Zcode Documentation

Release

Thierry Dumont

Sep 13, 2017

Contents:

CHAPTER 1

Node

1.1 Node definition

template <std::size_t *Dim*, typename *value_type* = std::size_t>
struct definitions

Public Static Attributes

const int **size** = sizeof(value_type)*8
size of *Node* in digits.

const int **nbfreebits** = 5
number of freebits used for MR.

const int **nblevelbits** = max_level(dim, *nbfreebits*, *size*)
number of digit for level.

const int **nlevels** = (*size*-*nbfreebits*-*nblevelbits*)/dim
max number of tree levels.

const int **treetype** = 1<<dim
bin (1d), quad(2d), octo (3d) trees.

const int **nfaces** = 2*dim
number of faces of each element

const int **nbbef** = 1<<(dim-1)
number of elements when you refine once on each face (1d: 1, 2d: 2, 3d: 4).

const int **nbd** = ipow(2, dim) << dim
number max of bound. elements (1d: 2, 2d: 12, 3d: 56).

const int **levelshift** = *size*-*nblevelbits*
amount of right shift needed to get level in the tree.

constexpr value_type **levelone** = one<<*levelshift*
first digit marking levels:

const value_type **levelmask** = AllSet2One<dim, *nblevelbits*, value_type>::value
mask for extracting level:

const value_type **levelzone** = (*levelmask*)<<*levelshift*
mask the part used to store level

constexpr value_type **maskpos** = AllSet2One<dim, dim**nlevels*, value_type>::value
mask for what is used for position:

const int **nbneighb** = ipow(3, dim)
how many neighbors for one *Node* (including itself).

const value_type **Xbit** = one<<(dim**nlevels*-1)
bits used in nodes computations (Zbit not used -2d problem!-)

leftmost digit for x position

const value_type **Ybit** = (dim==1)? 0: (*Xbit*>>1)
leftmost digit for y position

const value_type **Zbit** = (dim==3)? (*Xbit*>>2): 0
! not used (for compatibility with the 3-d case).

const value_type **IntOne** = one
! 1!

const value_type **AllExceptVoidbit** = allone - voidbit
mask for extracting all, but the void bit:

1.2 Node class

template <std::size_t *Dim*, typename *Value* = std::size_t>
struct Node
Inherits from *definitions< Dim, Value >*

Public Functions

bool **is_max**(direction *d*) **const**
test if the node as max coordinate

Parameters

- d: the direction.

bool **is_min**(direction *d*) **const**
test if the node as min coordinate

Parameters

- d: direction.

std::size_t **lastlevel**() **const**
get the last level digits of a node in an int (flushed right).

Note this can be applied to hashed and non hashed Nodes.

Parameters

- `node`:

bool **`is_minimal`**`()` **`const`**
is a *Node* minimal (ie has minimal abscissa) in his set of Brothers?

void **`setTags`**(*Node* &*n*) **`const`**
set the tag part of a *Node*

Note we do not check V.

Parameters

- `N`: pointer to the *Node*.
- `V`: tag value

Node **`hash`**`()` **`const`**
return the hash code for nodes.

Note we do not test if x is already hashed, except if DEBUG is set.

Parameters

- `x`: *Node*

Node **`unhash`**`()` **`const`**
For a given hasehd representation of a *Node*, we return the non hashed representation.

Parameters

- `x`: *Node*

bool **`isHashed`**`()` **`const`**
Is a node hashed?

1.3 Node family

Functions

`template` <std::size_t *dim*, **`typename`** value_type>
Node<dim, value_type> **`firstSon`**(*Node*<dim, value_type> **`const`** &*node*)
return the son of a *Node* which has the smallest absissa (ie, the 1rst one in the son's brotherhood.

Note we return a non hashed *Node*.

Parameters

- `u`: *Node*.

`template` **`<typename`** Node_type>
Node_type **`lastSon`**(Node_type **`const`** &*node*)
return the son of a *Node* which has the largestlest absissa (ie, the last one in the son's brotherhood.

Note we return a non hashed *Node*.

Parameters

- `u`: *Node*.

template <typename Node_type**>**
Node_type **father**(Node_type **const** &*node*)
compute the father of a node

Parameters

- `node`: node.

template <typename Node_type**>**
bool **isAncestor**(Node_type *A*, Node_type *X*)
test if a *Node* A is an ancestor of a *Node* X.

Note each *Node* is its own ancestor, too.

Parameters

- `A`: *Node*
- `X`: *Node*

template <typename Node_type**>**
bool **shareAncestor**(Node_type *A*, Node_type *B*, std::size_t *lv*)
Do 2 Nodes share the same ancestor of a given level ?

Parameters

- `A`: *Node*
- `B`: *Node*
- `lv`: the level.

template <typename Node_type, **typename** Node_array**>**
void **brothers_impl**(Node_type **const** &*node*, Node_array &*Brothers*, std::integral_constant<std::size_t, 1>)
template <typename Node_type, **typename** Node_array**>**
void **brothers_impl**(Node_type **const** &*node*, Node_array &*Brothers*, std::integral_constant<std::size_t, 2>)
template <typename Node_type, **typename** Node_array**>**
void **brothers_impl**(Node_type **const** &*node*, Node_array &*Brothers*, std::integral_constant<std::size_t, 3>)
template <typename Node_type, **typename** Node_array**>**
void **brothers**(Node_type **const** &*node*, Node_array &*Brothers*)
Make the list of the brothers of a minimal node in a brothers set.

Note node *must* be minimal in his brothers set. *NOT TESTED*, except if DEBUG is set.

Note Brothers[0] == node.

Note the output array Brothers is ordered.

Parameters

- `node`: the node for which we build the list.
- `Brothers`: the list of brothers.

1.4 Node neighbors

Functions

template <**typename** Node_type, std::size_t *nx*>
void **neighbors**(Node_type **const** &*node*, std::array<Node_type, nx> &*node_array*, std::array<int, nx> **const** &*stencilx*)
template <**typename** Node_type, std::size_t *nx*, std::size_t *ny*>
void **neighbors**(Node_type **const** &*node*, std::array<Node_type, nx *ny> &*node_array*, std::array<int, nx> **const** &*stencilx*, std::array<int, ny> **const** &*stencily*)
template <**typename** Node_type, std::size_t *ns*>
void **neighbors**(Node_type **const** &*node*, std::array<Node_type, ns> &*node_array*, std::array<std::array<int, 2>, ns> **const** &*stencil*)
template <**typename** Node_type, std::size_t *ns*>
void **neighbors**(Node_type **const** &*node*, std::array<Node_type, ns> &*node_array*, std::array<std::array<int, 3>, ns> **const** &*stencil*)
template <**typename** Node_type, std::size_t *nx*, std::size_t *ny*, std::size_t *nz*>
void **neighbors**(Node_type **const** &*node*, std::array<Node_type, nx *ny *nz> &*node_array*, std::array<int, nx> **const** &*stencilx*, std::array<int, ny> **const** &*stencily*, std::array<int, nz> **const** &*stencilz*)
template <std::size_t *stencil*, **typename** Node_type, **typename** Node_array>
void **boxNeighbors_impl**(Node_type **const** &*n*, Node_array &*node_array*, std::integral_constant<std::size_t, 1>)
template <std::size_t *stencil*, **typename** Node_type, **typename** Node_array>
void **boxNeighbors_impl**(Node_type **const** &*n*, Node_array &*node_array*, std::integral_constant<std::size_t, 2>)
template <std::size_t *stencil*, **typename** Node_type, **typename** Node_array>
void **boxNeighbors_impl**(Node_type **const** &*n*, Node_array &*node_array*, std::integral_constant<std::size_t, 3>)
template <std::size_t *stencil*, **typename** Node_type, **typename** Node_array>
void **boxNeighbors**(Node_type **const** &*n*, Node_array &*node_array*)
find a potential neighbor, depending on the position of u.

Parameters

- `u`: node.
- `P[]`: returned list(vector)

template <int *stencil*, **typename** Node_type, **typename** Node_array>
void **starNeighbors_impl**(Node_type **const** &*n*, Node_array &*node_array*, std::integral_constant<std::size_t, 1>)
template <int *stencil*, **typename** Node_type, **typename** Node_array>
void **starNeighbors_impl**(Node_type **const** &*n*, Node_array &*node_array*, std::integral_constant<std::size_t, 2>)
template <int *stencil*, **typename** Node_type, **typename** Node_array>
void **starNeighbors_impl**(Node_type **const** &*n*, Node_array &*node_array*, std::integral_constant<std::size_t, 3>)
template <int *stencil*, **typename** Node_type, **typename** Node_array>
void **starNeighbors**(Node_type **const** &*n*, Node_array &*node_array*)
find a potential neighbor, depending on the position of u.

Parameters

- `u`: node.
- `P[]`: returned list(vector)

1.5 Node refinment

Functions

template <**typename** Node_type, **typename** Node_array>
void **refine**(Node_type **const** &*n*, Node_array &*refined*)

refine n.

Results in refined[0: treetype-1].

Parameters

- `n`: node.
- `refined[]`: the refined nodes.

CHAPTER 2

slot

template <std::size_t *dim*, **typename** node_value_type = std::size_t>
struct slot

slot stucture, used to store Nodes with a hash code in an interval.

slot structures, store a set of Nodes.

Inherits from std::vector< Node< dim, node_value_type > >

Public Functions

auto **find**(node_type **const** &*node*)

find a *Node*.

Note we *do not check* if x is hashed.

Parameters

- x: *Node hashed*

void **put**(node_type *x*)

put a *Node* at the end.

Parameters

- x: a *hashed* node (not checked).

template <**typename** container>
void **put**(container **const** &*x*)

put a vector of *Node*'s at the end.

Parameters

- x: vector of *hashed* nodes (not checked).

int **lastPos**() **const**
position of the last entered *Node* in v[]

void **compress**(node_type *N* = node_type::voidbit)
compress: ie, supress void Nodes with a given value

Parameters

- N: supress nodes for which N&node !=0

void **compressany**()
compress all marked Nodes.

ie, supress Nodes with any value in the FreeBitsPart or marked as void

void **compress**(node_type *N*, node_type *M*)
compress: ie, supress void Nodes with given values

Note a *Node* K are supressed iff K&N==N *or* K&M==M

Parameters

- N: test value
- M: test value

void **setMark**(node_type *N*)
mark the slot with some value.

Parameters

- N: associated value.

unsigned char **getMark**()
get the mark tag.

void **unsetMark**(node_type *N*)
suppress a given mark

Note throw an exception if not marked "mark".

Parameters

- N: the mark

bool **markedOther**(node_type *mark*)
test if the slot has been marked by an other mark as "mark"

Parameters

- mark: for the test.

bool **hasvoidNodes**() **const**
does this slot contains void Nodes ?

void **sethasvoidNodes**()
mark the slot as containing void Nodes.

void **And**(node_type *N*)
make a logical "and" of all Nodes with a given value.

Parameters

- `N`: the value.

void **setTag**(node_type *N*)
Tag, ie add some value to all Nodes.

Note a "and" with the value must be zero. Not tested if DEBUG is not set. We do not want to set a value to Nodes, but to tag them.

Parameters

- `N`: the value.

void **empty**()
empty the slot. Do not change s1 and s2, do not deallocate space.

auto **cut**(std::size_t **const** *nc*)
cut the slot in nc slots.

Note sizes of the resulting slots are not garanted to be equal.

Parameters

- `nc`: number of slots
- `s`: result. Array of nc slot*.

auto **cutBefore**(std::size_t *pos*, node_type *s2new*)
cut this slot in 2 slots, at position pos, and then shrink it.

the returned slot is the first part containing v[0,pos[

Note we do not check that pos is correct, except if DEBUG is set.

Note for s2new: position part only; tested only if DEBUG set.

Parameters

- `pos`:
- `s2new`: value for s2 of the *new* slot, and s1 of this slot.

void **fusion**(**const** *slot* &*sl*)
fusion this slot with slot sl

Parameters

- `sl`: slot.

void **sort**()
sort by hash function.

bool **cutdown**(std::size_t *lim* = 2)
reallocate to reduce size;

Note return True iff slot is reduced.

Parameters

- `lim`: we reduce size if allocsize/size>= lim

void **forgetFreeBits**()
suppress all bits used to mark something (except voidbit).

void **uniq**()
suppress ex-aequo.

Note slot must be sorted (tested only if DEBUG is set).

bool **testWellFormed**(bool *throwexept* = true) **const**
test if all nodes have their abscissa between s1 and s2.

Parameters

- `throwexept`: throw an exception if true.

bool **exaequo**() **const**
look for ex-aequo

void **setStartRank**(std::size_t *r*)
set startrank

Parameters

- `r`:

int **Slotrank**() **const**
return slotrank.

void **setSlotrank**(std::size_t *r*)
set the slot rank

Parameters

- `r`:

void **dump**(std::ofstream &*f*)
Write slot to a file, already open.

_param f the file.

void **restore**(std::ifstream &*f*)
restore a slot from a dump.

Parameters

- `file`: the file to restore from.

CHAPTER 3

slotCollection

template <std::size_t *dim*, **typename** node_value_type>
struct slotCollection

Inherits from std::vector< std::shared_ptr< slot< dim, node_value_type > > >

Public Functions

void **copyInArray**(std::vector<node_type> &*array*)

An other "copy init".

Here we put the Nodes of each slot in a global array. This is supposed to reduce the number of allocations. We use this to store a local copy of a SlotCollection.

Parameters

- `SC`: slot collection to be copyed.
- `G`: global array.

void **clone**(**const** *slotCollection* &*C*)

make a "clone", ie copy all, but not the Nodes!

bool **inInterval**(node_type *s1*, node_type *s2*, node_type *x*) **const**

Is a *Node* abscissa in the interval [s1,s2[?

Note x must be *not* hashed.

Parameters

- `s1`:
- `s2`:
- `x`: *Node* to check.

void **insert**(node_type *x*, Cache<dim, node_value_type> &*cache*)

Store one node using a cache.

Parameters

- `x`: the node to be inserted.
- `cache`: an external Cache.

void **insert**(node_type *x*)
Store one node.

Parameters

- `x`: the node to be inserted.

std::size_t **nbNodes**() **const**
number of Nodes stored.

level_count_type **nbNodesByLevel**() **const**
Returns the number of nodes by level.

auto **ubound**(node_type *x*) **const**
return a pointer to a slot which *possibly* contains a *Node*.

Parameters

- `x`: *Node not hashed*

auto **ubound_hashed**(node_type *x*) **const**
return a pointer to a slot which *possibly* contains a *Node*.

Parameters

- `x`: *Node hashed*

std::size_t **findSlot**(node_type *N*, std::size_t *left*, std::size_t *right*) **const**
Given a *Node*, find his slot.

Note we search in range [left,right] of the SlotCollection

Note returns the rank of the slot which contains N

Parameters

- `N`: the *Node*
- `left`:
- `right`:

std::size_t **count**(node_type *x*, Cache<dim, node_value_type> &*cache*) **const**
Test if a *Node* exists within this *slotCollection*, using a cache.

Return the number of corresponding found node (either 0 or 1).

Note we do not directly check if the *Node* is really non hashed, but this is checked in "xh=hash(x)".

Parameters

- `x`: *Node* non hashed.
- `cach`: cache updated.

std::size_t **count**(node_type *x*) **const**
Test if a *Node* exists within this *slotCollection*.

Return the number of corresponding found node (either 0 or 1).

Note we do not directly check if the *Node* is really non hashed, but this is checked in "xh=hash(x)".

Parameters

- `x`: *Node* non hashed.

void **forgetFreeBits**()
remove all free bits from all nodes.

void **compress**(node_type *val* = node_type::voidbit)
suppress void Nodes, if any.

update the count of leaves.

void **clear**()
empty all the slots.

void **makeExtern**(SetNode &*setN*)
make a copy (in a set) of the Nodes.

Parameters

- `setN`: the set.

void **finalize**()
finalize: compute cumulsize (to allow rank function to work), and maximum size of slots;

void **relink**()
compute ranks, . . .

std::size_t **maxSlotSize**() **const**
return maximum size of slots.

Public Members

std::size_t **slot_max_size**
size of slot which triggers decomposition of a slot.

std::size_t **slot_min_size**
size of slot which triggers fusion of two slots.

struct ltNode
define order on the Nodes.

We use the Peano-Hilbert curve for indexation, and thus, we must suppress all what is not position.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Index

D

N

S