Image Processing using Graph Laplacian Operator

DAVID WOBROCK

Master Thesis in Computer Science
Date: March 22, 2018

Supervisor: Frédéric Nataf
ALPINES Team - INRIA Paris
Laboratoire Jacques-Louis Lions - Sorbonne Université

KTH Supervisor: Stefano Markidis
KTH Examiner: Erwin Laure
Swedish title: Bildbehandling med graf Laplaceoperatorn
School of Computer Science and Communication - KTH

INSA Supervisor: Christine Solnon
French title: Traitement d'image en utilisant le Laplacien de graphe
Département Informatique - INSA de Lyon

Abstract

The latest image processing methods are based on global data-dependent filters. These methods involve huge affinity matrices which cannot fit in memory and need to be approximated using spectral decomposition. The inferred eigenvalue problem concerns the smallest eigenvalues of the Laplacian operator which can be solved using the inverse iteration method which, in turn, involves solving linear systems. In this master thesis, we detail the functioning of the spectral algorithm for image editing and explore the behaviour of solving large and dense systems of linear equations in parallel, in the context of image processing. We use Krylov type solvers, such as GMRES, and precondition them using domain decomposition methods to solve the systems on high-performance clusters. The experiments show that Schwarz methods as preconditioner scale well as we increase the number of processors. However, we observe that the limiting factor is the Gram-Schmidt orthogonalisation procedure. We also compare the performances to the state-of-the-art Krylov-Schur algorithm.

Sammanfattning

De senaste bildbehandlingsmetoderna är baserade på global databeroende filter. Dessa metoder innefattar stora affinitetsmatriser som inte kan passa i minnet och måste approximeras med spektral sönderdelning. Det härledda egenvärdesproblemet gäller de minsta egenvärdena för den laplaciska operatören som kan lösas med hjälp av den inverse iterationsmetoden som i sin tur innebär att lösa linjära system. I denna masterprojekt beskriver vi hur spektralalgoritmen fungerar för bildredigering och utforskar beteendet att lösa stora och täta system av linjära ekvationer parallellt i samband med bildbehandlingVi använder Krylov-typlösare, till exempel GMRES, och förutsätter att dom använder sönderdelningsmetoder för att lösa systemen på högpresterande kluster. Experimenten visar att Schwarz-metoderna som preconditioner skala liksom vi ökar antalet processorer. Vi observerar emellertid att den begränsande faktorn är Gram-Schmidt-ortogonaliseringsproceduren. Vi jämför också prestationerna till den toppmoderna Krylov-Schur-algoritmen.

Résumé

Les méthodes récentes de traitement d'images sont basées sur des filtres globaux dépendant des données. Ces méthodes impliquent le calcul de matrices de similarités de grandes tailles qui ne peuvent pas être contenues en mémoire et nécessitent ainsi d'être approchées par une décomposition spectrale. Le problème aux valeurs propres qui en découle concerne les plus petites valeurs propres du Laplacien qui peuvent être calculées grâce à la méthode de la puissance inverse qui implique la résolution de systèmes linéaires. Dans cette thèse de master, nous détaillons le fonctionnement de l'algorithme spectral pour l'édition d'images et étudions le comportement de la résolution de systèmes d'équations linéaires sur de grandes matrices pleines en parallèle, dans le cadre du traitement d'images. Nous utilisons des solveurs de type Krylov, tel que GMRES, et les préconditionnons avec des méthodes de décomposition de domaines pour résoudre les systèmes sur des clusters haute performance. Les expériences montrent que les méthodes de Schwarz comme préconditionneur passent bien à l'échelle quand on augmente le nombre de processeurs. Toutefois, nous observons que le facteur limitant est la procédure d'orthogonalisation de Gram-Schmidt. Nous comparons également les performances avec l'algorithme de l'état de l'art Krylov-Schur.

Contents

1 Introduction **1**
- 1.1 Background . . . 1
- 1.2 Objective . . . 1
- 1.3 Related work . . . 2
  - 1.3.1 Spectral graph theory . . . 2
  - 1.3.2 Image processing - denoising . . . 3
  - 1.3.3 Linear solvers & domain decomposition methods 5
- 1.4 Delimitations . . . 7
- 1.5 Outline . . . 8

2 Image processing using the graph Laplacian operator **9**
- 2.1 Algorithm . . . 9
- 2.2 Variations . . . 15
  - 2.2.1 Sampling method . . . 16
  - 2.2.2 Affinity function . . . 17
  - 2.2.3 Graph Laplacian operator . . . 20

3 Implementation **21**
- 3.1 Algorithm details . . . 21
- 3.2 Parallel implementation . . . 22
- 3.3 Results . . . 25
  - 3.3.1 Entire matrix computation . . . 25
  - 3.3.2 Approximation computation . . . 27

4 Conclusion **36**
- 4.1 Discussions . . . 36
- 4.2 Perspectives . . . 37

Chapter 1

Introduction

1.1 Background

The talk [1] and articles [2] [3] by Milanfar, working at Google Research, about using the graph Laplacian operator for nonconformist image processing purposes awakes curiosity.

Indeed, Milanfar reports that these techniques to build image filters are used on smartphones, which implies a reasonable execution time with limited computational resources. Over 2 billion photos are shared daily on social media [1], with very high resolutions and most of the time some processing or filter is applied to them. The algorithm must be efficient to be deployed at such scale.

1.2 Objective

The aim of this degree project is not to explore and improve the state of image processing. Instead, the spectral methods used in the algorithm will be our focus point. Those will inevitably expose eigenvalue problems, which may involve solving systems of linear equations.

Concerning the challenges about solving linear systems, on one hand, the size of the systems can be large considering high-resolution images with millions of pixels, or even considering 3D images. We process huge matrices of size N^2, with N the number of pixels of the input image. On the other hand, those huge affinity matrices are dense, thus the linear systems are dense. Often, linear systems result from discretising partial differential equations (PDEs) yielding sparse ma-

trices, and therefore most linear solvers are specialised in sparse systems.

We want to explore the performance of linear solvers on dense problems, their stability and convergence. This will include preconditioning the linear systems, especially using domain decomposition methods, and analyse their behaviour on dense systems.

1.3 Related work

In this section, we start by a summary of spectral graph theory as an introduction to the project. It is followed by image processing techniques for denoising, with traditional patch-based methods and global image filters. And we finish by a quick overview of linear solvers and domain decomposition methods.

1.3.1 Spectral graph theory

Spectral graph theory has a long history starting with matrix theory and linear algebra that were used to analyse adjacency matrices of graphs. As a reminder, we define a graph as ordered pair $G = (V, E)$ with V a set of vertices linked together through E a set of edges. To represent a graph as a mathematical object, each finite graph can be represented as an adjacency matrix, in which the elements indicate whether pairs of vertices are connected through an edge or not. Spectral graph theory consists in studying the properties of graphs in relation to the eigenvalues and eigenvectors of the adjacency or Laplacian matrix.

Laplacian matrix Since the adjacency matrix of a graph only holds basic information about it, we usually augment it to the Laplacian matrix. Multiple definitions of the Laplacian matrix are given in [4] and [1], and each one has different properties. The most common ones are the normalised Laplacian and the Random Walk Laplacian. However, more convenient formulations, like the "Sinkhorn" Laplacian [5] and the re-normalised Laplacian [6], have been proposed since. A detailed summary of Laplacian matrices is given by the table 2.1.

The eigenvalues of such a matrix are called the spectrum of the graph. The second smallest eigenvalue has been called "algebraic con-

nectivity of a graph" by Fiedler [7], and is therefore also known as *Fiedler value*, because it contains interesting information about the graph. Indeed, it can show if the graph is connected, and by extending this property, we can count the number of connected components in the graph through the eigenvalues of the graph Laplacian and do graph partitioning.

The field of spectral graph theory is very broad and the eigendecomposition of graphs is used in a lot of areas. Spectral graph theory has many applications such as graph colouring, random walks and graph partitioning among others.

One of the most complete works about spectral graph theory is [4] by Fan Chung. This monograph exposes many properties of graphs, the power of the spectrum and how spectral graph theory links the discrete world to the continuous one.

The Spectral Theorem Most Laplacian definitions result in a symmetric matrix, which is a property that is particularly interesting for spectral theory because of the Spectral Theorem [8]. Let S be a real symmetric matrix of dimension n, $\Phi = [\phi_1 \phi_2 \dots \phi_n]$ the matrix of eigenvectors of S and $\forall i \in [0, n]$, let $\Pi = diag\{\lambda_i\}$ the diagonal matrix of the eigenvalues of S, then we have the eigendecomposition of S:

$$S = \Phi \Pi \Phi^T = \sum_{i=1}^{n} \lambda_i \phi_i \phi_i^T .$$

We note that the eigenvalues of S are real and that the eigenvectors are orthogonal, i.e., $\Phi^T \Phi = I$, with I the identity matrix.

The Laplacian operator is the foundation of the heat equation, fluid flow and essentially all diffusion equations. It can generally be thought that the Laplacian operator is a centre-surround average [1] of a given point. Therefore, applying it on an image results in smoothing. Generally, applying the graph Laplacian operator on an image provides useful information about it and enables possibilities of interesting image processing techniques.

1.3.2 Image processing - denoising

Background Even with high quality cameras, denoising and improving a taken picture remains important. The two main issues that have

to be addressed by denoising are blur and noise. The effect of blur is internal to cameras since the number of samples of the continuous signal is limited and it should hold the Shannon-Nyquist theorem [9], stipulating a sufficient condition on the number of samples required to discretise a continuous signal without losing information. Noise comes from the light acquisition system that fluctuates in relation to the amount of incoming photons.

To model these problems, a classical approach to formulate the deficient image considers z the clean signal vector, e a noise vector and y the noisy picture:

$$y = z + e.$$

What we want is a high-performance denoiser, capable of scaling up in relation to increasing the image size and keeping reasonable performances. The output image should come as close as possible to the clean image. As an important matter, it is now generally accepted that images contain a lot of redundancy. This means that, in a natural image, every small enough window has many similar windows in the same image.

Traditional, patch-based methods The image denoising algorithms review proposed by [9] suggests that the non-local means algorithm, compared to other reviewed methods, comes closest to the original image when applied to a noisy image. This algorithm takes advantage of the redundancy of natural images and for a given pixel predicts its value by using the pixels in its neighbourhood.

In [10], the authors propose the BM3D algorithm, a denoising strategy based on grouping similar 2D blocks of the image into 3D data arrays. Then, collaborative filtering is performed on these groups and return 2D estimates of all grouped blocks. This algorithm exposed state-of-the-art performance in terms of denoising at that time. The results are still one of the best for a reasonable computational cost.

Global filter In the last couple of years, global image denoising filters came up, based on spectral decomposition [2]. This approach considers the image as a complete graph, where the filter value of each pixel is computed by all pixels in the image. We shall go into the details of the graph representation in section 2.1. We define the result image z, W the huge data-dependent global filter of size $N \times N$, with

N the number of pixels and the input image y:

$$z = Wy.$$

To show an example of the size of the filter, a standard 10 megapixel picture will result in a filter matrix of 10^{14} elements, taking 800 TB of memory. This kind of filter is considered in this report.

Those huge matrices will need to be approximated using their eigen-decomposition. And these will need to solve systems of linear equations.

1.3.3 Linear solvers & domain decomposition methods

Solving a system of linear equations such that

$$Ax = b,$$

is often critical in scientific computing. When discretising equations coming from physics for example, a huge linear system can be obtained. Multiple methods exist to solve such systems, even when the system is large and expensive to compute. We present in the following the most used and known solvers.

Direct solvers The most commonly used solvers for systems of linear equations are direct solvers. They provide robust methods and optimal solutions to the problem. However, they can be hard to parallelise and have difficulties with large input. The most famous is the backslash operator from MATLAB which performs tests to determine which special case algorithm to use, but ultimately falls back on a LU factorisation [11]. The LU factorisation, which is a Gaussian elimination procedure, is hard to parallelise. Although, a block version of the LU factorisation exists that can be parallelised more easily. There are other parallel direct solvers, like MUMPS [12], but generally, sequential solvers reach their computational limit above 10^6 degrees of freedom in a 2D problem, and 10^5 in 3D.

Iterative solvers For larger problems, iterative methods must be used to achieve reasonable runtime performances. The two types of iterative solvers are fixed-point iteration methods and Krylov type methods. Both require only a small amount of memory and can often be

parallelised. The main drawback is that these methods tend to be less robust than direct solvers and convergence depends on the problem. Indeed, ill-conditioned input matrices will be difficult to solve correctly by iterative methods and do not necessarily converge. Generally, Krylov methods are preferred over fixed-point iteration methods. The most relevant iterative Krylov methods are the Conjugate Gradient (CG) and the Generalised Minimal Residual method (GMRES) [13] [14].

To tackle the ill-conditioned matrices problem, there is a need to precondition the initial system.

Preconditioners - Domain decomposition methods One of the ways to precondition systems of linear equations is to use domain decomposition. The idea goes back to Schwarz who wanted to solve a Poisson problem on a complex geometry. He decomposed the geometry into multiple smaller simple geometric forms, making it easy to work on subproblems. This idea has been extended and improved to propose fixed-point iterations solvers for linear systems. However, Krylov methods expose better results and faster convergence, but domain decomposition methods can in fact be used as preconditioners to the system. Famous Schwarz preconditioners are the restricted additive Schwarz method (RAS) and the additive Schwarz method (ASM). With M^{-1} the preconditioning matrix, we shall solve

$$M^{-1}Ax = M^{-1}b$$

which exposes the same solution as the original problem.

Domain decomposition methods are usually applied to solve problems of linear algebra involving partial differential equations. Solving the discretised problem leads to solving linear systems.

Our main reference will be [15] which focuses on the parallel linear iterative solvers for systems of linear equations. Domain decomposition methods are naturally parallel which is convenient for the current state of processor progress.

In general, the preconditioning matrix M is chosen as an approximated inverse of the input matrix A. Indeed, the convergence depends on the spectral properties of A and a good preconditioner will

transform the eigenvalues of A to cluster around 1 and still have the same solution. Let Ω be the entire domain and, with N the number of subdomains, $\Omega = \cup_{i=1}^{N} \Omega_i$. If the subdomains do not overlap, we have $\cap_{i=1}^{N} \Omega_i = \emptyset$. Let R_i be the restriction operator of a vector $U \in \mathbb{R}^N$ to a subdomain Ω_i, $1 \leq i \leq N$. The operator can be expressed a rectangular boolean matrix of size $\#N_i \times N$, where $\#N_i$ is the size of the ith subdomain Ω_i. Therefore, the extension operator is the transpose matrix R_i^T. The ASM preconditioner is defined as:

$$M_{ASM}^{-1} = \sum_{i=1}^{N} R_i^T (R_i A R_i^T)^{-1} R_i.$$

As we see, on each subdomain is computed the inverse of a block of the input matrix A and the results are merged back together in an additive manner. Let D_i be the partition of unity operator of the ith subdomain, a diagonal non-negative matrix of size $\#N_i \times \#N_i$, which attributes weights to the different values present on multiple subdomains. The RAS precondition is defined as:

$$M_{RAS}^{-1} = \sum_{i=1}^{N} R_i^T D_i (R_i A R_i^T)^{-1} R_i.$$

The partition of unity is used to balance the results from overlapping subdomains. In the case of non-overlapping subdomains, $D_i = I$, the identity matrix.

1.4 Delimitations

The purpose of this project, as written in the objectives, is not the image processing aspect. However, a visible result for the application of our methods is a pleasant outcome. The algorithm does not necessarily have to produce filtered images, or at least "better" images as in the input. Indeed, the concept of "better" images is not mathematically defined and requires more experience in this field. To solve the arising linear systems, we want to focus on Krylov type methods and especially domain decomposition methods for preconditioning. Exploring other solvers and preconditioning methods and their performances could be interesting, but not necessary in this thesis.

1.5 Outline

The document is organised in the following way. Chapter 2 introduces the global filter algorithm that has been studied during this project. It explains the image processing method in a general way, our adapted algorithm and then clarifies the variations that can be used. It serves as a reference to understand the algorithm and the problems that arise. Chapter 3 shows the work that has been done on the implementation side. It explains the used parallelism and exposes experimental results of this approach. Finally, we discuss the results and open perspectives for future work.

Chapter 2

Image processing using the graph Laplacian operator

Multiple image processing filters can be built using the graph Laplacian operator. As Milanfar mentions in [1] [2] [3], smoothing, deblurring, sharpening, dehazing, and other filters can be created. Laplacian operators can also be used as the basis for compression artifact removal, low-light imaging and image segmentation.

We shall consider an adapted version of the proposed algorithm in [2] to solve the eigenvalue problem by solving linear systems. Let's introduce step-by-step the algorithm, the approximations and the possible variations.

2.1 Algorithm

A global image filter consists of a function which outputs one pixel, taking all pixels as input and applying weights to them. Let z_i be the output pixel, W_{ij} the weights, y_j all input pixels and N the number of pixels in the image. We compute one output pixel with:

$$z_i = \sum_{j=1}^{N} W_{ij} y_j,$$

This means that a vector of weights exists for each pixel.

As a practical notation, we say that, with W the matrix of weights and y and z respectively the input and output images as vectors,

$$z = Wy.$$

The filter matrix W considered here is data-dependent and built upon the input image y. A more mathematical notation would consider W as a nonlinear function of the input image such as $z = W(y) \cdot y$.

Image as graph Let's think of an image as a graph. Each pixel is a node and has edges to other nodes. The simplest way to connect pixels to other pixels is their direct neighbours, in which case each node has four edges. To avoid losing any information, we will instead consider the case of a complete graph; each node connects to all other nodes.

To preserve the image information in the graph, the graph edges will be assigned a weight, measuring the similarity[1] between the two nodes, thus between two pixels.

There are multiple ways the similarity can be defined. The most intuitive definition considers spatial proximity. This means that similar pixels are spatially close, which, translated to a basic filter, is the same as a Gaussian filter which computes a weighted average of the pixel's neighbourhood and produces what is known as Gaussian blur. Another similarity definition is to consider the pixel's colour. A good compromise is to consider an average of both, spatial and colour closeness, with a certain weighting. A summary of some affinity functions can be found in section 2.2.2.

Once the similarity is defined, we can compute the adjacency matrix of the graph including the edge weights. We will call this matrix the affinity matrix[2] K which represents the similarity of each pixel to every other pixel in the image. Consequently, this matrix is symmetric and of size $N \times N$ with N the number of pixels in the image. Also, most similarity functions define bounds on the values of K such as $0 \leq K_{ij} \leq 1$.

Using this affinity matrix, we obtain the graph Laplacian $\mathcal{L}$, used to build the filter matrix W.

Building the filter Multiple graph Laplacian definitions, more or less equivalent, exist and can have slightly different properties. The table 2.1 proposes a summary of most Laplacian matrix definitions. In the case of image smoothing, the filter W is roughly defined such as $\mathcal{L} = I - W$ [1] and so $W = I - \mathcal{L}$. To get various filters using one Laplacian, we can apply some function f to $\mathcal{L}$ and obtain $W = I - f(\mathcal{L})$ which

[1] Also called affinity.

[2] Or similarity matrix, or kernel matrix

gives us more possibilities on the filter computation. Below the global filter algorithm if we compute the entire matrices:

Algorithm 1 Image processing using entire graph Laplacian operator

Input: y an image of size N, f the function applied to $\mathcal{L}$
Output: z the output image
Compute K (size $N \times N$)
Compute Laplacian matrix $\mathcal{L}$
$z \leftarrow (I - f(\mathcal{L}))y$

However, all these matrices K, $\mathcal{L}$ and W represent huge computational costs. Only storing one of these matrices is already a challenge since they have a size of N^2.

For example, a tiny test image of size 256×256 has 65 536 pixels, so one of these matrices has approximately 4.29×10^9 elements. Considering storing those with a 64 bits type, one matrix takes more than 34 GB of memory. Scaling to a modern smartphone picture, taken with a 10 megapixel camera, a matrix contains 10^{14} elements, meaning 800 TB of memory for each matrix.

Approximation by sampling and Nyström extension To avoid storing any of those huge matrices, approximation will be necessary. Following [16], we define the Nyström extension. It starts by sampling the image and only select a subset of p pixels, with $p \ll N$. Numerically, p should represent around 1% or less of the image pixels. The rows and columns of a matrix K are reorganised such as K_A the upper left affinity matrix of K of size $p \times p$, measuring the affinities between the sampled pixels. The submatrix K_B is holding the similarities between the sampled pixels and the remaining pixels and is of size $p \times (N - p)$. And the lower right submatrix K_C contains the affinities between the remaining pixels. We have:

$$K = \begin{bmatrix} K_A & K_B \\ K_B^T & K_C \end{bmatrix}.$$

Knowing that K_C is of size $(N - p) \times (N - p)$ and that $p \ll N$, this submatrix is still huge and must be avoided.

To have a numerical approximation of a symmetric (semi) positive definite matrix K, we use the eigendecomposition with Φ the orthonormal eigenvectors of K stored as a matrix and Π the eigenvalues

of K:

$$K = \Phi \Pi \Phi^T.$$

The article [16] suggests the Nyström extension to approximate K by $\tilde{K} = \tilde{\Phi}\tilde{\Pi}\tilde{\Phi^T}$, using the eigendecomposition of the submatrix $K_A = \Phi_A \Pi_A \Phi_A^T$, with $\tilde{\Pi} = \Pi_A$ and the approximated leading eigenvectors $\tilde{\Phi}$:

$$\begin{aligned} \tilde{\Phi} &= \begin{bmatrix} \Phi_A \\ K_B^T K_A^{-1} \Phi_A \end{bmatrix} \\ &= \begin{bmatrix} \Phi_A \\ K_B^T \Phi_A \Pi_A^{-1} \end{bmatrix} \end{aligned} \tag{2.1}$$

We can calculate

$$\begin{aligned} \tilde{K} &= \tilde{\Phi}\tilde{\Pi}\tilde{\Phi^T} \\ &= \begin{bmatrix} \Phi_A \\ K_B^T \Phi_A \Pi_A^{-1} \end{bmatrix} \Pi_A \begin{bmatrix} \Phi_A^T & \Pi_A^{-1} \Phi_A^T K_B \end{bmatrix} \\ &= \begin{bmatrix} \Phi_A \Pi_A \\ K_B^T \Phi_A \end{bmatrix} \begin{bmatrix} \Phi_A^T & \Pi_A^{-1} \Phi_A^T K_B \end{bmatrix} \\ &= \begin{bmatrix} K_A & K_B \\ K_B^T & K_B^T K_A^{-1} K_B \end{bmatrix} \end{aligned} \tag{2.2}$$

We can clearly see that the huge submatrix K_C is now approximated by $K_B^T K_A^{-1} K_B$. The quality of the approximation is measurable by the norm of the difference of the two above terms. We recognise the norm of the Schur complement $\|K_C - K_B^T K_A^{-1} K_B\|$.

Therefore, we will only need to compute two submatrices. For our previous examples, if we sample 1% of the pixels, we need to store 0.34 GB of data for each matrix, instead of 34 GB for the 256×256 image. For a 10 megapixel image, each matrix needs 8 TB of memory, which is still a lot of memory. However, as [16] and [2] study, the sampling rate can be lower than 1% and still contain most of the relevant image information.

Eigendecomposition We need to compute the largest eigenvalues of the filter W. The reason can be found by formulating the filter by its diagonalisation $W = \sum_i^N \lambda_i \phi_i \phi_i^T$. We know that the eigenvalues of W are non-negative. When the ith eigenvalue λ_i is small and tends to 0, the eigenvector product will be negligible, and therefore the largest eigenvalues of the filter W are the most relevant.

For the approximation, we will actually need to compute the largest eigenvalues of the sampled pixels submatrix W_A to use the Nyström extension. Using a sample of the image, and thus of the matrix, to compute the largest eigenvalues works to approximate the complete filter because these eigenvalues decay rapidly as shows the figure below:

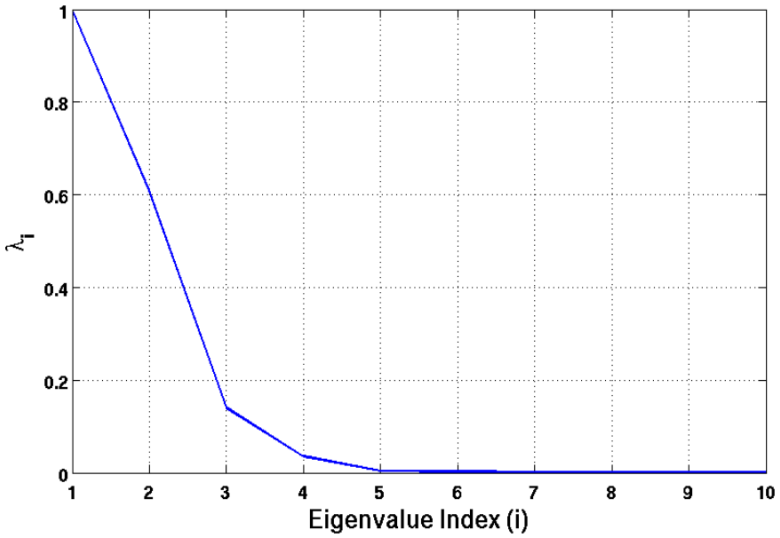

Figure 2.1 – Largest eigenvalues of an image filter, taken from [1].

We know that the eigenvalues of submatrix W_A are between the extremum eigenvalues of the filter W, which is known as the interlacing property of principal submatrices. This can be proven with the Courant-Fischer theorem and the Rayleigh quotient. We also know that computing the largest eigenvalues of the submatrix filter is equivalent to computing the smallest eigenvalues of the corresponding Laplacian operator. This can easily be proven by the filter formula. Let λ be an eigenvalue of W_A and x the associate eigenvector:

$$\begin{aligned} W_A x = \lambda x &\Leftrightarrow (I - \mathcal{L}_A)x = \lambda x \\ &\Leftrightarrow x - \mathcal{L}_A\, x = \lambda x \\ &\Leftrightarrow \mathcal{L}_A\, x = x - \lambda x \\ &\Leftrightarrow \mathcal{L}_A\, x = (1 - \lambda)x \end{aligned} \tag{2.3}$$

The goal of this observation is the way of computing the eigenvalues. For the largest eigenvalues, the most famous algorithm is

the power method. For the smallest eigenvalues, the inverse power method is a usual choice. Both methods converge faster when two successive eigenvalues are far from each other. In broad strokes, the power method requires many cheap iterations, whereas the inverse iteration needs few expensive iterations. In our case, the largest eigenvalues are close to each other; hence the inverse of these eigenvalues will be far from each other. We will therefore prefer the inverse power method.

The algorithm will, in an iterative manner, compute the associated eigenvector of an eigenvalue. This requires either to invert a matrix such as $x_{k+1} = A^{-1}x_k$, or to solve the linear system $Ax_{k+1} = x_k$. We will solve systems of linear equations to compute the first eigenvalues of the Laplacian in order to observe the behaviour of solvers on these dense matrices.

The main drawback of the Nyström method for us, is that it approximates the leading eigenvalues but we compute the trailing ones of the Laplacian $\mathcal{L}$ [17]. It is possible to obtain the eigendecomposition of the filter W, even when W is indefinite, through a method proposed by [16]. It consists of computing the inverse square root matrix $W_A^{-1/2}$, which could be done either by the complete eigendecomposition or by using Cauchy's integral formula. After this step, two more diagonalisation of matrices are required, demanding an important computation time.

Nevertheless, as stated in the objectives of the project, our main goal is not the image processing aspect, but the behaviour of linear solvers of these dense matrices using domain decomposition methods. We will therefore stick to computing the smallest eigenvalues of the Laplacian operator $\mathcal{L}$ and avoid spending too much time on the end of the algorithm implementation.

As the size of the image grows, computing the first p eigenvalues can easily represent computing more than thousand eigenvalues since we consider 1% of all pixels for the sampling step. Instead, we define m, with $m \leq p$, the number of computed eigenvalues of $\mathcal{L}_A$. It is not required to compute all eigenvalues of the sampled pixels matrix, only the first ones are the most relevant, since they correspond to the largest eigenvalues of the filter.

Output image Even if we cannot approximate the trailing eigenvectors of $\mathcal{L}$ through the eigenvectors of $\mathcal{L}_A$, we still define how the Laplacian is used to compute the output image. The summary [18] implicitly defines the filter as $W = I - f(\mathcal{L})$ with the function f that helps achieving various filters. To apply the function efficiently to the Laplacian operator, we apply it to the diagonal eigenvalue matrix such as $f(\mathcal{L}) = \Phi f(\Pi)\Phi^T$. The output image using the filter approximation $\tilde{W}$ can be expressed as:

$$\begin{aligned} \tilde{z} &= \tilde{W}y \\ &= (I - f(\tilde{\mathcal{L}}))y \\ &= (I - \tilde{\Phi} f(\tilde{\Pi})\tilde{\Phi^T})y \\ &= y - \tilde{\Phi} f(\tilde{\Pi})\tilde{\Phi^T}y \end{aligned} \tag{2.4}$$

Below a summary of the complete algorithm using spectral decomposition of the matrix to approximate it:

Algorithm 2 Image processing using approximated graph Laplacian operator

Input: y an image of size N, f the function applied to $\mathcal{L}$
Output: $\tilde{z}$ the output image by the approximated filter
{Sampling}
Sample p pixels, $p \ll N$
{Kernel matrix approximation}
Compute K_A (size $p \times p$) and K_B (size $p \times (N - p)$)
Compute the Laplacian submatrices $\mathcal{L}_A$ and $\mathcal{L}_B$
{Eigendecomposition}
Compute the m smallest eigenvalues Π_A and the associated eigenvectors Φ_A of $\mathcal{L}_A$
{Nyström extension and compute the filter}
See methods of solution proposed by [16]
$\tilde{z} \leftarrow \tilde{W}y$

2.2 Variations

Multiple steps in this algorithm are to be defined in concrete terms to implement them. For each, several methods exist, with different prop-

erties. We present some of these methods for the sampling, computing similarities and the Laplacian operator.

2.2.1 Sampling method

The sample requires to represent only less than 1% of the pixels of the image [16]. To achieve this, we can use different approaches. The chosen method is decisive for the application of the Nyström method. It should capture a snippet of all relevant information in the image.

Random sampling (RS) most common and simple sampling scheme, but no deterministic guarantee of the output quality. It can produce good results for images with poor resolution, but with a huge amount of data, random sampling is limited because it cannot reflect the structure of the data set [19].

K-means sampling (KS) associate to each pixel a 5-D space (R, G, B, X, Y) and divide the pixels into K clusters (K centroids). These clusters are a good sampling scheme for images with simple and uniform backgrounds [20] [21].

Uniform spatially sampling the uniformity of the sample gives good results for image sampling because of the spatial correlation of pixels. This method remains simple but effective [2].

Figure 2.2 – Spatially uniform sampling. Red pixels are sampled. Here 100 pixels are sampled, which only represents 0.04% of all pixels.

Incremental sampling (INS) is an adaptive sampling scheme, meaning that it selects points according to the similarity, so that we can have an approximate optimal rank-k subspace of the original image [19].

Mean-shift segmentation-based sampling this scheme performs good for complex backgrounds. The method consists in over-segmenting the image into n regions and only one pixel of each region will be sampled using the spatially closest pixel to the centre of the region given a formula in [20].

2.2.2 Affinity function

The kernel function measures the similarity between the pixel y_i and y_j. The chosen function is important because it decides on which features the similarity of pixels will be evaluated. Some of the most used affinity functions are:

Spatial Gaussian kernel takes only into account the spatial distance between two pixels [1]. The formula of this kernel is, with $\forall i, j \in$

$[1, N]$, x_i the coordinate vector of a pixel and h_x a normalisation parameter,

$$K(y_i, y_j) = exp(-\frac{||x_i - x_j||^2}{h_x^2}).$$

The greater the parameter is, the more a distant pixel will be considered a close neighbour of the current pixel.

Photometric Gaussian kernel considers the intensity and colour similarity of the pixels [1]. The formula of this kernel is, with z_i the colour or grayscale of a pixel,

$$K(y_i, y_j) = exp(-\frac{||z_i - z_j||^2}{h_z^2}).$$

Generally, the h parameter is a smoothing parameter. If h is small, it is more discriminating between the affinity of different pixels.

Bilateral kernel one of the most used kernel which smooths images by a nonlinear combination of the spatial and photometric Gaussian kernels [1] [2] [22]:

$$K(y_i, y_j) = exp(-\frac{||x_i - x_j||^2}{h_x^2}) \cdot exp(-\frac{||z_i - z_j||^2}{h_z^2}).$$

To generate the example below, we use the famous grayscale image of Barbara of size 512×512 pixels. The more a pixel is coloured in red, the more similar it is to the selected pixel, with respect to the chosen bilateral kernel. A blue coloured pixel is dissimilar to the considered pixel. These are two affinity vectors; the first one is of a pixel on the table leg and the second around Barbara's eye. Keep in mind that each affinity image shown represents only one row of the affinity matrix K.

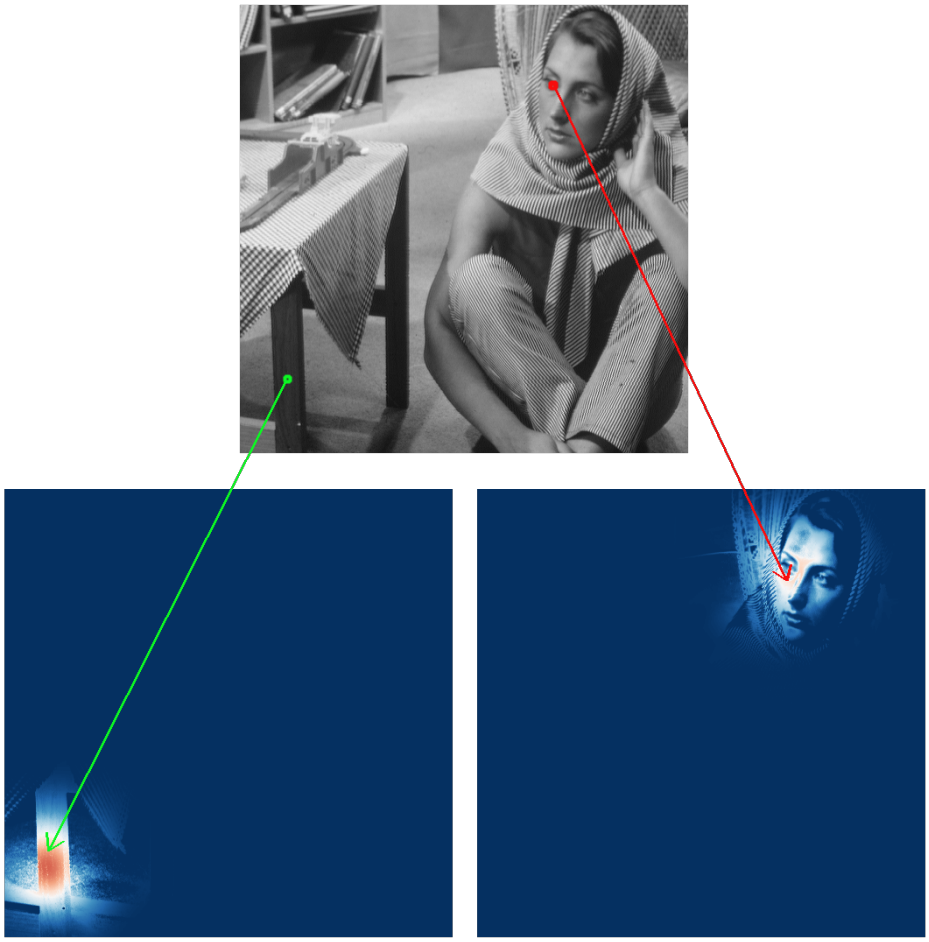

Figure 2.3 – Affinity matrices with $h_x = 50$ and $h_z = 35$.

In a very heterogeneous image, the bilateral kernel will be useful to keep the spatial similarity, while excluding very dissimilar neighbour pixels.

Non-local means (NLM) is similar to the bilateral kernel, a data-dependent filter, except that the photometric affinity is captured patch-wise [2] [23].

Locally adaptive regression kernel (LARK) uses the geodesic distance based on estimated gradients [5] [24].

2.2.3 Graph Laplacian operator

The graph Laplacian operator has multiple possible definitions and each has its own properties. A good summary can be found in [1] and [4]. A graph Laplacian can be symmetric which is important for the eigendecomposition of the matrix. The spectral range, corresponding to the range of the eigenvalues, is important because we can use the filters derived from the Laplacian multiple times, and if the eigenvalues are not between 0 and 1, then the filters tend to be unstable. With K being the affinity matrix, $d_i = \sum_j K_{ij}$, $D = diag\{d_i\}$ and $\bar{d} = \frac{1}{N}\sum_i d_i$:

Laplacian Name	Formula of $\mathcal{L}$	Symmetric	Spectral Range
Un-normalised	$D - K$	Yes	[0, n]
Normalised	$I - D^{-1/2}KD^{-1/2}$	Yes	[0, 2]
Random walk	$I - D^{-1}K$	No	[0, 1]
"Sinkhorn" [5]	$I - C^{-1/2}KC^{-1/2}$	Yes	[0, 1]
Re-normalised [6]	$\alpha(D - K), \alpha \approx \bar{d}^{-}1$	Yes	[0, n]

Table 2.1 – Overview of different graph Laplacian operator definitions.

Generally, it is a good practice to stick to one definition of the Laplacian.

Chapter 3

Implementation

This section presents the work that has been done on the implementation. We start by touching a few words on which variations have been used and what has been done before implementing our algorithm in parallel. After that, we explicit each step of the parallel implementation, what has been implemented by hand and the parts that come from a library. Finally, we present our experiments, the results and discuss them.

3.1 Algorithm details

Variations In our algorithm, we use spatially uniform sampling for the ease of implementation and robustness. The kernel function is the bilateral function with the spatial parameter $h_x = 40$ and the color intensity parameter $h_z = 30$ (see section 2.2.2). We use the re-normalised Laplacian $\mathcal{L} = \alpha(D - K)$ from [6] to avoid expensive computation and use a simple definition.

Prototyping Initially, we implemented the algorithm proposed by [2] in Python, using Numpy, in order to understand the mechanisms and issues of global filtering. Needless to say that this implementation is sequential and limited to small images that require only little computational resources.

3.2 Parallel implementation

To scale our algorithm to use usual camera pictures, but also much larger inputs, we implemented it in a parallel manner using the C language and the Portable, Extensible Toolkit for Scientific Computation (PETSc) [25]. This library is built upon MPI and contains distributed data structures and parallel scientific computation routines. The most useful are the matrix and vector data structures and the parallel matrix-matrix and matrix-vector products. Additionally, PETSc provides Krylov subspace methods and preconditioners for solving linear systems, also implemented in a scalable and parallel manner. In a nutshell, PETSc provides an impressive parallel linear algebra toolkit which is very useful to shorten the development time. As we are basically using MPI, the main parallelism technique that we apply is "single program, multiple data" (SPMD). It is possible to activate some "single instruction, multiple data" (SIMD) parallelism with PETSc but we do not consider it in our case. We want to point out to the reader that the distributed PETSc matrix data structure internally splits the data by default without overlap in a row-wise distribution manner.

In order to verify the correctness of our implementation, we used the Scalable Library for Eigenvalue Problem Computation (SLEPc) [26], which is based on PETSc and provides parallel eigenvalue problem solvers. Furthermore, we need the library Elemental [27] in order to do dense matrix operations in PETSc.

We present how we included parallelism in our algorithm step-by-step, starting with reading the image and sampling. Then follows the computation of the affinities of the sampled pixels. And we finish with the computation of the smallest eigenvalues using the inverse subspace iteration. The implementation associated to this project is open source and can be found on GitHub[1].

Initialisation and sampling During the initialisation phase, the input image is read into memory sequentially by process 0. Since we consider that the input image fits into memory, we broadcast the entire image from process 0 to all other processes. Every process will hold the entire input image which will be useful since every process needs every pixel to compute the affinities.

[1] `https://github.com/David-Wobrock/image-processing-graph-laplacian/`

The sampling step is also done by every process independently. All processes know the indices of the sampled pixels. This is possible because we use spatially uniform sampling, which is deterministic, fast to compute and doesn't require communication.

Submatrices computations The computation of the affinity submatrices K_A and K_B is done locally by each process using a formula from section 2.2.2. Indeed, each process computes the rows of the matrix that it will hold locally. In other words, each process computes the affinities between a subset of the sampled pixels and all pixels. Since every process holds the complete image, no communication is needed. The overhead is thus minimal and this part of the algorithm scales very well with respect to the number of processes.

Then, we compute the Laplacian submatrices $\mathcal{L}_A$ and $\mathcal{L}_B$ using a formula from table 2.1. The submatrix $\mathcal{L}_A$ requires to first compute the part D_A of the diagonal matrix D of normalisation coefficients. Again, each process can locally sum each row of $K_A + K_B$ because they have the same distribution layout, so no communication is needed. However, to compute the normalisation factor α in our Laplacian definition $\alpha(D - K)$ with $\alpha = \bar{d}^{-1}$ and $\bar{d} = \sum_{i=1}^{N} \frac{d_i}{N}$, we need communication to find the average of the normalisation coefficients. Nevertheless, the implied communication costs are not critical since we broadcast only one value for each processor.

Inverse subspace iteration The used algorithm to compute the smallest eigenvalues is the inverse subspace iteration inspired by [28]. With m the number of eigenvalues we will compute, p the sample size and $m \leq p$, we start the algorithm by selecting m random orthonormal independent vectors X_0 of size p.

The inverse iteration algorithm consists of outer and inner iterations, with k the index of the current outer iteration. The inner iteration consists of solving m linear systems in X_{k+1}, one for each vector of X_k that we approximate, such that $\forall i \in [1, m]$ and $X_k^{(i)}$ the ith vector of the subspace X_k:

$$AX_{k+1}^{(i)} = X_k^{(i)}.$$

The outer iteration consists of repeating this process and orthonormalising the new vectors X_{k+1} until convergence, meaning having a small enough residual norm. We define the residual R_k of X_k, at a certain

iteration k, as

$$\begin{aligned} R_k &= AX_k - X_k X_k^T AX_k \\ &= (I - X_k X_k^T) AX_k. \end{aligned} \tag{3.1}$$

We implemented a parallel Gram-Schmidt routine for orthogonalisation, based on the classical sequential one. A summary of the inverse subspace iteration algorithm:

Algorithm 3 Inverse subspace iteration

Input: A the matrix of size $p \times p$, m the number of required eigenvalues, ε a tolerance
Output: X_k the desired invariant subspace
Initialise m random orthonormal vectors X_0 of size p
$R_0 \leftarrow (I - X_0 X_0^T) AX_0$
For k=0, 1, 2, ...
while $\|R_k\| > \varepsilon$ **do**
  for i=1 **to** m **do**
    Solve $AX_{k+1}^{(i)} = X_k^{(i)}$
  end for
  $X_{k+1} \leftarrow$ Orthonormalise(X_{k+1})
  $R_{k+1} \leftarrow (I - X_{k+1} X_{k+1}^T) AX_{k+1}$
end while

Solving the systems of linear equations is done using the Krylov type solvers and the preconditioners included in PETSc. As a standard approach, we use GMRES as our solver and the RAS method as preconditioner, without overlap and 2 domains per process. Each subdomain is solved using the GMRES method also.

On each outer iteration, we must compute the residuals to see if we converged. This requires multiple matrix-matrix products and computing a norm, so communication cannot be avoided here.

Nyström extension and output image As stated previously, at the end of section 2.1, the Nyström extension finds the leading eigenvectors, whereas we would need the trailing ones, as explain the articles [17], [16] and [2]. So the algorithm will not compute the output image for now.

3.3 Results

Experimental setup The experiments are done on the test cluster of the Laboratory Jacques-Louis Lions at Sorbonne University (formerly University Pierre and Marie Curie). This computer has 32 CPUs of 10 cores each, clocked at 2.4 GHz, and a total memory of 2 TB. The setup of the experiments consists of running a specific test with different parameters, scaling the algorithm up to 192 processors. The code is compiled on this computer using GCC 6.3.0, without compiler flags, and the MPI implementation is Open MPI 1.8.3. The versions of other libraries are PETSc 3.8.3, SLEPc 3.8.2 and Elemental 0.87.7.

We start by executing the algorithm without approximation. This way, we will be able to see the results of the algorithm, even if the size of the input images will be limited. After that, we study the approximation and computation of the smallest eigenvalues of the Laplacian.

3.3.1 Entire matrix computation

Results We start by showing the result of the computation using the full matrices in order to see the image processing result. We limit ourselves to grayscale images for the beginning. As stated before, computing the entire matrices can only be done on small images. The image below contains 135 000 pixels, so each matrix needs around 145 GB.

Figure 3.1 – Left: input image. Right: sharpened image.

The cat's fur on the left-hand side, the person's hand and the cushion on the right-hand side appear to be more detailed. We observe that the already sharp part of the image, such as the cat's head, stays

nice and is not over-sharpened. We obtained this filter by defining $f(\mathcal{L}) = -3\,\mathcal{L}$ in the output image $z = (I - f(\mathcal{L}))y$.

This corresponds to the adaptive sharpening operator defined in [1] as $(I + \beta\,\mathcal{L})$ with $\beta > 0$. This approach remains a simple application of a scalar and doesn't require any eigenvalue computation. A more complete approach is called multiscale decomposition [3] and consists of applying a polynomial function to the Laplacian $\mathcal{L}$. It applies various coefficients to different eigenvalues of $\mathcal{L}$ because each eigenpair captures particular features of the image.

Performances and discussions We run the algorithm multiple times for each number of processors on the image shown above, up to 192 processors, and average the runs. Below the total runtime for each number of processors:

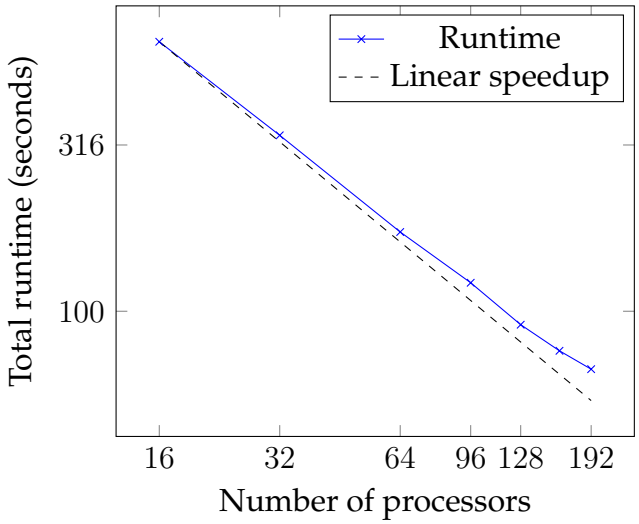

Figure 3.2 – Total runtime of the algorithm with entire matrix computation (log scale).

We observe that the runtime decreases significantly with respect to the number of processors. We can also see that, by doubling the number of processors, we nearly accelerate the runtime by a factor 2. It is an excellent result since we achieve strong scalability for the entire matrix computation case. However, some overhead will always be present and the matrix-vector operations necessarily require communication, limiting scalability. To observe if some parts scale better than others, we compare the proportion of runtime spent in each part:

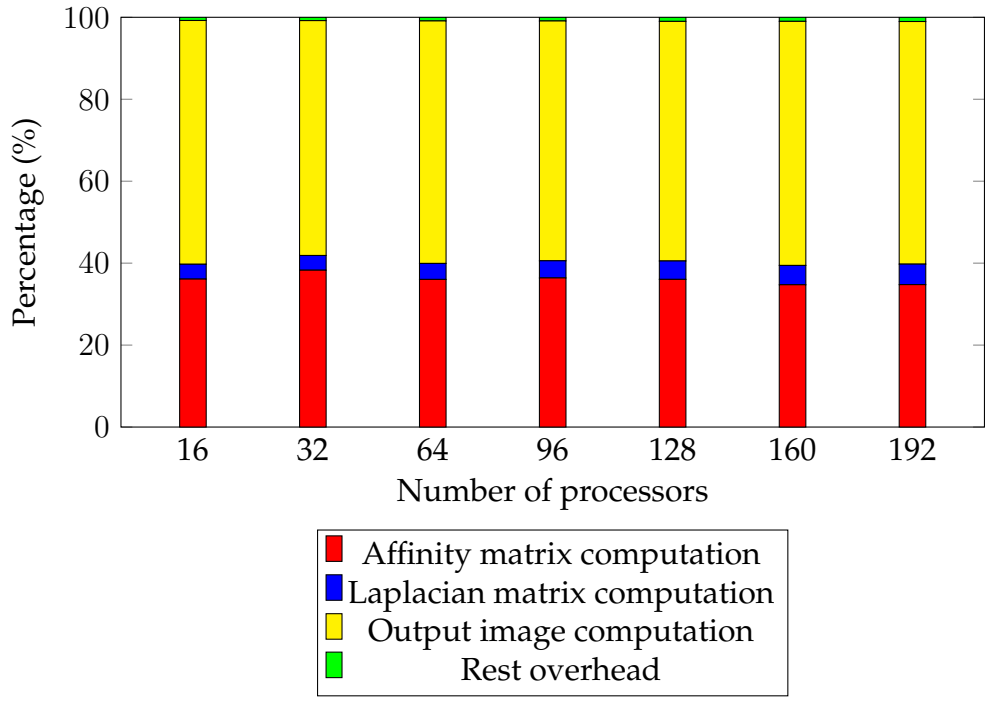

Figure 3.3 – Proportion of runtime spent in each step in the total execution of the algorithm with entire matrix computation.

We see that the proportion of each part remains the same over the increase of processors, meaning that the three main parts scale equivalently. However, when allocating an excessive amount of processors to this task compared to the input size, we may observe an increase of the runtime because we spend most time on communication overhead.

Overall, computing the entire matrices scales well because we only have matrix-matrix and matrix-vector products. With an appropriate cluster, those scale nicely. We now consider larger inputs, which require approximation and introduces linear algebra components which might slow down the algorithm.

3.3.2 Approximation computation

Eigenvalues We remind that the end of the algorithm, using matrix approximation to compute the filtered image, is not implemented. Nonetheless, we will present interesting results about the computation of the eigenvalues of the graph Laplacian operator. We consider

a picture with 402 318 pixels and we sample 1% of them, which corresponds approximately to 4000 sample pixels. Additionally to varying the number of processors, we also vary the number of computed eigenvalues from 50 up to 500. Here are the first 500 eigenvalues of the Laplacian submatrix:

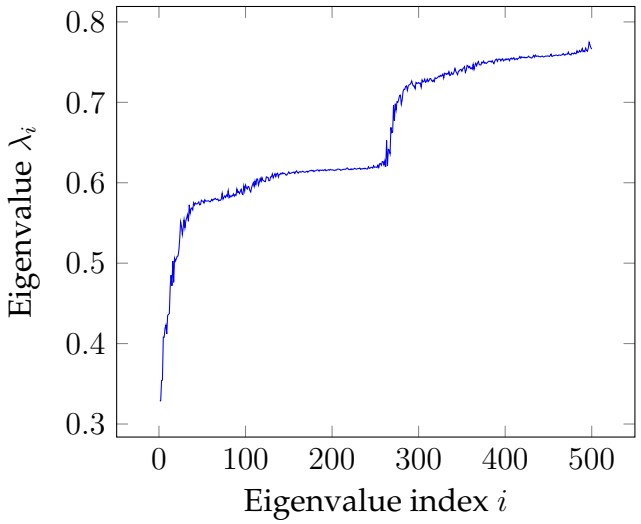

Figure 3.4 – First 500 eigenvalues of the Laplacian submatrix.

To compute these eigenvalues, we used the inverse subspace iteration [28]. We now look at the algorithm runtime performances.

Performances As a reminder, we used GMRES to solve the linear systems with RAS preconditioning and using GMRES on the subdomains. We sample 1% of the pixels of an image with $4 \cdot 10^5$ pixels using spatially uniform sampling. The performances of the inverse subspace iteration for 50 and 500 eigenvalues:

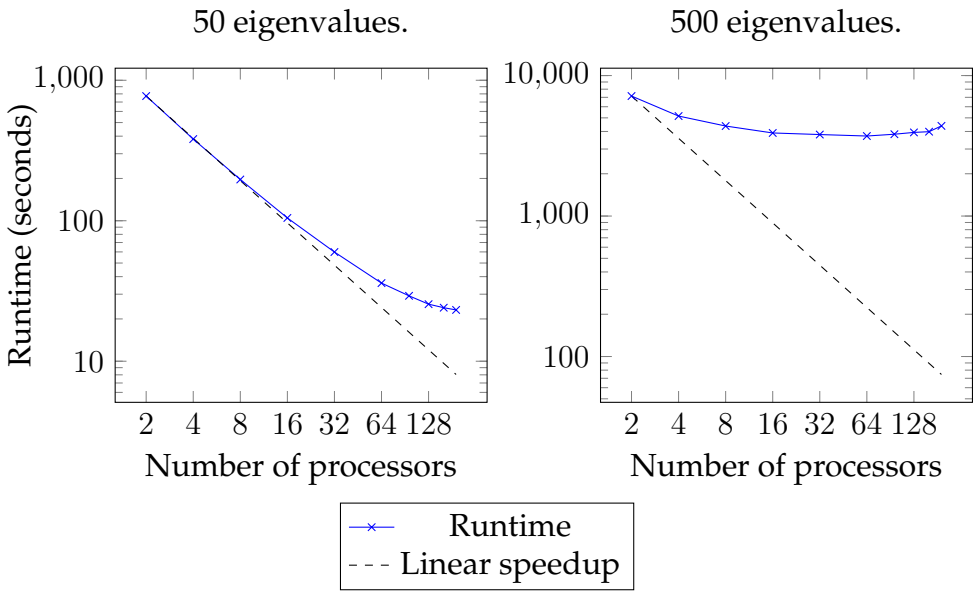

Figure 3.5 – Runtime of the inverse subspace iteration part of the algorithm (log scale).

When increasing a low number of processors, we see an improvement of the performances in both cases. But the runtime stagnates slowly for 50 eigenvalues and quickly for 500 eigenvalues. We even observe a raise of the runtime for 500 eigenvalues. The algorithm reaches its parallelisation limit and the communication overhead takes over. For 500 eigenvalues, the runtime for 2 processors is over 7000 seconds, and the fastest runtime is reached for 64 processors and is of 3700 seconds.

We know that the inverse iteration part of the algorithm is not scaling correctly compared to the other parts. For any amount of computed eigenvalues, when we increase the number of processes, the proportion of time spent computing the eigenvalues increases. For 500 eigenvalues and 128 processors, we spend more than 99% of the time computing the eigenvalues. This confirms that the algorithm does not quite scale yet.

We look at the internal steps of the inverse power method to see where lies the problem. The algorithm consists of iteratively solving m linear systems, orthonormalising the vectors and computing the residual norm. Here is the proportion of each step of the inverse subspace interation for the computation of 50 and 500 eigenvalues:

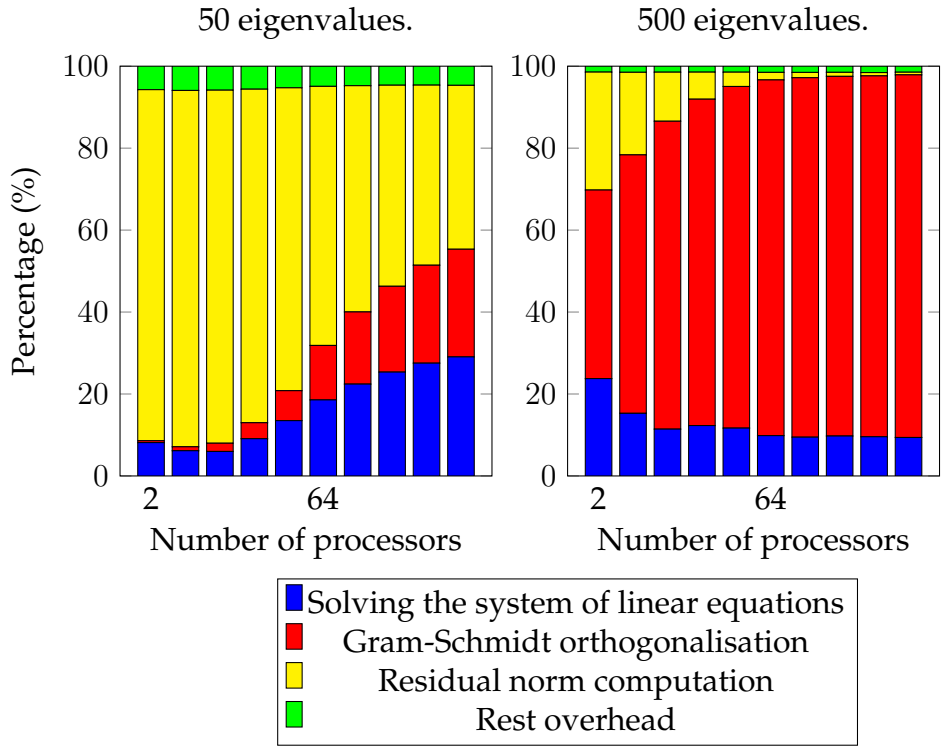

Figure 3.6 – Proportion of each step in the inverse subspace iteration.

We observe that the Gram-Schmidt orthogonalisation is the limiting factor and is the most time-consuming step of the inverse iteration as the number of processors grows. It is a well-known problem that the simple Gram-Schmidt process is actually difficult to parallelise efficiently. Small optimisations for a parallel Gram-Schmidt orthogonalisation exist [29] but they do not properly solve the problem. This issue will be difficult to overcome completely.

Skipping some orthogonalisations Fundamentally, the orthogonalisation is used to stabilise the algorithm. To accelerate our algorithm further, we try to orthogonalise the vectors X_k every other iteration instead of every iteration. We present below the resulting performances:

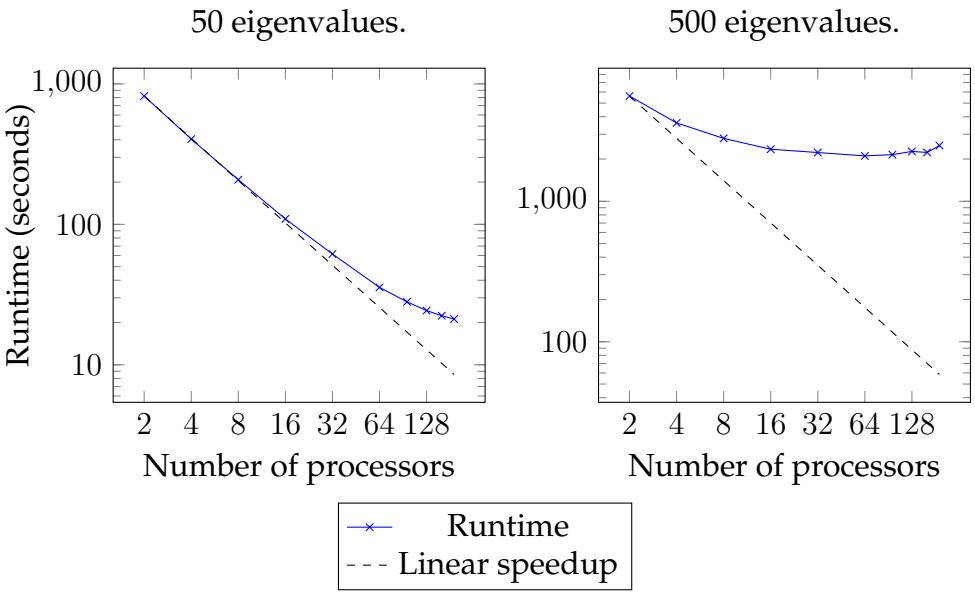

Figure 3.7 – Runtime of the inverse subspace iteration with skipping the Gram-Schmidt procedure every other iteration (log scale).

The performances for 50 eigenvalues are similar to the case when we are not skipping the Gram-Schmidt orthogonalisation every other iteration. We saw that only a small proportion of time is spent doing the orthogonalisation in this case, so the impact is not significant.

However, for computing 500 eigenvalues, the runtime with skipping the orthogonalisation every other iteration is much lower. Most time is spent doing the Gram-Schmidt process, so the execution is considerably sped up. For 2 processors, the runtime is around 5600 seconds and the fastest runtime is 2100 seconds for 64 processors. Even if the algorithm requires a few more outer iterations to converge, we nearly observe a factor 2 speed up with respect to the algorithm without skipping the Gram-Schmidt procedure. The communication overhead of the method remains a problem when the number of processors is large.

When skipping the Gram-Schmidt more often, we might see further improvements.

Below the runtime for 2 and 64 processors of the inverse subspace algorithm depending on the frequency of orthogonalisation:

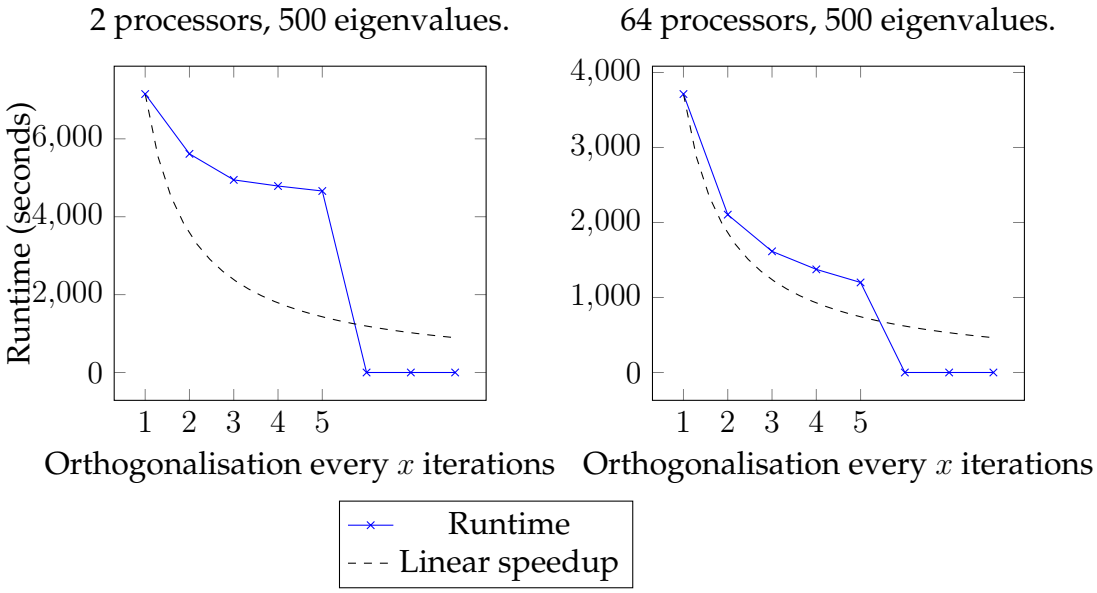

Figure 3.8 – Runtime of the inverse subspace iteration depending on the amount of Gram-Schmidt procedures.

For 2 processors, we see that the speedup stagnates quickly when skipping the Gram-Schmidt orthogonalisation more and more often. Indeed, the orthogonalisation represents a decent proportion of the execution time for 2 processors whereas it represents over 85% for 64 processors. Therefore we observe a speedup of the inverse subspace iteration that is nearly linear for 64 processors. The number of outer iterations between not skipping the Gram-Schmidt algorithm and applying it every 5 iterations only varies from 38 to 40 which explains the resulting runtime.

SLEPc comparison We compare the performances of our algorithm with the parallel eigenvalue problem solver SLEPc. We compute the same amount of eigenvalues with the same number of processors using the provided Krylov-Schur algorithm [30]. The performances of SLEPc in comparison of our method:

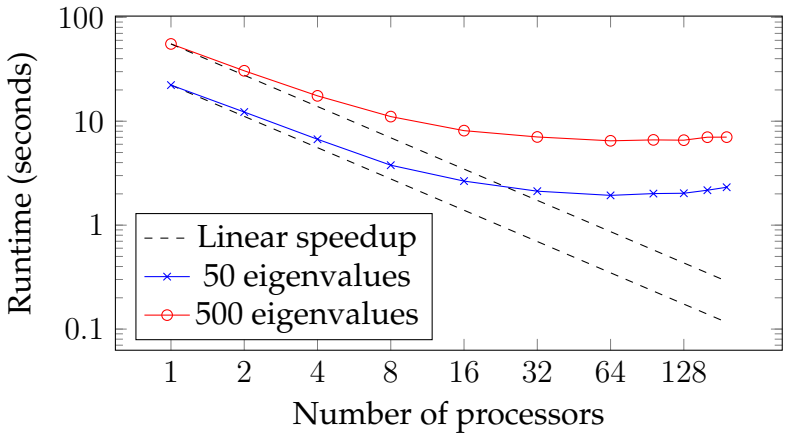

Figure 3.9 – Runtime of the Krylov-Schur algorithm in SLEPc (log scale).

First of all, it is notable that both execution times for 50 and 500 eigenvalues are close to each other. This comes from the way the internal Krylov-Schur algorithm functions. Also, we observe that both plots have the same profile and evolve in the same way with respect to the number of processors.

The algorithm, on this test case, is faster than our implementation. However, both approaches tend to have a light increase in the runtime after reaching a certain number of processors.

We should note that SLEPc is a library that has been developed over many years by experts and is specially designed for solving eigenvalue problems in parallel. It is the current state-of-the-art implementation in this field. Additionally, we do not use the same approach to compute the eigenvalues, so they are difficult to compare directly.

Linear solver performances

We want to explore the behavior of the linear solver using domain decomposition methods for our inverse iteration algorithm. We remind that we used GMRES and the RAS domain decomposition method with 2 domains per process for the previous examples. We compare the runtimes of solving 50 linear systems with different number of processors, with and without preconditioner:

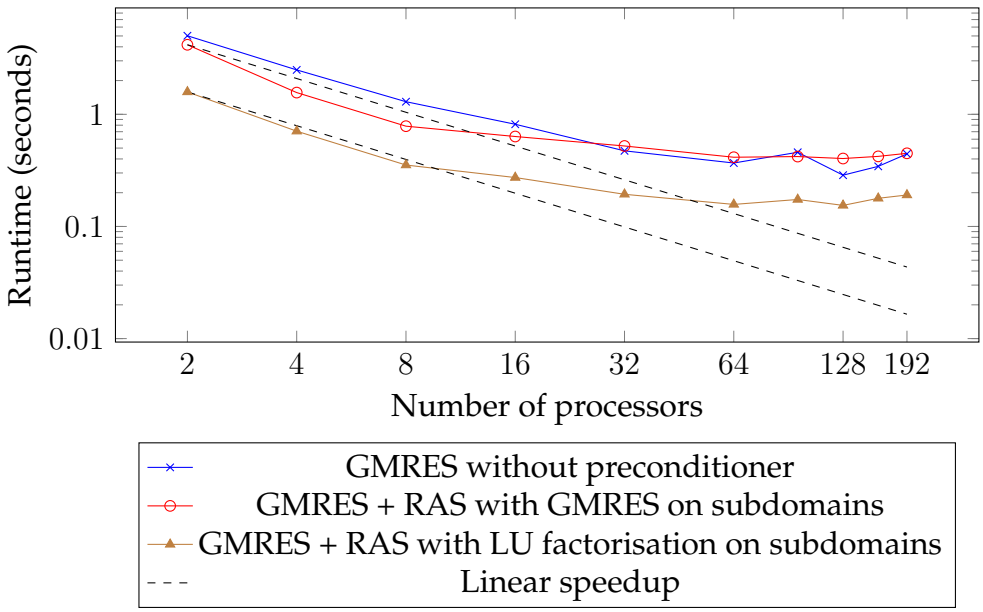

Figure 3.10 – Runtime of the linear solver for 50 eigenvalues for a 4000×4000 matrix (log scale).

Overall, we observe that using a domain decomposition method as preconditioner for our dense systems shows slightly better runtime performances. Between not using a preconditioner and applying domain decomposition with GMRES on the subdomains, the difference is visible for a small number of processors. When increasing the number of processors, both tend to have the same performances, so preconditioning takes the same time as solving the system with an ill-conditioned matrix.

Applying the LU factorisation on the subdomains exposes a much better runtime. Indeed, the input matrix is rather small in our case, and by splitting the problems into subdomains, the matrices are even smaller and suitable for a direct method.

For all methods, the speedup is nearly linear until 8 processors and we see a stagnation of the runtime after.

To explore further the behaviour of the solver, we now run the algorithm on a larger image, containing 1.44 million pixels and hence sampling 14400 pixels. We use the same setups as previously and show the runtimes for solving 50 linear systems:

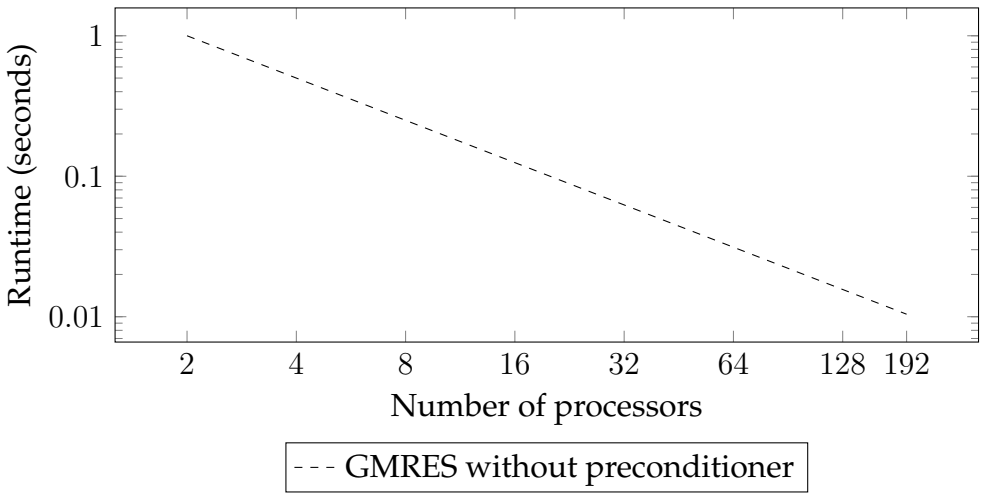

Figure 3.11 – Runtime of the linear solver for 50 eigenvalues for a 14400×14400 matrix (log scale).

As we can see...

Chapter 4

Conclusion

4.1 Discussions

Linear solvers & domain decomposition methods on dense matrices As we seen through the experiments, the resolution of linear systems scales slightly with respect to the number of processors. Domain decomposition methods improve the performances of solving dense systems of linear equations, even without overlap. In our small case, a direct solver showed the best results for preconditioning. This might not be the case for larger systems.

Gram-Schmidt process We saw that the orthogonalisation process is difficult to parallelise efficiently. Skipping the Gram-Schmidt procedure every other iteration, to stabilise the algorithm less often, gave an improvement, but we cannot totally avoid the cost of it when increasing the number of processors. This problem is well-known and one of the biggest limitations for scaling diverse algorithms to a large number of processors.

The Gram-Schmidt procedure orthogonalises a set of vectors by sequentially substracting from a vector the projections on the previously orthogonalised vectors. The inner product of two vectors is computed frequently, because of the projection, and since each vector is shared over all processors, a lot of communication is involved in this operation. Attempts for parallel implementation are numerous, like [29], but they either still have many communications or they suggest a different memory distribution schema.

4.2 Perspectives

Image processing An improvement of our algorithm would be to finish the part computing the output image. This requires to compute the inverse of the square root of the matrix. This can be done either by computing the entire eigendecomposition to get the inverse square root matrix. Another way to accomplish this, could be to consider the Cauchy integral formula such as:

$$A^{-\frac{1}{2}} = \frac{1}{2\pi i} \oint_C z^{-\frac{1}{2}} (zI - A)^{-1} \mathrm{d}z.$$

However, the time for this project was not enough to explore this possibility completely.

An easy improvement would be to also consider color images. This can be done by decomposing the RGB image to a YCC image, with one grayscale and two chroma components. The algorithm is applied to all three components, and then they are converted back to RGB.

A way to improve the filtering is multiscale decomposition. As explained in [3], instead of applying a linear function to all eigenvalues such as $f(W) = \phi f(\Pi)\phi^T$, we can actually use a polynomial function f. This is interesting because each eigenpair captured various features of the image and one can apply different coefficients on different aspects of the image.

For the state-of-the-art, the article [31] proposes an enhancement of global filtering. It argues that the eigendecomposition remains computationally too expensive and shows results of an improvement. The presented results and performances are astonishing; however, the method is hardly described and replicating it would be difficult. This is understandable since this algorithm seems to be in the latest Pixel 2 smartphone by Google and they want to preserve their market advantage in the field of image processing.

A real improvement would be to formulate a method for extending the trailing eigenvectors of the sampled Laplacian $\mathcal{L}_A$. This way, it would be possible to apply the spectral decomposition of the Laplacian, and thus apply a filter to the input image.

Linear solver A way to highly parallelise the matrix computations could be using graphical processing units (GPUs). Especially the matrix-matrix and matrix-vector products could be nicely improved with GPUs.

However, solving systems of linear equations is a task that GPUs are not designed for.

It would be interesting to explore more the impact of the number of sampled pixels, which corresponds to the input matrix of the linear system. The articles [16] and [2] started a study on the size of the samples, but only for small images. This work could be extended.

Bibliography

[1] Peyman Milanfar. *Non-conformist Image Processing with Graph Laplacian Operator*. 2016. URL: https://www.pathlms.com/siam/courses/2426/sections/3234.

[2] Hossein Talebi and Peyman Milanfar. "Global Image Denoising". In: *IEEE Transactions on Image Processing* 23.2 (Feb. 2014), pp. 755–768. ISSN: 1057-7149, 1941-0042. DOI: 10.1109/TIP.2013.2293425. URL: http://ieeexplore.ieee.org/document/6678291/.

[3] H. Talebi and P. Milanfar. "Nonlocal Image Editing". In: *IEEE Transactions on Image Processing* 23.10 (2014), pp. 4460–4473. ISSN: 1057-7149. DOI: 10.1109/TIP.2014.2348870.

[4] Fan R. K. Chung. *Spectral graph theory*. Vol. 92. CBMS Regional Conference Series in Mathematics. Published for the Conference Board of the Mathematical Sciences in Washington, DC; by the American Mathematical Society in Providence, RI, 1997, pp. xii+207. ISBN: 0-8218-0315-8.

[5] Peyman Milanfar. "Symmetrizing Smoothing Filters". en. In: *SIAM Journal on Imaging Sciences* 6.1 (Jan. 2013), pp. 263–284. ISSN: 1936-4954. DOI: 10.1137/120875843. URL: http://epubs.siam.org/doi/10.1137/120875843.

[6] Peyman Milanfar and Hossein Talebi. "A new class of image filters without normalization". In: *Image Processing (ICIP), 2016 IEEE International Conference on*. IEEE, 2016, pp. 3294–3298.

[7] Miroslav Fiedler. "Algebraic connectivity of graphs". eng. In: *Czechoslovak Mathematical Journal* 23.2 (1973), pp. 298–305. ISSN: 0011-4642. URL: https://eudml.org/doc/12723.

[8] Hao Zhang, Oliver Van Kaick, and Ramsay Dyer. "Spectral mesh processing". In: *Computer graphics forum*. Vol. 29. Wiley Online Library, 2010, pp. 1865–1894.

[9] Antoni Buades, Bartomeu Coll, and Jean-Michel Morel. "A Review of Image Denoising Algorithms, with a New One". In: *SIAM Journal on Multiscale Modeling and Simulation* 4 (Jan. 2005). DOI: 10.1137/040616024.

[10] K. Dabov et al. "Image Denoising by Sparse 3-D Transform-Domain Collaborative Filtering". In: *IEEE Transactions on Image Processing* 16.8 (Aug. 2007), pp. 2080–2095. ISSN: 1057-7149. DOI: 10.1109/TIP.2007.901238.

[11] MathWorks. *MATLAB - Solve systems of linear equations*. https://fr.mathworks.com/help/matlab/ref/mldivide.html.

[12] P. R. Amestoy et al. "A Fully Asynchronous Multifrontal Solver Using Distributed Dynamic Scheduling". In: *SIAM Journal on Matrix Analysis and Applications* 23.1 (2001), pp. 15–41.

[13] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Other Titles in Applied Mathematics. DOI: 10.1137/1.9780898718003. Society for Industrial and Applied Mathematics, Jan. 2003. ISBN: 978-0-89871-534-7. URL: http://epubs.siam.org/doi/book/10.1137/1.9780898718003.

[14] Y. Saad and M. Schultz. "GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems". In: *SIAM Journal on Scientific and Statistical Computing* 7.3 (July 1986), pp. 856–869. ISSN: 0196-5204. DOI: 10.1137/0907058. URL: http://epubs.siam.org/doi/abs/10.1137/0907058.

[15] Victorita Dolean, Pierre Jolivet, and Frédéric Nataf. *An introduction to domain decomposition methods*. Algorithms, theory, and parallel implementation. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2015. ISBN: 978-1-611974-05-8. URL: http://dx.doi.org/10.1137/1.9781611974065.ch1.

[16] Charless Fowlkes et al. "Spectral grouping using the Nystrom method". In: *IEEE transactions on pattern analysis and machine intelligence* 26.2 (2004), pp. 214–225. URL: http://ieeexplore.ieee.org/abstract/document/1262185/.

[17] Serge Belongie et al. “Spectral partitioning with indefinite kernels using the Nyström extension”. In: *European conference on computer vision*. Springer, 2002, pp. 531–542.

[18] Peyman Milanfar. “A Tour of Modern Image Filtering: New Insights and Methods, Both Practical and Theoretical”. In: *IEEE Signal Processing Magazine* 30.1 (Jan. 2013), pp. 106–128. ISSN: 1053-5888. DOI: 10.1109/MSP.2011.2179329. URL: http://ieeexplore.ieee.org/document/6375938/.

[19] Qiang Zhan and Yu Mao. “Improved spectral clustering based on Nyström method”. en. In: *Multimedia Tools and Applications* 76.19 (Oct. 2017), pp. 20149–20165. ISSN: 1380-7501, 1573-7721. DOI: 10.1007/s11042-017-4566-4. URL: http://link.springer.com/10.1007/s11042-017-4566-4.

[20] Chieh-Chi Kao et al. “Sampling Technique Analysis of Nyström Approximation in Pixel-Wise Affinity Matrix”. In: July 2012, pp. 1009–1014. ISBN: 978-1-4673-1659-0. DOI: 10.1109/ICME.2012.51.

[21] Kai Zhang, Ivor W. Tsang, and James T. Kwok. “Improved Nyström low-rank approximation and error analysis”. In: *Proceedings of the 25th international conference on Machine learning*. ACM, 2008, pp. 1232–1239. URL: http://dl.acm.org/citation.cfm?id=1390311.

[22] C. Tomasi and R. Manduchi. “Bilateral filtering for gray and color images”. In: *Sixth International Conference on Computer Vision (IEEE Cat. No.98CH36271)*. Jan. 1998, pp. 839–846. DOI: 10.1109/ICCV.1998.710815.

[23] C. Kervrann and J. Boulanger. “Optimal Spatial Adaptation for Patch-Based Image Denoising”. In: *IEEE Transactions on Image Processing* 15.10 (Oct. 2006), pp. 2866–2878. ISSN: 1057-7149. DOI: 10.1109/TIP.2006.877529.

[24] H. Takeda, S. Farsiu, and P. Milanfar. “Kernel Regression for Image Processing and Reconstruction”. In: *IEEE Transactions on Image Processing* 16.2 (Feb. 2007), pp. 349–366. ISSN: 1057-7149. DOI: 10.1109/TIP.2006.888330.

[25] Satish Balay et al. *PETSc Web page*. http://www.mcs.anl.gov/petsc. 2017. URL: http://www.mcs.anl.gov/petsc.

[26] Vicente Hernandez, Jose E. Roman, and Vicente Vidal. "SLEPc: A scalable and flexible toolkit for the solution of eigenvalue problems". In: *ACM Trans. Math. Software* 31.3 (2005), pp. 351–362.

[27] Jack Poulson et al. "Elemental: A New Framework for Distributed Memory Dense Matrix Computations". In: *ACM Trans. Math. Softw.* 39.2 (Feb. 2013), 13:1–13:24. ISSN: 0098-3500. DOI: 10.1145/2427023.2427030. URL: http://doi.acm.org/10.1145/2427023.2427030.

[28] G. El Khoury, Yu. M. Nechepurenko, and M. Sadkane. "Acceleration of inverse subspace iteration with Newton's method". In: *Journal of Computational and Applied Mathematics*. Proceedings of the Sixteenth International Congress on Computational and Applied Mathematics (ICCAM-2012), Ghent, Belgium, 9-13 July, 2012 259 (Mar. 2014), pp. 205–215. ISSN: 0377-0427. DOI: 10.1016/j.cam.2013.06.046. URL: http://www.sciencedirect.com/science/article/pii/S0377042713003440.

[29] Takahiro Katagiri. "Performance Evaluation of Parallel Gram-Schmidt Re-orthogonalization Methods". In: *High Performance Computing for Computational Science — VECPAR 2002*. Ed. by José M. L. M. Palma et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 302–314. ISBN: 978-3-540-36569-3.

[30] Gilbert W. Stewart. "A Krylov–Schur algorithm for large eigenproblems". In: *SIAM Journal on Matrix Analysis and Applications* 23.3 (2002), pp. 601–614.

[31] Hossein Talebi and Peyman Milanfar. "Fast Multilayer Laplacian Enhancement". In: *IEEE Transactions on Computational Imaging* 2.4 (Dec. 2016), pp. 496–509. ISSN: 2333-9403, 2334-0118. DOI: 10.1109/TCI.2016.2607142. URL: http://ieeexplore.ieee.org/document/7563313/.