View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by ZENODO

Available on-line at www.prace-ri.eu

Partnership for Advanced Computing in Europe

Using State-Of-The-Art Sparse Matrix Optimizations for Accelerating the Performance of Elmer

Vasileios Karakasis[1], Georgios Goumas[1], Konstantinos Nikas[2,*], Nectarios Koziris[1], Juha Ruokolainen[3], and Peter Råback[3]

[1]Institute of Communication and Computer Systems (ICCS), Greece
[2]Greek Research & Technology Network (GRNET), Greece
[3]CSC – IT Center for Science Ltd., Finland

Abstract

Multiphysics simulations are at the core of modern Computer Aided Engineering (CAE) allowing the analysis of multiple, simultaneously acting physical phenomena. These simulations often rely on Finite Element Methods (FEM) and the solution of large linear systems which, in turn, end up in multiple calls of the costly Sparse Matrix-Vector Multiplication (SpM×V) kernel. We have recently proposed the Compressed Sparse eXtended (CSX) format, which applies aggressive compression to the column indexing structure of the CSR format and is able to provide an average performance improvement of more than 40% over multithreaded CSR implementations. This work integrates CSX into the Elmer multiphysics simulation software and evaluates its impact on the total execution time of the solver. Despite its preprocessing cost, CSX is able to improve by almost 40% the performance of the Elmer's SpM×V component (using multithreaded CSR) and provides an up to 15% performance gain in the overall solver time after 1000 linear system iterations. To our knowledge, this is one of the first attempts to evaluate the real impact of an innovative sparse-matrix storage format within a 'production' multiphysics software.

1. Introduction

The number of cores in modern large scale HPC systems is growing fast, a fact attributed to the increase of both the total number of processing nodes and the number of cores within each node. However, scaling applications in current petascale and future exascale system poses a number of important challenges. Large classes of applications fail to scale either along the axis of nodes (typically by hitting the communication wall), or along the axis of cores within a node (typically by hitting the memory wall), or, even worse, along both. This work focuses on scalability issues arising within a single node when trying to utilize all the available cores. More specifically, we work on one of the most ubiquitous and challenging computational kernels, the Sparse Matrix-Vector Multiplication (SpM×V).

Multiphysics simulations are at the core of modern Computer Aided Engineering (CAE) allowing the analysis of multiple, simultaneously acting physical phenomena. These simulations often rely on Finite Element Methods (FEM) and the solution of large linear systems which, in turn, end up in multiple calls of the costly SpM×V kernel. Elmer [1] is a widely used, open source Finite Element software, and our preliminary experiments showed that 60–90% of the total execution time of the solver is actually spent in the SpM×V routine.

The major, and mostly inherent, performance problem of the SpM×V kernel is its very low *flop:byte* ratio; the algorithm must retrieve a significant amount of data from the memory hierarchy in order to perform a useful operation. In modern hardware, where the processor speed has far overwhelmed that of the memory subsystem, this characteristic becomes an overkill [2]. The widely adopted Compressed Sparse Row (CSR) storage format for sparse matrices cannot compensate for the very low flop:byte ratio of the SpM×V kernel, despite being relatively compact; it itself has a lot of redundant information.

We have recently proposed the Compressed Sparse eXtended (CSX) format [3], which applies aggressive compression to the column indexing structure of CSR. Instead of storing the column index of every non-zero element of the matrix, CSX detects dense substructures of non-zero elements and stores only the initial column index of each substructure (encoded as a delta distance from the previous one) and a two-byte descriptor of the substructure. The greatest advantage of CSX over similar attempts in the past [4, 5] is that it incorporates a variety of different dense substructures (incl. horizontal, vertical, diagonal and 2-D blocks) in a single storage

*To whom correspondence should be addressed. Email: knikas@cslab.ece.ntua.gr

format representation allowing high compression ratios, while its baseline performance, i.e., when no substructure is detected, is still higher than CSR's. The considerable reduction of the sparse matrix memory footprint achieved by CSX alleviates the memory subsystem significantly, especially for shared memory architectures, where an average performance improvement of more than 40% over multithreaded CSR implementations can be observed.

In this work, we integrate CSX into the Elmer multiphysics simulation software and evaluate its impact on the total execution time of the solver. Elmer employs iterative Krylov subspace methods for treating large problems using the Bi-Conjugate Gradient Stabilized (BiCGStab) method for the solution of the resulting linear systems. To ensure a fair comparison with CSX, we also implemented and compared a multithreaded version of the CSR used by Elmer. CSX amortized its preprocessing cost within less than 300 linear system iterations and built an up to 20% performance gain in the overall solver time after 1000 linear system iterations. To our knowledge, this is one of the first attempts to evaluate the real impact of an innovative sparse-matrix storage format within a 'production' multiphysics software.

The rest of the paper is organized as follows: Section 2. briefly describes the integration of the CSX storage format into Elmer, Section 3. presents our experimental evaluation process and the performance results, and Section 4. concludes the paper.

2. Integration of CSX into Elmer

2.1. The CSX storage format

The most widely used storage format for non-special (e.g., diagonal) sparse matrices is the Compressed Sparse Row (CSR) format. CSR compresses the row indexing information needed to locate a single element inside a sparse matrix by keeping only *number-of-rows* 'pointers' to the start of each row (assuming a row-wise layout of the non-zero elements) instead of *number-of-nonzeros* indices. However, there is still a lot of redundant information lurking behind the column indices, which CSR keeps intact in favor of simplicity and straightforwardness. For example, it is very common for sparse matrices, especially those arising from physical simulations, to have sequences of continuous non-zero elements. In such cases, it would suffice to store just the column index of the first element and the size of the sequence. CSX [3] goes even further by replacing the column indices with the delta distances between them, which can be stored with one or two bytes in most of the cases, instead of the typical four-byte integer representation of the full column indices.

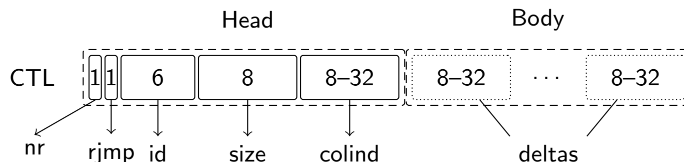


Fig. 1. The data structure used by CSX to encode the column indices of a sparse matrix.

Figure 1 shows in detail the data structure (`ctl`) used by CSX to store the column indices of the sparse matrix. The main component of the `ctl` structure is the *unit*, which encodes either a dense substructure or a sequence of delta distances of the same type. The unit is made up of two parts: the *head* and the *body*. The head is a multiple byte sequence that stores basic information about the encoded unit. The first byte of the head stores a unique 6-bit ID of the substructure being encoded (e.g., 2×2 block) plus some metadata information for changing and/or jumping rows, the second byte stores the size of the substructure (e.g., 4 in our case), while the rest store the initial column index of the encoded substructure as a delta distance from the previous one in a variable-length field. The body can be either empty, if the type ID refers to a dense substructure, or store the delta distances, if a unit of delta sequences is being encoded.

CSX supports all the major dense substructures that can be encountered in a sparse matrix (horizontal, vertical, diagonal, anti-diagonal and row- or column-oriented blocks) and can easily be expanded to support more. For each encoded unit, we use LLVM [6] to generate substructure-specific optimized code in the runtime. This adds significantly to the flexibility of CSX, which can support indefinitely many substructures, provided that only 64 are encountered simultaneously in the same matrix. The selection of substructures to be encoded by CSX is made by a heuristic favoring those encodings that lead to higher compression ratios. Detecting so many substructures inside a sparse matrix though, can be costly and this is not strange to CSX. Nonetheless, we have managed to considerably reduce the preprocessing cost without losing in performance by examining a mere 1% of the total non-zero elements using samples uniformly distributed all over the matrix [3].

2.2. CSX Integration

To integrate CSX into Elmer, we first needed to slightly modify the interface for computing sparse matrix vector products. Elmer is configured to optionally load a user supplied library for computing the sparse matrix vector multiplication product. This library is loaded at runtime and searches for a user specified function, which computes SpM×V. The original function signature of Elmer was

```
void matvec_(int *n, int *rowptr, int *colind, double *values, double *x, double *y);
```

where n is the number of rows of the matrix, *rowptr*, *colind* and *values* describe the sparse matrix in the CSR format and x and y are the input and output vectors, respectively. This interface, however, does not capture the notion of tuning the sparse matrix representation. The idea behind every tuned sparse matrix representation is that, first you tune the initial matrix, and then use that tuned matrix for the subsequent SpM×V operations [4]. Therefore the interface was changed to the following:

```
void matvec_(void **tuned, int *n, int *rowptr, int *colind, double *values, double *x,
    double *y, int *reinit);
```

Now **tuned* is a pointer to the tuned matrix representation (e.g., CSX) and the *reinit* flag serves the reconstruction of the tuned matrix, in case the original matrix changes. The construction of the final CSX matrix takes place when **tuned* is NULL or *reinit* is set and is performed in two phases. First, we build from the initial CSR matrix an internal representation that facilitates the mining of the substructures, and then we scan this internal representation for substructures and produce the final CSX matrix.

2.3. Tackling the preprocessing cost of CSX

Figure 2 shows the execution time breakdown for libcsx when using one or multiple threads on a dual SMP quad-core Intel Xeon E5405 to run the *vortex3d* test case modified to create a 1000-point mesh. This creates a 47740×47740 matrix with 4607344 nonzeros taking up 52.9 MB in CSR format. The matrix vector multiplication is performed 46 times. The execution time of single threaded CSR executed in libcsx is also presented.

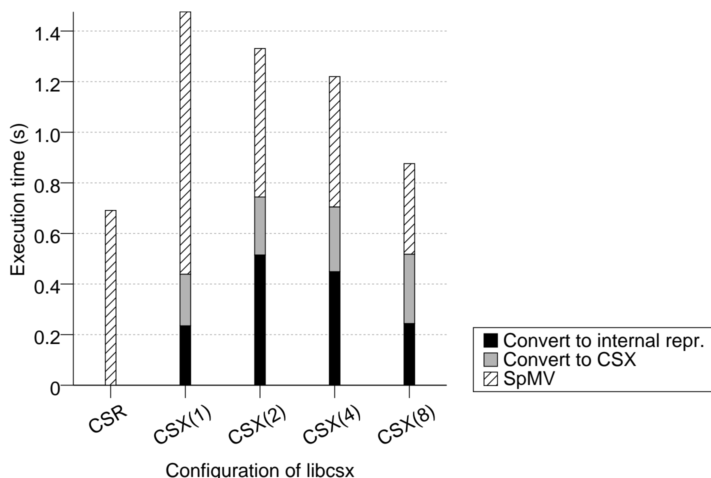


Fig. 2. Libcsx's execution time breakdown for vortex3d before applying preprocessing optimizations (in parenthesis the number of cores used).

Two key observations can be made here. First, the time required to convert CSR to the internal data structure takes up more than half of the preprocessing time. Second, the time to build CSX does not scale with the number of threads, despite being multithreaded.

In addition to the poor preprocessing performance, significant variation was observed in the preprocessing time, in spite of explicitly setting the affinity of the preprocessing threads to the different cores. Detailed profiling of the code revealed that the problem was caused by the memory allocations, performed both during the construction of the internal representation and the construction of the final CSX matrix. In both cases we used successive calls to the *realloc()* function with an allocation grain of 512 bytes for the allocations of the SpM×V data structures. For small data structures this is not a major problem; most probably, no actual reallocation will take place. For large data structures though, as is the case of those involved in SpM×V, actual reallocations will almost certainly happen, leading to huge memory copies and significant performance degradation. This problem is aggravated further with the use of multiple threads sharing the same virtual address space, since all the threads contend for very large continuous address regions. This explains not only the lack of speedup in the preprocessing phase, but also the degradation of the preprocessing performance when using 4 or 8 threads.

Having identified the performance bottleneck, the solution is straightforward: if possible, try to infer the exact allocation requirements right from the beginning; otherwise, be generous. In the first phase of the conversion, the exact size of the data structures of libcsx's internal representation of the sparse matrix can be computed directly from the size of the CSR input, and therefore we allocate it with a single large allocation. In the second phase of the construction of the final CSX matrix though, it is not possible to know the exact size of the output without iterating over the entire internal representation. To avoid unnecessary and costly iterations, we are generous with our initial allocation of the CSX's ctl structure which we set equal to the CSR's

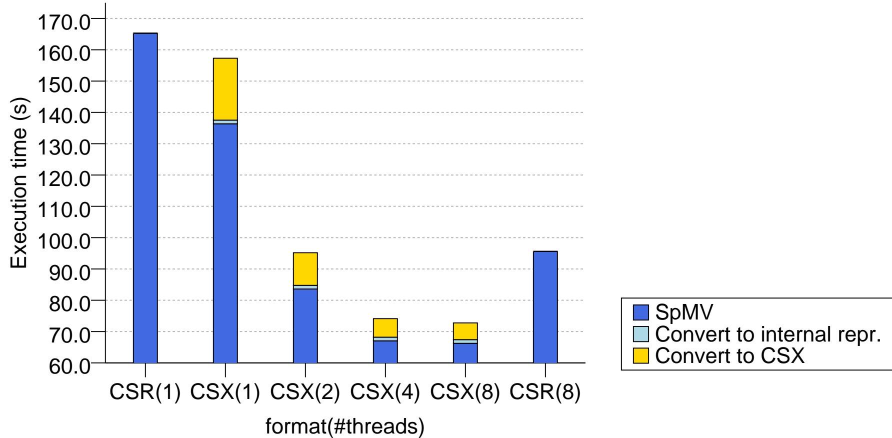


Fig. 3. Libcsx's execution time breakdown for vortex3d after applying preprocessing optimizations (in parenthesis the number of cores used).

colind. At the end, we truncate any trailing unused space with a call to *realloc()*, passing it the actual size of the structure.

Figure 3 shows the execution time breakdown on the same platform for the vortex3d problem for 2000 SpM×V calls after applying the aforementioned solution. Compared to the previous implementation, the preprocessing cost is now reasonable and consistent. The conversion time is now, as expected, minimal and constant across the different thread configurations, while the rest of the preprocessing time scales reasonably with the number of threads. Finally, it is apparent that the preprocessing cost can be easily amortized within much less than 600 linear system iterations. Indeed, our experiments showed that up to 300 linear system iterations are enough to amortize the preprocessing cost against the multithreaded CSR.

3. Experimental Evaluation

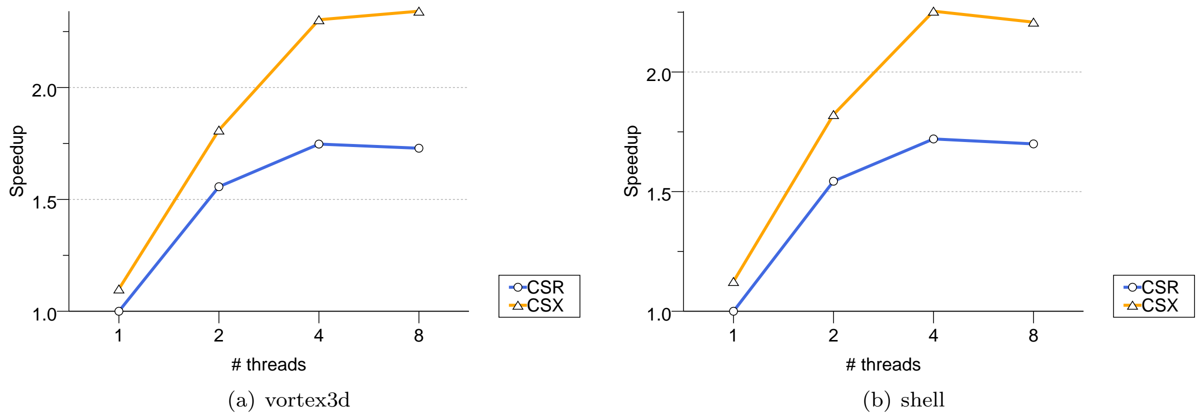
The default implementation of CSR inside Elmer is single-threaded, so we also implemented a multithreaded version to perform a fair comparison with the multithreaded CSX. The experimental platform consisted of 192 cores (24 nodes of two-way quad-core Intel Xeon E5405 processors interconnected with 1 Gbps Ethernet) running Linux 2.6.38. We used GCC 4.5 for compiling both Elmer (revision number 5477) and the CSX library along with LLVM 2.9 for the runtime code generation for CSX. Table 1 shows the 5 problems selected from the Elmer test suite for the evaluation of our integration. We appropriately increased the size of each problem to be adequately large for our system. Specifically, we opted for problem sizes leading to matrices with sizes larger than 576 MiB, which is the aggregate cache of the 24 nodes we used. Finally, we have used a simple Jacobi (diagonal) preconditioner for all the tested problems.

Table 1. The test problems used for the experimental evaluation.

Problem name	*Equations involved*	*SpM×V exec. time (%)*
fluxsolver	Heat + Flux	57.4
HeatControl	Heat	57.5
PoissonDG	Poisson + Discontinuous Galerkin	62.0
shell	Reissner-Mindlin	83.0
vortex3d	Navier-Stokes + Vorticity	92.3

Figure 4 presents the speedup achieved by multithreaded CSR and CSX within a single node of our system for two of the selected problems, namely vortex3d and shell. The results correspond to 1000 linear system iterations (equivalent to 3000 SpM×V calls) and the preprocessing cost is included in the case of CSX. For each thread count, we have selected the most favorable thread configuration for the SpM×V kernel, i.e., the one with the least possible sharing of the highest-level caches. For example, in the two-threads configuration, the threads are placed one per processor and in the four-threads one, the threads are placed one per L2 cache. This placement achieves the highest performance with the least possible threads and explains the plateau of the speedup encountered by both CSR and CSX in the eight-thread configuration, since in this case the contention in the common bus becomes apparent. In both problems, however, CSX was able to achieve considerable performance improvement over the multithreaded CSR implementation, despite its preprocessing cost. More specifically, CSX achieves a performance improvement of around 35% over CSR.

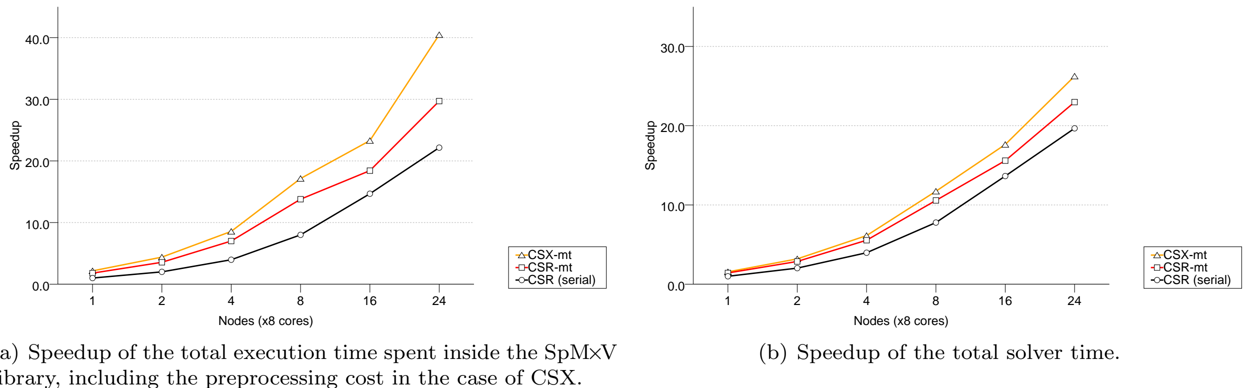
Figure 5 shows the average speedups achieved by simply the SpM×V code (Fig. 5(a)) and the total solver time (Fig. 5(b)) using the original Elmer CSR, our multithreaded CSR version and the CSX (including the preprocessing cost), respectively. In the course of 1000 linear system iterations, CSX was able to achieve a



(a) vortex3d

(b) shell

Fig. 4. Intra-node speedup (1000 linear system iterations).



(a) Speedup of the total execution time spent inside the SpM×V library, including the preprocessing cost in the case of CSX.

(b) Speedup of the total solver time.

Fig. 5. Average speedup of the Elmer code up to 192 cores using the CSX library (1000 linear system iterations).

significant performance improvement of 37% over the multithreaded CSR implementation, which translates to a noticeable 14.8% average performance improvement of the total execution time of the solver. Nevertheless, we believe that this improvement could be even higher if other parts of the solver exploited parallelism within a single node as well, since the SpM×V component would become then even more prominent, allowing a higher performance benefit from the CSX optimization. Regarding the preprocessing cost of CSX, we used the typical case of exploring all the candidate substructures using matrix sampling. Yet the cost was fully amortized within 224–300 linear system iterations.

4. Conclusions & Future Work

Scaling applications for petascale or exascale systems is very challenging as large classes of applications do not scale either with the number of nodes, or the number of cores within a single node, or, even worse, with both. In this paper we focused on scalability issues arising within a single node, when trying to utilize all the available cores. More specifically, we presented and evaluated the integration of the recently proposed Compressed Sparse eXtended (CSX) sparse matrix storage format into the widely used Elmer multiphysics software package. This is, to our knowledge, one of the first approaches of evaluating the impact of an innovative sparse matrix storage format on a 'real-life' production multiphysics software. CSX was found able to improve the performance of the SpM×V component by nearly 35% compared to the multithreaded CSR when executed on a single node and offered a 15% overall performance improvement of the solver in a 24-node, 192-core SMP cluster.

In the near future, we plan to expand our evaluation to NUMA architectures and even larger systems. Additionally, we are investigating ways for minimizing the initial preprocessing cost of CSX and also extensions to the CSX's interface to efficiently support problem cases where the non-zero values of the sparse matrix change during the simulation. Finally, we plan to investigate sparse matrix reordering techniques and how these affect the overall execution time of the solver using the CSX format.

Acknowledgments

This work was financially supported by the PRACE project funded in part by the EU's 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-211528 and FP7-261557.

References

1. M. Lyly, J. Ruokolainen, and E. Järvinen, “ELMER – a finite element solver for multiphysics,” in *CSC Report on Scientific Computing*, 1999–2000.
2. G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, and N. Koziris, “Performance evaluation of the sparse matrix-vector multiplication on modern architectures,” *The Journal of Supercomputing*, vol. 50, no. 1, pp. 36–77, 2009.
3. K. Kourtis, V. Karakasis, G. Goumas, and N. Koziris, “CSX: An Extended Compression Format for SpMV on Shared Memory Systems,” in *Proceedings of the 16th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP'11)*, (San Antonio, Texas, USA), pp. 247–256, ACM, 2011.
4. R. Vuduc, J. W. Demmel, and K. A. Yelick, “OSKI: A library of automatically tuned sparse matrix kernels,” *Journal of Physics: Conference Series*, vol. 16, no. 521, 2005.
5. A. Pinar and M. T. Heath, “Improving performance of sparse matrix-vector multiplication,” in *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, (Portland, OR, USA), ACM, 1999.
6. C. Lattner and V. Adve, “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation,” in *International Symposium on Code Generation and Optimization (CGO'04)*, (San Jose, CA, USA), IEEE Computer Society, 2004.