Parallel Oct-tree Library User's Manual

John Salmon and Michael Warren

1 Introduction

The oct-trees described in this manual are hierachical, adaptive data structures that allow for rapid searching of spatial data. They are motivated by a desire to perform large-scale, numerically intensive algorithms such as Barnes-Hut, Greengard-Rokhlin and relatives, but it is hoped that they are general enough to be used in other pursuits. The prefix "oct" refers to the fact that eight sub-cells arise from dividing a three-dimensional cube is in half in each dimension. The oct-tree libraries described here are *NOT* restricted to three dimensions. One and two-dimensional trees are also supported, and higher dimensions could be supported with a small amount of additional code. The data structures should more properly be called $2^d -$ trees, but that is both too hard to render in text, and too hard to pronounce. Since the majority of applications to-date have been in three-dimensions, we will continue to call the data structures oct-trees, and occasionally refer to "the eight daughter cells", and allow the reader to re-interpret for his own preferred dimensionality.

Oct-trees are a generalization of binary trees, which have been part of the repertoire of computer science since the very early days. They are also closely related to k-d trees and other data structures. The goal of this library is to make oct-trees available in a potentially distributed memory, parallel computing context. The library attempts to separate the "physics" from the "computer science", and to perform all the necessary bookkeeping so that users can focus on the physics (loosely defined) of their application without worrying about the bookkeeping details of managing a large, irregular parallel data structure. Out-of-core computation is also supported on one or more processors, so extremely large data structures can be managed by moderate computational resources.

For the purposes of this library, oct-trees are defined by a collection of points inside a rectangular volume. Conceptually, the volume is subdivided hierarchically into eight sub-subvolumes (called *cells*) by dividing it in half in each dimension. Each cell is further subdivided recursively, so that an infinite hierachy results. Thus, there is one cell at level 0, eight at level 1, 64 at level 2, and more generally, 8^l at level l. A set of points and an *occupancy parameter*, m, defines a subset of this infinite hierarchy by the simple rule:

> A cell is part of the oct-tree defined by the point set $\{p_i\}$ and the occupancy parameter, m, if and only if m or more of the points in the set are contained within the cell.

This particular tree definition is motivated by the primary use of the oct-trees in this library for locating neighbors and groups of neighbors in large particle simulations. Other applications (e.g., adaptive mesh refinement) might be better served by a somewhat different different definition. It is worth noting that in our oct-trees the refinement level of neighboring cells is not restricted in any way. Two adjacent cells may be at arbitrary levels of the hierarchy, depending on the local density of points in the defining point-set.

The design of the library is guided by the observation that modern supercomputers are

space-efficient

: As processors become faster and cheaper, large simulations more frequently limited by memory constraints than by cpu constraints. Thus, the tree library tries to pack data as tightly as possible, and avoids allocating space for objects which might not be required by all applications. In particular, there are no explicit parent pointers, or 'sideways' pointers in the data structure. If such constructs are needed by an application, it is up to the application (or perhaps an intermediate library) to record them.

out-of-core

Even with careful elimination of unnecessary data, it is often impossible to fit interesting calculations into available primary storage. The ability to operate on a limited subset of an entire data set, and to move objects to and from secondary storage on demand allows for out-of-core computation. Support for out-of-core computation should not significantly impact performance when primary storage is adequate.

latency tolerant

Accesses to secondary storage, network communications and even primary storage are affected by latency – roughly the amount of time between the request and the delivery of the first byte. Latency can be tolerated by moving data in large blocks. This has the effect of amortizing latency over an even longer data transfer. Of course, the data moved must be useful. It would be counter-productive to move a large block and then to disregard most of the contents.

bandwidth friendly

Bandwidth to the network, disk and main memory are frequently the limiting factors in supercomputer performance. Bandwidth usage can be controlled by explicit software-cacheing, i.e., retaining data in primary storage after it has been moved from secondary storage or a remote processor. This strategy works if the data will be reused many times before it is no longer needed, in which case, the cost of moving it is amortized by the number of times it is used. The oct-tree library encourages the application progrm to be bandwidth friendly by providing an ordering to the elements in the tree that will result in a minimum of data transfers and a maximum of software-cache reuse.

cache-friendly

Memory bandwidth on modern micro-processors is far below that needed to feed the CPU. Hardware designers attempt to compensate by providing caches – small pieces of very fast memory which allows for very fast access (a small number of CPU cycles) to frequently used data items. Writing cache-friendly code is a difficult and uncertain task because the cache can usually only be controlled indirectly. Operations that explicitly manage the cache are typically not available in high-level languages, if they are available at all. One important rule of thumb, which is adopted by the oct-tree library is that sequential memory accesses are often much faster if the addresses are adjacent. This exploits long cache-lines, in which the CPU reads many words at once into cache whenever one of the words is accessed. The emphasis on the `sib\_group` data structure rather than individual `cells` reflects this design choice.

2 Coordinates and Keys

The oct-tree library uses a specialized coordinate system that provides a unified way to refer to cells of the tree and to locations in three-dimensional space. The coordinate system is based on long integers *keys* (typically 64-bit) which are constructed by interleaving the bits of a more conventional cartesian representation. Floating point coordinates are avoided in the internal representations used by the oct-tree library because of their susceptibility to roundoff error, underflow, etc. However, utility functions are provided which allow for easy conversion between floating point and key-based coordinates.

At level L of the tree we have a $2^L \times 2^L \times 2^L$ array of (potential) cells. We can label each of these cells by a cartesian triple of L-bit binary values. For example, the L=3 level of a two-dimensional quad-tree is shown in Figure???.

The *Morton order* key for each cell in Figure??? is obtained by interleaving the bits of the cell's cartesian coordinate and prepending a one-bit to the beginning. The Morton order keys are shown in Figure ???. Keys defined this way are very useful for identifying cells in a hierarchy. They have the following useful properties:

- The root has key 1.
- The key of the parent of any cell is equal to the key of the cell right-shifted by the dimensionality of the tree (3 for oct-trees, 2 for quad-trees, etc.)
- The keys of the children of a cell are given by the key of the cell left-shifted by the dimensionality of the tree and then filling in the low-order bits with the values from 0 to 2^d-1.

The Morton-ordered keys have another interesting property. They are almost continuous. That is, the path defined by the the sequence of keys from 0 to $2^{Ld} - 1$, stays in one region of space for a long time and occasionally undergoes long "jumps" to another part of the domain. The line in Figure?? shows the path mapped out by increasing Morton-order keys. This continuity property is actually extremely useful. Locality of references in computer memory generally leads to good cache performance and can be exploited to minimize network and disk bandwidth as well.

In fact, there is another ordering that is even better than Morton ordering for prerserving locality. The *Peano-Hilbert* ordering is actually a continuous mapping from the one-dimensional key-space into the d-dimensional cartesian space. That is, the path defined by the sequence of Peano-Hilbert keys from 0 to $2^{Ld} - 1$, always moves between adjacent cells. The Peano-Hilbert mapping also satisifies the same three criteria mentioned above for Morton ordering. The Peano-Hilbert keys for the two-dimensional space are shown in Figure???.

The tree library works with either Morton keys Peano-Hilbert keys. Conversion routines are provided that allow for conversion from cartesian integer coordinates to and from Morton and Peano-Hilbert keys.

2.1 Key Arithmetic

The header file `<key.h>` defines the type `Key_t` and the constant `KEYBITS` and contains prototypes for a the functions that operate on `Key_ts`.

Key_t typedef

The *Key_t* type behaves like a long integer (64-bit or more). It is implemented in C as a structure containing one or more `unsigned long int` values.

`int` **KEYBITS** Constant

The number of bits in this implementation of `Key_t`.

2.1.1 Comparison functions

if the corresponding predicate is true for the two keys.

`int` **KeyGT** (`Key_t` *key1*, `Key_t` *key2*) Function

Returns non-zero only if *key1* is numerically greater than *key2*.

`int` **KeyLT** (`Key_t` *key1*, `Key_t` *key2*) Function

Returns non-zero only if *key1* is numerically less than *key2*.

`int` **KeyGE** (`Key_t` *key1*, `Key_t` *key2*) Function

Returns non-zero only if *key1* is numerically greater than or equal to *key2*.

`int` **KeyLE** (`Key_t` *key1*, `Key_t` *key2*) Function

Returns non-zero only if *key1* is numerically less than or equal to *key2*.

`int` **KeyEQ** (`Key_t` *key1*, `Key_t` *key2*) Function

Returns non-zero only if *key1* is numerically equal to *key2*.

`int` **KeyEQZ** (`Key_t` *key1*, `Key_t` *key2*) Function

Returns non-zero only if *key1* is numerically equal to zero.

`int` **KeyNEQ** (`Key_t` *key1*, `Key_t` *key2*) Function

Returns non-zero only if *key1* is numerically unequal to *key2*.

`int` **KeyCmp** (`Key_t` *key1*, `Key_t` *key2*) Function

Qsort-like comparison function. Return a positive value if *key1* is greater than *key2*, a negative value if *key1* is less than *key2* and zero if the two keys are equal.

2.1.2 Arithmetic operations

`Key_t` **KeyXOR** (`Key_t` *key1*, `Key_t` *key2*) Function

Return the binary exclusive-or of *key1* with *key2*

`Key_t` **KeyAnd** (`Key_t` *key1*, `Key_t` *key2*) Function

Return the binary and of *key1* with *key2*

`Key_t` **KeyOr** (`Key_t` *key1*, `Key_t` *key2*) Function

Return the binary or of *key1* with *key2*

`Key_t` **KeyNot** (`Key_t` *key1*) Function

Return the binary negation of *key1* with *key2*

`Key_t` **KeyRshift** (`Key_t` *key*, `int` *b*) Function

Return the key obtained by logically right-shifting the bits of *key* by *b* bits. Bits that move past the right end of the key are discarded.

`Key_t` **KeyLshift** (`Key_t` *key*, `int` *b*) Function

Return the key obtained by logically left-shifting the bits of *key* by *b* bits and filling in the low-order bits with 0. Bits that extend past the end of the key are discarded.

`Key_t` **KeyLshiftFill** (`Key_t` *key*, `int` *b*) Function

Return the key obtained by logically left-shifting the bits of *key* by *b* bits and filling in the low-order bits with ones. Bits that move past the left end of the key are discarded.

`Key_t` **KeyAdd** (`Key_t` *key1*, `Key_t` *key2*) Function

Return the key obtained by numerically adding *key1* and *key2*

`Key_t` **KeySub** (`Key_t` *key1*, `Key_t` *key2*) Function

Return the key obtained by numerically subtracting *key1* and *key2*

2.1.3 Mixed Key/Integer operations

`Key_t` **KeyInt** (`int vari`) Function

Return a key whose value is the integer *i*.

`Key_t` **KeyOrInt** (`Key_t` *key*, `int i`) Function

Equivalent to KeyOr((*key*, KeyInt(*i*)), but faster.

`int` **KeyAndInt** (`Key_t` *key*, `int i`) Function

Equivalent to KeyAnd(*key*, KeyInt(*i*)), but faster.

`Key_t` **KeyAndNotInt** (`Key_t` *key*, `unsigned int` *i*) Function

Equivalent to KeyAnd(*key*, KeyNot(KeyInt(*i*))), but faster.

`Key_t` **KeyAddInt** (`Key_t` *key1*, `int` *i*) Function

Equivalent to KeyAdd(*key1*, KeyInt(*i*)))

2.1.4 Mixed Key/Floating-point operations

`Key_t` **KeyInterpolate** (`Key_t` *lokey*, `Key_t` *hikey*, `double` *frac*) Function

Return the key approximately equal to KeyAdd(*lokey*, *frac**KeySub(*hikey*, *lokey*))

2.1.5 Functions for oct-tree coordinate keys

These functions typically require a specification of the dimensionality of the underlying problem, i.e., the *ndim* argument should be 3 for oct-trees, 2 for quad-trees, etc.

`int` **TreeLevel** (`Key_t` *key*, `int` *ndim*) Function

Return the level in the tree that described by *key*. I.e., the value of L such that: KeyEQ(KeyRshift(*key*, *ndim**L), KeyInt(1)) Undefined if there is no such L.

`int` **isBodyKey** (`Key_t` *key*, `int` *ndim*) Function

Return non-zero if *key* contains a bit indicating that it is a body-key. Equivalent to: TreeLevel(*key*, *ndim*) == ((KEYBITS-1)/*ndim*)

`int` **CommonLev** (`Key_t` *key1*, `Key_t` *key2*, `int ndim`) Function

`int` **KeyContained** (`Key_t outer`, `Key_t key`, `int ndim`) Function

`int` **KeyOverlap** (`Key_t` *key1*, `Key_t` *key2*, `int ndim`) Function

2.1.6 Miscellaneous operations

`char` ***PrintKey** (`Key_t` *key*) Function

Return a pointer to a static character string containing an ascii (octal) representation of the key. The string will be overwritten on subsequent calls.

2.2 Key Conversions

All key conversion routines (including the definition of `bbox_t` are in the header file `<keycvt.h>`. Key Conversion functions are logically layered on top of the Key Arithmetic functions. That is, they use the functions provided by `<key.h>` to perform all necessary manipulations of `Key_t` objects. There is nothing special about these conversion functions. They provide a convenient way of converting from floating-point or integer coordinate systems to `Key_ts`, but application codes are free to generate `Key_ts` objects in any way they wish.

2.2.1 Converting between keys and from Cartesian integers

`Key_t` **KeyFromInts** (`unsigned int` **xp*, `int` *ndim*, `int` *nbits*) Function

Treat *xp* as an array of *ndim* integers with *nbits* each. Generate a Morton-order key by interleaving the low-order *nbits* of *xp[i]* and prepending a one bit in the *ndim*nbits* position.

`int` **IntsFromKey** (`Key_t` *key*, `unsigned int` **ip*, `int` *ndim*) Function

Assume that *key* represents a Morton-order cell key and extract the corresponding cartesian integer keys into the array *ip*. Return the level associated with *key*, i.e., the value of *nbits* in the corresponding `KeyFromInts` call.

`Key_t` **PHKeyFromInts** (`unsigned int` **xp*, `int` *ndim*, `unsigned int` *nbits*) Function

Similar to `KeyFromInts`, but generate a Peano-Hilbert key instead.

int **IntsFromPHKey** (Key_t *key*, unsigned int **ip*, int *ndim*) Function

Similar to KeyFromInts, but input is a Peano-Hilbert key instead.

2.2.2 Bounding boxes

Conversion from a Cartesian floating point coordinate system to to Key_ts requires an intermediate conversion to a Cartesian integer system. One way to define such conversions is with respect to a *bounding box* in the floating point system. Integer coordinates are defined by dividing the bounding box into INT_MAX equal intervals in each cartesian direction. Typically, the application program will define a UniverseBbox and do all float to key conversions with respect to it.

tbbox typedef

The tbbox is a structure with the following members:

int ndim
: The dimensionality of the the cartesian space. Must be 5 or less.

float rmin[ndim]
: The coordinates of the lower corner of the bounding box.

float sz[ndim]
: The extent of the box.

These members are exposed, and may be manipulated by application code.

Several functions are defined to support common operations on tbboxs.

void **CornersBbox** (const tbbox **bbox*, float **min*, float **max*) Function

Return the minimum and maximum extent of *bbox* in *min* and *max* respectively.

void **CenterBbox** (const tbbox **bb*, float **center*, float **sz*) Function

The array of floats *center* is set to the center of *bb*, and the array of floats *sz* is set to the extent of *bb* in each cartesian dimennsion.

void **TightBbox** (const float **pstart*, int *nobj*, int *pstride*, int *ndim*, tbbox **bb*) Function

The result, *bb* is set to a bounding box that encloses the list of positions beginning at *pstart*. Positions are assumed to be stored as groups of *ndim* floats, spaced by *pstride* bytes from beginning-to-beginning. *Note that the stride is measured in bytes not in floats.* The result is a very *tight* bounding box, which can sometimes be problematical with roundoff, etc.

void **CubeBbox** (tbbox **bb*) Function

Make *bb* a cube (equal extent in all dimensions) by expanding the smaller dimensions.

void **InflateBbox** (tbbox **bb*, float *factor*) Function

Increase the linear dimension of *bb* by *factor* in all dimensions `int` **ContainsBbox** (`const tbbox *`*bb1*, `const tbbox *`*bb2*) Function

Return non-zero if and only if *bb1* completely contains *bb2*.

`void` **UnionBbox** (`const tbbox *`*bb1*, `const tbbox *`*bb2*, `tbbox *`*bbu*) Function

Construct *bbu*, the smallest bounding box that contains the union of *bb1* and *bb2*

2.2.3 Converting between cartesian integers and floats

Two functions are provided to convert between cartesian integer and floating point representations. Applications may use these in conjunction with `KeyFromInts` and `IntsFromKey`, or they may wish to use the more succinct (but less flexible) `GenerateKeys` instead.

`void` **IntsFromFloats** (`const float *`*x*, `unsigned int *`*ix*, `const tbbox *`*bb*, `int` *nbits*) Function

`void` **FloatsFromInts** (`const int *`*ix*, `float *`*x*, `const tbbox *`*bb*, `int` *nbits*) Function

2.2.4 Converting between keys and floating point

KeyOrder_e enum

The *KeyOrder_e* enum is an argument to `CellBBFromKey` and `GenerateKeys`, which selects between the two supported methods of ordering the key space. Accordingly, it may take on one of the two values:

`MORTON_ORDER`

`PEANO_ORDER`

`Key_t` **KeyFromFloats(const** `float *`*coords*, `const tbbox *`*universe*, `int` *nbits*, `enum KeyOrder_e` *ordering*) Function

Convert the floating point *coords* to a key by first calling `IntsFromFloats` and then the appropriate one of `KeyFromInts` or `KeyFromPHInts`, depending on the value of *ordering*. Return `KeyInt(0x0)` if thef coordinates are outside the bounding box specified by *universe*.

`void` **CellBBFromKey** (`Key_t` *key*, `tbbox *`*Universe*, `tbbox *`*cellbb*, `enum KeyOrder_e` *key_ordering*); Function

The *Universe* is treated as the bounding box that encloses an entire tree and *key* is the key of a cell in the tree, assuming the given *key_ordering*. The result *cellbb*, is the bounding box of the cell in floating point coordinates. Thus, this function provides the mechanism for converting from `Key_t` to a cartesian floating point coordinate system.

3 Paging and Out-of-core pointers

DISCUSSION!!! motivation, etc. Swapping policy, PgId, lack of coherence.

Only one paging system may be operational in a program. It is neither reentrant nor thread-safe.

3.1 Initialization and Termination

`void` **PgInit** (`int` *lgpgsz*, `int` *wsetpages*, `const char *`*filename*) Function

> Initialize the out-of-core paging system. It will use pages of size 2^{lgpgsz}. Space in primary storage for a working set of *wsetpages* will be allocated and devoted to the paging system. The file specified by *filename* will be opened for read/write and will be used to store pages that do not fit in the primary working set. If *filename* is NULL, a temporary filename will be obtained by calling `tmpnam`. In this case, `unlink` is called immediately after the file is opened. On Unix systems, this makes the file invisible to directory searches, but it remains open until processes that have it open (e.g., the process that called `PgInit`) terminate.

`void` **PgTerm** (`void`) Function

> Terminate the paging system and free the space allocated to its working set.

3.2 Allocating and deallocating PgIds

There is only "namespace" for all the pages used by a program - no matter whether they are used by libraries, user code, etc. `PgAlloc` and `PgFree` provide a mechanism whereby different subsystems can coordinate their allocation and deallocation of `PgIds` without interference.

`PgId` **PgAlloc** (`void`) Function

> Return a `PgId` that refers to an unused page in the calling process.

`void` **PgFree** (`Pgid` *pgid*) Function

> Free the *pgid*. It is an error if *pgid* was not previously allocated by `PgAlloc` in the calling process, or if it has already been freed.

3.3 Referencing and Unreferencing pages

PgHndl typedef

> A *PgHndl* is an opaque type returned by `PgRef`, and passed as an argument to `PgUnref`, `PgRefef` and `PgAddr`.

`PgHndl` **PgRef** (`PgId` *pgid*) Function

> Cause the page named by *pgid* to be brought into the working set (if it is not already there), and return a `PgHndl` object which can be used for subsequent accesses to the page. Increment the reference count associated with the page.

Every effort has been made to optimize the (hopefully) common case in which the page is alredy in memory, in which case, `PgRef` is very fast. If the page is not in memory, *PgRef* blocks until either I/O system (either disk or network) delivers the page. This may be up to several milliseconds for a page on disk. If the page is on a remote processor, it will not be returned until the remote processor calls `PagePoll`, or until it also blocks on a non-local `PgRef`.

`void *` **PgAddr** (`PgHndl` *hndl*) Function

Return a pointer to the beginning of a page. The *hndl* argument must have been returned by a previous `PgRef` call. A page will not move in primary storage until `PgUnref` is called, so the pointer returned by `PgAddr` is valid (at least) until `PgUnref` is called. `PgAddr` does nothing more than return a field from the opaque `PgHndl` structure. It is inlined where possible.

`void` **PgReref** (`PgHndl` *hndl*) Function

Increment the reference count on the page referred to by *hndl*. Use *PgReref* if you already have a `PgHndl`, and for some reason we want to guarantee that it stays around, but cannot avoid calling `PgUnref` (e.g., you're inside a callback in a tree traversal).

PgHygeine_e enum

An enumeration type with two possible values. Passed as an argument to `PgUnref`.

`PG_CLEAN`

`PG_DIRTY`

`void` **PgUnref** (`PgHndl` *hndl*, `enum PgHygiene_e` *dirtied*); Function

Unreference the page referred to by *hndl*. That is, decrease it's reference count by one. As long as the reference count is greater than 0, the page will not move in memory. That is, any pointer returned by `PgAddr` will remain valid at least until `PgUnref` is called. The application tells the library whether the contents of the page were modified via the *dirtied* argument. When the space in the primary working set is reclaimed, the library checks whether any of the accumulated `PgUnrefs` specified *PG_DIRTY*, and if so, the new data is written back to disk storage. Otherwise, the writeback is unnecessary and is skipped.

`void` **PgFlushRemote** (`void`) Function

Flush (i.e., force a copy to be read from the network when next requsted) all pages on remote processors. More fine-grained control will be provided in a subsequent version.

3.4 Diagnostic information

`char *` **PrintPgId** (`PgId pgid`) Function

Return a pointer to a static character string containing a human-readable representation of the *PgId*. The string will be overwritten on subsequent calls.

`int` **PgStatus** (`PgId` *pgid*) Function

Return information about about the status of *pgid*. The value returned is the bitwise OR of the following symbolic constants

`PG_STAT_REMOTE`
: set if the master copy of the page resides on a remote processor.

`PG_STAT_INMEM`
: set if the page is currently in memory.

`PG_STAT_REFED`
: set if one or more references are outstanding for the page.

`PG_STAT_DIRTY`
: set if one or more `PgUnrefs` have declared the page to be dirty since it was last written to disk.

The paging library maintains several counters which record cumulative information about the status of its internal data structures. In a future release, this set will be more complete, and the values will be fields in a `PgInfo` structure.

`int` **PgSize** Variable

How big are the pages, equal to 2***lgpgsz*, where *lgpgsz* was the argument to `PgInit`.

`int` **PgNrefcnt** Variable

The number of pages currently with non-zero reference counts.

`int` **PgReuse** Variable

The number of times `PgRef` found its target page already in memory.

`int` **PgInUse** Variable

How many pages are currently allocated *by this process*.

`int` **PgHwm** Variable

The high-water mark of allocated pages, i.e., maximum number of pages ever simultaneously allocated *by this process*.

3.5 Polling

The paging library is *polled*. It does not start a separate thread to respond to remote requests. Therefore, if your code is must respond to remote requests, then it must occasionally poll for them. These two routines do that.

`void` **PagePoll** (`void`) Function

Poll once for requests for pages residing on this processor. If no requests are pending, return immediately. If there are pending requests, then continue polling until the queue is drained.

`void` **PagePollTillDone(void)** Function

Poll for requests until all processors have called `PagePollTillDone`. This function acts as a barrier, but the caller continues to respond to remote page requests while waiting for their peers to reach the barrier.

3.6 Out-of-core pointers

oocptr typedef

An out-of-core pointer `oocptr` is a structure containing two fields. Its allows one to store information about out-of-core objects that are smaller than an individual page. As with `void` pointers in C, the intepretation of the data is left to the caller. The fields are:

`PgId pgid` The pgid on which the object is permanently stored.

`unsigned int offset`
The objects' offset from the beginning of the page.

`void *` **OOCFollow** (`oocptr` *oocp*, `PgHndl *`*hndlp*) Function

A helper function that dereferences the out-of-core pointer *oocp* and returns a bona-fide C pointer to an in-memory copy. It also sets the value of **hndlp* to the `PgHndl` that should be passed to `PgUnref` when the pointer is no longer needed. OOCFollow is implemented (where possible) as an inline function equivalent to the following:

```
INLINE void *OOCFollow(oocptr ooc, PgHndl *hndl){
  *hndl=PgRef(ooc.pgid);
  return (void *)( ((char *)PgAddr(*hndl)) + ooc.offset );
}
```

4 Tree construction

Since an oct-tree is defined by a point-set, it is natural to construct a tree by specifying a point-set. Points are added one at a time, and each point may be associated with arbitrary user-data when it is inserted into the tree. Exactly which cells exist in the tree are implicit in this scheme. That is, the application program does not explicitly identify the cells of the tree. The cells are constructed by the library. Whenever a cell is constructed callback functions (registered when the tree was initialized) are invoked so that the application can associate data with each cell in the tree.

4.1 Data Types

This section describes programmer-visible data types declared in `<tree17.h>`

tree_t typedef

An opaque data type used in tree construction. There are no publicly visible fields. A `tree_t` is an argument to all of the tree construction functions.

sib_group typedef

A `sib_group` represents a cubical (or square in two dimensions, etc.) volume of space.

tcell typedef

The following function pointer typedefs are used as for callback routines that the user must supply to the tree building functions.

makeIntermediateCell_t typedef

A typedef of a function pointer with prototype: size_t (*makeIntermediate-Cell_t) (Key_t *key*, int *ntbody*, const _tbody ***tbodyptrs*, int *ncell*, const _cellDataIntermediate *intermediates*, _cellDataIntermediate *result*)

makeFinalCell_t typedef

void * (*makeFinalCell_t)(_cellDataFinal **to*, const _cellDataIntermediate **from*, size_t *sz*);

The following 'pseudo-typedefs' are provided.

_tbody pseudo-typedef

_cellDataIntermediate pseudo-typedef

_cellDataFinal pseudo-typedef

The pseudo-typedefs are actually '`#define`' macro definitions of the form:

```
#ifndef _tbody
#define _tbody void
#endif
```

These are used as argument types in callback functions passed to the tree building routines. They exist to work around an annoying feature of ANSI C compilers with strict type checking. If the programmer inserts his own #definition for a pseudo-typedef *ahead of* the `#include "tree17.h"` then it is possible to declare the callback functions with correct, user-specific arguments. For example, one can write:

```
#define _tbody mybody
#define _cellDataIntermediate cofm
#include "tree17.h"

size_t computeCofm(Key_t, int, const mybody **, int, const cofm **, cofm *);
```

and `computeCofm` can be assigned to a `makeIntermediateCell_t` without generating spurious warnings about mismatched types. Without the pre-definition of `_tbody` and `_cellDataIntermediate`, warnings would be generated because a function that takes a `cofm *` argument is not strictly equivalent to a function that takes a `void *` argument.

4.2 Tree Building Functions

This section describes the functions available for building oct-trees.

`void` **TreeBuildStart** (`tree_t` **tp*, `int` *ndim*, `int` *max_occupancy*, `BuildFuncs_t` **bfuncs*) — Function

`void` **TreeBuildAdd** (`tree_t` **tp*, `Key_t key`, `_tbody` **tbody*, `size_t` *tbodysz*) — Function

`void` **TreeBuildFinish** (`tree_t` **tp*) — Function

`void` **TreeFree** (`tree_t` **tp*) — Function

5 Tree Traversal

Type Index

Function Index

Variable Index

Concept Index

(Index is nonexistent)

Table of Contents