UPCBLAS: A Library for Parallel Matrix Computations in Unified Parallel C

Jorge González-Domínguez, María J. Martín, Guillermo L. Taboada,
Juan Touriño, Ramón Doallo
Computer Architecture Group, University of A Coruña, Spain
Damián A. Mallón
Galicia Supercomputing Center (CESGA), Santiago de Compostela, Spain
Brian Wibecan
Hewlett-Packard, USA

September 11, 2012

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, especially on hierarchical architectures such as multicore clusters. This paper describes UPCBLAS, a parallel numerical library for dense matrix computations using the PGAS Unified Parallel C (UPC) language. The routines developed in UPCBLAS are built on top of sequential BLAS functions and exploit the particularities of the PGAS paradigm, taking into account data locality in order to achieve a good performance. Furthermore, the routines implement other optimization techniques, several of them by automatically taking into account the hardware characteristics of the underlying systems on which they are executed. The library has been experimentally evaluated on a multicore supercomputer and compared to a message-passing based parallel numerical library, demonstrating good scalability and efficiency.

keywords: Parallel Library; Matrix Computations; PGAS; UPC; BLAS

1 Introduction

The Partitioned Global Address Space (PGAS) programming model provides significant productivity advantages over traditional parallel programming paradigms. In the PGAS 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, such as Titanium [1], Co-Array Fortran [2] and Unified Parallel C (UPC) [3].

UPC is an extension of ANSI C for parallel computing. In [4] El-Ghazawi and Cantonnet established, through an extensive evaluation of experimental results, that UPC can potentially perform at similar levels to those of MPI. Besides, the one-sided communications present in languages such as UPC were demonstrated to be able to obtain even better performance than the traditional two-sided communications [5]. Barton et al. [6] further demonstrated that UPC codes can scale up to thousands of processors with the right support from the compiler and the run-time system. More up-to-date evaluations [7, 8] have confirmed this analysis. Currently, there are commercial and open source UPC compilers, such as Berkeley UPC [9], GCC UPC [10], HP UPC [11] or IBM UPC [12], for nearly all parallel machines. However, a barrier to a more widespread acceptance of UPC is the lack of parallel libraries for UPC developers.

This paper presents UPCBLAS, a parallel numerical library with a relevant subset of the BLAS routines [13, 14] implemented for UPC. The BLAS library provides standard building blocks for performing basic vector and matrix operations and it is widely used by scientists and engineers as it usually obtains good levels of performance through an efficient exploitation of the memory hierarchy. The developed UPCBLAS routines exploit the particularities of the PGAS paradigm and they call internally BLAS routines to perform the sequential computations within each thread.

The rest of this paper is organized as follows. Section 2 summarizes the related work. Section 3 provides an overview of the memory model in UPC, as background for the following sections. Section 4 describes the UPCBLAS design, including the parallel algorithms to perform the most representative routines. Section 5 explains the optimization techniques used in the implementation in order to provide efficient BLAS routines in UPC. Section 6 presents the analysis of the experimental results obtained on an HP supercomputer (Finis Terrae), as well as their comparison with PBLAS, a parallel numerical library based on MPI. Section 7 analyzes the advantages and drawbacks of UPC to develop a parallel numerical library. Finally, conclusions are discussed in Section 8.

2 Related Work

In the literature there are several numerical libraries with support for parallel dense matrix computations. Among them, PBLAS [15, 16], a subset of BLAS, and ScaLAPACK [17], a subset of LAPACK [18], are the most popular ones. Based on them, Aliaga et al. [19] made an effort to parallelize the open source numerical library GSL [20]. Moreover, many vendors provide their own parallel numerical libraries, such as Intel MKL [21], IBM PESSL [22] and HP MLIB [23]. All of them implement parallel numerical routines using the message-passing paradigm, assisting programmers of distributed memory systems.

The main drawback of the message-passing based libraries is that they need an explicit data distribution that increases the effort required to use them. Users are forced to deal with specific structures defined in the library and to work in each process only with the part of the matrices and vectors stored in the local memory. Therefore, users must be aware of the appropriate local indexes to use in each process, which increases the complexity of developing parallel codes [24]. In the literature there exist some proposals that try to ease the use of message-passing numerical libraries, such as PyScaLAPACK [25] and Elemental [26]. Following this trend, one of the goals of UPCBLAS is to increase programmability. The PGAS languages in general, and UPC in particular, offer productivity advantages compared to the message-passing model. In [27] the number of lines of code needed by the MPI and the UPC implementations of the NAS Parallel Benchmarks and other kernels are compared. Similar statistical studies with university students are presented in [28] and [29]. These works have demonstrated that the effort needed to solve the same problem is lower in UPC than in MPI. Furthermore, the global address space in UPC allows to hide the complex local index generation for matrices and vectors as well as data movement issues present in the message-passing approaches. Experimental results will show that simplicity does not significantly impact performance.

Regarding other PGAS libraries proposals, a parallel numerical library that combines the object-oriented-like features of Fortran 95 with the parallel syntax of Co-Array Fortran was presented in [30]. However, its object-oriented layer leads to use object maps, additional structures to work with distributed matrices and vectors similarly to the message-passing based libraries, which increases the effort needed to parallelize sequential numerical algorithms. Travinin and Kepner [31] developed pMatlab, a parallel library built on top of MatlabMPI (a set of Matlab scripts that implement a subset of MPI). It works with matrices and vectors distributed by simulating a pure PGAS scenario in order to take advantage of the ease of programming and a higher level of abstraction.

Focusing on numerical computations in UPC, Bell and Nishtala present in [32] a sparse triangular solver in UPC. In [33] Husbands and Yelick undertake the parallelization of the LU factorization. However, these works do not take advantage of the ease of use of the global shared memory in UPC as the matrices and vectors are initially distributed in the private memory of the threads, in the same way that in message-passing numerical libraries.

The main contribution of our work is to provide, for the first time, the design and implementation of a parallel BLAS library for a pure PGAS language that directly exploits the characteristics of its memory model. The starting point of this work is the one presented in [34], where a preliminary design and implementation of a dense PGAS-oriented BLAS library was shown. The design of the library has greatly evolved in order to:

1. Simplify and improve the interface and thus facilitate the use of the library. In the preliminary design the library had two types of functions per numerical routine: one function that worked with the data replicated in the private memory of the threads and another one that used shared memory. This duplicity of functions increased the difficulty of use of the library. As will be shown in the next sections, the current design, based on shared arrays, gathers in a single version the benefits (programmability and performance) provided separately by the previous function types.

2. Avoid some memory overheads due to the replication of matrices and vectors, especially important in BLAS3 routines. The new implementation avoids these overheads by using an algorithm by blocks obtaining even better performance. Besides, this algorithm adapts its behavior to the characteristics of the machine, being suitable for different architectures.

3. Support consecutive calls to the same or different parallel BLAS routines (a common case in practice) without needing to redistribute the data among the threads. With the previous design a data redistribution was necessary between two consecutive calls. Now the distribution of the data can be reused, avoiding the performance overhead due to redistributions.

4. Allow to work with submatrices, which is especially useful when using the BLAS routines in iterative algorithms.

5. Include support for complex datatypes, widely extended in numerical codes.

6. Include new optimization techniques that increase the performance of the library. Some of these techniques take advantage of the knowledge of some characteristics of the machine on which the library is installed so the UPCBLAS routines can adapt themselves to different systems.

3 Background: Overview of the Memory Model in UPC

This section will explain the most important features of the memory model in UPC which were taken into account to design the interface of the library, determine the most appropriate data distributions among threads and implement the numerical routines. Besides, an overview of the different types of pointers present in the language is also provided as their behavior is the basis for some of the optimization techniques included in UPCBLAS.

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, as shown in Figure 1. 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. This memory specification combines the advantages of both the shared and distributed programming models. On the one hand, the global shared memory space facilitates the development of parallel codes, allowing all threads to directly read and write remote data without explicitly notifying the owner. On the other hand, the performance of the codes can be increased by taking into account data affinity. 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,

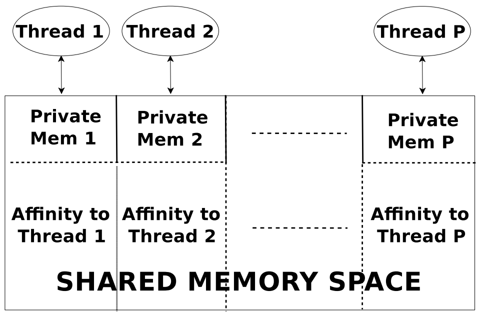


Figure 1: Memory model in UPC

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 the thread $\lfloor \frac{i}{BLOCK\_FACTOR} \rfloor mod(THREADS)$, being $THREADS$ the total number of threads in the UPC execution. If `BLOCK_FACTOR` is not specified, it is assumed to be 1 (i.e. cyclic distribution).

As an extension of the C language, UPC provides functionality to access memory through pointers. Due to the two types of memory available in the language, several types of pointers arise:

- Private pointers (*from private to private*). They are only available for the thread that stores them in its private memory and can reference addresses in the same private memory or in the part of the shared memory with affinity to the owner. Their syntax is the same of standard C pointers: `type *p`
- Private pointers to shared memory (*from private to shared*). They are only available for the thread that stores them in its private memory, but can have access to any data in the shared space. They contain three fields in order to know their exact position in the shared space: the *thread* where the data is located, the *block* that contains the data and the *phase* (the location of the data within the block). Thus, when performing pointer arithmetic on a pointer-to-shared all three fields will be updated, making the operation slower than private pointer arithmetic. As in shared arrays, the `BLOCK_FACTOR` can be specified: `shared [BLOCK_FACTOR] type *p`
- Shared pointers (*from shared to shared*). They are stored in shared memory (and therefore accessible by all threads) and they can access any data in the shared memory. Their complexity is similar to private pointers to shared memory. They are defined as: `shared [BLOCK_FACTOR] type *shared p`
- Shared pointers to private memory (*from shared to private*). They are stored in shared memory and point to the private space. However, their use is not advisable and they are not available in some compiler implementations.

4 UPCBLAS Design

Parallel numerical libraries based on the message-passing paradigm force the user to distribute the elements of the input vectors and matrices among processes. Therefore, new structures to handle these distributed inputs need to be created and these structures are passed as parameters to the parallel BLAS functions. In UPC shared arrays implicitly distribute their elements among the parts of the shared memory with affinity to the different threads. Unlike message-passing based codes, which require the input data to be distributed in the local memory of each process, PGAS functions can simplify this data distribution by using shared arrays.

UPCBLAS contains a subset of representative BLAS routines. Table 1 lists all the implemented routines grouped in three levels: BLAS1 (vector-vector operations), BLAS2 (matrix-vector operations) and BLAS3 (matrix-matrix operations). A total of 56 different functions were implemented: 14 routines and 4 datatypes per routine.

Table 1: UPCBLAS routines. All the functions follow the naming convention: `upc_blas_Tblasname`, where "T" represents the data type (s=float; d=double; c=single precision complex; z=double precision complex) and `blasname` is the name of the routine in the sequential BLAS library

BLAS level	Tblasname	Action
BLAS1	Tcopy	Copies a vector
	Tswap	Swaps the elements of two vectors
	Tscal	Scales a vector by a scalar
	Taxpy	Updates a vector using another one: $y = \alpha * x + y$
	Tdot	Dot product between two vectors
	Tnrm2	Euclidean norm of a vector
	Tasum	Sums the absolute value of the elements of a vector
	iTamax	Finds the index with the maximum value in a vector
	iTamin	Finds the index with the minimum value in a vector
BLAS2	Tgemv	Matrix-vector product
	Tger	Outer product between two vectors
	Ttrsv	Solves a triangular system of equations
BLAS3	Tgemm	Matrix-matrix product
	Ttrsm	Solves a block of triangular systems of equations

All the routines return a local integer error value which refers only to each thread execution. In order to ensure that no error has happened in any thread, the global error checking must be made by the programmer using the local error values. This is a usual practice in parallel libraries to avoid unnecessary synchronization overheads.

4.1 BLAS1 Routines

In order to favor the adoption of UPCBLAS among PGAS programmers the syntax of these functions is similar to the standard collectives library [3]. For instance, the syntax of the single precision dot product is:

```
int upc_blas_sdot(int block_size, int size, shared void *x,
                  shared void *y, shared float *dst);
```

being `x` and `y` the source vectors of length `size`; `dst` the pointer to shared memory where the dot product result will be written; and `block_size` the block factor (see `BLOCK_FACTOR` in Section 3) of the source vectors. For performance reasons, the block factor must be the same for both vectors. This function treats pointers `x` and `y` as if they had type `shared [block_size] float[size]`.

An important design decision in UPCBLAS is that, looking for efficiency, only one `block_size` parameter to indicate the same block factor for both shared arrays is included. Figure 2 shows two scenarios for the dot product. When both vectors have the same `block_size`, all the pairs of elements that must be multiplied are stored in the shared memory with affinity to the same thread. Thus, each thread only has to perform its partial dot product in a sequential way and the final result is obtained through a reduction operation over all threads. However, in the second case, several remote accesses that affect performance are necessary so that each thread can obtain all the data needed in its partial dot product.

Finally, in order to be able to work with subvectors, `x` and `y` do not need to point to the first element

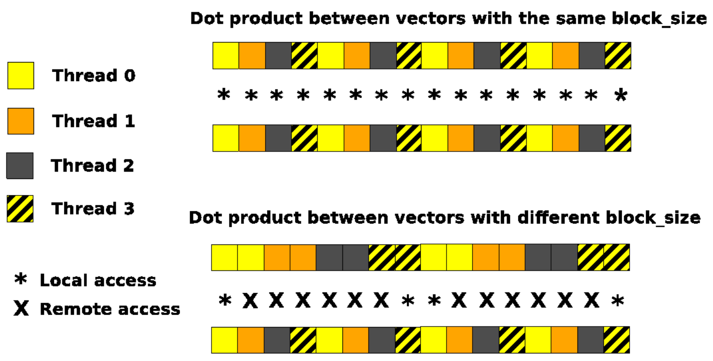

Dot product between vectors with the same block_size
Thread 0
Thread 1
Thread 2
Thread 3
Dot product between vectors with different block_size
* Local access
X Remote access
* * * * * * * * * * * * * * * *
* X X X X X X * * X X X X X X *

Figure 2: Remote and local accesses in `upc_blas_sdot` according to the block factor of the source vectors

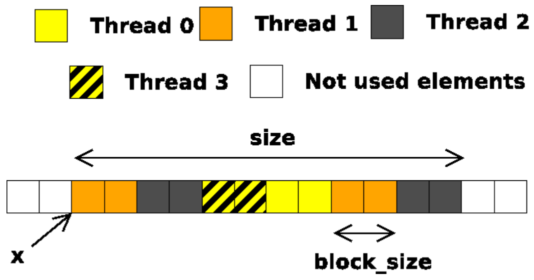


Figure 3: Meaning of the parameters of `upc_blas_sdot` when working with subvectors

of a shared array. Figure 3 illustrates an example for a subvector that is stored from position 2 to 13 of a shared array. The only restriction is that the subvector must start in the first position of a block (i.e. its phase must be 0). This is also a restriction of the standard UPC libraries (e.g. the collectives library) and the natural way to declare and allocate shared arrays.

4.2 BLAS2 Routines

Shared matrices in UPC can only be distributed in one dimension as the UPC syntax does not allow multidimensional layouts. The definition of multidimensional block factors has been proposed in [35], although currently there are no plans to include this extension in the language specification. Therefore, all the UPCBLAS routines rely on the one dimensional data distribution present in the standard. An additional parameter (`dimmDist`) is needed in the routines to indicate the dimension used for the distribution of the matrix. For instance, the UPCBLAS routine for the single precision matrix-vector product ($y = \alpha * A * x + \beta * y$) is:

```
int upc_blas_sgemv(UPCBLAS_DIMMDIST dimmDist, int block_size,
                   int sec_block_size, UPCBLAS_TRANSPOSE transpose,
                   int m, int n, float alpha, shared void *A,
                   int lda, shared void *x, float beta,
                   shared void *y);
```

being `A` and `x` the source matrix and vector, respectively; `y` the result vector; `transpose` an enumerated value to indicate whether matrix `A` is transposed; `m` and `n` the number of rows and columns of the matrix; `alpha` and `beta` the scale factors for `A` and `y`, respectively; and `dimmDist` another enumerated value to indicate if the source matrix is distributed by rows or columns. `lda` is a parameter inherited from the sequential BLAS library to work with submatrices. It expresses the memory distance between two elements in the same column and in consecutive rows of the submatrix. Figure 4 shows an example with the row and column distributions when `A` is a submatrix that discards two rows and two columns of the global array. Thanks to using arrays stored in shared memory and pointing directly to the first element of the submatrix, the `lda` parameter is enough to specify all the information to work with submatrices, as in sequential BLAS routines. Therefore, the syntax of the routines is simpler than in message-passing based numerical libraries where the first row and column of the submatrix must be explicitly specified through additional parameters. Similarly to the UPC BLAS1 routines, the only restriction is that the submatrix must start at the first row/column of a block in the row/column distribution. This approach is followed in all the UPC BLAS2 and BLAS3 routines to work with submatrices.

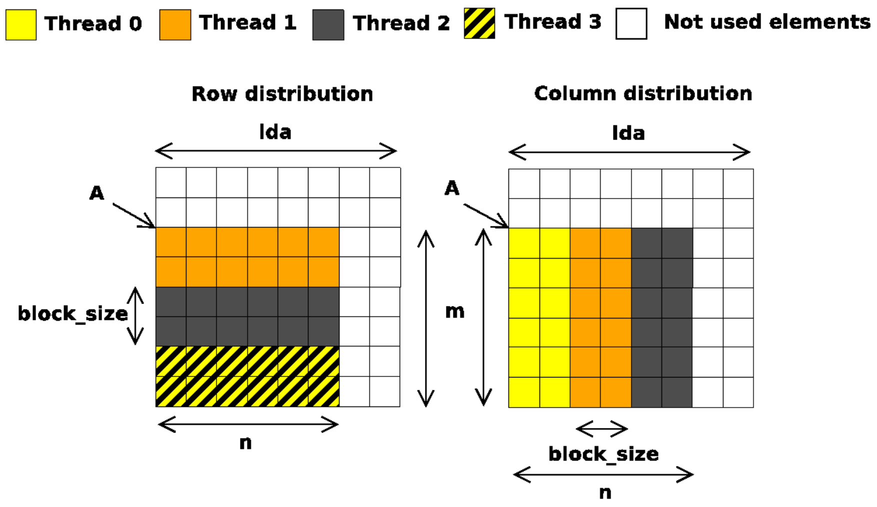


Figure 4: Meaning of the parameters of `upc_blas_sgemv` when working with a submatrix

The meaning of the `block_size` and `sec_block_size` parameters depends on the `dimmDist` value:

- If the source matrix `A` is distributed by rows (`dimmDist=upcblas_rowDist`), `block_size` is the number of consecutive rows with affinity to the same thread and `sec_block_size` the block factor related to the source vector `x`. For instance, in the non-transpose case, this function treats pointers:
  - `A` as `shared[block_size*lda] float[m*lda]`
  - `x` as `shared[sec_block_size] float[n]`
  - `y` as `shared[block_size] float[m]`
- If the source matrix is distributed by columns (`dimmDist=upcblas_colDist`), `block_size` is the number of consecutive columns with affinity to the same thread and `sec_block_size` the block factor related to the result vector `y`:
  - `A`: `shared[block_size] float[m*lda]`
  - `x`: `shared[block_size] float[n]`
  - `y`: `shared[sec_block_size] float[m]`

Figure 5 illustrates the behavior of the matrix-vector product when `A` is distributed by rows (in this example `block_size=2`). In order to exploit data locality as much as possible each thread only accesses the rows of the matrix with affinity to that thread. Then, by applying a sequential partial matrix-vector product with these rows and all the elements of `x`, each thread calculates a partial result that corresponds with its rows of `A`. Thus, if the distribution of the result vector matches the distribution of the matrix, all the partial results can be copied to their correct final positions working only with local memory. This is the reason why in the row case the parameter `block_size` indicates not only the distribution of the matrix, but also the distribution of the result vector. Thus, users are forced to declare `y` with a block factor equal to `block_size` in order to guarantee always a good performance. As all the elements of `x` must be used by all threads, its block factor does not need to be linked to the distribution of the matrix, and it is indicated through the `sec_block_size` parameter.

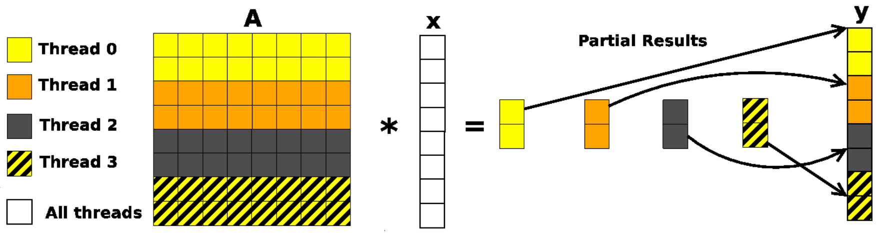


Figure 5: Matrix-vector product (*Tgemv*) using a row distribution for the matrix

Figure 6 shows the behavior of the routine with a column distribution. In this case the source vector `x` must have always the same distribution (`block_size`) as the matrix and the distribution of the result vector `y` is passed through the `sec_block_size` parameter. In order to compute the i^{th} element of the result, the i^{th} values of all partial results must be added. These additions need reduction operations involving all UPC threads, so their performance is usually poor.

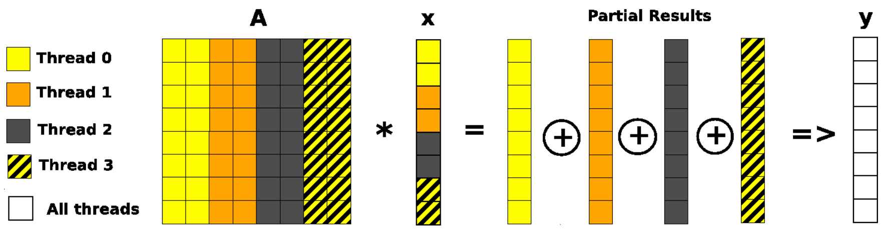


Figure 6: Matrix-vector product (*Tgemv*) using a column distribution for the matrix

The approach to parallelize the outer product (*Tger*) within UPCBLAS is very similar to the matrix-vector product. However, the routine to solve a triangular system of equations $M * x = b$ (*Ttrsv*) is a special case because there are a lot of data dependencies in the internal algorithm. An exhaustive study of the different parallel alternatives for this algorithm can be found in [36]. In the BLAS interface vector `b` is always overwritten by the solution vector `x`, so both are represented by the same parameter. According to this assumption, the syntax of the UPC BLAS2 triangular solver for single precision is:

```
upc_blas_strsv(UPCBLAS_DIMMDIST dimmDist, int block_size,
               UPCBLAS_UPLO uplo, UPCBLAS_TRANSPOSE transpose,
               UPCBLAS_DIAG diag, int n, shared void *M, int ldm,
               shared void *x);
```

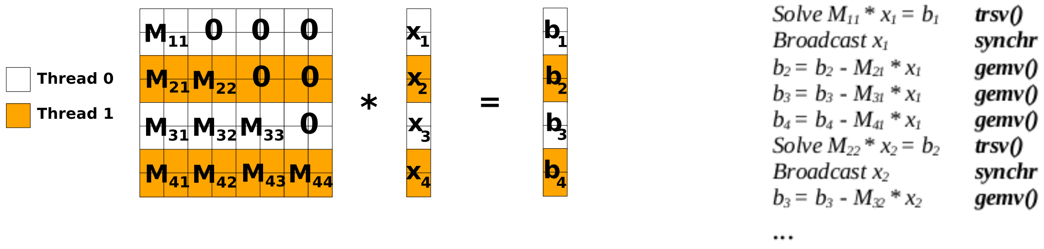
being `nxn` the size of the triangular matrix `M` and `n` the length of `x`. The enumerated values `uplo`, `transpose` and `diag` are included to determine the characteristics of `M` (upper/lower triangular, transpose/non-transpose, elements of the diagonal equal to one or not). In this routine, all the distributions are specified by `block_size` and the vector and the matrix must be stored in shared arrays with the following syntax:

– `M`: `shared[block_size*ldm] float[n*ldm]` if row distribution

– M: `shared[block_size] float[n*ldm]` if column distribution

– x: `shared[block_size] float[n]` in both cases

Figure 7 shows an example of the distribution by rows of 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 7 shows the parallel algorithm for this example. Once one thread computes its part of the solution (output of the sequential `trsv` routine), it is broadcast to all threads so that they can update their local parts of b with the sequential product (`gemv`). Thanks to specifying the distribution of both `M` and `x` with the same parameter (`block_size`), all sequential `trsv` and `gemv` computations can be performed without any communication except the broadcast. Note that all operations between two synchronizations (broadcasts) can be performed in parallel.



Thread 0
Thread 1
M11 0 0 0
M21 M22 0 0
M31 M32 M33 0
M41 M42 M43 M44
*
x1 x2 x3 x4
=
b1 b2 b3 b4
Solve M11 * x1 = b1 trsv()
Broadcast x1 synchr
b2 = b2 - M21 * x1 gemv()
b3 = b3 - M31 * x1 gemv()
b4 = b4 - M41 * x1 gemv()
Solve M22 * x2 = b2 trsv()
Broadcast x2 synchr
b3 = b3 - M32 * x2 gemv()
...

Figure 7: Matrix distribution and algorithm for the parallel BLAS2 triangular solver (*Ttrsv*)

The column distribution would involve a nearly sequential algorithm with poor performance due to the characteristics of its dependencies. However, it is also available in the *Ttrsv* routine to allow a distribution reuse just in case the source matrix uses that distribution in other UPCBLAS routines within the same application.

4.3 BLAS3 Routines

In the BLAS3 routines, as there is more than one matrix, the number of possible combinations of distributions of the matrices grows. In the design of UPCBLAS programmability is a must and thus, in order to simplify the understanding and use of the UPC BLAS3 routines, the parameter `dimmDist` makes always reference to the result matrix. Moreover, this choice allows to reuse the output data as input matrices in consecutive calls to UPCBLAS routines.

The interface of the UPCBLAS routine for a single precision matrix-matrix product ($C = \alpha * A * B + \beta * C$) is:

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

being `mxk`, `kxn` and `mxn` the sizes of `A`, `B` and `C`, respectively. `block_size` means the number of consecutive rows or columns of `C` (depending on the `dimmDist` value) with affinity to the same thread. Besides, `sec_block_size` is related to `B` in the row distribution and to `A` in the column one. The meaning of the rest of parameters is similar to the BLAS2 routine explained in the previous section. Thus, UPCBLAS `sgemm` with the row distribution treats pointers:

– `A` as `shared[block_size*lda] float[m*lda]`

– `B` as `shared[sec_block_size] float[k*ldb]`

– `C` as `shared[block_size*ldc] float[m*ldc]`

If the column distribution is used, the matrices must be stored in arrays declared as follows:

– `A`: `shared[sec_block_size] float[m*lda]`

– `B`: `shared[block_size] float[k*ldb]`

– `C`: `shared[block_size] float[m*ldc]`

Figure 8 shows an example for the row distribution of the matrix-matrix product. As `block_size` is related to the result matrix `C`, 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`. So, as in the equivalent distribution of the BLAS2 routine (*Tgemv*), `block_size` is related to `C` and `A`, and the distribution of `B` is determined by `sec_block_size`.

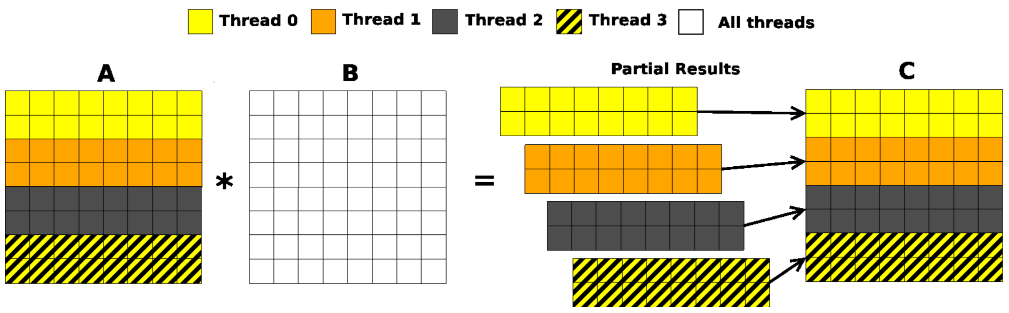


Figure 8: Matrix-matrix product (*Tgemm*) using a row distribution for matrix `C`

Figure 9 shows the same example when matrix `C` is distributed by columns. In order to perform the partial sequential matrix-matrix product each thread needs the whole matrix `A` but only the same columns of `B` than those of `C` with affinity to that thread. Thus, `block_size` also defines the distribution of `B` and `sec_block_size` is related to `A`. Unlike the column distribution of the BLAS2 routine (see Figure 6), no reductions are necessary in this case, avoiding the associated overhead at the end of the routine.

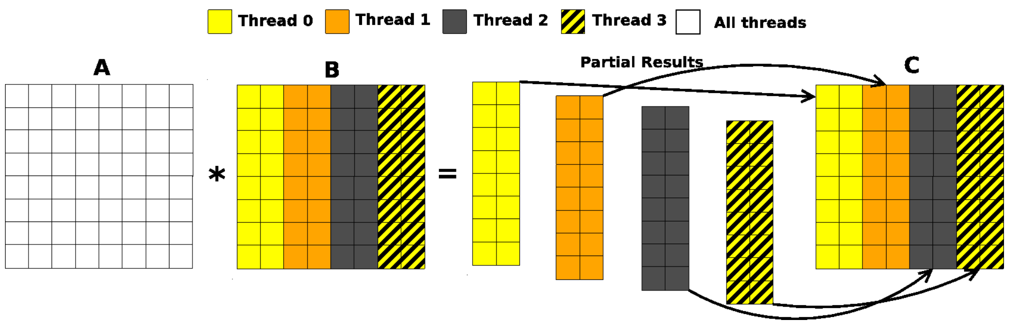


Figure 9: Matrix-matrix product (*Tgemm*) using a column distribution for matrix `C`

Regarding the BLAS3 triangular solver ($M * X = \alpha * B$, with `B` overwritten by the result matrix `X`), the parameters are quite similar to those of the BLAS2 counterpart, only changing the vectors by matrices and adding a new enumerated parameter (`side`) to indicate if the triangular matrix `M` is in the left or in the right part of the operation. `M` is `mxm` if it is left-sided or `nxn` if right-sided, and `X` is always `mxn`. The syntax for single precision is:

```
upc_blas_strsm(UPCBLAS_DIMMDIST dimmDist, int block_size,
               int sec_block_size, UPCBLAS_SIDE side,
               UPCBLAS_UPLO uplo, UPCBLAS_TRANSPOSE transpose,
               UPCBLAS_DIAG diag, int m, int n, float alpha,
               shared void *M, int ldm, shared void *X, int ldx);
```

As in the matrix-matrix product, the distribution specified by `dimmDist` and `block_size` is always related to the result matrix. In this case the choice between row or column distribution leads to apply a different algorithm. If the result matrix `X` is distributed by rows, the routine performs this solver with an algorithm similar to the one shown at the right of Figure 7, only replacing the sequential BLAS2 routines *Tgemv* and *Ttrsv* by their equivalent BLAS3 *Tgemm* and *Ttrsm*, respectively. Thus, the triangular matrix is forced to have the same distribution than the result one:

– M: `shared[block_size*ldm] float[m*ldm]`

– X: `shared[block_size*ldx] float[m*ldx]`

However, if the result matrix `X` is distributed by columns, a similar approach to the column distribution of the matrix-matrix product, with independent sequential computations, is applied. This approach treats pointers as follows:

– M: `shared[sec_block_size] float[m*ldm]`

– X: `shared[block_size] float[m*ldx]`

Figure 10 shows an example of this column distribution. As the source matrix `B` is overwritten by the result matrix `X`, they are represented by the same pointer and thus they have the same block factor. Therefore, each thread has access to all the elements of the triangular matrix and applies a sequential *Ttrsm* routine to the columns of `B` and `X` with affinity to it. This approach improves performance by avoiding the dependencies present in the row distribution.

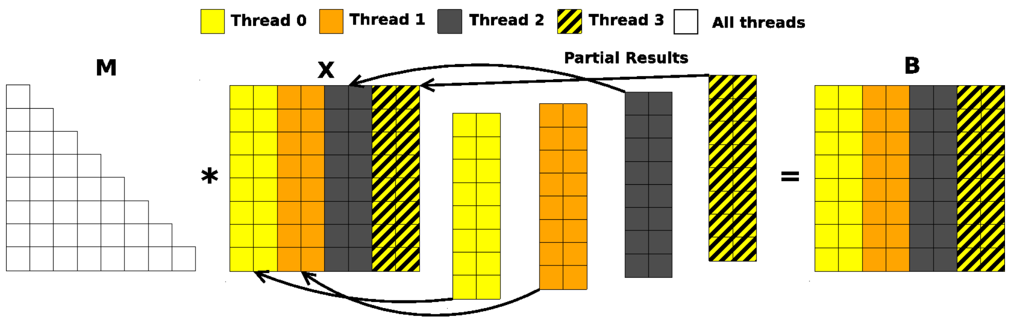


Figure 10: BLAS3 triangular solver (*Ttrsm*) using a column distribution

5 UPCBLAS Implementation

In order to increase the efficiency of UPCBLAS, a set of optimization techniques have been applied in the implementation of the routines to achieve the best possible performance.

5.1 UPC Optimization Techniques

There is a number of known optimization techniques that improve the efficiency and performance of the UPC codes [4, 7]. The following optimizations have been applied to the implementation of the UPCBLAS routines whenever possible:

- Space privatization: As explained in Section 3, working with shared pointers is slower than with private ones. Experimental measurements in [4] and [37] 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()` and `upc_memput()` functions on remote bulks of data required by a thread. For instance, the vector `x` in Figure 5 is replicated in all threads using bulk copies of `sec_block_size` elements.

- Use of phaseless pointers: Many UPC compilers (including Berkeley UPC [38]) implement an optimization for the common 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 internal shared arrays of the UPCBLAS routines are declared with block factor of zero in order to take advantage of this optimization.

5.2 Efficient Array-Based Reduction in UPC

As explained in Section 4.2, the column distribution for *Tgemv* needs a reduction operation for each element in the result vector (see Figure 6). The UPC standard collectives library [3] does not include a collective function to perform reduction operations on arrays as, for instance, MPI does. A solution could be the use of the `upc_all_reduce` function once per element of the destination array, however this method is quite inefficient. The approach followed in UPCBLAS to perform these array-based reductions consists of copying all the elements to a thread, using this thread to perform the operation and distributing the results again among the threads. We have proved experimentally that this approach is faster.

5.3 Efficient Broadcast Communication Model

As in the PGAS programming model any thread may directly read or write data located on a remote processor, 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 network contention 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 is therefore the communication model used by all the broadcast operations in our parallel routines (see, for instance, the BLAS2 triangular solver in Figure 7).

5.4 On-Demand Copies in BLAS3 Routines

In many of the UPCBLAS routines all threads have to access all the elements of a distributed matrix or vector (see, for instance, vector `x` in Figure 5 or matrix `B` in Figure 8). Thus, all threads must copy remote data to local memory before performing their sequential part of the computations. Auxiliary buffers in local memory are required to store the vector or the matrix.

In the UPC BLAS2 routines the vector is stored completely in private memory and then all the sequential computations related to a thread are performed in one go. However, copying the whole matrices in the BLAS3 routines could involve important memory overheads because the buffer could need to allocate a huge amount of private memory. Moreover, performance would be affected because all threads should wait to copy all these data before starting the sequential computations. Besides, they would access a large amount of remote data at the same time, which could lead to network contention in many systems.

In order to overcome these drawbacks the UPC BLAS3 routines are implemented using what we have called an *on-demand copies* technique. The matrix is copied by blocks into the auxiliary buffer, decreasing the memory requirements. Once one block is copied, the sequential computation that uses that part of the matrix starts. Besides, computations and communications are overlapped by using split-phase barriers to increase the scalability of the routine.

The size of the internal blocks is very important in this approach. On the one hand, if the blocks are too large, the memory and performance problems explained before would not be solved. On the other hand, if they are too small the performance of the communications would decrease because more calls to `upc_memget()` would be needed, each of them with less aggregation of remote accesses; moreover, the partial sequential computations would be almost negligible to be overlapped with communications.

The best size can be different depending on the characteristics of the system where the library is executed, specially on the communication network. UPCBLAS uses Servet [39, 40], a portable benchmark suite that allows to obtain the most relevant hardware parameters of clusters of multicores, to automatically select the best size for the auxiliary buffer. Among other characteristics, Servet obtains, for each communication layer in the system, the bandwidth as a function of the message size. This information is used to determine the message size for which the bandwidth stops increasing. This message size is considered as the ideal block size to use in the on-demand copies technique. For instance, Figure 11 shows the bandwidths of the intra-node and inter-node communications in the supercomputer that will be used for the experimental evaluation in Section 6. The size selected for the auxiliary buffer in this system is 4MB because it is the point where both bandwidths no longer show an increase.

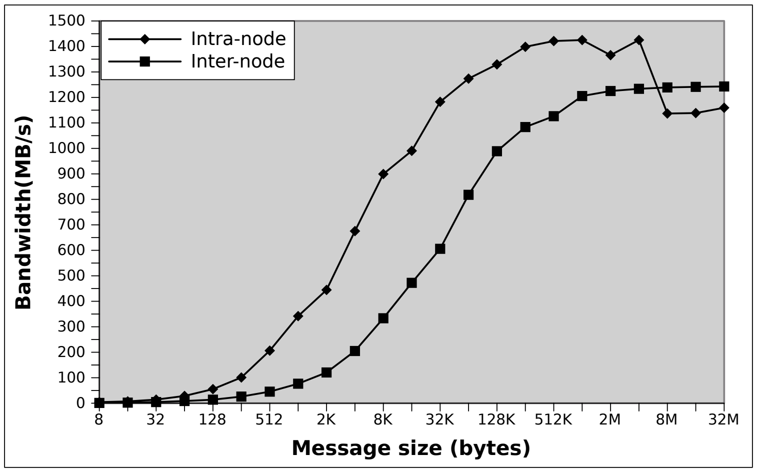


Figure 11: Example of the information provided by Servet about the communication bandwidth as a function of the message size

In order to make the system parameters available to UPCBLAS, Servet must have been installed and executed before the installation of the numerical library. Servet saves the relevant hardware parameters into a text file and provides an API to obtain the information from this file. Then, when a routine requires information about the hardware characteristics it resorts to calling the API of Servet instead of running any benchmark. Thus, the performance overhead caused by Servet is almost negligible in UPCBLAS.

5.5 Efficient Mapping of UPC Threads

Nowadays, most of parallel numerical codes run on clusters of multicores. Depending on the number of UPC threads required by the user and the total number of nodes and cores available in the system, there are usually many different ways to assign the threads to cores. The information about the hardware characteristics of the underlying system provided by Servet is used by the UPCBLAS library to automatically map the threads to certain cores in order to avoid communication and memory access bottlenecks. The applied mapping is based on the following information provided by Servet:

- Communication performance characterization: Servet is able to obtain the communication overhead among cores, as well as the communication hierarchy of the system based on these data.
- Effective shared memory access performance: The concurrent access by multiple threads to main memory can represent an important bottleneck (for instance if they share the bus to memory). Servet is able to characterize the shared memory access performance taking into account the placement of the threads in particular cores.

The mapping policy maps one UPC thread per core following the algorithm described in [41]. In order to apply this algorithm the routines must be characterized as memory bound or communication intensive. As all the threads involved in the UPCBLAS routines are continuously accessing memory (with much more local than remote accesses), the library is characterized as memory bound. This characterization leads to use a mapping policy where shared memory access overheads have a higher impact on performance than communication costs. Thus, the assignment of threads to cores always tries to maximize memory access throughput and, only when possible, minimize communication times.

An example of parallel system is shown in Figure 12. This system consists of two nodes, each with two dual-core processors. If cores in the same node access memory at the same time, their memory bandwidth would be probably penalized because they share the same memory module. Besides, threads in the same dual-core processor share the same memory bus, which could lead to even more conflicts when accessing memory concurrently. Using Servet, if a UPCBLAS routine were run using two threads in the multicore cluster depicted in the figure, they would be placed in cores in different nodes to avoid memory access overhead. In an execution with four threads it would not be possible to avoid all memory overheads, but UPCBLAS would minimize them by placing each thread in one core of different processors (for instance, cores 0, 2, 4 and 6). Experimental results in [41] show the efficiency of this approach.

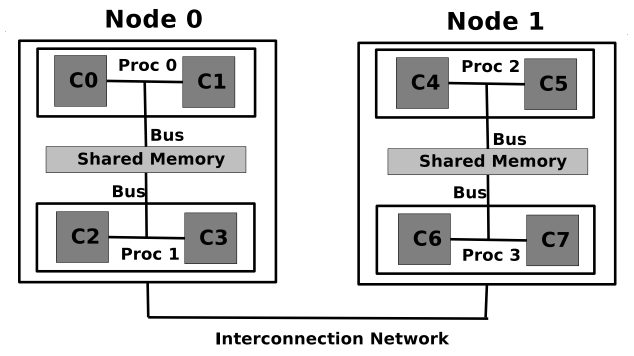


Figure 12: Example of a multicore cluster (two nodes, each with two dual-core processors)

5.6 Underlying Efficient Sequential Numerical Libraries

Besides applying the optimizations described in the previous subsections to improve the parallel behavior of the code it is necessary to rely on efficient sequential numerical libraries to obtain good performance. The UPCBLAS parallel functions call internally BLAS routines to perform the sequential computations in each thread. These calls can be linked to very optimized libraries such as Intel MKL. Just in case a numerical library is not available in the system, UPCBLAS provides an own sequential implementation using ANSI C. Thus, the UPC routines act as an interface to distribute data and synchronize the calls to the sequential ones in an efficient and transparent way.

6 Experimental Evaluation

In order to evaluate the performance of UPCBLAS runtime tests were performed on the Finis Terrae supercomputer [42] at the Galicia Supercomputing Center (CESGA). This system 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 4 dual-core processors, grouped in pairs that share the memory bus (8 cores and 64 GB of shared memory per cell). As for software, UPCBLAS was compiled using Berkeley UPC 2.12.1 [9] and linked to the Intel Math Kernel Library (MKL) version 10.1 [21], a highly tuned BLAS library for Itanium cores. The intra-node and inter-node communications are performed through shared memory and GASNet over InfiniBand, respectively.

In this supercomputer a memory overhead is caused when several cores that share the bus access memory at the same time. Thus, the automatic mapping policy available in the UPCBLAS routines (see Section 5.5) assigns two threads per cell in order to minimize the conflicts in the access to memory. Specifically, UPCBLAS maps threads to cores that do not share the bus to memory. Although the experiments do not use all cores/nodes available in the system, the performance evaluation was carried out in a real environment with almost 100% of the remaining cores running jobs of other users.

In this section different performance measures of representative UPCBLAS routines are shown, specifically: the dot (*Tdot*), matrix-vector (*Tgemv*) and matrix-matrix (*Tgemm*) products, as well as the BLAS3 triangular solver (*Ttrsm*). The performance of the four routines was measured for single precision using different distributions. Some of these routines were also used to evaluate the performance improvement obtained by some of the optimization techniques explained in the previous section (e.g. mapping policy and on-demand copies).

A comparison with the message-passing based implementation of the PBLAS routines [15,16] included in MKL 10.1 (using Intel MPI v3.2.1) is also provided for the BLAS2 and BLAS3 levels. The PBLAS routines were tested using different data distributions (by rows, by columns and two-dimensional distributions, using different block sizes), but the results presented in this section are those of the distribution that achieves the best execution time for each PBLAS routine and number of threads. Besides, in order to provide a fair comparison, the same mapping policy obtained by Servet within UPCBLAS has been manually applied to the PBLAS processes in order to avoid the overheads caused by the concurrent memory accesses.

The sizes of the vectors and matrices used in the experiments are the largest ones that can be allocated in the memory available for one core (in order to calculate speedups). All the speedups (both for the UPCBLAS and PBLAS routines) were obtained using as reference the execution time of the sequential MKL library (the fastest one). Although the variability of the performance obtained in different executions was almost negligible (always few milliseconds), the experiments with BLAS1 and BLAS2 routines were repeated 20 times both for UPCBLAS and PBLAS. The results shown in the following graphs are always the best ones for each library and scenario, thus discarding any source of variability. As the execution times for the BLAS3 routines are much longer, the influence of the variability (also some milliseconds) on the calculation of the speedups is even less significant. Thus, in these cases the experiments were performed only 3 times.

Figure 13 and Table 2 show the speedups and execution times for the BLAS1 dot product. Experimental results have been obtained using block, cyclic (i.e. `block_size` = 1) and block-cyclic distributions (with different `block_size` values) in order to analyze the behavior of the routine in several scenarios. The block distribution cannot be used with large vectors because the shared array would need a `block_size` higher than the largest allowed in Berkeley UPC (1024×10^3 elements). For illustrative purposes only the "extreme" distributions are presented throughout this section (i.e. cyclic and block-cyclic with the largest `block_size` allowed by the compiler). As we can see in Figure 13, the scalability is reasonable for both distributions taking into account that the execution times are very short (in the order of milliseconds). An analysis of the results obtained with more different values for `block_size` has demonstrated that it has no influence on the performance of this routine. Therefore, UPCBLAS users can achieve the best performance independently of the distribution (i.e. `block_size`) to be used in the application. In general this also applies to BLAS2 and BLAS3 routines as will be shown later.

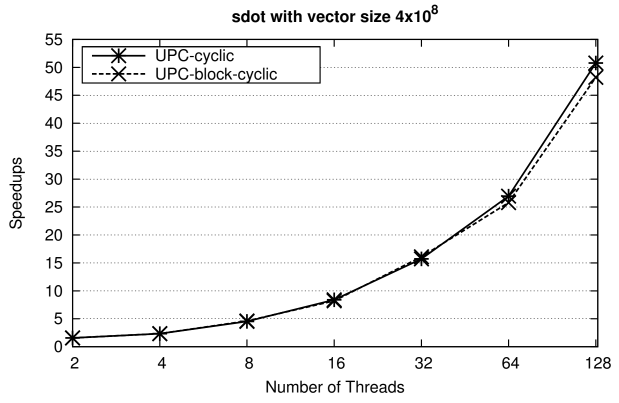


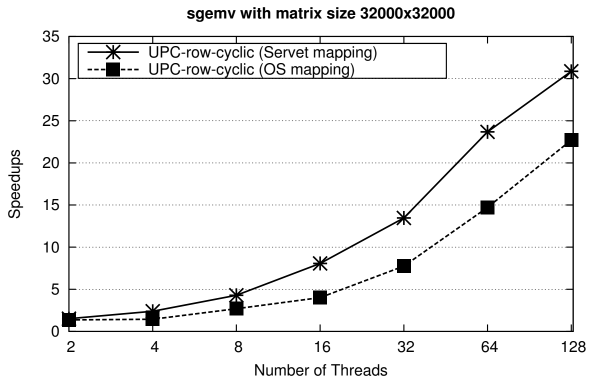
Figure 13: Speedups of the single precision dot product (*sdot*)

Table 2: Execution times (in milliseconds) of the single precision dot product (*sdot*)

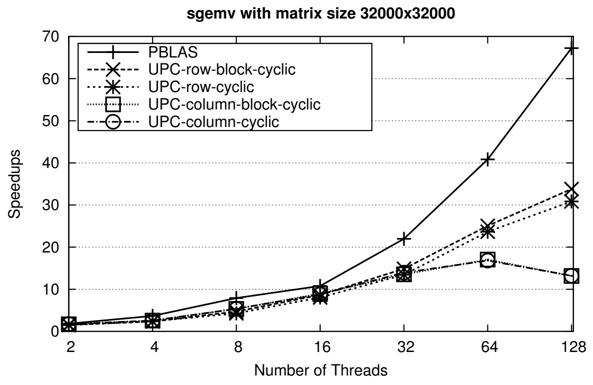
sdot with size 4×10^8 (ms)		
MKL Seq	582.91	
# THREADS ↓	Cyclic	Block-Cyclic
1	670.56	
2	369.31	373.02
4	249.32	249.77
8	128.49	126.65
16	69.42	70.97
32	37.10	36.21
64	21.63	22.60
128	11.48	12.09

Figure 14(a) shows, for illustrative purposes, the speedups for the single precision UPCBLAS matrix-vector product with the cyclic distribution of the matrix by rows using two different mapping policies of threads to cores: the one selected by Servet (as explained in Section 5.5) and the one provided by default by the OS. The Servet mapping significantly outperforms the OS one.e Although only one example is shown, we have checked that the improvement is similar in all routines for all distributions. Therefore, the Servet mapping policy was used in all experiments, either automatically in the UPCBLAS library or manually in PBLAS.

Figure 14(b) and Table 3 show the speedups and execution times for the matrix-vector product, using in UPCBLAS the cyclic and the block-cyclic distributions (with the largest possible `block_size`) by rows and columns. As expected, the reduction operation at the end of the routine when using the column distribution (see Section 4.2 and Figure 6) decreases performance and even leads the function to stop scaling for 128 threads. Moreover, these tests demonstrate that, as in the BLAS1 case, the efficiency of this UPCBLAS routine does not depend on the value of `block_size`. The speedups of `sgemv` for non-square matrices were also measured, but they are not shown as they are similar to those of Figure 14(b) using the same matrix size (the same occurs for UPC BLAS3 routines).



(a) UPCBLAS with the mapping policies provided by Servet and the OS



(b) UPCBLAS with different distributions and PBLAS

Figure 14: Speedups of the single precision matrix-vector product (*sgemv*)

Table 3: Execution times (in milliseconds) of the single precision matrix-vector product (*sgemv*)

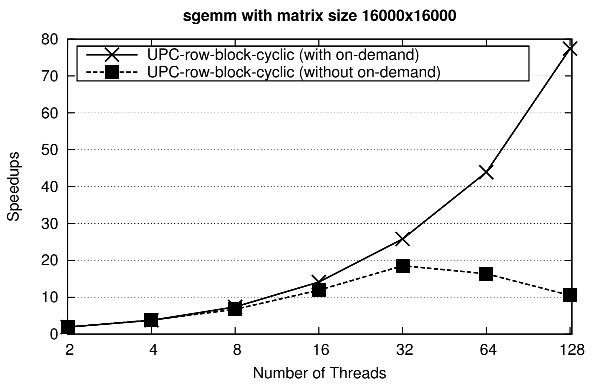
sgemv with matrix size 32000x32000 (ms)					
MKL Seq	1051.45				
# THREADS ↓	UPC Row		UPC Column		PBLAS
	Cyclic	Block-Cyclic	Cyclic	Block-Cyclic	
1	1217.66				1059.87
2	697.04	665.59	609.44	615.11	574.56
4	440.04	441.94	408.26	408.42	285,72
8	244.95	223.74	196.12	197.10	132.75
16	130.35	121.58	119.42	116.98	97.36
32	78.12	70.68	75.38	77.52	47.84
64	44.41	41.97	62.55	61.59	25.75
128	34.07	31.11	79.51	79.81	15.64

The best speedups for the PBLAS function were obtained with a two-dimensional distribution, being significantly higher than the UPCBLAS ones. However, looking only at the execution times, the UPC version with the row distribution is not much slower than the PBLAS routine.

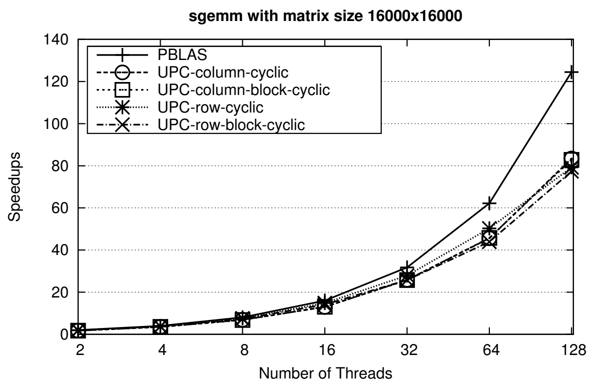
To assess the impact of the on-demand copies technique explained in Section 5.4, Figure 15(a) shows the speedups of the matrix-matrix product with the block-cyclic distribution by rows using the technique with a 4MB buffer and without it (i.e. replicating the whole matrix `B` at the beginning of the function). This experiment proves that the on-demand copies technique significantly improves performance as the number of threads increases when using an appropriate buffer size. This size is automatically set by Servet at installation time.

Figure 15(b) and Table 4 show the performance evaluation of the *sgemm* routine applying the on-demand copies in all the UPCBLAS experiments. As in the BLAS2 routine, the value of `block_size` hardly affects performance. Due to distributing always the result matrix and thus avoiding collective reductions (see

Section 4.3), the row and column distributions of the BLAS3 product do not show significant performance differences. Moreover, thanks to the on-demand copies technique, the scalability is high in all cases. As for the PBLAS matrix-matrix product the best results are obtained with a two-dimensional distribution that achieves a near linear speedup.



(a) UPCBLAS with and without on-demand copies



(b) UPCBLAS with different distributions and PBLAS

Figure 15: Speedups of the single precision matrix-matrix product (*sgemm*)

Table 4: Execution times (in seconds) of the single precision matrix-matrix product (*sgemm*)

sgemm with matrix size 16000x16000 (s)					
MKL Seq	1304.44				
# THREADS ↓	UPC Row		UPC Column		PBLAS
	Cyclic	Block-Cyclic	Cyclic	Block-Cyclic	
1	1347.25				1305.33
2	689.22	687.65	742.51	740.98	652.01
4	348.09	349.65	368.78	370.07	324.35
8	172.41	177.18	189.50	192.48	162.29
16	87.72	92.35	100.12	99.85	81.61
32	46.73	50.60	50.49	50.69	41.07
64	25.93	29.71	28.60	28.52	20.98
128	16.46	16.86	15.63	15.75	10.48

The experimental results for the BLAS3 triangular solver are shown in Figure 16 and Table 5. The behavior of the column distribution is quite similar to the matrix-matrix product, where the `block_size` has not a significant influence on performance. However, in the row distribution the `block_size` has a great impact on the speedups 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, so the cyclic distribution obviously becomes a wrong option. Anyway, the routine with the row distribution scales only up to 32 threads, even using the maximum possible size for the blocks and the pull model in the broadcast operations (see Section 5.3). After an analysis of the results, the overhead of the broadcasts in the internal algorithm were determined as the reason of the performance decrease, as explained in Section 4.3.

Regarding the performance of the PBLAS version, the lowest executions times (those shown in Table 5) are obtained using a block-cyclic distribution by rows, with different optimal sizes for the blocks depending on the number of processes. As can be seen, PBLAS only outperforms the row version of UPCBLAS. However, the algorithm used by the PBLAS routine does not avoid the data dependencies as in the UPCBLAS column

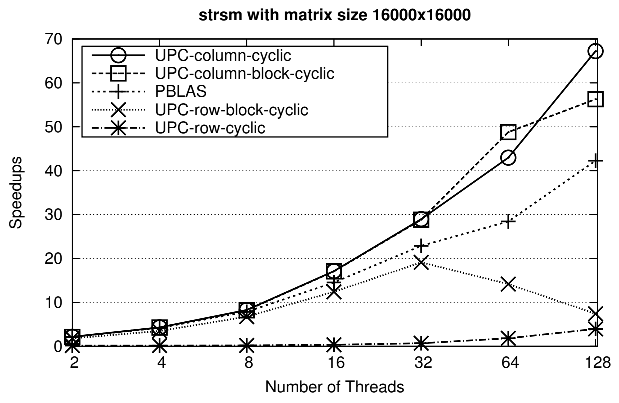


Figure 16: Speedups of the single precision BLAS3 triangular solver (*strsm*)

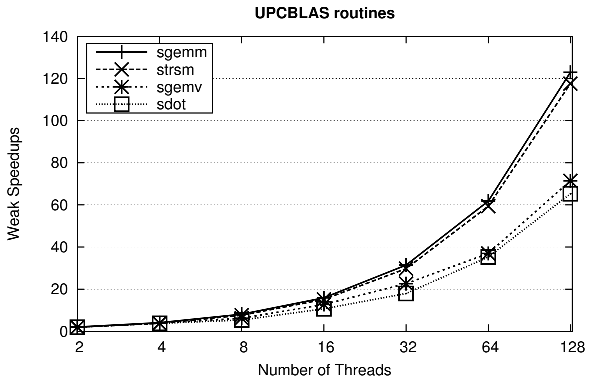
Table 5: Execution times (in seconds) of the single precision BLAS3 triangular solver (*strsm*)

strsm with matrix size 16000x16000 (s)					
MKL Seq	790.84				
# THREADS	UPC Row		UPC Column		PBLAS
↓	Cyclic	Block-Cyclic	Cyclic	Block-Cyclic	
1	845.40				795.35
2	5216.61	443.35	364.63	372.92	372.61
4	4778.02	228.05	184.58	185.29	188.64
8	4517.64	117.29	95.58	96.46	100.82
16	2332.84	63.81	46.20	46.37	54.39
32	1152.01	41.40	27.30	27.43	34.55
64	437.71	55.83	18.41	16.21	27.81
128	199.88	107.09	11.76	14.05	18.70

distribution, where the sequential computations are independent and the only overhead is due to gathering the triangular matrix in all threads through on-demand copies (see Figure 10).

In all the previous experiments the efficiency of the UPCBLAS routines decreases when increasing the number of threads because the execution times are significantly lower and the overhead of the communications becomes more important. However, UPCBLAS would obtain better performance when working with larger problems. Figure 17 shows weak scaling results (weak speedups and efficiencies) for the four representative UPCBLAS routines. In order to simplify the graphs, only the results for the best data distribution for each routine are shown. These results prove that the scalability of the routines is much better when increasing the size of the matrix with the number of threads (maintaining the number of elements per thread), especially for the BLAS3 routines, where weak efficiencies are always over 90%.

Finally, Table 6 provides another point of view of the results presented in this section by showing the percentage of the theoretical peak flops of the Finis Terrae supercomputer (6.4 GFlops per processor core) obtained by the UPCBLAS and PBLAS versions of each routine. This percentage is only shown for the best distribution in each case and the problem size is the same as in the previous experiments (see Figures 13-16).



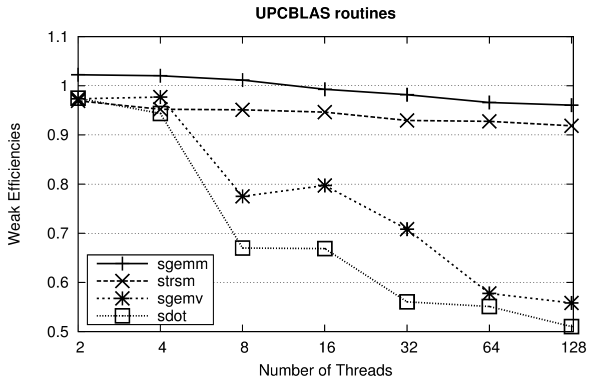


Figure 17: Weak speedups and efficiencies of the UPCBLAS routines using the best distribution

Table 6: Percentage of the theoretical peak flops of the machine obtained by each routine

# THREADS	*sdot*	*sgemv*		*sgemm*		*strsm*	
↓	UPC	UPC	PBLAS	UPC	PBLAS	UPC	PBLAS
1	18.64%	26.28%	30.19%	95.01%	98.06%	75.70%	80.47%
2	16.92%	24.04%	27.85%	92.83%	98.16%	87.76%	85.88%
4	12.53%	18.18%	28.00%	92.46%	98.66%	86.68%	84.82%
8	12.16%	17.88%	30.13%	92.80%	98.59%	83.70%	79.35%
16	11.25%	16.45%	20.54%	91.20%	98.03%	86.58%	73.54%
32	10.53%	14.15%	20.90%	85.60%	97.39%	72.91%	57.89%
64	9.02%	11.91%	19.42%	77.13%	95.33%	54.31%	35.96%
128	8.50%	8.03%	15.98%	60.75%	95.42%	42.52%	26.74%

7 Discussion: UPC as a Supporting Language for Numerical Libraries

This work has presented a parallel numerical library built on top of standard UPC. As this is the first numerical library developed for this language several issues (e.g. name and syntax of the functions, the way to represent the distributed vectors and matrices, available distributions...) have arisen during its design and implementation. The solution to all these issues has led to a good trade-off between programmability and performance.

In general, programs that use parallel numerical libraries must carry out the following steps: 1) create the structures to represent the distributed vectors and matrices; 2) distribute the data of the vectors and matrices into these structures; 3) call the numerical functions using the structures as parameters; 4) perform other operations with the distributed data (e.g. gathering or reducing some elements so that they are in the local memory of one process, write some elements of all or some processes in a file...); 5) release the structures.

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 2D distribution of the matrices 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 [24]. In contrast, UPC libraries can make use of shared arrays, making steps 1, 2, 4 and 5 almost trivial. Therefore UPC can significantly improve the ease of use of parallel numerical libraries and thus the productivity of numerical applications developers. Furthermore, UPCBLAS also facilitates the third step by simplifying the interface of the routines. As shown in Section 4, the syntax of the UPCBLAS functions is quite similar to the syntax of the corresponding sequential BLAS routines, only changing the type of the pointers (so they can point to shared memory) and including additional parameters to indicate the type of distribution.

However, the use of UPC shared arrays to distribute vectors and matrices imposes some limitations on the types of distributions that can be performed. On the one hand, the block factor must be constant for all the blocks. On the other hand, multidimensional distributions are not allowed. Thus, for some UPCBLAS routines, the distribution that theoretically obtains the best performance (e.g. 2D distributions for the matrix-matrix product) is not available. Nevertheless, UPCBLAS routines include optimization techniques in order to improve their performance. Some of these techniques are architectural-aware and they take advantage of the knowledge of some characteristics of the system in order to adapt the behavior of the routines to the machine on which the library is installed. The results shown in the previous section have proved that, with all these techniques, the performance of the library is more than acceptable.

8 Conclusions

PGAS languages provide programmability and good data locality exploitation on shared, distributed and hybrid shared/distributed memory architectures. In fact, PGAS languages such as UPC represent an interesting alternative for programming multicore clusters, where threads running on the same node can access their data efficiently through shared memory, whereas the use of distributed memory improves the scalability of the applications.

However, the lack of available libraries is preventing the acceptance of these languages. In order to solve this issue we have developed UPCBLAS, the first parallel numerical library to our knowledge developed for UPC. Up to now, in order to use BLAS routines, parallel programmers needed to resort to message-passing based libraries. With the library presented in this paper, UPC programmers can benefit as well from a portable and efficient BLAS-based library that can also be used as building block for higher level numerical computations (e.g. factorizations, iterative methods...).

The implemented library provides functions for dense computations where the ease of use has been an important factor in all the design decisions in order to preserve the programmability property of the PGAS languages and reduce the library learning curve. The library works with the data distributions provided by the user through the block factor specified in shared arrays. Thus, UPCBLAS is easier and more intuitive to use than message-passing based numerical libraries (e.g. PBLAS) thanks to using directly the vectors and matrices as source and result parameters of the routines as in sequential BLAS, instead of using the complex data structures that PBLAS routines need to handle distributed vectors and matrices. Besides, the ease of programming is twofold: on the one hand, the BLAS-like interface facilitates the use of the library to programmers used to sequential BLAS and, on the other hand, the syntax of UPCBLAS is very familiar to UPC programmers as it is similar to that of the UPC collectives library.

Several optimization techniques have been applied to improve performance. Some examples are privatization of shared pointers, bulk data movements, redesign of some collective operations or implementation of on-demand copies. Furthermore, sequential BLAS routines are embedded in the body of the corresponding UPC routines. 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.

The proposed library has been experimentally tested on a multicore cluster to show the suitability and efficiency of the library for hybrid architectures. We can assert that the ease of use of the UPCBLAS library does not lead to a much worse performance than the well-established and mature message-passing based numerical libraries.

As ongoing work the UPCBLAS routines are being used to implement, in an efficient way, more complex numerical routines, such as LU or Cholesky factorizations.

Acknowledgments

This work was funded by Hewlett-Packard (Project "Improving UPC Usability and Performance in Constellation Systems: Implementation/Extensions of UPC Libraries"), by the Ministry of Science and Innovation of Spain under Project TIN2010-16735, and by the Ministry of Education of Spain under the FPU research grant AP2008-01578. We gratefully thank Enrique Quintana-Ortí (Jaume I University, Spain) for his valuable comments on the UPCBLAS design.

References

[1] Titanium Project. http://titanium.cs.berkeley.edu/, Last visit: October 2011.

[2] Co-Array Fortran. http://www.co-array.org/, Last visit: October 2011.

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

[4] 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.

[5] Christian Bell, Dan Bonachaea, Rajesh Nishtala, and Katherine 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.

[6] Christopher Barton, Călin Caşcaval, George Almási, Yili Zheng, Montse Farreras, Siddhartha Chatterjee, and José N. Amaral. Shared Memory Programming for Large Scale Machines. In *Proc. 10th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'06)*, pages 108–117, Ottawa, Canada, 2006.

[7] 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)*, volume 5759 of *Lecture Notes in Computer Science*, pages 174–184, Espoo, Finland, 2009.

[8] Hongzhang Shan, Filip Blagojevic, Seung-Jai Min, Paul Hargrove, Haoqiang Jin, Karl Fuerlinger, Alice Koniges, and Nicholas J. Wright. A Programming Model Performance Study using the NAS Parallel Benchmarks. *Scientific Programming*, 18(3-4):153–167, 2010.

[9] Berkeley UPC Project. http://upc.lbl.gov, Last visit: October 2011.

[10] GCC Unified Parallel C. http://www.gccupc.org/, Last visit: October 2011.

[11] HP Unified Parallel C. http://www.hp.com/go/upc/, Last visit: October 2011.

[12] IBM XL UPC Compilers. http://www.alphaworks.ibm.com/tech/upccompiler/, Last visit: October 2011.

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

[14] 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.

[15] Parallel Basic Linear Algebra Subprograms (PBLAS) Library. http://www.netlib.org/scalapack/pblas_qref.html, Last visit: October 2011.

[16] Jaeyoung Choi, Jack J. Dongarra, Susan Ostrouchov, Antoine Petitet, David Walker, and R. Clinton Whaley. A Proposal for a Set of Parallel Basic Linear Algebra Subprograms. In *Proc. 2nd Intl. Workshop on Applied Parallel Computing, Computations in Physics, Chemistry and Engineering Science (PARA'95)*, volume 1041 of *Lecture Notes in Computer Science*, pages 107–114, Lyngby, Denmark, 1995.

[17] The ScaLAPACK Project. http://netlib2.cs.utk.edu/scalapack/index.html, Last visit: October 2011.

[18] Linear Algebra PACKage (LAPACK). http://www.netlib.org/lapack/, Last visit: October 2011.

[19] José I. Aliaga, Francisco Almeida, José M. Badía, Sergio Barrachina, Vicente Blanco, María Castillo, Rafael Mayo, Enrique S. Quintana-Ortí, Gregorio Quintana-Ortí, Alfredo Remón, Casiano Rodríguez, Francisco Sande, and Adrián Santos. Towards the Parallelization of GSL. *Journal of Supercomputing*, 48(1):88–114, 2009.

[20] GSL - GNU Scientific Library. http://www.gnu.org/software/gsl/, Last visit: October 2011.

[21] Intel Math Kernel Library. http://software.intel.com/en-us/articles/intel-mkl/, Last visit: October 2011.

[22] The IBM Engineering Scientific Subroutine Library (ESSL) and Parallel ESSL. http://www-03.ibm.com/systems/p/software/essl/index.html, Last visit: October 2011.

[23] HP's Mathematical Software Library (MLIB). http://www.hp.com/go/mlib, Last visit: October 2011.

[24] LAPACK and ScaLAPACK survey. http://icl.cs.utk.edu/lapack-forum/survey/, Last visit: October 2011.

[25] L. Anthony Drummond, Vicente G. Ibarra, Violeta Migallón, and José Penadés. Interfaces for Parallel Numerical Linear Algebra Libraries in High Level Languages. *Advances in Engineering Software*, 40(8):652–658, 2009.

[26] Jack Poulson, Brian Marker, Jeff R. Hammond, Nichols A. Romero, and Robert A. van de Geijn. Elemental: A New Framework for Distributed Memory Dense Matrix Computations (preprint submitted for publication), 2011. Available at http://users.ices.utexas.edu/∼poulson/publications/Elemental.pdf.

[27] François Cantonnet, Yiyi Yao, Mohamed Zahran, and Tarek El-Ghazawi. Productivity Analysis of the UPC Language. In *Proc. 18th Intl. Parallel and Distributed Processing Symp. (IPDPS'04)*, Santa Fe, NM, USA, 2004.

[28] Imran Patel and John R. Gilbert. An Empirical Study of the Performance and Productivity of Two Parallel Programming Models. In *Proc. 22nd Intl. Parallel and Distributed Processing Symp. (IPDPS'08)*, Miami, FL, USA, 2008.

[29] Carlos Teijeiro, Guillermo L. Taboada, Juan Touriño, Basilio B. Fraguela, Ramón Doallo, Damián A. Mallón, Andrés Gómez, José C. Mouriño, and Brian Wibecan. Evaluation of UPC Programmability Using Classroom Studies. In *Proc. 3rd Conf. on Partitioned Global Address Space Programming Models (PGAS'09)*, Ashburn, VA, USA, 2009.

[30] Robert W. Numrich. A Parallel Numerical Library for Co-Array Fortran. In *Proc. Workshop on Language-Based Parallel Programming Models (WLPP'05)*, volume 3911 of *Lecture Notes in Computer Science*, pages 960–969, Poznan, Poland, 2005.

[31] Nadya Travinin and Jeremy Kepner. pMatlab Parallel Matlab Library. *Intl. Journal of High Performance Computing Applications*, 21(3):336–359, 2007.

[32] Christian Bell and Rajesh Nishtala. UPC Implementation of the Sparse Triangular Solve and NAS FT, University of California, Berkeley,, 2004. Available at http://www.cs.berkeley.edu/∼rajeshn/pubs/bell_nishtala_spts_ft.pdf.

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

[34] 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)*, volume 5704 of *Lecture Notes in Computer Science*, pages 630–641, Delft, The Netherlands, 2009.

[35] 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.

[36] Jorge González-Domínguez, María J. Martín, Guillermo L. Taboada, and Juan Touriño. Dense Triangular Solvers on Multicore Clusters using UPC. In *Proc. 11th Intl. Conf. on Computational Science (ICCS 2011)*, volume 4 of *Procedia Computer Science*, pages 231–240, Singapore, 2011.

[37] Yili Zheng. Optimizing UPC Programs for Multi-Core Systems. *Scientific Programming*, 18(3-4):183–191, 2010.

[38] Wei-Yu Chen, Dan Bonachea, Jason Duell, Parry Husbands, Costin Iancu, and Katherine 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.

[39] 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 Intl. Parallel and Distributed Processing Symp. (IPDPS'10)*, Atlanta, GA, USA, 2010.

[40] The Servet Benchmark Suite Project. http://servet.des.udc.es/, Last visit: October 2011.

[41] Jorge González-Domínguez, Guillermo L. Taboada, Basilio B. Fraguela, María J. Martín, and Juan Touriño. Automatic Mapping of Parallel Applications on Multicore Architectures using the Servet Benchmark Suite. Technical Report, Computer Architecture Group, Department of Electronics and Systems, University of A Coruña, Spain,, 2011. Available at http://servet.des.udc.es/publications/TechnicalReportServet.pdf.

[42] Finis Terrae Supercomputer. http://www.top500.org/system/9500, Last visit: October 2011.