lib**rsb**: A Shared Memory Parallel Sparse BLAS Implementation using the **R**ecursive **S**parse **B**locks format

Michele Martone

High Level Support Team
Max Planck Institute for Plasma Physics
Garching bei Muenchen, Germany

Sparse Days'13
Toulouse, France
June 17–18, 2013

presentation outline

Focus of the Sparse BLAS operations

The numerical solution of **linear systems** of the form $Ax = b$ (with A a sparse matrix, x, y dense vectors) using **iterative methods** requires repeated (and thus, **fast**) computation of (variants of) *Sparse Matrix-Vector Multiplication* and *Sparse Matrix-Vector Triangular Solve*:

- *SpMV* : "$y \leftarrow \ \beta \ y \ + \ \alpha \ A \ x$"
- *SpMV-T* : "$y \leftarrow \ \beta \ y \ + \ \alpha \ A^T \ x$"
- *SymSpMV*: "$y \leftarrow \ \beta y + \alpha \ A^T \ x, \ A = A^T$"
- *SpSV* : "$x \leftarrow \ \alpha \ L^{-1} \ x$"
- *SpSV-T* : "$x \leftarrow \ \alpha \ L^{-T} \ x$"

`librsb`: a high performance Sparse BLAS implementation

- uses the **Recursive Sparse Blocks (RSB)** matrix layout
- provides (for true) a Sparse BLAS standard API
- most of its numerical kernels are *generated* (from GNU M4 templates)
- extensible to any *integral* C type
- pseudo-randomly generated testing suite
- get/set/extract/convert/... functions
- $\approx$ 47 KLOC of C (C99), $\approx$ 20 KLOC of M4, $\approx$ 2 KLOC of GNU Octave
- bindings to C and Fortran (ISO C Binding), as *OCT module* for GNU Octave
- LGPLv3 licensed

design constraints of the Recursive Sparse Blocks (RSB) format

- parallel, efficient *SpMV* /*SpSV* /*COO* → *RSB*
- in-place *COO* → *RSB* → *COO* conversion
- no oversized COO arrays / no fillin (e.g.: in contrast to BCSR)
- no need to pad x, y[1] vectors arrays
- architecture independent (only C code, POSIX)
- developed on/for shared memory cache based CPUs:
  - *locality of memory references*
  - coarse-grained *workload partitioning*

---

[1] e.g. in *SpMV*

A recursive matrix storage: *Recursive Sparse Blocks* (RSB)

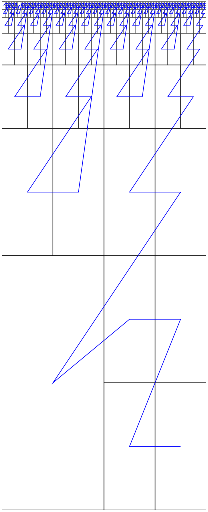
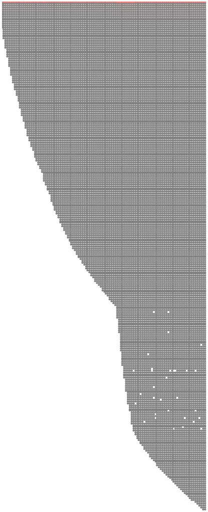
we propose:

- a *quad-tree* of sparse *leaf* submatrices
- outcome of recursive *partitioning* in *quadrants*
- leaf submatrices are stored by either *row oriented Compressed Sparse Rows* (CSR) or *Coordinates* (COO)
- an *unified* format for Sparse **BLAS**[2] operations and variations (e.g.: diagonal implicit, one or zero based indices, transposition, complex types, stride, ...)
- partitioning with regards to both the underlying cache size **and** available threads
- leaf submatrices are *cache blocks*

---

[2] Sparse Basic Linear Algebra Subprograms, e.g.: as in TOMS Algorithm 818 (Duff and Vömel, 2002). or the specification in `http://www.netlib.org/blas/blast-forum/chapter3.pdf`

Instance of an Information Retrieval matrix (573286 rows, 230401 columns, $41 \cdot 10^6$ nonzeroes):

sparse blocks layout:  spy plot: 

(courtesy from Diego De Cao, Univ. Roma Tor Vergata)

Adaptivity to threads count

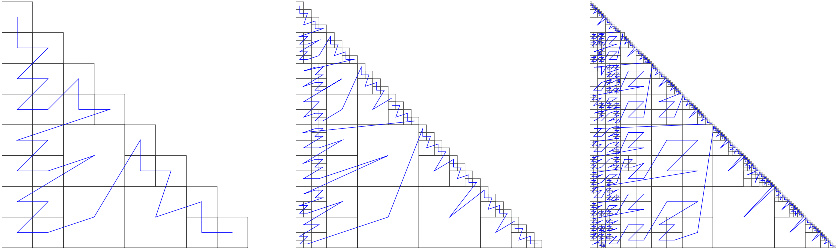


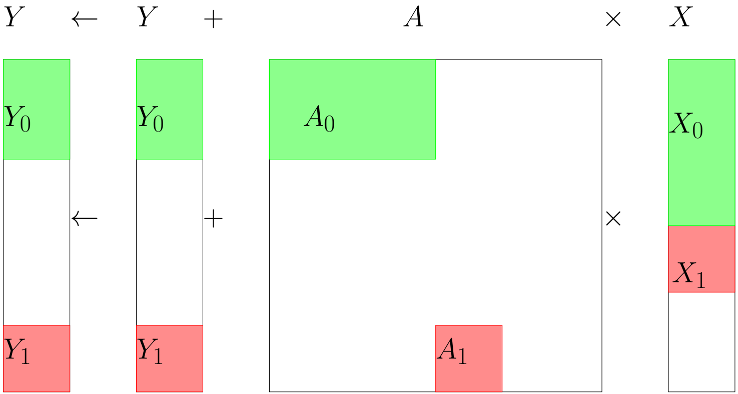
Figure: Matrix *audikw_1* (symmetric, 943695 rows, $3.9 \cdot 10^7$ nonzeroes) for 1, 4 and 16 threads on a Sandy Bridge.

Matrix layout described in (Martone et al., 2010).

Multi-threaded *SpMV* (1)

$$A = \sum_i A_i$$

$$y \leftarrow y + \sum_i A_i \times x_i, \text{ with leaf submatrices } A_i$$



Multi-threaded *SpMV* (2)

$$y \leftarrow y + \sum_i A_i \times x_i$$

Threads $t \in \{1..T\}$ execute concurrently:

$$y_{i_t} \leftarrow y_{i_t} + A_{i_t} \times x_{i_t}$$

we prevent *race conditions* performing busy wait[3]; we use

- per-submatrix visit information
- per-thread current submatrix information

to lock each y_{i_t} and avoid visiting submatrices twice.

The symmetric variant locks two intervals of y, corresponding to A_i and its transpose.

---

[3] To be improved!

Pros/Cons of RSB's operations in `librsb`

- + parallel *SpMV* /*SpMV-T* /*SymSpMV*
- + parallel *SpSV* /*SpSV-T* (though less scalable than *SpMV*)
- + many other common operations (e.g.: parallel matrix build algorithm)
- - a number of known cases (e.g.: unbalanced matrices) where parallelism is poor
- - some algorithms easy to express/implement for CSR are more complex for RSB

RSB vs MKL's CSR on Intel Sandy Bridge (presented at PMAA'12)

Experimental time efficiency comparison of our RSB prototype to the proprietary, highly optimized Intel's *Math Kernels Library* (MKL r.10.3-7) sparse matrix routines (`mkl_dcsrmv` – `double` precision case).

We report here results on a double *"Intel Xeon E5-2680 0 @ 2.70GHz"* (2×8 cores) and publicly available large ($> 10^7$ nonzeroes) matrices[4].

We compiled our code with the *"Intel C 64 Compiler XE, Version 12.1.1.256 Build 20111011"* using `CFLAGS="-O3 -xAVX -fPIC -openmp"` flags.

---

[4] See next slide for a list.

Matrices

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: Matrices used for our experiments: General (G), Symmetric (S).

Comparison to MKL, Unsymmetric *SpMV* /*SpMV-T* (presented at PMAA'12)

Summarizing:[5]

- untransposed *SpMV* even 60 % faster than MKL's CSR
- *SpMV-T* even 4 times faster (on *GL7d19*: here MKL does not scale)
- some matrices (e.g.: the *tall relat9*) are problematic
- *SpMV* and *SpMV-T* have almost same performance (unlike row or column biased formats)
- scales better than MKL

---

[5] Plots in extra slide: performance in Fig. 6, scalability in Fig. 7.

Comparison to MKL, *SymSpMV* (presented at PMAA'12)

Summarizing:[6]

- speedups up to around 200% in several cases; most exceeding 50%
- scales slightly less than MKL

---

[6] Plots in extra slide: performance in Fig. 8, scalability in Fig. 9.

Performance observations (presented at PMAA'12)

Summarizing:

- no architecture specific optimization employed
- *SpMV* /*SpMV-T* equally parallel
- *SpMV-T* /*SymSpMV* much faster than CSR
- parallel assembly (*CooToRSB*)
- 20-50 iterations of *SpMV-T* /*SymSpMV* to amortize conversion cost and start saving time w.r.t. MKL's `mkl_dcsrmv`[7]

---

[7] Relative conversion times plots in extra slide: Fig. 10 for general matrices; Fig. 11 for symmetric matrices. Relative amortization times plots: Fig. 12 for general matrices; Fig. 13 for symmetric matrices.

An RSB *matrix instance* may be non optimal

- memory bandwidth may saturate using part of available threads
- an excessively coarse partitioning limits parallelism (contention in accessing the result vector!)

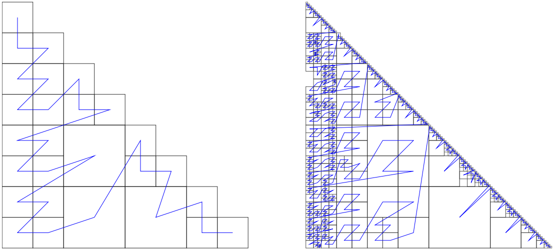


Figure: Left: max parallelism is 8; right, more parallelism, but also more indexing overhead.

Run-time Empirical Autotuning

Tuning algorithm or data structures **at run time** using **benchmarking**.

- notable *sparse* precursor: OSKI (Vuduc et al., 2005)
  - exposes tuning cost, support self-profiling
  - e.g.: best BCSR kernel(s)/decomposition choice
- *sparse*, planned: for CSB (sketched in Buluç et al., 2011)
  - tuning the operation data structures (not the matrix itself)
  - e.g.: multiple temporary result vectors

Simple Autotuning in RSB

The user specifies:[8]

- matrix and *SpMV* parameters (transposition, stride, number of right hand sides)
- optionally, time to spend in *SpMV* and initial thread count

Starting with 1 thread, repeat until no more improvement:

- thread count is increased linearly $(+1)$
- the RSB matrix is re-partitioned as we had $(1/4, 1/2, \times 1, \times 2, \times 4)$ the current cache
- each (matrix instance,threads) combination performance is measured
- the best improving combination is kept

---

[8] To be released soon.

SpMV auto-tuning

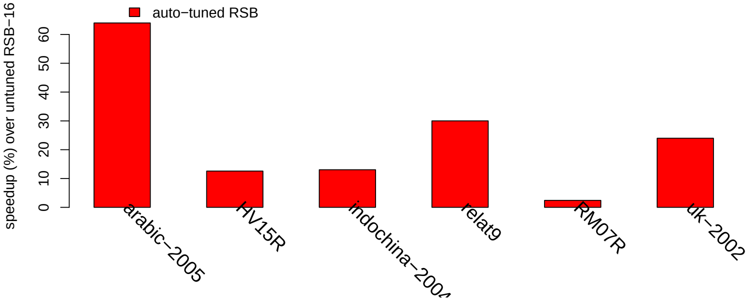


Figure: *SpMV* performance on Sandy Bridge, improvement over untuned 16 threaded RSB, general matrices.

- **arabic-2005:** 16 → 13 **threads,** 9088 → 2427 **leaves,** 2.64 **idx.bytes/nnz**
- *HV15R*: 16 → 11 threads, 2366 → 795 leaves, 2.6 idx.bytes/nnz
- *indochina-2004*: 16 → 13 threads, 2423 → 626 leaves, 2.43 → 2.40 idx.bytes/nnz
- **relat9:** 16 → 8 **threads,** 254 → 70 **leaves,** 7.9 → 7.0 **idx.bytes/nnz**
- *RM07R*: 16 → 9 threads, 430 leaves, 2.46 idx.bytes/nnz
- *uk-2002*: 16 → 11 threads, 4504 → 2233 leaves, 2.59 → 2.57 idx.bytes/nnz

SpMV-T auto-tuning

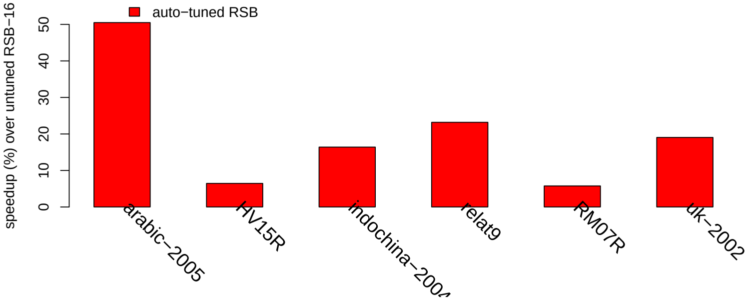


Figure: *SpMV-T* performance on Sandy Bridge, improvement over untuned 16 threaded RSB, general matrices.

- *arabic-2005*: 16 → 9 threads, 2427 leaves, 2.64 idx.bytes/nnz
- *HV15R*: 16 → 12 threads, 2366 → 795 leaves, 2.6 idx.bytes/nnz
- *indochina-2004*: 16 → 15 threads, 2423 → 626 leaves, 2.43 → 2.40 idx.bytes/nnz
- **relat9:** 16 → 11 **threads,** 254 → 1078 **leaves,** 7.9 → 7.0 **idx.bytes/nnz**
- **RM07R:** 16 → 11 **threads,** 430 → 100 **leaves,** 2.46 → 2.59 **idx.bytes/nnz**
- *uk-2002*: 16 → 9 threads, 4504 → 2233 leaves, 2.59 → 2.57 idx.bytes/nnz

SymSpMV auto-tuning

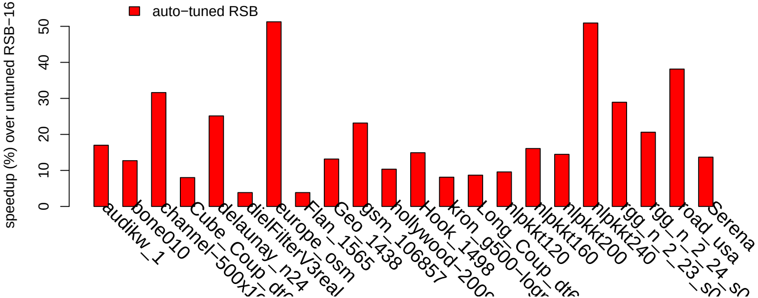


Figure: *SymSpMV* performance on Sandy Bridge, improvement over untuned 16 threaded RSB, symmetric matrices.

- **europe_osm:** 16 → 6 **threads,** 532 → 2638 **leaves,** 8.0 → 4.5 **idx.bytes/nnz**
- nlpkkt240: 16 → 8 threads, 450 → 545 leaves, 3.17 → 3.10 idx.bytes/nnz
- **road_usa:** 16 → 10 **threads,** 237 → 2211 **leaves,** 8.0 → 5.48 **idx.bytes/nnz**
- ...

Auto-tuning: observations

- very sparse and symmetric matrices may need additional subdivisions
- *SpMV* /*SpMV-T* tuning strategies can differ significantly for non square matrices
- current strategy costs thousands of *SpMV* 's

Conclusions

- on large matrices, `librsb` competes with Intel's highly optimized, proprietary CSR implementation
- many aspects can be improved; e.g.:
  - overcome busy wait based locking
  - optimize auto-tuning
  - ...
- a revision of Sparse BLAS to handle e.g.: auto-tuning semantics would be useful

References

Sparse BLAS:

- Iain S. Duff and Christof Vömel. *Algorithm 818: A reference model implementation of the Sparse BLAS in Fortran 95* In ACM Trans. on Math. Softw., n. 2, vol. 28, pages 268–283, ACM, 2002

`librsb`:

- `http://sourceforge.net/projects/librsb`
- Michele Martone, Salvatore Filippone, Salvatore Tucci, Marcin Paprzycki, and Maria Ganzha. *Utilizing recursive storage in sparse matrix-vector multiplication - preliminary considerations.* In Thomas Philips, editor, CATA, pages 300-305. ISCA, 2010.

Questions / discussion welcome!

Thanks for your attention.

Please consider testing `librsb`: spotting bugs/inefficiencies is essential for free software!

Comparison to MKL, *SpMV* & *SpMV-T*

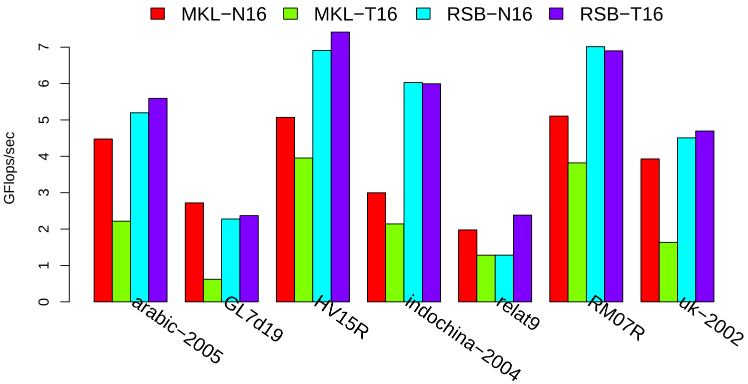


Figure: **T**ransposed/**N**on transposed *SpMV* performance on Sandy Bridge, versus MKL's CSR, 16 threads, unsymmetric matrices.

9

---

[9] Back to page 3.

Comparison to MKL, *SpMV* /*SpMV-T* scalability

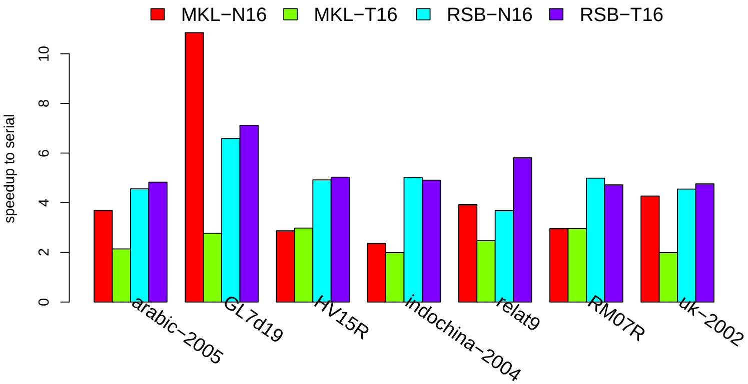


Figure: Unsymmetric matrices. *SpMV* /*SpMV-T* parallel speedup (16 to 1 threads performance ratio).

---

[10] Back to page 3.

Comparison to MKL, *SymSpMV*

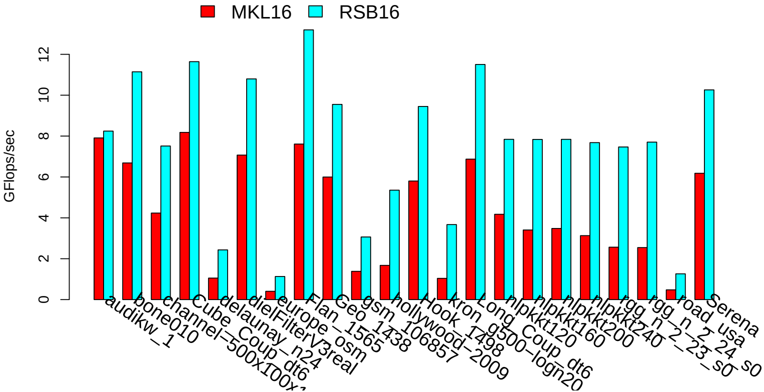


Figure: *SymSpMV* performance on Sandy Bridge, versus MKL's CSR, 16 threads (symmetric matrices).

11

---

[11] Back to page 4.

Comparison to MKL, *SymSpMV* scalability

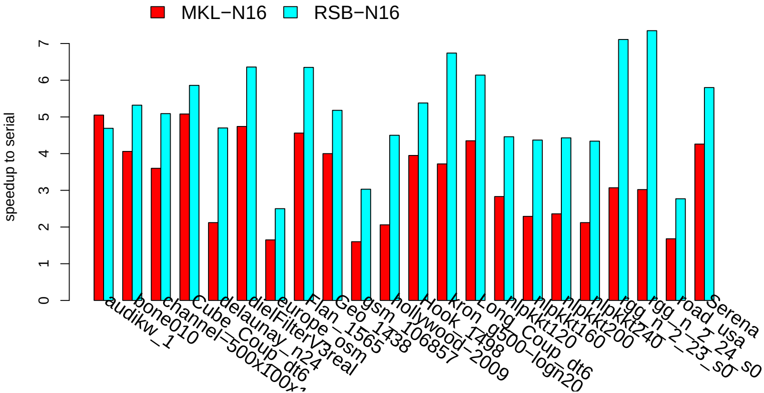


Figure: Symmetric matrices. *SymSpMV* parallel speedup (16 to 1 threads performance ratio).

[12]

---

[12] Back to page 4.

Relative Cost of (row sorted) COO to RSB conversion

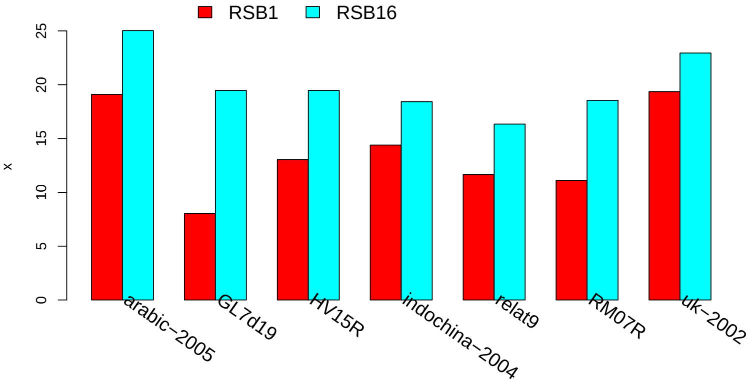


Figure: Non transposed conversion-to-*SpMV* times ratio on Sandy Bridge, unsymmetric matrices, 1 and 16 threads.

13

---

[13] Back to page 1.

Relative Cost of (row sorted) COO to RSB conversion

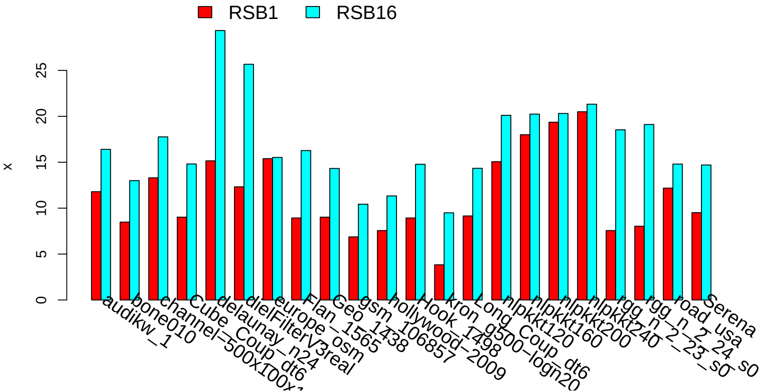


Figure: Non transposed conversion-to-*SpMV* times ratio on Sandy Bridge, symmetric matrices, 1 and 16 threads.

14

---

[14] Back to page 1.

Amortization of conversion cost, *SpMV* ,*SpMV-T*

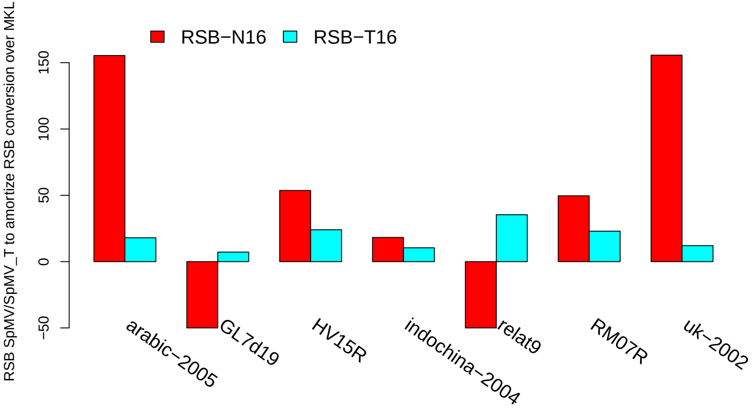


Figure: Unsymmetric matrices. Amount of *SpMV* /*SpMV-T* executions with RSB necessary to amortize time of *CooToRSB*, and get advantage over MKL.16 threads.

15

---

[15] Back to page 1.

Amortization of conversion cost, *SymSpMV*

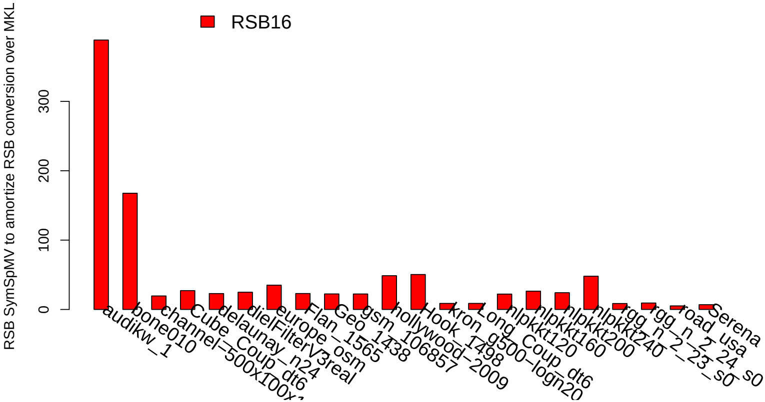


Figure: Symmetric matrices. Amount of *SymSpMV* executions with RSB necessary to amortize time of *CooToRSB*, and get advantage over MKL.16 threads.

16

---

[16] Back to page 1.