High-level strategies for parallel shared-memory sparse matrix–vector multiplication

A. N. Yzelman
D. Roose

Report TW 614, June 2012

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

High-level strategies for parallel shared-memory sparse matrix–vector multiplication

A. N. Yzelman
D. Roose

Report TW 614, June 2012

Department of Computer Science, KU Leuven

Abstract

The sparse matrix–vector multiplication is an important kernel, but is hard to efficiently execute even in the sequential case. The problems –namely low arithmetic intensity, inefficient cache use, and limited memory bandwidth– are magnified as the core count on shared-memory parallel architectures increases. Existing techniques are discussed in detail, and categorised chiefly based on their distribution types. Based on this new parallelisation techniques are proposed. The theoretical scalability and memory usage of the various strategies are analysed, and experiments on multiple NUMA architectures confirm the validity of the results. One of the newly proposed methods attains the best average result in experiments, in one of the experiments obtaining a parallel efficiency of 90 percent.

Keywords : Sparse matrix–vector multiplication, shared-memory parallelism, cache-oblivious, sparse matrix partitioning, matrix reordering, Hilbert space-filling curve, high-performance computing, NUMA architectures.
MSC : Primary : 65Y05, Secondary : 65F50, 68W10.

High-level strategies for parallel shared-memory sparse matrix–vector multiplication

A. N. Yzelman*[1,2] and D. Roose[1]

[1]Department of Computer Science, KU Leuven
[2]Intel ExaScience Lab

June 22, 2012

Abstract

The sparse matrix–vector multiplication is an important kernel, but is hard to efficiently execute even in the sequential case. The problems – namely low arithmetic intensity, inefficient cache use, and limited memory bandwidth– are magnified as the core count on shared-memory parallel architectures increases. Existing techniques are discussed in detail, and categorised chiefly based on their distribution types. Based on this new parallelisation techniques are proposed. The theoretical scalability and memory usage of the various strategies are analysed, and experiments on multiple NUMA architectures confirm the validity of the results. One of the newly proposed methods attains the best average result in experiments, in one of the experiments obtaining a parallel efficiency of 90 percent.

Keywords: Sparse matrix–vector multiplication, shared-memory parallelism, cache-oblivious, sparse matrix partitioning, matrix reordering, Hilbert space-filling curve, high-performance computing, NUMA architectures.

1 Introduction

The sparse matrix–vector (SpMV) multiplication is an important kernel in many areas of scientific computing: it arises as the most time-consuming part of iterative sparse linear system solvers such as CG [14], GMRES [27], Bi-CGstab [32], IDR(s) [29], of other iterative schemes such as Jacobi-Davidson [28], Lanczos [23], LSQR [22] or the PageRank algorithm [6], and of other methods.

Two factors impede creating an efficient SpMV multiplication kernel on shared-memory parallel architectures: the often non-regular data accesses, and the limited CPU-to-memory bandwidth. Both factors already are an issue during sequential SpMV multiplies [30, 16, 34], and if left unchecked, these effects will escalate when parallelising on shared-memory architectures.

*Corresponding author. Address: Celestijnenlaan 200A - bus 2402, B-3001 Heverlee, Belgium. Telephone: +32 163 275 38. E-mail: Albert-Jan.Yzelman@cs.kuleuven.be.

In this article, a summary of existing strategies is given. These are expanded upon, and new strategies for mitigating the effects of unpredictable memory accesses and high bandwidth requirements are constructed. The focus is on a high-level description of cache-oblivious strategies and a comparison thereof; low-level optimisations are left to the compiler. The different strategies are experimented with on three different non-uniform memory access (NUMA) architectures.

This section will continue with a brief overview of sparse matrix storage schemes, their corresponding SpMV multiplication kernels, and their effects on cache-based architectures. Section 2 then describes related work, and discusses sequential, shared-memory parallel, and distributed-memory parallel solutions. Some of those techniques are categorised and extended upon to obtain the high-level strategies found in Section 3. Section 4 puts those strategies to the test in experiments on various shared-memory machines. Finally, Section 5 concludes the paper and suggests future work.

1.1 Sparse matrix storage schemes

The coordinate format (COO) stores the nz nonzeroes of a sparse $m \times n$ matrix A in three arrays I, J, V. It does this by letting the kth nonzero $a_{ij} \in A$ correspond to a triplet $(i, j, a_{ij}) = (I_k, J_k, V_k)$. The storage requirement is $\Theta(3nz)$, and the corresponding SpMV multiplication kernel adds $V_k x_{J_k}$ to y_{I_k} for each $0 \leq k < nz$, where x is the dense $n \times 1$ input vector and y the dense $m \times 1$ output vector.

Reduction of the memory footprint by compressing the storage of A directly reduces the bandwidth requirement. The widely used Compressed Row Storage (CRS) [1] achieves this by a row-major ordering of the nonzeroes: nonzeroes are sorted first on row index, and second on column index, both in ascending order. This enables row-wise compression since the array I can now be replaced with an array $\hat{I}$ of size $m+1$, which records the start (and end) positions of each row within the arrays J and V. An outline of the CRS SpMV kernel clarifies this idea:

for $i = 0$ **to** $m - 1$ **do**
  for $k = \hat{I}_i$ **to** $\hat{I}_{i+1} - 1$ **do**
    add $V_k \cdot x_{J_k}$ to y_i

Employing a column-major nonzero ordering to apply compression on J instead of I yields the Compressed Column Storage (CCS). With CRS the memory requirement drops to $\Theta(2nz + m)$, and for CCS to $\Theta(2nz + n)$.

Let nonzeroes be in CRS order and ΔJ be the *incremental column index array*, with $\Delta J_0 = J_0$ and

$$\Delta J_k = J_k - J_{k-1} + \begin{cases} 0, \text{ if } I_k = I_{k-1} \\ n \text{ otherwise} \end{cases}, \text{ for } 1 \leq k < n.$$

Similarly, let $\Delta I_0 = I_0$ while $\Delta I_{r>0}$ stores the size of the rth row jump, so that $\sum_{q=0}^{i} \Delta I_q$ is the index of the $(i+1)$th nonempty row of A. Then $(\Delta I, \Delta J, V)$ is the Incremental CRS (ICRS) scheme [17]. All increments in ΔI and ΔJ are positive, and if there are no empty rows in A, all elements of ΔI equal 1, except for the first element which is 0. If negative increments are allowed, however, then $(\Delta I, \Delta J, V)$ is the Bi-directional ICRS (BICRS) scheme [40],

Algorithm 1 BICRS SpMV multiplication kernel

```
1: k = 0, r = 0, i = ΔI_0, j = ΔJ_0
2: while k < nz do
3:     while j < n do
4:         y_i = y_i + v_k · x_j
5:         j = j + ΔJ_k, k = k + 1
6:     j = j − n
7:     i = i + ΔI_r, r = r + 1
```

which can handle arbitrary nonzero orderings. In both cases, incrementing the column index beyond n, indicates a row change. The corresponding BICRS SpMV kernel is shown in Algorithm 1. Switching the roles of rows and columns yield the ICCS and BICCS schemes; note that these differ from their row-major counterparts if A is nonsymmetric.

The storage requirement of ICRS is $\Theta(2nz + \tilde{m})$, where $\tilde{m}$ is the number of nonempty rows. If all rows are nonempty ΔI need not be stored and the requirement drops to $\Theta(2nz)$. BICRS storage is $\Theta(2nz + m_j)$, with m_j the number of row-jumps according to the nonzero ordering used. Since $m_j \leq nz$, the storage requirement of BICRS is equal to that of COO, worst-case. For both ICRS and BICRS, storage improves on that of COO, and improves on that of CRS as well, if $\tilde{m}, m_j < m$ (as, e.g., occurs with sparse blocking; see Section 3.1).

1.2 Caches and multiplication

The irregular access patterns during SpMV multiplication greatly reduce the efficiency of caches typically available on shared-memory architectures. To describe this problem in greater detail, first some notions on caches are defined. A cache of size s consists of l cache lines of size s/l each. Such a line is the smallest unit of data which can be individually brought into cache, and one line typically contains multiple data elements. Main memory is likewise partitioned in aligned stripes of size s/l. When a core requests data within the zth memory stripe and z is in cache, then a *cache hit* occurs and the data is quick to arrive. Otherwise, a *cache miss* occurs and the memory stripe must be copied from main memory first. A core stalls during a cache miss if it has no other work. When the flop-to-data ratio, the *arithmetic intensity*, is low, such stalls are common; indeed, in the case of the SpMV multiplication, often only a fraction of peak performance is achieved in practice [30, 37, 34]. Increasing the cache hit-rate increases performance by having data available more quickly, but also by alleviating bandwidth pressure between cache and main memory; this happens even though the overall arithmetic intensity remained unchanged. Note that the data storage of A never benefits from caching since nonzeroes are used only once; the reuse pattern of elements from x and y is what determines cache efficiency.

Current architectures often have multiple caches placed in between processor and main memory. Caches close to a core, the lower-level caches, are faster but smaller than the higher-level caches closer to main memory. Parallel shared-memory architectures consist of multiple cores which may have individual caches, while groups of cores may share (higher-level) caches, or share a

single memory controller. If a shared cache or a shared memory controller is not shared between all cores, then memory access times between pairs of cores may vary. Such an architecture then is said to have Non-Uniform Memory Access (NUMA).

1.3 Ingredients for parallelisation

A parallel shared-memory SpMV multiplication scheme stores A efficiently and evenly splits the processing of nonzeroes between the p available cores. Each core executes a sequential SpMV kernel based on the chosen storage scheme, which can be a regular CRS or COO scheme, or some optimised scheme. Optimised strategies either depend on the specifics of the computer hardware and cache specifics (cache-aware), or they strive to attain good performance regardless of architecture specifics (cache-oblivious). Cache-aware strategies often necessitate knowledge about the structure of A to perform well. Both types of strategies may require pre-processing to find or induce beneficial matrix structures.

While the choice of storage scheme is independent of the distribution of work, it does matter how a distribution is enforced. On shared-memory architectures, the matrix and vectors can be allocated in one contiguous chunk of memory. Each core then accesses elements within that chunk according to the work distribution, and this is referred to as an implicit distribution. With an explicit distribution, each core allocates its own memory chunks, which contain only the data elements it operates on given a specific work distribution. The choice of implicit or explicit distribution can significantly impact performance, especially on NUMA architectures.

Distribution, whether implicit or explicit, deals with finding a mapping $\pi_A : \{a_{ij} \in A\} \to \{0, \ldots, p-1\}$ which assigns each nonzero to a process. If $\pi_A(a_{ij}) = s$, then a_{ij} is said to be local to s. Similarly, mappings π_x and π_y for the elements of x and y can be defined. If one of the parallel processes needs to work with elements not local to it, communication occurs; partitioning strategies minimise this while keeping the number of local elements balanced across the processes. One-dimensional (1D) strategies map elements a_{ij} according to either its row index i (row-wise 1D) or to its column index j (column-wise 1D), while two-dimensional (2D) strategies consider both indices simultaneously.

1D strategies for parallel shared-memory SpMV multiplication usually partition matrix rows only, so to split the output vector y in contiguous blocks (essentially taking $\pi_A = \pi_y$). Each core access only subvectors of y corresponding to a unique block, thus avoiding concurrent writes on the output vector. All cores generally still operate on the entire input vector range. A 2D strategy would split both the input and output vector in contiguous (but generally overlapping) blocks, each of which are again processed by a single core only. Another option, since vectors reside in shared memory, is to not distribute (ND) the vectors at all. In all cases the distribution of A determines the load-balance.

1.4 Sparse matrix partitioning

Distributed-memory parallel SpMV multiplication relies on sparse matrix partitioning as a pre-processing step. Nonzeroes of A are split into p disjoint subsets to form smaller local matrices $A^{(s)}$, with $0 \leq s < p$. Each parallel process performs a sequential SpMV multiply of $y^s = A^{(s)}x^s$, with x^s and y^s possibly

overlapping subsets of the input and output vectors. A good matrix partitioning keeps $nz(A^{(s)})$ close to nz/p (load balance), and minimises communication involving the overlapping sections of x^s and y^s. Examples of sparse matrix partitioners are Mondriaan [33], Zoltan [12], Scotch [24] and par$k$way [31].

Some of the strategies suggested in this paper are closely linked to how matrix partitioners operate, in particular those partitioners based on hypergraph models. These model the input matrix as a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$, where each nonzero a_{ij} is a vertex in $\mathcal{V}$. Each row i is represented by a net $n_i \subseteq \mathcal{V}$, such that $\forall j \in \{0, \ldots, n-1\}$ and $a_{ij} \neq 0$, $a_{ij} \in n_i$. The set of all row-nets is $\mathcal{N}^{\text{row}}$. In a similar fashion the set of column nets $\mathcal{N}^{\text{col}}$ is constructed, and its union with the row net gives the entire set of nets $\mathcal{N}$ corresponding to this hypergraph of A. In this *fine-grain* representation of A [11], partitioning corresponds to splitting $\mathcal{V}$ into p pairwise disjoint sets $\mathcal{V}_0, \ldots, \mathcal{V}_{p-1}$. The connectivity λ_i of a net $n_i \in \mathcal{N}$ then is defined as $\#\{s \in \{0, \ldots, p-1\} \mid n_i \cup \mathcal{V}_s \neq \emptyset\}$, that is, λ_i is the number of parts n_i spans. The communication volume incurred during SpMV multiplication is exactly given by $\sum_{i:n_i \in \mathcal{N}} (\lambda_i - 1)$; the so-called $(\lambda - 1)$ metric [18]. By grouping the vertices in $\mathcal{V}$ by row and considering only the nets in $\mathcal{N}^{\text{col}}$, the full 2D fine-grain model reduces to the 1D column-net model [10]. Similarly the 1D row-net model can be constructed. The *Mondriaan scheme* combines the row- and column-net models to efficiently construct a full 2D partitioning [33], for $p > 2$; the scheme is efficient since the computational cost is roughly twice that of a 1D hypergraph partitioning, instead of the cost of the larger and more complex fine-grain hypergraph partitioning. The quality of the results often are comparable [3].

2 Earlier work

Here an overview of existing techniques for efficient sparse matrix–vector multiplication is given, starting with sequential multiplication and ending with recently proposed parallelisation techniques.

2.1 Sequential techniques

In the sequential case, techniques include reducing the matrix bandwidth[1] to increase the cache hit-rate on vector elements [30]. This requires permuting the rows and columns of A, for instance by using the Cuthill-McKee reordering [9], or by using graph partitioning [36].

Yzelman and Bisseling employ hypergraph partitioning to permute sparse matrices into the cache-friendly Separated Block Diagonal (SBD) form [39, 40]. In case of the 2D fine-grain hypergraph model, this permutation is based on categorising the row- and column-nets of the partitioned hypergraph: for every bipartitioning of some $\mathcal{V}$ into $\mathcal{V}_{\text{left}}$ and $\mathcal{V}_{\text{right}}$, row-nets n_i are put in $\mathcal{N}_-^{\text{row}}$ if $n_i \cap \mathcal{V}_{\text{right}} = \emptyset$, in $\mathcal{N}_+^{\text{row}}$ if $n_i \cap \mathcal{V}_{\text{left}} = \emptyset$, and in $\mathcal{N}_c^{\text{row}}$ otherwise. In the exact same way, the sets $\mathcal{N}_{\{-,+,c\}}^{\text{col}}$ can be constructed, and the doubly Separated Block Diagonal (SBD) form is then defined as in Figure 1 (left). The large upper-left and lower-right blocks are the *pure blocks*, while the other five smaller ones form the *separator cross*. If this bipartitioning scheme is followed recursively

[1] not the memory bandwidth; the matrix bandwidth is the minimum of $k_1 + k_2 + 1$ s.t. $a_{ij} = 0\ \forall i, j$ with $j < i - k_1$ and $j > i + k_2$.

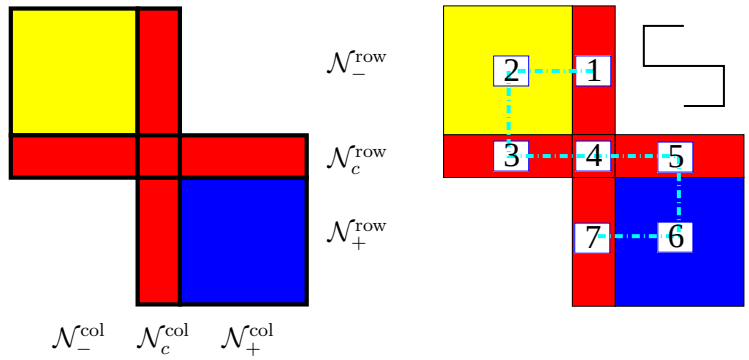


Figure 1: Separated Block Diagonal form of a sparse matrix. Construction (left) and optimised block traversal (right).

to obtain p partitions, then after reordering there are p pure blocks and $p-1$ separator crosses. Figure 3a illustrates the case where $p = 4$.

In case of the 1D row-net model, the corresponding SBD form misses the vertically-oriented blocks along the vertical axis of the separator cross, i.e., blocks $1, 4, 7$ from Figure 1 (right). Performing a slightly adapted sequential CRS-based SpMV multiplication on that form incurs a maximum of cache misses equal to the value given by the $(\lambda - 1)$ metric, if p is appropriately chosen based on cache parameters; minimisation of the communication volume thus is directly related to minimisation of cache misses [39]. Choosing p larger than necessary theoretically does not harm cache efficiency, so taking $p \to \infty$ yields a cache-oblivious method. This result does not hold for the general 2D SBD form. There, multiplication must be executed in a block-by-block fashion according to a recursively defined block order so to guarantee minimisation of cache misses; see Figure 1 (right) [40].

Another class of reordering strategies is based on adaptation of the order of processing of nonzeroes, instead of on matrix permutations. This requires a change from CRS-based multiplication, but the freedom in nonzero ordering makes it possible to access elements of the input and output vectors with improved locality. Haase et al. use an ordering induced by the Hilbert space-filling curve [13], and use the COO data structure to store the nonzeroes in this order. Although cache efficiency indeed increases, the lack of compressed storage also leads to a larger amount of data movement. Employing the Bi-directional Incremental CRS (BICRS) data structure instead reduces the memory footprint without detriment to the improved cache efficiency [41]. Section 3 discusses various parallelisation strategies based on this method.

Cache-aware optimisations often rely on blocking. Pinar et al. use matrix permutations to construct small dense blocks of nonzeroes, thus allowing for efficient register blocking [26]. Exploiting small dense blocks potentially reduces bandwidth requirements as well: the storage requirement becomes $\Theta(nz + 2b + m)$, with b the number of dense blocks. Vuduc and Moon look into similar methods to exploit slightly larger dense blocks of varying size [35]. These, as well as other cache-aware methods, are included in the cache-aware OSKI auto-tuning library for sequential sparse operations [34, 15]. It employs run-time

Algorithm 2 CRS-based 3-step parallel multiplication.

Let $(\hat{I}^s, J^s, V^s)$ be the CRS storage of $A^{(s)}$. Each processor $s \in \{0, 1, \ldots, p-1\}$ executes:

```
1:  retrieve all non-local input vector elements
    (all x^{π_x^s(j)}_{μ_x^s(j)} for which π_x^s(j) ≠ s, j = 0, ..., n − 1)
2:  for i = 0 to m_s − 1 do
3:      sum = 0
4:      for k = Î_i^s to Î_{i+1}^s − 1 do
5:          add V_k^s · x^s_{J_k} to sum
6:      if π_y^s(i) = s (if i is local) then
7:          y_i^s = sum
8:      else
9:          send (μ_y^s(i), sum) to process π_y^s(i)
10: synchronise and add incoming contributions to y^s
```

matrix reordering, matrix splitting and adapted storage schemes to achieve good performance.

2.2 Parallel techniques

The discussion of sequential strategies reveal two distinct (but non-exclusive) techniques for improving the efficiency of an SpMV kernel: adaptation of the matrix A, and adaptation of the storage scheme of sparse matrices. Either method (or combinations of methods) can improve vector element reuse as well as increase throughput by reduction of the matrix storage requirements.

Yzelman and Bisseling derive a parallel shared-memory SpMV multiplication from a distributed-memory version, which makes use of explicit sparse matrix partitioning as introduced in Section 1.4 [42]. It performs a multiplication in three steps. First each processor reads elements from x which are referenced by $A^{(s)}$ but are not present in x^s, and copies them into a local buffer. This is followed by a CRS-based local SpMV multiplication with one key difference; for each row i of $A^{(s)}$, it first is determined whether i is local to s. If it is, the contribution of multiplying x with the ith row is written to y^s; if not, the local contribution is sent to a remote process. A global synchronisation follows to ensure all processes are done with calculating remote contributions, so that in the third and final step those contributions are gathered into local output vectors.

Algorithm 2 summarises this approach. To enable fan-in and fan-out communication, each process requires maps derived from $\pi_{\{A,x,y\}}$, namely, $\pi^s_{\{x,y\}}$ and $\mu^s_{\{x,y\}}$. These maps enable process s to derive from where to obtain x^s_j: from position $\mu^s_x(j)$ of the subvector of x local to process $\pi^s_x(j)$. Similarly, each process sends its local contribution of the ith local row to $y^{\pi^s_y(i)}_{\mu^s_y(i)}$. These maps require $o(2(m+n))$ storage; Section 3.4 introduces a method which reduces this requirement.

Note this scheme is fully explicit; all $A^{(s)}, x^s$ and y^s are allocated locally. Yzelman explores an implicit 2D scheme where reordering is applied instead

of explicit partitioning [38, Chapter 5]: process s then primarily works on the pure blocks local to s, and inter-thread communication occurs only on separator crosses. To avoid write contention, $p-1$ synchronisation steps are applied during processing of the separators. Overpartitioning can be used to map multiple pure blocks to the same process, enabling optimisation by local multiplication using cache-oblivious SBD.

Several recent shared-memory parallel SpMV multiplication schemes revolve around the Morton curve [21], similarly to the sequential Hilbert-based SpMV scheme from Section 2.1. Lorton and Wise initially proposed a block Morton scheme for dense A, where A is subdivided into submatrices of size $\beta_m \times \beta_n$. The Morton ordering is then applied on the block-level, while nonzeroes within the blocks are processed using dense BLAS techniques [19].

Martone et al. [20] use a quadtree to store sparse A: the root corresponds to the full matrix, while each internal node splits its corresponding matrix into four contiguous rectangular submatrices, and each such submatrix corresponds to a child of that internal node. The four-way splitting is done with load-balancing in mind, and stops based on a cache-aware heuristic; the leaf nodes correspond to parts of a partitioning of A. An SpMV kernel on this quadtree starts at the root, where two concurrent threads are started. One thread traverses the quadtree starting from the nodes corresponding to the top row of submatrices, while the other thread traverses the bottom ones. Traversal is depth-first and happens in Morton-order. Each leaf node encountered during traversal triggers an SpMV multiplication with the nonzeroes stored there, using standard CRS/CCS.

Buluç et al. [8] divide A in blocks as well, but in a uniform way; each submatrix A_{ij} is of size $\beta \times \beta$. However, the blocks are processed in CRS order; the Morton-curve orders the nonzeroes within each block. Here, fine-grained parallelisation is employed; multiplication is split into many small tasks, more than the number of cores that actually are available. Schedulers map these tasks to actual cores; in this scheme Cilk is used [5]. A single task is given by a single row of blocks $A_{i0}, \ldots, A_{i,n-1}$. If such a block row consists of too many nonzeroes so that load balance cannot be guaranteed, it is cut into smaller tasks which may be processed in parallel. Similarly, if a block-row contained too few nonzeroes, it is merged with a neighbouring block rows. Each A_{ij} is stored in compressed COO format, and the combined scheme is called Compressed Sparse Blocks (CSB).

3 High-level strategies

In this section new strategies for efficient parallel SpMV multiplications are investigated. These are discussed per distribution scheme, starting with a simple distribution of A which does not take any vector distributions into account, followed by a one-dimensional distribution of A and y (where the input vector remains undistributed), and finally a full two-dimensional distribution.

Most of these new strategies use explicit distribution since otherwise efficiency on multi-socket NUMA systems is hard to achieve. Such architectures consist of multiple processors placed in multiple sockets, with each processor having direct access, through a single memory controller, to local memory banks. Accessing non-local memory banks results in inter-socket data movement, which is much costlier than on-socket data movement. On such systems, two meth-

ods of memory allocation are supported: either contiguous chunks are allocated on memory banks closest to the core that requests the allocation (local allocation), or contiguous chunks are cyclically allocated over all available memory banks (interleaved allocation). The latter maximises bandwidth usage when an implicit distribution over cores is employed, but only if the cores access the memory in a uniformly distributed manner. If the local allocation policy is used, explicit distribution enables maximum bandwidth usage regardless of memory access patterns as long as inter-socket communication is kept to a minimum.

To be complete, the existing techniques introduced in the previous sections are shortly categorised as well. Sequential strategies include cache-oblivious reordering of nonzeroes by use of the Hilbert curve (*CO-H*) or the Morton curve (*CO-M*). Inducing and exploiting an SBD form of A results in a cache-oblivious method as well (*CO-SBD*). Cache-aware techniques often are based on CRS and various proven strategies are implemented in the OSKI package [34]. The shared-memory method by Martone et al. [20] processes blocks of A in Morton order, and nonzeroes within blocks in CRS order. Its distribution strategy is implicit 1D and parallelisation is coarse-grained. CSB [8] uses the Morton curve within blocks instead, employs an implicit 1D distribution, and fine-grained parallelisation. Fully distributed SpMV kernels employ any sequential strategy for local SpMV multiplication, and use explicit 1D or 2D partitioning. These existing shared-memory SpMV strategies as well as the new parallel methods introduced later, rely heavily on sparse blocking. This places some demands on the matrix storage; this is discussed first.

3.1 Storage methods for blocked matrices

Blocking helps to increase cache-efficiency, since accesses to x and y are guaranteed to be local within a range proportional to the block size. CSB, for instance, stores mn/β^2 blocks A_{ij} of size $\beta \times \beta$. Each block would take $\Theta(2nz(A_{ij})+\beta)$ of storage if CRS were used; and summing over all blocks would yield a total storage requirement of $\Theta(2nz + \frac{n}{\beta}m)$. Thus matrix storage does not scale in matrix dimensions, and a parallel SpMV multiplication based on such a storage would not be competitive due to increased bandwidth requirements. Note that this discourages the use of the cache-aware CRS-based optimisation techniques within blocks. CSB solves this problem by using COO to store the (Morton-ordered) nonzeroes within each block, so that storage is independent of the block size. Furthermore, compression is applied by noting that the nonzero indices within blocks are always less than β, so the number of bits required to store the indices is $2\lceil \log_2 \beta \rceil$. These indices are stored in arrays of size nz, while an array of size mn/β^2 tracks which indices in those arrays correspond to which matrix block A_{ij}. For well-chosen β, the total CSB storage requirement is asymptotically equal to that of CRS on the whole of A [8].

This kind of compression can also be applied on other storage schemes by identifying the maximum absolute values either in nonzero index arrays, or in index increment arrays. E.g., for (B)ICRS, if $\max_{\Delta J}$ is the largest absolute value in ΔJ_k, excluding ΔJ_0, then ΔJ can be stored using $\Theta(\log_2 n + nz \log_2 \max_{\Delta J})$ bits instead of $\Theta(nz \log_2 n)$. The maximum values can be determined at runtime so to auto-tune the storage scheme used. Applying this on the ΔI and ΔJ arrays of BICRS yields the Compressed BICRS (CBICRS) scheme, which can be used to replace compressed COO within CSB storage. The worst-case

asymptotic memory usage then is equal to that of CSB, but since negative increments are allowed, the number of bits used in storage increases with one.

For orderings on the block level that are not row major, the starting indices of the various blocks cannot be automatically inferred. Extra indexing arrays are required; freedom in the ordering of blocks thus comes with the price of additional storage requirements. Martone et al. use a quadtree for this purpose which requires loglinear storage in the number of blocks, and handles recursively defined orderings only. Using block-level BICRS instead requires $\Theta(b + M_j)$ storage and enables the use of arbitrary orderings. Here, b is the total number of blocks and M_j is the number of row-wise jumps on the block level. Since $M_j \leq b$ the storage overhead is linear in b. Each element of the V array in block-level BICRS corresponds to a single matrix block; the block-level BICRS SpMV kernel thus remains as in Algorithm 1, but line 4 is replaced by a call to the corresponding SpMV kernel on the input vector x with offset j, and on the output vector y with offset i. Applying BICRS on the nonzero level as well, a per-block compressed scheme is obtained with a worst-case memory requirement of

$$\mathcal{O}\left((8nz + b\log_2(mn) + (nz + m_j - b)\log_2 c\right), \tag{1}$$

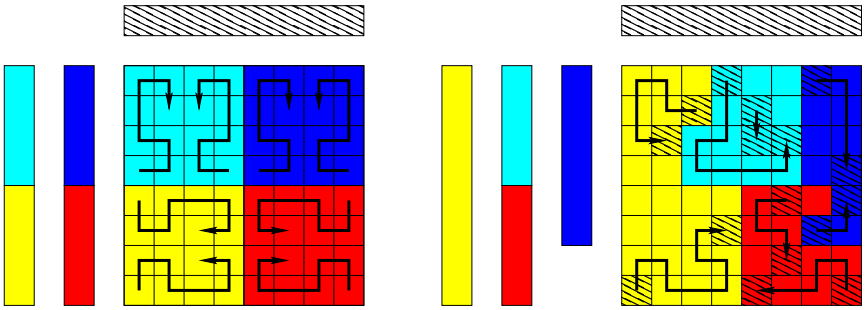
where c depends on the maximum values stored in the index increment arrays. This analysis also holds if ICRS is used on the nonzero level.

3.2 No vector distribution

In this section, the Hilbert scheme introduced in Section 2.1 is extended to a parallel scheme which cache-obliviously accounts for pairwise shared caches. Assuming the nonzeroes of A are sorted in ascending Hilbert order, then a simple partitioning scheme is to distribute the $(k+1)$th nonzero to process $\lfloor k \cdot p/nz \rfloor$. This yields a perfect load-balance, and is equivalent to partitioning $K = \{0, \ldots, nz-1\}$ into p equally-sized parts K^s, where each part K^s holds only consecutive values; a nonzero with (Hilbert-ordered) index k then is distributed to s precisely when $k \in K^s$. K^s corresponds to a subset of nonzeroes still in Hilbert-order, which can be locally exploited during SpMV multiplication.

This strategy does not take into account the location of nonzeroes within A, and so the row ranges and column ranges between the p processes will largely overlap. The case where p is a power of two and A has a uniform nonzero density, for instance, leads to the situation where each input and output vector element is accessed by $\Theta(\sqrt{p})$ processes during a single SpMV multiplication. This poses no problem on the input vector, but concurrent writes on output vector elements must be avoided. For this reason, each core allocates and writes to its own local output vector. After local multiplication, a post-processing step combines all local results. Figure 2 illustrates the case for $p = 4$. The local versions of the output vectors are therein illustrated on the left, and the global input vector on the top. Post-processing requires a summation of $\mathcal{O}(pm)$ elements. Vector storage similarly takes an $\mathcal{O}((p-1)m)$ overhead. Figure 2b shows a non-uniform distribution of nonzeroes and corresponding overlap of output vectors.

Adaptation of the Hilbert-sorting further improves cache-efficiency in case of shared caches. For $p = 2$, each processor would normally follow the Hilbert order in an ascending fashion. If process 0 processes nonzeroes in K^0 in *descending*



(a) Uniform nonzero distribution (b) Non-uniform nonzero distribution

Figure 2: Illustration of the non-distributing block CO-H+ strategys. Different colours indicate locality to different processes. Striped sections indicate global data; striped matrix entries indicate regions containing nonzeroes.

order, however, the scheme would also exploit locality of any shared caches available, at no expense to the efficiency on private caches.

For larger p, $p-2$ parts have two of such interfaces with neighbouring parts where locality may be exploited. To do this, the SpMV multiplication should interleave the ascending and descending Hilbert orders; i.e., process s first handles $\min K^s$, the minimum value from K^s, followed by $\max K^s$, then the second lowest value $\min K^s + 1$, then $\max K^s - 1$, and so on. The arrows within each processor-local block in Figure 2a illustrate this adapted Hilbert ordering (*CO-H+*). Note that the cyan process now has shared locality with both the yellow and the blue process thanks to alternating ascending/descending ordering; processes now exploit locality with all neighbouring processes during SpMV multiplication, while retaining cache-oblivious locality on any private caches. This works independent of the nonzero distribution, see Figure 2b. The effectiveness of the scheme does depend on the nonzero structure of A, however, just as for the sequential CO-H strategy [41].

Storage of CO-H+ using (Compressed) BICRS requires that the location of nonzero $\max K^s$ is known explicitly, causing a $\Theta(p)$ storage overhead. The (C)BICRS multiplication kernel then only needs a small adaptation as index increments can be applied in reverse. Although this scheme can be applied on the level of nonzeroes using (C)BICRS storage, blocked storage has the advantage of a reduced number of row jumps if nonzeroes within blocks are processed in CRS order. As discussed previously, CRS itself is unusable since its storage depends linearly on the block dimension; ICRS avoids this and also allows for compression if the blocking sizes are fixed beforehand.

The resulting strategy is referred to as *block CO-H+*. To optimise access of the input vector, x is allocated in an interleaved fashion since all cores can request any element from x. The p local output vectors, as well as the local parts of A, are locally allocated. Storage overhead is of $\Theta(p+(p-1)m)$, and the local perfectly load-balanced SpMVs are followed by a single global synchronisation step, after which the parallel post-processing step takes $\Theta(m)$ time. Thus the memory storage and the volume of data movement during post-processing do not scale for this method; for small p, however, this overhead may be hidden by its increased cache-efficiency.

3.3 One-dimensional distribution

Both CSB and the parallel shared-memory strategy by Martone et al. use an implicit 1D distribution, employ blocking, and make use of the Morton curve. Since the Hilbert curve has better overall locality properties, a corresponding change of ordering might yield an improvement in cache efficiency for both strategies. Improving locality within (small) blocks is not expected to have a tremendous impact, so Hilbert-ordered CSB is not investigated here. Employing the Hilbert order on the block-level should have a noticeable effect, however. Section 3.1 suggests to store such a scheme using block-level BICRS, with ICRS on the nonzero level.

First the matrix blocks A_{ij} are distributed in a coarse-grained fashion: consecutive ranges of block rows are distributed to processors such that each processor roughly hold an equal number of nonzeroes. Each process thus writes to a unique section of the output vector during multiplication, but read from the entire input vector. After row-wise distribution, each processor independently applies the Hilbert order on the A_{ij} distributed to it, whereas processing of nonzeroes within individual blocks happens in CRS order. This scheme inherently exploits shared caches, since processes use the same Hilbert curve, and cores process elements from x corresponding to the start of the curve first. Again, this sharing is oblivious to the exact cache layout, but also is oblivious to the structure of A; its effectiveness depends on the matrix structure.

Since cores are guaranteed to write to only a subset of y, an explicit distribution is used to guarantee performance on NUMA architectures. Interleaved allocation is used for the input vector. The resulting new *row-distributed block CO-H* strategy has the storage requirement given in Equation 1, the total of which is less than that of CRS storage of the full matrix A for appropriate block size β, as with CSB. Thanks to the explicit 1D distribution, local multiplication does not incur any overhead other than the inter-process data movements on elements from x; there is no redundancy in vector storage, and no synchronisation is required.

A question is whether CSB can benefit from an explicit distribution. Because of the fine-grained parallelism employed, it cannot be determined beforehand which core works on which part of the output vector, so it is not possible to allocate the y^s on the correct core. To nonetheless make CSB efficient on NUMA systems, interleaved memory allocation of all data structures is used, so that the bandwidth load may be distributed over all available memory controllers. Whether the memory access pattern then indeed is uniform depends on the task scheduler, and its interplay with the nonzero structure of A; explicit coarse-grained distributions avoid these issues.

3.4 Two-dimensional distribution

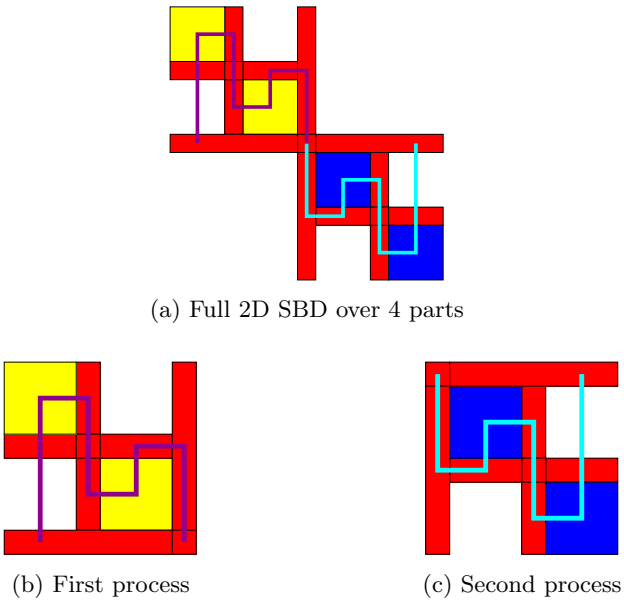
Full distribution is necessary on distributed-memory architectures, but can be applied on shared-memory architectures as well. The question is whether that leads to improvements in cache efficiency, and whether the required overhead remains within acceptable limits. A distinct advantage of explicit 2D partitioning, is that all local data $(A^{(s)}, x^s, y^s)$ can be locally allocated, thus completely avoiding the use of interleaved allocation on NUMA systems. In this section the strategy presented in Algorithm 2 is augmented using the CO-SBD reorder-

ing strategy, which will reduce the bandwidth requirements of general explicitly 2D-distributed strategies. Both these methods were introduced in Section 2.

Consider first the case where A is partitioned into p parts, such that $A^{(s)}$ is the $m_s \times n_s$ matrix local to process s. If the global A is permuted into 2D SBD form, the local matrices look like the global permuted A, but with $p-1$ pure blocks missing as those are distributed to other processes. Also, the $p-1$ separator crosses are less dense as only the nonzeroes distributed to s are stored there. According to this, $A^{(s)}$ is first split in two: nonzeroes $a_{ij}^{(s)}$ are contained in $L^{(s)}$ when it resides in a pure block (i.e., if $\pi_y^s(i) = \pi_x^s(j) = s$), and in $S^{(s)}$ otherwise. When processing $L^{(s)}$ no communication with other processes is required; communication only occurs on the rows and columns where $\pi_y^s(i)$ or $\pi_x^s(j)$ does not equal s. Let $n_s^- \leq n_s^+ \leq n_s$ be such that $\pi_x^s(j) \neq s$, for all $j \in \{0, \ldots, n_s^-\} \cup \{n_s^+, \ldots, n_s - 1\}$, as per the 2D SBD permutation these are the only two local column-ranges where $S^{(s)}$ contains nonzeroes. This means that during SpMV multiplication, it is not necessary to store the full range $\{0, \ldots, n_s - 1\}$ of the maps π_x^s and μ_y^s. Instead, only storage of $n_s^- + (n_s - n_s^+)$ elements is required. This reduces the memory usage of the fan-out maps from $2\sum_s n_s = o(n)$ to $2\sum_s (n_s^- + n_s - n_s^+)$. The reduced storage can be expressed in terms of the $(\lambda - 1)$ metric, since each tuple (s, j) stored in π_x^s and μ_x^s corresponds to a single fan-out message; the total storage requirement is $\Theta(2\sum_{i:n_i \in \mathcal{N}^{\text{col}}} (\lambda_i - 1))$.

The same strategy can be applied for the fan-in of Algorithm 2 as well, but the use of sends and receives comes at an additional storage penalty: all communication must be buffered to prevent data races. A buffer for fan-in messages stores tuples (i, v), with i the local index of where to add the remote contribution v. The total storage space required for this is equal to the total fan-in communication done, which brings the required fan-in storage of the maps and buffers combined to $\Theta(4\sum_{i:n_i \in \mathcal{N}^{\text{row}}} (\lambda_i - 1))$. Reversing the communication scheme, i.e., let each process retrieve remote contributions instead of them delivering contributions to a local buffer, avoids the need for buffering. The fan-in step then requires a triplet (r, k, i) for each remote contribution instead, where r is the remote process, k the remote index, and i the local index, so that process s knows that adding y_k^r to y_i^s is part of the fan-in step. Storage of these triplets is now $\Theta(3\sum_{i:n_i \in \mathcal{N}^{\text{row}}} (\lambda_i - 1))$, reduced, but still more expensive than the SBD-aware fan-out scheme presented earlier.

Each of the p pure local matrices $L^{(s)}$ are partitioned into q smaller parts, which are then locally also permuted into SBD form; i.e., $L^{(s)}$ is permuted into an SBD form with q parts. A parallel SBD-aware SpMV multiplication scheme can then proceed in four steps. First, processes request non-local elements required for local multiplication from remote processes (fan-out), as described above. Then, the p processes perform a local sequential CO-SBD SpMV multiplication on $L^{(s)}$ using its q local parts and the $q-1$ local internal separator crosses. The third step accounts for the remote multiplication on the remaining $p-1$ separator crosses; i.e., the multiplication with $S^{(s)}$ for which communication is required. Step four, the fan-in step, finally gathers all non-local contributions to the local output vector. Algorithm 3 details this approach. Note that both SBD reordering and distributed-memory SpMV multiplication share explicit matrix partitioning as a pre-processing step, and that the permutation of the p matrices $L^{(s)}$ can be done simultaneously with the top-level partitioning; that is, immediately partition A into $q \cdot p$ parts. Here, p is still the



(a) Full 2D SBD over 4 parts

(b) First process

(c) Second process

Figure 3: An SBD matrix distributed over two processes. Global view (top) and local process views (bottom).

number of cores, and q refers to the number of parts that will map to a single process. The added cost of this extra partitioning is logarithmic in q for hypergraph partitioners based on recursive bipartitioning, such as Mondriaan [3]. Figure 3 illustrates this cache-oblivious parallel strategy.

The storage requirement of the matrices $L^{(s)}$ and $S^{(s)}$ is $\Theta(2nz + b + m_j)$, with m_j the total number of row- or column-jumps within each SBD block, as derived in Section 3.1 from BICRS-based storage of the blocks and ICRS/ICCS storage of nonzeroes within blocks. Since SBD blocks are not of fixed size, there is no guarantee in this case that compression would lower this bound to equal that of CRS storage. The need for communication indicates that there is overlap between the p local input and output vectors. The total number of elements stored equals the sum of the total connectivity, while the minimum amount is $m + n$, which by definition equals the number of nets in the hypergraph. Hence the overlap is exactly given by the $(\lambda - 1)$-metric. All combined overhead in this 2D SBD strategy thus is $\Theta(b + 3 \cdot \sum_{i:n_i \in \mathcal{N}} (\lambda_i - 1) + \sum_{i:n_i \in \mathcal{N}^{\text{row}}} (\lambda_i - 1))$. Note that b here is linear in $q \cdot p$, and that the $(\lambda - 1)$ metrics only take the p top-level vertex parts into account to determine inter-process connectivity.

4 Experiments

Experiments are set up as follows. The execution time of a sequential CRS multiplication is used as the base sequential time t_s used to measure the speedup t_s/t_p of one of the parallel methods introduced in the previous section. Here, t_p is the time required for a single parallel multiplication. An overview of the methods experimented with is found in Table 1. There, OpenMP CRS refers to an implicit block-row distributed 1D scheme based on plain CRS, enabled

Algorithm 3 SBD-ordered 2D parallel multiplication.

Each processor $s \in \{0, 1, \ldots, p-1\}$ executes:

1: $o = c^s_+ - c^s_-$
2: **for** $j = 0$ to c^s_- **do**
3: set $x^s_j = x^{\pi^s_x(j)}_{\mu_x(j)}$
4: **for** $j = c^s_+$ to n^s **do**
5: set $x^s_j = x^{\pi^s_x(j-o)}_{\mu_x(j-o)}$
6: calculate $y^s = L^{(s)}x^s$ (CO-SBD)
7: calculate $y^s = S^{(s)}x^s$ (CBICRS)
8: synchronise and add remote contributions to y^s

by adding the openMP pragma "omp parallel for schedule(dynamic,8)" before the outer loop of the sequential CRS SpMV kernel (see page 2); this is a very naive parallelisation and the (lack of) performance of this strategy in no way reflects on the general usefulness of openMP itself. To the contrary, as per the 'dynamic' openMP schedule, this compact notation already elegantly enhances the load-balance of the strategy by using a fine-grained row-distribution, and assigns rows in contiguous chunks of minimum size 8 so to avoid false sharing on the output vector. This strategy can be taken as a baseline for parallel SpMV kernels.

Since performance depends on both hardware specifics as well as input matrix properties, experiments are performed on various matrices and various architectures. Table 2 lists all matrices used in experiments, and Table 3 all architectures. The matrices originate from a wide range of applications: FEM modelling on regular and adaptive meshes, linear programming, chip industry, chemistry, combinatorics, term-by-document indexing, and web crawls. Matrices can be subdivided into structured matrices (such as a five-diagonal matrix), and unstructured matrices (such as a matrix representation of the United States road network). Structured matrices possess a nonzero pattern that already induces a favourable cache reuse pattern; generally, cache-oblivious methods do not speed up multiplications on such matrices [13, 26, 40]. Lacking any favourable structure, unstructured matrices cause an inefficient cache reuse pattern during regular multiplication; here the beneficial effects of cache-oblivious reordering or permutations are visible most.

Method	Characteristics			Reference
OpenMP CRS	implicit	1D	-	Sec. 4
Compressed Sparse Blocks	implicit	1D	Oblv.	[8]
Interleaved CSB	implicit	1D	Oblv.	Sec. 3.3
pOSKI	explicit	1D	Aware	[2]
Row-distributed block CO-H	explicit	1D	Oblv.	Sec. 3.3
Fully distributed	explicit	2D	-	Sec. 3.4
Distr. CO-SBD	explicit	2D	Oblv.	Sec. 3.4
Block CO-H+	implicit	ND	Oblv.	Sec. 3.2

Table 1: Strategies used in experiments.

Structured matrices	Rows	Columns	Nonzeroes
fidap037	3 565	3 565	67 591
cavity17	4 562	4 562	131 735
s3rmt3m3	5 357	5 357	207 123
bcsstk17	10 974	10 974	428 650
memplus	17 758	17 758	99 147
lhr34	35 152	35 152	746 972
bcsstk32	44 609	44 609	1 029 655
s3dkt3m2	90 449	90 449	1 921 955
cage14	1 505 785	1 505 785	27 130 349
Freescale1	3 428 755	3 428 755	17 052 626
cage15	5 154 859	5 154 859	99 199 551
adaptive	6 815 744	6 815 744	13 624 320
Unstructured matrices			
rhpentium_new	25 187	25 187	258 265
tbdlinux	112 757	20 167	2 157 675
nug30	52 260	379 350	1 567 800
stanford	281 903	281 903	2 312 497
stanford_berkeley	683 446	683 446	7 583 376
ldoor	952 203	952 203	23 737 339
wikipedia-2005	1 634 989	1 634 989	19 753 078
GL7d18	1 955 309	1 548 650	35 590 540
wikipedia-2006	3 148 440	3 148 440	39 383 235
wikipedia-2007	3 566 907	3 566 907	45 030 389
road_usa	23 947 347	23 947 347	28 854 312

Table 2: Matrices used in experiments. For both structured and unstructured matrices, the datasets are divided by horizontal separators into small matrices (top) and large matrices (bottom). The matrices in the middle are considered large on some machines and small on others, depending on the exact L3 cache size (see Table 3).

Efficiency of the methods also highly depends on architecture specifics, hence experiments are repeated on the three different NUMA machines. Each consists of several multicore processors placed in different sockets. The first machine, the DL-2000, has two six-core Intel Xeon X5660 processors; the second machine, the DL-580, contains four ten-core Intel Xeon E7-4870 processors; finally, the DL-980 has eight eight-core Intel Xeon E7-2830 processors. All processors allocate private 32kB L1 and 256kB L2 caches to each core. L3 caches are shared amongst all cores on a processor, the size of which varies across the machines; see Table 3.

If a matrix is small enough so that its input and output vector fit into the L3 cache, no unnecessary data movement between main memory and processors occurs. This is quite different from the general case where the vectors do not fit into cache. To differentiate between these cases, the dataset is further subdivided into small and large matrices, a classification which differs for each machine experimented with.

Name	Specifications	
DL-2000	2 socket × 6 cores at 2.67 GHz,	12MB L3 cache
DL-580	4 socket × 10 cores at 2.4 GHz,	30MB L3 cache
DL-980	8 socket × 8 cores at 2.13 GHz,	24MB L3 cache

Table 3: Machines used in experiments.

For the DL2000, which has an L3 cache of 12MB, a total vector length of $m + n \leq \frac{12}{8} \cdot 2^{20} = 1\ 572\ 864$ would cause its (double-precision) input and output vectors to fit into cache; hence from the structured matrix cage14 and the unstructured matrix ldoor on, matrices are considered large. The DL580 has a 30MB cache, however, hence the maximum total matrix dimension becomes 3 932 160: only the matrices from Freescale1 (structured) and wikipedia-2006 (unstructured) on are considered large, while the others remain considered small, for this architecture. For the DL980, the structured large matrices start at the Freescale1 matrix as well, but the large unstructured category starts at the wikipedia-2005 matrix. These differences in categorisation are taken into account for the results presented later in this section.

An SpMV benchmark is run for each dataset, method, architecture, and varying numbers of threads. This benchmark consists of one thousand SpMV multiplications on random input vectors. The time taken is measured and averaged, and the experiment is repeated an additional 9 times so a sample variance can be constructed as well. If an experiment fails[2], the result is excluded from counting towards any of the aggregates presented in Section 4.1.

The SpMV benchmarking utilities for the strategies introduced in this paper are written in C++ using POSIX threads, are compiled using GCC version 4.3.3, and are freely available[3]. All machines used in experiments run a Linux operating system. External software used in experiments include CSB [8] and pOSKI [2], as well as the Mondriaan sparse matrix partitioner software [4]. CSB was compiled using the Cilk Arts compiler (build 8503, based on GCC 4.2.4), the other software using GCC version 4.3.3. pOSKI is used such that input matrices are copied into a custom pOSKI format and the timing of its SpMV kernel is preceded by an aggressive tuning step, so that any of the available optimisations are attempted, without trying to amortise the cost of this auto-tuning. The average time is measured over one thousand multiplications, and no hints towards input matrix structures are given. The Mondriaan code is slightly adapted from the current 3.11 version available online. It is used with the Mondriaan scheme for partitioning and SBD reordering enabled, and default parameters otherwise. The allowed load-imbalance percentage is set to $\epsilon = 0.3$.

4.1 Results

Tables 4–7 display the best attained average speedup on each architecture, per matrix category. An average of these values is given in Table 8. The number

[2]The causes of which include, amongst others, not being able to find a proper load-balanced distribution and not finding a proper SBD reordering. These have occurred sporadically for any of the parallel methods, typically for high p and small matrices. In case of pOSKI the issues were non-deterministic and negatively influenced aggregate results, so some experiments were re-run until a (verified) timing was found.

[3]See `http://albert-jan.yzelman.net/software`.

Methods	DL-2000	DL-580	DL-980
OpenMP CRS (1D)	1.6 (5)	1.7 (4)	1.6 (4)
CSB (1D)	5.8 (12)	11.2 (40)	7.8 (32)
Interleaved CSB (1D)	6.0 (12)	12.6 (40)	8.7 (48)
pOSKI (1D)	4.7 (12)	10.9 (20)	8.4 (16)
Row-distr. block CO-H (1D)	10.1 (12)	36.1 (40)	52.4 (64)
Fully distributed (2D)	3.1 (12)	2.3 (8)	2.1 (8)
Distr. CO-SBD, $qp = 32$ (2D)	3.1 (8)	2.5 (8)	2.3 (8)
Distr. CO-SBD, $qp = 64$ (2D)	3.6 (8)	3.1 (8)	2.8 (8)
Block CO-H+ (ND)	1.4 (4)	1.6 (5)	1.6 (4)

Table 4: Best average speedups for small structured matrices on the various machines. The number of threads this maximum occurs on, is noted in parenthesis.

Methods	DL-2000	DL-580	DL-980
OpenMP CRS (1D)	2.1 (8)	2.9 (8)	2.0 (8)
CSB (1D)	4.4 (12)	12.3 (40)	8.9 (32)
Interleaved CSB (1D)	4.8 (12)	17.9 (40)	13.8 (64)
pOSKI (1D)	6.3 (12)	20.3 (40)	15.5 (48)
Row-distr. block CO-H (1D)	8.7 (12)	32.6 (40)	49.0 (56)
Fully distributed (2D)	4.4 (12)	9.2 (40)	6.1 (16)
Distr. CO-SBD, $qp = 32$ (2D)	4.1 (8)	6.6 (16)	5.6 (16)
Distr. CO-SBD, $qp = 64$ (2D)	4.4 (8)	7.8 (16)	9.0 (16)
Block CO-H+ (ND)	3.2 (12)	4.0 (16)	3.7 (10)

Table 5: As Table 4, but for small unstructured matrices.

of cores with which this maximum is attained is added in parenthesis. When this number is lower than the total amount of cores a machine has available, it generally means the method slows down when more threads are used; i.e., the method does not scale. Examples of such methods are non-distributed block CO-H+, locally allocated CSB (on highly NUMA systems), and openMP CRS. For other methods, the execution time may stall as the number of threads increase, but does not increase.

The results show that for large p, interleaved CSB performs better than locally allocated CSB, a difference which manifests itself most clearly on the larger matrices. For smaller p, although not shown in the tables, the normal CSB strategy often is faster, but significant differences between the two were only observed on the DL-980 machine.

The non-distributing block CO-H+ strategy performs best on unstructured matrices, preferably small ones, using a limited number of threads. In that category it performs best for $p = 2$, and outperforms all methods on all architectures, except for the row-distributed block CO-H, explicit 2D with $qp = 64$, and pOSKI methods. On the DL-980 it outperforms pOSKI for $p = 2$, and performs on par with the other methods for slightly higher p $(4, 5)$ as well.

On all small matrices, however, no method performs better than the row-distributed block CO-H strategy. Its maximum speedups are quite close to the actual number of cores, especially for the structured small matrices. For

Methods	DL-2000	DL-580	DL-980
OpenMP CRS (1D)	2.5 (8)	3.4 (8)	4.1 (10)
CSB (1D)	3.8 (12)	8.6 (30)	8.4 (48)
Interleaved CSB (1D)	5.4 (12)	18.0 (40)	22.0 (64)
pOSKI (1D)	3.9 (12)	12.2 (40)	11.9 (64)
Row-distr. block CO-H (1D)	3.8 (12)	10.9 (40)	16.4 (64)
Fully distributed (2D)	3.5 (12)	10.6 (30)	10.2 (40)
Distr. CO-SBD, $qp = 32$ (2D)	3.2 (8)	9.6 (32)	9.6 (32)
Distr. CO-SBD, $qp = 64$ (2D)	3.6 (8)	7.8 (32)	8.8 (64)
Block CO-H+ (ND)	2.2 (12)	3.3 (24)	3.3 (16)

Table 6: As Table 4, but for large structured matrices.

Methods	DL-2000	DL-580	DL-980
OpenMP CRS (1D)	4.6 (8)	6.8 (8)	6.2 (8)
CSB (1D)	5.8 (10)	10.8 (16)	10.7 (64)
Interleaved CSB (1D)	7.9 (12)	24.3 (40)	26.3 (64)
pOSKI (1D)	6.6 (12)	19.4 (40)	16.2 (64)
Row-distr. block CO-H (1D)	5.4 (12)	19.2 (40)	24.6 (64)
Fully distributed (2D)	5.1 (10)	13.6 (40)	12.3 (64)
Distr. CO-SBD, $qp = 32$ (2D)	5.3 (8)	12.0 (32)	10.1 (32)
Distr. CO-SBD, $qp = 64$ (2D)	5.4 (8)	11.7 (32)	12.0 (64)
Block CO-H+ (ND)	3.7 (12)	4.4 (40)	4.8 (16)

Table 7: As Table 4, but for large unstructured matrices.

large matrices, interleaved CSB usually performs best, although on unstructured matrices and large p there is but a slight difference between the row-distributed block CO-H and the CSB strategy. Both methods perform better as more threads are added, with two exceptions on small structured matrices for $p = 56$ and $p = 64$ for interleaved CSB on the DL-980, and another two exceptions on large structured matrices with $p = 10$ and $p = 20$ for row-distributed block CO-H on the DL-580. All other methods experimented with do not share this favourable property, and performance may (momentarily) decrease when more threads are employed.

The 2D strategies seem to stall for larger p, and are completely uncompetitive for small matrices and high p, with the exception of small unstructured matrices on the smaller DL-2000 machine. On large machines and high p it performs on-par with locally allocated CSB. The best relative performance is attained on large unstructured matrices for $p = 2$, where it outperforms all other methods on all three architectures. This indicates that the overhead with increasing p presently is larger than the reduction in inter-core communication. Although not reported in this paper, the synchronising 2D SpMV strategy as described in the thesis of Yzelman [38, Chapter 5] is in all cases outperformed by the explicit 2D method introduced in this paper.

Table 8 presents the averages of the speedups reported in Tables 4–7. There, pOSKI is placed just behind CSB and the row-distributed block CO-H methods, well ahead in performance of the other remaining methods. It performs best on unstructured matrices, where it is faster than either CSB or row-distributed

Methods	DL-2000	DL-580	DL-980
OpenMP CRS (1D)	2.6 (8)	3.5 (8)	3.0 (8)
CSB (1D)	4.9 (12)	9.9 (30)	8.6 (32)
Interleaved CSB (1D)	6.0 (12)	18.2 (40)	17.7 (64)
pOSKI (1D)	5.4 (12)	14.8 (40)	12.2 (64)
Row-distr. block CO-H (1D)	7.0 (12)	24.7 (40)	34.0 (64)
Fully distributed (2D)	4.0 (12)	8.3 (40)	6.9 (32)
Distr. CO-SBD, $qp = 32$ (2D)	4.0 (8)	7.4 (32)	6.4 (32)
Distr. CO-SBD, $qp = 64$ (2D)	4.2 (8)	7.0 (16)	6.8 (64)
Block CO-H+ (ND)	2.5 (12)	3.1 (16)	3.2 (16)

Table 8: Averaged speedups from Table 4–7. The number of threads the maximum occurs on, is noted in parenthesis.

block CO-H, on small matrices, resp., large matrices. It also performs well on small matrices using a small number of cores ($p < 4$). By the same table, the row-distributed block CO-H strategy emerges as the best-performing strategy with almost a double average speedup compared to the second-fastest, interleaved CSB. This 1D Hilbert-based strategy furthermore consistently attains its best performance when using all available cores.

4.2 Results on parallel 2D cache-oblivious SBD

To investigate the effect of partitioning for qp parts, with $q > 1$, so to apply CO-SBD locally on each process, partitioning for a high number qp is necessary. Current software limits q, p to powers of two for $q > 1$, which, in turn, limits the choice of machines where this method can be investigated in detail on to the DL-980. The highest number of parts qp partitioned for is 512, and this was successful only on the cage15 matrix and adaptive matrix. These combinations enabled experiments for $q = \{1, 4, 8, 16, 32, 64\}$ and $p = \{8, 16, 32, 64\}$, with $pq \in \{8, 16, 32, 64, 256, 512\}$. Being able to use the full number of cores available while still able to increase the number of local refinements, yields a deeper insight in the effectiveness of the explicit 2D CO-SBD strategy for increasing q and p.

Table 9 displays the results of these experiments. The reported speedups are again relative to sequential CRS. Cage15 is a matrix which is proven to be hard to partition [40, 25], while partitioning is quite successful for the adaptive matrix. The results reflect this; speedups for the cage15 matrix are low, while much better speedups are attained for the adaptive matrix. Local reordering has an adverse effect for $p < 64$ on the cage15 matrix, while for $p = 64$ it results in significant speedups. This behaviour is visible for the adaptive matrix as well, but then for $p = 32$; in comparison, local reordering for $p = 64$ results in very modest speedups only.

5 Conclusions

In this paper, several possibilities for the parallelisation of the sparse matrix–vector multiplication were categorised, mainly on basis of their implicit or explicit matrix distribution scheme: either non-distributing, one-dimensional, or

	cage15				adaptive			
$q\backslash p$	8	16	32	64	8	16	32	64
1	3.0	3.9	5.7	4.8	6.0	11.1	11.7	15.3
4	2.5	3.5	–	5.5	6.4	9.7	–	15.8
8	2.5	–	5.9	6.7	5.8	–	14.8	16.3
16	–	3.7	5.3	–	–	10.9	17.5	–
32	2.2	3.4	–	–	5.5	10.9	–	–
64	2.0	–	–	–	6.0	–	–	–
Interl. CSB	3.2	6.0	10.6	14.6	8.4	15.9	28.5	38.1
1D block CO-H	5.2	6.9	9.2	14.7	4.4	6.5	10.7	18.8

Table 9: Speedups of the 2D SBD-CO strategy for various combinations p, q, on the DL-980. Performance of the two leading strategies are included for reference.

two-dimensional. Earlier work on parallel SpMV was discussed and classified. Some of these existing methods were improved, while some new methods were introduced as well.

Although theoretically only the 2D method scales in terms of inter-process (and, more importantly, inter-socket) data movement, the experiments show that current architectures posses a sufficiently fast interconnect for 1D distributions to be sufficient; i.e., there currently is a favourable balance between the number of cores per socket, the available bandwidth per socket, and the available bandwidth in-between sockets. More specifically, the overhead of using a 2D distribution is costlier than the extra data movement a 1D distribution incurs on the input vector. Both of these costs increase with p, but the 2D overhead is bounded by the $(\lambda - 1)$-metric, whereas the 1D data movement is relatively unbounded by $\mathcal{O}(pn)$. Indeed this becomes an issue for more highly-NUMA systems; comparing the results between the DL-580 and the DL-980, the 64-core system in fact displays a lower average speedup than the 40-core DL-580, for all implicitly distributed methods. If future systems move towards deeper and deeper NUMA hierarchies, these issues will deepen as well, so explicitly distributed (and perhaps 2D) strategies, will become the only viable choice.

The best-performing 1D method is the row-distributed block CO-H strategy, attaining good speedups especially for the small structured matrices, where a 36 times speedup is attained on the 40-core DL-580 machine. The runner-up is CSB using an interleaved memory allocation scheme, which outperforms the previous scheme on the larger matrices, but does that convincingly only on the large structured matrices. It is interesting to note that using an interleaved allocation in fact induces a group-cyclic distribution on the vectors; a simple 2D distribution hence already gives an advantage over a purely 1D method, especially if it does not incur any overhead, as is the case here. Still, the strategy itself must still be efficient too, as illustrated by the bad performance of the naive parallelisation done in openMP, where interleaved allocation was also employed.

Indeed the results show that high-level sparse matrix blocking is a necessary optimisation, as both best-performing methods employ this strategy. On

NUMA-architectures at least, these high-level optimisations even out-perform low-level tuning (register blocking, prefetching, densification, and so on), as pOSKI shows a lower speedup than either interleaved CSB and row-distributed block CO-H.

5.1 Future work

Only high-level strategies have been explored in this paper; combinations with low-level optimisation techniques have not been investigated, but is certainly viable. The potential gain is large, as indicated by the huge gap in performance between openMP CRS and pOSKI. To combine the same (auto-tuning) optimisations, however, the backing data structure should be based on (Bi-directional) Incremental CRS instead of normal CRS, so that high-level sparse matrix blocking becomes feasible in terms of memory usage. Such techniques can furthermore carry over to further optimised CSB implementations [7].

CSB performing better on larger matrices is likely due to load-balance issues; while the distribution of nonzeroes may be fair in quantity, some local parts may be slower to process due to an unfavourable local nonzero structure, causing relatively many cache misses on the (local) input and output vectors. Fine-grained parallelisation avoids this issue by defining many more tasks than the actual number of executing cores available, combined with a work-stealing scheduler. A hybrid solution thus poses an immediate contradiction; to optimise for NUMA systems, the coarse-grained explicit strategy should be applied first. But to achieve proper load-balance, the fine-grained strategy should be applied first, as per the above discussion. Finding a middle-ground between these two paradigms seems paramount for future high-performance algorithms, and a solution may be to fit memory affinity in an approach like the 'guided' openMP schedule; there, the grain-size of parallelisation is coarse at the start of the parallel computation, but progressively becomes more fine. The initial coarse-grained distribution will then benefit from data locality, while load-balance is guaranteed by a fine-grained distribution at the last stages of computation. Another possible avenue is to incorporate synchronisation and re-balancing within coarse-grained schemes, but to do this without a large overhead penalty seems very challenging.

Hybrid schemes still are worthwhile to investigate as well. For example, an explicit 2D method could be used at the socket-level, while a simpler scheme can be applied within sockets, yielding the benefit of per-socket data locality for both vectors without the penalty of overhead for the 2D scheme for large p. A secondary enhancement for 2D SpMV multiplication is the possible overlap of communication and computation; the second step of the explicit 2D scheme (local multiplication) can be applied simultaneously with the first step (fan-in).

All-in-all, attaining good performance of the SpMV on highly NUMA systems is still a challenge, and many aspects still are pending further investigation. As shown in the paper, however, for current architectures and those of the immediate future, current state-of-the-art algorithms perform sufficiently well.

Acknowledgements

This work is funded by Intel and by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT).

References

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

[2] Berkeley Benchmarking and Optimization Group. pOSKI: parallel Optimized Sparse Kernel Interface, 2012. `http://bebop.cs.berkeley.edu/poski/index.php`.

[3] R. H. Bisseling, B. O. Fagginger Auer, A. N. Yzelman, T. van Leeuwen, and Ü. V. Çatalyürek. Two-dimensional approaches to sparse matrix partitioning. In U. Naumann and O. Schenk, editors, *Combinatorial Scientific Computing*, pages 321–349. Chapman & Hall/CRC Press, 2012.

[4] Rob H. Bisseling, Bas Fagginger Auer, Tristan van Leeuwen, Wouter Meesen, Brendan Vastenhouw, and A. N. Yzelman. Mondriaan for sparse matrix partitioning, version 3.11, 2010. `http://www.math.uu.nl/people/bisseling/Mondriaan`.

[5] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, August 1995.

[6] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Comput. Netw. ISDN Systems*, volume 30, pages 107–117, 1998.

[7] A. Buluç, S. Williams, L. Oliker, and J. Demmel. Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 721–733. IEEE, 2011.

[8] Aydin Buluç, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 233–244, New York, NY, USA, 2009. ACM.

[9] D. A. Burgess and M. B. Giles. Renumbering unstructured grids to improve the performance of codes on hierarchical memory machines. *Advances in Engineering Software*, 28(3):189–201, 1997.

[10] Ü. V. Çatalyürek and C. Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Trans. Parallel Distrib. Systems*, 10(7):673–693, 1999.

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

[12] 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, Long Beach, CA, 2006.

[13] Gundolf Haase, Manfred Liebmann, and Gernot Plank. A Hilbert-order multiplication scheme for unstructured sparse matrices. *International Journal of Parallel, Emergent and Distributed Systems*, 22(4):213–220, 2007.

[14] M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49:409–436, 1952.

[15] E.-J. Im, K. Yelick, and R. Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *International Journal of High Performance Computing Applications*, 18(1):135–158, 2004.

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

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

[18] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. John Wiley and Sons, Chichester, UK, 1990.

[19] K. Patrick Lorton and David S. Wise. Analyzing block locality in Morton-order and Morton-hybrid matrices. *SIGARCH Comput. Archit. News*, 35(4):6–12, 2007.

[20] Michele Martone, Salvatore Filippone, Salvatore Tucci, Marcin Paprzycki, and Maria Ganzha. Utilizing recursive storage in sparse matrix-vector multiplication - preliminary considerations. In Thomas Philip, editor, *Proceedings of the ISCA 25th International Conference on Computers and Their Applications (CATA)*, pages 300–305, Hawaii, USA, 2010. ISCA.

[21] G. Morton. A computer oriented geodetic data base and a new technique in file sequencing. Technical report, IBM, Ottawa, Canada, March 1966.

[22] C. C. Paige and M. A. Saunders. LSQR: An algorithm for sparse linear equations and sparse least squares. *ACM Transactions on Mathematical Software*, 8:43–71, 1982.

[23] B. N. Parlett, D. Taylor, and Z. Liu. A look-ahead Lanczos algorithm for unsymmetric matrices. *Mathematics of Computation*, 44:105–124, 1985.

[24] F. Pellegrini and J. Roman. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *High-Performance Computing and Networking*, pages 493–498. Springer, 1996.

[25] Daan Pelt. Matrix partitioning: Optimal bipartitioning and heuristic solutions. Master's thesis, Utrecht University, Department of Mathematics, August 2010.

[26] A. Pinar and M. T. Heath. Improving performance of sparse matrix-vector multiplication. In *Proceedings Supercomputing 1999*, page 30. ACM Press, New York, 1999.

[27] Y. Saad and M. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computation*, 7:856–869, 1986.

[28] Gerard L. G. Sleijpen and Henk A. van der Vorst. A Jacobi-Davidson iteration method for linear eigenvalue problems. *SIAM Review*, 42(2):267–293, 2000.

[29] Peter Sonneveld and Martin B. van Gijzen. IDR(s): a family of simple and fast algorithms for solving large nonsymmetric linear systems. *SIAM Journal on Scientific Computing*, 31(2):1035–1062, 2008.

[30] S. Toledo. Improving the memory-system performance of sparse-matrix vector multiplication. *IBM J. Res. Dev.*, 41(6):711–725, 1997.

[31] A. Trifunovic and W. J. Knottenbelt. A parallel algorithm for multilevel k-way hypergraph partitioning. In *Proceedings 3rd International Symposium on Parallel and Distributed Computing*, pages 114–121. IEEE Press, Los Alamitos, CA, 2004.

[32] H. van der Vorst. BiCGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computation*, 13:631–644, 1992.

[33] B. Vastenhouw and R. H. Bisseling. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM Rev.*, 47(1):67–95, 2005.

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

[35] R. Vuduc and H.J. Moon. Fast sparse matrix-vector multiplication by exploiting variable block structure. *High Performance Computing and Communications*, pages 807–816, 2005.

[36] J. B. White, III and P. Sadayappan. On improving the performance of sparse matrix-vector multiplication. In *Proceedings 4th International Conference on High-Performance Computing*, pages 66–71. IEEE Press, Los Alamitos, CA, 1997.

[37] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Computing*, 35(3):178–194, 2009.

[38] A. N. Yzelman. *Fast sparse matrix-vector multiplication by partitioning and reordering*. PhD thesis, Utrecht University, Utrecht, the Netherlands, 2011.

[39] A. N. Yzelman and Rob H. Bisseling. Cache-oblivious sparse matrix–vector multiplication by using sparse matrix partitioning methods. *SIAM Journal on Scientific Computing*, 31(4):3128–3154, 2009.

[40] A. N. Yzelman and Rob H. Bisseling. Two-dimensional cache-oblivious sparse matrix–vector multiplication. *Parallel Computing*, 37(12):806 – 819, 2011.

[41] A. N. Yzelman and Rob H. Bisseling. A cache-oblivious sparse matrix–vector multiplication scheme based on the Hilbert curve. In M. Günther, A. Bartel, M. Brunk, S. Schöps, and M. Striebel, editors, *Progress in Industrial Mathematics at ECMI 2010*, pages 627–634, 2012.

[42] A. N. Yzelman and Rob H. Bisseling. An object-oriented bulk synchronous parallel library for multicore programming. *Concurrency and Computation: Practice and Experience*, 24(5):533–553, 2012.