PARALLEL IMAGE RESTORATION

A THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING
AND THE INSTITUTE OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
Tahir Malas
January, 2004

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Prof. Dr. Cevdet Aykanat (Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. Mustafa Pınar

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assist. Prof. Dr. Uğur Güdükbay

Approved for the Institute of Engineering and Science:

Prof. Dr. Mehmet B. Baray
Director of the Institute

ABSTRACT

PARALLEL IMAGE RESTORATION

Tahir Malas
M.S. in Computer Engineering
Supervisor: Prof. Dr. Cevdet Aykanat
January, 2004

In this thesis, we are concerned with the image restoration problem which has been formulated in the literature as a system of linear inequalities. With this formulation, the resulting constraint matrix is an unstructured sparse-matrix and even with small size images we end up with huge matrices. So, to solve the restoration problem, we have used the surrogate constraint methods, that can work efficiently for large size problems and are amenable for parallel implementations. Among the surrogate constraint methods, the basic method considers all of the violated constraints in the system and performs a single block projection in each step. On the other hand, parallel method considers a subset of the constraints, and makes simultaneous block projections. Using several partitioning strategies and adopting different communication models we have realized several parallel implementations of the two methods. We have used the hypergraph partitioning based decomposition methods in order to minimize the communication costs while ensuring load balance among the processors. The implementations are evaluated based on the per iteration performance and on the overall performance. Besides, the effects of different partitioning strategies on the speed of convergence are investigated. The experimental results reveal that the proposed parallelization schemes have practical usage in the restoration problem and in many other real-world applications which can be modeled as a system of linear inequalities.

Keywords: Parallel image restoration, distortion, parallel algorithms, linear feasibility, surrogate constraint method, hypergraph partitioning, rowwise partitioning, checkerboard partitioning, fine-grain partitioning, point-to-point communication, all-to-all communication, convergence rate.

ÖZET

PARALEL GÖRÜNTÜ ONARIMI

Tahir Malas
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Prof. Dr. Cevdet Aykanat
Ocak, 2004

Bu çalışmada, doğrusal eşitsizlikler sistemine dönüştürülmüş olan görüntü onarımı problemi üzerinde durulmuştur. Bu yöntemle elde edilen matrisler belli bir yapısal dizilime sahip olmayan seyrek matrislerdir. Ayrıca, küçük ölçekli görüntülerde dahi çok büyük ölçekli matrisler oluşmaktadır. Dolayısıyla, problemin çözümünde, büyük ölçekli problemler için verimli çalışabilen ve paralel gerçekleştirmelere uygun olan aracı kısıtlar yöntemleri kullanılmıştır. Önerilen yöntemler arasından, sağlanmayan kısıtların tümünü dikkate alan ve her adımda tek bir izdüşüm gerçekleştiren temel yöntem ve sağlanmayan kısıtların alt kümelerini dikkate alıp oluşan izdüşümlerin dışbükey birleşimini alan paralel yöntem kullanılmıştır. Çeşitli bölümleme stratejileri ve farklı iletişim modelleri kullanılarak bir çok paralel gerçekleştirimler yapılmıştır. Hiper-çizge temelli bölümlemeler kullanılarak iletişim maliyeti azaltılırken işlemciler arasındaki yük dengesi sağlanmıştır. Gerçekleştirimler yineleme bazında ve toplam bazda değerlendirilmiştir. Aynı zamanda, bölümlemelerin yakınsama hızına olan etkisi araştırılmıştır. Deney sonuçları, önerilen paralel yöntemlerin, görüntü onarımı probleminde ve doğrusal eşitsizlikler sistemine çevrilebilen gerçek uygulamalarda pratik kullanımı olduğunu göstermiştir.

Anahtar sözcükler: Paralel görüntü onarımı, bozunum, paralel algoritmalar, lineer fizibilite, aracı kısıtlar yöntemi, hiper-çizge parçalama, sırasal parçalama, damatahtası parçalama, ince tane parçalama, noktasal iletişim, herkes-herkese iletişim, yakınsama hızı.

Acknowledgement

I would like to express my gratitude to Prof. Dr. Cevdet Aykanat for his supervision, guidance, and suggestions throughout the development of this thesis.

I would like to thank the committee members Assist. Prof. Dr. Uğur Güdükbay and Assoc. Prof. Dr. Mustafa Pınar for reading and commenting on this thesis.

I would like to give my special thanks to Bora Uçar. He helped me a lot and gave invaluable support during the last year. I also thank to my office mates Emre Şahin and Gökhan Yavaş for their friendship.

Especially, I would like to thank my wife for her tolerance, patience, and moral support. I am also grateful for her future supports that she will provide for me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Images are produced in order to record or display useful information. However, due to imperfections in the electronic or photographic medium, the recorded image is sometimes distorted. The distortions may have many causes, but two types of causes are often dominant: blurring and noise. *Blurring* is a form of bandwidth reduction of the image due to imperfect image formation process. It can be caused by the relative motion between the camera and the original scene, by an optical system out of focus, by an atmospheric turbulence for aerial photographs, or by spherical aberrations of the electron lenses for electron micrographs. In addition to blurring effects, the recorded image can also be corrupted by *noises*. These may be introduced by the recording medium (i.e. film grain noise), transmission medium (i.e. a noisy channel), measurement errors due to the limited accuracy of the recording system, and quantization of the data for digital storage.

Image restoration is the estimation of the original scene from a distorted and noisy one. The characteristics of the degrading system and the noise are assumed to be known *a priori* for the image restoration methods [18].

Since the introduction of restoration in digital image processing in 1960s, a variety of image restoration methods have been developed. One class of the proposed methods is *iterative methods*. The advantage of using iterative algorithms is

that they allow a flexible and improved formulation to the solution of the restoration problem [18]. Furthermore, they are the most straightforward to implement and the large dimensions involved in image restoration also makes these methods favorable. However, the drawback of iterative methods is the computational demand.

One way of formulating an iterative solution to the restoration problem is to convert the problem into a *linear feasibility problem* [5] and then using iterative techniques to find a feasible point of the system. The linear feasibility problem is to find a feasible point with respect to a set of linear inequalities. In matrix notation, the problem can be formulated as follows. Given an $M \times N$ matrix A and $b \in R^M$, find a feasible point $x \in R^N$ such that

$$Ax \leq b. \tag{1.1}$$

Note that, for an $m \times n$ image, A in Eq. 1.1 is an $mn \times mn$ sparse matrix which has an irregular sparsity pattern. So, even for small size images, A becomes very large.

One class of commonly used iterative methods for the linear feasibility problem is the *projection methods*. This class of methods has been developed for the solution of the equality systems in 1930s by Kacmarz (cited in [26]), and Cimmino (cited in [5]). Later, Gubin et al. [9] extended Kacmarz's work, and Censor and Elfving [6] extended Cimmino's work to linear inequalities. Gubin's technique has been known in the literature as the *successive orthogonal projections method*. In this method, an initial guess is successively projected onto hyperplanes corresponding to the boundary of the violated constraints until a feasible point is found which satisfies all the constraints. Censor and Elfving's method has been known as the *simultaneous orthogonal projections method*. In this method, the current point is projected onto each of the violated constraint's hyperplanes simultaneously, and the new point is taken to be the convex combination of all projections.

It is impossible to adopt both approaches to image restoration process since the dimensions involved are very large. Both methods become computationally very expensive since a projection is made for each violated constraint from the

current point. The surrogate constraint methods proposed by Yang and Murty [29] eliminate this problem by processing a group of violated constraints at a time. In each iteration, instead of performing projections onto each violated constraint, a surrogate constraint is derived from a group of violated constraints and block projections are carried out. The current point is orthogonally projected onto this surrogate constraint treated as an equation, and the process is repeated until a feasible solution is found.

Three kind of surrogate constraint methods have been proposed by Yang and Murty in [29]. The *Basic Surrogate Constraint Method* (BSCM) takes all violated constraints in the system and makes successive projections of the current point. *Sequential Surrogate Constraint Method* (SSCM) and *Parallel Surrogate Constraints Method* (PSCM) on the other hand, work on a small subset of the violated constraints. SSCM is based on successive block projections, while PSCM is based on simultaneous block projections. However, PSCM converges very slowly compared to SSCM. To compensate this, Özaktaş et al. [23] introduced an adjusted step sizing rule and proposed an improved version of PSCM. Instead of taking the convex combination, block projections are added up, and then to increase the movement of the current point towards the feasible area, a step size adjustment rule is used.

Both of the works by Yang and Murty and Özaktaş et al. concentrated on SSCM and PSCM. However, considering the sequential behavior, even though requiring a large number of iterations, the simplicity of BSCM makes it faster than SSCM. Moreover, developments in the hardware technologies allow us to work on whole matrices instead of the smaller blocks.

On the other hand, recent developments in the parallel processing techniques allow us to efficiently parallelize the basic method, instead of performing simultaneous projections. So, in this study, we implemented both a parallel version of BSCM and the improved version of PSCM, then compared these with the sequential version of BSCM.

As well as the parallel algorithm employed, efficiency of a parallel system depends on the distribution of the data and the work among the processors. A good

partitioning scheme should evenly distribute the computations over the processors while minimizing interprocessor communication. For the problem at hand, computational times of the methods are dominated by the sparse matrix-vector multiplies, so the load of the processors depends on the number of nonzero elements in a partition. Hence, the matrix should be distributed to processors in such a way that each processor ends up with approximately the same number of nonzero elements. On the other hand, for the parallel system used, the interprocessor communication time depends on the number of elements communicated (volume of communication), the message latency (start-up time) which depends on the number of messages communicated, and maximum volume and number of messages handled by any *single* processor [11].

For the first parallelization scheme utilized, namely uniform row-wise striped partitioning, four of the mentioned communication-cost metrics are tried to be minimized, based on the *communication-hypergraph partitioning model* proposed by Uçar and Aykanat [2]. This model maintains the computational load balance as well. In the first phase of the partitioning process, using existing 1D partitioning methods, total message volume is minimized and load balance is maintained. Then, this partitioning is input to the second phase, in which the remaining three communication-cost metrics are encapsulated, while trying to attain the total message-volume bound as much as possible.

The second type of parallelization scheme is based on 2D checkerboard partitioning of matrix A. A two-phase multi-constraint hypergraph partitioning method is used [4] with two kinds of cutsize metrics. First, *connectivity* cutsize metric is used in the partitioning process, and a point-to-point, local communication scheme is employed. In this communication scheme, each processor sends(receives) the required vector components from the processors in its connectivity set for the expand(fold) operation. For a mesh of $K = r \times c$ processors, the proposed method enforces an upper bound of $r + c - 2$ on the number of messages handled by a single processor, while explicitly minimizing the communication volume. Secondly, the *cut net* metric is used in the partitioning process. In this way, matrix A is divided row-wise and column-wise into internal and external parts. The internal parts allow independent computations while external

parts induce interprocessor communications. Here minimizing cut net metric corresponds to minimizing the size of the external parts. The required expand and fold operations are carried out in the external parts using an efficient all-to-all broadcast operation based on the combining algorithm developed by Jacunski et al. for switch based clusters of workstations [14].

Final parallelization scheme employed is based on 2D decomposition of the problem in a fine-grain manner. The fine-grain hypergraph model proposed in [3] is used in the partitioning process. By using a point-to-point, local communication scheme, this method substantially decreases the communication volume.

The proposed methods are validated by restoring severely blurred images. Images are blurred using isotropic scaling, rotation motion, and a combined motion including translational and rotation motion, as well as isotropic scaling. Then they are deblurred using proposed parallel implementations.

The organization of the rest of the thesis is as follows: The next chapter gives a number of fundamentals issues related to the image restoration problem and clarifies its relation with the feasibility problem. Furthermore, some iterative methods used in the linear feasibility problems are reviewed. Chapter 3 introduces surrogate constraint methods. Then, Chapter 4 and Chapter 5 gives parallel implementations based on 1D and 2D partitioning schemes, respectively. Chapter 6 presents experimental results and evaluates restoration and parallel performance of the implementations. Finally, the thesis is concluded in Chapter 7.

Chapter 2

Image Restoration Problem

This chapter summarizes a number of fundamental issues related to the image restoration problem. First section describes digital image representation and introduces a few concepts related to digital images. Then, image restoration problem is defined and some restoration techniques are reviewed. The third section presents linear feasibility problem and clarifies its relation with the restoration problem. The chapter concludes with a discussion of the iterative techniques developed for the solution of the linear feasibility problem.

2.1 Digital Image Representation

Images are denoted by two-dimensional functions of the form $f(x, y)$. For monochromatic images, the value or amplitude of f at spatial coordinates (x, y) is a positive scalar quantity. We call that value of monochrome image at any coordinates (x, y) the *gray level* (l) of the image at that point. That is,

$$l = f(x, y). \tag{2.1}$$

The interval that l takes is called the *gray scale*. Common practice is to shift this interval numerically to the interval $[0, L-1]$, where $l = 0$ is considered black

and $l = L - 1$ is considered white on the gray scale. All intermediate values are shades of gray varying from black to white.

An image may be continuous with respect to x- and y-coordinates, and also in amplitude. Digital images are produced by sampling in both coordinates and in amplitude. Digitizing the coordinate values is called *sampling*. Digitizing the amplitude values is called *quantization*.

The result of the sampling and quantization is a matrix of real numbers. Suppose that an image $f(x, y)$ is sampled so that the resulting digital image has m rows and n columns. So the complete $m \times n$ image can be written in the following matrix form:

$$f(x,y) = \begin{bmatrix} f(1,1) & f(1,2) & \dots & f(1,n) \\ f(2,1) & f(2,2) & \dots & f(2,n) \\ \vdots & \vdots & \vdots & \vdots \\ f(m,1) & f(m,1) & \dots & f(m,n) \end{bmatrix} \tag{2.2}$$

Digitization process requires decisions about values for m, n, and for the number L, of discrete gray levels allowed for each pixel. There are no requirements on m and n, however due to processing, storage, and sampling hardware considerations, the number of gray levels is a power of two:

$$L = 2^k \tag{2.3}$$

The number, b, of bits required to store a digitized image is

$$b = mnk. \tag{2.4}$$

When $m = n$, this equation becomes

$$b = n^2 k \tag{2.5}$$

So, we can deduce that storage requirement for an $n \times n$ pixel monochromatic image is $\Theta(n^2)$ [8].

2.2 Digital Image Restoration

Due to imperfections in the electronic or photographic medium, the recorded image often represents a degraded version of the original scene. The degradations may have many causes, but two types of degradations are often dominant: blurring and noise. *Blurring* is a form of bandwidth reduction of the image due to image formation process. It can be caused by relative motion between the camera and the original scene, or by an optical system which is out of focus. Other blurring sources are nonlinearity of the electro-optical sensor, atmospheric turbulence in remote sensing or astronomy, spherical aberrations of the electron lenses for electron micrographs, and X-ray scatter for CT scans [18].

In addition to the blurring effects, the recorded image may also be corrupted by the *noise*. The noise may be introduced by the recording medium, transmission medium, measurement errors due to the limited accuracy of the recording system, and quantization of the data for digital storage.

The field of *image restoration* (also referred to as image deblurring or image recovery) is concerned with the recovery or estimation of the uncorrupted image from a distorted and noisy one. Essentially, it tries to perform an operation on the image which is the inverse of the imperfections in the image formation system.

In the use of image restoration methods, the characteristics of the degrading system and the noises are known to be *a priori*. Estimation of the properties of the imperfect imaging system is the subject of *image identification* [18].

In this work, we study restoration of monochromatic images. Extension of the methods to color images is straightforward if the color image is described by a vector with three components corresponding to the tri-stimilus values, red, green, and blue. Considering each of these as a monochromatic image itself, and neglecting mutual relations between the color components, the processing of a color image becomes equivalent to processing three independent monochromatic images [18].

2.2.1 Formulation of the Problem

Image formation is generally described by a linear spatially invariant relation and the noise is considered to be additive [18]. The observed or recorded image $g(i,j)$ is then given as

$$g(i,j) = d(i,j) * f(i,j) + w(i,j), \tag{2.6}$$

where $d(i,j)$ denotes the point-spread function of the image formation system, $f(i,j)$ is the ideal or original image that would have resulted from a perfect recording of the original scene, and $w(i,j)$ models the noise in recording image.

In terms of the mathematical model in Eq. 2.6, the purpose of image restoration can now be specified as the computation of an estimate $\hat{f}(i,j)$ of the original image $f(i,j)$ when $g(i,j)$ is observed, and some (statistical) knowledge of both $d(i,j)$ and $w(i,j)$ is available.

In [24], a more general formulation of the problem is given which can model a rather general class of image recovery problems having the following properties:

(i) *Nonseparable.* The two dimensions are coupled and the problem cannot be reduced to two one-dimensional problems.

(ii) *Anisotropic.* The distortion is different along different directions.

(iii) *Space Variant.* The distortion is different for different parts of the image; it is not space invariant.

(iv) *Nonlocal.* The value of the distorted image at a certain point may depend on values of the original image at distant points.

According to this formulation the image $g(r)$ recorded on the film is given by

$$g(r) = K \int_{t=0}^{T} f(\rho(r,t))\mathrm{d}t \tag{2.7}$$

where r denotes position vector (x,y), K is a constant, $\rho(r,t)$ represents the time varying, nonlinear distortion which can model the following type of motions:

(i) *Translational Motion.* $\rho(r,t) = r - r(t)$ where $\rho(r,t)$ is a given function representing the motion of the original image or camera as a function of time. Arbitrarily two dimensional motions with arbitrarily accelerations are possible.

(ii) *Isotropic Scaling.* $\rho(r,t) = r/m(t)$ where $m(t)$ is an arbitrarily scaling function of time. By properly choosing $m(t)$, it is possible to model the movement of the object towards or away from the camera.

(iii) *Rotation.* $\rho(r,t) = R_{\phi(t)}r$ where $R_{\phi(t)}r$ is the 2×2 rotation matrix $[\cos\phi(t), \sin\phi(t); -\sin\phi(t), \cos\phi(t)]$. Here, $\phi(t)$ is an arbitrary function of time representing the angle of rotation.

Other special cases and their combinations may also be considered.

Since, Eq. 2.7 represents a linear relation between g and f, it is possible to write it in the form

$$g(r) = \int_{r'} H(r,r')f(r')\mathrm{d}r', \tag{2.8}$$

where $H(r,r')$ represents the blurring system. To find $H(r,r')$ we use,

$$f(\rho(r,t)) = \int_{r'} f(r')\delta(r' - \rho(r,t))\mathrm{d}r' \tag{2.9}$$

in Eq. 2.7 to obtain

$$g(r) = \int_{r'} \left[K \int_{t=0}^{T} \delta(r' - \rho(r,t))\mathrm{d}t\right] f(r')\mathrm{d}r', \tag{2.10}$$

where K is a constant. Comparing Eq. 2.8 and Eq. 2.10 we conclude that,

$$H(r,r') = K \int_{t=0}^{T} \delta(r' - \rho(r,t))\mathrm{d}t. \tag{2.11}$$

In discrete domain this can be written as,

$$H[r,r'] = K \sum_{k=0}^{K} \delta[r' - \rho[r,t]]. \tag{2.12}$$

Equation 2.8 can also be written in discrete domain in the following form

$$g[r] = \sum_{r'} H[r,r']f[r']. \tag{2.13}$$

If the columns of the f and g matrices are stacked on top of each other, Eq. 2.13 can be converted to the matrix-vector multiplication form:

$$g = Hf, \tag{2.14}$$

where H is an $mn \times mn$ matrix for an $m \times n$ pixel image. For example, 800×600 pixel image produces a matrix H which has a size of 480000×480000. Normally, matrices of this size cannot be handled if they are not sparse.

2.2.2 Image Restoration Methods

All image restoration methods concentrate on inverting Eq. 2.6 in order to get an estimate of $\hat{f}(i,j)$ which is "as close as possible" to the original image $f(i,j)$. Image restoration methods can be classified as frequency domain methods, recursive methods, and iterative methods. *Frequency domain* methods are the most restrictive restoration techniques which can handle only space-invariant blurs. It restores an image by applying inverse filtering. *Recursive methods* make use of the recursive filters all of which are based on Kalman filtering. Recursive filters have also some restrictions on the modeling and have difficulty of imposing nonlinear constraints on the restoration result [18].

Iterative methods produce a sequence of vectors $x^{(0)}$, $x^{(1)}$, $x^{(2)}$, ... such that the sequence *converges* to $x^* = \hat{f}$ for a finite number of iterations. Of the three categories, iterative restoration methods are the most straightforward to implement and the most flexible to apply. The large dimensions involved in image restoration also makes these methods favorable.

The primary drawback of the iterative schemes is the computational demand. The most time consuming computational steps are matrix-vector products. The size of the matrices are large and obtaining a restoration may be much more expensive compared to recursive or frequency-domain methods.

2.3 Image Restoration and the Linear Feasibility Problem

2.3.1 The Linear Feasibility Problem

The linear feasibility problem is how to find a point in the non-empty intersection $\Phi = \cap_{i=1}^{m} \Phi_i \neq \emptyset$ of a finite family of convex sets $\Phi_i \subseteq R^n$, $i \in I = \{1, 2, \ldots, m\}$ in the n-dimensional Euclidean space R^n. When the sets are given in the form

$$\Phi_i = \{x \in R^n \mid y_i(x) \leq 0, \quad y_i \text{ is a convex function}\}, \tag{2.15}$$

we are faced with the problem of solving a system of inequalities with convex functions, of which the linear case, (i.e. $y_i(x) = A_i x - b_i$, where A_i is the ith row of an $m \times n$ constant matrix A) is an important and special case [5].

2.3.2 Formulation of the Restoration Problem as a Linear Feasibility Problem

Consider Eq. 2.14. Defining a suitable error tolerance parameter ϵ for the additive noise, we can rewrite the equation as

$$|(g' - Hf)_i| < \epsilon \qquad i = 1, \ldots, MN \tag{2.16}$$

where $(g' - Hf)_i$ is the ith component of $g' - Hf$. For $i = 1, \ldots, MN$, $|(g' - Hf)_i| < \epsilon$ implies that

$$\begin{array}{rcll} g'_i - Hf_i & < & \epsilon & \text{if } g'_i > Hf_i \\ -g'_i + Hf_i & < & \epsilon & \text{if } g'_i < Hf_i \end{array} \tag{2.17}$$

or,

$$\begin{array}{rcll} Hf_i & < & \epsilon + g'_i & \text{if } g'_i < Hf_i \\ -Hf_i & < & \epsilon - g'_i & \text{if } g'_i > Hf_i \end{array} \tag{2.18}$$

So, Equation Eq. 2.16 can be converted to a linear feasibility problem of the form $Ax \leq b$ by setting

$$A = \begin{bmatrix} H \\ -H \end{bmatrix}_{2MN \times MN}, \quad x = f_{MN \times 1}, \quad b = \begin{bmatrix} \varepsilon + g' \\ \varepsilon - g' \end{bmatrix}_{2MN \times 1} \tag{2.19}$$

where ε is an $MN \times 1$ vector of ϵ's.

2.4 Iterative Methods for Linear Feasibility Problems

Iterative methods are commonly used to solve inequality systems since they are based on simple computation steps and easy to program. Moreover, large dimensions confronted in image restoration and image reconstruction problems prohibit the use of direct methods. So, linear inequality solvers used in these areas are often based on iterative methods.

One class of iterative methods is the projection approach. Projection methods are based on the works of Kacmarz [15] and Cimmino [7] to solve linear equations. In Kacmarz's approach, *successive* projections are made onto hyperplanes which represent linear equations. This approach is highly sequential in nature. Cimmino's method makes *simultaneous* projections onto hyperplanes and allow a certain degree of parallelism.

Gubin et al. [9] developed the method of successive orthogonal projections (also known as the method of projections onto convex sets) which is an extension of the Kacmarz's method for solving linear inequality systems. In this method, a violated constraint is identified, and a projection is made onto violated convex sets successively. An orthogonal projection onto a single linear constraint is computationally inexpensive, but since this step is carried out for each constraint, the overall computation becomes costly for large systems. Moreover, this method is not suitable for parallel implementations.

On the other hand, Censor and Elfving [5] developed a Cimmino-type algorithm for linear inequalities. This method makes orthogonal projections simultaneously onto each of the violated constraints from the current point and takes the new point as a convex combination of those projection points. This method is

suitable for parallel implementation but making projections for each of the constraints is again not desirable. Moreover, instead of accumulating the projections, taking convex combination of them results in slow convergence.

Later, Yang and Murty proposed *surrogate constraint methods* which makes block projections by considering all or a subset of the violated constraints [29]. Surrogate constraint methods are able to process a group of constraints at the same time while retaining the computational simplicity of the projection methods. Moreover, they are amenable to parallel implementations. In the following chapter we will discuss these methods in more detail.

Chapter 3

Surrogate Constraint Methods

In order to solve the image restoration problem, we will concentrate on finding a feasible solution to the system,

$$Ax \leq b, \tag{3.1}$$

where A is an $M \times N$ matrix, x is a $N \times 1$ vector, and b is an $M \times 1$ vector. If A_i and b_i are the ith row of the system, then $A_i x \leq b_i$ defines the ith constraint.

As discussed in Section 2.4, to solve the system in Eq. 3.1, if at step t the current point x^t violates the ith constraint of the system, successive orthogonal projections method developed by Kacmarz [15] successively projects x^t onto hyperplanes $A_i x^t = b_i$ and generates the next point x^{t+1} as follows,

$$x^{t+1} = x^t - d^t, \tag{3.2}$$

where the projection vector d^t is computed as

$$d^t = \frac{(A_i x^t - b_i)}{||A_i||^2} A_i^T. \tag{3.3}$$

On the other hand, instead of dealing with each of the violated constraints, surrogate constraint methods proposed by Yang and Murty [29] derive surrogate hyperplanes from a set of the violated constraints and then take the projection of the current point onto surrogate hyperplanes. Surrogate hyperplanes eliminate the drawback of making projections for each of the violated constraints.

Among the methods proposed by Yang and Murty, *basic surrogate constraint method* (BSCM) derives surrogate hyperplanes from all of the violated constraints in the system, whereas *sequential surrogate constraint method* (SSCM) and *parallel surrogate constraint method* (PSCM) consider a subset of the constraints. In this chapter we will present these methods and discuss their performances.

3.1 Basic Surrogate Constraint Method

Basic surrogate constraint method is the simplest method proposed in [29]. This method combines all of the violated constraints and makes just one projection in each cycle as follows: If the current point x^t at step t violates the system in Eq. 3.1, an $1\times M$ weight vector π is generated such that $0 < \pi_i < 1$ if the ith constraint is violated (i.e. for i such that $A_i x^t > b_i$), and $\pi_i = 0$ otherwise. Moreover, for convenience we let $\sum_{i=1}^{M} \pi_i = 1$. Then, the surrogate constraint $(\pi A)x^t \leq (\pi b)$ is generated for which the corresponding surrogate hyperplane is

$$H_s = \{x : (\pi A)x^t = (\pi b)\}.$$

The next point x^{t+1} is generated by projecting x^t onto this surrogate hyperplane as follows:

$$x^{t+1} = x^t - \lambda d, \tag{3.4}$$

where the projection vector d is computed as,

$$d = \frac{\pi A x^t - \pi b}{||\pi A||^2}(\pi A)^T, \tag{3.5}$$

and λ is the relaxation parameter that determines the location of the next point which is in the line segment joining the current point and its reflection on the hyperplane. When $\lambda = 1$ the next generated point is the exact orthogonal projection of the current point. If $0 < \lambda < 1$, the step taken is shorter which refers to underrelaxation case and if $1 < \lambda < 2$, then the step taken is longer which refers to the overrelaxation case [5].

An important issue for the solution of the problem is the selection of the weight vector π. Weights may be distributed equally among all violated constraints or

they can be assigned in proportion to the amount of violations. We prefer to use the hybrid approach and compute the ith element of π as:

$$\pi_i = \begin{cases} w_1(A_i x^k - b_i)/\sum_{i=1}^{VC}(A_i x^k - b_i) + w_2/VC & \text{if } A_i x^k - b_i > 0 \\ 0 & \text{otherwise} \end{cases} \tag{3.6}$$

where w_1 and w_2 are two appropriate weights summing up to 1, and VC is the number of violated constraints at step t.

For convenience, we give in Fig. 3.1 the pseudocode of the serial version of BSCM method. The notations used in the algorithm are as follows:

- $\boldsymbol{A}$ is an $M\times N$ matrix containing Z non-zero entries. ($\boldsymbol{A} = \boldsymbol{H}$, see Eq. 2.19).
- $\boldsymbol{b^+} = \boldsymbol{\varepsilon} + \boldsymbol{g'}$ and $\boldsymbol{b^-} = \boldsymbol{\varepsilon} - \boldsymbol{g'}$ are $M\times 1$ vectors (see Eq. 2.19).
- $\boldsymbol{\pi}$, $\boldsymbol{\pi^+}$, and $\boldsymbol{\pi^-}$ are $1\times M$ vectors.
- $\boldsymbol{\delta^+}$ and $\boldsymbol{\delta^-}$ are $M\times 1$ vectors.
- $\boldsymbol{q}$, $\boldsymbol{d}$, and $\boldsymbol{x}$ are $N\times 1$ vectors.
- μ, γ, and λ are scalars.

Since our system in Eq. 3.1 is composed of the same matrices with different signs (see Eq. 2.19 in Section 2.3.2), only the positive H matrix is held during the computations. We call the system $Hx \leq \varepsilon + g$ as the upper system and $-Hx \leq \varepsilon - g$ as the lower system. Since $q^T = \pi^+ A + \pi^-(-A) = (\pi^+ - \pi^-)A$, we form the vector $\pi = \pi^+ - \pi^-$ and then perform the multiplication. To compute $y^- = -Ax$, simply $y = Ax$ is negated. In this way, we have also saved computational time since the sparse matrix vector multiplications $-Ax$ and $\pi^-(-A)$ are not performed.

In terms of the number of floating point operations (which are scalar addition, subtraction, and multiplication), the computational time of a single iteration of BSCM is:

$$T_s = (4Z + 12M + 5N)t_{flop}. \tag{3.7}$$

while *true* do		
$\boldsymbol{y} \leftarrow \boldsymbol{Ax}$	, multiply A from right	$\{t = 2Z\}$
$\boldsymbol{\delta}^+ \leftarrow \boldsymbol{y} - \boldsymbol{b}^+$	, error of the upper system	$\{t = M\}$
$\boldsymbol{\delta}^- \leftarrow -\boldsymbol{y} - \boldsymbol{b}^-$	, error of the lower system	$\{t = M\}$
$\boldsymbol{\pi}^+ \leftarrow$ updatePi($\boldsymbol{\delta}^+$)	, update π^+ using Eq. 3.6	$\{t = 3M\}$
$\boldsymbol{\pi}^- \leftarrow$ updatePi($\boldsymbol{\delta}^-$)	, update π^- using Eq. 3.6	$\{t = 3M\}$
if $\boldsymbol{\pi}^+ = \boldsymbol{0}$ and $\boldsymbol{\pi}^- = \boldsymbol{0}$ then	, check convergence	
exit		
$\boldsymbol{\pi} \leftarrow \boldsymbol{\pi}^+ - \boldsymbol{\pi}^-$	, compute π	$\{t = M\}$
$\boldsymbol{q} \leftarrow (\boldsymbol{\pi A})^T$	, multiply A from left	$\{t = 2Z\}$
$\mu \leftarrow \boldsymbol{\pi}^+\boldsymbol{\delta}^+ + \boldsymbol{\pi}^-\boldsymbol{\delta}^-$	, sum of inner products	$\{t = 3M\}$
$\gamma \leftarrow \boldsymbol{q}^T\boldsymbol{q}$	, inner product	$\{t = 2N\}$
$\boldsymbol{d} \leftarrow \mu/\gamma \ \ \boldsymbol{q}$	, compute projection vector	$\{t = N\}$
$\boldsymbol{x} \leftarrow \boldsymbol{x} - \lambda\, \boldsymbol{d}$	, update x	$\{t = 2N\}$
endwhile		

Figure 3.1: Basic surrogate constraint method

Verification of the convergence of the algorithm is based on the Fejer-monotonicity of the generated sequence $\{x^t\}_{k=0}^{\infty}$. If the feasibility check is allowed a certain degree of tolerance ϵ, so that $A_i x^t$ is compared with $b_i + \epsilon$, then the algorithm converges after a finite number of iterations [29].

3.2 Sequential Surrogate Constraint Method

Instead of working on the whole A matrix, sequential surrogate constraint method partitions the system in Eq. 3.1 into subsystems and then solves the feasibility problem by applying the basic method on the subsystems in a cyclic order. Specifically, let the matrix A be partitioned into K submatrices, and the right-hand

side vector b be partitioned conformably into K subvectors, as follows:

$$A = \begin{bmatrix} A_1 \\ \vdots \\ A_k \\ \vdots \\ A_K \end{bmatrix}, b = \begin{bmatrix} b_1 \\ \vdots \\ b_k \\ \vdots \\ b_K \end{bmatrix}, \tag{3.8}$$

where A_k is an $m_k \times N$ matrix having z_k nonzeros, and

$$\sum_{k=1}^{K} m_k = M, \quad \sum_{k=1}^{K} z_k = Z.$$

Surrogate constraints are defined as $\pi_k A_k x \leq \pi_k b_k$ for each block where π_k is an $1 \times m_k$ vector as defined before. Sequential surrogate constraint method solves the system in Eq. 3.1 by projecting current point onto surrogate hyperplanes $(\pi_k A_k)x = \pi_k b_k$ successively for $k = 1, \ldots, K$ in cyclic order. For each block, the next point is generated as:

$$x^{t+1} = x^t - \lambda d_k^t, \tag{3.9}$$

where d_k^t is the projection vector of block k, which is computed as,

$$d_k^t = \frac{\pi_k^t (A_k x^t - b_k)}{||\pi_k^t A_k||^2} (\pi_k^t A_k)^T. \tag{3.10}$$

The pseudocode given in Fig. 3.2 clarifies the steps followed in the method.

In terms of the number of floating operations, per iteration time of the method is:

$$T_s = \sum_{k=1}^{K} (4z_k + 12m_k + 5N) t_{flop} \tag{3.11}$$

$$T_s = (4Z + 12M + 5NK) t_{flop} \tag{3.12}$$

We see that the computational time of SSCM increases with the increasing number of blocks. Each block brings an extra computation time of $5N$ which can be a serious overhead for sparse systems.

```
while true do
   for k = 1 to K do
      y_k ← A_k x                                  , multiply A_k from right          {t = 2z_k}
      δ_k^+ ← y_k − b_k^+                          , error of the upper system        {t = m_k}
      δ_k^− ← − y_k − b_k^−                        , error of the lower system        {t = m_k}
      π_k^+ ← updatePi(δ_k^+)                      , update π^+ using Eq. 3.6         {t = 3m_k}
      π_k^− ← updatePi(δ_k^−)                      , update π^− using Eq. 3.6         {t = 3m_k}
      π_k ← π_k^+ − π_k^−                          , compute π_k                      {t = m_k}
      q_k ← (π_k A_k)^T                            , multiply A from left             {t = 2z_k}
      μ_k ← δ_k^+ π_k^+ + δ_k^− π_k^−              , sum of inner products            {t = 3m_k}
      γ_k ← q_k^T q_k                              , inner product                    {t = 2N}
      d_k ← μ_k/γ_k  q_k                           , compute projection vector        {t = N}
      x ← x − λ d_k                                , update x                         {t = 2N)}
   endfor
   if π_k^+ = 0 and π_k^− = 0 ∀ k then             , check convergence
      exit
endwhile
```

Figure 3.2: Sequential surrogate constraint method

3.3 Parallel Surrogate Constraint Method

In the sequential surrogate constraint method each point x^t that will be projected in block k is a result of the projection of x^{t-1} performed in the preceding block $k-1$. Thus, successive block projection imply a dependency between the blocks of the system, causing the employed algorithm to be highly sequential. So, for the parallel version of the surrogate constraint algorithm, Yang and Murty proposed a Cimmino type algorithm [7] which carries out simultaneous block projections and generates the next point as a convex combination of the block projections.

As in the case of SSCM, we will assume that matrix A is divided row-wise into K contiguous blocks as shown in Eq. 3.8. In iteration t of the parallel method, the next point x^{t+1} is computed as follows:

$$x^{t+1} = x^t - \lambda \sum_{k=1}^{K} \tau_k d_k^t, \tag{3.13}$$

where τ_k are nonnegative numbers summing up to 1, and the kth projection is defined in the same way as Eq. 3.10.

In Eq. 3.13 each projection has its own influence on the next point. This influence can be taken into account so as to accelerate the convergence of the system. Hence, τ_k can be taken to be proportional to the number of violated constraints or the cumulative error of the respective block.

However, as clarified in [22], no matter how τ_k is chosen, the movement of this original parallel routine is much shorter than the movement in the sequential case, leading to slow convergence. Actually this method is a variance of the Cimmino type algorithms which suffers from slow convergence. To compensate this behavior, Garcia Palomeras proposed an acceleration procedure for the Cimmino type algorithms [25]. They give an improved step for the generation of the next point. Later, Özaktaş et al. used this step sizing rule in the parallel surrogate algorithm and generated the next point as follows [23]:

$$x^{t+1} = x^t - \lambda \frac{\sum_{k=1}^{K} ||d_k^t||^2}{||\sum_{k=1}^{K} d_k^t||^2} \sum_{k=1}^{K} d_k^t, \tag{3.14}$$

where d_k^t is defined as in Eq. 3.10.

With this modification, the step sizes taken are enlarged so that the parallel method converges much more rapidly. In Fig. 3.3 we give the coarse-grain parallel formulation of the improved parallel surrogate method.

In terms of the floating point operations per iteration, the run time of the algorithm is

$$T_s = \sum_{k=1}^{K} (4z_k + 12m_k + 6N)t_{flop} + 4Nt_{flop} = (4Z + 12M + 6NK + 4N)t_{flop}. \tag{3.15}$$

3.4 Comparison of the Methods

As discussed in [23], accumulating the projections instead of taking convex combinations speeds up the convergence of SSCM compared to the original PSCM

```
while true do
    α ← 0
    d ← 0
    for k = 1 to K do
        y_k ← A_k x                          , multiply A_k from right          {t = 2z_k}
        δ1_k^+ ← y_k − b1_k                  , error of the upper system        {t = m_k}
        δ_k^− ← − y_k − b_k^−                , error of the lower system        {t = m_k}
        π_k^+ ← updatePi(δ_k^+)              , update π^+ using Eq. 3.6         {t = 3m_k}
        π_k^− ← updatePi(δ_k^−)              , update π^− using Eq. 3.6         {t = 3m_k}
        π_k ← π_k^+ − π_k^−                  , compute π_k                      {t = m_k}
        q_k ← (π_k A_k)^T                    , multiply A from left             {t = 2z_k}
        μ_k ← δ_k^+ π_k^+ + δ_k^− π_k^−      , sum of inner products            {t = 3m_k}
        γ_k ← q_k^T q_k                      , inner product                    {t = 2N}
        d_k ← μ_k/γ_k q_k                    , compute projection vector        {t = N}
        α ← α + d_k^T d_k                    , inner product sum                {t = 2N}
        d ← d + d_k                          , sum the projection vectors       {t = N}
    endfor
    if π_k^+ = 0 and π_k^− = 0 ∀k then       , check convergence
        exit
    β ← d^T d                                , inner product                    {t = 2N}
    x ← x − λ α/β d                          , update x                         {t = 2N}
endwhile
```

Figure 3.3: Coarse grain formulation of parallel surrogate constraint method

of Yang and Murty. We also observed that, in SSCM and improved PSCM, as the number of blocks increases, the number of iterations required for convergence decreases in general. However, as we showed in the corresponding sections, each block brings an extra computational time of $5Nt_{flop}$ and $6Nt_{flop}$, for SSCM and PSCM, respectively. These are serious overheads for sparse systems. In fact, they are in the order of a sparse-matrix vector multiply for the systems used in the restoration process. Hence, by increasing number of blocks even though the number of iterations decreases, the computational time increases for both SSCM and PSCM. We conclude that, for sequential implementations, BSCM is preferable to SSCM. However, considering the parallel implementations, modified version of PSCM can be preferable to BSCM, since PSCM converges with less number of

iterations compared to BSCM. Nonetheless, we implemented parallel versions of both BSCM and PSCM in order to make comparisons between them. We will consider serial version of BSCM in our performance analyses since it is the fastest sequential code among the surrogate constraint methods.

Chapter 4

Parallelization based on 1D Partitioning

In this chapter we will focus on parallelization of the surrogate constraint methods for 1D decomposition of the problem. The partitioning scheme is rowwise striped partitioning, as shown in Eq. 3.8. As mentioned in the previous chapter, we will perform parallel implementations of the basic surrogate constraint method and the parallel surrogate constraint method, for which a coarse-grain formulation is given.

The computational scheme and communication requirements of the surrogate constraint methods are common in many iterative methods used to solve unsymmetric linear systems [10, 28]. The kernel operation of these methods are repeated matrix-vector and matrix-transpose-vector multiplies in the form of $y = Ax$ and $w = A^T z$, where A is a sparse, unsymmetric, square or rectangular coefficient matrix. The input vectors and output vectors of these multiplications are obtained from each other through linear vector operations. That is, vector z, of the second multiply $w = A^T z$, is obtained from y; the output vector of the first multiply $y = Ax$, and vice versa. In a parallel implementation with a rowwise partitioning of data, $y = Ax$ multiplication requires an *expand* operation before the local

$y_k = A_k x$ multiply, in which each processor P_k sends some of its x-vector components to other processors that has a nonzero entry in the respective column. Similarly a *fold* operation is required to form w-vector after the local $w_k = A_k^T z_k$ multiply. In this operation, each processor P_k receives some of the w vector components and accumulates them. These two multiplications also take place in the surrogate constraints algorithms; $y = Ax$ multiply is used to check the feasibility of the system. Other multiplication $q = (\pi A)^T = A^T \pi^T$ is used to form surrogate hyperplanes. However, in PSCM, after the second multiply, we form the local projection vector d from q through some linear vector operations and then fold d vectors instead of q vectors, to form the sum of the projections. Hence, the communication pattern of BSCM is the same as that of PSCM.

In the literature, graph-theoretic decomposition methods have been developed to efficiently parallelize sparse-matrix vector multiplications. The aim is to minimize the the communication costs of the fold and expand operations while maintaining load balance. Among these methods, hypergraph partitioning based methods accurately models the communication requirements, and can handle unsymmetric partitioning as well.

So we have decomposed our problem data using hypergraph partitioning based methods. In the next section hypergraph partitioning based decomposition methods for matrix-vector multiplies will be introduced. Then parallel algorithms of BSCM and PSCM for 1D partitioning will be given. We will also mention some implementation details. Finally, performance of the parallel methods will be discussed.

4.1 Hypergraph Partitioning based 1D Decomposition

Decomposition is a preprocessing applied to a problem to minimize the overheads of parallel processing. There are mainly three sources of overhead for parallel

programs: Load imbalance, interprocessor communication, and extra computation [17]. *Load imbalance* occurs when different processors have different work loads during the parallel program execution. *Interprocessor communication* is the time to transfer data between processors and is usually the most significant source of parallel processing overhead. *Extra computations* takes place when the fastest known sequential algorithm for a problem is difficult or impossible to parallelize, and a poorer but easily parallelizable algorithm is preferred instead (this overhead will be discussed in Chapter 6).

To ensure computational load balance of a parallel system, work load should evenly be distributed among the processors. However, even if perfect load balance is achieved, interprocessor communication is an unavoidable overhead of parallel processing. For a parallel system with a cut-through routing scheme, the time t_{comm} to send a message of length l between two processors is $t_{comm} = t_s + lt_w$ where t_s is the latency or start-up time and t_w is the per-word transfer time. So, interprocessor communication time depends on the volume of communication and the number of messages communicated. Both the volume of communication and the number of messages communicated should be minimized to decrease the effect of communication overhead. Moreover, since the last terminating process of a parallel system determines the run time of a parallel program, the communication effort should be well balanced as well as the computational load. So, maximum volume and message handled by a single processor should also be considered during the decomposition process [11].

Because of these multi-constraints confronted in the decomposition problem, graph theoretical partitioning methods have been developed in the parallel processing community which can encapsulate data dependencies among the processors and the load distribution in the modeling. Minimization of the communication overhead while ensuring load balance is formulated as the well known *K-way graph partitioning problem*, where K is the number of processors in the target parallel architecture. Then existing *graph partitioning* methods are utilized to decompose the work and data for an efficient parallel computation. The computational load is represented by partition weights and interprocessor communication

is represented by edge crossings between the partitions. So, minimizing the number of edge crossings corresponds to minimizing the volume of communication, and ensuring balance between the parts corresponds to maintaining load balance among the processors.

The standard graph model though widely used, has some limitations and deficiencies in the modeling, as mentioned in [2, 11]. First of all, this model can only be used for symmetric square matrices, so it is not applicable to the solution of linear feasibility problems, in which rectangular matrices may arise since the output vector size (dimension of the problem) and the input vector size (number of constraints) are in general different. Even if the matrix is square, there is no computational dependency between the output and the input vectors, which is an assumption for this model. Furthermore, it tries to minimize wrong objective function, since the number of edge-crossings does not reflect the actual communication volume.

Later, Çatalyürek and Aykanat proposed the *computational hypergraph model* which can reflect the actual communication volume requirement and can handle a wide class of problems [2]. In their work, Çatalyürek and Aykanat concentrated on 1D symmetric decomposition of square matrices for parallel sparse-matrix vector multiplication, however, it can model non symmetric decomposition of square matrices and decomposition of rectangular matrices as well.

4.1.1 Hypergraph Partitioning Problem

A *graph* $G = (V, E)$ consists of a set of *vertices*, $V = \{v_1, v_2, \ldots, v_n\}$, and a set of pairwise relationships, $E \subset V \times V$, between the vertices which are called *edges*. A *hypergraph* $H = (V, N)$ is a generalization of a graph, such that an edge $n_i \in N$ can include more than two vertices and can be a subset of vertices, i.e. $n_i \subset V$. The term *net* is used instead of edge for hypergraphs. For the decomposition problem, the vertices of a graph or hypergraph represent atomic tasks, and the edges or nets encode data dependencies. Also, *weights* can be associated with the vertices to designate the amount of computation and *costs* can be associated with

edges or nets, to indicate the amount of dependency between the computations.

Definition of a K-way partitioning of a hypergraph can be made as follows: $\Pi = \{P_0, P_1, \ldots, P_{K-1}\}$ is a *K-way partition* of a hypergraph $H = (V, N)$ if the following conditions hold:

- each part P_k is a non-empty set of V, i.e. $P_k \subset V$ and $P_k \neq \emptyset$ for $0 \leq k \leq K-1$.
- union of K parts is equal to V, i.e. $\bigcup_{k=0}^{K-1} P_k = V$
- parts are pairwise disjoint, i.e. $P_k \cap P_l = \emptyset$ for $0 \leq k < l \leq K-1$.

A partition is said to be balanced if each part P_k satisfies the balance criterion

$$W_{avg}(1-\epsilon) \leq W_k \leq W_{avg}(1+\epsilon), \quad \text{for } k = 0, 1, \ldots, K-1 \tag{4.1}$$

where weight $W_k = \sum_{v_i \in P_k} w_i$ of a part is defined as the sum of the weights of vertices in the part P_k, $W_{avg} = (\sum_{v_i \in V} w_i)/K$ denotes the weight of each part under perfect load balance condition, and ϵ is the predetermined maximum imbalance ratio allowed.

In a partition Π of H, a net that has at least one pin (vertex) in a part is said to *connect* that part. *Connectivity set* Ψ of a net n_j is defined as the set of parts connected by n_j. *Connectivity* $\psi = |\Psi|$ of a net n_j denotes the number of parts connected by n_j. A net n_j is said to be *cut* if it connects more than one part (i.e. $\psi > 1$), and *uncut* otherwise (i.e. $\psi = 1$). The set of cut nets of a partition Π is denoted as N_ε. Two cutsize definitions used for the decomposition problem are:

$$\text{(a)} \quad \Upsilon(\Pi) = |N_\varepsilon| \qquad \text{and} \qquad \text{(b)} \quad \Upsilon(\Pi) = \sum_{n_j \in N_\varepsilon} (\psi_j - 1). \tag{4.2}$$

In Eq. 4.2.a, cut size is equal to the number of cut nets, and in Eq. 4.2.b, each cut net contributes $\psi_j - 1$ to the cut size. These two metrics are called as the *cutnet metric*, and as the *connectivity* metric [19]. With the help of these definitions hypergraph partitioning problem can be defined as the task of dividing a hypergraph into two or more parts so that cut size is minimized, while balance criterion in Eq. 4.1 is satisfied.

4.1.2 Column-net and Row-net Models

Two hypergraph models, namely *column-net* and *row-net* models have been proposed in [2] for rowwise and columnwise decomposition of sparse matrices, respectively.

In the column-net model, matrix A is represented by a hypergraph $H_R = (V_R, N_C)$ for rowwise decomposition of sparse matrices. Each row is represented by a vertex, and each column is represented by a net in the hypergraph. Net $n_j \subset V_R$ contains the vertices corresponding to the respective row entries which has a nonzero element. That is $v_i \in n_j$ if and only if $a_{ij} \neq 0$. Formally, a hypergraph $H_C = (V_R, N_C)$ is a column-net representation of an $M \times N$ sparse matrix A, if the following conditions hold:

- $V_R = \{v_1, v_2, \ldots, v_i, \ldots, v_M\}$, where v_i corresponds to the ith row of matrix A.
- $N_C = \{n_1, n_2, \ldots, n_j, \ldots, n_N\}$, where n_j corresponds to the jth column of matrix A.
- For each $v_i \in V_R$ and for each $n_j \in N_C$, $v_i \in n_j$ if and only if $a_{ij} \neq 0$.

Similarly, row-net hypergraph model $H_R = (V_C, N_R)$ can be used for columnwise decomposition of a sparse matrix $A \in R^{M \times N}$, if the following conditions hold:

- $V_C = \{v_1, v_2, \ldots, v_j, \ldots, v_N\}$, where v_j corresponds to the jth column of matrix A.
- $N_R = \{n_1, n_2 \ldots, n_i, \ldots, n_M\}$, where n_i corresponds to the ith row of matrix A.
- For each $v_j \in V_C$ and for each $n_i \in N_R$, $v_j \in n_i$ if and only if $a_{ji} \neq 0$.

In the column-net model, for the $y = Ax$ multiply, each vertex v_i corresponds to atomic task i of computing the inner product of row i with the x vector. For

the $q = \pi A$ multiply, v_i corresponds to atomic task i of computing the partial vector $q^i = \pi_i A_i$ where $q = \sum_{i=1}^{M} q^i$. For both operations we can assign the total number of nonzero elements in row i as the computational weight of w_i.

On the other hand, the nets of H_C represent dependency relations of the atomic tasks on the x-vector components during the fold and expand operations. Each net n_j contains the set of vertices (tasks) that needs x_j in the $y = Ax$ multiply and the set of vertices (tasks) that computes q_j in the $q = \pi A$ multiply.

In [2] it is shown that the proposed column-net model correctly reduces the rowwise decomposition problem to the K-way hypergraph partitioning problem. Minimizing the cut size corresponds to minimizing the actual communication volume, whereas maintaining the balance criterion corresponds to balancing the computational load among the processors. So, by equally distributing the vertices among the parts so that cut nets are split among as few processors as possible, the total communication volume of the fold and expand operations are minimized while computational load balance is maintained.

4.1.3 Minimizing Communication Costs

As well as the total communication volume, the message latency, (which depends on the number of messages communicated), maximum volume and number of messages handled by a *single* processor are the parallel overheads that should be considered in parallel implementations [11]. As mentioned before, there is no computational dependency between the input and output vectors for our problem. This fact avoids the restriction on the partitioning of input vector components contrary to symmetric partitioning. (In a symmetric partitioning we have to assign the corresponding input and output vector elements to the same processor.)

In order to handle the four cost factors, we used the two-phase approach proposed by Uçar and Aykanat [28]. In the first phase, this method tries to minimize total message volume using 1D hypergraph partitioning method while maintaining load balance among the processors. In the second phase, other metrics

are minimized using their *communication-hypergraph* partitioning model. With proper weighting, vertices of the communication-hypergraph represent primitive communication operations whereas nets represent processors. By partitioning the communication hypergraph into balanced parts so that nets are split among as few as vertex parts as possible, the model minimizes total number of messages communicated and also ensures the communication balance among the processors.

4.2 Parallel Algorithms

In this section we will present parallel methods of BSCM and PSCM for 1D rowwise decomposition. We assume that, matrix A is divided rowwise into K contiguous blocks, where K is the number of processors, and each processor P_k holds the kth row stripe for $k = 1, 2, \ldots, K$. The rowwise partitioning of matrix A defines a partition on the y-space vectors (vectors that goes into linear operations with the y-vector) as well, so each processor P_k holds and computes kth block of the y-space vectors. The x-space vectors (vectors that goes into linear operations with the x-vector) are also partitioned into K subvectors and processor P_k holds and computes kth block of the x-space vectors.

4.2.1 Parallelization of the Basic Method

In Fig. 4.1 we give the pseudocode of the parallel BSCM method. Each processor P_k accesses the following components in the algorithm:

- an $m \times N$ matrix $\boldsymbol{A_k}$ containing z non-zero entries, where $m \approx M/K$ and $z \approx Z/K$ are respectively, average number of rows and average number of nonzero elements in a block.
- x-space vectors:
  - $N \times 1$ global vector $\boldsymbol{x}$ which represents the current point.

- $n \times 1$ local vector $\boldsymbol{x_k}$ which is the kth block of x, where $n \approx N/K$ is the average length of x_k vectors.
- $N \times 1$ column vector $\boldsymbol{q^k}$ which is the partial vector resulting from the local matrix-vector multiply $\pi_k A_k$.
- $n \times 1$ column vector $\boldsymbol{q_k}$ which is the kth block of vector q that would result from the global matrix-vector multiply πA.
- $n \times 1$ column vector $\boldsymbol{d_k}$ which is the kth block of the projection vector d.

- y-space vectors:
  - $m \times 1$ local vectors $\boldsymbol{b_k^+}$ and $\boldsymbol{b_k^-}$ which are the right hand side vectors of our system.
  - $1 \times m$ local vectors $\boldsymbol{\pi_k}$, $\boldsymbol{\pi_k^+}$, and $\boldsymbol{\pi_k^-}$.
  - $m \times 1$ local vectors $\boldsymbol{\delta_k^+}$ and $\boldsymbol{\delta_k^-}$ which denotes the error in the upper and lower systems, respectively.

- scalars:
  - Global scalars μ and γ which are used to form the projection vector d.
  - Corresponding local scalars μ_k and γ_k.

Note that we use superscripts with vectors to indicate partial vectors and subscripts to denote the subvectors that reside in P_k.

As mentioned, the x-vector is partitioned among the processors so that each P_k is responsible to compute the kth block of the global x-vector, namely x_k. However, in order to check the feasibility of the subsystem and define a new surrogate plane in the infeasible case, processor P_k requires all x-vector components that corresponds to the nonzero columns of A_k. So an expand operation is performed at the beginning of the loop. Since we employ a point-to-point communication scheme, each processor sends some of its local vector components to the processors having a nonzero entry in that respective column. Then the local matrix-vector multiply $y_k = A_k x$ is performed and the error vectors are computed for the lower and upper systems. Then, local π_k vectors are generated

while *true* do		
$\boldsymbol{x} \leftarrow \text{expand}(\boldsymbol{x_k})$	, expand x_k vector	$\{t = t_{expand}\}$
$\boldsymbol{y_k} \leftarrow \boldsymbol{A_k x}$	, multiply A_k from right	$\{t = 2z\}$
$\boldsymbol{\delta_k^+} \leftarrow \boldsymbol{y_k} - \boldsymbol{b_k^+}$	, error of the upper system	$\{t = m\}$
$\boldsymbol{\delta_k^-} \leftarrow -\boldsymbol{y_k} - \boldsymbol{b_k^-}$	, error of the lower system	$\{t = m\}$
$\boldsymbol{\pi_k^+} \leftarrow \text{updatePi}(\boldsymbol{\delta_k^+})$	, update π^+ using Eq. 3.6	$\{t = 3m\}$
$\boldsymbol{\pi_k^-} \leftarrow \text{updatePi}(\boldsymbol{\delta_k^-})$	, update π^- using Eq. 3.6	$\{t = 3m\}$
if $\boldsymbol{\pi_k^+ = 0}$ and $\boldsymbol{\pi_k^- = 0}$ then	, check convergence	
$f_k \leftarrow true$	, block k feasible	
else		
$f_k \leftarrow false$	, block k not feasible	
$f \leftarrow \text{glblAnd}(f_k)$	, check the whole system	$\{t = (t_s + t_w) \log K\}$
if $f = true$ then		
exit		
$\boldsymbol{\pi_k} \leftarrow \boldsymbol{\pi_k^+} - \boldsymbol{\pi_k^-}$	, compute π_k	$\{t = m\}$
$\boldsymbol{q^k} \leftarrow (\boldsymbol{\pi_k}\ \boldsymbol{A_k})^T$	, multiply A from left	$\{t = 2z\}$
$\boldsymbol{q_k} \leftarrow \text{fold}(\boldsymbol{q^k})$	, fold q vector	$\{t = t_{fold}\}$
$\mu_k \leftarrow \boldsymbol{\delta_k^+ \pi_k^+} + \boldsymbol{\delta_k^- \pi_k^-}$	, sum of inner products	$\{t = 3m\}$
$\gamma_k \leftarrow (\boldsymbol{q_k})^T \boldsymbol{q_k}$	, inner product	$\{t = 2n\}$
$(\mu, \gamma) \leftarrow \text{glblSum}(\mu_k, \gamma_k)$	, form μ and γ	$\{t = (t_s + 2t_w) \log K\}$
$\boldsymbol{d_k} \leftarrow \mu/\gamma\ \ \boldsymbol{q_k}$	, form projection vector	$\{t = n\}$
$\boldsymbol{x_k} \leftarrow \boldsymbol{x_k} - \lambda\ \boldsymbol{d_k}$	, update x	$\{t = 2n\}$
endwhile		

Figure 4.1: 1D Basic surrogate constraint method

and the feasibility of the subsystems are checked. If all of the blocks are feasible, which means $\pi_k = 0$ for each block, then the method terminates. If this is not the case, then a new surrogate plane is generated with the current π_k vector. In order to update the kth block of the x-vector, processor P_k needs the kth block of the q vector which would result from the matrix-vector multiply πA. However, the result of the local matrix-vector multiply is the partial vector q^k. So, a fold operation is required in which P_k receives and adds partial vectors corresponding to the kth block. Finally, to obtain the global scalars μ and γ, one more communication operation is performed in which each processor receives and adds the local scalars. Then, each processor P_k updates the kth block of x_k and in the

next loop the feasibility of the system is checked again. This process repeats until a feasible solution is found.

Note that, in the algorithm, when we right multiply A_k with x, we map the problem from x-space to y-space. Then we perform linear operations on the y-space vectors and when we left multiply A_k with π_k, we again transform the problem from y-space to x-space. Hence there is no computational dependency between the the input and output vectors of the two multiplications.

4.2.2 Parallel Surrogate Constraint Method

The pseudocode of the improved parallel surrogate algorithm is given in Fig. 4.2 which is very similar to the parallel algorithm of BSCM. However, in this method, being different from BSCM, we form the local projection vectors d^k and then add up them to obtain the global combined projection vector d. Moreover, remember that in PSCM we apply a step sizing rule while generating the next point, which is actually a regulating scalar in the form of $\alpha/\beta = \sum_{k=1}^{K} \|d^k\|^2 / \|\sum_{k=1}^{K} d^k\|^2$. The first component of this ratio, α, is the sum of the inner products of the local projection vectors among all processors, and can be obtained using a global sum operation. After the fold operation each processor P_k ends up with d_k, the kth block of the combined vector d and can compute its local scalar, namely $\beta_k = \|d_k\|^2$. Since $\beta = \|d^2\| = \sum_{k=1}^{K} \|d_k^2\| = \beta_k^2$, the second component again can be obtained via a global sum operation among the processors.

4.3 Implementation Details

In order to implement these algorithms, first partition indicators and problem data should be distributed among the processors. Then, using these, each processor sets up the communication pattern and then pass to local indices for proper communication. Finally, local sparse matrix is assembled and then the surrogate methods are initiated. These steps are explained below.

while $true$ do		
$\boldsymbol{x} \leftarrow \text{expand}(\boldsymbol{x_k})$	, expand x_k vector	$\{t = t_{expand}\}$
$\boldsymbol{y_k} \leftarrow \boldsymbol{A_k x}$	, multiply A_k from right	$\{t = 2z\}$
$\boldsymbol{\delta_k^+} \leftarrow \boldsymbol{y_k} - \boldsymbol{b_k^+}$	, error of the upper system	$\{t = m\}$
$\boldsymbol{\delta_k^-} \leftarrow -\boldsymbol{y_k} - \boldsymbol{b_k^-}$	, error of the lower system	$\{t = m\}$
$\boldsymbol{\pi_k^+}$ updatePi$(\boldsymbol{\delta_k^+})$	, update π^+ using Eq. 3.6	$\{t = 3m\}$
$\boldsymbol{\pi_k^-} \leftarrow \text{updatePi}(\boldsymbol{\delta_k^-})$	, update π^- using Eq. 3.6	$\{t = 3m\}$
if $\boldsymbol{\pi_k^+ = 0}$ and $\boldsymbol{\pi_k^- = 0}$ then	, check convergence	
$f_k \leftarrow true$	, block k feasible	
else		
$f_k \leftarrow false$	, block k not feasible	
$f \leftarrow \text{glblAnd}(f_k)$	, check the whole system	$\{t = (t_s + t_w) \log K\}$
if $f = true$ then		
exit		
$\boldsymbol{\pi_k} \leftarrow \boldsymbol{\pi_k^+} - \boldsymbol{\pi_k^-}$	, compute π_k	$\{t = m\}$
$\boldsymbol{q^k} \leftarrow (\boldsymbol{\pi_k}\ \boldsymbol{A_k})^T$	, multiply A from left	$\{t = 2z\}$
$\mu_k \leftarrow \boldsymbol{\delta_k^+ \pi_k^+} + \boldsymbol{\delta_k^- \pi_k^-}$	, sum of inner products	$\{t = 3m\}$
$\gamma_k \leftarrow (\boldsymbol{q^k})^T \boldsymbol{q^k}$	, inner product	$\{t = 2N\}$
$\boldsymbol{d^k} \leftarrow \mu_k/\gamma_k\ \ \boldsymbol{q^k}$	, form local projection vector	$\{t = N\}$
$\alpha \leftarrow (\boldsymbol{d^k})^T \boldsymbol{d^k}$	, inner product	$\{t = 2N\}$
$\boldsymbol{d_k} \leftarrow \text{fold}(\boldsymbol{d^k})$	, fold d vector	$\{t = tfold\}$
$\beta \leftarrow \boldsymbol{d_k}^T \boldsymbol{d_k}$	, inner product	$\{t = 2n\}$
$(\alpha, \beta) \leftarrow \text{glblSum}(\alpha_k, \beta_k)$	, form α and β	$\{t = (t_s + 2t_w) \log K\}$
$\boldsymbol{x_k} \leftarrow \boldsymbol{x_k} - \lambda\alpha/\beta\ \boldsymbol{d_k}$	, update x	$\{t = 2n\}$

Figure 4.2: Improved parallel surrogate constraint method

4.3.1 Provide partition indicators and problem data to processors

Hypergraph decomposition methods provide us the partition indicators for the x- and y-space vectors. We also use the y vector partition indicator to determine the rows of the matrix that each processor owns. In our implementation a central processor reads these indicators and then broadcasts them to other processors. Then according to the partition data, central processor also reads and distributes the rows of the matrix and the right hand side vector components.

4.3.2 Setup communication

Consider the expand operation carried out before the local matrix-vector multiply $y_k = A_k x$. This operation is required to supply P_k the x-vector components that is not owned by P_k but for which it has a nonzero column. Remember that we employ a point-to-point local communication scheme in our 1D implementation, so each processor should know the processors and the corresponding vector components to be communicated. By examining the local sparse matrix A_k, processor P_k can determine from which processor it will receive which vector components, since each P_k has the partition indicator on the x-vector components. However, each processor P_k should also know to which processors it will send vector components and the corresponding indices. This is provided to processors via an all-to-all communication, in which each processor exchanges its data with the others and finally each P_k knows to which processor it will send which x-vector components that belongs to P_k.

To clarify the set up procedure, in Fig. 4.3, we give the steps followed to determine the communication pattern of expand operation for 1D decomposition. In this code $xSendCnts[p]$ denotes the number of x components that processor P_k has to send to processor p, and $xSendLists[p]$ is the pointer to the beginning of the corresponding component list. Similarly for $xRecvCnts$ and $xRecvLists$. This code is executed once in the beginning of the program and the communications

are performed using the data structures generated within this procedure.

---

for each nonzero $a_{ij} \in A_k$ **do**
  $p = xPart[j]$
  if j is not marked **and** $p \neq myId$ **then**
    mark j
    increase $xRecvCnts[p]$
endfor
send $xRecvCnts$, receive into $xSendCnts$ // All-to-all communication
for each column j **do**
  if j is marked **then**
    $p = xPart[j]$
    append j into xRecvLists[p]
endfor
send $xRecvLists$, receive into $xSendLists$ // All-to-all communication

---

Figure 4.3: Setting up communication for 1D decomposition

Consider the fold operation which is carried out after the local matrix-vector multiply $q^k = \pi_k A_k$. The output of this product is the partial results of q vector entries for all processors. Since a processor P_k is responsible of computing the the kth block of subvectors, it has to receive partial results of the corresponding vector components from other processors to end up with q_k. Actually, as clarified in [28, 10] these two communication operations are the duals of each other. That is, the processor numbers and the global vector indices sent/received are the same for fold and expand operations except that, the $xSendCnts$ and $xSendLists$ of the expand operation become, $xRecvCnts$ and $xRecvLists$ for the expand. To have a better understanding of this fact, suppose that x-space vector components that P_k accesses (updates or produces partial results) are divided into three parts:

Internal components ($xInt$) that P_k is responsible to compute, and for which no other processors produces partial results.

Border components ($xBorder$) that P_k is responsible to compute, and for which other processors also produces partial results. (Note that local components $x_k = xInt_k \cup xBorder_k$ for processor P_k.)

External components ($xExt$) that P_k is not responsible to compute, but produces partial results.

During a fold operation, other processors having nonzero entries which have the same column indices with $xBorder_k$ will compute and send partial results to P_k. Note that each processor sends the vector components corresponding to their external parts. Then, during the expand operation, the same processors will need the same indexed vector components that it has sent during the fold operation, to be able to perform the multiplication $y_k = A_k x$. So, now P_k will collect and send the vector components from $xBorder_k$ to the processors that it has received partial results in the fold operation.

4.3.3 Set Local Indices

It is customary to renumber x-vector components such that the entries belonging to the same processor have contiguous indices. This is necessary in order to send partial results to other processors without searching the indices at the beginning of the fold operation, and to make no searches after receiving the vector components in the expand operation. In 1D partitioning, we numbered the vector components according to the rank of the processors responsible on the components as in [27]. In this renumbering scheme, processor P_k gives labels to the external vector components belonging to some other processor p_j where $j < k$, then gives labels to the local vector components and then continue with labeling the external vector components belonging to some other processor p_j where $k < j$. Note that processor P_k can give labels to external vector components belonging to a processor p_j in any order; x_i can get label that is less than the label of x_j even processor p_j labels x_j before x_i. Since processors communicate global indices in Fig. 4.3 this does not cause any problem.

4.3.4 Assemble Local Sparse Matrix

To maximize communication and computation overlap, we explicitly split a processor's matrix into two sparse matrices *Aloc* and *Acpl* where *Aloc* contains all nonzeros a_{ij} where x_j is local to the processor and *Acpl* contains all nonzeros where x_j belongs to some other processor. Initially we store the matrix in coordinate format, but after distribution of the matrix and before entering to the main loop of the methods we pass to CSC format for both *Aloc* and *Acpl* (see appendix for the description of these storage schemes and corresponding matrix-vector multiplys). In this way P_k performs its local multiplication $y_k = Aloc\, x_k$ while sending its local components to other processors. Then, whenever P_k receives some of its external x components from a processor, it can continue multiplying with them before waiting all external x components to arrive.

4.4 Performance Analysis

The run time spent in the computational steps of the iterations is given on the right hand side of the parallel algorithms. In terms of the floating point operations per iteration, the parallel run times of BSCM and PSCM are,

$$T_{BSCM} = (4z + 12m + 5n)t_{flop} + T_{comm} \tag{4.3}$$

$$T_{PSCM} = (4z + 12m + 5N + 4n)t_{flop} + T_{comm} \tag{4.4}$$

where T_{comm} is the total time consumed for the communications. Assuming perfect load balance Eq. 4.3 and Eq. 4.4 can be simplified as,

$$\begin{aligned} T_{BSCM} &= \tfrac{T_s}{K} + T_{comm} \\ T_{PSCM} &= (4z + 12m + 5n + 5N - n)t_{flop} + T_{comm} \\ &= (\tfrac{T_s}{K} + 5N - n)t_{flop} + T_{comm} \end{aligned} \tag{4.5}$$

where T_s is the sequential run time of BSCM given in Eq. 3.7.

The overhead of the computation can be computed for both algorithms as:

$$T_{comm} = t_{expand} + t_{fold} + \log K(2t_s + 3t_w)$$

$$
\begin{aligned}
&= \psi_{avg}(t_s + \upsilon_{avg} t_w) + \psi_{avg}(t_s + \upsilon_{avg} t_w + \upsilon_{avg} t_{flop}) + \log K(2t_s + 3t_w) \\
&= t_s(2\psi_{avg} + 2\log K) + t_w(2\psi_{avg}\upsilon_{avg} + 3\log K) + t_{flop}(\psi_{avg}\upsilon_{avg})
\end{aligned}
$$

where ψ_{avg} is the average number of connectivity of all nets, (i.e. average number of messages), υ_{avg} is the average number of cut nets of each part (i.e. average message length), and t_{flop} is the time required for one floating point operation. Since total size of the messages ($\psi_{avg}\upsilon_{avg}$) is much larger than $\log K$, this can be approximated as:

$$
T_{comm} \approx 2t_s(\psi_{avg} + \log K) + 2t_w\psi_{avg}\upsilon_{avg} + t_{flop}\psi_{avg}\upsilon_{avg}. \tag{4.6}
$$

Speed up is a measure of the relative benefit gained by parallelizing a sequential algorithm which is defined by the ratio of the serial run time of the best sequential algorithm to the parallel run time. As clarified in Section 3.4, BSCM is the best sequential algorithm in our problem. So, the speed up S of the parallel algorithms become:

$$
S_{BSCM} = \frac{T_s}{T_{BSCM}} = \frac{T_s}{T_s/K + T_{comm}} = \frac{K}{1 + K\frac{T_{comm}}{T_s}} \tag{4.7}
$$

$$
S_{PSCM} = \frac{T_s}{T_{PSCM}} = \frac{T_s}{T_s/K + 5N - n + T_{comm}} = \frac{K}{1 + K\frac{5N-n+T_{comm}}{T_s}} \tag{4.8}
$$

Another performance metric for the parallel systems is the efficiency, which is defined as the ratio of the speed up to the number of processors. This can be calculated for BSCM and PSCM as follows:

$$
E_{BSCM} = \frac{S}{K} = \frac{1}{1 + K\frac{T_{comm}}{T_s}} \tag{4.9}
$$

$$
E_{PSCM} = \frac{S}{K} = \frac{1}{1 + K\frac{5N-n+T_{comm}}{T_s}} \tag{4.10}
$$

We see from Eq. 4.9 that as long as $T_s = \Theta(KT_{comm})$ and from Eq. 4.10 that as long as $T_s = \Theta(KT_{comm} + 5NK)$ our parallel implementations are cost optimal.

Chapter 5

Parallellization based on 2D Partitioning

In this chapter we will concentrate on parallel implementations of the surrogate constraint methods for 2D decomposition of the system $Ax \leq b$. First, *checkerboard partitioning* will be used in which the underlying system is decomposed rowwise and columnwise into smaller blocks. The matrix will be distributed among the processors using a two-phase decomposition method which is based on multiconstraint hypergraph partitioning proposed by Çatalyürek and Aykanat [4]. We employed two communication schemes for the required fold and expand communication operations. The first one is a point-to-point communication scheme which uses the *connectivity* metric as the hypergraph cut size. With this scheme we exploit the sparsity of the matrix, hence no redundancy occurs in the communication. The second communication scheme uses *cutnet* metric as the hypergraph cut size so that the partitioning method tries to minimize the external parts of the processor blocks which necessitate communication. A global all-to-all communication is performed in the external parts so that each processor ends up with the whole external vector.

Secondly, a *fine-grain* decomposition strategy is employed which distributes the matrix among the processors on a nonzero basis. We used the fine-grain

hypergraph model again proposed by Çatalyürek and Aykanat [3]. However, this method is only used with BSCM, since PSCM is based on the projection of the row blocks, which cannot be easily carried out with a fine-grain partitioning of the system.

5.1 Hypergraph Partitioning based 2D Decomposition

In this section we will briefly review hypergraph models proposed for the checkerboard partitioning [4] and fine-grain partitioning [3] of sparse matrices.

5.1.1 Checkerboard Partitioning

In checkerboard partitioning the matrix is divided rowwise into r contiguous row blocks, then each row block is again divided columnwise into c blocks, and this partitioning naturally maps into an $r \times c$ mesh of processors. In a matrix-vector multiply, this partitioning scheme incurs expand operation along the columns of the processor mesh as well as the fold operation along the rows of the processor mesh. Expand operation is carried out in each mesh column concurrently, and the fold operation is carried out in each mesh row concurrently. In their work Çatalyürek and Aykanat [4] concentrated on the parallelization of the sparse matrix-vector multiply for the checkerboard partitioning. Previous checkerboard partitioning schemes proposed only aim at load balancing and are suitable for dense matrices or sparse matrices with a structured sparsity pattern [12, 20, 21]. The proposed model of Çatalyürek and Aykanat on the other hand, exploits the sparsity for reducing communication volume and can handle decomposition of any sparse matrix.

The proposed model is a two-phase method. In the first phase, using existing hypergraph-partitioning model, matrix A is decomposed r-way into contiguous rows. So, total volume of expand operation is minimized. In the second phase,

for columnwise partitioning of the matrix, a c-way multi-constraint hypergraph-partitioning is performed using the row-net representation of the matrix. Multi-constraint hypergraph-partitioning is necessary in order to balance the computational loads that will be executed concurrently.

In the partitioning process for checkerboard decomposition of the matrix, we have used both of the cut size definitions given in Eq. 4.2. First, as in 1D case for point-to-point communication scheme, connectivity metric is used. Since for the second communication scheme an all-to-all type operation is carried out in the external parts, the communication time is proportional to the size of these external parts. So, for the second communication scheme, we used the cutnet metric given in Eq. 4.2.a, the number of cut nets and hence the size of the external parts are minimized.

5.1.2 Fine-grain Partitioning

For the fine-grain decomposition of the matrix, in another work, Çatalyürek and Aykanat proposed the fine-grain hypergraph model [3]. In this model each nonzero element is represented by a vertex and models scalar multiplication operation. Nets of the hypergraph represent the dependency relations of the scalar multiplication operations on the x- and y-vector entries. According to the connectivity cutsize metric, the proposed model reduces 2D fine-grain matrix decomposition problem into the K-way hypergraph partitioning problem.

5.2 Parallel Algorithms

5.2.1 Parallelization of the Basic Method

For checkerboard partitioning scheme with a parallel system of $K = r \times c$ processors, matrix A is decomposed into $r \times c$ blocks and each block is assigned to a processor. If we label the blocks in a row-major order, the distribution looks

like,

$$\mathbf{A} = \begin{array}{|c|c|c|c|c|} \hline P_{11} & \cdots & P_{1l} & \cdots & P_{1c} \\ \hline \vdots & \ddots & \vdots & \ddots & \vdots \\ \hline P_{k1} & \cdots & P_{kl} & \cdots & P_{kc} \\ \hline \vdots & \ddots & \vdots & \ddots & \vdots \\ \hline P_{r1} & \cdots & P_{rl} & \cdots & P_{rc} \\ \hline \end{array}.$$

The y-space vectors are partitioned first into r slices each assigned to a row block, and each slice is again partitioned into c slices, as follows,

$$\mathbf{y} = \begin{pmatrix} y_1 \\ \vdots \\ y_k \\ \vdots \\ y_r \end{pmatrix} \text{and } \mathbf{y}_k = \begin{pmatrix} y_{k1} & \cdots & y_{kl} & \cdots & y_{kc} \end{pmatrix}.$$

So, a y-space vector denoted as y_{kl} corresponds to the kth row and lth column, hence belong to processor P_{kl}.

The x-space vectors, on the other hand, partitioned first into c slices, and then each slice is partitioned again into r slices,

$$\mathbf{x} = \begin{pmatrix} x_1 & \cdots & x_l & \cdots & x_c \end{pmatrix} \text{and } \mathbf{x}_l = \begin{pmatrix} x_{1l} \\ \vdots \\ x_{kl} \\ \vdots \\ x_{rl} \end{pmatrix}.$$

So again an x-space vector x_{kl} belongs to processor P_{kl}.

In Fig. 5.1 we see the parallel pseudocode of BSCM. Each processor P_{kl} accesses the following components in the algorithm:

- an $m \times n$ matrix $\boldsymbol{A_{kl}}$ containing z non-zero entries, where $m \approx M/r$, $n \approx N/c$, and $z \approx Z/K$.
- x-space vectors:

- $n \times 1$ global vector $\boldsymbol{x_l}$ which belongs to the lth column of processor mesh.
- $n \times 1$ local vector $\boldsymbol{x_{kl}}$ where $n \approx N/K$.
- $n \times 1$ column vector $\boldsymbol{q_l^k}$ which is the partial vector resulting from the local matrix-vector multiply $\pi_k A_{kl}$.
- $n_k \times 1$ column vector $\boldsymbol{q_{kl}}$ which is the kth block of vector q_l that would result from the global matrix-vector multiply πA_{*l} where A_{*l} is the lth column block of A.
- $n_k \times 1$ local projection vector $\boldsymbol{d_{kl}}$.

- y-space vectors:
  - $m_k \times 1$ local vectors $\boldsymbol{b_{kl}^+}$ and $\boldsymbol{b_{kl}^-}$ where $m_k \approx M/K$.
  - $1 \times m_k$ local vectors $\boldsymbol{\pi_{kl}}$, $\boldsymbol{\pi_{kl}^+}$, and $\boldsymbol{\pi_{kl}^-}$.
  - $1 \times m$ global vector of a row block $\boldsymbol{\pi_k^-}$.
  - $m_k \times 1$ local vectors $\boldsymbol{\delta_{kl}^+}$ and $\boldsymbol{\delta_{kl}^-}$ which denotes the error in the upper and lower systems, respectively.

- scalars:
  - Global scalars μ and γ.
  - Corresponding local scalars μ_{kl} and γ_{kl}.

With the parallelization scheme described, each processor P_{kl} is responsible for computing the corresponding portion x_{kl} of the x-vector. Similar to the 1D case, in order to be able to perform the local matrix-vector multiply $A_{kl}x_l$, an expand operation is required along the mesh column so that each processor P_{kl} initially holding x_{kl} ends up with x_l. After the local matrix-vector multiply, processor P_{kl} has partial y vectors y_k^l. Since we have redistributed the kth block of y vectors y_k into c subvectors, each row of the processor mesh steps into rowwise fold operation so that each processor P_{kl} end up with the appropriate portion of the y_k vector, namely y_{kl}. Then, processors can find their local π_{kl} values and

---

while $true$ do		
$\boldsymbol{x_l} \leftarrow \text{colExpand}(\boldsymbol{x_{kl}})$	, expand x_{kl} in column mesh	$\{t = t_{colexpand}\}$
$\boldsymbol{y_k^l} \leftarrow \boldsymbol{A_{kl}} \boldsymbol{x_l}$	, multiply A_{kl} from right	$\{t = 2z\}$
$\boldsymbol{y_{kl}} \leftarrow \text{rowFold}(\boldsymbol{y_k^l})$	, fold y_k^l along the row mesh	$\{t = t_{rowfold}\}$
$\boldsymbol{\delta_{kl}^+} \leftarrow \boldsymbol{y_{kl}} - \boldsymbol{b_k^+}$	, error of the upper system	$\{t = m_k\}$
$\boldsymbol{\delta_{kl}^-} \leftarrow - \boldsymbol{y_{kl}} - \boldsymbol{b_k^-}$	, error of the lower system	$\{t = m_k\}$
$\boldsymbol{\pi_{kl}^+} \leftarrow \text{updatePi}(\boldsymbol{\delta_{kl}^+})$	, update π^+ using Eq. 3.6	$\{t = 3m_k\}$
$\boldsymbol{\pi_{kl}^-} \leftarrow \text{updatePi}(\boldsymbol{\delta_{kl}^-})$	, update π^- using Eq. 3.6	$\{t = 3m_k\}$
if $\boldsymbol{\pi_{kl}^+ = 0}$ and $\boldsymbol{\pi_{kl}^- = 0}$ then	, check convergence	
$f_k \leftarrow true$	, block k feasible	
else		
$f_k \leftarrow false$	, block k not feasible	
$f \leftarrow \text{glblAnd}(f_k)$	, check the whole system	$\{t = (t_s + t_w) \log K\}$
if $f = true$ then		
exit		
$\boldsymbol{\pi_{kl}} \leftarrow \boldsymbol{\pi_{kl}^+} - \boldsymbol{\pi_{kl}^-}$	, compute π_{kl}	$\{t = m_k\}$
$\boldsymbol{\pi_k} \leftarrow \text{rowExpand}(\boldsymbol{\pi_{kl}})$	, expand π_{kl} in row mesh	$\{t = t_{rowexpand}\}$
$\boldsymbol{q_l^k} \leftarrow \boldsymbol{\pi_k} \boldsymbol{A_{kl}}$	, multiply A from left	$\{t = 2z\}$
$\boldsymbol{q_{lk}} \leftarrow \text{colFold}(\boldsymbol{q_l^k})$	, fold q_l^k in column mesh	$\{t = t_{colfold}\}$
$\mu_{kl} \leftarrow \boldsymbol{\pi_{kl}^+ \delta_{kl}^+} + \boldsymbol{\pi_{kl}^- \delta_{kl}^-}$	, sum of inner products	$\{t = 3m_k\}$
$\gamma_{kl} \leftarrow (\boldsymbol{q_{lk}})^T \; \boldsymbol{q_{lk}}$	, inner product	$\{t = 2n_k\}$
$(\mu, \gamma) \leftarrow \text{glblSum}(\mu_{kl}, \gamma_{kl})$	, form μ and γ	$\{t = (t_s + 2t_w) \log K\}$
$\boldsymbol{d_k} \leftarrow \mu/\gamma \;\; \boldsymbol{q_{kl}}$	, form projection vector	$\{t = n_k\}$
$\boldsymbol{x_{kl}} \leftarrow \boldsymbol{x_{kl}} - \lambda \; \boldsymbol{d_{kl}}$	, update x	$\{t = 2n_k\}$
endfor		

Figure 5.1: 2D Basic surrogate constraint method

the feasibility check is performed. However, in order P_{kl} to perform local matrix-vector multiply $\pi_k A_{kl}$, it requires all π_k components belonging to the kth row of processor mesh. So, again an expand operation is performed, but this time along the row-mesh of processors. After the local matrix-vector multiply $\pi_k A_{kl}$, each processor has the vector q_l^k; the lth portion of the q vector due to the kth block, hence $q_l = \sum_{k=1}^{r} q_l^k$. So, a columnwise fold operation is carried out which leaves the appropriate portion q_{kl} of q_l on processor P_{kl}. Since, each processor has a subblock of the π, δ, and q vectors, global scalars μ and γ can be obtained via a global sum operation among all processors of the mesh. With these components at hand, processors can now compute the projection vector and then update the current value of x. In the next cycle, once more the feasibility check is made with this value and this process repeats until a feasible solution is found.

The pseudocodes of the implementations with the point-to-point and all-to-all communication schemes are the same; only the fold and expand operations are carried out in different ways. In the point-to-point communication scheme, each processor knows the processor numbers and the vector components to communicate, and performs the communication in a local manner with point-to-point send and receive type communication operations. In all-to-all communication scheme, each processor carries out a global reduce-scatter type operation for the fold operation and an all-to-all broadcast operation for the expand operation. In Section 5.3 we give the details.

With the fine-grain partitioning of the matrix A, we do not have a definite processor organization as in the checkerboard partitioning. Nonzero elements of A are distributed among the processors with no constraints. So, a single row or a single column of the matrix can be shared among all K processors. The pseudocode of the fine-grain implementation is almost identical to the algorithm shown in Fig. 5.1. However, fold and expand operations are carried out among all K processors whether they are performed rowwise or columnwise.

5.2.2 Parallel Surrogate Constraints Method

The 2D implementation of PSCM is similar to the 2D implementation of BSCM. However, in order to calculate the projection vector of the kth block, μ_{kl} and γ_{kl} is summed in the kth row mesh. Moreover, since $\alpha = \sum_{k=1}^{r} \sum_{l=1}^{c} \alpha_{kl}$ and $\beta = \sum_{k=1}^{r} \sum_{l=1}^{c} \beta_{kl}$, a global sum operation is required to apply the step sizing rule in the calculation of the next point.

We have performed 2D implementation of PSCM for point-to-point and all-to-all communication schemes. As mentioned, PSCM is composed of block projections, hence we require a row-block of the matrix to be distributed among a set of processors. Since, in fine-grain partitioning block structure definition is not preserved, it is not suitable for PSCM.

5.3 Implementation Details

In this section we will give the preprocessing steps that should be followed before the surrogate methods in order to maximize the parallel performance of the system. For point-to-point communication scheme which is used both in checkerboard partitioning and fine-grain partitioning the steps are similar to 1D case. For all-to-all communication scheme, we will first introduce the combining algorithm proposed by Jacunski et al. [14]. Then the preprocessing steps will be detailed.

5.3.1 Point-to-point Communication Scheme

For point-to-point communication scheme, similar to the 1D case, the following initialization steps should be performed before the main loop of the surrogate methods.

while $true$ do		
$\boldsymbol{x_l} \leftarrow$ colExpand($\boldsymbol{x_{kl}}$)	, expand x_{kl} in column mesh	$\{t = t_{colexpand}\}$
$\boldsymbol{y_k^l} \leftarrow \boldsymbol{A_{kl}x_l}$	, multiply A_{kl} from right	$\{t = 2z\}$
$\boldsymbol{y_{kl}} \leftarrow$ rowFold($\boldsymbol{y_k^l}$)	, fold y_k^l along the row mesh	$\{t = t_{rowfold}\}$
$\boldsymbol{\delta_{kl}^+} \leftarrow \boldsymbol{y_{kl}} - \boldsymbol{b_k^+}$	, error of the upper system	$\{t = m_k\}$
$\boldsymbol{\delta_{kl}^-} \leftarrow - \boldsymbol{y_{kl}} - \boldsymbol{b_k^-}$	, error of the lower system	$\{t = m_k\}$
$\boldsymbol{\pi_{kl}^+} \leftarrow$ updatePi($\boldsymbol{\delta_{kl}^+}$)	, update π^+ using Eq. 3.6	$\{t = 3m_k\}$
$\boldsymbol{\pi_{kl}^-} \leftarrow$ updatePi($\boldsymbol{\delta_{kl}^-}$)	, update π^- using Eq. 3.6	$\{t = 3m_k\}$
if $\boldsymbol{\pi_{kl}^+ = 0}$ and $\boldsymbol{\pi_{kl}^- = 0}$ then	, check convergence	
$f_k \leftarrow true$	, block k feasible	
else		
$f_k \leftarrow false$	, block k not feasible	
$f \leftarrow$ glblAnd(f_k)	, check the whole system	$\{t = (t_s + t_w) \log K\}$
if $f = true$ then		
exit		
$\boldsymbol{\pi_{kl}} \leftarrow \boldsymbol{\pi_{kl}^+ - \pi_{kl}^-}$	, compute π_{kl}	$\{t = m_k\}$
$\boldsymbol{\pi_k} \leftarrow$ rowExpand($\boldsymbol{\pi_{kl}}$)	, expand π_{kl} in row mesh	$\{t = t_{rowexpand}\}$
$\boldsymbol{q_l^k} \leftarrow \boldsymbol{\pi_k A_{kl}}$	, multiply A from left	$\{t = 2z\}$
$\mu_{kl} \leftarrow \boldsymbol{\pi_{kl}^+ \delta_{kl}^+ + \pi_{kl}^- \delta_{kl}^-}$	, sum of inner products	$\{t = 3m_k\}$
$\gamma_{kl} \leftarrow (\boldsymbol{q_l^k})^T \boldsymbol{q_l^k}$	, inner product	$\{t = 2n\}$
$(\mu_k, \gamma_k) \leftarrow$ rowSum(μ_{kl}, γ_{kl})	, form μ_k and γ_k	$\{t = (t_s + 2t_w) \log c\}$
$\boldsymbol{d_l^k} \leftarrow \mu_k/\gamma_k \ \ \boldsymbol{q_l^k}$	, form projection vector	$\{t = n\}$
$\alpha_{kl} \leftarrow (\boldsymbol{d_l^k})^T \boldsymbol{d_l^k}$	, inner product	$\{t = 2n\}$
$\boldsymbol{d_{kl}} \leftarrow$ colFold($\boldsymbol{d_l^k}$)	, fold d_l^k in column mesh	$\{t = t_{colfold}\}$
$\beta_{kl} \leftarrow \boldsymbol{d_{kl}}^T \boldsymbol{d_{kl}}$	, inner product	$\{t = 2n_k\}$
$(\alpha, \beta) \leftarrow$ glblSum(α_{kl}, β_{kl})	, form α and β	$\{t = (t_s + 2t_w) \log K\}$
$\boldsymbol{x_{kl}} \leftarrow \boldsymbol{x_{kl}} - \lambda\alpha/\beta \ \boldsymbol{d_{kl}}$	, update x	$\{t = 2n_k\}$ endfor

Figure 5.2: 2D Parallel surrogate constraint method

5.3.1.1 Provide partition indicators and problem data to processors

In 2D partitioning, we have the partition indicator of the nonzero elements of matrix A, as well as the x and y-space vectors' partition indicators. We need the partition indicator of the nonzero elements since the indicators of the x and y-space vectors tell just the responsibility for computing the components. The central processor reads this nonzero partition indicator and scatters the nonzero elements of the matrix. Then it also distributes the RHS vector values. Finally, it also reads the partition indicators of x and y-space vectors and broadcasts them to all processors in the parallel system.

5.3.1.2 Setup communication

Similar to the 1D case, each processor should know the processor numbers and the vector components that will be communicated in the fold and expand operations. We give the pseudocode which is used to prepare the data structures used in the expand operation in Fig. 5.3. Note that all-to-all communications are performed along the rows and columns of the processor mesh in 2D case. The fold operation is performed with the same data structures with an exchange of the meanings of the *recvCnts* with the *sendCnts* and *recvLists* with the *sendLists*.

5.3.1.3 Set Local Indices

In order to make in-place receives in expand operation and in-place sends in fold operation, the vector components to be received and the vector components to be sent should be numbered contiguously for each processor in the system. For this purpose, we numbered the indices of the local matrices beginning from the local indices, and then continuing with the external components according to the processor ranks.

Note that in point-to-point communication scheme, the components to be sent should be collected from the border vector components in the expand operation,

```
for each nonzero a_ij ∈ A_kl do
    p = xPart[j]
    if j is not marked and p ≠ myId then
        mark j
        rowId = row number of p in proc mesh
        increase xRecvCnts[rowId]
    endif
    p = yPart[i]
    if j is not marked and p ≠ myId then
        mark i
        colId = col number of p in proc mesh
        increase xRecvCnts[colId]
    endif
send xRecvCnts, receive into xSendCnts // All-to-all communication along column mesh
send yRecvCnts, receive into ySendCnts // All-to-all communication along row mesh
for each column j do
    if j is marked then
        p = xPart[j]
        rowId = row number of p in proc mesh
        append rowId into xRecvLists[p]
    endif
endfor
for each row i do
    if i is marked then
        p = xPart[i]
        colId = col number of p in proc mesh
        append colId into yRecvLists[p]
    endif
endfor
send xRecvLists, receive into xSendLists // All-to-all communication along column mesh
send yRecvLists, receive into ySendLists // All-to-all communication along row mesh
```

Figure 5.3: Setting up communication in 2D decomposition for the point-to-point communication scheme

and for the fold operation, after receiving the partial results, processors should determine the indices to be added and then perform the accumulation of the border components. So, we cannot make in-place communications for the sends in expand operation and for the receives in fold operation.

5.3.1.4 Assemble Local Sparse Matrix

In 2D case, we store the local sparse matrix both in CSR and CSC formats and overlap local matrix-vector multiply operation with the expand communication. While performing right multiplication of the matrix A with the x-vector, first we prepare send data and then initiate communications. Then A is multiplied with x_k; the local components of the x-vector and a partial result is obtained. Afterwards, whenever external components of the x-vector arrive they are multiplied with the corresponding matrix components and results are added up to the partial result. In order to make immediate access to the row indices that has nonzero elements in a given column, the storage scheme should be CCS. For the left multiplication operation on the other hand, in order processor P_k to continue multiplying before waiting all external components of the π_k vector, it should be able to make immediate access to the column indices. Hence, the storage scheme CSR is used for this operation.

5.3.2 All-to-all Communication Scheme

We performed the expand operation using an efficient all-to-all broadcast operation based on the combining algorithm developed by Jacunski et al. [14]. In the next section we will describe this algorithm, and then give the details of the implementation for all-to-all communication scheme.

5.3.2.1 Combining Algorithm

As mentioned, in all-to-all communication scheme, at the end of the expand operation all processors ends up with the concatenation of the external vectors. This type of communication is known as *all-to-all broadcast* in the parallel community [17], and as the *allgather* operation in the MPI community [16]. Using a hypercube algorithm it can be performed in optimum time of $t_{comm} = t_s \log_2 p + mt_w$ [17]. However, this algorithm is applicable in the switch-based clusters only when $K = 2^d$, where K is the number of processors and d is the dimension of the hypercube. Later, Jacunski et. al. made an extension to the hypercube algorithm which allows hypercube algorithm to be used even if K is not a power of 2. Proposed *combining algorithm* makes $\lceil \log_2 K \rceil$ steps during the communication operation [14] and in each step, the local message is combined with the receiving one, and in the next step this combined message is sent. So, in each step except the last one, the size of the message sent and received is doubled. In the last step, each processor sends a portion of the local message. In Fig. 5.4 we give the pseudocode of the algorithm, and in Fig. 5.5, we illustrate the algorithm for $K = 6$.

Fold operation can be performed with a similar algorithm by reversing the communication steps of the combining algorithm. However, after each step, instead of concatenating the incoming vectors, processors should add them up with the appropriate vector components. Fig. 5.6 illustrates this procedure for $K = 6$.

5.3.2.2 Setup communication

During the fold and expand operations, the vector components corresponding to cut nets will be communicated. These are the border components that has been defined in Section 4.3.2. In order to find the border components, processors first obtain the arrays *sendLists* and *recvLists* as in the point-to-point case. Then each processor examines its local vector components and the *sendLists* array in order to find the number of cut nets. Then with an allgather operation each processor obtains the number of cuts of all of the processors.

$d = \lfloor \log_2 K \rfloor$
$result = myMsg$
for $step = 0$ **to** d **do**
  $dest = (myId + 2^{step}) \bmod K$
  $srce = (myId - 2^{step}) \bmod K$
  send $result$ to $dest$
  receive msg from $srce$
  $result = result \cup msg$
endfor
if $\lceil \log_2 K \rceil > d$ **then**
  $dest = (myId + 2^d) \bmod K$
  $srce = (myId - 2^d) \bmod K$
  send first $K - 2^d$ msgs to $dest$
  receive msg from $srce$
  $result = result \cup msg$
endif

Figure 5.4: Combining algorithm

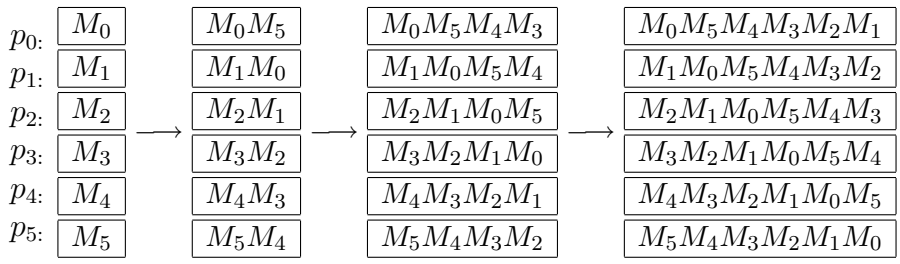


Figure 5.5: Expand operation based on the combining algorithm

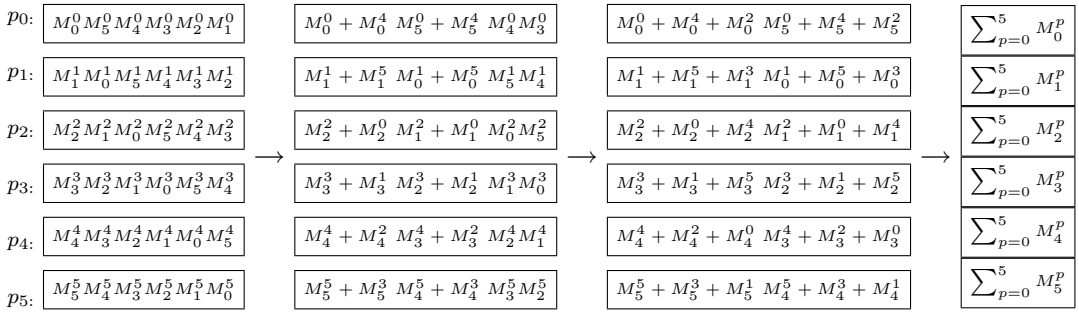


Figure 5.6: Fold operation based on the combining algorithm

5.3.2.3 Set Local Indices

Since the internal vector components of the processors can be computed independently, each processor in the system begins to number its local elements beginning from the internal indices. Then they label the border components and then continue with the vector components of other processors following the same pattern of the fold and expand operations. That is they renumber remaining indices in the order $myId-1, myId-2, \ldots, 1, 0, K-1, K-2, \ldots, myId+1$. In this way we can make in-place communication during both send and receives of the combining algorithm. This is an advantage of the all-to-all communication scheme on the point-to-point communication scheme, for which we should search and collect the components to be sent during the expand operation.

5.3.2.4 Assemble Local Sparse Matrix

In 2D implementations, we hold the matrix A in CSC format for the right multiplication, and in CSR format for the left multiplication in order to overlap the communication time of the expand operation with the computation time of matrix-vector multiplies. However, contrary to the point-to-point case, there is a strict order in the expand operation such that each processor should wait for the arrival of the vector components in order to send them in the next step. Moreover, during the fold operation of the point-to-point communication scheme, we can add

up the incoming vector components while waiting for the arrival of others. This is again not possible in the combining algorithm due to the strict order that should be followed. So we can deduce that we have a better chance of communication-computation overlap in the point-to-point communication scheme.

5.4 Performance Analysis

5.4.1 Point-to-point Communication

The parallel run times of the algorithms for point-to-point communication of 2D surrogate methods are as follows:

$$T_{BSCM} = (4z + 12m_k + 5n_k)t_{flop} + T_{comm} = \frac{T_s}{K} + T_{comm} \tag{5.1}$$

$$\begin{aligned} T_{PSCM} &= (4z + 12m_k + 5n + 4n_k)t_{flop} + T_{comm} \\ &= (4z + 12m_k + 5n_k + 5n - n_k)t_{flop} + T_{comm} \\ &= \tfrac{T_s}{K} + (5n - n_k)t_{flop} + T_{comm} \end{aligned} \tag{5.2}$$

where T_s is the sequential run time of BSCM, and T_{comm} is the total time consumed for the communications. The time consumed by the expand and fold operations are:

$$\begin{aligned} t_{colexpand} &= \psi_{col_{avg}}(t_s + \upsilon_{col_{avg}} t_w) \\ t_{rowexpand} &= \psi_{row_{avg}}(t_s + \upsilon_{row_{avg}} t_w) \\ t_{rowfold} &= \psi_{row_{avg}}(t_s + \upsilon_{row_{avg}} t_w + \upsilon_{row_{avg}} t_{flop}) \\ t_{colfold} &= \psi_{col_{avg}}(t_s + \upsilon_{col_{avg}} t_w + \upsilon_{col_{avg}} t_{flop}), \end{aligned}$$

where $\psi_{col_{avg}}$ is the average number of messages in a column mesh, $\upsilon_{col_{avg}}$ is the average message length in a column mesh, $\psi_{row_{avg}}$ is the average number of messages in a row mesh, $\upsilon_{row_{avg}}$ is the average message length in a row mesh and t_{flop} is the time required for one floating point operation. So, total time of communication spent in BSCM is:

$$T_{comm} = t_{colexpand} + t_{rowexpand} + t_{rowfold} + t_{colfold} + (2t_s + 3t_w)\log K \tag{5.3}$$

and for PSCM,

$$T_{comm} = t_{colexpand} + t_{rowexpand} + t_{rowfold} + t_{colfold} + (2t_s + 3t_w)\log K + (t_s + 2t_w)\log c \tag{5.4}$$

The speed up S of the parallel algorithms are:

$$S_{BSCM} = \frac{T_s}{T_{BSCM}} = \frac{T_s}{T_s/K + T_{comm}} = \frac{K}{1 + K\frac{T_{comm}}{T_s}} \tag{5.5}$$

$$S_{PSCM} = \frac{T_s}{T_{PSCM}} = \frac{T_s}{T_s/K + 5n - n_k + T_{comm}} = \frac{K}{1 + K\frac{5n - n_k + T_{comm}}{T_s}} \tag{5.6}$$

The efficiency of the programs can be calculated for BSCM and PSCM as follows:

$$E_{BSCM} = \frac{S}{K} = \frac{1}{1 + K\frac{T_{comm}}{T_s}} \tag{5.7}$$

$$E_{PSCM} = \frac{S}{K} = \frac{1}{1 + K\frac{5n - n_k + T_{comm}}{T_s}} \tag{5.8}$$

We see from Eq. 5.7 that as long as $T_s = \Theta(KT_{comm})$ 2D parallel implementation of BSCM is cost optimal. Since $nK \approx N/rK = Nc$, from Eq. 5.8 as long as $T_s = \Theta(KT_{comm} + 5Nc)$ 2D implementation of PSCM is also cost optimal.

5.4.2 All-to-all Communication

The computational time expressions of the implementations based on all-to-all communication schemes are the same as that of point-to-point scheme. The communication time of the fold and expand operations can be computed as follows: Let n_ε be the average number of cut nets reside in a column mesh and let each processor in the column mesh owns approximately n_ε/r of the cut nets. Assume that r is a power of 2 so it takes $\log r$ steps for the algorithm to finish. Since the message size is doubled in each phase of the combining algorithm, the communication time of an expand operation along a column mesh is:

$$T_{colexpand} = \sum_{k=1}^{\log r} ts + 2^{k-1}(n_\varepsilon/r)t_w = t_s \log r + t_w n_\varepsilon \frac{r-1}{r}. \tag{5.9}$$

In a fold operation, except the local part of the vector, each communicated message is summed. So the expression for the fold time is:

$$T_{colfold} = t_s \log r + t_w n_\varepsilon \frac{r-1}{r} + t_{flop} n_\varepsilon \frac{r-1}{r}. \tag{5.10}$$

Similarly the communication time of the operations in a row mesh are as follows:

$$T_{rowexpand} = \sum_{k=1}^{\log c} ts + 2^{k-1}(m_\varepsilon/c)t_w = t_s \log c + t_w m_\varepsilon \frac{c-1}{c}. \tag{5.11}$$

$$T_{rowfold} = t_s \log c + t_w m_\varepsilon \frac{c-1}{c} + t_{flop} m_\varepsilon \frac{c-1}{c} \tag{5.12}$$

where m_ε is the average number of cut nets reside in a row mesh.

In the next chapter, the experimental results and the evaluations of the implementations will be presented.

Chapter 6

Results

In this section we will evaluate the parallel performance and the restoration performance of the surrogate constraint methods. Parallel performance analysis will be first on iteration basis. This way, we will have a chance to see how much performance gain is achieved by the parallelization schemes. Then, we will consider overall performance results. Finally, we will give the examples of blurred and restored images. However, the data sets used in the experiments need to be explained first.

6.1 Data Sets

In our experiments we have used three types of blurs for the construction of the distortion system. First type of blur models isotropic scaling. This motion is represented by the function $\rho(r,t) = r/m(t)$ where $m(t)$ is an arbitrary scaling function of time and r is the position vector. We have chosen the components $u(x,y,t)$ and $v(x,y,t)$ of ρ as:

$$\begin{aligned} u(x,t) &= \frac{x}{(0.1t^2)} \\ v(y,t) &= \frac{y}{(0.1t^2)} \end{aligned}$$

and set the record time as 5 seconds.

The second blur is a result of rotation motion. We set the record time 8 seconds and let the image to rotate clockwise 6 degrees per second in the first 5 seconds and then rotate counterclockwise with the same angular speed in the remaining 3 seconds.

Third blur denotes a combined effect of translational motion, isotropic scaling, and rotation. For the first 5 seconds the image rotates clockwise 8 degrees per second and then for three seconds it rotates counterclockwise with the same angular speed. Meanwhile the image undergoes the same isotropic scaling effect of the first type blur and makes the translational motion shown in Fig. 6.1.

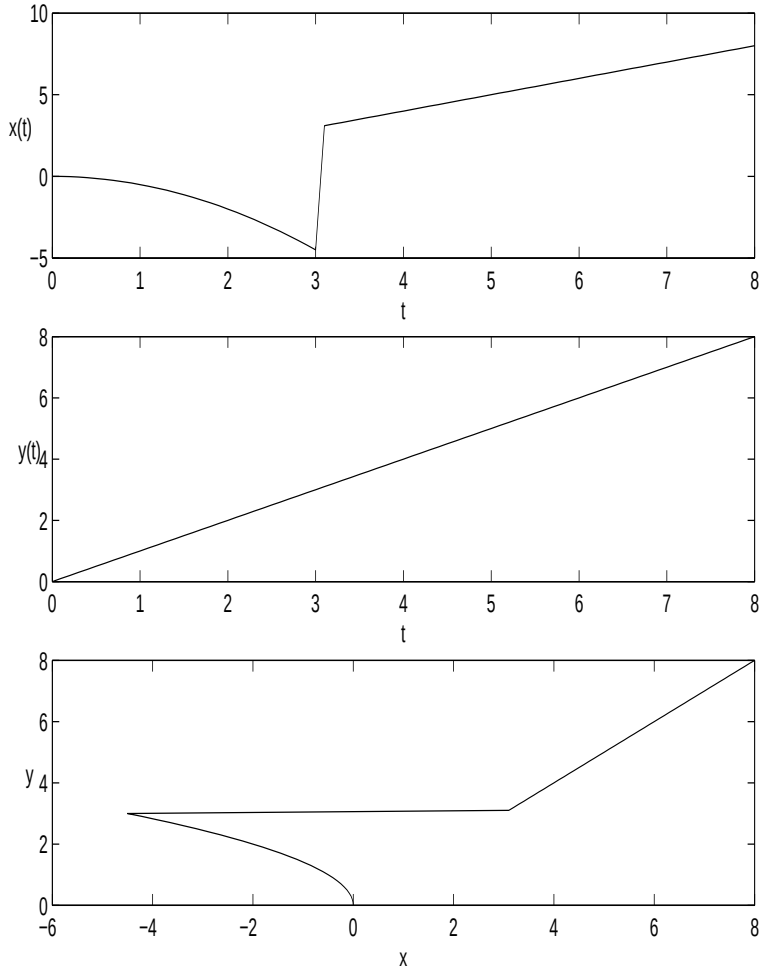
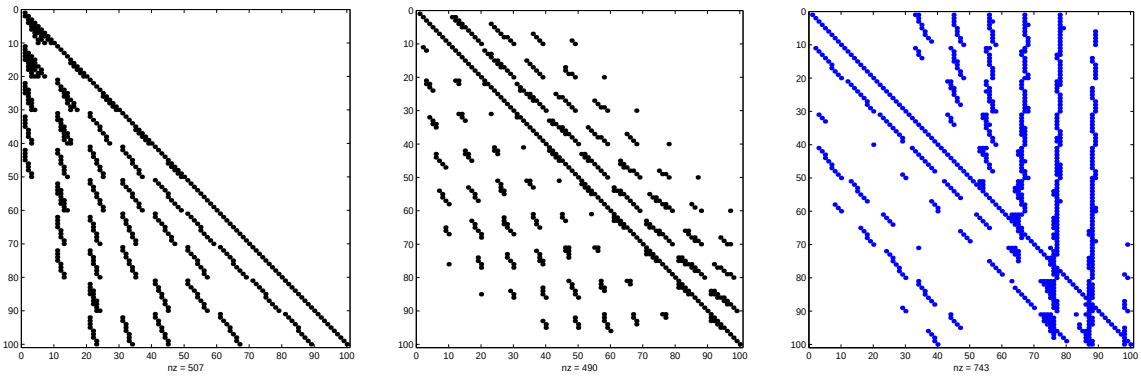


Figure 6.1: Translational motion observed in the combined blur.



(a) (b) (c)

Figure 6.2: Sparsity patterns corresponding to three types of blur: (a) isotropic blur, (b) rotation blur, (c) combined blur.

These blurring effects are modeled by H matrix which is constructed using Eq. 2.12. We have used three different image sizes of 200×150, 400×300, and 800×600 pixels. In Fig. 6.2 and Table 6.1, we see the sparsity patterns and the properties of the resulting matrices. We use the prefixes "iso", "rot", and "irt" to denote the isotropic, rotation, and combined (isotropic + rotation + translational) blurs, respectively. It should be noted that, the matrices have quite irregular and unstructured patterns.

6.2 Per Iteration Performance

Our experiments are carried on a Beowulf Cluster equipped with 400 MHz Intel Pentium II processors with 512KB cache size and having 64MB RAM. The operating system is Debian GNU/Linux 3.0 distribution with Linux kernel 2.4.14. The parallel algorithms are implemented using LAM/MPI 6.5.6 [1].

Tables 6.2 and 6.3 display per iteration run times of the implementations of BSCM and PSCM, in milliseconds. Besides, the serial run times and resulting speedups are given. As mentioned before, per iteration run time of BSCM is taken as the sequential run time. In Fig. 6.3 and 6.4 we depict the results graphically

Matrix	number of rows/cols	number of nonzeros					
		total	row/col avg	per row max	per row min	per col max	per col min
iso200x150	30,000	179,247	5.97	6	1	35	1
iso400x300	120,000	718,547	5.99	6	1	35	1
iso800x600	480,000	2,877,147	5.99	6	1	35	1
rot200x150	30,000	173,955	5.80	9	1	14	1
rot400x300	120,000	697,112	5.81	9	1	14	1
rot800x600	480,000	2,790,781	5.81	9	1	14	1
irt200x150	30,000	254,498	8.48	9	3	145	1
irt400x300	120,000	999,444	8.28	9	3	145	1
irt800x600	480,000	3,899,702	8.12	9	3	146	1

Table 6.1: Properties of test matrices

by giving the speedup curves of the parallel implementations.

We note that for all partitions we set the allowed imbalance ratio as 1%, hence we do not need to consider the overhead due to load imbalance in our discussions. The results of the per iteration times can be evaluated with respect to data sets, the implementation schemes used, and the algorithms employed.

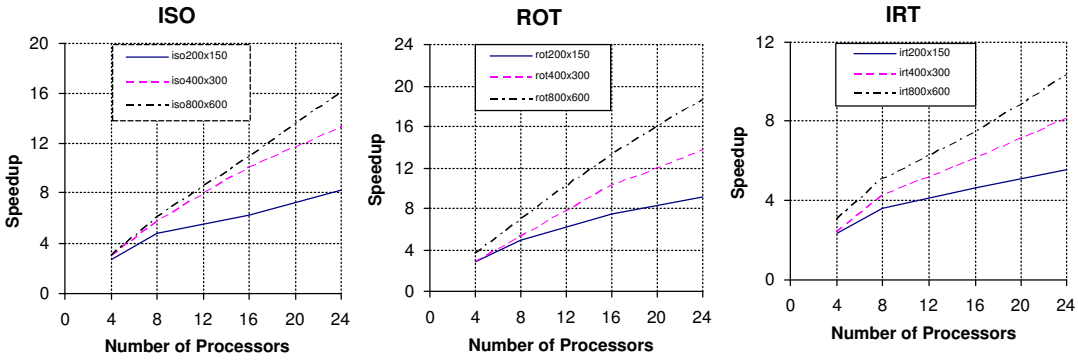
From the given results, as expected, for a fixed number of processors, as the problem size increases, speed up and hence efficiency increases. Among the blur types used, the data sets produced by the rotation blur leads to best speedups, whereas the data sets modeling combined blur leads to worst. Notice that from Table 6.1, matrices resulting from rotation blur are the most sparse, and the combined blur produces the densest matrices. In fact, with denser matrices we would expect higher speedups since the communication-computation ratio is likely to reduce. This unexpected situation results because of the sparsity pattern of the datasets. The nonzero entries of the rotation blur has a quite uniform distribution so they can be easily assigned to processors in a way to reduce the communication while preserving load balance. On the other hand, the combined blur produces fairly dense column blocks which may necessitate excessive columnwise communications. Moreover, they cannot be assigned to a single column mesh in 2D implementations since otherwise we would disturb the load balance. Hence

Data Sets	K	T_s	1D PtoP		2D PtP		2D AtoA		2D FG	
			T_p	S	T_p	S	T_p	S	T_p	S
iso200x150	4	55.7	20.4	2.73	16.3	3.42	17.0	3.28	17.1	3.26
	8	55.7	11.6	4.80	10.9	5.11	12.4	4.49	10.9	5.11
	16	55.7	8.9	6.26	7.6	7.33	11.6	4.80	8.6	6.48
	24	55.7	6.8	8.19	7.0	7.96	13.1	4.25	9.5	5.86
iso400x300	4	249.8	83.7	2.98	68.2	3.66	76.91	3.62	69.6	3.59
	8	249.8	43.9	5.69	38.1	6.56	43.0	5.81	36.4	6.86
	16	249.8	24.7	10.11	20.2	12.37	30.8	8.11	23.1	10.81
	24	249.8	18.8	13.29	16.1	15.52	28.6	8.73	20.1	12.43
iso800x600	4	1067.6	356.0	3.00	331.5	3.22	294.1	3.63	356.1	3.00
	8	1067.6	178.0	6.00	155.4	6.87	167.0	6.39	150.3	7.10
	16	1067.6	97.6	10.94	76.5	13.96	104.1	10.26	88.8	12.02
	24	1067.6	66.3	16.10	59.4	17.97	93.8	11.38	66.2	16.13
rot200x150	4	57.6	20.1	2.87	15.9	3.62	17.4	3.31	15.7	3.67
	8	57.6	11.8	4.88	9.6	6.00	11.4	5.05	9.6	7.60
	16	57.6	7.7	7.48	6.9	8.35	8.7	6.62	6.5	8.86
	24	57.6	6.3	9.14	6.4	9.00	9.0	6.40	7.0	8.23
rot400x300	4	254.4	90.5	2.81	69.4	3.67	73.1	3.48	71.3	3.57
	8	254.4	48.3	5.27	35.1	7.25	40.4	6.30	39.7	6.41
	16	254.4	24.9	10.22	20.1	12.66	23.0	11.06	20.4	12.47
	24	254.4	18.6	13.68	15.3	16.63	20.4	12.47	15.1	16.85
rot800x600	4	1365.7	377.3	3.62	306.9	4.45	315.9	4.32	313.8	4.35
	8	1365.7	199.2	6.86	149.4	9.14	162.7	8.39	158.6	8.61
	16	1365.7	103.2	13.23	80.7	16.92	87.0	15.70	83.5	16.36
	24	1365.7	73.4	18.61	58.6	23.31	67.1	20.35	61.3	22.28
irt200x150	4	66.2	28.3	2.34	26.8	2.47	29.6	2.24	26.8	2.47
	8	66.2	18.4	3.60	17.3	3.83	26.0	2.55	19.3	3.43
	16	66.2	14.4	4.60	15.4	4.30	25.3	2.62	17.2	3.85
	24	66.2	11.9	5.56	12.8	5.17	24.4	2.71	19.6	3.38
irt400x300	4	286.3	119.7	2.39	115.2	2.49	131.0	2.19	103.6	2.76
	8	286.3	68.1	4.20	64.6	4.43	83.2	3.44	77.1	3.71
	16	286.3	46.7	6.13	60.0	4.77	76.3	3.75	44.6	6.42
	24	286.3	35.1	8.16	44.0	6.51	71.2	4.02	41.4	6.92
irt800x600	4	1516.0	506.4	2.99	407.3	3.72	545.5	2.78	462.8	3.28
	8	1516.0	298.0	5.09	242.2	6.26	339.1	4.47	312.6	4.85
	16	1516.0	202.9	7.47	233.6	6.49	354.7	4.27	142.5	10.64
	24	1516.0	146.5	10.35	149.5	10.14	266.5	5.69	127.0	11.94

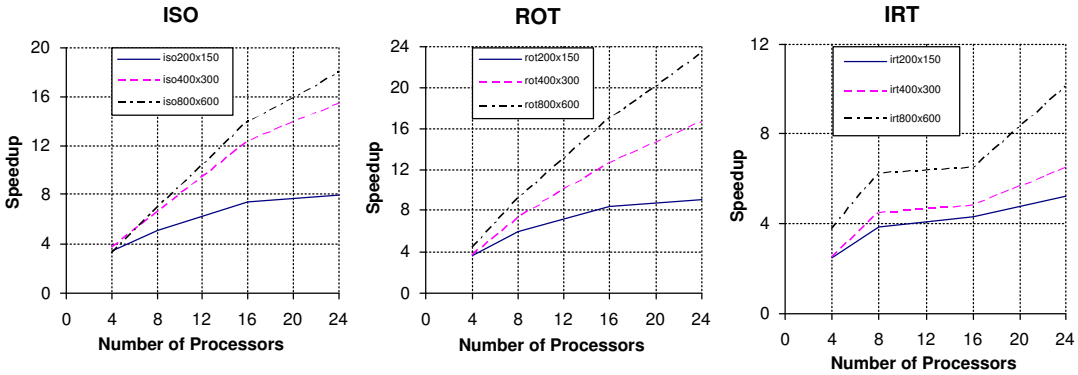
Table 6.2: Per iteration execution times of BSCM

Data Sets	K	T_s	1D PtoP		2D PtoP		2D AtoA	
			T_p	S	T_p	S	T_p	S
iso200x150	4	55.7	20.0	2.79	17.1	3.26	17.8	3.06
	8	55.7	11.8	4.72	10.7	5.21	13.0	2.64
	16	55.7	8.0	6.96	7.5	7.43	11.3	2.58
	24	55.7	7.4	7.53	7.0	7.96	10.7	2.48
iso400x300	4	249.8	84.5	2.96	70.3	3.55	70.8	3.53
	8	249.8	42.1	5.93	38.1	6.56	44.0	5.68
	16	249.8	25.0	9.99	20.3	12.31	30.7	8.14
	24	249.8	17.8	14.03	16.4	15.23	29.2	8.55
iso800x600	4	1067.6	366.1	2.92	318.9	3.35	295.6	3.61
	8	1067.6	184.3	5.79	159.1	6.71	174.7	6.11
	16	1067.6	99.7	10.71	81.2	13.15	105.5	10.12
	24	1067.6	68.1	15.68	57.2	18.66	93.1	11.47
rot200x150	4	57.6	20.3	2.84	17.1	3.37	18.2	3.16
	8	57.6	12.1	4.76	9.8	5.88	11.7	4.92
	16	57.6	8.4	6.89	7.4	7.78	8.9	6.47
	24	57.6	6.0	9.60	6.6	8.73	9.0	6.40
rot400x300	4	254.4	93.6	2.72	73.9	3.44	75.0	3.39
	8	254.4	48.5	5.25	36.5	6.97	41.2	6.17
	16	254.4	24.7	10.30	20.6	12.35	23.5	10.83
	24	254.4	18.6	13.68	45.2	17.69	20.9	12.17
rot800x600	4	1365.7	390.4	3.50	324.2	4.21	320.7	4.26
	8	1365.7	203.3	6.72	158.2	8.63	168.5	8.11
	16	1365.7	104.7	13.04	86.2	15.84	89.4	15.28
	24	1365.7	74.4	18.36	61.8	22.10	65.1	20.98
irt200x150	4	66.2	28.7	2.31	28.1	2.36	30.6	2.16
	8	66.2	18.1	3.66	17.7	3.74	26.5	2.49
	16	66.2	13.6	4.87	16.9	3.92	26.4	2.51
	24	66.2	12.8	5.17	12.8	5.17	24.9	2.66
irt400x300	4	286.3	122.7	2.33	121.4	2.36	132.3	2.16
	8	286.3	69.1	4.14	65.6	4.36	85.6	3.34
	16	286.3	46.7	6.13	57.8	4.95	76.6	3.74
	24	286.3	35.6	8.04	45.2	6.33	71.2	4.02
irt800x600	4	1516.0	521.4	2.91	439.0	3.45	516.9	2.93
	8	1516.0	310.5	4.88	254.9	5.95	341.1	4.44
	16	1516.0	201.5	7.52	232.8	6.51	364.4	4.16
	24	1516.0	148.0	10.24	146.3	10.36	277.0	5.47

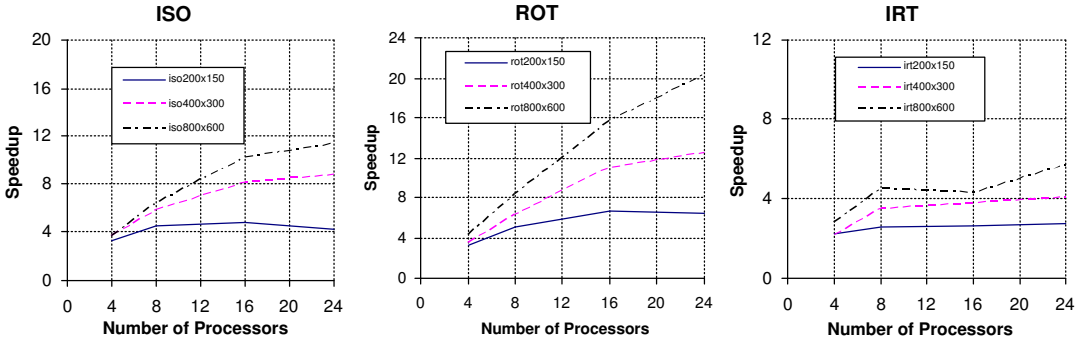
Table 6.3: Per iteration execution times of PSCM



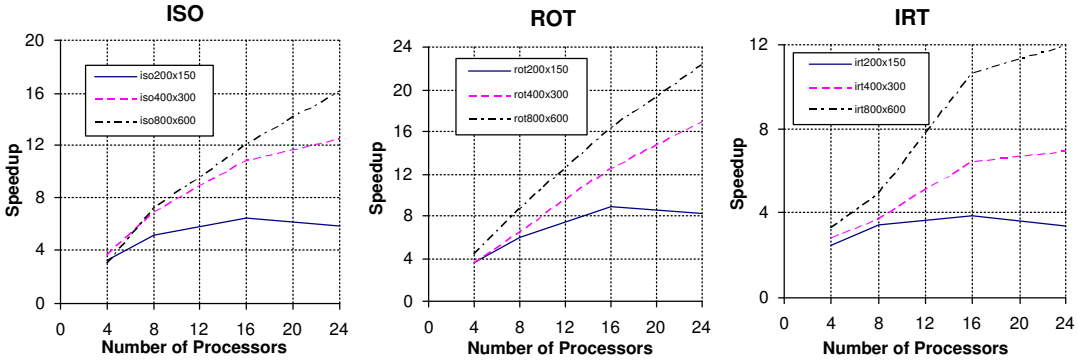
(a) 1D rowwise partitioning with point-to-point communication



(b) 2D checkerboard partitioning with point-to-point communication

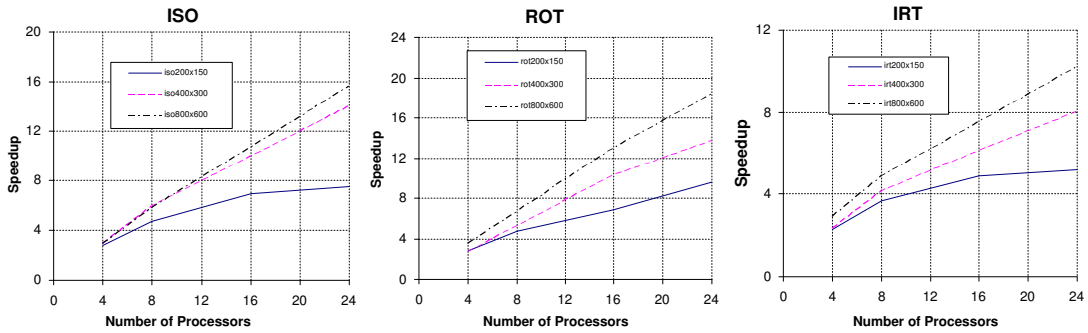


(c) 2D checkerboard partitioning with all-to-all communication

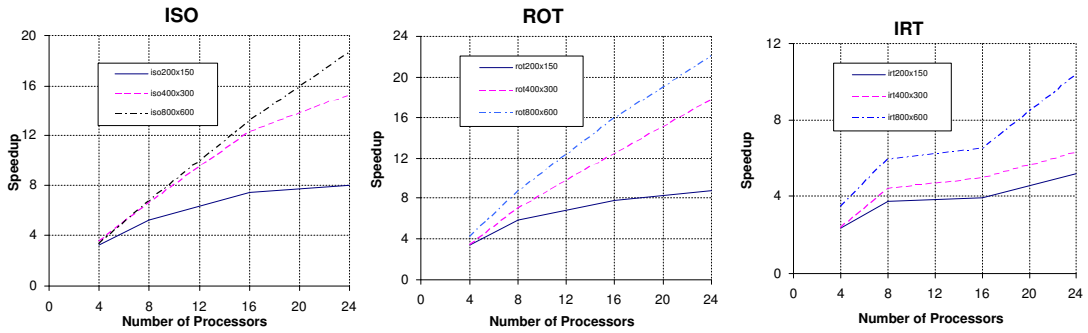


(d) 2D fine-grain partitioning with point-to-point communication

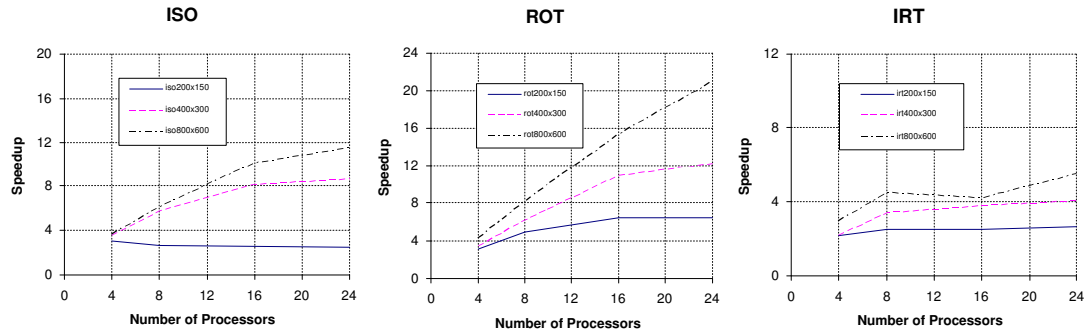
Figure 6.3: BSCM speedup curves



(a) 1D rowwise partitioning with point-to-point communication



(b) 2D checkerboard partitioning with point-to-point communication



(c) 2D checkerboard partitioning with all-to-all communication

Figure 6.4: PSCM speedup curves

these blocks also increase rowwise communication in 2D implementations. This situation can be better seen from Table 6.4 in which we give the total communication volume and the number of messages for the datasets corresponding to the 400×300 image. We give various partition results of the 2D implementations in Table 6.5. The values in these tables are the averages of ten different partitioning strategies. From these tables, it is clear that the partitioning tool has been able to reduce both the communication volume and number of messages of the rotation data set significantly better than the dataset resulting from the combined blur. The isotropic data set is somewhere in between.

To have a better chance of comparing the parallelization schemes, in Figures 6.5, 6.6, and 6.7, we give the bar charts of the parallel run times. From these figures, we can see that among the decomposition schemes utilized, for the rotation blur and the isotropic blur, 2D point-to-point communication performs better than other schemes. Even though we explicitly reduce the number of communications in 1D decomposition, 2D decomposition yields better results since it reduces the number of messages handled by a single processor from K to $r+c-2$, for a $K = r \times c$ mesh of processors. Moreover, fold and expand operations are performed along the columns and the rows of the mesh concurrently, hence communication overhead is decreased. As explained in Section 5.3.2.4, we have better communication-computation overlap in point-to-point communication compared to all-to-all communication scheme. Moreover, since in all-to-all communication scheme all processors ends up with the whole external vector, redundant communication takes place and the communication volume is increased. Fine-grain partitioning scheme though decreasing the total volume, increases the message-count since in the worst case, it induces $2(K-1)$ messages per processor. However, for the third data set, the sparsity pattern of the resulting matrices increases the columnwise and rowwise communication in 1D and 2D checkerboard partitioning schemes, hence a fine-grain partitioning strategy performs better than other parallelization schemes.

Finally, considering per iteration performance, we see that BSCM yields better speedups compared to PSCM. This is consistent with our performance analysis made in Sections 4.4 and 5.4.

Data Set	K	Total Message Volume	Total Message Num	Maximum Message Volume ($y = Ax$)	Maximum Message Num ($y = Ax$)	Maximum Message Volume ($q = \pi A$)	Maximum Message Num ($q = \pi A$)
1D Point-to-Point							
iso400x300	4	7,946.5	8.3	2,518.0	2.9	3,678.8	3.0
	8	9,956.9	27.1	1,579.8	5.3	4,150.5	6.7
	16	13,106.0	66.1	1,013.3	7.3	4,565.9	13.1
	24	15,851.6	110.2	837.2	11.8	4,279.4	17.8
rot400x300	4	616.9	6.1	178.7	2.1	281.6	2.1
	8	1,575.1	17.6	275.2	3.5	367.0	3.4
	16	3,596.6	39.8	326.7	4.7	370.8	4.2
	24	5,937.0	63.8	336.1	5.3	439.9	4.8
all400x300	4	43,011.1	12.0	11,697.1	3.0	12,318.7	3.0
	8	69,593.5	44.6	9,607.6	7.0	10,913.8	6.9
	16	104,833.5	188.4	6,977.4	15.0	7,736.8	14.5
	24	124,262.7	386.5	5,555.5	22.9	5,983.5	20.4
2D Checkerboard with Point-to-Point Communication Scheme							
iso400x300	4	6,830.2	8.0	1,905.8	2.0	1,905.8	2.0
	8	12,221.2	31.9	1,899.0	4.0	1,899.0	4.0
	16	25,198.3	95.8	2,602.1	6.0	2,602.1	6.0
	24	32,900.1	190.4	2,164.8	8.0	2,164.8	8.0
rot400x300	4	832.5	6.2	309.4	2.0	309.4	2.0
	8	2,100.0	26.0	353.1	4.0	353.1	4.0
	16	5,487.5	69.5	536.1	5.8	536.1	5.8
	24	8,881.5	136.0	610.1	7.5	610.1	7.5
all400x300	4	61,805.1	8.0	19,828.3	2.0	19,828.3	2.0
	8	96,042.2	32.0	14,186.9	4.0	14,186.9	4.0
	16	191,667.4	96.0	13,678.5	6.0	13,678.5	6.0
	24	219,250.1	192.0	10,407.2	8.0	10,407.2	8.0
2D Checkerboard with All-to-All Communication Scheme							
iso400x300	4	7,341.6	8.0	1,977.4	2.0	1,977.4	2.0
	8	23,009.7	24.0	3,165.2	3.0	3,165.2	3.0
	16	51,242.4	64.0	4,099.5	4.0	4,099.5	4.0
	24	78,321.1	120.0	4,278.0	5.0	4,278.0	5.0
rot400x300	4	699.2	8.0	267.7	2.0	267.7	2.0
	8	2,873.1	24.0	542.1	3.0	542.1	3.0
	16	10,964.7	64.0	1,096.9	4.0	1,096.9	4.0
	24	18,194.0	120.0	1,148.1	5.0	1,148.1	5.0
all400x300	4	55,349.6	8.0	17,000.4	2.0	17,000.4	2.0
	8	118,787.5	24.0	17,598.1	3.0	17,598.1	3.0
	16	291,516.0	64.0	20,749.0	4.0	20,749.0	4.0
	24	371,198.0	120.0	17,564.2	5.0	17,564.2	5.0
2D Fine-grain							
iso400x300	4	6,957.9	17.3	4,984.7	5.4	4,984.7	5.4
	8	8,191.9	50.5	4,601.5	9.3	4,601.5	9.3
	16	10,580.3	130.2	4,627.5	16.5	4,627.5	16.5
	24	12,804.2	206.2	4,567.2	21.6	4,567.2	21.6
rot400x300	4	575.6	12.8	208.3	4.4	208.3	4.4
	8	1,185.9	28.0	220.5	4.0	220.5	4.0
	16	2,463.9	61.2	230.9	4.6	230.9	4.6
	24	3,739.5	100.2	250.0	6.0	250.0	6.0
all400x300	4	40,291.2	24.0	20,244.2	6.0	20,244.2	6.0
	8	64,555.5	111.5	17,124.9	14.0	17,124.9	14.0
	16	97,661.5	441.4	15,981.1	30.0	15,981.1	30.0
	24	117,222.0	919.0	15,751.4	45.8	15,751.4	45.8

Table 6.4: Partition results

2D Point-to-Point									
		Expand Ax / Fold πA				Fold Ax / Expand πA			
		Volume		Number		Volume		Number	
Data Sets	K	Total	Max	Total	Max	Total	Max	Total	Max
iso	4 (2×2)	5,288.4	1,442.9	4.0	1.0	1,541.8	464.0	4.0	1.0
	8 (4×2)	8,322.1	1,400.4	23.9	3.0	3,899.1	783.5	8.0	1.0
	16 (4×4)	8,854.5	764.6	47.8	3.0	16,343.8	2,047.8	48.0	3.0
	24 (6×4)	10,892.2	575.5	118.4	5.0	22,007.9	1,741.9	72.0	3.0
rot	4 (2×2)	220.5	110.2	2.2	1.0	612.0	199.4	4.0	1.0
	8 (4×2)	894.5	190.2	18.0	3.0	1,205.5	217.0	8.0	1.0
	16 (4×4)	904.2	151.4	21.5	2.8	4,583.3	474.3	48.0	3.0
	24 (6×4)	1,579.0	135.6	64.0	4.5	7,302.5	544.6	72.0	3.0
all	4 (2×2)	16,146.4	6,154.8	4.0	1.0	45,658.7	13,674.0	4.0	1.0
	8 (4×2)	42,836.5	6,372.7	24.0	3.0	53,205.7	7,913.3	8.0	1.0
	16 (4×4)	43,063.0	3,413.6	48.0	3.0	148,604.4	10,944.1	48.0	3.0
	24 (6×4)	61,790.7	3,064.5	120.0	5.0	157,459.4	7,713.2	72.0	3.0
2D All-to-All									
iso	4 (2×2)	5,507.0	1,428.8	4.0	1.0	1,834.6	548.6	4.0	1.0
	8 (4×2)	18,919.8	2,441.8	16.0	2.0	4,089.9	723.4	8.0	1.0
	16 (4×4)	19,379.4	1,338.6	32.0	2.0	31,863.0	2,760.9	32.0	2.0
	24 (6×4)	37,604.5	1,722.1	72.0	3.0	40,716.6	2,555.9	48.0	2.0
rot	4 (2×2)	219.1	109.7	4.0	1.0	480.1	158.0	4.0	1.0
	8 (4×2)	1,733.7	297.1	16.0	2.0	1,139.4	245.0	8.0	1.0
	16 (4×4)	1,745.7	257.6	32.0	2.0	9,219.0	839.3	32.0	2.0
	24 (6×4)	4,706.0	334.6	72.0	3.0	13,488.0	813.5	48.0	2.0
all	4 (2×2)	16,178.7	5,369.8	4.0	1.0	39,170.9	11,630.6	4.0	1.0
	8 (4×2)	73,502.1	10,762.3	16.0	2.0	45,285.4	6,835.8	8.0	1.0
	16 (4×4)	73,054.8	5,416.0	32.0	2.0	218,461.2	15,333.0	32.0	2.0
	24 (6×4)	139,044.5	6,985.6	72.0	3.0	232,153.5	10,578.6	48.0	2.0
2D Fine-grain									
iso	4 (2×2)	6,716.6	4,917.8	10.6	3.0	241.3	94.1	6.7	2.4
	8 (4×2)	7,745.4	4,536.5	34.7	6.6	446.5	86.9	15.8	2.8
	16 (4×4)	9,563.0	4,543.1	95.7	13.4	1,017.3	117.8	34.5	3.3
	24 (6×4)	11,077.4	4,498.3	153.0	18.6	1,726.8	169.4	53.2	3.6
rot	4 (2×2)	244.0	89.7	6.4	2.2	331.6	120.1	6.4	2.2
	8 (4×2)	510.3	92.8	14.0	2.0	675.6	133.7	14.0	2.0
	16 (4×4)	1,050.7	102.4	30.6	2.3	1,413.2	135.9	30.6	2.3
	24 (6×4)	1,517.0	104.4	50.1	3.0	2,222.5	163.6	50.1	3.0
all	4 (2×2)	22,303.4	12,880.1	12.0	3.0	17,987.8	7,629.8	12.0	3.0
	8 (4×2)	35,346.9	12,632.1	55.7	7.0	29,208.6	5,889.5	55.8	7.0
	16 (4×4)	51,636.6	12,699.6	224.8	15.0	46,024.9	4,152.4	216.6	15.0
	24 (6×4)	61,067.5	13,394.2	476.2	23.0	56,154.5	3,591.9	442.8	22.8

Table 6.5: Partition results for 2D decompositions

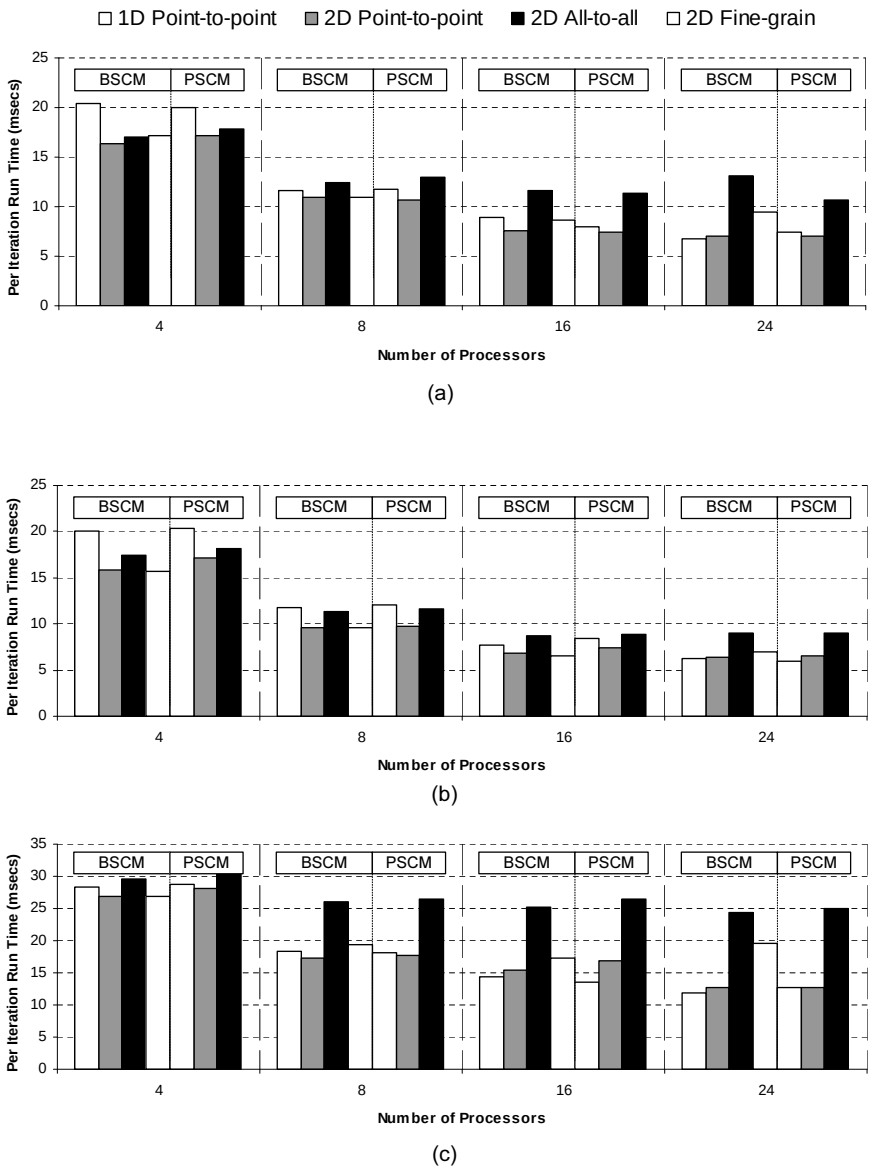


Figure 6.5: Parallel execution times of the implementations with 200×150 image: (a) isotropic blur, (b) rotation blur, (c) combined blur.

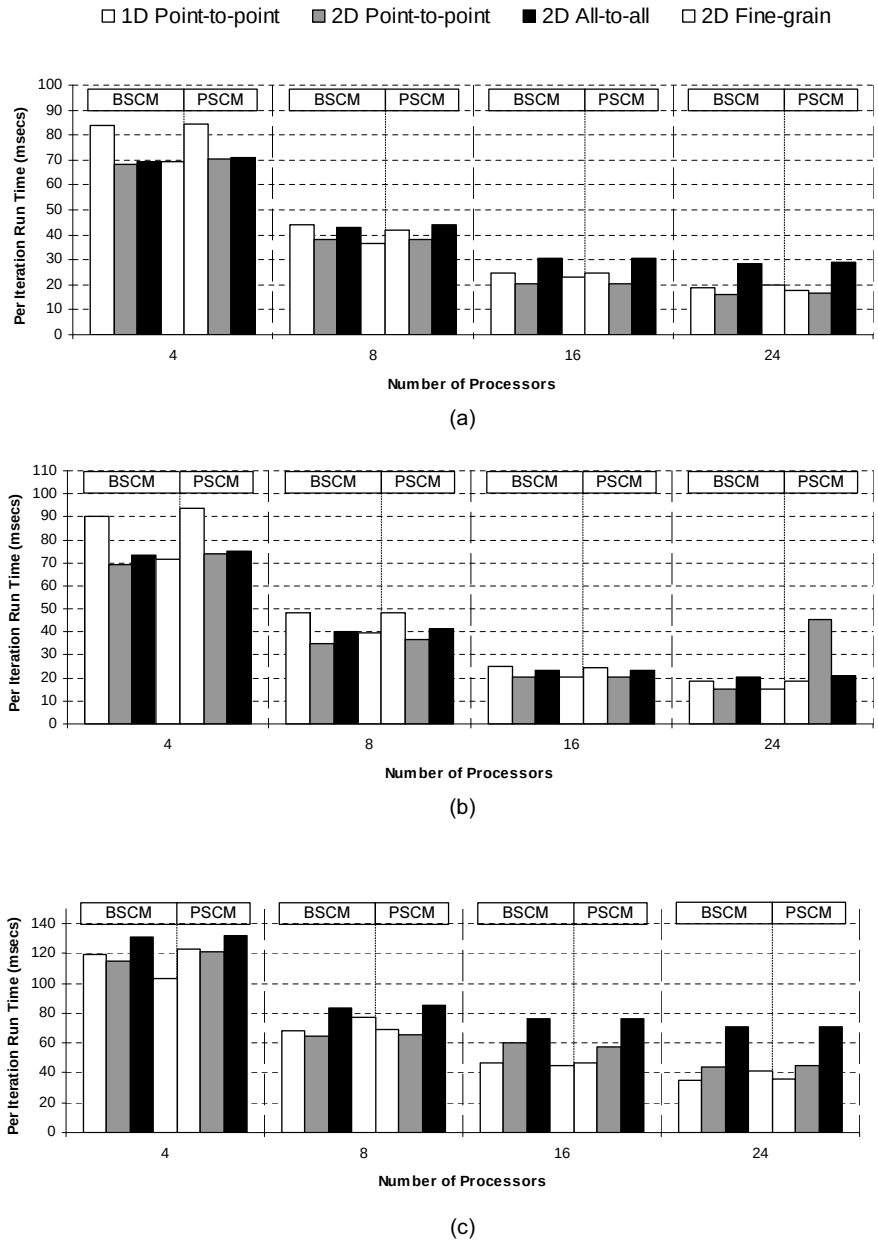


Figure 6.6: Parallel execution times of the implementations with 400×300 image: (a) isotropic blur, (b) rotation blur, (c) combined blur.

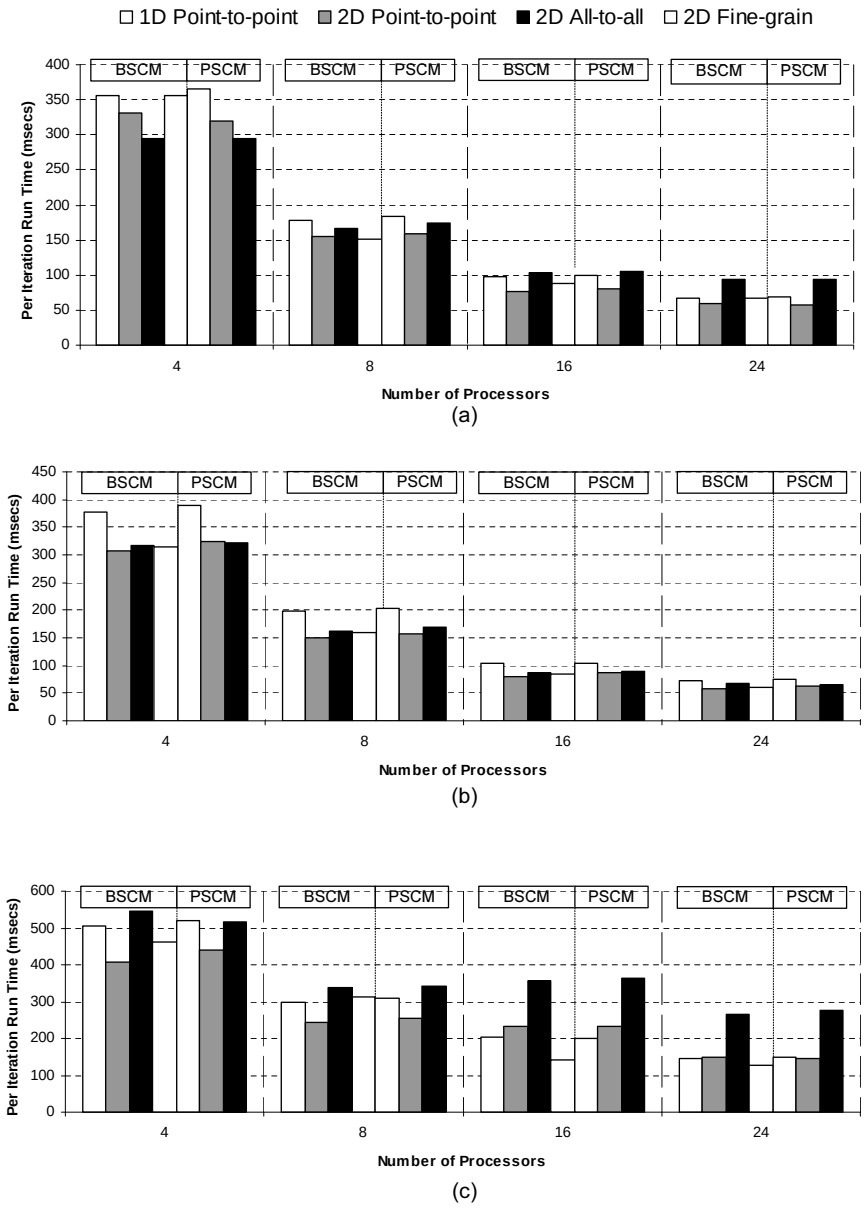


Figure 6.7: Parallel execution times of the implementations with 800×600 image: (a) isotropic blur, (b) rotation blur, (c) combined blur.

6.3 Overall Performance

Iterative methods have the flexibility that the system parameters effecting convergence can be explicitly set by the user according to the requirements of the application. As far as image restoration is concerned, by decreasing the tolerance parameter, one can have better restorations with the cost of increasing computational time. In our experiments we have set the tolerance parameter as 5% of the mean value of the observed image. The other parameter effecting convergence is the relaxation parameter λ, which is taken as 1.7. The starting point of the iterations is taken as the zero vector which means that every pixel in the image to be recovered is assumed to be black, initially. With these values the number of iterations required for convergence are given in Table 6.6. In this table the column 'Block' refers to block-striped partitioning in which each processor is assigned contiguous rows, 'Cyclic' means sequential distribution of the rows of the matrix in a wraparound manner, and 'HG' denotes the partitioning schemes obtained using hypergraph partitioning tools. In order to be able to make comparisons among different partitioning strategies, we have taken the results with different partitioning strategies.

From Table 6.6 we see that the permutation of the rows severely affects the convergence behavior of the surrogate constraint methods. The effect of the hypergraph partitioning is clustering nonzero elements to diagonal blocks to reduce the communication costs. When the blocks of a matrix are clustered in such a way, the effect is likely to reduce the dimension of the system. To see this, consider the extreme case in which all elements of a matrix A are clustered in the diagonal blocks and assume that matrix and the x-vector are partitioned into K blocks as in 1D case. Then kth row block of the system defines a constraint in the form $A_k x \leq b_k$, or $A_{k1}x_1 + \cdots + A_{kl}x_l + \cdots A_{kK}x_K$, where A_{kl} is the column block corresponding to x_k. But if, $x_l = 0$ for $l \neq k$, then each block consists of $\approx N/K$ size independent systems. In [29] it is given that speed of convergence is proportional to N^2/M, where N is the dimension of the system and M is the number of constraints. Hence, clustering results in reduced-decoupled systems and increases the convergence performance.

Data Sets	K	PSCM			BSCM
		Block	Cyclic	HG	
iso200x150	4	6591	3834	4334	8508
	8	5185	3398	3300	
	16	4526	3286	3041	
	24	4845	3421	3158	
iso400x300	4	20451	10370	11297	24721
	8	15112	9231	8965	
	16	12531	8002	7850	
	24	12665	8789	8516	
iso800x600	4	41312	30785	31781	64909
	8	38154	26351	24857	
	16	31159	21265	19124	
	24	26720	17421	14789	
rot200x150	4	442	387	192	511
	8	391	262	153	
	16	279	223	123	
	24	272	186	108	
rot400x300	4	849	342	386	1745
	8	809	341	281	
	16	507	560	233	
	24	501	577	199	
rot800x600	4	4804	1548	645	2647
	8	907	1113	323	
	16	798	1664	272	
	24	628	1677	300	
irt200x150	4	8949	1813	1477	12267
	8	6745	1513	1302	
	16	5885	1477	1178	
	24	5628	1519	1108	
irt400x300	4	48678	12984	8429	79044
	8	37284	11776	7385	
	16	31181	11746	6403	
	24	34257	11026	5767	
irt800x600	4	141928	38193	19479	171129
	8	88336	34113	12748	
	16	71759	33988	9249	
	24	49967	33197	8993	

Table 6.6: Overall number of iterations of the surrogate constraint methods

The decrease in the speed of convergence substantially improves the overall performance of PSCM especially with increasing size of the blocks. The overhead induced by the extra computations (which is mentioned in Section 3.4) does not much effect the parallel performance, however it has the nice effect of accelerating the convergence rate. In Fig. 6.8 we give the overall performance results of the parallel methods for the 400×300 pixel image. The increase in the number of iterations makes PSCM much more favorable compared to BSCM for the overall case. Moreover, we see that 1D implementation is superior to 2D in the overall performance, since in 1D case number of processors is equal to number of row blocks, where in 2D case number of blocks is equal to number of row-meshes. As the number of blocks increases, the iteration number decreases, hence with 1D partitioning we get better overall performance results.

Finally, we consider the preprocessing overhead of our parallel implementations. In Table 6.7 we give the partitioning times of the matrices expressed in terms of the number of iterations. With an increasing number of processors, the preprocessing time increases and may become comparable to the overall run times of the algorithms. However, as well as yielding efficient parallelization, hypergraph partitioning methods substantially decreases the iteration numbers. So, although producing high preprocessing overhead, this partitioning strategy increases the performance of the methods.

6.4 Restoration Results

To evaluate the restoration performance of the parallel methods, we have chosen the image f shown in Fig. 6.9. Blurred image g is generated using Eq. 2.7, or by simply multiplying f with H so that $g = Hf$. In Fig. 6.10 the resulting distorted images are shown for the three types of blurs.

With the same parameter values given in Section 6.3 we have restored the images by the surrogate constraint methods. The results corresponding to three blurs are given in Fig. 6.11. Even though the images become fairly deblurred

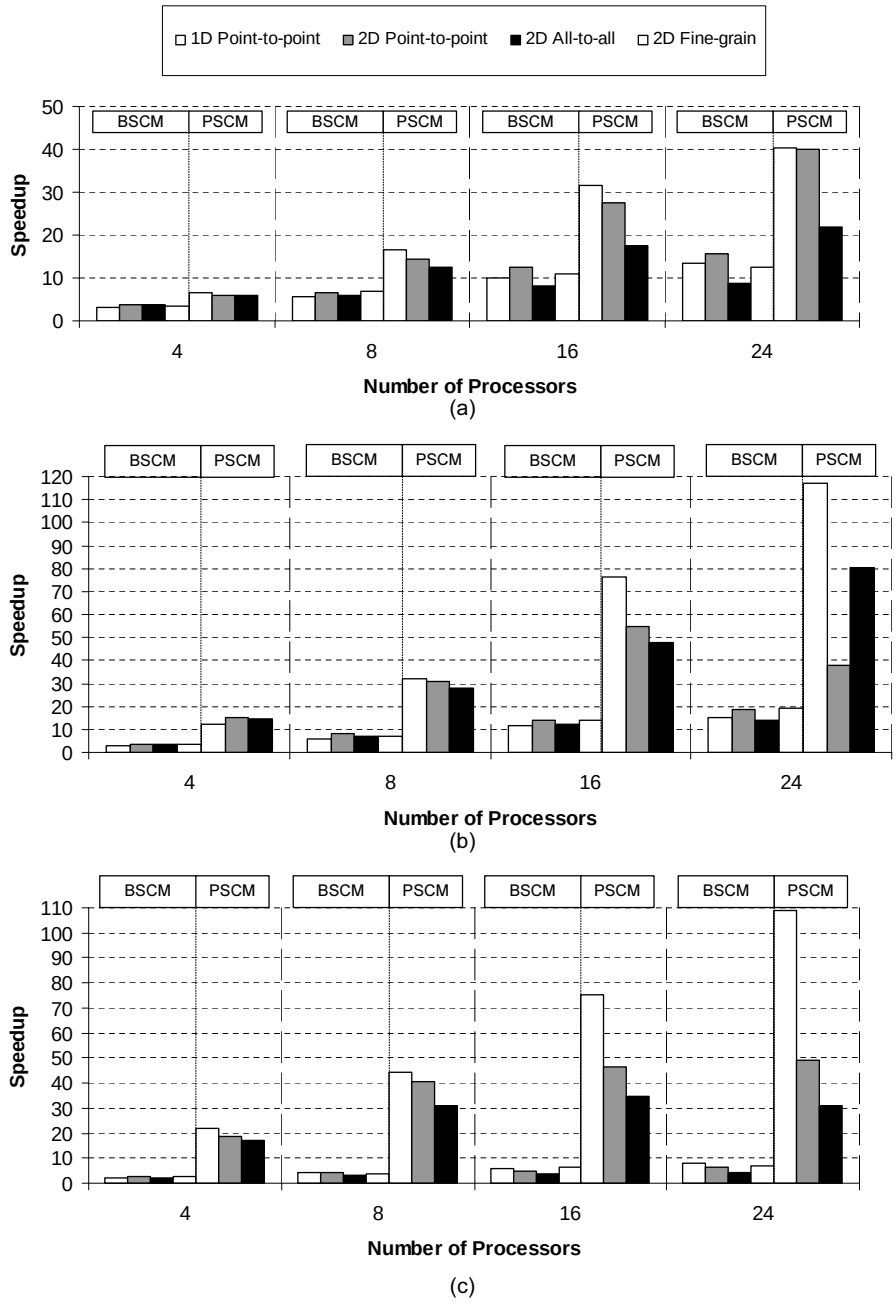


Figure 6.8: Overall speedup charts of the parallel methods:
(a) isotropic blur, (b) rotation blur, (c) combined blur.

Data Sets	K	Parallel Scheme			
		1D P2P	2D P2P	2D A2A	2D FG
iso400x300	4	125	115	220	285
	8	291	302	149	803
	16	618	835	110	1709
	24	912	1287	81	2187
rot400x300	4	68	83	144	285
	8	184	248	110	816
	16	458	553	70	1892
	24	699	858	55	2964
irt400x300	4	335	230	514	296
	8	1413	596	494	59
	16	4093	1290	258	1367
	24	5228	3106	247	1705

Table 6.7: Preprocessing times for the data sets of the 400×300 Image. Expressed in terms of the per iteration times of the corresponding PSCM implementation.

for this case, we can have better results by decreasing the tolerance parameter. One such result is given in Fig. 6.12 for the rotation blur with a tolerance value decreased to 0.1% of the mean value of the observed image.

Figure 6.9: Original image

(a) (b) (c)

Figure 6.10: Blurred images: (a) isotropic blur, (b) rotation blur, (c) combined blur.

(a) (b) (c)

Figure 6.11: Restored images: (a) isotropic blur, (b) rotation blur, (c) combined blur.

Figure 6.12: Restored image with decreased tolerance value

Chapter 7

Conclusion

In this thesis, we are concerned with the image restoration problem by formulating it as a system of linear inequalities. The coefficient matrix of the system is a non-structured sparse-matrix and even with small size images we end up with huge matrices. We have used the surrogate constraint methods proposed by Yang and Murty [29], which can work efficiently for the large size problems and are amenable for parallel implementations. Among the proposed methods, we have decided to use BSCM, which is the basic method proposed, and a modified version of the parallel method which is proposed by Özaktaş et al. [23]. For the solution of the system we have used several parallel implementation schemes. Since the nonzero patterns of the resulting matrices are irregular, decomposition of the problem data for efficient parallelization is a non-trivial process. We have used hypergraph partitioning based methods which minimizes communication overhead significantly and maintains the load balance at the same time.

For 1D decomposition of the problem data we have explicitly tried to minimize the total volume and the number of communications. At the same time, communication balance is tried to be maintained among the processors as well as the load balance. We have adopted a point-to-point communication scheme, in which the sparsity of the matrices are exploited and no redundant communication is performed. For 2D checkerboard decomposition, first we have performed the communications in a point-to-point manner using the connectivity metric of

the cut size definition, and then we tried to minimize the the external parts of the vector components using cutnet metric of the cut size. Then, in-place, all-to-all communications are performed along the external parts. Finally for BSCM, we have decomposed the matrix in a fine-grain manner on a nonzero basis, and performed the communications in a point-to-point manner.

To evaluate the performance of the parallel implementations and the restoration methods, we used three types of blurs with three different size images constituting 9 data sets. All of the blurs denote serious distortions which have non-local, space-variant, and anisotropic nature. The experimental results of the parallel implementations are evaluated on a per iteration basis and with respect to overall performance. According to our experimental results, we end up with the following arguments for the parallelization schemes utilized:

- 1D point-to-point communication scheme is not as scalable as its 2D counterpart, namely 2D checkerboard partitioning with point-to-point communication scheme. Though with 1D rowwise decomposition of the matrix we respect the row coherence and no rowwise communication takes place, concurrent communications performed along the rows and columns of the processor mesh decreases the communication overhead of the checkerboard partitioning significantly. However, considering the overall performance, 1D decomposition produces better speedup since the number of iterations required for convergence decreases significantly with the increasing number of row blocks of the system.

- Among 2D decomposition schemes, we see that point-to-point communication scheme produces better results compared to all-to-all communication scheme, even though we can perform in-place communications for all-to-all case. This is mainly because of the increasing communication volume confronted in all-to-all communication. Note that the performance of all-to-all communication scheme is fairly good for the rotation blur for which the communication volume is not significant. Moreover, in all-to-all communication scheme with checkerboard partitioning, number of communications

per processor scales with $\lceil \log_2(r+c) \rceil$, where r and c is the number of processors per column and per row respectively. So, with an increasing number of processors, all-to-all communication scheme is likely to produce better results.

- With a fine-grain partitioning of the problem data, for some data sets we can achieve satisfactory results. Note that, fine-grain partitioning decreases the communication volume significantly however is apt to increase the message-count since it does not respect neither row nor column coherence and can produce up to $2(K-1)$ messages per processor. However, if the communication volume requirements are high as in the combined blur case, fine-grain partitioning strategy can outperform other partitioning schemes.

Considering the parallel methods, BSCM produced better results compared to PSCM for the per iteration case. However, though causing some extra cost, increasing the number of blocks accelerates the convergence rate significantly, hence PSCM outperforms BSCM considering the overall performance.

We should also mention the positive effect of the partitioning strategy on the convergence rate. We have seen that the permutations performed on the rows of the coefficient matrix during the partitioning process significantly reduces the iteration number required for convergence. We think that, this is due to the decoupling effect induced on the blocks of the system during the partitioning process.

Concerning the restoration performance, we see that satisfactory restorations can be achieved especially by decreasing the tolerance parameter, with the incoming cost of increased computational time. Actually, the system parameters can be set according to the requirements of the application. Moreover, the iterative restoration technique has the advantage that, the image can be viewed during the restoration process and the process can be terminated after a while when the restoration level satisfies the application requirements.

Finally we note that the parallel implementations performed in this work can also be used in other real-world applications which can be formulated as a linear

feasibility problem and requires processing of large data sets. The process of *image reconstruction from projections* used in computerized tomography is such an example [13].

Bibliography

[1] G. Burns, R. Daoud, and J. Vaigl. Lam: an open cluster environment for mpi. In J. W. Ross, editor, *Proceedings of Supercomputing Symposium*, pages 179–186, 1994.

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

[3] Ü. V. Çatalyürek and C. Aykanat. A fine-grain hypergraph model for 2d decomposition of sparse matrices. In *Scientific Computing 2001*, 2001.

[4] Ü. V. Çatalyürek and C. Aykanat. A hypergraph-partitioning approach for coarse-grain decomposition. *Proceedings of Scientific Computing*, pages 10–16, November 2001.

[5] Y. Censor. *Parallel optimization: theory, algorithms, and applications*. Oxford University Press, 1997.

[6] Y. Censor and T. Elfving. New method for linear inequalities. *Linear Algebra and its Applications*, 42:199–211, 1982.

[7] G. Cimmino. Calcolo approssimato per le seluzioni dei sistemi di equizoni lineari. *Ricerca Scientifica*, 1:326–333, 1938.

[8] R. C. Gonzalez and R. E. Woods. *Digital Image Processing*. Addison-Wesley, 2001.

[9] L. G. Gubin, B. T. Polyak, and E. V. Raik. The method of projections for finding the common point of convex sets. *USSR Computational Mathematics and Mathematical Physics*, 6:326–333, 1967.

[10] B. Hendrickson and T. G. Kolda. Partitioning rectangular and structurally nonsymmetric sparse matrices for parallel processing. *SIAM Journal of Scientific Computing*, 21(6):2048–2072, 1998.

[11] B. Hendrickson and T. G. Kolda. Graph partioning models for parallel computing. *Parallel Computing*, 26:1519–1534, 2000.

[12] B. Hendrickson, R. Leland, and S. Plimpton. An efficient parallel algorithm for matrix-vector multiplication. *International Journal of High Speed Computing*, 7(1):73–88, 1995.

[13] G. T. Herman. *Image Recostruction from Projections, The Fundamentals of Computerized Tomography.* Academic Press, 1980.

[14] M. Jacunski, P. Saddayapan, and D. K. Panda. All-to-all broadcast on switch-based clusters of workstations. *IPPS / SPDP*, pages 325–329, 1999.

[15] S. Kacmarz. Angenherte auflösung von systemn linearer gleichungen. *Sciences Mathematiques et Naturelles*, 35:355–357, 1937.

[16] J. Kowalik, editor. *MPI: The Complete Reference.* The MIT Press, 1996.

[17] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing.* The Benjamin/Cummings Publishing Company, 1994.

[18] R. L. Lagendijk and J. Biemond. *Iterative Identification and Restoration of Images.* Kluwer Academic Publishers, 1991.

[19] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout.* Wiley-Teubner, 1990.

[20] J. G. Lewis, D. G. Payne, and R. A. van de Geijn. Matrix-vector multiplication and conjugate gradient algorithms on distributed memory computers. In *Proceedings of the Scalable High Performance Computing Conference*, pages 943–948, November 1994.

[21] J. G. Lewis and R. A. van de Geijn. Distributed memory matrix-vector multiplication and conjugate gradient algorithms. *IEEE, Proceedings of Supercomputing*, 1993.

[22] H. Özaktaş. *Algorithms for Linear and Convex Feasibility Problems: A Brief Study of Iterative Projection, Localization and Subgradient Methods.* PhD thesis, Bilkent University, 1996.

[23] H. Özaktaş, M. Ç. Pınar, and M. Akgül. The parallel surrogate constraint approach to the linear feasibility problem. *Lecture Notes in Computer Science*, pages 565–574, 1996.

[24] H. Özaktaş, M. Ç. Pınar, and M. Akgül. Restoration of space-variant global blurs caused by severe camera movements and coordinate distortions. *Journal of Optics*, 29(303-310), 1998.

[25] G. Palomares and G. Castano. Acceleration technique for solving convex (linear) systems via projection methods. Technical report, ESCOLA TECNICA SUPERIOR DE ENXENEIROS DE TELECOMUNICACION, 1996.

[26] Y. Saad. *Iterative Methods for Sparse Linear Systems.* PWS Publishing, Boston, second edition, January 1996.

[27] B. Uçar and C. Aykanat. Parmxvlib: A library for sparse-matrix vector multiplies. In *Proceedings of the 7th World Multiconference on Systemics, Cybernetics and Informatics (SCI'2003)*, pages 393–398, July 2003.

[28] B. Uçar and C. Aykanat. Encapsulating multiple communication-cost metrics in partitioning sparse rectengular matrices for parallel matrix-vector multiplies. *SIAM Journal of Scientific Computing*, to appear.

[29] K. Yang and K. G. Murty. New iterative methods for linear inequalities. *Journal of Optimization Theory and Applications*, pages 163–185, 1992.

Appendix A

Storage Schemes for Sparse Matrices

In order to take advantage of the large number of zero elements, special schemes are required to store sparse matrices. The main goal is to represent only the nonzero elements, and to be able to perform the common matrix operations in an efficient way.

The simplest storage scheme for sparse matrices is the so-called coordinate format. The data structure consists of three arrays to store an $M \times N$ matrix with Z nonzero entries:

- An array VAL of length Z containing all the values of the nonzero elements in any order.
- An integer array JA of length Z containing their column indices.
- An integer array IA of length Z containing their row indices

For example, tha matrix,

$$A = \begin{pmatrix} 1 & 0 & 0 & 2 & 0 \\ 0 & 3 & 0 & 4 & 0 \\ 5 & 0 & 0 & 6 & 7 \\ 0 & 0 & 8 & 9 & 0 \end{pmatrix}$$

can be represented by,

VAL	2	4	5	3	8	1	6	9	7
JA	3	3	0	1	2	0	3	3	4
IA	0	1	3	1	4	0	3	4	3

Note that the row and column indices begins from 0.

If the elements of the matrix were listed by row, the array IA which contains redundant information can be replaced by an array which points to the beginning of each row instead. This would involve nonnegligible savings in storage. The new data structure has three arrays with the following functions:

- An array VAL of length Z containing all the nonzero elements stored row by row, from row 0 to $M - 1$.
- An integer array JA of length Z containing the column indices of the nonzero elements in the array VAL.
- An integer array IA containing the pointers to the beginning of each row in the arrays VAL and JA. Thus, the content of IA is the position in arrays VAL and JA where the ith row starts. The length of IA is $M + 1$, with IA[M] containing the number IA[0] + z, i.e., the address in VAL and JA of the beginning of a fictitious row number M.

So, the above matrix may be stored as follows:

VAL	1	2	3	4	5	6	7	8	9
JA	0	3	1	3	0	3	4	2	3
IA	0	2	4	7	9				

This format is probably the most popular for storing general sparse matrices. It is called the Compressed Sparse Row (CSR) format. This scheme is preferred over the coordinate scheme because it is often more useful for performing typical computations. On the other hand, the coordinate scheme is advantageous for its simplicity and its flexibility. It is often used as an entry format in sparse matrix software packages.

There are a number of variations for the Compressed Sparse Row format. The most obvious variation is storing the columns instead of the rows. The corresponding scheme is known as the Compressed Sparse Column (CSC) scheme, in which an integer array JA holds the pointers to the beginning of each column.

Appendix B

Sparse Matrix-Vector Multiplies

The multiplication of a sparse matrix with a dense vector is one of the key operations in solving systems of linear ineqaulities, as well as in many other iterative methods. It often determines the overall computational complexity of the entire algorithm of which it is a part. Thus, efficient multiplication routines are crucial.

The product $y = Ax$ with A stored in CSR format can be expressed in the usual inner product form of multiplication $y_i = A_i x = \sum_j a_{i,j} x_j$ (A_i is the ith row of A and $a_{i,j}$ are the nonzero entries), since this traverses the rows of the matrix A. For an $M \times N$ matrix, $y = Ax$ multiplication is given in Figure B.1. Since only the non-zero matrix entries are multiplied, the operation count is two times the number of non-zero entries, which is a significant saving over the dense operation requirement of $2MN$.

The second type of a SpMxV is the *outer-product* form. We can view matrix-vector multiplication $y = Ax$ as the linear combination of the column vectors of A with the x-vector elements, hence we perform the multiplication as $y = \sum_j y^j$, where $y^j = a_{*,j} x_j$. Fig. B.2 outlines this multiplication scheme for the SpMxV $y = Ax$. This type of SpMxV is again results in $2Z$ time in terms of number of floating point operations performed.

```
for i = 0 to M − 1 do
    kstart ← IA[i]
    kend ← IA[i + 1] − 1
    sum ← 0.0
    for k = kstart to kend do
        j ← JA[k]
        sum ← sum + x[j]VAL[k]
    y[i] ← sum
```

Figure B.1: Inner Product Form of Sparse Matrix Vector Product

```
y ← 0̄
for j = 0 to N − 1 do
    kstart ← JA[j]
    kend ← JA[j + 1] − 1
    for k = kstart to kend do
        i ← IA[k]
        y[i] ← y[i] + x[j]VAL[k]
    endfor
endfor
```

Figure B.2: Outer Product Form of Sparse Matrix Vector Product