Auto-tuning shared memory parallel Sparse BLAS operations in `librsb`-1.2

Michele MARTONE (michele.martone@ipp.mpg.de)

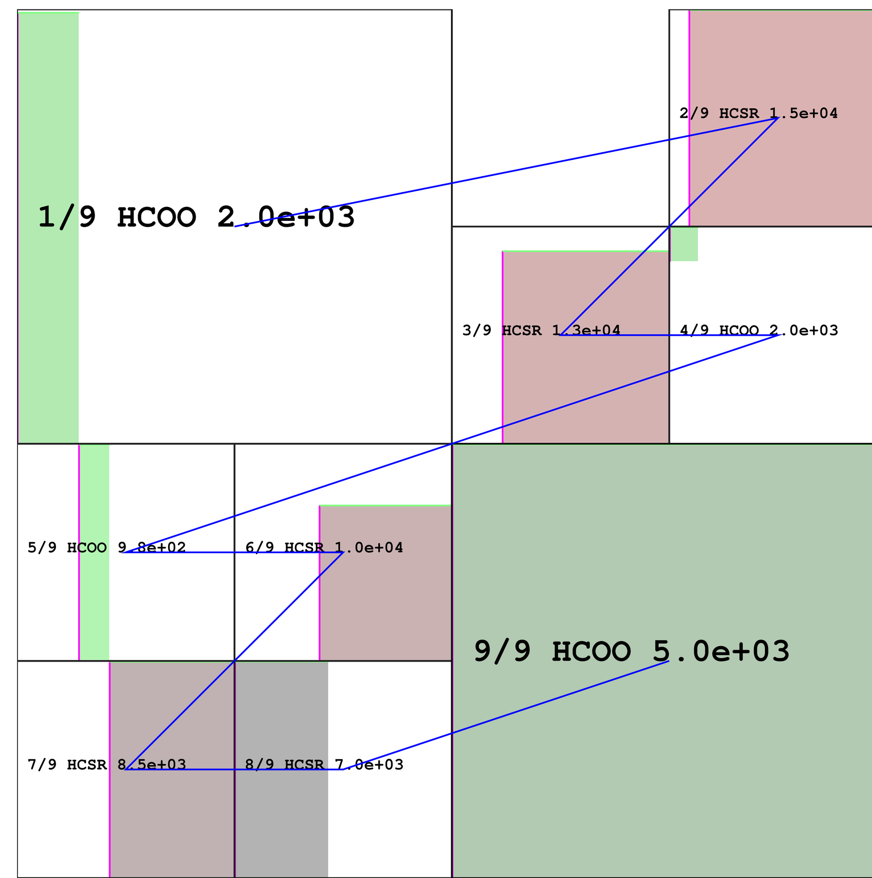
Max Planck Institute for Plasma Physics, Garching bei München, Germany

Intro

- Sparse BLAS (Basic Linear Algebra Subroutines) [2] specifies main kernels for iterative methods:
  - *sparse **M**ultiply by **M**atrix*: "**MM**"
  - *sparse triangular **S**olve by **M**atrix*: "**SM**"
- Focus on **MM**: $C \leftarrow C + \alpha op(A) \times B$, with
  - A has dimensions $m \times k$ and is *sparse* (nnz nonzeroes)
  - $op(A)$ can be either of $\{A, A^T, A^H\}$ (parameter $transA$)
  - *left hand side* (**LHS**) B is $k \times n$, *right hand side* (**RHS**) C is $n \times m$ (n=NRHS), both dense (eventually with *strided* access $incB, incC, ...$)
  - α is scalar
  - either single or double precision, either `real` or `complex`
- `librsb` implements the Sparse BLAS using the **RSB** (**Recursive Sparse Blocks**) data structure [3].
- Hand tuning for each operation variant is impossible.
- We propose **empirical auto-tuning** for `librsb`-1.2.

RSB: Recursive Sparse Blocks

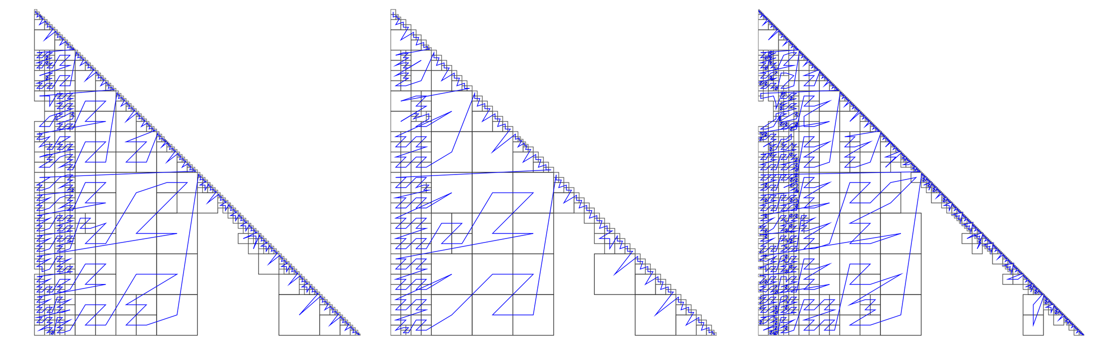
- *Sparse blocks* in COO or CSR [1].
- ...eventually with 16-bit indices ("HCOO" or "HCSR").
- *cache blocks* suitable for *thread parallelism*.
- *Recursive partitioning* of submatrices results in Z-ordered blocks.



- Instance of matrix *bayer02* (ca. $14k \times 14k, 64k$ nonzeroes).
- The black-bordered boxes are *sparse blocks*.
- Greener have fewer nnz than average, redder have more.
- Blocks rows (columns) of **LHS** (**RHS**) range during **MM**.
- Larger submatrices like "**9/9**" can have fewer nonzeroes than smaller ones like "**4/9**".

Merge / split based autotuning

- Optimal default blocking ?
  - Irregular matrix patterns !
  - Operands (especially $transA$, NRHS) change memory footprint !
- **Empirical auto-tuning**:
  - Given a Sparse BLAS operation, probe for a *better performing* blocking.
  - Search among *slightly coarser* or *finer* ones.

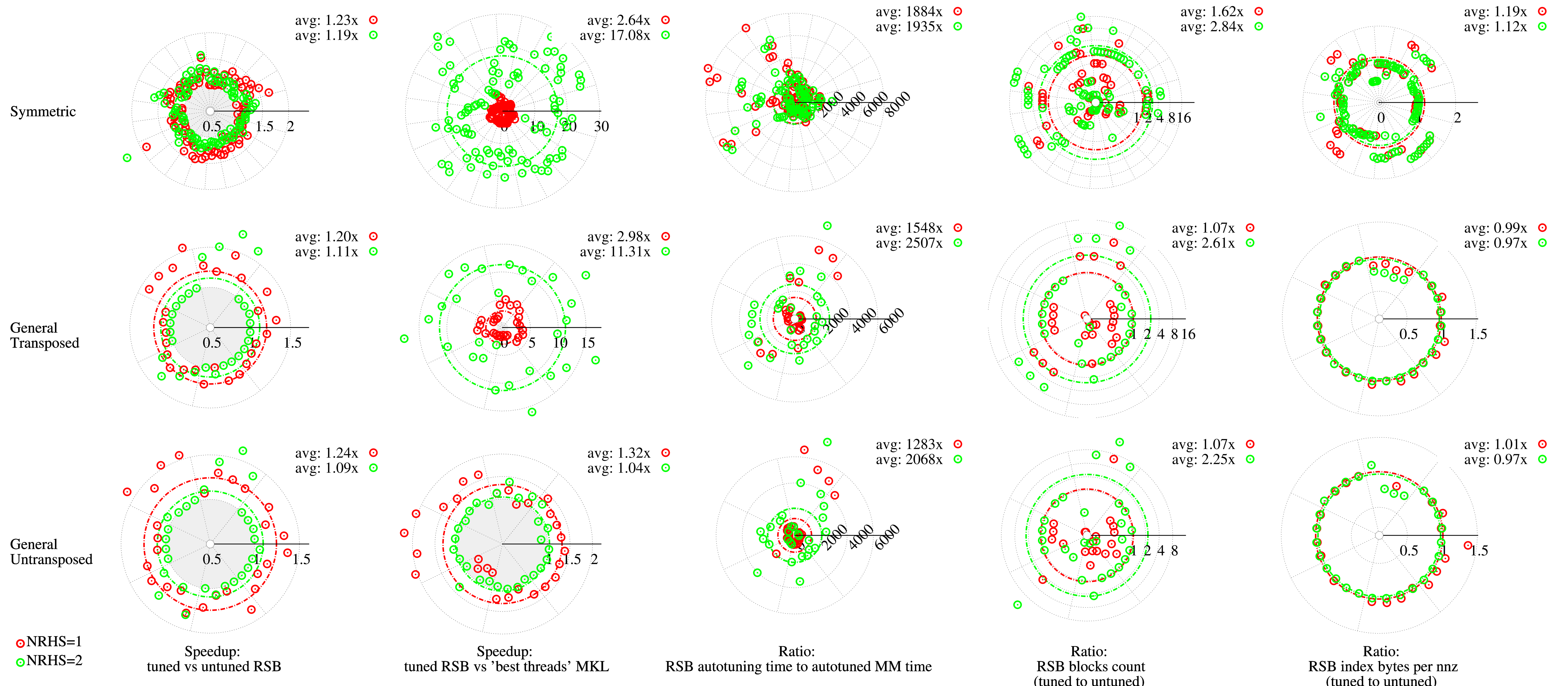


- Tuning example on symmetric matrix *audikw_1*.
- Here only lower triangle, ca. $1M \times 1M, 39M$ nonzeroes.
- On a machine with 256 KB sized L2 cache.
- Left one (625 blocks, avg 491 KB) is before tuning.
- Middle one (271 blocks, avg 1133 KB) after tuning (1.057x speedup, 6436.6 ops to amortize) for **MV** (**MM** with NRHS=1).
- Right one (1319 blocks, avg 233 KB) after tuning (1.050x speedup, 3996.2 ops to amortize) for **MM** with NRHS=3.
- Finer subdivision at NRHS=3 consequence of increased cache occupation of per-block **LHS**/**RHS** operands.

Sparse BLAS autotuning extension

```
1  ! Matrix-Vector Multiply: y <- alpha*op(A)*x+y
2  call USMV(transA,alpha,A,x,incx,y,incy,istat)
3  ! Tuning request for the next operation
4  call USSP(A,blas_autotune_next_operation,istat)
5  ! Matrix structure and threads tuning
6  call USMV(transA,alpha,A,x,incx,y,incy,istat)
7  ...
8  do ! A is now tuned for y <- alpha*op(A)*x+y
9    call USMV(transA,alpha,A,x,incx,y,incy,istat)
10 ...
11 ! Request autotuning again
12 call USSP(A,blas_autotune_next_operation,istat)
13 ! Now tune for C <- C + alpha * op(A) * B
14 call USMM(order,transA,nrhs,alpha,A,B,ldB,C,ldC,istat)
15 ! The RSB representation of A is probably different than before USMM
```

Experiment in MM tuning and comparison to MKL



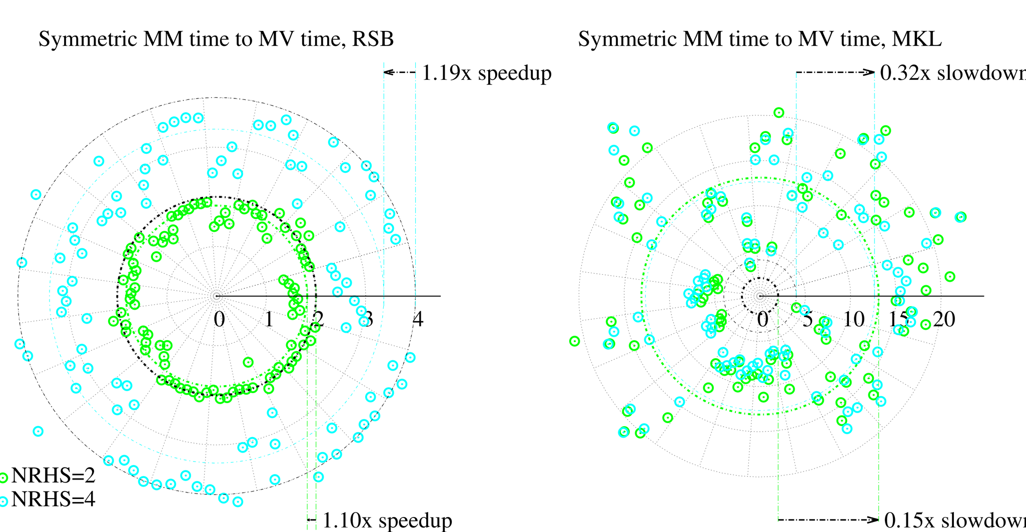
Setup

- `librsb` (`icc -O3 -xAVX`, v15) vs Intel MKL (v.11.2) CSR.
- 2× *"Intel Xeon E5-2680"*, 16 OpenMP threads.
- **MM** with NRHS={**1**,**2**}, four BLAS numerical types.
- 27 matrices in total (as in [3]), including symmetric.

Results Summary

- **Few dozen percent** improvement over untuned, costing **few thousand** operations.
- **Significantly faster than Intel MKL** on symmetric and transposed operation with NRHS=2 (> 1).
- **Autotuning** more effective on **symmetric** and **unsymmetric untransposed** with **NRHS=1**.
- Tuning mostly **subdivided further** for NRHS=2.

Highlight: symmetric MM vs MV performance



RSB Symmetric **MM** performance increases when NRHS>1, while Intel MKL CSR' falls. See here for NRHS=2 and NRHS=4.

Outlook

One may improve via:

- **Reversible in-place merge and split**: no need for copy while tuning.
- Best merge/split choice not obvious: **different merge and split rules**.
- **Non-time efficiency criteria** (e.g. use an energy measuring API when picking **better performing**).

References

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

[2] BLAS Forum. Sparse BLAS specification (BLAS specification, chapter 3). Technical report.

[3] M. Martone. Efficient multithreaded untransposed, transposed or symmetric sparse matrix-vector multiplication with the recursive sparse blocks format. *Parallel Computing*, (40):251–270, 2014.

EXASCALE15, Workshop *Sparse Solvers for Exascale: From Building Blocks to Applications*, 23-25 March 2015, Greifswald, Germany

Auto-tuning shared memory parallel Sparse BLAS operations in `librsb`-1.2

Michele MARTONE (michele.martone@ipp.mpg.de)

Max Planck Institute for Plasma Physics, Garching bei München, Germany

1. Introduction, definitions

The Sparse BLAS (Basic Linear Algebra Subroutines) standard [2] specifies the programming interface to performance critical computing kernels on sparse matrices: **multiply by dense matrix** ("**MM**") and **triangular solve by dense matrix** ("**SM**").

Here, **MM** is defined as: $C \leftarrow C + \alpha op(A) \times B$, where:

- A has dimensions $m \times k$ and is *sparse* (nnz nonzeroes)
- $op(A)$ can be either of $\{A, A^T, A^H\}$ (parameter $transA$)
- *left hand side* (**LHS**) B is $k \times n$,
  right hand side (**RHS**) C is $n \times m$,
  both dense (eventually with *strided* access $incB, incC$)
- α is a scalar
- either single or double precision, either `real` or `complex`

The matrix A can be *general* ($A \neq A^T$) or *symmetric* ($A = A^T$), eventually *Hermitian* ($A = \overline{A^T}$). In the following, using the BLAS jargon we will refer to n as NRHS. If NRHS=1, **MM** is equivalent to *sparse matrix-vector multiply*: "**MV**".

SM is defined as $B \leftarrow \alpha op(T)^{-1}B$, being T triangular sparse.

Gearing a Sparse BLAS implementation to be efficient in all the above cases with all the different possible inputs is challenging.

`librsb` is a library based on the RSB (Recursive Sparse Blocks) data structure [3] and implements Sparse BLAS with the aim of offering efficient multithreaded operation. Our approach in seeking both generality and efficiency employs **empirical auto-tuning**. Namely, the user specifies a Sparse BLAS operation to be iterated many times (using the same fixed-pattern matrix). Then a tuning procedure attempts to rearrange the sparse matrix data structure to perform that operation faster. If it fails, time taken by the autotuning procedure is lost. But if it succeeds, the now faster iterations will ultimately lead to amortize the invested tuning time.

This poster introduces the idea of tuning RSB for Sparse BLAS operations. Although the autotuning framework applies to **SM** as well, experiments presented here pertain **MM** only.

2. RSB and merge / split autotuning

The RSB format represents a numerical matrix in memory by means of smaller *sparse blocks*, laid out in memory according to a Z-order. This ordering arises by repeated (recursive) partitioning of the matrix coordinate domain in quadrants. Each sparse block represents the given submatrix using one of the conventional COO or CSR sparse formats [1], eventually with shorter, 16-bit indices ("HCOO", "HCSR"). **Figure 1** shows an RSB instance of a publicly available matrix.

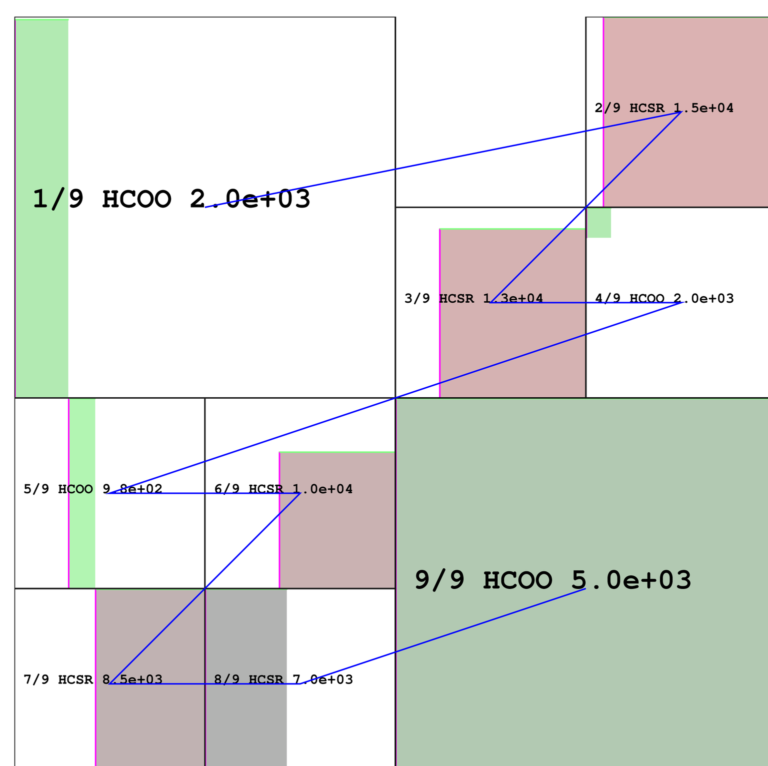


Figure 1: Graphical representation of an RSB data structure instance for matrix *bayer02* (ca. $14k \times 14k, 64k$ nonzeroes). The black-bordered boxes denote the *sparse blocks*, with the blue zig-zag line following their succession in memory. Each submatrix block is labeled with a number, its sparse format and contained nonzeroes. Inner coloured boxes delimit the row/column ranges of contained nonzeroes. Greener boxes have fewer nonzeroes than average; redder ones have more.

The magenta (green) segments aside of each box signal the range of **LHS** (**RHS**) matrix that will be updated (read) during untransposed **MM**.

Notice how larger submatrices (like "`9/9`") do not necessarily contain more nonzeroes, and how the area containing nonzeroes can be much smaller than the blocks' (like in "`4/9`").

During a Sparse BLAS operation, multiple threads can operate on separate blocks in parallel. Namely, each block will be assigned to a thread for the time of the operation on it. Then, the thread will update the corresponding **LHS** and use both the block and the **RHS** data.

It is well known that an opportune sizing of the data structure (i.e.*cache blocking*) favours efficiency. However, an *optimal* sparse blocks subdivision for most real problems is impossible to determine. `librsb`'s default is to arrange each matrix block plus its **LHS** and **RHS** (NRHS=1) occupation to be near to the (thread local) cache size.

In `librsb`-1.2 we attempt to improve this guess by means of an empirical auto tuning procedure, sketched in **Algorithm 1**. This procedure probes also for a better, matrix specific thread count (θ) choice.

- Call **PROBE** to get reference performance and if requested by the user, update θ.
- A bit-to-bit [a] copy of A, A' is made; then repeatedly:
  - Call **PROBE** on A' to get reference performance and eventually θ'.
  - Continue **coarsening** as long as performance improves.
- If the new (A', θ') pair performs better, go to the last step; otherwise continue.
- A fresh bit-to-bit copy of A, A' is made; then repeatedly:
  - Call **PROBE** on A' to get reference performance and eventually θ'.
  - Continue **subdividing** as long as performance improves.
- If the new (A', θ') pair performs better, autotuning failed.
- Otherwise, autotuning succeeded and (A, θ) is substituted with (A', θ') .

Algorithm 1: Sketch of **AUTOTUNE** procedure. It tunes an RSB matrix instance A for Sparse BLAS operation $\Omega \in \{$**MM**, **SM**$\}$. Uses θ threads specification and operands from $\omega = (transA, B, C, incB, incC, \alpha)$.

[a] Nevertheless, NUMA-wise the two may differ.

- If θ is specified, that will be used to compute execution time of operation Ω on instance A and operands from ω.
- If θ is not specified, it will be determined by repeating the above on a *reasonably* restricted range.
- A minimum of 10 executions is required.

Algorithm 2: Sketch of **PROBE** procedure, determining **reference performance** of a specified Sparse BLAS operation Ω on matrix A and operands from ω. **Thread count** specification θ may request threads probing as well.

With **Algorithm 1** it shall be possible to remedy to overly fine or coarse partitionings. Subdividing or coarsening is thread parallel and one full step takes the time of a few memory copies of the interested area. Coarsening converts and combines quadrants of possibly different formats; subdividing requires repeated searches to split a sparse submatrix structure.

Figure 2 shows the results of tuning for **MV** and **MM** with NRHS=3, leading to two different *tuned* partitionings.

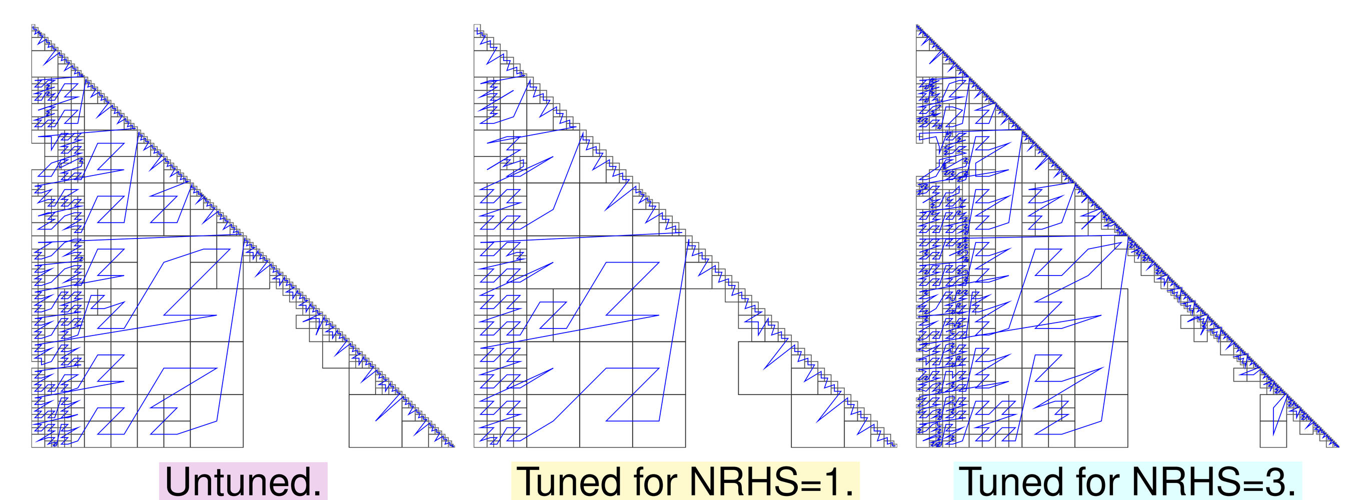


Figure 2: Sample instances of symmetric matrix *audikw_1* (ca. $1M \times 1M, 39M$ nonzeroes) on a machine with 256 KB sized L2 cache. The left one (625 blocks, avg 491 KB) is before tuning. Middle (271 blocks, avg 1133 KB) after tuning (1.057x speedup, 6436.6 ops to amortize) for **MV** (NRHS=1). Right (1319 blocks, avg 233 KB) after tuning (1.050x speedup, 3996.2 ops to amortize) for **MM** with NRHS=3. The middle structure resulted in merging the original's submatrices; the right one in subdividing further. The finer subdivision at NRHS=3 is a consequence of the increased cache occupation of the **LHS**/**RHS** vectors during operation.

3. Sparse BLAS autotuning extension

We propose a minimal extension that introduces the notion of autotuning in Sparse BLAS portably. Relying on the `USSP` (*unstructured sparse set property*) mechanism, a user may *request* autotuning to be performed on the *next* Sparse BLAS operation. The example in **Figure 3** illustrates its use.

```
 1 ! Matrix-Vector Multiply: y <- alpha*op(A)*x+y
 2 call USMV(transA,alpha,A,x,incx,y,incy,istat)
 3 ! Tuning request for the next operation
 4 call USSP(A,blas_autotune_next_operation,istat)
 5 ! Matrix structure and threads tuning
 6 call USMV(transA,alpha,A,x,incx,y,incy,istat)
 7 ...
 8 do ! A is now tuned for  y <- alpha*op(A)*x+y
 9   call USMV(transA,alpha,A,x,incx,y,incy,istat)
10 ...
11 ! Request autotuning again
12 call USSP(A,blas_autotune_next_operation,istat)
13 ! Now tune for  C <- C + alpha * op(A) * B
14 call USMM(order,transA,nrhs,alpha,A,B,ldB,C,ldC,istat)
15 ! A is now tuned possibly differently than before
```

Figure 3: We propose introduction of the `blas_autotune_next_operation` *BLAS property* keyword to enable implementation-agnostic portable tuning support.

4. Experimental results

In [3], we compared performance of `librsb` to Intel MKL CSR's for **MV** (`mkl_?csrmv`, equivalent to **MM** with NRHS=1), including symmetric and transposed variants, on a sample of 27 matrices. Here we extend that experiment to all of the four BLAS numerical types and consider **MM** (NRHS=2) as well. We measure effectiveness of the **AUTOTUNE** procedure (how much RSB improves) and compare to the results of MKL CSR (using `mkl_?csrmm` when NRHS=2). Since Intel MKL CSR does not provide autotuning, we determined the best thread count for each MKL sample ourselves. Additionally, we have collected autotuning and subdivision and tuning cost statistics. For detailed (e.g. type or matrix specific) results, please see the attached sheet.

- A **few dozen percent** improvement is often achievable at the cost of **few thousand** operations.
- RSB can be **significantly faster than Intel MKL**, especially on symmetric and transposed operation with NRHS$>$1 (`mkl_?csrmm`, zero-based indexing).
- RSB **autotuning** is more effective on **symmetric** and **unsymmetric untransposed** with **NRHS=1**.
- RSB has been slower than MKL in 1% of the symmetric cases, $\approx 1/3$ of the general untransposed cases.
- Tuning mostly **subdivided further** for NRHS=2.

5. Outlook

We have shown that efficiency and ease of use are not mutually exclusive, and they can be achieved while respecting a standard specification. At the same time, we believe that our techniques can be further improved by:

- **Reversible in-place merge and split.** The current coarsening/subdividing procedures are not reversible. Consequently in order to preserve the original input matrix structure an extra copy is being used.
- **Different merge and split policies.** The best merge/split choice is not obvious. A combination of the following strategies may prove effective: a) choosing the largest/smallest ones first; b) using a large block splitting strategy; c) reducing the indexing usage.
- **Non-time efficiency criteria.** The **PROBE** procedure might be changed to use e.g. an energy based metric.

References

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

[2] BLAS Forum. Sparse BLAS specification (BLAS specification, chapter 3). Technical report.

[3] M. Martone. Efficient multithreaded untransposed, transposed or symmetric sparse matrix-vector multiplication with the recursive sparse blocks format. *Parallel Computing*, (40):251–270, 2014.

EXASCALE15, Workshop *Sparse Solvers for Exascale: From Building Blocks to Applications*, 23-25 March 2015, Greifswald, Germany