Voxel Databases:
A Paradigm for Parallelism with Spatial Structure

Roy Williams
California Institute of Technology, Pasadena CA 91125

Abstract

This paper concerns parallel, local computations with a data structure such a graph or mesh, which may be structured or unstructured. The target machine is a distributed-memory parallel processor with vector or pipeline hardware on the processors, but software based on voxel databases also runs efficiently on shared-memory and uniprocessor machines with and without vector hardware.

A voxel database (VDB) is a distributed shared memory, where entities which share memory are those at the same geometric position. A VDB may be thought of as a dictionary of position-subscript pairs, so that data may be associated with points in space by using the subscript of that point as an index into data arrays. The use of subscripts allows ease of programming and vectorization. The shared memory at each point is weakly coherent and may be either single-writer, multiple reader, also known as the ghost node method; or multiple-combiner, multiple-reader memory. The VDB may be sorted to put special points at the beginning or end of the subscript list, and also to reduce cache-miss inefficiencies. Support is also provided for arbitrary distribution of data entities between processors for load-balancing.

The idea of VDBs and the corresponding software tool provide a clear and efficient way to program a large variety of mesh computations in Fortran or C, such that the execution of the program is independent of the distribution of data to processors, and the mesh may be topologically adapted and load balanced.

As an example we discuss the implementation of an unstructured Finite-Element elliptic solver using a conjugate gradient method. We also discuss the implementations of a finite-volume flux-split fluid solver and of local adaptive refinement for meshes of simplices such as triangles or tetrahedra.

1. Introduction

In this paper we shall consider a method for writing geometrically local algorithms. Such an algorithm is one where objects at the same geometric position may communicate: here such communication is effected by a shared memory at that point.

This view of data movement is exemplified, but not limited to, the Finite Element method [1]. A Finite Element mesh is a partitioning of the problem domain into elements, each of which is defined by a set of points in the domain, called nodes. The basic structure is that of a set of elements, where each element is a set of nodes. The computational kernel consists of each element reading and writing from and to its subset of the nodes. Writes to the nodes are separated from reads of those data by a synchronization. Each element refers to a node by local numbering; for example the nodes of a tetrahedron might be 1, 2, 3, 4. In addition to the local numbering, there must also be a global numbering, so that the nodes may be thought of as shared memories. Two elements referring to the same node will refer to the same global node number and thus share memory.

Thus we may split a Finite Element code into two parts, one which deals with the properties and nature of the data stored with elements and nodes and the calculations done with those data, and another part which maintains the shared memories at the nodes and supplies the mapping from local to global numbering. The latter part may be done with a Voxel Database (VDB), producing code to run on a uniprocessor or on a distributed machine.

Finite volume methods may also be coded with a VDB. In this case the problem domain is also partitioned, but the pieces are conventionally called cells rather than elements. Each cell reads data from neighboring cells in order to update itself. Suppose we associate the cell data with a unique point of the cell such as its center of mass. Then each of these points is a shared memory, with one cell writing there and the neighbors reading.

A Set of Sets of Points

The data structure for a VDB is thus a collection of subsets of a set of points. We shall call these sets of points *pointsets*. A tetrahedral Finite Element mesh, for example, is a set of all the nodes together with a set of pointsets (elements) of size four. A graph may be represented as a set of nodes plus a collection of pointsets (edges), each of which is of size two.

A set of pointsets may be thought of as a table of subscripts, such as that for the graph shown in Figure 1. This is a

Local Number	Pointset Number: 1	2	3	4	5	6	7	8	9	10	11	12
1	1	3	3	2	7	4	8	6	8	5	5	2
2	5	4	2	1	3	7	7	7	6	6	4	8

Global Number

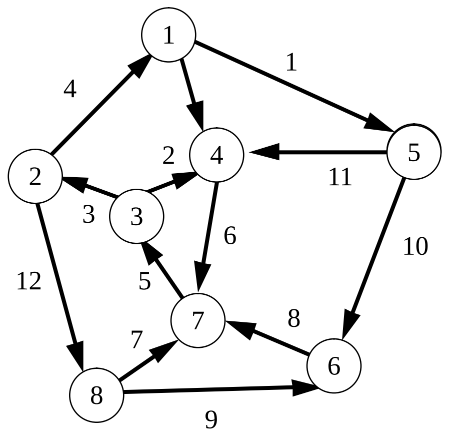


Figure 1: A graph represented as a collection of points and a collection of pointsets of size 2 (edges).

spatial graph, meaning that each of its nodes is associated with a geometric point. Since the structure represented is a graph, each pointset is of size two: the local numbering is at the left of the table. There are 12 pointsets at the top of the table, and each table entry is the corresponding global number. The global numbering for the points may or may not be *defragmented*: meaning that the points are numbered sequentially from 1 to the number of points. In Figure 1 the point numbering is defragmented because the eight points are numbered 1, 2, ..., 8 rather than say 3, 6, 11, 17, 13, 2, 16, 77. When the point numbering is defragmented, we may set up a loop over the points rather than always looping over pointsets and using the table of subscripts.

It should be pointed out that the above subscripts begin at 1, which is appropriate for the Fortran implementation of VDBs: in the C implementation subscripts begin at zero.

Programming with Subscripts

High computer performance may be expected from a defragmented numbering. The defragmentation means that vector or pipeline hardware may be efficiently used when cycling through the nodes, and reading data from a point may be achieved with a single add and a memory access.

There may also be performance increases hidden in the language. Fortran is a language in which compiler-writers strive to excel, not only because of its popularity, but also because the programmer is coerced into using arrays rather than lists or objects, and the simple structure of an array gives great freedom for optimization and automatic vectorization.

Unstructured data structures such as these are, unfortunately, subject to memory bottlenecks on many modern processors. Such processors have a cache which is much smaller than the main memory, and peak arithmetic performance can only be obtained when data is resident in the cache. But a table of subscripts implies selecting data from all over the main memory, resulting in cache misses and a loss of performance. A good implementation of a VDB can, however, improve this situation substantially by renumbering the points, so that when the pointsets are used in order, most of the points required will already be in cache from the previous pointset.

Previous Work

The type of algorithms addressed by this paper are where pointsets read from, compute with, and write to data associated with their points, where there is a synchronization of all the processors after a write and before a read of the data which was written.

Let us suppose that the pointsets are distributed among the processors of a distributed-memory parallel machine, so that each processor owns a subset of the pointsets, and each pointset is owned by exactly one processor. If we further suppose that each point is owned by exactly one processor, then some points of a pointset may be in the memories of different processors.

Previous work, such as the Kali [2, 3] language and the PARTI run-time library [4, 5], has succeeded in creating effective user interfaces for efficiently running unstructured mesh algorithms on distributed machines. However, both of these use static meshes; Kali compiles the code so that off-processor memory references are available when needed, and PARTI sets up the communication strategy at run-time after the mesh definition has been read in: off-processor references are fetched into an on-board cache before they are needed [6].

These methods, however, are based on static meshes, so that the complete computational mesh must be read in before calculations can start, resulting in a serious sequential bottleneck. An application that can adapt its mesh can read in a mesh of just a few elements, and adapt it to the necessary resolution in parallel, and furthermore adapt it locally as the simulation demands more or less resolution [7, 8, 9].

We might interpret writing data to a point as sending a message, and reading as the sending of a message by the neighbor. Because the messages are short, communication must therefore be overlapped with computation to achieve efficiency. The datum may be pre-fetched before it is needed, requiring a lot of intelligence from the runtime library [10].Alternatively, with an asynchronous dataflow language such as PCN [11], the application may be broken down into many threads so that the processor always has something else to do during communication: the result is that the source of inefficiency is shifted from communication to process switching.

An application written with VDBs leaves control of communication with the application by explicit synchronization calls, keeping local copies of data in all the processors that need them. The copies may have all the same status (the combining method), or may be copies of a master datum which are updated at synchronization. The assumption is that data objects (pointsets) need only communicate (i.e. share memory) if they share a geometric point in space. A VDB may be constructed, adapted and load-balanced in parallel with bottlenecks which are only logarithmic in the number of processors.

Creating a VDB

First a VDB is created, specifying the number of dimensions of the space in which the points lie, and then some points stored in it. When a point is stored by giving its coordinates, a subscript is returned. If that point has been previously stored, the same subscript is returned as previously; otherwise it is the next available unused subscript.

Two points need not have exactly the same position to be regarded as the same point. In creating the VDB, the user specifies a non-zero *tolerance*, such that any pair of points closer than the tolerance will be regarded as the same point, and points further away than a small multiple of the tolerance will be regarded as different points. This small multiple is three times the square root of the dimension of the space. Thus the tolerance is less than the smallest expected distance between points.

Thus if new names are put into a newly created VDB, the return values are 1, 2, 3, ... (or 0, 1, 2, .. for the C implementation). Finally the VDB is “synchronized”, which does nothing when running on a uniprocessor, but establishes communication paths between processors for a distributed machine, and will be explained fully below. This synchronization call may be thought of as making concrete all changes to the VDB made since the previous synchronization.

A simple use of a VDB is a follows. Suppose an application is given a description of a mesh expressed as a collection of pointsets, each consisting of some point positions. There may also be another file with records consisting of point positions and data to be associated with them, which we shall ignore for the moment. To read the mesh description, we initially create a VDB, and pass each point of each pointset to it and store the resulting subscripts. The code for this example is presented in Section 3. We may then read in the file of points and attach data to the arrays using the returned subscripts from the first stage.

Software Design

The criteria used in the design of the VDB software are:

- The results of an application code should be independent of the distribution of data entities among the processors of the parallel machine to make debugging possible;
- Shared-memory synchronization should be fast;
- Communication is explicitly initiated by the application code;
- Topological changes to the graph and load balancing should be possible, and preferably easy;
- Applications should be written with common programming languages such as Fortran or C, leaving control with the application code rather than providing a callback service.
- The application data is kept in vectors rather than linked lists to facilitate vectorization;
- The code should be scaleable to massive parallelism.

The software discussed below is written with the Express parallel processing environment [12], with a Fortran or C interface. The idea of VDBs is a generalization of the idea of local combining utilized in the DIME (Distributed Irregular Mesh Environment) software [7].

We shall discuss how some common algorithms may be implemented with VDBs, the examples being a finite-element solver on an unstructured mesh using the conjugate-gradient method, a flux-split finite-volume fluid solver, and local topological refinement of an unstructured tetrahedral mesh by the Rivara algorithm [13].

2. Distributed Shared Memory

We replace the concept of a point by an equivalence class of points which share memory, where two points are equivalent if they have the same position. Thus a point is actually a set of *alias* points, each of which has the same position.

The set of points of the pointsets owned by a processor are all within the processor, so that all accesses made by the pointset are local. When a pointset in one processor wishes to access the data of one of its points, it accesses a copy of the point (an alias) stored locally. The point has been replaced by a *distributed shared memory* [14, 15, 16], and as with any shared memory, we must be careful that the different access mechanisms (different aliases) produce the same results.

Coherence

The idea of shared memory requires some clarification, since there may be incoherence when several processors write to the same memory location. A shared memory may have *strong coherence*, meaning that as soon as a datum is written by any processor, then any reader is guaranteed to read the updated datum. This implies that timing variations between processors affect the results of the code, and we shall not consider this update method further.

Weak coherence means that code contains *weak blocks*[14]. A shared datum may only be read outside a block in which that datum is written, and may only be written inside such a block. At the end of a weak block, there is a synchronization [17] so that the effects of the writes may be reconciled.

Combining

Writing to a memory location implies replacing whatever in the location by something else. Another way to “write” to a shared memory is not to allow the conventional idea of write, but instead allow only *combining* with the location. An example of a combining operation is addition, so that a processor may add a number to a memory location. Thus if several processors add numbers to the shared memory location then synchronize, the result is independent of order. There is no need for global ordering of data access or singling out a processor as the one which is allowed to write.

Thus pointsets may read from their points, but may only combine with, rather than write to, a point. The combining block is ended with a synchronization where the aliases are combined with each other by the VDB software, after which that datum is available for reading.

Updating

A VDB also allows a more traditional method of writing to shared memory, based on the idea of write permission, which we shall call *updating*. Only one processor has write permission to the memory at any given time. Of all the aliasess at a given point, exactly one may write; this special alias of all those at a given position is the *secretary* alias. More accurately, any may write, but after synchronization the value in the memory is the value written by the secretary point.

3. Creating a VDB: Reading a File

A simple example was given above of using a VDB for reading a file describing a spatial graph. The code for this is shown in Code 1. There are `nedge` records in the data file, each representing an edge (pointset of size two), and we

```
      real fromnode(idim), tonode(idim)
      integer table(2, maxnode)
      character buf(ibufsz)
c--Create the VDB
      ivdb = VCREAT(idim, tolerance, 0, maxnode, buf, ibufsz)
c--Store the names
      do iedge = 1, nedge
         read(4, *) fromnode, tonode
         table(1,iedge) = VFJOIN(ivdb, fromnode)
         table(2,iedge) = VFJOIN(ivdb, tonode)
      enddo
c--Synchronize the VDB
      nnode = VSYNC(ivdb)
```

Code 1: Creating a VDB for a spatial graph from a file.

suppose the points to be in 3-dimensional space. The VDB is created with a dimension of `idim` and a spatial `tolerance`, the “gravity flag”, a given maximum number of entries, and a buffer and its size for internal use, and an integer tag for the VDB returned. The gravity flag indicates whether or not exact floating-point coordinates will be given to the VDB or not. The point positions `fromnode` and `tonode` are read in and stored in the VDB using `VFJOIN` (`VDJOIN`) for single (double) precision coordinates, and the returned subscripts kept in a table. Finally the number of points in the VDB is returned by the synchronization function, and data associated with the nodes (points) of the graph may now be stored and retrieved.

Parallel Operation

This simple application also runs on a distributed memory parallel processor, except for the decision of which processor gets which pointset. Either each processor may read a separate data file, or each processor could see all the records from a single global file and decide which records belong to it, perhaps by a round-robin method. The advantage of the latter is that the number of processors reading the file may be different from the number which wrote the file.

At the left of Figure 2 is shown a graph of eight nodes. At the right is the same graph distributed among three processors, each in a different shade of gray. Some points have been replaced by equivalence classes of alias points, and each processor has obtained from the synchronization function the number of nodes `nnode` as shown. The sum of all these is 15, which is of course greater then the 8 equivalence classes of aliases.

4. Meshes

We have seen a spatial graph represented as a collection of pointsets (edges) each owned uniquely by a processor, and each of which points to two shared nodes (points). We may generalize this to any collection of uniquely owned

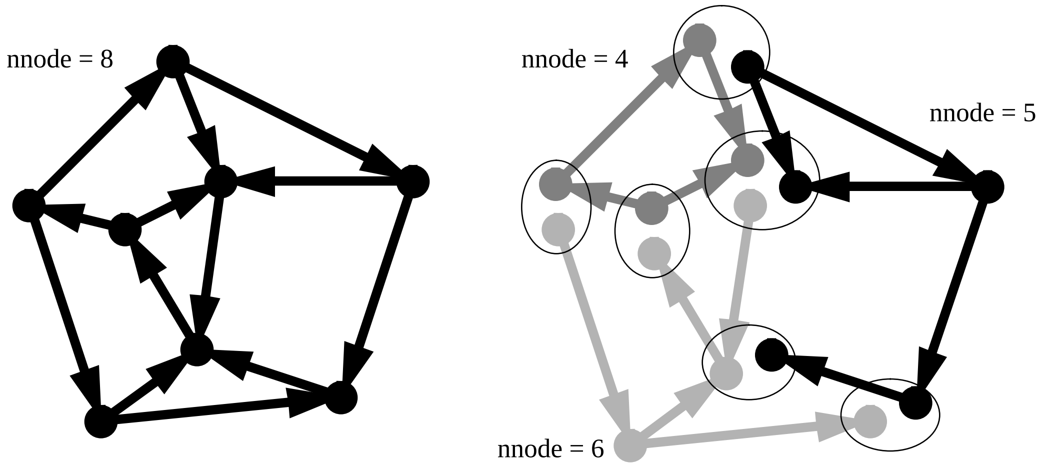


Figure 2: A graph and its representation on three processors of a parallel machine.

objects pointing to a subset of a set of shared objects. Figure 3 shows some examples of imposing additional structure

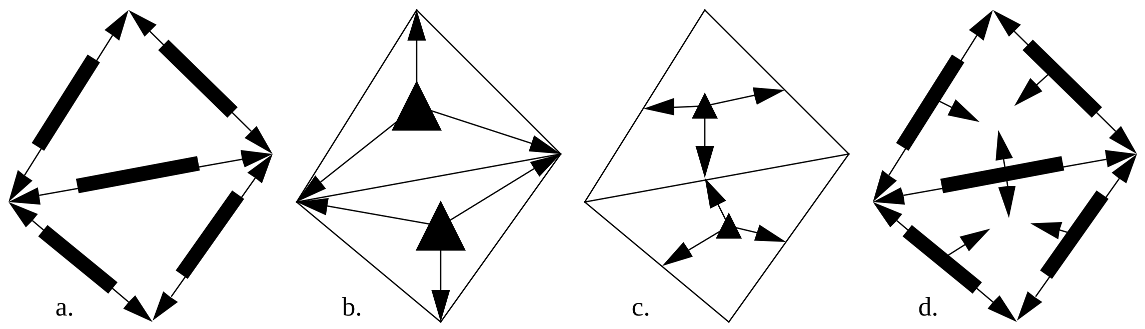


Figure 3: Four ways to use pointsets to represent spatial structure. a. Edges point to 2 nodes; b. Elements point to 3 nodes; c. Elements point to 3 edges; d. Edges point to 4 nodes.

on a triangular mesh of two triangles.

At the left the mesh is viewed a graph, with a collection of edges each pointing to nodes; this is the structure used by Mavriplis [18] in a Navier-Stokes solver. Next is a structure that might be useful for linear Finite Element calculations, with element structures each pointing to the three nodes at the vertices of the element. Figure 3c shows cells each pointing to three edge structures, which could be used in conjunction with the previous structure for quadratic Finite Elements. Figure 3d shows a structure where each edge points not only at the nodes at its ends, but also at the nodes opposite the edge. Such a structure would be relevant for the icosahedral algorithm for the shallow water equations [19].

5. Combining: Sparse Matrix Multiply

The core of a linear or implicit finite-element code [1, 20] is the solution of a matrix equation $Ax = b$, where the elements of the vectors b and x are associated with points, and the stiffness matrix A is a sum of local matrices, each associated with a pointset. Iterative methods for solving $Ax = b$ such as overrelaxation or conjugate gradient depend on being able to multiply a vector by the stiffness matrix.

There are node arrays `x` and `Ax`, to hold the vector to be multiplied and the result respectively. Each element of each of these vectors is one of the weakly coherent shared memories discussed above. Code 2 carries out the sparse

```
      real x(maxnode), Ax(maxnode), A(3,3,maxelmt), xloc(3), Axloc(3)
      external fsum
c-- Set result to zero
      do inode = 1, nnode
         Ax(inode) = 0
      enddo
c-- Weak block on Ax
      do ielmt = 1, nelmt
         do ivert = 1, 3
            inode = table(ivert, ielmt)
            xloc(ivert) = x(inode)
         enddo
c     Here multiply A(*, *, ielmt) by xloc(*) to get Axloc(*)
         do ivert = 1, 3
            inode = table(ivert, ielmt)
            Ax(inode) = Ax(inode) + Axloc(ivert)
         enddo
      enddo
      call VCOMB(ivdb, Ax, fsum, 4, 4)
c-- End weak block on Ax
      end

      subroutine fsum(a, b)
      real a, b
      a = a + b
      end
```

Same Operation

Code 2: Sparse matrix multiply for an unstructured triangular mesh.

multiplication for the structure shown in Figure 3b when each pointset is of size 3. Each element is associated with a 3x3 local stiffness matrix; for each element, we extract a locally numbered vector `xloc` from the global vector `x`, multiply this by the local stiffness matrix to produce the locally-numbered `Axloc`, then increment the globally-numbered result vector `Ax` by `Axloc`.

In this case we are using the combine method to store data in the shared memories, and the call to `VCOMB` in the last line synchronizes the combine over aliases of the points in other processors.

One reason for showing this code is to demonstrate that the main loop over elements may occur in parallel. At first sight it appears that this is not the case, since the value of the vector `Ax` after a loop iteration is dependent on the value of `Ax` before the loop iteration. However, at each loop iteration we are only adding numbers to `Ax`, and the sum of a set of numbers is independent of the order in which the numbers are added in. Binary operations with this property, such as arithmetic addition, are *combining operations*.

The code contains a weak block with respect to the array `Ax`, and values of `Ax` may not be reliably read inside this block. The block is ended by the call to `VCOMB`, giving the size in bytes of the data to be combined, and the stride in bytes, which is the separation between successive data elements. Before this call, each alias node has contributions from elements which are on the same processor, but we would like each alias to have a value which is the sum of contributions from all edges, which is precisely the case after the `VCOMB` call.

It is crucial that the method of combining data within the weak block (arithmetic addition) be the same as the operation accomplished by the `VCOMB` call as indicated in the sum function passed to it. If this were not the case, then results would not be independent of the distribution of the graph among the processors.

Arithmetic addition is not the only combining operation allowed on node data, as explained in the next Subsection.

Definition: Combining Operation

An operation $\oplus$ is a function of two members a, b of a given set T and returns a member of T. An operation is a combining operation if the following properties of $\oplus$ hold:

- (Identity) There is a member i of T such that $i \oplus a = a$,
- (Commutative) For all a, b in T, $a \oplus b = b \oplus a$,
- (Associative) For all a, b, c in T, $a \oplus (b \oplus c)) = ((a \oplus b) \oplus c)$.

Other examples of combining operations are:

- Arithmetic addition and multiplication
- Logical AND and OR
- Maximum and minimum
- Concatenation of records (if the order of the records is not important)

If we write a simple loop to go through a list of numbers to find the total, the result is independent of the order in which we go through the list, which is of course because addition is a combining operation.

Some combining operations, in particular arithmetic addition, are not exactly associative, but may be different in the last bit or so. If it is essential that each processor reads exactly the same values, then an combine using addition should be followed by another combine, but with maximum, which is exactly associative.

Message Passing by Combining

We have described a mechanism by which the processors of a parallel machine have a restricted kind of shared memory. A processor may read from the copies of its points, or write in the restricted sense that they do not overwrite the contents of the shared memory, but execute a combining operation with the memory. Independence of the distribution of cells over processors is assured if every node-array is created by setting its value to the identity of a combine operation (this is zero for arithmetic addition), then combining in other data, then using the same combining function to combine over those nodes shared between processors.

Experience shows that many if not most numerical applications may be implemented with no more communication than that provided by combining and updating operations. We should point out however that we may simulate a loosely synchronous message passing machine using combining operations, so that there is more generality than might be thought.

The architecture of this virtual machine is that of the mesh we have stored in the VDB. We may think of the pointsets in Figure 3 as virtual processors, which may send messages to other pointsets with which they share a node.

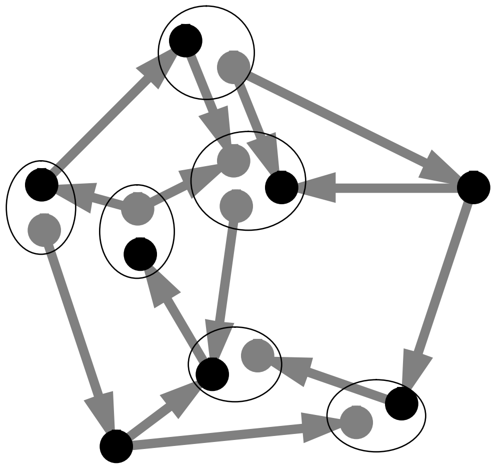
The idea is simply to use concatenation as the combining operation; each pointset makes some messages, each with an "end-of message" character, and combines this buffer of messages with an initially empty buffer in the nodes, with the combine operation being concatenation. After the call to `VCOMB`, each pointset may read the messages (in some arbitrary order) from the nodes.

6. Global Combining

Thus far we have described ways to do *local* operations in parallel, and we need a little more to make the VDB mechanism a useful tool for parallel graph and mesh calculations. Consider for example implementing the conjugate gradient algorithm [1] for solving a sparse matrix equation $Ax = b$. There are two basic ingredients in this algorithm, being matrix-vector multiplication and calculating scalar products. The former may be accomplished with Code 2 as previously discussed, and the latter requires computing a global sum over all the nodes.

Figure 4 shows the graph of Figure 2, but with some of the alias points black and some gray. The black points are *secretary* points, and there is exactly one of these for each equivalence class of points. Thus if we want a sum over all equivalence classes, which would be independent of the distribution of pointsets to processors, we sum over secretary points rather than all points. The function `VSECR` fills an integer array with 1 if the point with that subscript is a secretary node or else 0. The number of copies of a node depends on the distribution of cells to processors, but there is only one representative for each equivalence class of nodes.

Figure 4: For each equivalence class of point aliases, there is exactly one secretary point, shown in black. The number of classes is thus the number of secretary points.



The decision of which node of an equivalence class is the representative is made arbitrarily by the VDB software during the call to `VSYNC`: there is a call to change this, but if two processors claim to be the secretary of the same point, and error occurs.

Code 3 calculates the scalar product of the vectors `s` and `t`, putting the result in the variable `dot`. There are two

```
integer isecr(maxnode)
VSECR(ivdb, isecr)
dot = 0
do inode = 1, nnode
   if(isecr(inode) .eq. 1) then
      dot = dot + s(inode) * t(inode)
   endif
enddo
call VGLOB(dot, sum, 4, 4)
```

Code 3: Global combine operation used to compute a scalar product.

statements in this code segment which would not be necessary for a uniprocessor code: the `if` statement and the call to `VGLOB`.

We begin by setting the variable `dot` to the identity for arithmetic addition (zero), then loop through the points adding the contribution from each point to the scalar product, which is the product of the elements of the vectors `s` and `t`. Now each processor has a sum over the secretary points it owns, and we replace this by the sum over all the processors of values of `dot` with the call to `VGLOB`. If the values of `dot` on four processors were 1, 4, 2, 5 before the global combine, they would be 12, 12, 12, 12 afterward.

The `if` statement is thus because each point should only contribute once to the scalar product, rather than a number of times depending on the distribution of pointsets to processors. In particular we may ask for the total number of equivalence classes of points in the mesh, which would be the sum over processors of the sum over points of the return values of `VSECR`, a quantity independent of distribution.

The arguments to `VGLOB` are similar to those of the local combine `VCOMB`, except that `VGLOB` does not need a VDB tag since it is a global sum. This global combine is functionally identical to the global combine of Express [12].

7. Load Balancing, Defragmentation and Sorting

A load-balancer consists of two parts: a method of deciding which data entity should reside in which processor [9,21], and a method of migrating the data entities to their new processors and reconnecting the distributed data structure. This paper does not deal with algorithms for the former, but only with the migration process itself.

We assume that by some means that we have an array containing the processor number in which each pointset should reside after the load balancing. For each pointset, we check if the new processor is the same as the current processor, in which case no action is needed. If not, we package the pointset data and the data of its points into a message which may then be despatched. During the message transit time, we may call `VDSTRY` to destroy the VDBs, freeing their memory, and create new VDBs to hold the point data. Now we receive all the messages containing pointsets which are to reside in this processor. For each of these we copy the pointset information, and call `VFJOIN/VDJOIN` for each of the points. Finally we call `VSYNC` to synchronize the new VDBs. The load-balancing process fragments the point arrays, and they should now be defragmented.

Sorting the point arrays to minimize cache-miss inefficiencies is actually rather similar to load balancing, since the same recursive bisection methods may be used for point renumbering, except within a processor rather than between processors.

A clean user interface will be made for these processes in the near future.

8. Summary

In the previous Sections we have seen how to store a set of points in a VDB, and from each point extract an integer which may be used as a subscript. In this way we may attach data to a geometric point in space. When the same point is stored in the database by different processors of a parallel machine, a connection is set up between the processors so that the data associated with that point is shared between the processors.

Each processor may read at any time from these shared memories, but changes to a shared memory must be made by means of a combining operation or by having a representative called a secretary overwrite the others in its equivalence class. When running on a distributed-memory machine, data cannot be read from memory locations that have been combined or updated until the VDB is synchronized with `VSYNC`.

The Fortran interface to he VDB software is:

- `VCREAT` - To create a VDB (loosely synchronous),
- `VFJOIN/VDJOIN` - To store a point in the VDB, with single/double precision coordinates,
- `VSYNC` - To synchronize a VDB (loosely synchronous),
- `VCOMB` - To replace local data with data combined over the whole machine (loosely synchronous),
- `VUPDAT` - To replace local data with data from secretary aliases (loosely synchronous),
- `VGLOB` - To combine globally (loosely synchronous),
- `VSECR` - To fill an array with 1 if a point is a secretary of a class shared over processors, else 0.
- `VNMEMB` - To fill an array with the number of times `JOIN` was called at that position.
- `VDELET` - To delete a single point from the VDB,
- `VDSTRY` - To destroy the VDB and free the associated memory (loosely synchronous).

Those subprograms listed as loosely synchronous must be called in all the processors of the machine, not necessarily at the same wall-clock time, but at the same logical place in the program flow [17].

The example used as a framework for describing the VDB software was an implementation of a linear finite-element code, with a collection of cells initially distributed among the processors of the machine, which were then connected together. The computational kernels of multiplication of a vector by a stiffness matrix and scalar product were then presented, and timings are given in Section 12.

It should be noted that all data is retained in linear arrays rather than linked lists, so that vector or pipeline hardware may be used if it is available; also that both data and program flow are controlled by the application.

9. Implementation

In the introduction it was mentioned that one of the prime objectives of the VDB paradigm is that its implementation should be as efficient as possible to utilize the full power of a parallel machine.

VDBs are set up to make the combine/update operations as fast as possible. The setup must happen when a mesh is created, read in from a file, or when restructuring occurs such as adaptive refinement or load-balancing. The assumption is that these processes are much less frequent than combining/updating; that an application will spend much more time computing with a fixed mesh than with changing the mesh configuration.

The following information about an implementation of VDBs is not necessary for correct and efficient use.

The data structure for a VDB is a hash table [22]. When a point is passed to `VFJOIN/VDJOIN`, its is converted to integer coordinates (a voxel) by dividing by the tolerance for the VDB, then a hash function finds a place in the hash table where that integer point should be. If there is nothing at that place in the hash table or in the neighboring voxels, it is a new point which is not in the database; a new *voxel-entry* is put in that place in the hash table consisting of the voxel coordinates and the next available subscript, and the subscript returned. Alternatively, there may already be a voxel-entry at that place in the hash table, or more generally a linked list of voxel-entries. The list is searched for an exact match to the voxel; if it is found, the subscript of that voxel-entry is returned, or else the voxel is not in the database, so a new voxel-entry is added to the end of the linked list.

The hash-table size is set to be the largest prime less than 0.05 times the maximum number of points given as an argument when the VDB was created. The memory used by the hash table is four bytes times the table size. If the hash table size is too small, the creation of a VDB will be unduly inefficient. A subroutine is provided to print statistics about the hash table usage.

When `VSYNC` is called, all of the new voxel-entries are packaged into a message, and the messages passed from processor to processor until each has seen each message. The message-passing is implemented by arranging the processors in a ring, with messages sent in batches in the buffer provided in the call to `VCREAT`.

Each processor takes each voxel-entry from each other processor and looks it up in the hash table as above. If a match is found, an *alias* entry is attached to the voxel-entry, showing that another processor has been given a point in that voxel, containing the number of the other processor and a pointer into the memory of that processor for the relevant voxel-entry. There may be a linked list of such aliases in other processors attached to a voxel-entry.

In addition, a combine table is formed to make combining and updating as efficient as possible. The length of the combine table is the same as the number of processors being used, and from each entry begins a linked list though those aliases which point to the corresponding processor number.

10. Applications

Many Dimensional Problems

It should be pointed out that the dimensionality of the space is an input to the VDB software, so that code may be written to simulate phenomena in 1, 2, or 3 dimensions. Higher dimensions may also be useful: for example a sphere may be defined by the three coordinates of its center and its radius, so the sphere can be thought of as a point in a four-dimensional space.

Mesh Boundaries

Usually in mesh calculations, boundary surfaces are treated differently from the interior (the following discussion assumes a 3D mesh). There may be data associated with boundaries such as what type of boundary condition should be applied or values of external fields driving the interior physics. There may be even be an algorithm requiring communication within the boundary, such as solving a boundary element problem or finding the curvature of the boundary.

First we need to find which faces of the mesh cells are boundary faces. This can be done by setting up a VDB of cell faces: each cell uses the center of each of its faces as the point of shared memory. We use `VMEMB` to fill an array with the number of times a point was `JOIN`ed at that position, which is two for internal faces, 1 for boundary faces. The VDB may now be destroyed if desired.

These boundary faces form a two-dimensional mesh, and we may wish to form a VDB of boundary nodes and/or boundary edges for solving problems associated with the surface mesh.

Finite-Volume Flux-Split Fluid Solver

Another application of meshes is in finite-volume methods [23,24,25]. In this case the computational kernel is that

- Each interior face (in 3D) of the mesh takes the data from the two cells which share that face, and transforms that data with an approximate Riemann solver [24.
- Now each cell takes the data from each of its faces, and transforms its own data using the face data.

We would form a VDB of centers of faces as above. By exchanging centers of mass of cells which share each face, cells in other processors may be considered ghost cells, and their data updated with the update mechanism. Now we may loop through the faces applying the Riemann solver, using the ghost-cell data where that cell is in another processor, or else obtaining the data locally if possible.

Quadratic Finite Elements

Solving a finite-element problem with higher-order elements than linear makes no major changes to the code presented in Code 3. For the example of quadratic triangular elements, each element contributes to six nodes rather than three. Now points corresponding to not just the vertices of each triangle, but also the midpoints of the edges are sent to the database, and each local stiffness matrix is 6x6 rather than 3x3.

11. Local Refinement of Simplex Meshes

One of the major advantages of unstructured over structured meshes is the ability to adapt the mesh to improve resolution at a place in the simulation which needs it. So far we have only discussed static meshes, and now we show that the VDB paradigm is capable of supporting topological changes to the mesh structure. We shall be considering for this example meshes consisting of simplices, meaning a triangular or tetrahedral mesh. The following discussion is appropriate to both.

Adaptive refinement of a simplex mesh may be accomplished by the algorithm of Rivara [13], which is illustrated for triangles in Figure 5. The application code selects a set of cells to be refined, based presumably on either user input or

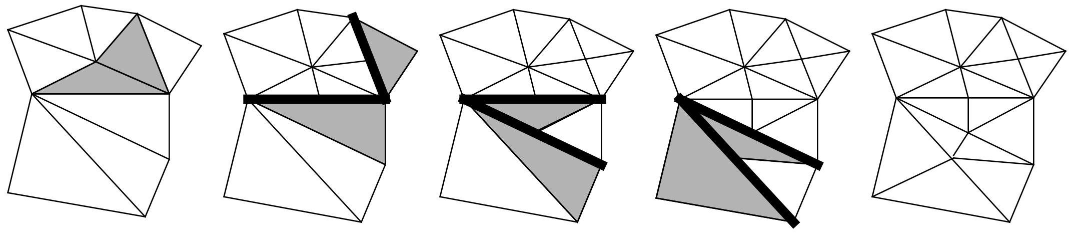


Figure 5: Stages of the Rivara refinement algorithm used to create a graded mesh. The algorithm is for general simplex meshes: here illustrated in two dimensions.

on the data representing the state of the simulation.

The first panel of the Figure shows a triangular mesh with two cells marked by shading, which are the cells that are to be refined. Now we loop around the following cycle until there are no more marked cells:

1. Each marked cell is bisected by a line (plane) passing through the midpoint of the longest edge of the simplex, and through the opposite vertex (vertices).
2. Each cell examines its edges looking for an edge that has been bisected by another cell which shares the edge. If any are found, the cell is marked for refinement.
3. If there are no marked cells, the refinement is finished, or else go back to (1).

Each panel of Figure 5 shows the situation at the end of subsequent cycles. The final panel shows the refined mesh.

More formally, we may consider each cell to have one of three states,'marked', 'unmarked' or 'deleted', and each edge to have one of two states, being 'deleted' or 'undeleted'. We loop through the cells, setting the state to 'marked' or 'unmarked' depending on whether refinement is required for that cell or not. All edges begin as 'undeleted'.

In step (1), we loop through the cells looking for those whose state is 'marked'. The states of this cell and its longest edge are set to 'deleted' and two new unmarked cells are created. For each of these new cells, some edges are identical to those of the original cell, and the new cell points to these; there is one (three) of these for each new cell in 2D (3D). Some new edges, including the two with half the length of the one which has been bisected, are created with the state 'undeleted' for the new cells to point to; there are two (three) of these for each new cell. These new edges are passed to `VJOIN` to obtain subscripts for them.

Between steps (1) and (2) is a call to `VSYNC` to connect the new edges into the VDB of edges. Now we combine over edges with `VCOMB`, taking the logical OR of the edge states, so that if any cell changed the state of an edge to 'deleted', then cells which share that edge can find out.

Steps (2) and (3) are as above, where the each cell checks for edges marked 'deleted', and if there is one, changes its state to 'marked'. If there are any marked cells, then repeat the loop, or else we may actually delete all the cells and edges whose state is 'deleted' with `VDELET.`

This refinement algorithm, together with an unrefinement algorithm in preparation, will enable coding of unstructured multigrid applications. A clean user interface for addressing simplex meshes will be made.

12. Timing

Times are shown in Figure 5 for the sparse-matrix multiply code of Code 2. An unstructured triangular mesh is being

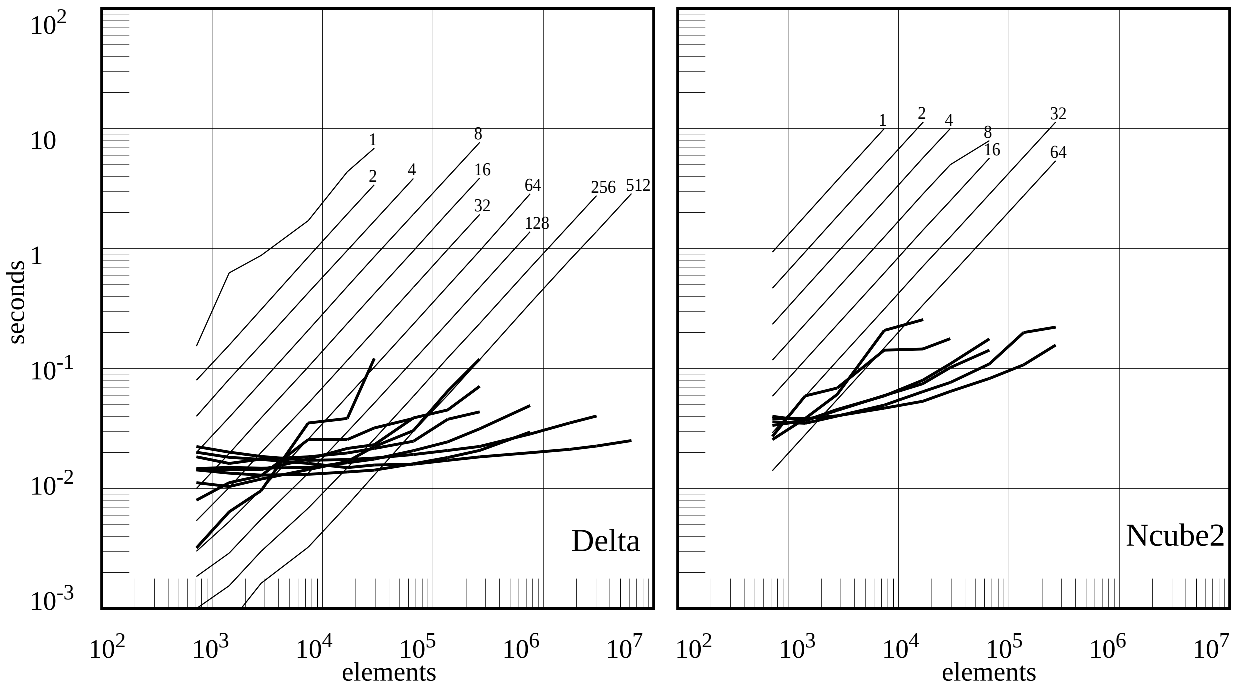


Figure 5: Timings for the sparse matrix multiply of Code 2 for the Intel Touchstone Delta and Ncube2 machines with Express. The parallel diagonal lines show calculation time, the lower lines show communication time, which is ~100 times less than the calculation time. The maximum number of elements was 6.5 million.

adaptively refined by the algorithm above, and load-balanced when the ratio of maximum to minimum number of cells per processor exceeds 1.5. The same sequence of refinements was carried out with different numbers of processors on the 512-processor Intel Touchstone Delta, and on a 64-processor Ncube2, in each case continuing until no more memory was available. The largest mesh, on 512 Delta processors, was 6.5 million cells.

The upper diagonal lines show the computation time for the matrix multiply, the lower lines show the time taken to synchronize the combine, which is the communication inefficiency of the matrix multiply. Notice that the communication times are 100 times less than the calculation times when the memories are full. These timing results were obtained with no optimization of point numbering, and therefore the computational kernel is probably unduly slow due to cache-miss rather than the communication being especially fast. We hope to improve these timings considerably.

It is interesting to note, however, that on the calculation part of the matrix multiply, the i860 processor in the Delta machine is only 6.5 times faster than the Ncube2 processor for this application which is bottlenecked by memory fetches rather than arithmetic speed.

13. Conclusions

Given sufficiently flexible software, an unstructured mesh is a powerful method for a computer to deal with a complex geometry because it is a framework of simple geometric entities such as tetrahedra or hexahedra. Such software is difficult to write, and even more difficult to write if it is to be robust. Distributed parallelism can produce a very fast and cheap computer, but seems to add immensely to the already difficult software costs. The goal of this paper is to show that parallelism need not add excessively to software cost while retaining the speed of the parallel machine.

This is done by separating the mesh algorithm from the parallelism, so that separate software modules may be used for the distinct tasks of mesh calculation, manipulation, communication and load balancing.

A rule of thumb with software is that most of the code is occupied with what takes a small proportion of the execution time, and a relatively small proportion of the code (the ‘computational kernel’) takes most of the time. In the case of unstructured mesh codes, the complicated software which takes little actual time is the creation, refinement and load-balancing of the mesh, and the kernel is calculation and combining/update operations. Thus in the VDB paradigm, the creation and connecting of the mesh involve perhaps surplus communication in exchange for the convenient assumption that data may be referenced by a geometric point rather than a subscript. On the other hand, extra time is taken by the VDB software during this setup to make the combining operation as efficient as possible.

14. References

1. C. Johnson, *Numerical Solutions of Partial Differential Equations by the Finite-Element Method*, Cambridge University Press, Cambridge, UK, 1987.
2. C. Koelbel and P. Mehrotra, *Compiling Global Name-space Programs for Distributed Execution*, ICASE report 90-70, Institute for Computer Science Application in Science and Engineering, Hampton VA, 1990.
3. C. Koelbel and P. Mehrotra, *Compiling Global Name-space Parallel Loops for Distributed Execution*, IEEE Transactions on Parallel and Distributed Systems, **2** (1991) 440-451.
4. J. Saltz, *Aggregation Methods for Solving Sparse Triangular Systems on Multiprocessors*, SIAM J. Sci. Stat. Comp. **11** (1990) 123-144.
5. J. Saltz, H. Berryman and J. Wu, *Runtime Compilation for Multiprocessors*, Concurrency, **3** (1991) 573-592.
6. S. Hiranandani, J. Saltz, P. Mehrotra and H. Berryman, *Performance of Hashed Cache Data Migration Schemes on Multicomputers*, J. Par. Dist. Comp. **12** (1991) 415.
7. R. D. Williams, *DIME: Distributed Irregular Mesh Environment*, Caltech Concurrent Computation Report C3P 861, 1990, also from anonymous ftp at `delilah.ccsf.caltech.edu`.
8. R. D. Williams, *Supersonic Flow in Parallel with an Unstructured Mesh*, Concurrency, **1** (1989) 51.
9. R. D. Williams, P*erformance of Dynamic Load-Balancing Algorithms for Unstructured Mesh Calculations*, Concurrency, **3** (1991) 457-481.

10. T. Mowry and A. Gupta, *Tolerating Latency Through Software-Controlled Prefetching in Shared-Memory Multiprocessors*, J. Par Dist. Comp. **12** (1991) 87.

11. K. M. Chandy and S. Taylor, *Introduction to Parallel Programming*, Jones and Bartlett, 1991.

12. *Express: An Environment for Parallel Computing*, ParaSoft Corp., Pasadena CA.

13. M.C. Rivara, *Selective Refinement/Derefinement algorithms for sequences of nested triangulations*, Int. J. Num. Meth. Engng., **28** (1989) 2889-2906.

14. M. Tam, J. M. Smith and D. J. Farber, *A Taxonomy-based Comparison of Several Distributed Shared Memory Systems*, Oper. Systems Rev **24** (1990) 40-67.

15. M. Stumm and S. Zhou, *Algorithms Implementing Distributed Shared Memory*, Computer, **23** (1990) 54.

16. W. K. Giloi, C. Hastedt, F. Shoen and W. Schroeder-Preikschat, *A Distributed Implementation of Shared Virtual Memory with Strong and Weak Coherence*, Lect. Notes in Comp. Sci., **487**, ed. A. Bode, Springer-Verlag, New York.

17. G. C. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon and D. Walker, *Solving Problems on Concurrent Processors*, Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

18. D. J. Mavriplis and A. Jameson, *Multigrid Solution of the Navier-Stokes Equations on Triangular Meshes*, AIAA J. **28** (1990) 1415.

19. I. Chern, *A Control Volume Method on an Icosahedral Grid for Numerical Integration of the Shallow Water Equations on a Sphere*, MCS Preprint MCS-P214-0291, Argonne National Lab, 1991.

20. O. Pironneau, *Finite Element Methods for Fluids*, John Wiley, Chichester, UK, 1989.

21. A. Pothen, H. D. Simon and K. P. Liou, *Partitioning Sparse Matrices with Eigenvectors of Graphs*, SIAM J. Matrix Anal., **11** (1990) 430-452.

22. D. E. Knuth, *The Art of Computer Programming*, Addison-Wesley, Reading, Massachusetts, 1973, vol. III, p. 506.

23. S. K. Godunov, *Finite Difference Methods for Numerical Computation of Discontinuous Solutions of the Equations of Fluid Dynamics*, Mat. Sb. **47** (1959) 271.

24. P. L. Roe, *Approximate Riemann Solvers, Parameter Vectors, and Difference Schemes*, J. Comp. Phys. **43** (1981) 357.

25. E. S. Oran and J. P. Boris, *Numerical Simulation of Reactive Flow*, Elsevier, New York, 1987.