

A Fast and Stable Adaptive Solver for Hierarchically Semi-separable Representations

S. Chandrasekaran* M. Gu† W. Lyons‡

May 14, 2004

1 Abstract

An optimal backward stable solver is presented for matrices in hierarchically semi-separable form with highly non-uniform partition trees. The algorithm is based on an implicit ULV factorization. Numerical experiments are provided.

2 Introduction

In [4] we introduced the concept of hierarchically semi-separable (HSS) representations for matrices for their potential in speeding up the solution of linear equations that arise from the discretization of partial differential and integral equations. The paper presented fast and stable algorithms for constructing HSS representations, and also fast and stable algorithms for solving systems of equations with HSS representations. However, [4] only considered the case when the binary tree underlying the HSS representation was essentially a complete uniform tree: every node had exactly two children, and lengths of all the paths from leaves to the root was identical (or nearly so).

In this paper we overcome that restriction and present a fast, numerically backward stable solver for the case when the binary tree is not uniform. The algorithm is based on the one presented in [4]. It uses an implicit ULV factorization, where U and V are unitary matrices and L is a lower-triangular matrix.

We begin by re-introducing the HSS representation for the sake of completeness, and then presenting the algorithm on a 4×4 block matrix case. We then present the full algorithm along with numerical timings and error reports.

*Electrical and Computer Engineering department, University of California, Santa Barbara, CA 93106-9560. email: shiv@ece.ucsb.edu.

†Mathematics department, University of California, Berkeley, CA. email: mgu@math.berkeley.edu.

‡Mathematics department, University of California, Santa Barbara, CA 93106-9560. email: lyons@math.ucsb.edu.

3 HSS Representation

The HSS representation of a matrix A is based on an hierarchical block row and column partitioning. This hierarchical partitioning is represented by a binary tree. For example, if a single row and column partition is used to break up the matrix into a 2×2 block matrix, then the HSS representation is given by

$$A = \begin{matrix} & \begin{matrix} n_1 & n_2 \end{matrix} \\ \begin{matrix} m_1 \\ m_2 \end{matrix} & \begin{pmatrix} A_{1;1,1} & A_{1;1,2} \\ A_{1;2,1} & A_{1;2,2} \end{pmatrix} \end{matrix} = \begin{pmatrix} D_{1;1} & U_{1;1}B_{1;1,2}V_{1;2}^H \\ U_{1;2}B_{1;2,1}V_{1;1}^H & D_{1;2} \end{pmatrix}$$

with the partition tree shown in figure 1. In the above partition the meaning (or role) of the various matrices is quite clear, and reveals that the primary intention of the HSS representation is to capture any low-rank structure present in the off-diagonal blocks of the matrix.

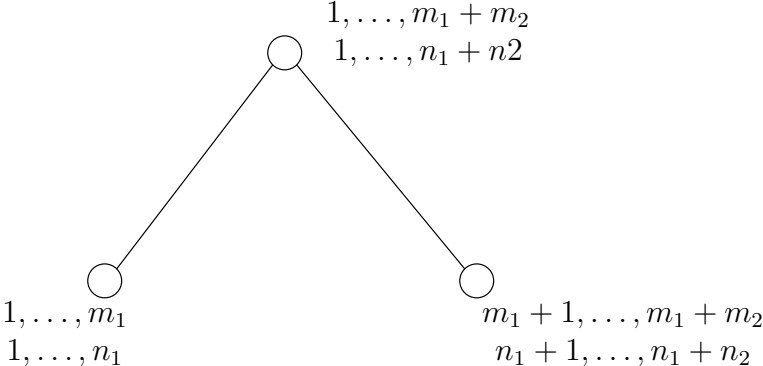


Figure 1: 2×2 HSS partition tree.

For convenience the elements of the HSS representation are also depicted on the partition tree as shown in figure 2.

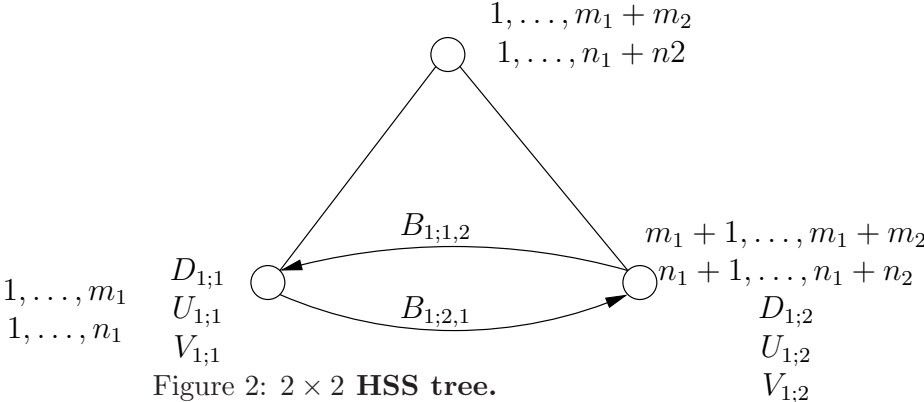


Figure 2: 2×2 HSS tree.

Due to space problems, and also since it is usually not necessary, we will not show the partitions themselves on the HSS tree any more.

However, the 2×2 example does not fully demonstrate the HSS representation. For that we must look at a

4×4 example. The following equation shows the recursive second set of partitions of A in forming its HSS representation:

$$A = \begin{pmatrix} \begin{pmatrix} D_{2;1} & U_{2;1}B_{2;1,2}V_{2;2}^H \\ U_{2;2}B_{2;2,1}V_{2;1}^H & D_{2;2} \end{pmatrix} & (U_{1;1}B_{1;1,2}V_{1;2}^H) \\ (U_{1;2}B_{1;2,1}V_{1;1}^H) & \begin{pmatrix} D_{2;3} & U_{2;3}B_{2;3,4}V_{2;4}^H \\ U_{2;4}B_{2;4,3}V_{2;3}^H & D_{2;4} \end{pmatrix} \end{pmatrix}.$$

Note that only the two diagonal blocks from the 2×2 partition are partitioned again. However, this is **not** the HSS representation of A . For that we need to introduce the **translation** operators $R_{k;i}$ and $W_{k;i}$ which are defined so that

$$\begin{aligned} U_{1;1} &= \begin{pmatrix} U_{2;1}R_{2;1} \\ U_{2;2}R_{2;2} \end{pmatrix} \\ U_{1;2} &= \begin{pmatrix} U_{2;3}R_{2;3} \\ U_{2;4}R_{2;4} \end{pmatrix} \\ V_{1;1} &= \begin{pmatrix} V_{2;1}W_{2;1} \\ V_{2;2}W_{2;2} \end{pmatrix} \\ V_{1;2} &= \begin{pmatrix} V_{2;3}W_{2;3} \\ V_{2;4}W_{2;4} \end{pmatrix}. \end{aligned}$$

Because $U_{1;i}$ and $V_{1;i}$ can be recovered from $U_{2;i}$, $R_{2;i}$, $V_{2;i}$ and $W_{2;i}$, in the actual HSS representation of A we do not store $U_{1;i}$ and $V_{1;i}$. See figure 3.

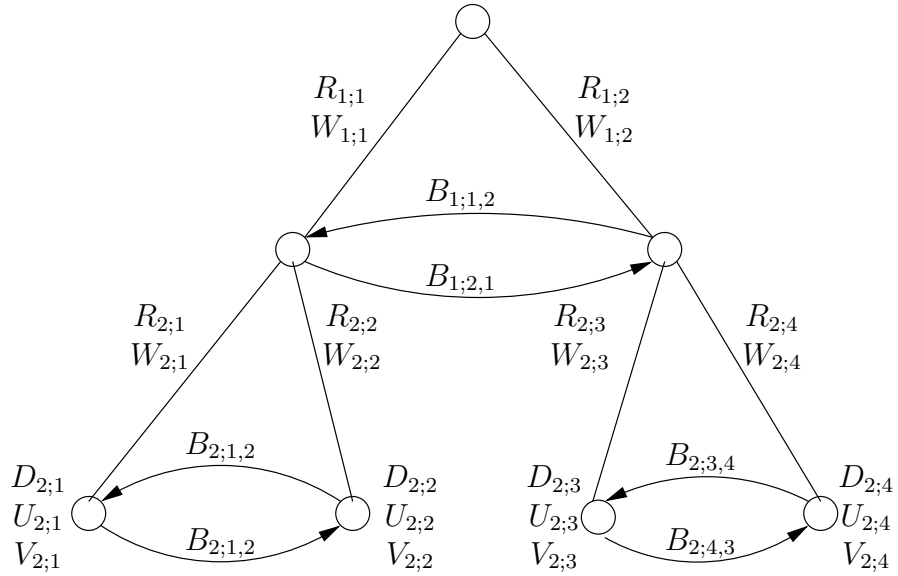


Figure 3: 4×4 HSS tree.

In [4] we presented a fast and stable solver for HSS representation where the binary tree was uniform, just like in figure 3. Paper [4] remarks that the algorithm will not be fast if the binary tree is highly non-uniform. In this paper we present an algorithm that is fast independent of the form of the binary tree. This is the common case in practice.

4 4×4 Example

Since the algorithm is fairly complicated, we begin by presenting it in the 4×4 block partitioned case. Let the system to be solved be $Ax = b$, where A is assumed to be in a 4×4 block-partitioned HSS form. Commensurate with that partitioning, we also partition x and b as follows:

$$x = \begin{pmatrix} x_{2;1} \\ x_{2;2} \\ x_{2;3} \\ x_{2;4} \end{pmatrix} \quad b = \begin{pmatrix} b_{2;1} \\ b_{2;2} \\ b_{2;3} \\ b_{2;4} \end{pmatrix}.$$

Following the ideas presented in [2, 3] we first modify the right-hand side as follows

$$b_{2;i} - U_{2;i}f_{2;i}, \quad i = 1, 2, 3, 4,$$

where $f_{2;i} = 0$ initially. This, of course, leaves the system unchanged. We now define f 's on all the nodes of the HSS tree, along with some g 's as follows:

$$g_{2;i} = 0, \tag{1}$$

$$g_{k;i} = W_{k+1;2i-1}^H g_{k+1;2i-1} + W_{k+1;2i}^H g_{k+1;2i}, \tag{2}$$

$$f_{0;1} = (), \tag{3}$$

$$f_{k+1;2i-1} = R_{k+1;2i-1} f_{k;i} + B_{k+1;2i-1,2i} g_{k+1;2i}, \tag{4}$$

$$f_{k+1;2i} = R_{k+1;2i} f_{k;i} + B_{k+1;2i,2i-1} g_{k+1;2i-1}. \tag{5}$$

Note that these recursions are consistent with $f_{2;i} = 0$.

The role of the f and g variables might appear mysterious right now, but they are actually the intermediate variables that appear during the fast multiplication of a matrix in HSS form with a vector. See [4]. Their usefulness will become clearer shortly.

We are now ready to describe the fast solver in a recursive manner. The algorithm involves two passes of the HSS tree. The first pass is depth-first. We begin at the left-most leaf. There are two possibilities.

4.1 Case 1: Compression

The first possibility is that the $U_{2;1}$ has more rows than columns. In this case we proceed to compress $U_{2;1}$ by computing its QL factorization

$$U_{2;1} = q_{2;1} \begin{pmatrix} 0 \\ \hat{U}_{2;1} \end{pmatrix}.$$

We then apply $q_{2;1}^H$ to the first row partition of the system. First, the right hand-side changes as follows:

$$q_{2;1}^H b_{2;1} - \begin{pmatrix} 0 \\ \hat{U}_{2;1} \end{pmatrix} f_{2;1}.$$

We also need to apply $q_{2;1}^H$ to $D_{2;1}$. We do so and compute the LQ factorization of the resulting matrix

$$q_{2;1}^H D_{2;1} = \begin{pmatrix} \hat{D}_{2;1;1,1} & 0 \\ \hat{D}_{2;1;2,1} & \hat{D}_{2;1} \end{pmatrix} w_{2;1}. \quad (6)$$

Commensurate with the zero rows that have been introduced in $U_{2;1}$ we partition the right hand side:

$$q_{2;1}^H b_{2;1} = \begin{pmatrix} \tilde{b}_{2;1;1} \\ \tilde{b}_{2;1;2} \end{pmatrix}.$$

We now apply $w_{2;1}^H$ to the first column partition of A . Its effect on $D_{2;1}$ is already known, so we only need to apply it on $V_{2;1}$. We do so and partition it in conformance with equation (6):

$$w_{2;1} V_{2;1} = \begin{pmatrix} \hat{V}_{2;1;1} \\ \hat{V}_{2;1} \end{pmatrix}.$$

Now that we have applied $w_{2;1}^H$ to the first column partition we must apply $w_{2;1}$ to $x_{2;1}$. We do so and partition it in a suitable manner:

$$w_{2;1} x_{2;1} = \begin{pmatrix} z_{2;1} \\ \hat{x}_{2;1} \end{pmatrix}.$$

At this stage the original equations $Ax = b - *$ has been converted to

$$\begin{pmatrix} \begin{pmatrix} \hat{D}_{2;1;1,1} & 0 \\ \hat{D}_{2;1;2,1} & \hat{D}_{2;1} \end{pmatrix} & \begin{pmatrix} 0 \\ \hat{U}_{2;1} \end{pmatrix} B_{2;1,2} V_{2;2}^H \\ \begin{pmatrix} U_{2;2} B_{2;2,1} (\hat{V}_{2;1;1}^H & \hat{V}_{2;1}^H) \\ (U_{1;2} B_{1;2,1} (W_{2;1}^H (\hat{V}_{2;1;1}^H & \hat{V}_{2;1}^H) & W_{2;2}^H V_{2;2}^H)) \end{pmatrix} & \begin{pmatrix} D_{2;2} \\ D_{2;3} \\ U_{2;4} B_{2;4,3} V_{2;3}^H \end{pmatrix} \\ \begin{pmatrix} \begin{pmatrix} 0 \\ \hat{U}_{2;1} R_{2;1} \end{pmatrix} B_{1;1,2} V_{1;2}^H \\ \begin{pmatrix} U_{2;2} R_{2;2} \\ U_{2;3} B_{2;3,4} V_{2;4}^H \\ D_{2;4} \end{pmatrix} \end{pmatrix} \end{pmatrix} \begin{pmatrix} z_{2;1} \\ \hat{x}_{2;1} \\ x_{2;2} \\ x_{2;3} \\ x_{2;4} \end{pmatrix}$$

equals

$$\begin{pmatrix} \tilde{b}_{2;1;1} \\ \tilde{b}_{2;1;2} \\ b_{2;2} \\ b_{2;3} \\ b_{2;4} \end{pmatrix} - \begin{pmatrix} 0 \\ \hat{U}_{2;1} f_{2;1} \\ U_{2;2} f_{2;2} \\ U_{2;3} f_{2;3} \\ U_{2;4} f_{2;4} \end{pmatrix}.$$

We observe that

$$\hat{D}_{2;1;1,1} z_{2;1} = \tilde{b}_{2;1;1}.$$

Since $\hat{D}_{2;1;1,1}$ is lower-triangular, we can obtain $z_{2;1}$ by forward substitution. Now that we know $z_{2;1}$ we can move the corresponding columns of the coefficient matrix to the right-hand side. However, that is a costly operation and must be done in a lazy manner. First we do the update

$$\hat{b}_{2;1} = \tilde{b}_{2;1;2} - \hat{U}_{2;1} f_{2;1} - \hat{D}_{2;1;2,1} z_{2;1}.$$

Then we carry out the rest of the subtraction from the right-hand side by setting

$$g_{2;1} = \hat{V}_{2;1;1}^H z_{2;1}.$$

Then via equations (1) to (5) the right-hand side is automatically updated, but the cost is kept low. This claim can be verified by some brute-force algebra on the explicit block-form of the equations given above.

Note that updating $g_{2;1}$ causes other g and f variables to be changed too. However, these changes will be computed as and when we traverse the tree during the recursive algorithm. For now, we only update $g_{2;1}$.

At this stage we observe that if we replace the HSS representation at the (2;1) leaf by hatted quantities we are left with a new system of equations to solve that is identical in form to the system we started with, except that it has fewer rows and columns.

$$\left(\begin{array}{cc} \hat{D}_{2;1} & \hat{U}_{2;1}B_{2;1,2}V_{2;2}^H \\ U_{2;2}B_{2;2,1}\hat{V}_{2;1}^H & D_{2;2} \end{array} \right) \begin{array}{c} (U_{1;1}B_{1;1,2}V_{1;2}^H) \\ \left(\begin{array}{cc} D_{2;3} & U_{2;3}B_{2;3,4}V_{2;4}^H \\ U_{2;4}B_{2;4,3}V_{2;3}^H & D_{2;4} \end{array} \right) \end{array} \begin{pmatrix} \hat{x}_{2;1} \\ x_{2;2} \\ x_{2;3} \\ x_{2;4} \end{pmatrix} = \begin{pmatrix} \hat{b}_{2;1} \\ b_{2;2} \\ b_{2;3} \\ b_{2;4} \end{pmatrix} - \begin{pmatrix} \hat{U}_{2;1}f_{2;1} \\ U_{2;2}f_{2;2} \\ U_{2;3}f_{2;3} \\ U_{2;4}f_{2;4} \end{pmatrix}.$$

So, by the magic of recursion, we can obtain the rest of the solution. Once that has been done we can recover $x_{2;1}$ using the formula

$$x_{2;1} = w_{2;1}^H \begin{pmatrix} z_{2;1} \\ \hat{x}_{2;1} \end{pmatrix}.$$

This part actually requires a second-pass of the HSS tree.

4.2 Case 2: Merging

We now consider the case when $U_{2;1}$ does not have more rows than columns. In this case, this node cannot be compressed. We proceed by considering node (2;2) now. The manner in which we treat is identical to the way in which we treated node (2;1). However, before we can do that we must remember to traverse the tree from leaf (2;1) to node (1;1) to leaf (2;2) updating the f and g variables along the way using equations (1) to (5):

$$f_{2;2} = R_{2;2}f_{1;1} + B_{2;2,1}g_{2;1}.$$

Now we can treat leaf (2;2) exactly as before. If $U_{2;2}$ has more rows than columns we do the compression step. In any case, after that, we proceed to node (1;1), the parents of leaves (2;1) and (2;2) and do a merge step. At the end of the merge step, the node (1;1) will be replaced with a leaf at (1;1). The HSS representation at this leaf is given by

$$\begin{aligned} \hat{D}_{1;1} &= \begin{pmatrix} \hat{D}_{2;1} & \hat{U}_{2;1}B_{2;1,2}\hat{V}_{2;2}^H \\ \hat{U}_{2;2}B_{2;2,1}\hat{V}_{2;1}^H & \hat{D}_{2;2} \end{pmatrix}, \\ \hat{U}_{1;1} &= \begin{pmatrix} \hat{U}_{2;1}R_{2;1} \\ \hat{U}_{2;2}R_{2;2} \end{pmatrix}, \\ \hat{V}_{1;1} &= \begin{pmatrix} \hat{V}_{2;1}W_{2;1} \\ \hat{V}_{2;2}W_{2;2} \end{pmatrix}. \end{aligned}$$

We must also remember to propagate g according to equations (1) to (5)

$$g_{1;1} = W_{2;1}^H g_{2;1} + W_{2;2}^H g_{2;2}.$$

At this stage we have a system that is identical in form to the one we started with, except that its HSS representation has two fewer leaves.

$$\left(\begin{array}{c} \hat{D}_{1;1} \\ (U_{1;2}B_{1;2,1}V_{1;1}^H) \end{array} \right) \begin{array}{c} (U_{1;1}B_{1;1,2}V_{1;2}^H) \\ \left(\begin{array}{cc} D_{2;3} & U_{2;3}B_{2;3,4}V_{2;4}^H \\ U_{2;4}B_{2;4,3}V_{2;3}^H & D_{2;4} \end{array} \right) \end{array} \begin{pmatrix} \hat{x}_{1;1} \\ x_{2;3} \\ x_{2;4} \end{pmatrix} = \begin{pmatrix} \hat{b}_{1;1} \\ b_{2;3} \\ b_{2;4} \end{pmatrix} - \begin{pmatrix} \hat{U}_{1;1}f_{1;1} \\ U_{2;3}f_{2;3} \\ U_{2;4}f_{2;4} \end{pmatrix}.$$

By the magic of recursion we can solve this system for the current unknowns. Once they become available $\hat{x}_{1,1}$ must be broken up as

$$\hat{x}_{1,1} = \begin{pmatrix} \hat{x}_{2,1} \\ \hat{x}_{2,2} \end{pmatrix},$$

and $\hat{x}_{2,1}$ must be propagated back to leaf (2;1) while $\hat{x}_{2,2}$ must be propagated back to leaf (2;2) where they can be used to finish the re-construction of $x_{2,1}$ and $x_{2,2}$ respectively.

5 Full Algorithm

We now describe the algorithm more formally. The algorithm is a tree-based recursive algorithm. We describe it for a depth-first traversal of the tree. Actually the tree is traversed twice.

Assume that the system to be solved is $Ax = b$, with A in HSS form. We first modify the right-hand side as follows

$$b_{k;i} - U_{k;i}f_{k;i},$$

where $f_{k;i}$ and $g_{k;i}$ satisfy the initial recursions

$$\begin{aligned} g_{k;i} &= 0, \quad (\text{at a leaf}) \\ g_{k;i} &= W_{k+1;2i-1}^H g_{k+1;2i-1} + W_{k+1;2i}^H g_{k+1;2i}, \quad (\text{at a node}) \\ f_{0,1} &= (), \\ f_{k+1;2i-1} &= R_{k+1;2i-1} f_{k;i} + B_{k+1;2i-1,2i} g_{k+1;2i}, \\ f_{k+1;2i} &= R_{k+1;2i} f_{k;i} + B_{k+1;2i,2i-1} g_{k+1;2i-1}. \end{aligned}$$

The behavior of the algorithm depends on whether we are at a leaf or a node.

5.1 Leaf

Suppose we are at leaf $(k;i)$.

5.1.1 Compression

First, we check if $U_{k;i}$ has more rows than columns. If it does we compress it using a QL factorization

$$U_{k;i} = q_{k;i} \begin{pmatrix} 0 \\ \hat{U}_{k;i} \end{pmatrix}.$$

Next we apply $q_{k;i}$ to $D_{k;i}$ and compute its LQ factorization

$$q_{k;i}^H D_{k;i} = \begin{pmatrix} \hat{D}_{k;i;1,1} & 0 \\ \hat{D}_{k;i;2,1} & \hat{D}_{k;i} \end{pmatrix} w_{k;i}.$$

Next we compute the relevant piece of the right-hand side

$$q_{k;i}^H b_{k;i} = \begin{pmatrix} \tilde{b}_{k;i;1} \\ \tilde{b}_{k;i;2} \end{pmatrix}.$$

We now apply $w_{k;i}$ to the rest of the first column partition

$$w_{k;i} V_{k;i} = \begin{pmatrix} \hat{V}_{k;i;1} \\ \hat{V}_{k;i} \end{pmatrix}.$$

We also change the unknowns suitably

$$w_{k;i} x_{k;i} = \begin{pmatrix} z_{k;i} \\ \hat{x}_{k;i} \end{pmatrix}.$$

Now we can solve for $z_{k;i}$

$$z_{k;i} = \hat{D}_{k;i;1,1}^{-1} \tilde{b}_{k;i;1}.$$

We can now move the corresponding columns to the right-hand side and update it

$$\hat{b}_{k;i} = \tilde{b}_{k;i;2} - \hat{U}_{k;i} f_{k;i} - \hat{D}_{k;i;2,1} z_{k;i}.$$

We do the rest of the update on the right-hand side lazily by updating $g_{k;i}$ instead

$$g_{k;i} = \hat{V}_{k;i;1}^H z_{k;i}.$$

The remaining system of equations is identical to the one we started with, except that it has fewer rows and columns. We solve it recursively. Then we can recover $x_{k;i}$ from the formula

$$x_{k;i} = w_{k;i}^H \begin{pmatrix} z_{k;i} \\ \hat{x}_{k;i} \end{pmatrix}.$$

5.1.2 No compression

If at a leaf $U_{k;i}$ is not compressible, then there is nothing to do! Recursion will automatically provide the solution $x_{k;i}$ at this leaf.

5.2 Node

We have fully specified the behavior of the algorithm at a leaf. Now we can specify its behavior at a node. Assume that we are at node $(k; i)$. The the algorithm first proceeds by calling itself on its left child node $(k + 1; 2i - 1)$. However, there could be some pending subtractions on the right-hand side that needs to be passed down. We do so

$$f_{k+1;2i-1} = R_{k+1;2i-1} R_{k;i}.$$

Since $g_{k+1;2i} = 0$, we can ignore it.

When the algorithm returns we need to pass on the information for updating the right-hand side to its right child $(k + 1; 2i)$

$$f_{k+1;2i} = R_{k+1;2i} f_{k;i} + B_{k+1;2i,2i-1} g_{k+1;2i-1}.$$

Notice that there are two pieces of information now: first, is the piece $f_{k;i}$ that it inherited from its parent, and second $g_{k+1;2i-1}$ is the new piece computed by its left child.

When the algorithm returns, we are guaranteed that both the left and right nodes are leaves now. So we merge them to convert the current node $(k; i)$ into a leaf, having the HSS elements

$$\begin{aligned} \hat{D}_{k;i} &= \begin{pmatrix} \hat{D}_{k+1;2i-1} & \hat{U}_{k+1;2i-1} B_{k+1;2i-1,2i} \hat{V}_{k+1;2i}^H \\ \hat{U}_{k+1;2i} B_{k+1;2i,2i-1} \hat{V}_{k+1;2i-1}^H & \hat{D}_{k+1;2i} \end{pmatrix}, \\ \hat{U}_{k;i} &= \begin{pmatrix} \hat{U}_{k+1;2i-1} R_{k+1;2i-1} \\ \hat{U}_{k+1;2i} R_{k+1;2i} \end{pmatrix}, \\ \hat{V}_{k;i} &= \begin{pmatrix} \hat{V}_{k+1;2i-1} W_{k+1;2i-1} \\ \hat{V}_{k+1;2i} W_{k+1;2i} \end{pmatrix}. \end{aligned}$$

At this stage we are left with a system of equations that looks identical to the original system, except that it is guaranteed to have two fewer nodes at least. So, we can solve the system recursively. Once that has been done, the information in $\hat{x}_{k;i}$ must be passed along to the children. To that end we partition it

$$\hat{x}_{k;i} = \begin{pmatrix} \hat{x}_{k+1;2i-1} \\ \hat{x}_{k+1;2i} \end{pmatrix}.$$

Then $\hat{x}_{k+1;2i-1}$ is passed to the left child and $\hat{x}_{k+1;2i}$ is passed to the right-child, for which they both are waiting. When the algorithm terminates the whole solution x has been found.

5.3 Complexity

The complexity estimates for the algorithm are identical to the algorithm presented in [4]. In particular, since the cost of multiplying by U , L , V and their inverses is an order of magnitude lower than the cost of factorization, the estimates from [4] hold exactly in this case too. In particular if all the $U_{k;i}$ and $V_{k;i}$ have fewer than p columns then the cost of the algorithm is $46np^2$ flops plus some lower-order terms. Paper [4] also looks at the cost of the algorithm in other practically interesting cases: for example, when the matrix is obtained by discretizing two and three dimensional kernels.

5.4 Stability

The numerical stability considerations are identical to those of the algorithm presented in [4]. If the HSS representation is in proper form ($\|W_{k;i}\| \leq 1$ and $\|R_{k;i}\| \leq 1$), then the algorithm presented here is backward stable. The proof hinges on the fact that we only use unitary transforms and a single forward substitution. Experimental evidence is presented in table 1.

6 Numerical Experiments

We consider a family of matrices A of the form

$$A_{i,j} = \sqrt{|x_i - x_j|},$$

where for an $n \times n$ matrix A the points x_i are chosen to be the n zeros of the n -th Chebyshev polynomial $T_n(x) = \cos(n \cos^{-1} x)$. These points cluster quadratically at -1 and 1 . We partition these points by recursively dividing the interval $[-1, 1]$ into equal halves. We stop partitioning an interval as soon as the number of points in that interval decreases below a pre-set threshold p . This partitioning is then used as the HSS partitioning of A . The HSS structure is then computed to the tolerance $1.5\text{e-}08$ using the algorithm specified in [4]. We did a series of experiments with such matrices A ranging in size from 256 to 131072. In each case we report the values of p , the cpu run-time in seconds, and the approximate backward error,

$$\frac{\|A\hat{x} - b\|_1}{n\|\hat{x}\|_1 + \|b\|_1},$$

where n is an approximation of the 1-norm of A . We also report the ratio of the largest path-length from a leaf to the root to the shortest path-length from a leaf to the root (termed *skew* in the table), as a crude measure of the non-uniformity of the tree. The results can be found in table 1.

Table 1: Numerical experiments

n	p	skew	CPU time (seconds)	backward error	<code>dge1s</code> (seconds)
256	13	2	0.02	2.8e-17	0.06
512	14	1.8	0.05	3.0e-17	0.28
1024	15	1.83	0.1	2.0e-17	1.89
2048	16	1.86	0.19	1.2e-17	13.72
4096	17	1.87	0.44	1.3e-17	106.6
8192	18	1.89	0.81	1.8e-17	1429.33
16384	19	1.9	1.73	1.3e-17	
32768	20	2	3.05	5.7e-17	
65536	21	2	6.36	2.3e-18	
131072	22	2	12.67	5.7e-18	

The experiments were carried out on a dual 1GHz PowerPC G4 with 2MB L3 cache per processor, and 1.5GB RAM, using vendor supplied BLAS in double precision. Only a single CPU was used. For comparison we provide the timings for the LAPACK solver `dge1s` using the same BLAS on the same machine in table 1. In some cases there was insufficient memory to do the timings. Those entries are left blank.

As can be seen from the results, the algorithm is backward stable, and can be several orders of magnitude faster than standard Gaussian elimination. In fact the CPU run-time in seconds required by the algorithm scales almost perfectly linearly in the problem size.

7 Conclusion

In this paper we presented a new fast and backward stable solver for matrices that are in HSS form. The method uses an implicit *ULV* decomposition that accounts for its backward stability. The method is as efficient as possible, even when the underlying partition tree of the HSS representation is highly non-uniform. In a future paper we also intend to present an *explicit* fast *ULV* decomposition algorithm. The algorithm presented here can also be generalized to more complex and refined fast multi-pole method (FMM) representations of matrices [1]. This will also be presented in a future paper.

References

- [1] J. Carrier, L. Greengard, and V. Rokhlin, *A fast adaptive multipole algorithm for particle simulations*, SIAM J. Sci. Stat. Comput., 9 (1988), pp. 669–686.
- [2] S. Chandrasekaran, P. Dewilde, M. Gu, T. Pals, X. Sun, A.-J. van der Veen and D. White, “Fast Stable Solvers for Sequentially Semi-separable Linear Systems of Equations and Least Squares Problems,” technical report, Department of Mathematics, University of California, Berkeley, CA, 2003.
- [3] S. Chandrasekaran, P. Dewilde, M. Gu, T. Pals, X. Sun, A.-J. van der Veen, and D. White, *Some Fast Algorithms for Sequentially Semi-separable Representations*, submitted for publication, 2003.
- [4] S. Chandrasekaran, M. Gu, and T. Pals, *Fast and Stable Algorithms for Hierarchically Semi-separable Representations*, submitted for publication, 2004. .