A Parallel Multithreaded Sparse Triangular Linear System Solver

İlke Çuğu, Murat Manguoğlu

Department of Computer Engineering
Middle East Technical University

21.12.2019 - HPCA 2019

Outline

Motivation

Sparse linear systems are found in many applications of science and engineering:

- Electromagnetics, circuit simulations, computational fluid dynamics, etc.

Sparse triangular systems arise in...

- Sparse matrix factorizations such as LU, QR, Cholesky, etc.
- Iterative solvers such as Gauss-Seidel, Successive Over Relaxations (SOR), Symmetric SOR, etc.

Parallel Sparse Triangular System Solvers

- Level-scheduling
- Self-scheduling
- Graph coloring
- Block partitioning and decoupling
    - The proposed algorithm

The Algorithm - Origins

The Spike algorithm...

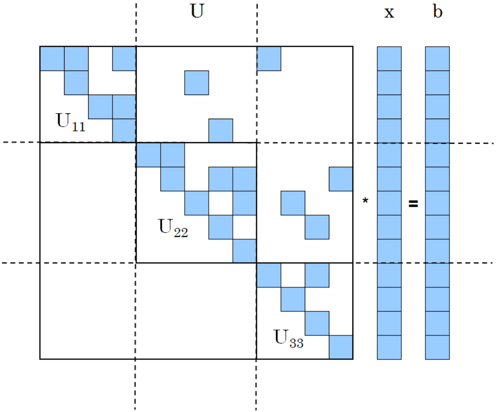
- was originally designed for general[1] and triangular[2] **banded** systems
- was generalized for general sparse systems[3]
- **is expanded and specialized for sparse triangular case by the proposed algorithm**

[1] Ahmed H Sameh and David J Kuck. "On stable parallel linear system solvers". In: Journal of the ACM (JACM) 25.1 (1978), pp. 81–91.

[2] A. Sameh and R. Brent. "Solving Triangular Systems on a Parallel Computer". In: SIAM Journal on Numerical Analysis 14.6 (1977), pp. 1101–1113.

[3] Ercan Selcuk Bolukbasi and Murat Manguoglu. "A multithreaded recursive and nonrecursive parallel sparse direct solver". In: Advances in Computational Fluid-Structure Interaction and Flow Simulation. Springer, 2016, pp. 283–292.

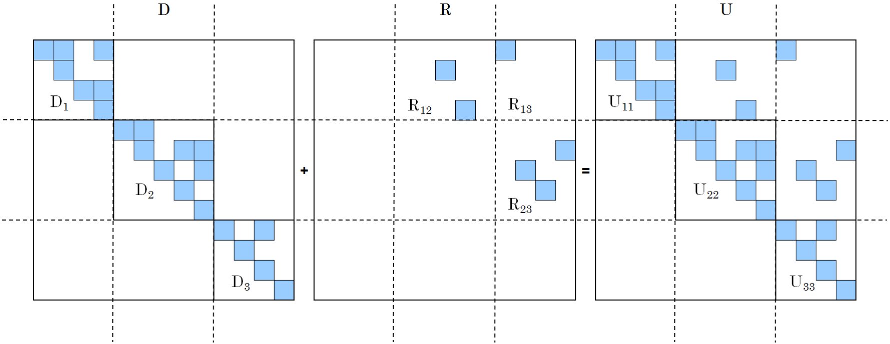
The Algorithm - The Original System



$$Ux = b$$

- The proposed algorithm is applicable to lower triangular case as well

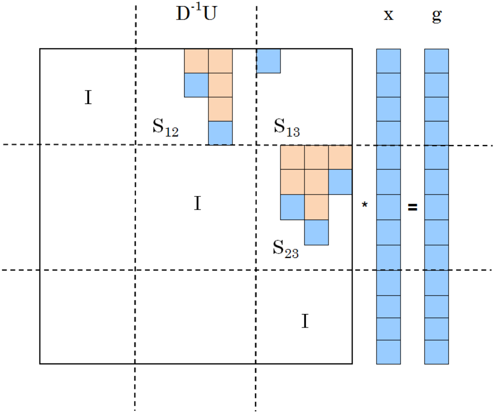
The Algorithm - Splitting U Matrix



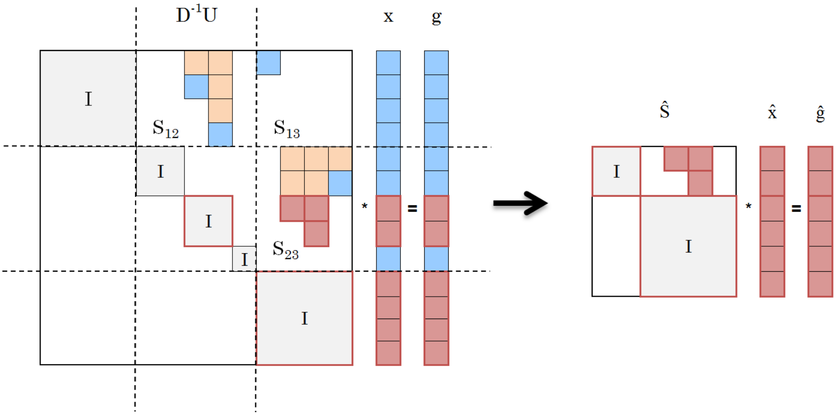
we multiply both sides of the original system $Ux = b$ with D^{-1}

$$\underbrace{D^{-1}U}_{\text{Spike matrix}} \; x = \underbrace{D^{-1}b}_{g}$$

The Algorithm - Structure of the Spike Matrix



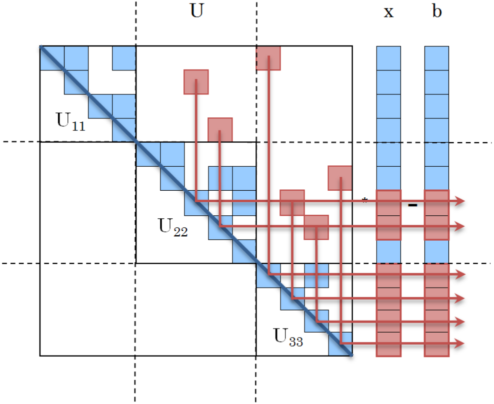
$$Sx = g$$



Construction of the reduced system

- $\widehat{S}$ is a $d \times d$ unit diagonal triangular matrix
- Solution of the reduced system requires $\mathcal{O}(nnz(\widehat{S}) - d)$ operations

The Algorithm - Dependency Elements Metaphor



The illustration of *light beams* as dependency mappings

The Algorithm - Preprocessing

Preprocessing phase covers operations independent from the right hand side vector b:

- Partitioning D matrix
- **Memory allocation for dense R and S parts**
- Compressing R into a dense form
- **Computing the partial S matrix**
- Load-balance optimization for the parallel blocks

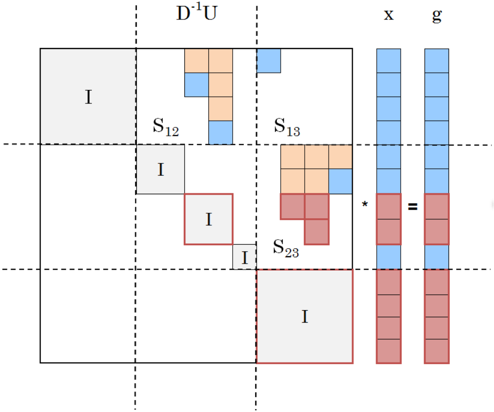
The Algorithm - Solution

Algorithm 1 PSTRSV

Input: Partitioned and factored coefficient matrix $U = DS$, reduced coefficient matrix $\hat{S}$, together with associated dependency information and b, the right-hand side vector
Output: x, solution vector of $Ux = b$

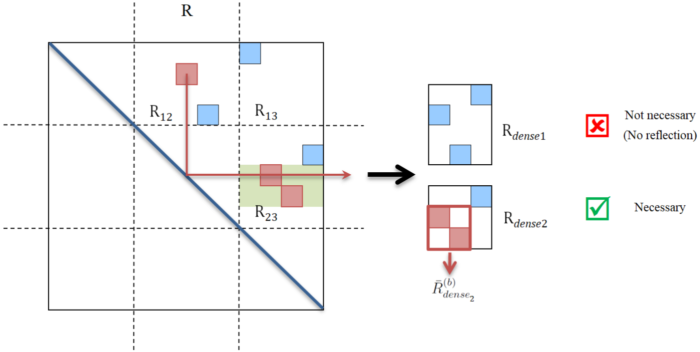
for each thread $i = 1, 2, ..., t$ **do**
  if $hasReflection_i$ or $isOptimized_i$ **then**
    Solve the triangular system $D_i^{(m;b)} g_i^{(m;b)} = b_i^{(m;b)}$ for $g_i^{(m;b)}$
  end if
  Wait until all threads reach this point
  for a single thread i **do**
    Solve the reduced system $\widehat{S}\widehat{x} = \widehat{g}$ for $\widehat{x}$
    Update the solution vector $x \leftarrow \widehat{x}$
  end for
  Wait until all threads reach this point
  if $hasDependence_i$ **then**
    $b_i^{(t;m)} := b_i^{(t;m)} - (\hat{R}_i x + P_i x_i^{(b)})$
  end if
  if $hasReflection_i$ or $isOptimized_i$ **then**
    Solve the triangular system $D_i^{(t;m)} x_i^{(t;m)} = b_i^{(t;m)}$ for $x_i^{(t;m)}$
  else
    Solve the triangular system $D_i x_i = b_i$ for x_i
  end if
end for
return x

Performance Constraints - Preprocessing



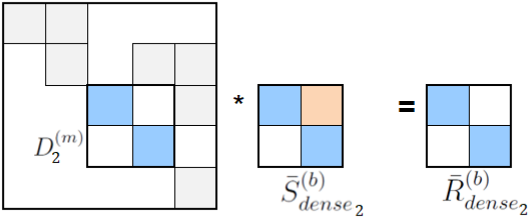
We only need to compute S matrix parts highlighted in red

Performance Constraints - $\bar{R}_i^{(b)}$ to $\bar{R}_{dense_i}^{(b)}$



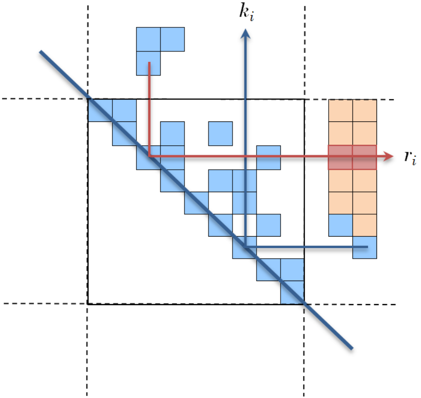
We transform the sparse $\bar{R}_i^{(b)}$ matrix to dense $\bar{R}_{dense_i}^{(b)}$ matrix

Performance Constraints - Computing $\bar{S}^{(b)}_{dense_i}$



Then, $\bar{R}^{(b)}_{dense_i}$ is used as the right hand side to compute $\bar{S}^{(b)}_{dense_i}$

Performance Constraints - Key Parameters



Two of the key performance parameters

- reflection r_i: Row index of the top-most *light beam* for each R_i
- k_i: Row index of the bottom-most *dependency element* for each R_i
- $nnz(\widehat{S}) - d$: $\#$ of off-diagonal nonzeros in $\widehat{S}$

Performance Constraints - Solution

Ideal scenarios:

- for $d_i = 0, \forall i \in \{1, 2, ..., t\}$ there is no reduced system
- for $r_i > k_i, \forall i \in \{1, 2, ..., t\}$, $\widehat{S}$ is the identity matrix

Numerical Experiments - Environment

Hardware:

- 2 sockets
- in each an Intel(R) Xeon(R) CPU E5-2650 v3 processor
- 10 cores per processor (20 cores in total)
- 16 GB of memory

Software:

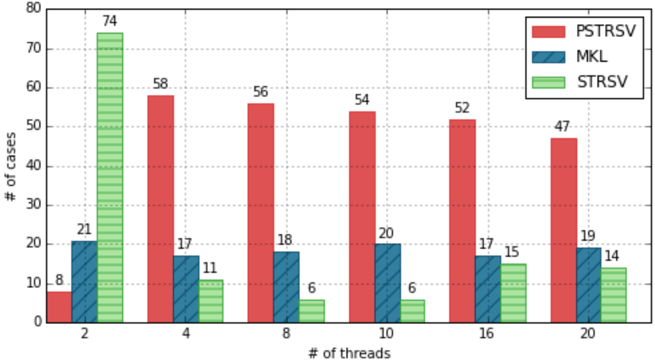
- Matrices are in Compressed Sparse Row (CSR) format
- Intel Math Kernel Library (MKL) 2018 is used
- PSTRSV is implemented in C with OpenMP
- KMP_AFFINITY = granularity = fine,compact,1,0

Numerical Experiments

In the experiments...

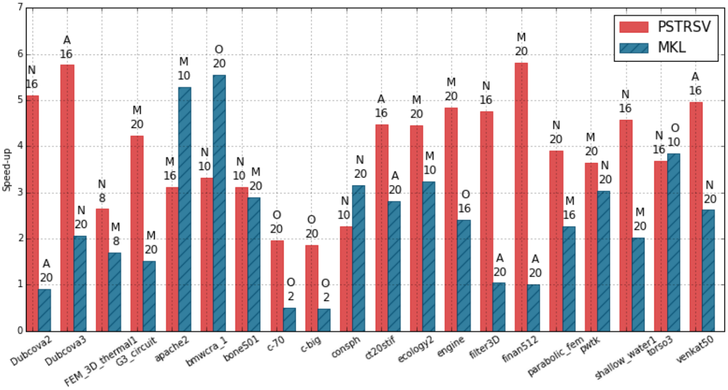
- 20 real world matrices are taken from SuiteSparse Matrix Collection
- METIS, Approximate Minimum Degree (AMD), ColPerm, Nested Dissection Permutation (NDP) and Reverse Cutthill-Mckee (RCM) orderings are used
- comparisons are done against a state-of-the-art multithreaded sparse triangular solver implementation in Intel Math Kernel Library (MKL) 2018
- each run is repeated 1,000 times and the average wallclock times are reported

Numerical Experiments - Solution



Overall performance comparison

Numerical Experiments - Solution



The highest speed-ups achieved by PSTRSV and MKL

- PSTRSV cannot amortize the preprocessing overhead in 9/120 cases
- MKL cannot amortize the preprocessing overhead in 21/120 cases

Numerical Experiments - Preprocessing

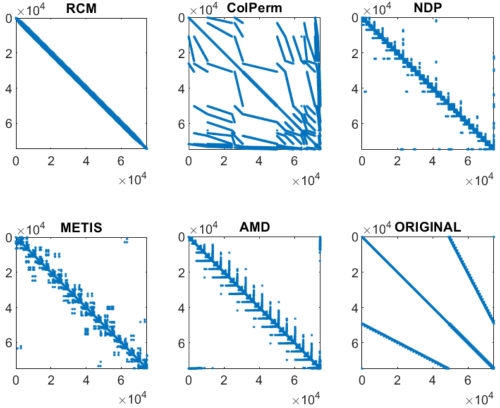
t	PSTRSV				MKL			
	min	max	avg	std	min	max	avg	std
2	1.19	37.83	13.52	9.65	4.11	251.50	78.77	60.23
4	2.28	2111.62	$\boxed{319.87}$	402.90	2.82	131.36	46.50	37.07
8	2.83	1167.28	$\boxed{227.06}$	256.19	2.17	114.80	32.89	27.84
10	2.99	824.10	$\boxed{197.22}$	210.96	2.58	118.37	31.32	27.63
16	3.03	762.25	$\boxed{192.87}$	201.35	0.19	115.57	27.41	22.40
20	3.07	770.03	$\boxed{188.94}$	199.06	0.44	264.46	35.85	35.65

Statistics of the preprocessing times of PSTRSV and MKL in milliseconds

- $t = 2$ is a special condition where $r_0 = 0$ and $k_1 = 0$ (no $\bar{R}_i^{(b)}$ or $\bar{S}_i^{(b)}$)

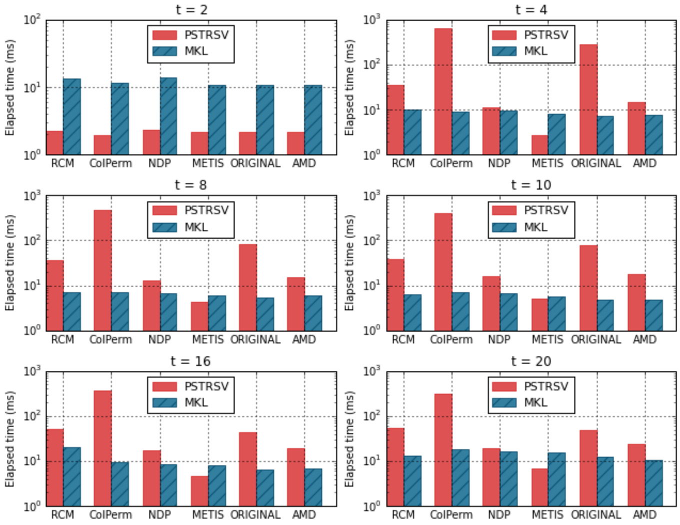
finan512

Portfolio optimization, 512 scenarios, Ed Rothberg, SGI, John Mulvey, Princeton.[4]

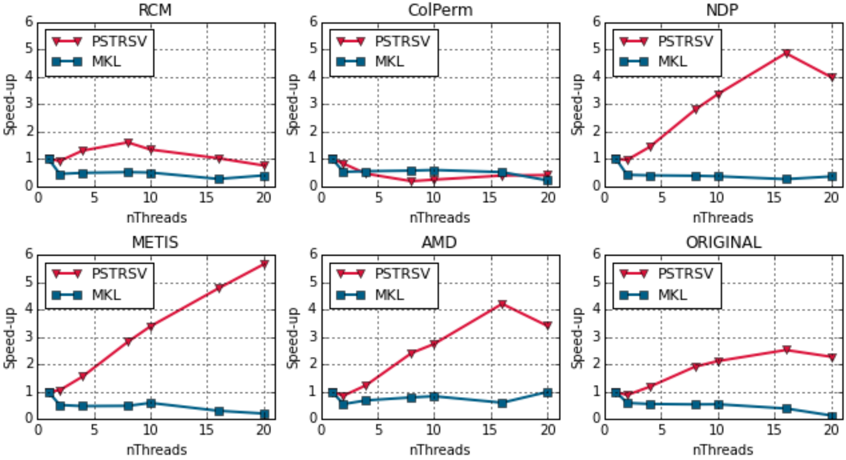


[4]The matrix and problem descriptions are obtained from: `https://www.cise.ufl.edu/research/sparse/matrices/Mulvey/finan512.html`

finan512 - Preprocessing time

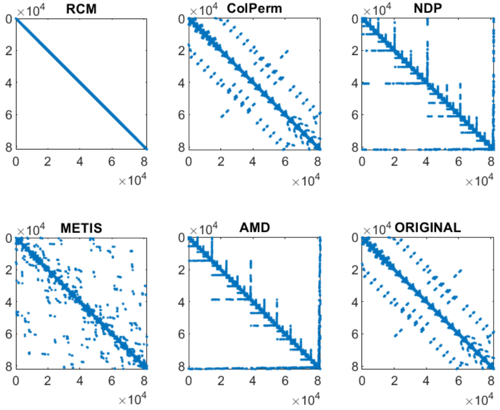


finan512 - Solution time speedup



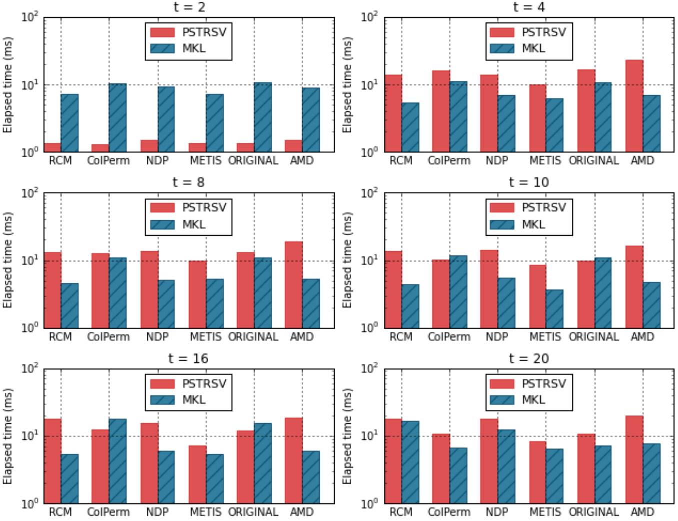
shallow_water1

Weather shallow water equations from the Max-Plank Institute of Meteorology.[5]

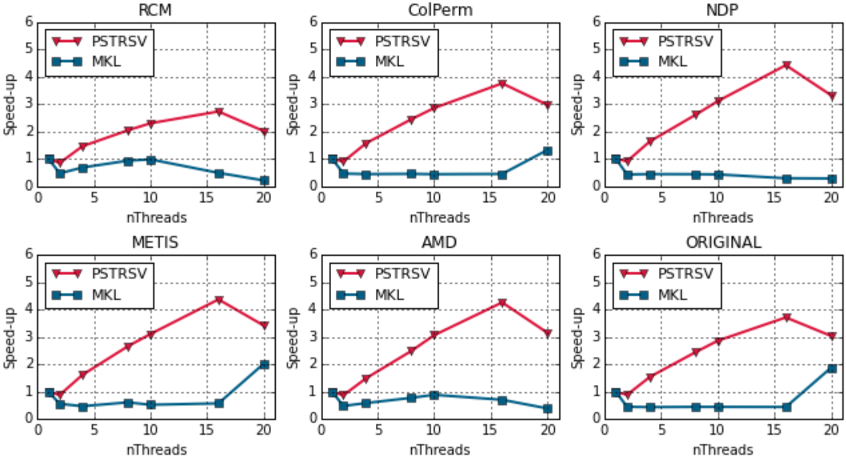


[5] The matrix and problem descriptions are obtained from: `https://www.cise.ufl.edu/research/sparse/matrices/MaxPlanck/shallow_water1.html`

shallow_water1 - Preprocessing time

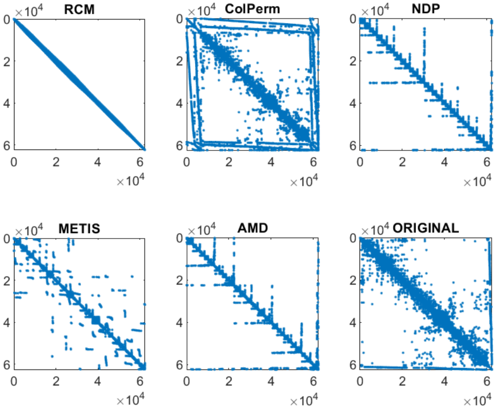


shallow_water1 - Solution time speedup



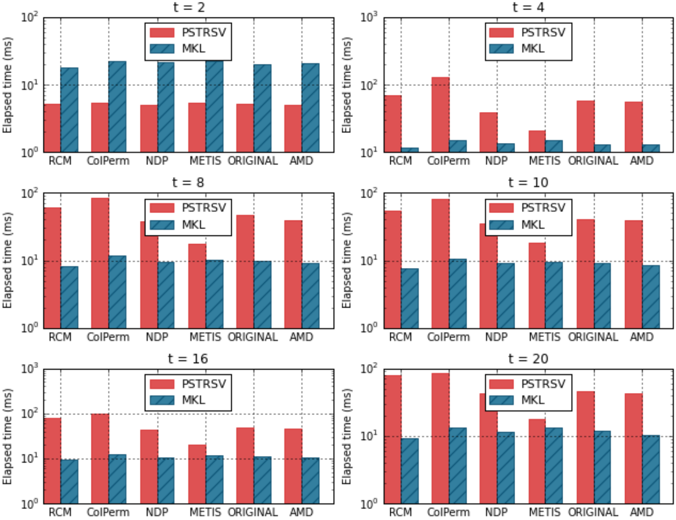
venkat50

Unstructured 2D Euler solver, V. Venkatakrishnan NASA, Timestep=50.[6]

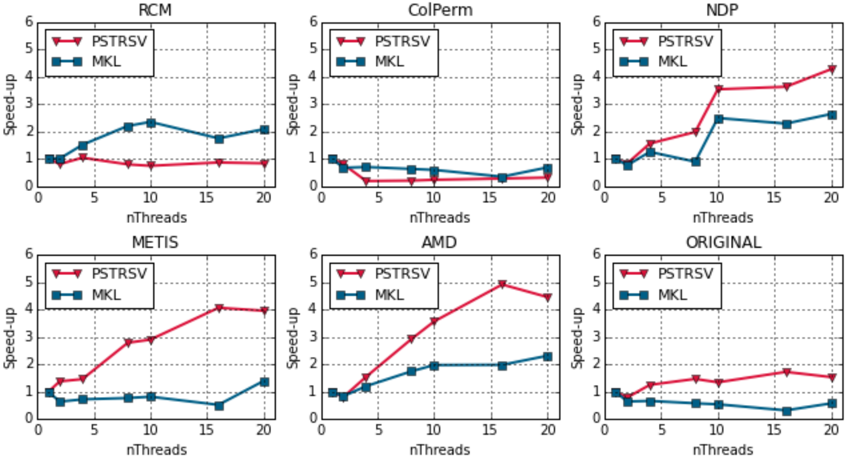


[6] The matrix and problem descriptions are obtained from:
`https://www.cise.ufl.edu/research/sparse/matrices/Simon/venkat50.html`

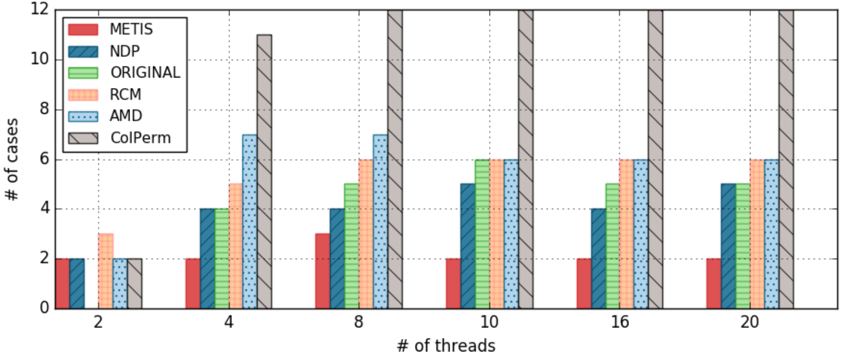
venkat50 - Preprocessing time



venkat50 - Solution time speedup



Numerical Experiments - Matrix Reordering



The number of cases where the employed reordering algorithms get memory error

Summary

spike_pstrsv...

- is implemented in C with OpenMP
- benefits from METIS, AMD and NDP orderings
- is tested with matrices taken from SuiteSparse Matrix Collection
- outperforms MKL in $\sim 80\%$ of cases by a factor of 2.47 on average
- achieves best speed-ups with..
  - 9/20 cases: NDP
  - 6/20 cases: METIS
  - 3/20 cases: AMD
  - 2/20 cases: Original
- is released under MIT license at GitHub[7]

[7] `https://github.com/cuguilke/spike_pstrsv`