HAL
open science

On shared-memory parallelization of a sparse matrix scaling algorithm

Umit V. Catalyurek, Kamer Kaya, Bora Uçar

To cite this version:

Umit V. Catalyurek, Kamer Kaya, Bora Uçar. On shared-memory parallelization of a sparse matrix scaling algorithm. 2012 41st International Conference on Parallel Processing, Sep 2012, Pittsburgh, PA, United States. pp.68–77. hal-00763553

HAL Id: hal-00763553

https://inria.hal.science/hal-00763553v1

Submitted on 19 Dec 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

On shared-memory parallelization of a sparse matrix scaling algorithm

Ümit V. Çatalyürek, Kamer Kaya
The Ohio State University
Dept. of Biomedical Informatics
{umit,kamer}@bmi.osu.edu

Bora Uçar
CNRS and LIP, ENS Lyon
Lyon 69364, France
bora.ucar@ens-lyon.fr

***Abstract*—We discuss efficient shared memory parallelization of sparse matrix computations whose main traits resemble to those of the sparse matrix-vector multiply operation. Such computations are difficult to parallelize because of the relatively small computational granularity characterized by small number of operations per each data access. Our main application is a sparse matrix scaling algorithm which is more memory bound than the sparse matrix vector multiplication operation. We take the application and parallelize it using the standard OpenMP programming principles. Apart from the common race condition avoiding constructs, we do not reorganize the algorithm. Rather, we identify associated performance metrics and describe models to optimize them. By using these models, we implement parallel matrix scaling algorithms for two well-known sparse matrix storage formats. Experimental results show that simple parallelization attempts which leave data/-work partitioning to the runtime scheduler can suffer from the overhead of avoiding race conditions especially when the number of threads increases. The proposed algorithms perform better than these algorithms by optimizing the identified performance metrics and reducing the overhead.**

***Keywords*-Shared-memory parallelization, sparse matrices, hypergraphs, matrix scaling**

I. Introduction

Sparse matrix computations working exclusively on nonzero entries are difficult to parallelize. The difficulty arises from the fact that the amount of computation per nonzero access is very small. Among the most well-known computations are sparse matrix-dense vector multiplies (SpMxV) of the form $\mathbf{y} \leftarrow \mathbf{Ax}$. In SpMxV, each nonzero is accessed, multiplied with an $\mathbf{x}$-vector entry, and the result is added to a $\mathbf{y}$-vector entry only once. That is, there are two reads and one read-and-write per each nonzero accessed. In this work, we parallelize an iterative matrix scaling algorithm where the computational core of each iteration is an operation that performs three reads and two read-and-writes per each nonzero accessed. In this light, the computations that we are targeting in this paper are harder to efficiently parallelize than the SpMxV operation. Our aim is to identify metrics to optimize parallel execution of the scaling algorithm and describe models for this purpose.

We assume that the algorithms will be parallelized using the standard OpenMP parallelization techniques, without a huge effort in reorganizing the algorithms. In particular, we take the point of view of a programmer that adopts loop-level parallelism and the so-called low-level, single-program multiple-data parallelism [1, p.32]. In both cases, to avoid race conditions, the programmer uses the now-standard technique of OpenMP directives, locks, atomic instructions, and/or large amount of private memory for each thread. These techniques incur overhead as additional work, a burden for CPU, or private read/write memory, a burden for cache efficiency. We propose models and methods to reduce the private memory usage, number of locks, atomic operations, and extra parallel work that is incurred. In all of these cases, we use hypergraphs and relate the objective of reducing the overhead to a suitable cost function in hypergraph partitioning problem.

Hypergraph partitioning-based models have been widely used to efficiently parallelize SpMxV and similar operations for distributed memory processor systems [2], [3], [4], [5], [6], [7]. These models identify the load balance and the total volume of communication as two relevant metrics for efficient parallelization. Although there are other communication cost metrics (such as the total number of messages or the maximum number or volume of messages per processors), the total volume of messages is deemed to be important to minimize first. Other communication cost metrics can be addressed during or after optimizing the partitioning for the total communication volume.

As should be evident, the sparse matrix storage format and the performance metrics are related. We focus on sparse matrix algorithms that use two of the most common formats. The coordinate format (COO) keeps an array of Z triplets of the form $\langle a_{ij}, i, j\rangle$ for a sparse matrix $\mathbf{A}$ with Z entries. Each triplet contains a nonzero entry a_{ij} and its row and column indices (i, j). In computations that use COO, it is therefore natural to assign works to threads based on triplets. The compressed row storage format (CRS) uses three arrays to store a $m \times n$ sparse matrix $\mathbf{A}$ with Z nonzeros. One array of size Z keeps the values of nonzeros where the nonzeros in a row are stored consecutively. Another array parallel to the first one keeps the column index of each nonzero. The third array keeps the starting index of the nonzeros at a given row where the ending index of the nonzeros at a row is one less than the starting index of the next row. In computations that use CRS, it is natural to assign works to threads by rows. We identify metrics and models related to the optimization

of algorithms that uses those formats.

We view the scaling algorithm that we parallelize as a model parallelization problem posing somewhat more challenges than the SpMxV operations. Nonetheless, most of the methods proposed for efficient parallelization of the SpMxV operations should be applicable to our problem. Some of the existing studies [8], [9], [10] reorganize the data, manipulate the data structures, and update the algorithms for parallel settings. It should be possible to apply our methods to at least some of these improved algorithms. This will require careful analyze of many different algorithms and surpass the limits and aims of this paper. Some other studies [11], [10], [12] apply architecture-aware techniques to greatly improve the performance of parallelized algorithms. It seems harder to add our techniques on top of these techniques. Yet some other studies [13], [14] improve the performance of the serial SpMxV computations by reordering the matrix and optimizing the cache performance. Our methods can be combined with these methods by reordering the submatrices assigned to each thread.

When compared with hypergraph partitioning, the execution time of matrix scaling is small. However, if one has several matrices with the same nonzero pattern and different nonzeros, which is the case for several applications, the reduction on the scaling time may compensate the cost for preprocessing. Furthermore, we chose scaling only as a model application. As the improvements proposed for SpMxV in the literature can be used for scaling, the models and techniques proposed in this paper can also be useful for SpMxV and related sparse-matrix problems. We refer the reader to [15], [16] for a comparison of the time spent for hypergraph partitioning with respect to the SpMxV time.

The organization of the paper is as follows. In Section II, we present the scaling algorithm and background material on hypergraph partitioning. In Section III, we present the shared-memory implementation of the scaling algorithm that uses the CRS and COO storage formats. We discuss two different standard OpenMP parallelization approaches and relate their performance metrics to well-known partitioning objectives in the literature. We evaluate the parallel algorithms experimentally in Section IV in order to show that the theoretical improvements due to the discussed models lead to improved running times in practice. We conclude the paper with a discussion in Section V.

II. Background

We present the scaling algorithm below. We then describe the hypergraph partitioning problem and three common objective functions. Hypergraph models to represent sparse matrices used in this paper are also presented.

A. The scaling algorithm

Matrix scaling algorithms have been used to improve the numerical stability in solving linear systems [17, Section 4.12]. Scaling a matrix $\mathbf{A}$ consists in pre- and post-multiplying $\mathbf{A}$ by two diagonal matrices, say $\mathbf{D}_r$ and $\mathbf{D}_c$, yielding a scaled matrix $\widehat{\mathbf{A}} = \mathbf{D}_r \mathbf{A} \mathbf{D}_c$.

We describe a scaling algorithm (originally proposed by Ruiz [18]; see also [19]) which finds scaling matrices $\mathbf{D}_r$ and $\mathbf{D}_c$ such that the rows and columns of $\widehat{\mathbf{A}} = \mathbf{D}_r \mathbf{A} \mathbf{D}_c$ has the same length in some norm. Common choices of the norm include 1-norm and ∞-norm scaling. We use $\|\mathbf{x}\|_1$ to denote the 1-norm of the vector $\mathbf{x}$ where $\|\mathbf{x}\|_1 = \sum |x_i|$ (similarly $\|\mathbf{x}\|_\infty$ is the ∞-norm of $\mathbf{x}$ where $\|\mathbf{x}\|_\infty = \max |x_i|$). Experiments with a sparse solver suggest the application of this algorithm both with 1- and ∞-norms [19], [20]. Here we focus on the 1-norm scaling, as the algorithm in ∞-norm converges very fast [19]. Since the 1-norm scaling is possible only for square matrices, we describe the algorithm with 1-norm on an $n \times n$, real, sparse matrix $\mathbf{A}$.

For $i = 1, \ldots, n$, let $\mathbf{r}_i = \mathbf{a}_{i*}^T \in \mathbb{R}^{n \times 1}$ denote the ith row of $\mathbf{A}$ as a vector, and for $j = 1, \ldots, n$, let $\mathbf{c}_j = \mathbf{a}_{*j} \in \mathbb{R}^{n \times 1}$ denote the jth column of $\mathbf{A}$. Let $\mathrm{diag}(\cdot)$ be an operator that takes a vector of size n and returns an $n \times n$ diagonal matrix having the entries of the vector in the main diagonal, i.e., for $\mathbf{D} = \mathrm{diag}(\mathbf{x})$ it holds that $d_{ii} = x_i$ and $d_{ij} = 0$ for $1 \leq i \neq j \leq n$. With these definitions, the scaling algorithm is formulated as shown in Algorithm 1.

Algorithm 1: 1-norm scaling algorithm

Input: $\mathbf{A}$: input matrix, square of size $n \times n$
Output: $\mathbf{D}_r$, $\mathbf{D}_c$: row and column scaling matrices
$\widehat{\mathbf{A}}^{(0)} \leftarrow \mathbf{A}$
$\mathbf{D}_r^{(0)} \leftarrow \mathbf{I}_n$
$\mathbf{D}_c^{(0)} \leftarrow \mathbf{I}_n$
for $k = 0, 1, 2, \ldots$ **until** *convergence* **do**
1 $\quad \mathbf{D}_1 \leftarrow \mathrm{diag}\left(\sqrt{\|\mathbf{r}_i^{(k)}\|_1}\right)_{i=1,\ldots,n}$
2 $\quad \mathbf{D}_2 \leftarrow \mathrm{diag}\left(\sqrt{\|\mathbf{c}_j^{(k)}\|_1}\right)_{j=1,\ldots,n}$
$\quad \widehat{\mathbf{A}}^{(k+1)} \leftarrow \mathbf{D}_1^{-1} \widehat{\mathbf{A}}^{(k)} \mathbf{D}_2^{-1}$
$\quad \mathbf{D}_r^{(k+1)} \leftarrow \mathbf{D}_r^{(k)} \mathbf{D}_1^{-1}$
$\quad \mathbf{D}_c^{(k+1)} \leftarrow \mathbf{D}_c^{(k)} \mathbf{D}_2^{-1}$

In this algorithm, $\widehat{\mathbf{A}}^{(k)}$ denotes the scaled matrix at the kth iteration whose ith row and jth column are represented, respectively, by $\mathbf{r}_i^{(k)}$ and $\mathbf{c}_j^{(k)}$. Clearly, one does not need to store the iterated $\widehat{\mathbf{A}}^{(k)}$, rather one can access it through left and right multiplications, respectively, with the current scaling matrices $\mathbf{D}_r^{(k)}$ and $\mathbf{D}_c^{(k)}$. In other words, one uses $\mathbf{d}_r^{(k)}(i) \times a_{ij} \times \mathbf{d}_c^{(k)}(j)$ while computing $\mathbf{r}_i^{(k)}$ and $\mathbf{c}_j^{(k)}$. The diagonal matrices $\mathbf{D}_1$ and $\mathbf{D}_2$ are created anew at each iteration (using the square root of the norms of the rows and columns, respectively). The output, scaling matrices $\mathbf{D}_r$ and $\mathbf{D}_c$, are updated at the end of each iteration, where those resulting from the last iteration are returned. The convergence is obtained if $\max_i \left\{ |1 - \|\mathbf{r}_i^{(k)}\|_1| \right\} \leq \varepsilon$ and

$\max_j \left\{ |1 - \|\mathbf{c}_j^{(k)}\|_1| \right\} \leq \varepsilon$ for a given value of $\varepsilon > 0$.

The computational core of the algorithm is the lines 1 and 2. In a proper sequential implementation of this algorithm, these two steps should be carried out as follows: $v \leftarrow \mathbf{d}_r^{(k)}(i) \times a_{ij} \times \mathbf{d}_c^{(k)}(j)$ and $\mathbf{d}_1(i) \leftarrow \mathbf{d}_1(i) + v$ and $\mathbf{d}_2(j) \leftarrow \mathbf{d}_2(j) + v$. In other words, one accesses a nonzero entry, retrieves the components of the $\mathbf{D}_r$ and $\mathbf{D}_c$ (three reads), performs the scalar multiplications, and then adds the result to the components of $\mathbf{D}_1$ and $\mathbf{D}_2$ (two read-and-writes). This high number of memory accesses per nonzero renders the overall algorithm hard to efficiently parallelize. Furthermore, our preliminary experiments verify that handling read/write conflicts during parallelization is crucial since the scaling algorithm does not self-stabilize in case of a conflict.

A distributed memory parallelization of this algorithm using MPI is reported by Amestoy et al. [21]. In that study, it has been shown that the communication volume requirements of such a parallel implementation is related to the communication volume requirements of a sparse matrix-vector multiply operation.

B. Hypergraphs and hypergraph models for sparse matrices

A hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ is defined as a set of vertices $\mathcal{V}$ and a set of nets (hyperedges) $\mathcal{N}$ among those vertices. A net $n \in \mathcal{N}$ is a subset of vertices, and the vertices in n are called its *pins*. The *size* of a net is the number of its pins, and the *degree* of a vertex is equal to the number of nets that contain it. Graph is a special instance of hypergraph such that each net has size two. Vertices can be associated with weights, denoted with $\mathsf{w}[\cdot]$, and nets can be associated with costs, denoted with $\mathsf{c}[\cdot]$.

A *K-way partition* of a hypergraph $\mathcal{H}$ is denoted as $\Pi = \{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_K\}$ where

- parts are pairwise disjoint, i.e., $\mathcal{V}_k \cap \mathcal{V}_\ell = \emptyset$ for all $1 \leq k < \ell \leq K$,
- each part $\mathcal{V}_k$ is a nonempty subset of $\mathcal{V}$, i.e., $\mathcal{V}_k \subseteq \mathcal{V}$ and $\mathcal{V}_k \neq \emptyset$ for $1 \leq k \leq K$,
- union of K parts is equal to $\mathcal{V}$, i.e., $\bigcup_{k=1}^{K} \mathcal{V}_k = \mathcal{V}$.

In a partition Π, a net that has at least one pin (vertex) in a part is said to *connect* that part. The number of parts connected by a net n, i.e., *connectivity*, is denoted as λ_n. A net n is said to be *uncut* (*internal*) if it connects exactly one part, and *cut* (*external*), otherwise (i.e., $\lambda_n > 1$).

Let W_k denote the total weight in $\mathcal{V}_k$ (i.e., $W_k = \sum_{v \in \mathcal{V}_k} \mathsf{w}[v]$) and W_{avg} denote the weight of each part when the total vertex weight is equally distributed (i.e., $W_{avg} = (\sum_{v \in \mathcal{V}} \mathsf{w}[v])/K$). If each part $\mathcal{V}_k \in \Pi$ satisfies the *balance criterion*

$$W_k \leq W_{avg}(1 + \varepsilon), \quad \text{for } k = 1, 2, \ldots, K \tag{1}$$

we say that Π is *balanced* where ε represents the maximum allowed imbalance ratio.

The set of external nets of a partition Π is denoted as $\mathcal{N}_E$. Let $\chi(\Pi)$ denote the cost, i.e., *cutsize*, of a partition Π. There are various cutsize definitions [22] such as:

$$\chi_{cut}(\Pi) = \sum_{n \in \mathcal{N}_E} \mathsf{c}[n] \,, \tag{2}$$

$$\chi_{conn}(\Pi) = \sum_{n \in \mathcal{N}} \mathsf{c}[n](\lambda_n - 1) \,, \tag{3}$$

$$\chi_{SOED}(\Pi) = \sum_{n \in \mathcal{N}_E} \mathsf{c}[n]\lambda_n \,. \tag{4}$$

The cutsize metric given in (2) will be referred to here as *cut-net* metric, the one in (3) will be referred as *connectivity-1* metric, and the one in (4) will be referred to as the SOED metric (widely used in the VLSI domain [23, p.10], [24], and recently found applications in the scientific computing domain [25]). Given ε and an integer $K > 1$, the hypergraph partitioning problem can be defined as the task of finding a balanced partition Π with K parts such that $\chi(\Pi)$ is minimized. The hypergraph partitioning problem is NP-hard [22] with any of the above objective functions. We used a state-of-the-art partitioning tool PaToH [26] which already has options to perform hypergraph partitioning in order to optimize cut-net and connectivity-1 metrics (while of course achieving balance). We have instrumented PaToH with the technique proposed by Yamazaki et al. [25] to address the SOED metric (4).

There are three well-known hypergraph models for sparse matrices. These are the column-net [3], row-net [3], and fine-grain models [27]. We describe these models below for a sparse matrix $\mathbf{A}$ of size $m \times n$ with Z nonzeros.

In the *column-net model*, $\mathbf{A}$ is represented as a unit-cost hypergraph $\mathcal{H}_{\mathcal{R}} = (\mathcal{V}_{\mathcal{R}}, \mathcal{N}_{\mathcal{C}})$ with $|\mathcal{V}_R| = m$ vertices, $|\mathcal{N}_C| = n$ nets, and Z pins. In $\mathcal{H}_{\mathcal{R}}$, there exists one vertex $v_i \in \mathcal{V}_{\mathcal{R}}$ for each row i. Weight $\mathsf{w}[v_i]$ of a vertex v_i is equal to the number of nonzeros in row i. There exists one unit-cost net $n_j \in \mathcal{N}_{\mathcal{C}}$ for each column j. Net n_j connects the vertices corresponding to the rows that have a nonzero in column j. That is, $v_i \in n_j$ if and only if $a_{ij} \neq 0$. The *row-net model* is the column-net model of the transpose of $\mathbf{A}$.

In the *fine-grain model*, $\mathbf{A}$ is represented as a unit-weight and unit-cost hypergraph $\mathcal{H}_{\mathcal{Z}} = (\mathcal{V}_{\mathcal{Z}}, \mathcal{N}_{\mathcal{RC}})$ with $|\mathcal{V}_{\mathcal{Z}}| = Z$ vertices, $|\mathcal{N}_{\mathcal{RC}}| = m+n$ nets and $2Z$ pins. In $\mathcal{V}_{\mathcal{Z}}$, there exists one unit-weight vertex v_{ij} for each nonzero a_{ij}. In $\mathcal{N}_{\mathcal{RC}}$, there exist one unit-cost row-net r_i for each row i and one unit-cost column-net c_j for each column j. The row-net r_i connects the vertices corresponding to the nonzeros in row i, and the column-net c_j connects the vertices corresponding to the nonzeros in column j. That is, $v_{ij} \in r_i$ and $v_{ij} \in c_j$ if and only if $a_{ij} \neq 0$.

III. Multithreaded algorithms for matrix scaling

The multithreaded implementations of Algorithm 1 differ in the way they store the matrix $\mathbf{A}$ and hence, in the way they compute the row and column norms. We use $\mathbf{d}_1[i]$ and $\mathbf{d}_2[j]$ to store the norm of row i and column j in the algorithms (instead of the notation $\mathbf{D}_1$ and $\mathbf{D}_2$) and call $\mathbf{d}_1$ and $\mathbf{d}_2$ as vectors; similarly $\mathbf{d}_r$ and $\mathbf{d}_c$ are used (instead of $\mathbf{D}_r$, $\mathbf{D}_c$).

As mentioned above, we do not update the entries of $\mathbf{A}$. Instead, for each nonzero a_{ij}, we compute the corresponding scaled nonzero $\mathbf{d}_r[i] \times a_{ij} \times \mathbf{d}_c[j]$. At each iteration, each scaled nonzero $\mathbf{d}_r[i] \times a_{ij} \times \mathbf{d}_c[j]$ needs to be added to $\mathbf{d}_1[i]$ and $\mathbf{d}_2[j]$. In a multithreaded setting, these write operations can result in race conditions on $\mathbf{d}_1$ or $\mathbf{d}_2$. With a naive approach, the threads can evade from write conflicts by using private $\mathbf{d}_1$s and/or $\mathbf{d}_2$s. Let τ be the number of the threads and $\mathbf{d}_1^t$ and $\mathbf{d}_2^t$ denote the private row and column norm arrays for a thread $t \in \{1, \ldots, \tau\}$, respectively.

In all the algorithms that we present, we distinguish three phases: **init**, **put**, and **get**. We will focus on these phases in which the algorithms compute the norms of the rows and columns of the iteration matrix $\widehat{\mathbf{A}}^{(k)}$. The implementation of other parts of the algorithms is almost the same—they are parallelized with OpenMP parallel-for directives. In the **init** phase, each thread t initializes its own private memory. In the **put** phase, the nonzeros of the matrix are processed. After reading a_{ij} from the memory and computing the corresponding scaled nonzero val, a thread t adds val to either shared $\mathbf{d}_1[i]$ or private $\mathbf{d}_1^t[i]$ (and similarly to $\mathbf{d}_2[j]$ or $\mathbf{d}_2^t[j]$). In the **get** phase, the values in private arrays are transferred from the private arrays to the shared array.

A naive approach which is suitable for the CRS format is given in Algorithm 2. While processing the nonzeros in Algorithm 2, a parallelization in the row-level is promising since the CRS format is used to store $\mathbf{A}$. That is, all the nonzeros in the ith row of $\mathbf{A}$ is processed by the same thread t which is the only thread writing to $\mathbf{d}_1[i]$. Thus there is no conflict and hence no need for a private memory for these writes. However, the accesses to $\mathbf{d}_2$ need to be synchronized carefully. In Algorithm 2, each thread t uses its private $\mathbf{d}_2^t$ array to store the partial norms of the columns and in **get**, these partial norms are added to the shared $\mathbf{d}_2$ array.

Algorithm 3 parallelizes the scaling process in nonzero-level when the matrix is stored in COO format. With this approach, in addition to the column norms, a row norm $\mathbf{d}_1[i]$ can also be updated by different threads at the same time. Hence, Algorithm 3 uses a private $\mathbf{d}_1^t$ array for each thread t to store the partial row norms. Besides, a thread cannot pack a row norm in a private variable sum^t while processing the nonzeros of a row. Similar to Algorithm 2, the partial row and column rows are computed in **put** phase and added back to the shared norm arrays in the **get** phase.

There are two overheads of conflict resolution by using

Algorithm 2: Simple parallel scaling with CRS

```
Input: A: n × n input matrix in CRS format
Output: d_r, d_c: row and column scaling vectors
for i = 1 to n in parallel do
    d_r[i] ← 1
    d_c[i] ← 1
while not converged do
    for i = 1 to n in parallel do
        d_1[i] ← 0
        d_2[i] ← 0
init
    for t = 1 to τ in parallel do
        for i = 1 to n do
            d_2^t[i] ← 0
put
    for i = 1 to n in parallel do
        ►t is the current thread id
        sum^t ← 0
        for each nonzero a_ij in row i do
            val ← d_r[i] × a_ij × d_c[j]
            add val to sum^t and d_2^t[j]
        d_1[i] ← sum^t
get
    for t = 1 to τ do
        for i = 1 to n in parallel do
            d_2[i] ← d_2[i] + d_2^t[i]
    error ← max(max_i(|1 − d_1[i]|), max_i(|1 − d_2[i]|))
    if error < ε then
        converged ← true
    else
        for i = 1 to n in parallel do
            d_r[i] ← d_r[i]/√d_1[i]
            d_c[i] ← d_c[i]/√d_2[i]
```

Algorithm 3: Simple parallel scaling with COO

```
Input: A: n × n input matrix in COO format
Output: d_r, d_c: row and column scaling vectors
...
while not converged do
    ...
init
    for t = 1 to τ in parallel do
        for i = 1 to n do
            d_1^t[i] ← 0
            d_2^t[i] ← 0
put
    for each nonzero a_ij in parallel do
        ►t is the current thread id
        val ← d_r[i] × a_ij × d_c[j]
        add val to d_1^t[i] and d_2^t[j]
get
    for t = 1 to τ do
        for i = 1 to n in parallel do
            d_1[i] ← d_1[i] + d_1^t[i]
        for i = 1 to n in parallel do
            d_2[i] ← d_2[i] + d_2^t[i]
    ...
```

private arrays: computation and memory. For the CRS based parallel algorithm, the total computation overhead due to **init** and **get** phases are $2\tau n$ at each iteration. For the COO based algorithm, this overhead is $4\tau n$. The total private memory used by these algorithms are τn and $2\tau n$, respectively.

A. Conflict resolution with partitioning for parallelization in the row-level

In Algorithm 2, each thread is given a set of rows during the execution of the scaling process. Hence, the rows are implicitly and dynamically partitioned among the threads by the scheduler. The decisions are given with respect to a scheduling policy without considering any metric or restriction other than load balancing. Here, we show that the memory and computation overhead can be reduced by modeling the problem as a hypergraph partitioning problem and using a static partition.

Let $\Pi = \{\mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_\tau\}$ be a partition of the row set $\{1, 2, \ldots, n\}$ in the column-net hypergraph model. Assume that thread t will process the nonzeros of the rows in $\mathcal{R}_t$. Hence, a conflict will occur only if there exists a column j such that a_{ij} and $a_{i'j}$ are nonzero and rows i and i' are in different parts. The number of such columns is equal to the number of external nets with respect to Π, i.e., $\chi_{cut}(\Pi)$. By using this fact, one can greatly reduce the size of private $\mathbf{d}_2$ arrays by partitioning the hypergraph with the metric $\chi_{cut}(\Pi)$ as in (2).

Given a partition Π, we first permute the rows and columns of the matrix with respect to Π. The rows are permuted in the partial order of $(\mathcal{R}_1\mathcal{R}_2 \cdots \mathcal{R}_\tau)$. The external columns are put before the other columns, for the rest of the columns, column j is put before column j' if j is internal to t, j' is internal to t' and $t < t'$.

Let us call a nonzero a_{ij} *external* if column j is in the cut. Otherwise, we say that a_{ij} is *internal*. We also call a row i internal if all a_{i*}s are internal. Otherwise, we call that row external. Let r_{ext} be the number of these external rows. Algorithm 4, CRS-Cut, parallelizes the scaling process in the row-level with static partitioning. To partition the rows among threads, we use τ pointers to keep the id of the first row of each part. The **init** and **get** phases are similar to the simple approach described in Algorithm 4. Note that the private array sizes are $\chi_{cut}(\Pi)$ instead of n. Hence, the total private memory used is $\tau\chi_{cut}(\Pi)$ and the total computation overhead is $2\tau\chi_{cut}(\Pi)$. To access to the internal/external rows in an efficient way, we separate them in the row lists of the parts. We then use an additional pointer set of size τ to keep the location of the first internal row for each part. Again for further efficiency, we partition the nonzero list of each external row into two by separating the internal and external nonzeros. We keep a set of pointers of size r_{ext} to store the location of the first internal nonzero in each list.

In the **init** and **get** phases of Algorithm 4, a private location $\mathbf{d}_2^t[j]$ is nonzero only if a nonzero a_{ij} exists such

Algorithm 4: Part. based scaling with CRS-Cut

Input: $\mathbf{A}$: $n \times n$ input matrix in CRS format and a partition $\Pi = \{\mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_\tau\}$ of rows
Output: $\mathbf{d}_r$, $\mathbf{d}_c$: row and column scaling vectors

```
...
while not converged do
    ...
init  for t = 1 to τ in parallel do
          for i = 1 to cut do
              d_2^t[i] ← 0
put   for t = 1 to τ in parallel do
          ►t is the current thread id
          for each external row i in R_t do
              sum^t ← 0
              for each external nonzero a_ij in row i do
                  val ← d_r[i] × a_ij × d_c[j]
                  add val to sum^t and d_2^t[j]
              for each internal nonzero a_ij in row i do
                  val ← d_r[i] × a_ij × d_c[j]
                  add val to sum^t and d_2[j]
              d_1[i] ← sum^t
          for each internal row i in R_t do
              sum^t ← 0
              for each nonzero a_ij in row i do
                  val ← d_r[i] × a_ij × d_c[j]
                  add val to sum^t and d_2[j]
              d_1[i] ← sum^t
get   for t = 1 to τ do
          for i = 1 to cut in parallel do
              d_2[i] ← d_2[i] + d_2^t[i]
    ...
```

that $i \in \mathcal{R}_t$. In hypergraph partitioning, this corresponds to the connectivity of net j with part t. For each column j, the number of nonzero $\mathbf{d}_2^t[j]$s is equal to its connectivity λ_j. Hence, at each iteration, the total number of nonzeros in private arrays is $\sum_{j=1}^{n} \lambda_j = \chi_{SOED}(\Pi)$ as defined in 4. By using the part-to-net connectivity information while traversing the private arrays in **init** and **put** phases, we propose a modified version of CRS-Cut as shown in Algorithm 5.

Algorithm 5, CRS-SOED, uses the same amount of private memory as CRS-Cut to store the partial column norms. To reduce the computation overhead, CRS-SOED also keeps the connectivity information for each part. The size of this additional information is $\tau + \chi_{SOED}(\Pi)$. Note that when $\tau = 2$, whole private memory will be traversed by CRS-SOED since each net in the cut is connected to both parts. Hence, considering its memory overhead over CRS-Cut, using CRS-SOED can be advantageous when $\tau > 2$.

B. Conflict resolution with partitioning for parallelization in the nonzero-level

When $\mathbf{A}$ is stored in COO format, a parallelization in the nonzero-level is more suitable as shown in Algorithm 3. In the simple approach, the scheduler assigns a set of nonzeros

Algorithm 5: Part. based scaling with CRS-SOED

Input: $\mathbf{A}$: $n \times n$ input matrix in CRS format and a partition $\Pi = \{\mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_\tau\}$ of rows
Output: $\mathbf{d}_r$, $\mathbf{d}_c$: row and column scaling vectors
...
while *not converged* **do**
  ...
init **for** $t = 1$ *to* τ **in parallel do**
    for each *external column* i *connected to* $\mathcal{R}_t$ **do**
      $\mathbf{d}_2^t[i] \leftarrow 0$
put ... ▶`As same as CRS-Cut`
get **for** $t = 1$ *to* τ **do**
    for each *external column* i *of* $\mathcal{R}_t$ **in parallel do**
      $\mathbf{d}_2[i] \leftarrow \mathbf{d}_2[i] + \mathbf{d}_2^t[i]$
  ...

Algorithm 6: Part. based scaling with COO-SOED

Input: $\mathbf{A}$: $n \times n$ input matrix in COO format and a partition $\Pi = \{\mathcal{Z}_1, \mathcal{Z}_2, \ldots, \mathcal{Z}_\tau\}$ of nonzeros
Output: $\mathbf{d}_r$, $\mathbf{d}_c$: row and column scaling vectors
...
while *not converged* **do**
  ...
init **for** $t = 1$ *to* τ **in parallel do**
    for each *external row* i *connected to* $\mathcal{Z}_t$ **do**
      $\mathbf{d}_1^t[i] \leftarrow 0$
    for each *external column* i *connected to* $\mathcal{Z}_t$ **do**
      $\mathbf{d}_2^t[i] \leftarrow 0$
put **for** $t = 1$ *to* τ **in parallel do**
    ▶t `is the current thread id`
    for *each row/col-external nonzero* a_{ij} *in* $\mathcal{Z}_t$ **do**
      $val \leftarrow \mathbf{d}_r[i] \times a_{ij} \times \mathbf{d}_c[j]$
      add val to $\mathbf{d}_1^t[i]$ and $\mathbf{d}_2^t[j]$
    for *each row-external nonzero* a_{ij} *in* $\mathcal{Z}_t$ **do**
      $val \leftarrow \mathbf{d}_r[i] \times a_{ij} \times \mathbf{d}_c[j]$
      add val to $\mathbf{d}_1^t[i]$ and $\mathbf{d}_2[j]$
    for *each col-external nonzero* a_{ij} *in* $\mathcal{Z}_t$ **do**
      $val \leftarrow \mathbf{d}_r[i] \times a_{ij} \times \mathbf{d}_c[j]$
      add val to $\mathbf{d}_1[i]$ and $\mathbf{d}_2^t[j]$
    for *each internal nonzero* a_{ij} *in* $\mathcal{Z}_t$ **do**
      $val \leftarrow \mathbf{d}_r[i] \times a_{ij} \times \mathbf{d}_c[j]$
      add val to $\mathbf{d}_1[i]$ and $\mathbf{d}_2[j]$
get **for** $t = 1$ *to* τ **do**
    for each *external row* i *conn. to* $\mathcal{Z}_t$ **in parallel do**
      $\mathbf{d}_1[i] \leftarrow \mathbf{d}_1[i] + \mathbf{d}_1^t[i]$
    for each *external col.* i *conn. to* $\mathcal{Z}_t$ **in parallel do**
      $\mathbf{d}_2[i] \leftarrow \mathbf{d}_2[i] + \mathbf{d}_2^t[i]$
  ...

to each thread during the execution of the scaling process. This is an implicit partitioning of the nonzeros among the threads where the assignments are done with respect to a scheduling policy by considering the load balance. Similar to the CRS-based versions, we show that COO-based storage can benefit from a static hypergraph partitioning but this time with the fine-grain model (see Section II-B).

Let $\Pi = \{\mathcal{Z}_1, \mathcal{Z}_2, \ldots, \mathcal{Z}_\tau\}$ be a partition of the nonzeros of the matrix $\mathbf{A}$. Let us call a nonzero a_{ij} *row/col-external* if both of its nets, which correspond to ith row and jth column, are external. If only row i (or column j) is external we call a_{ij} *row-external* (or *col-external*). Otherwise, we call a_{ij} internal. Algorithm 6, COO-SOED, uses a partition to distribute the nonzeros among threads when the matrix is in the COO format. Similar to the previous algorithms, for each external nonzero, the partial row and column norms are written to the private arrays. Otherwise, if the nonzero is internal, the shared norm arrays $\mathbf{d}_1$ and $\mathbf{d}_2$ are used. Hence, the number of private memory accesses in the **init** and **get** phases is equal to $\chi_{SOED}(\Pi)$. To distribute the work, given Π, we permute the nonzeros as in $(\mathcal{Z}_1 \mathcal{Z}_2 \cdots \mathcal{Z}_\tau)$ and use τ pointers to keep the location of the first nonzero of each part. For efficiency, we partition each $\mathcal{Z}_i$ into four to separate row/col-external, row-external, col-external, and internal nonzeros. For efficient access to these nonzero classes, we keep 3 additional pointers for each part to store the location of the first nonzero of each class.

C. Other approaches for shared-memory parallelization of scaling

Using private arrays is not the only way for conflict-free parallelization of the scaling process. A simple alternative is using locks and/or atomic operations for concurrency. The overheads of these approaches such as the number of locks can also be related with a partitioning metric. Consider a matrix stored in CRS format. Given a partition Π, the number of external nonzeros is equal to $\chi_{SOED}(\Pi)$ given in 2 where the cost of each net is equal to the number of nonzeros in the corresponding column. Note that the same is also true for the COO format and parallelization in the nonzero-level. Hence, if one uses an atomic operation for each external nonzero the number of atomic operations is equal to $\chi_{SOED}(\Pi)$.

For a further reduction on the number of nonzeros in private arrays, the algorithms can be modified as follows. We assumed that an external net is external to all parts. However, such nets can be assumed to be internal to only one part which permits only one thread to update the corresponding shared norm value in the **put** phase. Hence, there will be no conflicts in this phase. When this implementation is used the number of private memory accesses **init** and **get** phases will be equal to $\chi_{conn}(\Pi)$ given in (3). Although this yields to a reduction on the computational overhead for parallelization, it may have an undesirable effect. For example, in the current implementation of CRS-SOED and COO-SOED, the threads do not need to synchronize between **put** and **get** phases since the sets of shared memory locations accessed in these phases are disjoint. Hence, one can use the **nowait** directive of OpenMP in the implementation to avoid

a thread synchronization after the **put** phase. However, if the $\chi_{conn}(\Pi)$ metric and the related implementation is used, a synchronization is necessary since the sets of memory locations will not be disjoint anymore.

IV. Experimental Results

We used two different architectures to test the algorithms. The first one has a 2.27GHz dual quad-core Intel Xeon (Bloomfield) CPU and 48GB main memory. Each core in a socket has 32KB L1 and 256KB L2 caches. Each socket has an 8MB L3 cache shared by 4 cores. The second architecture has a dual quad-core AMD Opteron (Shanghai) processor and 32GB main memory. Each core in a socket has 64KB L1 and 512KB L2 caches. Each socket has a 6MB L3 cache shared by 4 cores. All of the algorithms are implemented in C and OpenMP. The compiler is icc version 12.0 and 11.1 for the first and the second architecture, respectively, and -O3 optimization flag is used. None of our optimization methods is architecture specific. We therefore expect similar behavior from the algorithms on the two architectures.

We used a set of real life matrices from different application domains that are available at the University of Florida (UFL) Sparse Matrix Collection (http://www.cise.ufl.edu/research/sparse/matrices). The properties of the matrices used in the experiments are given in Table I.

Table I
Properties of the matrices used in the experiments.

Matrix	n	Z	Avg. deg
atmosmodd	1,270,432	8,814,880	6.94
atmosmodl	1,489,752	10,319,760	6.93
cage14	1,505,785	27,130,349	18.02
Chebyshev4	68,121	5,377,761	78.94
Hamrle3	1,447,360	5,514,242	3.81
NotreDame	325,729	929,849	2.85
pre2	659,033	5,834,044	8.85
rajat21	411,676	1,876,011	4.56
rajat30	643,994	6,175,244	9.59
Stanford_Berk.	683,446	7,583,376	11.10
torso1	116,158	8,516,500	73.32
trans5	116,835	749,800	6.42

In the experiments, we do not compute the error on the norm values of the scaled matrix after each iteration. Instead, we iterate the process for 100 times to measure the execution time. To generate the partitions used by the algorithms, we used PaToH [26] with the appropriate objective for each algorithm and imbalance ratio 0.05.

We tested the validity of our motivation for minimizing the objectives χ_{cut} and χ_{SOED} as follows. For each τ, we partitioned the rows/nonzeros of the matrices with a load-balancing heuristic. To do this, by mimicking the *longest processing time first* rule of Graham [28], we first sort the rows in decreasing order by number of their nonzeros. We then visit each row in this order and assign it to the part with the least nonzeros assigned so far. For CRS based algorithms, we measured that the imbalances are always smaller than 10^{-5} for all of the matrices and τ values except when trans5 is partitioned to 8 threads. For COO based algorithms, it is clear that the heuristic partitions the nonzeros in the best way possible. Hence, the partitions we obtain provide almost perfect load balancing among the threads. After running a scaling algorithm with this partition and measuring the execution time, we do a normalization by dividing it to the time spent for scaling the same matrix with the partition generated by PaToH with the appropriate metric. The comparisons among the running time would therefore show that the difference in the practical execution times is related to the optimized cutsize metrics. Figure 1 shows the averages of these normalized values.

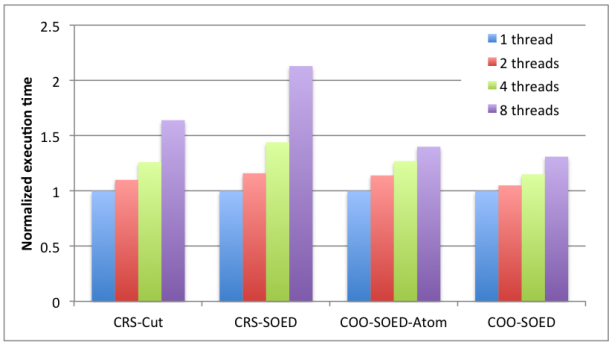

Figure 1. Average normalized execution times of the parallel algorithms using a perfectly balanced partition. For each matrix-τ pair, the execution time is normalized by dividing it to the execution time of the algorithm with the partition generated by PaToH. The figure shows that the cut-size minimized partitions improve the performance.

Algorithm 7: COO-SOED-Atom: **get**

for $t = 1$ *to* τ **in parallel do**
  for each *external row* i *conn. to* $\mathcal{Z}_t$ **do**
    (atomic) $\mathbf{d}_1[i] \leftarrow \mathbf{d}_1[i] + \mathbf{d}_1^t[i]$
  for each *external column* i *conn. to* $\mathcal{Z}_t$ **do**
    (atomic) $\mathbf{d}_2[i] \leftarrow \mathbf{d}_2[i] + \mathbf{d}_2^t[i]$

As Figure 1 shows, the algorithms perform bad when they use the balanced-only partitions instead of the ones generated by PaToH. For CRS-Cut, CRS-SOED and COO-SOED, the ratio of the performances are 1.64, 2.13, and 1.31 for 8 threads. These ratios show that minimizing the cut is an effective approach for sparse matrix computations in shared-memory architectures. For this experiment, we also implemented an experimental variant of COO-SOED by using atomic operations. This variant, COO-SOED-Atom, differs from COO-SOED only in the **get** phase. The implementation of this phase is given in Algorithm 7. As expected, COO-SOED-Atom is slower than COO-SOED in practice since atomic operations are costlier. In Figure 1, it can be seen that the normalized performance of COO-SOED-Atom with

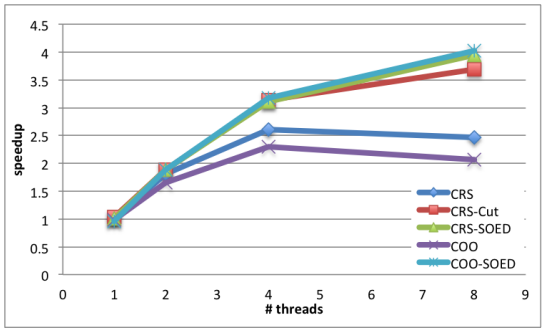

Figure 2. Average speedups of scaling algorithms on the Intel-based architecture. For CRS-based algorithms, the speedups are computed by using the execution time of the CRS-based sequential algorithm. For the COO-based ones, we used the COO-based sequential algorithm.

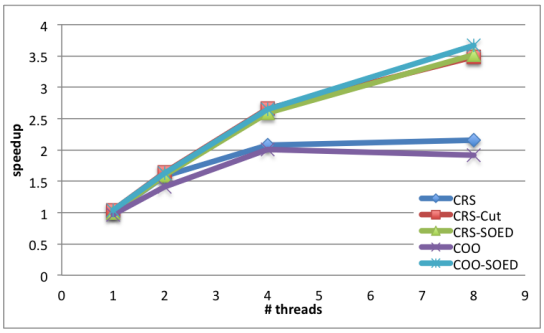

Figure 3. Average speedups of scaling algorithms on the AMD-based architecture. For CRS-based algorithms, the speedups are computed by using the execution time of the CRS-based sequential algorithm. For the COO-based ones, we used the COO-based sequential algorithm.

balanced partitions is worse (9%, 9%, and 6% more for 2, 4, and 8 threads, respectively) than those of COO-SOED with balanced partition. Since, the rest of the algorithms is exactly the same, we can argue that when PaToH partition is used instead of balanced partition, the reduction on COO-SOED-Atom's atomic operations improves the runtime more than COO-SOED. Note that for a partition Π, the number of atomic operations is equal to $\chi_{SOED}(\Pi)$. Thus, Figure 1 shows that minimizing the cut-size is helpful in practice.

Figures 2 and 3 show the average speedup values for the Intel-based and AMD-based architectures, respectively. In the figures, CRS and COO correspond to OpenMP versions of Algorithms 2 and 3 with *guided* scheduling policy and *chunksize* 50. These two algorithms do not actually scale after 4 threads. When τ is increased to 8, the execution time of these algorithms also increase. This can be expected especially for matrices with a few number of nonzeros per row/column. Note that in simple CRS- and COO-based algorithms, each additional thread decreases the number of nonzeros per thread but also requires an additional private memory of size $\Theta(n)$. In addition to the computation overhead due to **init** and **get** phases, the pressure on the cache will increase in the **put** phase. As Table II shows, the execution time of COO-Simple is decreased only for three matrices when τ is increased from 4 to 8. These matrices, cage14, Chebyshev4, and torso1 have 18.02, 78.94, and 73.32 nonzeros per row/column on the average (they are the top three matrices with respect to this criteria). For the remaining 9 matrices, we compute the increase in the execution time and pair these values with the average numbers of nonzeros. The scatter plot for these pairs is given in Figure 4. As the figure shows, there is an inverse correlation between the average number of nonzeros of each matrix and performance decrease when 8 threads are used.

Our parallel scaling algorithms based on static-partitioning obtain much better speedups compared with the simple approaches. In the Intel-based architecture, Figure 2, for 8 threads, the average speedups obtained by CRS-Cut,

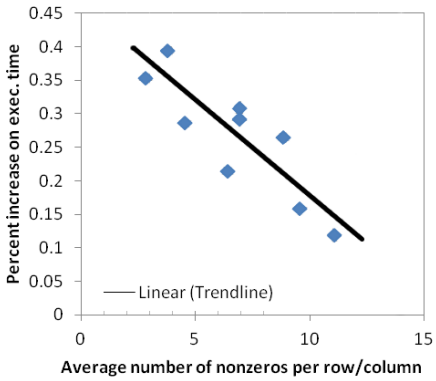

Figure 4. Scatter plot of the matrices for which an increase on the execution time of COO-Simple is observed when τ is increased from 4 to 8. Each matrix is represented by an (x, y) in the plot where x is the average number of nonzeros per row/column in the matrix and y is the percentage of the increase on the execution time of COO-Simple.

CRS-SOED, and COO-SOED are 3.69, 3.95, and 4.02, respectively. For the AMD-based architecture, Figure 3, these values are 3.49, 3.52, and 3.67. Although we are not aware of any multithreaded scaling algorithm, there exist several SpMxV algorithms designed for shared-memory architectures. In a recent one, the architectures used are similar to the ones in this paper [8]. Buluç et al. obtained average speedups between 3-3.5 for 8 threads. They also observed that on an Intel Harpertown architecture, which is a dual-socket quad-core system with 12MB of L2 cache, the average Mflops/sec on 8 threads is only 3.5% higher than on 4 processors. Similar to our experiments, the Mflops/sec even decreased for some matrices when τ is increased from 4 to 8. Note that in SpMxV, there is one input and one output vector. On the other hand, for scaling we have two of each. Hence the scaling problem can be considered as hard as SpMxV, or even more harder.

We used different sequential algorithms while computing the speedups for CRS-based algorithms and COO-SOED. Hence the speedup results do not imply that COO-SOED is

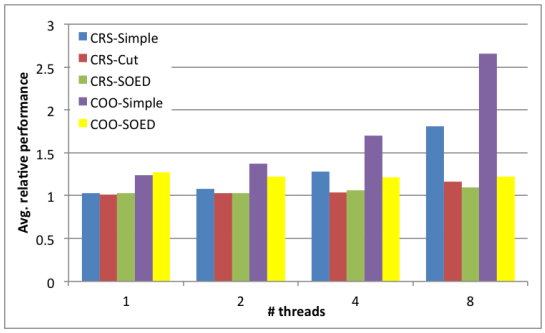

Figure 5. Average relative performances of parallel scaling algorithms on the Intel-based architecture. The relative performance of each algorithm on a particular problem is computed by dividing the average execution time of the algorithm by the best average time for the same problem.

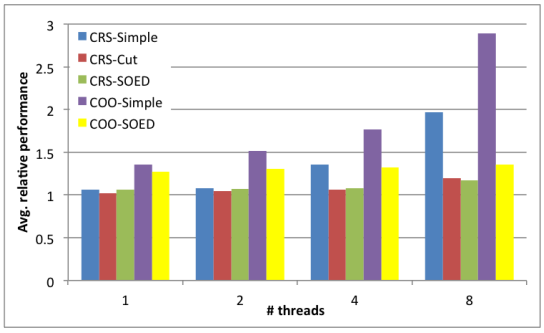

Figure 6. Average relative performances of parallel scaling algorithms on the AMD-based architecture. The relative performance of each algorithm on a particular problem is computed by dividing the average execution time of the algorithm by the best average time for the same problem.

the fastest parallel scaling algorithm in this paper. As the relative performances in Figures 5 and 6, and the execution times in Table II show, this is not true. The figures show the average relative performance of each algorithm for 1, 2, 4, and 8 threads. The relative performance is computed as follows: for each matrix, the execution time of the algorithm is divided to that of the best one. That is, for each matrix the best algorithm is considered as 1 and the other algorithms are evaluated with that value. The averages of these values are computed per each τ value. As Figures 5 and 6 show, COO-based algorithms are slower than the CRS-based ones. This is expected since the CRS format implicitly solves the conflict problem while computing the row norms. Furthermore, in a CRS-based algorithm, the partial results can be hold in a private variable (sum^t in the algorithms) until the final result is obtained. With this approach each location in $\mathbf{d}_1$ is accessed only once at each iteration.

For both of the architectures CRS-Cut is better than CRS-SOED on the average when τ is 1, 2, or 4. As mentioned before, the when τ is 2, CRS-SOED cannot do any reductions. However, its additional memory overhead will still be there. When τ increases to 8, the reduction on **init** and **get** phases starts to show itself and it compensates the overhead of CRS-SOED due to the storage of connectivity information. This experiment shows that minimizing $\chi_{SOED}(\Pi)$ can be promising when τ is large.

V. Conclusion and discussion

We discussed the relevance of hypergraph partitioning-based methods to minimize the parallelization overhead associated with the standard OpenMP parallelization techniques. In particular, we have argued that three common objective functions in the hypergraph partitioning problem correspond exactly to minimizing the parallelization overhead. We note that the use of objective functions of cut-net and connectivity-1 metrics correspond closely to their use in distributed memory parallelization problem. On the other hand, the use of the SOED metric in the shared memory parallelization, to the best of knowledge, does not have such a correspondence.

We have achieved significant speedups by making use of the discussed parallelization techniques. It is assumed that the parallelization would be performed using only OpenMP directives. Therefore, we did not consider hardware aware optimizations techniques (these can help for further performance optimization). Optimizing the codes that use those techniques would require revisiting the models and encapsulate the techniques. Although we believe this is a viable task, it can also be rather intricate, especially for architecture aware optimizations and partitioning which require mapping and partitioning to be performed together.

We did not put a lot of effort to optimize the the cache efficiency of a single thread. The methods proposed for optimizing cache performance in SpMxV operations can also be used to reorganize computations of a single thread (see [29, Chapter 5]). Here, the issues of partitioning for threads and optimizing a single thread's performance are separate issues and therefore they can be used together.

Acknowledgment

This work was supported in parts by the DOE grant DE-FC02-06ER2775 and by the NSF grants CNS-0643969, OCI-0904809, and OCI-0904802.

References

[1] B. Chapman, G. Jost, and R. van der Pas, *Using OpenMP: Portable Shared Memory Parallel Programming*. Cambridge, MA, London, England: The MIT Press, 2007.

[2] R. H. Bisseling and I. Flesch, "Mondriaan sparse matrix partitioning for attacking cryptosystems by a parallel block Lanczos algorithm–A case study," *Parallel Comput.*, vol. 32, pp. 551–567, 2006.

[3] Ü. V. Çatalyürek and C. Aykanat, "Hypergraph-partitioning based decomposition for parallel sparse-matrix vector multiplication," *IEEE T. Parall. Distr.*, vol. 10, pp. 673–693, 1999.

[4] Ü. V. Çatalyürek, C. Aykanat, and B. Uçar, "On two-dimensional sparse matrix partitioning: Models, methods, and a recipe," *SIAM J. Sci. Comput.*, vol. 32, pp. 656–683, 2010.

Table II
AVERAGE RUNNING TIMES OF 100 EXECUTIONS OF THE ALGORITHMS IN SECONDS FOR 100 ITERATIONS.

		CRS			COO	
Matrix	T	Simp.	Cut	SOED	Simp.	SOED
atmosmodd	1	5.54	5.37	5.55	6.72	6.71
	2	3.20	2.90	2.85	4.25	3.38
	4	2.45	1.82	1.84	3.53	2.14
	8	2.76	1.42	1.55	4.62	2.04
atmosmodl	1	6.95	6.71	6.93	8.33	8.27
	2	3.83	3.65	3.56	5.21	4.30
	4	3.03	2.13	2.28	4.27	2.70
	8	3.27	1.72	1.93	5.52	2.13
cage14	1	17.80	17.38	18.41	21.04	22.94
	2	9.10	9.77	9.70	11.97	10.87
	4	5.65	5.88	5.96	7.70	6.44
	8	4.93	5.70	4.59	7.69	5.20
Chebyshev4	1	1.93	1.91	1.92	2.86	3.16
	2	1.22	0.95	1.04	1.55	1.60
	4	1.10	0.67	0.65	0.88	0.79
	8	1.23	0.56	0.55	0.79	0.65
Hamrle3	1	6.13	5.96	6.19	6.76	6.62
	2	3.50	3.30	3.33	4.49	3.48
	4	2.67	1.86	2.01	3.53	2.50
	8	2.93	1.69	1.42	4.92	1.80
NotreDame	1	1.18	1.09	1.06	1.17	1.03
	2	0.68	0.60	0.60	0.69	0.64
	4	0.47	0.34	0.31	0.65	0.39
	8	0.57	0.27	0.20	0.88	0.25
pre2	1	4.02	3.99	4.03	4.67	4.84
	2	2.23	2.05	2.05	2.75	2.40
	4	1.39	1.23	1.17	1.96	1.48
	8	1.45	0.88	0.92	2.48	1.24
rajat21	1	1.90	1.83	1.88	2.04	2.03
	2	1.08	1.02	1.01	1.25	1.04
	4	0.74	0.58	0.60	1.01	0.70
	8	0.79	0.59	0.52	1.30	0.58
rajat30	1	4.14	4.20	4.33	4.75	5.04
	2	2.33	2.29	2.48	2.86	2.54
	4	1.61	1.66	1.61	2.08	1.51
	8	1.80	1.41	1.36	2.41	1.32
Stanford_Berk.	1	4.04	3.97	4.01	4.49	4.53
	2	2.24	2.22	2.08	2.69	2.39
	4	1.40	1.15	1.17	1.93	1.40
	8	1.46	0.93	0.83	2.16	1.13
trans5	1	0.57	0.57	0.58	0.65	0.66
	2	0.31	0.34	0.33	0.36	0.32
	4	0.20	0.19	0.21	0.28	0.17
	8	0.25	0.19	0.20	0.34	0.11
torso1	1	2.59	2.60	2.60	4.38	4.64
	2	1.32	1.38	1.43	2.25	2.29
	4	0.85	0.94	0.94	1.40	1.21
	8	0.78	0.78	0.78	1.22	1.01

[5] Ü. V. Çatalyürek, B. Uçar, and C. Aykanat, "Hypergraph partitioning," in *Encyclopedia of Parallel Computing*, 2011, pp. 871–881.

[6] B. Uçar and C. Aykanat, "Revisiting hypergraph models for sparse matrix partitioning," *SIAM Rev.*, vol. 49, pp. 595–603, 2007.

[7] B. Vastenhouw and R. H. Bisseling, "A two-dimensional data distribution method for parallel sparse matrix-vector multiplication," *SIAM Rev.*, vol. 47, pp. 67–95, 2005.

[8] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in *SPAA'09*, 2009, pp. 233–244.

[9] M. Shantharam, A. Chatterjee, and P. Raghavan, "Exploiting dense substructures for fast sparse matrix vector multiplication," *Int. J. High Perform. C.*, vol. 25, pp. 328–341, 2011.

[10] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," *Parallel Comput.*, vol. 35, pp. 178–194, 2009.

[11] H. Kotakemori, H. Hasegawa, T. Kajiyama, A. Nukada, R. Suda, and A. Nishida, "Performance evaluation of parallel sparse matrix-vector products on SGI Altix3700," in *IWOMP'05/IWOMP'06*, 2008, pp. 153–163.

[12] T. Wu, B. Wang, Y. Shan, F. Yan, Y. Wang, and N. Xu, "Efficient PageRank and SpMV computation on AMD GPUs," in *ICPP'10*, 2010, pp. 81–89.

[13] A. N. Yzelman and R. H. Bisseling, "Cache-oblivious sparse matrix–vector multiplication by using sparse matrix partitioning methods," *SIAM J. Sci. Comput.*, vol. 31, pp. 3128–3154, 2009.

[14] ——, "Two-dimensional cache-oblivious sparse matrix-vector multiplication," *Parallel Comput.*, vol. 37, pp. 806–819, 2011.

[15] B. Uçar and C. Aykanat, "A library for parallel sparse matrix-vector multiplies," Department of Computer Engineering, Bilkent University, Ankara, Turkey, Tech. Rep. BU-CE-0506, 2005.

[16] K. Akbudak, E. Kayaaslan, and C. Aykanat, "Hypergraph-partitioning-based models and methods for exploiting cache locality in sparse-matrix vector multiplication," *CoRR*, vol. abs/1202.3856, 2012.

[17] I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct Methods for Sparse Matrices*. London: Oxford University Press, 1986.

[18] D. Ruiz, "A scaling algorithm to equilibrate both rows and columns norms in matrices," Rutherford Appleton Laboratory, Oxon, UK, Tech. Rep. RAL-TR-2001-034, 2001.

[19] D. Ruiz and B. Uçar, "A symmetry preserving algorithm for matrix scaling," INRIA, Tech. Rep. RR-7552, 2011.

[20] P. Amestoy, A. Buttari, I. S. Duff, A. Guermouche, J.-Y. L'Excellent, and B. Uçar, "MUMPS," in *Encyclopedia of Parallel Computing*, 2011, pp. 1232–1238.

[21] P. R. Amestoy, I. S. Duff, D. Ruiz, and B. Uçar, "A parallel matrix scaling algorithm," in *VECPAR 2008*. Springer Berlin / Heidelberg, 2008, pp. 301–313.

[22] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*. Chichester, U.K.: Willey–Teubner, 1990.

[23] C. J. Alpert and A. B. Kahng, "Recent directions in netlist partitioning: A survey," *Integration*, vol. 19, pp. 1–81, 1995.

[24] G. Karypis and V. Kumar, "Multilevel k-way hypergraph partitioning," *VLSI Des.*, vol. 11, pp. 285–300, 2000.

[25] I. Yamazaki, X. S. Li, F.-H. Rouet, and B. Uçar, "Combinatorial problems in a parallel hybrid linear solver," Dept. Comp. Sci., RWTH Aachen Univ., Tech. Rep. AIB 2011-09, 2011.

[26] Ü. V. Çatalyürek and C. Aykanat, *PaToH: A Multilevel Hypergraph Partitioning Tool, Version 3.0*, Bilkent University, Department of Computer Engineering, Ankara, 06533 Turkey. Available at http://bmi.osu.edu/ umit/software.htm, 1999.

[27] ——, "A fine-grain hypergraph model for 2D decomposition of sparse matrices," in *IPDPS 15*, San Francisco, CA, 2001.

[28] R. L. Graham, "Bounds on multiprocessing timing anomalies," *SIAM J. Appl. Math.*, vol. 17, pp. 416–429, 1969.

[29] A.-J. N. Yzelman, "Fast sparse matrix-vector multiplication by partitioning and ordering," Ph.D. dissertation, Utrecht University, 2011.