Accelerating the Iterative Linear Solver for Reservoir Simulation on Multicore Architectures

†Wei Wu, *Xiang Li, †Lei He, *Dongxiao Zhang

†Electrical Engineering Dept.
University of California, Los Angeles
Los Angeles, United States
weiw@seas.ucla.edu, lhe@ee.ucla.edu

*Dept. of Energy and Resources Engineering
Peking University
Beijing, China
{lixiangcn, dxz}@pku.edu.cn

***Abstract*—Modern petroleum reservoir simulation serves as a primary tool for quantitatively managing reservoir production and planning new fields. It involves repeatedly solving the Jacobian of a set of strong nonlinear partial differential equations governing the mass and energy conduction and conservation. Most of the existing reservoir simulators adopt iterative solver with multiple stages of preconditioners, in which the incomplete LU (ILU) factorization is an outstanding universal smoother. However, it turns out that when the degree of freedom of each grid grows, ILU usually becomes the bottleneck of the solver. Moreover, ILU is difficult to parallelize due to its inherent data dependency. In this paper, we developed a sparse iterative solver with parallelized ILU and triangular solve using block-wise data structure. Compared with the state of art iterative solver on 14 industrial reservoir simulation matrices, the proposed ILU is 5.2x faster (on average) than the state of art iterative solver because of the block-wise data structure, which leads to 2.2x speedup on the total solver runtime. In addition, parallel ILU and triangular solve are developed to further accelerate the solver. To tackle the strong data dependency in ILU and triangular solve, we first partition the algorithm into separated tasks and construct a data flow graph to represent the data dependency. Then, tasks are scheduled in parallel according to the topological order of the data flow graph. On an 8-thread multicore architecture, we achieved another 3.6x speedup on ILU factorization, and 3.3x on triangular solve with good scalability.**

***Keywords*-Iterative solver; sparse matrix; incomplete LU factorization; Multicore architecture**

I. Introduction

In the past two decades, Reservoir simulation has evolved into a mature technology due to the advancement in the reservoir modeling theory and computer science. Nowadays, reservoir simulation serves as a primary tool for researchers and engineers to improve the understanding about reservoirs, design development plans and optimize the recovery processes. The reservoir simulation uses a set of partial differential equations to govern material and energy conduction and conservations. Under certain boundary and initial conditions, these governing equations are discretized in both time and space and then solved numerically. Due to the complicated physical/chemical phenomena and the increasing demand of high resolution in today's petroleum industry, the discrete system is usually large, sparse, and highly non-linear. In most simulators, the system is solved by Newton's method [1]. The non-zero elements of the Jacobian matrix can be packed as blocks. Each block corresponds to a grid-block and the block size is equal to the degree of freedom (DOF) of the grid-block. Consequently, the Jacobian matrix is stored sparsely and block-wise in reservoir simulators.

It is well known that the reservoir equations contain hyperbolic and nearly elliptic sub-systems [2]. The solution of the coupled system is mainly constrained by the solution of their elliptic (typically pressure) components. A two-stage constrained pressure residual (CPR) preconditioning method is the most popular approach to tackle such coupled systems [3], [4]. The first stage, which consists of extracting and solving the pressure subsystem, is usually considered the most expensive part [5]. Yet there are extensive researches into parallelizing the pressure solver [6], [7], [8]. The second stage consists of smoothing the remaining error of the whole solution, and is typically carried out by some general and computational efficient preconditioners such as incomplete LU factorization (ILU). However, when the DOF of grid blocks grows, like what happened in compositional models where the DOF is larger or equal to the number of components (usually 4 to 10), the time cost on the second stage becomes dominant, because the whole system is much larger than the pressure sub system.

In the CPR-type solution for reservoir models with millions of grid, the sparse ILU factorization, triangular solve together with the sparse matrix vector production (SpMV) take a considerable amount of time. K. Stuben etc. reported that the above three parts take 30% to 70% of the solver time in black oil models with grid of magnitude of millions [9]. Generally, in the sequential program, these three parts take 30% to 40% of the solver time and the solver takes more than 70% of the overall simulation time in black oil models. Y. Zhou etc. reported that in a commotional model with 4 components, the sparse ILU factorization, triangular solve and the SpMV parts take 49% of the solver time, and 33% of the total simulation time [10]. The runtime of these three parts will account for an increasing proportion as the number

of components grows. Moreover, only SpMV is known to be suitable for parallelization, while both sparse ILU and triangular solve are difficult to be parallelized because of the inherent, strong data dependency in the algorithm [11].

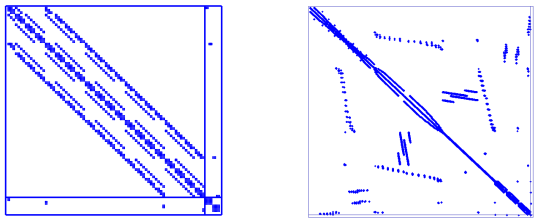
In this paper, we develop a block-wise sparse iterative solver, and parallelize the sparse ILU and triangular solve on multicore architecture without altering the results of the original sequential algorithm. By taking advantage of the block-wise data structure in reservoir simulation, we achieved 5.2x speedup on average on ILU and 2.2x speedup on the total runtime over ITSOL [12], the state of art iterative solver implemented without block-wise date structure. To tackle the data dependency that exists in ILU or triangular solve, we first consider it as multiple tasks and represent the data dependency between these tasks as a data flow graph. Next tasks in the data flow graph are partitioned into different groups according to their earliest start time (also following the topology of the data flow graph) and scheduled on multicore architecture in parallel. Experiments are performed on 14 industrial reservoir simulation matrices with an 8-thread multicore machine. The parallel algorithm achieved 2.3x-4.6x (3.6x on geometrical average) on ILU, and 2.4x-3.8x speedup (3.3x on geometrical average) on triangular solve. Furthermore, it also exhibits good scalability when the number of threads and the problem size increase.

The rest of this paper is organized as follows. The background, including the characteristics of reservoir matrix and the iterative linear solver algorithm, are reviewed in Section II. The parallelization of sparse ILU and triangular solve are explained in detail in Section III. Experiment results on 14 industrial examples are discussed in Section IV. This paper is concluded in Section V.

II. Background

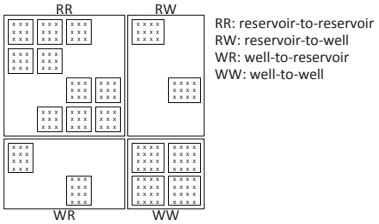
A. Characteristics of reservoir matrices

The reservoir simulation equations couple multiple sets of variable fields (e.g., the pressure field, the saturation fields, and the solubility fields). Consequently its Jacobian matrices naturally have block structure. The block size equals the DOF of the grid: in black oil models, the block size is 3; while in compositional models the block size is larger than or equal to Nc, which is the number of components. The flow in reservoir simulation only happens between neighbored grids, causing the Jacobian matrix to be quite sparse with respect to the number of blocks. It usually has only 5 to 20 blocks per row of blocks, depending on the flux stencil we choose. Complex well models [13] are often used to simulate well facilities. The well bores are discrete and the number of unknowns in each well node is not equal to the number of unknowns in each reservoir grid. Therefore the block sizes of the reservoir-to-reservoir part, reservoir-to-well part, well-to-reservoir part, and well-to-well part of the Jacobian matrix are all different.



(a) Matrix generated from a structured reservoir grid with 2 wells

(b) Matrix generated from an unstructured reservoir gird with 3 wells



(c) A sketch map of the block structure in reservoir simulation matrices

Figure 1. Example of reservoir matrices

Two examples of the reservoir matrices are illustrated in Figure 1(a) and Figure 1(b). Both these two matrices consist of 4 submatrices, which are illustrated schematically in Figure 1(c). The granularity of the reservoir matrices are dense data blocks with known size, as shown in Figure 1(c), allowing algorithms to take advantage of the block-wise data structure. On the other hand, the reservoir matrices usually have small condition numbers, unlike the circuit matrices which can be severely ill-conditioned because the matrix entries scale in a very large range [14]. It is more reliable to solve the circuit system using direct method [14], [15], [16], [17], [18]. However, due to the good numerical stability of the reservoir matrices, they are usually solved by iterative methods for better efficiency.

B. Overview of the sparse iterative solver

A typical iterative solver consists of two steps, the preconditioner and the iterative update. For accelerators, the number of iterations generally increases with the increase of condition number [11]. The goal of the preconditioning is to reduce the condition number using a preconditioner $\mathbf{P}$. Taking the Richardson iteration [11] for solving $\mathbf{A}x - b = 0$ as an example, without preconditioning each standard iteration calculates

$$x_{i+1} = x_i - \omega(\mathbf{A}x_i - b) \tag{1}$$

where ω is a scale parameter. To achieve a faster convergence, a mathematical equivalent system $\mathbf{P}^{-1}(\mathbf{A}x - b) = 0$ is solved instead. Then each iteration is solving

$$x_{i+1} = x_i - \omega \mathbf{P}^{-1}(\mathbf{A}x_i - b) \quad (2)$$

In the iterative method for a reservoir matrix, the incomplete LU factorization (ILU) is used to calculate the preconditioning matrix $\mathbf{P}$. The ILU preconditioner factors an original coefficient matrix $\mathbf{A}$ into a sparse lower triangular matrix $\mathbf{L}$, a sparse upper triangular matrix $\mathbf{U}$ and a residue matrix $\mathbf{R}$ that satisfies certain constrains.

$$\mathbf{A} = \mathbf{LU} - \mathbf{R} \quad (3)$$

where $\mathbf{P} = \mathbf{A} + \mathbf{R} = \mathbf{LU}$. After that, each iteration of the iterative update involves two major computations, the SpMV and the triangular solve. The triangular solve is applied because $\mathbf{P}^{-1}$ is not usually explicitly calculated due to its high complexity. In each iteration, the triangular solve is utilized to solve $\gamma = \mathbf{P}^{-1}(\mathbf{A}x_i - b)$ in (2).

In detail, the sparse ILU factorization algorithm is derived from sparse Gaussian elimination, which factors matrix $\mathbf{A}$ into two triangular matrix $\mathbf{L}$ and $\mathbf{U}$ satisfying $\mathbf{A} = \mathbf{LU}$. While the complete LU factorization keeps all the fills during Gaussian elimination, the ILU factorization only keeps part of them and drops certain predetermined off-diagonal fills. Due to the dropped fills, the complexity of calculating $\mathbf{L}$ and $\mathbf{U}$ is much smaller than a full LU factorization. On the other hand, it sacrifices the accuracy, which is reflected in residual matrix $\mathbf{R}$ in (3). There are different implementations of ILU based on how the symbolic structure of $\mathbf{L}$ and $\mathbf{U}$ is determined.

During the iterative update, the SpMV and the triangular solve are preformed iteratively and to approach the accurate solution. The triangular solve calculates x from $\mathbf{LU}x = b$ by solving two triangular systems $\mathbf{L}y = b$ and $\mathbf{U}x = y$. The parallelization of SpMV can be performed straightforwardly by mapping the computation of vector-vector production to multiple processors, then reducing them together as the result of SpMV. This has been extensively studied in existing literature [19], [20], [21]. On the other hand, the speedup of the triangular solve is not as obvious as SpMV because of the inherent data dependency. Most of the existing approaches implement the parallel triangular solve using level scheduling [22], [23]. However, the implementation of level scheduling is non-trivial because the computation in triangular solve is lighter weighted comparing with the thread communication for scheduling. An inefficient implementation of triangular solve will not speedup the computation, but slow it down by overwhelming scheduling overhead.

To figure out the hot spots of the iterative linear solver, we profiled the runtime of solving 14 reservoir matrices and illustrated them in Figure 2. The total runtime of solving these 14 matrices are normalized to 1. In Figure 2, ILU factorization takes a large portion of the total runtime, even though it is executed only once. This is due to the high complexity of the ILU algorithm itself. Apart from ILU, the triangular solve contributes the majority of the remaining runtime. Consequently, parallelizing the ILU factorization and triangular solve is of utmost importance.

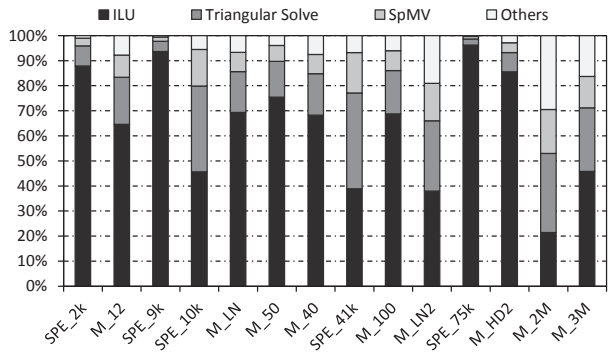


Figure 2. Runtime profiling on 9 reservoir matrices

III. Parallel ILU(p) and Triangular solve

A. Symbolic analysis for preconditioner: ILU(p)

ILU is derived from sparse Gaussian elimination, and it processes matrix $\mathbf{A}$ in a very similar way to the LU factorization. The row-wise ILU factorization is presented in Algorithm 1, while its column-wise counterpart is very similar. In this paper, the following discussions are all based on the row-wise representation.

Algorithm 1 Sparse incomplete LU factorization

```
1:  for i = 0 : N − 1 do
2:      for k = 0, ..., i − 1 and for (i, k) ∈ NZ(L) do
3:          a_ik = a_ik / u_kk;
4:          for j = k + 1, ..., n and for (k, j) ∈ NZ(LU) do
5:              Compute a_ij = a_ij − a_ik * u_kj;
6:          end for
7:      end for
8:      Assign u_ij = a_ij for (i, j) ∈ NZ(U);
9:      Assign l_ij = a_ij for (i, j) ∈ NZ(L);
10: end for
```

Algorithm 1 involves a symbolic analysis step and a numeric factorization step. Different implementations of ILU depend on how the symbolic structure of $\mathbf{L}$ and $\mathbf{U}$ is determined. The following discussions are based on the ILU(p), which is adopted in our parallel iterative solver.

ILU(p) determines the symbolic structure of $\mathbf{L}$ and $\mathbf{U}$ based on the level of fill (LOF), p, of each element. LOF is attributed to all the nonzero elements that is processed by Gaussian elimination. All nonzero elements a_{ij} originally existing in NZ($\mathbf{A}$) are considered as LOF $= 0$, while a_{ij}s that do not exist in NZ($\mathbf{A}$) are assigned LOF $= \infty$ initially. As shown in line 5 of algorithm 1, the matrix entry a_{ij} is updated according to $a_{ij} = a_{ij} - a_{ik} * u_{kj}$. In the meantime, the LOF of a_{ij} is also updated accordingly as follows:

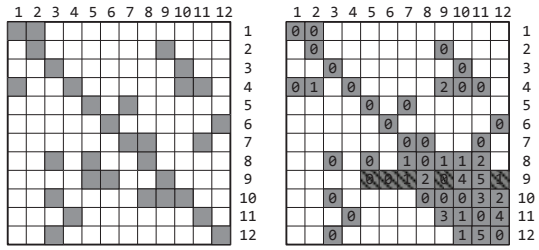
$$\text{LOF}_{ij} = min\{\text{LOF}_{ij}, \text{LOF}_{ik} + \text{LOF}_{kj} + 1\} \quad (4)$$

ILU(p) will only keep the fills whose LOF is less than or equal to "p". In particular, the symbolic structure of $\mathbf{L}$ and $\mathbf{U}$ in ILU(0) will be identical to $\mathbf{A}$. During the ILU factorization, the matrix $\mathbf{A}$ is processed and the entry values in $\mathbf{L}$ and $\mathbf{U}$ are calculated row by row, which correspond to line 2-9 in Algorithm 1. This row-wise update is the most expensive step and includes a symbolic and numeric factorization step. Let $b = \mathbf{A}(k,:)$ and G_U be the directed acyclic graph of $\mathbf{U}$ with n nodes. An edge $i \rightarrow j$ exists in G_U when there is a nonzero entry u_{ij} in NZ($\mathbf{U}$). Apparently, all edges are pointing from a smaller node i to a larger node j because $\mathbf{U}$ is an upper triangular matrix.

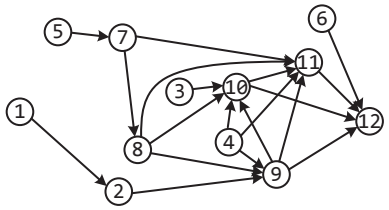
Define $\mathbf{B} = \{i | b_i \neq 0\}$ and $\mathbf{X} = \{i | u_{ik} \neq 0 \text{ or } u_{uk} \neq 0\}$ which represent the nonzero indices in b and the k^{th} row in NZ($\mathbf{U}$), respectively. Then the nonzero pattern $\mathbf{X}$ can be determined by:

$$\mathbf{X} = \mathit{Reach}_{G_U}(\mathbf{B}, p) \quad (5)$$

where $\mathit{Reach}_{G_U}(r,p)$ denotes the column index of all nodes in graph G_U with $\text{LOF}_{k,j} \leq p$ reachable via paths starting at node r. $\mathit{Reach}(\mathbf{B},p)$ applied to a set of $\mathbf{B}$ is the union of $\mathit{Reach}(r,p)$ for all nodes $r \in \mathbf{B}$.



(a) Symbolic Structure of A (b) L and U with level of fill



(c) Graph G_U

Figure 3. Computing $\mathbf{X}$ for ILU(1) starting at $\mathbf{B}$ using depth first search. e.g. $\mathbf{B} = \{5, 6\}$ for column 9. Then starting a depth-first search at node 5 get $\mathit{Reach}_{G_U}(5,1) = \{5,7,9,12\}$ where node 8, 10, and 11 are skipped because their LOFs are higher than 1, next $\mathit{Reach}_{G_U}(6,1) = \{6,12\}$, so $\mathbf{X} = \{6,5,7,9,12\}$, which is in topological order, but not in nature order.

The reachable problem in (7) is solved by depth first search considering the constraint of LOF. During the depth first search, the nodes j with $\text{LOF}_{k,j} > p$ are skipped. Since the depth first graph traversal produces the $\mathbf{X}$ according to the topology of G_U inherently, $\mathbf{X}$ is computed in topological order. Once $\mathbf{X}$ is sorted in topological order, all the data dependencies will be satisfied, e.g. x_i must be computed prior to x_j if there is an edge from i to j in G_U. The natural order $1, 2, ..., n$ satisfy the requirement of topological order, however, it takes extra time to sort the nodes in $\mathbf{X}$ into natural order. Similar to Gilbert/Peierls algorithm [24] for LU factorization, the symbolic and numeric factorization in ILU take time proportional to the number of floating-point operations without sorting. An example of column-wise update is illustrated in Figure 3.

B. Parallel numeric factorization

1) Data dependency analysis: Once the symbolic structure of $\mathbf{X}$ in each column is determined, the numeric factorization that calculates the value of entries in $\mathbf{L}$ and $\mathbf{U}$ can be performed according to Algorithm 1. During the reservoir simulation, the simulator solves linear problems with the same symbolic structure for multiple times. Therefore, the symbolic factorization in ILU(p) only needs to be performed once, while the numeric factorization is executed repeatedly in each Newton iteration.

During the numeric factorization, lines 2-9 of Algorithm 1 are executed repeatedly to factorize each row of matrix $\mathbf{A}$ based on the predetermined $NZ(\mathbf{LU})$. If there is a nonzero element a_{ik} in $NZ(\mathbf{L})$, row i will be updated by $\mathbf{U}(\mathbf{k},:)$, which corresponds to lines 3-6. Since $\mathbf{U}(\mathbf{k},:)$ is the factorization result of row k, the factorization of row i has to be scheduled after row k because of the data dependency.

We may consider the factorization of each row as a task. Then task i is dependent on task k if there is a nonzero element l_{ik} in the i^{th} row of matrix $\mathbf{L}$, $\mathbf{L}(\mathbf{i},:)$. In particular, if there is no nonzero element in $\mathbf{L}(\mathbf{i},:)$, task i can start instantly. As shown in Figure 4(b), the data dependency can be considered as a data flow graph [25], [26]. An edge pointing from node k to node i in the data flow graph indicates the data dependency from task i to task k, which is corresponding to a nonzero element l_{ik} in $\mathbf{L}$.

Considering the computation time of each task as the same unit, we can define the earliest start time (EST) of task i as

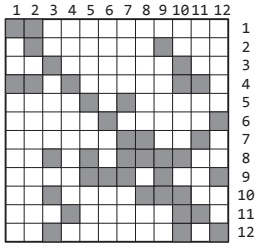
$$\text{EST}(i) = max\{-1, est(j)\} + 1 \quad (6)$$

$\forall j$ that satisfies $(i,j) \in NZ(\mathbf{L})$ and $j < i$.

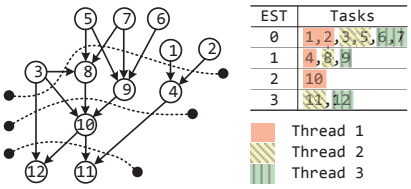
Apparently, if there is no off-diagonal nonzero element in $\mathbf{L}(\mathbf{i},:)$, $\text{EST}(i)$ will be 0. As an example, we calculate the EST for the predetermined $\mathbf{L}$ and $\mathbf{U}$ in Figure 3(b) considering $\text{LOF}(i) \leq 1$. The symbolic structure of ILU(1) and EST are presented in Figure 4.

Moreover, the following lemma exists under the definition of $\text{EST}(i)$.

Lemma 1: Let $\mathbf{S} = \{k_1, k_2, ..., k_m\}$ be a set of tasks whose EST are the same, then any two tasks in $\mathbf{S}$ are independent to each other.



(a) Symbolic structure



(b) Data dependency graph (c) Earliest start time

Figure 4. Calculating EST from the the symbolic structure of $\mathbf{L}$

Lemma 1 can be easily proved according to the definition of EST. Assuming k_j depends on the data calculated from k_i, then

$$\text{EST}(k_j) \geq \text{EST}(k_i) + 1 \tag{7}$$

satisfies. Therefore, $\text{EST}(k_j)$ and $\text{EST}(k_i)$ cannot be the same. On the other hand, if task $k_1, k_2, ..., k_m$ have the same EST, they are independent to each other.

2) Hybrid parallel mode: Based on the EST, we can consider the tasks as $r+1$ clusters, $\mathbf{S_0}$, $\mathbf{S_1}$, ... , $\mathbf{S_r}$, where r is the largest EST for all tasks, and $\mathbf{S_i}, i \in 0, 1, ..., r$ is the set of tasks whose EST is i. In this subsection, we will introduce two modes to process the tasks in parallel:

- ***Cluster*** mode: tasks are processed strictly according to their EST. Tasks in $\mathbf{S_{i+1}}$ can only be started after all the tasks in $\mathbf{S_i}$ are finished.
- ***Pipeline*** mode: Tasks in with different EST can be started simultaneously in multiple threads. These threads communicate with each other to make sure the required data is ready.

Since the *cluster* mode only process the tasks with the same EST at one time, it is not necessary to deal with the data dependency between tasks, which results in very little inter-thread communication overhead. However, some threads might be idle when there are insufficient tasks in one cluster to keep all the thread busy. The *pipeline* mode fix this problem by allowing tasks in different clusters to be processed simultaneously. However, the communication overhead might degrade the performance.

In the ILU application, we apply a ***hybrid*** mode, which combines both *cluster* and *pipeline* modes. In particular, when there are a large number of tasks in one cluster, we apply the *cluster* mode. Otherwise, once the number of tasks drops below a threshold at $\mathbf{S_j}$, the *pipeline* mode will take over the remaining clusters, $\mathbf{S_j}$, $\mathbf{S_{j+1}}$, ... ,$\mathbf{S_r}$.

C. Parallel Triangular Solve

In terms of runtime, the triangular solve is less expensive than full LU factorization. On the other hand, the computation granularity in triangular solver is much finer than LU and ILU factorization.

Algorithm 2 Sparse triangular solve $\mathbf{Ly} = \mathbf{b}$

1: $y = b$
2: **for** $i = 0, ..., n-1$ **do**
3: **for** $j = 0, ..., i-1$ and $(i, j) \in NZ(\mathbf{L})$ **do**
4: Compute $y_i - = l_{ij} * y_j$;
5: **end for**
6: **end for**

The pseudo-code of triangular solve is presented in Algorithm 2. Similar to ILU algorithm, data dependency occurs when y_i is trying to access y_j in line 4. Considering the calculation of y_i in lines 3-5 as task i, task i will depend on the result of task j if $(i, j) \in NZ(\mathbf{L})$. Just as in ILU, the data dependency in triangular solve can be determined from $NZ(\mathbf{L})$, and the same task scheduling techniques can be applied. However, the task in triangular solve consists of only a few arithmetic operations, which lead to a smaller ratio between computation and thread communication/synchronization. Hence, the parallelization of triangular solve was considered not efficient [16].

However, the aforementioned problems do not apply to the iterative linear solver for reservoir simulation. First, the triangular solve is performed repeatedly in each iteration of the iterative update. Therefore, it accounts for a considerable portion of the solver runtime, as shown in Figure 2. Second, the granularity in the reservoir matrix is small dense blocks with known size. Thus the computations in line 4 of Algorithm 2 are no longer arithmetic operations, but matrix-matrix multiplications and subtractions. Thus ratio between computation and thread communication/synchronization is raised in reservoir simulation applications.

Furthermore, when we schedule tasks on multiple threads, we only use the *cluster* mode in the parallel triangular solve. Therefore the communication overhead is reduced to minimal.

IV. Experiment Results

In this section, we present the benchmark results of the parallel ILU and triangular solve on a multicore machine.

A. Experiment settings

The multicore machine we used is equipped with Intel® Core™ i7-3820 operating at 3.60GHz. It issue at most 8 threads simultaneously with hyper-threading.

We implemented the proposed parallel iterative solving using block-wise data structure. The LOF of ILU is set to

Table I
COMPARISON ON THE SINGLE-THREAD RUNTIME ON MATRICES GENERATED FROM INDUSTRIAL RESERVOIR SIMULATOR

Test Cases	# of blcoks		Block size		# of rows	# of iterations	ILU(1) runtime (ms) and speedup			Total runtime (ms) and speedup			PARDISO
	RRP	WWP	RRP	WWP			ITSOL	block ILU[1]	Speedup	ITSOL	block ILU[1]	Speedup	
SPE_2k	2592	0	10	1	25920	12	440.0	87.5	5.0	660.0	158.8	4.2	1648.3
M_12	12344	59	3	4	37268	15	60.0	6.9	8.7	100.0	47.9	2.1	649.1
SPE_9k	9408	0	7	1	65856	4	610.0	128.0	4.8	1030.0	179.9	5.7	11472.0
SPE_10k	10368	0	10	1	103680	119	1820.0	327.9	5.5	2730.0	2945.3	0.9	24664.4
M_LN	43679	59	3	4	131273	12	250.0	27.5	9.1	380.0	146.0	2.6	5584.1
M_50	50000	20	5	3	250060	16	1250.0	327.0	3.8	2460.0	801.9	3.1	44496.8
M_40	100000	80	4	3	400240	12	680.0	416.6	1.6	1000.0	747.0	1.3	304841.8
SPE_41k	41472	0	10	1	414720	160	7780.0	1377.5	5.6	11580.0	15838.2	0.7	495332.9
M_100	100000	20	5	3	500060	22	2730.0	669.3	4.1	5230.0	1989.5	2.6	505445.2
M_LN2	260985	184	2	3	522522	31	N/A	97.0	N/A	N/A	991.8	N/A	35698.2
SPE_75k	75264	0	7	1	526848	2	5240.0	1031.7	5.1	7530.0	1285.4	5.9	1235692.9
M_HD2	368326	10	3	4	1105018	4	2170.0	246.1	8.8	2850.0	624.7	4.6	237800.7
M_2M	1094421	425	2	3	2190117	65	2560.0	493.8	5.2	5700.0	8141.0	0.7	N/A
M_3M	1094421	425	3	4	3284963	33	7790.0	1240.1	6.3	14780.0	9242.9	1.6	N/A

[1]**block ILU** refers to the single thread version of the proposed iterative solver with ILU

1 and the tolerance for stopping iteration is set to 1e-6. To achieve an apple to apple comparison, we setup the latest version of open-source iterative solver, ITSOL [12], with the same configuration. Here the ITSOL is a variable-wise solver without block-wise date structure. We also include the runtime of PARDISO [27] in the experiment as a reference of the direct solver, to confirm the superiority of iterative methods in reservoir simulation. All the programs are compiled with gcc-4.6.3 and the multi-threading parallelism is implemented based on OpenMP.

In the experiment, 14 reservoir matrices dumped from an industrial reservoir simulator were used to evaluate the performance of the proposed parallel iterative solver, especially the ILU and triangular solver. The matrix dimension and block size of these matrices are provided in Table I. These matrices consist of 4 sub-matrices with different dense block sizes as illustrated in Figure 1(c). The matrix dimensions in Table I are defined as the number of blocks. Take the largest matrix M_3M as an example: the top-left submatrix (RRP) has a dimension of 1094421 in terms of 3×3 blocks, which is 3,283,263 rows. Including the fact that there are another 1,700 rows in the bottom-right submatrix, the number of rows in the entire M_3M matrix is as large as 3,284,963 rows.

In the following discussing, we will present the experiments results in three aspects: 1) the runtime comparison of the sequential algorithms; 2) the scalability of the parallel algorithm; 3) the scalability of different parallel mode.

B. Results and discussions

1) Speedup due to block-wise processing: In the proposed iterative solver, all the matrices are stored as four separated submatrices as illustrated in Figure 1(c), while each submatrix is stored in block-wised compressed sparse row (BSR) format with different block sizes.

In Table I, we first compare the ILU runtime in ITSOL and the sequential version of the proposed solver. We can observe a 5.2x speedup on geometric average. We can also notice that runtime of ILU is not only related to the matrix dimension, but also the size of the sense data block. A larger block size means more FLOPs to process it. In particular, the matrix M_40 and SPE_41K has similar matrix dimensions, but it requires distinct runtime to get these to matrices solved.

In terms of total runtime, we can observe promising speedup over ITSOL. However, the speedup on the total runtime is relatively smaller because we have different implementations in the rest of the solver code. In addition, we include the runtime of PARDISO as a reference of the direct solver.Even though PARDISO takes advantage of the dense data structure by forming "Supernode", the runtime is still prohibitively long compared with the iterative solvers.

2) Parallel runtime and scalability: Apart from the speedup due to block-wise data structure, we obtain extra speedup by parallelizing ILU and triangular solve. All the runtime presented in this subsection are tested on programs using block-wise data structure.

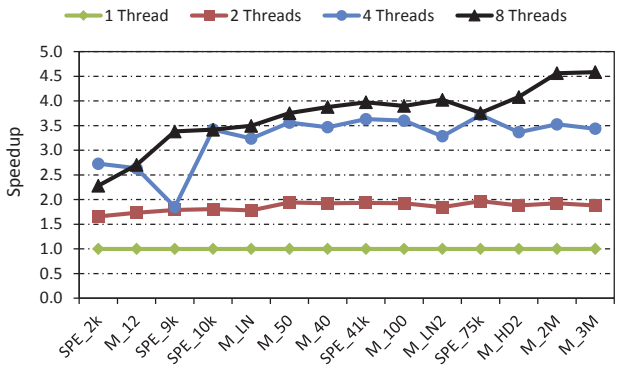


Figure 5. Speedup of ILU over the sequential program

The speedup of the parallel ILU over its sequential counterpart is presented in Figure 5. On these 14 benchmark

matrices, the 8-thread program runs 2.3x-4.6x (3.6x on geometrical average) faster than the sequential counterpart. In Figure 5, it is apparent that higher speedups are achieved on larger test cases. On small matrices, such as SPE_2K, M_12, and M_LN, the speedup is less than 3x because of the high communication/synchronization to computation ratio. In contrast, 4.6x speedup is achieved on the largest two test cases, M_2M and M_3M.

The speedup is also affected by the block size. Matrices M_100 and M_40 are very close in number of blocks, but differ in block sizes. That leads to slightly better speedup on M_100 compared with M_40. Similarly, we notice that M_3M achieves slightly higher speedup compared with M_2M. This is important because there is a push to use more complex models instead of the black oil model to simulate realistic reservoir behaviors. More complex models involve more variables per grid, and the block size (Nb) of the Jacobian matrix entries grows accordingly. If the grid size does not change, the FLOPs expended on ILU factorization is proportional to Nb^3, and the FLOPs expended on triangular solve and SpMV is proportional to Nb^2, while the pressure solution stage does not change. Because our parallel algorithms tend to get higher speedup on matrices with larger block size, it will have better scalability on more realistic problems.

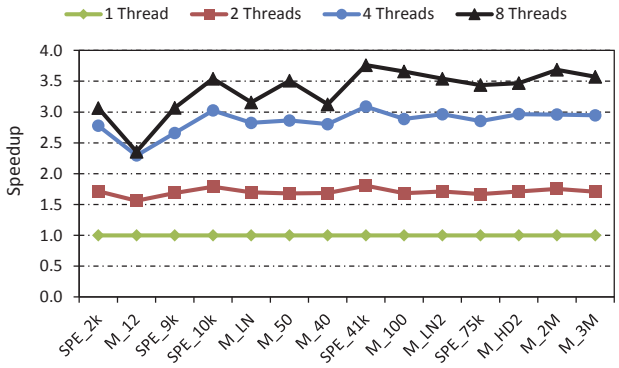
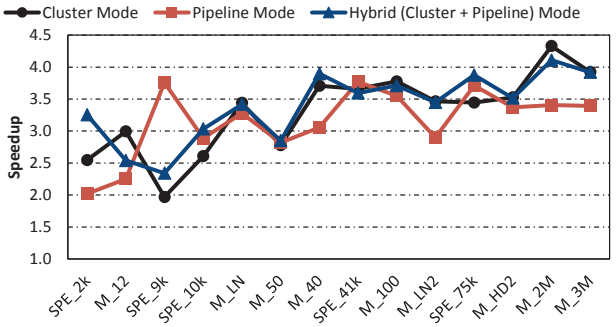


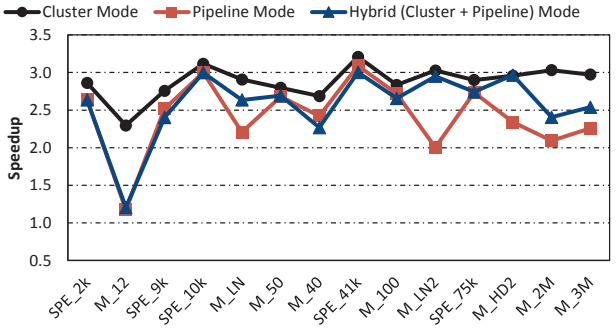
Figure 6. Speedup of triangular solve over the sequential program

A similar trend can be observed in the acceleration of triangular solve, which is illustrated in Figure 6. 2.4x-3.8x speedup (3.3x on geometrical average) is achieved on these 14 benchmark matrices. This is reasonable because the triangular solve shares the same data dependency graph with the ILU. The speedup on triangular solve is slightly smaller than ILU since it is less computational intensive, which leads to a higher communication/synchronization to computation ratio even when we use a pure *cluster* mode.

3) Impacts of different parallel modes: There are multiple modes to schedule the tasks in parallel, such as *cluster* mode, *pipeline* mode, and *hybrid* mode that consists of both *cluster* and *pipeline* mode. To understand the impact of different parallel modes on ILU and triangular solve, the tested the speedup of an 4-threaded program over the sequential counterpart using all these aforementioned three modes. The test results of ILU and triangular solve are illustrated in Figure 7(a) and 7(b) respectively.



(a) ILU(1)



(b) Triangular solve

Figure 7. Performance using pure *cluster* mode, *pipeline* mode, and the *hybrid* mode with multi-thread

In Figure 7(b), it is obvious that the pure *cluster* mode outperforms the *hybrid* mode on all these 14 cases. This is reasonable because the computation load in triangular solve is comparatively low. Hence communication/synchronization overhead in the *hybrid* mode becomes dominating. Therefore, the pure *cluster* mode is adopted to parallelize the triangular solve instead of the *hybrid* mode.

However, on the ILU case, it is not apparent which parallel mode is the winner. However, a closely observation reveals that the *hybrid* mode is the mode option in overall. It is slower than the *cluster* mode on small case, such as M_12. Also, when the computational load of each task is heavy, the tradeoff of leaving the thread idle becomes dominant. That explains why the pipeline mode is the fastest one on SPE_9k, which is consists of 10-by-10 dense blocks.

V. Conclusion

With the advancement of reservoir simulation techniques, the widely used ILU and triangular solve could become the bottleneck of the iterative solver. Moreover, the ILU factorization and the triangular solve are difficult to parallelize due to inherent data dependency. In this paper,

block-wise parallel algorithms are developed for ILU and triangular solve to accelerate the iterative linear solver. By taking advantage of the block-wise data structure in reservoir simulation, we achieved 5.2x speedup on ILU and 2.2x speedup on the total runtime over ITSOL, the state of art iterative solver implemented in variable-wise date structure. Moreover, to unleash the computing power of the parallel hardware, we have to tackle the strong data dependency in ILU and triangular solve. In particular, we first partition the algorithm into separated tasks and calculate the earliest start time (EST) of each task according to the data dependency. Then, three parallel modes, including the *cluster* mode, the *pipeline* mode, and the *hybrid* mode, are discussed to schedule those tasks. The *hybrid* mode and the pure *cluster* mode are applied to ILU and triangular solve, respectively. On 14 industrial benchmarks, we achieved 2.3x-4.6x (3.6x on geometrical average) speedup on ILU, and 2.4x-3.8x speedup (3.3x on geometrical average) on triangular solve using an 8-thread multicore machine. Furthermore, good scalability of the parallel algorithm can be observed when the number of threads increases.

REFERENCES

[1] H. Cao, "Development of techniques for general purpose simulators," Ph.D. dissertation, Stanford University, 2002.

[2] H. Cao, H. Tchelepi, J. Wallis, and H. Yardumian, "Parallel scalable unstructured cpr-type linear solver for reservoir simulation," in *SPE Annual Technical Conference and Exhibition*, 2005.

[3] J. Wallis, "Incomplete gaussian elimination as a preconditioning for generalized conjugate gradient acceleration," in *SPE Reservoir Simulation Symposium*, 1983.

[4] J. Wallis, R. Kendall, and T. Little, "Constrained residual acceleration of conjugate residual methods," in *SPE Reservoir Simulation Symposium*, 1985.

[5] K. Stuben, G. Brown, D. Chen, S. Gries, and D. Collins, "Preconditioning for efficiently applying algebraic multigrid in fully implicit reservoir simulations," in *2013 SPE Reservoir Simulation Symposium*, 2013.

[6] V. E. Henson and U. M. Yang, "BoomerAMG: A parallel algebraic multigrid solver and preconditioner," *Applied Numerical Mathematics*, vol. 41, no. 1, pp. 155–177, 2002.

[7] H. Tchelepi and Y. Zhou, "Multi-gpu parallelization of nested factorization for solving large linear systems," in *2013 SPE Reservoir Simulation Symposium*, 2013.

[8] (2011) XSAMG announcement. Fraunhofer Institute SCAI. [Online]. Available: http://www.scai.fraunhofer.de

[9] K. Stuben, G. Brown, D. Chen, S. Gries, and D. Collins, "Preconditioning for efficiently applying algebraic multigrid in fully implicit reservoir simulations," in *2013 SPE Reservoir Simulation Symposium*, 2013.

[10] Y. Zhou, Y. Jiang, and H. A. Tchelepi, "A scalable multistage linear solver for reservoir models with multisegment wells," *Computational Geosciences*, pp. 1–20, 2013.

[11] Y. Saad, *Iterative Methods for Sparse Linear Systems, Second Edition*. PWS, 2004.

[12] N. Li, B. Suchomel, D. Osei-Kuffuor, and Y. Saad, "ITSOL: Iterative solvers package," *University of Minnesota*, 2008.

[13] J. Holmes, T. Barkve, and O. Lund, "Application of a multi-segment well model to simulate flow in advanced wells," in *European Petroleum Conference*, 1998.

[14] R. S. G. A. Schenk, O., "The effects of unsymmetric matrix permutations and scalings in semiconductor device and circuit simulation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, pp. 400–411, 2004.

[15] T. A. Davis and E. Palamadai Natarajan, "Algorithm 907: KLU, a direct sparse solver for circuit simulation problems," *ACM Trans. Math. Softw.*, vol. 37, pp. 36:1–36:17, September 2010.

[16] X. Chen, Y. Wang, and H. Yang, "NICSLU: An adaptive sparse matrix solver for parallel circuit simulation," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 32, no. 2, pp. 261–274, 2013.

[17] W. Wu, Y. Shan, X. Chen, Y. Wang, and H. Yang, "Fpga accelerated parallel sparse matrix factorization for circuit simulations," in *Reconfigurable Computing: Architectures, Tools and Applications*. Springer, 2011, pp. 302–315.

[18] W. Wu, F. Gong, R. Krishnan, L. He, and H. Yu, "Exploiting parallelism by data dependency elimination: A case study of circuit simulation algorithms," *Design Test, IEEE*, vol. 30, no. 1, pp. 26–35, 2013.

[19] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix–vector multiplication on emerging multicore platforms," *Parallel Computing*, vol. 35, no. 3, pp. 178–194, 2009.

[20] J. Demmel, M. Hoemmen, M. Mohiyuddin, and K. Yelick, "Avoiding communication in sparse matrix computations," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE, 2008, pp. 1–12.

[21] N. Fujimoto, "Faster matrix-vector multiplication on GeForce 8800gtx," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE, 2008, pp. 1–8.

[22] G. Li and T. F. Coleman, "A parallel triangular solver for a distributed-memory multiprocessor," *SIAM Journal on Scientific and Statistical Computing*, vol. 9, no. 3, pp. 485–502, 1988.

[23] M. Naumov, "Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the gpu," *NVIDIA Corp., Westford, MA, USA, Tech. Rep. NVR-2011-001*, 2011.

[24] J. R. Gilbert and T. Peierls, "Sparse partial pivoting in time proportional to arithmetic operations," *SIAM J. Sci. Statist. Comput.*, vol. 9, pp. 862 – 874, 1988.

[25] X. Chen, W. Wu, Y. Wang, H. Yu, and H. Yang, "An escheduler-based data dependence analysis and task scheduling for parallel circuit simulation," *Circuits and Systems II: Express Briefs, IEEE Transactions on*, vol. 58, no. 10, pp. 702–706, 2011.

[26] N. Kapre and A. DeHon, "Parallelizing sparse matrix solve for spice circuit simulation using fpgas," in *International Conference on Field-Programmable Technology*, dec. 2009, pp. 190 –198.

[27] O. Schenk, K. Gatner, W. Fichtner, and A. Stricker, "PARDISO: a high-performance serial and parallel sparse linear solver in semiconductor device simulation," *Future Generation Computer Systems*, vol. 18, no. 1, pp. 69 – 78, 2001.