The Parallel Conjugate Gradient Method using Unified Parallel C

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 for implementing the parallel Conjugate Gradient (CG) method using Unified Parallel C (UPC), a Partitioned Global Address Space (PGAS) language. The paper focuses on analyzing how they impact communication and overall performance. Firstly, traditional 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 strategies are independent of sparse storage schemes. They are evaluated on a Intel Xeon-based cluster through a set of matrices that exhibit distinct sparse patterns, demonstrating that our proposed strategy outperforms the others.

Conjugate Gradient, UPC, Performance Optimization

1 Introduction

The Conjugate Gradient (CG) method and its variants are at the core of a number of applications. Basically, the algorithm requires a matrix-vector multiplication, together with vector updates and dot products.[a] 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]. Notably, most of the previous works in this area have focused on minimizing memory access time within SpMV using different storage formats [2, 3, 4]. In contrast, only few works have studied the optimization of the CG method as a whole [5, 6]. We examine four different strategies (each one with its own data distribution) for implementing the parallel CG method in its entirety, and

*j.gonzalez@uni-mainz.de

[a]For practical purposes, the algorithm is often used with preconditioners but this is not in the scope of this paper.

how they can play a role in reducing communication and affecting performance. These 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 UPC [7, 8], a Partitioned Global Address Space (PGAS) language that has been shown to outperform MPI in some cases [9, 10] and continues to be improved under the Dynamic Exascale Global Address Space (DEGAS) Project [11]. PGAS provides a data and execution model that can improve productivity and performance on highly parallel, multi-core architectures. One of the reasons is that it provides one-sided communication through put and get operations; therefore, it circumvents memory limitations that may be found in MPI. Although there are UPC implementations of the CG NAS benchmark [9, 12], to the best of our knowledge they have not attempted to reduce the overall communication overhead in the method.

The rest of the paper is organized as follows. In Sections 2 and 3 we discuss different strategies for implementing the parallel CG method. These implementations are evaluated in Section 4, using a set of matrices with distinct sparse patterns. Lastly, we summarize our findings in Section 5.

2 Parallel Conjugate Gradient

```
1  v = 0 w = b ; x = b
2  r_norm = w^T w                                                    (DOT)
3  while ( i < MAX_ITER) || (r_norm > r_tol) ) do
4  |   y = Ax                                                        (SpMV)
5  |   α = r_norm/(x^T y)                                           (DOT)
6  |   v = v + αx                                                    (AXPY)
7  |   w = w − αy                                                    (AXPY)
8  |   r_prev = r_norm
9  |   r_norm = w^T w                                                (DOT)
10 |   β = r_norm/r_prev
11 |   s = w − βx                                                    (AXPY)
12 end
```

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

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.

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

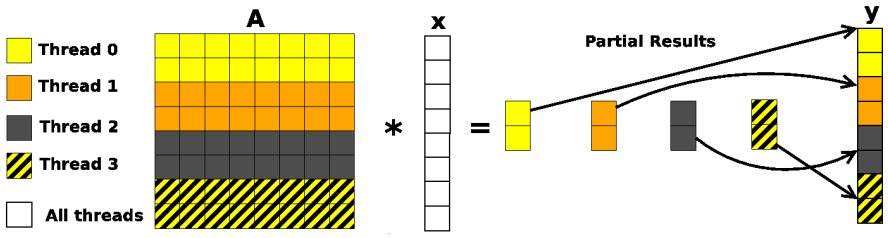


Figure 1: SpMV with a distribution of A by rows.

AXPY operations are performed without remote accesses. The next subsections examine the adequacy of UPC for the most common implementations of the CG method presented in previous works, with focus on the following aspects:

- How the distribution of the matrix determines the distribution of the vectors.
- The remote accesses that are necessary due to the distribution of the vectors.
- The influence these remote accesses have on the performance of a parallel CG method implemented in UPC.

2.1 One-Dimensional Distributions

Figure 1 sketches the product $Ax = y$ in a block distribution by rows (for clarity, we do not 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. These elements are copied with one-sided communications instead of the gather and broadcast of the whole vector, thus reducing the volume of 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. In UPC, 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 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.

Figure 2 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

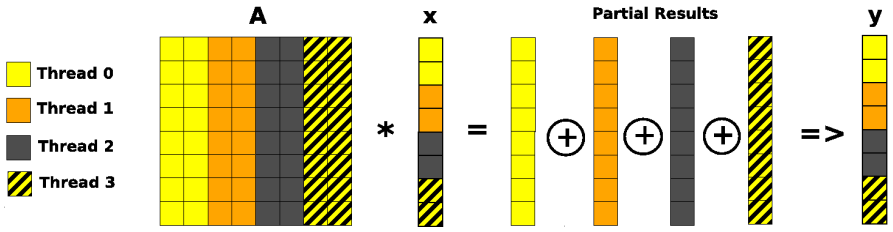


Figure 2: SpMV with a distribution of A by columns.

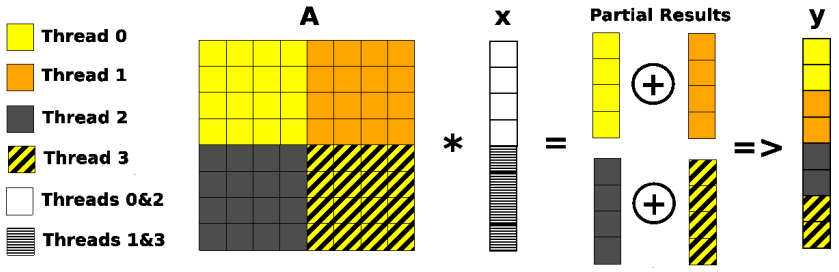


Figure 3: SpMV with a 2D distribution of A.

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.

2.2 Two-Dimensional Distribution

A two-dimensional approach that reduces communication on distributed memory machines was presented in [5]. Figure 3 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 [5] to obtain better performance than the 1D algorithms thanks to a reduction in communications. The reason is that both the initial replication of x and the final reduction of y involve fewer threads and fewer elements.

3 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:

- The reduction of the subsets of y among the threads within the same row of the grid.
- The synchronization of all threads after this reduction.
- 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 4 illustrates it: the main difference from Figure 3 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 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.

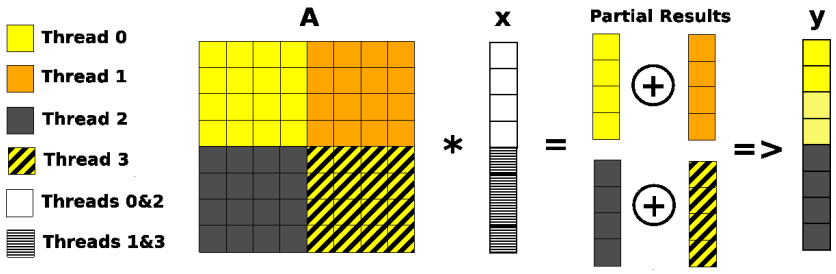


Figure 4: SpMV with a 2D distribution of A and asymmetric workload.

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 2, 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.

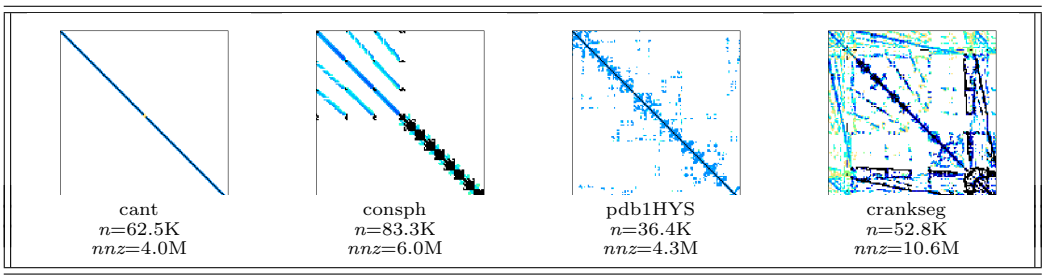
4 Experimental Evaluation

The details of Carver, the cluster installed at the National Energy Research Scientific Computing Center (NERSC) that was used for evaluating the algorithms discussed in the previous sections, are shown in Table 1. For this experimental evaluation we have selected four matrices, with different

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

System	Carver
Processor	Intel Xeon 5550X Nehalem
Clock rate	2.67 GHz
Peak performance per core	10.8 Gflops
Cores per node	8
Total number of nodes	1,120
Total number of cores	8,960
L1 cache	16 KB (private)
L2 cache	256 KB (private)
L3 cache	8 MB (shared per processor)
Memory per node	64GB
Interconnect	4X QDR InfiniBand
UPC Compiler	Berkeley UPC
MPI Compiler	Intel MPI

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.



sparsity patterns, from the University of Florida Matrix Collection [13]. 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.

Performance results for all matrices are shown in Figure 5. The execution times per iteration are shown for 128 and 256 threads. As the thread grid topology has a great influence on the performance, the tests for the 2D versions were repeated with different configurations and the graphs show the best experimental results. 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 bulk 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.

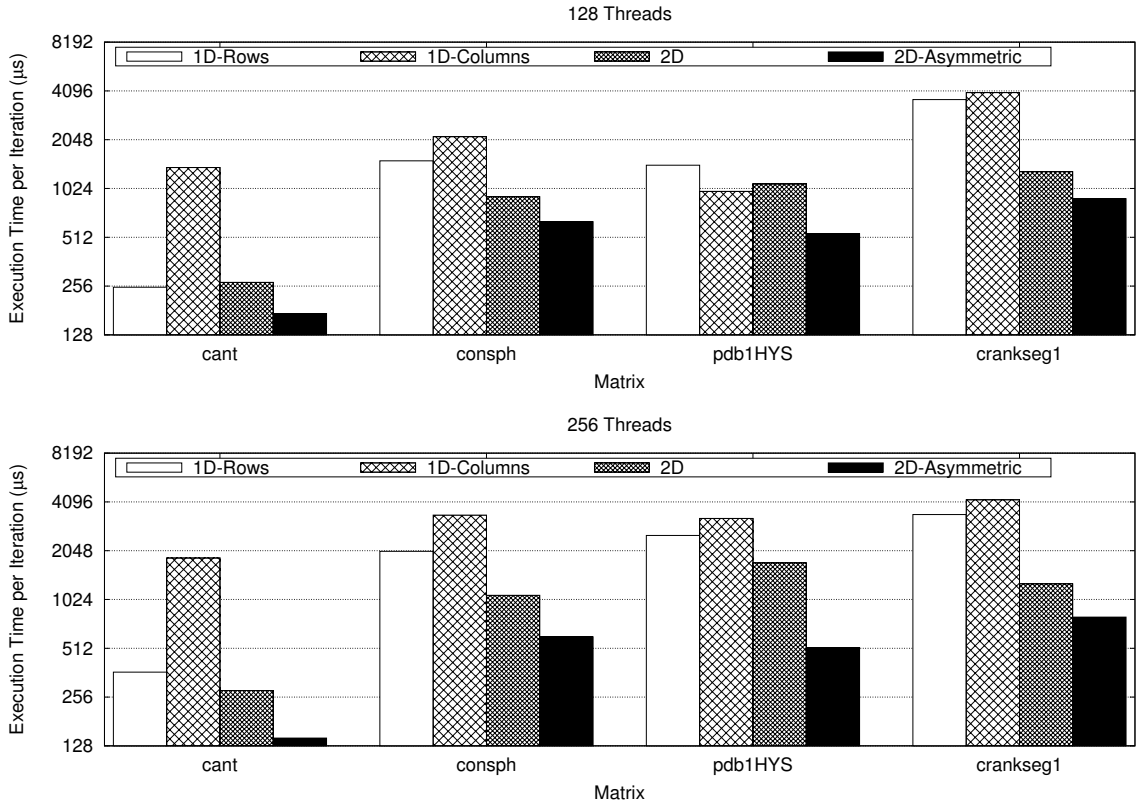


Figure 5: Comparison of the performance of the four CG implementations using different sparse matrices and numbers of threads.

The main conclusions obtained from Figure 5 is that the 2D version with asymmetric workload obtains the best performance in all scenarios and only the 2D-asymmetric version scales from 128 to 256 threads for all matrices (although its scalability is quite limited for `consph` and `pdb1HYS`). Furthermore, 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 is faster the column one, the 2D approaches outperform them for all the experiments.

5 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 some 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 on a cluster with Intel Xeon processors using four sparse matrices with distinct sparsity patterns obtained from real problems. We have shown that our proposed new strategy outperforms the others in all scenarios. In the road to the Exascale era, machines with a larger amount of nodes and cores will be developed, 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.

Acknowledgments

This work was funded by the Ministry of Economy and Competitiveness of Spain and FEDER funds of the EU (Project TIN2010-16735) and by the U.S. Department of Energy (Contract No. DE-AC03-76SF00098).

References

[1] NAS Parallel Benchmarks. `http://www.nas.nasa.gov/publications/npb.html`, Last visit: May 2014.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

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

[9] 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.

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

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

[12] 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.

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