Dense Triangular Solvers on Multicore Clusters using UPC

Jorge González-Domínguez, María J. Martín, Guillermo L. Taboada and Juan Touriño

Computer Architecture Group, University of A Coruña, Spain
{jgonzalezd, mariam, taboada, juan}@udc.es

Abstract

The popularity of Partitioned Global Address Space (PGAS) languages has increased during the last years thanks to their high programmability and performance through an efficient exploitation of data locality. This paper describes the implementation of efficient parallel dense triangular solvers in the PGAS language Unified Parallel C (UPC). The solvers are built on top of sequential BLAS functions and exploit the particularities of the PGAS paradigm. Furthermore, the numerical routines developed implement an automatic process that adapts the algorithms to the characteristics of the system where they are executed. The triangular solvers have been experimentally evaluated in two different multicore clusters and compared to message-passing based counterparts, demonstrating good scalability and efficiency.
Keywords: UPC, Triangular Solver, BLAS, Autotuning

1 Introduction

The PGAS programming model provides significant productivity advantages over traditional parallel programming paradigms. In this model all threads share a global address space, just as in the shared memory model. However, this space is logically partitioned among threads, just as in the distributed memory model. Thus, the data locality exploitation increases performance, whereas the shared memory space facilitates the development of parallel codes. As a consequence, the PGAS model has been gaining rising attention. A number of PGAS languages are now ubiquitous, being Unified Parallel C (UPC) [1] a representative example. However, a barrier to a more widespread acceptance of UPC, and in general of any PGAS language, is the lack of parallel libraries for developers.

A parallel numerical library for UPC was presented by the authors in [2]. This library contains a relevant subset of the BLAS routines [3, 4], including several types of matrix and vector products and dense triangular solvers (`trsv` and `trsm` routines). However, this first version of the library contained non-optimized routines and particularly the triangular solvers showed an unsatisfactory performance. This paper presents efficient parallel implementations of the UPC dense triangular solvers, both in terms of execution time and memory usage. The proposed codes exploit the particularities of the UPC language, taking into account data locality and the characteristics of the available synchronizations in order to obtain good efficiency.

The importance of designing high performance algorithms for solving linear systems is motivated by many scientific and engineering applications. A common method to solve these systems is the use of factorizations (LU, QR, Cholesky, etc.), which require an efficient implementation of the triangular solvers to obtain good performance. Recently Tomov et al. [5] faced the problem of solving dense triangular systems on multicore architectures with GPU accelerators. In [6] Bell and Nishtala presented a preliminar implementation of sparse triangular solvers in UPC, but there are not, to our knowledge, related works for the dense case.

The rest of this paper is organized as follows. Section 2 discusses the main issues concerning the implementation of the routines in UPC. Section 3 describes the algorithm for the UPC BLAS2 routine `trsv`, and justifies the implementation decisions taken. Section 4 explains the different algorithms that can be applied for the BLAS3 `trsm` routine and analyzes advantages and disadvantages of each proposal. Section 5 presents the analysis of the experimental results obtained on the two multicore cluster testbeds, as well as their comparison with a parallel numerical library based on MPI. Finally, conclusions are discussed in Section 6.

2 Implementation of Efficient UPC Numerical Routines

Numerical libraries are developed not only to improve the programmability of the languages but also to increase the performance of the codes that exploit them. This section discusses considerations and techniques that have been taken into account to design numerical routines in UPC and, more specifically, triangular solvers.

2.1 UPC Optimization Techniques

There is a number of known optimization techniques that improve the efficiency and performance of the UPC codes [7, 8]. The following optimizations have been applied to our codes whenever possible:

- Space privatization: A UPC pointer to shared memory contains 3 fields: *thread*, *block* and *phase*. Thus, when performing pointer arithmetic on a pointer-to-shared all three fields will be updated, making the operation slower than private pointer arithmetic. Experimental measurements in [7] have shown that the use of shared pointers increases execution times by up to several orders of magnitude. Thus, in our routines, when dealing with shared data with affinity to the local thread, the access is performed through standard C pointers instead of using UPC pointers to shared memory.

- Aggregation of remote shared memory accesses: Instead of the costly one-by-one accesses to remote elements, our routines perform remote shared memory accesses through bulk copies, using the `upc_memget()`, `upc_memput()` and `upc_memcpy()` functions on remote bulks of data required by a thread.

- Usage of phaseless pointers: Many UPC compilers (including Berkeley UPC [9]) implement an optimization for the common special case of cyclic and indefinite pointers to shared memory. Cyclic pointers are the ones with a block factor of one, and indefinite pointers with a block factor of zero. Therefore, their phases are always zero. These shared pointers are thus phaseless, and the compiler exploits this knowledge to schedule more efficient operations for them. All auxiliary shared arrays used within the functions to exchange data among threads are declared with indefinite block factor to take advantage of this optimization.

2.2 Efficient Broadcast Communication Model

In the PGAS programming model any thread may directly read or write data located on a remote processor. Therefore, two possible communications models can be applied to the broadcast operations:

- Pull Model: The thread that obtains the data to be broadcast writes them in its shared memory. The other threads are expected to read them from this position. This approach leads to remote accesses from different threads but, depending on the network, they can be performed in a parallel way.

- Push Model: The thread that obtains the data to be broadcast writes them directly in the shared spaces of the other threads. In this case the contention of the network decreases but the writes are sequentially performed.

The pull communication model has experimentally proved to be more efficient than the push one, particularly when the number of threads increases. This will be therefore the communication model used by all the broadcast operations of our parallel routines.

2.3 Subset Barrier Implementation

The algorithm to perform the BLAS3 solver with a multicore-aware distribution (see Section 4.3) requires synchronizations that only concern to a group of threads. However, currently there is no functionality to work with teams of threads in the UPC language. The topic of subsets (or teams) of threads in UPC has been addressed in previous works. Nishtala et al. [10] have proposed extensions to the UPC collective library in order to allow collective operations on teams of threads. Dinan et al. [11] have provided a workaround for teams in UPC by using hybrid UPC+MPI codes.

In our codes, in order to avoid synchronizations with unnecessary threads, global barriers (`upc_barrier` or `upc_notify+upc_wait`) were discarded. Instead, a barrier for a subset of threads was implemented. All threads that belong to a team access a control variable in shared memory (whose initial value must be 0) and, if its value is equal to the number of threads per team, they continue with the execution. Otherwise, they keep accessing the control variable until it reaches that value. To avoid multiple remote memory accesses to check the current value, there is one copy of the control variable in each shared space. Therefore, each thread only performs repetitive accesses to the copy available in its shared memory. When a thread enters into this special barrier it increases in one unit the value of all the copies of the control variable.

2.4 Data distributions

Data distributions have a serious impact on the performance of parallel programs. Parallel numerical libraries developed using the message-passing paradigm force the user to distribute the elements of the input vectors and matrices among all processes. In UPC shared arrays are implicitly distributed across the memories of the different threads. The default layout is cyclic, but UPC provides layout specifiers to allow block-cyclic distributions. Nevertheless, shared matrices in UPC can only be distributed in one dimension as the UPC syntax does not allow multidimensional layouts. The definition of multidimensional blocking factors has been proposed in [12], but currently there are no plans to include this extension in the language specification. Therefore, all the numerical UPC routines developed will apply one-dimensional matrix distributions.

2.5 Underlying Efficient Sequential Numerical Libraries

Besides the balance of workload among UPC threads through an efficient data layout and the use of scalable communications, it is necessary to rely on efficient sequential numerical libraries to obtain good performance. All the parallel algorithms proposed call internally to sequential BLAS routines to perform the computations in each thread. These calls can be linked to very optimized libraries like the Intel Math Kernel Library (MKL) [13]. Just in case a numerical library is not available in the system, sequential implementations using ANSI C are also provided.

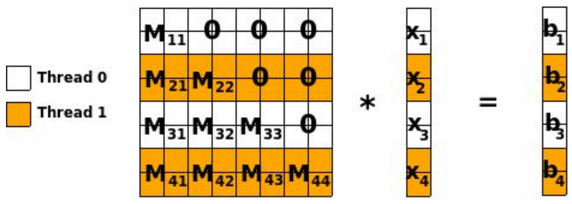
3 BLAS2 Triangular Solver

The `trsv` routine from the level 2 BLAS library solves a system of linear equations $M * x = b$, being M an mxm upper or lower triangular matrix, and x and b vectors of length m. In the parallel algorithm proposed vector b is overwritten by the solution vector x.

Triangular solvers are often part of direct methods to solve linear systems. They are usually preceded by a matrix factorization. The standard matrix distribution used for the parallel implementation of the factorization algorithm in a distributed-memory system is 2D block-cyclic. Thus, in order to avoid expensive redistributions of data, triangular solvers should adopt the data layout scheme used in the factorization process. This is the reason why the parallel version of this routine present in the PBLAS [14] and ScaLAPACK [15] libraries uses a 2D block-cyclic distribution of the triangular matrix in spite of the simpler 1D distributions that are more adequate for this routine [16].

The UPC versions, however, are not conditioned by 2D distributions in the factorization as the UPC language only allows 1D distributions. Thus, the more efficient 1D block-cyclic distribution was used. In the parallel algorithm, the rows of the triangular matrix are distributed across the threads in a block-cyclic way. A block version of the triangular solver algorithm is used to better exploit memory hierarchy. Each block of rows is logically divided in square submatrices. The triangular solver is then computed as a sequence of triangular solutions and matrix-vector multiplications, which can be performed with calls to the sequential BLAS2 `trsv` and `gemv` routines.

Figure 1 shows an example for a lower triangular coefficient matrix using two threads and two blocks per thread. The triangular matrix is logically divided in square blocks M_{ij}. These blocks are triangular submatrices if $i = j$, square submatrices if $i > j$, and null submatrices if $i < j$. The right part of Figure 1 shows the parallel algorithm for this example. Once one thread computes its part of the solution (output of the sequential `trsv`), it is broadcast to all threads so they can update their local parts of b with the sequential product (`gemv`). Note that all operations between two synchronizations can be performed in parallel.



Solve $M_{11} * x_1 = b_1$	***trsv()***
Broadcast x_1	***synchr***
$b_2 = b_2 - M_{21} * x_1$	***gemv()***
$b_3 = b_3 - M_{31} * x_1$	***gemv()***
$b_4 = b_4 - M_{41} * x_1$	***gemv()***
Solve $M_{22} * x_2 = b_2$	***trsv()***
Broadcast x_2	***synchr***
$b_3 = b_3 - M_{32} * x_2$	***gemv()***
...	

Figure 1: Matrix distribution and algorithm for the parallel BLAS2 triangular solver

3.1 Determination of the Number of Blocks

The block size has a great impact on the performace of the parallel solver. The more blocks the matrix is divided in, the more computations can be simultaneously performed, but the more synchronizations are needed too. Thus, it is necessary to find a good trade off between the benefits of parallelism and synchronization overhead.

We have followed an autotuning approach. The performance of the BLAS2 function was tested in several scenarios varying the size of the matrix, the number of threads and the number of blocks per thread in order to perform a regression analysis with the execution times to determine the most suitable number of blocks. Two main conclusions were taken from these experiments:

- The size of the problem has not significant influence on the most suitable number of blocks per thread. However, further experiments have shown that there is a minimum block size to exploit parallelism (1000 rows in the single precision case).
- The most suitable number of blocks per thread decreases less than linearly with the number of threads.

For illustrative purposes, graphs in Figure 2 show the execution times for some representative cases: matrices with 20000, 25000 and 30000 rows and columns; 4, 8 and 16 threads; and from 2 to 8 blocks per thread. The testbed was a small departmental x86_64 cluster with 16 nodes with InfiniBand network (20 Gbps). Each node has 2 Intel Xeon Nehalem quadcore E5520 CPUs at 2.27 GHz and 8 GBytes of memory. As for software, the code was compiled using Berkeley UPC 2.10.0 [17] and linked to MKL version 11.1 to perform the sequential computations. As we can see in the figure, the best execution times are obtained for 4 blocks per thread using 4 threads, 3 blocks for 8 threads and 2 blocks for 16 threads, independently of the matrix size.

A thorough analysis of all the results has shown that, if the matrix is large enough, the number of blocks per thread (*blocks_per_th*) can be specified as $\lceil F/\sqrt{THREADS}\rceil$, being F a constant related to the hardware characteristics of the system and the performance of the underlying sequential numerical library and $THREADS$ the total number of threads in the UPC execution. In Section 5 this formula will be proved to work for two multicore clusters with very different hardware characteristics. The estimate of parameter F is performed by executing the parallel BLAS2 triangular solver with different block sizes for a fixed matrix size (M_SIZE) and a fixed number of threads (NUM_TH). Once the block size that obtains the best performance for that experiment is established ($BEST\_BLK\_SIZE$), the value of F is calculated

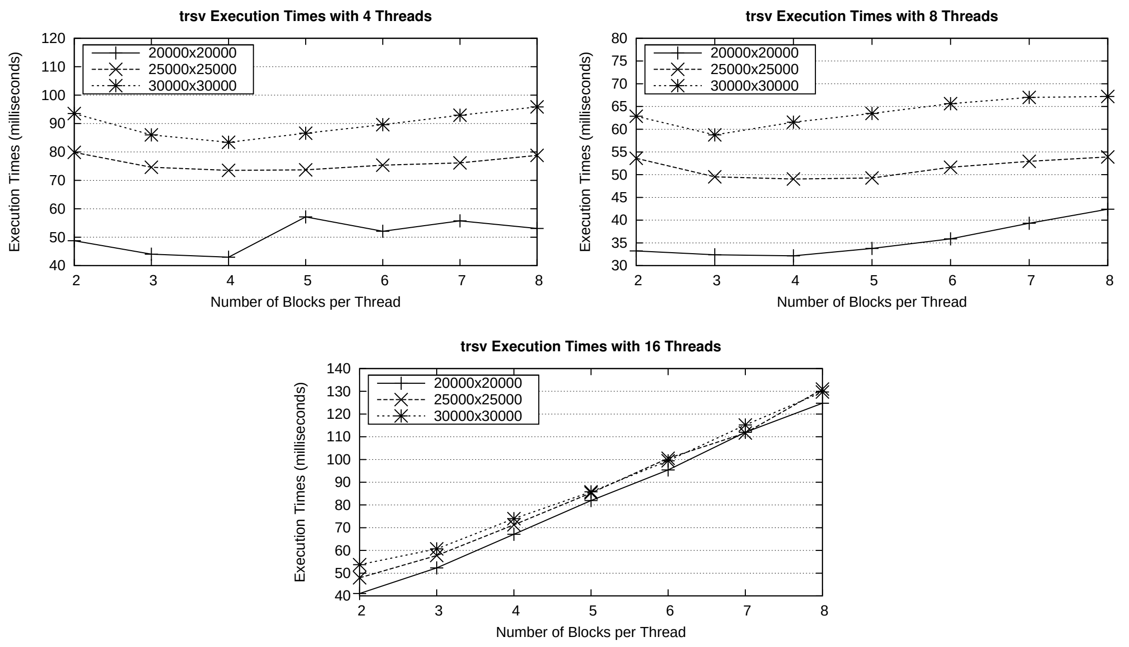


Figure 2: BLAS2 triangular solver performance according to the size of the matrix, the number of threads and the number of blocks per thread

as: $F = \frac{M\_SIZE}{BEST\_BLK\_SIZE * \sqrt{NUM\_TH}}$. This value is stored in an environment variable to be read in the following executions. The UPC routine reads this value and automatically applies the appropriate block size in the execution of the parallel routine.

4 BLAS3 Triangular Solver

The `trsm` routine from the level 3 BLAS library solves a triangular system of equations with multiple right hand sides of the form $M * X = B$, where M is an mxm triangular matrix and X and B are two mxn general matrices. Thus, in the BLAS3 triangular solvers not only the data distribution must be decided but also the matrices to be distributed. This section analyzes different data layout alternatives. In all the parallel algorithms proposed matrix B is overwritten by the solution matrix X.

4.1 Distribution of the Triangular and the General Matrices

The first approach to parallelize this routine consists of adapting the algorithm and data distribution studied in Section 3 for the BLAS2 triangular solver. In this case all matrices are distributed by rows in a block-cyclic way and using the mechanism shown in Section 3.1 to find the appropriate block size. The internal behavior is similar to the one shown in Figure 1 but using the sequential `trsm` and `gemm` level 3 BLAS routines, instead of the level 2 ones (`trsv` and `gemv`, respectively). This approach keeps one synchronization per block which, as will be shown in Section 5.2, limits the scalability of the routine. However, this will be the best choice if the input matrices are already distributed across the threads or in case of memory limitations.

4.2 Replication of the Triangular Matrix

The BLAS3 routine can be seen as a set of n independent BLAS2 triangular solvers (one per column of the general matrices). Therefore, from the parallelism point of view, a better option is that each thread performs a subset of BLAS2 solvers in order to avoid the internal synchronizations. This option requires to replicate the triangular matrix and to distribute the general matrices using a block-cyclic distribution by columns. Figure 3 despicts this approach for a triangular matrix with 6 rows and columns, general matrices with 8 columns, two threads and two blocks per thread. In this case the algorithm consists of using the whole triangular matrix in all threads to compute a sequential BLAS3 triangular solver only with the corresponding subset of columns of X and B (in Figure 3 each thread has a partial matrix with dimensions 6x4). Using this data layout no communications are needed and the algorithm is completely parallel. However, it presents greater memory requirements due to the replication of the triangular matrix.

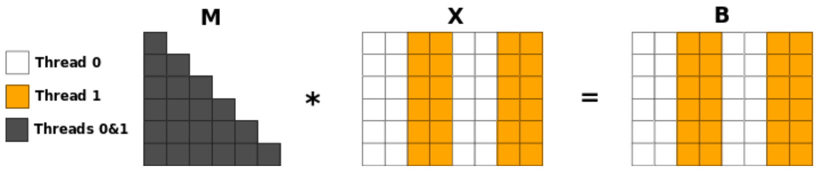


Figure 3: Example of the BLAS3 triangular solver with M replicated and X and B distributed by columns

4.3 Multicore-Aware Distribution

Nowadays, the most commonly deployed systems are multicore clusters where one core can communicate with other cores placed in the same node by using shared memory, but it communicates with other nodes through a network. These inter-node communications are usually much more expensive than the intra-node ones. Furthermore, the network can also represent a significant performance bottleneck when contention and congestion arise.

In this subsection a new distribution that takes into account the architecture of the underlying system is proposed. It consists of replicating M and distributing X and B by blocks of columns as in Section 4.2, but only among the different nodes. It means, each node performs a different partial BLAS3 triangular solver. However, in this case, the threads mapped into the same node perform the corresponding partial solver in a parallel way by applying the parallel algorithm explained in Section 4.1. Thus, within a node the threads only access a subset of the rows of the matrices, according to a block-cyclic internal distribution. Figure 4 shows the distribution of data applying this multicore-aware strategy for two nodes, two threads per node and the same matrix sizes as in Figure 3.

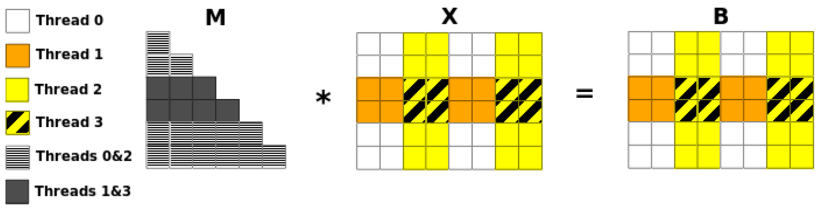


Figure 4: Example of the multicore-aware distribution for the BLAS3 triangular solver. Threads 0 and 1 are in the first node and threads 2 and 3 in the second one

In this algorithm the synchronizations are performed only among threads within the same node, thus reducing the overhead per block as compared to the distribution explained in Section 4.1. Besides, compared to the approach explained in Section 4.2, this distribution presents the same memory overhead but it should increase the reuse of data and, theoretically, improve the performance. As communications among subsets of threads are needed, the subset barrier explained in Section 2.3 is used.

Servet [18, 19], a benchmark suite to obtain the system parameters on multicore clusters (e.g. memory bandwidths and communication latencies), is used to know which threads are placed in the same node. The hardware information provided by Servet is automatically applied for the parallel routine.

5 Experimental Evaluation

Two different multicore clusters were used to evaluate our proposal, the x86_64 cluster described in Section 3.1 and the Finis Terrae supercomputer, located at the Galicia Supercomputing Center (CESGA). This supercomputer consists of 142 HP RX7640 nodes, each of them with 16 IA64 Itanium2 Montvale cores at 1.6 Ghz, 128 GB of memory and a dual 4X InfiniBand port (16 Gbps of theoretical effective bandwidth). The cores of each node are distributed in two cells, each of them with 8 cores and 64 GB of shared memory. Within each cell, there are 4 dual-core processors, grouped in pairs that share the memory bus. A mapping policy which assigns the threads to cores that do not share the bus to memory is applied in order to avoid the overheads due to concurrent memory accesses. As for software, the functions were linked to MKL version 10.1 and compiled with Berkeley UPC 2.8.0 (using shared memory for intra-node communications and the GASNet library over InfiniBand for inter-node ones).

Comparisons with the MPI triangular solvers provided by the ScaLAPACK library [15], included in MKL, were performed. To use ScaLAPACK all matrices and vectors must be distributed by the programmer prior to calling the numerical routine; so, to guarantee a fair comparison, several experiments with different matrix distributions and block sizes were completed. One-dimensional distributions by rows and by columns and two-dimensional distributions using different process topologies were tested, always obtaining the best performance with the one-dimensional distribution by rows. The best execution time is taken as reference in all the comparisons. Finally, in order to provide a fair comparison, all the speedups are calculated relative to the sequential times provided by the MKL triangular solver routines, which present the best performance.

5.1 Evaluation of the BLAS2 Routine

Execution times and speedups for the BLAS2 triangular solver for different problem sizes and both testbeds are shown in Tables 1 and 2. All the experiments were repeated 20 times, saving the best execution times as representative. The UPC implementation uses a pull model for the broadcasts and a one-dimensional distribution by rows, with the number of blocks per thread automatically calculated as explained in Section 3.1. Only results using up to 16 threads are included because this function does not scale with more threads, neither with the UPC nor with the ScaLAPACK version, due to the low computational times (in the order of milliseconds) and the synchronizations involved in the algorithm shown in Figure 1. This is an intrinsic problem when parallelizing this routine, also present in the ScaLAPACK version, that only obtains fairly better performance in the x86_64 cluster. Regarding the Itanium2 supercomputer, the UPC version significantly overcomes the ScaLAPACK one, which is not correctly optimized for the Itanium2 architecture.

BLAS2 Triangular Solver (ms)				
Dim →	20000x20000		30000x30000	
MKL	108.12		234.72	
THREADS ↓	UPC	ScaLAPACK	UPC	ScaLAPACK
1	129.31	116.98	264.00	241.96
2	68.82 (1.57)	53.99 (2.00)	135.77 (1.73)	126.98 (1.85)
4	42.95 (2.52)	41.99 (2.57)	83.40 (2.81)	82.99 (2.83)
8	32.40 (3.34)	31.00 (3.49)	58.75 (4.00)	54.99 (4.27)
16	36.88 (2.93)	30.40 (3.56)	53.73 (4.37)	47.99 (4.89)

Table 1: Execution times (in milliseconds) and speedups (in parentheses) of the single precision BLAS2 triangular solver in UPC compared to ScaLAPACK in the x86_64 cluster ($blocks\_per\_th = \lceil 8/\sqrt{THREADS} \rceil$)

5.2 Evaluation of the BLAS3 Routine

Tables 3 and 4 and graphs in Figure 5 show the execution times and speedups for the BLAS3 triangular solver, both for the 3 UPC versions (*M_dist*, *M_rep* and *multi*, described in Sections 4.1, 4.2 and 4.3, respectively) and the ScaLAPACK version. Three different scenarios were studied in each testbed according to the shapes

BLAS2 Triangular Solver (ms)				
Dim →	30000x30000		40000x40000	
MKL	447.97		780.01	
THREADS ↓	UPC	ScaLAPACK	UPC	ScaLAPACK
1	462.38	940.00	799.10	1928.00
2	267.88 (1.67)	444.00 (1.01)	470.01 (1.66)	872.00 (0.89)
4	148.08 (3.03)	276.00 (1.63)	258.84 (3.01)	456.00 (1.71)
8	81.08 (5.53)	172.00 (2.60)	141.74 (5.50)	288.00 (2.71)
16	50.28 (8.91)	148.00 (3.02)	70.21 (11.11)	216.00 (3.61)

Table 2: Execution times (in milliseconds) and speedups (in parentheses) of the single precision BLAS2 triangular solver in UPC compared to ScaLAPACK in the Itanium2 supercomputer ($blocks\_per\_th = \lceil 16/\sqrt{THREADS} \rceil$)

of X and B: square matrices, more rows than columns and more columns than rows. Only three executions per experiment were performed because the variability is almost negligible.

BLAS3 Triangular Solver (s)								
Dim →	10000x10000				12000x4000			
MKL	59.02				33.08			
THREADS ↓	UPC M_dist	UPC M_rep	UPC multi	ScaLAPACK	UPC M_dist	UPC M_rep	UPC multi	ScaLAPACK
1	61.95	61.95	61.95	59.41	36.02	36.02	36.02	34.13
2	30.08	31.03	31.63	29.55	17.06	18.14	18.18	16.97
4	16.66	15.42	16.69	17.06	9.56	8.93	9.52	9.48
8	9.39	7.76	8.86	9.97	5.40	4.49	4.81	5.36
16	7.22	3.94	4.85	5.93	4.05	2.46	2.93	3.22
32	10.96	2.00	2.87	4.95	5.61	1.36	1.66	2.78

Dim →	6000x25000			
MKL	53.14			
THREADS ↓	UPC M_dist	UPC M_rep	UPC multi	ScaLAPACK
1	56.26	56.26	56.26	54.07
2	27.08	28.33	28.62	26.69
4	15.84	14.14	14.93	16.15
8	9.80	6.99	8.13	10.04
16	10.11	3.56	4.90	6.80
32	15.61	1.81	3.18	5.98

Table 3: Execution times (in seconds) of the single precision BLAS3 triangular solver in UPC, with matrix M distributed (M_dist, $blocks\_per\_th = \lceil 64/\sqrt{THREADS} \rceil$), replicated ($M\_rep$) and the multicore-aware distribution (*multi*), compared to ScaLAPACK in the x86_64 cluster

As expected, the algorithm that distributes the triangular matrix obtains the worst scalability, because many synchronizations are necessary. Furthermore, although the synchronizations for the multicore-aware distribution are only performed among threads inside the same node, their overhead is significant enough to obtain lower efficiency than the distribution with the triangular matrix replicated. Thus, the multicore-aware distribution was discarded in the final version of the routine. The other two distributions are kept as an option to the user which should decide the best alternative in function of the initial conditions (triangular matrix distributed or replicated) and memory constraints.

Regarding the comparison with MPI, ScaLAPACK forces the user to distribute all matrices. With this assumption, the performance of ScaLAPACK is quite good (higher than the UPC counterpart *M_dist*) but it is worse than the UPC version where all threads have the triangular matrix replicated (*M_rep*). These experiments were also used to prove that replicating the triangular matrix is a good choice in any case (either the general matrices are square, or present more rows than columns or vice versa).

BLAS3 Triangular Solver (s)								
Dim →	12000x12000				15000x4000			
MKL	270.13				267.74			
THREADS ↓	UPC M_dist	UPC M_rep	UPC multi	ScaLAPACK	UPC M_dist	UPC M_rep	UPC multi	ScaLAPACK
1	276.46	276.46	276.46	304.35	145.63	145.63	145.63	156.25
2	146.85	140.36	142.10	150.96	75.44	73.72	74.08	80.27
4	76.60	70.30	72.28	82.33	38.92	36.63	37.41	41.76
8	41.82	35.81	39.10	45.60	20.77	18.55	20.26	22.49
16	23.08	17.92	20.46	27.96	11.61	9.44	10.50	12.81
32	16.99	9.09	11.27	19.02	7.39	5.00	5.81	9.12
64	25.98	4.64	6.76	16.04	10.43	2.77	3.60	5.41
128	44.84	2.62	4.92	14.66	61.30	1.45	2.62	4.33

Dim →	8000x25000			
MKL	140.52			
THREADS ↓	UPC M_dist	UPC M_rep	UPC multi	ScaLAPACK
1	271.47	271.47	271.47	304.35
2	141.79	141.43	136.70	148.79
4	73.67	65.38	68.58	83.00
8	40.87	33.28	37.88	49.05
16	23.94	16.45	20.14	33.33
32	20.32	8.31	11.42	24.64
64	33.22	4.24	7.24	23.27
128	65.30	2.16	5.45	13.89

Table 4: Execution times (in seconds) of the single precision BLAS3 triangular solver in UPC, with matrix M distributed (*M_dist*, $blocks\_per\_th = \lceil 96/\sqrt{THREADS} \rceil$), replicated (*M_rep*) and the multicore-aware distribution (*multi*), compared to ScaLAPACK in the Itanium2 supercomputer

6 Conclusions

This work has addressed the most important issues to implement, in an efficient way, the UPC BLAS2 and BLAS3 triangular solvers. Block forms of the sequential algorithms were used allowing the corresponding UPC parallel algorithms to rely on sequential BLAS routines to perform the local computations. Using sequential libraries not only improves efficiency, but it also allows to incorporate automatically new versions as soon as available without any change in the UPC code, taking advantage of the improvements included in new releases.

Besides, a special effort was made to find the best data distributions to improve the performance. In this work the one-dimensional block-cyclic distribution by rows was applied to the BLAS2 solver as it was proved to be the most efficient distribution. Furthermore, an autotuning mechanism to find the optimal block size was also presented.

As for the BLAS3 routine, several data distributions were analyzed. The best choice depends on memory constraints and initial distributions of the input matrices. In this regard, the BLAS3 routine implemented allows the user to select the data distribution that better fits his/her requirements.

All the proposals were evaluated on two different multicore clusters demonstrating scalability and efficiency according to their possibilities (the low sequential execution times in the BLAS2 solver limit scalability). The UPC versions were also compared to the MPI counterparts present in the ScaLAPACK library. We can assert that the ease of use of the UPC routines (a explicit data distribution is not needed in the UPC version) does not lead to significant worse performance. Furthermore, UPC routines behave even better in some cases.

The developed routines have been included in the UPCBLAS library, a portable and efficient parallel numerical library for dense and sparse computations using the UPC language. The global aim is to improve programmability and performance of UPC applications, and thus lead to a widespread acceptance of the language.

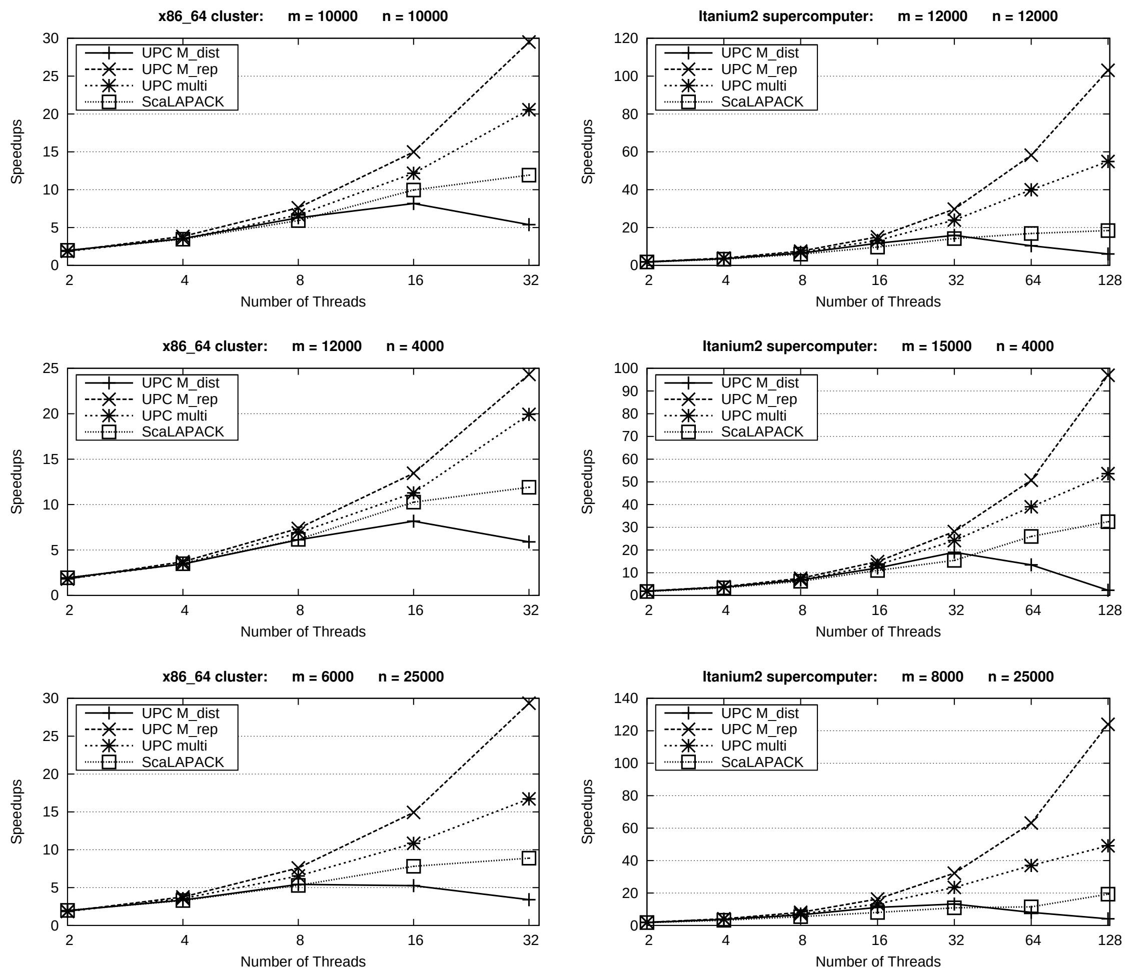


Figure 5: Speedups of the single precision BLAS3 triangular solver in UPC, with matrix M distributed (*M_dist*), replicated (*M_rep*) and the multicore-aware distribution (*multi*), compared to ScaLAPACK in the x86_64 cluster and the Itanium2 supercomputer

Acknowledgments

This work was funded by Hewlett-Packard (Project ”Improving UPC Usability and Performance in Constellation Systems: Implementation/Extensions of UPC Libraries”), and by the Ministry of Science and Innovation of Spain under Project TIN2010-16735 and under an FPU grant AP2008-01578. We gratefully thank CESGA (Galicia Supercomputing Center, Santiago de Compostela, Spain) for providing access to the Finis Terrae supercomputer.

References

References

[1] UPC Consortium. UPC Language Specifications, v1.2, 2005. `http://upc.lbl.gov/docs/user/upc_spec_1.2.pdf`.

[2] Jorge González-Domínguez, María J. Martín, Guillermo L. Taboada, Juan Touriño, Ramón Doallo, and Andrés Gómez. A Parallel Numerical Library for UPC. In *Proc. 15th Intl. European Conf. on Parallel and Distributed Computing (Euro-Par 2009)*, LNCS 5704, pages 630–641, Delft, The Netherlands, 2009.

[3] Basic Linear Algebra Subprograms (BLAS) Library. `http://www.netlib.org/blas/`, Last visit: March 2011.

[4] Jack J. Dongarra, Jeremy D. Croz, Sven Hammarling, and Richard J. Hanson. An Extended Set of FORTRAN Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw.*, 14(1):1–17, 1988.

[5] Stanimire Tomov, Rajib Nath, Hatem Ltaief, and Jack J. Dongarra. Dense Linear Algebra Solvers for Multicore with GPU Accelerators. In *Proc. 24th Intl. Parallel and Distributed Processing Symp. (IPDPS'10)*, Atlanta, GE, USA, 2010.

[6] Christian Bell and Rajesh Nishtala. UPC Implementation of the Sparse Triangular Solve and NAS FT. *CS267 Final Project, Computer Science Division, University of California at Berkeley, USA (2004)*, 2004.

[7] Tarek El-Ghazawi and François Cantonnet. UPC Performance and Potential: a NPB Experimental Study. In *Proc. 14th ACM/IEEE Conf. on Supercomputing (SC'02)*, pages 1–26, Baltimore, MD, USA, 2002.

[8] Damián A. Mallón, Guillermo L. Taboada, Carlos Teijeiro, Juan Touriño, Basilio B. Fraguela, Andrés Gómez, Ramón Doallo, and José C. Mouriño. Performance Evaluation of MPI, UPC and OpenMP on Multicore Architectures. In *Proc. 16th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'09)*, LNCS 5759, pages 174–184, Espoo, Finland, 2009.

[9] Wei-Yu Chen, Dan Bonachea, Jason Duell, Parry Husbands, Costin Iancu, and Katherine A. Yelick. A Performance Analysis of the Berkeley UPC Compiler. In *Proc. 17th Intl. Conf. on Supercomputing (ICS'03)*, pages 63–73, San Francisco, CA, USA, 2003.

[10] Rajesh Nishtala, George Almási, and Călin Casçaval. Performance without Pain = Productivity: Data Layout and Collective Communication in UPC. In *Proc. 13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP'08)*, pages 99–110, Salt Lake City, UT, USA, 2008.

[11] James Dinan, Pavan Balaji, Ewing Lusk, P. Sadayappan, and Rajeev Thakur. Hybrid Parallel Programming with MPI and Unified Parallel C. In *Proc. of the 7th ACM Intl. Conf. on Computing Frontiers (CF'10)*, pages 177–186, Bertinoro, Italy, 2010.

[12] Christopher Barton, Călin Casçaval, George Almási, Rahul Garg, José N. Amaral, and Montse Farreras. Multidimensional Blocking in UPC. In *Proc. 20th Intl. Workshop on Languages and Compilers for Parallel Computing (LCPC'07)*, volume 5234 of *Lecture Notes in Computer Science*, pages 47–62, Urbana, IL, USA, 2007.

[13] Intel Math Kernel Library. `http://www.intel.com/cd/software/products/asmo-na/eng/307757.htm`, Last visited: March 2011.

[14] PBLAS Home Page. `http://www.netlib.org/scalapack/pblasqref.html`, Last visited: March 2011.

[15] The ScaLAPACK Project. `http://netlib2.cs.utk.edu/scalapack/index.html`, Last visited: March 2011.

[16] Eunice E. Santos. On Designing Optimal Parallel Triangular Solvers. *Information and Computation*, 161(2):172–210, 2000.

[17] Berkeley UPC Project. `http://upc.lbl.gov`, Last visited: March 2011.

[18] The Servet Benchmark Suite Project. `http://servet.des.udc.es/`, Last visited: March 2011.

[19] Jorge González-Domínguez, Guillermo L. Taboada, Basilio B. Fraguela, María J. Martín, and Juan Touriño. Servet: A Benchmark Suite for Autotuning on Multicore Clusters. In *Proc. 24th IEEE Intl. Parallel and Distributed Processing Symp. (IPDPS'10)*, Atlanta, GA, USA, 2010.