A Parallel Numerical Library for UPC

Jorge González-Domínguez[1], María J. Martín[1], Guillermo L. Taboada[1], Juan Touriño[1], Ramón Doallo[1], and Andrés Gómez[2]

[1] Computer Architecture Group,
University of A Coruña, Spain
{jgonzalezd, mariam, taboada, juan, doallo}@udc.es
[2] Galicia Supercomputing Center (CESGA),
Santiago de Compostela, Spain
agomez@cesga.es

Abstract. Unified Parallel C (UPC) is a Partitioned Global Address Space (PGAS) language that exhibits high performance and portability on a broad class of shared and distributed memory parallel architectures. This paper describes the design and implementation of a parallel numerical library for UPC built on top of the sequential BLAS routines. The developed library exploits the particularities of the PGAS paradigm, taking into account data locality in order to guarantee a good performance. The library was experimentally validated, demonstrating scalability and efficiency.

Keywords: Parallel computing, PGAS, UPC, Numerical libraries, BLAS

1 Introduction

The Partitioned Global Address Space (PGAS) programming model provides important productivity advantages over traditional parallel programming models. 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. In this way, the programmer can exploit data locality in order to increase the performance, at the same time as the shared space facilitates the development of parallel codes. PGAS languages trade off ease of use for efficient data locality exploitation. Over the past several years, 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 standard C for parallel computing. In [4] El-Ghazawi et al. establish, through extensive performance measurements, that UPC can potentially perform at similar levels to those of MPI. Barton et al. [5] further demonstrate that UPC codes can obtain good performance scalability up to thousands of processors with the right support from the compiler and the runtime system. Currently there are commercial and open source based compilers of UPC for nearly all parallel machines.

However, a barrier to a more widespread acceptance of UPC is the lack of library support for algorithm developers. The BLAS (Basic Linear Algebra Subprograms) [6,7] are routines that provide standard building blocks for performing basic vector and matrix operations. They are widely used by scientists and engineers to obtain good levels of performance through an efficient exploitation of the memory hierarchy. The PBLAS (Parallel Basic Linear Algebra Subprograms) [8,9] are a parallel subset of BLAS, which have been developed to assist programmers on distributed memory systems. However, PGAS-based counterparts are not available. In [10] a parallel numerical library for Co-Array Fortran is presented, but this library focuses on the definition of distributed data structures based on an abstract object called object map. It uses Co-Array syntax, embedded in methods associated with distributed objects, for communication between objects based on information in the object map.

This paper presents a library for numerical computation in UPC that improves the programmability and performance of UPC applications. The library contains a relevant subset of the BLAS routines. The developed routines exploit the particularities of the PGAS paradigm, taking into account data locality in order to guarantee a good performance. Besides, the routines use internally BLAS calls to carry out the sequential computation inside each thread. The library was experimentally validated on the HP Finis Terrae supercomputer [11].

The rest of the paper is organized as follows. Section 2 describes the library design: types of functions (shared and private), syntax and main characteristics. Section 3 explains the data distributions used for the private version of the routines. Section 4 presents the experimental results obtained on the Finis Terrae supercomputer. Finally, conclusions are discussed in Section 5.

2 Library Design

Each one of the selected BLAS routines has two UPC versions, a *shared* and a *private* one. In the shared version the data distributions are provided by the user through the use of shared arrays with a specific `block_size` (that is, the blocking factor or number of consecutive elements with affinity to the same thread). In the private version the input data are private. In this case the data are transparently distributed by the library. Thus, programmers can use the library independently of the memory space where the data are stored, avoiding to make tedious and error-prone distributions. Table 1 lists all the implemented routines, giving a total of 112 different routines ($14 \times 2 \times 4$).

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

The developed routines do not implement the numerical operations (e.g. dot product, matrix-vector product, etc.) but they call internally to BLAS routines to perform the sequential computations in each thread. Thus, the UPC BLAS

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
	Tnrm2	Euclidean norm
	Tasum	Sums the absolute value of the elements of a vector
	iTamax	Finds the index with the maximum value
	iTamin	Finds the index with the minimum value
BLAS2	Tgemv	Matrix-vector product
	Ttrsv	Solves a triangular system of equations
	Tger	Outer product
BLAS3	Tgemm	Matrix-matrix product
	Ttrsm	Solves a block of triangular systems of equations

Table 1: UPC BLAS routines. All the routines follow the naming convention: `upc_blas_[p]Tblasname`. Where the character "p" indicates private function, that is, the input arrays are private; character "T" indicates the type of data (i=integer; l=long; f=float; d=double); and `blasname` is the name of the routine in the sequential BLAS library.

routines act as an interface to distribute data and synchronize the calls to the BLAS routines in an efficient and transparent way.

2.1 Shared Routines

The UPC language has two standard libraries: the collective library (integrated into the language specification, v1.2 [3]) and the I/O library [12]. The shared version of the numerical routines follows the syntax style of the collective UPC functions. Using the same syntax style as the UPC collective routines eases the learning process to the UPC users. For instance, the syntax of the UPC dot product routine with shared data is:

```
int upc_blas_ddot(const int block_size, const int size, shared
    const double *x, shared const double *y, shared double *dst);
```

being `x` and `y` the source vectors of length `size`, and `dst` the pointer to shared memory where the dot product result will be written; `block_size` is the blocking factor of the source vectors. This function treats `x` and `y` pointers as if they had type `shared [block_size] double[size]`.

In the case of BLAS2 and BLAS3 routines, an additional parameter (`dimmDist`) is needed to indicate the dimension used for the matrix distribution because shared arrays in UPC can only be distributed in one dimension. The

UPC routine to solve a triangular system of equations is used to illustrate this issue:

```
int upc_blas_dtrsv(const UPC_PBLAS_TRANSPOSE transpose, const
     UPC_PBLAS_UPLO uplo, const UPC_PBLAS_DIAG diag, const
     UPC_PBLAS_DIMMDIST dimmDist, const int block_size, const
     int n, shared const double *M, shared double *x);
```

being `M` the source matrix and `x` the source and result vector; `block_size` is the blocking factor of the source matrix; `n` the number of rows and columns of the matrix; `transpose`, `uplo` and `diag` enumerated values which determine the characteristics of the source matrix (transpose/non-transpose, upper/lower triangular, elements of the diagonal equal to one or not); `dimmDist` is another enumerated value to indicate if the source matrix is distributed by rows or columns. The meaning of the `block_size` parameter depends on the `dimmDist` value. If the source matrix is distributed by rows (`dimmDist=upc_pblas_rowDist`), `block_size` is the number of consecutive rows with affinity to the same thread. In this case, this function treats pointer `M` as if it had type `shared[block_size*n] double[n*n]`. Otherwise, if the source matrix is distributed by columns (`dimmDist=upc_pblas_colDist`), `block_size` is the number of consecutive columns with affinity to the same thread. In this case, this function treats pointer `M` as if it had type `shared[block_size] double[n*n]`.

As mentioned before, in the shared version of the BLAS2 and BLAS3 routines it is not possible to distribute the matrices by 2D blocks (as the UPC syntax does not allow it), which can be a limiting factor for some kinds of parallel computations. To solve this problem, an application layer with support for various dense matrix decomposition strategies is presented in [13]. A detailed discussion about its application to a particular class of algorithms is also included. However, such a support layer still requires considerable development to become useful for a generic numerical problem. In [14] an extension to the UPC language that allows the programmer to block shared arrays in multiple dimensions is proposed. This extension is not currently part of the standard.

2.2 Private Routines

The private version of the routines does not store the input data in a shared array distributed among the threads, but data are completely stored in the private memory of one or more threads. Its syntax is similar to the shared one, but the `block_size` parameter is omitted as in this case the data distribution is internally applied by the library. For instance, the syntax for the dot routine is:

```
int upc_blas_pddot(const int size, const int src_thread, const
   double *x, const double *y, const int dst_thread, double *dst);
```

being `x` and `y` the source vectors of length `size`; `dst` the pointer to private memory where the dot product result will be written; and `src_thread`/`dst_thread`

the rank number of the thread (0,1,...THREADS-1, being THREADS the total number of threads in the UPC execution) where the input/result is stored. If src_thread=THREADS, the input is replicated in all the threads. Similarly, if dst_thread=THREADS the result will be replicated to all the threads.

The data distributions used internally by the private routines will be explained in the next section. Unlike the shared version of the routines, in the private one a 2D block distribution for the matrices can be manually built.

2.3 Optimization Techniques

There exist a number of known optimization techniques that improve the efficiency and performance of the UPC codes. The following optimizations have been applied to the implementation of the routines whenever possible:

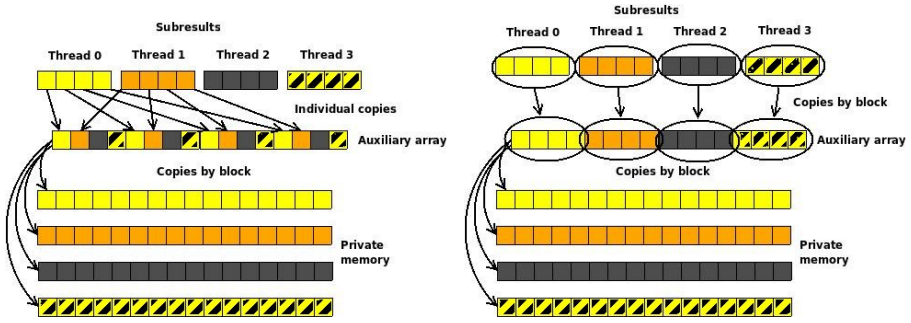
- Space privatization: When dealing with local shared data, they are accessed through standard C pointers instead of using UPC pointers to shared memory. Shared pointers often require more storage and are more costly to dereference. Experimental measurements in [4] have shown that the use of shared pointers increases execution times by up to three orders of magnitude. For instance, space privatization is widely used in our routines when a thread needs to access only to the elements of a shared array with affinity to that thread.
- Aggregation of remote shared memory accesses: This can be established through block copies, using upc_memget and upc_memput on remote blocks of data required by a thread, as will be shown in the next section.
- Overlapping of remote memory accesses with computation: It was achieved by the usage of split-phase barriers. For instance, these barriers are used in the triangular solver routines to overlap the local computation of each thread with the communication of partial results to the other threads (see Section 3.3).

3 Data Distributions for the Private Routines

UPC local accesses can be one or two orders of magnitude faster than UPC remote accesses. Thus, all the private functions have been implemented using data distributions that minimize accesses to remote memory in a transparent way to the user.

As mentioned in the previous section, if the parameter dst_thread=THREADS, the piece of result obtained by each thread is replicated to all the other ones. To do this, two different options were considered:

- Each thread copies its piece of result into THREADS shared arrays, each one with affinity to a different thread. This leads to THREADS-1 remote accesses for each thread, that is, THREADS $\times$ (THREADS - 1) accesses.



(a) cyclic distribution (b) block distribution

Fig. 1: Data movement using a block and a cyclic distribution

– Each thread first copies its piece of result into a shared array with affinity to thread 0. Then, each thread copies the whole result from thread 0 to its private memory. In this case the number of remote accesses is only $2 \times$ (`THREADS`-1), although in half of these accesses (the copies in private memory) the amount of data transfered is greater than in the first option.

A preliminary performance evaluation of these two options (using the benchmarks of Section 4) has shown that the second one achieves the highest performance, especially on distributed memory systems.

Due to the similarity among the algorithms used in each BLAS level, the chosen distribution has been the same for all the routines inside the same level (except for the triangular solver).

3.1 BLAS1 Distributions

UPC BLAS1 consists of routines that operate with one or more dense vectors. Figure 1 shows the copy of the result in all the threads for a cyclic and a block distribution using the procedure described above (the auxiliary array is a shared array with affinity to thread 0). The block distribution was chosen because all the copies can be carried out in block, allowing the use of the `upc_memput()` and `upc_memget()` functions.

3.2 BLAS2 Distributions

This level contains matrix-vector operations. A matrix can be distributed by rows, by columns or by 2D blocks. The matrix-vector product will be used as an example to explain the chosen distribution.

Figure 2 shows the routine behavior for a column distribution. In order to compute the $i-th$ element of the result, all the $i-th$ values of the subresults should be added. Each of these adds is performed using the `upc_all_reduceT`

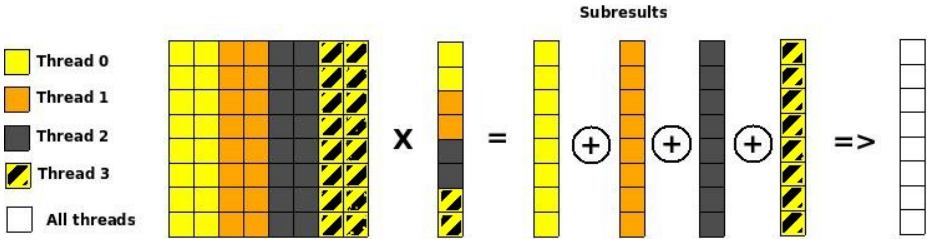


Fig. 2: Matrix-vector product using a column distribution for the matrix

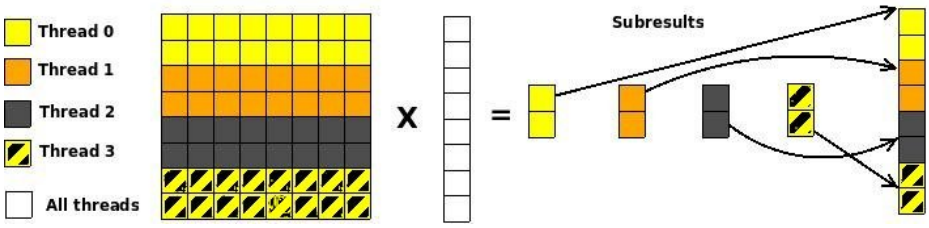


Fig. 3: Matrix-vector product using a row distribution for the matrix

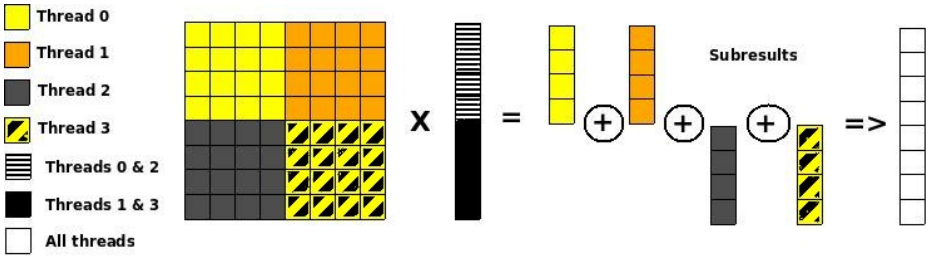


Fig. 4: Matrix-vector product using a 2D block distribution for the matrix

function (with `upc_op_t op=UPC_ADD`) from the collective library. Other languages allow to compute the set of all the reduction operations in an efficient way using a unique collective function (e.g. `MPI_Reduce`). However, UPC has not such a function, at least currently.

The row distribution is shown in Figure 3. In this case each thread computes a piece of the final result (subresults in the figure). These subresults only must be copied in an auxiliary array in the right position, and no reduction operation is necessary.

Finally, Figure 4 shows a 2D block distribution. This is a good alternative for cache locality exploitation. This option involves a reduction operation for each row. Each one of these reductions must be performed by all the threads (although not all the threads have elements for all the rows) because UPC does not allow to use a subset of threads in collective functions.

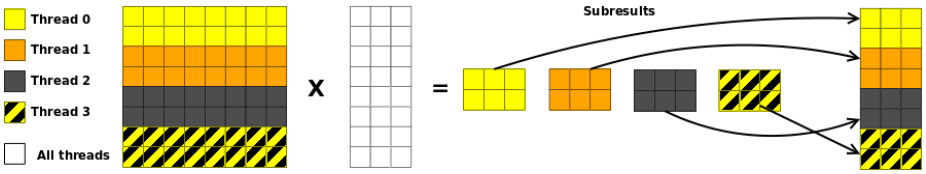


Fig. 5: Matrix-matrix product using a row distribution for the matrix

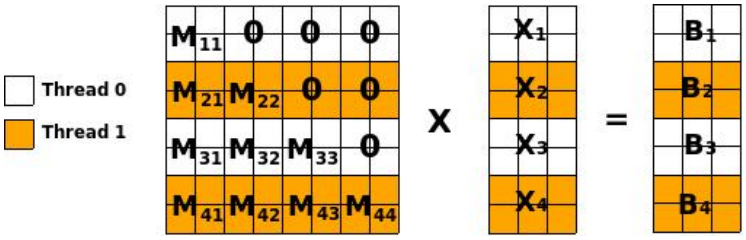


Fig. 6: Matrix distribution for the `upc_blas_pTtrsm` routine

Experimental results showed that the row distribution is the best option as no collective operation is needed. In [15] Nishtala et al. propose extensions to the UPC collective library in order to allow subsets (or teams) of threads to perform a collective function together. If these extensions were included in the UPC specification the column and 2D block distributions should be reconsidered.

3.3 BLAS3 Distributions

This level contains matrix-matrix operations. Once again, a matrix can be distributed by rows, by columns or by 2D blocks. Actually, the advantages and disadvantages of each distribution are the same discussed in the previous section for BLAS2 routines, so the row distribution was selected again. Figure 5 shows the routine behavior for the matrix-matrix product using this row distribution.

Regarding the matrix distribution, the routines to solve triangular systems of equations (`Ttrsv` and `Ttrsm`) are a special case. In these routines, the rows of the matrix are not distributed in a block way but in a block-cyclic one in order to increase the parallelism and balance the load. Figure 6 shows an example for the `Ttrsm` routine with two threads, two blocks per thread and a source matrix with the following options (see the syntax of the similar `Ttrsv` routine in Section 2.1): non-transpose (`transpose=upc_pblas_noTrans`), lower triangular (`uplo=upc_pblas_lower`), and not all the main diagonal elements equal to one (`diag=upc_pblas_nonUnit`). 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 algorithm used by this routine is shown in Figure 7. The triangular system can be computed as a sequence of triangular solutions (`Ttrsm`) and matrix-

```
1 X1 ← Solve M11 * X1 = B1                                  BLAS Ttrsm()
2 --- synchronization ---
3 B2 ← B2 − M21 * X1                                        BLAS Tgemm()
4 B3 ← B3 − M31 * X1                                        BLAS Tgemm()
5 B4 ← B4 − M41 * X1                                        BLAS Tgemm()
6 X2 ← Solve M22 * X2 = B2                                  BLAS Ttrsm()
7 --- synchronization ---
8 B3 ← B3 − M32 * X2                                        BLAS Tgemm()
9 ...
```

Fig. 7: UPC BLAS `Ttrsm` algorithm

matrix multiplications (`Tgemm`). Note that all operations between two synchronizations can be performed in parallel. The more blocks the matrix is divided in, the more computations can be simultaneously performed, but the more synchronizations are needed too. Experimental measurements have been made to find the block size with the best trade-off between parallelism and synchronization overhead. For our target supercomputer (see Section 4) and large matrices (`n`>2000) this value is approximately 1000/`THREADS`.

4 Experimental Results

In order to evaluate the performance of the library, different benchmarks were run on the Finis Terrae supercomputer installed at the Galicia Supercomputing Center (CESGA), ranked #427 in November 2008 TOP500 list (14 TFlops) [11]. It consists of 142 HP RX7640 nodes, each of them with 16 IA64 Itanium 2 Montvale cores at 1.6 Ghz distributed in two cells (8 cores per cell), 128 GB of memory and a dual 4X Infiniband port (16 Gbps of theoretical effective bandwidth).

As UPC programs can be run on shared or distributed memory, the results were obtained using the hybrid memory configuration (shared memory for intra-node communication and Infiniband transfers for inter-node communication). This hybrid architecture is very interesting for PGAS languages, allowing the locality exploitation of threads running in the same node, as well as enhancing scalability through the use of distributed memory.

Among all the possible hybrid configurations, four threads per node, two per cell, was chosen. The use of this configuration represents a good trade-off between shared-memory access performance and scalable access through the Infiniband network. The compiler used is the Berkeley UPC 2.6.0 [16], and the Intel Math Kernel Library (MKL) 9.1 [17], a highly tuned BLAS library for Itanium cores, is the underlying sequential BLAS library.

The times measured in all the experiments were obtained for the private version of the routines with the inputs (matrices/vectors) initially replicated in all threads (`src_thread`=`THREADS`), and results stored in thread 0 (`dst_thread`=0). Results for one core have been obtained with the sequential MKL version.

Results taken from shared routines are not presented because they are very dependent on the data distribution chosen by the user. If programmers choose the best distribution (that is, the same used in the private versions and explained in Section 3), times and speedups obtained would be similar to those of the private counterparts.

Table 2 and Figure 8 show the execution times, speedups and efficiencies obtained for different vector sizes and number of threads of the `pddot` routine, an example of BLAS1 level routine with arrays of doubles stored in private memory. The size of the arrays is measured in millions of elements. Despite computations are very short (in the order of milliseconds), this function scales reasonably well. Only the step from four to eight threads decrease the slope of the curve, as eight threads is the first configuration where not only shared memory but also Infiniband communications are used.

Times, speedups and efficiencies for the matrix-vector product (`pdgemv`, an example of BLAS2 level) are shown in Table 3 and Figure 9. Square matrices are used; the size represents the number of rows and columns. As can be observed, speedups are quite high despite the times obtained from the executions are very short.

Finally, Table 4 and Figure 10 show times (in seconds), speedups and efficiencies obtained from the execution of a BLAS3 routine, the matrix-matrix product (`dgemm`). Input matrices are also square. Speedups are higher in this function because of the large computational cost of its sequential version so that the UPC version benefits from the high ratio computation/communication time.

5 Conclusions

To our knowledge, this is the first parallel numerical library developed for UPC. Numerical libraries improve performance and programmability of scientific and engineering applications. Up to now, in order to use BLAS libraries, parallel programmers have to resort to MPI or OpenMP. With the library presented in this paper, UPC programmers can also benefit from these highly efficient numerical libraries, which allows for broader acceptance of this language.

The library implemented allows to use both private and shared data. In the first case the library decides in a transparent way the best data distribution for each routine. In the second one the library works with the data distribution provided by the user. In both cases UPC optimization techniques, such as privatization or bulk data movements, are applied in order to improve the performance.

BLAS library implementations have evolved over about two decades and are therefore extremely mature both in terms of stability and efficiency for a wide variety of architectures. Thus, the sequential BLAS library is embedded in the body of the corresponding UPC routines. Using sequential BLAS not only improves efficiency, but it also allows to incorporate automatically the new BLAS versions as soon as available.

DOT PRODUCT (pddot)			
Thr. \ Size	50M	100M	150M
1	147,59	317,28	496,37
2	90,47	165,77	262,37
4	43,25	87,38	130,34
8	35,58	70,75	87,60
16	18,30	35,70	53,94
32	9,11	17,95	26,80
64	5,22	10,68	15,59
128	3,48	7,00	10,60

Table 2: BLAS1 `pddot` times (ms)

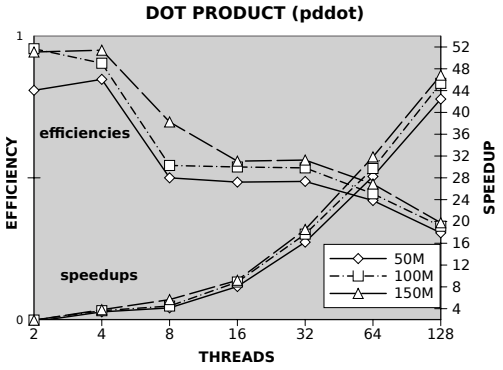


Fig. 8: BLAS1 `pddot` efficiencies/speedups

MATRIX-VECTOR (pdgemv)			
Thr. \ Size	10000	20000	30000
1	145,08	692,08	1.424,7
2	87,50	379,24	775,11
4	43,82	191,75	387,12
8	27,93	106,30	198,88
16	15,18	53,58	102,09
32	9,38	38,76	79,01
64	4,55	19,99	40,48
128	2,39	10,65	21,23

Table 3: BLAS2 `pdgemv` times (ms)

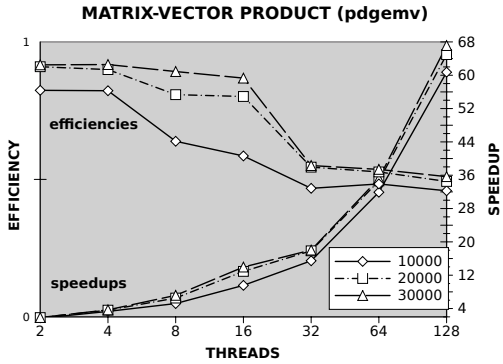


Fig. 9: BLAS2 `pdgemv` efficiencies/speedups

MATRIX-MATRIX (pdgemm)			
Thr. \ Size	6000	8000	10000
1	68,88	164,39	319,20
2	34,60	82,52	159,81
4	17,82	41,57	80,82
8	9,02	21,27	41,53
16	4,72	10,99	21,23
32	2,56	5,90	11,23
64	1,45	3,24	6,04
128	0,896	1,92	3,50

Table 4: BLAS3 `pdgemm` times (s)

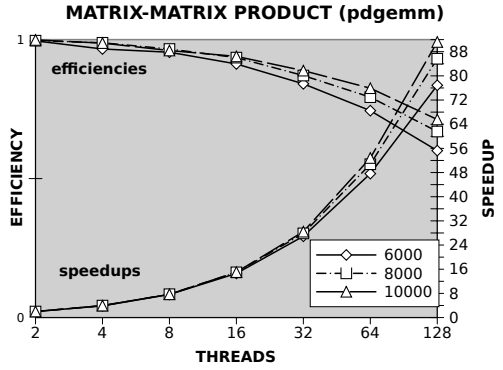


Fig. 10: BLAS3 `pdgemm` efficiencies/speedups

The proposed library has been experimentally tested, demonstrating scalability and efficiency. The experiments were performed on a multicore supercomputer to show the adequacy of the library to hybrid architectures (shared/distributed memory).

As ongoing work we are currently developing efficient UPC sparse BLAS routines for parallel sparse matrix computations.

Acknowledgments

This work was funded by the Ministry of Science and Innovation of Spain under Project TIN2007-67537-C03-02. We gratefully thank CESGA (Galicia Supercomputing Center, Santiago de Compostela, Spain) for providing access to the Finis Terrae supercomputer.

References

1. Titanium Project Home Page: `http://titanium.cs.berkeley.edu/` (Last visit: May 2009)
2. Co-Array Fortran: `http://www.co-array.org/` (Last visit: May 2009)
3. UPC Consortium: UPC Language Specifications, v1.2. (2005) `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), Baltimore, MD, USA (2002) 1–26
5. Christopher Barton, Călin Casçaval, George Almási, Yili Zheng, Montse Farreras, Siddhartha Chatterje and José Nelson Amaral: Shared Memory Programming for Large Scale Machines. In: Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'06), Ottawa, Ontario, Canada (2006) 108–117
6. BLAS Home Page: `http://www.netlib.org/blas/` (Last visit: May 2009)
7. Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling and Richard J. Hanson: An Extended Set of FORTRAN Basic Linear Algebra Subprograms. ACM Transactions on Mathematical Software **14**(1) (1988) 1–17
8. PBLAS Home Page: `http://www.netlib.org/scalapack/pblasqref.html` (Last visit: May 2009)
9. 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 International Workshop on Applied Parallel Computing, Computations in Physics, Chemistry and Engineering Science (PARA'95). Volume 1041 of Lecture Notes in Computer Science., Lyngby, Denmark (1995) 107–114
10. 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., Poznan, Poland (2005) 960–969
11. Finis Terrae Supercomputer: `http://www.top500.org/system/9500` (Last visit: May 2009)
12. Tarek El-Ghazawi, François Cantonnet, Proshanta Saha, Rajeev Thakur, Rob Ross and Dan Bonachea: UPC-IO: A Parallel I/O API for UPC v1.0. (2004) `http://upc.gwu.edu/docs/UPC-IOv1.0.pdf`.

13. Jonathan Leighton Brown and Zhaofang Wen: Toward an Application Support Layer: Numerical Computation in Unified Parallel C. In: Proc. Workshop on Language-Based Parallel Programming Models (WLPP'05). Volume 3911 of Lecture Notes in Computer Science., Poznan, Poland (2005) 912–919
14. Christopher Barton, Călin Casçaval, George Almási, Rahul Garg, José Nelson Amaral, Montse Farreras: Multidimensional Blocking in UPC. In: Proc. 20th International Workshop on Languages and Compilers for Parallel Computing (LCPC'07). Volume 5234 of Lecture Notes in Computer Science., Urbana, IL, USA (2007) 47–62
15. 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), Salt Lake City, UT, USA (2008) 99–110
16. Berkeley UPC Project: `http://upc.lbl.gov` (Last visit: May 2009)
17. Intel Math Kernel Library: `http://www.intel.com/cd/software/products/asmo-na/eng/307757.htm` (Last visit: May 2009)