Utilizing Recursive Storage in Sparse Matrix-Vector Multiplication—Preliminary Considerations

Michele Martone* Salvatore Filippone* Salvatore Tucci*
michele.martone@uniroma2.it salvatore.filippone@uniroma2.it tucci@uniroma2.it
Marcin Paprzycki†⋆ Maria Ganzha†‡
marcin.paprzycki@ibspan.waw.pl maria.ganzha@ibspan.waw.pl
*University of Rome "Tor Vergata", Italy †Polish Academy of Sciences, Poland
⋆Warsaw Management Academy, Poland ‡University of Gdańsk, Poland

Abstract

Computations with sparse matrices on *"multicore cache-based"* computers are affected by the irregularity of the problem at hand, and performance degrades easily. In this note we propose a recursive storage format for sparse matrices, and evaluate its usage for the Sparse Matrix-Vector (SpMV) operation on two multicore and one multiprocessor machines. We report benchmark results showing high performance and scalability comparable to current state of the art implementations.

1 Introduction

In this note we consider *sparse matrix-vector multiplication (SpMV)*. This operation is at the core of most iterative algorithms for sparse linear algebra problems [9]. While we focus on the SpMV, considerations presented here are applicable also to other operations involving sparse matrices. Note that we are concerned with *"cache-based computing architectures"*, therefore omitting hybrid architectures like the Sony/Toshiba *CBE (Cell Broadband Engine)* or *GPU (Graphical Processing Unit)*-based.To make use of the available floating point performance, sparse matrix codes have to confront a number of problems, like: 1) high ratio of integer to floating point operations (caused by the utilization of indirect addressing), leading to underutilization of floating point unit(s) within the processor; 2) low reutilization of data (*low temporal locality* of memory references), resulting in saturation of the memory bandwidth and causing processor stalls; 3) ever-growing depth of memory hierarchy and complication of processor architecture, particularly in the case of multicore processors. A fundamental technique to address these problems is called *register blocking*, which represents sparse matrices by storing adjacent nonzero elements in small rectangular blocks. Two prominent register blocked matrix formats are the blocked variants of the *Compressed Sparse Rows (CSR)* and the *Compressed Sparse Columns (CSC)* formats, called respectively *Block Compressed Sparse Rows (BCSR)* and *Block Compressed Sparse Columns (BCSC)* (see [5] for a discussion of register blocking). When utilizing register blocking, it is possible to achieve a considerable speedup over non blocked formats, although it is required for the matrices to already have a blocked structure. Matrices with such structure typically originate from partial differential equations discretized over physical domains (using techniques such as *finite differences*, the *Finite Element Method (FEM)*, or the *Finite Volume Method*) [9, ch.2]. However, there exists a large number of sparse matrices without any block structure, whose representation in a blocked format would incur performance loss. They can originate from: linear programming, least squares problems, information retrieval, optimization, or combinatorial problems. To deal with such matrices efficiently and achieve multicore satisfactory performance, we propose a variation of the traditional CSR/CSC data structures, which we name *Recursive CSR/CSC (RCSR/RCSC)*. This format, and its utilization in the implementation of the SpMV operation, is described in detail in the next section. We follow with the experimental setup used in benchmarking. In section 4 we discuss the performance data collected when running our code, and compare its behaviour to that of a state of the art code by Buluc et.al [1]. Finally, in section 5, we outline future research directions related to our matrix data structures.

2 A Recursive Storage

2.1 Matrix Partitioning

It is known that implementation of linear algebra algorithms based on recursively stored dense matri-

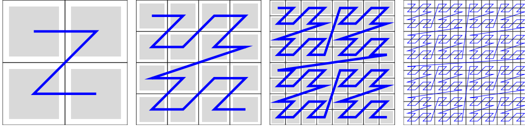

Figure 1: Z/Z^b sorted coordinates for 2x2, 4x4, 8x8,16x16 sized dense matrices.

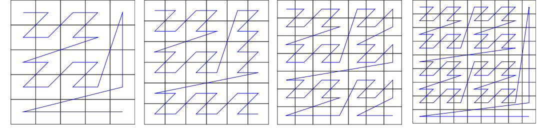

Figure 2: Z sorted coordinate for 5x5,6x6,7x7,9x9 sized dense matrices (imbalanced quad-partitions).

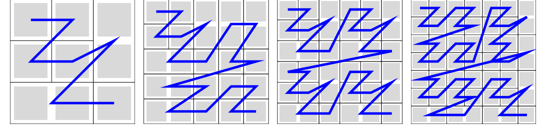

Figure 3: Z^b sorted coordinates for 3x3,5x5,6x6,7x7 dense matrices, sized as non power of 2 (balanced quad-partitions).

ces can be advantageous ([3]). However, little work has been done in trying to apply similar techniques to sparse matrices. Therefore, we have developed recursive variants of both CSR and CSC sparse matrix storage formats. Here, we present the main modifications necessary to the matrix assembly procedure (which became recursive):

- at the *root level*, the nonzero input elements are sorted in such a way to favour temporal locality of computations during the SpMV
- at the *intermediate levels* submatrices are partitioned recursively if specific criteria concerning number of nonzero elements and the submatrix size are met
- at the *leaf level*, submatrices end up with standard CSR/CSC structures

There exist multiple ways to assemble sparse matrices according to the above "schema." In this work we seek specifically to 1) allow the *recursive subdivision* into submatrices; 2) this subdivision should proceed in a quad-tree (i.e.: in *quadrants*) fashion. Moreover, we want the partitioning method to be adaptive to the machine cache size and (possibly) to other details of the processor architecture. Therefore, for computational and implementation ease, we have chosen a variant of Z-ordering (or Z-Morton, after G. Morton [4]), for the nonzero elements sorting at the root level of recursive matrix assembly. Let $x, y \in \mathbb{N}$, be the Cartesian coordinates of a point in $\mathbb{N}^2$, and T be a function $T : (x, y) \in \mathbb{N}^2 \rightarrow z \in \mathbb{N}$. Then, we can define the T-permutation Π_T of a vector $V = <(i_0, j_0), (i_1, j_1), ..., (i_{nz}, j_{nz})>$ as the vector (assuming no duplicates in V) $\Pi_T(V) = <\pi_0, \pi_1, ..., \pi_{nz}>$ such that $\pi_0<\pi_1<...<\pi_{nz}$ and $\pi_l<\pi_k$ hold whenever $T(i_{\pi_l}, j_{\pi_l})<T(i_{\pi_k}, j_{\pi_k})$. Given $i \in \mathbb{N}$, we define (adopting the notation of [8, section 2]) the 2-dilation of i, $\overrightarrow{i}$ (*"i dilated"*) as the result of interleaving a 0 bit between each meaningful bit in the binary representation of i. So, if $i = 2^8-1 = 11111111_2 = \mathtt{FF}_{16}$, its 2-dilation is $\overrightarrow{i} = 0101010101010101 = 5555_{16}$. Alike, we define $\overleftarrow{i} \stackrel{def}{=} 2\ \overrightarrow{i}$, which the left-shifted 2-dilation of i. Let us now define the mapping Z as: $Z(i, j) \stackrel{def}{=} \overrightarrow{i} + \overleftarrow{j}$.

Above, if we take T to be Z and apply to the coordinate vector V, then we induce a Z-order on V. In Figure 1 we depict the resulting ordering of elements for some small dense matrices. Experiments reported in [6] show that performing linear algebra on Z(Morton) sorted elements can reduce page faults for large dense matrices. We conjecture this to be true also for sparse matrices, as the sparseness of elements leads to non-linear (thus, not easily detectable by the prefetch engines) access patterns. By forcibly limiting the *leaf matrix* dimensions, while storing and tiling them in a recursive Z fashion, we increase the locality of memory accesses, regardless the matrix sparsity pattern. However, Z-ordering matrices, which are not square or not sized as powers of 2 leads to imbalanced partitionings (see Figure 2, where we depict small dense matrices with "singleton" leaves). To address this issue, we have modified the Z-ordering algorithm to handle non square matrices and non-power-of-two sized matrices. We call our modification *balanced Z ordering*, or Z^b. Let the matrix size be $m \times k$ and i, j a nonzero coordinate. Let $lbits(i)$ be the index of the highest bit in the binary representation of i: $lbits(i) \stackrel{def}{=} \lfloor log_2(i) \rfloor$, and let β_{mk} be $lbits(min(m, k))$. Then define: $\mu : i, m, \beta_{mk} \in \mathbb{N} \rightarrow i^* \in \mathbb{N}$ as $\mu(i, m, \beta_{mk}) \stackrel{def}{=} \gamma(i, \lfloor m/2 \rfloor) \cdot (2^{\beta_{mk}} + \mu(i - \lfloor m/2 \rfloor, m - \lfloor m/2 \rfloor, \beta_{mk}-1)) + (1-\gamma(i, \lfloor m/2 \rfloor))\mu(i, \lfloor m/2 \rfloor, \beta_{mk}-1)$.

With $\gamma(x, y) = 1$ when $x>y$ and 0 otherwise. The Z^b order function of interest is then defined as: $Z^b(i, j, m, k) \stackrel{def}{=} Z(\mu(i, m, \beta_{mk}), \mu(j, k, \beta_{mk}))$.

Figure 3 shows the Z^b-ordered elements of some small dense matrices. Note that using Z^b instead Z has a downside: Z^b is not bijective; but this is not a problem as long as we do not rely on this and use Z^b ordering for sorting purposes only (which is our case).

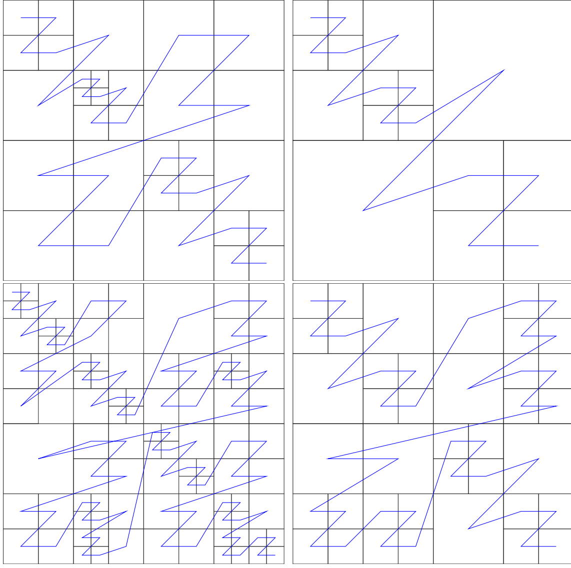

Figure 4: Matrices `ASIC_320k`(upper two) and `torso1` (lower two) δ-partitioned on a 1MB-sized outermost cache machine (**M1**) (left), and on a 2MB-sized outermost cache machine (**M3**) (right).

Knowing values of m, k, in a Z^b-ordered coordinates array, we can use binary search to easily locate *split points* delimiting the four submatrices. Then we partition submatrices after recursively locating split points. To prevent indefinite recursive splitting, we adopted a recursion decision function. Currently, this function (δ) is a heuristic working with matrix dimensions m, k, number of nonzeros nnz, outermost machine cache size cs, numerical and pointer element size es and ws.

$$eab(m, k, nnz, es, ws) \stackrel{def}{=} es * (nnz + nnz + m) + ws(m + nnz)$$

$$\delta(m, k, nnz, cs, es, ws) \stackrel{def}{=} \begin{cases} true, if\ eab(m, k, nnz, es, ws) > \alpha\ cs,\ or \\ true, if\ \ nnz * es > \beta\ cs \\ false, otherwise \end{cases}$$

Here, eab is an estimate of the accessed bytes during a CSR SpMV on a (sub) matrix with the given parameters. Term $es * (nnz + nnz + m)$ takes into account the nnz accessed multiplicand vector elements, the nnz matrix elements, and m written output vector elements. The $ws * (m + nnz)$ term takes into account the m row pointer indices and the nnz column index elements.

Figure 4 depicts two matrices partitioned with this heuristic, on machines with differing outermost cache size. Since our heuristic relies on the count of contiguous nonzeros, we are indeed applying a variant of *cache blocking* ([7]). This heuristic does not take into account many other possible factors, such as: the cache line size, the matrix pattern, or whether the submatrix is full rank or not.We will investigate this issue in the future.

2.2 Sparse Matrix-Vector Multiplication

We define the SpMV of a matrix A and a vector x, updating vector y, as $y \leftarrow y + Ax$. If A is stored recursively in four submatrices $A_{11}, A_{12}, A_{21}, A_{22}$, it is natural to implement the SpMV operation as follows:

$$\begin{vmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{vmatrix} \begin{vmatrix} x_1 \\ x_2 \end{vmatrix} = \begin{vmatrix} A_{11} & A_{12} \end{vmatrix} \begin{vmatrix} x_1 \\ x_2 \end{vmatrix} + \begin{vmatrix} A_{21} & A_{22} \end{vmatrix} \begin{vmatrix} x_1 \\ x_2 \end{vmatrix} \quad (1)$$

$$= \begin{vmatrix} A_{11}x_1 \\ A_{12}x_2 \end{vmatrix} + \begin{vmatrix} A_{21}x_1 \\ A_{22}x_2 \end{vmatrix} \quad (2)$$

Therefore, our SpMV algorithm for the RCSR/RCSC proceeds by descending a *quadtree* of submatrices, and performing computation at the leaf level only (where the conventional CSR/CSC SpMV algorithms are applied). As stated above, in section 2.1, we have developed criteria to control size of leaf matrices, thus limiting the recursion overhead (actually, the cost of descending a tree of pointers).Figure 4 represent leaf matrices after the recursive partitioning and a line indicating the order in which they are visited during the SpMV operation. As noted above, the main goal of our work is to deal with multicore processors (and, in general, introduce parallelism into the operations like the SpMV). In this note we evaluate a very simplistic approach to parallelization, which is limited to two processors/cores. Specifically, we implement it by overlapping the computation of two terms in (2), using the OpenMP `#pragma omp parallel for` directive (in a fixed 2-fold loop); applied to the upper and lower pairs of matrix quadrants. Thus, the two-core execution of the SpMV will spawn two execution threads, of which the first will visit submatrices in the upper two quadrants of the matrix, and the second one will visit the lower two. In the near future we plan to investigate ways of obtaining robust scalable recursive SpMV.

3 Experimental Setup

For space reasons we report results obtained on a limited experimental setup. In our experiments, we run SpMV (defined as $y \leftarrow y + Ax$) on the (nonsymmetric) matrices reported in Table 4. They originate from the *University of Florida Sparse Matrix*

machine	model	cpus/ cores	data caches
M1	AMD Opteron 246 1.0GHz	2/1	2xL1,2xL2: L2:1M/16-w/64B L1:64KB/2-w/64B
M2	AMD Athlon 64 X2 Processor 6000 3.0GHz	1/2	2xL2: L2:1MB/16-w/64B L1:64KB/2-w/64B
M3	AMD Opteron 2354 Quad-Core 2.2GHz	2/4	2xL3,2x4xL2,2x4xL1: L3:2MB/32-w/64B L2:512KB/16-w/64B L1:64KB/2-w/64B

Table 1: Test machines.

machine name	compiler
M2	`gcc version 4.1.2`
M1,**M3**	`gcc version 4.3.2`
all	`gcc version 4.2.4 (Cilk Arts build 8503)`

Table 2: Compilers on test machines.

Collection [2]. Our (RCSR/RCSC) SpMV kernel implementations have been ran with and without multicore parallelism, and are compared against the CSB prototype code released by authors of [1]. We have chosen to benchmark against CSB because, just like RCSR/RCSC, it was conceived to be used in a multicore context. The CSB stores Z-sorted elements in *sparse blocks* of 2^k size, whereas the RCSR/RCSC stores Z^b-sorted *submatrices* of arbitrary size; we find this duality interesting for comparison purposes. The CSB code is parallelized with the CILK++ system, which extends the C++ language and requires applications to be compiled by its special compiler. Then, the executable program file is linked to the CILK++ *runtime load balancer*. The codes were run on the 64 bit machines shown in Table 1; the used compiler versions are in Table 2; compilation flags in Table 3. We chose not to use machine specific optimization flags because of slight incompatibilities between the CILK++ compiler and compilers available in the Fedora Linux distributions installed on our machines. Both codes use `double` as the numerical type, 32 bit integer indices, and 64 bit pointers. With each experiment, we also report the measured performance of CSR/CSC (our implementation) and the CSC implementation of Buluc et al. [1] (in the plots we mark the measurements of

implementation	compilation flags
CSR/CSC/RCSR/RCSC (C99)	`-O3 -fopenmp -std=c99`
CSB*/CSC* (CILK++)	`-O3 -fno-rtti -fno-exceptions`

Table 3: Relevant (non-warnings) compiler flags used.

matrix	rows	columns	non zeros	n.z./r.	n.z./c.
ASIC_320k	321821	321821	2635364	8.19	8.19
Rucci1	1977885	109900	7791168	3.94	70.893
cont11_l	1468599	1961394	5382999	3.67	2.74
neos	479119	515905	1526794	3.19	2.96
rail4284	4284	1096894	11284032	2633.99	10.287
rajat31	4690002	4690002	20316253	4.33	4.33
sls	1748122	62729	6804304	3.89	108.47
sme3Dc	42930	42930	3148656	73.34	73.34
spal_004	10203	321696	46168124	4524.96	143.51
stomach	213360	213360	3021648	14.16	14.16
torso1	116158	116158	8516500	73.32	73.32

Table 4: Test matrices.

their code as CSC* and CSB*). We have modified the timing function of the CSB code to use a `double` (instead of `int`) variable, to limit precision loss (milliseconds are measured). Both codes use the `gettimeofday` POSIX function for timing. Performance is expressed in *Millions of FLoating Point Operations per Second (MFLOPS)* As conventional for the SpMV, we count two floating point operations for each matrix nonzero. We perform 100 SpMV kernel runs for each sample and report the best value (we have observed that in all cases best value differs from the average no more than 2%). Note that the actual CSB prototype code leaves apart portions of matrices, and thus taking into account this *leftover* in the computation would likely lead to somewhat differing results. We should also note that during benchmarks, the **M3** machine was also (lightly) loaded as a web server, and this could have affected adversely our measurements.

4 Experimental Results

Figures 5,6,7, summarize performance data collected running experiments with matrices found in Table 4 on machines reported in Table 1. Looking at them, we are interested primarily in: 1) scalability of RCSR/RCSC against that of CSB (from one to two cores), 2) performance of single core RCSR/RCSC against non recursive versions CSR/CSC, 3) performance of single core RCSR/RCSC against single core CSB, 4) which matrices perform better for which storage formats, 5) whether RCSR/RCSC is better than CSB on a particular machine. We observe that: 1) Generally, we find the scalability of our recursive partitioning comparable to that of CSB. RCSR/RCSC speedup ranges from 1.29 (**M1**, `neos`, RCSR) to 1.97 (**M3**, `spal_004`, RCSR), while CSB* both worst (0.91, `torso1`) and best (1.98, `cont11_l`) speedups occur on **M1**. We observe that **M1** favours the 2-core CSB* code over the RCSR/RCSC; both in terms of mean speedup (1.68 vs. 1.45) and mean performance (308.9

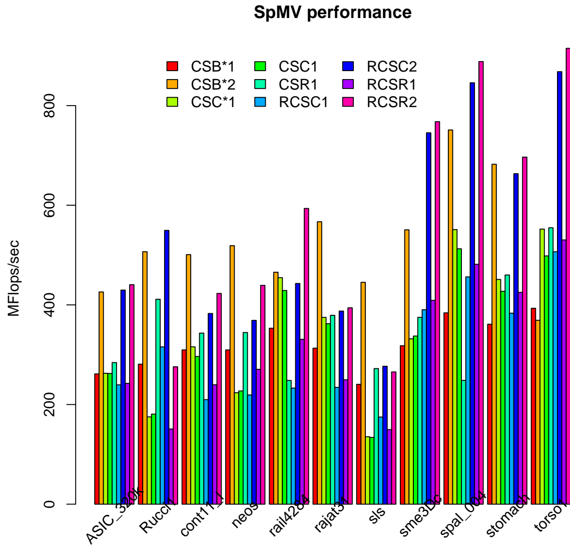

Figure 5: Results on **M2**.

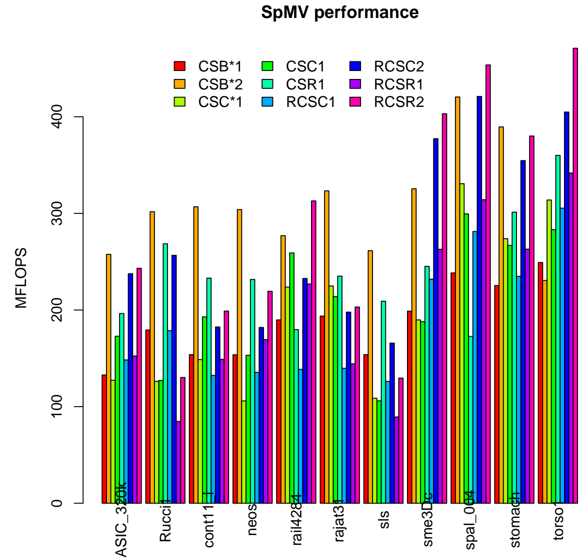

Figure 6: Results on **M1**.

MFLOPS; +9% more than the RCSR/RCSC). We conjecture this to be an advantage of CILK++ over plain OpenMP on **M1**'s multiprocessor architecture. On the newer machines (**M3**,**M2**) we observe the two-core RCSR/RCSC to perform ($\tilde{4}$50 and $\tilde{5}$47 MFLOPS vs 407.9 and 525.7) and scale (1.75 vs 1.65) (although very slightly) better than CSB. Please note that (see section 2.2) our current parallelization strategy does not assure load balance among the two processors: the first level recursive partitioning is influenced by the matrix dimensions only, thus introducing load imbalance for matrices with disparity of nonzero element count between the upper and lower quadrants. However, most of testbed matrices are quite balanced (51% on nonzeros in the upper quadrant, 49% in the lower one), except for `ASIC_320k`: (57%/43%), and `torso1`: (48%/52%). We observe that notwithstanding this imbalance, matrix `ASIC_320k` scales up well, even better than other matrices. 2) We observe that the utilization of recursive partitioning usually impairs the performance on single core, when compared to the non recursive counterpart. Consider matrix `Rucci1`. When using RCSR, it reaches only about half of the (quite good, on all three machines) performance of CSR. The same holds for RCSC. This performance drop is justified by the average nonzero per row count; for `Rucci1`, less than 4 elements. Indeed, with the current partitioning policy based on the δ decision function (which does not take in consideration the number nonzeros per row), a matrix like this, which is quite big (as it exceeds several times the outermost cache size of our machines) becomes partitioned into a significant number of smaller matrices (341 on **M1**, 85 on **M3**), thus increasing both tree traversal overhead, and possibly introducing very scarcely populated matrices, with a consequent high index overhead. Similar arguments hold also for `cont11_l, sls, rajat, neos`. 3) Similar observations concerns CSB too, which outperforms RCSR/RCSC on a single core. The CSB format performs better, because while it is based on submatrices partitioning, it does not incur in any recursion overhead.

A possible way to improve performance of the RCSR/RCSC would be taking in consideration a nonzero per row or per column count based threshold to prevent unnecessary subdivisions (unless the number of partitions is less than the number of computing cores). 4) The best performing matrix on all machines was `torso1`, stored in our recursive CSR format, in both single and two cores cases (see, Table 5). Indeed, matrices gaining the most from (single or multicore) RCSR/RCSC are `rail4208, sme3Dc, spal_004, stomach, torso1`. These are also the matrices with highest nonzeros per row count (as high as 4524.96 for the `spal_004`). On the other hand, we observe that CSR/RCSR beats CSC/RCSC in almost all cases (except `Rucci1` and `sls`). The reason is the differing read/write pattern of column and row based SpMV kernels. For algorithmic reasons, CSC/RCSC perform one write per matrix nonzero el-

machine	best	(1 core)		best	(2 cores)	
	MFLOPS	format	matrix	MFLOPS	format	matrix
M1	359.9	CSR	torso1	470.8	RCSR	torso1
M2	554.7	CSR	torso1	914.9	RCSR	torso1
M3	385.8	RCSR	torso1	714.5	RCSR	torso1

Table 5: Matrices/codes best performing, for each machine in our test set.

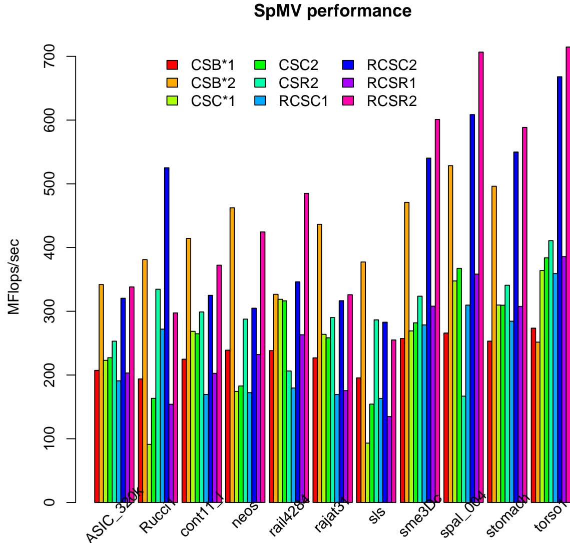

Figure 7: Results on **M3**.

ement, while CSR/RCSR perform one per matrix row. Because both (`Rucci1` and `sls`) matrices are *tall* (rows $\gg$ columns), the higher write rate of CSC/RCSC is not a problem, as compressing columns rather than rows decreases greatly memory traffic of row indices. This performance behaviour suggests us that comparing the nonzeros per column to the nonzeros per row count could give us hints on the memory traffic to be expected from a partitioning. Unlike row and column-based representations, CSB is not impacted by these parameters, as at the lower (*cache block*) level, it does not bias toward either rows or columns. 5) As we have observed earlier, measurements collected on **M1** favour the CSB format, while machines **M2**, **M3** favour RCSR/RCSC (see, Table 5). This may be a consequence of both good load balancing capabilities and low parallelization overhead of CILK++, as the overhead during the task recreation on the second processor on **M1** should be higher than the one incurred on the two cores involved on **M2** and **M3**.

5 Conclusions and Future Research

In this note we propose a recursive sparse matrix storage format favouring performance on multicore cache-based computers, achieved (also) by adapting to some relevant machine parameters. We have experimentally evaluated it on two multicore machines and a two-processor one, comparing scalability and performance to another state of the art storage format(CSB), with satisfactory results. Other strengths of our approach are: i)a natural way to organize a sparse matrix — it could pay off during the implementation of other (non-SpMV) operations; ii)the simplicity of its current parallelization schema; and the potential for high peak performance (especially with column or row biased matrices); iii)the use of standard OpenMP rather than specialized compilers, for parallelization; However, the current state of the partitioning policy leaves room for improvement: i) could be modified to choose on CSR or CSC at leaf level, if estimated as profitable ii)could take into account the number of computing cores and make sure there are enough partitions; In the near future, we plan to improve the matrix partitioning and SpMV parallelization strategy to overcome these limitations, and evaluate other sparse matrix algorithms in a recursive fashion.

References

[1] A. Buluc, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. 2009.

[2] T. Davis. University of florida sparse matrix collection. (92), 2009. submitted to ACM TOMS.

[3] E. Elmroth, F. Gustavson, I. Jonsson, and B. Kågström. Recursive blocked algorithms and hybrid data structures for dense matrix library software. *SIAM Review*, 1:3–45, 2004.

[4] G.M.Morton. A computer oriented geodetic data base and a new technique in file sequencing. *Tech. Rep.*, Mar. 1966.

[5] E. J. Im, K. Yelick, and R. Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *International Journal of High Performance Computing Applications*, 18(1):135, 2004.

[6] K. P. Lorton and D. S. Wise. Analyzing block locality in morton-order and morton-hybrid matrices. *SIGARCH Computer Architecture News*, (35), 2007.

[7] R. Nishtala, R. Vuduc, J. W. Demmel, and K. A. Yelick. When cache blocking sparse matrix vector multiply works and why. *Applicable Algebra in Engineering, Communication and Computing.*

[8] R. Ramani and D. S. Wise. Converting to and from dilated integers. *IEEE Trans. on Computers*, pages 567–573, 2008.

[9] Y. Saad. *Iterative Methods for Sparse Linear Systems, 2nd edition.* SIAM, Philadelpha, PA, 2003.