# Fast sparse matrix–vector multiplication by partitioning and reordering

*Thesis committee:*
Prof. dr. Patrick R. Amestoy, Université de Toulouse
Prof. dr. Fredrik Manne, University of Bergen
Prof. dr. Wil H. A. Schilders, Eindhoven University of Technology
Prof. dr. Sivan Toledo, Tel-Aviv University
Prof. dr. ir. Kees Vuik, Delft University of Technology

ISBN: 978-90-393-5590-9

# Fast sparse matrix–vector multiplication by partitioning and reordering

## Snelle ijle matrix–vectorvermenigvuldiging door partitionering en herordering

(met een samenvatting in het Nederlands)

Proefschrift

ter verkrijging van de graad van doctor aan de Universiteit Utrecht op gezag van de rector magnificus, prof. dr. G. J. van der Zwaan, ingevolge het besluit van het college van promoties in het openbaar te verdedigen op maandag 3 oktober 2011 des middags te 4.15 uur

door

Albert-Jan Nicholas Yzelman

geboren op 17 Juli 1984 te Zevenaar

Promotor: Prof. dr. R. H. Bisseling

# Preface

This thesis is the end result of four years of work, but may never have been written without the support, direct or indirect, from a lot of people. Since this is the case, I feel it is most fitting to acknowledge them first, in this preface.

Foremost I would like to thank my supervisor and my parents. Rob has given me ample opportunity, freedom and inspiration to do research, and I am very grateful for this. I consider myself very lucky to have been under his tutelage.

Wanneer het om de meer dagelijkse dingen des levens ging, hebben mijn ouders mij altijd ondersteund, en geholpen wanneer nodig. Het is in grote mate aan hen te danken dat ik nu kan doen wat ik het liefst doe. Pa, ma, en ook An: hiervoor mijn hartelijke dank. Natuurlijk mogen ook mijn broers en zusje hier niet ontbreken: Orry, Christian en Jennifer, ook jullie zijn onmisbaar. Ik dank ook mijn grootouders, ooms en tantes, en alle verdere familie voor hun steun, maar voornamelijk ook hun gezelligheid.

Apart from my extended family (many of which are from the Netherlands – I hope the reader will forgive me for having momentarily reverted to Dutch), other (often overlapping) groups of people I would like to thank are all my friends, my former colleagues at Utrecht University, my former teachers, as well as fellow researchers and PhD candidates whom I have had the pleasure to meet with. Finally I specifically want to thank Job, who managed to share an office with me for close to four years; our often non-mathematical conversations were particularly useful in keeping stress levels down (for me at least). I also owe a lot to Stijn, who together with Orry agreed to help me through my defence.

Voor de Nederlandstalige niet-wiskundigen die dit boekje in bezit hebben gekregen, heb ik na de Engelstalige wetenschappelijke verhandeling ook een Nederlandse uiteenzetting toegevoegd. Mijn hoop is dat ik ook voor jullie een mooie tekst heb samengesteld.

I hope you will have a pleasant read,

Albert-Jan Yzelman,
August 2011.

# Table of Contents

# Introduction

The sparse matrix–vector (SpMV) multiplication is an important kernel in many scientific computing applications. These include iterative solvers for sparse linear systems, such as CG, GMRES, BiCGstab, and IDR(s) [HS52, SS86, vdV92, SF93, SvG08], and iterative eigensystem solvers, such as the Lanczos [PTL85] or the Jacobi-Davidson [SvdV00] method. The power method also falls in this category, and has seen wide use within the PageRank method [LM06, BP98] and other web-indexing methods [Kle99]. Further examples include the least-squares solver LSQR [PS82], as well as interior point methods for solving optimisation problems; both also make heavy use of SpMV multiplication. Often, however, this important kernel only attains a fraction of peak performance [Tol97, DJ07, VDY05, Im00, WOV$^+$09] on most computer architectures, due to inefficient use of the available system caches. Improving the efficiency of this kernel, which is the main goal of the research presented in this thesis, thus may have a big part in increasing the efficiency of many other applications.

This thesis is divided into two parts, where the first part is on optimising the sequential SpMV multiplication. In a world where parallel machines are increasingly commonplace, however, a fast method should also be parallelisable. Thus the second part combines the insights of fast sequential SpMV multiplication and traditional distributed SpMV multiplication, to describe parallel implementations both for distributed-memory and shared-memory architectures. These two parts will solely focus on the concise description of the developed methods and investigations of their efficiency; this introduction will provide the necessary background on which the methods are built. Chapter 1 has also been published as a journal paper:

[YB09]: 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** (2009), no. 4, 3128–3154.

The other chapters are accepted for publication in various journals:

[YB11c]: A. N. Yzelman and Rob H. Bisseling, *Two-dimensional cache-oblivious sparse matrix–vector multiplication*, Parallel Computing (2011), to appear.

[YB11b]: A. N. Yzelman and Rob H. Bisseling, *An object-oriented BSP library for multicore programming*, Concurrency and Computation: Practice and Experience (2011), to appear.

Content from Chapter 3 is also to appear in the proceedings of the European Conference on Mathematics and Industry (ECMI) 2010 as:

[YB11a]: A. N. Yzelman and Rob H. Bisseling, *A cache-oblivious sparse matrix–vector multiplication scheme based on the Hilbert curve*, Progress in Industrial Mathematics at ECMI, Springer, Berlin, 2011, to appear.

This introduction contains elements from all of these papers. Section 1 describes the problems encountered in sparse matrix–vector multiplication on cache-based architectures. After this follows an introduction to the basic tools used in our solutions: Section 2 recalls how sparse matrices are modelled using hypergraphs, and Section 3 shows how these formulations are used to achieve sparse matrix partitioning. Whenever possible, objects defined in the thesis will be used consistently throughout the thesis. The most important ones are collected in Table 2, at the end of this introduction.

# 1 Cache-based architectures

Many important linear algebra kernels typically take a performance hit on modern cache-based computer architectures due to inefficient cache use [DJ07, IY01, Tol97]. This use is optimal when data from main memory is stored contiguously and is accessed in a single straight pass. Each data item is preferably used many times, and is not needed in further computation afterwards. But many non-trivial applications require to jump through data in main memory, even if data is stored contiguously. The sparse matrix–vector (SpMV) multiplication is a notorious example. In this application, the number of jumps in memory can be reduced in two ways:

1. by adapting the sparse matrix storage scheme, or

2. by adapting the sparse matrix structure.

Most earlier work in this area deals with the first aspect, as does Chapter 3. Chapters 1 and 2 deal with the second aspect. Fortunately, optimising one does not exclude optimising the other: for instance, the matrix reordering method described in Chapter 1 has successfully been combined with the OSKI software [VDY05, YB09].

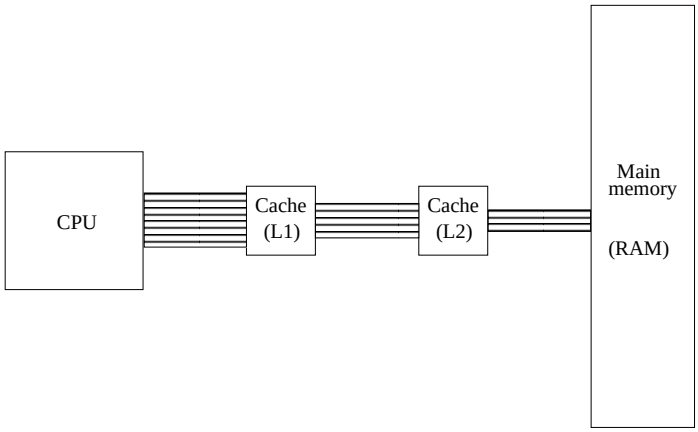



Figure 1: Illustration of a multi-level cache

## 1.1 A bridge between CPU and main memory

In comparison to CPU speeds, the main random access memory (RAM) lags behind in terms of data throughput. Caches form a solution: for smaller amounts of memory, throughput speeds can be increased, in essence trading storage space for access speed. Assuming many applications require operations on the same data, caches were introduced in between the CPU and main memory, so that data which is needed often is accessible in a much shorter time frame. Since it depends on the application which data is reused, caches determine on-the-fly what is worth keeping stored in cache, and what is not. When the CPU requires data, the system checks whether the requested data already is present in the cache. If it is, it is immediately sent to the CPU; this is a *cache hit*. Otherwise, in case of a *cache miss*, the data is fetched from main memory, stored in cache, and sent to the CPU.

Instead of using a single cache memory, it is also possible to place multiple caches in between the main memory and the CPU. The cache closest to the CPU is commonly called the level-1 (L1) cache; the second-closest the L2 cache, and so on. Generally, caches closer to main memory have greater capacity than the caches closer to the CPU, and caches closer to main memory are slower than the caches closer to the CPU; see Figure 1 for an illustration of such a multi-level cache.

The unit of data transfer between main memory and cache, or between two caches, is called a *cache line*. The size of a cache line varies per architecture, and is typically in between $8$ to $512$ bytes. When the CPU requests data already in the cache, there is a delay before the data is actually sent back. This delay is called the *latency*. The latency between CPU and cache is small; but if a cache miss occurs, the CPU-cache latency is increased by the cache to main memory latency, which is much larger. The throughput of a cache, or the speed in which large chunks of data can be read from or written to cache, is called its *bandwidth*.

In case of a cache miss, the question remains which data already in the cache is overwritten with newly requested data. Differences in handling this give rise to fundamentally different cache types. A *perfect cache* enforces some predefined policy when deciding which cache line to evict to bring in the new data. A first-in, first-out (FIFO) policy, so that the oldest data is the first to be evicted from the cache, is one option; however, when data is continuously re-used, the FIFO policy performs poorly.

An adaptation so that the least frequently used data is evicted instead, seems an improvement; this is called the LFU policy. However, when old data were very frequently requested but now no longer, this data will remain in cache much longer than what is optimal. With this observation a clue for a better suited cache policy becomes apparent: evicting the data from the least recently used cache line. This is called the LRU policy, and usually is the policy of choice. Worst-case examples can be constructed for any policy, however, including for LRU. Karlin et al. proved LRU is *competitively optimal*, that is, the number of cache misses is within the smallest possible constant factor from the optimal policy [KMRS88].

Implementation of these policies in hardware is complex, and thus costly, especially for larger caches. A cost-reducing way is to instead approximate in some way a more complex cache policy, or revert to a simpler policy. Such a cost-effective solution is, for example, the *direct mapped* policy. Here, memory addresses of requested data are translated to a specific cache line using a simple many-to-one translation, such as a modulo-based mapping.

## 1.2 Cache model

At the time of writing, caches typically use a hybrid form of direct mapping and the LRU policy; this will also be the cache model used throughout the remainder of the thesis. Such a cache is divided into $k$ parts of equal size, uses the direct mapping policy on each subcache, and the LRU policy in-between those subcaches.

This is referred to as a *$k$-way set-associative cache*. We denote such a cache by $C = (s, k, z, w)$, in which $s$ is the cache size, and $k$ is the number of subcaches. This implies $s$ must be perfectly divisible by $k$. A cache is not accessed by individual data *words*, but by cache *lines*. These lines are the smallest data piece retrievable from cache, and its size $z$ is usually $64$ bytes; a cache thus holds $L = s/z$ cache lines, implying that $s$ should also be perfectly divisible by $z$. A data word, in contrast, is the smallest data piece a given architecture can address. Its size is usually $64$ *bits*, and determines the number $w$ of data words which fit in a single cache line. A byte contains $8$ bits, thus in the case of $64$-bit data words and a cache with $z = 512$ bits, $w$ equals $8$ and requesting a single data word from cache thus brings along $7$ neighbouring data words as well.

When data from main memory is brought into cache, older elements have to be evicted from the cache to make room. The cache $C$ can be seen as an $(L/k) \times k$ matrix consisting of cache lines $C_{ij}$, and the main memory can be organised in consecutive chunks of size $z$, with each chunk given an identifier $x$ equal to the smallest memory address contained within. If $x$ is brought into cache, it will replace the cache line in

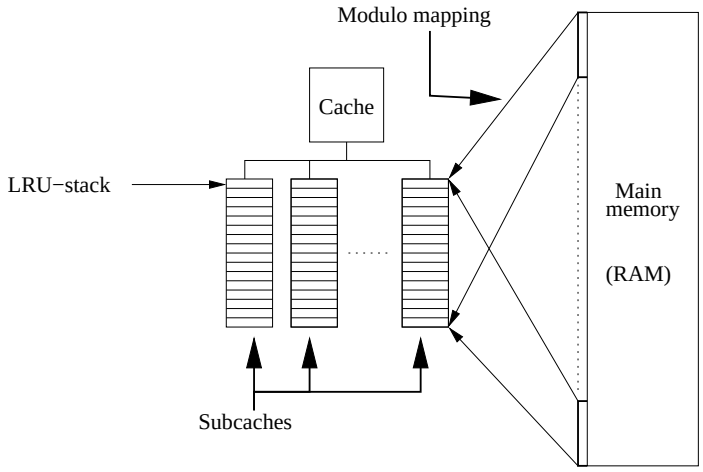



Figure 2: Illustration of a $k$-way set associative cache. Each of the $k$ subcaches has $L/k$ cache lines; the cache can be viewed as an $(L/k) \times k$ matrix. Each row of this matrix is a *LRU stack* and each column a subcache. Data from main memory is read in chunks of size $z$, the *cache line size*.

$C_{ij}$ with $i = x \mod (L/k)$ and $j$ a value in $\{0, 1, \ldots, k-1\}$. The row index $i$ is called the *set ID*, and is determined by modulo mapping. The column index $j$, the *line ID*, is determined by the Least Recently Used (LRU) policy: $j$ will be chosen so that $C_{ij}$ is the least recently used cache line out of the possible choices (with the set ID $i$ fixed). Figure 2 illustrates this cache model.

When $k \to \infty$, or rather, $k = L$, our cache model corresponds to the $(Z, L)$ ideal-cache model introduced by Frigo et al. [FLPR99]; when the cache is full and a new cache line has to be brought into cache, the least recently used line in the entire cache is evicted. In practice, most architectures have a set associativity $k$ in the order of $4$ or $8$ for caches close to the computing node, or up to $k = 48$ for caches closer to main memory.

## 1.3 Examples of cache-based architectures

In practice, processors employ multi-level caching, but not always with caches placed serially in between the CPU and main memory (e.g., the IBM Power6+ architecture); especially with multicore processors changes from serial cache hierarchies have appeared, so that cache hierarchies form tree-like structures. The caches themselves follow the set-associative paradigm with the LRU policy, or remain close to it. In determining when to propagate changes in cache lines back to main memory, the write-back policy is usually employed. Three architectures will be described in the following text: the dual-core IBM Power6+ which has 2 cores each running at $4.7$GHz

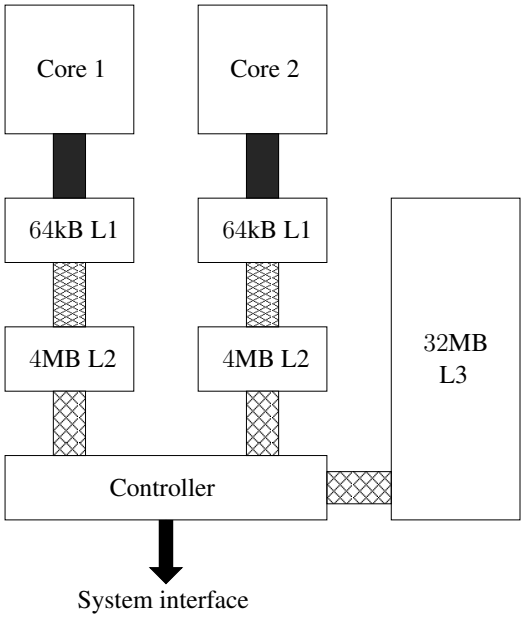



Figure 3: Diagram of the IBM Power6+ cache hierarchy

($2 \times 4.7$GHz), the quad-core Intel Core 2 Q6600 at $4 \times 2.4$GHz, and the quad-core AMD Phenom II 945e at $4 \times 3.0$ GHz. All three architectures are used in experiments further on in this thesis.

On the IBM Power6+, each core possesses its own $8$-way $64$kB L1 data cache. An $8$-way $4$MB L2 cache is 'semi-shared' between the two cores, meaning the caches (and therefore the cores) can communicate through a cache controller, instead of through a fully shared cache or main memory. A $16$-way $32$MB L3 cache is uniformly addressable by both cores through the same controller, and thus is fully shared, but not serially placed; Figure 3 provides an illustration. The architecture word size is $8$ bytes ($64$ bits), and the cache line size $z$ is $128$ bytes, and hence the number of words $w$ which can fit in a single cache line is $16$.

Each core on the quad-core Intel Core 2 (Q6600) processor has access to a 32 kB 8-way L1 cache, and two pairs of cores share a single 4 MB 16-way L2 cache. These two L2 caches are connected to the rest of the system; thus in essence this processor consists of two dual-core processors combined on one chip; see also Figure 4. The cache line size $z$ is $64$ bytes, while the word size is $64$ bits; hence $w = 8$.

The AMD Phenom II 945e processor provides the last example. Here, a single core is connected to a $64$kB L1 data cache and a $512$kB L2 data cache. All cores share a single $6$MB L3 cache; see Figure 5. The parameters $z$ and $w$ are the same as for the Intel Q6600. Table 1 provides a simple overview of this data.

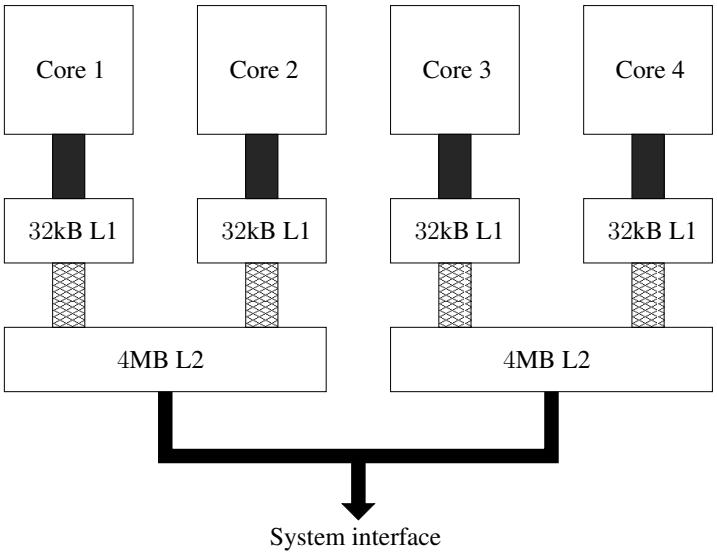



Figure 4: Diagram of the Intel Core 2 Q6600 cache hierarchy

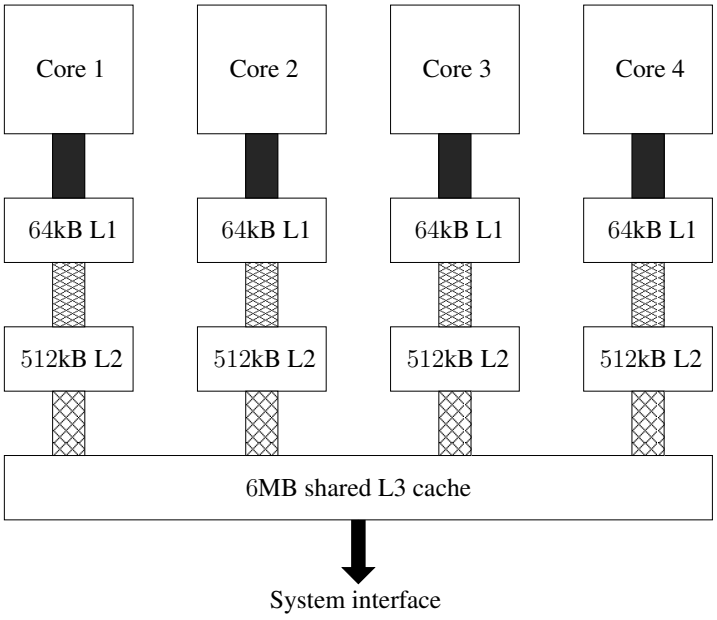



Figure 5: Diagram of the AMD Phenom II 945e cache hierarchy

| Architecture (cores & speed) | | Cache size (associativity) L1 | L2 | L3 | $z$ | $w$ |
|---|---|---|---|---|---|---|
| IBM Power6+ | (2 x 4.7GHz) | 64kB (8) | 4MB (8) | 32MB (16) | 128B | 16 |
| AMD 945e | (4 x 3.0GHz) | 64kB (2) | 512kB (16) | 6MB (48) | 64B | 8 |
| Intel Q6600 | (4 x 2.4GHz) | 32kB (8) | 4MB (16) | – | 64B | 8 |

Table 1: Cache parameters of various architectures [vdS10, Wor]. All these architectures employ a 64-bit word size.

## 1.4 Cache behaviour under dense SpMV multiplication

In the field of dense basic linear algebra subroutines (BLAS), Goto and van de Geijn [GG02] showed that by tweaking algorithms to specific architectures, large speedups can be obtained. This is achieved through minimising the number of jumps through memory, or improving cache efficiency in some other way. In particular, they achieve this not only for data caches, but also for the translation lookaside buffer (TLB). These optimisations were done separately, for each different architecture available, which is a general disadvantage of such *cache-aware* algorithms. Goto and van de Geijn provide a concise summary of previous work on the area of architecture-specific optimisation of dense BLAS [GG02, Section 2].

To avoid this, auto-tuning software libraries [KKHY06, WP05] have become a focus for much research, most notably FFTW (for fast Fourier transforms) [FJ98, FJ05], OSKI (basic sparse BLAS) [VDY05], and ATLAS (dense BLAS) [WPD01]. These libraries may run various benchmarks upon installation to help optimise algorithms for the hardware specifics of the target machine, or may even attempt this *during* real-time execution. Another approach is to design algorithms which are (asymptotically) optimal in terms of cache efficiency on any cache-based architecture. These *cache-oblivious* algorithms were introduced by Frigo et al., and for some applications have been shown to obtain asymptotically optimal bounds [FLPR99, BBF$^+$07].

Figure 6 shows a small example of cache effects in the context of dense MV multiplication $y = Ax$, with $A$ a dense $m \times n$ matrix, under the assumption of a perfect cache ($k \to \infty$). For simplicity, the size of one data word is set to fit exactly in one cache line, i.e., $w = 1$. The cache $C$ is given by a $1 \times L$ matrix and cache lines are 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 ($L = 2$), $x_0$ 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 from $x$ and $A$ are brought in first, pushing $y_0$ out of the cache just before it was needed again. Having such a small cache would cause $\mathcal{O}(mn)$ cache misses on the output vector $y$. If the cache could contain exactly three data words instead of two, $y_0$ would still have been available and these misses would not have occurred.

For larger caches similar considerations still hold. When the multiplication reaches

### *Architectures not reliant on caches*

This thesis mostly assumes cache-based systems, and the presented reordering methods may not work on other existing architectures. For instance, Graphical Processing Units (GPUs) consist of thousands of simple processing units, called *stream processors* (SPs), which operate on linear streams of data and do not require caching. SPs cannot follow more complex execution paths (such as branching based on if-statements), nor do they operate on scattered data (such as graph or unstructured sparse matrix computations). Recent GPUs are becoming more able on the last two fields, however, but still incur large penalties in computation speed when faced with non-linear instructions or data. Newer GPUs also gradually employ larger caches shared by a limited number of stream processors; and these caches may become programmable, allowing run-time switches between shallow or deep cache hierarchies.
Another example is the Cell Broadband Engine (Cell BE). This processor consists of one main processor (PPE) and several co-processors (SPUs). These are connected by a ring on which data circulates. The PPE is also connected to main memory, and the SPUs have access to a small amount of local memory. The PPE can transport data between the main memory and the data ring. SPUs can get data from the ring, perform local computations, and return the results back to the ring. The idea is to bring the data to the many computational units to avoid using complex cache hierarchies; this makes software creation more complex, however.

the next row (i.e., executes $y_1 = y_1 + a_{10}x_0$), then, depending on the cache size, there are two possibilities: either $x_0$ is still in cache and no cache miss occurs, or $x_0$ was evicted and must be brought back in. This results in $\mathcal{O}(n)$ cache misses on $x$, *for each row*; and hence $\mathcal{O}(mn)$ cache misses occur on the input vector $x$ during the whole algorithm.

These misses are non-compulsory: most can be avoided by limiting the number of subsequent accesses on $x$ by using some blocking parameter $q$, so that after processing $x_{q-1}$, the 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

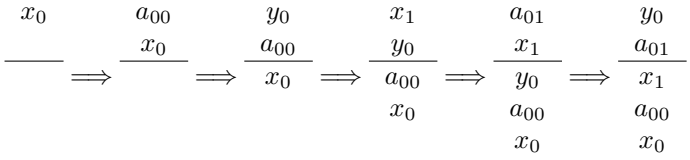



Figure 6: LRU stack progression during the first steps of dense MV multiplication. Whenever a new data element is pushed onto the stack, it is displayed on top. Least recently used elements thus appear lower in this figure. Note that if all memory lines below the bars are evicted, $y_0$ is evicted at step 5, while it is immediately reused the step after.

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 there are far fewer column blocks than matrix rows. Although the problem is less severe, still accesses on $y$ may be limited using some blocking variable $p$ (which need not be equal to $q$) and applying the same trick on the row indices. This method is appropriately named *cache blocking* [NVDY07] since, in effect, $A$ is subdivided into blocks of size $p \times q$ for which an MV algorithm does not incur unnecessary cache misses.

## 1.5 Difficulties under sparse SpMV multiplication

The difference between the dense case and the sparse case is caused by the way matrices are stored. In the dense case, storage is done by means of an $m$ by $n$ dense array. General sparse matrices, however, usually are stored in the so-called Compressed Row Storage (CRS), as, for example, described by Bai et al. [BDD$^+$00]. Here, nonzeroes are stored in a row-major ordering; see also Figure 1.1(left) on page 7. CRS utilises three arrays: one array for storing individual nonzero matrix entry values (*nzs*), one for storing the column index of those nonzeroes (*col_ind*), and one for storing at which index 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 nonzeroes of the $m \times n$ sparse matrix $A$. The array *row_start* requires only $m + 1$ words of space, bringing the total up to $2\mathrm{nz}(A) + m + 1$. For clarification, Algorithm 1 shows how to perform the SpMV multiplication $y = Ax$ on a matrix stored in CRS format. Note that vector and matrix elements are indexed starting from zero: the matrix elements are $a_{ij}$, $0 \leq i < m$ and $0 \leq j < n$. By switching the roles of matrix rows and columns, the Compressed Column Storage (CCS), which also sees much practical use, can be defined as well.

Another well-known sparse matrix storage method is the Triplet scheme, which is also known as the Coordinate (COO) scheme. Here, the three arrays used are *nzs*, *row_ind* and *col_ind*. The nonzero values array and column index array are the same as those for the CRS scheme. The *row_ind* stores the row index of each nonzero, similarly to *col_ind* array and the nonzero column indices. A SpMV multiplication

**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 initialise: $y = \mathbf{0}$
2: **for** $i = 0$ **to** $m - 1$ **do**
3: **for** $k = row\_start[i]$ **to** $row\_start[i + 1]-1$ **do**
4: $j = col\_ind[k]$
5: $y[i] = y[i] + nzs[k]*x[j]$
6: **return** $y$

**Algorithm 2** Matrix–vector multiplication using the Triplet scheme

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

1: Allocate $y$ of size $m$ and initialise: $y = \mathbf{0}$
2: **for** $k = 0$ **to** $nz - 1$ **do**
3: $y[row\_ind[k]] = y[row\_ind[k]] + nzs[k] * x[col\_ind[k]]$
4: **return** $y$

algorithm using the Triplet scheme is given in Algorithm 2. Note that this scheme uses 3nz($A$) words of memory, and since data movement costs time as well, the CRS scheme usually is more efficient in terms of both memory usage and execution speed.

Observing how CRS and the Triplet schemes work, and with the dense case described earlier in mind, the following remarks can be made:

- elements of the three matrix arrays are accessed exactly once during multiplication,
- the order of matrix element access determines the access patterns of the input and output vectors,
- the order of matrix element access is determined by its storage scheme.

The SpMV multiplication algorithm using the ordering induced by CRS can be analysed in much the same way as in the dense case: Figure 7 shows the LRU stack during SpMV multiplication using CRS. The major difference is that the column in-

dices are not guaranteed to be consecutive and that row jumps occur irregularly since rows typically do not contain the same number of nonzeroes.

This greatly increases the difficulty of predicting when cache misses occur, as it depends completely on the relative column-wise positions of the nonzeroes in $A$. Hence any blocking parameters $p$ and $q$ cannot be determined solely from $m, n$, and $s$; additional information on the structure of $A$ is required. Cache-aware blocking thus becomes much harder to apply, which motivates the development of run-time cache-aware auto-tuning kernels such as OSKI [VDY05].

# 2 Hypergraph representations of sparse matrices

The nonzero pattern of a sparse matrix $A$ can be represented as a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$, with $\mathcal{V}$ the set of vertices and $\mathcal{N}$ the set of nets. Let a vertex $v_j \in \mathcal{V}$ correspond to the $j$th 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 [ÇA99], since each matrix row is represented by a hyperedge. Figure 8 displays a small example sparse matrix and its row-net hypergraph. Other models to represent sparse matrices include the *column-net* [ÇA99] and *fine-grain* model [ÇA01]. In the column-net model, the roles of the row and column indices are interchanged as compared to the row-net model. The fine-grain model differs from the previous models in that the vertices correspond to individual nonzeroes $a_{ij}$. A net then contains nonzeroes sharing the same row or column: see Figure 9.

All models enable reasoning on the structure of sparse matrices. For example, the row-net model in Figure 8 shows that columns 3 and 4 are structurally isolated from the other columns, in the sense that they do not contain nonzeroes sharing rows with any other columns. It also shows that column 1 and 8 are connected through four matrix rows; there is a row connecting column 1 to column 5, one connecting 5 to 6, one connecting 6 to 2, and finally one from 2 to 8.

The fine-grain model, as its name implies, gives information on sparse matrix structure on a much finer scale; it enables us to learn about the interconnection of individual nonzeroes. Like the row-net model showed that columns 3 and 4 were

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

Figure 7: LRU stack progression during the first steps of SpMV 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 nonzeroes at row $i_r$ are exhausted and the algorithm proceeds with the following row.

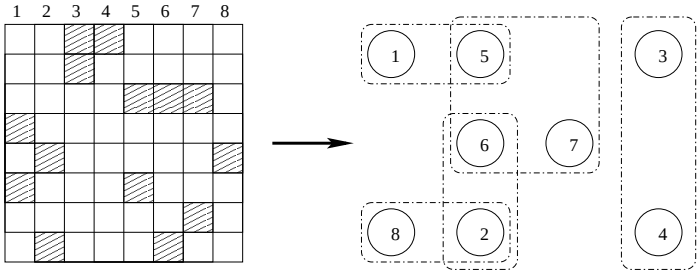



Figure 8: Translation from a sparse matrix to a hypergraph using the row-net model. Vertices 1 through 8 represent the columns. A net is cast over those vertices that share nonzeroes in a row; each net represents a row. Nets containing only one vertex are omitted.

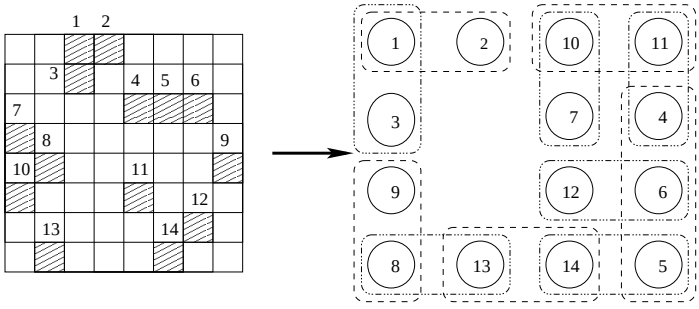



Figure 9: Translation from a sparse matrix to a hypergraph using the fine-grain model. Vertices 1 through 14 represent the nonzeroes. A net is cast over those vertices that share a row or column; nets corresponding to rows are dashed, while those corresponding to columns are dash-dotted.

isolated, the fine-grain representation in Figure 9 shows that the cluster of nonzeroes $1, 2, 3$ is isolated. The nonzeroes with the farthest interconnect are nonzero 7 and 9, tracing a path through 8 hyperedges. Note that this information is indeed a refinement from the row-net model, which already pointed to the columns 1 and 8, in which the nonzeroes 7 and 9 reside.

A recent adaptation of the fine-grain model regards structurally symmetric matrices. Formalising the fine-grain approach, the set of vertices $\mathcal{V}$ corresponds to the nonzero elements, and the set of hyperedges $\mathcal{N}$ consists of the union of two sets $\mathcal{N}^{\text{row}} \cup \mathcal{N}^{\text{col}}$; one corresponding to matrix rows, and the other to matrix columns. Yet in a symmetric matrix, the structure of the $k$th row is equal to the structure of the $k$th column. Hence modelling these twice seems inappropriate, and the number of hyperedges can be halved. Doing this, the number of nonzeroes in $\mathcal{V}$ cannot be reduced; if half of the nonzeroes were to disappear (i.e., only store a triangular part of the matrix $A$), then both categories of hyperedges needed to be retained to model the original matrix. Hence storing either half of the vertices *or* half of the hyperedges both yield valid symmetric fine-grain models. As the number of vertices $\text{nz}(A)$ usually is much higher than the number of hyperedges $m+n$, choosing to halve the nonzeroes results in much smaller memory requirements but also in a much more highly connected hypergraph. On the other hand, halving the number of hyperedges results in a more loosely coupled hypergraph, but retains the bulk of memory usage. Which formulation is appropriate for handling symmetry thus seems application dependent. When considering symmetric matrices, an additional argument appears in favour of halving the number of hyperedges: by letting rows and columns be modelled by the same hyperedges, symmetry is retained. See also page 49 for further considerations.

A major application of these hypergraph models specifically regarding sparse matrices, is their use in partitioning. Huge linear systems often need to be solved in parallel on supercomputers due to time or memory limitations. This entails partitioning the matrix, in such a way that operations on this matrix cause as little communication as possible: in the previous example, separating the nonzeroes in $\{1, 2, 3\}$ and $\{4, \ldots, 14\}$ yields two independent parts on which no communication is required during parallel execution of most common sparse matrix operations, such as sparse matrix–vector multiplication $Ax$, matrix-transpose multiplication $A^T A$, complete or incomplete LU decomposition $A = LU$, and others. Many of these applications will be discussed in the following chapters, but the focus will remain on sparse matrix–vector multiplication.

# 3 Sparse matrix partitioning

Assume an input matrix $A$ is represented by a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ by the row-net model. The vertex set $\mathcal{V}$ can be partitioned 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 $\cup_{s=0}^{p-1} \mathcal{V}_s = \mathcal{V}$. Such a partitioning is one-dimensional, as only rows are distributed over parts; similarly, a partitioning based on the column-net model would also be one-dimensional. Given a partitioning, the

vertices in a net $n_i$ may be distributed over several subsets (or parts), resulting in a *cut net*. The number of subsets $\lambda_i$ over which the $i$th net is cast is called the *connectivity* of the $i$th net. Using a factor $c_i$ to express the relative importance of each net, a cost can be assigned to this partitioning:

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

This cost function is commonly called the $\lambda - 1$ metric, or connectivity$-1$ metric, and originates from the field of electronic circuit simulation [Len90]. It is used extensively in parallel computing; see, e.g., work by Çatalyürek et al. [ÇA99]. In matrix partitioning, no preference is assumed as to which matrix rows are cut, and $c_i$ is set to 1. While partitioning a hypergraph in this manner, the function (1) is minimised. Other values for $c_i$ may be interesting in different applications, for example in clustering, domain decomposition methods or preconditioning.

When considering parallel SpMV multiplication, parts correspond to processors. A cut column-net in some partitioning indicates that the required component of the vector $x$ will be used by multiple processors. Since the vectors $x$ and $y$ involved with the parallel computation are distributed among the processors to achieve scalability, a cut column-net results in communication; 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*. This is the rationale for the cost function (1): when minimising communication, any cut nets must be avoided: ideally, $\lambda_i = 1$. Even if a net is cut, it should be cut in as few parts as is feasible, so that the number of processors which have to initiate communication over this net remains minimised.

A good load balance must be maintained as well. The number of nonzeroes 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 nonzeroes corresponding with the $j$th vertex (e.g., the number of nonzeroes in the $j$th column in case of a row-net model). We then also minimise:

$$\max_{0 \leq s < p} \sum_{j:v_j \in \mathcal{V}_s} w_j. \tag{2}$$

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

$$\epsilon = \frac{\max\{|\mathcal{V}_0|, |\mathcal{V}_1|\}}{(|\mathcal{V}_0| + |\mathcal{V}_1|)/2} - 1. \tag{3}$$

When $\epsilon = 0$ load balance is perfect; when $\epsilon = 1$ the computational load is fully imbalanced ($\mathcal{V}_0$ or $\mathcal{V}_1$ is empty and the other set is equal to $\mathcal{V}$).

| | |
|---|---|
| $A$ | A general (unless otherwise specified) sparse matrix, |
| $m, n$ | which is of size $m \times n$. |
| $\mathcal{H}$ | A hypergraph corresponding to $A$ according to a specified model; |
| $\mathcal{V}$ | the hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ has its vertices $v \in \mathcal{V}$, and |
| $\mathcal{N}$ | has its nets $n \in \mathcal{N}$ with $n \subseteq \mathcal{V}$. |
| $G$ | A graph usually modelling similarity of rows and/or columns of $A$; |
| $V$ | the graph $G = (V, E)$ has its vertices $v \in V$, and |
| $E$ | has its edges $e \in E = V \times V$. |
| $C$ | A computer cache with parameters $C = (s, k, z, w)$, such that |
| $s$ | $s$ is the total cache size in bytes, |
| $k$ | $k$ is the number of subcaches, |
| $z$ | $z$ is the cache line size in bytes, |
| $w$ | $w$ is the number of data words which fit in a single cache line, and |
| $L$ | $L = s/z$ is the number of cache lines. |

Table 2: Glossary of objects used throughout the thesis

The demand for a perfect load balance and for a minimised total connectivity cost usually conflict with each other. In an attempt to construct a partitioning which is acceptable in terms of both criteria, the maximum load imbalance $\epsilon$ is taken as a predefined parameter. A software package implementing this approach for sparse matrix partitioning is Mondriaan [VB05]. Mondriaan can additionally split the matrix in both dimensions, by switching between the row-net and the column-net model, each time choosing the best split. This semi two-dimensional method is referred to as the *Mondriaan scheme*.

An addition is to consider the fine-grain representation as well, thus choosing the best split out of the row-net, column-net, and fine-grain models; this is the *hybrid* scheme. Using the Mondriaan or the hybrid scheme increases partitioning time compared to one-dimensional methods, and usually results in a partitioning of better quality. This should be worth the increase of partitioning time, however; an investigation into this can be found in the book *Combinatorial Scientific Computing* [NS11]:

[BFAY+11]: Rob H. Bisseling, Bas O. Fagginger Auer, A. N. Yzelman, Tristan van Leeuwen, and Umit Çatalyürek, *Two-dimensional approaches to sparse matrix partitioning*, Combinatorial Scientific Computing (Uwe Naumann and Olaf Schenk, eds.), CRC Press, 2011, to appear.

There, quality is investigated by comparing partitioning time, and the theoretical resulting communication costs. This is backed by actual SpMV multiplication timings on a parallel machine using up to 32 IBM Power6+ processors.

# Part I

# Sequential methods

# Chapter 1

# One-dimensional cache-oblivious multiplication

This chapter describes a cache-oblivious sparse matrix–vector multiplication scheme. It works by adapting the sparse input matrix, permuting it into the *Separated Block Diagonal* (SBD) form. Permutation is two-sided: the reordered matrix can be written as $PAQ$ with $P, Q$ permutation matrices, where this permutation is found in a pre-processing step. The only assumption made is that the matrix is stored in Zig-zag CRS order, instead of plain CRS; see Figure 1.1. It will be shown that sparse matrices in SBD form attain good efficiency as cache use is improved, that the method minimises an upper bound on cache misses, and that this upper bound is strict when a specific number of blocks are created. Exceeding this number of blocks theoretically does not harm efficiency, thus taking this number as large as possible yields a cache-oblivious method. The presented reordering method is implemented in the Mondriaan sparse matrix partitioner [VB05], since version 3.

This chapter is based completely on published work [YB09]:

- 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** (2009), no. 4, 3128–3154.

## Earlier work

Earlier work dealing explicitly with improving cache efficiency in SpMV multiplication are the following. Kowarschik et al. [KUWK00] improve the speed of an iterative multigrid method for Poisson's equation on a two-dimensional grid by cache-aware combining of computations from different iterations and reordering of computations. A speedup of a factor of 4.6 is obtained by the latter technique for certain grid sizes.

The authors note that cache effects would be even stronger in the three-dimensional case.

Das et al. [DMS+94] use reordering techniques developed for reducing fill-in 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 [Tol97] performs an extensive study of such ordering techniques, and found that CM ordering works well on 3D finite-element test matrices when used in combination with blocking into small dense blocks. Additionally, CM is cheap to compute, requiring computing time equivalent to a few SpMV multiplications. Reverse CM ordering is found to be less competitive, as it leads to fewer dense blocks. Other techniques were studied as well, including finding small dense $1 \times 2$ and $2 \times 2$ blocks, and prefetching of data.

Pinar and Heath [PH99] try to create contiguous blocks of nonzeroes in the matrix rows by reordering the columns, formulating the problem as an instance of the Travelling 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 [VB05] and PaToH [ÇA99]. Small dense $1 \times 2$ 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 [VM05] 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. $1 \times 2$ or $1 \times 3$. A matrix with block size $1 \times 1$ is also included to represent the remaining nonzeroes in the standard CRS format. Blocks are created by greedily grouping rows and columns based on a scaled inner product metric.

Nishtala et al. [NVDY07] 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 [VDY05].

White and Sadayappan [WS97] use the Metis graph partitioner [KK98] 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 [SCFK04, SH04] 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 [HLP07] 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, but for SpMV multiplication with eight simultaneous input vectors instead of one. An enhancement of this method, making it efficient for regular SpMV multiplication, can be found in Chapter 3. The Hilbert approach has the advantage that it exploits naturally existing locality in the matrix at all cache levels, without knowing about the characteristics of the cache. Furthermore, this approach is two-dimensional. No attempt is made to enhance locality, however. It may be possible to combine the Hilbert method with the method presented in this chapter, by first running the matrix reordering presented here and then using fractal storage.

Summarising, only a few approaches are explicitly cache-aware or cache-oblivious. The approach incorporated in OSKI [NVDY07, VDY05] is cache-aware since it adapts to the cache sizes and other characteristics; this also holds for the approach of Kowarschik et al. [KUWK00]. The space-filling curve approach [HLP07], 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. For example, approaches based on using small dense blocks [DMS+94, PH99, Tol97], are sometimes called *register blocking*, and will improve use of the registers and the L1 cache, but not the L2 and L3 cache.

## 1.1 Sparse matrix storage formats

A drawback of standard CRS implementation is the use of the *row_start* array for keeping track which nonzeroes 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 is no other way of knowing when a row ends, optimised 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: the address (pointer) `px` of the currently used element from $x$ should be updated as `px=x+col_ind[k]`. Faster would be `px+=diff[k]`, with `diff[x]` equal to `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 [Kos02]. A sparse matrix–vector

algorithm utilising this ICRS scheme is given in Algorithm 3. Note that an increment in row index is signalled by causing $j$ to overflow (i.e., $j \geq n$) so that $j - n$ corresponds to the actual column index of 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 3 is a few lines larger than that of Algorithm 1, it can be more efficiently implemented by using suitable pointer arithmetic. This causes the instruction overhead to drop [Kos02, 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, saving memory bandwidth and thus increasing cache performance.

**Algorithm 3** 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 nonzeroes $\mathrm{nz}(A)$.
**Output:** a dense vector $y$, where $y = Ax$.

```
1:  Allocate y of size m and initialise: 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: return y
```

A further, cache-oblivious way to reduce the number of cache misses by adapting sparse matrix storage, is to prevent jumping from the last to the first column when a row increment occurs. Instead, the multiplication algorithm could just start at the last column of that row, and process nonzeroes in reverse order until it arrives at the first column. At the next row increment this recipe can be repeated. Assuming all rows are non-empty, the resulting kernel thus processes nonzeroes in the standard increasing order on even-numbered rows, and in decreasing order on odd-numbered rows. The resulting data structure is called *zig-zag CRS* (ZZ-CRS). Figure 1.1 illustrates both the CRS and zig-zag CRS orderings. Note that the Incremental adaptation of CRS

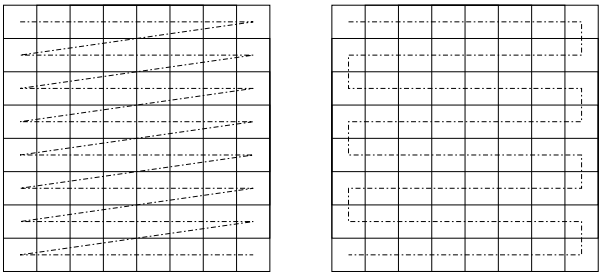


Figure 1.1: The CRS (left) and ZZ-CRS (right) orderings. The dash-dotted lines show the order in which matrix nonzeroes are stored.

and this zig-zag adaptation can be applied simultaneously to form the ZZ-ICRS data structure.

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, only $\mathcal{O}(n - L)$ cache misses per row are incurred, 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, there are $\mathcal{O}(\frac{n}{w} - L)$ misses instead of $\mathcal{O}(\frac{n}{w})$. This is only a tiny improvement, but it enables the use of the Mondriaan partitioner to find matrix permutations for more efficient sequential SpMV multiplication.

## 1.2 An adapted partitioning algorithm

The partitioning strategy followed is an adaptation of the Mondriaan partitioning scheme. From its construction follows the introduction of the SBD form, and why this form is well suited to the sequential SpMV multiplication.

Consider the following bipartitioning of a row-net hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ representing a sparse matrix $A$. $\mathcal{V}$ is partitioned into parts $\mathcal{V}_0, \mathcal{V}_1$ while taking into account the load imbalance and the $\lambda - 1$ cost function. Note that the indices of the columns corresponding to the vertices in each part do not have to be consecutive. A partitioning of the *nets* in $\mathcal{N}$ will be induced as well: the set of nets with vertices only in $\mathcal{V}_0$ is denoted by $\mathcal{N}_-$, and the set of nets with vertices only in $\mathcal{V}_1$ is denoted by $\mathcal{N}_+$. The remainder, the set of cut nets (nets which have vertices in both parts), is denoted by $\mathcal{N}_c$. This bipartitioning scheme can be recursively applied so that it generates $p$ vertex subsets, as well as $2p - 1$ different hyperedge subsets.

These partitionings of $\mathcal{V}$ and $\mathcal{N}$ can be used in SpMV multiplication as follows. The algorithm first visits the matrix elements in the rows corresponding to nets in $\mathcal{N}_-$, followed by those corresponding to $\mathcal{N}_c$, and finally by $\mathcal{N}_+$; and furthermore, the nonzeroes are processed in zig-zag order. Like with the CRS data structure, the SpMV multiplication visits matrix rows in succession, and no unnecessary cache misses are incurred on the output vector $y$. To minimise the cache misses on the input vector $x$,

the number of parts $p$ should equal

$$p = \frac{n}{wL}. \tag{1.1}$$

This can be interpreted as the number of caches that would be needed to store the complete input vector $x$: $wL$ then 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 nonzeroes 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, the row length $n$ in the right-hand side of Equation (1.1) is replaced 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 exact. Another important case is where the row is uncut after partitioning the matrix into $p$ sets of columns. For such an uncut row no cache misses are incurred, again under the assumption that the zig-zag ordering is used, and this corresponds exactly to the bound $L(\lambda_i - 1)$ with $\lambda_i = 1$, which is zero.

As a result, the matrix is 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, minimising the communication volume in the $\lambda - 1$ metric using hypergraph partitioning improves cache utilisation as well.

It does not harm the cache efficiency to partition beyond the value of $p$ given in Equation (1.1). Hence, $p$ can be taken as large as possible, i.e., $p \to \infty$. If during partitioning, the matrix is reordered accordingly (see Section 1.3), a beneficial matrix structure at all values of $p$ is created. This way of reordering also minimises cache misses on multilevel cache hierarchies since for lower $p$ the behaviour for the larger caches is optimised, while for higher $p$ this is done for the smaller caches, without harming the earlier optimisations.

Putting the collection of cut rows between those of $\mathcal{N}_-$ and $\mathcal{N}_+$ ensures that data loaded in by first visiting columns corresponding to $\mathcal{V}_0$ are still available when processing $\mathcal{N}_c$. Since $\mathcal{N}_c$ also contains data from $\mathcal{V}_1$, cache performance deteriorates as data corresponding to $\mathcal{V}_0$ is swapped out in favour of those of $\mathcal{V}_1$. Afterwards, the algorithm proceeds with $\mathcal{N}_+$, which only deals with data corresponding to $\mathcal{V}_1$, thus purging the remaining data corresponding to $\mathcal{V}_0$ while taking advantage of the $\mathcal{V}_1$ data already brought into cache; thus allowing for a gradual transition from one set of rows to the other. Such a transition is expected to be smooth if the rows are sparse, and if the partitioner manages to keep the number of cut rows small.

### *Reordering for sparse LU decomposition*

Using the same vertex and hyperedge partitioning as in Section 1.2, a different reordering of matrix rows and columns can be obtained. Processing rows in $\mathcal{N}_c$ strictly after those in $\mathcal{N}^-$ and $\mathcal{N}^+$, is useful in matrix reordering for, e.g., nested dissection [Geo73] for Cholesky factorisation [HR98] or reordering rectangular sparse matrices for LU or QR factorisation as explored by Aykanat et al. [APÇ04]. Grigori et al. [GBDD10] employ hypergraphs to reorder unsymmetric matrices for sparse LU factorisation as well. After application of permutations such as described in Section 1.3, but adapted for this different row order, the resulting matrix is said to be in *bordered block diagonal* (BBD) form.
For sparse LU and QR this makes sense, since this postpones and prevents cascading *fill-in*; during decomposition, fewer new nonzeroes are created. Another example where cut rows come last, is the work on the package Monet [HMB00], which tries to create a bordered block diagonal form based on hypergraph partitioning with the cut-net metric (i.e., Equation (1) with $\lambda_i - 1$ replaced by 1 if $\lambda_i \geq 2$, and 0 otherwise).

## 1.3 Permuting to Separated Block Diagonal form

Consider a single bipartition of $A$ given by $\mathcal{V}_0$, $\mathcal{V}_1$, $\mathcal{N}_-$, $\mathcal{N}_c$, and $\mathcal{N}_+$. This then defines a permuted matrix $A_1$ as illustrated in Figure 1.2. The permutation of columns can be written 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-side multiplication with another permutation matrix $P$.

Let $I = (e_0|e_1|\ldots|e_{n-1})$ be the $n \times n$ identity matrix. Then the right-sided permutation matrix is given by $Q = (e_{q_0}|e_{q_1}|\ldots|e_{q_{n-1}})$, with $(q_i)_{i\in[0,n-1]}$ a permutation of the vector $(0, 1\ldots, n-1)$. Similarly, $P = (e_{p_0}|e_{p_1}|\ldots|e_{p_{m-1}})^T$ with index permutation $(p_i)_{i\in[0,m-1]}$. Recall that since $P, Q$ are permutation matrices, $P^{-1} = P^T$ and $Q^{-1} = Q^T$. Of course, the indices at the start of $(q_i)$ correspond to the columns in $\mathcal{V}_0$ and 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 thus given by $A_1 = PAQ$.

Subsequent recursive bipartitioning results in similar row and column permutations on disjoint row and column ranges. Therefore, after $r$ recursive steps, the matrix $A_r$ can still be written as $PAQ$, for certain $P$ and $Q$. As such, the modified SpMV

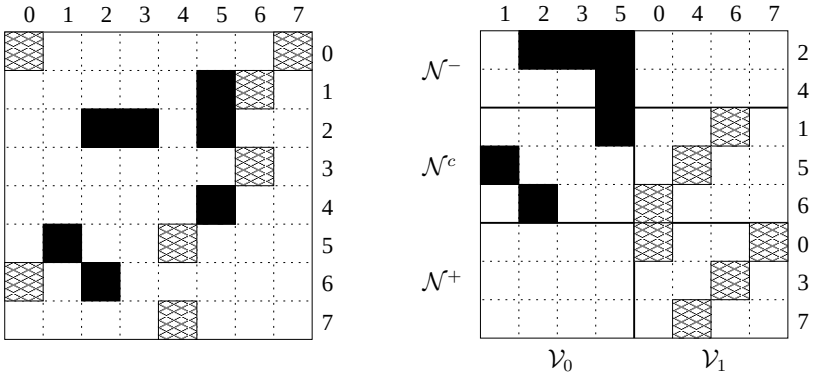



Figure 1.2: Original matrix (left) and permuted matrix (right) after one bipartitioning step. Shaded and black squares denote nonzero elements. Black columns denote vertices in $\mathcal{V}_0$ whereas shaded columns denote vertices in $\mathcal{V}_1$.

multiplication routine can be represented as follows:

$$y = Ax = P^T A_r Q^T x, \quad \text{so } \tilde{y} = A_r \tilde{x} \quad \text{with } \tilde{y} = Py, \quad \tilde{x} = Q^T x; \tag{1.2}$$

and a preprocessing technique for improving cache-efficiency is obtained, the form of which is quite similar to preconditioning techniques. Using this, the partitioning mechanism works for the following related linear algebra kernels, besides the SpMV multiplication:

- $y = Ax + \beta z$: $y = P^T(A_r Q^T x + \beta P z)$, which can be written as a standard SpMV multiplication $\tilde{y} = A_r \tilde{x} + \tilde{z}$ with $\tilde{y} = Py$, $\tilde{x} = Q^T x$, $\tilde{z} = Pz$.
- $y = A^T x$: $y = Q A_r^T P x$, so that $\tilde{y} = A_r^T \tilde{x}$ with $\tilde{y} = Q^T y$, $\tilde{x} = Px$.
- $y = AA^T x$: $y = P^T A_r A_r^T P x$, so that $\tilde{y} = A_r A_r^T \tilde{x}$ with $\tilde{y} = Py$, $\tilde{x} = Px$.

This shows that the cache-oblivious matrix reordering scheme can be applied for these operations, by simply running the corresponding sparse BLAS on the permuted input. Hence, the underlying BLAS software can be a cache-aware package, such as Sparsity [WPD01] or its successor, OSKI [VDY05], thus increasing cache efficiency beyond what is possible with either one method alone.

Some linear algebra kernels cannot be executed directly when using reordering. Most notably this applies to the kernel used in power method solvers, namely $y = A^s x$, $s \in \mathbb{N}$, $s > 1$. When supplying a permuted matrix, this kernel becomes $y = (PAQ)(PAQ)^{s-1}x$; between instances of the original matrix $A$, the matrix $QP$ appears. Since generally $QP \neq I$, a translating step between successive multiplications is required, which would be quite inefficient. This problem would arise, for instance, with the power method used in Google PageRank computations [BP98]. If

instead an alternative page ranking method is considered, such as the HITS method [Kle99] (see also the book [LM06, p. 117]), the power method is applied to the matrices $AA^T$ and $A^TA$ with $A$ a link matrix. After reordering of $A$, the matrix $(AA^T)^s$ then becomes $(PAQQ^TA^TP^T)^s = P(AA^T)^sP^T$, and similarly for $A^TA$, and the reordering scheme can be used without penalty. Alternatively, if reorderings are symmetric (i.e., $Q = P^T$), the PageRank method can also be used without translation. This requires an adapted two-dimensional partitioning and is further described on page 49.

Determining the column permutation is straightforward: the recursive bipartitioning yields an implicit order of subsets of $\mathcal{V}$. Recursively bipartitioning an arbitrary number of times, the resulting subsets can be numbered by describing the path followed in their construction: at each recursive step, a vertex subset either remains intact, is distributed left (0), or distributed right (1). A subset can thus be given the number $0110$, meaning it was first distributed left, then right, right again, and finally again on the left side. Interpreting this binary number yields a natural ordering on the vertex sets. Upon bipartitioning infinitely (that is, continue until each subset contains exactly one vertex), $n$ subsets each containing a single column are created. 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 done by looking at the set of *possible* final locations $Q_i$ of each row $i$ after permutation. At the start of the reordering, $Q_i = [0, m-1]$, which is the full range of matrix rows. After the first bipartitioning, three subsets $\mathcal{N}_{\{-,c,+\}}$ of nets are constructed. 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 1.2. This procedure is repeated after each following bipartitioning; see Figure 1.3. There, each horizontal straight line gives new row boundaries for the rows, with $Q_i$ the interval defined by those boundaries. After $n-1$ bipartitionings, a row ordering from the $Q_i$ can be deduced. Note that such an ordering need not be unique; for some $i$, $|Q_i| > 1$ is possible, even if $m < n$; consider, for instance, a dense input matrix.

Figure 1.4 shows the idealised structure that is obtained by recursively bipartitioning a sparse matrix and placing the cut rows in the middle. This new matrix structure is called the *separated block diagonal* (SBD) form, in analogy with the *bordered block diagonal* (BBD) form [DER86]. Note that this structure is created by using the $\lambda - 1$ metric, which tries to prevent further cuts inside already cut rows; cut rows may have internal structure which is not depicted in the figure. The cut-net metric does not have this advantage.

## 1.4 Cache simulation

The performance of the SpMV multiplication kernel depends on the actual structure of the sparse matrix itself. This dependency on matrix input makes it hard to analyse the effectiveness of reordering in terms of the introduced cache model. To make

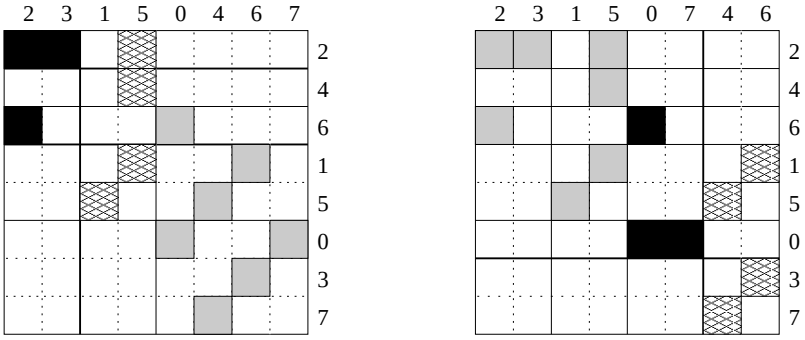



Figure 1.3: The reorderings after two (left) and three (right) bipartitionings of the original matrix in Figure 1.2. Grey squares denote nonzeroes not considered in the current bipartitioning step.

this possible nonetheless, a cache simulator is implemented, which enables accurate simulation of cache dynamics within the presented theoretical model: using this, theoretical cache performance on any sparse input matrix can be calculated and results can be correlated to actual wall-clock timings.

The cache simulator uses *run-time* memory allocation and memory access wrapper functions to catch and process data access. Upon access, a pointer to the allocated region is obtained. The cache simulator pushes the address $x$ onto the stack corresponding to the set ID $i$; this is where the simulator detects if a cache hit or miss has occurred. If the corresponding cache line already is present, it is removed from its current location in the stack and reinserted at the top; if it is not present, it is inserted at the top. If this would cause the stack size to overflow (i.e., would contain more than $k$ items), the address at the bottom of the stack is evicted. This process is repeated in case one cache line did not suffice for the requested data, with the appropriate larger memory address. No trace data is stored during the simulation process.

This simulation is only applied to accesses to the matrix $A$ and the input and output vectors $x$ and $y$, during the SpMV multiplication. This method enables simulation of caches with arbitrary parameters, as long as they fit in the $k$-way set associative category. It simulates only one level of cache, however.

## 1.5 Experiments

Input matrices are assumed to be stored in Matrix Market format. These are processed by the Mondriaan software package [VB05], in which the one-dimensional reordering method is implemented. Mondriaan keeps track of the permutation matrices $P$ and $Q$ while partitioning, as discussed in the previous section, and writes the reordered matrix $PAQ$ to file. The reordering time, in terms of SpMVs (on the original matrix),

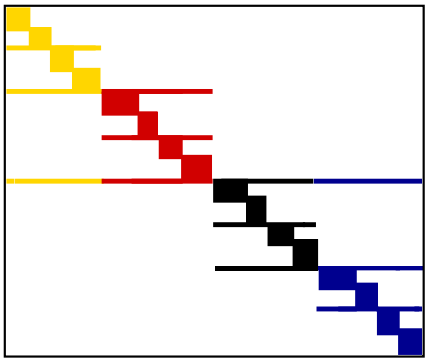


Figure 1.4: Separated block diagonal (SBD) structure of a sparse matrix obtained by recursively partitioning in the column direction and moving the cut rows to the middle. The recursion has been stopped after creating 16 diagonal blocks. The four colours indicate the block structure at $p = 4$.

is recorded as well. Due to the large number of columns of many test matrices, taking the number of parts to infinity ($p = n$) may take a very long time. Therefore, the number of parts is limited to an initial maximum of 400, or less for the larger matrices. These output files are then used by a SpMV multiplication benchmarking program. Wall-clock timings of sequential SpMV multiplications for various $p$ and $\epsilon$ are measured, using various storage schemes (CRS, ICRS, ZZ-ICRS), and the average time taken is reported.

Two computer architectures are used. The first machine is based on the quad core Intel Core 2 Q6600 processor and has 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 $z = 64$ bytes. The second machine is the Dutch national supercomputer Huygens at SARA in Amsterdam. This machine consists of 1664 dual-core IBM Power6+ processors divided over 104 nodes. Each core has its own 64 kB 8-way L1 data cache (besides a 64 kB L1 instruction cache), as well as 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 $z = 128$ bytes. Each node of 16 processors has 128 GB main memory. See also Table 1 and the various figures in Section I.1.3.

Several test matrices were taken from the University of Florida Sparse Matrix Collection [Dav11]. Table 1.1 shows all matrices and their properties. Matrices from finite element method (FEM) applications are fidap037, s3dkt3m2, and bmw7st1. From the field of chip design come the memplus and rhpentium matrices; lhr34 comes from chemical process modelling, and nug30 from optimisation. The rand matrices are generated randomly so that each entry has a constant probability of becoming a nonzero. Cage14 is a matrix arising from modelling DNA electrophoresis, and

| Name | rows | columns | nonzeroes | |
|---|---|---|---|---|
| rand10000 | 10000 | 10000 | 49987 | U |
| fidap037 | 3565 | 3565 | 67591 | S |
| memplus | 17758 | 17758 | 126150 | S |
| rhpentium | 25187 | 25187 | 258265 | U |
| lhr34 | 35152 | 35152 | 764014 | S |
| rand50000 | 50000 | 50000 | 1249641 | U |
| nug30 | 52260 | 379350 | 1567800 | S |
| s3dkt3m2 | 90449 | 90449 | 1921955 | S |
| tbdlinux | 112757 | 21067 | 2157675 | U |
| bmw7st1 | 141347 | 141347 | 3740507 | U |
| stanford | 281903 | 281903 | 2312497 | U |
| stanford-berkeley | 683446 | 683446 | 7583376 | U |
| wikipedia-2005 | 1634989 | 1634989 | 19753078 | U |
| cage14 | 1505785 | 1505785 | 27130349 | S |

Table 1.1: Matrices used in the experiments, sorted by category first and number of nonzeroes second. The first category contains matrices of smaller size, where input and output vectors typically fit into the L2 caches; the second category contains larger matrices. The final column indicates whether the matrix is considered structured (S) or unstructured (U).

tbdlinux is a term-by-document matrix of the Linux manual pages. The stanford, stanford-berkeley and wikipedia matrices all are link matrices; these are binary square matrices such that a single web page corresponds to a unique row and column index, and a nonzero entry at $(i, j)$ records when a link from web page $i$ to the web page $j$ exists.

Generally, the 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 a matrix is expected to fall in the first category, it is *structured*; otherwise, it is deemed *unstructured*. Random matrices may be seen to form a separate category. Their artificial construction makes partitioning inherently difficult. This is discussed in more detail on page 24.

Examples of structured matrices include FEM matrices on a structured grid. The matrix s3dkt3m2 is such a matrix, possessing relatively dense blocks along three diagonals. Another example is memplus, obtained from memory circuit simulation; see Figure 1.5 (left). For this matrix, the vector $x$ is consecutively accessed in up to four different areas, much like a four-diagonal matrix, with the only exceptions occurring on the first few rows and where straight lines of nonzeroes expand into triangle-like areas of nonzeroes. Reordering such a matrix will likely destroy this natural beneficial structure, and one-dimensional reordering will not yield improvements. For unstructured matrices, such as rhpentium (see Figure 1.5, right), large performance

gains are expected; if the connectivity contains enough hidden structure, the parts will become separated efficiently. On the other hand, if the columns are difficult to partition, many rows will be cut, thus resulting in large separators.

For a given $p$ and a given matrix, the gain of reordering will also depend on in which caches, if any, the input vector fits. Hence the results with different cache sizes for a fairly structured matrix (memplus), as well as an unstructured matrix (rhpentium) are discussed first. As mentioned earlier, no gain is expected by reordering an already favourably structured matrix. After a single bipartitioning, however, a structural improvement on the matrix memplus can be seen. In Figure 1.6 (left) the top half of the permuted matrix shows straight lines exactly corresponding to the beneficial original structure. The bottom half contains the less favourably structured parts, compressed together. At $p = 100$ (right), things have become far less favourably structured as the reordering method tries to compress the nonzeroes in smaller and smaller areas. Examining the unstructured matrix rhpentium, shown in Figure 1.7, a completely different behaviour appears. 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.

### 1.5.1 Results of cache simulation

The simulated cache use efficiency shown in Figures 1.8 and 1.9 reflects the expectations described above. The figures show the reduction of cache misses, with respect to the simulated cache size. For memplus, gains of about 2 percent are recorded for $p = 2$, while for $p = 100$ losses of the order of 10 percent can be seen. The miss ratios in Figures 1.8, 1.9 and Table 1.2 are calculated by dividing the number of cache misses for the reordered matrix by the number of cache misses for the original matrix. For rhpentium, large gains of up to 35 percent can be seen for all three data structures considered. 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 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 optimise for the larger caches, while subsequent refinements increase performance on the smaller (L1) caches.

Figure 1.9 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. Expected is 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 not much can be improved when most misses are inevitable.

Table 1.2 presents the same simulations as Figures 1.8 and 1.9, but now with a fixed cache size, a varying number of partitions, and a complete test set of matrices. Table 1.2 shows that for large matrices and larger $p$, the zig-zag variant of CRS is, on

| 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-2005 | 0.98 (C) | 0.94 (I) | 0.92 (Z) |
| cage14 | 1.02 (I) | 1.09 (Z) | 1.10 (I) |

Table 1.2: Simulated cache effect of reordering 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 out of 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, $s = 2^{15}, z = 64, k = 8$.

average, superior.

Comparing Table 1.1 and 1.2, the single-level cache simulation shows that matrices with already favourable structure indeed do not yield improved cache use after reordering. 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. 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, no gains were expected, and indeed, no change in cache efficiency is observed.

### 1.5.2 Timed SpMV multiplication experiments

The simulated gains should 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; what can be said is that the CPU has shorter data access times in 37 percent of the data accesses. To see what

the theoretical results obtained by simulation mean in practice, experiments on actual machines are needed.

Figures 1.10 and 1.11 show the gain in wall-clock timings of benchmark experiments for the smaller test matrices. Here, the ratio between the SpMV multiplication time for the reordered matrix and that for the original matrix is shown. 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 1.1, these test matrices have row and column dimensions much smaller than this size, causing reordering to alter mostly the L1 (and not L2) cache behaviour; hence cache effect enhancements translate to relatively small amounts of gain in execution time.

Figure 1.10 shows results for a straightforward SpMV implementation. For small matrices, reordering sometimes achieves modest gains; it almost never leads to losses. Exceptions are two small matrices, the rhpentium matrix and the rand10000 matrix, which showed significant gains 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 expected from the 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 affects behaviour not included in the cache model.

Figures 1.12 and 1.13 show results for much larger matrices. Partitioning is stopped at $p = 20$, since it 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. Indeed, 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 other link matrices, including the stanford matrix. Apparently, partitioning of the stanford-berkeley matrix is difficult, perhaps due to it describing two web subdomains, those of Stanford University and of the University of California at Berkeley, which may contain more interconnects than either domain alone.

The matrix cage14, where losses of more than $10$ percent are recorded, seems to be a hard case for improvement. A structure can be observed in all the cage matrices, which comes from the chosen numbering of states in the underlying Markov model used to study DNA electrophoresis, see [vHBB02]. Note that the losses reach their peak at $p = 4$, and that for $p \geq 4$ the losses decrease monotonically with $p$, indicating that the finer-grain structure of the matrix might eventually be exploited by reordering to improve the cache behaviour.

Results obtained using the OSKI auto-tuning library are shown in Figure 1.11. Note that OSKI uses its own optimised 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 a standard (I)CRS 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.

It is difficult for a straightforward CRS-based SpMV multiplication algorithm to compete with the optimised implementation from OSKI, which is always faster, except for the memplus matrix, as discussed earlier. When OSKI timings on the original matrix are compared to timings using (I)CRS or ZZ-ICRS on reordered matrices, reordering is more efficient for three out of the nine smaller matrices (excluding the memplus timings). For the larger matrices, this is the case for two out of four: stanford and wikipedia. In the case of stanford, timings 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, joining both methods by using OSKI on the reordered matrices yields further improvements with respect to OSKI on the original matrix in twelve out of the fourteen cases. The only exceptions are s3dkt3m2 and cage14, both cases where reordering did not gain anything or even caused performance loss compared to the original ordering.

Since the method is cache-oblivious, it should perform similarly on other architectures. To check this, experiments on the Dutch national supercomputer Huygens at SARA were also performed. Results should be similar to those on the Intel Core 2 machine, but performance could increase due to the presence of an L3 cache. Figure 1.14 shows wall-clock time results for the larger matrices processed on Huygens. Performance is similar on the stanford and wikipedia matrices, but note that already for $p = 2$ a 30 percent gain for the wikipedia matrix is obtained. This may be caused by the L3 cache which can store $4194304$ doubles and hence can contain the input and output vectors. The losses for the cage14 matrix are now less severe, but there no longer is monotonic speedup after $p = 4$; it may take a larger $p$ before the timings start to decrease on this architecture. ICRS on Huygens is frequently the fastest implementation and is noticeably faster than CRS. The stanford-berkeley matrix shows similar results as for the Intel architecture; and since the IBM Power6+ L2 cache is exactly of the same size as the Intel Q6600 L2 cache, the experiments on Huygens reveal no 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 1.3 presents the reordering time expressed in the number of SpMV multiplications. The SpMV multiplication time compared to is the time required for one SpMV *without* reordering. With increasing $p$, the number of multiplications required increases rapidly. Savings in SpMV multiplication time often decrease as $p$ increases, indicating 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 better-quality reorderings for the same number of

| 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 |
| wikipedia2005 | 2152 | 1992 | 2168 | 2570 | 7418 |
| cage14 | 4238 | 3987 | 3635 | 4583 | 5611 |

Table 1.3: Cost of reordering in terms of the number matrix multiplications on the original matrix. Here, $\epsilon = 0.1$. Construction times were measured on an Intel Core 2 (Q6600) machine.

SpMV multiplications may be obtained.

## 1.6 Conclusions

Introduced is the one-dimensional sparse 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. It is based on partitioning methods under load balancing constraints, as originally developed in the area of parallel matrix computations. Its central ideas are:

- 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, and so obtaining the Separated Block Diagonal (SBD) form, leading to a gradual transition between a cache filled with data from one column set ($\mathcal{V}_0$) to another set ($\mathcal{V}_1$) during the SpMV multiply;
- hypergraph partitioning to reduce the number of cut rows, using the $\lambda - 1$ metric, to prevent parts of cut rows from being cut further.

For obtaining the matrix reorderings, the hypergraph-based sparse matrix partitioner Mondriaan [VB05], in 1D (column direction) mode has been used. The reordering method does not depend on Mondriaan, however. Instead, one can use other hypergraph partitioners, such as PaToH [ÇA99], hMETIS [KK99], Zoltan [DBH+06], Monet [HMB00], and Par$k$way [TK04]. Using the parallel partitioner Zoltan, or Par$k$way, would enable a parallel reordering.

Experiments show 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 of efficiency is observed. The time needed to determine the reordering permutations can be amortised if the multiplication is carried out repeatedly, as happens in iterative linear system solvers and eigensolvers.

### 1.6.1 Future work

Although cache-oblivious matrix reordering may not always be as effective on small matrices when compared to cache-aware software such as OSKI [VDY05], especially on structured matrices, this is partly a consequence of not carrying through the partitioning till the end. 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 nonzeroes, nor the average number of nonzeroes per row. For this method to perform well in practice, further research is required to decrease the reordering times. Perhaps the largest difference with large $p$ and the situation here, is that load-balance no longer is a factor; partitioning without load-balancing thus is another possibility.

### 1.6.2 Additional results

Experiments were performed with different values for $\epsilon$ as well. These results showed that the choice of imbalance parameter $\epsilon$ 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. The theoretical number of cache misses was investigated as well, by using a custom cache simulator. This successfully showed that reordering leads to a lower number of cache misses in the cache model described in the introduction, precisely when the SpMV multiplication indeed executed faster, thus validating the method in an additional way. These results are compared with established methods; experiments with CRS and OSKI (Optimised Sparse Kernel Interface) [VDY05] were performed. Multiplications were done on the original as well as on the reordered versions, showing that in many cases OSKI was outperformed, but more importantly, also showing that both methods could be applied simultaneously to achieve larger performance boosts than either method alone. OSKI was integrated such that it always attempts a full tuning of the input matrix before multiplication. No hints were given to OSKI as to which structural properties of the input matrix

might be exploited; this might have resulted in optimisations of lesser quality, but maintained the assumption of matrix-obliviousness.

## Acknowledgements

The work presented in this chapter was supported by the Dutch supercomputing centre SARA in Amsterdam and the Netherlands National Computing Facilities foundation (NCF), who provided access to the Huygens supercomputer. Additionally, SARA provided technical assistance in using their system. The BSIK/BRICKS MSV1-2 program is also acknowledged for financial support of the research presented in this chapter.

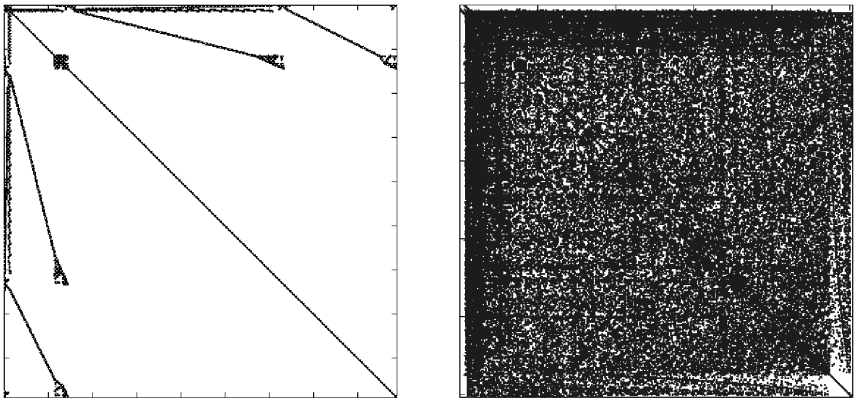


Figure 1.5: Plot of the original memplus (left) and rhpentium (right) matrices. Memplus has a favourable structure, whereas rhpentium looks unstructured.

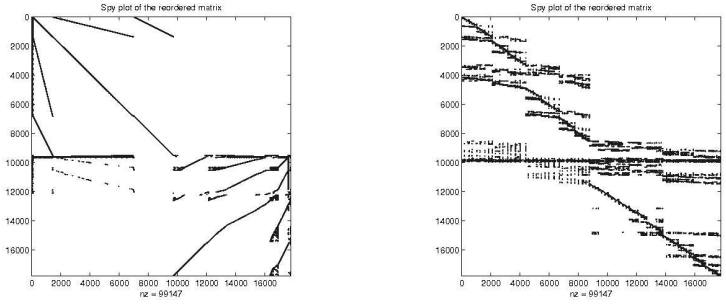



Figure 1.6: Plots of the memplus matrix; $p = 2$ (left) and $p = 100$ (right). The maximum load imbalance parameter $\epsilon$ was set to $0.1$.

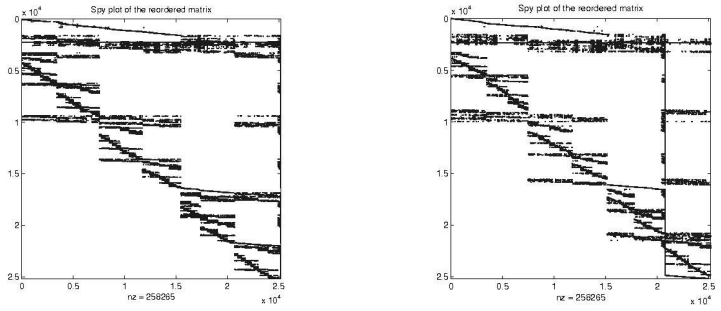



Figure 1.7: Plots of the rhpentium matrix for $p = 100$ (left) and $p = 400$ (right). Here, the maximum load imbalance parameter $\epsilon$ was set to $0.1$ as well.

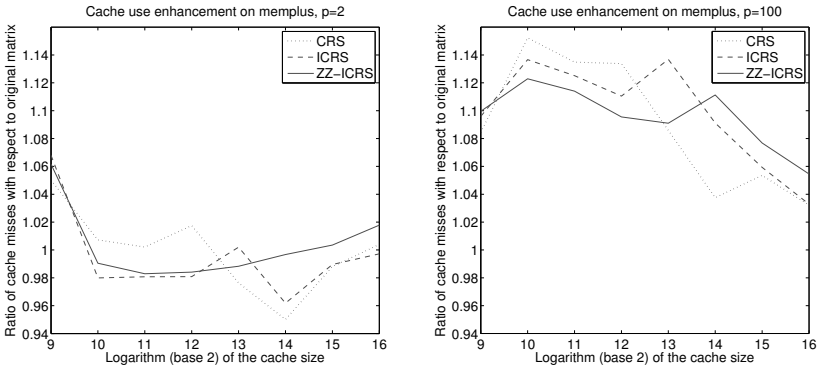



Figure 1.8: Simulated cache effect of reordering for the memplus matrix, plotted against different cache sizes. Given is the ratio between the number of cache misses for the reordered matrix and the number for the original matrix, for three data structures: CRS, ICRS, and ZZ-ICRS. The reordering was done with $p = 2$ (left), and $p = 100$ (right); both with $\epsilon = 0.1$. The cache line size is $z = 64$ bytes, so that $w = 8$. Assumed is an 8-way set-associative cache. Note that a cache size $s = 2^{15}$ bytes corresponds to an Intel Core 2 L1 cache.

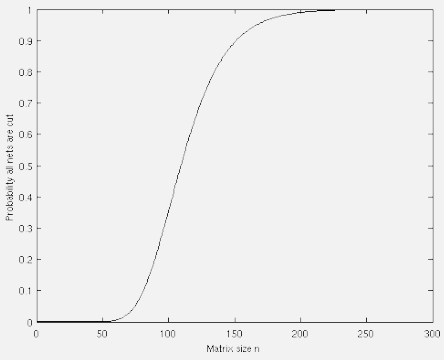
## *Partitioning random sparse matrices*

Random sparse matrices, although unstructured to the eye, form a separate category. In fact, for an arbitrary partitioning $\mathcal{V}_1$,$\mathcal{V}_2$ of $\mathcal{V}$, the probability that a net $n_i \in \mathcal{N}$ intersects with none of the columns corresponding to vertices in $\mathcal{V}_i$ is $(1-d)^{|\mathcal{V}_i|}$, where $d$ is the probability that a nonzero entry appears in the matrix. Thus the probability that a net ends up in $\mathcal{N}_c$ is

$$(1-(1-d)^{|\mathcal{V}_1|})(1-(1-d)^{|\mathcal{V}_2|}).$$

Assuming a perfect load balance, the probability that $n_i$ is in $\mathcal{N}_c$ is $p = (1-(1-d)^{n/2})^2$. The probability that at least $k$ of the $m$ hyperedges end up in $\mathcal{N}_c$ then is given by

$$\sum_{i=k}^{m} \binom{m}{i} p^i (1-p)^{m-i},$$




still under the assumption of a perfect load balance. The probability that all nets are cut, i.e., the above with $k = n$, is plotted above for various matrices sizes $n$, with $d = 0.1$. Plots for the probability that at least 95 per cent of the nets are cut ($k = \lfloor 0.95n \rfloor$) are similar, and the area in which few cut matrix rows are expected increases as the nonzero density $d$ decreases. Varying the nonzero density $d$, the smallest matrix size for which the probability that at least 95 per cent of nets are cut, is itself larger than 0.95, is as follows:

| $d$ | 0.01 | 0.02 | 0.03 | 0.04 | 0.05 | 0.1 |
|---|---|---|---|---|---|---|
| $n$ | 781 | 398 | 265 | 201 | 161 | 81 |

This is not to say heuristics cannot exploit specific instantiations of a random matrix; good partitioning software can consistently find better solutions than is expected. As the random matrix size increases, however, deviations in the matrix structure that make this possible will become more rare, leaving less room for heuristics to minimise the number of cut nets.

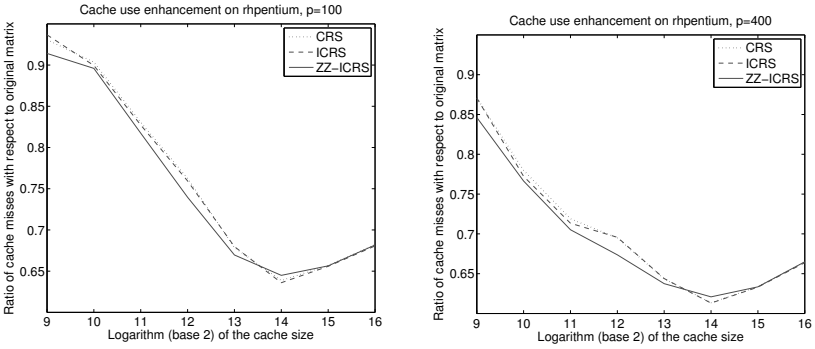



Figure 1.9: Simulated cache effect of reordering for the rhpentium matrix, plotted against different cache sizes. On the left, simulations with $p = 100$ are performed, while on the right $p = 400$ was used. In both cases, reordering was done with $\epsilon = 0.1$. Again, an $8$-way set-associative cache with $z = 64$ was simulated.

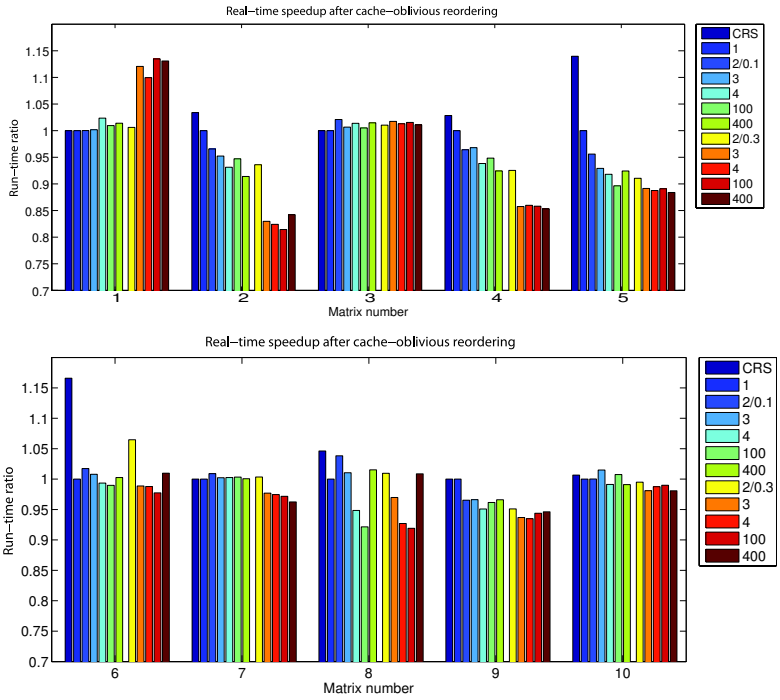



Figure 1.10: Wall-clock timings performed on an Intel Core 2 (Q6600) machine. The CRS bar denotes the timing of a straightforward CRS-based SpMV multiplication applied to the original matrix. The bar $p = 1$ corresponds to the best timing of CRS, ICRS, and ZZ-ICRS on the original matrix. The bars $p = 2/0.1, 3, 4, 100, 400$ show the results after reordering using $p$ parts, with $\epsilon$ 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: 1. memplus; 2. rhpentium; 3. s3dkt3m2; 4. rand10000; 5. fidap037; 6. lhr34; 7. rand50000; 8. nug30; 9. tbdlinux; 10. bmw7st1.

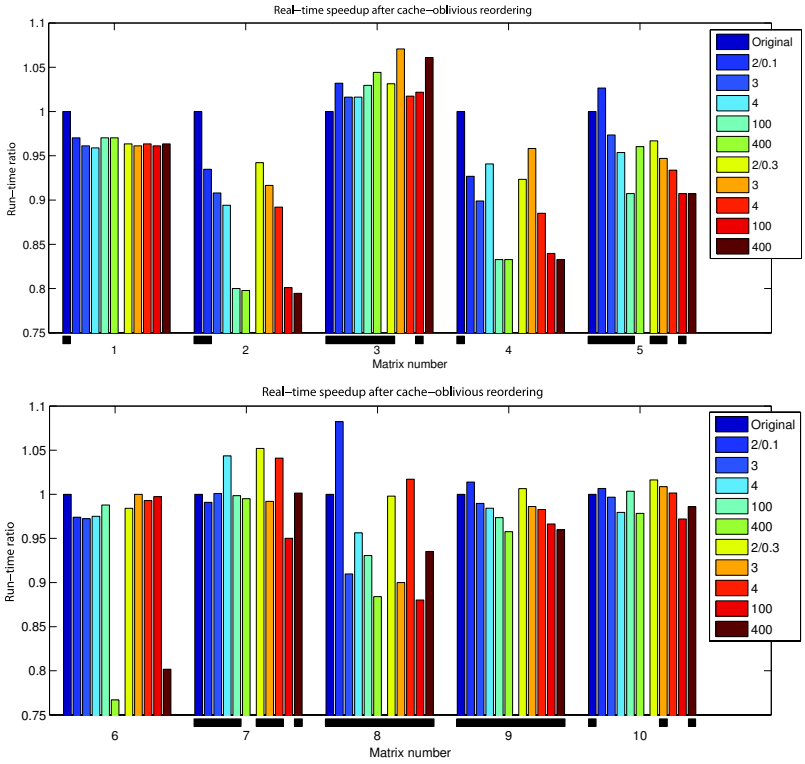



Figure 1.11: Wall-clock timings obtained by using the OSKI library for the SpMV multiplication, instead of plain data structure implementation. Note that OSKI sparse matrix storage is CRS-based. The cases where OSKI automatically applied cache-aware tuning are marked in black.

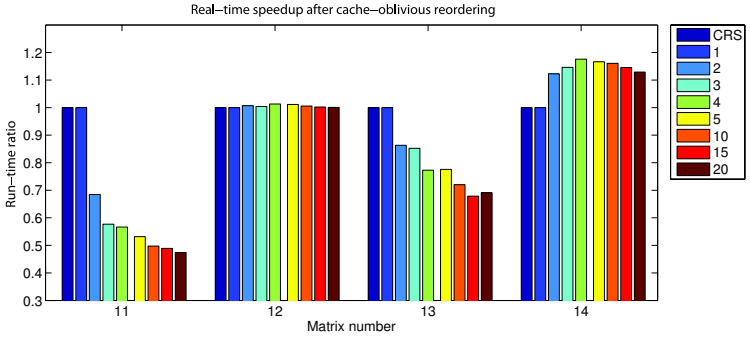



Figure 1.12: Wall-clock timings for the large matrices, where the input and output vector do not fit into the L2 cache. Interpretation is the same as for Figure 1.10, with $\epsilon = 0.1$ when the matrix reordering method is applied. The test matrices are: 11. stanford; 12. stanford-berkeley; 13. wikipedia-2005; 14. cage14.

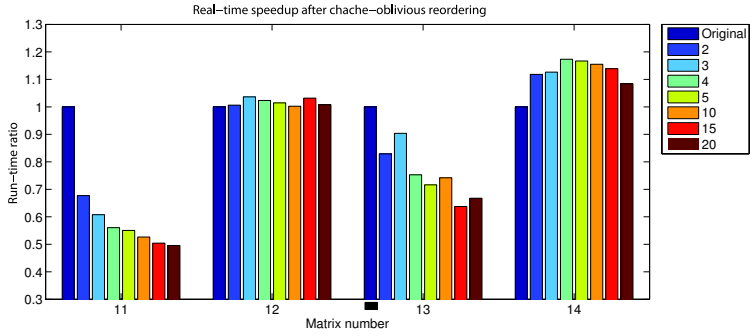



Figure 1.13: Wall-clock timings for the large matrices, but using the OSKI library.

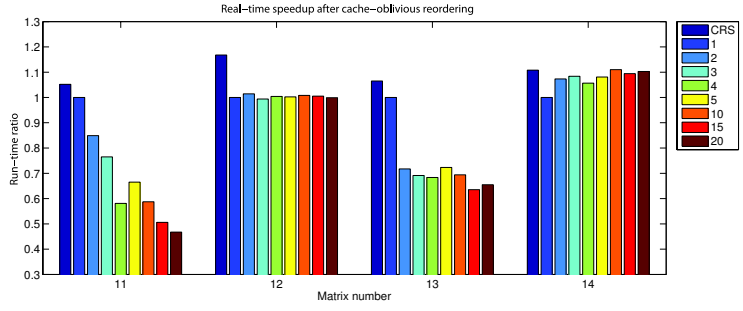



Figure 1.14: Experiments similar to Figure 1.12, but run on the Dutch national supercomputer Huygens.

# Chapter 2

# An extension to two dimensions

In the previous chapter a one-dimensional cache-oblivious sparse matrix–vector multiplication scheme, based on one-dimensional sparse matrix partitioning, was presented. Partitioning is often used in distributed-memory parallel computing for the SpMV multiplication, an important kernel in many applications. A logical extension is to move towards using a two-dimensional partitioning, which is done in this chapter. Still, only row and column permutations are allowed on the sparse input matrix. This extension requires a generalisation of the CRS data structure, to a block-based data structure. Experiments performed on three different architectures show further improvements of the two-dimensional method compared to the one-dimensional method, especially in those cases where the one-dimensional method already provided significant gains. The largest gain obtained by our new reordering is over a factor of 3 in SpMV speed, compared to the natural matrix ordering.

This chapter is based completely on the work [YB11c]:

- A. N. Yzelman and Rob H. Bisseling, *Two-dimensional cache-oblivious sparse matrix–vector multiplication*, Parallel Computing (2011), to appear.

## Related work

Since we are now making our way into 2D techniques for improving cache locality, other recent works become relevant, beyond those referred to in Chapter 1. Previously, cache locality in the input vector was improved, and the locality of the output vector was never in question thanks to the (ZZ-)CRS ordering imposed on the nonzeroes, which results in linear access of the output vector. This also enabled the use of auto-tuning methods like OSKI, introduced by Vuduc et al. [VDY05], in conjunction with our reordering method. A 2D method, however, tries to further increase locality in the input vector direction while actively attempting to preserve locality in the output vector direction, in the hope that the obtained locality in both dimensions can

outperform locality induced in only one of the dimensions. The objective is thus to minimise the sum of cache misses within both the input and output vectors, and this entails breaking the linear access to the output vector.

In dense linear algebra, the 2D approach has been successfully applied by *blocking*, the process of dividing the matrix into submatrices (or a hierarchy thereof) of appropriate sizes so that matrix operations executed in succession on these submatrices are faster than if they were performed on the single larger sparse matrix [GG02, VS02, WPD01]. By using a Morton ordering (i.e., a recursive Z-like ordering) [Mor66], the blocks can be reordered to gain even more efficiency; additionally, the Morton ordering can also be used on high-level blocks, while the low-level blocks still use row-major data structures such as standard CRS. This gives rise to so-called hybrid-Morton data structures which, like the method presented here, enable the use of specialised BLAS libraries on the low-level blocks; see Lorton and Wise [LW07]. Similar techniques for sparse matrices include sparse blocking into very small blocks [NVDY07], and using the Hilbert space-filling curve on the nonzeroes of sparse matrices [HLP07, YB11a]. The method presented in the next section is much in the same locality-enhancing spirit, even though its approach is different.

## 2.1 The generalised two-dimensional case

The straightforward extension of the original scheme to 2D is natural when employing the fine-grain model [ÇA01] (see also Section I.2). If $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ is the fine-grain representation of $A$, there are in essence two types of hyperedges: those corresponding to matrix rows, and those corresponding to matrix columns (i.e., $\mathcal{N} = \mathcal{N}^{\text{row}} \cup \mathcal{N}^{\text{col}}$). For each net, the connectivity $\lambda_i$ over a partitioning $\mathcal{V}_0, \ldots, \mathcal{V}_{p-1}$ of $\mathcal{V}$ can still be defined as before. In particular, in the case of a single bipartitioning, this enables us to define the set of cut row nets $\mathcal{N}_c^{\text{row}} \subset \mathcal{N}$, consisting of the row nets with connectivity larger than one, as well as the set $\mathcal{N}_c^{\text{col}}$, similar to the 1D case. Row and column permutations for $p = 2$ can then be used to bring the arbitrary sparse input matrix into the form depicted in Figure 2.1 (left), the *doubly separated block diagonal* (DSBD) form.

Note that there are now five different separator blocks, namely $\mathcal{N}_c^{\text{row}} \cap \mathcal{N}_c^{\text{col}}$, $\mathcal{N}_c^{\text{row}} \cap \mathcal{N}_\pm^{\text{col}}$, and $\mathcal{N}_\pm^{\text{row}} \cap \mathcal{N}_c^{\text{col}}$. These together form a *separator cross*, coloured red in Figure 2.1 (left). An example of a matrix in DSBD form obtained using this fine-grain scheme can be found in Figure 2.2 (lower-left).

This 2D scheme can be applied recursively. However, using a (ZZ-)CRS data structure will result in *more* cache misses for $p > 2$ due to the column-wise separator blocks, if the separator blocks are relatively dense; see Figure 2.1 (right). Nonzeroes thus are better processed block by block in a suitable order, and the data structure used must support this. These demands are treated separately in Section 2.1.1 and Section 2.1.2, respectively.

A separator tree can be defined in the 2D case, similar to the 1D case, but now with nodes containing nets corresponding to both matrix rows and columns. Internal nodes

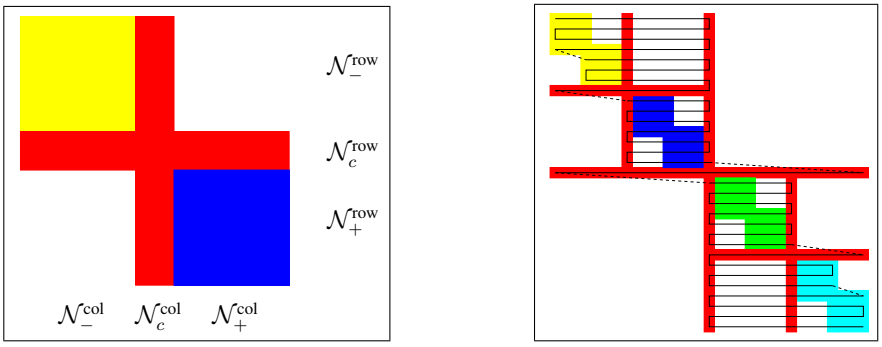



Figure 2.1: Schematic view of a 2D matrix reordering using the fine-grain model, for $p = 2$ (left) and $p = 4$ (right). The figure on the right side also includes the ZZ-CRS ordering curve.

in the tree contain both cut rows and columns belonging to the separator crosses. Leaf nodes correspond to the $p$ non-separator blocks. An example of a separator tree in the case of $p = 4$ is shown in Figure 2.3.

### 2.1.1 SBD block order

Figure 2.4 shows various ways of ordering the DSBD blocks. The main objective is to visit the same regions of the input and output vectors as few times as possible. Accesses in the vertical direction represent writes on the output vector, whereas horizontal accesses represent reads on the input vector. Write accesses are potentially more expensive; the block ordering avoiding the most unnecessary irregular accesses in the vertical direction thus theoretically performs best. From the orderings in Figure 2.4, this best performing block order is ZZ-CCS.

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

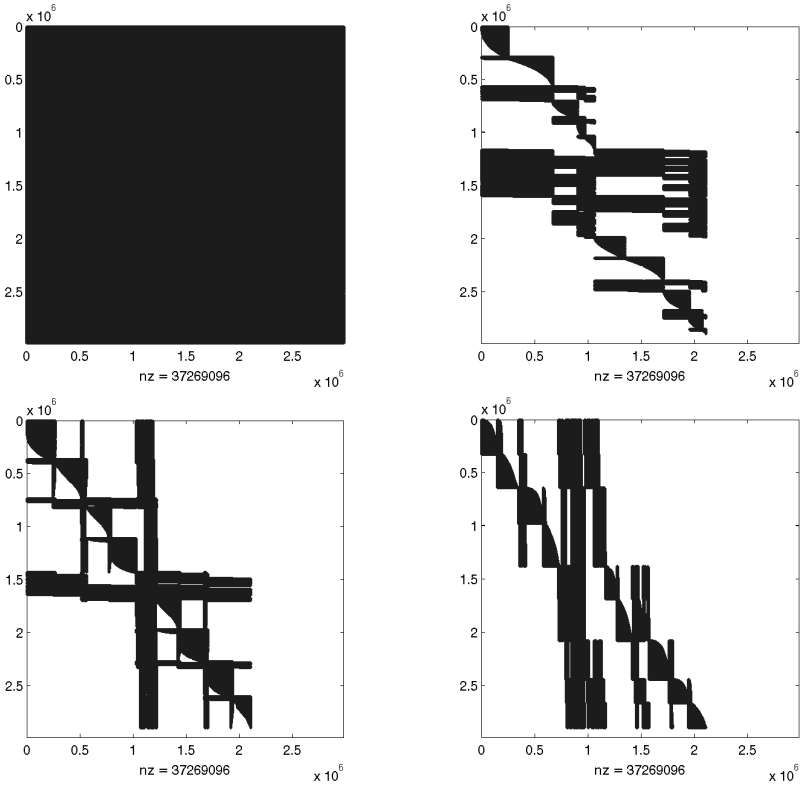



Figure 2.2: Plots of the wikipedia-2006 matrix, partitioned and reordered to (D)SBD form using the Mondriaan software [VB05]. The matrix has $2983494$ rows and columns, and $37269096$ nonzeroes. The upper-left plot shows the highly unstructured original link matrix. This matrix appears to be dense because of the size of the nonzero markers and the rather even spread of the nonzeroes: in fact every matrix row contains $12.5$ nonzeroes on average. The upper-right plot shows the result after 1D partitioning with $p = 10$. The lower-left picture shows the result using the 2D fine-grain scheme with $p = 8$, and on the lower-right, the 2D Mondriaan scheme with $p = 9$ was used. The load imbalance parameter is set to $0.1$ in all cases. Empty rows and columns correspond to web pages without outgoing or incoming links, respectively.

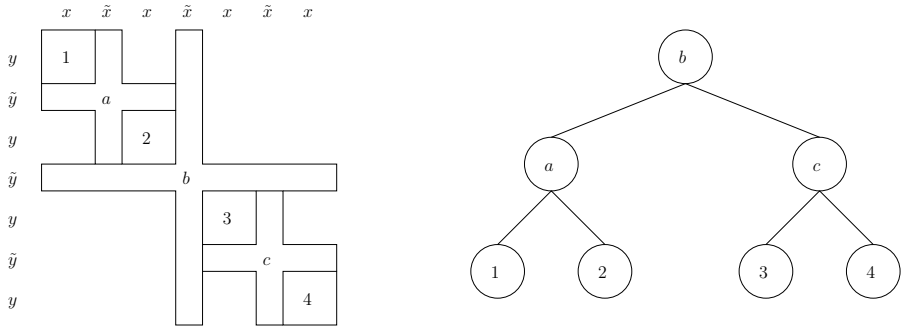


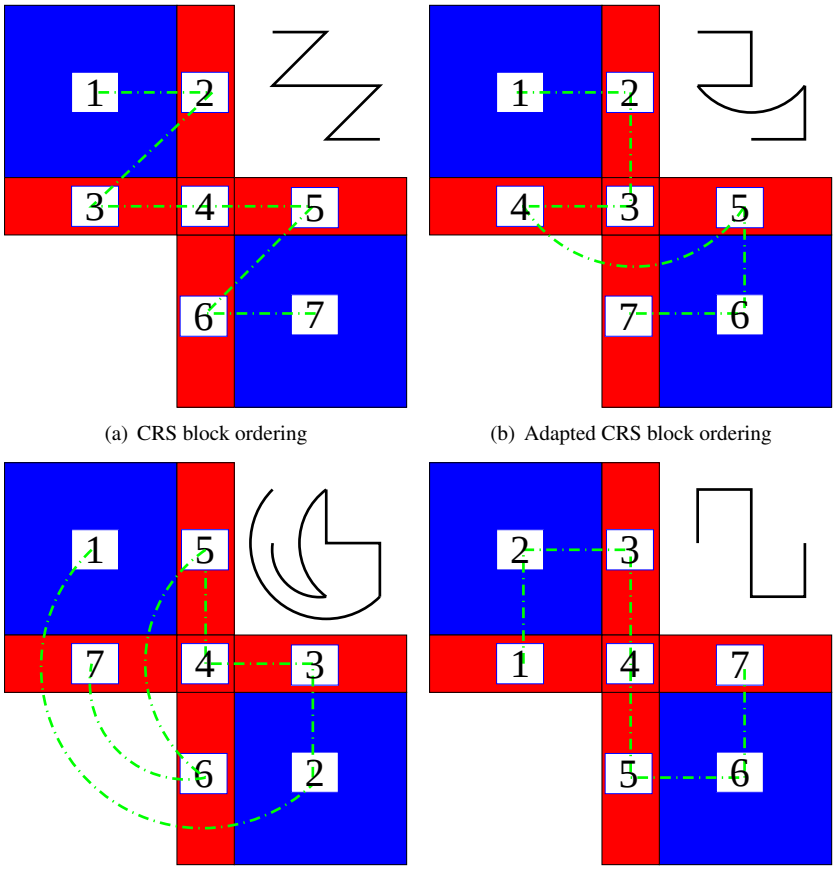

Figure 2.3: Illustration of a DSBD form with $p = 4$, and its corresponding separator tree. The sizes $x, y$ of the $y \times x$ non-separator blocks (1, 2, 3, 4) are also shown in the picture, as are the small widths of the separator cross, given by $\tilde{y}$ and $\tilde{x}$.

### 2.1.2 Block data structures

Two data structures will be introduced here: Bi-directional Incremental CRS (BICRS) and a simple block-based data structure (Block). With the ICRS storage scheme [Kos02], the difference between a nonzero column index and that of its preceding nonzero is stored, instead of the column indices themselves. While multiplying, a row change is indicated by overflowing the column index by storing too large differences, so that subtracting the number of columns $n$ afterwards yields the starting index of the input vector for the next row. This next row is determined by using a similarly constructed row difference array. See also Section 1.1 for more details. Note that ICRS jumps only to the nonempty rows, and therefore avoids unnecessary overhead. This is particularly useful for the separator blocks we encounter after 2D partitioning; there, many empty rows (columns) are found in the case of vertical (horizontal) separator blocks.

The Bi-directional Incremental CRS [YB11c] data structure is a simple extension to ICRS, which allows any (non-CRS) ordering of the nonzeroes by allowing negative increment values. Trivially, negative column increments enable jumping back and forth within a single row. Overflows in the column direction still signal row changes, and by allowing negative row increments, bi-directional row changes are made possible as well. Each column overflow implies a row jump, and each such jump has an instruction and storage overhead; note that hence the orientation (BICRS or BICCS) of the data structure still matters. The gain is that this additional overhead is only dependent on the number of row jumps $j$, and not directly on the number of parts $p$. Furthermore, we have that $m \leq j \leq$ nz, and the performance of BICRS is guaranteed to be in between that of ICRS and the Triplet data structure.

Another option for storing nonzeroes in a block-based order, is to store each matrix block in a separate (I)CRS or (I)CCS data structure. Upon performing an SpMV multiplication, the multiplication routines of the separate data structures are called




(a) CRS block ordering

(b) Adapted CRS block ordering

(c) Separators-last block ordering

(d) ZZ-CCS block ordering

Figure 2.4: Various possible DSBD orderings for SpMV multiplication for $p = 2$. The black curve in the upper right corners of the panels is included to highlight the curve trajectory.

in the order defined by the block order, on the same input and output vectors; this straightforwardly yields the *simple block-based data structure*. However, calling several multiplications in sequence this way incurs some overhead: when a switch between blocks is made, the pointers to the input and output vectors are reset to the location of the first nonzero of the next block, and only then the actual SpMV kernel is called. Also, the storage requirements increase with $p$: when using non-incremental CRS/CCS, storage scales as $\mathcal{O}((m+n)\log_2 p)$ (see Section A.1.6), and when using the incremental ICRS/ICCS data structure, storage is in between $2\text{nz} + m$ (best case) and 3nz (worst case). Hence the gain with respect to cache efficiency of using this simple block-based structure must be larger than the penalty incurred by this additional overhead; and this overhead can be reduced by lowering the number of blocks required. Instead of this simple block-based data structure, more advanced ideas may be exploited here, such as a scheme which also compresses index (or increment) values as presented by Buluç et al. [BFF+09]; although some additional effort would be required to make this work within our scheme.

## 2.2 Cache performance in recursion

In this section, the cache behaviour of the SpMV kernel for different block orderings will be analysed. The symbols $x$ and $y$ will not directly refer to the input or output vectors in this section, instead they are used to indicate the row-wise and column-wise sizes of the diagonal submatrices appearing in the DSBD form; see also Figure 2.3 (left). Some simplifying assumptions are made:

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

Cache misses are calculated by analysing the binary separator tree, starting with the root node, which contains the largest separator cross (spanning the entire sparse matrix). The two children of the root node contain the remaining two sparse blocks

that the partitioner has been recurring on. Only non-compulsory cache misses will be counted, since the compulsory ones are the same for all orderings. The analysis only pertains to the input and output vector data, since the matrix data are used only once and hence cannot benefit from the cache.

The number of non-compulsory cache misses can be expressed as a function on the root node, which recurs on its two children, et cetera. In fact, only the height of the nodes will be required to calculate the cache misses: for each block ordering a function $f(i)$ will be constructed, which gives an upper bound on the number of non-compulsory cache misses for a node with height $i + 1$. The height of the leaves is 0 by definition, hence an internal node of height 1 corresponds to a subpartitioning with $p = 2$, which is taken as the base case $f(0)$. See also Figure 2.3. Note that by the power-of-two assumption, the separator tree is complete.

A single function $f$ cannot be used to predict the cache misses for each node in every situation. Cache behaviour may differ between nodes as in some cases the elements in the input vector were already processed, thus making all accesses in the horizontal direction non-compulsory. This happens for example in Figure 2.4a, when block 3 is processed. In general, there are four different possibilities, each modelled by a separate function:

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

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

### 2.2.1 CRS block ordering

First, the CRS block ordering (Figure 2.4a) is analysed. Assume that no previous accesses have been made to the vectors. For block 1, $x$ accesses to the input vector are needed and $y$ to the output vector. Since this is the first time access is needed, both accesses are compulsory cache misses. For block 2, $y$ data are kept in cache, and $\tilde{x}$ are brought in for the first time, hence causing $\tilde{x}$ compulsory misses. For block 3, $x$ data are brought into cache again, with $x$ non-compulsory misses, and $\tilde{y}$ compulsory misses. For block 4, $\tilde{x}$ non-compulsory accesses are made, and $\tilde{y}$ data remain in cache. For block 5, there are $x$ compulsory misses, and $\tilde{y}$ data remain in cache. For block 6, $\tilde{x}$ non-compulsory misses, and $y$ compulsory ones occur. The final block incurs $x$ non-compulsory misses while $y$ data elements remain in cache. Summing the non-compulsory misses for all the blocks, the value of $f_0(0)$ becomes $2(x + \tilde{x})$. Now, assume all vector data have been accessed before. This means that

all $2x + \tilde{x} + 2y + \tilde{y}$ compulsory misses become non-compulsory, giving a total of $f_3(0) = 4x + 3\tilde{x} + 2y + \tilde{y}$ non-compulsory misses.

If the block ordering is applied recursively to obtain $f_0(1)$, blocks 1 and 7 are further refined. Assuming no previous accesses have been made, the following number of non-compulsory misses for blocks 1 to 7 are made: $f_0(0)$, $2y + \tilde{y}$, $2x + \tilde{x}$, $\tilde{x}$, $0$, $\tilde{x}$, $f_3(0)$; see also Figure 2.3. This yields a total of $f_0(1) = f_0(0) + f_3(0) + 2(x + y) + 3\tilde{x} + \tilde{y}$. A similar analysis can be made for $f_3(1)$. Note that the expressions $f_1$ and $f_2$ do not appear here due to the way blocks 1 and 7 are accessed. In full recursion, the upper bounds become:

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

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

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

direct formulae can be obtained:

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

so the direct formula for $f_0$ can be derived:

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

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

### 2.2.2 Zig-zag CCS block ordering

This order corresponds to the ordering shown in Figure 2.4d. Analysing this form, all four functions $f_0, \ldots, f_3$ are required as follows: $f_0(0) = 2\tilde{y}$, $f_1(0) = 2(x + \tilde{y}) + \tilde{x}$, $f_2(0) = 2y + 3\tilde{y}$, $f_3(0) = 2(x + y) + \tilde{x} + 3\tilde{y}$, and

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

Note that:

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

A direct formula for $f_3$ can be obtained:

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

and similarly for the above difference formulae,

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

This leads us to the following final form:

$$f_0(i) = f_3(i) - (f_3 - f_0)(i) = i2^{i+1}(x+y) + ((i-1)2^{i+1} + 2)\tilde{x} + (i+1)2^{i+1}\tilde{y}. \quad (2.1)$$

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

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

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

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

## 2.3 Using Mondriaan partitioning

Modelling a sparse matrix as a hypergraph using the 2D fine-grain model has two drawbacks: the increased size of the hypergraph compared to the simpler 1D row-net model, and the large number of blocks after permutation. The larger hypergraph leads to an increased partitioning time, hopefully justified by the increased quality of the 2D partitioning, whereas the large number of separator blocks incurs additional overhead as was discussed in Section 2.1.2. An alternative 2D method exists, the *Mondriaan scheme*, which results in less partitioning time and fewer separator blocks. It is implemented in the Mondriaan sparse matrix partitioner software [VB05], and combines two 1D methods as follows.

Apart from the row-net model, a column-net model can also be defined, identical to the row-net model but with the roles of rows and columns reversed: each row corresponds to a vertex in the hypergraph, and each column to a net. A cheaper way of obtaining a 2D partitioning for $p > 2$ then is to use a partitioner as described in Section 1.2, with the following modification: during each iteration, both the hypergraphs corresponding to a row-net and column-net representation are partitioned, and the solution yielding the lowest cost, as given by the $\lambda - 1$ metric, is chosen. In the next iteration, again both models are tried and the best is chosen; hence splits in both dimensions (row-wise and column-wise) are possible during partitioning, but never both during the same iteration. An example of a DSBD partitioning obtained using this scheme is shown in Figure 2.2 (lower-right).

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

It is worthwhile to exploit the form of the 2D DSBD ordering in the special case of bipartitioning by the row-net or column-net model. The 1D row-net model yields the picture in Figure 2.1 (left) with the centre block and the two vertical rectangular blocks removed from the separator cross (see also Figure 1.2 and 1.4). The column-net model is similar, having instead the centre block and the two horizontal rectangular blocks removed. As such, using the Mondriaan scheme instead of the fine-grain scheme in full recursive bipartitioning, nearly halves the number of blocks, from $p + 5(p - 1) = 6p - 5$ to $p + 2(p - 1) = 3p - 2$ blocks. Hence, if the quality of partitioning by the fine-grain and Mondriaan schemes is similar, the Mondriaan scheme is expected to perform better because there are fewer blocks.

## 2.4 Experimental results

Experiments for SpMV multiplication have been performed on three different architectures. The first is a supercomputer, called Huygens, which consists of $104$ nodes

each containing 16 dual-core IBM Power6+ processors. The others are an Intel Core 2 Q6600 machine, and an AMD Phenom II 945e; see Table 1 and the various figures in Section I.1.3 for extended details. On Huygens, one node was reserved to perform the sequential SpMV multiplications (on a single core), without interference from other processes. Since a Power6+ processor has an L1 cache of 64kB (data) per core, an SpMV multiplication on a matrix with $m + n = 8192$ would fit entirely into the L1 cache, assuming the vector entries are stored in double precision. The same applies with $m + n = 524288$ for the L2 cache and $m + n = 4194304$ for the L3 cache. On the Q6600, a matrix with dimensions $m + n$ exceeding $524288$ has to revert to main memory to store the required vectors, while matrices with dimensions lower than $4096$ fit into L1 cache entirely. The combined size for which the L1 cache of the AMD processor is exceeded is $8192$, that for the L2 cache is $65536$, and for the L3 cache it is $786432$.

For the matrix reordering by 2D methods, the Mondriaan sparse matrix partitioning software[1] [VB05] was used. The 1D method from the previous chapter employed an earlier version of Mondriaan; the version used here generally is faster while generating partitionings of slightly better quality. Three datasets have been constructed using the Mondriaan partitioner beforehand; the matrices in Table 2.1 were partitioned using a 1D (row-net) scheme, the 2D Mondriaan scheme, and the 2D fine-grain scheme. These are the same matrices used in Chapter 1, with the addition of a 2006 version of the wikipedia link matrix and the GL7d18 matrix. Most matrices from the dataset used are available through the University of Florida sparse matrix collection [Dav11]. In all cases, the load-imbalance parameter of the partitioner was set to $\epsilon = 0.1$, and the default options were used (except those that specify the hypergraph model and permutation type). The smaller matrices were partitioned for $p = 2, \ldots, 7, 10, 50, 100$, whereas the larger matrices were partitioned for $p = 2, \ldots, 10$. The construction times required for the smaller matrices are of the order of a couple of minutes; e.g., the most time-consuming partitioning, that of the matrix s3dkt3m2 in 2D by the fine-grain model with $p = 100$, takes $8$ minutes. This time typically decreases by more than a factor of two when partitioning in 1D mode, for instance taking 3 minutes for s3dkt3m2 with $p = 100$. The Mondriaan scheme results in partitioning times usually between these two, although in the particular case of s3dkt3m2, it is actually faster with 1 minute. The partitioning time for the larger matrices using the fine-grain scheme measures itself in hours: stanford with $p = 10$ takes about 2 hours, while wikipedia-2005 takes about 7 hours and wikipedia-2006 23 hours. In 1D, running times decrease by more than an order of magnitude, e.g., to $4$ minutes for stanford, half an hour for wikipedia-2005, and one hour for wikipedia-2006 with $p = 10$. The Mondriaan scheme, again with $p = 10$, runs in $5$ minutes for stanford, 7 hours for wikipedia-2005, and 21 hours for wikipedia-2006.

The SpMV multiplication software[2] was compiled by the IBM XL compiler on Huygens, using auto-tuning for the Power6+ processor with full optimisation en-

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

[2] Available at: http://albert-jan.yzelman.net/software/

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

Table 2.1: The matrices used in our experiments. The matrices are grouped into two sets by relative size, where the first set fits into the L2 cache, and the second does not. An S (U) indicates that the matrix is considered structured (unstructured). Symmetry is decided only on the basis of the nonzero structure; structurally symmetric matrices are indicated as symmetric in this table.

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

All combinations of data structures (plain CRS, plain ICRS, Block-based) and partitioning schemes (Row-net, Mondriaan, Fine-grain) are considered. In fact, the block-based data structure comprises several methods, using different block orderings (CRS, Adapted CRS, Separators-last, Zig-zag CCS, and variants) and data structures (simple block-based with ICRS/ICCS, and Bi-directional ICRS); see Figure 2.4 and Section 2.1.2. Note that plain CRS and plain ICRS do not make use of sparse blocking based on the separator blocks, and that the simple block-based data structures use ICRS on the diagonal blocks and the vertical separator blocks, and ICCS on the horizontal separator blocks. Any other combination of data structures (non-incremental,

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

or ICRS and ICCS switched) results in uncompetitive strategies. In total, 36 different methods per reordered matrix are tested, resulting in over 4000 individual experiments per architecture.

Of the partitioning schemes, the row-net scheme corresponds to a 1D partitioning, and the Mondriaan and fine-grain schemes to 2D partitioning. The 1D method combines the plain (I)CRS data structure with the row-net partitioning scheme. For the full 2D method, various block orders have been tested with the block-based data structure. The CRS and ICRS data structures combined with 2D partitioning have been included since they do not incur any overhead with increasing $p$ at all, and thus can potentially overtake the block-ordered data structures, especially when there are few nonzeroes in the vertical separator blocks. For the Intel Q6600 architecture, results using OSKI [VDY05] with 1D partitioning are again included to provide an easy comparison with the earlier results; the timings obtained by applying OSKI to the original matrices are included as well. In all cases, OSKI was forced to perform full aggressive tuning, but no hints as to any matrix structure were provided.

For each dataset, the best timing and its corresponding number of parts $p$ and data structure is reported. Table 2.2 shows the results for the IBM Power6+ architecture, Table 2.3 for the Intel Q6600 which includes the OSKI results, and Table 2.4 for the AMD 945e. As in the 1D case, most structured matrices are hardly, or even negatively, affected by the reordering scheme, both with 1D and 2D partitioning. Among the two exceptions is the nug30 matrix, in which the 1D scheme gains 9 percent and the Mondriaan scheme 7 percent on the Huygens supercomputer; the fine-grain scheme and the 1D scheme with blocking do not perform well there. For the AMD platform, the 1D scheme performs similarly while the gains for the Mondriaan scheme increase to 17 percent. The other partitioning schemes are more effective as well. On the Intel Q6600, comparing to the OSKI baseline, the 1D scheme gains up to 13 percent, while for the other three partitioning schemes gains go up to 26 percent. The second exception is the s3dkt3m2 matrix, on which gains of 6 percent appear when using the 1D blocking scheme or the Mondriaan scheme on the IBM Power6+; these gains increase to 8 percent for the AMD processor. The optimisations by OSKI are superior to those of our reordering method, however, and only slowdowns are reported for the Intel processor. Compared to a plain (I)CRS baseline, the gain would be 17 percent for the Intel processor, versus the 22 percent for OSKI on this structured matrix.

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

| Matrix | Natural | | 1D | | | 1D & Blocking | | |
|---|---|---|---|---|---|---|---|---|
| fidap037 | 0.12 | ICRS | *0.11* | ICRS | (100) | 0.12 | Block CRS | (2) |
| memplus | 0.31 | CRS | 0.31 | ICRS | (4) | 0.33 | Block CRS | (2) |
| rhpentium | 0.91 | ICRS | 0.65 | ICRS | (50) | 0.88 | Block CRS | (100) |
| lhr34 | 1.37 | ICRS | 1.36 | ICRS | (5) | 1.37 | Block CRS | (2) |
| nug30 | 5.35 | CRS | *4.85* | ICRS | (3) | 5.36 | Block CRS | (6) |
| s3dkt3m2 | 7.81 | CRS | 7.85 | CRS | (3) | 7.29 | Block CRS | (2) |
| tbdlinux | 6.43 | CRS | 6.09 | ICRS | (5) | *5.03* | Block CRS | (4) |
| stanford | 19.0 | CRS | 9.9 | CRS | (10) | 9.5 | Block CRS | (9) |
| stanford_berkeley | 20.9 | ICRS | 20.1 | ICRS | (4) | 19.3 | Block CRS | (9) |
| cage14 | *69.4* | CRS | 75.5 | ICRS | (2) | 76.5 | Block CRS | (2) |
| wikipedia-2005 | 249 | CRS | 155 | CRS | (10) | 142 | Block CRS | (9) |
| GL7d18 | 600 | CRS | 401 | CRS | (9) | *387* | Block CRS | (7) |
| wikipedia-2006 | 688 | CRS | 378 | CRS | (9) | 302 | Block CRS | (9) |

| Matrix | 2D Mondriaan | | | 2D Fine-grain | | |
|---|---|---|---|---|---|---|
| fidap037 | *0.11* | ICRS | (50) | *0.11* | ICRS | (100) |
| memplus | *0.30* | ICRS | (2) | 0.31 | ICRS | (3) |
| rhpentium | *0.63* | ICRS | (50) | 0.64 | ICRS | (50) |
| lhr34 | *1.34* | Block ZZ-CRS | (3) | 1.36 | Block ZZ-CCS | (5) |
| nug30 | 4.94 | CRS | (3) | 5.24 | Block SepLast | (3) |
| s3dkt3m2 | *7.27* | Block ZZ-CCS | (3) | 7.50 | Block ACRS | (3) |
| tbdlinux | 5.43 | Block ZZ-CRS | (7) | 5.43 | Block ZZ-CCS | (100) |
| stanford | *9.35* | CRS | (8) | 9.73 | CRS | (10) |
| stanford_berkeley | *19.2* | Block ACRS | (4) | 19.4 | Block SepLast | (9) |
| cage14 | 74.4 | ICRS | (2) | 75.1 | ICRS | (2) |
| wikipedia-2005 | *116* | CRS | (8) | 124 | Block ACRS | (8) |
| GL7d18 | 392 | Block ZZ-CRS | (10) | 435 | Block SepLast | (3) |
| wikipedia-2006 | *256* | CRS | (9) | 268 | Block ZZ-CCS | (8) |

Table 2.2: Time of an SpMV multiplication on the naturally ordered matrices, as well as on reordered matrices, both in 1D and 2D, on an IBM Power6+ architecture. Time is measured in milliseconds, and only the best average running time among the different numbers of parts $p$ and the different data structures used, is shown. The specific data structure used is reported directly after the best running time, and the number of parts $p$ used is shown between parentheses. In the 1D & Blocking category, the Block CRS is the only block order experimented with on this architecture. For each sparse matrix, the best timing among the various schemes is printed *italic*. For the smaller datasets (above the horizontal separator line), the number of parts used was $p = 2, \ldots, 7, 10, 50, 100$; for the larger ones (below the separator line), it was $p = 2, \ldots, 10$.

take the overhead from block-based data structures, even for large $p$. On the IBM Power6+, this situation is reversed. For the Mondriaan scheme, larger and unstructured matrices prefer direct CRS data structures over block-based data structures.

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

In principle, the results could differ when taking into account all different $p$ (and not only the best choices) for all partitioning schemes to find the best performing block orders, instead of using only the tables presented in this section. Doing this, on Huygens, we see the Adapted orderings appear most often, followed by the Zig-zag orderings. Looking at all experiments for the other two architectures reveals an interesting result: when using BICRS as a block-based data structure, there is a clear preference for the Block CRS ordering. Correcting for this bias by looking only at the simple block-based data structure, revealed no apparent preference for either Adapted or Zig-zag orderings; hence the conclusions remain somewhat in agreement with the theoretical results from Section 2.2, as indeed the Zig-zag and Adapted orderings are preferred over Block CRS and the SepLast orderings.

Considering instead which block data structures are used, on the Power6+, the BICRS data structure is completely uncompetitive compared to the simple block data structure. This is in direct contrast to the Q6600 and 945e processors, where with effective 1D and Mondriaan partitioning on the larger matrices, the BICRS data structure consistently outperforms the simple block data structure. When the number of separator blocks increases, however, such as happens with 2D fine-grain partitioning, the simple block data structure becomes more competitive.

Tests on the larger matrices display a more pronounced gain for the 2D methods. The matrices where the 1D partitioning was not effective, namely stanford_berkeley and cage14, perform slightly better in 2D, and the gains (or losses) remain limited. When partitioning did work in the original experiments, however, two-dimensional partitioning works even better, for nearly all of the test instances. It is also interesting to see that 1D partitioning combined with blocking outperforms the original 1D reordering method for larger $p$ (for the Intel architecture as well, although this is obscured from Table 2.3 by the additional gains obtained with OSKI). For the largest matrices from the whole test set, typically the Mondriaan scheme performs best. After that comes the fine-grain scheme, then followed by the blocked 1D method, and finally the original 1D method; this was also the case for the smaller matrices, with relatively few exceptions.

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

| Matrix | Natural | | 1D & OSKI | | | 1D & Blocking | | |
|---|---|---|---|---|---|---|---|---|
| fidap037 | 0.14 | CRS | 0.13 | ICRS | (100) | 0.15 | Block SepLast | (3) |
| memplus | 0.35 | CRS | *0.28* | CRS | (50) | 0.41 | Block ACRS | (3) |
| rhpentium | 0.90 | OSKI | 0.75 | OSKI | (100) | 1.03 | Block ZZ-CCS | (5) |
| lhr34 | 2.63 | OSKI | 2.41 | OSKI | (10) | 3.13 | Block ACRS | (3) |
| nug30 | 10.6 | OSKI | 9.3 | OSKI | (10) | *8.35* | BICRS ACRS | (6) |
| s3dkt3m2 | *10.1* | OSKI | 10.3 | OSKI | (2) | 13.7 | Block ACCS | (3) |
| tbdlinux | 8.08 | OSKI | *7.65* | OSKI | (100) | 9.16 | Block SepLast | (5) |
| stanford | 27.2 | OSKI | *11.5* | CRS | (8) | 12.3 | Block ACCS | (8) |
| stanford_berkeley | 31.7 | OSKI | *30.6* | OSKI | (4) | 34.6 | Block SepLast | (5) |
| cage14 | *112* | OSKI | 130 | OSKI | (2) | 141 | BICRS SepLast | (10) |
| wikipedia-2005 | 347 | OSKI | 223 | OSKI | (7) | 213 | BICRS ACRS | (9) |
| GL7d18 | 780 | CRS | 553 | OSKI | (8) | 613 | BICRS SepLast | (10) |
| wikipedia-2006 | 745 | OSKI | 495 | OSKI | (9) | 541 | Block CRS | (9) |

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

Table 2.3: As Table 2.2, but with timings performed on an Intel Core 2 Q6600 architecture. The table includes timings using OSKI, combined with 1D reordering, as in Chapter 1. In this table, the 1D & Blocking scheme includes block orderings and data structures other than Block CRS.

| Matrix | Natural | | 1D | | | 1D & Blocking | | |
|---|---|---|---|---|---|---|---|---|
| fidap037 | 0.13 | CRS | 0.13 | CRS | (2) | 0.17 | Block CRS | (2) |
| memplus | 0.32 | ICRS | *0.30* | CRS | (2) | 0.35 | Block SepLast | (2) |
| rhpentium | 0.98 | ICRS | 0.75 | ICRS | (50) | 1.02 | BICRS ZZ-CCS | (5) |
| lhr34 | 1.71 | ICRS | 1.71 | ICRS | (10) | 1.76 | Block ACCS | (3) |
| nug30 | 6.37 | CRS | 5.80 | CRS | (2) | 6.48 | Block ZZ-CCS | (6) |
| s3dkt3m2 | 8.64 | ICRS | 8.62 | ICRS | (2) | 8.66 | Block ZZ-CCS | (2) |
| tbdlinux | 7.17 | ICRS | 6.53 | ICRS | (4) | *6.15* | Block ACRS | (2) |
| stanford | 22.0 | CRS | 10.6 | ICRS | (9) | 10.3 | Block ZZ-CCS | (9) |
| stanford_berkeley | 24.6 | ICRS | 24.1 | ICRS | (3) | 23.8 | Block CRS | (7) |
| cage14 | 111 | ICRS | 116 | ICRS | (8) | 113 | Block ACCS | (2) |
| wikipedia-2005 | 264 | CRS | 179 | ICRS | (7) | 161 | Block ACRS | (9) |
| GL7d18 | 730 | CRS | 452 | ICRS | (8) | *412* | BICRS ZZ-CRS | (9) |
| wikipedia-2006 | 819 | CRS | 399 | ICRS | (9) | 383 | BICRS ACRS | (10) |

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

Table 2.4: As Table 2.3, but with timings performed on an AMD Phenom II 945e architecture.

noteworthy is that ICRS does not always outperform CRS: ICRS is better on either smaller matrices or when many empty rows (or columns) were encountered, such as in separator blocks. On larger matrices, CRS consistently outperforms ICRS, even after repeated partitioning. Within the simple block data structure, the incremental variants (ICRS/ICCS) do remain the only option, as the alternative non-incremental structure exhibits extremely poor scalability in $p$. Nevertheless, using 2D reordering based on the Mondriaan scheme with CRS-based data structures, or with the BICRS data structure on some architectures, consistently yields large speedups of around a factor 2, or higher.

## 2.5 Conclusions and future work

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

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

The 2D method presented here still uses only row and column permutations, permuting an input matrix $A$ to $PAQ$, with which the multiplications are carried out. This is unchanged from the original method. It is also still possible to combine this method with auto-tuning (cache-aware) software such as OSKI to increase the SpMV multiplication speed further; this software can be applied to optimise the separate

block data structures, but this has not been investigated here. This method can also be implemented using other partitioners than Mondriaan, such as, e.g., PaToH [ÇA99] or Zoltan [DBH+06], although modifications may be required to extract an SBD permutation and separator tree, or to perform partitioning according to the Mondriaan scheme.

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

Various items for future research can be readily identified:

- The simple block-based data structure presented here is very basic, and a more advanced data structure might be introduced to lower or eliminate overhead with increasing $p$, and thus gain even more efficiency; existing methods like the one proposed by Buluç et al. [BFF+09], could be extended and integrated to this end.

- The results indicate that the efficiency of specific block orders is dependent on matrix input. As the analysis already revealed differences between locality (and thus sizes) of blocks in the row and column direction, this can perhaps lead to an efficient heuristic when to choose which block order. This block order does not have to be constant in the recursion, and may require information on the density of separator blocks.

- The difference in performance between standard CRS and ICRS seems dependent on, amongst others, the matrix structure and size; it warrants future research to find the precise dependencies. Again this can lead to an efficient heuristic for adaptively choosing a data structure, also within a simple or advanced block-based data structure.

- The speed of building up the datasets, although already greatly improved when compared to those reported in Chapter 1, can still be increased. In particular, the preparation times with the Mondriaan scheme can still be larger than expected.

# Acknowledgements

The SARA supercomputer centre in Amsterdam and the Netherlands National Computer Facilities foundation (NCF) graciously donated computation time on the Huygens computer and made some of the experimental results possible. The anonymous referees of the paper [YB11c] corresponding to this chapter are also gratefully acknowledged.

## *Symmetric fine-grain and matrix reordering*

Some applications require that for a symmetric input matrix $A$, the resulting DSBD reordered matrix $PAQ$ is symmetric as well. This is for example useful in the case of repeated multiplication: the power method $y = A^s x$, as used in Google PageRank [LM06], can, after symmetric reordering, be rewritten to $Py = (PAP^T)^s(Px) = PA^sP^T(Px)$. As such, enhanced data-locality can be exploited by existing power method kernels.

A second example is domain decomposition; partitioning a problem into smaller subproblems can be done using matrix reordering techniques. Direct techniques which might not work on the complete problem due to, e.g., memory constraints, may work perfectly when applied to the smaller subproblems instead. If the direct techniques are based on symmetry of the specific problem it is applied to, then certainly the subproblems should remain symmetric as well.

Symmetric permutations can be achieved by using the symmetric fine-grain representation as presented in the Introduction, Section 2. There, the $k$th matrix row and $k$th matrix column are together modelled by a hyperedge $n_k = \{a_{ij} \in \mathcal{V} \text{ with } i = k \text{ or } j = k\}$. Since there is only one net category for both rows and columns, any split in $\mathcal{N}_{\{+,c,-\}}$ during the bipartitioning process immediately defines a symmetric permutation.

Symmetric reordering based on this strategy is, in general, of lesser quality. In a regular fine-grain scheme, it is possible that for a given partitioning of $\mathcal{V}$ and a given $i$, $n_i^{\text{row}}$ is cut, while $n_i^{\text{col}}$ is not. In the symmetric fine-grain scheme this distinction is not possible, and both the $i$th row and $i$th column would be cut in favour of symmetry.

The Mondriaan software package implements symmetric reordering based on the fine-grain scheme, and also supports symmetric variants of the other 1D and 2D schemes.

# Chapter 3

# Hilbert-ordered sparse matrix storage

This chapter details a method for sparse matrix storage based on the Hilbert curve, so that an SpMV multiplication algorithm accesses both the input and output vector in a data-local manner. This method can be applied in conjunction with the reordering methods described in Chapter 1 and 2. Still, when applied without reordering, for many matrices a large speedup is attained as shown in Section 3.2, while preprocessing times for this scheme are much faster than those for the reordering methods. This chapter is based on the paper [YB11a]:

- A. N. Yzelman and Rob H. Bisseling, *A cache-oblivious sparse matrix–vector multiplication scheme based on the Hilbert curve*, Progress in Industrial Mathematics at ECMI, Springer, Berlin, 2011, to appear.

## 3.1 An adapted nonzero ordering

A standard way of storing a sparse matrix $A$ is the Compressed Row Storage (CRS) format [BDD$^+$00]. As discussed earlier in Section I.1.5, this scheme stores data in a row-by-row fashion using three arrays:

1. $col\_ind$ of size nz$(A)$ storing the column index of each nonzero in $A$,
2. $nzs$ of size nz$(A)$ storing the numerical value of each nonzero in $A$,
3. $row\_start$ of size $m+1$ such that the nonzeroes at positions from $row\_start_i$ up to $row\_start_{i+1}-1$ of the arrays $col\_ind$ and $nzs$, correspond to the nonzeroes in the $i$th row of $A$.

A standard SpMV multiply algorithm using CRS is given in Algorithm 1. It writes to $y$ sequentially, and thus performs optimally (regarding $y$) in terms of cache

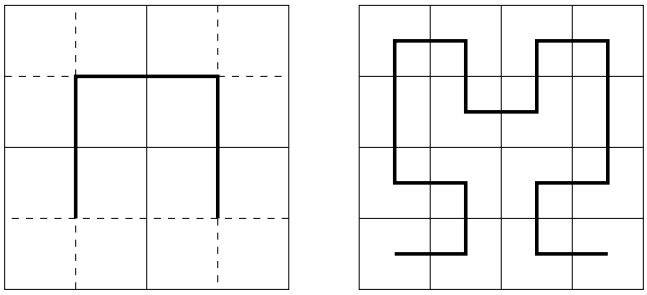


Figure 3.1: The Hilbert curve drawn within two-by-two and four-by-four matrices

efficiency; this is in contrast to accesses to $x$ which, in case of unstructured $A$, are unpredictable.

A way to increase performance is to force the SpMV multiply to work only on smaller and uninterrupted subranges of $x$, such that the vector components involved fit into cache; this can be done by permuting rows and columns from $A$, so that the resulting structure forces this behaviour when using standard CRS; this has been discussed in the Chapters 1 and 2. What is considered in this chapter is a change of *data structure*, instead of changing the input matrix structure. This means finding a data structure which accesses nonzeroes in $A$ in a more 'local' manner, that is, an order such that jumps in the input and output vector remain small and thus yield fewer cache misses. Earlier work in this direction includes the Blocked CRS format [PH99], the auto-tuning sparse BLAS library OSKI [VDY05] and its predecessor Sparsity [Im00], exploiting variable sized blocking [NVDY07, VM05], and several other approaches [Tol97, BBF$^+$07]. Of specific interest, but for the case of dense matrices, is the use of space-filling curves to improve cache locality; in particular the use of the Morton (Z-curve) ordering [Mor66, BFF$^+$09], more recently combined with regular row-major formats to form hybrid-Morton formats [LW07].

The foundation of this chapter was presented by Haase et al. in 2007 [HLP07]. They propose to store the matrix in an order defined by the Hilbert curve, making use of the good locality-preserving attributes of this space-filling curve. Figure 3.1 shows an example of a Hilbert curve within $2 \times 2$ and $4 \times 4$ matrices. This locality means that, from the cache perspective, accesses to the input and output vector remain close to each other when following the Hilbert curve. The curve is defined recursively as can be seen in the figure: any one of the four 'super'-squares in the two-by-two matrix can readily be subdivided into four subsquares, onto which a rotated version of the original curve is projected such that the starting point is on a subsquare adjacent to where the original curve entered the super-square, and similarly for the end point. A Hilbert curve thus can be projected on any $2^{\lceil \log_2 m \rceil} \times 2^{\lceil \log_2 n \rceil}$ matrix, which in turn can embed the sparse matrix $A$, imposing a 1D ordering on its nonzeroes. Haase et al. [HLP07] stored these nonzeroes in *triplet format*, see Section I.1.5 and A.1.1, with the nonzeroes stored in the order as determined by the Hilbert curve. The main

drawback is the difference in storage space required; this is $3{\cdot}\text{nz}(A)$, an increase of $\text{nz}(A) - m$ compared to the regular CRS data structure. The number of cache misses prevented thus must overtake this amount of extra data movement before any gain in efficiency becomes noticeable.

Instead of the triplet data structure, the BICRS structure, introduced in Section 2.1.2, can be applied as well; in fact, BICRS was first introduced in the paper [YB11a] on which this chapter is based. Using this data structure is what enables competitive and practical use of the Hilbert curve in SpMV multiplication.

Since the Hilbert curve generally does not traverse nonzeroes in a row-by-row manner, the BICRS row increments array becomes larger than the number of nonempty rows, and BICRS still is less efficient than plain CRS in terms of memory usage. Experiments will indicate whether the reduction in cache misses justifies the extra overhead. As discussed in the previous chapter, the worst case memory usage of BICRS is bounded by $\mathcal{O}(3\text{nz})$, which occurs precisely when all subsequent nonzeroes reside on different rows. This should occur rarely, however: in case of a dense matrix, the number of row jumps made when nonzeroes are ordered according to the Hilbert curve is about $\text{nz}(A)/2$, although this gives no guarantee for the number of jumps in the sparse case. There, the number of jumps is entirely dependent on the nonzero structure and a worst-case input can artificially be constructed. In general, however, this scheme should outperform a Hilbert ordering backed by the triplet scheme. Note that while the BICRS data structure is bi-directional, the data structure orientation still matters. For example, when the number of jumps is lower in the column direction in CCS order, BICCS is preferred over BICRS , and vice versa.

## 3.2 Experiments and conclusions

Experiments have been performed on two quad-core architectures: the Intel Core 2 Q6600 and the AMD Phenom II 945e. Only one of the four cores was used for the sequential SpMV multiplications. See Table 1 and the various figures in Section I.1.3 for details on these architectures. The SpMV kernels[1], based on CRS, ICRS and BICRS using Hilbert ordering, have been compiled using the gcc compiler[2]. These kernels are each executed $100$ times on given matrices, and report an average running time of a single SpMV multiplication. Experiments have been performed on $9$ sparse matrices, all taken to be large in the sense that the input and output vector do not fit into the L2 cache; see Table 3.1. All matrices are available through the University of Florida sparse matrix collection [Dav11]. Tests on smaller matrices were performed as well, but, in contrast to when using the reordering methods, any decrease in L1-cache misses did not result in a faster SpMV execution. Results on larger matrices in terms of wall-clock time are reported in Table 3.2 for the Q6600 system, and in Table 3.3 for the AMD 945e system. Also reported is the extra build time, that is, the

[1] The source code is freely available at `http://albert-jan.yzelman.net/software`

[2] with the following compiler flags: -O3 -m64 -fprefetch-loop-arrays -funroll-loops -ffast-math -march=native -mtune=native -DNDEBUG

| Name | Rows | Columns | Nonzeroes | | Origin |
|---|---|---|---|---|---|
| stanford | 281903 | 281903 | 2312497 | U | link matrix |
| cont1_l | 1918399 | 1921596 | 7031999 | S | optimisation |
| stanford-berkeley | 683446 | 683446 | 7583376 | U | link matrix |
| Freescale1 | 3428755 | 3428755 | 17052626 | S | circuit design |
| wikipedia-2005 | 1634989 | 1634989 | 19753078 | U | link matrix |
| cage14 | 1505785 | 1505785 | 27130349 | S | DNA |
| GL7d18 | 1955309 | 1548650 | 35590540 | U | combinatorial |
| wikipedia-2006 | 2983494 | 2983494 | 37269096 | U | link matrix |
| wikipedia-2007 | 3566907 | 3566907 | 45030389 | U | link matrix |

Table 3.1: Matrices used in the experiments. An S (U) indicates that the matrix is considered structured (unstructured). A matrix is said to be structured when it is expected, by inspecting its nonzero structure, to already perform well in terms of cache efficiency when using standard CRS.

| Matrix | | CRS | ICRS | Hilbert & BICRS | Build time |
|---|---|---|---|---|---|
| stanford | U | 30.22 | 40.24 | 25.74 | 1456 |
| cont1_l | S | 44.02 | 46.41 | 62.85 | 5085 |
| stanford-berkeley | U | 35.29 | 34.56 | 45.82 | 5578 |
| Freescale1 | S | 122.27 | 131.52 | 210.10 | 14458 |
| wikipedia-2005 | U | 366.45 | 374.82 | 253.45 | 12632 |
| cage14 | S | 136.19 | 141.07 | 165.21 | 20453 |
| GL7d18 | U | 774.55 | 856.16 | 372.25 | 22126 |
| wikipedia-2006 | U | 812.42 | 831.17 | 576.67 | 23839 |
| wikipedia-2007 | U | 1012.73 | 994.35 | 776.48 | 27345 |

Table 3.2: Experimental results on an Intel Q6600, using the matrices in Table 3.1. All timings are in milliseconds.

time required to build the Hilbert BICRS structure minus the time required to build a CRS data structure.

The cache-oblivious SpMV multiplication scheme based on the Hilbert curve works very well on large unstructured matrices. In the best case (the GL7d18 matrix on the Intel Q6600), the Hilbert scheme gains 51 percent in execution speed, compared to plain CRS. On both architectures, 5 out of the 6 unstructured matrices show significant gains, typically around 30 to 40 percent. The only exception is the stanford-berkeley matrix, taking a performance hit of 32 percent, on both architectures. Interestingly, the 1D and 2D reordering methods also do not perform well on this matrix; see Section 2.4. The method also shows excellent performance regarding pre-processing times, taking a maximum of 28 seconds for wikipedia-2007 on the Q6600 system. This is in contrast to 1D and 2D reordering methods, where pre-

| Matrix | | CRS | ICRS | Hilbert & BICRS | Build time |
|---|---|---|---|---|---|
| stanford | U | 22.15 | 27.52 | 18.48 | 832 |
| cont1_l | S | 31.07 | 26.99 | 48.05 | 3084 |
| stanford-berkeley | U | 26.05 | 24.52 | 34.29 | 3415 |
| Freescale1 | S | 98.55 | 95.00 | 148.04 | 8913 |
| wikipedia-2005 | U | 368.36 | 387.39 | 250.3 | 5850 |
| cage14 | S | 116.44 | 110.69 | 140.20 | 12095 |
| GL7d18 | U | 716.32 | 824.32 | 452.89 | 10064 |
| wikipedia-2006 | U | 823.53 | 879.53 | 550.00 | 11814 |
| wikipedia-2007 | U | 1033.95 | 1124.02 | 591.08 | 14753 |

Table 3.3: Experimental results on an AMD 945e, using the matrices in Table 3.1. All timings are in milliseconds.

processing times can take hours for larger matrices, e.g., 21 hours for wikipedia-2006 [YB11c]. Gains in efficiency when reordering, however, are more pronounced than for the Hilbert-curve scheme presented here. Note that the methods do not exclude each other: 1D or 2D reordering techniques can be applied before loading the matrix into BICRS using the Hilbert ordering to gain additional efficiency. The results also show that, as expected, the method cannot outperform standard CRS ordering when the matrix already is favourably structured, where it results in slowdowns.

For future improvement of the Hilbert-curve method, applying the Hilbert ordering to small (e.g., $8$ by $8$) sparse submatrices of $A$ instead of its individual nonzeroes, while imposing a regular CRS ordering on the nonzeroes contained within each such submatrix, promises further speedups. Such a hybrid scheme has also been suggested for dense matrices [LW07], although the motivation differs; in our case, instead of achieving better register reuse, this blocking can reduce the number of row jumps required by BICRS, assuming the rows of the submatrices contain several nonzeroes.

# Part II

# Parallel methods

# Chapter 4

# Bulk Synchronous Parallel SpMV multiplication

Since multicore processors are now in widespread use, writing parallel programs is becoming highly relevant to the world of sequential programming. For the parallel world, interest in shared-memory architectures, as opposed to classical distributed-memory architectures, is rekindled. A parallel programming model which is simple in design, yet able to cater to both memory models, can be of key importance.

This chapter shows that the Bulk Synchronous Parallel (BSP) model, originally designed for distributed-memory systems, can also be applied to shared-memory systems. Additionally, the proof-of-concept MulticoreBSP library is introduced and applied to the sparse matrix–vector multiplication. The chapter ends with an algorithm for SpMV multiplication on shared-memory architectures, which remains quite close to its distributed-memory version. This method does not yield optimal speedups, but provides a good basis nevertheless. Chapter 5 extends the method with insights from both Chapter 1 and 2 to attain a better performing algorithm.

This chapter is based completely on the work [YB11b]:

- A. N. Yzelman and Rob H. Bisseling, *An object-oriented BSP library for multicore programming*, Concurrency and Computation: Practice and Experience (2011), to appear.

## 4.1 Shared-memory Bulk Synchronous Parallel

For shared-memory multicore systems, sharing data is not free. Usually main memory is accessible only via a cache hierarchy, which may or may not be shared amongst cores; threads may thus disrupt each other and slow down computation. An extension to the BSP model to take caches explicitly into account has been proposed by

Valiant [Val08]. The BSP implementation described here, however, still follows the original BSP model [Val90]. This original model, as well as BSP communication libraries such as BSPlib [HMS+98] and PUB [BJvOR03], were introduced to enable portable parallel programming, while attaining predictable performance. As BSP was meant as a single program, multiple data (SPMD) model, a natural question is whether it also performs well in a shared-memory setting. To this end, the *MulticoreBSP* proof-of-concept library has been implemented. This communications library is an object-oriented adaption of BSPlib, written in Java, and targeting only shared-memory systems.

The performance of this library is examined by re-implementing the educational BSPedupack [Bis04] software package. This package contains parallel programs originally written in BSPlib for the following applications:

- dense vector inner-product calculation,
- dense LU decomposition,
- the fast Fourier transform, and
- sparse matrix–vector (SpMV) multiplication.

The performance of the MulticoreBSP library is examined using these applications. As a different system architecture is targeted, however, algorithms can differ to some extent from those originally presented by Bisseling [Bis04]. While experimental results are given for all these applications, the model and library are introduced by describing the inner-product calculation and SpMV multiplication only. The library and all applications are freely available[1].

The choice of Java as programming language underscores the portability goal of BSP. As an interpreted language, the library and applications built on it can be distributed to every machine for which a Java interpreter is available, without the need for recompiling. Furthermore, using Java means that the BSP model is implemented in an object-oriented way, making the model easier to use while also reducing the number of BSP primitives from the already low number of 20 primitives. This is important since it enables easy learning, and helps with keeping the (idealised) parallel machine transparent.

Restricting the number of primitives also restricts the programmer, but this is preferable to simplicity without restriction. With OpenMP[2] [DM98, CJP07], as an example, parallelism can be introduced in code by adding a parallel directive just before a for-loop. This is simple and nonrestrictive[3], but relies on programmer expertise to decide on the scope of variables; care must be taken to avoid race conditions and to actively avoid the unintended sharing of variables. In contrast, the MulticoreBSP

[1] See: `http://www.multicorebsp.com`

[2] See also: `http://www.openmp.org`

[3] The use of directives is furthermore also very portable; depending on the application, OpenMP has the distinct advantage of easily transforming existing sequential code into parallel code.

library adds robustness to algorithms by implicitly assuming all variables are local unless explicitly defined otherwise, while retaining programming simplicity. Also, by structuring computations in independent, sequential phases, separated by synchronisation barriers, another common trap in parallel programming is avoided: deadlocks. Usually, these errors are subtle and may occur only rarely, sometimes only once in hundreds of runs, making them hard to detect. In BSP, such deadlocks can only occur as a sequential error, not by misuse of communication primitives. In summary, the BSP model has the following desirable properties. It

- is easy to learn by programmers and very transparent to them,
- models both distributed-memory and shared-memory architectures,
- predicts performance, and
- is robust with respect to common parallel programming pitfalls.

This chapter has two aims: first, introduce the MulticoreBSP communications library, and second, demonstrate how to design shared-memory algorithms in BSP. In particular, it is shown that it is possible to exploit differences between distributed-memory and shared-memory architectures on the level of implementation. This is done by implementing a shared-memory version of the sparse matrix–vector (SpMV) multiplication, which remains close to the original distributed-memory version from BSPedupack. Similar efforts have been made for dense LU decomposition and the Fast Fourier transform.

Within the algorithm listings, all classes, member functions, and variables defined in the library are printed in *italic*, while class and function names not defined therein are printed in `typewriter` font. Function code in listings is printed in plain roman, with the only further exception of reserved words, such as **for** and **return**, which are printed **boldface**.

The remainder of the chapter is structured as follows. The BSP model is introduced in Section 4.2, and the MulticoreBSP library in Section 4.3. Using this proof-of-concept Java library, Section 4.4 lays the foundation for a parallel SpMV multiplication by first parallelising the dense vector inner-product calculation. Section 4.5 then details the final BSP SpMV algorithm for multicore systems, which is put to the test in Section 4.6. This section also reports on performance of the dense LU and FFT algorithms. Conclusions and future work follow in Section 4.7.

## 4.2 BSP model

The original BSP model [Val90] targets distributed-memory systems and models them using four parameters: $p, g, l,$ and $r$. The total number of parallel computing units (cores) of a BSP machine is given by $p$, while $r$ measures the speed of each such core, in flops per second. Each core executes the same given program, usually working on different data; that is, BSP is a single program, multiple data (SPMD)

model. The parallel program is broken down in *supersteps*, which are separated by *synchronisation barriers*. During a superstep, a process can execute any instruction, but it cannot communicate with other processes. It can, however, queue communication requests. When a process encounters a synchronisation barrier, execution halts until all processes encounter this barrier, upon which the next superstep is started concurrently. At synchronisation, all queued communication requests from the previous superstep are processed; the only communication thus occurs as part of synchronisation. The communication costs occurring this way are modelled in BSP by $l$, which models the time required to get past a synchronisation barrier, and $g$, which models the *communication gap*, the time gap between two data words sent by a process during communication; $g$ corresponds to the inverse of the bandwidth, the speed at which data is communicated.

Suppose there are $t$ supersteps, where each superstep $i$ contains $w^i(s)$ flops of work for process $s$, and where each such process receives $V_R^i(s)$ or sends $V_S^i(s)$ bytes of communication. The maximum number of bytes $V^i(s)$ communicated then is $V^i(s) = \max\{V_R^i(s), V_S^i(s)\}$, and the total running time (in seconds) on a BSP machine is:

$$T = \sum_{i=0}^{t-1} \left( \frac{1}{r} \cdot \max_s w^i(s) + g \cdot \max_s V^i(s) + l \right). \tag{4.1}$$

This is the *BSP cost model*, as applicable to distributed-memory parallel computing, and enables performance prediction of BSP algorithms.

## 4.3 Object-oriented bulk synchronous parallel library

When using object-oriented parallel programming on distributed-memory systems and communication occurs, entire *objects* need to be transferred. This is done by *marshalling* these objects in transferable code, a complex and time-consuming process: other objects that the object to transfer may depend on, have to be identified and recursively marshalled as well. While this is less an issue on shared-memory systems, as in principle all objects are stored in shared-memory, MulticoreBSP demands that each communicable object can be *cloned* in memory; it must be possible for the library to copy an object and all its dependencies so that the resulting clone is completely independent of the original variable and its dependencies. Note that cloning differs from marshalling in that no transferable code is required.

Object-oriented bulk synchronous parallel models have been researched before, for example by Lecomber [Lec94], targeting distributed-memory systems. Another parallel model is the Coarse Grained Model (CGM), by Dehne et al. [DFRC96]; its library CGMlib [CD05], by Chan and Dehne, also is object-oriented and targets distributed-memory systems by using an underlying communications library, such as MPI. Yelick et al. [YSP+98] introduced a Java dialect specifically for high performance computing and parallelism, using global memory space extended with ad-

vanced memory management based on programmer input. An object-oriented parallel model supporting both shared and distributed memory has been presented by Kaminsky [Kam07]. Similarly, there also exists a BSP library for the Python programming language [Hin03], usable on shared-memory multicore systems as well as on distributed-memory systems (by using a standard BSPlib implementation). A BSP model developed specifically for shared-memory systems has previously been researched by Tiskin [Tis98], who introduced the Bulk Synchronous Parallel random access machine (BSPRAM). This model assumes that each parallel process can access two types of memory: local memory, on which it can execute local computations, and a shared memory accessible by all processes, facilitating inter-process communication. This model is more akin to OpenMP than to the pure BSP model used in this chapter.

Before introducing the object-oriented BSP library for multicore systems, first some basic terminology from object-oriented languages is recalled. Programming code is grouped into classes, each class having its own (local) member variables as well as member functions. Multiple *instances* of a class can be created; much like there can be multiple data elements of the same primitive type (e.g., integer). An instance is created by calling the *constructor* function of its class. Such a constructor may require specific parameters, so that an instance cannot be created without supplying those parameters. A class may be extended from a given superclass, meaning that it inherits all the members, including their visibility (see below), from its superclass. Such a subclass normally also defines new member variables or functions. Member variables or functions have one of the following *visibilities*:

- *private*, visible only to instances of the same class, but excluding subclasses;
- *protected*, visible only to instances of the same class, including subclasses;
- *public*, visible to all instances regardless of class.

Member functions may be defined, yet not implemented; in such a case the function is called *purely virtual*, and its corresponding class *abstract*. A normal virtual member function is already implemented, but can be redefined in subclasses.

A generic BSP program is defined to be a class, *BSP_PROGRAM*, having at least the functions in Table 4.1 defined. Any specific parallel algorithm is extended from this superclass, and must implement the following two protected and purely virtual functions:

- *main_part()*: this code is only executed by a single process, and from the protected functions in Table 4.1 it can only call *bsp_begin(int)*,
- *parallel_part()*: this is the code run in parallel; once this function is reached, all BSP functions can be called, with the exception of *bsp_begin()*.

The sequential phase can be used to prepare data or determine algorithm parameters needed for the parallel phase. From within the sequential code, a call to *bsp_begin(int)*

| **purely virtual:** | |
|---|---|
| *main_part()* | will contain sequential code, |
| *parallel_part()* | will contain parallel code. |
| **protected:** | |
| *bsp_begin(int)* | starts the parallel execution, |
| *bsp_nprocs()* | returns the number of threads executing this algorithm, |
| *bsp_pid()* | returns the current, unique thread identification number, |
| *bsp_sync()* | synchronises all threads, |
| *bsp_abort()* | aborts the parallel execution. |
| **public:** | |
| *start()* | starts the program. |

Table 4.1: *BSP_PROGRAM* function list

starts the parallel phase using the given number of processes. Each such process executes the exact same code, defined in the *parallel_part()* function. The number of concurrent processes can be queried by calling *bsp_nprocs()*, and a unique process ID can be retrieved by using *bsp_pid()*. Parallel execution can be terminated prematurely by calling *bsp_abort()*. The function *bsp_sync()* is a barrier that synchronises all processes; a process will halt execution when encountering this function, only to continue when all processes were halted.

Defining only this BSP program class, a simple parallel application can already be written; see the Hello World example, Algorithm 4. Since a program is in essence a class, to run it, an instance of it has to be created first: "Hello_World myInstance = new Hello_World();". Creating an instance of a parallel job does not implicitly start it. To do that, the *start()* function must be used, so the job is executed by "myInstance.*start()*;". A short-hand way of doing this is shown in Algorithm 4. After starting, the program will print the following four lines, *in an undefined order*:

```
Thread 0 says: Hello World!
Thread 1 says: Hello World!
Thread 2 says: Hello World!
Thread 3 says: Hello World!
```

Every thread has its own instance of its parallel class, such that no two threads share the same variables. For most parallel jobs this is insufficient since data communication should be allowed to occur between threads, implying that inter-thread data transfer between variables should be possible. In this BSP framework, such variables are called *shared*, but a thread can, at all times, only see its own local value of the variable. Actual communication is only possible through explicit BSP functions, which will be defined shortly. It is essential to note that as such, these BSP-style shared variables are not at all shared in the classical sense: no two threads can read or

**Algorithm 4** Hello World example using the MulticoreBSP library

```
Class Hello_World extends BSP_PROGRAM {
protected function main_part():
 1: bsp_begin(4);

protected function parallel_part():

 1: print “Thread ”+bsp_pid()+“ says: Hello World!”;

}
```

Start using:

```
 1: (new Hello_World()).start();
```

write to the same memory location at the same time, and a BSP-style shared variable maintains a local instance for each active thread.

To facilitate this, an abstract BSP class *BSP_COMM* is defined. Any shareable object type will have to implement (or extend) this class; in other words, all shared variables have *BSP_COMM* as superclass. No other objects can be communicated. Since a shared variable has no meaning if not connected to a parallel program, the constructor of every shared variable must take a *BSP_PROGRAM* as parameter, thus linking the shared variable and the parallel program it is used in. Communication always entails that some source data is transferred from a source process to a destination variable at a destination process. The MulticoreBSP library facilitates three communication methods to do this: one being to *put* data in the memory local to another thread, another being a method to *get* data from variables local to another thread, and, finally, a method to *send* messages to a variable local to another thread. In all cases, the destination variable must be shared, since by definition any non-shared variable is not visible to other threads. These *bsp_put(...)*, *bsp_get(...)*, and *bsp_send(...)* functions are therefore defined as public functions within *BSP_COMM*; that is, as functions of destination variables. Thus if “destination" is a shared variable, or rather, an instance of a class derived from *BSP_COMM*, then the following communication functions are defined:

- destination.*bsp_put*(source, destination_thread),
- destination.*bsp_get*(source, source_thread),
- destination.*bsp_send*(source, destination_thread).

In all cases, the source parameter is optional, by default referring to the destination variable at the current process. The source and destination threads are designated by referring to their thread IDs: a thread can use the get-directive to extract data from memory local to another thread, while the put-directive allows a thread to push its data into memory local to another thread. All communication happens at synchronisation

time; multiple puts or gets to the same location at the same process result in only one of them coming through at the destination. This is in contrast to the message-passing *bsp_send*, where messages are queued at the destination variable. When a shared variable at some processor receives a message (after synchronisation), reading out the corresponding queue is facilitated by two functions:

- *bsp_qsize*() which returns the number of messages still in the queue, and
- *bsp_move*(), which moves an item from the local queue into its local variable.

If a queue is not empty when a synchronisation barrier is encountered, all remaining entries are evicted and hence are not available in the next superstep. A final function, *bsp_unregister*(), is used to invalidate a shared variable, freeing up all memory it uses at all threads; this should never be called inside a superstep where the variable is still used.

The functions introduced so far have been available for distributed-memory systems as well, for example in BSPlib [HMS+98]. MulticoreBSP additionally introduces a new communication directive, *bsp_direct_get(...)*, so that it can take full advantage of the shared-memory architecture it is designed to work with. The syntax is the same as that for *bsp_get*, but semantics differ in that the communication is not queued: instead, it is executed immediately, within a superstep. This is a major change from the BSP paradigm, but an acceptable one due to the similarity to the plain get-directive and a way to exploit this new directive systematically, as will be described shortly. Also, BSPlib [HMS+98] in fact contained similar primitives; the so-called high-performance versions of the get- and put-primitives, which could initiate communication inside supersteps as well. These function calls, did not wait for the communication to finish, however. In other words, all BSP communication primitives cache communication (and then immediately continue, i.e., all primitives are *non-blocking*), with the single exception of *bsp_direct_get(...)* in MulticoreBSP.

A standard distributed-memory BSP algorithm can systematically be adapted to exploit shared-memory systems using this new primitive. When a superstep only communicates via get-primitives while the data already is available at the remote process, these get-primitives can be substituted for direct-gets and the next synchronisation barrier can be removed; the direct-get functionality thus can reduce the number of supersteps of a classical BSP algorithm on shared-memory architectures. Summarising, *BSP_COMM* defines the purely virtual functions used for communication shown in Table 4.2.

Note that according to the BSP model, all communication is guaranteed to have been done after a synchronisation barrier is passed. Differences with the original BSPlib are that the bsp_init(), bsp_push_reg(), bsp_pop_reg(), and bsp_end() primitives no longer appear; bsp_end() is in effect assumed after the function *parallel_part* finishes, initialisation is moved to the program constructor, variable registration is moved to the variable constructor, and the bsp_pop_reg() function is replaced with the *bsp_unregister* function.

| **public:** | |
|---|---|
| *bsp_put(source, destination_pid)* | puts source to this variable at destination_pid, |
| *bsp_get(source, source_pid)* | gets source from source_pid and stores it here, |
| *bsp_direct_get(source, source_pid)* | like *bsp_get*, but does not wait for sync, |
| *bsp_send(source, destination_pid)* | sends source to the queue of destination ID, |
| *bsp_qsize()* | gets the number of messages still in queue, |
| *bsp_move()* | moves an item from the queue to this variable, |
| *bsp_unregister()* | frees the local memory used by this variable. |

Table 4.2: BSP_COMM virtual function list

Regarding the BSP cost model, a difference is caused by the direct-get primitive, which is executed during a superstep. In determining the total communication cost, the cost of a direct get is bounded by the cost of a normal get performed in synchronising. This way, when determining the total running time, the maximum communication volume is still used; the original BSP cost model shown in Equation (4.1), is retained.

Before proceeding with describing the implementation of the sparse matrix–vector multiplication using the library introduced here, a simple description of a parallel inner-product calculation algorithm is given so that several communication variables can be introduced. A generic shared variable is implemented as *BSP_REGISTER<T>*, which stores a variable of type *T*, accessible using read and write methods. It extends *BSP_COMM*, and, as discussed earlier, $T$ must support cloning. This class alone, although versatile, is not yet enough to attain efficient code.

## 4.4 Inner-product calculation

The problem is the parallel calculation of the inner product $(x, y) = \sum_{i=0}^{n-1} x_i y_i$ for two given vectors $x, y \in \mathbb{R}^n$. Assuming $p$ threads are available, an intuitive approach then is to divide the vectors $x$ and $y$ into $p$ blocks of (roughly) equal size, where each thread calculates the sum of the corresponding block. This is followed by communicating the partial results and summing them to obtain the final result. Assuming that this inner-product calculation is part of a larger parallel scheme, the vectors $x$ and $y$ already have been distributed in memory, and the only remaining task is to compute partial inner products and communicate the resulting partial sums. Each thread can define a shared array of variables storing double values, to which partial sums can be communicated. Such an array could be defined using a *BSP_REGISTER* and standard Java array constructs, yielding code such as "ArrayList<*BSP_REGISTER*<Double>> sums;". The downside is that each register added to this array has to be individually constructed. Afterwards, putting a value $x$ at index $i$ at process $s$ would be done by "sums.get($i$).*bsp_put*($x$,$s$);". This has several disadvantages:

| **public:** | |
|---|---|
| *bsp_put* | *(source, s_offset, destination_pid, d_offset, length)* |
| *bsp_get* | *(source, source_pid, s_offset, d_offset, length)* |
| *bsp_direct_get* | *(source, source_pid, s_offset, d_offset, length)* |

Table 4.3: BSP_COMM_ARR virtual function list, excluding those already defined by BSP_COMM

- low performance; a single entry must be accessed through multiple classes, and each register added to the array must be constructed separately,
- verbosity; a syntax-heavy approach to control a single element of the array,
- memory overhead; each vector element is registered separately.

Even worse is the syntax and overhead needed to put or get ranges of vectors, as, for example, required by the dense LU algorithm.

To improve this, the *BSP_COMM_ARR* subclass of *BSP_COMM* was introduced. The functions that *BSP_COMM_ARR* defines are in addition to those in Table 4.2, and are shown in Table 4.3. The put, get, and direct-get functions have been modified to copy a number of *length* elements of the array in a single communication request. If this number of elements is smaller than the array length, the use of *offsets* in either the source or destination array is convenient. For example, if $x$ and $y$ are arrays implementing the *BSP_COMM_ARR* class, the code "$y$.*bsp_put*$(x, i, s, j, l)$;" would replace $y$ at process $s$ by

$$(y_0, \ldots, y_{j-1}, x_i, \ldots, x_{i+l-1}, y_{j+l}, \ldots).$$

The (direct) get function works in the expected similar fashion. The main implementation of *BSP_COMM_ARR* is *BSP_ARRAY*$< T >$, which makes available an array with elements of type $T$ at each process. The type $T$ must still support cloning. For efficiency[4], when $T$ should be an `int` or `double` primitive type, the specialised *BSP_INT_ARRAY* and *BSP_DOUBLE_ARRAY* classes should be used. Both of these specialised variants also define the *getData()* method, giving the programmer access to the local underlying raw array. This reference is guaranteed constant with respect to the array object, from construction up until de-registration or end of the parallel phase. Note that any variable implementing *BSP_COMM_ARR*, actually stores $pn$ values, where $p$ is the number of threads used and $n$ the array size; just as a regular shared variable implementing *BSP_COMM* keeps track of $p$ local instances. Algorithm 5 shows a MulticoreBSP inner-product algorithm, using the constructs introduced above.

[4]in Java, "Double" differs from "`double`", the first being an object wrapping a raw double type. Accessing such a class incurs overhead, making *BSP_ARRAY*<Double> an inefficient construct.

**Algorithm 5** Inner-product calculation for identically pre-distributed vectors $x, y$

Calculates the inner product $(x, y) = \sum_i x_i y_i$. Local subvectors $\tilde{x}$ and $\tilde{y}$ are assumed available, and distributed identically.

```
protected function bsp_ip() {
```
1: sums = new *BSP_DOUBLE_ARRAY*(this, *bsp_nprocs*());
2: $\alpha = 0$;
3: **for** $i = 0$ **to** $\tilde{x}$.length$-1$ **do**
4: $\quad \alpha = \alpha + \tilde{x}[i] \cdot \tilde{y}[i]$;
5: **for** $i = 0$ **to** *bsp_nprocs*$() - 1$ **do**
6: $\quad$ sums.*bsp_put*$(\alpha, 0, i,$ *bsp_pid*$(), 1)$;
7: *bsp_sync*();
8: $\alpha = 0$, $a =$ sums.*getData*();
9: **for** $i = 0$ **to** *bsp_nprocs*$() - 1$ **do**
10: $\quad \alpha = \alpha + a[i]$;
11: **return** $\alpha$;
```
}
```

## 4.5 Sparse matrix–vector multiplication

To enable parallel computing of $y = Ax$, the matrix $A$ and both vectors $x$ and $y$ must be distributed. If $p$ processes are used, such distributions are given by the maps

$$\begin{aligned} \pi_A : [0, m-1] \times [0, n-1] &\rightarrow [0, p-1], \\ \pi_x : [0, n-1] &\rightarrow [0, p-1], \\ \pi_y : [0, m-1] &\rightarrow [0, p-1]. \end{aligned} \tag{4.2}$$

Partitioners such as Mondriaan [VB05] or Zoltan [DBH+06] preprocess sparse matrices to find distributions optimised for parallel SpMV multiplication; this chapter will not deal with partitioning methods, and simply assumes the maps from (4.2) are available. These maps, however, while containing in principle all necessary information, still need further preprocessing to make them directly suitable for parallel SpMV multiplication.

### 4.5.1 Mappings for sparse matrix and dense vector distributions

For each thread $s \in [0, p-1]$, a local matrix $A_s$ of size $m_s \times n_s$ is stored; this local matrix can be smaller than $A$ itself since it stores only those entries $a_{ij} \in A$ for which $\pi_A(i, j) = s$, then removes empty rows and columns, and renumbers nonempty rows and columns accordingly. A local input vector $x^s$ is also stored, so that it contains the elements from $x$ with index $j$ for which $\pi_x(j) = s$, and similarly for $y^s$ and $\pi_y$. Then, during local multiplication with elements from $A_s$, information is required at which process and index the corresponding element from the input vector $x$ resides,

and similarly for the output vector $y$. Thus the following maps are preferable to those from Equation (4.2):

- $\pi_r^s : [0, m_s - 1] \to [0, p - 1]$, so that $\pi_r^s(i)$ is the process index for the element of the output vector corresponding to the $i$th row of $A_s$,
- $\pi_c^s : [0, n_s - 1] \to [0, p - 1]$, so that $\pi_c^s(j)$ is the process index for the element of the input vector corresponding to the $j$th column of $A_s$,
- $I_r^s : [0, m_s - 1] \to [0, \max_k m_k - 1]$, so that $I_r^s(i)$ is the index of the local output vector $y^{\pi_r^s(i)}$ corresponding to the $i$th row of $A_s$, and
- $I_c^s : [0, n_s - 1] \to [0, \max_k n_k - 1]$, so that $I_c^s(j)$ is the index of the local input vector $x^{\pi_c^s(j)}$ corresponding to the $j$th column of $A_s$.

The latest version of the Mondriaan package (version 3.11) calculates these mappings automatically.

### 4.5.2 Partially buffered multiplication

Assumed is that the local matrices $A_s$ and vectors $x^s, y^s$ as well as the appropriate mappings from the previous section are available. During local multiplication, non-local elements from the input vector may be required. These values are buffered beforehand, by allocating an input vector buffer of size $n_s$, and copying local and non-local input vector elements into this buffer. This buffering of the input vector is called the *fan-out* step [Bis04, Chapter 4]; the only difference with the algorithm described there is that the shared-memory variant will make use of the direct-get primitive, which prevents the fan-out operation from having to be done in a separate superstep. This brings the parallel SpMV multiplication algorithm down from three supersteps, to only two.

After fan-out, upon entering the SpMV multiplication kernel, each process starts traversing the nonzeroes in its local matrix $A_s$ in a row-major order. For each encountered row, it looks up whether the corresponding output element from $y$ is local or not. If so, products of nonzero values and elements from $x$ are immediately added to the local $y^s$. If not, they are added to a local temporary double value, which, upon switching to the next row, is sent to the correct process using *bsp_send*. After local multiplication, all processes synchronise to prepare for the second and last superstep: the *fan-in*. In this step, the incoming messages are processed. From each message, the target index and the remote contribution is read and added to the correct element of the local output vector.

An implementation detail regards the use of the send primitive in this setting; this directive on a shared array transmits objects of the same type, which are entire arrays. For the SpMV multiplication, however, sending only an index and a double value is required instead. A partial sum for $y$ can then be added at the correct position of the local vector at the target process. To this end, a separate class (Pair) is defined, storing both an integer and a double value, and a shared variable of this class is created;

on this type of variable, message passing with *bsp_send* will perform as required. Algorithm 6 clarifies this by showing the resulting implementation.

If $|\{\pi_r^s = k\}|$ is the number of elements in $\pi_r^s$ equal to $k$, the number of flops performed by process $s$ can be expressed as:

$$\begin{cases} 2 \cdot \text{nz}(A_s) & \text{in superstep 1, and} \\ \sum_{i=0, i\neq s}^{p-1} |\{\pi_r^i = s\}| & \text{in superstep 2.} \end{cases}$$

Here, each nonzero is counted as two flops, and the number of nonzeroes of a matrix $A$ is given by $\text{nz}(A)$. The communication volume sent $V_S(s)$ or received $V_R(s)$ by process $s$ is:

$$\begin{aligned} V_S(s) &= |\{\pi_r^s \neq s\}| + \sum_{i=0, i\neq s}^{p-1} |\{\pi_c^i = s\}|, \\ V_R(s) &= |\{\pi_c^s \neq s\}| + \sum_{i=0, i\neq s}^{p-1} |\{\pi_r^i = s\}|, \text{ and} \\ V(s) &= \max\{V_S(s), V_R(s)\}. \end{aligned} \tag{4.3}$$

The time taken by this algorithm according to the BSP model then is:

$$T_{\text{SpMV}} = \frac{1}{r} \cdot \left( 2 \cdot \max_s \text{nz}(A_s) + \max_s \sum_{i=0, i\neq s}^{p-1} |\{\pi_r^i = s\}| \right) + g \cdot \max_s V(s) + l. \tag{4.4}$$

## 4.6 Experiments

SpMV experiments have been performed using the datasets presented in Table 4.4. These matrices are preprocessed using the Mondriaan software package[5], which partitions them for parallel SpMV and gives local versions of the input matrix, together with the final mappings introduced in Section 4.5.1. The preprocessing time required is reported in Table 4.4 as well. The BSP SpMV driver reads in the Mondriaan output, then performs 100 parallel SpMV multiplications as described in Algorithm 6, and reports the average time taken. This SpMV multiplication software is part of a re-implementation of BSPedupack and is freely available[6].

Experiments are run on three different architectures, namely:

- the AMD Phenom II 945e quad-core processor,

[5] Available freely at: `http://www.math.uu.nl/people/bisselin/Mondriaan`
[6] See: `http://www.multicorebsp.com/BSPedupack`

**Algorithm 6** Parallel sparse matrix–vector multiplication

Calculates $y = Ax$ in parallel. Based on buffering of input vector elements.

**Input**: local $m_s \times n_s$ submatrix $A_s$ of $A$ for process $s$,
local subvector $x^s$ of $x$,
local subvector $y^s$ of $y$,
mappings $\pi_r^s$, $\pi_c^s$, $I_r^s$, $I_c^s$.

```
class Pair() {
    int tag; double value;
    Pair( int _t, double _v ) {
        tag = _t; value = _v;
    }
}

protected function bsp_spmv() {
```

1: $s = bsp\_pid()$;
2: $\tilde{x} =$ new *BSP_DOUBLE_ARRAY*(this, $n_s$);
3: pairs = new *BSP_REGISTER*<Pair>(this);
4: **for** $j = 0$ **to** $n_s - 1$ **do**
5: $\quad \tilde{x}.bsp\_direct\_get(x^s,\ \pi_c^s(j),\ I_c^s(j),\ j,\ 1)$;
6: **for** $i = 0$ **to** $m_s - 1$ **do**
7: $\quad$ **if** $\pi_r^s(i) = s$ **then**
8: $\quad\quad$ **for all** $a_{ij} \neq 0$ in the $i$th row of $A_s$ **do**
9: $\quad\quad\quad y^s_{I_r^s(i)} = y^s_{I_r^s(i)} + a_{ij} \cdot \tilde{x}_j$;
10: $\quad$ **else**
11: $\quad\quad \alpha = 0$;
12: $\quad\quad$ **for all** $a_{ij} \neq 0$ in the $i$th row of $A_s$ **do**
13: $\quad\quad\quad \alpha = \alpha + a_{ij} \cdot \tilde{x}_j$;
14: $\quad\quad$ pairs.*bsp_send*(new Pair($I_r^s(i), \alpha$), $\pi_r^s(i)$);
15: *bsp_sync()*;
16: **while** pairs.*bsp_qsize*() $> 0$ **do**
17: $\quad$ pairs.*bsp_move*();
18: $\quad y^s_{\text{pairs.}read\text{().tag}} = y^s_{\text{pairs.}read\text{().tag}} +$ pairs.*read*().value;

```
}
```

| Name | Rows | Columns | Nonzeroes | $p = 4$ | $p = 64$ |
|---|---|---|---|---|---|
| west0497 | 497 | 497 | 1721 | 1 sec. | 1 sec. |
| fidap037 | 3565 | 3565 | 67591 | 1 sec. | 2 sec. |
| s3rmt3m3 | 5357 | 5357 | 106526 | 1 sec. | 5 sec. |
| memplus | 17758 | 17758 | 126150 | 1 sec. | 8 sec. |
| cavity17 | 4562 | 4562 | 138187 | 1 sec. | 3 sec. |
| bcsstk17 | 10974 | 10974 | 219812 | 3 sec. | 8 sec. |
| lhr34 | 35152 | 35152 | 764014 | 6 sec. | 16 sec. |
| bcsstk32 | 44609 | 44609 | 1029655 | 15 sec. | 31 sec. |
| nug30 | 52260 | 379350 | 1567800 | 43 sec. | 4 min. |
| s3dkt3m2 | 90449 | 90449 | 1921955 | 39 sec. | 1 min. |
| tbdlinux | 112757 | 21067 | 2157675 | 3 min. | 6 min. |
| stanford | 281903 | 281903 | 2312497 | 5 min. | 8 min. |
| stanford-berkeley | 683446 | 683446 | 7583376 | 13 min. | 21 min. |
| cage14 | 1505785 | 1505785 | 27130349 | 16 min. | 45 min. |
| wikipedia-2005 | 1634989 | 1634989 | 19753078 | 5.5 hr. | 9 hr. |
| wikipedia-2006 | 2983494 | 2983494 | 37269096 | 19 hr. | 23 hr. |

Table 4.4: The matrices used in MulticoreBSP SpMV experiments. The matrices are grouped into two sets by relative size, where the first set typically fits into the L2 cache, and the second does not. The last two columns show preprocessing times required for distributing the matrix over $p$ processes, using the Mondriaan software package with the default strategy. This reordering was done on an AMD Opteron 2378.

- the Intel Core 2 Q6600 quad-core processor, and
- the Sun UltraSPARC T2 processor.

The AMD and Intel systems both run a Linux operating system, and use the 1.6.0_23 version of the Sun Java compiler and runtime environment; the Sun platform runs on the Solaris operating system, and uses the 1.5.0_27 version of Sun Java. Details on the AMD and Intel architectures are found in Section I.1.3. What is important to note is that the AMD processor provides uniform access for its four cores, which is in contrast to the Intel design: the Q6600 has no L3 cache, and the two available $4$MB L2 caches available are each shared by two cores, so that any one core has a faster transfer speed with one specific other core, while the two remaining cores must be accessed through the main memory. As such, the Intel processor is said to be a *cache-coherent non-uniform memory access* (cc-NUMA) architecture. The BSP model used here does not take this into account; this would require the Multi-BSP model [Val08], as well as an adaptation of the Mondriaan partitioner. The BSP algorithms presented here are expected to perform better on the AMD architecture, due to the more uniform memory access.

The UltraSPARC T2 architecture, being built specifically for throughput-intensive applications, follows a completely different strategy. While typically operating at lower clock rates, there are $8$ cores available on a single processor with each core supporting fast interleaved execution of $8$ concurrent threads; a single processor thus can execute $64$ threads. The idea is that per core, when one of the threads stalls (e.g., due to a cache miss), one of the other threads can proceed. This hides the latency of data transfer. One UltraSPARC T2 core has two calculation units, meaning two threads can work simultaneously while the other $6$ wait, ideally for data still to arrive.

The UltraSPARC T2 has one $4$MB L2 cache, shared amongst all cores by means of a crossbar. Main memory is accessible through this cache, and is divided over four different memory controllers. The system I/O is connected via the same crossbar. Each single core has its own $8$kB L1 data cache, but note that this cache is shared amongst $8$ threads. It also has more extensive (compared to the Intel and AMD architectures) instruction caches and translation lookaside buffers. Overall, this architecture is expected to offer the $8$ processors a uniform memory access time, and should offer good scalability, especially for smaller problems. Figure 4.1 gives a schematic overview for this processor.

For all BSP algorithms, speedups for various problem instances are reported, using a varying number of threads, as appropriate for the architecture experimented on. Speedups are relative to the parallel program run with $p = 1$, as comparable stand-alone sequential algorithms in Java are not readily available. For the SpMV multiply and the FFT algorithm, however, the main computational kernel is in fact plain (but unoptimised) sequential code. Still, the reported values are optimistic speedups as some parallel overheads remain present for one thread, even if they are constant and do not scale with the problem size.

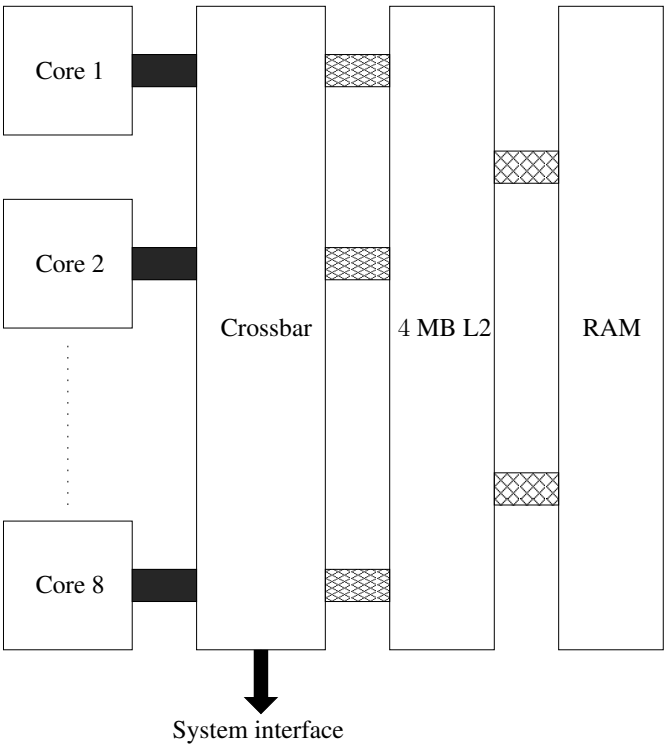



Figure 4.1: Diagram of the Sun UltraSPARC T2 processor. The processor consists of 8 cores (3 of which are displayed in this figure), and each core can execute 8 threads, two of which simultaneously. Each core has a local 8kB L1 data cache (not displayed). A single core and its local L1 cache is connected to a crossbar. Through this crossbar, communication to the other cores, the L2 cache, and the system interface is made possible. The L2 cache has eight banks, and is connected to main memory through four memory controllers (2 of which are displayed here).

### 4.6.1 Benchmarking

The parameters $r$, $g$, and $l$ are estimated via a benchmarking program, described and included in BSPedupack [Bis04]. In short, this benchmark sends messages of varying length to all processors, and measures the time it takes to send $h$ messages and synchronise, a total of $1000$ times. This measurement is conducted for $h = 1, \ldots, 128$. Afterwards, a least-squares fit is calculated to find the affine relation ($gh + l = t/1000$) between the message size $h$ and the time $t$ required to send $1000$ messages. The speed of a single core is measured using dense vector operations. Results are shown in Table 4.5.

The Intel architecture shows a constant computing rate per core (in flop per second), while latency increases as the number of cores $p$ is increased. As expected due to the sharing of the L2 caches, the communication gap and the latencies are much lower for $p = 2$ than for $p > 2$, where the communication gap is constant for $p = 3$ and $p = 4$. The AMD processor shows a more variable flop rate, and even shows a drop in performance for $p = 4$. When using more than one core the latency is constant, but slightly higher than that of the Intel processor. Like the Intel processor, the message gap increases for $p > 2$ but stabilises after $p = 3$, indicating that L3 cache sharing is faster for $p = 2$ than for higher $p$.

The Sun machine behaves as expected for smaller $p$: $r$ is constant, and $g, l$ increase only slightly as more processes are used. This indicates that processes are first allocated to a single core, and spread over cores when more than eight processes are required. The large increase in $g$ and $l$ from $p = 8$ to $p = 16$ supports this, as using more than one core means communication has to go through the crossbar. This effect increases as more and more cores are utilised. An anomaly is the decrease in process speed for $p \geq 32$; there, it is possible the number of threads is too high with respect to the data throughput, such that there is no room for more threads to work while other threads wait for data to arrive. Indeed, while benchmarking, the $4$kB of data used is quite local and does not cause long latency times. If the required data would take longer to arrive, the processor cores would have more time available to interleave other threads, thus retaining the performance speed per thread. This indicates benchmarking with localised data may not yield the correct value for $r$ with $p$ taken large, on this architecture, when predicting running times for applications with irregular data accesses.

### 4.6.2 Prediction

The benchmarks can be used to predict SpMV performance, as after benchmarking a BSP machine and partitioning an input matrix, all values in Equation (4.4) are known, and the theoretical running time of the SpMV algorithm can be obtained. From these values, it is also possible to calculate the theoretical speedups. Using the actual running times reported in the following subsection, both predicted values can be compared to actual measurements. Dividing the measured values by the predicted values then gives the error factors; Tables 4.6 and 4.7 show the result of this experiment.

| | Intel Q6600 | | | AMD 945e | | |
|---|---|---|---|---|---|---|
| $p$ | $r$ (Mflop/s) | $g$ (flops) | $l$ (flops) | $r$ (Mflop/s) | $g$ (flops) | $l$ (flops) |
| 1 | 593.62 | 76.51 | 666 | 819.20 | 84.9 | 379 |
| 2 | 597.99 | 174.52 | 7764 | 890.43 | 357.7 | 31921 |
| 3 | 605.35 | 281.22 | 12342 | 851.23 | 1001.1 | 28748 |
| 4 | 600.17 | 323.79 | 28947 | 663.91 | 935.1 | 32950 |

| | Sun UltraSPARC T2 | | |
|---|---|---|---|
| $p$ | $r$ (Mflop/s) | $g$ (flops) | $l$ (flops) |
| 1 | 47.08 | 50.19 | 228 |
| 2 | 47.98 | 74.78 | 3979 |
| 4 | 45.66 | 97.31 | 10579 |
| 8 | 44.66 | 119.17 | 24232 |
| 16 | 47.45 | 220.71 | 54409 |
| 32 | 34.43 | 577.39 | 76150 |
| 64 | 22.43 | 633.60 | 134631 |

Table 4.5: BSP benchmark data for an Intel Core 2 Q6600 Quad-core system, an AMD Phenom II 945e Quad-core system, and the Sun UltraSPARC T2 system consisting of eight cores with eight threads each.

Actual running time measurements are averaged over 100 SpMV multiplications.

Observed are errors of up to 11 times, compared to the actual measurements on the Intel platform. On the AMD platform slightly better error factors are obtained with a maximum of 8. Since these factors are not constant, the BSP model does not make very accurate predictions on the running time of the algorithm, when applied to the SpMV multiplication. The main cause is that the BSP benchmark measures process speed according to *dense* operations; sparse operations such as the SpMV multiply are known to perform at a fraction of peak performance, due to inefficient cache use [Tol97, VDY05, WOV+09]. If inefficient cache use indeed is the cause, prediction should be more successful for smaller and for structured sparse matrices, as also seen in Chapters 1,2 and 3. Input and output vectors corresponding to the Stanford matrix do not fit entirely into the L2 caches of both architectures, and the matrix, representing links within the Stanford domain from 2002, inherently is unstructured. On the other hand, the s3dkt3m2 matrix is smaller so its vectors do fit into L2 cache, while additionally the matrix comes from a FEM application on a regular grid and thus is structured. On all architectures, the prediction errors indeed are smaller for the s3dkt3m2 matrix, especially on the UltraSPARC T2 with large $p$. Other causes for discrepancy may include overhead caused by the MulticoreBSP library, which can depend on the parallel algorithm executed and can amplify the effects of cache misses. Hardware behaviour not taken into account with the BSP model (such as the case with the Intel Q6600), operating system or virtual machine jitter, can also be factors.

| | | Stanford | | |
|---|---|---|---|---|
| | | $p = 2$ | $p = 3$ | $p = 4$ |
| Partitioner output | superstep 1 | 2418012 | 1667496 | 1237054 |
| | superstep 2 | 225 | 600 | 607 |
| | $\max_s V(s)$ | 225 | 601 | 608 |
| Intel prediction | time | 4.1 | 3.1 | 2.4 |
| | error factor | 7.3 | 9.3 | 10.8 |
| | speedup | 1.9 | 2.6 | 3.2 |
| | error factor | 1.0 | 0.6 | 0.7 |
| AMD prediction | time | 2.8 | 2.7 | 2.8 |
| | error factor | 7.3 | 5.8 | 5.6 |
| | speedup | 2.0 | 2.1 | 2.0 |
| | error factor | 0.7 | 1.0 | 1.0 |
| Sun prediction | time | 50.8 | - | 28.6 |
| | error factor | 3.7 | - | 3.5 |
| | speedup | 1.9 | - | 3.4 |
| | error factor | 0.9 | - | 0.9 |

| | | s3dkt3m2 | | |
|---|---|---|---|---|
| | | $p = 2$ | $p = 3$ | $p = 4$ |
| Partitioner output | superstep 1 | 4120988 | 2719368 | 2056482 |
| | superstep 2 | 609 | 603 | 0 |
| | $\max_s V(s)$ | 609 | 1206 | 840 |
| Intel prediction | time | 7.1 | 5.1 | 3.9 |
| | error factor | 2.1 | 3.3 | 3.7 |
| | speedup | 1.8 | 2.5 | 3.2 |
| | error factor | 0.7 | 0.4 | 0.4 |
| AMD prediction | time | 4.9 | 4.6 | 4.3 |
| | error factor | 2.5 | 2.9 | 1.7 |
| | speedup | 1.9 | 2.0 | 2.1 |
| | error factor | 0.8 | 0.7 | 1.1 |
| Sun prediction | time | 86.9 | - | 47.1 |
| | error factor | 2.0 | - | 1.7 |
| | speedup | 1.8 | - | 3.4 |
| | error factor | 0.9 | - | 0.8 |

Table 4.6: Prediction of MulticoreBSP SpMV running time (in ms) and speedups with their error factors, defined as the measured value divided by the predicted value; an error factor of 7 in timing thus means the parallel run took 7 times longer than predicted, and an error factor of 1 indicates prediction matched exactly. Factors smaller than one indicate parallel execution was faster than predicted (or speedup was larger than predicted). Values are given for two test matrices. The BSP parameters $r, g, l$ for the architectures tested are taken from Table 4.5.

| | | Stanford | | |
|---|---|---|---|---|
| | | $p = 16$ | $p = 32$ | $p = 64$ |
| Partitioner output | superstep 1 | 317822 | 158942 | 79484 |
| | superstep 2 | 683 | 829 | 737 |
| | $\max_s V(s)$ | 684 | 829 | 738 |
| Sun prediction | time | 11.1 | 20.8 | 30.4 |
| | error factor | 3.5 | 2.1 | 1.5 |
| | speedup | 8.9 | 4.7 | 3.2 |
| | error factor | 0.8 | 1.7 | 2.0 |

| | | s3dkt3m2 | | |
|---|---|---|---|---|
| | | $p = 16$ | $p = 32$ | $p = 64$ |
| Partitioner output | superstep 1 | 516096 | 258048 | 129024 |
| | superstep 2 | 474 | 558 | 408 |
| | $\max_s V(s)$ | 804 | 624 | 424 |
| Sun prediction | time | 15.8 | 20.2 | 23.7 |
| | error factor | 2.2 | 2.1 | 2.0 |
| | speedup | 10.1 | 7.9 | 6.7 |
| | error factor | 0.7 | 1.0 | 1.0 |

Table 4.7: As Table 4.6, but for a larger number of concurrent threads as supported by the Sun UltraSPARC T2.

If library overheads or hardware peculiarities are the cause for bad prediction, and these overheads are constant regardless of the number of processors involved, speedup prediction should perform much better. Indeed, the speedup prediction error factors are much smaller compared to the predicted running time error factors. Also, speedups are more often underestimated (as opposed to overestimating absolute algorithm speed). It is possible to use the predicted speedup for instance to decide how many processes to allocate for a specific job, but otherwise it has limited practical applicability. Estimating speedups is more successful on the Stanford matrix and for the (non-NUMA) AMD architecture.

### 4.6.3 Sparse matrix–vector multiplication timings

The SpMV multiplication results for the Intel and AMD architectures are found in Table 4.8, and those for the Sun platform in Table 4.9. For the classical cache-based architectures, that is, the Intel and the AMD processors, speedups are only attained for larger matrices. This should come as no surprise, since for smaller matrices the input and output vectors both fit into L1 cache, which is not shared amongst cores. This differs for the UltraSPARC T2 architecture, where the L1 cache is much smaller and is shared amongst 8 threads simultaneously. For this machine, speedups are obtained even for the smaller matrices. The highest speedup obtained is 9.31 for the

| | Intel Q6600 | | | AMD 945e | | |
|---|---|---|---|---|---|---|
| Matrix | $p = 2$ | $p = 3$ | $p = 4$ | $p = 2$ | $p = 3$ | $p = 4$ |
| west0497 | 0.61 | 0.69 | 0.59 | 0.97 | **1.08** | 0.54 |
| fidap037 | 0.27 | 0.25 | 0.19 | 0.41 | 0.26 | 0.22 |
| s3rmt3m3 | 0.28 | 0.23 | 0.34 | 0.44 | 0.32 | 0.29 |
| memplus | 0.96 | 0.35 | 0.32 | 0.95 | 0.48 | 0.37 |
| cavity17 | 0.26 | 0.24 | 0.18 | 0.30 | 0.30 | 0.25 |
| bcsstk17 | 0.96 | 0.41 | **1.18** | **1.20** | 0.45 | 0.78 |
| lhr34 | **1.22** | 1.08 | 0.53 | 1.17 | **1.77** | 0.55 |
| bcsstk32 | 0.73 | 0.89 | 0.87 | 1.04 | 0.99 | **1.20** |
| nug30 | 0.69 | 0.64 | 0.46 | 0.74 | 0.75 | 0.59 |
| s3dkt3m2 | 1.20 | 1.09 | **1.26** | 1.46 | 1.29 | **2.33** |
| tbdlinux | **1.20** | 0.96 | 0.87 | **1.52** | 1.23 | 1.46 |
| stanford | 1.72 | 1.79 | **1.98** | 1.46 | 1.92 | **1.93** |
| stan-ber | 1.22 | 1.25 | **1.26** | 1.51 | 1.79 | **1.82** |
| cage14 | 0.85 | 0.93 | 0.78 | 1.14 | **1.46** | 1.34 |
| wikipedia-2005 | 1.97 | **2.31** | 2.21 | 2.33 | 2.68 | **3.34** |
| wikipedia-2006 | 1.83 | 2.06 | **2.17** | 1.38 | **2.17** | 2.00 |

Table 4.8: Measured speedup of MulticoreBSP sparse matrix–vector multiplication on the Intel Q6600 and AMD945e architectures. The speedup is calculated as the time taken for $p = 1$ divided by the time taken for the parallel run, and these timings are taken averaged over 100 SpMV runs. The best speedup for each architecture is shown in boldface.

Stanford-Berkeley matrix using $p = 32$ processes.

In general, better speedups are attained for the larger matrices, as the ratio of local work versus data movement due to communication is more favourable there. Increasing matrix size is not automatically beneficial for scalability, however; larger vectors may no longer fit into lower-level caches, and for higher-level caches the bandwidth is more limited, thus making minimisation of inter-core communication proportionally more important. In other words, for small matrices there is not enough work to attain efficient parallelisation, while for larger matrices communication costs may increase, as do the effects of inefficient cache use, especially when caches are shared.

### 4.6.4 The Fast Fourier transform

The fast Fourier transform (FFT) application from BSPedupack maps a complex-valued input vector of length $n$ to its discrete Fourier transform, and does this in two supersteps, if $p \leq \sqrt{n}$. For algorithm details, see Bisseling [Bis04, Chapter 3]; what now follows is but a brief description. The input vector is assumed to be distributed cyclically over the $p$ processors, and the algorithm starts with a parallel bit reversion,

| Matrix | $p = 2$ | $p = 4$ | $p = 8$ | $p = 16$ | $p = 32$ | $p = 64$ |
|---|---|---|---|---|---|---|
| west0497 | **1.20** | 0.94 | 0.81 | 0.62 | 0.39 | 0.24 |
| fidap037 | 0.58 | 0.54 | 0.64 | 0.50 | 0.68 | 0.37 |
| s3rmt3m3 | 0.91 | 1.42 | 1.47 | **1.63** | 0.93 | 0.68 |
| memplus | **1.50** | 1.18 | 1.01 | 0.86 | 0.78 | 0.44 |
| cavity17 | 0.85 | 0.98 | 1.27 | **1.33** | 0.72 | 0.60 |
| bcsstk17 | 1.79 | 2.95 | 2.57 | **2.96** | 2.20 | 1.36 |
| lhr34 | 1.98 | 2.60 | **4.34** | 4.33 | 3.68 | 2.19 |
| bcsstk32 | 1.59 | 2.71 | 4.68 | 6.40 | **7.02** | 4.43 |
| nug30 | 1.24 | **1.41** | 1.21 | 1.03 | 0.79 | 0.55 |
| s3dkt3m2 | 1.65 | 3.56 | 5.55 | 8.50 | **8.59** | 6.46 |
| tbdlinux | 1.89 | 3.01 | 3.97 | **4.19** | 2.75 | 1.51 |
| stanford | 1.64 | 3.05 | 5.03 | 6.80 | **8.02** | 6.42 |
| stan-ber | 1.64 | 2.95 | 5.00 | 7.31 | **9.31** | 9.27 |
| cage14 | 1.31 | 2.04 | 2.43 | **2.62** | 1.98 | 1.63 |
| wikipedia-2005 | 1.86 | 3.40 | **6.02** | 5.38 | 2.96 | 1.97 |
| wikipedia-2006 | 1.82 | 3.38 | **6.12** | 2.50 | 1.98 | 2.56 |

Table 4.9: Measured speedup of MulticoreBSP sparse matrix–vector multiplication on a Sun UltraSPARC T2 machine, obtained in the same way as for Table 4.8.

followed by a number of concurrent unordered sequential FFTs, and finally redistributes data and synchronises. The second and last superstep then proceeds with $p$ concurrent unordered sequential generalised FFTs. If $n$ is not large enough compared to the number of processors, more steps consisting of redistribution and concurrent unordered FFTs are required. Both the input vector size and the number of processors used by the algorithm are required to be powers of two. The advantage of this algorithm is that the sequential FFTs can be provided by external libraries like FFTW [FJ98], or can even be recursively parallelised, for example when applied within a hierarchy of parallel machines.

Table 4.10 shows scalability results of this algorithm. Speedups on the Intel and AMD platforms are only attained for large $n$; this must be due to the smaller vectors fitting into the local caches, so that communicating between cores is more expensive than doing the entire computation on a single core. The AMD 945e computation starts to attain speedups for $n \geq 2^{18}$, which corresponds to 4MB of data; for $p = 1$, this is eight times the size of an L2 cache, for $p = 4$, this is twice the size. The Intel computation starts getting speedups for $p = 2$ at 16MB of data, and for $p = 4$ at 32MB. Note that the Intel processor does not have an L3 cache and will communicate through main memory for $n > 2^{18}$. The AMD does have an L3 cache, but its capacity is exceeded for the same $n$ and it communicates through main memory as well.

Starting at $n = 2^{22}$ (64MB) for the AMD processor, or $n = 2^{23}$ (128MB) for the Intel processor, speedups become superlinear. This is explained by better cache use due to data locality: the first superstep of the parallel FFT algorithm performs $n/p$

unordered FFTs on contiguous blocks of size $p$, in parallel; that is, each processor performs $n/p^2$ unordered FFTs of small size, and thus performs extremely data-local computations. The second superstep performs $p$ unordered generalised FFTs of size $n/p$; these are less data-local and performance will drop if a vector of size $n/p$ no longer fits into local cache. This drop is in comparison to the first superstep, however, and does not seem to negatively affect overall scalability[7].

Table 4.11 reports speedups attained on the Sun architecture. Experiments start at a vector size $n = 2^{13}$ since the larger number of available threads would otherwise make more than one redistribution necessary. For this $n$ and $p = 64$, the only reported slowdown of a factor $0.8$ for the FFT on the UltraSPARC T2 occurs. While for the AMD and Intel processors no speedups could be obtained for the shorter input vectors, the Sun platform starts at a speedup of $5.3$ (for $p = 8$), with the lowest maximum speedup being $2.9$ for $n = 2^{15}$. As the problem size increases, the best speedups are attained for increasingly larger $p$. Superlinear speedups are attained as well, but, in contrast, only for $p < 16$ and sufficiently large $n$; and in all cases a larger (but sublinear) speedup can be obtained by using a larger $p$.

### 4.6.5 Dense LU decomposition

The BSP LU algorithm performs an LU decomposition of a dense $n \times n$ matrix $A$. It is a straightforward parallelisation of the well-known LU decomposition algorithm with partial pivoting, for example presented by Golub and van Loan [GVL96, Algorithm 3.4.1]. Note that this algorithm is not based on level-3 BLAS and that a better-performing algorithm may be obtained by using those; optimisation or parallelisation for high performance computing then typically requires hand-tuned [GG02] or auto-tuned [WPD01] software. The parallel algorithm used is described in detail by Bisseling [Bis04, Chapter 2]; again only a short description is given. A two-dimensional cyclic distribution of the input matrix over the $p = p_y p_x$ processes is assumed. This means that the $i$th row of $A$ is local to a process with $s_i = i \mod p_y$, and the $j$th column of $A$ is local to a process with $t_j = j \mod p_x$. Taking, e.g., the map $s = s_i p_x + t_j$, each combination of row and column properly corresponds to one unique process. For each stage $k$ from 0 to $n-1$, first a pivot element in the $k$th column is searched for, in parallel. When identified, the corresponding row is swapped with the $k$th row by using the get primitives. During this procedure, it is possible that $2p_x$ processes work and $(p_y - 2)p_x$ others idly stand by, thus negatively impacting the load balance. On the other hand, when the two swapped rows are local to the same processes, communication is local only, which may be preferable even if it causes load imbalance. These considerations do not contradict when $p_y$ is taken to be 1; then there is no load imbalance and only local computations during pivoting. When swapping is complete, all local elements of the lower-right $(n-k) \times (n-k+1)$ submatrix can be updated in parallel by caching (using direct-get primitives) the relevant

[7] See the results of $n \geq 2^{21}$ for the Intel architecture and $n \geq 2^{19}$ for the AMD; both exceed the L2 cache for the second superstep.

| | Intel Q6600 | | AMD 945e | |
|---|---|---|---|---|
| $\log_2 n$ | $p = 2$ | $p = 4$ | $p = 2$ | $p = 4$ |
| 9 | **1.1** | 0.9 | 0.8 | 0.6 |
| 10 | **1.1** | 0.9 | 0.6 | 0.7 |
| 11 | 0.3 | 0.2 | 0.7 | 0.8 |
| 12 | 0.4 | 0.5 | **1.2** | 1.1 |
| 13 | 0.5 | 0.3 | 0.9 | 0.9 |
| 14 | 0.7 | 0.4 | 1.0 | **1.2** |
| 15 | 0.7 | 0.4 | 0.9 | 1.0 |
| 16 | 0.4 | 0.2 | 0.8 | 0.8 |
| 17 | 0.5 | 0.3 | 0.9 | 1.0 |
| 18 | 0.6 | 0.4 | 1.0 | **1.2** |
| 19 | 1.0 | 0.6 | 1.3 | **1.5** |
| 20 | **1.5** | 0.7 | 1.7 | **2.2** |
| 21 | **1.7** | 1.5 | 1.9 | **2.7** |
| 22 | 1.8 | **1.9** | 2.3 | **3.2** |
| 23 | 2.3 | **2.4** | 2.6 | **3.6** |
| 24 | 2.2 | **2.2** | 2.9 | **4.9** |
| 25 | 2.6 | **3.2** | 2.9 | **5.0** |

Table 4.10: Measured speedups of MulticoreBSP FFT on two shared-memory architectures, the Intel Q6600 and the AMD 945e. The speedup is calculated as the time taken for $p = 1$ divided by the time taken for the parallel run, and these timings are taken averaged over 30 FFT runs. The input vector size varies between 8kB (for $n = 2^9$) to 512MB (for $n = 2^{25}$).

| $\log_2 n$ | $p = 2$ | $p = 4$ | $p = 8$ | $p = 16$ | $p = 32$ | $p = 64$ |
|---|---|---|---|---|---|---|
| 13 | 1.9 | 3.5 | **5.3** | 4.6 | 2.1 | 0.8 |
| 14 | 1.4 | 2.4 | 3.9 | **4.2** | 1.8 | 1.2 |
| 15 | 1.6 | 1.6 | 2.8 | 2.4 | **2.9** | 1.3 |
| 16 | 1.4 | 2.0 | 2.6 | **4.0** | 2.8 | 1.7 |
| 17 | 1.3 | 2.2 | 3.1 | 3.4 | **3.7** | 2.7 |
| 18 | 1.4 | 2.0 | **3.4** | 3.0 | 3.0 | 2.4 |
| 19 | 1.8 | 2.6 | 3.6 | **5.4** | 3.3 | 3.0 |
| 20 | 2.2 | 3.2 | 4.7 | **6.8** | 5.8 | 4.2 |
| 21 | 2.5 | 3.8 | 5.8 | 8.6 | **9.0** | 6.8 |
| 22 | 2.7 | 5.2 | 6.4 | 10.8 | **13.5** | 12.6 |
| 23 | 2.9 | 6.2 | 8.3 | 11.1 | 18.5 | **22.0** |
| 24 | 2.9 | 5.9 | 10.5 | 13.6 | 15.4 | **23.9** |
| 25 | 2.9 | 5.9 | 9.8 | 13.0 | 16.4 | **16.7** |

Table 4.11: Measured speedups of MulticoreBSP FFT on the Sun UltraSPARC T2, obtained in the same way as for Table 4.10. The input vector size varies between 128kB (for $n = 2^{13}$) to 512MB (for $n = 2^{25}$).

ranges of the $k$th row and the $k$th column, including the pivot value. After updating, the algorithm increments $k$ and continues with the next stage.

The LU algorithm differs from SpMV and FFT mainly in that the number of supersteps, as well as the amount of communication, are linearly related to $n$ instead of being constant; the overhead of the algorithm is thus expected to be higher. Another difference is that local matrix entries are reused in the calculation, instead of only the components of input or output vectors; this makes it easier to attain good speedups. In the experiments, $A$ is taken as a random matrix and 30 decompositions are timed, and the average time is reported. The matrix size must be divisible by both $p_y$ and $p_x$; in experiments the sizes $n = 120, 600, 1200$ are taken for this reason.

For the smallest size tested, $n = 120$, no speedup was attained on the Intel and AMD architectures, while the UltraSPARC T2 architecture gained a maximum factor $1.4$ for $p_y = 3$ and $p_x = 1$ and did not get slowdowns as long as $p_y p_x < 8$; these results are not included in tabular form. Speedups for $n = 600, 1200$ are reported in Table 4.12 and 4.13. For the Intel processor, no speedup is attained for $n = 600$. The AMD processor performs somewhat better with a factor $1.1$ for $2 \cdot 1$ processors. As expected, the results improve for yet larger $n$; both these architectures perform up to a factor $1.4$ better, again by using $2 \cdot 1$ processors. The Sun platform performs somewhat better, reporting speedups up to $3.0$ for $n = 600$ and $6.7$ for $n = 1200$, using $2 \cdot 3$, respectively, $6 \cdot 2$ processors. Larger $n$ will likely yield larger speedups using more processors, on this architecture. All architectures seem to prefer $p_y > p_x$, even though this causes load-imbalance as discussed earlier.

## 4.7 Conclusions

It was shown that the Bulk Synchronous Parallel [Val90] programming paradigm can be efficiently used for shared-memory parallel programming. A proof-of-concept BSP library, MulticoreBSP, targeted towards shared-memory architectures, has been implemented and is freely available. It differs from previous work in that it exploits the shared-memory architecture solely through a *direct-get* method, otherwise assuming processes only have local memory and communicate only through the original distributed-memory primitives: the *get*, the *put* and the *send*. A sparse matrix–vector multiplication (SpMV) algorithm has been described and implemented using this library, similar to the algorithms introduced in BSPedupack [Bis04], except that the number of supersteps has been reduced by use of the direct-get. Experiments on both the predictability and scalability of this algorithm have been performed. Predictions of the run time leave much to be desired; predictions of the speedup are more accurate.

Actual speedup measurements have been carried out on three different architectures, two quad-core machines (Intel and AMD) and one highly-threaded machine (Sun). Results of the SpMV multiplication show modest overall speedup, with the higher speedups reserved for the larger matrices, showing a speedup of $3.34$ for the wikipedia-2005 matrix with $p = 4$, and superlinear speedup of $2.33$ for $p = 2$ on

| | | Intel Q6600 | | | | AMD 945e | | | |
|---|---|---|---|---|---|---|---|---|---|
| $600 \times 600$ : | | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 |
| | 1 | 1.0 | 0.7 | 0.6 | 0.5 | 1.0 | 0.7 | 0.5 | 0.3 |
| | 2 | 0.8 | 0.5 | *0.3* | *0.3* | **1.1** | 0.4 | *0.2* | *0.2* |
| | 3 | 0.7 | *0.4* | *0.3* | *0.2* | 0.6 | *0.3* | *0.2* | *0.2* |
| | 4 | 0.4 | *0.3* | *0.2* | *0.2* | 0.4 | *0.2* | *0.2* | *0.2* |
| $1200 \times 1200$ : | | | | | | | | | |
| | 1 | 1.0 | 1.2 | 1.0 | 0.9 | 1.0 | 1.1 | 0.8 | 0.5 |
| | 2 | **1.4** | 1.2 | *0.9* | *0.8* | **1.4** | 0.7 | *0.4* | *0.4* |
| | 3 | 1.2 | *1.0* | *0.8* | *0.6* | 1.2 | *0.5* | *0.4* | *0.3* |
| | 4 | 1.3 | *0.9* | *0.7* | *0.6* | 0.8 | *0.5* | *0.3* | *0.3* |

Table 4.12: Measured speedups for the MulticoreBSP LU algorithm on the Intel Q6600 and the AMD 945e processors. The algorithm has been run for square dense matrices of size 600 and 1200. The matrices are distributed over processes both row-wise and column-wise; the number of processes $p_x$ used in the column direction are displayed horizontally in the four parts of the table, and the number of processes $p_y$ used in the row direction are displayed vertically. The total number of processes $p = p_x p_y$ thus exceeds the available number of cores (4) in half the cases presented; those speedups are printed in italic. The largest speedup for each of the four parts is printed in boldface.

| $600 \times 600$: | | 1 | 2 | 3 | 4 | 5 | 6 | 8 |
|---|---|---|---|---|---|---|---|---|
| | 1 | 1.0 | 1.8 | 2.3 | 2.6 | 2.8 | 2.2 | 2.1 |
| | 2 | 1.8 | 2.8 | **3.0** | 2.9 | 2.6 | 2.0 | 1.8 |
| | 3 | 2.4 | 3.0 | 2.9 | 2.4 | 2.2 | 1.6 | 1.4 |
| | 4 | 2.7 | 2.9 | 2.4 | 2.0 | 1.8 | 1.4 | 1.2 |
| | 5 | 2.9 | 2.6 | 2.2 | 1.9 | 1.6 | 1.2 | 1.0 |
| | 6 | 3.0 | 2.4 | 2.0 | 1.7 | 1.4 | 1.0 | 0.8 |
| | 8 | 2.9 | 2.2 | 1.6 | 1.3 | 1.0 | 0.8 | 0.7 |
| $1200 \times 1200$: | | | | | | | | |
| | 1 | 1.0 | 2.2 | 3.3 | 3.9 | 4.5 | 5.0 | 5.4 |
| | 2 | 2.1 | 4.3 | 5.4 | 5.9 | 6.1 | 6.1 | 5.7 |
| | 3 | 3.0 | 5.6 | 6.1 | 6.4 | 6.2 | 5.6 | 5.1 |
| | 4 | 4.1 | 6.3 | 6.3 | 6.2 | 5.7 | 5.4 | 4.7 |
| | 5 | 4.9 | 6.6 | 6.4 | 5.9 | 5.3 | 4.9 | 4.1 |
| | 6 | 5.5 | **6.7** | 6.1 | 5.6 | 5.0 | 4.3 | 3.7 |
| | 8 | 6.3 | 6.2 | 5.4 | 4.6 | 4.1 | 3.6 | 2.8 |

Table 4.13: Similar to Table 4.12, but for the Sun UltraSPARC T2 architecture.

the same matrix, both on the AMD architecture. Speedups for SpMV multiplication on the Sun UltraSPARC T2 processor are very good when compared to those of the other architectures for $p \leq 4$. For larger $p$, performance increases even more, with a largest measured speedup of $9.31$ for $p = 32$ threads, on the Stanford-Berkeley link matrix; this is also the matrix which was hardest to partition (see Section 1.5 and 2.4). This might not come as a surprise, since difficulty in partitioning implies that there is no apparent data locality to be found, thus causing many irregular accesses which is precisely where the UltraSPARC T2 architecture performs well. For the larger test matrices, the performance drops, gaining only good speedups for small $p$. By the same reasoning, it appears that for these matrices there is enough data locality to cause the processor core to saturate on a lower number of threads.

Other example applications have been implemented and experimented with as well. They are the fast Fourier transform and dense LU decomposition. Both are easier to parallelise: most data elements are used several times, instead of only once as is the case with SpMV multiplication. Good speedup, even superlinear speedup, is attained for the FFT: up to $6.2$ for $p = 4$ on the Sun UltraSPARC T2, $5.0$ for $p = 4$ on the AMD 945e, and $2.6$ for $p = 2$ on the Intel Q6600. The maximum speedup for the Intel is $3.2$ out of the maximum expected of $4$, and that of the UltraSPARC T2 hovers around a factor $24$, which still seems to increase for larger $n$. The LU decomposition performs less well on the AMD and Intel architectures. This is probably due to the lack of optimisation of the algorithm: during each iteration of the algorithm, threads move along the pivot column unchecked while updating the local submatrices, which generally do not fit into local caches. This may yield much, possibly conflicting, data movement for each thread in shared caches or main memory while inside a computation superstep; this points to another large difference between distributed-memory and shared-memory BSP. Solutions lie in employing existing sequential optimisation techniques (such as blocking) to limit unnecessary data movement, which should then lead to speedups when using multiple threads, and not just in a constant factor gain. Supporting this, on the UltraSPARC T2 where locality actually reduces potential scalability, the speedups are much more pronounced, going up to a factor $6.7$. Also for smaller problems, the UltraSPARC T2 attains a much better speedup than the more traditional Intel and AMD platforms. Latency hiding thus gives ample opportunity for speedups on highly unstructured problems, but it seems hard to predict the size of such speedups within the BSP model used here.

In general, it is shown that the BSP model, library, and algorithms can be adapted for shared-memory architectures, and that these algorithms can work well, as tested on a variety of architectures. Similarities, but also some of the large differences have been discussed, leading to the observation that sequential optimisation is of increasing importance in shared-memory computing; gains in data movement result in less contention between memory accesses of many threads, resulting in better scalability. Nevertheless, such optimisations will benefit distributed-memory computing as well. Hence, the same efficient and scalable BSP algorithms can be employed, for both distributed-memory and shared-memory architectures.

### 4.7.1 Future work

While the MulticoreBSP library shows that the BSP model is valid for current shared-memory systems, the Java implementation is not viable for high-performance computing. While it attains modest speedups and is proper for concept testing and educational use, the algorithms are seen to perform slower than expected. Construction of a similar, more competitive library in, e.g., the C++ programming language should be worthwhile. Such a library can build forth on existing technologies such as POSIX threads, and may be augmented with the Message Passing Interface (MPI), or distributed-memory BSP; combining both can result in a BSP library which is able to switch as required between shared-memory and distributed-memory implementations. Moreover, by incorporating the Multi-BSP model [Val08] instead of the flat BSP model, such a library can automatically distribute a BSP algorithm over a hybrid distributed-memory and shared-memory system, thereby completely eliminating the need for explicit hybrid parallel programming.

While predictability here only has been investigated for the SpMV algorithm, results indicated that processor speed is not always properly measured in flops per second. Instead, timings based on irregular data accesses may be of more importance, and may yield better predictions of BSP algorithms working on sparse problems. Such benchmarks may be integrated into the BSP benchmarking application, which would also benefit from an update to the Multi-BSP model; message size can be varied so that properties of the different levels of the memory hierarchy can be measured. Ideally, however, this also requires information on the exact memory size of each such memory level.

Regarding the SpMV multiplication specifically, interleaving the querying of $\pi_r^s$ for locality of output vector elements results in a constant overhead during execution of the SpMV multiplication. An alternative is to buffer the remote elements locally, thus removing this overhead, which results in a more efficient kernel. After execution of this kernel, the local buffer is read out and non-local elements can again be sent out using the send primitive. The resulting *fully buffered* SpMV multiplication kernel enables the use of external multiplication code, which may perform additional optimisation; such as, e.g., OSKI [VDY05] or the techniques from Chapter 3. Using a full buffer also enables trivial application of reordering techniques from Chapter 1 and 2. To attain better scalability in the half-buffered case, further optimisation surely is required. Applying partitioning to minimise communication between computing cores is not enough, as data access patterns of the input vector are not improved while bandwidth becomes more limited as more cores are involved in the computation. Future work should be directed towards combining communication minimisation on all levels, including enhancement of cache use, e.g., by reordering or by adapting the sparse matrix storage scheme [BFF$^+$09, MFT$^+$10, VDY05] (or both), while still retaining a way to perform fan-in on output vectors. This is discussed in more detail in Chapter 5.

## Acknowledgements

The results presented here were established with support from the Center for Computing and Communication of RWTH Aachen University, who graciously provided access to their Sun UltraSPARC T2 cluster and provided assistance with experiments. We also extend our special thanks to Ruud van der Pas from Oracle Corporation, who offered extensive feedback on this chapter, as well as valuable insights on the UltraSPARC T2 architecture and its performance during the experiments described here.

# Chapter 5

# Integration with the cache-oblivious method

The cache-oblivious methods introduced in Chapter 1 and 2 can be used in a parallel setting as well. As discussed in the previous chapter, in Section 4.5.1, a Bulk Synchronous Parallel algorithm requires that each process $s$ stores a local matrix $A_s$ consisting of a subset of nonzeroes from the global matrix $A$. Using either full buffering or partial buffering of input and output vectors, a local SpMV multiplication is performed on $A_s$. A straightforward combination of reordering and parallel techniques then is to perform local SpMV multiplications on $PA_sQ$, with $PA_sQ$ in (doubly) SBD form. This chapter builds on this idea, and constructs a scalable cache-oblivious parallel SpMV for shared-memory systems.

Other work in this same area includes using space-filling curves to achieve data-locality, while distribution of work is based on the same curve. For instance, Martone et al. [MFT+10] use the Morton curve [Mor66] combined with 1D row distributions. Buluç et al. [BFF+09] use a 2D distribution based on the same curve, and also employ index compression to reduce memory bandwidth consumption. Williams et al. [WOV+09] do not use space-filling curves, but employ many sparse optimisation techniques earlier applied in sequential auto-tuning [VDY05, Im00] and combine these with further architecture-aware optimisations: for instance, on cc-NUMA systems, they explicitly control which thread runs on which core, and adapt the data distribution accordingly. As in the sequential case, the difference with the method described in this chapter, is that speed gains will be mainly due to changing the structure of the input matrix, and not by adapting data structures.

## 5.1 Exploiting matrix reordering for parallelism

The easiest way of combining reordering with parallel SpMV multiplication is to reorder with $p \to \infty$, and then assign contiguous parts of the reordered matrix to the

actual processes. For example, Figure 1.4 showed a 1D reordering based on $p = 16$, while the four differently coloured parts indicate which data goes to which processor when four different processing units are available.

This is more accurately described as follows. Let the array $r$ store the row-wise starting points of the various blocks identifiable in the DSBD structure, and let $c$ be similarly defined for the column-wise starting point. Note that there are exactly $2p-1$ of such starting points. Additionally, let $r_{2p} = m$ and $c_{2p} = n$. Then, for even $i$, the index block $[r_i, r_{i+1}] \times [c_i, c_{i+1}]$ corresponds to the $i$th pure block of $PAQ$; for odd $i$, nonzeroes with row index in $[r_i, r_{i+1}]$ or nonzeroes with column index in $[c_i, c_{i+1}]$ correspond to separator blocks. Suppose $PAQ$ is in DSBD form with $p$ parts, while for parallel execution there are $t$ threads available, with $t < p$. Then the array $r$ of length $2p$ can be *upscaled* to an array $\tilde{r}$ of length $2t$ by copying each $\frac{p}{t}$th element from $r$ to $\tilde{r}$, and similarly for upscaling $c$ to $\tilde{c}$. Nonzeroes in the $i$th pure block according to $\tilde{r}, \tilde{c}$ were originally assigned to $p/t$ processors from the (not necessarily consecutive) range $[0, p-1]$, which now all should map to processor $i$ from the range $[0, t-1]$; all maps from Section 4.5.1 can be adapted using these new processor assignments. Given these adaptations, the Bulk Synchronous Parallel SpMV multiplication scheme from the previous chapter can be applied directly. While on distributed-memory architectures the SBD form cannot be further exploited than this, it is possible to enhance performance further on shared-memory systems.

In particular, the matrices need no longer be localised for use with local input and output vectors. Instead, one global input vector and one global output vector can be used. Combined with reordering, accesses on these global vectors are still localised in exactly the same manner as cache usage for sequential computation was optimised. However, there is a complication with global addressing: concurrent writes to the output vector can occur, resulting in undefined algorithm behaviour and errors in output. The troublesome areas correspond to the cut rows; that is, the row-indices corresponding to the ranges $[\tilde{r}_i, \tilde{r}_{i+1}]$ with $i$ odd. In these areas, the owner of a matrix row is not uniquely determined; that is, $\pi_y([\tilde{r}_i, \tilde{r}_{i+1}])$, with $i$ odd, maps to more than one processor. Note that many processors may have nonzeroes on a specific separator row, but only one of those processors has ownership over the output vector component corresponding to that row. This ownership is given by the mapping $\pi_y$.

A simple way of preventing concurrent writes is to let each processor cycle through the writeable output elements according to $\pi_y$: in the first superstep, processor $s$ writes output elements $i$ for which $\pi_y(i) = s$, while in the next superstep that processor writes to elements for which $\pi_y(i) = (s+1) \mod t$. This is repeated until $t-1$ supersteps have been performed; for a small number of threads $t$ this is an acceptable cost, while for larger $t$, the full BSP SpMV multiplication (see Algorithm 6) will be preferred due to its constant number of supersteps. It must be noted that performance now not only depends on the sparse matrix structure, but also on the distribution of the output vector, which brings further room for improvement. Mondriaan, for example, when having obtained a sparse matrix partitioning, tries to further minimise communication costs by constructing vector distributions in a smart way [BM05]. Due to this, the consecutive values of $\pi_y$ are, in general, quite irregular. This means

that threads still disrupt each other during multiplication on separator areas, as many elements sharing a cache line will not have the same processor as owner. A solution is to refine the permutation found by the reordering method, by ordering rows in separator areas according to their owner in $\pi_y$. After this operation, accesses to the output vector in separator areas are more localised, and conflicts only occur on transitions between processor owners. Optionally, these conflicts can be prevented by adding empty matrix rows in between transitions, so that each thread is guaranteed to operate on a different cache line [WOV+09]. In any case, conflicts can be reduced significantly by additional refinement of reorderings, and the final shared-memory parallel scheme is summarised in Algorithm 7.

## 5.2 Experiments

The parallel scheme has been applied on two architectures, namely the AMD Phenom II 945e, and the Intel Q6600. See Section I.1.3 for a complete description of these architectures. Comparisons to sequential schemes will be made, namely the Triplet, Compressed Row Storage (CRS), and Incremental CRS (ICRS) methods. Timings from the Hilbert scheme introduced in Chapter 3 are included as well. Also, the parallel code running on multiple processors is compared to the same code running on one processor; this is equivalent to a sequential method based on reordering, backed by the BICRS data structure without explicit block-based ordering, with negligible overhead from parallel codes. An initial version of the parallel scheme described in this chapter has been implemented in C++ using POSIX threads, and is part of the same sparse library also used in other experiments throughout this thesis[1]; thus, for all experiments, the exact same SpMV multiplication kernels are used. For these multi-threaded experiments, wall-clock timing had to be based on the system clock which has a low resolution of one second. To be able to nonetheless have accurate timings of up to a millisecond, timings are averaged over $1000$ runs instead of the $100$ used in most other experiments in this thesis. The matrices used in experiments are s3dkt3m2 and GL7d18. The s3dkt3m2 matrix is a structured square matrix of size $90449$, and contains about $1.9$ million nonzeroes; the GL7d18 matrix is an unstructured $1955309 \times 1548650$ matrix, containing about $35.6$ million nonzeroes. The structured matrix arises from a FEM application, while the GL7d18 matrix arises from pure mathematics.

Both test matrices have been reordered (and partitioned) by using Mondriaan with the load imbalance parameter $\epsilon = 0.1$ and with $p = 4, 16, 32, 64$. The partitioning strategy used is the Mondriaan scheme, the storage scheme for each pure thread-local matrix (i.e., the matrices $\tilde{A}_s$) is BICRS, and the storage scheme for nonzeroes on row-wise separators (the matrices in $S_s$, see Algorithm 7) is ICCS. As both test architectures are quad-core machines, the number of threads tested is $t = 1, 2, 4$. The various $p$ with $t = 1$ correspond to the reordering method described in Chapter 2,

[1] Freely available via `http://albert-jan.yzelman.net/software`

| | Intel Q6600 | | AMD 945e | |
|---|---|---|---|---|
| | s3dkt3m2 | GL7d18 | s3dkt3m2 | GL7d18 |
| Triplet | 18 | 1323 | 12 | 466 |
| CRS | 14 | 794 | 12 | 437 |
| ICRS | *13* | 856 | *9* | 610 |
| Hilbert | 28 | *415* | 16 | *349* |

Table 5.1: Sequential results for SpMV multiplication. Results for the Intel Q6600 are on the left, while those for the AMD 945e are found on the right. The fastest sequential results are printed in italic. All timings are in milliseconds.

without using block-based structures. To be complete, the sequential experiments on the original matrices have been repeated and results are reported in Table 5.1. Results for the parallel code on the Intel and AMD architectures are given in Table 5.2.

On the Intel architecture, no speedups versus optimised sequential codes are obtained for the s3dkt3m2 matrix; the same is true when instead comparing to $t = 1$. For the GL7d18 matrix, a speedup of $1.5$ (for $t = 2$) is obtained, compared to the Hilbert scheme. When comparing only against results with $t = 1$, speedups are in between $1.3$ ($p = 4$) and $1.8$ ($p = 16$) for $t = 2$. The speedups for the full number of threads, $t = 4$, are less impressive: those are in between $1.2$ ($p = 32$) and $1.6$ ($p = 4$). Note that nevertheless, when reordering with high enough $p$, parallel code on the reordered matrices outperforms the sequential cache-oblivious methods on this larger matrix.

Better results are obtained on the AMD platform, where modest speedups are attained on the smaller and structured s3dkt3m2 matrix. For this matrix, the best results are obtained for the maximum number of threads ($t = 4$), where speedups go up to a factor $1.5$ compared to sequential code. Compared to $t = 1$ no slowdowns are reported at all, and speedups lie in between $1.6$ (for $p = 16$) and $2.3$ (with $p = 32, t = 4$). For the GL7d18 matrix, the maximum speedup against the best sequential time is $1.8$, but there are also some slowdowns for low $p$ and low $t$.[2] Parallel code with $t = 2$ on the reordered matrices is only effective for $p = 32$. With $t = 4$, however, the parallel code always outperforms the sequential Hilbert code. Compared to $t = 1$ instead of the Hilbert scheme, speedups lie in the range of $1$ and $1.9$; for $p = 16$ and $t = 2$ a small slowdown is observed.

## 5.3 Conclusions

The matrix reordering method from Chapter 1 and 2 has been refined for exploiting parallelism, by also considering the effects of reordering on the distribution of the components of the output vector. A simple synchronising multiplication algo-

[2] In fact, reordering with $t = 1$ without any of the blocking schemes from Chapter 2 is less efficient than the Hilbert scheme from Chapter 3.

**Algorithm 7** Parallel, non-BSP SpMV multiplication on shared-memory systems

Let $\tilde{r}$, $\pi_{\tilde{A}}$, and $\pi_y$ as described in the text, and available for this algorithm. Let $\tilde{A}$ furthermore be the upscaled reordered matrix, and let $t$ be the total number of threads. Initialisation will build, for each thread $s$, a main matrix $\tilde{A}_s$ as well as $t-1$ smaller matrices stored in the array $S_s$, where the nonzeroes in the matrix $(S_s)_i$ will correspond to the nonzeroes of thread $s$ contained in row-wise separators, for which the output vector element is distributed to processor $i$.

`Initialisation:`

1: **for each** thread $s$ **do**
2: Allocate $\tilde{A}_s$, the main sparse matrix assigned to thread $s$.
3: Allocate $S_s$ as an array of size $t$ of sparse matrices, storing separator nonzeroes.
4: **for each** $i$th row-wise separator in $\tilde{A}$ **do**
5: Find a permutation of the row indices $[\tilde{r}_{2i+1}, \tilde{r}_{2i+2}]$ so that the corresponding values in $\pi_y([\tilde{r}_{2i+1}, \tilde{r}_{2i+2}]$ are ordered from low to high; apply this sub-permutation to $\tilde{A}$, $\pi_{\tilde{A}}$, and $\pi_y$.
6: **for each** nonzero $a_{ij}$ from $\tilde{A}$ **do**
7: **if** $\pi_{\tilde{A}(i,j)} = \pi_y(i)$ **then**
8: Assign $a_{ij}$ to $\tilde{A}_{\pi_{\tilde{A}(i,j)}}$.
9: **else**
10: Assign $a_{ij}$ to $(S_{\pi_{\tilde{A}(i,j)}})_{\pi_y(i)}$.

After initialisation, each thread $s$ executes the SpMV multiplication code of the various matrices created during initialisation. A call to an SpMV multiplication kernel of a matrix $A$, is simply denoted by $y = Ax$. The input vector $x$ and output vector $y$ are assumed to be globally addressable.

`Multiplication:`

1: Execute $y = \tilde{A}_s x$.
2: Synchronise.
3: **for** $k = 1$ **to** $t - 1$ **do**
4: Set $j = (s + k) \mod t$.
5: Execute $y = (S_s)_t x$.
6: Synchronise.

Intel Core 2 Q6600

| s3dkt3m2 $t\backslash p$ | 4 | 16 | 32 | 64 |
|---|---|---|---|---|
| 1 | 17 | *16* | 18 | 17 |
| 2 | 17 | *16* | 18 | 17 |
| 4 | 20 | *18* | 22 | 21 |

| GL7d18 $t\backslash p$ | 4 | 16 | 32 | 64 |
|---|---|---|---|---|
| 1 | 906 | 633 | 492 | *486* |
| 2 | 718 | 347 | 345 | *285* |
| 4 | 583 | 491 | 398 | *385* |

AMD Phenom II 945e

| $t\backslash p$ | 4 | 16 | 32 | 64 |
|---|---|---|---|---|
| 1 | *11* | *11* | 14 | *11* |
| 2 | 8 | *7* | 9 | 8 |
| 4 | *6* | 7 | *6* | *6* |

| $t\backslash p$ | 4 | 16 | 32 | 64 |
|---|---|---|---|---|
| 1 | 482 | 373 | *352* | 372 |
| 2 | 333 | 376 | *236* | 357 |
| 4 | 250 | 200 | *199* | 237 |

Table 5.2: Results for the Intel Core 2 Q6600 (left) and the AMD Phenom II 945e (right). The number $p$, displayed horizontally, indicates the number of parts used for the reordering scheme. The actual number of threads $t$ used in parallel SpMV multiplication are displayed vertically. Fastest timings for a given $t$ are printed in italic. All timings are in milliseconds.

rithm for globally addressable shared-memory architectures is introduced, which is capable of attaining speedups around a factor 2 when using 4 threads, compared to efficient sequential code. Compared to plain CRS implementations, the highest observed speedup is superlinear with a factor of 2.8 (with the GL7d18 matrix on the Intel Q6600, for $t = 2$ and $p = 64$). These speedups are obtained using flat data structures (ICCS and BICRS), without any sparse blocking as, e.g., introduced in Chapter 2. A reordering with a large number of parts was used, which was upscaled, and used in a straightforward parallel SpMV multiply which avoids row-wise writing conflicts by synchronising. Any column-wise distribution is not explicitly used by this parallel algorithm, as the data locality of the input vector is already improved by reordering. This already results in oblivious minimisation of communication during multiplication; that is, minimising communication on shared-memory systems directly translates to improving data locality.

## 5.4 Future work

The methods presented here are preliminary, as many avenues for improvement can be identified. As a first, since row-wise separators cause the need for a synchronised parallel SpMV multiply, it might be interesting to investigate a one-dimensional reordering scheme based on the column-net hypergraph model. Similarly, as partitioning is assumed to be based on recursive bipartitioning, it is possible that only a subset of processors have nonzeroes in a specific row-wise separator area. This means that a global synchronisation is wasteful, but even worse: due to the synchronising parallel SpMV algorithm, load imbalance is introduced between the processors with nonzeroes in the separator, and those without. The load-imbalance parameter only forces

a balance between vertices assigned to different sets, but it is possible many of the involved hyperedges are, in fact, cut due to this partitioning. Algorithm 7 does not take this into account, and by synchronising several times between row-wise separators, the assigned work is cut in a way not foreseen by the hypergraph partitioner; in other words, it only guaranteed that the amount of work in both the pure matrix and the separator corresponding to a specific bipartitioning is divided evenly. Note that employing subset synchronisation so that only the processors actually involved in a given separator wait on each other, also would fix the load-balance issue. Judging by the results on the cc-NUMA Intel architecture, which does not perform well with $4$ threads, adding this feature to the parallel code must be a worthwhile investment.

As due to the parallel algorithm used here, the vector distribution is also taken into account for reordering purposes, the integration of the vector distribution into the sparse matrix partitioning process and the resulting interplay with the reordering method becomes an interesting field for future work as well. Uçar and Aykanat [UA04] already presented various advanced methods for post-processing a matrix distribution to minimise more than just the total communication volume; and in fact, balancing and minimising the number of synchronisations during the SpMV algorithm presented here, is one of the metrics they take into account. On a deeper implementation level, this algorithm does not use any low-level optimisation such as sparse blocking, prefetching, index compression, et cetera. Integration is again possible by using external auto-tuning [VDY05] or cache-oblivious [BFF+09] software for multiplying the $\tilde{A}_s$, but Williams et al. [WOV+09] show that direct architecture-aware implementations can attain even better performance.

# Appendix A

# Sparse matrix data structures

Throughout this thesis, several sparse matrix data structures have been introduced and used within larger schemes, whether it be reordering, block-based traversal, multiplication based on space-filling curves, or parallel multiplication. In this appendix, the main sparse matrix data structures are investigated in more detail, and a performance summary in terms of both memory usage and complexity of the resulting SpMV multiplication kernel is given in Table A.1. Also, the multiplication kernels are given in more detail than was done previously, and an in-depth motivation[1] for the reported performance bounds is included. Implementation of the data structures and corresponding SpMV multiplication kernels are freely available, and can be found at:

`http://albert-jan.yzelman.net/software`

## A.1 Deriving memory usage and complexity

All sparse matrix data structures in principle use an array to store nonzero values in; this is denoted by $V$, in all cases. Whether it be compressed or otherwise adapted, row and column indices also correspond to arrays; these are $I$ for the rows, and $J$ for columns. The sparse input matrix is of size $m \times n$ and contains $nz$ nonzeroes. The dense input vector is $x$ and the output vector is $y$.

To enable a fair comparison of algorithmic complexities of the multiplication kernels, only a small number of functions are allowed. These are the following. In the first place there is the floating-point *multiply/add*; this represents the calculation of $y_i + a_{ij}x_j$ and storing this value at $y_i$. Second are *stream* operations: these include streaming an array of values to the processor, where they are used in a computational kernel such as the floating-point multiply/add. The terms under which array access is considered a stream, is that the access be linear and that elements are only considered

[1] This does require some knowledge on lower-level programming, since the differences between the SpMV multiplication kernels are subtle. Listings are kept close to C for this reason.

precisely once. For example, getting nonzeroes from $V$ in the order they are stored in is considered streaming, while getting elements from the input vector $x$ in the usual irregular pattern, is not a stream operation. A further operation is a while-loop on a single *condition*. This includes both checking for the condition and jumping based on its result.

Further there is the *pointer add/assign*; a pointer is in essence a memory address of, for instance, the start of an array (which is the same as a pointer to the first element of that array). When writing efficient code, it is worthwhile to program pointer movement explicitly, instead of referring to arrays by indices; see Section A.1.3 on the ICRS data structure for an example. The pointer add/assign (e.g., `px=x+1`) means that a value (1) is added to an existing pointer ($x$), the result of which is stored in a second pointer ($px$). Pointers can be 'de-referenced' to access the value they point at. This is denoted by a star in front of the pointer (e.g., `*px` which in this example equals $x_1$). The last operation is the *pointer add*, which adds a value to an existing pointer and stores the result at the same location, e.g., `px+=7` which increments the $px$ with 7. This is included as a separate instruction as on some (older) architectures, this can be done faster than a pointer add/assign. All SpMV kernels are written using only the constructs described here. Any initialisation required is not counted in Table A.1, but is added in parentheses in the kernel listings.

### A.1.1 The Triplet data structure

The Triplet scheme, otherwise also known as the coordinate (COO) scheme, stores three values for each nonzero: its row index, its column index, and its nonzero value. The nonzero values are stored in the array $V$, the row indices in $I$, and the column indices in $J$. All these arrays are of size *nz*, and thus the memory requirement equals $\mathcal{O}(3\text{nz})$. The SpMV multiplication kernel in terms of streams, multiply/adds and basic pointer arithmetic is the following:

```
(pV = V; Vend = V + nz; pI = I; pJ = j;)
while( pV < Vend ) {
     py  =  y + *pI++;
     px  =  x + *pJ++;
    *py += *pV++ * *px;
}
```

The values from $V$, $I$, and $J$ are all streamed, while the memory pointers $I$, $J$, $pV$, $Vend$, $px$, $py$, $y$ and $x$ are used in each iteration of the for loop; thus the kernel requires 8 memory registers. Counting the different operations appearing, the algorithm is seen to attain the complexity given in Table A.1.

### A.1.2 The CRS data structure

The Compressed Row Storage (CRS, also known as Compressed Sparse Row), for instance described by Bai et al. [BDD+00], assumes a row-major order on the nonzero

| | Triplet | (ZZ-)CRS | (ZZ-)ICRS | $\text{ICRS}_\text{n}$ | BICRS |
|---|---|---|---|---|---|
| Data words | $nz$ | $nz$ | $nz$ | $nz$ | $nz$ |
| Index words | $2nz$ | $nz + m + 1$ | $nz + \tilde{m}$ | $nz$ | $nz + j$ |
| Registers | 8 | 9 | 8 (9) | 7 | 8 |
| Multiply/add | $nz$ | $nz$ | $nz$ | $nz$ | $nz$ |
| Streams | $3nz$ | $2nz + m$ | $2nz + \tilde{m}$ | $2nz$ | $2nz + j$ |
| Conditionals | $nz$ | $nz + m$ | $nz + \tilde{m}$ | $nz + m$ | $nz + j$ |
| Pointer add/assign | $2nz$ | $nz + m$ | 0 | 0 | 0 |
| Pointer add | 0 | $m$ | $nz + 2\tilde{m}$ | $nz + 2m$ | $nz + 2j$ |

Table A.1: Comparison of different SpMV multiplication kernels. The first category (in between the horizontal separator lines) considers memory usage, while the second considers operation costs. ZZ-CRS uses the same kernel as CRS and is subject to the same storage requirements as well; the same is true for ZZ-ICRS and ICRS, but ZZ-ICRS requires one more register. $\text{ICRS}_\text{n}$ indicates the ICRS variant for sparse matrices without empty rows. For general matrices, the normal ICRS variant must be used, of which the complexity depends on the number of nonempty rows $\tilde{m}$ instead of the full number of rows $m$.

values of the input matrix, so that the index array of the Triplet scheme can be compressed; due to the CRS ordering it is known that a given range of nonzeroes resides on the same row, so it makes no sense to repeat those values in memory. Thus the $V$ and $J$ arrays are equal to those of the Triplet scheme, while $I$ only stores the ranges of the $m$ rows such that the nonzeroes in $[V_{I_i}, V_{I_{i+1}})$ are on the $i$th row of the input matrix. Thus $I$ contains the starting index of each row in the arrays $V$ and $J$, with additionally $I_m$ set to $nz$. The storage requirement thus is $\mathcal{O}(2nz + m + 1)$. Using only the operations defined earlier, the SpMV multiplication kernel using CRS is the following.

```
(pV = V; Vend = V + nz; pI = I; pJ = J;)
 k = V + *pI++; py = y;)
while( pV < Vend ) {
     while( pV < k ) {
          px  =  x + *pJ++;
         *py += *pV++ * *px;
     }
     k = V + *pI++;
     py++;
}
```

Note that the conditional of the while-loop is checked once for every nonzero, but also an additional time when an actual row change occurs; this is necessary since rows may be empty. Otherwise, counting the number of operations is straightforward and the complexity is given in Table A.1. The number of register variables required is 9, one more than for the Triplet scheme due to the row-start pointer $k$.

### A.1.3 The ICRS data structure

The ICRS data structure [Kos02] further optimises the CRS structure by eliminating the need for add/assigns on vector pointers; i.e., instead of letting $J$ explicitly control which element of $x$ is addressed, $J$ now stores the differences in column indices of successive nonzeroes, so that the pointer towards $x$ can be explicitly controlled during multiplication. As such, the vector pointer update needs only an increment, instead of adding the correct value to the vector start position, and assigning that to the vector pointer; on some architectures, this alone results in more efficient code. The control of row transitions also improves: instead of keeping an explicit pointer $k$, overflows on $xp$ are used to indicate row changes, after which $yp$ is changed based on consecutive row differences. This was observed to result in kernels with lower instruction overhead (see also Koster [Kos02, Figure 2.5]):

```
(pV = V; Vend = V + nz; pI = I; pJ = j;
 py = y; px = x + *pJ++; xend = x + n;)
while( pV < Vend ) {
     py += *pI++;
     while( px < xend ) {
          *py += *pV++ * *px;
           px += *pJ++;
     }
     px -= n;
}
```

However, the main advantage for ICRS is that the kernel only incurs operation cost for nonempty matrix rows; empty ones are efficiently skipped. With $\tilde{m}$ the number of nonempty rows, the operation counts for ICRS are included in Table A.1. Although instruction overhead may be decreased, the total number of operations remains very close to that of plain CRS. When there are no empty rows, no adaptive row jump is required and $py$ can be simply incremented, making the array $I$ obsolete. This 'nonempty' variant of ICRS, $\text{ICRS}_\text{n}$ reduces algorithm complexity and is also found in Table A.1. The required number of registers for ICRS is 8, and for $\text{ICRS}_\text{n}$ it is 7.

### A.1.4 Zig-zag variants of (Incremental) CRS

Zig-zag CRS (ZZ-CRS) only adapts the nonzero ordering of sparse matrices. The original CRS ordering is row-major, ordering the rows from low to high first, and the columns ordered from low to high second. ZZ-CRS remains row-major, first ordering rows from low to high, but differs in that on even rows columns are ordered from low to high, while on odd rows columns are ordered from high to low. For ZZ-CRS, this requires only a change in data structure construction; the memory requirements and the computational kernel remain exactly equal to those of plain CRS.

For ZZ-ICRS, construction also differs from normal ICRS, with no changes in memory requirements as well. The computational SpMV multiplication code does

change, but note that the computational operation count does not differ from that of ICRS. The number of required registers does increase by one, however.

```
(pV = V; Vend = V + nz; py = y;
 px = x + *J++; xend = x + n;)
while( pV < Vend ) {
     py += *I++;
     while( px < xend ) {
          *py += *pV++ * *px;
           px += *J++;
     }
     px -= n;
     if( pV >= Vend ) break;
     py += *I++;
     while( x <= px ) {
          *py += *pV++ * *px;
           px -= *J++;
     }
     px += n;
}
```

### A.1.5 The BICRS data structure

The BICRS data structure [YB11a] is a very straightforward adaptation of ICRS. First, the nonzero order is no longer assumed row-major; any arbitrary ordering can be followed. This is enabled by allowing the row and column increments to be negative, so that bi-directional traversal of the sparse matrix during multiplication is possible. By doing this, the number of row jumps $j$ may increase from $\tilde{m}$, but it still remains bounded by the total number of nonzeroes $nz$. The actual SpMV multiplication kernel remains the same as that of (normal) ICRS, but with $j$ substituted for $\tilde{m}$. This has been included in Table A.1.

On matrices in doubly separated block diagonal (DSBD) form, and with using sparse blocking as described in Chapter 2, BICRS performance worsens as the number of blocks increases. In theory, this poses a limit to how far $p$ can be taken. The performance gains of sparse blocking should therefore overtake the penalty for increasing $p$, or other data structures will become more efficient. In terms of memory usage, BICRS performs better than the Triplet scheme in all but the worst case; while on the other hand, the Triplet scheme employs a much simpler multiplication kernel.

### A.1.6 The simple block data structure

The simple block data structure assumes a matrix is divided in submatrices, and stores all those submatrices in multiple other data structures, which may be of a mixed type. Thus performance is a sum of the data structures used on the submatrices, and in

this section the concatenation of data structures for the various DSBD-form blocks is considered. The most basic case is when plain CRS and CCS are used for the various blocks, and this is considered first. Second, using the Incremental variants (ICRS/ICCS) is analysed. The case of BICRS already has been discussed briefly in the previous subsection. In the considerations, a full 2D SBD form is assumed, as would be obtained by using reordering with $p = 2^k$ based on the fine-grain scheme. On the block matrices on the diagonal the row-major structures are assumed, as is the case for the vertically-oriented separators. The horizontally-oriented separators use column-major structures. See Chapter 2 for details.

### Using CRS/CCS

In this case, the storage requirement is independent of the matrix structure. Ideally, all separator blocks are empty and the remaining $p$ blocks on the diagonal are of the size $(m/p) \times (n/p)$; this corresponds to perfect partitioning and perfect load-balance. Then, applying CRS on each block takes $\mathcal{O}(\text{nz} + m)$ storage, which is optimal; but this ideal case is highly unlikely to occur.

Assuming separator blocks are not empty, the number of centre separator blocks is $p - 1$, and the number of pure blocks is $p$. Furthermore, there are $2(p - 1)$ vertically oriented separator blocks, and $2(p - 1)$ horizontally oriented separator blocks. By switching between CRS and CCS, the total storage is of

$$\mathcal{O}(2\text{nz} + c_1 m + c_2 n), \tag{A.1}$$

where the $c_i$ depend on $p$, and are derived as follows.

Let the row-wise length of the $i$th horizontal separator block (numbering from top to bottom of the DSBD matrix) be denoted by $\tilde{y}_i$. Note that $\tilde{y}_i$ also is the row-wise length of the neighbouring centre block and neighbouring horizontally-oriented separator block; i.e., there are 3 separator blocks of the same row-wise length. Similarly, $\tilde{x}_j$ can be defined for the column-wise lengths. The row-wise length $y_i$ is the row-wise length of the $i$th pure block (again numbered from top to bottom), and similarly for $x_j$. Then, the total length of the $p$ pure blocks and the $p - 1$ centre blocks exactly equals $m$:

$$\left[\sum_{i=0}^{p-1} y_i\right] + \left[\sum_{i=0}^{p-2} \tilde{y}_i\right] = m. \tag{A.2}$$

The remaining blocks using CRS are the vertically oriented blocks. For $p = 2$, this length is $m - \tilde{y}_0$; the total length minus the length of the horizontal separators. For $p = 4$, the two recursively created DSBD structures add similarly constructed values: $y_0 + y_1$ for the top-left substructure, and $y_2 + y_3$ for the bottom-right substructure. Summing these two contributions, this becomes $m - \tilde{y}_0 - \tilde{y}_1 - \tilde{y}_2$, In the DSBD picture for $p = 4$ (see, e.g., Figure 2.1, right) this can be visually made clear by placing the two substructures above each other, and removing the row-wise separator intervals; the remainder is exactly the combined length of the vertically aligned

separator blocks. Alternatively, the sum can be rewritten using Equation (A.2) to obtain the same result. In any case, summing the results for $p = 2$ and $p = 4$ brings the total row-wise length of vertically aligned separators to $m - \tilde{y}_0 + m - \tilde{y}_0 - \tilde{y}_1 - \tilde{y}_2$, which equals

$$2m - 2\tilde{y}_0 - \tilde{y}_1 - \tilde{y}_2.$$

Taking this sum in full recursion for arbitrary $p = 2^k$, and when ordering the $\tilde{y}_i$ on value so that $\tilde{y}_0$ always corresponds to the separator with the largest length, $\tilde{y}_1$ to the one with second-largest length, and so on, the following formula for the total row-wise length is obtained:

$$\sum_{j=0}^{k-1} \left[ m - \sum_{i=0}^{2^{j+1}-2} \tilde{y}_i \right]. \tag{A.3}$$

As larger separator values decrease the total row-wise lengths of the vertical separators, and thus decrease the memory usage of the simple block data structure, the $\tilde{y}_i$ can be bounded by $\min_i \tilde{y}_i$ so Equation (A.3) can be bounded from above by:

$$mk - \min_i \tilde{y}_i (2p - k - 2). \tag{A.4}$$

With Equation (A.2) and (A.4), $c_1$ in Equation (A.1) becomes $m + m \log p - p \min_i \tilde{y}_i$.

Now only the total column-wise length of the vertically-aligned separator blocks, which counts towards the memory usage of CCS sub data-structures, remains to be counted. This length can be derived in exactly the same manner as the total length of the horizontally-aligned separator blocks, in fact adapting Equation (A.4) to $n(k - 1) - 2 \min_j \tilde{x}_j (p - k)$. This means $c_2$ in Equation (A.1) equals $n \log p - p \min_j \tilde{x}_j$. The total memory storage for the simple block-based data structure backed by CRS and CCS thus is:

$$\mathcal{O}(2nz + m + (m + n) \log_2 p - (\min_i \tilde{y}_i + \min_j \tilde{x}_j) p),$$

which means that the extra overhead compared to (I)CRS scales logarithmically in $p$. The effect of the minimum separator size is quite limited; in the worst case, $\min_i \tilde{y}_i = m/p$ and $\min_j \tilde{x}_j = n/p$ (otherwise there is a smaller separator), so $(\min_i \tilde{y}_i + \min_j \tilde{x}_j)p$ is bounded by $(m + n)$ and thus is asymptotically unimportant[2].

### Using ICRS/ICCS

In this case, much depends on the exact nonzero structure of the DSBD matrix. In the worst case, however, each nonzero causes a row or column jump within its block, thus causing the simple block-based data structure, like the BICRS scheme, to be bounded

[2] Also note that, perhaps ironically, $\min_i \tilde{y}_i \ll m$ and $\min_j \tilde{x}_j \ll n$ if partitioning was successful; which means that the better the partitioning, the worse memory usage becomes. This increase is only slight however, and with the exception of perfect partitioning.

by the Triplet scheme. In the best case, assuming separator blocks are nonempty, each separator block contains nonzeroes on one row only. Then, each instance of $I$ at all sub data-structures stores exactly one element, and the problem reduces to counting the number of separator blocks, which is $5p - 5$, as deduced at the start of this subsection. Using these best- and worst-case bounds, memory usage for the simple block data structures with ICRS/ICCS scales as

$$\mathcal{O}(2\text{nz} + \min\{\tilde{m} + 5p, \text{nz}\})$$

which performs well if $p$ is small compared to the number of rows, which is true in practice. If in future work $p$ can be taken much larger to enable cache-oblivious multiplication, then at worst $p$ will be of the order of $m + n$, and still much better scalability compared to the non-incremental version is obtained. Also, with such detailed partitioning it is expected that many of the separator blocks will become empty, in contrast to the larger separators encountered in this thesis; this bound on storage space thus is expected to be too pessimistic for large $p$. Still, a block-based data structure which does not scale in $p$ at all and remains competitive with plain (I)CRS, still remains something to strive for.

# Bibliography

[APÇ04] C. Aykanat, A. Pinar, and Ü. V. Çatalyürek, *Permuting sparse rectangular matrices into block-diagonal form*, SIAM J. Sci. Comput. **25** (2004), no. 6, 1860–1879.

[BBF+07] M. A. Bender, G. S. Brodal, R. Fagerberg, R. Jacob, and E. Vicari, *Optimal sparse matrix dense vector multiplication in the I/O-model*, Proceedings 19th Annual ACM Symposium on Parallel Algorithms and Architectures, ACM Press, New York, 2007, pp. 61–70.

[BDD+00] 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, PA, 2000.

[BFAY+11] Rob H. Bisseling, Bas O. Fagginger Auer, A. N. Yzelman, Tristan van Leeuwen, and Umit Çatalyürek, *Two-dimensional approaches to sparse matrix partitioning*, Combinatorial Scientific Computing (Uwe Naumann and Olaf Schenk, eds.), CRC Press, 2011, To appear.

[BFF+09] 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*, SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures (New York, NY, USA), ACM, 2009, pp. 233–244.

[Bis04] Rob H. Bisseling, *Parallel scientific computation: A structured approach using BSP and MPI*, Oxford University Press, Oxford, UK, March 2004.

[BJvOR03] Olaf Bonorden, Ben Juurlink, Ingo von Otte, and Ingo Rieping, *The Paderborn University BSP (PUB) library*, Parallel Comput. **29** (2003), no. 2, 187–207.

[BM05] Rob H. Bisseling and Wouter Meesen, *Communication balancing in parallel sparse matrix-vector multiplication*, Electronic Transactions on

Numerical Analysis **21** (2005), 47–65, Special Issue on Combinatorial Scientific Computing.

[BP98] S. Brin and L. Page, *The anatomy of a large-scale hypertextual web search engine*, Comput. Netw. ISDN Systems, vol. 30, 1998, pp. 107–117.

[ÇA99] Ü. V. Çatalyürek and C. Aykanat, *Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication*, IEEE Trans. Parallel Distrib. Systems **10** (1999), no. 7, 673–693.

[ÇA01] ———, *A fine-grain hypergraph model for 2D decomposition of sparse matrices*, Proceedings 8th International Workshop on Solving Irregularly Structured Problems in Parallel, IEEE Press, Los Alamitos, CA, 2001, p. 118.

[CD05] Albert Chan and Frank Dehne, *CGMgraph/CGMlib: Implementing and testing CGM graph algorithms on PC clusters and shared memory machines*, International Journal of High Performance Computing Applications **19** (2005), 81–97.

[CJP07] Barbara Chapman, Gabriele Jost, and Ruud van der Pas, *Using OpenMP: Portable shared memory parallel programming*, Scientific and Engineering Computation, The MIT Press, Cambridge, MA, 2007.

[Dav11] T. A. Davis, *University of Florida sparse matrix collection*, http://www.cise.ufl.edu/research/sparse/matrices, University of Florida, Department of Computer and Information Science and Engineering, Gainesville, FL, 1994-2011.

[DBH+06] K. D. Devine, E. G. Boman, R.T. Heaphy, R. H. Bisseling, and U. V. Catalyurek, *Parallel hypergraph partitioning for scientific computing*, Proceedings IEEE International Parallel and Distributed Processing Symposium 2006, IEEE Press, Long Beach, CA, 2006.

[DER86] 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.

[DFRC96] Frank Dehne, Andreas Fabri, and Andrew Rau-Chaplin, *Scalable parallel computational geometry for coarse grained multicomputers*, International Journal on Computational Geometry and Applications **6** (1996), 379–400.

[DJ07] J. M. Dennis and E. R. Jessup, *Applying automated memory analysis to improve iterative algorithms*, SIAM J. Sci. Comput. **29** (2007), no. 5, 2210–2223.

[DM98] L. Dagum and R. Menon, *OpenMP: an industry standard API for shared-memory programming*, Computational Science and Engineering **5** (1998), no. 1, 46–55.

[DMS+94] 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), no. 3, 489–496.

[FJ98] M. Frigo and S. G. Johnson, *FFTW: An adaptive software architecture for the FFT*, Proceedings IEEE International Conference on Acoustics, Speech, and Signal Processing, vol. 3, IEEE Press, Los Alamitos, CA, 1998, pp. 1381–1384.

[FJ05] ———, *The design and implementation of FFTW3*, Proceedings of the IEEE **93** (2005), no. 2, 216–231.

[FLPR99] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, *Cache-oblivious algorithms*, Proceedings 40th Annual Symposium on Foundations of Computer Science, IEEE Press, Los Alamitos, CA, 1999, p. 285.

[GBDD10] L. Grigori, E. Boman, S. Donfack, and T. A. Davis, *Hypergraph-based unsymmetric nested dissection ordering for sparse LU factorization*, SIAM J. Sci. Comput. **32** (2010), no. 6, 3426–3446.

[Geo73] A. George, *Nested dissection of a regular finite element mesh*, SIAM J. Numer. Anal. **10** (1973), no. 2, 345–363.

[GG02] K. Goto and R. van de Geijn, *On reducing TLB misses in matrix multiplication*, Tech. Report TR-2002-55, University of Texas at Austin, Department of Computer Sciences, 2002, FLAME Working Note #9.

[GVL96] Gene H. Golub and Charles F. Van Loan, *Matrix computations*, third ed., Johns Hopkins Studies in the Mathematical Sciences, The Johns Hopkins University Press, Baltimore, MD, 1996.

[Hin03] Konrad Hinsen, *High-level parallel software development with Python and BSP*, Parallel Processing Letters **13** (2003), 473–484.

[HLP07] 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** (2007), no. 4, 213–220.

[HMB00] 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), 1631–1647.

[HMS+98] Jonathan M. D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas, and Rob H. Bisseling, *BSPlib: The BSP programming library*, Parallel Computing **24** (1998), no. 14, 1947–1980.

[HR98] B. Hendrickson and E. Rothberg, *Improving the run time and quality of nested dissection ordering*, SIAM J. Sci. Comput. **20** (1998), no. 2, 468–489.

[HS52] M. R. Hestenes and E. Stiefel, *Methods of conjugate gradients for solving linear systems*, Journal of Research of the National Bureau of Standards **49** (1952), 409–436.

[Im00] Eun-Jin Im, *Optimizing the performance of sparse matrix-vector multiplication*, Ph.D. thesis, University of California at Berkeley, 2000.

[IY01] E.-J. Im and K. A. Yelick, *Optimizing sparse matrix–vector multiplication for register reuse in SPARSITY*, Proceedings International Conference on Computational Science, Part I, Lecture Notes in Computer Science, vol. 2073, 2001, pp. 127–136.

[Kam07] Alan Kaminsky, *Parallel Java: A unified API for shared memory and cluster parallel programming in 100% Java*, International Parallel and Distributed Processing Symposium, IEEE Press, Long Beach, CA, 2007, pp. 1–8.

[KK98] G. Karypis and V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM J. Sci. Comput. **20** (1998), no. 1, 359–392.

[KK99] _____, *Multilevel k-way hypergraph partitioning*, Proceedings 36th ACM/IEEE Conference on Design Automation, ACM Press, New York, 1999, pp. 343–348.

[KKHY06] 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), no. 1, 92–112.

[Kle99] J. M. Kleinberg, *Authoritative sources in a hyperlinked environment*, J. ACM **46** (1999), no. 5, 604–632.

[KMRS88] Anna Karlin, Mark Manasse, Larry Rudolph, and Daniel Sleator, *Competitive snoopy caching*, Algorithmica **3** (1988), 79–119, 10.1007/BF01762111.

[Kos02] Joris Koster, *Parallel templates for numerical linear algebra, a high-performance computation library*, Master's thesis, Utrecht University, Department of Mathematics, July 2002.

[KUWK00] M. Kowarschik, U. Rüde, C. Weiß, and W. Karl, *Cache-aware multigrid methods for solving Poisson's equation in two dimensions*, Computing **64** (2000), no. 4, 381–399.

[Lec94] David Lecomber, *An object-oriented programming model for BSP computations*, Proc. PPECC Workshop on Parallel and Distributed Computing, 1994.

[Len90] T. Lengauer, *Combinatorial algorithms for integrated circuit layout*, John Wiley and Sons, Chichester, UK, 1990.

[LM06] A. N. Langville and C. D. Meyer, *Google's PageRank and beyond: The science of search engine rankings*, Princeton University Press, Princeton, NJ, 2006.

[LW07] K. Patrick Lorton and David S. Wise, *Analyzing block locality in Morton-order and Morton-hybrid matrices*, SIGARCH Comput. Archit. News **35** (2007), no. 4, 6–12.

[MFT+10] Michele Martone, Salvatore Filippone, Salvatore Tucci, Marcin Paprzycki, and Maria Ganzha, *Utilizing recursive storage in sparse matrix-vector multiplication - preliminary considerations*, Proceedings of the ISCA 25th International Conference on Computers and Their Applications (CATA) (Hawaii, USA) (Thomas Philip, ed.), ISCA, 2010, pp. 300–305.

[Mor66] G. Morton, *A computer oriented geodetic data base and a new technique in file sequencing*, Tech. report, IBM, Ottawa, Canada, March 1966.

[NS11] Uwe Naumann and Olaf Schenk, *Combinatorial scientific computing*, CRC Press, 2011, To appear.

[NVDY07] 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), no. 3, 297–311.

[PH99] A. Pinar and M. T. Heath, *Improving performance of sparse matrix-vector multiplication*, Proceedings Supercomputing 1999, ACM Press, New York, 1999, p. 30.

[PS82] C. C. Paige and M. A. Saunders, *LSQR: An algorithm for sparse linear equations and sparse least squares*, ACM Transactions on Mathematical Software **8** (1982), 43–71.

[PTL85] B. N. Parlett, D. Taylor, and Z. Liu, *A look-ahead Lanczos algorithm for unsymmetric matrices*, Mathematics of Computation **44** (1985), 105–124.

[SCFK04] M. M. Strout, L. Carter, J. Ferrante, and B. Kreaseck, *Sparse tiling for stationary iterative methods*, Int. J. High Perf. Comput. Appl. **18** (2004), no. 1, 95–113.

[SF93] G. L. G. Sleijpen and D. R. Fokkema, *BiCGSTAB($\ell$) for linear equations involving unsymmetric matrices with complex spectrum*, Electronic Transactions on Numerical Analysis **1** (1993), 11–32.

[SH04] M. M. Strout and P. D. Hovland, *Metrics and models for reordering transformations*, Proceedings 2004 Workshop on Memory System Performance, ACM Press, New York, 2004, pp. 23–34.

[SS86] Y. Saad and M. Schultz, *GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems*, SIAM Journal on Scientific and Statistical Computation **7** (1986), 856–869.

[SvdV00] Gerard L. G. Sleijpen and Henk A. van der Vorst, *A Jacobi-Davidson iteration method for linear eigenvalue problems*, SIAM Review **42** (2000), no. 2, 267–293.

[SvG08] 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** (2008), no. 2, 1035–1062.

[Tis98] Alexandre Tiskin, *The bulk-synchronous parallel random access machine*, Theoretical Computer Science **196** (1998), no. 1-2, 109 – 130.

[TK04] A. Trifunovic and W. J. Knottenbelt, *A parallel algorithm for multilevel $k$-way hypergraph partitioning*, Proceedings 3rd International Symposium on Parallel and Distributed Computing, IEEE Press, Los Alamitos, CA, 2004, pp. 114–121.

[Tol97] S. Toledo, *Improving the memory-system performance of sparse-matrix vector multiplication*, IBM J. Res. Dev. **41** (1997), no. 6, 711–725.

[UA04] Bora Uçar and Cevdet Aykanat, *Encapsulating multiple communication cost metrics in partitioning sparse rectangular matrices for parallel matrix vector multiplies*, SIAM Journal on Scientific Computing **25** (2004), no. 6, 1837–1859.

[Val90] Leslie G. Valiant, *A bridging model for parallel computation*, Commun. ACM **33** (1990), no. 8, 103–111.

[Val08] _____, *A bridging model for multi-core computing*, Algorithms - ESA 2008, Lecture Notes in Computer Science, vol. 5193, Springer, Berlin, 2008, pp. 13–28.

[VB05] B. Vastenhouw and R. H. Bisseling, *A two-dimensional data distribution method for parallel sparse matrix-vector multiplication*, SIAM Rev. **47** (2005), no. 1, 67–95.

[vdS10] Aad J. van der Steen, *Overview of recent supercomputers 2010*, Tech. report, Netherlands National Computing Facilities Foundation, The Hague, the Netherlands, September 2010.

[vdV92] 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** (1992), 631–644.

[VDY05] R. Vuduc, J. W. Demmel, and K. A. Yelick, *OSKI: A library of automatically tuned sparse matrix kernels*, J. Phys. Conf. Series **16** (2005), 521–530.

[vHBB02] A. van Heukelum, G. T. Barkema, and Rob H. Bisseling, *DNA electrophoresis studied with the cage model*, J. Comput. Phys. **180** (2002), no. 1, 313–326.

[VM05] R. W. Vuduc and H.-J. Moon, *Fast sparse matrix-vector multiplication by exploiting variable block structure*, High Performance Computing and Communications 2005, Lecture Notes in Computer Science, vol. 3726, 2005, pp. 807–816.

[VS02] V. Valsalam and A. Skjellum, *A framework for high-performance matrix multiplication based on hierarchical abstractions, algorithms and optimized low-level kernels*, Concurrency and Computation: Practice and Experience **14** (2002), 805–839.

[Wor] CPU World, *Microprocessors/Central Processing Units*, retrieved on 27th of April, 2011.

[WOV+09] 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** (2009), no. 3, 178–194.

[WP05] R. C. Whaley and A. Petitet, *Minimizing development and maintenance costs in supporting persistently optimized BLAS*, Software: Practice and Experience **35** (2005), no. 2, 101–121.

[WPD01] R. C. Whaley, A. Petitet, and J. J. Dongarra, *Automated empirical optimizations of software and the ATLAS project*, Parallel Comput. **27** (2001), no. 1–2, 3–35.

[WS97] J. B. White, III and P. Sadayappan, *On improving the performance of sparse matrix-vector multiplication*, Proceedings 4th International Conference on High-Performance Computing, IEEE Press, Los Alamitos, CA, 1997, pp. 66–71.

[YB09] 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** (2009), no. 4, 3128–3154.

[YB11a] _____, *A cache-oblivious sparse matrix–vector multiplication scheme based on the Hilbert curve*, Progress in Industrial Mathematics at ECMI, Springer, Berlin, 2011, To appear.

[YB11b] _____, *An object-oriented BSP library for multicore programming*, Concurrency and Computation: Practice and Experience (2011), To appear.

[YB11c] _____, *Two-dimensional cache-oblivious sparse matrix–vector multiplication*, Parallel Computing (2011), To appear.

[YSP+98] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken, *Titanium: a high-performance Java dialect*, Concurrency: Practice and Experience **10** (1998), no. 11-13, 825–836.

# List of Figures

# List of Tables

# List of Algorithms

# List of Notes

# Samenvatting

***een uiteenzetting voor niet-experts***

Computers bewerken gegevens, maar voor computers maakt het nogal wat uit in welke volgorde deze gegevens behandeld worden. In dit proefschrift zijn bestaande technieken, oorspronkelijk bedoeld voor het verdelen van gegevens, aangepast en uitgebreid zodat zij het herordenen van data mogelijk maken. Dit herordenen zorgt ervoor dat de nieuwe volgorde van gegevens de computerberekeningen veel sneller maakt.

De oorzaak dat ongestructureerde gegevens het rekenen vertragen ligt bij de architectuur van de computer. Berekeningen op computers worden gedaan door een processor, terwijl de gegevens die een computer nodig heeft typisch niet direct op die processor aanwezig zijn; deze komen van de harde schijf, van compact discs, of van het internet. Een stukje gegeven dat nodig is om een berekening uit te voeren moet eerst dus een behoorlijke weg naar de processor afleggen. De infrastructuur die gegevens moeten gebruiken om bij de processor uit te komen, is de oorzaak dat voor ongestructureerde gegevens de berekeningen langzamer lopen.

We nemen voor het gemak aan dat alle nodige gegevens in het grote interne geheugen (het zogenaamde *Random Access Memory*, RAM) van een computer aanwezig zijn. Tussen de processor van een computer en het interne geheugen, zijn meerdere steeds kleinere (maar snellere) stukjes geheugen te vinden; deze worden ‘caches’ genoemd. Caches worden gebruikt om gegevens die het meest recent nodig waren dichtbij de processor te houden. Voor gestructureerde problemen levert dit een snelheidswinst op: als recente gegevens weer nodig zijn kunnen die uit het snelle geheugen worden gehaald, in plaats van uit het langzamere grotere geheugen. Echter zijn lang niet alle interessante problemen gestructureerd, en als er te vaak gerefereerd wordt aan het langzame geheugen, is de processor langer bezig met wachten op gegevens dan met het daadwerkelijk verwerken van die gegevens.

## Deel I

De oplossing welk wordt gepresenteerd in de eerste twee hoofdstukken, begint met het analyseren van afhankelijkheid tussen gegevens. Vervolgens worden gegevens in twee groepen ingedeeld, zodanig dat de groepen zo min mogelijk van elkaar afhanke-

lijk zijn. Elk van die twee stukken kan vervolgens recursief op dezelfde manier verder gesplitst worden; het origineel wordt in twee stukken verdeeld, die elk wederom gesplitst worden zodat er vier stukken ontstaan, daarna nogmaals voor acht stukken, enzovoorts. Hierop gebaseerd kunnen gegevens geordend worden, zodanig dat data in kleinere groepen worden behandeld en dus beter in de kleinere caches passen.

## De gegevens en de berekening

Dit proefschrift behandelt bovenstaande in de context van de *ijle matrix–vector vermenigvuldiging*; dat wil zeggen, we nemen aan dat de gegevens in de vorm van een matrix gegoten kunnen worden. Een matrix $A$ van dimensie $m \times n$ bevat $m$ maal $n$ stukjes gegevens, en wordt in het algemeen als een blok weergegeven. Het volgende is een $2 \times 3$ voorbeeldmatrix, met als type gegevens gewoonweg getallen:

$$A = \begin{pmatrix} 1 & \sqrt{2} & 0 \\ \sqrt{2} & 0 & 1 \end{pmatrix}.$$

Één getal uit de matrix wordt genoteerd door middel van haar coördinaten: zo is $A_{12}$ gelijk aan $\sqrt{2}$ en is $A_{23}$ gelijk aan 1. Een matrix waarvan één van de dimensies 1 is, wordt ook wel een vector genoemd. Een $n \times 1$ vector is bijvoorbeeld:

$$x = \begin{pmatrix} 0 \\ \sqrt{2} \\ 1 \end{pmatrix}.$$

Nergens in het proefschrift wordt overigens vereist dat de gegevens getallen *moeten* zijn, in principe zouden gegevens van elke soort behandeld kunnen worden. Wat wel wordt vereist, is dat sommatie ($+$) en vermenigvuldiging ($\cdot$) tussen twee stukjes gegevens mogelijk is, en dat er een nul-element bestaat wat niets doet als het ergens bij wordt opgeteld, en resulteert in het nul-element zelf wanneer het betrokken is bij een vermenigvuldiging (natuurlijk is dit vanzelfsprekend voor normale getallen, maar misschien niet voor andere soorten gegevens).

Sommatie en vermenigvuldiging zijn nodig om de *matrix–vector vermenigvuldiging* uit te voeren: voor een matrix $A$ en een vector $x$ als hierboven, is $A$ maal $x$ (simpelweg genoteerd als $Ax$) gelijk aan een vector $y$ van dimensie $m \times 1$, met het $i$-de getal van $y$ gelijk aan de sommatie van elk getal uit de $i$-de rij van $A$ vermenigvuldigd met het bijbehorende getal uit $x$. In wiskundige notatie is dit $y_i = \sum_{j=1}^{n} A_{ij} x_j$, en met het voorbeeld van hierboven krijgen we:

$$y = \begin{pmatrix} 1 & \sqrt{2} & 0 \\ \sqrt{2} & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ \sqrt{2} \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \cdot 0 + \sqrt{2} \cdot \sqrt{2} + 0 \cdot 1 \\ \sqrt{2} \cdot 0 + 0 \cdot \sqrt{2} + 1 \cdot 1 \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \end{pmatrix}.$$

Merk op dat $x$ en $y$ niet van dezelfde grootte zijn als $A$ niet vierkant is.

Voor de bovenstaande matrix–vector vermenigvuldiging waren er $6$ normale vermenigvuldigingen en $4$ sommaties nodig. Echter zagen we dat er relatief vaak vermenigvuldigd werd met het getal $0$, terwijl de uitkomst daarvan natuurlijk altijd $0$ is. Kijken we dieper dan zien we dat een $0$ in $A$ tot gevolg heeft dat het corresponderende element uit $x$ niet wordt bekeken. Aan de andere kant heeft een $0$ in de vector $x$ tot gevolg dat een complete kolom van $A$ wordt genegeerd. Simpelweg verwijderen van nullen uit $x$, tezamen met de bijbehorende kolommen uit $A$, levert dus een directe snelheidswinst.

Het verkrijgen van snelheidswinst uit nullen in $A$ zelf is wat lastiger, en kan pas worden gedaan wanneer de matrix overwegend uit nullen bestaat, ofwel, wanneer de matrix *ijl* is. Dit is precies de situatie waarvoor in dit proefschrift naar een versnelling wordt gezocht, en verklaart de eerste helft van de titel van het proefschrift: "*Snelle ijle matrix–vectorvermenigvuldiging...*".

## Een wiskundige representatie

Alvorens we overgaan tot uitleg van het tweede deel van de titel, is het eerst noodzakelijk te beschrijven hoe de structuur van niet-nullen kan worden gevangen in een wiskundig model. Hiervoor worden zogenaamde *hypergrafen* gebruikt, wat een generalisatie is van een normale graaf. Een graaf is een simpel concept, en bestaat alleen uit twee ingrediënten: punten en lijnen, waar lijnen twee bepaalde punten met elkaar verbinden. Het treinnetwerk is een simpel voorbeeld van een graaf; stations kunnen worden weergegeven door punten en sporen tussen verschillende stations door lijnen. Het hoeft dus zeker niet zo te zijn dat elk punt met elk ander punt is verbonden door een lijn; het hoeft zelfs niet zo te zijn dat alle punten indirect met elkaar verbonden zijn (het spoornetwerk van Nederland en van de Verenigde Staten zijn bijvoorbeeld niet verbonden).

Tussen grafen en matrices bestaat al een verband. Stel bijvoorbeeld dat het Nederlandse spoornetwerk bestaat uit vier stations: Utrecht, Groningen, Arnhem en Rotterdam. Qua spoor nemen we aan dat Utrecht verbonden is met alle andere stations, en dat er voor de rest een directe lijn is van Arnhem naar Groningen. Deze graaf is weer te geven als een $4 \times 4$ matrix met langs elke as de vier stations, en een $1$ in de matrix als er een verbinding is tussen de corresponderende stations:

| | U | G | A | R |
|---|---|---|---|---|
| Utrecht | - | 1 | 1 | 1 |
| Groningen | 1 | - | 1 | 0 |
| Arnhem | 1 | 1 | - | 0 |
| Rotterdam | 1 | 0 | 0 | - |

,

waar we de diagonaal even laten voor wat het is. In het geval van een veel groter aantal stations, is het te verwachten dat de bijbehorende matrix veel nullen gaat bevatten; ofwel, de matrix is ijl. Maar deze representatie stelt nog een andere eis aan de matrix: hij moet symmetrisch zijn (de linkeronderhoek is gespiegeld gelijk aan de

rechterbovenhoek, met de streepjes op de diagonaal als scheiding). Natuurlijk hebben we liever dat alle ijle matrices goed gerepresenteerd kunnen worden, en daarom gaan we nu kijken naar de hypergraaf.

Net als een normale graaf, bevat een hypergraaf punten. De 'lijnen' in een hypergraaf verbinden echter niet twee punten, maar een arbitraire hoeveelheid punten; zo een 'hyperlijn' wordt ook wel *net* genoemd omdat het meerdere punten vangt. Als we weer als voorbeeld steden als punten in een hypergraaf nemen, dan kunnen provincies bijvoorbeeld de netten voorstellen. Een ander voorbeeld is dat we alle steden met een treinstation in een net groeperen, en evenzo voor steden aan de snelweg, bij vliegvelden, of bereikbaar per bus. Dit laatste voorbeeld geeft aan dat punten in een graaf in meerdere netten kunnen voorkomen, naast dat netten meerdere steden kunnen bevatten. Het geval dat netten onderling nooit dezelfde punten bevatten, zoals het geval met steden en provincies, is juist een erg speciaal geval.

Het verband tussen hypergrafen en matrices kan op verschillende manieren worden gebouwd. De punten in een hypergraaf kunnen gegeven worden door de kolommen van een ijle matrix, en de netten corresponderen dan met de rijen van dezelfde matrix: het net behorende bij rij nummer $i$ verbindt dan de kolommen waar op die rij niet-nullen te vinden zijn: zie bijvoorbeeld Figuur 8 op pagina xxi. Natuurlijk is het ook mogelijk de rollen van rijen en kolommen om te draaien om zo een andere vertaling mogelijk te maken. Een hele andere manier is echter om elke niet-nul in de matrix tot een punt in de hypergraaf te benoemen. Vervolgens maken we twee verschillende groepen netten, ongeveer zoals we hierboven verschillende netten hadden voor steden met vliegvelden en steden met treinstations: maar nu voor niet-nullen in een bepaalde rij en niet-nullen in een bepaalde kolom. Dus als $A_{ij}$ een niet-nul is in rij $i$ en kolom $j$, dan zit deze niet-nul in de netten behorende bij rij $i$ en kolom $j$. Zie Figuur 9 voor een voorbeeld.

## Partitionering van hypergrafen

Nu op naar het tweede deel van de titel: "*...door partitionering en herordening*". Gegeven het voorgaande kunnen we elke ijle matrix omzetten naar een hypergraaf. Stel dat we de verzameling punten in de hypergraaf opschrijven als $\mathcal{V}$ en de verzameling van alle netten als $\mathcal{N}$. Dan wil *partitionering* in onze context zeggen dat $\mathcal{V}$ in stukjes wordt opgedeeld. Natuurlijk kan dit op veel verschillende manieren en moeten we wat eisen opleggen aan de partitionering om er iets van betekenis uit te krijgen. Daar de punten in $\mathcal{V}$ corresponderen met niet-nullen, en dus eigenlijk met werk voor de computer, is de eerste eis standaard dat de stukjes waarin $\mathcal{V}$ wordt opgehakt ongeveer even groot moeten zijn. Origineel wordt partitionering namelijk gebruikt om parallel rekenen mogelijk te maken; dat wil zeggen, het probleem wordt opgedeeld in ongeveer gelijke stukjes die zo min mogelijk met elkaar te maken hebben, zodat die stukjes zo onafhankelijk mogelijk van elkaar kunnen worden berekend op meerdere computers tegelijkertijd.

De tweede eis is dus dat de stukjes onafhankelijk moeten zijn, en hiervoor gaan we de gegevens van de verschillende netten gebruiken. Stel dat $\mathcal{V}$ in een arbitrai-

re hoeveelheid groepen is verdeeld, laten we zeggen $p$, en dat we deze groepen als $\mathcal{V}_1, \ldots, \mathcal{V}_p$ noteren. Een bepaald net $n_i$ kan nu punten uit meerdere van deze groepen bevatten; hoeveel precies noemen we de *verbindingsgraad*, en noteren we met $\lambda_i$. De som van de verbindingsgraden van alle netten geeft ons een idee van hoe goed de punten uit de verschillende groepen met elkaar verbonden zijn.

Over het algemeen maakt het ons niet uit als een net precies verbonden is met 1 groep; dat betekent namelijk dat de bijbehorende berekeningen compleet lokaal zijn aan die ene groep. Ook is het ene net meestal niet belangrijker dan enig andere, en dus gebruiken we de volgende kostenfunctie:

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

Dit wordt ook wel de $(\lambda - 1)$-metriek genoemd. De perfecte situatie correspondeert met een uitkomst 0 van bovenstaande som, dat wil zeggen, wanneer het probleem uiteenvalt in kleine stukjes die totaal niets met elkaar te maken hebben. Dit gebeurde bijvoorbeeld bij het voorbeeld van steden als punten en provincies als netten; het groeperen van punten aan de hand van provincies geeft een perfecte partitionering, als er tenminste evenveel steden zijn in elke provincie. Indien niet, is het geen eerlijke verdeling, en moeten er bij partitionering concessies gemaakt worden waardoor de kosten hoger uit zullen vallen.

## Partitionering en herordening

De nieuwe bijdragen van dit proefschrift zitten niet zozeer in technieken ter verbetering van partitionering. Er wordt ervan uitgegaan dat een manier om een goede partitionering te vinden al beschikbaar is, en bovendien aan bepaalde voorwaarden voldoet. Op basis van het resultaat van zo'n partitioneerder wordt herordening toegepast, en dat is waar dit proefschrift iets aan wil toevoegen in Hoofdstuk 1 en 2. Om de resultaten kracht bij te zetten is de herordeningsmethode wel toegevoegd aan het Utrechtse softwarepakket *'Mondriaan'*, welk voorheen alleen geschikt was voor het partitioneren van ijle matrices en hypergrafen.

De herordeningsmethode werkt door de link tussen matrices en hypergrafen verder uit te buiten. Niet alleen wordt de partitionering van de punten ($\mathcal{V}$) van de hypergraaf gebruikt, maar ook de informatie van de netten ($\mathcal{N}$) wordt betrokken bij het terugvertalen van hypergraaf naar matrix. Als we wederom aannemen dat $\mathcal{V}$ in twee groepen is verdeeld, namelijk $\mathcal{V}_L$ en $\mathcal{V}_R$, dan kunnen we $\mathcal{N}$ in drie groepen verdelen: een groep $\mathcal{N}^+$ met netten die alleen iets te maken hebben met punten in $\mathcal{V}_L$, een groep $\mathcal{N}^-$ met netten alleen in $\mathcal{V}_R$, en de groep $\mathcal{N}^c$ met overgebleven netten met verbindingsgraad 2. Dit betekent dat onze oorspronkelijke matrix $A$ na partitionering

kan worden ingedeeld als:

$$\tilde{A} = \begin{array}{c} \begin{array}{cc} \mathcal{V}_L & \mathcal{V}_R \end{array} \\ \left( \begin{array}{cc} A_1 & 0 \\ A_2 & A_3 \\ 0 & A_4 \end{array} \right) \begin{array}{c} \mathcal{N}^+ \\ \mathcal{N}^c \\ \mathcal{N}^- \end{array} \end{array} ;$$

ofwel, alle niet-nullen in $\mathcal{V}_L$ verplaatsen we naar links, die in $\mathcal{V}_R$ naar rechts, en tegelijkertijd verplaatsen we rijen bevat in $\mathcal{N}^+$ naar boven, de rijen in $\mathcal{N}^-$ naar beneden, en de resterende rijen in $\mathcal{N}^c$ laten we in het midden. De resulterende vorm van $\tilde{A}$ na herordening, ook hierboven afgebeeld, is nieuw en noemen we de *Separated Block Diagonal* (SBD) vorm, of, in goed Nederlands, de 'gescheiden blokdiagonaal'-vorm. Dit omdat de submatrices (blokken) $A_1$ en $A_4$ hierboven op een diagonaal lijken te liggen, gescheiden door een rij van matrices $A_2$ en $A_3$.

Waarom is deze vorm goed voor het probleem van ijle matrix–vector vermenigvuldiging? Dit wordt duidelijk wanneer we $y = Ax$ vervangen door $\tilde{y} = \tilde{A}\tilde{x}$, met $\tilde{x}, \tilde{y}$ op dezelfde manier geherordend als $\tilde{A}$. Herinner dat $A$ ijl is, en dat $A_1, \ldots, A_4$ dat dan ook zijn, terwijl $A$ en $\tilde{A}$ even groot zijn; dat wil zeggen, $A_1, \ldots, A_4$ zijn kleinere ijle matrices dan $A$ maar bevatten samen dezelfde informatie als de originele $A$ zelf. De vermenigvuldiging wordt:

$$\tilde{y} = \tilde{A}\tilde{x} = \begin{pmatrix} A_1 & 0 \\ A_2 & A_3 \\ 0 & A_4 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} A_1 x_1 \\ A_2 x_1 + A_3 x_2 \\ A_4 x_4 \end{pmatrix}.$$

Het eerste wat opvalt is dat de uitkomst $\tilde{y}$ uit drie delen bestaat ($A_1 x_1$, $A_2 x_1 + A_3 x_2$, en $A_4 x_2$), terwijl $\tilde{y}$ hetzelfde is als de originele $y$, maar in een andere volgorde. Ook zijn $x_1$ en $x_2$ samen even groot als $x$ (en ook gelijk, op herordening na), maar de vermenigvuldiging op zowel $x_1$ en $x_2$ wordt alleen gebruikt op het middelste deel van $\tilde{y}$; het eerste en het laatste deel gebruiken alleen $x_1$ of alleen $x_2$. Met andere woorden zijn de berekeningen nu geconcentreerd in het bovenste deel of het onderste deel van $\tilde{x}$, behalve wanneer het middelste deel $A_2 x_1 + A_3 x_2$ wordt berekend. En de hoogte van $A_2$ en $A_3$ is impliciet juist dat wat de partitioneerder minimaliseert met de $(\lambda - 1)$-metriek (zie de vorige sectie).

Als het in tweeën splitsen van berekeningen (dat wil zeggen, rekenen met $x_1$ en $x_2$) al lijkt te helpen, waarom zouden we dan stoppen met twee? Inderdaad is het mogelijk een $\tilde{x}$ te hebben die in 4 stukken is verdeeld, door simpelweg dezelfde truuk toe te passen met $A_1$ en $A_4$. In het hele herordeningsverhaal hierboven kunnen we namelijk $A$ vervangen door $A_1$, om zo een Separated Block Diagonal vorm van $A_1$ krijgen:

$$\tilde{A_1} = \begin{pmatrix} A_{1_1} & 0 \\ A_{1_2} & A_{1_3} \\ 0 & A_{1_4} \end{pmatrix},$$

en evenzo voor $A_4$. Deze manier van overnieuw toepassen heet *recursie*. Plakken we bovenstaand plaatje na recursie op $A_1$ en $A_4$ in het originele SBD-plaatje van $\tilde{A}$,

en hernoemen we de matrices met onderscheid tussen de diagonaalmatrices $A_i$ en de scheidingsmatrices $S_i$, dan krijgen we de volgende structuur:

$$\tilde{A} = \begin{pmatrix} A_1 & 0 & 0 & 0 \\ S_1 & S_2 & 0 & 0 \\ 0 & A_2 & 0 & 0 \\ S_3 & S_4 & S_5 & S_6 \\ 0 & 0 & A_3 & 0 \\ 0 & 0 & S_7 & S_8 \\ 0 & 0 & 0 & A_4 \end{pmatrix}.$$

En natuurlijk kunnen we doorgaan met recursie op de verschillende submatrices $A_i$, totdat niet meer verder kan worden gegaan; dat wil zeggen, totdat alle rijen in een scheidingsmatrix zitten, of totdat de submatrices zo klein zijn dat ze precies 1 kolom bevatten. Door op deze manier recursie en partitionering te combineren, kan de ijle matrix–vector vermenigvuldiging dus zo lokaal mogelijk worden gemaakt zodat het goed werkt op computers met caches, ongeacht de precieze grootte van die caches. In die zin zijn we ons niet bewust van de cache, of, in goed Engels, de herordeningsmethode is *cache-oblivious*.

Ook heeft het minimaliseren van de kostenfunctie $\sum_{i:n_i \in \mathcal{N}} (\lambda_i - 1)$ nog een bijkomend voordeel, naast dat het de verschillende scheidingsmatrices ($S_i$) rij-gewijs dun houdt: het zorgt ervoor dat de SBD-structuur ook wordt nagestreefd binnen de rijen van scheidingsmatrices zelf. Bovendien is er theoretisch, gegeven een bepaalde cache, een aantal recursiestappen aan te wijzen waarvoor de kostenfunctie exact aangeeft hoeveel elementen uit $x$ er maximaal van buiten de cache moeten worden gehaald. Ofwel, het is aangetoond dat de kostenfunctie inderdaad de onnodige beweging van gegevens minimaliseert, en dat terwijl het klassiek gezien de communicatie tijdens een parallelle ijle matrix–vector vermenigvuldiging minimaliseert.

In de praktijk zien herordeningen er uit zoals in de Figuren 1.5, 1.6 en 1.7 (te zien vanaf pagina 22). Figuren 1.10, 1.11 en verder laten de tijdwinst van daadwerkelijke vermenigvuldigingen zien. De hier uitgelegde methode kan worden uitgebreid zodat na herordening de zogenaamde *Doubly* SBD-vorm tevoorschijn komt. Deze ziet er na een splitsing in tweeën als volgt uit:

$$\begin{pmatrix} A_1 & S_1 & 0 \\ S_2 & S_3 & S_4 \\ 0 & S_5 & A_2 \end{pmatrix}.$$

Wederom kan er recursief gewerkt worden op $A_1$ en $A_2$. Het voordeel van deze vorm is dat dankzij introductie van een tweede dimensie (er is nu ook een verticale kolom scheidingsmatrices, in plaats van alleen een horizontale rij), de rij-hoogte van $S_{\{2,3,4\}}$ en de kolom-breedte van $S_{\{1,3,5\}}$ kleiner kunnen worden gemaakt dan wanneer er alleen één van die richtingen beschikbaar was. Om deze vorm in recursie te kunnen gebruiken, is er echter meer dan alleen herordening nodig; theoretisch is de vraag in welke volgorde de submatrices en scheidingsmatrices doorlopen moeten worden, en

praktisch is de vraag hoe de matrix opgeslagen moet worden in het computergeheugen. Dit alles is het domein van Hoofdstuk 2, en de resultaten verbeteren inderdaad die van de oorspronkelijke methode uit Hoofdstuk 1; zie de Tabellen 2.2, 2.3 en 2.4.

## Een andere oplossing

Het herordenen komt volgens bovenstaande dus uit op een methode die goed werkt ongeacht de specifieken van caches, maar heeft wel de hulp van een ijle matrix partitioneerder nodig. Hoofdstuk 3 behandelt echter een alternatief dat werkt vanuit een ander perspectief: het veranderen van de volgorde waarin elementen uit een matrix betrokken worden in een matrix–vector vermenigvuldiging. Tot nu toe hebben we impliciet aangenomen dat de volgorde rij-voor-rij is, van boven naar beneden, en binnen een rij van links naar rechts. Laten we dit los, dan kan het kiezen van een goede volgorde ervoor zorgen dat berekeningen lokaal blijven. Een geschikte volgorde wordt gegeven door de Hilbert curve; zie Figuur 3.1 op pagina 52. In die figuur wordt laten zien hoe een curve op een $4 \times 4$ matrix kan worden uitgebreid tot eentje op een $16 \times 16$ matrix. Elk $4 \times 4$ kwadrant van die grotere matrix bevat echter een geroteerde versie van de originele curve op de $4 \times 4$ matrix; er is dus een recept om elk kwadrant weer verder te verfijnen, om zo een Hilbert curve op een $32 \times 32$ matrix te verkrijgen. Wederom is een cruciaal ingrediënt recursie, en hiermee gewapend kunnen arbitrair grote matrices ingetekend worden met een Hilbert curve.

De truuk is nu om de volgorde van deze curve te laten bepalen in welke volgorde niet-nullen behandeld worden. Vanwege de recursieve manier waarop de curve gebouwd is, zitten opeenvolgende niet-nullen in deze volgorde ook ruimtelijk dicht bijelkaar; en dus worden er elementen uit $x$ en $y$ aangesproken die ook dicht bijelkaar zitten. Deze methode vereist verder alleen de berekening van de plek van niet-nullen op de Hilbert curve, een relatief veel simpelere berekening dan die partitionering met zich meebrengt. Deze methode was niet nieuw, echter was ze voorheen niet competitief vanwege de moeilijkheid hoe de niet-nullen in een arbitraire volgorde op te slaan. De voor de hand liggende methode (sla voor elke niet-nul de waarde en positie op), gebruikt noodgedwongen meer geheugen dan de standaard opslagmethode. Als de matrix meer geheugen nodig heeft, is er natuurlijk ook meer geheugenverkeer nodig om de vermenigvuldiging uit te voeren; en hier ging het mis. De extra lokaliteit werd teniet gedaan door het veroorzaakte extra verkeer. Dit proefschrift beschrijft een nieuwe datastructuur die dit probleem deels oplost, waardoor de methode met de Hilbert curve opeens wel competitief is geworden. Deze aangepaste datastructuur kan overigens ook worden gecombineerd met de twee-dimensionale herordeningsmethode uit Hoofdstuk 2.

Alhoewel dus sneller te prepareren en competitief gemaakt met de nieuwe datastructuur, zijn de resultaten met deze methode niet zo sterk als wanneer gebruik wordt gemaakt van herordening; zie de Tabellen 3.2 en 3.3. Het verdient wel op te merken dat de methodes van matrixherordening en de hier besproken methode van verandering van de niet-nul volgorde, elkaar totaal niet uitsluiten; er is niets op tegen eerst een matrix te herordenen, om vervolgens aan de slag te gaan met de Hilbert curve.

# Deel II

De voorheen behandelde hoofdstukken bevinden zich in een ander deel dan Hoofdstuk 4 en 5. Dit heeft een eenvoudige reden: tot nu toe is de ijle matrix–vector vermenigvuldiging behandeld als zijnde één enkele berekening, ofwel, een *sequentiële* berekening. Bij herordening wordt echter gebruik gemaakt van partitionering, welk oorspronkelijk parallel rekenen mogelijk maakte; een logische vraag is of we de gevonden methodes niet kunnen combineren met een parallelle vermenigvuldiging, en dit is waar Deel 2 van het proefschrift zich mee bezig houdt.

Een eerste vraag van de lezer is wellicht wat parallellisatie precies inhoudt. Simpel gezegd is het precies het verschil tussen één persoon een taak geven, of die taak geven aan een groep personen. Zijn er vier muren te schilderen en wordt dit gedaan door een enkele schilder, dan is dat een sequentiële oplossing. Wordt de taak volbracht door meerdere schilders, dan noemen we dat een parallelle oplossing. In de wereld van computerberekeningen zijn er, net zoals in het echte leven, triviale taken en moeilijke taken wat betreft parallellisatie. In het geval van vier schilders bijvoorbeeld, is er niet veel fantasie nodig om een initiële taakverdeling te bedenken: Er rijst dan wel direct de vraag wat er gebeurt als de muren niet even groot zijn; als de schilders met de kleinere muren eerder naar huis gaan en zij hun collega's met grotere muren alleen door laten werken, duurt het schilderen in zijn geheel langer dan wanneer samen zou worden gewerkt aan de overgebleven muren. We zeggen dan dat de parallellisatie slecht gebalanceerd is, want ideaal willen we dat de benodigde schildertijd precies gelijk is aan de sequentiële tijd (de tijd die één schilder nodig gehad zou hebben), gedeeld door het aantal schilders.

Computers zijn niet zo intelligent als mensen, en als de taak ook maar een beetje complex is, dan is parallellisatie niet makkelijk. Elke stap moet expliciet duidelijk gemaakt worden aan de computer: hoe moet het probleem initiëel verdeeld worden, hoeveel werkers moet ik tegelijkertijd opstarten, wat als één van die werkers al klaar is, en hoe combineer ik de individuele resultaten van al die werkers tot het eigenlijke eindresultaat? Ook zijn er nog steeds behoorlijk wat verschillende computers op de markt, die allemaal hun eigen communicatiemiddelen tussen werkers beschikbaar stellen; moeten geparallelliseerde taken specifiek voor een bepaalde computer geschreven worden?

Zelfs lastigere vragen doemen op wanneer het kan gebeuren dat werkers 'crashen', vergelijk het met een schilder die plotseling ziek wordt en naar huis gaat. Allicht nog kwalijker kan het in de toekomst ook nog zo zijn dat een computerwerker zijn berekening niet goed doet, en foute antwoorden geeft; zeg maar als een schilder die met de verkeerde pot verf aan de gang is gegaan. Computers, inclusief de voorziene toekomstige architecturen, zijn niet creatief en initiatiefrijk genoeg om werk van hun zieke collega over te nemen, of om zelf achter de fout te komen en vervolgens zelf dan maar de goede pot verf te halen om overnieuw te beginnen; al dat soort acties moeten expliciet geprogrammeerd worden.

## Bulk Synchronous Parallel en multicore rekenen

Of toch niet? Sommige van deze problemen komen al langer voor, en men heeft daar oplossingen voor bedacht in de vorm van *bridging models*; modellen die het idee van een parallelle computer (eentje die meerdere werkers op een enkele taak kan zetten), abstraheren tot een theoretisch apparaat dat iedereen kan programmeren. Vervolgens stellen computerontwikkelaars software-bibliotheken beschikbaar die programma's voor het theoretische apparaat kunnen vertalen naar code die werkt op de computers die zij verkopen. Klassieke voorbeelden van zulke modellen en bibliotheken zijn *Message Passing Interface* (MPI) en *Bulk Synchronous Parallel* (BSP), en richten zich voornamelijk op supercomputers die bestaan uit meerdere fysieke processoren, met elkaar verbonden middels een gespecialiseerd snel netwerk. Het parallelliseren van een berekening begint dan ook meestal met het kiezen van een bridging model, en vervolgens het nadenken en implementeren van een oplossing die snel zou moeten werken binnen dat model.

In het geval van programmeren in BSP, waar in Hoofdstuk 4 van wordt uitgegaan, zijn parallelle programma's opgedeeld in superstappen, en voert elke werker hetzelfde parallelle programma uit. In een superstap kan een werker alleen berekeningen uitvoeren met lokaal beschikbare data; het kan niet zelf bij de gegevens van andere werkers komen, noch kan het communiceren met andere werkers. Tussen superstappen in wordt *gesynchroniseerd*; dat wil zeggen, alle werkers wachten op elkaar zodat tegelijkertijd kan worden begonnen aan de volgende superstap. Het is alleen toegestaan dat tussen de superstappen in wordt gecommuniceerd. De opdrachten daartoe kunnen echter alleen tijdens de superstap worden gegeven. Data van andere werkers kunnen dus tijdens een superstap opgevraagd worden (of andersom, data kan aan andere werkers verzonden worden), maar deze communicatie wordt pas later uitgevoerd en de data is pas op de opgegeven bestemming beschikbaar wanneer de volgende superstap begint. Dit gaat expliciet uit van een parallelle computer met *gedistribueerd geheugen*; werkers hebben hun eigen lokale geheugen en communicatie tussen superstappen in, gebeurt via een netwerk. Een groot voordeel van deze opzet is dat BSP de benodigde parallelle rekentijd kan *voorspellen*. Dit komt omdat de snelheid van werkers bekend is, alsook de snelheid van het netwerk (hoe lang duurt het om zus en zoveel te versturen). Bovendien is de tijd nodig om te synchroniseren (zonder communicatie) meetbaar. Als vervolgens de maximale hoeveelheid berekeningen van de verschillende werkers per superstap bekend is, en als per werker ook bekend is wat de maximum hoeveelheid verstuurde of ontvangen data is, dan zijn alle ingrediënten bekend om de verwachte tijd uit te rekenen.

Tegenwoordig zijn de *multicore* processoren wijd verspreid. Deze combineren meerdere 'cores' (werkers) op een enkele processor; een gemiddelde computer heeft tegenwoordig twee cores, en de wat duurdere varianten hebben er vier of zes op één chip. Het grootste verschil met wat hierboven is beschreven, is dat omdat de werkers op één processor zitten, ze ook de geheugenstructuur delen. Hetzelfde globale geheugen wordt gedeeld, net zoals de caches die tussen de processor en dat geheugen zitten. Dit delen van caches maakt het interessant te bekijken wat de wisselwerking is

met de technieken uit het eerste deel, die ten doel hadden optimaal gebruik te maken van de caches. Bibliotheken specifiek voor gedeeld-geheugen parallellisme bestaan al geruime tijd; bekende voorbeelden zijn OpenMP en POSIX Threads (PThreads).

Hoofdstuk 4 houdt zich echter eerst bezig met het brengen van het oorspronkelijke BSP model naar parallelle computers met gedeeld geheugen. In dit model wordt nog steeds aangenomen dat werkers verbonden zijn met het netwerk, alleen is het netwerk hier nu synoniem met de caches die alle cores met elkaar verbinden; dat wil zeggen, het eigenlijke theoretische model is niet veranderd. Een versimpeling komt wel tevoorschijn bij de bijbehorende BSP software-bibliotheek; communicatie over dat netwerk is dan namelijk synoniem met het kopiëren van data, in plaats van codering en verzending over een netwerk. Omgekeerd maakt de gedeeld-geheugen architectuur het lezen van gegevens, ongeacht van welke werker deze gegevens zijn, veel makkelijker. Op dit gebied is de standaard BSP bibliotheek dan ook uitgebreid, en een concept software-bibliotheek genaamd *MulticoreBSP* is tegelijkertijd met dit proefschrift ontwikkeld.

## IJle matrix–vector vermenigvuldiging in BSP

Nu volgt een korte beschrijving van de parallelle implementatie van de ijle matrix–vector vermenigvuldiging in het BSP model. Het werk dat hoort bij deze vermenigvuldiging wordt veroorzaakt door de niet-nullen in de matrix $A$; elke niet-nul moet vermenigvuldigd worden met het juiste element uit $x$, en de uitkomst daarvan moet opgeteld worden bij het juiste element uit $y$. Dit betekent dat de niet-nullen verdeeld moeten worden over werkers, en wel op een manier zodat de werkers elkaar zo min mogelijk voor de voeten lopen door hetzelfde element uit $x$ op te vragen of te schrijven naar hetzelfde element uit $y$. Zoals eerder opgemerkt worden zulke verdelingen gevonden door ijle matrix partitioneerders zoals Mondriaan, maar wordt er in dit proefschrift niet ingegaan op de technieken van partitioneerders, en wordt simpelweg verondersteld dat externe partitioneerders beschikbaar zijn (en goed werken).

Het klassieke parallelle algoritme partitioneert niet alleen de matrix $A$, maar ook de vectoren $x$ en $y$. Dit is vanwege *schaalbaarheid*; een reden voor parallellisatie is niet alleen snelheidswinst, maar soms ook dat het verdelen over meerdere werkers een extreem groot probleem behandelbaar maakt (een probleem kan simpelweg te groot zijn voor een enkele computer). Om deze reden moet *alle* benodigde data dan ook daadwerkelijk verdeeld worden over alle werkers, in dit geval de matrix inclusief de betrokken vectoren. Bovendien zou er erg veel communicatie nodig zijn als elke werker beide vectoren accuraat bij moet blijven houden.

Alle niet-nullen voor een bepaalde werker worden gegroepeerd in een lokale matrix $\bar{A}$, en gelijk zo met de lokale versies $\bar{x}$ en $\bar{y}$ van $x$ en $y$, respectievelijk. De grootte van lokale objecten $\bar{A}, \bar{x}$ en $\bar{y}$ is kleiner dan of gelijk aan de grootte van de originele matrix en vectoren. Het is nu verleidelijk te zeggen dat elke werker $\bar{y} = \bar{A}\bar{x}$ moet uitrekenen en dat is alles; maar niets is minder waar. Omdat $x$ en $y$ verdeeld zijn, is het mogelijk dat niet-nullen in $\bar{A}$ corresponderen met elementen uit $x$ die niet lokaal zijn (niet in $\bar{x}$ zitten), maar juist bij een andere werker aanwezig zijn. Evenzo kunnen

er niet-nullen in $\bar{A}$ zitten die moeten communiceren met elementen uit $y$ aanwezig bij andere werkers. Het standaard BSP algoritme haalt derhalve in de eerste superstap alle lokaal nodige elementen uit $x$ vanuit de andere processoren. In de volgende superstap zijn deze elementen beschikbaar, en kan rij-voor-rij de daadwerkelijke matrix–vector vermenigvuldiging uitgevoerd worden. Als een rij niet lokaal is, dat wil zeggen, als de uitkomst bij een element van $y$ moet worden opgeteld dat niet aanwezig is in $\bar{y}$, dan wordt deze uitkomst doorgestuurd naar de werker die het element wel heeft. Hierna wordt weer gesynchroniseerd, en in de laatste superstap worden de binnenkomende bijdragen voor $\bar{y}$ opgeteld; het is immers net zo goed mogelijk dat andere werkers niet-nullen in een rij hadden die corresponderen met elementen in onze $\bar{y}$.

Op systemen met gedeeld geheugen is het opvragen van niet-lokale gegevens uit $x$ wel direct mogelijk, zonder superstap, maar ook hier gebeurt zoiets liever met mate. Dit omdat het inefficiënt cachegebruik waarschijnlijk maakt: cache-gebruik zou goed zijn juist wanneer alle werkers hetzelfde element op hetzelfde moment opvragen, maar om dit te garanderen moeten de werkers constant communiceren om ervoor te zorgen dat ze in hetzelfde tempo werken (ofwel, constant synchroniseren). Zoiets zou betekenen dat ze meer met elkaar praten dan dat ze daadwerkelijk werk verzetten. Voor $y$ is hetzelfde *niet* mogelijk. Stel dat element $y_3$ uit $y$ op het moment gelijk is aan $0.5$, en dat een werker het getal $3.5$ bij $y_3$ wil optellen terwijl er tegelijkertijd een tweede werker is die $5$ wil optellen bij $y_3$. De eerste werker telt lokaal $3.5$ bij $0.5$ op en komt uit op $4$, en schrijft dat terug naar $y_3$. De tweede werker is zich, net zo min als de eerste werker, van geen kwaad bewust en doet hetzelfde met $5$, en schrijft dan $5.5$ terug naar $y_3$. Beide schrijfopdrachten racen terug naar het globale geheugen waar de vector $y$ staat, en wie als laatste aankomt wint[3]. De waarde van $y_3$ wordt dus $5.5$ of $4$, beide ongelijk aan het eigenlijke antwoord $(9)$. Dit is dan ook de reden dat MulticoreBSP het lezen van data van een andere werker wel toestaat, maar het schrijven daartoe niet. Al met al is het algoritme voor systemen met gedeeld geheugen hetzelfde als het algoritme voor gedistribueerd geheugen wat hierboven beschreven is, behalve dat niet-lokale elementen uit $x$ direct kunnen worden uitgelezen en er dus een superstap minder nodig is. Wegens efficiënt cachegebruik met betrekking tot niet-lokale elementen uit $x$, is een goede partitionering in beide richtingen ($x$ en $y$) nog steeds belangrijk.

## Herordening en parallellisme op systemen met gedeeld geheugen

Zoals eerder opgemerkt worden ook caches gedeeld tussen werkers wanneer multicore systemen gebruikt worden. Hoofdstuk 5 combineert de inzichten van al het voorgaande om een cache-efficiënte parallelle vermenigvuldiging te bewerkstelligen. De methode daar gepresenteerd kan als volgt geschetst worden.

Stel dat we de beschikking hebben over twee cores, en dat de matrix in doubly

[3]Het is ook goed mogelijk dat deze fout door het systeem gedetecteerd wordt, wat doorgaans de hele parallelle berekening afbreekt met een foutmelding.

SBD-vorm is gebracht dankzij de technieken ontwikkeld in Deel I:

$$\tilde{A} = \begin{pmatrix} A_1 & S_1 & 0 \\ S_2 & S_3 & S_4 \\ 0 & S_5 & A_2 \end{pmatrix}.$$

Met de vorige sectie in gedachten, en onder aanname van geschikte vector-distributies voor $x$ en $y$, zien we dat $A_1$ en $A_2$ corresponderen met de niet-nullen die werkers 1 en 2 compleet onafhankelijk van elkaar kunnen behandelen. $S_1$ correspondeert met niet-nullen waarvoor werker 1 *mogelijk* een element uit $x$ van werker 2 moet halen, en evenzo voor $S_5$ maar met de rollen van de twee werkers omgedraaid. $S_2$ correspondeert met niet-nullen waarvoor werker 1 mogelijk bijdragen aan een element uit $y$ naar werker 2 moet sturen, en wederom omgekeerd hetzelfde voor $S_4$. $S_3$ heeft de hoofdprijs; hier moeten werker 1 en 2 zowel mogelijk elementen van $x$ van de andere werker opvragen, alsook mogelijk bijdragen aan $y$ sturen naar de andere werker.

Willen we gebruik maken van de herordening, dan is het makkelijker te veronderstellen dat er geen lokale versies van $\tilde{A}$, $x$ en $y$ gevormd worden, maar dat alle werkers op het origineel opereren. Dit betekent dat we buiten het BSP model om werken; de experimentele software ontwikkeld bij dit hoofdstuk is dan ook geschreven in PThreads in plaats van met MulticoreBSP. In de vorige sectie is beargumenteerd dat het lezen van elementen van $x$ bij andere werkers toegestaan is voor systemen met gedeeld geheugen. Dit betekent dat het voor werkers toegestaan is om zonder meer te lezen uit een globale $x$, omdat de herordening gebaseerd is op de partitionering. $S_{\{1,5\}}$ lijken zich in die zin automatisch goed te gedragen, maar $S_{\{2,3,4\}}$ zijn lastiger aangezien onder geen voorwaarde meerdere werkers naar hetzelfde deel van $y$ mogen schrijven. Dit is in het huidige hoofdstuk opgelost door meervoudige synchronisatie; eerst wordt alleen werker 1 toegestaan op $S_{\{2,3,4\}}$ te werken, en daarna, dus na synchronisatie, is werker 2 aan de beurt. Deze methode is verder verfijnd door de rijen van $S_{\{2,3,4\}}$ van te voren in tweeën te verdelen, zodat beide werkers op (ongeveer) de helft van de rijen van deze scheidingsmatrices kunnen opereren. Na synchronisatie wisselen de werkers van helft om de berekening af te maken. Op deze manier is de werkbalans optimaal; in de voorgaande versie zou telkens één werker louter toekijken hoe de ander aan de slag gaat, terwijl ze nu beiden ongeveer evenveel te doen hebben. Het enige nadeel is dat het aantal synchronisatiestappen proportioneel is aan het aantal werkers. Hoe meer werkers, hoe meer tijd er verloren gaat aan synchronisaties. Wil deze methode aantrekkelijk blijven, dan moet de grootte van de te vermenigvuldigen matrix dus meeschalen met het aantal werkers.

Net zoals bij Hoofdstuk 1 en 2 is het mogelijk meer te halen uit deze methode door gebruik te maken van recursie. Ook al zouden er alleen twee werkers beschikbaar zijn, het recursief opdelen van $A_1$ en $A_2$ zou voor beter cache-gebruik van die twee werkers moeten zorgen, omdat berekeningen van beide werkers lokaler worden in $x$ en $y$. Samenvattend is de truuk om het aantal partities dus hoger te nemen dan het aantal beschikbare werkers, zodat elke werker de SBD-vorm individueel kan uitbuiten, terwijl op hoger niveau de werkers elkaar dankzij de partitioneerder zo min

mogelijk voor de voeten lopen wat betreft de elementen die gedeeld moeten worden tussen meerdere werkers.

## Resultaten van parallelle methoden

Resultaten van parallellisatie met MulticoreBSP zijn te vinden in de Tabellen 4.8 en 4.9, vanaf pagina 80. De combinatie met herordening is minder uitgebreid getest, met de resultaten op twee architecturen en twee voorbeeldmatrices weergegeven in Tabel 5.2 op pagina 94. Het verdient nog een opmerking dat de MulticoreBSP software-bibliotheek zich niet beperkt tot de ijle matrix–vector vermenigvuldiging alleen; ook oplossingen van andere problemen zijn geïmplementeerd en onderzocht op efficiëntie. Deze andere problemen zijn het berekenen van het inwendig product van twee vectoren, de snelle Fouriertransformatie (zoals ook gebruikt in compressie van muziek en beeld), en LU decompositie (welke werkt op volle, in tegenstelling tot ijle, matrices). Hoofdstuk 4.6 zet alle resultaten verkregen met deze toepassingen uiteen. De combinatie van resultaten laat zien dat MulticoreBSP, inclusief de uitbreiding op de standaard BSP bibliotheken, inderdaad voor goede parallellisatie op huidige multicore computers kan zorgen.

# Curriculum Vitae

Albert-Jan Nicholas Yzelman was born on the 17th of July, 1984, at the Rijnstate hospital in Zevenaar, the Netherlands. He was raised in Duiven, where he attended the elementary school 'De Wiekslag'. After obtaining his Atheneum diploma in 2002 from the 'Candea College' high school, he pursued a bachelor's degree in Mathematics at Utrecht University. He graduated in both Mathematics and Computing Sciences, with his final research project on the subject of parallel radiosity rendering done under the supervision of Rob Bisseling.

Albert-Jan stayed at Utrecht University to continue with the master's programme Scientific Computing, which he concluded in 2007. For his thesis he moved to Rotterdam to start an internship at Alten Nederland. This involved researching R-trees in application to oil reservoir simulation software, and was performed in cooperation with Royal Dutch Shell. He was supervised by Arno Swart, Gerard Sleijpen, Eric Haesen and Hans Molenaar.

Following this, Albert-Jan took an opportunity to remain at Utrecht University as a PhD candidate under supervision of Rob Bisseling. Four years were spent in researching various aspects of high performance computing applied to the problem of sparse matrix–vector multiplication, which resulted in the thesis now before you.

His secondary duties at the university also included teaching; ranging from heading exercise sessions for undergraduates, to being solely responsible for teaching a graduate-level class. From 2009 onwards, he also participated in organising a yearly event for PhD students in Scientific Computing located in the Netherlands and Flanders.

Albert-Jan will continue research in parallel computing at the Katholieke Universiteit Leuven in Belgium, starting September 2011.