Some repetition

Note: many issues related to iterative solution of linear systems of equations are contradictory:

- numerical efficiency vs computational efficiency
- numerical efficiency vs parallelization

Example 1: Multiphace flow simulation (2D)

	Block preconditioner			Direct solver (MUMPS)		
DOF	$N1/N2$	$time(s)$	MB	$N1$	$time(s)$	MB
131 072	4/10	16.98	185	3	7.2	352
528 392	4/10	72.61	646	3	53.4	1 409
1 176 578	4/10	170	1 429	3	193.75	3 126
2 097 152	4/10	306.05	2 587	*Out of memory*		

Multiface flow: Direct vs iterative: run time and memory consumption

$72.61/16.98 = 4.28$ $53.4/7.2 = 7.4$

$170/72.61 = 2.34$ $193.75/53.4 = 3.62$

$306.05/170 = 1.8$?

Scalability of sparse direct solvers, cont.

Example 2: Linear elasticity, 2D

	Assembly time (s)		Solution time (s)		
N	Abaqus	Iterative	Abaqus	Iterative	
				time	iterations
			2D		
6043	1	0.2178	1.098	1.02 (0.4863)	13 (1,1)
23603	3.326	0.8857	4.718	4.225 (1.995)	12 (1,1)
93283	13.02	3.978	18.05	19.38 (9.813)	11 (2,1)
370883	50.54	17.71	72.98	89.34 (49.43)	11 (2,1)
1479043	269.1	77.7	317.5	431.8 (257.6)	12 (2,1)

4.2969	4.1422
3.8258	4.5870
4.0432	4.6099

Scalability of sparse direct solvers, cont.

Example 2: Linear elasticity, 3D

	Assembly time (s)		Solution time (s)		
N	Abaqus	Iterative	Abaqus	Iterative	
				time	iterations
			2D		
6043	1	0.2178	1.098	1.02 (0.4863)	13 (1,1)
23603	3.326	0.8857	4.718	4.225 (1.995)	12 (1,1)
93283	13.02	3.978	18.05	19.38 (9.813)	11 (2,1)
370883	50.54	17.71	72.98	89.34 (49.43)	11 (2,1)
1479043	269.1	77.7	317.5	431.8 (257.6)	12 (2,1)
			3D		
12512	1.525	1.899	3.049	8.009 (3.465)	12 (2,1)
89700	14.09	8.756	43.29	63.34 (33.08)	13 (2,1)
678116	110.3	65.8	1347	749.3 (506.8)	15 (4,1)

Hybrid solvers!

Preconditioning techniques - the task to combine numerical and parallel efficiency

Some repetition, cont.

Assume A, B and $\mathbf{b}$ are distributed and the initial guess $\mathbf{x}^{(0)}$ is replicated.

$$
\begin{array}{lrcll}
 & \mathbf{g}^{(0)} & = & A\mathbf{x}^{(0)} - \mathbf{b}, & \mathbf{g}^{(0)} = replicate(\mathbf{g}^{(0)}) \\
 & \mathbf{h} & = & C^{-1}\mathbf{g}^{(0)} & \\
 & \delta_0 & = & (\mathbf{g}^{(0)}, \mathbf{h}) & \mathbf{h} = replicate(\mathbf{h}) \\
 & \mathbf{d}^{(0)} & = & -\mathbf{h} & \\
\textit{For } k = 0, 1, \cdots & \textit{until convergence} & & & \\
(1) & \mathbf{h} & = & A\mathbf{d}^{(k)} & \\
(2) & \tau & = & \delta_0/(\mathbf{h}, \mathbf{d}^{(k)}) & \\
(3) & \mathbf{x}^{(k+1)} & = & \mathbf{x}^{(k)} + \tau\mathbf{d}^{(k)} & \\
(4) & \mathbf{g}^{(k+1)} & = & \mathbf{g}^{(k)} + \tau\mathbf{h}, & \mathbf{g}^{(k+1)} = replicate(\mathbf{g}^{(k+1)}) \\
(5) & \mathbf{h} & = & C^{-1}\mathbf{g}^{(k+1)}, & \\
(6) & \delta_1 & = & (\mathbf{g}^{(k+1)}, \mathbf{h}) & \mathbf{h} = replicate(\mathbf{h}) \\
(7) & \beta & = & \delta_1/\delta_0, & \delta_0 = \delta_1 \\
(8) & \mathbf{d}^{(k+1)} & = & -\mathbf{h} + \beta\mathbf{d}^{(k)} &
\end{array}
$$

Preconditioners

Requirements:

- Numerical efficiency
  - The condition number $\varkappa(C^{-1}A)$ should be as small as possible and independent of problem, discretization and method parameters.
    Wishes: $\varkappa(C^{-1}A) = O(1)$
    Eigenvalues clusteres in small intervals on the real axes or in a few tight clusters, well separated from the origin.
- Computational efficiency
  - The construction of C should be computationally cheap
  - The solution of systems with C should be much cheaper (easier) than with A
- Papallel efficiency
  - Both the construction and the solution with the preconditioner should be parallelizable

Clearly the goals are contradicting.

Types of preconditioners

- Left preconditioning $A\mathbf{x} = \mathbf{b} \Longrightarrow C^{-1}A\mathbf{x} = C^{-1}\mathbf{b}$
  $eig(XY) = eig(YX)$ up to some zero eigenvalues.
- Right Preconditioning $A\mathbf{x} = \mathbf{b} \Longrightarrow AC^{-1}\mathbf{y} = \mathbf{b}, \mathbf{x} = C^{-1}\mathbf{y}$
- Symmetric preconditioning

$$A\mathbf{x} = \mathbf{b} \Longrightarrow C_1^{-1}AC_2^{-1}\mathbf{y} = C_1^{-1}\mathbf{b}, \mathbf{x} = C_2^{-1}\mathbf{y}$$

- Approximate inverse (multiplicative preconditioning)
  $C \approx A^{-1}$. Then no solution but only multiplication with C occur.
- Implicitly defined preconditioners $[C^{-1}]A\mathbf{x} = [C^{-1}]\mathbf{b}$
- Variable (nonlinear) preconditioners C changes from one iteration to another or a number of times through the iterative process.
  Flexible GMRES, GCG, GCR.
- Inner-outer iterations (inner stopping tolerance)

Types of preconditioners

- 'Given-the-matrix' - only the matrix is given and the origin of the problem is not known or is not to be used during the solution process.
  Very general, thus, expected to be less numerically efficient.
- 'Given-the-problem' - we are in a position to use knowledge about the mesh, discretization technique, the origin of the problem (the PDE, for instance)

$$\begin{array}{rcl} \mathcal{L} & \overset{discretize}{\Longrightarrow} & A \\ (approx) \Downarrow & & \Downarrow (approx) \\ \mathcal{C} & \overset{discretize}{\Longrightarrow} & C \end{array}$$

Examples:
Linear elasticity (Korn's inequality)
Navier-Stokes

Start with LU factorization $A(m,n)$

$$
\begin{aligned}
&for\ k = 1, 2 \cdots m-1 \\
&\quad d = 1/a_{kk}^{(k)} \\
&\quad for\ i = k+1, \cdots m \\
&\qquad \ell_{ik}^{(k)} = -a_{ik}^{(k)}\, d \\
&\qquad for\ j = k+1, \cdots n \\
&\qquad\quad a_{ij}^{(k+1)} = a_{ij}^{(k)} + \ell_{ik} a_{kj}^{(k)} \\
&\qquad end \\
&\quad end \\
&end
\end{aligned}
$$

The operational count for the LU factorization can be obtained by integrating the loops:

$$Flops_{LU} = \int_1^{m-1} \int_k^m \int_k^n d_j d_i d_k \approx n^3/3\ (m = n)$$

Block LU factorization $A(m,n)$

$$
\begin{aligned}
&for\ k = 1, 2 \cdots m-1 \\
&\quad D = (A_{kk}^{(k)})^{-1} \\
&\quad for\ i = k+1, \cdots m \\
&\qquad L_{ik}^{(k)} = -A_{ik}^{(k)} D \\
&\qquad for\ j = k+1, \cdots n \\
&\qquad\quad A_{ij}^{(k+1)} = A_{ij}^{(k)} + L_{ik} A_{kj}^{(k)} \\
&\qquad end \\
&\quad end \\
&end
\end{aligned}
$$

The block version offers possibility to use BLAS3.

Incomplete LU factorization $A(n,n)$

$$
\begin{aligned}
&for\ k = 1, 2 \cdots m-1 \\
&\quad d = 1/a_{kk}^{(k)} \\
&\quad for\ i = k+1, \cdots m \\
&\quad\quad \ell_{ik}^{(k)} = -a_{ik}^{(k)}\, d \\
&\quad\quad for\ j = k+1, \cdots n \\
&\quad\quad\quad a_{ij}^{(k+1)} = a_{ij}^{(k)} + \ell_{ik} a_{kj}^{(k)} \\
&\quad\quad\quad \text{when some condition holds true, drop } a_{ij}^{(k+1)} \\
&\quad\quad end \\
&\quad end \\
&end
\end{aligned}
$$

Most often used conditions:

- $a_{ij}^{(k)}$ too small compared to some (relative) value
- $a_{ij}^{(k)}$ does not belong to a chosen sparsity pattern

Loss of information.

Incomplete LU factorization $A(n,n)$

How much ILU can improve the convergence of an iterative process?
Example: 2D, Discrete Laplace operator L(spd).

$\lambda_{min}(A) = h^2$, $\lambda_{max}(A) = O(1)$, $\varkappa(L) = \lambda_{max}(A)/\lambda_{min}(A) = O(h^{-2})$.

The convergence estimate for the CG method is:

$$\|\mathbf{e^k}\|_L \leq 2\left[\frac{\varkappa(L)+1}{\varkappa(L)-1}\right]^k \|\mathbf{e^0}\|_L$$

$$k > \frac{1}{2}\sqrt{\varkappa}\, ln(\frac{2}{\varepsilon})$$

$\Rightarrow k = O(h^{-1})$.
Let $C = LL^T$, where L is some ILU-obtained approximation of the exact Cholesky factor of L. Then

$$\varkappa(C^{-1}L) = O(h^{-1}).$$

Thus, only the constant is improved!

Incomplete Factorization Preconditioners

- pointwise and block ILU
- ILU and IC (when A is spd)
- MILU and MIC (Modified incomplete LU factorization)

  Instead of dropping $a_{ij}^{(k+1)}$, let $a_{k+1,k+1}^{(k+1)} = a_{k+1,k+1}^{(k+1)} + a_{ij}^{(k+1)}$

  Reference: Ivar Gustafsson, A class of first order factorization methods, BIT, 18 (1978), pp. 142-156.

  The information is not fully wasted; preserving positive definiteness.

Incomplete Factorization Preconditioners

- RILU (Relaxed ILU)

  Instead of dropping $a_{ij}^{(k+1)}$, let $a_{k+1,k+1}^{(k+1)} = a_{k+1,k+1}^{(k+1)} + \omega a_{ij}^{(k+1)}$

  Implemented in IFPACK (part of Trilinos, Sandia Nat. Lab.)

  User manual: *For most situations, RelaxValue should be set to zero.*
  - For certain kinds of problems, e.g., reservoir modeling, there is a conservation principle involved such that any operator should obey a zero row-sum property. MILU was designed for these cases and you should set the RelaxValue to 1.
  - For other situations, setting RelaxValue to some nonzero value may improve the stability of factorization, and can be used if the computed ILU factors are poorly conditioned.

ILU preconditioners, cont.

- ILU, based on apriori chosen spartity pattern (ILU(0)):
  the nonzero pattern of L and U coincides with that of the lower/upper part of A
- ILUT: threshhold-based ILU
  Reference: Yousef Saad, A dual threshold incomplete LU factorization, Numerical Linear Algebra with Applications, 1 (1994), 387–402.

ILU preconditioners, cont.

ILU(p)

For all nonzero elements a_{ij} define $u_{ij} = a_{ij}, lev(u_{ij}) = 0$
For $i = 2, \cdots, n$ do
  For $k = 1, \cdots, k-1$ and if $u_{ij} \neq 0$ do
    Compute l_{ik} = u_{ik}/u_{kk}, set $lev(l_{ik}) = lev(u_{ik})$
    Compute $u_{i*} = u_{i*} - l_{ik}u_{k*}$
    Update the levels of u_{i*} as follows
      $level(f_{ij}) = level(l_{ij}) + level(u_{kj}) + 1$
    Replace any element in row i with $lev(u_{ij}) > p$ by zero.
  EndFor
EndFor

ILU preconditioners, cont.

ILUT (Generic ILU with threshhold), Y. Saad, 1994

```
0   row(1:n) = 0
1   do i=2:n
2         row(1:n) = a(i,1:n) % sparse copy
3         for k=1:i-1 and where row(k) is nonzero, do
4               row(k)=row(k)/a(kk)
5               apply a dropping rule to row(k)
6               if row(k)≠0
7                     row(k+1,n)=row(k+1,n)-row(k)*u(k,k+1:n) % sparse update
8               endif
9         enddo
10        apply a dropping rule to row(1:n)
11        l(i,1:i-1)=row(1:i-1) % sparse copy
12        u(i,i:n) = row(i:n) % sparse copy
13        row(1:n) = 0
14        enddo
15  enddo
```

ILU preconditioners, cont.

ILUT: Dropping rules

5: an element is dropped if it is less than the relative tolerance τ_i, equal to $\tau \|row(k)\|$ (using the original row)

10: Drop all entries less than τ_i, keep the largest p entries in the L- and U-part and the diagonal element, which is always kept.

Observe, that sort operations are included.

ILU preconditioners, cont.

BILUT Block-ILUT

H. Sudan & H. Klie (ConocoPhillips); R. Li & Y. Saad (University of Minnesota):
A flexible variant of GMRES (FGMRES) is implemented using the CUDA programming model on the GPU platform using the Single Instruction Multiple Threads (SIMT) paradigm by taking advantage of thousands of threads simultaneously executing instructions.
The implementation on the GPU is optimized to reduce memory overhead per floating point operations, given the sparsity of the linear system. FGMRES relies on a suite of different preconditioners such as BILU, BILUT and multicoloring SSOR.

ILU preconditioners, cont.

BILUT Block-ILUT

General design of the solver: CUDA, SIMT (Single Instruction Multiple threads)

1. CRS sparse format
2. Reorder using RCM (Recursive Cuthill-McKee)
3. Use Metis to generate a balanced partitioning; reorder and repartition after long time intervals
4. FGMRES with some preconditioner per timestep.

Slides from the paper follow.

A044

High Performance Manycore Solvers for Reservoir Simulation

H. Sudan* (ConocoPhillips), H. Klie (ConocoPhillips), R. Li (University of Minnesota) & Y. Saad (University of Minnesota)

SUMMARY

The forthcoming generation of many-core architectures compels a paradigm shift in algorithmic design to effectively unlock its full potential for maximum performance. In this paper, we discuss a novel approach for solving large sparse linear systems arising in realistic black oil and compositional flow simulations. A flexible variant of GMRES (FGMRES) is implemented using the CUDA programming model on the GPU platform using the Single Instruction Multiple Threads (SIMT) paradigm by taking advantage of thousands of threads simultaneously executing instructions. The implementation on the GPU is optimized to reduce memory overhead per floating point operations, given the sparsity of the linear system. FGMRES relies on a suite of different preconditioners such as BILU, BILUT and multicoloring SSOR. Additionally, the solver strategy relies on reordering/partitioning strategies algorithms to exploit further performance. Computational experiments on a wide range of realistic reservoir cases show a competitive edge when compared to conventional CPU implementations. The encouraging results demonstrate the potential that many-core solvers have to offer in improving the performance of near future reservoir simulations.

4. Numerical Experiments

We consider 4 different simulation cases in our performance study. These cases include an oil-water case given by the SPE10 synthetic case (Christie and Blunt, 2001) and 3 field cases: a black-oil case (Case A) and 2 compositional cases (Cases B and C) involving 5 and 8 components, respectively. All cases are run using the IMPES option with implicit well treatment, so the benchmarks are based on the solution of the pressure system. Succinct description of these cases is summarized in Table 1. The four cases yield linear systems that vary significantly in size, sparsity and conditioning. As the number of mass balance equations increases, we observe that the fraction of the solver responsible for the total CPU time decreases. In the case of compositional simulation (cases B and C) a fraction of about 20% is spent in the thermodynamics of the phase behaviour calculation.

Table 1. Benchmark cases.

Case	Size	Model type	% Solver	N. wells	N. Timesteps	Most relevant characteristics
SPE10	60x220x85	Oil-water	90%	5	1423 (50 days)	Highly heterogeneous
A	140x230x44	Black-Oil	81%	83	2524 (34 years)	Waterflooding, highly heterogeneous
B	128x155x19	5 comp.	75%	445	19429 (35 years)	Highly faulted, water injection
C	257x228x21	8 comp.	72%	124	1641 (5 years)	Highly compartmentalized

We first focus our attention on comparing the performance of the FGMRES iterative solver using BILU(0), BILUT and MC-SSOR on a single core CPU and on the GPU. The fill-in and drop threshold for BILUT were set 30 and 1.D-4, respectively. The number of SSOR iterations was set to 2. These preconditioners can be considered the main work-horse options in any reservoir simulator solver available today. In all cases to be shown, the FGMRES solver was assumed to converge when a relative residual less or equal to 10^{-4} was achieved.

We hardware specifications are the following:

- Dual Intel Nehalem (Intel Xeon X5570, 2.93 Ghz, 8MB L3 Cache), 16GB RAM
- NVIDIA Tesla C2050 (installed with Intel Intel Nehalem X5570), 448 cores, 3.22 GB RAM

The Intel Nehalem family architecture is considered the fastest multicore CPU available nowadays. All runs are carried out on Linux. Our in-house simulator was compiled using the –O2 option and linked with the GPU Solver already compiled as a form of a library. The resulting object code was run it on the two hardware platforms mentioned above. Since we are mainly concerned with maintaining accuracy in real field simulations, we restrict our analysis to a double precision version of the GPU solver library. Hence, the compiler option for the C interface to CUDA was set to –arch sm_20 in order to enable the double precision option and global memory cache.

Tables 2, 3 and 4 shows the solver performance using the BILU(0), BILUT and MC-SSOR preconditioners on the GPU for isolated systems obtained from the 4 cases considered. The BILU set of preconditioners rely on METIS partitioning. We are omitting the RCM reordering since it did not show a major influence in the CPU timings for the cases considered. We can note that the BILUT becomes more effective than BILU(0) as the problems sizes increases. The MC-SSOR strategy outperforms these two BILU strategies in all cases. Most the gain here is due to the inexpensive construction of the preconditioner for the MC-SSOR. In each case, we can observe that the CPU Spmv operation is practically negligible compared to the application of the preconditioner.

The left side of Figure 2 compares the performance of the GPU-based solver using the BILU(0) preconditioner relative to the CPU-based solver using the ILU preconditioner upon completion of the simulation. The right side of Figure 2 shows the relative simulation performance using these two solvers. Note that the solver time represents a fraction of the whole simulation (see column 4 on Table 1). The GPU BILU(0) solver shows a gain factor of approximately *3x* with respect to Intel Nehalem. It is worth to mention that the GPU BILU solver is weaker than the ILU sequential implementation as the iterative solver generally takes more iterations. Nevertheless, GPU solver is still able to outperform, since thousands of GPU threads are exploited in each of the FGMRES iterations.

Table 2. Comparative performance assessment using the GPU BILU(0) solver.

Cases	Prec	Prec. Setup (s)	Prec. Appl. (s)	Matvec (s)	Remainder (s)	Total (s)	METIS (s)	N. Iterations
SPE10	BILU(0)	1.06	3.54	0.35	0.58	5.53	1.32	168
A	BILU(0)	0.25	0.58	0.06	0.30	1.19	1.03	58
B	BILU(0)	0.08	0.23	0.03	0.09	0.43	0.20	55
C	BILU(0)	0.08	0.08	0.01	0.02	0.19	0.14	22

Table 3. Comparative performance assessment using the GPU BILUT solver.

Cases	Prec	Prec. Setup (s)	Prec. Appl. (s)	Matvec (s)	Remainder (s)	Total (s)	METIS (s)	N. Iterations
SPE10	BILUT	1.06	2.88	0.27	0.56	4.77	1.32	120
A	BILUT	0.82	0.75	0.06	0.33	1.96	1.03	58
B	BILUT	0.28	0.50	0.03	0.10	0.91	0.20	51
C	BILUT	0.27	0.11	0.01	0.03	0.42	0.14	19

Table 4. Comparative performance assessment using the GPU CS-SSOR(2) solver.

Cases	Prec	Prec. Setup (s)	Prec. Appl. (s)	Matvec (s)	Remainder (s)	Total (s)	N. Iterations
SPE10	MC-SSOR	0.02	1.88	0.16	0.77	2.83	91
A	MC-SSOR	0.02	0.81	0.06	0.26	1.15	59
B	MC-SSOR	0.01	0.22	0.03	0.09	0.35	50
C	MC-SSOR	0.01	0.13	0.02	0.05	0.21	34

The results shown at the right side of Figure 2 are a immediate consequence of Amdahl's Law when a fraction of the original code is only parallelized, namely, the performance improvement PI is given by PI=1/(s + p), where s represents the sequential fraction and p represents the parallel fraction of the simulation code. For instance, the solver component of Case C represents 72% of the total time and the GPU solver is about 2.8 times faster than the CPU Intel Nehalem, thus the expected performance improvement in the simulation is approximately given by PI = 1/(.28 + .72/2.8) ~ 1.8x. This can be seen clearly reflected in the right side of Figure 2.

5. Conclusions and Future Work

We have evaluated the current capabilities that many-core GPU on a set of realistic black-oil and compositional reservoir scenarios. Despite that the study focused only on the solver component of the simulator and that there is still room for further improvements in the proposed implementation, the results provide clear indication of the enormous potential that many-core GPU can offer in near future simulation studies.

Block ILU preconditioners

- LDL^T and LDU preconditioners
- Let $A = D_A - L_A - U_A$.
  Symmetric Gauss-Seidel preconditioner:
  $C = (D_A - L_A)D_A^{-1}(D_A - U_A)$
  Note: $C = (D_A - L_A)D_A^{-1}(D_A - U_A) = D_A - L_A - U_A + LD_A^{-1}U = A + LD_A^{-1}U$

$$A = LU = \begin{bmatrix} I_1 & 0 & \cdots & 0 \\ L_{2,1} & I_2 & \cdots & 0 \\ \vdots & \vdots & \cdots & \vdots \\ L_{n,1} & L_{n,2} & \cdots & I_n \end{bmatrix} \begin{bmatrix} D_1 & U_{1,2} & \cdots & U_{1,n} \\ 0 & D_2 & \cdots & U_{2,n} \\ \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & \cdots & D_n \end{bmatrix}$$

$$\mathbf{y}_i = \mathbf{y}_i - \sum_{j=1}^{i-1} L_{ji}\mathbf{y}_j \qquad \mathbf{x}_i = D_i^{-1}(\mathbf{y}_i - \sum_{j=i+1}^{n} U_{ij}\mathbf{x}_j)$$

Block ILU preconditioners, cont.

Attractive: D - diagonal.
How do we obtain a permutation, such that we get D - diagonal?
Use multicoloring algorithm:
For $1 = 1 : n$
    Set color(i)=0
endfor
For $1 = 1 : n$
    Set color(i)=min($k > 0 : k \neq$color(j) for $j \in Adj(i)$
endfor

where $Adj(i) = \{j \neq i : a_{ij} \neq 0\}$.

Power(q)-pattern Method

Building of power(q)-pattern – ILU(p,q)

- Perform multi-coloring analysis for $|A|^q$ and obtain
  - corresponding permutation π
  - the number of colors B and local block sizes b_i
- Permute $A_\pi := \pi A \pi^{-1}$
- Apply a modified ILU(p) factorization to A_π

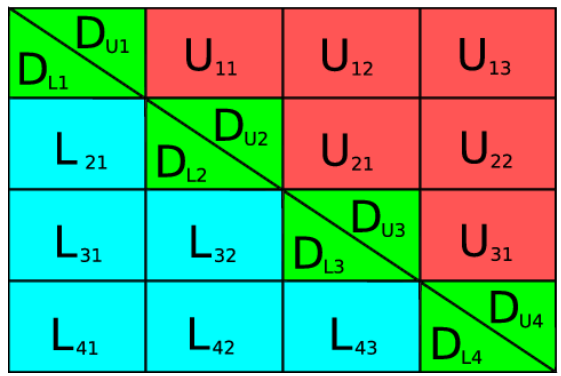
Central result: For $q = p + 1$ we obtain only diagonal elements in the diagonal blocks of L and U. No fill-ins here!
p - fill-ins levels
q - describes the degree of parallelism

ILU(p,q)

- LU sweeps, solve in parallel $LUz = r$
- Re-formulate into a block-form
- Use fine-grained parallelism on the block level
- Parallelism $= N/$Num blocks

$$x_i := D_{Li}^{-1}(r_i - \sum_{j=1}^{i-1} L_{i,j}x_j)$$

$$z_i := D_{Ui}^{-1}(x_i - \sum_{j=1}^{B-i} U_{i,j}z_{i+j})$$



ILU(*p*,*q*) constructs D_{Li}^{-1} and D_{Ui}^{-1} to be with diagonal elements only

Block-tridiagonal matrices

Let A be block-tridiagonal, and expressed as $\boxed{A = D_A + L_A + U_A}$.

One can envisage three major versions of the factorization algorithm:

(i) $\boxed{A = (D + L_A)D^{-1}(D + U_A)}$

(ii) $\boxed{A = (D^{-1} + L_A)D(D^{-1} + U_A)}$

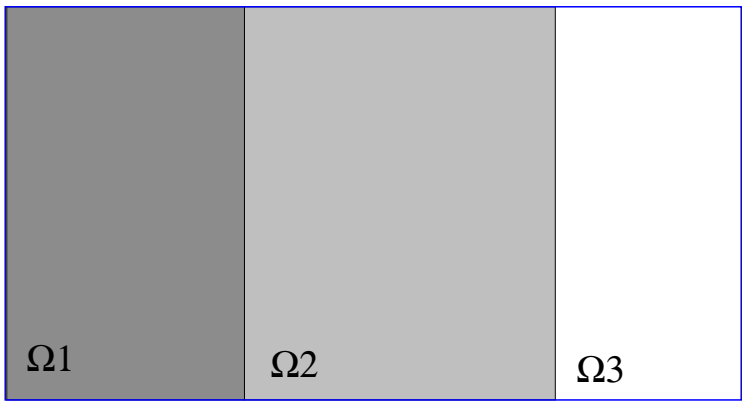
(iii) $\boxed{A = (I - \widetilde{L}_A)D^{-1}(I - \widetilde{U}_A)}$

$D_i = A_{ii} - A_{i,i-1}D_{i-1}^{-1}A_{i-1,i}, i \geq 2, D_1 = A_{11}$

$D_i = (A_{ii} - A_{i,i-1}D_{i-1}A_{i-1,i})^{-1}$, $i \geq 1, D_0 = 0$ (Inverse free substitutions),
where $\widetilde{L}_A = L_A D$, $\widetilde{U}_A = DU_A$.

Here $A^{-1} = (I - \widetilde{U}_A)^{-1}D(I - \widetilde{L}_A)^{-1}$
$(I - \widetilde{U}_A)^{-1} = (I + \widetilde{U}_A^{2^s}) \ldots (I + \widetilde{U}_A^2)(I + \widetilde{U}_A)$ and similarly for $(I - \widetilde{L}_A)^{-1}$.

Block-tridiagonal matrices, cont.



Consider a two-dimensional domain partitioned in strips. Assume that points on the lines of intersection are only coupled to their nearest neighbors in the underlying mesh (and we do not have periodic boundary conditions).
Hence, there is no coupling between subdomains except through the “glue” on the interfaces.

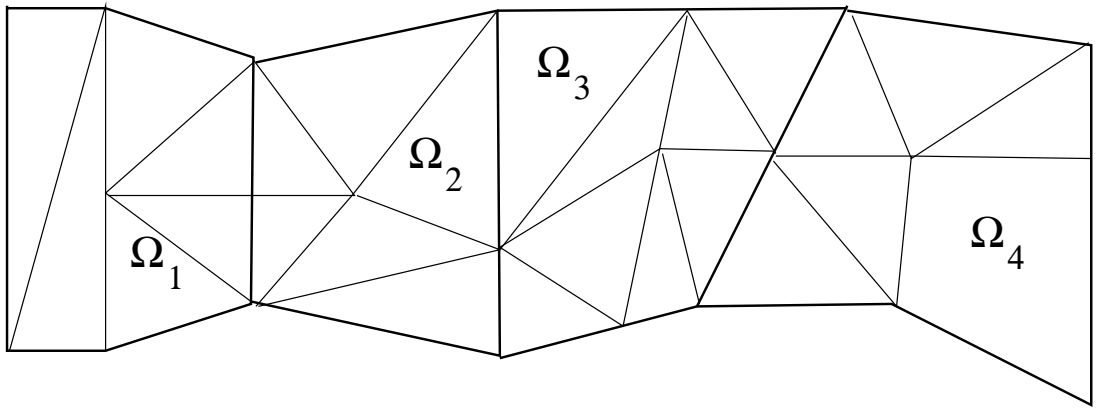
Block-tridiagonal matrices, cont.

When the subdomains are ordered lexicographically from left to right, a domain Ω_i becomes coupled only to its pre- and postdecessors Ω_{i-1} and Ω_{i+1}, respectively and the corresponding matrix takes the form of a block tridiagonal matrix $\boxed{A = \text{tridiag}\,(A_{i,i-1}, A_{i,i}, A_{i,i+1}),}$ or

$$A = \begin{bmatrix} A_{11} & A_{12} & & 0 \\ A_{21} & A_{22} & A_{23} & \\ & \ddots & \ddots & \ddots \\ 0 & & A_{n,n-1} & A_{n,n} \end{bmatrix}$$

For definiteness we let the boundary meshline $\overline{\Omega}_i \cap \overline{\Omega}_{i+1}$ belong to Ω_i.
In order to preserve the sparsity pattern we shall factor A without use of permutations.
Naturally, the lines of intersection do not have to be straight.

Block-tridiagonal matrices, cont.



Examples of subdomain decompositions

Block-tridiagonal matrices, cont.

Slides from Y. Saad, MacLahlan follow.

MODIFICATION AND COMPENSATION STRATEGIES FOR THRESHOLD-BASED INCOMPLETE FACTORIZATIONS*

S. MACLACHLAN†, D. OSEI-KUFFUOR‡, AND YOUSEF SAAD‡

Abstract. Standard (single-level) incomplete factorization preconditioners are known to successfully accelerate Krylov subspace iterations for many linear systems. The classical Modified Incomplete LU (MILU) factorization approach improves the acceleration given by (standard) ILU approaches, by modifying the non-unit diagonal in the factorization to match the action of the system matrix on a given vector, typically the constant vector. Here, we examine the role of similar modifications within the dual-threshold ILUT algorithm. We introduce column and row variants of the modified ILUT algorithm and discuss optimal ways of modifying the columns or rows of the computed factors to improve their accuracy and stability. Modifications are considered for both the diagonal and off-diagonal entries of the factors, based on one or many vectors, chosen a priori or through an Arnoldi iteration. Numerical results are presented to support our findings.

Key words. Incomplete factorization preconditioners, algebraic preconditioners, ILUT, modified ILU

1. Introduction. As physical models become ever more complex, they often result in the need to solve linear systems that are not only much larger than in the past, but also intrinsically more difficult. Due to their larger sizes, these systems cannot practically be solved by direct methods, and this increases the demand for reliable forms of iterative methods that can be substituted for direct solvers. Iterative techniques based on a combination of a preconditioner and a Krylov subspace accelerator are the most common alternatives to direct methods, as they offer a good compromise between cost and robustness. Much of the recent research effort on solving sparse linear systems by iterative techniques has been devoted to the development of effective preconditioners that scale well, while offering good reliability.

In this regard, multilevel methods that rely on incomplete LU (ILU) factorizations have been advocated by many authors in recent years [1, 5–7, 20, 22, 24–26, 34, 36]. While multigrid techniques [12, 38] and their algebraic counterparts (AMG) [31, 41] are known to be optimally efficient for solving some classes of discretized partial differential equations on regular meshes, they may become ineffective when faced with more general types of sparse linear systems. However, the 'multilevel' or 'multistage' ingredient of multigrid can be easily married with general-purpose qualities of ILU preconditioners to yield efficient, yet more general-purpose, solvers.

This paper does not aim at exploring new methods within the multilevel ILU class of techniques. It focuses instead on improving the basic component of ILU-based preconditioners, namely the ILU factorization itself. Among the various options of ILU considered in the literature is the Modified ILU factorization (MILU) proposed by Gustafsson [19] for the symmetric case (Modified Incomplete Cholesky or MIC). Note that for 5-point matrices, MIC(0), where the nonzero pattern of the resulting factors is restricted to match that of the original matrix, is equivalent to the method proposed in 1968 by Dupont, Kendall, and Rachford [16]. The modification in the MIC(0)

*Work supported by the NSF under grants ACI-0305120 and DMS-0811022, the DOE under grant DE-FG-08ER25841, and by the Minnesota Supercomputing Institute and the Institute for Mathematics and its Applications

†Department of Mathematics, Tufts University, 503 Boston Avenue, Medford, MA 02155. email: scott.maclachlan@tufts.edu

‡Department of Computer Science and Engineering, University of Minnesota, 200 Union Street S.E., Minneapolis, MN 55455. email: {dosei,saad}@cs.umn.edu

Grid	n	nnz	k	c_F	t_{setup}	t_{solve}	# iters.
65×65	3969	19593	3	2.54	0.004	0.03	18
129×129	16129	80137	3	2.57	0.02	0.18	31
257×257	65025	324105	3	2.59	0.07	1.28	46
513×513	261121	1303561	3	2.59	0.29	12.49	78
65×65	3969	19593	4	3.30	0.006	0.03	15
129×129	16129	80137	4	3.35	0.03	0.17	25
257×257	65025	324105	4	3.37	0.11	1.05	37
513×513	261121	1303561	4	3.39	0.43	8.98	61

Table 4.6: Performance of ILU(k) on 2D finite-difference Laplacian with levels of fill, $k = 3$, and $k = 4$.

Grid	n	nnz	τ	c_F	t_{setup}	t_{solve}	# iters.
65×65	3969	19593	0.010	3.15	0.01	0.03	16
129×129	16129	80137	0.010	3.35	0.06	0.17	26
257×257	65025	324105	0.010	3.47	0.22	1.25	43
513×513	261121	1303561	0.010	3.54	0.92	12.39	75
65×65	3969	19593	0.016	2.78	0.01	0.03	14
129×129	16129	80137	0.016	2.89	0.05	0.13	19
257×257	65025	324105	0.017	2.92	0.20	0.72	26
513×513	261121	1303561	0.017	2.96	0.78	4.62	38

Table 4.7: Performance of mILUT based on the constant vector on 2D finite-difference Laplacian.

match the preconditioner complexities of Table 4.5. Thus, in Table 4.6, we give results for levels of fill (k) of three and four, with slightly smaller and slightly larger overall complexities. Here, as expected, performance is slightly worse than that of ILUT when the preconditioner complexity is smaller, and somewhat better when the preconditioner complexity is larger. If memory requirements do not pose a constraint, we see that ILU(4) outperforms both ILUT and ILU(3) in terms of iteration counts and solution time on all grids.

We begin testing the modified ILUT algorithm in the setting of classical modified ILU; i.e., with the diagonal entries of the U factor modified so that $LU\mathbf{1} = A\mathbf{1}$, where $\mathbf{1}$ is the vector of all ones. Table 4.7 shows the results of these tests, first for the same drop tolerance, $\tau = 0.01$, as was used for ILUT in Table 4.5 and, then, with the drop tolerance adjusted so that the preconditioner complexities, c_F, nearly match those of ILUT.

The results in Table 4.7 are somewhat surprising. As expected, we see some improvement in the performance of MILUT over that of ILUT for the same, fixed, drop tolerance. In part, this is expected because of the well-known theoretical analysis of modified incomplete factorizations, but it is also to be expected because the preconditioner complexities are somewhat larger than those for ILUT. What is surprising is that when the drop tolerance is raised (so that fewer nonzero entries are kept in the preconditioner), the performance of the modified preconditioners uniformly improve. While unexpected, this is not impossible, as the modification of the diagonal entries in early rows of the matrix (which, of course, depends on the drop tolerance) has a sign-

Examples of ILU performance (numerical)

	Precond.with $\widehat{A}_0$		ILU	
dt	$N1/N2$	$time(s)$	$N1/N2$	$time(s)$
h	$4/10$	14.58		
$h/4$	$4/10$	16.98	*no convergence*	
$h/5$	$4/10$	16.77		
$h/10$	$4/10$	14.67	$4/42$	23.66
$h/20$	$4/10$	14.62	$4/13$	13.23
$h/40$	$4/10$	14.11	$4/10$	10.55

Multiphace flow with convection: Two preconditioners: Run time and number of iterations, $Pe = 1000$

Examples of ILU performance, cont.

	Precond.with $\widehat{A}_0$		ILU	
dt	$N1/N2$	$time(s)$	$N1/N2$	$time(s)$
h	$3/10$	16.66		
$h/4$	$3/10$	16.54		
$h/5$	$3/10$	16.53	*no convergence*	
$h/10$	$3/10$	16.28		
$h/20$	$3/10$	15.82		
$h/40$	$3/10$	15.59		

Problem 2: Two preconditioners: run time and number of iterations, $Pe = 1$.

ILU - pros and cons

PROS:

- Very general, no additional knowledge of the problem is required
- Relatively easy to implement
- Available in the software packages

CONS:

- Method parameters to tune; nonlinear behaviour wrt to τ
- Not very nummerically efficient (may not converge)
- Problem-dependent behavious
- Relatively less degrees of parallelism

Complexity of linear solvers

Time to solve the Poisson model problem on a regular grid with N points

Solver	1D	2D	3D
Sparse Choleski	$O(N)$	$O(N^{1.5})$	$O(N^2)$
Unprecond. CG	$O(N^2)$	$O(N^{1.5})$	$O(N^{1.33})$
IC-precond. CG	$O(N^{1.5})$	$O(N^{1.25})$	$O(N^{1.17})$
Multigrid	$O(N)$	$O(N)$	$O(N)$