High Performance Recursive Matrix Inversion for Multicore Architectures

Publisher: IEEE

Ryma Mahfoudhi ; Sami Achour ; Olfa Hamdi-Larbi ; Zaher Mahjoub **All Authors** •••

Abstract:
There are several approaches for computing the inverse of a dense square matrix, say A, namely Gaussian elimination, block wise inversion, and LU factorization (LUF). The... **View more**

Metadata
Abstract:
There are several approaches for computing the inverse of a dense square matrix, say A, namely Gaussian elimination, block wise inversion, and LU factorization (LUF). The latter is used in mathematical software libraries such as SCALAPACK, PBLAS and MATLAB. The inversion routine in SCALAPACK library (called PDGETRI) consists, once the two factors L and U are known (where ALU), in first inverting U (PDGETRF) then solving a triangular matrix system giving A-1. A symmetric way consists in first inverting L, then solving a matrix system giving A-1. Alternatively, one could compute the inverses of both U and L, then their product and get A-1. On the other hand, the Strassen fast matrix inversion algorithm is known as an efficient alternative for solving our problem. We propose in this paper a series of different versions for parallel dense matrix inversion based on the 'Divide and Conquer' paradigm. A theoretical performance study permits to establish an accurate comparison between the designed algorithms. We achieved a series of experiments that permit to validate the contribution and lead to efficient performances obtained for large matrix sizes i.e. up to 40% faster than SCALAPACK.

Published in: 2017 International Conference on High Performance Computing & Simulation (HPCS)

Date of Conference: 17-21 July 2017

DOI: 10.1109/HPCS.2017.104

Date Added to IEEE *Xplore*: 14 September 2017

Publisher: IEEE

ISBN Information:

Conference Location: Genoa, Italy

Contents

SECTION I.
Introduction

The continuous spread of multicore architecture make the review and the remodel of numerical libraries unavoidable to ensure full exploitation of the massive parallelism as well as the memory hierarchy design offered by this architecture. The LAPACK library has shown some weakness on such platforms since it can only reach a modest ratio of the theoretical performance [1] [2]. The reasons behind this are essentially

the overhead of its fork-join model of parallelism and the chosen coarse-grained task granularity.

It turns out that Dense Matrix Inversion (DMI) is a basic kernel used in several scientific applications such as model reduction and optimal control. In addition, it requires a high computational effort due to its cubic complexity. That is why several works addressed the design of efficient DMI algorithms. Our main objective here is in fact the design of an efficient parallel algorithm for MI that outperforms the SCALAPACK/PBLAS routines and which is based on both the fast Strassen method for MI and the LU factorization.

The reminder of the paper is organized as follows. Section 2 gives an overview on previous related works. Section 3 recalls block matrix inversion algorithms. Section 4 describes our parallel algorithms. Section 5 is devoted to the parallel implementations of these algorithms on shared-memory architectures and presents the performance results. Finally, Section 6 summarizes our contribution and proposes future perspectives.

SECTION II.
Related Works

This Section presents previous similar works as far as parallel matrix inversion implementation is concerned. In [3], Ezzati & al implemented two approaches for DMI, the first based on Gaussian elimination (i.e. LU factorization) and the second based on Gauss-Jordan elimination (GJE). Both approaches present similar computational cost but different properties. More recently, Benner & al [4] tackle the inversion of large-scale dense matrices via conventional matrix factorizations (LU, Cholesky, and LDLT) and the Gauss-Jordan method on hybrid platforms consisting of a multicore CPU and a many-core graphics processor (GPU). In [5], the authors presented a novel approach for LU factorization on multicore platform outperforming LAPACK and PLASMA libraries. Their approach used a parallel recursive fine-grained of the panel factorization. In [6], two blocked algorithms for dense matrix multiplication are presented and implemented on a shared memory architecture. The first is based on blocked matrices and the second used blocked matrices with the MapReduce framework. The experimental study shows that the designed algorithms outperform the LAPACK implementation by up to 50%.

In our previous work [7], we addressed the parallelization of a recursive algorithm for triangular matrix inversion (TMI) based on the 'Divide and Conquer' (D&C) paradigm. The theoretical study was validated by a series of experiments achieved on homogeneous and heterogeneous platforms. In [8], we studied the sequential algorithm. We also proposed a series of different versions for DMI based on the DlC paradigm. A theoretical performance study permitted to establish a comparison between the designed algorithms. We could achieved the goal of outperforming the efficiency of the well-known BLAS and LAPACK libraries for DMI.

This paper complements these efforts by parallelizing the above recursive algorithms and studying their practical performances.

SECTION III.
Recursive Algorithm for DMI

A. Strassen Algorithm

Let A and B an n*n real matrices. The number of scalar operations required for computing the matrix product C = AB by the standard method is $2n^3=O(n^3)$. In his seminal paper [9], Strassen introduced an algorithm for matrix multiplication, based on the D&C paradigm, whose complexity is only $O([1^{\log_2 7})$. This algorithm is based on block decomposition of both matrix A and its inverse.

For sake of simplicity let $n=2^q$ where q>1 (the general case where n is not a power of 2 may easily be generalised). A is partitioned into 4 sub-matrices of size k=n/2 as in figurel.

The number of matrix multiplications required for computing blocks X_{11}, X_{12}, X_{21} and X_{22} in the block form may be decreased below $O(n^3)$ through using the seven temporary matrices R_1, ..., R_7 according to the following relations [9] [10]:

(1) $R_1 = A_{11}^{-1}$

(2) $R_2 = A_{21}R_1$

(3) $R_3 = R_1A_{12}$

(4) $R_4= A_{21}R_3$

(5) $R_5 = R_4- A_{22}$

(6) $R_6 = R_5^{-1}$

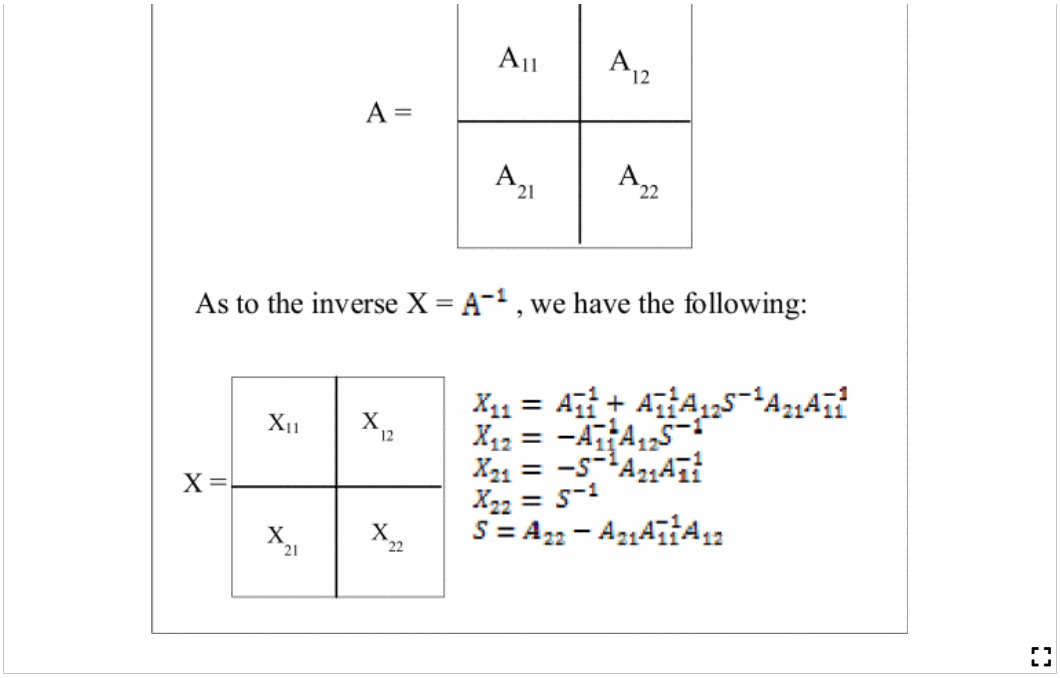


Figure 1.
Matrix decomposition for computing the inverse

The formulae may be used for a recursive computation of the inverse X. Both relations (1) and (6) perform inversions of matrices of smaller sizes (k=n/2). By recursively applying the same formulae on these submatrices, we derive the Strassen method for dense matrix inversion (DMI). The recursion may be pursued down to matrices of size 1. The original Strassen DMI algorithm is in fact based on the two following principles:

- P1. In steps 1 and 6, recursively compute the inverses of smaller size matrices. Recursion is continued down to size 1;
- P2. Use Strassen matrix-matrix multiplication (MM) method to perform all matrix multiplications (steps 2-4,7-9).

B. LU Factorisation Algorithm

LU factorization (LUF) is used in mathematical software libraries such as LAPACK xGETRI and MATLAB inv. The xGETRI method consists, once the two factors L and U are known (where A=LU), in inverting U then solving the triangular matrix system XL = U^{-1} (i.e. $L^T X^T = (U^{-1})^T$, thus X = A^{-1}. A symmetric way consists in first inverting L, then solving the matrix system UX = L^{-1} for X thus X=A^{-1}. Alternatively, one could invert both L and U, then compute the product X = U^{-1} L^{-1}. Each of these procedures involves at least one triangular matrix inversion (TMI).

As previously mentioned, three methods may be used to perform a DMI through LUF. The first one requires two triangular matrix inversions (TMI) and one triangular matrix multiplication (TMM) i.e. an upper one by a lower one. The two others require one triangular matrix inversion (TMI) and a triangular matrix system solving (TMSS)

The optimal LUF_DMI algorithm is based on the following steps:

- Recursively decompose the matrix, and recursion is continued down to size 1;
- Invert the upper triangular matrix
- Solve the system XL=U^{-1}

1) Recursive LUF

A blocked algorithm for LUF was proposed since 1974 [10]. Given a square matrix A of size n, the L and U

factors verifying A=LU may be computed with Algorithm1 as follows.

2) Triangular Matrix Inversion (TMI)

We first recall that the well-known standard algorithm (SA) for inverting a (lower or upper) triangular matrix, say A of size n, consists in solving n triangular systems. The complexity of (SA) is $n^3/3+n^2/2+n/6$ [11].

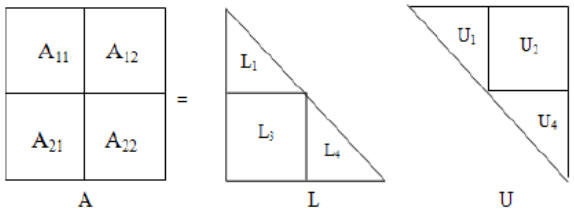
Using the D&C paradigm, Heller proposed in 1973 a recursive algorithm [12] for TMI. His main idea consists in decomposing matrix A as well as its inverse B (both of size n) into 3 sub-matrices of size n/2 (see Figure 2, where A is assumed to be lower triangular). The procedure is recursively repeated until reaching sub-matrices of size 1. We hence deduce:

$$B_1 = A_1^{-1}, B_3 = A_3^{-1}, B_2 = -B_3 A_2 B_1$$

View Source

Algorithm 1 LUF

*Input : A, n*n matrix*
Output : L : lower triangular matrix, U upper triangular matrix
Begin
If *(n=1)* ***Then***
L=1; U=A
Else */* split matrices into four blocks of sizes n/2*



$(L_1, U_1) = LUF(A_{11})$
U_2 = TMSS (L_1, A_{12}) / Triangular Matrix System Solving*
$L_3 = A_{21}U_1^{-1}$; $H = A_{22} - L_3U_2$; $(L_4, U_4) = LUF(H)$

Endif
End

Therefore, inverting a triangular matrix of size n consists in inverting 2 triangular submatrices of size *n/2* followed by two triangular-dense matrix products of size *n/2*.

In [13] Nasri & al. proposed a slightly modified version of the above algorithm. The authors showed that since we may write $A_3Q=A_2$, $B_2A_1=-Q$ hence, instead of two matrix products needed to compute matrix B_2, we have to solve 2 triangular matrix systems of size n/2 i.e. A3Q =A2 and $A_1^TB_2^T= -Q^T$. We precise that both versions are of $n^3/3$ +O (n^2) complexity.

In [14], we presented a fast recursive algorithm for triangular matrix system solving (RATSS) that outperforms the BLAS and LAPACK kernels for triangular matrix inversion.

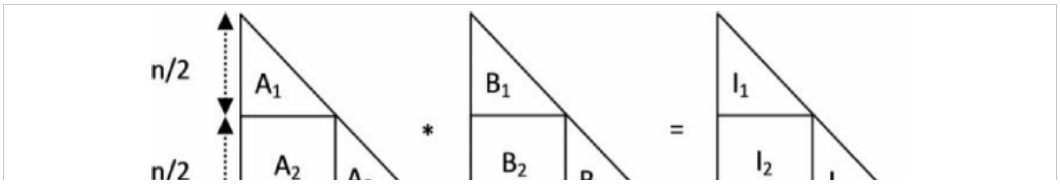


Figure 2.
Matrix decomposition in heller's algorithm

3) Triangular Matrix System Solving (TMSS)
We now discuss the implementation of solvers for triangular matrix systems with matrix right hand side (or equivalently left hand side). This kernel is commonly named trsm in the BLAS convention. In the Algorithm 2, we'll consider, without loss of generality, the resolution of a lower triangular matrix system with matrix right hand side (AX=B).

SECTION IV.
Recursive Parallel Algorithm for DMI

Block decomposition as well as fme-grain computations are becoming necessary in multicore environments. In order to well exploit the advantages of the hardware specifications, we must use one of the two methodologies. This is in fact exactly the purpose of recursive algorithms in dense linear algebra. The main idea is to split the original problem into sub-problems whose solving corresponds to tasks that are distributed among different processors.

A. Parallel Strassen-Based Matrix Inversion Algorithm

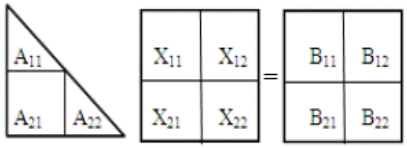
Algorithm 3 describes the parallel Strassen implementation. This Algorithm is designed by recursively applying the (block) decomposition until reaching a threshold size $n/2^k$ (the size beyond which recursion is not optimal).

Notations:

PStrassen Parallel Strassen for DMI
PMM Parallel Matrix Multiplication
PMS Parallel Matrix Substruction

Algorithm 2 TMSS

Begin
If *(n=1)* ***Then***
$X = B/A$
Else */* split matrices into four blocks of sizes n/2*



$X_{11} = TMSS(A_{11},B_{11})$
$X_{12} = TMSS(A_{11},B_{12})$
$X_{21} = TMSS(A_{22}, B_{21}\text{-}MM(A_{21},X_{11}))$
$X_{22} = TMSS(A_{22}, B_{22}\text{-}MM(A_{21},X_{12}))$

Endif
End

Algorithm 3 Pstrassen (A,N)

```
Begin
  If (n<n_k) then
    Dgetrf(A,n)
    Dgetri(A,n)
  Else /* split matrices into four blocks of  sizes n/2
    R1  = PStrassen (A11,n/2)
    R2  = PMM(A21,R1,n/2)
    R3  = PMM(R1,A12,n/2)
    R4  = PMM(A21,R3,n/2)
    R5  = PMS(R4,A22,n/2)
    R6  = PStrassen (R5,n/2)
    X12 = PMM(R3,R6,n/2)
    X21 = PMM(R6,R2,n/2)
    R7  = PMM(R3,X21,n/2)
    X11 = PMS(R1,R7,n/2)
    X22 = PMS(0,R6,n/2)
  Endif
End
```

Computing each one of the eleven matrices R_1... R_7, X_{11}, X_{12}, X_{21} and X_{22} corresponds to a task. We depict in Figure 3 the task directed acyclic graph (DAG) corresponding to the precedence tasks relations.

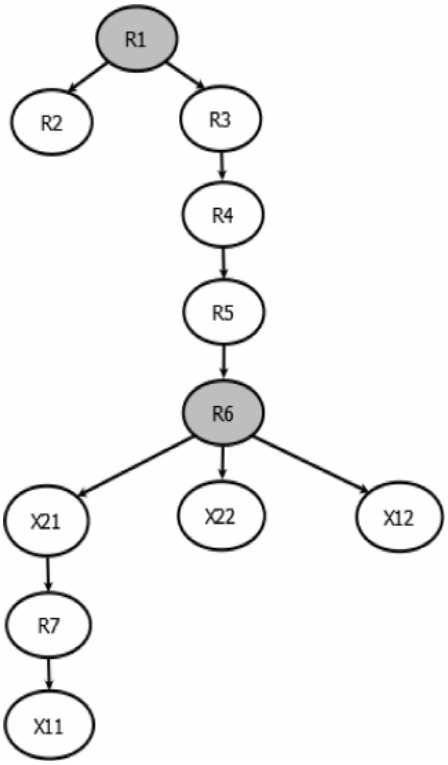


Figure 3.
Task DAG for pstrassen algorithm

If we consider PMM and PMS and the tasks they involve, it is easy to notice that the corresponding task DAGs are completely disconnected i.e. the tasks are completely independent thus it is possible to distribute them among the processors [15].

As to PStrassen, we underline that at the last level of recursion, tasks R1 and R2 will be assigned to a single processor.

The performance evaluation we developed consists in first measuring the speed-up S=Ts/Tp where Ts is the complexity of the sequential algorithm and Tp that of the parallel one when p processors; We then deduce the corresponding efficiency E=S/p.

We can easily notice that from the first recursion level, three-quarters of the work is completely parallel and the tasks distribution is perfectly balanced among the processors. Thus, neglecting communication times, the theoretical speed-up and efficiency for p processors may be written as follows:

$$S = 1/(\frac{3}{4p} + \frac{1}{4}), \quad \mathrm{E} = 1/(\frac{3}{4} + \frac{p}{4}) \text{i.e.}$$
$$S = 4p/(p+3), \quad \mathrm{E} =^{4} /(p+3)$$

View Source

For two levels of recursion we get:

$$S = 16p/(p+15), \quad \mathrm{E} = 16/(p+15)$$

View Source

We deduce for k ($1 \leq k < \log_2 n$) levels:

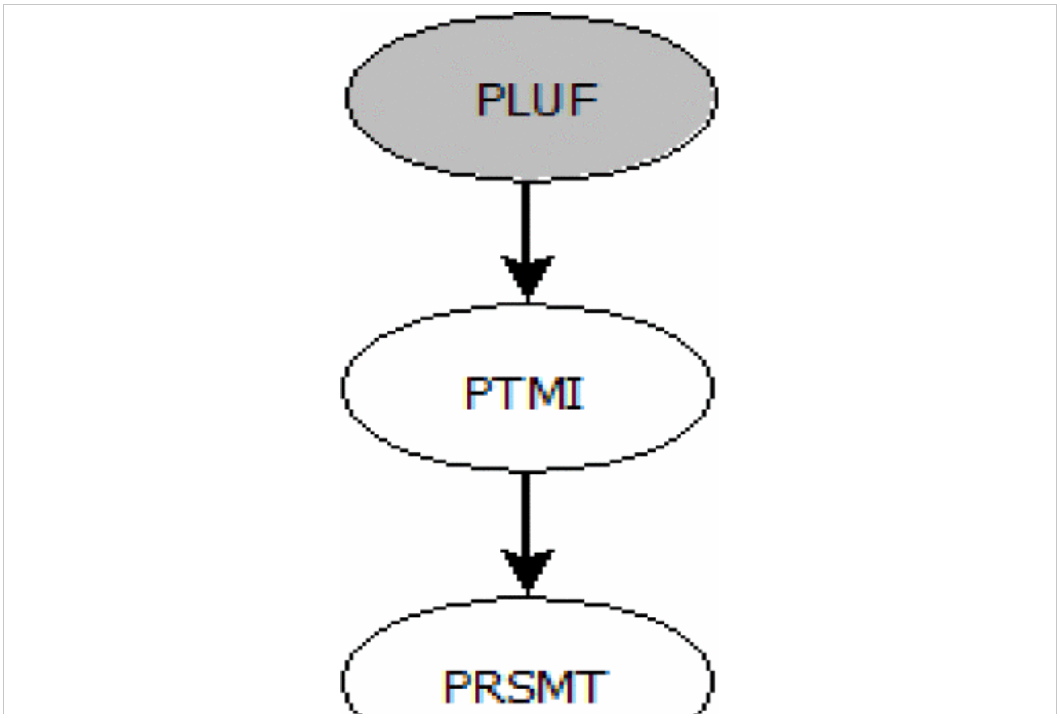
$$S = 4^k p/(p+4^k-1), \quad \mathrm{E} = 4^k/(p+4^k-1)$$

View Source

We depict in table I, S and E for particular values of p and k.

B. Block LU-Based Matrix Inversion Algorithm

The inversion procedure is decomposed into the three successive steps: (i) performing the LU factorization (PLUF), (ii) inverting the upper triangular U factor (PTMI), (iii) solving a linear matrix system (PTMSS) (see figures 4 and 5)



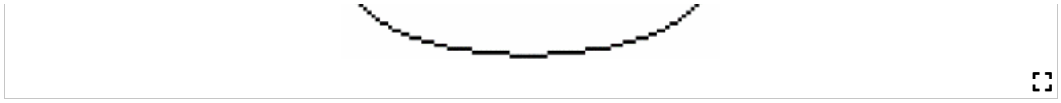


Figure 4.
Task DAG for PLUF-DMI

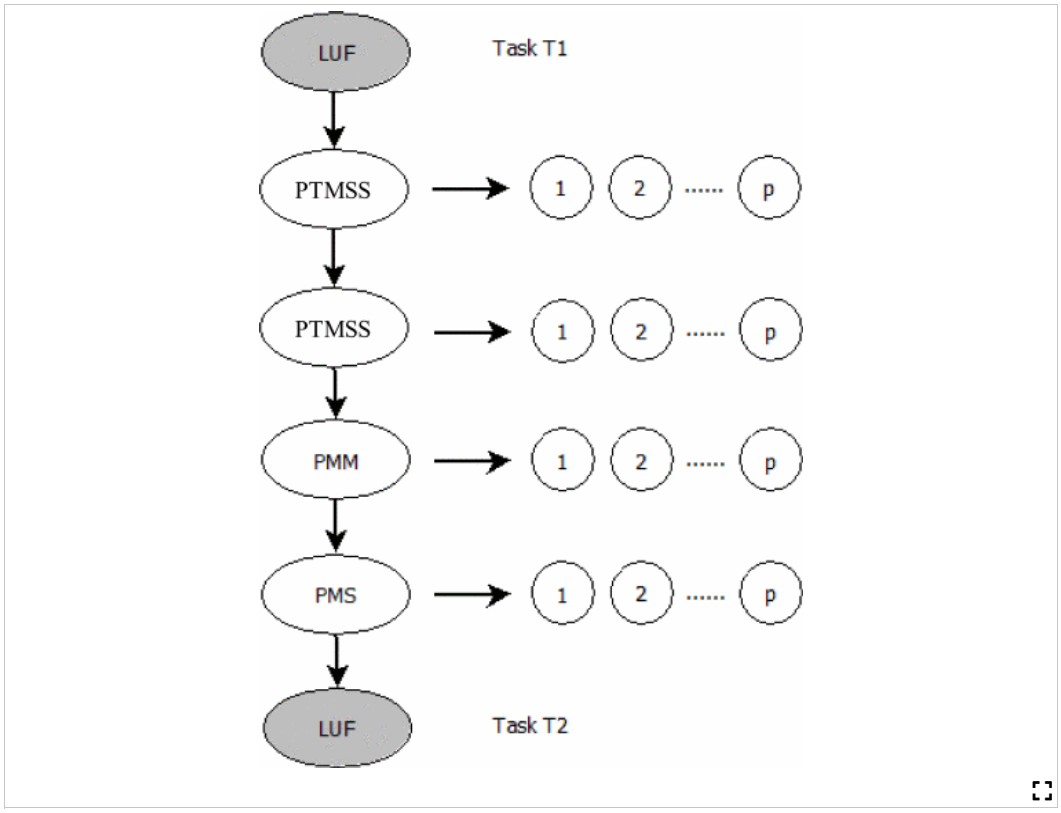


Figure 5.
Task DAG for the first step: LU factorization (LUF)

Table I. Speed-up and efficiency for different values of p and k - strassen algorithm (pstrassen)

p	k	S	E
2	1	1.6	0.80
2	2	1.88	0.94
2	4	1.99	0.99
4	1	2.28	0.57
4	2	3.36	0.84
4	4	3.95	0.98
8	1	2.90	0.36
8	2	5.56	0.69
8	4	7.78	0.97
16	1	3.36	0.21
16	2	8.25	0.51
16	4	15.11	0.94

Concerning PMM and PMS, PTMSS and parallel triangular matrix inversion (PTMI), it is easy to notice that the corresponding task DAG is completely disconnected [14] [15] i.e. the corresponding tasks are independent thus it is possible to distribute the work among the processors.

According to Figure 5, we can remark that last level of recursion, task T1 as well task task T2 will be assigned to a single processor i.e. each is serially executed.

From Figure 4 we can deduce that two thirds of the work are completely parallel and the associated tasks may be distributed with a perfect balancing between processors. The remainder of the work i.e. the last third corresponding to LU factorization, also involves a completely parallel part.

From the first recursion level, we can remark that 11/12 of the work is completely parallel. Thus, neglecting communication times, the theoretical speed-up S and efficiency E for p processors are as follows:

$$S = 12p/(p+11), \quad \mathrm{E} = 12/(p+11)$$

View Source

```
\begin{equation*} S=12p/{(p+11)},\ \ \ \mathrm{E}=12/{(p+11)} \end{equation*}
```

For two levels of recursion, we have:

$$S = 144p/(p+143), \quad \mathrm{E} = 144/(p+143)$$

View Source

And for k ($1 \leq k < \log_2 n$) levels:

$$S = 12^k p/(p+12^k-1), \quad \mathrm{E} = 12^k/(p+12^k-1)$$

View Source

We depict in table II, both speed-up and efficiency for particular values of p and k.

C. Results Analysis

The analysis of the results and according to Tables **I & II**, we may remark the following:

- For fixed number of processors (p), the speed-up (S) and the efficiency (E) increase with the recursion level (k). S (resp. E) converges to its optimal value p (resp. 1).
- For fixed recursion level (k), the speed-up (S) and the efficiency (E) decrease with the number of processors (p).
- PLU-DMI is more efficient than PStrassen
- For fixed p, the efficiency (E) increases with recursion level and it converges to its optimal value i.e. **1**.

Table II. Speed-up and efficiency for different values of p and k - block lu-based algorithm (plu-dmi)-

p	k	S	E
2	1	1.84	0.92
2	2	1.98	0.99
2	4	1.99	0.99
4	1	3.20	0.80
4	2	3.91	0.97
4	4	3.99	0.99
8	1	5.05	0.63
8	2	7.62	0.95
8	4	7.99	0.99
16	1	7.11	0.44
16	2	14.49	0.90
16	4	15.98	0.99

In the next section, we describe an experimental study achieved through an implementation of the two described recursive algorithms. A comparison is then done in order to see to which extent the experimental study confirms the theoretical one.

SECTION V.
Performance Results

A. Hardware Description and Setup

Our experiments were performed on two shared memory systems (SMS). We underline that SMSs are representative of a vast class of servers and workstations commonly used for computationally intensive workloads. The first SMS is a quad core bi-processor (8 cores/2.5 GHz/12MB cache Memory (L2)). The second SMS is an AMD Opteron 6376 processor (16 cores/ 2.3 GHz/ 16MB cache Memory (L2) and 16MB (L3)). OpenMP under Ubuntu 16.04 OS was used. The performance evaluation consists in measuring the

execution times of the parallel algorithm for n in the range [1000]–[2400] and p=2, 4, 8 and 16 processors. We then deduce the corresponding efficiency, denoted E(p,n). Remark that for each n and p, successive runs were achieved and the mean time is chosen.

B. Expermintal Study

We discuss in this section the variations of the efficiencies (E) in terms of the matrix size n. Algorithms parallel blas (Pblas), parallel Strassen (PStrassen) and parallel LU (PLU) for MI (for different values of processor) were implemented.

We limit here our description to some excerpts. In Tables III, IV, V and VI we give for some values of n, the execution time (Ext) of blas (Sequential), Pblas, PStrassen and PLU and the two following ratios:

- ρ_1 =Ext(PStrassen)/Ext(Pblas)
- ρ_2 =Ext(PLU-DMI)/Ext(Pblas) for p=2, 4, 8, 16.

In figures 6, 7, 8 and 9 we give for different values of n, the efficiencies of Pblas and PStrassen for p=2, 4, 8, 16.

1) First Target Platform

a) $p = 2$

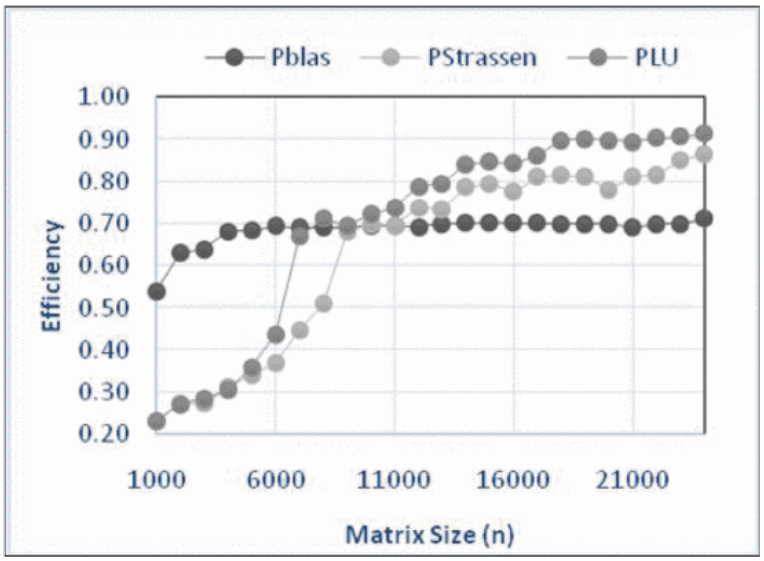


Figure 6.
Efficiency of pblas, pstrassen and PLU-DMI for p=2 and different values of n

Table III. Execution time and ratio for p=2 and different values of N

n	blas	Pblas	PStrassen	PLU-DMI	ρ_1	ρ_2
1 000	1.17	1.08	2.55	2.55	0.43	0.43
4000	81.69	60.07	131.76	136.15	0.45	0.45
8000	628.84	455.68	616.51	442.84	1.03	1.03
2000	2132.42	1545.23	1440.83	1349.63	1.14	1.14
16000	5466.48	3904.63	3504.15	3253.86	1.20	1.20
20000	6126.97	4376.41	3927.55	3403.87	1.28	1.28
24000	17324.85	12200.60	9956.81	9519.15	1.28	1.28

b) $p = 4$

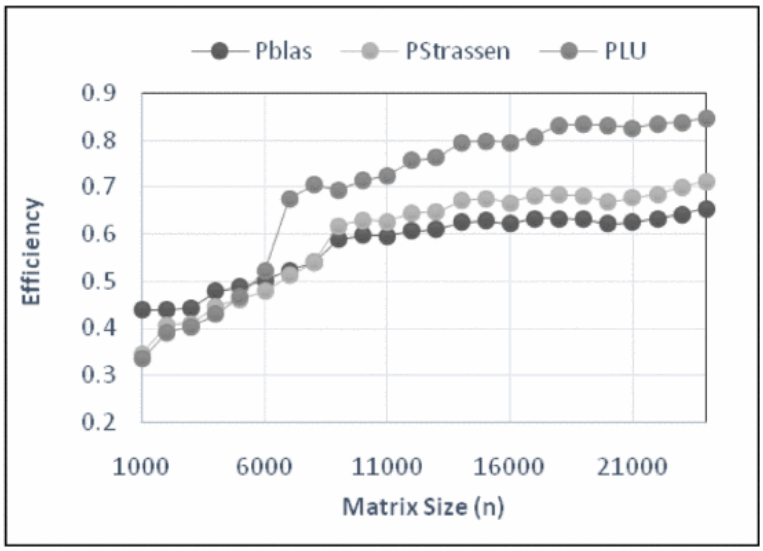


Figure 7.
Efficiency of pblas, pstrassen and PLU-DMI for p=4 and different values of n

Table IV. Execution time and ratio for p=4 and different values of N

n	blas	Pblas	PStrassen	PLU-DMI	ρ_1	ρ_2
1 000	1.17	0.67	0.84	0.86	0.79	0.76
4000	81.69	42.55	45.38	47.50	0.93	0.9
8000	628.84	291.13	291.13	224.58	1	1.31
12000	2132.42	873.94	820.16	701.45	1.06	1.25
16000	5466.48	2204.23	2039.73	1708.27	1.07	1.28
20000	6126.97	2470.55	2286.18	1845.47	1.07	1.34
24000	17324.85	6663.41	6100.30	5095.55	1.09	1.29

c) $p = 8$

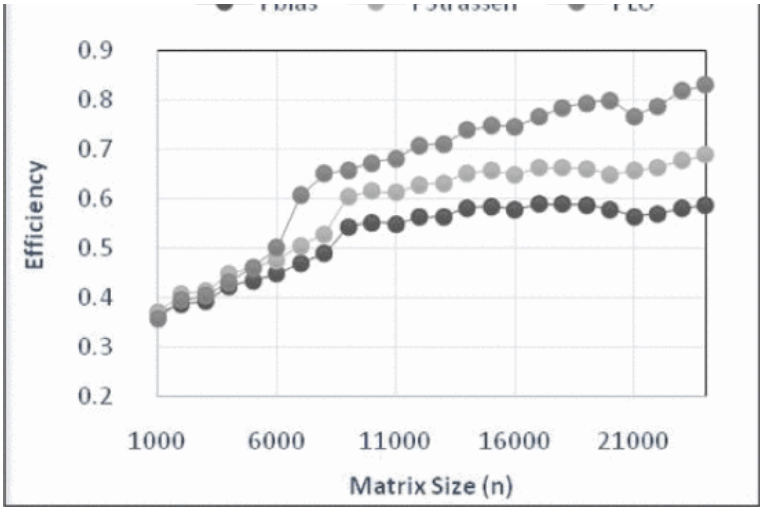


Figure 8.
Efficiency of pblas, pstrassen and PLU-DMI for p=8 and different values of n

Table V. Execution time and ratio for p=8 and different values of N

n	blas	Pblas	PStrassen	PLU-DMI	ρ_1	ρ_2
1 000	1.17	0.40	0.40	0.41	1	0.97
4000	81.69	24.31	22.69	23.75	1.06	1.02
8000	628.84	160.42	148.31	120.93	1.08	1.33
12000	2132.42	475.99	423.10	375.43	1.12	1.26
16000	5466.48	1178.12	1051.25	911.08	1.12	1.29
20000	6126.97	1320.47	1178.26	957.34	1.12	1.38
24000	17324.85	3670.52	3138.56	2609.16	1.17	1.41

2) Second Platform: $p = 16$

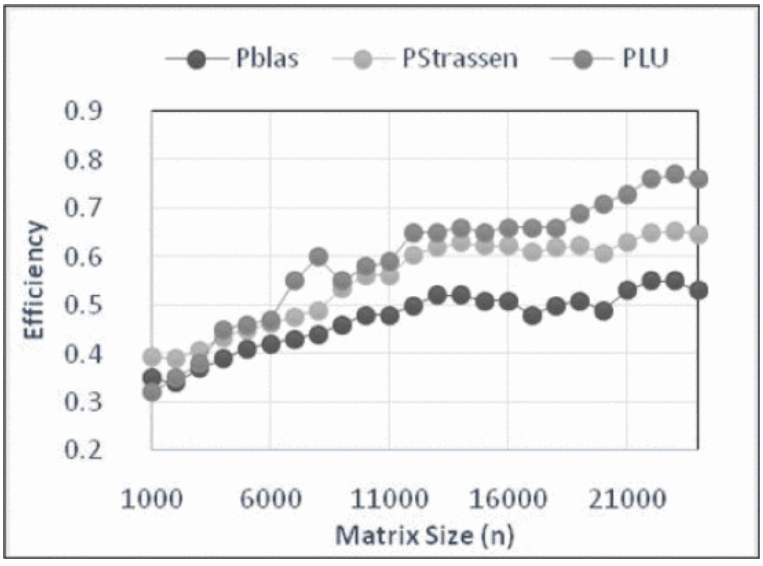


Figure 9.
Efficiency of pblas, pstrassen and PLU-DMI for p=16 and different values of n

Table V. Execution time and ratio for p=16 and different values of N

n	blas	Pblas	PStrassen	PLU-DMI	ρ_1	ρ_2
1 000	1.28	0.23	0.21	0.25	0.97	0.91
4000	94.79	15.19	13.78	13.16	1.11	1.15
8000	730.43	103.75	93.17	76.09	1.11	1.36
12000	2241.83	280.23	229.70	215.56	1.21	1.3
16000	6361.82	779.64	641.31	602.45	1.22	1.29
20000	7108.95	906.75	728.38	625.79	1.19	1.39
24000	18213.74	2147.85	1751.32	1497.84	1.22	1.43

The analysis of the results leads to the following remarks.

- For fixed p, the efficiency increases with n for the 3 different algorithms
- For large matrix sizes, recursive algorithms become more efficient than the Pblas algorithm.
- The implementation of the LU factorization is slightly better than the Strassen implementation.
- Parallel LU matrix inversion for large matrices (size larger than 5000) leads to at least 40% efficiency.
- For fixed n, the efficiency decreases with p for the 3 different algorithms. This fact may be due to parallelism and task management cost. In addition, we can notice that this decreasing becomes negligible for large n.

Finally, we can say that we've shown that our recursive algorithm for DMI may easily be adapted to multiprocessor environments and delivers efficient and scalable performance.

SECTION VI.
Summary and Future Work

We have presented an implementation of a matrix inversion algorithm based on the recursive LU factorization and Strassen method. The theoretical study was validated by a series of experiments achieved on multicore shared memory platforms. We observed that our implementations outperform one of the best known implementations namely Pblas. As a future direction, we consider extending our methodology to larger shared/distributed memory platforms involving a high number of homogenous processors and achieve an experimental study on heterogeneous multicore CPU/GPU systems.

Authors
Figures
References

Citations

Keywords

Metrics

More Like This

Some thoughts on “Algorithm Design and Analysis” teaching reform
2010 The 2nd International Conference on Industrial Mechatronics and Automation
Published: 2010

Differential GPS reference station algorithm-design and analysis
IEEE Transactions on Control Systems Technology
Published: 2000

Show More