Efficient Multithreaded Untransposed, Transposed or Symmetric Sparse Matrix-Vector Multiplication with the Recursive Sparse Blocks Format

Michele Martone[a,1,*]

[a]*Max Planck Society Institute for Plasma Physics, Boltzmannstrasse 2, D-85748 Garching bei München, Germany*

Abstract

In earlier work we have introduced the *"Recursive Sparse Blocks"* (RSB) sparse matrix storage scheme oriented towards cache efficient matrix-vector multiplication (*SpMV*) and triangular solution (*SpSV*) on cache based shared memory parallel computers. Both the transposed (*SpMV_T*) and symmetric (*SymSpMV*) matrix-vector multiply variants are supported. RSB stands for a meta-format: it recursively partitions a rectangular sparse matrix in quadrants; leaf submatrices are stored in an appropriate traditional format — either *Compressed Sparse Rows* (CSR) or *Coordinate* (COO). In this work, we compare the performance of our RSB implementation of *SpMV*, *SpMV_T*, *SymSpMV* to that of the state-of-the-art Intel Math Kernel Library (MKL) CSR implementation on the recent Intel's Sandy Bridge processor. Our results with a few dozens of real world large matrices suggest the efficiency of the approach: in all of the cases, RSB's *SymSpMV* (and in most cases, *SpMV_T* as well) took less than half of MKL CSR's time; *SpMV*'s advantage was smaller. Furthermore, RSB's *SpMV_T* is more scalable than MKL's CSR, in that it performs almost as well as *SpMV*. Additionally, we include comparisons to the state-of-the art format Compressed Sparse Blocks (CSB) implementation. We observed RSB to be slightly superior to CSB in *SpMV_T*, slightly inferior in *SpMV*, and better (in most cases by a factor of two or more) in *SymSpMV*. Although RSB is a non-traditional storage format and thus needs a special constructor, it can be assembled from CSR or any other similar row-ordered representation arrays in the time of a few dozens of matrix-vector multiply executions. Thanks to its significant advantage over MKL's CSR routines for symmetric or transposed matrix-vector multiplication, in most of the observed cases the assembly cost has been observed to amortize with fewer than fifty iterations.

Keywords: sparse matrix-vector multiply, symmetric matrix-vector multiply, transpose matrix-vector multiply, shared memory parallel, cache blocking, sparse matrix assembly

1. Introduction and Related Literature

Many scientific and engineering problems require the solution of large *sparse* linear systems of equations; that is, systems where the number of equations largely outnumbers the average number of unknowns per equation. Since most of the entries in the (floating point number) coefficient matrices associated to such systems are zeroes, it is advantageous to represent them in a computer by their non-zero coefficients only. Such matrices (and their data structures in a computer) are therefore called *sparse*. A class of techniques for solving such systems is that of *iterative methods* [1]. Their computational core is largely based on repeated Sparse Matrix-Vector Multiplication (*SpMV*, defined as "$y \leftarrow\ y\ +\ A\ x$"; with A being the matrix, and x, y vectors) executions. Methods as BiCG or QMR (see Barrett et al. [2, Ch. 2]) or *Krylov balancing* algorithms (see Bai et al. [3, Alg. 7.1]) require computation of the *transpose product* as well (*SpMV_T*, defined as "$y \leftarrow\ y\ +\ A^T\ x$"). Availability of an efficient algorithm for *SpMV_T* eliminates the need for an explicit transposed matrix representation. Many applications give rise to symmetric matrices (that

*Corresponding Author
Email address: michele.martone@ipp.mpg.de (Michele Martone)

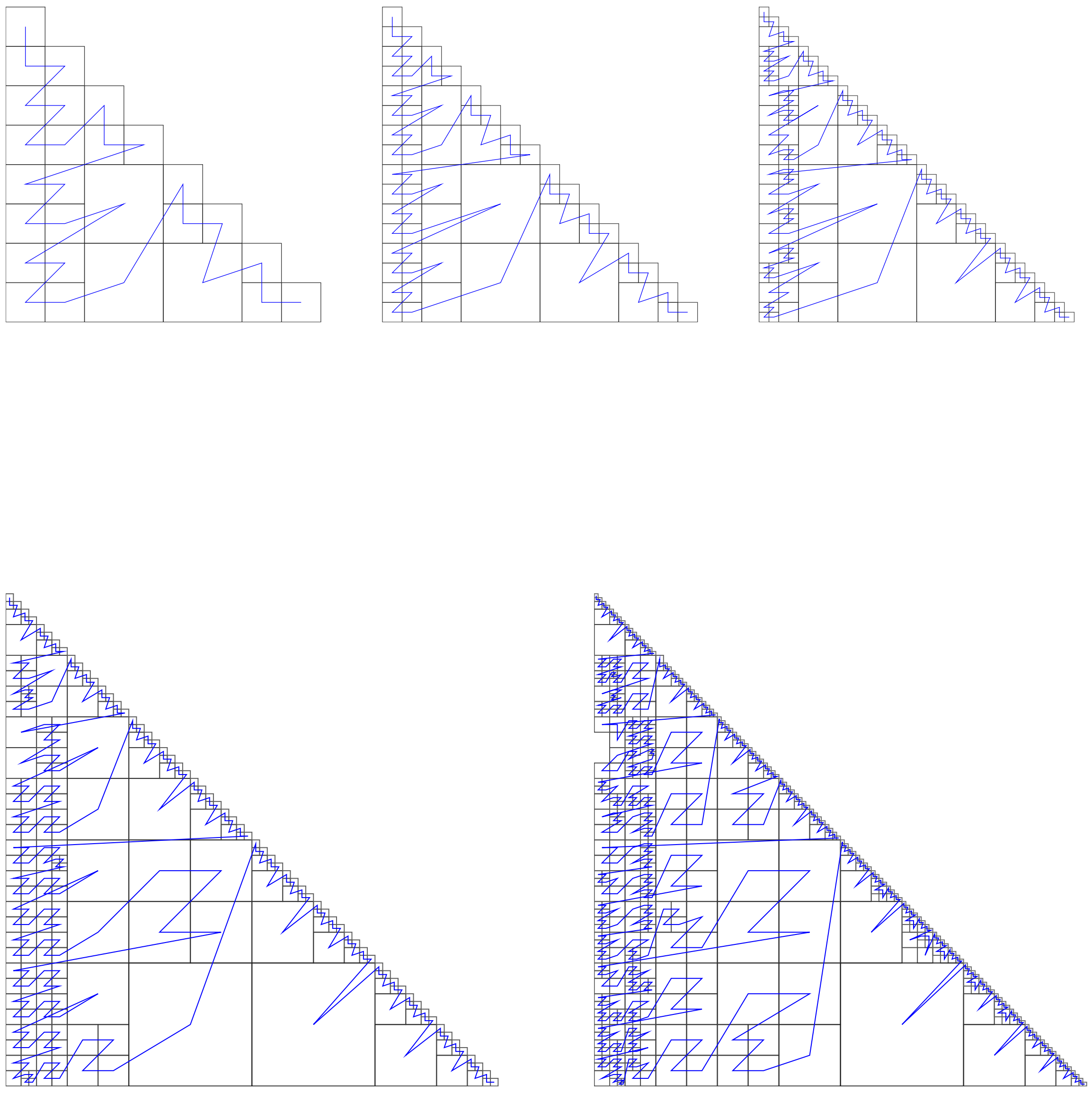

Figure 1: Matrix *audikw_1* (symmetric, 943695 rows, $3.9 \cdot 10^7$ nonzeroes excluding the upper triangle) partitioned differently for $1, 2, 4, 8$ and 16 threads RSB, on the same machine. The number of leaf submatrices is respectively $27, 64, 126, 247$ and 577; they are laid in memory in the same succession they are traversed by the broken line in the figure. Please note how most successive submatrices are adjacent, either vertically or horizontally; this has the chance of increasing reuse of cached operand vectors locations.

is, $A = A^T$). It is possible to take advantage of symmetry by omitting the explicit representation of the (strictly) upper triangle and use an appropriate *SpMV* algorithm exploiting the (non strictly) lower triangle. We denote such a variant by *SymSpMV*. In the application of iterative methods, the aforementioned *multiply* *kernel* variants often consume most of the computing time.

Contemporary multi-core architectures are increasingly inefficient in performing computations on sparse matrices, and research in algorithms capable of overcoming these inefficiencies (see the wider study of Asanovic et al. [4, Sec. 3.2]) is considered of strategic importance for forthcoming architectures. The main technological problem (see the microbenchmark-backed study by Schubert et al. [5] and the considerations by Buluç et al. [6, Fig.1]) is that for each new architecture the memory bandwidth to (floating point) compute rate is decreasing. This, while sparse operations/algorithms are already bandwidth (and latency) bound. In this context, designers of sparse matrix software should seek to optimize memory accesses, in both quantity (to cope with the limited bandwidth) and type (*local* accesses are preferred in order to avoid the high latency of cache misses).

Recently ([7]) we proposed the "RSB" (Recursive Sparse Blocks) hybrid data structure, aiming at improving (w.r.t. traditional formats) cache memory usage in a shared memory parallel context. RSB is a format built atop the traditional COO and CSR formats.

Similarly to other contemporary approaches, RSB uses a two dimensional matrix partitioning in blocks; see Fig. 1 for a visual example of it. However, unlike CSB (Buluç et al. [8]) the sparse blocks dimensions are not uniform, and unlike Yzelman and Bisseling's ([9]) our techniques are not hyper-graph based. Similarly to other approaches, selection of a data structure for blocks occurs, but without using completely novel formats, as Kourtis et al. [10] do with CSX or as Belgin et al. [11] do with PBR. Unlike approaches combining dense blocking and autotuning techniques (like BCSR in SPARSITY, by Im et al. [12]) RSB does not not require the representation of excess zeroes, but still has a potential for autotuning. The closest approach we are aware of is that of Šimeček et al.; in [13], authors use a quad-tree representation for (serial) *SpMV*, but with different blocking criteria and data layout, comparing results to an in-house CSR implementation; in [14] they target specifically index space minimization, but using an uniformly-dimensioned sparse blocks approach.

Although platform specific tuning is known to give significant efficiency improvements (see the study of Williams et al. [15]), we chose not to apply it here. In this way we keep RSB algorithms general and the code portable, thus retaining the possibility of further optimizations.

RSB supports the Level 2 Sparse BLAS (see Duff et al. [16]) operations (matrix-vector multiply and matrix-vector triangular solve) and their variants, that is: diagonal implicit, symmetry, transposition, upper/lower triangle.

In this article, we compare the performance of RSB's *SpMV*, *SpMV_T*, *SymSpMV* to that of a highly tuned proprietary implementation of the CSR format: the one in the Intel Math Kernel Library (MKL). To make our contribution more complete we also measure the practical cost of assembling RSB structures, thus exposing when using our RSB implementation can save overall execution time over MKL's CSR. Additionally, we include also results obtained with the state-of-the-art format Compressed Sparse Blocks (CSB) implementation (see Buluç et al. [8],[6]). However, our main emphasis is on the RSB vs MKL-CSR comparison, in that CSB exists only as a prototypal code, whereas the former two reside in two complete Sparse BLAS oriented libraries (MKL and `librsb`, respectively), and thus are of practical interest to users.

We carry out our study on a computer equipped with a recent multi-core shared-memory processor of Intel's Sandy Bridge family.

The next section proceeds by first recalling techniques and problems of the classical COO and CSR matrix representations, then discussing consequences of their usage with block partitioned matrices, and finally outlining the RSB data structures and its matrix-vector multiply algorithm. Then the experimental setup is presented, followed by results and their discussion.

2. The RSB Format and Algorithms

2.1. Vectors and Arrays Notation

Given a matrix A with nr rows and nc columns, we denote each of its entries $a_{i,j}$ by specifying its row and column indices: $1 \le i \le nr$ and $1 \le j \le nc$. We call *nonzeroes* the entries $a_{i,j}$ which are different from zero. Sometimes we use an array notation similar to that of the well known Matlab language. For instance,

Figure 2: *COO_SpMV*(s,x,y): *SpMV* listing for a COO submatrix s. Assuming index arrays $(s.IA, s.JA)$ contain local indices translated respectively by $s.or$ and $s.oc$. Arrays x and y are global.

```
for $l \leftarrow 1$ to $s.nnz$ do
    $i \leftarrow s.IA(l) + s.or$
    $j \leftarrow s.JA(l) + s.oc$
    $y(i) \leftarrow y(i) + s.VA(l)x(j)$
end
```

Figure 3: *COO_SpMV_T*(s,x,y): *SpMV_T* listing for a COO submatrix s.

```
for $l \leftarrow 1$ to $s.nnz$ do
    $i \leftarrow s.IA(l) + s.or$
    $j \leftarrow s.JA(l) + s.oc$
    $y(j) \leftarrow y(j) + s.VA(l)x(i)$
end
```

$x(i)$ denotes the i^{th} element of array x; $x(f : l)$ the elements in the range of indices f to l; $x(:)$ denotes the whole array. Similarly, $A(i, j)$ denotes $a_{i,j}$; $A(:, j)$ denotes the j^{th} column; $A(i, :)$ denotes the i^{th} row of A.

The sum of nr sized vector y and the product of A and nc sized vector x can be expressed as $\forall 1 \leq i \leq nr, y_i \leftarrow y_i + \sum_{j=1}^{nc} a_{i,j}x_j$; in vector notation $\forall 1 \leq i \leq nr, y_i \leftarrow y_i + a_{(i,:)}x$. The corresponding *transposed* operation is defined (this time for each of y's nc entries, with nr-sized x) as $\forall 1 \leq j \leq nc, y_j \leftarrow y_j + \sum_{i=1}^{nr} a_{i,j}x_i$; in vector notation $\forall 1 \leq j \leq nc, y_j \leftarrow y_j + a^T_{(:,j)}x$.

Given a data structure instance s, we refer to its individual *fields* with a dot notation similar to Matlab's (e.g.: $s.x$ is the x field of structure instance s).

2.2. Background: the COO and CSR formats

As a background for the discussion to follow, here we introduce basic variants of two common sparse matrix storage formats (COO and CSR) which constitute the computational core of RSB, and show basic *SpMV*/*SpMV_T*/*SymSpMV* algorithms for them.

The Coordinate (COO) format represents a matrix A by encoding its nnz non-zero coefficients (*nonzeroes*) using three arrays. Two index arrays IA, JA represent respectively row and column coordinates of the nonzeroes, while array VA stores their numerical values; that is, for each $1 \leq l \leq nnz$ we have that $a_{IA(l),JA(l)} := VA(l)$. An *SpMV* algorithm for COO executes $y(IA(l)) \leftarrow y(IA(l)) + VA(l)x(JA(l))$ on the whole range of l. Correspondingly, an algorithm for *SpMV_T* iterates on $y(JA(l)) \leftarrow y(JA(l)) + VA(l)x(IA(l))$. We assume a *row major* ordering of the nonzeroes; that is, $\forall 1 \leq p < q \leq nnz$ holds either $IA(p) < IA(q)$ or the both of $IA(p) = IA(q)$ and $JA(p) < JA(q)$.

We show basic COO pseudocode listings for *SpMV* in Fig. 2 and for *SpMV_T* in Fig. 3. The listings are general enough to handle a matrix or any of its submatrices (blocks) s. So, the submatrices indices are relative to a *row offset* ($s.or$) and *column offset* ($s.oc$), both 0 based. If an entire matrix is to be represented, then $s.or = s.oc = 0$.

The second format of interest is CSR (Compressed Sparse Rows). It employs two indices arrays (PA, JA) and one values array (VA). The JA and VA arrays have the same contents as in the previously defined COO. The *compressed indices* array PA is dimensioned $nr + 1$, and for each $1 \leq i \leq nr$, $PA(i + 1) - PA(i)$ is equal to the number of nonzeroes on row i. That is, $PA(i)$ is the index of the first nonzero corresponding to row i within JA and VA. If row i is empty, then $PA(i + 1) = PA(i)$. Any *SpMV* algorithm for CSR is equivalent to the execution of $y(i) \leftarrow y(i) + VA(PA(i) : PA(i + 1) - 1)x(JA(PA(i) : PA(i + 1) - 1))$ on each non empty row i.

Analogously, the *SpMV_T* variant is equivalent to $y(JA(l)) \leftarrow y(JA(l)) + VA(l)x(i)$, where $PA(i) \leq l \leq PA(i + 1) - 1$.

Basic CSR pseudocode for *SpMV* is shown in Fig. 4; for *SpMV_T*, in Fig. 5.

Figure 4: *CSR_SpMV*(s,x,y): *SpMV* listing for a CSR submatrix s.

```
for $i \leftarrow 1$ to $s.nr$ do
    for $l \leftarrow s.PA(i)$ to $s.PA(i+1) - 1$ do
        $j \leftarrow s.JA(l)$
        $y(i + s.or) \leftarrow y(i + s.or) + s.VA(l)x(j + s.oc)$
    end
end
```

Figure 5: *CSR_SpMV_T*(s,x,y): *SpMV_T* listing for a CSR submatrix s.

```
for $i \leftarrow 1$ to $s.nr$ do
    for $l \leftarrow s.PA(i)$ to $s.PA(i+1) - 1$ do
        $j \leftarrow s.JA(l)$
        $y(j + s.oc) \leftarrow y(j + s.oc) + s.VA(l)x(i + s.or)$
    end
end
```

Both the COO and CSR formats support a symmetric storage variant, where either the upper or the lower triangle of A can be omitted from representation. The corresponding symmetric matrix-vector multiplication variant (*SymSpMV*) is equivalent to performing *SpMV* on one triangle of A and *SpMV_T* on the off-diagonal elements of the same triangle, but with a single read of the matrix arrays. This is a gain in efficiency if compared to executing *SpMV*, *SpMV_T* in sequence, because with no additional matrix memory accesses, the ratio of compute operations to memory accesses is almost doubled. See the example listings *COO_SymSpMV* in Fig. 6 and *CSR_SymSpMV* in Fig. 7. Please note that depending on the assumptions on the matrix diagonal the innermost check in both listings could be safely omitted in some cases; for instance, when assuming a *diagonal implicit* or a lower triangle representation.

2.3. Background: Serial COO and CSR SpMV kernels for sparse blocks

As previously mentioned, the role of *JA*, *VA* arrays is the same in COO and CSR. The remaining arrays are the nnz sized *IA* for COO and the $nr + 1$ sized *PA* for CSR. With matrices encountered in common applications having $nr < nnz$, using a CSR representation rather than a COO one requires fewer index entries, because row indices information is compressed in the *PA* array instead of being explicitly stored as in *IA*. Assuming the same integer representation for PA, IA, JA (e.g.: C's `int` type) and $nr \ll nnz$, a CSR representation can save almost half of the indexing storage required by COO. Assuming an 8 bytes type for the numerical coefficients in *VA* and 4 bytes for the index type, a COO representation uses exactly $8 + 4 + 4 = 16$ bytes per nonzero, while CSR uses $((8 + 4)nnz + 4nr)/nnz$. That is, CSR uses between 12 and 16 bytes per nonzero, so up to 25% less than COO. This saving is beneficial in terms of memory traffic required to read the matrix arrays during the multiplication.

Figure 6: *COO_SymSpMV*(s,x,y): *SymSpMV* listing for a COO submatrix s.

```
for $l \leftarrow 1$ to $s.nnz$ do
    $i \leftarrow s.IA(l) + s.or$
    $j \leftarrow s.JA(l) + s.oc$
    $y(i) \leftarrow y(i) + s.VA(l)x(j)$
    if $i \neq j$ then
        $y(j) \leftarrow y(j) + s.VA(l)x(i)$
    end
end
```

Figure 7: *CSR_SymSpMV*(s,x,y): *SymSpMV* listing for a CSR submatrix s.

```
for $i \leftarrow 1$ to $s.nr$ do
    for $l \leftarrow s.PA(i)$ to $s.PA(i+1)-1$ do
        $j \leftarrow s.JA(l)$
        $y(i+s.or) \leftarrow y(i+s.or) + s.VA(l)x(j+s.oc)$
        if $i \neq j$ then
            $y(j+s.oc) \leftarrow y(j+s.oc) + s.VA(l)x(i+s.or)$
        end
    end
end
```

Apart from the amount of memory occupation, with the assumptions made so far, the shown CSR and COO kernels have similar memory access patterns, that we here summarize. The *VA*, *JA* arrays are traversed sequentially one location forward at a time in (both COO and CSR). The x array is read at irregularly spaced locations, specified by the column indices in *JA*. In case of the *SpMV* access type (Fig. 2, Fig. 4) the same columns, and thus x locations might be accessed repeatedly across successive rows; this is often the case when the matrix has some structure. At each access, an entire cache line (on the machine of our interest, 64 bytes, that is 8 x array locations) is loaded; if matrix rows are too long, eviction can occur before the cache line being reused in a subsequent row. This is cause of a big inefficiency: in addition to the latency due to cache misses, a relevant fraction of bandwidth (that of accessing one array out of five) is wasted. Access to y is different: since rows are visited in a strictly ascending order, potential of cache lines reuse is retained across consecutive memory locations updates. In the case of *SpMV_T* ($Fig.\ 3, Fig.\ 5$), the *cacheability* of x and y arrays is reversed: successive locations of x are read in one traversal (so, for a total of nr read elements), while a location of y is updated for each *JA* location (so, for a total of nnz updates). Since updating a memory location is more expensive than merely reading it, *SpMV_T* are slower when using a row major ordering, as it is the case here. Symmetric kernels ($Fig.\ 6, Fig.\ 7$) exhibit the behavior of both *SpMV* and *SpMV_T* at once.

Traversal of the *IA* array in COO and *PA* in CSR occurs only once and sequentially; as mentioned, the size of *IA* may exceed that of *PA* in large measure. Although generally being an advantage of CSR over COO, this may lead to cache misses on *PA* in case of very long rows, just as with y in *SpMV*.

In practice, there is a vast number of optimizations that can be applied to the base COO and CSR kernels we listed. We describe a very common one that can improve write access to the y array. Assuming $nr \ll nnz$, it is possible to accumulate the contributions for the y array in an auxiliary variable y_{aux}, thus postponing the update of y (line 4 in Fig. 4). The effective array update would be placed just outside the inner loop, with a statement like $y(i + s.or) \leftarrow y(i + s.or) + y_{aux}$. The memory access request rate would be then reduced down to one location per row. One can apply a similar optimization also to COO by rearranging the *COO_SpMV* listing (Fig. 2) in two loops: an outer one iterating over consecutive row indices, and an inner one iterating over consecutive column indices of the same row.

Another simple and effective optimization compatible with the aforementioned one can be *unrolling* (either explicitly or relying on compiler support) the inner of the two loops by a specified amount. In this way fewer loop control instructions would have to be executed, while the compiler may still have the possibility of arranging the arithmetic and memory store/load instructions efficiently.

The two aforementioned techniques contribute to the efficiency of CSR and are often used. In general, they assume $nr < nnz$ (which is true for most commonly encountered matrices) and are most effective when $nr \ll nnz$. However, by considering an arbitrary rectangular submatrix of a common matrix, this property is not generally valid, and the mentioned optimization techniques may not be effective anymore. Indeed, in the case of a submatrix with $nnz < nr$, CSR storage would use $(nr + 1 - nnz)$ integer index entries more than COO. Some authors call this property "hyper-sparsity".

It is now evident that given an arbitrary subdivision in sparse blocks (*submatrices*): 1) traditional COO and CSR optimization strategies may be less effective; 2) it is not obvious which representation (between

COO and CSR) can be more advantageous; 3) a different class of optimizations may have to be considered.

2.4. The RSB format and parallel sparse matrix-vector multiply

RSB (Recursive Sparse Blocks) is a hierarchical representation format conceived to work with algorithms that are both parallel and cache-efficient. It is based on the recursive partitioning of a matrix in quadrants [7]. At the root of recursion, the $(nr \times nc)$ sized matrix A is subdivided as follows:

$$A = \begin{vmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{vmatrix} \tag{1}$$

where A_{11} is dimensioned $(\lceil nr/2 \rceil \times \lceil nc/2 \rceil)$, A_{12} is $(\lceil nr/2 \rceil \times \lfloor nc/2 \rfloor)$, A_{21} is $(\lfloor nr/2 \rfloor \times \lceil nc/2 \rceil)$, A_{22} is $(\lfloor nr/2 \rfloor \times \lfloor nc/2 \rfloor)$. Each nonempty quadrant is subdivided further according to the same criterion. A data structure is allocated for each nonempty quadrant, thus defining a tree structure with *submatrices* as nodes. Recursion of subdivision terminates when a condition on the submatrix (rows, columns, and nonzeroes count) and machine parameters is satisfied; submatrices can be regarded as *cache blocks*. Nonzeroes are stored in *leaf* submatrices only. A leaf submatrix format can be either COO or CSR. When appropriate, 16 bit (C's `short unsigned int`) indices are used instead of the default 32 bit ones in COO's IA, JA, or for CSR's JA arrays (recall Section 2.2); this feature was first introduced in RSB in [17]. We discuss subdivision and indices choice criteria briefly in Section 2.5.

Within the matrix, leaf submatrices arrays are laid out in memory in a succession following the subdivisions (that is, *depth first*). This leads to a so-called Z-Morton layout (after Morton [18]); see Fig. 1 for a visual representation of it.

The matrix-vector product operation could be reorganized into descending the tree structure recursively and performing the computation at the leaf submatrices level, following this ordering:

$$\begin{vmatrix} y_1 \\ y_2 \end{vmatrix} \leftarrow \begin{vmatrix} y_1 \\ y_2 \end{vmatrix} + \begin{vmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{vmatrix} \begin{vmatrix} x_1 \\ x_2 \end{vmatrix} \equiv \begin{vmatrix} y_1 \\ y_2 \end{vmatrix} \leftarrow \begin{vmatrix} y_1 \\ y_2 \end{vmatrix} + \begin{vmatrix} A_{11}x_1 + A_{12}x_2 \\ A_{21}x_1 + A_{22}x_2 \end{vmatrix} \tag{2}$$

This decomposition relies on the independence of the y_1 and y_2 updates, thus suggesting that an implementation can use two distinct threads to compute them.

Since it is often the case that submatrices stored successively are adjacent (either vertically or horizontally — see Fig. 1), if the leaf submatrices are *small enough*, the chance of reusing cached x, y locations across submatrices visits during a multiply execution is increased.

We employed this approach of tree descending and dual threaded parallelism in [7] with satisfying results. However, in order to achieve a higher degree of parallelism we developed an additional algorithm, operating on the individual submatrices but still retaining some of the aforementioned locality properties.

Let us assume S is the array containing (references to) A's N_s leaf submatrices. Then, $\forall\ 1 \leq si \leq N_s,\ s = S(si)$ represents a rectangular submatrix of A, extending between rows $1 + s.or$ and $s.nr + s.or$ and columns from $1 + s.oc$ to $s.nc + s.oc$. If we consider $A_{\pi(si)}$ to be the $nr \times nc$ dimensioned matrix (so, fully dimensioned) containing only the nonzeroes in $S(si)$, then:

$$A = \sum_{si=1}^{N_s} A_{\pi(si)} \;\Rightarrow\; y + Ax = y + \sum_{si=1}^{N_s} A_{\pi(si)}x \tag{3}$$

By arranging the multiplication algorithm according to this decomposition, each s contributes to the update of a delimited interval of y and reads only a part of x. In array notation, for each $s = S(si)$:

$$\begin{gathered} y(1 + s.or : s.nr + s.or) \leftarrow y(1 + s.or : s.nr + s.or) + \\ A(1 + s.or : s.nr + s.or, 1 + s.oc : s.nc + s.oc) \cdot \\ x(1 + s.oc : s.nc + s.oc) \end{gathered} \tag{4}$$

Here, multiple execution threads can be employed at once in the computation of different contributions to y, each corresponding to a different submatrix. In the general case there could be multiple submatrices contributing to a given interval of y. The y array is written directly: no auxiliary buffer array is used. Because of the lack of *atomic* updates for floating point number arrays on the computer architectures of our interest ([19, V.3, Ch. 8.1]), multiple simultaneous updates to the same entries of y's array might cause a *race condition*, and consequently lead to incorrect results.

We use a synchronization technique to avoid this problem: when a thread is associated to a submatrix s (or, *picks* the submatrix), the interval $1 + s.or ... s.nr + s.or$ gets *locked* until operation on s is complete. As long as the interval is locked, no other thread is allowed to use any submatrix whose rows interval intersects s's. We implemented the lock mechanism in our code using the `omp parallel` and `omp critical` OpenMP [20] directives only.

In our implementation, the temporal order in which submatrices are multiplied is non deterministic; each thread keeps reading a shared bits vector to identify submatrices not visited yet. When a thread finds an unvisited submatrix on an available rows interval, it marks the interval as locked and the submatrix as visited, so no other thread can use it. Having potentially several threads reading concurrently a shared array might cause inefficiencies due to the cache coherency mechanisms. The cost of our locking technique is difficult to estimate theoretically, so we have performed an experiment to do so empirically; the experiment is described in Section 4.4. Although a non negligible impact has been observed on some matrices, we do not find this to be problematic. However, it is clear that in a perspective of even more executing threads contention is expected to grow, so we might consider alternative techniques as future work.

With slight modifications, the described *SpMV* algorithm for RSB has been extended to handle *SpMV_T* and *SymSpMV* as well. Adapting to *SpMV_T* is straightforward: locking is applied to the submatrices columns' intervals and *SpMV_T* kernels are invoked instead; by itself, transposed operation does not involve additional locking costs. Adjustment for *SymSpMV* is different: locking of both the rows and columns intervals is required to allow safe update of the two corresponding intervals of y. As a consequence of the doubled amount of locked intervals, potential parallelism of *SymSpMV* is less than for *SpMV*.

Pseudocode implementing either *SpMV*, *SpMV_T* or *SymSpMV* for RSB is sketched in Fig. 8.

2.5. Assembly of RSB matrices

In this section we give an outline of the *CooToRSB* procedure used to assemble an RSB matrix from the three row-major ordered COO arrays (*IA*, *JA*, *VA*, with 32 bit indices — recall Section 2.2); details of this procedure (such as pseudocode listings) have been published in [21]. RSB is a hybrid format representing submatrices as either CSR or COO, eventually using 16 bit indices instead of the common 32 bit ones. Given input matrix arrays, the exact subdivision in submatrices is determined after a recursive subdivision process. Within *CooToRSB*, we distinguish a first phase where only symbolic information about the destination submatrices is collected (*SubdToRSB*), and a second one, where the individual submatrices arrays are populated (*ShufToRSB*). The two phases operate consecutively, and both can be implemented as shared memory parallel.

SubdToRSB scans the input COO arrays repeatedly to identify quadrant submatrices. It is implemented with binary search operations and uses auxiliary compressed indices arrays; the amount of required *work* memory does not exceed a few times the original indexing amount. As submatrices are identified, they are considered for further subdivision.

Subdivision of different submatrices can be performed in parallel. Because of the irregular structure of most matrices, the participating threads need to coordinate when choosing which submatrices to operate on. Corresponding information is gathered in parallel as submatrices are being subdivided. This algorithm is non deterministic; namely executing *SubdToRSB* on the same input might produce different partitionings in different runs, because threads scheduling may cause a different runtime of some threads, and thus a different sequence of matrices considered for subdivision. Because of the dependency among individual subdivisions, parallelism of *SubdToRSB* is very limited in the beginning of the construction process; once a sufficient number of subdivisions has been produced, more threads can start to work.

Terminating subdivision on a given submatrix and selecting a format for it proceeds according to a rule, which takes into account: 1) a user specified or system detected *cache size* parameter, representative of the amount of memory each submatrix shall ideally occupy in order to favor cache efficient computations; 2) the number of available concurrent threads: subdivision is finer, the more threads are available.

Figure 8: *RSB_SpMV*(S,x,y)/*RSB_SpMV_T*(S,x,y)/*RSB_SymSpMV*(S,x,y): Sketch of multithreaded *SpMV*/*SpMV_T*-/*SymSpMV* for leaf submatrices of a RSB matrix. It operates on the submatrices array S and arrays x, y. For simplicity, we do not specify the lock mechanism. After the **continue** statement the thread execution continues at the next iteration in the loop.

```
S ← $(s_0, s_1, \ldots, s_{N_s-1})$ /*array of leaf submatrices*/
B ← $(0, 0, .., 0)$ /*a zero bit for each unvisited submatrix in S*/
n ← 0 /*count of visited submatrices*/
begin parallel
while $n < N_s$ /*enter the loop if any submatrix is unvisited*/ do
    begin critical
    $s$ ← pick up a submatrix $s = s_{si} = S(si)$ such that $B(si) = 0$ ($s$ is a yet unvisited submatrix)
    if want SpMV then
        $(f, l)$ ← (1+s.$or$, s.$or$+s.nr)
        if interval $(f, l)$ is locked then continue
        lock interval $(f, l)$ /*lock y on s's rows interval*/
    end
    if want SpMV_T then
        $(f', l')$ ← (1+s.$oc$, s.$oc$+s.nc)
        if interval $(f', l')$ is locked then continue
        lock interval $(f', l')$ /*lock y on s's columns interval*/
    end
    if want SymSpMV then
        $(f, l)$ ← (1+s.$or$, s.$or$+s.nr)
        $(f', l')$ ← (1+s.$oc$, s.$oc$+s.nc)
        if either of $(f, l)$ or $(f', l')$ intervals is locked then continue
        lock interval $(f, l)$ /*lock y on s's rows interval*/
        lock interval $(f', l')$ /*lock y on s's columns interval*/
    end
    $B(si)$ ← 1 /*mark submatrix $s_{si}$ as visited*/
    $n$ ← $n + 1$ /*increment visited submatrices counter*/
    end critical
    if want SpMV then
        if s is stored as COO then call COO_SpMV(s,x,y)
        if s is stored as CSR then call CSR_SpMV(s,x,y)
    end
    if want SpMV_T then
        if s is stored as COO then call COO_SpMV_T(s,x,y)
        if s is stored as CSR then call CSR_SpMV_T(s,x,y)
    end
    if want SymSpMV then
        if s is stored as COO then call COO_SymSpMV(s,x,y)
        if s is stored as CSR then call CSR_SymSpMV(s,x,y)
    end
    begin critical
    if want SpMV then unlock interval $(f, l)$
    if want SpMV_T then unlock interval $(f', l')$
    if want SymSpMV then unlock intervals $(f, l)$ and $(f', l')$
    end critical
end
end parallel
```

Then, the amount of memory accesses necessary for performing *SpMV* on a given submatrix s is estimated; if this amount is considered *small enough*, then subdivision terminates and s is kept as a *leaf submatrix*. If not, the submatrix is marked for further subdivision. The mentioned formula resembles a rough estimate of the *memory footprint* of CSR's *SpMV*: 1) the total extension of the submatrix storage arrays, to be read once; 2) the extent of the output vector y, to be updated once per location; 3) the extent of the input vector x, to be read possibly multiple times (no more than $s.nnz$ accesses). An important property of this mechanism is that only submatrices with more nonzeroes than rows will be stored as CSR; otherwise COO will be used. Submatrices dimensioned less than 2^{16} will use 16 bit indices for *JA* (in both COO and CSR) and *IA* (only in COO) arrays; remaining ones will use the default 32 bit indices. The interested reader can find details in [21].

ShufToRSB operates by visiting the tree structure generated by *SubdToRSB* and *shuffling* the original (input, unmodified) row-major ordered COO arrays according to the new ordering of submatrices, laying each one consecutively by following the recursion tree. As mentioned, this defines a Z-Morton ordering of the *sparse blocks*. The *ShufToRSB* phase can operate with a fairly high degree of parallelism if the matrix has been partitioned in enough submatrices: different threads will shuffle the arrays in parallel. After *ShufToRSB* the original *IA*, *JA*, *VA* arrays will be rearranged to host COO and CSR arrays of the different submatrices, at offsets determined by *SubdToRSB*.

Please see Fig. 1 for an example of a large matrix partitioned in the case of different thread numbers. Several optimizations may be applied to our algorithm for specific instances of the assembly problem, but we keep them as future work.

We present and discuss speed results of our RSB assembly implementation in Section 4.5.

2.6. The Compressed Sparse Blocks (CSB) format

CSB is a format of recent introduction. Since its inception (see Buluç et al. [8]) it has been devoted to provide equally efficient *SpMV*/*SpMV_T* and reduced index occupation (or better, reduced memory bandwidth at runtime operation). It has been recently extended to handle *SymSpMV* and it shows room for further improvement [6]. This format shares the following ideas with RSB: a) rearrangement of the matrix in *sparse blocks* sized roughly according to the cache size, and dynamic scheduling in the processing of these blocks; b) increased symmetry (w.r.t. to formats like CSR or CSC) in the performance of *SpMV*/*SpMV_T*; c) exploiting the locality properties of the Z-Morton curve: RSB arranges the sparse blocks on a two dimensional Z-Morton curve, where CSB does so for the individual nonzeroes of each sparse block.

The main differences with RSB are: a) CSB achieves shared memory parallelism by employing the "Cilk" extension syntax and runtime system for C++, whereas RSB uses OpenMP [20]; b) the symmetric format and algorithms of CSB differ significantly from the unsymmetric ones; c) the current implementation of CSB relies on machine specific optimizations (unlike RSB); d) CSB uses $O(nnz) + O(nr \cdot nc)$ storage for indices, where RSB uses $O(nnz)$. The CSB format algorithms differ substantially from RSB's or CSR's. For further details, please refer to the works of Buluç et al.: [8] and [6].

3. Experimental Setup and Methodology

We performed our experiments on a computer equipped with two *"Intel Xeon E5-2680"* ("Sandy Bridge") CPUs, 8 cores each. Each such CPU has 3 levels of caches; associativity/line size/capacity parameters are respectively for L1-data: 8/64/32KB, for L2-unified: 8/64/256KB, and for L3: 20/64/20MB. We run our benchmark code to up to 16 (OpenMP) executing threads; for brevity we consider only 1 and 16 threaded results. We compare our results to those of the proprietary, highly optimized Intel's *Math Kernel Library* (version string: *"MKL 10.3-7, Product, 20111003, Intel(R) Advanced Vector Extensions (Intel(R) AVX) Enabled Processor, Intel(R) 64 architecture"*) routines for CSR stored matrices; specifically, we compare with the `mkl_dcsrmv` routine. We chose the `double` precision numerical representation; we do not consider other representations for brevity reasons. According to the Intel MKL manual ([22, p. 2712]), elements within each CSR row shall be strictly ordered by column, and both the *row pointers* and *column indices* arrays have to be represented with 4 byte signed integers (C's `int`). We have chosen to compare our implementation of RSB to MKL'S CSR implementation for two reasons: a) the CSR format is a traditional, well known and widely used format; b) MKL is a widely used, highly efficient proprietary (as such, closed source) library specifically optimized for the CPUs as the one we use. Thus, this study aims to compare our RSB implementation with that of a reference standard real-world library using a typical sparse matrix format. We compiled our C99 ([23]) code with the Intel `icc` compiler (*"Intel(R) C Intel(R) 64 Compiler XE for applications running on Intel(R) 64, Version 13.0.1.117 Build 20121010"*); thread parallelism is obtained by means of OpenMP; we use the `-O3 -xAVX -fPIC -openmp` compilation flags.

We only use publicly available matrices among the largest ($> 10^7$ nonzeroes) from the *University of Florida Sparse Matrix Collection* (see Davis and Hu [24]). Such matrices do not fit in the cache memory: storage of 10^7 nonzeroes requires twice the total amount of available L3 cache already for the numerical arrays alone. See Table 1 for the list of matrices we consider. These matrices originate from a variety of different problems in science, engineering, information technology. Of these matrices, two are non-square: *GL7d19* (1911130×1955309) and *relat9* (12360060×549336). For plotting convenience, we shortened the name of matrix *channel-500x100x100-b050* to *channel-500x100.* We measure the performance by the

matrix	symm	nr	nc	nnz	nnz/nr
arabic-2005	G	22744080	22744080	639999458	28.14
audikw_1	S	943695	943695	39297771	41.64
bone010	S	986703	986703	36326514	36.82
channel-500x100x100-b050	S	4802000	4802000	42681372	8.89
Cube_Coup_dt6	S	2164760	2164760	64685452	29.88
delaunay_n24	S	16777216	16777216	50331601	3.00
dielFilterV3real	S	1102824	1102824	45204422	40.99
europe_osm	S	50912018	50912018	54054660	1.06
Flan_1565	S	1564794	1564794	59485419	38.01
Geo_1438	S	1437960	1437960	32297325	22.46
GL7d19	G	1911130	1955309	37322725	19.53
gsm_106857	S	589446	589446	11174185	18.96
hollywood-2009	S	1139905	1139905	57515616	50.46
Hook_1498	S	1498023	1498023	31207734	20.83
HV15R	G	2017169	2017169	283073458	140.33
indochina-2004	G	7414866	7414866	194109311	26.18
kron_g500-logn20	S	1048576	1048576	44620272	42.55
Long_Coup_dt6	S	1470152	1470152	44279572	30.12
nlpkkt120	S	3542400	3542400	50194096	14.17
nlpkkt160	S	8345600	8345600	118931856	14.25
nlpkkt200	S	16240000	16240000	232232816	14.30
nlpkkt240	S	27993600	27993600	401232976	14.33
relat9	G	12360060	549336	38955420	3.15
rgg_n_2_23_s0	S	8388608	8388608	63501393	7.57
rgg_n_2_24_s0	S	16777216	16777216	132557200	7.90
RM07R	G	381689	381689	37464962	98.16
road_usa	S	23947347	23947347	28854312	1.20
Serena	S	1391349	1391349	32961525	23.69
uk-2002	G	18520486	18520486	298113762	16.10

Table 1: Matrices used for our experiments. Symmetric are marked by S, general unsymmetric by G.

conventional "floating point operations per second" metric; that is, for each of the matrix nonzeroes, we canonically count two operations, and divide by the (wall clock) time the operation took. We measured timings using the POSIX ([25]) `gettimeofday()` function. We consider the minimal time after 50 repetitions of the operation (either *SpMV*, *SpMV_T* or *SymSpMV*), or 5 repetitions in the case of the matrix assembly operation. All our measurements were performed with *hot caches*; that is, we deliberately did not flush cache contents in between subsequent multiply calls. Nevertheless, we exclude accidental reuse of cached locations across the calls, because all measurements were performed on matrices much larger than the total of last level cache. In this way we avoid artificially high results. We use the following linking flags for MKL: "`-lm -lmkl_solver_lp64 -lmkl_intel_lp64 -lmkl_gnu_thread -lmkl_core`".

Additionally to measurements of RSB and MKL timings, we also considered the state-of-the-art CSB format from the proof-of-concept code Buluç et al. used in their 2011 paper [6]. Here, we used the same compiler suite, and C++ compilation flags as: `-O3 -xAVX -fPIC -fno-rtti -parallel -restrict -fno-inline-functions`. We modified slightly the CSB code to use the same benchmarking criteria as above.

4. Performance Results and Discussion

4.1. Notation, presentation notes

When discussing results, by "MKL" we intend MKL's CSR implementation of the `mkl_dcsrmv` routine (MKL's driver for *CSR_SpMV*, *CSR_SpMV_T*, *CSR_SymSpMV*). Since the sparse matrix-vector multiply operation is implemented with slightly different algorithms for symmetric (*SymSpMV*) and unsymmetric matrices (*SpMV*, *SpMV_T*), we use different figures for their results (Figs. 9, 10, 11, 16 pertain to unsymmetric matrices; Figs. 12, 14, 15, 17 pertain to symmetric ones). Since *SpMV* and *SpMV_T* algorithms are conceptually similar and apply to the same matrices, we chose to present their results on the same figure. In the figures' legend acronyms, transposed (untransposed) results are marked with a 'T' ('N') preceding the number of considered parallel threads, and following the implementation label (either 'MKL', 'CSB' or 'RSB').

4.2. Unsymmetric matrices: SpMV, SpMV_T

In the *SpMV* and *SpMV_T* results (Fig. 9), our first observation is that in contrast to MKL's CSR, RSB's *SpMV_T* performs almost the same as *SpMV*. Of the 7 considered matrices, 5 exhibit better RSB performance for *SpMV_T* than for *SpMV*, while MKL performance is always better for *SpMV*. For the *tall* ($nr > nc$) *relat9* this is especially marked (2.4 vs 1.3 GFlops — almost twice); we speculate it is so because a much shorter vector is updated during *SpMV_T* than *SpMV* (549336 vs 12360060 elements, so circa one twentieth), leading to higher chances of cache hit. Comparing MKL's *SpMV* to RSB's, most of the matrices favor RSB, being faster by some 15 to 50%. Exceptions are the information retrieval matrix *indochina-2004*, being multiplied almost twice as fast with RSB; *relat9*, giving 2/3 of MKL's performance; *GL7d19* faster by some 15% with MKL. A reason for *relat9*'s poor RSB performance is probably the index usage, circa 50% higher than CSR (7.92 vs 5.27 bytes/nnz — see Fig. 10); only matrix *GL7d19* had a significant increase in index usage compared to CSR, and it's also outperformed by it. Beside the balance in memory traffic (more for indices, less for numerical data), almost all of *relat9*'s submatrices (an most of *GL7d19*'s) are stored in the COO format, which is known not to be very efficient for *SpMV*.

In no case MKL performed *SpMV_T* faster than RSB; the measured performance ratios ranged from 1.16 (*arabic-2005*) to 3.8 (*GL7d19*).

In *SpMV* comparison with CSB, RSB has performed substantially better (more than twice the speed) in one case only, that of *indochina-2004*. In two cases (*HV15R* and *RM07R*) the results of CSB and RSB were very similar, for both *SpMV*/*SpMV_T*. CSB-16's *SpMV* has performed better than RSB-16 on *arabic-2005* and *GL7d19*; considerably (50%–100%) better on *relat9* and *uk-2002*. On graph matrices *arabic-2005*, *relat9*, *uk-2002*, CSB has not been able to provide symmetric performance: the transposed case is roughly slower by half; in contrast, RSB has been able to provide symmetric performance for all matrices except the tall *relat9*. Indeed, in the case of *SpMV_T*, RSB-16 has been found to be slower than CSB in one case only (*GL7d19*); in most remaining cases it has been faster than CSB's by 10–30%.

A different view over these results is that of parallel scaling; that is the ratio of 16 threaded execution performance to the single threaded one (Fig. 11). The best *SpMV*/*SpMV_T* scaling results are for CSB,

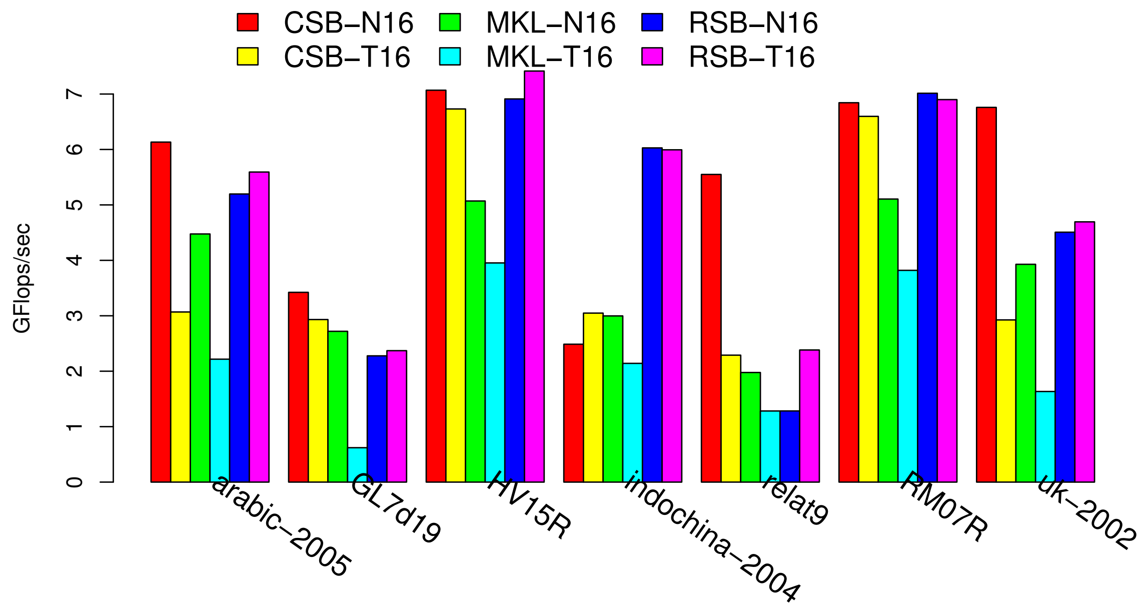

Figure 9: Unsymmetric matrices. *SpMV* and *SpMV_T* performance, for 16 threaded CSB, MKL and RSB.

reaching around 10× in most cases; the best scaling for RSB is 9×, reached on one case. In most of the cases, *SpMV*/*SpMV_T*'s RSB scales up to 4–5×, followed by MKL's *SpMV*.

MKL's best *SpMV* scaling is for matrix *GL7d19* (7.86×); for *SpMV_T* it is 3.13×, on *relat9*. MKL's *SpMV_T* scaling is usually inferior to 3×. For RSB, *SpMV* and *SpMV_T* scaling are close (within some 20% of difference), for each matrix. Similarly for CSB; except in the cases *arabic-2005* and *uk-2002*, where *SpMV* scales twice as *SpMV_T* (it's not clear to us why). The reason for the very high scalability properties of CSB is its relatively low performance for the single threaded case. CSB requires also less index usage (around 4 bytes per nonzero, whatever the matrix), but presumably more instructions (e.g.: indices arithmetic) to execute, so it pays off with more executing threads.

Closely related to the high index usage of some matrices is the (very low) density of nonzeroes per row, and the consequent negligible cache reuse of the vectors involved in *SpMV*/*SpMV_T*; *GL7d19* and *relat9* have the lowest densities (respectively, 19 and 3) among their group (see Table 1) and give the poorest performance.

We chose not to present the throughput of intermediate choices of threads; however we collected data also for the 8 threaded case and we comment it briefly. Although the relative results of MKL and RSB are pretty similar here, in a couple of cases *SpMV* and *SpMV_T* performance was better (by a few percent) with 8 threads. In all cases, MKL's *SpMV_T* implementation exhibited the same performance for 8 and 16 threads. The CSB results for 8 threads were in all cases inferior to the 16 threaded ones.

4.3. Symmetric matrices: SymSpMV

On a symmetric matrix $A(= A^T)$, the result of transposed multiply is the same as untransposed ($Ax = A^T x$). Both MKL's CSR and our RSB implementation of *SymSpMV* take advantage of the symmetry, so for each nonzero coefficient being read, two corresponding result vector entries are updated (by addition) instead of one. The same holds for CSB, although here the data structure differs significantly from the unsymmetric case (see [6, Sec.IV]). This leads to a write-to-read ratio higher than in *SpMV*/*SymSpMV*, therefore leading to a higher average floating point performance.

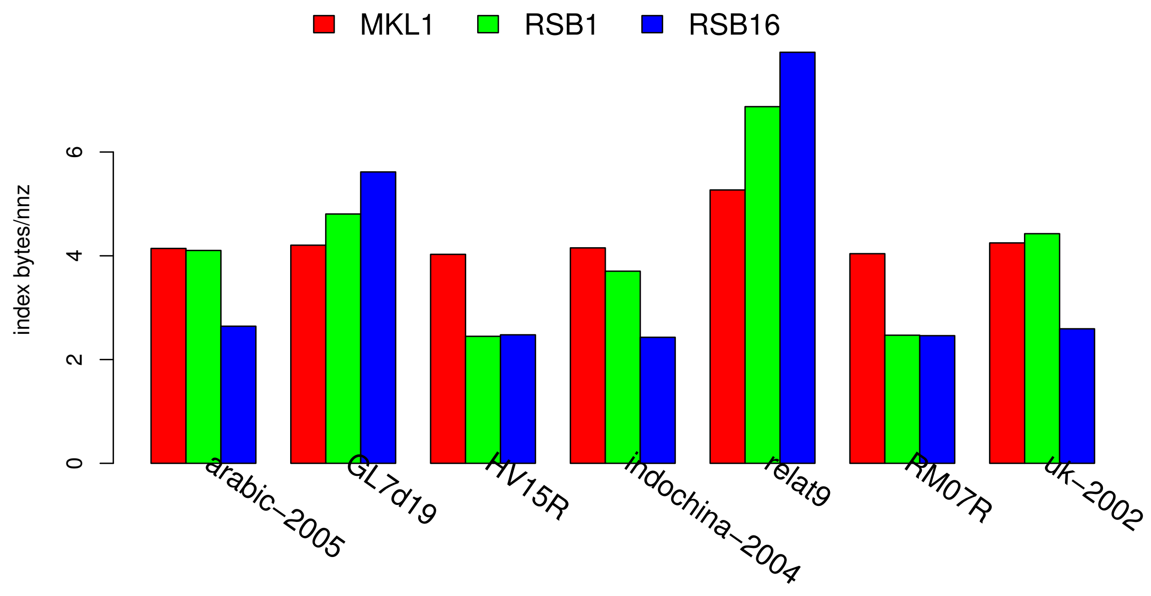

Figure 10: Unsymmetric matrices. Index bytes per nonzero for representing either 1 or 16 threaded RSB and CSR. Please note that unlike RSB, CSR index usage is independent from thread count. We omit reporting values for CSB since all matrices used almost the same (4–4.02) index bytes/nnz.

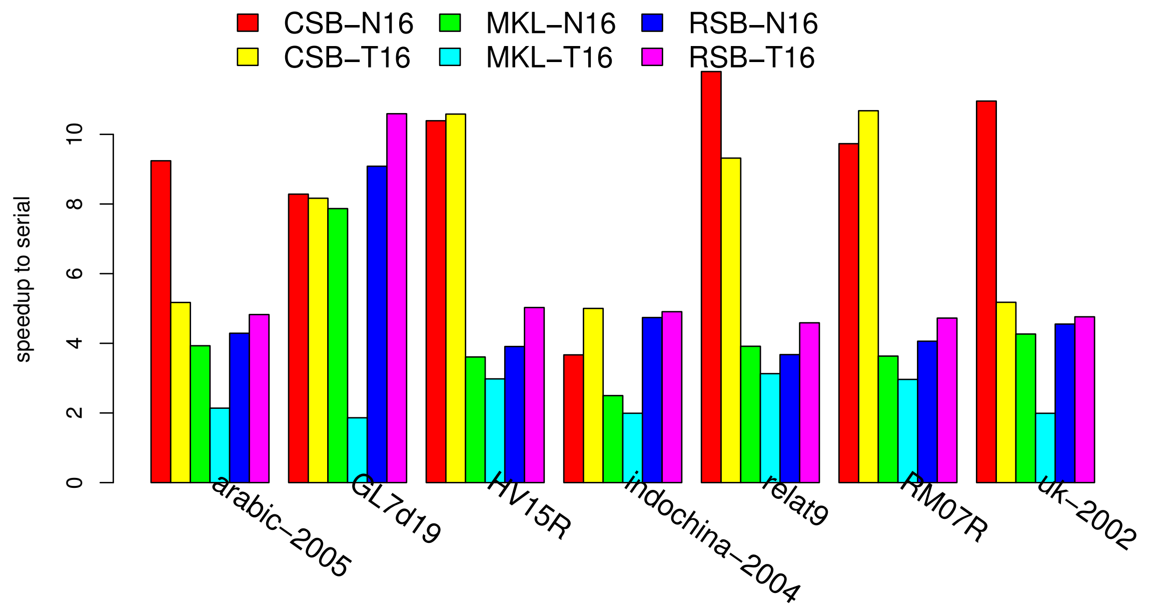

Figure 11: Unsymmetric matrices. *SpMV*/*SpMV_T* parallel speedup (16 to 1 threads performance ratio).

Indeed, results (see Fig. 12) are generally better than in the unsymmetric cases; moreover, at a first glance we notice an even higher advantage of RSB over MKL. The difference in performance ranges from 4% (8.2 vs 7.9 GFlops for *audikw_1*) to 217% (1.52 vs 0.48 GFlops for *road_usa*). The best results are circa 13 GFlops for RSB (*Flan_1565*) and around 8 GFlops for MKL (*Cube_Coup_dt6, audikw_1*); the worst ones are (in both implementations) with *europe_osm* (1.1 GFlops for RSB, 0.4 GFlops for MKL). CSB results are often better than MKL for matrices where both of RSB and MKL are relatively slow, and worse for the matrices exposing peak performance. Comparing RSB-16 to CSB-16 in the symmetric case shows similar results on matrix *gsm_106857*. In one case (*europe_osm*) CSB-16 performed more than twice as fast as RSB-16. Other cases where CSB-16 is better are: *channel-500x100x100-b050*, *delaunay_n24*, *rgg_n_2_23_s0*, *rgg_n_2_24_s0*, *road_usa*. On all the remaining symmetric cases, RSB-16 has been found to be superior to CSB-16; often twice as fast. For three matrices (*channel-500x100x100-b050*, *rgg_n_2_23_s0*, *rgg_n_2_24_s0*) CSB resulted to be the best format.

With single threaded RSB, matrices' index occupation is near to that of CSR (see Fig. 14). When using 16 threads, occupation is lower: mostly by almost 30–40%, in the range of 2.4–4 bytes/nnz.

With certain matrices (*delaunay_n24*, *europe_osm*, *road_usa*), indices occupy much with both CSR and RSB. Furthermore, with these matrices RSB indices occupy more with 16 threads than with 1. It is easy to see (Fig. 12) that the matrices leading to consistently the worst performance for both implementations (*delaunay_n24, europe_osm, road_usa, gsm_106857, kron_g500-logn20*) are also the ones with the highest index per nonzero average occupation. Indeed, most of these have a very low nonzeroes per row ratio, which forces the assembly algorithm to select often the COO format for their submatrices (recall that COO submatrices dimensioned less than 2^{16} are being stored with 4 bytes of indices per nonzero, while larger ones with 8 bytes of indices per nonzero). CSB employs very little (roughly 4 bytes) index data per nonzero, whatever the matrix. This is very likely to be the main factor in its relatively good performance with the above mentioned matrices.

Exception made for one case, the scaling (see Fig. 15) measure is better for RSB's *SymSpMV* than for MKL's. Best scaling for RSB is reached with *rgg_n_2_24_s0* (7.26×), while for MKL it is reached with *Cube_Coup_dt6* (6.14×). Matrix *rgg_n_2_24_s0* is also the one where RSB scales the most (9.14×, versus MKL's 3.02×).

Indeed, higher parallel performance and scaling of RSB correlates with smaller differences in performance for the single threaded (RSB vs MKL) comparison; there (we omit showing a plot for space reasons) RSB is slightly slower than MKL (by circa 10–20%). Comparing Fig. 15 to Fig. 11, one can see how on the average, the scalability of RSB's *SymSpMV* is slightly better than that of *SpMV*/*SpMV_T*.

The matrix on which the smallest advantage over MKL is found is *audikw_1* (only 12%, see Fig. 15) — as a consequence it is the only case with RSB *SymSpMV* scaling worse than MKL. Indeed, in most of the cases the single-threaded (not shown in the plots) RSB's *SymSpMV* is faster than MKL's, between 20 and 25%.

Just as in the unsymmetric case, CSB performance scales much better than RSB and MKL's CSR: up to 14×. The worst cases scale around 9×, and that is more than the best RSB case. This good scalability property is caused by the serial CSB *SymSpMV* performing significantly slower than MKL's CSR or RSB. Most of the RSB results when using 8 threads (omitted from the plots) are slower than with 16 threads (up to some 30% difference with *Flan_1565*). For matrices *europe_osm, nlpkkt120, nlpkkt160, nlpkkt200, nlpkkt240, rgg_n_2_23_s0, rgg_n_2_24_s0* RSB-16 performs more or less the same as RSB-8.

Results for MKL show a similar trend. It is worth to note that the three matrices with the highest index per nonzero ratio (*audikw_1, delaunay_n24, europe_osm*) don't exhibit any improvement between RSB-8 and RSB-16. In the case of CSB, the best results are for 16 threads. There is seemingly no performance correlation between the RSB and CSB formats here, although both seem to improve the performance of matrices as *delaunay_n24*, *europe_osm*, *gsm_106857*, *road_usa*, where MKL's CSR performing particularly slow.

4.4. Submatrices Lock Contention in practice

As mentioned in Section 2.4, the simultaneous update of different y intervals is kept free of accidental race conditions by the use of a custom row locking mechanism. This mechanism operates on ranges of the

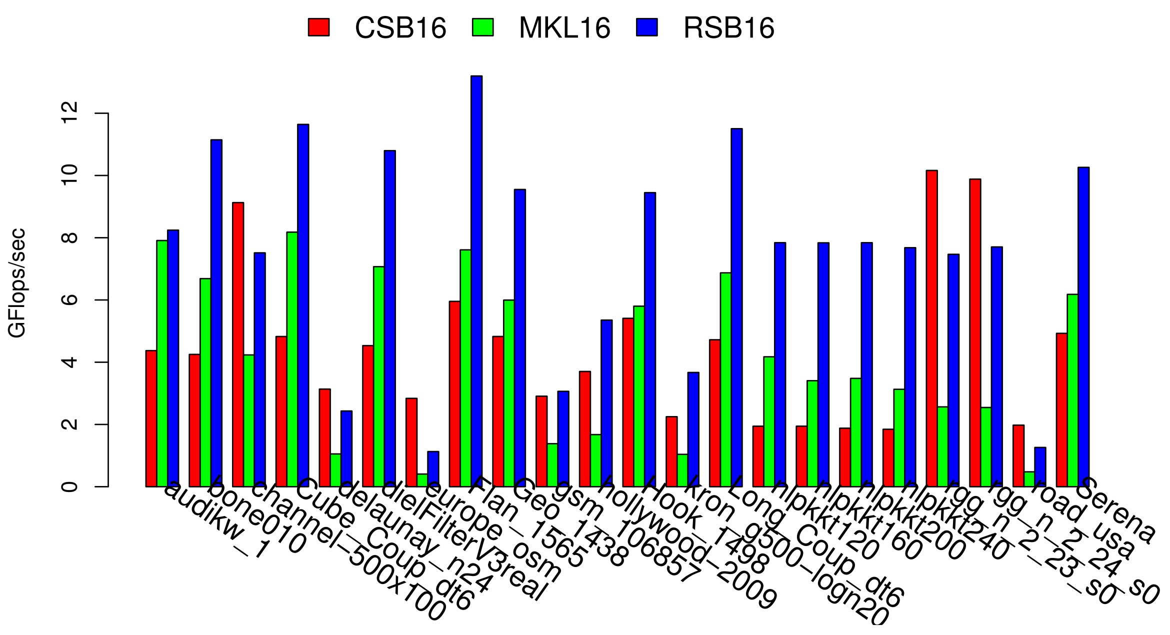

Figure 12: Symmetric matrices. *SymSpMV* performance, for 16 threaded CSB, MKL, and RSB.

x vector (recall Fig. 8). In the case of *SymSpMV*, two target y ranges (recall listings Fig. 6 and Fig. 7) are locked for each submatrix, thus increasing the likelihood of additional lock contention (the lock impedes other threads to use either of the two intervals).

To quantify the impact of the locking mechanism, we performed an experiment consisting in running the usual multiplication algorithm (Fig. 8) with a fictitious lock mechanism; i.e.: the checks at lines 10, 15, 21 were skipped. With the lock constraint relaxed, task parallelism is maximized, and so potentially execution speed. However, then the results are not guaranteed to be correct anymore, because floating point arithmetic instructions (employed by the y array update code) have no guarantee of executing atomically on the architecture we are using (see [19, V.3, Ch. 8.1]). In addition to the corrupt results, simultaneous concurrent writes to the same cache lines by different threads lead to the *false sharing* (see [26, 8.4.5]) problem, which induces an increased amount of expensive cache misses, thus degrading performance. Because of these two opposed effects, one can expect either the prevalence of an extreme (either degradation or speedup) or a certain mutual compensation, with no noticeable execution time differences.

In practice, out of the 22 symmetric matrices considered, one (*gsm_106857*) allowed *SymSpMV* to speed up almost 100%, followed by *audikw_1, relat9* with around 30%, and the rest below. Only matrix *kron_g500-logn20*) slowed down *SymSpMV* by some 70%. This suggests that: 1) the false sharing problem occurred rather seldom; 2) current matrix partitionings do not pose an excessive limit to parallelism — this is satisfactory, although the potential speedup for *gsm_106857* could be explored in the future.

In the group of the (seven) unsymmetric matrices we use, certain matrices (*arabic-2005, indochina-2004, kron_g500-logn20, GL7d19*) slowed down *SpMV* by some 20–50%; the remaining ones executed between 10 and 40% faster, somehow similarly to *SymSpMV*. A markedly different behavior can be seen in *SpMV_T*, because in no case a notable (more than a few percent) speedup was encountered, and in three cases, even a 40% slowdown was observed. At the COO/CSR level (recall Fig. 3, Fig. 5 and discussion in Section 2.3), *SpMV_T* exhibits a different pattern of memory accesses, namely updating y locations corresponding to column indices, which are often non consecutive. This makes *SpMV_T* of both COO/CSR (see Fig. 9 for the effect on *RSB_SpMV_T*) less efficient than *SpMV*, and occurrence of false sharing exacerbates the problem. Because of the limited scope and interest of this additional experiment, we omit plots.

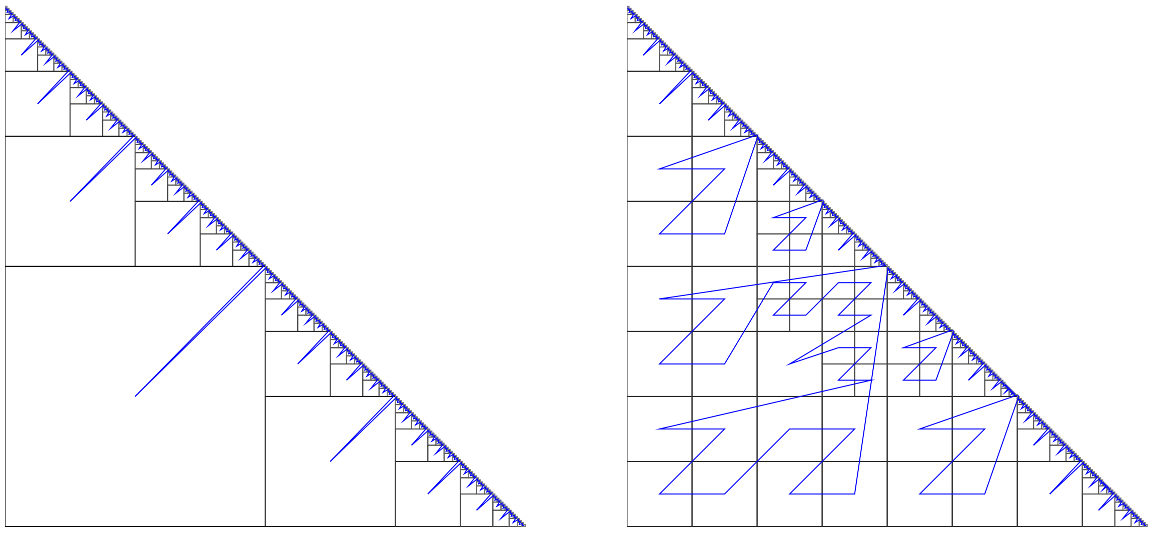
Figure 13: Symmetric matrices *channel-500x100-b050* (left) and *europe_osm* (right). While their partitionings look very similar to that of matrix *audikw_1* (see Fig. 1), which has 41.64 elements per row, they average respectively only 8.89 and 1.06 elements per row. Due to its regular structure and low index overhead, *channel-500x100-b050* performs slightly better than *audikw_1*, and almost three times faster than *europe_osm* (which has a regular structure, but almost twice the index overhead).

4.5. Cost of conversion from Row Ordered COO to RSB

In this section we consider the cost of obtaining instances of RSB data structures in practice. As introduced in Section 2.5, the conversion procedure (that we call *CooToRSB*) is made up of two phases: *SubdToRSB* and *ShufToRSB*. The first one is more exposed to latency, as it is intensive in binary searches and other operations leading to irregular memory accesses. The second one can suffer of bandwidth limitations, as it performs memory copy operations and linear array scans. Because of these differences, in addition to the whole conversion process, we measure the two components timings individually.

RSB has been developed mainly for iterative methods, so the metric of our choice is the ratio of conversion to matrix-vector multiply times (*SpMV* for unsymmetric matrices, *SymSpMV* for symmetric ones). Since, when optimizing a particular application the number of iterations — and thus the multiplications count — to solution is usually known and a conversion from COO is currently required in order to use RSB, it is convenient to quantify the additional overhead in terms of *equivalent matrix-vector multiplications*. Such information can be then used when deciding whether adopting RSB can bring overall speedup to an application. To relate scalings of the assembly algorithms to that of the multiply operations, we also display single threaded performance.

Results for unsymmetric matrices (Fig. 16) show that when using 16 threads, time spent in *CooToRSB* is equivalent to around 20 *SpMV*'s. When executing 1 threaded RSB, the relative cost is less, that is between 10 and 20 *SpMV*'s. This might suggest that *CooToRSB* scales less than *SpMV* does; this is only partially true, since (recall from Section 2.5) the amount of work of *CooToRSB* grows with the threads count, as additional subdivisions are needed. Indeed, the number of instantiated leaf submatrices increases

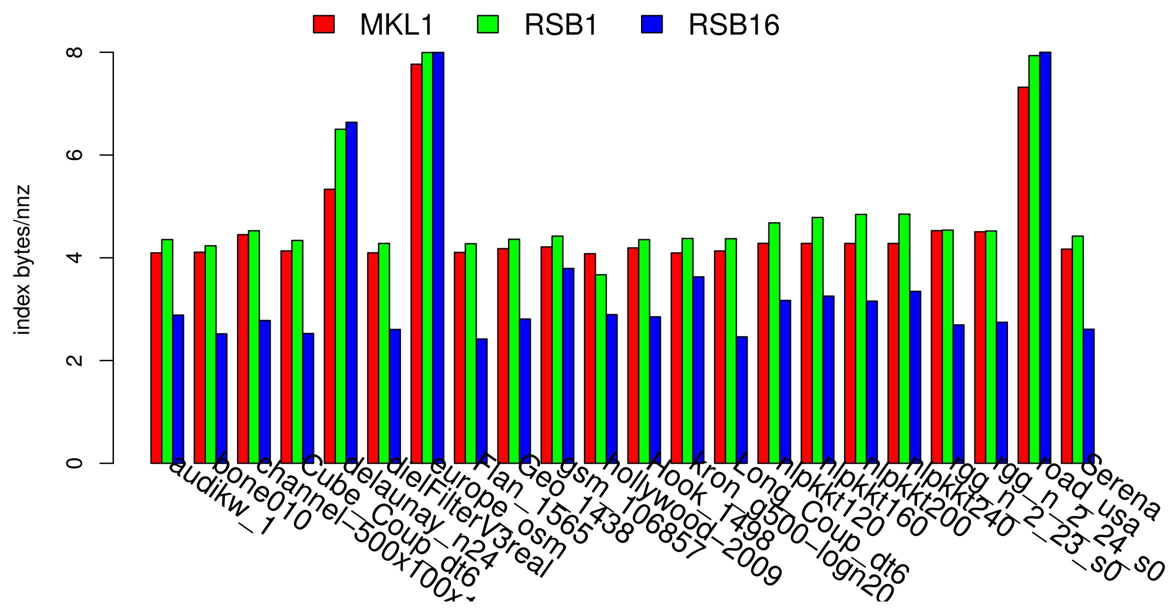

Figure 14: Symmetric matrices. Index bytes per nonzero for representing either 1 or 16 threaded RSB, as well as for CSR (independent from threads count) representations. We omit reporting values for CSB since all matrices used almost the same (4–4.18) index bytes/nnz.

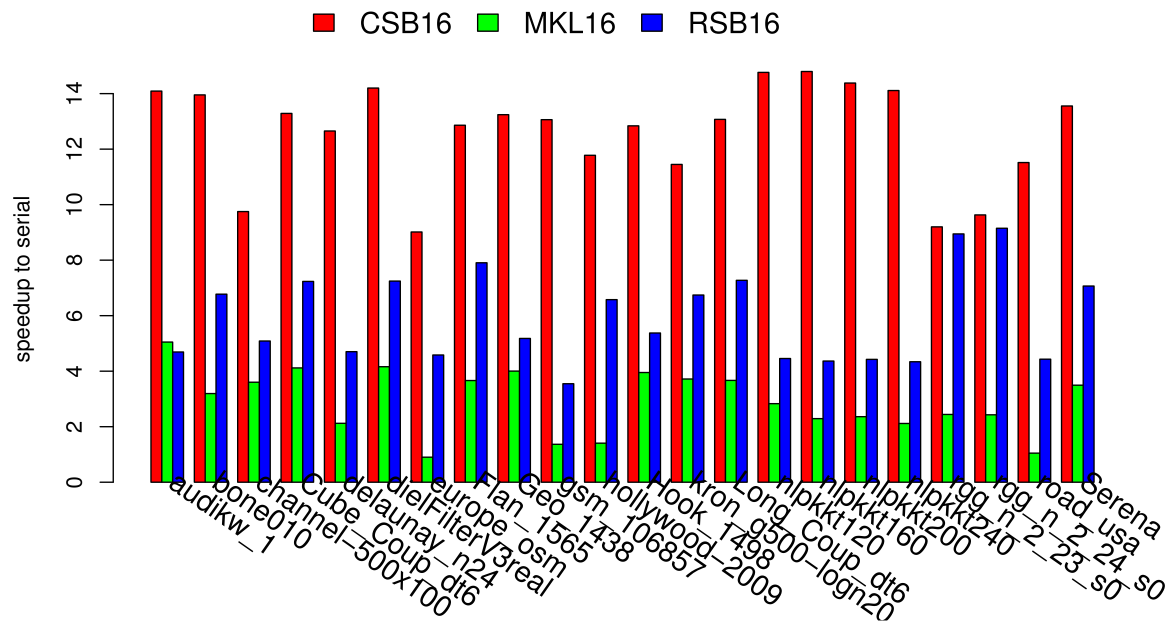

Figure 15: Symmetric matrices. *SymSpMV* parallel speedup (16 to 1 threads performance ratio).

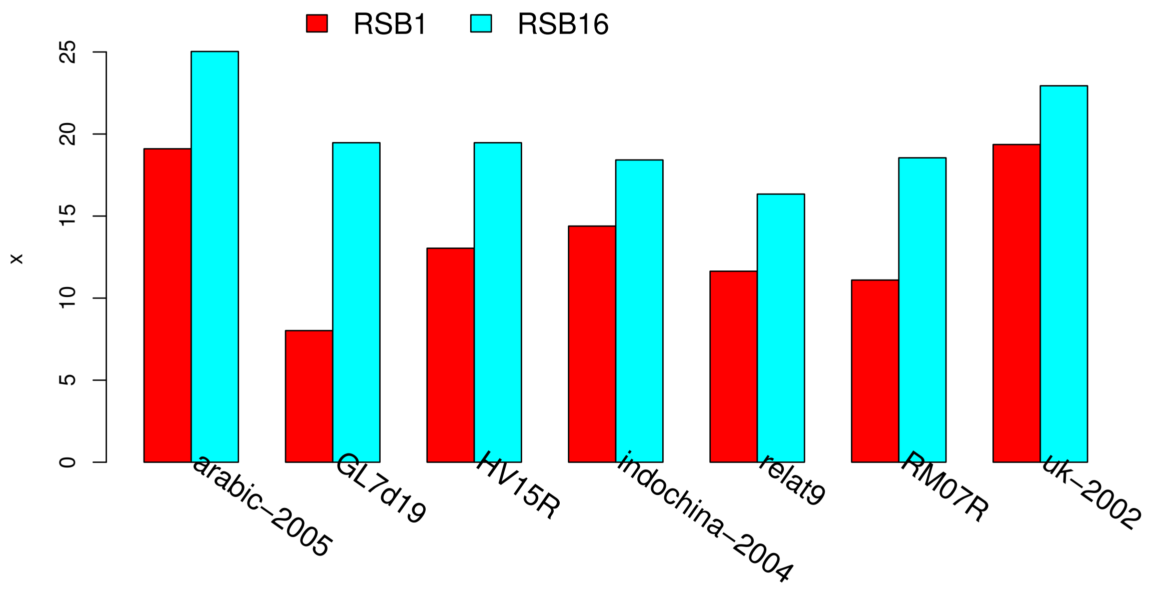

Figure 16: Unsymmetric matrices. Time ratio of *CooToRSB* to *SpMV* for 1 or 16 threads.

around tenfold for most matrices. This is reflected by the cost of *SubdToRSB* (Fig. 20) scaling worse than *ShufToRSB* (see Fig. 22). Indeed, *ShufToRSB* performs an almost constant amount of memory transfers and other mostly bandwidth bound operations. Since *SubdToRSB* scales less than *ShufToRSB* does, with more threads available it might end up dominating the *CooToRSB* cost. We consider its improvement as future research.

In the case of symmetric matrices, *CooToRSB* to *SymSpMV* (Fig. 17) ratios are similar to the ones for unsymmetric; that is mostly around 10 and 20 in the 16 threads case, and around half of that in the single threaded case. Some cases stand out: matrix *delaunay_n24* needs more than the time of 29 *SymSpMV*'s; *dielFilterV3real* around 26. Indeed for these matrices the *SubdToRSB* phase is particularly expensive (Fig. 21). *ShufToRSB* costs between 3 and 14 *SymSpMV*'s (Fig. 23), with much less variation than for *SubdToRSB*.

Since most of our results with *SpMV*, *SpMV_T*, *SymSpMV* gave an advantage over MKL's CSR, we can now compute how many (multiply) iterations are needed to amortize completely the cost of *CooToRSB* for an application and save overall execution time. Because of the moderate advantage in *SpMV* (see Fig. 18), depending on the matrix, from 19 to 155 iterations may be needed. If considering *SpMV_T*, the significant advantage of RSB over CSR enables execution times savings already after a few dozens of iterations. Similarly with symmetric matrices (Fig. 19): the number of *SymSpMV*'s necessary to justify adoption of RSB ranges from a few to a few hundred; however the vast majority needs only a few dozens of them.

These results indicate aptness of RSB as a replacement of MKL's CSR in applications meeting such requirements (that is, repeated *SymSpMV* or *SpMV_T*).

Unlike RSB, the CSB code is not distributed in form of a library, but rather in a form of a collection of prototype programs. The COO to CSB constructor is not explicitly timed in this prototype code; indeed, it's a serial procedure converting from CSC, and has not been written with benchmarking in mind. The CSB author confirmed this (by private communication), adding that the conversion process can be improved significantly. So for fairness and consistency reasons we chose to skip the inclusion of the to-CSB conversion process costs.

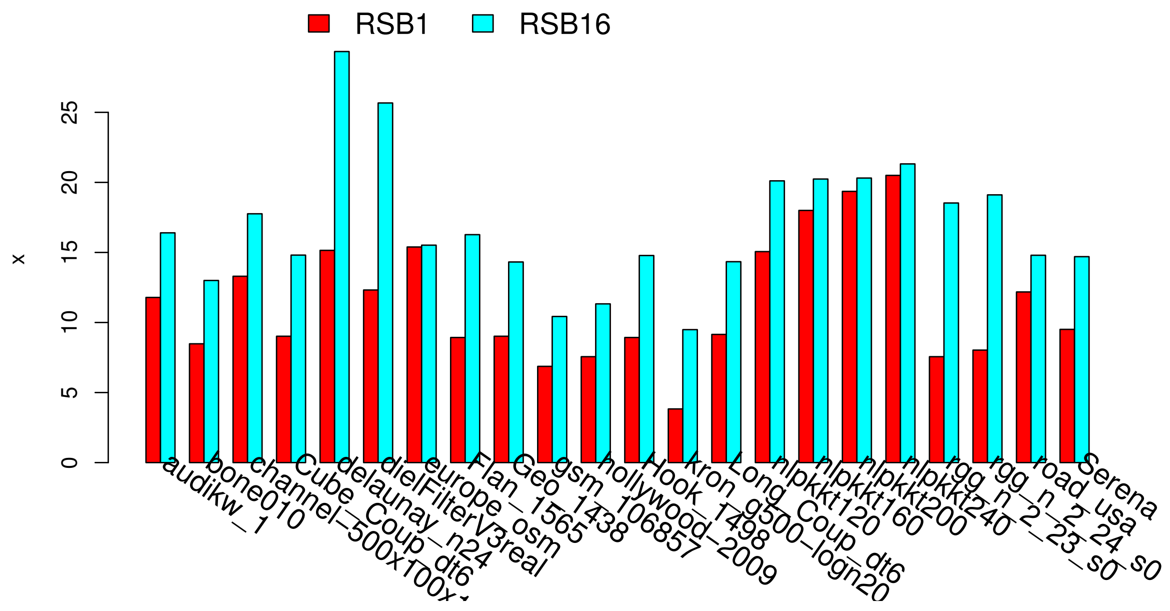

Figure 17: Symmetric matrices. Time ratio of *CooToRSB* to *SymSpMV* for 1 or 16 threads.

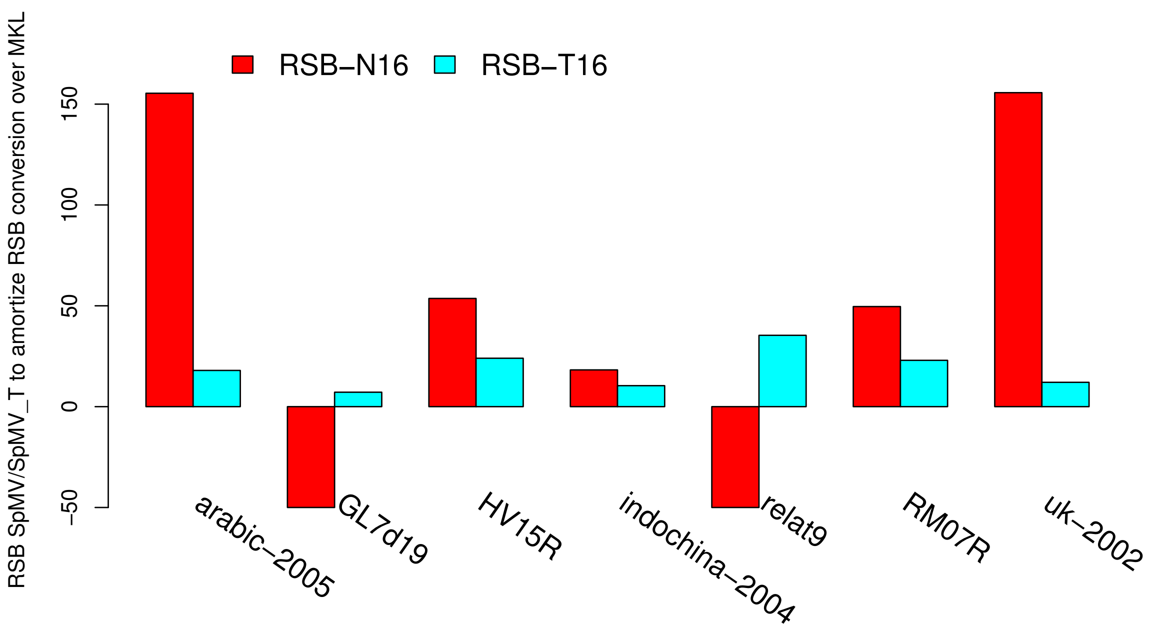

Figure 18: Unsymmetric matrices, 16 threads. Amount of *SpMV*/*SpMV_T* executions with RSB necessary to amortize time of *CooToRSB*, and get an advantage over MKL. Please note that RSB *SpMV* was slower on *GL7d19* and *relat9*; here we keep columns for them only for the sake of uniformity in the plots layout, and mark them with a fictitious negative value. Please note how much faster is RSB's *SpMV_T* than MKL's in amortizing conversion time.

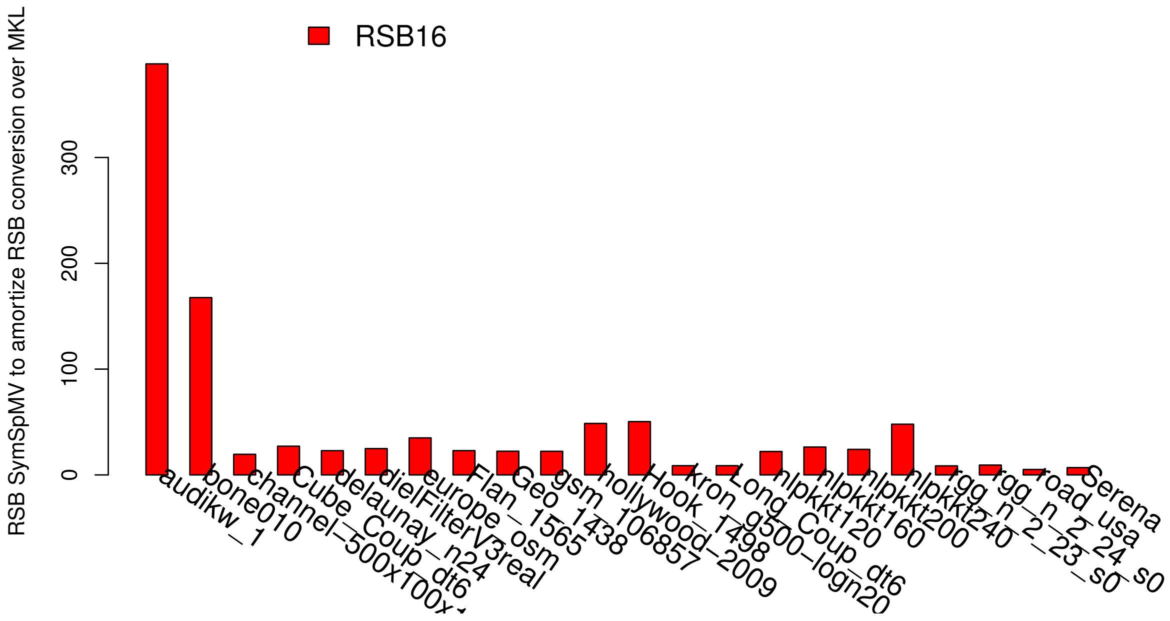

Figure 19: Symmetric matrices. Amount of *SymSpMV* executions with RSB necessary to amortize time of *CooToRSB*, and get advantage over MKL. 16 threads.

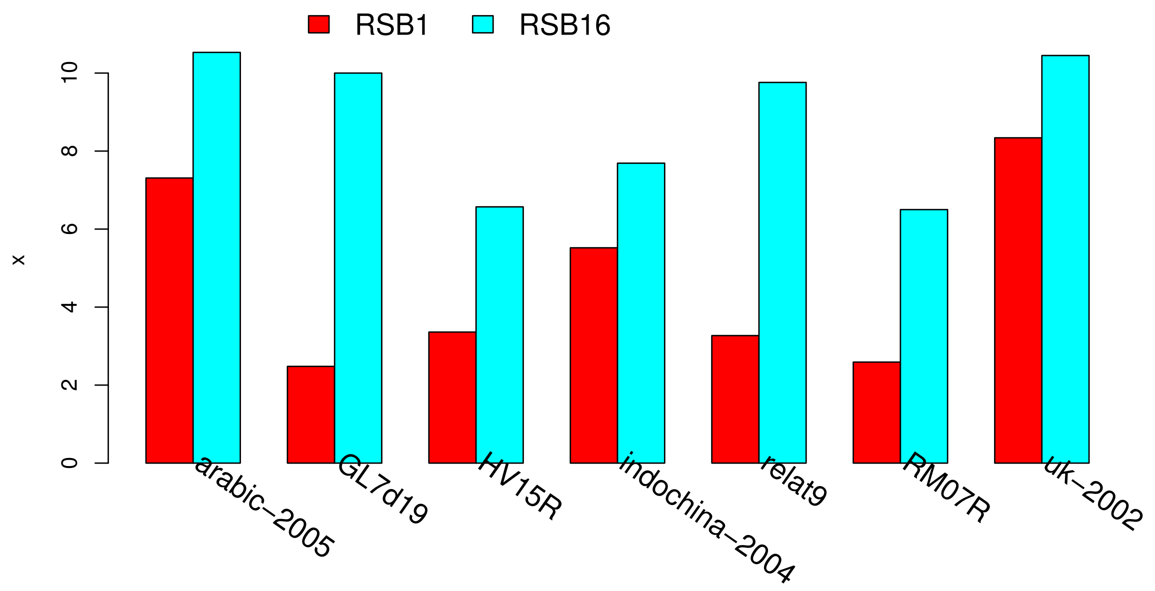

Figure 20: Unsymmetric matrices. *SubdToRSB* (matrix subdivision) to *SpMV* execution time ratio, 1 and 16 threads.

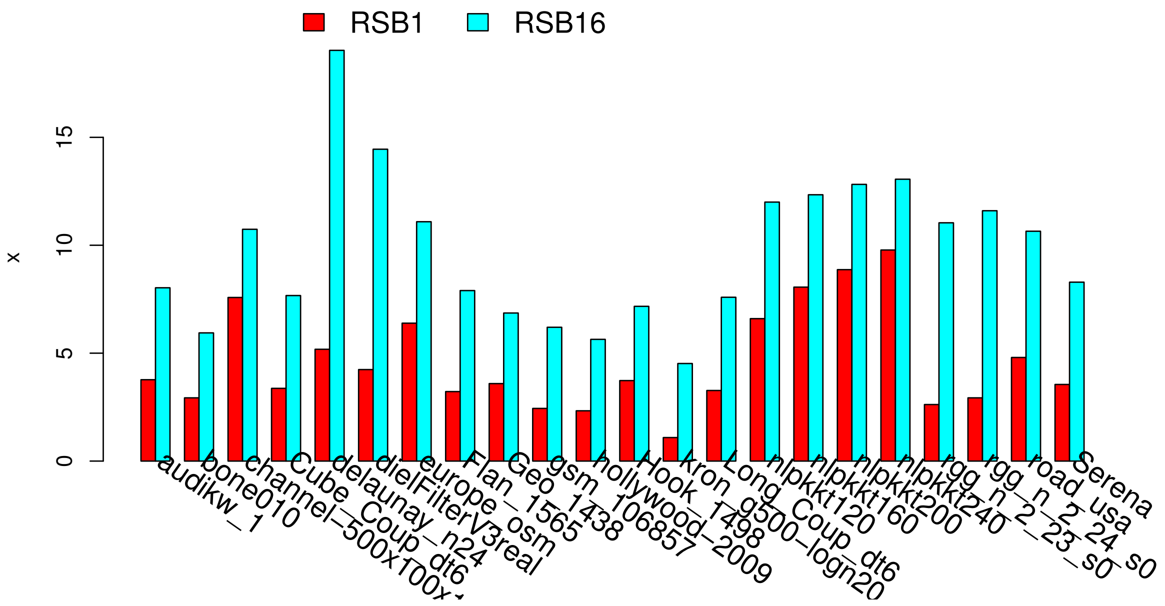

Figure 21: Symmetric matrices. *SubdToRSB* (matrix subdivision) to *SymSpMV* ratio, 1 and 16 threads.

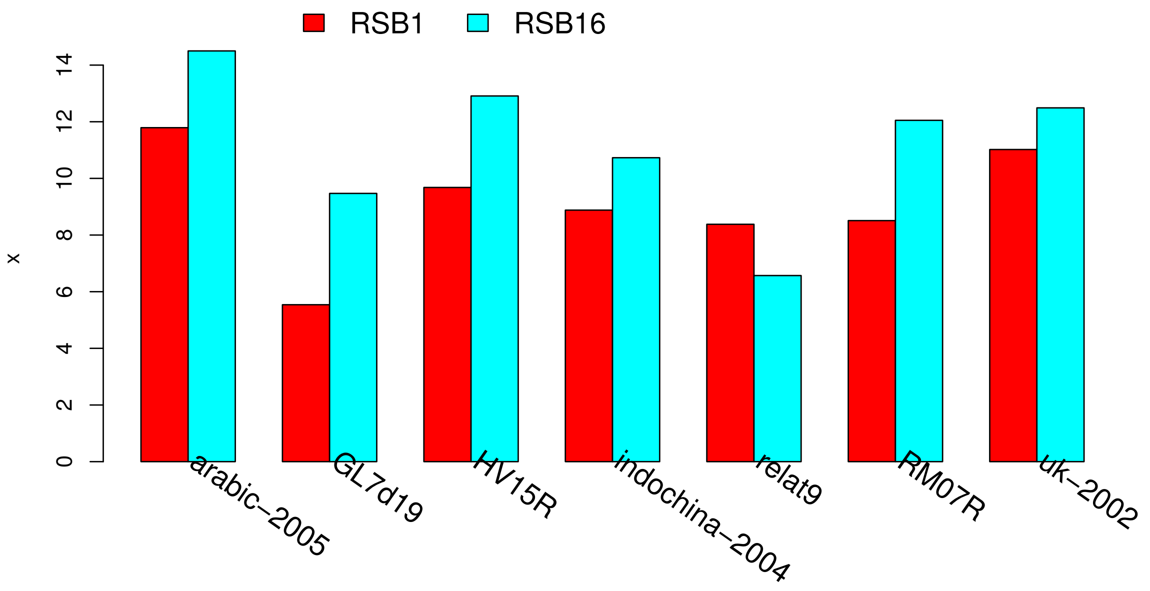

Figure 22: Unsymmetric matrices. *ShufToRSB* (input COO arrays shuffling) to *SpMV* execution time ratio for 1 and 16 threads.

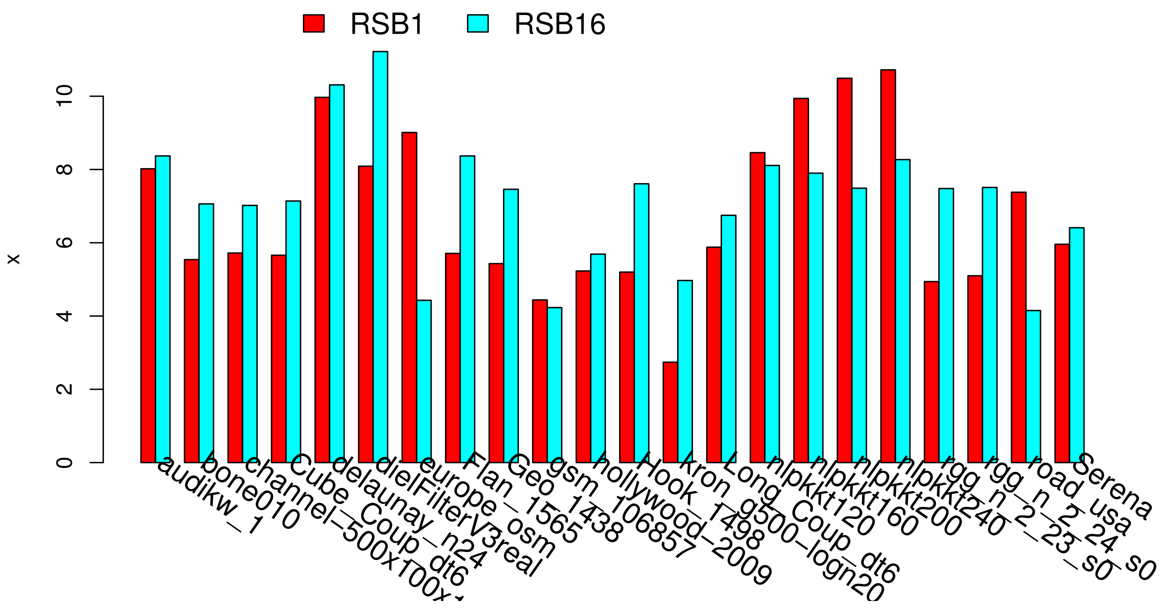

Figure 23: Symmetric matrices. *ShufToRSB* (input COO arrays shuffling) to *SymSpMV* execution time ratio for 1 and 16 threads.

5. Future Directions

There is a number of modifications that it may be worth investigating for improving RSB without impacting too much on its current design. Given RSB's hierarchical nature, potential optimizations may target either its parallel algorithms or the serial ones. In the following list we briefly discuss possible optimizations and their impact; some of these might be integrated with recent developments found in the literature.

- Given the non determinism of both the RSB instancing algorithm and the corresponding matrix-vector multiply algorithms, it may be desirable to refine a given matrix structure in a controlled way to fulfill some optimizing property. As an example, one may consider submatrices of an assembled RSB matrix for either aggregation or subdivision; in our experience, the former could lead into using less memory for indices, while the latter could increase parallelism, depending on the case. With a simple autotuning framework it would be possible to automatically explore the performance of different variants and retain the matrix instance yielding the most efficient results. Benefits of such a feature would probably be more limited than with efforts like the OSKI/pOSKI tuning framework with blocked formats (see Byun et al. [27]), but we think in our context they may still be significant.

- During the course of a multiplication with RSB, each submatrix should be visited exactly once for performing a local *SpMV*/*SpMV_T*/*SymSpMV* and updating the result vector (see Fig. 8) in either one or two intervals. We are using a *busy wait* technique in the submatrices locking mechanism. This is a cause for potential *false sharing* over the shared lock structure, and may be overcome by using alternative locking mechanisms. To this purpose, we may employ the `task` (see Ayguade et al. [28]) or other OpenMP constructs ([20]).

- Improvement of the scaling properties of the *SubdToRSB* portion of the COO to RSB assembly procedure (see Section 4.5) shall be addressed. As indicated in [21], this should be considered carefully because of the possible consequences for the partitioning quality with respect to multiplication performance.

- Belgin et al. ([11]) propose *pattern based representations* (PBR) targeted at matrices exhibiting non-contiguous nonzero patterns. Provided with apt matrices, RSB would probably benefit from such an approach while still retaining its cache blocking properties. However, an efficient implementation of PBR (according to its authors in [11]) should rely on machine specific *intrinsics*, and as such is of limited portability.

- Pichel et al. [29] experimented with recent NUMA (Non Uniform Memory Access) processors and different memory affinity options, obtaining an impact exceeding 30% in some cases. A memory affinity oriented approach applied to RSB would first require different submatrices to be stored in different arrays, allocated by different threads; then, a thread-to-submatrix mapping function should exist, and thus each thread would only operate on affine memory banks, thus avoiding the penalty in accessing *non-local memory.* Such an implementation of a memory affinity policy would be not practical with the current form of RSB. In the first place it would be not trivial to obtain an optimal thread-to-submatrix mapping — the execution order is currently determined only at runtime. A cheap way to compute an approximately good submatrix-to-thread mapping would be running one multiplication first, and annotating the visit ordering of submatrices (recall Fig. 8). Then the matrix tree structure would be rebuilt, this time having each submatrix arrays allocated with NUMA awareness on a different thread, according to the first execution annotation. Now on, multiplications on the given matrix would follow strictly the order defined by the original annotation, thus also leading to much reduced locking requirements. Such an approach, known as *partial execution* would certainly complicate usage scenarios. A second major reason discouraging from submatrix-based NUMA awareness is usability: in the current design, the RSB submatrices can be stored in the shuffled original input COO arrays. Clearly, a NUMA aware storage could not support such use case.

- *Zig-Zag CSR* (ZZ-CSR): that is, reversing each second CSR line representation, in both the column index and coefficients values arrays, thus increasing the chance of reuse of the right hand side vector, at least in the first and last columns of each row. This approach was suggested by Yzelman and Bisseling in [30, Sec. 5], and could be transparently applied to CSR and COO submatrices: these are processed serially and independently of each other.

- In [31, Table 3], Guo and Gropp introduced a *stream unrolling* optimization for the CSR kernels, in which more than one *sparse row* gets processed at a given time. This technique allowed the authors to exploit the multiple *memory streams* available on the machines they considered, and increase reuse of the right hand side vector, if cached. Each Sandy Bridge CPU features a hardware *prefetcher* mechanism capable of up to 32 simultaneous streams, either ascending or descending (in its microarchitecture jargon, "Streamer" — see [26, 2.1.5]). Considering that during *CSR_SpMV* arrays PA, y are accessed sequentially and once per row (see Section 2.3), arrays JA, VA sequentially and once per nonzero, and x irregularly but once per nonzero, four streams may be probably identified with success by the hardware, while for x this would not be possible in the general case. The situation would be similar in the transposed and COO cases. The symmetric case would have additional two series of accesses (one sequential read per row, one random write per nonzero), so for a total of five identifiable streams.

  We observe that using all 8 cores of a Sandy Bridge processor for running RSB's COO and CSR kernels, all the 32 prefetch streams are likely to be used. For this reason it is unclear whether increasing the number of streams per thread would be beneficial in our context, at least when using all available cores.

Finally, no machine specific tuning has been applied so far; therefore such optimizations could be investigated in forthcoming work, especially to address specific problem instances.

6. Conclusions

In this work we continued performance analysis of our hierarchical sparse matrix format (RSB) matrix-vector implementation in a real world scenario: we compared with Intel Math Kernel Library (MKL)'s `mkl_dcsrmv` routine for CSR matrices, on a set of 29 large sparse matrices from real world applications, either symmetric or unsymmetric. Results were very encouraging: RSB was able to deliver over twice the performance in *SymSpMV* and *SpMV_T*, and up to twice the performance in *SpMV*. Moreover, the transposed product of unsymmetric matrices scaled nearly as the untransposed. Of the considered matrices, only two (both not square, and with a high index bytes/nonzero ratio) did not outperform MKL in *SpMV*. In all cases, *SpMV_T* and *SymSpMV* outperformed MKL. According to our initial goals of maximum generality and leaving room for further optimization, we did not employ any non portable optimization technique (e.g.: assembly code, intrinsics, library, specific programming construct or language). The better efficiency of RSB over `mkl_dcsrmv` seems to be structural — the reorganization of a matrix in smaller sparse blocks (submatrices) is likely to increase cache reuse within each block; the obtained coarse level parallelism impacts especially favorably on *SpMV_T* and *SymSpMV*.

Since RSB is not a standard format, our analysis took also in consideration the time for assembling it from row ordered COO. We observe that for most of our symmetric matrices, assembly time can be amortized by the time saved with already a couple of dozens of (*SymSpMV*) multiplications. For the unsymmetric matrices considered, were necessary from a couple of dozens to a few hundred *SpMV*s. However, *SpMV_T* speedup allowed to amortize RSB conversion costs already with a few dozens of executions.

In our *SpMV*/*SymSpMV* performance results comparison, we also took in consideration the research format CSB. In *SpMV*, we found CSB to be slightly better than RSB, whereas RSB beats CSB in most symmetric cases (*SymSpMV*) by being twice as fast or even more; in *SpMV_T*, RSB prevails by a lesser amount.

These results suggest predisposition of the RSB format to iterative methods which are intensive in either symmetric matrices multiplication, or the transposed matrix-vector multiply operation.

7. Acknowledgements

The author wishes to thank (here in alphabetic order) colleagues Nitya Hariharan, Roman Hatzky, Tiago Ribeiro for having proofread and commented the manuscript, providing criticism, good advice and experience.

References

[1] Y. Saad, Iterative Methods for Sparse Linear Systems, 2nd edition, SIAM, Philadelphia, PA, 2003.

[2] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, H. V. der Vorst, Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition, SIAM, Philadelphia, PA, 1994.

[3] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, H. van der Vorst (Eds.), Templates for the solution of Algebraic Eigenvalue Problems: A Practical Guide, SIAM, 2000.

[4] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, K. A. Yelick, The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report No. UCB/EECS-2006-183, Technical Report, Electrical Engineering and Computer Sciences, University of California at Berkeley, 2006.

[5] G. Schubert, G. Hager, H. Fehske, Performance Limitations for Sparse Matrix-Vector Multiplications on Current Multi-Core Environments, in: S. Wagner, M. Steinmetz, A. Bode, M. M. Müller (Eds.), High Performance Computing in Science and Engineering, Garching/Munich 2009, Springer Berlin Heidelberg, 2010, pp. 13–26.

[6] A. Buluç, S. Williams, L. Oliker, J. Demmel, Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication, Parallel and Distributed Processing Symposium, International (2011) 721–733.

[7] M. Martone, S. Filippone, S. Tucci, M. Paprzycki, M. Ganzha, Utilizing recursive storage in sparse matrix-vector multiplication - preliminary considerations, in: T. Philips (Ed.), CATA, ISCA, 2010, pp. 300–305.

[8] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, C. E. Leiserson, Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks, in: F. M. auf der Heide, M. A. Bender (Eds.), SPAA, ACM, 2009, pp. 233–244.

[9] A. Yzelman, R. H. Bisseling, Two-dimensional cache-oblivious sparse matrix–vector multiplication, Parallel Computing 37 (2011) 806–819.

[10] K. Kourtis, V. Karakasis, G. Goumas, N. Koziris, CSX: an extended compression format for spmv on shared memory systems, in: Proceedings of the 16th ACM symposium on Principles and practice of parallel programming, PPoPP '11, ACM, New York, NY, USA, 2011, pp. 247–256.

[11] M. Belgin, G. Back, C. J. Ribbens, Pattern-based sparse matrix representation for memory-efficient SMVM kernels, in: Proceedings of the 23rd international conference on Supercomputing, ICS '09, ACM, New York, NY, USA, 2009, pp. 100–109.

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

[13] I. Simecek, Sparse matrix computations using the quadtree storage format, in: Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2009 11th International Symposium on, pp. 168–173.

[14] D. Langr, I. Simecek, P. Tvrdik, T. Dytrych, J. P. Draayer, Adaptive-blocking hierarchical storage format for sparse matrices, in: Computer Science and Information Systems (FedCSIS), 2012 Federated Conference on, pp. 545–551.

[15] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, J. Demmel, Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms, in: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, ACM New York, NY, USA.

[16] I. S. Duff, M. A. Heroux, R. Pozo, An overview of the sparse basic linear algebra subprograms: The new standard from the BLAS Technical Forum, ACM Trans. Math. Softw. 28 (2002) 239–267.

[17] M. Martone, S. Filippone, M. Paprzycki, S. Tucci, On the usage of 16 bit indices in recursively stored sparse matrices, in: Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2010 12th International Symposium on, pp. 57–64.

[18] G. M. Morton, A computer oriented geodetic data base and a new technique in file sequencing, Technical Report, IBM Ltd., Ottawa, Canada, 1966.

[19] Intel Corporation, Intel® 64 and IA-32 Architectures Software Developer's Manual (Combined Volumes 1, 2A, 2B, 3A, 3B, an 3C) (Order Number: 325462-044US), 2012.

[20] OpenMP Architecture Review Board, OpenMP Application Program Interface, Version 3.1, 2011.

[21] M. Martone, S. Filippone, S. Tucci, M. Paprzycki, Assembling recursively stored sparse matrices, in: Computer Science and Information Technology (IMCSIT), Proceedings of the 2010 International Multiconference on, pp. 317–325.

[22] Intel Corporation, Intel® Math Kernel Library, Reference Manual (Document Number: 630813-051US), 2012.

[23] ISO/IEC 9899 - Programming languages - C (standard), 1999.

[24] T. A. Davis, Y. Hu, The University of Florida sparse matrix collection, ACM Trans. Math. Softw. 38 (2011) 1:1–1:25.

[25] Standard for information technology— portable operating system interface (POSIX) (IEEE std 1003.1), 2008.

[26] Intel Corporation, Intel® 64 and IA-32 Architectures Optimization Reference Manual (Order Number: 248966-026), 2012.

[27] J.-H. Byun, R. Lin, J. W. Demmel, K. A. Yelick, pOSKI: Parallel Optimized Sparse Kernel Interface Library User's Guide for Version 1.0.0, 2012.

[28] E. Ayguade, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, G. Zhang, The design of OpenMP tasks design, Parallel and Distributed Systems, IEEE Transactions on 20 (2009) 404–418.

[29] J. C. Pichel, J. A. Lorenzo, D. B. Heras, J. C. Cabaleiro, T. F. Pena, Analyzing the execution of sparse matrix-vector product on the Finisterrae SMP-NUMA system, Journal of Supercomputing (2011) 195–205.

[30] A. N. Yzelman, R. H. Bisseling, Cache-oblivious sparse matrix–vector multiplication by using sparse matrix partitioning methods, SIAM Journal on Scientific Computing 31 (2009) 3128–3154.

[31] D. Guo, W. Gropp, Optimizing Sparse Data Structures for Matrix-vector Multiply, International Journal of High Performance Computing Applications 25 (2011) 115–131.