Parallel scalable induced dimension reduction
a tailored Krylov-type Maxwell solver

Landa, J.

Award date:
2017

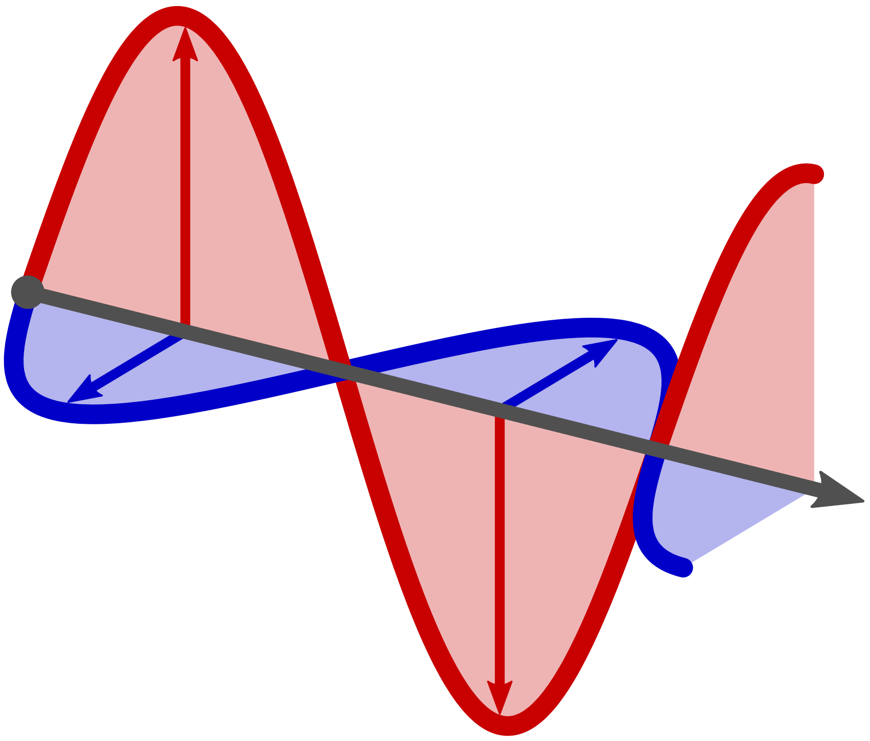
Eindhoven University of Technology
Department of Mathematics & Computer Science

Master Thesis

Parallel scalable Induced Dimension Reduction

A tailored Krylov-type Maxwell solver

J. Landa



TU/e supervision

dr. ir. Martijn Anthonissen

ASML mentorship

dr. Rohit Gupta
dr. Pavol Jancura
ir. Siu Hong Li

Abstract

This thesis provides guidelines on how to implement a Krylov-type method, IDR(s) in this case, for efficient parallel execution with OpenMP. The full linear system studied at ASML, obtained after discretizing Maxwell's time-harmonic equations, has a decomposition into sparse matrices with two different structures. Due to being a full system, the matrix-vector products required in a Krylov-type method take the most time to compute. We therefore look at the design of parallel sparse matrix-vector algorithms which take these two structures into account. Hereby, also CPU architecture is considered. Starting at CSR format algorithms which already account for a particular structure, improvements are imposed for both structures. The techniques used when designing these algorithms include the minimization of synchronization overhead of parallel programs, systematically counting the non-zeros of matrix with a certain structure to obtain efficient parallel loops, using a cheaper sparse matrix format and providing linear data access patterns. The resulting improvements are algorithms tailored towards the two matrix structures. Ultimately, the highest performing algorithm for each structure is integrated in the parallel solver. We confirm the benefits of these design techniques and conclude that for this particular system the total execution time is reduced significantly.

Keywords: Krylov-type methods; Induced Dimension Reduction; parallel computing; OpenMP; high-performance computing; time-harmonic Maxwell equations

Cover illustration: A polarized light wave with its electric and magnetic field.

"Science has achieved some wonderful things, of course, but I'd far rather be happy than right any day."
–
Douglas Adams
The Hitchhiker's Guide to the Galaxy

Acknowledgements

The people closest to me know I am a thankful person, so let us hope I do not forget someone.

First and foremost, my gratitude goes out to Martijn, for finding the time to supervise me on behalf of the TU/e while his sabbatical in Japan was already planned, and for being both a mathematician and a stylish person: uncommon yet exemplary. I would like to thank Rohit and Pavol for giving me the opportunity to fulfill an internship at ASML, for being true researchers and for reminding me of the value of sincerity, authenticity and creativity. Also, it was a pleasure to be mentored by Siu Hong, who always had time for a talk and was never shy to let his catchy laugh be heard. Further thanks go to Remco Dirks, Thomai Zacharopoulou and Mark van Kraaij for their involvement and willingness to explain some of the physics and mathematics.

My appreciation goes to the colleagues at YieldStar for all the talks over morning coffee, the interest they have shown in my project, every single intern joke I had to endure, and for their efforts to defeat a veteran student at table football. I especially enjoyed sitting next to software architects Johan and Carlo every day. The most hilarious moments I experienced during my project had to be with them.

My life as a student ends upon the completion of this work. The list of great people I have met during this era is immensely long yet distinguished. You know who you are. Special thanks go out to my fellows at the department, current and past, for all the fun times I could be part of. They made daily life into something nostalgic worth remembering. I particularly acknowledge my closest friends Bram, Pim and Simon for all the music, parties and personal stories shared over a beer or two.

This thesis could not possibly have been written without the everlasting support from my loving family. Regardless of a few unsuccessful attempts to explain them some mathematics, my parents have always showed genuine interest in whatever I was engaged in. Over the years, I was also truly lucky to visit my uncle Jan and aunt Caroline here in Eindhoven, for the occasional board game and a healthy bite. I am additionally mentioning my cousin Camiel. At least your name made it into a scientific publication now, even though you were unfortunately never meant to write one yourself.

The last and most prominent credits go to my wonderful sister, Esther, for guiding me through times she faced herself not too long ago, for sharing the passion for music and for being the most rocking nurse on the planet.

Jasper Landa
Eindhoven, October 2017

Contents

Chapter 1

Introduction

Phenomena in physics can generally be described by means of (partial) differential equations. There exist many families of differential equations, arising from families of these physical phenomena. Unfortunately, many differential equations do not have analytic solutions. A mathematician must then resort to numerical solving methods or simulations, which has shaped the mathematical subfield of *scientific computing*. Using discretization schemes, linear systems arise from such differential equations and their boundary conditions, the solutions of which are nowadays computed by Krylov-type linear solving methods. Coefficients of the matrices then represent a physical quantity (e.g. heat, pressure, density) measured at a position in the discretization space and at a certain moment in time.

1.1 Goals and motivation

The machines of ASML allow the assembly of computer chips with nanometer order precision. The various electric components are anchored on a *wafer*. After placements of these components, one obtains *profiles*: microscopic landscapes of different materials on a silicone substrate. Each of these profiles have a periodicity in either one or two dimensions. YieldStar, ASML's flagship metrology tool, is developed to measure the wafers a client can possibly design, to serve as an extra verification of correct chip production.

This means a wafer is aligned under a light source and a charge-coupled device, which then forms an image of a profile (a microscopic region of the wafer) and its embedded electric components: a *pupil*. In order to determine a reference pupil calculated from the profile, one needs to solve differential equations describing the behavior of light (Chapter 2), taking into account the reflective properties and shapes of the materials placed on the wafer. Ultimately, the reference pupil is subtracted from the measured pupil to detect prohibitive differences.

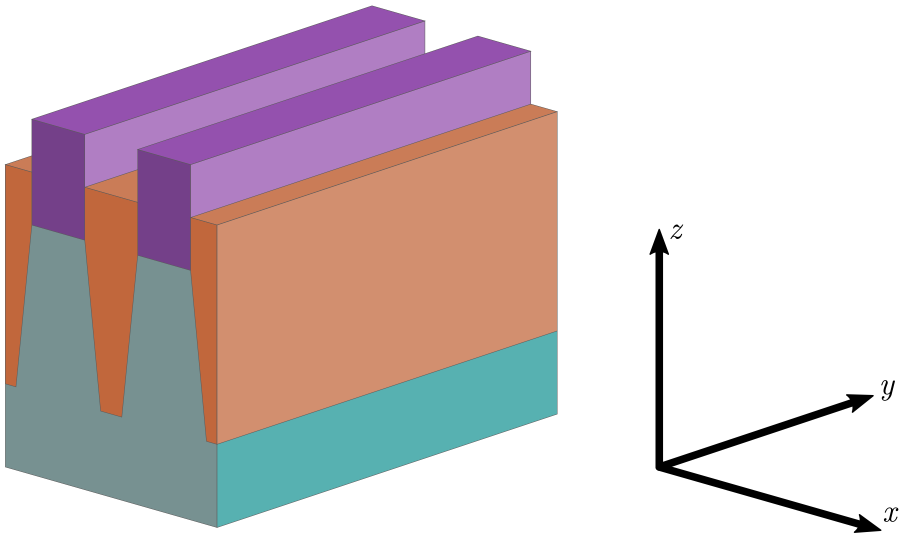


Figure 1.1: Example of a 1D-periodic profile.

The application computing this reference pupil solves the time-harmonic Maxwell equations twice to obtain the intensity of each pixel: once for both light polarization directions (x and y) parallel with the wafer. Currently, the intensity of each pixel is solved sequentially, i.e. the intensity of each pixel is computed by one processor core (Chapter 3). The cores of a system run concurrently to obtain multiple pixel intensities at once. This graduation project was set up as an exploratory research in how other parallel computing schemes can be deployed. It will therefore cover algorithms where all cores work on the same pixel at once. It is exactly the massively scalable algorithms that are well employable for costly parallel computations. After all, they perform well on computing clusters, where the CPUs have many cores at their disposal.

1.1.1 Thesis layout

An iterative solver commences with an initial guess and produces iterates to ultimately converge to the solution of the linear system. Such a solver inevitably requires matrix-vector products. They can occupy a respectable portion of the total linear solving time, as opposed to other linear algebra subroutines used in a linear solving algorithm, like inner products and vector scaling or addition. This depends on the sparsity and structure of the matrix.

After a preliminary analysis of the structure of the matrices which are obtained after the discretization and a study of hardware-optimized parallel algorithm design (Chapter 3), parallel scalable matrix-vector products will be presented and benchmarked (Chapter 4 and Chapter 5), in order to speed up the most time-consuming step of a linear solver. Afterwards, the best performing matrix-vector product algorithms will be incorporated in one choice of non-preconditioned Krylov-type solver (Chapter 6), the subroutines of which will also be parallelized where possible. The thesis concludes with a performance showcase for various matrix sizes.

Preconditioning will not be treated in this work. The rationale behind this choice is that the study of preconditioners is a rather large field. Furthermore, a preconditioned linear solver algorithm has steps where the preconditioner is applied to a vector. Naturally this also takes time, so one would also have to design a special matrix-vector product algorithm for this step, similar to the algorithms in Chapter 4 and Chapter 5, to mitigate the time spent on this operation.

1.1.2 Input assumptions and resources

The solving application takes profile geometries as input. Luckily, entire matrices could be exported by the application, given a geometry. The algorithms in this work will therefore act on stored matrices instead of these profiles. Since the input differs from the implementation, input assumptions will be made per algorithm, and the thesis will not emphasize on format conversion.

Ultimately, the algorithms will be implemented in the C++ language, aided by PARALUTION [11]: an OpenMP-based library providing pre-parallelized implementations of iterative solvers, equipped with pre-defined preconditioners and matrix and vector objects. In this thesis, only the basic version of PARALUTION (released under the GNU General Public License v3) is used. The imposed algorithms using PARALUTION are experimental and are therefore not included or integrated in any product of ASML Holding N.V.

Images are handmade, unless stated otherwise.

Chapter 2

Maxwell's equations

In 1864, James Clerk Maxwell (1831-1879) wrote down differential equations which describe electromagnetic laws in a general way. They hold on a microscopic to macroscopic scale and involve the dynamics of five vector fields: an electric field $\mathbf{E}(\mathbf{x}, t)$, the electric displacement $\mathbf{D}(\mathbf{x}, t)$, the magnetic induction $\mathbf{B}(\mathbf{x}, t)$, the magnetic field $\mathbf{H}(\mathbf{x}, t)$, and $\mathbf{J}(\mathbf{x}, t)$, which denotes the electric current density. These equations, stated below, are collectively known as Maxwell's equations:

$$\nabla \times \mathbf{E}(\mathbf{x}, t) = -\frac{\partial \mathbf{B}(\mathbf{x}, t)}{\partial t}, \tag{2.1}$$

$$\nabla \times \mathbf{H}(\mathbf{x}, t) = \mathbf{J}(\mathbf{x}, t) + \frac{\partial \mathbf{D}(\mathbf{x}, t)}{\partial t}, \tag{2.2}$$

$$\nabla \cdot \mathbf{D}(\mathbf{x}, t) = \rho(\mathbf{x}, t), \tag{2.3}$$

$$\nabla \cdot \mathbf{B}(\mathbf{x}, t) = 0, \tag{2.4}$$

where $\rho(\mathbf{x}, t)$ denotes a charge density function. These equations can be derived from, amongst other, the law of charge conservation. The five fields are related as follows:

$$\mathbf{D}(\mathbf{x}, t) = \varepsilon(\mathbf{x})\mathbf{E}(\mathbf{x}, t), \tag{2.5}$$

$$\mathbf{B}(\mathbf{x}, t) = \mu(\mathbf{x})\mathbf{H}(\mathbf{x}, t), \tag{2.6}$$

$$\mathbf{J}(\mathbf{x}, t) = \sigma(\mathbf{x})\mathbf{E}(\mathbf{x}, t). \tag{2.7}$$

Here, $\varepsilon, \mu, \sigma : \mathbb{R}^3 \to \mathbb{R}$ denote spatial functions for the electric permittivity, magnetic permeability and electric conductivity respectively. Note that these functions represent the material and geometry properties of the profile.

Magnetic fields can induce electric fields in closed loops (Faraday's law), the voltage of which is proportional to the changes in the magnetic field (2.1). The inverse occurs as well. A closed loop with an electric field induces a magnetic field (Ampère's law), which is proportional to the electric current and the changes in the electric field (2.2). The latter two are laws of Gauss: the electric flux $\nabla \cdot \mathbf{D}$ leaving a volume is proportional to the charge inside (2.3) and the total magnetic flux $\nabla \cdot \mathbf{B}$ through a closed surface is always zero (2.4).

A polarized light wave is a *transverse* or *plane* wave with electromagnetic properties. This means that at any spatial point $\mathbf{x}$ along the wave and any time t, there exists a propagation direction $\mathbf{k}$ such that

$$\begin{aligned}\mathbf{E}(\mathbf{x},t) \times \mathbf{H}(\mathbf{x},t) &= \beta\mathbf{k},\\ \big(\mathbf{E}(\mathbf{x},t), \mathbf{H}(\mathbf{x},t)\big) &= 0,\end{aligned}$$

for some $\beta \in \mathbb{R}$. Here, $\times$ and $(\cdot,\cdot)$ denote the outer and inner product respectively. The amplitude of the magnetic field is proportional to that of the electric field, the ratio of which is called the wave impedance η, which depends on the permittivity and permeability of the medium the wave is propagating in.

2.1 Derivation of the time-harmonic Maxwell equations

These electric and magnetic fields oscillate in-phase. Ordinary scalar sinusoidal oscillations are of the form $\alpha\cos(\omega t + \theta) = \operatorname{Re}\left(\alpha e^{i(\omega t+\theta)}\right)$, where α is the amplitude of the wave, $\omega = \frac{2\pi c}{\lambda}$ is the oscillation speed (with wavelength λ and light velocity c), and θ the phase angle.

Definition 2.1. *A field* $\mathbf{E}(\mathbf{x},t)$ *is time-harmonic if and only if for every* $\mathbf{x}$ *there exists an* $\mathbf{n} \in \mathbb{R}^3$ *and* $\theta \in \mathbb{R}$ *such that*

$$\mathbf{E}(\mathbf{x},t) = \mathbf{n}(\mathbf{x})\cos\big(\omega t + \theta(\mathbf{x})\big) = \mathbf{n}(\mathbf{x})\operatorname{Re}\left(e^{i(\omega t+\theta(\mathbf{x}))}\right).$$

Note that the entire oscillation is determined uniquely by $\mathbf{n}$, θ and ω. All time-harmonic fields can therefore be described by means of a complex-valued, time-independent phasor.

Definition 2.2. *The phasor of a time-harmonic field* $\mathbf{E}(\mathbf{x},t)$ *is the mapping* $\mathbf{E}_0 : \mathbb{R}^3 \to \mathbb{C}^3$ *with*

$$\mathbf{E}_0(\mathbf{x}) = \mathbf{n}(\mathbf{x})e^{i\omega\theta(\mathbf{x})},$$

such that

$$\mathbf{E}(\mathbf{x},t) = \operatorname{Re}\left(\mathbf{E}_0(\mathbf{x})e^{i\omega t}\right). \tag{2.8}$$

Note that this representation separates the dependency for space and time.

Naturally, these phasors also adhere to Maxwell's equations. Substituting the phasor representations for all fields in the time-varying Maxwell equations (2.1) through (2.4) yields the time-harmonic Maxwell equations, after canceling out the oscillatory, time-dependent part:

$$\begin{aligned}\nabla \times \mathbf{E}_0(\mathbf{x}) &= -i\omega\mathbf{B}_0(\mathbf{x}),\\ \nabla \times \mathbf{H}_0(\mathbf{x}) &= \mathbf{J}_0(\mathbf{x}) + i\omega\mathbf{D}_0(\mathbf{x}),\\ \nabla \cdot \mathbf{D}_0(\mathbf{x}) &= \rho(\mathbf{x}),\\ \nabla \cdot \mathbf{B}_0(\mathbf{x}) &= 0.\end{aligned}$$

Light waves are exposed down on the wafer. A part of this light is reflected by the various materials the profile consists of. The other part will be either transmitted or absorbed (Figure 2.1).

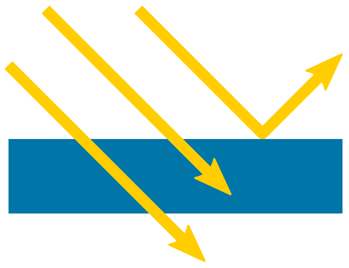


Figure 2.1: Transmission, absorption and reflection of light.

Each of these materials has reflection, absorption and transmission properties depending on the angle of incidence (Snell's law) and their respective permittivity and permeability. The result of the exposure therefore is a scatter of light waves with many directions, potentially (depending on profile complexity).

2.2 Computing a solution with the Volume Integral Method

Substituting the three field relations (2.5) through (2.7) in the first two time-harmonic equations results in

$$\begin{aligned}\nabla \times \mathbf{E}_0(\mathbf{x}) &= -i\omega\mu(\mathbf{x})\mathbf{H}_0(\mathbf{x}),\\ \nabla \times \mathbf{H}_0(\mathbf{x}) &= \big(\sigma(\mathbf{x}) + i\omega\varepsilon(\mathbf{x})\big)\mathbf{E}_0(\mathbf{x}).\end{aligned}$$

Because the response registered by the charge-coupled device is in free space, one may set the function μ equal to the constant μ_0, which denotes the permeability of free space. Then, by taking the curl of the first equation, the two equations above are combined into one equation for the electric field:

$$\begin{aligned}\nabla \times \big(\nabla \times \mathbf{E}_0(\mathbf{x})\big) &= -i\omega\mu_0\big(\nabla \times \mathbf{H}_0(\mathbf{x})\big)\\ &= -i\omega\mu_0\big(\sigma(\mathbf{x}) + i\omega\varepsilon(\mathbf{x})\big)\mathbf{E}_0(\mathbf{x})\\ &= \omega\mu_0\big(\omega\varepsilon(\mathbf{x}) - i\sigma(\mathbf{x})\big)\mathbf{E}_0(\mathbf{x})\\ &=: \gamma(\mathbf{x})\mathbf{E}_0(\mathbf{x}). \qquad (2.9)\end{aligned}$$

In this representation, γ could be considered a function of material properties, since the profile consists of materials, each with their own permittivity and conductivity, which are described by ε and σ.

From the linearity of Maxwell's equations, one can split up the electric field such that

$$\mathbf{E}_0(\mathbf{x}) = \mathbf{E}_0^b(\mathbf{x}) + \mathbf{E}_0^p(\mathbf{x}), \qquad (2.10)$$

$$\gamma(\mathbf{x}) = \gamma^b(\mathbf{x}) + \gamma^p(\mathbf{x}), \qquad (2.11)$$

where $\mathbf{E}_0^b(\mathbf{x})$ denotes the component of the electric field as a result from exposing light to an empty wafer, and $\mathbf{E}_0^p(\mathbf{x})$ denotes the component as a result from the profile materials. Let $\gamma^b(\mathbf{x})$ and $\gamma^p(\mathbf{x})$ be their respective property functions.

The property function γ^b of a plain wafer only depends on z and is therefore easier to use than γ^p. Then, by using γ^b in (2.9), conclude that

$$\nabla \times \big(\nabla \times \mathbf{E}_0^b(\mathbf{x})\big) - \gamma^b(\mathbf{x})\mathbf{E}_0^b(\mathbf{x}) = \mathbf{0}, \tag{2.12}$$

$$\nabla \times \big(\nabla \times \mathbf{E}_0^p(\mathbf{x})\big) - \gamma^b(\mathbf{x})\mathbf{E}_0^p(\mathbf{x}) = \big(\gamma(\mathbf{x}) - \gamma^b(\mathbf{x})\big)\mathbf{E}_0(\mathbf{x}). \tag{2.13}$$

The homogeneous differential equation (2.12) is solved analytically, yet the inhomogeneous equation (2.13) requires numerical solving by means of a volume integral.

Definition 2.3 (Green's function)**.** *Let a linear differential operator $\mathcal{L}(\mathbf{x})$ and a point $\mathbf{s}$ be given. A Green's function $G(\mathbf{x}, \mathbf{s})$ at the point $\mathbf{s}$ with respect to $\mathcal{L}$ is any solution to*

$$\mathcal{L}G(\mathbf{x}, \mathbf{s}) = \delta(\mathbf{x} - \mathbf{s})$$

where δ denotes the Dirac delta, i.e.

$$\delta(\mathbf{x} - \mathbf{s}) = \begin{cases} \infty & \textit{if } \mathbf{x} = \mathbf{s} \\ 0 & \textit{otherwise.} \end{cases}$$

These Green's functions [1] can be applied to solve a respectable amount of families of differential equations. A Green's function will aid in transforming the inhomogeneous equation (2.13) into a volume integral.

If one multiplies a Green's function with a function $f(\mathbf{s})$ and integrate with respect to $\mathbf{s}$ in a region Ω, one obtains

$$\begin{aligned} \int_\Omega \mathcal{L}G(\mathbf{x}, \mathbf{s}) f(\mathbf{s}) d\mathbf{s} &= \int_\Omega \delta(\mathbf{x} - \mathbf{s}) f(\mathbf{s}) d\mathbf{s}, \\ \mathcal{L}\left(\int_\Omega G(\mathbf{x}, \mathbf{s}) f(\mathbf{s}) d\mathbf{s}\right) &= f(\mathbf{x}), \end{aligned}$$

because on the left-hand side, the integral over $\mathbf{s}$ can be brought inside the differential operator, which only acts on $\mathbf{x}$, and because on the right-hand side, the expression reduces to $f(\mathbf{x})$ due to the Dirac delta. Hence, when solving a differential equation $\mathcal{L}u(\mathbf{x}) = f(\mathbf{x})$, the solution is given by

$$u(\mathbf{x}) = \int_\Omega G(\mathbf{x}, \mathbf{s}) f(\mathbf{s}) d\mathbf{s}. \tag{2.14}$$

Now, choose $\mathcal{L}$ to be the double curl operator and let Ω be the total exposure space, including the space occupied by profile materials. From (2.9, 2.10) and the linearity of $\mathcal{L}$ it follows that

$$\begin{aligned} \mathcal{L}\mathbf{E}_0(\mathbf{x}) &= \mathcal{L}\mathbf{E}_0^p(\mathbf{x}) + \mathcal{L}\mathbf{E}_0^b(\mathbf{x}), \\ &= \big(\gamma(\mathbf{x}) - \gamma^b(\mathbf{x})\big)\mathbf{E}_0^p(\mathbf{x}) + \gamma^b(\mathbf{x})\mathbf{E}_0^b(\mathbf{x}). \end{aligned}$$

Consequently, by (2.11) and (2.14), find that

$$\mathbf{E}_0^p(\mathbf{x}) = \int_\Omega G(\mathbf{x},\mathbf{s})\big(\gamma(\mathbf{x}) - \gamma^b(\mathbf{x})\big)\mathbf{E}_0(\mathbf{s})d\mathbf{s}, \tag{2.15}$$

$$\mathbf{E}_0(\mathbf{x}) = \mathbf{E}_0^b(\mathbf{x}) + \int_\Omega G(\mathbf{x},\mathbf{s})\big(\gamma(\mathbf{x}) - \gamma^b(\mathbf{x})\big)\mathbf{E}_0(\mathbf{s})d\mathbf{s}. \tag{2.16}$$

Since the difference in material properties $\gamma(\mathbf{x}) - \gamma^b(\mathbf{x})$, which can be considered a *contrast*, is not zero if and only if $\mathbf{x}$ is a spatial point occupied by profile materials, the region of integration Ω can be reduced to the volume occupied by these materials: a region that is in general much smaller than the entire exposure space. This allows for more efficient computation.

Because the profiles are considered infinitely periodic, the phasor $\mathbf{E}_0(\mathbf{x})$ and the material properties $\gamma(\mathbf{x})$ are, for a value of z, expanded as a Fourier series in x and y:

$$\mathbf{E}_0(\mathbf{x}) = \sum_{m_1,m_2=-\infty}^{\infty} \mathbf{E}'_{\mathbf{m}}(z)e^{2\pi i\left(\binom{x}{y},\kappa(\mathbf{m})\right)}, \tag{2.17}$$

$$\gamma(\mathbf{x}) = \sum_{m_1,m_2=-\infty}^{\infty} \gamma'_{\mathbf{m}}(z)e^{2\pi i\left(\binom{x}{y},\mathbf{Km}\right)}, \tag{2.18}$$

where $\mathbf{m} = (m_1, m_2)^T$ is an integer vector and where $\kappa : \mathbb{R}^2 \to \mathbb{R}^2$ is an affine map acting on $\mathbf{m}$, i.e., $\kappa(\mathbf{m}) = \mathbf{k}_0 + \mathbf{Km}$ for some vector $\mathbf{k}_0 \in \mathbb{R}^2$ and matrix $\mathbf{K} \in \mathbb{R}^{2\times 2}$. This affine map is profile-specific and defines the directions and phase of the two-dimensional periodicity. Terms of this series are computed for $|m_1|, |m_2| \leq M$, yielding a total of $(2M+1)^2$ vector terms, each with three components. Let $m_{xy} := 2M + 1$. Note that m_{xy} is odd. Verify that for a chosen z, there are $3m_{xy}^2$ unknowns.

In the z direction, certain heights are chosen to expand this series. If m_z denotes the amount of chosen z for the series expansion, the problem size totals to $n := 3m_{xy}^2 m_z$. The solution for the sampled z values, under the assumption that the electric field should be continuous, can be concatenated by means of, for example, local piecewise linear interpolation in z. To obtain a solution in the time domain, substitute this approximation into (2.8).

In [4], ways are described to arrive at a linear system from (2.16) and the Fourier expansions. The result is a system $(\mathbf{C} - \mathbf{GM})\mathbf{x} = \mathbf{b}$, where $\mathbf{C}$ represents the left-hand side of (2.16), which contains the coefficients for the entire electric field and the periodicity of the profile, $\mathbf{G}$ follows from the Green's function, $\mathbf{M}$ contains the contrast information and $\mathbf{b}$ harbors the series coefficients for $\mathbf{E}_0^b(\mathbf{x})$.

2.3 The linear systems

The matrices arising from the numerical scheme decompose into $\mathbf{A} := \mathbf{C} - \mathbf{GM}$, where $\mathbf{C}, \mathbf{G}, \mathbf{M} \in \mathbb{C}^{n\times n}$ are sparse.

Definition 2.4. *Let the sparsity of a complex-valued matrix be denoted by $\xi : \mathbb{C}^{n\times m} \to [0,1]$ such that*

$$\xi(\mathbf{A}) = \frac{\text{nnz}(\mathbf{A})}{nm}$$

where nnz *denotes the number of non-zeros.*

The matrix $\mathbf{A}$ is completely dense, i.e., $\xi(\mathbf{A}) = 1$. Storing $\mathbf{C}$, $\mathbf{G}$ and $\mathbf{M}$ separately therefore yields a lower memory cost. Solving this system will give a numerical solution to the differential equations, meaning that an $\mathbf{x} \in \mathbb{C}^n$ needs to be found such that $\mathbf{Ax} = \mathbf{b}$.

Since $\mathbf{C}$, $\mathbf{G}$ and $\mathbf{M}$ are sparse, computing $\mathbf{Ax}^{(k)}$ essentially costs three sparse matrix-vector (SpMV) products. Hence, applying $\mathbf{C}$ and $\mathbf{M}$ to iterate $\mathbf{x}^{(k)}$ mark the first steps of a tailored matrix-vector product. Fortunately, $\mathbf{C}$ and $\mathbf{M}$ share the same exploitable structure that will allow for less required memory (Chapter 4). Let $\mathbf{y}^{(k)} := \mathbf{Mx}^{(k)}$ and $\mathbf{z}^{(k)} := \mathbf{Cx}^{(k)}$. The next step should then be to compute $\mathbf{Gy}^{(k)}$ (Chapter 5). Combining the results takes one vector subtraction $\mathbf{z}^{(k)} - \mathbf{Gy}^{(k)}$.

Sparse matrix-vector products know many parallelizations, depending on the structure and properties of the matrix. Luckily the matrices $\mathbf{C}$, $\mathbf{G}$ and $\mathbf{M}$ incorporate structures (mentioned below) which can be exploited in parallel algorithm design (Chapter 4 and Chapter 5). Ultimately, these methods will be combined in Chapter 6 to obtain a parallel linear solver, the performance of which will be benchmarked.

2.3.1 Nested Toeplitz block structure

The matrices $\mathbf{C}$ and $\mathbf{M}$ have the following hierarchic structure (block Toeplitz with Toeplitz blocks). With each hierarchic level, indices of the levels above are omitted, for readability purposes.

$$\mathbf{C} = \left(\begin{array}{c|c|c|c|c} \mathbf{B}_0 & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} \\ \hline \mathbf{0} & \mathbf{B}_1 & \mathbf{0} & \dots & \mathbf{0} \\ \hline \mathbf{0} & \mathbf{0} & \mathbf{B}_2 & \dots & \mathbf{0} \\ \hline \vdots & \vdots & \vdots & \ddots & \vdots \\ \hline \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{B}_{m_z-1} \end{array}\right)$$

where each $\mathbf{B}_r$ $(0 \leq r < m_z)$ consists of 3×3 blocks

$$\mathbf{B}_r = \left(\begin{array}{c|c|c} \mathbf{D}_{0,0} & \mathbf{D}_{0,1} & \mathbf{D}_{0,2} \\ \hline \mathbf{D}_{1,0} & \mathbf{D}_{1,1} & \mathbf{D}_{1,2} \\ \hline \mathbf{D}_{2,0} & \mathbf{D}_{2,1} & \mathbf{D}_{2,2} \end{array}\right),$$

and where each $\mathbf{D}_{p,q}$ $(0 \leq p, q < 3)$ is a block Toeplitz matrix of $m_{xy} \times m_{xy}$ Toeplitz blocks $\mathbf{T}_{d_1}$ $(|d_1| < m_{xy})$, each of size $m_{xy} \times m_{xy}$, i.e.,

$$\mathbf{D}_{p,q} = \left(\begin{array}{c|c|c|c|c} \mathbf{T}_0 & \mathbf{T}_1 & \mathbf{T}_2 & \dots & \mathbf{T}_{m_y-1} \\ \hline \mathbf{T}_{-1} & \mathbf{T}_0 & \mathbf{T}_1 & \dots & \mathbf{T}_{m_y-2} \\ \hline \mathbf{T}_{-2} & \mathbf{T}_{-1} & \mathbf{T}_0 & \dots & \mathbf{T}_{m_y-3} \\ \hline \vdots & \vdots & \vdots & \ddots & \vdots \\ \hline \mathbf{T}_{1-m_y} & \mathbf{T}_{2-m_y} & \mathbf{T}_{3-m_y} & \dots & \mathbf{T}_0 \end{array}\right),$$

and

$$\mathbf{T}_{d_1} = \begin{pmatrix} \tau_0 & \tau_1 & \tau_2 & \cdots & \tau_{m_x-1} \\ \tau_{-1} & \tau_0 & \tau_1 & \cdots & \tau_{m_x-2} \\ \tau_{-2} & \tau_{-1} & \tau_0 & \cdots & \tau_{m_x-3} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \tau_{1-m_x} & \tau_{2-m_x} & \tau_{3-m_x} & \cdots & \tau_0 \end{pmatrix},$$

for some $\tau_{d_2} \in \mathbb{C}$ ($|d_2| < m_{xy}$).

Confirm from the structure that $n = 3m_{xy}^2 m_z$. During the time spent at ASML, typical parameter choices $m_{xy} = 7, 9, 11$ and $150 \leq m_z \leq 200$ were encountered. However, by virtue of displaying algorithm employability, higher parameter choices will also be found in this work. Furthermore, one can assume that typically the blocks $\mathbf{B}_r$ are full, implying a constant amount of non-zeros per matrix: $\text{nnz}(\mathbf{C}) = 3nm_{xy}^2 = 9m_{xy}^4 m_z$ and therefore a sparsity $\xi(\mathbf{C}) = \frac{1}{m_z}$. Also, all Toeplitz blocks generally contain different scalars. Note that storing the non-zeros requires

$$N_{\mathbf{CM}} := 16\,\text{nnz}(\mathbf{C}) = 16\,\text{nnz}(\mathbf{M}) = 144m_{xy}^4 m_z = 48nm_{xy}^2$$

bytes of memory when using double precision for both the real and imaginary part.

2.3.2 Diagonal structure

The matrix $\mathbf{G}$ has a less hierarchical structure than matrices $\mathbf{C}$ and $\mathbf{M}$, which has its benefits and drawbacks. It is a matrix where non-zeros only appear on diagonals

$$0, \pm m_{xy}^2, \pm 2m_{xy}^2, \ldots, \pm(3m_z - 1)m_{xy}^2,$$

implying a fixed $\text{nnz}(\mathbf{G}) = 3nm_z = 9m_{xy}^2 m_z^2$ and sparsity $\xi(\mathbf{G}) = \frac{1}{m_{xy}^2}$. Storing all of the non-zeros requires

$$N_{\mathbf{G}} := 16\,\text{nnz}(\mathbf{G}) = 144m_{xy}^2 m_z^2 = 48nm_z$$

bytes of memory. Comparing the sparsity of the two structures yields that $\mathbf{G}$ is less sparse for the typical parameter choices. ASML's desire however to support higher m_{xy} implies that in the future, $\mathbf{G}$ could actually surpass $\mathbf{C}$ and $\mathbf{M}$ in sparsity.

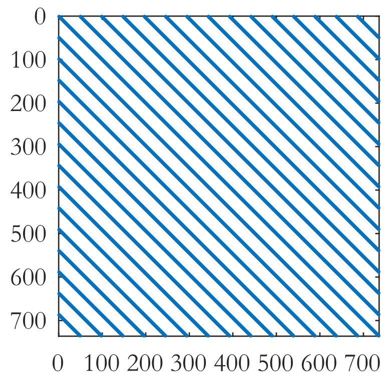


Figure 2.2: Sparsity plot of $\mathbf{G}$ for $m_{xy} = 7$ and $m_z = 5$.

Unfortunately **G** possesses no further properties like diagonal repetition or symmetry, implying every non-zero of **G** is indispensable when storing this matrix. However, the diagonally hatched[1] structure will be used in Chapter 5 to devise a systematic parallel multiplication.

2.3.3 Initial formatting

Sparse matrices are generally large (especially in industrial applications) and by definition contain very few non-zeros. Therefore, sparse matrices are stored differently. One only needs to store the non-zero entries of a sparse matrix, resulting in economical memory usage. There already exist well-known and widely used sparse matrix storage formats, such as the 'Coordinate' (COO), 'Compressed Sparse Column' (CSC) and 'Compressed Sparse Row' (CSR) formats. Each format has its advantages and disadvantages, depending on the operation applied to the matrix.

Since for the matrices **C** and **M** all blocks $\mathbf{B}_r$ are full, the amount of non-zeros in each row and column is constant. Likewise, due to the fact that **G** consists of equidistant non-zero diagonals, also **G** has a constant amount of non-zeros in each row and column. Therefore, one can store the matrices without complementary indices for the position of each non-zero in the matrix. These indices can be computed from the position in the matrix value list. This reduces the total storage space consequently to

$$N_{\text{tot}} := 2N_{\mathbf{CM}} + N_{\mathbf{G}} = 48n(2m_{xy}^2 + mz) \tag{2.19}$$

bytes of memory. The constant number of non-zeros per row or column also enables the matrix to be flattened into a vector, for easier single-index accessing.

Conclude however, that for the current implementation, the desired support for higher m_{xy} is constrained by the memory requirements. Luckily, Chapter 4 will impose a more economical use of memory, while maintaining the ability to define a parallel scalable sparse matrix-vector product.

[1]hatching: artistic technique where the artist draws closely spaced parallel lines to create shading effects

Chapter 3

Hardware-optimized parallel algorithm design

This chapter of the thesis showcases a study of low-level computer layout, i.e., how the different components of hardware came to be and ultimately how these components should be deployed in efficient parallel computations relevant to the research done in the remainder of this thesis. After all, it is this knowledge that ensures the design of algorithms which optimally use the available hardware in a computer system. The first section will contain a brief history [5] such that in the sections that follow, the properties of the various components can be taken into account [5, 16] when designing hardware-efficient algorithms.

The philosophy behind hardware-optimizing an algorithm in contrast to, for example, optimizing the asymptotic running time of an algorithm comes from the fact that computer memory truly constraints the usability of an algorithm. A personal favorite example is the class of matrix-matrix multiplication algorithms. The naive $\mathcal{O}(n^3)$ method, albeit its slow algorithmic running time, requires little more memory over storing the two matrices and their product. However, the Strassen or even the Coppersmith-Winograd multiplications [15] (with asymptotic running times $\mathcal{O}(n^{2.807})$ and $\mathcal{O}(n^{2.376})$, respectively), are so complex and in addition work recursively, resulting in extravagant memory usage when the problem size grows large, rendering them less scalable. Especially now, with the rise of parallel computing, memory management becomes more instrumental than ever before. This implies that algorithms with an execution time of lower orders only exhibit this speed when they do not violate the memory constraints of the particular system in use, i.e., when the problem sizes stay small. It is precisely the memory-economical algorithms that will therefore be more parallel scalable, which is why a study on computer memory is vital to a successful addressing of the problems in this thesis.

3.1 Brief history of CPU system components

Let there be no confusion: CPUs have been and still are essential in modern-day systems, for they have been the main computing units being developed throughout the history of the computer.

GPUs came into existence decades after, initially to deliver a hi-fi gaming experience, and have only recently been discovered to be serviceable for the acceleration of scientific computing. A GPU also cannot host a program by itself. Therefore this section accounts for CPU systems only.

The first computers were developed by a specific institute (company, hospital, etc.) to accommodate one or a couple institute-specific computations only. Accordingly, all components of that computer were in tune with one another, complying to their specific computational and memory constraints. The basic setup consisted a small-sized (hard) disk with data and a CPU to perform basic arithmetic. This arithmetic later got the name FLOPs (floating-point operations). Depending on the application, the machine also could have data input (through sensors or punch cards) and output (card puncher and later: printers) hardware, but for the sake of the computational focus in this section, these are omitted from consideration. A program made the CPU read data from the disk, performed operations on it, and wrote the outcome back to the disk.

With the commercialization of the computer, different companies appeared which focused on the development of particular hardware components only. This resulted in an unbalanced speed-up of the different components. For example: whereas the purely electronic processors became faster over time with higher clock speeds (the rate of the processor indicating how many times per second numbers can be received/sent/performed operations on), the CPU started idling increasingly because of the slow mechanical retrieval system of the hard disk, the speed of which did not grow as fast as that of the CPU.

This marked the dawning era of *Random Access Memory* (RAM): an electronic and therefore faster piece of hardware used to store temporary data. Upon invoking a program, data is loaded into RAM by the operating system, such that the retrieval time by CPUs was greatly curbed, implying a reduced CPU idle time. Presently, RAM can be divided into two categories: dynamic RAM (DRAM) and static RAM (SRAM). Each uses a different electronic component to store data (capacitors and latches respectively). This makes the SRAM transfer speed as fast as the CPU speed, yet its high power consumption and production cost prevent it from being used as the main choice for RAM. DRAM may be less expensive than SRAM, yet it is slower, since capacitors lose their charge after some time, making DRAM a volatile memory medium. Therefore DRAM has charge refresh periods, during which no memory can be read or written.

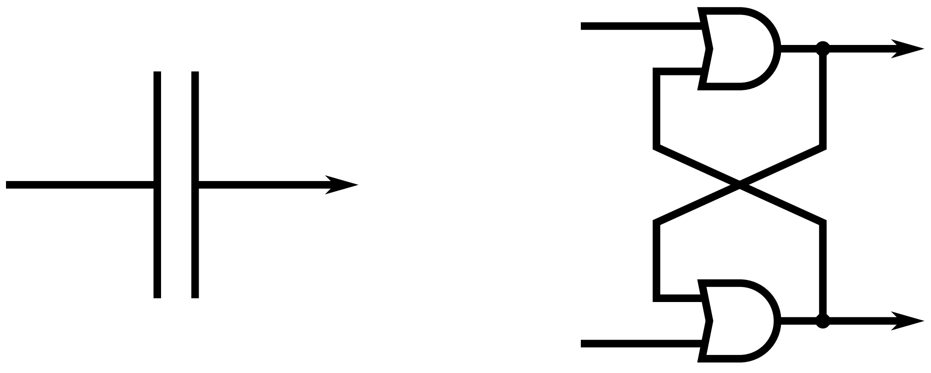


Figure 3.1: Electronic symbols for capacitors (left) and latches (right).

With the ever growing CPU clock speeds, even DRAM fell behind, yet the DRAM still worked orders of magnitude faster than disks. This is the reason computer system architects decided to put a tiny amount of SRAM in the CPU they called *cache*[1], from which data can then be retrieved in a time within clock speed order. A hardware-efficient algorithm should therefore optimally exploit the cache space such that the total retrieval time remains as small as possible. With recursive algorithms, for example those noted in the start of the chapter, this is generally hard for larger problem sizes.

3.2 CPU cores and cache layout

The actual arithmetic brain of a CPU are its cores: they are the components performing the floating point operations. CPUs have had multiple cores for quite some time now (currently, private customer CPUs typically have up to eight cores) and generally differ in micro-structure. Nevertheless the core-cache structure generally resembles Figure 3.2. Note that the cores are not directly connected to RAM ('Main Memory' in the figure) anymore, meaning that data and instructions assigned to a CPU core must pass through its cache. The CPU cache consists of different levels, the size and any inter-connections of which depends on the model and manufacturer.

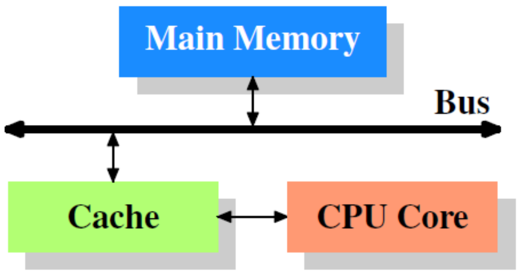


Figure 3.2: Basic cache setup with one core [5].

In the high performing computing cluster used to obtain the benchmarks in this thesis, the methods were ran on one so-called *node*. A node is a machine on its own, housing its own RAM and two *sockets*, where each socket can house one CPU. Usually the two socketed CPUs are of the same manufacturer and model. A computing cluster consists of many of these nodes.

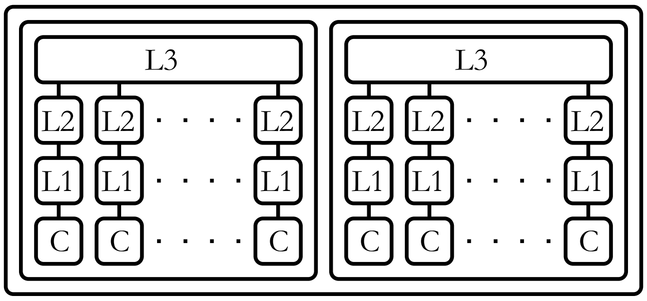


Figure 3.3: Dual-CPU node layout with emphasis on cache layout.

The Intel® Xeon® CPUs used to run the methods have three cache levels (Figure 3.3): RAM data

[1]From the French verb 'cacher', meaning 'to hide'.

enters the socket and is loaded into the L3 caches of both CPUs, which is the highest cache level. This cache level is physically the furthest from the core (C) and is used by all cores in one socket. In addition, each CPU loads data further down into the two core-local cache levels, the lowest of which (L1) is used to perform FLOPs. Computational results of all cores are collected in L3 again, before they are written back to main memory. Since this thesis accounts almost entirely for single-node implementations only, this implies a *shared memory parallel* paradigm. The nodes have a *Non-uniform Memory Access* (NUMA) design, meaning cores can access their local (L1 and L2) much faster than L3 or local caches of other cores. Naturally, having to access data from another socket is even more expensive.

This paradigm will in no way impede the design of a cache-optimized algorithm, since the operations required in this thesis (Chapter 4 and Chapter 5) can be partitioned into independent work to be divided over cores. This marks a fruitful start to any parallel program.

3.2.1 Cache loading, pre-fetching and cache hits/misses

The cache loads are performed by a component in the CPU (omitted in the previous pictures) called the *memory controller*. This is a rather complex CPU component. Its exact functioning will therefore not be treated in this thesis. However, the purpose of this component can be generically described as "getting the needed data from RAM and copying shared memory data further down into the lower levels". It is exactly that component which fills the caches.

As one can understand, retrieving data upon CPU request will result in a highly idle CPU since, for each computation, the memory controller will have to get the data for each core, lasting prohibitively many clock cycles if RAM access is necessary. Therefore the memory controller performs some analysis on the previously used memory addresses to make an educated guess on what the next requested addresses will be and caches this data before the CPU is ready to perform FLOPs thereon. Therefore, given that the memory controller guesses the needs of the cores correctly, the CPU will not have to idle. This is called *pre-fetching* and it greatly increased CPU efficiency.

Let it be evident that pre-fetching comes with a risk. After all, when a core is searching for its needed data, ideally this data would already be cached in L1, thanks to the clever memory controller. This is called a *cache hit*, i.e., the data is fetched from cache when the data is needed. However, as the randomness of the accessed addresses grows, the probability of fetching unneeded data increases likewise. Consequently, it occasionally occurs that the cache will not have required data readily available. The CPU will therefore try fetching the data from a higher cache level, during which the CPU is not performing operations. In the case the data is not present there as well, it has to overwrite one or more cache registers with the correct data it will have to get from RAM. In addition to being very expensive in terms of clock cycle count, there also exists a probability of overwriting data required in the clock cycles to come, and is therefore a worst case scenario called a *cache miss*.

This information on CPU architecture will prove crucial in the design of the parallel algorithms in this thesis, for a programmer needs to arrange the data for each thread such that the accessing pattern will be as predictable as possibly achievable and is therefore the very message of this chapter. The most canonical example would be to order the data such that the consequent memory addresses follow a linear pattern, i.e., the needed data is stored in a (periodically) contiguous region in memory.

An algorithm can then address this data with unit stride. The algorithms designed in this thesis will therefore try to realize these linear access patterns.

3.3 Shared memory parallel algorithms with OpenMP

This thesis will focus on a dual-CPU machine as computational resource. Since the resource is considered a single machine, the OpenMP Application Programming Interface [9] satisfies to program parallel algorithms with, which is also used in the PARALUTION library subroutines. With the free version used in this thesis, single-node computations can be performed. With the paid multi-node version, a Message Passing Interface (MPI) communication layer is added [11].

In sequential computing, execution of code is performed by a single *thread*: a sequence of programmed instructions. However, in a parallel program, *parallel regions* may exist in code [9]. These are parts of computer code which can be performed concurrently by multiple threads. When the single *master thread* encounters a parallel region, it creates a *team* of other threads, after which the team works on the execution of that parallel region until the computation is completed. Each thread of a parallel program is bound to one of the processor cores. Often, cores can only harbor one thread. Even though CPUs capable of hosting two threads exist, a core can generally host only one. In this thesis, only one thread per core is assumed. It implies the existence of a bijection between the IDs of the used cores and thread IDs. The number of threads the master thread is allowed to create and their binding can be controlled. Naturally no more threads than cores can exist.

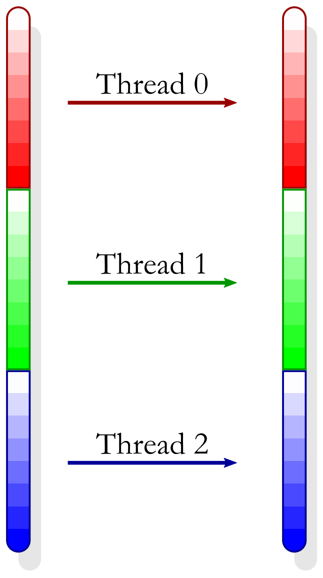


Figure 3.4: Three threads (red, green and blue) copying a vector: a simple parallel linear algebra subroutine.

3.3.1 Parallelizing loops

The parts of an algorithm to be parallelized in this thesis will solely be (nested) FOR-loops. Many basic linear algebra subroutines, e.g. vector addition and the inner product of two vectors, are essentially parallelizable FOR-loops. Although distributed memory methods with MPI are described in [2], the concept of work division remains the same and can be applied to divide the computations over the available threads.

When executing a single FOR-loop in parallel with indices from an integer set, i.e. $i \in I \subset \mathbb{Z}$, each thread gets assigned a unique i after which it performs the contents of the loop for that particular i. If for example $I = \{0, 1, 2\}$ and the amount of threads equals 2, the loop is iterated twice: once where the two threads are passed $i = 0$ and $i = 1$, and once where one thread executes for $i = 2$. The process of reiteration is called a *barrier*: finished threads wait until the entire team has finished, after which they get passed a new index each. Let p be the team size. Then, by the pigeonhole principle, the amount of barriers encountered is $\lceil \frac{|I|}{p} \rceil$, where $|\cdot|$ denotes the cardinality of a set and $\lceil \cdot \rceil$ denotes the ceiling function. In order for a FOR-loop to be parallel efficient, the work must be of equal size and the work must be thread-independent, i.e. the variables and outcome used must not depend on the variables or outcome of other threads.

Now assume a parallel region consisting of m FOR-loops nested in each other, each with their respective index sets $I_j \subset \mathbb{Z}$ for $j = 1, 2 \ldots, m$. With no additional directives, only the outermost loop is parallelized. This means that a thread will get one $i_1 \in I_1$ and will therefore execute the content inside the outermost loop by itself, which contains the $m-1$ innermost loops. By using the OpenMP `collapse(m)` clause [9, 10], one indicates the amount of nested loops to be combined in parallel execution, which results in a collapsed iteration space $I := \prod_{j \in \{1,2,\ldots,m\}} I_j$. The thread then gets passed m indices: $i_1 \in I_1, i_2 \in I_2, \ldots i_m \in I_m$, corresponding to exactly one element of $I = \prod_{j \in \{1,2,\ldots,m\}} I_j$, which can now be considered as a flattened iteration space (Figure 3.5, right). This is desirable over only parallelizing a single loop, because of thread efficiency.

Let for example $I_1 = I_2 = \{0, 1, 2\}$ and let only the inner loop I_2 be parallelized. Then, firstly, for each $i_1 \in I_1$ the single thread summoned at the start of the program (also known as the *master thread*) would encounter a parallel FOR-loop, implying teams are created redundantly many times. Moreover, there would only be a thread efficiency of $\frac{3}{4}$ (analogous to the paragraph above). If only the outer loop I_1 is parallelized (Figure 3.5, left), there would be less barriers in the execution, yet there will be a larger variance in the thread finishing times between those barriers, because even if two threads have the same set of instructions between two barriers, there will always be a small difference in the time they take. Also, again by the pigeonhole principle, there is a risk that only a few threads are active to complete an iteration space, in which case other threads will idle for a longer time.

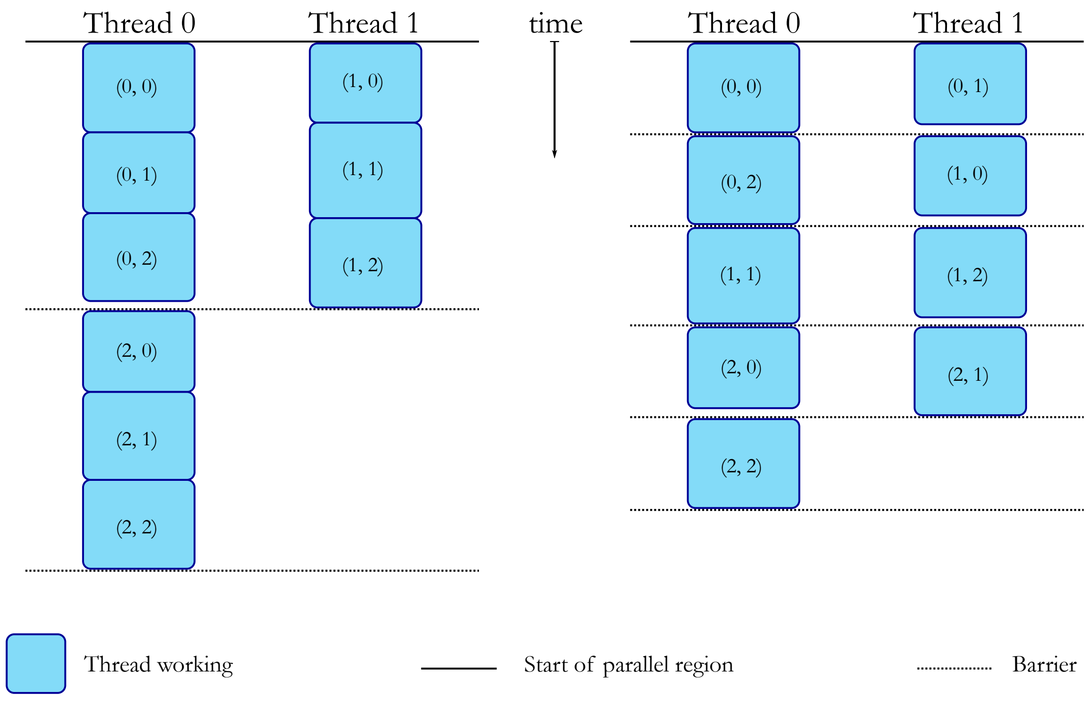


Figure 3.5: OpenMP work scheduling behavior (two threads) for the regular (left) and collapsed (right) iteration space $I = \{0, 1, 2\}^2$. Pairs (i_1, i_2) are inscribed.

Conclude that there is a choice to parallelize only a part of a program's nested loops. This choice determines the *granularity* of a parallel program. This can be coarse (less barriers but also larger variance) or fine (more barriers but higher thread efficiency).

3.3.2 Synchronization overhead

In parallel operations where thread results need to be shared, very costly *synchronization overhead* occurs: threads will try to output their results to the same memory address. Parallel vector norm calculation is an example of this. If the output is a shared number (hence in L3), all threads will pose an increment on its address, e.g. when calculating the norm of vector $\mathbf{v}$ in parallel, a thread with an assigned i will impose an increment of $(v_i)^2$ on the variable. Since these have to be processed sequentially for all $i \in I$, this could gravely slow the computation speed.

However, the overhead can be reduced by creating a thread-local subtotal which each thread can increment privately. This is achieved with the OpenMP `reduction` clause [9, 10]. Ultimately, these subtotals can be added, which costs only an amount of sequential increments equal to the amount of threads p, which is a mere fraction compared to $|I|$, especially when the iteration spaces grow large. Any other variable can be made thread-local by adding it to the `private` list [9, 10]. Iteration space indices are automatically private.

Since synchronization overhead is very costly, it is imperative to design parallel algorithms where threads work on mutually exclusive output addresses as much as possible.

3.3.3 Concurrent parallel programs

OpenMP also allows for concurrent parallel programs, i.e. multiple master threads, each with a respective program, each probably also with their own parallel regions. This enables a programmer to, for example, let half of the cores calculate a vector norm and the other half calculate the sum of two other vectors.

A direct implication of this kind of parallelism is a change in the cache size and amount of threads available per program. Even when executing multiple instances of the same program concurrently, each with the same number of threads, the cache memory will have to be divided. Each partition will then contain a set the program data and instructions. Yet, since two instances of the same program are executed, potentially many global program variables and many program instructions will be the same, implying inefficient use of precious cache memory.

An advantage to use this kind of parallelism however, is to improve the locality of each instance's data. For example, since the computational resource is a dual-CPU machine (each with an L3 cache), it could be desirable to let each CPU work on its own computation, which can be achieved by using correct thread bindings. This way, expensive cross-CPU data loading is eliminated by design. Therefore, the SpMV algorithms and ultimately the parallel solver presented in the next three chapters will be benchmarked on one of the two CPUs in the machine used as computational resource, such that two linear solves can be performed concurrently: each on one CPU.

Chapter 4

SpMVs with nested Toeplitz blocks

This thesis chapter will account for the design of a custom multiplication algorithm which runs in parallel as well as optimizes memory usage, to reduce computation time per iteration. In order to address this algorithm design problem, not only a study of the application's matrices **C** and **M**, but also the research on hardware layout (Chapter 3) and parallel algorithm design is required to gain computation speed.

The shared memory parallel guidelines (Section 3.3) imply a choice for independent thread output regions, which in case of SpMVs essentially hints to algorithms where each entry of the output vector is computed by a single thread, rendering the CSR matrix format a solid base for the proposed algorithms. The algorithms will be tested and benchmarked on ASML's High Performance Computing (HPC) cluster. Then, given various benchmarks of the different approaches, insights will be gained on how the granularity and memory influence the performance.

4.1 Method of circular convolution

It is possible to multiply every block $\mathbf{D}_{p,q}$ (sized $m_{xy}^2 \times m_{xy}^2$) with the corresponding vector part of $\mathbf{x}$ of size m_{xy}^2 using the Fourier Transform. Let, due to implementation syntax, vector indices henceforth start at 0.

Definition 4.1. *The Fourier Transform of a vector* $\mathbf{a} \in \mathbb{C}^n$ *and a matrix* $\mathbf{A} \in \mathbb{C}^{n\times n}$ *are given by* $\mathcal{F}(\mathbf{a}) \in \mathbb{C}^n$ *and* $\mathcal{F}(\mathbf{A}) \in \mathbb{C}^{n\times n}$ *such that:*

$$\big(\mathcal{F}(\mathbf{a})\big)_k = \sum_{p=0}^{n-1} a_p w_n^{kp}$$

$$\big(\mathcal{F}(\mathbf{A})\big)_{kl} = \sum_{p,q=0}^{n-1} a_{pq} w_n^{kp+lq}$$

for $k, l = 0, 1, 2, \ldots, n-1$ *and where* $w_n = e^{-\frac{2i\pi}{n}}$ *denotes the primitive* n^{th} *root of unity.*

Definition 4.2. *Let two vectors* $\mathbf{a}, \mathbf{b} \in \mathbb{C}^n$ *be given. Then the circular convolution* $\mathbf{a} * \mathbf{b} \in \mathbb{C}^n$ *and element-wise product* $\mathbf{a} \odot \mathbf{b} \in \mathbb{C}^n$ *are defined as*

$$\begin{aligned}(\mathbf{a} * \mathbf{b})_i &= \sum_{j=0}^{n-1} a_{i-j} b_j, \\ (\mathbf{a} \odot \mathbf{b})_i &= a_i b_i,\end{aligned}$$

where $i, j \in [0, n) := \{0, 1, \ldots, n-1\}$ *and where* $i - j$ *circulates in the same integer interval.*

The product with a block $\mathbf{D}_{pq}$ can be computed by means of a Fourier Transform because of the Circular Convolution Theorem [3]:

Theorem 4.1. *Let* $\mathcal{F}$ *denote the Fourier transform and let two vectors* $\mathbf{a}$ *and* $\mathbf{b}$ *of equal size be given. Then*

$$\mathcal{F}(\mathbf{a} * \mathbf{b}) = \mathcal{F}(\mathbf{a}) \odot \mathcal{F}(\mathbf{b}). \tag{4.1}$$

This theorem forms the foundation of the currently implemented SpMV algorithm.

4.1.1 The Fast Fourier Transform

One could expect that applying $\mathcal{F}$ is a very costly computation. In modern implementations, however, $\mathcal{F}$ is applied quickly by using Fast Fourier Transform (FFT) algorithms, which run in $\mathcal{O}(m \log m)$ asymptotic time on vectors in $\mathbb{C}^m$ or matrices in $\mathbb{C}^{m \times m}$. The inverse matrix application is implemented as 'Inverse Fast Fourier Transform' (IFFT). Different variants and implementations of FFT exist, yet the FFTW [7] for the C programming language is a most extensive collection used in industry.

The FFT algorithms come with a caveat: they are designed to run faster when the problem size is smooth. Recall that a natural number is d-smooth if and only if d is its largest prime divisor. This effectively implies that the input of FFT, which is a block of **C**, needs to be padded with extra zeros until the problem size is of a desired smoothness. Let therefore n_d be the smallest natural number such that $2m_{xy} - 1 \leq n_d$ and n_d is d-smooth. Therefore the padded input size of the FFT algorithm is n_d^2 and thus a total running time of $\mathcal{O}(n_d^2 \log n_d)$ per block $\mathbf{D}_{pq}$. Conclude that the running time totals to $\mathcal{O}(m_z n_d^2 \log n_d)$ for an entire matrix **C** or **M**.

Moreover, FFT works recursively. Since recursive algorithms generally do not scale well in parallel, the parallel FFT algorithms on matrices are still being developed and improved [7]. This method is therefore not benchmarked in this work.

4.1.2 Relation to Toeplitz blocks

Consider a Toeplitz block **T** containing scalars τ_{d_2}, with $|d_2| < m_{xy}$. Let $m = 2m_{xy} - 1$. A Toeplitz matrix can be expanded into a circulant matrix **S** of size $m \times m$. The circular matrices form a class of special Toeplitz matrices, for they are Toeplitz matrices where each row is a circular right-shift of the row above. The unfolding can be done as follows:

$$\mathbf{S} = \left(\begin{array}{cccc|cccc}
\tau_0 & \tau_1 & \cdots & \tau_{m_{xy}-1} & \tau_{1-m_{xy}} & \tau_{2-m_{xy}} & \cdots & \tau_{-1} \\
\tau_{-1} & \tau_0 & \ddots & \vdots & \tau_{m_{xy}-1} & \tau_{1-m_{xy}} & \cdots & \tau_{-2} \\
\vdots & \ddots & \ddots & \tau_1 & \vdots & \vdots & & \vdots \\
\tau_{1-m_{xy}} & \cdots & \tau_{-1} & \tau_0 & \tau_1 & \tau_2 & \cdots & \tau_{m_{xy}-1} \\
\hline
\tau_{m_{xy}-1} & \tau_{1-m_{xy}} & \cdots & \tau_{-1} & \tau_0 & \tau_1 & \cdots & \tau_{m_{xy}-2} \\
\tau_{m_{xy}-2} & \tau_{m_{xy}-1} & \cdots & \tau_{-2} & \tau_{-1} & \tau_0 & \ddots & \vdots \\
\vdots & \vdots & & \vdots & \vdots & \ddots & \ddots & \tau_1 \\
\tau_1 & \tau_2 & \cdots & \tau_{1-m_{xy}} & \tau_{2-m_{xy}} & \cdots & \tau_{-1} & \tau_0
\end{array}\right).$$

Note that $\mathbf{S}$ is circulant, has the aforementioned size and that its upper left $m_{xy} \times m_{xy}$ block equals $\mathbf{T}$. Let

$$\mathbf{r} = (\tau_0, \tau_{-1}, \ldots, \tau_{1-m_{xy}}, \tau_{m_{xy}-1}, \tau_{m_{xy}-2}, \ldots, \tau_1)^T \in \mathbb{C}^m, \tag{4.2}$$

be the first column of $\mathbf{S}$.

Definition 4.3. *A matrix* $\mathbf{A}$ *is diagonalizable if and only if there exist an invertible matrix* $\mathbf{P}$ *such that* $\mathbf{PAP}^{-1}$ *is a diagonal matrix.*

The relation with the Fourier Transform now becomes apparent, for the circulant $\mathbf{S}$ is diagonalized by the unitary Discrete Fourier Transform (DFT) matrix $\mathbf{F}_m$ [6]. Applying this matrix is equivalent with using the Fourier Transform. Hence, $\mathbf{S}_{d_1}$ can be written as

$$\mathbf{S} = \mathbf{F}_m^{-1} \operatorname{diag}(\mathbf{F}_m \mathbf{r}) \mathbf{F}_m, \tag{4.3}$$

with

$$\mathbf{F}_m = \frac{1}{\sqrt{m}} \begin{pmatrix}
1 & 1 & 1 & \cdots & 1 \\
1 & w_m & w_m^2 & \cdots & w_m^{m-1} \\
1 & w_m^2 & w_m^4 & \cdots & w_m^{2(m-1)} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
1 & w_m^{m-1} & w_m^{2(m-1)} & \cdots & w_m^{(m-1)(m-1)}
\end{pmatrix}. \tag{4.4}$$

Following from the unitary property of the DFT matrix, conclude that its inverse equals $\mathbf{F}_m^{-1} = \mathbf{F}_m^H$: its conjugate transpose.

Since applying the diagonal matrix $\operatorname{diag}(\mathbf{F}_m \mathbf{r})$ is equivalent to an element-wise product with its diagonal vector, which is $\mathbf{F}_m \mathbf{r}$, the circulant matrix-vector product effectively becomes

$$\mathbf{Su} = \mathbf{F}_m^{-1} \operatorname{diag}(\mathbf{F}_m \mathbf{r}) \mathbf{F}_m \mathbf{u} = \mathcal{F}^{-1}\big(\mathcal{F}(\mathbf{r}) \odot \mathcal{F}(\mathbf{u})\big). \tag{4.5}$$

Using the circulant property, note that $s_{ij} = r_{i-j}$, such that the matrix-vector product becomes

$$(\mathbf{Su})_i = \sum_{j=0}^{m-1} s_{ij} u_j = \sum_{j=0}^{m-1} r_{i-j} u_j, \tag{4.6}$$

where $i-j$ circulates in $[0, m)$. Conclude that the circulant matrix-vector product has now become a circular convolution $\mathbf{r} * \mathbf{u}$. Furthermore, combine (4.5) and (4.6) to confirm (4.1) and thereby the Circular Convolution Theorem.

This convolution method accounts for the multiplication with a circulant matrix. To ensure the product with $\mathbf{T}$ only, one should pick $\mathbf{u}$ such that the first m_{xy} entries match the corresponding vector part of the iterate $\mathbf{x}^{(k)}$ and the other entries are 0. The output will be a vector of length m, yet one can select the first m_{xy} entries to serve as output. This sub-vector will then equal the multiplication with $\mathbf{T}$.

This technique can be applied to obtain the products with all the Toeplitz blocks $\mathbf{T}_{d_1}$ contained in $\mathbf{C}$ or $\mathbf{M}$.

Example 4.1. *Let*

$$\mathbf{T}_{d_1} = \begin{pmatrix} 1 & 3 \\ 2 & 1 \end{pmatrix}, \mathbf{x} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}.$$

Then $\mathbf{r} = (1, 2, 3)^T$ *and choose* $\mathbf{u} = (1, 2, 0)^T$*, such that*

$$\mathbf{T}_{d_1}\mathbf{x} = \begin{pmatrix} 7 \\ 4 \end{pmatrix},$$

$$\mathcal{F}^{-1}\big(\mathcal{F}(\mathbf{r}) \odot \mathcal{F}(\mathbf{u})\big) = \begin{pmatrix} 7 \\ 4 \\ 7 \end{pmatrix}.$$

Here, the two obtained vectors match entries, marked in red.

4.1.3 Extending to nested Toeplitz blocks

This method can be extended to multiply an entire block $\mathbf{D}_{pq}$ (each harboring Toeplitz blocks $\mathbf{T}_{d_1}$) with a vector $\mathbf{x} \in \mathbb{C}^{m_{xy}^2}$. In [6], the extension is made to a matrix $\mathbf{N}$ which is a block circulant matrix with circulant blocks, i.e.,

$$\mathbf{N} = \left(\begin{array}{cccc|cccc}
\mathbf{S}_0 & \mathbf{S}_1 & \cdots & \mathbf{S}_{m_{xy}-1} & \mathbf{S}_{1-m_{xy}} & \mathbf{S}_{2-m_{xy}} & \cdots & \mathbf{S}_{-1} \\
\mathbf{S}_{-1} & \mathbf{S}_0 & \ddots & \vdots & \mathbf{S}_{m_{xy}-1} & \mathbf{S}_{1-m_{xy}} & \cdots & \mathbf{S}_{-2} \\
\vdots & \ddots & \ddots & \mathbf{S}_1 & \vdots & \vdots & & \vdots \\
\mathbf{S}_{1-m_{xy}} & \cdots & \mathbf{S}_{-1} & \mathbf{S}_0 & \mathbf{S}_1 & \mathbf{S}_2 & \cdots & \mathbf{S}_{m_{xy}-1} \\
\hline
\mathbf{S}_{m_{xy}-1} & \mathbf{S}_{1-m_{xy}} & \cdots & \mathbf{S}_{-1} & \mathbf{S}_0 & \mathbf{S}_1 & \cdots & \mathbf{S}_{m_{xy}-2} \\
\mathbf{S}_{m_{xy}-2} & \mathbf{S}_{m_{xy}-1} & \cdots & \mathbf{S}_{-2} & \mathbf{S}_{-1} & \mathbf{S}_0 & \ddots & \vdots \\
\vdots & \vdots & & \vdots & \vdots & \ddots & \ddots & \mathbf{S}_1 \\
\mathbf{S}_1 & \mathbf{S}_2 & \cdots & \mathbf{S}_{1-m_{xy}} & \mathbf{S}_{2-m_{xy}} & \cdots & \mathbf{S}_{-1} & \mathbf{S}_0
\end{array}\right),$$

where the $\mathbf{S}_{d_1}$ are circulant matrices. To multiply this matrix with a vector $\mathbf{u}$ of size m^2, one has to use the first column $\mathbf{r}$ of each circulant block $\mathbf{S}$, similar to the previous section. Let therefore $\mathbf{R} \in \mathbb{C}^{m \times m}$ be the matrix consisting of the first columns, i.e.,

$$\mathbf{R} = [\mathbf{r}_0, \mathbf{r}_{-1}, \mathbf{r}_{-2}, \ldots, \mathbf{r}_{1-m_{xy}}, \mathbf{r}_{1-m_{xy}}, \ldots, \mathbf{r}_1].$$

Let also $\mathbf{U} \in \mathbb{C}^{m \times m}$ be given by resizing $\mathbf{x}$ into a matrix, such that for all $i, j \in [0, m_{xy})$, u_{ij} equals entry i of the part of $\mathbf{x}$ corresponding to the $\mathbf{r}_{d_1}$ listed as column j of $\mathbf{R}$. Note that the bottommost $m_{xy} - 1$ rows and rightmost $m_{xy} - 1$ columns of $\mathbf{U}$ contain only zeros.

Ultimately, identical to the one-dimensional Fourier Transform multiplication method, applying $\mathbf{N}$ to a vector $\mathbf{x}$ is then equivalent to computing

$$\mathcal{F}^{-1}\big(\mathcal{F}(\mathbf{R}) \odot \mathcal{F}(\mathbf{U})\big), \tag{4.7}$$

yet only the upper-left $m_{xy} \times m_{xy}$ sub-matrix of the output should be considered, which needs to be flattened back into a column vector.

To obtain the multiplication with only the embedded Toeplitz blocks, one has to choose $\mathbf{u}$ wisely, similarly to the one-dimensional case. Therefore, again pick $\mathbf{U}$ to contain a repetition of: m_{xy} entries from the iterate $\mathbf{x}^{(k)}$, followed by $m_{xy} - 1$ zeros. This implies that $\mathbf{U}$ is a matrix with only the upper-left $m_{xy} \times m_{xy}$ sub-matrix containing scalars from $\mathbf{x}^{(k)}$.

Example 4.2. *Let*

$$\mathbf{D}_{p,q} = \left(\begin{array}{cc|cc} 1 & 3 & 7 & 9 \\ 2 & 1 & 8 & 7 \\ \hline 4 & 6 & 1 & 3 \\ 5 & 4 & 2 & 1 \end{array}\right), \mathbf{x} = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix}.$$

Then

$$\mathbf{R} = \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix},$$

and choose

$$\mathbf{U} = \begin{pmatrix} 1 & 3 & 0 \\ 2 & 4 & 0 \\ 0 & 0 & 0 \end{pmatrix},$$

such that

$$\mathbf{D}_{p,q}\mathbf{x} = \begin{pmatrix} 64 \\ 56 \\ 31 \\ 23 \end{pmatrix},$$

$$\mathcal{F}^{-1}\big(\mathcal{F}(\mathbf{R}) \odot \mathcal{F}(\mathbf{U})\big) = \begin{pmatrix} 64 & 31 & 61 \\ 56 & 23 & 53 \\ 66 & 33 & 63 \end{pmatrix}.$$

4.2 Parallel CSR algorithm

As stated at the beginning of this chapter, the CSR format complies well with the algorithm design guidelines. Since the index lists for all matrices in this work can be omitted (Section 2.3) and $\mathbf{C}$ and $\mathbf{M}$ are essentially block-diagonal matrices, a row-parallel algorithm can be set-up rather quickly.

In order to systematically count all non-zeros of **C** or **M**, one should consider the matrix structure. From counting systematically, a collapsible iteration space (Section 3.3.1) and therefore a parallelization can be found. After all, when looking at Figure 4.1, any non-zero of **C** and **M** can be pointed to by choosing a block $\mathbf{B}_r$ and a row and column inside it. Since each $\mathbf{B}_r$ is of size $3m_{xy}^2 \times 3m_{xy}^2$, deduce that the matrix position (i, j) of any non-zero can be uniquely written as

$$i = 3m_{xy}^2 r + i', \tag{4.8}$$

$$j = 3m_{xy}^2 r + j', \tag{4.9}$$

with $i', j' \in [0, 3m_{xy}^2)$. This yields a collapsible iteration space $I := [0, m_z) \times [0, 3m_{xy}^2)$ (Section 3.3.1) such that each matrix row i is a unique element of I. A thread on row i, corresponding to an element $(r, i') \in I$, then needs to perform multiplications with the non-zeros in that row, i.e. computing

$$y_i = \sum_{j \in [3m_{xy}^2 r, 3m_{xy}^2 (r+1))} c_{ij} x_j .$$

Since the thread already gets passed r, letting j' go from 0 through $3m_{xy}^2$ satisfies.

This results in a regular CSR algorithm, which will henceforth be referred to as CMPCSR (**C** *or* **M** **P***arallel algorithm using the* **CSR** *format.*).

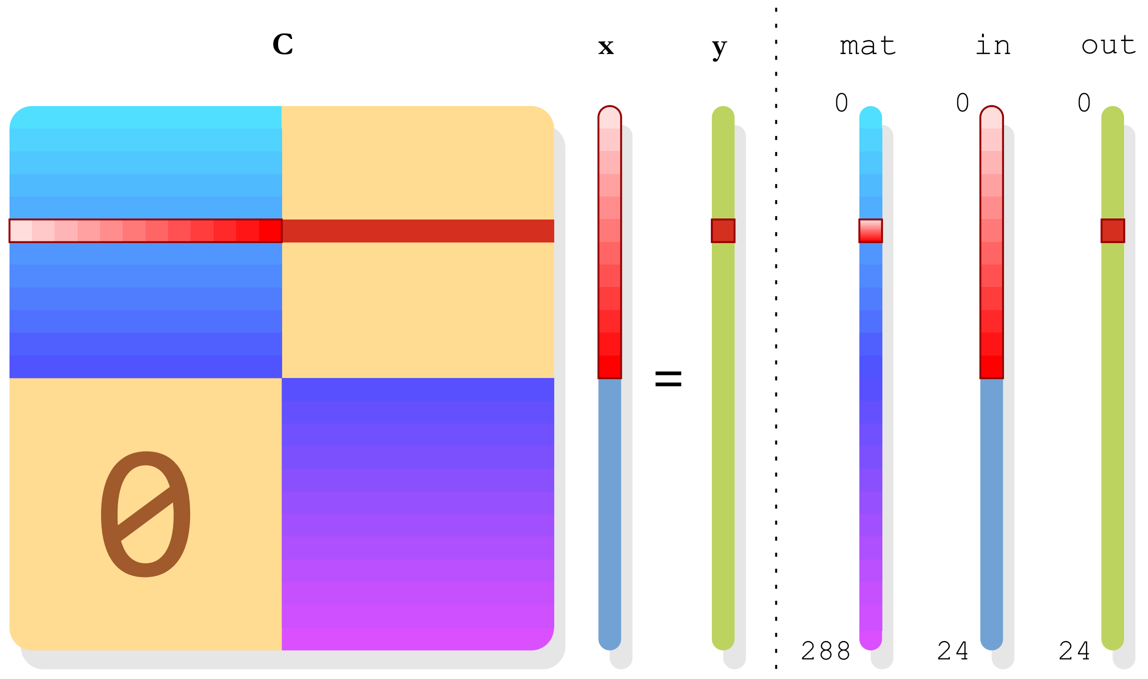


Figure 4.1: Mathematical (left) and digital (right) representation of the accesses of one thread (red) for $m_{xy} = m_z = 2$.

In the figure, the blue-to-purple gradient indicates where the non-zeros appear in the stored matrix list `mat`. The white-to-red gradient overlay on `mat` and `in` indicates the order one particular thread accesses the matrix and input vector data. These gradients are also found in the mathematical representation on the left. Note that the access patterns are linear and contiguous for both the matrix data and input vector. Therefore, no further improvements on this CSR-based algorithm are expected to be made.

4.3 Methods using Toeplitz diagonal compression

The nested Toeplitz structure of **C** and **M** implies the essential information of both matrices is many times smaller than when stored regularly, as already seen with and used by the FFT multiplication methods. An attempt to systematically gather this smaller sized essential data might result in considerably better data accessing rates.

When blocks $\mathbf{D}_{p,q}$ are stored in regular sparse formats, this results in m_{xy} copies of $\mathbf{T}_0$, $m_{xy}-1$ copies of both $\mathbf{T}_1$ and $\mathbf{T}_{-1}$, and so forth. This is most wasteful when m_{xy} grows large, implying the over-abundant use of memory. To avoid this, a tailored SpMV algorithm should use only the essential matrix info. Recalling the matrix structure in Section 2.3.1, note that every generally unique entry can be indexed by its specific block $\mathbf{B}_r$, its embedded blocks $\mathbf{D}_{p,q}$, each containing diagonally repeated block $\mathbf{T}_{d_1}$ and ultimately diagonally repeated scalars τ_{d_2}. After all, the MATLAB command for the construction of a Toeplitz matrix requires only the first row and column as argument. This suggests a compressed format, where only one copy of the diagonally repeated value is stored.

Whereas in the standard formats, indices $i, j \in [0, n)$ are accompanying the values v, matrices **C** and **M** can be indexed using $r \in [0, m_z)$, $p, q \in [0, 3)$ and $|d_1|, |d_2| < m_{xy}$. Before multiplications in this format can be defined, relations between regular indices (i, j) and the novel indices are first required. Additionally an index ordering needs to be chosen in order to completely define the matrix format.

4.3.1 Index set relations

Fortunately, it is possible to relate matrix row i and column j to a value in the compressed format and vice versa.

First, let it be evident that

$$j = 3m_{xy}^2 r + m_{xy}^2 q + m_{xy} c_1 + c_2, \tag{4.10}$$

$$i = 3m_{xy}^2 r + m_{xy}^2 p + m_{xy} r_1 + r_2, \tag{4.11}$$

where $c_1, c_2 \in [0, m_{xy})$ respectively select a column of **T**-blocks within a particular $\mathbf{D}_{p,q}$ and a scalar column therein. Analogously, let $r_1, r_2 \in [0, m_{xy})$ select a block and scalar row.

Secondly, use the "diagonal equals column minus row" principle to deduce that the diagonal repetition of a value appears on one global diagonal only. By (4.10) and (4.11),

$$j - i = m_{xy}^2(q-p) + m_{xy}(c_1 - r_1) + (c_2 - r_2) = m_{xy}^2(q-p) + m_{xy} d_1 + d_2. \tag{4.12}$$

Note that $(\mathbf{T}_{d_1})_{r_2,0} = \tau_{-d_2}$ and $(\mathbf{T}_{d_1})_{0,c_2} = \tau_{d_2}$. Since $\mathbf{T}_{d_1}$ has size $m_{xy} \times m_{xy}$, deduce that scalar or block d_l can be found in the following (block) rows and columns:

$$c_l - m_{xy} < d_l \leq c_l, \tag{4.13}$$

$$-r_l \leq d_l < m_{xy} - r_l, \tag{4.14}$$

for $l = 1, 2$.

Recalling the desire to order the data in such a way that linear contiguous access patterns are achieved, such an ordering must now be found. Naturally, one may assume vector entries to already occupy a linear contiguous memory region. In addition, to ensure the essential (periodically) linear addressing scheme, the values in the compressed format need an ordering to cluster consecutively needed positions. For example, non-zeros in the CSR format are, in ascending order, sorted on row and then on column . Generally, linear or contiguous memory access patterns are achieved most when sorting the values from coarse towards fine indices.

This suggests sorting on r, followed by p. Considering the coarse towards fine suggestion, the sorting should be continued with q, d_1 and d_2 (Section 4.3.3). Some improvements in the ordering of matrix values (4.3.4) and input vector (4.3.5) will be made to optimize the access patterns in both.

Refer to Section 2.3 to recall that all blocks $\mathbf{B}_r$ are full. Following the idea of Section 2.3.3, conclude that each combination of r, p, q, d_1 and d_2 defines the position of a non-zero in the stored matrix `mat` uniquely. Accordingly, each combination of indices can be completely reconstructed from the position in the matrix value vector, given a freshly obtained ordering.

4.3.2 Memory requirement analysis

Performing an analysis on the amount of memory saved by opting for the aforementioned format is not a laborious process. After all, every combination of these novel indices points to a non-zero, implying `mat` has length $9m_z(2m_{xy} - 1)^2$, with the corresponding memory requirement of

$$N'_{\mathbf{CM}} := 144m_z(2m_{xy} - 1)^2 \tag{4.15}$$

bytes for complex values with double precision for both the real and imaginary part.

Contrasting (4.15) with the current employment storage scheme (2.19) results in a memory deployment efficiency of

$$\eta_{m_{xy}} = \frac{N'_{\mathbf{CM}}}{N_{\mathbf{CM}}} = \left(\frac{2m_{xy} - 1}{m_{xy}^2}\right)^2 \leq \frac{4}{m_{xy}^2},$$

making this format increasingly relatively economical when m_{xy} grows large. The total storage cost then sums up to to

$$N'_{\text{tot}} := 2N'_{\mathbf{CM}} + N_{\mathbf{G}} = 2\eta_{m_{xy}} N_{\mathbf{CM}} + N_{\mathbf{G}} = 48n(2\eta_{m_{xy}} m_{xy}^2 m_z) \leq 48n(8 + m_z).$$

		m_z				
		150	175	200	225	250
m_{xy}	7	49.46 MiB	57.70 MiB	65.95 MiB	74.19 MiB	82.43 MiB
	$\eta_7 \approx 0.0704$	3.48 MiB	4.06 MiB	4.64 MiB	5.22 MiB	5.80 MiB
	9	135.15 MiB	157.68 MiB	180.20 MiB	202.73 MiB	225.25 MiB
	$\eta_9 \approx 0.0440$	5.95 MiB	6.95 MiB	7.94 MiB	8.93 MiB	9.92 MiB
	11	301.60 MiB	351.86 MiB	402.13 MiB	452.39 MiB	502.66 MiB
	$\eta_{11} \approx 0.0301$	9.08 MiB	10.60 MiB	12.11 MiB	13.63 MiB	15.14 MiB
	13	588.34 MiB	686.39 MiB	784.45 MiB	882.51 MiB	980.56 MiB
	$\eta_{13} \approx 0.0219$	12.87 MiB	15.02 MiB	17.17 MiB	19.31 MiB	21.46 MiB
	15	1.02 GiB	1.19 GiB	1.36 GiB	1.53 GiB	1.70 GiB
	$\eta_{15} \approx 0.0166$	17.32 MiB	20.21 MiB	23.10 MiB	25.99 MiB	28.87 MiB
	17	1.68 GiB	1.96 GiB	2.24 GiB	2.52 GiB	2.80 GiB
	$\eta_{17} \approx 0.0130$	22.43 MiB	26.17 MiB	29.91 MiB	33.65 MiB	37.39 MiB
	19	2.62 GiB	3.06 GiB	3.50 GiB	3.93 GiB	4.37 GiB
	$\eta_{19} \approx 0.0105$	28.20 MiB	32.90 MiB	37.60 MiB	42.30 MiB	47.00 MiB
	21	3.91 GiB	4.56 GiB	5.22 GiB	5.87 GiB	6.52 GiB
	$\eta_{21} \approx 0.0086$	34.63 MiB	40.40 MiB	46.17 MiB	51.94 MiB	57.71 MiB

Table 4.1: Sampled memory reduction for $150 \leq m_z \leq 250$ and $7 \leq m_{xy} \leq 21$. Each cell contains the current (above) and new (below) required memory to store $\mathbf{C}$ or $\mathbf{M}$.

Note that the memory efficiency solely depends on m_{xy} rather than m_z. For $m_{xy} = 7$, the Toeplitz compressed data is already more than one order of magnitude cheaper to store than the CSR data. Even two orders of magnitude can be achieved for $m_{xy} = 21$.

4.3.3 Ordering with grouped nested Toeplitz blocks

The ordering of the non-zeros where the indices are sorted on r, p, q, d_1 and d_2 consecutively, gives rise to the following identity. Let s be the position in the formatted value list `mat`. Then

$$s = 9(2m_{xy}-1)^2 r + 3(2m_{xy}-1)^2 p + (2m_{xy}-1)^2 q + (2m_{xy}-1)(d_1 + m_{xy} - 1) + (d_2 + m_{xy} - 1). \tag{4.16}$$

This ordering guarantees that the non-zeros of each $\mathbf{D}_{p,q}$ occupy a contiguous region in memory.

In order to let each thread process one row i of the matrix at a time in a parallel scalable algorithm, a collapsible iteration space I must be found, depending on the novel indices, such that each i is a unique element of I. This iteration space I follows from (4.11), since each i can be written as a dependence with indices r, p, r_1 and r_2. Hence, each i can be expressed as an element

$$(r, p, r_1, r_2) \in I = [0, m_z) \times [0, 3) \times [0, m_{xy})^2. \tag{4.17}$$

Subsequently with (4.14), use r_1 and r_2 to find which d_1 blocks and d_2 scalars appear on the respective (block) rows. This ultimately determines the set of input memory addresses. Unfortunately, since not all diagonally repeated scalars can be found on a row, this will not be a contiguous set of integers. The following figure shows the access behavior of this algorithm, which is hereby baptized as CMPT (**C** *or* **M P***arallel algorithm using* **T***oeplitz compression.*)

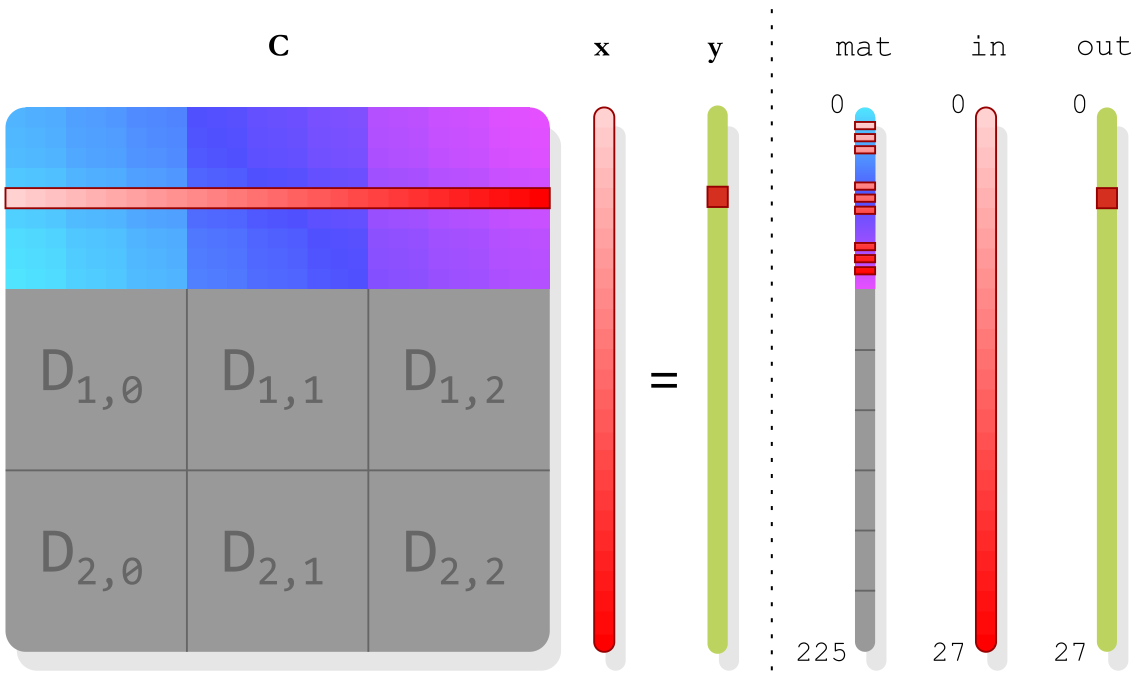


Figure 4.2: Mathematical (left) and digital (right) representation of the accesses of one thread for $m_{xy} = 3$ and $m_z = 1$.

Note that the input vector addressing is perfectly linear and contiguous. The accesses in `mat`, however, are quite scattered. Since Figure 4.2 shows that the accesses in each $\mathbf{T}_{d_1}$ (of size $m_{xy} \times m_{xy}$) are linear and contiguous, and each row contains $3m_{xy}^2$ non-zeros (Section 2.3.1), the matrix access memory addresses consists of $3m_{xy}$ contiguous subsets.

The next section will account for improving the accesses in the matrix data, by reordering it.

4.3.4 Improving the matrix access pattern

When setting q as the last sorting attribute, each scalar diagonal d_2 of block diagonal d_1 of blocks $\mathbf{D}_{p,0}$, $\mathbf{D}_{p,1}$ and $\mathbf{D}_{p,2}$ are grouped. The matrix non-zeros for these three blocks are now interleaved. The identity for s then becomes

$$s = 9(2m_{xy}-1)^2 r + 3(2m_{xy}-1)^2 p + 3(2m_{xy}-1)(d_1+m_{xy}-1) + 3(d_2+m_{xy}-1) + q. \quad (4.18)$$

Note that the collapsible iteration space in the previous section does not change. Again, by using (4.11) and (4.14), a new algorithm is obtained (CMPT2) which uses the following access pattern:

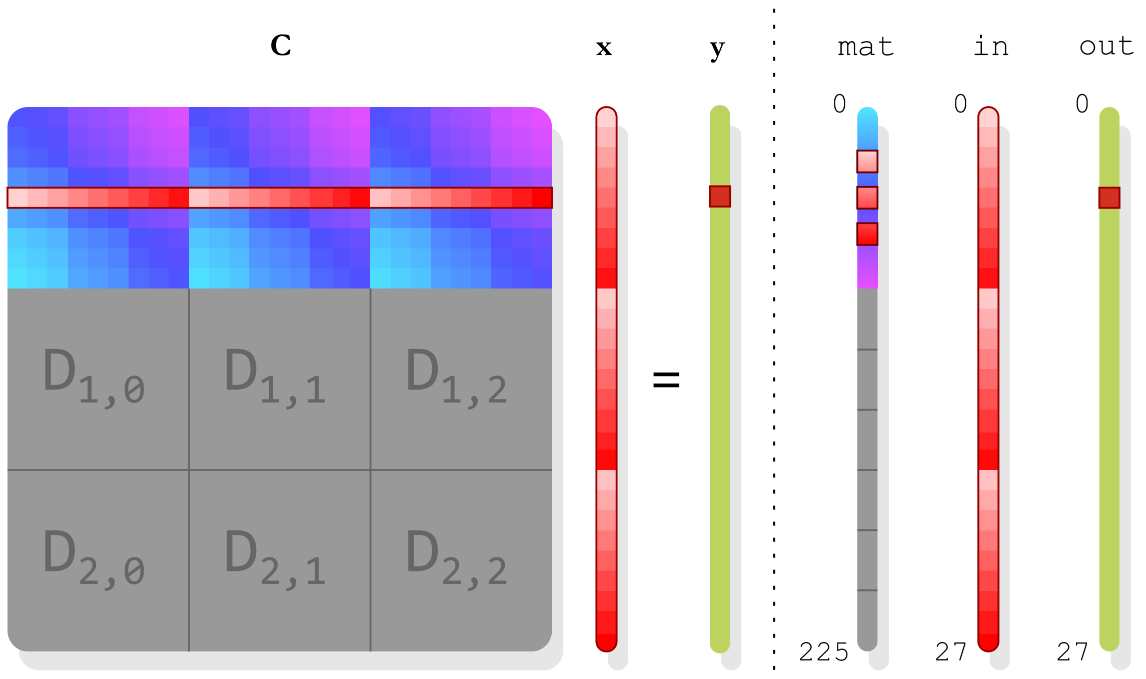


Figure 4.3: Mathematical (left) and digital (right) representation of the accesses of one thread for $m_{xy} = 3$ and $m_z = 1$.

Overall, this creates improved matrix value accesses for each row, by eliminating the separation by q, improving the patterns in the previous section. Now there are only m_{xy} linear contiguous access subsets, a substantial improvement over the previous algorithm. However, accessing the needed values of `mat` in ascending order means a highly irregular access pattern in the input vector. After all, q is the last sorting index, implying frequently occurring increments of m_{xy}^2. Yet, there exists a vector permutation resulting in a linear access pattern in the input vector.

4.3.5 Permuting the input vector

Considering Figure 4.3 in the previous section, one can see that the access pattern in the input vector is systematic, albeit being non-linear. This pattern emerged when q was set to be the last matrix value sorting attribute. Reproducing that for (4.10), one can define a permutation σ such that

$$\sigma(j) = \sigma(3m_{xy}^2 r + m_{xy}^2 q + m_{xy}c_1 + c_2) = 3m_{xy}^2 r + 3m_{xy}c_1 + 3c_2 + q. \tag{4.19}$$

and thereby also a parallel algorithm which copies a vector $\mathbf{x}$ to $\mathbf{x}'$ subject to this permutation, i.e. setting $\mathbf{x}'_{\sigma(j)} = \mathbf{x}_j$ for all $(r, q, c_1, c_2) \in [0, m_z) \times [0, 3) \times [0, m_{xy})^2$. In the following figure, the access behavior of yet a new variant (CMPT3) is displayed.

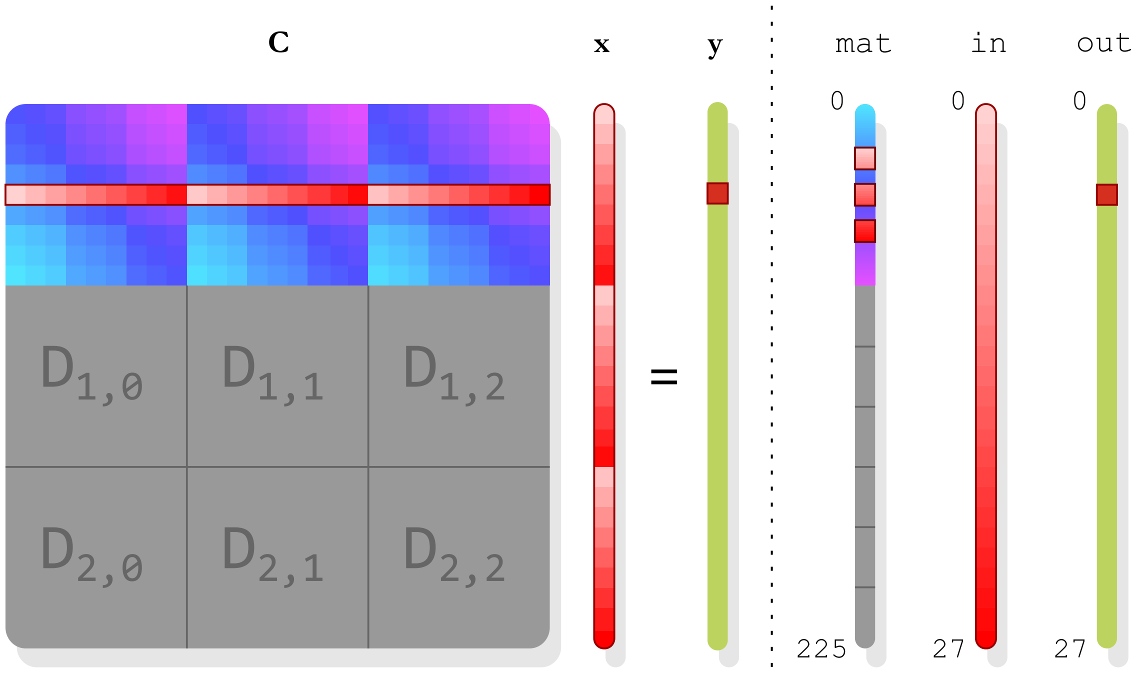


Figure 4.4: Mathematical (left) and digital (right) representation of the accesses of one thread (red) for $m_{xy} = 3$ and $m_z = 1$.

The section concludes with this algorithm. It features linear contiguous accesses in the input vector, and a linear pattern consisting of m_{xy} contiguous parts in the matrix data.

In Chapter 6 it will become clear that quite a portion of the SpMVs of the IDR method will be performed on the columns of an auxiliary matrix, implying copying a matrix column to a vector is required in any case. This copy method (also parallel) can therefore incorporate this permutation to save time.

4.4 Performance analysis

In order to find the parallel algorithm with the highest performance, two different benchmarks will be made: one where the size parameter m_{xy} varies and one for a varying m_z. These nanosecond precision benchmarks are obtained by using Linux `timespec` structs around the parallel region of each method. Naturally, all benchmarks will be obtained on the same CPU model.

To display parallel scalability of the algorithms, also a benchmark will be made with varying amount of OpenMP threads. This section will close with a profile report for a few parameter choices. This is a report made by the Intel® VTune™ tool, which can feedback a spectrum of runtime statistics: cache hit/miss rates, threads used, times for invoked voids, etc.

4.4.1 Benchmarks

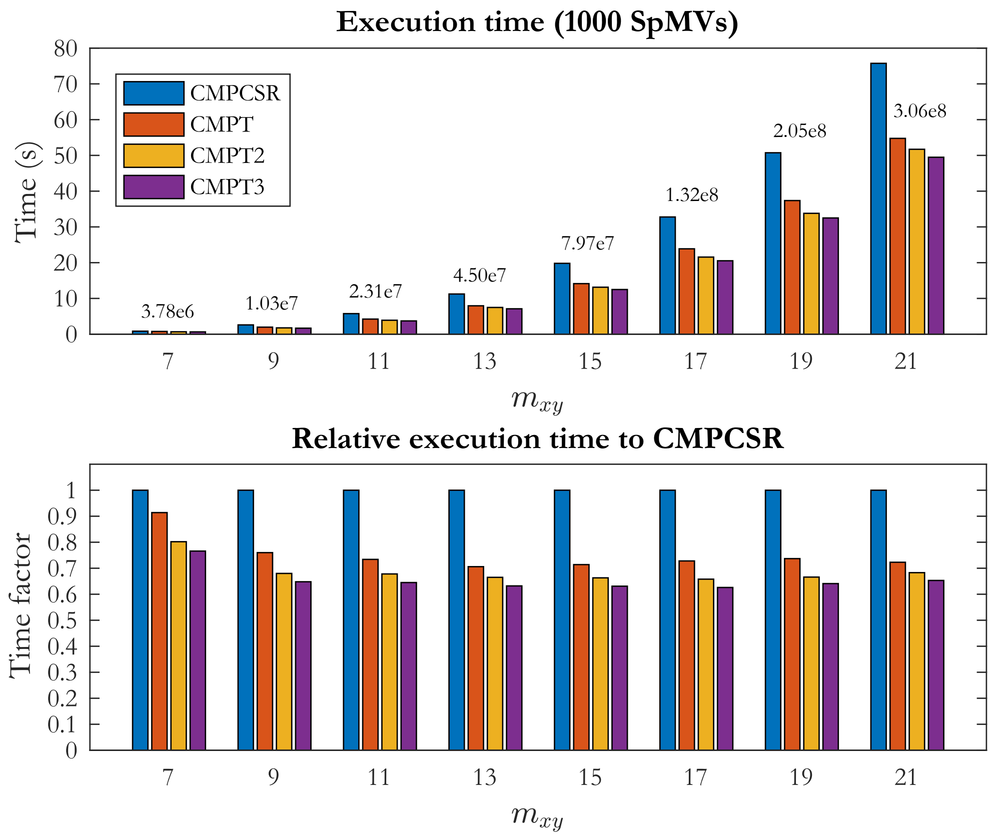


Figure 4.5: Absolute and relative execution time for different m_{xy} and fixed $m_z = 175$, obtained on a single Intel® Xeon® E5-2660 v4 CPU (14 OpenMP threads). The bars in the top graph are labeled with nnz($\mathbf{C}$).

As predicted in Section 2.3.3 and as seen in Figure 4.5, increasing the parameter m_{xy} strongly influences the execution time of a matrix-vector product, since the non-zeros of **C** and **M** have a fourth order dependency on m_{xy}. Note that with Toeplitz diagonal compression, the execution time is already reduced by around 25% of the original time, and that introducing this format resulted in the best improvement between the algorithm in this chapter. From the differences in execution time between the CMPT family algorithms, conclude that facilitating linear data access patterns (also called *unit stride accessing*) also pays off in performance, because CMPT3 has improved access patterns over CMPT and CMPT2. The top relative times lie around 60% and 65% of the parallel CSR method (CMPCSR, Section 4.2). From $m_{xy} = 9$ onwards the relative time gain is more or less constant, which hints to good employability for high parameter choices.

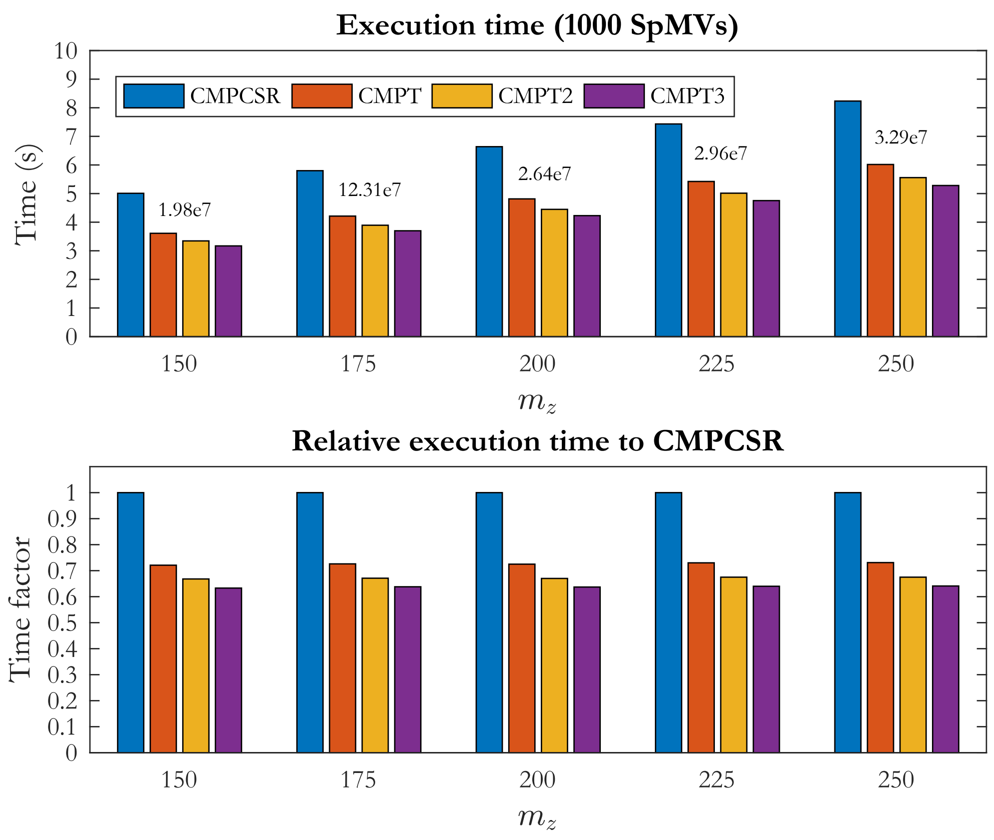


Figure 4.6: Absolute and relative execution time for different m_z and fixed $m_{xy} = 11$, obtained on a single Intel® Xeon® E5-2660 v4 CPU (14 OpenMP threads). The bars are labeled with nnz($\mathbf{C}$).

Figure 4.6 displays linear growth in execution with respect to m_z, which can be explained by the linear dependency of nnz($\mathbf{C}$) on m_z (Section 2.3.3). The relative time gain for the tailored methods are very constant for different choices for m_z. This implies a good employability for large choices of m_z, in addition to extreme m_{xy}. Again, the top time factors lie between 60% and 65%. Clearly the algorithm described in Section 4.3.5 has the highest performance. Conclude that the unit stride accessing as well as lowering the memory footprint are worth researching when designing an algorithm to boost performance.

4.4.2 Parallel scalability

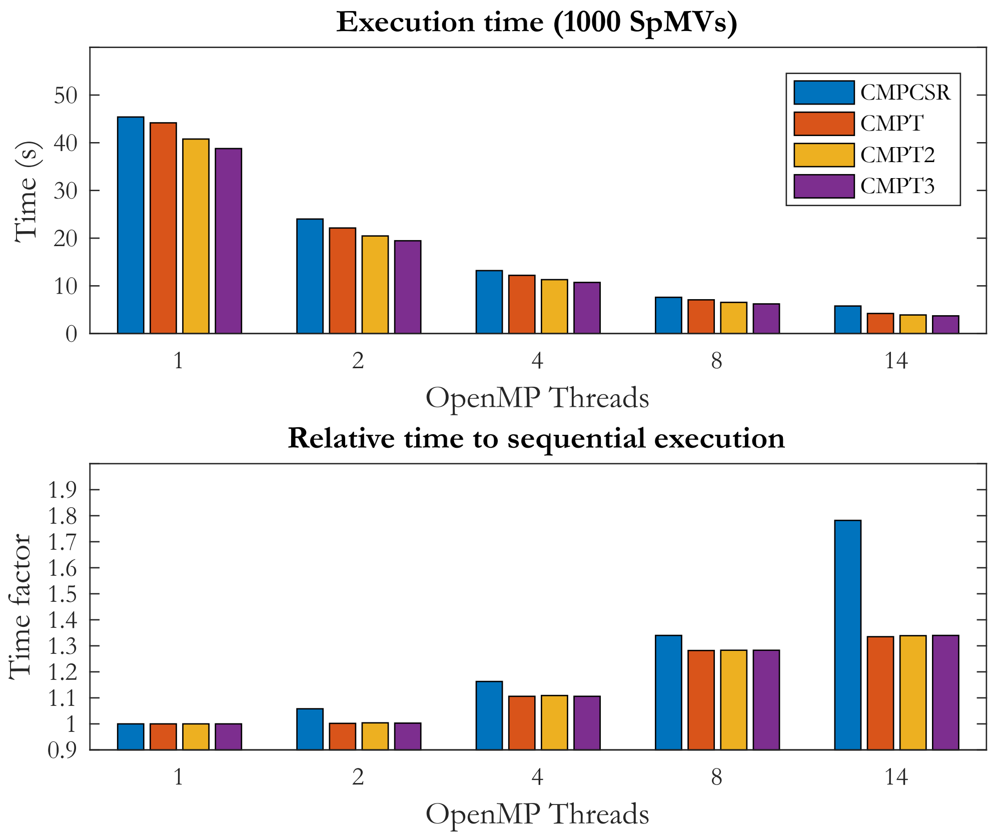


Figure 4.7: Absolute and relative execution time for $m_{xy} = 11$ and $m_z = 175$, for different amounts of OpenMP threads, obtained on a single Intel® Xeon® E5-2660 v4 CPU (max. 14 OpenMP threads). The time factors on the bottom graph are obtained by dividing the parallel time by its sequential execution time, multiplied by the amount of OpenMP threads.

Figure 4.7 clearly shows that operating on a more economic matrix format benefits the parallel scalability, since the three novel algorithms (the CMPT family) have competing time factors, yet they are a significant reduction on those of CMPCSR. After all, the figure shows that only about 35% of the time of a single execution is additionally needed to perform fourteen SpMVs with the CMPT3 algorithm with fourteen threads. This contrasts with the 80% additional needed time for CMPCSR.

4.4.3 Profile reports

Below some runtime statistics, gathered with the Intel® VTune™ profiler, are listed for four extreme parameter choices (low and high for both m_{xy} and m_z) to display lower and higher bounds for memory usage. Only four profiles were made, because they generally demand a respectable amount of storage space to store and quite some time to create.

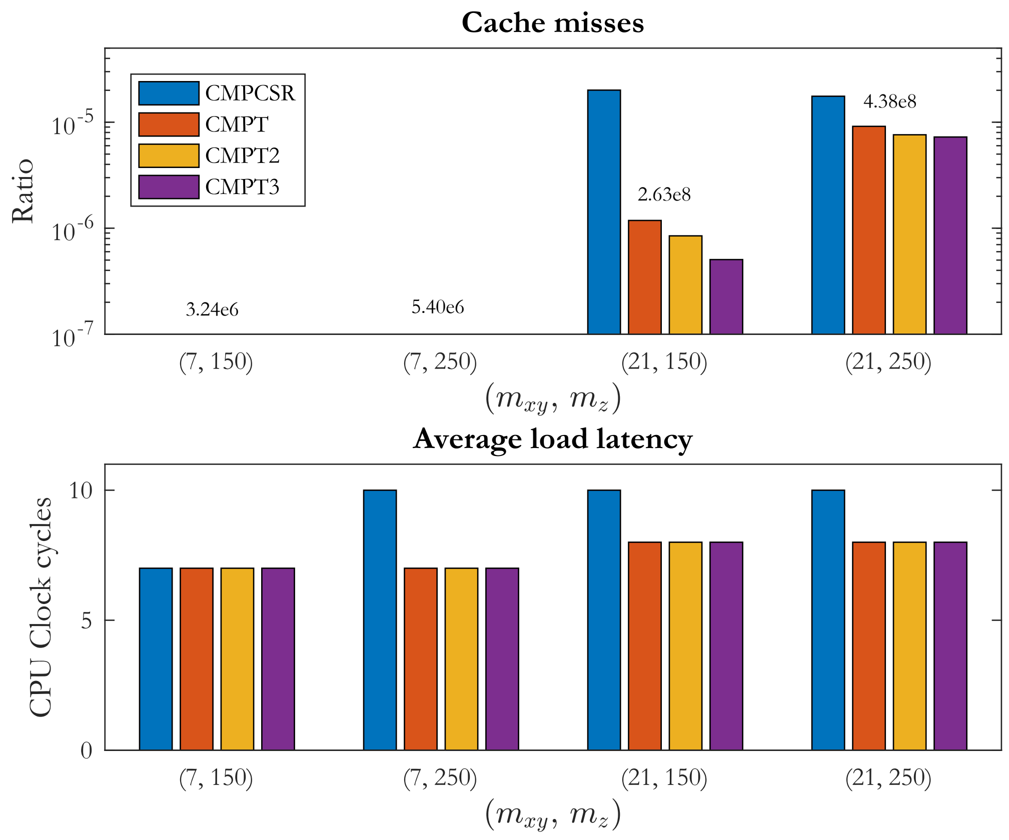


Figure 4.8: Cache miss ratio and average load latencies for 1000 SpMVs, four extreme parameter choices and 14 OpenMP threads.

Exceptionally enough, for $m_{xy} = 7$ all loads were done within the CPU cache, even for the regular CSR method. As Figure 4.8 also shows, the extreme choice of $m_{xy} = 21$ implies a significantly more efficient cache ratio for the tailored CMPT algorithms as opposed to CMPCSR. The CMPT loads also have at least the same, but mostly lower latencies than CMPCSR. Conclude that across the board, the custom algorithms exploit the CPU better, which is an explanation for the benchmarks in the previous sections.

4.5 Parallel heat diffusion simulation using FDMs

The Toeplitz compression researched in this chapter to tailor a proven pair of economical matrix format and SpMV algorithm can be modified to suit more differential equation solving purposes. After all, matrices with (nested) Toeplitz blocks frequently follow from the finite-difference methods, where the discretization points form a lattice.

Consider for example the simulation of a heat diffusion problem in time and three-dimensional space. Let Ω be a cube and let $u(\mathbf{x}, t)$ denote the temperature for $\mathbf{x} \in \Omega$ and $t \geq 0$. The heat diffusion in the cube is described by the computational scheme

$$\begin{aligned} \frac{u(\mathbf{x}, t+\Delta t) - u(\mathbf{x}, t)}{\Delta t} =& \alpha(\mathbf{x}) \Bigg(\frac{u(\mathbf{x}+\hat{\mathbf{e}}_1, t) - 2u(\mathbf{x}, t) + u(\mathbf{x}-\hat{\mathbf{e}}_1, t)}{(\Delta x)^2} + \\ & \frac{u(\mathbf{x}+\hat{\mathbf{e}}_2, t) - 2u(\mathbf{x}, t) + u(\mathbf{x}-\hat{\mathbf{e}}_2, t)}{(\Delta y)^2} + \\ & \frac{u(\mathbf{x}+\hat{\mathbf{e}}_3, t) - 2u(\mathbf{x}, t) + u(\mathbf{x}-\hat{\mathbf{e}}_3, t)}{(\Delta z)^2} \Bigg), \end{aligned} \tag{4.20}$$

for some real thermal diffusivity function $\alpha : \mathbb{R}^3 \to \mathbb{R}^+$, spatial step sizes Δx, Δy and Δz, time step size Δt, and where $\hat{\mathbf{e}}_k$ denotes a standard basis vector scaled to the specific step size in that direction, i.e. $\hat{\mathbf{e}}_1 = (\Delta x)\mathbf{e}_1$.

Assume a regular three-dimensional lattice in Ω consisting of $m \times m \times m$ discretization points and let h be the distance between any pair of neighboring points. Then by (4.20), using the second order central difference approximation, an initial heat function $u(\mathbf{x}, 0)$, time increments Δt and Dirichlet boundary conditions, one obtains

$$\begin{cases} u(\mathbf{x}, t) & = u(\mathbf{x}, 0) \quad \mathbf{x} \in \partial\Omega, t > 0, \\ u(\mathbf{x}, t+\Delta t) & = u(\mathbf{x}, t) + \frac{(\Delta t)\alpha(\mathbf{x})}{h^2}\big(-6u(\mathbf{x}, t) + \sum_{k=1}^{3} u(\mathbf{x} \pm \hat{\mathbf{e}}_k, t)\big) \quad \mathbf{x} \in \Omega \setminus \partial\Omega. \end{cases}$$

Labeling the discretization points systematically, effectively putting the initial function values into a vector $\mathbf{u}^{(0)}$, time iterations of this heat diffusion problem can be computed by means of computing $\mathbf{u}^{(t+\Delta t)} = \mathbf{L}\mathbf{u}^{(t)}$, where $\mathbf{L}$ is a matrix following from above, which has a three-fold nested Toeplitz block property. By the computational scheme above, however, one can conclude that this problem can be simulated without explicitly storing $\mathbf{L}$, because only a couple of (block) diagonals are non-zero. The computations can be performed as follows, given an initial heat distribution $\mathbf{u}$ and a vector α containing the diffusivity for each point in the lattice:

```
$t \leftarrow 0$
**while** $t < t_{\max}$ **do**
    Compute $\mathbf{v} : v_i = \begin{cases} u_{i\pm 1} + u_{i\pm m} + u_{i\pm m^2} - 6u_i & \text{for internal points,} \\ 0 & \text{for boundary points.} \end{cases}$    (i)
    $\mathbf{w} \leftarrow \mathbf{v} \odot \alpha$    (ii)
    $\mathbf{u} \leftarrow \mathbf{u} + \frac{\Delta t}{h^2}\mathbf{w}$    (iii)
    $t \leftarrow t + \Delta t$
**end while**
```

Each of the three vector operation steps in the WHILE-loop can be performed in parallel. The element-wise product and the addition of changes in $\mathbf{u}$ are each done with a single instruction, but the computation of $\mathbf{v}$ is rather irregular. In addition to having a condition, the two resulting cases have a different expected computation time.

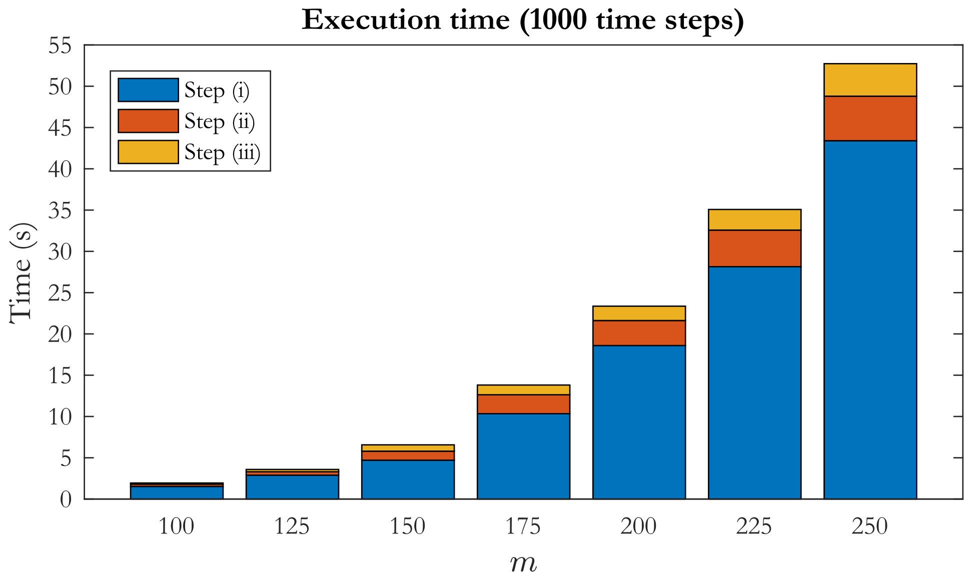


Figure 4.9: Benchmarks of parallel finite-difference heat diffusion simulation with Dirichlet boundary conditions for various m, obtained on two Intel® Xeon® E5-2660 v4 CPUs (28 OpenMP threads).

Note that almost all of the computation time is indeed used for the first operation in the WHILE-loop, which is the computation of the diffusion in **u**. Even though the concerned operation consists of just seven additions per vector position, the execution time of this part is proportionally large compared to the other two steps. This can be explained by the IF-statement which distinguishes boundary and non-boundary points, and the fact that **u** is accessed quite irregularly, namely at positions $\pm 1, \pm m, \pm m^2$ relative to those of **v**. The next chapter will show the importance of linear input accessing and the possible achievable speed gain.

For differential equations that result in nested Toeplitz matrices or blocks, it is very fruitful to look at discretization schemes where the amount of non-zero diagonals can be systematically counted. The very crux of this chapter therefore is that exploiting such schemes to design a cheaper matrix format and an associated parallel SpMV algorithm has proven to improve the speed by lowering the memory footprint, rendering them scalable for larger computational problems.

Chapter 5

SpMVs with equidistant diagonals

The previous chapter has shown that the ability to compress matrix data and design an SpMV algorithm thereon can benefits performance. Unfortunately, as already stated in Section 2.3, every non-zero of **G** is generally unique, implying no compression can be used in this case. However, the previous chapter also conveyed that it is worthwhile to research optimization of matrix and vector value accesses. This confirmed the content of Section 3.3 and will be the focus of the methods presented in this chapter.

This chapter will therefore also commence with a standard parallel method on CSR formatted matrices (Section 5.1), from which improvements will be made, common to those in Section 4.2. Since **G** has a diagonal structure, also element-wise vector-vector products can be exploited to design cache-efficient SpMV algorithms (Section 5.2).

5.1 Row-parallel algorithms

Recalling Section 2.3, **G** has a fixed amount of $3m_z$ non-zeros in each row, and since the diagonals all have the same fixed distance between them, the non-zeros too have a fixed distance between them. From the structure of **G** it follows that when linearly accessing the matrix values `mat`, there are accesses in the input vector with increments of m_{xy}^2, starting from a certain offset depending on the row i.

5.1.1 Parallel CSR multiplication

Like in the previous chapter, one must analyze the matrix structure and come up with a collapsible iteration space I such that each matrix row i is represented by a unique element of I, and such that the matrix data and input vector addresses can be constructed.

Since the distance between the non-zero diagonals is m_{xy}^2, g_{ij} is non-zero if and only if $j - i$ is a multiple of m_{xy}^2. This will aid in constructing an iteration space for this matrix structure. The row

i and column j of a matrix can therefore respectively be expressed as

$$i = m_{xy}^2 i' + k, \tag{5.1}$$

$$j = m_{xy}^2 j' + k, \tag{5.2}$$

with $i', j' \in [0, 3m_z)$ and $k \in [0, m_{xy}^2)$.

Consider once more Figure 2.2. Hence instead of passing $i \in [0, n)$ to a thread in a parallel algorithm, one should pass i' and k. This way, the row i of the matrix can be reconstructed. Moreover, the needed input vector addresses are given by increasing matrix column j from k up to $n-1$ with increments m_{xy}^2. Conclude that the iteration space $I = [0, 3m_z) \times [0, m_{xy}^2)$ is satisfactory for a parallel algorithm, which will bear the name GPCSR (**G P***arallel algorithm using the* **CSR** *format*).

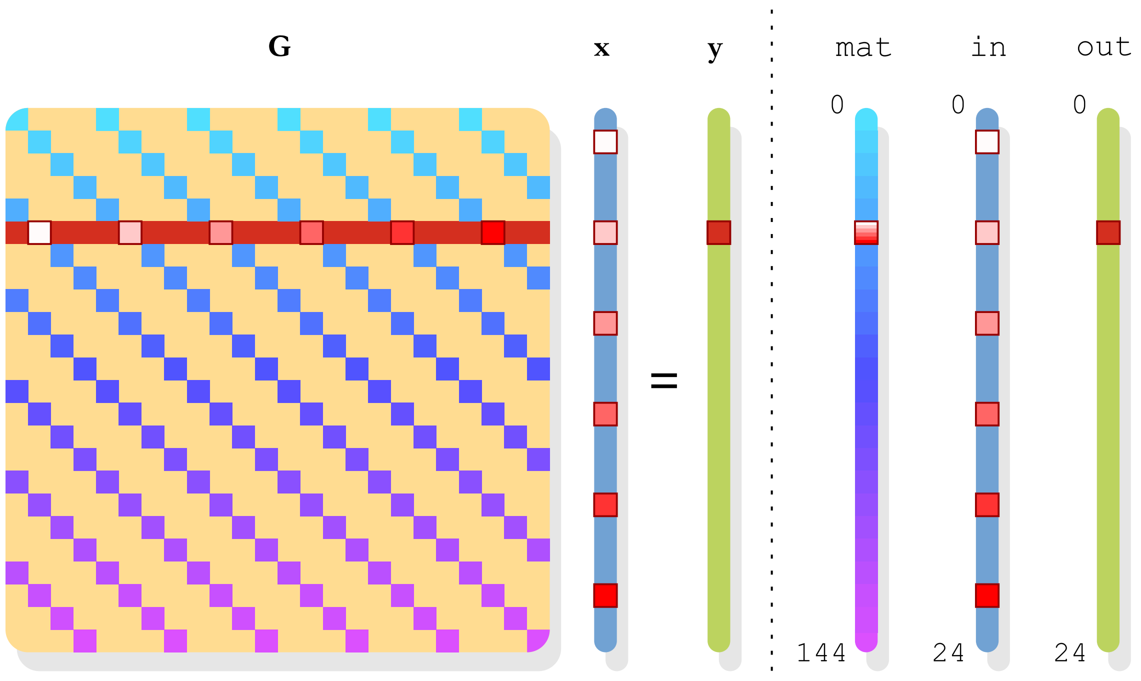


Figure 5.1: Mathematical (left) and digital (right) representation of the accesses of one thread (red) for $m_{xy} = 2$ and $m_z = 2$.

Note that the accesses in the input vector are not optimal yet similar to those in Section 4.3.4. Luckily the same permutation used in Section 4.3.5 can be used to remedy this.

5.1.2 Permuting the input vector

Equivalently to the previous chapter, i and j have already been dissected to accommodate systematic counting with a collapsible iteration space I (5.1). The dissection of j now becomes clear, for those j needed by a thread processing row i differ by m_{xy}^2, similarly to those in Section 4.3.4. Using a similar permutation, define

$$\sigma(j) = \sigma(m_{xy}^2 j' + k) = 3m_z k + j', \tag{5.3}$$

such that a parallel vector permutation method, which sets $\mathbf{x}'_{\sigma(j)} = \mathbf{x}_j$ for all such

$$(j', k) \in I = [0, 3m_z) \times [0, m_{xy}^2).$$

Now, each thread has a contiguous linear accesses pattern, because for each row i, g_{ij} is non-zero if and only if $\sigma(j) \in [3m_z k, 3m_z(k+1))$. The resulting algorithm, GPPIN (**G** **P***arallel algorithm with* **P***ermuted* **IN***put*), has the following access behavior.

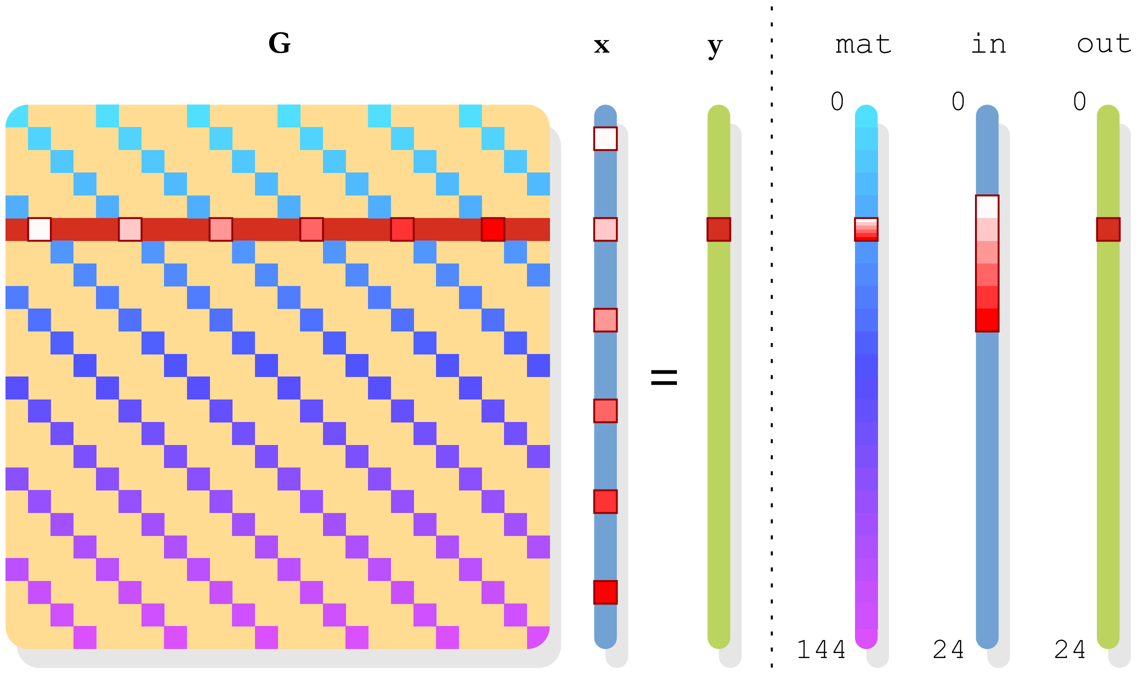


Figure 5.2: Mathematical (left) and digital (right) representation of the accesses of one thread (red) for $m_{xy} = 2$ and $m_z = 2$.

A final albeit small variant on this algorithm can be made by grouping the matrix rows which have the same offset.

5.1.3 Offset clustering

Using the same permutation on i results in

$$\sigma(i) = \sigma(m_{xy}^2 i' + k) = 3m_z k + i', \tag{5.4}$$

and reordering the matrix values on rows $\sigma(i)$ instead of i (used in regular CSR) will result in slightly more regular use of the (still permuted) input vector. This gives rise to a GPPIN variant, which will be aptly named GPPIN2.

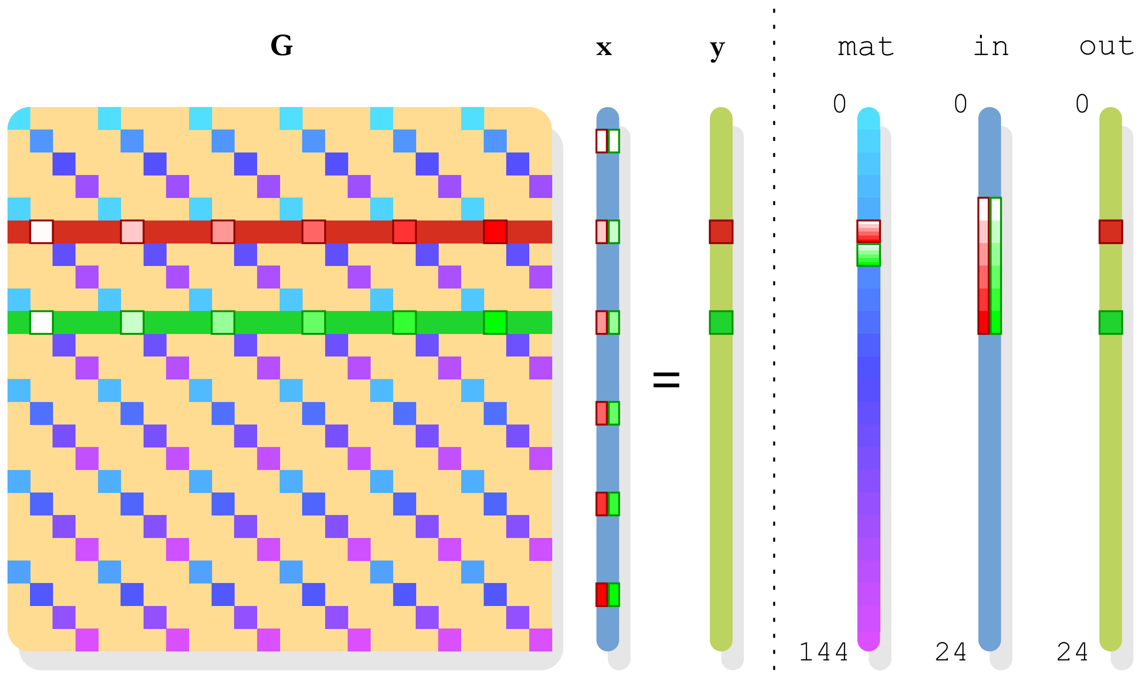


Figure 5.3: Mathematical (left) and digital (right) representation of the accesses of two threads (red and green) for $m_{xy} = 2$ and $m_z = 2$.

In Figure 5.3 one can see that if `mat` is now processed linearly, so will the input vector.

5.2 Element-wise product method

The diagonal structure of **G** also suggests a radically different way of multiplying it with a vector. Consider a diagonal matrix **D**, and let its diagonal in vector form be denoted by $\text{diag}(\mathbf{D}) = \mathbf{d}$. Then the product **D** with a vector **x** can be performed with a single highly parallelizable element-wise product $\mathbf{d} \odot \mathbf{x}$.

This concept can be generalized to fit a matrix with diagonals separated by a fixed distance. Firstly however, the non-zeros in each diagonal needs to be clustered for optimal accessing (Section 3.3), which requires a sensible format.

Apart from the main diagonal, every diagonal has a length smaller than n. Yet the fixed distance between the diagonals poses a way to concatenate them for a rather facile SpMV algorithm. Consider a sub-diagonal, i.e. with MATLAB index $d < 0$. Then that diagonal reaches the bottom of the matrix in column $n + d - 1$. Fortunately, due to the fixed distance between the diagonals, a diagonal starts at the top of the matrix in the very next column.

Concatenating these two diagonals results in a vector of length n which can be multiplied element-wise with the input vector. Since this concatenated diagonal started from the left at row $i = -d$, one must apply an i-fold circular shift of the outcome to the output vector, which is achieved by setting i equal to j plus an offset with modulus n.

If one can order the non-zeros such that all those concatenated diagonals starting at

$$i = 0, m_{xy}^2, 2m_{xy}^2, \ldots, (3m_z - 1)m_{xy}^2$$

appear in order, one can, by means of element-wise products and incrementing at circular positions, implement it such that the matrix values are accessed linearly and for any such diagonal, process the input vector linearly. Even the 'one thread per output position at the same time' paradigm is maintained, for one concatenated diagonal is processed at a time.

This algorithm, baptized as GPDIAG (**G** *algorithm using* **P***arallel* **DIAG***onal processing*), has the following stride pattern.

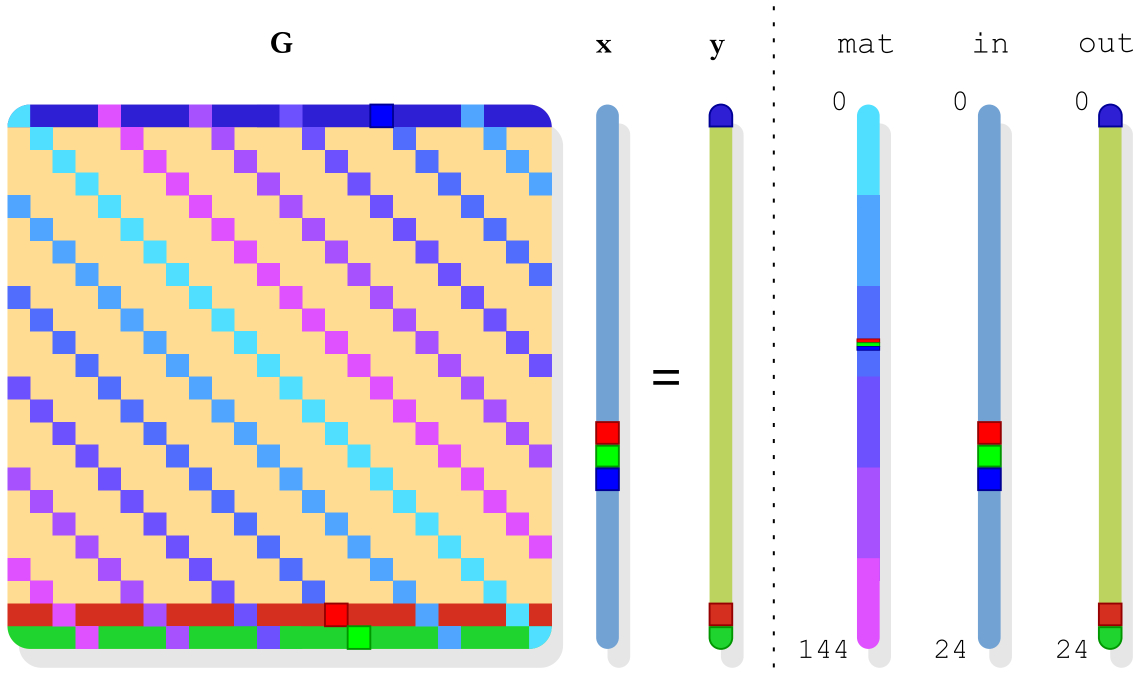


Figure 5.4: Mathematical (left) and digital (right) representation of the accesses of a size three team processing a diagonal.

This algorithm differs substantially from the other algorithms presented in this chapter. Here, a thread treats one non-zero at a time, instead of processing a whole matrix row. Since the algorithm is implemented such that the team is re-created for every diagonal, i.e. the team is working on exactly one diagonal simultaneously, the algorithm will not have multiple threads outputting to the same output vector position. This way, synchronization overhead is minimized.

5.3 Performance analysis

The benchmarks and profile reports in this section are obtained in the same way as those in the previous chapter.

5.3.1 Benchmarks

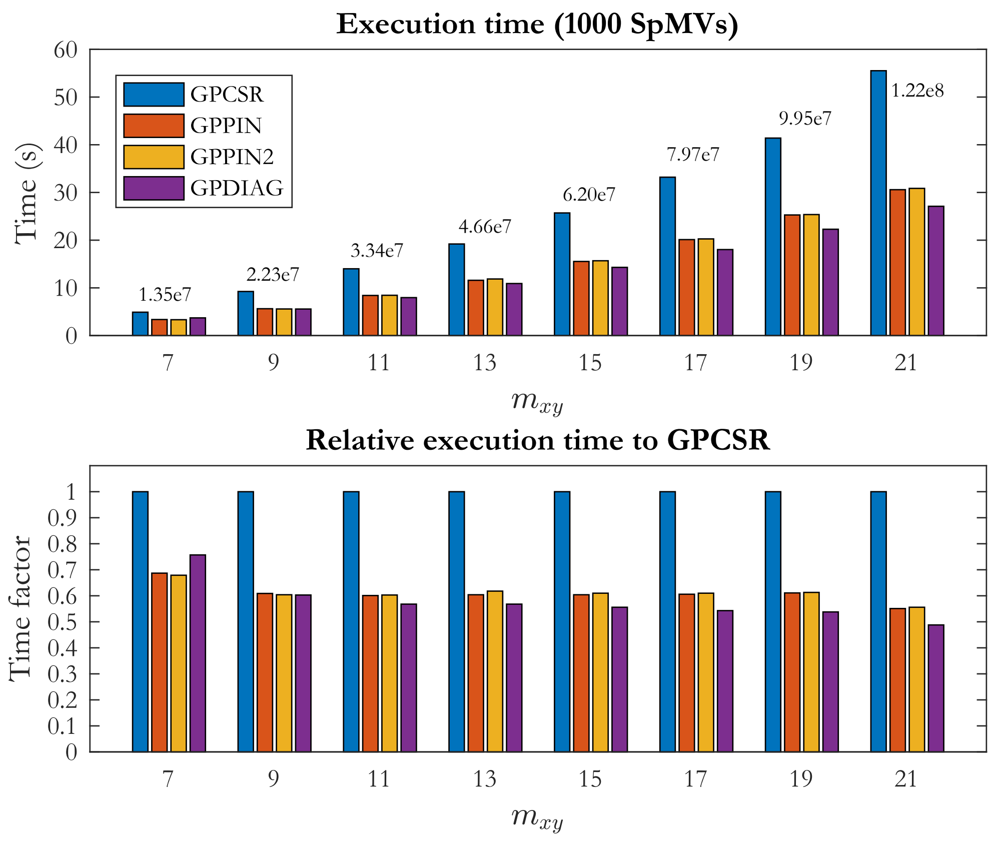


Figure 5.5: Absolute and relative execution time for different m_{xy} and fixed $m_z = 175$, obtained on a single Intel® Xeon® E5-2660 v4 CPU (14 OpenMP threads). The bars are labeled with nnz($\mathbf{G}$).

As seen in Figure 5.5, the execution time has super-linear growth in parameter m_{xy}, which can be explained by the expression for nnz($\mathbf{G}$) in Section 2.3.3. The tailored methods maintain a constant relative time gain opposed to GPCSR across the board, even for large choices of m_{xy}, meaning that they do not lose their benefit for high parameter choices. However, the last presented method (GPDIAG, Section 5.2) even seems to display improvements in the relative time gain when m_{xy} grows large. Interesting is to see that grouping the matrix data for a team iteration (GPPIN2, Section 5.1.3) does not seem to pay off in performance and is even slightly outperformed by GPPIN. Conclude that ordering the matrix and input vector data such that unit stride accessing is achieved for individual threads is satisfactory, as opposed to additionally grouping the matrix data for a whole team. Overall the best times lie between 50% and 62% of the general parallel CSR method.

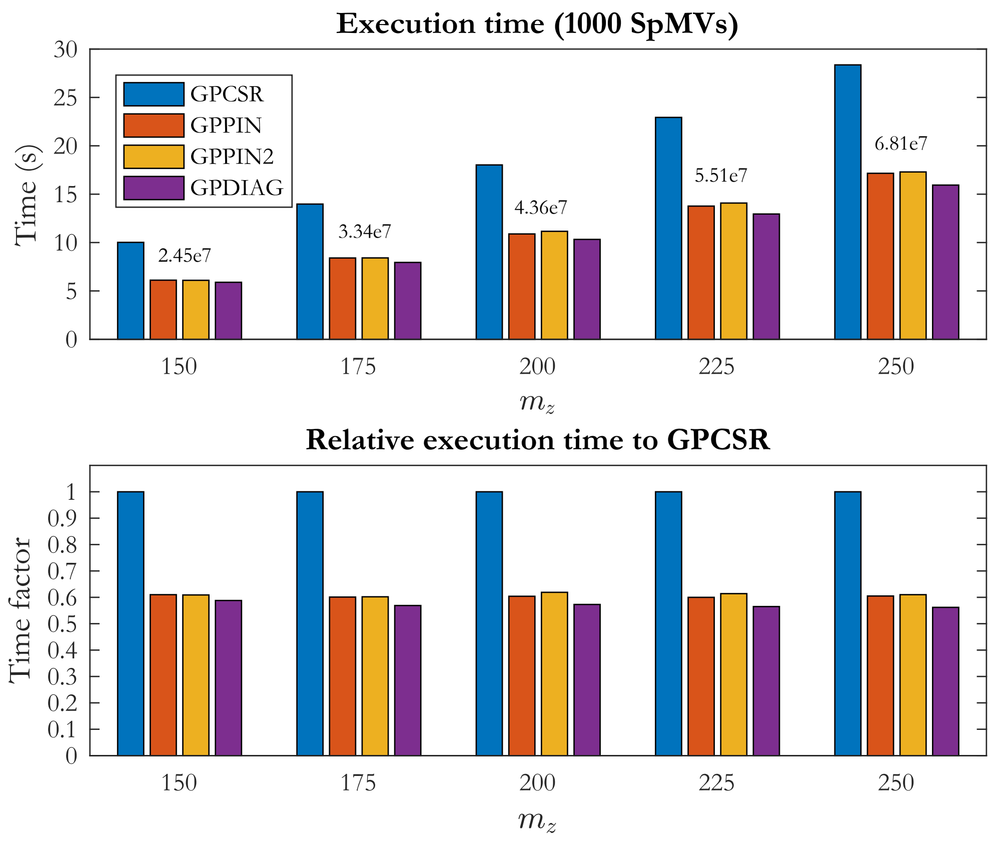


Figure 5.6: Absolute and relative execution time for different m_z and fixed $m_{xy} = 11$, obtained on a single Intel® Xeon® E5-2660 v4 CPU (14 OpenMP threads). The bars are labeled with nnz($\mathbf{G}$).

As seen in Figure 5.6, the growth of the execution time is super-linear in parameter m_z (again, this can be explained by the expression for nnz($\mathbf{G}$), even though it displays less erratic growth than that of parameter m_{xy}. Regardless of the choice for this parameter, the relative time gain is almost perfectly constant. This again implies that the algorithms do not lose their benefit for extreme choices of m_z. This benchmark shows that also for m_z, the GPPIN2 algorithm (Section 5.1.3) does not pose any improvements upon GPPIN, its original (Section 5.1.2).

The speed gains for this SpMV are purely achieved by facilitating unit stride data accessing, which, as shown in the two figures above, boosts performance by a factor of almost two.

5.3.2 Parallel scalability

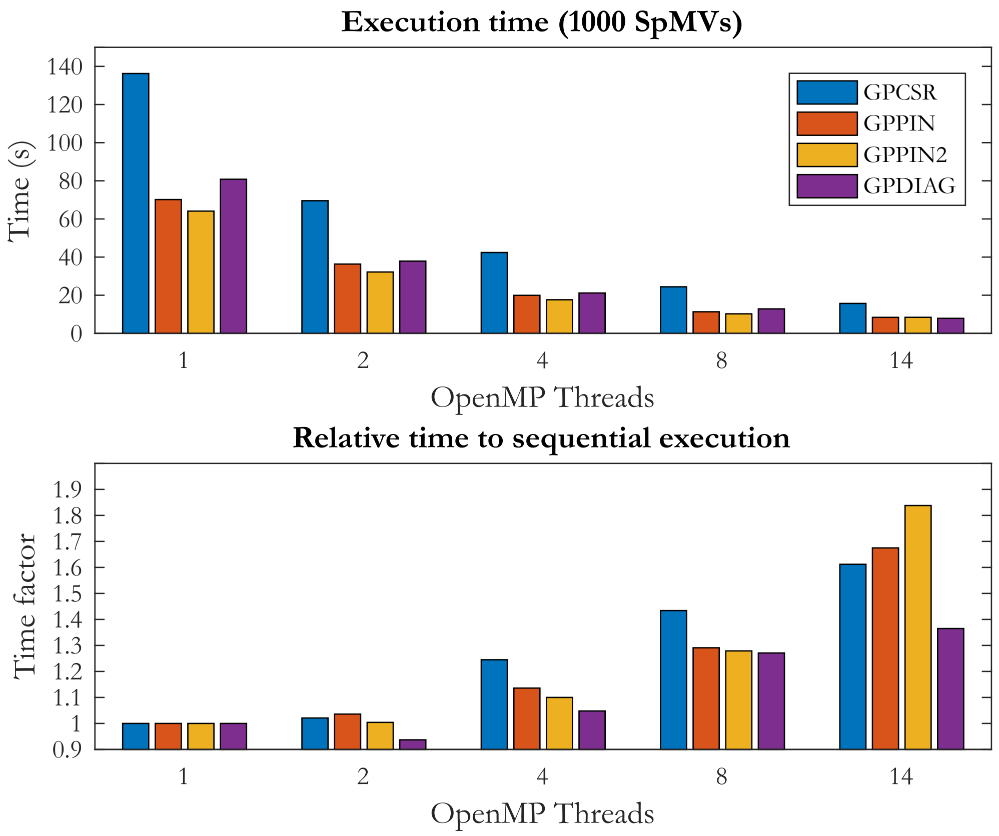


Figure 5.7: Absolute and relative execution time for $m_{xy} = 11$ and $m_z = 175$ and different amount of OpenMP threads, obtained on a single Intel® Xeon® E5-2660 v4 CPU (max. 14 OpenMP threads). The time factors on the bottom graph represent how many sequential executions fit into the amount of OpenMP threads times the corresponding execution time.

The figure above shows that even though an algorithm performs better in terms of absolute execution time, it might not scale as well in parallel. After all, even though GPDIAG is outperformed by the GPPIN family for smaller sizes (as also seen in the previous section), GPDIAG is more scalable than the other algorithms. This confirms the added value of the time factor metric for parallel algorithms. Also note that linear access patterns in the data benefit the scalability.

5.3.3 Profile reports

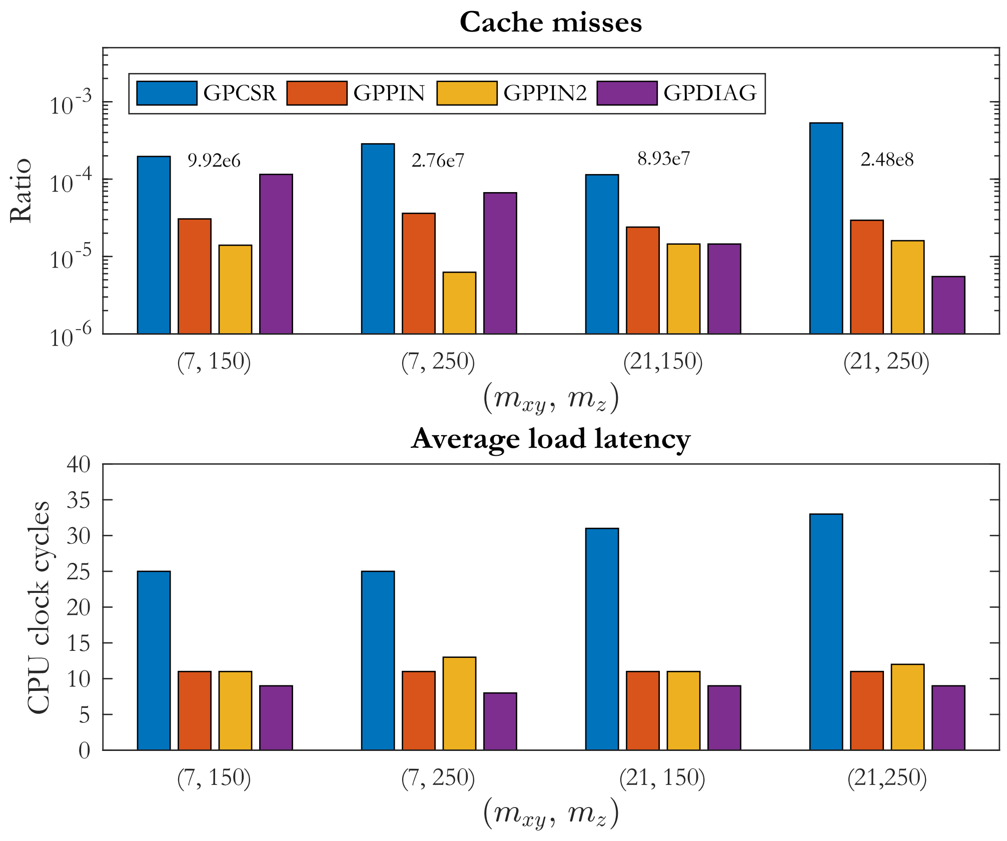


Figure 5.8: Cache miss ratio and average load latencies for 1000 SpMVs, four extreme parameter choices and 14 OpenMP threads.

This Figure shows the demerits of exceptionally large matrix data. The cache miss ratios and the average latencies do not hold a candle to those of the previous chapter, where a more economic format was introduced. However, since this chapter has emphasized on facilitating unit stride processing, the benefit of it becomes apparent. Even though all but the last algorithm proposed in this chapter are based on CSR methods with improved access patterns, the cache miss ratio and load latency plummet in comparison to those of GPCSR. Exceptional is the cache miss ratio behavior of GPDIAG, which only seems to improve for larger sizes, which explains the improving relative execution times in the benchmarks in the previous sections.

Chapter 6

Parallelizing the IDR method

In the previous chapters, parallel SpMV algorithms were presented. Naturally these are vital for any parallel linear solver. After all, each iteration step of any linear solver will cost at least one (sparse) matrix-vector product, and the time spent on computing these products forms a vast majority of the total time for this particular system, compared to other steps involved in linear solving, e.g. vector addition, vector scaling and inner products.

This chapter will account for incorporating these parallel algorithms in a linear solving method, as well as parallelizing the other solving steps. The choice of Krylov subspace solver falls to IDR(s) for larger or ill-conditioned systems encountered during the time spent at ASML. This is because it is a Krylov-type method for general matrices, i.e. it does not assume matrix properties like positive-(semi)definiteness or symmetry, and less auxiliary matrices and vectors are used compared to other Krylov-type methods, like BiCG-STAB or GMRES [14], making it memory-friendly.

Originally proposed by Sonneveld in 1980 as the *Induced Dimension Reduction* method, it has been elaborated by Sonneveld and Van Gijzen [14] and in the process baptized as IDR(s), where s denotes the algorithm's parameter. A variant (IDR(s)-biortho) recently proposed by its inventors has been shown to compete with and often outperform the original (distinguished as IDR(s)-proto) and BiCG-STAB [13], since it uses less operations and has superior numerical stability. Therefore the thesis focuses on building and benchmarking a parallel version of this linear solving algorithm.

6.1 IDR as Krylov subspace solver

Krylov-type methods are iterative solving algorithms for linear systems $\mathbf{Ax} = \mathbf{b}$. In this work, $\mathbf{A} = \mathbf{C} - \mathbf{GM}$, as used in the previous chapters. This matrix, as well as right-hand side $\mathbf{b}$, follow from the discretization (Chapter 2). A solution $\mathbf{x}^* \in \mathbb{C}^n$ to the system is desired.

Krylov-type methods, given an initial guess $\mathbf{x}^{(0)}$, produce iterates $\mathbf{x}^{(k)}$ satisfying

$$\mathbf{r}^{(k)} = \mathbf{b} - \mathbf{Ax}^{(k)} \in \mathcal{K}^k(\mathbf{A}, \mathbf{r}^{(0)}) := \langle \mathbf{r}^{(0)}, \mathbf{Ar}^{(0)}, \mathbf{A}^2\mathbf{r}^{(0)}, \dots, \mathbf{A}^{\mathbf{k}}\mathbf{r}^{(0)} \rangle, \tag{6.1}$$

where $\mathcal{K}^k$ denotes the Krylov subspace. A Krylov-type method then updates $\mathbf{x}^{(k)}$ by adding a vector from this subspace. Some methods update the iterate such that it minimizes the norm of a vector. However, this is not always possible, as it depends on specific properties (symmetry, positive-(semi)definiteness).

Definition 6.1. *Let a positive-definite matrix* $\mathbf{A} \in \mathbb{C}^{n\times n}$ *be given, i.e.* $0 < \mathrm{Re}(\mathbf{v}^H\mathbf{A}\mathbf{v})$ *for all* $\mathbf{v} \in \mathbb{C}^n$, $\mathbf{v} \neq \mathbf{0}$. *Then the vector norm* $||\cdot||_{\mathbf{A}}$*, induced by* $\mathbf{A}$*, is given by*

$$||\mathbf{v}||_{\mathbf{A}} = \sqrt{\mathrm{Re}(\mathbf{v}^H\mathbf{A}\mathbf{v})}.$$

Where the CG family of solvers minimizes $||\mathbf{x}^* - \mathbf{x}^{(k)}||_{\mathbf{A}}$ over $\mathcal{K}^k$ and GMRES minimizes $||\mathbf{x}^* - \mathbf{x}^{(k)}||_2$ over $\mathcal{K}^k$ [12], IDR(s) is based on the following theorem [13, 14], which allows the method to force residuals into a subspace of decreasing dimension.

Definition 6.2. *Let a subspace* $U \subset \mathbb{C}^n$ *and a matrix* $\mathbf{A}^{n\times n}$ *be given. Then* U *is* $\mathbf{A}$*-invariant if and only if*

$$\mathbf{A}\mathbf{u} \in U,$$

for all $\mathbf{u} \in U$.

Theorem 6.1 (IDR)**.** *Let* $\mathbf{A}$ *be a matrix in* $\mathbb{C}^{n\times n}$, $\mathbf{v}$ *any non-zero vector in* $\mathbb{C}^n$ *and define* $\mathcal{G}_0 := \mathcal{K}^n(\mathbf{A}, \mathbf{v})$. *Let also* $\mathcal{S}$ *be a proper subspace of* $\mathbb{C}^n$ *such that* $\mathcal{S}$ *and* $\mathcal{G}_0$ *do not share a non-trivial* $\mathbf{A}$*-invariant subspace. Now define a sequence of subspaces* $\mathcal{G}_j$ *with* $j = 1, 2, \ldots$ *as*

$$\mathcal{G}_j = (\mathbf{I} - \omega_j\mathbf{A})(\mathcal{G}_{j-1} \cap \mathcal{S}),$$

where the ω_j *are non-zero scalars. Then*

(i) $\mathcal{G}_j \subset \mathcal{G}_{j-1}$ *for all* $j = 1, 2, \ldots$.

(ii) $\dim \mathcal{G}_j = 0$ *for some* $j \leq n$.

Proof. Since $\mathcal{G}_0$ equals the full Krylov space $\mathcal{K}^n(\mathbf{A}, \mathbf{v})$,

$$\mathcal{G}_1 = (\mathbf{I} - \omega_1\mathbf{A})(\mathcal{G}_0 \cap \mathcal{S}) \subset (\mathbf{I} - \omega_1\mathbf{A})\mathcal{G}_0 \subset \mathcal{G}_0.$$

Assume $\mathcal{G}_j \subset \mathcal{G}_{j-1}$ for some $j > 0$, and let $\mathbf{u} \in \mathcal{G}_{j+1}$. Then there exists a $\mathbf{w} \in \mathcal{G}_j \cap \mathcal{S}$ such that $\mathbf{u} = (\mathbf{I} - \omega_{j+1}\mathbf{A})\mathbf{w}$. Then by $\mathcal{G}_j \subset \mathcal{G}_{j-1}$, also $\mathbf{w} \in \mathcal{G}_{j-1} \cap \mathcal{S}$, i.e. $(\mathbf{I} - \omega_j\mathbf{A})\mathbf{w} \in \mathcal{G}_j$. Hence, $\mathbf{A}\mathbf{w} \in \mathcal{G}_j$, and thus it follows that $(\mathbf{I} - \omega_{j+1}\mathbf{A})\mathbf{w} = \mathbf{u} \in \mathcal{G}_j$. Conclude that $\mathcal{G}_{j+1} \subset \mathcal{G}_j$, and that by induction (i) has been shown.

From $\mathcal{G}_{j+1} \subset \mathcal{G}_j$ one can distinguish two cases: $\dim \mathcal{G}_{j+1} < \dim \mathcal{G}_j$ and $\dim \mathcal{G}_{j+1} = \dim \mathcal{G}_j$. Consider the latter of the two cases. Note that it can solely occur if $\mathcal{G}_j \cap \mathcal{S} = \mathcal{G}_j$, otherwise $\dim \mathcal{G}_j \cap \mathcal{S} < \dim \mathcal{G}_j$ and therefore $\dim \mathcal{G}_{j+1} < \dim \mathcal{G}_j$. Deduce from $\mathcal{G}_j \cap \mathcal{S} = \mathcal{G}_j$ that $\mathcal{G}_j \subset S$, consequently $\mathcal{G}_{j+1} = (\mathbf{I} - \omega_{j+1}\mathbf{A})(\mathcal{G}_j \cap \mathcal{S}) = (\mathbf{I} - \omega_{j+1}\mathbf{A})\mathcal{G}_j$, and finally that $\mathcal{G}_j$ is invariant under $\mathbf{A}$. Since $\mathcal{G}_j \subset S$ and $\mathcal{G}_j \subset \mathcal{G}_0$ and that $\mathcal{G}_0$ cannot share a non-trivial $\mathbf{A}$-invariant subspace with $\mathcal{S}$, $\mathcal{G}_j = \{\mathbf{0}\}$ follows.

In total, the second case can only occur if $\dim \mathcal{G}_j = 0$. Therefore, the dimension is reduced with each step (the first case described above) until $\dim \mathcal{G}_j = 0$. By

$$\dim \mathcal{K}^n(\mathbf{A}, \mathbf{v}) = \dim\langle \mathbf{v}, \mathbf{A}\mathbf{v}, \mathbf{A}^2\mathbf{v}, \dots, \mathbf{A}^k\mathbf{v}\rangle \leq n,$$

the existence of a $j \leq n$ such that $\dim \mathcal{G}_j = 0$ follows. □

The method's parameter s equals the co-dimension of $\mathcal{S}$, i.e. $\dim \mathcal{S} = n - s$.

Krylov-type methods use a recurrence to iterate. From (6.1) it follows that $\mathbf{r}^{(k)}$ can be expressed as $\Phi_k(\mathbf{A})\mathbf{r}^{(0)}$, where Φ_k is a complex polynomial of degree k. From this polynomial sequence $(\Phi_0, \Phi_1, \dots, \Phi_k, \dots)$ describing the residuals in the Krylov subspaces, a sequence for $\mathbf{x}^{(k)}$ can be found with

$$\mathbf{A}(\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}) = \mathbf{r}^{(k)} - \mathbf{r}^{(k+1)} = \big(\Phi_k(\mathbf{A}) - \Phi_{k+1}(\mathbf{A})\big)\mathbf{r}^{(0)}. \tag{6.2}$$

Therefore, the recurrences of a Krylov-type method are of the form

$$\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - \alpha\mathbf{A}\mathbf{v}^{(k)} - \sum_{l=1}^{\hat{l}} \gamma_l \left(\mathbf{r}^{(k-l+1)} - \mathbf{r}^{(k-l)}\right), \tag{6.3}$$

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha\mathbf{v}^{(k)} - \sum_{l=1}^{\hat{l}} \gamma_l \left(\mathbf{x}^{(k-l+1)} - \mathbf{x}^{(k-l)}\right), \tag{6.4}$$

with $\mathbf{v}^{(k)} \in \mathcal{K}^k(\mathbf{A}, \mathbf{r}^{(0)}) \setminus \mathcal{K}^{k-1}(\mathbf{A}, \mathbf{r}^{(0)})$. Here, $\hat{l}$ is the depth of the recurrence. This recurrence depth can be constant (short recurrence) or equal to k (long recurrence). Long recurrence methods usually converge fast in terms of the amount of iterations, yet to implement such a method a vector needs to be additionally stored with each iteration. Long recurrence methods like GMRES are therefore less memory-friendly. Consequently, this renders short recurrence methods more attractive to implement.

The IDR(s) recurrence follows from the application of Theorem 6.1 on the recurrences (6.3) and (6.4). This linear solving method produces residuals that are forced to be in the subspaces $\mathcal{G}_j$, where j increases (though not strictly) with k. Recall from the theorem that

$$\mathbf{r}^{(k+1)} \in \mathcal{G}_{j+1} \Leftrightarrow \mathbf{r}^{(k+1)} = (\mathbf{I} - \omega_{j+1}\mathbf{A})\,\mathbf{v}^{(k)},$$

with $\mathbf{v}^{(k)} \in \mathcal{G}_j \cap \mathcal{S}$. Choose

$$\mathbf{v}^{(k)} = \mathbf{r}^{(k)} - \sum_{l=1}^{\hat{l}} \gamma_l \left(\mathbf{r}^{(k-l+1)} - \mathbf{r}^{(k-l)}\right), \tag{6.5}$$

to find the Krylov recurrence

$$\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - \omega_{j+1}\mathbf{A}\mathbf{v}^{(k)} - \sum_{l=1}^{\hat{l}} \gamma_l \left(\mathbf{r}^{(k-l+1)} - \mathbf{r}^{(k-l)}\right).$$

Since $\mathcal{S}$ has dimension $n - s$, $\mathcal{S}$ can be expressed as a null-space, i.e. $\mathcal{S} = \mathcal{N}(\mathbf{P}^H)$, with $\mathbf{P} \in \mathbb{C}^{n\times s}$ of rank s, which is how $\mathcal{S}$ is digitally represented in the algorithm [13, p. 5:10]. Since $\mathbf{v}^{(k)} \in \mathcal{S}$,

$$\mathbf{P}^H\mathbf{v}^{(k)} = \mathbf{0}. \tag{6.6}$$

Substitute (6.5) in (6.6) to see that

$$\mathbf{P}^H\mathbf{v}^{(k)} = \sum_{l=0}^{\hat{l}} \beta_l \mathbf{P}^H\mathbf{r}^{(k-l)} = \mathbf{0},$$

where $\beta_0 = 1 - \gamma_1$, $\beta_l = \gamma_l - \gamma_{l+1}$ for $l = 1, 2, \ldots, \hat{l} - 1$, and $\beta_{\hat{l}} = \gamma_{\hat{l}}$. This yields a small $s \times \hat{l}$ linear system for the $\hat{l}$ coefficients γ_l, which can only possibly have a unique solution if $\hat{l} = s$. Conclude that IDR(s) is a short recurrence method.

6.2 Adaptations for the linear system

In order to implement IDR(s)-biortho for this particular linear system, there are some adaptations to be made. These adaptations partially explain the source code in Appendix A. Refer to that appendix for a more technical explanation on the source.

6.2.1 Input arguments

First, implementations of any linear solver generally only take one matrix, **A**, as input argument (a possible preconditioner omitted). Since **A** is not computed explicitly, the adapted solver must have three matrix input arguments: one each for **C**, **M** and **G**. To have a deeper understanding of the time proportion used to compute the matrix-vector product with **A**, each of the three matrix SpMVs output timings will be collected in the process.

Second, the discretization parameters m_{xy} and m_z must be passed to the solver, since they have to be passed on to, among other things, the SpMV algorithms.

6.2.2 SpMV algorithm integration

Naturally the highest performing algorithms of the previous two chapters will be integrated in the custom IDR(s) implementation, and must replace the application of **A** found in the standard solver. For all problem sizes (except for $m_{xy} = 7$ perhaps), it is trivial to choose CMPT3 (Chapter 4) and GPDIAG (Chapter 5). These will therefore be integrated. Caveat: the input for the matrix-vector product must be permuted according to that of Section 4.3.5 to allow CMPT3 execution. Since all of the SpMV algorithms output vectors without permutation, no further SpMV algorithm integrating steps need to be done, except for creating auxiliary vectors to store the intermediate results: the products with **C**, **M** or **G** and permuted input vectors.

The performance will be compared against a variant which uses CMPCSR and GPCSR, to properly indicate the gain in performance.

6.2.3 Terminology alignment

Most mathematicians argue that an 'iteration' is defined as going through the content of the WHILE-loop, since the residuals $\mathbf{r}^{(k)}$ obtained at the end of the WHILE-loop content are in the Krylov subspace $\mathcal{K}^k$. This definition implies that every IDR(s) iteration costs $s + 1$ multiplications with $\mathbf{A}$, which is a natural way to describe this linear solver: the higher s is set, the longer it takes to compute an iterate in $\mathcal{K}^k$, because the s vectors found in each iteration are orthogonalized. However, the resulting iterate will generally be closer to the solution than those for lower s. During the time spent at ASML as well as in Sonneveld and Van Gijzen's IDR(s) MATLAB script, the term 'iteration' was used to describe the time between two modifications in the iterate $\mathbf{x}^{(k)}$, which happens $s + 1$ times in the content of the WHILE-loop. When looking closely at the algorithm in [13, p. 5:10], one can see that this is similar to counting the amount of multiplications with $\mathbf{A}$. This definition implies that each walkthrough of the WHILE-loop content consists of $s + 1$ iterations: one occurs inside the FOR-loop, looping s times, and one occurs outside. To tackle this ambiguity, the first definition of 'iteration' will be maintained in this work. Yet in the implementation the multiplications with $\mathbf{A}$ will be counted, as well as bounded by an input parameter which specifies the maximum number of matrix-vector products with $\mathbf{A}$ that are allowed before exiting. This is preferable because a multiplication with $\mathbf{A}$ pairs with a modification in the result vector and the residual. Since afterwards the relative residual norm is checked for tolerance (which could invoke a loop break-out), this allows for a more hands-on control on when the solver terminates. Since s is known, the amount of WHILE-loop walkthroughs can be calculated if the solver prints the amount of multiplications with $\mathbf{A}$ after exiting.

6.3 Performance benchmarks

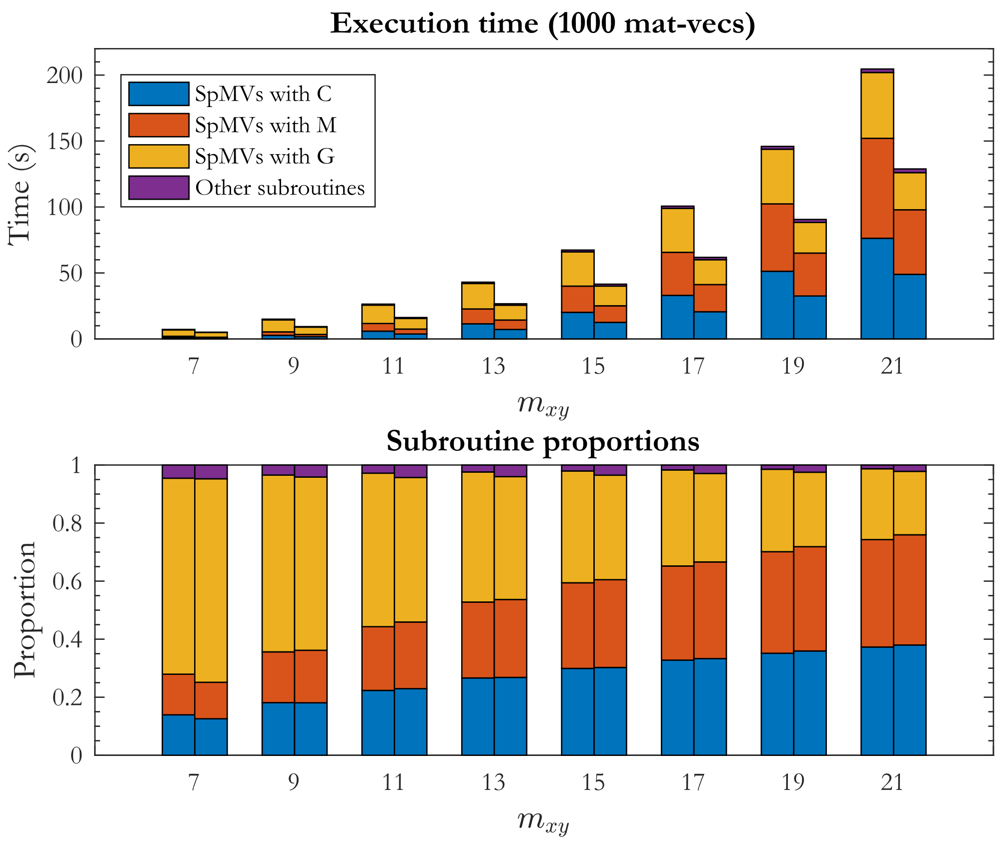


Figure 6.1: Execution time benchmark and subroutine proportions for IDR(6) obtained on one Intel® Xeon® E5-2660 v4 CPU, with $m_z = 175$ and 14 OpenMP threads, using CMPCSR-GPCSR (left stacks) and CMPT3-GPDIAG (right stacks).

First, from Figure 6.1 it can be concluded that when the tailored SpMV algorithms are used, as opposed to the CSR algorithms, only 60-65% of the time is needed to run IDR(6) up to the same amount of iterations. Second, the proportions on the bottom tell that the matrix-vector products with **A** occupy at least 95% of the IDR(6) total iteration time. This is proportionally large, yet it can be explained by the fact that **A** is completely dense, and that multiplications with it costs three SpMVs each. Third, the sparsity difference between **C** (or **M**) and **G** is on display. Since **C** and **M** have sparsity $\frac{1}{m_z}$ and **G** has sparsity $\frac{1}{m_{xy}^2}$, one expects the time proportions for **C** and **M** to increase, which is confirmed by the figure above. Note that for $m_{xy} = 17$, the SpMV time for the three matrices are approximately equal. This is curious, because equating the two sparsities (which effectively means $\text{nnz}(\mathbf{C}) = \text{nnz}(\mathbf{G})$) yields $m_{xy}^2 = m_z = 175$, which would imply $m_{xy} \approx 13$. Conclude that the SpMVs with **C** and **M** are proportionally faster, which once more displays the benefit of the economic matrix format imposed in Chapter 4.

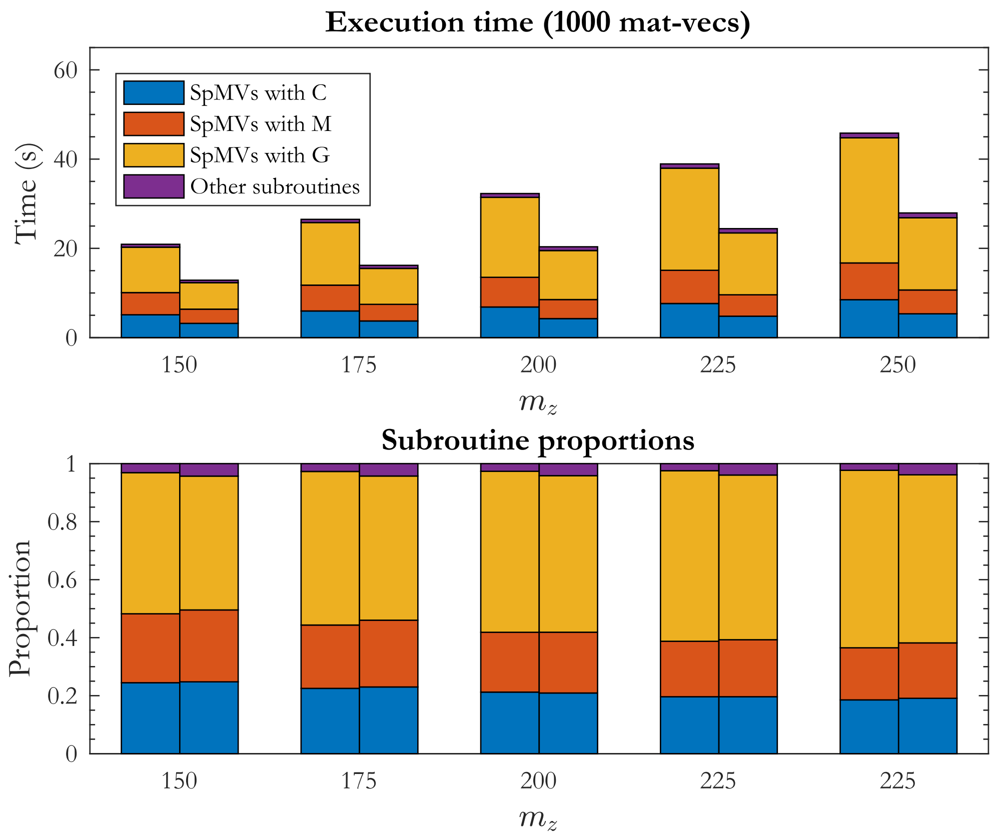


Figure 6.2: Execution time benchmark and subroutine proportions for IDR(6) obtained on one Intel® Xeon® E5-2660 v4 CPU, with $m_{xy} = 11$ and 14 OpenMP threads, using CMPCSR-GPCSR (left stacks) and CMPT3-GPDIAG (right stacks).

As already seen in the previous chapters, varying the parameter m_z does not result in erratic behavioral differences in execution time. The proportions also differ less than those of the previous benchmarks. This is due to different dependencies. For **G**, the proportions increase because its nonzero count has a second order dependency on m_z, while **C** and **M** only have a linear dependency on this parameter, as opposed to the fourth and second order encountered in the previous benchmark.

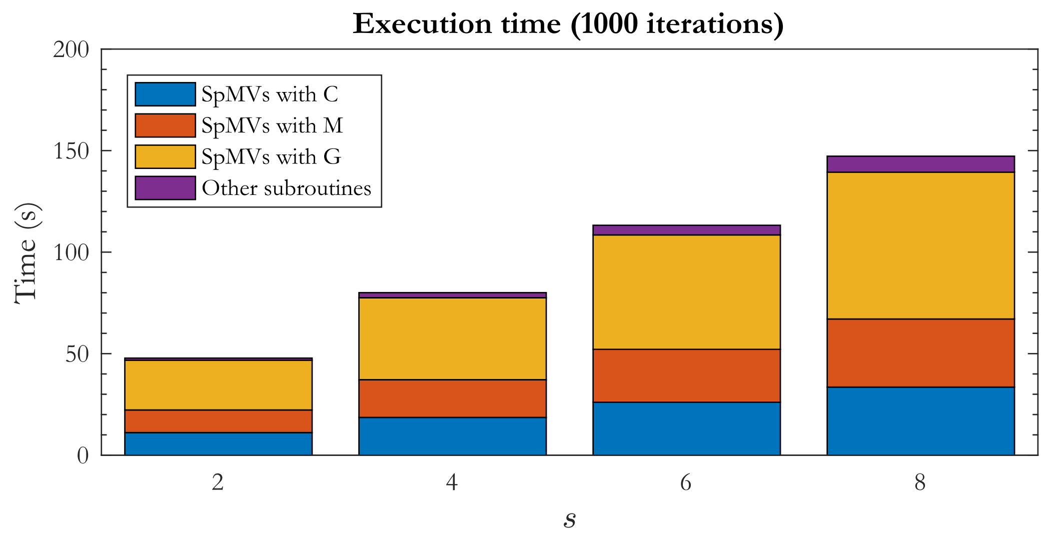


Figure 6.3: Execution time benchmark for IDR(s) obtained on one Intel® Xeon® E5-2660 v4 CPU, with $m_{xy} = 11$, $m_z = 175$ and 14 OpenMP threads, using the CMPT3-GPDIAG solver.

Note that in Figure 6.3 the time for one thousand IDR(s) iterations is measured, as opposed to the time for one thousand matrix-vector products with $\mathbf{A}$. Conclude that $1000(s + 1)$ products with $\mathbf{A}$ are performed. The SpMV execution times (and consequently also the products with $\mathbf{A}$) display linear growth with s, which is expected, given how many SpMVs are performed. The bottom graph indicates how the proportion of the other IDR(s) steps grow with s. The other subroutines take a time of order s^2n per IDR(s) iteration, because each new basis vector for the space $\mathcal{G}_j$ must be orthogonalized against previously found basis vectors in that iteration.

Chapter 7

Conclusions and recommended research

7.1 Conclusions

Since this thesis concerns parallel linear solving, we first paid attention to the linear systems arising from the particular computational problem (in this case the time-harmonic Maxwell's equations), as well as hardware-efficient parallelization of linear algebra subroutines, e.g. vector addition, inner products and the standard SpMV. Knowing that the three SpMVs belonging to this linear system take the most time to compute when linearly solving with IDR(s), improvements were suggested on standard SpMV algorithms to reduce this computation time. In the previous chapter, we fitted IDR(s) with the novel SpMV methods, which has effectively reduced the iteration time by about 35-40% compared to the solver with standard SpMV algorithms.

The essence of this work is therefore that one should always try to find structures and other dependencies in an algorithm and its input, and exploit those by using techniques such as those presented in this work, in order to realize parallel implementations specifically designed for those particular structures. The speed gain and scalability of the imposed algorithms can generally be explained by three algorithm design techniques.

First, efficient parallelization is possible when the threads of a program can perform their operations without having dependencies in the order of execution and when the threads do not collide while outputting their results. This reduces the so-called synchronization overhead. This knowledge was applied to linear algebra routines, e.g. by using `reduction` for inner products, by letting each thread complete the inner product of an entire matrix row at a time during an SpMV step, or by using any other division of labor where mutually exclusive thread outputting addresses were ensured.

Second, in Chapter 4, memory reducing techniques were applied. In this particular case, Toeplitz diagonals were compressed to create economical sparse matrix storage formats, the benefits of which were significant, even when there still were techniques yet to be used. After all, the largest speed gain achieved in Chapter 4 with one improvement step was the difference between the CMPCSR and CMPT algorithm. Widely known to be a contributor to performance, economical use of memory shows its purpose, as once more confirmed by this thesis.

Finally, as used in both SpMV chapters, providing a linear data access pattern has proven itself worthy, especially in Chapter 5. With the knowledge of the CPU caching performed by the pre-fetcher (Section 3.2.1), facilitating unit strides in data accessing results in a significantly lower cache miss ratio, which in its turn yields a better performance. The first improvement imposed on the SpMV algorithm in Chapter 6 (GPPIN) reduced the execution time by about 40%, merely by rearranging the input vector data. This example shows large potential of this algorithm design technique.

7.2 Recommended research

Naturally, there is more to just the parallel algorithm design guidelines presented and tested in this work. Due to the given time, some topics could not be included in its scope. The following sections contain possible further improvements and suggestions which could result into even faster implementations, and which could make parallel linear solving even more worthwhile to research.

By using the standard OpenMP parallelizations (reduction, collapsing nested loops, thread-private variables), the machine instructions obtained after compiling can be considered quite generic: they do not make specific assumptions on the data or the instructions. There are some code-specific OpenMP directives and clauses [9, 10] which enable low-level optimization. For example, loops can be vectorized with OpenMP using SIMD (**S***ingle* **I***nstruction,* **M***ultiple* **D***ata*) instructions, similar to how MATLAB performs its operations: the resulting code accounts for one single instruction operating on vector data. The compiler then no longer assumes any possible dependencies on the input data, which results in more efficient code. Essentially sequential, it should be possible to combine this with parallelization: code where the iterations of a (nested) FOR-loop are divided among the threads in a team (parallelization), where each iteration can be performed with SIMD instructions (vectorization). This way, the algorithms with unit stride accessing are expected to be even faster.

If single-node SpMV execution times are still unsatisfactory, one should surely look at SpMVs divided over multiple machines if they are available. With the commercial version of PARALUTION, MPI can be used to realize this. Of course, one should design the algorithms such that communication between CPU is minimized, for better scalability. For matrices $\mathbf{C}$ and $\mathbf{M}$, this can for example be realized by sending a set of blocks $\mathbf{B}_r$ and the respective parts of the input vector to a machine. This way, no intermediate communication is needed between the machines in a cluster.

Graphics cards (GPUs) are proven to be highly efficient when it comes down to massively parallel computations. They generally have more cores and also have optimized SIMD functionality. Also, they cannot host a program, which means they can be used as an accelerator only. Of course, this requires scientific programming knowledge of yet another interface, be it CUDA, OpenCL or other, which was the reason for not implementing the imposed algorithms on the GPU.

Where the emphasis in this thesis was laid on computational efficiency, linear systems are solved in significantly less iterations with a well-chosen preconditioner. Even at the cost of one more SpMV per iteration, the total solving time can still be reduced greatly. The SpMV algorithm should however be tailored towards its structure (the benefits of which is shown in this work) and implemented according to the presented techniques, such that its execution time is minimized.

Appendix A

Source code

This appendix contains the source code for the parallel SpMV algorithms and the parallel solver with the SpMV algorithm choice. Naturally, it needs some explanation as well as a color legend, for comprehension purposes.

The PARALUTION library has its own vector object structure. For the complete object structure, please refer to [11]. The essence of the free version (single node computations) is, when programming in an IDE (Integrated Development Environment), one uses the *Local* object to create vectors and invoke their methods. The library then makes a distinction between vector objects based on what kind of processor and/or API is being used. The actual API-specific code is defined for the following objects:

- *Host* objects, for computations on the CPU hosting the application (based on OpenMP).
- *Accelerator* objects, which are parent objects for the methods for GPUs (CUDA or OpenCL) and MIC-type[1] CPUs (OpenMP). An example line of MIC-type CPUs are the Intel® Xeon Phi® processor models[2].

Since this work focuses on CPU methods only, only the Host objects were used during the project. Due to the fact that emphasis was laid on different sparse matrix formats without any complementary index lists (as the case with COO, CSR, etc.), the matrix values are contained in a vector object. Hence, an extension on the *HostVector* object (named *HostVectorExt*) was required. The *LocalVector* class also needed extension (named *LocalVectorExt*), to enable invocation of the HostVectorExt methods. The reason for this is that the Local objects contain pointers to 'Host' and 'Accelerator' objects and their methods, such that if acceleration is defined and prepared by a user, the methods will automatically invoke the correct version of the method (OpenMP, CUDA, etc.).

To prevent misuse of this hierarchy, Host and Accelerator objects can only constructed by the library itself and not directly by a programmer. The SpMV algorithms will, for these reasons, be methods of HostVectorExt. The solver however needs to construct and manipulate auxiliary vectors, which makes it a LocalVectorExt method.

[1]Many Integrated Core.

[2]Not to be mistaken with the Intel® Xeon® CPUs used in this thesis.

A.1 Chapter 4 algorithms

The following colors are used in the SpMV algorithm source:

Blue	Regular C++ words: `for`, `void`, `int`, etc.
Teal	OpenMP-specific parts: parallel region declaration etc.
Purple	Parallel loop variables used for the iteration space I (Section 3.3).
Orange	Vector input and output indices.
Red	Running output subtotals.
Brown	Fields of objects or structs.
Gray	Comments.

In the following two sections, `in` denotes the input vector and `out` denotes the output vector. Since the invocations are chosen to be of the form `mat.SpMVmethod(in,out,params)`, `this` denotes the matrix non-zeros list (it is the object with which the method is called). After the method has terminated, the `time` output variable will hold the elapsed time of the parallel region. The input variables `mxy` and `mz` are self-explanatory.

Listing A.1: CMPCSR

```
void HostVectorExt<std::complex<double> >::CMPCSR(const HostVectorExt<std::complex<double> > &in,const int mxy,const int mz,
      HostVectorExt<std::complex<double> > *out,long int *time){

    const int smxy = mxy*mxy; //Constant integer: m_{xy}^2
    int r,bi,k,i,j,s;
    std::complex<double> t; //Running subtotal, local for each thread
    _set_omp_backend_threads(this->local_backend_,3*smxy*mz);
    struct timespec lo, hi;
    clock_gettime(CLOCK_REALTIME, &lo); //Start time
    #pragma omp parallel for collapse(2) private(r,bi,k,i,j,s,t)
    for(r=0;r<mz;r++){
        for(bi=0;bi<3*smxy;bi++){
            t = std::complex<double>(0E0,0E0); //Setting the subtotal to zero
            i = 3*smxy*r+bi; //Computes global matrix row
            s = 3*smxy*i; //Computes first address for the matrix data appearing on row i
            j = 3*smxy*r; //Computes first column of row i with a non-zero
            for(k=0;k<3*smxy;k++){
                t += this->vec_[s]*in.vec_[j]; //Add product to subtotal
                j ++;
                s ++;
            }
            out->vec_[i] = t; //Output the total to the output vector
        }
    }
    clock_gettime(CLOCK_REALTIME, &hi); //End time
    *time = 1E9*(static_cast<long int>(hi.tv_sec) - static_cast<long int>(lo.tv_sec))+hi.tv_nsec-lo.tv_nsec; //Computing the
          elapsed time in ns
}
```

Listing A.2: CMPT

```
void HostVectorExt<std::complex<double> >::CMPT(const HostVectorExt<std::complex<double> > &in,const int mxy,const int mz,
      HostVectorExt<std::complex<double> > *out,long int *time){

     const int dmxy = 2*mxy-1; //Constant integer: 2m_{xy}-1
     const int smxy = mxy*mxy; //Constant integer: m_{xy}^2
     int j,s,i,r,r1,r2,s1,s2,p,q;
     std::complex<double> t; //Running subtotal, local for each thread
     _set_omp_backend_threads(this->local_backend_, 3*smxy*mz);
     struct timespec lo, hi;
     clock_gettime(CLOCK_REALTIME, &lo); //Start time
     #pragma omp parallel for collapse(4) private(j,s,i,r,r1,r2,s1,s2,p,t,q)
     for(r=0;r<mz;++r){
          for(p=0;p<3;++p){
               for(r1=0;r1<mxy;r1++){
                    for(r2=0;r2<mxy;r2++){
                         t = std::complex<double>(0E0,0E0); //Setting the subtotal to zero
                         s = dmxy*(dmxy*(3*(3*r+p))+ mxy-1-r1) + mxy-1-r2; //Computes first address for the matrix data appearing
                              on row i
                         i = mxy*(mxy*(3*r+p)+r1)+r2; //Computes global matrix row
                         j = smxy*3*r; //Computes first column of row i with a non-zero
                         for(q=0;q<3;q++){
                              for(s1=0;s1<mxy;s1++){
                                   for(s2=0;s2<mxy;s2++){
                                        t += this->vec_[s]*in.vec_[j]; //Add product to subtotal
                                        j ++;
                                        s ++;
                                   }
                                   s += mxy-1;
                              }
                              s += (mxy-1)*dmxy;
                         }
                         out->vec_[i] = t; //Output the total to the output vector
                    }
               }
          }
     }
     clock_gettime(CLOCK_REALTIME, &hi); //End time
     *time = 1E9*(static_cast<long int>(hi.tv_sec) - static_cast<long int>(lo.tv_sec))+hi.tv_nsec-lo.tv_nsec; //Computing the
            elapsed time in ns
}
```

Listing A.3: CMPT2

```
void HostVectorExt<std::complex<double> >::CMPT2(const HostVectorExt<std::complex<double> > &in,const int mxy,const int mz,
      HostVectorExt<std::complex<double> > *out,long int *time){

     const int dmxy = 2*mxy-1; //Constant integer: 2m_{xy}-1
     const int smxy = mxy*mxy; //Constant integer: m_{xy}^2
     const int sdmxy = dmxy*dmxy; //Constant integer: (2m_{xy}-1)^2
     int j,s,i,r,r1,r2,s1,s2,p;
     std::complex<double> t; //Running subtotal, local for each thread
     _set_omp_backend_threads(this->local_backend_, 3*smxy*mz);
     struct timespec lo, hi;
     clock_gettime(CLOCK_REALTIME, &lo); //Start time
     #pragma omp parallel for collapse(4) private(j,s,i,r,r1,r2,s1,s2,p,t)
     for(r=0;r<mz;++r){
          for(p=0;p<3;++p){
               for(r1=0;r1<mxy;r1++){
                    for(r2=0;r2<mxy;r2++){
                         t = std::complex<double>(0E0,0E0); //Setting the subtotal to zero
                         s = 3*(sdmxy*(3*r+p)+dmxy*(mxy-1-r1) + mxy-1-r2); //Computes first address for the matrix data appearing
                              on row i
                         i = mxy*(mxy*(3*r+p)+r1)+r2; //Computes global matrix row
                         j = smxy*3*r; //Computes first needed address of the input vector
                         for(s1=0;s1<mxy;s1++){
                              for(s2=0;s2<mxy;s2++){
                                   t += this->vec_[s]*in.vec_[j]; //Add product to subtotal
                                   s ++;
                                   j += smxy;
                                   t += this->vec_[s]*in.vec_[j]; //Add product to subtotal
                                   s ++;
                                   j += smxy;
                                   t += this->vec_[s]*in.vec_[j]; //Add product to subtotal
                                   s ++;
                                   j -= 2*smxy-1;
                              }
                              s += 3*(mxy-1);
                         }
                         out->vec_[i] = t; //Output the total to the output vector
                    }
               }
          }
     }
     clock_gettime(CLOCK_REALTIME, &hi); //End time
     *time = 1E9*(static_cast<long int>(hi.tv_sec) - static_cast<long int>(lo.tv_sec))+hi.tv_nsec-lo.tv_nsec; //Computing the
            elapsed time in ns
}
```

Listing A.4: CMPT3

```
void HostVectorExt<std::complex<double> >::CMPT3(const HostVectorExt<std::complex<double> > &in,const int mxy,const int mz,
      HostVectorExt<std::complex<double> > *out,long int *time){

    const int dmxy = 2*mxy-1; //Constant integer: 2m_{xy}-1
    const int smxy = mxy*mxy; //Constant integer: m_{xy}^2
    const int sdmxy = dmxy*dmxy; //Constant integer: (2m_{xy}-1)^2
    int j,s,i,r,r1,r2,s1,s2,p;
    std::complex<double> t; //Running subtotal, local for each thread
    _set_omp_backend_threads(this->local_backend_, 3*smxy*mz);
    struct timespec lo, hi;
    clock_gettime(CLOCK_REALTIME, &lo); //Start time
    #pragma omp parallel for collapse(4) private(j,s,i,r,r1,r2,s1,s2,p,t)
    for(r=0;r<mz;++r){
        for(p=0;p<3;++p){
            for(r1=0;r1<mxy;r1++){
                for(r2=0;r2<mxy;r2++){
                    t = std::complex<double>(0E0,0E0); //Setting the subtotal to zero
                    s = 3*(sdmxy*(3*r+p)+dmxy*(mxy-1-r1) + mxy-1-r2); //Computes first address for the matrix data appearing
                         on row i
                    i = mxy*(mxy*(3*r+p)+r1)+r2; //Computes global matrix row
                    j = 3*smxy*r; //Computes first needed address of the input vector
                    for(s1=0;s1<mxy;s1++){
                        for(s2=0;s2<mxy;s2++){
                            t += this->vec_[s]*in.vec_[j]; //Add product to subtotal
                            s ++;
                            j ++;
                            t += this->vec_[s]*in.vec_[j]; //Add product to subtotal
                            s ++;
                            j ++;
                            t += this->vec_[s]*in.vec_[j]; //Add product to subtotal
                            s ++;
                            j ++;
                        }
                        s += 3*(mxy-1);
                    }
                    out->vec_[i] = t; //Output the total to the output vector
                }
            }
        }
    }
    clock_gettime(CLOCK_REALTIME, &hi); //End time
    *time = 1E9*(static_cast<long int>(hi.tv_sec) - static_cast<long int>(lo.tv_sec))+hi.tv_nsec-lo.tv_nsec; //Computing the
         elapsed time in ns
}
```

A.2 Chapter 5 algorithms

Listing A.5: GPCSR

```
void HostVectorExt<std::complex<double> >::GPCSR(const HostVectorExt<std::complex<double> > &in, const int mxy, const int mz,
      HostVectorExt<std::complex<double> > *out, long int *time){

    const int smxy = mxy*mxy; //Constant integer: distance between diagonals
    const int dm = 3*mz; //Constant integer: amount of diagonals
    int bi,bj,k,s,i,j;
    std::complex<double> t; //Running subtotal, local for each thread
    _set_omp_backend_threads(this->local_backend_, smxy*dm);
    struct timespec lo, hi;
    clock_gettime(CLOCK_REALTIME, &lo); //Start time
    #pragma omp parallel for collapse(2) private(bi,bj,k,s,i,j,t)
    for(bi=0;bi<dm;++bi){
        for(bj=0;bj<smxy;++bj){
            t = std::complex<double>(0E0,0E0); //Setting the subtotal to zero
            i = bi*smxy+bj; //Computing output address (matrix row)
            s = dm*i; //Computing first needed matrix non-zeros address
            j = bj; //Computing first needed input vector adres
            for(k=0;k<dm;++k){
                t += this->vec_[s]*in.vec_[j]; //Adding product to subtotal
                s ++;
                j += smxy;
            }
            out->vec_[i] = t; //Output the total to the output vector
        }
    }
    clock_gettime(CLOCK_REALTIME, &hi); //End time
    *time = 1E9*(static_cast<long int>(hi.tv_sec) - static_cast<long int>(lo.tv_sec))+hi.tv_nsec-lo.tv_nsec; //Computing the
         elapsed time in ns
}
```

Listing A.6: GPPIN

```
void HostVectorExt<std::complex<double> >::GPPIN(const HostVectorExt<std::complex<double> > &in, const int mxy, const int mz,
      HostVectorExt<std::complex<double> > *out, long int *time){

    const int smxy = mxy*mxy; //Constant integer: distance between diagonals
    const int dm = 3*mz; //Constant integer: amount of diagonals
    int bi,bj,k,s,i,j;
    std::complex<double> t;
    _set_omp_backend_threads(this->local_backend_, smxy*dm);
    struct timespec lo, hi;
    clock_gettime(CLOCK_REALTIME, &lo); //Start time
    #pragma omp parallel for collapse(2) private(bi,bj,k,s,i,j,t)
    for(bi=0;bi<dm;++bi){
        for(bj=0;bj<smxy;++bj){
            t = std::complex<double>(0E0,0E0); //Setting the subtotal to zero
            i = bi*smxy+bj; //Computing output address (matrix row)
            s = dm*i; //Computing first needed matrix non-zeros address
            j = bj*dm; //Computing first needed input vector adres
            for(k=0;k<dm;++k){
                t += this->vec_[s] * in.vec_[j]; //Adding product to subtotal
                s ++;
                j ++;
            }
            out->vec_[i] = t; //Output the total to the output vector
        }
    }
    clock_gettime(CLOCK_REALTIME, &hi); //End time
    *time = 1E9*(static_cast<long int>(hi.tv_sec) - static_cast<long int>(lo.tv_sec))+hi.tv_nsec-lo.tv_nsec; //Computing the
         elapsed time in ns
}
```

Listing A.7: GPPIN2

```
void HostVectorExt<std::complex<double> >::GPPIN2(const HostVectorExt<std::complex<double> > &in, const int mxy, const int mz,
       HostVectorExt<std::complex<double> > *out, long int *time){

    const int smxy = mxy*mxy; //Constant integer: distance between diagonals
    const int dm = 3*mz; //Constant integer: amount of diagonals
    int bi,bj,k,s,i,j;
    std::complex<double> t;
    _set_omp_backend_threads(this->local_backend_, smxy*dm);
    struct timespec lo, hi;
    clock_gettime(CLOCK_REALTIME, &lo); //Start time
    #pragma omp parallel for collapse(2) private(bi,bj,k,s,i,j)
    for(bj=0;bj<smxy;++bj){
        for(bi=0;bi<dm;++bi){
            t = std::complex<double>(0E0,0E0); //Setting the subtotal to zero
            s = dm*(dm*bj+bi); //Computing first needed matrix non-zeros address
            j = bj*dm; //Computing first needed input vector adres
            i = smxy*bi+bj; //Computing output address (matrix row)
            for(k=0;k<dm;++k){
                t += this->vec_[s] * in.vec_[j]; //Adding product to subtotal
                s ++;
                j ++;
            }
            out->vec_[i] = t; //Output the total to the output vector
        }
    }
    clock_gettime(CLOCK_REALTIME, &hi); //End time
    *time = 1E9*(static_cast<long int>(hi.tv_sec) - static_cast<long int>(lo.tv_sec))+hi.tv_nsec-lo.tv_nsec; //Computing the
         elapsed time in ns
}
```

Listing A.8: GPDIAG

```
void HostVectorExt<std::complex<double> >::GPDIAG(const HostVectorExt<std::complex<double> > &in, const int mxy, const int mz,
      HostVectorExt<std::complex<double> > *out, long int *time){

    const int smxy = mxy*mxy; //Constant integer: distance between diagonals
    const int dm = 3*mz; //Constant integer: amount of diagonals
    const int n = smxy*dm; //Constant integer: problem size n
    int j,i,k;
    _set_omp_backend_threads(this->local_backend_, n);
    struct timespec lo, hi;
    clock_gettime(CLOCK_REALTIME, &lo); //Start time
    #pragma omp parallel for private(i,j,k)
    for(j=0;j<n;++j){
                out->vec_[j] = this->vec_[j]*in.vec_[j];
    }

    for(k=1;k<dm;k++){
        #pragma omp parallel for private(i,j,k)
        for(j=0;j<n;++j){
            i=j+k*smxy; //Computing output address (matrix row)
            if(i>=n){
                i -= n;
            }
            out->vec_[i] += this->vec_[k*n+j]*in.vec_[j];
        }
    }
    clock_gettime(CLOCK_REALTIME, &hi); //End time
    *time = 1E9*(static_cast<long int>(hi.tv_sec) - static_cast<long int>(lo.tv_sec))+hi.tv_nsec-lo.tv_nsec; //Computing the
         elapsed time in ns
}
```

A.3 Parallel linear solver

In the solver source, the words are color-coded as follows:

Blue	Regular C++ words: `for`, `void`, `int`, etc.
Teal	Parallel subroutines and setting the OpenMP backend.
Purple	Auxiliary objects used by the IDR(s) method (Chapter 6).
Orange	IDR(s) parameters: s, maximum number of iterations, etc.
Red	Running subtotals: SpMV timings, iteration counter, etc.
Brown	Fields of objects or structs.
Gray	Comments.

There are some preliminary caveats on the source below.

- In addition to matrices **G** and **M** of the linear system, there are equally named matrices used in IDR(s)-biortho [13]. The matrix data objects will be named `ccm`, `mcm` and `gdia`, while the auxiliary algorithm objects will be called `G` and `M`. This notation will also be maintained in the comments, though in actual source code, complying to the legend above, `G` and `M` can by distinguished by their purple color.

- The decision to put `P` in the input of the void, roots in debugging motives. After all, the ability to use the same space $\mathcal{S}$ in both the C++ and MATLAB versions helped tracking down any programming mistakes.

- Some subroutines are not implemented in parallel (un-colored). Almost all of those concern the computation/retrieval of scalars. The lower triangular solve (void `SolveMf`) and setting `M` equal to the identity matrix (void `MI`) are done sequentially because `M` has such a small size ($s \times s$), performance will not benefit from parallel execution. Additionally the forward substitutions used in lower triangular system solving already hint to sequential execution.

- The auxiliary algorithm matrix objects (`P`, `G`, `U`, etc.) are also flattened into a vector object with column-major ordering to facilitate column copying, except for the lower triangular `M`, which has row-major ordering to facilitate the forward subtitution solving.
- Note that the auxiliary algorithm vector `t` is not used in the source, as other unused vectors can be used to store `t` temporarily. In the source, `gx` is used.
- The running subtotals (iteration count, SpMV times, initialization time and iteration time) are printed at the end of the void, but these print commands are omitted in this listing.
- By means of readability, the comments will maintain a notation strongly similar to MATLAB syntax.

Listing A.9: Non-preconditioned parallel IDR(s) with the custom SpMV algorithms

```
void LocalVectorExt<std::complex<double> >::IDRs(
      const int s, //IDR parameter
      const double kappa, //Threshold. If cos(angle)<kappa: perform a larger residual step, the angle being between Ar and r.
           Defaults to 0.7
      const LocalVectorExt<std::complex<double> > &ccm, //Matrix data for C in the CMPT3 Toeplitz compression format
      const LocalVectorExt<std::complex<double> > &mcm, //Matrix data for M in the CMPT3 Toeplitz compression format
      const LocalVectorExt<std::complex<double> > &gdia, //Matrix data for G in the concatenated diagonal format
      const LocalVectorExt<std::complex<double> > &P, //Defines the space S = ker(P^H)
      const int mxy, //Discretization accuracy m_{xy}
      const int mz, //Discretization accuracy m_z
      const LocalVectorExt<std::complex<double> > &b, //Right-hand side of the system
      const double tol, //Tolerance for the relative residual norm ||r||/||b||
      const int maxmatvec //Maximum number of mat-vecs allowed
      ){

    struct timespec loiter, hiiter, loinit, hiinit;
    clock_gettime(CLOCK_REALTIME, &loinit); //Start time of the initialization
    const int n = 3*mxy*mxy*mz; //Constant integer for the problem size n
    const double bn = sqrt(b.vector_host_->Dot(*b.vector_host_).real()); //Calculate ||b||
    _set_omp_backend_threads(this->local_backend_, n);
    long int t;
    double ct=double(0E0),mt=double(0E0),gt=double(0E0),permt=double(0E0); //Initialize timing subtotals
    double itert,initt;
    std::complex<double>  thr, tht, rhr; //Declare variables for t^Hr, t^Ht and r^Hr

    LocalVectorExt<std::complex<double> > r_,v_,G_,U_,f_,M_,c_,cmx_,gx_,inp_; //Declaring auxiliary algorithm data: r,v,G,U,f,
         M,c,etc.
    r_.Allocate("r",n); //Allocating space for the auxiliary algorithm data
    v_.Allocate("v",n);
    G_.Allocate("G",n*s);
    U_.Allocate("U",n*s);
    f_.Allocate("f",s);
    M_.Allocate("M",s*s);
    c_.Allocate("M",s);
    cmx_.Allocate("cx",n);
    gx_.Allocate("mx",n);
    inp_.Allocate("inp",n);
    LocalVectorExt<std::complex<double> > *r=&r_,*v=&v_,*G=&G_,*U=&U_,*f=&f_,*M=&M_,*c=&c_,*cmx=&cmx_,*gx=&gx_,*inp=&inp_; //
         Creating pointers

    std::complex<double>  alpha; //Declaring algorithm auxiliary complex numbers, setting omega=1
    std::complex<double>  beta;
    std::complex<double>  omega = std::complex<double> (1.0);
    std::complex<double>  rho;
    std::complex<double>  mu;
    double rhoabs; //Variable for the modulus of rho
    M->vector_host_->MI(s); //Setting M to the identity matrix
    int matvec = 0;
    r->vector_host_->Copy(*b.vector_host_); //Start with the zero vector as initial guess => r = b
    double rn = bn; //Norm of r equals norm of b
    clock_gettime(CLOCK_REALTIME, &hiinit); //End time of the initialization
    clock_gettime(CLOCK_REALTIME, &loiter); //Start time of the iteration process
    while(rn > tol * bn && matvec < maxmatvec){ //Start of while-loop
        f->vector_host_->PHr(*P.vector_host_,*r->vector_host_,n,s); //Calculate f = P'*r
        for(int k=0;k<s;k++){ //Start of s-loop
            c->vector_host_->SolveMf(*M->vector_host_,*f->vector_host_,k,s); //Sequential solve low. triang. system M*c = f
            v->vector_host_->GetV(*r->vector_host_,*G->vector_host_,*c->vector_host_,k,s); //Calculate v = r - Gc
            U->vector_host_->GetUk(omega, *v->vector_host_,*c->vector_host_,k,s,n); //Sets U(:,k) = U*c + omega*v
            inp->vector_host_->LoadU(*U->vector_host_,k,mxy,mz); //Copy U(:,k) to inp under the CMPT3-permutation
            mcm.vector_host_->CMPT3(*inp->vector_host_,mxy,mz,cmx->vector_host_,&t); //Computes Mcm*U(:,k)
            mt += t/1E9;
            gdia.vector_host_->GPDIAG(*cmx->vector_host_,mxy,mz,gx->vector_host_,&t); //Computes Gdia*Mcm*U(:,k)
            gt += t/1E9;
            ccm.vector_host_->CMPT3(*inp->vector_host_,mxy,mz,cmx->vector_host_,&t); //Computes Ccm*U(:,k)
            ct += t/1E9;
            G->vector_host_->StoreOff(*cmx->vector_host_,*gx->vector_host_,k,n); //Store G(:,k) = (Ccm-Gdia*Mcm)*U(:,k)
            for(int l=0;l<k;l++){
                alpha = P.vector_host_->PHG(*G->vector_host_,n,l,k); //Computes alpha = P(:,l) dot G(:,k)
                alpha /= M->vector_host_->LoadMu(l, l, s); //Computes alpha = alpha/M(l,l)
                G->vector_host_->AlphaSub(alpha,k,l,n); //Sets G(:,k) = G(:,k) - alpha * G(:,l)
                U->vector_host_->AlphaSub(alpha,k,l,n); //Sets U(:,k) = U(:,k) - alpha * U(:,l)
            }
            for(int l=k;l<s;l++){
                mu = P.vector_host_->PHDotG(*G->vector_host_,n,l,k); //Computes = P(:,l) dot G(:,k)
                M->vector_host_->StoreMu(l,k,s,mu); //Sets M(l,k) = mu
            }
            beta = f->vector_host_->GetBeta(*M->vector_host_,s,k); //Computes beta = f(k)/M(k,k)
            r->vector_host_->AddScaleGUTV(*G->vector_host_,k,-beta); //Sets r = r - beta * G(:,k)
            this->vector_host_->AddScaleGUTV(*U->vector_host_,k,beta); //Sets x = x + beta * U(:,k)
            matvec ++;
            rn = sqrt(r->vector_host_->Dot(*r->vector_host_).real()); //Computes norm(r)
            if(rn <= tol * bn || matvec == maxmatvec){ //Break out of s-loop if tolerance or maximum number of matvecs is
                 reached
                break;
            }
            f->vector_host_->UpdateF(*M->vector_host_,k,s,beta); //Sets f(i) = 0 iff i=0,..,k-1. Sets f(i) = f(i) - beta * M(i
                 ,k) otherwise.
        } //End of FOR-loop
        if(rn <= tol * bn || matvec == maxmatvec){ //Break out of while-loop if tolerance or maximum number of matvecs is
             reached
            break;
        }
        inp->vector_host_->CMPerm(*r->vector_host_,mxy,mz,&t); //Copies r to inp under the CMPT3 permutation
        permt += t/1E9;
        mcm.vector_host_->CMPT3(*inp->vector_host_,mxy,mz,cmx->vector_host_,&t); //Computes Mcm**r
        mt += t/1E9;
        gdia.vector_host_->GPDIAG(*cmx->vector_host_,mxy,mz,gx->vector_host_,&t); //Computes Gdia*(Mcm*r)
        gt += t/1E9;
        ccm.vector_host_->CMPT3(*inp->vector_host_,mxy,mz,cmx->vector_host_,&t); //Computes Ccm*r
        ct += t/1E9;
        gx->vector_host_->ScaleAdd1(std::complex<double> (-1.0),*cmx->vector_host_); //Stores t = A*r in vector gx
        thr = gx->vector_host_->Dot(*r->vector_host_); //Computes t^Hr
        tht = gx->vector_host_->Dot(*gx->vector_host_); //Computes t^Ht
        rhr = r->vector_host_->Dot(*r->vector_host_); //Computes r^Hr
        omega = thr/tht; //Sets new omega
        rho = thr/(sqrt(tht.real())*sqrt(rhr.real())); //Computes rho = (Ar,r)/(||Ar||*||r||)
        rhoabs = sqrt(rho.real()*rho.real() + rho.imag()*rho.imag()); //Computes |rho|, the angle between Ar and r
        if(rhoabs < kappa){ //If the angle is below the threshold, perform a larger residual step
            omega *= kappa/rhoabs;
        }
        this->vector_host_->ScaleAdd2(omega,*r->vector_host_); //Sets x = x + omega * r
        r->vector_host_->ScaleAdd2(-omega,*gx->vector_host_); //Sets r = r - omega * t
        rn = sqrt(r->vector_host_->Dot(*r->vector_host_).real()); //Computes new norm of r
        matvec ++;
    } //end of WHILE-loop
    clock_gettime(CLOCK_REALTIME, &hiiter); //End time of iteration process
    itert = static_cast<long int>(hiiter.tv_sec) - static_cast<long int>(loiter.tv_sec)+(hiiter.tv_nsec-loiter.tv_nsec)/1E9;
    initt = static_cast<long int>(hiinit.tv_sec) - static_cast<long int>(loinit.tv_sec)+(hiinit.tv_nsec-loinit.tv_nsec)/1E9;
}
```

Appendix B

Specifications of used CPUs

There is a convenient UNIX command, `lscpu`, which outputs, amongst other: the CPU model, cache sizes, base frequency, current frequency, number of cores and which core IDs belong to which socket in the machine. Also [8] provides some of these specifications. In the lowest cache level, the distinction is made between data and instructions.

	Intel® Xeon® E5-2660 v4
# cores	14
Base frequency (GHz)	2.00
Max. frequency (GHz)	3.20
L3 cache size (MiB)	35
L2 cache size (KiB)	256
L1 cache size (KiB)	32^{data}, $32^{\text{instructions}}$

Table B.1: Specifications of the used CPU models.

Bibliography

[1] George B. Arfken and Hans J. Weber. *Mathematical Methods for Physicists*. Sixth edition. Elsevier Academic Press, 2005.

[2] Rob H. Bisseling. *Parallel Scientific Computation*. First edition. New York: Oxford University Press, 2004.

[3] Ronald N. Bracewell. *The Fourier Transform and its applications*. Third edition. New York: McGraw-Hill, 1999.

[4] Yia-Chung Chang et al. “Efficient finite-element, Green’s function approach for critical-dimension metrology of three-dimensional gratings on multilayer films”. In: *Journal of the Optical Society of America A* 23.3 (2006), pp. 638–645.

[5] Ulrich Drepper. “What Every Programmer Should Know About Memory”. 2007.

[6] Liming Feng and Vadim Linetsky. “Pricing Options in Jump-Diffusion Models: an Extrapolation Approach”. In: *Operations Research* 58.2 (Apr. 2008), pp. 304–325.

[7] *FFTW website*. fftw.org.

[8] *Intel product specifications*. ark.intel.com.

[9] *OpenMP 4.5 Application Programming Interface*. openmp.org. Nov. 2015.

[10] *OpenMP 4.5.0 Application Programming Interface Examples*. openmp.org. Nov. 2015.

[11] *PARALUTION 1.1.0 User Manual*. paralution.com. Jan. 2016.

[12] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. Second edition. SIAM, 2003.

[13] Peter Sonneveld and Martin B. van Gijzen. “An Elegant IDR(s) Variant that Efficiently Exploits Bi-orthogonality Properties”. In: *ACM Transactions on Mathematical Software* 38.1 (2011), 5:1–5:19.

[14] Peter Sonneveld and Martin B. van Gijzen. “IDR(s): a Family of Simple and Fast Algorithms for Solving Large Nonsymmetric Systems of Linear Equations”. In: *SIAM Journal on Scientific Computing* 31.2 (2008), pp. 1035–1062.

[15] Andrew James Stother. “On the Complexity of Matrix Multiplication”. PhD thesis. University of Edinburgh, 2010.

[16] Gabriel Torres. *How The Cache Memory Works*. hardwaresecrets.com. 2007.

"'t Is gedaan precies."
–
Jonas Geirnaert
Kabouter Wesley