Fine-Grained Parallel Algorithms for Incomplete Factorization Preconditioning

Edmond Chow
School of Computational Science and Engineering
Georgia Institute of Technology, USA

SPPEXA Symposium
TU München, Leibniz Rechenzentrum
25-27 January 2016

Incomplete factorization preconditioning

- Given sparse A, compute $LU \approx A$
  with $S = \{(i,j) \mid l_{ij} \text{ or } u_{ij} \text{ can be nonzero}\}$

- Sparse triangular solves

Many existing parallel algorithms; all generally use level scheduling.

Fine-grained parallel ILU factorization

An ILU factorization, $A \approx LU$, with sparsity pattern S has the property

$$(LU)_{ij} = a_{ij}, \quad (i,j) \in S.$$

Instead of Gaussian elimination, we compute the *unknowns*

$$\begin{aligned} l_{ij}, &\qquad i > j, &\qquad (i,j) \in S \\ u_{ij}, &\qquad i \le j, &\qquad (i,j) \in S \end{aligned}$$

using the *constraints*

$$\sum_{k=1}^{\min(i,j)} l_{ik} u_{kj} = a_{ij}, \quad (i,j) \in S.$$

If the diagonal of L is fixed, then there are $|S|$ unknowns and $|S|$ constraints.

Solving the constraint equations

The equation corresponding to (i,j) gives

$$\begin{aligned} l_{ij} &= \frac{1}{u_{jj}}\left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj}\right), \quad i > j \\ u_{ij} &= a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj}, \qquad i \leq j. \end{aligned}$$

The equations have the form $x = G(x)$. It is natural to try to solve these equations via a fixed-point iteration

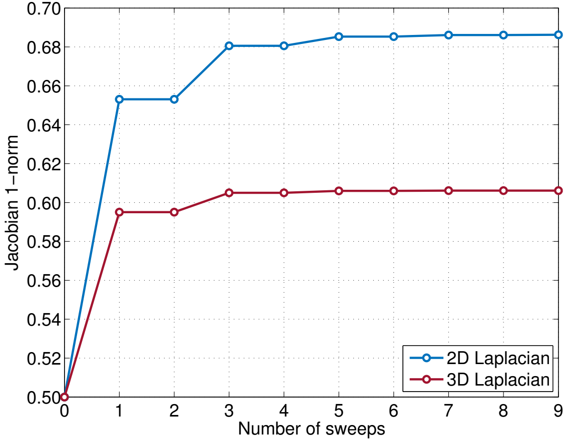
$$x^{(k+1)} = G(x^{(k)})$$

with an initial guess $x^{(0)}$.

Parallelism: can use one thread per equation for computing $x^{(k+1)}$.

Convergence is related to the Jacobian of G(x)

5-point and 7-point centered-difference approximation of the Laplacian



Values are independent of the matrix size

Measuring convergence of the nonlinear iterations

Nonlinear residual

$$\|(A - LU)_S\|_F = \left[\sum_{(i,j)\in S} \left(a_{ij} - \sum_{k=1}^{\min(i,j)} l_{ik} u_{kj} \right)^2 \right]^{1/2}$$

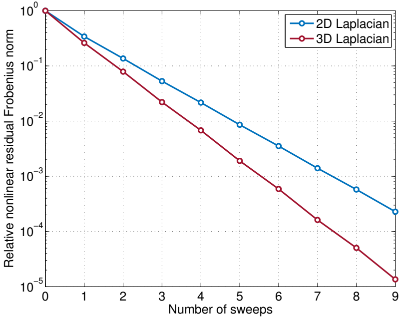
(or some other norm)

ILU residual

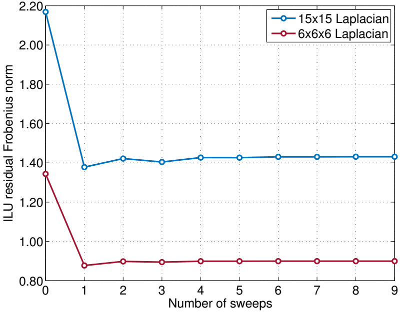
$$\|A - LU\|_F$$

Convergence of the preconditioned linear iterations is known to be strongly related to the ILU residual

Laplacian problem



Relative nonlinear residual norm



ILU residual norm

Asynchronous updates for the fixed point iteration

In the general case, $x = G(x)$

$$
\begin{aligned}
x_1 &= g_1(x_1, x_2, x_3, x_4, \cdots, x_m) \\
x_2 &= g_2(x_1, x_2, x_3, x_4, \cdots, x_m) \\
x_3 &= g_3(x_1, x_2, x_3, x_4, \cdots, x_m) \\
x_4 &= g_4(x_1, x_2, x_3, x_4, \cdots, x_m) \\
&\vdots \\
x_m &= g_m(x_1, x_2, x_3, x_4, \cdots, x_m)
\end{aligned}
$$

Synchronous update: all updates use components of x at the same "iteration"

Asynchronous update: updates use components of x that are currently available

- easier to implement than synchronous updates (no extra vector)
- convergence can be faster if overdecomposition is used: more like Gauss-Seidel than Jacobi (practical case on GPUs and Intel Xeon Phi)

Overdecomposition for asynchronous methods

Consider the case where there are p (block) equations and p threads (no overdecomposition)

- Convergence of asynchronous iterative methods is often believed to be *worse* than that of the synchronous version (the latest information is likely from the "previous" iteration (like Jacobi), but could be even older)

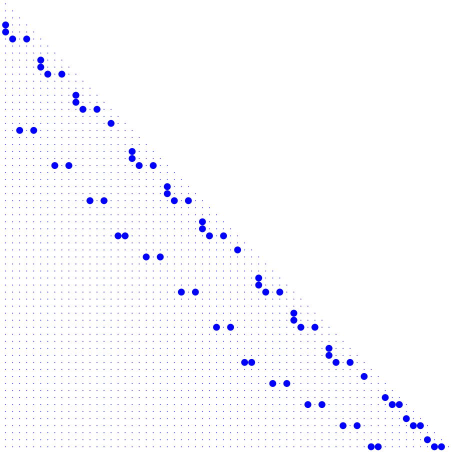
However, if we **overdecompose** the problem (more tasks than threads), then convergence can be *better* than that of the synchronous version. Note: on GPUs, we always have to decompose the problem: more thread blocks than multiprocessors (to hide memory latency).

- Updates tend to use fresher information than when updates are simultaneous
- Asynchronous iteration becomes more like Gauss-Seidel than like Jacobi
- Not all updates are happening at the same time (more uniform bandwidth usage, not bursty)

$x = G(x)$ can be ordered into strictly lower triangular form

$$
\begin{aligned}
x_1 &= g_1 \\
x_2 &= g_2(x_1) \\
x_3 &= g_3(x_1, x_2) \\
x_4 &= g_4(x_1, x_2, x_3) \\
&\vdots \\
x_m &= g_m(x_1, \cdots, x_{m-1})
\end{aligned}
$$

Structure of the Jacobian



5-point matrix on 4x4 grid

For all problems, the Jacobian is sparse and its diagonal is all zeros.

Update order for triangular matrices with overdecomposition

Consider solving with a triangular matrix using asynchronous iterations

$$\begin{pmatrix} a_{11} & 0 & 0 & 0 & 0 & 0 \\ a_{21} & a_{22} & 0 & 0 & 0 & 0 \\ a_{31} & a_{32} & a_{33} & 0 & 0 & 0 \\ a_{41} & a_{42} & a_{43} & a_{44} & 0 & 0 \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} & 0 \\ a_{61} & a_{62} & a_{63} & a_{64} & a_{65} & a_{66} \end{pmatrix}$$

- Order in which variables are updated can affect convergence rate
- Want to perform updates in top-to-bottom order for lower triangular system

On GPUs, don't know how thread blocks are scheduled

But, on NVIDIA K40, the ordering appears to be deterministic (on earlier GPUs, ordering appears non-deterministic)

(Joint work with Hartwig Anzt)

2D FEM Laplacian, $n = 203841$, RCM ordering, 240 threads on Intel Xeon Phi

	Level 0			Level 1			Level 2		
Sweeps	PCG iter	nonlin resid	ILU resid	PCG iter	nonlin resid	ILU resid	PCG iter	nonlin resid	ILU resid
0	404	1.7e+04	41.1350	404	2.3e+04	41.1350	404	2.3e+04	41.1350
1	318	3.8e+03	32.7491	256	5.7e+03	18.7110	206	7.0e+03	17.3239
2	301	9.7e+02	32.1707	207	8.6e+02	12.4703	158	1.5e+03	6.7618
3	298	1.7e+02	32.1117	193	1.8e+02	12.3845	132	4.8e+02	5.8985
4	297	2.8e+01	32.1524	187	4.6e+01	12.4139	127	1.6e+02	5.8555
5	297	4.4e+00	32.1613	186	1.4e+01	12.4230	126	6.5e+01	5.8706
IC	297	0	32.1629	185	0	12.4272	126	0	5.8894

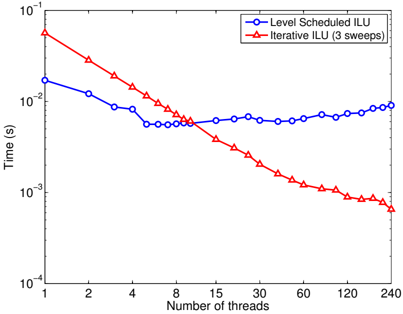
Very small number of sweeps required

Univ. Florida sparse matrices (SPD cases)
240 threads on Intel Xeon Phi

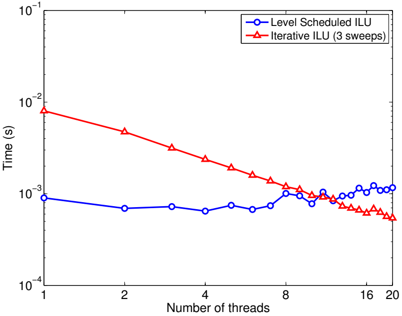
	Sweeps	Nonlin Resid	PCG iter
af_shell3	0	1.58e+05	852.0
	1	1.66e+04	798.3
	2	2.17e+03	701.0
	3	4.67e+02	687.3
	IC	0	685.0
thermal2	0	1.13e+05	1876.0
	1	2.75e+04	1422.3
	2	1.74e+03	1314.7
	3	8.03e+01	1308.0
	IC	0	1308.0
ecology2	0	5.55e+04	2000+
	1	1.55e+04	1776.3
	2	9.46e+02	1711.0
	3	5.55e+01	1707.0
	IC	0	1706.0
apache2	0	5.13e+04	1409.0
	1	3.66e+04	1281.3
	2	1.08e+04	923.3
	3	1.47e+03	873.0
	IC	0	869.0

	Sweeps	Nonlin Resid	PCG iter
G3_circuit	0	1.06e+05	1048.0
	1	4.39e+04	981.0
	2	2.17e+03	869.3
	3	1.43e+02	871.7
	IC	0	871.0
offshore	0	3.23e+04	401.0
	1	4.37e+03	349.0
	2	2.48e+02	299.0
	3	1.46e+01	297.0
	IC	0	297.0
parabolic_fem	0	5.84e+04	790.0
	1	1.61e+04	495.3
	2	2.46e+03	426.3
	3	2.28e+02	405.7
	IC	0	405.0

Timing comparison, ILU(2) on 100 × 100 grid (5-point stencil)



Intel Xeon Phi



Intel Xeon E5-2680v2, 20 cores

Results for NVIDIA Tesla K40c

	PCG iteration counts for given number of sweeps							Timings [ms]		
	IC	0	1	2	3	4	5	IC	5 swps	s/up
apache2	958	1430	1363	1038	965	960	958	61.	8.8	6.9
ecology2	1705	2014	1765	1719	1708	1707	1706	107.	6.7	16.0
G3_circuit	997	1254	961	968	993	997	997	110.	12.1	9.1
offshore	330	428	556	373	396	357	332	219.	25.1	8.7
parabolic_fem	393	763	636	541	494	454	435	131.	6.1	21.6
thermal2	1398	1913	1613	1483	1341	1411	1403	454.	15.7	28.9

IC denotes the exact factorization computed using the NVIDIA cuSPARSE library.

(Joint work with Hartwig Anzt)

Sparse triangular solves with ILU factors

Iterative and approximate triangular solves

Trade accuracy for parallelism

Approximately solve the triangular system $Rx = b$

$$x_{k+1} = (I - D^{-1}R)x_k + D^{-1}b$$

where D is the diagonal part of R. In general, $x \approx p(R)b$ for a polynomial $p(R)$.

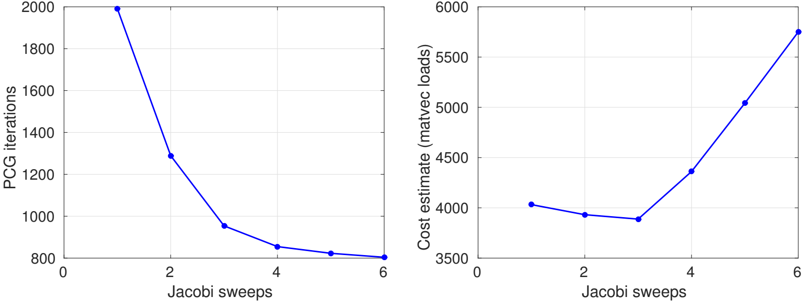
- implementations depend on SpMV
- iteration matrix $G = I - D^{-1}R$ is strictly triangular and has spectral radius 0 (trivial asymptotic convergence)
- for fast convergence, want the norm of G to be small
- R from stable ILU factorizations of physical problems are often close to being diagonally dominant
- preconditioner is fixed linear operator in non-asynchronous case

Related work

Above Jacobi method is almost equivalent to approximating the ILU factors with a truncated Neumann series (e.g., van der Vorst 1982)

Stationary iterations for solving with ILU factors using $x_{k+1} = x_k + Mr_k$, where M is a sparse approximate inverse of the triangular factors (Bräckle and Huckle 2015)

Hook_1498



FEM; equations: 1,498,023; non-zeroes: 60,917,445

IC-PCG with exact and iterative triangular solves on Intel Xeon Phi

	IC	PCG iterations		Timing (seconds)		Num.
	level	Exact	Iterative	Exact	Iterative	sweeps
af_shell3	1	375	592	79.59	23.05	6
thermal2	0	1860	2540	120.06	48.13	1
ecology2	1	1042	1395	114.58	34.20	4
apache2	0	653	742	24.68	12.98	3
G3_circuit	1	329	627	52.30	32.98	5
offshore	0	341	401	42.70	9.62	5
parabolic_fem	0	984	1201	15.74	16.46	1

Table: Results using 60 threads on Intel Xeon Phi. Exact solves used level scheduling.

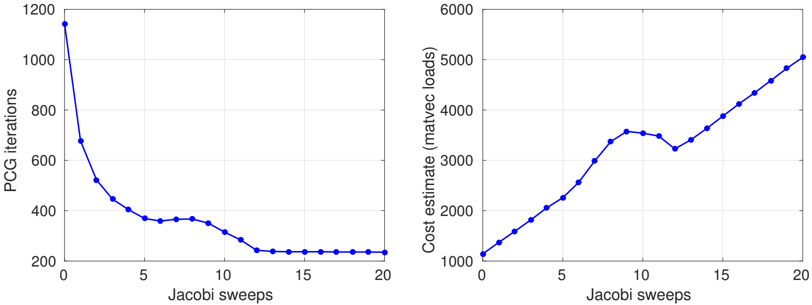
ILU(0)-GMRES(50) with exact and iterative triangular solves on Telsa K40c GPU

	Exact solve	Jacobi sweeps 1	2	3	4	5	6
chipcool0	75	201	108	95	78	86	84
	1.2132	0.2777	0.1710	0.1761	**0.1561**	0.1965	0.2127
stomach	10	22	13	12	11	11	10
	0.4789	0.0857	**0.0749**	0.0952	0.1110	0.1351	0.1443
venkat01	15	49	32	24	20	18	17
	0.5021	0.1129	0.0909	**0.0855**	0.0874	0.0940	0.1034

Table: Iteration counts (first line) and timings [s] (second line). Exact solves used cuSPARSE. RCM ordering was used. Performance will depend on performance of sparse matvec.

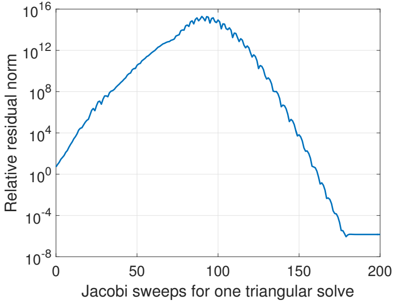
Results from Hartwig Anzt

Geo_1438



FEM; equations: 1,437,960; non-zeroes: 63,156,690

BCSSTK24 – solve with L



BCSSTK24 – solve with L

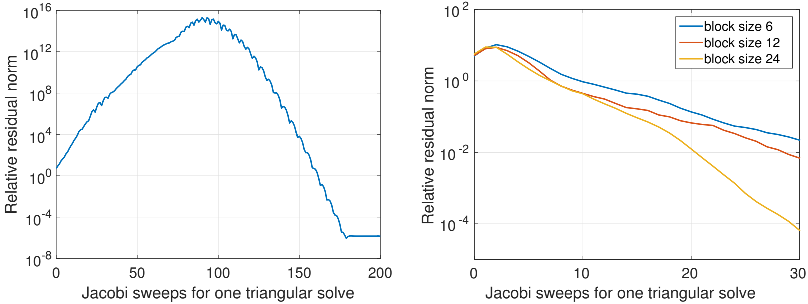


Figure: Relative residual norm in triangular solve with the lower triangular incomplete Cholesky factor (level 1), without blocking and with blocking.

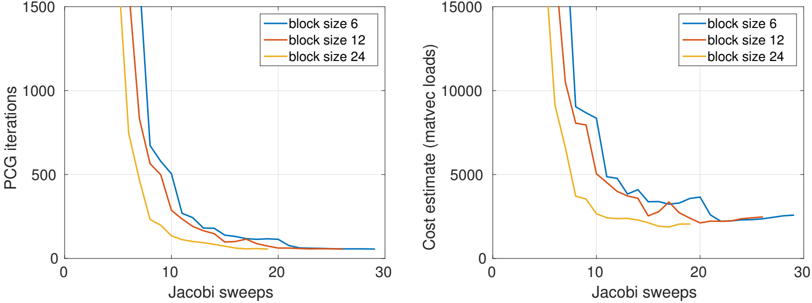
Block Jacobi for Triangular solves

$$x_{k+1} = (I - D^{-1}R)x_k + D^{-1}b$$

Blocking strategy

- Group variables associated with a grid point or element
- Further group these variables if necessary

BCSSTK24



PCG iterations
1500
1000
500
0
0
10
20
30
Jacobi sweeps
block size 6
block size 12
block size 24
Cost estimate (matvec loads)
15000
10000
5000
0
0
10
20
30
Jacobi sweeps
block size 6
block size 12
block size 24

Practical use

Will Jacobi sweeps work for my triangular factor?

- Easy to tell if it won't work: try a few iterations and check reduction in residal norm.
- Example: use 30 iterations and check if relative residual norm goes below 10^{-2}

How many Jacobi sweeps to use?

- Hard to tell beforehand.
- No way to accurately measure residal norm in asynchronous iterations (i.e., residual at what iteration?).
- Number of Jacobi sweeps could be dynamically adapated (restart Krylov method with preconditioner using a different number of sweeps)

For an arbitrary problem, could Jacobi work?

If so, how many sweeps are needed? (What is a bound?)

Comprehensive tests on SPD problems

Test all SPD matrices in the University of Florida Sparse Matrix collection with *nrows* $\geq$ 1000 except diagonal matrices: 171 matrices.

Among these, find all problems that can be solved with IC(0) or IC(1).

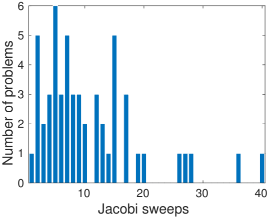
	IC(0)	IC(1)
Total	73	86
Num solved using iter trisol	54 (74%)	52 (60%)
Num solved using block iter trisol	68 (93%)	70 (81%)

Block iterative triangular solves used supervariable amalgamation with block size 12.

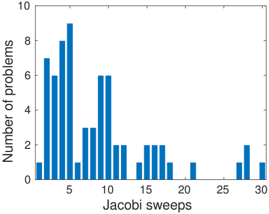
Caveat: IC may not be the best preconditioner for all these problems

Number of Jacobi sweeps for solution in same number of PCG iterations when exact solves are used

Iterative triangular solves, IC(0) and IC(1)



Block iterative triangular solves (max block size 12)



Conclusions

Fine-grained algorithm for ILU factorization

- More parallel work, can use lots of threads
- Dependencies between threads are obeyed asynchronously
- Does not use level scheduling or reordering of the matrix
- Each entry of L and U updated iteratively in parallel
- Do not need to solve the bilinear equations exactly
- Method can be used to *update* a factorization given a sligtly changed A
- Fixed point iteratins can be performed asynchronously (helps tolerate latency and performance irregularities)

Solving sparse triangular systems from ILU via iteration

- Blocking can add robustness by reducing the non-normality of the iteration matrices
- Must try to reduce DRAM bandwidth of threads during iterations

An Intel Parallel Computing Center grant is gratefully acknowledged. This material is based upon work supported by the U.S. DOE Office of Science, Office of Advanced Scientific Computing Research, Applied Mathematics program under Award Number DE-SC-0012538.