

SparseBLAS Products in UPC: an Evaluation of Storage Formats

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

Computer Architecture Group, Department of Electronics and Systems,
University of A Coruña, Spain

Abstract

The performance of a significant number of applications in High Performance Computing (HPC) is determined by the efficiency of the sparse matrix-vector and matrix-matrix products. These computational kernels generally present poor scalability due to the lack of memory locality exploitation. Selecting the most appropriate storage format, which generally depends on the specific application scenario, can significantly improve their efficiency. This paper presents an evaluation of the most common sparse storage formats using Unified Parallel C (UPC). UPC is a Partitioned Global Address Space (PGAS) language that provides high programmability and performance through an efficient exploitation of data locality, especially on hierarchical architectures such as multicore clusters. Different combinations of storage formats and data distributions for the SparseBLAS matrix products are analyzed. Experimental results on an HP supercomputer using representative sparse matrices show that a suitable combination of storage formats and parallel algorithms has a great influence on the performance of the sparse products.

Sparse Matrices, Storage Formats, SparseBLAS, PGAS, UPC

1 Introduction

Sparse matrices are pervasive in many scientific and engineering areas, and the efficiency in their processing is critical for the performance of many applications. Many storage formats have been proposed to represent them. The minimization of the storage requirements is not the only goal of these formats, but also the computational efficiency in sparse matrices operations. Thus, 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.

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

UPC is an extension of ANSI C for parallel computing. In [2] 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 [3]. Barton et al. [4] 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 [5, 6] have confirmed this analysis.

This paper presents an evaluation of the most suitable combinations of representative sparse storage formats (Coordinate, Compressed Sparse Row, Block Sparse Row, Compressed Sparse Column, Diagonal and Skyline) and parallel algorithms for the implementation of the SparseBLAS matrix-vector and matrix-matrix products using UPC. SparseBLAS products are core routines for most iterative solvers and matrix factorizations and thus their performance has a great influence on a wide variety of scientific and engineering applications.

The rest of this paper is organized as follows. Section 2 summarizes the related work. Section 3 describes the sparse storage formats evaluated. Sections 4 and 5 outline the different algorithms used to perform the sparse matrix-vector and matrix-matrix products, respectively, depending on the storage format. Section 6 presents the analysis of the experimental results obtained on an HP supercomputer (Finis Terrae). Finally, conclusions are discussed in Section 7.

2 Related Work

Due to the significant presence and impact in science and engineering of the sparse products, several optimization techniques have been proposed for their parallel implementation. Williams et al. [7] provide an efficient implementation of the matrix-vector product for multicore systems using the Compressed Sparse Row format by applying thread blocking together with sequential optimizations such as cache blocking, loop optimizations or software memory prefetching. Liu et al. [8] provide another implementation for the Block Sparse Row format using OpenMP. This work also evaluates three different types of load balancing, determining that the non-zero scheduling presented in [9] usually obtains the best performance. A new method for load balancing for the sparse matrix-vector product in heterogeneous systems was presented in [10].

Regarding the sparse matrix-matrix product, Buluc and Gilbert [11] compare different algorithms and data distributions for the multiplication of two sparse matrices. However, this routine is not the same as the one in the SparseBLAS library [12], which multiplies a sparse matrix by a dense one. Our paper will analyze this latter one together with the sparse matrix-vector product.

The selection of the most suitable storage format is one of the main decisions in order to perform efficient products, and this decision can be influenced by the size of the problem, the sparsity pattern of the matrix, the programming language or the architecture of the system. In [13] Luján et al. presented a performance evaluation of different storage formats for the sparse matrix-vector product in Java. This study was complemented in [14] with a similar evaluation using Fortran. Regarding parallel computing, Shahnaz et al. provide in [15] and [16] a comparison of the performance of the sparse matrix-vector product with seven different formats in a small cluster using MPI. Similar studies for GPUs are presented in [17] and [18].

Nevertheless, none of these works take advantage of the use of PGAS languages. Bell and Nishitani [19] deal with sparse matrices in UPC but restricted to the sparse triangular solver and the Compressed Sparse Row format. Therefore, the novelty of our work in the PGAS programming model

area is twofold: it is the first approach to the parallel sparse multiplications and it provides the first performance comparison among different sparse storage formats.

3 Sparse Matrix Storage Formats

The sparse storage formats define the structure to keep the data of the sparse matrices, and thus play an important role in achieving a good efficiency in sparse routines. This paper studies the formats described by Dongarra in [20], each of them tailored to particular sparsity patterns:

- Coordinate Format: It is the most intuitive, simple and flexible scheme to represent sparse matrices. 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) Format: This format is probably the most popular sparse representation. 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. It has an additional entry with the total number of non-zero elements in the matrix. Therefore, CSR presents a compressed view of Coordinate as the length of *rowPtr* is the total number of rows plus one instead of the total number of non-zero values.
- Block Sparse Row (BSR) Format: It is a variant of CSR, very useful for sparse matrices where the non-zero elements are grouped in blocks. It consists of dividing the matrix in a grid of blocks and keeping, for each block with non-zero entries, its values (including zeros) and the information of the position of the block within the grid according to the CSR scheme. The values are stored consecutively by blocks and, inside them, by rows.
- Compressed Sparse Column (CSC) Format: It is similar to CSR, but storing consecutively in array *values* the non-zero elements by columns, using *rows* for the row indices and *columnPtr* for keeping for each column the index of the entry in the array *rows* which is the first non-zero element of the given column.
- Diagonal Format: Many sparse matrices in scientific computing present their non-zero entries restricted to a small number of diagonals. In order to take advantage of this pattern the Diagonal scheme has been defined. In this case *values* stores consecutively all the elements of the diagonals with any non-zero element. Another 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 Format: This format has been specifically designed for sparse triangular matrices, which frequently arise when solving linear systems. 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. In both cases an additional entry with the total number of non-zero elements is needed.

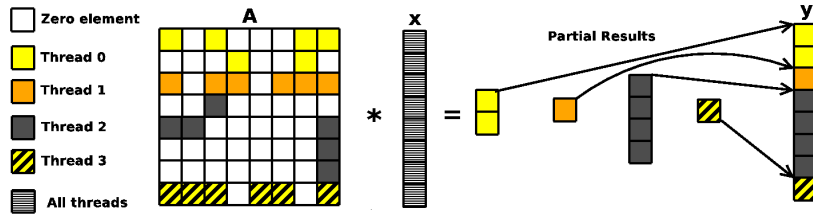


Figure 1: Sparse matrix-vector product using a row distribution for the matrix

4 Matrix-Vector Product

This section analyzes the SparseBLAS matrix-vector product: $\alpha * A * x + y = y$, where α is a scalar, A a sparse matrix and x and y dense vectors.

All PGAS languages, and thus UPC, expose a global shared address space to the user which is logically divided among threads, so each thread is associated or presents affinity to a part of the shared memory. Moreover, UPC also provides a private memory space per thread for local computations. Therefore, each thread has access to both its private memory and to the whole global space. However, the accesses to remote data will be much more expensive than the accesses to data in local memory (private memory or shared memory with affinity to the thread). Thus, data distributions will have a serious impact on the performance of the parallel codes. For the parallel implementation of the matrix-vector product three different data distributions will be considered (by rows, columns and diagonals).

Figure 1 illustrates the behavior of the sparse matrix-vector product distributing the matrix by rows. This distribution relies on the consecutive storage by rows of the data in the `values` array so it can be used in the Coordinate, CSR, BSR and Skyline (with lower matrices) formats. Previous works have pointed out that a key aspect in the performance of the sparse matrix-vector product is the computational load balance [7]. In order to achieve a good load balance the matrix is distributed by blocks of rows of different size, trying to evenly distribute the number of non-zeros per thread (in the example, six non-zero elements per thread). In order to exploit data locality as much as possible each thread only accesses the rows of the matrix that correspond to it. Then, by applying a sequential partial sparse 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, the distribution of y must match the distribution of the matrix so that the partial sums can be performed without remote accesses. Besides, in the common case that the result vector is needed completely stored in an array in the local memory of one thread, this distribution by blocks only requires one bulk copy of remote data per thread. This bulk copy is performed in one go with the `upc_memget` function which is much more efficient than copying all the elements one-by-one (the UPC default access).

For the CSC and Skyline (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 can represent an important performance overhead. The natural distribution for these formats is by blocks of columns with variable block sizes in order to achieve a good computational load balance. Figure 2 shows this distribution for the same sparse matrix used in the row example. In this case the source vector x must be always distributed according to the size of the blocks in the matrix. In order to compute the i^{th} element of the result, the i^{th} values of all partial results should be added. These

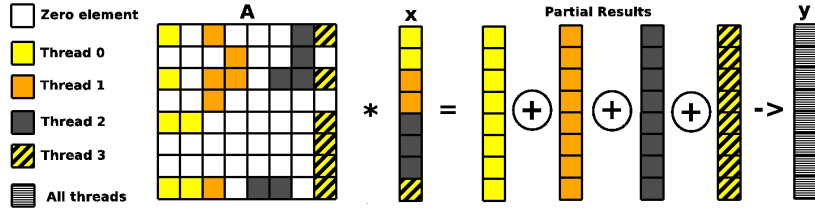


Figure 2: Sparse matrix-vector product using a column distribution for the matrix

additions need reduction operations involving all UPC threads, so their performance is usually poor.

Regarding the Diagonal format, as the elements in the `values` array are consecutively stored neither by rows nor by columns, none of the presented distributions is eligible. In this case, the sparse matrix is distributed by diagonals as shown in Figure 3 and the final reductions are also mandatory. Furthermore, as the number of non-zero elements per diagonal is unknown, the computational load might be unbalanced (in the example, seven non-zero elements for threads 0 and 1 and five non-zero elements for threads 2 and 3). Nevertheless, the impact of this drawback is alleviated by using a cyclic distribution which achieves a balanced load distribution in most sparse matrices.

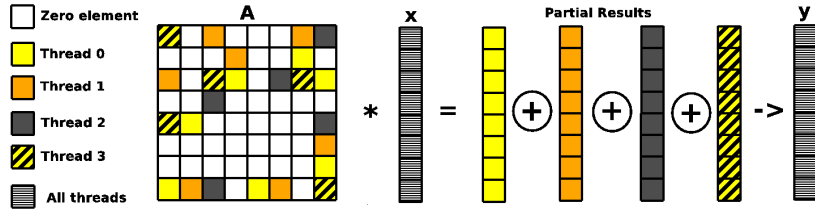


Figure 3: Sparse matrix-vector product using a diagonal distribution for the matrix

In UPC shared arrays can not be distributed with a variable block size. Thus, for all storage formats the sparse matrices are explicitly distributed into the memories of the threads using private arrays. Vectors in the matrix-vector product and dense matrices in the matrix-matrix product are instead stored in shared memory.

UPC provides functionality to access memory through pointers. A pointer to shared memory contains 3 fields: thread, block and phase. When performing pointer arithmetic on a pointer-to-shared all three fields must be updated, making the operations slower than with private pointer arithmetic. Thus, in all the implemented sparse products, 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.

5 Matrix-Matrix Product

This section focuses on the SparseBLAS matrix-matrix routine: $\alpha * A * B + C = C$, where α is a scalar value, A a sparse matrix, and B and C dense matrices.

The first approach to parallelize this kernel consists of adapting the matrix-vector distributions and algorithms to this problem. For instance, Figure 4 shows the adaptation of the row distribution for

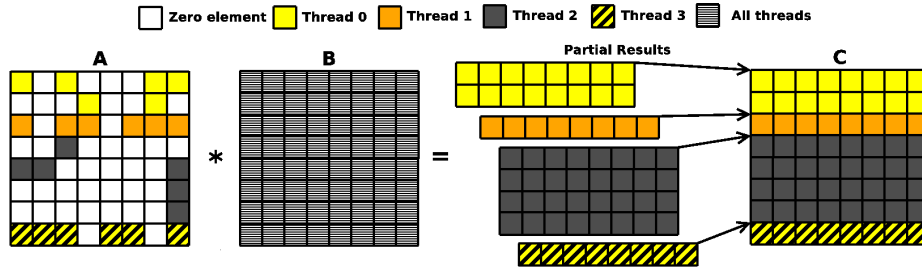


Figure 4: Sparse matrix-matrix product using a row distribution for the sparse matrix

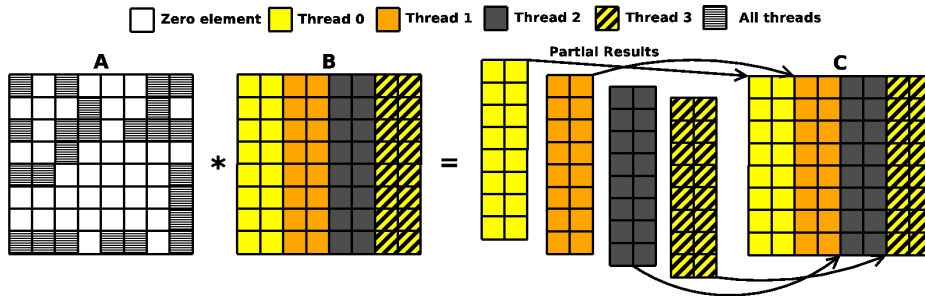


Figure 5: Sparse matrix-matrix product using a column distribution for the dense matrices

matrices with eight columns. This data distribution will be applied to the Coordinate, CSR, BSR and Skyline (with lower matrices) sparse formats. Each thread needs the whole matrix B and the same rows of C as in A . As in the matrix-vector product, only one bulk copy per thread is required in case all the elements of the result matrix need to be consecutively stored 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 5, which avoids all the reductions, has been developed for CSC, Diagonal and Skyline (with upper matrices) formats. 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 copies of all elements of the same row of C using only one call to `upc_memget` per thread and row in a scenario where the output elements must be in one array in local memory. This approach could also be used for the other sparse formats but it has been discarded because, in that scenario, it would lead to a greater number of copies (one bulk copy per row and thread) than in the row distribution of A (only one bulk copy per thread).

6 Performance Evaluation

The performance evaluation of the storage formats for SparseBLAS products in UPC has been conducted on the Finis Terrae supercomputer [21] 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, the code was compiled using Berkeley UPC 2.12.1 [22]. The intra-node and inter-node communications are performed through shared memory and GASNet over InfiniBand, respectively.

In this evaluation four representative matrices, with different sparsity characteristics, have been selected from the University of Florida Matrix Collection [23]. Their characteristics are shown in Table 1. Larger versions (labeled with “large”) have been obtained by replicating the original matrices, which preserves the sparsity and the pattern of the original ones. The larger versions have been used in the matrix-vector product whereas the original matrices have been used for the matrix-matrix product.

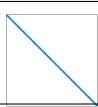
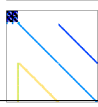
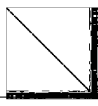
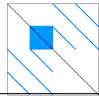
Plot	Name	Dimensions	Non-zeros	% sparsity
	nemeth26	9506x9506	760,633	0.842
	nemeth26_large	85554x85554	61,630,079	0.842
	TSOPF	18696x18696	4,396,289	1.258
	TSOPF_large	56088x56088	39,574,965	1.258
	gupta3	16783x16783	4,670,105	1.658
	gupta3_large	67132x67132	74,721,175	1.658
	exdata	6001x6001	1,137,751	3.159
	exdata_large	84014x84014	222,973,345	3.159

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

Figures 6 and 7 show the speedups of the double precision matrix-vector and matrix-matrix products, respectively, using up to 64 threads. These 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). For clarity purposes, results with less than 8 threads are not shown as there are no significant differences among the analyzed formats. Some storage formats are not appropriate for storing some matrices due to the significant number of zeros that the format would require to store them, namely `gupta3` with Diagonal, and `TSOPF` and `exdata` matrices with Skyline. Thus, these combinations have not been considered.

As expected, in the matrix-vector product row-based storage formats outperform significantly column and diagonal-based ones due to the avoidance of the final reduction operations, as shown in Section 4.

The analysis of the matrix-matrix results confirms that the differences between the two approaches presented in Section 5 is mainly due to the workload balance of the row distribution. Thus, when the workload is balanced with the row distribution approach, as in the case of `nemeth` and `gupta3` matrices, the formats that use this distribution (Coordinate, CSR, BSR and Skyline with lower matrices) are the best choice because they only need one data copy at the end of the algorithm (see Figures 4 and 5). However, for `TSOPF` and `exdata` matrices, workload is not completely balanced when relying on a row distribution because there are square blocks with a high number of non-zero elements. Therefore,

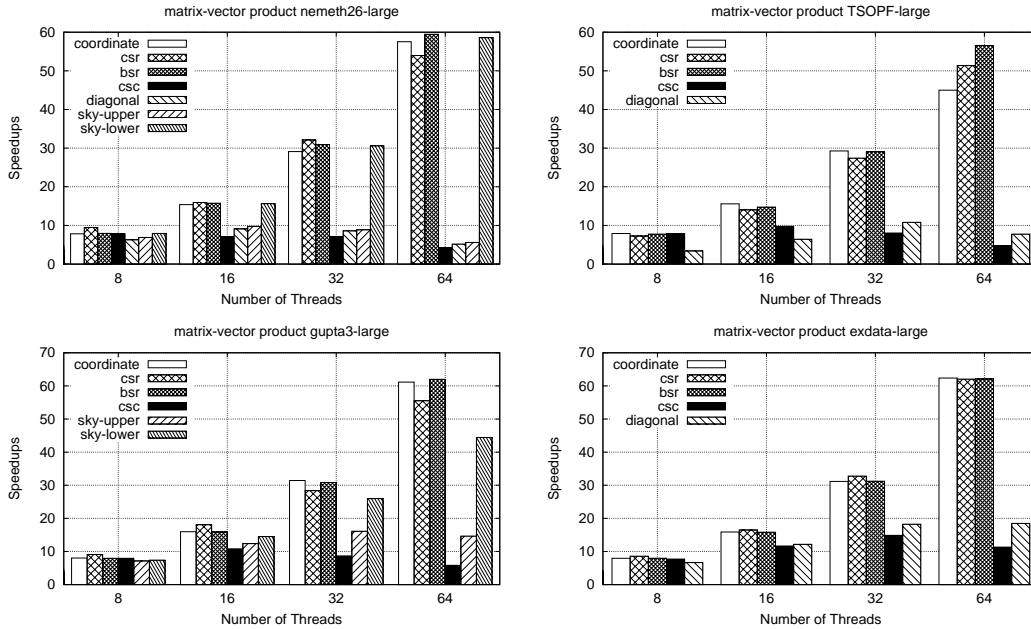


Figure 6: Speedups of the matrix-vector product

formats that distribute the dense matrices by columns show higher scalability. Finally, the poor speedups of the Diagonal 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 cache hierarchy provided by this format on the evaluated matrices. Nevertheless, when the data in the Diagonal format is distributed among several threads this cache efficiency decreases, showing significantly poorer scalability.

7 Conclusions

The efficiency of the sparse products is critical for the performance of many applications. In this paper, a PGAS language, UPC, has been used for the parallelization of these computational kernels, as it provides productivity advantages and good data locality exploitation, especially on hierarchical architectures such as multicore clusters. The parallel algorithms proposed take into account both the most suitable storage format of the sparse matrix and its influence on the data distributions in order to obtain a good efficiency. The performance evaluation of the routines on a supercomputer has shown the efficiency of the algorithms implemented, achieving speedups of up to 62 for the sparse matrix-vector product and 63 for the sparse matrix-matrix one on 64 cores. Furthermore, it has been assessed the suitability of the combination of different storage formats and workload distributions, depending on the matrix sparsity pattern.

The routines implemented will be included in a UPC sparse BLAS library to extend the dense counterpart described in [24].

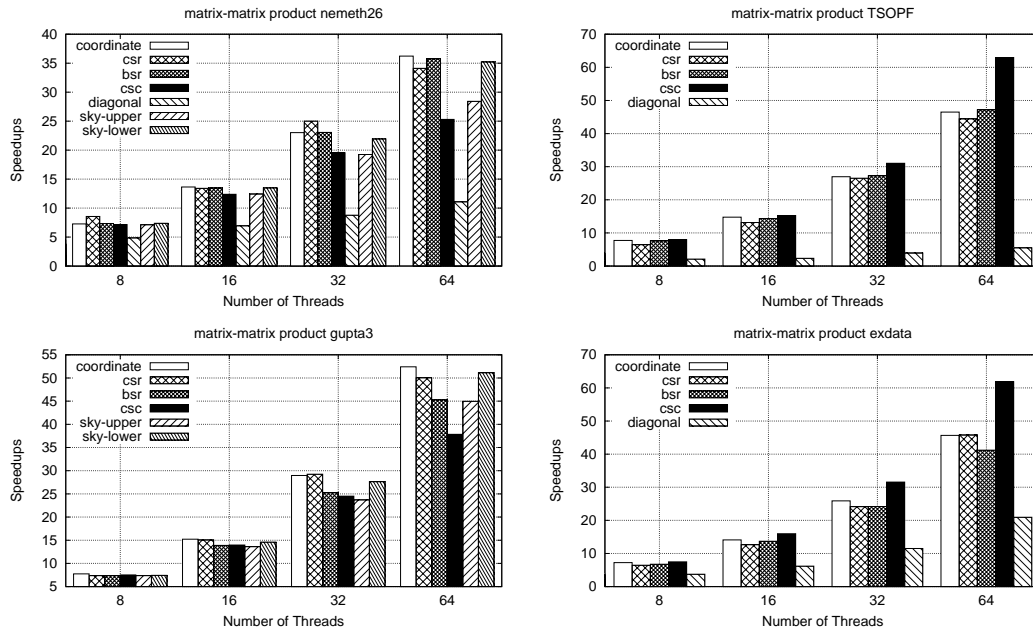


Figure 7: Speedups of the matrix-matrix product

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] UPC Consortium. UPC Language Specifications, v1.2, 2005. http://upc.lbl.gov/docs/user/upc_spec_1.2.pdf, Last visit: May 2011.
- [2] T. El-Ghazawi and F. Cantonnet. UPC Performance and Potential: a NPB Experimental Study. In *Proc. 15th ACM/IEEE Conf. on Supercomputing (SC’02)*, pages 1–26, Baltimore, MD, USA, 2002.
- [3] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick. Optimizing Bandwidth Limited Problems using One-Sided Communication and Overlap. In *Proc. 20th Intl. Parallel and Distributed Processing Symp. (IPDPS’06)*, Rhodes Island, Greece, 2006.
- [4] C. Barton, C. Casçaval, G. Almási, Y. Zheng, M. Farreras, S. Chatterjee, and J. N. Amaral. Shared Memory Programming for Large Scale Machines. In *Proc. ACM SIGPLAN Conf. on*

- Programming Language Design and Implementation (PLDI'06)*, pages 108–117, Ottawa, Canada, 2006.
- [5] D. A. Mallón, G. L. Taboada, C. Teijeiro, J. Touriño, B. B. Fraguera, A. Gómez, R. Doallo, and J. C. Mourino. Performance Evaluation of MPI, UPC and OpenMP on Multicore Architectures. In *Proc. 16th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'09)*, pages 174–184, Espoo, Finland, 2009.
 - [6] H. Shan, F. Blagojević, S.-J. Min, P. Hargrove, H. Jin, K. Fuerlinger, A. Koniges, and N. J. Wright. A Programming Model Performance Study using the NAS Parallel Benchmarks. *Scientific Programming*, 18(3-4):153–167, 2010.
 - [7] S. Williams, L. Oliker, R. W. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms. In *Proc. 20th ACM/IEEE Conf. on Supercomputing (SC'07)*, Reno, NV, USA, 2007.
 - [8] S. Liu, Y. Zhang, X. Sun, and R. Qiu. Performance Evaluation of Multithreaded Sparse Matrix-Vector Multiplication using OpenMP. In *Proc. 11th IEEE Intl. Conf. on High Performance Computing and Communications (HPCC'09)*, pages 659–665, Seoul, Korea, 2009.
 - [9] K. Kourtis, G. I. Goumas, and N. Koziris. Improving the Performance of Multithreaded Sparse Matrix-Vector Multiplication using Index and Value Compression. In *Proc. 37th Intl. Conf. on Parallel Processing (ICPP'08)*, pages 511–519, Portland, OR, USA, 2008.
 - [10] C. D. Jiogo, P. Manneback, and P. Kuonen. Well Balanced Sparse Matrix-Vector Multiplication on a Parallel Heterogeneous System. In *Proc. 8th IEEE Intl. Conf. on Cluster Computing (CLUSTER'06)*, Barcelona, Spain, 2006.
 - [11] A. Buluç and J. R. Gilbert. Challenges and Advances in Parallel Sparse Matrix-Matrix Multiplication. In *Proc. 37th Intl. Conf. on Parallel Processing (ICPP'08)*, pages 503–510, Portland, OR, USA, 2008.
 - [12] Sparse Basic Linear Algebra Subprograms (SparseBLAS) Library. <http://math.nist.gov/spblas>, Last visit: May 2011.
 - [13] M. Luján, A. Usman, T. L. Freeman, and John R. Gurd. Storage Formats for Sparse Matrices in Java. In *Proc. 5th Intl. Conf. on Computational Science (ICCS'05)*, pages 364–371, Atlanta, GA, USA, 2005.
 - [14] A. Usman, M. Luján, L. Freeman, and J. R. Gurd. Performance Evaluation of Storage Formats for Sparse Matrices in Fortran. In *Proc. 8th IEEE Intl. Conf. on High Performance Computing and Communications (HPCC'06)*, pages 160–169, Munich, Germany, 2006.
 - [15] R. Shahnaz, A. Usman, and I. R. Chughtai. 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, 2006.
 - [16] R. Shahnaz and A. Usman. Blocked-Based Sparse Matrix-Vector Multiplication on Distributed Memory Parallel Computers. *The International Arab Journal of Information Technology*, 8(2):130–136, 2011.

- [17] N. Bell and M. Garland. Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors. In *Proc. 22nd Intl. Conf. on Supercomputing (SC'09)*, Portland, OR, USA, 2009.
- [18] M. R. Hugues and S. G. Petiton. Sparse Matrix Formats Evaluation and Optimization on a GPU. In *Proc. 12th IEEE Intl. Conf. on High Performance Computing and Communications (HPCC'10)*, pages 122–129, Melbourne, Australia, 2010.
- [19] C. Bell and R. Nishtala. UPC Implementation of the Sparse Triangular Solve and NAS FT, 2004. http://www.cs.berkeley.edu/~rajeshn/pubs/bell_nishtala_spts_ft.pdf, Last visit: May 2011.
- [20] J. Dongarra. *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*, chapter 10. SIAM, 2000.
- [21] Finis Terrae Supercomputer. <http://www.top500.org/system/9500>, Last visit: May 2011.
- [22] Berkeley UPC Project. <http://upc.lbl.gov>, Last visit: May 2011.
- [23] The University of Florida Sparse Matrix Collection. <http://www.cise.ufl.edu/research/sparse/matrices/>, Last visit: May 2011.
- [24] J. González-Domínguez, M. J. Martín, G. L. Taboada, J. Touriño, R. Doallo, and A. Gómez. A Parallel Numerical Library for UPC. In *Proc. 15th Intl. European Conf. on Parallel and Distributed Computing (Euro-Par 2009)*, pages 630–641, Delft, The Netherlands, 2009.