FAKULTÄT FÜR INFORMATIK

TECHNISCHE UNIVERSITÄT MÜNCHEN

Interdisciplinary Project (IDP)

Evaluation of Preconditioners for Element-Oriented Conjugate Gradients Solvers

Author: Danila Klimenko
Supervisor: Dipl.-Inf. Oliver Meister

Contents

1 Introduction

Within the scope of this interdisciplinary project we were working with sam(oa)^2, a software package for highly-efficient parallel solution of 2D partial differential equations. While sam(oa)^2 primarily focuses on hybrid parallelization and memory- and cache-efficiency, fast and stable iterative solvers are still essential for the overall performance of the package. One commonly used approach to performance optimization of iterative solvers is the preconditioning procedure.

This project focuses on the problem of preconditioning of iterative methods, or specifically on preconditioned conjugate gradients method. Given a collection of constraints derived from the architecture and implementation specifics of sam(oa)^2, we try to find an efficient preconditioner that could have increased the convergence speeds for the systems of equations solved by sam(oa)^2.

The goals of this project included the analysis and evaluation of a set of preconditioners with an objective of determining any valid candidates for integration into sam(oa)^2 framework.

Chapter 2 of this report provides the technical background for this project. First, an overview of the sam(oa)^2 software package is given together with its possible applications. Next, a novel parallel incomplete LU factorization algorithm is described. Chapter 3 presents the preconditioners evaluated within the scope of this project, as well as some notable implementation details. Chapter 4 focuses on experimental results and their evaluations.

2 Theoretical Background

In this chapter we describe sam(oa)^2, a software framework for high-efficient parallel solution of 2D partial differential equations, as well as its application to multi-phase flows in heterogeneous media.

We also discuss a new parallel incomplete LU factorization technique that allows for efficient fine-grained parallelization and uses an iterative approach for improving the accuracy of the decomposition.

2.1 sam(oa)^2 Framework

sam(oa)^2 (Space-filling Curves and Adaptive Meshes for Oceanic and Other Applications) is a software package that provides an adaptive element-oriented solver for 2D partial differential equations (PDEs) and focuses on memory- and cache-efficiency as well as hybrid parallelization. It is available online at `https://github.com/meistero/samoa/`, and a comprehensive description of the package can be found in [5] and [6]. We will focus our discussion on the aspects of the algorithms used in sam(oa)^2 that are relevant with regard to preconditioning of the systems that are being solved.

In order to support different kinds of problem discretizations, namely Finite Elements, Finite Volumes and Discontinuous Galerkin, sam(oa)^2 uses several levels of abstraction to decouple actual operators that work with elements from the underlying mechanics of grid traversals, refinement and parallelization. A user of the package is presented with an event-based interface and provides application-specific kernels that are triggered automatically during different stages of grid traversal.

The actual underlying data representation is based on dynamically adaptive triangular grids created using newest vertex bisection. The traversal order for the grid is dictated by the Sierpinski Space-Filling Curve (SFC), which allows to use memory-efficient data structures like stacks and streams for data exchange during and in-between traversals. Sierpinski SFC-defined sequential order is also used for optimized load-balancing and parallelization.

Usage of a Sierpinski SFC order for grid cell traversals leads to great cache-efficiency,

but it also restrains the formulation of PDE solvers to be matrix-free and element-oriented. During a grid traversal, the unknowns are processed as a stream, but at the same time, each element is accessed multiple times (not necessarily in consequent steps). To organize unknowns during repeated accesses, stack data structures are used: after being read from the input stream, an unknown is stored to and accessed from a system of stacks until the final update for this unknown is delivered, after which it is pushed to the output stream. Access rules to these stacks are deterministic [1] and derived from the Sierpinski SFC order.

Application-specific kernels provide a strong abstraction from the underlying traversal schemes and focus on neighboring element interactions. A Degree of Freedom (DoF) kernel operates on a single degree of freedom to, e.g., apply a vector update, whereas a Volume kernel may be used to compute matrix-vector products (using local stencils). Every access to an unknown during a traversal triggers a Volume kernel, while the first and/or last accesses may invoke DoF kernels.

One important takeaway from the previous description is the fact that during a single grid traversal old unknowns are not updated until all the traversal steps that access a particular unknown are taken: after the first access, temporary intermediate values are stored and updated on the stacks, and the unknown is actually updated only after the last access.

2.1.1 Application: Two-Phase Porous Media Flow

Being fine-tuned for cache- and memory-efficiency, sam(oa)^2 is best suited for applications whose performance is mostly memory-bound. The original paper describes two example use cases for the system [5]. The first application uses Finite Elements discretization to simulate multi-phase porous media flow, while the second one simulates tsunami wave propagation using shallow water equations and Finite Volumes method. We will briefly discuss the former, as we were using the systems of linear equations (SLE) generated from the discretization of the flow pressure equation to evaluate performance of different preconditioners.

The multi-phase flow in a heterogeneous porous medium was simulated using an approach courtesy of [3]. The following system of equations relates unknown pressure p and saturation S:

$$0 = \epsilon \frac{\partial S}{\partial t} + \nabla \cdot (F(S)\mathbf{u}) \tag{2.1}$$

$$\mathbf{u} = -\lambda_t(S)\mathbf{K} \cdot \nabla p \tag{2.2}$$

$$\nabla u = q \tag{2.3}$$

where ϵ is porosity, F is the fractional flow of the wetting phase, $\mathbf{u}$ is velocity, λ_t is the total mobility, $\mathbf{K}$ is the permeability tensor, and q is the source term.

This system is solved using a two-step implicit pressure, explicit saturation (IMPES) algorithm, where pressure p and velocity $\mathbf{u}$ are computed first implicitly from equations (2.2) and (2.3) respectively using Finite Elements discretization and preconditioned Conjugate Gradients (CG) method. The saturation S is computed next from equation (2.1) using Finite Volumes method and explicit time stepping.

On every time step, FE discretization of the pressure equation yields an SLE which is to be solved to find the correction for p. Solution of such an SLE is known to have low computational intensity and is usually memory-bound. sam(oa)^2 uses Jacobi-preconditioned Conjugate Gradients method to solve this SLE, and this is the exact point that we are trying to improve upon by evaluating various preconditioners that could potentially substitute the currently employed Jacobi preconditioner.

2.1.2 Fused Conjugate Gradients

Of particular interest is the optimized version of the Conjugate Gradients algorithm that is used to solve the pressure equation in the two-phase porous media flow problem.

The original preconditioned Conjugate Gradients algorithm can be formulated with the following pseudo-code [10]:

Algorithm 1 Traditional Conjugate Gradients Method

$r = b - Ax, \quad d = \vec{0}, \quad \beta = 0, \quad s \leftarrow P^{-1}r, \quad \gamma \leftarrow r^T s$
while $\|r\| > \epsilon$ **do**
  $d \leftarrow s + \beta d, \quad u \leftarrow Ad, \quad \delta \leftarrow d^T u$
  $\alpha \leftarrow \gamma/\delta, \quad \gamma_{old} \leftarrow \gamma$
  $x \leftarrow x + \alpha d, \quad r \leftarrow r - \alpha u$
  $s \leftarrow P^{-1}r, \quad \gamma \leftarrow r^T s$
  $\beta \leftarrow \gamma/\gamma_{old}$
end while

where A is the system matrix, x is the vector of unknowns, b is a vector of fixed parameters, P is the preconditioner, and r represents the residual vector while d represents the conjugate search directions.

The major drawback of the presented solver within the scope of sam(oa)^2 package is the fact that, due to the existing data dependency (between α, r, β and d), each CG iteration requires two grid traversals. And as discussed before, while the problem being solved is typically memory-bound, every traversal turns out to be a very expensive operation, so the total amount of traversals should be minimized.

To improve the performance of the algorithm and reduce the number of required traversals, the ideas of [8, 7] were adopted in order to implement *Fused* Conjugate Gradients algorithm. The Fused CG differs from the original algorithm in that β is being precomputed without knowing the new residual value using

$$\gamma = (r - \alpha u)^T P^{-1} (r - \alpha u) = r^T P^{-1} r - 2\alpha u^T P^{-1} r + \alpha^2 u^T P^{-1} u \tag{2.4}$$

But we can also show that $\alpha u_i^T P^{-1} r_i = \alpha d_i^T A (d_i - \beta_i d_{i-1}) = \alpha d_i^T A d_i = r_i^T P^{-1} r_i$, because the search directions d_{i-1} and d_i are orthogonal. Thus, we can precompute β just by evaluating

$$\gamma = \alpha^2 u^T P^{-1} u - r^T P^{-1} r \tag{2.5}$$

Although we are introducing an additional dot product and might loose some accuracy due to indirect computation of γ, we are effectively reducing the number of required traversals by a factor of two, which gives a significant performance boost.

The pseudo-code for the Fused CG algorithm is given below:

Algorithm 2 Fused Conjugate Gradients Method

$r = b - Ax, \quad d = \vec{0}, \quad u = \vec{0}, \quad \alpha = 0, \quad \beta = 0$
while $\|r\| > \epsilon$ **do**
  $x \leftarrow x + \alpha d, \quad r \leftarrow r - \alpha u$
  $s \leftarrow P^{-1} r, \quad \gamma \leftarrow r^T s$
  $d \leftarrow s + \beta d, \quad u \leftarrow Ad, \quad t \leftarrow u^T P^{-1} u, \quad \delta \leftarrow d^T u$
  $\alpha \leftarrow \gamma / \delta, \quad \gamma_{old} \leftarrow \gamma$
  $\gamma = \alpha^2 t - \gamma_{old}, \quad \beta \leftarrow \gamma / \gamma_{old}$
end while

It should be also mentioned that the Fused CG algorithm always requires one

additional iteration because the update of the solution x is skipped in the very first iteration, effectively shifting all the iteration numbers by one.

2.2 Parallel Incomplete LU Factorization

In this section we present a novel parallel algorithm for computing incomplete LU (ILU) factorizations [2]. It uses a significantly different approach to ILU factorization ans offers a very fine-grained level of parallelization.

The objective of an LU factorization (decomposition) of a system matrix A is to find the lower and upper factors L and U of A, such that $LU = A$. The *Incomplete* LU factorization aims at finding the approximations of L and U under some constraints, such that $LU \approx A$. Usually, given a sparse matrix A, an incomplete LU factorization of A builds factors L and U that obey some given sparsity pattern S (often the sparsity pattern of A itself), meaning that non-zero elements in L are only allowed at specified locations $(i, j) \in S$, $i \geq j$, and non-zero elements in U are allowed at $(i, j) \in S$, $i \leq j$.

A common method for building LU factorizations is the Gaussian Elimination process. For ILU factorization, we just disallow any non-zero values outside of the sparsity pattern S. Unfortunately, due to inherently sequential nature of Gaussian Elimination, parallelization of such an algorithm is very challenging.

Previous techniques used for parallelization of the aforementioned process usually rely on the idea of finding groups of elements of the matrix that can be computed in parallel. Common methods include level scheduling, multicolor ordering and domain decomposition. The authors of [2] introduce a completely new approach to formulating and finding the ILU factorization.

2.2.1 Reformulating ILU Factorization

Given an ILU decomposition $LU \approx A$ the new algorithm builds upon the fact that

$$(LU)_{ij} = (A)_{ij}, \quad (i, j) \in S \tag{2.6}$$

or in other words, that the ILU decomposition is exact on the sparsity pattern S [9]. Instead of treating the decomposition problem as a Gaussian Elimination process, the problem is reformulated as one of finding elements l_{ij} and u_{ij} of factors L and U that obey the constraints specified by equation (2.6).

Factor L is normalized to have a unit diagonal, and the total number of unknowns l_{ij}, u_{ij} is $|S|$ (the number of elements in the sparsity pattern S), which matches the number of equations (2.6), that can be rewritten as:

$$\sum_{k=1}^{min(i,j)} l_{ik}u_{kj} = a_{ij}, \quad (i,j) \in S \tag{2.7}$$

Knowing that the number of unknowns matches the number of equations, and that l_{ij} can only be part of S if $i > j$ (remembering that we normalized the diagonal $l_{ii} = 1$) or u_{ij} can be part of S if $i \leq j$, we can write down explicit formulas for each of the unknowns in terms of other unknowns:

$$\begin{aligned} l_{ij} &= \frac{1}{u_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik}u_{kj} \right), && i > j \text{ and } (i,j) \in S \\ u_{ij} &= a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj}, && i \leq j \text{ and } (i,j) \in S \end{aligned} \tag{2.8}$$

Now if we consider x to be a vector containing all unknowns l_{ij}, u_{ij}, the system of equations (2.8) can be seen as $x = G(x)$, which means that it can be solved using fixed-point iteration

$$x^{(p+1)} = G\left(x^{(p)}\right), \quad p = 0, 1, 2, \ldots \tag{2.9}$$

with some initial guess x_0 (usually the corresponding elements a_{ij} of matrix A can be used as initial guess for l_{ij} and u_{ij}, although other options are available).

Interestingly, if the elements of the fixed-point iteration (2.9) were updated in the Gaussian Elimination order while using the latests values for consequent updates (which we will call "Gauss-Seidel-type iteration", as opposed to "Jacobi-type iteration" that uses only the "old" values from previous iteration), the whole iteration would correspond to conventional ILU decomposition via Gaussian Elimination, and would produce the final solution after just one iteration (2.9).

But it was shown in [2] that using any ordering for element updates in iterations (2.9), with or without reusing the recently updated values within the same iteration, will produce a good ILU preconditioner. In other words, equations (2.8), delivering updates for unknowns l_{ij}, u_{ij}, may be treated independently, and computed in parallel without any need for specific element reordering and/or level scheduling.

Let one fixed-point iteration (2.9) be called a "sweep". Authors of [2] demonstrate that a few sweeps are sufficient to produce an approximation of ILU decomposition that acts as a good preconditioner. They also note that a matrix A should be diagonally scaled (in the form of DAD, where D is a diagonal scaling matrix) to have a unit diagonal prior to starting the sweeps.

To conclude this description, the sketch of the new parallel ILU factorization algorithm is given below:

Algorithm 3 Fine-Grained Parallel ILU Decomposition

Initialize the unknown values l_{ij}, u_{ij} given an appropriate initialization scheme
for $sweep = 1, 2$ **to** MAX_SWEEP **do**
  parallel for $(i, j) \in S$ **do**
    if $i > j$ **then**
      $l_{ij} = \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj}\right) / u_{jj}$
    else
      $u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj}$
    end if
  end parallel for
end for

2.2.2 Approximate Triangular Solvers for Preconditioning

When applied to the problem of preconditioning, (incomplete) LU factorization heavily relies on triangular system solvers, as preconditioning a vector r for matrix A given its factors L and U means solving a system $LU\hat{r} = r$ for $\hat{r}$. Actually, two triangular systems need to be solved in order to obtain the preconditioned vector $\hat{r}$: first, $Lr' = r$ is solved for r', and then $U\hat{r} = r'$ is solved for $\hat{r}$.

In sequential setting, triangular systems may be easily solved using forward-substitution (for lower triangular systems L) and backward-substitution (for upper triangular systems U), which, importantly, works only given that recently updated values are reused for computing consequent updates within the same iteration (as before, referred to as Gauss-Seidel-type iteration).

Several techniques exist for solving triangular systems in parallel. Different variations of level scheduling try to find subsets of elements that can be updated in parallel. Many methods build upon the idea of approximating the inverse of a sparse triangular matrix L using either a product of other sparse triangular factors, or truncated Neumann series, or sparse approximate inverse techniques.

Authors of [2] introduce an iterative approximate solver for triangular systems that matches their parallel ILU factorization algorithm. Given a system $Lx = b$ with a triangular matrix L, an approximation for x is obtained via a small fixed number of fixed-point iterations of the form

$$x^{(k+1)} = \left(I - D^{-1}L\right) x^{(k)} + D^{-1}b \tag{2.10}$$

where D is a diagonal matrix containing only the diagonal elements l_{ii} of L. Such an algorithm no longer requires Gauss-Seidel-type iterations, and while a fixed number of iterations is used, the preconditioning operator remains a fixed linear operator. The fact that x remains an approximation is tolerable, because preconditioning itself is an approximate operation.

Experimental results provided in [2] show that an approximate iterative solver based on (2.10) (using a fixed number of iterations) may result in less accurate preconditioning, but significantly outperforms level scheduling techniques in terms of required time-to-solution.

3 Implementation

In this chapter we give an overview of all the preconditioners that were implemented and evaluated within the scope of this project, as well as some relevant or interesting implementation details.

With respect to the problems being solved, i.e. systems of linear equations (SLEs) resulting from an FE discretization of the pressure equation for a two-phase porous media flow (see. Chapter 2 for more details), we consider SLEs of the form $Ax = b$ where A is a sparse symmetric matrix, x is a vector of unknows to be found, and b is a vector of fixed right-hand-side parameters. We are looking at different preconditioners P to be used as an integral part of the Preconditioned (Fused) Conjugate Gradients method, which is applied to solve the aforementioned SLEs.

Another important point for consideration throughout this chapter will be the number of grid traversals required for every iteration of the actual solver. As mentioned in Section 2.1.1, the problem being solved has memory-bound performance, and every grid traversal is an expensive memory-scanning operation, so the total amount of traversals should be minimized. The Fused Conjugate Gradients method currently implemented in sam(oa)^2 integrates one CG iteration together with Jacobi preconditioning into one grid traversal. This result, if applicable, will be used as a reference point for other preconditioners.

All the algorithms, preconditioners, tests and benchmarks were implemented in Python (and partially C) using *Interactive Python* environment and *NumPy* and *SciPy* scientific computing libraries [4]. The system matrix A is always represented with a special *SciPy* sparse matrix class (using Compressed Sparse Row storage format), and all the vectors are represented with *NumPy* `ndarrays`.

3.1 Jacobi Preconditioner

Being one of the simplest forms of preconditioning, Jacobi (also known as Diagonal) preconditioner is usually easy to implement, and it gives moderate reduction in the number of CG iterations depending on the problem being solved.

Jacobi preconditioner P is built just by taking the main diagonal D of the system matrix A:

$$P = D, \qquad (D)_{ij} = \begin{cases} (A)_{ij} & \text{if } i = j \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

and applying it to a vector r is achieved by multiplying this vector with the inverse of P as in $\hat{r} = P^{-1}r$. To inverse P in this case means simply inversing all the diagonal elements of P.

Jacobi preconditioner is considered efficient for diagonally dominant matrices. More importantly for us, it does not require the reuse of updated values within the same iteration, and it can be integrated into the Fused CG algorithm of sam(oa)^2 without requiring additional grid traversals.

In terms of a Python implementation for our evaluation purposes, Jacobi preconditioner is straightforward and does not provide any point of interest.

3.2 Iterative Jacobi Preconditioner

The approach presented in this section is not generally considered as a preconditioner, and was inspired by the parallel iterative approximate solver for triangular systems described in Section 2.2.2.

We considered taking preconditioner P equal to the system matrix A, and using some fixed number N of regular Jacobi iterations for an SLE $P\hat{r} = r$ to obtain the preconditioned vector $\hat{r}$ (using vector r itself as the initial guess $r^{(0)}$):

$$\begin{aligned} r^{(0)} &= r \\ r^{(k+1)} &= D^{-1}\left(r - (A - D)\, r^{(k)}\right), \qquad k = 0, 1, \ldots, (N-1) \\ \hat{r} &= r^{(N)} \end{aligned} \tag{3.2}$$

Using this approach we, again, do not have to reuse the updated values of unknowns within the same iteration, but the number of required grid traversals per one CG iteration is now effectively equal to the number N of iterations (3.2) needed to apply the preconditioner.

3.3 Symmetric Gauss-Seidel Preconditioner

Symmetric Gauss-Seidel preconditioner builds upon the ideas of the regular Gauss-Seidel method and applies a series of a backward-substitution, a vector product, and a forward-substitution to obtain the preconditioned vector $\hat{r}$. It is also a special case of the Symmetric Successive Over-Relaxation (SSOR) for the weight ω taken equal to 1.

Let D be the main diagonal of the system matrix A, L be the strictly lower triangular part of A, and U be the strictly upper triangular part of A, which in our case of a symmetric matrix A is actually equal to L^T. Then the symmetric Gauss-Seidel preconditioner P can be represented as

$$P = (D + L)\, D^{-1}\, (D + L)^T \tag{3.3}$$

Thus, applying a Gauss-Seidel preconditioner (3.3) to a vector r is a three-stage process consisting of `(1)` solving the system $(D + L)\, r' = r$ for r' using forward-substitution, `(2)` solving $D^{-1} r'' = r'$ for r'' by multiplying r' with D, and `(3)` solving $(D + L)^T\, \hat{r} = r''$ for $\hat{r}$ using backward-substitution.

Symmetric Gauss-Seidel preconditioner usually performs significantly better than Jacobi preconditioner in terms of the reduction of the total number of CG iterations. But its reliance on the forward- and backward-substitution processes requires an ability to reuse recently updated values of unknowns within the same iteration, and makes the whole method inherently sequential. Although several different techniques for parallel application of a Symmetric Gauss-Seidel preconditioner do exist, none of them are directly applicable to the Fused CG solver implemented in sam(oa)^2 due to the way how updates are delivered to the unknowns during a single grid traversal.[1]

There is also a note of interest regarding the implementation of the Symmetric Gauss-Seidel preconditioner within the environment of this project. Initial test of the implementation have shown that neither *NumPy* nor *SciPy* libraries have built-in support for efficient forward- and/or backward substitution. In order to make the results of the Symmetric Gauss-Seidel preconditioner comparable to results of other preconditioners, a dedicated native-code module had to be implemented to extend the application programming interface (API) of *SciPy* sparse matrices.

[1] Consider, for example, a red-black Gauss-Seidel parallel algorithm. Basically, the idea is to partition all the unknowns into two disjoint groups such that elements within the groups can be updated in parallel. This is efficient only under assumption that processing half the elements takes (roughly) half of the original effort. But in sam(oa)^2, we cannot update half the elements without traversing the whole grid. End even if we skip the updates for half of the visited elements, the total effort will not decrease significantly because a grid traversal is a memory-bound operation with low computational intensity.

3.4 Incomplete Cholesky Preconditioner

Incomplete Cholesky preconditioner uses the matrix factorization of the same name and is a special case of an incomplete LU factorization for symmetric matrices. Given a sparse symmetric matrix A, its incomplete Cholesky factorization is given as $LL^T \approx A$, where L is a lower-triangular sparse matrix with some given sparsity pattern S (which is usually chosen to match the sparsity pattern of A).

Incomplete Cholesky preconditioner P for a system matrix A is built using the corresponding incomplete Cholesky factorization of A:

$$P = LL^T \tag{3.4}$$

Applying the preconditioner (3.3) to a vector r is, similarly to the case of Symmetric Gauss-Seidel preconditioner, a two-stage process involving consequent solutions of systems $Lr' = r$ and $L^T\hat{r} = r'$ for r' and $\hat{r}$ respectively.

Incomplete Cholesky preconditioners are usually considered to be quite efficient, but they, again, rely on the existence of an efficient solver for triangular systems. The same reasoning applies as with the Symmetric Gauss-Seidel preconditioner: there are no efficient parallel triangular solvers that are directly applicable to the Fused CG algorithm used in sam(oa)^2.

As discussed in more detail in the later Section 3.5.2, we try to apply the parallel iterative approximate solver courtesy of [2] to overcome the aforementioned issue of efficient application of the preconditioner.

3.5 Fine-Grained Parallel Incomplete LU Family

In this section we describe the implementation of a family of preconditioners based on the Fine-Grained Parallel Algorithm 3 for finding incomplete LU factorizations. The overview of the algorithm was given in Section 2.2 and its detailed description is provided in [2].

Given that our system matrix A is symmetric, the Algorithm 3 will actually try to compute an approximation of the incomplete Cholesky factorization (which is an approximation itself). The preconditioner (that will be henceforth called "the PILU

Preconditioner") will then be built from resulting factorization:

$$P = LL^T, \qquad \left(LL^T\right)_{ij} = (A)_{ij}, \quad (i,j) \in S \tag{3.5}$$

As both the computation of the ILU factorization and the application of the resulting PILU preconditioner (3.5) present their own challenges, we will split their discussion into two following separate Sections.

From now on, we will discern the computation of the factorization and the application of the preconditioner as two separate building blocks that, being combined, constitute an instance of a PILU preconditioner. It is worth noting that the implementation of the PILU preconditioner follows the same pattern, presenting a modular architecture that can integrate different types of factorizers and/or triangular solvers (used to apply the preconditioner).

3.5.1 ILU Factorizers

Being seen as a building block for the PILU preconditioner, a factorizer has an objective of producing an approximation of the incomplete LU decomposition given a system matrix A. Of the main interest here is the new fine-grained parallel incomplete LU factorization Algorithm 3, but it is worth mentioning that a conventional ILU factorization algorithm based on the Gaussian Elimination process can also be used as a factorizer for the PILU preconditioner.

A flexible factorizer built following the Algorithm 3 is parameterized on the number of sweeps (iterations) that are applied in order to approximate the factorization, and supports two types of iterations: the Gauss-Seidel-type iterations, that allow the reuse of freshly computed values for consequent updates within the same iteration, and the Jacobi-type iterations, that use only the "old" values from previous iteration.

In order to simulate (and test) the performance of the algorithm in a parallel setting, the relative order of equations (2.8) that are used to update the unknowns is randomized before every sweep. This randomization is supposed to replicate the ambiguity of the update order of the unknowns in case of a fine-grained parallel implementation of the factorizer.

As mentioned in the Section 2.2, the system matrix A should be scaled to unit diagonal prior to performing the sweeps of the algorithm. The scaling has a form of

$\tilde{A} = DAD$, where D is a diagonal scaling matrix with diagonal elements computed as

$$(D)_{ij} = \begin{cases} \frac{1}{\sqrt{(A)_{ij}}} & \text{if } i = j \\ 0 & \text{otherwise} \end{cases} \tag{3.6}$$

Now assuming that the Algorithm 3 returns an ILU factorization of $\tilde{A}$ in the form of $\tilde{L}\tilde{L}^T$, we can restore the original scale for L by left-multiplying $\tilde{L}$ with the inverse of D: $L = D^{-1}\tilde{L}$.

Finally, it is worth mentioning that our implementation was using the sparsity pattern S of the system matrix A as the sparsity pattern for the computed factors L and L^T, and the corresponding elements of matrix A were used as the initial guess for the elements of factor L.

3.5.2 Triangular Solvers

Having built the ILU decomposition LL^T of the system matrix A, the problem of applying it to a vector r arises. As discussed before, building an efficient parallel triangular solver is a challenging problem on its own. It becomes even more complicated in the settings of sam(oa)^2, where neither a reuse of updated values within the same iteration, nor an efficient update of a subset of unknowns are permitted.

The objective of a triangular solver, when considered as a building block for the PILU preconditioner, can be formulated as follows: given an ILU decomposition LL^T and a vector r, find a vector $\hat{r}$ that satisfies $LL^T\hat{r} = r$.

Again, of main interest here is the parallel iterative approximate solver courtesy of [2] that was described in Section 2.2.2, although other options are also considered.

The optimal triangular solver for our problem would still be the pair of a forward- and a backward-substitutions, which unfortunately remain inherently sequential. A naïve diagonal solver turns out to be illogical due to the fact that, as mentioned before, the factorization LL^T is considered to be exact on the sparsity pattern of A, which means that the diagonal of LL^T approximates the diagonal of A, and the resulting diagonal solver would just approximate the basic Jacobi preconditioner.

Being left with the parallel iterative approximate solver, while its implementation is quite straightforward, we should mention that every iteration of this solver would actually require two grid traversals, because we have to solve two triangular systems $Lr' = r$ and $L^T\hat{r} = r'$ in order to apply the preconditioner the a vector r.

4 Experiments and Evaluation

In this chapter we present the comparison results for the set of preconditioners described in Chapter 3 when applied to the FE discretized pressure equation being solved using a Fused Conjugate Gradients solver (both described in Chapter 2).

All experiments were conducted in an *Interactive Python* environment with included *NumPy* and *SciPy* scientific computing libraries [4] on a single-processor dual-core machine with a 1.7 GHz Intel Core i7 processor and 8 GB of RAM.

Both the numbers of CG iterations and the absolute running time values (if provided), that are found in this chapter, are averaged over multiple repeated runs of the measured algorithms. The residual threshold used to terminate the solvers is fixed and shared among all experiments (unless mentioned otherwise). Along with the numbers of CG iterations and the running times, the total number of grid traversals is considered for discussion when applicable.

First we present and discuss the results of a series of experiments conducted for a basic discrete Poisson equation problem. Next we proceed to experiments with the discretized pressure equations generated by the sam(oa)^2 package from the two-phase porous media flow simulations. Some observations and findings are discussed after the presentation of experimental results.

4.1 Poisson Test Problem

In order to verify the built environment, and test all the implemented algorithms and preconditioners, a simple 2D discrete Poisson equation

$$\left(\nabla^2 u\right)_{ij} = \frac{1}{\Delta x^2}\left(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{ij}\right) = g_{ij}, \quad (i,j) \in [1,n] \times [1,m] \tag{4.1}$$

was considered for solving. Rewriting equations (4.1) in a matrix-vector form

$$Au = b \tag{4.2}$$

yields a symmetric sparse matrix A and a vector u of $(n \cdot m)$ unknowns.

For our experiments, we consider a constant zero boundary and a constant non-zero right-hand-side vector b. The initial guess is initialized to a sine curve.

4.1.1 CG with Basic Preconditioners

First we used the regular Conjugate Gradients method combined with a set of different preconditioners to solve the system (4.2). The set of preconditioners included the Jacobi, Symmetric Gauss-Seidel and Incomplete Cholesky preconditioners, as well as the Parallel Incomplete LU (PILU) preconditioner equipped with a parallel factorizer that uses 3 Jacobi-type sweeps, and a forward-substituting triangular solver. The experimental results are presented in Table 4.1.

Size	Iterations					Running time				
	(none)	Jac	SGS	Chol	PILU	(none)	Jac	SGS	Chol	PILU
900	48	48	26	22	23	2.6	2.5	2.7	2.7	2.2
1600	63	63	34	29	30	3.8	4.3	5.3	4.4	5.0
2500	79	79	38	34	36	7.5	8.2	6.9	5.7	6.0
3600	95	95	44	38	39	9.5	10.4	10.0	8.2	8.4
4900	111	111	50	44	44	13.3	14.5	14.1	11.9	11.9

Table 4.1: Number of iterations and running times (ms) for regular CG method using different basic preconditioners (Poisson equation).

The results we see do match our theoretical expectations. The Jacobi preconditioner is known to be inefficient for the Poisson equation problem; the Symmetric Gauss-Seidel greatly reduces the number of CG iterations, but due to the higher cost of the preconditioning operation it is still outperformed by the non-preconditioned solver; Incomplete Cholesky preconditioner is the most efficient one both in terms of the number of CG iterations, and in terms of the running time (but the running time is still close to the non-preconditioned case, even though the number of iterations dropped by almost a factor of 3).

Interestingly, the selected configuration of the PILU preconditioner performs almost exactly as the Incomplete Cholesky preconditioner. While both use the same forward-

/backward-substitution process to apply the internal factorization to a vector, the factorization itself is acquired using completely different approaches. This allows us to conclude that the PILU Algorithm 3 builds a good approximation of the Incomplete LU decomposition (even when using the Jacobi-type sweeps).

4.1.2 Classical CG versus Fused CG

Now we aim to compare the relative performance of a regular and a Fused Conjugate Gradients solvers. Experiments are conducted using the Jacobi and the Incomplete Cholesky preconditioners, and the results are shown in Table 4.2.

Size	Iterations			Running time					
				Regular CG			Fused CG		
	(none)	Jac	Chol	(none)	Jac	Chol	(none)	Jac	Chol
900	48	48	22	2.5	2.5	2.5	2.6	2.8	3.0
1600	63	63	29	3.9	4.3	3.7	4.3	4.7	5.6
2500	79	79	34	6.0	7.2	5.5	6.6	7.6	8.6
3600	95	95	38	9.1	10.0	8.1	9.9	11.5	12.7
4900	111	111	44	13.0	14.5	12.0	14.5	16.5	18.9

Table 4.2: Number of iterations and running times (ms) for regular CG and Fused CG methods with different preconditioners (Poisson equation).

Important note here is that the number of CG iterations taken before reaching the threshold was the same for both the regular and the Fused CGs throughout all the experiments (taking into account the extra iteration required for the Fused CG (see Section 2.1.2)).

As a general observation, the Fused CG algorithm has running time that is (roughly) a fixed multiple of the running time of the regular CG. This may be explained, e.g., by the extra dot product that we have introduced to every iteration.

We can also note that the slowdown in case when we are using the Incomplete Cholesky preconditioner is more pronounced compared to other cases. This could be probably explained by the fact that the application of the Incomplete Cholesky preconditioner is a more expensive operation than, e.g., the application of Jacobi preconditioner, and that our implementation of the Fused CG algorithm directly follows the pseudo-code of Algorithm 2, where the preconditioner is actually applied twice in every iteration.

4.2 sam(oa)^2 Pressure Equation for Porous Media Flow Problem

We now use our environment to evaluate the relative performance of different preconditioners when applied to solving systems of linear equations generated by sam(oa)^2 during the simulation of a two-phase porous media flow. The systems to be solved can be formulated as

$$Ap = b \tag{4.3}$$

where A is a sparse symmetric system matrix and p is an unknown pressure vector.

Starting from this section, we will pay additional attention to the number of grid traversals required per one CG iteration, as the running time measurements obtained within our project environment are not expected to be representative of the actual running times one may observe from the runs of sam(oa)^2 package due to significant differences in implementation details. Instead, the number of grid traversals can be treated as an invariant for estimating relative performance within the sam(oa)^2 framework.

4.2.1 Basic Preconditioners versus Parallel ILU

Again, we first solve the system (4.3) using the Fused CG algorithm together with different preconditioners in order to get an overview of performance of those preconditioners. The set of used preconditioners consists of the Jacobi, Symmetric Gauss-Seidel, Incomplete Cholesky and PILU preconditioners, where the latter is using a 3 Jacobi-type sweeps factorizer and a forward-substitution triangular solver. The results of the experiments are provided in Table 4.3.

Size	Iterations					Running time				
	(none)	Jac	SGS	Chol	PILU	(none)	Jac	SGS	Chol	PILU
1985	185	150	60	59	58	14.4	12.9	13.9	14.8	12.2
3574	226	193	77	73	73	25.2	25.3	28.9	23.5	24.3
4763	239	187	75	73	73	32.1	29.4	36.8	30.2	30.5
5154	247	188	76	71	72	35.7	31.6	40.5	31.6	35.3

Table 4.3: Number of iterations and running times (ms) for Fused CG method with different preconditioners (Pressure equation).

Several interesting observations arise from these results. First, Jacobi preconditioner has become effective as it now reduces both the number of CG iterations and the running time of the respective solver. Symmetric Gauss-Seidel now shows the worst performance in terms of the running time among all tested preconditioners. Incomplete Cholesky, while still providing the best reduction of the number of CG iterations, is now sometimes outperformed in terms of running time by the Jacobi preconditioner. The PILU preconditioner still performs similarly to the Incomplete Cholesky.

But the most important observation is actually the following: each of the considered preconditioners, with exception for the Jacobi preconditioner, relies on the forward- and backward-substitution processes to actually apply the preconditioning operator to a given vector. As discussed in Section 2.2.2 and detailed throughout Chapter 3, such substitution processes are impossible within the sam(oa)^2 framework, which effectively renders useless all the preconditioners considered so far, except for the already implemented Jacobi preconditioner.

Thus, we are interested only in preconditioners that rely exclusively on Jacobi-type iterative processes. From all the preconditioners described within this project, only some of the PILU preconditioners obey these constraints: the ones that use Jacobi-type iterative factorizer and the parallel iterative approximate triangular solver.

4.2.2 Experimenting with Factorizer Parameters

With the following experiment, we want to evaluate how does the alteration of different settings available for the fine-grained parallel ILU factorizer affect the preconditioning properties of the respective PILU preconditioner. We use Jacobi and Incomplete Cholesky preconditioners as benchmarks, and consider PILU factorizers that use different number of sweeps (1 or 3 in the presented experiment), or different iteration-types of sweeps (Jacobi-type or Gauss-Seidel-type). The results of the experiment are presented in Table 4.4.

As seen from these results, both the type of iteration used (Jacobi- vs Gauss-Seidel-) and the number of performed sweeps barely affect the performance of the corresponding PILU preconditioner for our specific problem. Thus, even a single Jacobi-type sweep factorizer can be considered appropriate for building an efficient PILU preconditioner.

4.2.3 Comparing Triangular Solvers

Finally, to build a PILU preconditioner that can be used within the framework of sam(oa)^2, we have to use a Jacobi-type iterative triangular solver, and our main choice

Size	Iterations					
	Jac	Chol	Jacobi-type		Gauss-Seidel-type	
			1 sweep	3 sweeps	1 sweep	3 sweeps
1985	150	59	59	58	60	58
3574	193	73	74	73	75	73
4763	187	73	73	73	74	73
5154	188	71	72	72	74	72
	Running time					
1985	13.7	13.7	13.6	13.2	12.4	12.5
3574	24.2	25.5	24.7	23.8	24.6	23.7
4763	29.3	30.5	30.6	30.6	30.7	30.8
5154	31.3	32.0	33.5	32.5	33.8	32.6

Table 4.4: Number of iterations and running times (ms) for Fused CG method with varying parameters of PILU factorizer (Pressure equation).

here is the parallel iterative approximate solver described in Section 2.2.2. This solver is parameterized on the number of iterations it applies to precondition a vector, so we will evaluate the performance of respective PILU preconditioners with respect to varying number of those iterations. Again, we are using Jacobi and Incomplete Cholesky preconditioners as benchmarks, and the results of the experiment are shown in Table 4.5.

Size	Iterations					Running time				
	Jac	Chol	Triangular solver			Jac	Chol	Triangular solver		
			Jac 1	Jac 3	Jac 5			Jac 1	Jac 3	Jac 5
1985	150	59	167	66	61	14.1	13.9	39.9	36.6	48.6
3574	193	73	241	83	75	27.0	26.5	84.3	64.5	89.8
4763	187	73	199	81	74	33.0	34.3	98.4	85.1	116.4
5154	188	71	199	80	74	35.7	35.6	98.3	89.8	133.5

Table 4.5: Number of iterations and running times (ms) for Fused CG method with varying triangular solvers of PILU preconditioner (Pressure equation).

From the results of the experiment we can see that iterative approximate solver with just one preconditioning iteration actually increases the number of required CG iterations and is, thus, completely inapplicable. On the other hand, using too many iterations – 5 in case of this experiment, while reducing the number of required CG iterations to almost the level of Incomplete Cholesky preconditioner, actually more than triples the running time of the respective preconditioner. Using 3 iterations of the

approximate solver also provides a significant decrease in the number of CG iterations, but still doubles the running time of the preconditioner.

If, as discussed before, we ignore for a moment the running times of the preconditioned algorithms within the environment of this project, and consider instead the respective numbers of grid traversals, then we get even more disappointing results. Even one iteration of the iterative approximate solver requires two grid traversals, because we actually have to solve two triangular systems (one lower and one upper) to apply the preconditioner. In order for a preconditioner to be beneficial for the whole algorithm, it has to reduce the total number of grid traversals. This means that if the application of a preconditioner requires k grid traversals per one CG iteration, it has to reduce the total number of CG iteration at least by a factor of k. Considering the preconditioners from the last experiment, 1-iteration solver is still out of the question as it increases the number of CG iterations; 3-iteration solver reduces the number of CG iterations by a factor of $\approx 2.4\times$, while requiring 6 grid traversals per CG iteration, thus effectively increasing the total number of grid traversals by a factor of $\approx 2.5\times$. 5-iteration solver leads to an even worse increase in the number of grid traversals.

Thus we can conclude that none of the triangular solvers that use Jacobi-type iterations could be used to build an efficient PILU preconditioner for the Fused CG algorithm implemented in sam(oa)^2 framework.

5 Conclusions

As we have discussed throughout the Chapters 2 and 3, sam(oa)^2 imposes some strict constraints on the type and order of operations that can be applied to the grid elements of the system being solved. Specifically, no updates can propagate within one grid traversal, as the elements can be updated only after all their neighbors have been visited. Moreover, selective updates of subsets of elements, though not impossible, are inefficient as they still require a full grid traversal.

These limitations render most existing preconditioners and algorithms inapplicable to sam(oa)^2. This includes all the algorithms that rely on forward- and/or backward-substitution routines, such like Symmetric Gauss-Seidel preconditioners (and any Symmetric Successive Over-Relaxation in general) and Incomplete Cholesky preconditioners. Commonly used parallelization techniques like level scheduling or multicolor ordering are also not directly applicable to sam(oa)^2 due to strict inherent order of grid traversals.

The novel fine-grained parallel incomplete LU factorization algorithm, described in Section 2.2, indeed allows to easily and efficiently build an LU decomposition of a matrix, but the use of this decomposition as a preconditioner requires an efficient solver for triangular systems. Standard options include the substitution algorithms and/or scheduling techniques, both of which are inapplicable within the framework of sam(oa)^2. The proposed iterative approximate triangular solver can be implemented in sam(oa)^2, but the experimental results show that it will only deteriorate performance.

Thus, of all the preconditioners evaluated within the scope of this project, only the basic Jacobi preconditioner turns out to be a reasonable candidate for integration into sam(oa)^2 framework, and it is, in fact, already implemented and used by the software package.

Bibliography

[1] M. Bader, C. Böck, J. Schwaiger, and C. Vigh. "Dynamically Adaptive Simulations with Minimal Memory Requirement — Solving the Shallow Water Equations Using Sierpinski Curves." In: *SIAM Journal on Scientific Computing* 32.1 (2010), pp. 212–228. DOI: 10.1137/080728871.

[2] E. Chow and A. Patel. "Fine-grained Parallel Incomplete LU Factorization." In: *SIAM Journal on Scientific Computing* 37.2 (2015), pp. C169–C193. DOI: 10.1137/140968896.

[3] C. Chueh, M. Secanell, W. Bangerth, and N. Djilali. "Multi-level Adaptive Simulation of Transient Two-Phase Flow in Heterogeneous Porous Media." In: *Computers & Fluids* 39.9 (2010), pp. 1585–1596. DOI: 10.1016/j.compfluid.2010.05.011.

[4] E. Jones, T. Oliphant, P. Peterson, et al. *SciPy: Open source scientific tools for Python*. http://www.scipy.org/. 2001–.

[5] O. Meister, K. Rahnema, and M. Bader. "A Software Concept for Cache-Efficient Simulation on Dynamically Adaptive Structured Triangular Grids." In: *Applications, Tools and Techniques on the Road to Exascale Computing (Advances in Parallel Computing)* 22 (2012), pp. 251–260. DOI: 10.3233/978-1-61499-041-3-251.

[6] O. Meister, K. Rahnema, and M. Bader. "Parallel Adaptive Mesh Refinement for 2D Multiphase Flow and Tsunami Wave Propagation." In: *ACM Transactions on Mathematical Software* 9.4 (2010). Article 39. 24 pages.

[7] G. Meurant. "Multitasking the Conjugate Gradient Method on the {CRAY} X-MP/48." In: *Parallel Computing* 5.3 (1987), pp. 267–280. DOI: 10.1016/0167-8191(87)90037-8.

[8] Y. Saad. "Practical Use of Polynomial Preconditionings for the Conjugate Gradient Method." In: *SIAM Journal on Scientific and Statistical Computing* 6.4 (1985), pp. 865–881. DOI: 10.1137/0906059.

[9] Y. Saad. *Iterative Methods for Sparse Linear Systems: Second Edition*. Society for Industrial and Applied Mathematics, 2003. ISBN: 978-0898715347.

[10] J. R. Shewchuk. *An Introduction to the Conjugate Gradient Method without the Agonizing Pain*. Tech. rep. Carnegie Mellon University, Pittsburgh, PA, USA, 1994.