A Parallel Hierarchical Solver for the Poisson Equation

Raghunathan Sudarshan
darshan@mit.edu
Department of Civil and Environmental Engineering
Massachusetts Institute of Technology

Seung E. Lee
selee@mit.edu
Department of Mechanical Engineering
Massachusetts Institute of Technology

May 14, 2003
Abstract

In this term project report we describe the implementation of a hierarchical basis solver for the Poisson equation over a two-dimensional manifold on a parallel architecture. Finite difference and finite element approximations to the Poisson operator lead to condition number which grow as $O(h^{-2})$ as the grid is refined. However, even the simplest, unstabilized hierarchical basis discretizations produce stiffness matrices whose condition numbers grow as $O(|\log h|^2)$ while preserving the order of convergence of the finite element approximations. However, this improvement in condition number comes at the cost of sparsity of the matrix which leads to increased communication between processors in a parallel implementation. In this report we discuss the implementation of a relatively simple hierarchical basis solver and present some representative numerical results which examine some of the tradeoffs involved in using a hierarchical rather than a single-level discretization.

Keywords: Hierarchical Bases, Multilevel Solvers, Parallel Computation, Preconditioning
# Contents

1 **Introduction** .................................................. 2
   1.1 Key challenges and outline .................................. 2

2 **Theory** .......................................................... 2
   2.1 Notation ......................................................... 2
   2.2 Interpolating Subdivision ...................................... 3
   2.3 Multiresolution analysis from subdivision .................... 4
   2.4 Multilevel formulation of the Poisson equation ............... 5
   2.5 Element Formulation ........................................... 5

3 **Implementation** ................................................ 7
   3.1 Data structures ................................................ 7
      3.1.1 Data structures for storing the mesh ....................... 7
      3.1.2 Data structure for sparse matrices ......................... 8
      3.1.3 Data input and output formats ........................... 9
   3.2 Data organization on a parallel architecture ................. 9
      3.2.1 Mesh partitioning algorithm ................................ 9
      3.2.2 Distribution of $\hat{K}$ and $W^{-1}$ ....................... 10
   3.3 Algorithms ..................................................... 10
      3.3.1 The Oracle .................................................. 11
      3.3.2 Assembly .................................................... 12
      3.3.3 Solution .................................................... 12
      3.3.4 Reconstruction ............................................. 12
   3.4 Running the program .......................................... 12

4 **Numerical Results** ............................................. 13
   4.1 Square Domain ................................................ 13
   4.2 L Domain ....................................................... 16
   4.3 MIT domain .................................................... 18

5 **Conclusions and Further work** .................................. 20
   5.1 Further Work .................................................. 20

A **Source code highlights** ........................................ 22
   A.1 The driver program ............................................ 22
   A.2 The oracle ..................................................... 23
      A.2.1 Mesh subdivision .......................................... 23
      A.2.2 Mesh partitioning ......................................... 24
   A.3 The multilevel solver ......................................... 26
   A.4 Sparse matrix allocation .................................... 26
      A.4.1 Filling an entry in a sparse matrix ....................... 26
   A.5 Stiffness matrix assembly ................................... 27
1 Introduction

For solving large finite element problems, direct solution methods such as Gaussian elimination tend to be prohibitively expensive especially in higher dimensions. Therefore, there has been a lot of impetus in recent years towards the development of iterative Krylov subspace solvers such as PCG (Preconditioned Conjugate gradients) and GMRES (Generalized Minimum RESidual) [1, 2]. However, iterative solvers offer computational advantages only if a suitable preconditioner is available for a given problem. This is because discretization methods such as finite differences and finite elements lead to stiffness matrices whose condition number grows rapidly with the number of degrees of freedom. This is one of the main reasons for the growing popularity of multilevel methods such as multigrid and hierarchical bases since under certain conditions the number of iterations for convergence remains independent of the number of degrees of freedom in the mesh. An additional advantage of hierarchical methods is that mesh refinement and adaptivity are “hard-wired” rather than “strapped-on” to the basis functions which lead to significant ease of implementation.

In this project, we hoped to explore some of the main challenges involved in implementing a hierarchical basis solver on a parallel architecture and to study its effectiveness in terms of the number of iterations at the finest level of discretization and the speedup obtained on running the problem on more and more processors.

1.1 Key challenges and outline

The key challenges in our project were to design efficient data structures for storing the multilevel mesh, implement mesh subdivision and partitioning and to assemble, solve and reconstruct the solution at different levels of resolution.

We first present the main theory of hierarchical basis and wavelet methods in Section 2 and derive the element formulation for a four-node multilevel element. In Sections 3 and 4 we describe in detail our datastructures, algorithms and numerical results. In the appendix we list and discuss a few essential code segments from our implementation.

2 Theory

2.1 Notation

Consider a domain $\Omega \subset \mathbb{R}^2$ and let $\Omega_j$ be its discretization at resolution $j$, where $0 \leq j < J$. Denote the non-empty set of vertices in the domain $\Omega_j$ by $\mathcal{K}(j)$ and let $\mathcal{M}(j) = \mathcal{K}(j+1) \setminus \mathcal{K}(j)$ be the set of “child” vertices of $\mathcal{K}(j)$. To keep notation concise, the indices $k, m$ and $l$ refer to vertices in the sets $\mathcal{K}(j), \mathcal{M}(j)$ and $\mathcal{K}(j+1)$ respectively.

For a vertex $m \in \mathcal{M}(j)$, let $N(j, m)$ be its set of parent vertices in $\mathcal{K}(j)$ and let $n(j, k) = \{m \in \mathcal{M}(j) | k \in N(j, m)\}$. Finally, let $u_{j,k}$ (resp.$u_{j,m}$) denote the samples of a function at vertices in $\mathcal{K}(j)$ (resp. $\mathcal{M}(j)$).

2
Figure 1 shows a quadrilateral domain at two levels of discretization and illustrates the notation used in this section. The child vertices located at the centre of an edge are denoted as *edge vertices* and those located at the centroid of the face are denoted as *face vertices*.

![Figure 1: Illustration of a multilevel mesh. Observe that \( N(j, m_0) = \{k_0, k_3\} \) while \( N(j, m_4) = \{k_0, k_1, k_2, k_3\} \)](image)

2.2 Interpolating Subdivision

Given the samples of a function at a coarser resolution \( j \), subdivision, which in essence is a local averaging procedure, can be used to determine the samples of the function at level \( j + 1 \). An interpolating subdivision scheme preserves the samples of the function at the parent nodes while interpolating the values at the child nodes [4].

This can be represented as:

\[
\begin{align*}
  u_{j+1,k} &= u_{j,k} \\
  u_{j+1,m} &= \sum_{k \in N(j,m)} h_j[k,m] u_{j,k},
\end{align*}
\]

or in matrix notation as

\[
u_{j+1} = \begin{bmatrix} I & S \end{bmatrix} u_j \]

The simplest form of interpolating subdivision is *linear interpolating* subdivision where the value of the function at a child vertex is simply taken as the average of the values at its parent vertices.

The *scaling function*, \( \varphi_{j,k}(x) \) is defined as the limit of the subdivision process (assuming such a limit exists), starting with \( u_{j,k} = \delta_{k,k'} \) where \( k, k' \in \mathcal{K}(j) \). For linear interpolating subdivision, the scaling functions correspond to hat functions. It can be shown that the scaling functions from interpolating subdivision satisfy a refinement relation of the form

\[
\varphi_{j,k}(x) = \varphi_{j+1,k}(x) + \sum_{m \in n(j,k)} h_j[k,m] \varphi_{j+1,m}(x)
\]
2.3 Multiresolution analysis from subdivision

The spaces \( V_j = \text{clos span}\{\varphi_{j,k}(x) \forall k \in \mathcal{K}(j)\} \) arising from interpolating subdivision schemes can be shown to satisfy the following properties:

1. **Nestedness:** \( V_j \subset V_{j+1} \).

2. **Completeness:** \( \bigcup_{j=0}^{\infty} V_j = L^2(\Omega) \).

3. **Existence of Wavelet Spaces:** \( V_j + W_j = V_{j+1} \), where \( W_j = \text{clos span}\{w_{j,m}(x) \forall m \in \mathcal{M}(j)\} \).

The functions \( w_{j,m}(x) \) are known as wavelets. In the rest of this report, we assume that these functions are the classical unstabilized hierarchical bases \([6, 7]\), i.e.,

\[
w_{j,m}(x) = \varphi_{j+1,m}(x)
\]

Multiresolution analysis refers to the representation of a function \( f_{j+1}(x) \in L^2(\Omega_{j+1}) \) in terms of its projection on a coarse approximation space \( V_0 \) along with multiple levels of details, which are the projections of the functions on the wavelet spaces, \( W_i, i = 0, 1, \ldots, j \). This can be represented as:

\[
f_{j+1}(x) = \sum_{l \in \mathcal{K}(j+1)} u_{j+1,l} \varphi_{j+1,l}(x) = \sum_{k \in \mathcal{K}(j)} u_{j,k} \varphi_{j,k}(x) + \sum_{m \in \mathcal{M}(j)} u_{j,m} \varphi_{j+1,m}(x) = \sum_{k \in \mathcal{K}(0)} u_{0,k} \varphi_{0,k}(x) + \sum_{i=0}^{j} \sum_{m \in \mathcal{M}(i)} u_{i,m} \varphi_{i+1,m}(x),
\]

where the coefficients \( u_{j,k} \) and \( u_{j,m} \) are obtained from the coefficients \( u_{j+1,l} \) as:

\[
\begin{align*}
    u_{j,k} &= u_{j+1,k} \\
    u_{j,m} &= u_{j+1,m} - \sum_{k \in \mathcal{N}(j,m)} h_{j,k} u_{j,k}
\end{align*}
\]

The above equation represents the one-step fast interpolating wavelet transform, and can be trivially inverted as:

\[
\begin{align*}
    u_{j+1,m} &= u_{j,m} + \sum_{k \in \mathcal{N}(j,m)} h_{j,k} u_{j,k} \\
    u_{j+1,k} &= u_{j,k}
\end{align*}
\]

The primary advantage of multilevel representation is that for smooth function, the wavelet coefficients \( u_{i,m} \) decay rapidly with the resolution \( i \) and hence the multiscale approach facilitates the compact representation of functions. This is particularly appealing for functions which are smooth almost everywhere (for instance solutions to the non-linear Burger’s equation), since only those wavelet coefficients near the features of interest need to be computed or stored.
Equations 1 and 2 can be written in matrix notation as:

\[
\begin{pmatrix}
u_{j,k} \\
u_{j,m}
\end{pmatrix}
=\begin{pmatrix}
I & 0 \\
-S & I
\end{pmatrix}_W
\begin{pmatrix}
u_{j+1,k} \\
u_{j+1,m}
\end{pmatrix}
\quad \text{and} \quad
\begin{pmatrix}
u_{j+1,k} \\
u_{j+1,m}
\end{pmatrix}
=\begin{pmatrix}
I & 0 \\
S & I
\end{pmatrix}_{W^{-1}}
\begin{pmatrix}
u_{j,k} \\
u_{j,m}
\end{pmatrix}
\]

Note that both the wavelet transform matrix and its inverse are sparse and can be applied to a vector in linear time.

### 2.4 Multilevel formulation of the Poisson equation

Consider the Poisson equation,

\[
\nabla^2 u + 1 = 0 \quad \text{on} \quad \Omega
\]

(3)

with homogeneous Dirichlet boundary conditions,

\[u|_{\partial\Omega} = 0.\]

The weak form of Equation 3 can then be written as

\[a(u, v) = l(v),\]

where the \( u, v \in H^1_0(\Omega), a(u, v) = \int_{\Omega} \nabla u \cdot \nabla v \, d\Omega \) is a SPD bilinear form and \( l(v) = \int_{\partial\Omega} v \, d\partial\Omega \) is a linear form.

Discretizing Equation 4 using the Galerkin procedure using scaling functions at resolution \( j+1 \) (which correspond to finite element interpolation functions) leads to the following system of equations

\[K_{j+1} u_{j+1} = f_{j+1}\]

(5)

where \( K_{j+1}[r, s] = a(\varphi_{j+1,r}(x), \varphi_{j+1,s}(x)) \) and \( f_{j+1}[r] = l(\varphi_{j+1,r}(x)) \).

We can now derive the multiscale form of Equation 5 as follows:

\[
(K^{-1} \tilde{K} \tilde{u}_{j+1} = (K^{-1} \tilde{f}_{j+1})
\]

(6)

### 2.5 Element Formulation

We can now derive the element formulation for the multiscale stiffness matrix obtained using piecewise linear hierarchical bases. For ease of derivation and implementation, we restrict ourselves to tensor product bases (i.e., rectangular finite elements). A similar procedure can be derived for non-separable functions, though it is no longer possible to compute the entries in the stiffness matrix in closed-form.

Consider the mesh of rectangular elements in Figure 2. Nodes 0 through 3 are the parent nodes and nodes 4 through 8 are the child nodes.
Figure 2: A two-level finite element mesh

The single level stiffness matrix, $\mathbf{K}$ corresponding to the assemblage can be written as:

$$
\begin{bmatrix}
\frac{b}{3a} + \frac{a}{3b} & 0 & 0 & 0 & \frac{b}{6a} - \frac{a}{3b} & -\frac{b}{3a} + \frac{a}{6b} & 0 & 0 \\
0 & \frac{b}{3a} + \frac{a}{3b} & 0 & 0 & 0 & \frac{b}{6a} - \frac{a}{3b} & \frac{b}{6a} - \frac{a}{3b} & 0 \\
0 & 0 & \frac{b}{3a} + \frac{a}{3b} & 0 & 0 & 0 & -\frac{b}{3a} + \frac{a}{6b} & -\frac{b}{3a} + \frac{a}{6b} \\
\frac{b}{6a} - \frac{a}{3b} & 0 & 0 & \frac{b}{6a} - \frac{a}{3b} & \frac{b}{3a} + \frac{2a}{3b} & -\frac{b}{6a} - \frac{a}{6b} & 0 & -\frac{b}{3a} + \frac{a}{6b} \\
0 & \frac{b}{6a} - \frac{a}{6b} & -\frac{b}{6a} - \frac{a}{6b} & 0 & 0 & \frac{2b}{3a} + \frac{2a}{3b} & \frac{2b}{3a} + \frac{2a}{3b} & 0 \\
\frac{b}{6a} - \frac{a}{6b} & 0 & -\frac{b}{3a} + \frac{a}{6b} & -\frac{b}{3a} + \frac{a}{6b} & -\frac{b}{3a} + \frac{a}{6b} & \frac{b}{3a} - \frac{2a}{3b} & \frac{b}{3a} - \frac{2a}{3b} \\
\frac{b}{6a} - \frac{a}{6b} & \frac{b}{6a} - \frac{a}{6b} & \frac{b}{6a} - \frac{a}{6b} & \frac{b}{6a} - \frac{a}{6b} & \frac{b}{6a} - \frac{a}{6b} & \frac{b}{6a} - \frac{a}{6b} & \frac{b}{6a} - \frac{a}{6b} \\
\frac{b}{6a} - \frac{a}{6b} & \frac{b}{6a} - \frac{a}{6b} & \frac{b}{6a} - \frac{a}{6b} & \frac{b}{6a} - \frac{a}{6b} & \frac{b}{6a} - \frac{a}{6b} & \frac{b}{6a} - \frac{a}{6b} & \frac{b}{6a} - \frac{a}{6b}
\end{bmatrix}
$$

and the inverse wavelet transform matrix can be written as:

$$
\mathbf{W}^{-1} =
\begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
1/2 & 0 & 0 & 1/2 & 1 & 0 & 0 & 0 \\
1/2 & 1/2 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 1/2 & 1/2 & 0 & 0 & 0 & 1 & 0 \\
1/4 & 1/4 & 1/4 & 1/4 & 0 & 0 & 0 & 1
\end{bmatrix}
$$
The multilevel stiffness matrix $\hat{K}$ can then be obtained as

$$
\begin{bmatrix}
\frac{b}{3a} + \frac{a}{3b} & \frac{b}{3a} & \frac{a}{3b} & \frac{b}{4a} & \frac{a}{4b} & \frac{b}{4a} & \frac{a}{4b} & 0 \\
\frac{b}{3a} & \frac{b}{3a} & \frac{b}{3a} & \frac{b}{4a} & \frac{a}{4b} & \frac{b}{4a} & \frac{a}{4b} & 0 \\
\frac{b}{3a} & \frac{b}{3a} & \frac{b}{3a} & \frac{b}{4a} & \frac{a}{4b} & \frac{b}{4a} & \frac{a}{4b} & 0 \\
\frac{b}{6a} & \frac{a}{6b} & \frac{a}{6b} & \frac{b}{3a} & \frac{a}{3b} & \frac{b}{3a} & \frac{a}{3b} & \frac{b}{6a} \\
\frac{b}{6a} & \frac{a}{6b} & \frac{a}{6b} & \frac{b}{3a} & \frac{a}{3b} & \frac{b}{3a} & \frac{a}{3b} & \frac{b}{6a} \\
\frac{b}{6a} & \frac{a}{6b} & \frac{a}{6b} & \frac{b}{3a} & \frac{a}{3b} & \frac{b}{3a} & \frac{a}{3b} & \frac{b}{6a} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
$$

Note that the first four rows and columns $\hat{K}$ correspond exactly to the stiffness matrix of the parent element.

The multiscale load vector can be obtained in exactly the same manner as the stiffness matrix and its derivation is not discussed here.

While the above formulation is for a two level method, it can be applied recursively to any required level of subdivision.

### 3 Implementation

In this section, we briefly review some of the implementation details of our multilevel solver on a parallel architecture. Our implementation is in ANSI C and uses the MPI [3] library for communicating between the nodes. For solving the system of equation in parallel and to reconstruct the single-level solution, we used the Aztec[5] library.

#### 3.1 Data structures

**3.1.1 Data structures for storing the mesh**

For a single-level finite element implementation, one only needs to have data structures for elements and vertices. However, this is not suitable for a multilevel solver since additional operations such as subdividing the mesh must be performed and traversing the multilevel tree to assemble the stiffness matrix must be done efficiently.

To efficiently store and retrieve the geometry, we used a Vertex-Edge-Element datastructure which is described below:

1. **Vertex.** Each vertex stores its coordinates and a flag to indicate whether it is restrained or not.
2. **Edge.** Each edge stores the IDs of its bounding vertices and the ID of its two children edges. It also contains a flag to indicate whether it is restrained or not.

3. **Element.** Each element/face contains a list of four of its bounding edges along with the ID of the face vertex created during subdivision.

Note that we need to store the boundary conditions for nodes and edges since they both influence the boundary conditions for internal edges and nodes created during subdivision.

The vertex, edge and element data structures are implemented using arrays of C structs. To reduce runtime overhead, member access is provided using macros rather than functions. In addition to the mesh data structures, a separate data structure keeps track of the starting end ending edges and elements at a given level of subdivision.

In designing the above data structures, we have tried to strike a balance between the memory requirement, the cost of accessing the various topological elements and ease of implementation. In particular, our multilevel data structure has been implemented using arrays as opposed to linked-lists or quad-trees, which have significantly larger memory requirements. Moreover, the topological entities of interest can be accessed in constant time. For example, all the bounding vertices of an element can be determined using only eight memory references and all its child vertices can be accessed using only 12 memory references.

### 3.1.2 Data structure for sparse matrices

Our choice of data structure for sparse matrices was dictated by Aztec, since it has matrix-vector multiplication routines only for distributed matrices stored in the modified sparse-row and variable block-row formats.

The modified sparse-row (MSR) format stores sparse matrices using two arrays of length NNZ+1 (where NNZ is the number of non-zeros in the matrix), an index array, \( b\text{index} \) and a double precision array \( \text{val} \) containing the non-zero entries in the matrix, organized as follows:

\[
\begin{align*}
\text{bindex}[0] &= \text{NR} + 1, \text{ where NR is the number of rows in the matrix} \\
\text{bindex}[i+1] - \text{bindex}[i] \ (0 \leq i < \text{NR}) &= \text{Number of non-zero off-diagonal entries in the } i\text{th row of the matrix} \\
\text{bindex}[i] \ (\text{NR} + 1 \leq i \leq \text{NNZ}) &= \text{Column indices of the non-zero off-diagonal entries stored row-wise} \\
\text{val}[i] \ (0 \leq i < \text{NR}) &= \text{Diagonals of the matrix} \\
\text{val}[\text{NR}] &= \text{Unused} \\
\text{val}[i] \ (\text{NR} + 1 \leq i \leq \text{NNZ}) &= \text{Non-zero offdiagonals of the matrix stored row-wise.}
\end{align*}
\]

As an example, for the stiffness matrix

\[
K = \begin{bmatrix}
1 & -1 & 0 & 0 & 0 \\
-1 & 2 & -1 & 0 & 0 \\
0 & -1 & 2 & -1 & 0 \\
0 & 0 & -1 & 2 & -1 \\
0 & 0 & 0 & -1 & 1 \\
\end{bmatrix}
\]
the arrays \texttt{bindex} and \texttt{val} are given as:

\begin{verbatim}
bindex = [6, 7, 9, 11, 13, 14, 1, 0, 2, 1, 3, 2, 4, 3]
val = [1, 2, 2, 2, 1, *, -1, -1, -1, -1, -1, -1, -1]
\end{verbatim}

3.1.3 Data input and output formats

Our program requires the user to input only the data for the base mesh. The refined meshes are then automatically created by subdividing the elements and edges in the coarsest level mesh. The input data is organized as follows:

<table>
<thead>
<tr>
<th>N levels</th>
<th>Finest level desired (unsigned int)</th>
</tr>
</thead>
<tbody>
<tr>
<td>N DoF</td>
<td>Number of vertices at the coarsest level (unsigned int)</td>
</tr>
<tr>
<td>N edges</td>
<td>Number of edges at the coarsest level (unsigned int)</td>
</tr>
<tr>
<td>N elements</td>
<td>Number of elements at the coarsest level (unsigned int)</td>
</tr>
</tbody>
</table>

\begin{verbatim}
x_0 y_0 bc_0
x_1 y_1 bc_1
\vdots
\end{verbatim}

\begin{verbatim}
Orig_0 Dest_0 bc_0
Orig_1 Dest_1 bc_1
\vdots
\end{verbatim}

<table>
<thead>
<tr>
<th>(E_0)</th>
<th>(E_1)</th>
<th>(E_2)</th>
<th>(E_3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Origin, Destination and Boundary Condition of edges (resp. unsigned int, unsigned int and boolean)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bounding edges of elements in CCW order (resp. unsigned int, unsigned int, unsigned int, unsigned int)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For the edges, the origin and destination are chosen such that \(x_{\text{origin}} \leq x_{\text{destination}}\) and \(y_{\text{origin}} \leq y_{\text{destination}}\). For elements, the first edge is chosen to be right-most edge in the element.

The output data is organized in two files. The first file contains the x-coordinate, y-coordinate and solution at the vertices at the different subdivision levels while the second file contains a list of elements at the different levels with each element entry consisting of a list of its four bounding vertices.

3.2 Data organization on a parallel architecture

3.2.1 Mesh partitioning algorithm

For distributing the degrees of freedom at a particular level between the different processors, we use a simple linear partitioning rule which allocates roughly \(N_{\text{DoF}}/N_{\text{P}}\) degrees of freedom to each processor. The degrees of freedom are assigned to the processors without reordering and hence the partitioning rule will not in general be optimal from the point of view of reducing the communication between the processors.

Following the terminology used in Aztec[5], we denote the degrees of freedom assigned to processor \(P\) as its \textit{update set}, \(U_P\). Each processor then stores and operates with the elements of the load-vector, \(\hat{f}\), the rows of the stiffness matrix, \(\hat{K}\) and the elements of the solution vector \(\hat{u}\) corresponding to its update set.
In addition to its update set, each processor also has information about all the finite elements connected to each vertex in its update set. However, in our current implementation, all processors have information about all vertices (coordinates and boundary conditions) and edges (connectivity and boundary conditions) in the global mesh. Not doing so would have required a global to local mapping for the edges and vertices for each processor which would have significantly increased the complexity of the code. Our implementation, while being more memory intensive works acceptably well for 2D problems. For 3D problems, a better solution would be to pre-partition the mesh for each processor. Each processor could then assemble its local stiffness matrix in an almost embarrassingly parallel manner. The main disadvantage of this approach is that the number of processors needs to be fixed a priori.

### 3.2.2 Distribution of $\hat{K}$ and $W^{-1}$

As mentioned earlier, each processor assembles the rows of the stiffness matrix corresponding to the degrees of freedom in its update set. This results in the stiffness matrix (and the inverse wavelet transform matrix) being row-distributed. Since the dominant cost of the solution procedure is that of a matrix-vector product, row-distribution of the stiffness matrix does not hinder load-balancing. In fact, in all our numerical experiments, we observed excellent load-balancing between the processors for both the solution as well as the inverse transformation stages.

### 3.3 Algorithms

In this section, we describe the key algorithms in our implementation. The basic pseudocode is as follows (steps marked with a * are executed in parallel):

```plaintext
start
  read input data
  subdivide the mesh upto N\_levels
  * broadcast nodes and edges (all levels) to all processors
  for level = 0, 1, \ldots, N\_levels do
    assign node IDs and elements at current level to all processors
    * assemble stiffness matrix
    * solve system of equations
    * perform inverse wavelet transform to obtain the solution vector
    * consolidate solution at the root node
    use solution as guess to next level of refinement
  endfor
finish
```
3.3.1 The Oracle

The oracle consists of a set of functions that are performed at the root node. The main oracle functions are:

1. **Mesh Subdivision.** The mesh subdivision step involves creating new face and edge vertices along with edges and elements at the finer level. The pseudocode for mesh subdivision can be written as

   ```plaintext
   start
   for each element k in the current level do
       obtain list of edges, E and nodes, N of k
       for each edge j ∈ E do
           if j is not subdivided do
               create a new edge vertex
               create two child edges
               determine boundary conditions for the edge vertex and new edges
           endif
       endfor
   create face vertex
   create four new internal edges
   create four new child elements
   endfor
   finish
   ```

2. **Mesh Partitioning.** As mentioned in the previous section, the degrees of freedom in the mesh are distributed linearly between the processors.

3. **Solution Consolidation.** Once the system of equations has been solved in parallel, consolidating the solution on the root node is rather straightforward. In our implementation, this was done using `MPI_Gatherv`.

4. **Solution Refinement.** Solution refinement involves solving the problem on a subdivided mesh using the solution obtained on the coarse mesh as the initial guess. The vector of details in the subdivided mesh is initially set to all zeros. Recall that for smooth solutions, the details at each level tend to diminish with the number of levels of refinement. Hence, we expect that as we solve the problem on finer and finer meshes, the iterative solver will converge in only a few iterations. While in practice this ideal condition is not observed (since there is coupling between the coarse solution and the details), the number of iterations for convergence is still much less compared to solving the problem right at the finest level.
3.3.2 Assembly

The assembly of the stiffness matrix is embarrassingly parallel since all processors have the required vertex and edge information. The basic procedure is as follows:

```
start
  for each element \( k \) in the current level do
    obtain list of edges, \( E \) and nodes, \( N \) of \( k \)
    compute the dimensions \( a, b \) of the element
    assemble the element stiffness matrix from Equation 8
  endfor
  for each vertex \( v \in N \) do
    if \( v \in U_p \),
      if \( v \) is restrained, \( \hat{K}[v, v] = 1.0 \) and \( \hat{f}[v] = 0.0 \)
    else add contribution of element \( k \) to \( \hat{K}[v, :] \) and \( \hat{f}[v] \)
  endfor
finish
```

3.3.3 Solution

The system of equations is solved using the preconditioned conjugate gradient solver with symmetric diagonal scaling provided by Aztec. The iterations were performed till the relative norm of the residual dropped to 10\(^{-6}\) or till the number of iterations reached 500.

3.3.4 Reconstruction

The reconstruction step involves performing one level of inverse transform on the (distributed) solution vector. The most convenient way of implementing this is to directly assemble the inverse transform matrix, \( W^{-1} \) and then perform a parallel matrix-vector multiply. The construction of \( W^{-1} \) is carried out in a manner similar to the assembly of the multiscale stiffness matrix (see Equation 7) and is not discussed in greater detail in this report.

3.4 Running the program

Once the input file, say `mesh.txt` has been prepared in the required format, the program can be executed as:

```
mpirun -np N ./app_main.exe mesh.txt
```

The output files containing the solution are `mesh_sol.out` and `mesh_elem.out`. The number of iterations and solution times for each processor are stored in the file `mesh_time_proc_n.out`, \( 0 \leq n < N \).
4 Numerical Results

4.1 Square Domain

We first consider a square domain of length 2 units with homogeneous Dirichlet boundary conditions on all edges. The coarsest mesh had nine degrees of freedom (four elements) and the finest mesh had 4225 degrees of freedom (4096 elements). Table 4.1 summarizes the key results of this example. The number of iterations for convergence is $O(j \log \epsilon)$ [7] and hence we expect the number of iterations to grow linearly with the subdivision level. The results from table 4.1 seem to be slightly suboptimal in this respect. This could be because our implementation is essentially a two-level scheme rather than a full multilevel scheme. Figure 3(a)–(f) shows the solutions obtained at different levels of resolution.

Figure 4 shows the convergence of the solution in the infinity norm. Note that we achieve the same order of convergence (in this case $O(h^2)$) as single-level finite element methods for the same family of elements.

In Figure 5 we compare the number of iterations at the finest level with number of degrees of freedom for single-level and multilevel methods using four processors. We observe that using a hierarchical approach, one needs to perform lesser number of iterations on finer grids especially for a large number of degrees of freedom.

Figure 6 shows the solution time of the iterations at Level 8 (66049 DoF) with the number of processors. We timed the solver in two ways: the red line refers to the solution time reported by Aztec which accounts only for the solver iterations and not the setup time for the solver. The blue line indicates the time for the entire solution which takes into account the time spent by Aztec for setting up the necessary datastructures within the solver and synchronizing all the processors before starting the iterations. While it appears that the setup time dominates the time of the entire solution, we believe that for very large problems, the overall time will be governed by the time taken for the actual solution rather than the initial setup time. Hence, it can be safely conjectured that for larger problems there will be significant speedups by using more processors. Note that in our timing experiments, we did not take into account the time spend for computing the inverse wavelet transform of the solution; since the inverse transform is only a single sparse matrix-vector multiplication, its contribution to the overall cost of the solver is negligible.

<table>
<thead>
<tr>
<th>Subdivision Level</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Degrees of freedom</td>
<td>9</td>
<td>25</td>
<td>81</td>
<td>289</td>
<td>1089</td>
<td>4225</td>
</tr>
<tr>
<td>Number of Iterations</td>
<td>1</td>
<td>3</td>
<td>10</td>
<td>17</td>
<td>26</td>
<td>43</td>
</tr>
<tr>
<td>Time (sec)</td>
<td>0.004</td>
<td>0.006</td>
<td>0.013</td>
<td>0.019</td>
<td>0.039</td>
<td>0.123</td>
</tr>
</tbody>
</table>

Table 1: Results for the square domain

13
Figure 3: Solutions at levels 1 through 6; (a) 9 DoF, (b) 25 DoF, (c) 81 DoF, (d) 289 DoF and (e) 1089 DoF (f) 4225 DoF
Figure 4: Convergence of the solution in the infinity norm

Figure 5: Number of iterations at the finest level for the single-level and hierarchical methods

Figure 6: Solution time vs. Number of processors for level 8 (66049 DoFs)


4.2 L Domain

The next problem considered was a L shaped domain enclosed inside the square domain of the previous example. For this domain, we considered six subdivision levels starting with 21 degrees of freedom and ending with 12545 degrees of freedom. Table 2 summarizes the key results for running this example on four processors.

Figure 7 shows the solutions obtained at levels 1 and 4. Observe that due to the geometric increase in the number of degrees of freedom due to uniform refinement and the fact that the solution is smooth over the domain, we obtained rapid convergence to the true solution with only a few refinement levels. Figure 8 compares the number of iterations on the finest grid for the single-level and hierarchical methods. As in the previous example, we observe that for large problems, the hierarchical methods converge in a smaller number of iterations. In Figure 9, we compare the time taken to solve the problem at level 6 with different numbers of processors. A trend similar to the one in the previous example is observed, i.e., the solution time comes down whereas the total time increases due to setup time for the different processors.

<table>
<thead>
<tr>
<th>Subdivision Level</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Degrees of freedom</td>
<td>21</td>
<td>65</td>
<td>225</td>
<td>833</td>
<td>3201</td>
<td>12545</td>
</tr>
<tr>
<td>Number of Iterations</td>
<td>3</td>
<td>11</td>
<td>18</td>
<td>28</td>
<td>49</td>
<td>94</td>
</tr>
<tr>
<td>Time (sec)</td>
<td>0.005</td>
<td>0.013</td>
<td>0.018</td>
<td>0.030</td>
<td>0.109</td>
<td>0.280</td>
</tr>
</tbody>
</table>

Table 2: Results for the L shaped domain

Figure 7: Solution over the L shaped domain at (a) Level 1 (21 DoF) and (b) Level 4 (833 DoF)
Figure 8: Number of iterations at the finest level for the single-level and hierarchical methods.

Figure 9: Solution time vs. Number of processors for level 8.
4.3 MIT domain

The final domain we considered consisted of the letters M, I and T concatenated together. The base mesh had 132 degrees of freedom and the finest level mesh had 91520 degrees of freedom. Table 3 shows the main results for this example. The results for subdivision levels 1 through 5 were obtained on four processors whereas the results for level 6 were obtained on eight processors.

Figure 10 shows the solutions at levels 1 and 3. Observe once again that the solution converges rapidly with only a few refinement levels.

Figure 11 compares the number of iterations at the finest level for the single-level and hierarchical basis methods. Once again, we observe that for large problems, the HB method requires fewer number of iterations since the stiffness matrix is better conditioned. Figure 12 shows the solution times with the number of processors. We again observe the same trend as in the previous two examples.

<table>
<thead>
<tr>
<th>Subdivision Level</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Degrees of freedom</td>
<td>132</td>
<td>440</td>
<td>1584</td>
<td>5984</td>
<td>23232</td>
<td>91520</td>
</tr>
<tr>
<td>Number of Iterations</td>
<td>15</td>
<td>22</td>
<td>35</td>
<td>64</td>
<td>119</td>
<td>219</td>
</tr>
<tr>
<td>Time (sec)</td>
<td>0.012</td>
<td>0.02</td>
<td>0.042</td>
<td>0.233</td>
<td>0.510</td>
<td>4.770</td>
</tr>
</tbody>
</table>

Table 3: Results for the MIT domain

Figure 10: Solution over the MIT domain at (a) Level 1(132 DoF) and (b) Level 3 (1584 DoF)
Figure 11: Number of iterations at the finest level for the single-level and hierarchical methods

Figure 12: Solution time vs. Number of processors for level 8
5 Conclusions and Further work

In this report, we described the implementation of a parallel solver for the Poisson equation based on a hierarchical basis rather than on a nodal basis discretization. We compared our implementation against a single-level finite element solver and as expected, observed significant performance gains for larger problems in terms of the number of iterations.

In our timing experiments, we observed that due to the relatively low network bandwidth, there was a significant overhead associated with the initial synchronization steps performed by Aztec. However, the time for carrying out the iterations did reduce, which demonstrates the good scaling characteristics of the hierarchical basis approach. We believe that with a fast network and for larger problems, the initial setup time will no longer dominate the cost of solution.

5.1 Further Work

We would like to suggest the following extensions of the current implementation:

1. Optimization of mesh partitioning. Currently, the geometry is replicated on all processors. A more elegant solution would be to construct a local-to-global mapping for each processor and distribute only the required parts of the mesh to the processors. Another possible optimization is to renumber the mesh on the fly before partitioning to reduce communication overhead during solution.

2. Implementation of a full hierarchical solver. Our program implements a recursive two-level solver which is slightly suboptimal in terms of conditioning. Hence a natural extension would be to implement a fully hierarchical solver.

3. Implementation of additional element families. The tensor product elements considered in this report are restricted in terms of the geometry that they can represent. Since most of the framework has been laid out in the project, it would be rather straightforward to extend our formulation to triangular and more general quadrilateral elements.

4. Experiments with stabilized wavelets. The hierarchical bases considered in this report are not asymptotically Reisz stable. One way to stabilize them is to orthogonalize the wavelets using a Gram-Schmidt procedure, which results in constant condition numbers across all levels. However, these stabilized wavelets have a larger span than the classical hierarchical bases and hence could involve additional overhead in terms of communication. However, the use of locally supported orthogonal approximations is an interesting area to be explored.
References


### A Source code highlights

In this section, we present a few code snippets from our implementation. For brevity, we present only the significant code blocks without variable declarations. The complete source code can be downloaded at [http://wavelets.mit.edu/~darshan/18.337/app.tar.gz](http://wavelets.mit.edu/~darshan/18.337/app.tar.gz)

#### A.1 The driver program

```c
#include "aztec.h"
#include "app_prototypes.h"
#include "app_global_vars.h"
#include "app_oracle.h"
#include "app_utils.h"
#include "app_fe_code.h"

int main(int argc, char* argv[])
{
    app_enter(&argc, &argv);

    /* Tell Aztec about the number of processors */
    AZ_set_proc_config(proc_config, MPI_COMM_WORLD);

    /* Construct user-defined MPI data types */
    app_type_constructor();

    N_procs = proc_config[AZ_N_procs];
    myID = proc_config[AZ_node];

    if(!myID){
        app_oracle_read_geometry(&global_mesh, &info, argv[1]);
        app_oracle_subdivide_mesh(&global_mesh, &info);
    }

    MPI_Bcast(&N_levels, 1, MPI_INT, 0, MPI_COMM_WORLD);

    app_oracle_bcast_nodes_edges(proc_config, &global_mesh, &local_mesh, &info, N_levels-1);

    for(level = 0; level < N_levels; level++) {
        app_oracle_assign_elements(proc_config, &update, N_update_arr, &global_mesh, &local_mesh, &info, level);
        app_init_solver_options(options, params);
        app_init_datastructs(proc_config, &data_org, &update_index, update, &extern_index, &external,
                             &bindex, &val, &rhs, &sol, &local_mesh);
        app_oracle_assign_guess(proc_config, data_org, update_index, N_update_arr, sol, global_sol, &info, level);
        app_solve(proc_config, data_org, update_index, update, extern_index, external, options, params,
                  sol, rhs, bindex, val, &local_mesh, status);
        app_oracle_consolidate_solution(proc_config, N_update_arr, sol, &global_sol);
        app_finalize_datastructs(&data_org, &update_index, &update, &extern_index, &external,
                                 &bindex, &val, &rhs, &sol);
    }

    app_finalize_mesh(&local_mesh);
    app_exit(0);
}
```

22
A.2 The oracle

A.2.1 Mesh subdivision

In the following, lines 12 to 31 create new edge vertices and new edges whereas lines 32 to 56 create new a face vertex, four internal edges and four new elements at the next level. Notice how all access to the mesh datastructure is provided via macros rather than via function calls. This enables significant speedups since macro definitions do not have the overhead of functions.

```c
void app_oracle_subdivide_mesh(Mesh* mesh, const MeshInfo* info)
{
    for (l=0;l<N_levels;l++)
    {
        startidx = info->elem_info[l]; endidx = info->elem_info[l+1];
        for (k=startidx;k<endidx;k++)
        {
            edgeidx[0] = ELEM_EDGE(k, 0); edgeidx[1] = ELEM_EDGE(k, 1);
            edgeidx[2] = ELEM_EDGE(k, 2); edgeidx[3] = ELEM_EDGE(k, 3);
            nodeidx[0] = EDGE_NODE(edgeidx[0], 1); nodeidx[1] = EDGE_NODE(edgeidx[1], 0);
            nodeidx[2] = EDGE_NODE(edgeidx[2], 0); nodeidx[3] = EDGE_NODE(edgeidx[3], 1);
            for (i=0;i<4;i++)
            {
                if (!mesh->edges[edgeidx[i]].edge_kinder[0])
                {
                    orig = mesh->edges[edgeidx[i]].edge_nodes[0];
                    dest = mesh->edges[edgeidx[i]].edge_nodes[1];
                    NODE_X(N_DoF) = 0.5*(NODE_X(orig)+NODE_X(dest));
                    NODE_Y(N_DoF) = 0.5*(NODE_Y(orig)+NODE_Y(dest));
                    EDGE_IS_REST(N_DoF) = (EDGE_IS_REST(nodeidx[(i+4)%4]) && (EDGE_IS_REST(nodeidx[(i+3)%4]) &&
                                  (EDGE_IS_REST(edgeidx[i])));
                    EDGE_SET_NODES(N_edges , orig , N_DoF); EDGE_SET_NODES(N_edges+1, N_DoF, dest);
                    EDGE_SET_KINDER(N_edges, 0, 0); EDGE_SET_KINDER(N_edges+1, 0, 0);
                    EDGE_SET_KINDER(edgeidx[i], N_edges, N_edges+1);
                    N_DoF++;
                    N_edges += 2;
                }
            }
            NODE_X(N_DoF) = 0.25*(NODE_X(nodeidx[0])+NODE_X(nodeidx[1])+NODE_X(nodeidx[2])+NODE_X(nodeidx[3]));
            NODE_Y(N_DoF) = 0.25*(NODE_Y(nodeidx[0])+NODE_Y(nodeidx[1])+NODE_Y(nodeidx[2])+NODE_Y(nodeidx[3]));
            EDGE_IS_REST(N_DoF) = FALSE;
        }
    }
```
A.2.2 Mesh partitioning

```c
void apply_oracle_assign_elements(int proc_config[], int** update, int N_update_arr[],
const Mesh* global_mesh, Mesh* local_mesh, const MeshInfo* info,
int level)
{
    if(!myID){
        /* Start by assigning DoFs to the root node */
        N_update = temp_udof/temp_np;
        N_update_arr[0] = N_update;
        s_node_idx = 0;
        e_node_idx = s_node_idx + N_update - 1;
        N_elements = 0;
        *update = (int*) AZ_allocate(sizeof(int)*N_update);

        /* Assign the first \( N_{\text{DoF}} / N_{\text{procs}} \) degrees of freedom to the root node */
        for (i=s_node_idx;i<=e_node_idx;i++) (*update)[i] = i;

        /* Go through all the elements at the coarser level and see if */
        /* any of the nodes in the update set belong to it. */
        /* If it does, update that nodes local_element_id. */
        /* */
        for (l = info->elem_info[level]; l < info->elem_info[level+1]; l++){
            ELEM_FILL_NODES(l, nodeidx, edgeidx);
            for (i = 0; i<9; i++){
                row = AZ_quick_find(nodeidx[i], (*update), N_update, update_shift, update_bin);
                if (row != -1){
                    local_element_ids[N_elements++] = l;
                    break;
                }
            }
        }
    }
    for (i=0;i<N_elements;i++)
```

local_mesh->elements[i] = global_mesh->elements[local_element_ids[i]];

temp_ndof -= N_update;
temp_np --;

/* Assign chunks linearly to other nodes */
for (k = 1; k < N_procs; k++){
    buf_nupdate = temp_ndof/temp_np;
    N_update_arr[k] = buf_nupdate;
    s_node_idx = e_node_idx+1;
    e_node_idx = s_node_idx + buf_nupdate-1;

    buf_elements = 0;
    for (i=s_node_idx; i <= e_node_idx; i++) {
        buf_nupdate[i-s_node_idx] = i;
        buf_nodes[i-s_node_idx] = global_mesh->nodes[i];
    }

    for (l = info->elem_info[level]; l < info->elem_info[level+1]; l++){
        ELEM_FILL_NODES(l,nodeidx,edgeidx);
        for (i = 0; i < 9; i++){
            row = AZ_quick_find(nodeidx[l], buf_nupdate, buf_nupdate, update_shift, update_bin);
            if(row != -1){
                local_element_ids[buf_n_elements++] = l;
                break;
            }
        }
        MPI_Send(&buf_n_update , 1, MPI_UNSIGNED, k, 0, MPI_COMM_WORLD);
        MPI_Send(&buf_n_elements , 1, MPI_UNSIGNED, k, 0, MPI_COMM_WORLD);
        MPI_Send(buf_n_update , buf_n_elements , MPI_ELEMENT , k, 0, MPI_COMM_WORLD);
    }
    temp_ndof -= buf_n_update;
temp_np --;
}

else{
    MPI_Status cnt_status, node_status, element_status;
    int buf_n_update, buf_n_elements;
    MPI_Recv(&N_update , 1, MPI_UNSIGNED, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &cnt_status);
    MPI_Recv(&Nl_elements, 1, MPI_UNSIGNED, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &cnt_status);

    update = (int*)AZ_allocate(Nl_update*sizeof(int));
    MPI_Recv(update , Nl_update , MPI_INT , 0, MPI_ANY_TAG, MPI_COMM_WORLD, &node_status);
    MPI_Recv(local_mesh->elements, Nl_elements, MPI_ELEMENT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &element_status);
}
A.3 The multilevel solver

The Aztec library comes with an extensive set of example programs using distributed sparse matrices. The code for allocating an MSR matrix and for filling in an entry to the matrix were adapted from those examples.

A.4 Sparse matrix allocation

```c
void app_spalloc(int** bindex, double** val, int Nnz)
{
    int Nelements = Nnz+1;
    *bindex = (int*) AZ_allocate(Nelements*sizeof(int));
    *val = (double*) AZ_allocate(Nelements*sizeof(double));

    if(*val == NULL)
        app_abort("Insufficient memory

    for (i = 0;i < Nelements;(*bindex)[i] = -1, (*val)[i] = 0.0, i++)
        (*bindex)[0] = Nupdate+1;

    for (i = 0;i < Nupdate;i++)
        (*bindex)[i+1] = (*bindex)[i]+Nnz-1;
}
```

A.4.1 Filling an entry in a sparse matrix

```c
void app_set_element(int bindex[], double val[], int row, int col, double value, int is_diag)
{
    if(is_diag)
        val[row] = value;
    else {
        int i;
        int start = bindex[row], end = bindex[row+1];
        for (i = start; i < end; i++)
            if(bindex[i] == -1 || bindex[i] == col){
                bindex[i] = col;
                val[i] = value;
                break;
            }
    }
}
```
A.5 Stiffness matrix assembly

```c
void app_create_fe_matrix_and_load_vector(int update[], int* bindex[], double* val[], double* rhs[],
                                                                      const Mesh* mesh, int myID)
{
    double rfac[] = {0.25,0.25,0.25,0.25,0.125,0.125,0.125,0.125,0.25};
    /**< Allocate space for the stiffness matrix and right-hand side */
    *rhs = (double*) malloc(N_update*sizeof(double));

    for (k = 0; k < Nl_elements; k++){
        ELEM_FILL_NODES(k, nodeidx, edgeidx);
        a = (mesh->nodes[nodeidx[0]].x) - (mesh->nodes[nodeidx[1]].x);
        b = (mesh->nodes[nodeidx[1]].y) - (mesh->nodes[nodeidx[3]].y);

        for (i = 0; i < 9; i++){
            row = FIND_IF_INUPDATE(nodeidx[i]);
            if(row == -1)
                continue;
            else if(NODE_IS_REST(nodeidx[i])){
                app_set_element(*bindex, *val, row, nodeidx[i], 1.0, TRUE);
                (*rhs)[row] = 0.0;
            }
            else{
                for (j = 0; j < 9; j++)
                    (NODE_IS_REST(nodeidx[j]) == 0)?
                        app_add_element(*bindex, *val, row, nodeidx[j], K[i][j], i==j):0;
                (*rhs)[row] += a*b*rfac[i];
            }
        }
    }
    /**< Remove extra allocated memory */
    app_compress_matrix(*bindex, *val);
}
```