SIAM J. SCI. COMPUT.
Vol. 31, No. 4, pp. 3128–3154

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 recursive hypergraph-based sparse matrix partitioning scheme so that the resulting matrix induces cache-friendly behavior 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. The separated block-diagonal (SBD) form is shown to be the appropriate matrix structure for cache enhancement. We have implemented a run-time cache simulation library enabling us to analyze cache behavior for arbitrary matrices and arbitrary cache properties during matrix–vector multiplication within a k-way set-associative idealized cache 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 the case of a large link matrix.

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

AMS subject classifications. 65F10, 65F50, 65Y05, 65Y20

DOI. 10.1137/080733243

1. Introduction.

Many important linear algebra kernels typically take a performance hit on modern cache-based computer architectures [9, 21, 34] 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 nontrivial applications require one to jump through data in main memory, even if data is stored contiguously. A notorious example is sparse matrix–vector (SpMV) multiplication. In these cases, we can try to minimize the number of jumps or otherwise improve cache efficiency. As recent work in the field of the BLAS by Goto and van de Geijn [16] 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 [24, 41], most notably FFTW (for fast Fourier transforms) [12, 13], OSKI (basic sparse BLAS) [39], and ATLAS (dense BLAS) [42]. These libraries may run various benchmarks upon installation to help optimize 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*

*Received by the editors August 20, 2008; accepted for publication (in revised form) March 23, 2009; published electronically July 31, 2009. This work was supported by the BSIK/BRICKS MSV1-2 program.

http://www.siam.org/journals/sisc/31-4/73324.html

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

TABLE 1.1
List of parameters used throughout this paper.

$m \times n$	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

algorithms have been researched to some extent, and for some applications have been shown to obtain asymptotically optimal bounds [3, 14].

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. Related cache-aware and cache-oblivious methods for the SpMV multiplication are discussed in section 4. In section 5, we connect the SpMV multiplication problem to hypergraph partitioning theory, and formulate the basic idea of our method. In section 6, we formalize 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 is still 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 of our run-time cache simulator, which we introduce in section 7. We finish with experimental results on this simulator and on actual hardware in section 8 and general conclusions in section 9.

The contributions of the present work can be summarized as follows: (i) we present a new variant of the CRS data structure, zig-zag CRS, which reduces end-of-row cache misses; (ii) we introduce the separated block-diagonal (SBD) form as a matrix reordering that achieves better cache use; (iii) we present a new cache-oblivious approach to SpMV and demonstrate that it can work well in practice; (iv) we establish a connection between matrix reordering for cache enhancement and hypergraph partitioning, which can be used to detect and exploit beneficial structure in a matrix.

Throughout the paper, we assume a k-way set-associative idealized cache model, with k subcaches, where the case $k \to \infty$ corresponds to the ideal-cache model introduced in [14]. 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 modeled as an $(L/k) \times k$ matrix, where entries $c_{i,j}$ correspond to individual cache lines and each matrix column $c_{*,j}$ to a subcache. A matrix row $c_{i,*}$ is called a *set*. 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 onto Set IDs and that the Line ID is selected 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 $i = r \mod \frac{L}{k}$. 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 $c_{i,j}$. Such behavior can be modeled by using a *stack* of size k; memory contents may be (re)inserted at the top of the

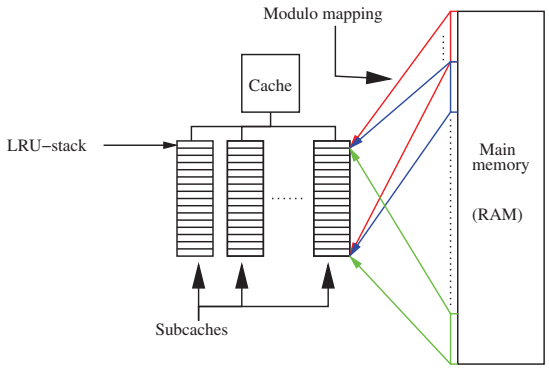

FIG. 1.1. *Schematic view of the k-way set-associative cache model with modulo mapping of main memory to subcache.*

ALGORITHM 1 Matrix–vector multiplication using CRS.

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

1: Allocate y of size m and initialize: $y = \mathbf{0}$
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

stack, while the least recently used data are evicted from the bottom. Each cache set $c_{i,*}$ thus maps to a single LRU stack. See also Figure 1.1 for an illustration of this model. For a further introduction to cache architecture, one may consult [36].

2. Incremental CRS. A standard storage scheme for sparse matrices is the compressed row storage (CRS) format [2]. This format utilizes 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 $\mathrm{nz}(A)$ space in memory, where $\mathrm{nz}(A)$ 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 $2\mathrm{nz}(A) + m$. An algorithm to perform the SpMV 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}, $0 \leq i < m$, and $0 \leq j < n$.

A drawback of standard CRS implementation is the use of the *row_start* array for keeping track which nonzeros belong to which row. Looking at Algorithm 1, a for-loop was started on line 3 along indices k relevant to the row i being processed. Since there

ALGORITHM 2 Matrix–vector multiplication using ICRS.

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

```
1: Allocate y of size m and initialize: y = 0
2: i = row_jump[0], j = diff[0], k = 0, r = 1
3: while k < nz(A) do
4:    y[i] = y[i] + nzs[k]*x[j]
5:    k = k + 1
6:    j = j + diff[k]
7:    if j ≥ 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
```

is no other way of knowing when a row ends, optimized CRS multiplication code will always have to keep track of k. Furthermore, although the array *nzs* is straightforwardly accessed according to k, x is accessed according to $j = col\_ind[k]$; this kind of *index translation* also causes instruction overhead: if we store the address (pointer) `xp` of the currently used element from x, this is updated like `xp=x+col_ind[k]`. Faster would be `xp+=diff[k]`, with `diff[x]=col_ind[k]-col_ind[k-1]` and `diff[0]=col_ind[0]`; this, combined with an alternative for the *row_start* array, leads to the incremental CRS (ICRS) scheme proposed by Koster in his master's thesis [26]. A sparse matrix–vector algorithm utilizing this ICRS scheme is given in Algorithm 2. Note that an increment in row index is signaled 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*. Although the pseudocode of Algorithm 2 is a few lines larger than that of Algorithm 1, we can implement it more efficiently by using suitable pointer arithmetic. This causes the instruction overhead to drop dramatically [26, Figure 2.5].

Changing CRS to handle data incrementally does not change its worst-case 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 $2\mathrm{nz}(A) + m$. If there are some empty rows, however, memory requirements as well as memory accesses are reduced slightly, resulting in further speedup. If there are no empty rows, the array *row_jump* need not be stored since all its entries equal 1.

3. Cache performance for dense and sparse matrix–vector 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*, i.e., consecutively from front to back. Contiguous access is necessary since multiple data words may fit into a single cache line.

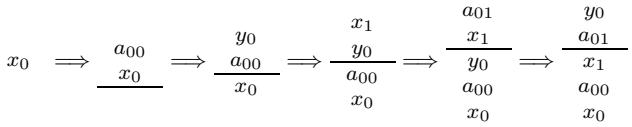

FIG. 3.1. *LRU stack progression during the first steps of dense MV multiplication. Whenever a new data element is pushed onto the stack, it is displayed on top. Older elements thus appear lower in this figure, with the least recently used element at the bottom. Note that if all memory lines below the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

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 idealization $k \to \infty$; 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 *cache blocking* [30] 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.

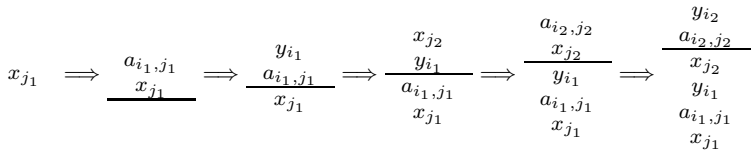

FIG. 3.2. *LRU stack progression during the first steps of sparse MV multiplication using CRS. The indices i_r and j_r are the row and column index of the* r*th nonzero element from A. Note that for most r, $j_r \neq j_{r+1}$, while i_r only differs from i_{r+1} when the nonzeros at a given row are exhausted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

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, by discussing the CRS ordering, we refer to both the CRS and ICRS data structures. We may analyze 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 SpMV 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; this also requires information on the structure of A. Cache-aware blocking thus becomes much harder to apply, amongst others motivating the development of run-time cache-aware auto-tuning kernels such as OSKI [39].

4. Related work on sparse matrix–vector multiplication. In this section, we present an overview of earlier work dealing explicitly with improving cache efficiency in SpMV multiplication. Kowarschik et al. [27] improve the speed of a multigrid method for Poisson's equation on a two-dimensional grid by exploiting knowledge of the cache characteristics. The multigrid example they use is based on a red-black ordered Gauss–Seidel operation with a 5-point finite-difference stencil. This operation can be viewed as multiplying the vector representing the grid points by a certain Laplacian sparse matrix. Among the techniques applied are: loop blocking, i.e., combining computations from different iterations of the main loop; reordering of the computations, obtained by sliding a small window in which computations are carried out across the whole grid, instead of performing the computations row by row. A speedup of a factor of 4.6 is obtained by the latter technique for certain grid sizes. The authors note that cache effects will be even stronger in the three-dimensional case.

Das et al. [7] use reordering techniques developed for reducing fill-in (i.e., new nonzeros) in direct solvers, such as the Cuthill–McKee (CM) method and the reverse CM method, to reduce the bandwidth of the matrix for the purpose of avoiding cache misses in SpMV multiplication. Toledo [34] performs an extensive study of such ordering techniques, and other techniques such as finding small dense 1×2 and 2×2 blocks, and prefetching of data. He found that CM ordering works well for his test matrices (all 3D finite-element matrices) when used in combination with blocking into small dense blocks. He notes that CM is cheap to compute, requiring computing time equivalent to a few SpMV multiplications. Reverse CM ordering is not as good, as it leads to fewer dense blocks.

Pinar and Heath [31] try to create contiguous blocks of nonzeros in the matrix rows by reordering the columns, formulating the problem as an instance of the traveling

salesman problem (TSP). In their formulation, cities represent the matrix columns and distances the inner products between columns (considering the sparsity pattern only, not the numerical values). This inner product metric is the same as the column similarity metric used in many partitioners, including Mondriaan [38] and PaToH [5]. Small dense 1×2 blocks are then used to halve the integer overhead of the SpMV, and also to reduce the number of loads of a cache line: for $w > 1$, the two required adjacent values x_j and x_{j+1} will often be in the same cache line. In their experiments, Pinar and Heath achieve on average 21 percent reduction in SpMV time by using the TSP ordering, and only 5 percent by using the reverse CM ordering. The largest gain observed for TSP is 33 percent.

Vuduc and Moon [40] write the sparse matrix A as the sum of several matrices, each of which is stored in a blocked variant of the CRS format, with its own block size and alignment of the starting nonzero. Here, each block is small and dense, e.g., 1×2 or 1×3. A matrix with block size 1×1 is also included to represent the remaining nonzeros in the standard CRS format. Blocks are created by greedily grouping rows and columns based on a scaled inner product metric.

Nishtala et al. [30] investigate cache blocking for SpMV multiplication by splitting the matrix into several smaller $p \times q$ sparse submatrices (blocks), using the CRS data structure for each separate block. They present an analytic cache-aware model to determine the optimal block size and demonstrate that this scheme works well in practice. The authors achieve speedups for over half of their test matrices with a maximum speedup of 2.93. They note that gains are largest for $m \times n$ matrices with $m \ll n$. Their algorithms and software are included in the OSKI package [39].

White and Sadayappan [43] use the Metis graph partitioner [22] to reorder the matrix for better SpMV performance on machines with a single-level cache. They report no gains by this method, which they attribute to the natural well-structured ordering of their test matrices. Nevertheless, reordering by partitioning did show some speedup compared to a random ordering. The authors propose to unroll loops and change the CRS data structure, e.g., by sorting the rows by length and creating blocks of rows of the same length.

Strout and Hovland [32, 33] use (regular) graph partitioning to achieve a better data ordering in memory. They use hypergraph partitioning to improve inter-iteration (temporal) locality, which can be done in some iterative algorithms where it is not necessary to finish one iteration before partially continuing with the next.

Haase, Liebmann, and Plank [18] use the Hilbert space-filling curve to order the elements of the matrix A, storing the nonzero elements in the order they are encountered along the curve. Each element a_{ij} is stored as a triple (i, j, a_{ij}). They call the resulting data structure *fractal storage*. Experimental results on finite-element matrices show speedups of up to 50 percent in computing rate compared to the common CRS format, for SpMV multiplication with eight simultaneous input vectors. This approach has the advantage that it exploits naturally existing locality in the matrix at all levels of detail simultaneously, without knowing about the characteristics of the cache; the Hilbert space-filling approach thus is cache-oblivious. This approach is two-dimensional, in contrast to our own approach, which is one-dimensional since it is based on storing matrix rows. No attempt is made to enhance locality, in contrast to our own method. It may be possible to combine the Hilbert method with ours, by first running the matrix reordering presented here and then using fractal storage.

Summarizing the related work, we can say that only a few approaches are explicitly cache-aware or cache-oblivious. The approach incorporated in OSKI [30, 39] is cache-aware since it adapts to the cache sizes and other characteristics; this also holds

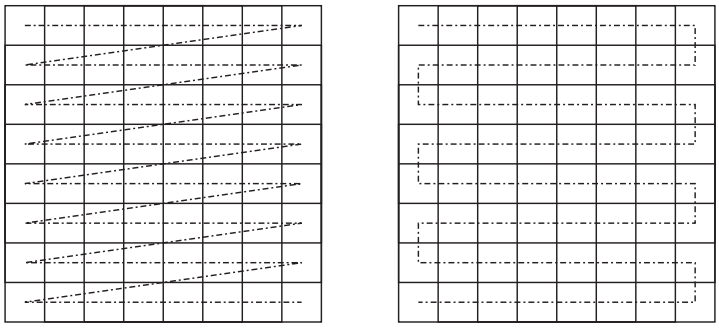

FIG. 5.1. *The CRS (left) and ZZ-CRS (right) orderings. The dash-dotted lines show the order in which matrix nonzeros are stored.*

for the approach of [27]. The space-filling curve approach [18], like ours, can be called cache-oblivious since by its recursive nature it exploits locality at all levels. Most approaches, however, fall outside these two categories. They try to enhance cache use by reordering or splitting the matrix, without knowing the cache size, but also without tackling all levels simultaneously. Often, the approach is based on using small dense blocks [7, 31, 34], sometimes called *register blocking*, and this will improve use of the registers and the L1 cache, but not the L2 and L3 cache. In the next sections, we will present our own approach.

5. 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 nonempty, we thus process in the standard increasing order on even-numbered rows, and in decreasing order on odd-numbered rows. We call the resulting data structure *zig-zag CRS* (ZZ-CRS). Figure 5.1 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 per row, 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 observed earlier. To this end, we quickly review basic hypergraph-based partitioning methods for sparse matrices.

5.1. Hypergraph partitioning. 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 [5], since each matrix row is represented by a net. Figure 5.2 displays a small example sparse matrix and its row-net hypergraph. Other models to represent sparse matrices include the *column-net* [5] and *fine-grain* model [6]. In the column-net model, the roles of the

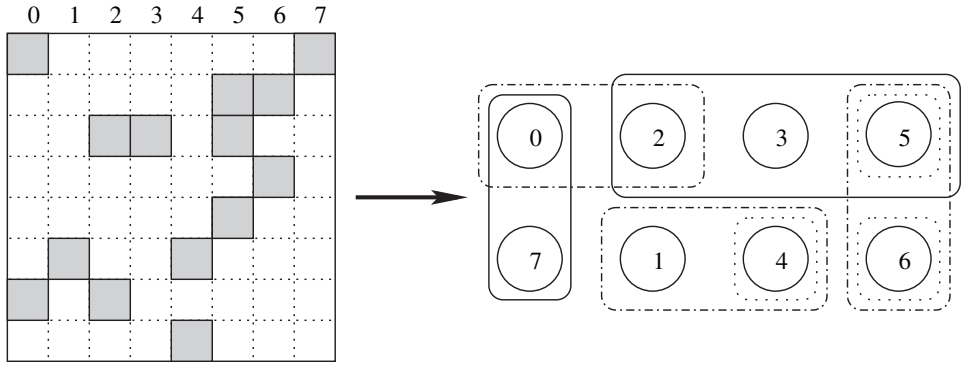

FIG. 5.2. *Example translation from a sparse matrix to a hypergraph using the row-net model. Vertices* 0 *through* 7 *represent the columns. A net is cast over those vertices that share nonzeros in a row. Hence, each net represents a row.*

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 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 net n_i are distributed over several subsets (or parts), resulting in a *cut net*. Let us denote the number of subsets 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:

$$\sum_{i:n_i \in \mathcal{N}} c_i(\lambda_i - 1). \tag{5.1}$$

This cost function is commonly called the $\lambda - 1$ metric, or connectivity-1 metric, and is used extensively in parallel computing; see [5]. It originates in the field of electronic circuit simulation; see [29]. 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 minimize the function (5.1).

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 part 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 minimize

$$\max_{0 \le s < p} \sum_{j:v_j \in \mathcal{V}_s} w_j. \tag{5.2}$$

Consider a recursive algorithm that 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

$$\epsilon = \frac{\max\{|\mathcal{V}_0|, |\mathcal{V}_1|\}}{(|\mathcal{V}_0| + |\mathcal{V}_1|)/2} - 1. \tag{5.3}$$

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 minimized total connectivity cost usually conflict with each other. In an attempt to construct a partitioning that is acceptable in terms of both criteria, we may decide to allow a load imbalance not larger than some predefined parameter ϵ. A software package implementing this approach in sparse matrix partitioning is Mondriaan [38]. 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 8, 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, parts 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 SpMV multiplication. The same goes for components of y: some processors may have to add values to components of y governed by other processors, thus also requiring communication. This happens after the computational work and is called the *fan-in*.

The rationale for the cost function (5.1) is that if we attempt to minimize communication, we try to avoid any cut nets; ideally, we have $\lambda_i = 1$ for all i. 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 that have to communicate with each other is minimized.

5.2. Cache-oblivious approach. Now, consider the following bipartitioning routine working on a row-net hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ representing a sparse matrix A, where vertices represent columns and rows represent nets. We partition $\mathcal{V}$ into parts $\mathcal{V}_0, \mathcal{V}_1$ while taking into account the load imbalance constraint and the cost function. Note that the indices of the columns corresponding to the vertices in each part 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 parts, is denoted by $\mathcal{N}^c$.

Let the SpMV multiplication algorithm visit matrix elements in the following order. First, the rows corresponding to nets in $\mathcal{N}^-$ are processed, followed by those in $\mathcal{N}^c$, and finally those in $\mathcal{N}^+$; see Figure 5.3. The matrix elements in consecutive rows are furthermore processed in zig-zag order. Since such an algorithm, like ZZ-CRS based SpMV multiplication, visits matrix rows in succession, no unnecessary cache misses are incurred on the vector y.

To minimize the cache misses on x, we define

$$p = \frac{n}{wL}, \tag{5.4}$$

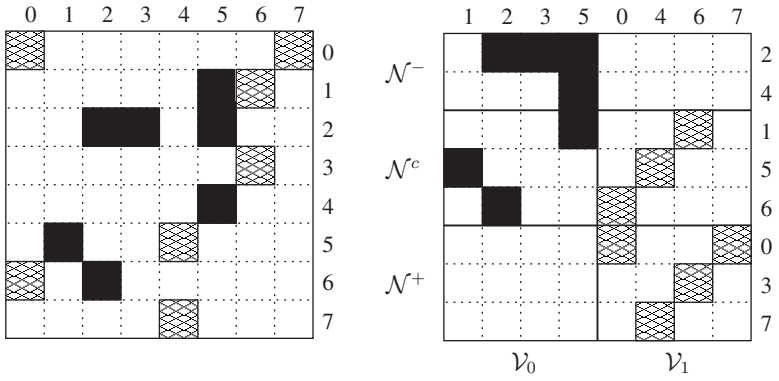

FIG. 5.3. *Original matrix (left) and permuted matrix (right) after one bipartitioning step. Shaded and black squares denote nonzero elements. Black columns denote vertices in $\mathcal{V}_0$ whereas shaded columns denote vertices in $\mathcal{V}_1$.*

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 parts (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 (5.4) 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 sufficiently dense, with at least one out of every w matrix elements 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 SpMV multiplication times L gives an upper bound on the number of cache misses. Thus, minimizing the communication volume in the $\lambda - 1$ metric using hypergraph partitioning is expected to improve cache utilization. This is the main motivation for our approach.

In practice, we are oblivious of the values for w, L and we do not know the appropriate value of p. It does not harm the cache behavior to partition beyond the value of p given in (5.4). 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 6), we create beneficial matrix structure at all values of p. This way of reordering also minimizes cache misses on multilevel cache hierarchies since for lower p we optimize for the larger (e.g., L2) caches while for higher p we optimize for the smaller (L1) caches, without harming the upper-level optimizations.

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 favor of those of $\mathcal{V}_1$. We then proceed with $\mathcal{N}^+$, which deals only 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 [15] for Cholesky factorization [19] or LU factorization; Aykanat et al. [1] use hypergraphs to reorder rectangular matrices for sparse LU or QR factorization. Recent work by Grigori et al. [17] also employs hypergraphs to reorder unsymmetric matrices for sparse LU factorization. For factorization, 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 [20], which tries to create a bordered block-diagonal form based on hypergraph partitioning with the cut-net metric (i.e., (5.1) with $\lambda_i - 1$ replaced by 1 if $\lambda_i \geq 2$, and 0 otherwise).

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.

6. 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 illustrated in Figure 5.3. The permutation of columns can be written in linear algebra form as 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-sided multiplication with another permutation matrix P.

Let us denote the $n \times n$ identity matrix by $I = (e_0|e_1|\dots|e_{n-1})$. Then we have that the right-sided permutation matrix is given by $Q = (e_{q_0}|e_{q_1}|\dots|e_{q_{n-1}})$, for some index permutation $(q_i)_{i\in[0,n-1]}$ of the tuple $(0, 1 \dots, n-1)$. Similarly, $P = (e_{p_0}|e_{p_1}|\dots|e_{p_{m-1}})^T$, 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 the remaining indices 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 SpMV multiplication routine can be represented as follows:

$$y = Ax = P^T A_r Q^T x, \quad \text{so } \tilde{y} = A_r \tilde{x} \quad \text{with } \tilde{y} = Py, \quad \tilde{x} = Q^T x. \tag{6.1}$$

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 SpMV multiplication:

- We can express $y = Ax + \beta z$ as follows: $y = P^T(A_r Q^T x + \beta P z)$, which can be written as a standard SpMV multiplication $\tilde{y} = A_r \tilde{x} + \beta \tilde{z}$ with $\tilde{y} = Py$, $\tilde{x} = Q^T x$, and $\tilde{z} = Pz$.

- Also, $y = A^T x$ can be expressed as a permutation: $y = QA_r^T Px$, so that we have $\tilde{y} = A_r^T \tilde{x}$ with $\tilde{y} = Q^T y$ and $\tilde{x} = Px$.
- Finally, $y = AA^T x$ is also possible: $y = P^T A_r A_r^T Px$, so that $\tilde{y} = A_r A_r^T \tilde{x}$ with $\tilde{y} = Py$ and $\tilde{x} = Px$.

This shows that one can apply our cache-oblivious matrix reordering scheme by permuting only 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 [39], 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 $y = A^s x$, $s \in \mathbb{N}$, and $s > 1$. 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 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 [25] (see also the book [28, p. 117]), the power method is applied to the matrices AA^T and $A^T A$ with A a link matrix. After reordering of A, the matrix $(AA^T)^s$ becomes $(PAQQ^T A^T P^T)^s = P(AA^T)^s P^T$, and similarly for $A^T A$. 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 by (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 nets in $\mathcal{N}^-$ can then map to $[0, |\mathcal{N}^-| - 1]$, those in $\mathcal{N}^c$ to $[|\mathcal{N}^-|, m - |\mathcal{N}^+| - 1]$, and those in $\mathcal{N}^+$ to $[m - |\mathcal{N}^+|, m-1]$; see also Figure 5.3. This procedure is repeated after each following bipartitioning; see Figure 6.1. There, each horizontal straight line gives new row boundaries for the rows, with Q_i the interval defined by those boundaries. 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 6.2 shows the overall structure that is obtained by recursively bipartitioning a sparse matrix and placing the cut rows in the middle. In analogy with the *bordered block-diagonal* (BBD) form [11], we call the matrix structure obtained after a sequence of recursive bipartitionings the *separated block-diagonal* (SBD) form. At

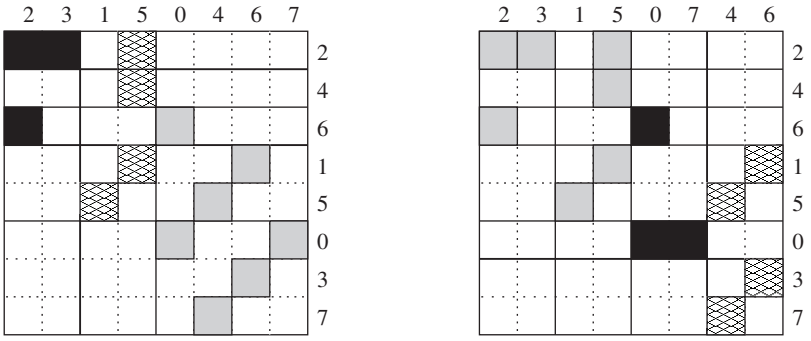

FIG. 6.1. *The reorderings after two (left) and three (right) bipartitionings of the original matrix in Figure* 5.3. *Grey squares denote nonzeros not considered in the current bipartitioning step.*

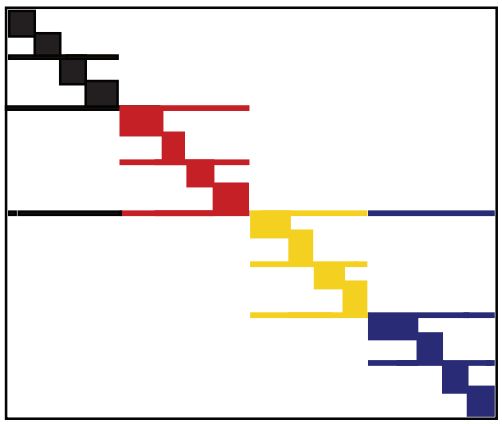

FIG. 6.2. *Separated block-diagonal (SBD) structure of a sparse matrix obtained by recursively partitioning in the column direction and moving the cut rows to the middle. The recursion has been stopped after creating* 16 *diagonal blocks. The four colors indicate the block structure at* $p = 4$.

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 advantage.)

7. 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 analyze 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 analyze *theoretical* cache performance on any sparse input matrix and correlate these results to actual wall-clock timings, as will be done in section 8.

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 reinserted at the top of the stack. If it is not present, we have a miss and x is inserted at the top. If this would cause the stack size to overflow (i.e., become larger than k items), the address at the bottom of the stack is evicted. This process is repeated in case a single cache line does not suffice, with $\tilde{x}_{\text{new}} = x_{\text{old}} + L_S$ and $s_{\text{new}} = s_{\text{old}} - L_S + \tilde{x}_{\text{old}} - x_{\text{old}}$. These calculations are done on-the-fly, so that no trace data needs to be stored.

For our purposes, we are interested only 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. Our cache simulator 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]

8. 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. Finally, we express the reordering time in terms of SpMVs (on the original matrix).

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 [38], 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 specialized CRS, ICRS, and ZZ-ICRS benchmark or cache simulation programs. To compare our results with established methods, we also run experiments using OSKI (optimized sparse kernel interface) [39] on the original matrix as well as on reordered versions thereof. We thus have obtained results of the reordered matrix using two different SpMV routines, both based on CRS or one of its variants. Our OSKI benchmarking program forces OSKI always to attempt a full tuning of the input matrix before multiplication, assuming any tuning can be amortized by sufficiently many SpMVs; however, this does not guarantee that OSKI actually tunes every input matrix. It is possible to ask OSKI whether or not a matrix has been tuned; we will do this to gain additional insights regarding the OSKI timings. Furthermore, a user who has intimate knowledge of the input matrix can provide hints to OSKI as to which structural properties may be exploited. We have not given OSKI any such hints; if we had done so, better optimizations might have been found. All benchmarking programs perform 1000 matrix–vector multiplications, of which the normalized average execution time 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)

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

TABLE 8.1

Matrices used in our experiments, sorted by category first and number of nonzeros second. The first category is that of the matrices we discuss in-depth in section 8. The second category contains the remaining matrices of smaller size, where input and output vectors typically fit into L2 cache. Finally, the third category contains larger matrices, causing more intensive use of the L2 cache. The sixth column indicates whether the matrix is considered structured (S) or unstructured (U).

Name	Rows	Columns	Nonzeros	$\frac{\text{nz}}{\text{row}}$		Remarks
memplus	17758	17758	126150	7.1	S	struct. symm., chip design
rhpentium	25187	25187	258265	10.3	U	chip design
s3dkt3m2	90449	90449	1921955	21.2	S	symm., FEM
rand10000	10000	10000	49987	5.0	U	random pattern
fidap037	3565	3565	67591	19.0	S	struct. symm., FEM
lhr34	35152	35152	764014	21.7	S	chemical process
rand50000	50000	50000	1249641	25.0	U	random pattern
nug30	52260	379350	1567800	30.0	S	quadratic assignment
tbdlinux	112757	21067	2157675	19.1	U	term-by-document
bmw7st1	141347	141347	3740507	26.5	U	unstructured, FEM
stanford	281903	281903	2312497	8.2	U	link matrix
stanford-berkeley	683446	683446	7583376	11.1	U	link matrix
wikipedia-20051105	1634989	1634989	19753078	12.1	U	link matrix
cage14	1505785	1505785	27130349	18.0	S	struct. symm., DNA electrophoresis

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 cache line size is $L_S = 64$ bytes. 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 L1 data cache (besides a 64 kB L1 instruction cache) and a 4 MB 8-way L2 cache which is semi-shared among two cores. Furthermore, there is a 32 MB 16-way L3 cache which is fully shared among the two cores. The cache line size is $L_S = 128$ bytes. Each node of 16 processors has 128 GB main memory.

Several test matrices were taken from the University of Florida Sparse Matrix Collection [8]. We also include some test matrices used in [38]. Table 8.1 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 occurring in finite element methods (FEM) on a *regular* grid. The matrix `s3dkt3m2` is such a matrix, possessing relatively dense blocks along three diagonals.

Another structured matrix is `memplus`, obtained from memory circuit simulation; see Figure 8.1 (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 8.1 (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 a few cut rows, and thus we expect large performance gains. Note that the matrix may seem very dense in Figure 8.1 (right), created by using the `spy` plotting command from Matlab^{TM}, but in fact nz/row ≈ 10.3. The seemingly high density

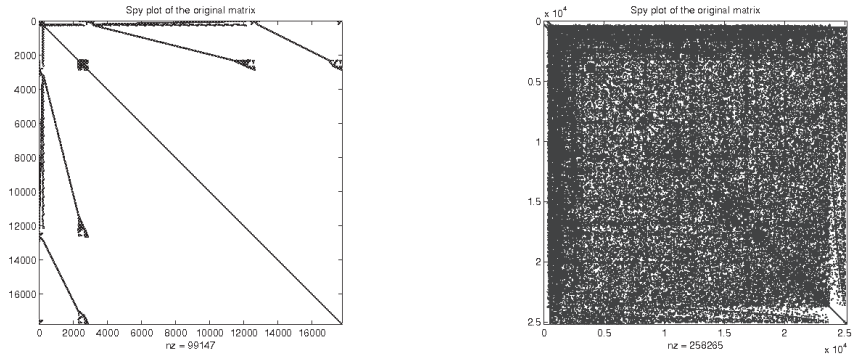

FIG. 8.1. *Plot of the original memplus (left) and rhpentium (right) matrices. Memplus has a favorable structure, whereas rhpentium looks unstructured. Accidental zeros are removed from the data structure in this plot; therefore, the given number of nonzeros can differ from that in Table* 8.1.

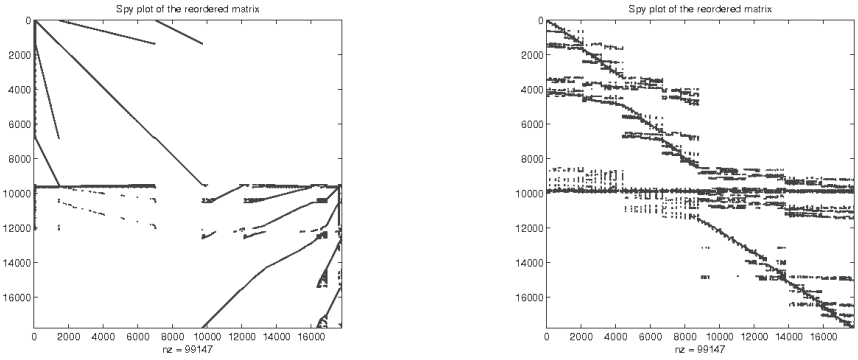

FIG. 8.2. *Plots of the memplus matrix for* $p = 2$ *(left) and* $p = 100$ *(right).*

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 rows 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, so our method may still work well.

Due to the large number of columns of our test matrices, taking the number of parts to infinity (in fact, $p = n$) may take a very long time. We therefore limit the number of parts to an initial maximum of 400, or even less in some experiments. One should note that for a given p 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 sizes 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 favorably structured matrix. After a single bipartitioning, however, we do see a structural improvement on the matrix `memplus`. In Figure 8.2 (left) the top half of the permuted matrix shows straight lines exactly corresponding to the beneficial original structure. The bottom half contains the less favorably structured parts, compressed

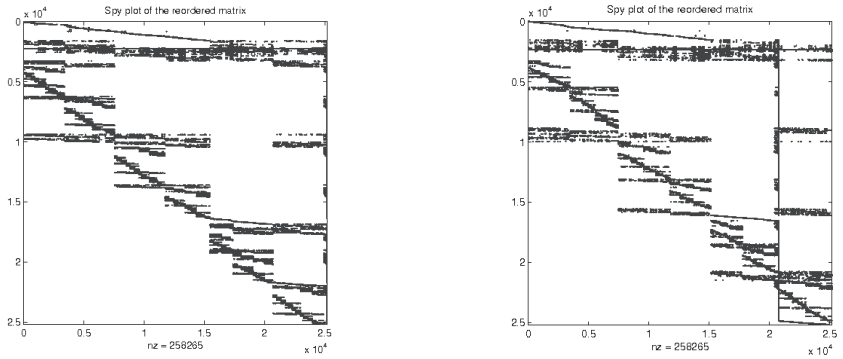

FIG. 8.3. *Plots of the rhpentium matrix for $p = 100$ (left) and $p = 400$ (right).*

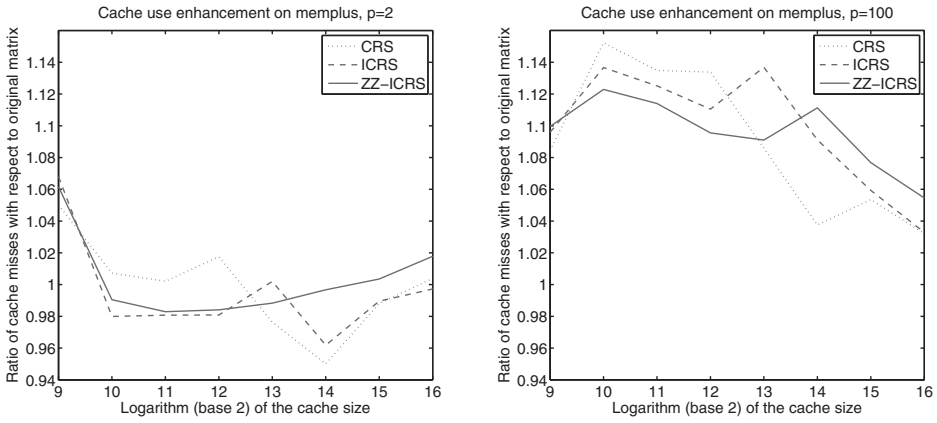

FIG. 8.4. *Simulated cache effect of our reordering method for the memplus matrix, plotted against different cache sizes. Given is the ratio between the number of cache misses for the reordered matrix and the number for the original matrix, for three data structures, namely CRS, ICRS, and ZZ-ICRS. On the left we have simulations with $p = 2$, while on the right we have $p = 100$. The cache 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.*

together. At $p = 100$ (right), things have become far less favorably structured as our method tries to compress the nonzeros in smaller and smaller areas.

Examining the unstructured matrix `rhpentium`, shown in Figure 8.3, we see a completely different behavior. At $p = 100$, a good structure has already surfaced, which is improved at $p = 400$ as the thickness of the rows corresponding to $\mathcal{N}^c$ has been reduced.

The simulated cache use efficiency shown in Figures 8.4 and 8.5 reflects the behavior observed above. These figures show the reduction of cache misses, with respect to the simulated cache size. For `memplus`, $p = 2$ gains of about 2 percent are recorded, while for $p = 100$ losses of the order of 10 percent can be seen. For `rhpentium`, large gains of up to 35 percent can be seen for all three data structures. It is interesting to see that the gain curve for $p = 400$ improves for lower cache sizes when compared to the 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, e.g., when using the L1 cache of an Intel Core 2 processor. In the figure, the largest gain is about 37 percent. On the other hand, partitioning to infinity makes

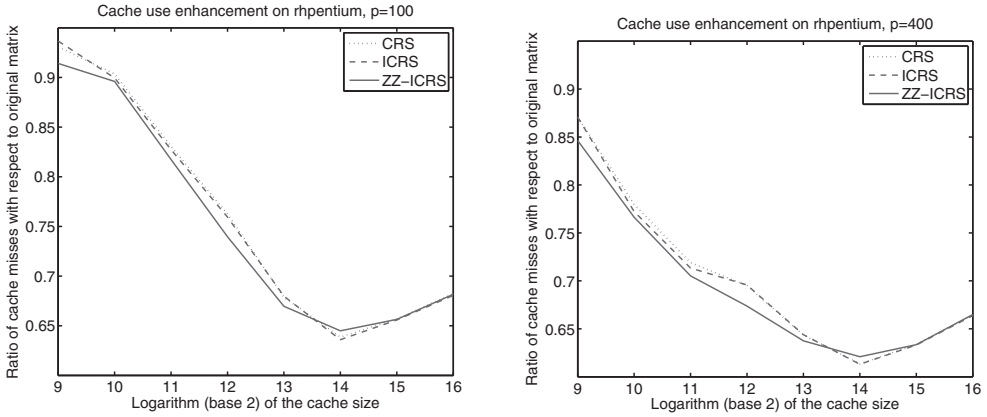

FIG. 8.5. *Simulated cache effect of our reordering method for the rhpentium matrix, plotted against different cache sizes. On the left we have simulations with* $p = 100$*, while on the right we have* $p = 400$*. We again are simulating an* 8*-way set-associative cache with* $L_S = 64$*.*

the ordering truly cache-oblivious since it works well irrespective of the actual cache size. This also hints at good performance on multilevel cache architectures: the early partitionings optimize for the larger caches, while subsequent refinements increase performance on the smaller (L1) caches.

Figure 8.5 also shows that the gain of reordering eventually tends to decrease as the cache size increases; this indicates that the more data fit into cache, the less reordering can improve upon cache misses. 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 8.2 presents the same simulations as Figures 8.4 and 8.5, but now with a fixed cache size, a varying number of parts, and a complete test set of matrices. Table 8.2 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 Tables 8.1 and 8.2, it can be seen that matrices with already favorable 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 $p = 2$. For the highly structured matrix `s3dkt3m2`, we did not expect any gains, and indeed we do not observe any change.

The miss ratios in Figures 8.4, 8.5, and Table 8.2 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 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.

Figures 8.6 and 8.7 show the gain in wall-clock timings of benchmark experiments for the smaller test matrices. Here, we show the ratio between the SpMV multiplica-

TABLE 8.2

Simulated cache effect of our reordering method for the complete matrix test set and for a varying number of parts 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 parentheses. 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, $S = 2^{15}$, $L_S = 64$, and $k = 8$.

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)

tion time for the reordered matrix and that for the original matrix. The matrices used in these benchmarks are relatively small: an Intel Core 2 L2 cache has size $S = 2^{22}$ bytes (4 MB), 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 8.1, 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 behavior; hence cache effect enhancements translate to relatively small amounts of gain in execution time, especially since the partitioning has been stopped at $p = 400$, with p still far from n. Carrying the partitioning much further would probably show more gains from the L1 cache.

Figure 8.6 shows results for our own straightforward SpMV implementation. 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. 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. 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 behavior not included in our cache simulation model.

OSKI results are shown in Figure 8.7. Note that OSKI uses its own optimized SpMV multiplication implementation, and the ratios are given with respect to the OSKI benchmark on the original matrix. It is surprising to see that OSKI achieves some improvement on `memplus`, reporting larger gains as p increases; however, closer inspection reveals that in this case, the OSKI SpMV routine performs worse than our own standard implementation. These slow SpMVs are improved by reordering, but even with $p = 400, \epsilon = 0.3$ the OSKI SpMV routine is still slower (0.42 ms) than a standard implementation (0.35 ms). Regarding most other matrices, the OSKI routine is noticeably faster.

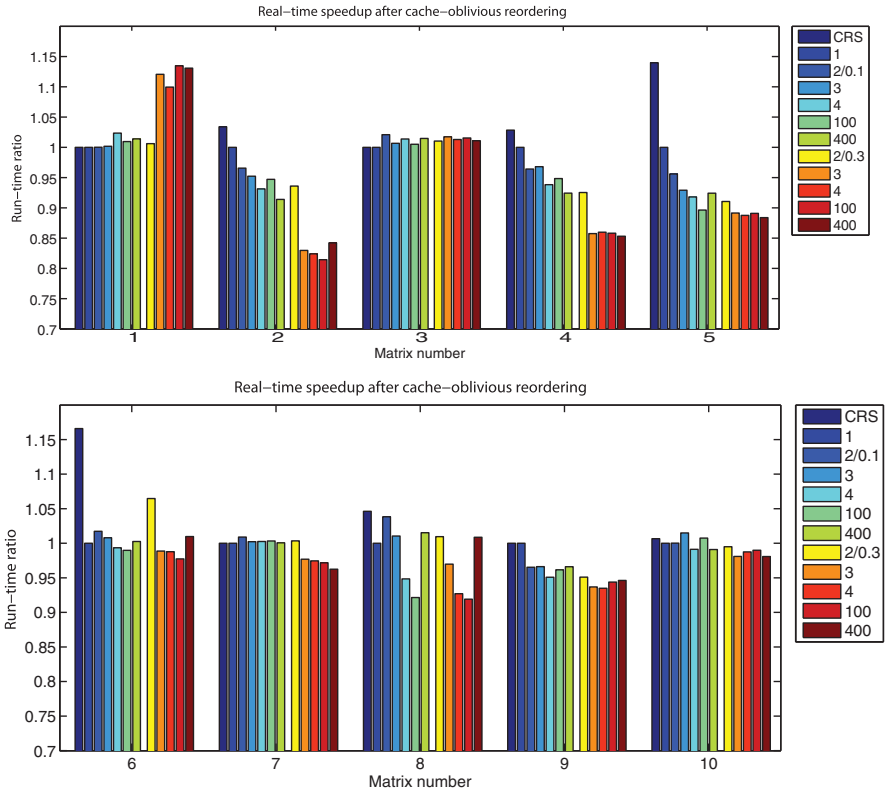

FIG. 8.6. *Wall-clock timings performed on an Intel Core* 2 *(Q*6600*) machine. The CRS bar denotes the timing of a straightforward CRS-based SpMV multiplication applied to the original matrix. The bar* $p = 1$ *corresponds to the best timing of CRS, ICRS, and ZZ-ICRS on the original matrix. The bars* $p = 2/0.1, 3, 4, 100, 400$ *show the results after reordering using* p *parts, 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 test matrices are as follows:* 1. *memplus;* 2. *rhpentium;* 3. *s*3*dkt*3*m*2*;* 4. *rand*10000*;* 5. *fidap*037*;* 6. *lhr*34*;* 7. *rand*50000*;* 8. *nug*30*;* 9. *tbdlinux;* 10. *bmw*7*st*1*.*

Figures 8.8 and 8.9 show results for much larger matrices. We feature several link matrices, as well as the `cage14` matrix. (A *link* matrix A is a matrix with $a_{ij} \neq 0$ if and only if 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 already leads to substantial gains, and increasing p gives even better results, with a gain of over 50 percent for the `stanford` matrix.

The `stanford_berkeley` matrix breaks about even in run-time, which is very different from the `stanford` matrix which shows much gain. Apparently, detecting a favorable structure on the `stanford_berkeley` matrix is more difficult than for the other link matrices. 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.

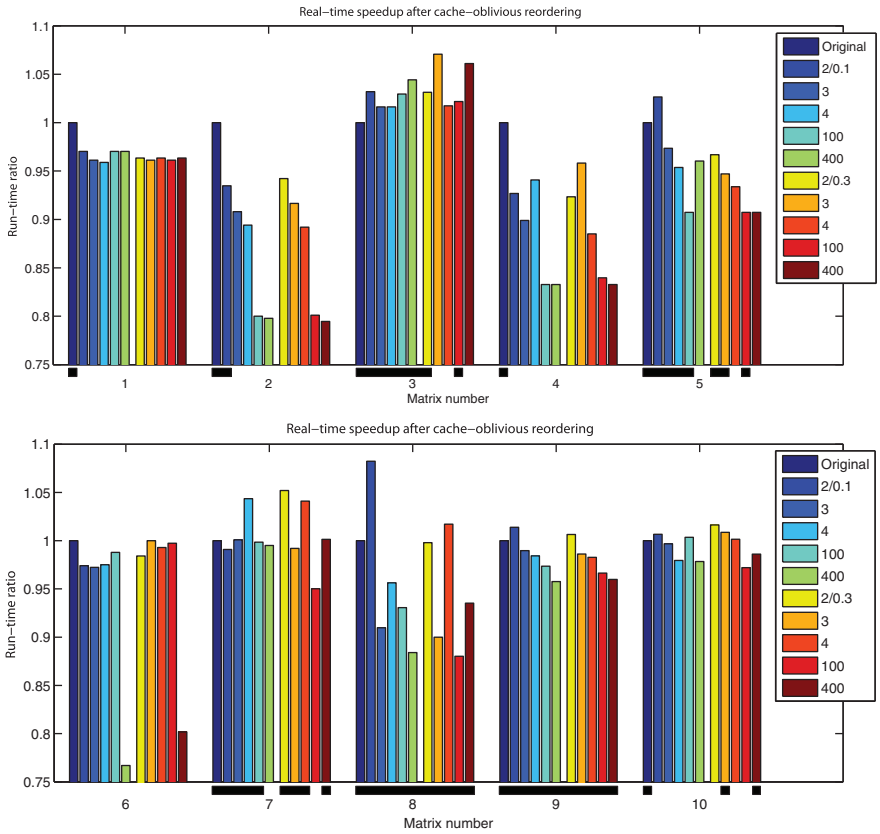

FIG. 8.7. *Wall-clock timings obtained by using the OSKI library for the SpMV multiplication, instead of our own implementation. Note that OSKI sparse matrix storage is CRS-based. The cases where OSKI automatically applied cache-aware tuning are marked.*

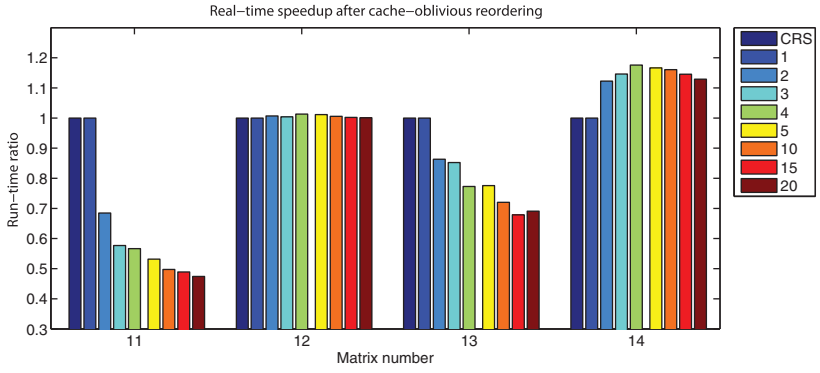

FIG. 8.8. *Wall-clock timings for very large matrices, where the input and output vector do not fit into the L2 cache. Interpretation is the same as for Figure* 8.6, *with* $\epsilon = 0.1$ *when our matrix reordering method is applied. The test matrices are as follows:* 11. *stanford;* 12. *stanford_berkeley;* 13. *wikipedia*-20051105*;* 14. *cage*14.

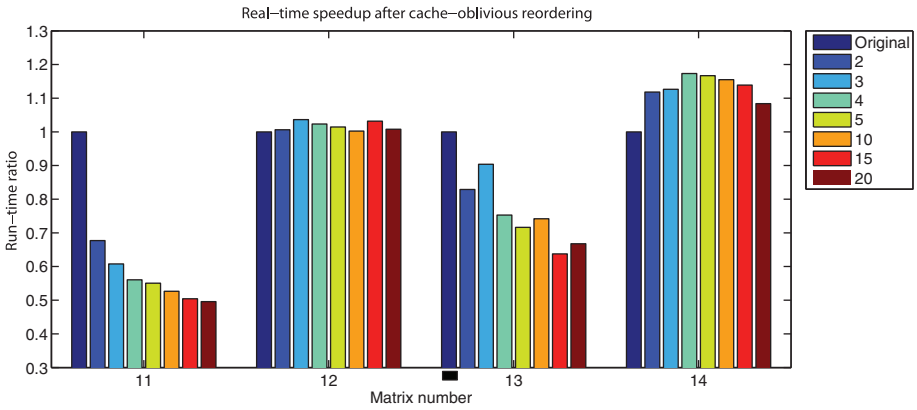

FIG. 8.9. *Wall-clock timings for very large matrices, but using the OSKI library.*

The matrix `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 [37]. 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 behavior.

It is difficult for a straightforward CRS-based SpMV multiplication algorithm to compete with OSKI's optimized implementation (OSKI is always faster, except for the `memplus` matrix, as discussed earlier). When we compare the OSKI timings on the original matrix to timings using our own implementation on reordered matrices, we see that reordering was more efficient for three out of the nine smaller matrices (we exclude the `memplus` timings). For the larger matrices, this is the case for two out of four: `stanford` and `wikipedia`. In the case of `stanford`, we observe a decrease from 27.4 ms per SpMV multiplication (OSKI on the original matrix) to 15.4 ms (reordering with $p = 20$, $\epsilon = 0.1$, CRS without OSKI), a 44 percent speedup. As shown earlier, we can join both methods and use OSKI on the reordered matrices. This yields improvements with respect to OSKI on the original matrix in twelve out of the fourteen cases, the only exceptions being `s3dkt3m2` and `cage14`, both cases where reordering did not gain anything or even caused performance loss compared to the original ordering.

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 better performance due to the presence of an L3 cache on larger problems. Figure 8.10 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, but we no longer monotonically get faster after $p = 4$; it may take a larger p before the timings start to decrease on this architecture. We observed that ICRS on Huygens is frequently the fastest implementation and that it is noticeably faster than CRS. The `stanford_berkeley` matrix again shows similar results; the

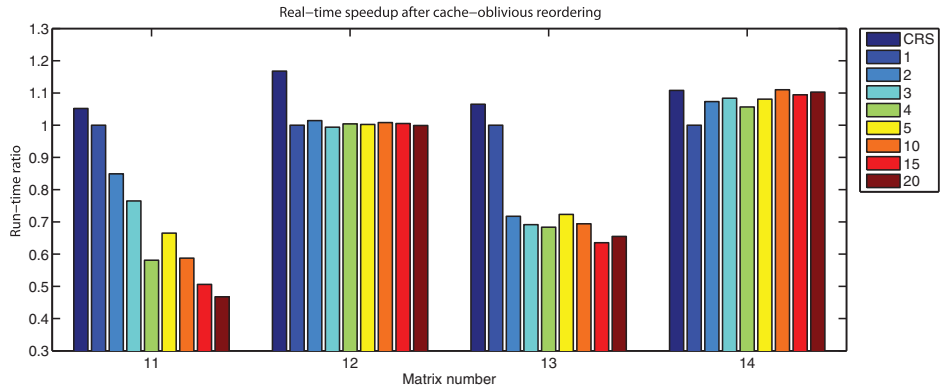

FIG. 8.10. *Experiments similar to Figure* 8.8*, but run on the Dutch national supercomputer Huygens.*

TABLE 8.3

Cost of reordering in terms of the number of matrix multiplications on the original matrix. Here, $\epsilon = 0.1$*. Construction times were measured on an Intel Core* 2 *(Q*6600*) machine.*

Name	$p = 2$	$p = 3$	$p = 4$	$p = 100$	$p = 400$
memplus	1531	1914	1818	12793	101744
rhpentium	5090	6752	7303	17064	60251
s3dkt3m2	603	673	740	1934	5181
rand10000	1560	1411	1820	23179	103565
fidap037	1005	1068	1131	5657	12761
lhr34	635	708	730	1599	6513
rand50000	2868	4065	5135	34710	120070
nug30	1594	1950	2204	10961	54588
tbdlinux	2258	3219	3659	26233	137332
bmw7st1	642	725	758	1946	5059

Name	$p = 2$	$p = 3$	$p = 4$	$p = 10$	$p = 20$
stanford	5139	7603	6828	7922	8332
stanford_berkeley	11305	13208	15093	19138	21260
wikipedia20051105	2152	1992	2168	2570	7418
cage14	4238	3987	3635	4583	5611

Power6+ L2 cache has exactly the same size as the Intel Core 2 L2 cache, and hence we do not gain additional information as to the lack of speedup.

A relevant question is whether the number of SpMVs is large enough to justify reordering the matrix first. Of course, the time required for reordering increases with p, as well as with the matrix size. Table 8.3 presents the reordering time expressed in the number of SpMV multiplications. The SpMV multiplication time we compare with is the time required for one SpMV *without* reordering. One sees that with increasing p, the number of multiplications required increases rapidly. In conjunction with the diminishing savings in SpMV multiplication time as p increases, we note that taking $p \to \infty$ may not be practical. Still, research towards decreasing the construction time for reordering would be useful in the sense that the required number of SpMV multiplications to justify reordering would decrease, so that we may obtain better quality reorderings for the same number of SpMV multiplications.

9. 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 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 amortized 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 [39], this is partly a consequence of not carrying through the partitioning till the end. Further research on improving the partitioning speed should take the number of parts 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 as follows:

- using a zig-zag variant of the CRS data structure, thus avoiding unnecessary cache misses at the end of rows;
- placing cut rows in the middle during the partitioning process, leading to a gradual transition between a cache filled with data from one column set $\mathcal{V}_0$ to another set $\mathcal{V}_1$;
- 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 [38], 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 [5], hMETIS [23], Zoltan [10], Monet [20], and Par$k$way [35]. Using the parallel partitioner Zoltan, or Par$k$way, would enable a parallel reordering.

Mondriaan is, as of yet, not designed with taking the number of parts to infinity in mind. This translates to high reordering times. These depend on p and the matrix structure, but not directly on the number of nonzeros (nor the average number of nonzeros per row). For this method to perform well in practice, further research is required to decrease the reordering times. It is expected that additional speedups can be obtained by removing time-consuming optimizations that mainly make sense for smaller 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 can be done by either modeling the matrix by using the fine-grain method [6], 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 the choice of fine-grain, 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.

Acknowledgments. We thank Fatima Abu Salem for helpful discussions on cache-oblivious algorithms. We thank Sarai Bisseling for creating Figure 6.2. We are grateful to the anonymous referees who helped us a lot by their careful reading and constructive remarks. 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.

REFERENCES

[1] C. Aykanat, A. Pinar, and Ü. V. Çatalyürek, *Permuting sparse rectangular matrices into block-diagonal form*, SIAM J. Sci. Comput., 25 (2004), pp. 1860–1879.

[2] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst, eds., *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*, SIAM, Philadelphia, 2000.

[3] M. A. Bender, G. S. Brodal, R. Fagerberg, R. Jacob, and E. Vicari, *Optimal sparse matrix dense vector multiplication in the I/O-model*, in Proceedings of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures, ACM Press, New York, 2007, pp. 61–70.

[4] S. Brin and L. Page, *The anatomy of a large-scale hypertextual web search engine*, in Comput. Networks ISDN Systems, 30 (1998), pp. 107–117.

[5] 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.

[6] Ü. V. Çatalyürek and C. Aykanat, *A fine-grain hypergraph model for 2D decomposition of sparse matrices*, in Proceedings of the 8th International Workshop on Solving Irregularly Structured Problems in Parallel, IEEE Press, Los Alamitos, CA, 2001, p. 118.

[7] R. Das, D. J. Mavripilis, J. Saltz, S. Gupta, and R. Ponnusamy, *Design and implementation of a parallel unstructured Euler solver using software primitives*, AIAA J., 32 (1994), pp. 489–496.

[8] T. A. Davis, *University of Florida sparse matrix collection*, http://www.cise.ufl.edu/research/sparse/matrices, Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL, 1994-2008.

[9] J. M. Dennis and E. R. Jessup, *Applying automated memory analysis to improve iterative algorithms*, SIAM J. Sci. Comput., 29 (2007), pp. 2210–2223.

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

[11] 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.

[12] M. Frigo and S. 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.

[13] M. Frigo and S. G. Johnson, *The design and implementation of FFTW3*, Proc. IEEE, 93 (2005), pp. 216–231.

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

[15] A. George, *Nested dissection of a regular finite element mesh*, SIAM J. Numer. Anal., 10 (1973), pp. 345–363.

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

[17] L. Grigori, E. Boman, S. Donfack, and T. A. Davis, *Hypergraph-based unsymmetric nested dissection ordering for sparse LU factorization*, Technical report 6520, INRIA Saclay, Orsay, France, April 2008.

[18] G. Haase, M. Liebmann, and G. Plank, *A Hilbert-order multiplication scheme for unstructured sparse matrices*, Int. J. Parallel Emergent Distrib. Syst., 22 (2007), pp. 213–220.
[19] B. Hendrickson and E. Rothberg, *Improving the run time and quality of nested dissection ordering*, SIAM J. Sci. Comput., 20 (1998), pp. 468–489.
[20] Y. F. Hu, K. C. F. Maguire, and R. J. Blake, *A multilevel unsymmetric matrix ordering algorithm for parallel process simulation*, Comput. Chem. Engrg, 23 (2000), pp. 1631–1647.
[21] E.-J. Im and K. A. Yelick, *Optimizing sparse matrix–vector multiplication for register reuse in SPARSITY*, in Proceedings of the International Conference on Computational Science, Part I, Lecture Notes in Computer Science 2073, 2001, pp. 127–136.
[22] G. Karypis and V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM J. Sci. Comput., 20 (1998), pp. 359–392.
[23] G. Karypis and V. Kumar, *Multilevel k-way hypergraph partitioning*, in Proceedings of the 36th ACM/IEEE Conference on Design Automation, ACM Press, New York, 1999, pp. 343–348.
[24] T. Katagiri, K. Kise, H. Honda, and T. Yuba, *ABCLibScript: A directive to support specification of an auto-tuning facility for numerical software*, Parallel Comput., 32 (2006), pp. 92–112.
[25] J. M. Kleinberg, *Authoritative sources in a hyperlinked environment*, J. ACM, 46 (1999), pp. 604–632.
[26] J. Koster, *Parallel templates for numerical linear algebra, a high-performance computation library*, Master's Thesis, Department of Mathematics, Utrecht University, 2002.
[27] M. Kowarschik, U. Rüde, C. Weiss, and W. Karl, *Cache-aware multigrid methods for solving Poisson's equation in two dimensions*, Computing, 64 (2000), pp. 381–399.
[28] A. N. Langville and C. D. Meyer, *Google's PageRank and Beyond: The Science of Search Engine Rankings*, Princeton University Press, Princeton, NJ, 2006.
[29] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*, John Wiley and Sons, Chichester, UK, 1990.
[30] 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.
[31] A. Pinar and M. T. Heath, *Improving performance of sparse matrix-vector multiplication*, in Proceedings Supercomputing 1999, ACM Press, New York, 1999, p. 30.
[32] M. M. Strout, L. Carter, J. Ferrante, and B. Kreaseck, *Sparse tiling for stationary iterative methods*, Int. J. High Perf. Comput. Appl., 18 (2004), pp. 95–113.
[33] M. M. Strout and P. D. Hovland, *Metrics and models for reordering transformations*, in Proceedings of the 2004 Workshop on Memory System Performance, ACM Press, New York, 2004, pp. 23–34.
[34] S. Toledo, *Improving the memory-system performance of sparse-matrix vector multiplication*, IBM J. Res. Dev., 41 (1997), pp. 711–725.
[35] A. Trifunovic and W. J. Knottenbelt, *A parallel algorithm for multilevel k-way hypergraph partitioning*, in Proceedings of the 3rd International Symposium on Parallel and Distributed Computing, IEEE Press, Los Alamitos, CA, 2004, pp. 114–121.
[36] R. van der Pas, *Memory hierarchy in cache-based systems*, Tech. Report 817-0742-10, Sun Microsystems, Inc., Santa Clara, CA, 2002.
[37] A. van Heukelum, G. T. Barkema, and R. H. Bisseling, *DNA electrophoresis studied with the cage model*, J. Comput. Phys., 180 (2002), pp. 313–326.
[38] 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.
[39] 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.
[40] R. W. Vuduc and H.-J. Moon, *Fast sparse matrix-vector multiplication by exploiting variable block structure*, in High Performance Computing and Communications 2005, Lecture Notes in Computer Science 3726, 2005, pp. 807–816.
[41] R. C. Whaley and A. Petitet, *Minimizing development and maintenance costs in supporting persistently optimized BLAS*, Software Pract. Exper., 35 (2005), pp. 101–121.
[42] 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.
[43] J. B. White, III and P. Sadayappan, *On improving the performance of sparse matrix-vector multiplication*, in Proceedings of the 4th International Conference on High-Performance Computing, IEEE Press, Los Alamitos, CA, 1997, pp. 66–71.