This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2014.2311804, IEEE Transactions on Parallel and Distributed Systems

A Novel Method for Scaling Iterative Solvers: Avoiding Latency Overhead of Parallel Sparse-Matrix Vector Multiplies

R. Oguz Selvitopi, Mustafa Ozdal, and Cevdet Aykanat

Abstract—In parallel linear iterative solvers, sparse matrix vector multiplication (SpMxV) incurs irregular point-to-point (P2P) communications, whereas inner product computations incur regular collective communications. These P2P communications cause an additional synchronization point with relatively high message latency costs due to small message sizes. In these solvers, each SpMxV is usually followed by an inner product computation that involves the output vector of SpMxV. Here, we exploit this property to propose a novel parallelization method that avoids the latency costs and synchronization overhead of P2P communications. Our method involves a computational and a communication rearrangement scheme. The computational rearrangement provides an alternative method for forming input vector of SpMxV and allows P2P and collective communications to be performed in a single phase. The communication rearrangement realizes this opportunity by embedding P2P communications into global collective communication operations. The proposed method grants a certain value on the maximum number of messages communicated regardless of the sparsity pattern of the matrix. The downside, however, is the increased message volume and the negligible redundant computation. We favor reducing the message latency costs at the expense of increasing message volume. Yet, we propose two iterative-improvement-based heuristics to alleviate the increase in the volume through one-to-one task-to-processor mapping. Our experiments on two supercomputers, Cray XE6 and IBM BlueGene/Q, up to 2048 processors show that the proposed parallelization method exhibits superior scalable performance compared to the conventional parallelization method.

Index Terms—Parallel linear iterative solvers, sparse matrix vector multiplication, point-to-point communication, inner product computation, collective communication, message latency overhead, avoiding latency, hiding latency, iterative improvement heuristic.

1 Introduction

Iterative solvers are the defacto standard for solving large, sparse, linear systems of equations on large-scale parallel architectures. In these solvers, two basic types of operations are repeatedly performed at each iteration: sparse matrix vector multiply (SpMxV) of the form $q = Ap$ and linear vector operations. Linear vector operations can further be categorized as inner product and DAXPY-like operations.

In the parallelization of these iterative solvers, linear vector operations are regular in nature since they operate on dense vectors and hence, they are easy to parallelize. On the other hand, SpMxV in general constitutes the most time consuming operation and it is hard to parallelize due to irregular task-to-task interaction caused by the irregular sparsity pattern of the coefficient matrix. Thus, the parallelization of iterative solvers are usually carried out by performing intelligent partitioning of matrix A that balances computational loads of the processors while minimizing the communication overhead that occurs during parallel SpMxV operations. Several sparse-matrix partitioning models and methods [2], [4], [14], [27], [28], [30] have been proposed and used in conjunction with respective parallel SpMxV algorithms. The matrix partitions obtained by using these models and methods are also decoded as partitioning linear vector operations among processors.

With the above-mentioned partitioning and parallelization schemes, parallel SpMxV computations incur irregular point-to-point (P2P) communication, and inner product operations incur regular global collective communication, whereas DAXPY-like linear vector operations do not incur any communication. Hence, both SpMxV and inner product computations cause separate synchronization points in the parallel solver. In general, the matrix partitioning schemes proposed and utilized in the literature mainly aim at minimizing the total communication volume, and this loosely relates to reducing the total message latency. However, on the current large-scale high performance computing systems, the message latency overhead is also a crucial factor affecting the performance of the parallel algorithm. Our analysis on two such well-known large scale systems, IBM BlueGene/Q and Cray XE6, shows that single message latency (i.e., startup time) is as high as transmitting four-to-eight kilobytes of data. Specifically, the message latency overhead caused by the processor that handles

- *R. Oguz Selvitopi and Cevdet Aykanat are with the Department of Computer Engineering, Bilkent University, Turkey, 06800. {reha, aykanat}@cs.bilkent.edu.tr*
- *Muhammet Mustafa Ozdal is with the Strategic CAD Labs of Intel Corporation, Hillsboro, OR 97124 US. mustafa.ozdal@intel.com*

This work was financially supported by the PRACE-2IP project funded in part by the EUs 7th Framework Programme (FP7/2007-2013) under grant agreement RI-283493 and FP7-261557.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2014.2311804, IEEE Transactions on Parallel and Distributed Systems

the maximum number of messages becomes the deciding factor for scaling the parallel algorithm. For example, in a row-parallel SpMxV algorithm [28] that utilizes 1D rowwise matrix partitioning, a dense column in matrix A necessitates a processor to send a message to almost all other processors, which significantly degrades the overall performance due to high latency overhead.

The motivation of this work is based on our observation that each SpMxV computation is followed by an inner product computation that involves the output vector of SpMxV in most of the Krylov subspace methods, which are among the most important iterative techniques available for solving large-scale linear systems. These two successive computational phases performed at each iteration contain write/read dependency due to the use of the output vector of the SpMxV computation with the following inner product computation. This in turn incurs dependency in the communications involved in these two successive phases. This observation is directly applicable to the following Krylov subspace methods: basic Arnoldi Method and its variants, basic GMRES, the Lanczos Algorithm, Conjugate Gradient, Conjugate Residual Method, Biconjugate Gradient, Biconjugate Gradient Stabilized, CGNR and CGNE. The reader is referred to [26] for analyzing computational dependencies in these Krylov subspace methods.

In this work, we exploit the above-mentioned property of the iterative solvers to propose a novel parallelization method that contains a computational and communication rearrangement scheme. The computational rearrangement resolves the computational write/read dependency between two successive computational phases so that the respective communication dependency can also vanish. This in turn enables combining P2P communications of SpMxV computations with the collective communications of inner products into a single communication phase. In other words, the computational rearrangement paves the way for communication rearrangement, which is realized with embedding P2P communication into collective communication operations.

Although invaluable in reducing the overhead due to the synchronization points, the proposed computational rearrangement causes redundant computations in DAXPY-like operations. However, we do not alter the computational structure of the iterative solver, thus, our method does not cause any numerical instability. In addition, the redundant computations are confined to communicated vector elements. Hence, the objective of minimizing total communication volume utilized in the existing intelligent partitioning methods also minimizes the total redundant computation.

The communication rearrangement is achieved by embedding vector elements communicated via P2P communication into global collective communication on scalars of local inner product results. This approach completely eliminates the message latency costs associated with the P2P communications and reduces the average and maximum number of messages handled by a single processor (both sent and received) to $\lg K$ for a system with K processors (K being a power of 2) regardless of the matrix used in the parallel solver. However, this embedding scheme causes extra communication due to forwarding of certain vector elements. We favor reducing the message latency costs at the expense of increasing message volume, which is invaluable for the scalability of the parallel algorithm, especially on systems with high message startup costs.

TABLE 1: Notation used throughout the paper.

Notation	Description
$\mathbf{Ax}=\mathbf{b}$	Sparse linear system being solved.
$\mathbf{p}$	Input vector.
$\mathbf{q}$	Output vector.
$\mathbf{r}$	Residual vector.
$\pi, \kappa, \alpha, \beta, \rho$	Scalars.
P_k	k^{th} processor.
$\mathbf{A}_k$	Rowwise portion of the matrix owned by P_k.
$\mathbf{p}_k(\mathbf{q}_k, \mathbf{r}_k, \mathbf{x}_k)$	Portion of the vector $\mathbf{p}$ ($\mathbf{q}$, $\mathbf{r}$, $\mathbf{x}$) owned by P_k.
$\widehat{\mathbf{p}}_k(\widehat{\mathbf{q}}_k)$	Augmented $\mathbf{p}$-vector ($\mathbf{q}$-vector) owned by P_k.
π^k, κ^k, ρ^k	The partial scalars computed by P_k.
n_k	Number of matrix row blocks or vector entries owned by P_k.
$\widehat{n}_k$	Size of the augmented vectors owned by P_k.
$\langle \mathbf{p}, \mathbf{q} \rangle$	Inner product of vectors $\mathbf{p}$ and $\mathbf{q}$.
ALL-REDUCE	Collective reduction operation.
SendSet(P_k)	Set of processors P_k will send vector entries to.
RecvSet(P_k)	Set of processors P_k will receive vector entries from.
$\mathbf{p}_{k\rightarrow l}$, $\mathbf{q}_{k\rightarrow l}$	Set of vector entries sent from P_k to P_l.

To address the increase in message volume, we propose two iterative-improvement-based algorithms. The main motivation of both algorithms is to keep the processors that communicate high volume of data close to each other in terms of communication pattern of collective operations so that the communicated vector elements cause less forwarding. The heuristics differ in their search space definitions. The first heuristic utilizes full space while the second one restricts it by considering only the directly communicating processors in collective communication operations. We show that the restricted space algorithm is feasible, and on the average, its running time remains lower than the partitioning time up to 2048 processors.

We show the validity of the proposed method on Conjugate Gradient (CG) algorithm, which is one of the best known iterative techniques used for solving sparse symmetric positive definite linear systems. Row-parallel SpMxV is adopted for the parallelization of CG, and column-net hypergraph model is used for intelligent partitioning of the sparse matrix [4]. We tested our parallelization method on two well-known large-scale systems Cray XE6 and IBM BlueGene/Q up to 2048 processors, comparing it to the conventional parallelization of CG using 16 symmetric matrices selected from University of Florida Sparse Matrix Collection [8]. The results on these two architectures show that reducing message latencies is critical for scalable performance, as our method obtains much better speedup results.

The rest of the paper is organized as follows. Section 2 presents the necessary background and the literature survey. In Section 3, our computational rearrangement

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2014.2311804, IEEE Transactions on Parallel and Distributed Systems

Algorithm 1: Basic Conjugate Gradient.

▷ Choose an initial $\mathbf{x}$ vector.
▷ Let $\mathbf{r} = \mathbf{p} = \mathbf{b} - \mathbf{A}\mathbf{x}$ and compute $\rho = \langle \mathbf{r}, \mathbf{r} \rangle$.
1 **while** $\rho > \epsilon$ **do**
2 $\quad \mathbf{q} = \mathbf{A}\mathbf{p}$
3 $\quad \pi = \langle \mathbf{p}, \mathbf{q} \rangle$
4 $\quad \alpha = \rho / \pi$
5 $\quad \mathbf{x} = \mathbf{x} + \alpha \mathbf{p}$
6 $\quad \mathbf{r} = \mathbf{r} - \alpha \mathbf{q}$
7 $\quad \rho_{new} = \langle \mathbf{r}, \mathbf{r} \rangle$
8 $\quad \beta = \rho_{new} / \rho$
9 $\quad \rho = \rho_{new}$
10 $\quad \mathbf{p} = \mathbf{r} + \beta \mathbf{p}$

scheme as well as the conventional parallelization of CG algorithm are presented. Section 4 explains the communication rearrangement scheme where message embedding is accomplished. Two heuristics for reducing extra communication volume are given in Section 5. Section 6 presents the experimental results. The Appendix is provided in the supplementary files.

2 Background and Related Work

Algorithm 1 displays the basic CG method [21], [26] used for solving $Ax = b$, where A is an $n \times n$ symmetric positive definite sparse matrix. The algorithm contains one SpMxV computation (line 2), two inner product computations (lines 3 and 7) and three DAXPY operations (lines 5, 6 and 10). The input vector p of the SpMxV at the subsequent iteration is obtained from the output vector q of the SpMxV of the current iteration through DAXPY operations (lines 6 and 10). Furthermore, the inner product at line 3 involves both the input and the output vector of the SpMxV operation. So, for parallelization, in order to avoid the communication of vector entries during linear vector operations, a symmetric partitioning scheme is usually adopted [4], [29], where all vectors in the solver are divided conformally with the partitioning of the sparse matrix. As seen in Algorithm 1, the SpMxV (line 2) and the two inner product computations (lines 3 and 7) are mutually interdependent. Hence there are three synchronization points: one due to the P2P communications of the SpMxV operation, and two separate collective communications for reducing the results of the local inner product computations at all processors.

The studies that address communication requirements of parallel CG usually adopt one or a combination of the approaches below:

- Reducing P2P communication overhead of parallel SpMxV with alternative partitioning strategies;
- Addressing communication requirements of inner products by utilizing alternative collective routines;
- Overlapping communication and computation;
- Reformulating CG to reduce the communication overhead of collective communication operations.

There are many works [4], [5], [15], [16], [20], [27], [28], [30] addressing communication requirements of parallel SpMxV operations. These studies generally center around sophisticated combinatorial models and intelligent partitioning methods which try to reduce the communication overhead of SpMxV operations and achieve scalability. Graph and hypergraph models are commonly employed in these works. The partitioning methods utilized in these works usually fall under the category of 1D and/or 2D sparse matrix partitioning.

In [10], authors argue that the communication overhead of inner products in CG and GMRES(m) become more significant and affect the scalability negatively with increasing number of processors. To this end, they suggest various methods to reduce this overhead. For CG, they restructure the parallel algorithm to overlap computations with inner product communications without affecting numerical stability of the iterative solver.

A thorough performance and scalability analysis of parallel CG is given in [13] on a variety of parallel architectures. Authors study block-tridiagonal and unstructured sparse matrices and analyze the effects of using a diagonal and a truncated Incomplete Cholesky preconditioner. They conclude that intelligent partitioning techniques are mandatory for scaling unstructured sparse matrices to improve the efficiency of parallel CG.

The work in [17] uses non-blocking collectives to reduce the communication requirements of parallel CG and aims at overlapping communication and computation by avoiding unnecessary synchronization. Note that although non-blocking interfaces of collective operations are included in MPI-3 standard, they are not realized in the widely adopted MPI-2 standard.

A recent work [11], [12] based on a reformulation of CG described in [7] propose a pipelined CG where the latency of global reduction is hidden by overlapping it with the computations of SpMxV or preconditioner. The authors use a single reduction in an iteration of the CG. They conduct extensive experiments to measure the stability of pipelined CG and test their method on a medium-scale cluster. The experimental results indicate that their method achieves better scalability while obtaining comparable convergence rates with the standard CG for the tested matrices. This work differs from our work in the sense that we aim at hiding latency of the communication due to the SpMxV computations rather than the latency of the global reduction operation.

Several other works [1], [9], [22], [24], [25] suggest a reformulation of the CG method in which the two distinct inner product computations can be performed in successive steps. This enables reducing results of inner product computations with a single global collective communication phase in a possible parallel implementation, reducing synchronization overheads. Usually, further experimental evaluations are performed for testing stability of these reformulations.

In this study, we use one of these reformulated versions [1], [24] for parallelization, which we present in

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2014.2311804, IEEE Transactions on Parallel and Distributed Systems

Algorithm 2: Reformulated Conjugate Gradient.

▷ Choose an initial $\mathbf{x}$ vector.
▷ Let $\mathbf{r} = \mathbf{p} = \mathbf{b} - \mathbf{A}\mathbf{x}$ and compute $\rho = \langle \mathbf{r}, \mathbf{r} \rangle$.
1 **while** $\rho > \epsilon$ **do**
2 $\quad \mathbf{q} = \mathbf{A}\mathbf{p}$
3 $\quad \pi = \langle \mathbf{p}, \mathbf{q} \rangle$
4 $\quad \kappa = \langle \mathbf{q}, \mathbf{q} \rangle$
5 $\quad \alpha = \rho / \pi$
6 $\quad \beta = \alpha \cdot \kappa / \pi - 1$
7 $\quad \rho = \beta \cdot \rho$
8 $\quad \mathbf{x} = \mathbf{x} + \alpha \mathbf{p}$
9 $\quad \mathbf{r} = \mathbf{r} - \alpha \mathbf{q}$
10 $\quad \mathbf{p} = \mathbf{r} + \beta \mathbf{p}$

Algorithm 2. In contrast to the basic CG algorithm, the inner products in the reformulated version at lines 3 and 4 are independent. Thus, the results of the two local inner products can be reduced in a single collective communication phase. However, both of these independent inner product computations still depend on the output vector of the SpMxV computation. Observe that the property that SpMxV is followed by inner product computation(s) holds both in the basic and the reformulated CG algorithms given in Algorithms 1 and 2, respectively. In fact, this property also holds in other reformulated versions [9], [22] as well. In the rest of the paper, we focus on this reformulated version, and whenever we mention the CG algorithm, we will be referring to Algorithm 2. It is reported in [24] that in this variant of CG, β can become negative due to rounding error. So, β should be checked at each iteration and in the case it is negative, it should be computed again using the classical formulation.

3 Computational Rearrangement

This section presents two parallelization methods for CG, both of which utilize row-parallel SpMxV. The first one is the conventional parallelization widely adopted in the literature, where a single iteration consists of two synchronization points. The other one is the proposed parallelization with computational rearrangement. The computational rearrangement provides an opportunity to perform P2P and collective communications in a single communication phase, and reduces the number of synchronization points from two to one. We opted to explain the conventional parallelization in this section to facilitate the presentation of the computational rearrangement and to make our contribution more clear and distinctive through direct comparisons.

In the parallel algorithms presented in this section (Algorithms 3 and 4), a subscript k denotes a local submatrix or subvector maintained or computed by processor P_k, whereas a superscript k denotes the result of a local inner product performed by P_k. A variable without a subscript or a superscript denotes a local copy of a global scalar.

3.1 Conventional Parallelization

The row-parallel SpMxV algorithm is based on a given 1D rowwise partition of $n \times n$ sparse matrix A of the form:

$$A = \left[\begin{array}{cccc} A_1^T & \dots & A_k^T & \dots & A_K^T \end{array} \right]^T,$$

where row stripe A_k is an $n_k \times n$ matrix for $k = 1, \dots, K$. Processor P_k stores row stripe A_k and is held responsible for computing $q_k = A_k p$ according to the owner computes rule [19]. The row-parallel algorithm requires a pre-communication phase in which p-vector entries are communicated through P2P messages to be used in the following local SpMxV operations. This communication phase contains expand-like operations, where individual p-vector entries are multicast to the processor(s) that need them. More details about the row-parallel algorithm can be found in [4], [27], [28].

Algorithm 3 presents the conventional parallelization of the CG method. Note the two distinct communication phases which are illustrated as the highlighted regions: P2P communication (lines 2-5) and collective communication (line 9). At the beginning of each iteration, processors perform the P2P communications (lines 2-5) necessary for local SpMxV operations. The sets of processors which P_k needs to send and receive vector entries are denoted by $SendSet(P_k)$ and $RecvSet(P_k)$, respectively. Note that $SendSet(P_k) = RecvSet(P_k)$ since A is symmetric. P_k needs to receive the entries in $p_{l \rightarrow k}$ from each $P_l \in RecvSet(P_k)$ (`RECV`$(P_l, \mathbf{p}_{l \rightarrow k})$) and send the entries in $p_{k \rightarrow l}$ to each $P_l \in SendSet(P_k)$ (`SEND`$(P_l, \mathbf{p}_{k \rightarrow l})$). Here, $p_{l \rightarrow k}$ denotes the set of p-vector entries that are received by P_k from P_l. After P_k receives *all* necessary non-local p-vector entries, it forms its *augmented* p vector, which is denoted as $\widehat{p}_k$ and contains $\widehat{n}_k \geq n_k$ elements.

After P2P communications, each processor P_k performs its local SpMxV $q_k = A_k \widehat{p}_k$ (line 6). Then, P_k computes the local inner products $\pi^k = \langle p_k, q_k \rangle$ and $\kappa^k = \langle q_k, q_k \rangle$ (lines 7 and 8). Since all processors need a copy of the global scalars α and β for the local DAXPY operations, they all need to know the final inner-product results π and κ, computed from local inner-product results as $\pi = \sum_{k=1}^{K} \pi^k$ and $\kappa = \sum_{k=1}^{K} \kappa^k$. For this purpose, a global reduction (`ALL-REDUCE`) is performed to compute π and κ (line 9). After this reduction operation, each processor P_k computes local copies of the scalars α and β so that it can update its local x_k, r_k and p_k vectors through DAXPY operations (lines 13, 14 and 15).

3.2 Proposed Alternative Parallelization

The conventional parallelization that adopts the row-parallel SpMxV necessitates P2P communications on the input vector entries prior to the local SpMxV computations. The main purpose of the computational rearrangement is to embed the P2P communications of SpMxV computations into the following collective communications of inner product computations, which involve

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2014.2311804, IEEE Transactions on Parallel and Distributed Systems

output vector of SpMxV. To enable this, P2P communications should be performed on the output vector entries immediately after local SpMxV computations.

Algorithm 4 shows our proposed technique for parallelizing the CG method. This simple yet effective computational rearrangement scheme presents an alternative way to form the local augmented vector $\widehat{p}_k$, which constitutes the main dependency among iterations. In contrast to Algorithm 3, the augmented input vector $\widehat{p}_k$ is not directly formed by P2P communication at the beginning of the current iteration, but rather computed using the respective $\widehat{q}_k$ and $\widehat{r}_k$ vectors in DAXPY operations of the previous iteration (lines 14-15). This is achieved by communicating q-vector entries in P2P communication (instead of p-vector entries) and forming the local augmented vector $\widehat{q}_k$ at each processor (lines 6-9). Then, P_k simply performs its two DAXPY operations on augmented vectors (lines 14-15); first updating $\widehat{r}_k$ by setting it to $\widehat{r}_k - \alpha\widehat{q}_k$ and then updating $\widehat{p}_k$ by setting it to $\widehat{r}_k + \beta\widehat{p}_k$. The computed local augmented vector $\widehat{p}_k$ is then used in the local SpMxV computations of the next iteration. The P2P communication of q_k entries in Algorithm 4 is performed together with the reduction operations, thus combining two communication phases of Algorithm 3 into a single communication phase (illustrated in the highlighted parts of the algorithm). Note that the DAXPY operation on x vector need not be performed using local augmented entries since it is not used in forming $\widehat{p}_k$. One DAXPY operation (line 13) in Algorithm 4 is performed on local vectors with n_k elements while two remaining DAXPY operations (lines 14-15) are performed on local augmented vectors with $\widehat{n}_k$ elements. Note that different from the conventional parallelization, the $\widehat{p}_k$ vector needs to be formed once before the iterations begin. After the first iteration, $\widehat{p}_k$ is not formed through communication but through DAXPY computations.

Compared to conventional parallelization, the drawback of our parallelization is the redundant computation performed by each processor P_k in two DAXPY operations (lines 14-15) for $\widehat{n}_k - n_k$ elements. Note that $\widehat{n}_k - n_k = |RecvVol\,(P_k)\,|$, where $RecvVol\,(P_k)$ denotes the set of vector elements that P_k receives. That is, each vector element received by P_k through P2P communication will incur two redundant multiply-and-add operations in the local DAXPY operations. Hence, the total redundant computation in terms of the number of multiply-and-add operations is two times the total message volume in terms of words transmitted.

Since the main computational burden in a single iteration is on the SpMxV operation in the CG method, this redundant computation in two linear vector operations is not of much concern. Nevertheless, the intelligent partitioning schemes utilized in the literature [4], [15] for partitioning matrix A, aim at minimizing the total message volume incurred in P2P communications. Hence, the partitioning objective of minimizing the total volume of communication corresponds to minimizing the total redundant computation as well.

Algorithm 3: Conventional parallelization.

▷ Choose an initial $\mathbf{x}$ vector.
▷ Let $\mathbf{r}_k = \mathbf{p}_k = \mathbf{b}_k - \mathbf{A}_k\mathbf{x}$ and compute $\rho^k = \langle \mathbf{r}_k, \mathbf{r}_k \rangle$.
▷ Reduce ρ^k to form ρ.

1 **while** $\rho > \epsilon$ **do**

▷ Communicate updated $\mathbf{p}_k$ entries.
2 **for** $P_l \in \mathit{SendSet}(P_k)$ **do**
3 SEND$(P_l, \mathbf{p}_{k\rightarrow l})$
4 **for** $P_l \in \mathit{RecvSet}(P_k)$ **do**
5 RECV$(P_l, \mathbf{p}_{l\rightarrow k})$ and update $\widehat{\mathbf{p}}_k$ entries

6 $\mathbf{q}_k = \mathbf{A}_k\widehat{\mathbf{p}}_k$
7 $\pi^k = \langle \mathbf{p}_k, \mathbf{q}_k \rangle$
8 $\kappa^k = \langle \mathbf{q}_k, \mathbf{q}_k \rangle$

▷ Reduce π^k and κ^k to obtain global coefficients.
9 $(\pi, \kappa) =$ ALL-REDUCE(π^k, κ^k)

10 $\alpha = \rho/\pi$
11 $\beta = \alpha \cdot \kappa/\pi - 1$
12 $\rho = \beta \cdot \rho$
13 $\mathbf{x}_k = \mathbf{x}_k + \alpha\mathbf{p}_k$ ▷ for n_k elements.
14 $\mathbf{r}_k = \mathbf{r}_k - \alpha\mathbf{q}_k$ ▷ for n_k elements.
15 $\mathbf{p}_k = \mathbf{r}_k + \beta\mathbf{p}_k$ ▷ for n_k elements.

Algorithm 4: Proposed parallelization.

▷ Choose an initial $\mathbf{x}$ vector.
▷ Let $\mathbf{r}_k = \mathbf{p}_k = \mathbf{b}_k - \mathbf{A}_k\mathbf{x}$ and compute $\rho^k = \langle \mathbf{r}_k, \mathbf{r}_k \rangle$.
▷ Reduce ρ^k and communicate $\mathbf{p}_k$ to form ρ and $\widehat{\mathbf{p}}_k$.

1 **while** $\rho > \epsilon$ **do**
2 $\mathbf{q}_k = \mathbf{A}_k\widehat{\mathbf{p}}_k$
3 $\pi^k = \langle \mathbf{p}_k, \mathbf{q}_k \rangle$
4 $\kappa^k = \langle \mathbf{q}_k, \mathbf{q}_k \rangle$

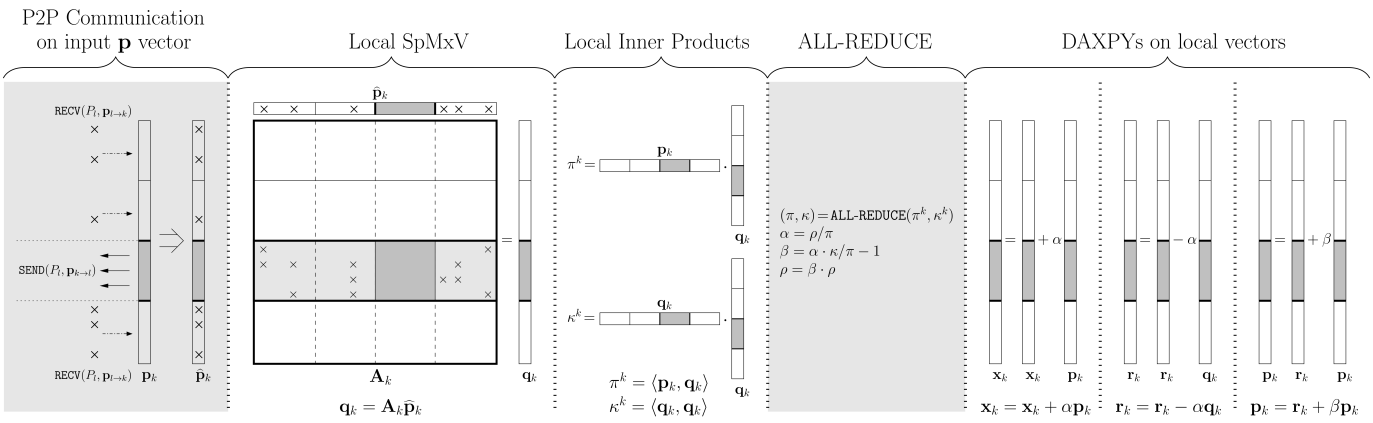
▷ Reduce π^k and κ^k to obtain global coefficients.
5 $(\pi, \kappa) =$ ALL-REDUCE(π^k, κ^k)

▷ Communicate $\mathbf{q}_k$ entries.
6 **for** $P_l \in \mathit{SendSet}(P_k)$ **do**
7 SEND$(P_l, \mathbf{q}_{k\rightarrow l})$
8 **for** $P_l \in \mathit{RecvSet}(P_k)$ **do**
9 RECV$(P_l, \mathbf{q}_{l\rightarrow k})$ and update $\widehat{\mathbf{q}}_k$ entries

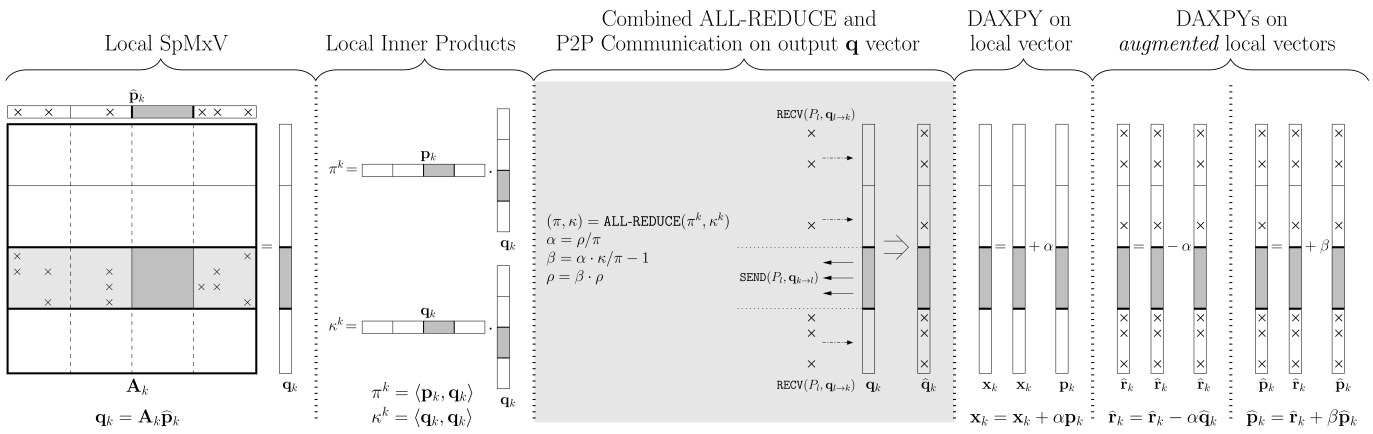
10 $\alpha = \rho/\pi$
11 $\beta = \alpha \cdot \kappa/\pi - 1$
12 $\rho = \beta \cdot \rho$
13 $\mathbf{x}_k = \mathbf{x}_k + \alpha\mathbf{p}_k$ ▷ for n_k elements.
14 $\widehat{\mathbf{r}}_k = \widehat{\mathbf{r}}_k - \alpha\widehat{\mathbf{q}}_k$ ▷ for $\widehat{n}_k \geq n_k$ elements.
15 $\widehat{\mathbf{p}}_k = \widehat{\mathbf{r}}_k + \beta\widehat{\mathbf{p}}_k$ ▷ for $\widehat{n}_k \geq n_k$ elements.

Fig. 1 presents a pictorial comparison of the conventional and alternative parallelization methods for $K = 4$ processors. The gray parts of the A matrix and the vectors visualize the submatrix and the subvectors assigned to processor P_k (for $k = 3$) and the computations per-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2014.2311804, IEEE Transactions on Parallel and Distributed Systems



(a) Conventional parallelization.



(b) Alternative parallelization.

Fig. 1: Illustration of conventional and alternative parallelization of conjugate gradient method.

formed by P_k on them. Light gray and dark gray blocks of A_k matrix illustrate the off-diagonal and diagonal blocks, respectively. In the figure, $\times$'s denote the nonzero off-diagonal column segments at row stripe A_k and the respective vector entries to be received by P_k. The figure also distinguishes the distinct phases of both algorithms as indicated at the top of their respective phases. The gray regions in the figure display the communication phases. As seen in Fig. 1a, the P2P communications on the input vector are performed just before the local SpMxV computations, whereas in Fig. 1b, the P2P communications on the output vector are performed after the local SpMxV computations. Fig. 1 clearly shows that the conventional parallelization scheme requires two communication phases, whereas the proposed scheme requires only one.

4 Embedding P2P Communications Into Collective Communication

In this section, we describe how to perform P2P and collective communication operations simultaneously. The main idea here is to use the underlying communication pattern of collective communication operations (ALL-REDUCE) for also communicating output vector entries.

In ALL-REDUCE, each processor P_k has its own buffer and ends up with receiving the result of an associative operation on the buffers of all other processors. The ALL-REDUCE operation can be performed in $\lg K$ communication steps [6], [23] in a K-processor system, where K is a power of two. This reduction algorithm is called bidirectional exchange and works by simultaneous exchange of data between processors. In step d, each processor exchanges a message with the processor in its 2^{d-1}-distance and updates the values in its local buffer with those in the received message using an associative operator. We adopt this communication pattern for the reduction operation and assume that K is a power of two for the simplicity of presentation.

In the ALL-REDUCE algorithm described above, P_k does not directly communicate with all processors in $SendSet(P_k)$. Now assume that P_k needs to send a set of q-vector entries to one such processor, P_l. Since it is definite that a message from P_k will *eventually* reach P_l in the reduction operation, it is possible to embed the vector elements that P_k needs to send to P_l into the corresponding messages. In other words, P_k may need to send some vector elements with the help of the processors it directly communicates with by embedding the necessary vector elements into its messages. Then, these processors would simply forward them to target

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2014.2311804, IEEE Transactions on Parallel and Distributed Systems

processors in $SendSet(P_k)$ that P_k does not directly communicate with.

Fig. 2 illustrates the communication steps of the ALL-REDUCE algorithm. The embedding process of P_1 with $SendSet(P_1) = \{P_0, P_2, P_4, P_6\}$ is displayed via solid arrows in the figure. In this example, P_1 can directly send the vector elements required by P_0 in Step 1 without any need for embedding. For P_1 to send its vector elements to P_2, it needs to embed them into its message to P_0 at Step 1, which are then forwarded from P_0 to P_2 at Step 2. For sending vector elements to P_4, P_1 also embeds them into its message to P_0 at Step 1, then P_0 waits for one step and forwards them to P_4 at Step 3. For P_6, P_1 embeds them into its message to P_0 at Step 1, which is then forwarded from P_0 to P_2 at Step 2, and from P_2 to its destination P_6 at Step 3. Note that the vector elements that are sent by P_1 to processors P_2, P_4 and P_6 are forwarded in certain steps of the algorithm.

Embedding vector elements into the communication pattern of ALL-REDUCE avoids startup costs for all messages due to P2P communications and establishes an exact value on the average and maximum number of messages being handled (sent and received) by a processor, which is $\lg K$. As will be shown by experiments, this is a significant advantage, and it is the key factor that leads to better scalability of the parallel solver. On the down side however, the message volume is likely to increase due to store-and-forward overhead associated with the forwarding of respective vector entries embedded in ALL-REDUCE operations. There exists a trade-off between avoiding message startup costs and increasing total volume of communication. We exploit the fact that if the number of communicated vector elements is not large, the startup costs can still be the dominating factor in total communication cost in spite of the increased volume. Thus, avoiding them will possibly compensate the increase in the message volume.

Note that the embedding scheme requires buffering due to the store-and-forward overhead. In the worst case, where each processor needs to send a message to every other processor in the system, the buffering overhead of a processor at a single step of the ALL-REDUCE algorithm is bounded by $O(K)$.

5 Part to Processor Mapping

Consider a given row partition $\mathcal{R} = \{R_1, R_2, \ldots, R_K\}$ of matrix A and a set of processors $\mathcal{P} = \{P_1, P_2, \ldots, P_K\}$, where the number of row parts is equal to the number of processors. In row partition $\mathcal{R}$, a column c_i is said to be a coupling column if more than one row parts contain at least one nonzero in c_i. Observe that, in the conventional parallel algorithm, only the input vector (i.e., p) entries associated with the coupling columns necessitate communication, whereas in the proposed parallel algorithm, only the output vector (i.e., q) entries associated with such columns necessitate communication. Let $\Lambda(c_i)$ denote the set of row parts that contain

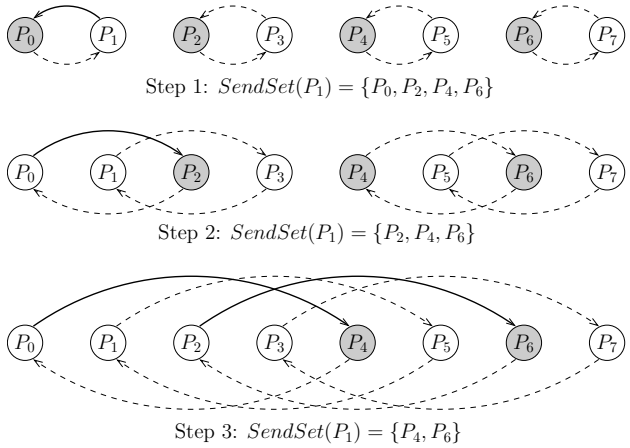


Fig. 2: Embedding messages of P_1 into ALL-REDUCE for $SendSet(P_1) = \{P_0, P_2, P_4, P_6\}$.

at least one nonzero in c_i. Without loss of generality, let row r_i be assigned to row part $R_k \in \mathcal{R}$. Now consider an identity mapping function $M : \mathcal{R} \rightarrow \mathcal{P}$ where the row block R_k is mapped to processor $P_{M(k)=k}$, for $1 \leq k \leq K$. Then, due to the symmetric partitioning requirement, q_i is assigned to P_k. Besides, since all diagonal entries are nonzero, we have $R_k \in \Lambda(c_i)$. So, $\{P_l : R_l \in \Lambda(c_i)$ and $R_l \neq R_k\}$ denotes the set of processors to which q_i should be sent (multicast) by processor P_k. Thus, $|\Lambda(c_i)|-1$ gives the volume of communication that is incurred by coupling column c_i. We define the set of processors that participate in the communication of q_i as $ProcSet(q_i) = \{P_l : R_l \in \Lambda(c_i)\}$, which includes the owner P_k of q_i as well, hence, $|ProcSet(q_i)| = |\Lambda(c_i)|$. For any arbitrary mapping, this definition becomes

$$ProcSet(q_i) = \{P_l : \exists R_m \in \Lambda(c_i) \text{ s.t. } M(m) = l\}. \quad (1)$$

For conventional parallelization, the total message volume is independent of the mapping, i.e., different part-to-processor mappings incur the same amount of message volume, which is:

$$ComVol(\mathcal{R}) = \sum_{q_i:\ c_i \in cc(\mathcal{R})} (|\Lambda(q_i)| - 1), \quad (2)$$

where $cc(\mathcal{R})$ denotes the set of coupling columns of the row partition $\mathcal{R}$. However, in the proposed parallelization scheme, the total message volume depends on the mapping of parts to processors due to forwarding of vector elements in the embedding process.

As an example, in Fig. 2, assume that two parts R_a and R_b are mapped to processors P_1 and P_6, respectively, and P_1 needs to send vector elements to P_6. These vector elements need to be forwarded in two steps, increasing communication volume compared to a single P2P communication between these two processors. However, if R_b were mapped to P_0 (or P_3, or P_5), these vector elements would not be forwarded, and they would incur no extra communication volume at all.

Based on this observation, the objective of mapping should be to minimize the extra communication volume due to forwarding. In other words, we should try to keep

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2014.2311804, IEEE Transactions on Parallel and Distributed Systems

the pairs of processors that communicate a large number of vector elements *close* to each other. The closeness here is defined in terms of the communication pattern of the ALL-REDUCE algorithm described in the previous section.

We now introduce assumptions and notations used to discuss the formulation adopted for computing total cost of a mapping M for a given row partition $\mathcal{R}$. We assume that the number of processors is an exact power of two (i.e., $K = 2^D$) and the processors are organized as a virtual D-dimensional hypercube topology H as the utilized ALL-REDUCE algorithm implies. In H, each processor is represented by a D-bit binary number. A dimension d is defined as the set of 2^{D-1} virtual bidirectional communication links connecting pairs of neighboring processors of which only differ in bit position d. Tearing along dimension d is defined as halving H_d into two disjoint $(d-1)$-dimensional subcubes, H_d^0 and H_d^1, such that their respective processors are connected along dimension d in a one-to-one manner. In this view, step d of the ALL-REDUCE algorithm can be considered as $K/2$ processors exchanging information along the $K/2$ virtual links of dimension d for $d = 0, 1, \ldots, D-1$.

For any coupling column c_i, the cost of communicating vector entry q_i is defined to be the number of ALL-REDUCE steps in which q_i is communicated. If q_i is communicated in step d of the ALL-REDUCE operation, we define the corresponding communication cost of q_i in this step as one, regardless of how many times q_i is communicated in this step because all communications of q_i in a single step are handled concurrently. Thus, in step d, q_i incurs a cost of one if the processors in $ProcSet(q_i)$ are scattered across different subcubes H_d^0 and H_d^1 of the tearing along dimension d. Otherwise, q_i does not incur any communication which corresponds to the case where all processors in $ProcSet(q_i)$ are confined to the same subcube of the tearing. Note that this latter case can be identified as all processors having the same value (either 0 or 1) at bit position d in their D-bit binary representations. Therefore, the communication cost of $q_i \in cc(\mathcal{R})$ is defined as:

$$cost(q_i) = \sum_{d=0}^{D-1} \Big(\bigwedge P_{k,d} \quad \otimes \quad \bigvee P_{k,d} \Big)_{P_k \in ProcSet(q_i)}. \quad (3)$$

In this equation, $P_{k,d}$ denotes the d^{th} bit of P_k in its D-bit binary representation, and $\wedge$, $\otimes$, and $\vee$ denote the logical "AND", "XOR", and "OR" operators, respectively. Then, the total cost of mapping M is simply given by:

$$cost(M) = \sum_{q_i:\ c_i \in cc(\mathcal{R})} cost(q_i). \quad (4)$$

We should note here that the cost definition in (4) captures an objective that is in between the total and concurrent communication overheads. In fact, it represents the sum of the number of distinct q-vector entries communicated in each step of the ALL-REDUCE algorithm. In other words, (3) corresponds to the total concurrent cost associated with forwarding q_i to the processors that it should be sent. The total message volume could easily be captured by counting exactly how many times q_i is communicated in each step of ALL-REDUCE instead of counting it only once. We preferred this cost definition in order to capture some form of concurrency in the optimization objective.

In order to find a good mapping, we propose two Kernighan-Lin (KL) [18] based heuristics. As typical in KL-type algorithms, the proposed heuristics start from a given initial mapping and perform a number of moves in the search space to improve the given mapping. For both heuristics, the move operator is defined as the swapping of the processor mapping of two row blocks. The gain of a swap operation is given as the reduction in the total communication cost of the mapping, as defined in (4). Both heuristics perform a number of passes till their improvement rate drops below a predetermined threshold. In each iteration of a single pass, the swap operation with the highest gain is chosen, tentatively performed and the respective row blocks are locked to prevent any further operations on them in the same pass. Best swaps with negative gains are also allowed to be selected in order to enable hill-climbing. At the end of a pass, a prefix of the performed swap operations with the highest cumulative cost improvement is selected as the resultant mapping to be used in the following pass.

Although both heuristics utilize the same move operators, they differ in their move neighborhood definitions. The first heuristic, KLF, considers the full move neighborhood with all possible $K(K-1)/2$ swaps, whereas the second heuristic, KLR, restricts the neighborhood over the adjacent processors of the virtual hypercube topology. In other words, KLR allows swapping only the parts at the processors that directly communicate in the ALL-REDUCE algorithm. Restricting the swap neighborhood has the following advantages over searching the full neighborhood: (i) Initial number of swaps reduces from $K(K-1)/2$ to $K \lg K/2$, (ii) gain updates performed after a swap operation become confined to the swap operations that are in the same dimension as the performed swap, and (iii) gain updates performed after a swap operation can be done in *constant* time. The obvious disadvantage of KLR is the possible loss in the quality of the generated mappings compared to KLF. However, as we show in the experiments, this loss is very small, only around 10%. In this sense, there is a tradeoff between running time and mapping quality, where KLR favors time and KLF favors quality.

In this paper, we only focus on describing the KLR heuristic because of its significantly better running time performance and algorithmic elegance. The detailed algorithms of KLR and a comprehensive complexity analysis are provided in Section 1 of Appendix.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2014.2311804, IEEE Transactions on Parallel and Distributed Systems

TABLE 2: Test matrices and their properties.

	Number of		Nonzeros per row/col		
Matrix	rows/cols	nonzeros	avg	min	max
bcsstk25	15,439	252,241	16.34	2	59
ncvxbqp1	50,000	349,968	7.00	2	9
tandem-dual	94,069	460,493	4.90	2	5
finan512	74,752	596,992	7.99	3	55
cbuckle	13,681	676,515	49.45	26	600
cyl6	13,681	714,241	52.21	36	721
copter2	55,476	759,952	13.70	4	45
Andrews	60,000	760,154	12.67	9	36
pli	22,695	1,350,309	59.50	11	108
pcrystk03	24,696	1,751,178	70.91	24	81
598a	110,971	1,483,868	13.37	5	26
opt1	15,449	1,930,655	124.97	44	243
wave	156,317	2,118,662	13.55	3	44
pkuskt07	16,860	2,418,804	143.46	39	267
kkt-power	2,063,494	12,771,361	7.28	2	96
crankseg-2	63,838	14,148,858	221.64	48	3423

6 Experiments

6.1 Experimental Framework

Four schemes are tested in the experiments: CONV, EMB, EMB-KLF and EMB-KLR. CONV refers to the conventional parallelization scheme described in Section 3.1 (Algorithm 3). EMB, EMB-KLF and EMB-KLR refer to the proposed parallelization scheme described in Section 3.2 (Algorithm 4). Hereafter, we will use notation EMB* to refer to these three embedded schemes. In all four schemes, row-parallel SpMxV algorithm is utilized, where the row partitions are obtained using the hypergraph partitioning tool PaToH on the column-net model [4] with default parameters. This model aims at minimizing total communication volume under the computational load balancing constraint. The load imbalance for all schemes is set to 10%. CONV and EMB rely on random row-part-to-processor mapping. EMB-KLF and EMB-KLR utilize the `KLF` and `KLR` row-part-to-processor mapping heuristics described in Section 5.

The number of passes for `KLF` and `KLR` is set to 10 and 20, respectively. Although lower number of passes could be used for these heuristics, we opted to keep them high to improve the mapping quality to a greater extent. In fact, a few number of passes would have been sufficient for `KLF` as it searches the full move neighborhood, whereas $\lg K$ passes would have been sufficient for `KLR` as it restricts the move neighborhood to the particular steps of the `ALL-REDUCE`.

Table 2 displays the properties of 16 structurally symmetric matrices collected from University of Florida Sparse Matrix Collection [8]. Matrices are sorted with respect to their nonzero counts.

We used two parallel systems in the experiments: Cray XE6 (XE6) and IBM Blue Gene/Q (BG/Q). A node on XE6 consists of 32 cores (two 16-core AMD processors) with 2.3 GHz clock frequency and 32 GB memory. The nodes are connected with a high speed 3D torus network called CRAY Gemini. A node on BG/Q consists of 16 cores (single PowerPC A2 processor) with 1.6 GHz clock frequency and 16 GB memory. The nodes are connected with 5D torus chip-to-chip network. We used $K \in 16, 32, \ldots, 1024$ cores on XE6 and $K \in 16, 32, \ldots, 2048$ cores on BG/Q for running parallel CG.

TABLE 3: Performance comparison of mapping heuristics `KLF` and `KLR` averaged over 16 matrices.

	mapping time normalized wrt partitioning time		% improvement in mapping cost wrt random map.	
K	KLF	KLR	KLF	KLR
16	0.02	0.05	39.4	32.7
32	0.11	0.10	44.1	39.2
64	0.42	0.14	46.5	41.9
128	1.45	0.24	45.9	41.0
256	4.71	0.44	47.8	42.9
512	13.24	0.61	46.5	41.1
1024	42.35	0.67	45.1	40.1
2048	129.64	1.21	41.4	37.9

6.2 Mapping Performance Analysis

Table 3 compares the `KLF` and `KLR` heuristics in terms of preprocessing time and mapping cost (computed according to Equation (4)). The mapping times in the table are normalized with respect to the partitioning times of PaToH. For each instance, first, a row partition of the input matrix is computed using PaToH, and a random part-to-processor mapping is generated. Then, the `KLF` and `KLR` heuristics are applied separately on this initial solution to obtain two different mapping results. The improvement rates obtained using these heuristics are reported separately as average over all 16 test matrices.

As seen in Table 3, `KLR`'s lower algorithmic complexity is reflected on its running time; as K increases, the average increase in `KLR`'s mapping time is much lower than that of `KLF`'s. Especially for large K values, `KLR` is more preferable than `KLF` because `KLR`'s mapping time becomes higher than the partitioning time. The mapping time of `KLR` remains well below the partitioning time up to 2048 processors. `KLR`'s faster mapping times are due to its successful move neighborhood restriction.

As Table 3 illustrates, `KLF` obtains better mappings than `KLR` because it uses a broader search space. The mappings obtained by `KLR` are marginally worse, only 8%–12% on average. There is a trade-off between running time and mapping quality. The trade-off here actually favors `KLR` since it is orders of magnitude faster than `KLF`, but it generates only slightly worse mappings.

6.3 Communication Requirements Assessment

Table 4 compares the performance of four parallel schemes in terms of their communication requirements averaged over 16 test matrices. Message counts of CONV include both P2P and collective communication phases. For CONV, the maximum message volume value refers to the maximum volume of communication handled during P2P operations, whereas for EMB* schemes, it refers to the sum of the communication volume values of the processors that handle maximum amount of

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2014.2311804, IEEE Transactions on Parallel and Distributed Systems

TABLE 4: Communication statistics averaged over 16 matrices.

	message count			message volume							
	CONV		EMB*	total				max			
K	avg	max	max (=avg)	CONV	EMB	EMB-KLF	EMB-KLR	CONV	EMB	EMB-KLF	EMB-KLR
16	9.8	12.8	4	0.522	0.780	0.651	0.719	0.102	0.115	0.103	0.108
32	13.2	19.3	5	0.839	1.509	1.186	1.277	0.083	0.115	0.099	0.108
64	16.1	27.4	6	1.304	2.595	1.951	2.158	0.070	0.114	0.096	0.099
128	19.3	34.5	7	1.986	4.550	3.170	3.453	0.055	0.111	0.086	0.092
256	22.2	43.5	8	2.989	7.901	5.012	5.497	0.047	0.112	0.076	0.078
512	25.2	53.8	9	4.522	14.053	7.981	8.673	0.036	0.114	0.064	0.068
1024	28.2	71.1	10	6.831	25.631	13.118	13.650	0.029	0.116	0.058	0.061
2048	31.3	85.0	11	10.669	49.821	25.211	25.130	0.025	0.122	0.054	0.052

In "message count" column, avg and max denote the average and maximum number of messages, respectively, sent by a single processor. In the "message volume" column, max denotes maximum message volume handled (sent and received) by a single processor. Message volume values are given in terms of number of floating points words and they are scaled by the number of rows/columns of the respective matrices.

communication in each step of ALL-REDUCE. Since each processor sends/receives a single message in each step of ALL-REDUCE, the maximum message volume effectively represents the concurrent communication volume as well. The detailed results per matrix basis are given in Section 2 of Appendix.

As seen in Table 4, for CONV, maximum message counts are significantly larger than average message counts for each K. This is due to the irregular sparsity patterns of the matrices which incur irregular P2P communications in parallel SpMxV computations. On the other hand, in EMB* schemes, average and maximum message counts are both equal to $\lg K$ for K processors independent of the sparsity pattern of the matrix.

In a parallel algorithm, the message latency overhead is actually determined by the processor that handles *maximum* number of messages. In that sense, as seen in Table 4, EMB* schemes perform significantly better than CONV for all K values. For example, for pkustk07 test matrix, the maximum message counts are $16, 25, 34, 34, 47, 60, 90, 96$ in CONV, while they are only $4, 5, 6, 7, 8, 9, 10, 11$ in embedded schemes, for $K = 16, 32, \ldots, 2048$ processors, respectively. This performance gap between CONV and EMB* schemes increases with increasing number of processors in favor of embedded schemes. For example, with K increasing from 16 to 2048 processors, the maximum message count increases 7.08 times for CONV whereas it only increase 2.75 times for EMB*, on the average.

As expected, EMB* schemes increase both total and maximum communication volumes compared to CONV. Even so, this increase remains rather low, especially for EMB-KLF and EMB-KLR schemes that utilize intelligent mapping heuristics. Besides, this increase also remains considerably low compared to the increase in the message latency overhead of CONV. The message latency overhead of CONV compared to those of EMB* schemes is greater than the communication volume overhead of EMB* schemes compared to that of CONV. For example, at $K = 2048$, CONV incurs 7.73 times the message latency overhead of EMB* while EMB-KLR incurs only 2.34 times the total message volume overhead and 2.63 times the maximum message volume overhead of CONV.

The mapping quality improvement rates of KLF and KLR (utilized in EMB-KLF and EMB-KLR) are roughly reflected in their reduction of message volume in the actual runs compared to the random mapping (utilized in EMB), especially for $K \geq 256$. For instance, as seen in Table 3, for $K = 1024$, the KLF and KLR improve the cost of the random mapping on the average by 45.1% and 40.1%, respectively. In the actual runs, although not presented explicitly (these values can easily be produced from Tables 1 and 2 in Section 2 of Appendix), compared to EMB, EMB-KLF obtains 46.5% less total message volume and 36.7% less maximum message volume, and EMB-KLR obtains 42.0% less total message volume and 32.8% less maximum message volume on the average for $K = 1024$. In that sense, it can be said that the objective used for mapping heuristics serves the purpose of reducing both total and maximum message volume successfully in the actual runs.

The communication cost of parallel SpMxV operations mainly depends on the communication cost of the bottleneck processor, which is by large determined by the maximum message count and maximum message volume requirements. As seen in Table 4, for all schemes, the maximum message volume requirements tend to decrease with increasing K. On the other hand, for CONV, maximum message counts tend to increase sharply with increasing K, whereas for EMB* schemes, maximum message counts increase very slowly (logarithmic growth) with increasing K. This implies that as the number processors increases, the message latency becomes more and more dominant in the overall communication cost. This fact enables embedded schemes to scale better, which is confirmed by the speedup curves reported in the next section.

Recall that EMB* schemes perform redundant computation due to computational rearrangement. On average, the EMB* schemes perform $0.1\%, 0.2\%, 0.3\%, 0.4\%, 0.6\%, 0.8\%, 1.2\%, 1.7\%$ more computation than CONV per processor for $K = 16, 32, \ldots, 2048$, respectively. This computational increase is very low and thus negligible.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2014.2311804, IEEE Transactions on Parallel and Distributed Systems

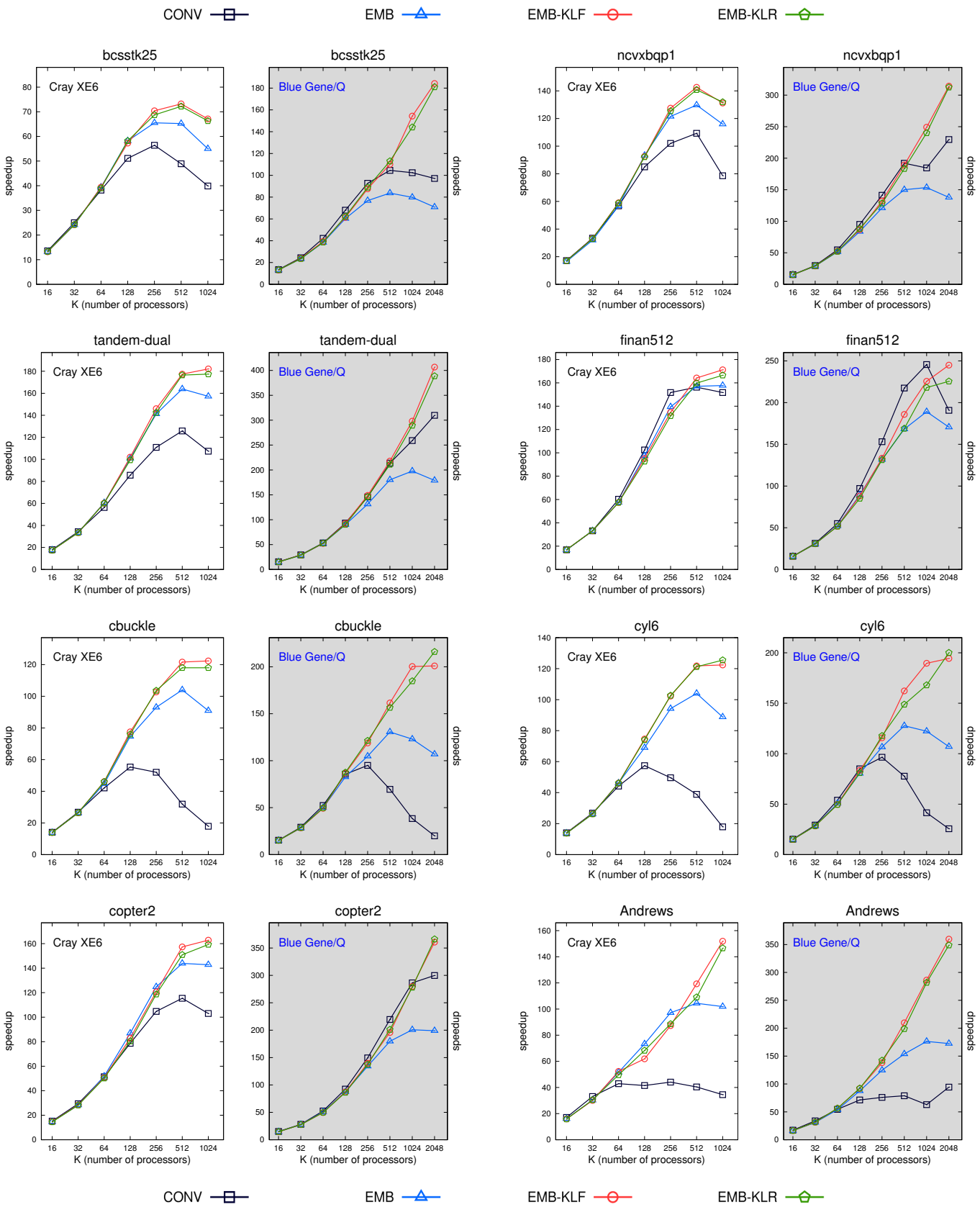


Fig. 3: Speedup curves for the first 8 of 16 test matrices.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2014.2311804, IEEE Transactions on Parallel and Distributed Systems

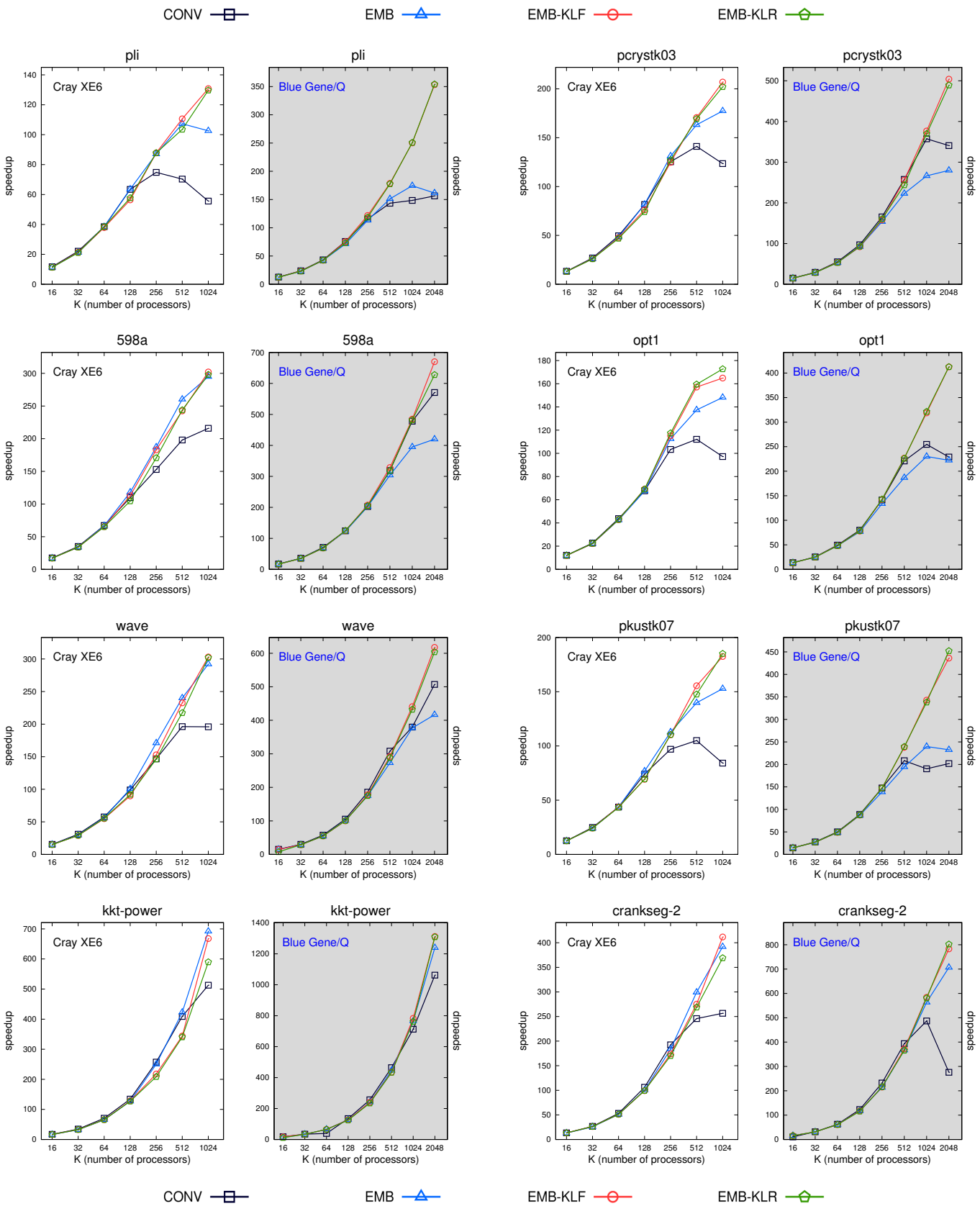


Fig. 4: Speedup curves for the last 8 of 16 test matrices.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2014.2311804, IEEE Transactions on Parallel and Distributed Systems

6.4 Speedup Results

Figs. 3 and 4 present the speedup curves of four tested schemes. The results obtained on XE6 and BG/Q supercomputers are illustrated with white and gray plots, respectively. These plots are grouped by test matrices for the ease of readability.

In both architectures, all schemes similarly scale up to $K = 64$ or $K = 128$ and then their distinctive characteristics begin to establish themselves with increasing number of processors. On XE6, all embedded schemes scale better than CONV, and EMB-KLF and EMB-KLR scale better than EMB by obtaining roughly the same speedup values. On BG/Q, EMB-KLF and EMB-KLR usually scale better than CONV and EMB, while CONV and EMB can scale better with respect to each other depending on the test matrix. We can say that the effect of message latency is more dominant on XE6, which leads to embedded schemes having better speedup values despite the increased message volume in general. Moreover, the embedded schemes start scaling better at lower K values compared to BG/Q. On the other hand, on BG/Q, this impact is not as dramatic as on XE6 and the effect of increased message volume in embedded schemes on speedup values is more prominent. This is basically due to relatively slow communication on BG/Q, which overshadows the benefits of reducing maximum message counts by making the embedded schemes' performance more sensitive to the increases in message volume. As seen in the plots that belong to BG/Q, EMB-KLF and EMB-KLR are usually able to obtain better speedup values at relatively higher K values where the message startup costs completely dominate the message volume costs.

Regarding the plots in Figs. 3 and 4, among 16 matrices, the lowest speedup values and poorest scalability characteristics belong to `Andrews`, `cbuckle` and `cyl6` matrices on both architectures for CONV scheme. They exhibit quite poor scaling performance where the speedup values start deteriorating very early at low K values compared to other test instances. For these matrices, the speedup values of CONV scheme are below 60 on XE6 with 1024 processors, and below 100 on BG/Q with 2048 processors. These three matrices have the highest communication requirements in terms of maximum message counts. The corresponding values are $83, 96, 108, 128$ for `Andrews` matrix, $36, 52, 82, 126$ for `cbuckle` matrix, and $36, 54, 78, 126$ for `cyl6` matrix for $K = 128, 256, 512, 1024$, respectively (see Table 2 in Appendix). This poor performance is basically because of the high latency overhead which becomes the decisive factor in communication and overall execution times with increasing number of processors. On the other hand, observe that the embedded schemes have much better scalability characteristics for these matrices due to their lower latency overheads. Note that sparsity patterns of the matrices, which depend on the application, along with the partitioning process as a whole, determine the communication requirements of the parallel solver.

Speedups on BG/Q are typically higher than XE6 since according to our running time analysis, the computation on XE6 is approximately 8 to 10 times faster than BG/Q. This enables computation to communication ratio to remain high and processors to be computationally intensive even at high K values for BG/Q, thus leading to higher speedups.

7 Conclusions and Future Work

We presented a novel parallelization scheme for linear iterative solvers, where point-to-point communications incurred by sparse-matrix vector multiplies and collective communications incurred by inner product computations can be performed in a single communication phase. Our parallelization provides an opportunity to reduce the synchronization overheads and establishes an exact value on the number messages communicated. We realized this opportunity by embedding point-to-point communications into collective communication operations. Embedding allows us to avoid all message startup costs of point-to-point communications at the cost of increasing message volume. Further, we presented two iterative-improvement-based heuristics to address this increase in the volume. The experiments were conducted on a Cray XE6 machine with up to 1024 processors and on a IBM BlueGene/Q machine with up to 2048 processors for test matrices from various domains. The results indicate that the message latencies become the determinant factor for the scalability of the solver with increasing number of processors. The results also show that our method, compared to conventional parallelization, yields better scalable performance by providing a low value on the number of messages communicated.

We plan to investigate applicability of the proposed embedding and rearrangement scheme to preconditioned iterative solvers. We believe that the proposed embedding scheme is directly applicable to the explicit preconditioning techniques such as approximate inverses or factored approximate inverses [3], [28]. Such preconditioners introduce one or two more SpMxV computations into the iterative solver. Since each SpMxV (either with the coefficient matrix or the preconditioner matrices) is often preceded/followed by global reduction operation(s), embedding of P2P communications of SpMxV operations into collective communication primitives should be viable. However, the computational rearrangement scheme may need modification according to the utilized preconditioning technique and the respective partitioning method used for it.

Acknowledgments

We acknowledge PRACE for awarding us access to resources Hermit (Cray XE6) based in Germany at High Performance Computing Center Stuttgart (HLRS) and Juqueen (Blue Gene/Q) based in Germany at Jülich Supercomputing Centre.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2014.2311804, IEEE Transactions on Parallel and Distributed Systems

REFERENCES

[1] C. Aykanat, F. Özgüner, F. Ercal, and P. Sadayappan. Iterative algorithms for solution of large sparse systems of linear equations on hypercubes. *IEEE Trans. Comput.*, 37(12):1554–1568, December 1988.

[2] Cevdet Aykanat, Ali Pinar, and Ümit V. Çatalyürek. Permuting sparse rectangular matrices into block-diagonal form. *SIAM J. Sci. Comput.*, 25:1860–1879, June 2004.

[3] Michele Benzi, Jane K. Cullum, and Miroslav Tuma. Robust approximate inverse preconditioning for the conjugate gradient method. *SIAM J. Sci. Comput.*, 22(4):1318–1332, April 2000.

[4] Umit Catalyurek and Cevdet Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Trans. Parallel Distrib. Syst.*, 10:673–693, July 1999.

[5] Ümit V. Çatalyürek, Cevdet Aykanat, and Bora Uçar. On two-dimensional sparse matrix partitioning: Models, methods, and a recipe. *SIAM J. Sci. Comput.*, 32(2):656–683, February 2010.

[6] Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert van de Geijn. Collective communication: theory, practice, and experience: Research articles. *Concurr. Comput. : Pract. Exper.*, 19(13):1749–1783, September 2007.

[7] A.T. Chronopoulos and C.W. Gear. s-step iterative methods for symmetric linear systems. *Journal of Computational and Applied Mathematics*, 25(2):153 – 168, 1989.

[8] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, December 2011.

[9] E. F. D'Azevedo, V. L. Eijkhout, and C. H. Romine. Conjugate Gradient Algorithms With Reduced Synchronization Overheads on Distributed Memory Processors. Technical Report 56, Lapack Working Note, 1993.

[10] E. de Sturler and H. A. van der Vorst. Reducing the effect of global communication in gmres(m) and cg on parallel distributed memory computers. *Appl. Numer. Math.*, 18(4):441–459, October 1995.

[11] P. Ghysels and W. Vanroose. Hiding global synchronization latency in the preconditioned conjugate gradient algorithm. Technical Report 12.2012.1, ExaScience Lab, Intel Labs Europe, December 2012.

[12] P. Ghysels and W. Vanroose. Hiding global synchronization latency in the preconditioned conjugate gradient algorithm. *Parallel Computing*, (0):–, 2013.

[13] A. Gupta, V. Kumar, and A. Sameh. Performance and scalability of preconditioned conjugate gradient methods on parallel computers. *Parallel and Distributed Systems, IEEE Transactions on*, 6(5):455–469, 1995.

[14] Bruce Hendrickson and Tamara G. Kolda. Partitioning rectangular and structurally unsymmetric sparse matrices for parallel processing. *SIAM J. Sci. Comput.*, 21(6):2048–2072, December 1999.

[15] Bruce Hendrickson and Tamara G. Kolda. Graph partitioning models for parallel computing. *Parallel Comput.*, 26(12):1519–1534, November 2000.

[16] Bruce Hendrickson, Robert Leland, and Steve Plimpton. An efficient parallel algorithm for matrix-vector multiplication. *International Journal of High Speed Computing*, 7:73–88, 1995.

[17] Torsten Hoefler, Peter Gottschling, Andrew Lumsdaine, and Wolfgang Rehm. Optimizing a conjugate gradient solver with non-blocking collective operations. *Parallel Comput.*, 33(9):624–633, September 2007.

[18] B.W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Tech. J.*, 49:291–307, 1970.

[19] Vipin Kumar. *Introduction to Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[20] J. G. Lewis and R. A. van de Geijn. Distributed memory matrix-vector multiplication and conjugate gradient algorithms. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, Supercomputing '93, pages 484–492, New York, NY, USA, 1993. ACM.

[21] David Luenberger and Ye. *Linear and Nonlinear Programming*. Springer, third edition, 2008.

[22] Grard Meurant. Multitasking the conjugate gradient method on the {CRAY} x-mp/48. *Parallel Computing*, 5(3):267 – 280, 1987.

[23] Sanjay Ranka and Sartaj Sahni. *Hypercube algorithms: with applications to image processing and pattern recognition*. Springer-Verlag New York, Inc., New York, NY, USA, 1990.

[24] Y. Saad. Practical use of polynomial preconditionings for the conjugate gradient method. *SIAM Journal on Scientific and Statistical Computing*, 6(4):865–881, 1985.

[25] Y. Saad. Krylov subspace methods on supercomputers. *SIAM J. Sci. Stat. Comput.*, 10(6):1200–1232, November 1989.

[26] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2003.

[27] Bora Uçar and Cevdet Aykanat. Encapsulating multiple communication-cost metrics in partitioning sparse rectangular matrices for parallel matrix-vector multiplies. *SIAM J. Sci. Comput.*, 25(6):1837–1859, 2004.

[28] Bora Uçar and Cevdet Aykanat. Partitioning sparse matrices for parallel preconditioned iterative methods. *SIAM J. Sci. Comput.*, 29(4):1683–1709, June 2007.

[29] Bora Uçar and Cevdet Aykanat. Revisiting hypergraph models for sparse matrix partitioning. *SIAM Rev.*, 49:595–603, November 2007.

[30] Brendan Vastenhouw and Rob H. Bisseling. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM Rev.*, 47:67–95, January 2005.

R. Oguz Selvitopi received his B.S. degree in Computer Engineering from Marmara University (2008) and M.S. degree (2010) in Computer Engineering from Bilkent University, Turkey where he is currently a PhD candidate. His research interests are parallel and distributed systems, parallel computing, scientific computing and bioinformatics.

Muhammet Mustafa Ozdal Muhammet Mustafa Ozdal received his B.S. degree in electrical engineering (1999), and M.S. degree in computer engineering (2001) from Bilkent University, Turkey. He obtained the Ph.D. degree in computer science from the University of Illinois at Urbana-Champaign in 2005. He was a recipient of the IEEE William J. McCalla ICCAD Best Paper Award in 2011, and the ACM SIGDA Technical Leadership Award in 2012. He has served as the program and general chair of IEEE/ACM SLIP, contest and publicity chair of ACM ISPD, and technical program committee member of the following IEEE/ACM conferences: ICCAD, DAC, DATE, ISPD, ISLPED, and SLIP. He is currently a research scientist in the Strategic CAD Labs of Intel Corporation. His research interests include heterogeneous computing, hardware/software co-design, high-performance computing, and algorithms for VLSI CAD.

Cevdet Aykanat received the BS and MS degrees from Middle East Technical University, Ankara, Turkey, both in electrical engineering, and the PhD degree from Ohio State University, Columbus, in electrical and computer engineering. He worked at the Intel Supercomputer Systems Division, Beaverton, Oregon, as a research associate. Since 1989, he has been affiliated with the Department of Computer Engineering, Bilkent University, Ankara, Turkey, where he is currently a professor. His research interests mainly include parallel computing, parallel scientific computing and its combinatorial aspects. (co)authored about 70 technical papers published in academic journals indexed in ISI and his publications received above 600 citations in ISI indexes. He is the recipient of the 1995 Young Investigator Award of The Scientific and Technological Research Council of Turkey and 2007 Parlar Science Award. He has served as a member of IFIP Working Group 10.3 (Concurrent System Technology) since 2004 and as an Associate Editor of IEEE Transactions of Parallel and Distributed Systems between 2008 and 2012.