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

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: **for** $k = 1, 2, \ldots$ **until** convergence **do**
3: $\quad \mathbf{D}_1 \leftarrow \operatorname{diag}\left(\sqrt{\|\mathbf{r}_i^{(k)}\|_\ell}\right) i = 1, \ldots, m$
4: $\quad \mathbf{D}_2 \leftarrow \operatorname{diag}\left(\sqrt{\|\mathbf{c}_j^{(k)}\|_\ell}\right) j = 1, \ldots, n$
5: $\quad \mathbf{A}^{(k+1)} \leftarrow \mathbf{D}_1{}^{(k+1)} \mathbf{A} \mathbf{D}_2{}^{(k+1)}$
6: $\quad \mathbf{D}_r{}^{(k+1)} \leftarrow \mathbf{D}_r{}^{(k)}\, \mathbf{D}_1{}^{-1}$
7: $\quad \mathbf{D}_c{}^{(k+1)} \leftarrow \mathbf{D}_c{}^{(k)}\, \mathbf{D}_2{}^{-1}$

Reminder

$\mathbf{r}_i^{(k)}$: ith row at it. k

$\|\mathbf{x}\|_\infty = \max\{|x_i|\}$

$\|\mathbf{x}\|_1 = \sum |x_i|$

Notes

ℓ: any vector norm (usually ∞- and 1-norms)
Convergence is achieved when

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

Features

Some properties (Ruiz'01; Knight, Ruiz, U. '12)

- Preserves symmetry; permutation independent; amenable to parallelization
- With ∞-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 $\varepsilon = 1.0e\text{-}4$.

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

matrix type	statistics	SK-1	SK-2	A-1	A-2	A-∞
unsymmetric (64)	min	1	47	1	6	2
	med	2135	4905	2436	4897	8
	max	116205	177053	307672	519249	19
symmetric (104)	min	8	1	3	1	2
	med	238	700	32	33	13
	max	11870	22302	10307	18925	19
sym pos def (46)	min	73	46	7	3	
	med	444	1494	14	12	
	max	11271	14418	17	18	

Helps numerically (experiments with MUMPS)

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

strategy	unsymmetric matrices				general symmetric matrices			
	usuc.	$\frac{actM}{estM}$	$\frac{cnd(DAE)}{cnd(A)}$	off-piv.	usuc.	$\frac{actM}{estM}$	$\frac{cnd(DAE)}{cnd(A)}$	off-piv.
no-scaling	7	1.02	——	308	23	1.09	——	3454
$[0, 3^{(1)}, 0]$	3	1.02	1.23e-03	67	4	1.04	1.89e-03	3458
$[1, 3^{(1)}, 0]$	3	1.01	1.17e-03	44	3	1.04	1.83e-03	3454
$[0, 10^{(1)}, 0]$	3	1.01	1.54e-03	157	1	1.03	3.94e-03	3462
$[1, 10^{(1)}, 0]$	3	1.01	1.54e-03	160	1	1.03	3.86e-03	3462
$[1, 100^{(1)}, 0]$	3	1.01	1.21e-05	148	0	1.05	4.97e-03	3580
$[0, 3^{(2)}, 0]$	0	1.02	3.78e-02	8	3	1.01	2.26e-02	5504
$[1, 3^{(2)}, 0]$	0	1.01	3.54e-02	9	2	1.02	1.74e-02	5504
$[0, 10^{(2)}, 0]$	0	1.01	3.21e-02	8	1	1.02	1.32e-02	5504
$[1, 10^{(2)}, 0]$	0	1.01	3.39e-02	8	1	1.02	1.36e-02	5504
$[1, 100^{(2)}, 0]$	0	1.01	3.50e-02	8	2	1.01	1.71e-02	5504
Bunch					1	1.03	5.83e-02	5504
SK10	0	1.01	3.52e-02	9				

Moral: [Bunch'71] for symmetric matrices, sequential environment; "SK" for unsymmetric matrices; proposed one for symmetric matrices, parallel environment.

Outline

1: $\mathbf{D}_r^{(0)} \leftarrow \mathbf{I}_{m \times m} \qquad \mathbf{D}_c^{(0)} \leftarrow \mathbf{I}_{n \times n}$
2: **for** $k = 1, 2, \ldots$ **until** convergence **do**
3: $\quad \mathbf{D}_1 \leftarrow \operatorname{diag}\left(\sqrt{\|\mathbf{r}_i^{(k)}\|_\ell}\right) \; i = 1, \ldots, m$
4: $\quad \mathbf{D}_2 \leftarrow \operatorname{diag}\left(\sqrt{\|\mathbf{c}_j^{(k)}\|_\ell}\right) \; j = 1, \ldots, n$
5: $\quad \mathbf{A}^{(k+1)} \leftarrow \mathbf{D}_1^{(k+1)} \mathbf{A} \mathbf{D}_2^{(k+1)}$
6: $\quad \mathbf{D}_r^{(k+1)} \leftarrow \mathbf{D}_r^{(k)} \mathbf{D}_1^{-1}$
7: $\quad \mathbf{D}_c^{(k+1)} \leftarrow \mathbf{D}_c^{(k)} \mathbf{D}_2^{-1}$

Parallelization: Data distribution

Data

$\hat{\mathbf{A}}^{(k)}$, $\mathbf{A}$, $\mathbf{D}_R{}^{(k)}$ $\mathbf{D}_C{}^{(k)}$, $\mathbf{D_1}$ and $\mathbf{D_2}$.

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

Do not store $\hat{\mathbf{A}}^{(k)} = \mathbf{D}_R{}^{(k)}\mathbf{A}\mathbf{D}_C{}^{(k)}$ explicitly; access $a_{ij}{}^{(k)}$ by

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

- Distribute $\mathbf{A}, \mathbf{D}_R$, and $\mathbf{D}_C$. At every iteration, $\mathbf{D_1}$ and $\mathbf{D_2}$ (the row and column norms) are computed afresh.
  - Matrix $\mathbf{A}$ 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 $\mathbf{D}_R$ and $\mathbf{D}_C$ among processors.

Problem definition

Given a partition on $\mathbf{A}$, find the best partitions for $\mathbf{D}_R$ and $\mathbf{D}_C$.

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 ∞-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

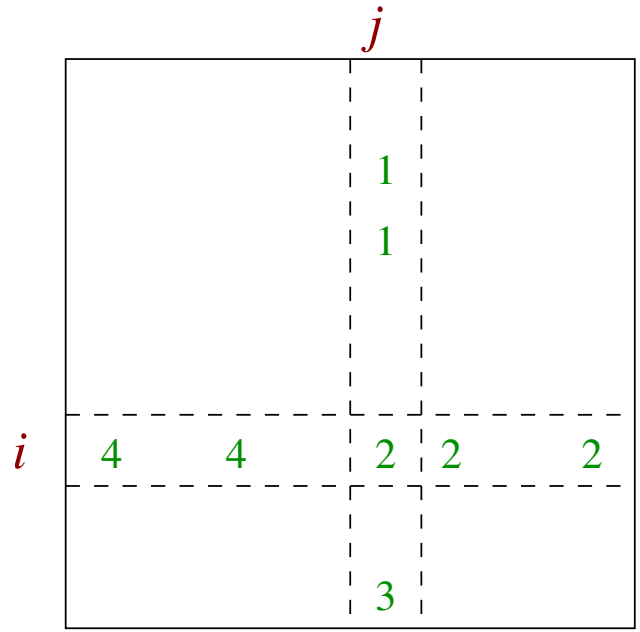
The partial results should be combined/reduced for each $d_R{}^{(k+1)}(i)$.
The owner of $d_R(i)$ should set, in ∞-norm,

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

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

- Similar discussion for $d_C(j)$.

Parallelization: ∞-norm algorithm for step k



Row $\mathbf{r}_i$

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 $\mathbf{c}_j$

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 $d_C{}^{(k+1)}(j)$.

Parallelization: Communication requirements

Common communication cost metric: the total volume.

Communication for $\mathbf{D}_R$

- 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 $\mathbf{r}_i$ 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 $\mathbf{D}_C$

Similar observations.

Parallelization: Partitioning D_R and D_C

Communication requirements

Nonzeros in row $\mathbf{r}_i$ are split among $s_r(i)$ processors: total volume of communication is equal to

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

(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 $\mathbf{r}_i$.

Similar observation for the column $\mathbf{c}_j$.

Twice the requirements of parallel sparse matrix-vector multiply operation.

Summary of computational and communication requirements

Computations (per iteration)

Op.	SpMxV	1-norm	∞-norm
add	$\text{nnz}(\mathbf{A})$	$2 \times \text{nnz}(\mathbf{A})$	0
mult	$\text{nnz}(\mathbf{A})$	$2 \times \text{nnz}(\mathbf{A}) + m + n$	$2 \times \text{nnz}(\mathbf{A}) + m + n$
comparison	0	0	$2 \times \text{nnz}(\mathbf{A})$

Communication (per iteration)

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

$$\mathbf{y} \leftarrow \mathbf{A}\mathbf{x}$$
$$\mathbf{x} \leftarrow \mathbf{A}^T\mathbf{y}$$

when the partitions on $\mathbf{x}$ and $\mathbf{y}$ are equal to the partitions on $\mathbf{D}_R$ and $\mathbf{D}_C$.

Parallelization: Our partitioning approach

What we did?

- To avoid extra work, use simple strategies.
- Ensure that each scaling entry (those of $\mathbf{D}_R$ or $\mathbf{D}_C$) 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, $1000 \leq n < \mathrm{nnz}(\mathbf{A}) \leq 2.0e+6$
  A total of 213 matrices out of 1877 (as of Sep.'07).

Number of iterations with convergence criteria of $\varepsilon = 1.0e-6$

- ∞-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

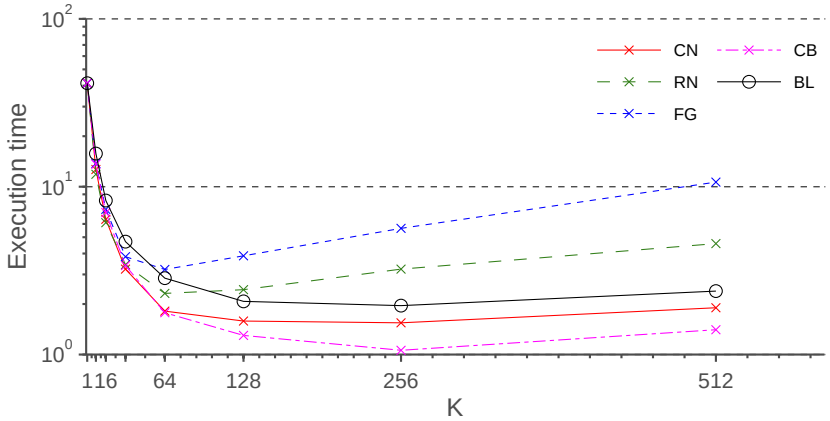
matrix	Seq. Time (s.)	Number of processors 2	4	8	16
aug3dcqp	8.30	1.7	2.9	4.1	4.5
	3.06	1.9	3.8	4.3	3.6
a2nnsnsl	20.71	1.8	3.1	4.0	4.8
	7.24	1.5	1.8	2.1	3.3
a0nsdsil	20.92	1.8	3.1	4.0	4.6
	7.22	1.5	1.8	2.1	3.2
lhr71	78.25	2.0	3.8	7.3	13.5
	18.10	2.0	3.4	6.8	14.0
G3_circuit	455.25	1.8	3.8	7.4	14.0
	173.11	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

- 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

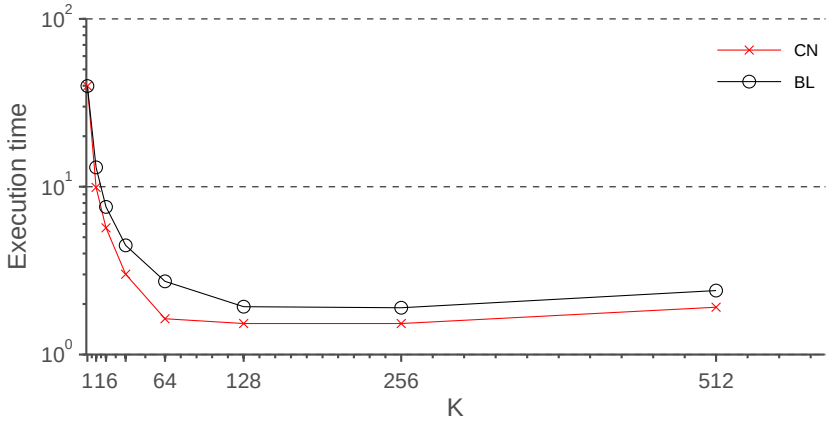
- Best three and worst three speedup values are shown—speedup tends to be higher for matrices with larger number of nonzeros.
- The partitions are such that they result in reduced total communication volume, κ_{conn}.

Parallelization results: Speedup values

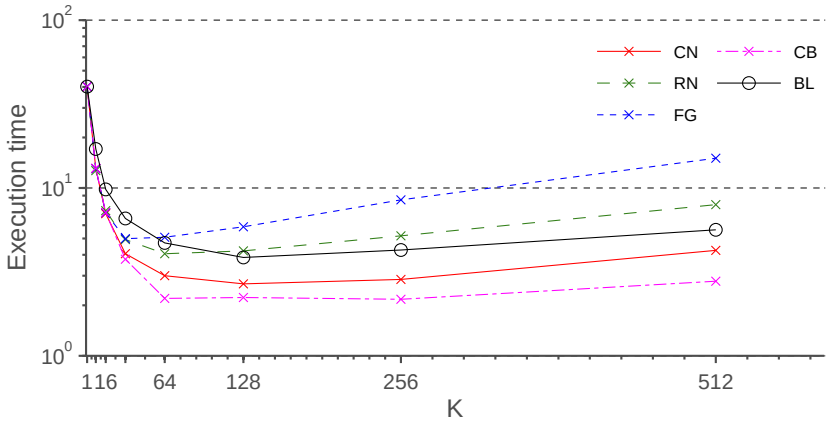
SpMxV in a more recent system:



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: $\mathbf{D}_r^{(0)} \leftarrow \mathbf{I}_{m \times m} \qquad \mathbf{D}_c^{(0)} \leftarrow \mathbf{I}_{n \times n}$
2: **for** $k = 1, 2, \ldots$ **until** convergence **do**
3: $\quad \mathbf{D}_1 \leftarrow \operatorname{diag}\left(\sqrt{\|\mathbf{r}_i^{(k)}\|_\ell}\right) \; i = 1, \ldots, m$
4: $\quad \mathbf{D}_2 \leftarrow \operatorname{diag}\left(\sqrt{\|\mathbf{c}_j^{(k)}\|_\ell}\right) \; j = 1, \ldots, n$
5: $\quad \mathbf{A}^{(k+1)} \leftarrow \mathbf{D}_1^{(k+1)} \mathbf{A} \mathbf{D}_2^{(k+1)}$
6: $\quad \mathbf{D}_r^{(k+1)} \leftarrow \mathbf{D}_r^{(k)} \mathbf{D}_1^{-1}$
7: $\quad \mathbf{D}_c^{(k+1)} \leftarrow \mathbf{D}_c^{(k)} \mathbf{D}_2^{-1}$

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).

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

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

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

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

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

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

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

Data structures and the approach (2)

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`:

```
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 ]
```

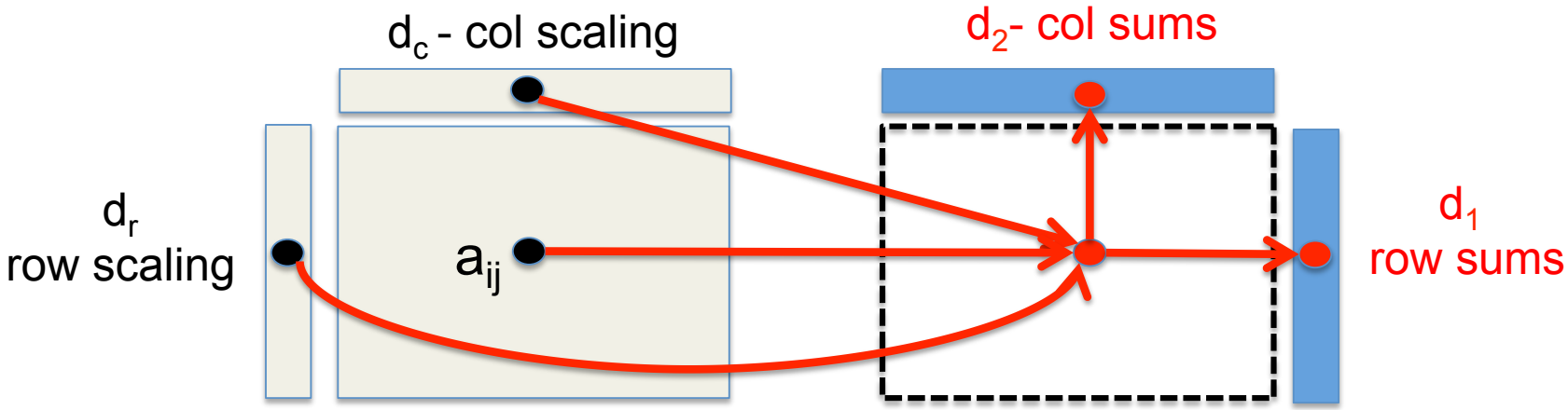
Coordinate format (COO)

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

```
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 ]
```

Shared memory parallelization

We do not store the scaled matrix; access its elements and compute (say 1-norm):



We parallelize each iteration with τ 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)

Algorithm 2: Simple parallel scaling with CRS

Input: $\mathbf{A}$: $n \times n$ input matrix in CRS format
Output: $\mathbf{d}_r$, $\mathbf{d}_c$: row and column scaling vectors

for $i = 1$ *to* n **in parallel do**
  $\mathbf{d}_r[i] \leftarrow 1$
  $\mathbf{d}_c[i] \leftarrow 1$

while *not converged* **do**
  for $i = 1$ *to* n **in parallel do**
    $\mathbf{d}_1[i] \leftarrow 0$
    $\mathbf{d}_2[i] \leftarrow 0$
init **for** $t = 1$ *to* τ **in parallel do**
    for $i = 1$ *to* n **do**
      $\mathbf{d}_2^t[i] \leftarrow 0$
put **for** $i = 1$ *to* n **in parallel do**
    ▶t is the current thread id
    $sum^t \leftarrow 0$
    for each *nonzero* a_{ij} *in row* i **do**
      $val \leftarrow \mathbf{d}_r[i] \times a_{ij} \times \mathbf{d}_c[j]$
      add val to sum^t and $\mathbf{d}_2^t[j]$
    $\mathbf{d}_1[i] \leftarrow sum^t$
get **for** $t = 1$ *to* τ **do**
    for $i = 1$ *to* n **in parallel do**
      $\mathbf{d}_2[i] \leftarrow \mathbf{d}_2[i] + \mathbf{d}_2^t[i]$
  $error \leftarrow \max(\max_i(|1 - \mathbf{d}_1[i]|), \max_i(|1 - \mathbf{d}_2[i]|))$
  if $error < \varepsilon$ **then**
    $converged \leftarrow$ **true**
  else
    for $i = 1$ *to* n **in parallel do**
      $\mathbf{d}_r[i] \leftarrow \mathbf{d}_r[i]/\sqrt{\mathbf{d}_1[i]}$
      $\mathbf{d}_c[i] \leftarrow \mathbf{d}_c[i]/\sqrt{\mathbf{d}_2[i]}$

- 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 run time by OpenMP scheduling policy.

Parallelization with CRS: Improvement (1)

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]
    ...
```

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 κ_{cut})

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

Parallelization with CRS: Improvement (1)

We need private memory only for columns touching more than one parts (call them $\mathcal{N}_C$).

- for a partition Π, the extra memory per thread is

$$\kappa_{cut}(\Pi) = \sum_{n \in \mathcal{N}_C} 1$$

- The computational overhead is

$$2\tau\kappa_{cut}(\Pi)$$

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

$\lambda = [\ 2 \quad 2 \quad 1 \quad 2 \quad 1\ ]$

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: $\mathbf{A}$: $n \times n$ 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**
    $\mathbf{d}_2[i] \leftarrow \mathbf{d}_2[i] + \mathbf{d}_2^t[i]$
...

threads in columns: $\lambda = [2\ 2\ 1\ 2\ 1]$

$$\kappa_{soed} = 2 + 2 + 2 = 6$$

- 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 κ_{soed}).
- Total computational overhead is $2\kappa_{soed}$.

Parallelization with CRS: Using atomic operations

Objective: Reduce the number of atomic operations or locks.

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

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

- Rows are partitioned among threads
  No conflict for row-sum writes
- Use private memory for columns that span multiple threads.
- Writes to d_2s, column-sum array, use atomic operations (or locks).

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

Parallelization with COO (assume 1-norm scaling)

Algorithm 3: Simple parallel scaling with COO

Input: $\mathbf{A}$: $n \times n$ input matrix in COO format
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* n **do**
    $\mathbf{d}_1^t[i] \leftarrow 0$
    $\mathbf{d}_2^t[i] \leftarrow 0$

put — **for** *each nonzero* a_{ij} **in parallel do**
  ▶t is the current thread id
  $val \leftarrow \mathbf{d}_r[i] \times a_{ij} \times \mathbf{d}_c[j]$
  add val to $\mathbf{d}_1^t[i]$ and $\mathbf{d}_2^t[j]$

get — **for** $t = 1$ *to* τ **do**
  for $i = 1$ *to* n **in parallel do**
    $\mathbf{d}_1[i] \leftarrow \mathbf{d}_1[i] + \mathbf{d}_1^t[i]$
  for $i = 1$ *to* n **in parallel do**
    $\mathbf{d}_2[i] \leftarrow \mathbf{d}_2[i] + \mathbf{d}_2^t[i]$

...

- 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 run time by OpenMP scheduling policy.

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

Experiments: Setup

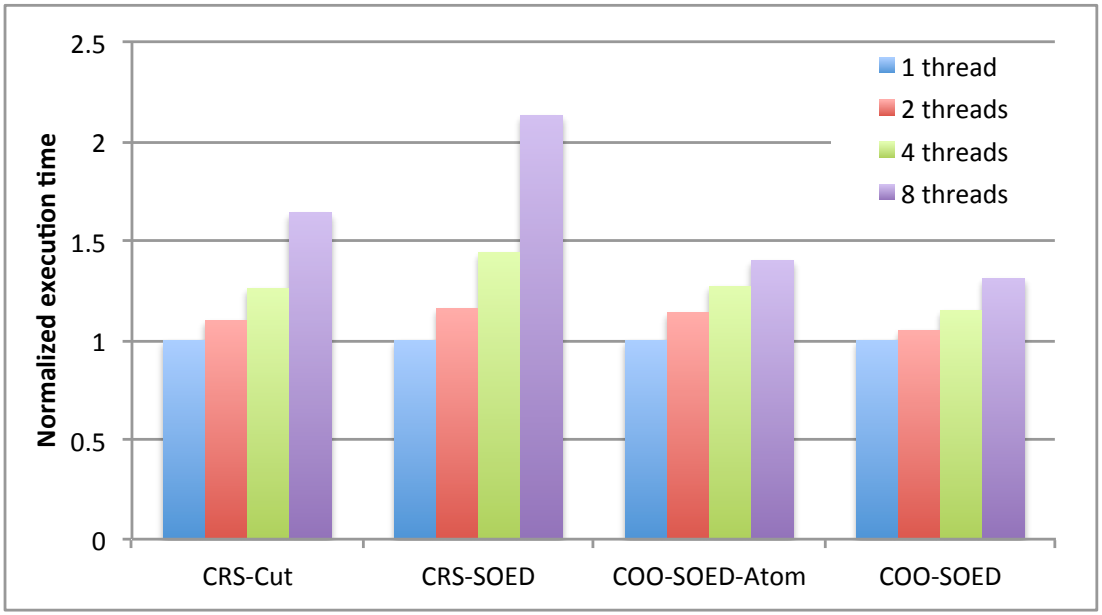
- Dual quad-core Intel Xeon (Bloomfield)
  - 48GB memory
  - 32KB L1, 256KB L2 caches per core
  - 8MB L3 cache per socket
- Dual quad-core AMD Opteron (Shanghai)
  - 32GB memory
  - 64KB L1, 512KB L2 caches per core
  - 6MB L3 cache per socket
- Algorithms are implemented in C and OpenMP
- icc 12.0 and 11.1 with -O3 optimization flag

Experiments: Matrices

Properties of the matrices used in the experiments.

Matrix	n	nnz	Avg. deg
trans5	116,835	749,800	6.42
NotreDame	325,729	929,849	2.85
rajat21	411,676	1,876,011	4.56
Hamrle3	1,447,360	5,514,242	3.81
Chebyshev4	68,121	5,377,761	78.94
pre2	659,033	5,834,044	8.85
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
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

Experiments: Effects of minimized metrics



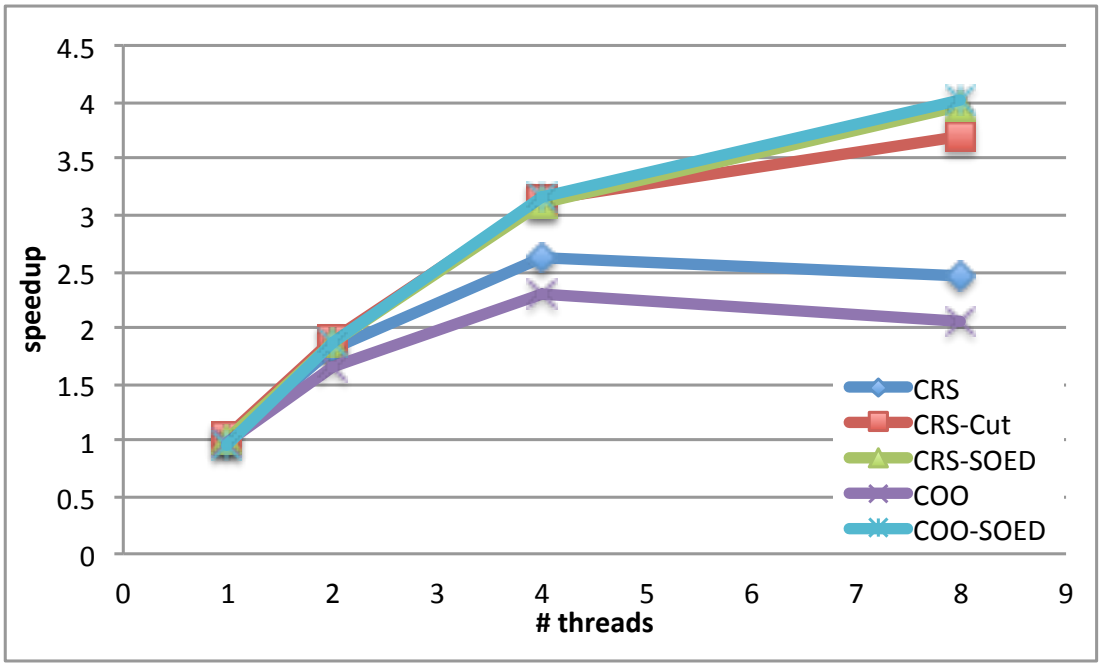
The average execution time without cut minimization but with perfect near perfect load balance

divided by the

execution time with cut minimization (using PaToH).

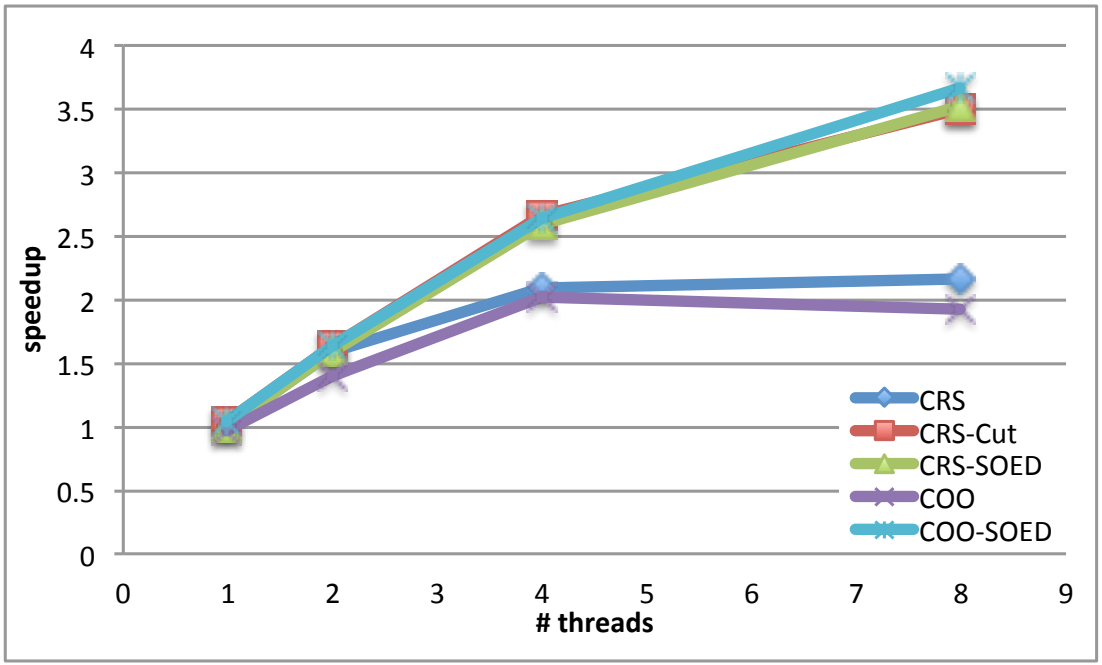
The cut-size minimized partitions lead to better performance.

Experiments: Speedups on Intel



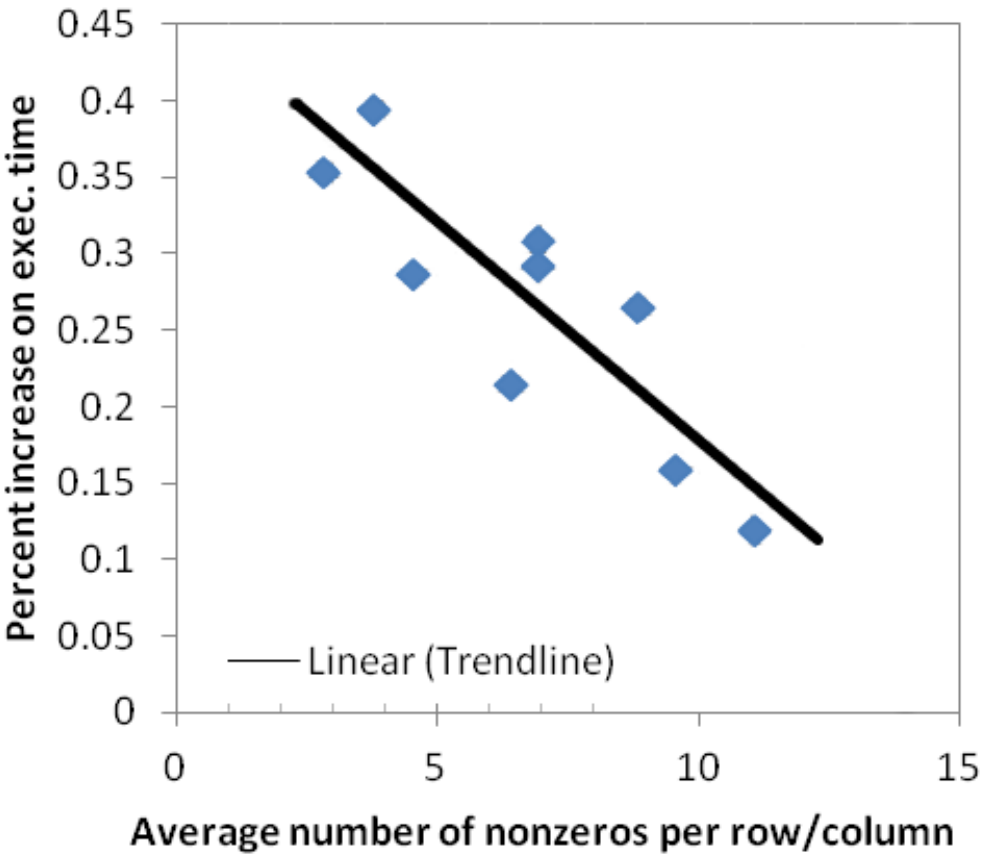
The speedups are computed by using the execution time of the CRS- and COO-based sequential algorithms, respectively.

Experiments: Speedups on AMD



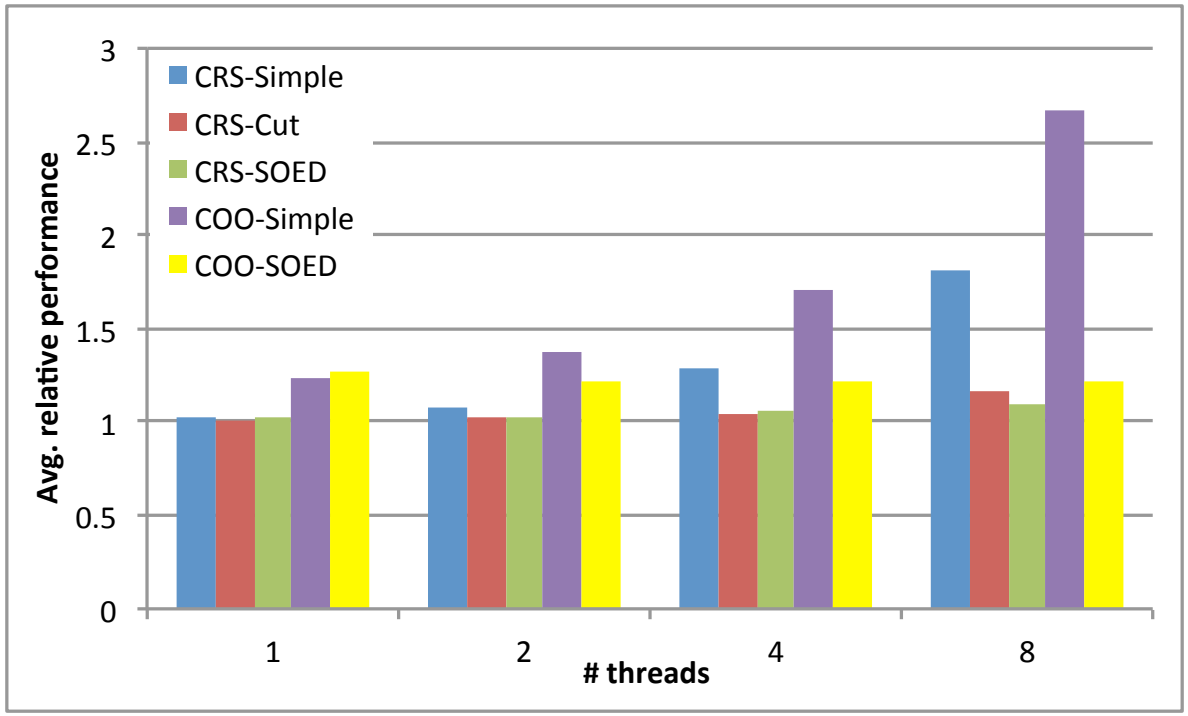
The speedups are computed by using the execution time of the CRS- and COO-based sequential algorithms, respectively.

Experiments: Speed-downs from 4 to 8



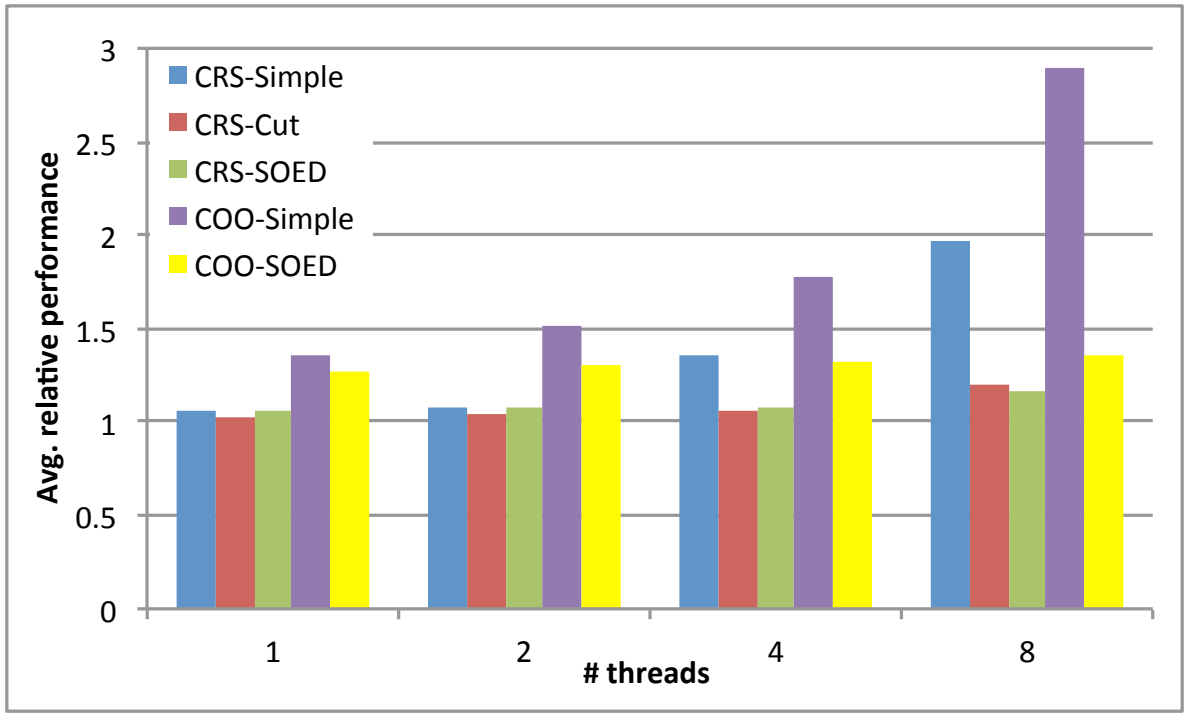
Scatter plot of the matrices for which an increase on the execution time of COO-Simple is observed when the number of threads τ is increased from 4 to 8.

Experiments: Average relative performance on Intel



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

Experiments: Average relative performance on AMD



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 $\kappa_{conn} = \sum(\lambda_j - 1)$, where λ_j is the number of processors in which the nonzeros in column j reside
- A shared memory implementation with OpenMP:
  - Memory overhead is $\kappa_{cut} = |\{j : \lambda_j > 1\}|$
  - Computational overhead is $\kappa_{soed} = \kappa_{conn} + \kappa_{cut}$
  - Number of atomic operations is κ_{soed} or κ_{conn}
- Not discussed (but can!): the $\mathcal{K}$s, the overhead functions, are well-known objective functions of the hypergraph partitioning problem. Great tools are at our disposal

Further information

Thank you for your attention.

- A symmetry preserving algorithm for matrix scaling,
  P. A. Knight, D. Ruiz, and B. Uçar, INRIA tech rep RR-7552.
- A parallel matrix scaling algorithm,
  P. R. Amestoy, I. S. Duff, D. Ruiz, and B. Uçar, VecPar'08.
- On shared memory parallelization of a matrix scaling algorithm,
  Ü. V. Çatalyürek, K. Kaya, and B. Uçar, ICPP 2012.

`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.

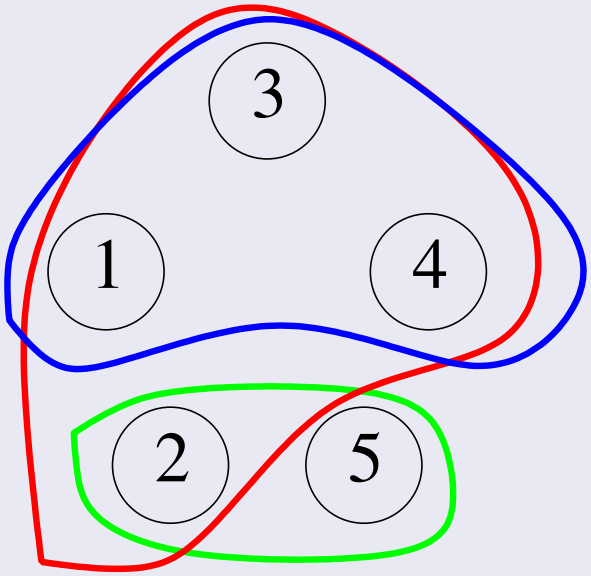
A weight $w(v)$ is associated with each vertex v.

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

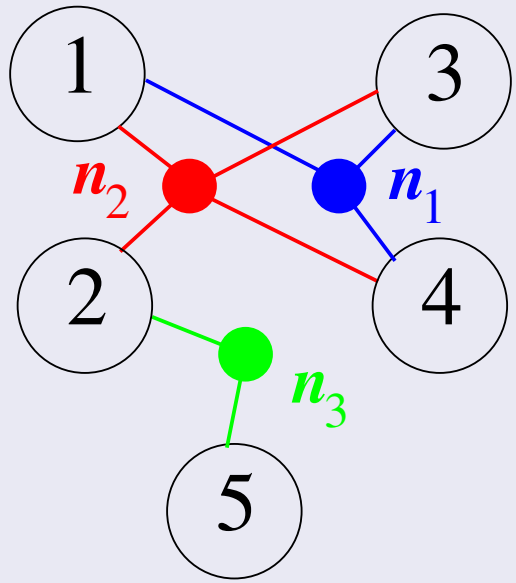
Hypergraphs: Example

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

Venn diagram-like representation



Graph-like representation



Hypergraphs: Partitioning

Partition

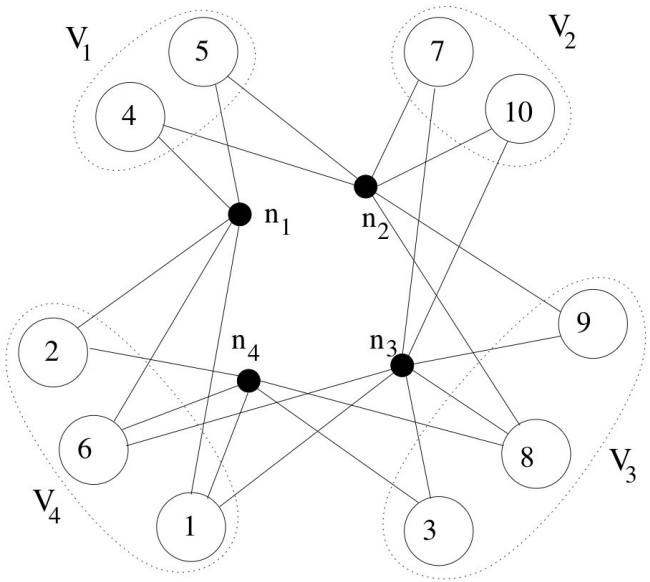
$\Pi = \{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_K\}$ is a K-way vertex partition if

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

Constraint: balanced part weights $\sum_{v \in \mathcal{V}_k} w(v) \leq (1+\varepsilon) \frac{\sum_{v \in \mathcal{V}} w(v)}{K}$.
Objective: Minimize a function of $\lambda(\cdot)$s over the cut nets.
Hypergraph partitioning problem is NP-complete.

Hypergraphs partitioning: Example

$\mathcal{H} = (\mathcal{V}, \mathcal{N})$ with 10 vertices and 4 nets, partitioned into four parts.
$V_1 = \{4, 5\}$ $V_2 = \{7, 10\}$ $V_3 = \{3, 8, 9\}$ $V_4 = \{1, 2, 6\}$

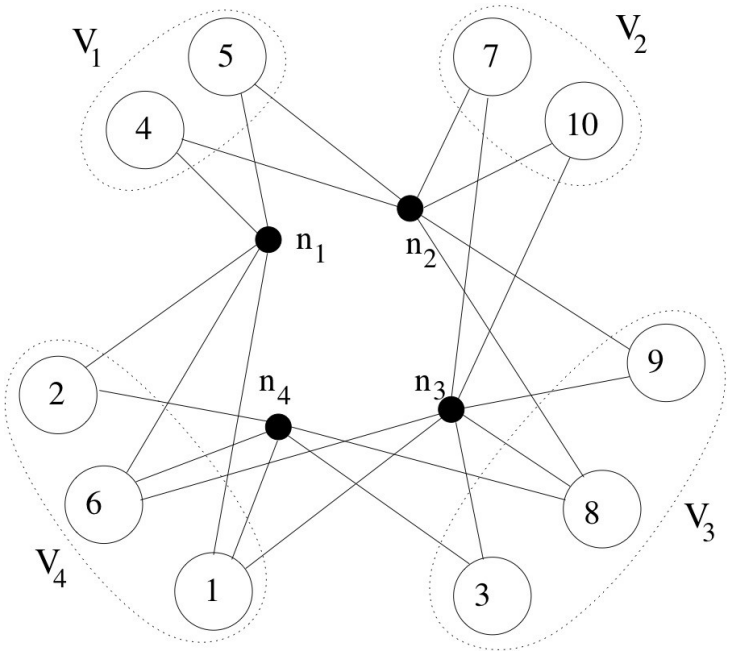


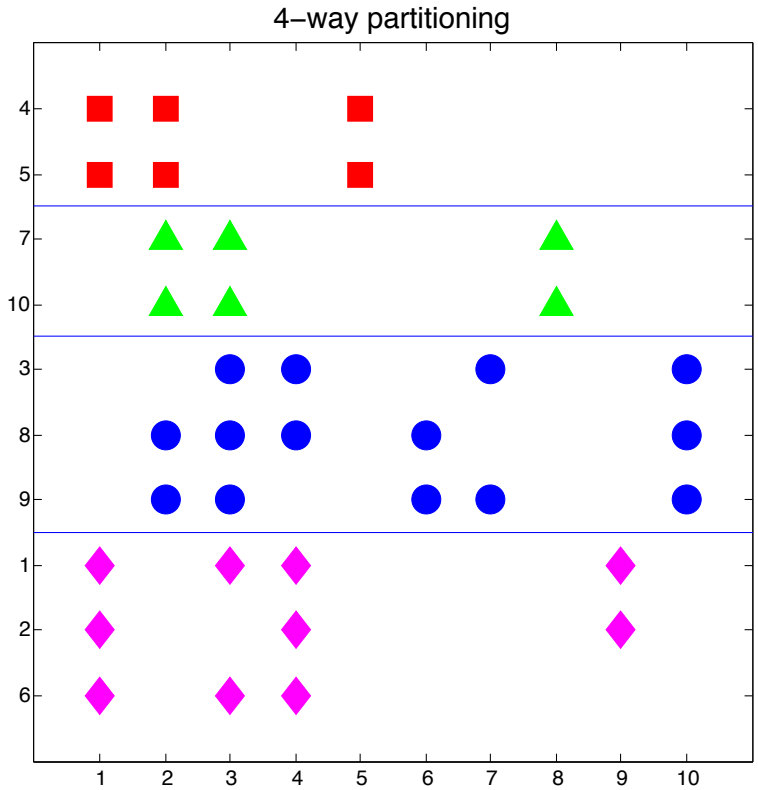
Objective functions:

$$\kappa_{cut}(\Pi) = \sum_{n \in \mathcal{N}_C} 1$$
$$= 1 + 1 + 1 + 1 = 4$$
$$\kappa_{conn}(\Pi) = \sum_{n \in \mathcal{N}_C} \lambda_n - 1$$
$$= 1 + 2 + 2 + 1 = 6$$
$$\kappa_{soed}(\Pi) = \sum_{n \in \mathcal{N}_C} \lambda_n$$
$$= 2 + 3 + 3 + 2$$

Hypergraphs partitioning: Example

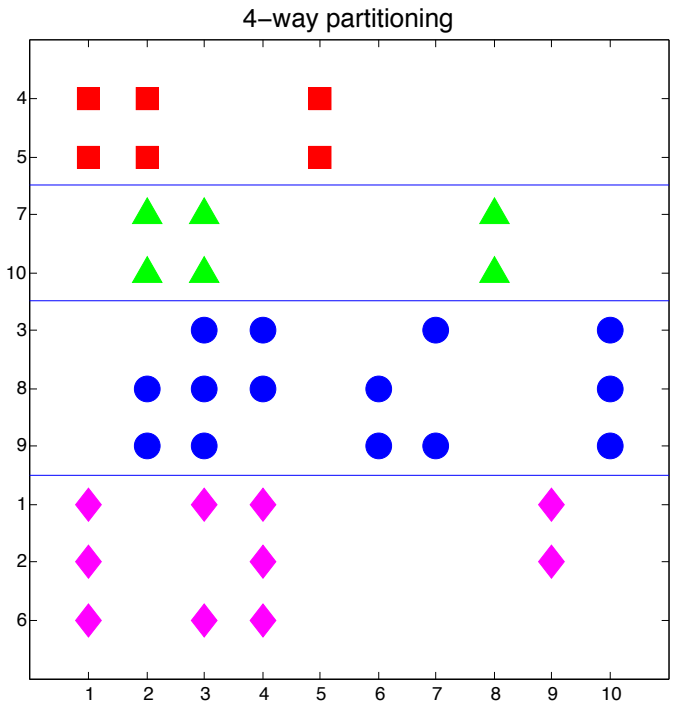
Column net model of a matrix: $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ where $\mathcal{V}$ corresponds to the rows, and $\mathcal{N}$ corresponds to the columns.





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.



Two objective functions in shared memory:

$$\kappa_{cut}(\Pi) = memory = \sum_{n \in \mathcal{N}_C} 1$$
$$= 1 + 1 + 1 + 1 = 4$$
$$\kappa_{soed}(\Pi) = atomic\ ops = \sum_{n \in \mathcal{N}_C} \lambda_n$$
$$= 2 + 3 + 3 + 2$$