A sparse matrix scaling algorithm and its efficient parallelization

Bora Uçar

CNRS and LIP, ENS Lyon, France

HPCSE2013, 27–30 May, 2013, Soláň, Czech Republic

Joint work with:

Patrick R. Amestoy
Ümit V. Çatalyürek
Iain S. Duff

Kamer Kaya
Philip A. Knight
Daniel Ruiz
Outline

1. Theory

2. Distributed memory parallelization
   - Experiments

3. Shared-memory parallelization
   - Experiments
Matrix scaling

**Definition**

Given an \( m \times n \) sparse matrix \( \mathbf{A} \), find diagonal matrices \( \mathbf{D}_1 > 0 \) and \( \mathbf{D}_2 > 0 \) such that all rows and columns of the scaled matrix

\[
\hat{\mathbf{A}} = \mathbf{D}_1 \mathbf{A} \mathbf{D}_2
\]

have equal norm.

**Motivations**

- Equilibration, balancing, good pivoting strategy, numerical/optimal properties.
- Scaling combined with permutations can avoid many numerical difficulties [Duff and Pralet '05] during LU factorization:
  - Provides (weak) diagonal dominance
  - Increases robustness of the factorization algorithms
  - May improve the condition number
The sequential algorithm (Ruiz’01)

1: \( \mathbf{D}_r^{(0)} \leftarrow \mathbf{I}_{m \times m} \quad \mathbf{D}_c^{(0)} \leftarrow \mathbf{I}_{n \times n} \)

2: \textbf{for } k = 1, 2, \ldots \textbf{ until } \text{ convergence } \textbf{ do}

3: \( \mathbf{D}_1 \leftarrow \text{diag} \left( \sqrt{\| \mathbf{r}_i^{(k)} \|_\ell} \right) \quad i = 1, \ldots, m \)

4: \( \mathbf{D}_2 \leftarrow \text{diag} \left( \sqrt{\| \mathbf{c}_j^{(k)} \|_\ell} \right) \quad j = 1, \ldots, n \)

5: \( \mathbf{A}^{(k+1)} \leftarrow \mathbf{D}_1^{(k+1)} \mathbf{A} \mathbf{D}_2^{(k+1)} \)

6: \( \mathbf{D}_r^{(k+1)} \leftarrow \mathbf{D}_r^{(k)} \mathbf{D}_1^{-1} \)

7: \( \mathbf{D}_c^{(k+1)} \leftarrow \mathbf{D}_c^{(k)} \mathbf{D}_2^{-1} \)

Reminder

\( \mathbf{r}_i^{(k)}: \) \( i \)th row at \( i \).

\( \| \mathbf{x} \|_\infty = \max \{|x_i|\} \)

\( \| \mathbf{x} \|_1 = \sum |x_i| \)

Notes

\( \ell \): any vector norm (usually \( \infty \)- and 1-norms)

Convergence is achieved when

\[
\max_{1 \leq i \leq m} \left\{ 1 - \| \mathbf{r}_i^{(k)} \|_\ell \right\} \leq \varepsilon \quad \text{and} \quad \max_{1 \leq j \leq n} \left\{ 1 - \| \mathbf{c}_j^{(k)} \|_\ell \right\} \leq \varepsilon
\]
Some properties (Ruiz’01; Knight, Ruiz, U. ’12)

- Preserves symmetry; permutation independent; amenable to parallelization
- With $\infty$-norm, linear convergence with asymptotic rate of $1/2$
- With 1-norm, convergence under some structural conditions (as in some other well-known algorithms [Sinkhorn and Knopp’67])
  - For symmetric matrices, converges linearly with a rate depending on the spectrum of the scaled matrix
  - For unsymmetric ones, converges linearly with a rate depending on the second largest singular value of the scaled matrix
- Sequential codes available in HSL library as MC77 [Ruiz’01]
- Parallel codes available—also have been plugged into MUMPS [Amestoy, Duff, and L’Excellent’00]
### Number of iterations

SK-1 and SK-2: Sinkhorn–Knopp algorithm in 1- and 2-norms; A-1, A-2, and A-∞: proposed algorithm; error tolerance $\varepsilon = 1.0e^{-4}$.

214 matrices from UFL: real, $1000 \leq n \leq 121000$, $2n \leq \text{nnz} \leq 1790000$, without explicit zeros, fully indecomposable, not a matrix of $\{-1, 0, 1\}$.

<table>
<thead>
<tr>
<th>matrix type</th>
<th>statistics</th>
<th>SK-1</th>
<th>SK-2</th>
<th>A-1</th>
<th>A-2</th>
<th>A-∞</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsymmetric (64)</td>
<td>min</td>
<td>1</td>
<td>47</td>
<td>1</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>med</td>
<td>2135</td>
<td>4905</td>
<td>2436</td>
<td>4897</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>max</td>
<td>116205</td>
<td>177053</td>
<td>307672</td>
<td>519249</td>
<td>19</td>
</tr>
<tr>
<td>symmetric (104)</td>
<td>min</td>
<td>8</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>med</td>
<td>238</td>
<td>700</td>
<td>32</td>
<td>33</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>max</td>
<td>11870</td>
<td>22302</td>
<td>10307</td>
<td>18925</td>
<td>19</td>
</tr>
<tr>
<td>sym pos def (46)</td>
<td>min</td>
<td>73</td>
<td>46</td>
<td>7</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>med</td>
<td>444</td>
<td>1494</td>
<td>14</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td></td>
<td>max</td>
<td>11271</td>
<td>14418</td>
<td>17</td>
<td>18</td>
<td></td>
</tr>
</tbody>
</table>
Helps numerically (experiments with MUMPS)

Unsuccessful (usuc.), if MUMPS 4.10 returns a warning or an error message.

<table>
<thead>
<tr>
<th>strategy</th>
<th>unsymmetric matrices</th>
<th>general symmetric matrices</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>usuc.</td>
<td>actM / estM</td>
</tr>
<tr>
<td>no-scaling</td>
<td>7</td>
<td>1.02</td>
</tr>
<tr>
<td>[0, 3(^{(1)}), 0]</td>
<td>3</td>
<td>1.02</td>
</tr>
<tr>
<td>[1, 3(^{(1)}), 0]</td>
<td>3</td>
<td>1.01</td>
</tr>
<tr>
<td>[0, 10(^{(1)}), 0]</td>
<td>3</td>
<td>1.01</td>
</tr>
<tr>
<td>[1, 10(^{(1)}), 0]</td>
<td>3</td>
<td>1.01</td>
</tr>
<tr>
<td>[1, 100(^{(1)}), 0]</td>
<td>3</td>
<td>1.01</td>
</tr>
<tr>
<td>[0, 3(^{(2)}), 0]</td>
<td>0</td>
<td>1.02</td>
</tr>
<tr>
<td>[1, 3(^{(2)}), 0]</td>
<td>0</td>
<td>1.01</td>
</tr>
<tr>
<td>[0, 10(^{(2)}), 0]</td>
<td>0</td>
<td>1.01</td>
</tr>
<tr>
<td>[1, 10(^{(2)}), 0]</td>
<td>0</td>
<td>1.01</td>
</tr>
<tr>
<td>[1, 100(^{(2)}), 0]</td>
<td>0</td>
<td>1.01</td>
</tr>
<tr>
<td>Bunch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SK10</td>
<td>0</td>
<td>1.01</td>
</tr>
</tbody>
</table>

Moral: [Bunch’71] for symmetric matrices, sequential environment; “SK” for unsymmetric matrices; proposed one for symmetric matrices, parallel environment.
1: $D_r^{(0)} \leftarrow I_{m \times m}$, $D_c^{(0)} \leftarrow I_{n \times n}$

2: for $k = 1, 2, \ldots$ until convergence do

3: $D_1 \leftarrow \text{diag} \left( \sqrt{||r_i^{(k)}||_\ell} \right)$ for $i = 1, \ldots, m$

4: $D_2 \leftarrow \text{diag} \left( \sqrt{||c_j^{(k)}||_\ell} \right)$ for $j = 1, \ldots, n$

5: $A^{(k+1)} \leftarrow D_1^{(k+1)} AD_2^{(k+1)}$

6: $D_r^{(k+1)} \leftarrow D_r^{(k)} D_1^{-1}$

7: $D_c^{(k+1)} \leftarrow D_c^{(k)} D_2^{-1}$
Parallelization: Data distribution

### Data

\( \hat{A}^{(k)} \), \( A \), \( D_R^{(k)} \), \( D_C^{(k)} \), \( D_1 \) and \( D_2 \).

### The scaled matrix \( \hat{A}^{(k)} \)

Do not store \( \hat{A}^{(k)} = D_R^{(k)} AD_C^{(k)} \) explicitly; access \( a_{ij}^{(k)} \) by

\[
d_r^{(k)}(i) \times |a_{ij}| \times d_c^{(k)}(j)
\]

- Distribute \( A \), \( D_R \), and \( D_C \). At every iteration, \( D_1 \) and \( D_2 \) (the row and column norms) are computed afresh.
  - Matrix \( A \) is already distributed (in another context).
    - Each processor holds a set of entries \( a_{ij} \) and their indices \((i, j)\).
  - Partition the diagonal elements of \( D_R \) and \( D_C \) among processors.

### Problem definition

Given a partition on \( A \), find the best partitions for \( D_R \) and \( D_C \).
Parallelization: Computations and computational dependencies

Local computations

Each processor $p$ should use each $(i, j, a_{ij})$ triplet to compute partial results on $d_1(i)$ and $d_2(j)$, e.g., in $\infty$-norm, sets

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

Communication operations

The partial results should be combined/reduced for each $d_R^{(k+1)}(i)$. The owner of $d_R(i)$ should set, in $\infty$-norm,

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

The owner should send $d_R^{(k+1)}(i)$ back to the contributing processors.

- Similar discussion for $d_C(j)$. 

Matrix scaling
Parallelization: $\infty$-norm algorithm for step $k$

Row $r_i$

Processors 2 and 4 contribute to $d_R^{(k+1)}(i)$. Whichever owns $d_R(i)$, receives one unit of data and sends one unit of data after computing the final $d_R^{(k+1)}(i)$.

Column $c_j$

Processors 1, 2, and 3 contribute to $d_C^{(k+1)}(j)$. Whichever owns $d_C(j)$, receives two units of data and sends two units of data after computing the final $d_C^{(k+1)}(j)$. 

Matrix scaling
Parallelization: Communication requirements

Common communication cost metric: the total volume.

Communication for $D_R$

- The volume of data a processor receives while reducing a $d_{R(k+1)}(i)$ is equal to the volume of data it sends after computing $d_{R(k+1)}(i)$.
- Nonzeros in row $r_i$ are split among $s_r(i)$ processors
  - All contribute to $d_{R(k+1)}(i)$.
  - Reduction on $s_r(i)$ partial results.
  - If one of those $s_r(i)$ processors owns $d_R(i)$, $s_r(i) - 1$ partial results will be send to the owner.
  - If owned by somebody else, then $s_r(i)$ partial results will be send to the owner.

Communication for $D_C$

Similar observations.
Parallelization: Partitioning $D_R$ and $D_C$

**Communication requirements**

Nonzeros in row $r_i$ are split among $s_r(i)$ processors: total volume of communication is equal to

$$2 \times \sum (s_r(i) - 1) = 2 \times k_{conn}$$

(half for receiving contributions, half for sending back the results).

- The total volume of communication is the same for any $d_R(i)$ to processor assignment as long as that processor has at least one nonzero from row $r_i$.

Similar observation for the column $c_j$.

Twice the requirements of parallel sparse matrix-vector multiply operation.
# Summary of computational and communication requirements

## Computations (per iteration)

<table>
<thead>
<tr>
<th>Op.</th>
<th>SpMxV</th>
<th>1-norm</th>
<th>∞-norm</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>(\text{nnz}(A))</td>
<td>(2 \times \text{nnz}(A))</td>
<td>(0)</td>
</tr>
<tr>
<td>mult</td>
<td>(\text{nnz}(A))</td>
<td>(2 \times \text{nnz}(A) + m + n)</td>
<td>(2 \times \text{nnz}(A) + m + n)</td>
</tr>
<tr>
<td>comparison</td>
<td>0</td>
<td>0</td>
<td>(2 \times \text{nnz}(A))</td>
</tr>
</tbody>
</table>

## Communication (per iteration)

The communication operations both in the 1-norm and ∞-norm algorithms are the same as those in the computations

\[
y \leftarrow Ax
\]
\[
x' \leftarrow A^T y
\]

when the partitions on \(x\) and \(y\) are equal to the partitions on \(D_R\) and \(D_C\).
Parallelization: Our partitioning approach

What we did?

- To avoid extra work, use simple strategies.
- Ensure that each scaling entry (those of $D_R$ or $D_C$) is assigned to a processor that contributes to that entry
- the minimum total volume of communication under a given partition of matrix elements.

$d_R(i)$: assign to the processor $p$ that has an entry $a_{ij}$ with $j$ giving $\min\{|i - j|\}$; in case of ties to the processor with the smallest rank.

$d_C(j)$: assign to the processor $p$ that has an entry $a_{ij}$ with $i$ giving $\min\{|i - j|\}$; in case of ties to the processor with the smallest rank.
## Experiments

### Data set
- Matrices from University of Florida sparse matrix collection
- real, $1000 \leq n < \text{nnz}(A) \leq 2.0e+6$
- A total of 213 matrices out of 1877 (as of Sep.’07).

### Number of iterations with convergence criteria of $\varepsilon = 1.0e-6$
- $\infty$-norm: Always converges very fast. Average 11.
- $1$- and $2$-norms: Did not converge for 10 and 17 matrices in 5000 iterations, respectively.
  - Average number of iterations in converged cases are 206 and 257,
  - Matrices from two groups (GHS_indef and Schenk.ibmna) cause problems (larger number of iterations as well). 60 matrices from these groups.
  - Excluding those matrices, the averages are 26 and 29.
Parallelization results: Speedup values

<table>
<thead>
<tr>
<th>matrix</th>
<th>Seq. Time (s.)</th>
<th>Number of processors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>aug3dcqp</td>
<td>8.30</td>
<td>1.7</td>
</tr>
<tr>
<td></td>
<td>3.06</td>
<td>1.9</td>
</tr>
<tr>
<td>a2nnsnsnl</td>
<td>20.71</td>
<td>1.8</td>
</tr>
<tr>
<td></td>
<td>7.24</td>
<td>1.5</td>
</tr>
<tr>
<td>a0nsdsil</td>
<td>20.92</td>
<td>1.8</td>
</tr>
<tr>
<td></td>
<td>7.22</td>
<td>1.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>matrix</th>
<th>Seq. Time (s.)</th>
<th>Number of processors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>lhr71</td>
<td>78.25</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>18.10</td>
<td>2.0</td>
</tr>
<tr>
<td>G3_circuit</td>
<td>455.25</td>
<td>1.8</td>
</tr>
<tr>
<td></td>
<td>173.11</td>
<td>1.9</td>
</tr>
<tr>
<td>thermal2</td>
<td>573.24</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>208.20</td>
<td>1.6</td>
</tr>
</tbody>
</table>

- Averages of 10 different partitions (with PaToH [Çatalyürek and Aykanat, Tech.Rep (1999)]).
- PC cluster with a Gigabit Ethernet switch (Intel Pentium IV 2.6 GHz), PC cluster with an Infiniband interconnect (dual AMD 150 Opteron processors).
- 1000 iterations’ running time in seconds

- Best three and worst three speedup values are shown—speedup tends to be higher for matrices with larger number of nonzeros.
- The partitions are such that they result in reduced total communication volume, \( \kappa_{\text{conn}} \).
Parallelization results: Speedup values

SpMxV in a more recent system:

In-house

PETSc

Trilinos

64-nodes; each node has a 2.27GHz dual quad-core Intel Xeon (Bloomfield) CPU; 20Gbps DDR InfiniBand. All MPI (mvapich2).
### Outline

1. **Theory**
2. **Distributed memory parallelization**
3. **Shared-memory parallelization**
   - Experiments

---

<table>
<thead>
<tr>
<th>Step</th>
<th>Equation/Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>( D_r^{(0)} \leftarrow I_{m \times m} \quad D_c^{(0)} \leftarrow I_{n \times n} )</td>
</tr>
<tr>
<td>2.</td>
<td>for ( k = 1, 2, \ldots ) until convergence do</td>
</tr>
<tr>
<td>3.</td>
<td>( D_1 \leftarrow \text{diag} \left( \sqrt{| r_i^{(k)} |_\ell} \right) \quad i = 1, \ldots, m )</td>
</tr>
<tr>
<td>4.</td>
<td>( D_2 \leftarrow \text{diag} \left( \sqrt{| c_j^{(k)} |_\ell} \right) \quad j = 1, \ldots, n )</td>
</tr>
<tr>
<td>5.</td>
<td>( A^{(k+1)} \leftarrow D_1^{(k+1)} AD_2^{(k+1)} )</td>
</tr>
<tr>
<td>6.</td>
<td>( D_r^{(k+1)} \leftarrow D_r^{(k)} D_1^{-1} )</td>
</tr>
<tr>
<td>7.</td>
<td>( D_c^{(k+1)} \leftarrow D_c^{(k)} D_2^{-1} )</td>
</tr>
</tbody>
</table>
Data structures and the approach (1)

The algorithm will be parallelized using the standard OpenMP techniques (locks, atomic instructions, and/or private memory).

Point of view of a programmer who adopts loop-level parallelism and single-program multiple-data paradigm, without too much adaptations.

Goal: Reduce the associated overhead (size of the private memory, number of locks, number of atomic operations, extra parallel work).

\[
\begin{align*}
1: & \quad D_r^{(0)} \leftarrow I_{m \times m} \quad D_c^{(0)} \leftarrow I_{n \times n} \\
2: & \quad \text{for } k = 1, 2, \ldots \text{ until convergence do} \\
3: & \quad D_1 \leftarrow \text{diag} \left( \sqrt{\|r_i^{(k)}\|_\ell} \right) \quad i = 1, \ldots, m \\
4: & \quad D_2 \leftarrow \text{diag} \left( \sqrt{\|c_j^{(k)}\|_\ell} \right) \quad j = 1, \ldots, n \\
5: & \quad A^{(k+1)} \leftarrow D_1^{(k+1)} A D_2^{(k+1)} \\
6: & \quad D_r^{(k+1)} \leftarrow D_r^{(k)} D_1^{-1} \\
7: & \quad D_c^{(k+1)} \leftarrow D_c^{(k)} D_2^{-1}
\end{align*}
\]
The programmer knows CSR and COO:

\[
\begin{bmatrix}
1.1 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 2.2 & 0.0 & 2.4 & 0.0 \\
3.1 & 0.0 & 3.3 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 4.4 & 0.0 \\
0.0 & 5.2 & 0.0 & 5.4 & 5.5 \\
\end{bmatrix}
\]

and also knows how to perform operations on matrices stored that way.

Compressed row storage (CRS)

Two integer arrays (ia, jcn) and a double array A:

\[
\begin{align*}
ia &= [1, 2, 4, 6, 7, 10] \\
jcn &= [1, 2, 4, 1, 3, 4, 2, 4, 5] \\
A &= [1.1, 2.2, 2.4, 3.1, 3.3, 4.4, 5.2, 5.4, 5.5]
\end{align*}
\]

Coordinate format (COO)

Two integer arrays (irn, jcn) and a double array A:

\[
\begin{align*}
irn &= [1, 2, 2, 3, 3, 4, 5, 5, 5] \\
jcn &= [1, 2, 4, 1, 3, 4, 2, 4, 5] \\
A &= [1.1, 2.2, 2.4, 3.1, 3.3, 4.4, 5.2, 5.4, 5.5]
\end{align*}
\]
We do not store the scaled matrix; access its elements and compute (say 1-norm):

We parallelize each iteration with $\tau$ threads:
- CRS-based storage: partition the rows among the processors.
- COO-based storage: partition the nonzeros among the processors.
Parallelization with CRS (assume 1-norm scaling)

- Rows are partitioned among threads
  - No conflict for row-sum writes
- Use private memory for column sums (size $n$)
- Total computational overhead is $2\tau n$.
- The rowwise partitioning is determined dynamically at runtime by OpenMP scheduling policy.

**Algorithm 2: Simple parallel scaling with CRS**

<table>
<thead>
<tr>
<th>Input: $A$: $n \times n$ input matrix in CRS format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: $d_r$, $d_c$: row and column scaling vectors</td>
</tr>
</tbody>
</table>

```
for $i = 1$ to $n$ in parallel do
    $d_r[i] \leftarrow 1$
    $d_c[i] \leftarrow 1$

while not converged do
    for $i = 1$ to $n$ in parallel do
        $d_1[i] \leftarrow 0$
        $d_2[i] \leftarrow 0$

    init
    for $t = 1$ to $\tau$ in parallel do
        for $i = 1$ to $n$ do
            $d_1[i] \leftarrow 0$

    put
    for $t = 1$ to $\tau$ do
        for $i = 1$ to $n$ in parallel do
            $d_2[i] \leftarrow d_2[i] + d_2[i]$

    error \leftarrow \max\left(\max_{i}(1 - d_1[i]), \max_{i}(1 - d_2[i])\right)$
    if error < $\varepsilon$ then
        converged \leftarrow true
    else
        for $i = 1$ to $n$ in parallel do
            $d_r[i] \leftarrow d_r[i]/\sqrt{d_1[i]}$
            $d_c[i] \leftarrow d_c[i]/\sqrt{d_2[i]}$
```

```
Parallelization with CRS: Improvement (1)

We need **private memory** only for columns whose nonzeros are assigned to different threads.

- Rows are partitioned statically among threads (we know the assignment)
- **No conflict** for row-sum writes
- Use **private memory** for columns that span multiple threads (size $\kappa_{\text{cut}}$)

**External nonzero $a_{ij}$**: there are two or more threads on column $j$. 

---

**Algorithm 4: Part. based scaling with CRS-Cut**

**Input**: $A$: $n \times n$ input matrix in CRS format and a partition $\Pi = \{R_1, R_2, \ldots, R_\tau\}$ of rows

**Output**: $d_r$, $d_c$: row and column scaling vectors

```
while not converged do

init
for $t = 1$ to $\tau$ in parallel do
for $i = 1$ to cut do
  $d_2^t[i] \leftarrow 0$

put
for $t = 1$ to $\tau$ in parallel do
  $\triangleright t$ is the current thread id
  for each external row $i$ in $R_t$ do
    $\text{sum}^t \leftarrow 0$
    for each external nonzero $a_{ij}$ in row $i$ do
      $\text{val} \leftarrow d_r[i] \times a_{ij} \times d_c[j]$
      add $\text{val}$ to $\text{sum}^t$ and $d_2^t[j]$
    for each internal nonzero $a_{ij}$ in row $i$ do
      $\text{val} \leftarrow d_r[i] \times a_{ij} \times d_c[j]$
      add $\text{val}$ to $\text{sum}^t$ and $d_2^t[j]$
    $d_1[i] \leftarrow \text{sum}^t$
  for each internal row $i$ in $R_t$ do
    $\text{sum}^t \leftarrow 0$
    for each nonzero $a_{ij}$ in row $i$ do
      $\text{val} \leftarrow d_r[i] \times a_{ij} \times d_c[j]$
      add $\text{val}$ to $\text{sum}^t$ and $d_2^t[j]$
    $d_1[i] \leftarrow \text{sum}^t$

get
for $t = 1$ to $\tau$ do
for $i = 1$ to cut in parallel do
  $d_2[i] \leftarrow d_2[i] + d_2^t[i]$
```
We need private memory only for columns touching more than one parts (call them $\mathcal{N}_C$).

- for a partition $\Pi$, the extra memory per thread is
  \[ \kappa_{cut}(\Pi) = \sum_{n \in \mathcal{N}_C} 1 \]

- The computational overhead is
  \[ 2\tau\kappa_{cut}(\Pi) \]

There are three columns in $\mathcal{N}_C$ so $\kappa_{cut} = 3$
Parallelization with CRS: Improvement (2)

We need private memory only for columns whose nonzeros are assigned to different threads. But a thread is not concerned with all:

**Algorithm 5**: Part. based scaling with CRS-SOED

| Input: A: n × n input matrix in CRS format and a partition Π = \{R_1, R_2, ..., R_τ\} of rows |
| Output: d_r, 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 \( R_t \) do |
|   |   |   |   | \( d_t^2[i] \leftarrow 0 \) |
|   | put |
|   |   | get |
|   |   | for t = 1 to τ do |
|   |   |   | for each external column i of \( R_t \) in parallel do |
|   |   |   |   | \( d_2[i] \leftarrow d_2[i] + d_t^2[i] \) |
|   |   |   |   | "As same as CRS-Cut" |
| |   |   | θ threads in columns: \( \lambda = [2 2 1 2 1] \) |
| | \( K_{soed} = 2 + 2 + 2 = 6 \) |

- Rows are partitioned among threads
- No conflict for row-sum writes
- Use private memory for columns that span multiple processors.
- A thread knows the entries it is concerned with (+extra space is \( K_{soed} \)).
- Total computational overhead is \( 2K_{soed} \).
Parallelization with CRS: Using atomic operations

Objective: Reduce the number of atomic operations or locks.

- Rows are partitioned among threads
- No conflict for row-sum writes
- Use private memory for columns that span multiple threads.
- Writes to $d_2$s, column-sum array, use atomic operations (or locks).

The total number of atomic operations/locks is $\kappa_{soed}$.

We can reduce the total number of atomic operations/locks to $\kappa_{soed} - \kappa_{cut} = \kappa_{conn}$ with an additional synchronization.

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

```
for t = 1 to \tau in parallel do
  for each external column $i$ conn. to $C_t$ do
    (atomic) $d_2[i] \leftarrow d_2[i] + d^{t}_2[i]$
```
Parallelization with COO (assume 1-norm scaling)

- Nonzeros are partitioned among threads
  - **Conflicts** for row and column-sum writes
- Use private memory for columns and rows (each of size $n$, so $2n$ per thread)
- Total computational overhead is $4\tau n$.
- The nonzero partitioning is determined dynamically at runtime by OpenMP scheduling policy.

Improvements similar to CRS and an implementation using locks and/or atomic operations are possible.
Experiments: Setup

- Dual quad-core Intel Xeon (Bloomfield)
  - 48GB memory
  - 32KB L1, 256KB L2 caches per core
  - 8MB L3 cache per socket

- Dual quad-core AMD Opteron (Shanghai)
  - 32GB memory
  - 64KB L1, 512KB L2 caches per core
  - 6MB L3 cache per socket

- Algorithms are implemented in C and OpenMP

- icc 12.0 and 11.1 with -O3 optimization flag
### Experiments: Matrices

Properties of the matrices used in the experiments.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>( n )</th>
<th>( nnz )</th>
<th>Avg. deg</th>
</tr>
</thead>
<tbody>
<tr>
<td>trans5</td>
<td>116,835</td>
<td>749,800</td>
<td>6.42</td>
</tr>
<tr>
<td>NotreDame</td>
<td>325,729</td>
<td>929,849</td>
<td>2.85</td>
</tr>
<tr>
<td>rjat21</td>
<td>411,676</td>
<td>1,876,011</td>
<td>4.56</td>
</tr>
<tr>
<td>Hamrle3</td>
<td>1,447,360</td>
<td>5,514,242</td>
<td>3.81</td>
</tr>
<tr>
<td>Chebyshev4</td>
<td>68,121</td>
<td>5,377,761</td>
<td>78.94</td>
</tr>
<tr>
<td>pre2</td>
<td>659,033</td>
<td>5,834,044</td>
<td>8.85</td>
</tr>
<tr>
<td>rjat30</td>
<td>643,994</td>
<td>6,175,244</td>
<td>9.59</td>
</tr>
<tr>
<td>Stanford_Berk.</td>
<td>683,446</td>
<td>7,583,376</td>
<td>11.10</td>
</tr>
<tr>
<td>torso1</td>
<td>116,158</td>
<td>8,516,500</td>
<td>73.32</td>
</tr>
<tr>
<td>atmosmodd</td>
<td>1,270,432</td>
<td>8,814,880</td>
<td>6.94</td>
</tr>
<tr>
<td>atmosmodl</td>
<td>1,489,752</td>
<td>10,319,760</td>
<td>6.93</td>
</tr>
<tr>
<td>cage14</td>
<td>1,505,785</td>
<td>27,130,349</td>
<td>18.02</td>
</tr>
</tbody>
</table>
The average execution time without cut minimization but with perfect near perfect load balance divided by the execution time with cut minimization (using PaToH).

The cut-size minimized partitions lead to better performance.
Experiments: Speedups on Intel

The speedups are computed by using the execution time of the CRS- and COO-based sequential algorithms, respectively.
The speedups are computed by using the execution time of the CRS- and COO-based sequential algorithms, respectively.
Scatter plot of the matrices for which an increase on the execution time of COO-Simple is observed when the number of threads $\tau$ is increased from 4 to 8.
Experiments: Average relative performance on Intel

The relative performance: average execution time of an algorithm over the best average time.
The relative performance: average execution time of an algorithm over the best average time.
Concluding remarks

- Discussed a matrix scaling algorithm which helps in solving linear systems with direct methods.

- A distributed memory, message passing implementation:
  - Communication overhead was expressed to be related to
    \[ \kappa_{\text{conn}} = \sum (\lambda_j - 1), \]
    where \( \lambda_j \) is the number of processors in which the nonzeros in column \( j \) reside.

- A shared memory implementation with OpenMP:
  - Memory overhead is \( \kappa_{\text{cut}} = |\{j : \lambda_j > 1\}| \)
  - Computational overhead is \( \kappa_{\text{soed}} = \kappa_{\text{conn}} + \kappa_{\text{cut}} \)
  - Number of atomic operations is \( \kappa_{\text{soed}} \) or \( \kappa_{\text{conn}} \)

- Not discussed (but can!): the \( \kappa \)'s, the overhead functions, are well-known objective functions of the hypergraph partitioning problem. Great tools are at our disposal.
Thank you for your attention.


http://perso.ens-lyon.fr/bora.ucar
Hypergraphs: Definitions

A hypergraph is a two-tuple $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ where $\mathcal{V}$ is a set of vertices and $\mathcal{N}$ is a set of hyperedges.

A hyperedge $h \in \mathcal{N}$ is a subset of vertices. We call them nets for short.

A weight $w(v)$ is associated with each vertex $v$.

An undirected graph can be seen as a hypergraph where each net contains exactly two vertices.
Hypergraphs: Example

\[ \mathcal{H} = (\mathcal{V}, \mathcal{N}) \] with \( \mathcal{V} = \{1, 2, 3, 4, 5\} \) \( \mathcal{N} = \{n_1, n_2, n_3\} \) where

\[ n_1 = \{1, 3, 4\} \quad n_2 = \{1, 2, 3, 4\} \quad n_3 = \{2, 5\} \]
Hypergraphs: Partitioning

Partition

\( \Pi = \{ V_1, V_2, \ldots, V_K \} \) is a \( K \)-way vertex partition if

- \( V_k \neq \emptyset \),
- parts are mutually exclusive: \( V_k \cap V_\ell = \emptyset \),
- parts are collectively exhaustive: \( V = \bigcup V_k \).
- In \( \Pi \), a net connects a part if it has at least one vertex in that part, i.e., \( h \) connects \( V_k \) if \( h \cap V_k \neq \emptyset \).
- The connectivity \( \lambda(h) \) of a net is equal to the number of parts connected by \( h \).

Constraint: balanced part weights
\[
\sum_{v \in V_k} w(v) \leq (1 + \varepsilon) \frac{\sum_{v \in V} w(v)}{K}.
\]

Objective: Minimize a function of \( \lambda(\cdot) \)s over the cut nets.

Hypergraph partitioning problem is NP-complete.
Hypergraphs partitioning: Example

\( \mathcal{H} = (V, N) \) with 10 vertices and 4 nets, partitioned into four parts.

\( V_1 = \{4, 5\} \quad V_2 = \{7, 10\} \quad V_3 = \{3, 8, 9\} \quad V_4 = \{1, 2, 6\} \)

**Objective functions:**

- \( \kappa_{cut}(\Pi) = \sum_{n \in N_C} 1 \)
  \[= 1 + 1 + 1 + 1 = 4\]

- \( \kappa_{conn}(\Pi) = \sum_{n \in N_C} \lambda_n - 1 \)
  \[= 1 + 2 + 2 + 1 = 6\]

- \( \kappa_{soed}(\Pi) = \sum_{n \in N_C} \lambda_n \)
  \[= 2 + 3 + 3 + 2\]
Hypergraphs partitioning: Example

Column net model of a matrix: $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ where $\mathcal{V}$ corresponds to the rows, and $\mathcal{N}$ corresponds to the columns.
Hypergraphs partitioning: Example

Column net model of a matrix: \( \mathcal{H} = (\mathcal{V}, \mathcal{N}) \) where \( \mathcal{V} \) corresponds to the rows, and \( \mathcal{N} \) corresponds to the columns.

4-way partitioning

Two objective functions in shared memory:

- \( \kappa_{cut}(\Pi) = \text{memory} = \sum_{n \in \mathcal{N}_C} 1 \)
  \[
  = 1 + 1 + 1 + 1 = 4
  \]

- \( \kappa_{soed}(\Pi) = \text{atomic ops} = \sum_{n \in \mathcal{N}_C} \lambda_n \)
  \[
  = 2 + 3 + 3 + 2
  \]