Parallel Preconditioner

SPAI Algorithm

YingZhang's work

Speaker: Ying Zhang Jan 5, 2024

Teamwork/Mywork

At the project's outset, Chen was assigned the first three issues, Guo took on the fourth, and I handled the fifth. In the initial weeks, Guo opted to focus on exams, while Chen proposed a new idea just days before the project deadline. He wanted to explore the fifth issue independently, so I am now solely responsible for it. My project builds upon the framework Chen established, and I have incorporated his work into mine. Despite our divergent paths, I continue to integrate his contributions into my focused work on the fifth issue.

I'm currently working on the m5 branch of the project. Here's the link to the README file: https://github.com/AMSC22-23/matvetcg-chen-zhang-guo/blob/m5/README.md where you can find detailed information about my contributions.

Implement a parallel preconditioner based on approximate inverse

ApproximateInversePreconditioner.pdf *d*escribes the SParse Approximate Inverse(SPAI) algorithm.

The SPAI algorithm computes a sparse approximate inverse M of a general sparse matrix A. It is inherently parallel, since the columns of M are calculated independently of one another.

SIAM J. SCI. COMPUT.
Vol. 18, No. 3, pp. 838–853, May 1997

009

PARALLEL PRECONDITIONING WITH SPARSE APPROXIMATE INVERSES*

MARCUS J. GROTE† AND THOMAS HUCKLE‡

Abstract. A parallel preconditioner is presented for the solution of general sparse linear systems of equations. A sparse approximate inverse is computed explicitly and then applied as a preconditioner to an iterative method. The computation of the preconditioner is inherently parallel, and its application only requires a matrix-vector product. The sparsity pattern of the approximate inverse is not imposed a priori but captured automatically. This keeps the amount of work and the number of nonzero entries in the preconditioner to a minimum. Rigorous bounds on the clustering of the eigenvalues and the singular values are derived for the preconditioned system, and the proximity of the approximate to the true inverse is estimated. An extensive set of test problems from scientific and industrial applications provides convincing evidence of the effectiveness of this approach.

Key words. preconditioning, approximate inverses, parallel algorithms, sparse matrices, sparse linear systems, iterative methods

AMS subject classifications. 65F10, 65F35, 65F50, 65Y05

PII. S1064827594276552

1. Introduction. We consider the linear system of equations

$$Ax = b, \quad x, b \in \mathbb{R}^n . \tag{1}$$

Here A is a large, sparse, and nonsymmetric matrix. Due to the size of A, direct solvers become prohibitively expensive because of the amount of work and storage required. As an alternative we consider iterative methods such as generalized minimal residual (GMRES), biconjugate gradient (BCG), biconjugate gradient stabilized (Bi-CGSTAB), and conjugate gradient (CG) applied to the normal equations [6]. Given the initial guess x_0, these algorithms compute iteratively new approximations x_k to the true solution $x = A^{-1}b$. The iterate x_m is accepted as a solution if the residual $r_m = b - Ax_m$ satisfies $\|r_m\|/\|b\| \leq \text{tol}$. In general, the convergence is not guaranteed or may be extremely slow. Hence, the original problem (1) must be transformed into a more tractable form. To do so, we consider a preconditioning matrix M and apply the iterative solver either to the right or to the left preconditioned system

$$AMy = b , \quad x = My , \quad \text{or} \quad MAx = Mb . \tag{2}$$

Therefore, M should be chosen such that AM (or MA) is a good approximation of the identity. As the ultimate goal is to reduce the total execution time, both the computation of M and the matrix-vector product My should be evaluated in parallel. Since the matrix-vector product must be performed at each iteration, the number of nonzero entries in M should not exceed that in A.

*Received by the editors May 6, 1994; accepted for publication (in revised form) September 14, 1995.
http://www.siam.org/journals/sisc/18-3/27655.html
†Scientific Computing and Computational Mathematics, Building 460, Stanford University, Stanford, California, 94305 (grote@cims.nyu.edu). The research of this author was supported by an IBM fellowship.
‡Institut für Angewandte Mathematik und Statistik, Universität Würzburg, D-97074 Würzburg, Germany (huckle@informatik.tu-muenchen.de). The research of this author was supported by a research grant of the Deutsche Forschungsgemeinschaft.

Idea

Consider the linear system of equations:

$$A\mathbf{x} = \mathbf{b}, \quad \mathbf{x}, \mathbf{b} \in \mathbb{R}^n$$

Here A is a large, sparse and nonsymmetric matrix.

This paper considers a preconditioning matrix M and apply the iterative solver either to the right or to the left preconditioned system

$$AMy = b, \quad x = My, \quad or \quad MAx = Mb\,.$$

THE SPAI ALGORITHM

For every column m_k of M:

(a) Choose an initial sparsity $\mathcal{J}$.

(b) Compute the row indices $\mathcal{I}$ of the corresponding nonzero entries and the QR decomposition (6) of $A(\mathcal{I}, \mathcal{J})$. Then compute the solution m_k of the least squares problem (4) and its residual r given by (8).
While $\|r\|_2 > \varepsilon$:

(c) Set $\mathcal{L}$ equal to the set of indices ℓ for which $r(\ell) \neq 0$.

(d) Set $\tilde{\mathcal{J}}$ equal to the set of all new column indices of A that appear in all $\mathcal{L}$ rows but not in $\mathcal{J}$.

(e) For each $j \in \tilde{\mathcal{J}}$ solve the minimization problem (10).

(f) For each $j \in \tilde{\mathcal{J}}$ compute ρ_j given by (12) and delete from $\tilde{\mathcal{J}}$ all but the most profitable indices.

(g) Determine the new indices $\tilde{\mathcal{I}}$ and update the QR decomposition using (17). Then solve the new least squares problem, compute the new residual $r = Am_k - e_k$, and set $\mathcal{I} = \mathcal{I} \cup \tilde{\mathcal{I}}$ and $\mathcal{J} = \mathcal{J} \cup \tilde{\mathcal{J}}$.

Implemented Versions of SPAI

In the folder **/src/objective5**

1. **normal** version without parallel - `spai.hpp` `test_spai.cpp`

2. **openmp** version parallelized by OpenMP based on normal version - `spai_openmp.hpp` `test_openmp.cpp`

3. **mpi** version parallelized by MPI based on normal version - `spai_mpi.hpp` `test_mpi.cpp`

4. use `**MatrixWithVecSupport**` class as the base matrix (**comes from Mattteochen's work: https://github.com/AMSC22-23/matvetcg-chen-zhang-guo/blob/m5/src/shared/MatrixWithVecSupport.hpp**) based on normal version instead of sparse matrix of the Eigen library in the above work - `spai_mbase.hpp` `test_mbase.cpp`

Code Structure

Basically, for the application of the four SPAI algorithms, the process follows:

the test_*.cpp file reads one mtx file, stores it in matrix A, then calls the SPAI function in the corresponding spai_*.hpp file to obtain the sparse approximate inverse M. After that, both sides of the equation $Ax = b$ are multiplied by matrix M. Next, an iterative algorithm (CG and BiCGSTAB) is used to compute the equation.

Code Structure

`spai.hpp`

Main process

```
template <class Matrix, typename Scalar>
int SPAI(const Matrix &A, Matrix &M, const int max_iter, Scalar epsilon) {
```

Extract repeated work from the main process(Code contains detailed explanation.)

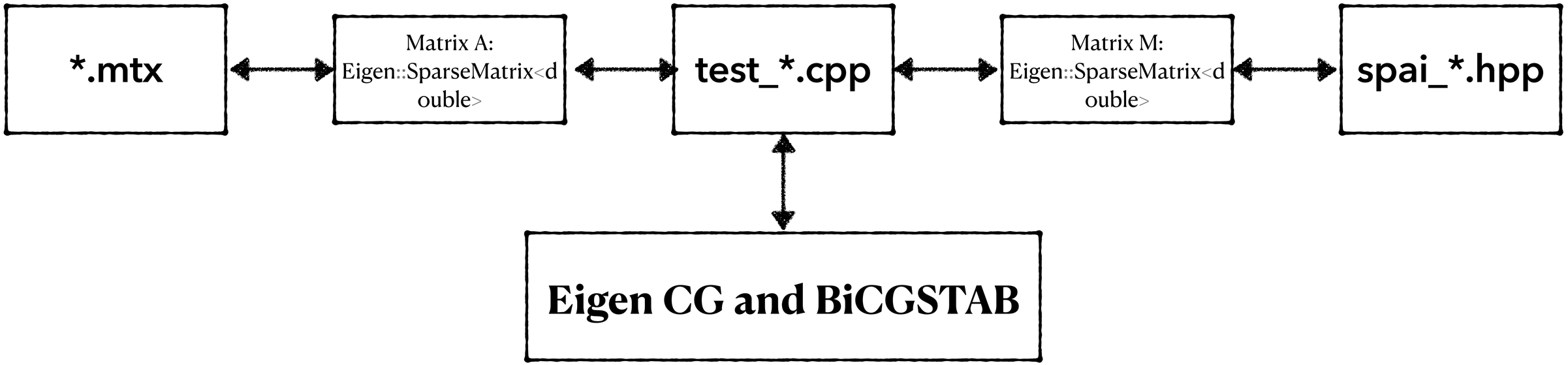
```
template <class Matrix, typename Scalar>
void computeRAndMk(const Matrix&A, const int Size, const Eigen::MatrixXd &Q,
    const Eigen::MatrixXd &R, Eigen::VectorXd &r, Eigen::VectorXd &m_k,
    const Eigen::VectorXd &e_k, const std::vector<int> &I, const std::vector<int> &J, int
```

```
void qrDecomposition(const Eigen::MatrixXd &A, Eigen::MatrixXd &Q, Eigen::MatrixXd &R) {
```

```
template <class Matrix>
void getBlockByTwoIndicesSet(const Matrix&A, Eigen::MatrixXd &AIJ,
    const std::vector<int> &I, const std::vector<int> &J) {
```

Code Structure

For **normal** version/**openmp** version/**mpi** version, matrix A and M is an instance of Eigen::SparseMatrix<double> class.



(Code Optimisation: use smart pointers **unique_ptr** to enforce mandatory memory release for Eigen instances.)

Parallelization Strategy

For **openmp** version, we optimise the main process by OPENMP because SPAI is inherently parallel and the columns of M are calculated independently of one another.

`spai_openmp.hpp`

```
// parallel
#pragma omp parallel for

// for every column of M
for (int k=0; k<Size; k++) {
```

Parallelization Strategy

For **mpi** version, we optimise the main process by MPI.

At the beginning...

```
// number of cols to be processed in this thread
std::vector<int> count_recv(mpi_size);
std::vector<int> displacements(mpi_size);
if (mpi_rank == 0) {
    int chunk = Size / mpi_size;
    int rest = Size % mpi_size;
    for (auto i = 0; i < mpi_size; i++) {
        count_recv[i] = i<rest? (chunk+1)*Size: chunk*Size;
    }
    displacements[0]=0;
    for (auto i=1;i<mpi_size;++i)
        displacements[i]=displacements[i-1]+count_recv[i-1];
}
```

```
// for every thread
int local_chunk;
MPI_Scatter(count_recv.data(),1,MPI_INT, &local_chunk,1,MPI_INT,0,mpi_comm);
local_chunk = local_chunk / Size;
int location_of_start;
MPI_Scatter(displacements.data(),1,MPI_INT, &location_of_start,1,MPI_INT,0,mpi_comm);
location_of_start = location_of_start / Size;
// std::cout << "Hello, this is thread " << mpi_rank << ", I am processing " << local_
std::vector<double> local_m(local_chunk*Size);
MPI_Scatterv(MM.data(), count_recv.data(), displacements.data(), MPI_DOUBLE,
        local_m.data(), local_chunk*Size, MPI_DOUBLE, 0, mpi_comm);
// std::cout << "Hello, this is thread " << mpi_rank
//           << ", I received partial data:\n";
// for (auto i:local_m) {std::cout<<i<<" ";}

// for every column of M
for (int k = location_of_start; k < location_of_start+local_chunk; k++) {
```

Parallelization Strategy

At the end...

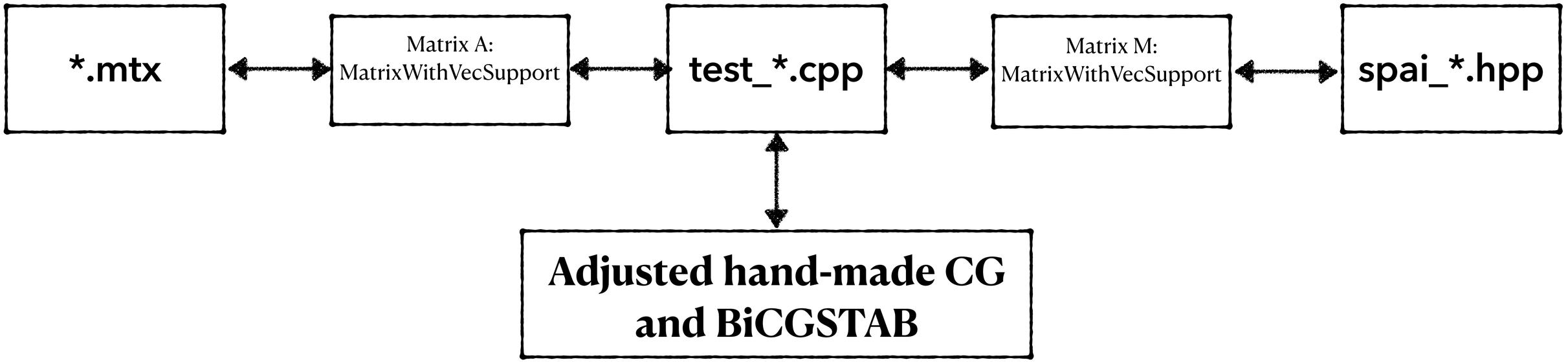
```
    // copy m_k to world_result
    kk = k - location_of_start;
    for(int i=0; i<Size; i++) {
        auto value = (*m_k_ptr)(i);
        if(std::isnan(value)) { local_m[kk*Size+i]=0; }
        else{ local_m[kk*Size+i]=value; }
    }
}

// std::cout << "thread " << mpi_rank << " completes for loop and gather is beginning" << std:
MPI_Barrier(mpi_comm);
MPI_Gatherv(local_m.data(), local_chunk*Size, MPI_DOUBLE, MM.data(),
            count_recv.data(), displacements.data(), MPI_DOUBLE, 0, mpi_comm);

if (mpi_rank == 0) {
    for (int i = 0; i < Size; i++) {
        for (int j = 0; j < Size; j++) {
            M.insert(i, j) = MM[j*Size+i];
        }
    }
    M.makeCompressed();
}
```

Code Structure

For the fourth version, matrix A and M is an instance of MatrixWithVecSupport class.

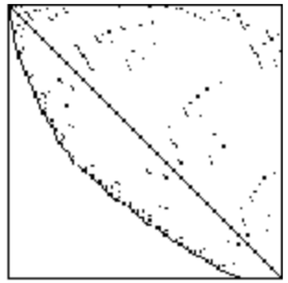


Numerical Experiment

Default setting: 4 threads for openmp and mpi threads setting

gre_115.mtx: n = 115 and nz = 421 entries

Structure Plot



Initial Settings

For iterative solvers:

like Eigen CG/CG and Eigen BiCGSTAB/BiCGSTAB, the initial guess is a zero vector, the stopping criterion is 1e-8, the maximum iteration is 1000, b=A*e (e is a 1-vector).

Numerical Experiment

Convergence results: max_iter =10 , epsilon = 0.6

	Spai	Spai with OpenMP	Spai with MPI	No preconditioner
Eigen CG/CG	>1000	>1000	>1000	>1000
Eigen BiCGSTAB/ BiCGSTAB	21	21	21	698

Numerical Experiment

Execution time(microseconds) for getting M: max_iter =10 , epsilon = 0.6

	Spai	Spai with OpenMP	Spai with MPI
	887813	800305	722501

Conclusion

1. The SPAI algorithm is very effective in improving the convergence of iterative solvers.
2. Parallelization strategies can decrease execution time.