2008

Construction and application of hierarchical matrix preconditioners

Fang Yang
University of Iowa

CONSTRUCTION AND APPLICATION OF HIERARCHICAL MATRIX PRECONDITIONERS

by

Fang Yang

An Abstract

Of a thesis submitted in partial fulfillment of the
requirements for the Doctor of Philosophy
degree in Computer Science
in the Graduate College of
The University of Iowa

December 2008

Thesis Supervisor: Professor Suely Oliveira

ABSTRACT

$\mathcal{H}$-matrix techniques use a data-sparse tree structure to represent a dense or a sparse matrix. The leaves of the tree store matrix sub-blocks that are represented in full-matrix format or Rk-matrix (low rank matrix) format. $\mathcal{H}$-matrix arithmetic is defined over the $\mathcal{H}$-matrix representation, which includes operations such as addition, multiplication, inversion, and LU factorization. These $\mathcal{H}$-matrix operations produce approximate results with almost optimal computational complexity.

Based on the properties of $\mathcal{H}$-matrices, the $\mathcal{H}$-matrix preconditioner technique has been introduced. It uses $\mathcal{H}$-matrix operations to construct preconditioners, which are used in iterative methods to speed up the solution of large systems of linear equations ($Ax = b$). To apply the $\mathcal{H}$-matrix preconditioner technique, the first step is to represent a problem in $\mathcal{H}$-matrix format. The approaches to construct an $\mathcal{H}$-matrix can be divided into two categories: geometric approaches and algebraic approaches.

In this thesis, we present our contributions to algebraic $\mathcal{H}$-matrix construction approaches and $\mathcal{H}$-matrix preconditioner technique. We have developed a new algebraic $\mathcal{H}$-matrix construction approach based on matrix graphs and multilevel graph clustering approaches. Based on the new construction approach, we have also developed a scheme to build algebraic $\mathcal{H}$-matrix preconditioners for systems of saddle point type. To verify the effectiveness of our new construction approach and $\mathcal{H}$-matrix preconditioner scheme, we have applied them to solve various systems of linear equations arising from finite element methods and meshfree methods. The experimental results

show that our preconditioners are competitive to other $\mathcal{H}$-matrix preconditioners based on domain decomposition and existing preconditioners such as JOR and AMG preconditioners. Our $\mathcal{H}$-matrix construction approach and preconditioner technique provide an alternative effective way to solve large systems of linear equations.

Abstract Approved: _____
Thesis Supervisor

_____
Title and Department

_____
Date

CONSTRUCTION AND APPLICATION OF HIERARCHICAL MATRIX PRECONDITIONERS

by

Fang Yang

A thesis submitted in partial fulfillment of the
requirements for the Doctor of Philosophy
degree in Computer Science
in the Graduate College of
The University of Iowa

December 2008

Thesis Supervisor: Professor Suely Oliveira

Graduate College
The University of Iowa
Iowa City, Iowa

CERTIFICATE OF APPROVAL

PH.D. THESIS

This is to certify that the Ph.D. thesis of

Fang Yang

has been approved by the Examining Committee for the thesis requirement for the Doctor of Philosophy degree in Computer Science at the December 2008 graduation.

Thesis Committee: _____
Suely Oliveira, Thesis Supervisor

_____
Weimin Han

_____
Alberto Segre

_____
David Stewart

_____
Chris Wyman

ACKNOWLEDGEMENTS

I am deeply grateful to my advisor Dr. Suely Oliveira for her guidance and encouragement throughout my study as a PhD. student. Without her advice and inspiration during my thesis work, I could not reach this far. I am grateful to Dr. Stewart for the discussions about Meschach library. I also would like to thank Dr. Han, Dr. Segre, and Dr. Wyman for their effort and time as part of my thesis committee.

Last I wish to thank my family and my friends. They make my life more meaningful.

ABSTRACT

$\mathcal{H}$-matrix techniques use a data-sparse tree structure to represent a dense or a sparse matrix. The leaves of the tree store matrix sub-blocks that are represented in full-matrix format or Rk-matrix (low rank matrix) format. $\mathcal{H}$-matrix arithmetic is defined over the $\mathcal{H}$-matrix representation, which includes operations such as addition, multiplication, inversion, and LU factorization. These $\mathcal{H}$-matrix operations produce approximate results with almost optimal computational complexity.

Based on the properties of $\mathcal{H}$-matrices, the $\mathcal{H}$-matrix preconditioner technique has been introduced. It uses $\mathcal{H}$-matrix operations to construct preconditioners, which are used in iterative methods to speed up the solution of large systems of linear equations ($Ax = b$). To apply the $\mathcal{H}$-matrix preconditioner technique, the first step is to represent a problem in $\mathcal{H}$-matrix format. The approaches to construct an $\mathcal{H}$-matrix can be divided into two categories: geometric approaches and algebraic approaches.

In this thesis, we present our contributions to algebraic $\mathcal{H}$-matrix construction approaches and $\mathcal{H}$-matrix preconditioner technique. We have developed a new algebraic $\mathcal{H}$-matrix construction approach based on matrix graphs and multilevel graph clustering approaches. Based on the new construction approach, we have also developed a scheme to build algebraic $\mathcal{H}$-matrix preconditioners for systems of saddle point type. To verify the effectiveness of our new construction approach and $\mathcal{H}$-matrix preconditioner scheme, we have applied them to solve various systems of linear equations arising from finite element methods and meshfree methods. The experimental results

show that our preconditioners are competitive to other $\mathcal{H}$-matrix preconditioners based on domain decomposition and existing preconditioners such as JOR and AMG preconditioners. Our $\mathcal{H}$-matrix construction approach and preconditioner technique provide an alternative effective way to solve large systems of linear equations.

TABLE OF CONTENTS

LIST OF TABLES

Table

LIST OF FIGURES

Figure

CHAPTER 1
INTRODUCTION

1.1 Background

Systems of linear equations arise from discretization of partial differential equations and other problems. To solve these large systems of linear equations

$$Ax = b, \tag{1.1}$$

we can apply iterative methods [35, 60]. These iterative methods start with an approximate solution x_0, and then modify the approximation x_i at each iteration step i until convergence is reached. The iterative methods include classic methods like the Jacobi, Gauss-Seidel, and SOR (Successive Overrelaxation) methods [64], as well as Krylov subspace methods like Conjugate Gradient (CG), Minimal Residual (MINRES), and General Minimal Residual (GMRES) methods [27, 61]. Even though these methods are theoretically founded, they may suffer from slow convergence. To speed up the convergence of iterative methods, preconditioning is a key technique, especially for Krylov subspace methods. The idea of preconditioning technique is that instead of solving the system (1.1) directly, we can solve the following preconditioned system:

$$M^{-1}Ax = M^{-1}b. \tag{1.2}$$

Here the preconditioner M should be a matrix approximating to A, and $Mx = b$ is inexpersive to solve compared to (1.1). There are many existing preconditioning techniques such as incomplete LU factorization [52] and multigrid preconditioners [23, 50].

In this thesis we focus on the hierarchical-matrix ($\mathcal{H}$-matrix in short) preconditioning technique to solve the system (1.1). $\mathcal{H}$-matrix techniques are relatively new, introduced in 1998 by W. Hackbusch [38, 40]. $\mathcal{H}$-matrix techniques were first developed to approximate densely populated matrices arising in boundary element methods or inverses of sparse matrices in finite element methods to solve partial differential equations [4, 10, 49]. To approximate these dense matrices, $\mathcal{H}$-matrix techniques use a data-sparse representation. $\mathcal{H}$-matrix representation is based on a hierarchical tree structure, called block cluster tree $T_{I\times I}$, which describes hierarchical block partitioning of a matrix and store the data. In a block cluster tree $T_{I\times I}$, its root represents the whole matrix; the internal nodes are the matrix sub-blocks that are partitioned further on the next level; the leaves are the matrix blocks that are not partitioned further. These leaf blocks are either low-rank matrices (Rk-matrix) or full matrices. The building blocks of an $\mathcal{H}$-matrix are Rk-matrices. An Rk-matrix uses the product of two full rectangular matrices to represent a low-rank matrix: for a matrix $M_{n\times n}$ with $\text{rank}(M) \leq k$, its Rk-matrix representation is $M = A_{n\times k}B^T_{n\times k}$. If $k << n$, the Rk-matrix representation $A_{n\times k}B^T_{n\times k}$, which has $2nk$ entries, can save data storage. Another advantage of Rk-matrix representation is that it can reduce the computational complexity of matrix operations, like matrix-vector multiplication and matrix-matrix multiplication. Based on $\mathcal{H}$-matrix representation, $\mathcal{H}$-matrix arithmetic has been defined [10, 38, 40]. The operations defined in $\mathcal{H}$-matrix arithmetic includes operations such as $\mathcal{H}$-matrix addition, $\mathcal{H}$-matrix multiplication, $\mathcal{H}$-matrix inversion, and $\mathcal{H}$-matrix LU factorization. These operations take advantage of Rk-

matrix leaves, and use the truncated singular value decomposition to achieve optimal computational complexity of $O(n \log^{\alpha} n)$ for an $n \times n$ system, where $\alpha = 2$ or 3 is a moderate parameter [10, 49].

To use $\mathcal{H}$-matrix techniques, one critical step is to represent a problem in $\mathcal{H}$-matrix format. The approaches to construct $\mathcal{H}$-matrices can be divided into classic approaches (geometric approaches) and algebraic approaches. The classic $\mathcal{H}$-matrix construction approaches use the geometric information underlying a problem such as the domain information and the basis functions used for discretization [38, 49]. In some applications, system matrices are given but geometric information may not be available. In these cases, the classic $\mathcal{H}$-matrix construction approaches can not be applied, but the algebraic $\mathcal{H}$-matrix construction approaches which only relay on matrix information can be used. The algebraic $\mathcal{H}$-matrix construction approach based on domain decomposition is introduced in [34]. We proposed another algebraic construction approach based on multilevel clustering in [55], which will be discussed thoroughly in this thesis.

One point we need to make clear is that $\mathcal{H}$-matrix arithmetic usually produces approximate results. But optimal computational complexity makes it suitable to construct preconditioners for iterative methods. Possible $\mathcal{H}$-matrix preconditioners are: $\mathcal{H}$-matrix inverses, $\mathcal{H}$-matrix-LU factors, and $\mathcal{H}$-matrix Cholesky factors. In comparison, $\mathcal{H}$-matrix-LU and $\mathcal{H}$-matrix Cholesky factors are more attractive than $\mathcal{H}$-matrix inverses, since they are less expensive to compute and provide better convergence rates [16, 34]. The $\mathcal{H}$-matrix preconditioning technique has been shown to

be effective to solve various problems [34, 55].

1.2 Structure of the Thesis

In this thesis, our contributions to $\mathcal{H}$-matrix techniques emphasize on the construction and application of $\mathcal{H}$-matrix preconditioners to solve various problems: first we have developed a new algebraic $\mathcal{H}$-matrix construction approach based on multilevel graph clustering, which is simple to implement yet effective [55]; second, we have expanded our $\mathcal{H}$-matrix construction approach to develop a scheme to construct $\mathcal{H}$-matrix preconditioners for problems of saddle point type; third, we have shown the performance of our preconditioners by applying them to linear systems arising from FEM and meshfree methods.

The thesis is organized as follows. Chapter 2 is an introduction to $\mathcal{H}$-matrices and $\mathcal{H}$-matrix arithmetic, including the concept of $\mathcal{H}$-matrices and the operations defined in $\mathcal{H}$-matrix arithmetic. Chapter 3 is about the new algebraic $\mathcal{H}$-matrix construction approach we have developed, which is based on multilevel graph clustering. In this chapter, the numerical results of applying the new construction approach to solve partial differential equations are also presented. Chapter 4 is about the construction and application of $\mathcal{H}$-matrix preconditioners to solve saddle point systems arising form meshfree methods. Chapter 5 discusses the application of $\mathcal{H}$-matrix preconditioners to solve optimal control and invariant probability distribution problems. Chapter 6 shows the implementation of $\mathcal{H}$-matrices and the algebraic $\mathcal{H}$-matrix construction approach. Chapter 7 is the summary of this thesis.

Short definitions of the keywords used in the thesis are given below:

Binary tree. A tree whose internal nodes have no more than two children.

Index cluster tree, T_I. A tree to describe a hierarchical partitioning of an index set I. The root of T_I is set I.

Block cluster tree, $T_{I\times I}$. A tree to describe a hierarchical partitioning of the product of an index set I. The root of $T_{I\times I}$ is $I \times I$.

Full matrix format. A matrix representation format, where all $m \times n$ entries of a matrix $M_{m\times n}$ are stored, for example, in the row major order.

Rk-matrix, or Rk-matrix representation. A matrix $M_{m\times n}$ with $\text{rank}(M_{m\times n}) \leq k$ and is represented in the matrix product form $M_{m\times n} = A_{m\times k}B^T_{n\times k}$, where A and B are both stored in the full matrix format.

Admissibility condition. A condition used to decide whether a matrix block $t \times s$ shall be approximated by an Rk-matrix.

Hierarchical matrix representation. An hierarchical matrix $\mathcal{H}(T_{I\times I}, k)$, whose admissible subblocks are stored in the Rk-matrix format and inadmissible subblocks are stored in the full matrix format.

$\mathcal{H}$-matrix arithmetic. An group of matrix operations defined over $\mathcal{H}$-matrices.

Matrix graph $G(V, E)$. An indirected graph built based on a symmetric matrix $M_{I\times I}$: the vertex set $G(V)$ is I; there is an edge $e_{i,j} \in G(E)$ between node i and node j if and only if $m_{i,j} \neq 0$; the edge weight $w_{i,j} = |m_{i,j}|$.

Graph coarsening. A process of collapsing together a subset of the vertices of a graph in order to form a relatively coarser graph. Graph coarsening is used in

multilevel methods.

Multilevel method. A method working with a graph at multiple levels of granularity.

Matching of a graph. A set of edges, no two of which are incident on the same vertex.

Edge cut. The number of edges connecting vertices between different partitions.

Heavy edge matching. A heuristic to find a matching with maximum edge weight.

NP-Complete. A set of computational decision problems which is a subset of NP, with the additional property that it is also NP-hard.

Computational complexity. The number of steps an algorithm takes to solve a problem as a function of the size of the input .

CHAPTER 2
INTRODUCTION TO HIERARCHICAL-MATRICES

This chapter introduces hierarchical matrices ($\mathcal{H}$-matrices) and $\mathcal{H}$-matrix arithmetic.

2.1 Overview

To solve partial differential equations, we may apply boundary element methods (BEM) or finite element methods (FEM). BEM's need to solve integral equations. Since the kernel functions used for discretization typically have global support, discretizations of integral equations gives dense matrices. FEM's yield sparse matrices but the inverses of these sparse matrices are usually dense. In scientific computing, researchers try to avoid dealing with dense matrices directly because dense matrices need $O(n^2)$ storage space without compression, and matrix operations involving densely populated matrices are usually quite expensive. For example, for an $n \times n$ dense matrix, the computational complexity of matrix-matrix multiplication may be up to $O(n^3)$. Researchers have long been working on the methods to deal with dense matrices more efficiently. Wavelet approaches [28] use special basis functions in discretization, so that the obtained matrices are sparse. Panel clustering methods [37, 44] and multipole methods [36] were developed in 1980's based on similar ideas. They approximate kernel functions using Taylor expansions so that some sub-matrices can be approximated by low rank matrices. In 1998 W. Hackbusch expanded the idea of panel clustering methods and first proposed what are called hierarchical matrices,

or in short $\mathcal{H}$-matrices [38], since this new format of matrices adapts a hierarchical tree structure. In [38] the definition and approach to construct $\mathcal{H}$-matrices for one dimensional problems were presented and the complexity of $\mathcal{H}$-matrices operations was analyzed for two specific $\mathcal{H}$-matrix tree structures. In general, the $\mathcal{H}$-matrix approach uses a data sparse representation to approximate fully populated matrices, in which certain low rank matrix blocks are represented in the Rk-matrix format. The data sparse representation of $\mathcal{H}$-matrices uses a tree structure, called block cluster tree $T_{I\times I}$, which describes a hierarchical block partitioning of a matrix $M_{I\times I}$: the root of the tree is $I \times I$ representing the whole matrix; an internal node $s \times t \in T_{I\times I}$ represents a matrix block that is partitioned further on the next level; the leaves of $T_{I\times I}$ represent the smallest blocks that are not partitioned further and the leaf blocks are either approximated by Rk-matrices or just represented as full matrices. Following the introduction of $\mathcal{H}$-matrices [38], the second paper [40] shows the approach to construct $\mathcal{H}$-matrices for FEM and BEM in two and three dimensions and it also shows that the orders of complexity of various $\mathcal{H}$-matrix operations is same as those introduced in [38]. In [41], the general approach to construct $\mathcal{H}$-matrices on rectangular or triangular meshes of one, two, and three dimensions is proposed and analyzed.

Following the introduction, a lot of work has been done on the theories and applications of $\mathcal{H}$-matrices.

The paper [49] gives a detailed analysis of the complexity of $\mathcal{H}$-matrix arithmetic. The parallelization of $\mathcal{H}$-matrix construction and $\mathcal{H}$-matrix arithmetic in-

cluding matrix-vector multiplication, matrix multiplication, and matrix inversion is implemented in [48], which reuses most of the sequential algorithms and allows an efficient computation of $\mathcal{H}$-matrix arithmetic on shared memory systems. A variant of $\mathcal{H}$-matrices called $\mathcal{H}^2$-matrices is introduced in [10, 43], which improves the complexity of matrix-vector multiplication to $O(n)$. The paper [12] presents algorithms which compute the best approximation of sum and product of $\mathcal{H}^2$-matrices in linear complexity. The paper [6] proves that the inverses of stiffness matrices arising from finite element discretization of elliptic partial differential equations can be approximated by $\mathcal{H}$- and $\mathcal{H}^2$-matrices. The introduction of a new $\mathcal{H}$-matrix operation called $\mathcal{H}$-LU decomposition is introduced in [51], which produces LU factors in $\mathcal{H}$-matrix format. Compared to $\mathcal{H}$-matrix inverses, $\mathcal{H}$-LU factors can be computed much faster. The parallelization of $\mathcal{H}$-LU on distributed memory systems is implemented in [33].

Besides the application of $\mathcal{H}$-matrices to solve integral equations arising from boundary element methods [5, 10], another main application area of $\mathcal{H}$-matrices is construction of efficient preconditioners for iterative methods as discussed in [11]. Before the introduction of $\mathcal{H}$-LU factorization, $\mathcal{H}$-inverses were used as preconditioners. In [14, 15] $\mathcal{H}$-inverses are used as preconditioners to solve two-dimensional convection-diffusion equations. In [4], approximate $\mathcal{H}$-matrix inverses were used as preconditioners to solve partial differential equations with uniformly elliptic operators. The multilevel construction of hierarchical matrix approximations to inverses of finite element stiffness matrices, which are used as preconditioners, is proposed in [18]. The introduction of $\mathcal{H}$-LU factorization moves $\mathcal{H}$-matrix techniques to a new stage

since $\mathcal{H}$-LU factors are usually cheaper to obtain and also provide better convergence speed than $\mathcal{H}$-inverses. In [14, 19, 21, 34] $\mathcal{H}$-LU factors are used as preconditioners to solve convection-diffusion problems. In [13, 17, 55, 57], $\mathcal{H}$-LU factors are used as preconditioners to solve systems of saddle point type.

The $\mathcal{H}$-matrix approach can also be applied to solve matrix functions as discussed in [10, 11].

2.2 Model Problem

Let's first look at the model problem, which introduces $\mathcal{H}$-matrices. Consider an integral operator of the form:

$$L(x) = \int_{\Omega} g(x,y)u(y)dy, \tag{2.1}$$

on a subdomain $\Omega \subset \mathbb{R}$ with $g : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ as a kernel function. Discretize (2.1) using Galerkin's method with basis functions $\{\varphi_0, \ldots, \varphi_{n-1}\}$, where $\varphi_i : \Omega \to \mathbb{R}$. We can obtain the following discretized form:

$$Lu = f,\ L_{i,j} = \int_{\Omega}\int_{\Omega} \varphi_i(x)g(x,y)\varphi_j(y)dx\,dy,\ i,j \in \{0,\ldots,n-1\}. \tag{2.2}$$

L usually is not a sparse matrix since g is non-local.

In typical applications, the kernel function g is asymptotically smooth [10], that is the singularities only occur on the diagonal of $\Omega \times \Omega$ and g is smooth elsewhere. Then g can be approximated by a degenerate approximate kernel function:

$$\widetilde{g}(x,y) = \sum_{i=1}^{k} g_{1,i}(x)g_{2,i}(y), \tag{2.3}$$

by the truncated Taylor expansion. Replacing g by its degenerated approximation in (2.2), we obtain the following approximation to L:

$$\widetilde{L}_{i,j} = (AB^T)_{i,j}, \tag{2.4}$$

where

$$A_{i,k} = \int_\Omega g_{1,k}(x)\varphi_i(x)dx, \text{ and } B_{j,k} = \int_\Omega g_{2,k}(y)\varphi_j(y)dy. \tag{2.5}$$

Let $s, t \in I = \{0, \ldots, n-1\}$ and their corresponding supports be defined as $\Omega_s = \bigcup_{i\in s} \text{supp}\varphi_i$ and $\Omega_t = \bigcup_{j\in t} \text{supp}\varphi_j$ respectively. The sub-matrix $\widetilde{L}_{s\times t} = A_{s\times k}B^T_{t\times k}$ and the rank of $\widetilde{L}_{s\times t}$ is at most k, where k is a constant independent of s and t. The approximation of the sub-blocks of a full matrix L by low rank matrices in a hierarchical way leads to $\mathcal{H}$-matrix techniques.

2.3 Concept of $\mathcal{H}$-matrices

The key elements of $\mathcal{H}$-matrix techniques are index cluster tree T_I, block cluster tree $T_{I\times J}$, admissibility conditions, and Rk-matrices.

Before we introduce the concept of $\mathcal{H}$-matrices, we define the symbols and notations that are used to describe $\mathcal{H}$-matrices in this thesis. Let $\#s$ denote the number of the elements in a set s and $\mathcal{L}(T)$ denote the set of the leaves of a tree T. $S(i)$ is the set of the children of a given node i in a tree; s^l denotes a node s on the level l in a tree. We assume that the level of the root is 0. If the level of a node i is l, then its children are on the level $l+1$. We let $I = \{0, 1, 2, ..., n-1\}$ be an index set. In this thesis, we assume that an index set I is ordered, which means $I = \{0, 1\}$ and $I = \{1, 0\}$ are two different sets. For geometric (or classic) $\mathcal{H}$-matrix construction

approaches, an index $i \in I$ represents an index set of finite elements or boundary element basis functions $(\phi_i)_{i \in I}$; for algebraic $\mathcal{H}$-matrix construction approaches, I represents a row or column index set of a matrix.

2.3.1 Index Cluster Tree T_I

An index cluster tree T_I describes a hierarchical partitioning over an index set $I = \{0, \ldots, n-1\}$. A tree T_I is called index cluster tree if and only if it satisfies the following properties:

1. The root of T_I is the index set I;
2. Each node $s^l \in T_I$ is either a leaf, or an internal node with children $S(s^l)$;
3. The children of the same parent are pairwise disjoint, that is $\forall j_1^{l+1}$, $j_2^{l+1} \in S(i^l)$ and $j_1^{l+1} \neq j_2^{l+1}$, then $j_1^{l+1} \cap j_2^{l+1} = \emptyset$.
4. The parent node $s^l = \bigcup_{t^{l+1} \in S(s^l)} t^{l+1}$.
5. $\mathcal{L}(T)$ forms a partition of I.

If each internal node in T_I has exactly two children, then T_I is a binary tree. In general, the number of children of each internal node is not necessarily to be 2. A T_I can be constructed top-down: starting from the root, recursively split each set into subsets and make subsets as the children. Or it can be built bottom-up: starting from the leaves (smallest sets), recursively build clusters over the sets on the same level and make each cluster the parent to the sets in the cluster until the cluster equals I, which is the root of the tree.

2.3.2 Block Cluster Tree $T_{I\times I}$

A block cluster tree $T_{I\times I}$ describes an hierarchical partitioning over the cartesian product of an index set I. If I is the row and column index set of a matrix $M_{I\times I}$, then $T_{I\times I}$ describes a hierarchical block partitioning over $M_{I\times I}$. Given an index cluster tree T_I, a block cluster tree $T_{I\times I}$ is related to the cartesian product of T_I. An admissibility condition (which is defined later) is used to determine if a node $s \times t \in T_{I\times I}$ is an Rk-matrix leaf or not. The algorithm to construct a block cluster tree $T_{I\times J}$ is described follows:

1. The root of $T_{I\times I}$ is $I \times I$.

2. If a node $s^l \times t^l \in T_{I\times I}$ satisfies the given admissibility condition, then $s^l \times t^l$ is an Rk-matrix leaf on level l.

3. If $s^l \times t^l$ does not satisfy the given admissibility condition, but $\#s^l \leq N_s$ or $\#t^l \leq N_s$, then $s^l \times t^l$ is a full-matrix leaf.

4. If $s^l \times t^l$ is an internal node then its children on level $l+1$ are defined as: $S(s^l \times t^l) = \{\, i^{l+1} \times j^{l+1} \mid i^{l+1} \in S(s^l) \text{ and } j^{l+1} \in S(t^l)\}$.

5. Repeat step 2, 3 and 4, until each leaf $s \times t$ either satisfies the admissibility condition, or $\#s \leq N_s$ or $\#t \leq N_s$.

Here N_s is a constant to control the size of the smallest leaves in order to maintain the efficiency of $\mathcal{H}$-matrix arithmetic. Usually it is set to $N_s \in [10, 100]$.

2.3.3 Rk-matrix

Rk-matrices (low rank matrices) are the basic building blocks of $\mathcal{H}$-matrices. An $m \times n$ matrix $M_{m\times n}$ is called Rk-matrix if $rank(M_{m\times n}) \leq k$ and it is represented in the form of matrix product:

$$M_{m\times n} = A_{m\times k} B^T_{n\times k}, \tag{2.6}$$

with A and B in full-matrix format.

The storage for a matrix $M_{m\times n}$ in full-matrix format is $m \times n$, but the storage in Rk-matrix format is $k(m + n)$. The computational complexity of matrix-vector multiplication in full matrix format is $(2nm - n)$, but if a matrix is an Rk-matrix then the complexity is reduced to $(2k(n + m) - n - k)$. So if k is much smaller than m and n, by representing a matrix in Rk-matrix format we can reduce its storage required and computational complexity significantly.

2.3.4 Admissibility Condition

Admissibility conditions are used to determine whether to approximate a matrix block $s \times t$ by an Rk-matrix during the construction of $\mathcal{H}$-matrices. If a block $s \times t$ is admissible, then it will be approximated by an Rk-matrix. The admissibility conditions vary for different $\mathcal{H}$-matrix construction approaches.

In classic $\mathcal{H}$-matrix construction approaches, admissibility conditions are defined using the geometric information underlying a problem such as the domain information and the supports of index clusters. Given T_I, T_J, and $s \times t \in T_{I\times J}$, the general form of an admissibility condition in classic $\mathcal{H}$-matrix construction approaches can

defined as:

$$s \times t \text{ is admissible } \iff \min\{\text{diam}(\Omega_s), \text{diam}(\Omega_t)\} \leq \mu \text{ dist}(\Omega_s, \Omega_t), \tag{2.7}$$

where $\mu \in \mathbb{R}$ is a parameter to control the number of R-kmatrix blocks. Ω_s and Ω_t are the supports of the cluster s and t respectively. The diameter of a cluster diam(s) and the distance between two clusters dist(s, t) are defined using Euclidean norm as follows:

$$\begin{aligned} \text{diam}(s) &= \max\{\|x_i - x_j\| : x_i, x_j \in \Omega_s\}, \\ \text{dist}(s,t) &= \min\{\|x_i - x_j\| : x_i \in \Omega_s \text{ and } x_j \in \Omega_t\}. \end{aligned} \tag{2.8}$$

In practice boxes B_s (see Section 2.4) are used to replace cluster supports Ω_s in the admissibility condition (2.7) in order to reduce the computational complexity:

$$s \times t \text{ is admissible } \iff \min\{\text{diam}(B_s), \text{diam}(B_t)\} \leq \mu \text{ dist}(B_s, B_t). \tag{2.9}$$

Equation (2.9) is called the standard admissibility condition [10, 49] for geometric $\mathcal{H}$-matrix construction approaches. If minimum in (2.9) is replaced by maximum, we get a strong admissibility condition [49]:

$$s \times t \text{ is admissible } \iff \max\{\text{diam}(B_s), \text{diam}(B_t)\} \leq \mu \text{ dist}(B_s, B_t). \tag{2.10}$$

(2.10) is the admissibility condition used in $\mathcal{H}^2$-matrix construction approaches. The weak admissibility condition [42] is given as follows:

$$s \times t \text{ is admissible } \iff s \neq t, \tag{2.11}$$

which generates a coarser cluster tree compared to the standard admissibility condition, since all the off diagonal blocks are admissible by the weak admissibility condition and will be approximated using Rk-matrices. The definition of weak admissibility condition is independent of basis functions. The admissibility condition for the $\mathcal{H}$-matrix construction approach based on domain decomposition [21] is:

$$\begin{aligned} s \times t \text{ is admissible } \Longleftrightarrow & s \neq t \text{ are domain clusters, or} \\ & s \times t \text{ is admissible by the strong admissibility condition.} \end{aligned} \tag{2.12}$$

The admissibility condition for algebraic $\mathcal{H}$-matrix construction approaches is simply defined using the information contained in a matrix graph instead of the geometric information underlying a problem [55]:

$$s \times t \text{ is admissible } \Longleftrightarrow s \text{ and } t \text{ are not connected in the matrix graph.} \tag{2.13}$$

2.3.5 Definition of $\mathcal{H}$-matrices

Now given a block cluster tree $T_{I\times I}$, the minimum block size N_s, the admissibility condition, and the rank k, a set of $\mathcal{H}$-matrices can be defined as:

$$\mathcal{H}(T,k) = \{M_{I\times I} : \forall s \times t \in \mathcal{L}(T_{I\times I}), s \times t \text{ is admissible or } \min\{\#s, \#t\} \leq N_s\}, \tag{2.14}$$

where the admissible leaf blocks are represented as Rk-matrices and the non-admissible leaf blocks are represented as full-matrices.

Figure 2.1 shows an example of T_I, $T_{I\times I}$, and the corresponding $\mathcal{H}$-matrix. In this example each internal node in T_I has exactly two children.

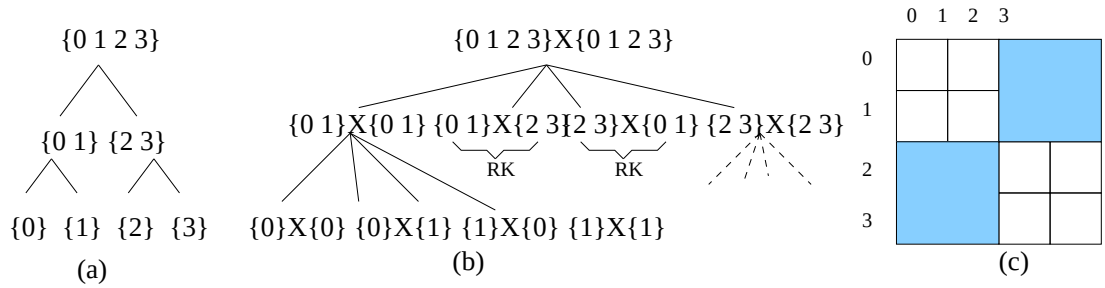


Figure 2.1: An example of an index cluster tree T_I, a block cluster tree $T_{I\times I}$, and an $\mathcal{H}$-matrix. (a) is T_I. (b) is $T_{I\times I}$. (c) is the corresponding $\mathcal{H}$-matrix. In (b) RK denotes a block satisfying the given admissibility condition. In (c) the dark color blocks are Rk-matrices and the white color blocks are full matrix blocks.

2.4 $\mathcal{H}$-matrix Construction Approaches

This section reviews the approaches to construct $\mathcal{H}$-matrices. The process to construct $\mathcal{H}$-matrices involves three steps: the first step is to construct an index cluster tree T_I; the second step is to define an admissibility condition; the last step is to construct the block cluster tree $T_{I\times I}$ using the index cluster tree and the admissibility condition. Depending on the information used in the $\mathcal{H}$-matrix construction process, construction approaches can be divided into two categories: geometric construction approaches and algebraic construction approaches.

When the concept of $\mathcal{H}$-matrices was first introduced, the construction of $\mathcal{H}$-matrix relied on the underlying geometric information of problems [38, 49]. The paper [51] shows a block decomposition approach for constructing a class of $\mathcal{H}$-matrices to represent the 2D stiffness and mass matrices arising from FEM applications, which uses the domain and grid information to construct the index cluster tree and block cluster tree.

In some applications, the geometric information underlying partial differential

equations is not available. In these cases, geometric $\mathcal{H}$-matrix construction approaches are not applicable. In [34, 55] algebraic approaches are proposed, which do not need geometric information. They only use matrix graphs to construct index cluster trees and block cluster trees.

2.4.1 Geometric Approaches

Geometric $\mathcal{H}$-matrix construction approaches are built upon the geometric information underlying problems.

First, let's assume that the set of basis functions used to discretize a problem is $\{\phi_0, \phi_1, \ldots, \phi_{n-1}\}$ and the domain of the problem is $\Omega \subset \mathbb{R}^d$. Geometric $\mathcal{H}$-matrix construction approaches associate each index $i \in I = \{0, 1, \ldots, n-1\}$ with a finite or boundary element ϕ_i. The support of a basis function ϕ_i is denoted as:

$$\Omega_i := \text{ supp } \phi_i. \tag{2.15}$$

Then the support of a cluster $s = \{i, i+1, \ldots, j\}$ is defined as:

$$\Omega_s = \bigcup_{i \in s} \Omega_i. \tag{2.16}$$

To simplify construction algorithms, instead of dealing with supports directly, for each support a point $x_i \in \Omega_i$ is selected to represent Ω_i. Then a box B_I can be defined as the smallest d-dimensional box that includes all the points x_i:

$$B_I = [a_1, b_1] \times \cdots \times [a_j, b_j] \times \cdots \times [a_d, b_d], \tag{2.17}$$

where $a_j = \min_{i \in I} < x_i, e_j >$, $b_j = \max_{i \in I} < x_i, e_j >$, and $\{e_1, \ldots, e_d\} \in \mathbb{R}_d$ are unit vectors. B_I does not necessarily contain all the supports $\Omega_{i \in I}$.

Here we introduce three frequently used geometric index cluster tree construction approaches: cardinality balanced clustering, geometrically balanced clustering, and clustering based on domain decomposition. Cardinality balanced clustering and geometrically balanced clustering are based on bisection.

2.4.1.1 Cardinality Balanced Clustering

The idea of cardinality balanced clustering [49] is to split a box associated with an index cluster into two smaller boxes so that they contain the same number of indices. The algorithm of cardinality balanced clustering to build T_I over an index set I is as follows:

1. Start with I as the root of T_I and the box B_I including all the points x_i.

2. For each $s \in T_I$ with $\#s > N_s$, split the box B_s into two boxes B_{s1} and B_{s2} such that B_{s1} and B_{s2} contain the same number of points. Define the children of s as $S(s) = \{s1, s2\}$: choose the direction $k = \text{argmax}_{j=1,\ldots,d}|b_j - a_j|$; sort all the indices in s, such that $\{x_{i_1,k}, \ldots, x_{i_{\#s},k}\}$ are in increasing order; then split s into $s_1 = \{i_1, \ldots, i_{\#s/2}\}$ and $s_2 = \{i_{\#s/2+1}, \ldots, i_{\#s}\}$.

3. Repeat the second step until the size of all the leaves is less than N_s: $\forall s \in \mathcal{L}(T_I), \#s \leq N_s$.

The advantage of cardinality balanced clustering is that it builds a balanced cluster tree with minimum depth. A tree is called balanced, if the children of the same node have roughly equal number of indices.

2.4.1.2 Geometrically Balanced Clustering

The idea of geometrically balanced clustering [49] is to find the direction of maximal extent of a box and split the box into two boxes in the same direction. The algorithm of geometric balanced clustering is defined as follows:

1. Start with I as the root of T_I and the box B_I including all the points x_i;

2. For each $s \in T_I$ with $\#s > N_s$, split the box B_s into two boxes B_{s1} and B_{s2}: find the direction $k := \text{argmax}|a_j - b_j|$; split B_s in the direction k into $B_{s1} = [a_1, b_1] \times \cdots \times [a_k, \frac{a_k+b_k}{2}] \times \cdots \times [a_d, b_d]$ and $B_{s2} = [a_1, b_1] \times \cdots \times [\frac{a_k+b_k}{2}, b_k] \times \cdots \times [a_d, b_d]$; split s into $s1 = \{i | x_i \in B_{s1}\}$ and $s2 = \{i | x_i \in B_{s2}\}$ and define the children of s as $S(s) = \{s1, s2\}$.

3. Repeat the second step until the size of all the leaves is less than N_s: $\forall s \in \mathcal{L}(T_I), \#s \leq N_s$.

These two boxes, B_{s1} and B_{s2}, may contain different number of points, which means the cluster tree obtained by geometric balanced clustering may not be as balanced as the tree obtained by cardinality balanced clustering.

Figure 2.2 shows an index cluster tree obtained by cardinality balanced clustering and Figure 2.3 shows an index cluster tree obtained by geometric balanced clustering when they are applied to the same problem.

2.4.1.3 Clustering Based on Domain Decomposition

Domain decomposition combined with the $\mathcal{H}$-matrix technique was first introduced in [39]. In [39] a domain Ω is decomposed into p non-overlapped sub-domains

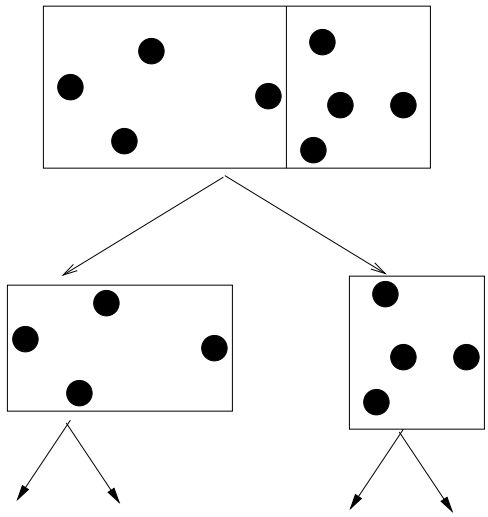


Figure 2.2: An example of T_I obtained by cardinality balanced clustering.

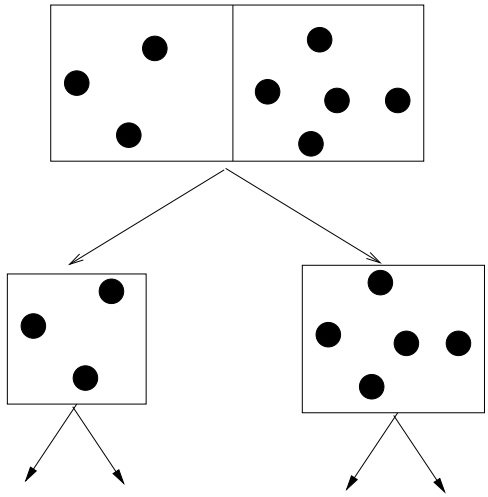


Figure 2.3: An example of T_I obtained by geometric balanced clustering.

$\{\Omega_1, \ldots, \Omega_p\}$ and one interior boundary Γ. The index set I is also decomposed and ordered in the sequence: $I = \{I_1, \ldots, I_p, I_\Gamma\}$. Then in each sub-domain cardinality balanced clustering approach is applied independently to build an $\mathcal{H}$-matrix.

The paper [34] introduces clustering based on domain decomposition which unifies the process of domain decomposition with index cluster tree construction to build $\mathcal{H}$-matrices. Its idea is that each domain Ω_s associated with a cluster s is decomposed into three sub-domains $\{\Omega_{s_1}, \Omega_{s_2}, \Omega_{s_3}\}$, where Ω_{s_1} and Ω_{s_2} are disconnected from each other and Ω_{s_3} is the domain that separates Ω_{s_1} from Ω_{s_2}. The domain decomposition also divides the cluster s into three subsets $\{s_1, s_2, s_3\}$. The set s_1 and s_2 are called domain clusters, while s_3 is called interface cluster. The clustering algorithm based on domain decomposition works in the following recursive way:

1. Start with the set I as the root of T_I and the box B_I that contains the domain Ω. The set of domain clusters $C_{dom} = \{\Omega\}$ and the set of interface clusters $C_{int} = \varnothing$.

2. For each domain cluster $s \in T_I$ with $\#s > N_s$, its corresponding domain box $B_s = [a_1, b_1] \times \cdots \times [a_k, b_k] \times \cdots \times [a_d, b_d]$ is split into two boxes B_{s_1} and B_{s_2} in the direction of maximal extent as described in geometric balanced clustering. The box B_{s_3} of the interface domain is constructed as $B_{s_3} = [a, b] \times \ldots \times [\widetilde{a}_k, \widetilde{b}_k] \times \cdots \times [a_d, b_d]$, where $\widetilde{a}_k = 1/2(a_k + b_k) - \max_{j \in s_3} \mathrm{diam}(\mathrm{supp}\varphi_j)$ and $\widetilde{b}_k = 1/2(a_k + b_k) + \max_{j \in s_3} \mathrm{diam}(\mathrm{supp}\varphi_j)$. Correspondingly the children of s is defined as $S(s) = \{s_1, s_2, s_3\}$, where $s_1 = \{i \in s \mid x_i \in B_{s_1}\}$, $s_2 = \{i \in s \mid$

$\text{supp} \bigcap \Omega_{s_1} = \varnothing\}$, and $s_3 = s - s_1 - s_2$. Then update $C_{dom} = C_{dom} \bigcup \{s_1, s_2\}$ and $C_{int} = C_{int} \bigcup s_3$.

3. For an interface cluster $t \in C_{int}$, its children are defined as: $S(t) = \{t\}$, if its level is divided by d; Otherwise its box B_t is split into two boxes $\{B_1, B_2\}$ in the direction of maximal extent, and $S(t) = \{t_1, t_2\}$, where $t_i = \{j \in t \mid x_j \in B_i\}$.

4. Repeat the second and third steps until $\forall s \in \mathcal{L}(T_I)$ has $\#s \leq N_s$.

Since interface clusters contain less indices than the domain clusters, the interface clusters are split at every dth step in order to adjust the size of interface clusters with the size of domain clusters.

2.4.2 Algebraic Approaches

If the geometric information underlying a problem is not available and only the matrix obtained by discretization is available, then geometric $\mathcal{H}$-matrix construction approaches can not be applied. In this case we may use algebraic $\mathcal{H}$-matrix construction approaches, which work on matrix graphs directly. The algebraic $\mathcal{H}$-matrix construction approach based on bisection simply splits every cluster into two subsets. The approach based on multilevel graph clustering builds a cluster tree bottom-up [55]. Its process is presented in Chapter 3. The algebraic approach based on nested dissection [34] builds a cluster tree top-down, which is discussed in the following section.

2.4.2.1 Nested Dissection

Nested dissection was originally proposed by Alan George in 1970's [30]. It is one of the reordering methods that are used in direct methods such as Gaussian elimination and LU factorization to solve sparse systems of linear equations. The reordering methods reorder matrices to reduce fill-ins introduced during Gaussian elimination or LU factorization process, therefore reduce computational complexity of these operations. Various nested dissection approaches were developed [30, 31, 32, 45].

In general, nested dissection uses a divide-conquer strategy. Given a graph $G(V, E)$, the basic idea of nested dissection is to find a separator $V_3 \subset V$ for the whole graph. A separator is a small set of vertices whose removal divides the graph G into smaller subgraphs $G(V_1, E_1), G(V_2, E_2)$, such that $V = V_1 \bigcup V_2 \bigcup V_3$, while $G(V_1, E_1)$ and $G(V_2, E_2)$ are disconnected from each other. V_1 and V_2 may be connected to V_3. The vertices in V are reordered such that the vertices in V_1 are numbered first, then the vertices in V_2, and the vertices in V_3 are numbered last. Recursively apply the same process to the subgraph $G(V_1, E_1)$ and $G(V_2, E_2)$.

The ordering generated by nested dissection does not introduce fill-ins to the large off diagonal zero blocks in LU factorization and maintain the matrix sparsity structure.

2.4.2.2 Clustering Based on Nested Dissection

The clustering approach based on nested dissection to build index cluster trees was proposed in [34]. The algorithm assumes that the graph is symmetric and con-

nected. The distance between the node i and node j is defined as the shortest path length between them. The shortest path $\text{dist}(i,j)$ is calculated using Dijkstra's algorithm. For a cluster $v \in I$, assume the connected subset of v is $\widetilde{v}$. The splitting algorithm which partition $\widetilde{v}$ into two parts is defined as follows:

1. Arbitrarily choose a node $i \in \widetilde{v}$.

2. Find a node $j \in \widetilde{v}$ with the longest path length to i and find a node $k \in \widetilde{v}$ with the longest path length to j.

3. Partition $\widetilde{v}$ into two subsets $\{\widetilde{v_1}, \widetilde{v_2}\}$ as

$$\widetilde{v_1} = \{i \in \widetilde{v} \mid \text{dist}(i,j) \leq \text{dist}(i,k)\},\ \widetilde{v_2} = \widetilde{v} - \widetilde{v_1}. \tag{2.18}$$

Then the clustering algorithm to build T_I based on the above splitting algorithm works in the following way:

1. The root of the cluster tree T_I is I and the set of domain clusters is $C_{dom} = I$.

2. For each domain cluster v, partition it into three parts $\{v_1, v_2, v_3\}$ as

$$v_1 = \widetilde{v_1}, v_2 = \{i \in \widetilde{v} \mid \text{dist}(i,k) \leq \text{dist}(i,j) - 1\}, \text{ and } v_3 = v - v_1 - v_2, \tag{2.19}$$

   where $\widetilde{v_1}$ is obtained by the splitting algorithm. Set the children of v to be $S(v) = \{v_1, v_2, v_3\}$. Add v_1, v_2 to the domain cluster set and add v_3 to the separator cluster set C_{sep}.

3. For each separator cluster v, divide it into two subsets using the splitting algorithm and define its children as $S(v) = \{\widetilde{v_1}, \widetilde{v_2}\}$.

Let $\text{dep}(T_I)$ be the depth of T_I and N be the number of vertices in the graph G, the complexity of the above clustering algorithm to build a T_I is $O(\text{dep}(T_I))$ [34]. The modified algorithms to build T_I for disconnected or directed graphs are also discussed in [34].

2.5 $\mathcal{H}$-matrix Arithmetic

There are two types of $\mathcal{H}$-matrix arithmetic: fixed rank $\mathcal{H}$-matrix arithmetic and adaptive rank $\mathcal{H}$-matrix arithmetic. Fixed rank $\mathcal{H}$-matrix arithmetic keeps the rank of Rk-matrix blocks below a fixed constant k. Adaptive $\mathcal{H}$-matrix arithmetic adapts the rank of individual Rk-matrix block to enforce the approximation accuracy: for each Rk-matrix block $s \times t$, its rank is set to $\text{rank}(M_{s\times t}) = \min\{k \mid \sigma_k \leq \delta\sigma_1\}$, where σ_i denotes the ith largest singular value of $M_{s\times t}$ and $\delta \in (0, 1]$ is a constant to determine the accuracy within each block.

Operations defined in $\mathcal{H}$-matrix arithmetic include addition, multiplication, inversion, and LU factorization. These operations are defined recursively based on the block cluster tree structure of $\mathcal{H}$-matrices. In [34, 49] the computational complexity of these $\mathcal{H}$-matrix operations is comprehensively analyzed. Based on certain assumptions it is proved that the computational complexity of $\mathcal{H}$-matrix operations is optimal and bounded by $O(n \log^{\alpha} n)$, where n is the size of a problem and $\alpha = 2$ or 3 is a moderate constant. The following sections give a brief description of each $\mathcal{H}$-matrix operation.

2.5.1 Rk-matrix Addition

The exact addition of two rank k matrices gives a rank $2k$ matrix. In order to maintain the low rank, the truncated singular value decomposition [10] is used in Rk-matrix addition. The truncated SVD approximates the exact sum which is an R2k-matrix by an Rk-matrix with computational complexity of $O(k^2(n+m)+k^3)$. Let $M_1 = A_1 B_1^T$, $M_2 = A_2 B_2^T$ be two Rk-matrices, then the truncated SVD which truncates the sum $M = [A_1, A_2][B_1, B_2]^T$ to an Rk-matrix $\widetilde{M} = [\widetilde{A}, \widetilde{B}^T]$ is defined in Algorithm 2.1.

Algorithm 2.1 Truncated Singular Value Decomposition

1: Calculate a truncated QR-decomposition of $[A_1, A_2] = Q_A R_A$.

2: Calculate a truncated QR-decomposition of $[B_1, B_2] = Q_B R_B$.

3: Calculate a singular value decomposition of $R_A R_B^T = U \Sigma V^T$.

4: Pick the first k columns of U and V: $\widetilde{U} = [U_1, \dots U_k], \widetilde{V} = [V_1, \dots V_k]$

5: Calculate the Rk-matrix representation of the truncated sum as: $\widetilde{M} = \widetilde{A}\widetilde{B}^T$, where $\widetilde{A} = Q_A \widetilde{U} \mathrm{diag}(\Sigma_{11}, \dots, \Sigma_{kk})$, $\widetilde{B} = Q_B \widetilde{V}$.

$\widetilde{M}$ is the best approximation to M with respect to Frobenius and spectral norm in the set of Rk-matrices [49].

2.5.2 $\mathcal{H}$-matrix-vector Multiplication

The multiplication of an $\mathcal{H}$-matrix $\mathcal{H}(T_{I\times J})$ and a vector v is defined recursively as the multiplication of each block in $\mathcal{H}(T_{I\times J})$ with the corresponding segment in v. If the block is an Rk-matrix, then Rk-matrix-vector multiplication is called; if it is a full matrix block, then full-matrix-vector multiplication is called; otherwise $\mathcal{H}$-matrix-vector multiplication is recursively called for each child block. The pseudo code of $\mathcal{H}$-matrix-vector multiplication is given as follows:

```
hmat_vec_mul (H, v, r)
{
  if H is an Rk-matrix
    r <- rkmat_vec_mul(H, v);
  else if H is a full-matrix
    r <- fmat_vec_mul(H, v);
  else
    for each child H_i of H
      hmat_vec_mul(H_i, v, r);
}
```

$\mathcal{H}$-matrix-vector multiplication yields the exact result and no approximation is involved.

2.5.3 $\mathcal{H}$-matrix-matrix Multiplication

Based on $\mathcal{H}$-matrix-vector multiplication, the multiplication of a $\mathcal{H}$-matrix with a full matrix can be defined. The multiplication of an $\mathcal{H}$-matrix $\mathcal{H}(T_{I\times I})$ with a full matrix M yields a full matrix R. Each column $R[i]$ is obtained by calling $\mathcal{H}$-matrix-vector multiplication with $\mathcal{H}(T_{I\times J})$ and the column $M[i]$ as inputs. The pseudo code of $\mathcal{H}$-matrix-matrix multiplication is given as follows:

```
hmat_mat_mul (H, M, R)
```

```
{
  for each column i
    hmat_vec_mul(H, M[i], R[i]);
}
```

As $\mathcal{H}$-matrix-vector multiplication, $\mathcal{H}$-matrix-matrix multiplication also yields exact results.

2.5.4 $\mathcal{H}$-matrix Addition $+_{\mathcal{H}}$

$\mathcal{H}$-matrix addition $+_{\mathcal{H}}$ is defined to add two $\mathcal{H}$-matrices with the same block cluster tree structure and gives a sum with the same tree structure. The operation works in a recursive way: if both operands are full-matrices then it calls full-matrix addition to add them together and gives a full matrix; if they are Rk-matrices then the operation calls the truncated SVD to obtain an Rk-matrix; otherwise it recursively calls $+_{\mathcal{H}}$ for each subblock on the next level. The pseudo code of $\mathcal{H}$-matrix addition is given as follows:

```
hmat_hmat_add (H, A , B)
{
  if H is a full-matrix
    fullmat_add(H, A, B);
  else if H is an Rk-matrix;
    rkmat_add(H, A, B);
  else
    for each child H_i of H
      hmat_hmat_add(H_i, A_i. B_i);
}
```

H is an $\mathcal{H}$-matrix to store the result, which is allocated before calling the function. The operation $+_{\mathcal{H}}$ gives an approximation to the exact sum since the truncated SVD is used to add two Rk-matrix blocks.

2.5.5 $\mathcal{H}$-matrix Multiplication $*_{\mathcal{H}}$

In general, multiplication of two $\mathcal{H}$-matrices $\mathcal{H}_1(T_{I\times K})$ and $\mathcal{H}_2(T_{K\times J})$ yields another $\mathcal{H}$-matrix $\mathcal{H}(T_{I\times J})$. Depending on the cluster tree structure of the operands and the result, there could be four cases:

1. If $\mathcal{H}_1$, $\mathcal{H}_2$ and $\mathcal{H}$ are all subdivided on the next level, then recursively call $*_{\mathcal{H}}$ for the subblocks on the next level.

2. If $\mathcal{H}$ has sub-blocks on the next level but $\mathcal{H}_1$ or $\mathcal{H}_2$ does not, then Rk-matrix multiplication or $\mathcal{H}$-matrix-matrix multiplication is called to calculate the product of $\mathcal{H}_1$ and $\mathcal{H}_2$, which is either an Rk-matrix or a full matrix.

3. If $\mathcal{H}$ is a full matrix, the product of $\mathcal{H}_1$ and $\mathcal{H}_2$ is added to $\mathcal{H}$ directly.

4. If $\mathcal{H}$ is an Rk-matrix, then the hierarchical multiplication and truncation is called to get an approximation of the product in the Rk-matrix format.

The hierarchical multiplication and truncation is based on the idea that if an $\mathcal{H}$-matrix $\mathcal{H}$ only has Rk-matrix subblocks, then it can be treated as the sum of Rk-matrices as shown in (2.20):

$$\begin{bmatrix} R_1 & R_2 \\ R_3 & R_4 \end{bmatrix} = \begin{bmatrix} R_1 & 0 \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & R_2 \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ R_3 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & R_4 \end{bmatrix}. \tag{2.20}$$

By calling the truncated SVD we can add these Rk-matrices together and obtain an Rk-matrix approximation to $\mathcal{H}$. The hierarchical multiplication and truncation

works in a recursive way as shown in Figure 2.4: if $\mathcal{H}_1$ or $\mathcal{H}_2$ are not subdivided then the product is truncated into an Rk-matrix directly; else recursively call the hierarchical multiplication and truncation to do blockwise multiplication and truncate the approximate product, which has only Rk-matrix subblocks due to the previous truncation, to an Rk-matrix block. The pseudo code of $\mathcal{H}$-matrix multiplication is

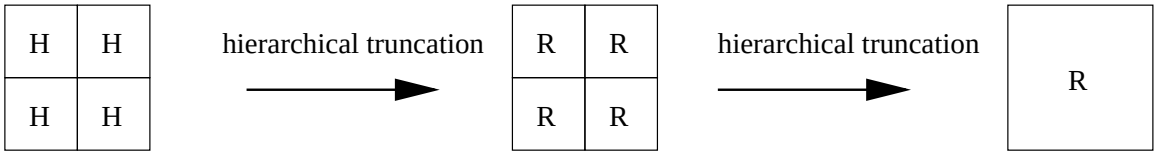


Figure 2.4: The process of the hierarchical multiplication and truncation.

given as follows:

```
hmat_hmat_mul (H, A , B)
{
  if H, A, and B all have children
    for i=1 to H->block_cols
      for j=1 to H->block_rows
        for k=1 to A->block_rows
          hmat_hmat_mul(H_i_j , A_i_k , B_k_j);
  else if H has children
    if A or B is an Rk-matrix
      hmat_rkmat_mul(H, A, B);
    else //A or B is a full-matrix
      hmat_mat_mul(H, A, B);
  else //H is an Rk-matrix;
    hierarchical_mul_trunc(H, A, B);
}
```

where H is an $\mathcal{H}$-matrix to store the result.

2.5.6 $\mathcal{H}$-matrix Inversion $inv_{\mathcal{H}}$

The operation $inv_{\mathcal{H}}$ is defined based on the block Gauss-Jordan elimination, in which the exact matrix operations are replaced by the corresponding $\mathcal{H}$-matrix operations defined in $\mathcal{H}$-matrix arithmetic. $inv_{\mathcal{H}}$ yields an approximate inverse. Let $+_{\mathcal{H}}$ and $*_{\mathcal{H}}$ be $\mathcal{H}$-matrix addition and multiplication as described in the previous sections. Assume $\mathcal{H} = \begin{bmatrix} \mathcal{H}_{11} & \mathcal{H}_{12} \\ \mathcal{H}_{21} & \mathcal{H}_{22} \end{bmatrix}$ has 2×2 blocks on the top level. The Schur complement S can be approximated by $S = \mathcal{H}_{22} -_{\mathcal{H}} \mathcal{H}_{21} *_{\mathcal{H}} inv_{\mathcal{H}}(\mathcal{H}_{11}) *_{\mathcal{H}} \mathcal{H}_{21}$. Then the $\mathcal{H}$-matrix inversion can be obtained recursively as:

$$inv_{\mathcal{H}}(\mathcal{H}) = \begin{bmatrix} \mathcal{H}'_{11} & \mathcal{H}'_{12} \\ \mathcal{H}'_{21} & \mathcal{H}'_{22} \end{bmatrix}, \tag{2.21}$$

where

$$\mathcal{H}'_{11} = inv_{\mathcal{H}}(\mathcal{H}_{11}) +_{\mathcal{H}} inv_{\mathcal{H}}(\mathcal{H}_{11}) *_{\mathcal{H}} \mathcal{H}_{12} *_{\mathcal{H}} inv_{\mathcal{H}}(S) *_{\mathcal{H}} \mathcal{H}_{21} *_{\mathcal{H}} inv_{\mathcal{H}}(\mathcal{H}_{11}),$$

$$\mathcal{H}'_{12} = -inv_{\mathcal{H}}(\mathcal{H}_{11}) *_{\mathcal{H}} \mathcal{H}_{12} inv_{\mathcal{H}}(S),$$

$$\mathcal{H}'_{21} = -inv_{\mathcal{H}}(S) *_{\mathcal{H}} H_{21} *_{\mathcal{H}} inv_{\mathcal{H}}(\mathcal{H}_{11}), \text{ and}$$

$$\mathcal{H}'_{22} = inv_{\mathcal{H}}(S).$$

2.5.7 $\mathcal{H}$-LU Factorization

$\mathcal{H}$-LU factorization was proposed in [38]. The algorithm to compute $\mathcal{H}$-LU factors was presented in [19, 34, 51]. $\mathcal{H}$-LU factors obtained by $\mathcal{H}$-LU factorization are in $\mathcal{H}$-matrix format and can be done recursively with computational complexity of $O(n \log^{\alpha} n)$.

To define $\mathcal{H}$-LU factorization, first a triangular solver needs to be defined. A triangular solver solves a triangular system $AX = B$, where X is an unknown $\mathcal{H}$-matrix, B is a given $\mathcal{H}$-matrix, and A is a given upper or lower triangular $\mathcal{H}$-matrix. The triangular solver works in the following way:

1. If both X and B are Rk-matrices (that is $X = X_1 X_2^T$ and $B = B_1 B_2^T$), then $X_2 = B_2$ and $X_1 = A^{-1} B_1$ by calling full matrix LU factorization.

2. Else if both X and B are full matrices, then call full matrix LU factorization to solve $AX = B$.

3. Otherwise, if X and B have subblocks on the next level, the solution X is obtained by solve the block triangular system. Let's assume X and B have 2×2 subblocks on the next level and A is a lower triangular $\mathcal{H}$-matrix as in (2.22):

$$\begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{bmatrix} = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}, \tag{2.22}$$

where L_{11} and L_{22} are lower triangular $\mathcal{H}$-matrices. Then X_{11} and X_{12} can be obtained by recursively calling the triangular solver to solve $L_{11} X_{11} = B_{11}$ and $L_{11} X_{12} = B_{12}$; X_{21} is obtained by solving $L_{22} X_{21} = B_{21} -_{\mathcal{H}} L_{21} \times_{\mathcal{H}} X_{11}$; finally, X_{21} is obtained by solving $L_{22} X_{22} = B_{22} -_{\mathcal{H}} L_{21} \times_{\mathcal{H}} X_{12}$.

Given the triangular the algorithm of $\mathcal{H}$-LU factorization for

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{bmatrix} \tag{2.23}$$

is defined by Algorithm 2.5.7. Using triangular solvers, $\mathcal{H}$-Cholesky factorization can

Algorithm 2.2 $\mathcal{H}$-LU Factorization

1: If A_{11} or A_{22} is a full matrix leaf, then call full-matrix LU-factorization.

2: Otherwise, recursively apply $\mathcal{H}$-LU factorization to A_{11} to obtain L_{11} and U_{11}.

3: Apply a triangular solver to $A_{12} = L_{11}U_{12}$ with L_{11} obtained in the previous step to get U_{12}.

4: Apply a triangular solver to $A_{21} = L_{21}U_{11}$ with U_{11} obtained in the previous step to get U_{11}.

5: Finally recursively apply $\mathcal{H}$-LU factorization to $L_{22}U_{22} = A_{22} -_{\mathcal{H}} L_{21} *_{\mathcal{H}} U_{12}$ to obtain L_{22} and U_{22}.

also be defined in a similar way.

2.6 $\mathcal{H}$-matrix Preconditioner Technique

The $\mathcal{H}$-matrix operations like $\mathcal{H}$-matrix inversion, $\mathcal{H}$-LU factorization and $\mathcal{H}$-Cholesky factorization general yields approximate results with optimal computational complexity, which are possible candidates as preconditioners in iterative methods to solve systems of linear equations [13, 17, 20, 55, 57]. The $\mathcal{H}$-matrix preconditioner

technique to solve $Ax = b$ involves three steps: first build an $\mathcal{H}$-matrix $A_{\mathcal{H}}$ for A, either by geometric approaches or algebraic approaches; then compute an $\mathcal{H}$-matrix preconditioner $B_{\mathcal{H}}$, either by $\mathcal{H}$-matrix inversion or $\mathcal{H}$-LU factorization; at last $B_{\mathcal{H}}$ is used in iterative methods like GMRES or MINRES. In [15], an $\mathcal{H}$-matrix inverse is used as a preconditioner to solve convection-domain problems discretized by finite element methods. The construction of $\mathcal{H}$-matrix inverses is relatively expensive due to the large constants in the estimate of computational complexity. In [19], an approach to use $\mathcal{H}$-matrix LU factors as preconditioners is pursued to solve convection-domain problems, for both constant and non-constant convection directions. Compared to multigrid methods, $\mathcal{H}$-LU preconditioners usually need more set-up time but provide better convergence rates. [21] presents a $\mathcal{H}$-LU preconditioner based on domain decomposition to solve convection-domain problems. The approaches to build $\mathcal{H}$-LU preconditioners to solve problems of saddle point type are discussed in [13, 16, 17, 20].

CHAPTER 3
ALGEBRAIC $\mathcal{H}$-MATRIX CONSTRUCTION APPROACH BASED ON MULTILEVEL CLUSTERING

This chapter introduces a new algebraic $\mathcal{H}$-matrix construction approach we have developed, which is based on multilevel clustering. Compared to the algebraic approach based on nested dissection, which starts from the entire index set and then recursively splits the sets into sub-sets, our approach starts with the individual index and recursively aggregates these indices together until reaches the whole index set. This chapter also presents the numeric results of applying these two approaches to solve positive definite systems arising from finite element discretization of Poisson equations.

3.1 Multilevel Clustering

Graph partitioning is about to partition the vertices of a graph into roughly n equal parts such that the edge-cut is minimized. Graph partitioning problem arises in many areas such as scientific computing and engineering. For example graph partitioning can be applied to sparse matrix reordering to reduce fillings during factorization. Graph partitioning problem is NP-complete and is not expected to be solved in polynomial time. But there are many approximation algorithms: spectral partitioning methods [3] produce excellent partitions but are quite expensive; geometric partitioning methods [53] are much faster than spectral partitioning methods but they depend on the geometric information of graphs, which in some cases is not avail-

able; compared to the other approximation methods, multilevel graph partitioning methods [24, 46, 47] give good partitions at a moderate cost.

The basic idea of multilevel graph partitioning is to approximate the original graph using a sequence of coarser graphs built by merging vertices together. A multilevel graph partitioning usually has three phases: coarsening phase, in which a graph is coarsened down to a sequence of coarser graphs by multilevel clustering; partitioning phase, in which a partition is calculated over the coarsest graph; uncoarsening phase, in which the partition over the coarsest graph is projected back to the finest graph. Here we focus on the coarsening phase and use its idea to build index trees for $\mathcal{H}$-matrices.

3.2 An $\mathcal{H}$-matrix Construction Approach

In this section, we describe the new algebraic approach for $\mathcal{H}$-matrix construction, which is based on multilevel clustering [55, 56, 57]. This construction approach uses the information obtained by multilevel clustering to build an index cluster tree and a block cluster tree for an $\mathcal{H}$-matrix. It has three steps: first a sequence of coarser graphs are built over the original graph, which is analogous to the multilevel clustering coarsening process; in the second step an index cluster tree is constructed based on the sequence of coarser graphs and the clusters obtained during the coarsening process; last, the admissibility condition is defined and a block cluster tree is constructed.

3.2.1 Clusters and Coarser Graphs

The idea to build a coarser graph G_{i+1} over G_i is: first build clusters over the vertices of G_i; then G_{i+1} is constructed by merging the vertices of each cluster together as one vertex on the coarser graph. The approach to build clusters is based on heavy edge matching, which finds the maximal matching over a graph. A matching is a set of edges, no two of which are incident on the same vertex. The matching of maximal size is called maximal matching. Some approaches to find the maximal matching of a graph are: random matching, heavy edge matching, light edge matching, and heavy clique matching. Random matching (RM) randomly selects two unmatched nodes u and v, marks them as matched, and adds the edge (u, v) in the matching. The idea of Heavy edge matching (HEM) is to find a matching with maximal weight. HEM randomly selects an unmatched node v, among its unmatched neighbors the vertex u with the maximum edge weight is selected, and the edge (u, v) is added to the matching. Heavy clique matching (HCM) collapses two vertices with high edge connection. The complexity of these algorithms is $O(\#E)$. Among them HEM aims to decrease the edge weight of the coarser graph and the experiments show that HEM is a good matching algorithm resulting in good partition[47]. The original HEM algorithm to build a matching C_i over $G_i(V_i, E_i)$ is given in Algorithm 3.1, in which vertices with no unmatched neighbors are matched by themselves.

The problem of the original HEM is that it may give a cardinality unbalanced index cluster tree T_I, especially when it is applied to a sparse graph. The reason is that in the matrix graph of a sparse matrix, each vertex has $O(1)$ neighbors, and as

Algorithm 3.1 HEM($G(V, E)$)

$C = \varnothing$;

while $V \neq \varnothing$ **do**

randomly pick up a node $v \in V$;

if the set of adjacent unmatched vertices of v: $Adj(v) \neq \varnothing$ **then**

pick up a node $u \in Adj(v)$ with the heaviest edge weight;

mark v and u as matched and $V = V - \{v, u\}$;

$C = C \bigcup \{(v, u)\}$;

else

mark v as matched and $V = V - \{v\}$;

$C = C \bigcup \{(v)\}$;

end if

end while

HEM continues marking nodes as matched, the unmatched nodes will have higher chance to stay isolated (that is they have no unmatched neighbors). If a vertex or its corresponding vertex on coarser graphs remains isolated as we continue building coarser graphs, the resulting cluster tree T_I, as discussed in Section 3.2.2 may not be cardinality balanced. Figure 3.2.1 shows an example of cardinality unbalanced tree obtained by the original HEM. To make T_I more cardinality balanced, we modify HEM by splitting the vertices of $G_i(V_i, E_i)$ into two subsets $S_{i,1}$ and $S_{i,2}$, where $V_i = S_{i,1} \bigcup S_{i,2}$. $S_{i,1}$ contains the vertices build by the isolated vertices of G_{i-1} and

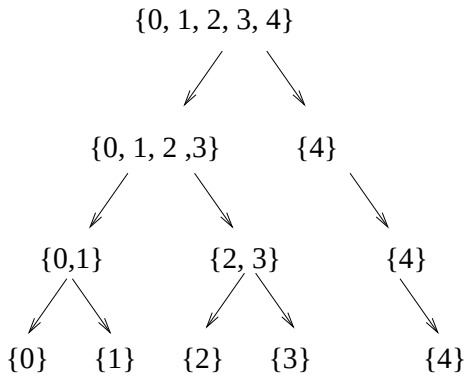


Figure 3.1: An example of a cardinality unbalanced cluster tree T_I obtained by the original HEM algorithm.

$S_{i,2}$ contains the vertices built by merging two matched vertices of G_{i-1}. The vertices in $S_{i,1}$ are with higher priority to be selected than those in $S_{i,2}$. The modified HEM is given in Algorithm 3.2.

After apply HEM over $G_i(V_i, E_i)$, a set of clusters $C_i = \{C_i^0, \ldots, C_i^m\}$ are built over V_i. The size of each cluster $\#C_i^k$ is either 1 or 2. The algorithm to build a coarser graph $G_{i+1}(V_{i+1}, E_{i+1})$ is given in Algorithm 3.3.

Recursively applying the above multilevel clustering process gives a sequence of graphs $\{G_0, G_1, \ldots, G_h\}$. We end this sequence with G_h, when the size of its vertex set $\#V_h$ is sufficiently small. The value of h is close to $\log(\#V_0)$ by the modified HEM algorithm. The complexity of the modified HEM algorithm is bounded by $O(\#E_0 \log(\#V_0))$. Figure 3.2 illustrates the multilevel clustering process with 2 levels

Algorithm 3.2 Modified_HEM($G_i(V_i, E_i), S_{i,1}, S_{i,2}$)

$C_i = \varnothing$;

while $S_{i,1} \neq \varnothing$ **do**

randomly pick up an node $v \in S_{i,1}$;

if the set of adjacent unmatched vertices of v: $Adj(v) \neq \varnothing$ **then**

randomly pick up a vertex $u \in Adj(v)$ with the highest edge weight;

mark v and u as matched and $V = V - \{v, u\}$;

$C_i = C_i \bigcup \{(v, u)\}$;

else

mark v as matched; $S_{i,1} = S_{i,1} - \{v\}$; $C_i = C_i \bigcup \{(v)\}$;

end if

end while

while $S_{i,2} \neq \varnothing$ **do**

randomly pick up an node $v \in S_{i,1}$;

if the set of adjacent unmatched vertices of v: $Adj(v) \neq \varnothing$ **then**

randomly pick up an node $u \in Adj(v)$ with the highest edge weight;

mark v and u as matched and $V = V - \{v, u\}$;

$C_i = C_i \bigcup \{(v, u)\}$;

else

mark v as matched; $S_{i,2} = S_{i,2} - \{v\}$; $C_i = C_i \bigcup \{(v)\}$;

end if

end while

Algorithm 3.3 The Algorithm to Build a Coarser Graph $G_{i+1}(V_{i+1}, E_{i+1})$

1: Initially $S_{i+1,1} = \varnothing$, $S_{i+1,2} = \varnothing$, $V_{i+1} = \varnothing$, and $E_{i+1} = \varnothing$.

2: For each cluster $C_i^k \in C_i$ build a node k in G_{i+1}: $V_{i+1} = V_{i+1} \bigcup \{k\}$.

3: If $\#C_i^k = 1$ then $S_{i+1,1} = S_{i+1,1} \bigcup \{k\}$, else $S_{i+2,1} = S_{i+2,1} \bigcup \{k\}$;

4: Add an edge (k, l) to $E_{i+1} \Longleftrightarrow \exists s \in C_i^k, t \in C_i^l$, and $(s, t) \in E_i$;

5: The edge weight of $(k, l) \in E_{i+1}$can be computed by:

$$w_{k,l} = \sum_{s \in C_i^k} \sum_{t \in C_j^l} e_{(s,t)},\ e_{(s,t)} \in E_i. \tag{3.1}$$

for a graph defined by the following matrix:

$$M = \begin{bmatrix} * & 4 & 0 & 1 & 0 & 0 \\ 4 & * & 0 & 1 & 1 & 2 \\ 0 & 0 & * & 3 & 0 & 1 \\ 1 & 1 & 3 & * & 0 & 0 \\ 0 & 1 & 0 & 0 & * & 3 \\ 0 & 2 & 1 & 0 & 3 & * \end{bmatrix} \tag{3.2}$$

3.2.2 Building T_I

Using the sequence of graphs $(G_0, G_1, \ldots, G_h)$ and the clusters obtained by the multilevel clustering process described above, we can build an index cluster tree T_I.

To build T_I, first we build a tree $\widetilde{T}_I$, whose leaves correspond to the indices of

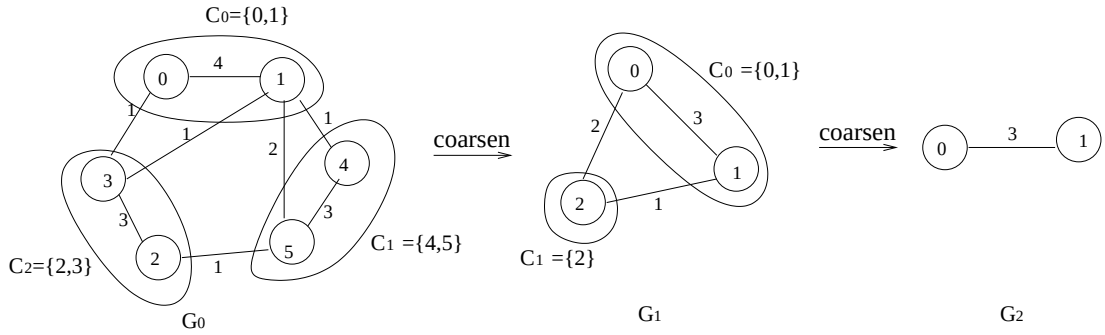


Figure 3.2: An example of the multilevel clustering process with 2 levels. The graph G_0 is defined by the matrix of (3.2).

the original matrix M, from bottom to top as follows:

1. For each vertex $v_i \in V(G_0)$ there is a leaf $i \in \widetilde{T}_I$.

2. The node $s \in \widetilde{T}_I$ on level i is the parent of the node t of level $i-1$, if and only if $t \in C_{i-1}^s$ obtained on G_{i-1}.

Figure 3.4(a) shows the tree $\widetilde{T}_I$ based on the sequence of graphs and clusters in Figure 3.2. A set $\widetilde{I}$ is created by listing the leaves of $\widetilde{T}_I$ from left to right. Different from I, $\widetilde{I}$ may not have the indices in order because of the clustering process. To connect the indices of $\widetilde{I}$ with those of I, a mapping function is built by mapping each index in $\widetilde{I}$ to an index in I at the same position, as shown in Figure 3.3. The

Old indexes	{0,	1,	4,	5,	2,	3}
	↓	↓	↓	↓	↓	↓
New indexes	{0,	1,	2,	3,	4,	5}

Figure 3.3: Index mapping built for T_I

definition of the mapping function leads to the definition of a permutation matrix P. Symmetrically permuting the original matrix M as $M_{reorder} = P^T MP$, we have the reordered matrix $M_{reorder}$. Apply the mapping function to the leaves of $\widetilde{T}_I$ and we have a new tree $\widetilde{T}_I$ given in Figure 3.4(b). Finally we build T_I by remapping each node in $\widetilde{T}_I$ from the leaves to the root by a set of indices $L_j^i \subset I$ (see Figure 3.4(c)):

1. If a node j^0 is a leaf of $\widetilde{T}_I$ then in T_I it is represented as $L_j^0 = \{j\}$;

2. If a node $j^i \in \widetilde{T}_I$ has children t^{i-1} and k^{i-1}, then $L_j^i \in T_I$ has two children L_t^{i-1} and L_k^{i-1}, where $L_j^i = L_t^{i-1} \bigcup L_k^{i-1}$.

3. If a node $j^i \in \widetilde{T}_I$ has only one child k^{i-1}, then $L_j^i \in T_I$ has only one child L_k^{i-1} and $L_j^i = L_k^{i-1}$.

T_I obtained by the above process is an index cluster tree for the reordered matrix $M_{reorder}$, which has the following properties:

1. T_I has $h+1$ levels, where h is the number of levels in the multilevel coarsening process.

2. The root of T_I is set $I = \{0, 1, 2, \ldots, n-1\}$, whose elements represent the indices of the permuted matrix $M_{reorder}$.

3. The nodes on the same level formed a partitioning over the index set I.

The whole process to build T_I is shown in Figure 3.4. The following matrix:

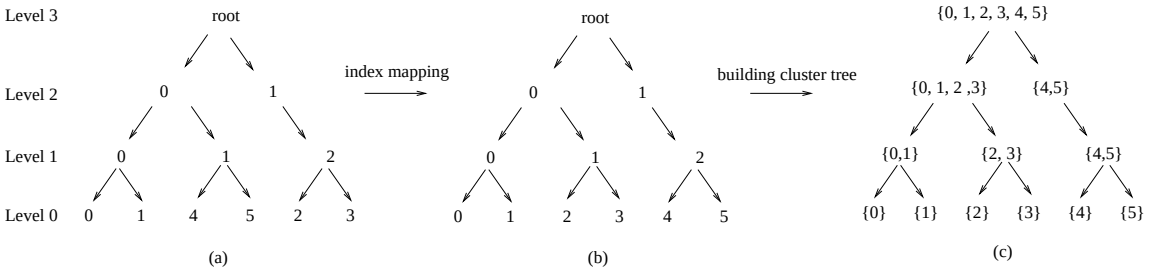


Figure 3.4: The process to build an index cluster tree T_I using the graphs in Figure 3.2. (a) is the tree $\widetilde{T}_I$ before the index mapping. (b) is the tree $\widetilde{T}_I$ after the index mapping. (c) is the final index cluster tree T_I.

$$M_{reorder} = \begin{bmatrix} * & 4 & 0 & 0 & 0 & 1 \\ 4 & * & 1 & 2 & 0 & 1 \\ 0 & 1 & * & 3 & 0 & 0 \\ 0 & 2 & 3 & * & 1 & 0 \\ 0 & 0 & 0 & 1 & * & 3 \\ 1 & 1 & 0 & 0 & 3 & * \end{bmatrix}, \tag{3.3}$$

is the reordered matrix $M_{reorder}$ obtained by permuting M of Equation (3.2) using the index mapping function of Figure 3.3.

3.2.3 Building $T_{I\times I}$

The block cluster tree $T_{I\times I}$ describes a multilevel block partitioning over the reordered matrix $M_{reorder}$. To build $T_{I\times I}$, we use the cluster tree T_I and the multilevel graphs $(G_0, G_1, G_2, \ldots, G_h)$. If two vertices s and t are not connected in G_i, then the corresponding matrix block $s \times t$ of $M_{reorder}$ is a block with only zeros. Those blocks of zeros are low rank matrices that can be represented exactly using Rk-matrices. So we define the following admissibility condition for the block cluster tree construction

approach:

$$s \times t \text{ is admissible } \iff s \text{ and } t \text{ are disconnected in } G_i. \tag{3.4}$$

The algorithm to build a block cluster tree is given in Algorithm 3.4. The whole

Algorithm 3.4 Block Cluster Tree Building Algorithm Based On Multilevel Clustering

1: The root of $T_{I\times I}$ is $I \times I$.

2: If $L_r^i \times L_s^i$ is a node of $T_{I\times I}$, r is connected to s in G_i, and $\#L_r^{(i)}$, $\#L_s^i > N_s$, then the children of $L_r^{(i)} \times L_s^{(i)}$ are $L_v^{i-1} \times L_w^{i-1}$ where L_v^{i-1} is a child of L_r^i and L_w^{i-1} is a child of L_s^i in T_I.

3: Else if $L_r^i \times L_s^i$ is a node of $T_{I\times I}$, r is *not* connected to s in G_i, and $\#L_r^i$, $\#L_s^i > N_s$, then $L_r^i \times L_s^i$ is a Rk-matrix leaf node in $T_{I\times I}$.

4: Else if $L_r^i \times L_s^i$ is a node of $T_{I\times I}$, and $\#L_r^i \leq N_s$ or $\#L_s^i \leq N_s$, then $L_r^i \times L_s^i$ is a dense leaf node in $T_{I\times I}$.

process the algebraic $\mathcal{H}$-matrix construction approach based on multilevel clustering is put together in Figure 3.5. Notice that in our example we used $N_s = 1$ for the minimal block size. However the process is similar for other values of N_s except that we stop sooner (higher) in the tree.

The difference between our $\mathcal{H}$-matrix construction method and the classic methods for building $\mathcal{H}$-matrix is that the classic methods need the geometric information underlying the problem to determine whether or not a block $L_s^i \times L_t^i$ in $T_{I\times I}$ to

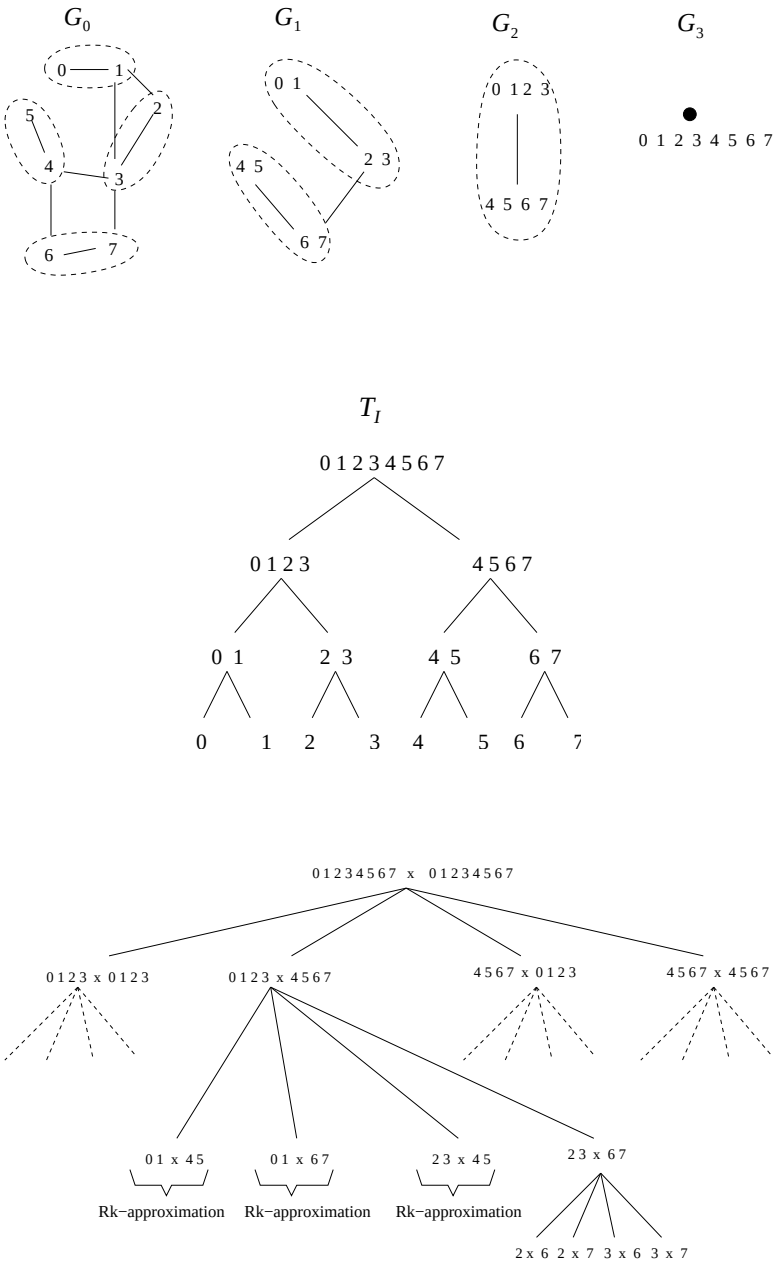


Figure 3.5: The illustration of the algebraic $\mathcal{H}$-matrix construction approach based on multilevel clustering. The top figure is the process to build the sequence of coarser graphs and clusters. The middle figure shows the corresponding cluster tree T_I. The bottom figure shows the block clustering tree $T_{I\times I}$ built using T_I and the coarser graphs.

be approximated by an Rk-matrix. In our algebraic method we use the edge weights in the coarser graphs obtained by multilevel clustering to decide how to build $T_{I\times I}$. Particularly, if $w_{st}^{(i)} = 0$ then the block $L_s^i \times L_t^i$ is a zero block, and it is represented in Rk-matrix format; $w_{st}^{(i)} \neq 0$ implies that the block $L_k^{(i)} \times L_t^{(i)}$ is a non-zero block and it will be partitioned into smaller blocks on the next level. In this way the original matrix can be representeded exactly as an $\mathcal{H}$-matrix and no approximation is needed. The difference between our multilevel coarsening based approach and the algebraic $\mathcal{H}$-matrix construction approach based on nested dissection is as follows. The multilevel clustering based algorithm constructs a cluster tree bottom-up: starts with the leaves and successively clusters them together until the root is reached. The nested dissection based approach in [34]) starts with the root and successively subdivides clusters until the leaves are reached, so its is a top-down approach.

3.3 Experimental Results

Using our algebraic construction approach combined with $\mathcal{H}$-matrix arithmetic, we can build various preconditioners to solve large systems of linear equations. In this section we show the experimental results of using $\mathcal{H}$-matrix preconditioners to solve the systems of linear equations arising from FEM discretization of Poisson equations.

The model problem of our experiment is the following Poisson equation:

$$\begin{cases} \nabla^2 u = f, u \in \Omega \\ \\ u(x,y) = e^{2x}\cos(2y) + x^3 - 3xy^2, u \in \partial\Omega. \end{cases} \tag{3.5}$$

Discretization of (3.6) using finite element methods yields the following system of linear equations:

$$Kx = b, \tag{3.6}$$

here K is a stiffness matrix. K is constructed using the grid generator developed by Persson and Strang [59] and the piecewise linear finite element method. Note that K is sparse, symmetric, and positive definite. Figure 3.6 shows an example of a mesh with the element size $h_0 \approx 0.2$ and the distribution of the non-zero entries in K based on the mesh.

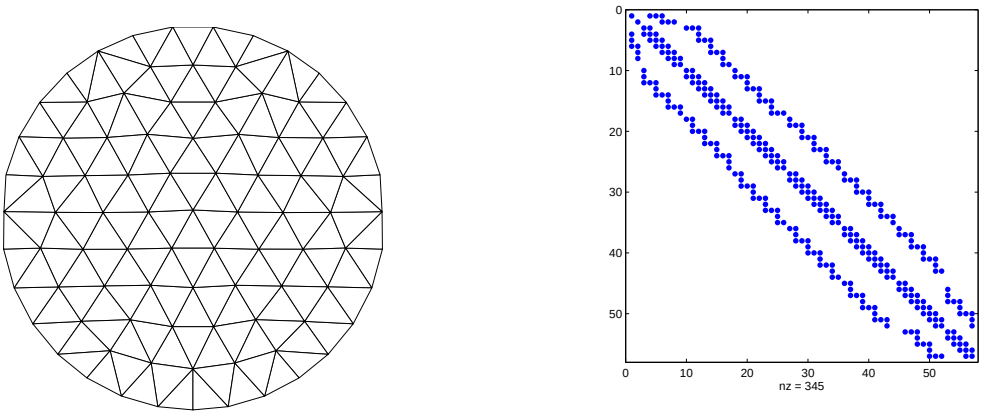


Figure 3.6: The left side is a mesh on a unit circle built by the grid generator with the element size $h_0 \approx 0.2$. The right side is the distribution of the non-zero entries (black dots) in the stiffness matrix K obtained by the discretization of the Poisson equation (3.6) using the mesh on the left.

In our test, to generate different size of probems we chose $h_0 \approx 0.020$, 0.015, 0.012, 0.010, 0.007, and 0.005. The corresponding numbers of unknowns are: $n =$ 8753, 15697, 24657, 35632, 73131, and 143834 respectively.

To show the performance our construction approach we compare it with the algebraic approach based on nested dissection (ND) [34]. We also compare $\mathcal{H}$-matrix preconditioners with other existing preconditioners. The preconditioners built in this experiment are: $\mathcal{H}$-inverse based on HEM (HEM-$\mathcal{H}$-INV), $\mathcal{H}$-LU factors based on HEM (HEM-$\mathcal{H}$-LU), $\mathcal{H}$-Cholesky factors based on HEM (HEM-$\mathcal{H}$-CH), $\mathcal{H}$-LU factors based on ND (ND-$\mathcal{H}$-LU), $\mathcal{H}$-Cholesky factors based on ND (ND-$\mathcal{H}$-CH), as well as Jacobi Over-Relxation (JOR). Here we use HEM to indicate the $\mathcal{H}$-matrix construction approach based on multilevel clustering and ND to indicate the $\mathcal{H}$-matrix construction approach base on nested dissection.

These preconditioners are used in the iterative method GMRES. The iteration of GMRES stops where the original residual is reduced by a factor of 10^{-12}. The convergence rate a is defined as the average decreasing speed of residual in each iteration. a is calculated by solving the equation: $a^t = 10^{-12}$, where t is the number of iterations.

To compute $\mathcal{H}$-LU and $\mathcal{H}$-Cholesky factors, we use the $\mathcal{H}$-matrix arithmetic with adaptive rank: the rank of each Rk-matrix block $M_{s\times t}$ approximating a matrix block A satisfies that $\text{rank}(M_{s\times t}) = \min\{\, k \mid \Sigma_k \leq \alpha\Sigma_1 \,\}$, where Σ_i is the ith largest singular value of A and α is a parameter to control the accuracy. We choose $\alpha =$ 0.0625. We also set the size of all the leaf blocks in an $\mathcal{H}$-matrix to $N_s = 40$.

To compute $\mathcal{H}$-inverses, we use the $\mathcal{H}$-matrix arithmetic with fixed rank: the rank of each Rk-matrix block $M_{s\times t}$ is less than k, since the fixed rank $\mathcal{H}$-matrix arithmetic gives better performance and convergence rates. Here we set $k = 4$. To make GMRES converge for calculating $\mathcal{H}$-inverses based on nested dissection (ND-$\mathcal{H}$-INV) the rank $k >= 8$ in our test. But its performance is much worse than HEM-$\mathcal{H}$-INV, so we do not include ND-$\mathcal{H}$-INV in our comparison. The experiments are carried out on a dual processor computer with 64-bit Athlon 4200++ CPU and 3GB memory.

Figure 3.7 compares the time consumed by the algebraic $\mathcal{H}$-matrix construction approach HEM and the approach ND to build an $\mathcal{H}$-matrix over the given sparse matrix K. Figure 3.7 shows that HEM based $\mathcal{H}$-matrix construction approach is a simpler approach and works faster than ND based approach. This is because HEM builds $\mathcal{H}$-matrices by clustering vertices together base on the edge weights of each randomly picked vertices while ND builds $\mathcal{H}$-matrices by split the set of vertices into three subsets and the process of finding the separate clusters needs more work.

Figure 3.8 compares the time taken to compute $\mathcal{H}$-matrix preconditioners by applying the $\mathcal{H}$-matrix arithmetic to the $\mathcal{H}$-matrices constructed by HEM and ND. Figure 3.9 compares the memory storage in Megabyte (MB) for these $\mathcal{H}$-matrix preconditioners. Figure 3.10 shows the time taken by the preconditioned GMRES iterations to converge using the given preconditioners.

Figure 3.11 compares the total running time (the time to construct preconditioners plus the time of preconditioned GMRES iterations). Figure 3.12 shows the

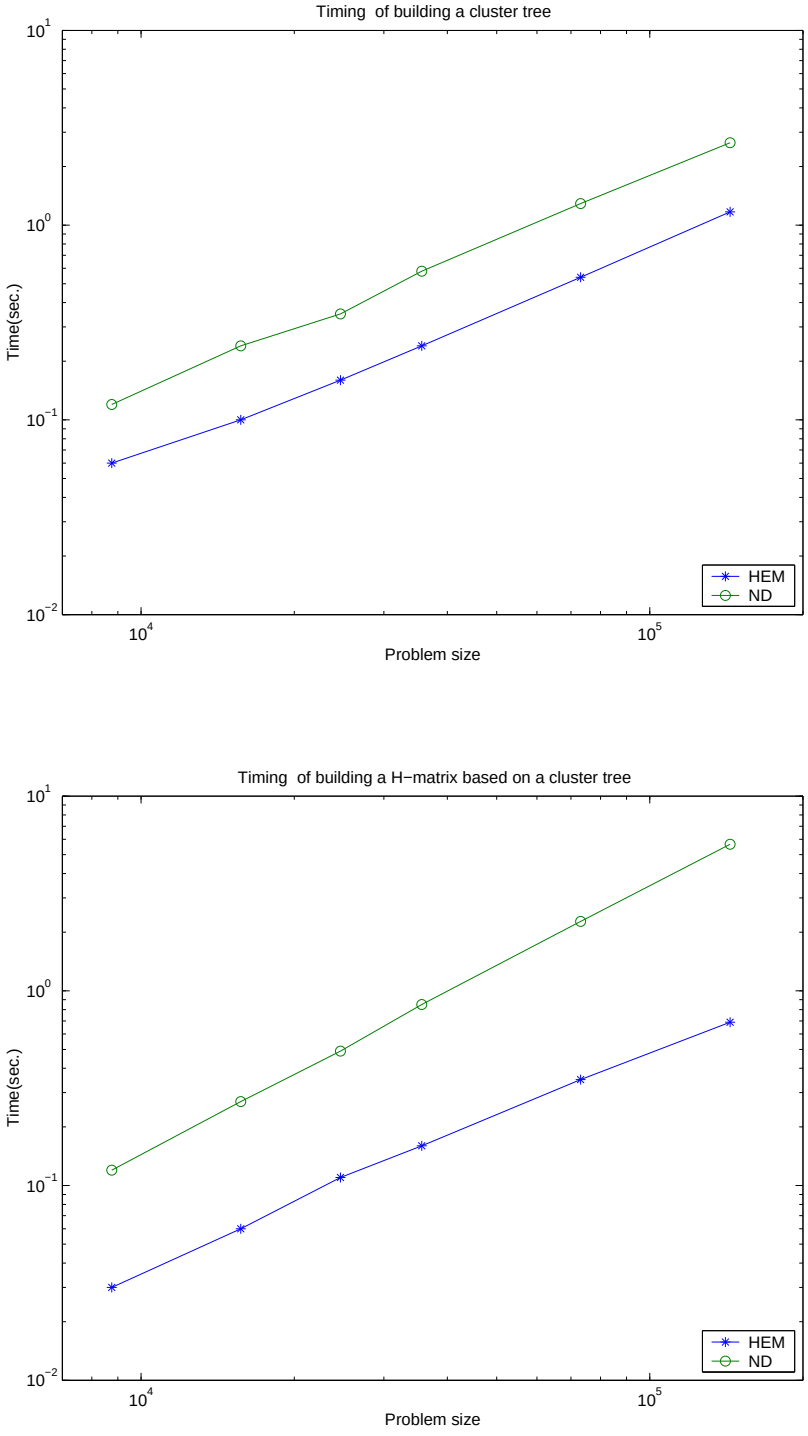


Figure 3.7: Comparison of the time complexity of the algebraic $\mathcal{H}$-matrix matrix construction approach based on multilevel clustering (HEM) and the approach based on nested dissection (ND) for various problem sizes. The top is the comparison of the time to build T_I. The bottom is the comparison of the time to build $T_{I\times I}$.

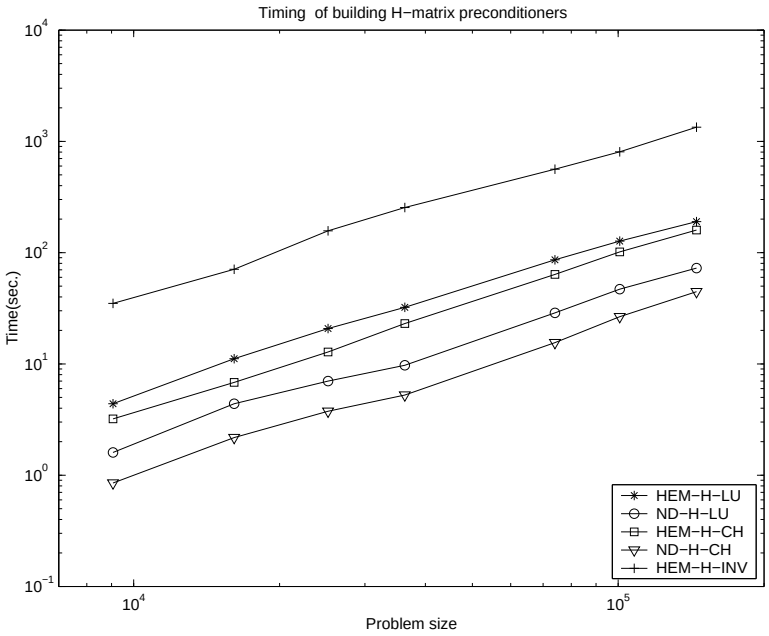


Figure 3.8: Comparison of the time to build $\mathcal{H}$-matrix preconditioners based on the $\mathcal{H}$-matrices constructed by HEM and ND.

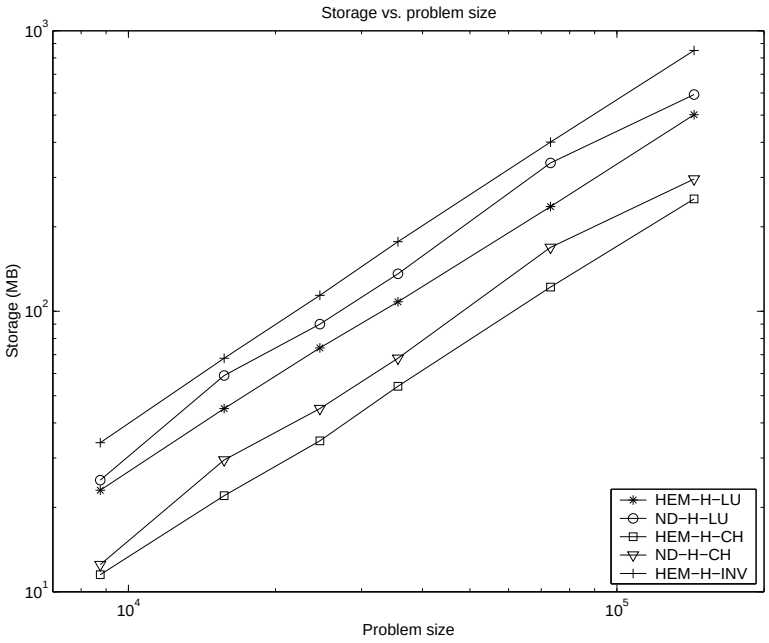


Figure 3.9: Comparison of the storage of various $\mathcal{H}$ matrix preconditioners obtained by HEM and ND.

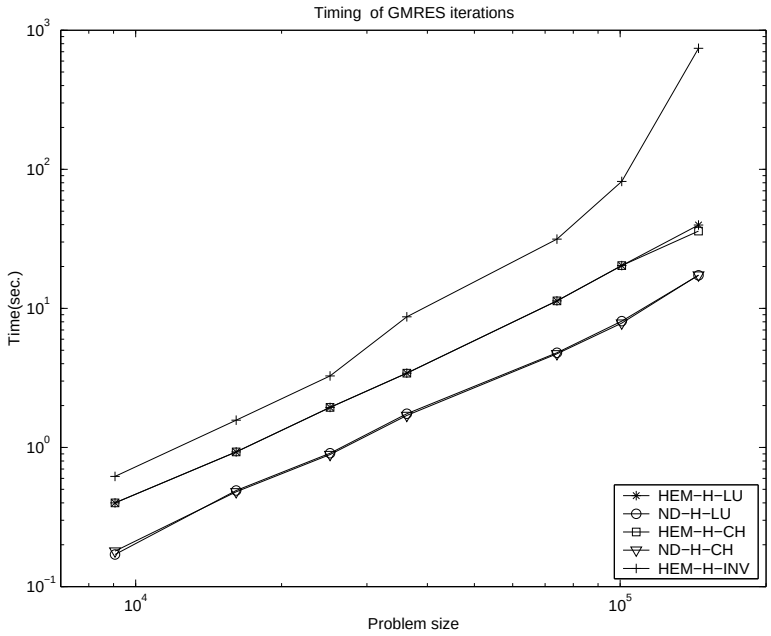


Figure 3.10: Comparison of the time for preconditioned GMRES iterations to converge using various preconditioners.

convergence rates of GMRES using JOR preconditioner and the $\mathcal{H}$-matrix preconditioners based on HEM and ND.

Based on the above figures, we can see that with respect to the convergence rate, all the $\mathcal{H}$-matrix preconditioners outperform JOR preconditioner. As to the total running time, all the $\mathcal{H}$-matrix preconditioners shows the same rate of time increase and the run time of the $\mathcal{H}$-matrix preconditioners increases much more slowly with problem size than JOR preconditioner. $\mathcal{H}$-inverses outperform JOR when the problem size is bigger than 10^5. In all, the $\mathcal{H}$-matrix preconditioners are much better than JOR preconditioner to solve the given system. The experiment results also show that $\mathcal{H}$-LU and $\mathcal{H}$-Cholesky factors are cheaper to compute than $\mathcal{H}$-inverses, which verifies the computational complexity analysis of the $\mathcal{H}$-matrix operations in

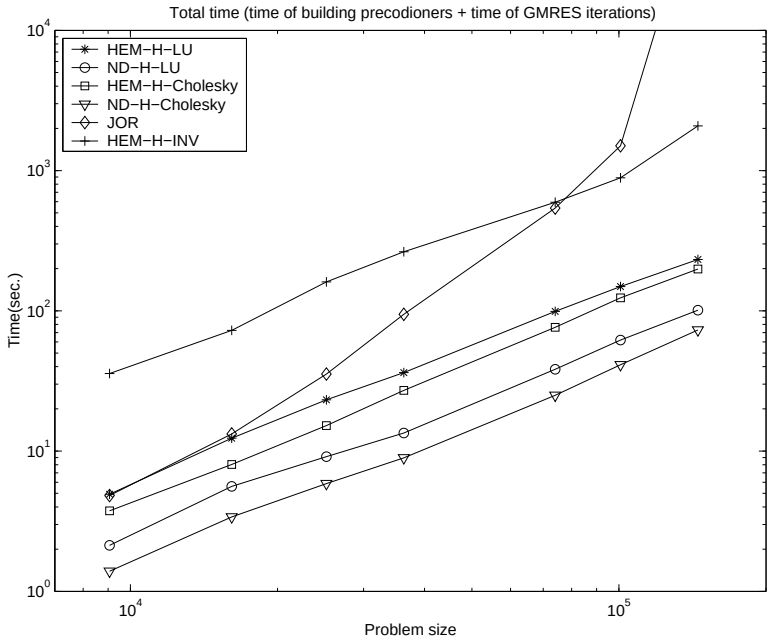


Figure 3.11: Comparison of the total time to solve the system of 3.6, which includes the time to build the preconditioners and the time of GMRES iterations.

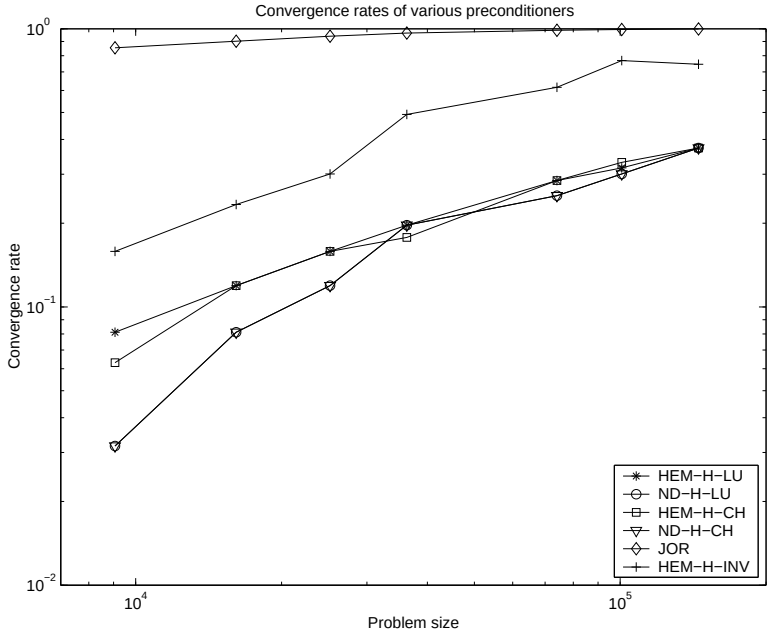


Figure 3.12: Comparison of the convergence rates of GMRES with JOR preconditioner and $\mathcal{H}$-matrix preconditioners.

Chapter 2. The $\mathcal{H}$-matrix construction approach based multilevel clustering and the $\mathcal{H}$-matrix construction approach based on nested dissection are comparable to each other, as they show the same behavior as regard to the memory requirement, computational complexity, and the convergence rates. But the $\mathcal{H}$-matrix construction approach based multilevel clustering is a simpler approach and it also can be used to construct more efficient $\mathcal{H}$-inverse preconditioners.

CHAPTER 4
$\mathcal{H}$-MATRIX PRECONDITIONERS FOR SADDLE-POINT SYSTEMS

In this section, we present the scheme to use our algebraic $\mathcal{H}$-matrix construction combine with $\mathcal{H}$-matrix arithmetic to solve systems of saddle point type arising from meshfree discretization of partial differential equations [20].

Meshfree methods are suitable for solving problems on irregular domains, avoiding the use of a mesh. To deal with the boundary conditions, Lagrange multipliers approach can be used which results in a sparse, symmetric, and indefinite system of saddle-point type.

Due to the indefiniteness and often poor spectral properties, saddle point problems represent a significant challenge for solver developers. In recent years numerous solution techniques have been proposed for this type of systems. Though no single best method exists, for some important classes of problems, very effective methods have been developed. A comprehensive survey [9] reviews a large selection of solution methods, including direct solvers, stationary iterative methods, Krylov subspace solvers, preconditioners, and multilevel methods, with an emphasis on iterative methods.

In [50] a preconditioner based on smoothed Algebraic Multigrid (AMG) is proposed. The $\mathcal{H}$-matrix based block preconditioners are discussed in [16, 17] to solve saddle point systems arising in Oseen equations and Stokes equations. In [13] a joint approach is presented, which computes an approximate $\mathcal{H}$-matrix LU-factorization of the matrix using domain-decomposition clustering with an additional local pivoting

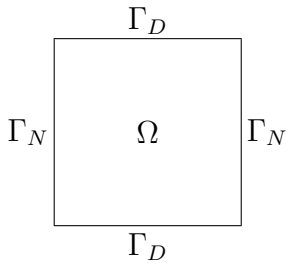
strategy to order the index set. In [55] we presented an algebraic method to construct an LU based preconditioner for the saddle-point system obtained by meshfree methods, which combines the multilevel clustering method with $\mathcal{H}$-matrix arithmetic. But the corresponding preconditioner has both $\mathcal{H}$-matrix and sparse matrix subblocks. In this chapter we present the refined method [20] to construct a pure $\mathcal{H}$-matrix preconditioner for saddle point problem. We compare the new method with the old method [55], JOR, and smoothed algebraic multigrid methods (AMG) [50]. The numerical results show that the new preconditioner outperforms the preconditioners based on the other methods.

4.1 Model Problem

The model problem is a second-order partial differential equation defined on a domain $\Omega \subset \mathbf{R}^2$ [50]:

$$\begin{cases} -\nabla^2 u(\boldsymbol{x}) = f(\boldsymbol{x}), & \boldsymbol{x} \in \Omega \\ u(\boldsymbol{x}) = g(\boldsymbol{x}), & \boldsymbol{x} \in \Gamma_D \\ (\partial u/\partial n)(\boldsymbol{x}) = h(\boldsymbol{x}), & \boldsymbol{x} \in \Gamma_N \end{cases} \tag{4.1}$$

where $\Gamma_D \cup \Gamma_N$ is the boundary of Ω. For our numerical tests we let the domain Ω be $(0,1) \times (0,1) \subset \mathbb{R}^2$ with the boundary Γ_D and Γ_N as shown below.



A meshfree scheme, based on the Reproducing Kernel Partical Method (RKPM) [26], is used to discretize the continuous problem (4.1). RKPM approximates a function $u(\boldsymbol{x})$ by $u^h(\boldsymbol{x}) = \sum_{i=1}^{NP} u_i \Psi_i(\boldsymbol{x})$, where NP stands for the number of particles, Ψ_i is the basis function for a particle i, and u_i is the coefficient of the basis function Ψ_i. Ψ_i is usually constructed by the product of a given kernel function $\Phi_a(\boldsymbol{x} - \boldsymbol{x}_i)$ and a correction function $C(\boldsymbol{x}; \boldsymbol{x} - \boldsymbol{x}_i)$. A correction function is chosen to ensure that for any d dimensional problem the discrete reproducing kernel condition is satisfied:

$$\boldsymbol{x}^\alpha = \sum_{i=1}^{NP} \Psi_i(\boldsymbol{x})\, \boldsymbol{x}_k^\alpha, \tag{4.2}$$

where $\boldsymbol{x}^\alpha = x_1^{\alpha_1} x_2^{\alpha_2} \cdots x_d^{\alpha_d}$, and the relationship holds for all non-negative integer vectors α where $|\alpha| := \sum_{k=1}^{d} \alpha_k \leq p$ for some pre-determined p. We let

$$\Psi_i(\boldsymbol{x}) = C(\boldsymbol{x}; \boldsymbol{x} - \boldsymbol{x}_i)\, \Phi_a(\boldsymbol{x} - \boldsymbol{x}_i). \tag{4.3}$$

The correction function $C(\boldsymbol{x}; \boldsymbol{x} - \boldsymbol{x}_i)$ is then determined by (4.2), which leads to the equation

$$C(\boldsymbol{x}; \boldsymbol{x} - \boldsymbol{x}_i) = H(\boldsymbol{x} - \boldsymbol{x}_i)^T R(\boldsymbol{x})^{-1} H(0), \tag{4.4}$$

where $H(\boldsymbol{s}) = [\, \boldsymbol{s}^\alpha \mid |\alpha| \leq p \,]$, and $R(\boldsymbol{x}) = \sum_{i=1}^{NP} H(\boldsymbol{x} - \boldsymbol{x}_i) H(\boldsymbol{x} - \boldsymbol{x}_i)^T\, \Phi_a(\boldsymbol{x} - \boldsymbol{x}_i)$. The smoothness of the kernel functions $\Phi_a(\boldsymbol{x} - \boldsymbol{x}_i)$ determines the smoothness of the basis functions. Provided the points $\boldsymbol{x}_i$ are chosen appropriately, the order of convergence of the method is $p + 1$ [7, 25].

The Φ_a functions are typically chosen to be tensor products of B-splines: $\Phi_a(\boldsymbol{x} - \boldsymbol{x}_i) = \prod_{k=1}^{d} \varphi((x_k - (\boldsymbol{x}_i)_k)/a_k)$ where φ is a standard B-spline function.

To apply the above RKPM to the model problem (4.1), two sets of basis functions are generated separately on the domain Ω and the boundary Γ_D. We have seen how to construct the basis functions Ψ_i over the domain Ω. For the boundary Γ_D, we construct in a similar way a family of basis functions $\widetilde{\Psi}_j$ over Γ_D using points $\widetilde{\boldsymbol{x}}_j \in \Gamma_D$. The Lagrange Multiplier approach is used to handle the essential boundary conditions. A Ritz–Galerkin method is used to discretize the resulting equations which leads to a meshfree linear system $Kx = F$ of saddle-point type:

$$\begin{bmatrix} A & B^T \\ B & 0 \end{bmatrix} \begin{bmatrix} u \\ \lambda \end{bmatrix} = \begin{bmatrix} c \\ d \end{bmatrix}, \tag{4.5}$$

where $A_{ij} = \int_\Omega ((\nabla\Psi_i)^T \nabla\Psi_j + \Psi_i\Psi_j)\, d\boldsymbol{x}$, $B_{ij} = \int_{\Gamma_D} \widetilde{\Psi}_i \Psi_j dS$, $c_i = \int_\Omega f\Psi_i\, d\boldsymbol{x} + \int_{\Gamma_N} h\Psi_i\, dS$, and $d_i = \int_{\Gamma_D} g\widetilde{\Psi}_i\, dS$. The matrix $K := \begin{bmatrix} A & B^T \\ B & 0 \end{bmatrix}$ in (4.5) is symmetric but indefinite, but the submatrix A is symmetric positive semi-definite. Krylov subspace method GMRES [61] is used to solve the system. Eigenvalue distribution of (4.5) is generally considered unfavourable for solution by Krylov subspace methods. And without preconditioning, Krylov subspace methods tend to converge poorly. It has been observed that a simple transformation as show in (4.6) can be used to obtain an equivalent linear system which is positive semidefinite [8, 9, 35]. The equivalent nonsymmetric system obtained by the transformation is given as follows:

$$\begin{bmatrix} A & B^T \\ -B & 0 \end{bmatrix} \begin{bmatrix} u \\ \lambda \end{bmatrix} = \begin{bmatrix} c \\ -d \end{bmatrix}, \tag{4.6}$$

which is positive semidefinite and Krylov methods can be used to solve the above system.

Even with this way of representing the problem, we still need to ensure that the "inf-sup" or Ladyzhenskaya–Babuška–Brezzi (LBB) condition [22] is satisfied in order to ensure that the results are accurate:

$$\inf_{w} \sup_{z} \frac{\langle w,\, Bz \rangle}{\|w\|\, \|z\|_A} \geq \beta, \qquad \text{where } \|z\|_A = \sqrt{\langle z,\, Az \rangle}. \tag{4.7}$$

In order to expect convergence of the discrete approximations (4.5) we require that (4.7) holds with the same $\beta > 0$ regardless of how fine the discretization is.

With meshfree basis functions and $\widetilde{\Psi}_j = \Psi_j$, unfortunately, the LBB condition (4.7) is rarely satisfied. If the support of a basis function does not intersect the boundary then it causes no problems for the LBB condition. However, if the support of a basis function intersects the boundary just a little, there are severe problems for the LBB condition. Unlike standard finite element methods, there is no mesh to control the supports of the basis functions in meshfree methods, so this is a likely occurrence for meshfree methods. Penalty methods in particular are likely to perform very badly with meshfree methods.

In order to overcome these problems, we use an independently generated set of basis functions on the boundary [50]. These can also be generated as meshfree functions, but using a different family of kernel functions $\widetilde{\Phi}_i$ on the boundary $\partial\Omega$:

$\widetilde{\Phi}_i(\boldsymbol{x}) = \Phi_a(\boldsymbol{x} - \widetilde{\boldsymbol{x}}_i)$ where $\boldsymbol{x} \in \partial\Omega$, and the points $\widetilde{\boldsymbol{x}}_i$ are chosen appropriately from $\partial\Omega$. As for the basis functions Ψ_i over Ω, we construct $\widetilde{\Psi}_i(\boldsymbol{x}) = \widetilde{C}(\boldsymbol{x}, \boldsymbol{x} - \widetilde{\boldsymbol{x}}_i)\,\widetilde{\Phi}_i(\boldsymbol{x})$ where $\widetilde{C}(\boldsymbol{x}, \boldsymbol{x} - \widetilde{\boldsymbol{x}}_i) = H(\boldsymbol{x} - \widetilde{\boldsymbol{x}}_i)^T \widetilde{R}(\boldsymbol{x})^{-1} H(0)$ and $\widetilde{R}(\boldsymbol{x}) = \sum_{i=1}^{NBP} H(\boldsymbol{x} - \widetilde{\boldsymbol{x}}_i) H(\boldsymbol{x} - \widetilde{\boldsymbol{x}}_i)^T\, \widetilde{\Phi}_a(\boldsymbol{x} - \widetilde{\boldsymbol{x}}_i)$. The size of the supports of the boundary basis functions $\widetilde{\Psi}_j$ should not be small in comparison with the size of the supports of the basis functions Ψ_i on Ω.

4.2 $\mathcal{H}$-matrix Preconditioners

To speed up the convergence of Krylov subspace methods (e.g. GMRES), preconditioners can be used. The conventional preconditioners like Jacobi, incomplete factorizations, sparse approximate inverses can not be applied directly to the saddle point systems, since the indefiniteness and lack of diagonal dominance make these preconditioners often unsatisfactory. Instead, the construction of high-quality preconditioners need to exploit the block structure of the problems, with knowledge of the problems and structure of the various blocks.

In this section, we present our $\mathcal{H}$-matrix preconditioners construction approach which are based on block LDU-factorization [20]. Compared to the approaches in [13, 16, 17], our approach is purely algebraic.

Since A is positive definite, the saddle point problem in (4.5) admits the following block LDU factorization:

$$K = \begin{bmatrix} A & B^T \\ B & 0 \end{bmatrix} = \begin{bmatrix} L_1 & 0 \\ L_2 & L_3 \end{bmatrix} \begin{bmatrix} I & 0 \\ 0 & -I \end{bmatrix} \begin{bmatrix} L_1^T & L_2^T \\ 0 & L_3^T \end{bmatrix}, \tag{4.8}$$

where L_1, L_1^T are the Cholesky factors of A. L_2 can be obtained by solving the

following lower triangular system: $L_1 L_2^T = B$, and then L_3 can be obtained by LU factorization of the Schur complement: $L_3 L_3^T = L_2 L_2^T$. Though K is sparse, its exact LU factors obtained by (4.8) can be dense and expensive to compute. To reduce the computational complexity of factorization, we use $\mathcal{H}$-matrix technique. First we reorder and represent K in the $\mathcal{H}$-matrix format while the block structure of K is maintained. Then a $\mathcal{H}$-matrix version of block LDU factorization is performed and the obtained LU factors are in the $\mathcal{H}$-matrix format. These $\mathcal{H}$-LU factors are not exact, but they can be used as preconditioners in Krylov subspace methods and they are much cheaper to compute with optimal computational complexity of $O(nlog^{\alpha}n)$ with a moderate parameter α [21].

To represent K in the $\mathcal{H}$-matrix format while maintain the block structure of K, we need to represent subblock A and B into $\mathcal{H}$-matrix format respectively. Let I denote the row and column index set of the submatrix A and J denote the row index set of B, while A and B share the same column index set I. To build an $\mathcal{H}$-matrix over A, the algebraic approach based on multilevel clustering as introduced in Chapter 3 is applied directly to A. First, a sequence of coarse graphs is built based on the multilevel clustering method; then we can construct an index cluster tree T_I and a permutation matrix P_A. Combining the hierarchy of coarse graphs and T_I we get an $\mathcal{H}$-matrix $A_{\mathcal{H}}$ over the reordered matrix $\widetilde{A} = P_A A P_A^T$. To build an $\mathcal{H}$-matrix over B, we can not apply the above method directly to B, since B is not symmetric and has different row and column index sets. Its column index set I is same as the column index set of A, so we use T_I, obtained in the step to build the $\mathcal{H}$-matrix over

A, as the column index cluster tree of B. To build T_J over the row index set J of B, we apply the algebraic $\mathcal{H}$-matrix construction approach to the matrix graph with weights given by BB^T, which also gives a permutation matrix P_B. Dense rows of B would result in BB^T being a dense matrix. However, dense rows usually do not indicate strong local connections between indices in J, and so can usually be safely ignored for constructing T_J. With T_I and T_J available, the block cluster tree $T_{J\times I}$ for $\widetilde{B} = P_B B P_A^T$ is built by the Algorithm 4.1. Since in our model problem, the size of

Algorithm 4.1 Construction of the block cluster tree $T_{J\times I}$

1: The root of $T_{J\times I}$ is $J \times I$.

2: For each node $s \times t$, if its corresponding block has only zero entries, then it is an Rk-matrix leaf node in $T_{J\times I}$;

else if $\#s > N_s$ and $\#t > N_s$, then the children nodes of $s \times t$ are defined as:

$S(s \times t) = \{j \times i | j \in S(s) \text{ and } j \in T_J; i \in S(t) \text{ and } i \in T_I\}$;

else if $\#s \leq N_s$ and $\#t > N_s$ then its children are:

$S(s \times t) = \{s \times i | i \in S(t) \text{ and } i \in T_I\}$;

else if $\#s \leq N_s$ and $\#t \leq N_s$ then $s \times t$ is a leaf represented by a full matrix;

3: Repeat the above process to each non-leaf node.

the row index set J is much smaller than the column index set I, in the above process to build the block cluster tree, we stop partitioning the row index set J while continue partitioning the column index set I until it is small enough (e.g. $\leq N_s$). In this way

the leaves of the obtained $T_{J\times I}$ are almost square, which speeds up computation as shown in our experiment. The $\mathcal{H}$-matrix representation of K obtained by the above process is given as follows:

$$K_{\mathcal{H}} = \begin{bmatrix} P_A & 0 \\ 0 & P_B \end{bmatrix} K \begin{bmatrix} P_A & 0 \\ 0 & P_B \end{bmatrix}^T = \begin{bmatrix} A_{\mathcal{H}} & B_{\mathcal{H}}^T \\ B_{\mathcal{H}} & C_{\mathcal{H}} \end{bmatrix}. \tag{4.9}$$

Here $C_{\mathcal{H}}$ is an $\mathcal{H}$-matrix obtained by the block tree structure of $T_{J\times J}$ and it only contains zero entries. All the leaves of $C_{\mathcal{H}}$ are full matrices, since $C_{\mathcal{H}}$ is a much smaller matrix compared to $A_{\mathcal{H}}$ and $B_{\mathcal{H}}$, and the experiment results show that it gives better performance than representing some off-diagonal blocks in the R-kmatrix format.

After building $A_{\mathcal{H}}$, $B_{\mathcal{H}}$, and $C_{\mathcal{H}}$ we compute $\mathcal{H}$-LU factors for

$$K_{\mathcal{H}} \approx \begin{bmatrix} L_{1\mathcal{H}} & 0 \\ L_{2\mathcal{H}} & -L_{3\mathcal{H}} \end{bmatrix} \begin{bmatrix} L_{1\mathcal{H}}^T & L_{2\mathcal{H}}^T \\ 0 & L_{3\mathcal{H}}^T \end{bmatrix}, \tag{4.10}$$

in the following process:

1. Since $A_{\mathcal{H}}$ is symmetric, positive, and semi-definite and $A_{\mathcal{H}} \approx L_{1\mathcal{H}} L_{1\mathcal{H}}^T$, we employ $\mathcal{H}$-Cholesky factorization to $A_{\mathcal{H}}$ and obtain $L_{1\mathcal{H}}$ and $L_{1\mathcal{H}}^T$.

2. Since $B_{\mathcal{H}}^T \approx L_{1\mathcal{H}} L_{2\mathcal{H}}^T$, we use an $\mathcal{H}$-matrix lower triangular solve to get $L_{2\mathcal{H}}$.

3. Since $L_{3\mathcal{H}} L_{3\mathcal{H}}^T \approx L_{2\mathcal{H}} L_{2\mathcal{H}}^T$, first the $\mathcal{H}$-matrix multiplication is called to get the product $L_{2\mathcal{H}} L_{2\mathcal{H}}^T$, which is represented in $\mathcal{H}$-matrix format. Then $\mathcal{H}$-LU factorization is called to factor the product and generate $L_{3\mathcal{H}}$, which is induced by the lower triangular part of $C_{\mathcal{H}}$.

Compared to the above process, In the approach proposed in [55] L_1 is obtained by $\mathcal{H}$-Cholesky factorization, while L_2 is obtained by sparse matrix operations. The matrix L_3 is the ordinary LU factorization of the Schur complement and as the size of the problem increases, the time to compute L_2 and L_3 contributes a significant part to the factorization time.

4.3 Experimental Results

In this section, we show the numerical results using the $\mathcal{H}$-matrix preconditioners obtained by the process in Section 4.2 to solve the saddle-point system (4.6).

In our experiments, the numbers of the basis functions for the domain Ω are $N_\Omega = 1600$, 6400, 25600, 102400 and the corresponding numbers of boundary basis functions are $N_\Gamma = 80$, 160, 320, 640. Thus the total problem sizes are $n = 1680$, 6560, 25920, and 103040 respectively.

We use $\mathcal{H}$-matrix arithmetic with adaptive rank to obtain the $\mathcal{H}$-LU factors. The parameter to control the accuracy of $\mathcal{H}$-matrix arithmetic are set as $\alpha = 0.0625$. We also set minimum block size $N_s = 40$.

The GMRES iteration stops when the original residuals are reduced by the factor of 10^{-12}. The convergence rate a, defined as the average decreasing speed of residuals in each iteration, can be obtained by solving the equation: $a^t = 10^{-12}$, where t is the number of iterations.

In the experiments we compare the performance of four preconditioners in GMRES: JOR preconditioners[29], AMG preconditioners [50], the $\mathcal{H}$-matrix method

in [55], and the $\mathcal{H}$-LU preconditioners described in Section 4.2.

The results are plotted in log-log scale. All the experiments were performed on a Dell workstation with dual processor-Xeon 2.4GHz clock speed, and 1GB memory.

Figure 4.1 shows the total time (the time of building $\mathcal{H}$-matrices, building the preconditioners and GMRES iterations). 'HEM-H-CH' indicates the factorization method in [55] and 'HEM-HMAT-LU' indicates the factorization scheme described in Section 4.2.

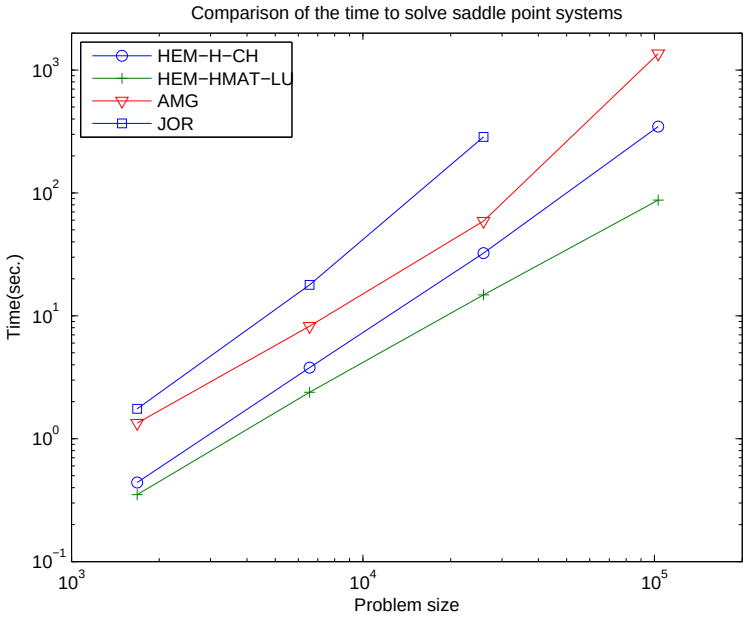


Figure 4.1: Comparison of the total time to solve the saddle-point system with JOR, AMG, and $\mathcal{H}$-LU preconditioners.

Figure 4.2 shows the average convergence rates of JOR, AMG and $\mathcal{H}$-LU preconditioners.

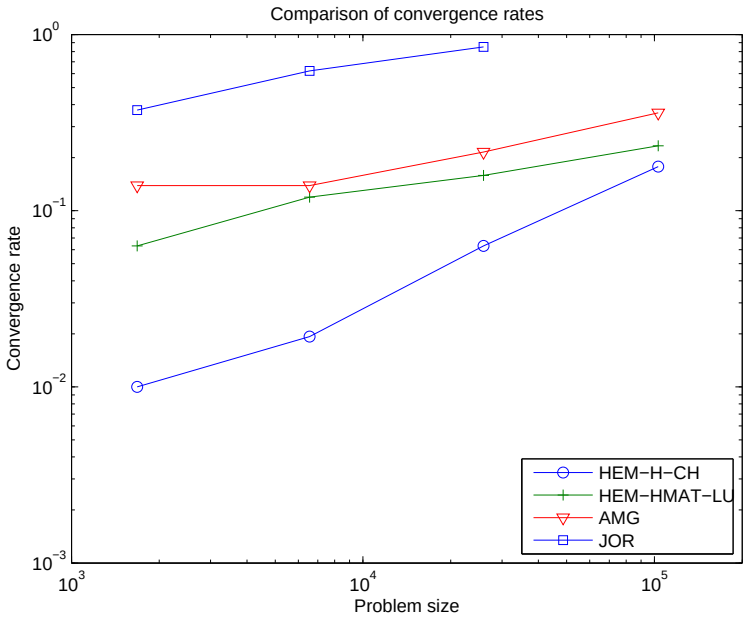


Figure 4.2: Comparison of the convergence rates of GMRES with JOR, AMG and $\mathcal{H}$-LU preconditioners.

With respect to the total running time and the convergence rates, $\mathcal{H}$-matrices based preconditioners give better performance than JOR and AMG preconditioners. When the problem size is bigger than 10^4 the time of 'HEM-H-CH' shows a sharp increase. As the problem size increases to around 10^5, the average convergence rates of 'HEM-H-CH' and 'HEM-HMAT-LU' get very close to each other.

Figure 4.3 and Figure 4.4 give more detailed comparisons between the $\mathcal{H}$-matrix based preconditioners as to the stages of building LU factors and GMRES iterations. Note that the time needed for building the cluster index and block cluster trees is a small fraction of the time needed to build the preconditioners.

Figure 4.3 shows that the time to build $\mathcal{H}$-matrix based preconditioner 'HEM-HMAT-LU' is less than and increases slower than that of 'HEM-H-CH'. The reason is

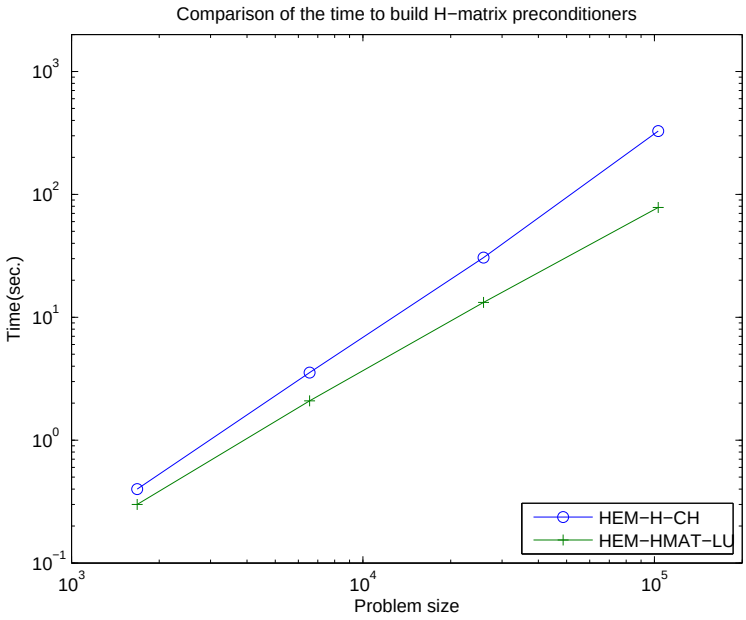


Figure 4.3: Comparison of the time to build the $\mathcal{H}$-matrix based preconditioners for the saddle point problem.

that the calculation of 'HEM-HMAT-LU' only involves $\mathcal{H}$-matrix arithmetic, which maintains the cost to almost optimal. But the calculation of 'HEM-H-CH' involves sparse matrix operations, which does not have the properties provided by the $\mathcal{H}$-matrix arithmetic.

Figure 4.4 shows the time of GMRES iteration for the $\mathcal{H}$-matrix based preconditioners. By representing all the subblocks of LU factors in the $\mathcal{H}$-matrix format, 'HEM-HMAT-LU' manages to maintain the sparsity of LU factors, which reduces the computational complexity in the factorization stage and in the GMRES iteration stage. Overall, both $\mathcal{H}$-matrices based preconditioners outperform JOR and AMG preconditioners. With the increase of the problem size 'HEM-HMAT-LU' shows overall better performance than 'HEM-H-CH'.

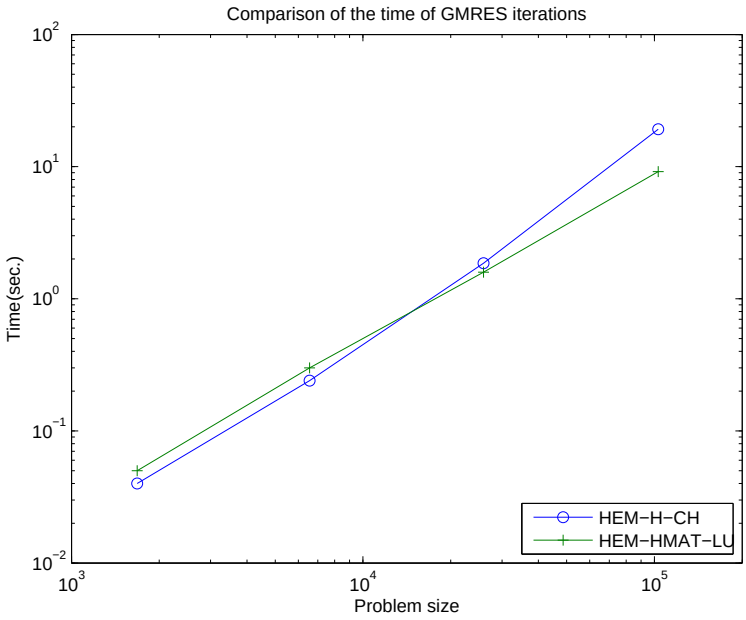


Figure 4.4: Comparison of the time of GMRES iterations with the $\mathcal{H}$-matrix based preconditioners.

Comparisons are also made between two iterative methods: GMRES and MINRES, applied to the symmetric indefinite form of the saddle point problem. On one hand, MINRES uses short recurrences, but may use more iterations than GMRES applied to the unsymmetric form. As can be seen in Figure 4.5, preconditioned MINRES actually gives slightly slower convergence than preconditioned GMRES for this problem.

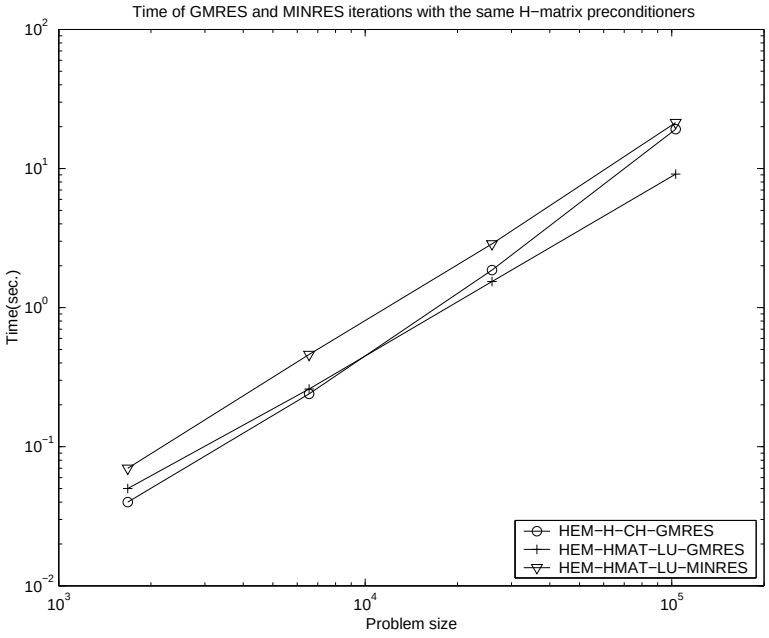


Figure 4.5: Comparison of the time of GMRES and MINRES iterations for solving the saddle point problem.

CHAPTER 5
OTHER APPLICATIONS

In this chapter we present the numerical results of applying $\mathcal{H}$-matrix preconditioners to solve problems arising in other fields.

5.1 Optimal Control Problem

In this section we consider the finite time linear-quadratic optimal control problems governed by parabolic partial differential equations. To solve these problems, in [62] the parabolic partial differential equations are discretized by finite element methods in space and by θ-scheme in time; the cost function J to be minimized is discretized using midpoint rule for the state variable and using piecewise constant for the control variable in time; Lagrange multipliers are used to enforce the constraints, which result a system of saddle point type; then iterative methods with block preconditioners are used to solve the system.

We use the discretization process described in [62] and the $\mathcal{H}$-matrix preconditioning technique to solve the problem. First we apply the algebraic $\mathcal{H}$-matrix construction approach to represent the system in $\mathcal{H}$-matrix format; then $\mathcal{H}$-LU factorization is adapted to the block structure of the system to compute the approximate $\mathcal{H}$-LU factors; at last, these factors are used as preconditioners in iterative methods. The numerical results show that the $\mathcal{H}$-matrix preconditioned approach is competitive and effective to solve the above optimal control problem.

5.1.1 Model Problem

The model problem [62] is to minimize the following quadratic cost function:

$$\begin{aligned} J(z(u),u) :=& \frac{q}{2}\|z(v)-z_*\|^2_{L^2(t_0,t_f;L^2(\Omega))} + \frac{r}{2}\|v\|^2_{L^2(t_0,t_f;\Omega)} \\ &+ \frac{s}{2}\|z(v)(t_f,x)-z_*(t_f,x)\|^2_{L^2(\Omega)}, \end{aligned} \tag{5.1}$$

under the constraint of the state equation:

$$\begin{cases} \partial_t z + \mathcal{A}z &= \mathcal{B}v, \quad t \in (t_0,t_f) \\ z(t,\partial\Omega) &= 0, \\ z(t_0,\Omega) &= 0, \end{cases} \tag{5.2}$$

where the state variable $z \in Y = H_0^1(\Omega)$ and the control variable $v \in U = L^2(t_0,t_f;\Omega)$. $\mathcal{B}$ is an operator in $\mathcal{L}(L^2(t_0,t_f;\Omega), L^2(t_0,t_f;Y'))$ and $\mathcal{A}$ is an uniformly elliptic linear operator from $L^2(t_0,t_f;Y)$ to $L^2(t_0,t_f;Y')$. The state variable z is dependent on v and z_* is a given target function.

5.1.1.1 Discretization in Space

The system is first discretized in space by fixing the time variable t. Considering the discrete subspace $Y_h \in Y$ and $U_h \in U$, the discretized weak form of (5.2) is given as:

$$(\dot{z}_h(t),\eta_h) + (\mathcal{A}z_h(t),\eta_h) = (\mathcal{B}u_h(t),\eta_h), \quad \forall \eta_h \in Y_h \text{ and } t \in (t_0,tf). \tag{5.3}$$

Let$\{\phi_1,\phi_2,..,\phi_n\}$ be a basis of Y_h and $\{\psi_1,\psi_2,..,\psi_m\}$ be a basis of U_h, where $m \leq n$. Apply the finite element methods to (5.3), we obtain the following system of ordinary differential equations:

$$M\dot{y} + Ay = Bu, \quad t \in (t_0,t_f). \tag{5.4}$$

Here $A_{i,j} = (\mathcal{A}\phi_j, \phi_i)$ is a stiffness matrix, $M_{i,j} = (\phi_j, \phi_i)$ and $R_{i,j} = (\psi_j, \psi_i)$ are mass matrices, and $B_{i,j} = (\mathcal{B}\psi_i, \phi_j)$. The semi-discrete solution is $z_h(t,x) = \sum_i y_i(t)\phi_i(x)$ with control function $u_h(t,x) = \sum_i u_i(t)\psi_i(x)$.

We can apply the analogous spatial discretization to the cost function (5.1), and obtain:

$$J(y,u) = \int_{t_f}^{t_0} e(t)^T Q(t) e(t) + u(t)^T R(t) u(t)\, dt + e(t_f)^T C(t) e(t_f), \tag{5.5}$$

where $e(\cdot) = y(\cdot) - y_*(\cdot)$ is the difference between the state variable and the given target function.

5.1.1.2 Discretization in Time

After spatial discretization, the original optimal problem is transferred into the problem to minimize the equation of (5.5) under the constraint of n ordinary differential equations of (5.4). θ-scheme is used to discretize the above problem.

First the time scale is subdivided into l intervals of length $\tau = (t_f - t_0)/l$. Let $F_0 = M + \tau(1-\theta)A$ and $F_1 = M - \tau\theta A$. The discretization of equation (5.4) is given by:

$$E\mathbf{y} + N\mathbf{u} = \mathbf{f}, \tag{5.6}$$

where

$$E = \begin{bmatrix} -F_1 & & & \\ & \ddots & \ddots & \\ & & F_0 & -F_1 \end{bmatrix}, \qquad N = \tau \begin{bmatrix} B & & \\ & \ddots & \\ & & B \end{bmatrix}, \quad \mathbf{y} \approx \begin{bmatrix} y(t_1) \\ \vdots \\ y(t_n) \end{bmatrix}, \quad \text{etc} \ . \tag{5.7}$$

Then the cost function (5.5) is discretized by using piecewise linear functions to approximate the state variable and piecewise constant functions to approximate the control variable, which results the following discrete form of (5.5):

$$J(\mathbf{y}, \mathbf{u}) = \mathbf{u}^T G \mathbf{u} + \mathbf{e}^T K \mathbf{e}, \tag{5.8}$$

where $\mathbf{e} = \mathbf{y} - \mathbf{y}_*$ and the target trajectory $z_*(t, x) \approx z_{*,h}(t, x) = \sum_i (y_*)_i(t)\phi_i(x)$. A Lagrange multiplier vector p is introduced to enforce the constraint of (5.6), and we have the Lagrangian:

$$\mathcal{L}(\mathbf{y}, \mathbf{u}, \mathbf{p}) = \frac{1}{2}(\mathbf{u}^T G \mathbf{u} + \mathbf{e}^T K \mathbf{e}) + \mathbf{p}^T (E\mathbf{y} + N\mathbf{u} - \mathbf{f}). \tag{5.9}$$

To find $\mathbf{y}$, $\mathbf{u}$ and $\mathbf{p}$ where $\nabla\mathcal{L}(\mathbf{y}, \mathbf{u}, \mathbf{p}) = 0$ in (5.9), we need to solve the following system:

$$\begin{bmatrix} K & 0 & E^T \\ 0 & G & N^T \\ E & N & 0 \end{bmatrix} \begin{bmatrix} \mathbf{y} \\ \mathbf{u} \\ \mathbf{p} \end{bmatrix} = \begin{bmatrix} M\mathbf{y}_* \\ 0 \\ \mathbf{f} \end{bmatrix}, \tag{5.10}$$

which is of saddle point type.

5.1.2 Hierarchical-Matrix Preconditioner

The construction of $\mathcal{H}$-matrix preconditioners for the system (5.10) is based on the block LU factorization.

First the matrix in (5.10) is converted to an $\mathcal{H}$-matrix. Since the nonzero entries of each subblock are centered around the diagonal, we apply $\mathcal{H}$-matrix construction approach based on bisection approach to submatrix K, G, E and N respectively.

Then we obtain the following $\mathcal{H}$-matrix, which is on the left side of the equation, where subscript $_{\mathcal{H}}$ indicates a block in $\mathcal{H}$-matrix format:

$$\begin{bmatrix} K_{\mathcal{H}} & 0 & E_{\mathcal{H}}^T \\ 0 & G_{\mathcal{H}} & N_{\mathcal{H}}^T \\ E_{\mathcal{H}} & N_{\mathcal{H}} & 0 \end{bmatrix} = \begin{bmatrix} L1_{\mathcal{H}} & 0 & 0 \\ 0 & L2_{\mathcal{H}} & 0 \\ M1_{\mathcal{H}} & M2_{\mathcal{H}} & L3_{\mathcal{H}} \end{bmatrix} \begin{bmatrix} U1_{\mathcal{H}} & 0 & M1_{\mathcal{H}}^T \\ 0 & U3_{\mathcal{H}} & M2_{\mathcal{H}}^T \\ 0 & 0 & U3_{\mathcal{H}} \end{bmatrix}. \tag{5.11}$$

The block cluster tree $T_{I\times I}$ of $L1_{\mathcal{H}}$, $L2_{\mathcal{H}}$, $M1_{\mathcal{H}}$, and $M2_{\mathcal{H}}$ is same as the block cluster tree structure of $K_{\mathcal{H}}$, $G_{\mathcal{H}}$, $E_{\mathcal{H}}$, and $N_{\mathcal{H}}$ respectively. The block cluster tree structure of $L3_{\mathcal{H}}$ is based on the block tree structure of $E_{\mathcal{H}}$; the block tree of $L3_{\mathcal{H}}$ is symmetric; the tree structure of the lower-triangular of $L3_{\mathcal{H}}$ is same as the tree structure of the lower-triangular of $E_{\mathcal{H}}$; the tree structure of the upper-triangular of $L3_{\mathcal{H}}$ is the transpose of the tree structure of the lower triangular. $L1_{\mathcal{H}}$ and $L2_{\mathcal{H}}$ are obtained by applying $\mathcal{H}$-Cholesky factorization to $K_{\mathcal{H}}$ and $G_{\mathcal{H}}$: $K_{\mathcal{H}} = L1_{\mathcal{H}} *_{\mathcal{H}} U1_{\mathcal{H}}$ and $G_{\mathcal{H}} = L2_{\mathcal{H}} *_{\mathcal{H}} U2_{\mathcal{H}}$. Then using the $\mathcal{H}$-matrix upper triangular solve, we can get $M1_{\mathcal{H}}$ by solving $M1_{\mathcal{H}}U1_{\mathcal{H}} = E_{\mathcal{H}}$. $M1_{\mathcal{H}}$ has the same block tree as $E_{\mathcal{H}}$. In the same way we can compute $M2_{\mathcal{H}}$, which has the same block cluster tree structure as $N_{\mathcal{H}}$. At last we construct the block cluster tree for $L3_{\mathcal{H}}$ and then apply $\mathcal{H}$-LU factorization to get $L3_{\mathcal{H}}$ by solving the equation: $L3_{\mathcal{H}}U3_{\mathcal{H}} = M1_{\mathcal{H}} *_{\mathcal{H}} M1_{\mathcal{H}}^T +_{\mathcal{H}} M2_{\mathcal{H}} *_{\mathcal{H}} M2_{\mathcal{H}}^T$.

5.1.3 Experimental Results

In this section, we present the numerical results of solving the optimal control problem (5.1) constrained by following equations:

$$\begin{cases} \partial_t z - \partial_{xx} z &= v, t \in (0,1), x \in (0,1) \\ z(t,0) &= 0, z(t,1) = 0 \\ z(0,x) &= 0, x \in [0,1] \end{cases}, \tag{5.12}$$

with the target function $z_*(t,x) = x(1-x)e^{-x}$. The parameters in the control function J are $q = 1$, $r = 0.0001$, and $s = 0$.

GMRES iteration stops where the original residuals are reduced by the factor of 10^{-12}. The convergence rate a ia defined as the average decreasing speed of residuals in each iteration. Fixed-rank $\mathcal{H}$-matrix arithmetic is used and we set the rank of each Rk-matrix block to be ≤ 2. The tests are performed on a Dell workstation with AMD64 X2 Dual Core Processors (2GHz) and 3GB memory.

Table 5.1 shows the time to compute the different parts of the $\mathcal{H}$-LU factors and the time of GMRES iterations (in second). n is the size of the problem, $n1$ and $n2$ are the numbers of rows of K and G respectively. Based on Table 5.1, the time to compute $L3_{\mathcal{H}}$ contributes the biggest part of the total time to set up the preconditioner.

Figure 5.1 shows the convergence rate of the $\mathcal{H}$-LU preconditioned GMRES. Figure 5.2, plotted on a log-log scale, shows the time to build $\mathcal{H}$-matrix preconditioners and the time of GMRES iterations.

Based the results, we can see that $\mathcal{H}$-LU speeds up the convergence of GMRES

Table 5.1: Time of computing $\mathcal{H}$-LU factors and GMRES iterations (in second).

n(n1/n2)	$L1_{\mathcal{H}}$	$L2_{\mathcal{H}}$	$M1_{\mathcal{H}}$	$M2_{\mathcal{H}}$	$L3_{\mathcal{H}}$	time of GMRES iterations	number of GMRES iterations
592(240/112)	0	0	0.01	0	0.01	0	1
2464(992/480)	0.01	0	0.01	0	0.04	0	1
10048(4032/1984)	0.03	0.01	0.13	0.02	0.39	0.04	1
40576(16256/8064)	0.21	0.06	0.84	0.26	4.13	0.32	3
163072(65280/32512)	1.09	0.42	4.23	1.74	25.12	2.66	6

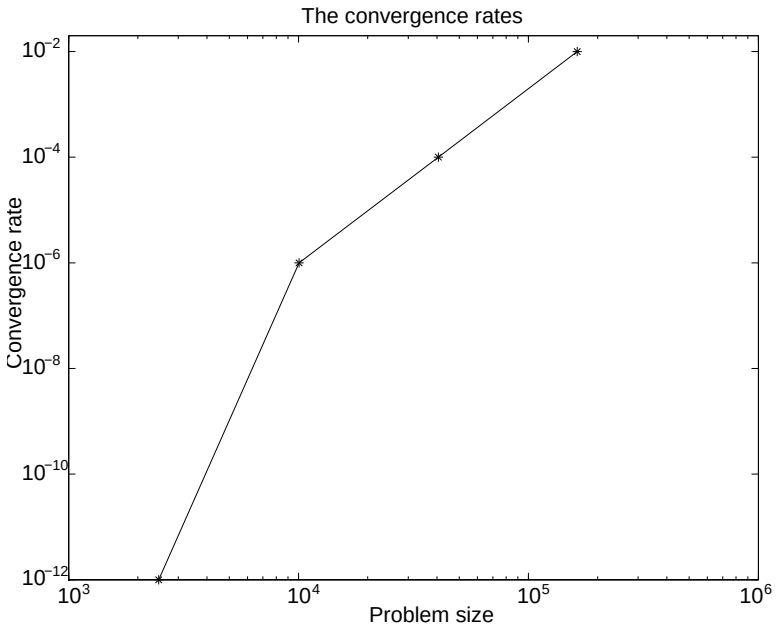


Figure 5.1: The convergence rates of GMRES iterations.

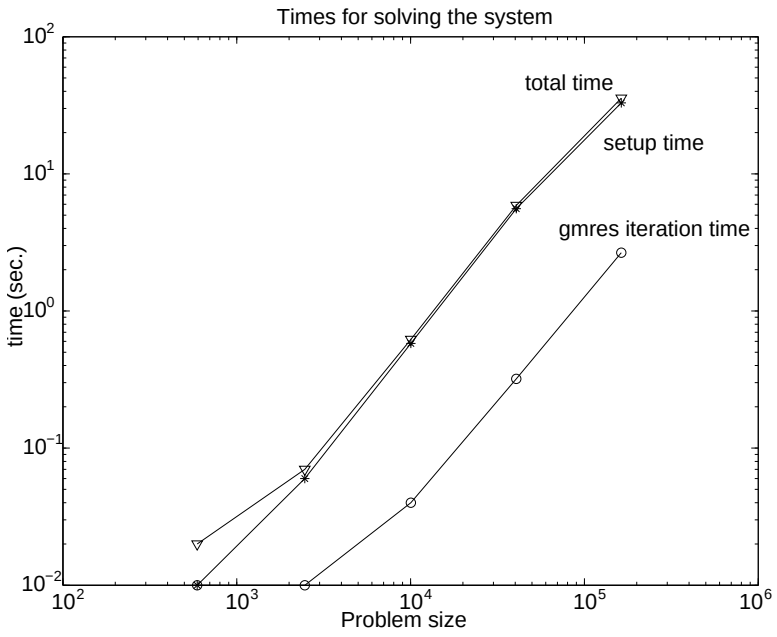


Figure 5.2: The total time to solve the system, including the time to build $\mathcal{H}$-matrix preconditioners and GMRES iterations.

iteration significantly. The problem in our implementation is that the time to compute L_3 still consists a significant part of the LU-factorization time.

5.2 Invariant Probability Distribution

The problem considered in this section is to compute the invariant probability distribution p in the dynamic system $x_{t+1} = f(x_t)$, where f is the shift function. To get p results in solving a dense system. Instead of solving the dense system directly, we use algebraic $\mathcal{H}$-matrix construction approach to partition and convert the dense matrix into an $\mathcal{H}$-matrix, which reduces the storage as well as the computational complexity. Then $\mathcal{H}$-matrix arithmetic is applied to the $\mathcal{H}$-matrix to obtain the $\mathcal{H}$-matrix-LU factors, which are used as preconditioners in iterative methods. The

numerical results show that the $\mathcal{H}$-LU preconditioners are cheap to calculate, yet they speed up the convergence of GMRES greatly.

5.2.1 Model Problem

The problem is to find the invariant probability distribution p in the following dynamic system:

$$x_{t+1} = f(x_t), x \in [0,1], f(x) \in [0,1]. \tag{5.13}$$

To discretize (5.13), the interval $[0,1]$ is divided into n subintervals of equal length $l = 1/n$. In our case f is defined as $f = \alpha x(1-x)$, where α is a constant. So for each $x \in [x_i, x_{i+1}]$, f maps x to some interval: $f(x) \in [x_j, x_{j+1}]$.

A matrix $A = [a_{ij}]$ can be constructed, where $a_{i,j}$ is the probability that the function f maps $x \in [x_j, x_{j+1}]$ to the interval $[x_i, x_{i+1}]$. The matrix A has the following properties: A is sparse and nonsymmetric, and $\sum_j a_{ij} = 1$. Figure 5.3 shows distribution of nonzero entries of A.

Each entry p_i in the invariant probability distribution vector p is the probability that $x \in [x_i, x_i + 1]$, and $Ap = p$. To obtain p, we need to solve the following system of linear equations:

$$(A - I)p = 0, \text{ where } e^T p = 1, \tag{5.14}$$

where e is a vector of 1's.

The system (5.14) is singular. To avoid solving a singular system, we solve the following nonsingular linear system:

$$(A + ee^T - I)p = e. \tag{5.15}$$

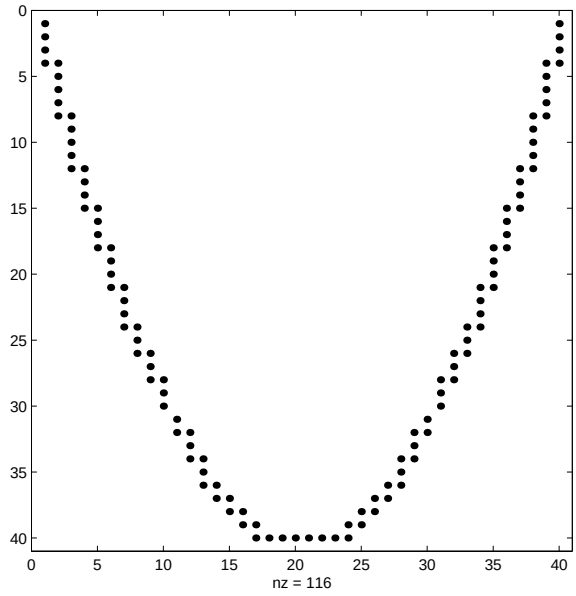


Figure 5.3: The distribution of nonzero entries in A. The black dots represent nonzero entries.

The system (5.15) is nonsingular, but it is full. Most entries of $(A + ee^T - I)$ are 1's, which means that some of its blocks can be represented exactly in Rk-matrix format of rank 1. So we can use iterative methods with $\mathcal{H}$-matrix preconditioners to solve the above system.

5.2.2 $\mathcal{H}$-matrix Construction

To build $\mathcal{H}$-matrix preconditioners for (5.15), first we need to represent matrix $(A + ee^T - I)$ in $\mathcal{H}$-matrix format. $(A + ee^T - I)$ is nonsymmetric, so the algebraic $\mathcal{H}$-matrix construction approaches which are based on the matrix graphs can not be applied directly. We choose the algebraic $\mathcal{H}$-matrix construction based on bisection.

So the process to construct an $\mathcal{H}$-matrix H for (5.15) based on bisection works in the following way: the root of H is $I \times I$; for each node in H if it corresponds to

a block of rank 1, then it is a leaf and the block is represented in Rk-matrix format; otherwise if the number of rows or columns of the block is $\leq N_s$, the node is a leaf and the block is represented in full matrix format; otherwise the block is split into four subblocks of roughly equal size and the node has four children. The above process can represent $A + ee^T - I$ exactly as an $\mathcal{H}$-matrix. Figure 5.4 shows an example of $\mathcal{H}$-matrix representation of the matrix $A + ee^T - I$ with 8 rows and 8 columns. In this example, $Ns = 1$ and the blocks marked by a letter of R represent Rk-matrices of rank 1. The blocks with dots inside represent full matrix blocks.

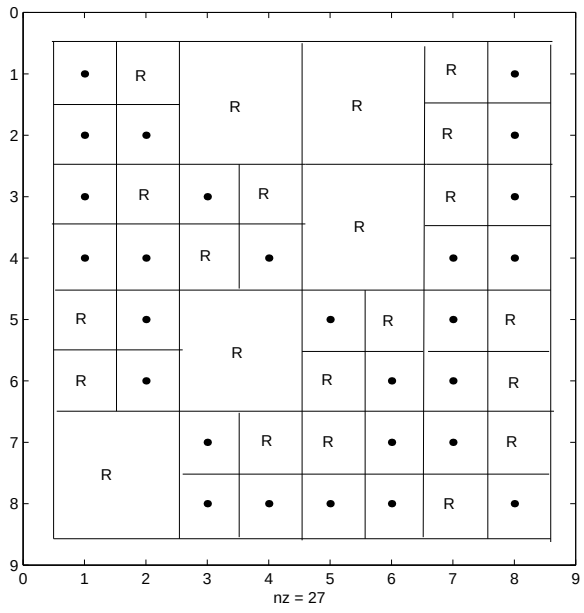


Figure 5.4: An example of $\mathcal{H}$-matrix representation of matrix $A + ee^T - I$. A letter R indicates an Rk-matrix block and a black dot indicates a full matrix block.

5.2.3 Experimental Results

In this section, we present the numerical results of applying the $\mathcal{H}$-matrix preconditioning technique to solve the system (5.15).

To solve the problem, we first represent the matrix of (5.15) in $\mathcal{H}$-matrix format. Then we compute $\mathcal{H}$-LU factors, which are used as preconditioners for GMRES. In our experiment, we use fixed-rank $\mathcal{H}$-matrix arithmetic, since it gives better overall performance than adaptive $\mathcal{H}$-matrix arithmetic. In fixed-rank $\mathcal{H}$-matrix arithmetic, we set $k = 4$, which means the ranks of RK-matrix blocks remain ≤ 4. We also set the constant $Ns = 40$ to control the size of leaf blocks. The sizes of the problems tested are $1024, 8192, 65536$ and 261344. The experiments were carried out on a dual processor computer with 64-bit Athlon 6 4200++ CPUs and 3GB of memory.

To see the computational complexity at each stage, we split the total time needed to solve the problem into two parts: the time to compute $\mathcal{H}$-LU preconditioners (set-up time) and the time of GMRES iterations (GMRES iteration time). Figure 5.5 shows the set-up time, the time of GMRES iterations, and the total time (set-up time + GMRES iteration time). Based on Figure 5.5, the set-up time contributes a major portion of the total time, compared to the time of GMRES iterations. Yet the time to compute $\mathcal{H}$-LU preconditioners increases almost linearly as we increase the size of the problem, even though (5.15) is a dense system. That means $\mathcal{H}$-matrix arithmetic is efficient to compute preconditioners to solve these dense systems. Figure 5.6 shows the convergence rates. As the size of the problem increases, we can see that the convergence rates decrease gracefully.

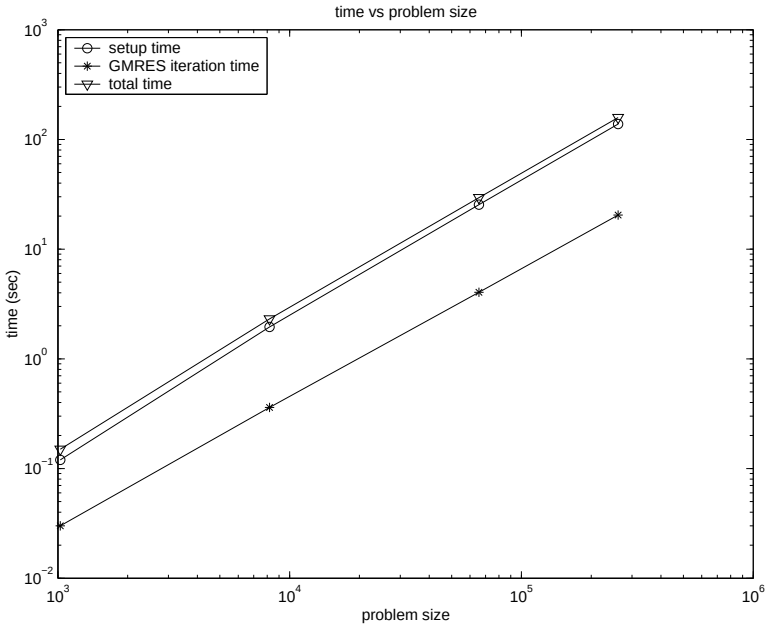


Figure 5.5: The plot of the set-up time, the GMRES iteration time, and the total time. The set-up time contributes the most of the time needed to solve the invariant probability distribution problem.

Based on above results, we can see that $\mathcal{H}$-LU preconditioners are cheap to compute yet they speed up GMRES iterative method greatly.

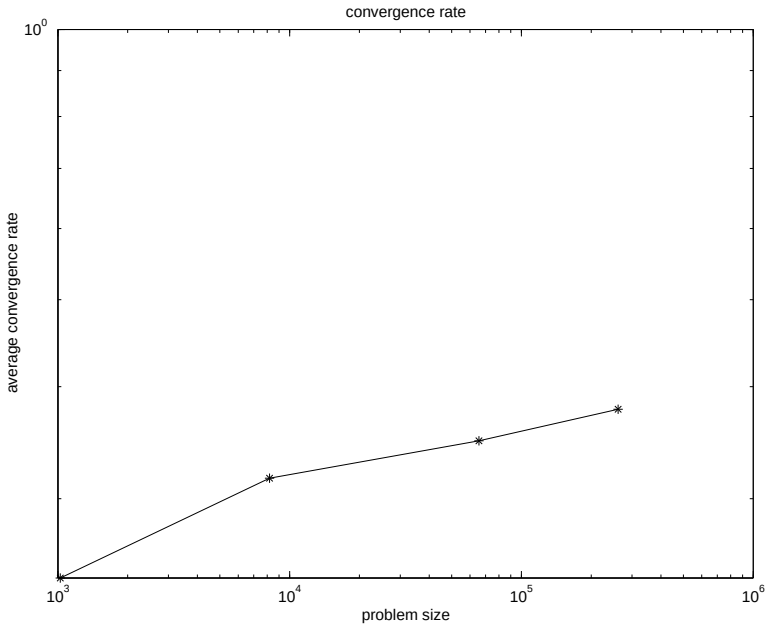


Figure 5.6: The convergence rates of GMRES with the $\mathcal{H}$-LU preconditioners to solve the invariant probability distribution problem.

CHAPTER 6
IMPLEMENTATION

This chapter presents the implementation of the algebraic $\mathcal{H}$-matrix preconditioner technique.

6.1 Overview

Figure 6.1 shows the main modules of the algebraic $\mathcal{H}$-matrix preconditioner technique: first it takes a sparse (or data-sparse) symmetric matrix and converts it into an $\mathcal{H}$-matrix; then it uses $\mathcal{H}$-matrix arithmetic to compute approximate $\mathcal{H}$-matrix inverses, $\mathcal{H}$-matrix LU-factors, or Cholesky facotors; finally these $\mathcal{H}$-inverses, $\mathcal{H}$-LU, or $\mathcal{H}$-Cholesky factors are used as preconditioners in iterative methods.

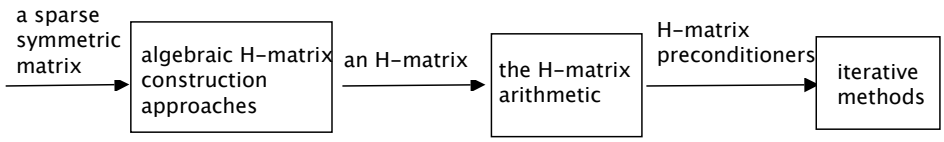


Figure 6.1: The main function modules of the algebraic $\mathcal{H}$-matrix preconditioner technique.

The programming language used to implement the algebraic $\mathcal{H}$-matrix preconditioner technique is C language. To simplify the implementation, we also use the existing Meschach library [63]. Meschach library provides data structures for dense and sparse matrix representation, basic matrix and vector operations, and various

numerical algebraic algorithms. The implementation of $\mathcal{H}$-matrix representation and $\mathcal{H}$-matrix arithmetic follows the description given in [10, 11, 49]. In all, our implementation includes the following five main parts:

- Data structures used to represent $\mathcal{H}$-matrices.
- Implementation of $\mathcal{H}$-matrix arithmetic.
- Implementation of the algebraic $\mathcal{H}$-matrix construction approach based on multilevel graph clustering.
- Iterative methods with various $\mathcal{H}$-matrix preconditioners.

In the following sections, we focus on the implementation of $\mathcal{H}$-matrix representation, the algebraic $\mathcal{H}$-matrix construction approach, and the $\mathcal{H}$-matrix LU factorization.

6.2 $\mathcal{H}$-matrix Representation

The representation of $\mathcal{H}$-matrices includes the representation of full matrix blocks, the representation of Rk-matrix blocks, and the representation of $\mathcal{H}$-matrix block trees.

The full matrix blocks in an $\mathcal{H}$-matrix are represented using *MAT* structure defined in Meschach library:

```
struct
{
  u_int m, n;
  u_int max_m, max_n, max_size;
  Real **me, *base;
} MAT;
```

Here m is the row size and n is the column size of a matrix. Matrix entries are stored in the array *me* in the row major order, where *me[i]* points to the starting position of ith row of the array. To access the matrix entry at ith row and jth column, we use *me[i][j]*.

The structure of Rk-matrix blocks is built on *MAT* structure. For an RK-matrix AB^T, its structure is:

```
struct rkmatrix
{
  int rk;
  int rows, cols;
  MAT *A;
  MAT *B;
};
```

The row size of the Rk-matrix is stored in *rows* and the column size is stored in *cols*. The variable *rk* indicates the rank of an Rk-matrix.

After define the structure of full matrices and Rk-matrices, we can define the structure of an $\mathcal{H}$-matrix, which is based on a tree structure. The structure for an $\mathcal{H}$-matrix called *supermatrix* is defined as follows:

```
struct supermatrix
{
  int type;
  int rows, cols;
  int block_rows, block_cols;
  struct rkmatrix *r;
  MAT *f;
  struct supermatrix **s;
};
```

The size of an $\mathcal{H}$-matrix is given by $rows \times cols$. The variable *block_rows* and the variable *block_cols* are used to store the number of children of the current node in

each row and column. The field *type* indicates the type of a *supermatrix* node. Based on the definition of $\mathcal{H}$-matrices, there are three types of *supermatrix* nodes:

- If $type = 1$, the node is an Rk-matrix leaf. The pointer r points to an *rkmatrix*, which stores the data. The pointer f and the pointer s are all *null* pointers.
- If *type=2*, the node is a full-matrix leaf. The pointer f points a MAT, which stores the data. The pointer r and the pointer s are null pointers.
- If *type=3*, the node is an internal node. The pointer f and the pointer r are *null* pointers. The pointer s is an array of size $block\_rows \times block\_columns$, pointing to the children of the current node.

The following is an example of the array s:

$$\begin{pmatrix} s[0][0] & \dots & s[0][block\_columns-1] \\ \dots & \dots & \dots \\ s[block\_rows-1][0] & \dots & s[block\_rows-1][block\_columns-1], \end{pmatrix} \tag{6.1}$$

, where $s[i][j]$ points an $\mathcal{H}$-matrix. In this way *supermatrix* defines a tree which can be used to represent an $\mathcal{H}$-matrix.

6.3 $\mathcal{H}$-matrix Construction Approach

In our implementation of the algebraic $\mathcal{H}$-matrix construction approach, we use the structure $SPMAT$ defined in Meschach library to represent a graph. The structure $SPMAT$ is used for sparse matrix representation:

```
typedef struct sp_mat
{
```

```
  int m, n, max_m, max_n;
  char flag_col, flag_diag;
  SPROW *row;
  int *start_row;
  int *start_idx;
} SPMAT;
```

The *SPMAT* structure is built on the sparse row structure *SPROW*. Here m and n indicate the size of a matrix graph. If there is an edge between node i and node j, then the corresponding entry in *SPMAT* equals its edge weight.

We use the structure *spnode* to store the clusters and coarser graph $G(V, E)$ obtained by merging the vertices in the same cluster together. The structure *spnode* is a linked list:

```
struct spnode
{
  SPMAT *S;
  int snum;
  IVEC *sons;
  struct spnode *up;
};
```

In the structure, *up* points to the coarser level. The variable *snum* indicates the number of clusters and S points to the graph constructed by merging the vertices of the same clusters together. The clusters built over the graph of the finer level are stored in an integer vector *sons* of type $IVEC$, which is an integer vector defined in Meschach:

```
typedef struct IVEC
{
  u_int dim, max_dim;
  Real *ve;
}
```

The elements of an integer vector are stored in the array ve. The indices of vertices belonging to the same cluster C_i are stored continuously in the integer vector $sons$: $sons->ve[2*i]$ and $sons->ve[2*i+1]$. If a cluster C_i contains just one vertex, then $sons[2*i+1] = -1$.

The function *multilevel_hem* implements the multilevel clustering algorithm, which calls the modified HEM algorithm *build_cluster* to build clusters and calls *spmtrmm_mlt* to build the coarser graph until the size of the obtained graph is small enough. The main body of the function *multilevel_hem* is given as follows:

```
void multilevel_hem(struct spclustertree *spctree)
{
  ......
  //continue building coarse graphes
  while(gptr->m > GMIN){
    //build clusters on the current level
    repeat =
        build_cluster(ptr, gptr, E, group, &groupsize);
    if(repeat == 0){
      break;
    }else{
      level++;
    }
    //allocate memory for the coarser graph
    ptr->S = sp_get(E->m, E->m, SPMIN);
    //construct the coarser graph
    spmtrmm_mlt(E, gptr, ptr->S);
    //create new level
    if(ptr->S->m > GMIN){
      gptr = ptr->S;
      ptr->up =
          (struct spnode*) malloc (sizeof(struct spnode));
      assert(ptr->up != NULL);
      //point to the coarer level
      ptr = ptr->up;
      ptr->snum = 0;
      ptr->sons = iv_get(gptr->m*2);
    } else{
```

```
      break;
    }
  }//end of while
  spctree->level = level;
  ......
}
```

The integer vector *group* holds the vertices with higher priority to be picked up in HEM. The function *build_cluster* implements the modified HEM algorithm, which builds clusters over the sparse matrix graph S. The code of the function *build_cluster* is given as follows:

```
int build_cluster(struct spnode *p, SPMAT *S,
  MAT *E, IVEC *group, int *groupsize)
{
  ......
  //randomly permute the nodes
  randperm(node_state, nodenum);
  ......
  //start building clusters
  *groupsize = 0;
  clusternum = 0;
  idx = 0;
  matched = 0;
  while(idx < nodenum-1 ){
    //if the node is already matched
    if(node_state->ive[node_heap->ive[idx]] == -1){
      idx++;
      continue;
    }
    ro = node_heap->ive[idx];
    max_value = 0;
    len = S->row[ro].len;
    for(j=0; j<len; j++){
      co = S->row[ro].elt[j].col;
      if(co != ro && node_state->ive[co] != -1){
        val = S->row[ro].elt[j].val;
        if(val > max_value){
          max_value = val;
          max_j = node_state->ive[co];
```

```
        }
      }
    }
    //the matched node is found
    if(max_value != 0){
      co = node_heap->ive[max_j];
      p->sons->ive[p->snum*2] = ro;
      p->sons->ive[p->snum*2+1] = co;
      E->me[clusternum][0] = ro;
      E->me[clusternum][1] = co;
      node_state->ive[node_heap->ive[idx]] = -1;
      node_state->ive[node_heap->ive[max_j]] = -1;
      matched = 1;
    }else{
      //the matched node is not found.
      p->sons->ive[p->snum*2] = ro;
      p->sons->ive[p->snum*2+1] = -1;
      E->me[clusternum][0] = ro;
      E->me[clusternum][1] = -1;
      group->ive[(*groupsize)++] = p->snum;
      node_state->ive[node_heap->ive[idx]] = -1;
    }
    p->snum++;
    idx++;
    clusternum++;
  }// end of while
  //deal with the last node
  if(node_state->ive[node_heap->ive[idx]] != -1){
    node_state->ive[node_heap->ive[idx]] = -1;
    ro = node_heap->ive[idx];
    p->sons->ive[p->snum*2] = ro;
    p->sons->ive[p->snum*2+1] = -1;
    E->me[clusternum][0] = ro;
    E->me[clusternum][1] = -1;
    node_state->ive[node_heap->ive[idx]] = -1;
    group->ive[(*groupsize)++] = p->snum;
    p->snum++;
    clusternum++;
  }
  ......
  return matched;
}
```

The process to build a coarser graph is based matrix operations. Let S be a graph and E be a matrix containing the information of clusters, then a coarser graph SC can be defined as $SC = E \times S \times E^T$. Each row of E represents a cluster built over S by the modified HEM algorithm. The column size of E is 2 since each cluster contains at most 2 vertices. The coarser graph SC is constructed by merging two rows of S whose indices are in the same cluster into one row and two columns of S whose indices are the same cluster into one column. To speed up the column merging process, we call the function *sp_col_access* of Meschach to build extra column access paths for the sparse matrix S. The following function *spmtrmm_mlt* implements the process to build the coarser graph SC:

```
void  spmtrmm_mlt(MAT *E, SPMAT *S, SPMAT *SC)
{
  ......
  SPMAT *temp = SMNULL;
  //allocate memory for temp
  ......
  //temp = E*S by row merging
  for(i=0; i<n; i++){
    i1 = E->me[i][0];
    i2 = E->me[i][1];
    if(i1!=-1 && i2!=-1){
      //merge row i1 and row i2 together
      sprow_add(&S->row[i1],&S->row[i2],
          0 ,&temp->row[i],TYPE_SPMAT);
    }else{
      len = S->row[i1].len;
      for(j=0; j<len; j++)
          sp_set_val(temp, i, S->row[i1].elt[j].col,
             S->row[i1].elt[j].val);
    }
  }
  //Construct temp*E' = (E*temp')' by column merging
  sp_col_access(temp);
  for(i=0; i<n; i++){
```

```
j1 = E->me[i][0];
j2 = E->me[i][1];
//merge column together
if(j2!=-1){
  si1 = temp->start_row[j1];
  j_idx1 = temp->start_idx[j1];
  si2 = temp->start_row[j2];
  j_idx2 = temp->start_idx[j2];
  while(si1>=0 && si2>=0){
    if(si1<si2){
      sp_set_val(SC, si1 ,i,
            temp->row[si1].elt[j_idx1].val);
      elt = &(temp->row[si1].elt[j_idx1]);
      i_tmp = elt->nxt_row;
      j_idx1 = elt->nxt_idx;
      si1 = i_tmp;
    }else if(si1>si2){
      sp_set_val(SC, si2 ,i,
            temp->row[si2].elt[j_idx2].val);
      elt = &(temp->row[si2].elt[j_idx2]);
      i_tmp = elt->nxt_row;
      j_idx2 = elt->nxt_idx;
      si2 = i_tmp;
    }else{
      sp_set_val(SC, si2 ,i,
            temp->row[si2].elt[j_idx1].val
            +temp->row[si2].elt[j_idx2].val);
      elt = &(temp->row[si1].elt[j_idx1]);
      i_tmp = elt->nxt_row;
      j_idx1 = elt->nxt_idx;
      si1 = i_tmp;
      elt = &(temp->row[si2].elt[j_idx2]);
      i_tmp = elt->nxt_row;
      j_idx2 = elt->nxt_idx;
      si2 = i_tmp;
    }
  }
  if(si1>=0){
    while(si1>=0){
      sp_set_val(SC, si1 ,i,
          temp->row[si1].elt[j_idx1].val);
      elt = &(temp->row[si1].elt[j_idx1]);
      i_tmp = elt->nxt_row;
      j_idx1 = elt->nxt_idx;
```

```
            si1 = i_tmp;
          }
        }else if(si2>=0){
          while(si2>=0){
            sp_set_val(SC, si2 ,i,
              temp->row[si2].elt[j_idx2].val);
            elt = &(temp->row[si2].elt[j_idx2]);
            i_tmp = elt->nxt_row;
            j_idx2 = elt->nxt_idx;
            si2 = i_tmp;
          }
        }else{
          si1 = temp->start_row[j1];
          j_idx1 = temp->start_idx[j1];
          while(si1>=0){
            sp_set_val(SC, si1 ,i,
                temp->row[si1].elt[j_idx1].val);
            elt = &(temp->row[si1].elt[j_idx1]);
              i_tmp = elt->nxt_row;
            j_idx1 = elt->nxt_idx;
            si1 = i_tmp;
          }
        }
      }
    }
    ......
}
```

To build an index cluster tree T_I using the results obtained by the above multilevel coarsening process, a group of pointer arrays are created: *ctree* is a group of pointers pointing to the integer vector *sons* of *spnode* on each level; *ncluster* is built by mapping each cluster in *ctree* back to the cluster consisting of vertices of the original graph G_0. Figure 6.2 shows an example of an index cluster tree described by *ctree* and *ncluster*. The $\mathcal{H}$-matrix construction function *build_hmatrix* builds the root of the block cluster tree. It recursively calls *build_hsons* to build the lower levels of the tree using the sequence of graphs and clusters obtained by the multilevel clustering

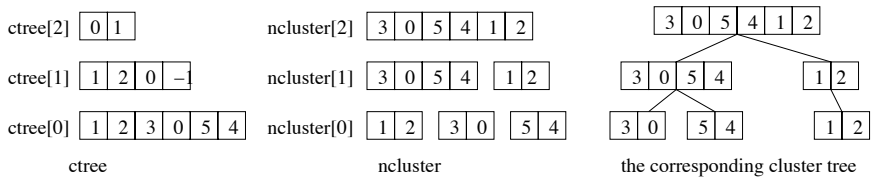


Figure 6.2: An example of the index cluster tree described by the structure *ctree* and *ncluster*.

process. The code of *build_hmatrix* is given as follows:

```
struct supermatrix* build_hmatrix(
    struct supermatrix *root,IVEC **ctree,
    IVEC ***ncluster, int *cnum,
    SPMAT **G, SPMAT *K, int level)
{
  ......
  root->type = HMAT;
  root->rows = K->m;
  root->cols = K->n;
  root->block_rows = cnum[level-1];
  root->block_cols = cnum[level-1];
  root->s = (struct supermatrix**)
      malloc(root->block_rows*root->block_cols
      *sizeof(struct supermatrix*));
  ......
  k = root->block_rows;
  for(j=0; j<root->block_cols; j++) {
    for(i=0; i<root->block_rows; i++) {
      root->s[i+k*j] = (struct supermatrix*)
          malloc(sizeof(struct supermatrix));
      ......
      root->s[i+k*j] = build_hsons(root->s[i+k*j],
          ctree, ncluster, cnum,
          G, K, i, j, level-1);
    }
  }
  return root;
}
```

Here G points to the sequence of coarser graphs, K is the original graph G_0, and

cnum is the number of clusters on each level. The main body of the recursive function *build_hsons* is given as follows:

```
.....
if(sp_get_val(G[level], rnd, cnd) == 0){
    sm->type = RKMAT;
    sm->rows = ncluster[level][rnd]->dim;
    sm->cols = ncluster[level][cnd]->dim;
    sm->block_rows = 1;
    sm->block_cols = 1;
    sm->r = new_rkmatrix(0, sm->rows, sm->cols);
    sm->f = MNULL;
    sm->s = NULL;

}else if(ncluster[level][rnd]->dim <= SBLOCK
    || ncluster[level][cnd]->dim <= SBLOCK){
  sm->type = FMAT;
  sm->rows = ncluster[level][rnd]->dim;
  sm->cols = ncluster[level][cnd]->dim;
  sm->block_rows = 1;
  sm->block_cols = 1;
  sm->r = NULL;
  sm->s = NULL;
  sm->f = mysp_bget(K, sm->f,
      ncluster[level][rnd]->ive,
      ncluster[level][rnd]->dim,
      ncluster[level][cnd]->ive,
      ncluster[level][cnd]->dim);
}else{
  sm->type = HMAT;
  sm->rows = ncluster[level][rnd]->dim;
  sm->cols = ncluster[level][cnd]->dim;
  if(ctree[level]->ive[rnd*2+1]== -1)
    sm->block_rows = 1;
  else
    sm->block_rows = 2;
  if(ctree[level]->ive[cnd*2+1]== -1)
    sm->block_cols = 1;
  else
    sm->block_cols = 2;
  sm->s = (struct supermatrix**)
      malloc(sm->block_rows*sm->block_cols
      *sizeof(struct supermatrix*));
```

```
    k = sm->block_rows;
    for(j=0; j<sm->block_cols; j++){
      for(i=0; i<sm->block_rows; i++){
        rnd1 = ctree[level]->ive[rnd*2+i];
        cnd1 = ctree[level]->ive[cnd*2+j];
        sm->s[i+k*j] = (struct supermatrix*)
            malloc(sizeof(struct supermatrix));
        ......
        sm->s[i+k*j] = build_hsons(sm->s[i+k*j],ctree,
            ncluster, cnum, G, K, rnd1, cnd1, level-1);
      }
    }
  }
  ......
```

The function *build_hsons* checks the edge between two vertices decide the type of an $\mathcal{H}$-matrix block. If a block needs to be partitioned further on the next level, then *build_hsons* recursively calls itself to build the child $\mathcal{H}$-matrix blocks.

6.4 $\mathcal{H}$-LU Factorization

A is an input $\mathcal{H}$-matrix, L and U store the $\mathcal{H}$-LU factors, and W is a workspace used to store the intermediate results. The memory and tree structures have been created for all the parameters before they are passed to the function. The following is the implementation of $\mathcal{H}$-LU factorization:

```
void LU_hmat(struct supermatrix *A, struct supermatrix *L,
    struct supermatrix *U, struct supermatrix *W)
{
    ......
    if(A->type == FMAT){
      //LU decomposition on a full matrix
      L->f = m_get(A->f->m, A->f->n);
      U->f = m_get(A->f->m, A->f->n);
      myLUfactor(A->f, L->f, U->f);
  }else{
      //block LU factorization
```

```
      bm = A->block_rows;
      bn = A->block_cols;
      ......
      for(k=0; k<bm; k++){
        LU_hmat(A->s[k+bm*k], L->s[k+bm*k],
             U->s[k+bm*k], W->s[k+bm*k]);
        for(i=k+1; i<bm; i++){
          LU_usv(A->s[i+bm*k], L->s[i+bm*k],
               U->s[k+bm*k], W->s[i+bm*k]);
          LU_lsv(A->s[k+bm*i], L->s[k+bm*k],
               U->s[k+bm*i], W->s[k+bm*i]);
        }
        for(j=k+1; j<bm; j++){
          for(i=k+1; i<bm; i++){
               hmat_submlt(A->s[i+bm*j], L->s[i+bm*k],
                    U->s[k+bm*j], W->s[i+bm*j]);
          }
        }
      }
    }
    ......
}
```

The function *LU_hmat* is a recursive function. If A is a full matrix block, then LU factorization for full matrices is called. The function *LU_usv* is an $\mathcal{H}$-matrix upper triangular solve and the function *LU_lsv* is an $\mathcal{H}$-matrix lower triangular solve.

CHAPTER 7
SUMMARY

In this thesis, we take a deep look at construction of effective preconditioners for iterative methods to solve large systems of linear equations. There has been a lot of research work on building preconditioners to speed up the convergence of iterative methods: the preconditioners that target matrices with nonzero diagonal entries, like JOR and SOR; the preconditioners that target systems arising from partial differential equations, like multigrid methods [23, 58]. Depending on the information used for building preconditioners, preconditioners can also be divided into geometric preconditioners and algebraic preconditioners. The construction of geometric preconditioners needs the domain information underlying problems. The construction of algebraic preconditioners uses only the information contained in matrices.

$\mathcal{H}$-matrix techniques were introduced in 1998, which use a data sparse format to represent a dense or a sparse matrix and $\mathcal{H}$-matrix arithmetic to perform matrix operations. $\mathcal{H}$-matrix techniques reduce storage as well as computational complexity of matrices. $\mathcal{H}$-matrix techniques provide an alternative way to build effective preconditioners for iterative methods to solving large systems of equations. One of the key steps of the $\mathcal{H}$-matrix preconditioner technique is to construct $\mathcal{H}$-matrices. $\mathcal{H}$-matrices can be constructed in geometric way and algebraic way. The geometric construction approaches approximate matrix blocks by low rank matrices. The algebraic construction methods work on matrix graphs to build block cluster trees. The algebraic construction methods based on nested dissection (domain decomposi-

tion) have been successfully applied to solve problems such as convection-dominated problems [19].

In my thesis, I have presented a new algebraic $\mathcal{H}$-matrix construction approach. Our approach is based on multilevel clustering: it uses the modified HEM algorithm to build a sequence of coarser graphs and clusters over the original graph; a balanced index tree is constructed using the information obtained during the clustering process. The $\mathcal{H}$-matrices constructed by our approach represent the original matrices exactly, while the dense blocks are clustered around the diagonal and some of the off-diagonal blocks are in Rk-matrix format. Our approach can be used to construct effective preconditioners for iterative methods to solve partial differential equations, which speed up the convergence of iterative methods greatly [55]. We have also expanded our approach and built a scheme to construct $\mathcal{H}$-matrix preconditioners to solve saddle point problems [20, 56]. The experimental results show that our preconditioners are competitive to other H-matrix preconditioners and existing preconditioners such as JOR and AMG preconditioners. Our H-matrix construction approach and preconditioner technique provide an alternative effective way to solve large systems of linear equations.

REFERENCES

[1] M. Anitescu and F. Potra. Formulating dynamic multi-rigid-body contact problems with friction as solvable linear complementarity problems. *Nonlinear Dynamics*, 14:231–247, 1997.

[2] M. Anitescu, F. Potra, and D. Stewart. Time-stepping for threedimensional rigid body dynamics. *Comp. Methods Appl. Mech. Engineering*, 177, 1998.

[3] Stephen T. Barnard and Horst D. Simon. A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. *Concurrency: Practice and Experience*, 6:101–107, 1994.

[4] M. Bebendorf and W. Hackbusch. Existence of $\mathcal{H}$-matrix approximants to the inverse FE-matrix of elliptic operators with l^∞-coefficients. *Numerische Mathematik*, 95, 2003.

[5] M. Bebendorf and S. Rjasanow. Adaptive low-rank approximation of collocation matrices. *Computing*, 70(1):1–24, 2004.

[6] M. Bebendorf and S. Rjasanow. Approximation of solution operators of elliptic partial differential equations by $\mathcal{H}$- and $\mathcal{H}^2$-matrices. 2007. submitted.

[7] T. Belytschko, Y. Krongauz, J. Dolbow, and C. Gerlach. On the completeness of meshfree particle methods. *International Journal for Numerical Methods in Engineering*, 43(5):785 – 819, 1998.

[8] M. Benzi and G. H. Golub. A preconditioner for generalized saddle point problems. *SIAM J. Matrix Anal. Appl.*, 26(1):20–41, 2004.

[9] M. Benzi, G. H. Golub, and J. Liesen. Numerical solution of saddle point problems. *Acta Numer.*, 14:1–137, 2005.

[10] S. Borm, L. Grasedyck, and W. Hackbusch. Introduction to hierarchical matrices with applications. *Engineering Analysis with Boundary Elements*, 27(5):405–422, 2003.

[11] S. Borm, L. Grasedyck, and W. Hackbush. Hierarchical matrices. Technical Report 24, Max-Planck-Institute for Mathematics in the Sciences, Leipzig, Germany, 2005.

[12] S. Le Borme. $\mathcal{H}^2$-matrix arithmetics in linear complexity. *Computing*, 77(1):1–28, 2006.

[13] S. L. Borne and S. Oliveira. Joint domain-decomposition h-lu preconditioners for saddle point problems. *Electronic Transactions on Numerical Analysis*, 26:285–298, 2007.

[14] S. Le Borne. H-matrices for convection-diffusion problems with constant convection. *Computing*, pages 261–274, 2003.

[15] S. Le Borne. Hierarchical matrices for convection-dominated problems. In *Domain Decomposition Methods in Science and Engineering*, volume 40 of *LNCSE*, pages 631–638, 2005.

[16] S. Le Borne. Hierarchical matrix preconditioners for the stokes equations. In *Proceedings of the 8th European Multigrid Conference on Multigrid, Multilevel and Multiscale Methods*, pages 27–30, 2005.

[17] S. Le Borne. Hierarchical matrix preconditioners for the Oseen equations. *Comput. Vis. Sci.*, 2006.

[18] S. Le Borne. Multilevel hierarchical matrices. *SIAM J. Matrix Anal. Appl*, 28(3):871–889, 2006.

[19] S. Le Borne and L. Grasedyck. $\mathcal{H}$ preconditioners in convection-dominated problems. *SIAM J. Matrix Anal. Appl.*, 27(4):1172–1183, 2006.

[20] S. Le Borne, S. Oliveira, and F. Yang. H-matrices preconditioners for symmetric saddle-point systems from meshfree discretizations. *to appear in Numerical Linear Algebra and its Applications.*

[21] Sabine Le Borne, Lars Grasedyck, and Ronald Kriemann. Domain-decomposition based $\mathcal{H}$-LU preconditioners. In *Proceedings of the 16th International Conference on Domain Decomposition Methods*, volume 55 of *LNCSE*, pages 661–668, 2006.

[22] F. Brezzi and M. Fortin. *Mixed and hybrid finite element methods.* Springer-Verlag New York, Inc., New York, NY, USA, 1991.

[23] W. L. Briggs. *A Multigrid tutorial.* SIAM, Philadelphia, 1987.

[24] T. Bui and C. Jones. A heuristic for reducing fill in sparse matrix factorization. *In 6th SIAM Conf. Parallel Processing for Scientific Computing*, pages 445–452, 1993.

[25] Zhiji Cai. Convergence and error estimates for meshless Galerkin methods. *Appl. math. comput.*, 184(2):908–916, 2007.

[26] J.S. Chen, C. Pan, C. T. Wu, and W. K. Liu. Reproducing kernel particle methods for large deformation analysis of non-linear structures. *Compt. Methods Appl. Engrg.*, 139:195–227, 1996.

[27] P. Concus, G. H. Golub, and D. P. O'Leary. *A generalized conjugate gradient method for the numerical solution of elliptic partial differential equations.* in sparse matrix computations, J. R. Bunch and D. J. Rose, eds., Academic Press, New York, 1976.

[28] Wolfgang Dahmen and Reinhold Schneider. Wavelets on manifolds I: Construction and domain decomposition. *SIAM Journal on Mathematical Analysis*, 31(1):184–230, 2000.

[29] N. Dyn and Jr. W. E. Ferguson. The numerical solution of equality-constrained quadratic programming problems. *Mathematics of Computation*, 41(163):165–170, 1983.

[30] J.A. George. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10:345–367, 1973.

[31] J.A. George and J.W. Liu. An automatic nested dissetion algorithm for irregular finite elment problems. *SIAM Journal on Numerical Analysis*, 15:1053–1069, 1978.

[32] J.R. Gilbert and R. E. Tarjan. The analysis of a nested disection algorithm. *Numerische mathematik*, 50:377–404, 1987.

[33] L. Grasedyck, S. Le Borne, and R. Kriemann. Parallel blackbox h-lu preconditioning for elliptic boundary value problems. *Comput. Vis. Sci., to appear*, 2007.

[34] L. Grasedyck, R. Kriemann, and S. Le Borne. Domain decomposition based H-LU preconditioning. Technical Report 115, Max-Planck-Institute for Mathematics in the Sciences, Leipzig, Germany, 2007.

[35] Anne Greenbaum. *Iterative Methods for Solving Linear Systems.* SIAM Publ., Philadelphia, PA, 1997.

[36] L. Greengard and V. Rokhlin. A new version of the fast multipole method for the laplace equation in three dimensions. *Acta Numerica*, 6, 1997.

[37] W. Hachbusch. the panel clusering algorithm. In *MAFELAP*, pages 339–348, 1990.

[38] W. Hackbusch. A sparse matrix arithmetic based on h-matrices. part i: introduction to h-matrices. *Computing*, 62(2):89–108, 1999.

[39] W. Hackbusch. Direct domain decomposition using the hierarchical matrix technique. In *Domain Decomposition methods in science and engineering*, pages 39–50, 2003.

[40] W. Hackbusch and B. Khoromskij. A sparse h-matrix arithmetic. part 2: Application to multi-dimensional problems. *Computing*, 64(1):21–47, 2000.

[41] W. Hackbusch and B. N. Khoromskij. A sparse h -matrix arithmetic: general complexity estimates. *J. Comput. Appl. Math.*, 125(1-2):479–501, 2000.

[42] W. Hackbusch, B. N. Khoromskij, and R. Kriemann1. Hierarchical matrices based on a weak admissibility criterion. *Computing*, 73(3):207–243, 2004.

[43] W. Hackbusch, B. N. Khoromskij, and S. A. Sauter. On h^2-matrices. pages 9–29, 2002.

[44] W. Hackbusch and Z. P. Nowak. On the fast matrix multiplication in the boundary element method by panel clustering. *Numerische Mathematik*, 54(4), 1989.

[45] B. Hendrickson and E. Rothberg. Improving the run time and quality of nested dissection ordering. *SIAM J. Sci. Comput.*, 20(2):468–489, 1998.

[46] Bruce Hendrickson and Robert W. Leland. A multi-level algorithm for partitioning graphs. In *Supercomputing*, 1995.

[47] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[48] R. Kriemann. Parallel $\mathcal{H}$-matrix arithmetics on shared memory systems. *Computing*, 74:273–297, 2004.

[49] W. Hackbusch L. Grasedyc and. Construction and arithmetics of h-matrices. *Computing*, 70, 2003.

[50] K. H. Leem, S. Oliveira, and D. Stewart. Algebraic multigrid (AMG) for saddle point systems from meshfree discretizations. *Numerical Linear Algebra and Applications*, 11(3):293–308, 2004.

[51] M. Lintner. The eigenvalue problem for the 2d laplacian in $\mathcal{H}$-matrix arithmetic and application to the heat and wave equation. *Computing*, 72(3-4):293–323, 2004.

[52] J. A. Meijerink and H. A. van der Vorst. An iterative solution method for linear systems of which the coefficient matrix is a symmetric m-matrix. *Math. Comp.*, 31:148–162, 1977.

[53] Gary L. Miller, Shang-Hua Teng, and Stephen A. Vavasis. A unified geometric approach to graph separators. In *Proceedings of the 32nd annual symposium on Foundations of computer science*, pages 538–547, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[54] S. Oliveira, R. Ortiz, and D. Stewart. Newton/amg iterative algorithm for solving rigid body dynamics problems with frictional impacts. *submitted*, 2008.

[55] S. Oliveira and F. Yang. An algebraic approach for h-matrices preconditioners. *Computing*, 80:169–188, 2007.

[56] S. Oliveira and F. Yang. H-matrix preconditioners for saddle-point systems from meshfree discretization. In *Proceedings of the 14th International Conference on Computational Experimental Engineering and Sciences*, pages 567–574, 2007.

[57] S. Oliveira and F. Yang. Hierarchical preconditioners for parabolic optimal control problems. *Lectures Notes in Computer Science 4487*, pages 221–228, 2007.

[58] Vaněk P, Mandel J, and Brezina M. Algebraic multigrid by smoothed aggregation for second and fourth order elliptic problems. *Computing*, pages 179–196, 1996.

[59] P. Persson and G. Strang. A simple mesh generator in Matlab. *SIAM Review*, 46(2):329–345, 2004.

[60] Y. Saad. *Iterative Methods for Sparse Linear Systems*. PWS Pub. Co., Boston, MA, 1996.

[61] Y. Saad and M.H. Schultz. Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 7:856–869, 1986.

[62] C. Schaerer, T. Mathew, and M. Sarkis. Block iterative algorithms for the solution of parabolic optimal control problems. *VECPAR*, 2006.

[63] D. E. Stewart and Z. Leyk. *Meschach: Matrix Computations in C*, volume 32 of *Proceedings of the CMA*. The Australian National University, 1994.

[64] D. M. Young. *Iterative solution of Large Linear Systems*. Academic Press, New York, 1971.