Università degli Studi di Roma "Tor Vergata"

Facoltà di Ingegneria

Dottorato di Ricerca in
Informatica ed Ingegneria dell'Automazione
Ciclo XXIII

A Quad-Tree Based Sparse BLAS Implementation for Shared Memory Parallel Computers

Michele Martone

Tutor:	Prof. Salvatore Tucci
Co-relatori:	Prof. Salvatore Filippone, Prof. Marcin Paprzycki
Coordinatore:	Prof. Daniel P. Bovet

"Ladies and gentleman, this is your captain speaking. I have some good news and I have some bad news. The good news is, that we have a very strong tail wind, and we are doing one thousand four hundred miles per hour over land. The bad news is, that all of our navigation instruments are out, and we don't know where we are, and we don't know where we are going."

Joseph Weizenbaum

"*Rebel at Work*", a documentary film about Joseph Weizenbaum, by Peter Haas and Silvia Holzinger, 2007. Words spoken between 1:19:54 and 1:20:45.

Contents

List of Figures

List of Tables

Acknowledgements

When I've heard for the first time Prof. Salvatore Filippone's words *"the issue of efficient multiplication of a sparse matrix by a dense vector is worth the effort of an entire PhD"* I said to myself I would never bother with that.

Now I am grateful to Prof. Salvatore Tucci, who gave me the chance of taking PhD studies, and Prof. Salvatore Filippone who proved me that I was wrong. Indeed, he proved to me that I have been wrong many times, each time providing me with balanced hints and motivation for further investigation.

I also had the delight of being confuted by Prof. Marcin Paprzycki, who encouraged me into the design of a fully featured sparse matrix library. Then, I said to him: *"no way, that's impossible"*. Now I owe him great encouragement and motivation, and great satisfaction in the problems I had to solve during this work. He helped me into focusing my work by conceiving tangible, clearly stated goals, and still providing with a very solid collaboration.

Thanks go to Paweł Gepner, for allowing us to use computational facilities at Intel EMEA, with the technical support from Jamie Wilcox and Victor Gamayunov. Thanks also to Krzysztof Luka of **AMD** Polska for giving us access to their facilities.

This thesis would not have been possible without the technical support of Bartłomiej Solarz-Niesłuchowski at IBS PAN in Warsaw, who tirelessly helped us when our workstations melted, and granted us continuous access and data storage into one of the main production servers at IBS PAN.

I am also grateful to Prof. Przemysław Stpiczyński (Marie Curie University of Lublin) for his encouragement and sheer interest into my work, as well as his unpaired friendliness.

Working on this thesis has been hard, although relieved by a number of people who kindly provided me with suggestions and critiques. I am grateful to Salvatore Filippone for having read and suggested improvements to early versions of the draft. I wish to thank Prof. Patrick Amestoy from ENSEEIHT-IRIT Toulouse and Prof. Anne C. Elster from NTNU Trondheim for having read the whole thesis and providing several useful suggestions for improving its form, as well and having spotted a number of typographical errors. I also wish to thank committee member Prof. Paolo Franciosa for having spotted a subtle error in the thesis draft, and promply suggesting a fix to it. I have received several accurate suggestions on how to improve the introductory sections, as well as the overall exposition, from Prof. Valeria Cardellini. Prof. Alberto Pettorossi has been able to expose some weak points in the introductory sections, by just gleaning over the thesis first pages; his critical spirit (that I have always appreciated as a student in his classes) did the rest.

I also wish to thank Prof. Charles Leiserson and Dr. Matteo Frigo for having provided me with a hard-to-find digital copy of an article by Morton ([G.M66]).

In the making of this thesis, I owe the most to my aunt Anna, who tirelessly provided me with many suggestions on how to improve the English form of this thesis, especially with the finer nuances I was simply not aware of.

I wish to thank my parents, and my beloved *babcia*, because they keep teaching me what it really matters in life, and no PhD will ever teach.

I wish to thank friends who accompanied me in train and walk journeys around Europe, bike trips around Warsaw woods, and getting lost in Italian mountains. I also thank friends who accompanied me into thinking and appreciating the beauty of what was already next to me. It has been great fun being regarded (me!) as the most renowned cook around, in the dormitory that hosted me in Warsaw; it has also been a pleasure making Ukraininan friends, there.

My PhD studies have been funded by a scholarship by the Italian Ministry of Higher education; my years as an undergraduate have been partially funded by scholarships, as well. I feel lucky for having had this opportunity.

E infine vorrei ringraziare con affetto proprio te, che *sparisci nella notte*.

Introduction

Although it is one of the oldest branches of mathematics, the field of numerical analysis (intended as *the study of algorithms for the problems of continuous mathematics*—see [TB97, p.323]), has seen its major development over the last sixty years, most likely due to the introduction, development and growth of computer systems, as well as computer science.

Nowadays, the solution to many *continuous problems* occurring in engineering, science, and finance requires the solution of *linear systems*, or computation of *eigenvalues* and *eigenvectors*.

Numerical linear algebra is the branch of numerical analysis which studies the solution to these problems. Usually, most of the information describing the mentioned problems classes is represented and manipulated by means of *tuples* of numbers, known as *vectors* and *matrices*. The foremost technology used to automate this computation today is that of the *digital computer* which mechanizes the handling of *arrays* of data representing vectors and matrices.

This thesis discusses techniques for performing efficiently some *sparse numerical linear algebra* computations[1] on currently available digital computers.

While generally, the efficiency of *computer algorithms* must be defined and estimated theoretically around some abstract *computation model*, one of the tasks of a computer engineer is to use both algorithmic and technical computer knowledge in order to implement working algorithms that are efficient in the *physical resources* usage.

Three prominent resources to consider are: energy, time, and space. To be regarded as *efficient*, an *implementation*[2] of a *computational method* should not exceed the use of any of the aforementioned resources.

The metric most used in this thesis is that of *time* and, to a certain extent, *space*, as we will often regard computations as being *efficient* (or in jargon, *high performance*) if they take a relatively *short time to run*, use a *modest amount of memory*, and/or employ a *limited number* of processing units. Since all of our experiments have been carried out on single computers, and since savings in memory usage usually implied savings in time, we had to focus mostly on *time*.

[1]That is, the algebraic computations involving *numerical matrices* which are *sparse*. A sparse matrix is *"any matrix with enough zeros that it pays to take advantage of them"*; this definition is attributed to James H. Wilkinson (see [Dav07]), one of the fathers of numerical linear algebra.

[2]Intended as a combination of computer *hardware* and *software*.

We were lucky indeed, as it seems that arranging *time-efficient* computations into also being necessarily *energy efficient*, will be an additional problem computer engineers shall face in the forthcoming future[3].

[3]Energy consumption and power issues have been taken in consideration during the whole history of microchip technology development, but only recently these have become major constraints in both chip and overall systems design (see Patterson and Hennessy [PH04, § 1.5,§ 3.6] for an overview; see Fuller et al. [Com11, Ch. 3] for a whole chapter about the subject).

Contributions

This thesis is devoted to the study of efficient computer codes for linear algebra operations on *sparse* matrices.

Specifically, we develop techniques for cache based, shared memory parallel machines; that is, nowadays, the vast majority of the general purpose computers in commerce[4]. In the near future, this kind of machines is expected to be even more popular. We seek efficiency mainly on the problem instances where known techniques bring inefficiencies, namely problems with matrices which are *large*, in the sense they occupy a relevant portion of a single computer's [5] random access memory.

Our ultimate goal is that of designing a sparse matrix layout and algorithms capable of supporting a set of operations broad enough for a whole Sparse BLAS library[6]. As we will see, our techniques are based on a *quad-tree* based organization of sparse matrices; that is, we subdivide a matrix in quadrants repeatedly, building a tree structure having up to four children for each internal node.

Our (recent) literature starting points are primarily the works of Alfredo Buttari and Richard Vuduc. We follow their suggestion for automatically generating specialized, high performance numerical codes (e.g.: see their theses [Vud03], [But06]), and to *adapt* or *choose* data structures with regard to both the matrix and machine at hand. During the development of the core of this thesis material, we discovered Buluç's (e.g.:[Bul10]) and Yzelman and Bisseling's (e.g.:[YB09]) work about sparse kernels with non-linear layouts. While Yzelman and Bisseling develop *cache oblivious* sparse matrix formats[7], and Buluç uses some cache oblivious techniques as part of a whole, our data structures and

[4]We have chosen to focus on traditional *general purpose computers* CPUs, rather than on recently developed *graphical processing units* (GPUs; e.g.: see Baskaran and Bordawekar [BB09] for an application in sparse matrices computations) or non-traditional computer architectures (like *gaming consoles*; see Buttari et al. [APJ$^+$07] for an example) capable of general purpose computations.

[5]Intended here as "single node, multiple core computers".

[6]**BLAS** stands for Basic Linear Algebra Subroutines; an *application programmer interface* for reusable, high performance implementations, comprehensive of sparse extensions see (Duff et al. [DHP02]).

[7]Or rather cache oblivious *algorithms* associated to that format, as given in Prokop's definition ([Pro99, p.10]): *"We define an algorithm to be cache aware if it contains parameters (set at either compile-time or runtime) that can be tuned to optimize the cache complexity for the particular cache size and line length. Otherwise, the algorithm is cache oblivious."* rather than that in Frigo et al. [FLPR]: *"no variables dependent on hardware parameters, such as cache size and cache-line length, need to be tuned to achieve optimality"*.

techniques are ultimately *cache aware*, even if they are *cache oblivious* down to a certain degree of approximation.

Although not described in detail in this thesis, a fundamental practical tool we used throughout our work was a custom system for code generation; with it, we have been able to both abstract from the *numerical types* at hand, and perform careful code specialization in producing the various numerical kernels a Sparse BLAS implementation offers.

Our published contributions so far are about:

- The development of the recursively quad-partitioned *CSR* format (*RCSR*): ([MFT+10]) — see §2.3.
- The development of shared memory parallel algorithms for fundamental Sparse **BLAS** kernels for *RCSR*: *SpMV* ("$y \leftarrow \ y \ + \ A \ x$")[8], and *SpSV* ("$x \leftarrow \ \alpha \ L^{-1} \ x$")[9]([MFPT10b]) — see §3.
- Performance tuning of *RCSR* for the aforementioned kernels ([MFPT10c]; see §4.1), and a further format tuning and generalization towards a *hybrid* format — *RSB* ([MFG+10]; see §4.3).
- A study of an aspect of sparse matrix computations often neglected in published research: the time needed for instancing the matrices data structures, measured relatively to the time of a single *SpMV*, in the context of a thread-parallel implementation ([MFPT10a]) — see §5.

In addition, the format we have developed allows shared memory parallel implementations of the *SpMV-T* kernel (the transposed variant of *SpMV*) with a degree of parallelism (and thus likely, performance) higher than allowed by *CSR*: see §C. This last feature is perhaps one of the most valuable contributions of our work.

Besides our literature contributions, we also wish to pay a tribute to the *free software* community by soon releasing our prototypal code with a free software[10] licensing, and by interfacing it to an existing (free software) project: the

[8]That is, the update "$\forall i \in [1, .., m], \ y_i \leftarrow \ y_i \ + \sum_{j=1}^{k} \ a_{ij} \ x_j$".

[9]That is, the update (performed in the order $1, .., m$ to meet dependencies) "$\forall i \in [1, .., m], \ x_i \leftarrow \ (x_i \ - \sum_{j=1}^{i-1} \ a_{ij} \ x_j)/a_{ii}$". For more details about our notation conventions, see §D.

[10]That is, software whose licensing lets *"users freedom to run, copy, distribute, study, change and improve the software"*, according to the *Free Software Foundation*'s definition (see [FSF10]).

PSBLAS library for *distributed* memory parallel Sparse BLAS computations (Filippone and Colajanni [FC00]).

Without availability of *free software*, Information Technology would be nowhere as interesting as it is today.

Thesis Outline

Here we briefly outline the organization of this thesis. In Ch.1 we introduce well-known, non hierarchical sparse matrix representation formats and algorithms. We describe the *COO* and *CSR* formats (§1.1 and §1.2), discussing variants of algorithms for the implementation of the basic operations of our interest: multiplication by a vector (*SpMV*) and triangular solution (*SpSV*). When presenting these formats and algorithms (as pseudo-code; see §D for our notation), we also give mention to many performance-related aspects when implementing them in a compiled language (say C or Fortran) on a machine of our interest (*shared memory parallel, cache based*). In the discussion, we also mention modifications for supporting *transposed*, *symmetric*, and other format/operation variants; two variations of the *CSR* format (§1.2.4); other useful operations (§1.3). Since a central issue to performance is the memory access pattern of the discussed algorithms, we summarize this information in tabular format, in §1.4. We close the chapter with literature references (§1.5).

In Ch.2 we introduce the topic of *hierarchical* representation formats of sparse matrices. After a literature introduction (§2), we present two hierarchical matrix formats which are of particular interest to us: *CB* and *CSB* (respectively in §2.1 and §2.2). Then we are ready to introduce our own techniques, beginning with the hierarchical, *recursive CSR* (*RCSR*) layout, in §2.3. There we give both a *serial* and a trivial *dual-thread parallel* formulation of the computational algorithms of interest. In §2.4 we present performance experiments for a first implementation of this layout.

In Ch.3, we develop multi-threaded shared memory algorithms for the *RCSR* format (§3.1, §3.2). We present performance results for these techniques in §3.3.

By taking note of the performance results obtained for the *RCSR* format in Ch.3, we devote Ch.4 to the *tuning* of *RCSR*. We first introduce a technique for storing *RCSR* matrices with *shorter* indices in §4.1, looking at its performance in §4.2. Then we extend the definition of our recursive layout into supporting *leaf submatrices* in *COO* format in §4.3, presenting the performance results in §4.4. We name the hybrid format resulting from these modifications *Recursive Sparse Blocks* (*RSB*).

In Ch.5, after a literature introduction (§5.1) and a review of basic properties of the *RSB* data structure (§5.2), we present partially parallel algorithms for building *RSB* matrices (§5.3, §5.4). In the chapter, we report ratios of the time for building an *RSB* matrix instance to that for executing a single *SpMV* operation, both serially and in parallel. We close the chapter outlining an enhanced parallel build algorithm in §5.6.

We close the thesis with Ch.6. There we also present some future work: a number of possible *minor* enhancements to *RSB* (§6.2), as well as *major* ones (§6.3).

We have chosen to give the reader some supplementary material which is not essential into following the main thesis discourse. In §A, we give details for the setup of the experiments made throughout the thesis. In §B, we perform some proof-of-concept *memory scanning* experiments in order to justify a number of claims made in the thesis. In §C, we present some extra performance results of our *RSB* prototypal code, when compared to a proprietary, highly optimized computational library. A description of notation conventions used in the thesis is given in §D.

1 Representation of Sparse Matrices

Overview

In this chapter, we give an overview of the most common ways for representing sparse matrices and performing computations on them on the currently available general purpose computers; that is, using their Central Processing Units (CPUs)[1].

We describe data structures and provide the well-known algorithms for performing most common operations on them, like *multiplication* or *triangular solve* by a dense vector.

Although the term *data structure* or *sparse matrix layout in memory* would be more appropriate than *(storage) format*, we will often use the latter for historical reasons.

Reports on exploiting particular data structures for sparse matrices date back to late sixties. During the seventies and eighties, as sparse computations research blossomed, representation formats were developed to suit particular algorithms and/or architectures (see Pissanetzky [Pis], Dongarra et al. [DDSvdV98, Ch. 1])[2].

Reference information for "classical" sparse matrix formats and algorithms

[1] Recent development of technologies for doing these computations on GPUs (Graphical Processing Units) has sometimes led to different techniques, although one of the recurrent themes for optimization here, is often similar to that for CPUs: arranging data for locality and avoiding code branches. For instance in [BB09, § 4], Baskaran and Bordawekar describe what seems a *sparse blocked* (see §2.1) *BCSR* (see §1.2.4) format-based optimization.

[2] And to a good extent, we will be doing so in the research presented in this thesis.

$$A = \begin{pmatrix} 0.66667 & 0.36656 & 0.30011 & 0.36656 & 0.30011 \\ 0.10004 & 0.53341 & -1 & 0.20007 & 0 \\ 0.12219 & 0 & 0.5777 & 0 & 0.24437 \\ 0.05002 & 0.10004 & 0 & 0.28331 & 0.18328 \\ 0.06109 & 0 & 0.12219 & 0.15006 & 0.27224 \end{pmatrix}$$

$$nnz(A) = 20 \quad m(A) = 5 \quad k(A) = 5$$

Figure 1.1: Matrix *cage3**. We denote the number of non zero entries of A by $nnz(A)$; its rows count by $m(A)$; its columns count by $k(A)$. For a sparse representation of A, we would like to use as little data as possible, in order to minimize use of available *random access memory* and prevent unnecessary computations (as usually operations involving zeroes are).

is contained in many books: like Barrett et al. ([BBC+94, § 4.3.1]), Saad ([Saa03, § 3.4]), or Pissanetzky ([Pis]).

In §1.1 and §1.2, we introduce the two most well known formats in use today; namely *COO* and *CSR*, and some variations of them, as well as discussing possible implementations for the operations we are interested in. In §1.3, we mention other operations commonly performed on sparse matrices, and comment on them, in relation to our choices (in our role as sparse matrix format designers). We come back with a summary discussion, and a concise account of the *memory access patterns* of the algorithms presented in §1.1,§1.2, in §1.4.

Finally, in §1.5.1, we mention some more formats which may be relevant to our future discussion, and conclude by giving a number of related literature pointers in §1.5.2.

Throughout this and the following chapter, we use matrix *cage3**, shown in Fig. 1.1, as an example for illustrating the layout of sparse matrices in memory, across *formats*. Matrix *cage3** is obtained by adding (for our convenience) the -1 value in position $(2, 3)$ of matrix *cage3*. Matrix *cage3* belongs to the *University of Florida sparse matrix collection*. This collection, authored by Tim Davis, is the most complete publicly available one; it also incorporates existing historical collections of representative sparse matrix problems and matrices. See Davis ([Dav10]) for a description of the above mentioned collection[3]. Most of the experiments in this thesis were conducted on matrices from the above collection. Of course, for these experiments, we have chosen matrices which are much *larger*

[3]See also the *Test for Large Systems of Equations* project (`http://www.gridtlse.org/`, led by Patrick Amestoy) for a comprehensive database of publicly available linear systems, software, and related statistics.

$$A = \begin{pmatrix} 0.66667^{(3)} & 0.36656^{(1)} & 0.30011^{(2)} & 0.36656^{(4)} & 0.30011^{(5)} \\ 0.10004^{(6)} & 0.53341^{(7)} & -1^{(8)} & 0.20007^{(9)} & 0 \\ 0.12219^{(10)} & 0 & 0.5777^{(11)} & 0 & 0.24437^{(12)} \\ 0.05002^{(13)} & 0.10004^{(14)} & 0 & 0.28331^{(15)} & 0.18328^{(16)} \\ 0.06109^{(17)} & 0 & 0.12219^{(18)} & 0.15006^{(19)} & 0.27224^{(20)} \end{pmatrix}$$

IA=(1 1 1 1 1 2 2 2 2 3 3 3 4 4 4 4 5 5 5 5)
JA=(2 3 1 4 5 1 2 3 4 1 3 5 1 2 4 5 1 3 4 5)
VA=(0.36656 0.30011 0.66667 0.36656 0.30011 0.10004 0.53341 -1 0.20007 0.12219 0.5777 0.24437 0.05002 0.10004 0.28331 0.18328 0.06109 0.12219 0.15006 0.27224)

Figure 1.2: A *COO* representation of matrix *cage3**. Superscripts of each matrix entry represent the position index of that entry in the input arrays. Notice that nonzeroes are listed in no particular order in the three arrays.

and *sparser* than *toy example matrix cage3**. When describing a sparse matrix A, we will often denote the number of its non zero entries (or *nonzeroes*) with $nnz(A)$; its rows count by $m(A)$; its columns count by $k(A)$.

1.1 Coordinate Representations

The simplest of the sparse storage formats is commonly known as *COO*, as it represents a matrix A with m rows and k columns just as a *list of coordinates* on a two dimensional grid, with associated values; that is, by enumerating explicitly its *nonzero entries.*

This representation uses two *integer* arrays for storing the coordinate indices: *IA*, *JA*, and one (*VA*) with the actual numerical values.

All arrays are long $nnz(A)$. In *IA* and *JA*, it is customary to store indices starting with 1 when programming in the Fortran language, and with 0 in the C language. In practical applications, sometimes it is desirable to store explicitly certain zero values for later modification. In these cases, the explicit zeroes (in jargon, *structural nonzeroes*) are stored in addition to the effective nonzeroes, contributing to the overall stored *nonzeroes* count.

See Fig. 1.2 for an example instance of matrix *cage3** represented with *COO* arrays. The superscripts above each matrix A nonzero coefficient show the (one-based) position of the coefficient data (row index, column index, numerical value) in the arrays *IA*, *JA*, *VA*.

$$A = \begin{pmatrix} 0.66667^{(1)} & 0.36656^{(2)} & 0.30011^{(3)} & 0.36656^{(4)} & 0.30011^{(5)} \\ 0.10004^{(6)} & 0.53341^{(7)} & -1^{(8)} & 0.20007^{(9)} & 0 \\ 0.12219^{(10)} & 0 & 0.5777^{(11)} & 0 & 0.24437^{(12)} \\ 0.05002^{(13)} & 0.10004^{(14)} & 0 & 0.28331^{(15)} & 0.18328^{(16)} \\ 0.06109^{(17)} & 0 & 0.12219^{(18)} & 0.15006^{(19)} & 0.27224^{(20)} \end{pmatrix}$$

IA=(1 1 1 1 1 2 2 2 2 3 3 3 4 4 4 4 5 5 5 5)
JA=(1 2 3 4 5 1 2 3 4 1 3 5 1 2 4 5 1 3 4 5)
VA=(0.66667 0.36656 0.30011 0.36656 0.30011 0.10004 0.53341 -1 0.20007 0.12219 0.5777 0.24437 0.05002 0.10004 0.28331 0.18328 0.06109 0.12219 0.15006 0.27224)

Figure 1.3: *COR* (coordinates ordered by rows) representation of matrix *cage3**. Notice that the sole difference with Fig. 1.2 is the order of elements in the three arrays.

Usually, when referring to *COO*, one does not assume any ordering among the elements (see Saad [Saa03, 3.4]). However, for performance reasons, software using *COO* defines some particular order: for instance, **PSBLAS** and **SPARSKIT** (see Saad [Saa94]) do not assume any ordering in the representation, in general. **PSBLAS**, however[4], sorts by rows the *COO* representation whenever appropriate according to the required operations.

From now on, we will denote by *COR* a "*COO* ordered by rows", and by *COC* a "*COO* ordered by columns", while continuing using *COO* when no order is specified. In our definition, we also assume that *COR* has coefficients ordered *by column index* within rows, and *COC* is ordered *by row index* within columns. In Fig. 1.3, we show an example of the row-major (*COR*) *COO* representation; in Fig. 1.4, its transposed ordering (*COC*).

Assuming $a_{i,j} \neq 0$ to be the numerical value of matrix A at the i^{th} row and j^{th} column[5], for all of the above representations we have: $VA(l) = a_{i,j}$; $IA(l) = i$; $JA(l) = j$ for some $1 \leq l \leq nnz(A)$.

For *COR* and $\forall 1 \leq p < q \leq nnz(A)$, we also have[6] either:

$IA(p) < IA(q)$

or

[4] Version 2.4.

[5] For more details about our notation conventions, see §D.

[6] In the presented algorithms, we assume no duplicates in these arrays. In real applications, duplicate values are handled according to some policy, as it could be keeping the first occurrence, keeping the maximum value, summing duplicates, or averaging.

$$A = \begin{pmatrix} 0.66667^{(1)} & 0.36656^{(6)} & 0.30011^{(9)} & 0.36656^{(13)} & 0.30011^{(17)} \\ 0.10004^{(2)} & 0.53341^{(7)} & -1^{(10)} & 0.20007^{(14)} & 0 \\ 0.12219^{(3)} & 0 & 0.5777^{(11)} & 0 & 0.24437^{(18)} \\ 0.05002^{(4)} & 0.10004^{(8)} & 0 & 0.28331^{(15)} & 0.18328^{(19)} \\ 0.06109^{(5)} & 0 & 0.12219^{(12)} & 0.15006^{(16)} & 0.27224^{(20)} \end{pmatrix}$$

IA=(1 2 3 4 5 1 2 4 1 2 3 5 1 2 4 5 1 3 4 5)
JA=(1 1 1 1 1 2 2 2 3 3 3 3 4 4 4 4 5 5 5 5)
VA=(0.66667 0.10004 0.12219 0.05002 0.06109 0.36656 0.53341 0.10004 0.30011 -1 0.5777 0.12219 0.36656 0.20007 0.28331 0.15006 0.30011 0.24437 0.18328 0.27224)

Figure 1.4: *COC* (coordinates ordered by columns) representation of matrix *cage3**. Notice that the sole difference with Fig. 1.2 is the order of elements in the three arrays.

$IA(p) = IA(q)$ and $JA(p) < JA(q)$.

For *COC*, the same relation holds, after exchanging the two index arrays.

On current cache based machines, whether arranging the data layout and algorithms for *COR* or for *COC* can have an impact on the computation performance, as the actual order of memory references is different. Most CPUs are capable of *prefetching* memory locations in cache, often favouring sequential accesses rather than *random accesses*; see manuals for popular CPUs: Intel [AMD07, § 3.9] or AMD [Int08a, Ch. 7]. Even if not completely random, the *indirect accesses* (for instance, accessing a sequence of memory locations at the offsets given by an indices array) trick current prefetch engines into failure.

If the CPU had a prefetch device capable of preloading memory locations by *looking ahead* of the *contents* of the indices array (rather than its address only), the negative impact of indirect references would be much lower[7]. However, it is not clear whether such a CPU functionality would be general enough to be taken in consideration by current CPU manufacturers.

In the following sections, we provide listings for common algorithms for the

[7]Indeed, *vector processors* built between 1980's and 1990's supported exactly this kind of indirect addressing, also known as *sparsed vector gather*. In Dekeyser et al. ([DMP90]): *"..Now, nearly all vector supercomputers provide instructions in their instruction set to realize such operations (Cray X-MP/4, Cray 2, Cray Y-MP, Fujitsu VP-200, NEC SX-2)."*; see also Cheng ([Che89]) for an overview on the IBM 3090 and Cray X-MP machines. Current *vector extensions* to *scalar* CPUs do not support indirect addressing, but things may change, as additional instruction sets are being developed by the major chip manufacturers (see also Gebis and Patterson — [GP07]).

unordered and two ordered variants of *COO*.

1.1.1 *SpMV* for *COO/COR/COC*

The general form of the *SpMV update*, required by iterative methods is that of "$y \leftarrow \beta\ y\ +\ \alpha\ A\ x$". Algorithm listings we give here implement *SpMV* as "$y \leftarrow\ y\ +\ A\ x$", which does not make substantial difference, in our discussion.

In the following, we discuss the basic algorithms, in some cases with alternative listings, and point out differences and potential shortcomings.

When no particular ordering of elements is present in the input arrays, an *SpMV* algorithm for *COO* can not make any assumption on subsequent nonzero positions. For this reason, the listing in Fig. 1.5 performs two index reads, one floating point number load[8], and one vector update (of course, besides floating point *multiply* and *add* operations) for each encountered nonzero.

Figure 1.5: *SpMV* listing for *COO*.

```
for $l \leftarrow 1$ to $nnz$ do
    $i \leftarrow IA(l)$
    $j \leftarrow JA(l)$
    $y(i) \leftarrow y(i) + VA(l)x(j)$
end
```

Performing the *transposed SpMV*, defined as "$y \leftarrow\ y\ +\ A^T\ x$" (we will refer to as *SpMV-T*) is equivalent to performing *SpMV* with *JA* and *IA* arrays swapped. See listing in Fig. 1.6.

Figure 1.6: *SpMV-T* listing for *COO*.

```
for $l \leftarrow 1$ to $nnz$ do
    $i \leftarrow IA(l)$
    $j \leftarrow JA(l)$
    $y(j) \leftarrow y(j) + VA(l)x(i)$
end
```

Optimization of such an algorithm's code is difficult to achieve: explicit data reuse is impossible, because indices and coefficients arrays are only used once,

[8]We use terms *load* and *read* interchangeably, and do similarly for *store* and *write*.

since a single matrix pass is performed. Therefore, using some *explicit prefetch* instruction would only work for the aforementioned arrays, whose entries are being used only once. The right-hand side[9]and result arrays, since they are accessed at locations which depend on the current values of $IA(l)$ and $JA(l)$, are potential *cache polluters*, too, since nothing could be said about their reuse. The only relevant possible optimization for this loop is explicit *unrolling*; see Fig. 1.7 for an example of *loop unrolled SpMV* listing.

Figure 1.7: *SpMV* listing for *COO*, unrolled once.

```
$l \leftarrow 1$
while $l + 1 \leq nnz$ do
    $i_0 \leftarrow IA(l+0)$; $i_1 \leftarrow IA(l+1)$
    $j_0 \leftarrow JA(l+0)$; $j_1 \leftarrow JA(l+1)$
    $y(i_0) \leftarrow y(i_0) - VA(l+0)x(j_0)$
    $y(i_1) \leftarrow y(i_1) - VA(l+1)x(j_1)$
    $l \leftarrow l + 2$
end
if $l \leq nnz$ then $y(i) \leftarrow y(i) + VA(l)x(JA(l))$
```

Such an unrolled variation of code would effectively halve the count of loop-control instructions executed at runtime, compared to the computational instructions. Since in practice the number of nonzeroes is quite high, unrolling as much as a dozen of times may seem an attractive option to smash the loop control instructions impact. Another beneficial effect is the slightly reduced impact on branch prediction hardware. In practice, on the architecture we have taken in consideration, phenomena of *register spilling* may occur. That is, while handling more variables than available registers, the compiler may produce code which moves data back and forth from and to the memory, thus generating unnecessary traffic. For this reason, excessive loop unrolling shall be avoided.

If the *COO* input is known to be ordered by rows, a more effective code could be developed, as listed in Fig. 1.8.

Here, an *accumulator* variable *acc* is used and at most one write instruction per row is issued. Most real world matrices feature more nonzeroes than rows, so such a formulation reduces the count of memory writes, when compared to

[9]Informally, the symbolic object at the right of the matrix symbol, in an expression; here, we mean the x vector (or its corresponding *array*, if thinking in terms of memory areas rather than in terms of mathematical objects) in "$y \leftarrow y + A\,x$".

Figure 1.8: *SpMV* listing for *COR*.

```
$l \leftarrow 1$
while $l \leq nnz$ do
    $i \leftarrow IA(l)$
    $acc \leftarrow 0.0$
    repeat
        $j \leftarrow JA(l)$
        $acc \leftarrow acc + VA(l)x(j)$
        $l \leftarrow l + 1$
    until $l > nnz$ or $IA(l) > i$
    $y(i) \leftarrow y(i) + acc$
end
```

the unordered *COO* variant[10]. We observe that the amount of memory which is being read is the same; but as observed before, accessing contiguous elements is much cheaper. So here, the elements in the y array are updated in sequence, and within each line, x is accessed at monotonically increasing locations. Let's ponder some corner case such a code may face. We notice that the inner loop condition depends on two checks: a first one, checking whether the updated nonzero index l is still within boundaries (nnz); a second one, checking whether the row i has changed. Since the first check is likely to succeed nnz times, the second check will be performed each time, too. In the past such a situation would have been undesirable, but nowadays the cost of an integer comparison is irrelevant to the cost of loading from memory and processing some dozen of bytes[11]. If absolutely necessary, one could remove the inner "$l > nnz$" comparison and forcing the routine users to put an extra *marker* numerical index at $IA(nnz+1)$. This would allow the inner loop to terminate properly while performing a single comparison per iteration, and since the removed check is actually guarding the outer loop, no further modification would be necessary. However, in the context of code reuse and modularity, resorting to this solution would be not desirable, because it would break common usage habits, and introduce incompatibility to a vast amount of existing code. Without a guarantee of no *empty row occurrence*, there is no simple way for avoiding the double check in the inner loop (without

[10]However, note that this is not true for arbitrary *submatrices* of common matrices.

[11]Here, we would rather be concerned with the cost of incorrect branch predictions, or the memory load latency, as we will discuss in §1.4.

extra, precomputed information, of course).

The important thing to keep in mind with this formulation is that sequences of very sparsely populated (or empty) rows (even with an overall high *ratio of nonzeroes to rows*) would make the inner loop iterate over very few (or no) elements, losing the advantage of using the *acc* accumulator variable with the $y(i) \leftarrow y(i) + acc$ update at the end of the outer loop.

We will not discuss more variations of this listing, but point out only that knowledge of the matrix for a given problem may be helpful in the optimization of such a seemingly simple computational kernel.

For transposed *SpMV* on *COR* (see Fig. 1.9), the x and y vectors are accessed in transposed order; the update of y elements is not anymore sequential, and it cannot be reduced to one per loop only; on the other hand, the multiplicand element $x(i)$ remains the same through the entire row sweep, so it could be cached with profit with an *accumulator* variable x_i.

Figure 1.9: *SpMV-T* listing for *COR*.

```
$l \leftarrow 1$
while $l \leq nnz$ do
    $i \leftarrow IA(l)$
    $i_0 \leftarrow i$
    $x_i \leftarrow x(i)$
    repeat
        $j \leftarrow JA(l)$
        $y(j) \leftarrow y(j) + VA(l)x_i$
        $l \leftarrow l + 1$
        $i \leftarrow IA(l)$
    until $l > nnz$ or $i > i_0$
end
```

Summing up, while the listing in Fig. 1.8 performs m sequential, cacheable writes and *nnz* unpredictable (or rather, indirect) memory accesses (in a k-sized array–the right-hand side vector), listing in Fig. 1.9 performs *nnz* unpredictable writes (in an m-sized array–the result vector) and m sequential, cacheable reads. Since memory store operations are usually more costly than load ones, *SpMV-T* for *COR* performs usually slower than *SpMV*, especially if $nnz \gg m$.

Listings for *COC* ordering are analogous to the aforementioned, with a simple input modification: namely, the algorithm of *SpMV* for *COR* would compute

$$L = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0.10004 & 0 & 0 & 0 & 0 \\ 0.12219 & 0 & 0 & 0 & 0 \\ 0.05002 & 0.10004 & 0 & 0 & 0 \\ 0.06109 & 0 & 0.12219 & 0.15006 & 0 \end{pmatrix}$$
$$D = \begin{pmatrix} 0.66667 & 0 & 0 & 0 & 0 \\ 0 & 0.53341 & 0 & 0 & 0 \\ 0 & 0 & 0.5777 & 0 & 0 \\ 0 & 0 & 0 & 0.28331 & 0 \\ 0 & 0 & 0 & 0 & 0.27224 \end{pmatrix}$$
$$U = \begin{pmatrix} 0 & 0.36656 & 0.30011 & 0.36656 & 0.30011 \\ 0 & 0 & -1 & 0.20007 & 0 \\ 0 & 0 & 0 & 0 & 0.24437 \\ 0 & 0 & 0 & 0 & 0.18328 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Figure 1.10: Strictly lower triangle L, strictly upper triangle U, and diagonal D of matrix *cage3**. For our convenience, when discussing symmetric matrix representation or the triangular solve operation, we will use L and U for denoting the non-strictly lower and upper triangles (that is, triangles comprehending the diagonal D).

SpMV-T, if called with *IA* and *JA* *COC* arrays swapped; the algorithm of *SpMV-T* for *COR* would compute *SpMV*, if called with the *COC* arrays swapped.

1.1.2 *SpMV* for *COO*/*COR*/*COC*, Symmetric

Coordinate format variants could handle the *symmetric matrix*-vector multiply operation as well. By definition of a symmetric matrix A, it holds $A = A^T$. This means that the *strictly lower* triangle L equals the transposed *strictly upper* triangle U^T.

Since representing explicitly $U = L^T$ would be redundant, only the (non strictly) lower triangle is stored. The *SpMV* algorithm is modified accordingly, in a way to perform an additional, *transposed update*, for each *non diagonal* matrix element.

For (the unordered) *COO*, a symmetric *SpMV* algorithm performs for each nonzero element $a_{i,j} \overset{def}{=} a_{j,i}, i \neq j$, both "$y_i \leftarrow\ y_i + a_{ij}\, x_j$" and "$y_j \leftarrow\ y_j + a_{ji}\, x_i$". The algorithm is listed in Fig. 1.11.

Figure 1.11: *SpMV* listing for *COO*, symmetric.

```
for $l \leftarrow 1$ to $nnz$ do
    $i \leftarrow IA(l)$
    $j \leftarrow JA(l)$
    $y(i) \leftarrow y(i) + VA(l)x(j)$
    if $i \neq j$ then  $y(j) \leftarrow y(j) + VA(l)x(i)$
end
```

Since $A = A^T$, we have $y + Ax = y + A^T x$, and so a symmetric *SpMV-T* reverts to the Fig. 1.11 algorithm (for both *COR* and *COC*).

By combining the use of auxiliary variables, just as in Fig. 1.8, the listing in Fig. 1.11 may be modified to save some array write. Also similar considerations pertain to individual pointer arithmetics and load/store count optimizations. Notice that here, the *nnz* unpredictable (but potentially cacheable) writes are unavoidable.

A last consideration should be made for the case of a "$y \leftarrow \ \beta\ y\ +\ \alpha\ A\ x$" implementation. In the case when y vector scaling is needed ($\beta \neq 1$), each of y's elements should be scaled by β exactly once. By inspecting listings in Fig. 1.11, Fig. 1.9, Fig. 1.8, Fig. 1.6, Fig. 1.5, it is clear that there is no simple way for this update, unless prepending a loop for scaling y ahead of each listing. A valid alternative for *COR*, would be to reformulate the inner loops to cycle over all of the matrix rows, and scaling the y vector regardless of the possible presence of empty rows. This formulation would allow keeping exactly one y array update per row.

1.1.3 *SpSV* for *COR*

With *SpSV*, we either refer to the operation "$x \leftarrow \ L^{-1}\ x$" (*forward substitution*) or "$x \leftarrow \ U^{-1}\ x$" (*backward elimination*). In either the case, performing *SpSV* on unordered *COO* would be extremely inefficient, because of the dependencies posed by the operation; a row or column structure is needed for efficient variable substitutions.

For *SpSV* on the *COR* representation of a lower triangle, see listing in Fig. 1.12. For the upper triangle version, see listing in Fig. 1.13.

Both formulations of *SpSV* access elements of the matrix coefficients *VA* in linear monotonic order (ascending in the case of Fig. 1.12; descending in the case of Fig. 1.13). In both cases, x is accessed repeatedly during substitution at

Figure 1.12: *SpSV* listing for *COR*, lower triangle, left-looking variant.

```
$l \leftarrow 1$
for $i \leftarrow 1$ to $m$ do
    while $JA(l) < i$ do
        $j \leftarrow JA(l)$
        $x(i) \leftarrow x(i) - x(j)\,VA(l)$
        $l \leftarrow l + 1$
    end
    $x(i) \leftarrow x(i) / VA(l)$
    $l \leftarrow l + 1$
end
```

elements determined by the matrix nonzeroes pattern. If a variable is used for substitutions, no more than one vector write per row is performed.

For the transposed or *COC* variations of these algorithms, please see the next section (*CSR*/*CSC* versions differ only slightly from *COR*/*COC* formulations, indeed).

Figure 1.13: *SpSV* listing for *COR*, upper triangle, left-looking variant.

```
$l \leftarrow nnz$
for $i \leftarrow m$ down to $1$ do
    $d \leftarrow VA(l)$
    $l \leftarrow l - 1$
    while $IA(l) = i$ do
        $j \leftarrow JA(l)$
        $x(i) \leftarrow x(i) - x(j)\,VA(l)$
        $l \leftarrow l - 1$
    end
    $x(i) \leftarrow x(i) / d$
end
```

$$A = \begin{pmatrix} 0.66667^{(1)} & 0.36656^{(2)} & 0.30011^{(3)} & 0.36656^{(4)} & 0.30011^{(5)} \\ 0.10004^{(6)} & 0.53341^{(7)} & -1^{(8)} & 0.20007^{(9)} & 0 \\ 0.12219^{(10)} & 0 & 0.5777^{(11)} & 0 & 0.24437^{(12)} \\ 0.05002^{(13)} & 0.10004^{(14)} & 0 & 0.28331^{(15)} & 0.18328^{(16)} \\ 0.06109^{(17)} & 0 & 0.12219^{(18)} & 0.15006^{(19)} & 0.27224^{(20)} \end{pmatrix}$$

PA=(1 6 10 13 17 21)
JA=(1 2 3 4 5 1 2 3 4 1 3 5 1 2 4 5 1 3 4 5)
VA=(0.66667 0.36656 0.30011 0.36656 0.30011 0.10004 0.53341 -1 0.20007 0.12219 0.5777 0.24437 0.05002 0.10004 0.28331 0.18328 0.06109 0.12219 0.15006 0.27224)

Figure 1.14: *CSR* representation of matrix *cage3**. In the following, when it will be clear from the context that we are dealing with *CSR*, we will call *PA* simply *IA*.

1.2 Compressed Sparse Stripes

In this section, we present pseudocode for the classical (see Carney et al. [CHL+96, § 3.3.3], Barrett et al. [BBC+94, § 4.3.1]) *Compressed Sparse Rows* (*CSR*) and *Compressed Sparse Columns* (*CSC*) ([CHL+96, § 3.3.2], [BBC+94, § 4.3.1]) formats and outline the algorithms for performing the most common matrix operations on them. We regard the *CSR* and *CSC* as *Compressed Sparse Stripes* formats.

CSR stores data in three arrays: *VA*,*JA*,*PA*. Nonzero elements are laid out on consecutive rows; rows information is compressed by means of the *rows pointers* array *PA*. Therefore in *CSR*, numerical values (*VA*) and their column indices (*JA*) are stored in the same exact way as in *COR*, and contain nnz elements. Similarly in *CSC* numerical values (in *VA*) and their row indices (in the *IA* array) are stored by columns, just as in *COC*.

In *CSR* (*CSC*) there is one *row (column) pointer* entry for each one of the matrix m rows (k columns): the i-th row (column) pointer contains the index of the first nonzero element of row (column) i in the remaining two *VA*,*JA* (*VA*,*IA*) arrays. The row (column) pointers array is sized $m+1$ ($k+1$): the last array element being the first index *after* the last element of the *VA* array (customarily, nnz in C, $nnz+1$ in Fortran). This extra element is used as an *end of loop* delimiter; see algorithm listings in the next sections for its usage.

See representations of *cage3** with *CSR* in Fig. 1.14 and with *CSC* in Fig. 1.15.

$$A = \begin{pmatrix} 0.66667^{(1)} & 0.36656^{(6)} & 0.30011^{(9)} & 0.36656^{(13)} & 0.30011^{(17)} \\ 0.10004^{(2)} & 0.53341^{(7)} & -1^{(10)} & 0.20007^{(14)} & 0 \\ 0.12219^{(3)} & 0 & 0.5777^{(11)} & 0 & 0.24437^{(18)} \\ 0.05002^{(4)} & 0.10004^{(8)} & 0 & 0.28331^{(15)} & 0.18328^{(19)} \\ 0.06109^{(5)} & 0 & 0.12219^{(12)} & 0.15006^{(16)} & 0.27224^{(20)} \end{pmatrix}$$

IA=(1 1 1 1 1 2 2 2 2 3 3 3 4 4 4 4 5 5 5 5)
PA=(1 6 9 13 17 21)
VA=(0.66667 0.36656 0.30011 0.36656 0.30011 0.10004 0.53341 -1 0.20007 0.12219 0.5777 0.24437 0.05002 0.10004 0.28331 0.18328 0.06109 0.12219 0.15006 0.27224)

Figure 1.15: *CSC* representation of matrix *cage3**. In the following, when it will be clear from the context that we are dealing with *CSC*, we will call *PA* simply *JA*.

1.2.1 *SpMV* for *CSR*/*CSC*

The *SpMV* algorithm for *CSR* (Fig. 1.16) works by traversing the matrix by rows. The memory access pattern for arrays *VA* and *JA* is the same as that of *COR* (see Fig. 1.8). As with *COR*, it is possible to arrange the code for exactly m updates to the y array (one per row visit). To achieve this it is sufficient to modify the Fig. 1.16 listing in a way to move the y update to the outer loop, right after the inner loop. In the inner loop, an accumulator variable shall be used instead (and it is likely that the compiler will rearrange it to reside in a register). The accumulator variable shall be re-initialized to zero at the beginning of each row, i.e.: before each inner loop.

Figure 1.16: *SpMV* listing for *CSR*.

```
for $i \leftarrow 1$ to $m$ do
    for $l \leftarrow PA(i)$ to $PA(i+1)-1$ do
        $j \leftarrow JA(l)$
        $y(i) \leftarrow y(i) + VA(l)x(j)$
    end
end
```

With *CSC* (see algorithm in Fig. 1.17), the access pattern to arrays *IA* and *VA* is the same as that of *COC* (see Fig. 1.8): y vector elements are updated for each nonzero entry.

Note that for a "$y \leftarrow \ \beta\ y\ +\ \alpha\ A\ x$" update on *CSR*, a statement like $y_i \leftarrow \beta y_i$ could be issued at the beginning of each row i, in Fig. 1.16, whether row i is empty or not. Contrary to *CSR*, listings for *CSC* cycle on columns, so there is no direct way to scale y using exactly one operation per row, in the outer loop. The easiest alternative for handling this case would therefore be adding an extra y vector update loop at the beginning of Fig. 1.17 listing. The obvious performance consideration of this update concerns the y vector, which is likely to be cached during access, but with no guarantee of reuse.

Figure 1.17: *SpMV* listing for *CSC*.

```
for $j \leftarrow 1$ to $k$ do
    for $l \leftarrow PA(j)$ to $PA(j+1)-1$ do
        $i \leftarrow IA(l)$
        $y(i) \leftarrow y(i) + VA(l)x(j)$
    end
end
```

Since it is often the case that memory write is slower than read, for a given matrix, *CSC SpMV* could be slightly slower than *CSR SpMV*, especially if $nnz \gg k$ (which is almost the norm, for most practical applications). For the same reason, *SpMV-T* for *CSC* would be faster than for *CSR*.

Figure 1.18: *SpMV-T* listing for *CSR*.

```
for $i \leftarrow 1$ to $m$ do
    for $l \leftarrow PA(i)$ to $PA(i+1)-1$ do
        $j \leftarrow JA(l)$
        $y(j) \leftarrow y(j) + VA(l)x(i)$
    end
end
```

In Fig. 1.18 and Fig. 1.19 we list code for the computation of transposed *SpMV* for *CSR* and *CSC*. Notice that here, the implementation of a $y \leftarrow \beta y$ scaling could be possible in a single pass only for *CSC*, as in the transposed case, y is k-sized, and our listing loop steps in each column exactly once. The transposed *CSR* version could scale the y vector only by means of an outer loop

Figure 1.19: *SpMV-T* listing for *CSC*.

```
for $j \leftarrow 1$ to $k$ do
    for $l \leftarrow PA(j)$ to $PA(j+1)-1$ do
        $i \leftarrow IA(l)$
        $y(j) \leftarrow y(j) + VA(l)x(i)$
    end
end
```

on the k entries of y.

After all, the semantics of the *SpMV-T* update on *CSR* is that of the *SpMV* update for *CSC*, after the appropriate swap of m, k and aliasing the *JA* array as *CSC*'s *IA*.

1.2.2 *SpMV* for *CSR*/*CSC*, Symmetric, and Variants

A notable variant which is interesting to handle is that of *symmetric* matrices. With *CSR* and *CSC*, symmetric matrices are handled in the same manner to *COR*/*COC* (as seen in §1.1.2), that is by avoiding redundant storage, and by using specialized, *symmetric kernels*. Since $A = A^T$, then $L = U^T$ and $(L+D) = (U+D)^T$, so the specialized kernel computes $y \leftarrow y + Ax$ as $y \leftarrow y + (L + D + L^T)x$. See Fig. 1.20 for a listing capable of handling a symmetric matrix stored as either lower or upper triangle.

In the case of *CSC*, exactly the same code would be used, since symmetric matrices are square ($m = k$) and the update is symmetric also. There is one complication affecting the result vector scaling version of this code, in the case of an upper stored matrix. Namely, scaling of the y vector would not possible in the external loop, because the upper storage would lead into updating rows before scaling them. For this reason, to use a y-scaling version of this code on an upper stored matrix, one should reverse the order of rows visit (from the last one to the first one), or simply add an extra outer scaling loop (which would be costly, of course).

Another notable implementation variant to be handled would be that for a *diagonal implicit* representation. In that case, handling an empty row i would still push the need for the single, unsymmetric contribution of $a_{ii}x_i$. Usually, an implicit a_{ii} is assumed to be unitary, therefore its contribution can be computed by the summation of the entire x vector to y. See the adapted listing in Fig. 1.21 for this. Notice also that an accumulator variable could be used for caching

Figure 1.20: *SpMV* listing for *CSR*, lower/upper triangle, symmetric.

```
for $i \leftarrow 1$ to $m$ do
    if $PA(i) = PA(i+1)$ then continue
    $j \leftarrow JA(PA(i))$
    $y(i) \leftarrow y(i) + VA(PA(i))x(j)$
    if $j \neq i$ then
        $y(j) \leftarrow y(j) + VA(PA(i))x(i)$
    end
    for $l \leftarrow PA(i) + 1$ to $PA(i+1) - 2$ do
        $j \leftarrow JA(l)$
        $y(i) \leftarrow y(i) + VA(l)x(j)$
        $y(j) \leftarrow y(j) + VA(l)x(i)$
    end
    if $PA(i+1) = PA(i) + 1$ then continue
    $j \leftarrow JA(PA(i+1) - 1)$
    $y(i) \leftarrow y(i) + VA(PA(i+1) - 1)x(j)$
    if $i \neq j$ then
        $y(j) \leftarrow y(j) + VA(PA(i+1) - 1)x(i)$
    end
end
```

partial results pertaining to the current row (or column, in the case of *CSC*), thus reducing the number of necessary memory writes from *nnz* random ones, to m sequential, monotonically increasing ones.

Figure 1.21: *SpMV* listing for *CSR*, lower triangle, symmetric, diagonal implicit.

```
for $i \leftarrow 1$ to $m$ do
    $y(i) \leftarrow y(i) + x(i)$
    if $PA(i) = PA(i+1)$ then continue
    for $l \leftarrow PA(i)$ to $PA(i+1) - 1$ do
        $j \leftarrow JA(l)$
        $y(i) \leftarrow y(i) + VA(l)x(j)$
        $y(j) \leftarrow y(j) + VA(l)x(i)$
    end
end
```

In all of the considered listings, the most profitable optimization would be that of explicit inner *loop unrolling*. In the case of highly populated rows, this optimization would lead to a more effective usage of registers and lessen branch mispredictions impact (because of the reduced number of loops performed). See the example in Fig. 1.22 for a basic loop unrolled version of *SpMV* for *CSR*, and notice the two inner loops: the first one proceeding three nonempty columns at a time; the second processing the remaining one or two non-empty columns. It is clear that an input matrix having many loosely populated rows (say with up to three nonzeroes per row) would not take advantage from the first unrolled loop, because with less than three nonzeroes, the loop will not be entered, thus resulting in a wasted comparison operation, and possible penalties due to the repeated failed branch prediction. On the other hand, on a matrix with more than three average nonzeroes per row (the most common case), the total number of comparisons performed in the first inner loop can be reduced asymptotically (in the number of the sparse row nonzeroes) by two thirds of the original. In the listing, we have also used three different accumulator variables (y_0, y_1, y_2): in most programming languages, using a single accumulator variable three times would introduce an undesired *data dependency*. Using three accumulators instead allows the compiler to take advantage of possible low level parallelization strategies (like *vector extensions* of existing or forthcoming CPUs) for their update. Finally, as we have seen before, the use of accumulators allows us to update the memory location for $y(i)$ only once per row.

As an alternative to the inner loop unrolling, it would be possible to unroll the external loop, but the extent of possible efficiency gain, here, would be probably less.

Figure 1.22: *SpMV* listing for *CSR*, loop unrolled.

```
for $i \leftarrow 1$ to $m$ do
    $y_0 \leftarrow 0$; $y_1 \leftarrow 0$; $y_2 \leftarrow 0$ $l \leftarrow 0$
    for $l \leftarrow PA(i)$ to $PA(i+1)-3$ incrementing by 3 do
        $j_0 \leftarrow JA(l+0)$
        $j_1 \leftarrow JA(l+1)$
        $j_2 \leftarrow JA(l+2)$
        $y_0 \leftarrow y_0 + VA(l+0)x(j_0)$
        $y_1 \leftarrow y_1 + VA(l+1)x(j_1)$
        $y_2 \leftarrow y_2 + VA(l+2)x(j_2)$
    end
    for $l \leftarrow l$ to $PA(i+1)-1$ do
        $j \leftarrow JA(l)$
        $y_0 \leftarrow y_0 + VA(l)x(j)$
    end
    $y(i) \leftarrow y(i) + y_0 + y_1 + y_2$
end
```

As we see, just as with coordinate formats, detailed knowledge of the input could come in help when thinking of an optimized execution for compressed stripes-based formats, too.

1.2.3 *SpSV* for *CSR*/*CSC*

Triangular solution is the essential operation in the implementation of many *preconditioning* techniques in the solution of linear systems. In that context, it often happens that a matrix representing a triangle, as part of a factorization of another matrix, has a *unitary diagonal.* We can take advantage of this, and save a few bits of computation with an ad-hoc kernel code. We list pseudocode for the solution of lower triangular systems in *CSR* with diagonal explicit and implicit, respectively in Fig. 1.23 and Fig. 1.24.

The inner loop cycles on column indices, which for each given i, are strictly less than it (we have a lower triangle). We notice that for the inner loop, $x(i)$

Figure 1.23: *SpSV* listing for *CSR*, diagonal explicit.

```
$x(1) \leftarrow y(1)/VA(1)$
**for** $i \leftarrow 2$ **to** $m$ **do**
    $x(i) \leftarrow 0$
    **for** $l \leftarrow PA(i)$ **to** $PA(i+1)-2$ **do**
        $j \leftarrow JA(l)$
        $x(i) \leftarrow x(i) + VA(l)x(j)$
    **end**
    $x(i) \leftarrow (y(i) - x(i))/VA(l)$
**end**
```

Figure 1.24: *SpSV* listing for *CSR*, diagonal implicit.

```
$x(1) \leftarrow y(1)$
**for** $i \leftarrow 2$ **to** $m$ **do**
    $x(i) \leftarrow 0$
    **for** $l \leftarrow PA(i)$ **to** $PA(i+1)-1$ **do**
        $j \leftarrow JA(l)$
        $x(i) \leftarrow x(i) + VA(l)x(j)$
    **end**
    $x(i) \leftarrow (y(i) - x(i))$
**end**
```

could be stored in a variable declared in the outer loop, and therefore likely to be cached in some register by the compiler. If using explicitly a separate accumulator variable in the loop and $x(i)$ initialization, the algorithm could work on a single vector y in place, with no additional penalty. We also observe that if doing so, the y vector would have higher chances of cache reuse, given the inner loop potential re-runs on the whole vectors. Reuse would be particularly effective on matrices with a strongly triangular pattern (as opposed to a *banded* pattern).

Figure 1.25: *SpSV-T* listing for *CSR*, diagonal explicit.

```
for $i \leftarrow m$ down to 1 do
    $l \leftarrow PA(i+1) - 1$
    $y(i) \leftarrow y(i)/VA(l)$
    for $l \leftarrow PA(i+1) - 2$ down to $PA(i)$ do
        $j \leftarrow JA(l)$
        $y(j) \leftarrow y(j) - VA(l)y(i)$
    end
end
```

In Fig. 1.25, we see a formulation of transposed *SpSV* for *CSR*. With some abuse of notation, we mean both the loops to iterate on descending indices, both ends inclusive. As we had with *SpMV*, here the inner loop is capable of issuing *nnz* writes to unpredictable memory locations (that is, locations are accessed indirectly), with a moderate degree of locality. Notice also that while the external loop is required to cycle *backwards*, the inner one is not required to do so; however, in this way we allow for some extra memory locality after the diagonal element access (the first instruction in the external loop).

Some extra efficiency may be achieved for the *SpSV* kernels by unrolling the inner loop. As it is the case with loop unrolling, it will be effective if the average loop interval is relevant; namely, if many of the rows are populated *more* than the unroll factor.

Notice also that in all of the *diagonal explicit SpSV* kernels we have presented, subarrays corresponding to rows (in *CSR*) or columns (in *CSC*) assume that the last element is the one with the higher index; that is, the element on the diagonal. The presented *diagonal implicit SpSV* kernels, instead, relax even this constrain and allow unsorted subarrays.

$$A = \begin{pmatrix} 0.66667^{(1)} & 0.36656^{(2)} & 0.30011^{(3)} & 0.36656^{(4)} & 0.30011^{(5)} \\ 0.10004^{(9)} & 0.53341^{(8)} & -1^{(7)} & 0.20007^{(6)} & 0 \\ 0.12219^{(10)} & 0 & 0.5777^{(11)} & 0 & 0.24437^{(12)} \\ 0.05002^{(16)} & 0.10004^{(15)} & 0 & 0.28331^{(14)} & 0.18328^{(13)} \\ 0.06109^{(17)} & 0 & 0.12219^{(18)} & 0.15006^{(19)} & 0.27224^{(20)} \end{pmatrix}$$

IA_{ZZ-CSR}=(1 6 10 13 17 21)

JA_{ZZ-CSR}=(1 2 3 4 5 4 3 2 1 1 3 5 5 4 2 1 1 3 4 5)

VA_{ZZ-CSR}=(0.66667 0.36656 0.30011 0.36656 0.30011 0.20007 -1 0.53341 0.10004 0.12219 0.5777 0.24437 0.18328 0.28331 0.10004 0.05002 0.06109 0.12219 0.15006 0.27224)

Figure 1.26: *Zig-Zag CSR* representation of matrix *cage3**. Notice how the order elements are laid out: at the end of each odd row, ordering proceeds by traversing even rows backwards.

1.2.4 Two Variations: *Zig-Zag CSR* and *BCSR*

In this section, we show two variations of *CSR* documented in the literature. The first one is a rather modest modification which enables the reuse of the *SpMV*/*SpMV-T* algorithms (as presented in Fig. 1.16,1.18) unmodified, while achieving greater cache reuse. The second one is a completely different format, still based on the idea of compressing rows; to be precise, *rows of small, dense blocks*. Although the ideas behind these two formats could be combined (they are quite independent), we discuss them separately, to point out some interesting facts.

The first modification to *CSR* is named *Zig-Zag CSR*, as introduced by Yzelman and Bisseling [YB09, § 5]. The rationale for *Zig-Zag CSR* is extremely simple: given the basic *SpMV* matrix sweep of *CSR* arrays (see Fig. 1.16), we notice that towards the end of the first row ($i = 1$) traversal, right-hand side vector (y) locations *around* column index j are likely to be cached. When stepping in the inner loop on row $i + 1 = 2$, these cache lines, if unmodified, would have a big chance for being used again. In practice, though, when proceeding with a new row, it is likely that before an eventual reuse (towards the end of this new row), cache lines of the preceding row may be *evicted*. The way *Zig-Zag CSR* may enhance locality here is simple: by reversing the order of row 2 elements (that is, *JA* and *VA* entries at locations $[PA(2)...PA(3) - 1]$), and elements of all *even* rows $4, 6, ..,$ any *SpMV* algorithm for *CSR* is likely to work with improved cache locality of the y array. The pattern of access for *VA* and *JA* arrays would be the same: monotonically ascending. Access to the right-hand side vector would be unpredictable as usual, as mandated by the use of indices in *JA*; however, even if alternating the direction of consecutive y requests (on each new row) may trick the prefetch hardware, the chance of reuse is increased, since it is likely to have in cache some locations of *nearby indices* of last used y entries. Matrix *cage3** in the *Zig-Zag CSR* layout is shown in Fig. 1.2.4. Notice also that the *SpSV* algorithm for *CSR* could be adapted to *Zig-Zag CSR* with very modest changes. The basic (and reasonable) assumption of *Zig-Zag CSR* is that if some nonzero entry exists at $a_{i,j}$, it is often the case that some other nonzero exists at locations $a_{i\pm1,j+\Delta}$, with Δ a small integer (negative or positive) number.

The second "variant" of *CSR* we present, is called *BCSR* (Blocked *CSR*), and is indeed very well documented in the literature, and used in many implementations; prominently, it is the main format of the OSKI (see Vuduc et al. [VDY05a]) package; see also the related research by Im and Vuduc, e.g.: [Im00], [Vud03]. The *BCSR* format applies the same compression idea of *CSR*, but to *dense blocks of a fixed size*, $br \times bc$, rather than to *individual nonzeroes*. With *BCSR*, the pointer array is of *shorter* length: rather than having $m+1$ entries, it has $\lceil m/br \rceil + 1$ entries, pointing to *block rows* bi. Also the indices array is shorter: instead of having one column index per nonzero, it has *no less than* (see the following discussion) $\lceil nnz/(br \cdot bc) \rceil$ *block column* indices bj. The routine for *SpMV*, here, is similar to that for *CSR*, but handles entire $br \times bc$ block elements instead of individual entries. This ensures ready reuse of both result and right-hand side vector arrays, as here the minimal update would not be $y_i \leftarrow y_i + a_{i,j}x_j$, but rather (with some abuse of notation): $y_{bi\cdot br:bi\cdot br+br-1} \leftarrow y_{bi\cdot br:bi\cdot br+br-1} + a_{bi\cdot br:bi\cdot br+br-1,bj\cdot bc:bj\cdot bc+bc-1}x_{bj\cdot bc:bj\cdot bc+bc-1}$[12].

A consequence of this is that in order to apply a given choice of $br \times bc$ (*blocking*) to a matrix, it may happen for some zeroes to fall under some dense block. Think of an $n \times n$ matrix, whose upper half is populated by groups of horizontally pairwise adjacent nonzeroes, while the lower half would have all nonzeroes isolated. Clearly, a 1×2 blocking would be appropriate in the representation of the upper half matrix. However, applying this blocking to the lower half would require the allocation of blocks which would convey only half *useful* numerical data (that is, *effective* nonzeroes). That is, the existence of these extra, *fill-in* nonzeroes does not change the numerical results of most computations, but introduces unnecessary operations and the need for a larger

[12]If we interpret the bi and bj indices as offsets in the *unblocked* matrix (rather than in *block coordinates*), we allow *unaligned* blocks, and therefore have a different update: $y_{bi:bi+br-1} \leftarrow y_{bi:bi+br-1} + a_{bi:bi+br-1,bj:bj+bc-1}x_{bj:bj+bc-1}$. For more research about this variant (called *UBCSR*), see Vuduc [Vud03, 5.1] or Buttari et al. [BELF07].

VA array.

For instance, with our (artificial) example matrix *cage3** (recall Fig. 1.1), there exists no blocking for *BCSR* that would prevent from storing all of its zeroes as *fill-in* entries.

The potential performance benefits of *BCSR* are well documented; *BCSR* may greatly speed up the *SpMV*. However, the needs for avoiding *fill-in* and guessing *a good blocking*[13] (or even *split* the matrix in multiple overlays with differing blocking)[14] is a substantial difficulty towards the optimal usage of *BCSR*. The fact that some matrices do not even have a block structure at all, further restricts *BCSR* from general applicability.

Given the presence of *fill-in*, we distinguish here *raw performance* from *effective performance*: the first one taking in consideration the rate at which *any* floating point instructions are performed; the second one considering only the number of floating point instructions which actually contribute to the numerical result, i.e.: the count of instructions not involving the zeroes. See the works of Buttari et al. ([BELF07]) and PhD thesis ([But06]) of Buttari for a methodology for the modeling and optimization of *BCSR* performance. In this work we will not discuss these techniques in further details.

1.3 Overview of Other Operations

So far, we have described and discussed with some detail possible implementations of the two core **Sparse BLAS** operations: *SpMV* and *SpSV*. Maximizing the efficiency of these computational kernels is the primary problem we address in our research. In a broader context, however, an audience of designers, implementors, or users of sparse matrix techniques may take in consideration also the availability (and/or the efficiency) of algorithms for other operations.

The very first algorithm to be implemented for any given matrix layout is the one for building (or *assembling*, in jargon) the matrix data structure in memory. One or more routines for efficient matrix *assembly* is desirable: a slow procedure would only be acceptable when undeniable benefits could be achieved by subsequent high speed computations. For instance, if the matrix is going to be used once, it may be the case that *COO* (that is, a mere list of matrix nonzeroes) will be the proper representation, at least for the simplest

[13]Excessive *fill-in* resulting from a poor blocking choice may push effective performance too low, since a lot of extra *bogus operations* summing zeroes would be performed.

[14]In general, finding an optimal *tiling* of a sparse matrix is a difficult problem; see Pinar and Vassilevska in [PV05] or [VP04].

operations. Conversion algorithms/routines from and to the target format are desirable; if two matrix formats require row major ordering, it is likely that interchange between these formats will be efficient; conversely, it is also likely that conversion between a row major-based and a column major-based format will be less efficient.

Besides matrix instantiation algorithms and vector multiplication/solve, some *general purpose* applications like interactive systems for numerical computations (e.g.: **GNU Octave** or **Matlab**) prefer a data structure capable of several operations. This, because such a package is generally unaware of the operations that will be carried out on a sparse matrix at build time, unless explicit hints are given by the user (and this is not currently contemplated in the two mentioned packages).

Some high performance solver libraries, like **PSBLAS**, offer interfaces for *pluggable* custom sparse matrix formats implementations. In the case of **PSBLAS**, a *plugged* format must contemplate much more operations than the two **Sparse BLAS** operations mentioned above. We list and briefly discuss a number of operations a low-level sparse matrix package could support, and their possible application contexts. Knowledge of what operation is possible/optimal and what is not for a given sparse matrix technology exposes the limits and potentials for its application in a given context.

- **random (read/write) access of elements**: Update of boundary conditions in a time-based, evolving simulation.

- **extraction of sorted sparse/dense subblocks (e.g.:rows/columns)**: For instance, computing a *preconditioner matrix*[15], step by step[16].

- **modification of the sparsity pattern**: Needed when the topology of a simulated domain changes.

- **pattern-only representation**: For keeping connectivity information in unweighted graphs.

- **infinity/one norm computation**: Also computable with a *SpMV* or *SpMV-T*; used in iterative methods, for the convergence criteria.

- **rows/columns scaling**: For proper conditioning of a triangular matrix, or many other algorithms.

[15]For a discussion about *preconditioning* techniques, see Saad [Saa03, Ch. 9-10,12].

[16]As the **PSBLAS** package does when handling “opaque” sparse matrix formats.

- **incomplete, *pattern preserving* factorizations**: Typically, preconditioning during the iterative solution of a linear problem.
- **incomplete, *pattern altering* factorizations**: Typically, preconditioning during the iterative solution of a linear problem.
- **complete (direct) factorizations**: In the context of iterative solvers they are typically employed on a reduced subproblem instance used in the preconditioning of the original one.
- **sum/difference of sparse matrices**: When assembling a system of equations.
- **multiplication of sparse matrices**: Sparse multiplications may also arise in the context of building matrices to be used as *multilevel preconditioners*; see the work of D'Ambra, Serafino and Filippone in the MLD2P4 and PSBLAS packages ([DdSF10, p.14]).
- **matrix powers**: For the acceleration of some iterative processes.
- **symmetry handling**: Symmetric matrices arise in many problems, and a specialized handling allows saving memory and computation time (as we recall from §1.2.2).
- **matrix transposition**: The need for an explicit transpose of a sparse matrix may arise in some preconditioner implementations; see for instance MLD2P4 (D'Ambra, Serafino and Filippone [DdSF10]).
- **diagonal extraction**: Diagonal-based preconditioning.
- **matrices pattern intersection**: Graph theoretic applications.
- **any of the previous, with shared memory parallelism**: For speeding up computations.
- **any of the previous, with distributed memory parallelism**: For speeding up computations, or handling problems exceeding the physical memory available on a single computer.

Indeed, seldom most of these operations need to be implemented and (absolutely) optimized in a carefully written, high performance code. However, we would like to stress that what often happens with data structures is that while some operations could be "natural" and "cheap" on one data structure, they may be impractical and inefficient on another one.

1.4 Memory Access Patterns for Basic Sparse Matrix Operations

In §1.1 and §1.2, we have presented two of the most common sparse matrix layouts (coordinate list and compressed stripes), with pseudocode listings and discussion about their possible optimization in practical implementations. In this section, we summarize the discussion in those sections by means of Table 1.1 and some notation; we focus on the memory access patterns for a number of *format* and *operation* combinations. We focus here on the operations which are most relevant to the implementation of iterative solvers. The discussion in the previous sections should be enough to justify all of the expressions present in the table, even if the algorithm listing corresponding to a particular table entry was not presented.

The effective number of floating point operations executed in a given operation may vary slightly between the various formats implementations (and also between different implementations of the same operation and format). For instance, non-transposed, α-scaled *SpMV* (see the table) on row-major *COR* allows arranging the code into using a hardware register for storing the current y (result) vector entry, for the whole inner loop execution. Thus saving $nnz - m$ multiplications into iterating that inner loop, overall.

Such considerations, however, are not of our concern here: on the architectures of our interest, most inefficiencies happen in the form of *stalls* (wasted CPU cycles, caused by an excessive and unnecessary movement of data in the memory hierarchy).

Therefore, in the table, we report only counts of memory writes (or *stores*) and reads (*loads*). Most of the writes occurring here are *accumulating*: either add or multiply-and-add operations; but we do not report the load operation implied by such a store, for simplicity. In the table, we mark memory accesses as being either *sequential* or *random*. *Random* accesses occur because the referred location address is computed using some indices, read from (*pointer* or coordinate) arrays (which are accessed sequentially). In all of the cases we consider, there are three arrays which are accessed once and sequentially (and thus, thanks to prefetch, are likely to cause no cache miss excepts their first access): the nonzeroes (matrix numerical coefficients) array, the nonzeroes column/row indices array, and its row/column pointers array, in the compressed stripes formats. A second attribute we give to *groups* of memory accesses regards the chance of reuse of the locations which are cached during access[17]. So we count

[17]All of these assumptions hold because the architectures we consider have *multi-way* (or

separately accesses which are: *one-shot*, if no reuse is foreseen, and early cache eviction—just after the write/read—would cause no miss; *likely* to fill cache lines with data which would be *reused soon*—for instance, when non consecutive, *nearby*[18] locations are accessed; and finally, accesses pushing to the cache data that it is *very unlikely* to be reused—for instance, when non consecutive, quite *afar* locations are accessed.

In some cases (e.g.: the *SpSV* listing for *CSR*, in Fig. 1.25), certain arrays are accessed proceeding *backwards*; this should not be a problem with the current prefetch engines, so we did not mention this in the table.

In Table 1.2, we give, for each corresponding entry in Table 1.1, the ratio of *indirect* to *direct* memory accesses. This second table could be obtained after a different grouping and counting of the quantities in the first table.

Assumptions we make here about hardware prefetch do not hold if using excessive *stride* in accessing arrays; that is, spacing adjacent vector entries with a fixed number of array entries exceeding the so-called *trigger threshold* would drive prefetch engines into failure. The consequence would be that all accesses to the excessively strided vector would cause a cache miss.
Please consult §B for some experiments quantifying these considerations.

1.5 More Literature and Related Topics

1.5.1 Overview of Other Formats

Historically, a vast number of sparse matrix formats have been developed; sometimes to suit particular applications/algorithms; sometimes to pursue efficiency on particular machines. In this subsection we mention some of them, as described in Barrett et al. ([BBC+94, 4.3.1]).

- **Compressed Diagonal Storage (*CDS*)**: *CDS* is a format which stores *supra* and *sub diagonals* of a sparse matrix, also known as *band*. It is suitable for matrices used in iterative methods. In the case the band contains zeroes in some of the diagonals, *CDS* could end up by storing them. Despite the name, it does not compress diagonals in the way *CSR* does with

group associative) caches, and support *multiple streams* of prefetched sequences. For instance, most of recent Intel processors have up to eight prefetched streams, eight ways of cache associativity, and a *trigger threshold* (maximal distance, in bytes, for regarding two temporally close memory accesses as part of a *stream*, and thus activating the prefetch engine) of about 256 ([Int08a, § 2.4.2]).

[18] Falling into the same cache line, for instance.

operation	*COO*	*COR*	*COC*	*CSR*	*CSC*
$a_{i,j} \leftarrow \kappa$	$O(n)R_{se} + W_{re}$	$\Theta(lg_2(n))R_{se} + W_{re}$	as *COR*	$\Theta(\gamma)R_{se} + W_{re}$	as *CSR*
$d \leftarrow D(A)$	$O(n)R_{re} + mW_{se}$	$\Theta(m)(R_{re} + W_{se})$	as *COR*	$\Theta(m)(R_{re} + W_{se})$	as *CSR*
$y \leftarrow \beta y + \alpha Ax$	$3nR_{se} + nR_{ru}$ $+nW_{ru} + mW_{se}$	$3nR_{se} + nR_{rl}$ $+2mW_{se}$	$3nR_{se} + nR_{se}$ $+nW_{rl} + mW_{se}$	$(2n+m)R_{se} + nR_{rl}$ $+2mW_{se}$	$(2n+k)R_{se} + nR_{se}$ $+nW_{rl} + mW_{se}$
$y \leftarrow y + \alpha Ax$	$3nR_{se} + nR_{ru}$ $+nW_{ru}$	$3nR_{se} + nR_{rl}$ $+mW_{se}$	$3nR_{se} + nR_{se}$ $+nW_{rl}$	$(2n+m)R_{se} + nR_{rl}$ $+mW_{se}$	$(2n+k)R_{se} + nR_{se}$ $+nW_{rl}$
$y \leftarrow y + \alpha A^T x$	$3nR_{se} + nR_{ru}$ $+nW_{ru}$	$3nR_{se} + nR_{se}$ $+nW_{rl}$	$3nR_{se} + nR_{rl}$ $+kW_{se}$	$(2n+m)R_{se} + nR_{se}$ $+nW_{rl}$	$(2n+k)R_{se} + nR_{rl}$ $+kW_{se}$
$y \leftarrow y + \alpha Ax$ $^{(\star)}$	$3nR_{se} + 2nR_{ru}$ $+2nW_{ru}$	$(3n+m)R_{se} + nR_{rl}$ $+mW_{se} + nW_{rl}$	as *COR*	$(2n+2m)R_{se} + nR_{rl}$ $+mW_{se} + nW_{rl}$	as *CSR*
$y \leftarrow \alpha L^{-1} y$	*N.A.*	$3nR_{se} + nR_{rl}$ $+mW_{se}$	$3nR_{se} + nR_{se}$ $+nW_{rl}$	$(2n+m)R_{se} + nR_{rl}$ $+mW_{se}$	$(2n+k)R_{se} + nR_{se}$ $+nW_{rl}$
$y \leftarrow \alpha L^{-T} y$	*N.A.*	$3nR_{se} + nR_{se}$ $+nW_{rl}$	$3nR_{se} + nR_{rl}$ $+kW_{sl}$	$(2n+k)R_{se} + nR_{se}$ $+nW_{rl}$	$(2n+m)R_{se} + nR_{rl}$ $+kW_{sl}$
$\forall i,j\ a_{i,j} \leftarrow x_i a_{i,j}$	$nR_{se} + nR_{rl}$ $+nW_{se}$	$nR_{se} + mR_{se}$ $+nW_{se}$	$nR_{se} + nR_{rl}$ $+nW_{se}$	$mR_{se} + mR_{se}$ $+nW_{se}$	$nR_{se} + nR_{rl}$ $+nW_{se}$

Table 1.1: Overview of the memory access patterns for *COO*, *COC*, *COR*, *CSR*, *CSC*.
Assumptions: the matrix diagonal ($D(A)$) is always explicitly stored; either rows or column are ordered ascendingly (except *COO*, of course); the update operation $a_{i,j} \leftarrow \kappa$ assumes that $a_{i,j}$ is a stored nonzero; either diagonal extraction or the update operation imply a binary search in *COR*/*COC*/*CSR*/*CSC*; *SpMV* for *COO* as shown in Fig. 1.5; *SpMV-T* for *COO* as in Fig. 1.6; *SpMV* for *COR* as in Fig. 1.8; symmetric *SpMV* for *COO* as in Fig. 1.11; *SpMV-T* for *COR* as in Fig. 1.9; *SpMV* for *CSR* as in Fig. 1.16; *SpMV* for *CSC* as in Fig. 1.17; *SpMV-T* for *CSR* as in Fig. 1.18; *SpMV-T* for *CSC* as in Fig. 1.19; *SpSV* is assumed to use a single input/output vector; for all operations, we assume unitary vectors stride. To avoid complicating the above expressions, we assume that: for *CSR*/*COR*, $nnz > m$ and no empty row exists; for *CSC*/*COC*, $nnz > k$ and no empty column exists. With some abuse of notation, with $O(n)$ we denote a quantity which on the average is n; with $\Theta(n)$ we denote a quantity which is n in the *worst case*.

Legend
O_{ar}: a read (load) operation if O is R, a write (store) operation if O is W; accessed sequentially if a is s, randomly (by means of indirection) if a is r; with likely (temporally) element reuse if r is l, unlikely (temporally) element reuse if r is u, no reuse at all if r is e
m: matrix rows
k: matrix columns
n: short for nnz (matrix nonzeroes)
γ: nnz/m if *COR* or *CSR*; nnz/k otherwise
N.A.: Not Applicable ($O(n^2)$ complexity implied)
$\star$: symmetric matrix (only lower representation, but computing the upper triangle contribution also); from the overall writes count, should subtract the count of elements on the diagonal; $m = k$ is implied

operation	*COO*	*COR*	*COC*	*CSR*	*CSC*
$a_{i,j} \leftarrow \kappa$	$\frac{0}{O(n)}$	$\frac{0}{\Theta(lg_2(n))}$	as *COR*	$\frac{0}{\Theta(\gamma)}$	as *CSR*
$d \leftarrow D(A)$	$\frac{O(n)}{m}$	$\frac{\Theta(m)}{\Theta(m)}$	as *COR*	$\frac{\Theta(m)}{\Theta(m)}$	as *CSR*
$y \leftarrow \beta y + \alpha A x$	$\frac{2n}{3n+m}$	$\frac{n}{3n+2m}$	$\frac{n}{4n+m}$	$\frac{n}{2n+3m}$	$\frac{n}{3n+k+m}$
$y \leftarrow y + \alpha A x$	$\frac{2n}{3n}$	$\frac{n}{3n+m}$	$\frac{n}{4n}$	$\frac{n}{2n+2m}$	$\frac{n}{3n+k}$
$y \leftarrow y + \alpha A^T x$	$\frac{2n}{3n}$	$\frac{n}{4n}$	$\frac{n}{3n+k}$	$\frac{n}{3n+m}$	$\frac{n}{2n+2k}$
$y \leftarrow y + \alpha A x\ ^{(\star)}$	$\frac{4n}{3n}$	$\frac{2n}{3n+2m}$	as *COR*	$\frac{2n}{2n+3m}$	as *CSR*
$y \leftarrow \alpha L^{-1} y$	$N.A.$	$\frac{n}{3n+m}$	$\frac{n}{4n}$	$\frac{n}{2n+2m}$	$\frac{n}{3n+2k}$
$y \leftarrow \alpha L^{-T} y$	$N.A.$	$\frac{n}{4n}$	$\frac{n}{3n+k}$	$\frac{n}{3n+k}$	$\frac{n}{2n+2k}$
$\forall i,j\ a_{i,j} \leftarrow x_i a_{i,j}$	$\frac{n}{2n}$	$\frac{0}{2n}$	$\frac{n}{2n}$	$\frac{0}{n+2m}$	$\frac{n}{2n}$

Table 1.2: Overview of the amount of *indirect memory accesses*. Each entry in the table gives a ratio of indirect memory accesses to direct memory accesses. We regard a memory access as indirect when its location address was computed using an index which was either loaded or computed from some other array. Namely, the count of indirect accesses for an operation, comprehends the sum of both read (R) an write (W) accesses marked as *random* (r) in Table 1.1. With an analogy, the count of direct accesses for an operation, comprehends the sum of both read (R) an write (W) accesses marked as *sequential* (s) in that table.
Legend
$\star$: symmetric matrix (only lower representation); to be more precise, from the overall writes count, one should subtract the count of elements on the diagonal; $m = k$ is implied
All the remaining assumptions made in Table 1.1 hold here as well.

rows; namely, it is capable of merging more than one incomplete diagonal stripe in one, if possible. It was developed for *vector processors*.

- **ITPACK Storage (*ITP*)**: *ITP* stores compressed columns, but giving each compressed stripe the same length by means of padding with zeroes.

- **Jagged Diagonal Storage (*JDS*)**: *JDS* (also known as JAD) represents the (compressed) diagonals occurring in the matrix obtained by sorting the *CSR* index arrays according to their population. *JDS* is a format particularly suitable for vector machines. See Tiyyagura et al. ([TKB06]), or Mills et al. ([MDF]) for developments on recent architectures.

- **Skyline Storage (*SKS*)**: Also called *variable band storage*, it uses a row pointer array, but keeps nonzeroes in dense subvectors, without storing column indices, since they can be inferred by visiting the band. This format allows the efficient execution of some direct solution methods, as the Gaussian elimination for instance.

As we see, these formats differ substantially from the *COO*/*CSR* variants we have described so far. Indeed, in this thesis we will develop techniques based on the *CSR*/*COO* formats. However, extending the short study made in §1.4 to these formats and related algorithms may lead to interesting developments (see §6.3).

1.5.2 Considerations and Literature Pointers

In the algorithms presented in the preceding sections, we often made some assumptions: the rows are sorted in ascending order, no duplicate elements occur, and so on. Common variations which occur in the **Sparse BLAS** specification involve: all combinations of untransposed/transposed/hermitian, symmetric/unsymmetric, four **BLAS** numerical types (either *single* or *double* precision *real* or *complex* floating point numbers), arbitrary vectors stride, diagonal implicit/explicit storages. It is clear that an attempt into handling all of these cases in a correct and optimized way is a daunting task. A possible way to overcome the correctness, maintainability, and others difficulties (to say, coherent source documentation and automated testing also), would be to model some specification of an optimal *source* code and generate it automatically. Different aspects of these considerations have been studied and reported in past research efforts. The **Sparsity** ([IYV04b]) framework by Im, Yelick, Vuduc employs code generation techniques for creating source code for *BCSR* and *CB* (see §2.1), among other formats. With the OSKI (Optimized Sparse Kernels Interface) library, Vuduc uses automatically generated kernels and employs heuristic search techniques for inferring the best *BCSR* blocking and matrix *splitting*[19](see Vuduc et al. [VDY05b]). Initially motivated by research in compilers, Bik and Wijshoff have created a source-to-source code translator, or *sparse matrix compiler*, named **MT1**. Initially, **MT1** dealt with the problem of hiding a *sparse semantics* behind a *dense* syntax; that is, the user is required to specify his matrix algorithms in Fortran, with a number of annotations in the comments (see [BBW97], [BW96]). The compiler's role, in **MT1**, is that of interpreting both the user's annotations and the (coded as) dense algorithm specification, and producing appropriate declarations and code reformulated as sparse. In further research, Bik et al. focused on the generation of primitives for sparse operations (see [BBKW98]), discussing the code generated by their compiler for matrix-vector product, triangular solve, matrix-matrix product, triangular multi-vector-solve. One of their conclusions was that the higher level the annota-

[19]That is, representing a matrix as the *sum* of matrices which are disjoint by sparsity pattern (namely, each given effective nonzero at (i, j) is represented in exactly one of the "summand" matrices), and have differently *blocked* representations, rather than a single rows/columns partitioning.

tions/specifications are, the higher the chances for an appropriate optimization of the whole transformation. We believe that the best approach into producing a high performance and maintainable **Sparse BLAS** library is that of using all of these techniques. With a cautious approach, and to a limited extent, we have applied the knowledge gained from research efforts we have cited into producing a matrix format of our own, as the following chapters will describe.

2 Hierarchical Representations of Sparse Matrices

Overview

By *hierarchical representation* we mean a representation of sparse matrices offering *multiple levels* of addressability; for instance, organizing a matrix in *submatrices*, by means of some additional data structure on the top of them.

Generally, we distinguish *flat*, two-level representations, based on a single, two dimensional *blocking* of the matrix, from multi-level, (possibly *recursive*) representations. A recursive, two-dimensional partitioning of a matrix would allow the addressing of submatrices data at various *levels*. In this chapter, we will describe three sparse matrix formats, covering the mentioned combinations of flat and recursive layouts.

The first documented hierarchical representations of matrices date back to the early history of information technology. A motivation for the development of these techniques was to optimize the complexity of input/output with external storage; over the years, with different, evolving technologies. As a first thing, hierarchical representations enabled *out of core* computations; that is, computations on datasets exceeding the amount of available physical memory. This was an especially valued feature, given the more limited amount of memory computer users had in the past. For instance, back in 1969, McKellar and Coffman ([MC69]) considered dense matrix computations on *paged memory sys-*

tems. They considered three ways of storage: by rows, *row blocks*, *recursive*[1]. Since their problems involved the arrangement of submatrices in *memory pages*, their considerations focused on the problem of maximizing *mean page residence time*, also in view of the extra storage needed to adapt to the page size. The algorithms they proposed address operations of addition, multiplication, transposition, and inversion of dense matrices. They also assumed the applicability of the so-called *Belady's algorithm*(See Belady [Bel66]), which assumes the program is both able and allowed to control the memory pages replacement policy[2].

Subsequently, *hypermatrix* (multilevel indexing based) techniques have been applied for the solution of linear systems. In 1972, von Fuchs et al. (see [vFRS72]) report the use of such techniques for Cholesky factorization of sparse *structural stiffness* matrices. However, representation of individual submatrices during factorization is still dense.

Curiously, notable software packages developed for iterative methods in the 1980's and early 1990's neglected the use of hierarchical/recursive representations of sparse matrices (see some of Vuduc's bibliographical material in [Vud03, § 2.3]). With the re-emergence of shared memory parallelism and higher impact of memory access latency[3], since the first 2000's, hierarchical representations of sparse matrices have been used more and more for factorizations: see the works of Irony et al. on Cholesky factorization in [IST04], Dongarra et al. on LU factorization ([DEL01]), and again on Cholesky factorization with the works of Herrero and Navarro ([HN08]).

On the other hand, in the field of (distributed memory) parallel computing, data partitioning was used extensively. For recent work, see [VB05], where Vastenhouw et al. use asymmetrical *recursive bipartitioning* of sparse matrices in a distributed computing context. Sparse hypermatrix techniques were also reportedly used for distributed-memory operations in the **PERMAS** proprietary package for FEM analysis (Fischer et al. [FALM]). The topic of the optimal balancing of sparse matrix computations across distributed processors was often considered; for instance, see Pinar and Aykanat in [PA97].

[1] In their paper, respectively named *row storage*, *packed rows storage*, and *submatrix storage*.

[2] An assumption which holds for many modern architectures, especially ones using *scratchpad memories*: small and fast (compared to main RAM) memory areas under the programmer's control, with programmable transfers from the RAM. Also *cache control* instructions present on modern processors allow, to a limited extent,Belady's assumption (see [AMD07, §3.9.6],[Int08a, Ch. 7]).

[3] Over the years, memory access latency has been improving relatively much less than CPU processing speed (in terms of Millions of Instructions for Second (MIPS)); see Patterson and Hennessy [PH04, § 1.4,Fig. 7.37].

While the interest in hierarchical representations for sparse matrix computations has been quite modest, research about the use of hierarchical representations of point or black/white image data has been very widely studied.

Research on the application of the *quad-tree* structure (introduced by Finkel and Bentley, see [FB74]) has been extensive, especially in computer graphics; see for instance the works of Samet ([Sam84], or [Sam06, § 2.1.2.4]).

In the following sections, we will first introduce two recent, hierarchical sparse matrix layouts conceived primarily for *SpMV*. These are the *Cache Blocking* format (*CB*), in §2.1, and the *Compressed Sparse Blocks* format (*CSB*), in §2.2.

Then, in §2.3, we will proceed with the exposition of *Recursive CSR*, (*RCSR*), a quad-tree-based hierarchical layout for sparse matrices designed and implemented by us. This matrix layout develops ideas and techniques present in the mentioned literature, as well as in *CB* and *CSB*. The rest of the thesis is devoted to this sparse matrix format and related algorithms.

2.1 CB: Cache Blocking

Introduced in this context by Im and Yelick in [IY99], also described in [Im00, Ch. 4], the term *Cache Blocking* refers to techniques for performing optimized *SpMV* computations on individual blocks of a sparse matrix, sized in a way that the *working set* fits in (some level of) cache memory. With *CB*, at any given time, only elements in some specified $r_{cache} \times c_{cache}$-sized submatrix could be referenced.

Authors of *CB* distinguish between techniques for *static* and *dynamic* cache blocking. While static cache blocking specifies both a data structure and a *SpMV* algorithm, dynamic *CB* refers only to a particular arrangement of code to process the $r_{cache} \times c_{cache}$-sized sparse submatrices, one at a time, by keeping the whole matrix stored as *CSR*. Now on, with *CB*, we will refer to *static cache blocking* only, as described in [Im00, Ch. 4].

This format is hierarchical, for there is an array whose entries *point* to individual *block rows*, represented as *CSR* sub-arrays enclosed in three arrays.

Please see Fig. 2.1 for an instance of *CB* of an example matrix.

CB has been originally developed for the **Sparsity** code package (see Im et al. [IYV04b]); later on, it has been implemented in the **Sparsity**-based OSKI sparse kernels library by Vuduc (see [VDY05b]).

By effectively storing small *sparse blocks*, one increases locality into accessing the x and y vectors during *SpMV*, although at the cost of a slightly higher

$$A = \begin{pmatrix} \begin{pmatrix} 0.66667^{(1)} & 0.36656^{(2)} & 0.30011^{(3)} \\ 0.10004^{(4)} & 0.53341^{(5)} & -1^{(6)} \end{pmatrix} & \begin{pmatrix} 0.36656^{(7)} & 0.30011^{(8)} \\ 0.20007^{(9)} & 0 \end{pmatrix} \\ \begin{pmatrix} 0.12219^{(10)} & 0 & 0.5777^{(11)} \\ 0.05002^{(12)} & 0.10004^{(13)} & 0 \end{pmatrix} & \begin{pmatrix} 0 & 0.24437^{(14)} \\ 0.28331^{(15)} & 0.18328^{(16)} \end{pmatrix} \\ \begin{pmatrix} 0.06109^{(17)} & 0 & 0.12219^{(18)} \end{pmatrix} & \begin{pmatrix} 0.15006^{(19)} & 0.27224^{(20)} \end{pmatrix} \end{pmatrix}$$

IA_{CB}=(1 4 7 9 10 12 14 15 17 19 19 21)
JA_{CB}=(1 2 3 1 2 3 4 5 4 1 3 1 2 5 4 5 1 3 4 5)
VA_{CB}=(0.66667 0.36656 0.30011 0.10004 0.53341 -1 0.36656 0.30011 0.20007 0.12219 0.5777 0.05002 0.10004 0.24437 0.28331 0.18328 0.06109 0.12219 0.15006 0.27224)
BP_{CB}=(1 7 10 14 17 19)

Figure 2.1: Cache Blocked representation of matrix *cage3** (with $r_{cache} = 2, c_{cache} = 3$). The individual sparse blocks are marked by parentheses. We have trimmed the blocks on the right and lower sides to not exceed the overall matrix dimensions: for this reason these blocks are dimensioned less than 2×3. We have named the arrays with a notation of our convenience.

indexing overhead. Contrarily to *BCSR* (see §1.2.4), no nonzeroes are stored explicitly, in any case.

The feature of *cache blocking CB* introduces for sparse matrices storage, will serve as a basis for the *CSB* format, presented in the following section.

2.2 CSB: Compressed Sparse Blocks

CSB (See Buluç et al. [BFF+09]) is a format introduced by Buluç et al. with the aim of enabling a scalable parallel execution of both *SpMV* and *SpMV-T*. The original paper about *CSB* describes an implementation relying on the **CILK** (see Blumofe et al. [BJK+95]) system as a scheduler for supporting its shared memory parallel execution.

Given a square matrix A, sized[4] $n \times n$, and given a parameter β, *CSB* partitions A as:

[4]Non square dimensions are supported as well, at the condition of keeping square sparse blocks (see [BFF+09, § 3]).

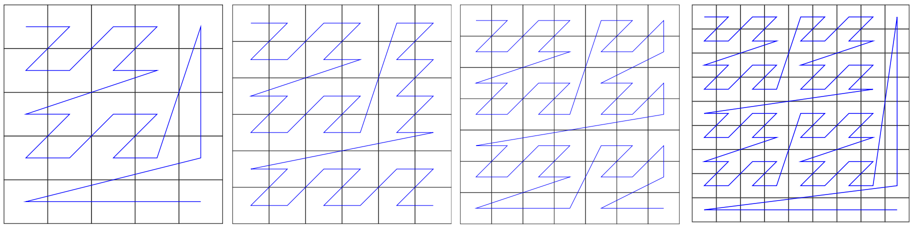

Figure 2.2: Z-sorted coordinates for 5x5,6x6,7x7,9x9 sized dense matrices. The line follows the order of the coordinates, beginning at the top left. Please notice both the imbalance regarding the matrix original dimensions, and the existing rows/columns symmetry.

$$A = \begin{pmatrix} 0.66667^{(1)} & 0.36656^{(2)} & 0.30011^{(5)} & 0.36656^{(6)} & 0.30011^{(14)} \\ 0.10004^{(3)} & 0.53341^{(4)} & -1^{(7)} & 0.20007^{(8)} & 0 \\ 0.12219^{(9)} & 0 & 0.5777^{(12)} & 0 & 0.24437^{(15)} \\ 0.05002^{(10)} & 0.10004^{(11)} & 0 & 0.28331^{(13)} & 0.18328^{(16)} \\ 0.06109^{(17)} & 0 & 0.12219^{(18)} & 0.15006^{(19)} & 0.27224^{(20)} \end{pmatrix}$$

IA=(1 1 2 2 1 1 2 2 3 4 4 3 4 1 3 4 5 5 5 5)
JA=(1 2 1 2 3 4 3 4 1 1 2 3 4 5 5 5 1 3 4 5)
VA=(0.66667 0.36656 0.10004 0.53341 0.30011 0.36656 -1 0.20007 0.12219 0.05002 0.10004 0.5777 0.28331 0.30011 0.24437 0.18328 0.06109 0.12219 0.15006 0.27224)

Figure 2.3: Z-Morton ordered *COO* representation of matrix *cage3**.

$$A = \begin{pmatrix} A_{00} & A_{01} & \cdots & A_{0,n/\beta-1} \\ A_{10} & A_{11} & \cdots & A_{1,n/\beta-1} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n/\beta-1,0} & A_{n/\beta-1,1} & \cdots & A_{n/\beta-1,n/\beta-1} \end{pmatrix} \tag{2.1}$$

Nonzeroes from each block are stored contiguously, thus making *CSB* a particular form of *cache blocking*. Within, each block is stored using the three arrays of *COO* (recall §1.1), Morton-ordered. See Fig. 2.2 for a visual representation of Z-Morton ordering for some small matrix, and Fig. 2.3 for our example matrix stored in Z-Morton ordered *COO*. This ordering of nonzeroes within blocks ensures the same cache behaviour (statistically, in terms of cache misses) for the matrix, right-hand side, and result arrays accesses, regardless of the execution of either *SpMV* or *SpMV-T*. A simple explanation for the non-bias comes from observing that without any assumption on the nonzeroes distribution, when sweeping the block coordinates, transitions changing row index have the same probability of transitions changing column index. A side effect of such dynamics is a possible lesser hit rate of the prefetch engines when guessing new x and y addresses, of course.

Since blocks are sparse, and many of them are possibly *empty* (think of a banded matrix), a *block pointers array* is used to store the offset of each sparse block within the global *COO* arrays. The block pointers array has $(n/\beta) \cdot (n/\beta) = n^2/\beta^2$ integer elements; one per block, whether empty or not. Therefore, *random access* to the memory location of a particular coordinate requires, first, a lookup in this array, and then, if the block is non-empty, a search in the sparse block indices arrays. Among themselves, blocks are stored row-major, thus breaking the row/column symmetry blocks have within their contained coordinates, therefore limiting the *cache obliviousness* of *CSB* to the individual blocks. Authors of [BFF+09] report this as not being a source of difference between the transposed and untransposed *SpMV* performance, in practice.

Fig. 2.4 shows a *CSB* instance of matrix *cage3**; that is, the two arrays with the row and column indices, the array with the numerical values, and the block pointers array.

Since within a sparse block, the row and column offsets are known, there is no need to store indices as *global*: it is sufficient for them to be *local* to the block submatrix, instead. This means that in practice, only indices in the $[1 \ldots \beta]$ range occur, and an implementation could use less bits than it would necessary to store values in the $[1 \ldots n]$ range. A side effect of this, is that by using an index type shorter than the usual; for instance, 16 bits integers instead 32 bit ones[5], storage required for *COO* indices may be less than that needed by *COO* or *CSR* formats.

Authors of *CSB* report some limitations of this format, at the point of their current implementation. One is the missing support for specialized, symmetric updates when multiplying symmetric matrices; this technique is usually exploited to attain a nearly double write-to-read rate; see the case for *CSR* in §1.2.2. Other open questions pertain to the existence of efficient algorithms for matrix factorization, triangular solve, or factorizations for *CSB*; formulations for *CSR*/*CSC* exist and are well known.

[5]This has been the authors choice in their *CSB* prototypal code distribution.

$$A = \begin{pmatrix} \begin{pmatrix} 0.66667^{(1)} & 0.36656^{(2)} \\ 0.10004^{(3)} & 0.53341^{(4)} \end{pmatrix} & \begin{pmatrix} 0.30011^{(5)} & 0.36656^{(6)} \\ -1^{(7)} & 0.20007^{(8)} \end{pmatrix} & \begin{pmatrix} 0.30011^{(9)} \\ 0 \end{pmatrix} \\ \begin{pmatrix} 0.12219^{(10)} & 0 \\ 0.05002^{(11)} & 0.10004^{(12)} \end{pmatrix} & \begin{pmatrix} 0.5777^{(13)} & 0 \\ 0 & 0.28331^{(14)} \end{pmatrix} & \begin{pmatrix} 0.24437^{(15)} \\ 0.18328^{(16)} \end{pmatrix} \\ \begin{pmatrix} 0.06109^{(17)} & 0 \end{pmatrix} & \begin{pmatrix} 0.12219^{(18)} & 0.15006^{(19)} \end{pmatrix} & \begin{pmatrix} 0.27224^{(20)} \end{pmatrix} \end{pmatrix}$$

IA_{CSB}=(2 2 1 1 2 2 1 1 2 2 1 1 2 1 2 1 2 2 2 2)
JA_{CSB}=(2 1 2 1 2 1 2 1 2 2 2 1 2 1 2 2 2 2 1 2)
VA_{CSB}=(0.66667 0.36656 0.10004 0.53341 0.30011 0.36656 -1 0.20007 0.30011 0.12219 0.05002 0.10004 0.5777 0.28331 0.24437 0.18328 0.06109 0.12219 0.15006 0.27224)
BP_{CSB}=(1 5 9 10 13 15 17 18 20)

Figure 2.4: *CSB*-ordered representation of matrix *cage3**(with $\beta = 2$). Note that the *CSB* paper does not specify how to handle the case when n is not divided by β, (as in the case of *cage3**). Probably, the most reasonable solution would be that of handling *peripheral* blocks separately, as a corner case.

2.3 RCSR: A Recursive Layout

In §2.1 and §2.2, we have introduced layouts partitioning matrices in a cache-friendly manner. In this section, we propose a layout based on a *recursive quad-partitioning* of sparse matrices. This format, besides offering a form of cache blocking, is capable of supporting the various **Sparse BLAS** (Duff et al. [DHP02]) matrix variants; that is, diagonal implicit and/or symmetric representations, and both multiplication and triangular solve operations. In the following chapters, we will develop and tune thread-level parallel algorithms for these operations.

We label this format (perhaps improperly) *RCSR*: *Recursive CSR*.

Given a matrix A, we define its *RCSR* representation in memory as the *quad-tree* (see Finkel and Bentley [FB74]) Q_A, having:

- as **root**, the whole matrix A
- as **leaves**, submatrices of A, represented with *CSR* arrays
- as **intermediate nodes**, the quadrant submatrices of A, and the submatrices resulting from their recursive subdivision in quadrants

With *recursive subdivision in quadrants*, or *recursive quad-subdivision* of matrix A, sized $m \times k$, we mean the quadrants $A_{11}, A_{12}, A_{21}, A_{22}$, sized respectively (in clockwise order, from the upper left) $\lceil \frac{m}{2} \rceil \times \lceil \frac{k}{2} \rceil$, $\lceil \frac{m}{2} \rceil \times \lfloor \frac{k}{2} \rfloor$, $\lfloor \frac{m}{2} \rfloor \times \lceil \frac{k}{2} \rceil$, and $\lfloor \frac{m}{2} \rfloor \times \lfloor \frac{k}{2} \rfloor$.

That is, for A:

$$A = \begin{vmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{vmatrix} \tag{2.2}$$

Subdivision could proceed in any of the quadrant submatrices.

If subdivision results in some submatrix with no nonzero element, at some *level*, that submatrix is not subdivided further. Therefore we do not represent *empty* submatrices in *RCSR*. Each submatrix node contains information about the rows and columns range within the matrix, as well as the enclosed nonzeroes; thus a square zeroes-only region may be identified *by exclusion*, as its corresponding *pointer* is missing, but its extent is known, given the above stated subdivision rule for submatrices.

According to the above definition, no two *leaf submatrices* $(s, s^{'})$ of a given matrix may *overlap*.

In the case A is square, we have that:

- each submatrix intersecting the diagonal is also aligned to it
- each submatrix intersecting the diagonal is square

See Fig. 2.5 for an example of quad-tree partitioning a matrix.

The following subsections show the basic way for implementing the **BLAS**-oriented operations *SpMV* and *SpSV* when the matrix is partitioned recursively with *RCSR*. Note that the actual *leaf submatrices* format is irrelevant to the proposed operations breakdown. After giving more details about our implementation of the *RCSR* layout in the following sections, in §2.4 we will report some experiments comparing both *RCSR* and *RCSC* layouts performance to that of other software/formats.

2.3.1 *SpMV* for a Recursive Subdivision Layout

Considering the basic decomposition in four quadrants of matrix A, multiplication by a vector x ("$y \leftarrow \ y \ + \ A \ x$")[6] could be formulated in the following way.

$$Ax = \begin{vmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{vmatrix} \begin{vmatrix} x_1 \\ x_2 \end{vmatrix} = \begin{vmatrix} A_{11} & A_{12} \\ 0 & 0 \end{vmatrix} \begin{vmatrix} x_1 \\ x_2 \end{vmatrix} + \begin{vmatrix} 0 & 0 \\ A_{21} & A_{22} \end{vmatrix} \begin{vmatrix} x_1 \\ x_2 \end{vmatrix}$$

[6]The same breakdown would hold for any other dense matrix x, of course.

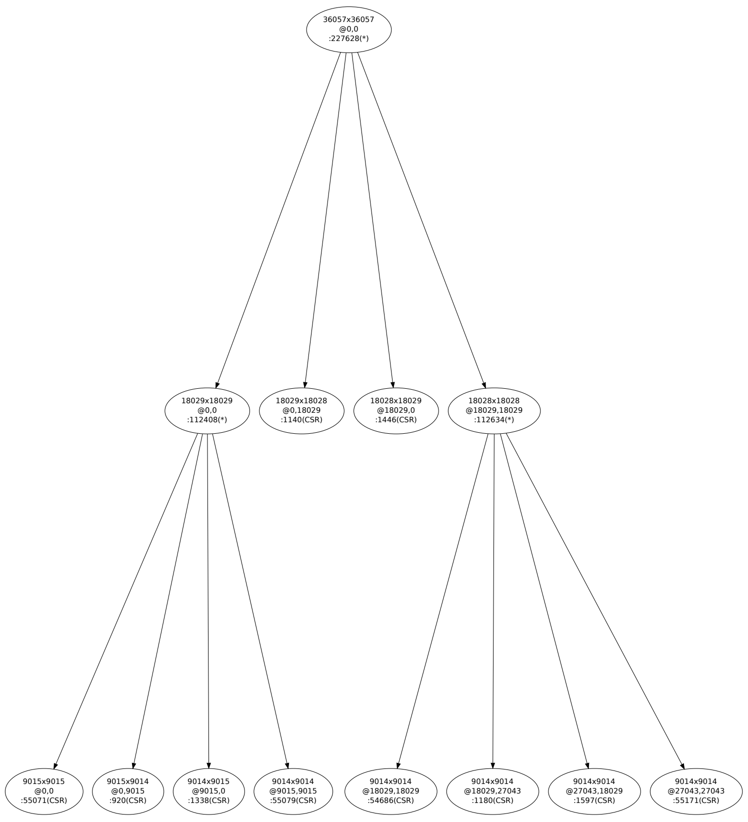

Figure 2.5: Quad-tree partitioning for matrix *onetone*, on machine **M6**. Labels at each submatrix node report: dimensions, 0-based location offset in the matrix, number of the nonzeroes enclosed in that submatrix, and format, in the case of leaves.

$$= \begin{vmatrix} A_{11}x_1 + A_{12}x_2 \\ 0 \end{vmatrix} + \begin{vmatrix} 0 \\ A_{21}x_1 + A_{22}x_2 \end{vmatrix} \tag{2.3}$$

Because of the recursive matrix layout, the above computation breakdown is valid on any other submatrix of A and corresponding subvector of x.

For the transposed case ("$y \leftarrow \ y \ + \ A^T \ x$") we have:

$$A^T x = \begin{vmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{vmatrix}^T \begin{vmatrix} x_1 \\ x_2 \end{vmatrix} = \begin{vmatrix} A_{11}^T & A_{21}^T \\ A_{12}^T & A_{22}^T \end{vmatrix} \begin{vmatrix} x_1 \\ x_2 \end{vmatrix} = \begin{vmatrix} A_{11}^T & A_{21}^T \\ 0 & 0 \end{vmatrix} \begin{vmatrix} x_1 \\ x_2 \end{vmatrix} + \begin{vmatrix} 0 & 0 \\ A_{12}^T & A_{22}^T \end{vmatrix} \begin{vmatrix} x_1 \\ x_2 \end{vmatrix}$$

$$= \begin{vmatrix} A_{11}^T x_1 + A_{21}^T x_2 \\ 0 \end{vmatrix} + \begin{vmatrix} 0 \\ A_{12}^T x_1 + A_{22}^T x_2 \end{vmatrix}$$

Note that when computing $A_{21}^T x_2$ and $A_{21}^T x_2$, in order to write to the appropriate destination subvector, also the submatrix position should be taken into account. Therefore, the *SpMV*/*SpMV-T* algorithms in Fig. 1.16/Fig. 1.18 should be slightly modified for this purpose.

Let us consider now a symmetric representation of matrix A, storing only A's (non strictly) lower triangle L. In this case, a specialized kernel to compute simultaneously $y_1 \leftarrow y_1 + L_{21}^T x_2$ and $y_2 \leftarrow y_2 + L_{21} x_1$ could still be used, in a way to avoid storing (or visiting) twice the L_{21} submatrix. However, the traditional *CSR* kernel (as listed in Fig. 1.20) needs some modification, in order to accommodate the appropriate destination subvector offset, *and* to act consequently whether the given submatrix straddles or not the main diagonal of A (think of the diagonal as a vector D). In the case the submatrix straddles D, as with L_{11} or L_{22}, the symmetric *SpMV* kernel should compute both the transposed and untransposed contributions, but taking into account multiplying diagonal elements (that is, elements in D) only once. In the case the submatrix does not cross D, as with L_{21}, the symmetric *SpMV* kernel should compute both the transposed and untransposed contributions, without having to skip the diagonal elements.

$$Ax = \begin{vmatrix} L_{11} & L_{21}^T \\ L_{21} & L_{22} \end{vmatrix} \begin{vmatrix} x_1 \\ x_2 \end{vmatrix} = \begin{vmatrix} L_{11} & L_{21}^T \\ 0 & 0 \end{vmatrix} \begin{vmatrix} x_1 \\ x_2 \end{vmatrix} + \begin{vmatrix} 0 & 0 \\ L_{21} & L_{22} \end{vmatrix} \begin{vmatrix} x_1 \\ x_2 \end{vmatrix}$$

$$= \begin{vmatrix} L_{11}x_1 + L_{21}^T x_2 \\ 0 \end{vmatrix} + \begin{vmatrix} 0 \\ L_{21}x_1 + L_{22}x_2 \end{vmatrix}$$

The proposed solutions of computation breakdown are valid recursively, in that they are valid on each recursive submatrix and subvector, given an implementation appropriately handling on-diagonal submatrices and submatrix positions.

In the most general formulation of *SpMV* ("$y \leftarrow \beta\ y\ +\ \alpha\ A\ x$"), scaling of the result vector should be handled separately (actually, before) the outlined schema.

2.3.2 *SpSV* for a Recursive Subdivision Layout

When matrix A is lower triangular ($A = L$); that is, when $i < j \Rightarrow a_{ij} = 0$, the solution x of a triangular system $Lx = b$ with a recursive subdivision layout could be computed according to the following equations[7], or equivalently, with the procedure in Fig. 2.6.

$$Lx = b \Rightarrow \begin{vmatrix} L_1 & 0 \\ M & L_2 \end{vmatrix} \begin{vmatrix} x_1 \\ x_2 \end{vmatrix} = \begin{vmatrix} b_1 \\ b_2 \end{vmatrix} \tag{2.4}$$

$$x = \begin{vmatrix} x_1 \\ x_2 \end{vmatrix} = L^{-1}b = \begin{vmatrix} L_1 & 0 \\ M & L_2 \end{vmatrix}^{-1} \begin{vmatrix} b_1 \\ b_2 \end{vmatrix} = \begin{vmatrix} L_1^{-1}b_1 \\ L_2^{-1}(b_2 - Mx_1) \end{vmatrix} \tag{2.5}$$

This decomposition leads us to the following steps:

Figure 2.6: Recursive Blocked Triangular Solve operation breakdown.

```
1 Solve $L_1x_1 = b_1$ for $x_1$ (SpSV)
2 Compute $b_2 - Mx_1$ (SpMV)
3 Solve $L_2x_2 = (b_2 - Mx_1)$ to find $x_2$ (SpSV)
```

When encountering a leaf submatrix on the diagonal, a traditional *SpSV* kernel (as that in Fig. 1.23) should be used, instead of the recursive wrapper. Multiplication of the M submatrix (and its quadrants, recursively) by x_1 follows the recursive rules presented in §2.3.1.

In the case a scaled ($x = \alpha L^{-1}b$) or a transposed ($x = \alpha L^{-T}b$) solution should be computed, some trivial modifications should be made to the above schema.

2.3.3 Sorting for Recursive Partitioning

Having outlined algorithms for performing *SpMV* and *SpSV* on this irregular blocking we propose, in this section we sketch a basic algorithm for building such

[7]Recall, from §2.3, that submatrices on the diagonal are square, and thus correct as input to a common triangular solve kernel.

matrices. The purpose of the algorithm at hand is to obtain in output a matrix instance in the *RCSR* layout, after having processed input arrays specified as unsorted *COO* (recall §1.1). Please notice that unsorted *COO* is the most general input an application could handle, and is especially common when using higher level scripting languages.

As a first thing, we define a criteria for ordering the input coordinate elements in an appropriate, *recursive* way. Then, we will explain how to proceed using this ordering for recursive partitioning.

We present here a variant of Z-ordering we will use to sort *all* input nonzeroes according to.

Let $x, y \in \mathbb{N}$, be the Cartesian coordinates of a point in $\mathbb{N}^2$, and T be a function $T : (x, y) \in \mathbb{N}^2 \rightarrow z \in \mathbb{N}$. Then, we can define the T-permutation Π_T of a vector $V = \langle (i_0, j_0), (i_1, j_1), ..., (i_{nnz}, j_{nnz}) \rangle$ as the vector (assuming no duplicates in V) $\Pi_T(V) = \langle \pi_0, \pi_1, ..., \pi_{nnz} \rangle$ such that $\pi_0 {<} \pi_1 {<} ... {<} \pi_{nnz}$ and $\pi_l {<} \pi_k$ hold whenever $T(i_{\pi_l}, j_{\pi_l}) {<} T(i_{\pi_k}, j_{\pi_k})$. Given $i \in \mathbb{N}$, we define (adopting the notation of Raman et al. in [RW08, section 2]) the 2-dilation of i, $\overrightarrow{i}$ (*"i dilated"*) as the result of interleaving a 0 bit between each meaningful bit in the binary representation of i. So, if $i = 2^8 - 1 = 11111111_2 = \mathtt{FF}_{16}$, its 2-dilation is $\overrightarrow{i} = 0101010101010101 = 5555_{16}$. Similarly we define $\overleftarrow{i} \overset{def}{=} 2\,\overrightarrow{i}$, which is the left-shifted 2-dilation of i. Let us now define the mapping Z as: $Z(i, j) \overset{def}{=} \overrightarrow{j} + \overleftarrow{i}$. Above, if we take T to be Z and apply element-wise to the *coordinate vector* V, then we induce a Z-order on V. In Fig. 2.8 we depict the resulting ordering of elements for some small dense matrices. Experiments reported by Lorton and Wise ([LW07]) show that performing linear algebra on Z(Morton) sorted elements could reduce page faults for large dense matrices. We conjecture this to be true for sparse matrices too, as the sparseness of elements leads to some non-linear (thus, not easily detectable by the prefetch engines—recall §1) access patterns. By forcibly limiting the *leaf matrix* dimensions, while storing and tiling them in a recursive Z fashion, we increase the locality of memory accesses when a right-hand side vector is involved, regardless of the matrix sparsity pattern. However, Z-ordering nonzeroes of matrices which are not square or not sized as powers of 2 could lead to a somewhat imbalanced partitioning (see Fig. 2.2, where we depict small dense matrices with "singleton" leaves).

To address this issue, we have modified the Z-ordering algorithm to handle non-square matrices and non-power-of-two sized matrices.

We call our modification *balanced Z ordering*, or Z^b. Let the matrix size be $m \times k$ and i, j a nonzero coordinate (1-based). Let $lbits(i)$ be the index of the highest bit in the binary representation of i: $lbits(i) \overset{def}{=} \lfloor log_2(i) \rfloor$, and let β_{mk} be $lbits(min(m,k))$.

Then define: $\mu : i, m, \beta_{mk} \in \mathbb{N} \to i^* \in \mathbb{N}$ as $\mu(i, m, \beta_{mk}) \overset{def}{=} \gamma(i, \lfloor m/2 \rfloor) \cdot (2^{\beta_{mk}} + \mu(i - \lfloor m/2 \rfloor, m - \lfloor m/2 \rfloor, \beta_{mk} - 1)) + (1 - \gamma(i, \lfloor m/2 \rfloor))\mu(i, \lfloor m/2 \rfloor, \beta_{mk} - 1)$.

With $\gamma(x, y) = 1$ when $x > y$ and 0 otherwise. The Z^b order function of interest is then defined as: $Z^b(i, j, m, k) \overset{def}{=} Z(\mu(i, m, \beta_{mk}), \mu(j, k, \beta_{mk}))$.

Figure 2.8 shows the Z^b-ordered elements of some small dense matrices. Note that using Z^b instead of Z has a downside: Z^b is not bijective. This is not a problem for our application, as long as we do not need a bijection, and use Z^b ordering for sorting purposes only. Knowing values of m, k, in a Z^b-ordered coordinates array, we can use binary search to easily locate *split points* delimiting the four submatrices; see the steps listed in Fig. 2.7.

Once the first four quadrants are delimited, the procedure could be applied to the individual quadrants again, recursively.

By itself, the procedure for locating the submatrices requires repeated binary searches over the input arrays. Since after each run of the proposed search routine the number of enclosed nonzeroes is known, one could define a simple criteria when to stop subdividing.

In any case, the physical relocation of the input arrays (or *shuffling*), could be postponed *after* all the boundaries of *leaf submatrices* are located.

Figure 2.7: $FIND\_QUAD\_SPLIT\_POINTS(I, J, n, frow, fcol, rows, cols)$

```
1 /*Assumes that the n-sized row and column indices arrays $I, J$ are
  $Z^b$-sorted*/
2 /*Also assumes that elements in $I$ are contained in the
  $[frow...frow + rows]$ interval, and that elements in $J$ are contained in
  the $[fcol...fcol + cols]$ interval.*/
3 Binary search for the first index $m \in [1...n]$ s.t. $I[m] \geq \lceil \frac{frow+rows}{2} \rceil$
4 Binary search for the first index $u \in [1...m]$ s.t. $J[u] \geq \lceil \frac{fcol+cols}{2} \rceil$
5 Binary search for the first index $l \in [m...n]$ s.t. $J[l] \geq \lceil \frac{fcol+cols}{2} \rceil$
6 /*The four quadrants are located (clockwise) in the intervals:
  $[1...u), [u...m), [m...l), [l...n]$ of the COO arrays*/
7 **return** $u, m, l$
```

We notice that some of the quadrants are empty, whenever any of the following holds: $1 = u$, $l = m$, $m = l$, $l = n+1$. In Fig. 2.7, we denote an ascending

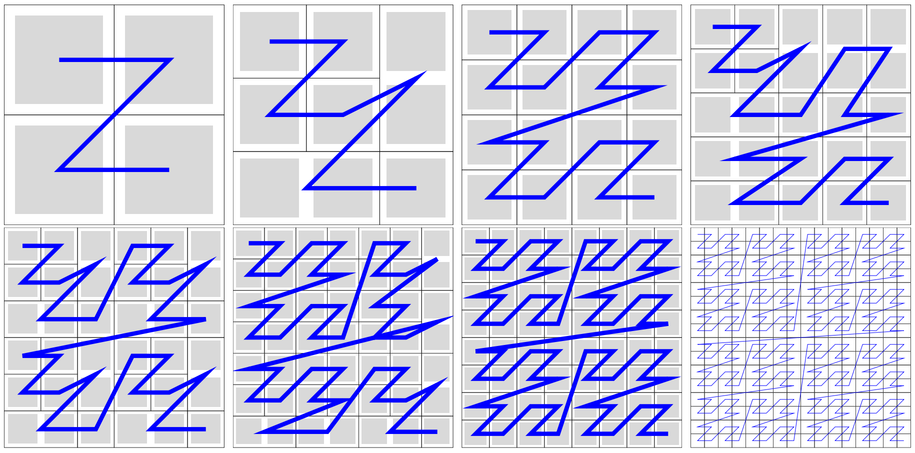

Figure 2.8: Z^b sorted coordinates for 2x2,3x3,4x4,5x5,6x6,7x7,8x8,16x16 dense matrices, sized as non power of 2. Notice the resulting balanced quad-partitions.

interval including x and excluding y with $[x, ..., y)$.

In the following section, we deal with the topic of *when stopping* subdivisions.

2.3.4 Recursive Subdivision

To prevent indefinite recursive splitting, we introduce a *recursion decision* (or *cutoff*) function. Currently, this function (δ) is a heuristic working with matrix dimensions m, k, number of nonzeroes nnz, outermost machine cache size CS, numerical and pointer element sizes (expressed in *bytes*); respectively ES and WS.

$$eab(m, k, nnz, ES, WS) \overset{def}{=} \tag{2.6}$$

$$ES \cdot (nnz + nnz + m) + WS(m + nnz) \tag{2.7}$$

$$\delta(m, k, nnz, CS, ES, WS) \overset{def}{=} \tag{2.8}$$

$$\textbf{True}, if\ eab(m, k, nnz, ES, WS) > \alpha\ CS,\ \textbf{or} \tag{2.9}$$

$$\textbf{True}, if\ \ nnz \cdot ES > \beta\ CS \tag{2.10}$$

$$\textbf{False}, otherwise \tag{2.11}$$

$$\tag{2.12}$$

Here, *eab* is an estimate of the accessed bytes during a *CSR SpMV* on a (sub) matrix with the given parameters. Term $ES \cdot (nnz+nnz+m)$ takes into account the nnz accessed multiplicand vector elements, the nnz matrix elements, and m written output vector elements. The $WS \cdot (m+nnz)$ term takes into account the m row pointer indices and the nnz column index elements.

Figure 2.9 depicts two matrices partitioned with this heuristic, on machines with differing outermost cache size.

Since our heuristic relies on the count of contiguous nonzeroes, we are indeed applying a variant of *cache blocking* (see §2.1). This heuristic does not take into account many other possible factors, such as: the cache line size, the matrix pattern, or whether the submatrix is full rank or not. We leave these points open for a future discussion.

2.3.5 Random Access Operations

Algorithms for many operations on *CSR*/*CSC* are still useful with the recursive *RCSR*/*RCSC* layouts. In the context of Sparse BLAS computations, besides the *SpMV*/*SpSV* kernels, it is also desirable to implement the (*pattern preserving*) nonzero coefficient set/get operations, as well as the set/get operation for the matrix diagonal elements. These operations may be useful in the applications of our interest, so we discuss briefly their implementation.

Since in each quad-tree node of *RCSR*/*RCSC* we store the (submatrix) relative location within the whole matrix, these operations may be trivially implemented by combining traditional *CSR*/*CSC* kernels at the *leaf submatrix* level with a *tree visit* mechanism. For a single nonzero set/get operation, only a single traversal of the tree structure is needed. That is, in addition to the *CSR*/*CSC* set/get operation (involving a single random array access to locate the row/column address, followed by a binary search and the write/read on the found memory location — see §1.4), linked nodes of a tree shall be traversed. The number of nodes to be traversed (and thus, the cost of indirect memory accesses) may vary between leaf submatrices, since it depends on the nonzeroes density in the matrix. By the way, since the leaf submatrices are *sized* proportionally to a constant (a hardware parameter — the outermost cache memory size CS), the worst case for the matrix quad-tree height h can be estimated as proportional to $log_4(nnz/CS)$; that is $O(ln(nnz))$.

For the diagonal set/get operation, each of the submatrices (at leaf level or not, in a recursive formulation) laying on the main diagonal has to be accessed once. A worst case estimate for the number of submatrices to be traversed is 2^h; in any case, this number cannot exceed the submatrices count, which is bound in proportion with the cache size (see §2.3.4).

Other operations (for instance, rows/columns extraction) may be implemented with similar techniques, revolving around the idea of *wrapping* a traditional *CSR*/*CSC* algorithm with a recursive traversal mechanism. The additional cost of traversing the tree structure may be amortized in two ways: first, by the limited number of leaves (roughly cache-sized); secondly, by the smaller (with regards to a full *CSR*/*CSC* array) amount of indices arrays to be scanned by the binary search routines, at the leaf level. Moreover, breakdown in submatrices may allow coarse grained parallelism in heavier operations, like diagonal or block extraction.

2.4 First Experiments with RCSR

In this section we report performance results of a first, basic usage of our *RCSR* partitioning. These experiments are also documented in [MFT+10]. Please refer to §A.1 for the experimental setup: matrices, machines, and adopted methodology.

First, we want to compare the performance of the execution of *SpMV* for *RCSR* to that of *CSR* with a single thread/core. Then, we take a preliminary, simplistic approach to a parallelization, which is limited to two processors/cores. Specifically, we implement parallel *SpMV* by overlapping the computation of two terms in (2.3), using the OpenMP `#pragma omp parallel for` directive (in a fixed loop over a range of two); applied to the upper and lower pairs of matrix quadrants. Thus, the two-core execution of the *SpMV* will spawn two execution threads, of which the first will visit submatrices in the upper two quadrants of the matrix, and the second one will visit the lower two. To get means of comparison, we also run the same experiments using the publicly available *CSB*

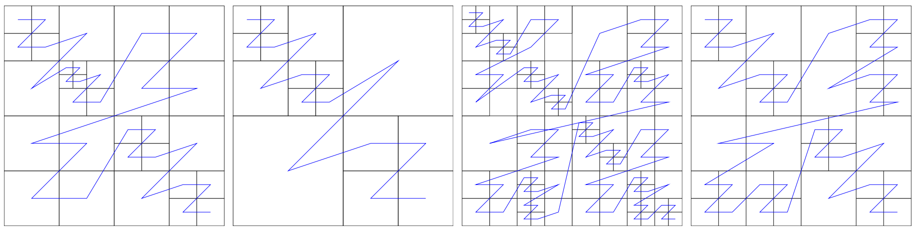

Figure 2.9: Matrices *ASIC_320k* (left two) and *torso1* (right two) δ-partitioned on a 1MB-sized outermost cache machine (**M7**) (first, third from left), and on a 2MB-sized outermost cache machine (**M2**) (second, fourth from left).

(see §2.2) prototypal code[8]. In subsequent chapters we will deal with the topic of a scalable parallelization (that is, for more than two threads) of *SpMV* and *SpSV* for *RCSR*.

Figures 2.10,2.11,2.12 summarize performance data collected running experiments with matrices found in Table A.4, on machines reported in Table A.2. Looking at them, we are interested primarily in:

- scalability of *RCSR*/*RCSC* against that of *CSB* (from one to two cores)
- performance of single core *RCSR*/*RCSC* against non recursive versions *CSR*/*CSC*
- performance of single core *RCSR*/*RCSC* against single core *CSB*
- which matrices perform better for which storage formats
- whether *RCSR*/*RCSC* is better than *CSB* on a particular machine

We observe that:

1) Generally, we find the (double threaded-)scalability of our recursive partitioning comparable to that of *CSB*. *RCSR*/*RCSC* speedup ranges from 1.29 (**M7**, *neos*, *RCSR*) to 1.97 (**M2**, *spal_004*, *RCSR*), while *CSB** both worst (0.91, *torso1*) and best (1.98, *cont11_l*) speedups occur on **M7**. We observe that **M7** favours the 2-core *CSB** code over the *RCSR*/*RCSC*; both in terms of mean speedup (1.68 vs. 1.45) and mean performance

[8]We refer to the archive `csb_code.tgz`, distributed on Aydın Buluç's website, dated July 10, 2009, with md5 checksum `14c12c6c6f0bd548d06b2f6f4b78d118`, sized 19067 bytes.

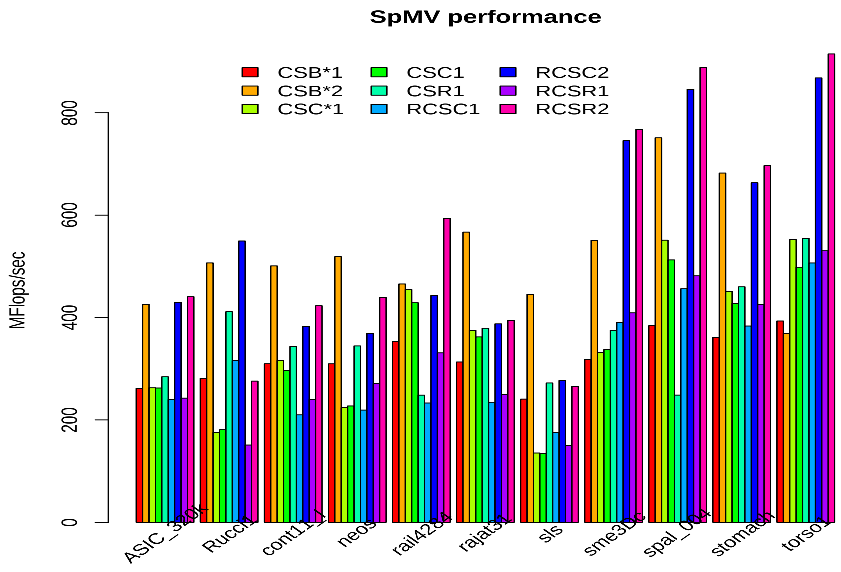

Figure 2.10: *SpMV* performance on **M5**, compared to *CSB*.

(308.9 MFLOPS; +9% more than the *RCSR*/*RCSC*). We conjecture this to be an advantage of CILK++ over plain OpenMP on **M7**'s multiprocessor architecture. On the newer machines (**M2**,**M5**) we observe the two-core *RCSR*/*RCSC* to perform ($\approx$ 450 and $\approx$ 547 MFLOPS vs 407.9 and 525.7) and scale (1.75 vs 1.65) slightly better than *CSB*. Note that our current parallelization strategy does not assure load balance among the two threads: the first level recursive partitioning is influenced by the matrix dimensions only, thus introducing load imbalance for matrices with disparity of nonzero element count between the upper and lower quadrants. However, most of testbed matrices are quite balanced (51% on nonzeroes in the upper quadrant, 49% in the lower one), except for *ASIC_320k*: (57%/43%), and *torso1*: (48%/52%). We observe that notwithstanding this imbalance, matrix `ASIC_320k` scales up well, even better than other matrices.

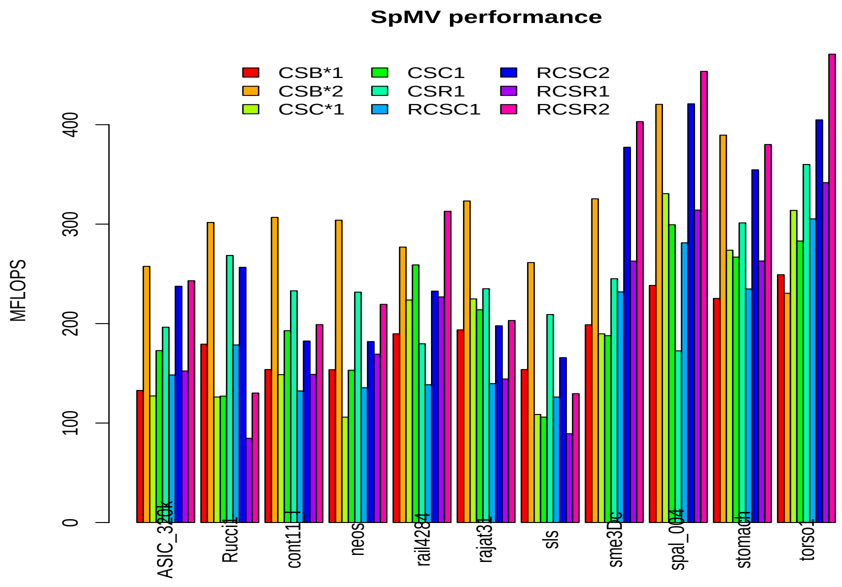

Figure 2.11: *SpMV* performance on **M7**, compared to *CSB*.

2) We observe that the utilization of recursive partitioning usually impairs the performance on single core, when compared to the non recursive counterpart. Consider matrix *Rucci1*. When using *RCSR*, it reaches only about half of the (quite good, on all three machines) performance of *CSR*. The same holds for *RCSC*. This performance drop is justified by the average nonzero per row count; for *Rucci1*, less than 4 elements. Indeed, with the current partitioning policy based on the δ decision function (which does not take in consideration the number nonzeroes per row), a matrix like this, which is quite big (as it exceeds several times the outermost cache size of our machines) becomes partitioned into a significant number of smaller matrices (341 on **M7**, 85 on **M2**), thus increasing both tree traversal overhead, and possibly introducing very scarcely populated matrices, with a consequent high index overhead. Similar arguments hold also for *cont11_l, sls, rajat, neos.*

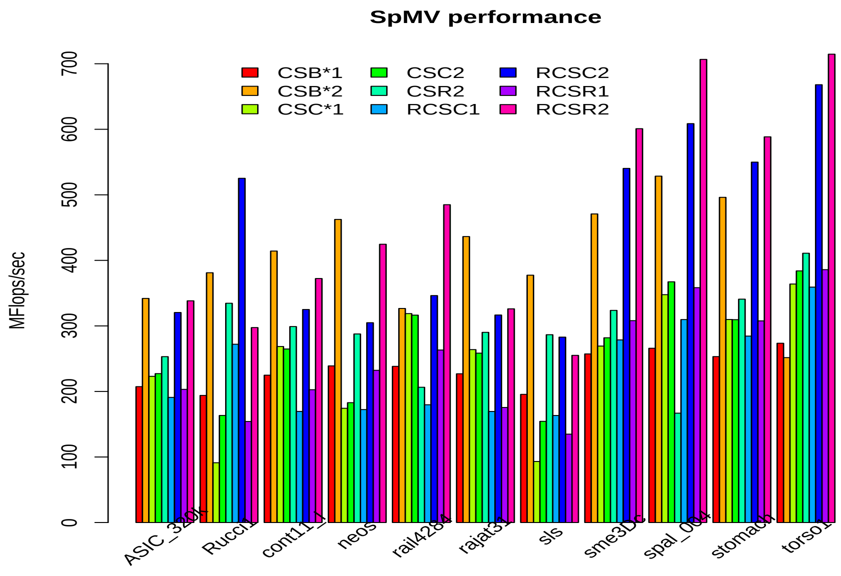

Figure 2.12: *SpMV* performance on **M2**, compared to *CSB*.

3) Similar observations concern *CSB* too, which outperforms *RCSR*/*RCSC* on a single core. The *CSB* format performs better, because while it is based on submatrices partitioning, it does not incur in any recursion overhead.

A possible way to improve performance of the *RCSR*/*RCSC* would be taking in consideration a nonzero per row or per column count based threshold to prevent unnecessary subdivisions (unless the number of partitions is less than the number of computing cores).

4) The best performing matrix on all machines was *torso1*, stored in our recursive *CSR* format, in both single and two cores cases (see, Table 2.1). Indeed, matrices gaining the most from (single or multicore) *RCSR*/*RCSC* are *rail4208, sme3Dc, spal_004, stomach, torso1*. These are also the matrices with highest nonzeroes per row count (as high as 4524.96 for the *spal_004*). On the other hand, we observe that *CSR*/*RCSR* beats *CSC*/*RCSC* in almost all cases (except *Rucci1* and *sls*). The reason is the differing read/write pattern of column and row based *SpMV* kernels. For algorithmic reasons, *CSC* in *RCSC* perform one write per matrix nonzero element (see Fig. 1.16), while *CSR* in *RCSR* perform one per matrix row (see Fig. 1.17). Because both (*Rucci1* and *sls*) matrices are *tall* (rows $\gg$ columns), the higher write rate of *CSC*/*RCSC* is not a problem, as compressing columns rather than rows decreases greatly memory traffic of row indices. This performance behaviour suggests us that comparing the nonzeroes per column to the nonzeroes per row count could give us hints on the memory traffic to be expected from a partitioning. Unlike row and column-based representations, *CSB* is not impacted by these parameters, as at the lower (*cache block*) level, it does not bias toward either rows or columns.

5) As we have observed earlier, measurements collected on **M7** favour the *CSB* format, while machines **M5**, **M2** favour *RCSR*/*RCSC* (see, Table 2.1). This may be a consequence of both good load balancing capabilities and low parallelization overhead of CILK++, as the overhead during the task recreation on the second processor on **M7** should be higher than the one incurred on the two cores involved on **M5** and **M2**.

machine	**best** MFLOPS	**(1 core)** format	matrix	**best** MFLOPS	**(2 cores)** format	matrix
M7	359.9	*CSR*	torso1	470.8	*RCSR*	torso1
M5	554.7	*CSR*	torso1	914.9	*RCSR*	torso1
M2	385.8	*RCSR*	torso1	714.5	*RCSR*	torso1

Table 2.1: Matrices/codes best performing, for each machine in our test set.

2.4.1 Conclusions from the First *RCSR*/*RCSC* Experiment

In §2.4, we have compared the *RCSR* and *RCSC* formats we have introduced in §2.3 to a publicly available, high performance prototype for *SpMV* computations (*CSB*). We have compared the performance of our matrix layout and algorithms, and found them close to that of *CSB*, with a simple single or dual threaded parallelization. An advantage of the approach we propose, over *CSB*, is the adoption of traditional *CSC*/*CSR* ordering for the *recursive sparse blocks* of *RCSR*/*RCSC*. With this ordering, we allow our formats to easily support and adapt well known algorithms originally supported by *CSR*/*CSC*; what is necessary, here, is to write appropriate *recursive wrappers* to the various algorithms. In forthcoming sections, we will develop: a *SpMV* algorithm supporting the parallel execution of more than two threads (see §3.1); parallel triangular solve (see §3.2); tuning techniques for higher efficiency (§4), and parallel matrix build algorithms (§5).

2.5 More Literature and Related Topics

At the beginning of this chapter (see §2) we have introduced, with literature examples, the concept of a hierarchical representation of sparse matrices.

Our approach, as well as that of others, has been motivated by the need for efficient implementations of numerical algorithms. In the field of numerical analysis, a completely different research effort has been carried out for algebraically-meant hierarchical representations of (a class of) matrices, and consequently (computationally) improved methods for the solution of many matrix problems. This research direction is totally uncorrelated to our effort of code and data structures engineering, but is worth mentioning, notably because it results in the application of algorithmic techniques which may seem similar to ours. For a reference, see Hackbusch ([Hac99]).

In both of the *CSB* and *RCSR* expositions we have seen the usage of *space filling curves* for obtaining lower cache miss rates during sweeps of array structures representing sparse matrices. The development of the first space filling curve is attributed to Giuseppe Peano (see Sagan's book [Sag96, p.1]), who was dealing with the problem of finding a *continuous mapping* from the $[0, 1]$ segment to $[0, 1]^2$ (or in general, any two-dimensional region). With the development of digital computing machines, some of such mappings (of course, with a discrete formulation) were considered as favouring efficient access to arrays laid in storage devices. Indeed, G.M.Morton's original work ([G.M66]) dealt with minimizing average latencies when accessing (logically) bi-dimensionally stored *geodetic data*[9] on linearly addressed storage banks, via a custom mapping between geographic coordinates and *data frames* on hard disks.

This mapping has been known as Z-order or Morton-order since then.

Z-order finds use in Data Base Management Systems (DBMS's); see Ramakrishnan and Gehrke's book ([RG, Ch. 28.4]). It was only lately, with the

[9]To be precise, data about 600000 square miles of Canadian territory, in the frame of the *Canadian Land Inventory* project.

increase of memory latencies-to-register access ratios, that such techniques were being employed increasingly for the purpose of memory access efficiency.

Since the computational core of Z-ordering based techniques is bit-manipulation based, the key to an efficient implementation lies often in the efficient use of assembly instructions or machine-specific, low level programming interfaces(see also Arndt's book [Arn10])[10].

We report here a number of studies with Morton/or other space-filling-curve-based arrays order (for short, MO); in [TBK03], Thiyagalingam et al. evaluate MO for dense linear algebra kernels execution; in [LK00], Lawder and King explore MO for multi-dimensional indexing, in the context of a data base management system. There have also been experiments in the systematical or transparent application of such techniques in existent systems/programs. For example, in [JMC05] Jin and Mellor-Crummey evaluate techniques for the efficient enumeration of Morton indices; in [GW04], Gabriel and Wise use a modified C compiler for the transparent usage of Morton-ordered arrays. Finally, Raman and Wise [RW08] give some algorithm for the generation of Z-Morton indices coordinates from two-dimensional ones.

Regarding the use of recursive subdivision techniques for dense linear algebra, there is a number of works by to Gustavson and Wasniewski; for instance, [GRW07]. A study of recursive layouts for dense matrices multiplication is presented by Chatterjee et al. in [CLPT02].

Research efforts most similar to ours can be seen in Gottschling et al. [GWJ08], where recursive layouts for dense matrices are described.

In [PPP04], Park et al. give proofs for cache-obliviousness of recursive (storage) structures. In [PHP03], the same authors compare various block layouts to those of Morton.

We also report the recent works of Yzelman and Bisseling, who use Hilbert curves for sparse matrix–vector multiplication in [YB10]. The same authors develop techniques for reordering and partitioning matrices, leading to recursively subdivided layouts that are somehow similar to that of *RCSR*; see [YB10]. However their later work in [YB10] is focused on cache locality rather than parallelism; so their techniques are only partially comparable to ours.

[10]Or popular on-line resurces on bit-based techniques like `http://www.cs.utk.edu/~vose/c-stuff/bithacks.html` or `http://www.jjj.de/hakmem/`.

3 Shared Memory Parallel Algorithms for Recursively Quad-Partitioned Blocks

Overview

In the previous chapter (§2.3), we have first introduced our *RCSR* layout for sparse matrices, and then, developed (and experimented with) a basic algorithm for performing multiplication by a vector.

In this chapter, we present algorithms for performing the two core **Sparse BLAS** operations (*SpMV* and *SpSV*) on matrices stored using the hierarchical recursive blocks structure we have introduced. These algorithms are much more parallel than the ones used in §2.3.1, as they do not map threads to submatrices statically: threads are bound to *leaf submatrices* at runtime, changing the working submatrix on a dynamical basis during the duration of a single *SpMV* or *SpSV*. Therefore the unit of *workload partitioning* here, is the operation on single leaf submatrices; threads coordinate among themselves in the choice of submatrices via shared variables and a lock structure. This choice, coupled with the choice of matching each submatrix (in terms of its nonzeroes occupation) approximately to the size of the cache memory[1] helps in avoiding an excessive lock overhead. These algorithms make no assumption about the submatrices' internal format/layout, thus allowing future design changes at the leaf level. We have chosen not to use the quad-tree structure information directly in these

[1]We do not give mention here about the *level* of the cache (it is a crucial choice, as we will explain later), but we wish to point out our key idea.

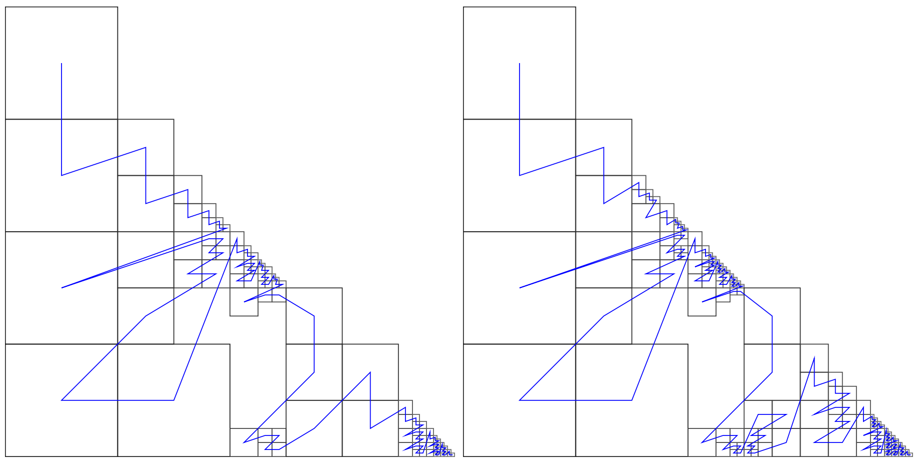

Figure 3.1: Recursive subdivisions of L factors of matrix *g7jac180* (available from [Dav10]) instantiated on machine **M2** (left) and on the same machine, if it had half the outermost cache size (right). Only leaf matrices are shown, with a line joining them.

algorithms — we may use it in the future — now, only *leaf level* information is used.

We present the *SpMV* algorithm in §3.1, and the *SpSV* algorithm in §3.2. Performance results of the two algorithms implementations are presented in §3.3, and we sum up concluding remarks in §3.4. Details of machines and matrices used for these experiments are listed in §A.2.

Figure 3.1 shows the recursive decomposition of the L factor (obtained using SuperLU—see Demmel et al. [DEG+99]) of the *g7jac180* matrix, on different machines. The blue line follows the order (which we call a *balanced Z-order, or* Z^b—see §2.3.3) that the submatrices follow, logically, within the whole matrix.

3.1 Parallel *SpMV*

Our multithreaded (*shared memory parallel*) *SpMV* is presented in Fig. 3.2. It operates on leaf matrices, i.e. on the set of *CSR* matrices at the last level of recursion (which depends on the density of nonzeroes in each submatrix region).

We explicitly form a temporary vector S of references to the actual N leaf submatrices; we also allocate a temporary bitmap B, with one bit for each leaf submatrix. Since nested subdivision guarantees that leaf matrices are pairwise disjoint, we can use B to keep track of the "visited" matrix regions, in the context of a single *SpMV*. We also keep track of the number of matrices visited in a *completed matrices* counter, n, shared among threads. Workload is thus managed among the active threads using the shared bitmap B, the counter n and a lock structure. Each thread repeatedly scans the bitmap looking for workload, until $n = N$. When an unvisited submatrix s is found, a lock is requested and applied to the rows interval on which s is situated (in the original recursive matrix). The lock is necessary because the *SpMV* on s will have to update the output vector y in that range of rows. After the local *SpMV* is completed, the lock on the rows interval of s ([s.roff, s.roff+s.rows]) is released, and the bit corresponding to s is set. Our implementation makes use of *critical sections* (in OpenMP, via the `#omp critical` directive) to control concurrent access to the shared structures. Since the lock operates on individual submatrices row intervals, there is no need to lock each submatrix, but only the selected interval of rows. When another thread will pick an unvisited submatrix s', it will get the lock only if there is no intersection among the row intervals of s and s'. Execution terminates after all of the submatrices have been visited, that is when $n = N$.

For symmetric matrices $A = A^T = L + L^T + D$, we store only the *(non strict)* lower triangle $(L + D)$ elements. For *SpMV*, we apply a variation to the listing in Fig. 3.2. The computation corresponding to the upper triangle $U = L^T$ can be performed using the lower triangle in a "transposed" form. This, however, requires a specialized symmetric *CSR SpMV* code. Moreover, since the symmetric kernel performs both the *SpMV* on s and on $(s^T - D)$ (s transposed, minus the diagonal, if present in s), it updates two intervals of the destination vector y; hence both intervals have to be locked. Note that this *double* lock strategy could impact negatively on the achievable parallel performance.

For this reason, we conjecture that by subdividing symmetric matrices *more* than unsymmetric ones, we would gain some parallelism back from them. Since a detailed analysis of such a trade-off is beyond the scope of this section, we omit its detailed investigation.

Note that the given *SpMV* algorithms do not specify any particular order in visiting the leaf matrices; threads are free to cycle among submatrices repeatedly looking for "available" submatrices. In practice, this is not a big waste of resources: for each leaf submatrix, we allocate a single bit in the bitmap B, and a pointer (possibly with offset and dimension indices) in S. Since each leaf submatrix is likely to occupy $O(CS)$ (CS being the outermost cache size) bytes,

Figure 3.2: Multithreaded *SpMV* for leaf submatrices of a *RCSR* matrix.

```
1  S ← $[s_0, s_1, \ldots, s_{N-1}]$ /*an array of terminal submatrices, in any order*/
2  B ← $[0, 0, .., 0]$ /*a zero bit for each submatrix*/
3  n ← 0 /*count of visited submatrices so far*/
4  while $n < N$ do
5      begin parallel
6      $s$ ← pick an unvisited submatrix $s$ from S
7      /*(should have picked up $s \leftarrow S[i]$, with $B[i] = 0$)*/
8      $[f, l]$ ← [s.roff , s.roff+s.rows]
9      if locked($[f \ldots l]$) then cycle
10     lock($[f \ldots l]$) /*we lock $y$ on s's rows interval*/
11     /*perform SpMV on s and $x$[s.coff:s.coff+s.columns] into $y[f : l]$*/
12     $y[f : l] \leftarrow y[f : l] + s \cdot x$[s.coff:s.coff+s.columns]
13     $B[i] \leftarrow 1; n \leftarrow n + 1$
14     unlock($[f \ldots l]$)
15     end parallel
16 end
```

the memory traffic associated in accessing B, when looking for submatrices that are "available" is negligible; in most cases the bitmap will fit in the first level cache, and scanning repeatedly through it will not stress the memory hierarchy. Repeated scans of S, instead, might cause overhead; however, actual data arrays of submatrix s, at index i in S is only needed in the case when $B[i] = 0$, and this last memory access has a high hit probability (since a lock on the interested output vector intervals is the only remaining constraint preventing the usage of that submatrix). A possible resource-wasteful situation would be a repeated lock contention on behalf of a single thread, when the rows lock is not available; this situation would lead to the overuse of cache snooping circuitry among cores/CPUs. This is not expected to be a problem (on current architectures, employing variants of the MESI (see Drepper [Dre07, §3.3.4]) cache coherence protocol), since this situation would imply that other threads are busy performing the *SpMV*, and thus likely not to have any shared variable cached. Obviously, this problem is exacerbated when the last submatrices are visited, and there is no more actual work to be available; thus, it could be detected by comparing the n counter with N and the available work items.

It should be stressed that the proposed approach will also work, with minimal modification, for the transposed case (employed in iterative methods such as BiCG or QMR—see Barrett et al. [BBC+94]), whereas with a *CSR* representation, a parallel transposed *SpMV* would be challenging. With our approach, (or generally, with any coarsely blocked format; see Buluç et al. [BFF+09, s.1] for a brief discussion) this task is more likely to be efficient.

3.2 Parallel *SpSV*

Let us look at the recursive breakdown of the *lower triangular solve* operation ($x \leftarrow L^{-1}x$):

$$\begin{vmatrix} x_1 \\ x_2 \end{vmatrix} \leftarrow \begin{vmatrix} L_1 & 0 \\ M & L_2 \end{vmatrix}^{-1} \begin{vmatrix} x_1 \\ x_2 \end{vmatrix} \Rightarrow \begin{vmatrix} x_1 \\ x_2 \end{vmatrix} \leftarrow \begin{vmatrix} L_1^{-1} x_1 \\ L_2^{-1}(x_2 - ML^{-1}x_1) \end{vmatrix} \tag{3.1}$$

Decomposition Fig. 3.1 is quite straightforward, but without any further structure, it offers limited support for parallelism, as this would be only possible *within* the *SpMV* operation occurring in "$x_2 \leftarrow L_2^{-1}(x_2 - M(L^{-1}x_1))$". This dependency requires that the *SpSV* computation on the diagonal blocks can only be performed after all blocks on its left have been visited by the *SpMV* computation. Thus, our *SpSV* algorithm also operates on leaf matrices only, ignoring the intermediate matrices of the recursive structure. Note that we make explicit use of the fact that the recursive partitioning results in square diagonal blocks. Listing 3.3 outlines the *SpSV* algorithm, which operates on the leaf matrices of a recursively partitioned matrix; this algorithm could be applied with any matrix partitioning resulting in disjoint submatrices which are square on the main diagonal.

Observe that in this algorithm, as a first thing, we sort leaf submatrices in a way that allows to perform the *SpSV* without any sophisticated data structures. As the listing shows, each submatrix not on the diagonal is involved once in the *SpMV* kernel. On the other hand, the *SpSV* kernels are executed on the diagonal submatrices only. Since the core update in an in-place lower triangular solve on a matrix L and vector x is "$x_i \leftarrow (x_i - \sum_{j=1}^{i-1} x_j L_{ij})/L_{ii}$" there is a *horizontal dependency* (*SpMV*), which must be satisfied before performing *SpSV* on the diagonal blocks. Formally, the sorting criteria for a pair (s, s') of submatrices follows the total order defined as: (i) if any one of the matrices (say, s) lies on the diagonal (as mentioned before, due to recursive subdivision, a submatrix intersecting the diagonal is necessarily square), then it comes after s' only if the last row of s' is less than or equal to the last row of s; (ii) if neither of s, s' lies on the diagonal, the one with the smaller last column index comes first, with ties broken according to the smaller first row. With this ordering, if the submatrix

Figure 3.3: Multithreaded Lower Triangular Solve for an *RCSR* Matrix

```
S $\leftarrow [s_0, s_1, \ldots, s_{N-1}]$ /*a sorted array of terminal submatrices*/
B $\leftarrow [0, 0, .., 0]$ /*a zero bit for each submatrix*/
D $\leftarrow [d_1, d_2, .., d_{N-1}]$ /*dependencies, for each submatrix*/
n $\leftarrow 0$ /*count of visited submatrices so far*/
while $n < N$ do
    begin parallel
    $s \leftarrow$ pick an unvisited submatrix $s$ from S (say $s \leftarrow S[i]$, $B[i] = 0$ )
    $[f, l] \leftarrow$ [s.roff , s.roff+s.rows]
    /*pick another submatrix if this row interval is locked*/
    if locked($[f \ldots l]$) then cycle
    if $s.roff = s.coff$ then
        if $B[D[i]] = 1\& \ldots \&B[i-1] = 1$ then
            lock($[f \ldots l]$) /*s is a diagonal block ; we lock x on its rows*/
            /*perform SpSV on s*/
            $x[f : l] \leftarrow s^{-1}x[f : l]$
            $B[i] \leftarrow 1; n \leftarrow n + 1$
            unlock($[f \ldots l]$)
        else
            cycle /*pick another submatrix */
        end
    else
        if $B[D[i]] = 1$ then
            lock($[f \ldots l]$) /*s is not a diagonal block; we lock x on its
            rows*/
            /*perform SpMV on s*/
            $x[f : l] \leftarrow x[f : l] + s \cdot x$[s.coff:s.coff+s.columns]
            $B[i] \leftarrow 1; n \leftarrow n + 1$
            unlock($[f \ldots l]$)
        else
            cycle /*pick another submatrix */
        end
    end
    end parallel
end
```

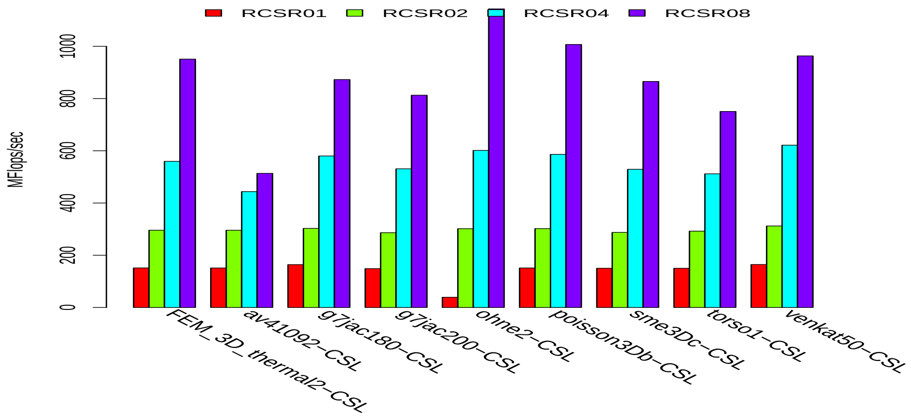

Figure 3.4: *SpMV* performance on **M1**, L factor matrices.

s_i is on the diagonal, it can be visited only after all of the matrices s_j, with $j < i$ were visited. In particular, if s_i and s_j are both on the diagonal, $j < i$, and the last row of s_j comes immediately before the first row of s_i, we say that *s_j is a dependency of s_i*[2] . As far as the implementation is concerned, we put dependency information in a temporary, shared vector D, which we compute by scanning S. The actual operation is in a way similar to the *hybrid parallel triangular solve by block anti-diagonals and block columns*, proposed by Mayer in [May09]. The main difference is that our approach is "implicit," as threads run through "available" matrices and "parallel zones" are unlocked only after portions of the solution are solved.

3.3 Experimental Results for *SpMV* and *SpSV*

The bar plots in this section report the performance in MFlops (millions of floating point operations per second) for a specific matrix, algorithm and number of utilized cores[3].

[2]This is an instance of *topological sorting*—see Knuth [Knu97, § 2.2.3/p.261] or Cormen et al. [CLRS09, § 22.4].

[3]Please refer to §A.2 for a full description of the experimental setup we used.

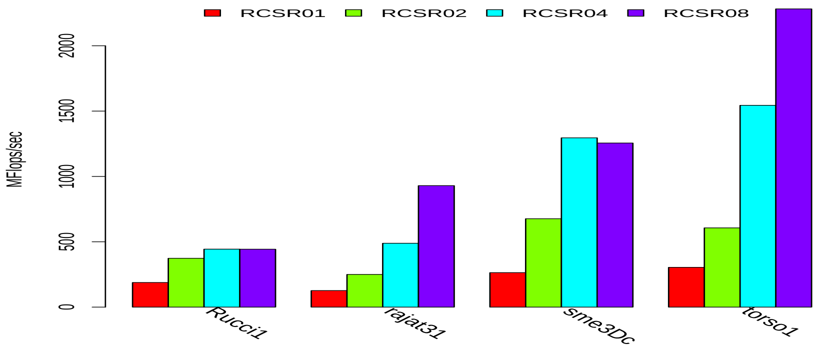

Figure 3.5: *SpMV* performance on **M1**, unsymmetric matrices.

As seen in section 3.2, the *SpSV* algorithm we propose is based on the use of *SpSV* and *SpMV* kernels for the *CSR*; hence an efficient *SpMV* is needed for an efficient *SpSV*. During our experimental runs, we also collected single-threaded performance of the plain *CSR SpSV* and *SpMV* on the triangular matrices. We found that: i) *CSR SpSV* performance was slightly higher (by no more than 5%) than that of *CSR SpMV*; ii) *CSR SpMV* usually (but not always) outperformed the single-threaded *RCSR* by a few percent. Considering the *SpMV* on **M2** (Fig. 3.6,3.5) and **M1** (Fig. 3.9,3.8), when executing 1 or 2 threads, we notice a stable performance level (almost) regardless of the matrix. For 4-8 threads, we see more variation, as there is more memory channel usage, and the memory access pattern becomes less regular (remember the naive nature of the algorithm in Fig. 3.2). On **M3** (Fig. 3.12,3.11), we do not observe such regularities, and we witness what seems a memory bottleneck when moving to 4 and 8 threads. The symmetric kernels (Fig. 3.12) encounter the first scalability problems on 4 threads, as they saturate the memory channel at a write-to-read rate which is double respect to unsymmetric kernels. On the other hand, on **M2** and **M1**, the performance of the *SpMV* on symmetric matrices grows up to 8 threads (Fig. 3.6,3.9). **M1** is the only machine able to achieve nearly linear speedup for the *SpMV* kernels. Compared to *CSB*, *RCSR* performs better with a smaller number of threads, but then encounters a scalability (and performance)

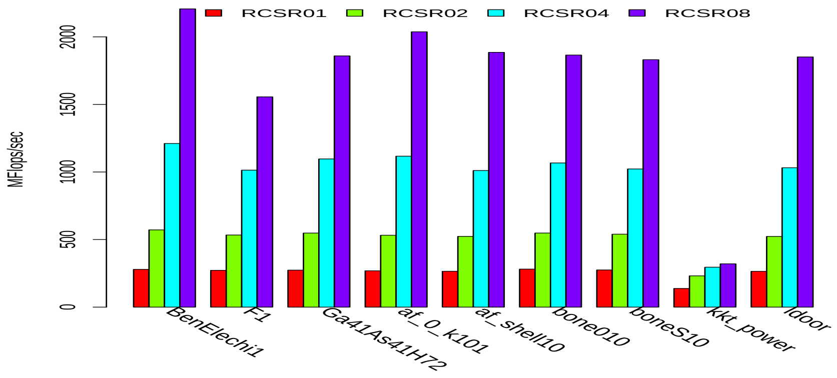

Figure 3.6: *SpMV* performance on **M1**, symmetric matrices.

limit before *CSB*. Note that on matrix *torso1*, *RCSR* performs better, just as we experienced in §2.4 with a different parallelization strategy. The reason for these performance patterns in *RCSR* and *CSB* can be explained by: i) the use of *shorter* indices in *CSB* (See Buluç [BFF+09]), which in many cases alleviate the memory bandwidth bottleneck; ii) regular access pattern in the *CSR* kernels operating on submatrices, leading to high performance at the cost of earlier limits from memory bandwidth.

There are cases in which performance is bad across the board; this is particularly true for matrices that have as few as 4 elements per row (*kkt_power*, *Rucci1*, *rajat31*). On these matrices, loading each right-hand side vector element requires fetching of an entire line of cache, and with little or no spatial locality, this is far too expensive. On **M1**, we witness a seemingly superlinear (Fig. 3.10) scaling in the *SpMV* for the matrix *ohne2*: the overall results on **M1** indicate bad behaviour of serial *RCSR*, but this will need further investigation.

Looking at the results of the *SpSV* we note that despite the similarity of the algorithms in Fig. 3.3 and 3.2, the performance scaling is sublinear, even on the **M1**. This is not surprising (see also Mayer's considerations in [May09]), as good performance of the parallel *SpSV* depends on the structure of the lower triangular matrix L; it must have a sufficient amount of *parallel regions*. In our case, matrices should have enough off-diagonal submatrices, to parallelize the *critical*

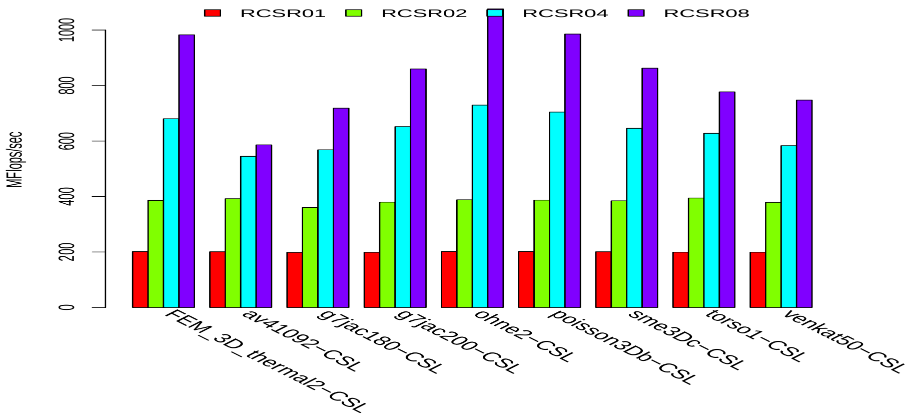

Figure 3.7: *SpMV* performance on **M2**, L factor matrices.

path computations. While not astonishing, the observed performance speedups — approximately 3 on **M1**(Fig. 3.13), to 2.5 on **M2** (Fig. 3.14), and up to 1.5 on **M3** (Fig. 3.15) — conform to those reported in Mayer [May09]. As an example of *a difficult* matrix, consider the L-matrix obtained from *venkat50*. After the *LU* decomposition, the majority of its nonzeroes are in the diagonal blocks, and most blocks are on the main diagonal. For such matrices ("almost banded"), computation involves solving the diagonal subsystems serially, following with almost-serial operations on the few off-diagonal submatrices. Matrices likely to achieve a reasonable speedup from the multithreaded *SpSV* are the larger ones, as their fraction of submatrices located on the diagonal is smaller. For instance, the L factor of the matrix *ohne2*, on **M2** (Fig. 3.13), is broken down in 1601 submatrices, of which only 10% are located on the diagonal. As a consequence of this, we get an almost 3-fold *SpSV* speedup on **M1**.

3.4 Conclusions

The results presented in this chapter show the potential of the *RCSR* storage format for the implementation of main kernels (*SpSV* and *SpMV*) of a sparse level 2 BLAS. For both operations our unified approach was found to be competitive,

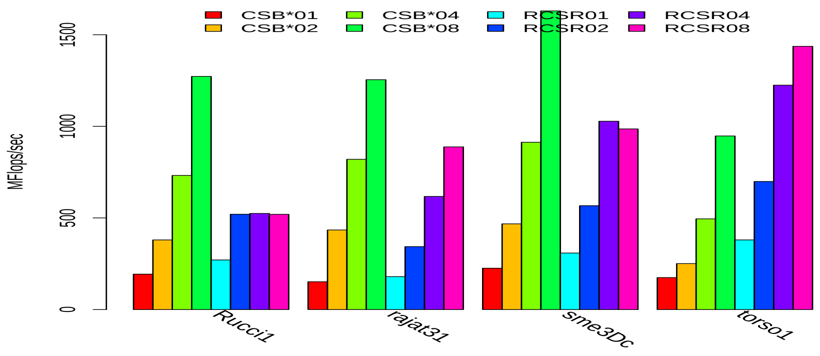

Figure 3.8: *SpMV* performance on **M2**, unsymmetric matrices.

against approaches based on specialized data structures (see Mayer [May09] and Buluç et al. [BFF+09]). It has to be stressed that we did not employ any fine-tuning to the basic versions of the proposed format and algorithms. At the same time, there exist many possibilities for modifying both the algorithms and the data format itself, that are likely to improve the performance of the *RCSR* without impairing its generality and/or functionality.

Here, observe that the sparsity pattern of matrices is a determinant factor of parallelism in the *RCSR*: lower banded triangular matrices tend to limit parallelism of the *SpSV*, which can easily use all the memory bandwidth available, especially if the matrix sparsity pattern is very irregular. One possible approach to limiting stalls due to insufficient memory bandwidth is the use of *shorter* column indices in *CSR* leaves; this will be explored in §4.1. We notice that in a number of cases, our *RCSR* format has encountered scaling problems. We will look for improvement in chapter §4.

In the perspective of many-core environments (and in light of the *hypersparsity* property; see Buluç et al. [BG08]), approaches other than using *CSR* leaves may be advantageous, too. In §4.3 we will evaluate the usage of heterogeneous *leaf submatrix* formats.

Another possibility would be the parallelization of the execution of the leaf *SpMV* dependencies in *SpSV* by a different row locking strategy.

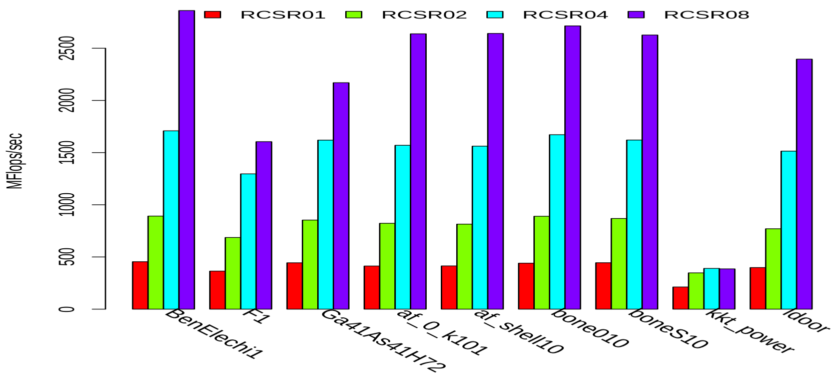

Figure 3.9: *SpMV* performance on **M2**, symmetric matrices.

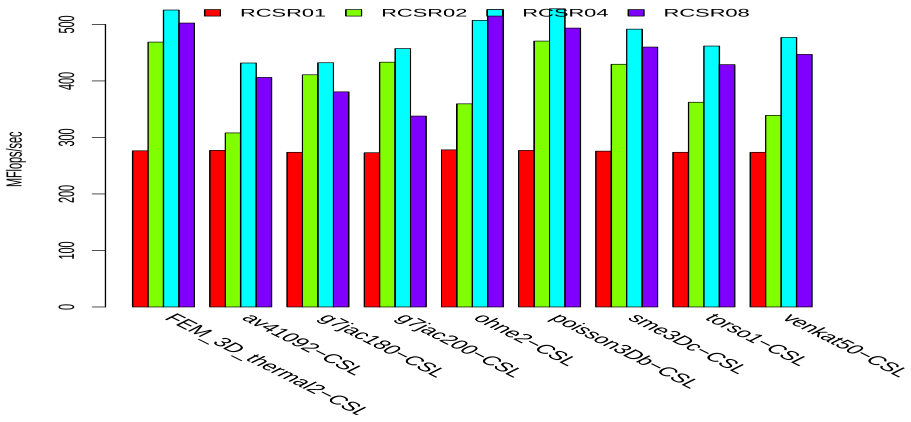

Figure 3.10: *SpMV* performance on **M3**, L factor matrices.

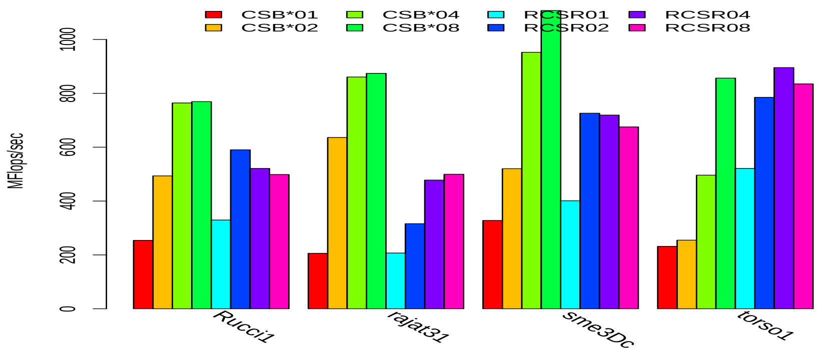

Figure 3.11: *SpMV* performance on **M3**, unsymmetric matrices.

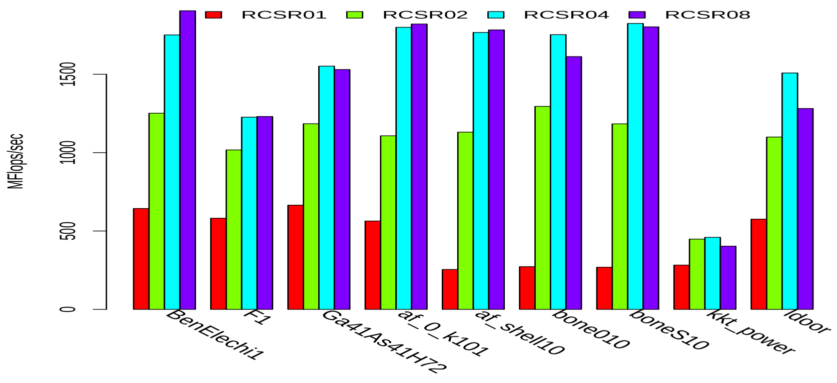

Figure 3.12: *SpMV* performance on **M3**, symmetric matrices.

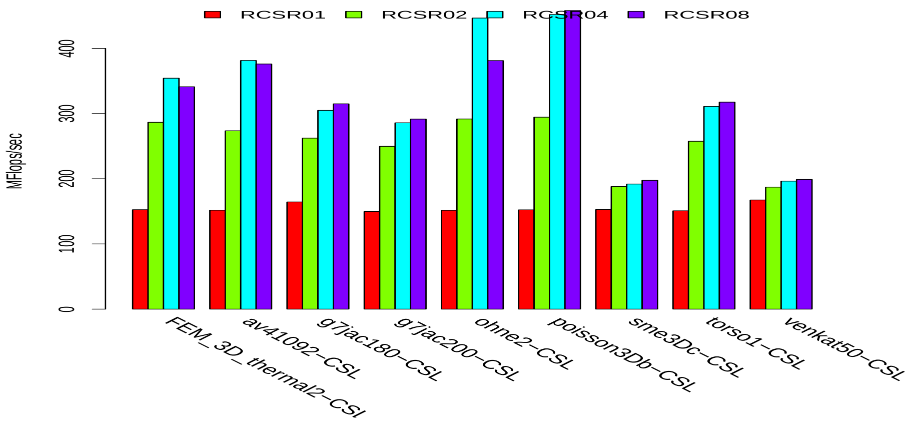

Figure 3.13: *SpSV* performance on **M1**, L factor matrices.

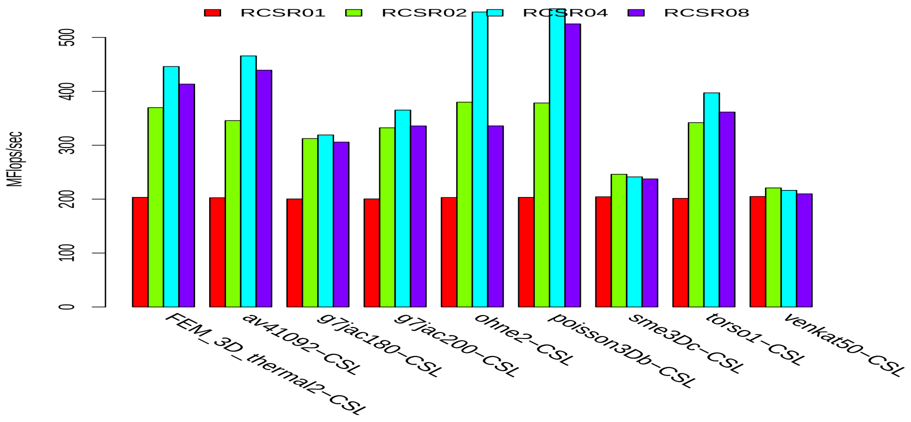

Figure 3.14: *SpSV* performance on **M2**, L factor matrices.

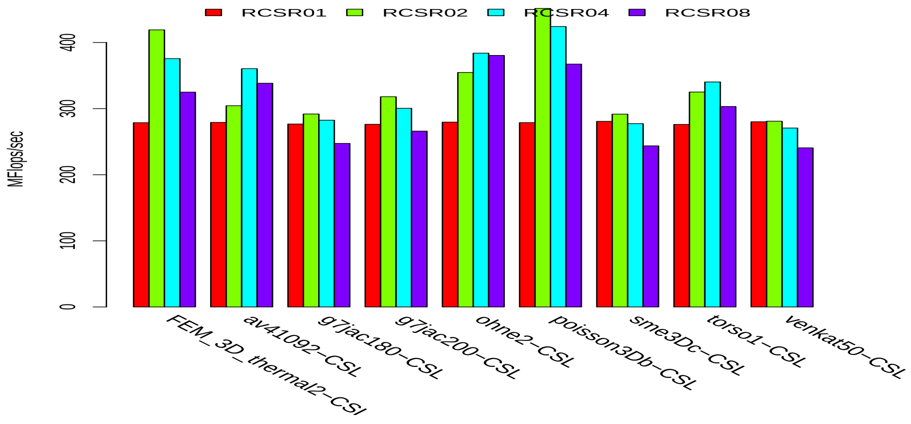

Figure 3.15: $SpSV$ performance on **M3**, L factor matrices.

4 Tuning *RCSR*: Recursive Sparse Blocks

Overview

In this chapter, we develop two modifications to the *RCSR* format. The goal of these modifications is the improvement of time efficiency of *SpMV*/*SpSV* operations, without making substantial changes to the underlying serial kernels implementation.

In §4.1, we propose a technique for reducing the *memory footprint* of the *SpMV* algorithm by employing a shorter type for coordinate indices, even on large matrices. In §4.2, we look at the experimental results of this modification. In §4.3 we introduce a second modification: we allow for some of the submatrices to be stored in a row-major coordinate format, and in §4.4 we look into the experimental results of this modification. In §4.5 we draw some combined conclusions for the two proposed modifications.

4.1 Reducing Index Usage in RCSR with Short Indices

4.1.1 Recursion Stop Criteria, Revisited

At the heart of *RCSR* lies the mechanism for the recursive partitioning. As we have seen in §2.3.4, given an input matrix in a standard *coordinate* (*COO*) storage, we subdivide it recursively into four quadrant submatrices. We terminate the recursion (and consolidate leaf submatrices) only when a specific condition is reached. Here, we modify the *recursion stop decision* function δ presented in §2.3 to δ_h, as in:

$$eab(m, k, nnz, ES, WS) \overset{def}{=} \tag{4.1}$$

$$ES(2 \cdot nnz + m) + WS(m + nnz) \tag{4.2}$$

$$\delta_h(m, k, nnz, CS, ES, WS) \overset{def}{=} \tag{4.3}$$

$$\textbf{True},\ if(\ nnz \cdot ES > 2 \cdot CS \textbf{ and } m > 2^{16} \textbf{ and } k > 2^{16}), \tag{4.4}$$

$$otherwise \tag{4.5}$$

$$\textbf{True}, if(eab(m, k, nnz, ES, WS) > \alpha \cdot CS \textbf{ and } nnz/m > \mu), \tag{4.6}$$

$$otherwise \tag{4.7}$$

$$\textbf{False} \tag{4.8}$$

Here, an $m \times k$ submatrix with nnz elements is considered a candidate for subdivision. Some constants that are involved are: ES (*element size*) is the byte occupation of a single matrix nonzero entry; WS (*word size*) is the byte occupation of a full index element (4 bytes); CS (*cache size*) is a machine parameter (see below); μ is the (*minimum nonzeroes per row*) limit, assuring that the block is not too sparse (3 in our experiments).

The δ_h function is structured to take into account the combined effect of the submatrix visits as well as the traffic on the right-hand side, and the result vectors. Note that with this formulation the decision is essentially independent of the thread-count. Contrary to §2.3.4, for the CS parameter we take into account the size of the L2 cache (instead of the L3 cache). This is motivated by the increased number of cores we are going to run our algorithm on. Here, partitioning the matrix into very big chunks may cause higher contention for cache lines when accessing the x and y vectors arrays. On the other hand, sizing leaves around the L2 cache size may result in smaller leaves. Thus, we are trading off a possibly higher overhead for better cache reuse potential. We remind the reader that in *CSR*, accesses to the numerical values and the column indices arrays proceed unidirectionally. As a result, prefetch engines make often perfect predictions, but prefetched indices are used only once (per result vector). On the other hand, caching of the right-hand side and the result vectors would be desirable, but is difficult to achieve, because of the unpredictable access sequence caused by the pattern of nonzeroes in the input submatrix (recall the discussion in §1.2). Because of these reasons, it is difficult to determine, or even to define, an "optimal" CS parameter; thus the cutoff function δ_h, above, would need further study which is beyond the scope of this experiment.

4.1.2 Support for 16 bit Indices

The memory requirement for index arrays of a square $d \times d$ matrix with n nonzeroes, stored in the *CSR* format, with an 8-byte numerical type (like `double` or `float complex`) and 32-bit indices is $4 \cdot d + (8+4) \cdot n$ bytes. If instead of using 32 bits, we store the column indices using 16 bits, the requirement would become $4 \cdot d + (8+2) \cdot n$ bytes. Therefore, for $n \gg d$, this means a saving of $\approx 2/12 = 16\%$.

For a `float` numerical type, the advantage could be even greater: moving from $4 \cdot d + (4+4) \cdot n$ to $4 \cdot d + (4+2) \cdot n$ bytes means savings as high as $\approx 2/8 = 25\%$. Since the standard in engineering and scientific applications is double precision floating point numbers, we will report experiments with this type.

In order to use 16 bit indices on leaf arrays, we have two choices: (1) make sure that recursive subdivision proceeds until all leaves are dimensioned under 2^{16}, or (2) accept having some "emptier" but "large-dimensioned" leaves stored using 32 bit column indices. The first approach has the advantage of keeping all the *nnz* indices small, but has the flaw of potentially abusing the subdivision process and leading to an excessive number of submatrices. This, in turn, would imply emptier sub-rows, and potentially more indexing space wasted in *row pointers* arrays, with an outcome of using more indexing than before the modification. The second approach allows cases which don't benefit from the 16 bit variation. We chose a mid-way approach: we force subdivisions as long as matrix indices don't fit into 16 bit indices and nonzero elements occupation is relevant; after that subdivisions are still allowed, but with stricter rules. Note that even with this approach:

- Some matrices may not use 16 bit indices at all (if they do not contain enough nonzeroes to be split, but have a big dimension).
- The potential *overall* bandwidth saving could be more than 16% (if we consider that extra horizontal splits could prevent from storing the row indices of submatrices with empty *lower quadrants*).

In the cutoff function δ_h, we also prevent matrices with an excessively small nonzeroes/rows ratio (μ) from being further subdivided.

In the following, we call *RCSR* the format of matrices determined by the cutoff function δ_h without line 4; we call *RCSRH* the format of matrices determined by the full cutoff function δ_h, with submatrices *suitable* for 16 bit indices.

Please note that algorithms for accessing randomly the individual matrix nonzeroes as discussed in §2.3.5 are still valid for *RCSRH*, as long as appropriate 16 bit indices *CSR* algorithms are implemented.

4.2 Experimental Evaluation of RCSR with Compressed Indices

To have an insight into the implications of the modified subdivision heuristic described in §4.1.1, let us analyze the results of our experiments. In the following, we are concerned with the transition from *RCSR* (32 bit column indices for the Recursive *CSR*) to *RCSRH* (*RCSR* with *CSR* leaves that have 16/32 bit column indices), with emphasis on results for largest numbers of cores and scalability. For the sake of comparison with **M2**, we used only 8 out of the 12 cores available on **M4**. Note also that here we are not primarily concerned with absolute performance values, although we will comment on them in the most interesting cases.

Please refer to §A.3 for a full description of the experimental setup we used.

4.2.1 Unsymmetric Matrices

We depict performance on machines **M4** and **M2** for square matrices in Fig. 4.2 and 4.4, and for non-square ones in Fig. 4.1 and 4.3.

With the usage of *RCSRH* we notice:

- For most matrices we have a performance improvement;
- Some matrices show poor performance;
- Some matrices have better scalability, but lower absolute performance;

Among the ones which do scale and improve with use of 16 bit indexing, on **M4** (Fig. 4.2 and 4.1), for the non-square matrices, we gain: 25% for *c8_mat11_I*, 20% for *rail2586*, 15% for *spal_004*. For the square matrices on the same machine we gain: 17% for *venkat01*, 30% for *rma10*, 19% for *sme3Dc*. Since matrices *rma10, c8_mat11_I* and *venkat01* are only few times the size of the L3 cache and we measure performance with *hot caches*, their speedup is probably higher due to a limited cache reuse phenomenon across *SpMV*'s. It is thus less representative than that obtained for bigger matrices: *sme3Dc*, *rail2586*, or *spal_004*.

On **M2** (Fig. 4.4 and 4.3), we observe a similar performance pattern. The benefits seem slightly milder here, though. The highest improvements are gained for matrices *rma10* (11%), *raefsky3* (10%) and *c8_mat11_I* (9.5%).

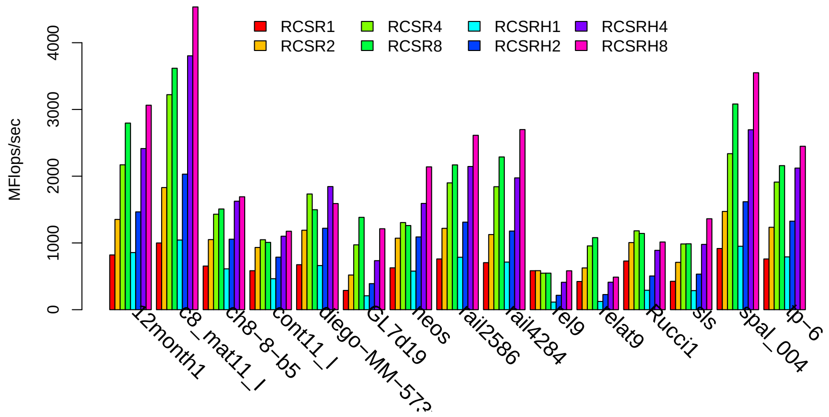

Figure 4.1: *SpMV* performance on **M4**, rectangular matrices.

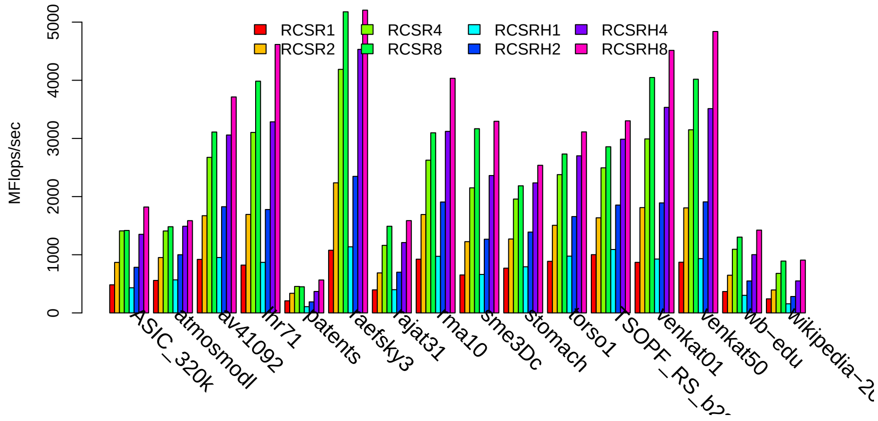

Figure 4.2: *SpMV* performance on **M4**, square matrices.

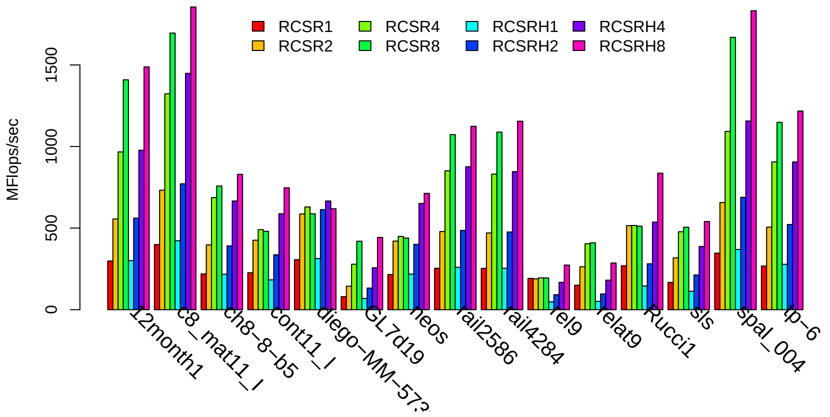

Figure 4.3: *SpMV* performance on **M2**, rectangular matrices.

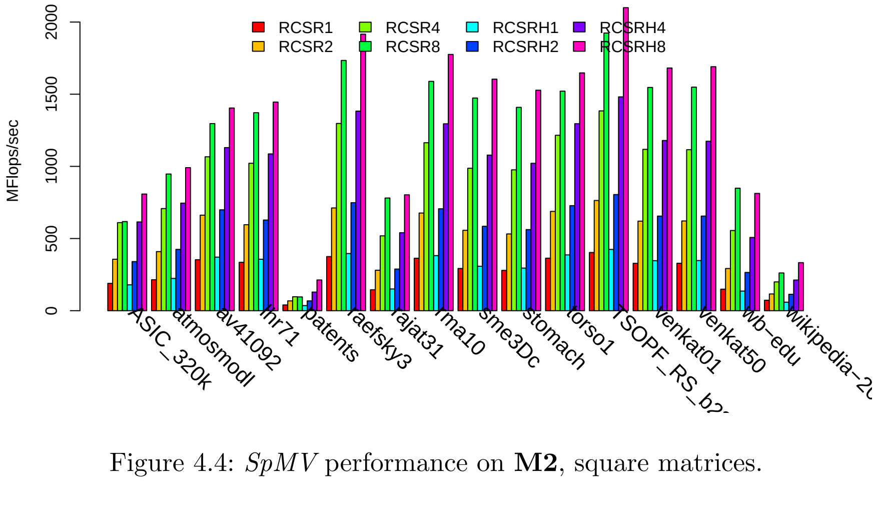

Figure 4.4: *SpMV* performance on **M2**, square matrices.

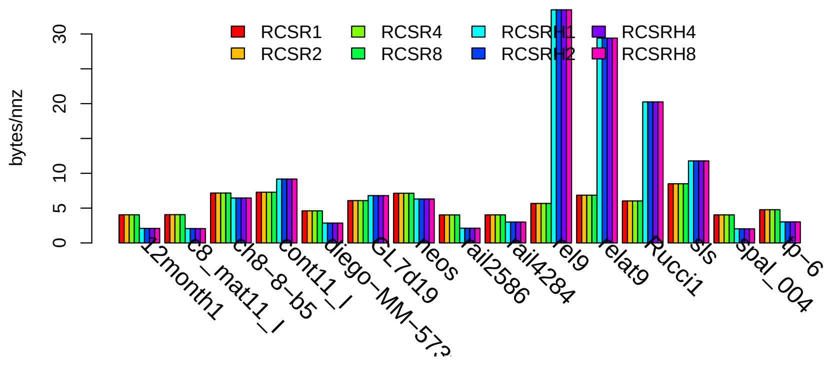

Figure 4.5: Index usage (bytes per nonzero) on **M4**, rectangular.

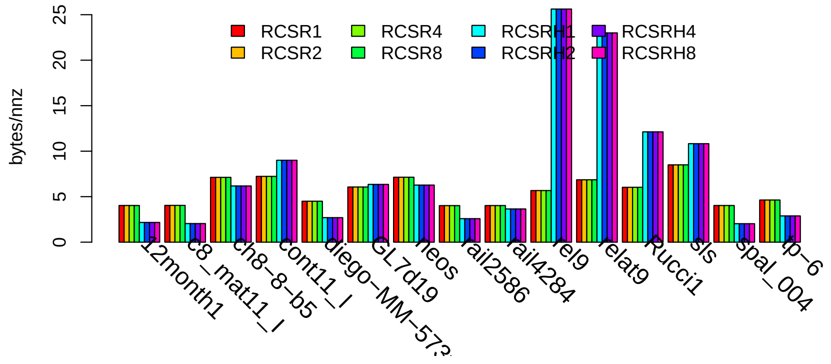

Figure 4.6: Index usage (bytes per nonzero) on **M2**, rectangular.

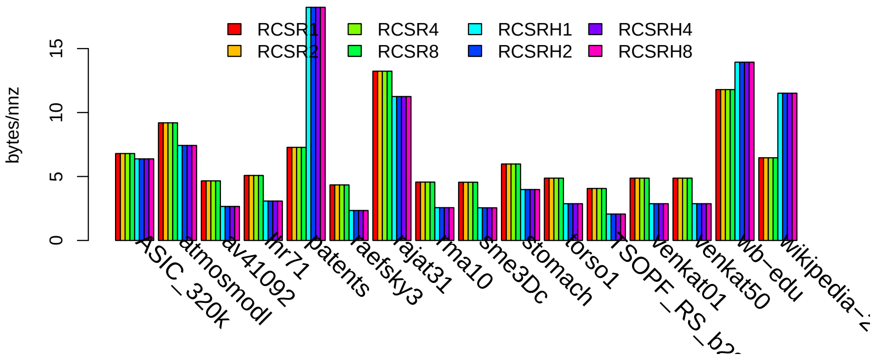

Figure 4.7: Index usage (bytes per nonzero) on **M4**, square.

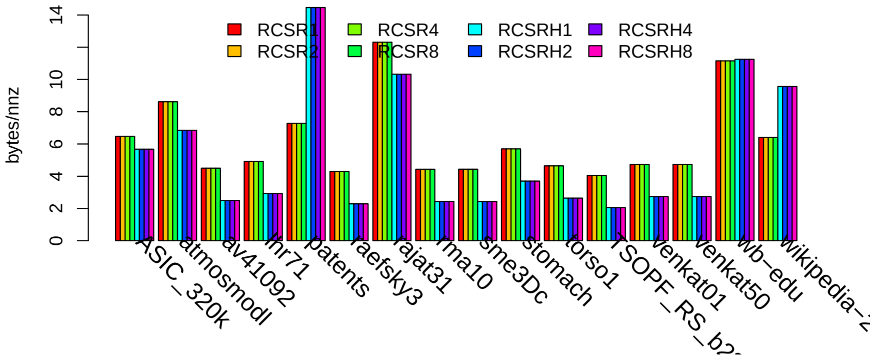

Figure 4.8: Index usage (bytes per nonzero) on **M2**, square.

Figures 4.7.4.8,4.5,4.6 show the relative index usage (column and row pointer indices byte usage per nonzero element) for non-symmetric matrices on both machines. There, we observe that the aforementioned matrices, which both scale and gain from index compression[1] have less index overhead when stored as *RCSRH*. The saving on indexing storage in these matrices is about 50%, which means that almost all of their submatrices have been converted to use 16 bit column indices. Savings near to 50% mean a high relevance of shorter column indices to row pointers (which remain at 32 bit). In fact this was possible because of the high nonzeroes/rows ratio for these last matrices (see Table A.8): 539 for *c8_mat11*, 3097 for *rail25986*, 4524 for *spal_004*, 73 for both *sme3Dc* and *torso1*. Matrices with high nonzeroes/rows ratio are also the ones on which *RCSR* results in better performance.

We consider the obtained 10%..16% improvements reasonable: the saving in index overhead reduces stalls, thus allowing for more input to the arithmetic units, and thus higher floating point throughput.

Let us now consider the cases where there was almost no performance gain. Here we deal with matrices which (a) don't scale either in *RCSR* or in *RCSRH*, (b) ones which scale only in *RCSRH*, and (c) ones which lose performance when going from *RCSR* to *RCSRH*.

Weak scaling for *RCSR* may originate from a combined effect of limited bandwidth and bad partitioning. A bandwidth bottleneck is what usually limits scaling in matrices which are partitioned into "too many" leaves (e.g. *Rucci1* in *RCSRH*, *tp-6*). Let us take a closer look at the number of leaf submatrices (due to space limitations, we report only the selected cases) of those matrices which show poor or weak scaling in *RCSR*. On **M4**, these are: *patents, rajat31, wb-edu* among the square ones, and *cont11_l, diego-MM-573x230k, rajat31, cage15, neos, patents, rel9, relat9, Rucci1* among the non-square ones. Except for *diego-MM-573x230k* and *cage15*, all of them have a nonzeroes/rows ratio smaller than 6, which is quite low. While a low nonzeroes/row ratio is not *per se* a reason for limited scaling, if these rare nonzeroes are not compactly distributed within the matrix, index overhead could be high due to the subdivisions (which introduce row pointer arrays), thus preventing acceptable scaling. Another consequence of a low nonzeroes/row ratio is that the partitioning function δ_h tries to prevent degenerate cases with submatrices too sparse. Therefore, it could happen for such matrices, especially among the smallest ones (even if outgrowing the L2 cache dozens of times), that they get subdivided into a number of submatrices

[1] Here, we use the term *index compression* as a shorthand for *index representation overhead reducing*. A more appropriate use of the term would be in the context of specific *encoding* techniques on indices; see Kourtis et al. [KGK08].

not large enough to scale with the available threads.

For instance, on both machines, *Rucci1* gets partitioned by *RCSR* into 4 leaves only. When using *RCSRH*, *Rucci1* gets subdivided into 64 leaves on **M2**, and 256 on **M4**. These subdivisions on **M4** are enough to make *RCSRH* performance drop below that of *RCSR* (Fig. 4.1). The performance drop is a consequence of the very short sub-rows induced by subdivisions, which almost quadruple the indexing overhead (Fig. 4.5). At the same time, on **M2** the indexing overhead almost doubles (Fig. 4.6), but here, the bad scalability of *RCSR* prevents the code from running efficiently with more than 4 cores on *Rucci1*, and *RCSRH* results in being faster. An optimal handling of this situation is hard to achieve.

This phenomena occur on the matrices which can be regarded as "badly partitioned" by δ_h. Eight core performance of *RCSRH* on *cont11_l, neos, rel9, relat9* (Fig. 4.3) outperforms *RCSR* only because the extra subdivisions allow having more submatrices than cores (and thus, achieve scalability), while *RCSR* is stuck with too few subdivisions. On **M4**, the same matrices show a similar behaviour, but on them, *RCSRH* does not outperform *RCSR*.

There are some exceptions, though. The *GL719d* matrix on **M4**, for instance, scales with both *RCSR* and *RCSRH*, but the latter has worse performance. The reason for this is to be sought in the number of leaf matrices it gets partitioned into: from 218 of *RCSR*, to 719 of *RCSRH*. This is because of the dimension of the matrix, which is $\approx 2 \cdot 10^6$. Here, the δ_h recursion decision function has to subdivide the submatrix until the candidate (dense enough) submatrices fit into 16 bit indices. Alas, only 658 matrices out of 719 get to use these shorter indices, and the overall index overhead per byte (Fig. 4.5) raises by some 20%. Consequently, performance drops slightly.

The effect of forced subdivisions on submatrices is evident when looking at the single threaded *RCSRH* performance: it performs often worse than *RCSR*, except on matrices dimensioned less than 2^{16}, which do not require such forced subdivisions at all (matrices *raefsky3, av41092, c8_mat11_I, rma10, sme3Dc, venkat01*).

Fixing situations in which bad scaling occurs due to the exceptional sparsity and big dimension of the matrix is not possible at the present state of the δ_h function, because it only uses local information related to leaf submatrices. Such cases should be handled by subdividing the matrix more, if the total number of submatrices is deemed insufficient when compared to the number of available threads, coupled with a mechanism to prevent excessively small leaves. We have investigated into poor scaling of matrix *diego-MM-573x230k*. Despite 802/820 *RCSR*/*RCSRH* leaves on **M2**, and 1696/1765 *RCSR*/*RCSRH* leaves on **M4**, its

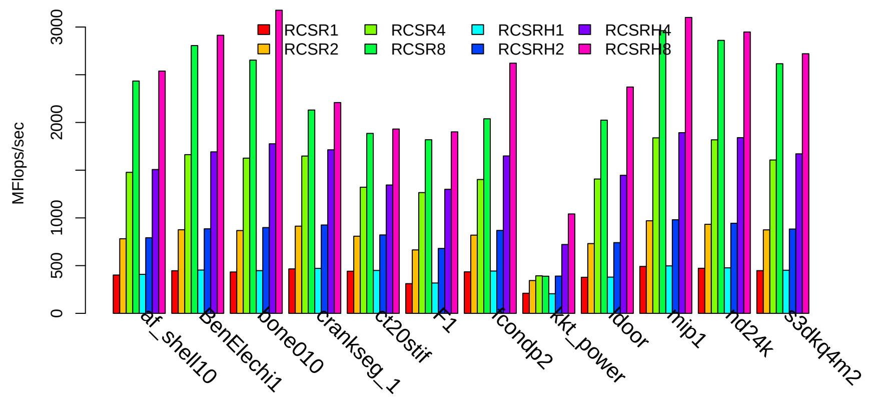

Figure 4.9: *SpMV* performance on **M2**, symmetric matrices.

nonzeroes density is much higher in the upper rows; thus creating an excessive contention for the upper submatrices (located in a limited row interval), and not allowing effective workload distribution among threads.

Summarizing, we can state that for unsymmetric matrices, we have obtained gains from using 16 bit indices, but in some cases we were also confronted with a performance loss. The latter cases involved mostly excessive subdivision resulting in additional overhead. Considering as a common feature of these badly performing matrices their very low nonzeroes/rows count, we would say that at the current state, *RCSR*/*RCSRH* is not the optimal format for them.

4.2.2 Symmetric Matrices

As we see in Fig. 4.10 and 4.9, the average performance of *SpMV* on symmetric matrices is much higher than that on unsymmetric ones. The gain over the unsymmetric cases is an immediate consequence of the representation of these matrices (recall discussion in §1.2.2); coefficients strictly above the diagonal ($U == L^T$) are not stored explicitly and their contribution to the *SpMV* ($y \leftarrow y + Ux$) is computed as ($y \leftarrow y + L^T x$), together with the lower contribution ($y \leftarrow y + (L + D)x$). This nearly halves the memory bandwidth needed when

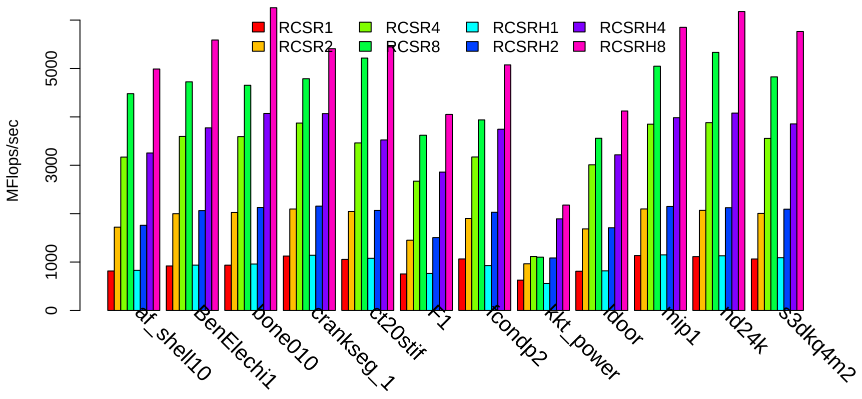

Figure 4.10: *SpMV* performance on **M4**, symmetric matrices.

reading the matrix; looking at Fig. 4.10 and 4.9, we see scaling almost for every matrix.

The only matrix with limited *RCSR* scaling is *kkt_power*, which is partitioned into 57/204 leaves for *RCSR*/*RCSRH* on **M4**, and 40/132 on **M2**. Here, recall the lower level of parallelism of the symmetric *SpMV* (see §3.1). Matrix *kkt_power* is not the (symmetric) matrix with the smallest number of leaves (that one is *ct20stif* — 31 leaves on **M2**, 60 on **M4**), but has a big dimension and is very sparse. We conjecture that the reason for the bad scaling of *RCSR* is that its partitioning on both **M4** and **M2** is such that eight (two times four) row intervals are enough to "cover" the whole range of rows. In the symmetric case this is more likely to happen, since every submatrix not laying on the diagonal will have both its row and column intervals in the result vector locked, when "active".

We report a plot of the recursion structure of the *kkt_power* matrix on **M4** in Fig. 4.13. Notice the visible and relevant change in layout when going from *RCSR* to *RCSRH*.

The best results from using *RCSRH* on the symmetric matrices were obtained on matrices *bone010* (34% improvement on **M4**) and *fcondp2* (28% improvement on both). As it can be seen in Figures 4.12 and 4.11, all matrices but

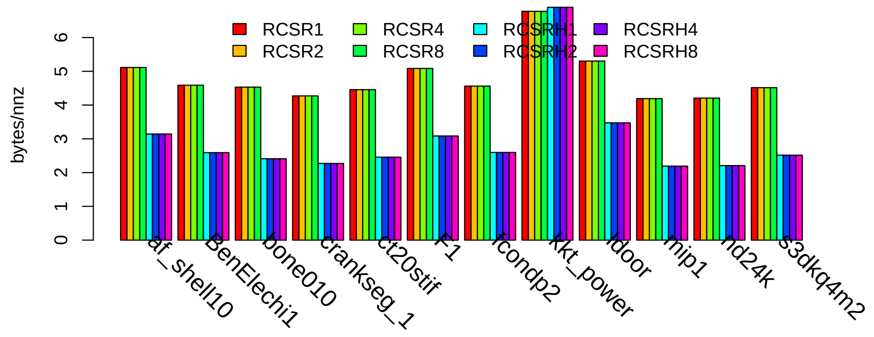

Figure 4.11: Index usage (bytes per nonzero) on **M2**, symmetric.

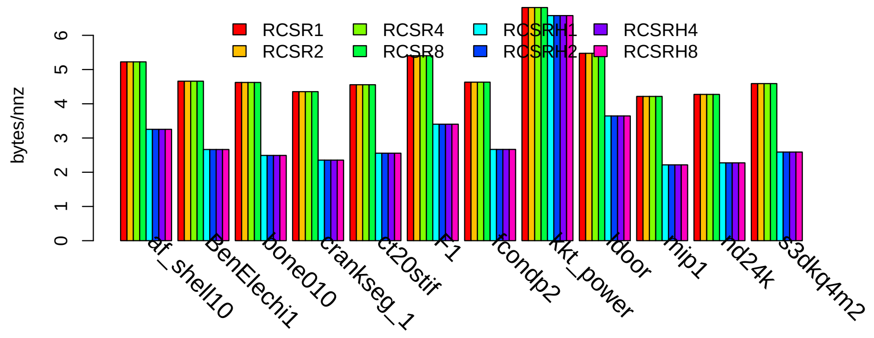

Figure 4.12: Index usage (bytes per nonzero) on **M4**, symmetric.

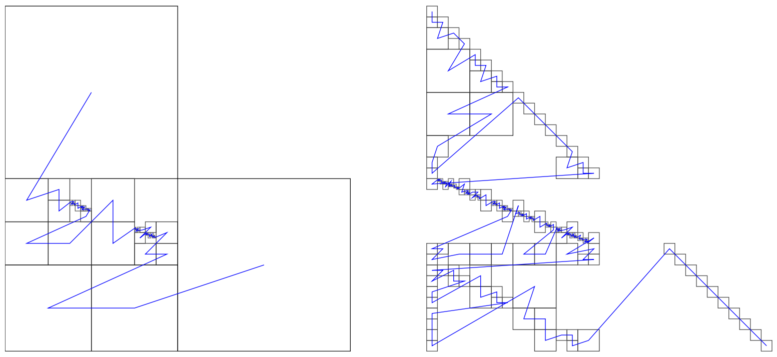

Figure 4.13: Matrix *kkt_power* as partitioned on **M4**, in *RCSRH* (right) and *RCSR* (left). Notice the visible and relevant change in layout. The *RCSRH* layout allowed overcoming severe scaling problems for this matrix and resulted in a two-fold speedup (see Fig. 4.10).

one nearly halve their index overhead. In §4.1.2 we have stated that the best bandwidth saving to be expected from *CSR* on 64 bit floating point number matrices could be around 16%, but the overall saving for the whole *RCSR* may be more or less, depending on the applied subdivision (which in turn, depends on both system parameters and matrix structure). So the exact combination of factors leading to a performance increase exceeding 16% on *fcondp2* is difficult to predict, but can be understood.

Figure 4.14 depicts a plot of the recursion structure of *fcondp2* on **M4**. Note that while the leaves count differs very slightly (passing from 255 to 257), the resulting performance gain is 28%. Clearly, breaking down the big lower-left submatrix improved the parallel execution of *SpMV*.

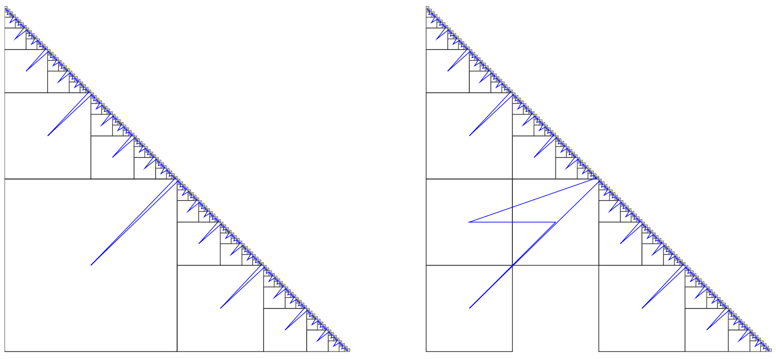

Figure 4.14: Matrix *fcondp2* as partitioned on **M4**, in *RCSRH* (right) and *RCSR* (left). The leaves count differs very slightly (from 255 to 257): the partitioning heuristic broke the big lower-left submatrix in three. This change in the matrix partitioning results in a 28% speedup of the parallel *SpMV* execution, because of a higher granularity in the locking of the results vector, limiting threads *starving*.

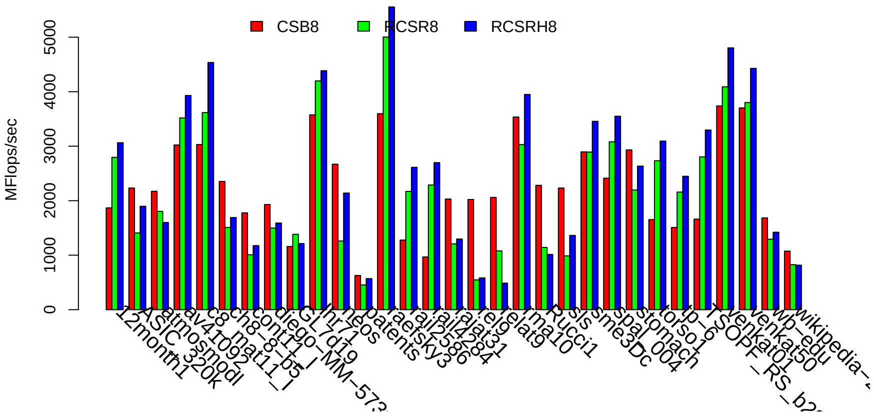

Figure 4.15: *CSB* vs *RCSR SpMV* performance on **M4**, unsymmetric.

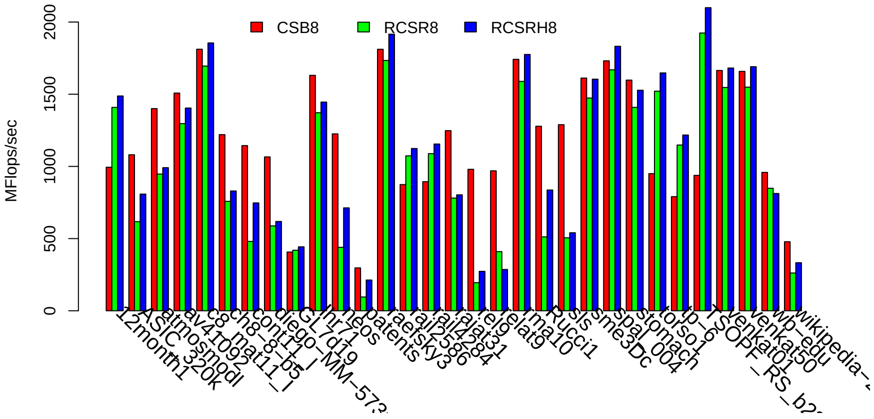

Figure 4.16: *CSB* vs *RCSR SpMV* performance on **M2**, unsymmetric.

4.2.3 Experimental Comparison with *CSB*

Finally, Figures 4.15 and 4.16 compare the performance obtained with *RCSR* and *RCSRH* to that of *CSB*. Since the *CSB* format (recall our brief description in §2.2, or see Buluç et al. [BFF+09, § 8]) currently does not handle symmetric matrices (or rather, *symmetric updates*), we can only compare the results for unsymmetric ones. Due to space limitations, we report results for 8 cores only. We skip matrix *cage15* in the results (the one with biggest dimensions and nonzeroes count), because *CSB* was unable to handle it (it needed more memory than it was available).

On **M4** (Fig. 4.15), we see that *CSB* performs better on *tall* matrices: *rel9, relat9, Rucci1, diego-MM-573x230k, contl11_l*, as well as on a *wide* one: *neos*, and square *rajat31, wb-edu, atmosmodl, patents*. *RCSR* and/or *RCSRH* prevail on *wide* matrices: *12month1, c8_mat11_I, GL7d19, spal_004, rail2586*, square ones: *av41092, lhr71, raefsky3, rma10, sme3Dc, torso1, venkat01*, and a *tall* one: *tp-6*. On **M2** (Fig. 4.16), we observe exactly half of the matrices favouring *CSB*, and half favouring *RCSRH*.

4.2.4 Conclusions From the Introduction of Short Indices

We have discussed simple modifications to our base sparse matrix format (*RCSR*) which allowed usage of *halfword* indices. This modification exploits the hierarchical structure of *RCSR* and involves changes in the way that the recursive subdivision is performed. As a result, we were able to achieve speedups between 10% and 25% on unsymmetric matrices (and up to 34% on symmetric ones) on eight active cores; without substantially changing neither the *SpMV* algorithm(s), nor the recursive matrix format. In some cases, the performance boost was not possible because of complex interactions between the matrix structure and the subdivision mechanism. For the same reasons, in some cases we have observed a performance drop; mainly due to subdivision policy generating too much recursion. From this, we conclude that the usage of short indices should be pursued whenever possible, as the saving in memory traffic pays off especially when many cores are active. However, we should investigate further cases where excessive subdivisions result in growth (rather than drop) in index usage, and find a way around the excessive subdivision. Also, subdividing with regard to the underlying available threads will have to be considered. Moreover, a technique for parallelizing the execution of *SpMV* in cases of very large and very sparse matrices should be addressed.

Separately, usage of some other appropriate (non *CSR*) format on selected leaf submatrices (regardless of the index type size) needs to be investigated. This could prevent excessive memory usage on submatrices where $rows > nnz$.

Finally, we would like to note that as a consequence of using the *dense block width/height* as an index multiplier (recall from §1.2.4 or see Im et al. [IYV04a]), dense blocking techniques allow leaf submatrices bigger than 2^{16} while still using 16 bit indices.

4.3 Heterogeneous (*COO*/*CSR*) Leaves:RSB

4.3.1 Recursive *CSR* and Index Overhead

With *RCSR*/*RCSRH*, we (logically) organize a sparse matrix as a *quad-tree* structure, with nodes consisting of submatrices arising from a recursive partitioning into quadrants. While intermediate nodes are used only as a pointer structure, leaf nodes hold actual subarrays with index and numerical values. The *SpMV* algorithm described in §3.1 is independent from the actual format of leaf matrices. It only assumes a *coarse* recursive partitioning in leaf submatrices. Similarly to blocking techniques used in dense matrix computations (see for instance, one of Gustavson's works—[Gus97]), submatrices at the leaf level should be *sized* (in terms of their *memory footprint* during the *SpMV*) in relation to the *cache sizes of the machine.*

In this context, in §4.1, we have investigated a variation to the leaf matrices format, obtained by converting some of the *Compressed Sparse Rows* (*CSR*) leaves of a matrix to use 16 bit column indices (and thus, reducing the memory traffic). As motivated before, (and in the literature; e.g.: see Kourtis et al. [KGK08]), techniques for saving memory bandwidth during computation are particularly effective with many active cores. Here, techniques which may not be optimal on a single core (because of a slight memory-bandwidth-to-computation trade-off, in the form of pointer arithmetics) may show their potential when working with multiple cores (where the memory traffic is heavier). As a motivation for our "16-bit" approach, we observe that after partitioning a large sparse matrix (in the *RCSR* format), it is likely to have many of the leaf submatrices *dimensioned* less than 2^{16}. Thus, using a 16 bit (*halfword*) index type in their *CSR column indices* arrays is possible, and could lead to savings in memory traffic. We named this variant *RCSRH*. Obviously, for matrices dimensioned less than 2^{16}, the conversion to *RCSRH* is possible for all submatrices. The outcome of our experiments (documented in [MFPT10c]; see §4.2) was encouraging: using halfword indices by itself yielded up to a 25% floating point speedup (with a saving in memory usage up to a 16%) on unsymmetric matrices, and 30% on symmetric ones. However, in a number of cases, the *RCSRH* variant was not helpful. One of the perceived reasons was that *CSR* itself does not always fit into leaf submatrices, and thus we have decided to convert some leaf matrices to the *COOrdinate* (*COO*) format. Let us discuss this change with more detail in §4.3.2.

4.3.2 Recursive Storage Format with *CSR* and *COO* Leaves

In this section, we motivate quantitatively why and when storing some submatrices as *COO* instead of *CSR* could reduce index overhead, and the way we have chosen to use *COO* to enhance *RCSR*.

A matrix is stored in the *RCSR* format as a quad-tree structure with *CSR* (recall §1.2) submatrices at the leaf level of a *recursive bipartitioning* (recall §2.3 or see [MFT+10]). To store an $r \times c$ matrix with n nonzeroes in *CSR*, we use an array JA (of size n) with *column indices*, and a *row pointers* array PA (of size $r+1$), referencing *rows* in the JA array. Array JA stores column indices for nonzeroes in a *row-major* order. The array of coefficients (VA) is laid in the same order as JA. To store a matrix in a plain *COO* format, two n-sized arrays for (row,column) indices (IA,JA) are required. By denoting as $I(r,n)$ the index space requirements for an $r \times c$ matrix (with n nonzeroes) instance we have $I_{CSR}(r,n) \overset{def}{=} 4(r+1)+4n$ and $I_{COO}(r,n) \overset{def}{=} 4n+4n$ bytes. Let us call *CSRH* the *CSR* format implementation with 16 bit JA indices, and *COOH*, a *COO* format implementation with 16 bit IA and JA indices. For these variants, we have $I_{CSRH}(r,n) \overset{def}{=} 4(r+1)+2n$ and $I_{COOH}(r,n) \overset{def}{=} 2n+2n$ bytes. This means that for some values of (r,n), *COO*/*COOH* would use less indexing space than *CSR*/*CSRH*; specifically, $I_{COO}(r,n) < I_{CSR}(r,n)$ when $n < r+1$, and $I_{COOH}(r,n) < I_{CSRH}(r,n)$ when $n < 2r+2$. For this experiment, we modified the matrix constructor code to use *CSRH* whenever a *CSR* submatrix is dimensioned less than 2^{16}. Similarly, we use *COOH* whenever a *COO* submatrix is dimensioned less than 2^{16}; we choose to use *COO* when $n < r+1$. We adopt *COO*/*COOH* as row-major sorted[2] (so we have the same memory access pattern of *CSR* for JA and VA arrays). In earlier sections (see §2.3 and §4.1), we have described the *cutoff* function δ as our heuristic regulating subdivision into submatrices; in this section, we use slightly differing matrix assembly criteria. While we still use the δ_h function from §4.1.1, here we limit subdivisions by forcing each submatrix not to use more indexing space than a

[2]In §1.2 we called this variant *COR*.

fullword COO storage of it would require. The remaining rules for subdivision are still the same as imposed by δ_h. We call the hybrid format resulting from these modifications *Recursive Sparse Blocks* (*RSB*). With this layout, algorithms for the random access/update of individual matrix nonzeroes as discussed in §2.3.5 are still valid *in between* submatrices. But the presence of matrices in the coordinate storage requires the implementation of specialized *binary search* code for delimiting the individual rows boundaries first, and then column indices in the *COO* arrays. Refer to Table 1.1 for the memory access patterns (and thus, different computational complexity) of the *random nonzero set* and *diagonal extraction* operations on individual *COO* leaves.

4.4 Experimental Evaluation of RSB

We structure the analysis of results as in the previous experiments. Note that for brevity, we sometimes reference the k-threaded *RSB* as *RSB*-k. In most cases we start by commenting on the 8-threaded performance, discussing the particularly problematic cases first, and leaving the best performing cases discussion as last.

Please refer to §A.4 for a full description of the experimental setup we used.

4.4.1 Unsymmetric Matrices

For the unsymmetric matrices on **M4**, we observe an improvement when switching from *RCSR* to *RSB* in nearly all of the test set matrices; up to 67% on square ones, and up to 33% on non-square ones (Fig. 4.17,4.18).

The only matrices "suffering" from the switch are: square *av41092* and *raefsky3* (Fig. 4.17), non-square *c8_mat11_I* and *diego-smtxMM-573x230k*, and two borderline cases: *rail2586* and *sme3Dc*.

On machine **M2** (Fig. 4.19,4.20), we see improvements up to 128% for square matrices, and 65% for non-square ones, and a single case of a performance drop: a 3% fall for the non-square matrix *cont11_l*.

In Fig. 4.21,4.22,4.23,4.24 we observe index usage saving almost always. Out of 24 non-symmetric matrices, we experience three cases where index usage raises: square matrix *patents* (Fig. 4.21,4.23) and non-square matrices, *rel9*, *relat9* (Fig. 4.22,4.24). We note, however, that the effect of *RSB* is actually an improvement of the performance on these matrices, notwithstanding the increased index usage. Among these matrices, problematic cases remain: *patents* performs better, but continues scaling poorly, (remaining the "slowest" of our

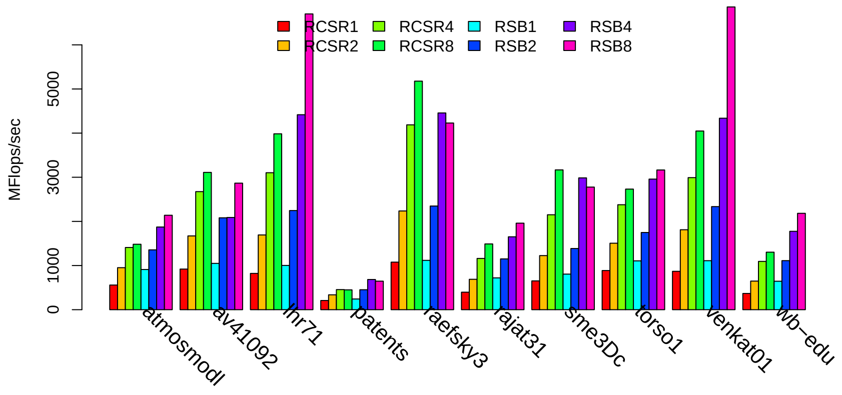

Figure 4.17: *SpMV* performance on **M4**, square matrices.

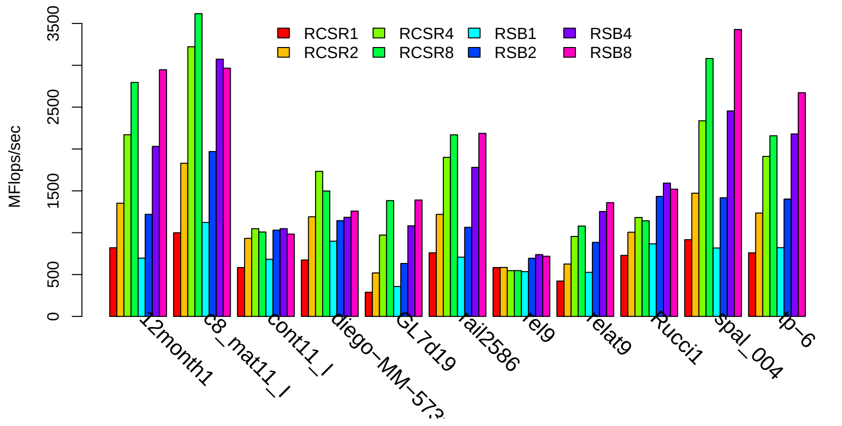

Figure 4.18: *SpMV* performance on **M4**, rectangular matrices.

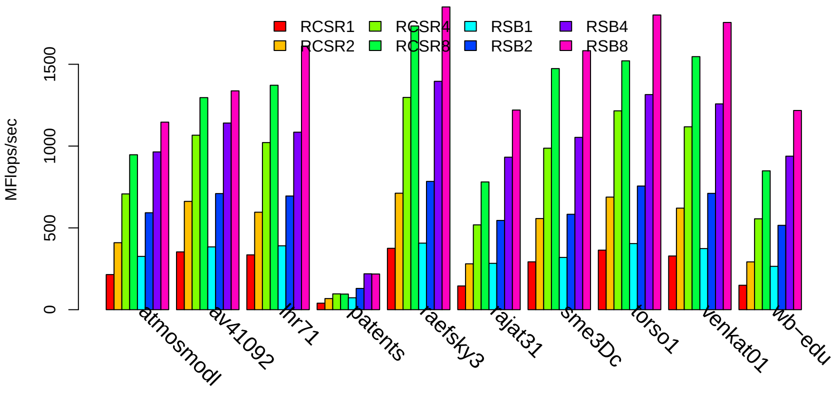

Figure 4.19: *SpMV* performance on **M2**, square matrices.

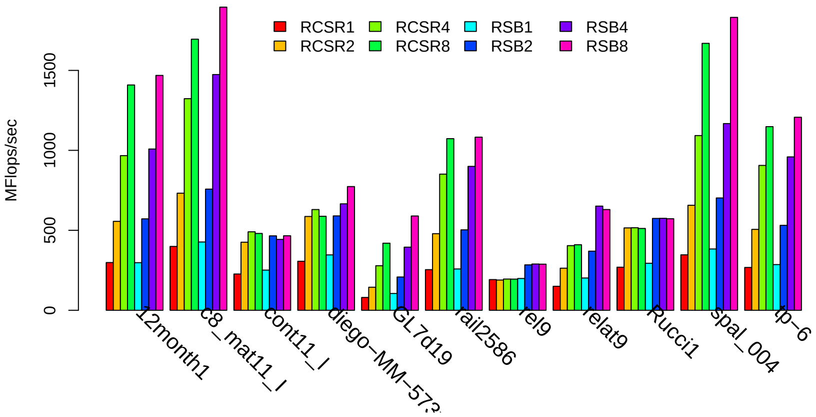

Figure 4.20: *SpMV* performance on **M2**, rectangular matrices.

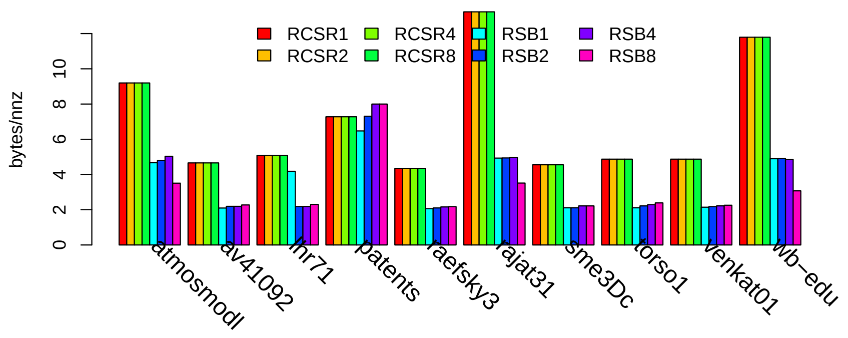

Figure 4.21: Index usage (bytes per nonzero) on **M4**, square.

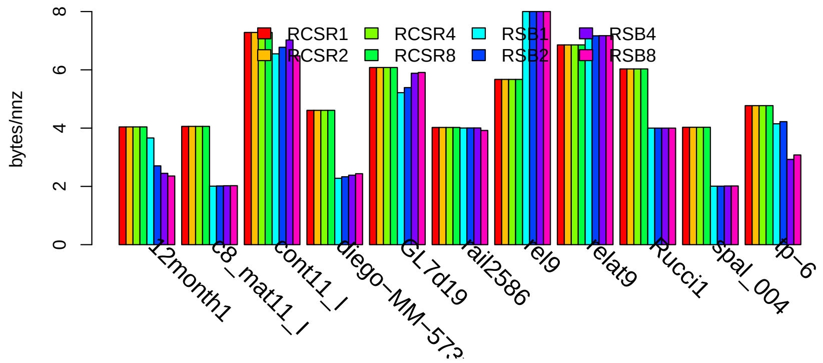

Figure 4.22: Index usage (bytes per nonzero) on **M4**, rectangular.

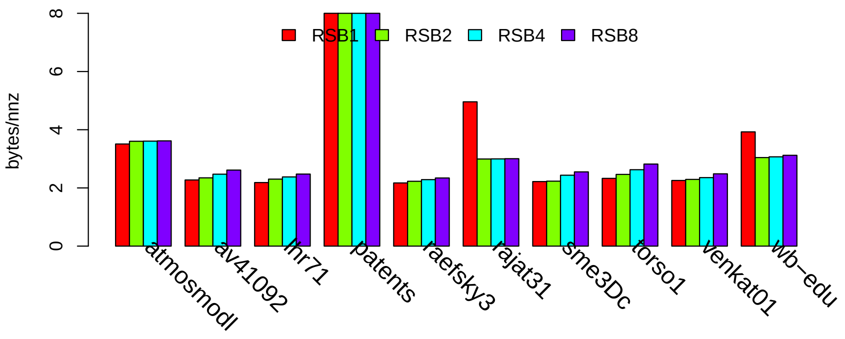

Figure 4.23: Index usage (bytes per nonzero) on **M2**, square.

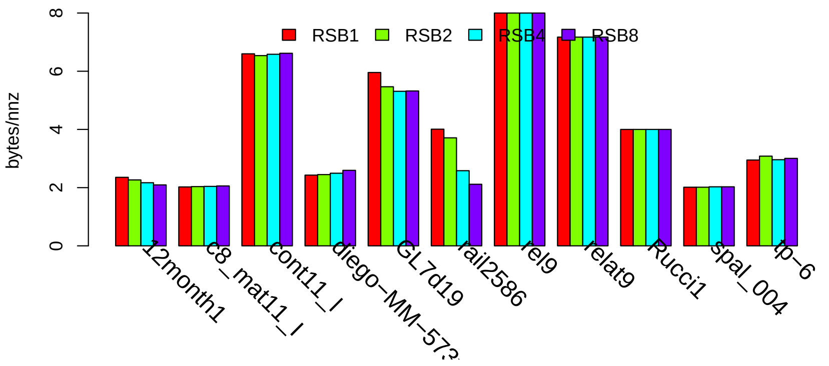

Figure 4.24: Index usage (bytes per nonzero) on **M2**, rectangular.

entire test set); *relat9* suffers from poor scaling, too (especially on 8 cores **M2**); *rel9* continue not scaling at all.

These matrices have a feature in common: a very low nonzeroes/row elements ratio: 2.39 for *rel9*, 3.15 for *relat9* (see Table A.12) 3.97 for *patents* (see Table A.11). Although for such matrices one cannot expect high efficiency for either *CSR* or *COO* formats, we have realized why this is also the case for our recursive format (see the previous sections), so now we present only the particular case for *RSB*.

Although very poorly performing already with 1 thread, *patents* scales up to no more than 4 threads. In fact *patents* is assembled in 37 *COO* leaves, regardless of the thread count. When working with 8 threads, we observe that scaling is inhibited: this means that particular partitioning leaves a number of threads starving, while most of row intervals are *locked* by other threads. This is a situation occurring when the thread count approaches the number of submatrices in disjoint row intervals (see Fig. 4.25); and thus threads contend for available row intervals to operate on. In the current formulation of *RSB*, further partitioning of this matrix is not allowed, for it does not have enough nonzeroes per row. On **M2**, the case for *patents* is similar: while on 1,2,4,8 threads, the matrix is partitioned respectively into 13,25,37,37 *COO* leaves.

The cases of *rel9* and *relat9* (Fig. 4.18,4.20) are similar. Since *relat9* has a little higher nonzeroes/row count than *rel9*, it succeeds in scaling in a limited way (up to 30 *COO* leaves, on both machines), but *rel9* gets partitioned in 7 leaves only, in all cases. Therefore, for *rel9*, more than 2 threads contend for row locking on 7 submatrices, with no possible scaling. Notice, however, that *RSB* is capable of allowing dual threaded parallelism in these *very sparse* cases, whereas *RCSR* was not.

The cases we have just discussed are worst/limit cases, and as such are not the primary target of our modifications, so we tolerate them here, and use them as means for comparison.

Although quite different, two matrices (*sme3Dc*, *raefsky3*) suffer similar problems, when instantiated as *RSB* on **M4**. That is, while they are well-performing on *RCSR* and loosing index overhead from the *RSB* switch, they also get partitioned into less leaves, giving rise to the same *SpMV* scalability problem. In fact while *RCSR*-8 partitions these matrices respectively into 115 (113 *CSRH*, 2 *COOH*) and 94 (CSRH) leaves, *RSB*-8 produces 16 (all *CSRH*) and 13 (11 *CSRH*, 2 *COOH*) leaves. Given the lock-based nature of our *SpMV* algorithm, and the distribution of submatrices, *RSB*-8 suffers from contention problems on both matrices. It is interesting to note that on **M2**, these matrices get subdivided respectively in 115 and 94 leaves, and we observe in Fig. 4.19

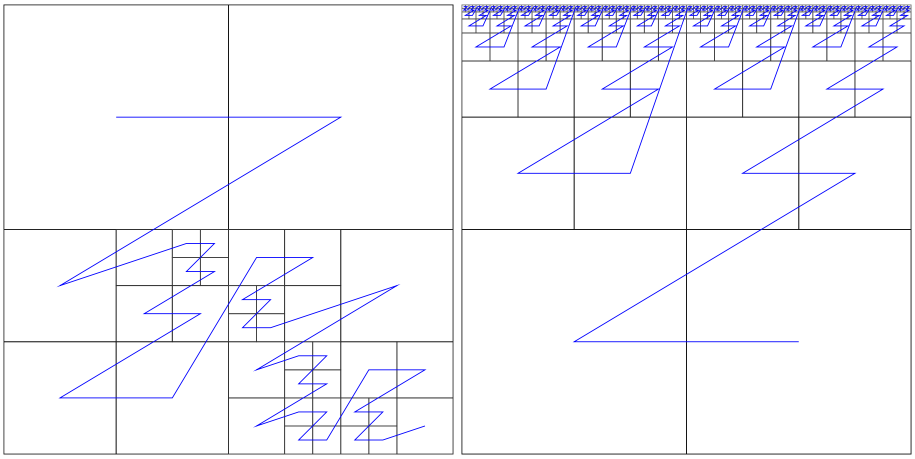

Figure 4.25: On the left, matrix *patents* as partitioned on **M4**. On the right (widened, for viewing convenience) *diego-smtxMM-573x230k* on **M4**. Both are partitioned with the heuristic updated for *RSB*.

that this suffices to scale and experience, respectively, a 7% and a 6.7% improvement. Index overhead shifts from 4.44. to 2.55 bytes/nonzero for *sme3Dc*, and from 4.28 to 2.34 bytes/nonzero for *raefsky3*.

Matrix *av41092* on **M4** experiences the same problem *sme3Dc* and *raefsky3* did: insufficient partitioning. While **M4** partitions this matrix in 10 (9 *CSRH*, 1 *COOH*) submatrices only, **M2**, due to its smaller caches, partitions it in 72 leaves (64 *CSRH*, 8 *COOH*). So, the halving in index overhead experienced on **M4** (from 4.65 to 2.27 bytes/nonzero) could not bring advantage to *RSB*-8, while on **M2**, the 42% index saving (from 4.5 to 2.61 bytes/nnz) allows for scaling and a modest 3% performance increase.

The remaining three cases with a missing improvement are non-square matrices *c8_mat11_I*, *diego-smtxMM-573x230k*, and *rail2586* (Fig. 4.18). Matrix *c8_mat11_I*, alike to the matrices we have seen before on **M4**, suffers from poor partitioning, here: *RSB* partitions it in respectively 1,4,10,13 leaves for 1,2,4,8 threads. On 8 threads, the 13 leaves are not enough to ensure the parallel operation of all the threads, thus leaving some of them *starving*. Similarly to the previous cases, **M2** divides the matrix in much more leaves, thus avoiding the scaling problem.

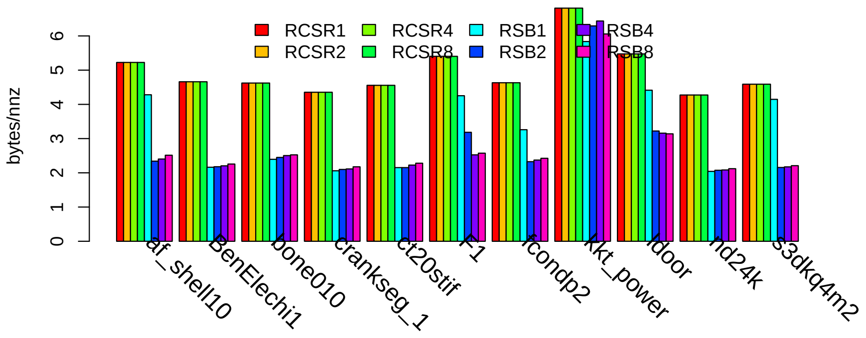

Figure 4.26: Index usage (bytes per nonzero) on **M4**, symmetric.

The case for matrix *diego-smtxMM-573x230k* is different (and interesting). On **M4**, this matrix performs best as *RCSR*, while on **M2**, best as *RSB*. On both machines, though, while not scaling up to 8 threaded *RCSR*, it scales (although very *slightly*) for *RSB*, up to 8, but poorly. Poor scaling is evident: *RSB*-8 on **M4** is only 88% faster than *RSB*-1; on **M2**, only 123%. By looking at the number of submatrices, we could not say their number is too low. It is only after inspecting the distribution of submatrices (see Fig. 4.25), that we notice a big unbalance: actually, most of the submatrices are located on the top of the matrix, and it seems that *RSB* arranged submatrices in "block rows". Given the row-lock-based nature of our *SpMV* algorithm, such a distribution is enough to destroy the parallelism of the computation on this matrix. Here, after completing the bigger-dimensioned submatrices across various row intervals of the matrix, threads will try to acquire a lock on the intervals located on the upper border, with no success for most of them: only a few of them will be able to work at a time, on the upper submatrices. Contention will last during the whole computation for most of the threads, then, because our current *SpMV* algorithm has no mechanism for concurrent update of a single subvector.

Matrix *rail2586* constitutes another special case. For being *wide*, it fits particularly well when stored in a row-oriented storage as *CSR*. However, for having its nonzeroes scattered quite uniformly around the matrix, it would end up hav-

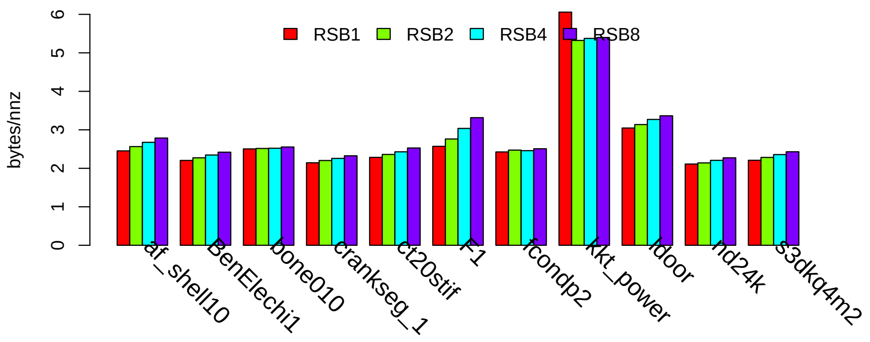

Figure 4.27: Index usage (bytes per nonzero) on **M2**, symmetric.

ing very sparse submatrices, if it had not as much as 3097 average nonzeroes per row. But it happens that for being so wide, the proper introduction of *CSRH* leaves is only possible after a certain number of subdivisions. On **M4** (Fig. 4.26), it happens that there are not enough subdivisions for switching much of the submatrices to *CSRH*. So, the use of *RSB* for *rail2586* on **M4** does not reduce the index overhead significantly (it remains at about 4 bytes per nonzero), and the performance remains the same (notwithstanding the submatrices reduction: from *RCSR*'s 352, to *RSB*-8's 55). For architectural reasons, *RSB* on **M2** ends up partitioning the matrix more finely, and thus falling to switch to *CSRH* in 335, out of the 352 leaves of *RSB*-8. The matrix is thus partitioned in number of matrices which is the double of *RCSR*'s. However, in this case, the performance gain expected from *RSB* is negligible: less than 1%. We conjecture that the *flat* distribution of submatrices in the matrix, and its considerable width, cause a considerable overhead to the memory subsystem, which in turn is forced to continuously load elements from the right-hand side vector, which would barely fit in the cache.

We notice that some matrices gain a considerable speedup from the *RSB* representation: *rajat31* (56%), *lhr71* (17%), *torso1* (18%) on **M2** (Fig. 4.20), *venkat01* (67%), *cage15* (50%) on **M4** (Fig. 4.18), *wb-edu* on both (68% on **M4**, 43% on **M2**). The assembled instances of these matrices as *RSB* differs from

RCSR, for the relevant number of *COO*/*COOH* submatrices. On **M2**, *rajat31* gets partitioned in 1534 leaves, of which 896 *COOH*, and 126 *COO*; *wb-edu* in 4336 leaves, of which 2511 *COOH*, 254 *COO*; *torso1* in 357 leaves, of which 39 *COOH*; *lhr71* in 87 leaves, of which 34 *COOH*. In all these cases, index overhead is cut down approximately in a half. On matrices *rajat31* and *wb-edu*, index overhead falls down respectively from 12.3 to 3 bytes/nnz and from 11.15 to 3.12 bytes/nnz. This means that *RSB cures* cases where *RCSR* alone produced subdivisions abusing from *CSR* leaves; that is, producing *CSR* leaves with less nonzeroes than rows. The case for matrix *cage15* on **M4** is alike, in that it gets partitioned in 751 leaves, 132 of which are *COO*, 316 *COOH*, 6 *CSR*, 297 *CSRH*. With *RSB*, this configuration of *cage15* saves approximately 30% index overhead (from 6.3 bytes/nonzero), which is not much compared to other cases. So probably, the gain is due to the *fuller* submatrices (*RSB*-8 assembles 751 of them; *RCSR* as much as 4457). Performance gain on *torso1* is probably due only to index overhead saving: in *RSB*-8 on **M4**, it gets partitioned in 59 *CSRH* leaves only, (from 176 *CSR*), saving 64% of indexing overhead (from 4.6 bytes/nonzero, Fig. 4.21), which is quite good.

4.4.2 Symmetric Matrices

Bar plots in Fig. 4.29 and 4.28 present the comparative performance results of *RCSR* and *RSB* for symmetric matrices. We observe performance enhancements nearly in all cases. There are three exceptions, though: *crankseg_1*, *ct20stif*, *F1* on **M4**. We comment these exceptions first, and the remaining cases next.

On **M4**, matrix *F1* in *RSB* (Fig. 4.29) does not scale from 4 to 8 threads. On less than 8 threads, *F1* is processed faster with *RCSR*; e.g.: with 1 thread, *F1* gets partitioned by *RSB* in 10 submatrices only, all fullword *CSR*. But with 8 threads, *RSB* partitions *F1* in 72 leaves, of which 70 are *CSRH* and 2 *COOH*. With *RCSR*, a number of 573 leaves were obtained, which is much more. Given the higher number of subdivisions, load balancing in *RCSR* ran for sure smoother, while *RSB* did fall in a lock contention problem here, it seems. Please recall (see §3.1) that our symmetric *SpMV* implementation variant incurs in a higher locking overhead than unsymmetric. On **M2**, the situation is almost reversed: for 8 cores, it is *RSB* that partitions *F1* in more leaves (573: 504 *CSRH* and 69 *COOH*), while *RCSR* divides the matrix in 278 leaves *only*. The index overhead of *RCSR* is quite high on *F1*: 5.08 bytes/nnz on **M2**, 5.4 on **M4**; on *RSB* it is always less than this, on both machines. However, the *RSB* index overhead depends on the threads count: on **M2** (Fig. 4.27) with more threads, the overhead tends to grow too, from 2.6 to 3.3 bytes/nnz, suggesting

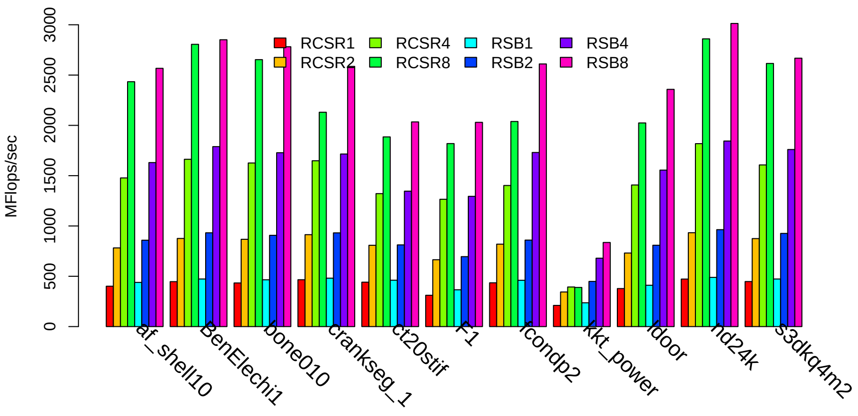

Figure 4.28: *SpMV* performance on **M2**, symmetric matrices.

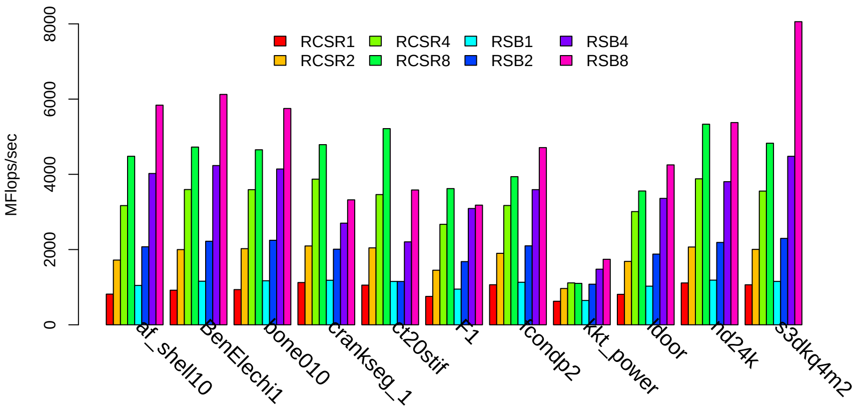

Figure 4.29: *SpMV* performance on **M4**, symmetric matrices.

that further subdivisions could degrade performance. On the other hand, on **M4**, when going from 1 to 8 threads, this overhead decreases from 4.25 to 2.52 bytes/nnz (Fig. 4.26). These observations suggest us that the performance improvement over 1-core *RCSR* (on both **M4** and **M2**) is due to less index overhead, which itself is a consequence of less submatrices fragmentation. We believe that some *optimum partitioning* for 8 cores *F1* is between all of these four instances of *RCSR*/*RSB* on **M2**/**M4**; that is, the algorithm should have partitioned *F1* less coarsely on *RSB*/**M4**, more coarsely on *RCSR*/**M4**, and so on.

The cases for matrices *ct20stif* and *crankseg_1* (still on **M4**) are different. With *ct20stif* we observe that 2-threaded *RSB* fails from partitioning, thus cutting off two-cores parallelism completely (Fig. 4.29). On more cores the heuristic succeeds partitioning the matrix, but too coarsely to gain a sufficient workload balance. Note that this matrix is among the smallest in our test set ($1.3 \cdot 10^6$ nonzeroes), stressing the limit of our rule of thumb (*sizing* matrices around the cache sizes). On both **M4** and **M2** machines, index usage for *ct20stif* keeps very low: for *RSB* it ranges from 2.27 to 2.52 bytes per nonzero, coming from *RCSR*'s approximate 4.5. With an analogy to the previous case, on machine **M2**, partitioning is finer than on **M4**, from the single thread case on (1-threaded *RSB* partitions *ct20stif* to 7 submatrices), and an adequate workload balancing follows. Thus with *ct20stif* on **M2**, we do not loose the 8 threaded case, and *RSB*'s performance is higher than *RCSR*'s. Here, the sparser leaf submatrices are assembled as *COOH* (2 out of 7 on 8 cores **M4**, 2 out of 60 on **M2**), the remaining ones in *CSRH*. Notice that both *F1* and *ct20stif* matrices had more than 25 nonzeroes/row, which is quite sufficient to achieve good results with *RCSR*/*RSB*. Matrix *crankseg_1* is a little bit sparser (10 nnz/row). It suffers from the same *poor partitioning* problem on **M4**, having respectively 3,10,16,39 leaves for 1,2,4,8 threads, and loosing 30% of performance on 8 threads. On the other hand, on **M2**, matrix *crankseg_1* performs quite well, achieving an improvement to *RCSR*. The improvement itself is about 21% on 8 cores, when the matrix is partitioned in 37 *COOH* and 202 *CSRH* submatrices.

After having discussed the problematic cases, let's look at the remaining ones.

In one case there is almost no change: *nd24k* on **M4** (Fig. 4.29). Here, *RCSR* partitions the matrix in 503 *CSR* leaves, *RSB* in 87 *CSRH* leaves. The index overhead (Fig. 4.26) gets almost halved (from 4 bytes bytes/nonzero). We are not aware of the reason for the missing performance increase, here, but note that this is our symmetric matrix with the higher nnz/row count (199, see Table A.10). On **M2** (Fig. 4.28), the same matrix witnesses a slight (5%) speedup, while being partitioned by *RSB* in 503 (all *CSRH*, except 5 *COOH* ones) pieces, and 278 ones by *RCSR*. The index overhead (Fig. 4.27) similarly to that of **M4**, halves from *RCSR* (4.2 bytes/nnz) to *RSB* (2.1 bytes/nnz). We conjecture that the 87 leaves on **M4** somehow limited parallelism, but we would need to investigate further to confirm this.

In one case, on **M4**, *RSB* performance boosts up as high as 66%, when compared to *RCSR*: it is for matrix *s3dkq4m2* (Fig. 4.29). Here, *RCSR* partitions in 127 leaves, while *RSB* in 15 only (8 *CSRH*, 7 *COOH*). We observe the index overhead (Fig. 4.26) is almost halved, switching from *RCSR* to *RSB* (for > 1 threads). We deem that this speedup is due to a case in which the matrix offers caching potential (the whole result vector and a matrix portion): on **M2**, where the L3 cache is considerably smaller than on **M4**, the performance of *s3dkq4m2* improves by only 2%, passing from 63 leaves of *RCSR* to 120 *CSRH* and 7 *COOH* leaves of *RSB*. Performing a run with *cold caches* (that is, making sure that any location caching the matrix or the involved vectors gets overwritten between each *SpMV*), on **M4** the performance of *RSB* is approximately 7% lower, while on **M2** it made no difference (and the boost becomes 55%, rather than 66%). Note that the smallest symmetric matrix in the test set is not *s3dkq4m2* but *ct20stif*, which we have commented before.

When switching from *RCSR* to *RSB* on **M2** (Fig. 4.28), we observe speedups in all cases. Probably, L3 cache on **M2**, smaller than on **M4**, induced too coarse partitionings, thus limiting the scalability of our symmetric *SpMV*.

We can now comment the cases where the biggest improvement was observed: *af_shell10* (30%), *BenElechi1* (29%), *bone010* (24%), *fcondp2* (20%), *ldoor* (19%) on **M4** (Fig. 4.26), and *fcondp2* (28%), *crankseg_1* (21%), *ldoor* (16%), *F1* (12%) on **M2** (Fig. 4.27). For *af_shell10* on **M4**, we observe that *RSB* instantiates 255 submatrices (192 *CSRH*, 48 *COOH*, 15 *COO*), while *RCSR* used to instantiate 1534 *CSR* leaves. This matrix is also the one to experience the higher saving in index overhead: from 5.22 to 2.5 bytes per nonzero (more than 50%, Fig. 4.26). Matrix *BenElechi1* gets partitioned by *RSB* in 63 leaves: 32 *CSRH*, 30 *COOH*, 1 *COO*; by *RCSR* in 382 *CSR* matrices. Index usage (Fig. 4.26) halves: from 4.66 to 2.25 bytes per nonzero. Similarly to the *af_shell10* case, we experience a smaller number of leaf matrices, a more appropriate leaf matrix selection, and a consequent reduction in indexing overhead. On **M2** (Fig. 4.28), the same matrix improves only by 1.6%. By looking at its partitioning, we notice that it is partitioned in 127 leaves by *RCSR*, which is much less than *RSB*'s 255 leaves (238 *CSRH*, 16 *COOH*, 1 *COO*). For *bone010*, *RCSR* assembles 1316 *CSR* matrices; *RSB* assembles 170 *CSRH*, 2 *CSR*, and 5 *COO*. Index usage is reduced down from 4.6 to 2.5 bytes/nnz (Fig. 4.27). On **M2**, *RSB* assembles

1054 *CSRH*, 279 *COOH*, and 5 *COO* submatrices, while *RCSR* allocates 630 *CSR* leaves (index overhead shifting from 4.53 to 2.55 bytes/nonzero). Again, it seems the partitioning proceeded too deeply. Matrix *fcondp2* is partitioned in 31 leaves (19 *CSRH*, 1 *CSR*, 11 *COOH*) with *RSB*, and with *RCSR* in 255 leaves. Index overheads falls from 4.63 to 2.42 bytes/nnz. On the same matrix, on **M2** the improvement is even higher, this time. Here, *RSB* partitions in 257 leaves (182 *CSRH*, 75 *COOH*), while *RCSR* in 127 leaves only. Index overhead falls from 4.56 to 2.5 bytes/nonzero. So, in contrast to the preceding cases, matrix *fcondp2* benefits from increased subdivision, on **M2**. Matrix *ldoor* is partitioned in 157 leaves (122 *CSRH*, 5 *CSR*, 26 *COOH*, 4 *COO*, 3.14 bytes/nnz) by *RSB*, and 789 leaves by *RCSR* (5.47 bytes/nnz, Fig. 4.26). On **M2**, the performance gain is smaller than on **M4** (16%, rather than 19%). Partitioning of *ldoor*, here, produces 804 (471 *CSRH*, 329 *COOH*, 4 *COO*) submatrices, while *RCSR* produces 431 leaves. Also index overhead falls more gently: from 5.30 to 3.36 bytes/nnz.

We conclude by observing that there is a strong correlation between the index saving and performance gain: milder index savings on **M2** showed milder performance improvements, while bigger index savings on **M4** were accompanied by higher improvements.

4.4.3 Comparative Analysis

Let us now look at the performance of all matrices as *RCSR*, *RCSRH*, and *RSB*, using 8 threads. For unsymmetric matrices, we also give performance results for the *CSB* prototype. Unfortunately, we had to skip matrix *cage15* (the one with the highest nonzeroes count), because *CSB* was unable to instantiate it (the *CSB* implementation needed more memory than the 24 GB available on **M4**).

We observe that for **M2** (Fig. 4.31): matrices which favour *RSB* most (over CSB) are *c8_mat11_I*,*spal_004*,*wb-edu*; one matrix looses against *RCSR* (*cont11_l*); the majority of *RSB* cases is faster than *RCSR* (19 matrices out of 20). Summarizing, *RSB* performs faster than *CSB* (and is also the fastest among the four cases) in 7 cases out of 20. CSB is the fastest in 12 cases; in one case it is faster than *RSB*, but not the fastest one.

On **M4** (Fig. 4.30) we observe that: RSB is much faster than *CSB* on *wb-edu* and *venkat01*; 6 matrices seem to perform very similarly in both *CSB* or *RSB*; the remaining ones perform better in one of the two formats. Some matrices loose performance in *RSB*, over *RCSR*: matrices *av41092*,*c8_mat11_I*,*cont11_l*; (slightly) *diego-smtxMM-573x230k*,*sme3Dc*; other matrices favour *RSB* over *RCSR*: about 15, out of 20.

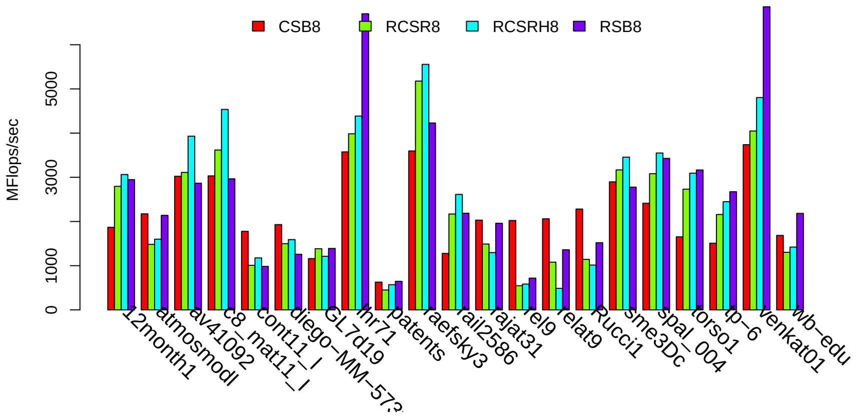

Figure 4.30: Results for 8 cores on **M4**, comparing *CSB*, *RCSR*, *RCSRH*, and *RSB* (unsymmetric matrices).

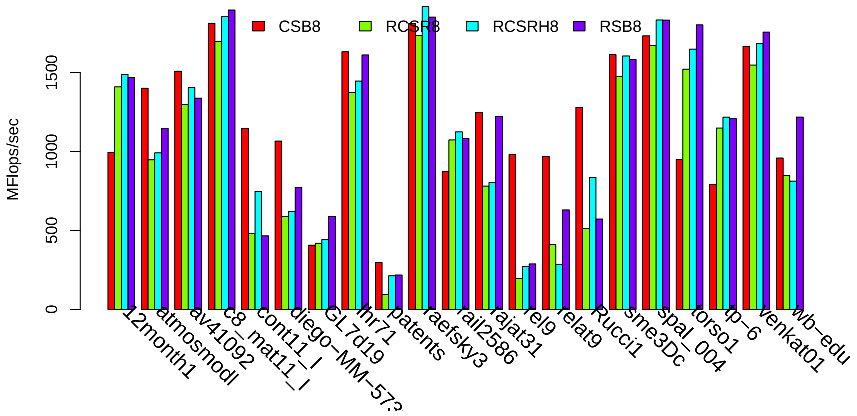

Figure 4.31: Results for 8 cores on **M2**, comparing *CSB*, *RCSR*, *RCSRH*, and *RSB* (unsymmetric matrices).

For space reasons, we omit figures showing comparative performance for symmetric matrices on *RCSR*, *RCSRH*, *RSB* formats, but include some general comments.

On **M2**, we notice *RSB* as the fastest format 5 times out of 12; on **M4**, 4 times. Here, *RCSRH* is the fastest in 7 cases; in all cases, very near to *RSB*. On **M4**, we see a similar situation, but notice a performance degradation in some additional cases: they are due to the poor partitioning problem discussed in §4.4.2. In no case *RCSR* was the fastest format for symmetric matrices (exception made for the poorly scaling three matrices) on **M4**: (*crankseg_1*, *ct20stif*, *F1*).

4.4.4 Conclusions From the Introduction of *COO* Leaves

In these sections, we have shown a possible improvement of our BLAS-oriented recursive storage for sparse matrices. We have found that by using index compression and format diversification techniques, we can improve the floating point performance of *SpMV*. We have also found that for unsymmetric matrices, the performance of our modified format (*RSB*) is comparable to that of a scalable sparse matrix format (*CSB*: currently for unsymmetric only). In the comparison with *RCSR* and *CSB*, we noticed some particular cases that expose *weak points* of both *RSB* and *RCSR*; consequently allowing us to identify room for further improvement: (i) To redefine our format in order to obtain some estimate on the parallelism expected from a given partitioning (in §4.4.2, we noticed that despite the apparently adequate partitioning, some instances of matrices (e.g.: smaller symmetric) did not scale on 8-threaded *SpMV*). (ii) To modify the *SpMV* algorithm to be more parallel, by working around the need for row locking (e.g.: by using temporary vectors, as *CSB* does [BFF$^+$09, § 4], although this may be challenging in our case). (iii) While our primary interest is focused on bigger matrices, tuning the partitioning algorithm for small matrices could prove useful to ensure parallelism in these cases, too. (iv) Properly subdividing matrices which are big, but with an extremely low nonzeroes/row ratio would be challenging (and fruitful), as well.

Some ideas we have introduced should be developed further. For instance, a more aggressive form of tuning could diversify index types at the *leaf level* and continue using traditional *CSR* or *COO* layouts, when profitable. Probably forthcoming architectures (with much higher number of cores, and even higher risks for stall due to higher memory latencies and longer instruction pipelines) would render such approaches advantageous.

In summary, we can state that our work illustrates that combinations of hierarchical indexing and index compression techniques can be useful to achieve high efficiency of computing on sparse matrices (on general purpose hardware). In this light, we see the *RSB* format as a candidate format for a complete multicore sparse BLAS implementation (that is, support for symmetric storage, solve operations, parallel transposed *SpMV*, and so on).

4.5 Closing Remarks

In this chapter, we have inquired about two modifications to the *RCSR* format, with experiments for the assessment of the impact on *SpMV* performance. The first change did not attempt at substantially changing any of the core algorithms involved; instead, it focused in impacting mainly on the *memory footprint* of *SpMV*, by simply employing a *shorter* numerical integer type for the column indices of the *CSR* leaves. The experiment succeeded, in that (see conclusions in §4.2.4) we found confirmation that reducing memory traffic improves efficiency, in spite of using potentially some extra integer arithmetics (for conversions and pointer arithmetics). In our second modification (see conclusions in §4.4.4), we go further applying even more the idea of saving memory bandwidth: we established a lower sparsity threshold to decide when using the traditional *COO* (in its row-major variant) rather than *CSR*. In both modifications, we had to pay attention to the way heuristics in the matrix build work: the potential parallelism and memory footprint of *SpMV* for *RSB* (the resulting, hybrid format) depend on these. These are two reasons why the next chapter is devoted to developing a flexible procedure for building *RSB* matrices. The remaining reasons for concentrating on the build process is the need for shared memory parallel build algorithms: this is a prerequisite for a completely parallel Sparse BLAS candidate format.

5 Building *RSB* Matrices

Overview

We have described basic rules for the construction of a quad-tree-based recursive *CSR* representation (*RCSR*) in §2.3.4, §2.3.3 (also published in [MFT$^+$10]). Here, we give a complete procedure for building (or *assembling*, in jargon) matrices in the *RSB* format; that is, the original quad-tree recursive representation, with applied the modifications from §4.

This chapter will go through a literature introduction in §5.1, then state some properties of the *quad-tree* structures of interest to *RSB* in §5.2, and then exposing our main idea for *RSB* matrices assembly in §5.3 and §5.4. An extensive commentary to the performance evaluation of the proposed algorithms implementation is presented in §5.4.1. In §5.5 we draw some conclusions useful for future work. The chapter terminates with the sketch of an enhanced *RSB* building procedure, in §5.6.

5.1 Literature Overview

In §2, we gave a brief historical summary of *hierarchical* data structures for the representation of sparse matrices. Interestingly enough, while hypermatrix-based approaches have been applied to sparse matrix computations, almost no research has been reported as to what concerns assembly of such matrices. We were able to find a research article in a spirit similar to ours in [GL09]. There, Gottschling and Lindbo document algorithms and discuss patterns of usage in the assembly of sparse matrices in the context of their *serial* “MTL” package. Our discussion is narrower in scope but more in-depth than theirs, as we are concerned with a single pattern of construction: the conversion of *COO* input arrays to *RSB*, to be used on multi-core computers.

5.2 Some Properties of the Quad Trees Used in RSB Matrices

Given an $m \times k$ matrix A, we build a graph structure (*quad-tree*) q with nodes corresponding to *quadrant submatrices.* The four quadrants are sized respectively (in clockwise order, from the upper left) $\lceil \frac{m}{2} \rceil \times \lceil \frac{k}{2} \rceil$, $\lceil \frac{m}{2} \rceil \times \lfloor \frac{k}{2} \rfloor$, $\lfloor \frac{m}{2} \rfloor \times \lceil \frac{k}{2} \rceil$, and $\lfloor \frac{m}{2} \rfloor \times \lfloor \frac{k}{2} \rfloor$. This subdivision (or *bipartition*) is applied recursively to the quadrants; quadrants with no nonzero are not represented. Only leaf nodes are associated with actual data arrays, while inner ones contain only pointers. A simple *cutoff* function is used to balance the tree in order to obtain *leaf submatrices* with neither too many, nor too few nonzeroes. Fig. 5.27 depicts a matrix subdivided into *RSB.*

Let us now review some properties of our quad-trees, which will be useful during the discussion of matrix assembly.

Let us call q_h the *complete* quad-tree of height h; that is, the quad-tree having $N_i(q_h) \overset{def}{=} \sum_{i=0}^{h-1} 4^i$ intermediate nodes and $N_l(q_h) \overset{def}{=} 4^h$ leaf nodes, We indicate with $H(q)$ the height of quad-tree q. We assume that any quad-tree could be constructed by adding nodes to the singleton quad-tree q_0 (the one which is associated to the entire matrix). Let Q be the set of quad-trees with height ≥ 1. We call q' a $k-$*derivation* (or *derivation*, for short, if we ignore k) of quad-tree q, if q' can be built from q, by making one leaf an intermediate node, and adding $1 \leq k \leq 4$ leaves. We call q' an *indirect derivation* of quad-tree q, if q' can be built from q after a sequence of derivations. Observe that if q' is a $k-$derivation of q, then $N_i(q') = N_i(q) + 1$, and $N_l(q') = N_l(q) + k - 1$.

Property 1. *For any q among the possible quad-trees with height 1, we have $\frac{N_i(q)}{N_l(q)} \geq \frac{1}{4}$, and $\frac{N_i(q_1)}{N_l(q_1)} = \frac{1}{4}$.*

Proof. By explicit enumeration of possible cases. □

Property 2. *For any $q \in Q$ with $H(q) > 1$,we have $\frac{N_i(q)}{N_l(q)} \geq \frac{1}{4}$*

Proof. Let q be a quad-tree having $\frac{N_i(q)}{N_l(q)} < \frac{1}{4}$, necessarily a derivation of a quad-tree q' having $\frac{N_i(q')}{N_l(q')} \geq \frac{1}{4}$. In the case q is a $k-$derivation of q', indicating $i = N_i(q'), l = N_l(q')$, we have $\frac{i}{l} \geq \frac{1}{4}$ and $\frac{i+1}{l-1+k} < \frac{1}{4}$. But this implies $4i - l \geq 0$ and $4i - l < k - 5$, which is impossible, for $1 \leq k \leq 4$. In the case q is an indirect derivation of q', it must be a derivation of some quad-tree q" having $\frac{N_i(q")}{N_l(q")} < \frac{1}{4} \leq \frac{N_i(q')}{N_l(q')}$, but existence of such q" is impossible, as we have seen. □

If some internal node of q has one child only, we call q *degenerate* (with some terminology abuse; see Knuth [Knu97, Sec. 2.3.4.5]).

Property 3. *For any sparse matrix M with no empty rows, if its corresponding quad-tree q is not degenerate, we have* $\frac{N_i(q)}{N_l(q)} \leq 1$.

Proof. Since M has no missing rows, it has some leaf node of q covering each row interval. Since q is not degenerate, at each level > 1, there are at least two nodes, or no node at all. Therefore, quad-tree q can be built by inserting additional $k \geq 0$ leaves to some binary tree q'. A non degenerate binary tree q' has $N_l(q') = N_i(q') + 1$, So we have $\frac{N_i(q)}{N_l(q)} = \frac{N_i(q')}{N_i(q')+k+1}$, whose upper limit is 1, for $k = 0$, and $N_i(q) \to \infty$. □

Property 3 guarantees that for *non degenerate* quad-trees, there will be no more internal nodes than leaves; nevertheless, we do allow degenerate trees in come of our *RSB* matrices, since this simplifies some implementation details.

5.3 Overview of *COO* to RSB Conversion

In §5.4, we will describe in detail our approach for the conversion of an $m \times k$ matrix A with *nnz* nonzeros, expressed in (row-major sorted) *COO* (*IA*, *JA* coordinate arrays and the *VA* numerical values array; see §1.1) into *RSB* order.

In this section, we sketch briefly the whole process, to allow the impatient reader to grasp the main ideas behind it.

The goal of the proposed procedure is to build a quad-tree structure for A, allocating a *small* number of auxiliary structures for the submatrix nodes, and *reusing* input arrays *IA*, *JA*, *VA*. We require input elements in row major order, both because the user application often produces input in that order, and because it gives us a starting point for matrix assembly.

The core of our procedure essentially lies in two stages: *subdivision* and *shuffle*. The first stage analyzes the input arrays, collects structure information,

$$A = \begin{pmatrix} \mathbf{0.66667}^{(1)} & 0.36656^{(2)} & 0.30011^{(3)} & 0.36656^{(4)} & 0.30011^{(5)} \\ \mathbf{0.10004}^{(6)} & 0.53341^{(7)} & -1^{(8)} & 0.20007^{(9)} & 0 \\ \mathbf{0.12219}^{(10)} & 0 & 0.5777^{(11)} & 0 & 0.24437^{(12)} \\ \mathbf{0.05002}^{(13)} & 0.10004^{(14)} & 0 & 0.28331^{(15)} & 0.18328^{(16)} \\ \mathbf{0.06109}^{(17)} & 0 & 0.12219^{(18)} & 0.15006^{(19)} & 0.27224^{(20)} \end{pmatrix}$$

IP=(1 6 10 13 17 21)

Figure 5.1: Row pointers creation, during *RSB* assembly of matrix *cage3**. Matrix entries displayed with a bold typeface are the ones pointed by the row pointers array. The last array entry points to the first index *after* the last nonzero; that is, $nnz + 1$.

and produces a candidate quad-tree for a partitioning in *submatrices*. The second stage *shuffles* the input *COO* arrays in an order which is Z^b (recall §2.3.3) among *submatrices*, but still row-major at the submatrix level.

As an example, we illustrate the first stage with three pictures: Fig. 5.1 Fig. 5.2. Fig. 5.3, and then the second stage in Fig. 5.4.

$$A = \begin{pmatrix} \mathbf{0.66667}^{(1)} & 0.36656^{(2)} & 0.30011^{(3)} & \mathbf{0.36656}^{(4)} & 0.30011^{(5)} \\ \mathbf{0.10004}^{(6)} & 0.53341^{(7)} & -1^{(8)} & \mathbf{0.20007}^{(9)} & 0 \\ \mathbf{0.12219}^{(10)} & 0 & 0.5777^{(11)} & 0 & \mathbf{0.24437}^{(12)} \\ \mathbf{0.05002}^{(13)} & 0.10004^{(14)} & 0 & \mathbf{0.28331}^{(15)} & 0.18328^{(16)} \\ \mathbf{0.06109}^{(17)} & 0 & 0.12219^{(18)} & \mathbf{0.15006}^{(19)} & 0.27224^{(20)} \end{pmatrix}$$

LP=(1 6 10 13 17 21)
RP=(4 9 12 15 19 21)

Figure 5.2: First vertical split computed on matrix *cage3**. Left and right pointers arrays are shown. Notice the boldface entries in the matrix; these are the first entries on their respective rows, in the sparse *CSR* representation.

The very first step (Fig. 5.1) consists in filling a *row pointers* array with a count of elements on each row.

With this information, quadrants for subdivision are identified (Fig. 5.2) and if possible, compressed sparse rows info is produced for quadrants (Fig. 5.3).

After these steps, a quad-tree structure is already built, and knowledge about the location of the actual data arrays in the input is ready. Therefore, input arrays are *shuffled* in-place and transformed in a way to obtain the whole matrix laid in the *RSB* layout.

$$A = \begin{pmatrix} \mathbf{0.66667}^{(1)} & 0.36656^{(2)} & 0.30011^{(3)} & \mathbf{0.36656}^{(4)} & 0.30011^{(5)} \\ \mathbf{0.10004}^{(6)} & 0.53341^{(7)} & -1^{(8)} & \mathbf{0.20007}^{(9)} & 0 \\ \mathbf{0.12219}^{(10)} & 0 & 0.5777^{(11)} & 0 & \mathbf{0.24437}^{(12)} \\ \mathbf{0.05002}^{(13)} & 0.10004^{(14)} & 0 & \mathbf{0.28331}^{(15)} & 0.18328^{(16)} \\ \mathbf{0.06109}^{(17)} & 0 & 0.12219^{(18)} & \mathbf{0.15006}^{(19)} & 0.27224^{(20)} \end{pmatrix}$$
IA_{TMP}=(1 6 10 13 4 9 12 15 0 0 0 0 0 0 0 0 0 0 0 0)
NV=(8 4 4 4)

Figure 5.3: Information from the first matrix split is collected as compressed rows pointers, and stored in a *nnz*-sized array. Notice that many row pointers are set to zero; this is because their respective submatrices *CSR* representation does not fit in their array portion, which is proportional to their nonzeroes count. These matrices will be copied as *COO* in a later phase.

Our sample matrix in the *RSB* layout is shown in Fig. 5.4.
Section §5.4 will deal with all the details of the proposed procedure.

5.4 Assembling RSB from Sorted *COO*

Unless otherwise stated, in the following, by *matrix* we will refer to A only, and denote as a *submatrix* any of the *quadrant submatrices* obtained by recursive bipartitioning (defined in §5.2). In the algorithms we expose, we assume no duplicates in the input, although duplicates actually occur in publicly available matrices (like ones from the University of Florida sparse matrix collection; see Davis [Dav10]), and thus should be dealt with[1].

There are three stages of assembly: first the *subdivision* of A in *COO_to_RSB_s*, where the input is repeatedly scanned, and a quad-tree structure is built; then the *shuffling* of rows laid in *COO* order to the rows of *RSB* submatrices (Fig. 5.12, 5.13), and finally *compression of indices* in *RSB_Leaf_Switch* (Fig. 5.14). Accordingly, we break down the *RSB* assembly pseudo code into three listings, called from procedure *COO_to_RSB*, in Fig. 5.5.

Procedure *COO_to_RSB_s* (Fig. 5.6), performs a cycle, identifying bounds for candidate submatrices. This information is stored in auxiliary arrays L, M, R. A *row pointers* array P is constructed (line 5), kept and returned for later usage. At each iteration, the *largest open submatrix* s (in terms of number of nonzeroes)

[1]We do not include here algorithms and timings for duplicate/zeros removal, although we have implemented and applied them in our experiments.

$$A = \begin{pmatrix} \begin{pmatrix} 0.66667^{(1)} & 0.36656^{(2)} & 0.30011^{(3)} \\ 0.10004^{(4)} & 0.53341^{(5)} & -1^{(6)} \\ 0.12219^{(7)} & 0 & 0.5777^{(8)} \end{pmatrix} & \begin{pmatrix} 0.36656^{(9)} & 0.30011^{(10)} \\ 0.20007^{(11)} & 0 \\ 0 & 0.24437^{(12)} \end{pmatrix} \\ \begin{pmatrix} 0.05002^{(13)} & 0.10004^{(14)} & 0 \\ 0.06109^{(15)} & 0 & 0.12219^{(16)} \end{pmatrix} & \begin{pmatrix} 0.28331^{(17)} & 0.18328^{(18)} \\ 0.15006^{(19)} & 0.27224^{(20)} \end{pmatrix} \end{pmatrix}$$

IA_{RSB}=((1 4 7 9 0 0 0 0)(1 1 2 3)(1 1 2 2)(1 1 2 2))
JA_{RSB}=((1 2 3 1 2 3 1 3)(1 2 1 2)(1 2 1 3)(1 2 1 2))
VA_{RSB}=((0.66667 0.36656 0.30011 0.10004 0.53341 -1 0.12219 0.5777)(0.36656 0.30011 0.20007 0.24437)(0.05002 0.10004 0.06109 0.12219)(0.28331 0.18328 0.15006 0.27224))

Figure 5.4: After shuffle, *cage3** is represented in the *RSB* layout in the original three *COO* arrays. These submatrices are kept in the original input arrays, and their offsets are kept in their enclosing quad-tree father node. Notice that two submatrices are represented with *CSR*, and two as *COO*.

is selected; then it is analyzed, and either subdivided in quadrants (and marked as *closed node*) or marked as a *closed leaf*. In either case, each cycle *closes* submatrix s and *opens* up to four submatrices. Therefore, the loop iterates a number of times equal to the number of the nodes (both inner and leaf) in the produced quad-tree.

In Fig. 5.7 we present the cutoff function δ_r which decides if subdivision of s should proceed.

Since the input *COO* arrays are row-major sorted, in order to identify quadrants of s in them, we need to mark, for each row, indices for: the leftmost element of the two left quadrants, the leftmost of the two right quadrants, and the first one after the rightmost of the two right quadrants; that is, pin-point *subrows* in each quadrant. To this end, IA and JA are scanned in *Subrow_Split*, and subrows information is stored in the three *row pointers arrays* L, M, R. Row pointers data will be reused when assembling submatrices in *CSR*. The first invocation of *Subrow_Split* requires L, R for the whole A in order to compute the first *middle row pointers* array M. Notice that for any row i of A, $L[i+1] \equiv R[i]$. For this reason, before entering the loop, we pre-compute a single row pointers array P, and set the initial L, R as *pointer aliases* of P. That is, P can serve as L, and aliased after its first element, as R does; in Fig. 5.6 and 5.11, we have used "$\overset{P}{\leftarrow}$" to signify pointer aliasing[2]. P is computed by *COO_RowP*, listed in

[2]For more details about our notation conventions, see §D.

Figure 5.5: $COO\_to\_RSB(IA, JA, VA)$.

```
1  /*Matrix $A$ is expressed using arrays $IA, JA, VA$*/
2  Instantiate the root matrix node $s_A$, marked RSB and “open”
3  $[P, s_A] \leftarrow COO\_to\_RSB\_s(s_A, IA, JA)$/*Symbolic subdivision*/
4  /*Now $s_A$ is the root of a quad-tree for $A$, with empty leaves*/
5  /*$P$ is a rows pointer array for $IA, JA, VA$*/
6  $COO\_to\_RSB\_V(s_A, P, VA)$    /*Numerical arrays shuffling*/
7  $COO\_to\_RSB\_J(s_A, P, JA)$ /*Indices shuffling/displacement*/
8  /*$P$ is no longer needed and $IA, JA, VA$ are in RSB order*/
9  $RSB\_Leaf\_Switch(s_A)$     /*Indices switch*/
10 /*A number of leaf matrices has halfword indices, now.*/
11 **return** $s_A$  /*Return $s_A$, now quad-tree for $A$*/
```

Fig. 5.8.

After boundaries are identified, and nonzeroes counts are known for each quadrant, at line 17, we invoke the *RSB_Split_Node*. It will add an *open* leaf submatrix for each non-empty quadrant, and copy the L, M, R arrays in appropriate offsets of the *IA* array. In this way, *IA* is used for storing submatrices rows information, and subsequent invocations of *Subrow_Split* will use the L, R arrays recovered from there.

In the case the δ_r does not make s a candidate for subdivision, s gets *closed* as a leaf matrix, and marked to contain data in the *CSR* or *COO* format (depending on the available index space; lines 19-23). In the case s is the root node for A (s_A), and fitting *CSR* arrays ($nnz > m$), L (aliasing P) is copied at the appropriate offset of *IA*, overwriting original row indices (not needed anymore).

After assembling the quad-tree for the s_A, the original *JA*, *VA* arrays storing column indices and values of the matrix coefficients are still unmodified, and ready for being displaced to their destination location. The *IA* array, instead, has been overwritten. For submatrices marked for *CSR* storage, *IA* already stores a *row pointers array*, which a *CSR* representation requires. For submatrices marked for *COO* storage, the relevant subarrays for *IA* could have been overwritten during parent node subdivision, and therefore they should be reinitialized to their original values. Actually, each submatrix node has information on the count of nonzero elements in its own quadrants. Recall, that in *RSB_Split_Node*, the *nonzero offset* of each submatrix in the quad-tree representation was computed. Now, each submatrix s could be extracted to a temporary storage, row by row, from the original matrix specified in subsequent rows,

Figure 5.6: $COO\_to\_RSB\_s(s_A, IA, JA)$.

```
$Q_{nnz} \leftarrow [0,0,0,0]$ /*nonzeroes count for quadrants */
Allocate four $(s.m+1)$-sized arrays $L, M, R, P$
/*CS: Cache(s) Size, ES(= 8 for double): Element Size*/
$s_A.N_S \leftarrow 0$; $s_A.MAX_S \leftarrow (s_A.nnz \cdot ES)/(CS/N_{threads})$
$COO\_RowP(IA, JA, P, s_A.nnz, s_A.m)$ /*fill row pointers in P*/
$s_A.L \overset{p}{\leftarrow} P$; $s_A.R \overset{p}{\leftarrow} P+1$
/*$s_A.L$ points to row beginnings, $s_A.R$ points to row endings (aliasing the
second element of P)*/
while Some leaf submatrix is still "open" do
    $s_A.N_S \leftarrow s_A.N_s + 1$;
    $s \leftarrow$ "largest by nnz" open submatrix
    if $\delta_r(s.m, s.k, s.nnz, CS, ES, WS)$ then
        /*copy subrow pointers stored in $s.IA, s.JA$ */
        $L \leftarrow s.IA$; $R \leftarrow s.JA$;
        /*get quadrants info $Q_{nnz}$, fill middle pointers array M*/
        $Q_{nnz} \leftarrow Subrow\_Split(s, L, R, M, JA)$
        /*split s, appending up to four quadrant submatrices*/
        $RSB\_Split\_Node(s, Q_{nnz}, L, M, R, IA, JA)$
    else
        /*closing (marking as terminal)*/
        if $s$ is $s_A$ and $s.nnz \geq s.m+1$ then $s.IA \leftarrow L$
        /*For $s_A$, a copy is necessary.*/
        if $s.nnz \geq s.m+1$ then Mark as CSR
        else Mark as COO
    end
end
return $[P, s_A]$/*Arrays $L, M, R$ can be freed.*/
```

Figure 5.7: $\delta_r(m, k, n, CS, ES, WS)$.

```
/*WS(= 4): Word Size of index element, $\mu = 3$ */
if $s_A.N_S \geq s_A.MAX_S$ then return False
if $n \cdot ES > 2 \cdot CS$ and $m < 2^{16}$ and $k < 2^{16}$ then return True
if $(ES\ (2 \cdot n + m) + WS \cdot (m+n)) > \alpha\ CS$ and $n/m > \mu$ then return
True
return False
```

Figure 5.8: $COO\_RowP(IA, JA, P, nnz, m)$.

```
1  $P[:] \leftarrow \mathbf{0}$ /* fill with zeros*/
2  for $n \leftarrow 0$ to $nnz - 1$ do  $P[IA[n]+1] \leftarrow P[IA[n]+1]+1$
3  for $i \leftarrow 0$ to $m - 1$ do  $P[i+1] \leftarrow P[i+1] + P[i]$
4  /*for each $i$, $P[i]$ now has the offset of row $i$ in $IA, JA$*/
```

Figure 5.9: $Subrow\_Split(s, L, R, M, JA)$.

```
1  $n_{00} \leftarrow 0$; $n_{01} \leftarrow 0$; $n_{10} \leftarrow 0$; $n_{11} \leftarrow 0$;
2  for $i \leftarrow 0$ to $\lfloor (s.m+1)/2 \rfloor$ do
3      $M[i] \leftarrow Search(JA, L[i], R[i], s.koff + \lceil s.k/2 \rceil)$
4      $n_{00} \leftarrow n_{00} + (M[i] - L[i])$; $n_{01} \leftarrow n_{01} + (R[i] - M[i])$
5  end
6  for $i \leftarrow \lceil (s.m+1)/2 \rceil$ to $s.m - 1$ do
7      $M[i] \leftarrow Search(JA, L[i], R[i], s.koff + \lceil s.k/2 \rceil)$
8      $n_{10} \leftarrow n_{10} + (M[i] - L[i])$; $n_{11} \leftarrow n_{11} + (R[i] - M[i])$
9  end
10 return $[n_{00}, n_{01}, n_{10}, n_{11}]$
```

Figure 5.10: $Search(JA, l, r, h)$.

```
1  Binary search for the smallest $m$ such that $JA[m] \geq h$ and $l \leq m \leq r$
2  return $m$
```

at the submatrix offset $s.nzoff$ (computed by *RSB_Split_Node*, in Fig. 5.11). To keep the shuffling algorithm simple, we have chosen to allocate two temporary JA_t and VA_t arrays; gather there the displaced rows for coefficients and indices, and copy back to JA, VA. Since different submatrices should be laid in separate intervals of JA and VA, the shuffling algorithm can be parallelized on a submatrix basis in a parallel cycle. Once shuffled, the temporary arrays are copied back using a simple OpenMP-parallel wrapper around the standard `memcpy` ([pos08]) function.

Figure 5.11: $RSB\_Split\_Node(s, Q_{nnz}, L, M, R, IA, JA)$.

```
$Q \leftarrow$ [...] /*allocate a submatrix structure for each nonempty quadrant of s; then for each quadrant $q_{ij}$, set info for nonzeroes, dimensions, and row,column,nonzeroes offsets relative to the whole matrix; then copy portions from the subrow pointer arrays from $L, M, R$*/
if $n_{00} > 0$ then
    $q_{00}.m \leftarrow \lceil s.m/2 \rceil$; $q_{00}.k \leftarrow \lceil s.k/2 \rceil$;
    $q_{00}.moff \leftarrow s.moff + 0$; $q_{00}.koff \leftarrow s.koff + 0$;
    $q_{00}.nzoff \leftarrow s.nzoff + 0$; $q_{00}.nnz \leftarrow n_{00}$;
    $q_{00}.IA \overset{p}{\leftarrow} IA + q_{00}.nzoff$; $q_{00}.JA \overset{p}{\leftarrow} JA + q_{00}.nzoff$
    if $q_{00}.nnz > 2 \cdot q_{00}.m + 2$ then
        $q_{00}.IA \leftarrow IL[1 : q_{00}.m]$; $q_{00}.JA \leftarrow IM[1 : q_{00}.m]$;
    end
end
if $n_{01} > 0$ then
    $q_{01}.m \leftarrow \lceil s.m/2 \rceil$; $q_{01}.k \leftarrow \lfloor s.k/2 \rfloor$;
    $q_{01}.moff \leftarrow s.moff + 0$; $q_{01}.koff \leftarrow s.koff + q_{00}.k$;
    $q_{01}.nzoff \leftarrow s.nzoff + n_{00}$; $q_{01}.nnz \leftarrow n_{01}$;
    $q_{01}.A \overset{p}{\leftarrow} IA + q_{01}.nzoff$; $q_{01}.JA \overset{p}{\leftarrow} JA + q_{01}.nzoff$
    if $q_{01}.nnz > 2 \cdot q_{01}.m + 2$ then
        $q_{01}.IA \leftarrow IM[1 : q_{01}.m]$; $q_{01}.JA \leftarrow IR[1 : q_{01}.m]$;
    end
end
. . . /*And so on for $q_{10}, q_{11}$.*/ . . .
```

The shuffling procedures for JA (Fig. 5.13) and VA (Fig. 5.12) are similar. For VA (*COO_to_RSB_V*), only rows shuffling is needed, but for JA (*COO_to_RSB_J*), besides shuffling, we need also to adjust indices relative to the submatrix location, and restore indices of *IA*. After the shuffling phase, submatrices are either stored as *fullword* (by default, 32 bit) *COO* or *CSR*. *RSB* (see §4.3) allows smaller leaves to have 16 bit coordinate (for *COO*) or column (for *CSR*) indices. For this, we use a separate procedure, RSB_Leaf_Switch, operating an *in place* conversion on the arrays of the candidate submatrices. Note that interleaving shuffling and conversion could save a substantial fraction of memory accesses; however the constructor logic would be much more involved. After this (last) step, the matrix is assembled as *RSB* and ready for use.

The presented assembly procedure consists of a *serial* stage (*subdivision*), followed by two stages exploiting parallelism (*shuffling* and *conversion*). Initially, we considered to propose a parallel subdivision step. However, we observed that this would require us to use more complicated techniques, and would also entail differences in the computed partitions. For instance, we could have let threads subdivide the matrix concurrently, but non-determinism in the order of subdivision could lead to non-deterministic quad-tree shape/matrix partitioning. In such a case, we would have either to accept the algorithm as non-deterministic (which we did not want), or use complicated *backtracking* techniques to revert unnecessarily subdivided submatrices and an equivalent tree. On the other hand, we have found strategies for the parallelization of the current subdivision algorithm routines (based on fine-grained parallelism) to be problematic regarding synchronization, and therefore shortsighted, in the perspective of many-core computations, expected in forthcoming computers. Therefore, for the time being, we have chosen a simple serial strategy, and left other enhancements for future developments. Indeed, besides being serial, the subdivision stage faces a growing amount of work, as more subdivisions are performed on a matrix; and thus, it will slow down further, the more threads will participate in the *SpMV* computation (recall line 4 in Fig. 5.6). Each subdivision of a submatrix s requires (a) the copy of two arrays, (b) $s.m$ binary searches during split, and (c) one array write per search. In the worst case, this involves about $s.m$ random accesses in the binary searches, (which perform non-linear accesses), but the remaining accesses are linear, and could be performed taking advantage of the available prefetching engine on the CPU.

Analysis of the complexity of subdivision is beyond the scope of this study; a gross, pessimistic estimate we could provide for the memory traffic would be up to $o(h \cdot nnz)$ memory writes (where h is the height of the quad-tree). This would be the case where all of the submatrices would fit exactly as *CSR*: if some were *COO*, binary searches would be performed on their parent matrices, but with no subsequent row pointers copy (matrices are assigned as *COO* if they don't fit *CSR*, with no further subdivision). If some submatrices had rows

Figure 5.12: $COO\_to\_RSB\_V(s_A, P, VA)$.

```
Allocate a temporary vector $VA_t$, fitting $VA$.
parallel foreach $s \in S$ do
    $VA_s \overset{p}{\leftarrow} VA_t[s.nzoff]$
    if $s.nnz \geq 2 \cdot s.m + 2$ then
        for $i \leftarrow 0$ to $s.m - 1$ do
            Append subrow $VA[s.L[i] : s.R[i]]$ to $VA_s$
        end
    else
        for $i \leftarrow 0$ to $s.m - 1$ do
            $l \leftarrow P[s.moff + i]; r \leftarrow P[s.moff + i + 1]$
            $l \leftarrow Search(JA, l, r, s.koff)$
            $r \leftarrow Search(JA, l, r, s.koff + s.k)$
            Append subrow $VA[l : r]$ to $VA_s$
        end
    end
end
$MEMCPY\_Parallel(VA, VA_t)$/*$VA \leftarrow VA_t$*/
```

Figure 5.13: $COO\_to\_RSB\_J(s_A, P, JA)$.

```
Allocate a temporary vector $JA_t$, fitting $JA$.
parallel foreach $s \in S$ do
    $JA_s \overset{p}{\leftarrow} JA_t[s.nzoff]$
    if $s.nnz > 2 \cdot s.m + 2$ then
        for $i \leftarrow 0$ to $s.m - 1$ do
            Append subrow $JA[s.L[i] : s.R[i]]$ to $JA_s$
            Make a CSR row pointer in IA, using L, R
        end
    else
        for $i \leftarrow 0$ to $s.m - 1$ do
            $l \leftarrow P[i]; r \leftarrow P[i + 1]$
            $l \leftarrow Search(JA, l, r, s.koff)$
            $r \leftarrow Search(JA, l, r, s.koff + s.k)$
            Append subrow $JA[l : r]$ to $JA_s$
            if $s.nnz < s.m + 1$ /*COO case*/ then
                Set array $s.IA$ with value $i$
            else
                Make a CSR row pointer in IA, using L, R
            end
        end
    end
    Adjust $s.JA$ indices, by subtracting the offset $s.koff$.
end
$MEMCPY\_Parallel(JA, JA_t)$ /*$JA \leftarrow JA_t$*/
```

Figure 5.14: $RSB\_Leaf\_Switch(s_A)$.

```
parallel foreach leaf node s of quad-tree $s_A$ do
    if Marked for halfword indices then
        if CSR format then
            Convert $JA$ into using 16 bit indices, in place
        end
        if COO format then
            Convert $IA$, $JA$ into using 16 bit indices, in place
        end
    end
end
```

denser ($s.nnz > s.m + 1$), it would mean that only $O(s.m + 1)$ elements would be moved (out of $s.nnz$).

The shuffle stage is different: it involves two transfers of contents of arrays *VA* and *JA*; and between m and nnz element moves for *IA*. If not coupled with the copy operation, the index adjustment for *JA* accounts for further, up to $O(nnz)$, accesses; similarly for restoring the *IA* arrays of *COO* leaves. Similarly, the complexity of the compression stage involves modifications of up to $2 \cdot nnz$ memory locations (once). Besides the `memcpy`-like operations, when shuffling the *COO* submatrices, the *JA* array would be *binary-searched* repeatedly for the identification of subrows bounds (after determining bounds for search using P). The same binary-search based algorithm is needed for the *CSR* submatrices having $s.nnz \leq m$ (since the corresponding *IA* subarray would not contain both right and left subrows pointer arrays). For *CSR* leaves having $s.nnz > s.m$, right and left subrow pointers are recovered from *IA*, subrows in *JA* and *VA* are located, and no search is needed at all. Notice the independence from the quad-tree height (and thus, from the matrix size).

5.4.1 Experimental Results

For space reasons, we won't be able to present a comprehensive analysis of the constructor performance, and thus we will focus on the most important topics (please refer to §A.5 for a full description of the experimental setup we used). Our exposition is geared towards iterative methods; here, the affordability of the constructor code is inversely proportional to the number of *SpMV*'s that are expected to be performed after matrix instantiation. Hence, performance

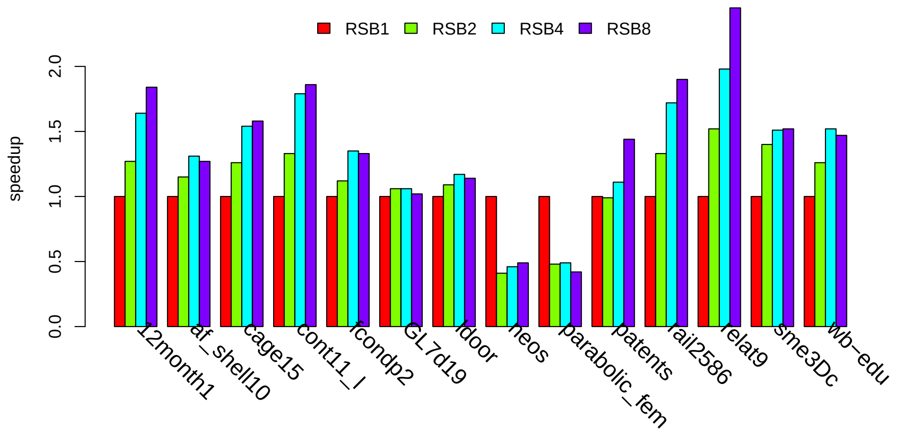

Figure 5.15: *RSB* matrix assembly scaling on **M2**.

profiles for both *SpMV* and construction operations are needed. We will thus present the constructor performance considering two metrics: the number of *SpMV* that are time-equivalent to a constructor run on the given matrix, and the scalability of the constructor with respect to the single core case.

In our previous work (See §4.1, §4.3), using 8 cores on **M4** and **M2**, we have encountered a *SpMV* speedup of up to 5×. In §5.4, we have motivated the reasons for keeping a part of our constructor code serial. Therefore, the observed scalability is indeed weak, as depicted in Fig. 5.16, 5.15. We see that the maximum speedup on both machines is 2.45× on **M4** and 2.86× on **M2**; this is approximately half than observed for the *SpMV*. We notice the best speedup for matrices *relat9* and *rail2586* on **M4**; *patents* and *parabolic_fem* on **M2**. In two cases (*neos* and *parabolic_fem* on **M4**) we notice a slow-down. Due to the increasingly loaded serial stage; in both cases, this happens after a no-subdivisions instantiation, for 1-core (for space reasons, we omit graphs with submatrix counts).

Relating constructor and *SpMV* times, we notice the constructor dominating the *SpMV*, in Fig. 5.17,5.18. We observe the maximal ratio for matrix *wb-edu* (up to 52.8× on **M4**, up to 27.7× on **M2**); a minimal one for matrix *rail2586* (from

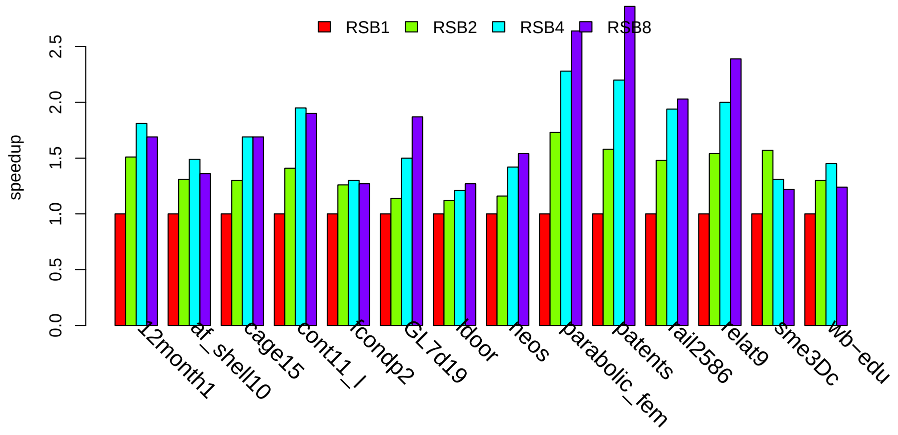

Figure 5.16: *RSB* matrix assembly scaling on **M4**.

2.8× to 4.4×, on **M4**). In two cases (matrices *cont11_l*, *patents*), it happens that the constructor and *SpMV* times keep a similar pace (around 10×, on both machines). Indeed, the *SpMV* performance of matrix *cont11_l* does not increase with more cores, and matrix *patents* gets partitioned in the same number of leaf matrices, regardless of the cores count. We notice worse ratios for bigger matrices: *cage15*, *wb-edu*, and *GL7d19*. Here, *patents* is big, but it performs *SpMV* exceptionally slow (see §4.4).

Let us break down the constructor performance in the serial (*subdivision*) and parallel (*shuffle* and *conversion*–we will include this last one in the shuffle results, for convenience) stages. As discussed in §5.4, the subdivision code is expected to perform a number of passes on the input growing with the number of threads available for *SpMV*. In Fig. 5.19 and 5.20, we see the *scaling-down* of subdivision performance; we encounter a near-to 5-fold slow-down for matrices *parabolic_fem* and *neos*. This is due to no subdivision being performed in the 1-core case; in the remaining cases, we do not notice more than a 2-fold slow-down.

In Fig. 5.21,5.22, we can see the growing gap between the subdivision and *SpMV*. For 1 or 2 cores, this ratio is always lower than 7.0, but for more, it can

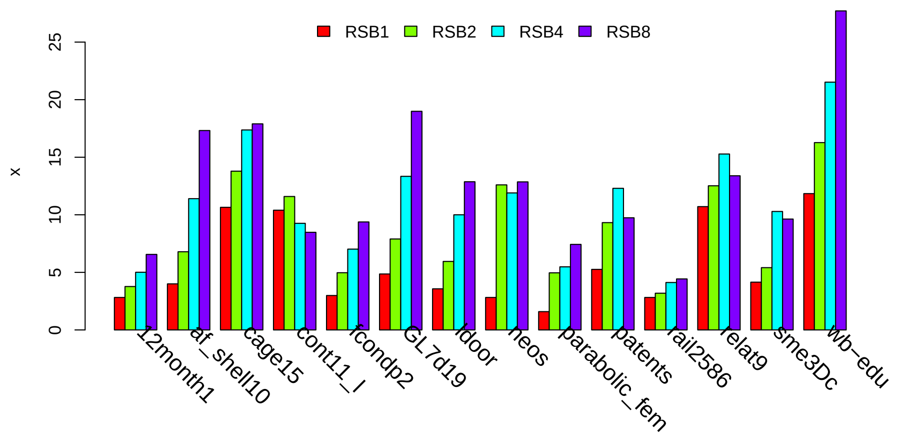

Figure 5.17: *RSB* matrix assembly to *SpMV* time ratio on **M4**.

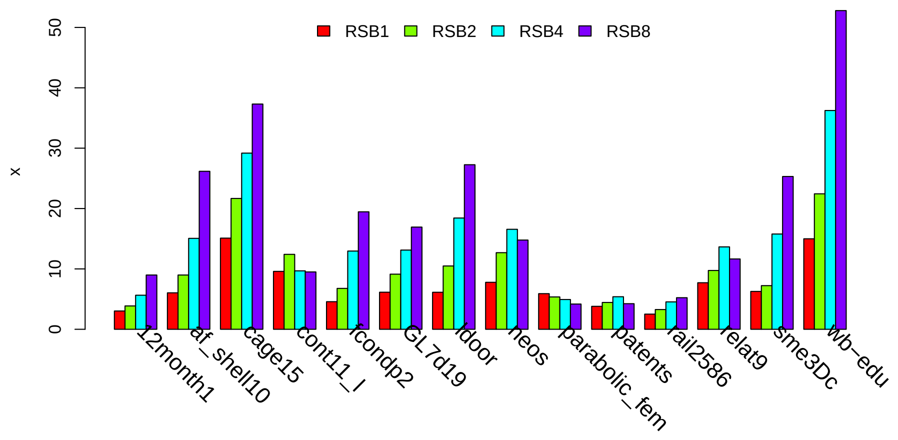

Figure 5.18: *RSB* matrix assembly to *SpMV* time ratio on **M2**.

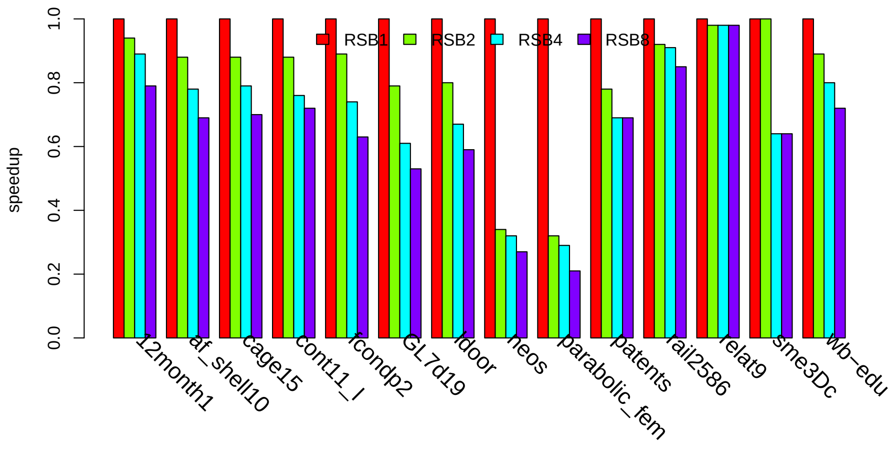

Figure 5.19: Subdivision scaling on **M4**.

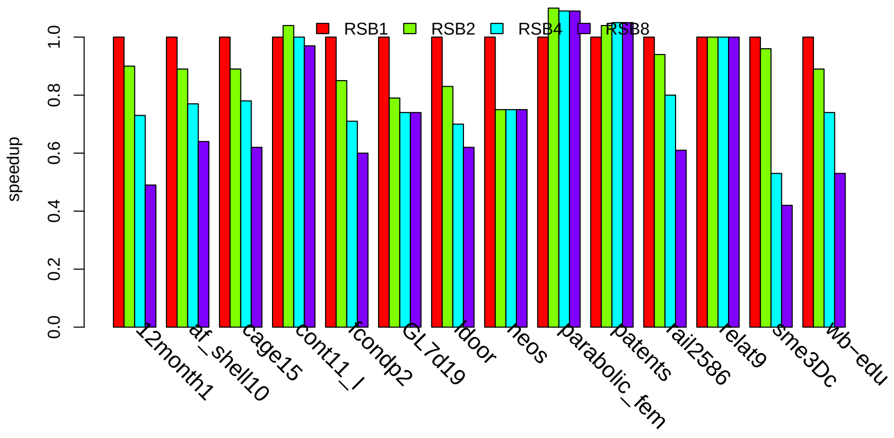

Figure 5.20: Subdivision scaling on **M2**.

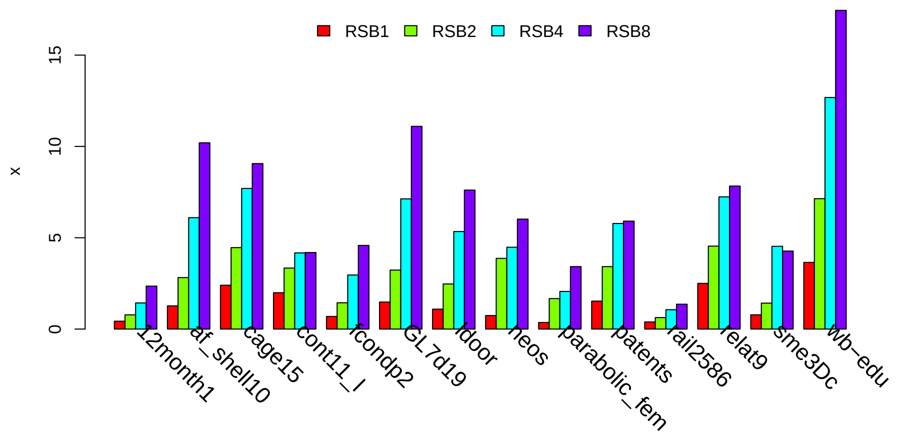

Figure 5.21: Subdivision to *SpMV* time ratio on **M4**.

grow much: for matrix *wb-edu* on **M2**, the subdivision takes 5.1× for 1 core, and up to 42.4 times *SpMV* time, for 8 cores.

On the other hand, the *shuffle* stage scales quite regularly on all matrices in the test set; see Fig. 5.23, 5.24. Recall that with higher cores counts, notwithstanding the growing number of submatrices to handle, the shuffle operation moves approximately the same amount of memory locations (see §4.3.2 for a discussion on indexing space). As noted in §5.4, in the shuffle (comprehensive of index compression) phase, the amount of involved traffic depends on the leaves format; prevalence of *COO* leaves will trigger more traffic; since compression happens after the copy operations, it contributes to additional traffic.

We also notice that the ratio of shuffle-to-*SpMV* times remains very close, regardless of the active cores count. This is satisfactory, because it indicates that both the operations scale similarly: see Fig. 5.25,5.26. Indeed, both operations seems to be memory bound; shuffle more than *SpMV*, as it doesn't involve floating point operations, which could be slower than integer operations. During stand-alone benchmarking our naive parallel `memcpy` wrapper (MEMCPY_Parallel, used in §5.4), we experienced at most 8.4GB/s on **M4**, 6.4GB/s on **M2**, and speedups respectively up to 3.1 and 2.2. We expect this limit to

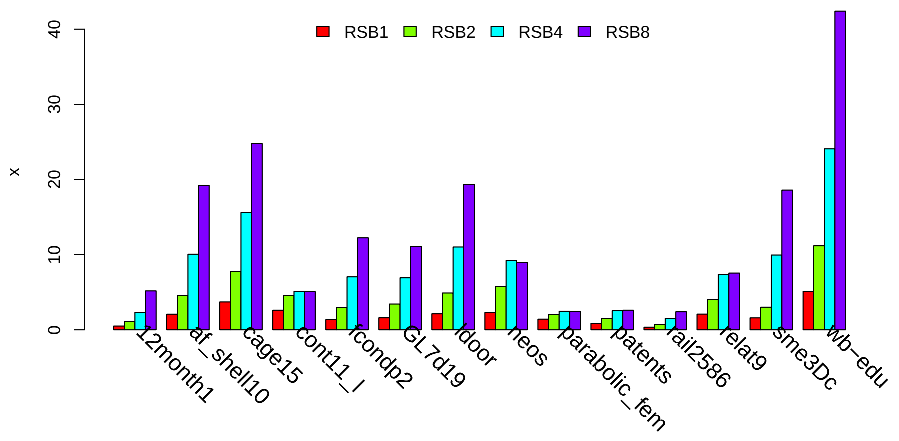

Figure 5.22: Subdivision to *SpMV* time ratio on **M2**.

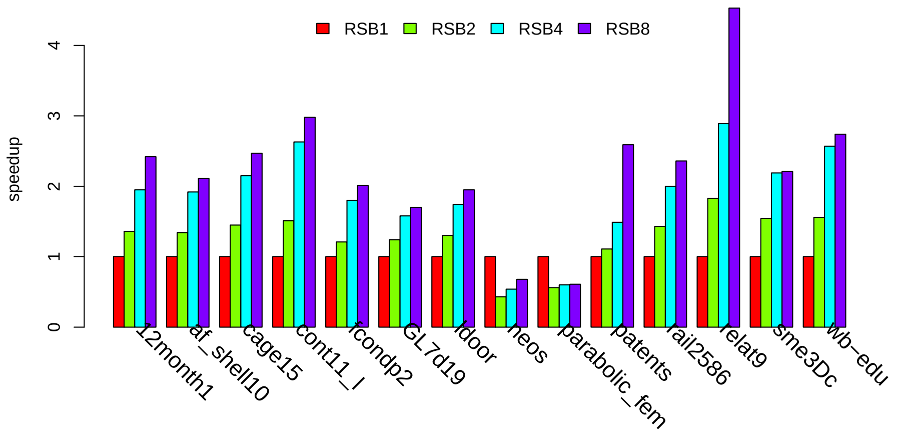

Figure 5.23: Shuffle scaling on **M4**.

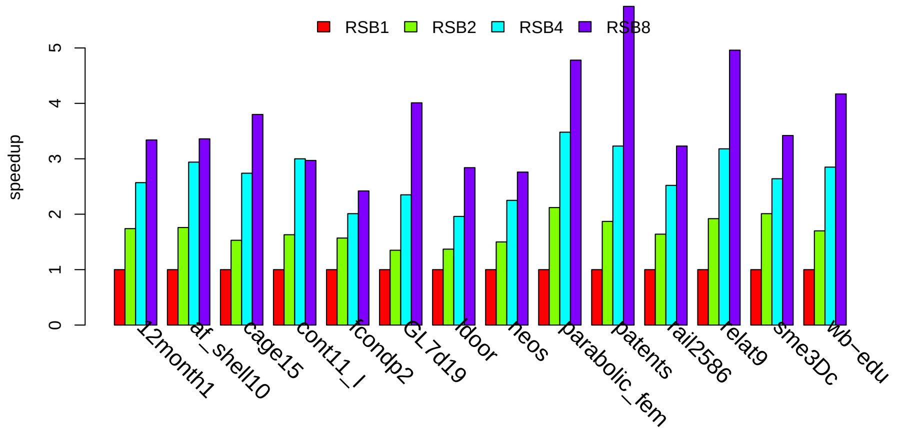

Figure 5.24: Shuffle scaling on **M2**.

contribute with a relevant fraction to the shuffle stage.

By comparing, respectively, Fig. 5.21 to Fig. 5.25 and Fig. 5.22 to Fig. 5.26, we notice that on both machines, the subdivision (serial) stage becomes dominant over the shuffle (parallel) at around 4 active threads. Clearly, this situation is not desirable in the perspective of more computing cores, so we recognize the need for a scaling parallel subdivision stage. Also, by allowing degenerate subtrees (see §5.4) input could be scanned repeatedly and generating no new subdivision; this case should also be dealt with.

5.5 Conclusions from the Serial-Parallel RSB Constructor Experiments

We have shown a multi-threaded algorithm for the instantiation of *RSB* matrices out of row major sorted *COO* arrays. Experimentally, we established that the conversion execution speed seems tightly bound to the peak memory bandwidth; even more than *SpMV*. Contrarily to previous chapters experiments, here, we did not present a *comparative* benchmarking of the matrix build phase, as

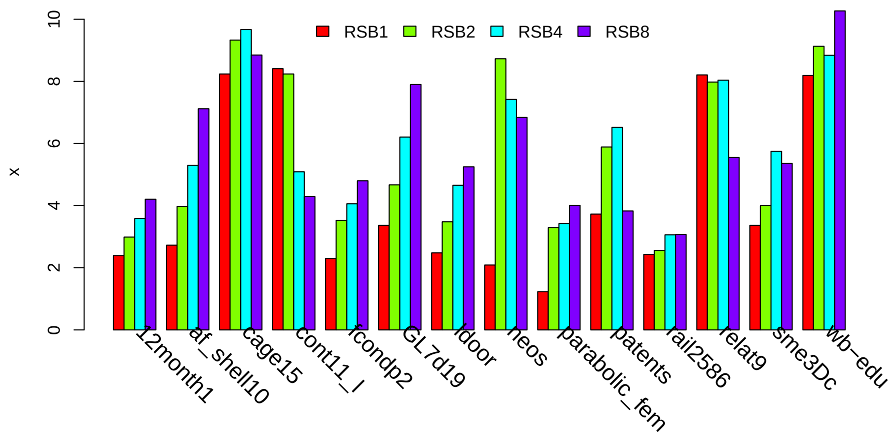

Figure 5.25: Shuffle to *SpMV* time ratio on **M4**.

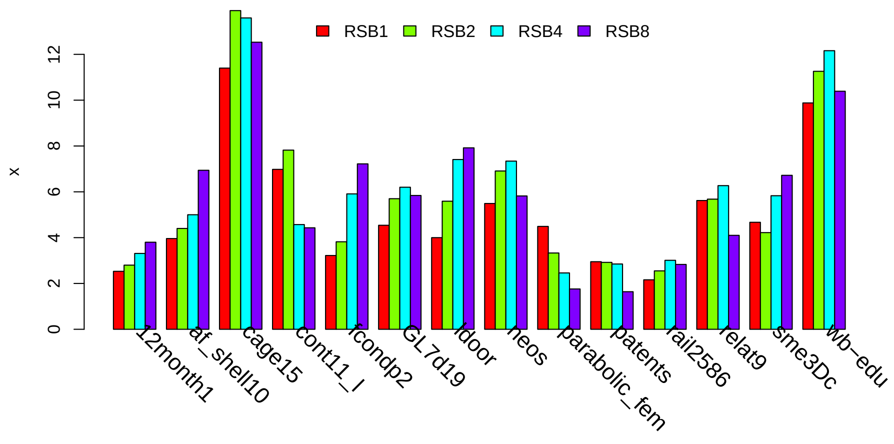

Figure 5.26: Shuffle to *SpMV* time ratio on **M2**.

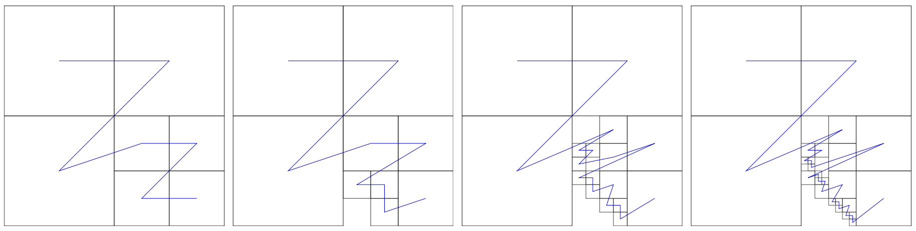

Figure 5.27: Recursive subdivisions of matrix *cont11_l* for respectively 1,2,4,8 threads on **M4**. Notice the blue line joining (nonempty) leaf submatrices in the order they are stored in the *RSB* arrays. Notice also that the more threads are active, the finer is the partitioning.

we were not aware of any non proprietary sparse matrix computations package with shared memory parallel assembly routines[3]. Our procedure features a serial *subdivision* stage, where binary search and arrays copy operations are predominant, followed by a parallel *shuffle* stage, where arrays are displaced and indices adjusted. The shuffle stage scales smoothly; its performance seems strictly memory bandwidth-bound. While shaping the subdivision stage, we observed that an efficient parallel reformulation of it would require us to modify the definition of our format. We did not want to proceed this way as we wanted it to remain comparable with our earlier work, and so we have decided to leave the subdivision serial, for now. In practice, we observed the constructor-to-*SpMV* time ratio to be 1.6..15.1 times for 1 core, 2.5..22.4/2 cores, and 4.2..52.8/8 cores. Indeed, we have observed that the serial phase begins to dominate the constructor time as soon as at about 4 threads. For this reason, we recognize need of further research to develop a scalable, parallel algorithm to perform the initial subdivision, as this is the key to a scalable *RSB* matrix constructor. We deem also interesting to study parallel conversion/extraction mechanisms for interfacing to other formats, and consider the performance impact of building preconditioners, while solving linear systems. Of course, a number of trivial but effective optimizations (see §5.4) may also be applied.

An important remark we wish to make is that, since row major sorted *COO*

[3]Actually, we measured *CSB* constructor timings with the publicly available prototype. We found that comparing our timings to that of *CSB* would be unfair, firstly because the *CSB* build routines are not parallel, and secondly because they were not optimized (being definitely much slower than ours).

input is almost equivalent to *CSR* (the two differing in the contents of one of the three arrays; see §1.2), and since the cost of a *COO-RSB* conversion exceeds from a few to several times the cost of a *SpMV*, the overhead of converting to *RSB* may be excessive for the iterative methods based solution of certain systems requiring very few steps and thus giving limited savings over *CSR*[4]. However, even in these case of few iterations per iterative method invocation, *RSB* may be advantageous over *CSR* if *repeated invocation* of the method is needed, just like it happens often during the simulation of evolving physical phenomena.

5.6 Enhancing Build Parallelism

In the preceding sections, we have presented a partially parallel algorithm for the instantiation of sparse matrices in the *RSB* format. There, we also found that the serial bottleneck was the *subdivision* phase, so here we present a parallel procedure for that phase.

Since the most time consuming task performed in the subdivision phase consists in the copy of row pointers arrays, a parallel subdivision routine should allow multiple threads into this. The copy operation occurs on ever smaller submatrices, starting with the whole matrix; parallelizing the copy of individual submatrices, however, would require repeated threads synchronization (branch/join), for each submatrix. Given the ever increasing number of cores/threads available on computers, this solution would clearly be not appealing.

The alternative approach is coarse grained: each thread would obtain, in turn, exclusive access to an existing submatrix s, and would subdivide it. Allowing multiple threads at once into this procedure requires coordinating multiple subdivisions occurring in parallel.

It is clear that by allowing some threads into getting exclusive access to submatrices, while other are still subdividing, the resulting order of submatrices' subdivision would be different from that in Fig. 5.6.

That is, our parallel subdivision procedure produces *non deterministic* results: the organization of each *RSB* matrix depends partly on the system's scheduling choices. From the computation efficiency standpoint, however, having a fully parallel matrix constructor is a very desirable feature, even at the cost of having a non deterministically built data structure.

[4]For *SpMV*/*SpMV-T* results when comparing to a high performance *CSR* implementation, see §C.

Figure 5.28: $COO\_to\_RSB\_s\_Parallel(s_A, IA, JA)$.

```
Allocate a $(s.m+1)$-sized array $P$
$COO\_RowP(IA, JA, P, s_A.nnz, s_A.m)$ /*Fill row pointers in $P$*/
$N_t \leftarrow$ available threads count; $N_c \leftarrow 0$; $N_i \leftarrow 0$; $N_o \leftarrow 1$
$N_s \leftarrow (\alpha_p \cdot s_A.nnz \cdot ES)/CS$ /*Shared variables:$Q_{nnz}, N_t, N_c, N_o, N_i$*/
begin parallel
while $N_o > 0$ or $N_i > 0$/*Some leaf submatrix is still "open"*/ do
    begin critical section
        $s \leftarrow$ "largest by nnz" open submatrix
        if $s \neq nil$/*If such submatrix exists and is available*/ then
            /*Update the in-progress, closed, open submatrices counters*/
            $N_i \leftarrow N_i + 1$; $N_c \leftarrow N_c + 1$; $N_o \leftarrow N_o - 1$;
        end
    end critical section
    if $s \neq nil$ then
        /*Is it both possible and profitable to subdivide ?*/
        $S_s \leftarrow (N_o + N_c + 4 < N_s)$ and $\delta_p(s.m, s.k, s.nnz, CS, ES, WS, s_A)$
        if $S_s =$ True then
            /*Get quadrants info $Q_{nnz} = [n_{00}, n_{01}, n_{10}, n_{11}]$*/
            $Q_{nnz} \leftarrow Subrow\_Split\_Search\_Only(s, P, IA, JA)$
            $N_n \leftarrow$count of nonempty quadrants in $s$ (in $[1...4]$)
            $S_s \leftarrow$is the partitioning in $Q_{nnz}$ balanced for $s$?
        end
        if $S_s =$ True then
            begin critical section
                $N_o \leftarrow N_o + N_n$/*Update the open submatrices count*/
            end critical section
            subdivide $s$: copy index arrays and info in the new submatrices
            begin critical section
                add the new submatrices as children of $s$
                $N_i \leftarrow N_i - 1$/*Decreasing the "in progress" counter*/
            end critical section
        else
            mark $s$ for either COO or CSR format
        end
    end
end
end parallel
```

In Fig. 5.28 we show a parallel replacement for the *subdivision* stage, originally formulated as in Fig. 5.6. By applying this replacement, the listing in Fig. 5.5 executes all of its stages in a parallel fashion.

The algorithm in Fig. 5.28 is only *sketched out* here, although it is fully implemented in our prototypal code. We chose to omit details in order to make the algorithm listing clear in its main ideas.

The idea is that of allowing the parallel execution of the subdivision phase; that is, the routines counting nonzeroes in quadrants, and the copy of index arrays. Coordination among threads is obtained by the use of *shared variables* (line 3 and surrounding ones). The variables occurring within the **begin parallel/end parallel** constructs are meant to be local to each thread. Access to shared variables is arbitrated by appropriate **critical section**s; for instance, at line 8, only one thread at a time is allowed to access the current matrix tree and obtain an "open" leaf submatrix to work on. Submatrices are marked either "open" or "closed"; "open" ones are candidates for further subdivision, "closed" ones are not modified anymore. Right after being chosen for possible subdivision, a submatrix s is checked against a cutoff function δ_p (see Fig. 5.29) if subdivision is desirable. In δ_p, a small constant ($\gamma \approx 1$) is used as a multiplier in a conditional expression allowing subdivision on a nonzeroes-per-thread basis, thus overriding the already presented δ_r cutoff function.

Figure 5.29: $\delta_p(m, k, nnz, CS, ES, WS, s_A)$.

```
if $nnz > \gamma(s_A.nnz/N_t)$ then
    return True
else
    return $\delta_r(m, k, n, CS, ES, WS)$
end
```

Besides using δ_p, we also check some counters for not exceeding a maximum desired number of submatrices N_s, at line 16. If the submatrix is a candidate for subdivision, a further check is performed: this time, using information which was not available before a submatrix indices arrays scan. At line 19, input arrays are scanned and nonzeroes counts are determined for each quadrant. With this information, we may identify very unbalanced submatrices, like those having one populated quadrant and almost empty remaining quadrants. Detecting such cases would be desirable, in order to inhibit further subdivision, especially if the count of *submatrices so far* subdivided is enough to balance the workload among threads. In the case we are confident an appropriate balance is reached, we proceed with updating a shared counter variable: we mark that some (N_n) new submatrices are being allocated,in a critical section area. Right outside the critical section area, we perform the heaviest operations; that is, (line 27) the copy of index data arrays into the new submatrices (recall Fig. 5.6), and initialization of submatrices quadrants[5]. To this end, the information computed at lines 18-20 is combined with that of the s submatrix. After this operation, we enter again into the critical section, update the N_i counter, mark that there are N_i *incomplete* "open" matrices being processed (see line 29). The N_i counter is essential: without it, almost all of the threads would exit the outer cycle in the first loop. When entering, only one submatrix could be subdivided, and there would be no other submatrices to work on.

Whenever a submatrix s is not considered as a candidate for subdivision (see line 33), it gets marked as closed, and information about a candidate format (between *COO* and *CSR*) is attached to it. Update of its arrays according to the chosen format will be performed in the *shuffle* phase, following subdivision—see line 6 in Fig. 5.5.

The parallel subdivision procedure finishes, and a submatrices tree is complete, with each leaf having attached information about its nonzeroes count, offset, dimensions, (and index arrays, in the case of *CSR* leaves), and ready for the *shuffle* operations, as explained in the previous sections.

[5]To simplify readability, we have chosen to initialize variables Q_{nnz}, N_n at lines 18-20, even if their lexical scope is extended to the next conditional construct.

Figure 5.30: $Subrow\_Split\_Search\_Only(s, P, IA, JA)$.

```
$n_{00} \leftarrow 0$; $n_{01} \leftarrow 0$; $n_{10} \leftarrow 0$; $n_{11} \leftarrow 0$;
for $i \leftarrow 0$ $to$ $\lfloor (s.m+1)/2 \rfloor$ do
    $lp \leftarrow Search(JA, P[i], P[i+1], s.koff)$
    $mp \leftarrow Search(JA, P[i], P[i+1], s.koff + \lceil s.k/2 \rceil)$
    $rp \leftarrow Search(JA, P[i], P[i+1], s.koff + s.k)$
    $n_{00} \leftarrow n_{00} + (mp - lp)$; $n_{01} \leftarrow n_{01} + (rp - mp)$
end
for $i \leftarrow \lceil (s.m+1)/2 \rceil$ $to$ $s.m - 1$ do
    $lp \leftarrow Search(JA, P[i], P[i+1], s.koff)$
    $mp \leftarrow Search(JA, P[i], P[i+1], s.koff + \lceil s.k/2 \rceil)$
    $rp \leftarrow Search(JA, P[i], P[i+1], s.koff + s.k)$
    $n_{10} \leftarrow n_{10} + (mp - lp)$; $n_{11} \leftarrow n_{11} + (rp - mp)$
end
return $[n_{00}, n_{01}, n_{10}, n_{11}]$
```

6 Conclusions and Future Work

6.1 Conclusions

In this dissertation, we have guided the reader throughout our considerations about the traditional, well-known layouts and algorithms for sparse matrix computations (in the introductory chapter §1); through recent developments in *hierarchical* sparse matrix formats (in §2,§2.1,§2.2); and then towards the development of a new, hybrid memory layout for the representation of sparse matrices we have named *RSB—Recursive Sparse Blocks*, beginning with §2.3. The central idea of this work has been the use of recursive subdivision of a matrix in quadrants (half the dimension of each submatrix), ending with sparse *cache blocked* leaf submatrices whose data structure depends on the enclosed nonzeroes pattern, on a submatrix basis.

Due to the relevant number of factors playing in opposite directions (e.g.: coarse grained partitioning vs scalability), as well as hard to foresee parameters (e.g.: performance in updating a vector with an irregular access pattern), trying to optimize the code and algorithms by means of some theoretical performance model would have been very difficult, if not impossible to achieve. For this reason, our research has been guided by the feedback from subsequent experiments. The course of incremental development of our techniques (and key findings) is summarized here.

- With the development of the recursively quad-partitioned *CSR* format (*RCSR*) (in §2.3 — see [MFT+10]), we achieved a serial/dual threaded performance (see §2.4) which is comparable or better than that of a scalable research format prototype (*CSB*). The advantage of our approach, though, has been the chance for reuse of well known *CSR* algorithms and techniques for other operations (say, random access ones; see §2.3.5).

- We have developed a non-deterministic shared memory parallel algorithm for the general/symmetric *RCSR* sparse matrix-vector multiplication operation (§3.1 — see [MFPT10b]). We ran experiments on three different machines using up to 8 hardware threads; again, comparing to the *CSB* prototype. Generally, we have found the performance of our multithreaded *SpMV* for *RCSR* superior to *CSB*'s with fewer threads, but often scaling up poorly (performing worse than *CSB*). The main reason for this being often the excessive indices usage; insufficient partitioning (leading to a lack of parallelism) in other cases. We have improved this in §4. We also ran experiments on matrices stored as symmetric (not supported by *CSB*, so no comparison possible here), achieving as much as a 5x speedup.

- We have developed a non-deterministic shared memory parallel algorithm for the *RCSR* sparse triangular solve operation (§3.2 — see [MFPT10b]). Here, we achieved a parallel speedup up to 3x, using the same data structure we use for *SpMV*, in our effort of proposing an *unified* approach. For this operation, however, data dependencies are very constraining, and not every matrix may have a nonzeroes pattern allowing an effective parallel speedup.

- We have tuned *RCSR* (into *RCSRH*) by introducing *short indices* (§4.1 — see [MFPT10c]), with the intent of reducing the *memory footprint* of *SpMV* (§4.2). In this way, we have improved both scalability and performance of *SpMV* in *RCSR* (§3.1), up to 8 threads. *RCSRH* has been found to be comparable or better than the *CSB* prototype on unsymmetric matrices (§4.2.3,§4.2.1), while still supporting symmetric storage and *SpSV* operations (§4.2.2). In some cases though, we noticed an additional index usage, and related inefficiencies.

- We have tuned the *RCSRH* format further (naming the resulting, hybrid format *Recursive Sparse Blocks* (*RSB*)), still keeping a great degree of generality, by further diversification of leaf submatrices and introducing *COO* leaves (§4.3 — see [MFG+10]). In this way, we improved some inefficiency encountered in §4.1, and confirming the strong link between indices storage saving and performance improvements for both unsymmetric (§4.4.1) and symmetric (§4.4.2) cases. When comparing (unsymmetric) results to

CSB (§4.4.3), we noticed that *RSB* results are much closer to *CSB* than before; we see this as being an effect of the increased similarities between the two formats.

- We have devoted a whole chapter (§5 — see [MFPT10a]) to an important problem — that of *building* efficiently an *RSB* matrix in memory, giving a partially parallel/scalable algorithm for this purpose. Here, we establish experimentally the build-to-*SpMV* time ratios for 1 to 8 threads for a number of matrices. This knowledge is essential when deciding about the adoption of *RSB* in a given application. We were not aware of any other freely available shared memory code with parallel build routines, so we did not have means to perform comparative benchmarking. In this work, we also found that the matrix build process is even more memory bandwidth constrained than *SpMV* is, and thus the effort of further study of improving our techniques may be useful. In the chapter closing (§5.6), we suggested an enhanced build algorithm offering even more parallelism. This new algorithm would have slightly changed the definition of *RSB* into being *non deterministic*, so we have chosen to study it further in the future as an interesting development.

- In §C, we have presented experiments of our *RSB* prototype when running *SpMV* and *SpMV-T*, and comparing results to that of the highly optimized *CSR* routines present in the Intel's **MKL** library. We found our approach superior on most large matrices, especially on unsymmetric ones, in both parallel and serial runs.

- Finally, §C.4 has shown comparative results for **MKL**'s *CSR* against *RSB* in the *SpSV* operation, on large triangular matrices. Here we noticed that *RSB* has advantages over *CSR*, in the serial case. As we have seen in §3.2, *RSB* is capable of obtaining a moderate speedup with its parallel *SpSV* algorithm, but we did not explore this option here.

With the closing experiments in (§C), we have confirmed that *RSB* is superior to *CSR* for *SpMV*/*SpMV-T* computations on large matrices, and we recognize that it is ready to host further, processor specific optimization (we did not adopt any such optimization in our code).

In addition, our *RSB* implementation performance was found to be comparable to that of the scalable *CSB* prototype (along with the reduced *SpMV*/*SpMV-T* performance gap), while still retaining many advantages of *COO*/*CSR* (e.g.:

symmetric storage, random access—§2.3.5)[1].

A result of our research is a prototypal software library, which will be released soon. This library is also being integrated as a *shared memory parallel* component in the existing **PSBLAS** library for distributed memory parallel sparse linear algebra computations. We will publish technical information about our library separately.

We think that many enhancements are possible for our techniques; we have summarized most of them in §6.2, §6.3. All of these improvements are topics for possible future research.

Additional interesting topics for future research, which we would like to point out, are: the error analysis of our *SpMV* and *SpSV* algorithms, especially in the light of their non-deterministic formulation; the optimization of our techniques in view of an energy-aware performance metric (as motivated by many current studies; see [Com11, Ch. 3] for a broad technological overview of the problem).

6.2 Minor Enhancements to RSB

This section gives an overview of modifications to the *RSB* format/subroutines bringing enhancements in specific operation areas. These modifications would require little work to be implemented, and have a minor impact on the many *service* routines used for handling matrices.

- **Kernels with multiple right-hand sides.** Variants of *SpMV* or *SpSV* using more than one *right-hand side* vector at a time (that is, multiplying (*SpMM*) or solving (*SpSM*) by a dense matrix) with *specialized low-level kernels* have been reported (for instance, see Im [Im00, Ch. 5]) as being faster than doing the same, one (dense matrix) vector at a time using *SpMV*/*SpSV* operations. The reason for this improvement is the reuse of the sparse matrix vectors arrays: here, they are reused as many times as the *width* of the right-hand side dense matrix. In order to be useful, such computational kernels should be also coherent with the computation at hand. Indeed, they are seldom used in the iterative solution of linear systems; they find more applications in Block-Arnoldi methods for eigenvalue problems (see [Saa03, Ch. 6.3]), for instance. An additional consideration should be made about the number of right-hand sides, here: typically, the prefetch engines of a CPU are capable of *detecting* a fixed number of

[1]In [BWOD11], Buluç et al. extend the *CSB* format for computing symmetric *SpMV* as well.

streams (see [Int08a, § 7.2] for some details on Intel microprocessors); if too many (different) arrays are accessed in turn, the prefetch hardware may have problems in recognizing even simple linear patterns.

- **Convert *COO* leaves to Z-order.** It is known that Z-sorted *COO* has favourable properties when multiplying *sparse blocks* against a dense vector (see §2.2). Z-sorting *COO* leaves of *RSB* may favour cache reuse more than *COR*, on certain matrices; its use does not require the modification of *COO SpMV* routines (see Fig. 1.5).

- **A combined *SpMV/SpMV-T* kernel.** Some iterative methods require both *SpMV* and *SpMV-T* operations at each cycle (the *Biconjugated Gradient*, for instance—see Barrett et al. [BBC+94, Fig.2.7]), on independent vectors (both result and multiplicand). This gives us the chance to formulate a specific kernel listing, which would compute both, with a single matrix visit. However, some modification should be made to the outer *SpMV* algorithm; the required *row locking* strategy should control both the access to a certain rows interval to one results vector, and another one (the transposed) on the second. The *SpMV* listing in Fig. 3.2, and related lock primitives should be modified for this. The lock overhead of such modification would be, however, no more than that of symmetric *SpMV*. Using the symmetric *SpMV* locking variant (as described in §3.1) for this kernel would allow for no further modifications on the outer *RSB SpMV machinery*.

- **Convert *CSR* leaves to *Zig-Zag CSR*.** *Zig-Zag CSR* (see §1.2.4) is a simple variation of *CSR*, which is very likely to give some locality improvement with the existing *CSR* code for *SpMV*. However, since it requires the reversal of each second row, it breaks the *ascending ordered* assumption many *CSR* service routines rely upon. For this reason, a proper handling of *Zig-Zag CSR* requires some more changes in the entire code base.

  Of course, the idea of reversing even rows could be readily extended to the row-major ordered *COO* variant (*COR*, see §1.1), as well as reversing even columns could be applied to *COC*/*CSR*.

- **The use of temporary vectors in *SpMV*.** Some of the matrices we have seen (for instance matrix *diego-MM-573x230k*; see Fig. 4.25) may be very unbalanced, in the proportion of nonzeroes per row. Namely, the average number of nonzeroes per row could be very low for some interval of consecutive rows, and very high for some other consecutive rows interval.

In these cases (see §4.4.4), the row-based lock in our regular *SpMV* algorithm (recall Fig. 3.2) is not efficient, as the unbalance in the nonzeroes distribution may lead to the quick "exhaustion" of the blocks placed on the sparser rows, thus forcing most threads into contention for the blocks on the most "populated" rows. A possible fixup, here, would be that of detecting the unbalance either statically (by computing this *unbalance* information at assembly time), or dynamically (by computing a row blocks contention statistic during *SpMV*), and react by allocating a number of additional vectors for the accumulation of *SpMV* results. However it is unclear to what extent the usage of such temporary vectors (or *subvectors*, if done on an interval basis) would be beneficial, as both the temporary work areas to-zero initialization, and their subsequent *reduction* by summation could be costly, in the presence of many vectors.

- **Reproducibility of *SpMV*/*SpSV* results.** Algorithms we have presented in Fig. 3.2 and Fig. 3.3 are non deterministic, in that the order of execution of operations on the individual submatrices (sparse blocks) depends on the actual scheduling of threads. Since the way floating point numbers are represented on computers does not ensure neither distributivity of multiplication over addition, nor associativity of the two, different orders of execution of operations on the individual submatrices may lead to differing results. In some contexts, the reproducibility of results may be necessary.

  In *RSB*, we could obtain reproducible computation results by forcing a particular order of visit to the submatrices. This feature would be trivial to design in the case of a serial execution, but it would be quite challenging if it is going to be implemented with an *efficient* thread parallelism.

- **Finer partitioning for *SpSV*.** We recognize that the *SpSV* algorithm as presented in §3.2 has an inherent poor parallelism due to the data dependencies among submatrices. We observe that a finer subdivision would permit a higher degree of parallelism, because more threads would be allowed in the *parallel region* at once.

- **Fine grained control over recursive subdivision, after build.** We speculate that a functionality for targeted, parameter-based *further subdivision* (that it, a stand-alone formulation of §5.11) or *subdivisions rollback* of matrix leaves would give the user a cheap way for tuning the data structure of a particular matrix on the fly, and verify the performance of the tuned data structure on the operation of choice (e.g.:*SpMV* or *SpSV*).

Such a functionality would serve for a limited-scope empirical optimization purpose. A more elaborated framework would automate this empirical verification of performance and eventually integrate it during the matrix build process. See Whaley et al. [WPD01] for the topic of *Automated Empirical Optimization* and Vuduc et al. [VDY05a] for an application to sparse matrix structures in the context of a subroutines library.

6.3 Major Enhancements to RSB

In this section, we present a number of possible, quite relevant modifications to the *RSB* format (and routines). These modifications require either the change of substantial concepts of the *RSB* format as it was presented in this thesis, or a relevant amount of work, so we group them separately from the minor changes presented in the previous section.

- **Separate leftover matrices for grouping tiny leaves.** As we have seen, the current matrix assembly algorithm (see §5), although using some heuristics guaranteeing balancing the nonzeroes among quadrants (for instance, choosing the *biggest leaf* before each subdivision), there is still a conflicting, but necessary constrain: that mandating a minimal number of leaf submatrices overall, for load balancing during computations. In some circumstances, quadrants with very few elements could occur. Think of a very big matrix (some orders of magnitude bigger than outermost cache size), with all quadrants quite populated except one, which hosts only few elements. It is clear that the best thing to do here is to subdivide the matrix: no parallel operations would be possible at all without subdivisions. In this situation, then, that quadrant with few elements will be instantiated and used during the parallel *SpMV* and other operations. However, during *SpMV*, a lock will have to be acquired, in order to operate on this tiny submatrix (see listing Fig. 3.2). After that one or more subroutines will have to be entered, and various branch and loop instructions will occur before the few operations on that submatrix could be executed. Handling such inefficiencies would be desirable, especially if *many* such leaves occur. If the overall number of such submatrices is high, one could resort into regrouping them in one or more *leftover matrices*, and performing *SpMV* on it/them *after* the main, parallel *RSB-SpMV*. Since there exists no exact technique to estimate the number of such occurrences, we leave such an empirical study as a possible future development. Note that decomposing a matrix in this way would be somehow in analogy to the *variable-block* *splitting* technique used by Vuduc, in the context of *BCSR*–see [Vud03, Ch. 5]. Here, however, the application context is quite different.

- **Use decision trees for more variants.** One could bring further the considerations discussed when presenting Table 1.1, possibly having even more code variants to chose from, for the same formats in §1. This could be coupled with a mechanism active at assembly time, making some further consideration in the choice of *leaves* submatrices format, in Fig. 5.6. We observe that here, we need formats optimized for *submatrices*; that is, matrices having properties differing from *full matrices*. For instance, the assumption of having no empty rows is usually appropriate for a full matrix; it is not on an arbitrary submatrix. In such a context, the development of very particular formats (for instance, a *list of sparse rows*-based representation), would be appropriate.

- **Static submatrix-thread mapping**. It would be desirable to rearrange the *SpMV* operation in a way to limit the needed coordination among threads, and still writing the result to disjoint output array intervals. This would be achievable, if using some static mapping technique; i.e.: defining *when* the computation on *which* submatrix should occur, *before* entering *SpMV*; possibly using some partial ordering technique. This would need some metric for the estimation of performance in the individual leaves, in the frame of some performance model which would assure balancing. If coupled with the use of *thread local storage*[2], for *groups of leaf submatrices*, this mapping could give some performance benefits, especially with the very high number of cores that should be expected in the future[3]. On the other hand, a *tree-level* locking strategy, arbitrating both locality of access and the row locking, would be a viable solution, regardless of the storage. As we see, such modification may imply many modifications to the existing system. Note that the choice of using OpenMP for our research was motivated mainly by our need for portability and focus on algorithms; it was a deliberately system-agnostic choice.

- **More formats for leaf submatrices.** Specialization of leaves could bring performance benefits; think of using *BCSR* with some specific $br_n \times bc_n$ blocking giving low *fill-in*, specific to each leaf submatrix s_n (see §1.2.4). However, a full integration of *BCSR* may be difficult. Even in the case of

[2]That is, having the *COO*/*CSR* submatrices as a number of disjoint arrays, each one allocated in a physically different memory area/bank for each different thread.

[3]Increased non-uniformity of memory access is to be expected, too.

a uniform $br \times bc$-blocking choice, we observe an incompatibility with the current definition of *RSB*, now requiring an $m \times k$ matrix to be split in four quadrants: the first upper left one being sized $\lceil m/2 \rceil \times \lceil k/2 \rceil$; the remaining quadrants consequently. Since it may happen to have $\lceil m/2 \rceil$ not divided by br or $\lceil k/2 \rceil$ not divided by bc, one should either decide whether to change the *RSB* definition to accommodate matrices' subdivision congruently to some given blocking, or rather to handle the *BCSR* leaves in some custom way[4], in a way to avoid *off-the-boundary* vector read and write operations caused by the eventual extra rows/columns.

- **Extra clearance between submatrices.** Currently, the submatrices' subarrays of *RSB* fit into the original *COO* arrays (see §5). There are situations (i.e.: sparse sums, pattern modifications, data structure change) where supporting some *clearance* between the arrays of each submatrix would allow for some in-place reorganization at the matrix level. Of course, the assembly algorithms seen in §5 would need to be modified for this.

- **Use of code generators for processor-specific optimizations.** We mentioned the use of code generators for producing the variants of our computational kernels. Some system-specific optimization is easy to perform in this context. For instance, the use of software prefetch instructions could bring benefits. For instance, Intel's `prefetchnta` instruction[5]; see manuals ([AMD07, § 3.9.6],[Int08a, Ch. 7.4]) informs the CPU about the *non-temporal allocation* of some (specified) cache line. That is, the CPU is requested not to cache a specified cache line after its access; instead, that cache line is written back to memory *before* explicit eviction occurs. This could be of use in the many cases arrays are read or write once, without reuse (recall Table 1.1). There could be many other optimizations based on some on specific *C compiler `pragma`*, *compiler intrinsic*, *code annotation*, or *assembly instruction*. The use of code generation technology may ease the application of optimizations of this kind.

- **Integration with GPU techniques.** The investigations presented in this thesis deal with techniques for commonly available *shared memory parallel, cache based* CPUs. However, the recent enhancements in GPU

[4]For instance storing the last $mod(m, br)$ rows as plain *CSR*. Of course, the last $mod(k, bc)$ columns should be dealt with also, in some way, and so on for the remaining quadrants, recursively.

[5]Present in the **SSE** ("Streaming SIMD Extensions") extension of Intel architecture's instruction set ([Int08b, § 5.5,Ch. 10-12]).

based technology (numerical precision, software tools for debugging, and the potential for a high floating point performance) make GPUs increasingly attractive for sparse matrix computations. In this context, it would be interesting considering an approach with a number of *RSB* leaf submatrices to be *offloaded* to a GPU unit during *SpMV* computation (if necessary, storing the leaf submatrices of interest in a GPU-specific format). Such *hybrid* CPU-GPU software solutions are increasingly popular; see Papadrakakis et al. [PSK11] for an example application.

A Appendix: experimental setup

This appendix chapter keeps track of the setup of most experiments carried out during the making of this thesis. Each of the following sections details information about a particular experiment, and gives reference of the section where results are commented. Sections are independent and self-contained.

A.1 Setup for §2.4 Experiments

machine	**model**	**cpus/ cores**	**data caches**
M7	AMD Opteron 246 1.0GHz	2/1	2xL1,2xL2: L2:1M/16-w/64B L1:64KB/2-w/64B
M5	AMD Athlon 64 X2 Processor 6000 3.0GHz	1/2	2xL2: L2:1MB/16-w/64B L1:64KB/2-w/64B
M2	AMD Opteron 2354 Quad-Core 2.2GHz	2/4	2xL3,2x4xL2,2x4xL1: L3:2MB/32-w/64B L2:512KB/16-w/64B L1:64KB/2-w/64B

Table A.1: Test machines for §2.4 experiments.

For space reasons we report results obtained on a limited experimental setup. In our experiments, we run *SpMV* (defined as $y \leftarrow y + Ax$) on the (non-symmetric) matrices reported in Table A.4. They originate from the *Univer-*

machine name	compiler
M5	`gcc version 4.1.2`
M7,**M2**	`gcc version 4.3.2`
all	`gcc version 4.2.4`
	`(Cilk Arts build 8503)`

Table A.2: Compilers on test machines for §2.4 experiments.

implementation	compilation flags
CSR/CSC/RCSR/RCSC (C99)	`-O3 -fopenmp -std=c99`
CSB*/CSC* (**CILK++**)	`-O3 -fno-rtti -fno-exceptions`

Table A.3: Relevant (non-warnings) compiler flags used for §2.4 experiments.

sity of Florida Sparse Matrix Collection [Dav10]. Our (*RCSR*/*RCSC*) *SpMV* kernel implementations have been run with and without multicore parallelism, and are compared against the *CSB* (see §2.2) prototype code released by Buluç and authors of [BFF$^+$09]. We have chosen to benchmark against *CSB* because, just like *RCSR*/*RCSC*, it was conceived to be used in a multicore context. The *CSB* format stores Z-sorted elements in *sparse blocks* of 2^k size, whereas the *RCSR*/*RCSC* stores Z^b-sorted *submatrices* of arbitrary size; we find this duality interesting for comparison purposes. The *CSB* code is parallelized with the **CILK**++ system, which extends the C++ language and requires applications to be compiled by its special compiler. Then, the executable program file is linked to the **CILK**++ *runtime load balancer.* The codes were run on the 64 bit machines shown in Table A.2; the used compiler versions are in Table A.2; compilation flags in Table A.3. We chose not to use machine-specific optimization flags because of slight incompatibilities between the **CILK**++ compiler and compilers available in the Fedora Linux distributions installed on our machines. Both codes use `double` as the numerical type, 32 bit integer indices, and 64 bit pointers. With each experiment, we also report the measured performance of *CSR*/*CSC*(our implementation) and the *CSC* implementation of Buluç et al. [BFF$^+$09] (in the plots we mark the measurements of their code with an asterisk, as in *CSC** and *CSB**). We have modified the timing function of the *CSB* code to use a `double` (instead of an `int`) variable, to limit precision loss (milliseconds are measured). Both codes use now the `gettimeofday` POSIX function for timing. Performance is expressed in *Millions of FLoating Point Operations per Second (MFLOPS)* As conventional for the *SpMV*, we count two floating point operations for each matrix nonzero. We perform 100 *SpMV* kernel runs for each sample and report the best value (we have observed that in all cases best value differs from the average by no more than 2%). Note that the actual *CSB* prototype code leaves apart portions of matrices, and thus taking into account this *leftover* in the computation would likely lead to somewhat differing results. We should also note that during benchmarks, the **M2** machine was also (lightly) loaded as a web server, and this could have adversely affected our measurements.

matrix	**rows**	**columns**	**nonzeros**	**n.z./r.**	**n.z./c.**
ASIC_320k	321821	321821	2635364	8.19	8.19
Rucci1	1977885	109900	7791168	3.94	70.893
cont11_l	1468599	1961394	5382999	3.67	2.74
neos	479119	515905	1526794	3.19	2.96
rail4284	4284	1096894	11284032	2633.99	10.287
rajat31	4690002	4690002	20316253	4.33	4.33
sls	1748122	62729	6804304	3.89	108.47
sme3Dc	42930	42930	3148656	73.34	73.34
spal_004	10203	321696	46168124	4524.96	143.51
stomach	213360	213360	3021648	14.16	14.16
torso1	116158	116158	8516500	73.32	73.32

Table A.4: Test matrices for §2.4 experiments.

A.2 Setup for §3 Experiments

To illustrate the efficiency of the proposed approach, we report performance results for the *SpMV* and the *SpSV* operations, respectively defined as $y \leftarrow Ax$ and $x \leftarrow L^{-1}x$. As representative samples for the *SpSV* we have selected the lower triangles (L matrices) originating from the *LU* factorizations of a mix of the matrices used by authors of [May09] and [VKH+02]. These matrices are publicly available at the *University of Florida sparse matrix collection* (see, [Dav10]). Their *LU* factorizations were computed using SuperLU 3.1 [DEG+99] after reordering with **COLAMD** [DGLN04], called indirectly by **GNU Octave** 3.0.3 [BA06]. To test the performance of the *SpMV*, we used (1) the same L matrices as for the *SpSV*, (2) (unsymmetric) matrices that have been utilized in experiments reported in [MFT+10] (or §A.1), but run on *different* machines and/or with larger number of cores, and (3) nine symmetric matrices. In the following we report data for the most significant subset of cases. We have also utilized the *CSB* prototype code (see §2.2 or [BFF+09]), but applied it only to the group of unsymmetric, non-triangular matrices, as this code does not support symmetric ones. The performance measurements we report are the best ones, after performing 100 runs (for either code). However, we found no big variability in the performance of the runs, which indicates that even though the proposed algorithms are non-deterministic, their performance is stable.

We canonically count two floating point operations for each nonzero for the *SpMV* for general matrices or the *SpSV*, and four floating point operations for each nonzero for the *SpMV* for the symmetric matrices. We employ `double` as our floating point type, 64 bit pointers, and 32 bit integer indices.

Table A.2 contains informations about the CPUs of the machines on which we ran our experiments, and the C compilers we used for our code. We used the `-O3 -q64 -bmaxdata:0x1000000000 -qarch=pwr5 -qtune=pwr5 -qsmp=omp -qlanglvl=extc99 -qkeyword=restrict` compilation flags on **M1**, and `-O3 -fopenmp -std=c99` on the remaining machines. We have compiled the *CSB* code using the required specialized **CILK++** compiler, based on 64 bit GCC-4.2.4, *build 8503*. We modified this code according to the guidelines found in [BFF+09] to correctly provide it with machine cache parameters (the $L2$ cache size and the cache line length). We were not able to collect results of the *CSB* code on the **M1** machine, as its architecture is not supported by **CILK++**. Note that the **M2** machine is a lightly loaded network server, so its results may include some noise.

We are aware of high level approaches to multi-threading like Intel's Thread Building Blocks ([Int10]) or the aforementioned **CILK++** ([Int09]), but these approaches would force us to use C++ and restrict ourselves to the Intel architecture. So we have chosen to implement our algorithms in C, using OpenMP for the parallelization, for reasons of availability, standardization (C99 standard), and compatibility of such approach.

A.3 Setup for §4.1 Experiments

We collected performance data on two machines and 36 matrices. For each sample, we performed 100 runs of the *SpMV* operation on the matrix A, the right-hand side vector x, and the result vector y, defined as $y \leftarrow y + A \cdot x$. Among these 100 runs, we report the best result, however the variation between the best and the worst result stayed always below 5%, showing that our approach is performance-stable. The reported performance is measured in MFlops (millions of floating point operations per second). We measured timings using

matrix	rows	columns	nnz	n./r.
unsymmetric				
Rucci1	1977885	109900	7791168	4
rajat31	4690002	4690002	20316253	4
sme3Dc	42930	42930	3148656	73
torso1	116158	116158	8516500	73
symmetric				
BenElechi1	245874	245874	6698185	27
F1	343791	343791	13590452	40
Ga41As41H72	268096	268096	9378286	35
af_0_k101	503625	503625	9027150	18
af_shell10	1508065	1508065	27090195	18
bone010	986703	986703	36326514	37
boneS10	914898	914898	28191660	31
kkt_power	2063494	2063494	8130343	4
ldoor	952203	952203	23737339	25
lower triangles				
FEM_3D.. 's L	147900	147900	107067049	724
av41092's L	41092	41092	18963133	461
g7jac180's L	53370	53370	14561594	273
g7jac200's L	59310	59310	18181493	307
ohne2's L	181343	181343	322813873	1780
poisson3Db's L	85623	85623	101532912	1186
sme3Dc's L	42930	42930	20871702	486
venkat50's L	62424	62424	10412687	167
torso1's L	116158	116158	28372106	244

Table A.5: Matrices for §3 experiments. "n./r." means "nnz/rows".

the POSIX ([pos08]) `gettimeofday()` function. Conventionally, we counted 2 Flops per nonzero element for non-symmetric machines, and 4 for symmetric. We used double precision arithmetic (C's `double` type). Our measurements were performed with *hot caches*, that is, we deliberately do not flush cache contents after each *SpMV*. To avoid artificially high results, we restricted our measurements to matrices not fitting entirely in the caches.

We report the matrices used in our test set in Table A.7 and Table A.8. They all originate from the *University of Florida Sparse Matrix Collection* [Dav10], except for *diego-MM-573x230k*, an *information retrieval* document-term matrix, which was obtained by the courtesy of Diego De Cao from the Tor Vergata University, Italy. Out of 36 matrices, 12 are symmetric, while among the unsymmetric ones, 12 are square. They cover a broad range of applications: *patterns/relations*: *12month*; *graphs*: *cage15, patents, wb-edu*; *linear programming*: *cont11_l, neos, rail2586, spal_004, tp-6*; *fluid dynamics*: *atmosmodl, raefsky3, rma10, venkat01*; *2/3D problems*: *av41092, torso1, BenElechi1, nd24*; *combinatorial problems*: *c8_mat11_I, GL7d19, rel9, relat9*; *chem-*

machine	model	cpus/cores	data caches	compiler
M1	IBM POWER 5 (91188-575)		1xL3:36MB (off-chip,not considered)	xlc 7.0
		16/1	L2:1.92MB/10-w/128B	(AIX 5)
	1.5 GHz		L1:(2x)32KB/4-w/128B	
M2	AMD Opteron 2354	2/4	2xL3,2x4xL2,2x4xL1:	gcc 4.3.2
	Quad-Core		L3:2MB/32-w/64B	(Red Hat)
	2.2GHz		L2:512KB/16-w/64B	
			L1:64KB/2-w/64B	
M3	Intel Xeon E5405	2/4	2xL2,2x4xL1:	gcc 4.3.2
	Quad-Core		L2:6MB/8-w/64B	(Red Hat)
	2.0GHz		L1:32KB/24-w/64B	

Table A.6: Summary of test environments for §3 experiments.

ical simulations: *lhr71*; *circuit simulation*: *rajat31*; *least squares*: *Rucci1*; *structural problems*: *sme3Dc, af_shell10, crankseg_1, ct20stif, F1, fcondp2, ldoor, s3dkq4m2*; *optimization*: *kkt_power, mip1*; *model reduction*: *bone010*; *information retrieval*: *diego-MM-573x230k*. Note that in our experiments, we used a much broader test set. Matrices discussed here cover the most representative/particular cases. We conducted our experiments on the two computers summarized in Table A.9. Machine **M2** was a lightly loaded network server, while **M4** was a dedicated machine.

We compiled our codes with the Intel `icc` version 11 on **M4**, and `gcc`, version 4.3 on **M2**.

We compiled it on both machines using the **CILK**++ compiler ([Int09]); version ("Cilk Arts build 8503"), based on `gcc`, v.4.2.4. All codes have been compiled using the `-O3` flag only (besides the OpenMP enabling flags).

A.4 Setup for §4.3 Experiments

In order to compare the new approach with previously documented experiments using *RCSR* format (see [MFPT10c]), we measured performance on the same test set of 36 matrices: 12 of them are symmetric (See Table A.10), 12 are square unsymmetric (See Table A.11), and 12 are non-square (See Table A.12). For readability reasons, in Sec. 4.4 we left matrices with less significant results (marked with an asterisk (*), in the tables) out of the plots, so the commentary on them is indirect. Furthermore, we have used the same two machines (summarized in Table A.13). Recall, that **M2** is a lightly loaded network server, while **M4** is a dedicated machine.

For each *matrix/cores* sample, we ran our *RSB* code, performing 100 times the *SpMV* operation, and we and report the best result. However, timing variation was below 5%, so our results were consistent. We measured timings using the POSIX ([pos08]) `gettimeofday()` function. Figures in section 4.4 depict results, expressed in MFlops (millions of floating point operations per second). Conventionally, we counted 2 Flops per nonzero element for non-symmetric matrices, and 4 for symmetric. We use double precision arithmetic (C's `double` type). Our measurements were performed with *hot caches*; that is, we did not flush deliberately cache contents between subsequent *SpMV*'s; therefore, to avoid artificially high results, all measurements were performed on matrices not fitting entirely in the caches.

Our codes were compiled with the Intel `icc` version 11 on **M4**, and `gcc`, version 4.3 on **M2**. In Section 4.4.3 we compare our results to that obtained with a publicly available *CSB* prototype (see §2.2 or [BFF+09]). On both machines we compiled it using the **CILK++** compiler; version ("`Cilk Arts build 8503`"), based on the `gcc` (*GNU C Compiler*), v.4.2.4. To unify the test environment, all codes were compiled using the `-O3` flag only (besides the OpenMP enabling flags).

A.5 Setup for §5 Experiments

Our experimental setup is similar to that of [MFG+10] (or §A.4) : same machines, same compilers, same methodology, but for space reasons, we selected only an *essential* subset of the matrices used there (see Table A.14). We compiled and ran our codes on machines **M4** (AMD Opteron 2354; 2×4-core CPU; caches: 2×2MB L3, 4× 512KB L2 and 64KB L1) and **M2** (Intel Xeon 5670; 2×6-core CPU; caches: 2×12MB L3, 4× 256KB L2 and 32KB L1), using `-O3` as the only optimization flag, with `icc` v.11 on **M4**, and `gcc` v.4.3 on **M2**. The time samples employed are the *best ones*, after 100 runs for the *SpMV* operation, and 10 runs for the constructor. **M2** is a lightly loaded network server.

A.6 Setup for §B Experiments

In Table A.15, we show some information about machines **M6** and **M8**, which were used for the experiments reported in §B.

matrix	r	c	nnz	nnz/r
12month1	12471	872622	22624727	1814.19
atmosmodl	1489752	1489752	10319760	6.93
av41092	41092	41092	1683902	40.98
c8_mat11_I	4562	5761	2462970	539.89
cage15	5154859	5154859	99199551	19.24
cont11_l	1468599	1961394	5382999	3.67
diego-MM-573x230k	573286	230401	41694697	72.73
GL7d19	1911130	1955309	37322725	19.53
lhr71	70304	70304	1528092	21.74
neos	479119	515905	1526794	3.19
patents	3774768	3774768	14970767	3.97
raefsky3	21200	21200	1488768	70.22
rail2586	2586	923269	8011362	3097.97
rajat31	4690002	4690002	20316253	4.33
rel9	9888048	274669	23667183	2.39
relat9	12360060	549336	38955420	3.15
rma10	46835	46835	2374001	50.69
Rucci1	1977885	109900	7791168	3.94
sme3Dc	42930	42930	3148656	73.34
spal_004	10203	321696	46168124	4524.96
torso1	116158	116158	8516500	73.32
tp-6	142752	1014301	11537419	80.82
venkat01	62424	62424	1717792	27.52
wb-edu	9845725	9845725	57156537	5.81

Table A.7: General matrices for §4.1 experiments.

matrix	r	c	nnz	nnz/r
af_shell10	1508065	1508065	27090195	17.96
BenElechi1	245874	245874	6698185	27.24
bone010	986703	986703	36326514	36.82
crankseg_1	52804	52804	5333507	101.01
ct20stif	52329	52329	1375396	26.28
F1	343791	343791	13590452	39.53
fcondp2	201822	201822	5748069	28.48
kkt_power	2063494	2063494	8130343	3.94
ldoor	952203	952203	23737339	24.93
mip1	66463	66463	5209641	78.38
nd24k	72000	72000	14393817	199.91
s3dkq4m2	90449	90449	2455670	27.15

Table A.8: Symmetric matrices for §4.1 experiments.

machine	**model**	**cpus× cores**	**data caches**
M4	Intel Xeon 5670 6-Core 2.93GHz	2×6	2xL3,2x6xL2,2x6xL1: L3:12MB/16-w/64B L2:256KB/8-w/64B L1:32KB/8-w/64B
M2	AMD Opteron 2354 Quad-Core 2.2GHz	2× 4	2xL3,2x4xL2,2x4xL1: L3:2MB/32-w/64B L2:512KB/16-w/64B L1:64KB/2-w/64B

Table A.9: Test machines for §4.1 experiments.

matrix	r	c	nnz	nnz/r
af_shell10	1508065	1508065	27090195	17.96
BenElechi1	245874	245874	6698185	27.24
bone010	986703	986703	36326514	36.82
crankseg_1	52804	52804	5333507	101.01
ct20stif	52329	52329	1375396	26.28
F1	343791	343791	13590452	39.53
fcondp2	201822	201822	5748069	28.48
kkt_power	2063494	2063494	8130343	3.94
ldoor	952203	952203	23737339	24.93
mip1*	66463	66463	5209641	78.38
nd24k	72000	72000	14393817	199.91
s3dkq4m2	90449	90449	2455670	27.15

Table A.10: Symmetric matrices for §4.3 experiments.

matrix	r	c	nnz	nnz/r
atmosmodl	1489752	1489752	10319760	6.93
av41092	41092	41092	1683902	40.98
cage15	5154859	5154859	99199551	19.24
lhr71	70304	70304	1528092	21.74
patents	3774768	3774768	14970767	3.97
raefsky3	21200	21200	1488768	70.22
rajat31	4690002	4690002	20316253	4.33
rma10*	46835	46835	2374001	50.69
sme3Dc	42930	42930	3148656	73.34
torso1	116158	116158	8516500	73.32
venkat01	62424	62424	1717792	27.52
wb-edu	9845725	9845725	57156537	5.81

Table A.11: General square matrices for §4.3 experiments.

matrix	r	c	nnz	nnz/r
12month1	12471	872622	22624727	1814.19
c8_mat11_I	4562	5761	2462970	539.89
cont11_l	1468599	1961394	5382999	3.67
diego-MM-573x230k	573286	230401	41694697	72.73
GL7d19	1911130	1955309	37322725	19.53
neos*	479119	515905	1526794	3.19
rail2586	2586	923269	8011362	3097.97
rel9	9888048	274669	23667183	2.39
relat9	12360060	549336	38955420	3.15
Rucci1	1977885	109900	7791168	3.94
spal_004	10203	321696	46168124	4524.96
tp-6	142752	1014301	11537419	80.82

Table A.12: General non-square matrices for §4.3 experiments.

machine	**model**	**cpus× cores**	**data caches**
M4	Intel Xeon 5670 6-Core 2.93GHz	2×6	2xL3,2x6xL2,2x6xL1: L3:12MB/16-w/64B L2:256KB/8-w/64B L1:32KB/8-w/64B
M2	AMD Opteron 2354 Quad-Core 2.2GHz	2× 4	2xL3,2x4xL2,2x4xL1: L3:2MB/32-w/64B L2:512KB/16-w/64B L1:64KB/2-w/64B

Table A.13: Test machines for §4.3 experiments.

matrix	symm	r	c	nnz	nnz/r
12month1	G	12471	872622	22624727	1814.19
af_shell10	S	1508065	1508065	27090195	17.96
cage15	G	5154859	5154859	99199551	19.24
cont11_l	G	1468599	1961394	5382999	3.67
fcondp2	S	201822	201822	5748069	28.48
GL7d19	G	1911130	1955309	37322725	19.53
ldoor	S	952203	952203	23737339	24.93
neos	G	479119	515905	1526794	3.19
patents	G	3774768	3774768	14970767	3.97
rail2586	G	2586	923269	8011362	3097.97
relat9	G	12360060	549336	38955420	3.15
sme3Dc	G	42930	42930	3148656	73.34
wb-edu	G	9845725	9845725	57156537	5.81

Table A.14: Matrices test-set for §5 experiments.

machine	**model**	**cpus× cores**	**data caches**
M8	Intel Pentium III (Coppermine) 866MHz	1×1	1x1xL2,1x1xL1: L2:256KB/8-w/32B L1:16KB/4-w/32B
M6	Intel Atom N450 (Pineview) Two-Core 1.66GHz	1× 2	1x1xL2,1x1xL1: L2:512KB/8-w/64B L1:24KB/6-w/64B

Table A.15: Test machines for §B experiments. To use Intel terminology, the number of "hardware" *cores* on the **M6** machine is really only one, with two being the number of *hyper-threads* available to the system.

B
Appendix: patterns of indirect memory access, with stride

In this appendix chapter, we present a basic experiment quantifying the *relative* performance in accessing an array directly or by means of an index vector indirection. In either case, we consider only array read operations. This section gives support to the discussion in §1.4.

The target of our experimentation will be both the cache subsystem and the (automatic) hardware prefetch engine. Except one machine without an automatic prefetch engine, the remaining machines have both. Since neither the hardware prefetch engine, nor the cache subsystem could be *turned off*, we will use some architecture specific limit to bypass their effect. For the hardware prefetch equipped machines, it is documented (see [Int08a, § 2.4.2]) that a (bytes) distance between temporally local accesses exceeding 512, will not trigger an anticipated *fetch* of memory locations into the cache hierarchy. Therefore, it is sufficient to access memory repeatedly with a fixed *stride* value exceeding this distance (called *trigger threshold*—t_t), to disable automatic prefetch.

We perform our experiment on four machines, with either 64-bit ($w_s = 8$)or 32-bit word sizes ($w_s = 4$). In our experimental code, we allocate a fixed number of *words*, say w_t, into a dynamically allocated, appropriately aligned array. We want to scan repeatedly this array, with an increasing stride, from 1 to κ. We want each visit to touch the same number of different locations, say w_n. To arrange this, we set $w_t w_s = \kappa w_n w_s$. In this way, when accessing the array with unitary stride, only the first w_n words will be touched. With maximal stride (κ), the same number of words (w_n) will be touched. This time, however, there will be a gap of κw_s bytes separating each word's address.

We perform three types of visit, for measuring three different access types. For each given visit type and stride value combination, we measure the rate of accessed *words per second*, by repeating a fixed number of times (l_n) the scan over the whole array.[1]

The three visit types are:

- **Linear, direct.** Elements $1, w_n$ are repeatedly accessed, in sequence, for each stride value in $s \in [1, \kappa]$; that is, for stride value s, elements at array offset $sw_s i$, with $i \in [1, w_n]$ are accessed.

- **Linear, indirect.** At each array index i in $1, ..., w_n$, we fill a helper (w_n-sized) array with exactly value i at location i. Then we access the main array at these locations, by using the helper array as an indices array. The memory pattern in accessing the main array, thus, is the same as the linear, direct scan. Overall, however, the helper array is accessed in order to load the index, for each location. And just as in the direct, linear scan, we apply a growing stride.

- **Pseudo-random, indirect.** We compute a *repeatable* pseudo-random sequence of numbers (not a permutation of $1, .., w_n$), and assign this sequence to an indices array. Then, we use the indices array to access the data array repeatedly, at the specified pseudo-random locations, again indirectly. We repeat for each stride value (and with the same pseudo-random sequence). Note that since we do not generate a permutation, multiple visits to the same address are possible.

Producing a valid, general *micro-benchmark* is a tricky task. The one we have specified so far is rather simple, but it serves well our purpose: finding the bottom line for memory access speed. To be sure some compiler optimization would not drive the program into producing fake results, we have compiled (and we ran) the benchmark program in two instances: one with a quite high compiler optimization flag (`-O3`), and another with no optimization at all (`-O0`). As we will see, both program versions will converge to the same speed rates, for all three memory scan types.

[1]When accessing each given location, we sum its contents (an integer sum is an operation so cheap it should not have impact on the benchmark) to an accumulator variable. Then, at the program's end, a hash value of the sum accumulated over all the visits is printed out. This is a trick for preventing the compiler from some optimizations, which could completely remove the loop code from the compiled code.

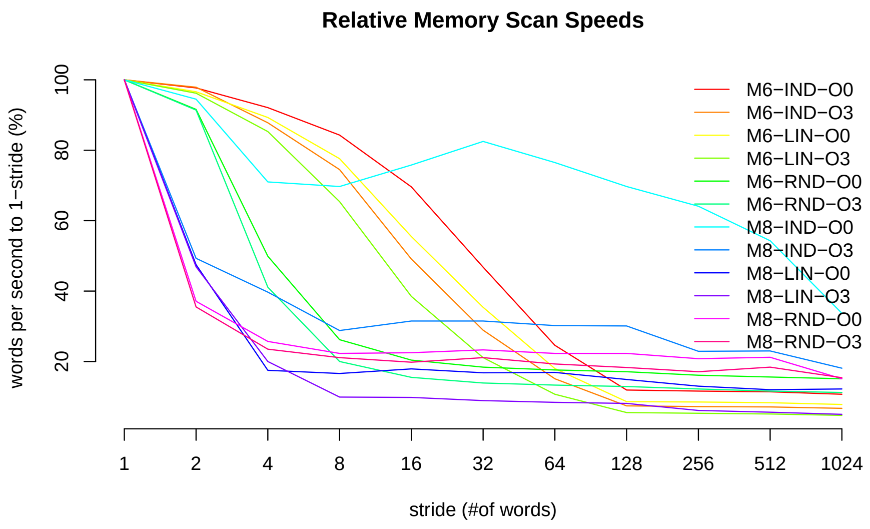

Figure B.1: The relative performance of some linear scan primitives on **M6**,**M8**. We have parameters $w_s = 8, \kappa = 1024, w_n = 128 \cdot 1024$. One of the lines for **M8** looks like it is not monotonically decreasing—indeed, that line represents a scan whose speed is almost constantly at the bottom; the plot only magnifies *noise* present in the measurement. For some specification of machines **M8** and **M6** see Table A.15.

In each one of the presented plots, for each machine we show six curves: three scans performed by the same program, compiled twice; with and without compiler optimizations.

We have two types of plots: the first one, depicting the *words per second* rate each visit type has achieved, for each stride value; and the second one, showing this rate normalized by the value at unitary stride (individually, for each one of the six curves).

In Fig. B.2, we see the normalized rate curves for the two faster machines, **M4** and **M2**. In contrast, see Fig. B.1, where the old **M8** (an Intel Pentium 3—an architecture without an automatic hardware prefetch engine) machine and the recent but weak **M6** were used.

For all machines, we witness a *convergence* of the six curves down to about 20% of the peak. Machine **M8** shows a lesser *relative* loss in performance for some indirect accesses, but the remaining curves for it fall down below 20% of peak speed.

The faster machines loose less (that is, their curves descent slower), in both the indirect linear and indirect random access modes. The explanation of this is simple: indirect accesses were already the slowest ones, because of the dependency on address computation, and thus, the possible speed drop was less. The absolute access rates are shown in Fig. B.3 and Fig. B.4.

The *trigger threshold* for the prefetch-equipped machines was no more than 512 bytes. Since **M4** and **M2** are 64 bit machines, this means that a stride higher than $512/8 = 64$ is enough to prevent the prefetch circuitry from triggering. Indeed, at stride values from 128 on, we do not notice a further speed drop, since latency in memory access is completely exposed.

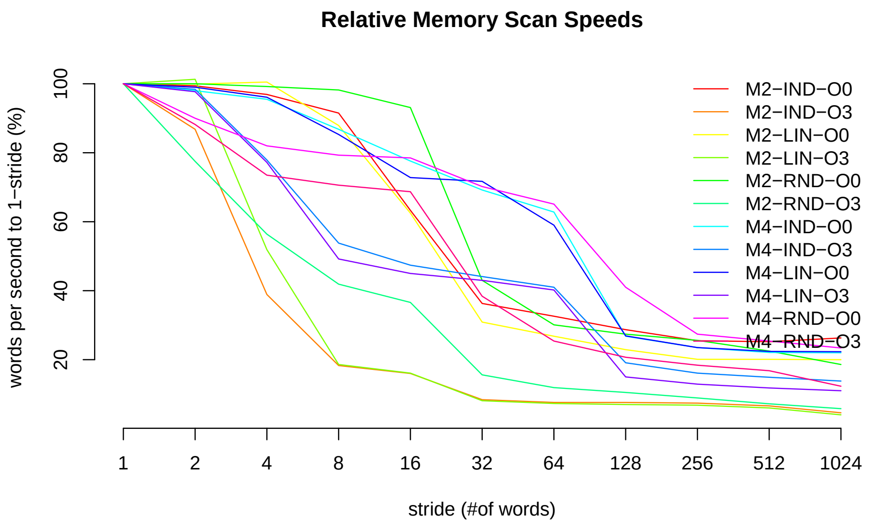

Figure B.2: The relative performance of some linear scan primitives on **M2**,**M4**. We have parameters $w_s = 8, \kappa = 1024, w_n = 128 \cdot 1024$. For some specification of machines **M4** and **M2** see Table A.9.

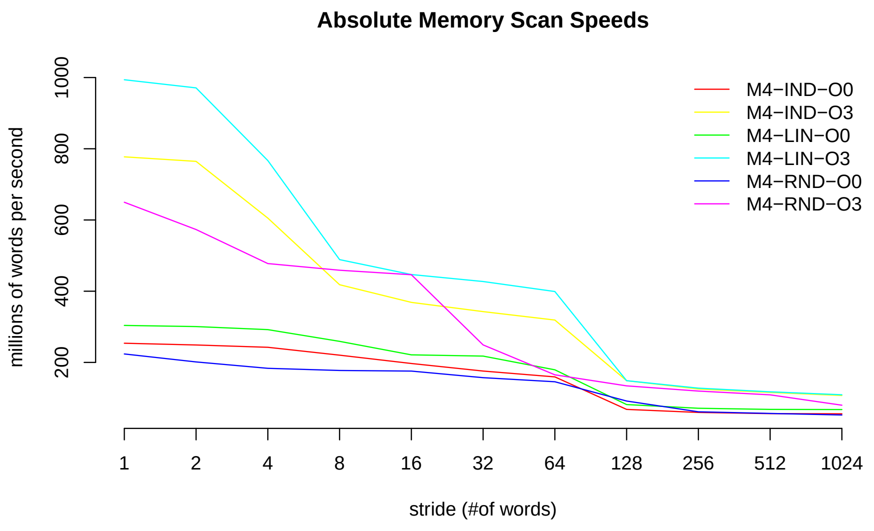

Figure B.3: The absolute performance of some linear scan primitives on **M4**.

In this discussion, we did not mention cache parameters, because our interest was to expose the latency, bypassing both cache usage and hardware prefetch.

However, there are some facts we would like to point out. For stride values up to the number of words in a cache line (for 64 bit, 64/8 = 8 bytes), we have some cache line reuse: for instance, having stride 2, half of each cache line data is reused. When stride is higher than this, but still lower than the trigger threshold value divided by word size, cache is not reused at all: each cache line is used exactly once. Indeed, it is between these values, that we notice the steepest descent on each of the curves shown. We also notice that on the newer machines the curves tend to drop the most on stride values higher than on the weaker or older machines (see Fig. B.1, Fig. B.2).

Notice that one machine (**M8**:see Fig. B.1) does not have an automatic prefetch engine, and for this reason, it reaches its bottom line for memory access speed at stride 8: much before the other machines do; see Fig. B.5.

We omit curves for the absolute scan speeds of machine **M2**, as these are similar to that of **M4**.

From this small experiment we conclude that codes where memory reuse is low and indirect addressing occur—and this is the case for sparse matrices codes—making good use of the prefetch hardware is an important factor to performance.

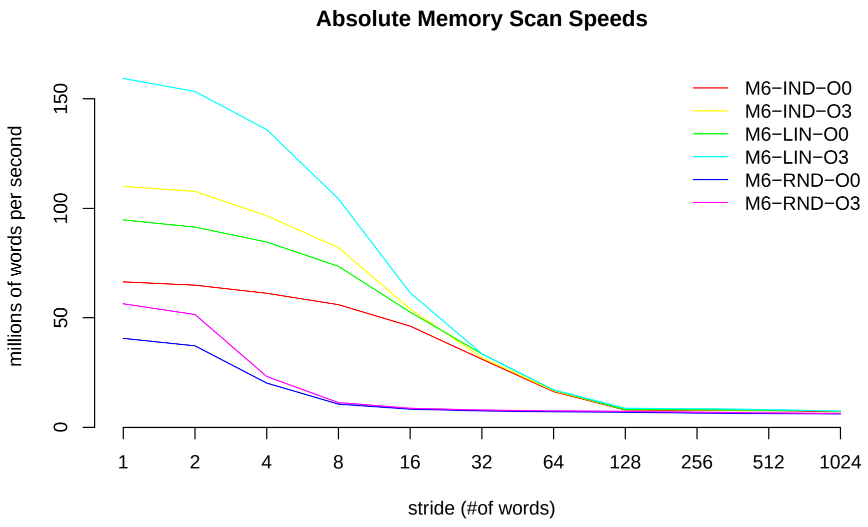

Figure B.4: The absolute performance of some linear scan primitives on **M6**.

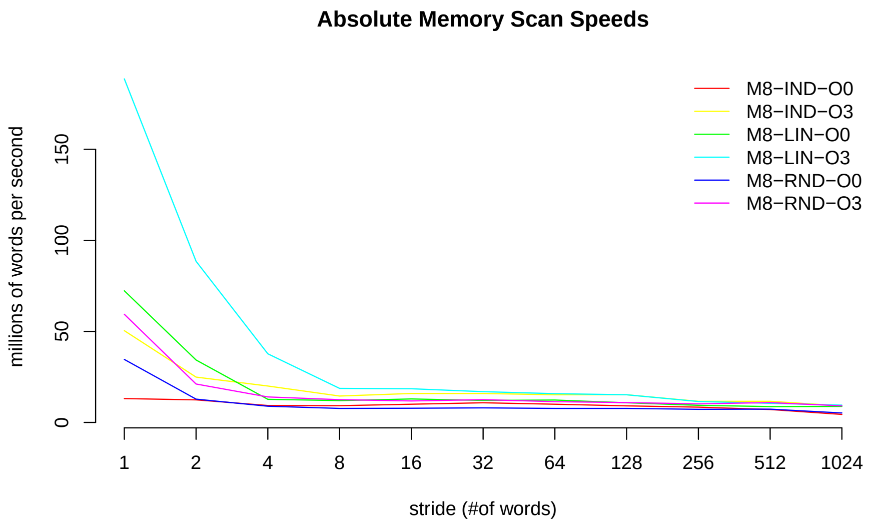

Figure B.5: The absolute performance of some linear scan primitives on **M8**.

C Appendix: some more experiments with *RSB*

C.1 Description of Experiments

Earlier in this thesis document (see §2.3), we have explained that our way of organizing a sparse matrix in submatrices (or *sparse blocks*: see §2.2) is a particular form of *cache blocking*. This organization of a sparse matrix was conceived to reduce *cache misses* during the shared memory parallel execution of *SpMV*,*SpMV-T*,*SpSV*, or *SpSV-T*. It is known (see §2.1 and the mentioned literature) that an effective implementation of a cache blocking technique prevents excessive cache misses (and as a direct consequence, raises the floating point rate) on matrices larger (in terms of *memory footprint*) than the outermost (or *higher level*) cache sizes. In this appendix chapter we present the results of a number of experiments (performed on machine **M4**; see Table A.9) aimed at the empiric assessment of the quality of our design and (shared memory parallel) implementation for both *SpMV* and its *transposed* variant (*SpMV-T*). In §1.1 and §1.2, we have discussed the inefficiencies of serial *SpMV-T* on row-ordered formats, like *CSR*/*COR*. It is also generally known that a *parallel SpMV-T* formulation for *CSR*/*COR* is seldom expected to be as efficient as *SpMV* (see Buluç et al. [BFF$^+$09, s.1]). The *RSB* format attempts at mitigating these inefficiencies, by partitioning the matrix in *sparse blocks* which are laid out in memory along a *space filling curve*-like ordering. See §2.3.3 for a definition and discussion of this ordering, and §5 for the details of a possible build procedure. The organization of blocks we propose allows a straightforward implementation of *SpMV-T*, both serial (see the sketch of *SpMV-T* for *RCSR/RSB* in §2.3.1) and parallel (see the discussion in §3.1). Due to space and time constraints, we cannot go through a very detailed discussion: we leave a thorough analysis of results as future work. Nevertheless, given our choice of experiments[1], the obtained results give evidence that our arguments about the efficiency of *RSB* for both *SpMV* and *SpMV-T* are sound. Throughout this appendix, we compare the *SpMV* performance of our *RSB* implementation to that of a proprietary, highly optimized, and architecture specific implementation of *CSR* routines in Intel's *Math Kernels Library* (**MKL**)[2]. In §C.2 we look at the performance of *SpMV* and *SpMV-T*, when running on the maximum number of threads; in §C.3 we look at the performance of *SpMV*,*SpMV-T*,*SpSV/SpSV-T*, but using only one thread[3]. Then, in §C.4 we look again at the performance of *SpMV* and *SpMV-T*; but this time on the *largest* (by nonzeroes count) matrices available in the *University of Florida sparse matrix collection* (see [Dav10]) which have not benchmarked in §C.2. Information for the symmetric matrices is shown in Table C.4; information for the unsymmetric ones in Table C.5. Some of the matrices we use in this appendix (the ones labeled with the *-l* suffix) are lower factors of an LU decomposition (see §A.2 for details) of the corresponding matrix, and thus with a substantially different nonzeroes pattern. Finally, we sum up our conclusions in §C.5.

C.2 Results for *SpMV* and *SpMV-T*, versus MKL

We have grouped results of a 12-threaded (the maximum available on **M4**), comparative run of both *SpMV* and *SpMV-T* in three plots: Fig. C.1 for square matrices, Fig. C.2 for non-square matrices, Fig. C.3 for symmetric (and thus, square) matrices. By the definition of symmetry, we have that *SpMV-T* on a symmetric matrix yields the same results as *SpMV*, (and in our implementation, using the same code): therefore Fig. C.3 shows only *SpMV* results. We consider 12 threads only (the maximum available) on the matrices summarized in Tables C.2, C.3, C.4. Notice how on most matrices, *RSB*'s *SpMV* performs better than **MKL**'s, and the gap between transposed and non-transposed *SpMV* is much smaller in *RSB* than it is in **MKL**.

[1]From these experiments, we have excluded smaller matrices (the ones with less than 2 millions of nonzeroes), as they may lead to cache reuse between subsequent *SpMV*s.

[2]Our copy of **MKL** was version "`10.3-0, Product, 20100927`". The **MKL** library runs (by design) on Intel architectures only.

[3]For the triangular solve operation we compare only *single threaded* executions, as **MKL** implementation of *SpSV* appears to be serial (its speed does not scale up with more threads).

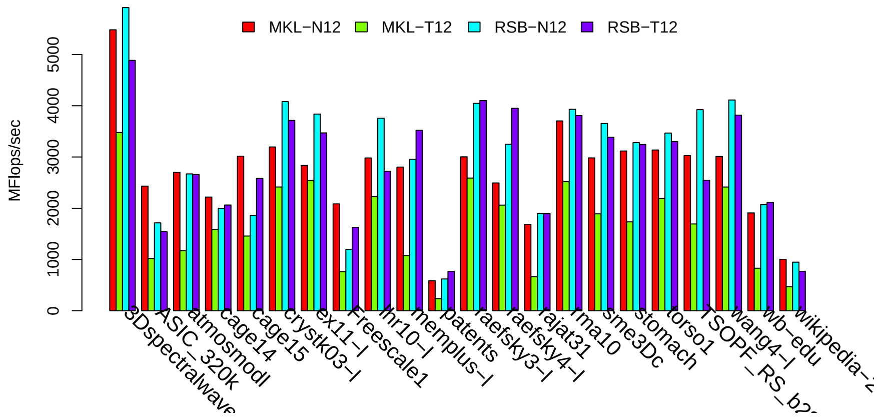

Figure C.1: *SpMV* performance on **M4**, versus **MKL**, 12 threads, square matrices from Table C.2.

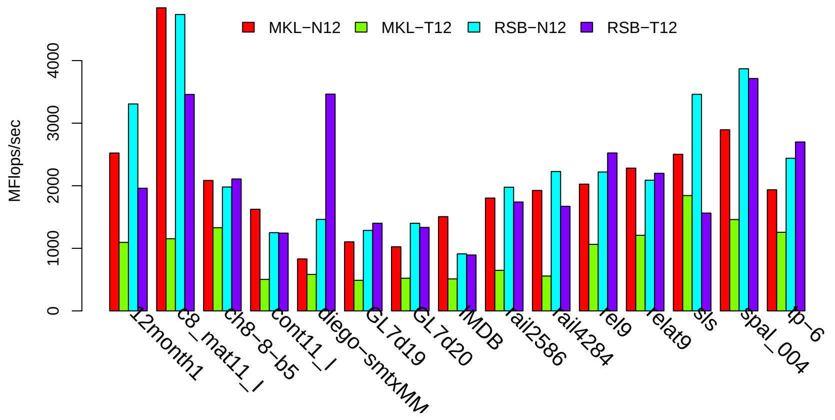

Figure C.2: *SpMV* performance on **M4**, versus **MKL**, 12 threads, non-square matrices from Table C.3.

matrix	r	c	nnz	nnz/r
af_shell9	504855	504855	9046865	17.92
as-Skitter	1696415	1696415	11095298	6.54
audikw_1	943695	943695	39297771	41.64
gsm_106857	589446	589446	11174185	18.96
human_gene1	22283	22283	12345963	554.05
human_gene2	14340	14340	9041364	630.50
mouse_gene	45101	45101	14506196	321.64
nlpkkt120	3542400	3542400	50194096	14.17
nlpkkt160	8345600	8345600	118931856	14.25
pkustk14	151926	151926	7494215	49.33
Si41Ge41H72	185639	185639	7598452	40.93

Table C.1: Additional large symmetric matrices.

C.3 Single Threaded, RSB versus MKL

Although importance of the peak performance (parallel) execution of *SpMV*/*SpMV-T* is critical to many applications, it is also interesting to compare the efficiency of our format for single threaded execution. As in the previous section, we have grouped results for square matrices (see Fig. C.4), non-square ones (see Fig. C.5), and (square) symmetric matrices (see Fig. C.6). We note that our implementation of *RSB* outperforms **MKL** in nearly all of the considered cases. Since it is highly likely that the **MKL** implementation applies machine specific code optimization techniques (while we did not), we draw two immediate conclusions. The first one is that presumably, the memory layout and *short* indices usage of *RSB* (see §4) played a role in the efficiency of *RSB*. The second one is that it is very likely that by applying machine-specific code optimizations techniques to the *RSB* implementation, one could obtain even higher performance. Finally, in Fig. C.7 the performance of *RSB* is compared to that of **MKL** when using a single thread in the execution of the *SpSV* kernel. Notice that results are comparable. Since the **MKL** implementation of *SpSV*/*SpSV-T* appears to be serial, there is no means of comparison to **MKL**'s partially parallel implementation (see §3.2).

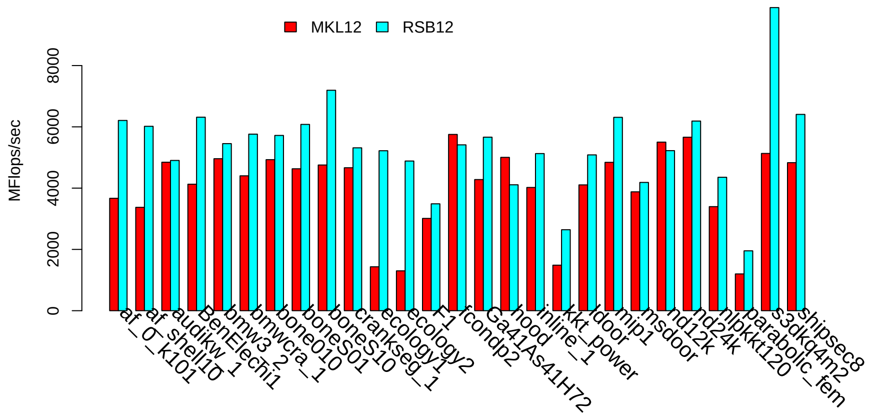

Figure C.3: *SpMV* performance on **M4**, versus **MKL**, 12 threads, symmetric matrices from Table C.4.

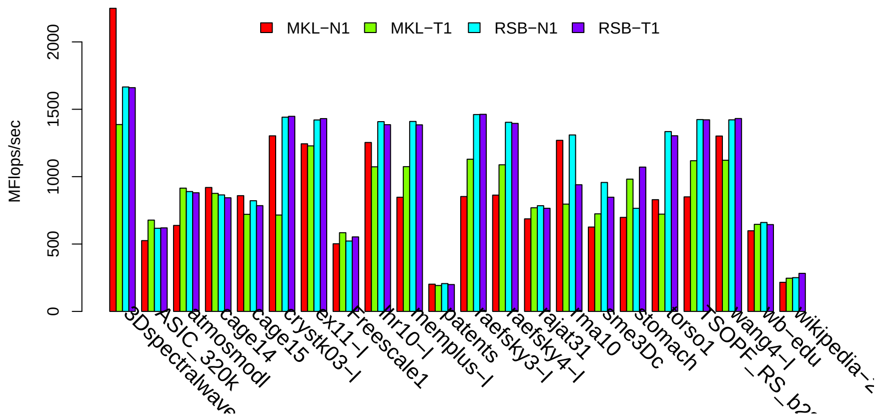

Figure C.4: *SpMV* performance on **M4**, versus **MKL**, one thread, square matrices.

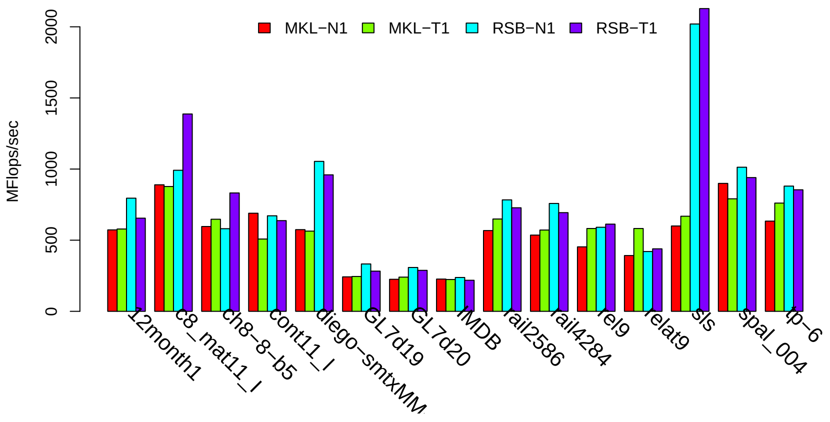

Figure C.5: *SpMV* performance on **M4**, versus **MKL**, one thread, non-square matrices.

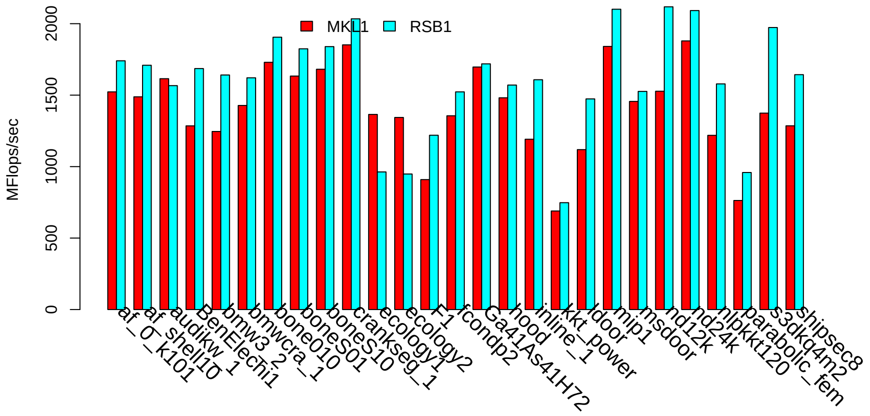

Figure C.6: *SpMV* performance on **M4**, versus **MKL**, one thread, symmetric matrices.

matrix	r	c	nnz	nnz/r
3Dspectralwave	680943	680943	17165766	25.21
ASIC_320k	321821	321821	2635364	8.19
atmosmodl	1489752	1489752	10319760	6.93
cage14	1505785	1505785	27130349	18.02
cage15	5154859	5154859	99199551	19.24
crystk03-l	24696	24696	13674087	553.70
ex11-l	16614	16614	6313293	380.00
Freescale1	3428755	3428755	18920347	5.52
lhr10-l	10672	10672	10749856	1007.30
memplus-l	17758	17758	11102358	625.20
patents	3774768	3774768	14970767	3.97
raefsky3-l	21200	21200	14944936	704.95
raefsky4-l	19779	19779	16534538	835.96
rajat31	4690002	4690002	20316253	4.33
rma10	46835	46835	2374001	50.69
sme3Dc	42930	42930	3148656	73.34
stomach	213360	213360	3021648	14.16
torso1	116158	116158	8516500	73.32
TSOPF_RS_b2383	38120	38120	16171169	424.22
wang4-l	26068	26068	22689509	870.40
wb-edu	9845725	9845725	57156537	5.81
wikipedia-20060925	2983494	2983494	37269096	12.49

Table C.2: Additional square matrices.

C.4 Big Matrices, versus MKL

Here we compare performance of *SpMV*/*SpMV-T* on a sample of the *biggest* (in terms of nonzeroes) matrices from the University of Florida collection not already considered in §C.2. Matrices data is summarized in Table C.4 and Table C.5.

With 12 threads (the maximum), see Fig. C.8 for unsymmetric matrices, and Fig. C.11 for the symmetric ones. For a single threaded execution, see Fig. C.9 for unsymmetric matrices, and Fig. C.10 for the symmetric ones. By comparing Fig. C.8 to Fig. C.9, we see that the advantage of *RSB* over **MKL** increases with the cores count, on unsymmetric matrices. To a lesser degree, this happens

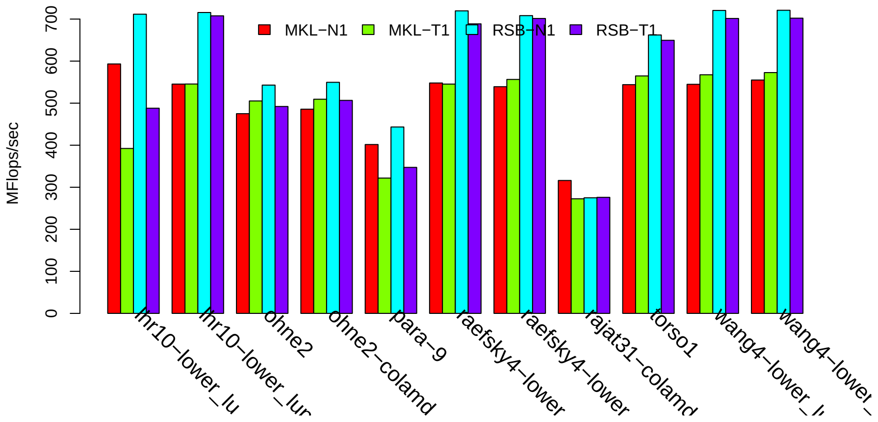

Figure C.7: *SpSV* performance on **M4**, versus **MKL**, single thread.

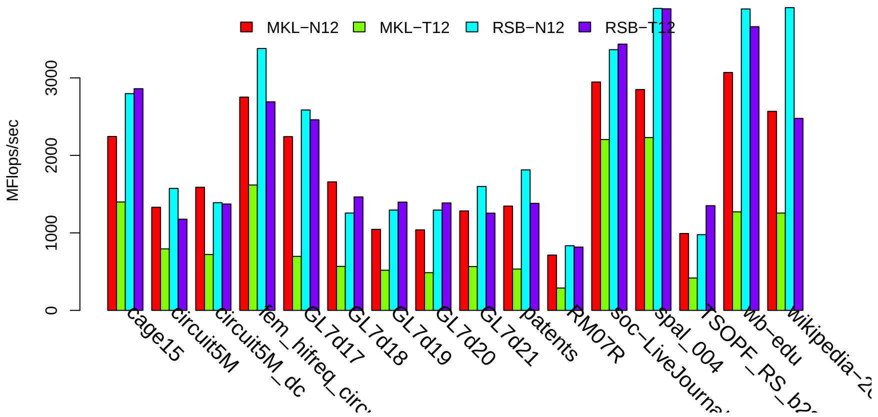

Figure C.8: *SpMV* performance on **M4**, versus **MKL**, 12 threads, large unsymmetric matrices summarized in Table C.5 .

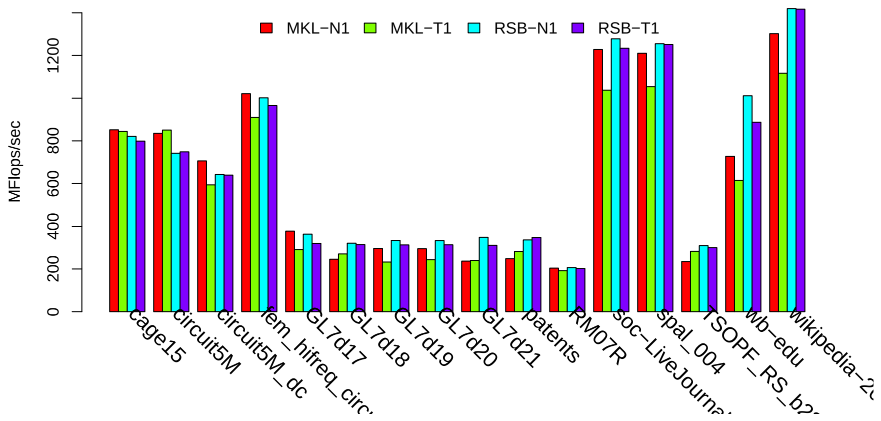

Figure C.9: *SpMV* performance on **M4**, versus **MKL**, 1 thread, large unsymmetric matrices.

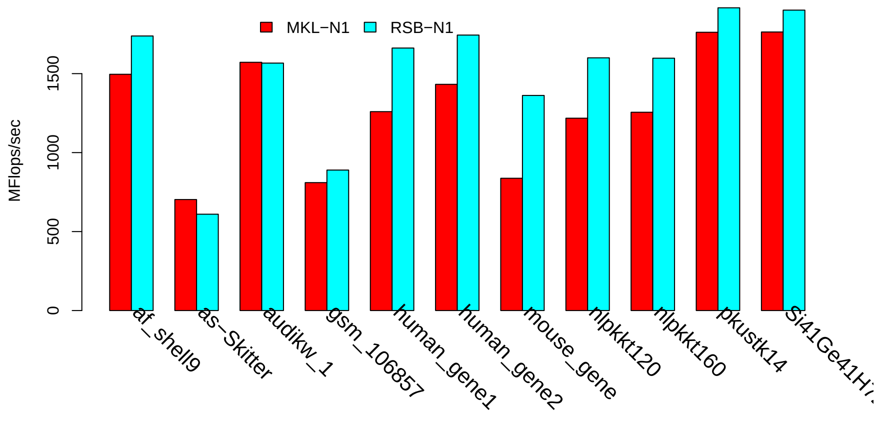

Figure C.10: *SpMV* performance on **M4**, versus **MKL**, 1 threads large symmetric matrices summarized in Table C.4 .

matrix	r	c	nnz	nnz/r
12month1	12471	872622	22624727	1814.19
c8_mat11_I	4562	5761	2462970	539.89
ch8-8-b5	564480	376320	3386880	6.00
cont11_l	1468599	1961394	5382999	3.67
diego-smtxMM-573x230k	573286	230401	41694697	72.73
GL7d19	1911130	1955309	37322725	19.53
GL7d20	1437547	1911130	29893084	20.79
IMDB	428440	896308	3782463	8.83
rail2586	2586	923269	8011362	3097.97
rail4284	4284	1096894	11284032	2633.99
rel9	9888048	274669	23667183	2.39
relat9	12360060	549336	38955420	3.15
sls	1748122	62729	6804304	3.89
spal_004	10203	321696	46168124	4524.96
tp-6	142752	1014301	11537419	80.82

Table C.3: Additional non-square matrices.

on symmetric matrices also; compare Fig. C.11 to Fig. C.10. On the symmetric matrices, however, the single threaded execution of *RSB* performs better than **MKL**.

C.5 Concluding remarks

The experiments presented in this appendix aimed at assessing the performance of our shared memory parallel implementation of two important computational kernels (*SpMV*/*SpMV-T*). As a reference for comparison, we have used a highly optimized, proprietary **Sparse BLAS** implementation: the one present in Intel's **MKL** library.

We can summarize the outcome of our experiments as:

- By noticing the performance gap between (parallel) *SpMV* and *SpMV-T* in **MKL**, we confirm that with the *CSR* format (used by **MKL**) it is inherently difficult to arrange *SpMV-T* computations to be as efficient as *SpMV*. We also notice that our *RSB* implementation not only greatly reduces the performance gap between *SpMV* and *SpMV-T*, but performs

matrix	r	c	nnz	nnz/r
af_shell9	504855	504855	9046865	17.92
as-Skitter	1696415	1696415	11095298	6.54
audikw_1	943695	943695	39297771	41.64
gsm_106857	589446	589446	11174185	18.96
human_gene1	22283	22283	12345963	554.05
human_gene2	14340	14340	9041364	630.50
mouse_gene	45101	45101	14506196	321.64
nlpkkt120	3542400	3542400	50194096	14.17
nlpkkt160	8345600	8345600	118931856	14.25
pkustk14	151926	151926	7494215	49.33
Si41Ge41H72	185639	185639	7598452	40.93

Table C.4: Additional large symmetric matrices.

often consistently better than **MKL**'s *CSR*.

- While we are unaware whether **MKL** uses (dynamical) *cache blocking* techniques (see §2.1) or not, we observe that non-parallel (single-threaded) performance of *RSB* is comparable, and often better than that of **MKL**. By recalling that a single-threaded execution is affected by memory bandwidth saturation *less* than multi-threaded execution is, and given the absence of machine specific optimizations in our code, our explanation for the superiority of *RSB* in the considered cases is that the *sparse blocking* and index saving (see §4.1) techniques of *RSB* have played the difference.

- The performance of *RSB*'s *symmetric SpMV* implementation is almost always superior to that of **MKL**. We regard this result as being the combination of the two previous points about *SpMV-T*[4] and *cache friendliness*.

Therefore, the conclusions we draw are positive: the *RSB* format not only favours efficient parallel execution of both *SpMV* and *SpMV-T* (matrices which are large enough to be *cache blocked*), but it is also ready to accommodate further, machine specific optimizations, while being readily comparable to an efficient, commercial library.

An extension of the work we have presented in this appendix would comprehend a detailed evaluation of results, with a comparison to the *CSB* prototype

[4]Recall from §1.4 that symmetric kernels have a memory access pattern that combines that of both *SpMV* and *SpMV-T*.

matrix	r	c	nnz	nnz/r
cage15	5154859	5154859	99199551	19.24
circuit5M	5558326	5558326	59524291	10.71
circuit5M_dc	3523317	3523317	19194193	5.45
fem_hifreq_circuit	491100	491100	20239237	41.21
GL7d17	1548650	955128	25978098	16.77
GL7d18	1955309	1548650	35590540	18.20
GL7d19	1911130	1955309	37322725	19.53
GL7d20	1437547	1911130	29893084	20.79
GL7d21	822922	1437547	18174775	22.09
patents	3774768	3774768	14970767	3.97
RM07R	381689	381689	37464962	98.16
soc-LiveJournal1	4847571	4847571	68993773	14.23
spal_004	10203	321696	46168124	4524.96
TSOPF_RS_b2383	38120	38120	16171169	424.22
wb-edu	9845725	9845725	57156537	5.81
wikipedia-20070206	3566907	3566907	45030389	12.62

Table C.5: Additional large general matrices.

implementation (see §2.2) also; recall that the *CSB* format was designed for being *friendly* both to memory/cache traffic and to the shared memory parallel *SpMV-T*.

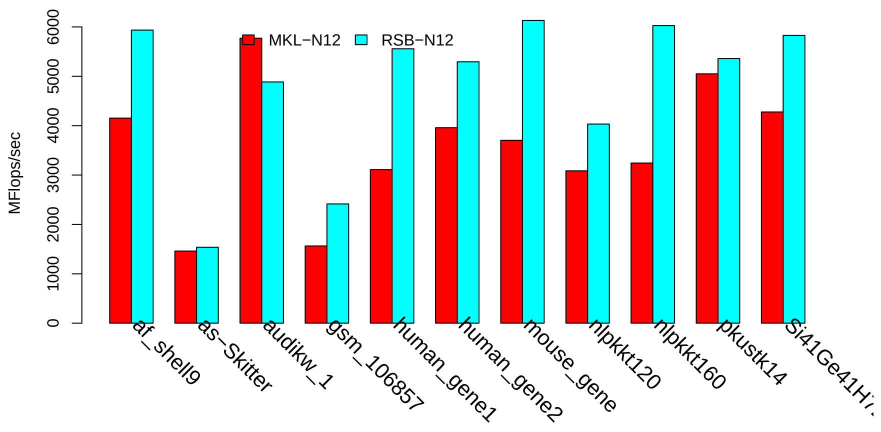

Figure C.11: *SpMV* performance on **M4**, versus **MKL**, 12 threads, large symmetric matrices.

D
Appendix: notation and conventions

Throughout the whole text, we tried to be consistent with the traditional notation used for expressing algorithms in most textbooks, as that of Cormen et al.: [CLRS09]. Additionally, we follow a number of style conventions commonly encountered in numerical linear algebra books, prominently, that of Golub et al.: [GL96, Ch. 1]. In this **Matlab**-like style, we access numerical vectors or matrices, with 1-based indices.

We also use subscripts (often characters i, j or k, l, or p, q) to mean indices, and lowercase letters for submatrices or individual elements of matrices. We refer to the element of matrix A at row i and column j with a_{ij}. Sometimes, for clarity, we use a comma between the indices, as in $a_{i,j+1}$.

So, for $A = \begin{pmatrix} 11 & 12 \\ 21 & 22 \end{pmatrix}$, we have $a_{11} = 11, a_{21} = 21, a_{12} = 12, a_{22} = 22$.

As encouraged by **Matlab**'s programming language, we have a notation for submatrices; that is $a_{i_0:i_1,j_0:j_1}$ refer to that submatrix of A which is enclosed between rows i_0 and i_1 inclusive, and columns j_0 and j_1 inclusive.

So, if $B = \begin{pmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \\ 31 & 32 & 33 \end{pmatrix}$, then $b_{1:2,2:3} = b_{1:1+1,2:4-1} = \begin{pmatrix} 12 & 13 \\ 22 & 23 \end{pmatrix}$ and $b_{1,:} = b_{1,1:3} = \begin{pmatrix} 11 & 12 & 13 \end{pmatrix}$. We use symbols $\alpha, \beta, \epsilon, \mu$ for scalar constants, characters x, y, z for vectors, m, k, n for integer scalar numbers, and uppercase characters A, L, U, D, T for matrices. We indicate the transpose of A with A^T.

When we wish to express an assignment, we use the left arrow ("$\leftarrow$") pointing from the expression being evaluated to the variable being assigned. So, with $y \leftarrow \beta y + \alpha A^T x$, we are overwriting the y vector variable, with its value scaled by β, and adding the product of the αA matrix transposed, by the x vector.

In the algorithms, we use the "=" sign to express equality, not assignment.

Although linear algebra assumes operation on *fields* (as *real* or *complex* numbers are), the computer representations of numbers we work with do not have the field property[1], and we leave the accurate study of the numerical properties of our algorithms as a separate problem.

In many algorithms, we use uppercase acronyms for *index arrays* (arrays whose entries are used to access other arrays); i.e.: $IA = (1, 2, 3)$. With these arrays, we follow the convention of having 1-based index entries.

Sometimes (e.g.: Fig. 5.5), we have used square brackets for arrays or index arrays; so rather than using $IA = (11, 22, 33)$, we have used $IA = [11, 22, 33]$. This choice has been motivated by the context (non-numerical algorithms, possible ambiguities if we had used round brackets for arrays access). When accessing arrays, we use the chosen bracket notation, always $1-$based; so in both the two above mentioned cases, we have $IA(2) = 22$, $IA[2] = 22$, $IA(1:2) = (11, 22)$ and $IA[1:2] = [11, 22]$.

In §5, we also use some more advanced **Matlab** notation; that of `struct` variables (which could be regarded as fixed key-value pairs). Namely, we use a *dot* (".") on a variable identifier to access its members. So, $s.m \leftarrow s.nnz - 1$ means that we assign to member m of variable s, the value of member nnz of variable s, subtracted by one. In §5, we also use some custom notation: we use "$\overset{p}{\leftarrow}$" to signify assignment to a *pointer variable*, which likewise to the C language, stores an address, and allows its use with a bracketed notation—that is, array semantics. Only in the case of *pointer assignment*, we use a 0-based notation. So, if $V = (11, 22, 33)$, then $P \overset{p}{\leftarrow} V + 1$ means that $P(1) = 22$ and $P(2) = 33$; this because P *points* to the *second* element of V.

An additional notation we use is that of **parallel** loops, used in §3 and §5. We have used two parallel constructs: **begin parallel**/**end parallel**, and **parallel foreach**. In the first construct, a shared memory parallel region is created; that is, newly declared variables are treated as *private* to a given *execution thread*, while the previously allocated variables are assumed to remain *shared*. The **parallel foreach** construct is similar when it regards the *scoping* of its variables. Differently from the previous construct, this one iterates the execution of a given region *for each* possible value available over the range of the **foreach** construct, by assigning values to the threads as they become available.

In the algorithms listings, we often insert comments, enclosed by "/*" and "*/" markers.

[1]For instance, with floating point numbers, multiplication is **not** distributive over addition.

Bibliography

[AMD07] AMD Corporation. *AMD64 Architecture Programmer's Manual Volume 1: Application Programming Rev 3.14*, 2007.

[APJ$^+$07] Buttari Alfredo, Luszczek Piotr, Kurzak Jakub, Dongarra Jack, and Bosilca George. SCOP3: A rough guide to scientific computing on the PlayStation 3. Technical report, Innovative Computing Laboratory, University of Tennessee Knoxville, April 2007. (UT-CS-07-595).

[Arn10] Jörg Arndt. *Matters Computational.* Springer, 2010.

[BA06] David Bateman and Andy Adler. Sparse matrix implementation in Octave. In *Octave Workshop, NIST, Gaithersburg, MD, USA, April 20-21, 2006*, 2006.

[BB09] Muthu Manikandan Baskaran and Rajesh Bordawekar. Optimizing sparse Matrix-Vector multiplication on GPUs. Technical report, IBM Corporation, 2009.

[BBC$^+$94] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition.* SIAM, Philadelphia, PA, 1994.

[BBKW98] Aart J. C. Bik, Peter Brinkhaus, Peter M. W. Knijnenburg, and Harry A. G. Wijshoff. The automatic generation of sparse primitives. *ACM Trans. Math. Softw.*, 24(2):190–225, 1998.

[BBW97] Aart J. C. Bik, Peter J. H. Brinkhaus, and Harry A. G. Wijshoff. The sparse compiler MT1: A reference guide. Technical report, Leiden University, The Nederlands, 1997.

[Bel66] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Syst. J.*, 5(2):78–101, 1966.

[BELF07] Alfredo Buttari, Victor Eijkhout, Julien Langou, and Salvatore Filippone. Performance optimization and modeling of blocked sparse kernels. *IJHPCA*, 21:467–484, 2007.

[BFF$^+$09] Aydın Buluç, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In Friedhelm Meyer auf der Heide and Michael A. Bender, editors, *SPAA*, pages 233–244. ACM, 2009.

[BG08] Aydın Buluç and John R. Gilbert. On the Representation and Multiplication of Hypersparse Matrices. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS 2008)*, April 2008.

[BJK$^+$95] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, Santa Barbara, California, July 1995.

[Bul10] Aydin Buluç. *Linear Algebraic Primitives for Parallel Computing on Large Graphs.* PhD thesis, University of California Santa Barbara, 2010. Supervised by John R. Gilbert. Available as of June 2011 on `http://gauss.cs.ucsb.edu/~aydin/Buluc_Dissertation.pdf`.

[But06] Alfredo Buttari. *Software Tools for Sparse Linear Algebra Computations.* PhD thesis, Università degli studi di Roma Tor Vergata, 2006. Supervised by Salvatore Tucci. Available as of June 2011 on `http://graal.ens-lyon.fr/~abuttari/mypapers/thesis.pdf`.

[BW96] Aart J.C. Bik and Harry A.G. Wijshoff. Automatic data structure selection and transformation for sparse matrix computations. In *IEEE Trans. On Parallel And Distributed Systems*, volume 7, February 1996.

[BWOD11] Aydın Buluç, Samuel Williams, Leonid Oliker, and James Demmel. Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication. In *Proc. IPDPS*, 2011.

[Che89] Hui Cheng. Vector pipelining, chaining, and speed on the IBM 3090 and Cray X-MP. *IEEE Computer*, 22(9):31–46, 1989.

[CHL$^+$96] Sandra Carney, Michael A. Heroux, Guangye Li, Roldan Pozo, Karin A. Remington, and Kesheng Wu. A revised proposal for a Sparse BLAS toolkit. Technical report, 1996. (SPARKER working note #3).

[CLPT02] S. Chatterjee, A.R. Lebeck, P.K. Patnala, and M. Thottethodi. Recursive array layouts and fast matrix multiplication. *Parallel and Distributed Systems, IEEE Transactions on*, 12(11), November 2002.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms.* MIT Press, 3rd ed. edition, 2009.

[Com11] Committee on Sustaining Growth in Computing Performance. *The Future of Computing Performance: Game Over or Next Level?* National Academy Press, 2011.

[Dav07] Tim Davis. Jim Wilkinson's definition of a sparse matrix. January 2007. As of June 2011, appearing on the Numerical Algorithms mailing list (Digest, Volume 7, Number 6), on `http://www.netlib.org/na-digest-html/07/v07n06.html#2`.

[Dav10] Tim Davis. University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software, to appear*, 2010. (submitted to ACM TOMS; as of June 2011, on `http://www.cise.ufl.edu/~davis/techreports/matrices.pdf`).

[DdSF10] P. D'Ambra, D. di Serafino, and S. Filippone. MLD2P4: a package of parallel algebraic multilevel domain decomposition preconditioners in Fortran 95. *ACM Transactions on Mathematical Software*, 37(3), 2010.

[DDSvdV98] J.J. Dongarra, Iain S. Duff, Danny C. Sorensen, and Henk van der Vorst. *Numerical linear algebra for high-performance computers.* Society for Industrial Mathematics, 1998.

[DEG$^+$99] James W. Demmel, Stanley C. Eisenstat, John R. Gilbert, Xiaoye S. Li, and Joseph W. H. Liu. A supernodal approach to sparse partial pivoting. *SIAM J. Matrix Analysis and Applications*, 20(3):720–755, 1999.

[DEL01] Jack Dongarra, Victor Eijkhout, and Piotr Luszczek. Recursive approach in sparse matrix lu factorization. *Scientific Programming*, 9(1):51–60, 2001.

[DGLN04] T. A. Davis, J. R. Gilbert, S. Larimore, and E. Ng. Algorithm 836: COLAMD, a column approximate minimum degree ordering algorithm. *ACM Transactions on Mathematical Software*, 30(3):377–380, September 2004.

[DHP02] Iain S. Duff, Michael A. Heroux, and Roldan Pozo. An overview of the sparse basic linear algebra subprograms: The new standard from the BLAS Technical Forum. *ACM Trans. Math. Softw.*, 28(2):239–267, 2002.

[DMP90] J.-L. Dekeyser, Ph. Marquet, and Ph. Preux. Vector addressing processor for direct and indirect accesses. *Microprocessing and Microprogramming*, 30:657 – 664, 1990.

[Dre07] Ulrich Drepper. What every programmer should know about memory. 2007. Available as of June 2011 on http://lwn.net/Articles/250967/.

[FALM] R. Fischer, M. Ast, J. Labarta, and H. Manz. A dynamic task graph parallelization approach. In *Proceedings of IASS-IACM-2000: Fourth International Colloquium on Computation of Shell and Spatial Structures, June 4-7, 2000 in Chania-Crete, Greece.*

[FB74] Raphael Finkel and J.L. Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9, March 1974.

[FC00] Salvatore Filippone and Michele Colajanni. PSBLAS: A library for parallel linear algebra computation on sparse matrices. *ACM Transactions on Mathematical Software*, 26(4):527–550, December 2000.

[FLPR] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *In the 40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA.*

[FSF10] FSF. The free software definition, v.1.92 (web page), March 2010. As of June 2011, on `http://www.gnu.org/philosophy/free-sw.html`.

[GL96] Gene H. Golub and Charles F. Van Loan. *Matrix computations (3rd ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[GL09] Peter Gottschling and Dag Lindbo. Generic compressed sparse matrix insertion: algorithms and implementations in MTL4 and FEniCS. In *POOSC '09: Proceedings of the 8th workshop on Parallel/High-Performance Object-Oriented Scientific Computing*, pages 1–8. ACM, 2009.

[G.M66] G.M.Morton. A computer oriented geodetic data base and a new technique in file sequencing. *Tech. Rep.*, Mar. 1966.

[GP07] Joe Gebis and David A. Patterson. Embracing and extending 20th-century instruction set architectures. *IEEE Computer*, 40(4):68–75, 2007.

[GRW07] Fred G. Gustavson, John K. Reid, and Jerzy Waśniewski. Algorithm 865: Fortran 95 subroutines for Cholesky factorization in block hybrid format. *ACM Trans. Math. Softw.*, 33(1):8, 2007.

[Gus97] F. G. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM J. Res. Dev.*, 41(6):737–756, 1997.

[GW04] Steven T. Gabriel and David S. Wise. The Opie compiler from row-major source to Morton-ordered matrices. In *WMPI '04: Proceedings of the 3rd workshop on Memory performance issues*, pages 136–144, New York, NY, USA, 2004. ACM.

[GWJ08] Peter Gottschling, David S. Wise, and Adwait Joshib. Generic support of algorithmic and structural recursion for scientific computing. *The International Journal of Parallel, Emergent and Distributed Systems*, (0), August 2008.

[Hac99] Wolfgang Hackbusch. A sparse matrix arithmetic based on H-matrices. Part I: Introduction to H-matrices. *Computing*, (62):89–108, 1999.

[HN08] José R. Herrero and Juan J. Navarro. Hypermatrix oriented supernode amalgamation. *The Journal of Supercomputing*, 46(1):84–104, October 2008.

[Im00] Eun-Jin Im. *Optimizing the Performance of Sparse Matrix-Vector Multiplication*. PhD thesis, University of California Berkeley, Jun 2000. As of June 2011, available on `http://www.eecs.berkeley.edu/Pubs/TechRpts/2000/5556.html`; supervised by Katherine A. Yelick.

[Int08a] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. December 2008.

[Int08b] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture*. November 2008.

[Int09] Intel Corporation. *Intel® Cilk++ SDK Programmer's Guide*. October 2009.

[Int10] Intel Corporation. *Intel® Threading Building Blocks, Reference Guide, v.1.20*. May 2010.

[IST04] Dror Irony, Gil Shklarski, and Sivan Toledo. Parallel and fully recursive multifrontal sparse Cholesky. *Future Generation Computer Systems*, 20(3):425 – 440, 2004.

[IY99] Eun-Jin Im and Katherine Yelick. Optimizing sparse matrix vector multiplication on SMPs. In *In Proc. of the 9th SIAM Conf. on Parallel Processing for Sci. Comp*, 1999.

[IYV04a] E. J. Im, K. Yelick, and R. Vuduc. SPARSITY: Optimization framework for sparse matrix kernels. *International Journal of High Performance Computing Applications*, 18(1):135, 2004.

[IYV04b] Eun-Jin Im, Katherine A. Yelick, and Richard Vuduc. SPARSITY: Framework for optimizing sparse matrix-vector multiply. *International Journal of High Performance Computing Applications*, 18(1):135–158, February 2004.

[JMC05] Guohua Jin and John Mellor-Crummey. Using space-filling curves for computation reordering. *Proceedings of the Los Alamos Computer Science Institute Sixth Annual Symposium*, October 2005.

[KGK08] Kornilios Kourtis, Georgios Goumas, and Nectarios Koziris. Improving the performance of multithreaded sparse matrix-vector multiplication using index and value compression. *Computing Frontiers*, pages 87–96, 2008.

[Knu97] Donald E. Knuth. *The art of computer programming, volume 1 (3rd ed.): fundamental algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[LK00] J. K. Lawder and P. J. H. King. Using space-filling curves for multi-dimensional indexing. In *Lecture Notes in Computer Science*, pages 20–35, 2000.

[LW07] K. Patrick Lorton and David S. Wise. Analyzing block locality in Morton-order and Morton-hybrid matrices. *SIGARCH Computer Architecture News*, (35), 2007.

[May09] Jan Mayer. Parallel algorithms for solving linear systems with sparse triangular matrices. *Computing*, 86(4):291–312, 2009.

[MC69] A. C. McKellar and E. G. Coffman, Jr. Organizing matrices and matrix operations for paged memory systems. *Commun. ACM*, 12(3):153–165, 1969.

[MDF] Richard Tran Mills, Eduardo F. Dazevedo, and Mark R. Fahey. Progress towards optimizing the PETSc numerical toolkit on the Cray X1. In *Proceedings of the Cray User Group 2005 Technical Meeting, Albuquerque, NM, May 16-19, 2005.*

[MFG$^+$10] Michele Martone, Salvatore Filippone, Paweł Gepner, Marcin Paprzycki, and Salvatore Tucci. Use of hybrid recursive CSR/COO data structures in sparse matrices-vector multiplication. In *Proceedings of the International Multiconference on Computer Science and Information Technology*, Wisła, Poland, October 2010.

[MFPT10a] Michele Martone, Salvatore Filippone, Marcin Paprzycki, and Salvatore Tucci. About the assembly of recursive sparse matrices. In *Proceedings of the International Multiconference on Computer Science and Information Technology*, Wisła. Poland, October 2010.

[MFPT10b] Michele Martone, Salvatore Filippone, Marcin Paprzycki, and Salvatore Tucci. On BLAS operations with recursively stored sparse matrices. In *Proceedings of the International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, Timisoara, Romania, September 2010.

[MFPT10c] Michele Martone, Salvatore Filippone, Marcin Paprzycki, and Salvatore Tucci. On the usage of 16 bit indices in recursively stored sparse matrices. In *Proceedings of the International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, Timisoara, Romania, September 2010.

[MFT$^+$10] Michele Martone, Salvatore Filippone, Salvatore Tucci, Marcin Paprzycki, and Maria Ganzha. Utilizing recursive storage in sparse matrix-vector multiplication - preliminary considerations. In Thomas Philips, editor, *CATA*, pages 300–305. ISCA, 2010.

[PA97] Ali Pinar and Cevdet Aykanat. Sparse matrix decomposition with optimal load balancing. In *High-Performance Computing, 1997. Proceedings. Fourth International Conference on*, pages 224 – 229, 1997.

[PH04] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 4th edition, 2004.

[PHP03] N. Park, B. Hong, and V.K. Prasanna. Tiling, block data layout, and memory hierarchy performance. *Parallel and Distributed Systems, IEEE Transactions on*, 14(7), July 2003.

[Pis] Sergio Pissanetzky. *Sparse matrix technology (electronic edition)*. self-published. As of June 2011, available by contacting the author through `http://www.scicontrols.com`.

[pos08] Standard for information technology— portable operating system interface (POSIX) (IEEE std 1003.1), 2008.

[PPP04] J.S. Park, M. Penner, and V.K. Prasanna. Optimizing graph algorithms for improved cache performance. *Parallel and Distributed Systems, IEEE Transactions on*, 15(9), 2004.

[Pro99] Harald Prokop. Cache-oblivious algorithms. Master's thesis, Department of Electrical Engineering and Computer Science at the Massachussets Institute of Technology, June 1999. Thesis supervisor: Charles E. Leiserson. As of May 2011, available at `http://supertech.csail.mit.edu/papers/Prokop99.pdf`.

[PSK11] M. Papadrakakis, G. Stavroulakis, and A. Karatarakis. A new era in scientific computing: Domain decomposition methods in hybrid cpu-gpu architectures. *Computer Methods in Applied Mechanics and Engineering*, 200:1490 – 1508, 2011.

[PV05] Ali Pinar and Virginia Vassilevska. Finding nonoverlapping substructures of a sparse matrix. *Electronic Transaction on Numerical Analysis*, 21:107–124, 2005.

[RG] Raghu Ramakrishnan and Johannes Gehrke. *Database management systems.* McGraw-Hill, 2nd ed. edition.

[RW08] Rajeev Raman and David S. Wise. Converting to and from dilated integers. *IEEE Trans. on Computers*, pages 567–573, 2008.

[Saa94] Yousef Saad. SPARSKIT: a basic tool kit for sparse matrix computations, version 2. Technical report, Computer Science Department, University of Minnesota, Minneapolis, June 1994.

[Saa03] Y. Saad. *Iterative Methods for Sparse Linear Systems, 2nd edition.* SIAM, Philadelphia, PA, 2003.

[Sag96] Hans Sagan. *Space Filling Curves.* Springer-Verlag, 1996.

[Sam84] H. Samet. The quadtree and related hierarchical data structures. Technical Report 16, June 1984.

[Sam06] Hanan Samet. *Foundations of Multidimensional and Metric Data Structures.* Morgan Kaufmann, August 2006.

[TB97] L.N. Trefethen and D. Bau. *Numerical linear algebra.* Society for Industrial Mathematics, 1997.

[TBK03] Jeyarajan Thiyagalingam, Olav Beckmann, and Paul H. J. Kelly. An exhaustive evaluation of row-major, column-major and Morton layouts for large two-dimensional arrays. pages 340–351, 2003.

[TKB06] Sunil Tiyyagura, Uwe Küster, and Stefan Borowski. Performance improvement of sparse matrix vector product on vector machines. 3991:196–203, 2006.

[VB05] Brendan Vastenhouw and Rob H. Bisseling. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM Review*, (47):47–95, 2005.

[VDY05a] R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Journal of Physics: Conference Series*, volume 16, pages 521–530. Institute of Physics Publishing, 2005.

[VDY05b] Richard Vuduc, James Demmel, and Katherine Yelick. An interface for a self-adapting sparse kernel library. Technical report, Berkeley, CA, USA, September 2005.

[vFRS72] G. von Fuchs, J. R. Roy, and E. Schrem. Hypermatrix solution of large sets of symmetric positive-definite linear equations. *Computer Methods in Applied Mechanics and Engineering*, 1(2):197 – 216, 1972.

[VKH+02] R. Vuduc, S. Kamil, J. Hsu, R. Nishtala, J. W. Demmel, and K. A. Yelick. Automatic performance tuning and analysis of sparse triangular solve. In *ICS 2002: Workshop on Performance Optimization Via High-Level Languages and Libraries*, 2002.

[VP04] Virginia Vassilevska and Ali Pinar. Finding nonoverlapping dense blocks of a sparse matrix. Technical report, University of California, Feb 2004.

[Vud03] Richard Wilson Vuduc. *Automatic performance tuning of sparse matrix kernels (phd thesis)*. PhD thesis, University Of California, Berkeley, 2003. Available as of June 2011 on http://bebop.cs.berkeley.edu/pubs/vuduc2003-dissertation.pdf; supervised by Jim Demmel.

[WPD01] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27:3–35, 2001.

[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(4):3128–3154, 2009.

[YB10] A. N. Yzelman and Rob H. Bisseling. A cache-oblivious sparse matrix–vector multiplication scheme based on the Hilbert curve. (Unpublished manuscript, available on http://www.math.uu.

nl/people/yzelman/publications/yzelman10a_pre.pdf as of June 2011), 2010.