A Flexible Datastructure for Parallel Refinement in 3D*

Linda Stals[1]

Department of Computer Science, Old Dominion University, Norfolk, VA 23529-0162, USA and ICASE, NASA Langley Res. Ctr, Hampton, VA 23681-2199, USA.
stals@cs.odu.edu

Abstract. We describe a node-edge data structure which can be used to store parallel finite element grids. This is a very flexible data structure and allows us to refine the grid, build the system of equations, solve the system of equations and balance the load, all in parallel. The elements, such as the triangles, are not explicitly stored in the data structure. Consequently, when we recently extended the code to handle 3D domains we were able to use most of the framework developed for 2D domains, with the exception of the refinement routines. These refinement routines are also described in the paper.

1 Introduction

A very powerful and efficient method for solving elliptic partial differential equations is the multigrid method. We have been working on a parallel multigrid program which uses adaptive finite element grids.

Finite element grids are often stored in triangle based data structures. However, the elements do not have to be triangles, they can be for example quadralaterals, or tetrahedrons. With this argument in mind, we decided to use a node-edge data structure similar to the one described by Rüde [7]. This is a very flexible data structure and allows us to refine the grid, build the system of equations, solve the system of equations and balance the load, all in parallel.

As part of the multigrid algorithm, the program must be able to pass information between different grid levels. The extra degree of data dependencies defined by these inter-grid connections greatly increases the parallel complexity, as compared to a single grid scheme. However, by using the node-edge data structure these dependencies can be handled in a clean way.

The program has been used to solve a variety of non-trivial problems in two-dimensions (2D), including the modeling of plasma source ion implantation [6, 10] and radiation transport [12].

We recently extended the code to include three-dimensional (3D) domains and found that most of the structure which was built for 2D domains could be directly used in the 3D case. The modules which balance the load, handle the

* Supported by the U.S. Department of Energy under the ASCI ASAP program (subcontract B347882 from the Lawrence Livermore National Laboratory)

communication patterns and solve the (linear) system of equations did not need to be changed at all. The modules which build the system of equations required some minor modifications. Most of the work went into the refinement routine which was unavoidable since the algorithms used in 2D refinement are markedly different to those used in 3D refinement.

In summary, when extending the code to include 3D domains the actual node-edge data structure which stores the grids was not changed at all and most of the modules built upon this data structure required only minor modifications, if any. Consequently we feel that these experiences fully validate the use of such a flexible data structure and that is the theme which we would like to explore in this paper.

2 Multigrid Data Structure

For a description of the multigrid algorithm we refeer the reader to one of numerous reference on this subject, such as [3, 5]. To clarify the discussion below we need to introduce some notation. One of the fundamental requirements of the multigrid algorithm is a nested sequence of grids which we shall represent by $\mathcal{M}^1 \subset \mathcal{M}^2 \subset ... \subset \mathcal{M}^n$. Furthermore, let I_{m-1}^m be the interpolation matrix which moves information from the coarse grid $\mathcal{M}^{m-1}$ to the fine grid $\mathcal{M}^m$. The restriction matrix, I_m^{m-1}, is defined to be the transpose of the interpolation matrix and moves the information from the fine grid $\mathcal{M}^m$ to the coarse grid $\mathcal{M}^{m-1}$. We use linear interpolation and focus on elliptic partial differential equations. The discretisation method is the finite element method.

Given a grid $\mathcal{M}^m$ we may extract its geometrical, topological and algebraic attributes. The geometrical and topological attributes are simply the set of *nodes*, $\mathcal{N}^m$, and *edges*, $\mathcal{E}^m$. In our implementation the algebraic information is the set of *connections*, $\mathcal{C}^m$, where

$$\mathcal{C}^m = \mathcal{C}_A^m \cup \mathcal{C}_I^m \cup \mathcal{C}_R^m,$$

for $1 < m < n$ and

$$\mathcal{C}^1 = \mathcal{C}_A^1 \cup \mathcal{C}_I^1,$$
$$\mathcal{C}^n = \mathcal{C}_A^n \cup \mathcal{C}_R^n.$$

The set (or graph) $\mathcal{C}_A^m$ holds the stiffness matrix, while $\mathcal{C}_I^m$ and $\mathcal{C}_R^m$ hold the interpolation and restriction matrices respectively.

In summary, the grid $\mathcal{M}^m$ is given by $\mathcal{M}^m = \{\mathcal{N}^m, \mathcal{E}^m, \mathcal{C}^m\}$.

This node-edge data structure is similar to the one described by Rüde [7] and is presented in more detail in [8], [9] and [11].

The program is written in C^{++}. The set of nodes, edges and connections easily fall into three different classes which we call the `NodeTable`, `EdgeTable` and `ConnectTable` class. The edges and connections are stored as directed edges and directed connections. This makes the parallel implementation easier and the directed connections allows the storage of non-symmetric matrices while directed edges are needed for 3D refinement.

Notice that the node-edge data structure does not **explicitly** contain any information about the elements in the grid. We do however need to find the elements when, say, calculating the stiffness matrix. To find, for example, the tetrahedrons in the grid, define the set of *neighbour endpoints*, $\mathcal{B}$, for a Node N_i to be,

$$\mathcal{B}(\mathcal{E}, N_i) = \{N_j : (N_i, N_j) \in \mathcal{E}\},$$

then apply the following algorithm:

Algorithm 1
$\mathcal{T} \leftarrow$ *Find_Tetrahedron($\mathcal{N}, \mathcal{E}$)*
{
 for $N_i \in \mathcal{N}$
  for $N_j \in \mathcal{B}(\mathcal{E}, N_i)$
   for $N_k \in \mathcal{B}(\mathcal{E}, N_i)$
    if $(N_j, N_k) \in \mathcal{E}$
     for $N_l \in \mathcal{B}(\mathcal{E}, N_i)$
      if $((N_j, N_l) \in \mathcal{E}) \wedge ((N_k, N_l) \in \mathcal{E})$
       $\mathcal{T} \leftarrow (N_i, N_j, N_k, N_l) \cup \mathcal{T}$
}

This is an $O(s)$ algorithm where $s = \#\mathcal{N}$. Similar algorithms can also be used to find quadrilateral or triangles. The set of neighbour endpoints are stored in the `EdgeTable` class, so most of the information required by Algorithm 1 is very easy and cheap to obtain.

Iterator classes have been developed to loop through the nodes and endpoints so the structure of the code closely mimics our conceptual view as shown in Algorithm 1.

When working on unstructured grids in parallel it is difficult to place any assumptions on the grids, so the flexibility of the node-edge data structure is also advantageous here. In the next section we describe how the node-edge data structure can be extended for use in parallel algorithms.

3 Parallel Multigrid Data Structure

The above data structure must be modified before it can be used in a parallel program. The first modification is to split the node table up into full nodes and ghost nodes. Suppose we have P processors, then the table of *full nodes*, $\mathcal{F}_p^m$, for Processor p satisfies the following conditions,

$$\bigcup_{1 \leq p \leq P} \mathcal{F}_p^m = \mathcal{N}^m, \tag{1}$$

$$\mathcal{F}_p^m \cap \mathcal{F}_q^m = \phi \quad \text{if } p \neq q.$$

If we let $\mathcal{F}_p^0 = \mathcal{F}_p^{n+1} = \phi$ (and $\mathcal{C}^0 = \mathcal{C}^{n+1} = \phi$), then for $1 \leq m \leq n$ the table of *ghost nodes*, $\mathcal{G}_p$, for Processor p is given by,

$$\begin{aligned}
\mathcal{G}_p^m = \{ & \{N_j \in \mathcal{N}^m : \exists N_i \in \mathcal{F}_p^m \text{ s.t. } (N_i, N_j) \in \mathcal{E}^m\} \\
& \cup \{N_j \in \mathcal{N}^m : \exists N_i \in \mathcal{F}_p^m \text{ s.t. } (N_i, N_j) \in \mathcal{C}^m\} \\
& \cup \{N_j \in \mathcal{N}^m : \exists N_i \in \mathcal{F}_p^{m+1} \text{ s.t. } (N_i, N_j) \in \mathcal{C}^{m+1}\} \\
& \cup \{N_j \in \mathcal{N}^m : \exists N_i \in \mathcal{F}_p^{m-1} \text{ s.t. } (N_i, N_j) \in \mathcal{C}^{m-1}\}\} \setminus \mathcal{F}_p^m.
\end{aligned} \tag{2}$$

If the table of connections was not included in the definition then the ghost node table would be similar to the communication buffer often used in the parallel implementation of finite element grids. See for example the ghost cells used in [2] and artificial boundary in [5]. By including the table of connections the inter-grid data dependencies defined by the multigrid algorithm are automatically incorporated into the communication pattern.

The communication pattern is stored in the *neighbour node* table, $\mathcal{Q}_p^m$. Such an addition to the data structure is necessary since the grids are unstructured. Each processor stores two sets of neighbour node tables, one set for the full nodes, $\mathcal{Q}_{\mathcal{F}_p}^m$, and one set for the ghost nodes, $\mathcal{Q}_{\mathcal{G}_p}^m$. We shall represent this structure by

$$\mathcal{Q}_{\mathcal{F}_p}^m(q) = \{N \in \mathcal{N}^m : N \in \mathcal{F}_p^m \cap \mathcal{G}_q^m\},$$

and

$$\mathcal{Q}_{\mathcal{G}_p}^m(q) = \{N \in \mathcal{N}^m : N \in \mathcal{G}_p^m \cap \mathcal{F}_q^m\}.$$

In the parallel implementation we also take,

$$\mathcal{E}_p^m = \{(N_i, N_j) \in \mathcal{E}^m : N_i, N_j \in \mathcal{G}_p^m \cup \mathcal{F}_p^m\}$$

and

$$\mathcal{C}_p^m = \{(N_i, N_j) \in \mathcal{C}^m : N_i, N_j \in \mathcal{G}_p^m \cup \mathcal{F}_p^m\}.$$

When the values for the full nodes are changed the updates have to be sent to any corresponding ghost node. However, the cost of communication can often be reduced by using the ghost nodes to help overlap the communication and computation.

One of the most difficult steps to implement during refinement is the addition of new nodes. When a new node is added to the grid the program must decide if it should be added to the full node table or the ghost node table. One and only one processor may add the node to the full node table. Furthermore the communication pattern stored in $\mathcal{Q}_p^m$ must be updated to take the addition of the new nodes into account. By exploiting the definitions given above we are able to define a set of rules which lets each processor independently decide where to put the new nodes and how to update the communication pattern. Consequently during uniform refinement no communication is required, even for unstructured grids. Unfortunately, some communication is required during adaptive refinement to decide which elements need to be subdivided, but once that is done the elements are subdivided independently across the processors.

The parallel communication is handled by a wrapper class which allows the program to be used with either PVM or MPI.

4 3D Refinement

As mentioned previously, the multigrid algorithm requires several layers of grids. One way of building these layers is to start with some initial coarse grid and then recursively create the next layer by using refinement. The grids may be refined uniformly or adaptively. In this paper we focus on uniform refinement, the issue of adaptive refinement is address in [13].

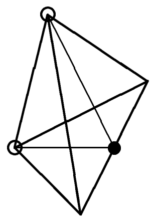

Fig. 1. Bisection of a tetrahedron. The midpoint is joined to the two nodes sitting opposite it.

To subdivide a tetrahedron during refinement we use bisection. This is a fairly simple technique where an edge of the tetrahedron is bisected and the midpoint is joined to the two vertices sitting opposite it (see Figure 1). The edges can not be bisected in an arbitrary way though, rather the shape of the resulting grid must satisfy some conditions which depend upon the properties of the underlying PDE. In this paper we consider only the common situation in which we require a sequence of *non-degenerate conforming* meshes. In 3D there is a large number of possible orderings of the edges, so finding the correct order is far more difficult then in 2D, both in terms of defining a simple algorithm and proving that the resulting tetrahedron have the required shape. One of the simpler algorithms, and proofs, is presented in a paper by Maubach [4] which looks at the more general problem of the bisection of n-simplices. The difficulty, however, is that Maubach's algorithm assumes that the tetrahedron are stored as **ordered** sets of vertices and the node-edge data structure does not contain such an ordering. What we have done instead is to assign different types to the edges and use them to implicitly order the vertices in the tetrahedron.

The refinement routine presented by Arnold et al. [1] is also similar to Maubach's algorithm. However they use a tetrahedron based data structure (i.e. they explicitly store the tetrahedrons) and assign certain types to the tetrahedrons. As we do not explicitly store the tetrahedron we are unable to implement their method.

Another important consequence of using our approach can be found in the parallel implementation. By setting up the initial coarse grid in the way we have we can guarantee that if the edge is bisected in a given tetrahedron then it will also be bisected in all neighbouring tetrahedron which share that edge (including those sitting on different processors). Consequently the techniques used to update the communication pattern during 2D refinement can be used without modification for 3D refinement.

4.1 Maubach's Algorithm

Maubach's algorithm is built on the assumption that each tetrahedron in the mesh is defined by an ordered set of vertices. Let $\tau = \{x_0, x_1, x_2, x_3\}$ be such a tetrahedron. Then the bisection algorithm defined by Maubach is equivalent to:

Algorithm 2
$\mathcal{T}^3 \leftarrow$ *Bisect_Tetrahedron*(τ)
{
$\mathcal{T}^2 \leftarrow \emptyset$, $\mathcal{T}^3 \leftarrow \emptyset$
$\mathcal{T}^1 \leftarrow$ *Bisect_1*(τ) *[Algorithm 3]*
for $\tau^1 \in \mathcal{T}^1$
$\mathcal{T}^2 \leftarrow \mathcal{T}^2 \cup$ *Bisect_2*(τ^1) *[Algorithm 4]*
for $\tau^2 \in \mathcal{T}^2$
$\mathcal{T}^3 \leftarrow \mathcal{T}^3 \cup$ *Bisect_3*(τ^2) *[Algorithm 5]*
return $\mathcal{T}^3$
}

Algorithm 3
$\mathcal{M} \leftarrow$ *Bisect_1*(τ)
{
$c \leftarrow \frac{1}{2}(x_0 + x_3)$
$\tau_1 \leftarrow \{x_0, x_1, x_2, c\}$
$\tau_2 \leftarrow \{x_1, x_2, x_3, c\}$
return $\{\tau_1, \tau_2\}$
}

Algorithm 4
$\mathcal{M} \leftarrow$ *Bisect_2*(τ)
{
$c \leftarrow \frac{1}{2}(x_0 + x_2)$
$\tau_1 \leftarrow \{x_0, x_1, c, x_3\}$
$\tau_2 \leftarrow \{x_1, x_2, c, x_3\}$
return $\{\tau_1, \tau_2\}$
}

Algorithm 5
$\mathcal{M} \leftarrow$ *Bisect_3*(τ)
{
$c \leftarrow \frac{1}{2}(x_0 + x_1)$
$\tau_1 \leftarrow \{x_0, c, x_1, x_3\}$
$\tau_2 \leftarrow \{x_1, c, x_2, x_3\}$
return $\{\tau_1, \tau_2\}$
}

Algorithm 2 shows how a single tetrahedron should be recursively bisected. In his paper, [4], Maubach proves that if this algorithm is followed then the resulting children tetrahedron will be conforming and non-degenerate. We could also re-apply Algorithm 2 to the tetrahedron in $\mathcal{T}^3$ and the resulting tetrahedron will still have the required shape. The next question to ask is what happens if the initial grid has more then one tetrahedron. For example, consider the two neighbouring tetrahedron $\tau_1 = \{N_2, N_3, N_1, N_0\}$ and $\tau_2 = \{N_2, N_3, N_4, N_1\}$ which share the face $\{N_1, N_2, N_3\}$ ($N_0, N_1, N_2, N_3, N_4 \in \mathcal{N}$) as shown in Figure 2. If we pass τ_1 into Algorithm 3 then the two children tetrahedron are $\{N_2, N_3, N_1, N_{\{0,2\}}\}$ and $\{N_3, N_1, N_0, N_{\{0,2\}}\}$, where $N_{\{0,2\}}$ is the midpoint of the edge (N_0, N_2). If τ_2 is passed into Algorithm 3 the two children tetrahedron are $\{N_2, N_3, N_4, N_{\{2,1\}}\}$ and $\{N_3, N_4, N_1, N_{\{2,1\}}\}$, where $N_{\{2,1\}}$ is the midpoint of the edge (N_2, N_1). Hence the resulting grid is not compatible (see Figure 2).

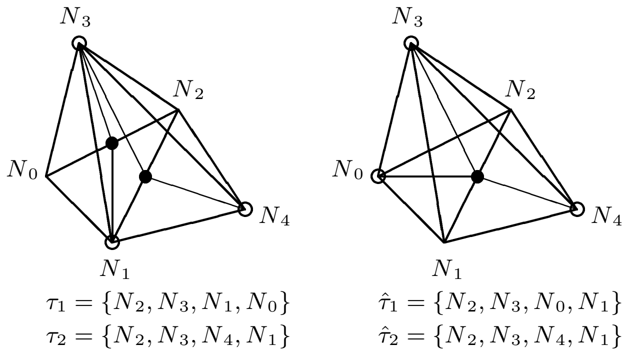

Fig. 2. Example showing the dependence of Maubach's algorithm on the ordering of the vertices. The diagrams show the children created after Algorithm 3 is applied to τ_1 and τ_2 (and $\hat{\tau}_1$ and $\hat{\tau}_2$).

There are techniques which may be used to handle the non-conforming tetrahedron. In fact, for uniform refinement if we waited until Algorithms 4 and Algorithms 5 had also been applied (rather then just considering the grid created by Algorithm 3) then the resulting grid would be conforming. In other words the grid returned by Algorithm 2 is conforming. However, in order to update the communication pattern during refinement we require that the grids returned by Algorithms 3, 4 and 5 are conforming. This also helps during adaptive refinement.

Lets look at the grid in Figure 2 again and renumber it so that $\hat{\tau}_1 = \{N_2, N_3, N_0, N_1\}$ and $\hat{\tau}_2 = \{N_2, N_3, N_4, N_1\}$. Then the children produced by Algorithm 3 are $\{N_2, N_3, N_0, N_{\{2,1\}}\}$, $\{N_3, N_0, N_1, N_{\{2,1\}}\}$ (for $\hat{\tau}_1$) and $\{N_2, N_3, N_4, N_{\{2,1\}}\}$, $\{N_3, N_4, N_1, N_{\{2,1\}}\}$ (for $\hat{\tau}_2$) and hence the resulting tetrahedron are conforming. It is the aim of paper [13] to prove that by ordering the tetrahedron in the initial coarse grid in a certain way will guarantee that the conformality condition is satisfied during each step of the algorithm.

4.2 Edge Based Ordering of Tetrahedron

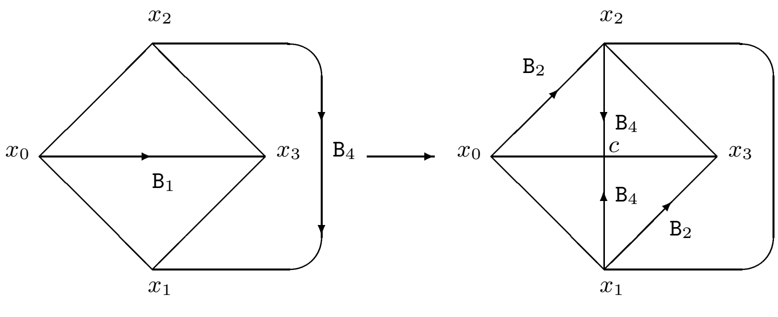

Fig. 3. Diagrammatic representation of Algorithm 3. Edges not marked are of type $\mathtt{B}_5$.

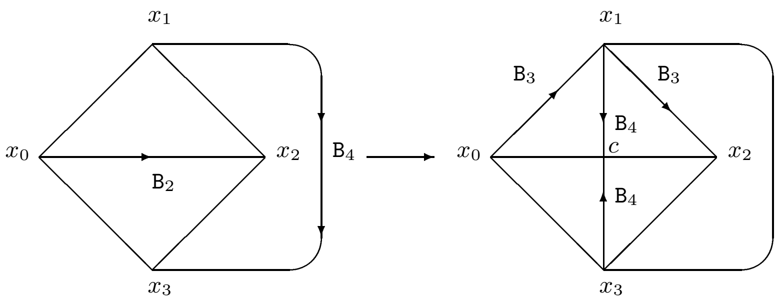

Fig. 4. Diagrammatic representation of Algorithm 4. Edges not marked are of type $\mathtt{B}_5$.

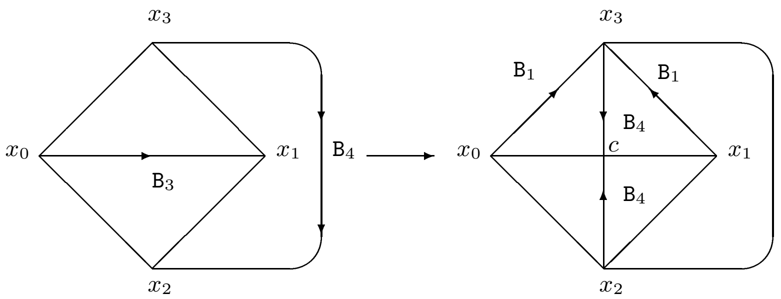

Fig. 5. Diagrammatic representation of Algorithm 5. Edges not marked are of type $\mathtt{B}_5$.

The ordering of the vertices is very important in Maubach's algorithm, however there is no such inherit ordering placed on the nodes in the node-edge data structure. To overcome this problem different types are assigned to the edges. The types place an implicit ordering on the nodes. The types assigned to the edges are; Base 1 ($\mathtt{B}_1$), Base 2 ($\mathtt{B}_2$), Base 3 ($\mathtt{B}_3$), Base 4 ($\mathtt{B}_4$) and Base 5 ($\mathtt{B}_5$).

The general set-up of the refinement algorithms are shown pictorially in Figures 3, 4 and 5. For example in Figure 3 we know, by definition, that the $\mathtt{B}_1$ edge implicitly defines the vertices x_0 and x_3 and the B_4 is used to place an ordering on the other two vertices, x_2 and x_1. Hence we can find the tetrahedron $\tau = \{x_0, x_1, x_2, x_3\}$ which is passed into Algorithm 3. Algorithm 3 creates two children tetrahedron, $\tau_1 = \{\bar{x}_0, \bar{x}_1, \bar{x}_2, \bar{x}_3\} \leftarrow \{x_0, x_1, x_2, c\}$ and $\tau_2 = \{\hat{x}_0, \hat{x}_1, \hat{x}_2, \hat{x}_3\} \leftarrow \{x_1, x_2, x_3, c\}$. If we now look at Algorithm 4 (and Figure 4) we see that the first and third vertices of the children tetrahedron should be bisected next. Hence the edges $\{x_0, x_1\}$ (τ_1) and $\{x_1, x_3\}$ (τ_2) of the children tetrahedron are updated to be of type $\mathtt{B}_2$.

Diagrams 3, 4 and 5 give a pictorial representation of what happens if the algorithm is applied to a single tetrahedron, but what happens if it is applied to a group of tetrahedrons? The results presented in [13] show that there is no conflict between neighbouring tetrahedrons.

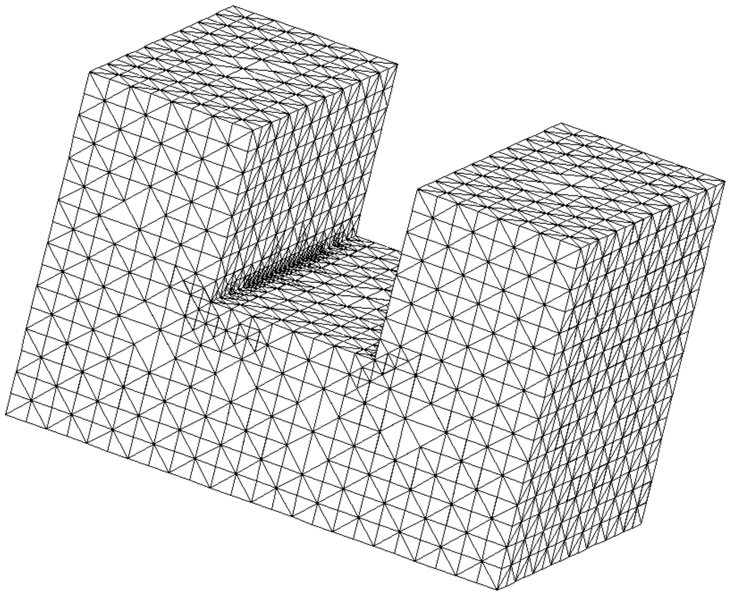

Fig. 6. Example grid created during the solution of Equation 3. Note the extra refinement along the edges and corners. $x \in [0, 3]$, $y \in [0, 2]$ and $z \in [0, 2]$.

5 Example Grids

To finish the discussion we give one example result. We look at the solution of

$$-\Delta u = 1 \tag{3}$$

on the 'horse shoe' domain shown in Figure 6 with boundary condition $u = 0$. This problem has a corner and edge singularity which is captured by the adaptive refinement as shown in Figure 6. Figure 7 shows what the solution looks like if we cut along the $z = 1$ axis.

References

1. Arnold, D. N., Mukherjee, A., Poly L.: Locally Adapted Tetrahedral Meshes Using Bisection. SIAM J Sci Comput., to appear
2. Baden, S. B., Kohn, S. R., Figeira, S. M., Fink, S. J.: The LPARX User's Guide, V 1.0.. Department of Computer Science and Engineering, University of California, San Diego, La Jolla, California 92093-0114, 11 April 1994
3. Briggs. W.: A Multigrid Tutorial. SIAM, Philadelphia (1987)
4. Maubach, J. M.: Local Bisection Refinement For n-Simplicial Grids Generated by Reflection. SIAM J. Sci. Comput., **16(1)** (Jan. 1995) 210–227
5. McBryan, O. A, Frederikson, P. O., Linden, J., Schüller, A., Solchenbach, K., Stüben, K., Thole, C., Trottenberg, U.: Multigrid Methods on Parallel Computers - A Survey of Recent Developments. IMPACT Comput. Sci. Engrg., **3** (1991) 1–75

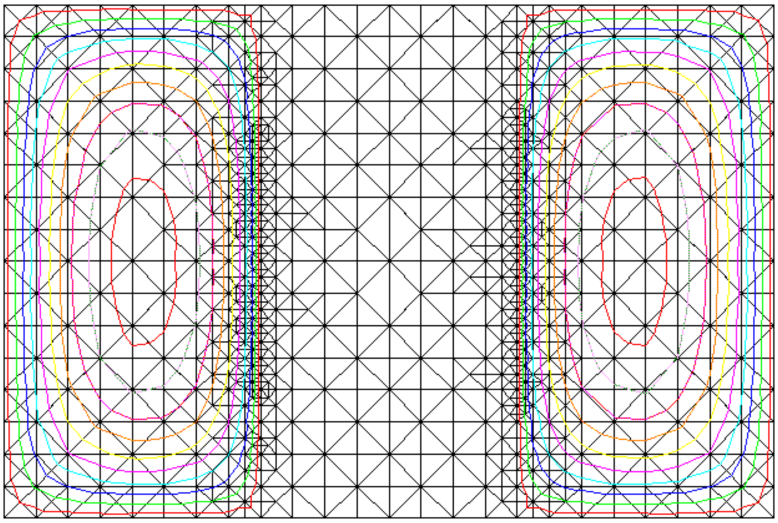

Fig. 7. Example solution of Equation 3. Plot shows 2D slide cut along $z = 1$ axis.

6. Paulus, M., Stals, L., Rüde, U., Rauschenbach, B.: Two-Dimensional Simulation of Plasma-Based Ion Implantation. Journal of Applied Physics, **Vol 85, No. 2**, (Jan. 1999) 761–766
7. Rüde, U.: Mathematical and Computational Techniques For Multilevel Adaptive Methods. SIAM, Philadelphia (1993)
8. Stals, L.: Parallel Multigrid on Unstructured Grids Using Adaptive Finite Element Methods. PhD thesis, Department of Mathematics, Australian National University, Canberra, 0200, Australia (1995)
9. Stals, L.: Adaptive Multigrid in Parallel. In: May, R., Easton, A. (eds): Computational Techniques and Applications:CTAC95. Melbourne, Australia. World Scientific, Singapore New Jersy London Hong Kong (1995) 717–724
10. Stals, L., Rüde, U., Paulus, M., Rauschenbach, B.: Numerical Modelling of the Plasma Source Ion Implantation Process in 2D. In: Noye, J., Teubner, M., Gill, A. (eds): Computational Techniques and Applications:CTAC97. Adelaide, Australia. World Scientific, Singapore New Jersy London Hong Kong (1998) 663–671
11. Stals, L.: A Flexible Data Structure For the Adaptive Refinement of Unstructured Grids in Parallel. The Proceedings of The 14th Kiel GAMM Seminar on Concepts of Numerical Software, January 1997, to appear
12. Stals, L.: Solution Techniques For Radiation Transport Equations. Submitted to Proceedings of the Ninth Copper Mountain Conference on Multigrid Methods, Copper Mountain, Colorado, U.S.A., April 11-16 1999. ETNA (2000)
13. Stals, L., Graham, I.: Efficient Parallel Refinement in 3D. In preparation