TACC Technical Report TR-07-02

Sparse Direct Factorizations through Unassembled Hyper-Matrices

Paolo Bientinesi[*], Victor Eijkhout[†], Kyungjoo Kim[‡], Jason Kurtz[‡],
Robert van de Geijn[§]

November 3, 2009

[*]   Computer Science Department, RWTH Aachen, Germany
[†]   Texas Advanced Computer Center, The University of Texas at Austin, *corresponding author*
[‡]   Aerospace Engineering and Engineering Mechanics, The University of Texas at Austin
[§]   Computer Science Department and Institute for Computational Engineering and Sciences, The University of Texas at Austin

Abstract

We present a novel strategy for sparse direct factorizations that is geared towards the matrices that arise from hp-adaptive Finite Element Methods. In that context, a sequence of linear systems derived by successive local refinement of the problem domain needs to be solved. Thus, there is an opportunity for a factorization strategy that proceeds by updating (and possibly downdating) the factorization. Our scheme stores the matrix as unassembled element matrices, hierarchically ordered to mirror the refinement history of the domain. The factorization of such an 'unassembled hyper-matrix' proceeds in terms of element matrices, only assembling nodes when they need to be eliminated. The main benefits are efficiency from the fact that only updates to the factorization are made, high scalar efficiency since the factorization process uses dense matrices throughout, and a workflow that integrates naturally with the application. We present tests on 2D problems that bear out the large savings possible with hyper-matrix factorizations.

1   Introduction

Many scientific applications spend a large amount of time in the solution of linear systems, often performed by sparse direct solvers. We argue that traditional matrix storage schemes, whether dense or sparse, are a bottleneck, limiting the potential efficiency of the solvers. We propose a new data structure, the Unassembled Hyper-Matrix (UHM). This data structure preserves useful information that can be provided by the application, and that can make the solver, as well as various other operations on the matrix, more efficient. In particular, we will use this storage format to implement an efficient sparse direct solver for hp-adaptive[1] Finite Element Method (FEM) problems.

The improvement in efficiency will come from rethinking the conventional approach where sparse direct solvers are used as a black-box, where a linear system is passed as input and a solution is returned as output. Much progress has been made on making such a black-box procedure as efficient as possible. However, traditional solvers are intrinsically handicapped by ignoring domain information. Furthermore, what is not exploited by such solvers is the fact that once a solution for a given discretized problem has been computed, modifications to this existing discretization are made. This means that what should be the real measure of efficiency is how fast a solution of a somewhat modified (refined) problem can be computed given a factorization of the current problem (discretization). We argue that this formulation of the problem leads to dramatically different data structures and factorization approaches, on which the existing literature on sparse direct solvers has little bearing.

In this introduction we sketch the demands on a matrix storage scheme in a FEM application, and show how traditional linear algebra software insufficiently addresses these demands. In the rest of this paper we will then show how the UHM scheme overcomes these limitations. We limit ourselves to symmetric positive-definite (SPD) problems.

1.1   The workflow of advanced FEM solvers

Adaptive discretization techniques are recognized to be the key to the efficient and accurate solution of complicated FEM problems, for instance, problems with singularities around re-entrant corners. (We will give a brief overview of adaptive and in particular hp-adaptive FEM in Section 2.)

A typical hp-adaptive FE computation proceeds as follows.

1.   An initial discretization is generated to represent the given geometry and material data.
2.   The global stiffness matrix and load vector are computed, stored either in an assembled sparse format or as unassembled element contributions.

---

1. The designation 'hp' refers to the simultaneous refinement of the space discretization h, and the polynomial degree p.

3. The sparse linear system is solved via a standard solver package. Here the choice of package depends on whether the solution is computed via a direct or iterative solver. For sparse direct solution, widely used packages include MUMPS [1, 30], NASTRAN (various commercial versions), SuperLU [12, 13, 26], and UMFPACK [8, 37]. For iterative solution current favorites include PETSc [2, 3] and Trilinos [20].
4. Based on *a posteriori* error estimates, it is determined whether to break or merge elements (h-refinement or unrefinement) and/or whether to increase (or decrease) the order of approximation p. These decisions are made locally, i.e., on an element-by-element basis, though refinement of one element may induce surrounding elements to be refined too, in some circumstances.
5. Steps 2–4 are repeated until a stopping criterion is met.

1.2   Shortcomings of the existing approach

There are two essential limitations with the traditional approach to FEM solvers outlined above.

One obvious problem is in Step 3, where the solver has no way of knowing whether it was invoked before, and what the relation is between its input data in successive invocations. Since successive linear systems are clearly related, considerable opportunity for efficiency is left unexplored.

There is a further problem in that, by formulating the linear system as a matrix equation, much information about the application is lost. This information is then laboriously, and imperfectly, reconstructed by the graph partitioners used in sparse solver packages.

These shortcomings of the matrix-based interface are not purely academic. In Section 2.3 we will show anecdotal evidence that fairly simple manual preprocessing of the linear systems can significantly improve the efficiency of a standard direct solver. Clearly, certain knowledge of the linear system that is available to a human can only imperfectly be discovered by a black-box solver. Our improved data structure and solver preserve and exploit such knowledge.

1.2.1 Inflexibility in an application context

Current linear algebra software has little provision for the reality that often a sequence of related linear systems is to be solved. Even a slight change to the matrix forces a solver package to recomputate the factorization from scratch, with no information preserved.

However, in the setting of adaptive FEM solvers, the next linear system is often derived from the previous one by refining part of the physical domain, either in space, or in the order of the FEM basis functions. Traditional matrix storage is not flexible enough to accomodate

insertion of matrix rows easily. Instead, a whole new matrix needs to be allocated, with the old data copied over or even recomputed, at considerable overhead. Furthermore, solver packages cannot preserve parts of a factorization that are not affected by such a refinement. Our UHM storage scheme remedies both shortcomings.

While it can be argued that the use of a high-quality graph partitioner favors the current approach to successive substructuring for a single solution, storing the stiffness matrix as an UHM conformal to the hierarchy in the domain has the potential for greatly reducing the cost of subsequent solves with refined data. There is some memory overhead associated with Element-By-Element (EBE) codes: anecdotal evidence suggests 30% for matrix storage in a 2D case with low polynomial degree [6, 5], and possibly more with higher degrees, see table 1 in [32]. However, this is outweighed by advantages in performance and flexibility. Also, the overhead from EBE storage for the factorization is considerably less.

1.2.2 Loss of application information

The representation of a matrix as a two-dimensional array of numbers, whether stored densely or using a sparse storage format, represents a bottleneck between the application and the solver library. Relevant application knowledge is lost, such as geometry and other properties of the domain, various facts about the nature of the mesh, and any history of refinement that led to the current system of equations. Much of the development of sparse direct solvers goes into reconstructing, algebraically, this information.

1.3   Relation to existing factorizations

Our factorization scheme contains some novel elements, foremost the fact that we retain the refinement history of the Finite Element (FE) grid. Of course, there are various connections to the existing literature. In this section we highlight a few. (For a recent overview of the field of sparse direct factorizations, see the book by Davis [9].)

1.3.1 Substructuring

Techniques of bisection and recursive bisection have long been a successful strategy, although not the only one, for deriving direct solvers. Initial research on solvers on a regular domain showed considerable savings in storage for two-dimensional problems [16, 17]. These results have been extented to arbitrary finite element meshes [27, 28], including a proof that in the three-dimensional case no order improvement exists as in the 2D case: in 2D, the naïve space bound of $O(N^{3/2})$ can be improved to $O(N \log N)$; in 3D, no reduction of the naïve $O(N^{5/3})$ bound is known.

More recently, spectral bisection techniques have been explored as a way of deriving multiple partitioning of a set of variables [15, 18, 19, 22, 34]. Another direction in graph partitioning is

that of partitioning methods based on space-filling curves [31, 33]. These methods have been used primarily for partitioning elements in work related to iterative solvers. Again, such techniques are based on algebraic properties of the matrix graph, and can only imperfectly reconstruct any division that is natural to the problem.

1.3.2 Supernodes

With the realization that Level 3 Basic Linear Algebra Subprograms (BLAS [24, 7]) operations are the path to high performance in linear algebra codes (see for instance [14, 4]), researchers of sparse direct solvers have devoted considerable effort to finding 'supernodes': blocks of rows or columns that have similar sparsity patterns, and can thus be tackled with dense block algorithms [36, 25] when combined. However, this block structure derives from the elements in a Finite Element mesh, so we conclude that the linear algebra software aims at reconstructing information that was present in the application and was lost in the traditional solver interface.

It is clear that a matrix representation that preserves information about the operator and the discretization has a potential advantage over traditional storage formats.

1.3.3 Hierarchical methods

The idea of using a tree structure in the factorization of a matrix has occurred to several authors and in several contexts. However, this is typically done in an *a posteriori* fashion, where the matrix or the domain is recursively partitioned, giving both parallelism and favourable fill-in properties. We mention the nested dissection method of George and Liu [17], which recursively partitions a domain to reduce fill-in, and the hypermatrix method of Herrero and Navarro [21], which partitions a matrix for increased performance. Our method differs from these in that we do not take a completed matrix as our starting point, but rather derive the tree structure from the originating process of the matrix.

1.4   Outline

In the remainder of this paper we briefly introduce the hp-adaptive FEM, which gives us our application context. We then present the UHM factorization and outline certain practical issues and future research directions.

2   Finite Element background

In this section we give a brief introduction to the hp-adaptive FEM, its practical importance, and the demands it puts on solvers. For a full treatment of the hp-adaptive FEM, see the book by Demkowicz [11].

2.1 hp-Adaptive FEM

The FEM is a method for discretizing and obtaining an approximate solution to partial differential equations (PDE) arising from a broad spectrum of physical and engineering applications. In FEMs, the approximate solution is represented as a piecewise polynomial function. Of obvious importance is the estimation and reduction of the corresponding discretization error.

The most classical method for reducing the error in a discretization is the h-method, where elements are broken, either uniformly or in an adaptive way, to decrease the element size h. The polynomial order of approximation p is uniform and fixed (usually quite low: $p = 2, 3$ at most). In the p-method, convergence is achieved by increasing the order of approximation p, either uniformly or in an adaptive way, while the element size h is fixed. hp-methods combine these two approaches, allowing for local combinations of h-refinement and p-enrichment [35].

In many interesting physical applications, adaptive methods are preferred because the solution is smooth throughout most of the computational domain. Singularities arise only at a few localized features such as re-entrant (non-convex) edges and vertices, or material interfaces. For such problems, the separate h or p-methods only converge algebraically with respect to the total number of degrees of freedom N (whereas the p-method converges exponentially for a globally smooth solution). The combined hp-method essentially isolates the effect of singularities through local h-refinement and uses p-enrichment where the solution is smooth. With an appropriate combination, the hp-method can deliver exponential convergence *even* for problems with singularities.

2.2   Use of sparse direct solvers in hp-adaptive FEM

While hp-methods are capable of delivering a given accuracy with a minimal number of degrees of freedom, they suffer a considerable setback in terms of the complexity of the implementation. Over the past 20 years research has mainly focused, not on the efficiency of the implementations, but on controlling this complexity. A significant advance in this direction was the introduction of a node-based, hierarchical data structure in the code 3Dhp90 [10]. In this data structure, successive h-refinements are supported by growing so-called "refinement trees" out of a given initial mesh. When a (parent) node is broken, connectivities from parent to child (and vice versa) are maintained. The current mesh consists of the leaves of this data structure and element-to-node connectivities are reconstructed using the refinement history. Traditional sparse direct solvers make no explicit use of this hierarchy.

2.3   Evidence for our case

In tests performed in [23], MUMPS was used as a solver, both by itself, and preceded by a stage of manually executed Static Condensation, which essentially corresponds to the manual

elimination of supernodes. In Figure 1 we compare the time and space complexity of a code exclusively using MUMPS to one that explicitly performs static elimination, and only uses MUMPS for the quotient graph that is so obtained.

The graphs show that using static condensation leads to a 50% reduction in memory and 20–40% reduction in runtime, demonstrating the limitations of a general purpose sparse solver, and in particular the limitations of METIS in finding the optimal ordering. In our UHM library, this static condensation step would be executed in the process of eliminating interior variables. We note that these are automatically identified, without the need for any graph analysis.

2.4   A dual grid hierarchy

Our approach consists in determining an optimal refinement strategy for a given *coarse grid* by examining the solution on a corresponding *fine grid* obtained by a global *hp*-refinement. From the fine grid we then conclude the proper locations for doing local refinement on the coarse grid. This process is illustrated in Figure 2.

At first, this seems to imply multiple applications of global refinement, which would invalidate our approach of local updates. However, a more detailed consideration of the grid hierarchy shows that we are dealing with two sequences, one of coarse grids and one of fine grids, and in each sequence the grids are derived by local refinement of their predecessors. This process is illustrated in a simplified manner in Figure 3.

3   The factorization scheme

Above, we have sketched how the *hp*-adaptive FEM gives a refinement tree of elements, where the actual elements making up the FE matrix are the leaves of the tree. In most factorizations schemes, the key to space and time efficiency consists in finding the proper elimination ordering. The underlying observation for our method is that a (partial) ordering, induced by the refinement tree, is directly available.

In its most stark statement, the whole factorization is based on the recursion where *children of one parent are to be eliminated, forming a Schur complement matrix on the parent element.*

Intuitively, the reader will see that refinement leads to new subtrees, and the factorization only needs to be updated by the factorization of that subtree and its parents. We will argue this point in detail below. We will briefly touch on the matter of the space and time complexity in Section 6.1.

We will now explain our factorization by showing in detail the mechanism in a one-dimensional example. We will then address how the mechanism extends to higher dimensions.
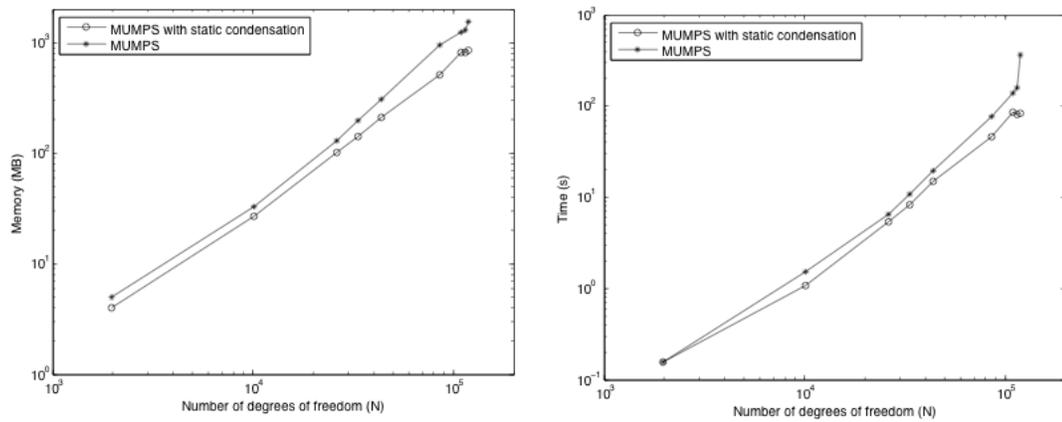
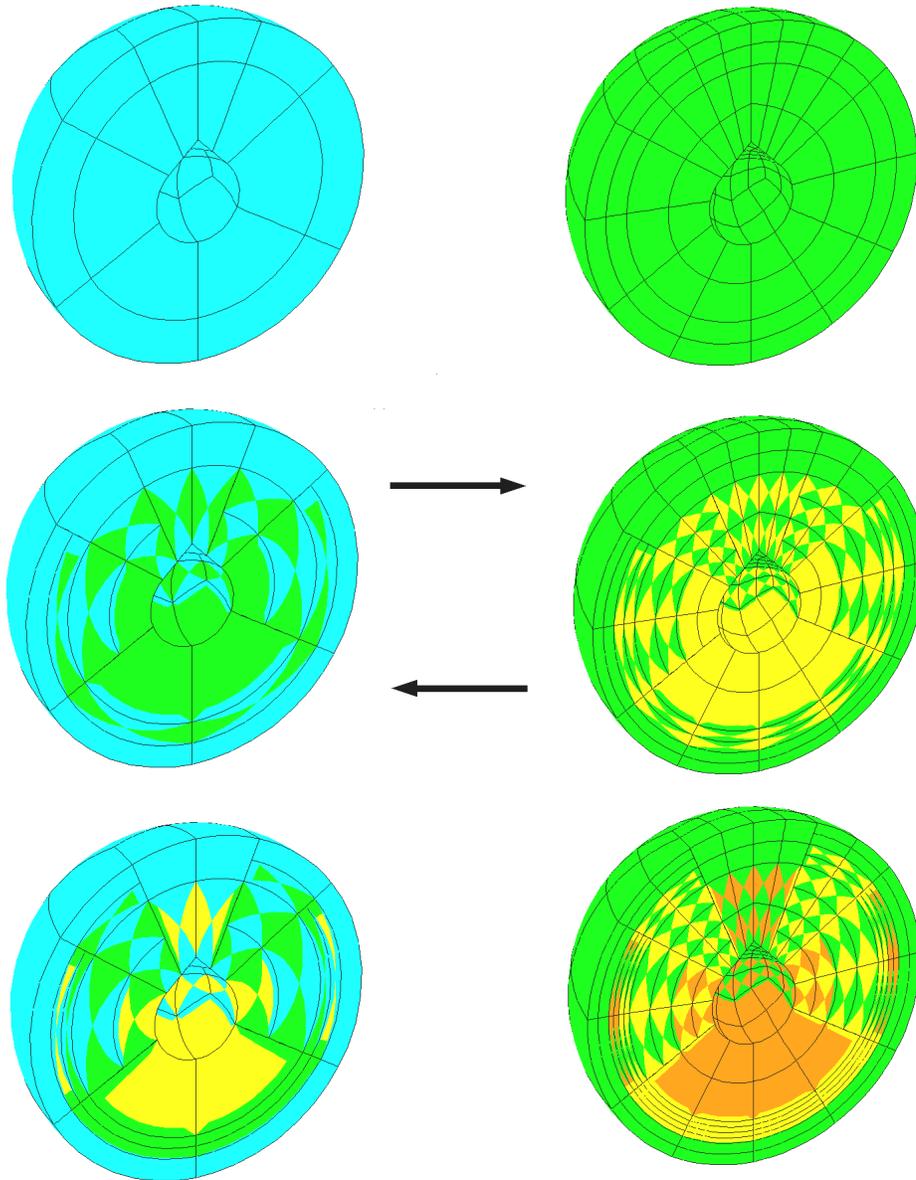Figure 1: Memory and time savings from Static Condensation before the application of the MUMPS solver.
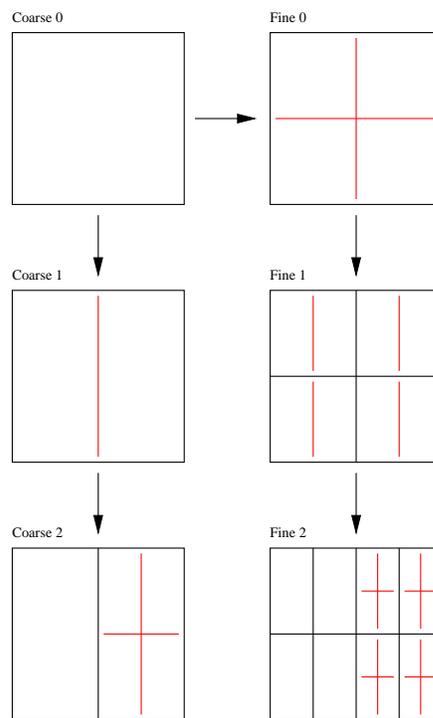
Figure 2: Sequence of coarse and fine grids.

Figure 3: Derivation of coarse and fine grids through local refinement.

3.1   One-dimensional example

We can convey the essence of our factorization scheme by considering a one-dimensional example. This will give a slightly simplified situation from the general two or three-dimensional case: we will address the differences in the next section.

Consider a domain that consists of two elements obtained by refining a single top level element. Each element has three (groups of) unknowns: vertex functions on its left and right vertex, and edge functions on its interior. For the whole domain this gives five (blocks of) unknowns, since one vertex is shared by the two elements.
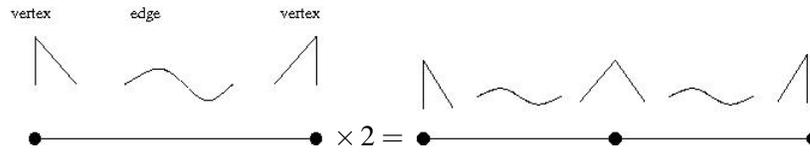
Algebraically this manifests itself as the FE matrix being a sum of two 3×3 element matrices:

$$
\begin{pmatrix}
a_{11} & a_{12} & a_{13} & & \\
a_{21} & a_{22} & a_{23} & & \\
\hline
a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\
& & a_{43} & a_{44} & a_{45} \\
& & a_{53} & a_{54} & a_{55}
\end{pmatrix}
=
\begin{pmatrix}
a_{11} & a_{12} & a_{13} \\
a_{21} & a_{22} & a_{23} \\
a_{31} & a_{32} & a_{33}^{(\ell)}
\end{pmatrix}
\oplus
\begin{pmatrix}
a_{33}^{(r)} & a_{34} & a_{35} \\
a_{43} & a_{44} & a_{45} \\
a_{53} & a_{54} & a_{55}
\end{pmatrix}
$$

where the (ℓ) and (r) superscripts indicate contributions from the left and right element respectively.

We now observe that eliminating the interiors of the elements can be done within each element independently of the other element:

$$
\begin{pmatrix}
\tilde{a}_{11} & \tilde{a}_{13} & \\
\tilde{a}_{31} & \tilde{a}_{33} & \tilde{a}_{35} \\
& \tilde{a}_{53} & \tilde{a}_{55}
\end{pmatrix}
=
\begin{pmatrix}
a_{11} & a_{13} \\
a_{31} & a_{33}^{(\ell)}
\end{pmatrix}
-
\begin{pmatrix}
a_{12} \\
a_{32}
\end{pmatrix}
a_{22}^{-1}
\begin{pmatrix}
a_{21} & a_{23}
\end{pmatrix}
$$

$$
\oplus
\begin{pmatrix}
a_{33}^{(r)} & a_{35} \\
a_{53} & a_{55}
\end{pmatrix}
-
\begin{pmatrix}
a_{34} \\
a_{54}
\end{pmatrix}
a_{44}^{-1}
\begin{pmatrix}
a_{43} & a_{45}
\end{pmatrix}
$$

After eliminating the interiors this way, we wind up with a 3×3 (block) matrix, that is, a matrix with the same structure as a matrix on the original unrefined top element. We now recursively eliminate its interior, giving again a 2×2 matrix; if this element itself was obtained through refinement, we can now continue the recursive factorization.

If we make this argument recursive, it becomes clear that updates to one element (in our case further refinement) do not influence the factorization of the other element. In particular, the

factorization of whole subtrees can be preserved if other subtrees (that are not contained in it) are altered. When an updated factorization of one element is combined with the preserved factorization of its sibling (or siblings, in the more general case), their non-interior nodes are combined, and have to be re-eliminated. We note that in hp-adaptive FE the interior typically has the highest polynomial degree, making the re-elimination of vertices (and edges or faces in the general case) relatively inexpensive.

3.2   Extension to higher dimensions

The above discussion showed how in one space dimension the hyper-matrix ordering leads to a particularly simple factorization scheme. In particular, if an element on level ℓ gets divided into elements on level $\ell + 1$, the factorization of one sibling is unaffected by changes (such as further refinement) in other siblings.

In higher dimensions certain complications arise which we will briefly sketch in this section. In spite of this, the basic conclusion still holds that large amounts of work can be saved over a full factorization of the updated matrix.

To explain the problem in higher dimensions, consider two steps of uniform refinement:
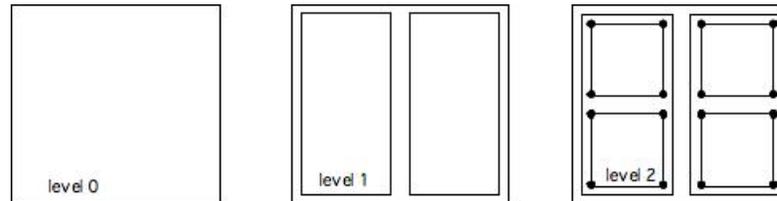
Applying the factorization scheme of the one-dimensional example, we would now start by eliminating all level 2 nodes that are not on level 1. However, this is not as simple in higher dimensions as in 1D. In one space dimension, nodes that need to be assembled (such as the middle node in the example) have a common parent one level up. As seen in Figure 4, in higher dimensional problems certain nodes (the middle node in this example) will have a parent two levels up, and it can be seen that nodes can have a 'least common parent' an arbitrary number of levels up. This means that their assembly on the current level would violate the recursive formulation of the factorization, so instead we carry some unassembled nodes up to the higher level. As a result, after one elimination step, at level 1 we end up with slightly more nodes than we would have had without the refinement step.

In eliminating the level 1 refinement, we now assemble and eliminate the level 1 nodes that do not exist on level 0, but also the partially assembled level 2 nodes that were carried upward.

By contrast, in one space dimension, we could have eliminated the level 2 nodes entirely by operations on level 1, leaving us with an element matrix on level 1 with exactly the same structure as if the refinement to level 2 had never happened.
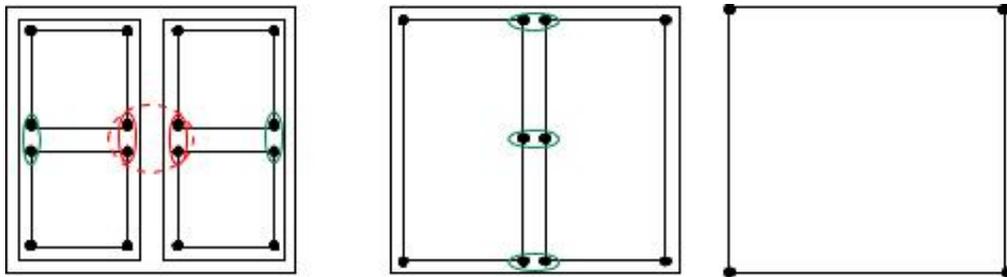
Figure 4: Node assembly process in two space dimensions

4 Library design

The UHM library offers two Application Programmer Interface (API)s: one for the end user, and one for the FE library writer.

- The FE library writer would use the UHM library to define a collection of elements and refinement schemes.
- The end-user would instantiate elements, indicating their connectivity, and would specify which elements get refined using which refinement scheme.

In both cases, this interface gives its users a simple way to handle very complicated data structures. (Note that at present we have only a prototype library implemented; any details below are solely to illustrate the design principles.)

4.1 APIs

We first show the FE library API, which is used to define refinement schemes. To this purpose, the library writer has lower level tools that relate the constituent vertices, edges, etc. of a parent element to those of its children under refinement.

For instance, to indicate that an edge is broken into edges of the children with a joining node, we define dividers:

```
ierr = CreateDivider(&edge);
ierr = DividerSetName("break_edge", edge);
ierr = DividerSetScheme("edge_0,vertex_1,edge_0", edge);
```

Such dividers are then taken together to form a refinement scheme:

```
ierr = CreateRefinement(&hor);
ierr = RefinementSetName("horizontal", hor);
ierr = RefinementAddBreaknode(6, edge, hor);
ierr = RefinementAddBreaknode(8, edge, hor);
ierr = RefinementAddBreaknode(9, interior_oneway, hor);
```

The end-user only has access to nodes and to refinement schemes to apply to them:

```
ierr = ElementGetChildByChildnum(top, 1, &elt);
ierr = ElementRefineElement(hor, elt);
```

4.2 Initial mesh

Our code progresses fully recursively if the domain is derived through the refinement of a single element. In practice this will not be the case: there will be an initial mesh that fits the geometry of the object being modeled. We handle this initial mesh by describing it as the result of refinement of a single virtual top element. This refinement will be somewhat laborious to specify, but the instructions can be generated automatically from the FE specification of the initial mesh.

4.3   Implementation

The main design decision in the UHM library concerns where the mesh connectivity and relations (both hierarchical and horizontal) between nodes and elements are stored. It would, for instance, be possible to derive and store the information regarding node connectivity from the refinement scheme. This has a few disadvantages.

Firstly, it leads to a complicated solver code, with overlapping functionality with the FE code. Secondly, it prevents the solver from working with FE codes that use different refinement scheme. Third is that UHM should provide the objects which correspond to all kinds of existing FE's and their refinement schemes. Furthermore, a refinement scheme is a user defined object, and no library can support all the different needs from the FE world.

For this reason, in its current incarnation, the UHM library maintains only the element hierarchy and a global node numbering. This is sufficient information to determine which nodes can be eliminated on a level, and which need to be carried up to higher levels. The global numbering also allows finding neighbouring elements in order to merge nodes, without having this connectivity information stored explicitly.

5   Numerical Experiments

We have implemented an UHM solver, and interfaced it to the *hp*-adaptive FEM code *3Dhp90* written by Dr Demkowicz's group [11]. In this section we present comparisons against MUMPS 4.8.4 [30] running in sequential mode. METIS 4.0 [29] was used for the internal ordering of the MUMPS solver.

5.1   Laplace problem in 2D L-shape domain

Our first example is the Laplace equation on the L-shaped domain [11]. A manufactured solution, which has a singularity at the reentrant corner, was used in order to impose Neumann boundary condition. Performance was measured on the grids generated by two different strategies. One is a regular mesh created by uniform refinements. The other is an irregular mesh adaptively refined from the initial mesh.

5.1.1 Performance comparison on regular mesh generated by uniform refinements

Figure 6 shows the solution of a regularly refined grid on an L-shaped domain. Since all finite elements were created by subdividing of parent element into 4 children elements, the elimination ordering of the UHM factorization is very similar to one of the nested dissection method.

$$
\begin{aligned}
-\Delta u &= 0 & \text{in } \Omega \\
u &= 0 & \text{on } \Gamma_D \\
\frac{\partial u}{\partial n} &= \frac{\partial u^{exact}}{\partial n} & \text{on } \Gamma_N \\
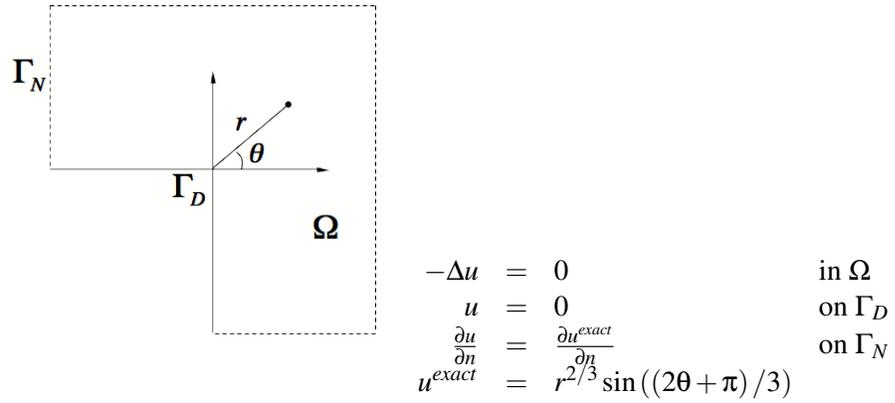u^{exact} &= r^{2/3} \sin\left((2\theta + \pi)/3\right)
\end{aligned}
$$

Figure 5: The Laplace equation on the L-shaped domain

The graphs in Figure 7 compare the solution time (measured by wall clock time) and memory usage relative to MUMPS, as a function of the polynomial order p. We observe that UHM performance improves as the order p increases, since it is fully based on dense matrix operations. Also, UHM has approximately half the memory demands of MUMPS. This can probably be attributed to the fact that, storing dense element matrices, we need far less indexing stored than MUMPS.

5.1.2 Performance comparison on irregular mesh generated by h-adaptive refinements

In the next example, it is assumed that problem domain is initially discretized with a moderate number of finite elements. Since the problem has a singularity at the reentrant corner, the hp-adaptive algorithm will refine the finite elements around the corner. (For this example, we only consider h-refinement, and use a constant p-order.) Now a large part of the UHM factorization can be reused when a next refinement level is introduced.

The graphs in Figure 9 show how the updating process of the UHM factorization leads to substantial savings in run time. After UHM and MUMPS solve the problem on a initial mesh, UHM reuses parts of the factorization on the previous grid when it solves new system obtained by the local refinements. This feature of UHM save about 75 percents in runtime, while MUMPS solves the new system from scratch. There is some memory overhead in UHM from storing all Schur complements at the intermediate steps. If necessary, this overhead could be removed by storing the Schur complements on the external memory (out-of-core).

If we compare the slopes of the lines in Figure 9, we see that UHM at some point may become more expensive than MUMPS. The reason for this is that the elimination ordering is statically determined by the refinement history, which is the feature that allows us to reuse parts of the factorization. However, this elimination ordering may not be optimal, and in fact in the case of refinement to a re-entrant corner it is not: the UHM ordering gives a similar complexity to nested dissection, but an ordering based on level sets (corresponding to refinement levels) has a lower complexity.

As a consequence, the UHM ordering, in certain cases, produces more fill-in than MUMPS in a mesh obtained through adaptive refinements. The performance of an updating factorization is then dependent on the problem, according to how much of the factorization is being reused, and how much local refinements on finite elements affect the complexity of UHM. We are researching a strategy for maintaining at least parity with MUMPS, even in these cases.

6   Discussion

Above, we have outlined the basic ideas of our novel factorization scheme for linear systems arising in hp-adaptive FEM problems. In that context, our scheme offers the following

advantages over existing factorizations.

Workflow integration. We use the structure of the application directly in determining the flow of the factorization. This should lead to a more natural integration of the solver in the physics code; see Section 6.4. Moreover, this integration has great advantages for the solver itself.

Elimination ordering. Unlike in classical factorizations, our scheme does not require us to find an elimination ordering, since we inherit it from the refinement history. This ordering leads to great savings in the adaptive context, and there are indications that even on a single system it can be efficient; see Section 6.1 below. In other circumstances (such as refinement around a reentrant corner) it is not optimal, and we are devising a general graph reordering mechanism both to achieve competitiveness in the single-system case, and to maintain our savings in the adaptive case.

Cliques and superblocks. Traditional factorizations expend considerable energy in identifying cliques in the matrix graph or equivalently finding superblocks in the matrix. We obtain this information for free: the superblocks derive from the element matrices which are dense. Thus, high scalar efficiency of our scheme will be guaranteed.

Integration in an adaptive context. As we have indicated above, our method will lead to reuse of large parts of the factorization in the context of adaptively constructed domains. This is yet another outcome of our use of the refinement history as elimination ordering.

In the remainder of this section we will sketch a number of practical issues that will be the topic of future research.

6.1 Complexity

Our factorization, especially when applied to the uniform refinement of a Cartesian domain, has certain elements in common with nested dissection [17, 27].

Consider a uniform refinement of $\Omega = (0,1)^2$ by ℓ levels, that is, there will be $n = 2^\ell$ points per side, giving a matrix of size $N = n^2$. We observe that a node can only be eliminated if it can be fully assembled on the current level. This means that nodes that are part of an edge that was introduce at an earlier level, have to be carried up to that level before they can be eliminated. This means that at each level, the element edges carry an 'imprint' of all lower levels. In particular, at level $\ell - k$ (where $k \le \ell$) we have $2^{\ell-k}$ grid lines with 2^ℓ nodes each. At the highest level $k = \ell$ this leaves us with a total of 2^ℓ nodes, which are most likely fully connected. In summary, as in the case of nested dissection, we are left with a dense matrix of size $\sqrt{N}$.

In the case of irregular refinement, we wind up with a system of the size of the number of nodes introduced along it.

6.2   Pivoting

Our discussion so far has exclusively dealt with a factorization without pivoting, which makes it suitable only for the symmetric positive definite case. We will research and implement an extension, first to the symmetric indefinite case, which can be handled with symmetric permutations, and later to the general case, where partial pivoting by rows is necessary.

While pivoting adds considerable complexity to the data structures and algorithms, we note that the need for pivoting does not invalidate our basic approach. The reason for this is that because of the tree structuring, all connections of an edge variable are contained within a leaf element, and the connections of a vertex variable that is shared between elements are contained in the subtree of the least common ancestor. Thus, pivoting is a local operation in a subtree, and we preserve the essential cleanliness of our model, and all advantages such as easy extension of a matrix, or preserving parts of the factorization under refinement.

6.3   Load balancing

Achieving a well-balanced parallel application is difficult in an hp-adaptive context, since local refinement by definition affects any balance of the work load. For this reason we will initially only target shared memory parallelism.

However, we note that our tree structured algorithm makes it easy to redistribute workloads: if we move a subtree to a different processor, this will not induce new communications during factorization of the subtree. Communication is to be performed only when the Schur complement of the subtree is combined with those of its sibling subtrees.

6.4   Application integration

Our matrix storage scheme allows for tighter integration of the linear algebra and the physics parts of the application. Since we never need a global ordering of the variables, matrix formation does not need to be postponed until the entire domain has been discretized, nor does it have to be redone after changes to the domain. Since we only use local parent-child and sibling relations between elements, an element matrix becomes part of the matrix data structure immediately upon formation.

The precise API that our library will offer to applications is still under development.

7   Conclusion

In this paper we have presented a direct factorization scheme that is aimed at the kind of problems that arise in hp-adaptive finite elements. The basic idea is the use of the refinement

history for deriving the factorization ordering. The matrix is thus represented as a tree structure, and element matrices, rather than fully formed matrix elements, are stored.

The immediate consequence of these decisions is that after refinement (or de-refinement) of the domain, large parts of the factorization can be reused, making our scheme preferable over the traditional workflow where after each change to the matrix a full factorization needs to be performed. However, even when applied to a single linear system, our scheme is of low space and time complexity. This will be analyzed in detail in a followup paper.

References

[1] P.R. Amestoy, I.S. Duff, J.-Y. L'Excellent, and J. Koster. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM J. Matrix Anal.*, 23:15–41, 2001. also ENSEEIHT-IRIT Technical Report RT/APO/99/2.

[2] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhauser Press, 1997.

[3] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. PETSc home page. http://www.mcs.anl.gov/petsc, 1999.

[4] E. Barragy and R. van de Geijn. Blas performance for selected segments of a high *p* EBE finite element code. *Inter. J. on Nume. Meth. in Eng.*, 38:1327–1340, 1995.

[5] William Barth, Graham F. Carey, Benjamin Kirk, and Robert McLay. Parallel distributed solution of viscous flow with heat transfer on workstation clusters. In *High Performance Computing '00 Proceedings*, Washington, D.C., April 2000.

[6] William L. Barth. *Simulation of Non-Newtonian Fluids on Workstation Clusters.* PhD thesis, The University of Texas at Austin, May 2004.

[7] http://www.netlib.org/blas.

[8] T. A. Davis and I. S. Duff. A combined unifrontal/multifrontal method for unsymmetric sparse matrices. *ACM Trans. Math. Software*, 25:1–19, 1999.

[9] Timothy A. Davis. *Direct Methods for Sparse Linear Systems.* SIAM, Philadelphi, PA, 2006.

[10] L. Demkowicz, D. Pardo, and W. Rachowicz. 3D hp-adaptive finite element package (3dhp90). version 2.0. the ultimate (?) data structure for three-dimensional, anisotropic *hp* refinements. Technical Report TICAM Report 02-24, 2002.

[11] Leszek Demkowicz. *Computing with hp-Adaptive Finite Elements, vol 1, One and Two Dimensional Elliptic and Maxwell Problems.* Chapman & Hall/CRC, 2007.

[12] James W. Demmel, Stanley C. Eisenstat, John R. Gilbert, Xiaoye S. Li, and Joseph W. H. Liu. A supernodal approach to sparse partial pivoting. *SIAM J. Matrix Analysis and Applications*, 20(3):720–755, 1999.

[13] James W. Demmel, John R. Gilbert, and Xiaoye S. Li. An asynchronous parallel supernodal algorithm for sparse gaussian elimination. *SIAM J. Matrix Analysis and Applications*, 20(4):915–952, 1999.

[14] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, March 1990.

[15] Miroslav Fiedler. Algebraic connectivity of graphs. *Czechoslovakian Mathematics Journal*, 23:298–305, 1973.

[16] A. George. *Computer Implementation of the Finite Element Method.* PhD thesis, 1971.

[17] Alan George and Joseph H-W. Liu. *Computer Solution of Large Sparse Positive Definite Systems.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1981.

[18] Bruce Hendrickson and Robert Leland. A multilevel algorithm for partitioning graphs. In *Proc. Supercomputing '95.* ACM.

[19] Bruce Hendrickson and Robert Leland. An improved spectral graph partitioning algorithm for mapping parallel computations. *SIAM J. Sci. Comput.*, 16:452–469, 1995.

[20] Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, Alan Williams, and Kendall S. Stanley. An overview of the trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423, 2005.

[21] José R. Herrero and Juan J. Navarro. Sparse hypermatrix cholesky: Customization for high performance. *IAENG International Journal of Applied Mathematics*, 36:1:6–12, 2007.

[22] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. Technical Report TR 95-035, Department of Computer Science, University of Minnesota, 1995. A short version appears in Intl. Conf. on Parallel Processing 1995.

[23] Jason Kurtz. *Fully Automatic hp-Adaptivity for Acoustic and Electromagnetic Scattering in Three Dimensions.* PhD thesis, Institute for Computational Engineering and Sciences, The University of Texas at Austin, 2006.

[24] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for FORTRAN usage. *Transactions on Mathematical Software*, 5:308–323, 1979.

[25] Xiaoye S. Li. *Sparse Gaussian Eliminiation on High Performance Computers.* PhD thesis, 1996.

[26] Xiaoye S. Li and James W. Demmel. SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Trans. Mathematical Software*, 29(2):110–140, June 2003.

[27] Richard J. Lipton, Donald J. Rose, and Robert Endre Tarjan. Generalized nested dissection. *SIAM J. Numer. Anal.*, 16:346–358, 1979.

[28] Richard J. Lipton and Robert Endre Tarjan. A separator theorem for planar graphs. *SIAM J. Appl. Math.*, 36:177–189, 1979.

[29] http://glaros.dtc.umn.edu/gkhome/metis/metis/overview.

[30] http://graal.ens-lyon.fr/MUMPS/.

[31] J. Oden and A. Patra. A parallel adaptive strategy for hp finite element computations, 1994.

[32] D. Pardo and L. Demkowicz. Integration of hp-adaptivity and a two grid solver for elliptic problems. *Computer Methods in Applied Mechanics and Engineering*, 195:674–710, 2006.

[33] Abani K. Patra, Jingping Long, and Andras Laszloffy. Efficient parallel adaptive finite element methods using self-scheduling data and computations. In *HiPC*, pages 359–363, 1999.

[34] Alex Pothen, Horst D. Simon, and Kang-Pu Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM Journal of Matrix Analysis and Applications*, 11(3):430–452, July 1990.

[35] Ch. Schwab. *p- and hp-Finite Element Methods: Theory and Applications in Solid and Fluid Mechanics*. Oxford University Press, 1998.

[36] http://www.nersc.gov/~xiaoye/SuperLU/.

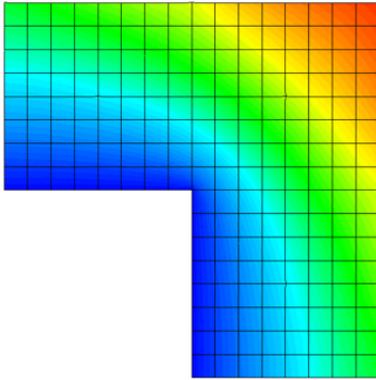[37] http://www.cise.ufl.edu/research/sparse/umfpack/.

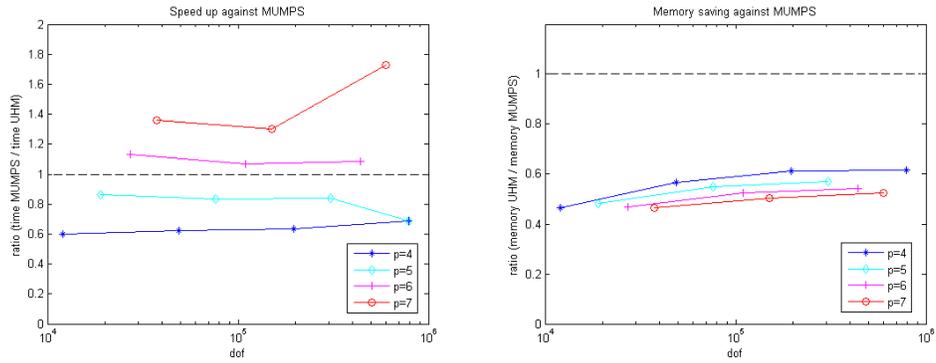Figure 6: Solution on a regularly refined L-shaped domain

Figure 7: Speed up and memory savings over MUMPS on the regularly refined L-shaped domain
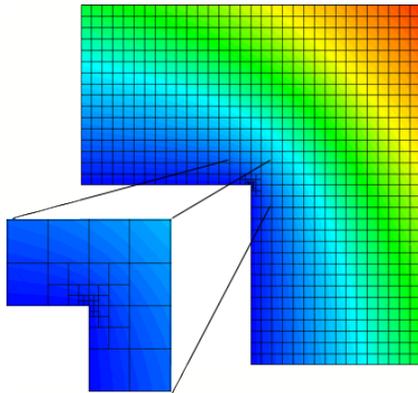

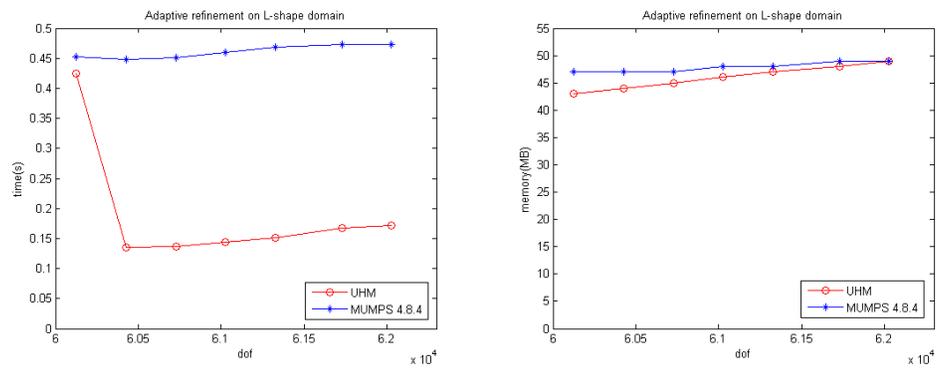Figure 8: An L-shaped domain with adaptive refinement

Figure 9: Time and memory measurements on an adaptively refined L-shaped domain