Two-dimensional cache-oblivious sparse matrix–vector multiplication

A.N. Yzelman[a], Rob H. Bisseling[a]

[a] *Mathematical Institute, Utrecht University, P.O. Box 80010, 3508 TA Utrecht, The Netherlands*

Abstract

In earlier work, we presented a one-dimensional cache-oblivious sparse matrix–vector (SpMV) multiplication scheme which has its roots in one-dimensional sparse matrix partitioning. Partitioning is often used in distributed-memory parallel computing for the SpMV multiplication, an important kernel in many applications. A logical extension is to move towards using a two-dimensional partitioning. In this paper, we present our research in this direction, extending the one-dimensional method for cache-oblivious SpMV multiplication to two dimensions, while still allowing only row and column permutations on the sparse input matrix. This extension may require a generalisation of the Compressed Row Storage data structure, to a block-based data structure. Experiments performed on three different architectures show further improvements of the two-dimensional method compared to the one-dimensional method, especially in those cases where the one-dimensional method already provided significant gains. The largest gain obtained by our new reordering is over a factor of 3 in SpMV speed, compared to the natural matrix ordering.

Keywords: matrix–vector multiplication, sparse matrix, parallel computing, recursive bipartitioning, fine-grain, cache-oblivious
2010 MSC: 65F50, 65Y20, 05C65, 65Y04

1. Introduction

Our earlier work [20] presents a cache-oblivious method to reorder arbitrary sparse matrices so that performance increases during sequential sparse matrix–vector (SpMV) multiplication $\mathbf{y} = A\mathbf{x}$. Here, A is a sparse matrix, $\mathbf{x}$ the input vector, and $\mathbf{y}$ the output vector. This method is based on a one-dimensional (1D) scheme for partitioning a sparse matrix, with the goal of efficiently parallelising the SpMV multiply. Today, parallel applications are moving towards using two-dimensional (2D) partitioning methods in preference over 1D methods. In this paper, we show that we can use the better-quality 2D partitioning in sequential SpMV multiplication as well, in some instances gaining additional factors over the original 1D method in terms of SpMV efficiency.

Email addresses: A.N.Yzelman@uu.nl (A.N. Yzelman), R.H.Bisseling@uu.nl (Rob H. Bisseling)

We first give a brief introduction to cache architectures related to SpMV multiplication. A *cache* is a fast but small computer memory which can be used to speed up calculations by keeping repeatedly accessed data close to the CPU. Modern computer architectures have several layers of cache, from the fastest and smallest L1 cache to larger but slower L2 or L3 caches. In dense linear algebra, algorithms are often reformulated in terms of matrix–matrix multiplication to benefit from the reuse of data; in sparse linear algebra, however, exploiting the cache is much harder. In particular, the SpMV multiplication is notorious: each element of the sparse matrix is used only once, and repeated access can only be made for vector data. The key to achieving the best performance is then to access the matrix elements in an order that keeps the associated vector components in cache as long as possible [8, 13]. If a data element is accessed while it resides in cache, a *cache hit* occurs; if the element did not reside in cache, a *cache miss* occurs. The first time a cache miss occurs on a particular data element, the miss is *compulsory*; bringing the element into cache at least once cannot be avoided. Subsequent misses of the same element are *non-compulsory*, as they might be avoided. For a further exposition of cache architectures, see, e.g., van der Pas [15].

Our goal is to obtain the best performance without knowledge of the cache parameters, such as cache size, line size, et cetera. This approach was introduced by Frigo et al. and is called *cache-oblivious* [5]. The advantage of such methods is that they work irrespective of hardware details, which may be intricate and can vary tremendously from machine to machine. Cache-oblivious approaches are often based on recursion, as this enables subsequent decreases of the problem size until the problem fits in cache. In the case of SpMV multiplication, permuting the rows and columns of the matrix A, while permuting the vectors $\mathbf{x}$ and $\mathbf{y}$ accordingly, can be done in a cache-oblivious way to improve cache use. A further improvement is changing the order of access to the individual nonzero elements of A. Both of these methods are explored in this work.

The organisation of this paper is as follows: we first proceed with briefly explaining the 1D method in Section 1.1 and presenting related work in Section 1.2, and immediately follow up with the extension to 2D in Section 2. These methods are subjected to numerical experiments in Section 3. We draw our conclusions in Section 4.

1.1. The one-dimensional scheme

The sparsity structure of an $m \times n$ matrix A can be modelled by a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ using the *row-net* model, which will briefly be described here; for a broader introduction, see Çatalyürek et al. [2]. The columns of A are modelled by the vertices in $\mathcal{V}$, and the rows by the nets (or hyperedges) in $\mathcal{N}$, where a net is a subset of the vertices. Each net contains precisely those vertices (i.e., columns) that have a nonzero in the corresponding row of A. A *partitioning* of a matrix into p parts is a partitioning of $\mathcal{V}$ into nonempty subsets $\mathcal{V}_0, \ldots, \mathcal{V}_{p-1}$, with each pair of subsets disjoint and $\cup_j \mathcal{V}_j = \mathcal{V}$. Given such a partitioning, the *connectivity* λ_i of a net n_i in $\mathcal{N}$ can be defined by

$$\lambda_i = |\{\mathcal{V}_j | \mathcal{V}_j \cap n_i \neq \emptyset\}|,$$

that is, the number of parts the given net spans. Note that this model is 1D in the sense that only the columns are partitioned, and not the rows. Such a partitioning can be used to obtain a sequential cache-oblivious sparse matrix–vector (SpMV) multiplication scheme [20], if the partitioning is constructed in the following iterative way.

First, the algorithm starts with a partitioning consisting of a single part, namely the complete $\mathcal{V}$. A single iteration of the partitioner then *extends* any given partitioning of k parts to $k+1$ parts by splitting a chosen part $\mathcal{V}_i$ ($i \in [0, k-1]$) into two, yielding a new partitioning $\mathcal{V}'_0, \ldots, \mathcal{V}'_k$. The goal of the partitioner is to obtain, after $p-1$ iterations, a final partitioning for which the load balance constraint $\mathrm{nz}(\mathcal{V}_j) \leq (1+\epsilon)\frac{\mathrm{nz}(A)}{p}$ for all $j \in [0, p-1]$ holds, with the following quantity minimised:

$$\sum_{i:n_i \in \mathcal{N}} (\lambda_i - 1). \tag{1}$$

After each iteration that splits a subset of $\mathcal{V}$ into two parts, $\mathcal{V}_{\text{left}}$ and $\mathcal{V}_{\text{right}}$, the hypergraph nets can be divided into three categories:

- $\mathcal{N}_-$: nets containing only nonzeroes from $\mathcal{V}_{\text{left}}$,
- $\mathcal{N}_c$: nets containing nonzeroes from both $\mathcal{V}_{\text{left}}$ and $\mathcal{V}_{\text{right}}$ (the *cut* nets), and
- $\mathcal{N}_+$: nets containing only nonzeroes from $\mathcal{V}_{\text{right}}$.

The key idea of the cache-oblivious SpMV multiplication is to apply row and column permutations according to the data available after each iteration. Having defined these categories, we can now reorder the rows of the matrix as follows. All rows in $\mathcal{N}_+$ are permuted towards the bottom of the matrix, whereas those in $\mathcal{N}_-$ are permuted towards the top. The remaining rows are left in the middle. This creates two row-wise *boundaries* in the matrix, i.e., dividing lines between blocks of rows. The columns are permuted according to the split of the vertex subset: columns in $\mathcal{V}_{\text{left}}$ are permuted to the left, and the others, from $\mathcal{V}_{\text{right}}$, to the right. Here, a single column boundary appears. Altogether, the boundaries split the permuted matrix into six blocks. The two blocks corresponding to $\mathcal{N}_c \cap \mathcal{V}_{\text{left}}$ and $\mathcal{N}_c \cap \mathcal{V}_{\text{right}}$ are referred to as *separator blocks*. The resulting form of the permuted sparse matrix is called *Separated Block Diagonal* (SBD).

To preserve the existing SBD structure, it is important that boundaries already created between blocks are not violated in subsequent permutations and subdivisions. Thus, a row that needs to be permuted towards the top of the matrix, moves towards its closest boundary in the upper direction. Note that this need not be the same boundary for all rows in the same net category. A similar restriction applies to column permutations. After $p-1$ iterations of this scheme, the row and column permutations applied to A can be written using two permutation matrices P, Q such that PAQ corresponds to the permuted matrix. Note that this permutation generally is unsymmetric. An example of a 1D partitioning and permutation according to this scheme can be found in Figure 1(right).

Every subset $\mathcal{V}_i$ contains a set of consecutive matrix columns of PAQ. When multiplying this matrix with a dense vector $\mathbf{x}$, the $\mathcal{V}_i$ thus correspond to small ranges of this input vector. The key point is that if these ranges fit into cache, any cache misses are incurred only on the rows that span multiple parts $\mathcal{V}_i$, provided the reordered matrix is stored row-by-row such as with the well-known compressed row storage (CRS) data structure. In fact, when $\max|\mathcal{V}_i|$ elements from the input and output vector *exactly* fit into cache, when the rows in the cut nets are dense enough, and when the Zig-zag CRS data structure (ZZ-CRS, as depicted by the dashed curve in Figure 2, right) is used, an upper bound on the cache misses incurred is $\sum_i(\lambda_i - 1)$ [20, Section 5.2], which is exactly

the quantity minimised by the partitioner. Applying further iterations of the partitioner, thus further decreasing the corresponding range in the vector $\mathbf{x}$, theoretically does not harm cache efficiency; this method is therefore cache-oblivious as it can be applied iteratively as far as possible and is then still expected to yield good performance during SpMV multiplication.

From the hypergraph, a *separator tree* can be constructed during the partitioning; this will be a useful tool during analysis in Section 2. After the first iteration, the pair $(\mathcal{V}, \mathcal{N})$, with $\mathcal{V}$ the vertex set that was partitioned, becomes the root of this tree. This root has two children, the left node which contains the vertices and nets in $(\mathcal{V}_{\text{left}}, \mathcal{N}_{-})$, and the right node $(\mathcal{V}_{\text{right}}, \mathcal{N}_{+})$. As the partitioner works recursively on $\mathcal{V}_{\text{left}}$ and $\mathcal{V}_{\text{right}}$, these left and right nodes are replaced with subtrees, the roots of which are constructed recursively in the same way as described above. The resulting tree will be binary. Usually, depending on the partitioner, the depths of the leaf nodes differ by at most 1, so that the tree is balanced.

1.2. Related work

Since we are now making our way into 2D techniques for improving cache locality, other recent works become relevant, beyond those referred to in our earlier paper [20]. Previously, cache locality in the input vector was improved, and the locality of the output vector was never in question thanks to the (ZZ-)CRS ordering imposed on the nonzeroes, resulting in linear access of the output vector. This also enabled the use of auto-tuning methods like OSKI, introduced by Vuduc et al. [17], in conjunction with our reordering method. A 2D method, however, tries to increase locality in the output vector as well, in the hope that the obtained locality in both dimensions (input and output) can outperform locality induced in only one of the dimensions. The objective is thus to minimise the sum of cache misses within both the input and output vectors, and this entails breaking the linear access to the output vector.

In dense linear algebra, the 2D approach has been successfully applied by *blocking*, the process of dividing the matrix into submatrices (or a hierarchy thereof) of appropriate sizes so that matrix operations executed in succession on these submatrices are faster than if they were performed on the single larger sparse matrix [6, 14, 18]. By using a Morton ordering (i.e., a recursive Z-like ordering) [11], the blocks can be reordered to gain even more efficiency; additionally, the Morton ordering can also be used on high-level blocks, while the low-level blocks still use row-major data structures such as standard CRS. This gives rise to so called hybrid-Morton data structures which, like the method presented here, enable the use of specialised BLAS libraries on the low-level blocks; see Lorton and Wise [10]. Similar techniques for sparse matrices include sparse blocking into very small blocks [12], and using the Hilbert space-filling curve on the nonzeroes of sparse matrices [7, 19]. Our method presented in the next section is much in the same locality-enhancing spirit, even though our approach is different.

2. The two-dimensional scheme

The straightforward extension of the original scheme to 2D is natural when employing the fine-grain model [3], which is briefly described as follows. The sparse matrix A is again modelled by a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$, but now such that each nonzero a_{ij} corresponds

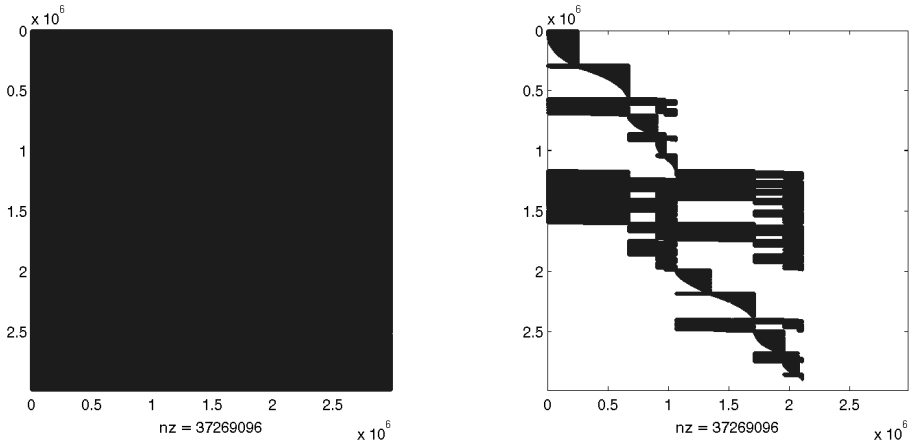


Figure 1: Spy plots of the wikipedia-2006 matrix, partitioned and reordered to SBD form using the Mondriaan software [16]. The matrix has 2983494 rows and columns, and 37269096 nonzeroes. The left spy plot shows the highly unstructured original link matrix. This matrix appears to be dense because of the size of the nonzero markers and the rather even spread of the nonzeroes; in fact every matrix row contains only 12.5 nonzeroes on average. The right spy plot shows the result after 1D partitioning with $p = 10$ and the load-imbalance parameter $\epsilon = 0.1$. Empty rows and columns correspond to web pages without incoming or outgoing links, respectively.

to a vertex in $\mathcal{V}$. Each row and column is modelled by a net in $\mathcal{N}$. A net contains exactly those nonzeroes that appear in its row or column. For each net, the connectivity λ_i over a partitioning $\mathcal{V}_0, \ldots, \mathcal{V}_{p-1}$ of $\mathcal{V}$ can still be defined as before. This enables us to define the set of cut row nets $\mathcal{N}_c^{\text{row}} \subset \mathcal{N}$, consisting of the row nets with connectivity larger than one, which have vertices (nonzeroes) in two or more parts of the partitioning, as well as the set $\mathcal{N}_c^{\text{col}}$, similar to the 1D case. Row and column permutations for $p = 2$ can then be used to bring the arbitrary sparse input matrix into the form depicted in Figure 2(left), the *doubly separated block diagonal* (DSBD) form. Note that there are now five different separator blocks, namely $\mathcal{N}_c^{\text{row}} \cap \mathcal{N}_c^{\text{col}}$, $\mathcal{N}_c^{\text{row}} \cap \mathcal{N}_{\pm}^{\text{col}}$, and $\mathcal{N}_{\pm}^{\text{row}} \cap \mathcal{N}_c^{\text{col}}$. These together form a *separator cross*, coloured red in Figure 2(left). An example of a matrix in DSBD form obtained using this fine-grain scheme can be found in Figure 3(left).

This 2D scheme can be applied recursively. However, using a CRS data structure will result in *more* cache misses for $p > 2$ due to the column-wise separator blocks, if the separator blocks are relatively dense; see Figure 2(right). Nonzeroes thus are better processed block by block in a suitable order, and the data structure used must support this. These demands are treated separately in Section 2.1 and Section 2.2, respectively.

A separator tree can be defined in the 2D case, similar to the 1D case, but now with nodes containing nets corresponding to both matrix rows and columns. Internal nodes in the tree contain both cut rows and columns belonging to the separator crosses. Leaf nodes correspond to the p non-separator blocks. An example of a separator tree in the case of $p = 4$ is shown in Figure 4.

2.1. SBD block order

Figure 5 shows various ways of ordering the DSBD blocks. The main objective is to visit the same regions of the input and output vectors as few times as possible. Accesses

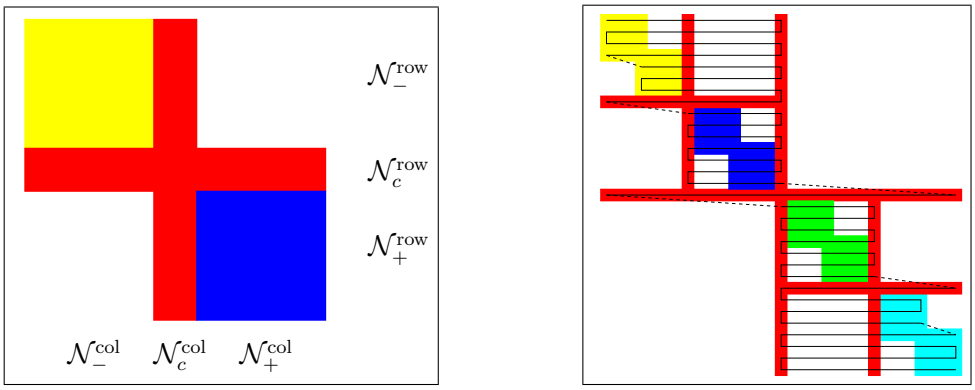


Figure 2: Schematic view of a 2D matrix reordering using the fine-grain model, for $p = 2$ (left) and $p = 4$ (right). The figure on the right side also includes the ZZ-CRS ordering curve.

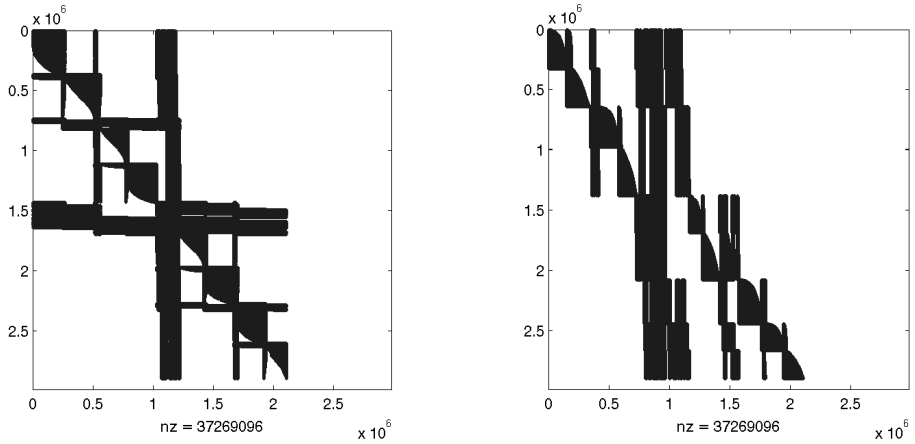


Figure 3: Spy plots of the wikipedia-2006 matrix, like Figure 1, but now using 2D partitioning. The left picture shows the result using the fine-grain scheme with $p = 8$. On the right, the Mondriaan scheme with $p = 9$ was used. The load imbalance parameter is set to 0.1 in both cases.

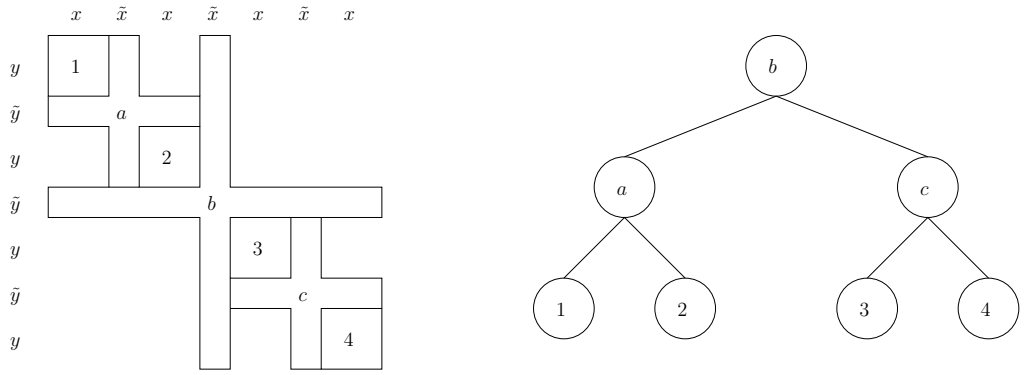


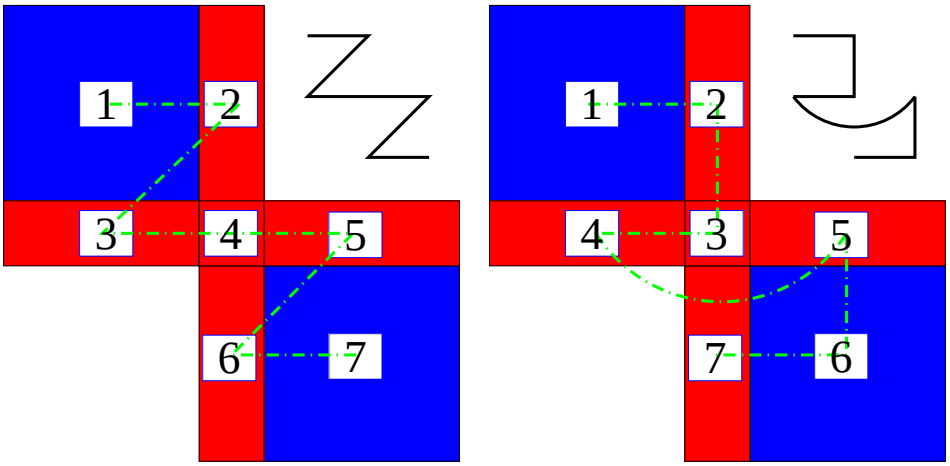
Figure 4: Illustration of a DSBD form with $p = 4$, and its corresponding separator tree. The sizes x, y of the $y \times x$ non-separator blocks (1, 2, 3, 4) are also shown in the picture, as are the small widths of the separator cross, given by $\tilde{y}$ and $\tilde{x}$.

in the vertical direction represent writes on the output vector, whereas horizontal accesses represent reads on the input vector. Write accesses are potentially more expensive; the block ordering avoiding the most unnecessary irregular accesses in the vertical direction thus theoretically performs best. From the orderings in Figure 5, this best performing block order is ZZ-CCS.

When considering suitable data structures for the vertical separator blocks, CRS is the perfect ordering for the nonzeroes within a block; the width in the column direction is typically small by grace of good partitioning, so that the small range from the input vector fits into cache. Since rows are treated one after the other, access to the output vector is regular, even linear, so cache efficiency with regards to the output vector is good as well. This changes for the horizontal separator blocks: there, the range in the output vector is limited and fits into cache, but the range of the input vector is large and access is in general completely irregular. Demanding instead that the horizontal separator blocks use the CCS ordering for individual nonzeroes, increases performance to mirror that of the vertical separator blocks: input vector accesses then are linear, and output vector accesses are limited to a small range typically fitting into cache. Note that this scheme can be viewed as a 2D sparse blocking method, and that it also can be applied to the original 1D method.

2.2. Block data structures

Three data structures will be introduced here: Incremental CRS (ICRS), Bi-directional Incremental CRS (BICRS), and a simple block-based data structure (Block). With the ICRS storage scheme [9], the main idea is to store for each nonzero, instead of its column index, the *difference* between its column index and that of the previous nonzero in the row. During the SpMV multiplication, a pointer to the input vector then needs to be incremented with the difference, an overflow of this pointer indicating a row change. The pointer overflows such that subtracting the number of columns n afterwards yields the starting element of the input vector for the next row. This next row is determined by using a *row increment* array, similar to the column increment array, thus replacing and avoiding consultation of the traditional CRS array controlling which row corresponds to



(a) CRS block ordering

(b) Adapted CRS block ordering

(c) Separators-last block ordering

(d) ZZ-CCS block ordering

Figure 5: Various possible DSBD orderings for SpMV multiplication for $p = 2$. The black curve in the upper right corners of the panels is included to highlight the curve trajectory.

which range of nonzeroes. The hope is that this alternative implementation uses less instructions for the SpMV kernel and reduces the data structure overhead per row; instead of looping over each, possibly empty, row of the sparse matrix, ICRS jumps only to the nonempty rows, therefore avoiding unnecessary overhead. This is particularly useful for the separator blocks we encounter after 2D partitioning; there, many empty rows (columns) are found in the case of vertical (horizontal) separator blocks. Note that on the other hand, if there are no empty rows in A, the row increment array does not have to be stored at all (since all increments would equal one), thus further simplifying the SpMV kernel and yielding an additional gain in efficiency.

An option is to store each matrix block in a separate (I)CRS or (I)CCS data structure. Upon performing an SpMV multiplication, the multiplication routines of the separate data structures are called in the order defined by the block order, on the same input and output vectors; this straightforwardly yields the *simple block-based data structure*. However, calling several multiplications in sequence this way incurs some overhead: when a switch between blocks is made, the pointers to the input and output vectors are reset to the location of the first nonzero of the next block, and only then the actual SpMV kernel is called. Also, the storage requirements increase with p: when using non-incremental CRS/CCS storage scales as $\mathcal{O}((m+n)\log_2 p)$, and when using the incremental ICRS/ICCS data structure, storage is in between $2\text{nz}+m$ (best case) and 3nz (worst case). Hence the gain with respect to cache efficiency of using this simple block-based structure must be larger than the penalty incurred by this additional overhead; and this overhead can be reduced by lowering the number of blocks required.

Instead of this simple block-based data structure, more advanced ideas may be exploited here, such as a scheme which also compresses index (or increment) values as presented by Buluç et al. [1]; although some additional effort would be required to make this work within our scheme. An alternative is provided by a new data structure. The Bi-directional Incremental CRS [19] is a simple extension to the ICRS data structure, which allows any (non-CRS) ordering of the nonzeroes by allowing negative increment values. Trivially, negative column increments enable jumping back and forth within a single row. Overflows in the column direction still signal row changes, and by allowing negative row increments, bi-directional row changes are made possible as well. Each column overflow implies a row jump, and each such jump has an instruction and storage overhead; note that hence the orientation (BICRS or BICCS) of the data structure still matters. The gain is that this additional overhead is only dependent on the number of row jumps j, and not directly on the number of parts p. Furthermore, we have that $m \leq j \leq \text{nz}$, and the performance of BICRS is guaranteed to be in between that of ICRS and the Triplet data structure.

2.3. The Mondriaan partitioning scheme

Modelling a sparse matrix as a hypergraph using the 2D fine-grain model has two drawbacks: the increased size of the hypergraph compared to the simpler 1D row-net model, and the large number of blocks after permutation. The larger hypergraph leads to an increased partitioning time, hopefully justified by the increased quality of the 2D partitioning, whereas the large number of separator blocks incur additional overhead as was shown in the previous subsection. An alternative 2D method exists, the *Mondriaan scheme*, which results in less partitioning time and fewer separator blocks. It is imple-

mented in the Mondriaan sparse matrix partitioner software [16], and combines two 1D methods as follows.

Apart from the row-net model, a column-net model can also be defined, identical to the row-net model but with the roles of rows and columns reversed: each row corresponds to a vertex in the hypergraph, and each column to a net. A cheaper way of obtaining a 2D partitioning for $p > 2$, then is to use a partitioner as described in Section 1.1, with the following modification: during each iteration, both the hypergraphs corresponding to a row-net and column-net representation are partitioned, and the solution yielding the lowest cost, as given by Equation (1), is chosen. In the next iteration, again both models are tried and the best is chosen; hence splits in both dimensions (row-wise and column-wise) are possible during partitioning, but never both during the same iteration. An example of a DSBD partitioning obtained using this scheme is shown in Figure 3(right).

It is easily seen that every solution arising from bipartitioning a row-net or column-net hypergraph corresponds to a very specific solution in the fine-grain hypergraph, namely the solution in which vertices are grouped by column or row as they are partitioned. This means that the partitioning method based on the Mondriaan scheme can be represented by a fine-grain hypergraph throughout the partitioning method; hence the separator tree and the permutation strategy still work as presented.

It is worthwhile to exploit the form of the 2D DSBD ordering in the special case of bipartitioning by the row-net or column-net model. The 1D row-net model yields the picture in Figure 2(left) with the centre block and the two vertical rectangular blocks removed from the separator cross (this corresponds to the original SBD form in [20]). The column-net model is similar, having instead the centre block and the two horizontal rectangular blocks removed. As such, using the Mondriaan scheme instead of the fine-grain scheme in full recursive bipartitioning, nearly halves the number of blocks, from $p+5(p-1) = 6p-5$ to $p+2(p-1) = 3p-2$ blocks. Hence, if the quality of partitioning by the fine-grain and Mondriaan schemes is similar, the Mondriaan scheme is expected to perform better because there are fewer blocks.

2.4. Cache performance in recursion

In this section, we will analyse the cache behaviour of the SpMV kernel for different block orderings. Let us make some simplifying assumptions:

1. the storage scheme within each sparse block on the diagonal is ICRS,
2. the storage scheme for horizontally oriented off-diagonal separator blocks is ICCS,
3. the storage scheme for vertically oriented off-diagonal separator blocks is ICRS,
4. the number of parts in the partitioning is a power of two,
5. the horizontal block sizes (corresponding to a part of the input vector) are equal and are denoted by x,
6. the maximum of the column separator widths is $\tilde{x}$,
7. the vertical block sizes (corresponding to a part of the output vector) are equal and are denoted by y, and
8. the maximum of the row separator widths is $\tilde{y}$;

see also Figure 4(left).

We calculate cache misses by analysing the binary separator tree, starting with the root node, which contains the largest separator cross (spanning the entire sparse matrix).

The two children of the root node contain the remaining two sparse blocks that the partitioner has been recurring on. We only count non-compulsory cache misses, because the compulsory ones, related to the first time data are accessed, are the same for all orderings. In our analysis, we ignore data that are brought into cache because they are in the same cache line as data that is used. Our analysis only pertains to the input and output vector data, since the matrix data are used only once and hence cannot benefit from the cache.

The number of non-compulsory cache misses can be expressed as a function on the root node, which recurs on its two children, et cetera. In fact, only the height of the nodes will be required to calculate the cache misses: for each block ordering a function $f(i)$ will be constructed, which gives an upper bound on the number of non-compulsory cache misses for a node with height $i + 1$. The height of the leaves is 0 by definition, hence an internal node of height 1 corresponds to a subpartitioning with $p = 2$, which we take as the base case $f(0)$. See also Figure 4. Note that by the power-of-two assumption, the separator tree is complete.

A single function f cannot be used to predict the cache misses for each node in every situation. Cache behaviour may differ between nodes as in some cases the elements in the input vector were already processed, thus making all accesses in the horizontal direction non-compulsory. This happens for example in Figure 5a, when block 3 is processed. In general, there are four different possibilities, each modelled by a separate function:

1. f_0: no previous accesses in the corresponding input and output subvector have been made,
2. f_1: the corresponding input subvector has been brought into cache before,
3. f_2: the corresponding output subvector has been brought into cache before,
4. f_3: both the corresponding input and output subvectors have been brought into cache before.

If $k + 1$ is the height of the root node, $f_0(k)$ will give an upper bound on the total theoretical number of non-compulsory cache misses.

2.4.1. CRS block ordering

First, we analyse the CRS block ordering (Figure 5a). In this case, $f_0(0) = 2(x + \tilde{x})$, and $f_3(0) = 4x + 3\tilde{x} + 2y + \tilde{y}$. This can be understood by looking at the figure as follows. Assume that no previous accesses have been made to the vectors. For block 1, x accesses to the input vector $\mathbf{x}$ are needed and y to the output vector $\mathbf{y}$. Since this is the first time access is needed, both accesses are compulsory cache misses. For block 2, y data are kept in cache, and $\tilde{x}$ are brought in for the first time, hence causing $\tilde{x}$ compulsory misses. For block 3, x data are brought into cache again, with x non-compulsory misses, and we also have $\tilde{y}$ compulsory misses. For block 4, $\tilde{x}$ non-compulsory accesses are made, and $\tilde{y}$ data remain in cache. For block 5, we have x compulsory misses, and $\tilde{y}$ data remain in cache. For block 6, we have $\tilde{x}$ non-compulsory misses, and y compulsory ones. For block 7, we have x non-compulsory misses, and y data remain in cache. Summing the non-compulsory misses for all the blocks, we obtain $2(x + \tilde{x})$, which becomes the value of $f_0(0)$. Now, assume all vector data have been accessed before. This means that all $2x + \tilde{x} + 2y + \tilde{y}$ compulsory misses become non-compulsory, giving a total of $f_3(0) = 4x + 3\tilde{x} + 2y + \tilde{y}$ non-compulsory misses.

If we carry out the block ordering recursively to obtain $f_0(1)$, blocks 1 and 7 are further refined. Assuming no previous accesses have been made, we get the following number of non-compulsory misses for blocks 1 to 7: $f_0(0)$, $2y+\tilde{y}$, $2x+\tilde{x}$, $\tilde{x}$, 0, $\tilde{x}$, $f_3(0)$; see also Figure 4. This yields a total of $f_0(1) = f_0(0) + f_3(0) + 2(x+y) + 3\tilde{x} + \tilde{y}$. Note that the expressions f_1 and f_2 do not appear here. A similar analysis can be made for $f_3(1)$, and also here f_1 and f_2 do not appear. Similarly, in full recursion, the upper bounds become:

$$\begin{aligned} f_0(i) &= f_0(i-1) + f_3(i-1) + 2^i(x+y) + (2^i+1)\tilde{x} + (2^i-1)\tilde{y}, \\ f_3(i) &= 2 \cdot f_3(i-1) + 2^{i+1}(x+y) + (2^{i+1}+1)\tilde{x} + (2^{i+1}-1)\tilde{y}. \end{aligned}$$

By using $f_3(0) - f_0(0) = 2(x+y) + \tilde{x} + \tilde{y}$, and

$$(f_3 - f_0)(i) = (f_3 - f_0)(i-1) + 2^i(x+y+\tilde{x}+\tilde{y}),$$

direct formulae can be obtained:

$$\begin{aligned} f_3(i) &= i2^{i+1}(x+y) + ((i+{}^1\!/_2)2^{i+1} - 1)\tilde{x} + ((i-{}^1\!/_2)2^{i+1} + 1)\tilde{y} + 2^i f_3(0), \\ (f_3 - f_0)(i) &= 2^{i+1}(x+y) + (2^{i+1}-1)(\tilde{x}+\tilde{y}), \\ f_0(i) &= f_3(i) - (f_3 - f_0)(i) \\ &= (i+1)2^{i+1}x + i2^{i+1}y + (i+1)2^{i+1}\tilde{x} + ((i-1)2^{i+1} + 2)\tilde{y}. \end{aligned}$$

Hence, when using this block ordering throughout the recursion, more cache misses occur on the input vector. This formula will enable direct comparison to other block orderings.

2.4.2. Zig-zag CCS block ordering

This order corresponds to the ordering shown in Figure 5d. Analysing this form, all four functions $f_0, \ldots, f_3$ are required as follows: $f_0(0) = 2\tilde{y}$, $f_1(0) = 2(x+\tilde{y}) + \tilde{x}$, $f_2(0) = 2y + 3\tilde{y}$, $f_3(0) = 2(x+y) + \tilde{x} + 3\tilde{y}$, and

$$\begin{aligned} f_0(i) &= f_1(i-1) + f_2(i-1) + 2^i(x+y) + (2^i-1)\tilde{x} + (2^i+1)\tilde{y}, \\ f_1(i) &= f_1(i-1) + f_3(i-1) + 2^{i+1}x + 2^i y + (2^{i+1}-1)\tilde{x} + (2^i+1)\tilde{y}, \\ f_2(i) &= f_2(i-1) + f_3(i-1) + 2^i x + 2^{i+1}y + (2^i-1)\tilde{x} + (2^{i+1}+1)\tilde{y}, \\ f_3(i) &= 2 \cdot f_3(i-1) + 2^{i+1}(x+y) + (2^{i+1}-1)\tilde{x} + (2^{i+1}+1)\tilde{y}. \end{aligned}$$

Note that:

$$\begin{aligned} (f_3 - f_0)(i) &= (2f_3 - f_2 - f_1)(i-1) + 2^i(x+y+\tilde{x}+\tilde{y}), \\ (f_3 - f_1)(i) &= (f_3 - f_1)(i-1) + 2^i(y+\tilde{y}), \\ (f_3 - f_2)(i) &= (f_3 - f_2)(i-1) + 2^i(x+\tilde{x}). \end{aligned}$$

A direct formula for f_3 can be obtained:

$$\begin{aligned} f_3(i) &= i2^{i+1}(x+y) + ((i-{}^1\!/_2)2^{i+1} + 1)\tilde{x} + ((i+{}^1\!/_2)2^{i+1} - 1)\tilde{y} + 2^i f_3(0) \\ &= (i+1)2^{i+1}(x+y) + (i2^{i+1}+1)\tilde{x} + ((i+2)2^{i+1} - 1)\tilde{y}, \end{aligned}$$

and similarly for the above difference formulae,

$$\begin{aligned}(f_3 - f_1)(i) &= (2^{i+1} - 2)(y + \tilde{y}) + (f_3 - f_1)(0)\\ &= 2^{i+1}y + (2^{i+1} - 1)\tilde{y},\\ (f_3 - f_2)(i) &= (2^{i+1} - 2)(x + \tilde{x}) + (f_3 - f_2)(0)\\ &= 2^{i+1}x + (2^{i+1} - 1)\tilde{x},\\ (f_3 - f_0)(i) &= 2^{i+1}(x + y) + (2^{i+1} - 1)(\tilde{x} + \tilde{y}).\end{aligned}$$

This leads us to the following final form:

$$f_0(i) = f_3(i) - (f_3 - f_0)(i) = i2^{i+1}(x + y) + ((i - 1)2^{i+1} + 2)\tilde{x} + (i + 1)2^{i+1}\tilde{y}. \quad (2)$$

The difference in non-compulsory number of cache misses between the CRS block order and the ZZ-CCS block order is given by $2^{i+1}x + (2^{i+2} - 2)\tilde{x} + (2 - 2^{i+2})\tilde{y}$; hence asymptotically, the ZZ-CCS block order is more efficient than the CRS block order when $x + 2\tilde{x} > 2\tilde{y}$; assuming $\tilde{x} = \tilde{y}$, we may conclude that ZZ-CCS *always* is preferable to the CRS block order.

The expected cache misses for the ACRS and the Separators-last block ordering can be obtained in similar fashion; for brevity, only the final forms are given below:

$$\begin{aligned}f_{\mathrm{ACRS}}(i) &= (i + {}^1\!/_2)2^{i+1}x + i2^{i+1}(y + \tilde{x}) + ((i - 1)2^{i+1} + 2)\tilde{y},\\ f_{\mathrm{SepLast}}(i) &= (i + {}^1\!/_2)2^{i+1}x + (i + 1)2^{i+1}y + ((i - 1)2^{i+1} + 2)\tilde{x} + ((i - {}^1\!/_2)2^{i+1} + 1)\tilde{y}.\end{aligned}$$

If the partitioning works well, meaning that $\tilde{x} \ll x$ (and $\tilde{y} \ll y$), then the ZZ-CCS block order is superior to the ACRS block order, which in turn is better than the CRS block order. The separators-last block order is expected to perform worst. Note that for large i, the relative differences in cache misses become small; this may become apparent especially if the matrix size is much larger than the cache size.

3. Experimental results

Experiments for SpMV multiplication have been performed on three different architectures. The first is a supercomputer, called Huygens, which consists of 104 nodes each containing 16 dual-core IBM Power6+ processors. For all experiments, one node was reserved to perform the sequential SpMV multiplications (on a single core) without interference from other processes. The Power6+ processor has a speed of 4.7GHz per core, an L1 cache of 64kB (data) per core, an L2 cache of 4MB semi-shared between the cores, and an L3 cache of 32MB per processor. This means that an SpMV multiplication on a matrix with $m + n = 8192$ would fit entirely into the L1 cache, assuming the vector entries are stored in double precision. The same applies with $m + n = 524288$ for the L2 cache and $m + n = 4194304$ for the L3 cache.

The second architecture is the Intel Core 2 Q6600, which is a quad-core processor with four 32kB L1 data caches, and two 4MB L2 caches each shared among two cores. Each core runs at the speed of 2.4GHz. Given these values, a matrix with dimensions $m + n$ exceeding 524288 has to revert to main memory to store the required vectors. Matrices with dimensions lower than 4096 fit into L1 cache entirely. The third architecture is the

Name	Rows	Columns	Nonzeroes		Symmetry, origin
fidap037	3565	3565	67591	S	struct. symm., FEM
memplus	17758	17758	126150	S	struct. symm., chip design
rhpentium	25187	25187	258265	U	chip design
lhr34	35152	35152	764014	S	chemical process
nug30	52260	379350	1567800	S	quadratic assignment
s3dkt3m2	90449	90449	1921955	S	symm., FEM
tbdlinux	112757	21067	2157675	U	term-by-document
stanford	281903	281903	2312497	U	link matrix
stanford-berkeley	683446	683446	7583376	U	link matrix
wikipedia-2005	1634989	1634989	19753078	U	link matrix
cage14	1505785	1505785	27130349	S	struct. symm., DNA
GL7d18	1955309	1548650	35590540	U	algebraic K-theory
wikipedia-2006	2983494	2983494	37269096	U	link matrix

Table 1: The matrices used in our experiments. The matrices are grouped into two sets by relative size, where the first set fits into the L2 cache, and the second does not. An S (U) indicates that the matrix is considered structured (unstructured).

AMD Phenom II 945e; it also has four cores, each running at 3.0GHz. A single core is connected to a 64kB L1 data cache and a 512kB L2 data cache. All cores share a single 6MB L3 cache; thus, the combined size $m+n$ for which the L1 cache is exceeded is 8192, that for the L2 cache is 65536, and for the L3 cache it is 786432.

For the matrix reordering by 2D methods, the recently released Mondriaan 3.01 sparse matrix partitioning software[1] [16] was used. The original 1D method also employed Mondriaan, but used an earlier test version of Mondriaan 3.0; the current one is potentially faster, depending on the precise options given, and generates partitionings of slightly better quality. Mondriaan now natively supports SBD and DSBD permutation of matrices. Three datasets have been constructed using the Mondriaan partitioner beforehand; the matrices in Table 1 were partitioned using a 1D (row-net) scheme, the 2D Mondriaan scheme, and the 2D fine-grain scheme. These are the same matrices used in our previous work [20], with the addition of a 2006 version of the wikipedia link matrix and the GL7d18 matrix. Most matrices from the dataset used are available through the University of Florida sparse matrix collection [4]. In all cases, the partitioner load-imbalance parameter was taken to be $\epsilon = 0.1$, and the default options were used (except those that specify the hypergraph model and permutation type). The smaller matrices have been partitioned for $p = 2, \ldots, 7, 10, 50, 100$, whereas the larger matrices were partitioned for $p = 2, \ldots, 10$. The construction times required for the smaller matrices are of the order of a couple of minutes; e.g., the most time-consuming partitioning, that of the matrix s3dkt3m2 in 2D by the fine-grain model with $p = 100$, takes 8 minutes. This time typically decreases by more than a factor of two when partitioning in 1D mode, for instance taking 3 minutes for s3dkt3m2 with $p = 100$. The Mondriaan scheme results in partitioning times usually between these two, although in the particular case of s3dkt3m2, it is actually faster with 1 minute.

The partitioning time for the larger matrices using the fine-grain scheme measures

[1] Available at: http://www.math.uu.nl/people/bisseling/Mondriaan/

itself in hours: stanford with $p = 10$ takes about 2 hours, while wikipedia-2005 takes about 7 hours and wikipedia-2006 23 hours. In 1D, running times decrease by more than an order of magnitude, e.g., to 4 minutes for stanford, half an hour for wikipedia-2005, and one hour for wikipedia-2006 with $p = 10$. The Mondriaan scheme, again with $p = 10$, runs in 5 minutes for stanford, 7 hours for wikipedia-2005, and 21 hours for wikipedia-2006.

The SpMV multiplication software[2] was compiled by the IBM XL compiler on Huygens, using auto-tuning for the Power6+ processor with full optimisation enabled[3]. On the Intel Q6600 and AMD 945e platforms, the GNU compiler collection is used, using similar performance flags. The SpMV multiplication software has been written such that it reads text files containing information on the SBD reordered matrix (whether 1D or 2D), as well as information on the corresponding separator tree; thus any partitioner capable of delivering this output can be used. In a single experiment, several multiplications are performed: a minimum of $N = 900$ for the smaller matrices, and a minimum of $N = 100$ for the larger matrices, to obtain an accurate average running time. To ensure the results are valid, $\sqrt{N}$ SpMV multiplications were executed and timed as a whole so as not to disrupt the runs too often with the timers. This was repeated $\sqrt{N}$ times to obtain a better estimate of the mean and a running estimate of the variance. This variance was always a few orders of magnitude smaller than the mean.

We compare all combinations of data structures (plain CRS, plain ICRS, Block-based) and partitioning schemes (Row-net, Mondriaan, Fine-grain). In fact, the block-based data structure comprises several methods, using different block orderings (CRS, Adapted CRS, Separators-last, Zig-zag CCS, and variants) and data structures (simple block-based with ICRS/ICCS, and Bi-directional ICRS); see Figure 5 and Section 2.2. Note that plain CRS and plain ICRS do not make use of sparse blocking based on the separator blocks, and that the simple block-based data structures use ICRS on the diagonal blocks and the vertical separator blocks, and ICCS on the horizontal separator blocks. Any other combination of data structures (non-incremental, or ICRS and ICCS switched) results in uncompetitive strategies. In total, 36 different methods per reordered matrix are tested, resulting in over 4000 individual experiments, per architecture.

Of the partitioning schemes, the row-net scheme corresponds to a 1D partitioning, and the Mondriaan and fine-grain schemes to 2D partitioning. The original 1D method from our earlier paper [20], is in fact the plain CRS or plain ICRS data structure combined with the row-net partitioning scheme. For the full 2D method, various block orders have been tested with the block-based data structure. The CRS and ICRS data structures combined with 2D partitioning have been included since they do not incur any overhead with increasing p at all, and thus can potentially overtake the block-ordered data structures, especially when there are few nonzeroes in the vertical separator blocks. For the Intel architecture, which was also used in our original work [20], results using OSKI [17] instead of plain (I)CRS with 1D partitioning are again included to provide an easy comparison with the earlier results; the timings obtained by applying OSKI to the original matrices are included as well. In all cases, OSKI was forced to perform full aggressive tuning, but no hints as to any matrix structure were provided.

[2] Available at: http://www.math.uu.nl/people/yzelman/software/

[3] *Compiler flags: -O3 -q64 -qarch=auto -qtune=auto -DNDEBUG*

For each dataset, the best timing and its corresponding number of parts p and data structure is reported. Table 2 shows the results for the IBM Power6+ architecture, Table 3 for the Intel Q6600 which includes the OSKI results, and Table 4 for the AMD 945e. As in our earlier work [20], most structured matrices are hardly, or even negatively, affected by the reordering scheme, both with 1D and 2D partitioning. Among the two exceptions is the nug30 matrix, in which the 1D scheme gains 9 percent and the Mondriaan scheme 7 percent on the Huygens supercomputer; the fine-grain scheme and the 1D scheme with blocking do not perform well there. For the AMD platform, these gains increase to 14 percent and the other partitioning schemes are more effective. On the Intel Q6600, gains are almost 17 percent compared to the OSKI baseline, for all three partitioning schemes. The second exception is the s3dkt3m2 matrix, on which gains of 6 percent appear when using the 1D blocking scheme or the Mondriaan scheme on the IBM Power6+; these gains increase to 7 percent for the AMD processor. The optimisations by OSKI are superior to those of our reordering method, however, and only slowdowns are reported for the Intel processor. Compared to a plain (I)CRS baseline, the gain would be 17 percent for the Intel processor, versus the 22 percent for OSKI on this structured matrix.

Reordering on the unstructured matrices is much more effective. Gains are larger than 10 percent in all but two cases: the stanford-berkeley matrix on any of the three architectures, and the tbdlinux matrix, but only on the Intel. Overall, the Mondriaan scheme works best. Blocking with 1D, OSKI with 1D, or the original 1D method were preferred only a few times; and the fine-grain scheme was preferred on only one of the test matrices. On the Intel and AMD platforms, the 2D methods (both Mondriaan and fine-grain) perform best with a block-based data structure, i.e., Block or BICRS, instead of with plain (I)CRS. On the IBM Power6+, this situation is reversed. As an example, for the tbdlinux matrix on the Intel Q6600, we see that a block-based structure with $p = 100$ is preferred above non-blocked (I)CRS; thus definitely the gain in cache efficiency thanks to 2D blocking can overtake the overhead from block-based data structures, even for large p.

Looking at the various block orderings, we observe a difference in tendencies for the Power6+ on one hand, and the Q6600 and 945e on the other hand. From the tables, we can deduce that the Zig-zag ordering is preferred on Huygens, and that the Adapted and SepLast orderings are the least preferred. On the other two architectures, the Adapted orderings win most often, and SepLast wins the least.

One could suspect the results may differ when taking into account all different p (not only the best choices) for all partitioning schemes to find the best performing block orders, instead of using only the tables presented in this section. Doing this, on Huygens, we see the Adapted orderings appear most often as winner, followed by the Zig-zag orderings. Looking at all experiments for the other two architectures reveals an interesting result: when using BICRS as a block-based data structure, there is a clear preference for the Block CRS ordering. Correcting for this bias by looking only at the simple block-based data structure, revealed no apparent preference for either Adapted or Zig-zag orderings; hence the conclusions remain somewhat in agreement with the theoretical results from Section 2.4, as indeed the Zig-zag and Adapted orderings are preferred over Block CRS and the SepLast orderings.

Considering instead which block data structures are used, we see that on the Power6+ the BICRS data structure is completely uncompetitive with the simple block data struc-

Matrix	Natural	1D [20]	1D & Blocking
fidap037	0.116 ICRS	*0.113* ICRS (100)	0.117 Block CRS (2)
memplus	0.308 CRS	0.305 ICRS (4)	0.326 Block CRS (2)
rhpentium	0.913 ICRS	0.645 ICRS (50)	0.877 Block CRS (100)
lhr34	1.366 ICRS	1.362 ICRS (5)	1.373 Block CRS (2)
nug30	5.350 CRS	*4.846* ICRS (3)	5.355 Block CRS (6)
s3dkt3m2	7.806 CRS	7.847 CRS (3)	7.285 Block CRS (2)
tbdlinux	6.428 CRS	6.085 ICRS (5)	*5.027* Block CRS (4)
stanford	18.99 CRS	9.92 CRS (10)	9.52 Block CRS (9)
stanford_berkeley	20.93 ICRS	20.10 ICRS (4)	19.26 Block CRS (9)
cage14	*69.36* CRS	75.47 ICRS (2)	76.48 Block CRS (2)
wikipedia-2005	248.6 CRS	154.9 CRS (10)	142.3 Block CRS (9)
GL7d18	599.6 CRS	401.1 CRS (9)	*386.7* Block CRS (7)
wikipedia-2006	688.4 CRS	378.3 CRS (9)	302.4 Block CRS (9)

Matrix	2D Mondriaan	2D Fine-grain
fidap037	*0.113* ICRS (50)	*0.113* ICRS (100)
memplus	*0.300* ICRS (2)	0.307 ICRS (3)
rhpentium	*0.627* ICRS (50)	0.635 ICRS (50)
lhr34	*1.336* Block ZZ-CRS (3)	1.361 Block ZZ-CCS (5)
nug30	4.943 CRS (3)	5.239 Block SepLast (3)
s3dkt3m2	*7.269* Block ZZ-CCS (3)	7.503 Block ACRS (3)
tbdlinux	5.433 Block ZZ-CRS (7)	5.433 Block ZZ-CCS (100)
stanford	*9.35* CRS (8)	9.73 CRS (10)
stanford_berkeley	*19.18* Block ACRS (4)	19.41 Block SepLast (9)
cage14	74.37 ICRS (2)	75.13 ICRS (2)
wikipedia-2005	*115.6* CRS (8)	124.2 Block ACRS (8)
GL7d18	392.4 Block ZZ-CRS (10)	435.2 Block SepLast (3)
wikipedia-2006	*256.4* CRS (9)	267.5 Block ZZ-CCS (8)

Table 2: Time of an SpMV multiplication on the naturally ordered matrices, as well as on reordered matrices, both in 1D and 2D, on an IBM Power6+ architecture. Time is measured in milliseconds, and only the best average running time among the different number of parts p and the different data structures used, is shown. The specific data structure used is reported directly after the best running time, and the number of parts p used is shown between parentheses. In the 1D & Blocking category, the Block CRS is the only block order experimented with on this architecture. For each sparse matrix, the best timing among the various schemes is printed *italic*. For the smaller datasets (above the horizontal separator line), the number of parts used were $p = 2, \ldots, 7, 10, 50, 100$; for the larger ones (below the separator line), these were $p = 2, \ldots, 10$.

Matrix	Natural		1D & OSKI [20, 17]			1D & Blocking		
fidap037	0.142	CRS	0.134	ICRS	(100)	0.145	Block SepLast	(3)
memplus	0.352	CRS	*0.278*	CRS	(50)	0.407	Block ACRS	(3)
rhpentium	0.897	OSKI	0.748	OSKI	(100)	1.034	Block ZZ-CCS	(5)
lhr34	2.625	OSKI	2.411	OSKI	(10)	3.127	Block ACRS	(3)
nug30	10.61	OSKI	9.28	OSKI	(10)	*8.348*	BICRS ACRS	(6)
s3dkt3m2	*10.06*	OSKI	10.28	OSKI	(2)	13.68	Block ACCS	(3)
tbdlinux	8.083	OSKI	*7.647*	OSKI	(100)	9.159	Block SepLast	(5)
stanford	27.23	OSKI	*11.51*	CRS	(8)	12.32	Block ACCS	(8)
stanford_berkeley	31.72	OSKI	*30.64*	OSKI	(4)	34.61	Block SepLast	(5)
cage14	*111.6*	OSKI	130.4	OSKI	(2)	140.5	BICRS SepLast	(10)
wikipedia-2005	347.4	OSKI	223.1	OSKI	(7)	212.5	BICRS ACRS	(9)
GL7d18	780.3	CRS	552.5	OSKI	(8)	612.8	BICRS SepLast	(10)
wikipedia-2006	745.4	OSKI	495.1	OSKI	(9)	541.1	Block CRS	(9)

Matrix	2D Mondriaan			2D Fine-grain		
fidap037	0.135	ICRS	(50)	*0.133*	ICRS	(100)
memplus	0.310	CRS	(2)	0.311	CRS	(3)
rhpentium	*0.855*	ICRS	(100)	0.870	ICRS	(100)
lhr34	*2.357*	Block SepLast	(2)	2.938	ICRS	(100)
nug30	8.360	Block ZZ-CCS	(10)	8.367	Block SepLast	(2)
s3dkt3m2	10.81	Block CRS	(4)	12.99	ICRS	(3)
tbdlinux	8.059	Block CRS	(2)	8.406	BICRS ACRS	(100)
stanford	12.59	BICRS ACCS	(7)	12.52	BICRS CRS	(7)
stanford_berkeley	32.81	BICRS ACRS	(10)	34.84	BICRS CRS	(8)
cage14	130.4	BICRS CRS	(10)	147.4	CRS	(7)
wikipedia-2005	*136.7*	BICRS CRS	(8)	167.3	BICRS ACRS	(10)
GL7d18	*549.5*	CRS	(10)	567.6	BICRS ZZ-CRS	(10)
wikipedia-2006	*311.8*	BICRS CRS	(9)	395.6	BICRS CRS	(10)

Table 3: As Table 2, but with timings performed on an Intel Core 2 Q6600 architecture. The table includes timings using OSKI, combined with 1D reordering, as in our earlier paper [20]. In this table, the 1D & Blocking scheme includes block orderings and data structures other than Block CRS.

Matrix	Natural	1D [20]	1D & Blocking
fidap037	0.132 CRS	0.128 CRS (2)	0.167 Block CRS (2)
memplus	0.315 ICRS	*0.301* CRS (2)	0.348 Block SepLast (2)
rhpentium	0.980 ICRS	0.754 ICRS (50)	1.024 BICRS ZZ-CCS (5)
lhr34	1.712 ICRS	1.711 ICRS (10)	1.756 Block ACCS (3)
nug30	6.369 CRS	5.797 CRS (2)	6.478 Block ZZ-CCS (6)
s3dkt3m2	8.636 ICRS	8.619 ICRS (2)	8.660 Block ZZ-CCS (2)
tbdlinux	7.165 ICRS	6.531 ICRS (4)	*6.151* Block ACRS (2)
stanford	21.95 CRS	10.57 ICRS (9)	10.29 Block ZZ-CCS (9)
stanford_berkeley	24.64 ICRS	24.08 ICRS (3)	23.79 Block CRS (7)
cage14	111.3 ICRS	116.1 ICRS (8)	113.3 Block ACCS (2)
wikipedia-2005	264.0 CRS	179.0 ICRS (7)	160.8 Block ACRS (9)
GL7d18	729.6 CRS	451.7 ICRS (8)	*412.4* BICRS ZZ-CRS (9)
wikipedia-2006	819.1 CRS	399.4 ICRS (9)	382.6 BICRS ACRS (10)

Matrix	2D Mondriaan	2D Fine-grain
fidap037	*0.127* CRS (4)	0.128 CRS (7)
memplus	0.306 CRS (2)	0.308 CRS (2)
rhpentium	*0.736* ICRS (50)	0.754 ICRS (50)
lhr34	*1.566* BICRS ZZ-CCS (2)	1.705 ICRS (10)
nug30	*5.422* CRS (3)	5.931 CRS (3)
s3dkt3m2	*7.996* BICRS CRS (3)	8.640 Block ZZ-CCS (2)
tbdlinux	6.254 Block ACRS (2)	6.243 BICRS ACRS (2)
stanford	*9.86* BICRS SepLast (10)	10.44 BICRS ACRS (9)
stanford_berkeley	*23.66* Block ACRS (2)	24.02 ICRS (2)
cage14	*109.5* ICRS (10)	110.7 CRS (2)
wikipedia-2005	*119.4* BICRS ACCS (7)	138.3 Block ZZ-CCS (9)
GL7d18	424.6 BICRS ACRS (10)	422.7 BICRS ACRS (5)
wikipedia-2006	*261.3* BICRS ZZ-CCS (9)	346.7 BICRS ACRS (9)

Table 4: As Table 3, but with timings performed on an AMD Phenom II 945e architecture.

ture. This is in direct contrast to the Q6600 and 945e processors, where with effective 1D and Mondriaan partitioning on the larger matrices, the BICRS data structure consistently outperforms the simple block data structure. When the number of separator blocks increases, however, such as happens with 2D fine-grain partitioning, the simple block data structure becomes more competitive.

Tests on the larger matrices display a more pronounced gain for the 2D methods. The matrices where the 1D partitioning was not effective in our previous work [20], namely stanford_berkeley and cage14, perform slightly better in 2D, and the gains (or losses) remain limited. When partitioning did work in the original experiments, however, two-dimensional partitioning works even better, for nearly all of the test instances. It is also interesting to see that 1D partitioning combined with blocking outperforms the original 1D reordering method for larger p (for the Intel architecture as well, although this is obscured from Table 3 by the additional gains obtained with OSKI). For the largest matrices from our whole test set, typically the Mondriaan scheme performs best. After that comes the fine-grain scheme, then followed by the blocked 1D method, and finally the original 1D method; this was also the case for the smaller matrices, with relatively few exceptions.

The most impressive gain is found on the wikipedia-2006 matrix; a 68 percent gain, more than a factor of three speedup, is obtained with the Mondriaan scheme and the ZZ-CCS block order on the AMD 945e. The runner-up block orderings, ACRS and Block CRS, remain competitive with 67 percent. The worst performing block orders gain only 40 percent, however, and these large differences in performance become more apparent as matrices increase in size, on all architectures.

Noteworthy is that ICRS does not always outperform CRS; this only happened on either smaller matrices or when many empty rows (or columns) were encountered, such as in separator blocks. On larger matrices, CRS consistently outperforms ICRS, even after repeated partitioning. Within the simple block data structure, the incremental variants (ICRS/ICCS) do remain the only option, as the alternative non-incremental structure exhibits extremely poor scalability in p.

4. Conclusions and future work

A one-dimensional cache-oblivious sparse matrix–vector multiplication scheme [20] has been extended to fully utilise 2D partitioning, using the fine-grain model for a hypergraph partitioning of sparse matrices. Alternatively, the Mondriaan scheme for partitioning can be used, which combines two different 1D partitioning schemes to obtain a 2D partitioning in recursion. Generalising the permutation scheme from 1D to 2D shows the usual data structures for sparse matrices can be suboptimal in terms of cache efficiency when the separator blocks are dense enough. To alleviate this, we propose to process the nonzeroes block-by-block, each block having its own data structure storing the nonzeroes in CRS or CCS order, using either BICRS or a sequence of data structures to store a single matrix. This can be seen as a form of sparse blocking. The incremental implementation of CRS and CCS should always be preferred when used within a simple block data structure; many rows and columns in matrix sub-blocks are empty, thus using standard CRS or CCS will cause poor scalability as one word per row or column is used, regardless if it is empty. This type of blocking has also been introduced into our previous 1D scheme.

Experiments were performed on an IBM Power6+, an Intel Q6600, and an AMD 945e machine. The results for the smaller matrices, where the input and output vector fit into L2 cache, showed definitive improvement over the original method, especially on the unstructured matrices. This tendency is also observed in the case of the larger matrices; the improved 1D method, utilising sparse blocking on a 1D reordered matrix, improved on the original scheme in a more pronounced manner than on the smaller matrices. However, in both cases, all methods are dominated by the Mondriaan scheme.

The 2D method presented here still uses only row and column permutations, permuting an input matrix A to PAQ, with which the multiplications are carried out. This is unchanged from the original method. It is also still possible to combine this method with auto-tuning (cache-aware) software such as OSKI [17] to increase the SpMV multiplication speed further; this software can be applied to optimise the separate block data structures, but this has not been investigated here. This method can also still be implemented using other partitioners than Mondriaan, such as, e.g., PaToH [2], although modifications may be required to extract an SBD permutation and separator tree, or to perform partitioning according to the Mondriaan scheme.

A major difference with our earlier work is the use of sparse blocking. When used, as the number of blocks increases, the overhead of having several data structures for storing the permuted sparse matrix increases. The instruction overhead is linear in p and although the matrix storage overhead is bounded by the number of nonzeroes, taking $p \to \infty$ does eventually harm efficiency when using sparse blocking, thus posing a theoretical limit to the cache-oblivious nature of the reordering.

Various items for future research can be readily identified:

- The simple block-based data structure presented here is very basic, and a more advanced data structure might be introduced to lower or eliminate overhead with increasing p, and thus gain even more efficiency; existing methods [1] could be extended and integrated to this end.
- The results indicate that the efficiency of specific block orders is matrix-dependent. As the analysis already revealed differences between locality (and thus sizes) of blocks in the row and column direction, this can perhaps lead to an efficient heuristic when to choose which block order. This block order does not have to be constant in the recursion, and may require information on the density of separator blocks.
- The difference in performance between standard CRS and ICRS seems dependent on, amongst others, the matrix structure and size; it warrants future research to find the precise dependencies. Again this can lead to an efficient heuristic for adaptively choosing a data structure, also within a simple or advanced block-based data structure.
- The speed of building up the datasets, although already greatly improved when compared to the results in the original paper [20], can still be increased. In particular, the preparation times with the Mondriaan scheme can still be large.

Acknowledgements

We like to thank the SARA supercomputer centre in Amsterdam and the Netherlands National Computer Facilities foundation (NCF) for computation time and help with

access to the Huygens computer. We also extend our gratitude to the anonymous referees whose comments have greatly helped in improving this paper.

References

[1] A. Buluç, 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 '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures, New York, NY, USA, 2009, ACM, pp. 233–244.

[2] Ü. V. Çatalyürek and C. Aykanat, *Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication*, IEEE Trans. Parallel Distrib. Systems, 10 (1999), pp. 673–693.

[3] ———, *A fine-grain hypergraph model for 2D decomposition of sparse matrices*, in Proceedings 8th International Workshop on Solving Irregularly Structured Problems in Parallel, IEEE Press, Los Alamitos, CA, 2001, p. 118.

[4] T. A. Davis and Y. Hu, *The University of Florida Sparse Matrix Collection*, ACM Transactions on Mathematical Software, (2011). Accepted for publication.

[5] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, *Cache-oblivious algorithms*, in Proceedings 40th Annual Symposium on Foundations of Computer Science, IEEE Press, Los Alamitos, CA, 1999, p. 285.

[6] K. Goto and R. van de Geijn, *On reducing TLB misses in matrix multiplication*, Tech. Rep. TR-2002-55, University of Texas at Austin, Department of Computer Sciences, 2002. FLAME Working Note #9.

[7] G. Haase and M. Liebmann, *A Hilbert-order multiplication scheme for unstructured sparse matrices*, International Journal of Parallel, Emergent and Distributed Systems, 22 (2007), pp. 213–220.

[8] E.-J. Im and K. A. Yelick, *Optimizing sparse matrix–vector multiplication for register reuse in SPARSITY*, in Proceedings International Conference on Computational Science, Part I, vol. 2073 of Lecture Notes in Computer Science, 2001, pp. 127–136.

[9] J. Koster, *Parallel templates for numerical linear algebra, a high-performance computation library*, Master's thesis, Utrecht University, Department of Mathematics, July 2002.

[10] K. P. Lorton and D. S. Wise, *Analyzing block locality in Morton-order and Morton-hybrid matrices*, SIGARCH Comput. Archit. News, 35 (2007), pp. 6–12.

[11] G. Morton, *A computer oriented geodetic data base and a new technique in file sequencing*, tech. rep., IBM, Ottawa, Canada, March 1966.

[12] R. Nishtala, R. W. Vuduc, J. W. Demmel, and K. A. Yelick, *When cache blocking of sparse matrix vector multiply works and why*, Appl. Algebra Engrg. Comm. Comput., 18 (2007), pp. 297–311.

[13] S. Toledo, *Improving the memory-system performance of sparse-matrix vector multiplication*, IBM J. Res. Dev., 41 (1997), pp. 711–725.

[14] V. Valsalam and A. Skjellum, *A framework for high-performance matrix multiplication based on hierarchical abstractions, algorithms and optimized low-level kernels*, Concurrency and Computation: Practice and Experience, 14 (2002), pp. 805–839.

[15] R. van der Pas, *Memory hierarchy in cache-based systems*, Tech. Rep. 817-0742-10, Sun Microsystems, Inc., Santa Clara, CA, Nov. 2002.

[16] B. Vastenhouw and R. H. Bisseling, *A two-dimensional data distribution method for parallel sparse matrix-vector multiplication*, SIAM Rev., 47 (2005), pp. 67–95.

[17] R. Vuduc, J. W. Demmel, and K. A. Yelick, *OSKI: A library of automatically tuned sparse matrix kernels*, J. Phys. Conf. Series, 16 (2005), pp. 521–530.

[18] R. C. Whaley, A. Petitet, and J. J. Dongarra, *Automated empirical optimizations of software and the ATLAS project*, Parallel Comput., 27 (2001), pp. 3–35.

[19] A. Yzelman and R. H. Bisseling, *A cache-oblivious sparse matrix–vector multiplication scheme based on the Hilbert curve*, in Progress in Industrial Mathematics at ECMI 2010, Springer, 2011. Accepted for publication.

[20] A. N. Yzelman and R. H. Bisseling, *Cache-oblivious sparse matrix–vector multiplication by using sparse matrix partitioning methods*, SIAM Journal on Scientific Computing, 31 (2009), pp. 3128–3154.