# Design and Performance Issues of Cholesky and LU Solvers using UPCBLAS

Jorge González-Domínguez*, Osni A. Marques†, María J. Martín*, Guillermo L. Taboada*, Juan Touriño*

* Computer Architecture Group, Department of Electronics and Systems, University of A Coruña, Spain
Email: {*jgonzalezd, mariam, taboada, juan*}*@udc.es*
† Computational Research Division, Lawrence Berkeley National Laboratory, CA, USA
Email: *OAMarques@lbl.gov*

***Abstract*—Partitioned Global Address Space (PGAS) languages offer programmers a shared memory view that increases their productivity and allow locality exploitation to obtain good performance on current large-scale distributed memory systems. UPCBLAS is a parallel numerical library for dense matrix computations using the PGAS Unified Parallel C (UPC) language. The interface of this library exploits the characteristics of the PGAS memory model and thus it is easier to use than MPI-based libraries. This paper addresses the implementation of solvers of systems of equations through Cholesky and LU factorizations in UPC using UPCBLAS. The developed codes are experimentally evaluated and compared to the MPI versions using ScaLAPACK. Parallel solvers of equations are present in many parallel numerical applications and they have been traditionally developed in MPI. This work shows that UPCBLAS can be considered as a good alternative to the MPI-based libraries for increasing the productivity of numerical application developers.**

***Index Terms*—PGAS, UPC, UPCBLAS, ScaLAPACK, Matrix Computations, Cholesky, LU**

## I. Introduction

The popularity of Partitioned Global Address Space (PGAS) languages has increased during the last years thanks to their good trade-off between programmability, because of offering a global memory shared by all threads, and performance, through an efficient exploitation of data locality. These languages can take advantage of shared memory support when it exists, and they rely on the partitioned nature of the address space to be efficiently used on distributed memory machines. Thus, they are specially attractive on hierarchical architectures such as multicore clusters. Some examples of PGAS languages are Unified Parallel C (UPC) [1], Co-Array Fortran [2] or Titanium [3].

UPC is an extension of ANSI C for parallel computing that follows the PGAS paradigm. There are two main advantages of using UPC instead of the conventional message-passing programming model, such as MPI. First, the shared address space simplifies programming as programmers can use distributed arrays which can be directly accessed by all threads. Second, the global memory allows the efficient use of one-sided communications. Most of the recent works related to UPC are focused on demonstrating that UPC can outperform MPI thanks to one-sided communications [4]–[7].

The importance of designing high performance algorithms for solving linear systems is motivated by many scientific and engineering applications. The whole process of solving a non-triangular system of equations has not been studied in UPC. However, some previous UPC works addressed some parts of this issue by taking advantage of one-sided communications but without exploiting the mechanisms provided by the language to facilitate parallel programming. For instance, different data distributions are studied in [8] for triangular solvers by distributing the matrix data among the private memories of all threads. Therefore, the properties of the global shared memory are not exploited in this work. Husband and Yelick [9] undertook the parallelization of the LU factorization using own-designed structures to store the matrix data in shared memory. Thus, in order to use this implementation, UPC programmers would be forced to deal with these ad-hoc structures that represent the distributed data. Therefore, as in the MPI-based libraries, users should be aware of the appropriate local indexes to use in each process, which increases the complexity of developing parallel codes [10] and does not take advantage of the ease of programming offered by shared arrays in UPC.

This paper presents the implementation of solvers of equations in UPC through Cholesky and LU factorizations. These implementations are based on UPCBLAS [11], a parallel numerical library with a relevant subset of BLAS routines [12], [13] implemented for UPC. This library increases the productivity of programmers by using shared arrays to represent distributed vectors and matrices and thus exploiting the programmability of the PGAS paradigm. UPCBLAS also takes advantage of one-sided communications to perform the parallel routines and even automatically optimizes them according to the hardware characteristics of the underlying machine. Thus, UPCBLAS and, consequently, the solvers presented in this work, preserve the two main advantages of the PGAS languages: one-sided communications and usability.

The rest of the paper is organized as follows. Section II provides an overview of UPCBLAS, as background for the following sections. Sections III and IV describe the algorithms to solve systems of equations using Cholesky and LU factorizations, respectively. Section V presents the analysis of the experimental results obtained on an IBM supercomputer (Carver), as well as their comparison with the ScaLAPACK library [14]. Finally, conclusions are discussed in Section VI.

## II. Overview of the UPCBLAS Library

As was previously mentioned, UPCBLAS is a parallel numerical library built on top of standard UPC which provides a version of the BLAS routines that follows the PGAS paradigm. All PGAS languages, and thus UPC, expose a global shared address space to the user which is logically divided among threads, so each thread is associated or presents affinity to a part of the shared memory. Moreover, UPC also provides a private memory space per thread for local computations. Therefore, each thread has access to both its private memory and to the whole global space, even the parts that do not present affinity to it. Typically the accesses to remote data will be much more expensive than the accesses to local data (i.e. accesses to private memory and to shared memory with affinity to the thread).

Shared arrays are employed to implicitly distribute data among all threads, as shared arrays are spread across the threads. The syntax to declare a shared array `A` is: `shared [BLOCK_FACTOR] type A[N]`, being `BLOCK_FACTOR` the number of consecutive elements with affinity to the same thread, `type` the datatype, and `N` the array size. It means that the first `BLOCK_FACTOR` elements are associated to thread 0, the next `BLOCK_FACTOR` ones to thread 1, and so on. Thus, the element `i` in the array has affinity to thread $\lfloor \frac{i}{BLOCK\_FACTOR} \rfloor mod(THREADS)$, being $THREADS$ the total number of threads in the UPC execution.

With regard to the design of a parallel numerical library, the main difference between UPC and a message-passing paradigm (such as MPI) is that the latter does not provide any structure in the language to deal with vectors and matrices distributed among the processes. Therefore, developers of message-passing numerical libraries have to create additional structures to represent distributed vectors and matrices. Both the new structures and the distribution of the matrices among the processes are concepts that pose an important challenge for most of the users of parallel numerical libraries (researchers and engineers from different areas), as can be seen in the results of the survey presented in [10]. In contrast, UPCBLAS makes use of shared arrays, significantly improving the ease of use of the library and thus the productivity of numerical applications developers. However, the use of UPC shared arrays to distribute vectors and matrices limits the possible types of distribution to 1D. Experimental results in [11] demonstrated that, for some BLAS routines, the 2D distribution obtains better performance than 1D. As will be seen in the experimental evaluation in Section V, this limitation also influences the performance of the solvers of equations. Nevertheless, the implementations based on UPCBLAS offer a good trade-off between programmability and performance.

The BLAS3 matrix-matrix product will be used in this section to summarize the design and implementation of UPCBLAS, as most of the algorithms presented in Sections III and IV are based on this routine. In order to facilitate the adoption of UPCBLAS among PGAS programmers the syntax of these functions is similar to the standard collectives library [1]. For instance, the syntax of the matrix-matrix product in double precision ($C = \alpha * A * B + \beta * C$) is:

```
upc_blas_dgemm(UPCBLAS_DIMMDIST dimmDist,
int block_size, int sec_block_size,
UPCBLAS_TRANSPOSE transposeA,
UPCBLAS_TRANSPOSE transposeB, int m, int
n, int k, double alpha, shared void *A,
int lda, shared void *B, int ldb, double
beta, shared void *C, int ldc);
```

being $A$, $B$ and $C$ `mxk`, `kxn` and `mxn` matrices, respectively; and `alpha` and `beta` the scale factors for $A$ and $C$, respectively. Most of the parameters have the same name and meaning than in sequential BLAS [12]. The only differences are that the pointers that represent the input and output matrices point to shared memory and that there are some additional parameters (the first three ones) to specify the distribution of the matrices. `dimmDist` is an enumerate value to specify if the output matrix is distributed by rows or by columns. If $C$ is distributed by rows then `block_size` indicates the number of consecutive rows with affinity to the same thread. In this case the algorithm implemented by the routine follows the structure shown in Figure 1. In order to perform its sequential partial matrix-matrix product each thread only needs to access the same rows of $A$ than those of $C$ with affinity to that thread, but all the elements of $B$. Therefore, as matrices $A$ and $C$ must have the same distribution, `block_size` is also related to $A$. `sec_block_size` specifies the number of consecutive elements of $B$ with affinity to the same thread.

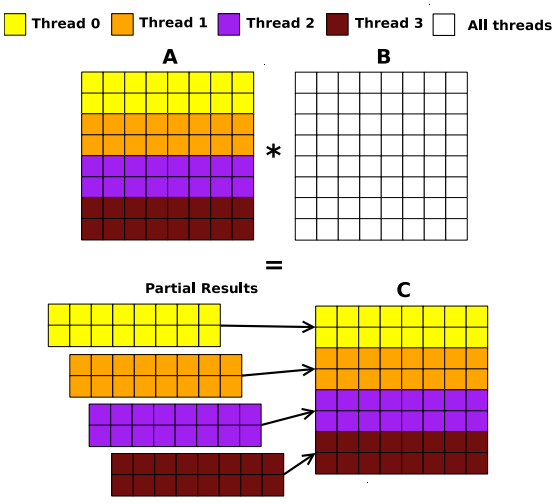


Fig. 1: UCBLAS matrix-matrix product (*gemm*) using a row distribution for matrix `C`

In comparison, Figure 2 describes the behavior of the UPCBLAS matrix-matrix product when the output matrix is distributed by columns. In this case, each thread needs to access the whole matrix $A$ but only the same columns of $B$ than those of $C$ with affinity to that thread. Thus, `block_size` defines the distribution of $C$ and $B$, and `sec_block_size`

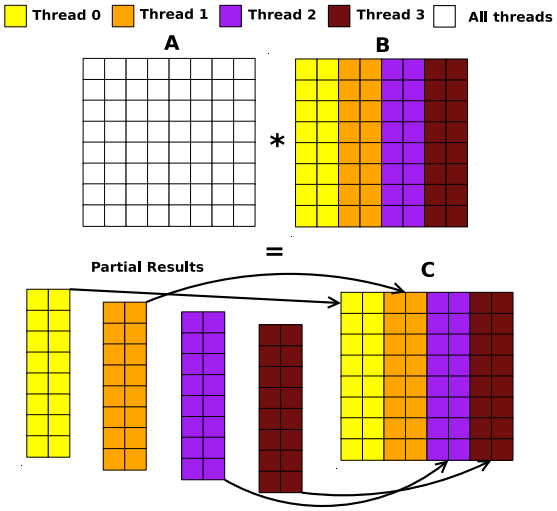


Fig. 2: UCBLAS matrix-matrix product (*gemm*) using a column distribution for matrix `C`

is related to $A$.

This example illustrates the ease of use of UPCBLAS as programmers only need to know how to work with the UPC shared arrays and their block factor in order to exploit the functionalities of the library. Shared arrays are implicitly partitioned across threads, thus the complex steps of declaring and distributing vectors and matrices required by the MPI-based libraries are avoided.

## III. Cholesky Solver

The Cholesky factorization is mainly used for the numerical solution of linear equations $A * X = B$ when $A$ is symmetric and positive definite. The system can be solved by first computing the Cholesky factorization $A = LL^T$ (being $L$ a lower triangular matrix with strictly positive diagonal entries), then solving $L*Y = B$ for $Y$, and finally solving $L^T * X = Y$ for $X$. A similar approach is applied if the system has the form $X * A = B$.

Two different algorithms by blocks have been studied to implement the Cholesky solver. They operate on submatrices instead of on individual matrix entries. They are very adequate for parallel numerical codes as they are based on BLAS3 routines, which obtain good scalability.

As explained in the previous section, UPCBLAS is limited to the distributions available for shared arrays in UPC which, up to now, are 1D. Thus, users can choose between the block-cyclic distribution by rows or by columns. For the sake of simplicity, all the algorithms in this paper will only be explained for a block-cyclic distribution by rows, such as the one shown in Figure 3, where $A_{ij}$ are submatrices. The column distribution versions can be easily inferred.

As UPCBLAS is focused on increasing programmability, the syntax of all its functions is very similar to the sequential BLAS. Furthermore, UPCBLAS functions work with arrays as in the sequential numerical libraries instead of with ad-hoc data structures as in the MPI-based ones. Therefore, the

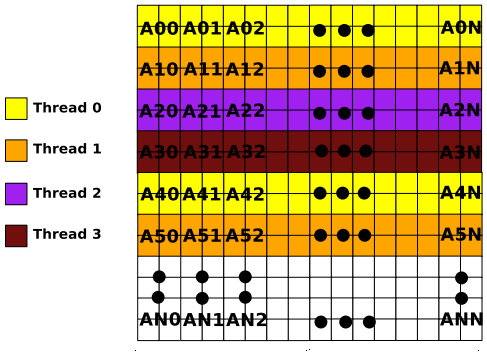


Fig. 3: Example of input matrix distributed by rows in a block-cyclic way

first approach for the Cholesky factorization using UPCBLAS is derived from the sequential algorithm available in the LAPACK library [15]. Figure 4 shows this approach, which is based on the matrix-matrix product of general matrices (`gemm` routine). The input matrix $A$ is distributed in $NB$ blocks of $BS$ rows. The input/output matrix $B$ ($X$ overwrites $B$) does not have to follow the same distribution. Taking into account the experimental results shown in [11], $B$ is distributed by columns if the system is $A * X = B$, and by rows if it is $X * A = B$, in order to obtain the best performance.

**for** *i=0;i<NB;i=i+1* **do**
  **if** *MYTHREAD has affinity to block i* **then**
    $A_{i,i} = A_{i,i} - A_{i,0..i-1} * A_{i,0..i-1}^T \rightarrow syrk$
    $Sequential\ Cholesky\ Factorization\ of\ A_{i,i}$
  **end**
  $A_{i+1..N,i} =$
    $A_{i+1..N,i} - A_{i+1..N,0..i-1} * A_{i,0..i-1}^T \rightarrow gemm$
  $Solve\ Z * A_{i,i}^T = A_{i+1..N,i} \rightarrow trsm$
  $A_{i+1..N,i} = Z$
**end**
$Solve\ Y * A^T = B \rightarrow trsm$
$Solve\ X * A = Y \rightarrow trsm$

Fig. 4: Algorithm based on parallel `gemm` for the Cholesky solver

The algorithm consists of a loop in which two operations per iteration are carried out in parallel using UPCBLAS: `trsm` (triangular solver) and `gemm` (matrix-matrix product). Besides, there is one thread in each iteration that must perform some additional computations only accessing data stored in its local memory: the `syrk` routine (product of a symmetric matrix by its transpose) and the sequential Cholesky factorization. The `syrk` routine is performed by calling a sequential BLAS routine. However, no routine could be used to perform the sequential Cholesky factorization of one block. Although LAPACK has a routine to perform this factorization, it only works if matrices have their elements ordered in a column-wise way, the common format in Fortran. As UPCBLAS follows the UPC and ANSI C format (row ordering of elements),

transposing the matrix in each block is the only way to use the LAPACK interface, which was obviously discarded due to its high overhead. An own row-wise C routine for the sequential Cholesky factorization was therefore implemented. This routine was parallelized for multicore architectures (shared memory) using OpenMP directives.

After the loop, the lower triangular part of $A$ stores the entries of $L$, which can be directly used as input of the UPCBLAS triangular solvers to solve the system of equations.

Two data dependencies arise when studying this algorithm:

- No thread can start the parallel `trsm` in iteration $i$ before the thread with affinity to the block $A_{i,i}$ has finished the Cholesky factorization of this block.
- No thread can start the parallel `gemm` in iteration $i$ before the thread with affinity to the block $A_{i-1,i-1}$ has finished its part of the parallel `trsm` of the previous iteration.

There are not dependencies between the parallel `gemm` and the sequential computations in each iteration. Thus, the proper mechanisms of synchronization have been used in order to parallelize all these computations.

The second approach developed for implementing the Cholesky solver using UPCBLAS is an adaptation of the algorithm used by ScaLAPACK [14] and is described in Figure 5. The UPCBLAS `syrk` and `trsm` routines are used for the parallelization. Note that the `syrk` routine was later included in UPCBLAS as a result of this work. The main advantage of this algorithm is that there are less sequential computations than in the previous one (the sequential `syrk` per iteration is avoided). These computations have not disappeared but they are included in the parallel `syrk`.

**for** *i=0;i<NB;i=i+1* **do**
  **if** *MYTHREAD has affinity to block i* **then**
    $Sequential\ Cholesky\ Factorization\ of\ A_{i,i}$
  **end**
  $Solve\ Z * A_{i,i}^T = A_{i+1..N,i} \rightarrow trsm$
  $A_{i+1..N,i} = Z$
  $A_{i+1..N,i+1..N} =$
    $A_{i+1..N,i+1..N} - A_{i+1..N,i} * A_{i+1..N,i}^T \rightarrow syrk$
**end**
$Solve\ Y * A^T = B \rightarrow trsm$
$Solve\ X * A = Y \rightarrow trsm$

Fig. 5: Algorithm based on parallel `syrk` for the Cholesky solver

Two dependencies appear in this algorithm too:

- No thread can start the parallel `trsm` in iteration $i$ before the thread with affinity to the block $A_{i,i}$ has finished the sequential Cholesky factorization of this block.
- No thread can start the parallel `syrk` in iteration $i$ before all threads have finished the parallel triangular solver in that iteration.

The main drawback of this algorithm is that dependencies are stronger and closer than in the first one. Therefore, the overhead due to synchronizations increases.

In both algorithms the selection of $NB$ and $BS$ is key to obtain good performance. The more blocks the matrix is divided in, the more computations can be simultaneously performed, but the more synchronizations are needed too.

## IV. LU Solver

If the input matrix A does not fulfill the requirements of the Cholesky factorization, the LU decomposition can be used instead. In this case matrix $A$ is decomposed in two matrices $L$ and $U$, lower and upper triangular, respectively. Thus, $A$ can be exchanged by $L * U$ and the system of equations can be solved in two steps. First, $L * Y = B$ is solved and then $U * X = Y$.

Figure 6 shows the basic block algorithm based on BLAS3 routines to perform the parallel LU solver. It is derived from the algorithms included in LAPACK and ScaLAPACK. In this case the loop computes the LU factorization, so $L$ and $U$ are stored in the lower and upper triangular parts of $A$, respectively. Then, the first final `trsm` performs the triangular solver with the lower part of the matrix and the second `trsm` uses the upper one.

This algorithm is mainly based on the UPCBLAS implementations of `trsm` and `gemm`. Besides, as for the Cholesky case, a C version of the LU factorization had to be developed, as the available libraries could not work with row-wise matrices. This LU implementation is also parallelized using OpenMP directives.

The only dependency among the computations performed by different threads is that no thread can start the parallel `trsm` in iteration $i$ before the thread with affinity to the block $A_{i,i}$ has finished the LU factorization of this block. The structure of this algorithm is quite similar to the one described in Figure 5. However, this algorithm should obtain better scalability as no thread needs any remote data of the output of the parallel `trsm` and thus the second dependency present in the algorithm of Figure 5 is avoided. Besides, the LU algorithm has been optimized by moving forward the sequential computations of the next iteration, performing them in parallel with the `gemm` routine and thus minimizing the impact of this overhead.

Depending on the characteristics of the input matrix $A$, the algorithm shown in Figure 6 could lead to inconsistencies because of dividing by zero within the sequential LU factorizations. Partial pivoting must be performed in order to avoid this issue and the system has the form $P * L * U * X = P * B$, where $P$ is a permutation matrix with exactly one entry equal to one in each row and column.

The algorithm for solving this system is shown in Figure 7. If the matrix is, as in the example, distributed by rows (see Figure 3), the pivoting is performed by columns so that all the swaps of columns can be parallelized. $P$ is a vector of size $N$ stored in shared memory and distributed among threads according to $BS$. It is used to store the information about the columns that must be swapped in each iteration. This information is computed before the LU factorization and it is

**for** *i=0;i<NB;i=i+1* **do**
  **if** *MYTHREAD has affinity to block i* **then**
    $Sequential\ LU\ Factorization\ of\ A_{i,i..N}$
  **end**
  $Solve\ Z * A_{i,i}^T = A_{i+1..N,i} \rightarrow trsm$
  $A_{i+1..N,i} = Z$
  $A_{i+1..N,i+1..N} =$
    $A_{i+1..N,i+1..N} - A_{i+1..N,i} * A_{i,i+1..N} \rightarrow gemm$
**end**
$Solve\ A * Y = B \rightarrow trsm$
$Solve\ A * X = Y \rightarrow trsm$

Fig. 6: Algorithm for the LU solver without pivoting

**for** *i=0;i<NB;i=i+1* **do**
  **if** *MYTHREAD has affinity to block i* **then**
    $P_i = Partial\ Pivoting\ of\ A_{i,i..N}$
    $Swap\ A_{i,i..N}\ according\ to\ P_i$
    $Sequential\ LU\ Factorization\ of\ A_{i,i..N}$
  **end**
  $Swap\ A_{0..N,i..N}\ according\ to\ P_i$
  $Solve\ Z * A_{i,i}^T = A_{i+1..N,i} \rightarrow trsm$
  $A_{i+1..N,i} = Z$
  $A_{i+1..N,i+1..N} =$
    $A_{i+1..N,i+1..N} - A_{i+1..N,i} * A_{i,i+1..N} \rightarrow gemm$
**end**
$Swap\ B\ according\ to\ P$
$Solve\ A * Y = B \rightarrow trsm$
$Solve\ A * X = Y \rightarrow trsm$

Fig. 7: Algorithm for the LU solver with partial pivoting by columns

available to all threads as the vector is stored in shared memory. Next, all threads have to swap the elements of the rows with affinity to them according to $P_i$ before the `trsm` in each iteration. A UPC parallel swapping by columns was developed to efficiently perform these computations. Furthermore, in this algorithm the sequential computations of the next iterations cannot be moved forward because the thread involved in these computations needs to finish its part of all the previous parallel `trsm` and `gemm` to be sure that the pivoting information is well determined.

After the factorization, vector $P$ contains all the information about the pivoting. The columns of matrix $B$ are swapped in parallel before the final triangular solvers.

Again, the correct selection of $NB$ and $BS$ is key to obtain good performance in the algorithms that use the LU factorization.

## V. Performance Evaluation

In order to evaluate the performance of the algorithms explained in Sections III and IV runtime tests were performed on the Carver supercomputer [16] at the National Energy Research Supercomputing Center (NERSC) of the Lawrence Berkeley National Laboratory. This system consists of 320 nodes, each of them with 2 quad-core Intel Xeon 5550X (Nehalem) processors (8 cores at 2.67 Ghz per node) and 24 GB of memory. The compute nodes are interconnected by a 4X QDR InfiniBand network (32 Gbps of theoretical effective bandwidth). As for software, the code was compiled using Berkeley UPC 2.12.2 [17] and linked to the Intel Math Kernel Library (MKL) version 10.2.2 [18], a library with highly tuned BLAS routines for Intel machines. The inter-node communications are performed through GASNet over InfiniBand. All the tests were run using one UPC thread per node and 8 OpenMP threads per UPC thread (one per core). As explained in Sections III and IV, the sequential factorizations are implemented with OpenMP support. Besides, UPCBLAS is linked to the OpenMP multithreaded implementation of the MKL library. The ScaLAPACK library version 1.8.0 was used for comparison purposes.

The performance evaluation of the different algorithms was done using weak scaling and with double precision elements. In a weak scaling study the size of the input matrices increases with the number of cores. As parallel algorithms with many cores are generally used to solve problems that cannot be performed with less resources, weak scaling allows to evaluate the behavior of the algorithms in a more realistic scenario than strong scaling. In this study the average computational workload (number of floating point operations) per thread was fixed for all experiments. Thus, programs with perfect scalability would obtain the same execution times for any number of cores and the speedups are calculated as $(T_1/T_n) * n$, being $T_1$ the sequential execution time and $T_n$ the parallel execution time obtained when using $n$ cores. All the speedups are calculated relative to execution times obtained from the ScaLAPACK library running with only one MPI process.

The UPC algorithms were compared with the MPI implementation using ScaLAPACK. Two experimental results were obtained for each ScaLAPACK routine. The line labeled as `ScaLAPACK-2D` shows in all the graphs the best results obtained by the MPI routine, which are always with a 2D grid of processes. However, as the UPC algorithms can only use 1D data distributions, the results of ScaLAPACK using the best 1D distribution, labeled as `ScaLAPACK-1D`, are also shown for comparative purposes.

As it was explained in Sections III and IV, the size of the blocks in all the block-cyclic distributions has a significant influence on the performance of the solvers. In order to provide a fair comparison, all the experimental results shown were obtained with the best block size for each routine, either for the UPC or the ScaLAPACK routines.

Graphs in Figure 8 compare the execution times (in seconds) and speedups of the two algorithms explained in Section III to implement the Cholesky solver. Results with row and column distributions are shown for both UPC algorithms. These results indicate that the approach that parallelizes the LAPACK algorithm, based on the parallel `gemm` routine, is better than the adaptation of the ScaLAPACK one, based on the parallel `syrk` routine, due to several reasons:

- The dependencies of the algorithm based on `syrk` are

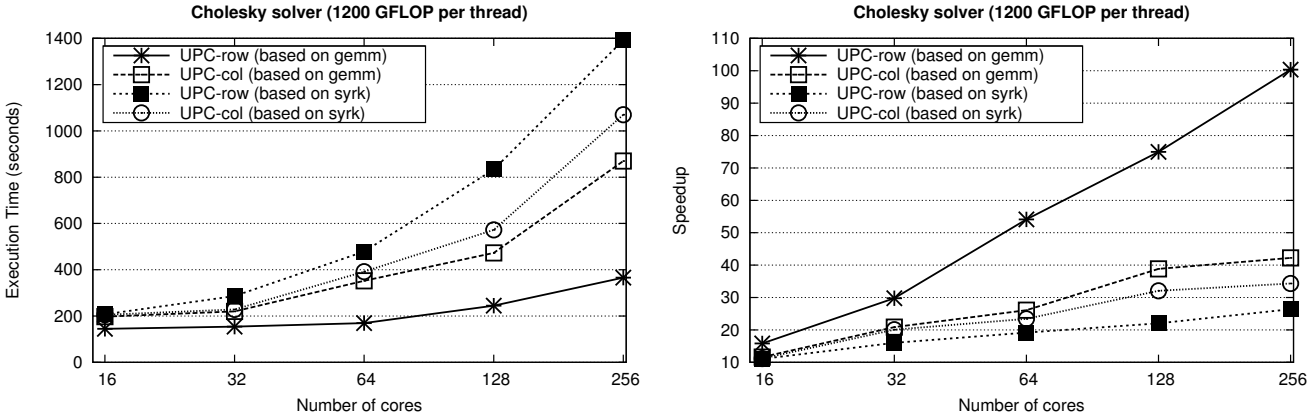


Fig. 8: Execution times and speedups of the UPC implementations of the Cholesky solver

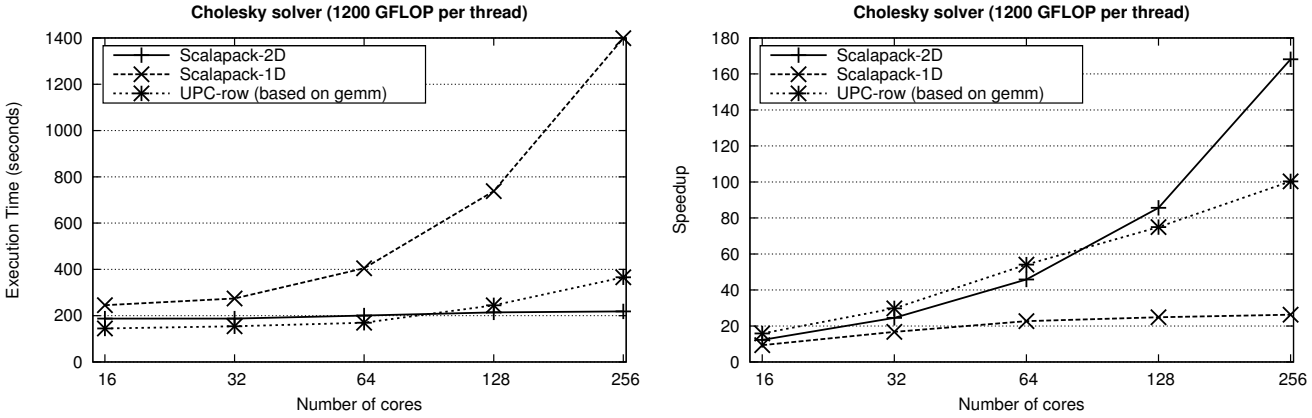


Fig. 9: Execution times and speedups of the MPI and UPC Cholesky solvers

very strong, so more synchronizations are needed, as it was explained in Section III.

- The implementation based on `gemm` has been optimized to minimize the overhead of the sequential computations by overlapping them with other parallel computations. This optimization cannot be included in the other algorithm because of the dependencies.
- The internal pattern of the remote accesses within the parallel `gemm` obtains better performance than the pattern within the parallel `syrk`.

Regarding the best algorithm (the one based on `gemm`) the distribution by rows is better than the distribution by columns. The reason is that the main part of the algorithm is performed using the UPCBLAS version of `gemm` and `trsm`. In the distribution by rows these routines follow the behavior described in Figure 1 for a matrix product $A * B = C$, with matrix $B$ also distributed by rows. As all the elements in the same row have affinity to the same thread, the accesses to one row of $B$ only need one call to the standard UPC function `upc_memget()` per thread. In contrast, in the column distribution case matrix $A$ is also distributed by columns so the elements of each row of $A$ are stored in parts of the shared memory with affinity to different threads (see Figure 2). Therefore, several calls to `upc_memget()` per thread, with less amount of elements per call, are necessary to access each row of $A$. In UPC accessing data using large blocks is much more efficient than splitting them in several smaller accesses.

Figure 9 shows again the results of the best UPC Cholesky solver (the approach based on `gemm` by rows) and compares them to the results obtained by the ScaLAPACK routine. The ScaLAPACK Cholesky solver outperforms the best UPC version for a large number of threads. The reason is that, as explained in Sections I and II, ScaLAPACK sacrifices ease of use and productivity for performance by using complex data structures to be able to work with 2D distributions. Although the UPC 1D algorithms by rows are competitive up to 64 cores, their performance decreases with larger core counts because the blocks become very irregular (few rows and many columns per block). Besides, the configuration of the Berkeley UPC compiler in Carver has a maximum block factor for the shared arrays that leads UPC not being able to use the most appropriate block size in the distribution by rows. In this machine the maximum block factor is 1044000 elements. For instance, the experiment with 256 cores uses matrices of 50500x50500 elements. Thus, the maximum number of rows that can be used in one block is 20 (1044000/50500). This

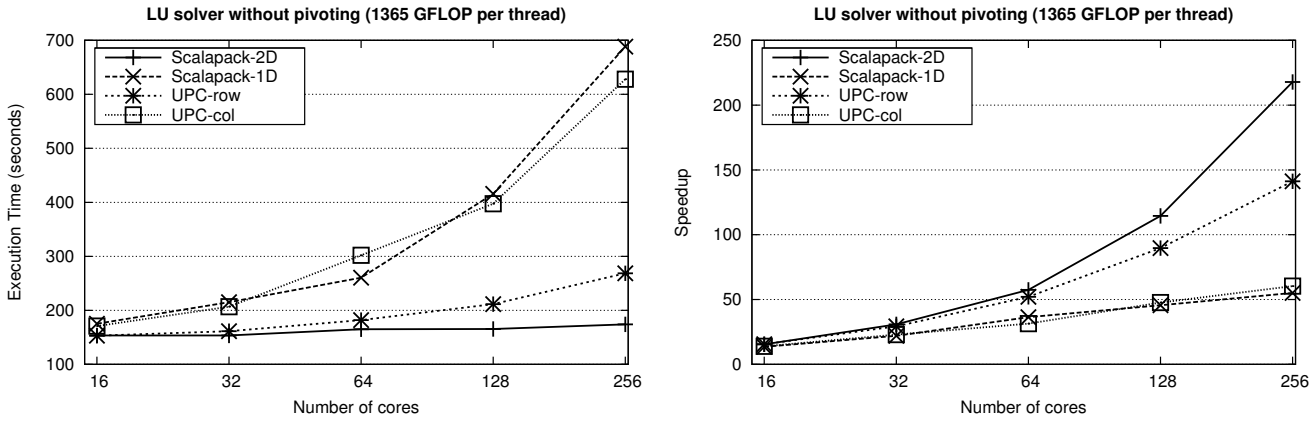


Fig. 10: Execution times and speedups of the MPI and UPC LU solvers without pivoting

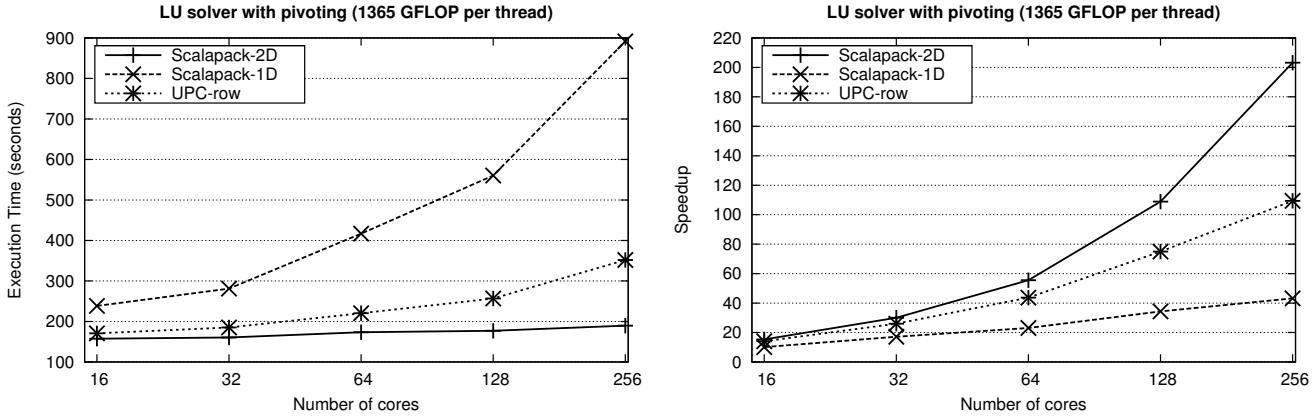


Fig. 11: Execution times and speedups of the MPI and UPC LU solvers with partial pivoting by columns

limitation of the compiler is forcing the UPC algorithms by rows to use more synchronizations than in the ideal scenario. A larger maximum block factor would increase the overall performance for the largest experiments. However, even with these limitations, the UPC approaches were demonstrated to be well implemented (not only the algorithms explained in this work but also the part within UPCBLAS) as their performance is much better than the performance of ScaLAPACK using 1D distributions.

Figure 10 shows the experimental results for the LU solver without pivoting. The conclusions are similar to those of the algorithm using Cholesky based on `gemm`. On the one hand, distributing matrix $A$ by rows obtains better performance than doing it by columns because of the behavior of the UPCBLAS routines. On the other hand, the UPC implementation is much more efficient than the ScaLAPACK 1D routine but less efficient than its 2D counterpart.

Finally, execution times and speedups of the LU solver with partial pivoting are presented in Figure 11. In order to show an extreme scenario, the input matrix $A$ has been selected so that all the columns must be swapped. Regarding the UPC version, only the performance of the distribution by rows is presented as, from the results of Figure 10, it can be inferred to be better than the distribution by columns. Comparing to the UPC algorithm without pivoting, it can be seen that pivoting decreases scalability. The reason is not the time needed to perform the swapping (always only about two seconds per experiment) but the fact that the sequential computations cannot be overlapped with the parallel `gemm` of the previous iteration, as explained in Section IV.

Looking at these results, if the UPC language allows multidimensional distributions in the future, it would be expected for the algorithms explained in Sections III and IV to obtain similar or even better performance than the ScaLAPACK ones. In that case UPCBLAS would provide a new version that would work with these 2D distributions without needing to change the interface of the routines. Only additional options to the enumerate UPCBLAS `dimmdist` should be specified (see Section II). Hence, no changes to the UPCBLAS solvers would be necessary to benefit from the 2D distributions.

## VI. Conclusions

Although UPC and, in general, the PGAS programming model provides important productivity advantages over traditional parallel programming models, most of the works that study numerical algorithms in UPC sacrifice these advantages

for performance. This work has presented UPC implementations of solvers of equations using the Cholesky and LU factorizations (with and without pivoting) based on the UPCBLAS library. All the described algorithms can be implemented as routines to be used for numerical application developers to parallelize their codes. Thanks to the design characteristics of UPCBLAS (syntax similar to sequential BLAS and use of shared arrays), all these routines would have a syntax and use similar to the LAPACK ones. As can be inferred from [10], programming using LAPACK routines is much easier and faster than using the corresponding parallel versions in ScaLAPACK. Therefore, these UPC routines would increase the programmability of parallel numerical codes, producing less error-prone programs and improving the productivity of their users.

The proposed algorithms have been experimentally tested on a multicore cluster to show their suitability and efficiency for hybrid architectures (shared/distributed memory). The obtained results were used in order to determine the most suitable approach for the UPC solvers. On the one hand, two algorithms were implemented for the Cholesky solver and the experimental evaluation has determined that the best choice is the algorithm based on the BLAS3 `gemm` routine. On the other hand, the results obtained in the testbed proved that the distribution by rows, using the largest block size allowed by the compiler, is the most suitable for the solvers, using either Cholesky (based on `gemm`) or LU factorization.

The experimental evaluation has also demonstrated that taking into account the 1D distribution limitation imposed by the UPC language, UPCBLAS solvers achieve good performance. In fact, they obtain much better performance than the ScaLAPACK 1D counterparts. Furthermore, although the ScaLAPACK solvers with 2D distributions obtain the best scalability, the experimental results have confirmed that the approach of UPCBLAS based on shared arrays is a good trade-off between programmability and performance. Besides, if multidimensional distributions were allowed in UPC shared arrays in the future, no changes would be necessary in the UPCBLAS interface to take advantage of the 2D distributions. In that case the algorithms presented in this work (and any other numerical code based on UPCBLAS) could achieve similar or even better performance than ScaLAPACK routines without sacrificing ease of use.

## ACKNOWLEDGMENTS

This work was funded by the Ministry of Science and Innovation of Spain under Project TIN2010-16735, by the Ministry of Education of Spain under the FPU research grant AP2008-01578, and partially funded by the Director, Office of Computational and Technology Research, Advanced Scientific Computing Research Division of the U.S. Department of Energy under contract No. DE-AC03-76SF00098. We gratefully thank Enrique Quintana-Ortí (Jaume I University, Spain) for his valuable help in the development of the Cholesky factorization.

## REFERENCES

[1] UPC Consortium, "UPC Language Specifications, v1.2," 2005, available at http://upc.lbl.gov/docs/user/upc_spec_1.2.pdf. Last visit: January 2012.

[2] Co-Array Fortran, "http://www.co-array.org/," Last visit: January 2012.

[3] Titanium Project, "http://titanium.cs.berkeley.edu/," Last visit: January 2012.

[4] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick, "Optimizing Bandwidth Limited Problems Using One-Sided Communication and Overlap," in *Proc. 20th Intl. Parallel and Distributed Processing Symp. (IPDPS'06)*, Rhodes Island, Greece, 2006.

[5] R. Nishtala, P. H. Hargrove, D. Bonachea, and K. Yelick, "Scaling Communication-Intensive Applications on BlueGene/P Using One-Sided Communication and Overlap," in *Proc. 23rd Intl. Parallel and Distributed Processing Symp. (IPDPS'09)*, Rome, Italy, 2009.

[6] H. Shan, N. Wright, J. Shalf, K. Yelick, M. Wagner, and N. Wichmann, "A Preliminary Evaluation of the Hardware Acceleration of the Cray Gemini Interconnect for PGAS Languages and Comparison with MPI ," in *Proc. 2nd Intl. Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems (PMBS'11)*, Seattle, WA, USA, 2011, pp. 13–14.

[7] R. Nishtala, Y. Zheng, P. Hargrove, and K. Yelick, "Tuning Collective Communication for Partitioned Global Address Space Programming Models," *Parallel Computing*, vol. 37, no. 9, pp. 576–591, 2011.

[8] J. González-Domínguez, M. J. Martín, G. L. Taboada, and J. Touriño, "Dense Triangular Solvers on Multicore Clusters using UPC," in *Proc. 11th Intl. Conf. on Computational Science (ICCS 2011)*, ser. Procedia Computer Science, vol. 4, Singapore, 2011, pp. 231–240.

[9] P. Husbands and K. Yelick, "Multi-threading and One-Sided Communication in Parallel LU Factorization," in *Proc. ACM/IEEE Conf. on High Performance Networking and Computing (SC'07)*, Reno, NV, USA, 2007, pp. 31–41.

[10] LAPACK and ScaLAPACK survey, "http://icl.cs.utk.edu/lapack-forum/survey/," Last visit: January 2012.

[11] J. González-Domínguez, M. J. Martín, G. L. Taboada, J. Touriño, R. Doallo, D. A. Mallón, and B. Wibecan, "UPCBLAS: A Library for Parallel Matrix Computations in Unified Parallel C," *Concurrency and Computation: Practice and Experience*, 2012 (In Press), available at http://dx.doi.org/10.1002/cpe.1914.

[12] Basic Linear Algebra Subprograms (BLAS) Library, "http://www.netlib.org/blas/," Last visit: January 2012.

[13] J. J. Dongarra, J. D. Croz, S. Hammarling, and R. J. Hanson, "An Extended Set of FORTRAN Basic Linear Algebra Subprograms," *ACM Trans. Math. Softw.*, vol. 14, no. 1, pp. 1–17, 1988.

[14] The ScaLAPACK Project, "http://netlib2.cs.utk.edu/scalapack/index.html," Last visit: January 2012.

[15] LAPACK - Linear Algebra PACKage, "http://www.netlib.org/lapack/," Last visit: January 2012.

[16] Carver IBM iDataPlex Supercomputer, "http://www.nersc.gov/systems/carver-ibm-idataplex/," Last visit: January 2012.

[17] Berkeley UPC Project, "http://upc.lbl.gov," Last visit: January 2012.

[18] Intel Math Kernel Library, "http://software.intel.com/en-us/articles/intel-mkl/," Last visit: January 2012.