

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

Michele Martone<sup>1</sup>, Marcin Paprzycki<sup>2</sup>, and Salvatore Filippone<sup>1</sup>

<sup>1</sup> University of Rome “Tor Vergata”, Via del Politecnico 1, 00133 Rome, Italy

<sup>2</sup> 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.

```

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

```

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

```

1 for  $l \leftarrow 1$  to  $s.nnz$  do
2    $i \leftarrow s.IA(l)$ ;
3    $j \leftarrow s.JA(l)$ ;
4    $s.y(j + s.coff) \leftarrow s.y(j + s.coff) + s.VA(l)s.x(i + s.roff)$ ;
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 “ $y \leftarrow Ax$ ” 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  $n = 1, \dots, N$ )  $y_{s_n.roff:s_n.roff+s_n.rows} \leftarrow y_{s_n.roff:s_n.roff+s_n.rows} + a_{s_n.roff:s_n.roff+s_n.rows, s_n.coff:s_n.coff+s_n.cols} x_{s_n.coff:s_n.coff+s_n.cols}$ . 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*,