Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 18 (2013) 581 – 590

Procedia Computer Science

International Conference on Computational Science, ICCS 2013

A scalable parallel LSQR algorithm for solving large-scale linear system for tomographic problems: a case study in seismic tomography

He Huang[a,*], John M. Dennis[b], Liqiang Wang[a], Po Chen[c]

[a]*Department of Computer Science, University of Wyoming. Laramie, WY 82071, U.S.A.*
[b]*National Center for Atmospheric Research. Boulder, CO 80301, U.S.A.*
[c]*Department of Geology and Geophysics, University of Wyoming. Laramie, WY 82071, U.S.A.*

Abstract

Least Squares with QR-factorization (LSQR) method is a widely used Krylov subspace algorithm to solve sparse rectangular linear systems for tomographic problems. Traditional parallel implementations of LSQR have the potential, depending on the non-zero structure of the matrix, to have significant communication cost. The communication cost can dramatically limit the scalability of the algorithm at large core counts. We describe a scalable parallel LSQR algorithm that utilizes the particular non-zero structure of matrices that occurs in tomographic problems. In particular, we specially treat the *kernel* component of the matrix, which is relatively dense with a random structure, and the *damping* component, which is very sparse and highly structured separately. The resulting algorithm has a scalable communication volume with a bounded number of communication neighbors regardless of core count. We present scaling studies from real seismic tomography datasets that illustrate good scalability up to $\mathcal{O}(10,000)$ cores on a Cray XT cluster.

Keywords: tomographic problems; seismic tomography; structural seismology; parallel scientific computing; LSQR; matrix vector multiplication; scalable communication; MPI

1. Introduction

Least Squares with QR factorization (LSQR) algorithm [1] is a member of the Conjugate Gradients (CG) family of iterative Krylov algorithms and is typically reliable when a matrix is ill-conditioned. The LSQR algorithm, which uses a Lanczos iteration to construct orthonormal basis vectors in both the model and data spaces, has been shown to converge faster than other algorithms in synthetic tomographic experiments [2].

Noninvasive tomographic problems that focuses on determining characteristics of an object (its shape, internal constitution, etc.) based on observations made on the boundary of the object is an important subject in the broad mathematical field known as inverse problems. Each observation d made on the boundary can usually be expressed as a projection of the unknown image $m(x)$ onto an integration kernel $K(x)$, whose form is highly problem-dependent. In mathematical form, this type of problems can often be expressed as $\int K(x)m(x)\,\mathrm{d}V(x)$. This integration equation can be transformed to a linear algebraic equation of the unknown image by discretizing x. The resulting inverse problem can be highly under-determined, as the number of observations can be much less

*Corresponding author. Email address: hhuang1@uwyo.edu

1877-0509 © 2013 The Authors. Published by Elsevier B.V.
Selection and peer review under responsibility of the organizers of the 2013 International Conference on Computational Science
doi:10.1016/j.procs.2013.05.222

than the total number of unknown parameters. Under such conditions, the inverse problem needs to be regularized and a regularization matrix, which is usually highly sparse with diagonal or block-diagonal structure, is appended below the kernel matrix.

Unfortunately, it can be computationally very challenging to apply LSQR to such tomographic matrix with a relatively dense kernel component appended by a highly sparse damping component, because it is simultaneously compute-, memory-, and communication-intensive. The coefficient matrix is typically very large and sparse. For example, a modest-sized dataset of the Los Angeles Basin (ANGF) for structural seismology has a physical domain of $496 \times 768 \times 50$ grid points. The corresponding coefficient matrix has 261 million rows, 38 million columns, and 5 billion non-zero values. The number of non-zeros in kernel is nearly 90% of the total while damping takes approximately 10%. The nearly dense rows within the coefficient matrix can generate excessive communication volume for a traditional row-based partitioning approach. Advances in structural seismology will likely increase the order of the design matrix by a factor of three. Clearly, an algorithm that scales with both problem size and core count is necessary.

In this paper, we address the computational challenges of using LSQR in seismic tomography which is made as a representative of tomographic problems. We propose a partitioning strategy and a computational algorithm that is based on the special structure of the matrix. Specially, SPLSQR contains a novel data decomposition strategy that treats different components of the matrix separately. SPLSQR algorithm results in an algorithm with scalable communication volume between a fixed and modest number of communication neighbors. SPLSQR algorithm enables scalability to $O(10,000)$ cores for the ANGF dataset in seismic tomography.

2. Related Work

LSQR is applied in a wide range of fields that involve reconstruction of images from a series of projections. It has been widely used in geophysical tomography to image subsurface geological structures using seismic waves, electromagnetic waves, or Bouguer gravity anomalies, etc. In structural seismology, the coefficient matrix is usually composed of kernel and damping component. For ray-theoretical travel-time seismic tomography, each row of the kernel component is computed from the geometry of the ray path that connects the seismic source and the seismic receiver, which usually results in a very sparse kernel component [3]. For full-wave seismic tomography, each row of the kernel which represents Frechet derivative of each misfit measurement vector is nearly dense [4, 5, 6]. The purpose of the damping is to regularize the solution of the linear system. The damping can be computed from the inverse of the model covariance. In practice, to penalize the roughness and the norm of the solution, a combination of the Laplacian operator implemented through finite-differencing and the identity operator can be used as the damping component.

The LSQR method is one of the most efficient algorithms so far for solving very large linear tomography systems, whether they are under-determined, over-determined or both [7, 2, 8]. There are several existing implementations of parallelized LSQR. Baur and Austen [9] presented a parallel implementation of LSQR by means of repeated vector-vector operations. PETSc [10] has an optimized parallel LSQR solver. PETSc is a well-optimized and widely-used scientific library, but it does suffer performance issues when using sparse matrices with random non-zero structure [11]. Liu *et al.* [12] proposed a parallel LSQR approach for seismic tomography. They partitioned the matrix into blocks by row and gave an approach to compute matrix-vector multiplication in parallel based on distributed memory. Their approach requires reduction on both vector x and y in each iteration. An MPI-CUDA implementation of LSQR (PLSQR) is described in [13]. The matrix vector multiplication uses a transpose free approach and requires only one reduction on the relative smaller vector x. Its major computation portions have been ported to GPU, and considerable speedup has been achieved.

3. Algorithm Overview

Figure 1 summarizes the idea of SPLSQR algorithm based on Message Passing Interface (MPI) programming model. The pseudo code describes the behavior of each MPI task. In particular, we describe the necessary matrix reordering, *i.e.*, line (01) to (04) in Section 4, data partitioning strategy, *i.e.*, line (05) to (08) in Section 5, and the calculation of the sparse matrix-vector multiplication, *i.e.*, line (10) to (26), in Section 6. Communication, *i.e.*, line (16) and (22), incurred during matrix-vector multiplication is detailed in Section 7.

(01) Reorder $Ad \in \mathcal{R}^{n_d \times m}$ to minimize bandwidth
(02) Obtain reduced B from Ad
(03) $p \leftarrow symrcm(BB^T)$ (Calculate row permutation)
(04) $Ad \leftarrow Ad(p, :)$ (Apply row permutation to Ad)
(05) Partition matrices: Ak, Ad and Adt:
(06) $Ak_i \leftarrow Ak$ (Partitioning across columns)
(07) $Ad_i \leftarrow Ad$ (Partitioning across rows)
(08) $Adt_i \leftarrow Adt$ (Partitioning across columns)
(09) Initialize Krylov vectors
(10) Iterate until converged
(11) Calculate: $y \leftarrow A \times x + y$
(12) Kernel component:
(13) $yk_i \leftarrow Ak_i \times x_i$, (partial results)
(14) $yk \leftarrow \sum_i yk_i$ (sum up partial results) [global sum]
(15) Damping component:
(16) Communicate with neighbors, reconstruct extended x'_i
(17) $yd_i \leftarrow Ad_i \times x'_i$
(18) Calculate: $x \leftarrow A^T \times y + x$
(19) Kernel component:
(20) $xk_i \leftarrow Ak_i^T \times yk$
(21) Damping component:
(22) Communicate with neighbors, reconstruct extended yd'_i
(23) $xd_i \leftarrow Adt_i \times yd'_i$
(24) Construct $x_i \leftarrow xk_i + xd_i$
(25) Construct and apply next orthogonal transformation
(26) Test convergence

where:
n_k: The number of rows of kernel submatrix.
n_d: The number of rows of damping submatrix.
n: The number of rows of the matrix. $n \leftarrow n_k + n_d, n_k \ll n, n_d \approx n$.
m: The number of columns of the matrix.
Ak: The kernel submatrix. $Ak \in \mathcal{R}^{n_k \times m}$,
Ad: The damping submatrix. $Ad \in \mathcal{R}^{n_d \times m}$,
B: The derived matrix by keeping the first non-zero in each row of Ad.
Adt: The transpose of Ad.
Ak_i: The piece of the kernel submatrix on task i.
Ad_i: The piece of the damping submatrix on task i.
Ak_i^T: The transpose of Ak_i.
Adt_i: The piece of the transposed damping submatrix on task i.
x_i: The $x \in \mathcal{R}^m$ vector on task i.
x'_i: The extended vector of x_i on task i from vector reconstruction.
xk_i: The $x \in \mathcal{R}^m$ vector on task i from $Ak_i^T \times yk$.
xd_i: The $x \in \mathcal{R}^m$ vector on task i from $Adt_i \times yd'_i$.
yk_i: The kernel component of vector $y \in \mathcal{R}^{n_k}$ on task i from $Ak_i \times x_i$.
yk: The reduced kernel component of vector $y \in \mathcal{R}^{n_k}$.
yd_i: The damping component of vector $y \in \mathcal{R}^{n_d}$ on task i from $Ad_i \times x'_i$.
yd'_i: The extended vector of yd_i on task i from vector reconstruction.
symrcm(): Calculates a permutation based on reverse Cuthill-McKee (RCM) algorithm.

Fig. 1: SPLSQR algorithm

4. Matrix Reordering

We next describe the necessary matrix reordering to prepare the original matrix used by SPLSQR algorithm. Note that this section represents lines (01) - (04) in Figure 1. The original damping submatrix with big bandwidth results in huge communication volume because there are large overlaps between MPI tasks after decomposition. Matrix reordering has long been used to reduce matrix bandwidth. The reverse Cuthill-McKee algorithm (RCM) [14] is a commonly used algorithm to reduce the bandwidth of sparse symmetric matrices. We want to utilize RCM reordering to restructure the non-zero pattern of matrix Ad. However, we cannot apply RCM directly to our matrix because of the following reasons:

- RCM only works on square matrix, while our matrix is a rectangular matrix.
- RCM requires applying both row and column permutation. Applying row permutation to damping does not require reordering kernel submatrix. But applying column permutation to damping submatrix requires applying the same column permutation to kernel submatrix as well in order to keep the final solution correct. Moving columns of kernel certainly introduces significantly additional overhead.
- RCM often offers one large band near the diagonal area, which introduces large volume of communication after matrix vector multiplication when computed in parallel because there are overlaps between MPI tasks.

We create a new square matrix $M = B \cdot B^T \in \mathcal{R}^{n_d \times n_d}$, where $B \in \mathcal{R}^{n_d \times m}$ for which we calculate a row permutation that will be used to reorder $Ad \in \mathcal{R}^{n_d \times m}$. Thus, column permutation is avoided. Such reordering of Ad has a small number of bands, where each band has its bandwidth minimized. The matrix B is derived using the first non-zero in each row of Ad, and ignoring any remaining non-zero elements in the row. By using only the first non-zero in each row of Ad, we preserve the banded structure while minimizing the bandwidth of each band.

Figure 2 illustrates procedures of reordering. The original damping matrix Ad, shown in panel(a), is inherently derived from Tikhonov regularization in three dimensions. The reduced matrix B in panel(b) is constructed from the first non-zero value in each row of (a). Panel(c) is the resulting reordered matrix obtained by applying the row permutation to the original matrix Ad. Panel(d) is an enlargement of the top part of the reordered matrix

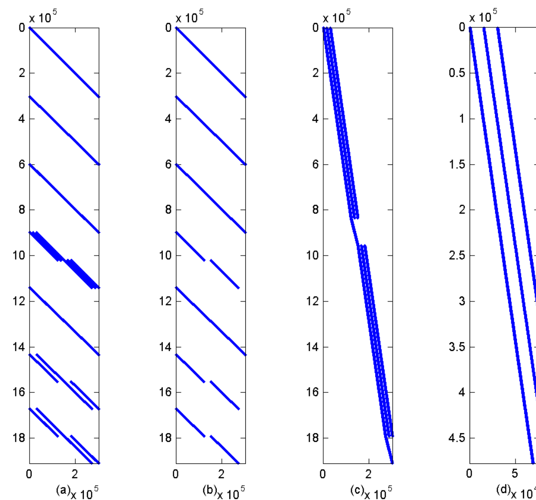


Fig. 2: The impact of RCM-based reordering algorithm. (a): original Ad; (b): reduced B; (c): reordered Ad; (d): enlargement of (c)

in panel(c). The desired multi-band structure is apparent in both panel(c) and panel(d). Reordering significantly reduces the bandwidth of each band.

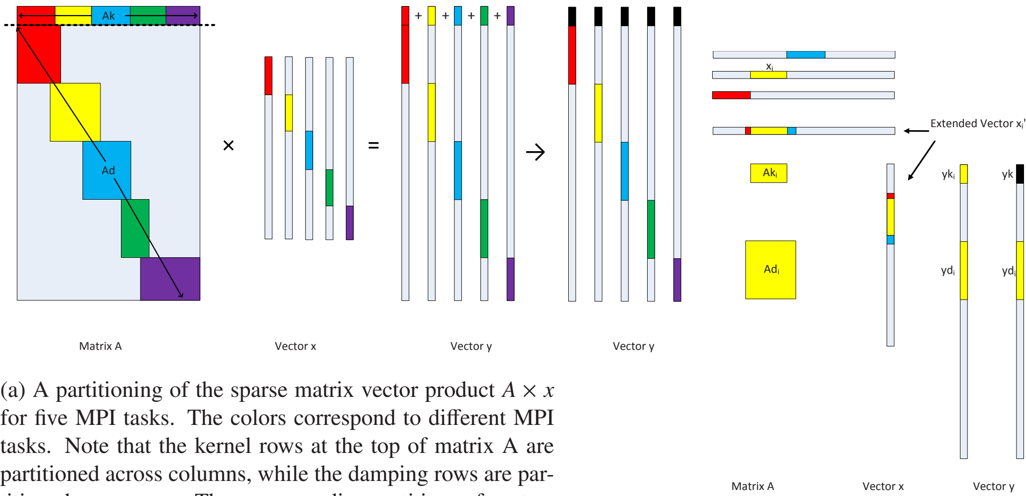
In seismic experiment, for a specific geological region corresponding to a kernel dataset, researchers often try different weights of damping matrix. Weights refer to the values of non-zeros in damping matrix. For the same kernel data, the non-zeros' positions in different damping submatrices are the same, but weights vary. Using RCM to calculate row permutation is both memory and calculation intensive, while applying permutation to matrix only moves the data in memory. The RCM algorithm is only sensitive to the position of non-zeros, but not sensitive to the non-zeros' values (weights). So the relative expensive row permutation (RCM reordering) only needs to be calculated once for the same geological region. For difference damping sbumatrices with different weights, we can reuse the same row permutation.

5. Data Decomposition

This section describes the partitioning of our data structures that correspond to lines (05) - (08) of our algorithm in Figure 1. We use compressed sparse row (CSR) format and compressed sparse column (CSC) format as matrix storage. Both formats preserve the values of non-zeros as well as their positions. Our partitioning approach is based on the particular structure of the coefficient matrix A. Recall that the number of rows in the kernel component n_k and the number of rows in the damping component n_d, we have $n_k \ll n_d$. For our ANGF dataset $n_k \approx 0.00001 n_d$. The number of non-zeros in kernel submatrix $Ak \in \mathcal{R}^{n_k \times m}$ is nearly 90% of the total. The damping matrix Ad contains approximately 10% non-zeros while there are a maximum of four non-zeros per row. The kernel component is nearly dense while the damping component is extremely sparse. Therefore, we use different approaches on partitioning the two different components. In particular, Ak is partitioned by columns while Ad is partitioned by rows.

We illustrate our partitioning of the coefficient matrix A in Figure 3a. Different colors represent different MPI tasks for both kernel Ak and damping Ad components. Note that Ad is already reordered into multi-band form. For simplicity and clarity, we do not include the multi-band structure in Figure 3a but rather just view Ad as having a single band. For the kernel component Ak, an equal number of columns are assigned to each MPI task. For the damping component Ad, an equal number of rows are allocated to each MPI task.

The partitioning of matrix A^T is illustrated in Figure 4a. While Ak^T maintains the same partitioning for Ak, the partitioning for Adt is different, where $Adt \in \mathcal{R}^{m \times n_d}$ is partitioned across columns, the same partitioning approach as Ak^T. Note that for consistency we will refer to any partitioning of A^T with respect to the original matrix A. Because Ak^T maintains the same partitioning, we are able to use a single copy of matrix Ak stored in CSC format in memory. Unlike the kernel component, we maintain two copies of the damping component, *i.e.*, one partitioned by rows Ad and one partitioned by columns Adt. The duplicate copy of the damping component simplifies the



(a) A partitioning of the sparse matrix vector product $A \times x$ for five MPI tasks. The colors correspond to different MPI tasks. Note that the kernel rows at the top of matrix A are partitioned across columns, while the damping rows are partitioned across rows. The corresponding partitions of vectors x and y are also included. Note that the piece of vector y that corresponds to the kernel rows is replicated across all tasks.

(b) Vector reconstruction and $A \times x$ ($Ak_i \times x_i$ and $Ad_i \times x_i'$) in one MPI task. Yellow represents local task.

Fig. 3: Illustration matrix vector multiplication: $A \times x$

algorithm and does not represent an excessive overhead because only 10% of the memory storage for the entire matrix A is a result of the damping component. Note that we do not use a superscript T for the transposed damping component but rather Adt to reflect that it is a separate copy of matrix within our algorithm. Ad is stored in CSR format and Adt is stored in CSC format.

The vector x is partitioned across columns and is illustrated in Figures 3a and 4a. Vector y has two parts in each task, a replicated kernel piece and a partitioned damping piece corresponding to the damping matrix. The length of kernel part of yk_i is the same as the number of rows in kernel Ak. The damping part of yd_i is partitioned according to the number of rows of damping Ad_i in each task.

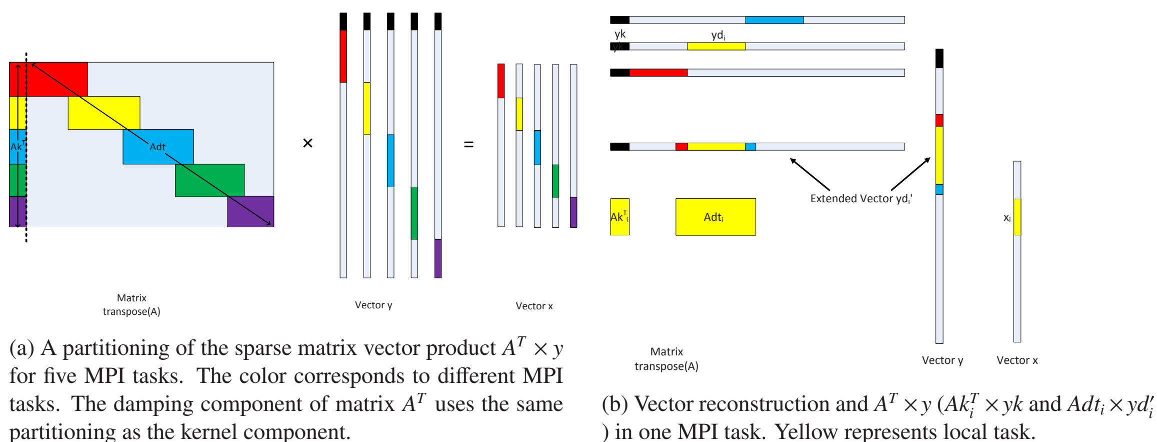
6. Parallel Computation

We next describe the most computationally expensive and complex section of SPLSQR algorithm, *i.e.*, the calculations of $y \leftarrow A \times x + y$ and $x \leftarrow A^T \times y + x$. The calculation of $y \leftarrow A \times x + y$ represents lines (11) - (17) in Figure 1, while $x \leftarrow A^T \times y + x$ represents lines (18) - (24). Both calculations require the reconstruction of the necessary pieces of vectors x and y by communicating with other MPI tasks. However they also differ in key characteristics due to the particular partitioning of matrices Ad and Ak.

6.1. $y \leftarrow A \times x + y$

Figure 3b illustrates the matrix vector multiplication $y \leftarrow A \times x$ from the perspective of the yellow task. We begin with a description of calculating yk that involves the kernel matrix. Each task multiplies its local piece of kernel Ak_i with local piece of x_i and yields its kernel part of vector, yk_i, as indicated by the top part of vector y in Figure 3b. Note that because of the column-based partitioning of Ak, yk_i is a partial result. Due to Ak's random and nearly dense feature, the resulting yk_i has overlap with all the other tasks. Hence a reduction across all tasks is performed on yk_i to combine the partial results. The final result is illustrated in black in Figure 3b. Because the number of rows in the kernel n_k is very small relative to the entire number of rows n, the total cost of the reduction only has a modest impact.

Next we describe the calculation of the damping component of yd_i. Unlike the calculation of yk_i, the calculation of yd_i requires additional pieces of x that the yellow task does not currently own. Note that in Figure 3b, the width of the Ad_i matrix is greater than x_i. The required additional pieces of x are owned by its neighboring



(a) A partitioning of the sparse matrix vector product $A^T \times y$ for five MPI tasks. The color corresponds to different MPI tasks. The damping component of matrix A^T uses the same partitioning as the kernel component.

(b) Vector reconstruction and $A^T \times y$ ($Ak_i^T \times yk$ and $Adt_i \times yd_i'$) in one MPI task. Yellow represents local task.

Fig. 4: Illustration of matrix vector multiplication: $A^T \times y$

red and blue tasks. Therefore, the yellow task needs to gather multiple pieces of data from its neighboring tasks to reconstruct an extended vector x_i'. The extended x_i' is multiplied with local damping Ad_i and yields local vector yd_i. No reduction is required for yd_i because of the row-based partitioning of Ad. In practice, communication with more than two neighbors may be necessary. The creation of both yk and yd_i completes the calculation of $y \leftarrow A \times x + y$.

6.2. $x \leftarrow A^T \times y + x$

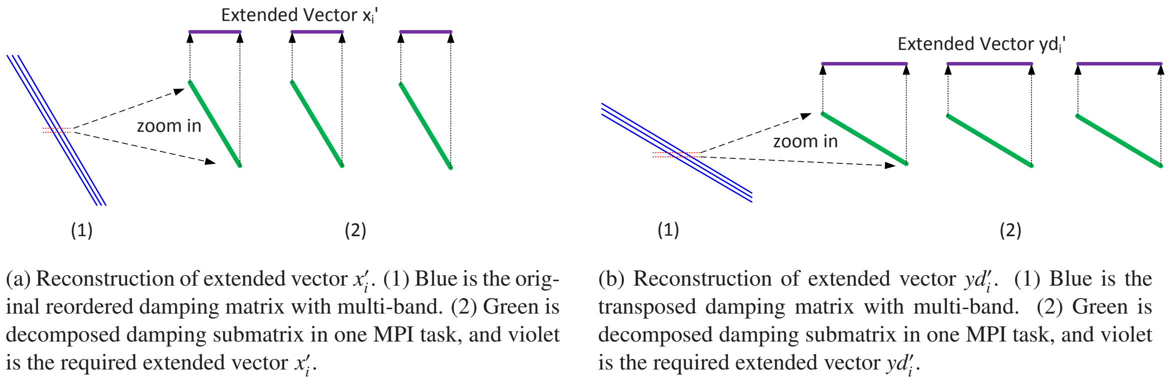
Figure 4b illustrates the multiplication of transposed matrix A^T with vector y for the yellow task from Figure 4a. We first multiply the transposed kernel Ak^T with yk. Here we utilize the original kernel matrix Ak_i^T stored in CSC format. We multiply a column of the matrix with kernel part of yk, which is equivalent to multiplying a row of the transposed matrix with yk. This technique is preferable to either explicitly transposing Ak or storing a duplicate copy of Ak. We have now constructed xk_i.

Next, each MPI task multiplies its local damping Adt_i matrix with local yd_i. Like vector x, vector reconstruction is necessary for vector y because local y_i is not sufficient to complete the multiplication for Adt_i. As Figure 4b illustrates, the data must be gathered by the yellow task from the red and blue tasks. The reconstructed vector yd_i' is multiplied with local damping Adt_i and yields xd_i. A local sum of xk_i and xd_i generates x_i to complete the calculation of $x \leftarrow A^T \times y + x$.

7. Communication

As a result of the data decomposition discussed in Section 5, communication cost is reduced in the vector reconstruction operations. Because x and y are partitioned (no overlap) among tasks, the resulting pieces of vector x and y after the matrix-vector multiplication in each iteration do not overlap with those owned by other MPI tasks. Another benefit is that data at a specific offset is located in only one of the MPI tasks. This avoids retrieving data from multiple neighbors during the vector reconstruction phase, and therefore simplifies communication. Also, decomposing the damping submatrix by row guarantees that there is no overlap in yd_i from $Ad_i \times x_i'$. Similarly, decomposing the transposed damping matrix by column, which is the same as the kernel submatrix, ensures that the resulting xk_i and xd_i have the same length on each task, with no overlap.

The multi-band structure of the reordered damping matrix helps reduce the amount of data that must be communicated with neighbors. Because the bandwidth of each band is very small, the gaps between bands are large, and therefore the required communication is low. Figures 5a and 5b illustrate vector reconstruction phase. Figure 5a (1) shows the multi-band structure of damping matrix after reordering. In this case, we assume that it has three bands, as seen in the simplified form of Figure 2 (c). More bands are possible depending on the internal structure



(a) Reconstruction of extended vector x_i'. (1) Blue is the original reordered damping matrix with multi-band. (2) Green is decomposed damping submatrix in one MPI task, and violet is the required extended vector x_i'.

(b) Reconstruction of extended vector yd_i'. (1) Blue is the transposed damping matrix with multi-band. (2) Green is decomposed damping submatrix in one MPI task, and violet is the required extended vector yd_i'.

Fig. 5: Reconstruction of extended vector

of the damping matrix. Figure 5a (2) is the decomposed submatrix of Figure 5a (1) on one MPI task, as shown in green. Then we project the matrix to the x-axis to obtain the required extended vector x_i'. The local task compares the required extended vector x_i' to its local x_i, and decides which MPI tasks it needs to communicate with. Note that there are gaps in the extended vector, which means that it only needs to gather three small pieces of vector x, as shown in violet color. The gap helps reduce the number of communication neighbors. Figure 5b (1) (blue) is the transpose form of Figure 5a (1). Figure 5b (2) (green) is the decomposed of Figure 5b (1) in one MPI task. We project the local submatrix to the x-axis to obtain the desired extended vector yd_i' shown in violet. The following actions are similar to the reconstruction of vector x_i'. The extended vector yd_i' also has gaps.

The communication volume for each MPI tasks of SPLSQR and PLSQR can be expressed as Equation 1 and Equation 2, respectively.

$$volume(SPLSQR) \leq 2 \cdot n_k \cdot (1 - \frac{1}{p}) + \frac{m}{p} \cdot g_x + \frac{n_d}{p} \cdot g_y \tag{1}$$

$$volume(PLSQR) = 2 \cdot m \cdot (1 - \frac{1}{p}) \tag{2}$$

where p is the core count, and g_x and g_y indicate size of the sending list and receiving list for vector x and y, respectively. Because $m \gg n_k$, the non-scalable communication volume, *i.e.*, 1st term of Equation 1, of SPLSQR is significantly less than the volume of PLSQR. When combined with scalable communication (2nd and 3rd terms in Equation 1) volume, the total volume of SPLSQR is still much less than that of PLSQR at large core count.

8. Performance Evaluation

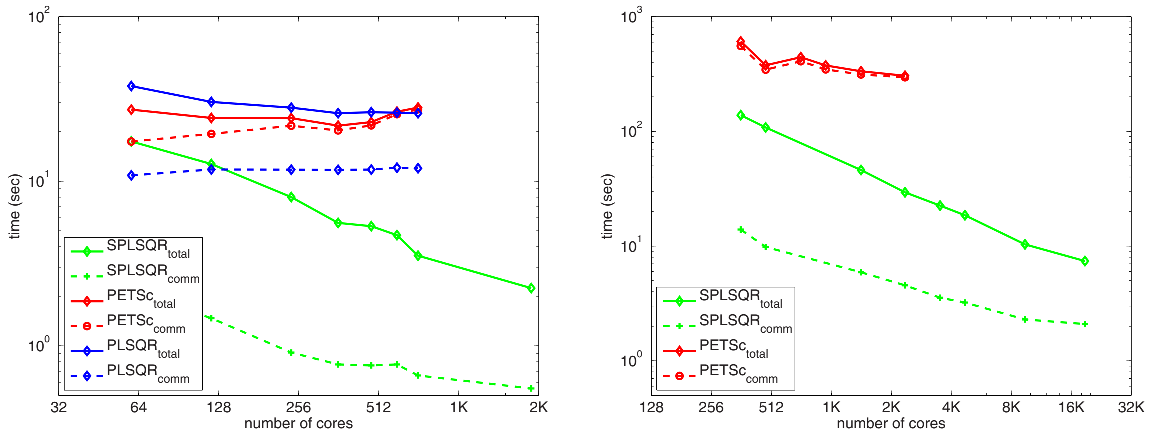
Table 1: Characteristics of seismic datasets.

		DEC3	ANGF
nx, ny, nz	physical domain ($nx \times ny \times nz$)	$165 \times 256 \times 16$	$496 \times 768 \times 50$
m	# column	1,351,680	38,092,800
n_k	# rows in kernel	3,543	3,543
n_d	# of rows in damping	8,877,544	261,330,576
nnz_{Ak}	# non-zeros kernel	183,113,885	5,321,630,642
nnz_{Ad}	# non-zeros damping	8,877,544	818,542,016

We examine the performance of SPLSQR algorithm on Kraken, a Cray XT5 system at the National Institute

for Computational Sciences (NICS) located at Oak Ridge National Laboratory. [1] Table 1 lists the characteristics of our two experimental datasets: ANGF and DEC3. We compare the performance characteristics of our SPLSQR algorithm implemented by MPI C with PLSQR (pure MPI C implementation without CUDA interference) and the PETSc implementation of the LSQR algorithm. Note that all three algorithms take a different parallelization approach. The PLSQR algorithm [13] assigns one or more kernel and damping rows to each MPI task. The partial calculation of the result $x \leftarrow A^T \times y$ requires the use of a global reduction on vector x. The PETSc algorithm uses the same partitioning of kernel and damping rows but uses vector-scatter operations to perform the vector reconstruction described in Sections 6.1 and 6.2. We use PETSc version 3.1.05 compiled with gcc 4.6.2.

8.1. Performance analysis



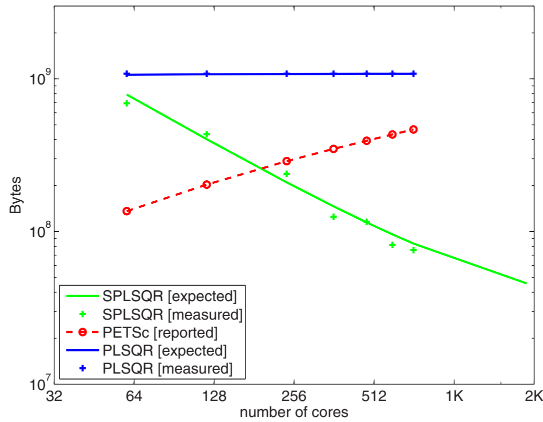
(a) Total and communication time for 100 iterations of the SPLSQR, PLSQR, and PETSc implementations of LSQR for the small DEC3 dataset from 60 to 1920 cores of a Cray XT5.

(b) Total and communication time for 100 iterations of the SPLSQR and PETSc implementations of LSQR for the modest ANGF dataset from 360 to 19,200 cores of a Cray XT5.

Fig. 6: Total and communication time on a small dataset DEC3 and a modest dataset ANGF

We provide the total execution time as well as the time spent on communication for the SPLSQR, PLSQR, and PETSc algorithms using 8-byte floating-point values in Figure 6a. We measure the time by performing 100 iterations of LSQR using our own timers for the SPLSQR and PLSQR algorithms, and the built-in timers for PETSc that are accessed using the flag "-log_summary". Note that for SPLSQR and PLSQR implementations, we use barriers to delineate the various sections of the algorithm, which does lead to a minor increase in overall time. For small core counts the PETSc algorithm has a somewhat smaller total time than for the PLSQR algorithm, while they have nearly identical times for core counts of 600 and 720. It is noteworthy that neither algorithm demonstrates much speedup on larger core counts. The PETSc algorithm achieves a speedup of 1.3x when core counts are increased from 60 to 360, while the PLSQR algorithm achieves a speedup of 1.5x when core counts are increased from 60 to 720. Neither PLSQR nor PETSc demonstrates any decrease in execution time after 360 cores. The SPLSQR algorithm demonstrates a significantly different performance profile. In particular not only is the execution time of the SPLSQR algorithm 1.7x less than the PETSc algorithm at 60 cores, it is 7.8x less at 720 cores. The SPLSQR algorithm achieves a speedup of 7.6x when the core counts are increased from 60 to 1920. The reason for the significantly different scaling characteristics between SPLSQR and the other two algorithms is due to differences in the communication cost. Figure 6a also illustrates the total communication cost associated with the sparse matrix-vector multiplication. While the communication costs for the PLSQR and

[1]Each of the 9,408 compute nodes in the Kraken system consists of two hex-core AMD Opteron 2435 (Istanbul) processors running at 2.6 GHz, for a total of 112,896 cores. Each compute node also has 16GB of DDR2-800, and nodes are connected with Cray SeaStar 2+ routers in a three-dimensional torus geometry.



(a) Send communication volume for the SPLSQR, PLSQR, and PETSc.

p	# core	60	360	720
	x vector reconstruction			
g_x	# neighbor	4.6	5.7	5.9
	msg length (Kbytes)	93	15	7.9
	y vector reconstruction			
g_y	# neighbor	4.6	5.7	5.9
	msg length (Kbytes)	315	54	28

(b) Average number of neighbors and message length in bytes for x and y vector reconstruction in SPLSQR.

Fig. 7: Communication on DEC3 dataset

PETSc algorithm are a significant component of overall cost, the communication cost for the SPLSQR algorithm is, relatively speaking, insignificant. In particular, the communication cost for SPLSQR is over 50x less than either the PLSQR or PETSc algorithm.

We next look at the scalability of the SPLSQR algorithm on large core counts. We focus on a comparison of the PETSc and SPLSQR algorithm for the ANGF dataset based on the results on DEC3. Figure 6b illustrates the total execution time and the communication time for the sparse matrix-vector multiplications for the SPLSQR and PETSc implementations of LSQR algorithm using the ANGF dataset on core counts of 360 to 19,200 on Kraken. Due to memory constraints, for the 360 and 480 core configurations we utilize 6 and 8 cores per node, respectively. All other core counts utilize 12 cores per node. Note that due to the particular partitioning of rows across MPI tasks, the total number of MPI tasks is limited to the total number of rows in the kernel component for PETSc. For the ANGF dataset, this limits PETSc to a maximum of 3543 MPI tasks. SPLSQR does not have a similar limit due to its different partitioning approach. It is apparent from Figure 6b that the SPLSQR algorithm has significantly lower execution time for all core counts. The reduction in execution time for SPLSQR versus PETSc varies from a low of 4.3x on 360 cores to a high of 9.9x on 2400 cores. The significant reduction in execution time achieved by SPLSQR is a direct result of reducing the communication cost.

Figure 6b also illustrates the communication cost for the SPLSQR and PETSc implementations using the ANGF dataset. As with the DEC3 dataset, we concentrate on the communication costs associated with the calculations of the sparse matrix-vector multiplications. It is very apparent from Figure 6b that the SPLSQR algorithm significantly reduces communication cost versus PETSc implementations by greater than a factor of 100x.

We next examine the reason for the vastly different communication costs between the three different algorithms. In Figure 7a, we provide the total amount of bytes transferred for the three different algorithms. We utilize Equation 1 to calculate the expected amount of send volume for the SPLSQR algorithm and Equation 2 for the PLSQR algorithm. Figure 7a also includes the amount of send volume as measured by the Cray Performance Analysis Tools [15] for the SPLSQR and PLSQR algorithms, as well as the amount of message traffic reported by the PETSc build-in profiling capabilities. Note that there is an excellent correlation between the measured and expected amount of send volume for the SPLSQR and PLSQR algorithms. As expected, the amount of communication volume per task is fixed for the PLSQR algorithm and scales with core count for the SPLSQR algorithm. The send volume increases with core count for the PETSc algorithm. Unfortunately the increase in send volume for the PETSc algorithm is an expected result due to the nearly dense kernels rows and the necessary vector reconstruction, because PETSc is designed for non-random sparse matrices. Closer examination of the message passing statistics for the PETSc algorithm indicates that not only does the total volume increase with core count but the average size decreases from 18 Kbytes on 60 cores to 1 Kbyte on 720 cores. Both the increase in volume

and decrease in message size explain why the communication cost for the PETSc algorithm increases with core count. The PLSQR algorithm that has a fixed communication volume regardless of core and fixed message size has a communication cost that increases very modestly with core count.

Table 7b shows the average number of communication neighbors and message length for the SPLSQR algorithm using 60, 360, and 720 core counts. Note that the x vector reconstruction, which is needed for $A \times x + y$, has a modest number of communication neighbors (g_x) and messages that are 10 times larger than the PETSc algorithm for similar core counts. Similarly, the y vector reconstruction, which is needed for $A^T \times y + x$, has a similar number of communication neighbors (g_y) and even longer messages. Note that the bounded number of communication neighbors and relatively long message size result in the very low communication cost of the SPLSQR algorithm.

9. Conclusions and Future Work

LSQR is a widely used numerical method to solve large sparse linear systems in tomographic problems. We describe the SPLSQR algorithm that utilizes particular characteristics of coefficient matrix that include both pseudo-dense and sparse components. We demonstrate that the SPLSQR algorithm has scalable communication volume and significantly reduces communication cost compared with existing algorithms. We also demonstrate that on a small seismic tomography dataset, the SPLSQR algorithm is 9.9 times faster than the PETSc algorithm on 2,400 cores of a Cray XT5. The current implementation of the SPLSQR algorithm on 19,200 cores of a Cray XT5 is 33 times faster than the fastest PETSc configuration on the modest ANGF dataset. In the future, we will extend SPLSQR to utilize additional parallel programming approaches, *e.g.*, OpenMP or CUDA.

Acknowledgements

We would like to acknowledge the Summer Internship in Parallel Computational Science (SIParCS) sponsored by the National Center for Atmospheric Research (NCAR) which initiated this collaboration. This work was supported through the National Science Foundation Cooperative Grant NSF01, which funds NCAR, and through other NSF grants, NSF-CAREER-1054834 and NSF-0941735.

References

[1] C. Paige, M. Saunders, LSQR: An algorithm for sparse linear equations and sparse least squares, ACM Transactions on Mathematical Software (TOMS) 8 (1) (1982) 43–71.
[2] G. Nolet, Solving or resolving inadequate and noisy tomographic systems, Journal of computational physics 61 (3) (1985) 463–482.
[3] H. Zhang, C. Thurber, Development and applications of double-difference seismic tomography, Pure and Applied Geophysics 163 (2) (2006) 373–403.
[4] L. Zhao, T. Jordan, K. Olsen, P. Chen, Fréchet kernels for imaging regional earth structure based on three-dimensional reference models, Bulletin of the Seismological Society of America 95 (6) (2005) 2066–2080.
[5] P. Chen, T. Jordan, L. Zhao, Full three-dimensional tomography: a comparison between the scattering-integral and adjoint-wavefield methods, Geophysical Journal International 170 (1) (2007) 175–181.
[6] P. Chen, L. Zhao, T. Jordan, Full 3d tomography for the crustal structure of the los angeles region, Bulletin of the Seismological Society of America 97 (4) (2007) 1094–1120.
[7] G. Nolet, Inversion and resolution of linear tomographic systems, EOS, Trans. Am. Geophys. Union 64 (1983) 775–776.
[8] J. Scales, Tomographic inversion via the conjugate gradient method, Geophysics 52 (1987) 179–185.
[9] O. Baur, G. Austen, A parallel iterative algorithm for large-scale problems of type potential field recovery from satellite data, in: Proceedings Joint champ/grace Science Meeting, Geoforschungszentrum Potsdam, 2005.
[10] S. Balay, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, H. Zhang, PETSc Web page, http://www.mcs.anl.gov/petsc (2009).
[11] A. Yoo, A. H. Baker, R. Pearce, V. E. Henson, A scalable eigensolver for large scale-free graphys using 2d graph partitoining, in: International Conference for High Performance Computing, Networking, Storage and Analysis, SC11, 2011, pp. 1–11.
[12] J. Liu, F. Liu, J. Liu, T. Hao, Parallel LSQR algorithms used in seismic tomography, Chinese Journal of Geophysics 49 (2) (2006) 540–545.
[13] H. Huang, L. Wang, E.-J. Lee, P. Chen, An MPI-CUDA implementation and optimization for parallel sparse equations and least squares (LSQR), in: 2012 International Conference on Computational Science (ICCS) (main track), Procedia Computer Science, Elsevier, 2012.
[14] E. Cuthill, J. McKee, Reducing the bandwidth of sparse symmetric matrices, in: Proceedings of the 24th national conference, ACM Press, New York, NY, USA, 1969, pp. 157–172.
[15] L. DeRose, B. Homer, D. Johnson, S. Kaufmann, H. Poxon, Cray performance analysis tools, Tools for High Performance Computing (2008) 191–199.