Journal of Computational and Applied Mathematics 236 (2011) 326–341

Contents lists available at SciVerse ScienceDirect

Journal of Computational and Applied Mathematics

journal homepage: www.elsevier.com/locate/cam

Parallel direct methods for solving the system of linear equations with pipelining on a multicore using OpenMP

Panagiotis D. Michailidis [a,*], Konstantinos G. Margaritis [b]

[a] Department of Balkan Studies, University of Western Macedonia, Florina, Greece

[b] Department of Applied Informatics, University of Macedonia, Thessaloniki, Greece

ARTICLE INFO

Keywords:
Matrix computations
Direct methods
Parallel computing
MultiCore
OpenMP
Parallel algorithms

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 lie at the heart of many applications in scientific computing. This paper describes and analyzes three parallel versions of the dense direct methods such as the Gaussian elimination method and the LU form of Gaussian elimination that are used in linear system solving on a multicore using an 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 direct methods with the pipelining technique.

© 2011 Elsevier B.V. All rights reserved.

1. Introduction

The solving of dense and large scale linear algebraic systems lies at the core of many scientific and computational economic applications such as linear programming, computational statistics, econometrics and theory game. Many practical problems can be translated into a large scale linear algebraic system. Therefore, the parallel solving of dense and large scale linear algebraic systems is of great importance in the field of scientific computation.

Methods for a linear algebraic system generally fall into two categories: direct methods and iterative methods. In direct solution 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 solution 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 Gaussian elimination algorithm and the *LU* form of Gaussian elimination that are used in dense linear systems solving. Many parallel implementations have been studied on modern high performance systems, such as multicore and special purpose architectures/accelerators, such as Graphics Processing Units (GPUs) or the CELL BE for solving numerical computations. More specifically, many parallelization schemes of Gaussian elimination have been presented back to the

* Corresponding author.
E-mail addresses: pmichailidis@uowm.gr, panosm@uom.gr (P.D. Michailidis), kmarg@uom.gr (K.G. Margaritis).

0377-0427/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.cam.2011.07.023

80s [1]. In recent years, Buttari et al. [2–4] present parallel tiled linear algebra algorithms for "standard" (x86 and alike) multicore processors with framework Parallel Linear Algebra for Scalable Multicore Architectures (PLASMA). PLASMA is a redesign of LAPACK [5] and ScaLAPACK [6] for shared memory computers based on multi-core processor architectures. PLASMA relies on tile algorithms, which provide fine granularity parallelism. Seminal work leading to the tile QR algorithm was done in [7–9]. Gunter et al. [10] 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 in [11]. Some work has been done on load balancing schemes for the *LU* factorization algorithm such as the paper in [12], 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.

Thompson et al. [13] implement matrix multiplication on a GPU, achieving three times higher performance compared to a simple CPU implementation. Larsen et al. [14] compare their GPU implementation to ATLAS [15], a cache-optimized CPU implementation. Both Hall et al. [16] and Fatahalian et al. [17] describe improved algorithms for modern hardware. In addition to the problem of matrix multiplication, there are a wide variety of dense linear algebra operations running on the GPU: the conjugate gradient method [18,19], the Gauss–Seidel method [18], the projected Jacobi method [19] and the direct factorization methods, such as *LU* factorization [20–23]. The static pipelining technique for dense matrix factorizations on the CELL processor was originally implemented in [24,25] and an extensive performance study of popular matrix factorization methods on CELL processor was conducted in [26]. Finally, Agullo et al. [27] develop a project called Matrix Algebra on GPU and Multi-core Architectures (MAGMA). The MAGMA project aims to develop a dense linear algebra library similar to LAPACK but for heterogeneous/hybrid architectures, starting with current "Multicore + GPU" systems.

Despite these research results, no attempt has been made yet to implement and analyze the performance of several parallelization strategies for the dense direct methods such as the Gaussian elimination method and the *LU* form of Gaussian elimination on a multicore using an 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 a 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 2, the Gaussian elimination method and the *LU* version of Gaussian elimination are recalled. In Section 3, the three parallel implementations of the Gaussian elimination method and the *LU* form of Gaussian elimination are discussed and so is the corresponding performance model for the pipelining implementation. Performance results are given in Section 4. Comparison of our work with some related works is presented in Section 5. Finally, some conclusions are drawn in Section 6.

2. Direct methods for linear systems

Consider the following problem of solving a system of linear equations of the form

$$\begin{aligned}
&a_{0,0}x_0 + a_{0,1}x_1 + \cdots + a_{0,n-1}x_{n-1} = b_0\\
&a_{1,0}x_0 + a_{1,1}x_1 + \cdots + a_{1,n-1}x_{n-1} = b_1\\
&\vdots\\
&a_{n-1,0}x_0 + a_{n-1,1}x_1 + \cdots + a_{n-1,n-1}x_{n-1} = b_{n-1}.
\end{aligned}$$

In matrix notation, this system is written as $Ax = b$,

$$\begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0n-1}\\ a_{10} & a_{11} & \cdots & a_{1n-1}\\ \cdot & \cdot & \cdot & \cdot\\ \cdot & \cdot & \cdot & \cdot\\ \cdot & \cdot & \cdot & \cdot\\ a_{n-10} & a_{n-11} & \cdots & a_{n-1n-1} \end{bmatrix} \begin{bmatrix} x_0\\ x_1\\ \cdot\\ \cdot\\ \cdot\\ x_{n-1} \end{bmatrix} = \begin{bmatrix} b_0\\ b_1\\ \cdot\\ \cdot\\ \cdot\\ b_{n-1} \end{bmatrix}$$

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

The system could be solved by direct methods such as elimination methods and factorization methods. The Gaussian elimination consists of two phases. First, through a series of algebraic manipulations, the original system of equations is reduced to an upper triangular system of the form $Ux = y$, where U is a unit upper-triangular matrix. In the second phase of solving a system of linear equations, the upper-triangular system is solved for the variables in a reverse order from x_{n-1} to x_0 by a procedure known as backward substitution [28]. A serial version of the Gaussian elimination algorithm shown in Algorithm 1 consists of three nested loops. For each iteration of the outer loop, there is a division step and an elimination step. As the computation proceeds, only the lower-right $k \times k$ submatrix 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 the matrix U shares storage with A and overwrites the upper-triangular portion of A. The elements $a_{i,k}$ computed on line 4 of Algorithm 1 is actually $u_{i,k}$. Similarly, the elements $a_{k,k}$ equated to 1 on line 6 is $u_{k,k}$.

Algorithm 1: Sequential Gaussian elimination algorithm

```
1  for k ← 0 to n − 1 do
2      /* Division step */
3      for i ← k + 1 to n − 1 do
4          a[i][k] ← a[i][k]/a[k][k]
5      y[k] ← b[k]/a[k][k]
6      a[k][k] ← 1
7      /* Elimination step */
8      for i ← k + 1 to n − 1 do
9          for j ← k + 1 to n − 1 do
10             a[i][j] ← a[i][j] − a[i][k] ∗ a[k][j]
11         b[i] ← b[i] − a[i][k] ∗ y[k]
12         a[i][k] ← 0
```

In the sequential Gaussian elimination algorithm, during kth iteration: division takes $(n-k-1)$ arithmetic operations and elimination takes $(n-k-1)^2 \times 2$ arithmetic operations. Assuming that division, multiplication and subtraction 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 $O(n^3)$.

Although we discuss Gaussian elimination in the context of upper-triangularization, a similar procedure can be used to factorize matrix A as the product of a lower-triangular matrix L and a unit upper-triangular matrix U so that $A = LU$. This factorization is commonly referred to as the *LU* form of Gaussian elimination. When only the *LU* form of Gaussian elimination is needed, the right-hand side of the linear system does not have to be transformed. An *LU* form uses the Gaussian elimination 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 [28]. The *LU* form of Gaussian elimination can be obtained if the steps on lines 5, 6, 11 and 12 of Algorithm 1 are not performed. We will adopt the *LU* form of Gaussian elimination for parallel algorithms in the remainder of this paper because it is a general case of the Gaussian elimination algorithm. After the termination of the *LU* algorithm, the matrices L and U share storage with A and overwrite the lower-triangular and upper-triangular portions of A, respectively. The advantage of the *LU* form of Gaussian elimination over the elimination method is that the factorization into L and U is done only once but can be used to solve several linear systems with the same matrix A and different right-hand side vectors b without repeating the elimination process.

3. Parallel algorithms for linear systems

Because the complexity of the elimination or matrix factorization, $O(n^3)$, is greater than that of the forward or backward substitution we will focus our attention on parallelization of the *LU* form of Gaussian elimination (GE). The parallelization of the *LU* form of GE is based on accessing the matrix by rows instead of columns because accessing the matrix by rows has good performance according the experimental study which is presented in Section 4. In the following section, we discuss three OpenMP parallel algorithms for the *LU* form of Gaussian elimination. 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. We must note that identical parallel algorithms and performance model can be obtained for the Gaussian elimination method.

3.1. 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 $\left\lceil \frac{n}{p} \right\rceil$ contiguous rows of the matrix. Algorithm 2, which is called RowBlock shows an OpenMP parallel algorithm that uses the row block data distribution.

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

```
1  for k ← 0 to n − 1 do
2      /* Division step */
3      #pragma omp parallel for
4      for i ← k + 1 to n − 1 do
5          a[i][k] ← a[i][k]/a[k][k]
6      /* Elimination step */
7      #pragma omp parallel for
8      for i ← k + 1 to n − 1 do
9          for j ← k + 1 to n − 1 do
10             a[i][j] ← a[i][j] − a[i][k] ∗ a[k][j]
```

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 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, which implies that OpenMP divides the iterations into p blocks of equal size of $\left\lceil \frac{n}{p} \right\rceil$ 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.

3.2. 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 form of GE

```
1  for k ← 0 to n − 1 do
2      /* Division step */
3      #pragma omp parallel for schedule(static, bs)
4      for i ← k + 1 to n − 1 do
5          a[i][k] ← a[i][k]/a[k][k]
6      /* Elimination step */
7      #pragma omp parallel for schedule(static, bs)
8      for i ← k + 1 to n − 1 do
9          for j ← k + 1 to n − 1 do
10             a[i][j] ← 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.

3.3. 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 $\left\lceil \frac{n}{p} \right\rceil$ contiguous rows of the matrix. The general idea of a pipeline algorithm is that each thread executes the $\left\lceil \frac{n}{p} \right\rceil$ successive division and elimination steps of the *LU* form of Gaussian elimination 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 $k = my\_rank * \frac{n}{p}, \ldots, my\_rank * \frac{n}{p} + \frac{n}{p}$, 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.

3.3.1. Implementation of pipelining in OpenMP

In the pipelining model, a stream of data items is processed one after another by a sequence of threads $T_0, T_1, \ldots, T_n$, 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}.

Algorithm 4: (Pipe) OpenMP pipelining algorithm of LU form of GE

```
1  #pragma omp parallel private(k, i, j, row) shared(a)
2  {
3  long my_rank = omp_get_thread_num();
4  bsize = n/p;
5  if (my_rank != 0) then
6      for k ← (my_rank * bsize) to (my_rank * bsize) + bsize do
7          row = Get(my_rank);
8          Put(my_rank + 1, row);
9          /* Division step */
10         for i ← k to k + bsize do
11             a[i][row] = a[i][row]/a[row][row];
12         /* Elimination step */
13         for i ← k to k + bsize do
14             for j ← k to n + 1 do
15                 a[i][j]− = (a[i][row] * a[row][j]);
16 else
17     for k ← (my_rank * bsize) to (my_rank * bsize) + bsize do
18         Put(my_rank + 1, k);
19         /* Division step /*
20         for i ← k + 1 to k + bsize do
21             a[i][k] = a[i][k]/a[k][k];
22         /* Elimination step */
23         for i ← k + 1 to k + bsize do
24             for j ← k + 1 to n + 1 do
25                 a[i][j]− = (a[i][k] * a[k][j]);
26 }
```

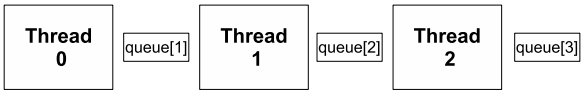

Fig. 1. Pipelining model.

Fig. 1 illustrates the overall structure of the pipeline model. Each thread acts simultaneously as a producer and 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. First, 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 that 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. First, 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;
}
```

3.3.2. 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

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

  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 the elimination step in the kth iteration is $2(n/p)(n-k-1)$. Therefore, the total time T_{elim} is given by

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

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 the one `Get()` and one `Put()` operations. The total operations spent in the communication step on a thread is $2(n/p)$. Then, the total time T_{comm} is given by

$$T_{comm} = 2\left(\frac{n}{p}\right) t_{comm} \tag{3}$$

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

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

We must note that identical performance model can be analyzed and obtained for the Gaussian elimination method.

4. Performance results

4.1. 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 an 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.

4.2. Analysis based on performance results

We know that the six permutations of indices i, j and k of Algorithm 1 give six different loop orders of Gaussian elimination and an *LU* form of Gaussian elimination, which we call the "*ijk*" form. 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. The *kij* and *kji* forms differ only in accessing the matrix by rows 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 Tables 1 and 2 for different matrix sizes on one core for the *LU* form and Gaussian elimination, respectively. 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 a *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.

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 of two versions of Gaussian elimination for different block sizes (such as $bs = 1, 2, 4, 8, 16, 32, 64$) on a multicore. The performance results of the RowCylic algorithm for Gaussian elimination and the *LU* form of Gaussian elimination were identical to those given in Tables 1 and 2 for the *kij* form respectively. From the results it is shown that 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 algorithm with a block of size 1 for the comparison to the other two parallel algorithms.

Figs. 2 and 3 (or see Tables 3–8 of appendix) show the average execution time of all parallel algorithms for the variable matrix size on one, two and four cores for the *LU* form of Gaussian elimination and Gaussian elimination methods, respectively. As can be seen from Figs. 2 and 3, 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.

Figs. 4 and 5 present the way the performance of all parallel algorithms was affected when parallel processed using OpenMP for 1–4 threads, for small and large matrix sizes for the *LU* form of Gaussian elimination and Gaussian elimination methods, respectively. 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 that the performance of the

Table 1
Execution times (in seconds) of the *kij* and *kji* forms for different matrix sizes and for the *LU* form of Gaussian elimination on 1 core.

	kij	*kji*
32	0.000080	0.000120
64	0.000310	0.000700
128	0.001560	0.005480
256	0.01890	0.049870
512	0.155090	0.544740
1024	1.243250	5.623180
2048	9.750050	43.741910
4096	77.055230	391.604810

Table 2
Execution times (in seconds) of the *kij* and *kji* forms for different matrix sizes and for the Gaussian elimination on 1 core.

	kij	*kji*
32	0.000080	0.000120
64	0.000300	0.000700
128	0.002810	0.00621
256	0.021680	0.05808
512	0.165940	0.57716
1024	1.244720	5.62122
2048	9.639700	43.44745
4096	76.125190	390.23631

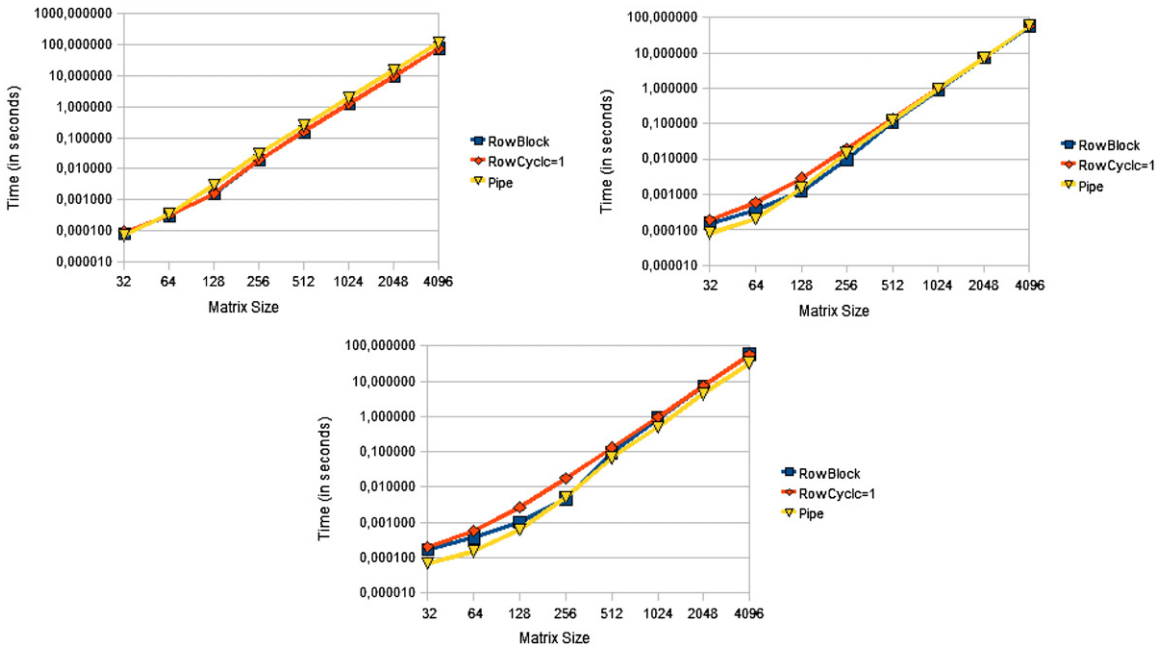

Fig. 2. Execution times (in seconds) on log scale of all parallel algorithms for variable matrix sizes on 1 core (top left), 2 cores (top right) and 4 cores (bottom) for the *LU* form of Gaussian elimination.

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.

Furthermore, Figs. 6 and 7 present the speedups of all parallel algorithms for small and large matrix sizes for the *LU* form of Gaussian elimination and Gaussian elimination methods, respectively. We observe that the speedups of the RowBlock and RowCyclic algorithms range from 1 to 1.6 for 1–4 threads, whereas the speedups of the Pipe algorithm range from 1

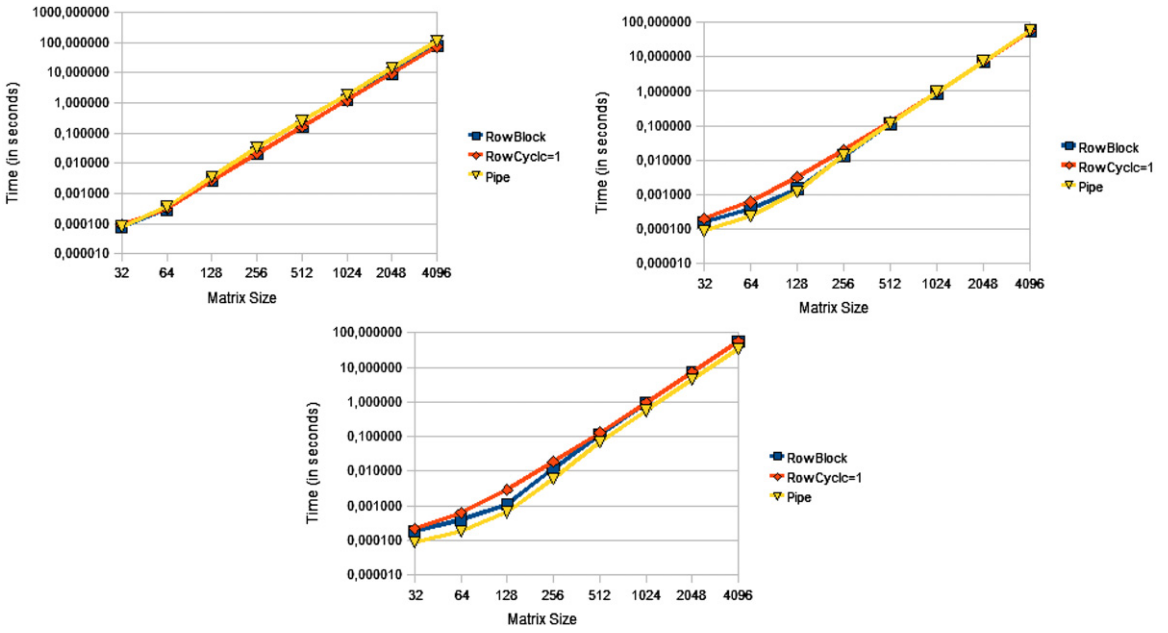

Fig. 3. Execution times (in seconds) on log scale of all parallel algorithms for variable matrix sizes on 1 core (top left), 2 cores (top right) and 4 cores (bottom) for the Gaussian elimination method.

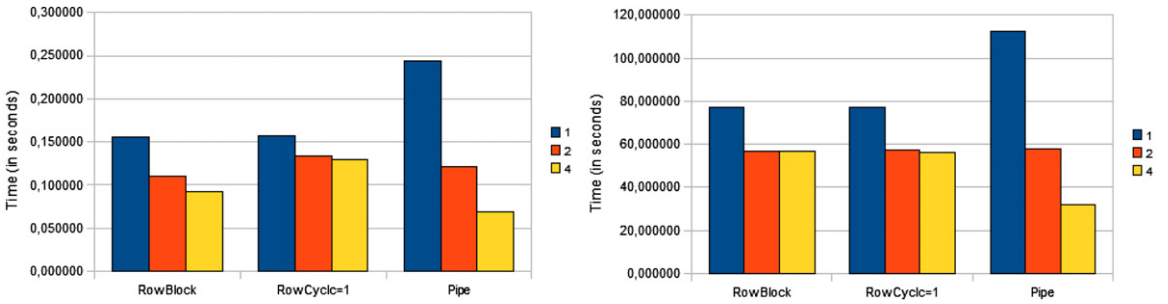

Fig. 4. Execution times (in seconds) of all parallel algorithms for 1 to 4 threads for matrix of size 512 (left) and matrix of size 4096 (right) for the *LU* form of Gaussian elimination.

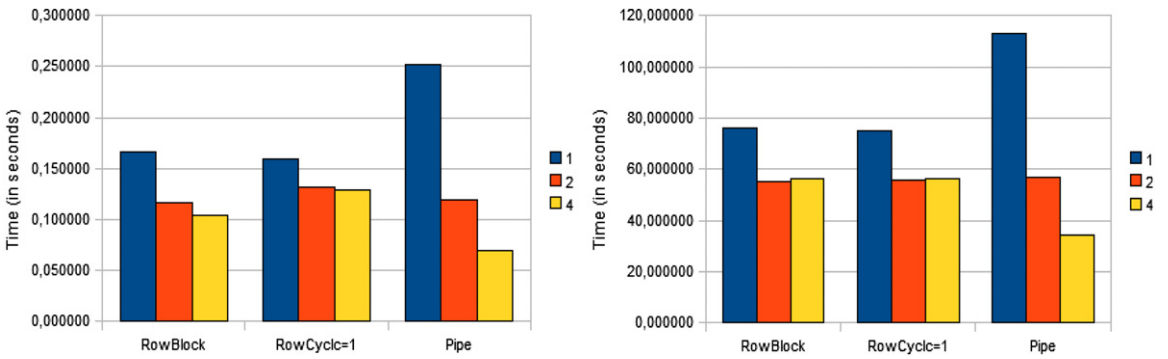

Fig. 5. Execution times (in seconds) of all parallel algorithms for 1 to 4 threads for matrix of size 512 (left) and matrix of size 4096 (right) for the Gaussian elimination method.

to 3.6 for 1–4 threads. As we can see in the speedup 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.

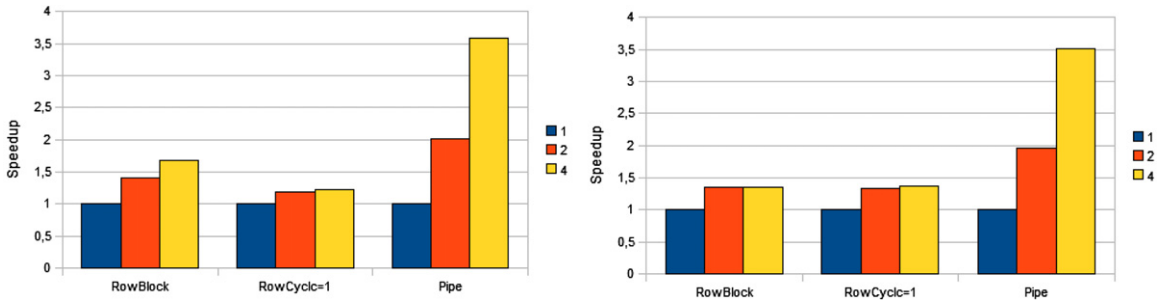

Fig. 6. Speedups of all parallel algorithms for 1–4 threads for matrix of size 512 (left) and matrix of size 4096 (right) for the *LU* form of Gaussian elimination.

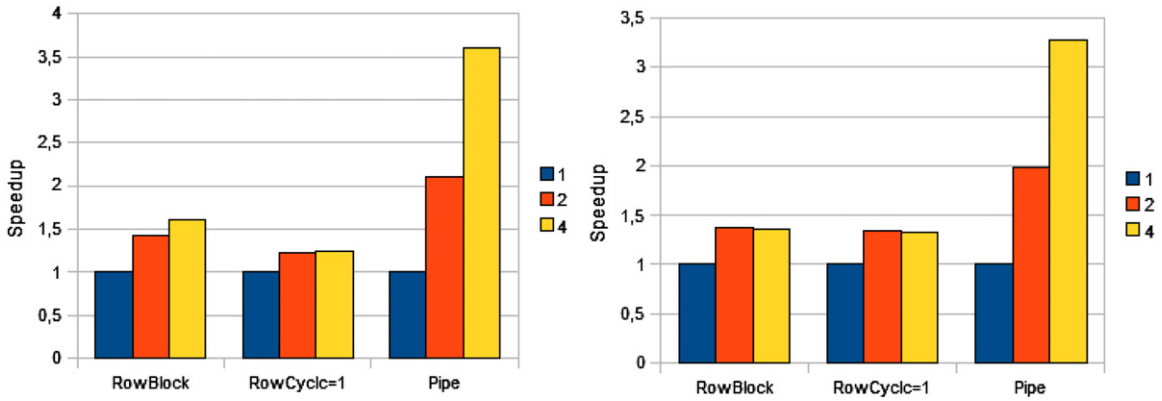

Fig. 7. Speedups of all parallel algorithms for 1–4 threads for matrix of size 512 (left) and matrix of size 4096 (right) for the Gaussian elimination method.

4.3. 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 for the *LU* form of Gaussian elimination of Eq. (4) is plotted in Fig. 8 using time parameters. Similarly, the performance model for the Gaussian elimination method is plotted in Fig. 9. In order to obtain these predicted results, we have determined the time parameters for division, elimination and communication 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 Fig. 10.

As can be seen from Figs. 8 and 9, 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.

5. Related work

Existing work on asynchronous-parallel algorithms for classic dense linear algebra covers the Cholesky, *LU*, and QR factorizations within the framework PLASMA, as well as the so-called "two-sided" transformations, Hessenberg, tridiagonal and bidiagonal reduction [3,29,30]. PLASMA relies on tile algorithms, which provide fine granularity parallelism and standard linear algebra algorithms can then be represented as a Directed Acyclic Graph (DAG) where nodes represent tasks, either panel factorization or update of a block-column, and edges represent dependencies among them. The programming model enforces asynchronous, out of order scheduling of operations. The implementations are based on some combination of schedulers and multicore programming frameworks based on domain-specific abstraction (e.g., SuperMatrix [29]), or hand-coded or pragma-directed schemes (e.g., SMP Superscalar (SMPPs) [30], Cilk [31]). The performance of the above implementations is marginally worse and this is due to the various overheads of multicore programming frameworks.

In addition, a pipelined model of parallel execution for dense matrix factorizations on the CELL processor and standard multicore processors was originally implemented in [24,25,32], respectively. This proposed idea combined with a look ahead technique, is applied to block formulations of the matrix factorizations, the Cholesky, the OR and the *LU* factorizations. This

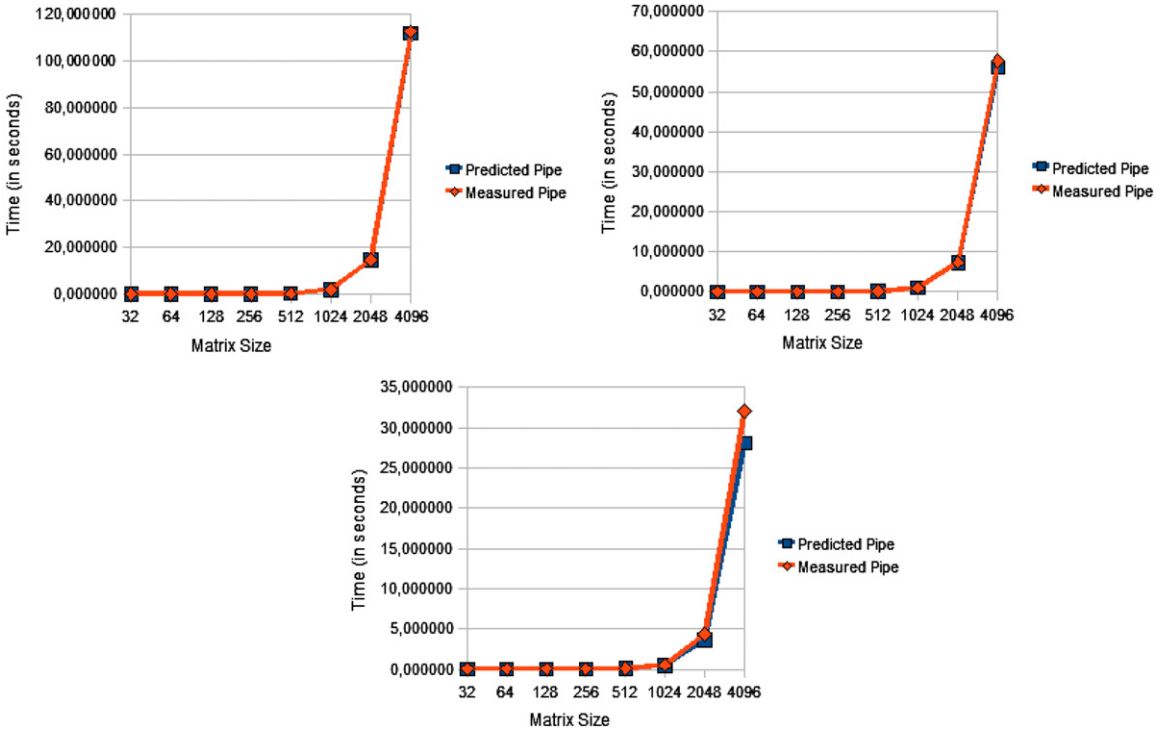

Fig. 8. The measured and predicted execution times of the pipeline algorithm on 1 core (top left), 2 cores (top right) and 4 cores (bottom) for the *LU* form of Gaussian elimination.

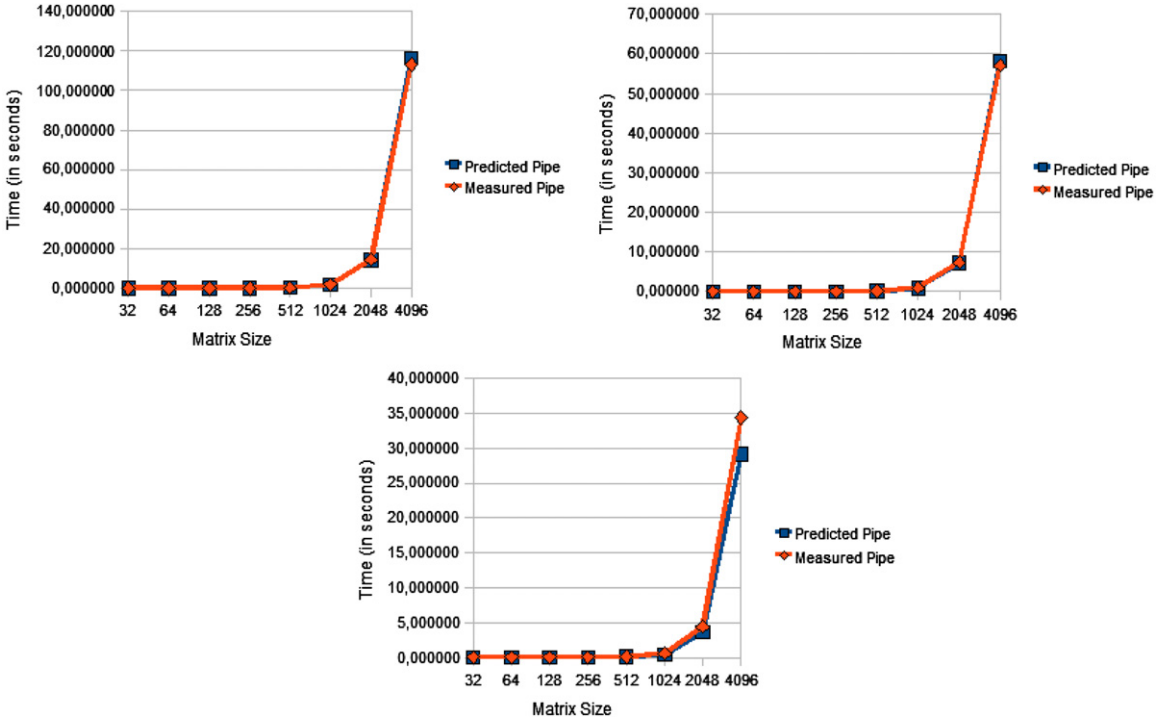

Fig. 9. The measured and predicted execution times of the pipeline algorithm on 1 core (top left), 2 cores (top right) and 4 cores (bottom) for the Gaussian elimination method.

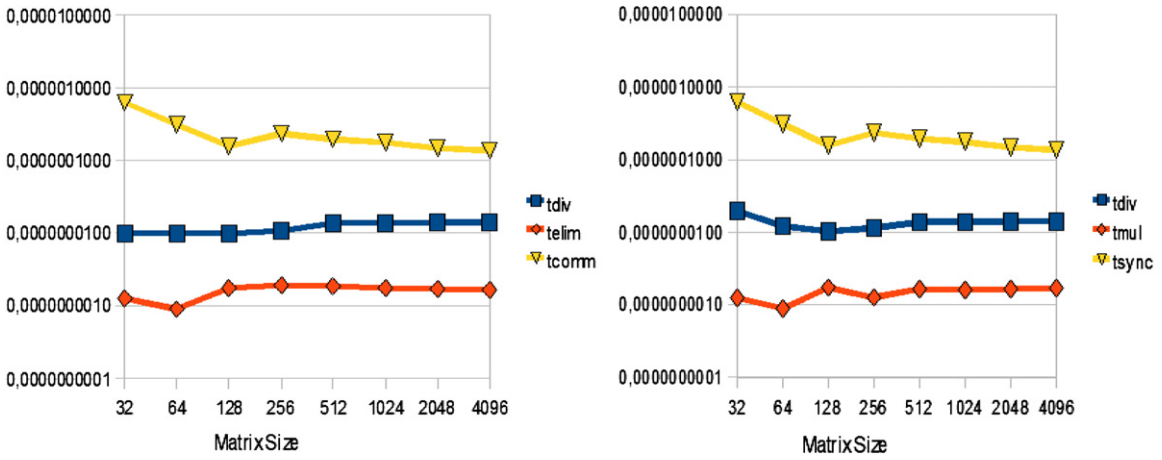

Fig. 10. Multicore execution performance as a function of the matrix size for operations division, elimination and communication on one core for the *LU* form of Gaussian elimination (left) and the Gaussian elimination (right).

pipeline implementation is a hand-written code using POSIX threads and primitive synchronization mechanisms (such as volatile progress table and busy-waiting [31]). This approach implements one look ahead, where look ahead of varying depth is implemented by processors which run out of work. Implementing more levels of look ahead would further complicate the code.

The present study contributes working experience in the implementation of pipeline technique for the Gaussian elimination algorithm and the *LU* version of Gaussian elimination which are based on 1D row partitioning scheme. Moreover, this work implements the pipelining technique in an OpenMP environment using simple queue data structure. The implementation of the queue is realized with the help of the proposed simple `Get()` and `Put()` procedures in C and OpenMP. To the best of our knowledge, there is no the implementation and performance study of the pipelining technique in an OpenMP environment for direct methods such as the Gaussian elimination algorithm and the *LU* form of Gaussian elimination on a multicore system. This presented work exploit the parallelism similar to the work [32] where the proposed technique of the queue in OpenMP is simple and provides good locality of reference and load balance for regular computation, like dense matrix computations. However, the technique of the queue is a potential scalability bottleneck but it does not pose problem on small-scale shared memory multicore processors for small to medium matrix sizes. Finally, to understand the performance of the pipelining method and to be able to predict performance on a target multicore system, we developed and carried out a performance model. More specifically, we presented our effort to quantify each component of the pipeline implementation according to a performance model that is able to predict the parallel execution time of a specific data distribution scheme for any problem size (i.e. matrix size), number of available processors/cores and processor characteristics. We must note that there are no published works with accurate performance models for the dense direct methods on multicore systems. The advantage of the proposed performance model is that it helps in predicting the execution time with very small complexity.

6. Conclusion

In this paper, we presented and evaluated three OpenMP parallel algorithms for the dense direct methods such as the Gaussian elimination method and the *LU* form of Gaussian elimination 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, 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 simple `Get()` and `Put()` procedures. Contrary to previous works, no work has been done to implement the pipeline technique for the dense direct methods 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 pipeline 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 direct methods with partial 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 more than four.

Table 3
Execution times (in seconds) of all parallel algorithms for variable matrix sizes and for the *LU* form of Gaussian elimination on 1 core.

	RowBlock	RowCyclc=1	Pipe
32	0.000080	0.000090	0.000070
64	0.000310	0.000320	0.000340
128	0.001560	0.001600	0.003000
256	0.018980	0.019450	0.029180
512	0.155090	0.156890	0.243670
1024	1.243250	1.249660	1.885020
2048	9.750050	9.771560	14.533280
4096	77.055230	77.083650	112.661830

Table 4
Execution times (in seconds) of all parallel algorithms for variable matrix sizes and for the *LU* form of Gaussian elimination on 2 cores.

	RowBlock	RowCyclc=1	Pipe
32	0.000150	0.000190	0.000080
64	0.000360	0.000590	0.000200
128	0.001210	0.002830	0.001500
256	0.009650	0.019260	0.014560
512	0.110520	0.133100	0.121450
1024	0.903170	0.960620	0.945970
2048	7.184030	7.358990	7.350170
4096	56.795430	57.486940	57.650460

Table 5
Execution times (in seconds) of all parallel algorithms for variable matrix sizes and for the *LU* form of Gaussian elimination on 4 cores.

	RowBlock	RowCyclc=1	Pipe
32	0.000170	0.000200	0.000070
64	0.000370	0.000580	0.000150
128	0.001030	0.002640	0.000620
256	0.004590	0.017600	0.005180
512	0.092330	0.128780	0.068130
1024	0.890840	0.953610	0.485940
2048	7.174530	7.273750	4.310110
4096	56.672300	56.396170	32.041210

Table 6
Execution times (in seconds) of all parallel algorithms for variable matrix sizes and for the Gaussian elimination method on 1 core.

	RowBlock	RowCyclc=1	Pipe
32	0.000080	0.000090	0.000080
64	0.000300	0.000310	0.000360
128	0.002810	0.002680	0.003450
256	0.021680	0.021280	0.033280
512	0.165940	0.159440	0.251630
1024	1.244720	1.223860	1.892170
2048	9.639700	9.466650	14.411060
4096	76.125190	74.720080	113.098470

Acknowledgments

The authors would like to thank the anonymous reviewers for many helpful comments and suggestions, which have greatly improved the presentation of this paper.

Appendix

See Tables 3–8.

Table 7
Execution times (in seconds) of all parallel algorithms for variable matrix sizes and for the Gaussian elimination method on 2 cores.

	RowBlock	RowCyclc=1	Pipe
32	0.000160	0.000200	0.000090
64	0.000380	0.000620	0.000240
128	0.001480	0.003230	0.001220
256	0.013110	0.019510	0.014390
512	0.115890	0.131120	0.119740
1024	0.899570	0.949510	0.944960
2048	7.055990	7.151610	7.318420
4096	55.293920	55.842480	57.018130

Table 8
Execution times (in seconds) of all parallel algorithms for variable matrix sizes and for the Gaussian elimination method on 4 cores.

	RowBlock	RowCyclc=1	Pipe
32	0.000180	0.000220	0.000090
64	0.000400	0.000630	0.000180
128	0.001110	0.002910	0.000650
256	0.011980	0.018630	0.006180
512	0.113390	0.128870	0.069760
1024	0.907540	0.960260	0.563050
2048	7.130180	7.264190	4.389910
4096	56.108300	56.065810	34.432080

References

[1] M. Cosnard, D. Trystram, Parallel Algorithms and Architectures, International Thomson Pub., 1995.
[2] A. Buttari, J. Langou, J. Kurzak, J.J. Dongarra, Parallel tiled QR factorization for multicore architectures, Concurrency and Computation: Practice and Experience 20 (13) (2008) 1573–1590.
[3] A. Buttari, J. Langou, J. Kurzak, J.J. Dongarra, A class of parallel tiled linear algebra algorithms for multicore architectures, Parallel Computing Systems & Applications 35 (2009) 38–53.
[4] A. Buttari, J.J. Dongarra, P. Husbands, J. Kurzak, K. Yelick, Multithreading for synchronization tolerance in matrix factorization, in: Proc. of Scientific Discovery through Advanced Computing, Journal of Physics: Conference Series 78 (2007) 012028.
[5] E. Anderson, Z. Bai, C. Bischof, L.S. Blackford, J.W. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, D. Sorensen, LAPACK Users Guide, SIAM, 1992.
[6] L.S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, R.C. Whaley, ScaLAPACK Users Guide, SIAM, 1997.
[7] E. Elmroth, F.G. Gustavson, Applying recursion to serial and parallel QR factorization leads to better performance, IBM Journal of Research and Development 44 (4) (2000) 605–624.
[8] E. Elmroth, F.G. Gustavson, New serial and parallel recursive QR factorization algorithms for SMP systems, in: Proc. of International Workshop Applied Parallel Computing, Large Scale Scientific and Industrial Problems, in: Lecture Notes in Computer Science, vol. 1541, 1998, pp. 120–128.
[9] E. Elmroth, F.G. Gustavson, High performance library software for QR factorization, in: Proc. of 5th International Workshop Applied Parallel Computing, New Paradigms for HPC in Industry and Academia, in: Lecture Notes in Computer Science, vol. 1947, 2000, pp. 53–63.
[10] B.C. Gunter, R.A. van de Geijn, Parallel out-of-core computation and updating the QR factorization, ACM Transactions on Mathematical Software 31 (1) (2005) 60–78.
[11] M. Marques, G. Quintana-Orti, E.S. Quintana-Orti, R. van de Geijn, Out-of-core computation of the QR factorization on multi-core processors, in: Proc. of the 2009 Euro-Par Parallel Processing, in: Lecture Notes in Computer Science, vol. 5704, 2009, pp. 809–820.
[12] S.F. McGinn, R.E. 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.
[13] C.J. Thompson, S. Hahn, M. Oskin, Using modern graphics architectures for general computing: a framework and analysis, in: Proc. 35th IEEE/ACM Int'l Symposium Microarchitecture, 2002, pp. 306–317.
[14] E.S. Larsen, D. McAllister, Fast matrix multipliers using graphics hardware, in: Proc. High Performance Networking and Computing Conference, 2001, p. 55.
[15] R.C. Whaley, A. Petitet, J.J. Dongarra, Automated empirical optimizations of software and the ATLAS project, Parallel Computing 27 (2001) 3–35.
[16] J.D. Hall, N.A. Carr, J.C. Hart, Cache and bandwidth aware matrix multiplication on the GPU, Technical Report UIUCDCS-R-2003-2328, University of Illinois, 2003.
[17] K. Fatahalian, J. Sugerman, P. Hanrahan, Understanding the efficiency of GPU algorithms for matrix-matrix multiplication, in: Proc. SIGGRAPH/EUROGRAPHICS Workshop Graphics Hardware, 2004, pp. 133–137.
[18] J. Kruger, R. Westermann, Linear algebra operators for GPU implementation of numerical algorithms, ACM Transactions on Graphics 22 (2003) 908–916.
[19] A. Moravanszky, Dense matrix algebra on the GPU, 2003. http://www.shaderx2.com/shaderx.pdf.
[20] V. Volkov, J.W. Demmel, Benchmarking GPUs to tune dense linear algebra, in: Proc. of the 2008 ACM/IEEE Conference on Supercomputing, 2008, pp. 1–11.
[21] N. Galoppo, N.K. Govindaraju, M. Henson, D. Manocha, *LU*-GPU: efficient algorithms for solving dense linear systems on graphics hardware, in: Proc. of the 2005 ACM/IEEE Conference on Supercomputing, 2005, p. 3.
[22] I. Fumihiko, M. Manabu, G. Keigo, H. Kenichi, Performance study of *LU* decomposition on the programmable GPU, in: Proc. of the High Performance Computing, 2005, pp. 83–94.
[23] G. Quintana-Orti, F.D. Igual, E.S. Quintana-Orti, 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.
[24] J. Kurzak, J.J. Dongarra, QR factorization for the CELL processor, Scientific Programming 17 (1–2) (2009) 31–42.

[25] J. Kurzak, A. Buttari, J. Dongarra, Solving systems of linear equation on the CELL processor using Cholesky factorization, IEEE Transactions on Parallel and Distributed Systems 19 (9) (2008) 1175–1186.
[26] B.C. Vishwas, A. Gadia, M. Chaudhuri, Implementing a parallel matrix factorization library on the cell broadband engine, Scientific Programming 17 (1–2) (2009) 3–29.
[27] E. Agullo, J. Demmel, J. Dongarra, N. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, S. Tomov, Numerical linear algebra on emerging architectures: the PLASMA and MAGMA projects, Journal of Physics: Conference Series 180 (2009).
[28] G.H. Golub, Ch. Van Loan, Matrix Computations, Johns Hopkins University Press, 1996.
[29] E. Chan, E.S. Quintana-Orti, G. Quintana-Orti, R. van de Geijn, SuperMatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures, in: Proc. SPAA, 2007, pp. 116–125.
[30] H. Ltaeif, J. Kurzak, J. Dongarra, Scheduling two-sided transformations using algorithms-by-tiles on multicore architectures, Technical Report UT-CS-09-637, Univ. of Tenn., Knoxville, 2009.
[31] J. Kurzak, H. Ltaief, J. Dongarra, R.M. Badia, Scheduling dense linear algebra operations on multicore processors, Concurrency and Computation: Practice and Experience 22 (1) (2010) 15–44.
[32] J. Kurzak, J. Dongarra, Implementing linear algebra routines on multi-core processors with pipelining and a look ahead, in: Proc. of the 8th International Conference on Applied Parallel Computing: State of the Art in Scientific Computing, PARA 2006, in: Lecture Notes in Computer Science, vol. 4699, 2006, pp. 147–156.