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 data structure, to a datastructure based on blocks with compressed row or column storage within each block. Experiments performed on a set of sparse matrices 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 almost a factor of 3 in SpMV speed, compared to the natural matrix ordering.

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

1. Introduction

Our earlier work in [14] 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 also in sequential SpMV multiplication, 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)

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 [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 row of A corresponding to the net. A *partitioning* of a matrix into p parts corresponds to 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 [14], 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 $|\mathcal{V}_j| \leq (1+\epsilon)\frac{\text{nz}(A)}{p}$ 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. Row permutations are made according to the classifications of the nets corresponding to the rows of A. All rows corresponding to nets in $\mathcal{N}_+$ are permuted towards the bottom of the matrix, whereas those corresponding to $\mathcal{N}_-$ are permuted towards the top. Other rows are left in the middle. This defines row-wise *boundaries* in the matrix; 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 a *separator blocks*. Column permutation is done according to the vertex subset split: columns corresponding to vertices 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. The resulting form of the permuted sparse matrix is called *Separated Block Diagonal* (SBD).

Important is the demand that subsequent iterations do not violate any boundaries set by earlier iterations. A row which 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 $\mathcal{V}_i$ corresponds to 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) datastructure. In fact, when $\max|\mathcal{V}_i|$ elements from the input and output vector *exactly* fit into cache, when the rows corresponding to the cut nets are dense enough, and when Zig-zag CRS storage (ZZ-CRS, as introduced in [14] and depicted by the dashed curve in Figure 2, right) is used, an upper bound on the cache misses incurred is $\sum_i(\lambda_i - 1)$ [14, Section 5.2], i.e. 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 $(\mathcal{V}_i, \mathcal{N}_c)$, with $\mathcal{V}_i$ 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 [14]. 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. [12], 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 execute more efficiently than when executed on the single larger sparse matrix [4, 10, 13]. By using a Morton ordering (i.e., a recursive Z-like ordering) [8], 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 datastructures such as the standard CRS. This gives rise to so called hybrid-Morton datastructures which, like the method presented here, enable the use of specialised BLAS libraries on the low-level blocks; see Lorton and Wise [7]. Similar techniques for sparse matrices include [9] for sparse blocking into very small blocks, and [5] for using the Hilbert space-filling curve on the nonzeroes of sparse matrices. 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]. Using this model, a sparse matrix A is again modelled by a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$, but now such that each nonzero a_{ij} corresponds 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 the corresponding 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 cut row nets $\mathcal{N}^c_{\text{row}} \subset \mathcal{N}$ as row nets with connectivity larger than one, as well as $\mathcal{N}^c_{\text{col}}$, similar to the 1D case. Row and column permutations for $p = 2$ can then be used to bring the arbitrary sparse input matrix in the form depicted in Figure 2(left), the *doubly separated block diagonal* (DSBD) form. Note that there are now five different separator blocks, namely $\mathcal{N}^{\text{row}}_c \cap \mathcal{N}^{\text{col}}_c$, $\mathcal{N}^{\text{row}}_c \cap \mathcal{N}^{\text{col}}_\pm$, and $\mathcal{N}^{\text{row}}_\pm \cap \mathcal{N}^{\text{col}}_c$. 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 datastructure 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 datastructure used must support this. These demands are treated separately in Section 2.1 and Section 2.3, 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, corresponding 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 in the vertical direction correspond with write accesses on the output vector, whereas horizontal accesses correspond with read access on the input vector. Write accesses are potentially more expensive; the block ordering avoiding the most unnecessary irregular

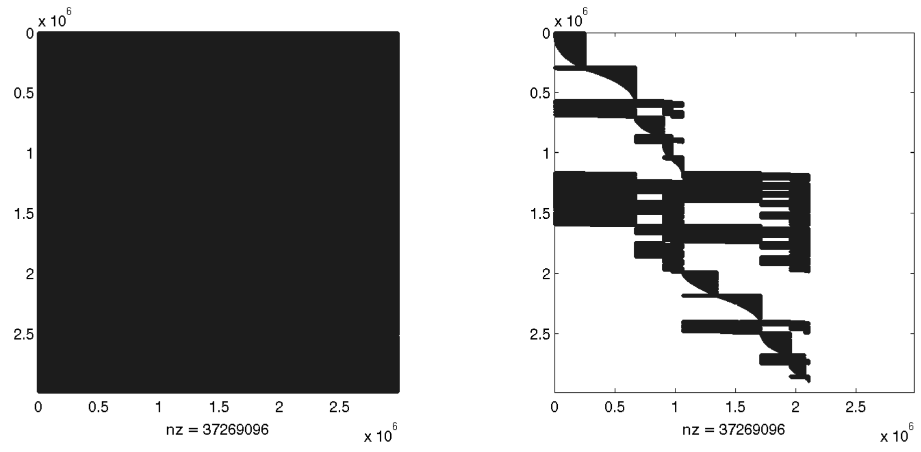


Figure 1: Spy plots of the wikipedia-2006 matrix, partitioned and reordered to SBD form using the Mondriaan software [11]. The matrix has 2983494 rows and columns, and 37269096 nonzeroes. The left spy plot shows the highly unstructured original link matrix. This matrix appears to dense because of the size of the nonzero markers; 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.

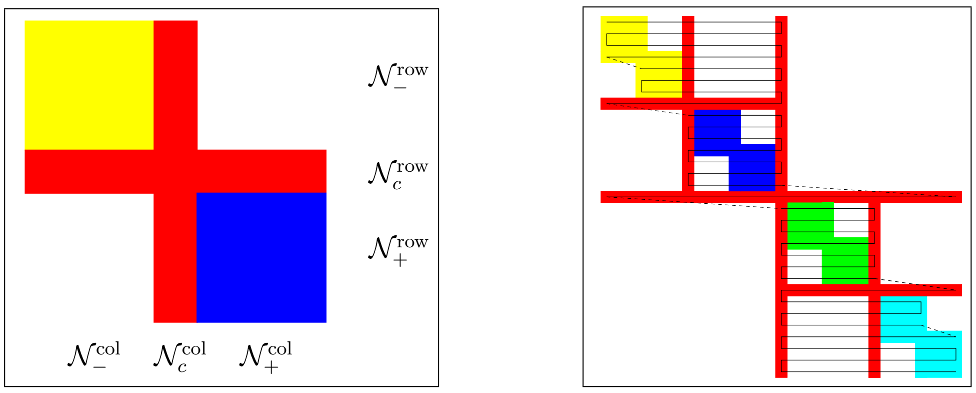

$\mathcal{N}_-^{\text{row}}$
$\mathcal{N}_c^{\text{row}}$
$\mathcal{N}_+^{\text{row}}$
$\mathcal{N}_-^{\text{col}}$
$\mathcal{N}_c^{\text{col}}$
$\mathcal{N}_+^{\text{col}}$

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.

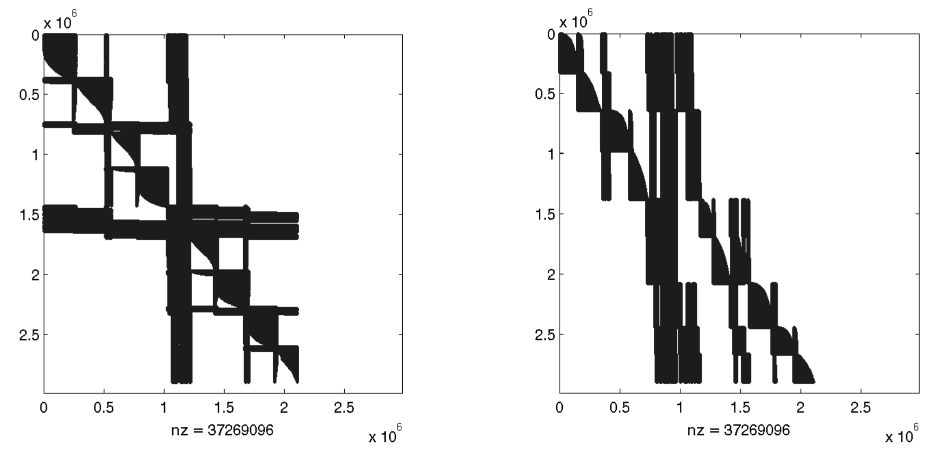
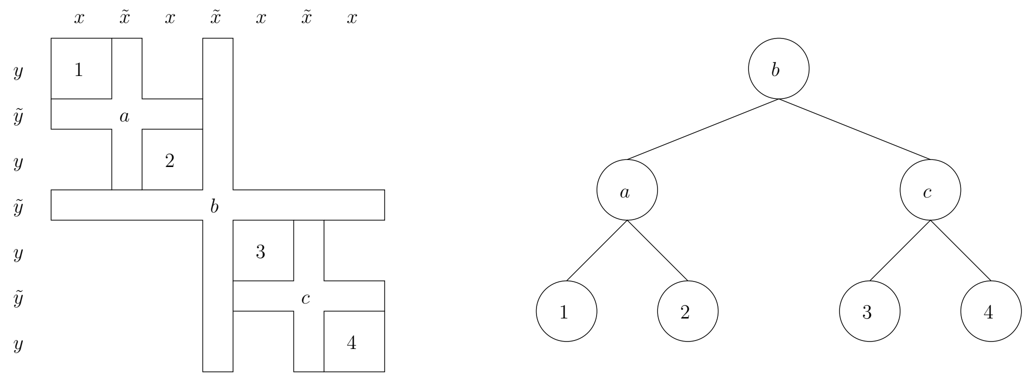


Figure 3: Spy plots of the wikipedia-2006 matrix, like Figure 1, but now using 2D partitioning. The left picture shows the result using the fine-grain scheme with $p = 8$. On the right, the Mondriaan scheme with $p = 9$ was used. The load imbalance parameter is set to 0.1 in both cases.



x $\tilde{x}$ x $\tilde{x}$ x $\tilde{x}$ x
y $\tilde{y}$ y $\tilde{y}$ y $\tilde{y}$ y
1
a
2
b
3
c
4
b
a
c
1
2
3
4

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

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 the vertical separator blocks, CRS is the perfect ordering for the nonzeroes; the width in the column direction is typically small by grace of good partitioning, and so the corresponding 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 would fit into cache, but the range corresponding to the input vector is large and access is in general completely irregular. Demanding instead that the horizontal separators 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 the output vector accesses are limited to a small range typically fitting into cache. Note that this scheme can also be applied on the original 1D method, thus augmenting an ordering obtained by 1D partitioning with a 2D sparse blocking scheme.

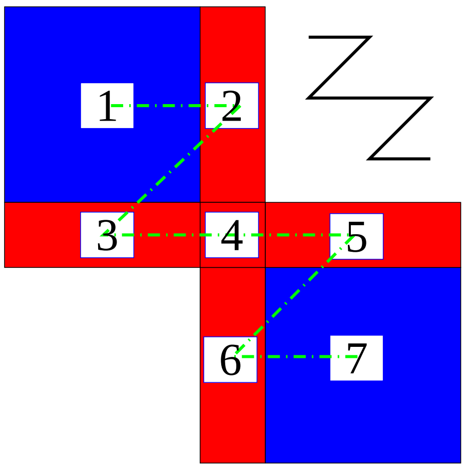
2.2. The Mondriaan partitioning scheme

Modelling a sparse matrix as a hypergraph using the fine-grain model has as a main drawback the increased size of the hypergraph, compared to the simpler row-net model. This generally leads to an increased partitioning time, hopefully justified by the increased quality of the now 2D partitioning. Another drawback is the number of separator blocks created when using the fine-grain scheme; each block, as mentioned earlier, incurs some overhead and therefore reducing the number of blocks while retaining a 2D partitioning increases efficiency. A compromise exists however, which is implemented in the Mondriaan sparse matrix partitioner software [11]. It combines two 1D methods as follows.

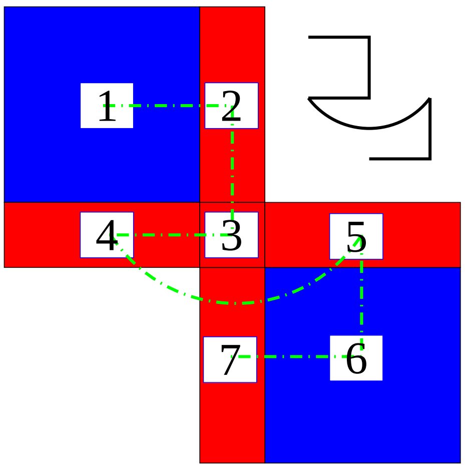
Apart from the row-net model, a column-net model can also be defined, identical to the row-net model but with the roles of rows and columns reversed: each row corresponds to a vertex in the hypergraph, and each column to a net. A cheaper way of obtaining a 2D partitioning for $p > 2$, then is to use 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. We will refer to this partitioning method as the *Mondriaan scheme*. 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.

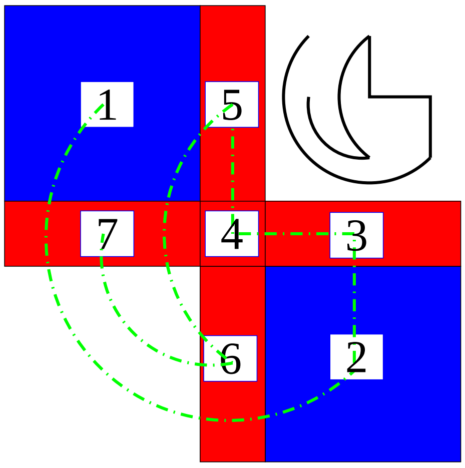
It is worthwhile to exploit the form of the 2D DSBD ordering in the special case of bipartitioning by the row-net or column-net model. The 1D row-net model yields the picture in Figure 2(left) with the centre block and the two vertical rectangular blocks removed from the separator cross (this corresponds to the original SBD form in [14]). The column-net model is similar, having instead the centre block and the two horizontal



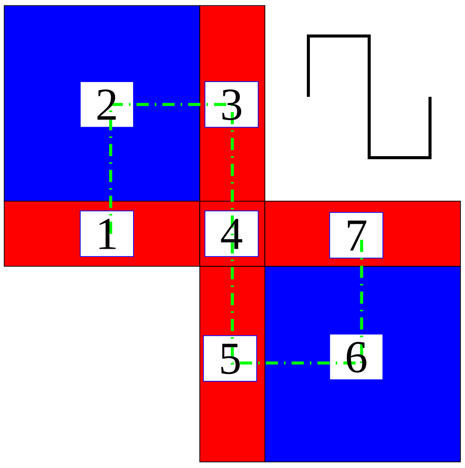
(a) CRS block ordering



(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 show the curve trajectory in an unobfuscated view.

rectangular blocks removed. As such, using the Mondriaan scheme instead of the fine-grain scheme in full recursive bipartitioning, nearly halves the number of separator blocks, from $p + 5(p - 1) = 6p - 5$ to $p + 2(p - 1) = 3p - 2$ blocks.

2.3. A block-based datastructure

Two datastructures will be introduced here: ICRS, and a block-based datastructure. The first can be viewed as an alternative implementation of the more standard CRS datastructure. This datastructure can, just as standard CRS, be integrated into a block-based datastructure.

With the Incremental CRS (ICRS) storage scheme [6], 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. This 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 which range of nonzeroes. The hope is that this alternative implementation uses less instructions for the SpMV kernel and reduces the datastructure 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.

We propose to store each matrix block in a separate (I)CRS or (I)CCS datastructure. Upon performing a SpMV multiplication, the multiplication routines of the separate datastructures are called in the order defined by the block order, on the same input and output vectors; this yields a straightforward block-based datastructure. 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 corresponding of the first nonzero of the next block, and only then the actual SpMV kernel is called. Hence the gain with respect to cache efficiency of using this block-based structure must be larger than the penalty incurred by this additional overhead; if the quality of partitioning by the fine-grain and Mondriaan schemes is similar, then the Mondriaan scheme is expected to perform better because there are fewer blocks. Instead of this straightforward block-based datastructure, more advanced ideas may be exploited here, such as a scheme which also compresses index (or increment) values as presented by Buluç et al. in [1]; although some additional effort would be required to make this work within our scheme.

2.4. Cache performance in recursion

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 corresponds to the largest separator cross (spanning the entire sparse matrix). The two children of the root node then correspond to the remaining two sparse blocks the partitioner has been recurring on. We only count non-obligatory cache misses, because the obligatory ones, related to the first time data are accessed, are the same for all orderings. The number of non-obligatory 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-obligatory 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-obligatory. This happens for example in Figure 5a, when the separator block with index 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-obligatory cache misses.

2.4.1. CRS block ordering

First, the CRS block ordering (Figure 5a) is considered. In this case, $f_0(0) = 2(x+\tilde{x})$, and $f_3(0) = 4x + 3\tilde{x} + 2y + \tilde{y}$. The expressions f_1, f_2 will not appear in this analysis. The recursive upper bounds are:

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

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.

$$\begin{aligned} f_0(0) &= 2\tilde{y}, & f_0(i) &= f_1(i-1) + f_2(i-1) + 2^i(x+y) + (2^i - 1)\tilde{x} + (2^i + 1)\tilde{y}, \\ f_1(0) &= 2(x + \tilde{y}) + \tilde{x}, & f_1(i) &= f_1(i-1) + f_3(i-1) + 2^{i+1}x + 2^i y + (2^{i+1} - 1)\tilde{x} + (2^i + 1)\tilde{y}, \\ f_2(0) &= 2y + 3\tilde{y}, & f_2(i) &= f_2(i-1) + f_3(i-1) + 2^i x + 2^{i+1}y + (2^i - 1)\tilde{x} + (2^{i+1} + 1)\tilde{y}, \\ f_3(0) &= 2(x+y) + \tilde{x} + 3\tilde{y}, & f_3(i) &= 2 \cdot f_3(i-1) + 2^{i+1}(x+y) + (2^{i+1} - 1)\tilde{x} + (2^{i+1} + 1)\tilde{y}. \end{aligned}$$

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:

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

The difference in non-obligatory 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 can conclude 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 in between the ACRS and CRS block orders, with regards to expected performance.

3. Experimental results

Experiments have been performed on 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.

For the matrix reordering by 2D methods, the recently released Mondriaan 3.01 sparse matrix partitioning software[1] [11] 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 we used in our previous work [14], with the addition of a 2006 version of the wikipedia link matrix. 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 $p = 2$–$7, 10, 50, 100$, whereas the larger matrices were partitioned for $p = 2$–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, e.g. taking 3 minutes for s3dkt3m2 with $p = 100$. The Mondriaan

[1] Available at: http://www.math.uu.nl/people/bisseling/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 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]. The 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. Several multiplications are done: 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.

In our experiments, we compare the following nine combinations of datastructures and partitioning schemes:

$$\begin{pmatrix} \text{CRS} \\ \text{ICRS} \\ \text{Block-based} \end{pmatrix} \begin{pmatrix} \text{Row-net} & \text{Mondriaan} & \text{Fine-grain} \end{pmatrix}.$$

Here, the CRS and ICRS datastructures correspond with plain implementations, not making use of any kind of sparse blocking based on the separator blocks. Although here indicated as a single datastructure, 'block-based' actually refers to several datastructures, each with a different block order, including those depicted in Figure 5. These block-based datastructures use ICRS on the diagonal blocks, ICCS on the vertical separator blocks, and ICRS on the horizontal separator blocks. Any other combination of datastructures (non-incremental, or ICRS and ICCS switched) results in uncompetitive strategies. Of the partitioning schemes, note that the row-net scheme corresponds to a 1D partitioning, and the Mondriaan and fine-grain schemes correspond to 2D partitioning. The original 1D method as introduced in [14] corresponds to CRS or ICRS combined with the row-net partitioning scheme. For the full 2D method, various block orders have been tested with the block-based datastructure. The CRS and ICRS datastructures 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 datastructures, especially when there are few nonzeroes in the vertical separator blocks.

For each dataset, the best timing and its corresponding number of parts p is reported in Table 2 for the smaller matrices, and Table 3 for the larger matrices. The best

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

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

corresponding datastructure is also reported. As in [14], 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; the fine-grain scheme and the 1D scheme with blocking do not perform well. The second structured exception is s3dkt3m2, which gains about 6 percent when using the 1D blocking scheme or the Mondriaan scheme. The original scheme and the fine-grain scheme are the less efficient schemes for this matrix.

The unstructured matrices all have gains larger than 10 percent, with again the Mondriaan scheme working best, only being surpassed once by the original 1D method on the structured s3dkt3m2 matrix. The fine-grain scheme is less efficient, and is sometimes being surpassed by the 1D method using blocking. It is notable that the Mondriaan scheme performs better with the block datastructure only four out of the seven times, and that in two of those cases the number of parts is quite large (50 and 100). The same holds for the fine-grain scheme. In general, the Zig-zag block orders seem most efficient, scoring fastest in four out of eight cases. The runner-up is ACRS with two out of eight, as expected from the analysis, but with a smaller margin. Also as expected, the CRS block order never is the most efficient block order in any of the experiments. Most noteworthy is the gain found on the tbdlinux matrix; a 63 percent gain, almost a factor of three speedup, obtained with the Mondriaan scheme and the ACRS block order. The runner-up datastructures ACCS, Separators-last and the Zig-zag block orders attain about 50 percent; significantly less than ACRS.

Tests on the larger matrices, shown in Table 3, display a more pronounced gain for the 2D methods. The matrices where the 1D partitioning was not effective in our previous work [14], namely stanford_berkeley and cage14, perform only slightly better in 2D, and the gains (stanford_berkeley, 8 percent) or losses (cage14, also 8 percent) remain limited. When partitioning did work in the original experiments, however, the 2D partitioning works even better, with the top gain observed with wikipedia-2006: 62 percent (Mondriaan scheme) versus the original gain of 45 percent. For the largest matrices from our whole test set, the fine-grain scheme outperforms the 1D methods, and in turn the Mondriaan-scheme dominates all other schemes; this was also the case for the smaller matrices. The 1D scheme with blocking usually outperforms the original 1D scheme except on cage14.

When using the Mondriaan scheme on the larger matrices, in the 4 out of 5 cases where partitioning was successful, the CRS datastructure outperforms all block-based datastructures. When using the fine-grain scheme, we observe the opposite. This indicates that the number of separator blocks became large enough to profit from the block ordering, while loss of efficiency caused by the overhead in the block datastructures did not yet surface, with the values for p used here. This was also confirmed by some of the positive results on the smaller dataset with large p which employed the Adapted and Zig-zag block block datastructures. No clear preference for a specific block order surfaced. Combined with the results on tbdlinux, this choice is suspected to be highly matrix (and partitioning) dependent.

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. With the block-based datastructures, ICRS did perform better

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-20051105	1634989	1634989	19753078	U	link matrix
cage14	1505785	1505785	27130349	S	struct. symm., DNA
wikipedia-20060925	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).

than CRS when used within the smaller non-separator blocks.

4. Conclusions and future work

The cache-oblivious sparse matrix–vector multiplication scheme in [14] has been extended to fully utilise 2D partitioning, using the fine-grain model for a hypergraph partitioning of sparse matrices. Alternatively, also 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 showed that the usual datastructures 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 datastructure storing the nonzeroes in CRS or CCS order, thus using a sequence of datastructures 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 as such a 'sub-datastructure'. This type of blocking can and has also been introduced into our previous 1D scheme from [14].

Experiments were performed on an IBM Power6+ machine, and 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; for the larger matrices without the need for sparse blocking.

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 [12] to increase the SpMV multiplication speed further; this software can be applied to optimise the separate block

Matrix	Natural	1D [14]	1D & Blocking	2D Mondriaan	2D Fine-grain
fidap037	0.116^2	0.113^2 (100)	0.117 (2)	0.113^2 (50)	0.113^2 (100)
memplus	0.308^1	0.305^2 (4)	0.326 (2)	0.300^2 (2)	0.307^2 (3)
rhpentium	0.913^2	0.645^2 (50)	0.877 (100)	0.515^4 (100)	0.635^2 (50)
lhr34	1.366^2	1.362^2 (5)	1.373 (2)	1.336^3 (3)	1.361^3 (5)
nug30	5.350^1	4.846^2 (3)	5.355 (6)	4.943^1 (3)	5.239^7 (3)
s3dkt3m2	7.806^1	7.847^1 (3)	7.285 (2)	7.269^4 (3)	7.503^5 (3)
tbdlinux	6.428^1	6.086^2 (10)	5.027 (4)	2.362^5 (50)	5.433^3 (100)

1:	Using the CRS datastructure
2:	Using the Incremental CRS datastructure
3:	Using the Zig-zag CRS block datastructure
4:	Using the Zig-zag CCS block datastructure
5:	Using the Adapted CRS block datastructure
6:	Using the Adapted CCS block datastructure
7:	Using the Separators-last block datastructure

Table 2: Time of an SpMV multiplication on the naturally ordered matrices, as well as on reordered matrices, both in 1D and 2D, for the group of smaller matrices. Time is measured in milliseconds, and only the best average running times from $p = 2$–$7, 10, 50, 100$ are given, followed by the number of parts p (between parentheses) used to obtain the fastest SpMV. Footnotes indicate which datastructure was employed. The '1D & Blocking' category always uses the CRS block datastructure.

Matrix	Natural	1D [14]	1D & Blocking	2D Mondriaan	2D Fine-grain
stanford	18.99^1	9.92^1 (10)	9.52 (9)	9.35^1 (8)	9.73^1 (10)
stanford_berkeley	20.93^2	20.10^2 (4)	19.26 (9)	19.18^5 (4)	19.41^7 (9)
cage14	69.36^1	75.47^2 (2)	76.48 (2)	74.37^2 (2)	75.13^2 (2)
wikipedia-2005	248.63^1	154.93^1 (10)	142.32 (9)	115.56^1 (8)	124.18^5 (8)
wikipedia-2006	688.42^1	378.30^1 (9)	302.35 (9)	256.43^1 (9)	267.47^4 (8)

Table 3: Time of an SpMV multiplication on the naturally ordered matrices, as well as on reordered matrices, both in 1D and 2D, for the group of larger matrices. Time is measured in milliseconds, and only the best average running times from $p = 2$–10 are given, together with the number of parts p resulting in the fastest SpMV. Footnotes indicate the exact datastructure used, as given in Table 2.

datastructures. Other research into sparse blocking, such as found in [1], may be integrated as well. 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 datastructures backing the permuted sparse matrix increases. This overhead is linear in p, and this means that 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. Sparse blocking in this way, was not found to be absolutely necessary to obtain greater speedups when using the Mondriaan partitioning scheme.

Various items for future research can be readily identified:

- The block-based datastructure presented here is very basic, and a more advanced datastructure might be introduced to lower overhead with increasing p, and thus gain even more efficiency.

- 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 even 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 datastructure, also within a simple or advanced block-based datastructure.

- The speed of building up the datasets, although already greatly improved when compared to the results in the original paper [14], 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.

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] 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.

[5] 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.

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

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.