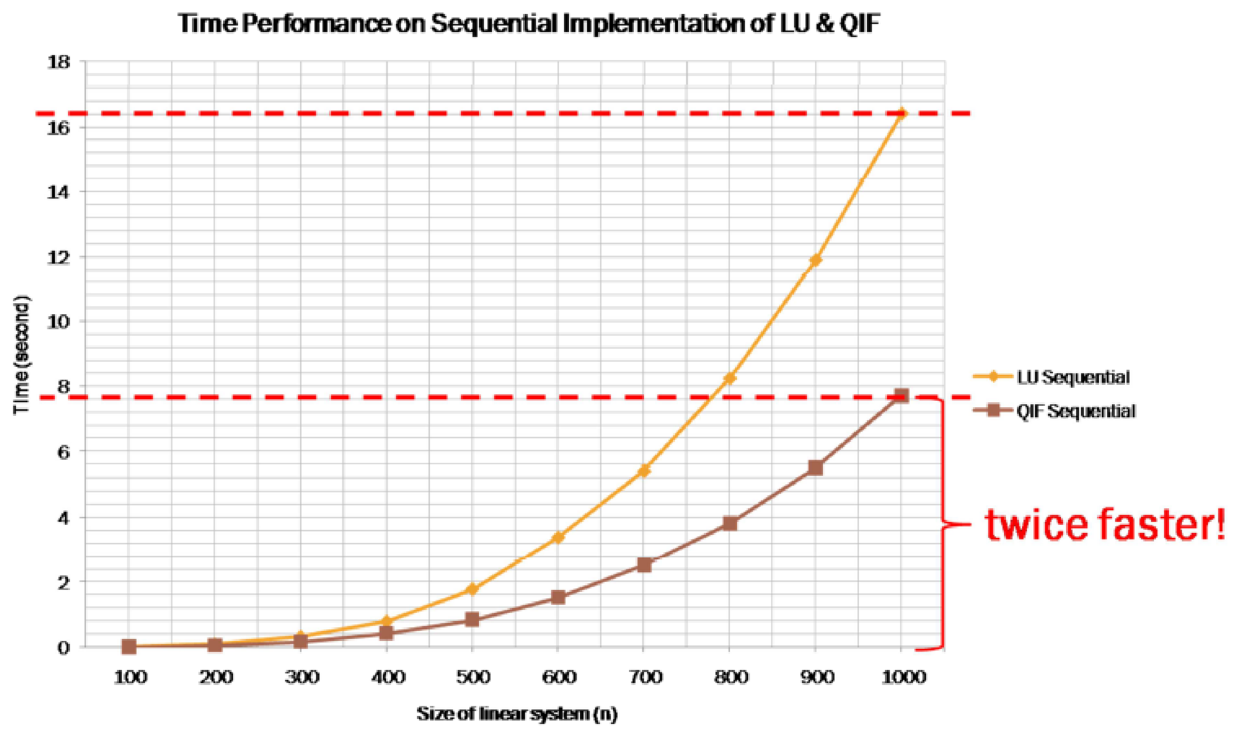


Figure 5: QIF is half faster than LU

Figure 4 and 5 show the time performance of sequential of LU Factorization and Quadrant Interlocking

Factorization (QIF). As we expected, two different algorithms which are LU Factorization and Quadrant Interlocking Factorization (QIF) produced different result. In overall, sequential QIF executed faster than sequential LU Factorization as the size of the linear system increased. This occurred because of the complexity of the algorithms implemented on the programs.

The complexity of the QIF algorithm is less complex thus produce larger throughput compare to LU Factorization. QIF algorithm was solved in factor of four within the loops. We can conclude that QIF Algorithm performs better in performance compared to LU Factorization algorithm as it takes a small step to solve the problem. In addition, the performance of execution time is depended on the algorithm implemented on the program.

Time performance for OpenMP implementation of the algorithms while Figure 6 below is the time performance graph plotted. As we expected, the OpenMP version of QIF has better execution time compared to LU Factorization as the number of thread (N) increases.

When the number of thread increases in size, the execution time reduces. We can see that from the transition between thread 1 to 2, it shows a significant reduces in execution time. Then from the thread transition between 3 to 8 thread, the execution time reduces in time thus give better time performance. The increment of number of threads which means more workers will be available to execute the parallel thus accommodates tasks. For example if we have two threads available, the parallel fraction will be executed by both workers. As the number of workers increase, the workers can execute jobs in fraction of 1/N of the time of one worker.

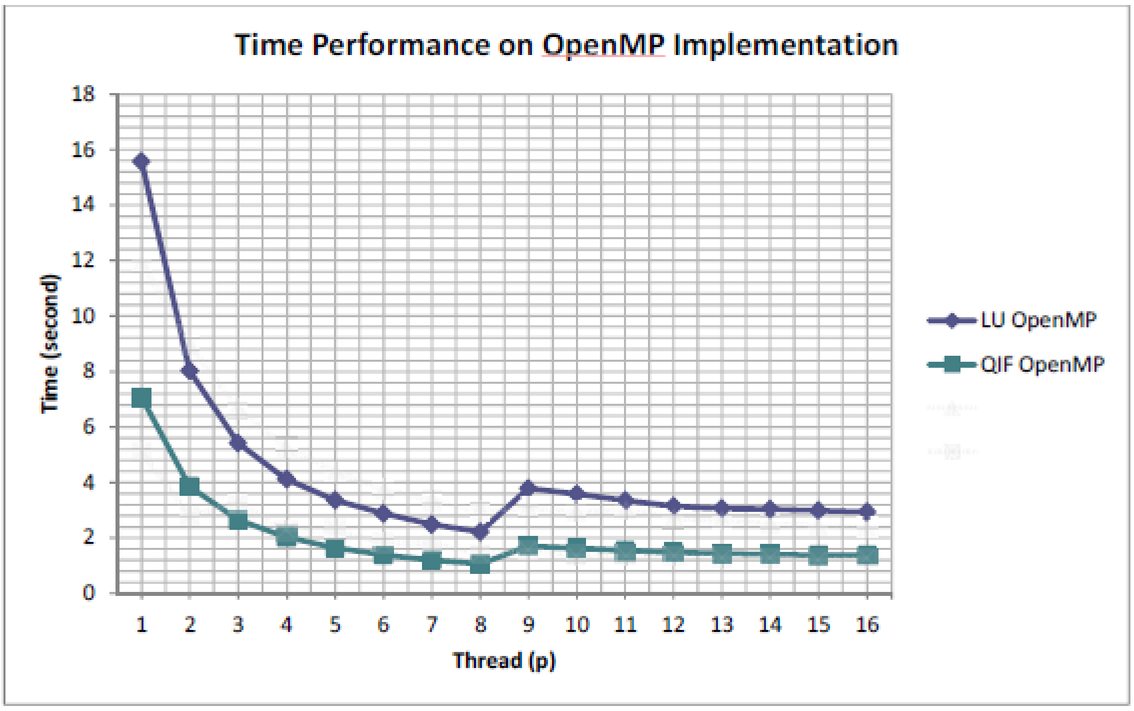


Fig.(6):- Time Performance between LU Factorization & QIF

Performance degradation can be seen from the transition between thread 8 to 9. This occurred because the number of thread exceed number of logical processors. As the number of thread increases (from thread 9 to 16), we can see a small improvement on execution time then the time performance refuses to improve and remain constant.

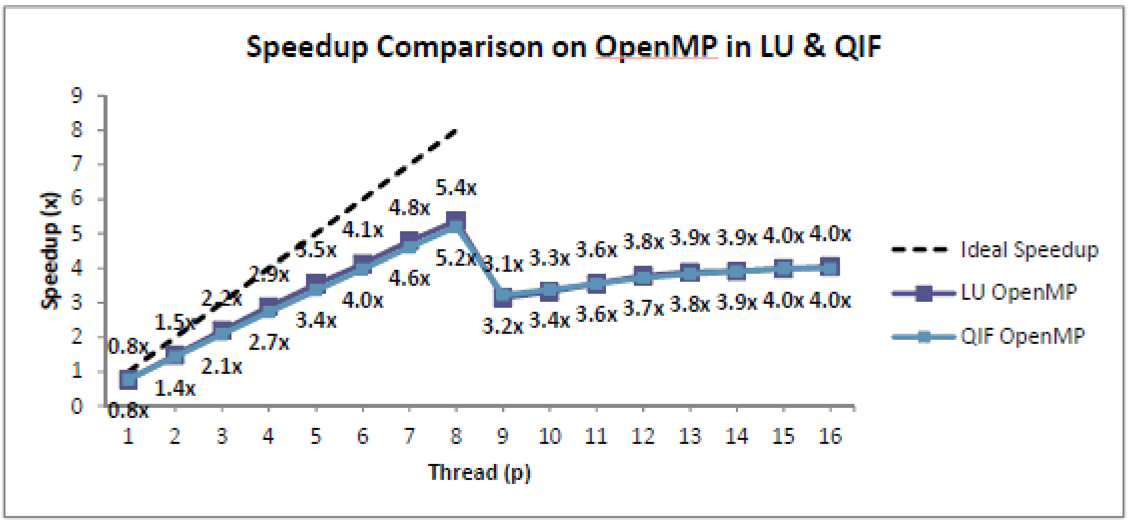
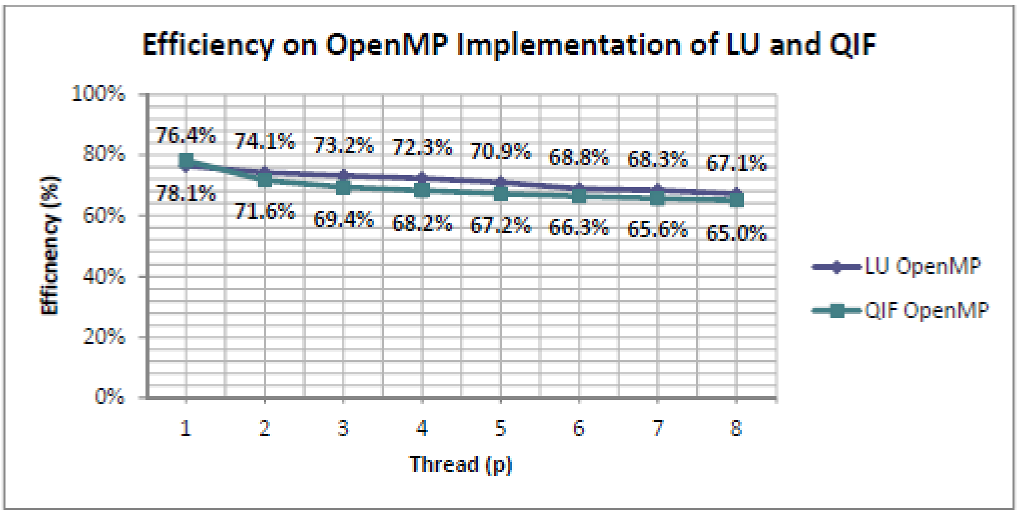


Fig.(7): -Speedup comparison on both LU and QIF

Figure 7 shows the speedup comparison on OpenMP implementation of LU and QIF. From the result, all speedup of both algorithms shows slightly linear speedup and it performs under the ideal speedup. The ideal speedup is driven by Amdahl’s law. LU and QIF algorithms both have as high as 5.4 and 5.2 respectively, using 8 logical processors, which is far below ideal speedup.

DOI: https://doi.org/10.26682/sjuod.2018.20.1.5

Then the speedup started to reduce from thread 8 to 9 as threads exceed number of logical processors. This is expected as one of the reasons is the logical processors limit the potential parallel speedup. Besides that, sequential fraction also limits the potential speedup. Ideal speedup is obtained when the parallel fraction is 100%, which means that the parallel fraction is the whole program. In another words, there is no sequential fraction in your program as it nearly impossible to do that. In order to fully benefit the parallel speedup, we need to increase the parallel fraction and reduce the sequential fraction so that we can get optimum value of parallel speedup.



Fig(8): - Efficiency comparison on both LU and QIF

Figure 8 shows the efficiency comparison between both algorithms. From our observation, the efficiency reduces as the number of threads increase as no significant degradation seen from the graph. As the result, we can prove that our speedup result (on Figure 6) is relatively true to our efficiency result since there is no significant increase observed in our speedup graph.

CONCLUSION

In this paper, we implemented the LU decomposition method and QIF Factorization using OpenMp. We implemented openMP using Dell, Operating System: Windows 7 Home Premium 64-bit Processor: Intel core i7 ,(8 CPUs), memory: RAM 4 GB. We conclude that the QIF has better execution time compared to LU Factorization as the number of thread (N) increases. When the number of thread increases in size, the execution time reduces and speedup is increased.

REFERENCE

- Abdullah, R. U. (2011). Benchmark of Parallel Bipartite Graph Matching and Parallel Smith-Waterman on the Intel® Manycore Testing Lab.
- Amdahl, G. (1967). The Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. AFIPS Conf. Proc. , 483-485.
- Asenjo, R., Ujaldon, M., & Zapata, E. (1993). Parallel WZ factorization on mesh multiprocessors. presented at Microprocessing and Microprogramming , 319-326.
- Benaini, A., & Laiymani, D. (1994). Parallel Block Generalized WZ Factorization. Proc. ICPADS , 174-181.
- Evans, D. (2006). Parallel strategies for linear systems of equations. International Journal of Computer Mathematics , 81 (4), 417-416.
- Matrices and Determinants,(2017) http://teachers.dadeschools.net/rvancol/BlitzerPrecalculusStudentBook/Chapter8/Ch8_Section5.pdf