An Universal Shared-Memory Parallel Sparse BLAS for C++/Fortran/Octave/Python and a Use Case in LQCD

Michele MARTONE*, **Simone BACCHIO'**, **Jacob FINKENRATH'**, **Luc GIRAUD''**, **Matthieu SIMONIN''**

Leibniz Supercomputing Centre (LRZ)*, The Cyprus Institute', Inria''

Context: Project LyNcs: HPC APIs for LQCD

- project "LyNcs": *"Linear Algebra, Krylov methods, and multi-grid API and library support for the discovery of New Physics"*
- subproject of *PRACE-6IP Grant agreement ID: 823767*, Work Package 8 (*Forward-looking software solutions towards Exascale*), between May 2019 and Dec. 2021
- **L**attice **Q**uantum **C**hromo**d**ynamics (LQCD) is a field of subatomic physics, and accounts for a major fraction of HPC usage
- LyNcs is a collaboration towards Exascale Solvers for LQCD
- synergy of expertises and codes spanning the entire software stack:
  - LQCD, The Cyprus Institute: DDalphaAMG, LyNcs-API
  - numerical linear algebra, Inria Bordeaux (France): Fabulous
  - **portable performance kernels**, LRZ: LIBRSB
- **this poster** introduces the ideas behind LIBRSB, gives an overview of its possible usage modes, mentions its newest developments, and presents performance samples on relevant **LQCD input**

The authors gratefully acknowledge the Gauss Centre for Supercomputing e.V. for funding this project by providing computing time on the GCS Supercomputer SuperMUC-NG at LRZ

Numerical Techniques of Interest to LyNcs

- iterative methods: *block Krylov*
- require efficient Sparse Matrix-Matrix multiplication aka SpMM

SpMM in matrix form (jargon: with m right hand sides — mRHS):

$$\overbrace{\begin{bmatrix} c_{11} & \dots & c_{1m} \\ \vdots & \ddots & \vdots \\ c_{n1} & \dots & c_{nm} \end{bmatrix}}^{updated\ dense\ C} \leftarrow \beta \overbrace{\begin{bmatrix} c_{11} & \dots & c_{1m} \\ \vdots & \ddots & \vdots \\ c_{n1} & \dots & c_{nm} \end{bmatrix}}^{dense\ C} + \alpha \overbrace{\begin{bmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{nn} \end{bmatrix}}^{sparse\ A} \overbrace{\begin{bmatrix} b_{11} & \dots & b_{1m} \\ \vdots & \ddots & \vdots \\ b_{n1} & \dots & b_{nm} \end{bmatrix}}^{dense\ B}$$

More on the methods of our interest:

R. B. Morgan. *Restarted block GMRES with deflation of eigenvalues*, Appl. Numer. Math., 54(2):222–236, 2005. ▪ M. Robbe and M. Sadkane. *Exact and inexact breakdowns in the block GMRES method*, Linear Algebra Appl., 419:265–285, 2006. ▪ E. Agullo, L. Giraud, and Y.-F. Jing. *Block GMRES method with inexact breakdowns and deflated restarting*, SIAM J. Matrix Anal. Appl., 35(4):1625–1651, 2014.

LIBRSB: A Sparse BLAS Library

- *Recursive Sparse Blocks* (RSB) layout (see next section for overview)
- >100KLOC of C99 + OpenMP + generated specialized kernels
- node-level **shared-memory**-parallel operations
  - Sparse BLAS: matrix assembly/destroy, **SpMM**, triangular solve
  - operations for *interactive* applications (e.g. matrix update, sparse-sparse sum/multiplication, conversions)
  - operations for *distributed-memory applications* (block extract, update, etc.)
- dual API:
  - own interface in C/C++ and Fortran
  - Sparse BLAS (BLAS Technical Forum Standard)
- portable **LGPLv3-licensed** *free software*

project page: `http://librsb.sf.net`

Recursive Sparse Blocks (RSB) Layout and Algorithms

- intended for *large* matrices, where it supports
  - \+ cache locality
  - \+ coarse thread parallelism
- supports *online empirical autotuning* (wrt *blocking*)

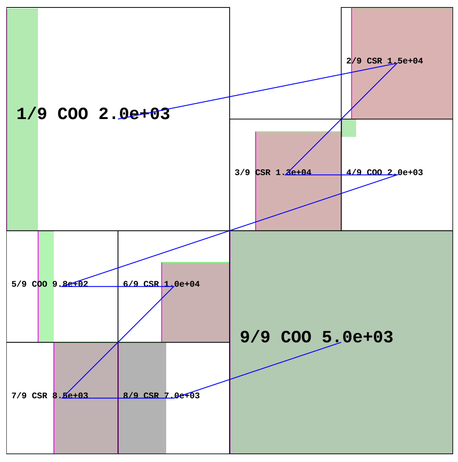

Figure 1: RSB instance of classical test matrix *bayer02* ($14k \times 14k, 64k$ nonzeroes). Black-bordered boxes are *sparse blocks*, and are Z-ordered. Greener have fewer nnz than average, redder have more. Can be either in *"Coordinate"* format (COO) or *"Compressed Sparse Rows"* (CSR). Blocks' rows (LHS) and columns (RHS) ranges evidenced (left and top side).

More on RSB:

M. Martone. Efficient multithreaded untransposed, transposed or symmetric sparse matrix-vector multiplication with the recursive sparse blocks format. Parallel Computing, (40):251–270, 2014. `http://dx.doi.org/10.1016/j.parco.2014.03.008` (open access preprint here)

Matrices of Interest

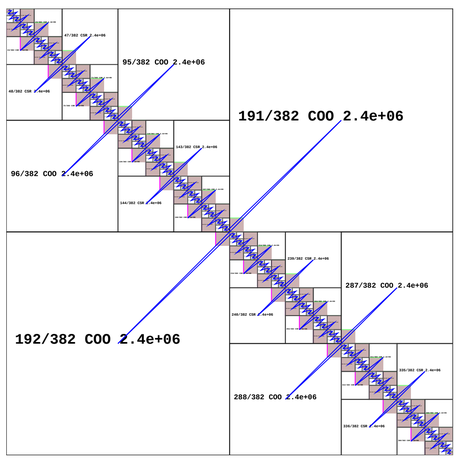

Figure 2: Instance of matrix representing the Wilson Dirac Operator. Used in Lattice QCD to simulate quarks. Square, $12M$ equations, $597M$ nonzeroes, symmetric, complex.

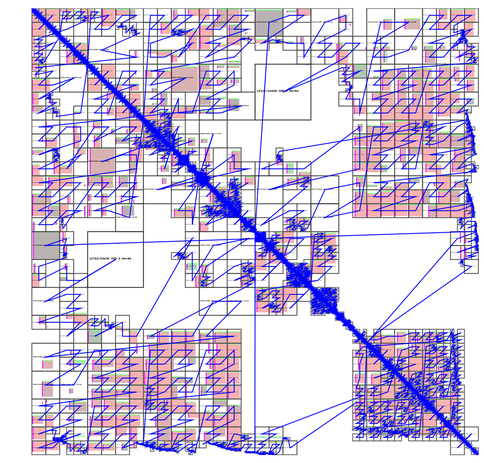

Figure 3: Nanoscale light-matter interactions equations matrix. Square complex symmetric (symmetry expanded here), with $2M$ equations and $281M$ nonzeroes. Here, as 1934 COO and 22479 CSR sparse blocks. This instance occurs during autotuning's *search* for a well-performing blocking on an AMD ROME EPYC 7742, with 16 threads spread across the cores. Courtesy HiePACS and Dr. Stéphane Lanteri, Inria.

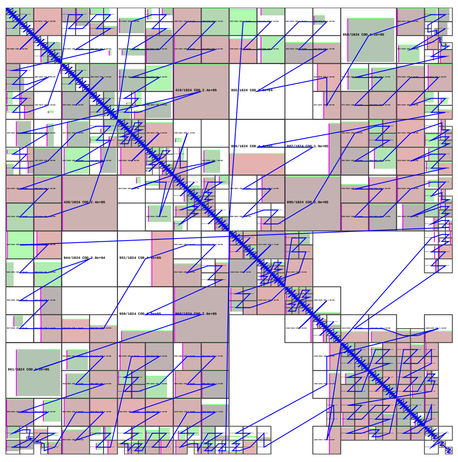

Figure 4: Same matrix. *Best* autotuning-determined layout for SpMM with 2 RHS: 170 COO and 1530 CSR blocks. Using ***sh**ort* indices (therefore **H**COO, **H**CSR): indexing averages to only 2.64 bytes per nnz. Performed SpMM 40% faster than an initial "first guess" structure with 7173 blocks. Autotuning took less than a minute. SpMM operation time slightly (12%) faster than CSR with Intel MKL "20200822".

New in LIBRSB-1.3

- **improved SpMM kernels**
- new *modern C++* interface
- quality improvement of internals (fixes, extensive CI/CD, high-coverage test suite)
- tested on partners' use cases, novel architectures (LRZ and Inria testbeds)

One Library, many Interfaces

- original: `rsb.h`, **new**: `rsb.hpp`
- machine-translated:
  - Fortran *module*, via script
  - Sparse BLAS headers and module, via GNU M4
  - Python: via script + Cython
- GNU Octave: written in C++ using `rsb.h` and `liboctave`

Features access:

- all: C/C++
- most: Fortran
- many: PyRSB, Octave-SparseRSB
- limited: Sparse BLAS API

Performance:

- in C and Fortran, **zero overhead**
- limited but **portable** if using precompiled binaries (e.g. `# apt install librsb0 octave-sparsersb`)
- **better performing** if compiled with an optimizing compiler (e.g. via Spack or Guix-HPC or EasyBuild)

PyRSB vs SciPy CSR and SparseRSB vs Octave CSC:

- likely several times faster (on large matrices)
- reason: SciPy CSR and Octave CSC both serial
- PyRSB and SparseRSB have intrinsic *copy overheads*

For Julia users: interface by D. C. Jones
`https://github.com/dcjones/RecursiveSparseBlocks.jl`

GNU Octave + liboctave + LIBRSB = SparseRSB

- GNU Octave: a MATLAB-like interactive numerical language
- liboctave: access Octave via C++
- SparseRSB: *package* (as Oct-File) to access LIBRSB **transparently**

```
octave:1> R = ( rand (3) > .6 )
R =

   0   0   0
   0   0   0
   1   0   1

octave:2> A_octave = sparse (R)
A_octave =

Compressed Column Sparse (rows = 3, cols = 3, nnz = 2 [22%])

  (3, 1) ->  1
  (3, 3) ->  1

octave:3> A_librsb = sparsersb (R)
A_librsb =

Recursive Sparse Blocks  (rows = 3, cols = 3, nnz = 2 [22%])

  (3, 1) -> 1
  (3, 3) -> 1
```

Figure 5: Package with only one new keyword. Most arithmetical and conversion operators and builtins acting on sparse work on sparsersb object as well.

`http://octave.sourceforge.net/sparsersb/`

Python + Cython + LIBRSB = PyRSB

- SciPy: popular Python scientific computing API
- Cython: *optimising static compiler* for C extensions to Python
- PyRSB: extension module to access LIBRSB **transparently**

```
import numpy
import scipy
from scipy.sparse import csr_matrix
from rsb          import rsb_matrix

V=[ 11.,12.,22.]
I=[   0,  0,  1]
J=[   0,  1,  1]

c = csr_matrix((V, (I, J)))

a = rsb_matrix((V, (I, J)))
a = rsb_matrix((V, (I, J)), [3,3])
a = rsb_matrix((V,  I, J))

y = y + a * x; # equivalent to y = y + c * x
```

Figure 6: `rsb_matrix` usage is styled after SciPY's `csr_matrix`.

`https://github.com/michelemartone/pyrsb`

Symmetric SpMM Performance

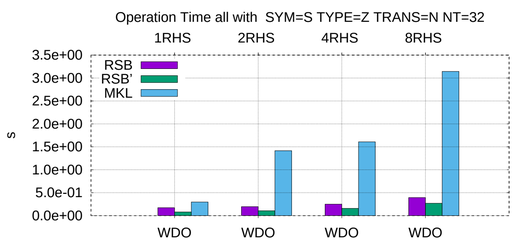

Figure 7: Symmetric SpMM wallclock times.

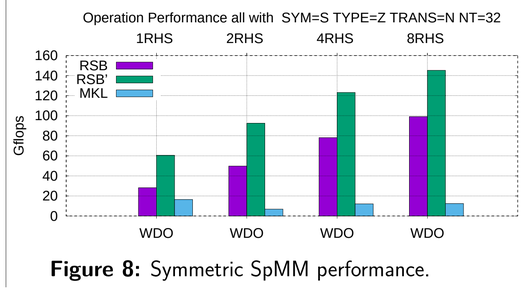

Figure 8: Symmetric SpMM performance.

Unsymmetric SpMM Performance

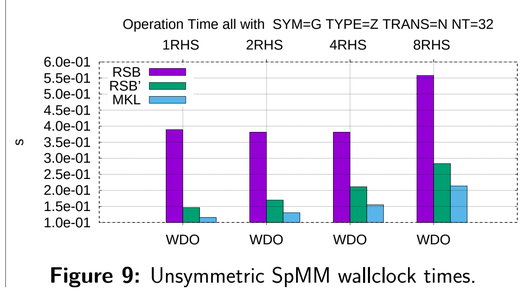

Figure 9: Unsymmetric SpMM wallclock times.

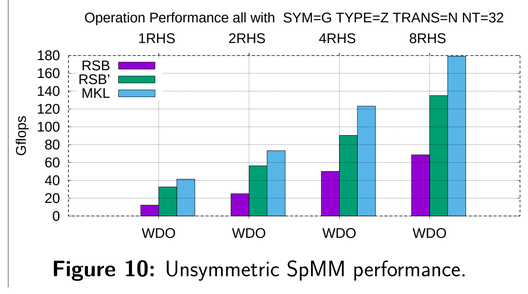

Figure 10: Unsymmetric SpMM performance.

Performance Experiment Setup

We show performance and timing (*wallclock* time) results of LIBRSB's (RSB) with "Intel OneAPI Math Kernel Library Inspector-Executor" CSR (MKL). RSB' results come from calling the blocks-autotuning routine beforehand. We time SpMM of the Wilson-Dirac Operator (WDO) matrix from Fig.3 with respectively 1,2,4, and 8 right hand sides (RHS). With a symmetric matrix as here we have (and exercise) the option to use a symmetry-exploiting SpMM algorithm. We express performance as usual in SpMM literature, by means of "canonical" FLOPS. For our `double complex` matrix (BLAS type Z) this is nonzeroes count (5.97e+08) times 8 divided by time. In symmetric SpMM one only uses the lower matrix triangle – each non-diagonal coefficient is being utilized for the *transposed update*, too – so we count the triangle's nonzeroes twice. We use 32 cores of an Intel Skylake "Xeon Platinum 8174".

Performance Observations

RHS	MKL	RSB'	$\frac{MKL}{RSB'}$	MKL	RSB'	$\frac{MKL}{RSB'}$	$\frac{MKL_{best}}{RSB'_{best}}$
1	2.970e-01	**8.062e-02**	3.684	1.156e-01	1.466e-01	0.789	**1.434**
2	1.415e+00	**1.056e-01**	13.40	1.305e-01	1.699e-01	0.768	**1.236**
4	1.609e+00	1.584e-01	10.16	**1.550e-01**	2.113e-01	0.734	0.979
8	3.144e+00	2.693e-01	11.67	**2.139e-01**	2.835e-01	0.755	0.794

Figure 11: In addition to the figures, this table shows **timings** (in seconds) of respectively **symmetric (left)** and **unsymmetric (right)** SpMM. Non-autotuned RSB results are omitted. Best for each is shown in **bold**. The rightmost column compares ratio of best MKL to best RSB' (is *speedup* of RSB' wrt MKL).

Symmetric RSB' performed 3–13.4× as fast as MKL, depending on the right-hand-sides count. Looking at **unsymmetric** SpMM timings, and picking up best times overall reveals that for 1-2 RHS, RSB' performed faster by 23–43%. But with 4 RHS they do similarly (within 2%), and for 8 RHS, MKL prevails by ca. 25%. If not a conclusive result, this reminds of the usefulness of having compatible, replaceable performance libraries. Other observations worth note: **a**) Compute work rises linearly with the RHS count, but times rise less than that: this testifies that the new low-level SpMM kernels (not present in version 1.2) used here exhibit data level reuse (please also note the involved data size exceeds involved caches by far); **b**) One can notice a relevant gap between untuned (RSB) and tuned (RSB') performance: this shows that an adequate trade-off between block size, parallelism granularity (each block is processed by a thread) is very important for it – "optimal" blocking (not necessarily achieved here) will differ based on RHS count, numerical type, run-time affinity, etc.. The value of LIBRSB lies in its competitiveness despite using **no machine-specific** code. Improving its autotuning mechanism is future work.