D-SAB: A Sparse Matrix Benchmark Suite

Pyrrhos Stathis, Stamatis Vassiliadis, and Sorin Cotofana

Computer Engineering Laboratory, Electrical Engineering Department,
Delft University of Technology,
2600 GA Delft, The Netherlands
{pyrrhos,stamatis,sorin}@dutepp0.et.tudelft.nl

Abstract. In this paper we present the Delft Sparse Architecture Benchmark (D-SAB) Suite for evaluating sparse matrice architectures. The focus is on providing a benchmark suite which is flexible and easy to port on (novel) systems, yet complete enough to expose the main difficulties which are encountered when dealing with sparse matrices. The novelty compared to previous benchmarks is that it is not limited by the need for a compiler. The D-SAB comprises of two parts: (1) the benchmark algorithms and (2) the sparse matrix set. The benchmark algorithms (operations) are categorized in (a) value related operations and (b) position related operations.

1 Introduction

Dealing with sparse matrices has always been problematic in the scientific computing world. The reason for this, simply put, is that computers, and especially vector computers, are best in dealing with regularity. One of the problems associated with sparse matrices is the determination of a common way to evaluate new architectural features. In this paper we introduce D-SAB, a benchmark suite to be used in early architectural developments The contributions of this paper can be summarized as follows:

- We propose the Delft Sparse Architecture Benchmark (D-SAB), a benchmark suite comprising of a set of operations and a set of sparse matrices for the evaluation of novel architectures and techniques. By keeping the operations simple D-SAB is not dependent on the existence of a compiler on the benchmarked system.
- Although keeping the code simple D-SAB maintain coverage and exposes the main difficulties that arise during sparse matrix processing. Moreover, the pseudo-code definition of the operations allows for a higher flexibility of the implementation.
- Unlike most other sparse benchmarks, D-SAB makes use of matrices from actual applications rather than utilizing automatically generated matrices.

The remainder of the paper is organized as follows: In the next section, Section 2 we discuss previous work on the field, and give our motivation and goals for the development of D-SAB. Subsequently, in Sections 3 and 4 we describe the operations and the matrix collection that comprise the benchmark. Finally, in Section 5 we give some conclusions.

2 Previous Work, Motivation, and Goals

Up to now, several efforts have been made that address the problem of benchmarking the performance of various architectures on sparse matrix operations. Some of the most important are listed below:

- The **Perfect Club** [2] is a collection of 13 full applications from engineering and scientific computing written in Fortran. A number of these include code involving operations on sparse matrices, mainly linear iterative solvers.
- The **NAS Parallel Benchmarks** [1] are a set of 8 programs designed to help evaluate the performance of parallel supercomputers. The benchmarks, which are derived from computational fluid dynamics applications, consist of five kernels and three pseudo-applications and aim at providing a performance metric for both dense and sparse systems.
- **SparseBench: a Sparse Iterative Benchmark** This benchmark [6] uses common iterative methods, preconditioners, and storage schemes to evaluate machine performance on typical sparse operations. The benchmark components are: Conjugate Gradient and GMRES iterative methods, Jacobi and ILU preconditioners.
- **SPARK: A benchmark package for sparse computations** [9] (see also [10]) is a benchmark developed by Saad and Wijshoff to evaluate the behavior of various architectures on the field of sparse computations. The main rationale behind the SPARK approach for designing the benchmark is to try and capture the main kernels that expose the problems encountered in sparse computing.

All the above mentioned benchmarks (except the NAS benchmark) assume a fully functioning system including a compiler. Although this is a very useful approach that reflects real system settings and is easy to use, it cannot be used for architectures in an early stage of development that don't have a compiler or architectures that simply don't make use of a compiler. Furthermore, the benchmarks (except the NAS benchmark) define the storage method of the sparse matrices. Although the storage methods utilized are usually the most commonly utilized in the scientific world, this approach may fail to reveal the full potential of an architecture since the matrix format determines the way it will be accessed and processed. Therefore the flexibility of these benchmarks is limited and does not allow for a novel way of storing, accessing and operating on the sparse matrix. Additionally, the above benchmarks use matrices that are automatically generated. This is mainly done for reasons of memory efficiency. However, we believe that the parameters that are used to generate those matrices cannot capture the diversity of sparsity patterns that are observed in matrices obtained from actual applications.

Our proposed benchmark aims removing the above shortcomings of the currently existing sparse matrix benchmarks while keeping their benefits. Our benchmark is partly inspired by the NAS and SPARK benchmarks regarding their design philosophy.

3 The Benchmark Operations

To construct the benchmark we need to construct a set of algorithms that can cover the basic operations that make up most of the sparse matrix related applications. We have examined a number of toolkits and packages to extract the operations including the following: SPARSEKIT [8], The NIST Sparse BLAS [7, 4], LASPACK, and Sparselib++ [5].

We have observed that although most packages offer an extensive set of functions, there is a plurality of functions performing the same operation. After an initial analysis of the packages we have chosen to divide the basic operations in two parts:

1. *Value Related Operations* (VROs). These operations include arithmetic operations such as multiplication, addition, inner product, etc.
2. *Position Related Operations* (PROs). These include operations for which the actual values of the elements are not important for the outcome, such as element searching, element insertion, matrix transposition, etc.

We have chosen 5 operations from each of the VROs and PROs which we believe represent the most basic operations of sparse matrix applications and are listed in Tables 1 and 2 respectively.

Table 1. Value Related benchmark operations

Name	Operation	Description
1. Multiplication	$C = AB$	Multiplication of two sparse matrices. This operation has a high degree of value reuse and indicates how a method can deal with this fact.
2. Addition	$C = A + B$	Matrix addition exposes fill-in (i.e. the addition of extra nonzero elements in a sparse matrix)
3. SMVM	$y = Av$	Sparse Matrix - dense Vector Multiplication. This operation in one of the most important in sparse matrix computations in terms of execution time.
4. Gaussian Elimination, Pivoting	see text	Operations used in Direct Methods for linear system solving and the construction of preconditioners
5. (Bi)Conjugate Gradient	see Fig 1	(Bi)CG, 2 iterative solvers, typical sparse matrix applications. The main benchmark for most existing sparse matrix benchmarks

Operations 1 till 3 are self explanatory. Figure 1 depicts the code for the BiCG algorithm where x, p, z, q, $\tilde{p}$, $\tilde{z}$ and $\tilde{q}$ denote dense vectors and Greek letters denote scalars. The $\Rightarrow$ signs indicate the code lines of interest since they form the asymptotic execution of the code. Therefore only this code needs to be executed for benchmarking. We have included the BiCG code along the CG code because it includes SMVM with both the A and A^T. Performing both in the same code is considered troublesome and is avoided in practice in spite the fact that algorithmically it can offer advantages. However we believe that this is precisely the reason to include it in our benchmark.

```
Compute $r_0 = b - Ax_0$ using initial guess $x_0$
$\tilde{r}_0 = r_0$
for $i = 1, 2, 3, \ldots$
    if $i = 1$
        $p_i = z_{i-1}$
        $\tilde{p}_i = \tilde{z}_{i-1}$
    else
$\Rightarrow$       $p_i = z_{i-1} + \beta p_{i-1}$
$\Rightarrow$       $\tilde{p}_i = \tilde{z}_{i-1} + \beta_{i-1}\tilde{p}_{i-1}$
    endif
$\Rightarrow$   $q_i = Ap_i$
$\Rightarrow$   $\tilde{q}_i = A^T\tilde{p}_i$
$\Rightarrow$   $\alpha_i = \rho_{i-1}/\tilde{p}_i^T q_i$
$\Rightarrow$   $x_i = x_{i-1} + \alpha_i p_i$
$\Rightarrow$   $r_i = r_{i-1} + \alpha_i q_i$
$\Rightarrow$   $\tilde{r}_i = \tilde{r}_{i-1} + \alpha_i \tilde{q}_i$
    check convergence; continue if necessary
end for
```

Fig. 1. The Non-preconditioned Bi-Conjugate Gradient Iterative Algorithm.

Pivoting and Gaussian elimination are operations that relate to the direct methods for solving linear systems as well as for constructing preconditioners for iterative methods. Several methods exist to perform these operations. For D-SAB we have chosen to use the most straightforward version described below:

```
for $j = 0$ to $n - 2$ do
$\Rightarrow$ Find position $(q)$ of max abs in column $C_j$
$\Rightarrow$ if $q <> j$ then exchange rows$(j, q)$
    for $i = j$ to $n - 2$ do
        $R_{j+1} = R_{j+1} - a_{ij}a_{jj}R_j$
    end for
end for
```

where C_k denotes the k^th row, $R_k = (a_{1k}, a_{2k}, \ldots, a_{nk})$, R_k denotes the k^th row, $R_k = (a_{k1}, a_{k2}, \ldots, a_{kn})$ and a_{ij} the element of A at position (i, j). The part of the algorithm that is used for pivoting is denoted by the $\Rightarrow$ sign.

Position Related Operations:

All ten named benchmarks are to be executed using the benchmark matrices that are listed in the following section. Wherever a second matrix is needed (i.e.

Table 2. Position Related benchmark operations

Name	Description
6. Sub-matrix Extraction	Create a new matrix from by extracting a sub-matrix from matrix A. Start from position $(5, 10)$ Use sizes 10x10, 100x100, 1000x1000, 10000x10000 if the original matrix size permits to do so.
7. Transposition (A^T)	Create a new matrix that is the transpose of the original.
8. Get element from matrix	Return the time needed to access an element in the matrix averaged over 50 values randomly chosen over the whole matrix. At least 10 should return a non-zero value.
9. Extract Lower Triangular Part	Create a new matrix that comprises only of the elements a_{ij} of matrix A where $i \geq j$.
10. Insert or Modify Element	Modify a non-zero Element in the matrix or Insert an element in the matrix (modify a zero entry) .

the B Matrix in Addition and Multiplication) we construct it as follows: The B matrix is the A matrix mirrored around the second diagonal, that is, the top right to bottom left diagonal. We have chosen to do so because the sparse matrix suits from which we have chosen our benchmark matrices do not offer pairs of matrices of the same dimensions. Therefore, to avoid the fill-in free addition of the matrix with itself we mirror the matrix at the diagonal where most of the matrices are not symmetric.

4 The Sparse Matrix Suit

The benchmark matrices for the D-SAB suite were chosen from a wide variety of matrices that are available from the Matrix Market Collection [3]. The collection offers 551 matrices collected from various applications and includes several other collections of sparse matrices and is therefore the most complete we could get access to. Of these matrices we have selected 132 matrices taking care not to select similar matrices in terms of application, size and sparsity patterns in order to reduce the number of matrices while keeping the variety intact. The 132 matrices matrices have been sorted using 3 different criteria that relate to various matrix properties. For an extensive discussion of the criteria refer to [11].

Sorting the matrices by the three named criteria resulted in three sets. From each of these sets ten matrices have been chosen to represent the set due to space limitation. The steps are constant in a logarithmic scale since we observed from the data that the distributions after sorting for each criterion was logarithmic rather than linear. Therefore for instance for the matrix size criterion each matrix is approximately 3.5 times larger than the previous one. See [11] for the precise list of the used matrices.

5 Conclusions

In this paper we introduced the Delft Sparse Architecture Benchmark (D-SAB) suite, a benchmark suite comprising of a set of operations and a set of sparse matrices for the evaluation of novel architectures and techniques. By keeping the operations simple D-SAB does not depend on the existence of a compiler meant to map the benchmark code on the benchmarked system. Although keeping the code simple D-SAB maintain coverage and exposes the main difficulties that arise during sparse matrix processing. Moreover, the pseudo-code definition of the operations allows for a higher flexibility for the way the operation is implemented. Unlike most other sparse benchmarks, D-SAB makes use of matrices from actual applications rather than utilizing synthetic matrices.

References

1. D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.
2. M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, L. Pointer, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Scheider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, R. Goodrum, and J. Martin. The PERFECT club benchmarks: Effective performance evaluation of supercomputers. *The International Journal of Supercomputer Applications*, 3(3):5–40, 1989.
3. R. F. Boisvert, R. Pozo, K. Remington, R. Barrett, and J. J. Dongarra. The Matrix Market: A web resource for test matrix collections. In R. F. Boisvert, editor, *Quality of Numerical Software, Assessment and Enhancement*, pages 125–137, London, 1997. Chapman & Hall.
4. S. Carney. A revised proposal for a sparse blas toolkit, 1994.
5. J. Dongarra, A. Lumsdaine, R. Pozo, and K. Remington. A sparse matrix library in c++ for high performance architectures, 1994.
6. J. J. Dongarra and H. A. Van der Vorst. Performance of various computers using standard linear equations software in a Fortran environment. *Supercomputer*, 9(5):17–30, Sept. 1992.
7. K. Remington and R. Pozo. Nist sparse blas: user's guide, 1996.
8. Y. Saad. SPARSKIT: A basic tool kit for sparse matrix computations. Technical Report 90-20, NASA Ames Research Center, Moffett Field, CA, 1990.
9. Y. Saad. Wijshoff: Spark: A benchmark package for sparse computations, 1990.
10. Y. Saad and H. Wijshoff. A benchmark package for sparse matrix computations. In *Proceedings of the 1988 International Conference on Supercomputing*, pages 500–509, St. Malo, France, 1988.
11. P. Stathis, S. Vassiliadis, and S. Cotofana. D-sab: Delft sparse architecture benchmark, http://ce.et.tudelft.nl/iliad/d-sab/, 2003.