Final Report: Fast solver and H-matrix inversion

Chao Chen
ICME, Stanford University

April 18, 2015

1 Introduction

This report summaries my class project for CME 335 (ATiNLA) at Stanford. A fast solver for hierarchical matrices under weak admissibility has been implemented. Assuming the off-diagonal rank is constant, the algorithm scales as $O(n \log^2(n))$. The solver can be used either as a direct solver with very accurate low rank approximation or as a preconditioner for iterative methods. In practice, the later seems to be a more efficient choice. Implementing this solver is a crucial step towards computing the inverse of weak-admissible matrices, which can be used as the initial guess for Newton-Schulz iteration to compute the inverse of standard-admissible h-matrices.

The code is written in c++ and can be found at my githup H-Matrix-Inverse repository. The HMat class takes a finite difference matrix from 2D problem, stores it in sixteen-nary tree structure (every node has at most 16 children) and solves the linear system. The implementation uses Eigen linear algebra library. When I first started, I have referred to the following two codes for the hierarchical tree structure:DMHM and HMat.jl. In the rest of the report, the fast solver algorithm is presented in section 2, with complexity analysis and performance statistics. In section 3, a discussion on using the fast solver as preconditioner is presented. Section 5 has the conclusion with the future plan.

2 Fast Solver Algorithm

In this section, we present the details of the fast solver for solving 2D finite difference matrices. The speed of this algorithm depends on the rank of the off-diagonal blocks. In the case of 2D five-point finite difference matrix, the off-diagonal blocks are exactly rank $\sqrt{n}$ where n is the block size, so the solver actually scales as $O(n^3)$. However, if the rank if constant, i.e. does not depend on the matrix size, we will show the complexity is $O(r^2\, n \log^2(n))$. So the solver is preferred to be used as preconditioner for iterative methods, which will be discussed in the section 3 section.

2.1 Algorithm

First, we review the algorithm to solve a 2x2 block matrix where the off-diagonal blocks are low rank. The details can be found in [cite Amir]. Given such a matrix A, we can factorize it by pulling

out the diagonal blocks as follows.

$$A = \begin{pmatrix} D_0 & U_0 V_1^T \\ U_1 V_0^T & D_1 \end{pmatrix} = \begin{pmatrix} D_0 & \\ & D_1 \end{pmatrix} \begin{pmatrix} I & u_0 V_1^T \\ u_1 V_0^T & I \end{pmatrix}$$

where $D_0 u_0 = U_0, D_1 u_1 = U_1$. This factorization is useful because the second part can be inverted easily with Woodbury matrix identity.

$$\begin{pmatrix} I & u_0 V_1^T \\ u_1 V_0^T & I \end{pmatrix}^{-1} = I - \begin{pmatrix} u_0 & \\ & u_1 \end{pmatrix} \begin{pmatrix} I & V_1^T u_1 \\ V_0^T u_0 & I \end{pmatrix}^{-1} \begin{pmatrix} & V_1^T \\ V_0^T & \end{pmatrix}$$

Therefore, we have come to a recursive algorithm to solve such linear systems with hierarchical off-diagonal low-rank structure.

Algorithm 1 solve 2×2 block matrix

Problem:

$$\begin{pmatrix} D_0 & U_0 V_1^T \\ U_1 V_0^T & D_1 \end{pmatrix} \begin{pmatrix} x0 \\ x1 \end{pmatrix} = \begin{pmatrix} b0 \\ b1 \end{pmatrix}$$

Algorithm:

(1) **Recursively** solve: $D_0[d_0, u_0] = [b_0, U_0]$ and $D_0[d_1, u_1] = [b_1, U_1]$

(2) Solve a **small** linear system:

$$\begin{pmatrix} I & V_1^T u_1 \\ V_0^T u_0 & I \end{pmatrix} \begin{pmatrix} \eta_0 \\ \eta_1 \end{pmatrix} = \begin{pmatrix} V_1^T d_1 \\ V_0^T d_0 \end{pmatrix}$$

(3) **Assemble** the solution of the original problem:

$$\begin{pmatrix} x_0 \\ x_1 \end{pmatrix} = \begin{pmatrix} d_0 - u_0 \cdot \eta_0 \\ d_1 - u_1 \cdot \eta_1 \end{pmatrix}$$

Second, for problems in two dimensions, when the problem domain is sub-divided in every dimension i.e. the domain becomes $2^2 = 4$ pieces and correspondingly, the hierarchical matrix becomes 4 by 4 blocks. As a result, the above algorithm can not be used directly. Fortunately the dense blocks only appear on the diagonal block due to weak admissibility. The other $16 - 4 = 12$ off-diagonal blocks are all low rank and thus the matrix can be view as a two level 2x2 block matrix as shown in figure 1.

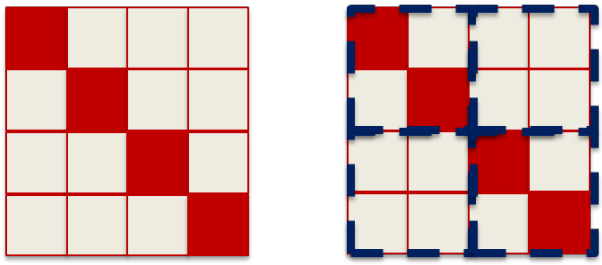

Figure 1: The 4x4 hierarchical matrix can be viewed as a 2x2 block matrix where each diagonal block is again a 2x2 block matrix

Therefore we come to the final algorithm if we apply algorithm 1 multiple times.

Algorithm 2 solve 4 by 4 block matrix

(1) call algorithm 1 for two 2x2 diagonal blocks
(2) compress two off-diagonal 2x2 low rank blocks
(3) call algorithm 1 again to solve the 2x2 block view of the original matrix

Note that algorithm (1) will call algorithm (2) for solving the diagonal hierarchical blocks, so what happens here is a zig-zag order of function call, as shown in figure 2.

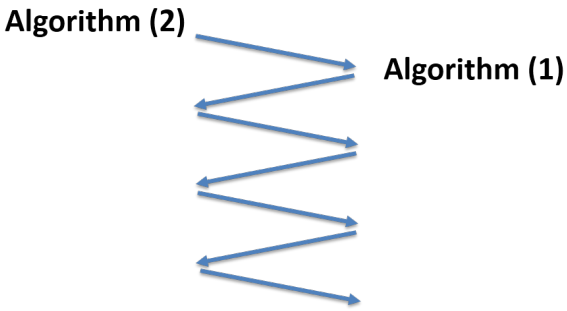

Figure 2: Zig-zag function call

At last, we want to address step (2) in algorithm 2 i.e. how to find the low rank representation for a 2x2 low rank blocks. Observe the following equality

$$\begin{pmatrix} U_0V_0^T & U_1V_1^T \\ U_2V_2^T & U_3V_3^T \end{pmatrix} = \begin{pmatrix} U_0 & & U_1 & \\ & U_2 & & U_3 \end{pmatrix} \begin{pmatrix} V_0^T & & & \\ V_2^T & & & \\ & & V_1^T & \\ & & V_3^T & \end{pmatrix} = \begin{pmatrix} U_0 & U_1 & & \\ & & U_2 & U_3 \end{pmatrix} \begin{pmatrix} V_0^T & & V_1^T & \\ & & & \\ V_2^T & & & \\ & & V_3^T & \end{pmatrix} \equiv UV^T$$

Note that all the matrices above are in fact skinny matrices, so the shape shown above is not realistic. The the big U matrix should be skinny and the big V^T matrix should be fat. Apparently, this is not an optimal representation without further compressing $[U_0, U_1]$, $[U_2, U_3]$, $[V_0^T; V_1^T]$ and $[V_1^T; V_3^T]$. But in the case of five-point finite difference matrices, U_1, U_2, V_1, V_2 are all zero matrices because they correspond to the interaction of two grid blocks sitting diagonally to each other. So the above representation is optimal for five-point finite difference matrices.

2.2 Complexity Analysis

The complexity of the algorithm depends on the rank of the off-diagonal blocks. A simple case would be that the rank is fixed for off-diagonal blocks, then the complexity is $O(r^2\, n \log^2(n))$, where r is the rank. The analysis goes as follows. Let n be the matrix size and p be the number of right hand size to be solved, then algorithm 1 shows that

$$\begin{aligned} f(n,p) &= 2f(n/2, p+r) + r^2 n + 2rnp \\ &= 2f(n/2, p+r) + rn(r+2p) \end{aligned}$$

Since f is linear in the second parameter p, i.e. $f(n,p) = p \cdot f(n,1)$. we will consider solving for a single right hand size below

$$\begin{aligned} f(n,1) &= 2f(n/2, 1+r) + rn(r+2) \\ &\approx r^2 n + 2[2f(n/4, 1+2r) + r \cdot 3rn/2] \\ &= r^2 n(1+3) + 4f(n/4, 1+2r) \\ &= \ldots \\ &= r^2 n(1+3+\ldots+(2\cdot L+1)) + 2^L f(n/(2^L), 1+Lr) \end{aligned}$$

where the summation is the dominant term and is approximately $r^2 n \log^2(n)$. So

$$f(n,p) \approx p \cdot r^2 n \log^2(n)$$

In the ideal case where the rank r is constant, the solver scales super-linearly. However, for the case of five-point finite difference matrix, the rank of the off-diagonal block of the first level is $\sqrt{n/4}$ and thus the complexity increases dramatically to $O(n^2 \log^2(n))$.

2.3 Performance Results

Here shows the timing results for using the method as direct solver, meaning that the result is exact (up to round-off error). Again the test matrix is five-point finite difference matrix. Since the

rank of the off-diagonal blocks are $\sqrt{a}$ where a is the size of the block, the solver should scale as $O(n^2)$ where n is the total number of grid points. All experiments are done on a workstation with the following processor

```
Intel(R) Xeon(R) CPU:  5606  @ 2.13GHz
cache size: 8192 KB
gcc version: 4.8.2
```

Also the code is compiled with -O2 flag and every single timing result shown here is the average of five runs. The first table shows the speed of the fast solver versus the build-in Eigen::lu() routine.

Grid ($n \times n$)	16	32	64	128
Fast Solver	0.00063	0.0067	0.077	0.99
Eigen::LU	0.0021	0.098	4.9	292

Table 1: Timing results: fast solver v.s. Eigen::lu()

If the rank is fixed, the super-linear scalability as below. Since the off-diagonal blocks are exactly low rank, the residual becomes $O(1)$ once we reduce the rank under $\sqrt{n/2}$. However, it will be very useful if the solver is used as preconditioner for iterative methods, which is demonstrated in the next section.

Grid ($n \times n$)	16	32	64	128
Time(s)	0.00063	0.0042	0.026	0.15

Table 2: Fix rank=8

At last, the problem size is fixed to be 128×128 2d grid points. The timing corresponding to different the ranks is shown as below

rank	64	32	16	8
Time(s)	0.99	0.59	0.31	0.15

Table 3: Fix problem size: 128×128 2d grid

The largest problem size this current code can handle is 128×128 2d grid, and there are two limitations: computing the low rank factorization and applying the z-order permutation. Currently, the Eigen::JacobiSVD() method is used to compute the low rank factorization of the off-diagonal blocks, which is very expensive. The plan is to use the randomized SVD algorithm [cite martinson]. The other place, the most expensive part of the code actually has nothing to do with the solve. It lies in applying the z-order permutation matrix. Now the five-point finite difference matrix is stored in dense matrix, so applying the permutation becomes extremely expensive. The plan is to switch to sparse matrix representation, so it is able to start the solver.

3 Preconditioned Conjugate Gradient

Since the five-point finite difference matrix is symmetric positive definite, the conjugate gradient method is the ideal iterative solver. In this section, a discuss on how to combine the fast solver with the conjugate gradient method is presented. For a general problem $Ax = b$, there are three ways to use a preconditioner $M \approx A$: left preconditioning

$$M^{-1}Ax = M^{-1}b$$

right preconditioning

$$AM^{-1}x = bM^{-1}$$

and split preconditioning

$$L^{-1}AR^{-1}y = L^{-1}b$$

where $M = LR$. For the five-point finite difference matrix A, it is symmetric positive definite and it seems left preconditioning and right preconditioning would destroy the symmetry, whereas split preconditioning requires the factorization of M, which is not available. Actually all three different formulations are equivalent in this case and the algorithm is derived with the performance result at the end.

First of all, the original well-known conjugate gradient method is as follows

Algorithm 3 CG

1. Compute $r - 0 = b - Ax_0, p_0 = r_0$
2. For $j = 0, 1, ...$ until convergence
3. $\quad \alpha = r_j^T r_j / p_j^T A p_j$
4. $\quad x_{j+1} = x_j + \alpha p_j$
5. $\quad r_{j+1} = r_j - \alpha A p_j$
6. $\quad \beta = r_{j+1}^T r_{j+1} / r_j^T r_j$
7. $\quad p_{j+1} = r_{j+1} + \beta p_j$
8. EndFor

Suppose we have the cholesky factorization $M = LL^T$, the split preconditioned system would be

$$L^{-1}AL^{-T}y = L^{-1}b$$

where $y = L^T x$ and using algorithm 3 yields the following procedure

almost every character here has a hat, meaning that they are not the same value as in algorithm 3. The difficulty to use algorithm 4 directly is that it requires the explicit cholesky decomposition of the preconditioner, but it is not necessary and the hat values are closely related to the unhat values. Observe that

$$\hat{r}_i = L^{-1}r_i$$
$$\hat{p}_i = L^{-1}p_i$$
$$\hat{x}_j = L^T x_j$$

Algorithm 4 split preconditioned CG

1. Compute $\hat{r}_0 = L^{-1}b - L^{-1}Ax_0, \hat{p}_0 = \hat{r}_0$
2. For $j = 0, 1, ...$ until convergence
3. $\quad \hat{\alpha} = \hat{r}_j^T \hat{r}_j / \hat{p}_j^T L^{-1} A L^{-T} \hat{p}_j$
4. $\quad \hat{x}_{j+1} = \hat{x}_j + \hat{\alpha}\hat{p}_j$
5. $\quad \hat{r}_{j+1} = \hat{r}_j - \hat{\alpha} L^{-1} A L^{-T} \hat{p}_j$
6. $\quad \hat{\beta} = \hat{r}_{j+1}^T \hat{r}_{j+1} / \hat{r}_j^T \hat{r}_j$
7. $\quad \hat{p}_{j+1} = \hat{r}_{j+1} + \hat{\beta}\hat{p}_j$
8. EndFor

and introduce some new variables

$$\begin{aligned}
\hat{r}_j &= L^{-1} r_j \\
\tilde{p}_j &= L^{-T} \hat{p}_j = M^{-1} p_j \\
z_j &= L^{-T} \hat{r}_j = M^{-1} r_j \\
\tilde{x}_j &= L^{-T} \hat{x}_j = x_j
\end{aligned}$$

the algorithm becomes

Algorithm 5 preconditioned CG

1. Compute $r_0 = b - Ax_0, z_0 = M^{-1} r_0, \tilde{p}_0 = z_0$
2. For $j = 0, 1, ...$ until convergence
3. $\quad \tilde{\alpha} = r_j^T z_j / \tilde{p}_j^T A \tilde{p}_j = \hat{\alpha}$
4. $\quad \tilde{x}_{j+1} = \tilde{x}_j + \tilde{\alpha}\tilde{p}_j \quad$ i.e. $x_{j+1} = x_j + \tilde{\alpha}\tilde{p}_j$
5.1 $\quad r_{j+1} = r_j - \hat{\alpha} A \tilde{p}_j$
5.2 $\quad z_{j+1} = M^{-1} r_{j+1}$
6. $\quad \tilde{\beta} = r_{j+1}^T z_{j+1} / r_j^T z_j = \hat{\beta}$
7. $\quad \tilde{p}_{j+1} = z_{j+1} + \tilde{\beta}\tilde{p}_j$
8. EndFor

or the cleaner version as follows

Algorithm 6 preconditioned CG: clean version

1. Compute $r_0 = b - Ax_0, z_0 = M^{-1} r_0, \tilde{p}_0 = z_0$
2. For $j = 0, 1, ...$ until convergence
3. $\quad \tilde{\alpha} = r_j^T z_j / \tilde{p}_j^T A \tilde{p}_j$
4. $\quad x_{j+1} = x_j + \tilde{\alpha}\tilde{p}_j$
5.1 $\quad r_{j+1} = r_j - \hat{\alpha} A \tilde{p}_j$
5.2 $\quad z_{j+1} = M^{-1} r_{j+1}$
6. $\quad \tilde{\beta} = r_{j+1}^T z_{j+1} / r_j^T z_j$
7. $\quad \tilde{p}_{j+1} = z_{j+1} + \tilde{\beta}\tilde{p}_j$
8. EndFor

Below is shown the results of using the fast solver with conjugate gradient method on 128×128 grid. The original method without preconditioning takes 493 iterations and 108 seconds to converge.

Rank	64	60	50	40	30	20	10
iter #	1	21	40	51	65	86	112
Time(s)	1.2	24.1	40.4	53.4	50.1	50.3	45.4

Table 4: Iteration number and timing using different ranks for CG

For the five-point finite difference matrix, the off-diagonal blocks are exactly low rank, so the fast solver with the exact rank performs the best. Finally, I have also implemented fixed-point iteration with the fast solver as preconditioner and the same effect is seen below. The problem is 64×64 grid (128 takes too long). The original Gauss-Seidel iteration takes iterations and seconds to converge.

Rank	32	30	20	10
iter #	1	117	368	741
Time(s)	0.087	9.78	24.2	33.1

Table 5: Iteration number and timing using different ranks for GS

4 Conclusion and Future Work

I have implemented a fast solver for weak-admissible matrices, which can be used either as direct solver or preconditioner for iterative solvers. It is also crucial to invert a weak-admissible matrix which may be a good initial guess for the New-Schultz iteration. One of the things on the todo list is to extend the current solver to 3D. The plan is to use C++ templates, so there will be a single class for both 2D and 3D problems. The next step towards inverting weak-admissible matrices is clear following algorithm below. The next step is to address step (4) in the algorithm i.e. updating a hierarchical matrix with a low rank matrix.

Algorithm 7 Inverse of H-matrix

Problem:

$$\begin{aligned} A^{-1} &= \begin{pmatrix} I & \bar{U}_0 V_1^T \\ \bar{U}_1 V_0^T & I \end{pmatrix}^{-1} \begin{pmatrix} D_0^{-1} & \\ & D_1^{-1} \end{pmatrix} \\ &= \begin{pmatrix} D_0^{-1} - \bar{U}_0 B_{00} V_1^T D_1^{-1} & -\bar{U}_0 B_{01} V_0^T D_0^{-1} \\ -\bar{U}_1 B_{10} V_1^T D_1^{-1} & D_1^{-1} - \bar{U}_1 B_{11} V_0^T D_0^{-1} \end{pmatrix} \end{aligned}$$

where

$$\begin{pmatrix} I & V_1^T \bar{U}_1 \\ V_0^T \bar{U}_0 & I \end{pmatrix}^{-1} = \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

Proposed algorithm:
(1) Recursively compute D_0^{-1} and D_1^{-1}
(2) Compute matrix-matrix product $\bar{U}_0 = D_0^{-1} U_0$, $\bar{U}_1 = D_1^{-1} U_1$, $V_0^T D_0^{-1}$ and $V_1^T D_1^{-1}$
(3) Compute the inverse of a small matrix, i.e. $B_{00}, B_{01}, B_{10}, B_{11}$
(4) Update D_0^{-1} and D_1^{-1} by $-\bar{U}_0 B_{00} V_1^T D_1^{-1}$ and $-\bar{U}_1 B_{11} V_0^T D_0^{-1}$ respectively