An Improved Sparse Matrix-Vector Multiply Based on Recursive Sparse Blocks Layout

Michele Martone[1], Marcin Paprzycki[2], and Salvatore Filippone[1]

[1] University of Rome "Tor Vergata", Via del Politecnico 1, 00133 Rome, Italy
[2] Systems Research Institute, Polish Academy of Sciences, ul. Newelska 6, 01-447 Warsaw, Poland

Abstract. The *Recursive Sparse Blocks* (*RSB*) is a sparse matrix layout designed for coarse grained parallelism and reduced cache misses when operating with matrices, which are larger than a computer's cache. By laying out the matrix in sparse, non overlapping blocks, we allow for the shared memory parallel execution of transposed *SParse Matrix-Vector* multiply (*SpMV*), with higher efficiency than the traditional *Compressed Sparse Rows* (CSR) format. In this note we cover two issues. First, we propose two improvements to our original approach. Second, we look at the performance of standard and transposed shared memory parallel *SpMV* for unsymmetric matrices, using the proposed approach. We find that our implementation's performance is competitive with that of both the highly optimized, proprietary Intel MKL Sparse BLAS library's CSR routines, and the *Compressed Sparse Blocks* (CSB) research prototype.

1 Introduction and Related Work

Many scientific/computational problems require the solution of systems of partial differential equations (PDEs). Often, discretization of these problems result in *sparse* matrices. A common approach for the solution of sparse linear systems is through the use of *iterative methods*, whose computational core requires sparse matrix-vector multiplication. In this document, we focus on the efficient implementation of sparse matrix-vector multiplication, on *cache based, shared memory* computers. In this context, we have recently proposed a sparse matrix format, called *Recursive Sparse Blocks* (*RSB*) [3,4]. The central idea of RSB is a recursive partitioning-based organization of matrices, with either *Compressed Sparse Rows* (CSR) or *Coordinate* (COO) format *leaves* of a *quad-tree* structure over matrices. In this paper, we present some optimizations to our RSB-based *SpMV* implementation, and compare performance of the modified approach to that of the Intel's MKL proprietary Sparse BLAS implementation, and the publicly available CSB (see [1]) prototype. To this end, we briefly recall the way that the *SpMV* /*SpMV_T* computational kernels work and behave on computers of our interest in § 2. Next, we introduce the proposed optimizations in § 3. Finally in § 5, we discuss the efficiency of our prototype, by comparing it to the mentioned highly efficient MKL's CSR and CSB implementations.

I. Lirkov, S. Margenov, and J. Wańsiewski (Eds.): LSSC 2011, LNCS 7116, pp. 606–613, 2012.

```
1 for l ← 1 to s.nnz do
2     i ← s.IA(l);
3     j ← s.JA(l);
4     s.y(i + s.roff) ← s.y(i + s.roff) + s.VA(l)s.x(j + s.coff);
5 end
```

Fig. 1. *SpMV* listing for a COO submatrix s

```
1 for l ← 1 to s.nnz do
2     i ← s.IA(l);
3     j ← s.JA(l);
4     s.y(j + s.coff) ← s.y(j + s.coff) + s.VA(l)s.x(i + s.roff);
5 end
```

Fig. 2. *SpMV_T* listing for a COO submatrix s

2 *SpMV* and Transposed *SpMV*

We define the *sparse matrix-vector multiply* (*SpMV*) operation as "$y \leftarrow A\ x$" and its transposed version (*SpMV_T*) as "$y \leftarrow A^T\ x$" (where A is a sparse matrix, while x, y are vectors). With RSB, A is *recursively partitioned* into submatrices, and then the individual N *leaf submatrices* $s_1 : s_N$ are represented in either COO or CSR format (eventually using 16 bits for the local indices); for details, see [3,4]. The leaf submatrices are all disjoint; each submatrix s covers rows indices $[s.roff : s.roff + s.rows]$ and column indices $[s.coff : s.coff + s.cols]$. For this reason, the *SpMV* operation may be decomposed into the following N steps (for $n = 1, ..., N$) $y_{s_n.roff:s_n.roff+s_n.rows} \leftarrow y_{s_n.roff:s_n.roff+s_n.rows} + a_{s_n.roff:s_n.roff+s_n.rows,s_n.coff:s_n.coff+s_n.cols} x_{s_n.coff:s_n.coff+s_n.cols}$. Note that some steps may be executed in parallel by two or more threads. In the case with two different threads i, j operating on two different submatrices s_p, s_q, updating the two y intervals $s_p.roff : s_p.roff + s_p.rows$ and $s_q.roff : s_q.roff + s_q.rows$ is allowed, as long as the intervals do not intersect. In the case when the two intervals intersect, a *race condition* may occur; that is, concurrent updates of vector y may lead to inconsistent results in the intersecting y subvector. In the same spirit, the *SpMV_T* operation may be decomposed into $y_{s_n.coff:s_n.coff+s_n.cols} \leftarrow y_{s_n.coff:s_n.coff+s_n.cols} + a_{s_n.coff:s_n.coff+s_n.cols,s_n.roff:s_n.roff+s_n.rows} x_{s_n.roff:s_n.roff+s_n.rows}$. Clearly, while in the untransposed case the requirement for avoiding race conditions is on the *rows interval*, in the transposed case the *columns intervals* of the participating submatrices shall be disjoint. Our basic shared memory parallel algorithm for RSB/*SpMV* is outlined in Fig. 5 (see also [4]); in the listing, at line 8, assume for the time being $s.r_h = 0, s.r_t = 0$. Workload is partitioned among threads by means of a parallel section (lines 5-15). Repeatedly, each participating thread picks up a submatrix and updates the y array with its contribution to the product. When picking up a submatrix, a thread locks y array's interval corresponding to the submatrix rows interval. The same listing is suitable for *SpMV_T*,

after a slight modification; namely, the pairwise exchange in all occurrences of: *s.roff* and *s.coff*, *s.rows* and *s.cols*. The bulk of computation is executed at the leaf level, when either COO or CSR submatrices are multiplied by the corresponding subvector. See Fig. 3 for the CSR submatrices code (again, assume $s.r_h = 0, s.r_t = 0$), and Fig. 1 for the COO version of the *SpMV*. For the *SpMV_T*, see listings Fig. 4 and Fig. 2. We use the most common variant of CSR storing rows in ascending order, and column indices in ascending order within each row. The COO submatrices of RSB are organized exactly in the same way. A consequence of this layout ordered by rows is that for most real world matrices, for each given nonzero coefficient $a_{i,j}$, it is likely that the next stored nonzero $a_{i,j'}$ is quite near, i.e. with $\Delta = j' - j$ *reasonably small*. If $\Delta < C_l/N_s$, with C_l the *cache line length*, and N_s the floating point number size, both expressed in bytes, then after computing the contribution $y_i \leftarrow y_i + a_{ij}x_j$ (line 4 in Fig. 1, line 4 in Fig. 3), loading of element $x_{j'}$ will, with very high probability, only require a single fetch from cache memory, with no further cache misses. Normally both CSR and COO *SpMV* algorithms are written such that the compiler uses *registers* for the accumulation of the y_i contribution, while updating the y_i memory location no more than once per submatrix. In the case of CSR and listing Fig. 3, this is straightforward to achieve — it requires referencing a local variable instead y_i in the inner loop, and update of the y_i location right after the inner loop. In the case of COO listing, Fig. 1, one should reorganize in two loops: an outer one cycling on rows, and an inner one cycling on a single row nonzeroes. If the number of nonzeroes of a COO submatrix is likely to be less than the number of rows, as is the case for our leaf matrices because of the criteria for COO in [3], and discussion of *hypersparsity* in [2], we clearly have a problem, since such a double loop would perform $O(s.nnz + s.rows)$ control instructions, which is always more than $O(s.nnz)$ as in Fig. 1. In the case of CSR submatrices, we are constrained to a $O(s.nnz + s.rows)$ loop complexity by the nature of CSR. However, here we have a guarantee that the number of nonzeroes exceeds the number of rows (by the definition of RSB leaves—see [3]), so the double loop is not a concern.

3 Two Optimizations to RSB *SpMV* /*SpMV_T*

In this section we introduce two closely related modifications to our *SpMV* /*SpMV_T* algorithms for RSB. First, we note that our implementations for CSR

```
1 for i ← 1 + s.r_h to s.rows − s.r_t do
2     for l ← s.PA(i) to s.PA(i + 1) − 1 do
3         j ← s.JA(l);
4         s.y(i + s.roff) ← s.y(i + s.roff) + s.VA(l)s.x(j + s.coff);
5     end
6 end
```

Fig. 3. *SpMV* listing for a CSR submatrix s, with head/tail skipping

```
1 for i ← 1 + s.r_h to s.rows − s.r_t do
2     for l ← s.PA(i) to s.PA(i + 1) − 1 do
3         j ← s.JA(l);
4         s.y(j + s.coff) ← s.y(j + s.coff) + s.VA(l)s.x(i + s.roff);
5     end
6 end
```

Fig. 4. *SpMV_T* listing for a CSR submatrix s, with head/tail skipping

(Fig. 3, Fig. 4, considering $s.r_h, s.r_t, s.c_h, s.c_t$ set to zero) visit the whole PA (*rows pointer*) array once, reading exactly $s.rows+1$ locations. From [3] we have that for a matrix A stored in RSB, any of its submatrices s is stored in CSR only if $s.nnz > s.rows$ (or $\kappa_s \stackrel{def}{=} \frac{s.nnz}{s.rows} > 1$). In such a situation, storage of s uses exactly $I_{CSR}(s) \stackrel{def}{=} s.nnz + s.rows + 1$ indices, and these $I_{CSR}(s)$ indices are all read from memory during *SpMV* /*SpMV_T*. Even if A has no empty rows (rows with only zeroes), we have no guarantee that the same applies to any given s, especially given the way RSB *recursive subdivision* of matrices works. Moreover, it is very likely that for most matrices of interest to us, there will be some $s.r_h$ *empty heading rows*. Similarly, it is reasonable to assume that there are some $s.r_t$ *empty tail rows* also. Values of $s.r_h, s.r_t$ may be easily computed at matrix build time, and may be used in Fig. 3 and Fig. 4 to work on the non-empty interval of rows only, and simply *skipping* iterations on the empty ones. For a square submatrix s having $r_\epsilon(s) \stackrel{def}{=} s.r_h + s.r_t$, the use of this row skipping technique allows reducing the amount of indices read up to 50% (consider a square s with $s.nnz = s.rows+1$ nonzeroes distributed on two rows) with 4 byte indices and up to 66% with 2 bytes indices. In a more common situation, say $r_\epsilon(s) = s.rows/\nu_s$ (for some ν_s), one would save $s.rows/\nu_s$ accesses out of $s.rows(\kappa_s + 1) + 1$. For $s.rows \gg 1$, the saved fraction is $\frac{1}{\kappa_s \nu_s}$; for realistic cases, e.g.: $\kappa_s = 2, \nu_s = 2$, this amounts to 25%, which is not bad. A good property of this optimization is that in the case of no empty rows, there is no runtime performance loss compared to the base implementation. We also observe that this optimization is valid for both *SpMV* and *SpMV_T*. The second optimization also relies on the computation of $s.r_h/s.r_t$, but is applied to the outer parallel algorithm shown in Fig. 5. The base version of this algorithm considers to be zero both $s.r_h, s.r_t$, on all submatrices; thus when a submatrix is picked up by a thread, the entire $s.roff...s.roff+s.rows$ interval of the output vector y may have to be updated, and therefore is locked. But if a submatrix s has $s.r_h$ empty heading rows, and $s.r_t$ empty tail rows, then rows outside the $s.roff + s.r_h...s.roff + s.rows - s.r_t$ range will not be modified; therefore we observe that only the corresponding subvector of y must be really locked. We apply our optimization by locking only the effective rows interval, thus allowing for a reduced degree of resource contention among threads and enhancing potential parallelism. The worst case is when $s.r_h = 0$ and $s.r_t = 0$, and this is no worse than (indeed, identical to) without the optimization. The best case may be when N_s submatrices extending on the same rows range exists, but

```
1  S ← [s_0, s_1, ..., s_{N-1}] /*an array of terminal submatrices, in any order*/ ;
2  B ← [0, 0, .., 0] /*a zero bit for each submatrix*/ ;
3  n ← 0 /*count of visited submatrices so far*/;
4  while n < N do
5      begin parallel ;
6      s ← pick an unvisited submatrix s from S;
7      /*(should have picked up s ← S[i], with B[i] = 0)*/ ;
8      [f, l] ← [s.roff + s.r_h , s.roff + s.rows − s.r_t] ;
9      if locked([f ... l]) then cycle ;
10     lock([f ... l]) /*we lock y on s's effective rows interval*/ ;
11     /*perform SpMV on s and x[s.coff:s.coff+s.cols] into y[f : l]*/ ;
12     y[f : l] ← y[f : l] + s · x[s.coff :s.coff + s.cols] ;
13     B[i] ← 1; n ← n + 1 ;
14     unlock([f ... l]) ;
15     end parallel;
16 end
```

Fig. 5. Multithreaded *SpMV* for leaf submatrices of a RSB matrix, with head/tail skipping

each submatrix has its nonzeroes laid out only on a contiguous group of rows, in a way that there is no intersection of non-empty row intervals, for any given pair of submatrices. In this limit case, all of the submatrices may be processed in parallel, with potential N_s-fold parallelism. For a more realistic case, consider a pair of matrices s, s', whose nonzeroes have no common row (think of a case when a large banded matrix is subdivided), but both have the same row offset ($s.roff = s'.roff$) and extension ($s.rows = s'.rows$). In this case, we double the potential parallelism with little effort. This optimization may also be applied to the transposed *SpMV*, if we use the *empty heading columns* $s.c_h$, instead of $s.r_h$, *empty tail columns* $s.c_t$ instead of $s.r_t$, and swap usage of *s.roff* with *s.coff* in Fig. 5.

4 Experimental Setup and Methodology

For space reasons we report only a limited set of experimental data. We chose to use a sample of large (exceeding hardware cache), sparse square matrices obtained from the *University of Florida Sparse Matrix Collection* (see [5]). Matrices information is summarized in Table 1. We report results of experiments performed on an Intel Xeon 5670, supporting up to 12 hardware threads, with 3 levels of cache memory (sized respectively 32KB/256KB/12MB). Our codes were implemented in C99, and compiled with Intel's ICC v.12.0.2 compiler, with the optimization `-O3` flag (no machine specific optimization flags were used). Our parallel RSB implementation using OpenMP is compared against the CSR implementation present in the proprietary Intel MKL 10.3-2 library, and the publicly available CSB (see Buluç et al. [1]) prototype (compiled with the special purpose CILK++ compiler, version 8503, with `-O3 -fno-rtti -fno-exceptions` flags).

Table 1. List of matrices used in the experiments, with their row/column dimensions (r/c), nonzeroes count (nnz), average nonzeroes per row (nnz/r) count

matrix	r	c	nnz	nnz/r
cage15	5154859	5154859	99199551	19.24
circuit5M_dc	3523317	3523317	19194193	5.45
fem_hifreq_circuit	491100	491100	20239237	41.21
GL7d18	1955309	1548650	35590540	18.20
patents	3774768	3774768	14970767	3.97
RM07R	381689	381689	37464962	98.16
TSOPF_RS_b2383	38120	38120	16171169	424.22
wikipedia-20070206	3566907	3566907	45030389	12.62

Here we also use explicit *loop unrolling* (four-fold) on the COO and inner CSR loops of our codes. We express the performance of a computation in MFLOPS (that is, *time efficiency*); we count 2 operations for each nonzero of a matrix involved in the multiplication by a vector.

5 Results

Let us now discuss the experimental results. In Fig. 6 we report results obtained using 12 threads; in Fig. 7 results for a single thread. The first thing we note, is that the MKL (CSR) results for $SpMV\_T$ are consistently lower than for $SpMV$. This performance gap is due to the row-major layout of CSR, which requires the transposed update of the results array to be written at *random* locations (unlike the normal update, which reads random locations of the multiplicand vector, but updates a sequentially accessed array). This gap is almost absent in the case of CSB: recall (see [1]) the unbiased Z-ordering in its *sparse blocks*. Regarding the $SpMV$ / $SpMV\_T$ gap, RSB falls in between MKL and CSB, due to its storage of consecutive rows of *sparse submatrices*. We notice that running in parallel (Fig. 6), the aforementioned efficiency gap is much more pronounced for CSR. The reason for this is the lack, in the CSR format, of immediate information for the serialization of the threads write instructions in updating the result

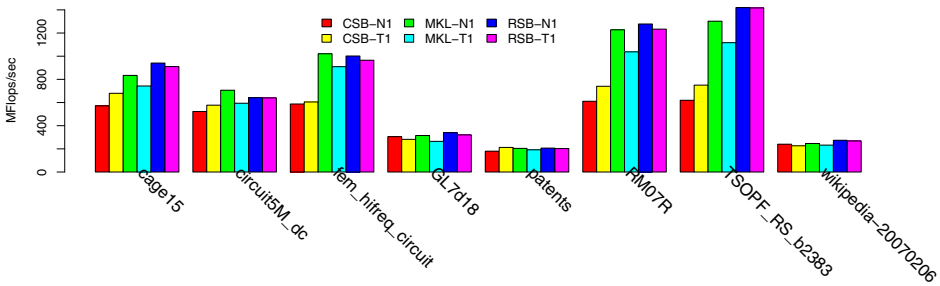

Fig. 6. $SpMV$ and $SpMV\_T$ performance, 12 threads

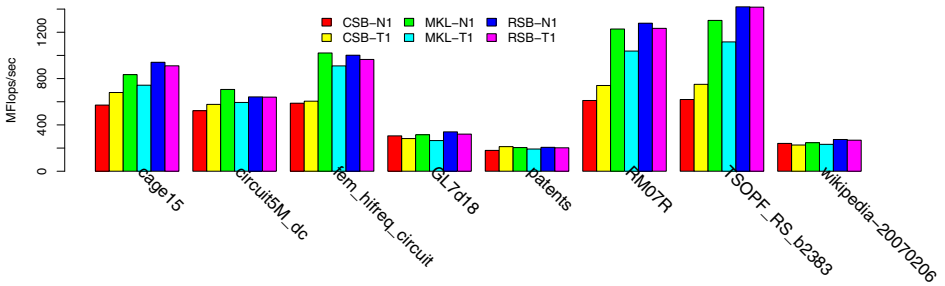

Fig. 7. *SpMV* and *SpMV_T* performance, 1 thread

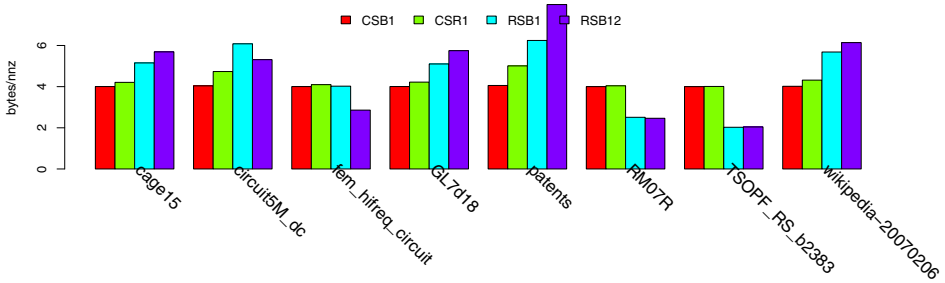

Fig. 8. Bytes per stored nonzero for CSB, CSR, and RSB-1/RSB-12

vector. CSB and RSB are structured in blocks, offering a coarse grained way to parallelization of *SpMV_T*. Summarizing, for the chosen set of matrices, RSB performs always better than CSB and MKL with a single thread, and almost always, in parallel runs. In interpreting performance results for the three formats (especially when running in parallel, when the CPU-memory communications channels are likely to be saturated), we may consider the *bytes per indexing nonzero* metric. Since RSB's index usage depends on the subdivisions/threads count, in Fig. 8 we report this value for both single and 12 threaded runs. For CSB, we report the size of the arrays allocated in the source code, although we do not take into account here the *block pointers* array (see [1]), which also contributes as index-related memory traffic. It is straightforward to see that usually, RSB's higher performance cases in Fig. 6 coincide with the shortest index usage cases, and vice-versa. We also note that the relative performance of CSB/MKL seems related to the average indexing usage.

6 Concluding Remarks

In this paper, we have proposed two simple optimizations to our RSB algorithms for *SpMV* /*SpMV_T*, and performed experiments on large sparse matrices.

Since the two optimizations have no potential negative impact, and since their benefit is difficult to quantify in advance, we skipped the comparison to the past code versions, and compared the code directly to two different, efficient *SpMV* implementations: Intel MKL's CSR and CSB (see Buluç et al. [1]). Our main finding is that the block structure of RSB allows the parallel implementations of both *SpMV* /*SpMV_T* to be efficient without the prominent performance gap which is inherent in a CSR implementation (in this case, MKL's). We also find confirmation that RSB's hybrid structure (a recursive layout on outside, with a row-major layout on the inside) is advantageous when performing *SpMV* /*SpMV_T* on large matrices serially. Furthermore, the technique of enhancing RSB's parallelism by using empty rows information applies to triangular solve and symmetric *SpMV* kernels as well.

We wish to thank Paweł Gepner and Jamie Wilcox at Intel Corporation for giving us access and technical support for the machine used in the experiments.

References

1. Buluç, A., Fineman, J.T., Frigo, M., Gilbert, J.R., Leiserson, C.E.: Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In: auf der Heide, F.M., Bender, M.A. (eds.) SPAA, pp. 233–244. ACM (2009)
2. Buluç, A., Gilbert, J.R.: On the Representation and Multiplication of Hypersparse Matrices. In: IEEE International Parallel and Distributed Processing Symposium (IPDPS 2008), pp. 1–11 (April 2008)
3. Martone, M., Filippone, S., Paprzycki, M., Tucci, S.: About the assembly of recursive sparse matrices. In: Proceedings of the International Multiconference on Computer Science and Information Technology, Wisła. Poland, pp. 317–325. IEEE Computer Society Press, Los Alamitos (2010)
4. Martone, M., Filippone, S., Paprzycki, M., Tucci, S.: On BLAS operations with recursively stored sparse matrices. In: Proceedings of the International Symposium on Symbolic and Numeric Algorithms for Scientific Computing. Timisoara, Romania, pp. 49–56. IEEE (September 2010)
5. The University of Florida Sparse Matrix Collection, `http://www.cise.ufl.edu/research/sparse/matrices`