Parallel Image Restoration Using Surrogate Constraint Methods ⋆

Bora Uçar [a], Cevdet Aykanat [a,*], Mustafa C. Pınar [b], Tahir Malas [c]

[a] *Department of Computer Engineering, Bilkent University, 06800, Ankara, Turkey*

[b] *Department of Industrial Engineering, Bilkent University, 06800, Ankara, Turkey*

[c] *Department of Electrical and Electronics Engineering, Bilkent University, 06800, Ankara, Turkey*

Abstract

When formulated as a system of linear inequalities, the image restoration problem yields huge, unstructured, sparse matrices even for images of small size. To solve the image restoration problem, we use the surrogate constraint methods that can work efficiently for large problems. Among variants of the surrogate constraint method, we consider a basic method performing a single block projection in each step and a coarse-grain parallel version making simultaneous block projections. Using several state-of-the-art partitioning strategies and adopting different communication models, we develop competing parallel implementations of the two methods. The implementations are evaluated based on the per iteration performance and on the overall performance. The experimental results on a PC cluster reveal that the proposed parallelization schemes are quite beneficial.

Key words: parallel computing, image restoration, linear feasibility problem, surrogate constraint method

⋆ This work is partially supported by the Scientific and Technological Research Council of Turkey (TÜBİTAK) under project EEEAG-103E028.

* Corresponding author, Tel: +90 (312) 290-1625; Fax: +90 (312) 266-4047
Email addresses: ubora@cs.bilkent.edu.tr (Bora Uçar), aykanat@cs.bilkent.edu.tr (Cevdet Aykanat), mustafap@ie.bilkent.edu.tr (Mustafa C. Pınar), tahir@bilkent.edu.tr (Tahir Malas).

1 Introduction

The purpose of the present paper is to use state-of-the-art parallel algorithms for the restoration of heavily distorted digital images. An ideal recording device would be expected to record an image with the following idealized property: the intensity of a pixel of the recorded image should be directly proportional to the corresponding section of the scene being recorded. However, this property is rarely observed in practice. Either the recorded intensity of a pixel is related to the intensity in a larger neighborhood of the corresponding section of the scene (blurring), or the recorded intensities are contaminated by random noise [21,26]. Image restoration is concerned with estimating the original scene from a distorted and noisy one. Restoration of images that have been blurred by various factors is usually posed as a linear estimation problem obtained from a discretization process, where the characteristics of the blurring system and the noise are assumed to be known a priori [21]. The mathematical model of the recording operation which is usually an integral equation is discretized to yield a linear system that is solved by a host of direct and iterative methods [12,22,26,28]. Among these methods, we focus on iterative methods. The advantage of iterative methods is that they allow a flexible and improved formulation of the restoration problem [21], and the large dimensions involved in image restoration make these methods favorable.

We follow the work [30] and pose the image restoration problem as a *linear feasibility problem* [8]. The linear feasibility problem asks for a point that satisfies a set of linear inequalities. In matrix notation, given an $M \times N$ matrix A and $M \times 1$ vector b, the problem is to find an $N \times 1$ vector x such that

$$Ax \leq b. \tag{1}$$

We use iterative methods for solving the linear feasibility problem. An important class of iterative methods for the linear feasibility problem is the *projection methods* developed for the solution of the linear systems by Kacmarz (cited in [32]), and Cimmino (cited in [8]). Kacmarz's and Cimmino's works are extended to linear inequalities by Gubin et al. [11] and Censor and Elfving [7]. The method in [11] is known 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 satisfying all inequalities is found. The method in [7] is known as the *simultaneous orthogonal projections method*, where 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.

Kacmarz and Cimmino type methods become computationally very expensive when applied to image restoration problem, mainly because a projection is

made for each violated constraint. The surrogate constraint methods proposed by Yang and Murty [35] and the one that we use here eliminate this problem by processing a group of violated constraints at a time. At each iteration, a surrogate constraint is derived from a group of violated constraints, and the current point is orthogonally projected onto this surrogate constraint. The process is repeated until a feasible solution is found.

Yang and Murty [35] proposed three variants of surrogate constraint methods (see Section 3 for an overview). The *Basic Surrogate Constraint Method* (BSCM) takes all violated constraints in the system and makes successive projections of the current point. The *Sequential Surrogate Constraint Method* (SSCM) and the *Parallel Surrogate Constraint 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. Although the original PSCM converges slowly compared to SSCM, it becomes competitive with an adjusted step sizing rule [30].

Since BSCM has been shown to be faster than SSCM [30], we give efficient parallelizations of BSCM and the improved version of PSCM. State-of-the-art partitioning strategies are employed in the present paper for the parallelization of these two methods. Both BSCM and PSCM involve repeated matrix-vector and matrix-transpose-vector multiplies, and regular operations such as inner products and vector updates. Partitioning sparse matrices is a crucial issue in the parallelization of BSCM and PSCM, since matrix-vector multiplies incur irregular dependencies. Both one-dimensional (1D), e.g., rowwise, and two-dimensional (2D), e.g., checkerboard, sparse matrix partitioning techniques are investigated (Sections 4 and 5, respectively). The recently proposed hypergraph partitioning models [3,4,6,34] are used for load balancing and communication overhead minimization for both partitioning frameworks. Parallel algorithms for BSCM and PSCM are implemented for a message-passing multicomputer based on the above mentioned partitioning frameworks. The performance of the parallel implementations, the effects of different partitioning strategies on the parallel performance and on the speed of convergence, and the restoration abilities of the surrogate constraint methods are investigated experimentally in Section 6.

2 Background

2.1 Formulation of the problem

A general formulation of the image restoration problem with nonseparable, anisotropic, space variant and nonlocal distortions is given in [31], where the

image $g(r)$ recorded on the film of an image $f(\rho(r,t))$ is given by

$$g(r) = c \int_{t=0}^{T_r} f(\rho(r,t))\mathrm{d}t. \tag{2}$$

Here, t denotes time, T_r denotes the duration of the recording period, r denotes the position vector on the 2D image, c is a constant, $f(\rho(r,t))$ represents the observed (distorted) image, $\rho(r,t)$ represents the time varying, nonlinear distortion which can model the following types of motion:

(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 (arbitrary two dimensional motions and accelerations are possible);
(ii) *Isotropic Scaling:* $\rho(r,t) = r/\mathrm{m}(t)$, where $\mathrm{m}(t)$ is an arbitrary scaling function of time (by properly choosing $\mathrm{m}(t)$, 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 $R_{\phi(t)}$ is the 2×2 rotation matrix for $\phi(t)$ which is an arbitrary function of time representing the angle of rotation.

The image restoration problem consists of recovering f from g. Since, Eq. 2 represents a linear relation between g and f, it is possible to write it as

$$g(r) = \int_{r'} H(r,r')f(r')\mathrm{d}r', \tag{3}$$

where $H(r,r')$ represents the blurring system. The discrete counterpart of Eq. 3 is simply the linear system of equations $g = Hf$, where g and f are $mn\times 1$ vectors and H is an $mn\times mn$ matrix for an image of size $m \times n$.

Typically, there is a measurement error or noise associated with the observation g, leading to an inconsistent system of equations. Denoting the noisy observation by g', the problem can be expressed as a system of inequalities:

$$|(g' - Hf)_i| \leq \epsilon \qquad i = 1, \ldots, mn \tag{4}$$

where $(g' - Hf)_i$ is the ith component of $g' - Hf$, and ϵ is a suitable error tolerance parameter which is usually taken as a percentage of the mean value of g. For $i = 1, \ldots, mn$, $|(g' - Hf)_i| \leq \epsilon$ implies that

$$\begin{aligned} Hf_i &\leq \epsilon + g'_i \ \text{ if } g'_i \leq Hf_i \\ -Hf_i &\leq \epsilon - g'_i \ \text{ if } g'_i \geq Hf_i \ . \end{aligned} \tag{5}$$

Thus, the above system is converted to a linear feasibility problem of the form

$$\begin{array}{ccc} Ax \leq b & & \text{by setting} \\ A = \begin{bmatrix} H \\ -H \end{bmatrix}_{2mn \times mn}, & x = f_{mn \times 1}, & b = \begin{bmatrix} \varepsilon + g' \\ \varepsilon - g' \end{bmatrix}_{2mn \times 1} \end{array} \tag{6}$$

where ε is an $mn \times 1$ vector of ϵ's. For further details, the reader is referred to Chapter 5 of [29].

2.2 Parallel matrix-vector multiplies

2.2.1 Algorithms based on 1D matrix partitioning

Suppose that rows and columns of an $M \times N$ matrix H are permuted to yield a $K \times K$ block structure as

$$\begin{bmatrix} H_{11} & H_{12} & \cdots & H_{1K} \\ H_{21} & H_{22} & \cdots & H_{2K} \\ \vdots & \vdots & \ddots & \vdots \\ H_{K1} & H_{K2} & \cdots & H_{KK} \end{bmatrix} \tag{7}$$

for *rowwise* partitioning among K processors. Each processor P_k holds the kth row stripe $H_k = [H_{k1} \cdots H_{kK}]$ of size $m_k \times N$, where $\sum m_k = M$. The kth column stripe $[H_{1k}^T \cdots H_{Kk}^T]^T$ is of size $M \times n_k$, where $\sum n_k = N$.

In *row-parallel* $y \leftarrow Hx$ multiply, the y and x vectors are partitioned as $y = [y_1^T \cdots y_K^T]^T$ and $x = [x_1^T \cdots x_K^T]^T$, where processor P_k is responsible for computing subvector y_k of size m_k while holding subvector x_k of size n_k. In this setting, the common algorithm [13,33,34] executes the following steps at each processor P_k:

(1) For each nonzero off-diagonal block $H_{\ell k}$, send sparse vector $\hat{x}_k^\ell$ to processor P_ℓ, where $\hat{x}_k^\ell$ contains only those entries of x_k corresponding to the nonzero columns in $H_{\ell k}$. For each nonzero off-diagonal block $H_{k\ell}$, receive $\hat{x}_\ell^k$ from processor P_ℓ.
(2) Perform the local matrix-vector multiply $y_k \leftarrow H_k \times \tilde{x}_k$, where $\tilde{x}_k$ is the union of the local x_k vector and $\hat{x}_\ell^k$ subvectors received in step 1.

In step 1, P_k might be sending the same x_k-vector entry to different processors according to the sparsity pattern of the respective column of H. This multicast-like operation is referred to as *expand* operation.

In *column-parallel* $q \leftarrow H^T\pi$ multiply, processor P_k effectively stores $(H_k)^T = H_k^T$ which is the kth column stripe of H^T. The π and q vectors are partitioned as $\pi = [\pi_1^T \cdots \pi_K^T]^T$ and $q = [q_1^T \cdots q_K^T]^T$, where processor P_k is responsible for computing subvector q_k of size n_k while holding subvector π_k of size m_k. In this setting, each processor P_k executes the following steps:

(1) Perform the local matrix-vector multiply $q_k \leftarrow H_k^T\pi_k$.
(2) For each nonzero off-diagonal block $H_{\ell k}^T$, form sparse vector $\hat{q}_\ell^k$ which contains only those results of $q_\ell^k = H_{\ell k}^T \times \pi_k$ corresponding to the nonzero rows in $H_{\ell k}^T$. Send $\hat{q}_\ell^k$ to processor P_ℓ. For each nonzero off-diagonal block $H_{k\ell}^T$ receive partial result $\hat{q}_k^\ell$ from processor P_ℓ, and update $q_k \leftarrow q_k + \hat{q}_k^\ell$.

In step 2, the multinode accumulation performed on q_k-vector entries is referred to as *fold* operation.

2.2.2 Algorithms based on 2D matrix partitioning

Consider a 2D checkerboard partitioning on H for the computations of the form $y \leftarrow Hx$. In this partitioning scheme, the rows and the columns of matrix H are divided into R row stripes and C column stripes yielding an $R \times C$ block structure. This block structure generalizes the one given in Eq. 7, and can be mapped naturally onto a 2D mesh (R rows and C columns) of $K = R \times C$ processors. Therefore, the parallel system is considered as a logical 2D mesh [20]. In this setting, block $H_{k\ell}$ is assigned to processor $P_{k\ell}$. Note that nonzeros in any row (column) of matrix H are assigned to the processors in the same row (column) of the processor mesh.

For the sake of clarity, we define a two level partitioning on the vectors x and y. In the first level, the row and column stripes of the H matrix define R- and C-way partitions on the y and x vectors, respectively. In the second level, each x and y subvector is assumed to be further partitioned into R and C subsubvectors, respectively. For example, y_k denotes the kth subvector of y which contains subsubvector $y_{k\ell}$ for $\ell = 1, \ldots, C$. In a dual manner, x_ℓ denotes the ℓth subvector of x which contains $x_{k\ell}$ for $k = 1, \ldots, R$. A dual scheme is adopted in indexing the x and y subsubvectors so that each processor holds the subsubvectors of x and y with the same indices.

In 2D *row-column-parallel* $y \leftarrow Hx$ multiply, each processor $P_{k\ell}$ is responsible for computing the subsubvector $y_{k\ell}$ while holding subsubvector $x_{k\ell}$. In this setting, C row-parallel submatrix-vector multiplies algorithms are concurrently performed along the columns of the processor mesh to compute C partial result vectors y^ℓ for $\ell = 1, \ldots, C$, where $y = \sum_\ell y^\ell$. That is, for each $\ell = 1, \ldots, C$, all of the R processors in the ℓth column of the processor mesh execute the row-parallel algorithm for computing the submatrix-vector multiply $y^\ell \leftarrow H_{*\ell}\, x_\ell$, where $H_{*\ell}$ denotes the ℓth column stripe of matrix H in block

structure. At the end of this step, processor $P_{k\ell}$ holds the kth portion y_k^ℓ of y^ℓ. Then, R fold operations are concurrently executed along the rows of the processor mesh to compute $y \leftarrow \sum_\ell y^\ell$. That is, for each $k = 1, \ldots, R$, all of the C processors in the kth row of the processor mesh perform multinode accumulation on the y_k-subvector entries so that $P_{k\ell}$ ends up with the subsubvector $y_{k\ell}$. This last step effectively corresponds to step 2—fold communication step—of the column-parallel matrix-vector multiply algorithm $y_k \leftarrow H_{k*}x$, where H_{k*} denotes the kth row stripe of matrix H in block structure.

3 Surrogate constraint methods

The successive orthogonal projections method developed by Kacmarz (cited in [32]) successively projects iterate x^t at iteration t onto hyperplanes $a_i x^t = b_i$ corresponding to those inequalities violated by x^t, where a_i denotes the ith row of A. Surrogate constraint methods [35], on the other hand, derive surrogate hyperplanes from a set of violated constraints, and 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 [35], the *basic surrogate constraint method* (BSCM) derives surrogate hyperplanes from all of the violated constraints in the system, whereas the *sequential surrogate constraint method* (SSCM) and the *parallel surrogate constraint method* (PSCM) consider a subset of the constraints.

3.1 Basic surrogate constraint method (BSCM)

BSCM combines all of the violated constraints and makes just one projection at each iteration t: if the current point x^t violates the system $Ax \leq b$, a $1 \times M$ weight vector π is generated where $0 < \pi_i < 1$ if the ith constraint is violated (i.e., $a_i x^t > b_i$), and $\pi_i = 0$ otherwise. The π_i values are normalized so that $\sum_{i=1}^{M} \pi_i = 1$ [35]. Then, the surrogate constraint $(\pi A)x^t \leq (\pi b)$ is generated for which the corresponding surrogate hyperplane is $S_h = \{x : (\pi A)x^t = (\pi b)\}$. The next point x^{t+1} is obtained by projecting x^t onto S_h as

$$x^{t+1} = x^t - \lambda d^t \ , \quad \text{where} \quad d^t = \frac{\pi A x^t - \pi b}{||\pi A||^2}(\pi A)^T \ . \tag{8}$$

Here, d^t is the projection vector, and λ is a relaxation parameter that determines the location of the next point which is in the line segment joining the current point and its projection on the hyperplane. When $\lambda = 1$ the next point is the exact orthogonal projection of the current point. If $0 < \lambda < 1$, the

step taken is shorter (underrelaxation) and if $1 < \lambda < 2$, then the step taken is longer (overrelaxation) [8].

The selection of the weight vector π is an issue for the solution of the problem. Weights may be distributed equally among all violated constraints or they can be assigned in proportion to the amount of violations. We use a hybrid approach to compute π_i corresponding to the violated constraint i as

$$\pi_i = w_1 \frac{(a_i x^t - b_i)}{\sum_{j \in VC}(a_j x^t - b_j)} + \frac{w_2}{|VC|} \quad , \tag{9}$$

where w_1 and w_2 are two appropriate weights summing up to 1, and VC is the set of indices of the violated constraints at iteration t [29].

Verification of the convergence of the algorithm is based on the strict Fejer-monotonicity of the generated sequence $\{x^t\}_{t=0}^{\infty}$, i.e., for any feasible x and iteration t it is true that $\|x^{t+1} - x\| < \|x^t - x\|$. If the feasibility check allows 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 [35].

Figure 1 displays the pseudocode of BSCM applied to the system in Eq. 6. Since the system is composed of two copies of the same matrix with different signs, only the positive one 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 $q = \pi^+ H + \pi^-(-H) = (\pi^+ - \pi^-)H$, we form the vector $\pi \leftarrow \pi^+ - \pi^-$ and then perform the multiply πH. To compute $y^- \leftarrow -Hx$, simply $y \leftarrow Hx$ is negated. As a result the sparse matrix vector multiplies $-Hx$ and $\pi^-(-H)$ are avoided. Note that the original form of BSCM can be recovered by removing the operations involving vectors with a "$-$" superscript. In Fig. 1 and in the following pseudocodes, bold upper case letters denote matrices, bold lowercase letters denote column vectors of appropriate dimensions, plain lowercase letters denote scalars, and $\langle \cdot , \cdot \rangle$ denotes the inner product of two vectors.

The run time of a single iteration of BSCM applied to Eq. 6 is:

$$T_{BSCM} = (4Z + 13M + 5N)t_{flop}, \tag{10}$$

where Z denotes the number of nonzeros in matrix H, and t_{flop} denotes the time taken for a single scalar addition or multiplication operation.

while *true* do		
$\boldsymbol{y} \leftarrow \boldsymbol{Hx}$	▷ right multiply H with x	$\{2Zt_{flop}\}$
$\boldsymbol{\delta}^+ \leftarrow \boldsymbol{y} - \boldsymbol{b}^+$	▷ error of the upper system	$\{Mt_{flop}\}$
$\boldsymbol{\delta}^- \leftarrow -\boldsymbol{y} - \boldsymbol{b}^-$	▷ error of the lower system	$\{Mt_{flop}\}$
$\boldsymbol{\pi}^+ \leftarrow$ updatePi($\boldsymbol{\delta}^+$)	▷ update π^+ using Eq. 9	$\{3Mt_{flop}\}$
$\boldsymbol{\pi}^- \leftarrow$ updatePi($\boldsymbol{\delta}^-$)	▷ update π^- using Eq. 9	$\{3Mt_{flop}\}$
if $\boldsymbol{\pi}^+ = \boldsymbol{0}$ and $\boldsymbol{\pi}^- = \boldsymbol{0}$ then exit	▷ check convergence	
$\boldsymbol{\pi} \leftarrow \boldsymbol{\pi}^+ - \boldsymbol{\pi}^-$	▷ compute π	$\{Mt_{flop}\}$
$\boldsymbol{q} \leftarrow \boldsymbol{H^T \pi}$	▷ left multiply H with π	$\{2Zt_{flop}\}$
$\mu \leftarrow \langle \boldsymbol{\pi}^+, \boldsymbol{\delta}^+ \rangle + \langle \boldsymbol{\pi}^-, \boldsymbol{\delta}^- \rangle$	▷ sum of inner products	$\{4Mt_{flop}\}$
$\gamma \leftarrow \langle \boldsymbol{q}, \boldsymbol{q} \rangle$	▷ inner product	$\{2Nt_{flop}\}$
$\boldsymbol{d} \leftarrow \mu/\gamma \ \ \boldsymbol{q}$	▷ compute projection vector	$\{Nt_{flop}\}$
$\boldsymbol{x} \leftarrow \boldsymbol{x} - \lambda \, \boldsymbol{d}$	▷ update x	$\{2Nt_{flop}\}$

Fig. 1. BSCM applied to Eq. 6.

3.2 Sequential surrogate constraint method (SSCM)

Instead of working on the entire A matrix, SSCM partitions the system into subsystems and then solves the feasibility problem by applying the basic method on the subsystems in a cyclic order. Specifically, let matrix A be partitioned rowwise into K submatrices, and the right-hand side vector b be partitioned conformably into K subvectors, as follows:

$$A = \begin{bmatrix} A_1 \\ \vdots \\ A_k \\ \vdots \\ A_K \end{bmatrix}, b = \begin{bmatrix} b_1 \\ \vdots \\ b_k \\ \vdots \\ b_K \end{bmatrix}. \tag{11}$$

Here, A_k is an $m_k \times N$ submatrix having z_k nonzeros, b_k is an $m_k \times 1$ subvector, and $\sum_{k=1}^{K} m_k = M$ and $\sum_{k=1}^{K} z_k = Z$. Surrogate constraints are defined as $(\pi_k A_k)x \leq \pi_k b_k$ for each block, where π_k is of dimension m_k and is as defined above. SSCM proceeds by projecting the current iterate onto surrogate hyperplanes $(\pi_k A_k)x = \pi_k b_k$ successively for $k = 1, \ldots, K$ in cyclic order and increments t at each block. For each block, the next point is computed as:

$$x^{t+1} = x^t - \lambda d_k^t, \ , \quad \text{where} \quad d_k^t = \frac{\pi_k^t (A_k x^t - b_k)}{||\pi_k^t A_k||^2} (\pi_k^t A_k)^T. \tag{12}$$

Here, d_k^t is the projection vector of the kth block.

In SSCM each point x^t that is projected on block k is a result of the projection of x^{t-1} performed in the preceding block $k-1$. Thus, successive block

projections imply a dependency between the blocks of the system, causing the algorithm to be highly sequential.

The run time of SSCM applied to Eq. 6 is:

$$T_{SSCM} = \sum_{k=1}^{K} (4z_k + 13m_k + 5N)t_{flop} = (4Z + 13M + 5NK)t_{flop} \ . \qquad (13)$$

Note that each block brings an extra computation time of $5Nt_{flop}$.

3.3 Parallel surrogate constraint method (PSCM)

Yang and Murty [35] proposed PSCM as a coarse-grain parallel variant of SSCM which overcomes its serial nature. PSCM carries out simultaneous block projections and generates the next point as a convex combination of these projections. As in SSCM, A is divided rowwise into K contiguous blocks as shown in Eq. 11. At iteration t of PSCM, the next point x^{t+1} is computed as:

$$x^{t+1} = x^t - \lambda \sum_{k=1}^{K} \tau_k d_k^t \ . \qquad (14)$$

Here, τ_k are nonnegative numbers summing up to 1, and d_k^t is as defined in Eq. 12.

In Eq. 14, each projection has its own influence τ_k. This influence is taken into account to accelerate the convergence. 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 [29], no matter how τ_k is chosen, the progress of PSCM is much slower than that of SSCM. Actually this method is a variant of the Cimmino type algorithms which are known to suffer from slow convergence. To alleviate this problem, García-Palomares and Gonzales-Castaño [9] proposed an acceleration procedure for the Cimmino type algorithms by giving an improved step sizing rule. Later, Özaktaş et al. [30] used this rule in the parallel surrogate algorithm and generated the next point as follows:

$$x^{t+1} = x^t - \lambda \frac{\sum_{k=1}^{K} ||d_k^t||^2}{||\sum_{k=1}^{K} d_k^t||^2} \sum_{k=1}^{K} d_k^t, \qquad (15)$$

where d_k^t is defined as in Eq. 12. With this modification, the step sizes taken are enlarged so that the parallel method converges much more rapidly. Figure 2 displays PSCM applied to Eq. 6.

$\underline{\text{while}}$ *true* $\underline{\text{do}}$
  $\alpha \leftarrow 0$
  $\boldsymbol{d} \leftarrow \boldsymbol{0}$
  $\underline{\text{for}}$ $k = 1$ $\underline{\text{to}}$ K $\underline{\text{do}}$
    $\boldsymbol{y}_k \leftarrow \boldsymbol{H}_k \boldsymbol{x}$ ▷ right multiply H_k with x $\{2z_k t_{flop}\}$
    $\boldsymbol{\delta}_k^+ \leftarrow \boldsymbol{y}_k - \boldsymbol{b}_k$ ▷ error of the upper system $\{m_k t_{flop}\}$
    $\boldsymbol{\delta}_k^- \leftarrow -\boldsymbol{y}_k - \boldsymbol{b}_k^-$ ▷ error of the lower system $\{m_k t_{flop}\}$
    $\boldsymbol{\pi}_k^+ \leftarrow \text{updatePi}(\boldsymbol{\delta}_k^+)$ ▷ update π^+ using Eq. 9 $\{3m_k t_{flop}\}$
    $\boldsymbol{\pi}_k^- \leftarrow \text{updatePi}(\boldsymbol{\delta}_k^-)$ ▷ update π^- using Eq. 9 $\{3m_k t_{flop}\}$
    $\boldsymbol{\pi}_k \leftarrow \boldsymbol{\pi}_k^+ - \boldsymbol{\pi}_k^-$ ▷ compute π_k $\{m_k t_{flop}\}$
    $\boldsymbol{q}_k \leftarrow \boldsymbol{H}_k^T \boldsymbol{\pi}_k$ ▷ left multiply H_k with π $\{2z_k t_{flop}\}$
    $\mu_k \leftarrow \langle \boldsymbol{\delta}_k^+, \boldsymbol{\pi}_k^+ \rangle + \langle \boldsymbol{\delta}_k^-, \boldsymbol{\pi}_k^- \rangle$ ▷ sum of inner products $\{4m_k t_{flop}\}$
    $\gamma_k \leftarrow \langle \boldsymbol{q}_k, \boldsymbol{q}_k \rangle$ ▷ inner product $\{2Nt_{flop}\}$
    $\boldsymbol{d}_k \leftarrow \mu_k / \gamma_k \, \boldsymbol{q}_k$ ▷ compute projection vector $\{Nt_{flop}\}$
    $\alpha \leftarrow \alpha + \langle \boldsymbol{d}_k, \boldsymbol{d}_k \rangle$ ▷ inner product sum $\{2Nt_{flop}\}$
    $\boldsymbol{d} \leftarrow \boldsymbol{d} + \boldsymbol{d}_k$ ▷ sum the projection vectors $\{Nt_{flop}\}$
  $\underline{\text{if}}$ $\boldsymbol{\pi}_k^+ = \boldsymbol{0}$ $\underline{\text{and}}$ $\boldsymbol{\pi}_k^- = \boldsymbol{0}$ $\forall k$ $\underline{\text{then}}$ $\underline{\text{exit}}$ ▷ check convergence
  $\beta \leftarrow \langle \boldsymbol{d}, \boldsymbol{d} \rangle$ ▷ inner product $\{2Nt_{flop}\}$
  $\boldsymbol{x} \leftarrow \boldsymbol{x} - \lambda \, \alpha / \beta \, \boldsymbol{d}$ ▷ update x $\{2Nt_{flop}\}$

Fig. 2. PSCM applied to Eq. 6.

The run time of PSCM applied to Eq. 6 is:

$$T_{PSCM} = \sum_{k=1}^{K} (4z_k + 13m_k + 6N)t_{flop} + 4Nt_{flop} \tag{16}$$

$$= (4Z + 13M + 6NK + 4N)t_{flop}. \tag{17}$$

For SSCM and improved PSCM, as the number of blocks, K, increases, the number of iterations required for convergence is likely to decrease. However, each block brings an extra run time of $5Nt_{flop}$ and $6Nt_{flop}$ for SSCM and PSCM, respectively. Hence, the run times of SSCM and PSCM may increase by the increasing K, especially for highly sparse systems.

4 Parallelization based on 1D matrix partitioning

As seen in Figs. 1 and 2, repeated matrix-vector and matrix-transpose-vector multiplies of the forms $y \leftarrow Hx$ and $q \leftarrow H^T\pi$ constitute the computational kernels of both BSCM and PSCM. These methods also involve linear vector operations such as inner products and vector updates. In terms of the dependencies between matrix-vector multiplies and linear vector operations, the vectors can be classified as x-space and y-space vectors so that the linear vector operations occur only between the vectors that are in the same space. In

this setting, x, q, and d are x-space vectors, whereas y, b^+, b^-, π, π^+, π^-, δ^+, and δ^- are y-space vectors.

In 1D parallelization of BSCM, the matrix H may be partitioned rowwise or columnwise. However, since the PSCM formulation is based on rowwise blocking, it requires rowwise partitioning of matrix H. Thus, for the sake of simplicity, we assume a K-way rowwise partitioning of H for both methods, where K denotes the number of processors. A rowwise partition on H induces a columnwise partition on H^T. Therefore, the row-parallel algorithm is adopted for $y \leftarrow Hx$, and the column-parallel algorithm is adopted for $q \leftarrow H^T\pi$.

4.1 Basic surrogate constraint method

Parallel BSCM based on 1D partitioning executes the steps given in Fig. 3 at each processor P_k. In this figure and the following figures, subscripts are used to denote the subvectors and submatrices that are stored locally in a processor, whereas superscripts are used to denote the partial results computed by a processor. For example, H_k denotes the kth row stripe of the global H matrix stored by the processor P_k; x_k denotes the kth stripe of the global x vector maintained by processor P_k; $\gamma^k \leftarrow \langle q_k, q_k \rangle$ denotes partial inner product result for the global inner product $\gamma \leftarrow \langle q, q \rangle$ computed by processor P_k. The global communication operators *globalAnd* and *globalSum* combine the input arguments of each processor using the operations *and* and *sum* respectively, and distribute the result back to all processors. As seen in Fig. 3, scalars μ and γ are reduced together in order to amortize the latency overhead in the reduction operations.

4.2 Parallel surrogate constraint method

Let the number of blocks, K, be equivalent to the number of processors, that is each processor is given a single block. Then, parallel PSCM executes the steps given in Fig. 4 at each processor P_k. Since the resulting vector q for $q \leftarrow H^T\pi$ multiply need not to be constructed, the parallelization of PSCM does not necessitate the execution of the column-parallel multiply algorithm as a whole. As seen in the figure, the local projection vector d^k is obtained from the local intermediate vector q^k through linear vector operations, and then the sum of the local projection vectors is computed through a fold operation on these local d^k vectors. This fold operation on the local d^k vectors can be considered as the delayed execution of the fold operation in step 2 of the column-parallel multiply algorithm. Note that this fold operation also provides each processor P_k with the subvector d_k needed to update x_k. The scalar α is obtained through reducing the inner products of local projection vectors

while *true* do		
$\boldsymbol{x} \leftarrow \text{expand}(\boldsymbol{x}_k)$	▷ expand x_k vector	$\{t_{expand}\}$
$\boldsymbol{y}_k \leftarrow \boldsymbol{H}_k\, \boldsymbol{x}$	▷ multiply H_k from right	$\{2z_k t_{flop}\}$
$\boldsymbol{\delta}_k^+ \leftarrow \boldsymbol{y}_k - \boldsymbol{b}_k^+$	▷ error of the upper system	$\{m_k t_{flop}\}$
$\boldsymbol{\delta}_k^- \leftarrow -\boldsymbol{y}_k - \boldsymbol{b}_k^-$	▷ error of the lower system	$\{m_k t_{flop}\}$
$\boldsymbol{\pi}_k^+ \leftarrow \text{updatePi}(\boldsymbol{\delta}_k^+)$	▷ update π^+ using Eq. 9	$\{3m_k t_{flop}\}$
$\boldsymbol{\pi}_k^- \leftarrow \text{updatePi}(\boldsymbol{\delta}_k^-)$	▷ update π^- using Eq. 9	$\{3m_k t_{flop}\}$
if $\boldsymbol{\pi}_k^+ = \boldsymbol{0}$ and $\boldsymbol{\pi}_k^- = \boldsymbol{0}$ then	▷ check convergence	
$f^k \leftarrow true$	▷ block k feasible	
else $f^k \leftarrow false$	▷ block k not feasible	
$f \leftarrow \text{globalAnd}(f^k)$	▷ check the whole system	$\{(t_s + t_w)\log K\}$
if $f = true$ then exit		
$\boldsymbol{\pi}_k \leftarrow \boldsymbol{\pi}_k^+ - \boldsymbol{\pi}_k^-$	▷ compute π_k	$\{m_k t_{flop}\}$
$\boldsymbol{q}^k \leftarrow \boldsymbol{H}_k^T \boldsymbol{\pi}_k$	▷ multiply H_k from left	$\{2z_k t_{flop}\}$
$\boldsymbol{q}_k \leftarrow \text{fold}(\boldsymbol{q}^k)$	▷ fold q vector	$\{t_{fold}\}$
$\mu^k \leftarrow \langle \boldsymbol{\delta}_k^+, \boldsymbol{\pi}_k^+ \rangle + \langle \boldsymbol{\delta}_k^-, \boldsymbol{\pi}_k^- \rangle$	▷ sum of inner products	$\{4m_k t_{flop}\}$
$\gamma^k \leftarrow \langle \boldsymbol{q}_k, \boldsymbol{q}_k \rangle$	▷ inner product	$\{2n_k t_{flop}\}$
$(\mu, \gamma) \leftarrow \text{globalSum}(\mu^k, \gamma^k)$	▷ form μ and γ	$\{(t_s + 2t_w)\log K\}$
$\boldsymbol{d}_k \leftarrow \mu/\gamma\, \boldsymbol{q}_k$	▷ form projection vector	$\{n_k t_{flop}\}$
$\boldsymbol{x}_k \leftarrow x_k - \lambda \boldsymbol{d}_k$	▷ update my portion of x	$\{2n_k t_{flop}\}$

Fig. 3. Parallel BSCM based on 1D partitioning of the H matrix.

$\alpha^k = \|d^k\|^2$ using the *globalSum* operator. The scalar β is obtained through computing the inner product of global projection vector d. Computing β in parallel requires reducing the $\beta^k = \|d_k\|^2$ values. The scalars α and β are reduced together for efficiency issues as in parallel BSCM.

4.3 Load balancing and communication-overhead minimization

Sparse matrix partitioning for parallel matrix-vector multiplies of the form $y \leftarrow Hx$ is formulated in terms of the graph [17] and hypergraph partitioning problems [4]. Both of these problems are NP-complete [10,23]. We use the hypergraph-based formulation for two reasons. First, the objective in hypergraph-based formulation is an exact measure of the total communication volume. Second, the matrices in our methods are unsymmetric; standard graph partitioning is not readily applicable [4,13,14].

We use the column-net hypergraph model of Çatalyürek and Aykanat [4] to obtain a K-way rowwise partition on matrix H. The rowwise partition on H induces a columnwise partition on H^T. As is known [13,34], the communication requirements of the row-parallel $y \leftarrow Hx$ and the column-parallel $q \leftarrow H^T\pi$ multiplies are the same in terms of total volume and number of messages. Therefore, the above partitioning enables efficient multiplies with H^T as well.

while $true$ do		
$\boldsymbol{x} \leftarrow \text{expand}(\boldsymbol{x}_k)$	▷ expand x_k vector	$\{t_{expand}\}$
$\boldsymbol{y}_k \leftarrow \boldsymbol{H}_k \boldsymbol{x}$	▷ multiply H_k from right	$\{2z_k t_{flop}\}$
$\boldsymbol{\delta}_k^+ \leftarrow \boldsymbol{y}_k - \boldsymbol{b}_k^+$	▷ error of the upper system	$\{m_k t_{flop}\}$
$\boldsymbol{\delta}_k^- \leftarrow -\boldsymbol{y}_k - \boldsymbol{b}_k^-$	▷ error of the lower system	$\{m_k t_{flop}\}$
$\boldsymbol{\pi}_k^+ \leftarrow \text{updatePi}(\boldsymbol{\delta}_k^+)$	▷ update π^+ using Eq. 9	$\{3m_k t_{flop}\}$
$\boldsymbol{\pi}_k^- \leftarrow \text{updatePi}(\boldsymbol{\delta}_k^-)$	▷ update π^- using Eq. 9	$\{3m_k t_{flop}\}$
if $\boldsymbol{\pi}_k^+ = \boldsymbol{0}$ and $\boldsymbol{\pi}_k^- = \boldsymbol{0}$ then	▷ check convergence	
$f^k \leftarrow true$	▷ block k feasible	
else $f^k \leftarrow false$	▷ block k not feasible	
$f \leftarrow \text{globalAnd}(f^k)$	▷ check the whole system	$\{(t_s + t_w) \log K\}$
if $f = true$ then exit		
$\boldsymbol{\pi}_k \leftarrow \boldsymbol{\pi}_k^+ - \boldsymbol{\pi}_k^-$	▷ compute π_k	$\{m_k t_{flop}\}$
$\boldsymbol{q}^k \leftarrow \boldsymbol{H}_k^T \boldsymbol{\pi}_k$	▷ multiply H_k from left	$\{2z_k t_{flop}\}$
$\mu^k \leftarrow \langle \boldsymbol{\delta}_k^+, \boldsymbol{\pi}_k^+ \rangle + \langle \boldsymbol{\delta}_k^-, \boldsymbol{\pi}_k^- \rangle$	▷ sum of inner products	$\{4m_k t_{flop}\}$
$\gamma^k \leftarrow \langle \boldsymbol{q}^k, \boldsymbol{q}^k \rangle$	▷ inner product	$\{2N t_{flop}\}$
$\boldsymbol{d}^k \leftarrow \mu^k / \gamma^k \, \boldsymbol{q}^k$	▷ form local projection vector	$\{N t_{flop}\}$
$\alpha^k \leftarrow \langle \boldsymbol{d}^k, \boldsymbol{d}^k \rangle$	▷ inner product	$\{2N t_{flop}\}$
$\boldsymbol{d}_k \leftarrow \text{fold}(\boldsymbol{d}^k)$	▷ fold d vector	$\{t_{fold}\}$
$\beta^k \leftarrow \langle \boldsymbol{d}_k, \boldsymbol{d}_k \rangle$	▷ inner product	$\{2n_k t_{flop}\}$
$(\alpha, \beta) \leftarrow \text{globalSum}(\alpha^k, \beta^k)$	▷ form α and β	$\{(t_s + 2t_w) \log K\}$
$\boldsymbol{x}_k \leftarrow \boldsymbol{x}_k - \lambda \alpha / \beta \boldsymbol{d}_k$	▷ update my portion of x	$\{2n_k t_{flop}\}$

Fig. 4. Parallel PSCM based on 1D partitioning of the H matrix.

As mentioned earlier, the x- and y-space vectors do not undergo linear vector operations. Hence, an unsymmetric partitioning on H is allowable. We use the techniques discussed in [34] to exploit this freedom in order to minimize the communication overhead due to the total number of messages, maximum volume and number of messages handled by a single processor as well as the total volume of messages.

In parallelizing BSCM and PSCM, we consider balancing the computational loads of the processors only for the matrix-vector and matrix-transpose-vector multiplies. The loads of the processors during linear vector operations are determined by the partitioning on H. Therefore, imbalances in processors' loads may occur during these operations. Obtaining balance in the vector operations is possible within multi-constraint partitioning framework [5,18,19]. In this work, we omit obtaining computational balance during linear vector operations for two reasons. First, imbalances in linear vector operations are tolerable, because these operations are not costly. Second, multi-constraint formulation restricts the search space in a hypergraph partitioning problem; this restriction may lead to a higher communication cost.

while *true* do		
$\boldsymbol{x}_\ell \leftarrow \text{colExpand}(\boldsymbol{x}_{k\ell})$	▷ expand $x_{k\ell}$ in column mesh	$\{t_{colexpand}\}$
$\boldsymbol{y}_k^\ell \leftarrow \boldsymbol{H}_{k\ell}\boldsymbol{x}_\ell$	▷ multiply $H_{k\ell}$ from right	$\{2zt_{flop}\}$
$\boldsymbol{y}_{k\ell} \leftarrow \text{rowFold}(\boldsymbol{y}_k^\ell)$	▷ fold y_k^ℓ along the row mesh	$\{t_{rowfold}\}$
$\boldsymbol{\delta}_{k\ell}^+ \leftarrow \boldsymbol{y}_{k\ell} - \boldsymbol{b}_k^+$	▷ error of the upper system	$\{m_k t_{flop}\}$
$\boldsymbol{\delta}_{k\ell}^- \leftarrow -\boldsymbol{y}_{k\ell} - \boldsymbol{b}_k^-$	▷ error of the lower system	$\{m_k t_{flop}\}$
$\boldsymbol{\pi}_{k\ell}^+ \leftarrow \text{updatePi}(\boldsymbol{\delta}_{k\ell}^+)$	▷ update π^+ using Eq. 9	$\{3m_k t_{flop}\}$
$\boldsymbol{\pi}_{k\ell}^- \leftarrow \text{updatePi}(\boldsymbol{\delta}_{k\ell}^-)$	▷ update π^- using Eq. 9	$\{3m_k t_{flop}\}$
if $\boldsymbol{\pi}_{k\ell}^+ = \mathbf{0}$ and $\boldsymbol{\pi}_{k\ell}^- = \mathbf{0}$ then	▷ check convergence	
$f^k \leftarrow true$	▷ block k feasible	
else $f^k \leftarrow false$	▷ block k not feasible	
$f \leftarrow \text{globalAnd}(f^k)$	▷ check the whole system	$\{(t_s + t_w)\log K\}$
if $f = true$ then exit		
$\boldsymbol{\pi}_{k\ell} \leftarrow \boldsymbol{\pi}_{k\ell}^+ - \boldsymbol{\pi}_{k\ell}^-$	▷ compute $\pi_{k\ell}$	$\{m_k t_{flop}\}$
$\boldsymbol{\pi}_k \leftarrow \text{rowExpand}(\boldsymbol{\pi}_{k\ell})$	▷ expand $\pi_{k\ell}$ in row mesh	$\{t_{rowexpand}\}$
$\boldsymbol{q}_{\boldsymbol{\ell}}^{\boldsymbol{k}} \leftarrow \boldsymbol{H}_{k\ell}^T\boldsymbol{\pi}_k$	▷ multiply $H_{k\ell}$ from left	$\{2zt_{flop}\}$
$\boldsymbol{q}_{\ell k} \leftarrow \text{colFold}(\boldsymbol{q}_\ell^k)$	▷ fold q_ℓ^k in column mesh	$\{t_{colfold}\}$
$\mu^{k\ell} \leftarrow \langle\boldsymbol{\pi}_{k\ell}^+, \boldsymbol{\delta}_{k\ell}^+\rangle + \langle\boldsymbol{\pi}_{k\ell}^-, \boldsymbol{\delta}_{k\ell}^-\rangle$	▷ sum of inner products	$\{4m_k t_{flop}\}$
$\gamma^{k\ell} \leftarrow \langle\boldsymbol{q}_{\ell k}, \boldsymbol{q}_{\ell k}\rangle$	▷ inner product	$\{2n_k t_{flop}\}$
$(\mu, \gamma) \leftarrow \text{globalSum}(\mu^{k\ell}, \gamma^{k\ell})$	▷ form μ and γ	$\{(t_s + 2t_w)\log K\}$
$\boldsymbol{d}_k \leftarrow \mu/\gamma\boldsymbol{q}_{k\ell}$	▷ form projection vector	$\{n_k t_{flop}\}$
$\boldsymbol{x}_{k\ell} \leftarrow \boldsymbol{x}_{k\ell} - \lambda\boldsymbol{d}_{k\ell}$	▷ update my portion of x	$\{2n_k t_{flop}\}$

Fig. 5. Parallel BSCM based on 2D checkerboard partitioning of the H matrix.

5 Parallelization based on 2D matrix partitioning

The row-column-parallel algorithm is used for both the matrix-vector multiply $y \leftarrow Hx$ and matrix-transpose-vector multiply $q \leftarrow H^T\pi$. There is a duality between the interprocessor communication patterns of $y \leftarrow Hx$ and $q \leftarrow H^T\pi$ multiplies. The communication pattern of the expand operation in $q \leftarrow H^T\pi$ is exactly the same as that of the fold operation in $y \leftarrow Hx$ and vice versa.

5.1 Basic surrogate constraint method

Parallel BSCM based on checkerboard partitioning executes the steps given in Fig. 5 at each processor $P_{k\ell}$. We use the same convention on the usage of subscripts and superscripts. This time, however, we use two indices $k\ell$ to designate processor $P_{k\ell}$'s data. The single indices k or ℓ are used for the results pertaining to the kth row-stripe or ℓth column-stripe respectively. Recall that the communication operations required for matrix-vector multiplies are confined to the rows or columns of the processor mesh. Only the scalars μ and γ are obtained via a global sum operation among all processors.

while *true* do		
$\boldsymbol{x}_\ell \leftarrow \text{colExpand}(\boldsymbol{x}_{k\ell})$	▷ expand $x_{k\ell}$ in column mesh	$\{t_{colexpand}\}$
$\boldsymbol{y}_k^\ell \leftarrow \boldsymbol{H}_{k\ell}\boldsymbol{x}_\ell$	▷ multiply $H_{k\ell}$ from right	$\{2zt_{flop}\}$
$\boldsymbol{y}_{k\ell} \leftarrow \text{rowFold}(\boldsymbol{y}_k^\ell)$	▷ fold y_k^ℓ along the row mesh	$\{t_{rowfold}\}$
$\boldsymbol{\delta}_{k\ell}^+ \leftarrow \boldsymbol{y}_{k\ell} - \boldsymbol{b}_k^+$	▷ error of the upper system	$\{m_k t_{flop}\}$
$\boldsymbol{\delta}_{k\ell}^- \leftarrow -\boldsymbol{y}_{k\ell} - \boldsymbol{b}_k^-$	▷ error of the lower system	$\{m_k t_{flop}\}$
$\boldsymbol{\pi}_{k\ell}^+ \leftarrow \text{updatePi}(\boldsymbol{\delta}_{k\ell}^+)$	▷ update π^+ using Eq. 9	$\{3m_k t_{flop}\}$
$\boldsymbol{\pi}_{k\ell}^- \leftarrow \text{updatePi}(\boldsymbol{\delta}_{k\ell}^-)$	▷ update π^- using Eq. 9	$\{3m_k t_{flop}\}$
if $\boldsymbol{\pi}_{k\ell}^+ = \mathbf{0}$ and $\boldsymbol{\pi}_{k\ell}^- = \mathbf{0}$ then	▷ check convergence	
$f^k \leftarrow true$	▷ block k feasible	
else $f^k \leftarrow false$	▷ block k not feasible	
$f \leftarrow \text{globalAnd}(f^k)$	▷ check the whole system	$\{(t_s + t_w)\log K\}$
if $f = true$ then exit		
$\boldsymbol{\pi}_{k\ell} \leftarrow \boldsymbol{\pi}_{k\ell}^+ - \boldsymbol{\pi}_{k\ell}^-$	▷ compute $\pi_{k\ell}$	$\{m_k t_{flop}\}$
$\boldsymbol{\pi}_k \leftarrow \text{rowExpand}(\boldsymbol{\pi}_{k\ell})$	▷ expand $\pi_{k\ell}$ in row mesh	$\{t_{rowexpand}\}$
$\boldsymbol{q}_\ell^k \leftarrow \boldsymbol{H}_{k\ell}^T\boldsymbol{\pi}_k$	▷ multiply $H_{k\ell}$ from left	$\{2zt_{flop}\}$
$\mu^{k\ell} \leftarrow \langle\boldsymbol{\pi}_{k\ell}^+, \boldsymbol{\delta}_{k\ell}^+\rangle + \langle\boldsymbol{\pi}_{k\ell}^-, \boldsymbol{\delta}_{k\ell}^-\rangle$	▷ sum of inner products	$\{2m_k t_{flop}\}$
$\gamma^{k\ell} \leftarrow \langle\boldsymbol{q}_\ell^k, \boldsymbol{q}_\ell^k\rangle$	▷ inner product	$\{2nt_{flop}\}$
$(\mu^k, \gamma^k) \leftarrow \text{rowSum}(\mu^{k\ell}, \gamma^{k\ell})$	▷ form μ^k and γ^k	$\{(t_s + 2t_w)\log C\}$
$\boldsymbol{d}_\ell^k \leftarrow \mu^k/\gamma^k\boldsymbol{q}_\ell^k$	▷ form projection vector	$\{nt_{flop}\}$
$\alpha^{k\ell} \leftarrow \langle\boldsymbol{d}_\ell^k, \boldsymbol{d}_\ell^k\rangle$	▷ inner product	$\{2nt_{flop}\}$
$\boldsymbol{d}_{k\ell} \leftarrow \text{colFold}(\boldsymbol{d}_\ell^k)$	▷ fold d_ℓ^k in column mesh	$\{t_{colfold}\}$
$\beta^{k\ell} \leftarrow \langle\boldsymbol{d}_{k\ell}, \boldsymbol{d}_{k\ell}\rangle$	▷ inner product	$\{2n_k t_{flop}\}$
$(\alpha, \beta) \leftarrow \text{globalSum}(\alpha^{k\ell}, \beta^{k\ell})$	▷ form α and β	$\{(t_s + 2t_w)\log K\}$
$\boldsymbol{x}_{k\ell} \leftarrow \boldsymbol{x}_{k\ell} - \lambda\alpha/\beta\boldsymbol{d}_{k\ell}$	▷ update my portion of x	$\{2n_k t_{flop}\}$

Fig. 6. Parallel PSCM based on 2D checkerboard partitioning of the H matrix.

5.2 Parallel surrogate constraint method

Different from the 1D partitioning, the number of row blocks for PSCM is R not K. Parallel PSCM based on checkerboard partitioning executes the steps given in Fig. 6 at each processor $P_{k\ell}$. As seen in Fig. 6, the 2D implementation of PSCM is similar to the 2D implementation of BSCM. However, in order to calculate the projection vector of the kth row-stripe, local $\mu^{k\ell}$ and $\gamma^{k\ell}$ scalars are computed and are added up in the kth row of the processor mesh. Since $\alpha = \sum_{k=1}^{R}\sum_{\ell=1}^{C}\alpha^{k\ell}$ and $\beta = \sum_{k=1}^{R}\sum_{\ell=1}^{C}\beta^{k\ell}$, a global sum operation is required to apply the step sizing rule.

5.3 Load balancing and communication-overhead minimization

A number of different techniques for checkerboard partitioning of sparse matrices are given in [6,15,24,25]. Among these, only the hypergraph partitioning

model of Çatalyürek and Aykanat [6] exploits sparsity to reduce communication cost. Therefore, we use hypergraph partitioning model to obtain an $R \times C$ checkerboard partition on the matrix H.

Since the communication operations are confined to the rows and columns of the processor mesh, the maximum number of messages handled by a single processor for a parallel system with $K = R \times C$ processors is at most $R+C-2$ (compare to $K-1$ in 1D). Therefore, the checkerboard partitioning method leads to reduced communication latency overhead without explicit effort.

The row-column-parallel matrix-vector multiply algorithm given in Section 2.2.2 uses point-to-point (personalized) communication scheme for the expand and fold operations. Since the expand and fold operations are confined to a smaller number of processors in 2D checkerboard partitioning, all-to-all communication schemes are viable, i.e., an *all-to-all broadcast* and a *multinode accumulation* can be performed for the expand and fold operations. These algorithms can be implemented using hypercube algorithms to reduce the maximum number of messages handled by a processor to $\lceil \log R \rceil + \lceil \log C \rceil \approx \lceil \log K \rceil$. The 2D hypergraph model [6] can be extended to handle the minimization of communication volume overhead in this all-to-all communication scheme [2].

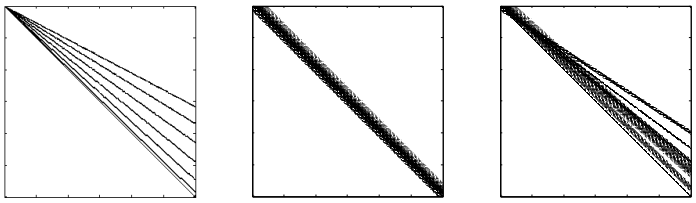
6 Results

The parallel performance and the restoration abilities of the surrogate constraint methods are evaluated experimentally. Parallel performance analysis is first carried on an iteration basis. This approach shows the amount of gain achieved by the proposed parallelization schemes. Then, overall performance results and examples of blurred and restored images are given.

The experiments were carried out on a Beowulf Cluster equipped with 400 MHz Intel Pentium II processors with 512KB cache size and 128MB RAM. The operating system is Debian GNU/Linux 3.0 distribution with Linux kernel 2.4.14. The interconnection network is comprised of a 3COM SuperStack II 3900 managed switch connected to Intel Ethernet Pro 100 Fast Ethernet network interface cards at each node. The parallel algorithms were implemented using LAM/MPI 6.5.6 [1].

6.1 Experimental setup

Three types of blurs were used for the construction of the distorted images. In all of the blurs, the record time was set as 3.5 seconds and the movement



(a) iso150x200 (b)rot150x200 (c) irt150x200

Fig. 7. Sparsity patterns of the H matrices corresponding to the three types of blur.

of the object is sampled at 0.5 second intervals. The first type of blur models isotropic scaling. The x-axis and y-axis coordinates of the function ρ were chosen as $x/(1 + 0.1s^2)$ and $y/(1 + 0.1s^2)$. The second blur is a result of rotation motion. The object was rotated clockwise 6 degrees per second for the first 1.5 seconds and then was rotated counter-clockwise 4 degrees per second for the remaining 2 seconds. The third blur denotes a combined effect of translational motion, isotropic scaling, and rotation. In the translational motion, the object accelerates with $0.5m/s^2$ in the x direction for the first 1.5 seconds and then it turns back with a constant speed of $1.0m/s$, and moves along the y direction with a constant speed of $1.0m/s$ throughout the recording time. Isotropic scaling and rotation effects of this blur are the same as those of the first and second blurs, respectively.

Using these blurring effects, three different images of 150×200, 300×400, and 600×800 pixels were produced with zero boundary condition, i.e., pixels outside the borders of the images are black [28]. Noise was simulated by adding %1 normally distributed zero mean random variables with a standard deviation of one, e.g., normally distributed random noise scaled such that 2-norm of the noise is 0.01 times the 2-norm of the blurred image. This type of noise is typical in similar works (see [26–28] and the references therein). Table 1 and Figure 7 display the properties and the sparsity patterns of the resulting H matrices. The prefixes "iso", "rot", and "irt" are respectively used to denote the isotropic, rotation, and combined (isotropic + rotation + translational) blurs.

The hypergraph partitioning tool PaToH [5] was used with the default parameters to obtain the desired 1D and 2D partitionings. Since PaToH incorporates randomized algorithms, it was run 10 times starting from different seeds for every partitioning instance. In all partitioning instances, the observed imbalance ratios were below 5%. Averages of the parallel performance and convergence results obtained from these runs are displayed in the following tables and bar charts. In all tables and figures, "P2P" and "A2A" refer to the point-to-point and all-to-all communication schemes, respectively. In 2D partitionings, the number of processors in rows and columns of the processor mesh are not restricted to be powers of two. Therefore, all-to-all communication primitives

Table 1
Properties of the H matrices

H matrix	number of rows/cols	number of nonzeros					
		total	per				
			row/col	row		col	
			avg	min	max	min	max
iso150x200	30000	209377	6.98	1	16	1	7
iso300x400	120000	839377	6.99	1	16	1	7
iso600x800	480000	3359377	7.00	1	16	1	7
rot150x200	30000	168775	5.63	1	8	1	6
rot300x400	120000	681907	5.68	1	8	1	6
rot600x800	480000	2734319	5.70	1	8	1	6
irt150x200	30000	205633	6.85	1	19	3	7
irt300x400	120000	823661	6.86	1	19	3	7
irt600x800	480000	3294639	6.86	1	19	3	7

needed in fold and expand operations are implemented using the all-to-all broadcast algorithm proposed by Jacunski et al. [16].

6.2 Per iteration performance

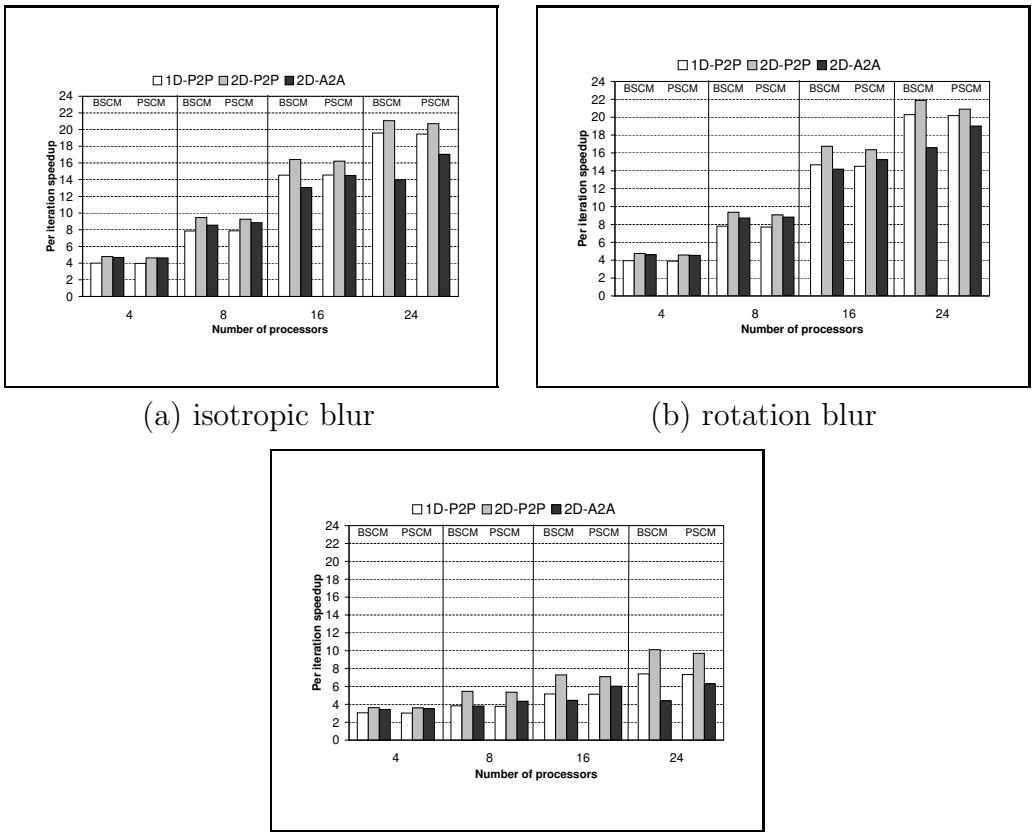
Table 2 displays per iteration run times of the parallel implementations of BSCM and PSCM, and the serial run time of BSCM. The bottom of the table displays the speedup values averaged over all instances for each possible number of processors. The per iteration run time of BSCM is taken as the sequential run time in calculating the speedups. The bar charts for the individual speedup values are displayed in Fig. 8 for 300×400 images.

As seen in Table 2, the 2D-P2P scheme leads to better performance than the other two schemes. In particular, the 2D-P2P scheme leads to faster execution in 34 and 31 instances out of 36 instances for parallel BSCM and parallel PSCM, respectively. The 1D-P2P scheme obtains faster execution times in only 5 instances for both BSCM and PSCM. The 2D-A2A scheme is the fastest only in 4-way parallelization of PSCM for iso600x800 and irt600x800. As seen in Table 2, BSCM and PSCM display comparable parallel performance, where BSCM performs slightly better on the average. This performance difference slightly increases in favor of parallel BSCM in 2D partitioning. These experimental findings were expected because PSCM incurs extra computation as discussed in Section 3. Moreover, parallel PSCM requires an extra communication along the rows of processor mesh for computing the μ^k and γ^k values in 2D partitioning as seen in Fig. 6.

As seen in Table 2 and Fig. 8, among the blur types used, the data sets produced by the isotropic and rotation blur lead to comparable speedup values,

Table 2
Per iteration execution times of parallel BSCM and PSCM (in msecs)

H matrix	Sequential BSCM	K	Parallel BSCM 1D P2P	Parallel BSCM 2D P2P	Parallel BSCM 2D A2A	Parallel PSCM 1D P2P	Parallel PSCM 2D P2P	Parallel PSCM 2D A2A
iso150x200	75.1	4	19.7	15.9	17.1	19.7	16.5	17.3
		8	11.3	9.8	11.2	11.2	10.1	10.5
		16	7.3	6.9	9.6	7.1	7.1	8.0
		24	6.0	6.3	10.5	6.0	6.5	8.0
iso300x400	324.1	4	81.1	67.7	69.4	81.8	70.2	70.1
		8	41.3	34.2	37.9	41.2	35.0	36.6
		16	22.3	19.7	24.8	22.3	20.0	22.3
		24	16.5	15.4	23.2	16.7	15.6	19.0
iso600x800	1430.1	4	335.4	281.8	288.8	343.4	294.9	295.1
		8	171.8	141.5	149.0	175.0	146.8	149.3
		16	86.5	73.5	85.1	87.6	76.5	81.3
		24	60.1	51.6	64.7	60.5	52.8	59.0
rot150x200	71.2	4	17.9	14.6	15.6	18.0	15.4	16.0
		8	10.0	8.7	9.8	10.1	9.0	9.5
		16	6.3	6.1	8.0	6.4	6.3	6.9
		24	5.3	5.7	8.5	5.3	6.0	6.7
rot300x400	299.5	4	75.9	63.0	65.1	76.9	65.4	66.0
		8	38.4	32.0	34.3	38.9	33.0	34.0
		16	20.4	17.9	21.1	20.6	18.3	19.6
		24	14.8	13.7	18.1	14.8	14.3	15.8
rot600x800	1319.3	4	320.8	255.9	269.2	329.7	269.6	275.0
		8	164.7	132.5	138.7	169.1	138.0	140.9
		16	85.1	69.1	74.9	86.2	71.9	74.3
		24	56.2	47.8	56.1	56.7	49.6	53.1
irt150x200	75.6	4	25.2	20.7	22.4	25.3	21.1	21.9
		8	17.3	15.1	23.0	17.5	15.5	18.3
		16	12.8	11.7	21.6	13.0	12.2	15.2
		24	10.9	9.8	23.0	10.9	10.2	15.7
irt300x400	325.1	4	106.3	89.5	94.7	107.5	90.1	92.5
		8	84.8	59.6	85.4	86.1	60.7	74.8
		16	63.0	44.6	73.0	63.3	45.8	53.9
		24	43.9	32.1	74.1	44.3	33.5	51.6
irt600x800	1419.7	4	508.6	407.8	412.8	518.5	418.5	416.3
		8	351.7	291.1	383.0	356.8	294.7	363.8
		16	274.7	171.1	301.4	277.3	171.0	259.8
		24	246.5	130.7	274.6	245.8	129.7	224.4
Average speedup		4	3.7	4.5	4.3	3.6	4.3	4.2
		8	6.4	7.8	6.8	6.4	7.6	7.1
		16	11.0	12.9	10.1	11.0	12.6	11.2
		24	14.8	16.8	11.6	14.7	16.3	13.6



(a) isotropic blur (b) rotation blur

(c) combined blur

Fig. 8. Per iteration speedup charts of BSCM and PSCM for the images of size 300×400.

whereas the data sets produced by combined blur lead to inferior parallel performance. This phenomenon can be attributed to the absence of columns with only one nonzero in "irt" matrices, i.e., these matrices are likely to yield harder partitioning instances in terms of communication overhead minimization.

Tables 3 and 4 are presented in order to further investigate the effect of the matrix partitioning schemes on the parallel performance of BSCM and PSCM. Table 3 displays the communication patterns in the parallel matrix-vector and matrix-transpose-vector multiplies obtained by the 1D and 2D partitioning schemes for the images of size 300×400. Table 4 shows the dissection of the communication patterns into expand and fold phases for 2D schemes.

In Tables 3 and 4, message-volume values are given in terms of the words communicated. In terms of the total communication volume, the 1D partitioning scheme produces the best partitions in 7 out of 12 instances, whereas the 2D-P2P scheme produces the best partitions in 8-, 16-, and 24-way partitionings of irt300x400, and the 2D-A2A scheme produces best partitions in 4-way partitioning of iso300x400 and irt300x400. This experimental outcome may be due to the fact that 1D rowwise partitioning disturbs only column

coherence, whereas the 2D partitioning schemes disturb both row and column coherences. In terms of the total number of messages, 1D scheme competes with the 2D-A2A scheme, where 2D-P2P displays the worst performance. The 1D and 2D-A2A schemes produce, respectively, the best partitions in 9 and 3 instances out of 12 instances. The performance of the 1D partitioning scheme relies on the enhancement given in [34]. The relatively better performance of 2D-A2A is expected as discussed in Section 5.3. Note that the number of messages in 2D-A2A is always the same for a given number of processors because of the regular communication operations. In terms of maximum message volume, the 1D scheme produces best results in all partitioning instances of rot300x400 and 8-, 16-, and 24-way partitioning of iso300x400, whereas 2D-P2P produces best results in all partitioning instances of irt300x400 and 4-way partitioning of iso300x400. This relatively better performance of the 1D scheme can also be attributed to the enhancement [34] which involves explicit effort towards balancing the communication-volume loads of processors. In terms of maximum number of messages handled by a single processor, the 2D schemes produce considerably better results than the 1D scheme, where 2D-A2A is slightly better than 2D-P2P.

Combining these experimental outcomes for communication pattern results and considering the interconnection network of our PC cluster, we expect 2D-P2P to be the best because of its lower number of message requirements and moderate communication volume requirements. In fact, the parallel performance of BSCM and PSCM given in Table 2 and Fig. 8 confirm those expectations. However, depending on the machine architecture, the aforementioned communication metrics would have different impacts on the parallel performance of BSCM and PSCM. For example, on an interconnection network with a high communication bandwidth, 2D-A2A may perform better than 2D-P2P since it mainly suffers from high communication volume.

6.3 Overall performance

In our experiments, the tolerance parameter was set to 0.8% of the mean value of the observed image. In general, the smaller the tolerance parameter ϵ, the better the quality of the restored images, and the larger the iteration numbers. For 0-to-255 gray-scale images, this value of the tolerance parameter provides high quality restorations. The relaxation parameter λ was taken as 1.7 as in [35]. The algorithms were initialized from the zero vector meaning that every pixel in the image to be recovered was assumed to be initially black. With these values, the number of iterations required for convergence are given in Table 5 for the proposed partitioning schemes. The overall convergence results for partitionings based on natural ordering are also included. The reason for this inclusion is to demonstrate that the partitioning schemes used

Table 3
Communication patterns for the H matrices corresponding to the images of size 300×400

		Total Message		Maximum Message			
		Volume	Number	Volume	Number	Volume	Number
H matrix	K	$y = Hx/q = H^T\pi$		$y = Hx$		$q = H^T\pi$	
1D-P2P							
iso300x400	4	1721	7.4	560	2.5	702	2.8
	8	3682	21.2	654	4.2	803	4.6
	16	7794	48.5	703	6.5	859	5.4
	24	12539	76.9	745	7.5	962	6.1
rot300x400	4	598	6.2	187	2.1	248	2.1
	8	1452	16.0	260	3.3	306	3.2
	16	3159	35.2	297	4.2	333	3.6
	24	5451	59.4	317	5.1	400	4.7
irt300x400	4	33827	6.4	11089	2.2	15417	3.0
	8	100993	29.8	15086	5.8	18833	5.8
	16	163814	104.7	11579	10.4	14327	10.3
	24	180096	168.3	8777	15.0	10904	11.3
2D-P2P							
iso300x400	4	1800	7.6	590	2.0	590	2.0
	8	4075	29.9	733	4.0	733	4.0
	16	10547	82.4	1170	6.0	1170	6.0
	24	17303	163.3	1282	8.0	1282	8.0
rot300x400	4	645	6.0	238	2.0	238	2.0
	8	1945	26.6	357	4.0	357	4.0
	16	6073	67.1	784	5.5	784	5.5
	24	9756	124.2	819	7.5	819	7.5
irt300x400	4	27430	8.0	10112	2.0	10112	2.0
	8	79212	31.2	11963	4.0	11963	4.0
	16	145351	94.0	11813	6.0	11813	6.0
	24	175723	191.8	9198	8.0	9198	8.0
2D-A2A							
iso300x400	4	1405	8.0	557	2.0	557	2.0
	8	6225	24.0	1034	3.0	1034	3.0
	16	20701	64.0	1867	4.0	1867	4.0
	24	37169	120.0	2413	5.0	2413	5.0
rot300x400	4	655	8.0	246	2.0	246	2.0
	8	2692	24.0	480	3.0	480	3.0
	16	10524	64.0	989	4.0	989	4.0
	24	19555	120.0	1239	5.0	1239	5.0
irt300x400	4	27346	8.0	10107	2.0	10107	2.0
	8	121924	24.0	18252	3.0	18252	3.0
	16	259999	64.0	18478	4.0	18478	4.0
	24	391431	120.0	18341	5.0	18341	5.0

Table 4
Communication patterns of the 2D partitionings—dissected into fold and expand phases—for the H matrices given in Table 3. Processors are orginized into dimensional meshes of size 2×2 for $K = 4$, 4×2 for $K = 8$, 4×4 for $K = 16$, and 6×4 for $K = 24$.

		Expand Hx / Fold $H^T\pi$				Fold Hx / Expand $H^T\pi$			
		Volume		Number		Volume		Number	
H matrix	K	Total	Max	Total	Max	Total	Max	Total	Max
P2P									
iso300x400	4	678	237	3.6	1.0	1122	354	4.0	1.0
	8	2223	452	21.9	3.0	1852	346	8.0	1.0
	16	1878	276	33.6	2.9	8717	1095	48.0	3.0
	24	3700	322	91.3	5.0	13603	1074	72.0	3.0
rot300x400	4	197	99	2.0	1.0	448	140	4.0	1.0
	8	829	153	18.6	3.0	1116	215	8.0	1.0
	16	818	148	19.1	2.5	5255	712	48.0	3.0
	24	1458	133	52.2	4.5	8298	746	72.0	3.0
irt300x400	4	12157	5920	4.0	1.0	15273	4192	4.0	1.0
	8	44535	7744	23.2	3.0	34677	5597	8.0	1.0
	16	44722	4237	46.0	3.0	100629	8984	48.0	3.0
	24	77991	4034	119.8	5.0	97733	5313	72.0	3.0
A2A									
iso300x400	4	597	296	4.0	1.0	808	260	4.0	1.0
	8	4410	637	16.0	2.0	1815	396	8.0	1.0
	16	4313	493	32.0	2.0	16388	1373	32.0	2.0
	24	10924	833	72.0	3.0	26246	1581	48.0	2.0
rot300x400	4	201	100	4.0	1.0	454	146	4.0	1.0
	8	1625	278	16.0	2.0	1067	203	8.0	1.0
	16	1632	221	32.0	2.0	8892	768	32.0	2.0
	24	4280	312	72.0	3.0	15275	928	48.0	2.0
irt300x400	4	12144	5921	4.0	1.0	15202	4186	4.0	1.0
	8	87538	12874	16.0	2.0	34386	5378	8.0	1.0
	16	87635	6858	32.0	2.0	172364	11620	32.0	2.0
	24	226086	10275	72.0	3.0	165345	8066	48.0	2.0

for achieving efficient parallel implementations do not degrade overall convergence performance. In Table 5, "Block" and "Cyclic" refer to block-striped and cyclic partitionings applied to the natural row order of the H matrix. Finally, "HP" denotes the partitionings obtained using hypergraph models.

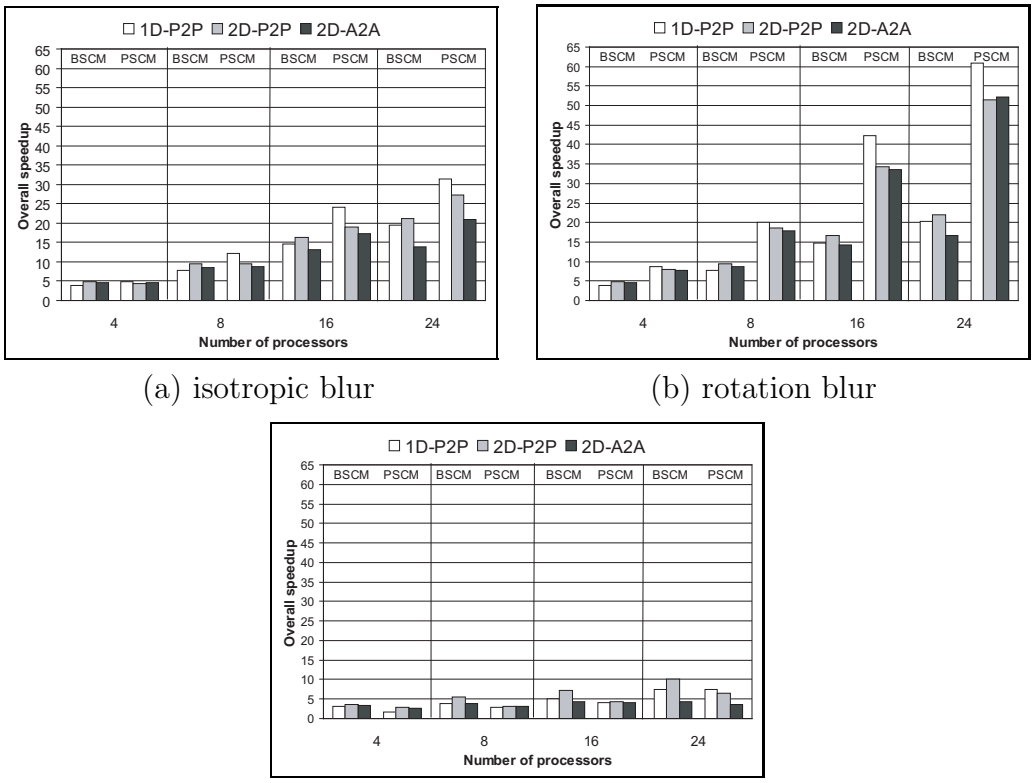
Comparison of PSCM and BSCM highlights the fact that increasing number of blocks reduces the number of iterations to convergence, in general. Furthermore, comparison of PSCM results for 1D partitionings shows that HP-based partitionings do not have a negative impact on the number of iterations to convergence. Comparison of 1D and 2D results reveals that 2D leads to larger number of iterations for a given number of processors. This is to be expected since the number of row blocks determines the speed of convergence of PSCM.

Table 5
The number of iterations to convergence for BSCM and parallel PSCM

			PSCM				
			1D			2D	
H matrix	BSCM	K	block	cyclic	HP	HP-P2P	HP-A2A
iso150x200	1818	4	1400	2490	1237	1400	1511
		8	1330	2243	932	1239	1172
		16	1234	1666	967	1207	1327
		24	1034	1458	800	1164	1094
iso300x400	2055	4	2332	6569	1700	2205	2018
		8	1837	4666	1336	1997	2097
		16	1802	3703	1236	1754	1724
		24	1794	2824	1270	1565	1674
iso600x800	4550	4	5797	20027	3570	4505	4103
		8	6280	13056	2694	3510	3558
		16	4154	10769	2625	3162	3615
		24	4347	9032	2254	2801	3466
rot150x200	2414	4	538	4309	994	1315	1419
		8	666	1248	813	978	1072
		16	521	1112	682	1021	1047
		24	443	1248	582	933	909
rot300x400	6055	4	2151	7206	2731	3525	3570
		8	2455	3143	2328	2941	2975
		16	3490	4345	2072	2881	2741
		24	3405	3939	2004	2463	2210
rot600x800	10640	4	4119	11332	4287	5376	5241
		8	4892	6761	2761	3676	3774
		16	3688	5768	2380	3721	3778
		24	2094	4088	2229	3226	3019
irt150x200	2363	4	1621	3981	1708	1890	2003
		8	1375	4206	1564	2003	1781
		16	1257	2650	1374	1817	1919
		24	1166	1927	1475	1968	1594
irt300x400	2957	4	5437	7517	5058	3790	3802
		8	4124	7500	3984	4907	4214
		16	7055	5029	3679	4781	4324
		24	5427	4286	2922	4355	5165
irt600x800	4390	4	9723	28890	9128	7659	8013
		8	8875	19905	9277	10198	11041
		16	7933	14512	6759	11160	9242
		24	6353	11131	5977	10525	9219

Recall that the number of row blocks in a 2D mesh of $K = R \times C$ processors is R as opposed to K in a 1D K-way partitioning.

Figure 9 displays the overall speedup values of parallel BSCM and PSCM for



(a) isotropic blur (b) rotation blur

(c) combined blur

Fig. 9. Overall speedup charts of BSCM and PSCM for the images of size 300×400.

the images of size 300×400. As seen in the figure, PSCM is superior to BSCM in all instances except for the irt300x400 matrix. Although 2D-P2P leads to better speedup on the per iteration basis, the winner with respect to overall performance is not clear. Note that the superlinear speedups for the isotropic and rotation cases are because of the reduced number of iterations.

Finally, we consider the preprocessing overhead of our parallel implementations. Table 6 gives the partitioning times of the matrices expressed in terms of the per iteration run time of the PSCM. This table shows that the cost of preprocessing is amortized in achieving accurate results.

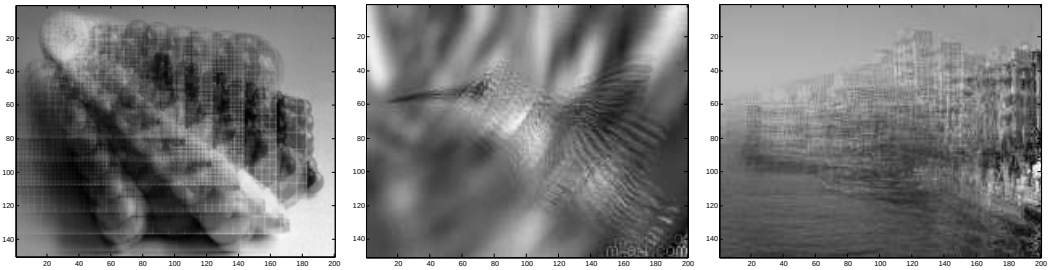
6.4 Restoration results

We evaluate the restoration performance of the parallel methods using the three images shown in Fig. 10(a). Blurred image g is generated using Eq. 2, or by simply multiplying f (original image) with H so that $g = Hf$ and adding the noise described earlier in this section. In Fig. 10(b), (c), and (d), the resulting distorted images are shown for the isotropic, rotational, and combined blurs, respectively.

Table 6
Preprocessing times for the images of size 400×300, expressed in terms of the per iteration times of the respective PSCM implementation

		Partitioning Scheme		
H matrix	K	1D P2P	2D P2P	2D A2A
iso400x300	4	125	115	220
	8	291	302	149
	16	618	835	110
	24	912	1287	81
rot400x300	4	68	83	144
	8	184	248	110
	16	458	553	70
	24	699	858	55
irt400x300	4	335	230	514
	8	1413	596	494
	16	4093	1290	258
	24	5228	3106	247

(a) Original images



(b) Noisy isotropic blur, (c) Noisy rotational blur, (d) Noisy combined blur

(e) Restore b, (f) Restore c, (g) Restore d

Fig. 10. Blurred and restored images.

With the same parameter values given earlier, we have restored the images by the surrogate constraint methods. The results corresponding to the isotropic, rotational, and combined blurs are given in Fig. 10(e), (f), and (g).

7 Conclusion

We studied the image restoration problem by formulating it as a system of linear inequalities. We used the surrogate constraint methods which are well suited to large problems and amenable to parallelization. The study concentrated on BSCM, which is the basic method, and on an improved version of the parallel method PSCM. We developed several parallel implementations. For efficient parallelization based on 1D and 2D partitionings of the coefficient matrix, we used state-of-the-art hypergraph partitioning schemes that minimize communication overhead while maintaining the load balance. Restoration abilities of the surrogate constraint implementations are validated using the parallel implementations for restoring severely blurred images.

The parallel implementation of BSCM was observed to produce better results compared with PSCM as far as the per iteration performance is concerned. However, increasing the number of blocks accelerates the speed of convergence significantly, hence PSCM outperforms BSCM with respect to the overall performance. In parallel PSCM, although 2D partitioning scheme leads to better speedup than the 1D scheme on the per iteration basis, 1D partitioning scheme performs comparably based on overall performance.

Note that satisfactory restorations can be achieved by decreasing the tolerance parameter at the expense of increased running 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 as soon as the restoration level satisfies the application requirements.

Acknowledgment

We are indebted to anonymous referees whose comments helped improve the presentation of Section 6.

References

[1] G. Burns, R. Daoud, J. Vaigl, LAM: an open cluster environment for MPI, in: J. W. Ross (Ed.), Proceedings of Supercomputing Symposium, 1994, pp. 179–186.

[2] Ü. V. Çatalyürek, Hypergraph Models for Sparse Matrix Partitioning and Reordering, Ph.D. thesis, Computer Engineering and Information Sciences, Bilkent University, 1999.

[3] Ü. V. Çatalyürek, C. Aykanat, Decomposing irregularly sparse matrices for parallel matrix-vector multiplication, Lecture Notes in Computer Science 1117 (1996) 75–86.

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

[5] Ü. V. Çatalyürek, C. Aykanat, PaToH: A multilevel hypergraph partitioning tool, version 3.0, Tech. Rep. BU-CE-9915, Computer Engineering Department, Bilkent University, 1999.

[6] Ü. V. Çatalyürek, C. Aykanat, A hypergraph-partitioning approach for coarse-grain decomposition, in: Proceedings of Scientific Computing 2001 (SC2001), Denver, Colorado, 2001, pp. 10–16.

[7] Y. Censor, T. Elfving, New method for linear inequalities, Linear Algebra and its Applications 42 (1982) 199–211.

[8] Y. Censor, S. A. Zenios, Parallel Optimization: Theory, Algorithms, and Applications, Oxford University Press, 1997.

[9] U. M. García-Palomares, F. J. Gonzalez-Castaño, Acceleration technique for solving convex (linear) systems via projection methods, Tech. rep., Escola Tecnica Superior de Enxeneiros de Telecomunicacion, 1996.

[10] M. R. Garey and D. S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W. H. Freeman & Co., New York, NY, USA, 1979.

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

[12] M. Hanke, Conjugate gradient type methods for ill-posed problems, Pitman Research Notes in Mathematics Series, Longman Scientific and Technical, Essex CM20 2JE, England, 1995.

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

[14] B. Hendrickson, T. G. Kolda, Graph partitioning models for parallel computing, Parallel Computing 26 (2000) 1519–1534.

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

[16] M. Jacunski, P. Sadayappan, D. K. Panda, All-to-all broadcast on switch-based clusters of workstations, in: IPPS '99/SPDP '99, Proceedings of the 13th International Symposium on Parallel Processing and the 10th Symposium on Parallel and Distributed Processing, IEEE Computer Society, Washington, DC, USA (1999) pp. 325–329.

[17] G. Karypis, V. Kumar, MeTiS: A software package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices Version 4.0, University of Minnesota, Department of Computer Science/Army HPC Research Center, Minneapolis, September 1998.

[18] G. Karypis, V. Kumar, Multilevel algorithms for multi-constraint graph partitioning, Tech. Rep. 98-019, University of Minnesota, Department of Computer Science/Army HPC Research Center, Minneapolis, May 1998.

[19] G. Karypis, V. Kumar, Multilevel algorithms for multi-constraint hypergraph partitioning, Tech. Rep. 99-034, University of Minnesota, Department of Computer Science/Army HPC Research Center, Minneapolis, November 1998.

[20] V. Kumar, A. Grama, A. Gupta, G. Karypis, Introduction to Parallel Computing: Design and Analysis of Algorithms, The Benjamin/Cummings Publishing Company, 1994.

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

[22] K. P. Lee, J. G. Nagy, Steepest descent, CG and iterative regularization of ill-posed problems, BIT 43 (2003) 1003–1017.

[23] T. Lengauer, Combinatorial Algorithms for Integrated Circuit Layout, Wiley-Teubner, Chichester, U.K., 1990.

[24] J. G. Lewis, R. A. van de Geijn, Distributed memory matrix-vector multiplication and conjugate gradient algorithms, in: Proceedings of the 1993 ACM/IEEE conference on Supercomputing, IEEE, Portland, Oregon, USA, 1993, pp. 484–492.

[25] J. G. Lewis, D. G. Payne, 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, Knoxville, TN, USA, 1994, pp. 542–550.

[26] J. G. Nagy, D. P. O'Leary, Restoring images degraded by spatially-variant blur, SIAM Journal on Scientific Computing 19 (1998) 1063–1082.

[27] J. G. Nagy, D. P. O'Leary, Fast iterative image restoration with a spatially-varying PSF, in: F. T. Luk (Ed.), Advanced Signal Processing Algorithms, Architectures, and Implementations IV, 3162 (1997), pp. 388–399.

[28] J. G. Nagy, K. Palmer, L. Perrone, Iterative methods for image deblurring: A Matlab object oriented approach, Numerical Algorithms 36 (2004) 73–93.

[29] H. Özaktaş, Algorithms for linear and convex feasibility problems: A brief study of iterative projection, localization and subgradient methods, Ph.D. thesis, Department of Industrial Engineering, Bilkent University, 1998.

[30] H. Özaktaş, M. Ç. Pınar, M. Akgül, The parallel surrogate constraint approach to the linear feasibility problem, Lecture Notes in Computer Science 1184 (1996) 565–574.

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

[32] Y. Saad, Iterative Methods for Sparse Linear Systems, PWS Publishing Co., Boston, 1996.

[33] R. S. Tuminaro, J. N. Shadid, S. A. Hutchinson, Parallel sparse matrix vector multiply software for matrices with data locality, Concurrency: Practice and Experience 10 (1998) 229–247.

[34] B. Uçar, C. Aykanat, Encapsulating multiple communication-cost metrics in partitioning sparse rectangular matrices for parallel matrix-vector multiplies, SIAM Journal on Scientific Computing 25 (2004) 1827–1859.

[35] K. Yang, K. G. Murty, New iterative methods for linear inequalities, Journal of Optimization Theory and Applications 72 (1992) 163–185.