Noname manuscript No.
(will be inserted by the editor)

Performance Evaluation of Sparse Matrix Products in UPC

Jorge González-Domínguez · Óscar García-López · Guillermo L. Taboada · María J. Martín · Juan Touriño

Received: date / Accepted: date

Abstract Unified Parallel C (UPC) is a Partitioned Global Address Space (PGAS) language whose popularity has increased during the last years thanks to its high programmability and reasonable performance through an efficient exploitation of data locality, especially on hierarchical architectures like multi-core clusters. However, the performance issues that arise in this language due to the irregular structure of sparse matrix operations have not yet been studied. Among them, the selection of an adequate storage format for the sparse matrices can significantly improve the efficiency of the parallel codes. This paper presents an evaluation, using UPC, of the most common sparse storage formats with different implementations of the matrix-vector and matrix-matrix products, which are key kernels in many scientific applications.

Keywords PGAS · UPC · Sparse Products · Performance Evaluation

1 Introduction and Related Work

UPC is an extension of ANSI C for parallel computing that follows the Partitioned Global Address Space (PGAS) programming model. Several studies have shown that UPC is able to obtain similar or even better performance than traditional parallel libraries such as the Message Passing Interface (MPI) [3, 6, 13, 16]. Furthermore, UPC can scale up to thousands of processors with the right support from the compiler and the run-time system [1, 12, 15].

Sparse matrix-vector and matrix-matrix products represent the main core of many iterative solvers or matrix factorizations that arise in a wide variety of scientific and engineering problems. Due to the importance of the sparse products, a large number of parallel solutions exist in the literature [4, 9, 10,

Jorge González-Domínguez · Óscar García-López · Guillermo L. Taboada · María J. Martín · Juan Touriño
Computer Architecture Group, University of A Coruña, Spain
E-mail: {jgonzalezd, oscar.garcia, taboada, mariam, juan}@udc.es

20]. Nevertheless, none of these works take advantage of the use of PGAS languages. The work in [2] is the only one addressing parallel sparse computation in the PGAS model, but it is restricted to provide a UPC implementation of the sparse triangular solver using the Compressed Sparse Row format.

Sparse formats [5] are strongly involved in achieving high performance because they define the matrix data structure in memory and the most appropriate algorithms to perform the sparse multiplications. Sparse numerical libraries must take into account the most suitable combination of storage formats and algorithms, especially when it comes to their processing in parallel on hybrid shared/distributed memory architectures. Several related works have evaluated the suitability of different storage formats to different architectures and languages such as Java [11], Fortran [19], MPI on multicore clusters [14] and CUDA in Graphic Processing Units (GPUs) [8].

This paper presents an evaluation of the suitability of the following storage formats for the Sparse Basic Linear Algebra Subprograms (SparseBLAS) [17] products in UPC:

- Coordinate (COO): It consists of three arrays, *values*, *rows* and *columns*, which store the values, row indices and column indices of the non-zero entries, respectively. In most occasions (and always in this work) the non-zero elements of the same row are assumed to be stored contiguously.
- Compressed Sparse Row (CSR): It explicitly stores subsequent non-zero values of the matrix rows in array *values*. Array *columns* keeps the column indices. A third array *rowPtr* stores, for each row, the index of the entry in the array *columns* which is the first non-zero element of the given row, and an additional entry with the number of non-zero elements in the matrix.
- Block Sparse Row (BSR): It is a variant of CSR using blocks.
- Compressed Sparse Column (CSC): It is similar to CSR, but storing the non-zero elements consecutively by columns.
- Diagonal (DIA): In this case *values* stores consecutively all the elements of the diagonals with any non-zero element. The array *distance* represents, for each stored diagonal, its offset from the main diagonal. Diagonals above and below the main one have positive and negative distance, respectively.
- Skyline (SKY): This format has been specifically designed for sparse triangular matrices and the concrete storage of the elements depends on whether the matrix is lower or upper triangular. The values of all the entries from the first non-zero element to the diagonal in each row/column are consecutively stored in *values* in the lower/upper case. Besides, an additional array *ptr* is necessary. In lower/upper matrices, it keeps for each row/column the index of the entry of *values* with the first element of this row/column.

Therefore, our work is not only the first approach to the parallel sparse products in any PGAS language but also the first performance comparison among different sparse storage formats in this programming model.

The rest of the paper is organized as follows. Section 2 provides an overview of the memory model in UPC, as background for the following sections. Section 3 describes the different approaches used in UPC to implement the sparse

matrix-vector and matrix-matrix products depending on the storage format. Section 4 presents the analysis of the experimental results obtained on an HP supercomputer (Finis Terrae). Finally, conclusions are discussed in Section 5.

2 Background: Overview of the Memory Model in UPC

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 to a part of the shared memory. This association between a portion of the shared address space and a given thread is called affinity. Moreover, UPC also provides a private memory space per thread. Therefore, each thread has access to both its private memory and to the whole global space. However, the accesses to remote data will be more expensive than the accesses to data in local memory (private memory or shared memory with affinity to the thread).

Shared arrays are employed to implicitly distribute data among threads. The syntax to declare a shared array `A` is: `shared [BLOCK_SIZE] type A[N]`, being `BLOCK_SIZE` the number of consecutive elements with affinity to the same thread, `type` the datatype, and `N` the array size. It means that the first `BLOCK_SIZE` elements are associated to thread 0, the next `BLOCK_SIZE` ones to thread 1, and so on. The block size in shared arrays cannot be variable.

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

- Private pointers (*from private to private*). The standard C pointers. 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.
- 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.
- 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.

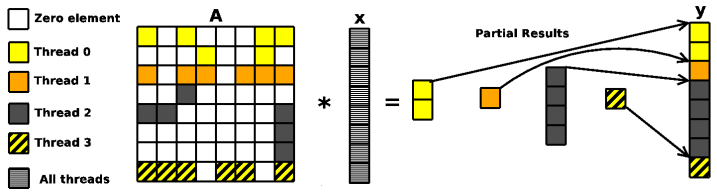
The pointers to shared memory contain three fields in order to know their exact position: the *thread* where the data is located, the *block* that contains the data and the *phase* (the location of the data within the block). Thus, when performing pointer arithmetic on a pointer-to-shared all three fields will be updated, making the operation slower than private pointer arithmetic.

3 Implementation of Sparse Matrix Products in UPC

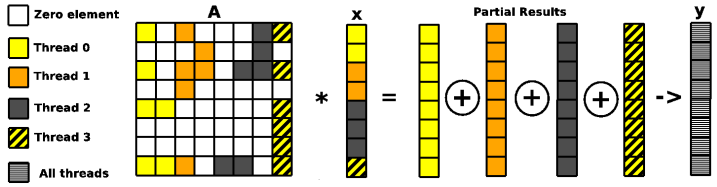
This section analyzes the implementation of the sparse matrix-vector and matrix-matrix products in UPC. The syntax of these products is the same

as in the SparseBLAS library [17], because it is widely used in scientific computations: $y = \alpha * A * x + y$ and $C = \alpha * A * B + C$ (being α a scalar value, A a sparse matrix, x and y dense vectors, and B and C dense matrices). The output vector or matrix is expected to be completely stored in an array in the local memory of one thread in both cases.

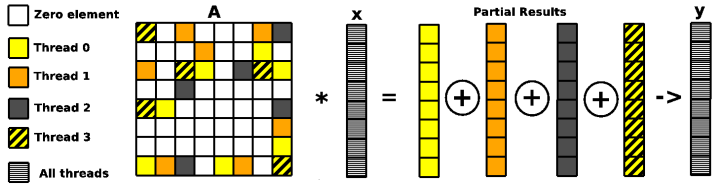
Figure 1 illustrates the three different approaches implemented for the matrix-vector product, each of them suitable for a different type of storage format. The first approach distributes the sparse matrix by rows and then each thread calculates a partial result by applying a sequential partial sparse matrix-vector product with the rows that correspond to it and all the elements of x. Thus, the distribution of y must match the distribution of the matrix so that the partial sums can be performed without remote accesses. Besides, this approach only requires one bulk copy of remote data per thread to place the partial results in their correct positions of the output array. In order to perform this distribution the non-zero elements must be consecutively stored by rows in the *values* array so it can be used in the COO, CSR, BSR and SKY (with lower matrices) formats.



(a) Distribution by rows (COO, CSR, BSR and SKY with lower matrices)



(b) Distribution by columns (CSC and SKY with upper matrices)



(c) Distribution by diagonals (DIA)

Fig. 1 Approaches for the sparse matrix-vector product

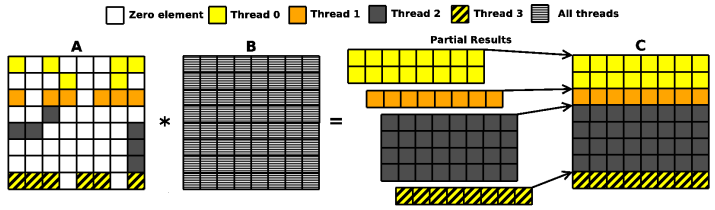
For the CSC and SKY (with upper matrices) formats, where the data in the *values* array are consecutively stored by columns, the use of this row

distribution would lead to several data movements, which would represent an important performance overhead. The natural distribution for these formats is by blocks of columns and with the source vector x distributed according to the size of the blocks in the matrix as represented in Figure 1(b). Each thread performs a partial sequential matrix-vector product with its local data. Then, in order to compute the i^{th} element of the result, the i^{th} values of all partial results should be added. These additions need reduction operations involving all UPC threads, so their performance is usually poor. The approach followed by the DIA format, which can be seen in Figure 1(c), is very similar to the previous one but the sparse matrix is distributed by diagonals.

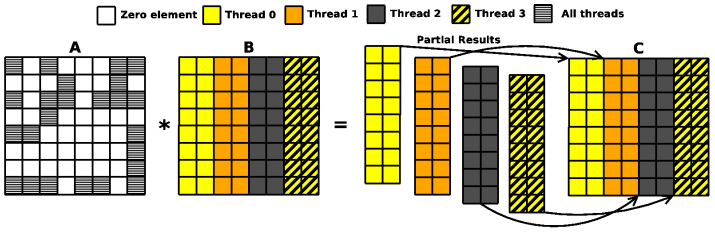
Previous works have pointed out that a key aspect in the performance of the sparse matrix-vector product is the computational load balance [20]. In order to achieve a good load balance the first and the second approaches try to distribute the matrices by providing the same or similar number of non-zero elements per thread (in the examples, six non-zero elements per thread). This implies using row/column distributions with a variable block size. This distribution must be explicitly performed in the private memories of the threads as UPC shared arrays can not have variable block size (see Section 2). In order to apply a similar mechanism with the DIA format, all the entries should be read before the distribution to know the number of non-zero elements per diagonal. This overhead is avoided by using always a cyclic distribution, which achieves a balanced load distribution in most sparse matrices (in Figure 1(c), seven non-zero elements for threads 0 and 1 and five for threads 2 and 3).

Figure 2 shows the two approaches employed for the matrix-matrix product. The first one is an adaptation of the matrix-vector distribution by rows to this problem. Each thread needs the whole matrix B and the same rows of C as in A. Only one bulk copy per thread is required to place all the elements of the output matrix consecutively in one local memory. Nevertheless, the adaptation of the distribution by columns and diagonals employed in the matrix-vector multiplication would eventually involve a significant number of final reductions, leading to a very poor performance. Therefore, the approach illustrated in Figure 2(b) was developed for CSC, DIA and SKY with upper matrices. Each thread needs to access the whole sparse matrix but only the same columns of B and C. The block distribution is used because it allows to aggregate the final copies of all elements of the same row of C using only one bulk copy of remote data per thread and row.

In all the implemented approaches, 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 in order to improve the performance of the sparse products, as explained in Section 2. All the bulk copies are performed in one go using private pointers to shared memory and the `upc_memget` function, which is much more efficient than copying all the elements one-by-one (the UPC default access).



(a) Distribution by rows (COO, CSR, BSR and SKY with lower matrices)



(b) Distribution by columns (CSC, DIA and SKY with upper matrices)

Fig. 2 Approaches for the sparse matrix-matrix product

4 Performance Evaluation

The evaluation of the sparse matrix products in UPC has been conducted on the Finis Terrae supercomputer 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. As for software, the code was compiled using Berkeley UPC 2.12.1 with the Intel C compiler (icc) version 10.1 as backend. In this machine there is a memory overhead when several cores that share the memory bus access memory at the same time. In order to avoid this overhead only 4 threads per node are used. The intra-node and inter-node communications are performed through shared memory and GASNet over InfiniBand, respectively.

In this evaluation seven representative square matrices, with different sparsity patterns, have been selected from the University of Florida Matrix Collection [18]. Their characteristics are shown in Table 1. Larger versions (labeled with "*large*") have been obtained by scaling the original matrices, which preserves the sparsity and the pattern of the original ones. The larger versions have been used for the matrix-vector product whereas the original matrices have been used for the matrix-matrix product. The DIA and SKY formats are not appropriate for storing some matrices due to the significant number of zeros that the format would require to store them, and thus these combinations have not been considered. All the results have been obtained discarding the overhead of the initial data distribution (for many applications several consecutive products are performed with the same input data distributions).

Figure 3 shows the speedups of the double precision sparse matrix-vector product using 32 and 64 threads as representative scenarios. The speedup is

Table 1 Overview of the sparse square matrices used in the evaluation

Plot	Name	Rows	Sparsity	Plot	Name	Rows	Sparsity
	nemeth26	9506	0.842%		ramage02	16830	0.509%
	nemeth26_large	85554	0.842%		ramage02_large	67320	0.509%
	nd3k	9000	2.03%		gupta3	16783	1.658%
	nd3k_large	81000	2.03%		gupta3_large	67132	1.658%
	pattern1	19242	1.26%		exdata_1	6001	3.159%
	pattern1_large	57726	1.26%		exdata_1_large	84014	3.159%
	TSOPF	18696	1.258%				
	TSOPF_large	56088	1.258%				

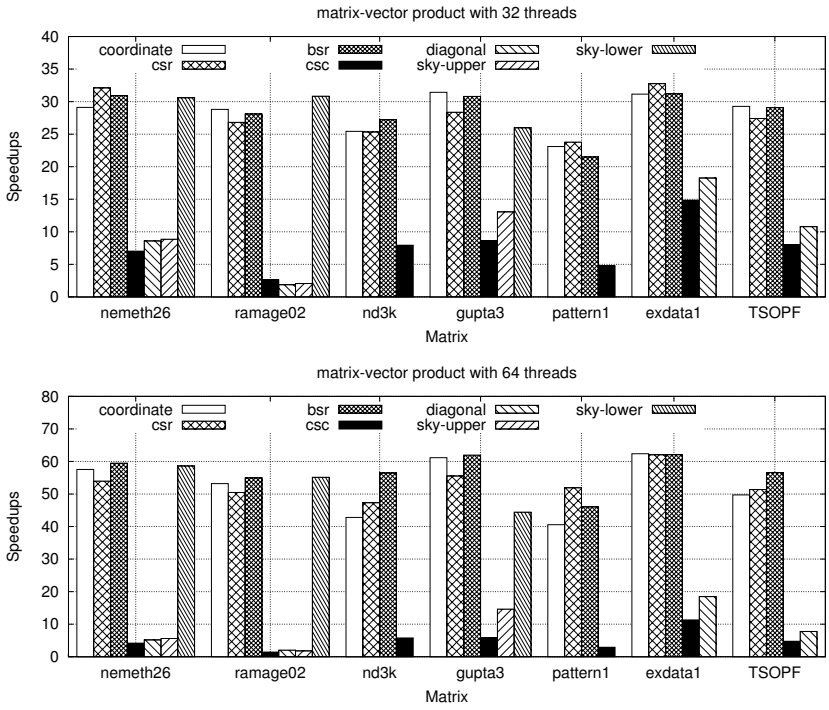


Fig. 3 Speedups of the matrix-vector product

relative to the execution time using UPC with only one thread. As expected, the row-based storage formats outperform column- and diagonal-based ones due to the avoidance of the final reduction operations, as shown in Section 3.

Figure 4 shows the results of the sparse matrix-matrix product. When working with matrices with a quite similar number of non-zero elements per row (for instance `nemeth26`, `ramage02` or `nd3k`) the row distribution obtains better performance than the column one due to the efficiency of the final copies

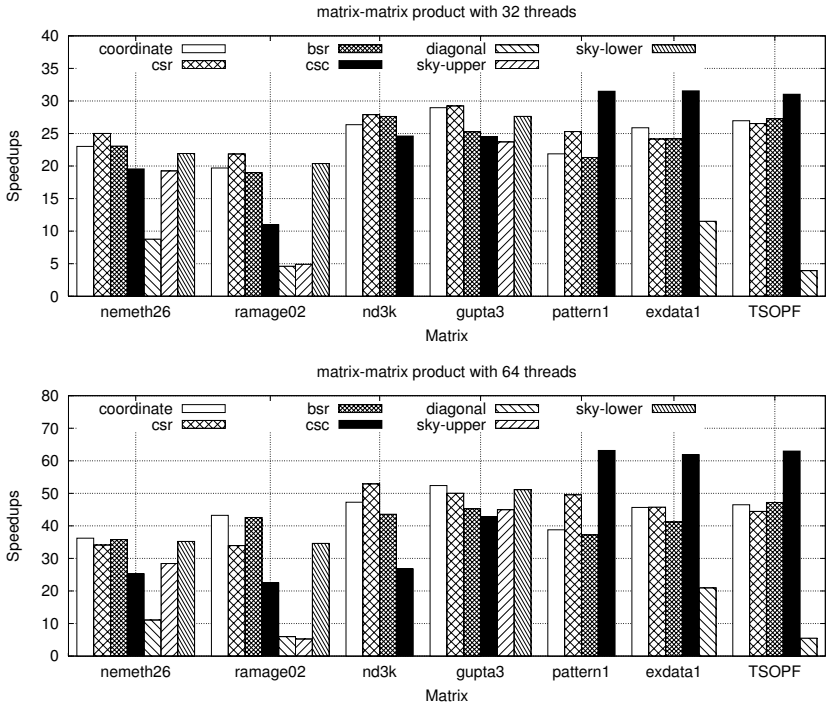


Fig. 4 Speedups of the matrix-matrix product

of data to the output array. As can be seen in Figure 2, the first approach gathers the output data with only one bulk copy per thread, which in UPC is more efficient than using one bulk copy per row and thread, as in the second approach. However, if the matrix presents a very irregular sparsity pattern such as `TSOPF`, `exdata_1` or `pattern1`, the distribution by rows tries to balance the number of non-zero elements per thread and this leads to a different number of rows per thread (see Section 3). Therefore, the computational workload of the final additions and data copies is very unbalanced, obtaining less efficiency than distributing the dense matrices by columns. Finally, the poor speedups of the DIA format for this routine is due to the fact that the sequential times are lower than using other formats thanks to the efficient exploitation of the memory hierarchy provided by this format on the evaluated matrices. Nevertheless, when the data in the DIA format is distributed among several threads this cache efficiency decreases, showing significantly poorer scalability.

The column distribution was initially only implemented for CSC, DIA and SKY with upper matrices. However, as the sparse matrix is not distributed, this approach can be applied to all the formats. Experiments with the formats COO, CSR, BSR and SKY with lower matrices using this distribution were performed and similar results to CSC were obtained for all the matrices. These results were not included due to space limitations. Thus, these storage formats

that order the non-zero elements consecutively by rows could always obtain the best performance also in the sparse matrix-matrix product if the appropriate distribution is chosen. The column distribution should be the approach to be applied in case of sparse matrices with very irregular sparsity pattern. Otherwise, the approach with the row distribution is the best choice.

5 Conclusions

This paper has presented the performance evaluation of the sparse matrix products in UPC using six representative sparse storage formats. The algorithms proposed have taken into account the influence of the storage format on the data distribution in order to achieve a high efficiency. Some optimization techniques for UPC codes such as using private pointers to access shared memory with affinity to the thread or minimizing the number of bulk copies in the remote data movements have also been included.

The evaluation was performed using sparse matrices with different characteristics and the experimental results have shown that the speedups depend not only on the storage format but also on the sparsity pattern of the matrix. The variability of the efficiency among the storage formats is very significant. Furthermore, we can assert that formats that store the non-zero elements consecutively by rows (Coordinate, Compressed Sparse Row, Block Sparse Row and Skyline with lower matrices) are able to obtain the same or better performance than the other ones in all scenarios. All these row-based formats achieve for the sparse matrix-vector product speedups between 40 and 63 on 64 threads even though the sequential times are very low. Regarding the sparse matrix-matrix product, speedups between 37 and 63 are obtained if the appropriate distribution according to the sparsity pattern of the matrix is selected.

As future work, the sparse matrix products will be included in the UPCBLAS library [7]. Although some changes in the interface will be necessary to adapt them to this library (for instance, input and output matrices distributed among all threads) the experimental evaluation done in this work will be very helpful for the choice of the most appropriate storage formats.

Acknowledgements This work was funded by Hewlett-Packard (Project "Improving UPC Usability and Performance in Constellation Systems: Implementation/Extensions of UPC Libraries"), the Ministry of Science and Innovation of Spain (Project TIN2010-16735), the Ministry of Education (FPU grant AP2008-01578), and the Spanish network CAPAP-H3 (Project TIN2010-12011-E). We gratefully thank CESGA (Galicia Supercomputing Center) for providing access to the Finis Terrae supercomputer.

References

1. Barton C, Casçaval C, Almási G, Zheng Y, Farreras M, Chatterjee S, Amaral JN (2006) Shared Memory Programming for Large Scale Machines. In: Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'06), Ottawa, Canada, pp 108–117

2. Bell C, Nishtala R (2004) UPC Implementation of the Sparse Triangular Solve and NAS FT. (Last visit: April 2012) http://www.cs.berkeley.edu/~rajeshn/pubs/bell_nishtala_spts_ft.pdf
3. Bell C, Bonachea D, Nishtala R, Yelick K (2006) Optimizing Bandwidth Limited Problems using One-Sided Communication and Overlap. In: Proc. 20th Intl. Parallel and Distributed Processing Symp. (IPDPS'06), Rhodes Island, Greece
4. Buluç A, Gilbert JR (2008) Challenges and Advances in Parallel Sparse Matrix-Matrix Multiplication. In: Proc. 37th Intl. Conf. on Parallel Processing (ICPP'08), Portland, OR, USA, pp 503–510
5. Dongarra J (2000) Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide, SIAM, chap 10
6. El-Ghazawi T, Cantonnet F (2002) UPC Performance and Potential: a NPB Experimental Study. In: Proc. 15th ACM/IEEE Conf. on Supercomputing (SC'02), Baltimore, MD, USA
7. González-Domínguez J, Martín MJ, Taboada GL, Touriño J, Doallo R, Mallón DA, Wibecan B (2012) UPCBLAS: A Library for Parallel Matrix Computations in Unified Parallel C. Concurrency and Computation: Practice and Experience (Available Online doi:10.1002/cpe.1914)
8. Hugues MR, Petiton SG (2010) Sparse Matrix Formats Evaluation and Optimization on a GPU. In: Proc. 12th IEEE Intl. Conf. on High Performance Computing and Communications (HPCC'10), Melbourne, Australia, pp 122–129
9. Jiogo CD, Manneback P, Kuonen P (2006) Well Balanced Sparse Matrix-Vector Multiplication on a Parallel Heterogeneous System. In: Proc. 8th IEEE Intl. Conf. on Cluster Computing (CLUSTER'06), Barcelona, Spain
10. Liu S, Zhang Y, Sun X, Qiu R (2009) Performance Evaluation of Multithreaded Sparse Matrix-Vector Multiplication using OpenMP. In: Proc. 11th IEEE Intl. Conf. on High Performance Computing and Communications (HPCC'09), Seoul, Korea, pp 659–665
11. Luján M, Usman A, Freeman TL, Gurd JR (2005) Storage Formats for Sparse Matrices in Java. In: Proc. 5th Intl. Conf. on Computational Science (ICCS'05), Atlanta, GA, USA, pp 364–371
12. Mallón DA, Taboada GL, Teijeiro C, Touriño J, Fraguela BB, Gómez A, Doallo R, Mouriño JC (2009) Performance Evaluation of MPI, UPC and OpenMP on Multicore Architectures. In: Proc. 16th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'09), Espoo, Finland, pp 174–184
13. Nishtala R, Hargrove PH, Bonachea D, Yelick K (2009) Scaling Communication-Intensive Applications on BlueGene/P Using One-Sided Communication and Overlap. In: Proc. 23rd Intl. Parallel. and Distributed Processing Symp. (IPDPS'09), Rome, Italy, 2009
14. Shahnaz R, Usman A, Chughtai IR (2006) Implementation and Evaluation of Parallel Sparse Matrix-Vector Products on Distributed Memory Parallel Computers. In: Proc. 8th IEEE Intl. Conf. on Cluster Computing (CLUSTER'06), Barcelona, Spain
15. Shan H, Blagojević F, Min SJ, Hargrove P, Jin H, Fuerlinger K, Koniges A, Wright NJ (2010) A Programming Model Performance Study using the NAS Parallel Benchmarks. Scientific Programming 18(3-4):153–167
16. Shan H, Wright N, Shalf J, Yelick K, Wagner M, Wichmann N (2011) A Preliminary Evaluation of the Hardware Acceleration of the Cray Gemini Interconnect for PGAS Languages and Comparison with MPI. In: Proc. 2nd Intl. Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems (PMBS'11), Seattle, WA, USA, pp 13–14
17. Sparse Basic Linear Algebra Subprograms (SparseBLAS) Library (Last visit: April 2012) http://math.nist.gov/spblas
18. The University of Florida Sparse Matrix Collection (Last visit: April 2012) http://www.cise.ufl.edu/research/sparse/matrices/
19. Usman A, Luján M, Freeman L, Gurd JR (2006) Performance Evaluation of Storage Formats for Sparse Matrices in Fortran. In: Proc. 8th IEEE Intl. Conf. on High Performance Computing and Communications (HPCC'06), Munich, Germany, pp 160–169
20. Williams S, Oliker L, Vuduc RW, Shalf J, Yelick K, Demmel J (2007) Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms. In: Proc. 20th ACM/IEEE Conf. on Supercomputing (SC'07), Reno, NV, USA