HPCLatAm 2013, pp. 167-178 (full paper)
Session: CPU and Multicore Architectures and Applications
C. Garcia Garino and M. Printista (Eds.)
Mendoza, Argentina, July 29-30, 2013

Strategies to optimize the LU factorization algorithm on multicore computers

Janet Soler[1], Javier Ortiz[1], and Gustavo Wolfmann[1]

Laboratorio de Computación - Facultad Cs. Exactas Físicas y Naturales
Universidad Nacional de Córdoba
Av. Vélez Sársfield 1611 - Córdoba - Argentina
{janetsoler,ortizjavier,gwolfmann}@gmail.com

Abstract. The number of cores in multicore computers has an irreversible tendency to increase. Also, computers with multiple sockets to insert multicore chips are based on a complex hardware design and are becoming more common. To parallelize the algorithms that run on this type of computers in order to obtain a higher performance rate, is a goal that can only be achieved by taking into account hardware architecture. As hardware evolves, so must software. This leads to old parallelization strategies quickly become obsolete. This paper presents a series of alternatives for parallelization the LU factorization algorithm and its results intended to running on a multicore system. Simple strategies lead to poor results. This study presents complex strategies that merge double levels of parallelism with asynchronous scheduling whose results reach up to the State-of-the-art in the field and even go further.

1 Introduction

In linear algebra one of the most popular algorithms is the LU matrix factorization algorithm, which is used to solve linear equation systems. LU is also used in the LINPACK benchmark to measure the computation power of a computer system. The algorithm factorizes a square Matrix M in the form $M = PLU$, where L is a unit lower triangular matrix, U is an upper triangular matrix and P is a permutation matrix.

Most LAPACK library implementations compute LU by dividing the matrix in column blocks and then following a series of steps to complete the processing [1]. Basically, the algorithm performs *panel factorization* and *submatrix update* operations. The xGETRF function is used for panel factorization; for update, the xLASWP, xSTRSM and xGEMM functions are used, where $x = \{S|D\}$ according single or double precision functions are used. Details can be seen in [2, 3]. xGEMM is the function that performs matrix multiplication and is the predominant one in the algorithm. This causes LU to have $O(n^3)$ order.

The algorithm parallelization follows the outline of LAPACK by dividing data in column blocks and using the functions cited before, and assigning tasks to different processors to improve performance. Data dependency between tasks imposes limits to parallelism, and is the major obstacle to overcome.

Raw swapping constitutes an important data dependency imposed by numerical accuracy. The swap pattern is defined after the panel factorization, and must be applied to the rest of the matrix, before any update is performed. Despite the fact that swapping involves only data interchange, it generates a strong limitation for parallelization, because the remaining submatrix must be swapped.

On the other hand, the latest advances in microprocessors have been the development of chips with multiple cores; therefore, the existence of computers with a single core is becoming less frequent. This fact changes the programming paradigm by imposing the development of parallel programs in order to obtain the maximum possible performance of the new processors. In multicore architecture, all processors share the same memory space of the computer and for this reason this architecture is called "shared memory model".

Currently, the most common technique for programming parallel algorithms in a shared-memory computer is OpenMP [4], available in almost avery current FORTRAN or C compiler. OpenMP simplifies launching and managing of parallel threads, synchronization and data sharing.

The cache memory is an important factor in getting good performance in parallel programming. Unlike the main memory, the cache memory is shared only by a subset of cores. This generates problems like cache misses and cache coherency. These problems affect performance when the selection of a set of processors to run parallel task is not done properly, since the loss of spatial locality has a negative impact on the result.

Parallel execution of some linear algebra library routines on multicore systems presents scalability problems. The crucial point that motivates this research is the decreasing rate of speedup values obtained when increasing the number of cores. As an example, table 1 shows DGETRF function execution times in a 32-core system using ACML on four AMD 6128 dies of eight cores each and 48 GBytes RAM, running with different numbers of cores. Speedup increases until a given number of used cores are used, which depends on the matrix size. Above that limit, the speedup begins to decrease reaching an extreme point when the use of all available cores yields not positive speedup.

Matrix Range	12000			18000			24000		
Cores	Time	SpeedUp	Gflops	Time	SpeedUp	Gflops	Time	SpeedUp	Gflops
1	160,94	1,00	7,16	536,74	1,00	7,24	1262,76	1,00	7,30
2	88,07	1,83	13,08	406,73	1,32	9,56	781,53	1,62	11,79
4	109,01	1,48	10,57	275,49	1,95	14,11	570,56	2,21	16,15
8	59,29	2,71	19,43	223,60	2,40	17,39	505,13	2,50	18,24
16	120,94	1,33	9,53	207,87	2,58	18,70	357,90	3,53	25,75
32	497,61	0,32	2,31	760,11	0,71	5,11	821,61	1,54	11,22

Table 1: DGTREF function performance, ACML library, 32 cores AMD 6128 processors, double precision, time in seconds.

As for the low scalability problems cited before, this study has the purpose of researching two topics which are supposed to be the cause of the scalability problems: cache faults in multicore systems and the synchronization points of the parallel algorithm. The hypothesis is that by executing an adequate data division and managing task synchronization, speedup can be improved.

1.1 Previous research

In the case of a multicore system with multiple processor sockets, it is obvious that cache problems will arise when performing parallel processing of a common data block using more cores (threads) than those physically existent in a single die. The number of cores contained in a single die sets the theoretical limit for attaining a reasonable speedup in a single task. The proposed approach is to run different tasks with different data sets on each socket. This solution follows the Multiple Instruction, Multiple Data (MIMD) parallel programming style. In this case, task management becomes more complex because of data dependencies that should be taken into account.

Tiled algorithms emerge as a solution to the problem of load balancing for dense linear algebra algorithms on multicore processors [5]. This kind of algorithms have evolved from column block or row block-based algorithms to "tiles" of data, i.e. small square blocks. Tiled algorithms present, as many other LAPACK algorithms, two fundamentals steps: panel factorizations and updates of trailing submatrix.

PLASMA is a project developed by the University of Tennessee [6, 7] that aims at optimizing dense linear algebra functions running on multicore architectures, is based on two main concepts, termed as "tile blocks" and "dynamic scheduling". The project is oriented to improving CPU usage, and performs parallelism based on preexisting BLAS / LAPACK libraries.

By using "tile blocks", PLASMA evolves from previous column or row block division to small square blocks which are more efficiently managed by cache memory and define a fine-grain granularity of tasks, determining a large number of tasks to improve parallelism. Large numbers of tasks must be managed by a scheduler that launches tasks "dynamically" -as opposed to fork-join scheduling- in order to keep many cores processing in parallel.

Two points that are not clear in the PLASMA implementation of LU. First, small square data blocks are not in line with the requirements of LU algorithm to swap rows for numerical accuracy [8]. Second, the swap pattern can only be correctly determined by considering the whole column; thus, tile data blocks would leave parts of the column without consideration [9].

A scheduling can be considered "dynamic" when the time to launch a task is defined during runtime according to the evolution of execution. The scheduler of PLASMA is based on a directed acyclic graph (DAG) of data dependency which defines an "out of order" task execution flow. Nevertheless, the selection

criteria to launch tasks has not been explained, particulary when many tasks are available to run and they are out of the "critical path" of the DAG [8].

This paper presents a series of parallel algorithms that evolve from fork-join algorithms to double granularity ones, with an asynchronous task scheduler. This configuration clearly explains the performance improvement, that reach State-of-the-art PLASMA performance levels, and points to future research aimed at further performance improvements.

2 LU Factorization Optimizations

With the aim of obtaining better performance in the LU algorithm on a multicore computer, this research is focused on how to divide data and how to schedule the tasks determined by the algorithm. LU was chosen because of its strong data dependency. Each task is one of the four BLAS / LAPACK functions named before, and it is executed by using the specific function implemented by the linear algebra library. The primary strategy used to divide data is tile division.

Since the starting point of this research is the loss of scalability associated with the use of a higher number of processors, the set of processors is divided in two levels: the first level determines a set of groups that will execute tasks in parallel, and the second determines the number of cores to be used in parallel to execute each task. Since all cores must to be used, there are few possible group/core combinations in a 32 cores computer: 1 x 32, 2 x 16, 4 x 8 and so on. A key point to determine in the research is the optimal number of cores to be used for the execution of each task. Internally, each task is executed in parallel using the parallel version of the library. The parallel implementation of each function is used as provided by the library and will not be analyzed in this research.

The computer system used in this work, has 4 AMD Opteron 6128 microchips with 8 cores each, 64 GB of main memory. The size of the L3 cache of these processors is 12MB but it is split in two blocks of 6MB which are shared among 4 cores. In order to reduce communication costs between processors, cores that share the same cache memory are made to work together. The Blas / Lapack implementation used was ACML 5.1, also from AMD [10].

To take advantage of this hardware cofiguration, computation is distributed among 8 teams of 4 threads each. These teams are created using OpenMP parallel pragmas and C language, which defines a first level of parallelism. The second level is given by the use of the parallel version of ACML routines.

It has already been mentioned that in order to avoid cache misses, threads should be mapped to processors. This is done by using affinity masks; thus, the execution has one-to-one correspondence. Additionally, to allow multiple thread divisions, nested parallelism capability should be set. At the moment of starting this work, these two functionalities for AMD processors were only available using the gcc compiler, version 4.7.2 [11].

One of the consequences of Amdahl's law is that the speedup atained in a system by the parallelization of one of its components is directly proportional to

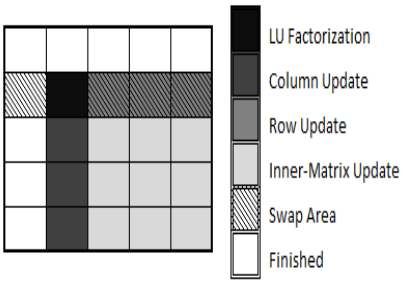
the fraction of time this component is used. Therefore, the optimization efforts should be focused on those functions that take more computing time. In blocked LU factorization, the dominant function is matrix multiplication for matrix update at each iteration.

In order to get the best performance for each operation, is necessary to find the optimal block size for minimizing L3 cache misses. Since the computer used has has blocks of 6MB of this kind of memory, and the function uses 3 different matrices, the range of each one should be less than 480 double precision numbers. This decision is also limited by the fact that small blocks need more iterations to complete the algorithm, which implies that block size should be the biggest that fit into L3, namely 480 elements in this case. Table 2 confirms this: a maximum of 103.84 Gflops is reached in matrix multiplication performed in parallel by 8 teams of 4 cores each, using ACML's DGEMM function. The theoretical performance peak of the machine used is 256 Gflops.

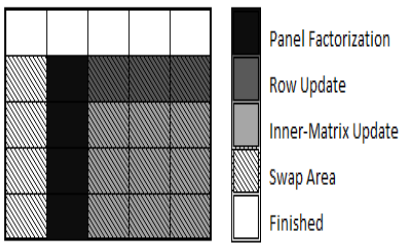
Block range	300	400	480	500
Time in secs.	687.30	281.17	266.25	350.32
Gflops.	40.23	98.33	103.84	78.91

Table 2: Matrix multiplication times, ACML DGEMM function, range 24000, 8 tasks, 4 cores each.

In a tiled data division, the data unit is a block of the matrix and the tasks take this units as their input. Figure 1 shows two different approaches to computing LU factorization of a matrix. The first one uses square blocks and presents more parallelization opportunities whereas the second one uses columns and



a)Tile block swap area



b)Column block swap area

Fig. 1: Tiled data and tasks division. Differences in the swap area according to data division are noticeable.

yields more accurate results. We explain the block algorithm in the first place because optimizations are more visible; next we will cover the column algorithm.

The block algorithm consists of a serie of iterations that carry out the same steps for every row. First, the diagonal block will be LU-factorized using the xGETR2 function, i.e. panel factorization. Next, those blocks that are in the same row at the right of the diagonal block and the blocks in the same column under the diagonal will be updated using the xTRSM function. Finally, matrix multiplications should be performed on the blocks in the remaining inferior right part of the matrix. The last two functions perform the matrix update.

In this algorithm, every operation of a given step can be parallelized. The scheduler is static and synchronous, compounding a fork-join style parallel algorithm. Figure 2 shows an execution timeline for this case using static scheduling with OpenMP pragmas. This graphic shows that there is much time during which processors are idle because of synchronization barriers at the end of every steps, which are represented by white areas. On the right side of the figure, a data column shows the rate of activity of the processors. It is evident the there is a poor load balance and that the processors have to wait a long time for the factorization of a diagonal block.

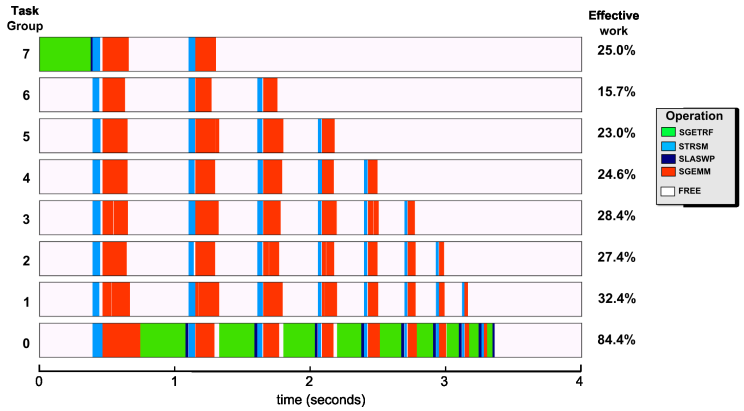


Fig. 2: Static and synchronous scheduling timeline. Range: 4800. BlockSize: 480.

2.1 Dynamic Scheduling

Tiled algorithm is aimed at avoiding delays caused by synchronous tasks execution by increasing the number of tasks availables for running in parallel and by running them "out-of-order". For example, LU factorization of a main diagonal block can be computed as soon as the matrix multiplication from the previous iteration is completed. This allows an execution of the algorithm in which tasks can be performed once their dependencies are satisfied.

Since the input of this algorithm is the tile data block, numerical accuracy is disregarded because the swap operation is performed only among the rows within the tile height. Nevertheless, it is presented in order to compare the results, since the number of operations is the same as if the whole column where taken into account to define the swap, which shows the difference between both algorithms.

To implement this algorithm, a task table is used, which contains information about the function, state, blocks involved and dependencies of each task. This table is inspected by 8 threads that look for a task that is ready for execution, guaranteeing that every task will be executed as soon as its dependencies have been computed and there is an available processor.

The task dependency table has a specific order that defines the priority when two tasks are enabled to be executed. When the sequence of the tasks prioritization follows the static parallelization execution flow, the results are very similar for both schedulers. This is due to the fact that operations of the same iteration have the same time of execution and therefore, once all matrix multiplication of an iteration have been computed, there is only one LU factorization whose dependencies have been completed, and a "bottleneck effect" occurs. Therefore, task prioritization is very important to obtain performance improvements.

When the execution of tasks prioritize the computation of LU factorization for diagonal blocks and their dependencies, higher speed up values are obtained because processors are constantly solving tasks whose dependencies have been completed and do not have to wait for other synchronization barriers. These results are shown in table 3.

	DGETRF 1 Thread			**Static Sched**			**Dynamic Sched**		
Matrix Range	Time	Speed Up	Gflops	Time	Speed Up	Gflops	Time	Speed Up	Gflops
12000	160.94	1.00	7.16	36.47	0.88	31.59	15.13	2.11	76.13
18000	536.74	1.00	7.24	115.58	1.39	33.64	6.92	3.43	82.86
24000	1262.76	1.00	7.30	302.99	6.66	30.42	120.00	16.81	76.80

Table 3: Static and Dynamic performance for block computation of LU factorization. Matrix Range: 24000. Double precision. Time in seconds.

2.2 Column based LU Factorization

Although the tiled version of the algorithm produces good speedup through dynamic scheduling, the output lacks numerical accuracy. After panel factorization, matrix rows are pivoted in order to avoid making divisions by small numbers. In blocked execution, the largest number is searched by using only the rows of the tile. A more precise algorithm use the entire column, thus guaranteeing that the maximum number will be used when swapping rows.

The tasks in this algorithm is shown in fig. 1b). There are two major changes: first, columns are not updated through the xTRSM function anymore and second, xGETRF and xLASWP are applied over the entire column. Using whole columns to compute row swaps implies more dependencies between tasks and therefore less parallelization to exploit. Figure 3 shows a tasks dependencies graph.

When using dynamic scheduling, the algorithm yielded poor speed up values as compared to those presented in previous section. The decrease in performance was first attributed to the existence of more synchronization points; but upon inspection of a particular execution, it was observed that there is a factor that has a higher impact: growing cache misses.

The block size selected before had been dependent on the matrix multiplication operation. In this case, LU factorizations take so much time because columns do not fit in cache L3 anymore. For instance, in a matrix with a range of 12000 and using blocks of 480 elements, the first factorization would be computed using 12000 x 480 double precision elements, which occupy approximately 44MB, several times bigger than the available L3 cache memory.

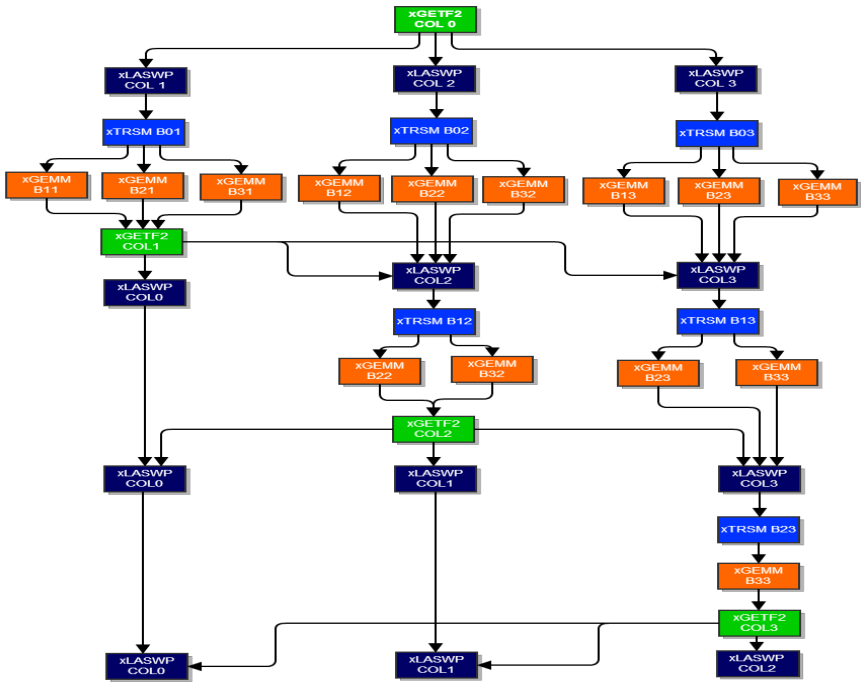


Fig. 3: Tasks dependencies graph of LU algorithm, 3 tile division, columns swaps

2.3 Double Granularity Algorithm

To solve the problem of high cache misses in the whole column based algorithm, a different implementation of parallelization is proposed, which has two levels of granularity in the computation of LU factorization. All operations, except for the panel factorization task, use the same tile size as before. For the panel factorization, another granularity is applied when the routine xGETF2 is called. The LU factorization is calculated appliyng the column-based data partition with a smaller column width, so as to allow the whole column to fit in the L3 cache, and thus to avoid misses. This algorithm is called "Double Granularity" because there are two block sizes: tile blocks, used for matrix updates, and narrow columns, used for panel factorizations.

The use of "Double Granularity" can be seen graphically in fig. 4. It is assumed that the first row and column matrix blocks have already been computed. The charts in the first row of the figure show tasks that have been executed with tile block granularity. The second row of charts shows the steps followed by the algorithm when executing a panel factorization of chart 1, using the second level of granularity. In the example, the second column is subdivided in narrower columns, which determines a sequence of subtaks needed to complete the panel factorization, following algorithm dependency.

When appliyng these improvements, the execution behaves as shown in the timeline of fig. 5, where it can be seen that the decrease in execution time at each LU factorization enables more parallelization in the computing process. Results

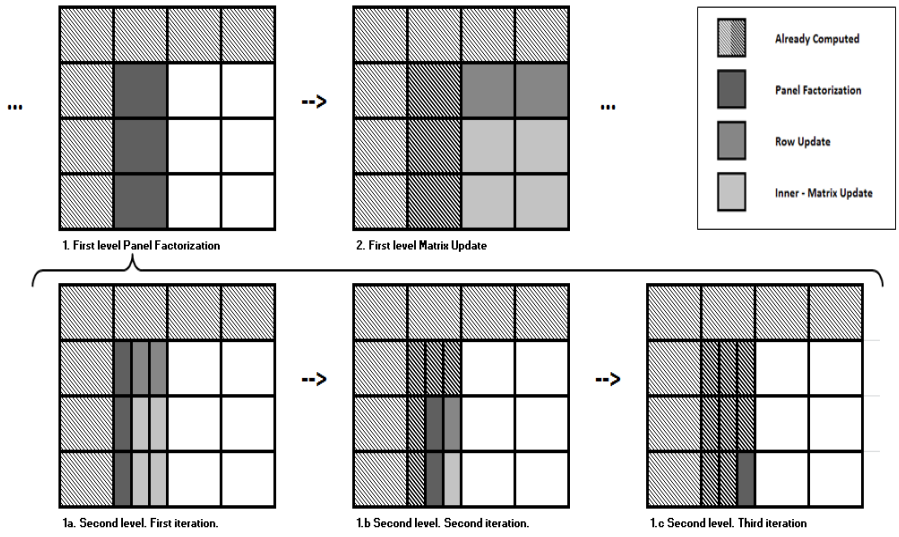


Fig. 4: Double Granularity graphic. First row and column matrix are already computed. Charts 1 and 2 show operations at tile block level. Charts 1a, 1b and 1c show the subtasks needed to complete panel factorization of chart 1

of the application of this algorithm to matrices of different size are presented in table 4. It also shows PLASMA running time, using the same ACML library version. PLASMA is presented in both static and dynamic versions in order to make a complete comparison. It is highlighted that the performance attained by using a double granularity algorithm is similar to or better than the best of PLASMA, which is the point this research aims at demonstrating.

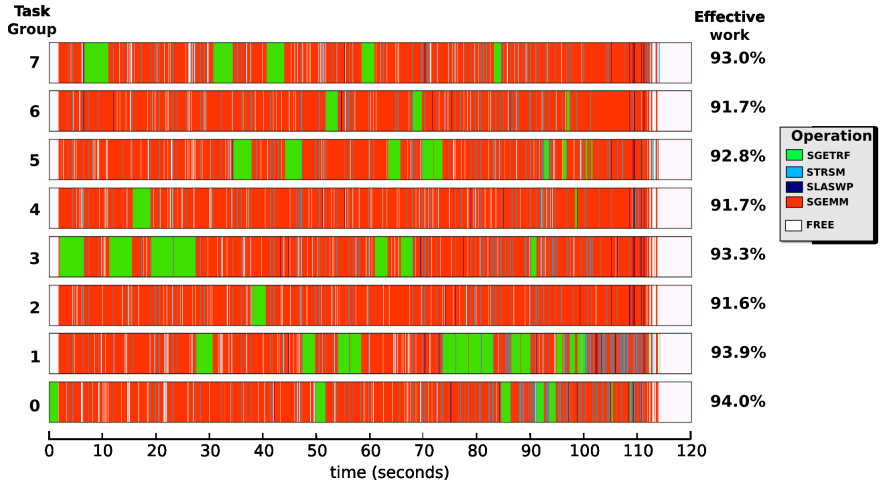


Fig. 5: Timeline double granularity, 24000 range, 480 block range, 8 groups, 4 cores each. Effective work is the time dedicated to computing the algorithm.

		PLASMA static			PLASMA dynamic			Double Granularity		
Matrix Range	**Cores**	Time	Speed Up	Gflops	Time	Speed Up	Gflops	Time	Speed Up	Gflops
12000	1	184.1	1.00	6.26	219.8	1.00	5.24	167.8	1.00	6.86
12000	8	47.2	3.90	24.42	50.0	4.39	23.03	38.1	4.40	30.19
12000	16	21.9	8.42	52.67	21.9	10.04	52.62	21.1	7.95	54.54
12000	32	20.6	8.93	55.87	20.6	10.68	55.98	15.4	10.90	74.85
18000	1	726.5	1.00	5.35	717.7	1.00	5.42	780.1	1.00	4.98
18000	8	186.9	3.89	20.81	187.8	3.82	20.70	128.4	6.08	30.28
18000	16	67.4	10.79	57.72	71.1	10.10	54.69	77.2	10.10	50.33
18000	32	65.7	11.07	59.22	66.6	10.78	58.41	48.6	16.04	79.96
24000	1	1802.2	1.00	5.11	1851.9	1.00	4.98	1860.4	1.00	4.95
24000	8	487.3	3.70	18.91	481.1	3.85	19.16	272.5	6.83	33.82
24000	16	151.6	11.89	60.79	155.2	11.93	59.39	187.1	9.94	49.25
24000	32	114.8	15.69	80.25	127.5	14.52	72.27	116.1	16.02	79.37

Table 4: PLASMA and double granularity times and performance, 32 cores AMD 6128 processors, double precision, time in seconds.

3 Conclusions and Future Work

In this work, we show the significant impact of good or bad uses of the cache memory. This can be observed in two different situations. First, adequate assignment of tasks to cores that are physically close decreases internal communication costs. Second, choosing the correct data set size minimizes cache misses. The compiler plays a major role in providing affinity and nested parallelism, two functionalities that make these performance improvements feasible.

Nested parallelism is proposed as a way of efficiently using hardware architecture. Multiple cores over multiple dies imply that the software should use a double level of task division in order to get the maximum possible performance.

After a certain number of attempts, two important marks have been reached in performance and scalability. Almost 80 Gflops in a machine with a theoretical peak of 256, with LU algorithms, is not easy to improve with the tools used. Speedups values of 10 and 16 using 16 and 32 cores, respectively, show that algorithm is effective.

It is also remarkable that rows swap produces an impact in overall performance. In this algorithm, numerical accuracy imposes not only more data dependency but also increases difficulties in data division, thus bringing about the need for double granularity, as a way to solve cache memory problems.

Optimized parallel software design must be carried out in accordance with the hardware architecture on which the software will be used. To reach state-of-the-art performance, or even to go further is not impossible provided hardware details are taken into account and data division is properly defined.

A potential problem detected in this research is the complexity posed by the number of tasks and the dependencies between them, which the parallel algorithm must manage. As the number of blocks into which data is divided changes, so do the number and dependencies of tasks, thus making the algorithm hard to optimize. To overcome this problem, research in modeling algorithms with colored Petri Nets will be done in the future.

Finally, finding high performance dedicated computers that are configured with an heterogeneous set of multicore and multigpu devices, is becoming easier. GPU units are faster than CPU, but their memory capacity is lower. Also, a scheduler that is able to combine efficiently tasks launched on CPUs with tasks launched on GPUs is an object of study for future research.

References

1. Anderson, E., Bai, Z., Bischof, C., Blackford, L.S., Demmel, J., Dongarra, J.J., Du Croz, J., Hammarling, S., Greenbaum, A., McKenney, A., Sorensen, D.: LAPACK Users' guide (third ed.). Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (1999)
2. Golub, G.H., Van Loan, C.F.: Matrix computations (3rd ed.). Johns Hopkins University Press, Baltimore, MD, USA (1996)
3. Grama, A., Karypis, G., Gupta, A., Kumar, V.: Introduction to Parallel Computing: Design and Analysis of Algorithms. Addison-Wesley (2003)

4. the OpenMP Architecture Review Board: Openmp. the openmp api specification for parallel programming. http://openmp.org/
5. Buttari, A., Langou, J., Kurzak, J., Dongarra, J.J.: A class of parallel tiled linear algebra algorithms for multicore architectures. Technical Report 191, LAPACK Working Note (September 2007)
6. the University of Tennessee: The plasma project. parallel linear algebra for scalable multi-core architectures. http://icl.cs.utk.edu/plasma/
7. Agullo, E., Demmel, J., Dongarra, J., Hadri, B., Kurzak, J., Langou, J., Ltaief, H., Luszczek, P., Tomov, S.: Numerical linear algebra on emerging architectures: The plasma and magma projects. Journal of Physics: Conference Series **Vol. 180** (2009)
8. Dongarra, J., Faverge, M., Ltaief, H., Luszczek, P.: Achieving numerical accuracy and high performance using recursive tile lu factorization. Technical Report 259, LAPACK Working Note (December 2011)
9. Kurzak, J., Luszczek, P., Faverge, M., Dongarra, J.: Lu factorization with partial pivoting for a multi-cpu, multi-gpu shared memory system. Technical Report 266, LAPACK Working Note (April 2012)
10. AMD Advanced Micro Devices, I.: Core math library (acml). http://developer.amd.com/tools-and-sdks/cpu-development/amd-core-math-library-acml/
11. Free Software Fundation, I.: Gcc, the gnu compiler collection. http://gcc.gnu.org/