Implementing Parallel LU Factorization with Pipelining on a MultiCore Using OpenMP

Conference Paper · December 2010

DOI: 10.1109/CSE.2010.39 · Source: DBLP

2 authors:

Panagiotis D Michailidis
University of Macedonia
85 PUBLICATIONS **428** CITATIONS
SEE PROFILE

Konstantinos G. Margaritis
University of Macedonia
424 PUBLICATIONS **3,586** CITATIONS
SEE PROFILE

Implementing Parallel LU Factorization with Pipelining on a MultiCore using OpenMP

Panagiotis D. Michailidis
University of Western Macedonia
Florina, Greece
Email: pmichailidis@uowm.gr

Konstantinos G. Margaritis
Department of Applied Informatics
University of Macedonia
Thessaloniki, Greece
Email: kmarg@uom.gr

***Abstract*—Recent developments in high performance computer architecture have a significant effect on all fields of scientific computing. Linear algebra and especially the solution of linear systems of equations lies at the heart of many applications in scientific computing. This paper describes and analyzes three parallel versions of the dense LU factorization method that is used in linear system solving on a multicore using OpenMP interface. More specifically, we present two naive parallel algorithms based on row block and row cyclic data distribution and we put special emphasis on presenting a third parallel algorithm based on the pipeline technique. Further, we propose an implementation of the pipelining technique in OpenMP. Experimental results on a multicore CPU show that the proposed OpenMP pipeline implementation achieves good overall performance compared to the other two naive parallel methods. Finally, in this work we propose a simple, fast and reasonably analytical model to predict the performance of the LU decomposition method with the pipelining technique.**

***Keywords*-Linear algebra; factorization; LU; multicore; OpenMP; parallel algorithms;**

I. Introduction

The solving of dense and large scale linear algebraic systems lies at the heart of many scientific and computational economic applications. Methods for a linear algebraic system generally fall into two categories: direct methods and iterative methods. In direct methods, the exact solution, in principle, is determined through a finite number of arithmetic operations (in real arithmetic, leaving aside the influence of round-off errors). Iterative methods generate a sequence of approximations to the solution by repeating the application of the same computational procedure at each step of the iteration. A key consideration for the selection of a solution method for a linear system is its structure. Roughly speaking, direct methods are best for full (dense) matrices, whereas iterative methods are best for very large and sparse matrices.

This paper presents high performance algorithms based on the direct methods for solving a linear algebraic system. We focus on the LU factorization algorithm for the matrix factorization used to solve dense linear systems. Many parallel implementations have been studied on modern high performance systems, such as multicore and special purpose architectures, such as GPUs or the CELL BE for solving numerical computations. In recent years, Buttari et al [1][2][3] present parallel tiled linear algebra algorithms for ”standard” (x86 and alike) multicore processors with framework Parallel Linear Algebra for Multicore Architectures (PLASMA). Seminal work leading to the tile QR algorithm was done by Elmroth et al [4] [5]. Gunter et al [6] presented an ”out-of core” (out of memory) implementation. Performance results for the out-of-core computation of the QR factorization on a multi-core processor were produced by Marque et al[7]. Some work has been done on load balancing schemes for the LU factorization algorithm such as the paper by McGinn et al [8], which compares static and dynamic distribution schemes in OpenMP and MPI; these latter schemes were run on a non-multicore platform such as IBM RS/6000 SP.

There are a wide variety of dense linear algebra operations running on the GPU: the conjugate gradient method [9][10], the Gauss-Seidel method [9], the projected Jacobi method [10] and the direct factorization methods, such as LU factorization [11][12][13][14]. Finally, the static pipelining technique for dense matrix factorizations on the CELL processor was originally implemented by Kurzak et al [15][16] and an extensive performance study of popular matrix factorization methods on CELL processor was conducted by Vishwas et al [17].

Despite these research results, no attempt has been made yet to implement and analyze the performance of several parallelization strategies for the dense LU factorization method on a multicore using OpenMP interface. We implement two strategies that are based on trivial parallelizations of the original algorithm and we put special emphasis on implementing a third strategy, which is based on pipelining technique but is able to extract a much higher degree of parallelism. These strategies are compared and evaluated through performance measurements on a multicore platform. To the best of our knowledge, the key contributions of this paper are (1) the implementation of the pipelining technique in an OpenMP programming environment using the queue data structure and (2) the first OpenMP pipeline implementation and performance analysis for LU decomposition.

The remainder of the paper is organized as follows: In Section II, the Gaussian elimination and LU decomposition method is recalled. In Section III, the three parallel implementations of LU decomposition method are discussed and so is the corresponding performance model for the pipelining implementation. Performance results are given in Section IV. Finally, some conclusions are drawn in Section V.

II. Gaussian Elimination and LU decomposition

Consider the following real linear algebraic system $Ax = b$, where $A = (a_{ij})_{n\times n}$ is a known nonsingular $n \times n$ matrix with nonzero diagonal entries, $b = (b_0, b_1, ..., b_{n-1})^T$ is the right-hand side and $x = (x_0, x_1, ..., x_{n-1})^T$ is the vector of the unknowns.

Gaussian elimination can be used in the solution of a system of equations $Ax = b$. This process transforms the matrix A in a triangular form with an accompanying update of the right-hand side, so that the solution of the system $Ax = b$ is straightforward by a triangular solve. This triangular system is solved using the backward substitution algorithm [18].

LU factorization uses the same algorithm and converts the matrix A into two matrices L and U, where $A = LU$ and L and U are the lower and upper triangular, respectively. The solution to the original $Ax = b$ problem can be found by solving two triangular systems: $Ly = b$ and $Ux = y$. These systems are solved using forward and backward substitution algorithms [18]. A serial version of the LU factorization algorithm shown in Algorithm 1 consists of three nested loops, which we will adopt for parallel algorithms in the remainder of this paper. For each iteration of the outer loop, there is a division step and an elimination step. As the computation proceeds, only the lower-right $k \times k$ sub-matrix of A becomes active. Therefore, the amount of computation increases for elements in the direction of the lower-right corner of the matrix causing a non-uniform computational load. We assume that matrices L and U share storage with A and overwrite the lower-triangular and upper-triangular portions of A, respectively.

Algorithm 1: Sequential LU factorization algorithm

```
for $k \leftarrow 0$ to $n-1$ do
    /* Division step */
    for $i \leftarrow k+1$ to $n-1$ do
        $a[i][k] \leftarrow a[i][k]/a[k][k]$
    /* Elimination step */
    for $i \leftarrow k+1$ to $n-1$ do
        for $j \leftarrow k+1$ to $n-1$ do
            $a[i][j] \leftarrow a[i][j] - a[i][k] * a[k][j]$
```

In sequential LU decomposition, during kth iteration: division takes $(n-k-1)$ arithmetic operations and eliminating takes $(n-k-1)^2 \times 2$ arithmetic operations. Assuming division, multiplication and substraction each takes a unit time, the total sequential execution time is given by $T = \Sigma_{k=0}^{n-1}(n-k-1) + 2\Sigma_{k=0}^{n-1}(n-k-1)^2 = \frac{2}{3}n^3 - \frac{n^2}{2} - \frac{n}{6}$ which leads to an asymptotic runtime $O(n^3)$.

The six permutations of indices i, j and k give six different loop orders of LU decomposition, which we call "ijk" forms. The kij and kji forms are immediate update algorithms in that the elements of A are updated when the necessary multipliers are known. This is in opposition to the other forms, which are delayed update algorithms. kij and kji forms differ only in accessing matrix by row or columns, respectively. We focus on the kij form because it has good performance compared to the kji form, according to the preliminary experimental study of Table I for different matrix sizes on one core. The good performance of the kij form is due to the fact that it uses the C programming language with high spatial locality because it references the matrix in the same row-major order in which the matrix is stored. On the other hand, the low performance of kji form is due to the fact that it has poor spatial locality because it scans the matrix column by column instead of row by row and it leads to the occurrence of a higher cache miss rate in relation to the kij form.

Table I
Execution Time (in seconds) of the kij and kji forms for different matrix sizes on 1 core

	kij	kji
32	0,000080	0,000120
64	0,000310	0,000700
128	0,001540	0,005480
256	0,012760	0,049870
512	0,143320	0,544740
1024	1,218400	5,623180
2048	9,684760	43,741910
4096	77,215980	391,604810

III. Parallel Algorithms for LU Decomposition

In the following section, we discuss three OpenMP parallel algorithms for the LU factorization method. These parallel algorithms are based on the three different data distribution schemes among the available threads such as row block, row cyclic and pipeline. The first two algorithms are trivial parallelizations of the original algorithm 1, while the third algorithm is a pipeline parallelization but is able to extract a much higher degree of parallelism. We then give an implementation of the pipelining technique in OpenMP using the queue data structure. Finally, we present a performance model for the third parallel algorithm.

A. Algorithm with Row Block Data Distribution

In row block data distribution, the $n \times n$ coefficient matrix A is block-striped among p threads or cores such that each core is assigned $\lceil \frac{n}{p} \rceil$ contiguous rows of the matrix. Algorithm 2, which is called RowBlock shows an OpenMP parallel algorithm that uses the row block data distribution.

The specific distribution of iterations to threads is done by a scheduling strategy. OpenMP supports different scheduling strategies specified by the `schedule` clause. OpenMP provides

Algorithm 2: (RowBlock) OpenMP parallel row block algorithm of LU factorization

```
for $k \leftarrow 0$ to $n-1$ do
    /* Division step */
    #pragma omp parallel for
    for $i \leftarrow k+1$ to $n-1$ do
        $a[i][k] \leftarrow a[i][k]/a[k][k]$
    /* Elimination step */
    #pragma omp parallel for
    for $i \leftarrow k+1$ to $n-1$ do
        for $j \leftarrow k+1$ to $n-1$ do
            $a[i][j] \leftarrow a[i][j] - a[i][k] * a[k][j]$
```

the programmer with a set of scheduling clauses to control the way the iterations of a parallel loop are assigned to threads. The scheduling clauses when used with the chunk size parameter, can greatly affect the performance of the algorithm. In row block data distribution we used the static schedule without a specified chunk size that implies the OpenMP divides the iterations into p blocks of equal size of $\lceil \frac{n}{p} \rceil$ and is statically assigned to the threads in a blockwise distribution. The allocation of iterations is done at the beginning of the loop, and each thread will only execute those iterations assigned to it.

B. Algorithm with Row Cyclic Data Distribution

In row cyclic data distribution, rows of matrix A are distributed among the p threads or cores in a round-robin fashion. Algorithm 3, which is called RowCyclic, shows an OpenMP parallel algorithm that uses the row cyclic data distribution.

Algorithm 3: (RowCyclic) OpenMP parallel row cyclic algorithm of LU factorization

```
for $k \leftarrow 0$ to $n-1$ do
    /* Division step */
    #pragma omp parallel for schedule(static, bs)
    for $i \leftarrow k+1$ to $n-1$ do
        $a[i][k] \leftarrow a[i][k]/a[k][k]$
    /* Elimination step */
    #pragma omp parallel for schedule(static, bs)
    for $i \leftarrow k+1$ to $n-1$ do
        for $j \leftarrow k+1$ to $n-1$ do
            $a[i][j] \leftarrow a[i][j] - a[i][k] * a[k][j]$
```

In row cyclic data distribution we used the static schedule with a variable specified chunk size that specifies a static distribution of iterations to threads which assign blocks of size bs in a round-robin fashion to the threads available, where bs takes the values 1, 2, 4, 8, 16, 32 and 64 rows.

C. Algorithm with Pipelining

We assume that the rows of matrix A are distributed among the p threads or cores such that each core is assigned $\lceil \frac{n}{p} \rceil$ contiguous rows of the matrix. The general idea of a pipelining algorithm is that each thread executes the $\lceil \frac{n}{p} \rceil$ successive division and elimination steps of LU algorithm on the rows that is holds. To do so, it must receive the index of the pivot row, send it immediately to the next thread and then proceed with the division and elimination. Generally, the parallel algorithm is as follows:

For $k = my\_rank * \frac{n}{p}, ..., my\_rank * \frac{n}{p} + \frac{n}{p}$, do in each thread T_{my_rank}, where my_rank is the rank of the thread and are numbered from 0 to $p-1$:

1) Receive the index of the kth row from thread T_{my_rank}.
2) Send the index of the kth row just received to the thread T_{my_rank+1}.
3) Perform the division and elimination step number k for all rows m, $m > k$ that belong to thread T_{my_rank}.

The implementation of the receive and send operations in OpenMP can be realized with the help of the proposed `Get()` and `Put()` procedures, respectively. The `Get()` procedure can take as an argument the rank of the thread which receives the data and it returns a data element, whereas procedure `Put()` can take two arguments: one is for the rank of thread which will send the data and the other is a data element. The internal details of the `Get()` and `Put()` procedures will be discussed in the following subsection. Algorithm 4, which is called Pipe, shows an OpenMP parallel pipeline algorithm.

Algorithm 4: (Pipe) OpenMP pipelining algorithm of LU factorization

```
#pragma omp parallel private(k, i, j, row) shared(a)
{
long $my\_rank$ = omp_get_thread_num();
$bsize = n/p$;
if $(my\_rank$ != 0) then
    for $k \leftarrow (my\_rank * bsize)$ to $(my\_rank * bsize) + bsize$ do
        $row$ = Get($my\_rank$);
        Put($my\_rank + 1, row$);
        /* Division step */
        for $i \leftarrow k$ to $k + bsize$ do
            $a[i][row] = a[i][row]/a[row][row]$;
        /* Elimination step */
        for $i \leftarrow k$ to $k + bsize$ do
            for $j \leftarrow k$ to $n+1$ do
                $a[i][j] -= (a[i][row] * a[row][j])$;
else
    for $k \leftarrow (my\_rank * bsize)$ to $(my\_rank * bsize) + bsize$ do
        Put($my\_rank + 1, k$);
        /* Division step /*
        for $i \leftarrow k+1$ to $k + bsize$ do
            $a[i][k] = a[i][k]/a[k][k]$;
        /* Elimination step */
        for $i \leftarrow k+1$ to $k + bsize$ do
            for $j \leftarrow k+1$ to $n+1$ do
                $a[i][j] = (a[i][k] * a[k][j])$;
}
```

1) Implementation of Pipelining in OpenMP: In the pipelining model, a stream of data items is processed one after another by a sequence of threads $T_0, T_1, ..., T_n$, where each thread T_i performs a specific operation on each element of the data stream and passes the element onto the next thread T_{i+1}. This results in an input/output relation between the threads: Thread T_i receives the output of thread T_{i-1} as input and produces data elements for thread T_{i+1}, $1 < i < n$. Thread T_0 reads the sequence of input elements, thread T_{n-1} produces the sequence of output elements. After a start-up phase with $n - 1$ steps, all threads can work in parallel and can be executed by different cores in parallel. The pipeline model requires some coordination between the cooperating threads: For this purpose, we introduced a channel C between two threads. Thread T_1 can forward its output element to channel C and thread T_2 can start the computation of its corresponding stage only if channel C has provided the input data element. When a thread tries to read an empty channel C, then execution of the thread is automatically postponed until some other thread writes a data value into the channel C. Therefore, a channel will initially be empty, thread T_2 will have to wait if it attempts to read channel C before thread T_1 writes a value in C. This ensures that coordination between threads will be correct.

It is known that in the pipeline model, a thread can not only send a value but a stream of data values to channel C. In this case, the channel behaves like a queue that contains values. The values are written in channel C and stored in the queue until these can be read by other threads. As the values written by thread T_i, entering into channel from the left and flow to the right, where these values can be read by thread T_{i+1}. If thread T_i writes multiple values in the channel, they are stored and then can be read at any time by thread T_{i+1}.

Figure 1 illustrates the overall structure of the pipeline model. Each thread acts simultaneously as a producer and

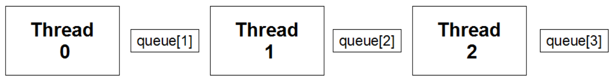

Figure 1. Pipelining model

as a consumer. The thread consumes data from the left side of the channel and produces data to the right of the channel. The coordination of the threads of the pipeline stages can be used with the help of an array consisting of queues. Each element of the array refers to a queue of type `buff_list` and each queue has two pointers, a head and a tail of type `record_s`. Therefore, each queue is implemented as a linked list. For a pipeline model with stages n, a data structure array `buff[n]` of type `buff_list` is used, see below.

```
struct record_s {
   double val;
   long prod;
   struct record_s* next_p;
};

struct buf_list {
    struct record_s* head_p;
    struct record_s* tail_p;
};

struct buf_list buff[n];
```

The reference of any item of the array of queues is like the reference of any common elements of an array. `buff[0]` is the first queue of the array while `buff[1]` is a second queue of the array, and so forth. For a thread with rank `my_rank`, a data value can be sent to the queue of the thread with rank `my_rank + 1`, i.e. `buff[my_rank + 1]`, with the help of the `Put(my_rank + 1, data)` procedure. On the other hand, a thread with rank `my_rank` can read/receive a data value from the queue `buff[my_rank]` with the help of the `data = Get(my_rank)` procedure.

The details of the `Put()` procedure are shown below. Firstly, it creates a record `rec_p` that contains a data value through the `Create_record()` procedure. (The details of the `Create_record()` procedure are omitted as this is known and simple.) Next, it adds the record to the queue, `buff[my_rank]` through the known `Enqueue()` procedure. The `Enqueue()` procedure is bounded by a critical region, `pragma omp critical`, so that a thread can add a data value to a specific queue at a time. Further, the `Put()` procedure uses an array `producers_done[n]` for a pipeline with stages n and each element of the array refers to a counter. This counter corresponds to the number of data items are contained in a queue with rank `my_rank`. The details of the `Enqueue()` procedure are shown below.

```
void Put(long my_rank, double data) {
   struct record_s* rec_p;

   rec_p = Create_record(my_rank, data);
   #pragma omp critical(queue)
   {
      Enqueue(my_rank, rec_p);
   }
   #pragma omp critical(done)
   producers_done[my_rank]++;
}

void Enqueue(long my_rank, struct record_s* rec_p)
{
   if (buff[my_rank].tail_p == NULL) {
      buff[my_rank].head_p = rec_p;
      buff[my_rank].tail_p = rec_p;
   } else {
      buff[my_rank].tail_p->next_p = rec_p;
      buff[my_rank].tail_p = rec_p;
   }
}
```

The details of the `Get()` procedure are shown below. Firstly, it checks if a specific queue with rank `my_rank` is full and if it is then it removes a node from the queue through the known `Dequeue()` procedure and returns as a record `rec_p`. The `Dequeue()` procedure is bounded by a critical region, `pragma omp critical`, so that a thread can remove a node from a specific queue at a time. Finally, the `Get()` procedure returns a data value with the help of the field `rec_p->val`. The details of the `Dequeue()` procedure are shown below.

```
double Get(long my_rank) {
   struct record_s* rec_p;
   double data;
```

```
    while (producers_done[my_rank] < 1 ||
     buff[my_rank].head_p != NULL) {
          #pragma omp critical (queue)
          {
              rec_p = Dequeue(my_rank);
          }
          if (rec_p != NULL) {
              data = rec_p->val;
              free(rec_p);
              return data;
          }
    }
}

struct record_s* Dequeue(long myrank) {
    struct record_s* rec_p;

    if (buff[myrank].head_p == NULL) {
       return NULL;
     } else if (buff[myrank].head_p == buff[myrank].tail_p) {
       // One record in queue
       rec_p = buff[myrank].head_p;
       buff[myrank].head_p = buff[myrank].tail_p = NULL;
    } else {
       // Multiple record in queue
       rec_p = buff[myrank].head_p;
       buff[myrank].head_p = buff[myrank].head_p->next_p;
    }
    return rec_p;
}
```

2) Performance Analysis of the Pipeline Algorithm: The performance model of the pipeline algorithm depends on two main aspects: the computational cost and the communication cost. In the case of multicore, communications are performed through direct `Put()/Get()` operations by two or more threads. An analytical model based on the number of operations and their cost in CPU times can be used to determine the computational cost. Similarly, to predict the communication cost we calculate the number of get/put operations between threads and measure the number of cycles for put and get operations.

The execution time of the pipeline algorithm can be broken up into three terms:

- $T_{division}$: It is the total time to perform the division step in parallel. The number of division operations on a maximally-loaded thread in the kth iteration is $(n/p-k-1)$. Hence, the total time $T_{division}$ is given by:

$$T_{division} = \Sigma_{k=0}^{n-1}(n/p-k-1)t_{div} = \frac{n/p(n/p-1)}{2}t_{div} \quad (1)$$

  where t_{div} is the division time for a floating point operation.

- T_{elim}: It is the total time to perform the elimination step in parallel. The elimination step involves multiplications and subtractions. The number of arithmetic operations on a maximally-loaded thread in the kth iteration involves $n/p(n-k-1)$ multiplications and subtractions. The total operations spent in elimination step in the kth iteration is $2(n/p)(n-k-1)$. Therefore, the total time T_{elim} is given by:

$$T_{elim} = 2(\frac{n}{p})\Sigma_{k=0}^{n-1}(n-k-1)t_{elim} = 2(\frac{n}{p})\frac{n(n-1)}{2}t_{elim} \quad (2)$$

  where t_{elim} is the elimination time for a floating point operation.

- T_{comm}: It is the total communication time to exchange elements between threads of the pipeline. The communication step involves the `Put()` and `Get()` operations. The number of communication operations on a maximally-loaded thread in the kth iteration involves one `Get()` and one `Put()` operation. The total operations spent in communication step on a thread is $2(n/p)$. Then, the total time T_{comm} is given by:

$$T_{comm} = 2(\frac{n}{p})t_{comm} \quad (3)$$

  where t_{comm} is the communication (put and get) time for a floating point operation.

The total parallel execution time of the pipeline algorithm, T_{pipe}, using p threads is the summation of the three terms and is given by:

$$T_{pipe}(p) = T_{division} + T_{elim} + T_{comm} \quad (4)$$

IV. PERFORMANCE RESULTS

A. System Platform and Experimental Process

For our experimental evaluation we used an Intel Core 2 Quad CPU with four processor cores, a 2.40 GHz clock speed and 8 Gb of memory. The system ran GNU/Linux, kernel version 2.6, for the x84 64 ISA. All programs were implemented in C using OpenMP interface and were compiled using gcc, version 4.4.3, with the "-O2" optimization flag.

Several sets of test matrices were used to evaluate the performance of the parallel algorithms, a set of randomly generated input matrices with sizes ranging from 32×32 to 4096×4096. To compare the parallel algorithms, the practical execution time was used as a measure. Practical execution time is the total time in seconds an algorithm needs to complete the computation, and it was measured using the `omp_get_wtime()` function of OpenMP. To decrease random variation, the execution time was measured as an average of 50 runs.

B. Analysis based on Performance Results

It is known that the RowCyclic algorithm works for several values of block size in comparison to the other two parallel algorithms. In order to examine the relation between the execution time of the RowCyclic algorithm and the block size, we ran the RowCyclic algorithm for different block sizes. Tables II, III and IV present the average execution time of the RowCyclic algorithm for different block sizes on one, two and four cores, respectively. As can be seen from the Tables, the execution time of the RowCyclic algorithm is not affected by changes in the block size. This is due to the fact that the RowCyclic method has poor locality of reference. For this reason we have used the RowCyclic

Table II
EXECUTION TIME (IN SECONDS) OF THE ROW CYCLIC ALGORITHM FOR DIFFERENT BLOCK SIZES AND MATRIX SIZES ON 1 CORE

	1	2	4	8	16	32	64
32	0,000090	0,000090	0,000090	0,000090	0,000090	0,000090	0,000120
64	0,000320	0,000320	0,000330	0,000310	0,000310	0,000310	0,000370
128	0,001600	0,001620	0,001610	0,001520	0,001580	0,001540	0,001910
256	0,019450	0,016240	0,016290	0,014250	0,012970	0,012240	0,013280
512	0,156890	0,154610	0,153820	0,146940	0,145960	0,142700	0,144230
1024	1,249660	1,248460	1,235450	1,245460	1,224400	1,220060	1,246580
2048	9,771560	9,812090	9,741170	9,731280	9,705160	9,734660	9,882660
4096	77,083650	77,758700	77,475390	77,431230	77,449560	77,877120	77,551210

Table III
EXECUTION TIME (IN SECONDS) OF THE ROW CYCLIC ALGORITHM FOR DIFFERENT BLOCK SIZES AND MATRIX SIZES ON 2 CORES

	1	2	4	8	16	32	64
32	0,000190	0,000170	0,000170	0,000160	0,000150	0,000170	0,000160
64	0,000590	0,000510	0,000480	0,000381	0,000380	0,000400	0,000440
128	0,002830	0,002270	0,001900	0,001300	0,001330	0,001370	0,001450
256	0,019260	0,014420	0,011130	0,009220	0,007380	0,006890	0,008270
512	0,133100	0,119100	0,108470	0,089770	0,096710	0,088760	0,083520
1024	0,960620	0,932230	0,916440	0,921230	0,886090	0,878080	0,872780
2048	7,358990	7,252270	7,242080	7,345110	7,137880	7,159010	7,212390
4096	57,486940	56,813990	57,827630	57,567100	57,056400	56,748830	57,341700

Table IV
EXECUTION TIME (IN SECONDS) OF THE ROW CYCLIC ALGORITHM FOR DIFFERENT BLOCK SIZES AND MATRIX SIZES ON 4 CORES

	1	2	4	8	16	32	64
32	0,000200	0,000180	0,000180	0,000170	0,000170	0,000190	0,000170
64	0,000580	0,000510	0,000460	0,000460	0,000400	0,000450	0,000500
128	0,002640	0,001970	0,001550	0,001410	0,001130	0,001210	0,001730
256	0,017600	0,011890	0,008140	0,005121	0,005510	0,005550	0,005830
512	0,128780	0,114220	0,092960	0,091341	0,086480	0,084230	0,077100
1024	0,953610	0,912340	0,877420	0,878110	0,862070	0,860460	0,853750
2048	7,273750	7,175600	7,130310	7,119210	7,078810	7,092450	7,132170
4096	56,396170	56,324500	57,053530	56,121200	56,388660	56,571100	56,848600

algorithm with a block of size 1 for the comparison to the other two parallel algorithms.

Figure 2 shows the average execution time of all parallel algorithms for variable matrix size on one, two and four cores. As can be seen from Figure 2, the execution time of all algorithms is increased as the matrix size is increased. We observe that the parallel algorithms run fast on small matrix sizes (from 32 to 2048) because these sizes fit entirely in the L1 and L2 cache. For large matrix sizes there is a slowdown in the execution time because the matrices are larger than the cache (which is the more likely case) and this leads to being served primarily by the slow main memory.

Figure 3 presents the way the performance of all parallel algorithms was affected when parallel processed using OpenMP for 1 to 4 threads, for small and large matrix sizes. As can be seen, the performance of the algorithms improved with each additional thread. We observe that the performance of the RowBlock and RowCyclic algorithms increased at a decreasing rate. This is due to the fact that the implicit synchronization cost of parallel 'for' loops (i.e. start and stop parallel execution of the threads) dominates the execution time. In this case, the cost of synchronization is about n^2. On the other hand, it is clear the performance of the Pipe algorithm on two and four cores resulted in the approximate doubling and quadrupling of its performance. With the Pipe algorithm, the decrease in performance is much slower than it is with the others. Therefore, for the Pipe algorithm there is a strong inverse relation between the parallel execution time and the number of threads, since the total communication and overhead time is much lower than the processing time on each thread. Finally, we expect that the performance of the Pipe algorithm will be better than that of the other two algorithms for large numbers of cores, such as 8 and 16 cores.

As we can see in the results, the ranking of the parallel algorithms is obvious. The parallel algorithm with the best resulting performance is the Pipe, the second best is RowBlock and the third best is RowCyclic.

C. Evaluation of the Performance Model

The final experiment was conducted to verify the correctness of the proposed performance model for the pipeline algorithm. The performance model of equation 4 is plotted in Figure 4 using time parameters. In order to obtain these

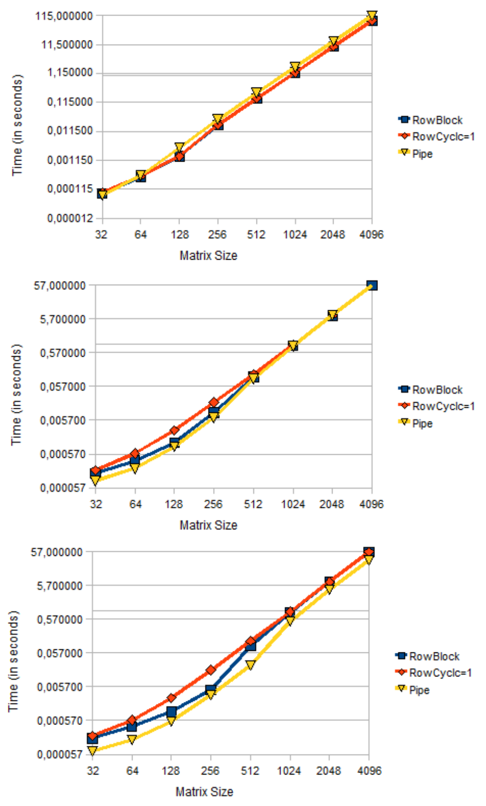

Figure 2. Execution times (in seconds) on log scale of all parallel algorithms for variable matrix sizes on 1 core (top), 2 cores (center) and 4 cores (bottom)

predicted results, we have determined the time parameters for division, elimination and communications operations (i.e. t_{div}, t_{elim}, t_{comm}) for different matrix sizes. These parameters of the target machine are determined experimentally with the results shown in Figure 5.

As can be seen from Figure 4, the predicted execution times are quite close to the real ones. This verifies that the proposed performance model for the pipeline algorithm is fairly accurate and hence it provides a means to test the viability of the pipeline implementation on any multicore (i.e. dual core, quad core) without taking the burden of real testing. Further, the proposed performance model is able to predict the parallel performance and the general behavior of the implementation. However, there are minor differences between the measured and predicted results in the pipeline implementation.

V. Conclusion

In this paper, we presented and evaluated three OpenMP parallel algorithms for the LU decomposition method on a multicore platform. We presented two naive OpenMP parallel algorithms such as RowBlock and RowCyclic and we placed special emphasis on presenting the third OpenMP parallel algorithm such as the pipeline technique. Further,

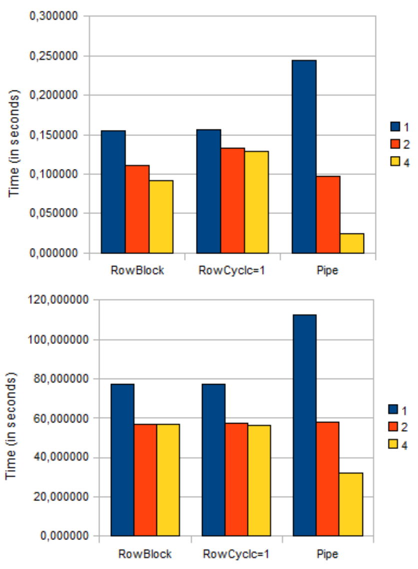

Figure 3. Execution times (in seconds) of all parallel algorithms for 1 to 4 threads for matrix of size 512 (top) and matrix of size 4096 (bottom)

we proposed an implementation of the pipelining technique in OpenMP using the queue data structure. The queue is realized with the help of the proposed `Get()` and `Put()` procedures. Contrary to previous works, no work has been done to implement the Pipeline technique for the dense LU factorization method in an OpenMP programming environment. We showed through our experimental studies that the pipeline algorithm achieves the best overall performance in comparison with the other two naive algorithms, RowBlock and RowCyclic.

Moreover, in this paper we proposed a performance model to predict the performance of the Pipe implementation. We verified this model with experimental measurements and showed that this model is practical. The accuracy of our model is reasonable given its simplicity.

As future work, it would be interesting to use the proposed OpenMP pipeline implementation for the LU factorization method with pivoting and other factorization methods, too, such as Cholesky factorization and QR factorization. Moreover, we will extend the performance study of the three parallel algorithms on platforms with the number of cores is more than four.

References

[1] A. Buttari, J. Langou, J. Kurzak, and J. J. Dongarra, "Parallel tiled QR factorization for multicore architectures," *Concurrency Computat.: Pract. Exper.*, vol. 20, no. 13, pp. 1573–1590, 2008.

[2] ——, "A class of parallel tiled linear algebra algorithms for multicore architectures," *Parallel Comput.*, vol. 35, no. 1, pp. 38–53, 2009.

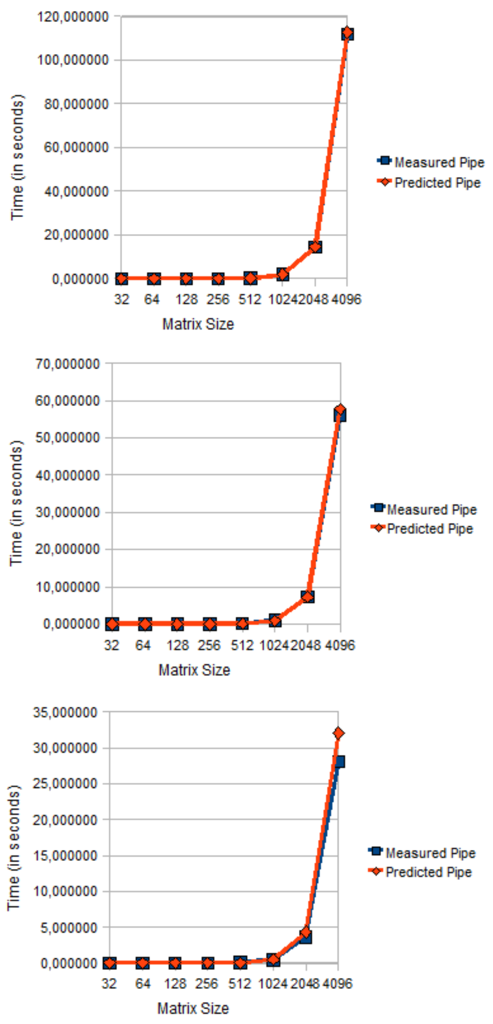

Figure 4. The measured and predicted execution times of the pipeline algorithm on 1 core (top,) 2 cores (center) and 4 cores (bottom)

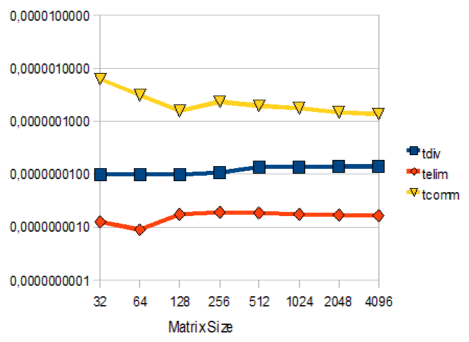

Figure 5. Multicore execution performance as a function of the matrix size for operations division, elimination and communication on one core

[3] A. Buttari, J. J. Dongarra, P. Husbands, J. Kurzak, and K. Yelick, "Multithreading for synchronization tolerance in matrix factorization," in *Journal of Physics: Conference Series 78(1):012028*, 2007.

[4] E. Elmroth and F. G. Gustavson, "Applying recursion to serial and parallel QR factorization leads to better performance," *IBM J. Res. & Dev.*, vol. 44, no. 4, pp. 605–624, 2000.

[5] ——, "High performance library software for QR factorization," in *Proc. of 5th International Workshop Applied Parallel Computing, New Paradigms for HPC in Industry and Academia*, LNCS 1947, 2000, pp. 53–63.

[6] B. C. Gunter and R. A. van de Geijn, "Parallel out-of-core computation and updating the QR factorization," *ACM Transactions on Mathematical Software*, vol. 31, no. 1, pp. 60–78, 2005.

[7] M. Marques, G. Quintana-Orti, E. Quintana-Orti, and R. van de Geijn, "Out-of-core computation of the QR factorization on multi-core processors," in *Proc. of the Euro-Par Parallel Processing*, LNCS 5704, 2009, pp. 809–820.

[8] S. McGinn and R. Shaw, "Parallel gaussian elimination using OpenMP and MPI," in *Proc. of the 16th Annual International Symposium on High Performance Computing Systems and Applications*, 2002, pp. 169–173.

[9] J. Kruger and R. Westermann, "Linear algebra operators for GPU implementation of numerical algorithms," *ACM Transactions Graphics*, vol. 22, pp. 908–916, 2003.

[10] A. Moravanszky, "Dense matrix algebra on the GPU," in *http://www.shaderx2.com/shaderx.pdf*, 2003.

[11] V. Volkov and J. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *Proc. of the ACM/IEEE Conference on Supercomputing*, 2008, pp. 1–11.

[12] N. Galoppo, N. Govindaraju, M. Henson, and D. Manocha, "LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware," in *Proc. of the ACM/IEEE Conference on Supercomputing*, 2005, p. 3.

[13] I. Fumihiko, M. Manabu, G. Keigo, and H. Kenichi, "Performance study of LU decomposition on the programmable GPU," in *Proc. of the High Performance Computing*, 2005, pp. 83–94.

[14] G. Quintana-Orti, F. Igual, E. Quintana-Orti, and R. van de Geijn, "Solving dense linear algebra problems on platforms with multiple hardware accelerators," in *Proc. of the 14th ACM/SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2009, pp. 121–130.

[15] J. Kurzak and J. J. Dongarra, "QR factorization for the CELL processor," *Scientific Programming*, vol. 17, no. 1-2, pp. 31–42, 2009.

[16] J. Kurzak, A. Buttari, and J. J. Dongarra, "Solving systems of linear equation on the CELL processor using cholesky factorization," *IEEE Trans. Parallel Distrib. Syst.*, vol. 19, no. 9, pp. 1175–1186, 2008.

[17] B. C. Vishwas, A. Gadia, and M. Chaudhuri, "Implementing a parallel matrix factorization library on the cell broadband engine," *Scientific Programming*, vol. 17, no. 1-2, pp. 3–29, 2009.

[18] G. Golub and C. V. Loan, *Matrix Computations*. Johns Hopkins University Press, 1996.