CACHE-OBLIVIOUS SPARSE MATRIX–VECTOR MULTIPLICATION BY USING SPARSE MATRIX PARTITIONING METHODS

A.N. YZELMAN* AND ROB H. BISSELING†

Abstract. In this article, we introduce a cache-oblivious method for sparse matrix–vector multiplication. Our method attempts to permute the rows and columns of the input matrix using a hypergraph-based sparse matrix partitioning scheme so that the resulting matrix induces cache-friendly behaviour during sparse matrix–vector multiplication. Matrices are assumed to be stored in row-major format, by means of the compressed row storage (CRS) or its variants incremental CRS and zig-zag CRS. The zig-zag CRS data structure is shown to fit well with the hypergraph metric used in partitioning sparse matrices for the purpose of parallel computation. We present the separated block-diagonal (SBD) form as the appropriate matrix structure for cache enhancement.

We use a k-way set-associative idealised cache model, and we have implemented a run-time cache simulation library enabling us to analyse cache behaviour for arbitrary matrices and arbitrary cache properties during matrix–vector multiplication within this model. The results of these simulations are then verified by actual experiments run on various cache architectures. In all these experiments, we use the Mondriaan sparse matrix partitioner in one-dimensional mode. The savings in computation time achieved by our matrix reorderings reach up to 50 percent, in case of a large link matrix.

Key words. matrix–vector multiplication, sparse matrix, parallel computing, recursive bipartitioning, cache-oblivious.

1. Introduction.

Many important linear algebra kernels typically take a performance hit on modern cache-based computer architectures [27, 18, 7] due to inefficient use of the system cache. Cache use is best when data from main memory is stored contiguously and accessed in a single straight pass. Each data item is preferably used many times, and is not needed in further computation afterwards. Unfortunately, many non-trivial applications require to jump through data in main memory, even if data is stored contiguously. A notorious example is sparse matrix–vector (MV) multiplication. In these cases, we can minimise the number of jumps or otherwise improve cache efficiency. As recent work in the field of the BLAS by Goto and van de Geijn [14] has shown, tweaking algorithms to specific architectures results in large speedups.

Such *cache-aware* algorithms have as a disadvantage the need to adapt existing code to new architectures, every time they become available. To resolve this, auto-tuning software libraries have become a focus for much research [33, 20], most notably FFTW (for fast Fourier transforms) [10, 11], OSKI (basic sparse BLAS) [32], and ATLAS (dense BLAS) [34]. These libraries may run various benchmarks upon installation to help optimise algorithms for the hardware specifics of the target machine, or may even attempt this *during* real-time execution.

Another approach is to design algorithms in such a way that optimal cache efficiency is achieved on any (regular) machine architecture. These *cache-oblivious* algorithms have been researched to some extent, and for some applications have been shown to obtain asymptotically optimal bounds [12, 3].

In this article, we propose a cache-oblivious sparse matrix–vector multiplication algorithm. We use techniques from load-balancing for parallelism to increase data locality for cache reuse. Before introducing our method, we first describe our sparse matrix storage format in Section 2. We then proceed in Section 3 with an analysis of cache efficiency and extend this knowledge in Section 4 to obtain a better performing scheme. We link this scheme to hypergraph partitioning theory, and formulate the basic idea of our MV multiplication method. In Section 5, we formalise this idea in terms of matrix permutations.

Theoretical cache efficiency analysis depends very much on the exact matrix input; even when this is known, analysis still is difficult due to algorithm complexity. Experimental results on actual machines, on the other hand, suffer from inflexibility in the cache parameters, such as cache size, leading to observations with only a limited range of applicability. This motivated the development

*Mathematical Institute, Utrecht University, P.O. Box 80010, 3508 TA Utrecht, The Netherlands (A.N.Yzelman@uu.nl).

†Mathematical Institute, Utrecht University, P.O. Box 80010, 3508 TA Utrecht, The Netherlands (R.H.Bisseling@uu.nl).

$m \times n$	matrix dimensions
A, x, y	matrix and vectors
C	the cache
S	cache size in bytes
L_S	cache line size in bytes
L	number of cache lines ($L = S/L_S$)
w	number of data words in a cache line
k	number of subcaches

TABLE 1.1
List of parameters used throughout this paper.

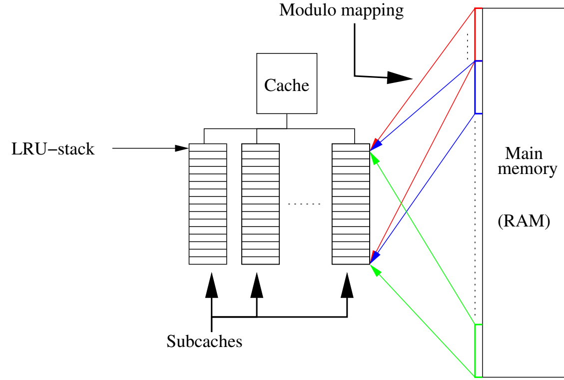

FIG. 1.1. *Schematic view of the k-way set-associative cache model with modulo mapping of main memory to subcache.*

of our run-time cache simulator, which we introduce in Section 6. We finish with experimental results on actual hardware and general conclusions in Sections 7 and 8, respectively.

Throughout the paper, we assume a k-way set-associative idealised cache model, with k subcaches, where the case $k \to \infty$ corresponds to the ideal-cache model introduced in [12]. In our model, we assume a total cache size equal to S bytes, and a line size of L_S bytes. The total number of cache lines is thus $L = S/L_S$. A cache C can then be modelled as an $L/k \times k$ matrix, where entries c_{ij} correspond to individual cache lines and each matrix column to a subcache. The number i is commonly called the *Set ID*, while j is called the *Line ID*. Table 1.1 provides an easy reference of the parameters introduced here.

A further assumption we make regarding the cache model, is that data is modulo-mapped into one of the k subcaches by the Least Recently Used (LRU) policy. This means that the data in main memory (RAM) at the bytes in the range $[rL_S, (r+1)L_S)$, for any $r \in \mathbb{N}_0$, is mapped to the Set ID $i = r \mod \frac{L}{k}$ (hence the name *modulo mapped*). The Line ID j is determined by selecting the cache line containing the least recently used data item, and the cache line we select thus becomes c_{ij}. See also Figure 1.1 for an illustration of this model. For a further introduction to cache architecture, one may consult [29].

2. Incremental CRS. A standard storage scheme for sparse matrices is the Compressed Row Storage (CRS) format [2]. This format utilises three arrays: one array for storing individual nonzero matrix entry values (*nzs*), one for storing the column index of those nonzeros (*col_ind*), and one for indexing where in the previous two arrays individual rows start (*row_start*).

The arrays *nzs* and *col_ind* each require $\text{nz}(A)$ space in memory, where $\text{nz}(A)$ denotes the number of nonzeros of the $m \times n$ sparse matrix A. The array *row_start* requires only m words of space, bringing the total up to $2\text{nz}(A) + m$. An algorithm to perform the MV multiplication $y = Ax$ on a matrix stored in CRS format is given in Algorithm 1. Note that we index vector and matrix elements starting from 0: the matrix elements are a_{ij}, $0 \le i < m$ and $0 \le j < n$.

A drawback of standard CRS implementation becomes immediately clear on line 4 of Algo-

rithm 1. A for-loop was started on line 3 along indices k relevant to the row i being processed. The array *nzs* is straightforwardly accessed according to k, in contrast to x which is accessed according to $j =$*col_ind[k]*; this kind of *index translation* causes much instruction overhead.

Koster proposes in his master's thesis [23] to store column index *increments* instead of storing column indices. Hence, an MV algorithm only needs to increment a local column variable instead of using index translation. An MV algorithm utilising this Incremental CRS (ICRS) storage scheme is given in Algorithm 2. Note that an increment in row index is signalled by causing j to overflow (i.e., $j \geq n$) so that $j - n$ corresponds to the actual column index corresponding to the first nonzero in the new row. The increase in the row index i after each column overflow is stored in the array *row_jump*.

Changing CRS to handle data incrementally does not change its memory requirements; *nzs* and the new difference array *diff* both use $\mathrm{nz}(A)$ space while *row_jump* is still of size m in the worst case. The total memory requirement thus also equals $2\mathrm{nz}(A) + m$.

Although the pseudocode of Algorithm 2 is a few lines larger than that of Algorithm 1, the lack of index translation and ability to implement the latter algorithm efficiently using pointer arithmetic causes the instruction overhead to drop dramatically [23, Figure 2.5].

Algorithm 1 Matrix–vector multiplication using CRS

Input: *nzs*, *col_ind*, and *row_start* corresponding to some sparse matrix A; the dimensions m and n of A; the dense input vector x required to calculate Ax.
Output: A dense vector y, where $y = Ax$.

1: Allocate y of size m and initialise: $y = \mathbf{0}$
2: **for** $i = 0$; $i < m$; $i = i + 1$ **do**
3: **for** $k =$*row_start*$[i]$; $k <$*row_start*$[i+1]$; $k = k + 1$ **do**
4: $j =$*col_ind*$[k]$
5: $y[i]= y[i]+$*nzs*$[k]*x[j]$
6: **end for**
7: **end for**
8: **return** y

Algorithm 2 Matrix–vector multiplication using ICRS

Input: *nzs*, *diff*, and *row_jump* corresponding to some sparse matrix A; the dimensions m and n of A; the dense input vector x required to calculate Ax; the number of nonzeros $\mathrm{nz}(A)$.
Output: A dense vector y, where $y = Ax$.

1: Allocate y of size m and initialise: $y = \mathbf{0}$
2: $i =$*row_jump*$[0]$, $j =$*diff*$[0]$, $k = 0$, $r = 1$
3: **while** $k < \mathrm{nz}(A)$ **do**
4: $y[i]= y[i]+$*nzs*$[k]*x[j]$
5: $k = k + 1$
6: $j = j+$*diff*$[k]$
7: **if** $j \geq n$ **then**
8: $j = j - n$
9: $i = i+$*row_jump*$[r]$
10: $r = r + 1$
11: **end if**
12: **end while**
13: **return** y

3. Cache performance for dense and sparse MV multiplication. Cache performance is deemed optimal if an algorithm does not cause more cache misses than necessary. This is most

easily achieved when data is stored contiguously and accessed only *once* with *stride one*; that is, consecutively from front to back. Contiguous access is necessary since multiple data words may fit into a single cache line. However, in most cases previously accessed data is reused which complicates finding an optimal access pattern.

We consider the dense case first. For simplicity, we assume that one data word fits exactly in one cache line, i.e., $w = 1$. Figure 3.1 shows a small example of cache effects in the context of dense MV multiplication, where we assume the perfect cache idealisation $k \to \infty$; that is, the cache C is given by a $1 \times L$ matrix and cache lines are solely selected by use of the LRU policy. The algorithm starts off with $y_0 = y_0 + a_{00}x_0$, pushing the elements from x, A and y onto the top of the LRU stack. If the cache size S is such that the cache can contain exactly two data words, i.e., $L = 2$, we see that x_0 is pushed out of the cache at the end of this instruction.

The next instruction is $y_0 = y_0 + a_{01}x_1$, and again the variables x and A are brought in first, pushing y_0 out of the cache just before it was needed again. If the cache could contain exactly three data words instead of two, y_0 would still have been available. Note that when the MV algorithm reaches the next row (i.e., executes $y_1 = y_1 + a_{10}x_0$), then, depending on the cache size S, there are two possibilities: either x_0 is still in cache and no cache miss occurs, or x_0 was evicted and it must be brought back in. This results in $\mathcal{O}(n)$ cache misses on x, *on each row*; and hence $\mathcal{O}(mn)$ cache misses during the whole algorithm.

A way to prevent those cache misses on x is to limit the number of subsequent accesses using some blocking parameter q so that after processing x_{q-1}, the MV algorithm proceeds with the next row. Obviously, we choose q so that elements from x are not prematurely evicted. When the last row has been processed, the algorithm jumps back to the first row and goes on to process x_q until it reaches $x_{\max\{2q-1,n-1\}}$, et cetera.

At this point, y is repeatedly accessed from index 0 until $m - 1$, as each column-block is processed. Thus, it is possible that elements from y are prematurely evicted in the same way as originally for x, causing $\mathcal{O}(m)$ cache misses each time a column block is processed. This already is quite an improvement since we typically have far fewer column blocks than we have rows. Although the problem is less severe, we still may want to limit access on y using some blocking variable p (which need not be equal to q) and apply the same trick on the row indices. This method is appropriately named *blocking* since, in effect, we are subdividing A into blocks of size $p \times q$ for which an MV algorithm does not incur unnecessary cache misses.

Let us now discuss the sparse case. With the dense case described above in mind, we observe the following regarding general sparse MV multiplication algorithms:

- every matrix element is accessed exactly once during MV multiplication;
- the order of matrix element access determines the access patterns of the x and y vectors;
- the order of matrix element access is determined by its storage scheme.

From this last point, we expect that CRS and ICRS obtain the same performance in terms of cache efficiency since the order of element access remains unchanged; thus for readability we will only discuss the CRS ordering. We may analyse a sparse MV multiplication algorithm using the ordering induced by CRS in much the same way as in the dense case; Figure 3.2 shows that the LRU stack during a sparse MV multiplication using CRS behaves quite similarly to the stack of a dense MV multiplication. The major difference is that most of the time, the column indices are not consecutive. This greatly increases the difficulty of predicting when cache misses occur; it all depends on the relative column-wise positions of the nonzeros in A. Hence the blocking parameters p and q cannot be determined solely from m, n, and the cache size S, but also requires information on the structure of A. Cache-aware blocking thus becomes much harder to apply, motivating run-time cache-aware auto-tuning kernels such as OSKI [32].

4. Zig-zag CRS data structure and its connection to hypergraphs. A first, cache-oblivious way to reduce the number of cache misses in CRS, is to prevent jumping from the last to the first column when a row increment occurs. Instead, we could just start at the last column of that row, and then process nonzeros in reverse order until we arrive at the first column. At the next row increment we repeat this recipe. Assuming all rows are non-empty, we thus process in the standard increasing order on even-numbered rows, and in decreasing order on odd-numbered

$$x_0 \Longrightarrow \begin{array}{c} a_{00} \\ x_0 \\ \hline \end{array} \Longrightarrow \begin{array}{c} y_0 \\ a_{00} \\ \hline x_0 \end{array} \Longrightarrow \begin{array}{c} x_1 \\ y_0 \\ \hline a_{00} \\ x_0 \end{array} \Longrightarrow \begin{array}{c} a_{01} \\ x_1 \\ \hline y_0 \\ a_{00} \\ x_0 \end{array} \Longrightarrow \begin{array}{c} y_0 \\ a_{01} \\ \hline x_1 \\ a_{00} \\ x_0 \end{array}$$

Fig. 3.1. *LRU stack progression during the first steps of dense MV multiplication. Whenever a new data element is pushed onto the stack, it is displayed on top. Older elements thus appear lower in this figure, with the least recently used element at the bottom. Note that if all memory lines below the bars are evicted, y_0 is evicted at step 4 (denoted by the fourth arrow), while it is immediately reused the step after.*

$$x_{j_1} \Longrightarrow \begin{array}{c} a_{i_1,j_1} \\ x_{j_1} \\ \hline \end{array} \Longrightarrow \begin{array}{c} y_{i_1} \\ a_{i_1,j_1} \\ \hline x_{j_1} \end{array} \Longrightarrow \begin{array}{c} x_{j_2} \\ y_{i_1} \\ \hline a_{i_1,j_1} \\ x_{j_1} \end{array} \Longrightarrow \begin{array}{c} a_{i_2,j_2} \\ x_{j_2} \\ \hline y_{i_1} \\ a_{i_1,j_1} \\ x_{j_1} \end{array} \Longrightarrow \begin{array}{c} y_{i_2} \\ a_{i_2,j_2} \\ \hline x_{j_2} \\ y_{i_1} \\ a_{i_1,j_1} \\ x_{j_1} \end{array}$$

Fig. 3.2. *LRU stack progression during the first steps of sparse MV multiplication using CRS. The indices i_r and j_r are the row and column index of the* r*th nonzero element from A. Note that for most r, $j_r \neq j_{r+1}$, while i_r only differs from i_{r+1} when the nonzeros at a given row are exhausted and the algorithm moves to the next row.*

rows. We call the resulting data structure *zig-zag-CRS* (ZZ-CRS). Figure 4.1 illustrates both the CRS and zig-zag-CRS orderings.

When using the zig-zag scheme, a cache that is too small will not cause the full $\mathcal{O}(n)$ cache misses on the vector x (assuming $w = 1$) for each row in the dense case. Instead, we incur only $\mathcal{O}(n - L)$ cache misses, where L is the total number of cache lines. For the general case $w \geq 1$, where more than one data word may fit into a cache line, we have $\mathcal{O}(\frac{n}{w} - L)$ misses instead of $\mathcal{O}(\frac{n}{w})$. This improvement may seem small, but we do not advocate zig-zag ordering for this reason alone. In fact, we shall now show that by using this particular storage scheme, we can find similarities between established matrix partitioning schemes and the cache dynamics we have just observed. To this end, we quickly review basic hypergraph-based partitioning methods for sparse matrices.

It is possible to represent a sparse matrix A as a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$, as follows. We let a vertex $v_j \in \mathcal{V}$ correspond to the jth column of A. The net (or *hyperedge*) $n_i \in \mathcal{N}$ is a subset of $\mathcal{V}$ that contains exactly those vertices v_j for which $a_{ij} \neq 0$. This is called the *row-net* model [21], since each matrix row is represented by a net. Other models to represent sparse matrices include the *column-net* [21] and *fine-grain* model [5]. In the column-net model, the roles of the row and column indices are interchanged as compared to the row-net model. The fine-grain model differs from the previous models in that the vertices correspond to individual nonzeros a_{ij}. A net then contains nonzeros sharing the same row or column.

We will use the row-net model because of the rowwise storage of CRS. Suppose we partition the vertex set $\mathcal{V}$ into subsets $\mathcal{V}_0, \mathcal{V}_1, \ldots, \mathcal{V}_{p-1}$, where we have that the $\mathcal{V}_s \subset \mathcal{V}$ are pairwise disjoint while $\cup_{s=0}^{p-1} \mathcal{V}_s = \mathcal{V}$. Given a specific partitioning, it can happen that the vertices in a given net n_i are distributed over several subsets, resulting in a *cut net*. Let us denote the number of partitions over which the ith net is cast, by λ_i; this is also called the *connectivity* of the ith net.

Given the net cost c_i, which is a factor expressing the relative importance of each net, and the net connectivity λ_i, a cost can be assigned to a given partitioning by means of a cost function:

$$\sum_{i: n_i \in \mathcal{N}} c_i(\lambda_i - 1). \tag{4.1}$$

This cost function is commonly called the $\lambda - 1$ metric. Generally, in our matrix partitioning, we assume there is no preference of any kind as to which matrix rows are cut, and thus take $c_i = 1$ for all i. While partitioning a hypergraph in this manner, we strive to minimise the function (4.1).

When applying hypergraph partitioning with parallel distribution of sparse matrices in mind, we also want to maintain a good load balance; that is, the number of nonzeros in each partition

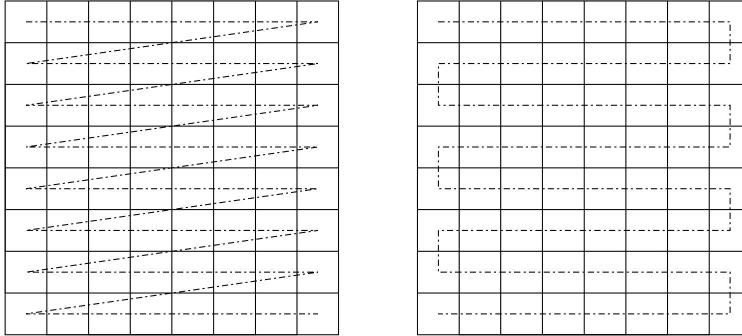

FIG. 4.1. *The CRS (left) and ZZ-CRS (right) orderings. The dash-dotted lines show the order in which matrix nonzeros are stored.*

should deviate only by a small factor from the average number. To achieve this, each vertex v_j is weighted with $w_j = \mathrm{nz}_j$, the number of nonzeros in the jth column of the matrix. We then also minimise:

$$\max_{0 \leq s < p} \sum_{j: v_j \in \mathcal{V}_s} w_j. \tag{4.2}$$

Consider a recursive algorithm which repeatedly splits sets of vertices $\mathcal{V}$ into two subsets and aims to construct a partitioning of a hypergraph this way. A quantity that is useful when reasoning about load balance is

$$\epsilon = \frac{\max\{|\mathcal{V}_0|, |\mathcal{V}_1|\}}{(|\mathcal{V}_0| + |\mathcal{V}_1|)/2} - 1. \tag{4.3}$$

When $\epsilon = 0$ we have perfect load balance; we will denote ϵ as the *load-imbalance factor*.

The demands of a perfect load balance and a minimised total partition cost usually conflict with each other. In an attempt to construct a partitioning which is acceptable in terms of both criteria, we may decide to allow a load imbalance ϵ' not larger than some pre-defined imbalance parameter ϵ so that $\epsilon' \leq \epsilon$. A software package implementing this approach in sparse matrix partitioning is Mondriaan [31]. Mondriaan splits the matrix in both dimensions, each time choosing the best from a split in the row direction and the column direction. We use this partitioner in our experiments in Section 7, although not in its full 2D generality. We only need splits in the column direction, and hence use Mondriaan in 1D mode.

When considering parallel sparse MV multiplication, partitions correspond to processors. In a full 2D partitioning, a cut column in some partitioning indicates that the required component of the vector x is shared among different processors. This results in communication, since the components of x are distributed among the processors to achieve *scalability*; processors that require components from x which they do not locally store, have to request these values from other processors. This happens before any computational work, and is called the *fan-out* of the parallel MV multiplication. The same goes for components of y: some processors may have to add values to elements of y not governed by themselves and thus also require to communicate. This happens after the computational work and is called *fan-in*.

The rationale for the cost function (4.1) is that if we attempt to minimise communication, we try to avoid any cut nets; ideally we have $\lambda_i = 1$ for all i. And even if we do have to cut a net, we should cut it in as few parts as possible so that the number of processors which have to communicate with each other, is minimised.

Now, consider the following bipartitioning routine working on a row-net hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ representation of some sparse matrix A, where vertices represent columns and rows represent nets. We partition $\mathcal{V}$ into $\mathcal{V}_0, \mathcal{V}_1$ while taking into account the cost function and load imbalance. Note that the indices of the columns corresponding to the vertices in each partition do

not have to be consecutive. In effect, we now have partitioned the matrix A columnwise. We also induce a partitioning on the *nets* in $\mathcal{N}$ corresponding to this partitioning. The set of nets with vertices only in $\mathcal{V}_0$ is denoted by $\mathcal{N}^-$. The set of nets with vertices only in $\mathcal{V}_1$ is denoted by $\mathcal{N}^+$. The set of cut nets, i.e., nets which have vertices in both partitions, is denoted by $\mathcal{N}^c$.

Let us consider the case in which an MV multiplication algorithm visits matrix elements in the following order. First, we process the rows corresponding to nets in $\mathcal{N}^-$, then those in $\mathcal{N}^c$, and finally those in $\mathcal{N}^+$. The matrix elements in subsequent rows are furthermore processed in zig-zag order. Since such an algorithm, like ZZ-CRS based MV multiplication, visits matrix rows in succession, no unnecessary cache misses are incurred on the vector y.

To minimise the cache misses on x, we define

$$p = \frac{n}{wL}, \tag{4.4}$$

which can be interpreted as the number of caches that would be needed to store the complete input vector x. (Note that wL equals the number of data words that can be stored by the cache.) This value p defines a natural number of partitions (or processors) for storing a row of the matrix. If the matrix were dense, the number of cache misses per row would be $n/w - L = L(p-1)$, provided the zig-zag ordering is used. This would also be the number of misses for a sparse matrix that is relatively dense and has its nonzeros well-spread, e.g. in a random sparsity pattern.

If the matrix row i is only nonzero in an interval $j \in [l_i, r_i)$, the number of cache misses for that row is at most $(r_i - l_i)/w - L$, provided the interval is the same as for the previous or next row (this is needed for the zig-zag ordering to be beneficial). Here, we replace the row length n in the right-hand side of eqn (4.4) by the interval length $r_i - l_i$, giving a number of nonempty row parts (of the size of a complete cache) equal to $\lambda_i = (r_i - l_i)/(wL)$, and the corresponding number of cache misses is then $L(\lambda_i - 1)$. If the interval is dense, this upper bound is exact. If the interval is only relatively dense, with each cache line of w data words having at least one nonzero, the bound is still exact. Another important case is where the row is uncut after partitioning the matrix into p sets of columns. This happens very often for a good partitioner, as this is the main aim. For such an uncut row no cache misses are incurred, again under the assumption that the zig-zag ordering works, and this corresponds exactly to the bound $L(\lambda_i - 1)$ with $\lambda_i = 1$, which is just zero.

As a result, we can view the matrix as being partitioned into blocks of n/p columns, and the corresponding communication volume of parallel sparse MV multiplication times L gives an upper bound on the number of cache misses. Thus, minimising the communication volume in the $\lambda - 1$ metric using hypergraph partitioning is expected to improve cache utilisation. This is the main motivation for our approach.

In practice, we are oblivious of the values for w, L and do not know the appropriate value of p. It does not harm the cache behaviour to partition beyond the value of p given in eqn (4.4). Hence, we partition infinitely, $p \to \infty$, or as far as we can go. If we perform the partitioning by recursive bipartitioning, and reorder the matrix accordingly, see Section 5, we create beneficial matrix structure at all values of p. This way of reordering also minimises cache misses on multi-level cache hierarchies since for lower p we optimise for the larger, upper level caches while for higher p we optimise for the smaller, lower-level caches, such as the L1 cache, without harming the upper-level optimisations.

Putting the collection of cut rows between those of $\mathcal{N}^-$ and $\mathcal{N}^+$ ensures that data loaded in by first visiting columns corresponding to $\mathcal{V}_0$ are still available when processing $\mathcal{N}^c$. Since $\mathcal{N}^c$ also contains data from $\mathcal{V}_1$, cache performance deteriorates as data corresponding to $\mathcal{V}_0$ is swapped out in favour of those of $\mathcal{V}_1$. We then proceed with $\mathcal{N}^+$, which only deals with data corresponding to $\mathcal{V}_1$; thus purging the remaining data corresponding to $\mathcal{V}_0$ while taking advantage of the $\mathcal{V}_1$ data already brought into cache. This procedure is preferred to, say, processing rows in $\mathcal{N}^c$ strictly after those in $\mathcal{N}^-$ and $\mathcal{N}^+$, as is done in matrix reorderings for other purposes, e.g. in nested dissection [13] for Cholesky factorisation [16] or LU factorisation, see e.g. work by Aykanat et al. [1] which uses hypergraphs to reorder rectangular matrices for sparse LU or QR factorisation, and recent work by Grigori et al. [15], which uses hypergraphs to reorder unsymmetric matrices for sparse

LU factorisation. For factorisation, it makes sense to place cut rows or columns last, since this postpones and prevents *fill-in*, the creation of new nonzeros. Another example where cut rows come last, is the work on the package Monet [17], which tries to create a bordered block-diagonal form based on hypergraph partitioning with the cut-net metric (i.e., eqn (4.1) with $\lambda_i - 1$ replaced by 1 if $\lambda_i \geq 2$, and 0 otherwise). If cut rows come last, they will bring into cache some elements which may also have been used by $\mathcal{N}^+$, but certainly all data corresponding to $\mathcal{V}_0$ will have been evicted from cache. This data will have to be reloaded.

Concluding, our method of placing cut rows in the middle ensures, with high probability, that we have cache hits when moving from one net to another in the set of cut nets. This allows for a gradual transition from one set of rows to the other. This transition is expected to be smooth if the rows are sparse, and if the partitioner manages to keep the number of cut rows small.

The idea of using graphs or hypergraphs to help optimise cache efficiency has been proposed before: Strout and Hovland introduced a similar concept in [26, 25], but they use (regular) graph partitioning to achieve a better data ordering, directly deducing a better ordering of data in memory. They use hypergraph partitioning to improve inter-iteration locality; this can be done in some iterative algorithms, where it is not necessary to finish one iteration before partially continuing with the next. This can be exploited to improve cache locality.

5. Matrix Permutations.

Consider now a single bipartition of A defined by $\mathcal{V}_0$, $\mathcal{V}_1$, $\mathcal{N}^-$, $\mathcal{N}^c$ and $\mathcal{N}^+$. We can then define a permuted matrix A_1 as in Figure 5.1. The permutation of columns can be written in linear algebra form by AQ, where A is the original matrix and Q a permutation matrix of corresponding dimensions. Similarly, the permutation of rows to achieve the net-based ordering $\mathcal{N}^-, \mathcal{N}^c, \mathcal{N}^+$ can be written as a left-side multiplication with another permutation matrix P.

Let us denote the $n \times n$ identity matrix by $I = (e_1|e_2|\ldots|e_n)$. Then we have that the right-sided permutation matrix is given by $Q = (e_{q_0}|e_{q_1}|\ldots|e_{q_{n-1}})$, for some index permutation $(q_i)_{i\in[0,n-1]}$ of the tuple $(0, 1\ldots, n-1)$. Similarly, $P = (e_{p_0}|e_{p_1}|\ldots|e_{p_{m-1}})^T$, with index permutation $(p_i)_{i\in[0,m-1]}$. Recall that since P, Q are permutation matrices, $P^{-1} = P^T$ and $Q^{-1} = Q^T$. Of course, the indices at the start of (q_i) correspond to the columns in $\mathcal{V}_0$ and are followed by those corresponding to the columns in $\mathcal{V}_1$; the order of (p_i) is similarly induced by the net order. The permuted matrix A_1 after one bipartitioning step is then given by $A_1 = PAQ$.

Subsequent recursive bipartitioning results in similar row and column permutations on disjoint submatrices of A. We can therefore still represent the matrix A_r after r recursive steps by $A_r = PAQ$, for certain P and Q, and our modified MV multiplication routine can be represented as follows:

$$y = Ax = P^T A_r Q^T x, \quad \text{so } \tilde{y} = A_r \tilde{x} \quad \text{with } \tilde{y} = Py, \quad \tilde{x} = Q^T x. \tag{5.1}$$

One can see a clear similarity to general preconditioning techniques. Also by using this notation, we can extend our partitioning mechanism to other related linear algebra kernels besides MV multiplication:

- We can express $y = Ax + \beta z$ as follows: $y = P^T(A_r Q^T x + \beta P z)$, so that the procedure can be written as a standard MV multiplication $\tilde{y} = A_r \tilde{x} + \tilde{z}$ with $\tilde{y} = Py$, $\tilde{x} = Q^T x$, $\tilde{z} = Pz$.
- Also, $y = A^T x$ can be expressed as a permutation: $y = Q A_r^T P x$ so that we have $\tilde{y} = A_r^T \tilde{x}$ with $\tilde{y} = Q^T y$, $\tilde{x} = Px$.
- Finally, $y = AA^T x$ is also possible: $y = P^T A_r A_r^T P x$ so that $\tilde{y} = A_r A_r^T \tilde{x}$ with $\tilde{y} = Py$, $\tilde{x} = Px$.

This shows that one can apply our cache-oblivious matrix reordering scheme by only permuting input matrices and vectors, and run the corresponding standard BLAS kernels (or similar) on the permuted input data. The underlying BLAS software can even be a cache-aware package, such as OSKI [32], thus perhaps increasing cache efficiency even beyond what is possible with either OSKI alone or with our method combined with a simple BLAS kernel.

We can also show that some linear algebra kernels cannot be executed directly using our partitioning method. Notably this applies to the kernel used in power method solvers, namely $y = A^s x$,

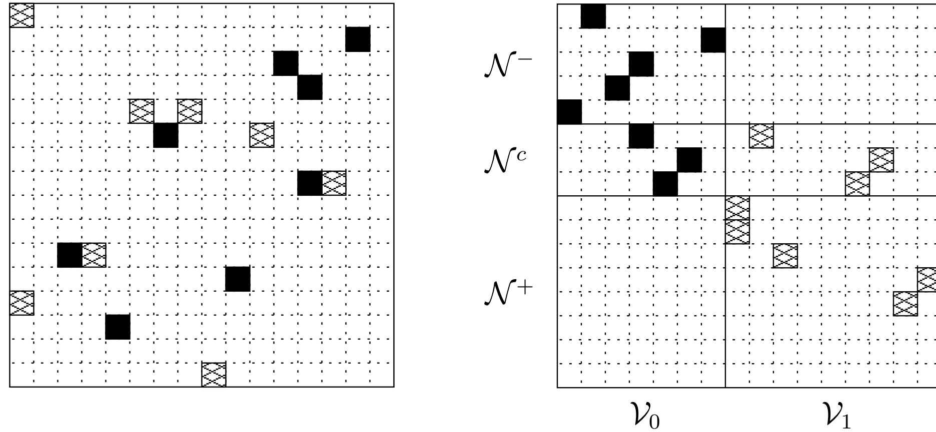

Fig. 5.1. *Original matrix (left) and permuted matrix (right) after one bipartitioning step. Shaded and black squares denote nonzero elements. Black columns denote vertices in $\mathcal{V}_0$ whereas the shaded columns denote vertices in $\mathcal{V}_1$.*

$s \in \mathbb{N}$, $s > 1$. When supplying a permuted matrix, this kernel becomes $y = (PAQ)(PAQ)^{s-1}x$; between instances of the original matrix A we thus find the matrix QP. Since generally $QP \neq I$, we cannot simply feed the matrix to an arbitrary $A^s x$ kernel; we need a translating step between successive MV multiplications, which would be quite inefficient. This is for instance the case for the power method used in Google PageRank computations [4]. However, if we instead consider an alternative page ranking method based on hubs and authorities, called the HITS method [22] (see also the book [24, p. 117]), the power method is applied to the matrices AA^T and A^TA with A a link matrix. After reordering of A, the matrix $(AA^T)^s$ becomes $(PAQQ^TA^TP^T)^s = P(AA^T)^sP^T$, and similarly for A^TA. In this case, our reordering scheme can be used without penalty.

Determining the column permutation is straightforward: the recursive bipartitioning yields an implicit order of subsets of $\mathcal{V}$. Indeed, if we recursively bipartition an arbitrary number of times, we may denote the resulting subsets by describing the path followed in their construction; that is, at each recursive step a vertex subset either remains intact, is distributed left (0), or distributed right (1). A subset can thus be denoted like (0110), meaning it was first distributed left, then right, right again, and finally again on the left side. Interpreting this representation as a binary number gives us a natural ordering on the vertex sets. Note that if we bipartition infinitely (that is, continue until each subset contains exactly one vertex), we end up with n subsets each containing a single column. Reordering those subsets according to their binary representation and reading out the column indices of the corresponding vertices yields the final column reordering.

Determining the row permutation is more difficult. To do this, we look at the set of *possible* final locations Q_i of each row i after permutation. At the start of the reordering, we of course have that $Q_i = [0, m-1]$, which is the full range of matrix rows. After the first bipartitioning, we obtain three net subsets $\mathcal{N}^{\{-,c,+\}}$. The rows corresponding to those in $\mathcal{N}^-$ can then end up in $[0, |\mathcal{N}^-| - 1]$, those in $\mathcal{N}^c$ in $[|\mathcal{N}^-|, m - |\mathcal{N}^+| - 1]$ and those in $\mathcal{N}^+$ in $[m - |\mathcal{N}^+|, m-1]$; see also Figure 5.1. This procedure can be repeated after each following bipartitioning; see Figure 5.2. There, each horizontal straight line gives new row boundaries for the rows, with Q_i containing those boundary rows. After infinite bipartitioning, we can then deduce a row ordering from the Q_i. Note that such an ordering need not be unique; for some i, $|Q_i| > 1$ is possible, even if $m \leq n$.

Figure 5.3 shows the overall structure that is obtained by recursively bipartitioning the sparse matrix and placing the cut rows in the middle. In analogy with the *bordered block-diagonal* (BBD) form [9], we call the matrix structure obtained after a sequence of recursive bipartitionings the *separated block-diagonal* (SBD) form. At each level of the recursion, the cut rows separate the two sets of uncut rows from each other. Note that the cut rows may have internal structure, which is not depicted. This structure is created by using the $\lambda - 1$ metric, which tries to prevent further cuts inside already cut rows. (The cut-net metric does not have this advantageous property.)

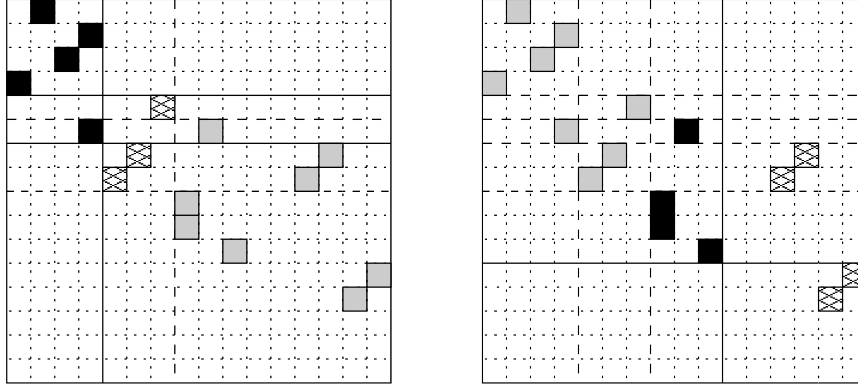

FIG. 5.2. *The reorderings after one resp. two further bipartitionings after the bipartitioning in Figure 5.1. Grey squares denote nonzeros not considered in the latest bipartitioning step.*

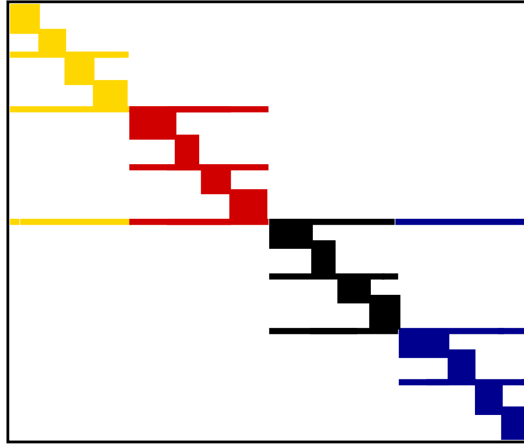

FIG. 5.3. *Separated block-diagonal (SBD) structure of a sparse matrix obtained by recursively partitioning in the column direction and moving the cut rows to the middle. The recursion has been stopped when* 16 *diagonal blocks have been created. The four colours indicate the block structure at* $p = 4$.

6. Cache Simulation. The performance of sparse MV multiplication kernels depends on the actual structure of the sparse matrix itself. This dependency on matrix input makes it hard to analyse our method in terms of the cache model we introduced; we therefore implemented a cache simulator library, which enables us to simulate cache dynamics accurately within the theoretical model. Using this library, we are able to analyse *theoretical* cache performance on any sparse input matrix, and correlate these results to actual wall-clock timings in Section 7.

Our cache simulator, implemented in C++, uses *run-time* memory allocation and memory access wrapper functions to catch and process data access. Upon allocation, a pointer to the allocated region $\tilde{x}$ is stored. This pointer is regarded as a physical RAM address. The address $x = \tilde{x} - (\tilde{x} \mod L_S)$ is then calculated. We use this address instead of $\tilde{x}$ to account for the possibility that several variables share the same cache line; we only store the start of the cache line brought into cache. Upon accessing s bytes from $\tilde{x}$, the simulator proceeds with calculating to which cache set ID x maps, that is, $i = \frac{x}{L_S} \mod \frac{L}{k}$. Let us first assume that $\tilde{x} - x + s \leq L_S$ so that one cache line suffices. The address x is then pushed onto the stack corresponding to the set ID i.

The act of pushing an address onto the stack is where the simulator detects if a cache hit or miss has occurred. For every set ID i, we have stored a stack of size k. When pushing x onto such a stack, either x already is in cache, or it is not. If it is present, we have a hit and x is removed from its current location in the stack and once again inserted at the front of the stack. If it is

not present, we have a miss and x is inserted at the front. If this would cause the stack size to overflow (i.e., become larger than k items), the address at the back of the stack is evicted. If the data called exceeds the size of a single cache line, the above process is repeated for the next cache line, with $\tilde{x}_{\text{new}} = x_{\text{old}} + L_S$ and $s_{\text{new}} = s_{\text{old}} - L_S + \tilde{x}_{\text{old}} - x_{\text{old}}$.

For our purposes, we are only interested in cache effects caused by the matrix A, and its input and output vectors x and y. We therefore only apply cache simulations to calls to memory corresponding to A, x, and y. Further advantages of using the cache simulator is that it enables us to obtain theoretical performance gains for caches with arbitrary parameters, that is, we can simulate a variety of caches. Our simulator is limited to handling one level of cache only. It is publicly available.[1]

7. Experimental Results. Here, we present experimental results of our reordering scheme. We are interested in obtaining wall-clock timings for various p and ϵ to measure the effectiveness of our method in practice. We also give the theoretical number of cache misses by using our cache simulator, to check whether the model indeed approaches realistic cache dynamics.

Our experiments are set up as follows. We assume as input a matrix stored in Matrix Market format. This is read into an adapted version of the recently released Mondriaan 2.0 software package [31], which keeps track of the permutation matrices P and Q while partitioning. The permuted matrix PAQ is then written in triplet format to a binary file, and can be read in by specialised CRS, ICRS, and ZZ-ICRS benchmark or cache simulation programs. To compare our results with established methods, we also run experiments using OSKI [32] on the original matrix as well as on reordered versions thereof. The OSKI benchmarking program forces OSKI always to fully tune the input matrix before multiplication (by means of using the ALWAYS_TUNE_AGGRESSIVELY flag). All benchmarking programs perform 1000 matrix–vector multiplications, of which the average execution time (in milliseconds) is reported.

The benchmark applications to obtain the wall-clock timings are performed on two architectures. The first architecture is a quadcore 2.4 GHz Intel Core 2 (Q6600) machine with 8 GB of main memory. Each single core has a 32 kB 8-way L1 cache, and each pair of cores shares a 4 MB 16-way L2 cache. The second architecture is the Dutch national supercomputer Huygens at SARA in Amsterdam. This machine consists of 1664 dual-core 4.7 GHz IBM Power6+ processors divided over 104 nodes. Each core possesses its own 64 kB 8-way associative L1 data cache (besides a 64 kB L1 instruction cache) and a 4 MB L2 cache which is semi-shared (among two cores). The two cores of each processor share a 32 MB L3 cache. Each node of 16 processors has 128 GB main memory.

Several matrices have been pulled from the Florida Matrix Collection [6] and were used in our experiments. We also include some test matrices used in [31]. Table 7.1 shows all matrices and their properties. Generally, matrices can be divided into two classes: those that already possess a structure beneficial in terms of cache reuse, and those that do not. If we expect a matrix to fall in the first category, we call it *structured*, otherwise we call it *unstructured*. Examples of the first category include matrices resulting from finite element methods on a *regular* grid, resulting in d-diagonal matrices. The matrix `s3dtk3m2` is such a matrix, possessing relatively dense block matrices along three diagonals.

Another structured matrix is `memplus`, obtained from memory circuit simulation; see Figure 7.1 (left). For this matrix, the vector x is consecutively accessed in up to four different areas, much like a four-diagonal matrix. The only exceptions occur on the first few rows, and on those rows where a straight line of nonzeros expands into a triangle-like area of nonzeros. Due to this structure, we expect our method not to yield great improvements. On the other hand, we have unstructured matrices such as `rhpentium`, which originates in circuit simulation of a part of the Intel Pentium processor; see Figure 7.1 (right). Since the matrix is so unstructured, and the number of nonzeros per row (nz/row) is rather small, column partitioning is expected to give only few cut nets, and thus we expect large performance gains. Note that the matrix may seem very dense in Figure 7.1 (right), created by using the `spy` plotting command from Matlab^{TM}, but in fact

[1]The cache simulator can be obtained from `http://www.math.uu.nl/people/yzelman/software/`

Name	rows	columns	nonzeros	$\frac{\text{nz}}{\text{row}}$	remarks
memplus	17758	17758	126150	7.1	struct. symmetric, structured
rhpentium	25187	25187	258265	10.3	unstructured
s3dtk3m2	90449	90449	1921955	21.2	symmetric, structured
rand10000	10000	10000	49987	5.0	random pattern
fidap037	3565	3565	67591	19.0	struct. symmetric, structured
lhr34	35152	35152	764014	21.7	structured
rand50000	50000	50000	1249641	25.0	random pattern
nug30	52260	379350	1567800	30.0	structured
tbdlinux	112757	21067	2157675	19.1	unstructured
bmw7st1	141347	141347	3740507	26.5	unstructured
stanford	281903	281903	2312497	8.2	unstructured, link matrix
stanford-berkeley	683446	683446	7583376	11.1	unstructured, link matrix
wikipedia-20051105	1634989	1634989	19753078	12.1	unstructured, link matrix
cage14	1505785	1505785	27130349	18.0	struct. symmetric, structured

TABLE 7.1
Matrices used in our experiments, sorted by category first and the number of nonzeros second. The first category is that of the matrices we discuss in-depth in Section 7. The second shows all matrices of the smaller category, where input and output vectors typically fit into L2 cache. Finally, the third category shows matrices of much larger dimensions, causing more intensive use of the L2 cache.

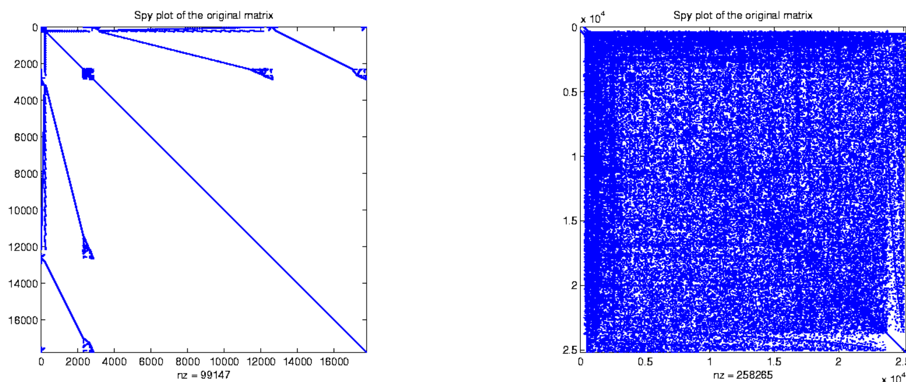

FIG. 7.1. *Plot of the original memplus (left) and rhpentium (right) matrices. Memplus has a favourable structure, whereas rhpentium looks unstructured.*

nz/row ≈ 10.3. The seemingly high density is an artefact due to the relative size of the markers representing the nonzeros. Still, the plot shows very well that nonzeros are spread throughout the matrix. If nz/row is large and the columns are difficult to partition, we may expect many cut nets resulting in badly reordered matrices. Based on this, we can expect the unstructured matrices `rhpentium` and `rand10000` to perform well after reordering; both have relatively small nz/row. A large nz/row does not necessarily mean partitioning will yield catastrophic results: it is possible a matrix can be partitioned quite well even though nz/row is relatively large, and our method may still work well.

Due to the large number of columns of our test matrices, taking the number of partitions to infinity ($p = n$) may take a very long time. We therefore limit the number of partitions to an initial maximum of 400, although some experiments may have used even less partitions. One should note that for a given number of partitions and a given matrix structure, the gain of reordering will depend mainly on the cache size. Hence we start off with discussing the results of cache simulation with different cache size for a fairly structured matrix (`memplus`) as well as an unstructured matrix (`rhpentium`).

As mentioned earlier, we do not expect to gain much by reordering an already favourably

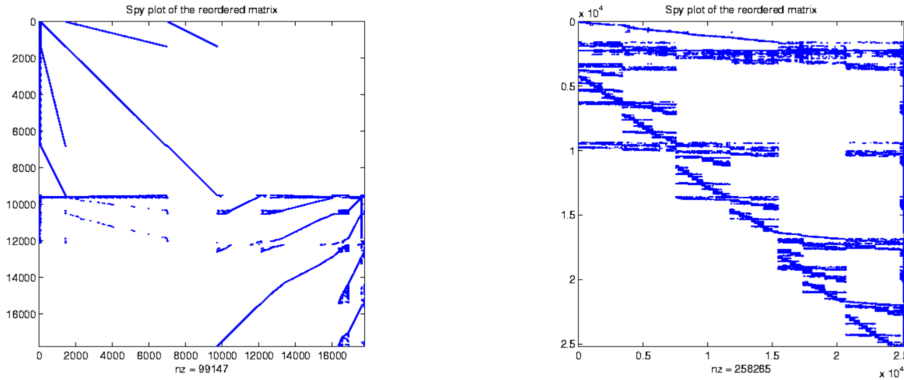

FIG. 7.2. *Plots of the memplus matrix after one bipartitioning (left) and the rhpentium matrix after* 100 *bipartitionings (right).*

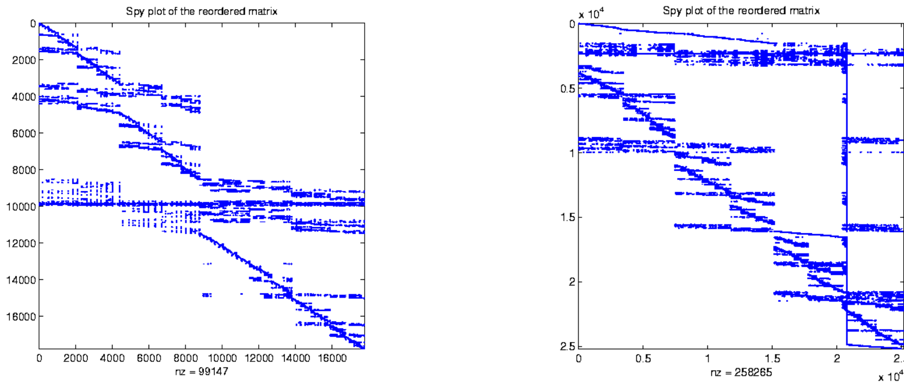

FIG. 7.3. *Plots of the memplus matrix after* 100 *bipartitionings (left) and the rhpentium matrix after* 400 *bipartitionings (right).*

structured matrix. After a single bipartitioning, however, we do see a structural improvement on the matrix `memplus`. In Figure 7.2 (left) the top half of the permuted matrix shows straight lines exactly corresponding to the beneficial original structure. The bottom half contains the less favourably structured parts, compressed together. After 100 bipartitionings (Figure 7.3, left), things have become far less favourably structured as our method tries to compress the nonzeros in smaller and smaller areas. Figure 7.4 reflects this; at $p = 2$ gains of about 2 percent are recorded, while at $p = 100$ losses of the order of 10 percent can be seen.

Examining the unstructured matrix `rhpentium`, Figures 7.2 (right) and 7.3 (right), we see a completely different behaviour. At $p = 100$, a good structure has already surfaced, which is improved at $p = 400$ as the thickness of the rows corresponding to $\mathcal{N}^c$ have been reduced.

Figure 7.5 shows large gains of up to 35 percent for all three data structures. Interesting to note here is that for $p = 400$, the gain curve improves for lower cache sizes when comparing to the same curve for $p = 100$ regardless of which data structure is used. Refining further beyond $p = 100$ is not worth the effort if the actual cache used already was large enough, which would have been the case if the cache used was the L1 cache of an Intel Core 2 processor. In the figure, the largest gain becomes about 37 percent. On the other hand, partitioning to infinity makes the ordering truly cache-oblivious since it works well irrespective of the actual cache size. This also hints at good performance on multi-level cache architectures: the early partitionings optimise for the larger higher-level caches, while subsequent refinements increase performance on the smaller lower-level caches.

Figure 7.5 also shows that the gain for reordering tends to decrease as the cache size increases;

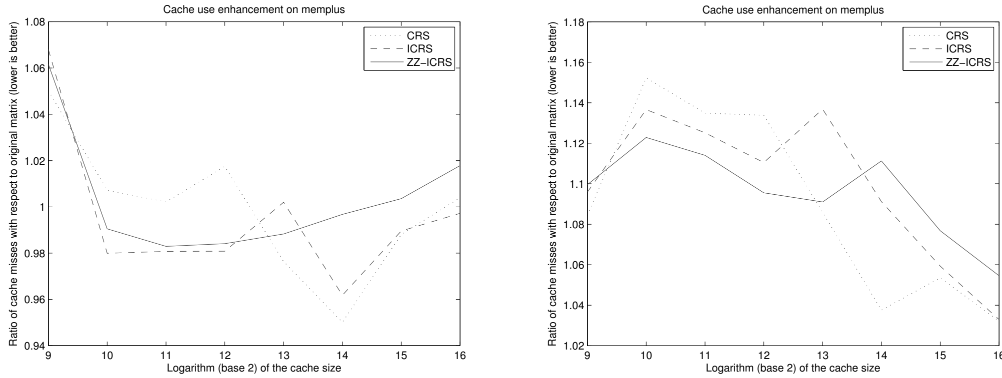

Fig. 7.4. *Simulated cache effect of our reordering method for the memplus matrix, plotted against different cache sizes. Given is the ratio between the number of cache misses for the reordered matrix and the number for the original matrix, for three data structures, namely CRS, ICRS, and ZZ-ICRS. On the left we have simulations with $p = 2$, while on the right we have $p = 100$. The line size is $L_S = 64$ bytes, so that $w = 8$. We assume an 8-way set-associative cache. Note that a cache size $S = 2^{15}$ bytes corresponds to an Intel Core 2 L1 cache.*

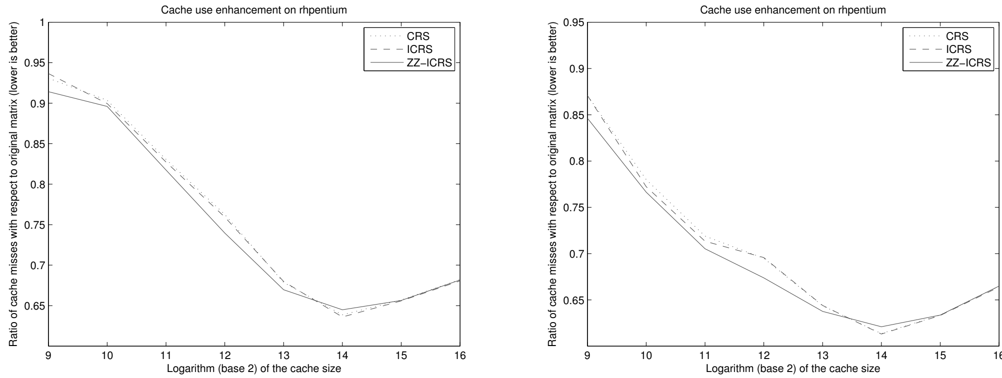

Fig. 7.5. *Simulated cache effect of our reordering method for the rhpentium matrix, plotted against different cache sizes. On the left we have simulations with $p = 100$, while on the right we have $p = 400$. We again are simulating an 8-way set-associative cache with $L_S = 64$.*

this indicates that the more data fit into cache, the less reordering can improve upon cache misses. Eventually one would expect that as $S \to \infty$, the ratio shown tends to one. The reverse also holds; as the cache size tends to a minimum, the ratio tends to one as we cannot improve much when most misses are inevitable.

Table 7.2 presents the same simulations as Figures 7.4 and 7.5, but now with a fixed cache size, a varying number of partitions, and a complete test set of matrices. Table 7.2 shows that for large matrices and larger p, the zig-zag variant of CRS is superior, although we cannot say this for all test instances.

Comparing Table 7.1 and 7.2, it can be seen that matrices with already favourable structure indeed do not yield improved performance upon reordering in our single-level cache simulation. Losses in miss ratio are at most 8 percent, namely for the structured matrix `memplus`. The gain is largest at 37 percent for `rhpentium`, followed by 30 and 25 percent gain for `rand10000` and `stanford` respectively. We also note that although reordering with large p is never found to be beneficial for structured matrices, reordering with small p can still yield modest improvements; see `memplus` and more notably `nug30` which gains 11 percent with $p = 2$.

The miss ratios in Figure 7.4, 7.5 and Table 7.2 are calculated by dividing the number of cache misses for the reordered matrix by the number of cache misses for the original matrix. We expect that these gains correlate to the actual running time of a single multiplication, but not

Name	$p = 2$	$p = 100$	$p = 400$
memplus	0.99 (C)	1.05 (C)	1.08 (Z)
rhpentium	0.96 (Z)	0.66 (Z)	0.63 (I)
s3dkt3m2	1.00 (I)	1.00 (C)	1.00 (C)
rand10000	0.91 (C)	0.72 (C)	0.70 (I)
fidap037	0.98 (C)	1.00 (C)	1.01 (C)
lhr34	1.00 (C)	1.01 (Z)	1.02 (C)
rand50000	1.00 (I)	0.98 (I)	0.98 (I)
nug30	0.89 (Z)	1.05 (I)	1.06 (C)
tbdlinux	0.97 (Z)	0.90 (Z)	0.90 (Z)
bmw7st1	1.00 (I)	0.99 (I)	0.99 (I)

Name	$p = 2$	$p = 10$	$p = 20$
stanford	0.98 (Z)	0.86 (Z)	0.75 (Z)
stanford_berkeley	1.00 (Z)	1.00 (Z)	0.98 (Z)
wikipedia-20051105	0.98 (C)	0.94 (I)	0.92 (Z)
cage14	1.02 (I)	1.09 (Z)	1.10 (I)

Table 7.2

Simulated cache effect of our reordering method for the complete matrix test set and for a varying number of partitions p. Given is the ratio between the number of cache misses for the reordered matrix and the number for the original matrix, for the best of three data structures, namely CRS, ICRS, and ZZ-ICRS. The best data structure is shown in parenthesis. In the partitioning, the load imbalance is chosen as $\epsilon = 0.1$. The results are obtained by simulating an Intel Core 2 L1 cache; that is, $S = 2^{15}, L_S = 64, k = 8$.

in a directly proportional manner: incurring 37 percent less cache misses will not mean running time improves with 37 percent. Rather, we can say the CPU has shorter data access times in 37 percent of the data accesses. To see what the insights obtained by simulation mean in practice, we need experiments on actual machines.

Figure 7.6 shows the gain in wall-clock timings of benchmark experiments for the smaller test matrices. Here, the MV multiplication time for the reordered matrix is compared to the time for the original matrix, which is normalised at the value 1. The matrices used in the benchmarks of Figure 7.6 are relatively small: an Intel Core 2 L2 cache has size $S = 2^{22}$ bytes, while a double requires 8 bytes of storage. Hence a vector of size $2^{19} = 524288$ can fit entirely in L2 cache. As seen in Table 7.1, these test matrices have row and column dimensions much smaller than this size, causing our reordering method to alter mostly the L1 (and not L2) cache behaviour; hence cache effect enhancements translate to relatively small amounts of real-time gain, especially since the partitioning has been stopped at $p = 400$, with p still far from n. Carrying the partitioning much further would probably show gains from the L1 cache.

Figure 7.6 shows that for small matrices our reordering method sometimes achieves modest gains; it almost never leads to losses. The two small matrices that obtained large gains in simulations, `rhpentium` and `rand10000`, also show significant gains here, of about 20 percent and 15 percent, respectively. These cases indeed display larger gains with increasing p, indicating that partitioning with even higher p is desirable. In six out of ten cases, the cache-aware OSKI [32] method is superior, as one would expect since this method uses blocking and can exploit the L1 cache. In seven out of ten cases, combining our reordering with OSKI gives the best result.

As we expect from our previous analysis, reordering performs poorly on the already well-structured matrix `memplus`: with $\epsilon = 0.1$ it breaks even, and with $\epsilon = 0.3$ it shows losses of up to 10 percent. Note that OSKI performs even worse on this problem. The gain in execution time in the case of `fidap037` is surprising, since cache simulation did not show any effect at all, indicating that reordering may affect behaviour not included in our cache simulation model.

Apparently, our reordering method results sometimes in a matrix that OSKI is particularly well-suited to handle, see the `lhr34` matrix. The reordered matrix has visible block structure which may not have been easily discernible in the original matrix. This block structure is then

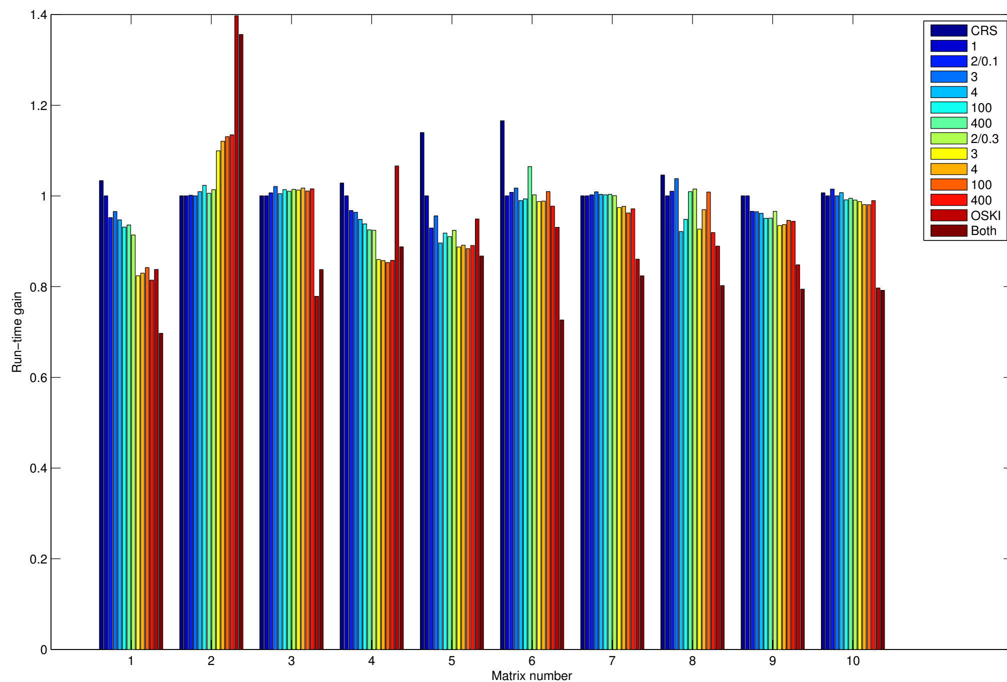

Fig. 7.6. *Wall-clock timings performed on an Intel Core 2 (Q6600) machine. The CRS bar denotes the timings of a basic CRS implementation of MV multiplication applied to the original matrix. The bar $p = 1$ corresponds to the best timing of CRS, ICRS, and ZZ-ICRS also applied to the original. The bars $p = 2/0.1, 3, 4, 100, 400$ show the results after reordering using p partitions, with ϵ set to 0.1, and similarly for the p from $2/0.3$ with $\epsilon = 0.3$. The best timing of the CRS, ICRS, and ZZ-ICRS schemes is shown. The 'OSKI'-bar shows the performance of OSKI [32] when aggressively tuning the original matrix. The 'Both'-bar contains the timing obtained by selecting the reordered matrix with the best results and feeding that matrix to OSKI. Note that OSKI internally uses the CRS scheme. The test matrices are: 1. rhpentium; 2. memplus; 3. s3dkt3m2; 4. rand10000; 5. fidap037; 6. lhr34; 7. rand50000; 8. nug30; 9. tbdlinux; 10. bmw7st1.*

exploited by OSKI.

Figure 7.7 shows results for much larger matrices. We feature several link matrices, as well as the `cage14` matrix. (A *link* matrix A is a binary matrix with entries a_{ij} nonzero iff there is a link from web page i to j.) We stop partitioning at $p = 20$, because partitioning is very time-consuming for these large matrices. Since the input and output vector together do not fit in the L2 cache, and for the largest three matrices even a single vector does not fit, the effects of reordering are expected to become much clearer here. Judging by the results, this is indeed the case; for $p \geq 2$ reordering is already faster than OSKI for two of the four matrices, and increasing p gives even better results, with a gain of over 50 percent for the `stanford` matrix. OSKI outperforms our reordering methods on the smaller matrices, indicating well-optimised behaviour for lower-level caches. Reordering, however, outperforms OSKI on larger matrices, because it uses the higher-level caches better. Therefore, both methods nicely complement each other.

The `stanford_berkeley` matrix breaks about even in run-time, which is very different from the `stanford` matrix which gains much. We do not know the reason for this, and can only speculate. The `stanford_berkeley` matrix consists of two web subdomains, those of Stanford University and of the University of California at Berkeley, which may pose a bigger challenge for finding exploitable structure than the `stanford` matrix which represents a single subdomain. Another possible cause may be that the matrix dimensions ($m = n = 281903$) together are close to the cache size (524288 doubles), making this a border case. If the data from the original matrix

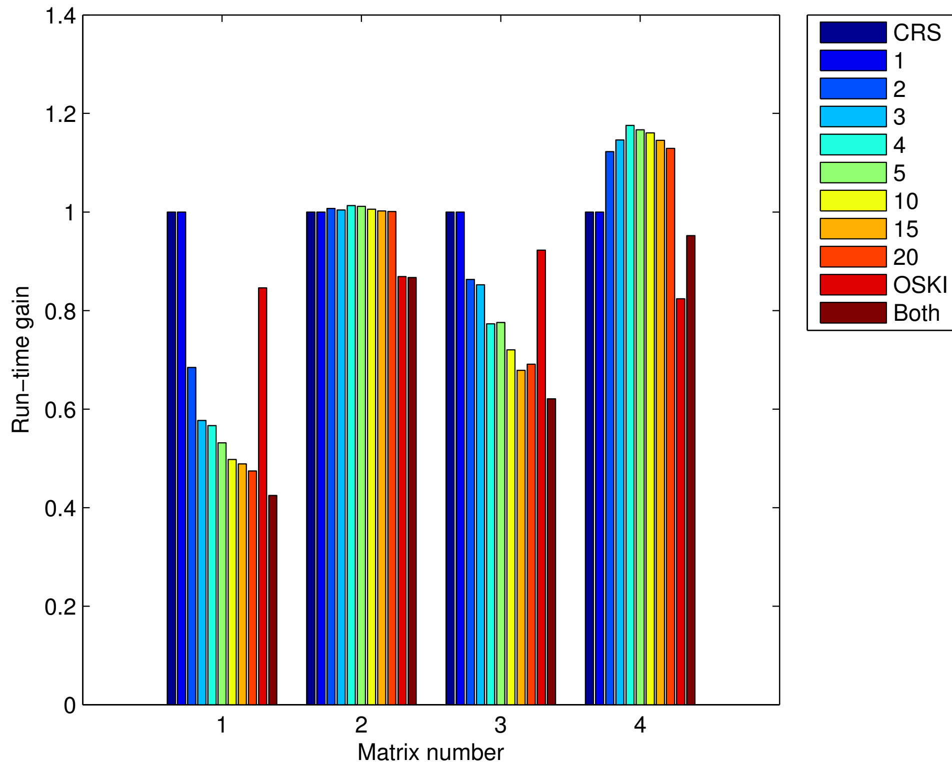

FIG. 7.7. *Wall-clock timings regarding very large matrices; that is, matrices for which the input and output vector do not fit into cache. Interpretation is the same as for Figure 7.6, with $\epsilon = 0.1$ when our matrix reordering method is applied. The test matrices are: 1. stanford; 2. stanford_berkeley; 3. wikipedia-20051105; 4. cage14.*

and vectors often do not fit in the cache, but for the reordered matrix they do, the gains can be particularly large.

The case of `cage14` where losses of more than 10 percent are recorded seems to be a hard case for improvement. One can observe a structure in all the `cage` matrices, which comes from the chosen numbering of states in the underlying Markov model used to study DNA electrophoresis, see [30]. For `cage14`, the original state space had 6^{13} states, which was reduced to 1505785 states (the value of m and n) by exploiting various symmetries. Note that the losses reach their peak at $p = 4$, and that for $p \geq 4$ the losses decrease monotonically with p, indicating that the finer-grain structure of the matrix might eventually be exploited by reordering to improve the cache behaviour.

Since our method is cache-oblivious, it should perform similarly on other architectures. To check this, we also performed experiments on the Dutch national supercomputer Huygens at SARA. We expect similar results as for the Intel Core 2 machine, but perhaps a more increased performance due to the presence of an L3 cache on larger problems. Figure 7.8 shows wall-clock time results for the larger matrices processed on Huygens. We indeed see similar performance on the `stanford` and `wikipedia` matrices, but note that already for $p = 2$ we have a 30 percent gain for the `wikipedia` matrix. This may be caused by the L3 cache which can store 4194304 doubles and hence can just fit the input and output vectors. The losses for the `cage14` matrix are now less severe.

8. Conclusions and future work. We have introduced a matrix reordering method which permutes the matrix rows and columns to improve the cache efficiency of a sparse matrix–vector

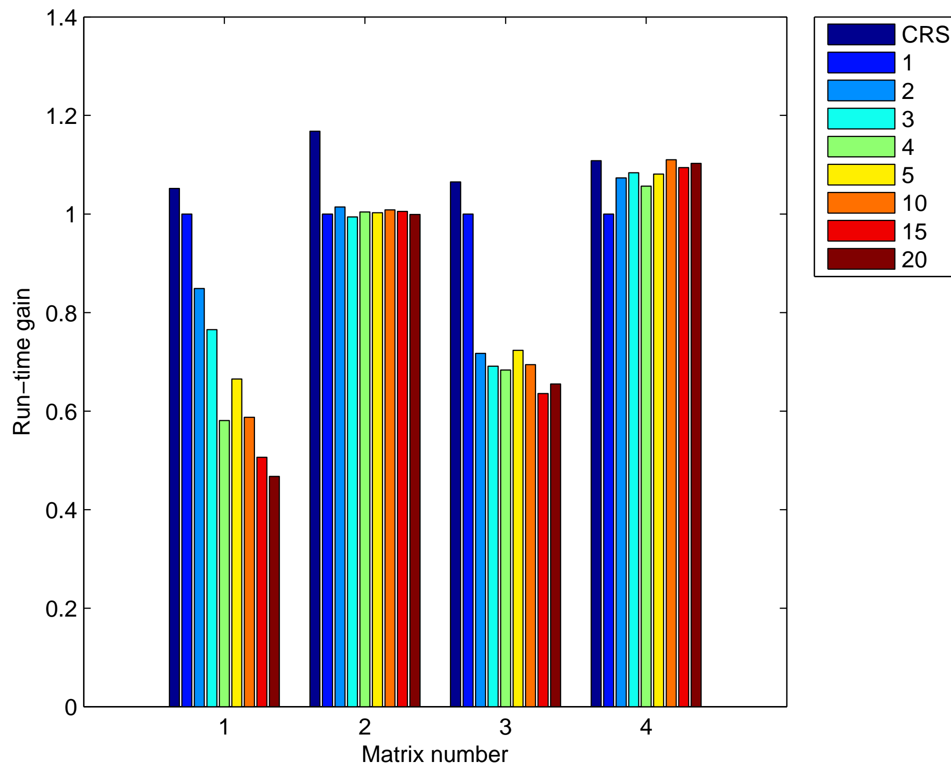

Fig. 7.8. *Experiments similar to Figure 7.7, but run on the Dutch national supercomputer Huygens. No results with OSKI are available. The test matrices are the same: 1. stanford; 2. stanford_berkeley; 3. wikipedia-20051105; 4. cage14*

multiplication algorithm in a cache-oblivious manner. This method is based on partitioning methods for load balancing from the area of parallel matrix computations. By use of both a cache simulator as well as wall-clock timings on two different computer architectures, we have shown experimentally that this method yields considerable savings in computation time for certain matrices during sparse matrix–vector multiplication. For matrices that already have a cache-friendly structure, the gains are modest or even a small loss is observed. This should not deter us from using the reordering method on an unknown matrix, since the potential gains are large, and the possible losses small. The time needed to determine the reordering permutations can be amortised if the multiplication is carried out repeatedly, as happens in iterative linear system solvers and eigensolvers.

Although cache-oblivious matrix reordering may not seem to be as effective on small matrices as cache-aware software such as OSKI [32], this is partly a consequence of not carrying through the partitioning till the end. Further research on improving the partitioning speed should bringer the number of partitions p closer to the number of matrix columns n. For larger matrices, we have observed cases where the time of sparse matrix–vector multiplication was more than halved, e.g. for the large link matrix `stanford`. Cache-aware methods can still be used after cache-oblivious reordering, for fine-tuning to achieve the ultimate in cache use. We have seen that such a combined method works well in practice.

The central ideas of our matrix reordering are:

- using a zig-zag variant of the CRS data structure, thus avoiding unnecessary cache misses at the start and end of rows;
- placing cut rows in the middle during the partitioning process, leading to a gradual tran-

sition between a cache filled with data from one column set $\mathcal{V}_0$ to another one, $\mathcal{V}_1$;
- hypergraph partitioning to reduce the number of cut rows;
- using the $\lambda - 1$ metric in the hypergraph partitioning, to prevent parts of cut rows from being cut further.

For obtaining the matrix reorderings, we have adapted the hypergraph-based sparse matrix partitioner Mondriaan [31], version 2.0, in 1D (column direction) mode; that is, not in its full 2D generality. The adaptations we made for reordering will be incorporated in a future version of Mondriaan. The reordering method does not depend on Mondriaan, however. Instead, one can use other hypergraph partitioners, such as PaToH [21], hMETIS [19], Zoltan [8], Monet [17], and Par*k*way [28].

Mondriaan is, as of yet, not designed with taking the number of partitions to infinity in mind. It is expected that additional speedups can be obtained by removing time-consuming optimisations that only make sense for small p. Also, our results show that the choice of imbalance parameter ϵ is less critical than in the case of parallel computations (where it should reflect the ratio between the computation rate and the communication rate of the hardware). It will be interesting to study better strategies for choosing ϵ in a sequence of matrix splits.

Another way to improve this method is to use 2D partitioning instead of 1D only. This higher-dimensional partitioning can be done by either modeling the matrix by using the fine-grain method [5], or by allowing a 1D method to bipartition recursively in either the column or row direction, whichever yields the best results; the latter is Mondriaan's default mode of operation. Depending on using fine-grain or a row or column partitioning, we end up with six possible sets $\mathcal{R}^{-,c,+}$, $\mathcal{C}^{-,c,+}$ after every bipartition, where $\mathcal{R}, \mathcal{C}$ denote the row and column sets, respectively. It warrants further investigation how a reordering based on these sets affects sparse matrix-vector multiplication when using plain CRS and to see whether other data structures are more appropriate.

Acknowledgements. We thank the Dutch supercomputing centre SARA in Amsterdam and the Netherlands National Computing Facilities foundation NCF for providing access to the Huygens supercomputer and for help in our experiments. We thank the BSIK/BRICKS MSV1-2 program for financial support. Finally, we thank Sarai Bisseling for creating Figure 5.3.

REFERENCES

[1] Cevdet Aykanat, Ali Pinar, and Ümit V. Çatalyürek, *Permuting sparse rectangular matrices into block-diagonal form*, SIAM Journal on Scientific Computing, 25 (2004), pp. 1860–1879.

[2] Zhaojun Bai, James Demmel, Jack Dongarra, Axel Ruhe, and Henk van der Vorst, eds., *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*, SIAM, Philadelphia, PA, 2000.

[3] Michael A. Bender, Gerth Stølting Brodal, Rolf Fagerberg, Riko Jacob, and Elias Vicari, *Optimal sparse matrix dense vector multiplication in the I/O-model*, in SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures, New York, NY, USA, 2007, ACM Press, pp. 61–70.

[4] Sergey Brin and Lawrence Page, *The anatomy of a large-scale hypertextual web search engine*, in Computer Networks and ISDN Systems, vol. 30, 1998, pp. 107–117.

[5] Ümit V. Çatalyürek and Cevdet Aykanat, *A fine-grain hypergraph model for 2D decomposition of sparse matrices*, in Proceedings Eighth International Workshop on Solving Irregularly Structured Problems in Parallel (Irregular 2001), IEEE Press, Los Alamitos, CA, 2001, p. 118.

[6] Timothy A. Davis, *University of Florida sparse matrix collection*, online collection, http://www.cise.ufl.edu/research/sparse/matrices, Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL, 1994-2004.

[7] John M. Dennis and Elizabeth R. Jessup, *Applying automated memory analysis to improve iterative algorithms*, tech. report, University of Colorado at Boulder, Department of Computer Science, 2006.

[8] K. D. Devine, E. G. Boman, R.T. Heaphy, R. H. Bisseling, and U. V. Catalyurek, *Parallel hypergraph partitioning for scientific computing*, in Proceedings IEEE International Parallel and Distributed Processing Symposium 2006, IEEE Press, 2006.

[9] I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct Methods for Sparse Matrices*, Monographs on Numerical Analysis, Oxford University Press, Oxford, UK, 1986.

[10] Matteo Frigo and Steven G. Johnson, *FFTW: An adaptive software architecture for the FFT*, in Proceedings IEEE International Conference on Acoustics, Speech, and Signal Processing, vol. 3, IEEE Press, Los Alamitos, CA, 1998, pp. 1381–1384.

[11] Matteo Frigo and Steven G. Johnson, *The design and implementation of FFTW3*, Proceedings of the IEEE, 93 (2005), pp. 216–231. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[12] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran, *Cache-oblivious algorithms*, in FOCS '99: Proceedings of the 40th Annual Symposium on Foundations of Computer Science, Washington, DC, USA, 1999, IEEE Computer Society, p. 285.

[13] Alan George, *Nested dissection of a regular finite element mesh*, SIAM Journal on Numerical Analysis, 10 (1973), pp. 345–363.

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

[15] Laura Grigori, Erik Boman, Simplice Donfack, and Timothy A. Davis, *Hypergraph-based unsymmetric nested dissection ordering for sparse LU factorization*, Tech. Report 6520, Orsay, France, 2008.

[16] Bruce Hendrickson and Edward Rothberg, *Improving the run time and quality of nested dissection ordering*, SIAM Journal on Scientific Computing, 20 (1998), pp. 468–489.

[17] Y. F. Hu, K. C. F. Maguire, and R. J. Blake, *A multilevel unsymmetric matrix ordering algorithm for parallel process simulation*, Computers and Chemical Engineering, 23 (2000), pp. 1631–1647.

[18] Eun-Jin Im and Katherine A. Yelick, *Optimizing sparse matrix–vector multiplication for register reuse*, in Proceedings of the International Conference on Computational Science, May 2001.

[19] George Karypis and Vipin Kumar, *Parallel multilevel k-way partitioning scheme for irregular graphs*, SIAM Review, 41 (1999), pp. 278–300.

[20] Takahiro Katagiri, Kenji Kise, Hiroki Honda, and Toshitsugu Yuba, *ABCLibScript: a directive to support specification of an auto-tuning facility for numerical software*, Parallel Computing, 32 (2006), pp. 92–112.

[21] Ümit V. Çatalyürek and Cevdet Aykanat, *Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication*, IEEE Transactions on Parallel and Distributed Systems, 10 (1999), pp. 673–693.

[22] Jon M. Kleinberg, *Authoritative sources in a hyperlinked environment*, J. ACM, 46 (1999), pp. 604–632.

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

[24] Amy N. Langville and Carl D. Meyer, *Google's PageRank and Beyond: The Science of Search Engine Rankings*, Princeton University Press, July 2006.

[25] Michelle Mills Strout, Larry Carter, Jeanne Ferrante, and Barbara Kreaseck, *Sparse tiling for stationary iterative methods*, The International Journal of High Performance Computing Applications, 18 (2004), pp. 95–113.

[26] Michelle Mills Strout and Paul D. Hovland, *Metrics and models for reordering transformations*, in MSP '04: Proceedings of the 2004 workshop on Memory system performance, New York, NY, USA, 2004, ACM, pp. 23–34.

[27] S. Toledo, *Improving the memory-system performance of sparse-matrix vector multiplication*, IBM Journal of Research and Development, 41 (1997), pp. 711–725.

[28] Aleksandar Trifunovic and William J. Knottenbelt, *A parallel algorithm for multilevel k-way hypergraph partitioning*, in Proceedings Third International Symposium on Parallel and Distributed Computing, Cork, Ireland, July 2004.

[29] Ruud van der Pas, *Memory hierarchy in cache-based systems*, tech. report, Sun Microsystems, Inc., Santa Clara, Nov 2002.

[30] A. van Heukelum, G. T. Barkema, and R. H. Bisseling, *DNA electrophoresis studied with the cage model*, Journal of Computational Physics, 180 (2002), pp. 313–326.

[31] Brendan Vastenhouw and Rob H. Bisseling, *A two-dimensional data distribution method for parallel sparse matrix-vector multiplication*, SIAM Review, 47 (2005), pp. 67–95.

[32] Richard Vuduc, James W Demmel, and Katherine A Yelick, *OSKI: A library of automatically tuned sparse matrix kernels*, Journal of Physics: Conference Series, 16 (2005), pp. 521–530.

[33] R. Clint Whaley and Antoine Petitet, *Minimizing development and maintenance costs in supporting persistently optimized BLAS*, Software: Practice and Experience, 35 (2005), pp. 101–121.

[34] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra, *Automated empirical optimizations of software and the ATLAS project*, Parallel Computing, 27 (2001), pp. 3–35.