PyRSB: Portable Performance on Multithreaded Sparse BLAS Operations

Michele Martone[‡*], Simone Bacchio[§]

Abstract—This article introduces **PyRSB**, a Python interface to the **LIBRSB** library. LIBRSB is a portable *performance library* offering so called *Sparse BLAS* (Sparse Basic Linear Algebra Subprograms) operations for modern multicore CPUs. It is based on the *Recursive Sparse Blocks* (**RSB**) format, which is particularly well suited for matrices of large dimensions. PyRSB allows LIBRSB usage with an interface styled after that of **SciPy**'s *sparse matrix* classes, and offers the extra benefit of exploiting multicore parallelism. This article introduces the concepts behind the RSB format and LIBRSB, and illustrates usage of PyRSB. It concludes with a user-oriented overview of speedup advantage of `rsb_matrix` over `scipy.sparse.csr_matrix` running general sparse matrix-matrix multiplication on a modern shared-memory computer.

Introduction

Sparse linear systems solving is one of the most widespread problems in numerical scientific computing. The key to timely solution of sparse linear systems by means of iterative methods resides in fast multiplication of sparse matrices by dense matrices. More precisely, we mean the update: $C \leftarrow C + \alpha AB$ (at the element level, equivalent to $C_{i,k} \leftarrow C_{i,k} + \alpha A_{i,j} B_{j,k}$) where B and C are dense rectangular matrices, A is a *sparse* rectangular matrix, and *alpha* a scalar. If B and C are vectors (i.e. have one column only) we call this operation *SpMV* (short for *Sparse Matrix-Vector product*); otherwise *SpMM* (short for *Sparse Matrix-Matrix product*).

PyRSB [PYRSB] is a package suited for problems where: i) much of the time is spent in SpMV or SpMM, ii) one wants to exploit multicore hardware, and iii) sparse matrices are large (i.e. occupy a significant fraction of a computer's memory).

The PyRSB interface is styled after that of the sparse matrix classes in SciPy [Virtanen20]. Unlike certain similarly scoped projects ([Abbasi18], [PyDataSparse]), PyRSB is restricted to 2-dimensional matrices only.

Background: LIBRSB

LIBRSB [LIBRSB] is a LGPLv3-licensed library written primarily to speed up solution of large sparse linear systems using iterative methods on shared-memory CPUs. It takes its name from the Recursive Sparse Blocks (RSB) data layout it uses. The RSB format is geared to execute multithreaded SpMV and SpMM as fast as possible. LIBRSB is not a solver library, but provides most of the functionality required to build one. It is usable via several languages: C, C++, Fortran, GNU Octave [SPARSERSB], and now Python, too. Bindings for the Julia language have been authored by D.C. Jones [RSB_JL].

LIBRSB has been reportedly used for: Plasma physics [Stegmeir15], sub-atomic physics [Klos18], data classification [Lee15], eigenvalue computations [Wu16], meteorology [Browne15T], and data assimilation [Browne15M].

It is available in pre-compiled form in popular GNU/Linux distributions like Ubuntu [UBUNTU], Debian [DEBIAN], OpenSUSE [OPENSUSE]; this is the best way have a LIBRSB installation to familiarize with PyRSB. However, pre-compiled packages will likely miss compile-time optimizations. For this reason, the best performance will be obtained by bulding on the target computer. This can be achieved using one of the several source-based code distributions offering LIBRSB, like Spack [SPACK], or EasyBuild [EASYBUILD], or GUIX [GUIX]. LIBRSB has minimal dependencies, so even bulding by hand is trivial.

PyRSB [PYRSB] is a thin wrapper around LIBRSB based on Cython [Behnel11]. It aims at bringing native LIBRSB performance and most of its functionality at minimal overhead.

Basic Sparse Matrix Formats

The explicit (**dense**) way to represent any numerical matrix is to list each of its numerical entries, whatever their value. This can be done in Python using e.g. `scipy.matrix`.

```
>>> from scipy import matrix
>>>
>>> A = matrix([[11., 12.], [ 0., 22.]])
matrix([[11., 12.],
        [ 0., 22.]])
>>> A.shape
(2, 2)
```

This matrix has two rows and two columns; it contains three non-zero elements and one zero element in the second row. Many scientific problems give rise to systems of linear equations with many (e.g. millions) of unknowns, but relatively few coefficients which are different than zero (e.g. *<1%*) in their matrix-form representation. It is usually the case that representing these zeroes in memory and processing them in linear algebraic operations does not impact the results, but takes *compute time* nevertheless. In these cases the matrix is usually referred as **sparse**, and appropriate **sparse data structures** and algorithms are sought.

The most straightforward sparse data structure for a numeric matrix is one listing each of the non-zero elements, along with its *coordinate* location, by means of three arrays. This is called **COO**.

* Corresponding author: michele.martone@lrz.de
‡ Leibniz Supercomputing Centre (LRZ), Garching near Munich, Germany
§ CaSToRC, The Cyprus Institute, Nicosia, Cyprus

It's one of the classes in `scipy.sparse`; see the following listing, whose output also illustrates conversion to dense:

```
>>> from scipy.sparse import coo_matrix
>>>
>>> V = [11.0, 12.0, 22.0]
>>> I = [0, 0, 1]
>>> J = [0, 1, 1]
>>> A = coo_matrix((V, (I, J)))
<2x2 sparse matrix of type '<class 'numpy.float64'>'
     with 3 stored elements in COOrdinate format>
>>> B = A.todense()
>>> B
matrix([[11., 12.],
        [ 0., 22.]])
>>> A.shape
(2, 2)
```

Even if yielding the same results, the algorithms beneath differ considerably. To carry out the $C_{i,k} \leftarrow C_{i,k} + \alpha A_{i,j} B_{j,k}$ updates the `scipy.coo_matrix` implementation will get the matrix coefficients from the `V` array, its coordinates from the `I` and `J` arrays, and use those (notice the **indirect access**) to address the operand's elements.

In contrast to that, a dense implementation like `scipy.matrix` does not use any index array: the location of each numerical value (including zeroes) is in direct relation with its row and column indices.

Beyond the `V,I,J` arrays, COO has no extra structure. COO serves well as an exchange format, and allows expressing many operations.

The second most straightforward format is CSR (Compressed Sparse Rows). In CSR, non-zero matrix elements and their column indices are laid consecutively row after row, in the respective arrays `V` and `J`. Differently than in COO, the row index information is compressed in a *row pointers* array `P`, dimensioned one plus rows count. For each row index `i`, `P[i]` is the count of non-zero elements (*nonzeroes*) on preceding rows. The count of nonzeroes at each row `i` is therefore `P[i+1]-P[i]`, with `P[0]==0`. SciPy offers CSR matrices via `scipy.csr_matrix`:

```
>>> import scipy
>>> from scipy.sparse import csr_matrix
>>>
>>> V = [11.0, 12.0, 22.0]
>>> P = [0, 2, 3]
>>> J = [0, 1, 1]
>>> A = csr_matrix((V, J, P))
>>> A.todense()
matrix([[11., 12.],
        [ 0., 22.]])
>>> A.shape
(2, 2)
```

CSR's `P` array allows direct access of each *sparse row*. This helps in expressing row-oriented operations. In the case of the SpMV operation, CSR encourages accumulation of partial results on a per-row basis.

Notice that indices' occupation with COO is strictly proportional to the non-zeroes count of a matrix; in the case of CSR, only the `J` indices array. Consequently, a matrix with more nonzeroes than rows (as usual for most problems) will use less index space if represented by CSR. But in the case of a particularly sparse block of such a matrix, that may not be necessarily true. These considerations back the usage choice of COO and CSR within the RSB layout, described in the following section.

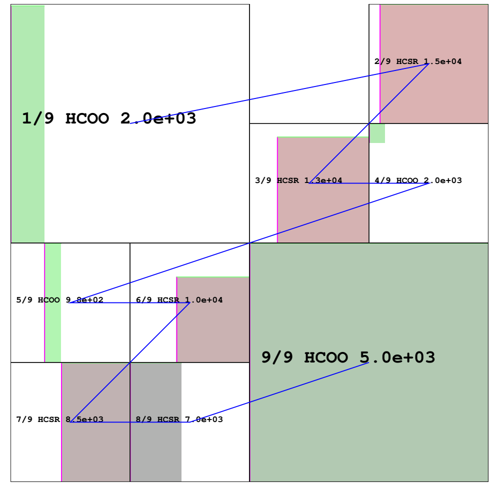

Fig. 1: *Rendering of an RSB instance of classical matrix* `bayer02` *(sized $14k \times 14k$ with $64k$ nonzeroes, from the SuiteSparse Matrix Collection [SSMC]); each sparse block is labeled with its own format (the 'H' prefix indicating use of a shorter integer type); each block's effectively non-empty rectangle is shown, in colour; greener blocks have fewer nonzoeroes than average; rosier ones have more. Blocks' rows and columns ranges are highlighted (respectively magenta and green) on the blocks' sides. Note that larger blocks (like* `"9/9"`*) may have fewer nonzeroes than smaller ones (like* `"4/9"`*).*

From RSB to PyRSB

Recursive Sparse Blocks in a Nutshell

The Recursive Sparse Blocks (RSB) format in LIBRSB [Martone14] represents sparse matrices by exploiting a hierarchical data structure. The matrix is recursively subdivided in halves until the individual submatrices (also: *sparse blocks* or simply *blocks*) occupy approximately the amount of memory contained in the CPU caches. Each submatrix is then assigned the most appropriate format: COO if very sparse, CSR otherwise.

Any operation on an RSB matrix is effectively a *polyalgorithm*, i.e. each block's contribution will use an algorithm specific to its format, and the intermediate results will be combined. For a more detailed description, please consult [Martone14] and further references from there.

The above details are useful to understand, but not necessary to use PyRSB. To create an `rsb_matrix` object one proceeds just as with e.g. `coo_matrix`:

```
>>> from pyrsb import rsb_matrix
>>>
>>> V = [11.0, 12.0, 22.0]
>>> I = [0, 0, 1]
>>> J = [0, 1, 1]
>>> A = rsb_matrix((V, (I, J)))
>>> A.todense()
matrix([[11., 12.],
        [ 0., 22.]])
>>> A.shape
(2, 2)
```

Direct conversion from `scipy.sparse` classes is also supported. Instancing an RSB structure is computationally more demanding than with COO or CSR (in both memory and time). Exploiting multiple cores and the savings from faster SpMM's shall make the extra construction time negligible.

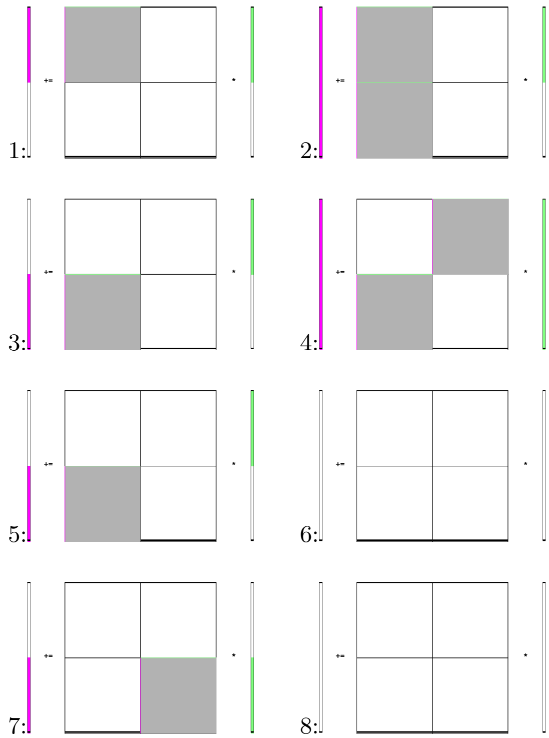

Fig. 2: *SpMV goes through steps leading to the following states: 1) upper left block becomes active; 2) lower left block becomes active; 3) upper left block is done (not active anymore); 4) upper right block becomes active; 5) upper right block is done; 6) lower left block is done; 7) lower right block is now active; 8) lower right block is done.*

Multi-threaded Sparse Matrix-Vector Multiplication with RSB

The following sequence of pictures schematizes eight states of a two-threaded SpMV on an RSB matrix consisting of four (non-empty sparse) blocks. At any moment, up to two blocks are being object of concurrent SpMV (*active*). Here each active block has a gray background; its rows and column ranges are highlighted. Left of the matrix, a (out-of-horizontal-scale) result vector is depicted. For each of the active blocks, the corresponding *active range* (corresponding to the rows) is highlighted on the vector. Similarly, right of the matrix, the (out-of-horizontal-scale) operand vector is shown; its active ranges (corresponding to each blocks' column range) are highlighted.

The idea behind the algorithm is that a thread won't write to a portion of the result array which is currently being updated by another thread. Beyond that, there is no further synchronization of threads.

This algorithm applies to square as well as non-square matrices. It supports transposed operation (in which case the ranges of each block are swapped). Symmetric operation is supported, too; in this case, an additional *transposed* contribution is considered for each block.

As depicted in the first RSB illustration (Fig. 1), the order of the sparse blocks in memory proceeds along a *space-filling curve*. That order of processing the individual blocks can help to deliver data from the memory to the cores faster. For this reason the individual cores attempt to follow that order whenever possible.

To have enough work for each thread, RSB arranges to have more blocks than threads. For this and other trade-offs involved, as well for a formal description of the multiplication algorithm, see [Martone14] and further literature about RSB listed there.

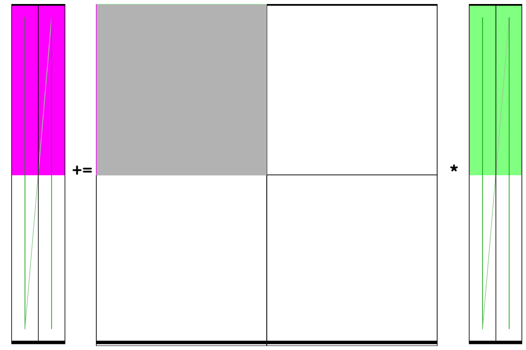

Fig. 3: *A Matrix and its SpMM operands, in **columns-major** order. Matrix consisting of four sparse blocks, of which one highlighted. Left hand side and right hand side operands consist of two vectors each. These are stored one column after the other (memory follows blue line). Consequently, the two column portions operands pertaining a given sparse block are not contiguous.*

The SpMV algorithm sketched above is what happens *under the hood* in PyRSB. In practice, `rsb_matrix` is used in SpMV just as with `scipy.sparse` classes seen earlier:

```
>>> from numpy import ones
>>> B = ones([2], dtype=A.dtype)
>>> C = A * B
```

Multi-threaded Sparse Matrix-Matrix Multiplication with RSB

With multiple column operands (in jargon, *multiple right hand sides*), the operation result is equivalent to that of performing correspondingly many SpMVs.

In these cases it comes naturally to lay the columns one after the other (consecutively) in memory, and have the resulting *rectangular dense matrix* as operand to the SpMM. Also here the same notation of the previous section is supported; see this example with 2 right hand sides:

```
>>> from numpy import ones
>>> B = ones([2,2], dtype=A.dtype)
>>> C = A * B
```

Let's look at how to deal with this when using the RSB layout. As anticipated, the individual right hand sides may lay after each other, as columns of a rectangular dense matrix. See Fig. 3, where a broken line follows the two operands' layout in memory, also *by columns*.

A straightforward SpMM implementation may run two individual SpMV over the entire matrix, one column at a time. That would have the entire matrix (with all its blocks) being read once per column.

A first RSB-specific optimization would be to run all the per-column SpMVs at a block level. That is, given a block, repeat the SpMVs over all corresponding column portions. This would increase chance of reusing cached matrix elements as the operands are visited. This reuse mechanism is being exploited by LIBRSB-1.2. The *by columns* layout (or *order*) is the recommended one for SpMM there.

The most convenient thing though, would be to read the entire matrix only once. That is the case for LIBRSB-1.3 (scheduled for release in summer 2021): for small column counts, block-level

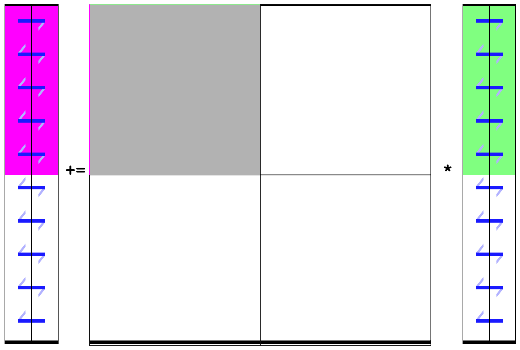

***Fig. 4:** A Matrix and its SpMM operands, in **rows-major order**. Matrix consisting of four sparse blocks, of which one highlighted. Left hand side and right hand side operands consist of two vectors each, interspersed (memory follows blue line). Consequently, the two column portions operands pertaining a given sparse blocks are contiguous.*

SpMM goes through all the columns while reading a block exactly once.

The aforementioned SpMM algorithm is to be regarded as LIBRSB-specific internals, with not much user-level control over it.

But there is another factor instead, that plays a certain role in the efficiency of SpMM, where the PyRSB user has a choice: the layout of the SpMM operands.

SpMM with different Operands Layout

The **by-columns** layout described earlier and shown in Fig. 3 appears to be the most natural one if one thinks of the columns as laid in successive **multiple arrays**. However, one may instead opt to choose a **by-rows** layout instead, shown in figure 4.

A by-rows layout can be thought as interspersing all the columns, one index at a time. Here in the figure, the blue line follows their **order in memory**. At SpMM time, given one of the input columns, an element at a given index is multiplied by nonzeroes located at that column index. Similarly, given one of the output columns, an element at a given index receives a contribution from the nonzeroes located at that row coordinate. With a by-rows layout of the operands, SpMM may proceed by reading a nonzero once, read all right hand sides at that row index (they are adjacent), and then update the corresponding left hand sides' elements (which are also adjacent). On current cache- and register-based CPUs, the locality induced by this layout leads often to a slightly faster operation than with a by-columns layout.

The by-columns and by-rows layouts go by the respective names of Fortran (`'F'`) and C (`'C'`) order. A user can choose which dense layout to use when creating operands for SpMM. Their physical layouts differ, but NumPy makes their results are interoperable; see e.g.:

```
>>> import scipy, numpy, rsb
>>>
>>> size = 1000
>>> density = 0.01
>>> nrhs = 10
>>>
>>> A = scipy.sparse.random(size, size, density)
>>> A = rsb.rsb_matrix(A)
>>>
>>> B = numpy.random.rand(size, nrhs)
>>>
>>> B_c = numpy.ascontiguousarray(B)
>>> B_f = numpy.asfortranarray(B)
>>>
>>> assert B.flags.c_contiguous
>>> assert B_c.flags.c_contiguous
>>> assert B_f.flags.f_contiguous
>>>
>>> C = A * B
>>> C_c = A * B_c
>>> C_f = A * B_f
```

While both layouts are supported, the `'C'` layout is the recommended one for SpMM operands when using PyRSB with LIBRSB-1.3. Also notice that SpMV is a special case of SpMM with one left-hand side and one right-hand side, so the two layouts are equivalent here. In the following, we will often refer to **right-hand sides count** as by **NRHS**.

Using PyRSB: Environment Setup and Autotuning

Usage of PyRSB requires no knowledge beyond its documentation. However, the underlying LIBRSB library can be configured in a variety of ways, and this affects PyRSB. To begin using PyRSB, a distribution-provided installation shall suffice. To expect best performance results, a *native* LIBRSB build is recommended. The next section comments some basic facts to control LIBRSB and make the most out of PyRSB.

Environment Variables

PyRSB does not use any environment variable directly; it is affected via underlying LIBRSB and Python. By default, LIBRSB it is built with shared-memory parallelism enabled via OpenMP [OPENMP]. As a consequence, a few dozen OpenMP environment variables (all prefixed by `OMP_`) apply to LIBRSB as well. Of these, the most important is the one setting the active threads count: `OMP_NUM_THREADS`. Administrators of HPC (High Performance Computing) systems customarily set this variable to recommended values. Even if unset, chances are good the OpenMP runtime will guess the right value for this. Most other OpenMP variables will be of less use to PyRSB, except one: setting `OMP_DISPLAY_ENV=TRUE` will get current defaults printed at program start (very useful when debugging a configuration).

In addition to the above, there are environment variables affecting specifically LIBRSB. All of those are prefixed by `RSB_`, so to avoid any clash. One recommended to end users is `RSB_USER_SET_MEM_HIERARCHY_INFO`, and is used to override cache hierarchy information detected at runtime or *hard-coded* at build time. Essentially, one can use it to force a finer or coarser blocking. For its usage, and for verification of further LIBRSB defaults, please see its documentation (accessible from [LIBRSB]). Modifying the variables mentioned in this section will be mostly useful on very new or not fully configured systems, or for tuning a bit over the defaults.

RSB Autotuning Procedure for SpMM

Cores count, cache sizes, operands data layout, and matrix structure all play a role in RSB performance. The default blocks layout chosen when assembling an RSB instance may not be the most efficient for the particular SpMM to follow. In practice, given an RSB instance and an SpMM context (vector and scalar operands info, transposition parameter, run-time threads count), it may be the case that a better-performing layout can be found by exploring slightly *coarser* or *finer* blockings, An automated (*autotuning*)

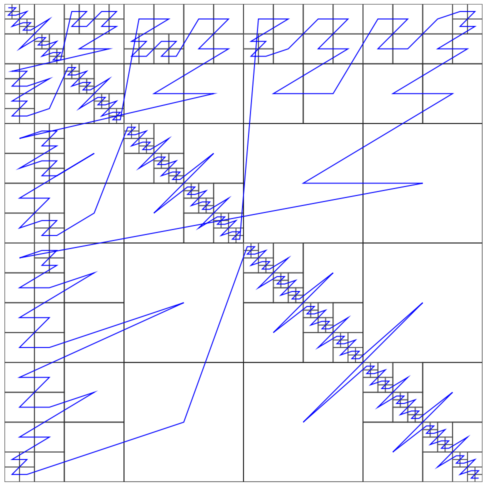

***Fig. 5:** Rendering of an RSB instance matrix `audikw_1` (for this and other matrices, see table) as `dtype=numpy.float32` (or S) after `autotune(order='C',nrhs=1)` on our setup. Autotuning merged an initial 766 blocks guess into 295, bringing a 1.56× speedup to `rsb_matrix` SpMV time. With `rsb_matrix` it now takes 1/34th of (1-threaded) `csr_matrix` time; before autotuning, it took 1/22th. Autotuning itself took the time of 1.5 `csr_matrix` SpMV iterations, or 34 pre-autotuning `rsb_matrix` SpMV iterations.*

procedure for this exists and is accessible via `autotune`. The following example shows how to use it on matrix `audikw_1` from [SSMC].

```
>>> import sys, rsb, numpy
>>> dtype=numpy.float32
>>>
>>> A = rsb.rsb_matrix("audikw_1.mtx",dtype=dtype)
>>> print(A) # original blocking printed out
>>> sf = A.autotune(verbose=False)
>>> print("autotune speedup for SpMV  : %.2e x" %sf )
>>> print(A) # updated blocking printed out
>>>
>>> A = rsb.rsb_matrix("audikw_1.mtx",dtype=dtype)
>>> print(A) # original blocking printed out
>>> sf = A.autotune(verbose=False, transA='N',
>>>         order='C', nrhs=8)
>>> print("autotune speedup for SpMM-8: %.2e x" %sf )
>>> print(A) # updated blocking printed out
```

In scenarios where SpMM is to be iterated many times, time spent autotuning an instance shall amortize over the now faster iterations. See the comments of instances of autotuning on Fig. 5, Fig. 6. and Fig. 7 for realistic use cases.

The reader impatient to see further speedup figures achievable by `autotune` can already peek at Fig. 10.

Experiments with SpMM and Autotuning

Purpose of this section is to present **statistics of speedups** one may encounter by using PyRSB instead of SciPy CSR in practical usage. In our choice of experiments, and in the exposition, we favour **breadth** over depth. So **differently than in a paper with HPC in focus**, we focus on the achievable speedup, and not on performance. We also take **shortcuts** which we would not take otherwise, like mixing statistics from *single precision* computations with *double precision* ones, or real-valued and complex-valued ones. Also the very focus of the article, namely comparing directly **threaded RSB to serial CSR** in SciPy would be ill-posed, were we interested to compare the parallelism grade of the two implementations. On the plots that will follow, samples

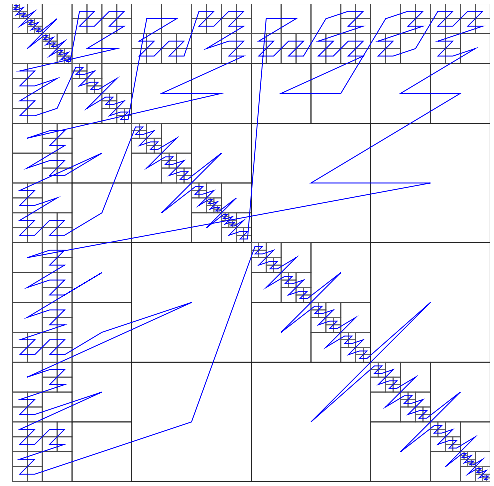

***Fig. 6:** Same matrix as Fig. 5, but autotuned with `nrhs=2`. Here the initial 766 blocks have been merged into 406, with 1.14× speedup. Before autotuning, it took 1/22th of a (1-threaded) `csr_matrix` time; now it's 1/31th. Here too, it took the time of 1.5 `csr_matrix` SpMM iterations, or 34 with the pre-autotuning `rsb_matrix` instance.*

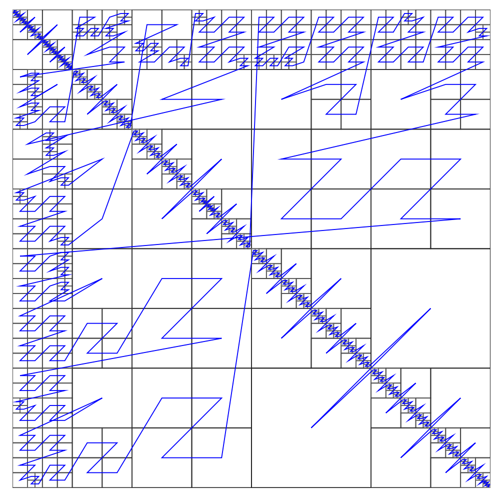

***Fig. 7:** Differently than with `nrhs=1` or `nrhs=2`, `autotune(nrhs=8)` did not find a better blocking than the original 766 blocks. Still, the procedure costed the time of 11 `csr_matrix` SpMM's, or 234 `rsb_matrix` ones. Though not autotuned, (threaded) RSB takes merely 1/22th the time of CSR here.*

are grouped by matrix; for each one, a *five-number summary* (minimum and maximum, first quartile, second (median) and third quartiles) is drawn with a *boxes and whiskers* representation.

Experimental Setup

We use a *AMD EPYC 7742* node with 64 cores. Scaling of memory bandwidth in STREAM-like loops here is around 10×. Considering we are dealing with memory-bound operations, we chose `OMP_NUM_THREADS=24`, `OMP_PROC_BIND=spread`, and `OMP_PLACES=cores`. `RSB_USER_SET_MEM_HIERARCHY_INFO` was set to `"L2:4/64/16000K,L1:8/64/32K"`. We use CSR from `csr_matrix` in SciPy `e171a1` from Feb 20, 2021, PyRSB `8a6d603` from Jun 08, 2021, pre-release LIBRSB-1.3. For both, we use `-Ofast -march=native -mtune=native` flags and `gcc version 10.2.1 20210110 (Debian 10.2.1-6)`. We use matrices which were also used in

[Martone14], available from https://sparse.tamu.edu/ ([SSMC]); see the table below. Many of these are symmetric; differently than `rsb_matrix`, `csr_matrix` does not support *symmetric SpMM*; therefore in both cases we expand their symmetry and perform only *unsymmetric* (general) SpMM. Before starting any measurement, we run `autotune` on a temporary matrix to *warm-up* the OpenMP environment, once. Then we do one non-timed *warm-up* SpMM before iterating for 0.2s and taking the fastest sample. We repeat this for each of the 28 matrices, right-hand-sides (NRHS) in `1,2,4,8`, order among `'C'` and `'F'`, *BLAS numerical types* in `C,D,S,Z`. When using `rsb_matrix`, we measure both non-autotuned, and autotuned with `autotune(nrhs=...,order=...,tmax=0)`. So the above totals to $28 \cdot 4 \cdot 2 \cdot 4 = 896$ records with samples in SpMM and tuning timing. To avoid also timing repeated allocation of the SpMM result (`C` in `C=A*B`), we allocate it once, and then instead of the `*` operator, we use the functions underneath it, which take `C` as argument (**this can be of interest to many performance-conscious users**).

	matrix	nonzeroes	rows	ratio
1	arabic-2005	6.40e+08	2.27e+07	28.1
2	audikw_1	7.77e+07	9.44e+05	82.3
3	bone010	7.17e+07	9.87e+05	72.6
4	channel-500x100x100-b050	8.54e+07	4.80e+06	17.8
5	Cube_Coup_dt6	1.27e+08	2.16e+06	58.8
6	delaunay_n24	1.01e+08	1.68e+07	6.0
7	dielFilterV3real	8.93e+07	1.10e+06	81.0
8	europe_osm	1.08e+08	5.09e+07	2.1
9	Flan_1565	1.17e+08	1.56e+06	75.0
10	Geo_1438	6.32e+07	1.44e+06	43.9
11	GL7d19	3.73e+07	1.91e+06	19.5
12	gsm_106857	2.18e+07	5.89e+05	36.9
13	hollywood-2009	1.14e+08	1.14e+06	99.9
14	Hook_1498	6.09e+07	1.50e+06	40.7
15	HV15R	2.83e+08	2.02e+06	140.3
16	indochina-2004	1.94e+08	7.41e+06	26.2
17	kron_g500-logn21	1.82e+08	2.10e+06	86.8
18	Long_Coup_dt6	8.71e+07	1.47e+06	59.2
19	nlpkkt160	2.30e+08	8.35e+06	27.5
20	nlpkkt200	4.48e+08	1.62e+07	27.6
21	nlpkkt240	7.74e+08	2.80e+07	27.7
22	relat9	3.90e+07	1.24e+07	3.2
23	rgg_n_2_23_s0	1.27e+08	8.39e+06	15.1
24	rgg_n_2_24_s0	2.65e+08	1.68e+07	15.8
25	RM07R	3.75e+07	3.82e+05	98.2
26	road_usa	5.77e+07	2.39e+07	2.4
27	Serena	6.45e+07	1.39e+06	46.4
28	uk-2002	2.98e+08	1.85e+07	16.1

SpMM Speedup: from `csr_matrix` to `rsb_matrix`

Figure 8 summarizes the speed ratio of non-autotuned `rsb_matrix` over `csr_matrix`. Speedup without RSB autotuning ranges from $4\times$ to $64\times$, with median $15\times$. Half of observed speedup cases falls between $11\times$ and $20\times$. A *streaming memory access* benchmark we ran on this machine scaled up to circa $10\times$, which just less than the observed median speedup (remember `rsb_matrix` is running with multiple cores, but `csr_matrix` cannot exploit that).

For the reader who is not practical of SpMM performance: the memory access pattern of SpMM is typically very irregular, and largely dependent on the sparsity structure of the matrix. For this reason, for most layouts the multicore scaling of SpMM performance (in particular SpMV) tends to be worst than a streaming memory access scaling. But here we are comparing speed ratios of different algorithms, and these ratios differ as well. That reflects

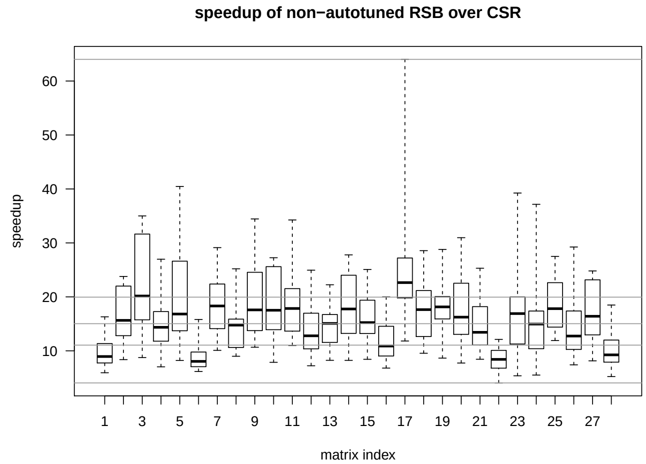

Fig. 8: Performance samples grouped by matrices. Each box represents a group of measurements on the different numerical type, NRHS, and operands layout. The middle horizontal line is the median speedup of RSB vs CSR, corresponding to $15\times$. The other lines are the extremes, and the first and third quartiles in between (the second quartile being the median value). Notice autotuned *results in Fig. 9 improve this further.*

the better or worse aptness of a given format to a given matrix. For instance, matrix 17 has nonzeroes scattered quite regularly over the entire matrix, not much clustered: this favours RSB and the *cache blocking* induced by its structure rather than CSR (serial or not). Conversely, matrix 9 has most of its nonzeroes adjacent to some other, which is more CSR-friendly, and a contribution to the lesser improvement when switching to RSB here. See [Martone14] for more RSB-vs-CSR commentary.

The speedups shown so far and those in Fig. 8 rely on default RSB layouts. As said earlier, the RSB format is suited best to scenarios with large matrices and repeated SpMM applications. These are also the scenarios where the usage of `autotune`, which refines the default layout according to the operands at hand, is most convenient.

Figure 9 shows results with autotuned instances. Here `autotune` has been called for each combination of matrix, operands layout, NRHS, and numerical type. The median speedup over CSR here (circa $28.8\times$) is almost twice the one before autotuning.

With respect to non-autotuned RSB samples, the application of `autotune` brought a median improvement of $1.6\times$. This includes all samples, inclusive of the lower quartile, with speedup between $1\times$ (no speedup) and $1.2\times$, which we nevertheless regard as *ineffective* (see next subsection's discussion). An overview of which matrix benefited more, and which less from autotuning is given by Fig. 10. There is no clear trend to see here. We observe that most of the cases (70%) benefited from autotuning. It's worth mentioning that the longer the time limit chosen to run SpMM before taking each performance sample, the less the fluctuation we would have encountered here, and times we chose were quite tight.

Speedups of tuned RSB vs CSR have median $29\times$ with the `'C'` layout, and $28.6\times$ with `'F'` layout; also within RSB the `'C'` layout performs a few percentage points better than `'F'`.

As seen in this section, autotuning can speedup RSB a further bit, but not always. The next section quantifies the cost of autotun-

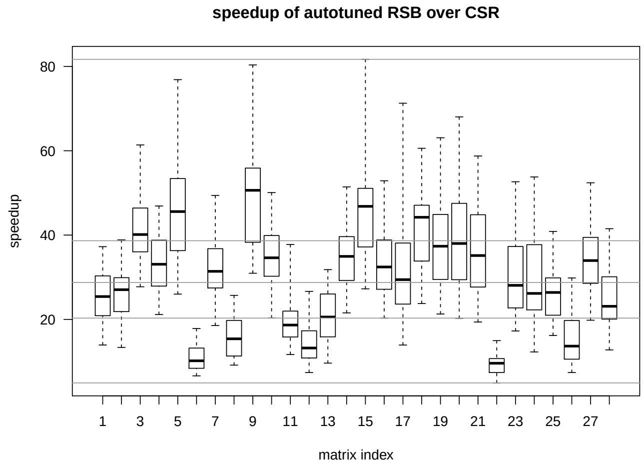

Fig. 9: *We observe speedup over CSR from a few up to* $81.7\times$*, with median of* $28.8\times$*. Certain matrices benefit from RSB more (see matrices 5, 9, 15, 18), while others less (6,22,..). Compare the relevant improvement over non-autotuned results in Fig. 8, or see Fig. 10 for the per-matrix ratios.*

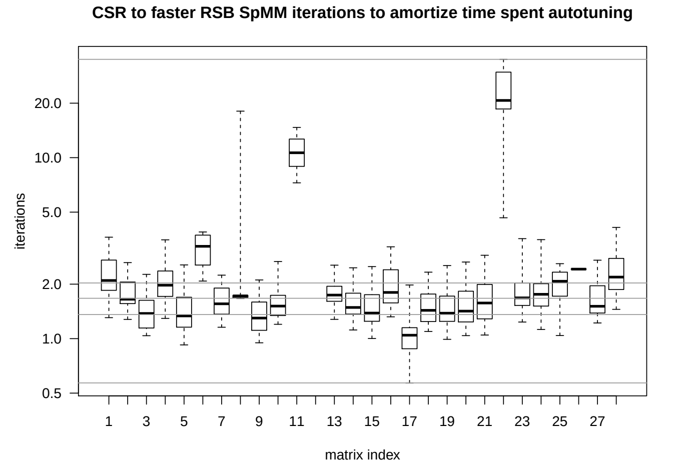

Fig. 11: *Were one to use RSB instead of CSR, and obtain an autotuned instance via* `autotune`*, then this would amortize in few iterations. Notice than in the intended scenarios, where thousands of SpMM are foreseen, this is completely negligible. Note: autotuning was effective in 70% of the cases, represented here and in Fig. 12.*

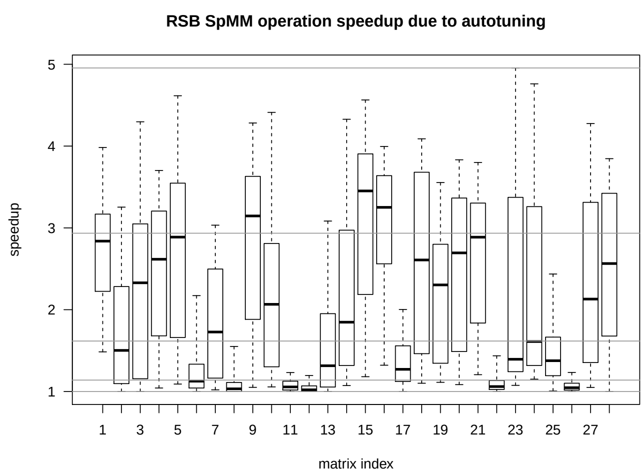

Fig. 10: *Per-sample autotuning effectiveness statistics: autotuned RSB SpMM speed to non-autotuned one. Half of the cases improve by* $>1.6\times$*, 25% of the cases by* $>2.9\times$*. Matrices 8,11,12,22,26 seem to barely profit from it. These are the same ones that exhibit the highest ineffective autotuning cost on Fig. 13.*

ing in practical terms, for both effective and ineffective outcomes.

The Cost of RSB Autotuning

As introduced earlier, `autotune` adapts the structure of an RSB matrix, seeking instances which execute a specified operation (here, SpMM) faster. A consistent fraction of the autotuning time is spent measuring SpMM timings of *prospective RSB instances*. It's important to remark: what one wants here is not merely faster execution of SpMM *after* autotuning. What one wants is that autotuning plus all following SpMM iterations shall take less time than the same count of iterations with a non-autotuned matrix. In other words, if the time savings of faster SpMM's cannot cover the autotuning duration, autotuning time is lost. For this reason it is convenient to quantify the number of iterations to reach the first SpMM bringing actual time saving (*amortization*); this is the duration of `autotune` divided by the time *saved* at each iteration (that is, *slow* time with *old RSB blocking*, minus *faster* time with *new RSB blocking*).

For the purpose of this article, we chose to declare autotuning as *effective* if it brings a speedup of 20% or more. With this threshold set, while 94.5% of the cases get *some* speedup, it is 70% that qualify also as effective.

What one observes among effectively autotuned cases (see Fig. 11) is that in 75% of those cases, merely 2.5 CSR iterations are enough to amortize the autotuning time. This is thanks to the large speedup going from (serial) CSR to (parallel) RSB.

If as cost unit we consider going from non-autotuned to autotuned RSB instead, then the relative gain is less (because threaded non-autotuned RSB is already much faster than serial CSR), and consequently, it takes more to amortize it; see Fig. 12.

When autotuning was ineffective (30% of the cases with our $1.2\times$ threshold, though only 5.5% exhibit no speedup at all), we regard its time as lost; in our test setup this was from a few dozen to a few hundred RSB iterations, with median 33; see Fig. 13. If expressed in terms of serial CSR iterations, these would be <2.8 iterations in half of the cases, <8 in 75% of the cases.

These results shall convince users that using `autotune` is a good option most of the times.

Conclusions and Future Work

Full utilization of the parallelism potential is important in achieving efficient operations on current CPUs. **PyRSB** does that by giving Python users transparent access to the shared-memory parallel *performance library* **LIBRSB**. Differently than classes in current `scipy.sparse`, but with a very similar usage interface, PyRSB's `rsb_matrix` readily exploits shared-memory parallelism. This article's results section gave a wide sample of speedup statistics with respect to SciPy's `csr_matrix`, on the SpMM operation. Observed median speedup with respect to `csr_matrix` exceeded the known memory bandwidth speedup on the machine; with autotuning, it doubled that, speaking for the good implementation in LIBRSB. Trade-off considerations in

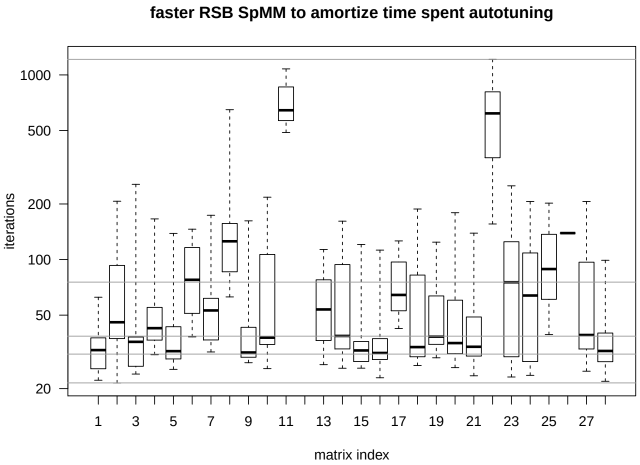

Fig. 12: *If one were to start autotuning from RSB (thus with less improvement potential than with CSR), the amortization times* cost *more iterations (here, median is* 38.4×*, 75% of the cases below* 76×*). Nevertheless, for many problems, where thousands of iterations are foreseen, this is perfectly acceptable.*

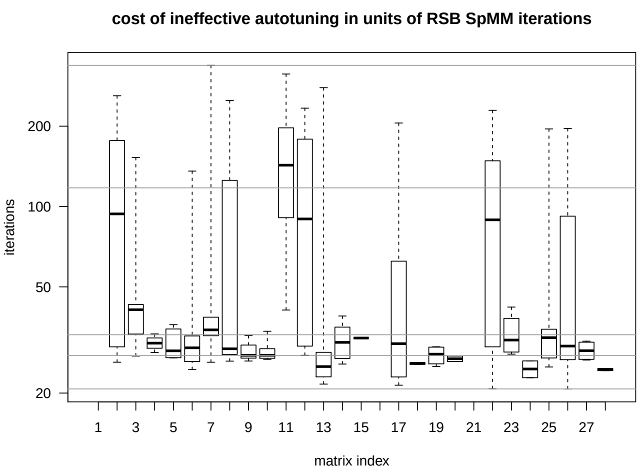

Fig. 13: *There is no guarantee autotuning improves SpMM performance. Actually, autotuning would be unnecessary, if we were able to guess blockings optimal under all circumstances. Indeed, without further analysis, one may even speculate that the default RSB blocking matrices where autotuning was ineffective, was also the* best*. In our experiment, ineffective autotuning searches* ***cost*** *33× RSB (only 2.8× CSR) SpMM iterations in the median case. Note that for certain matrices (1,16,21) autotuning was always effective: this is why these have no associated box here.*

using PyRSB effectively by means of autotuning have also been delineated.

SpMM and autotuning are the *workhorses* of PyRSB and we addressed their use here. Follow-up studies may address or reflect improvements on the LIBRSB side, special use cases, as well as mostly usability-related aspects on the PyRSB side, especially in striving for SciPy interoperability in the user interface. Comparing symmetric SpMM of PyRSB to that of specific *symmetric formats* in SciPy may also be of interest.

Acknowledgments

This work has been financed by **PRACE-6IP**, under Grant agreement ID: 823767, under Project name *LyNcs*. LyNcs is one of 10 collaborations supported by PRACE-6IP, WP8 *"Forward Looking Software Solutions"*. Performance results have been obtained on systems in the test environment **BEAST** (*Bavarian Energy Architecture & Software Testbed*) at the Leibniz Supercomputing Centre. We are grateful to reviewer Meghann Agarwal for her energetic help in improving the quality of this article.

References

[PYRSB] *PyRSB*. (2021, May). Retrieved May 28, 2021, https://github.com/michelemartone/pyrsb

[LIBRSB] *LIBRSB*. (2021, May). Retrieved May 28, 2021, https://librsb.sf.net

[Martone14] Michele Martone. "Efficient multithreaded untransposed, transposed or symmetric sparse matrix-vector multiplication with the Recursive Sparse Blocks format". Parallel Comput. 40(7): 251-270 (2014). http://dx.doi.org/10.1016/j.parco.2014.03.008

[Virtanen20] P.Virtanen, R.Gommers, T.Oliphant, et al. "SciPy 1.0: fundamental algorithms for scientific computing in Python". Nat Methods 17, 261–272 (2020). https://doi.org/10.1038/s41592-019-0686-2

[Behnel11] S.Behnel, R.Bradshaw, C.Citro, L.Dalcin, D.S.Seljebotn and K.Smith. "Cython: The Best of Both Worlds", in Computing in Science & Engineering, vol. 13, no. 2, pp. 31-39, March-April 2011, doi: https://doi.org/10.1109/MCSE.2010.118

[RSB_JL] *RecursiveSparseBlocks.jl*, (2021, April 08). Retrieved April 08, 2021, from https://github.com/dcjones/RecursiveSparseBlocks.jl.git

[Abbasi18] H.Abbasi, "Sparse: A more modern sparse array library", Proceedings of the 17th Python in Science Conference (SciPy 2018), July 9-15, 2018, Austin, Texas, USA. http://conference.scipy.org/proceedings/scipy2018/hameer_abbasi.html

[PyDataSparse] *PyDataSparse.jl*, (2021, April 08). Retrieved April 08, 2021, from https://github.com/pydata/sparse.

[Lee15] M.Lee, W.Chiang and C.Lin, "Fast Matrix-Vector Multiplications for Large-Scale Logistic Regression on Shared-Memory Systems," 2015 IEEE International Conference on Data Mining, Atlantic City, NJ, USA, 2015, pp. 835-840, doi: https://doi.org/10.1109/ICDM.2015.75

[Stegmeir15] A.Stegmeir (Jan 2015). "GRILLIX: A 3D turbulence code for magnetic fusion devices based on a field line map". Available from INIS: http://inis.iaea.org/search/search.aspx?orig_q=RN:46119630

[Klos18] P.Klos, S.König, H.-W.Hammer, J.E. Lynn, and A.Schwenk. "Signatures of few-body resonances in finite volume". Phys. Rev. C 98, 034004 – Published 24 September 2018, doi: https://doi.org/10.1103/PhysRevC.98.034004

[Wu16] L.Wu. "Algorithms for Large Scale Problems in Eigenvalue and Svd Computations and in Big Data Applications" (2016). Dissertations, Theses, and Masters Projects. Paper 1477068451. http://doi.org/10.21220/S2S880

[Browne15T] P.A. Browne, P.J. van Leeuwen. "Twin experiments with the equivalent weights particle filter and HadCM3". Quarterly Journal of the Royal Meteorological Society, vol. 141, no. 693, pp. 3399-3414, https://doi.org/10.1002/qj.2621

[Browne15M] P.A. Browne, S. Wilson. "A simple method for integrating a complex model into an ensemble data assimilation system using MPI". Environmental Modelling & Software, vol. 68, pp. 122-128, https://doi.org/10.1016/j.envsoft.2015.02.003

[SPACK] *Spack*. (2021, May). Retrieved May 28, 2021, https://spack.io

[EASYBUILD] *EasyBuild*. (2021, May). Retrieved May 28, 2021, https://easybuild.io

[DEBIAN] *Debian*. (2021, May). Retrieved May 28, 2021, http://www.debian.org

[UBUNTU] *Ubuntu*. (2021, May). Retrieved May 28, 2021, http://www.ubuntu.com

[OPENSUSE] *OpenSUSE*. (2021, May). Retrieved May 28, 2021, from https://www.opensuse.org

[GUIX] *GuixHPC*. (2021, May). Retrieved May 28, 2021, from https://hpc.guix.info/

[SPARSERSB] *SparseRSB*, (2021, April 09). Retrieved April 09, 2021, from https://octave.sourceforge.io/sparsersb/

[SSMC] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. ACM Transactions on Mathematical Software 38, 1, Article 1 (December 2011), 25 pages. doi: https://doi.org/10.1145/2049662.2049663

[OPENMP] *OpenMP*, (2021, May). Retrieved May 28, 2021, from https://www.openmp.org/