A 2D Algorithm with Asymmetric Workload for the UPC Conjugate Gradient Method

Jorge González-Domínguez*[1], Osni A. Marques[2], María J. Martín[3], and Juan Touriño[3]

[1]Parallel and Distributed Architectures Group, Johannes Gutenberg University, Germany
[2]Computational Research Division, Lawrence Berkeley National Laboratory, CA, USA
[3]Computer Architecture Group, University of A Coruña, Spain

October 14, 2014

Abstract

This paper examines four different strategies, each one with its own data distribution, for implementing the parallel Conjugate Gradient (CG) method and how they impact communication and overall performance. Firstly, typical 1D and 2D distributions of the matrix involved in CG computations are considered. Then, a new 2D version of the CG method with asymmetric workload, based on leaving some threads idle during part of the computation to reduce communication, is proposed. The four strategies are independent of sparse storage schemes and are implemented using Unified Parallel C (UPC), a Partitioned Global Address Space (PGAS) language. The strategies are evaluated on two different platforms through a set of matrices that exhibit distinct sparse patterns, demonstrating that our asymmetric proposal outperforms the others except for one matrix on one platform.

Conjugate Gradient; PGAS; UPC; Performance Optimization; Data Distribution

1 Introduction

The Conjugate Gradient (CG) method and its variants are at the core of a number of applications. Basically, the algorithm requires the action of a matrix on a vector (i.e. a matrix-vector multiplication), together with vector updates and dot products.[1] Of particular interest is how to achieve an efficient implementation of the matrix-vector multiplication involving a sparse matrix (SpMV), since it requires irregular memory access and communication besides being the costliest part of the algorithm in terms of floating point operations. For this reason, CG has long been used for performance measurements, e.g. in the NAS Parallel Benchmarks [1]. Recently, Dongarra and Heroux have proposed a metric, the High Performance Conjugate Gradient (HPCG), as a new high performance benchmark [2]. The authors' premise is that the renowned Linpack benchmark [3] that has been the basis for the Top500

*j.gonzalez@uni-mainz.de
[1]For practical purposes, the algorithm is often used with preconditioners but this is not in the scope of this paper.

ranking of supercomputer sites [4] and influenced computer vendors is no longer indicative of how well a system can perform under realistic workloads.

The present work finds motivation in the resurgence of CG as a metric. Notably, most of the previous works in this area have focused on minimizing memory access time within SpMV using different storage formats (see Section 2). In contrast, we examine four different strategies (each one with its own data distribution) for implementing the parallel CG method in its entirety, and how they can play a role in minimizing communication and affecting performance. These four strategies can be used with any storage format, which makes our work complementary to attempts of minimizing memory access time through such formats. We implement those strategies using Unified Parallel C (UPC) [5, 6], a Partitioned Global Address Space (PGAS) language that has been shown to outperform MPI in some cases [7, 8, 9] and continues to be improved under the Dynamic Exascale Global Address Space (DEGAS) Project [10].

The rest of the paper is organized as follows. In Section 2 we summarize related work. In Sections 3 and 4 we discuss different strategies for implementing the parallel CG method. Section 5 outlines implementation issues related to the UPC CG. These implementations are evaluated on two different platforms in Section 6, through a set of matrices that exhibit distinct sparse patterns. Lastly, our findings and future work are discussed in Section 7.

2 Related Work

The NAS Parallel Benchmarks [1] consist of a set of operations that are representative of the computations and data movements involved in large-scale parallel computational fluid dynamics applications. In the benchmarks, the CG method is used to compute an approximation for the smallest eigenvalue of a sparse symmetric positive definite matrix. This enables the testing of irregular long-distance communication through matrix-vector multiplications. The benchmarks have been rewritten in various languages, including UPC [11, 12, 13]. To the best of our knowledge, UPC implementations of the CG benchmark have not attempted to reduce the overall communication overhead in the method. These implementations, for example, have focused on improving the reduction operations in the CG loop. Instead of starting with a similar implementation of CG, we resorted to a more basic version of the algorithm, for three main reasons: 1) enable experiments with matrices of distinct sparse patterns stemming from real applications; 2) enable experiments with different data distributions (as discussed in the next two sections); 3) provide conclusions useful for different implementations of the CG method (e.g. the conclusions of this paper could help to optimize the HPCG benchmark).

Several previous works focused on optimizing the SpMV due to the great impact it can have on the CG (and other similar kernels) performance. For example, strategies for optimizing SpMV have been implemented in the OSKI library [14], which consists of a set of low-level primitives that provide automatically tuned computational kernels for sparse matrices. Given a matrix, e.g. in Compressed Sparse Row (CSR) format, and a computer hardware, OSKI rearranges the data and applies code transformations to attain an optimal kernel. The tuning is performed on the fly, but the potential overhead it introduces is usually amortized by the overall performance improvements it brings to iterative processes. Another approach to improve the SpMV through a data reorganization is presented in [15]. Other works (e.g. [16, 17, 18, 19]) proposed formats that exploit regularities or substructures within the matrix to compress the structure of non-zero elements of the matrix and then minimize the impact of memory accesses. A study about the performance of different SpMV approaches in UPC was performed in [20], which shows that the efficiency of different storage formats can vary significantly.

Only few works have studied the optimization of the CG method as a whole. Ismail [21] analyzes the complexity of the different parts of the algorithm and establishes a dependency graph that helped us to devise a novel 2D distribution with asymmetric workload. To the best of our knowledge, the most comprehensive study of different strategies and data distributions for the CG method on clusters is given in [22], where the authors conclude that the 2D blocked distribution is the best approach. We will take this work as basis for comparisons in our experimental evaluation.

3 Parallel Conjugate Gradient

If A is an $n \times n$ sparse symmetric positive-definite matrix and v and b vectors, the CG method can theoretically reach a solution v for the system $Av = b$ in n iterations. Algorithm 1 shows the basic steps of the CG method. It starts with a random initial guess of the solution v. Then, it performs a loop where a new approximate solution v is obtained in each iteration until the maximum number of iterations MAX_ITER is reached or the residual r_{norm} is small enough (which indicates that v has satisfied the convergence criterion). MAX_ITER and the tolerance for the residual (r_{tol}) are specified by the user. In order to calculate the approximate solution and corresponding residual, one SpMV, two dot products (DOT) and three vector updates (AXPY) are performed per iteration.

```
1  $w = b$ ; $x = b$ ; $v = 0$
2  $r_{norm} = w^T w$                                   (DOT)
3  while $\Big(i < MAX\_ITER) \parallel (r_{norm} > r_{tol})\Big)$ do
4      $y = Ax$                                         (SpMV)
5      $\alpha = r_{norm}/(x^T y)$                      (DOT)
6      $v = v + \alpha x$                               (AXPY)
7      $w = w - \alpha y$                               (AXPY)
8      $r_{prev} = r_{norm}$
9      $r_{norm} = w^T w$                               (DOT)
10     $\beta = r_{norm}/r_{prev}$
11     $x = w - \beta x$                                (AXPY)
12 end
```

Algorithm 1: Idealized pseudo-code for the CG method.

The SpMV complexity is bounded by $O(n \times nz)$, being nz the maximum number of non-zero elements per row, whereas the DOT and AXPY complexities are $O(n)$. Therefore, most of the computing time is spent within SpMV. Parallel implementations of the CG method distribute the matrix and the vectors among threads. Typically, these (sub)vectors have the same distribution so that the DOT and AXPY operations are performed without remote accesses. The next subsections examine the adequacy of UPC for implementations of the CG method presented in previous works, with focus on the following aspects: 1) how the distribution of the matrix determines the distribution of the vectors; 2) the remote accesses that are necessary due to the distribution of the vectors; 3) the influence these remote accesses have on the performance of a parallel CG method implemented in UPC.

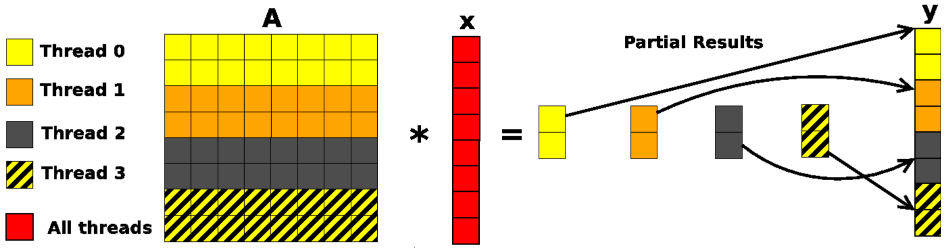


Figure 1: SpMV with a distribution of A by rows.

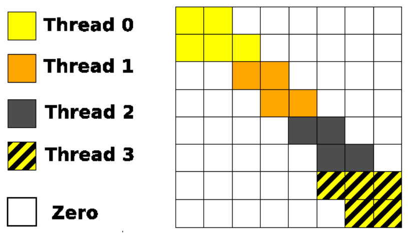


Figure 2: Example of a sparse matrix that allows the reduction of remote accesses to the vector x with a distribution of A by rows.

3.1 One-Dimensional Distribution by Rows

The most common storage formats to represent sparse matrices, such as Compressed Sparse Row (CSR) or Blocked Compressed Sparse Row (BCSR), order the non-zero elements consecutively by rows. A matrix A can be easily distributed by rows to take advantage of this feature. Figure 1 sketches the product $Ax = y$ in this case (for clarity, we do no use any specific storage format for A). Each thread performs a multiplication of its local rows by the whole vector x and the (sub)result is kept in the positions of y that correspond to the rows of A.

The vector y is distributed in the same way as the rows of A to avoid remote accesses when saving the results of the SpMV. As previously noted, all other vectors will have the same distribution to avoid remote accesses in DOT and AXPY. This implies remote accesses to x prior to SpMV. In the worst-case scenario (i.e. a dense matrix), all threads need to replicate the whole vector x, and this is usually performed through gather and broadcast collectives (with a synchronization between them). However, for most sparse matrices each thread only needs a subset of the elements of the vector. For instance, for a sparse matrix as the one shown in Figure 2, each thread only needs to copy 3 elements of the vector x. These elements are copied with one-sided communications. The main drawback of this approach is that all threads perform the copies of the subset of x at the same time. When the number of threads increases, the communication time increases due to two factors. On the one hand, the vector x is distributed among more threads, so each thread performs more copies of fewer elements each. This is less efficient than fewer copies of more elements. On the other hand, more threads perform remote copies at the same time. This implies more messages sharing the network resources, which usually decreases the efficiency of the copies.

Additionally, for a large number of threads the efficiency of the DOT products decreases: the

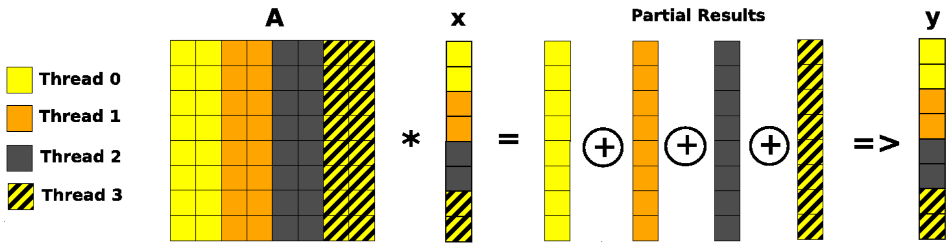


Figure 3: SpMV with a distribution of A by columns.

increase of the overhead due to the reduction of the subresults among all threads is more significant than the decrease of time since each thread performs fewer products.

3.2 One-Dimensional Distribution by Columns

Figure 3 sketches the product $Ax = y$ when A is distributed by columns. In this case the vectors follow the same distribution as the columns of the matrix to avoid remote accesses to the vector x within SpMV. As a result, each thread generates an entire vector of length n, although this vector is only a partial sum that needs to be added to the partial results of the other threads. After this addition (usually performed through an array-based reduction) and one synchronization, each thread takes the corresponding values of y for subsequent calculations in CG with one-sided copies, similarly to the copy of the necessary parts of x as illustrated in Figure 1. Consequently, the column-based approach is always less efficient than the row-based one, as it adds the overhead of the array-based reduction.

3.3 Two-Dimensional Distribution

A two-dimensional approach that reduces communication on distributed memory machines was presented in [22]. Figure 4 illustrates this approach when the processes are arranged in a $P \times Q$ grid. As in the row-based approach, each thread copies to private memory some elements of vector x. However, in this case the P threads that belong to the same column of the grid take the same elements of x. Now, the Q threads within the same row perform independent calculations by columns, working only with its subset of x. After ensuring (through synchronization) that all grid rows have reduced their partial results, all threads take the elements of y necessary to parallelize the other pieces of the CG method among all threads.

This 2D approach requires remote accesses for both x and y, but it has been shown in [22] to obtain better performance than the 1D algorithms thanks to a reduction in communications. Being T the total number of threads and A an $n \times n$ matrix, [22] shows that communications decrease from $\frac{(T-1)n}{T}$ for the 1D distributions to $\frac{(\sqrt{T}-1)n}{T}$ for the 2D distribution with a $\sqrt{T} \times \sqrt{T}$ grid. The reason is that both the initial replication of x and the final reduction of y involve fewer threads and fewer elements.

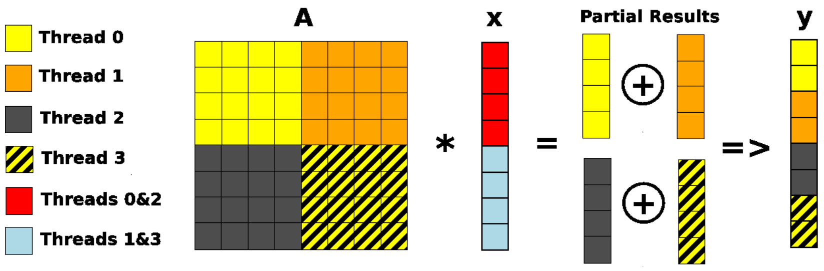


Figure 4: SpMV with a 2D distribution of A.

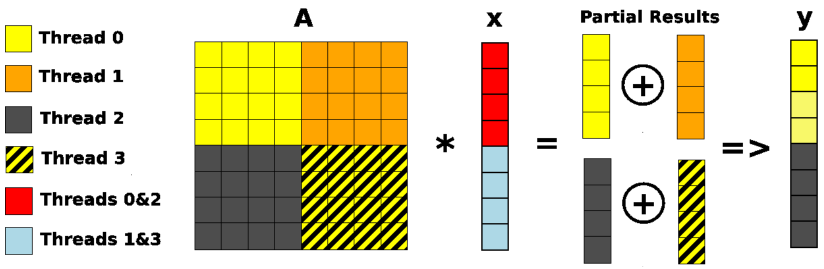


Figure 5: SpMV with a 2D distribution of A and asymmetric workload

4 Two-Dimensional Conjugate Gradient with Asymmetric Workload

As discussed in the previous section, a 2D SpMV can help to reduce the time spent at the remote copies of x. However, it adds three sources of overhead compared to the row-based approach: 1) the reduction of the subsets of y among the threads within the same row of the grid; 2) the synchronization of the threads in the same row after this reduction; and 3) the redistribution of y so that the DOT and AXPY routines can be parallelized among all threads.

In an attempt to avoid the last two drawbacks, a new algorithm is proposed. Figure 5 illustrates it: the main difference from Figure 4 is that there is no redistribution of the vector y. Once the subsets are reduced, SpMV finishes with the data of y only distributed among the P threads that belong to the first column of the grid. Therefore, after the product, all the DOT and AXPY routines are only parallelized among these threads of the first column, the other threads remaining idle. The main advantage of this approach is the avoidance of the synchronization and the redistribution among all threads at the end of the SpMV. Additionally, only P threads are involved in the final reduction within the DOT products, so the overhead is less significant. Last but not least, as all vectors follow the same distribution, x is only shared among P threads. Consequently, there are fewer blocks of larger size and thus the copies of the elements of vector x at the beginning of the SpMV are more efficient.

The main drawback of this algorithm is that the DOT and AXPY routines are parallelized only among P threads. Therefore, resources remain unused. However, as discussed in Section 3, the complexity of SpMV is typically higher than DOT or AXPY. In terms of performance, the decrease of the communication overhead within the SpMV is much more significant than the time lost by not using all threads for DOT and AXPY.

5 Implementation Issues

Although various previous works have focused on the development of efficient sparse storage formats to improve memory accesses within SpMV (see Section 2), we have adopted the CSR format as it is a widely used format for sparse computations [23, 24]. As discussed in Section 1, our goal is the reduction of communications, which is complementary to the optimization of memory accesses. Therefore, all conclusions obtained in this work related to the communications within the CG method using the CSR format can be extended to other formats.

In order to increase the efficiency of the UPC CG method, a set of optimization techniques have been applied to all the algorithms to provide a fair comparison among the different approaches:

- *Balance of the non-zero elements per thread.* Our assumption is that the computational load balance during the routines with complexity $O(n)$ (DOT and AXPY) is not important. However, previous works have pointed out that this is a key aspect in the performance of SpMV [25]. Therefore, in order to achieve a good load balance in this routine, we evenly distribute the number of non-zero elements per thread, assigning blocks of rows/columns accordingly (i.e. with different sizes). Furthermore, we have chosen a block distribution for all the strategies. It is the most suitable for these algorithms as it allows to copy the subresults to their position in the vector at once (see, for instance, Figure 1). As UPC shared arrays can not be distributed with a variable block size, we have designed new structures that keep in the private memory of each thread the elements of the matrix and the vectors assigned to it.

- *Space privatization.* Experimental results in [12] and [9] have shown that the use of shared pointers increases execution times by up to several orders of magnitude in UPC. Thus, when dealing with shared data with affinity to the local thread, all our accesses are performed through standard C pointers instead of using UPC pointers to shared memory.

- *Aggregation of remote shared memory accesses.* As mentioned in Section 3.1, instead of the costly one-by-one accesses to remote elements, all our algorithms perform remote shared memory accesses through bulk copies, using the `upc_memget()` function on remote bulks of data required by a thread.

- *Minimization of remote accesses.* As mentioned in Section 3, not all threads need to access all the elements of vector x at the beginning of SpMV. Instead of replicating the whole vector as in [22], each thread only copies the elements of x that it uses.

- *Wrap to extended collectives.* The collectives in the current UPC 1.3 specification do not support operations involving a subset of threads (i.e., there is no equivalent to an MPI communicator) and do not include array-based reductions. However, the 2D approaches need an array-based reduction involving only the threads within the same rows of the grid. This limitation was addressed with a wrapper from UPC to MPI collectives developed by the Berkeley UPC Project [5].

Table 1: Characteristics of the computer platforms used in the tests.

System	Carver	Hopper
Processor	Intel Xeon 5550X Nehalem	AMD Magny-Cours
Clock rate	2.67 GHz	2.1 GHz
Peak performance per core	10.8 Gflops	8.4 Gflops
Cores per node	8	24
L1 cache	16 KB	64 KB
L2 cache	256 KB	512 KB
L3 cache	8 MB	6 MB
Memory per node	64GB	32 GB
Interconnect	4X QDR InfiniBand	Gemini 3D Torus
UPC Compiler	Berkeley UPC	Cray UPC
MPI Compiler	Intel MPI	Cray MPI

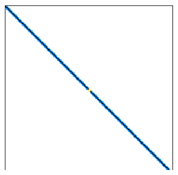
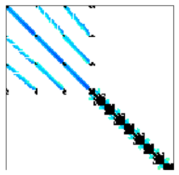
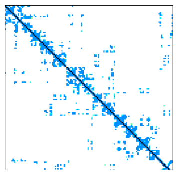
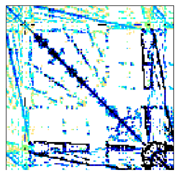
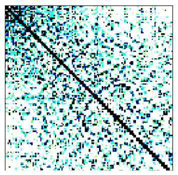
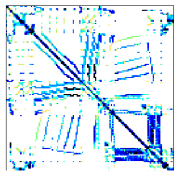
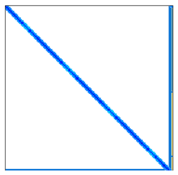
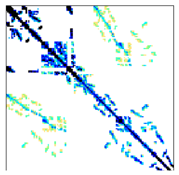
6 Experimental Evaluation

Two platforms, with different software and hardware characteristics (see details in Table 1), installed at the National Energy Research Scientific Computing Center (NERSC) were used for evaluating the algorithms discussed in the previous sections. For this experimental evaluation we have selected eight matrices, with different sparsity patterns, from the University of Florida Matrix Collection [26]. The sparsity pattern and characteristics of the matrices are shown in Table 2. As these matrices are symmetric, the Florida Matrix Collection file only contains the upper half, but all the non-zero values of both halves are stored in memory. Note that the four algorithms discussed in the previous sections are mathematically equivalent. Therefore, the execution time per iteration is enough to compare the performance of the different approaches. All the experiments used between 5,000 and 50,000 iterations (depending on the matrix) and one UPC thread per core. All graphs show the average execution time per iteration.

Firstly, the graphs of Figure 6 show the execution times per iteration of the 2D version with asymmetric workload for three representative matrices when varying the grid of threads. Specifically, we only study the three matrices in the Williams group (*cant*, *consph* and *pdb1HYS*), which are widely employed to test performance of sparse numerical computations. Results for a representative number of threads (256 for Carver and 192 for Hopper) are shown. One can observe that the most efficient grid depends not only on the characteristics of the matrix but also on the system. On the one hand, the more rows are used, the more remote copies with fewer elements each are performed at the same time to copy the bulks of x at the beginning of the SpMV (similarly to the 1D distribution by rows). On the other hand, the more columns are specified, the more threads are involved in each reduction at the end of the SpMV, as in the 1D distribution by columns. However, the impact of increasing the number of threads involved in a reduction or performing more simultaneous messages depends on the machine.

Performance results for all matrices are shown in Figure 7. The tests for the 2D versions were repeated with different grids and the graphs show the best experimental results. The execution times per iteration are shown for the number of threads that scales the best for any of the four algorithms. This number is indicated in parentheses beside the matrix name. Due to the low percentage of non-zero elements in the sparse matrices from the Florida Matrix Collection (which come from real-world

Table 2: Snapshot of the sparse matrices used in the evaluation. n is the number of rows and columns and nnz the number of non-zero elements.

 cant n=62.5K nnz=4.0M	 consph n=83.3K nnz=6.0M	 pdb1HYS n=36.4K nnz=4.3M	 crankseg n=52.8K nnz=10.6M
 nd6k n=18.0K nnz=6.9M	 ship001 n=34.9K nnz=3.9M	 cbuckle n=13.7K nnz=0.7M	 smt n=25.7K nnz=3.7M

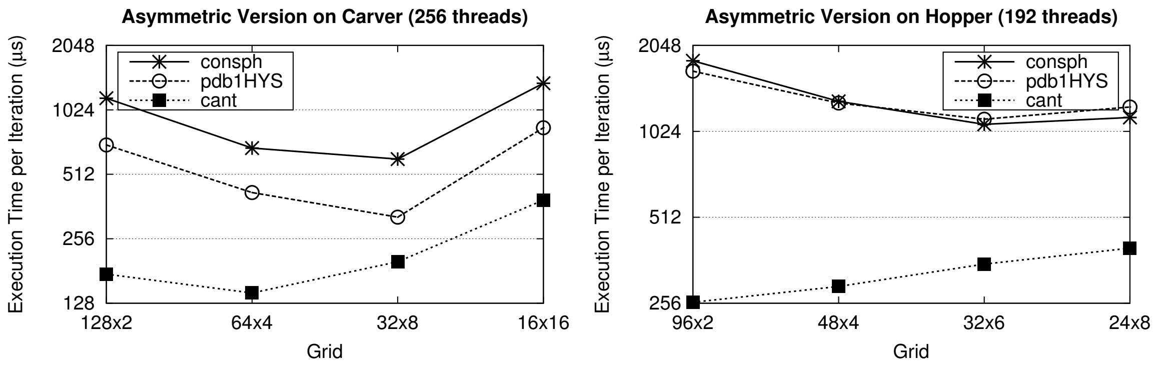


Figure 6: Evaluation of the 2D CG with asymmetric workload using different grids.

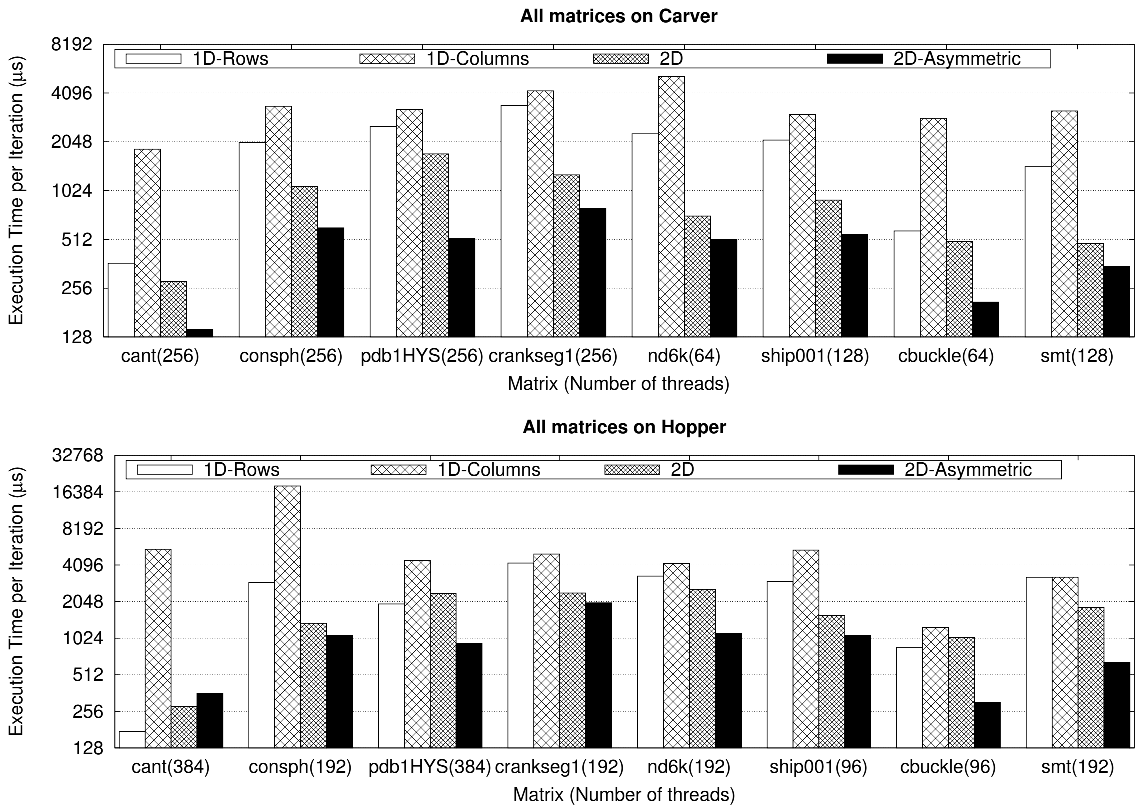


Figure 7: Comparison of the performance of the four CG implementations using different sparse matrices and platforms.

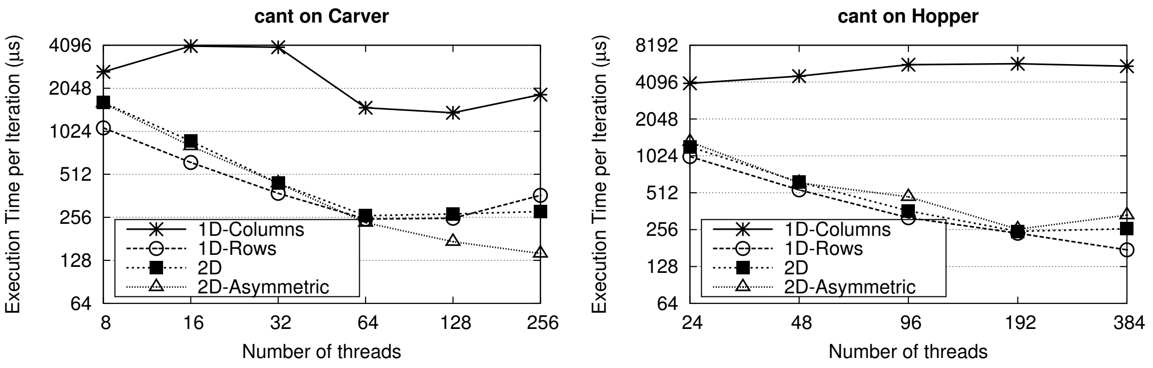


Figure 8: Evolution of the performance with the number of threads of the four CG implementations for *cant*.

problems), the execution time per iteration is very low (in terms of microseconds), which limits the scalability of the algorithms. Note that the 2D version with asymmetric workload obtains the best performance in all scenarios but for the banded matrix *cant* on Hopper. The average speedup obtained by the asymmetric approach over the second best one (the traditional 2D approach) is similar on both platforms: 1.95x and 1.99x on Carver and Hopper, respectively.

Figure 8 shows additional results using the banded matrix *cant* in order to better illustrate the behavior of the algorithms. It compares the performance of the four approaches on the two computer platforms using different number of threads, starting with a number of threads that fit in a node. Although not shown, the results with all the other matrices in both systems follow the same trend as the results in the left graph. The first conclusion that can be drawn is that the 1D distribution by columns always leads to the worst performance, which confirms that the reductions among all threads involve significant overheads. Although the 1D distribution by rows obtains the best performance for experiments with few nodes, the 2D approaches usually outperform it for larger thread counts. For instance, on Carver both 2D versions outperform the row-based algorithm for all matrices with experiments larger than 32 threads thanks to a reduction in communications. On Hopper the only exception is the experiment with the banded matrix *cant*. Nevertheless, the 2D versions obtain better performance for more unstructured matrices. Finally, the new 2D distribution with asymmetric workload always outperforms the balanced one but for the *cant* matrix on Hopper.

Figure 9 shows the time breakdown of CG for the best three algorithms on the two computer platforms using again the banded matrix *cant*. Each platform was analyzed using a small and a large number of threads. We have not included the time breakdown of the 1D distribution by columns in order to make the figure clearer (by keeping the y axis within reasonable bounds). The figure shows the time spent in the computational kernels (SpMV, DOT and AXPY), the reduction included in the SpMV of the 2D approaches (Reduce-SpMV), and the additional communications and synchronizations. Again, the left graphs can be seen as representative for all the other matrices on both computers.

For the smaller number of threads (32 and 48 on Carver and Hopper, respectively) the 1D distribution by rows is always the best choice. Nevertheless, for a larger number of cores (256 and 384 on Carver and Hopper, respectively) the overhead due to communications and synchronizations is very

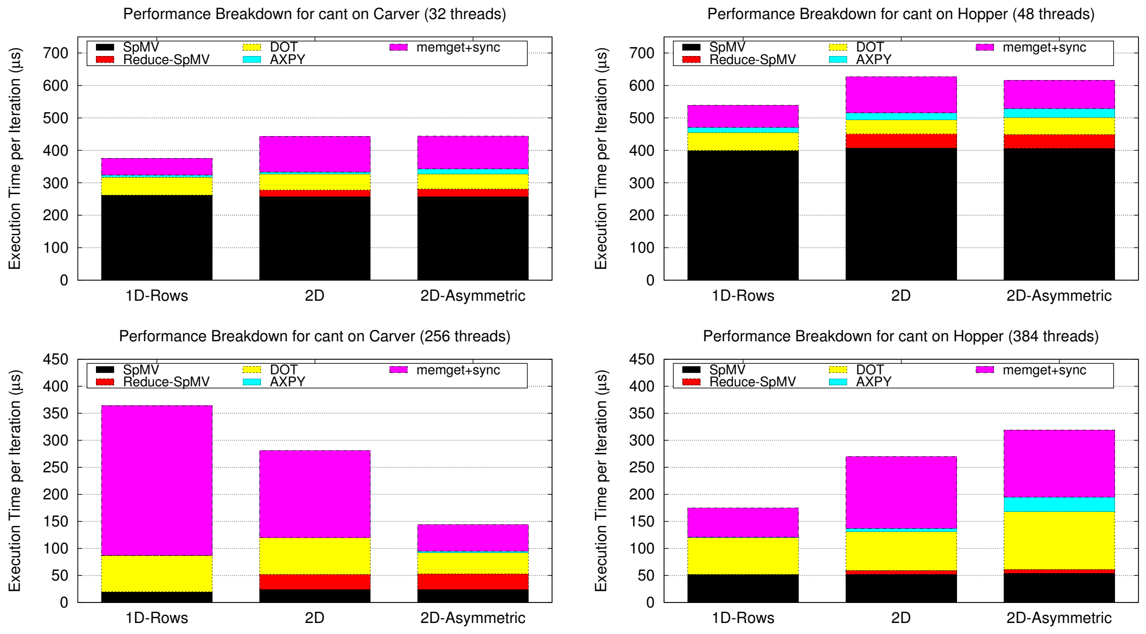


Figure 9: Time breakdown of the best three approaches for *cant*.

significant in the 1D distribution by rows, as discussed in Section 3.1. The 2D approaches reduce communication overhead even though they include a reduction within the SpMV. This overhead is further reduced with the asymmetric approach. The time breakdown of the only special case on Hopper (*cant*) is shown on the right graph. It is a banded matrix, so in the distribution by rows each thread only needs to access a small subset of the elements of x at the beginning of the SpMV. The Gemini network available on Hopper allows to perform small remote accesses with `upc_memget` faster than on the Carver InfiniBand network and thus the overhead due to the copies of x is not very significant. Consequently, the overhead due to the reduction within the 2D SpMV approaches and the additional time to perform the DOT and AXPY routines in the asymmetric version makes this strategy underperforming.

7 Conclusions

In this paper we examined four different strategies for implementing the parallel CG method in UPC, studying the impact of their data distribution on performance: two 1D approaches (with the matrix distributed by rows and by columns), a traditional 2D algorithm and a novel 2D approach where some threads are idle during part of computations to reduce communication. While most previous works focused only on the improvement of the parallel SpMV, the new approach aimed at reducing the communication overhead between the SpMV, DOT and AXPY kernels.

We tested the four approaches in two very different architectures using eight sparse matrices with distinct sparsity patterns obtained from real problems. We have shown that our proposed new strategy outperforms the others except for one matrix on one architecture with fast one-sided communications.

Machines with a larger amount of nodes and cores are becoming more and more common, so new algorithms that reduce the communication overheads must be designed. The novel 2D approach for the CG method presented in this paper is one step in the development of algorithms with asymmetric workload to reduce communications.

UPC and, in general, PGAS languages are of great importance due to their high programmability and productivity compared to message-passing counterparts as MPI. Our work will help to increase the performance of future UPC numerical routines. As future work, we would like to include our new strategy into a UPC implementation of the HPCG benchmark in order to optimize its performance. Moreover, performance models that take into account the characteristics of the machine and the sparse matrix would be useful to determine, for each scenario, which parallel CG version would lead to the best performance.

Acknowledgments

This work was funded by the Ministry of Economy and Competitiveness of Spain and FEDER funds of the EU (Project TIN2013-42148-P), by the Galician Government (Consolidation Program of Competitive Reference Groups GRC2013/055) and by the U.S. Department of Energy (Contract No. DE-AC03-76SF00098).

References

[1] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatooh, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *Intl. J. of High Performance Computing Applications*, 5:63–73, 1991.

[2] Jack Dongarra and Michael A. Heroux. Toward a New Metric for Ranking High Performance Computing Systems. Technical Report SAND2013-4744, Sandia National Laboratories, USA, 2013.

[3] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers. `http://www.netlib.org/benchmark/hpl`, Last visit: August 2014.

[4] Top500 Supercomputer Sites. http://top500.org/, Last visit: August 2014.

[5] Berkeley UPC Project. `http://upc.lbl.gov`, Last visit: August 2014.

[6] Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. *UPC: Distributed Shared-Memory Programming.* Wiley-Interscience, 2003.

[7] D. A. Mallón, A. Gómez, J. C. Mouriño, G. L. Taboada, C. Teijeiro, J. Touriño, B. B. Fraguela, R. Doallo, and B. Wibecan. UPC Performance Evaluation on a Multicore System. In *Proc. 3rd Conf. on Partitioned Global Address Space Programming Models (PGAS'09)*, Ashburn, Virginia, USA, 2009.

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

[10] The DEGAS Project. `https://www.xstackwiki.com/index.php/DEGAS`, Last visit: August 2014.

[11] 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.

[12] Tarek El-Ghazawi and François Cantonnet. UPC Performance and Potential: A NPB Experimental Study. In *Proc. 14th ACM/IEEE Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC'02)*, pages 1–26, Baltimore, MD, USA, 2002.

[13] Haoqiang Jin, Robert Hood, and Piyush Mehrotra. A Practical Study of UPC Using the NAS Parallel Benchmarks. In *Proc. 3rd Conf. on Partitioned Global Address Space Programming Models (PGAS'09)*, Ashburn, Virginia, USA, 2009.

[14] Richard Vuduc, James W Demmel, and Katherine A Yelick. OSKI: A Library of Automatically Tuned Sparse Matrix Kernels. *J. of Physics: Conference Series*, 16(1):521–530, 2005.

[15] Juan C. Pichel, Dora B. Heras, José C. Cabaleiro, A J. García-Loureiro, and Francisco F. Rivera. Increasing the Locality of Iterative Methods and its Application to the Simulation of Semiconductor Devices. *Intl. J. of High Performance Computing Applications*, 24(2):136–153, 2010.

[16] M. Belgin, G. Back, and C. J. Ribbens. Pattern-Based Sparse Matrix Representation for Memory-Efficient SMVM Kernels. In *Proc. 23rd Intl. Conf. on Supercomputing (ICS'09)*, pages 100–109, Yorktown Heights, NY, USA, 2009.

[17] K. Kourtis, G. Goumas, and N. Koziris. Optimizing Sparse Matrix-Vector Multiplication Using Index and Value Compression. In *Proc. 5th Conf. on Computing Frontiers (CF'08)*, pages 87–96, Ischia, Italy, 2008.

[18] Kornilios Kourtis, Vasileios Karakasis, Georgios Goumas, and Nectarios Kozirisl. CSX: An Extended Compression Format for SpMV on Shared Memory Systems. In *Proc. 16th ACM SIGPLAN Annual Symp. on Principles and Practice of Parallel Programming (PPoPP'11)*, pages 12–16, San Antonio, TX, USA, 2011.

[19] J. Willcock and A. Lumsdaine. Accelerating Sparse Matrix Computations via Data Compression. In *Proc. 20th Intl. Conf. on Supercomputing (ICS'06)*, pages 307–316, Cairns, Australia, 2006.

[20] Jorge González-Domínguez, Oscar García-López, Guillermo L. Taboada, María J. Martín, and Juan Touriño. Performance Evaluation of Sparse Matrix Products in UPC. *The J. of Supercomputing*, 64(1):63–73, 2012.

[21] Leila Ismail. Communication Issues in Parallel Conjugate Gradient Method Using a Star-Based Network. In *Proc. 1st Intl. Conf. on Computer Applications and Industrial Electronics (ICCAIE'10)*, Kuala Lumpur, Malasya, 2010.

[22] Fei Chen, Kevin B. Theobald, and Guang R. Gao. Implementing Parallel Conjugate Gradient on the EARTH Multithreaded Architecture. In *Proc. 6th IEEE International Conference on Cluster Computing (CLUSTER'04)*, pages 459–469, San Diego, CA, USA, 2004.

[23] Richard Barrett, Michael Berry, Tony F. Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* SIAM, Philadelphia, PA, USA, 2nd edition, 1994.

[24] Yousef Saad. *Iterative Methods for Sparse Linear Systems.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2003.

[25] 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. 19th ACM/IEEE Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC'07)*, Reno, NV, USA, 2007.

[26] The University of Florida Sparse Matrix Collection. `http://www.cise.ufl.edu/research/sparse/matrices`, Last visit: August 2014.