Proceedings of the International Multiconference on Computer Science and Information Technology pp. 317–325
ISBN 978-83-60810-27-9
ISSN 1896-7094

Assembling Recursively Stored Sparse Matrices

Michele Martone, Salvatore Filippone, Salvatore Tucci
University of Rome "Tor Vergata", Via del Politecnico 1, 00133 Rome, Italy
Email: {michele.martone,salvatore.filippone,tucci}@uniroma2.it
Marcin Paprzycki
Warsaw Management Academy, Poland Email: marcin.paprzycki@ibspan.waw.pl

***Abstract*—Recently, we have introduced an approach to multi-core computations on sparse matrices using recursive partitioning, called *Recursive Sparse Blocks* (*RSB*). In this document, we discuss issues involved in assembling matrices in the RSB format. Since the main expected application area is iterative methods, we compare the performance of matrix assembly to that of matrix-vector multiply (*SpMV*), outlining both scalability of the method and execution times ratio.**

I. Introduction

IN RECENT papers, we have introduced a recursive data structure for performing Sparse BLAS ([1]) operations on cache based multi-core architectures. Initial performance results for the proposed format and its modifications/improvements, for both matrix-vector multiplication and triangular solve have been very encouraging (see, [2], [3], [4], [5]). We call our hybrid format *Recursive Sparse Blocks* (*RSB*), as it features the recursive subdivision of a matrix, with sparse submatrices in the leaves of a quad-tree.

Our efforts are targeted primarily towards *iterative methods*, appearing in the solution of either linear or eigenvalue problems ([6]). Here, the usual case for the user, is to select a specific iterative method (and preconditioner) for a given computational problem. For instance, the choice of a method over another could be influenced by the available computer hardware. With different sparse matrix storage formats, different operations could have different performance patterns on variously configured core/processor hardware. An important motivation for our work (as well as that of others, e.g.: CSB [7]) is the need for a format capable of performing thread-parallel Sparse Matrix-Vector Multiply (*SpMV*) with the same ease and comparable performance to the *Transposed SpMV* (*SpMV_T*). Since many iterative methods require both *SpMV* and *SpMV_T* (e.g. CGS or BiCGSTAB; see [8, section 2.4]), the development of a unified parallel approach to both operations is needed. Here, the common matrix formats, as *Compressed Sparse Rows* (*CSR*), and the plain *COOrdinate* (*COO*) formats do not support efficiently parallel *SpMV_T*.

At the problem level, the use of iterative methods often occurs in the solution of partial differential equations (PDEs), e.g. arising from simulating some physical phenomenon. Here, different usage patterns could arise; for instance: i) a sparse matrix could be instantiated and solved once; ii) a sparse matrix could be instantiated once and solved against a number of (updated/not known *a priori*) right hand sides. In choosing the best solution method for a given problem, the developer should take into account the overall (wall-clock) computation time; that is, the time to assembly a sparse matrix (and for updating it, if needed), as well as solving the problem. Additionally, the time for computing a preconditioner (See [6, Ch.10]) should be taken into account, but this issue is outside the scope of this paper.

The assembly of a sparse matrix could proceed in a number of ways. The author of [9, Ch.4] presents some common cases for the assembly of symmetric and unsymmetric matrices—from *usual* Finite Element Methods (FEM) data arrays as input, into COO and CSR formats. Overall, the most general routine for assembling a sparse matrix should be able to process as input the full list of coordinates of the matrix elements (*structural nonzeroes*), defining the *nonzeroes pattern* of the matrix. The numerical values of the matrix elements could be specified or updated in a later moment. The RSB matrix "*constructor*" we discuss here, works by converting the original array specifying the matrix as row-major sorted COO (now on, we will refer to COO assuming it as row-major sorted) into the RSB layout. Note that, from the assembly routines we present, it would be easy to derive routines for the extraction/conversion of rows or block-rows of RSB matrices.

We start by reviewing related works, in Sec. II. Next, we outline selected properties of RSB's quad-trees (in Sec. III). We follow by a presentation of our algorithm for converting a row-major sorted COO matrix to RSB, in Sec. IV. Then, in Sec. VI we report, for a representative set of matrices, how many *SpMV*'s are time-equivalent to a single matrix assembly. While we do not claim optimality of either of our techniques (the assembly and the *SpMV*), presented results illustrate the relevant relation between performance of these two operations. We have selected this particular comparison since, as stated above, *SpMV* is the basic operation for most standard iterative methods. Experiments were performed on machines and matrices described in Sec. V.

II. Related Work

The use of recursion in numerical linear algebra is a *recurring* theme (e.g., see [10]). We found proposals for *hypermatrix* (multilevel indexed matrices) based approaches to the solution of linear systems dating back to 1972 [11] (usage of hypermatrices with dense submatrices), and 1969 [12] (out-of-core dense matrix computations). Further, with regards to sparse matrices, Herrero and Navarro [13] investigated a hypermatrix-based Cholesky solver, but without discussing

performance of hypermatrix assembly. In [14] authors used asymmetrical *recursive bipartitioning* of sparse matrices in a distributed computing context. Sparse hypermatrix techniques were reportedly used for distributed-memory operations in the PERMAS proprietary package for FEM analysis [15]. The techniques most similar to the ones investigated here, reported in [16], deal with the optimal balancing of sparse matrix computations across distributed processors. Interestingly enough, while hypermatrix-based approaches have been applied to sparse matrix computations, almost no research has been reported as to what concerns assembly of such matrices. However, we found research in a similar spirit to ours in [17]. It documents algorithms and discusses patterns of usage in the assembly of sparse matrices in the context of their *serial* "MTL" package. Our discussion is more limited but in-depth than that, as we are concerned with a single pattern of construction: the conversion of COO input arrays to RSB, to be used on multi-core computers.

III. Some Properties of the Quad Trees Used in RSB Matrices

We described our rules for the construction of a quad tree-based recursive matrix representation in [2]. Given an $m \times k$ matrix A, we build a graph structure (*quad-tree*) q with nodes corresponding to *quadrant submatrices*. The four quadrants are sized respectively (in clockwise order, from the upper left) $\lceil \frac{m}{2} \rceil \times \lceil \frac{k}{2} \rceil$, $\lceil \frac{m}{2} \rceil \times \lfloor \frac{k}{2} \rfloor$, $\lfloor \frac{m}{2} \rfloor \times \lceil \frac{k}{2} \rceil$, and $\lfloor \frac{m}{2} \rfloor \times \lfloor \frac{k}{2} \rfloor$. This subdivision (or *bipartition*) is applied recursively to the quadrants; quadrants with no nonzero are not represented. Only leaf nodes are associated with actual data arrays, while inner ones contain only pointers. A simple *cutoff* function is used to balance the tree in order to obtain *leaf submatrices* with neither too many, nor too few nonzeroes. Fig. 23 depicts a matrix subdivided into RSB.

Let us now review some properties of our quad-trees, which will be useful during the discussion of matrix assembly.

Let us call q_h the *complete* quad-tree of height h; that is, the quad-tree having $N_i(q_h) \stackrel{def}{=} \sum_{i=0}^{h-1} 4^i$ intermediate nodes and $N_l(q_h) \stackrel{def}{=} 4^h$ leaf nodes, We indicate with $H(q)$ the height of quad-tree q. We assume that any quad-tree could be constructed by adding nodes to the singleton quad-tree q_0 (the one which is associated to the entire matrix). Let Q be the set of quad-trees with height ≥ 1. We call q' a $k-$*derivation* (or *derivation*, for short, if we ignore k) of quad-tree q, if q' can be built from q, by making one leaf an intermediate node, and adding $1 \leq k \leq 4$ leaves. We call q' an *indirect derivation* of quad-tree q, if q' can be built from q after a sequence of derivations. Observe that if q' is a $k-$derivation of q, then $N_i(q') = N_i(q) + 1$, and $N_l(q') = N_l(q) + k - 1$.

Property 1. *For any q among the possible quad-trees with height 1, we have $\frac{N_i(q)}{N_l(q)} \geq \frac{1}{4}$, and $\frac{N_i(q_1)}{N_l(q_1)} = \frac{1}{4}$.*

Proof: By explicit enumeration of possible cases. ■

Property 2. *For any $q \in Q$ with $H(q) > 1$,we have $\frac{N_i(q)}{N_l(q)} \geq \frac{1}{4}$*

```
1  /*Matrix A is expressed using arrays I, J, V*/
2  Instantiate the root matrix node s_A, marked RSB and "open"
3  [P, s_A] ← COO_to_RSB_s(s_A, I, J)/*symbolic subdivision*/
4  /*Now s_A is the root of a quad-tree for A, with empty leaves*/
5  /*P is a rows pointer array for I, J, V*/
6  COO_to_RSB_V(s_A, P, V)     /*numerical arrays shuffling*/
7  COO_to_RSB_J(s_A, P, J) /*indices shuffling/displacement*/
8  /*P is no longer needed and I, J, V are in RSB order*/
9  RSB_Leaf_Switch(s_A)      /*indices switch*/
10 /*A number of leaf matrices has halfword indices, now.*/
11 return s_A  /*return s_A, now quad-tree for A*/
```

Fig. 1. $COO\_to\_RSB(I, J, V)$

Proof: Let q be a quad-tree having $\frac{N_i(q)}{N_l(q)} < \frac{1}{4}$, necessarily a derivation of a quad-tree q' having $\frac{N_i(q')}{N_l(q')} \geq \frac{1}{4}$. In the case q is a $k-$derivation of q', indicating $i = N_i(q'), l = N_l(q')$, we have $\frac{i}{l} \geq \frac{1}{4}$ and $\frac{i+1}{l-1+k} < \frac{1}{4}$. But this implies $4i - l \geq 0$ and $4i - l < k - 5$, which is impossible, for $1 \leq k \leq 4$. In the case q is an indirect derivation of q', it must be a derivation of some quad-tree q" having $\frac{N_i(q")}{N_l(q")} < \frac{1}{4} \leq \frac{N_i(q')}{N_l(q')}$, but existence of such q" is impossible, as we have seen. ■

If some internal node of q has one child only, we call q *degenerate* (with some terminology abuse;see [18, Sec. 2.3.4.5]).

Property 3. *For any sparse matrix M with no empty rows, if its corresponding quad-tree q is not degenerate, we have $\frac{N_i(q)}{N_l(q)} \leq 1$.*

Proof: Since M has no missing rows, it has some leaf node of q covering each row interval. Since q is not degenerate, at each level > 1, there are at least two nodes, or no node at all. Therefore, quad-tree q can be built by inserting additional $k \geq 0$ leaves to some binary tree q'. A non degenerate binary tree q' has $N_l(q') = N_i(q') + 1$, So we have $\frac{N_i(q)}{N_l(q)} = \frac{N_i(q')}{N_i(q')+k+1}$, whose upper limit is 1, for $k = 0$, and $N_i(q) \to \infty$. ■

Property 3 guarantees that for *non degenerate* quad-trees, there will be no more internal nodes than leaves. Please note that, for the time being, for simplicity of implementation, degenerate quad-trees are allowed in RSB.

IV. Building RSB From COO

Let us now describe in detail our approach for the conversion of an $m \times k$ matrix A with nnz nonzeros, expressed in (row-major sorted) COO (I, J coordinate arrays and the V numerical values array) into RSB order. The proposed procedure builds a quad-tree structure for A, allocating a *small* number of auxiliary structures (see the previous section) for the submatrix nodes, and reusing I, J, V. Unless otherwise stated, in the following, by *matrix* we will refer to A only, and denote as a *submatrix* any of the *quadrant submatrices* obtained by recursive bipartitioning (defined in Sec. III). Here, we assume no duplicates in the input (which happen in publicly available matrices; e.g.: ones from [19]).

There are three stages of assembly: first the *subdivision* of A in $COO\_to\_RSB\_s$, where the input is repeatedly scanned, and a quad-tree structure is built; then the *shuffling* of rows laid in COO order to the rows of RSB sub-

```
1  N ← [0, 0, 0, 0] /*nonzeroes count for quadrants */
2  Allocate four (s.m + 1)-sized arrays L, M, R, P
3  /*CS: Cache(s) Size, ES(= 8 for double): Element Size*/
4  s_A.N_S ← 0; s_A.MAX_S ← (s_A.nnz · ES)/(CS/N_threads)
5  COO_RowP(I, J, P, s_A.nnz, s_A.m) /*fill row pointers in P*/
6  s_A.L ←(p) P; s_A.R ←(p) P + 1
7  /*s_A.L points to row beginnings, s_A.R points to row endings
   (aliasing the second element of P)*/
8  while Some leaf submatrix is still "open" do
9      s_A.N_S ← s_A.N_s + 1;
10     s ← "largest" open submatrix
11     if δ_r(s.m, s.k, s.nnz, CS, ES, WS) then
12         /*copy subrow pointers stored in s.I, s.J */
13         L ← s.I; R ← s.J;
14         /*get quadrants info N, fill middle pointers array M*/
15         N ← Subrow_Split(s, L, R, M, J)
16         /*split s, appending up to four quadrant submatrices*/
17         RSB_Split_Node(s, N, L, M, R, I, J)
18     else
19         /*closing (marking as terminal)*/
20         if s is s_A && s.nnz ≥ s.m + 1 then s.I ← L
21         /*For s_A, a copy is necessary.*/
22         if s.nnz ≥ s.m + 1 then Mark as CSR
23         else Mark as COO
24     end
25 end
26 return [P, s_A]/*Arrays L, M, R can be freed.*/
```

Fig. 2. $COO\_to\_RSB\_s(s_A, I, J)$

matrices (Fig. 8,9), and finally *compression of indices* in RSB_Leaf_Switch (Fig.10). Accordingly, we break down the RSB assembly pseudo code into three listings, called from procedure COO_to_RSB, in Fig. 1.

Procedure $COO\_to\_RSB\_s$ (Fig. 2), performs a cycle, identifying bounds for candidate submatrices. This information is stored in auxiliary arrays L, M, R. A *row pointers* array P is constructed (line 5), kept and returned for later usage. At each iteration, the *largest open submatrix* s (in terms of number of nonzeroes) is selected; then it is analyzed, and either subdivided in quadrants (and marked as *closed node*) or marked as a *closed leaf*. In either case, each cycle *closes* submatrix s and *opens* up to four submatrices. Therefore, the loop iterates a number of times equal to the number of the nodes (both inner and leaf) in the produced quad-tree. In Fig. 3 we present the cutoff function δ_r which decides if subdivision of s should proceed.

Since the input COO arrays are row-major sorted, in order to identify quadrants of s in them, we need to mark, for each row, indices for: the leftmost element of the two left quadrants, the leftmost of the two right quadrants, and the first one after the rightmost of the two right quadrants; that is, pin-point *subrows* in each quadrant. To this end, I and J are scanned in $Subrow\_Split$, and subrows information is stored in the three *row pointers arrays* L, M, R. Row pointers data will be reused when assembling submatrices in CSR. The first invocation of $Subrow\_Split$ requires L, R for the whole A in order to compute the first *middle row pointers* array M. Notice that for any row i of A, $L[i+1] \equiv R[i]$. For this reason, before

```
1  /*WS(= 4): Word Size of index element, μ = 3 */
2  if s_A.N_S ≥ s_A.MAX_S then return False
3  if n · ES > 2 · CS && m < 2^16 && k < 2^16 then return
   True
4  if (ES (2 · n + m) + WS · (m + n)) > α CS &&n/m > μ
   then return True
5  return False
```

Fig. 3. $\delta_r(m, k, n, CS, ES, WS)$

```
1  P[:] ← 0/* fill with zeros*/
2  for n ← 0 to nnz − 1 do P[I[n] + 1] ← P[I[n] + 1] + 1
3  for i ← 0 to m − 1 do P[i + 1] ← P[i + 1] + P[i]
4  /*for each i, P[i] now has the offset of row i in I, J*/
```

Fig. 4. $COO\_RowP(I, J, P, nnz, m)$

entering the loop, we pre-compute a single row pointers array P, and set the initial L, R as *pointer aliases* of P. That is, P can serve as L, and aliased after its first element, as R does; in Fig. 2 and 7, we have used "$\overset{p}{\leftarrow}$" to signify pointer aliasing. P is computed by COO_RowP, listed in Fig. 4.

After boundaries are identified, and nonzeroes counts are known for each quadrant, at line 17, we invoke the RSB_Split_Node. It will add an *open* leaf submatrix for each non-empty quadrant, and copy the L, M, R arrays in appropriate offsets of the I array. In this way, I is used for storing submatrices rows information, and subsequent invocations of $Subrow\_Split$ will use the L, R arrays recovered from there.

In the case the δ_r does not make s a candidate for subdivision, s gets *closed* as a leaf matrix, and marked to contain data in the CSR or COO format (depending on the available index space; lines 18-23). In the case s is the root node for A (s_A), and fitting CSR arrays ($nnz > m$), L (aliasing P) is copied at the appropriate offset of I, overwriting original row indices (not needed anymore).

After assembling the quad-tree for the s_A, the original J, V arrays storing column indices and values of the matrix coefficients are still unmodified, and ready for being displaced

```
1  n_00 ← 0; n_01 ← 0; n_10 ← 0; n_11 ← 0;
2  for i ← 0 to ⌊(s.m + 1)/2⌋ do
3      M[i] ← Search(J, L[i], R[i], s.j_0 + ⌈s.k/2⌉)
4      n_00 ← n_00 + (M[i] − L[i]); n_01 ← n_01 + (R[i] − M[i])
5  end
6  for i ← ⌈(s.m + 1)/2⌉ to s.m − 1 do
7      M[i] ← Search(J, L[i], R[i], s.j_0 + ⌈s.k/2⌉)
8      n_10 ← n_10 + (M[i] − L[i]); n_11 ← n_11 + (R[i] − M[i])
9  end
10 return [n_00, n_01, n_10, n_11]
```

Fig. 5. $Subrow\_Split(s, L, R, M, J)$

```
1  Binary search for the smallest m such that J[m] ≥ h and
   l ≤ m ≤ r
2  return m
```

Fig. 6. $Search(J, l, r, h)$

```
1  Q ← [...] /*allocate a submatrix structure for each nonempty
   quadrant of s; then for each quadrant q_ij, set info for
   nonzeroes, dimensions, and row,column,nonzeroes offsets
   relative to the whole matrix; then copy portions from the
   subrow pointer arrays from L, M, R*/
2  if n_00 > 0 then
3      q_00.m ← ⌈s.m/2⌉; q_00.k ← ⌈s.k/2⌉;
4      q_00.moff ← s.moff + 0; q_00.koff ← s.koff + 0;
5      q_00.nzoff ← s.nzoff + 0; q_00.nnz ← n_00;
6      q_00.I ←^p I + q_00.nzoff; q_00.J ←^p J + q_00.nzoff
7      if q_00.nnz > 2 · q_00.m + 2 then
8          q_00.I ← IL[1 : q_00.m]; q_00.J ← IM[1 : q_00.m];
9      end
10 end
11 if n_01 > 0 then
12     q_01.m ← ⌈s.m/2⌉; q_01.k ← ⌊s.k/2⌋;
13     q_01.moff ← s.moff + 0; q_01.koff ← s.koff + q_00.k;
14     q_01.nzoff ← s.nzoff + n_00; q_01.nnz ← n_01;
15     q_01.A ←^p I + q_01.nzoff; q_01.J ←^p J + q_01.nzoff
16     if q_01.nnz > 2 · q_01.m + 2 then
17         q_01.I ← IM[1 : q_01.m]; q_01.J ← IR[1 : q_01.m];
18     end
19 end
20 .../*And so on for q_10, q_11.*/ ...
```

Fig. 7. $RSB\_Split\_Node(s, N, L, M, R, I, J)$

to their destination location. The I array, instead, has been overwritten. For submatrices marked for CSR storage, I already stores a *row pointers array*, which a CSR representation requires. For submatrices marked for COO storage, the relevant subarrays for I could have been overwritten during parent node subdivision, and therefore they should be reinitialized to their original values. Actually, each submatrix node has information on the count of nonzero elements in its own quadrants. Recall, that in RSB_Split_Node, the *nonzero offset* of each submatrix in the quad-tree representation was computed. Now, each submatrix s could be extracted to a temporary storage, row by row, from the original matrix specified in subsequent rows, at the submatrix offset $s.nzoff$ (computed by RSB_Split_Node, in Fig. 7). To keep the shuffling algorithm simple, we have chosen to allocate two temporary J_t and V_t arrays; gather there the displaced rows for coefficients and indices, and copy back to J, V. Since different submatrices should be laid in separate intervals of J and V, the shuffling algorithm can be parallelized on a submatrix basis in a parallel cycle. Once shuffled, the temporary arrays are copied back using a simple OpenMP-parallel wrapper around the standard `memcpy` ([20]) function .

The shuffling procedures for J (Fig.9) and V (Fig.8) are similar. For V ($COO\_to\_RSB\_V$), only rows shuffling is needed, but for J ($COO\_to\_RSB\_J$), besides shuffling, we need also to adjust indices relative to the submatrix location, and restore indices of I. After the shuffling phase, submatrices are either stored as *fullword* (by default, 32 bit) COO or CSR. RSB (see [5]) allows smaller leaves to have 16 bit coordinate (for COO) or column (for CSR) indices. For this, we use a separate procedure, RSB_Leaf_Switch, operating an *in place* conversion on the arrays of the candidate submatrices.

```
1  Allocate a temporary vector V_t, fitting V.
2  parallel foreach s ∈ S do
3      V_s ←^p V_t[s.nzoff]
4      if s.nnz ≥ 2 · s.m + 2 then
5          for i ← 0 to s.m − 1 do
6              Append subrow V[s.L[i] : s.R[i]] to V_s
7          end
8      else
9          for i ← 0 to s.m − 1 do
10             l ← P[s.moff + i];r ← P[s.moff + i + 1]
11             l ← Search(J, l, r, s.koff)
12             r ← Search(J, l, r, s.koff + s.k)
13             Append subrow V[l : r] to V_s
14         end
15     end
16 end
17 MEMCPY_Parallel(V, V_t)/*V ← V_t*/
```

Fig. 8. $COO\_to\_RSB\_V(s_A, P, V)$

```
1  Allocate a temporary vector J_t, fitting J.
2  parallel foreach s ∈ S do
3      J_s ←^p J_t[s.nzoff]
4      if s.nnz > 2 · s.m + 2 then
5          for i ← 0 to s.m − 1 do
6              Append subrow J[s.L[i] : s.R[i]] to J_s
7              Make a CSR row pointer in I, using L, R
8          end
9      else
10         for i ← 0 to s.m − 1 do
11             l ← P[i];r ← P[i + 1]
12             l ← Search(J, l, r, s.koff)
13             r ← Search(J, l, r, s.koff + s.k)
14             Append subrow J[l : r] to J_s
15             if s.nnz < s.m + 1/*COO case*/ then
16                 Set array s.I with value i
17             else
18                 Make a CSR row pointer in I, using L, R
19             end
20         end
21     end
22     Adjust s.J indices, by subtracting the offset s.koff.
23 end
24 MEMCPY_Parallel(J, J_t)/*J ← J_t*/
```

Fig. 9. $COO\_to\_RSB\_J(s_A, P, J)$

Note that interleaving shuffling and conversion could save a substantial fraction of memory accesses; however the constructor logic would be much more involved. After this (last) step, the matrix is assembled as RSB and ready for use.

The presented assembly procedure consists of a *serial* stage (*subdivision*), followed by two stages exploiting parallelism (*shuffling* and *conversion*). Initially, we considered to propose a parallel subdivision step. However, we observed that this would require us to use more complicated techniques, and would also entail differences in the computed partitions. For instance, we could have let threads subdivide the matrix concurrently, but non-determinism in the order of subdivision could lead to non-deterministic quad-tree shape/matrix partitioning. In such case, we would have either to accept the

```
1  parallel foreach leaf node s of quad-tree s_A do
2      if Marked for halfword indices then
3          if CSR format then
4              Convert J to use 16 bit indices, in place
5          end
6          if COO format then
7              Convert I, J to use 16 bit indices, in place
8          end
9      end
10 end
```

Fig. 10. $RSB\_Leaf\_Switch(s_A)$

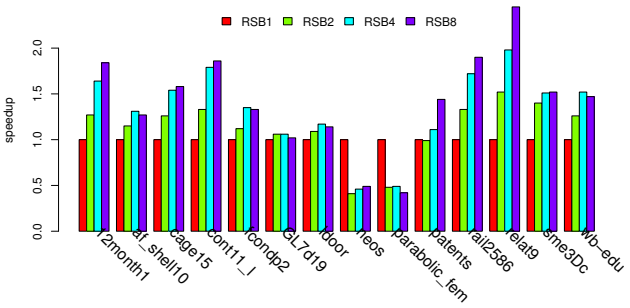

Fig. 11. RSB matrix assembly scaling on **M1**.

algorithm as non-deterministic (which we did not want), or use complicated *backtracking* techniques to revert unnecessarily subdivided submatrices and an equivalent tree. On the other hand, we have found strategies for the parallelization of the current subdivision algorithm routines (based on fine-grained parallelism) to be problematic regarding synchronization, and therefore shortsighted, in the perspective of many-core computations, expected in the forthcoming computers. Therefore, for the time being, we have chosen a simple serial strategy, and left other enhancements for future developments. Indeed, besides being serial, the subdivision stage faces a growing amount of work, as more subdivisions are performed on a matrix; and thus, it will slow down further, the more threads will participate in the *SpMV* computation (recall line 1 in Fig. 2). Each subdivision of a submatrix s requires (a) the copy of two arrays, (b) $s.m$ binary searches during split, and (c) one array write per search. In the worst case, this involves about $s.m$ random accesses in the binary searches, (which perform non-linear accesses), but the remaining accesses are linear, and could be performed taking advantage of the available prefetching engine on the CPU. Analysis of the complexity of subdivision is beyond the scope of this paper; a gross, pessimistic estimate we could provide for the memory traffic would be up to $o(h \cdot nnz)$ memory writes (where h is the height of the quad-tree). This would be the case where all of the submatrices would fit exactly as CSR: if some were COO, binary searches would be performed on their parent matrices, but with no subsequent row pointers copy (matrices are assigned as COO if they don't fit CSR, with no further subdivision). If some submatrices had rows denser ($s.nnz > s.m + 1$), it would mean that only $O(s.m + 1)$ elements would be moved (out of $s.nnz$).

The shuffle stage is different: it involves two transfers of contents of arrays V and J; and between m and nnz element moves for I. If not coupled to the copy operation, the index adjustment for J accounts for further, up to $O(nnz)$, accesses; similarly for restoring the I arrays of COO leaves. Similarly, the complexity of the compression stage involves modifications of up to $2nnz$ memory locations (once). Besides the `memcpy`-like operations, when shuffling the COO submatrices, the J array would be *binary-searched* repeatedly for the identification of subrows bounds (after determining bounds for search using P). The same binary-search based algorithm is needed for the CSR submatrices having $s.nnz \leq m$ (since the corresponding I subarray would not contain both right and left subrows pointer arrays). For CSR leaves having $s.nnz > s.m$, right and left subrow pointers are recovered from I, subrows in J and V are located, and no search is needed at all. Notice the independence from the quad-tree height (and thus, from the matrix size).

V. Experimental Setup and Methodology

TABLE I
MATRICES TEST-SET, OBTAINED FROM [19].

matrix	symm	r	c	nnz	nnz/r
12month1	G	12471	872622	22624727	1814.19
af_shell10	S	1508065	1508065	27090195	17.96
cage15	G	5154859	5154859	99199551	19.24
cont11_l	G	1468599	1961394	5382999	3.67
fcondp2	S	201822	201822	5748069	28.48
GL7d19	G	1911130	1955309	37322725	19.53
ldoor	S	952203	952203	23737339	24.93
neos	G	479119	515905	1526794	3.19
patents	G	3774768	3774768	14970767	3.97
rail2586	G	2586	923269	8011362	3097.97
relat9	G	12360060	549336	38955420	3.15
sme3Dc	G	42930	42930	3148656	73.34
wb-edu	G	9845725	9845725	57156537	5.81

Our experimental setup is similar to that of [5]: same machines, same compilers, same methodology, but for space reasons, we selected only an *essential* subset of the matrices used there (see Table I). We compiled and ran our codes on machines **M1** (AMD Opteron 2354; 2×4-core CPU; caches: 2×2MB L3, 4× 512KB L2 and 64KB L1) and **M2** (Intel Xeon 5670; 2×6-core CPU; caches: 2×12MB L3, 4× 256KB L2 and 32KB L1), using `-O3` as the only optimization flag, with `icc` v.11 on **M1**, and `gcc` v.4.3 on **M2**. The time samples employed are the *best ones*, after 100 runs for the *SpMV* operation, and 10 runs for the constructor. **M2** is a lightly loaded network server.

VI. Results

For space reasons, we won't be able to present a comprehensive analysis of the constructor performance, and thus we will focus on the most important topics. Our exposition is

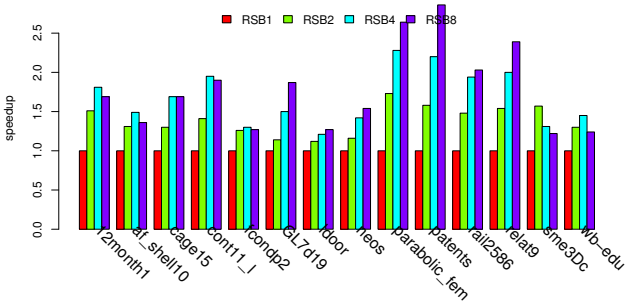

Fig. 12. RSB matrix assembly scaling on **M2**.

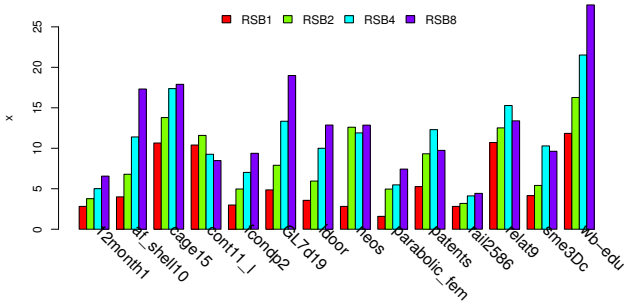

Fig. 13. RSB matrix assembly to *SpMV* time ratio on **M1**.

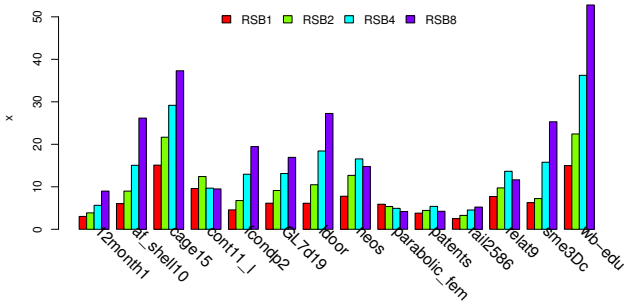

Fig. 14. RSB matrix assembly to *SpMV* time ratio on **M2**.

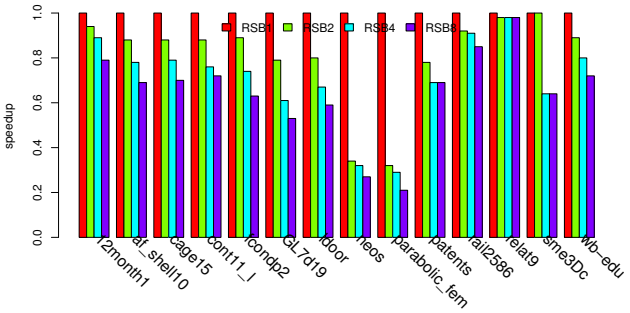

Fig. 15. Subdivision scaling on **M1**.

geared towards iterative methods; here, the affordability of the constructor code is inversely proportional to the number of *SpMV*'s that are expected to be performed after matrix instantiation. Thus, performance profiles for both *SpMV* and construction operations are needed. We will thus present the constructor performance considering two metrics: the number of *SpMV* that are time-equivalent to a constructor run on the given matrix, and the scalability of the constructor with respect to the single core case.

In our previous work (See [5],[4]), using 8 cores on **M1** and **M2**, we have encountered a *SpMV* speedup of up to $5\times$. In Sec. IV, we have motivated the reasons for keeping a part of our constructor code serial. Therefore, the observed scalability is indeed weak, as depicted in Fig. 11,12. We see that the maximum speedup on both machines is $2.45\times$ on **M1** and $2.86\times$ on **M2**; this is approximately half than observed for the *SpMV*. We notice the best speedup for matrices *relat9* and *rail2586* on **M1**; *ppatents* and *parabolic_fem* on **M2**. In two cases (*neos* and *parabolic_fem* on **M1**) we notice a slow-down. Due to the increasingly loaded serial stage; in both cases, this happens after a no-subdivisions instantiation, for 1-core (for space reasons, we omit graphs with submatrix counts).

Relating constructor and *SpMV* times, we notice the constructor dominating the *SpMV*, in Fig. 13,14. We observe the maximal ratio for matrix *wb-edu* (up to $52.8\times$ on **M1**, up to $27.7\times$ on **M2**); a minimal one for matrix *rail2586* (from $2.8\times$ to $4.4\times$, on **M1**). In two cases (matrices *cont11_l*, *ppatents*), it happens that the constructor and *SpMV* times keep a similar pace (around $10\times$, on both machines). Indeed, the *SpMV* performance of matrix *cont11_l* does not increase with more cores, and matrix *ppatents* gets partitioned in the same number of leaf matrices, regardless the cores count. We notice worse ratios for bigger matrices: *cage15*, *wb-edu*, and *GL7d19*. Here, *ppatents* is big, but it performs *SpMV* exceptionally slow (see [5]).

Let us break down the constructor performance in the serial (*subdivision*) and parallel (*shuffle* and *conversion*–we will include this last one in the shuffle results, for convenience) stages. As discussed in Sec.IV, the subdivision code is expected to perform a number of passes on the input growing with the number of threads available for *SpMV*. In Fig. 15 and 16, we see the *scaling-down* of subdivision performance; we encounter a near-to 5-fold slow-down for matrices *parabolic_fem* and *neos*. It is due to no subdivision being performed in the 1-core case, on them. In the remaining cases, we do not notice more than a 2-fold slow-down.

In Fig. 17,18, we can see the growing gap between the subdivision and *SpMV*. For 1 or 2 cores, this ratio is always lower than 7.0, but for more, it can grow much: for matrix *wb-edu* on **M2**, the subdivision takes $5.1\times$ for 1 core, and up to 42.4 times *SpMV* time, for 8 cores.

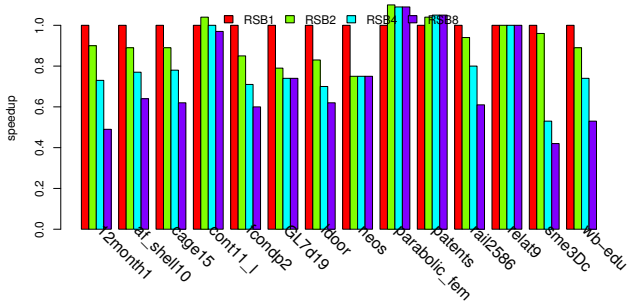

Fig. 16. Subdivision scaling on **M2**.

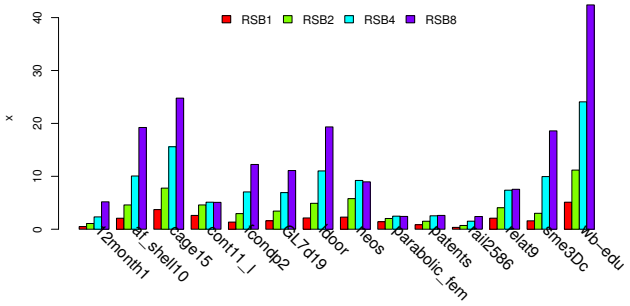

Fig. 18. Subdivision to *SpMV* time ratio on **M2**.

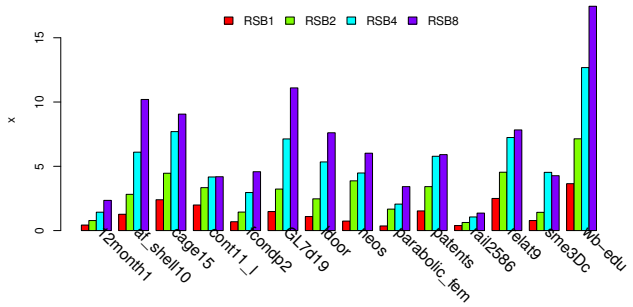

Fig. 17. Subdivision to *SpMV* time ratio on **M1**.

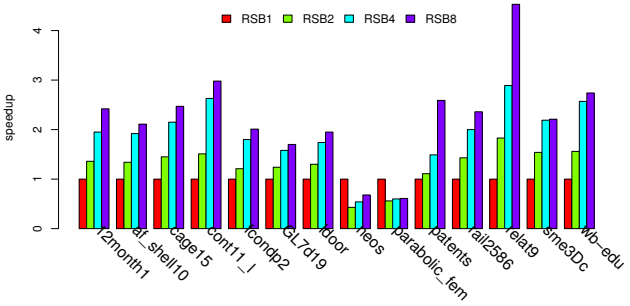

Fig. 19. Shuffle scaling on **M1**.

On the other hand, the *shuffle* stage scales quite regularly on all matrices in the test set; see Fig. 19, 20. Recall that, with higher cores counts, notwithstanding the growing number of submatrices to handle, the shuffle operation moves approximately the same amount of memory locations (see [5] for a discussion on indexing space). As noted in Sec. IV, in the shuffle (comprehensive of index compression) phase, the amount of involved traffic depends on the leaves format; prevalence of COO leaves will trigger more traffic; since compression happens after the copy operations, it contributes to additional traffic.

We also notice that the ratio of shuffle-to-*SpMV* times remains very close, regardless active cores count. This is satisfactory, because it indicates that both the operations scale similarly: see Fig. 21,22. Indeed, both operations seems to be memory bound; shuffle more than *SpMV*, as it doesn't involve floating point operations, which could be slower than integer operations. During stand-alone benchmarking our naive parallel `memcpy` wrapper (MEMCPY_Parallel, used in Sec. IV), we experienced at most $8.4GB/s$ on **M1**, $6.4GB/s$ on **M2**, and speedups respectively up to 3.1 and 2.2. We expect this limit to contribute with a relevant fraction to the shuffle stage.

By comparing, respectively, Fig. 17 to Fig. 21 and Fig. 18 to Fig. 22, we notice that on both machines, the subdivision (serial) stage becomes dominant over the shuffle (parallel) at around 4 active threads. Clearly, this situation is not desirable in the perspective of more computing cores, so we recognize the need for a scaling parallel subdivision stage. Also, by allowing degenerate subtrees (see Sec. IV) input could be scanned repeatedly and generating no new subdivision; this case should also be dealt with.

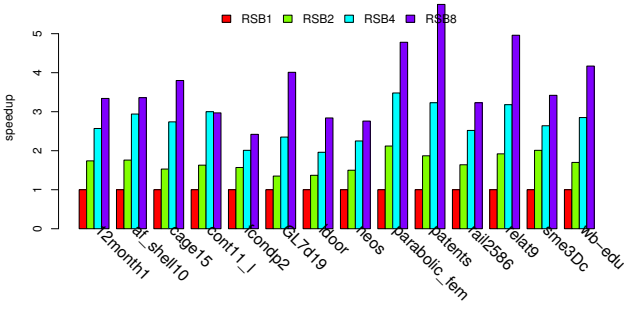

Fig. 20. Shuffle scaling on **M2**.

VII. Concluding Remarks

We have shown a multi-threaded algorithm for the instantiation of RSB matrices out of row major sorted COO arrays.

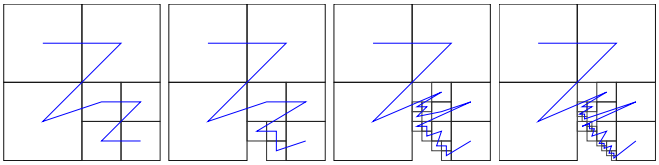

Fig. 23. Recursive subdivisions of matrix *cont11_l* for respectively 1,2,4,8 threads on **M1**. Notice the blue line joining (nonempty) leaf submatrices in the order they are stored in the RSB arrays. Notice that the more threads are active, the finer is the partitioning.

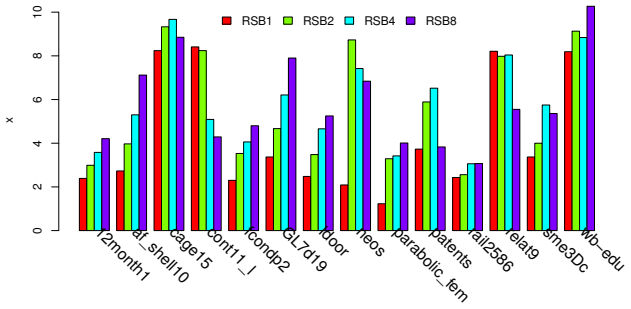

Fig. 21. Shuffle to *SpMV* time ratio on **M1**.

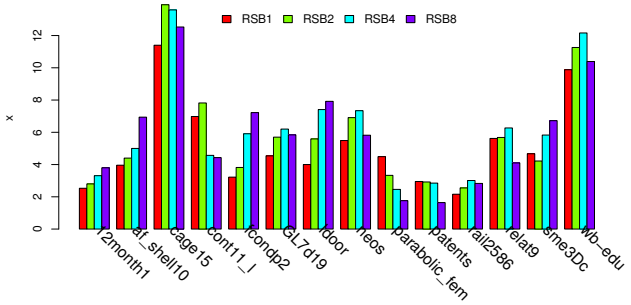

Fig. 22. Shuffle to *SpMV* time ratio on **M2**.

Experimentally, we established that its execution speed seems tightly bound to the peak memory bandwidth; even more than *SpMV*. Our procedure features a serial *subdivision* stage, where binary search and arrays copy operations are predominant, followed by a parallel *shuffle* stage, where arrays are displaced and indices adjusted. The shuffle stage scales smoothly; its performance seems strictly memory bandwidth-bound. While shaping the subdivision stage, we observed that an efficient parallel reformulation of it would require us to modify the definition of our format. We did not want to proceed this way as we wanted it to remain comparable with our earlier work, and so we have decided to leave the subdivision serial, for now. In practice, we observed the constructor-to-*SpMV* time ratio to be 1.6..15.1 times for 1 core, 2.5..22.4/2 cores, and 4.2..52.8/8 cores. Indeed, we have observed that the serial phase begins to dominate the constructor time as soon as at about 4 threads. For this reason, we recognize need of further research to develop a scalable, parallel algorithm to perform the initial subdivision, as this is the key to a scalable RSB matrix constructor. We deem also interesting to study parallel conversion/extraction mechanisms for interfacing to other formats, and consider the performance impact of building preconditioners, while solving linear systems. Of course, a number of trivial but effective optimizations (see Sec. IV) may also be applied.

REFERENCES

[1] I. S. Duff, M. A. Heroux, and R. Pozo, "The sparse BLAS," Tech. Rep., 2001.

[2] M. Martone, S. Filippone, S. Tucci, M. Paprzycki, and M. Ganzha, "Utilizing recursive storage in sparse matrix-vector multiplication - preliminary considerations," in *CATA*, T. Philips, Ed. ISCA, 2010, pp. 300–305.

[3] M. Martone, S. Filippone, M. Paprzycki, and S. Tucci, "On BLAS operations with recursively stored sparse matrices," in *Proceedings of the International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, September 2010.

[4] ——, "On the usage of 16 bit indices in recursively stored sparse matrices," in *Proceedings of the International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, September 2010.

[5] ——, "Use of hybrid recursive CSR/COO data structures in sparse matrices-vector multiplication," in *Proceedings of the International Multiconference on Computer Science and Information Technology*, October 2010.

[6] Y. Saad, *Iterative Methods for Sparse Linear Systems, 2nd edition*. Philadelphia, PA: SIAM, 2003.

[7] A. Buluc, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in *SPAA*, F. M. auf der Heide and M. A. Bender, Eds. ACM, 2009, pp. 233–244.

[8] 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*. Philadelphia, PA: SIAM, 1994.

[9] D. T. Nguyen, *Finite Element Methods: Parallel-Sparse Statics and Eigen-Solutions*. Springer US, 2006.

[10] V. Strassen, "Gaussian elimination is not optimal," *Numerische Mathematik*, vol. 13, no. 4, pp. 354– 356, 1969.

[11] G. von Fuchs, J. R. Roy, and E. Schrem, "Hypermatrix solution of large sets of symmetric positive-definite linear equations," *Computer Methods in Applied Mechanics and Engineering*, vol. 1, no. 2, pp. 197 – 216, 1972.

[12] A. C. McKellar and E. G. Coffman, Jr., "Organizing matrices and matrix operations for paged memory systems," *Commun. ACM*, vol. 12, no. 3, pp. 153–165, 1969.

[13] J. R. Herrero and J. J. Navarro, "Hypermatrix oriented supernode amalgamation," *The Journal of Supercomputing*, vol. 46, no. 1, pp. 84–104, Oct. 2008.

[14] B. Vastenhouw and R. H. Bisseling, "A two-dimensional data distribution method for parallel sparse matrix-vector multiplication," *SIAM Review*, no. 47, pp. 47–95, 2005.

[15] R. Fischer, M. Ast, J. Labarta, and H. Manz, "A dynamic task graph parallelization approach," in *Proceedings of IASS-IACM-2000: Fourth International Colloquium on Computation of Shell and Spatial Structures, June 4-7, 2000 in Chania-Crete, Greece*.

[16] A. Pinar and C. Aykanat, "Sparse matrix decomposition with optimal load balancing," in *High-Performance Computing, 1997. Proceedings. Fourth International Conference on*, 1997, pp. 224 – 229.

[17] P. Gottschling and D. Lindbo, "Generic compressed sparse matrix insertion: algorithms and implementations in MTL4 and FEniCS," in *POOSC '09: Proceedings of the 8th workshop on Parallel/High-Performance Object-Oriented Scientific Computing*. ACM, 2009, pp. 1–8.

[18] D. E. Knuth, *The art of computer programming, volume 1 (3rd ed.): fundamental algorithms*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1997.

[19] T. Davis, "University of Florida sparse matrix collection," *ACM Transactions on Mathematical Software, to appear*, 2010.

[20] "Standard for information technology— portable operating system interface (POSIX) (IEEE std 1003.1)," 2008.