On BLAS operations with recursively stored sparse matrices

Michele Martone*, Salvatore Filippone*, Marcin Paprzycki†, and Salvatore Tucci*
* University of Rome “Tor Vergata”
Via del Politecnico 1, 00133 Rome, Italy
Email: {michele.martone,salvatore.filippone,tucci}@uniroma2.it
† Systems Research Institute, Polish Academy of Sciences
ul. Newelska 6, 01-447 Warsaw, Poland
Email: marcin.paprzycki@ibspan.waw.pl

Abstract—Recently, we have proposed a recursive partitioning based layout for multi-core computations on sparse matrices. Based on positive results of our initial experiments with matrix-vector multiplication, we discuss how this storage format can be utilized across a range of BLAS-style matrix operations.

I. INTRODUCTION

It is well-known that utilization of recursive storage is advantageous when implementing dense matrix operations ([1]). Following this idea, we have recently proposed the utilization of *recursively subdivided* compressed sparse row (CSR) format in sparse matrix operations (calling it RCSR; [2]). There, we have verified experimentally that the single and dual threaded performance of the SpMV operation is competitive with that obtained when using the compressed sparse blocks-based sparse matrix representation (CSB format; [3]). This initial success prompted us to extend our work to other BLAS-like operations, and to adapt to architectures with more cores. In this paper, we discuss how the RCSR represented matrices can be used in the Sparse Triangular Solve (SpSV) and the Sparse Matrix-Vector Multiplication (SpMV) (for both symmetric and unsymmetric matrices) performed on “truly multi-core” architectures. The SpMV (known as USMV in BLAS jargon [4]) is the main computational kernel of iterative methods for linear systems and eigenvalue problems, while the SpSV (also known as USSV) is utilized in preconditioning of iterative methods and in the direct solution of linear systems using factorizations [5], [6], [7]. The literature indicates that the study of these two operations has somewhat diverged, in that attempts at optimizing storage and/or performance are made separately; typical examples would be [8] for the SpMV, and [9] for the SpSV. Our work is an attempt to revisit the basic ideas underlying the development of BLAS kernels to propose a unified approach to matrix representation leading to good performance on both “kinds” of sparse matrix operations. Even though it is clear that, almost certainly, there is no “silver bullet”, our results in [2] indicate that our approach is competitive at least for some real-world matrix classes. We start in section II, with a brief introduction to the RCSR format, following with a short description of multi-core SpSV and SpMV (symmetric and unsymmetric) operations. Next, we describe our experimental setup and present and discuss the performance results we obtained while running our codes. We conclude the paper by outlining directions for future investigation.

II. THE RECURSIVE STORAGE FORMAT

Let us start by briefly describing our approach to obtaining a balanced recursive partitioning of the elements of a sparse matrix (more details can be found in [2]). We break down a matrix recursively, obtaining a *quad tree* of submatrices with *leaf submatrices* in the CSR format and intermediate submatrices as “helper indexing” structures. Recursive subdivision terminates following a predefined criteria relating the matrix “size” (in terms of elements and width/height), and properties of the machine the code is to run on (*its cache size cs*). Figure 1 depicts recursive decomposition of the L factor (obtained using SuperLU [10]) of the g7jac180 matrix, on different machines. The blue line is the order (which we call a *balanced Z-sort, or Zb*) that the elements of the matrix are stored in.

![Recursive subdivisions of L factors of matrix g7jac180](image)

**Fig. 1.** Recursive subdivisions of L factors of matrix g7jac180 (available from [11]) instantiated on machine M2 (left) and on the same machine, if it had half the outermost cache size (right). Only leaf matrices are shown, with a line joining them.

III. TRIANGULAR SOLVE AND MATRIX-VECTOR MULTIPLY

A key concept in our recursive storage is that each submatrix at the leaf level constitutes a “reasonable” unit of work (at least for the most common computations). The same applies...
Multithreaded SpMV for leaf matrices of a RCSR matrix.

1. $S \leftarrow [s_0, s_1, \ldots, s_{N-1}]$/*an array of terminal submatrices, in any order*/
2. $B \leftarrow [0, 0, \ldots, 0]$/*a zero bit for each submatrix*/
3. $n \leftarrow 0$/*count of visited submatrices so far*/
4. while $n < N$ do
5. begin parallel
6. $s \leftarrow$ pick a non visited submatrix $s$ from $S$ /*(say $s \leftarrow S[i], B[i] = 0$)*/
7. $[f, l] \leftarrow s$.roff, s.roff+s.rows
8. if locked($[f \ldots l]$) then cycle
9. lock($[f \ldots l]$)/*we lock y on s’s rows interval*/
10. /*perform SpMV on s against $y[x.s.coff:s.coff+s.columns]$ into $y[f : l]$*/
11. $y[f : l] \leftarrow y[f : l] + s \cdot x[s.coff:s.coff+s.columns]$  
12. $B[i] \leftarrow 1; n \leftarrow n + 1$
13. end parallel
14. end parallel

This, however, requires a specialized symmetric CSR SpMV code. Moreover, since the symmetric kernel performs both the SpMV on $s$ and on $s^T - D$ ($s$ transposed, minus the diagonal), it updates two intervals of the destination vector $y$; hence both intervals have to be locked, impacting the achievable parallel performance.

For this reason, at the cost of subdividing symmetric matrices more, we could gain some parallelism back from them; but since a detailed analysis of such a trade-off is beyond the scope of this document, we leave this as a topic for future investigations.

Note that the SpMV algorithm does not specify any particular order in visiting the leaf matrices; threads are free to cycle among submatrices repeatedly looking for “available” submatrices. In practice, this is not a big waste of resources: for each leaf matrix, we allocate a single bit in the bitmap $B$, and a pointer (possibly with offset and dimension indices) in $S$. Since each leaf matrix is likely to occupy $O(cs)$ ($cs$ being the outermost cache size) bytes, the memory traffic associated in looking, in $B$, for submatrices that are “available” is negligible; in most cases the bitmap will fit in the first level cache, and scanning repeatedly will not stress the memory hierarchy. Repeated scans of $S$ might cause overhead; however, data concerning matrix $s$ at index $i$ in $S$ is only needed in the case when $B[i] = 0$, and this access has a high hit probability (since a lock on the interested output vector intervals is the only constraint on the usage of that submatrix). A possible resource-wasteful situation would be a repeated lock contention on behalf of a single thread, when rows lock is not available; this situation would lead to the overuse of cache snooping circuitry among cores/CPU. This is not expected to be a problem (on current architectures, employing variants of the MESI cache coherence protocol) since this situation would imply that other threads are busy performing the SpMV, and thus likely not to have any shared variable cached. Obviously, this situation occurs when the last submatrices are visited, and there is no more actual work to be available; thus, it can be detected comparing the $n$ counter with $N$ and the available work items.

It should be stressed that the proposed approach will also work, with minimal modification, for the transposed case (employed in iterative methods such as BiCG, CGS, QMR [5]), whereas with a CSR representation, a parallel transposed SpMV would be challenging. With our approach, (or generally, with any coarsely blocked format; see [3, s.1] for a brief discussion) this task is much easier.

B. Triangular Solve

Let us look at the recursive breakdown of lower triangular solve ($x \leftarrow L^{-1} \cdot x$):

$$
\begin{bmatrix}
x_1 \\
x_2
\end{bmatrix}
\leftarrow
\begin{bmatrix}
\frac{L_1}{M} & 0 \\
L_2 & L_2^{-1}
\end{bmatrix}^{-1}
\begin{bmatrix}
x_1 \\
x_2
\end{bmatrix}
= L_2^{-1} \frac{L_1}{M} L_1^{-1} x_1 
$$

Decomposition 1 is quite straightforward, but without any further structure, it offers limited support for parallelism, as...
Fig. 3. Multithreaded Lower Triangular Solve for an RCSR Matrix

begin parallel
s ← pick a non visited submatrix s from S (say s ← S[i], B[i] = 0)
[f, l] ← [s. roff , s. roff + s. rows]
/*pick another submatrix if this row interval is locked*/
if locked(f . . . l) then cycle
if s. roff = s. coff then
if B[D[i]] = 1 & . . . & B[i - 1] = 1 then
lock(f . . . l) /*s is a diagonal block; we lock x on its rows*/
/*perform SpSV on s*/
x[f : l] ← s-1 x[f : l]
B[i] ← 1; n ← n + 1
unlock(f . . . l)
else cycle /*pick another submatrix */
end
if B[D[i]] = 1 then
lock(f . . . l) /*s is not a diagonal block; we lock x on its rows*/
/*perform SpMV on s*/
x[f : l] ← x[f : l] + s . x[s. coff : s. coff + s. columns]
B[i] ← 1; n ← n + 1
unlock(f . . . l)
else cycle /*pick another submatrix */
end
end parallel
end

this would be only possible in the SpMV operation in the “x2 ← L-1(x2 - M(L-1 x1)).” This dependency requires that the SpSV computation on the diagonal blocks can only be performed after all of its left blocks have been visited by the SpMV computation. Thus, our SpSV algorithm also operates on leaf matrices only, ignoring the intermediate matrices of the recursive structure. Note that we make explicit use of the fact that the recursive partitioning results in square diagonal blocks.

Listing 3 outlines the SpSV algorithm which operates on the leaf matrices of a recursively partitioned matrix; this algorithm can be applied for any matrix partitioning with disjoint submatrices which are square on the main diagonal. Observe that, in this algorithm, we sort leaf submatrices in a way that allows to perform the SpSV without any sophisticated data structures.

As the listing shows, each matrix not on the diagonal is involved once in the SpMV kernel. On the other hand, the SpSV kernels are executed on the diagonal submatrices only. Since the core update in an in-place lower triangular solve on a matrix L and vector x is x[i] ← (x[i] - \sum_{j=1}^{i-1} x[j] L_{ij}) / L_{ii}, there is a horizontal dependency (SpMV), which must be satisfied before performing SpSV on the diagonal blocks. Formally, the sorting criteria for a pair (s, s’) of submatrices follows the total order defined as: (i) if any one of the matrices (say, s) lies on the diagonal (we remind that due to recursive subdivision, a matrix intersecting the diagonal is necessarily square), then it comes after s’ only if the last row of s’ is less than or equal to the last row of s; (ii) if neither of s, s’ is on the diagonal, the one with the smaller last column index comes first (exception: in the case of same last row, the one with smaller first row). With this ordering, if the matrix s1 is on the diagonal, it can be visited only after all of the matrices s_j with j < i were visited. In particular, i if s1 and s2 are both on the diagonal, j < i, and the last row of s_j comes immediately before the first row of s_i, we say that s_j is a dependency of s_i. As far as the implementation is concerned, we put dependency information in a temporary, shared vector D, which we compute by scanning S. The actual operation is in a way similar to the hybrid parallel triangular solve by block anti-diagonals and block columns, proposed by Mayer in [6]. The main difference is that our approach is “implicit,” as threads run through “available” matrices and “parallel zones” are unlocked only after portions of the solution are solved.

IV. EXPERIMENTAL SETUP AND METHODOLOGY

To illustrate the efficiency of the proposed approach, we report performance results for the SpMV and the SpSV operations, respectively defined as y ← Ax and x ← L-1 x. As representative samples for the SpSV we have selected the lower triangles (L matrices) originating from the LU factorizations of a mix of the matrices used by authors of [6] and [9]. These matrices are publicly available at [11]. Their LU factorizations were computed using SuperLU 3.1 [10] after reordering with COLAMD [12], called indirectly by Octave 3.0.3 [13]. To test the performance of the SpMV, we used (1) the same L matrices as for the SpSV, (2) (unsymmetric) matrices that have been utilized in experiments reported in [2], but run on different machines and/or with larger number of cores, and (3) nine symmetric matrices. In the following we report data for the most significant subset of cases. We have also utilized the CSB prototype code ([3]), but applied it only to the group of unsymmetric, non-triangular matrices, as this code does not support symmetric ones. The performance measurements we report are the best ones, after performing 100 runs (for either code). However, we found no big variability in the performance, which indicates that even though the proposed algorithms are non-deterministic, their performance is stable.

We canonically count two floating point operations for each nonzero for the SpMV for general matrices or the SpSV, and four floating point operations for each nonzero for the SpMV for the symmetric matrices. We employ double as our floating point type, 64 bit pointers, and 32 bit integer indices.

Table IV contains informations about the CPUs of the machines on which we ran our experiments, and the C compilers we used for our code. We used the -03 -g64 -bmaxdata:0x1000000000 -qarch=pwr5 -gtune=pwr5 -qsmp=omp -qlanglvl=extc99 -qkeyword=restrict compilation flags on M1, and -03
We have compiled the CSB code using the required specialized CILK++ compiler, based on 64 bit GCC-4.2.4, build 8503. We modified this code according to the guidelines found in [3] to correctly provide it with machine cache parameters (the L2 cache size and the cache line length). We were not able to collect results of the CSB code on the M1 machine, as its architecture is not supported by CILK++. Note that the M2 machine is a lightly loaded network server, so its results may include some noise.

We are aware of high level approaches to multi-threading like Intel’s Thread Building Blocks ([14]) or the aforementioned CILK++ ([15]), but these approaches would force us to use C++ and restrict ourselves to the Intel architecture, so was not general enough for us. So we have chosen to implement our algorithms in C, using OpenMP for the parallelization, for reasons of availability, standardization, and compatibility of such approach.

V. RESULTS AND ANALYSIS

Results obtained in our experiments are summarized in figures 4,5,6,7 for machine M1; figures 12,13,14,15 for machine M3; and figures 8,9,10,11 for machine M2.

In all cases we report the performance in MFlops for a specific matrix, algorithm and number of utilized cores. As seen in section III-B, the SpSV algorithm we propose is based on the use of SpSV and SpMV kernels for the CSR; hence an efficient SpMV is needed in SpSV. During our experimental runs, we also collected single-threaded performance of the plain CSR SpSV and SpMV on the triangular matrices. We found that: i) CSR SpSV performance was slightly higher (by no more than 5%) than that of CSR SpMV; ii) CSR SpMV usually (but not always) outperformed the single-threaded RCSR by a few percent. Considering the SpMV on M2 (Fig. 7,6) and M1 (Fig. 11,10), when executing 1 or 2 threads, we notice a stable performance level (almost) regardless of the matrix. For 4-8 threads, we see more variation, as there is more memory channel usage, and the memory access pattern becomes less regular (remember the naive nature of the algorithm in Fig. 2). On M3 (Fig. 15,14) , we do not observe such regularities, and we witness what seems a memory bottleneck when moving to 4 and 8 threads. The symmetric kernels (Fig. 15 ) encounter the first scalability problems on 4 threads, as they saturate the memory channel at a write-to-read rate which is double respect to unsymmetric kernels. On the other hand, on M2 and M1, the performance of the SpMV on symmetric matrices grows up to 8 threads (Fig. 7,11). M1 is the only machine able to achieve nearly linear speedup for the SpMV kernels. Compared to CSB, RCSR performs better with smaller number of threads, but then encounters a scalability (and performance) limit before CSB. Note that on matrix torso1, RCSR performs better, just as we experienced in [2] with a different parallelization strategy. The reason for these performance patterns in RCSR and CSB can be explained by: i) use of compressed indices in CSB (See [3]), which in many cases alleviate the memory bandwidth bottleneck; ii) regular access pattern in the CSR kernels operating on submatrices, leading to high performance at the cost of earlier limits from memory bandwidth.

<table>
<thead>
<tr>
<th>matrix</th>
<th>rows</th>
<th>columns</th>
<th>nnz</th>
<th>n./r.</th>
<th>matrix</th>
<th>rows</th>
<th>columns</th>
<th>nnz</th>
<th>n./r.</th>
</tr>
</thead>
<tbody>
<tr>
<td>torso1</td>
<td>1977885</td>
<td>109900</td>
<td>7791168</td>
<td>4</td>
<td>rajat3i</td>
<td>4690002</td>
<td>4690002</td>
<td>20316253</td>
<td>4</td>
</tr>
<tr>
<td>sme3Dc</td>
<td>42930</td>
<td>42930</td>
<td>3148656</td>
<td>73</td>
<td>torso1</td>
<td>116158</td>
<td>116158</td>
<td>8516500</td>
<td>73</td>
</tr>
</tbody>
</table>

TABLE I

 MATRICES WE USED IN OUR EXPERIMENTS. "N./R." MEANS "NNZ/ROWS".
### TABLE II

**SUMMARY OF TEST ENVIRONMENTS.**

<table>
<thead>
<tr>
<th>Machine Model</th>
<th>CPUs/Cores</th>
<th>Data Caches</th>
<th>Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1 IBM POWER 5 (91388-575)</td>
<td>16/1</td>
<td>1xL3:36MB (off-chip, not considered), L1:3.92MB/10-w/128B, L1:2.33KB/4-w/128B</td>
<td>Xlc 7.0 (AIX 5)</td>
</tr>
<tr>
<td>M2 AMD Opteron 2354 Quad-Core</td>
<td>2/4</td>
<td>2xL3:2.4x4xL2:1.92MB/10-w/128B (AIX 5)</td>
<td>Gcc 4.3.2 (Red Hat)</td>
</tr>
<tr>
<td>M3 Intel Xeon E5405 Quad-Core</td>
<td>2/4</td>
<td>2xL3:2MB/32-w/64B, L2:6MB/8-w/64B, L1:32KB/24-w/64B</td>
<td>Gcc 4.3.2 (Red Hat)</td>
</tr>
</tbody>
</table>

---

There are cases in which performance is bad across the board; this is particularly true for matrices that have as few as 4 elements per row (kkt_power, Rucci1, rajat31). On these matrices loading each right hand side vector element requires fetching of an entire line of cache, and with little or no spatial locality, this is too expensive. On M1, we witness a seemingly superlinear (Fig. 12) scaling in the SpMV for the matrix ohne2: the overall results on M1 indicate bad behaviour of serial RCSR, but this will need further investigation.

Looking at the results of the SpSV we note that, despite the similarity of the algorithms in Fig. 3 and 2, the performance scaling is sublinear, even on the M1. This is not surprising (see also considerations in [6]), as good performance of the parallel SpSV depends on the structure of the lower triangular matrix L; it must have a sufficient amount of parallel regions. In our case, matrices should have enough off-diagonal submatrices, to parallelize the critical path computations. While not astonishing, the observed speedups up to approximately 3 on M1(Fig. 5), to 2.5 on M2 (Fig. 9), and up to 1.5 on M3 (Fig. 13) conform to those reported in [6]. As an example of a difficult matrix, consider the L-matrix obtained from venkat50. After the LU decomposition, the majority of its nonzeroes in the diagonal blocks, and most blocks are on the main diagonal. For such matrices (“almost banded”), computation involves solving the diagonal subsystems serially, following with almost-serial operations on the few off-diagonal submatrices. Matrices likely to achieve a reasonable speedup from the multithreaded SpSV are the larger ones, as their fraction of submatrices located on the diagonal is smaller. For instance, the L factor of the matrix ohne2, on M2 (Fig. 5), is broken down in 1601 submatrices, of which only 10% are located on the diagonal. As a consequence of this we get an almost 3-fold SpSV speedup on M1.

### VI. CONCLUDING REMARKS

The results presented in this paper intend to illustrate the potential of the RCSR storage format for the implementation of main kernels (SpSV and SpMV) of a sparse level 2 BLAS. For both operations our unified approach was found to be competitive, against approaches based on specialized data structures ([6] and [3]). It has to be stressed that we did not employ any fine-tuning to the basic versions of the proposed format and algorithms. At the same time, there exist many possibilities for modifying both the algorithms and the data format itself that are likely to improve the performance of the RCSR without impairing its generality and/or functionality.
Here, observe that the sparsity pattern of matrices is a determinant factor of parallelism in the RCSR: lower banded triangular matrices tend to limit parallelism of the SpSV, which can easily use all the memory bandwidth available, especially if the matrix sparsity pattern is very irregular. One possible approach to limiting stalls due to insufficient memory bandwidth is the use of compressed indices, which we are currently exploring; in the perspective of many-cores environments (and the hypersparsity property; see [16]), approaches other than using CSR leaves may be advantageous. Another possibility would be the parallelization of the execution of the leaf SpMV dependencies in SpSV by a different row locking strategy. An important issue to consider, in the sparse BLAS-implementation context, is the application of recursive formats to all relevant numerical types and operation variants, and thus validation of the goodness of the recursion policy; the choice of the $cs$ parameter (see Sec. III-A) should be also be correlated to the number of cores/threads employed in the computation. For future assessments of the RCSR format, we deem as interesting the development of algorithms for Incomplete LU factorization and SpGEMM (multiplication of sparse matrices) algorithms.

Finally, we would like to thank the anonymous reviewers for the constructive feedback we received.

REFERENCES

Fig. 11. SpMV on M2, symmetric matrices.

Fig. 12. SpMV on M3, L factor matrices.

Fig. 13. SpSV on M3, L factor matrices.

Fig. 14. SpMV on M3, unsymmetric matrices.


Fig. 15. $SpMV$ on $M3$, symmetric matrices.

