A Parallel Matrix Scaling Algorithm[⋆]

Patrick R. Amestoy[1], Iain S. Duff[2,3], Daniel Ruiz[1], and Bora Uçar[3]

[1] ENSEEIHT-IRIT, 2 rue Camichel, 31071, Toulouse, France
amestoy@enseeiht.fr, Daniel.Ruiz@enseeiht.fr
[2] Atlas Centre, RAL, Oxon, OX11 0QX, England
i.s.duff@rl.ac.uk
[3] CERFACS, 42 Av. G. Coriolis, 31057, Toulouse, France
duff@cerfacs.fr, ubora@cerfacs.fr

Abstract. We recently proposed an iterative procedure which asymptotically scales the rows and columns of a given matrix to one in a given norm. In this work, we briefly mention some of the properties of that algorithm and discuss its efficient parallelization. We report on a parallel performance study of our implementation on a few computing environments.

Key words: sparse matrices; matrix scaling; equilibration; parallel computing

1 Introduction

Scaling a matrix consists of pre- and post-multiplying the original matrix by two diagonal matrices. We consider the following scaling problem: given a large, sparse matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, find two positive diagonal matrices $\mathbf{D}_1$ and $\mathbf{D}_2$ so that all rows and columns of the scaled matrix $\widehat{\mathbf{A}} = \mathbf{D}_1\mathbf{A}\mathbf{D}_2$ have the same magnitude in some norm. Two common choices for the norm are the ∞- and the 1-norm. Recently, we proposed an iterative algorithm for this purpose [16]. In this paper, we present the algorithm briefly and discuss how we parallelize it. We report experimental results with the parallel code on three parallel systems that have different processors and interconnection networks.

Scaling or equilibration of data for linear systems of equations is a topic of great importance that has already been the subject of several scientific publications, with many different developments depending on the properties required from the scaling. It has given rise to several well known algorithms; see, for example, [10, 17]. If we denote by $\widehat{\mathbf{A}}$ the scaled matrix $\widehat{\mathbf{A}} = \mathbf{D}_1\mathbf{A}\mathbf{D}_2$, we then solve the equation $\widehat{\mathbf{A}}\hat{\mathbf{x}} = \hat{\mathbf{b}}$, where $\hat{\mathbf{x}} = \mathbf{D}_2^{-1}\mathbf{x}$ and $\hat{\mathbf{b}} = \mathbf{D}_1\mathbf{b}$.

A standard and well known approach to scaling is to do a row or column scaling. For row scaling, each row in the original matrix is divided by the norm of the row (using different norms, such as the ∞-norm or the 1-norm, depending

[⋆] This work was supported by "Agence Nationale de la Recherche" through Solstice project ANR-06-CIS6-010.

on the application). Column scaling is identical to row scaling, except that it considers the columns of the original matrix. A different approach that considers the matrix entries more globally is the one used in the HSL [13] routine `MC29`, which aims to make the nonzeros of the scaled matrix close to one by minimizing the sum of the squares of the logarithms of the moduli of the nonzeros in the scaled matrix; see [7]. `MC29` reduces this sum in a global sense and therefore should be useful on a wide range of sparse matrices. There is also the routine `MC30` in HSL that is a variant of the `MC29` routine for symmetric matrices. Scaling can also be combined with permutations; see [11] and the HSL routine `MC64`. In this approach, the matrix is first permuted so that the product of absolute values of entries on the diagonal of the permuted matrix is maximized (other measures such as maximizing the minimum element are also options). Then the matrix is scaled so that the diagonal entries are one and the off-diagonals are less than or equal to one. This provides a useful preprocessing tool for pivoting for sparse direct solvers, as well as for building good preconditioners for an iterative method.

A good scaling will normally improve (i.e., reduce) the condition number of the matrix. Although this is not the whole story, for example in determining the efficacy of scaling for direct methods, this is a metric that we will later use to compare scaling algorithms.

The scaling algorithm and some of its properties are introduced in Section 2. We discuss our parallelization approach in Section 3. Section 4 contains the experimental results.

2 The algorithm

Consider a general $m \times n$ real matrix $\mathbf{A}$, and denote by $\mathbf{r}_i = \mathbf{a}_{i\cdot}^T \in \mathbb{R}^{n\times 1}$, $i = 1, \ldots, m$, the row-vectors from $\mathbf{A}$ and by $\mathbf{c}_j = \mathbf{a}_{\cdot j} \in \mathbb{R}^{n\times 1}$, $j = 1, \ldots, n$, the column-vectors from $\mathbf{A}$. Denote by $\mathbf{D}_R$ and $\mathbf{D}_C$ the $m \times m$ and $n \times n$ diagonal matrices given by:

$$\mathbf{D}_R = \operatorname{diag}\left(\sqrt{\|\mathbf{r}_i\|_\infty}\right)_{i=1,\ldots,m} \quad \text{and} \quad \mathbf{D}_C = \operatorname{diag}\left(\sqrt{\|\mathbf{c}_j\|_\infty}\right)_{j=1,\ldots,n} \tag{1}$$

where $\|\cdot\|_\infty$ stands for the ∞-norm of a real vector (that is the maximum entry in absolute value; sometimes called the max-norm). If a row (or a column) in $\mathbf{A}$ has all entries equal to zero, we replace the diagonal entry in $\mathbf{D}_R$ (or $\mathbf{D}_C$ respectively) by 1. In the following, we will assume that this does not happen, considering that such cases are fictitious in the sense that zero rows or columns should be taken away and the system reduced.

We then scale matrix $\mathbf{A}$ on both sides, forming the scaled matrix $\widehat{\mathbf{A}}$ in the following way

$$\widehat{\mathbf{A}} = \mathbf{D}_R^{-1}\mathbf{A}\mathbf{D}_C^{-1} \, . \tag{2}$$

The idea of the proposed algorithm is to iterate that process, resulting in Algorithm 1. Convergence is obtained when

$$\max_{1\leq i\leq m}\left\{|(1-\|\mathbf{r}_i^{(k)}\|_\infty)|\right\} \leq \varepsilon \text{ and } \max_{1\leq j\leq n}\left\{|(1-\|\mathbf{c}_j^{(k)}\|_\infty)|\right\} \leq \varepsilon \qquad (3)$$

for a given value of $\varepsilon > 0$. We have shown [16] that the algorithm has fast linear convergence with an asymptotic rate of 1/2.

Algorithm 1 Simultaneous row and column iterative scaling in ∞-norm

```
1: $\mathbf{D}_1^{(0)} = \mathbf{I}_m$
2: $\mathbf{D}_2^{(0)} = \mathbf{I}_n$
3: for $k = 0, 1, 2, \ldots$ until convergence do
4:     $\mathbf{D}_R = \text{diag}\left(\sqrt{\|\mathbf{r}_i^{(k)}\|_\infty}\right)_{i=1,\ldots,m}$
5:     $\mathbf{D}_C = \text{diag}\left(\sqrt{\|\mathbf{c}_j^{(k)}\|_\infty}\right)_{j=1,\ldots,n}$
6:     $\mathbf{D}_1^{(k+1)} = \mathbf{D}_1^{(k)}\,\mathbf{D}_R^{-1}$
7:     $\mathbf{D}_2^{(k+1)} = \mathbf{D}_2^{(k)}\,\mathbf{D}_C^{-1}$
8:     $\widehat{\mathbf{A}}^{(k+1)} = \mathbf{D}_1^{(k+1)}\mathbf{A}\mathbf{D}_2^{(k+1)}$
```

For nonnegative square matrices, using the 1-norm, instead of the ∞-norm in lines 4 and 5 results in a scaling algorithm for the 1-norm, i.e., in the scaled matrix, the 1-norm of each row and column is asymptotically equal to 1. Convergence in the 1-norm case of both $\mathbf{A}^{(k)}$ and $\mathbf{D}_1^{(k)}$ and $\mathbf{D}_2^{(k)}$ is guaranteed for nonnegative matrices with total support—a square matrix is said to have total support if all entries can appear in some zero-free diagonal after row or column permutations. If a matrix does not have total support but just support (i.e., there exists a zero-free diagonal after row or column permutations), then the algorithm converges only for the $\mathbf{A}^{(k)}$ iterates; see [16] for details. We have observed in practical experiments that convergence for the 1-norm is fast for matrices with total support; for matrices with support but without total support, some entries should asymptotically go to zero, and a painfully slow convergence can be observed. Rothblum et al. have shown [15, page 13] that the problem of scaling a matrix $\mathbf{A}$ in the l_p-norm, $1 < p < \infty$ can be reduced to the problem of scaling in the 1-norm the pth Hadamard power of $\mathbf{A}$, i.e.,the matrix $\mathbf{A}^{[p]} = [a_{ij}^p]$. We applied that discussion to Algorithm 1 by replacing the matrix $\mathbf{A}$ with $\mathbf{A}^{[p]}$ and by taking the Hadamard pth root, e.g., $\mathbf{D}_1^{[1/p]} = [d_{ii}^{1/p}]$, of the resulting iterates. Hence, we argue that all of the convergence results that hold for the 1-norm hold for any of the l_p norms for $1 < p < \infty$.

We emphasize that the proposed iterative scaling procedure preserves the symmetry of the original matrix. If the given matrix $\mathbf{A}$ is symmetric, then the diagonal matrices $\mathbf{D}_R$ and $\mathbf{D}_C$ in Eq. (1) are equal and, consequently, matrix $\widehat{\mathbf{A}}$ in Eq. (2) is symmetric, as is the case for the matrices $\widehat{\mathbf{A}}^{(k)}$ at any iteration in

Algorithm 1. This is not the case for most scaling algorithms which alternately scale rows followed by columns or vice-versa.

In the case of unsymmetric matrices, one may consider the use of the Sinkhorn-Knopp iterations [18] with the ∞-norm in place of the 1-norm. This method simply normalizes all rows and then columns in $\mathbf{A}$, and iterates on this process until convergence. In the ∞-norm, this is obtained after a single step. Because of its simplicity, this method is very appealing. Notice, however, that the Sinkhorn-Knopp iteration may provide very different results when applied to $\mathbf{A}$ or $\mathbf{A}^T$. On the contrary, Algorithm 1 provides exactly the same results when applied to $\mathbf{A}$ or $\mathbf{A}^T$ in the sense that the scaled matrix obtained from $\mathbf{A}^T$ is the transpose of that obtained from $\mathbf{A}$. Another related property of Algorithm 1 is that it is independent of matrix permutations. In other words, the scaling factors of the permuted matrix are equivalent to the permuted scaling factors of the original matrix.

3 Parallelization

Algorithm 1 involves the scaled matrix $\widehat{\mathbf{A}}^{(k)}$, the original matrix $\mathbf{A}$, the two scaling (diagonal) matrices $\mathbf{D}_1^{(k)}$ and $\mathbf{D}_2^{(k)}$, and two temporary (diagonal) matrices $\mathbf{D}_R$ and $\mathbf{D}_C$ to compute the next iterates. To reduce the memory requirements, it is advisable not to store the scaled matrix $\widehat{\mathbf{A}}^{(k)} = \mathbf{D}_1^{(k)}\mathbf{A}\mathbf{D}_2^{(k)}$ explicitly; an individual matrix entry $a_{ij}^{(k)}$ at iteration k can be computed using $d_1^{(k)}(i) \times |a_{ij}| \times d_2^{(k)}(j)$, where $d_1^{(k)}(i)$ and $d_2^{(k)}(j)$ correspond to the ith and jth diagonal entries of the respective scaling matrices. Therefore, a parallelization of the algorithm on distributed memory processors necessitates the distribution of the matrices $\mathbf{A}$, $\mathbf{D}_1$, $\mathbf{D}_2$, $\mathbf{D}_R$ and $\mathbf{D}_C$. Observe that $\mathbf{D}_1$ and $\mathbf{D}_2$ are kept and updated at each iteration, whereas $\mathbf{D}_R$ and $\mathbf{D}_C$ are computed afresh at every iteration.

Assume that the matrix $\mathbf{A}$ is distributed among P processors. At this point we do not assume a particular distribution. Rather, we deal with the most general case in which each processor holds a set of nonzeros a_{ij} along with the corresponding row and column indices, i.e., each processor holds a set of triplets of the form $\langle i, j, a_{ij}\rangle$. We use $a_{ij} \in p$ to denote that the processor p has the nonzero a_{ij}. At each iteration, we first compute the contribution to the matrices $\mathbf{D}_R$ and $\mathbf{D}_C$ on each processor, using $\mathbf{D}_R^p$ and $\mathbf{D}_C^p$—the latter two matrices denote the matrices belonging to the processor p such that $d_R^p(i)$ and $d_C^p(j)$ denote the contributions of processor p to $d_R(i)$ and $d_C(j)$, respectively. These two matrices are then reduced to update the diagonal matrices $\mathbf{D}_1$ and $\mathbf{D}_2$ that are distributed among the processors; i.e., the partial results $d_R^p(i)$ and $d_C^p(j)$ should be combined at certain processors according to the partitions on $\mathbf{D}_1$ and $\mathbf{D}_2$. Hence, our problem reduces to partitioning the diagonal matrices $\mathbf{D}_1$ and $\mathbf{D}_2$ for a given partition on $\mathbf{A}$ in order to have an efficient parallelization of Algorithm 1. The most common communication cost metric addressed in similar parallelization problems is the total communication volume. Therefore our aim is to find partitions on $\mathbf{D}_1$ and $\mathbf{D}_2$ for a given partition on $\mathbf{A}$ to minimize the total communication volume.

In order to solve the partitioning problem, let us examine the computational dependencies. Each processor p should use its triplets $\langle i, j, a_{ij} \rangle$ to compute partial results for $d_R(i)$ and $d_C(j)$, e.g., for the ∞-norm processor p should compute

$$d_R^p(i) = \max_j \left\{ d_1^{(k)}(i) \times |a_{ij}| \times d_2^{(k)}(j) : a_{ij} \in p \right\} .$$

The partial results should be reduced for each $d_1^{(k+1)}(i)$ and $d_2^{(k+1)}(j)$, e.g., in the ∞-norm the owner of $d_1(i)$ should compute

$$d_1^{(k+1)}(i) = d_1^{(k)}(i) \times \frac{1}{\sqrt{\max\{d_R^p(i) : 1 \le p \le P\}}} .$$

Note that the communication operations take place during these reduction operations. That is, the partial results $d_R^p(i)$ from each processor p, where $1 \le p \le P$ and there exist a $a_{ik} \in p$ for some k, should be sent to the processor which is responsible for computing $d_1^{(k+1)}(i)$. After computing $d_1^{(k+1)}(i)$, the owner should send the new values back to the contributing processors to enable the computation of $\widehat{\mathbf{A}}^{(k+1)}$. That is, the owner sends the updated $d_1^{(k+1)}(i)$ to each processor p having a nonzero in row i, e.g., to a processor p where $a_{ik} \in p$ for some k. Therefore, the volume of data a processor receives to compute $d_1^{(k+1)}(i)$ is equal to the volume of data it sends after computing the final value.

If the nonzeros in row $\mathbf{r}_i$ are split among s processors, then a reduction on s partial results will be necessary. If one of those processors owns $d_1(i)$, then $s-1$ partial results will be sent to the owner; if not, then s partial results will be sent to the owner. Hence, for a given partition on $\mathbf{A}$, the minimum volume of communication regarding $\mathbf{r}_i$ is $s-1$. The same assertions hold for $d_2(j)$ with respect to the nonzeros in column $\mathbf{c}_j$. Therefore, if the nonzeros in row $\mathbf{r}_i$ and column $\mathbf{c}_j$ are split among $s_r(i)$ and $s_c(j)$ processors, respectively, then the minimum total communication volume is

$$2 \times \sum (s_r(i) - 1) + 2 \times \sum (s_c(j) - 1) , \tag{4}$$

where half of the communication volume is incurred while reducing the new values and the other half is incurred while sending back the updated values. The minimum total communication volume can be achieved for any partition on $\mathbf{A}$ as long as each $d_1(i)$ and $d_2(j)$ are assigned to a processor which has nonzeros in row $\mathbf{r}_i$ and column $\mathbf{c}_j$, respectively. Furthermore, any $d_1(i)$ to processor p (or $d_2(j)$ to processor p) assignment will attain the same minimum as long as the processor p has at least one nonzero in row $\mathbf{r}_i$ (or column $\mathbf{c}_j$).

It can be seen from Eq. (4) that the communication volume requirements of the proposed algorithm are closely related to those of repeated sparse matrix-vector multiply operations; see for example [4, 12]. In fact, the communication operations in an iteration of Algorithm 1 are the same as those in the computations $\mathbf{y} \leftarrow \mathbf{A}\mathbf{x}$ followed by $\mathbf{x} \leftarrow \mathbf{A}^T\mathbf{y}$, when the partitions on $\mathbf{x}$ and $\mathbf{y}$ are equal to the partitions on $\mathbf{D}_2$ and $\mathbf{D}_1$, respectively. Having observed that, we can use hypergraph models, see for example [4, 20, 22], to partition the matrix $\mathbf{A}$, and then follow the above development to partition $\mathbf{D}_1$ and $\mathbf{D}_2$ to achieve efficient parallelization. Moreover, due to the equivalence between the communication operations of the proposed algorithm and those of sparse matrix-vector multiply operations, we can adopt the vector partitioning techniques discussed in [1, 21] to partition $\mathbf{D}_1$ and $\mathbf{D}_2$.

We wanted to have a parallelization of the scaling algorithm independent of the matrix partitioning. This is because we imagine the use of the algorithm in a parallel linear system solver context where the matrix is already distributed. Therefore, as an alternative to the existing partitioning methods [1, 21], we developed the following parallel algorithm to partition $\mathbf{D}_1$ and $\mathbf{D}_2$ among P processors. In the conceived partitioning although, each $d_1(i)$ will be assigned to the processor which has the closest entry to a fictitious diagonal (on a square matrix of order $\max\{m, n\}$). The same strategy is used for each $d_2(j)$ with respect to the columns. If, for example, the matrix $\mathbf{A}$ is square and has a zero-free diagonal, then the proposed approach will partition $\mathbf{D}_1$ and $\mathbf{D}_2$ in such a way that the processor which holds a_{ii} will own $d_1(i)$ and $d_2(i)$. This is the common approach taken in standard matrix partitioning approaches, see for example [4, 6]. The MPI standard [14] defines an operation, `minloc`, which can be used to accomplish this task. In general, `minloc` can be used to compute a global minimum and the rank of the process whose data contain that minimum value. Below, we discuss how we went about implementing this operation for our partitioning method (essentially the same as `minloc` operation).

In our implementation, we perform a reduction operation on two arrays of sizes $2 \times m$ and $2 \times n$. Each processor p initializes a local copy of both arrays by setting $g_r^p(i) = m + n$ for $1 \le i \le m$ and $g_r^p(i) = p$ for $i > m$, and similarly $g_c^p(j) = m+n$ for $1 \le j \le n$ and $g_c^p(j) = p$. Then, each processor sweeps over its triplets $\langle i, j, a_{ij} \rangle$ and computes its shortest distance to the diagonal entry in row i and its shortest distance to the diagonal entry in column j. That is, processor p computes

$$g_r^p(i) = \min\{|i - j| : a_{ij} \in p\}$$

and

$$g_c^p(j) = \min\{|j - i| : a_{ij} \in p\} \ .$$

As we see, at the end of these operations, the shortest distances are stored in the first halves of the arrays whereas the second halves store the rank of the associated processor. A global all-reduce operation is performed on these two arrays to yield an array g_r of size $2 \times m$ and another one g_c of size $2 \times n$ on all processors. The reduction operation is performed with the minimum operation to set

$$g_r(i) = \min_p\{g_r^p(i) : 1 \le p \le P\} \ \text{ for } \ 1 \le i \le m$$

and

$$g_c(j) = \min_p\{g_c^p(j) : 1 \le p \le P\} \ \text{ for } \ 1 \le j \le n \ .$$

The second halves of the arrays are used to store the ranks of the processor that give the minimum value stored in the corresponding entries in the first halves of the arrays. The second halves are necessary to guarantee a unique partitioning vector on all processors. If there is a tie for an entry in the first half of the arrays, the processor with the smaller rank is declared as the one giving the minimum—a unique partitioning vector is not possible without further communication if, for example, the second halves were not used and the ties were broken at random. Note that, although different implementations are possible, MPI's `minloc` operation also uses an array of size twice the number of data items to store the rank of the processor that owns a copy of the minimum value.

We make a few observations about the proposed partitioning algorithm. Firstly, the proposed diagonal matrix partitioning approach tries to exploit the given partition of the matrix $\mathbf{A}$ and obtains the minimum total volume of communication possible (with respect to the given partition on $\mathbf{A}$). This is achieved by assigning each $d_1(i)$ or $d_2(j)$ to a processor having nonzeros entries in the respective row or column of the matrix. In other words, the proposed algorithm achieves the minimum given in Eq. (4); if the matrix is partitioned with the objective of minimizing the total communication volume, then the total communication volume found there is retained intact by the algorithm. Secondly, we believe that the algorithm is likely to achieve a balance on the number of $\mathbf{D}_1$ and $\mathbf{D}_2$ matrix entries assigned to the processors, hence in a way it will achieve a balance on communication loads of the processors. We investigate the issue of the balance achieved in the communication loads of the processors in the next section. We note that the problem of optimizing the partitioning of $\mathbf{D}_1$ and $\mathbf{D}_2$ for some other communication cost metrics such as the total number of messages with a balancing constraint on the communication volume loads of processors, or the maximum volume of messages sent and received by a single processor is NP-complete; see [20] and [1], respectively. Rather than addressing such communication cost metrics explicitly, we prefer the proposed partitioning algorithm, as it is easy to implement and is fast to run in parallel.

4 Experiments

We have implemented a parallel program for the proposed matrix scaling algorithm in C using LAM/MPI [3]. The experiments were carried out on up to 16 nodes of two PC clusters of Beowulf class [19]. In the first cluster, the nodes are Intel Pentium IV 2.6 GHz processors with 1GB of RAM, and they run Debian/GNU Linux. This cluster has a Gigabit Ethernet switch. The cluster has a measured latency of 37 microseconds and a measured bandwidth of 75MB/s. The second cluster has an Infiniband interconnection network and is based on Dual 250 Opteron AMD processors each having 4GB of RAM. In this cluster, latency and bandwidth are measured as 3.3 microseconds and 772MB/s, respectively. In both of the systems, the program is compiled with `gcc` using the optimization option `-O3`.

We ran the program on a set of matrices from the University of Florida sparse matrix collection [8]. The characteristics of the matrices are shown in Table 1.

Table 1. Matrices used in measuring the parallel performance, their size, number of nonzeros, and the number of iterations to converge in the ∞- and 1-norms with error tolerance of 1.0e-6. The number 1000 indicates cases where the method did not converge in 1000 iterations (those matrices, except `Hamrle3`, do not have total support). Matrices are listed in increasing order of the number of nonzeros.

			number of iterations	
matrix	n	nnz	∞-norm	1-norm
aug3dcqp	35543	128115	26	50
a5esindl	60008	255004	2	107
a2nnsnsl	80016	355034	22	115
a0nsdsil	80016	355034	22	106
blockqp1	60012	640033	2	48
olesnik0	88263	744216	23	1000
c-71	76638	859554	24	1000
boyd1	93279	1211231	25	28
twotone	120750	1224224	24	1000
lhr71c	70304	1528092	27	1000
H2O	67024	2283760	2	16
filter3D	106437	2813616	3	20
Hamrle3	1447360	5514242	23	1000
G3_circuit	1585478	9246304	2	19
thermal2	1228045	9808358	2	18
SiO2	155331	11438834	2	16

We have observed that usually 25–30 iterations of the discussed scaling algorithm is sufficient to improve the condition number of the matrices. We used the performance profiles discussed in [9] to generate the plot shown in Fig. 1. The plot compares estimates of the condition numbers for the scaled matrices resulting from four different scaling algorithms and those of the original matrices. For a given τ, the plot shows the probability for a scaling algorithm that the condition estimate due to this algorithm is within τ times the best (among all 5 condition estimates). Therefore, the higher the probability the more preferable the scaling method. We have plotted the performance profiles up to $\tau = 5$. As seen in the plot, the condition estimate of the original matrix has the worst profile; at any τ, the condition estimate of the original matrix has the least probability to be the best. As also seen from the plot, the discussed scaling algorithm with any of the norms (1-, 2-, or ∞) has higher probability to be better than that of Bunch's for τ a little larger than 1.5. We note that Bunch's algorithm is a direct approach; it computes the scaling factors without any iterations. Note that for these results we run the parallel scaling algorithms for at most 25 iterations. Although the 1- and 2-norm scaling algorithms did not fully converge in 136 of the 245 cases, the resulting condition estimates after 25 iterations were close to the best value, e.g., with a high probability, the results are within a small τ of the best. The Sinkhorn-Knopp algorithm gave almost the same condition estimates as the parallel scaling algorithm with the 1- and 2-norms.

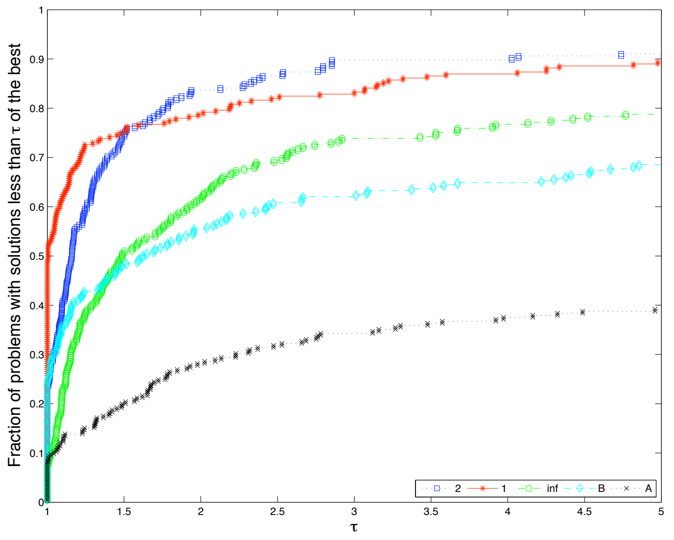

Fig. 1. Performance profiles for the condition number estimates for 245 matrices. `A` marks the condition number estimate of the original matrix; `B` marks that of Bunch's algorithm [2]; inf, 1, and 2 mark that of the parallel scaling algorithm with ∞-, 1-, and 2-norms (with at most 25 iterations). At, for example $\tau = 3$, the curves from top to bottom correspond to the labels given in the legend from left to right.

To measure the average running time of an iteration, we ran the program for 1000 iterations, without testing convergence. We used the fine-grain hypergraph model [6] and the hypergraph partitioning tool PaToH [5] with default options to partition the matrices. In the fine-grain model, the nonzeros of a matrix are partitioned independently, i.e., nonzeros in a row or a column are not necessarily assigned to a common processor. We compute the partitions on $\mathbf{D}_1$ and $\mathbf{D}_2$ with the parallel algorithm proposed towards the end of Section 3.

Table 2 shows the speedups we have obtained for the matrices in our data set. Note that we measure the time spent in the iterations, and hence assume that the matrix is already distributed among the processors. During these experiments, the convergence tests are not performed, and hence the reported time

Table 2. Speedup values of the parallel scaling algorithm with ∞-norm, on $P = 2, 4, 8$, and 16 processors for two different parallel systems. For each matrix, the first and second lines correspond to the experiments run on, respectively, PC cluster with Intel processors and PC cluster with AMD processors. For each matrix, the sequential running time of the scaling algorithm for 1000 iterations is listed in units of seconds under the column Seq.Time.

		P			
matrix	Seq.Time	2	4	8	16
aug3dcqp	8.30	1.7	2.9	4.1	4.5
	3.06	1.9	3.8	4.3	3.6
a5esindl	15.09	1.8	3.0	4.1	4.8
	5.12	1.5	1.9	2.3	3.8
a2nnsnsl	20.71	1.8	3.1	4.0	4.8
	7.24	1.5	1.8	2.1	3.3
a0nsdsil	20.92	1.8	3.1	4.0	4.6
	7.22	1.5	1.8	2.1	3.2
blockqp1	32.55	1.9	3.4	5.5	7.4
	8.97	1.6	2.4	3.3	4.9
olesnik0	46.08	1.9	3.7	6.9	12.3
	14.91	1.9	3.9	7.5	13.6
c-71	51.60	1.8	3.3	5.4	7.6
	17.54	1.6	3.3	5.3	6.7
boyd1	70.34	1.9	3.6	6.3	10.2
	24.57	1.8	3.1	4.9	7.6
twotone	74.76	1.9	3.7	7.0	11.8
	25.40	1.9	3.7	6.9	11.3
lhr71	78.25	2.0	3.8	7.3	13.5
	18.10	2.0	3.4	6.8	14.0
H2O	111.33	1.9	2.8	2.4	6.7
	29.33	1.6	2.5	4.2	7.7
filter3D	146.83	1.9	3.7	7.1	13.3
	52.66	2.1	3.5	6.7	12.7
Hamrle3	337.99	1.9	3.8	7.3	13.9
	146.15	1.9	3.8	7.0	12.6
G3_circuit	455.25	1.8	3.8	7.4	14.0
	173.11	1.9	3.3	6.9	14.5
thermal2	573.24	2.0	3.9	7.6	14.4
	208.20	1.6	3.4	6.5	13.1
SiO2	545.90	1.9	3.7	6.9	11.3
	180.09	1.9	3.6	5.9	9.5

of the iterations does not include the time spent doing the convergence checks. The speedups are the averages of 10 different matrix partitions obtained with the fine-grain model. As is seen in the table, good results are obtained for the bigger (in terms of number of nonzeros) matrices (except for `c-71` and `H2O`). That is, most of the time, we obtain better speedups for matrices with a larger number of nonzeros. This is expected as the computation to communication ratio is small for sparse matrix-vector multiply type operations. Therefore, if the matrix has a small number of nonzeros, the communication overhead becomes significant and degrades the performance. We investigated the communication patterns in an attempt to understand the performance of the proposed parallelization approach. Notice that the load balance and the total communication volume are determined according to the given matrix distribution. In all cases, the load imbalance was less than 0.03; we measure the imbalance as $(w_{max} - w_{avg})/w_{avg}$, where w_{max} is the maximum load and w_{avg} is total load divided by the number processors, so the value zero would indicate perfect balance, a value of 1 that the maximum load was twice the average and a value greater than 1 would indicate severe imbalance. The algorithm proposed for partitioning $\mathbf{D}_1$ and $\mathbf{D}_2$ resulted in acceptable imbalances among the communication loads of the processors. In terms of number messages sent by a single processor, the imbalance among loads of the processors, is on the average, 0.25 with a maximum of 1.45. In terms of volume of messages sent by a single processor, the average imbalance is 0.4, with a maximum of 3.27. We further investigated the communication patterns for 64- and 128-way partitions of the matrices in our data set. Although we have seen some large numbers, the average imbalance is around 3.2 using a metric of the maximum number of messages per processor, and 4.3 with the metric being the maximum volume of messages per processor.

In an attempt to verify empirically that the proposed algorithm for partitioning $\mathbf{D}_1$ and $\mathbf{D}_2$ works well for a number of systems, we performed experiments on the nodes of a CRAY XD1 system at CERFACS. This system has two AMD Opteron 2.4 GHz processors per node, each having 2GBytes of memory. The nodes are connected with a RapidArray interconnect with an MPI latency of 1.7 microseconds and a bandwidth of 4GB/s between nodes. The speedups obtained in this system are similar to the reported results.

We have also investigated a sensible alternative partitioning approach on the three parallel systems mentioned so far. The alternative approach is to assign $d_1(i)$ to the processor with the smallest rank among those having nonzeros in row $\mathbf{r}_i$; assign $d_2(j)$ to the processor with the largest rank among those having nonzeros in column $\mathbf{c}_j$. Note that this alternative will also achieve the same minimum in the total volume of communication metric, see Eq. (4). On the PC cluster with AMD processors and Infiniband interconnect and also on the CRAY XD1, the use of this alternative resulted in speedups similar to those resulting from the proposed partitioning approach. However, the alternative did not perform as well on the PC cluster mentioned before. Note that the alternative can produce high imbalance among the number of messages sent by a single processor. Furthermore, the messages are usually short. Combined with the relatively high message latency overhead, this is the most probable reason behind the PC cluster being intolerant to simple partitioning algorithms. In fact, we have observed that the alternative resulted in imbalances, on the average, of around 1.0 for the communication cost metrics of number of messages and communication volume per processor, both in terms of sends and receives, with the maximum being 7.0 for all of the metrics, which is really a very bad imbalance.

5 Conclusion

In this work, we reviewed an iterative algorithm which scales the l-norm, for $l = 1, 2, \ldots, \infty$, of the rows and columns of a matrix to 1 and briefly mentioned some of its properties. We discussed the parallelization of the algorithm. We argued that the parallelization requires a careful partitioning of two diagonal matrices in addition to a standard sparse matrix partitioning for parallel matrix-vector multiply operations. We proposed a method based on an all-reduce operation to partition the diagonal matrices. We discussed performance results on different parallel systems where good speedups are obtained for matrices having a reasonably large number of nonzeros.

Acknowledgements

We thank Prof. C. Aykanat and members of the parallel and distributed computing research group of Bilkent University, Ankara, Turkey; and Prof. Ü. V. Çatalyürek of the Department of Biomedical Informatics at the Ohio State University, USA for granting us exclusive access to their parallel machines. We also thank Dr. Jean-Yves L'Excellent of Laboratoire de l'Informatique du Parallélisme, ENS Lyon, France for pointing out the MPI's `minloc` operation.

References

1. R. H. Bisseling and W. Meesen. Communication balancing in parallel sparse matrix-vector multiplication. *Electronic Transactions on Numerical Analysis*, 21:47–65, 2005.
2. J. R. Bunch. Equilibration of symmetric matrices in the max-norm. *Journal of the ACM*, 18(4):566–572, 1971.
3. G. Burns, R. Daoud, and J. Vaigl. LAM: an open cluster environment for MPI. In John W. Ross, editor, *Proceedings of Supercomputing Symposium '94*, pages 379–386. University of Toronto, 1994.
4. Ü. V. Çatalyürek and C. Aykanat. Hypergraph-partitioning based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):673–693, 1999.
5. Ü. V. Çatalyürek and C. Aykanat. PaToH: A multilevel hypergraph partitioning tool, version 3.0. Technical Report BU-CE-9915, Computer Engineering Department, Bilkent University, 1999.

6. Ü. V. Çatalyürek and C. Aykanat. A fine-grain hypergraph model for 2d decomposition of sparse matrices. In *Proceedings of 15th International Parallel and Distributed Processing Symposium (IPDPS)*, San Francisco, CA, April 2001.
7. A. R. Curtis and J. K. Reid. On the automatic scaling of matrices for Gaussian elimination. *IMA Journal of Applied Mathematics*, 10(1):118–124, 1972.
8. T. A. Davis. University of Florida sparse matrix collection: `http://www.cise.ufl.edu/research/sparse/matrices`. *NA Digest*, 92/96/97, 1994/1996/1997.
9. E. D. Dolan and J. J. Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91(2):201–213, 2002.
10. I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices.* Oxford University Press, London, 1986.
11. I. S. Duff and J. Koster. On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM Journal on Matrix Analysis and Applications*, 22(4):973–996, 2001.
12. B. Hendrickson and T. G. Kolda. Partitioning rectangular and structurally unsymmetric sparse matrices for parallel processing. *SIAM Journal on Scientific Computing*, 21(6):2048–2072, 2000.
13. HSL: A collection of Fortran codes for large-scale scientific computation. `http://www.cse.scitech.ac.uk/nag/hsl`, 2004.
14. MPI: A Message-Passing Interface Standard, Version 2.1. `http://www.mpi-forum.org/docs/`, 2008.
15. U. G. Rothblum, H. Schneider, and M. H. Schneider. Scaling matrices to prescribed row and column maxima. *SIAM Journal on Matrix Analysis and Applications*, 15(1):1–14, 1994.
16. D. Ruiz. A scaling algorithm to equilibrate both rows and columns norms in matrices. Technical Report RAL-TR-2001-034 and RT/APO/01/4, Rutherford Appleton Laboratory, Oxon, UK and ENSEEIHT-IRIT, Toulouse, France, 2001.
17. M. H. Schneider and S. Zenios. A comparative study of algorithms for matrix balancing. *Operations Research*, 38(3):439–455, 1990.
18. R. Sinkhorn and P. Knopp. Concerning nonnegative matrices and doubly stochastic matrices. *Pacific Journal of Mathematics*, 21(2):343–348, 1967.
19. T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranaweke, and C. V. Packer. BEOWULF: A parallel workstation for scientific computation. In *Proceedings of the 24th International Conference on Parallel Processing*, 1995.
20. B. Uçar and C. Aykanat. Encapsulating multiple communication-cost metrics in partitioning sparse rectangular matrices for parallel matrix-vector multiplies. *SIAM Journal on Scientific Computing*, 25(6):1827–1859, 2004.
21. B. Uçar and C. Aykanat. Revisiting hypergraph models for sparse matrix partitioning. *SIAM Review*, 49(4):595–603, 2007.
22. B. Vastenhouw and R. H. Bisseling. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM Review*, 47(1):67–95, 2005.