From Numerical Cosmology to Efficient Bit Abstractions for the Standard Library

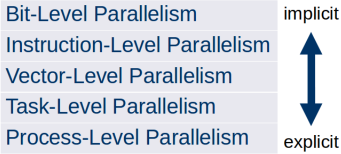
Vincent Reverdy

Department of Astronomy
University of Illinois at Urbana-Champaign (UIUC)

September 21, 2016

Prologue

What is this talk about?



Levels of parallelism from Bryce's talk

Alternative titles

- Bit-level parallelism (manipulating bit sequences by chunks)
- Abstracting bits
- On bit manipulation algorithms for the standard library
- What's wrong with you std::vector<bool>?
- Reinventing std::bitset and std::vector<bool>
- Counting bits 100× faster than with std::vector<bool>
- Playing with 0 and 1

Using bit utilities in less than 2 minutes (1/2)

Step 1: Clone and download

- With GIT: `git clone https://github.com/vreverdy/bit.git`
- Without GIT: go on the page `https://github.com/vreverdy/bit`, click on `Clone or download`→`Download ZIP`, download, and unzip it.

Step 2: Run a minimal test case

```cpp
#include <iostream>
#include "./bit/cpp/bit.hpp" // Your path to bit.hpp
using namespace bit;

int main(int argc, char* argv[]) {
  using uint_t = unsigned int;
  uint_t n = 42;
  auto first = bit_iterator<uint_t*>(&n);
  auto last = bit_iterator<uint_t*>(&n + 1);
  for (; first != last; ++first) std::cout<<*first;
  std::cout<<std::endl;
  return 0;
}

// Compilation with GCC: g++ -std=c++14 -pedantic -O3 main.cpp -o main
// Output: 01010100000000000000000000000000
```

Compile it and run it, it should display the bits of `n` from the LSB to the MSB.

Using bit utilities in less than 2 minutes (2/2)

Step 3: And that's it

You are all set...

Contact and links

- GitHub: `https://github.com/vreverdy/bit`
- Contact: `vince.rev@gmail.com`
- ISO C++ proposal (description): `P0237R0`
- ISO C++ proposal (last wording): `P0237R2` (in progress)
- Collaborations are very welcome! Same for comments! Same for benchmarks!

Chapter I: An astrophysical motivation

Once upon a time…

…in a galaxy far far away…

...on a small piece of rock...

...wandering aimlessly in a Vast Universe...

...a team of astrophysicists was wondering
about the nature of life, the Universe, and everything.

Because they knew some maths, some physics some computer science, and some programming,

typename std::enable_if<std::is_integer<T>::value>::type

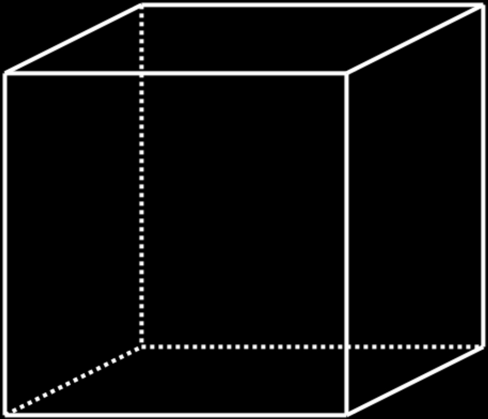
template <class T, std::size_t N>

$$G\mu\nu = \frac{8\pi G}{c^4} T\mu\nu$$

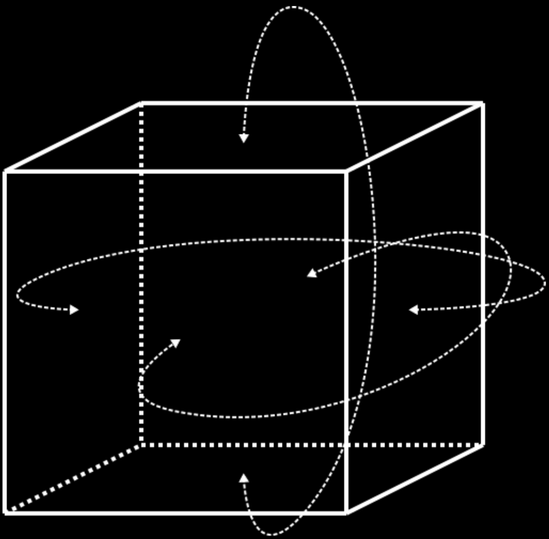
$$\nabla_r^2 \Phi = 4\pi G\rho$$

$$\dot{\rho} = -3H\left(\rho + \frac{p}{c^2}\right)$$

they decided to design a code that could answer their (meta)physical questions.

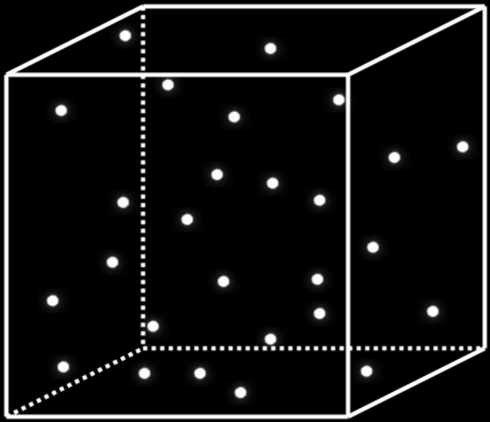


They said: "Let's take an enormous box...



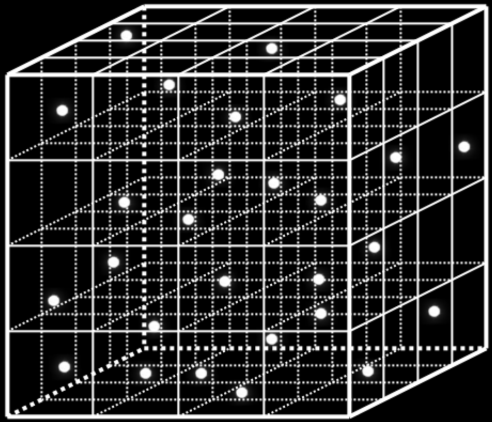
...with periodic boundary conditions...

(right/left, top/bottom, and front/back connected together)

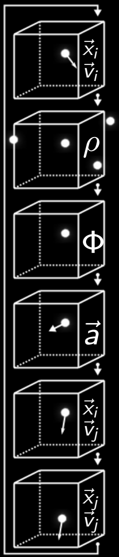


...and let's fill that enormous box with particles weighing the mass of millions or billions of suns...

(note: yes, that's kind of huge)



Now, divide the box in cells using a regular grid and apply the following recipe:



1) For each cell c containing particles with position $\vec{x}$ and velocity $\vec{v}$

2) Interpolate the density ρ in the cell c depending on surrounding particles

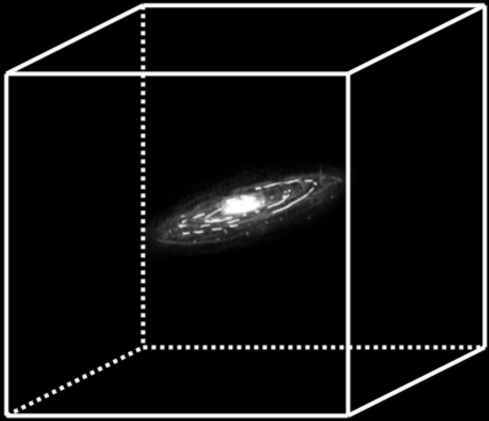
3) From ρ, compute the gravitational potential Φ

4) From Φ, interpolate back the acceleration $\vec{a}$ at the position of particles

5) From $\vec{a}$, compute the new speed $\vec{v}$ of each particle

6) From $\vec{v}$, compute the new position $\vec{x}$ of each particle

7) Restart at 1) with the updated position $\vec{x}$ and speed $\vec{v}$

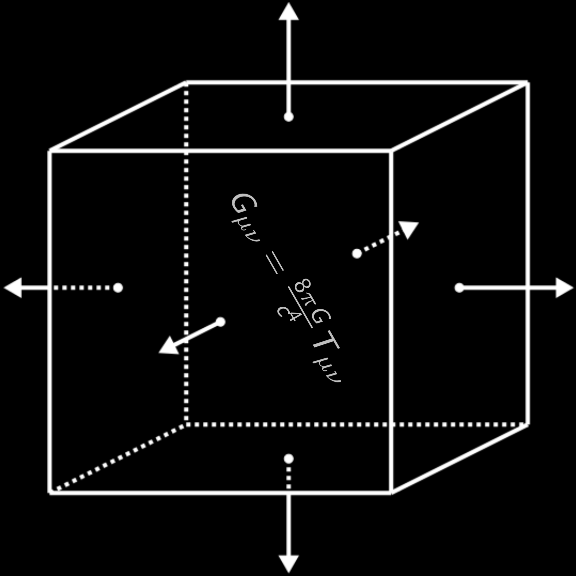


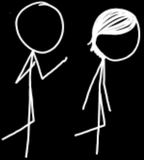
Using this recipe with millions of particles we can simulate galaxy formation!"

Simulating galaxies is nice, but it was not answering their (meta)physical questions.

So they decided to to do better. “Let’s try to investigate larger scales with galaxy-sized particles” they said.

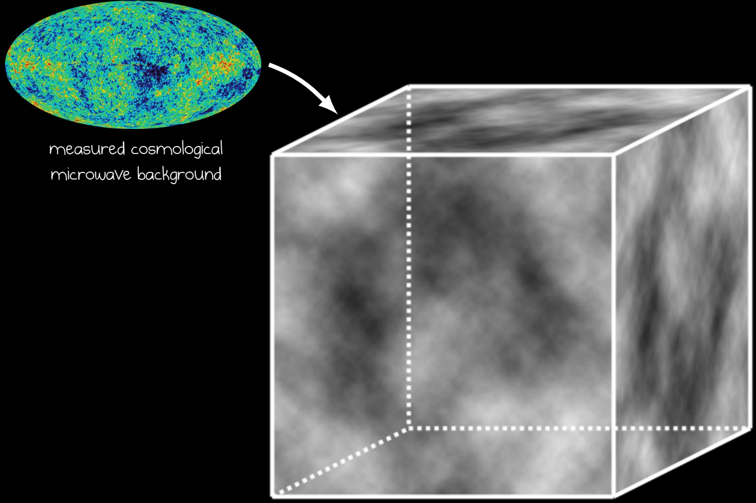
First, they took a supercomputer.





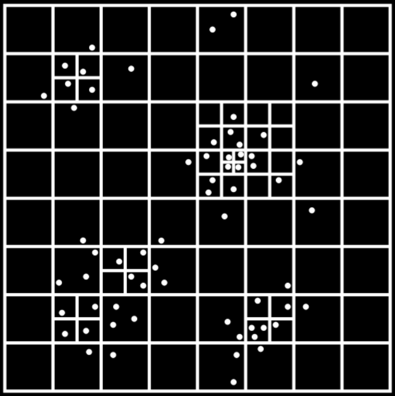
Second, they made the box expand as the Universe does✫ according to the theory of General Relativity (GR)

(✫here, they did not consider backreaction effects)

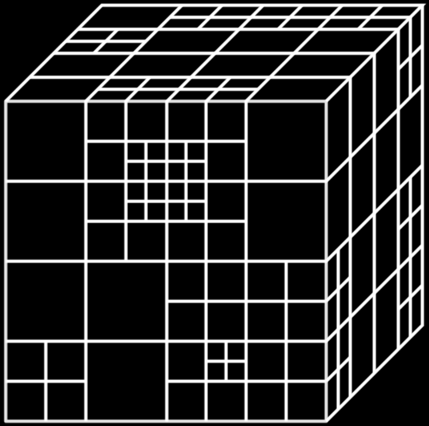


Third, they filled the box with billions of particles with the same distribution (statistically speaking) as the matter in the primordial Universe

Quadtree (2D)



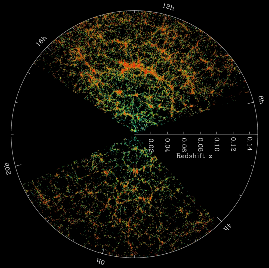
Octree (3D)



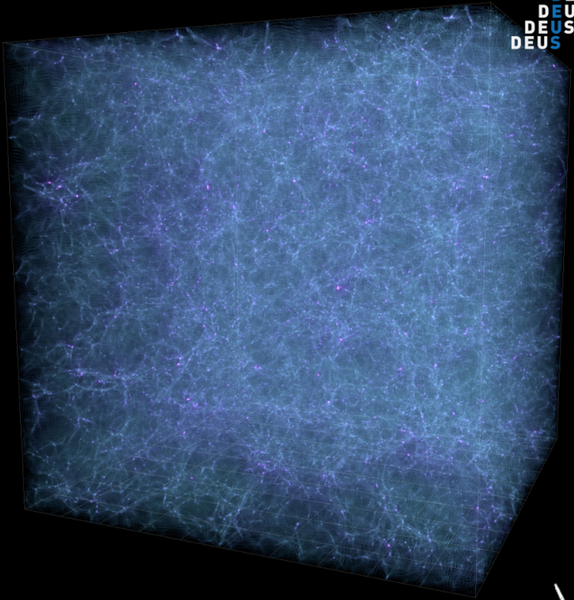
Fourth, they updated their algorithm using an Adaptive Mesh Refinement (AMR) strategy to increase the resolution in regions of interest.

And finally, after all this work, they ran their simulation, using millions of computing hours over thousands of cores and generating hundreds of terabytes of data.

And this is what they obtained.



Actual cosmic
web structure
observed by the SDSS

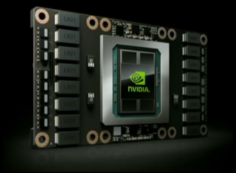


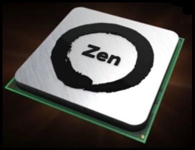
Simulated cosmic web structure
(each point is of the mass of a Milky Way)

It was nice and exciting,

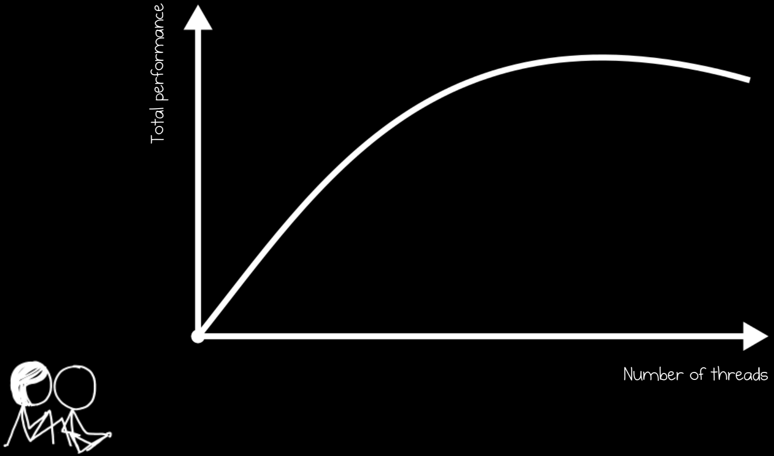
But to answer their questions,
they needed far more computing power.

Thankfully, new architectures and supercomputers were coming...





But they soon realized that there was a MAJOR problem: their code would not scale up to millions of cores!



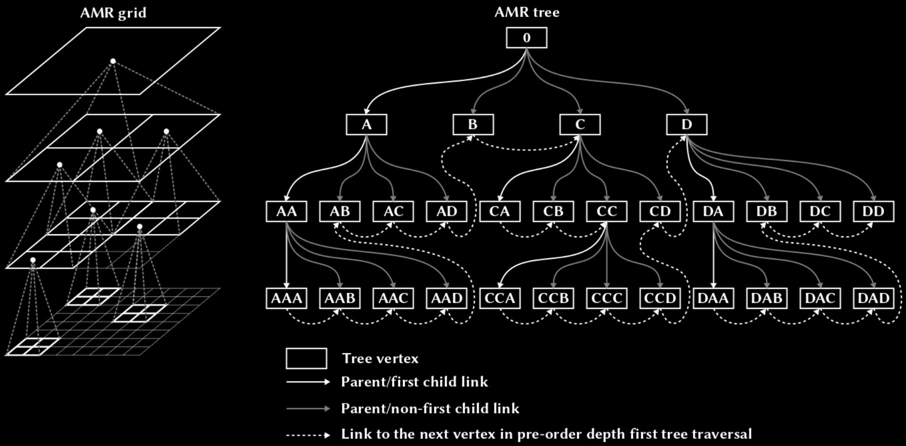
And a part of this problem can be boiled down to data structures: on supercomputers, pure computing capabilities are improving faster than memory performances

One cycle on a 3 GHz processor	1	ns			
L1 cache reference	0.5	ns			
Branch mispredict	5	ns			
L2 cache reference	7	ns			14x L1 cache
Mutex lock/unlock	25	ns			
Main memory reference	100	ns			20x L2, 200x L1
Compress 1K bytes with Snappy	3,000	ns			
Send 1K bytes over 1 Gbps network	10,000	ns	0.01	ms	
Read 4K randomly from SSD*	150,000	ns	0.15	ms	
Read 1 MB sequentially from memory	250,000	ns	0.25	ms	
Round trip within same datacenter	500,000	ns	0.5	ms	
Read 1 MB sequentially from SSD*	1,000,000	ns	1	ms	4X memory
Disk seek	10,000,000	ns	10	ms	20x datacenter RT
Read 1 MB sequentially from disk	20,000,000	ns	20	ms	80x memory, 20X SSD
Send packet CA->Netherlands->CA	150,000,000	ns	150	ms	

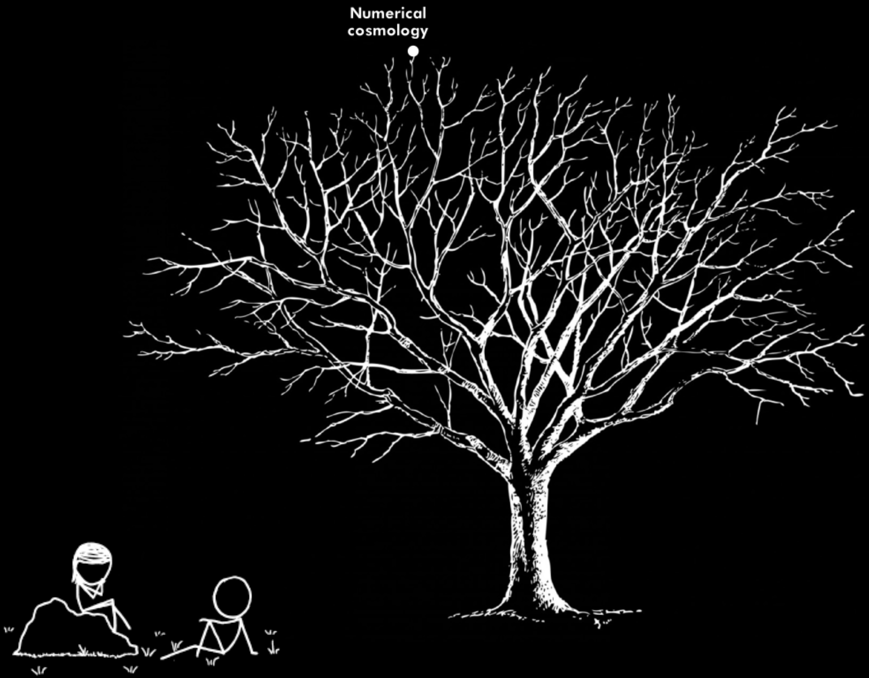
From Chandler Carruth "Efficiency with Algorithms, Performance with Data Structures" (cppcon 2014)

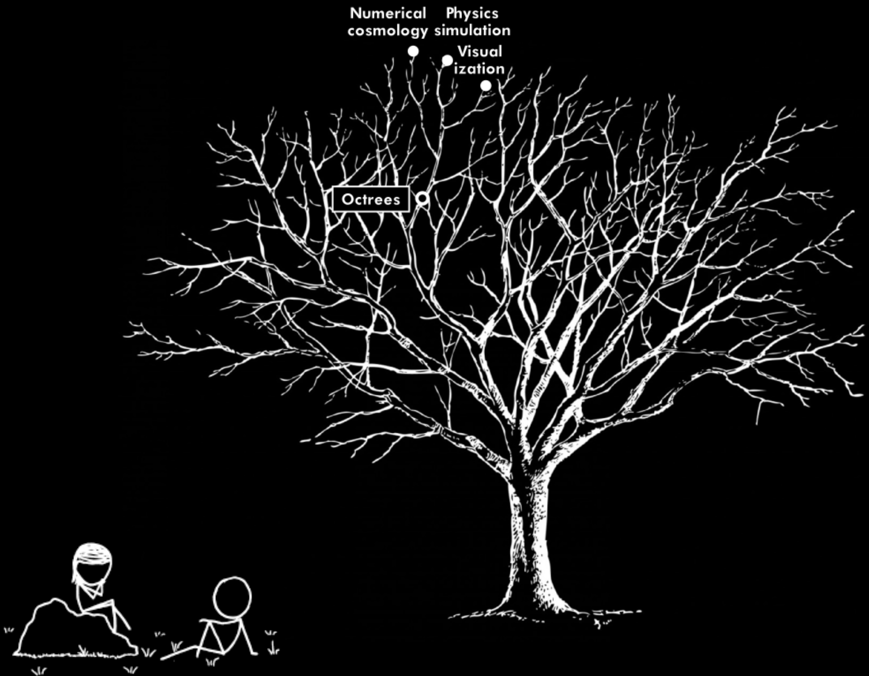
Credit: Jeffrey Dean, Google Research

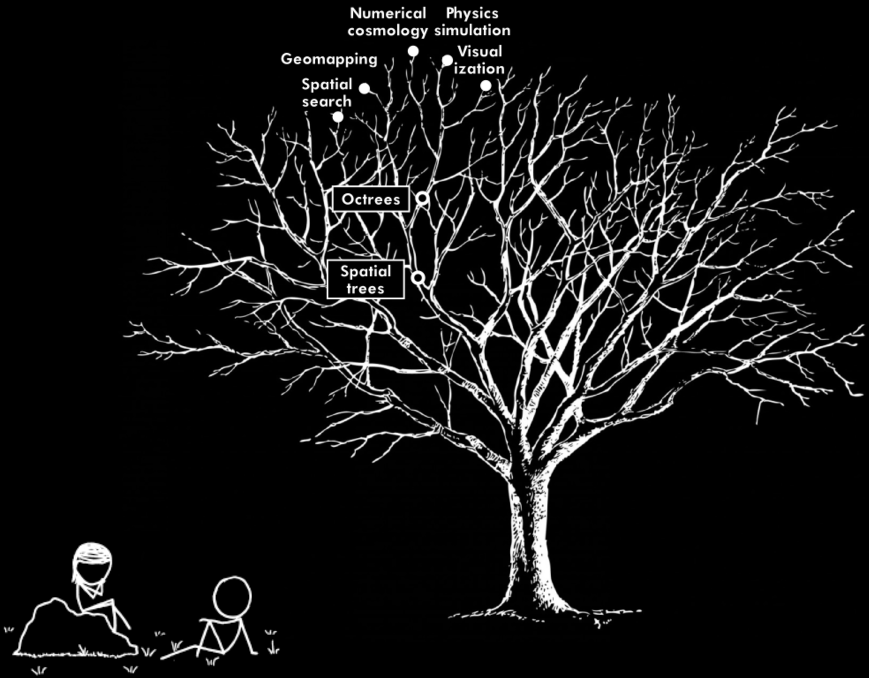
Explicit trees kill performances because of poor cache-awareness



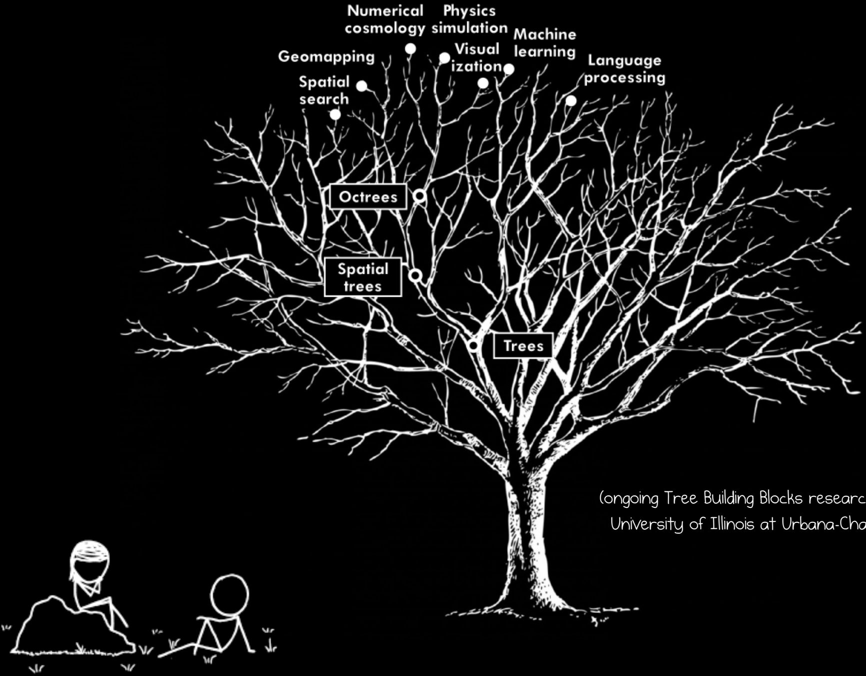
So they decided to get rid of explicit trees
to make the most of supercomputers.


Numerical
cosmology

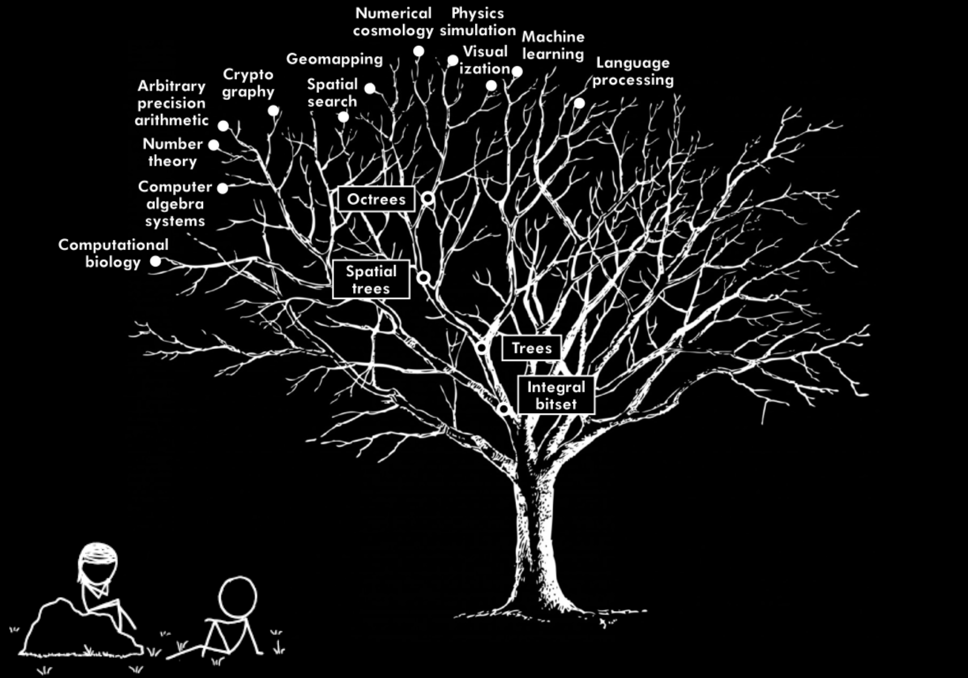

Numerical cosmology
Physics simulation
Visualization
Octrees



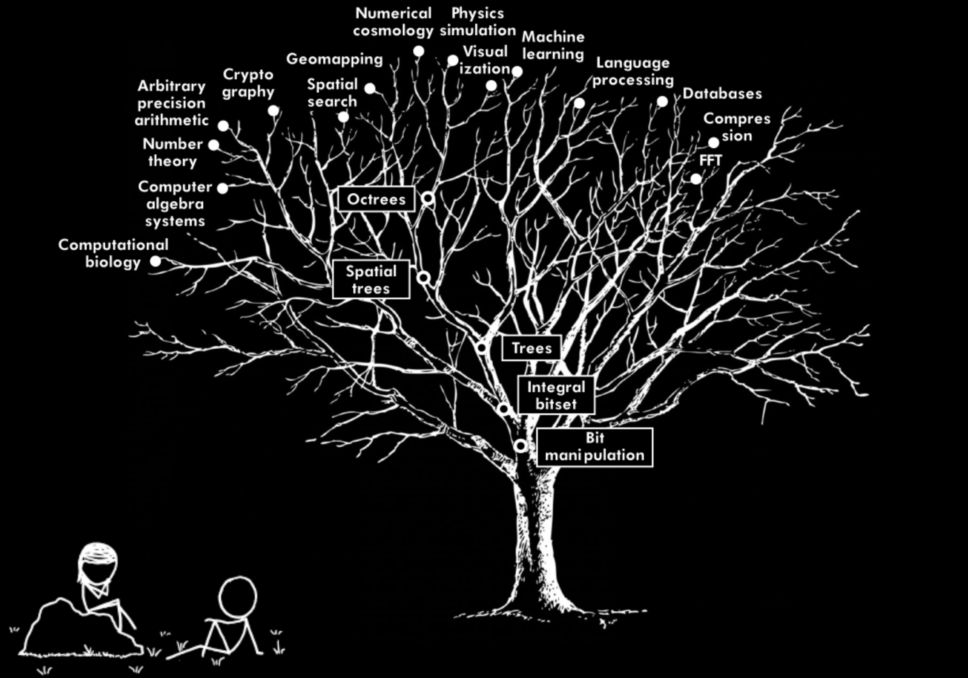
Numerical cosmology
Physics simulation
Geomapping
Visual ization
Spatial search
Octrees
Spatial trees

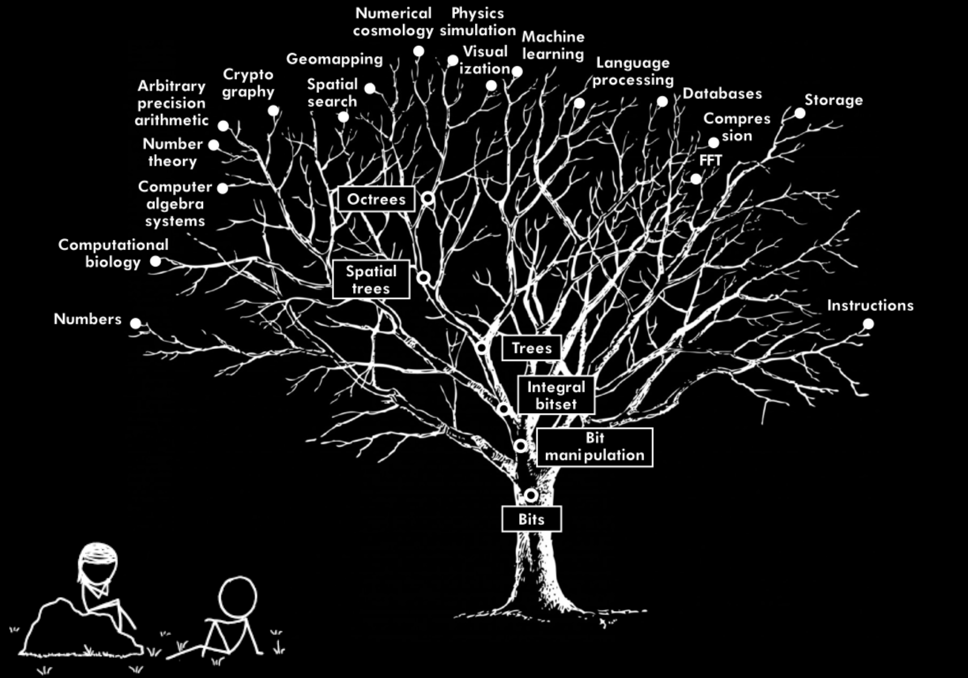


(ongoing Tree Building Blocks research project, University of Illinois at Urbana-Champaign)



Numerical cosmology
Physics simulation
Machine learning
Visual ization
Geomapping
Language processing
Crypto graphy
Spatial search
Arbitrary precision arithmetic
Number theory
Computer algebra systems
Octrees
Computational biology
Spatial trees
Trees
Integral bitset


Numerical cosmology
Physics simulation
Machine learning
Geomapping
Visual ization
Language processing
Crypto graphy
Spatial search
Arbitrary precision arithmetic
Databases
Compres sion
Number theory
FFT
Computer algebra systems
Octrees
Computational biology
Spatial trees
Trees
Integral bitset
Bit mani pulation



Numerical cosmology
Physics simulation
Machine learning
Geomapping
Visual ization
Language processing
Crypto graphy
Spatial search
Arbitrary precision arithmetic
Databases
Storage
Compres sion
Number theory
FFT
Computer algebra systems
Octrees
Computational biology
Spatial trees
Instructions
Numbers
Trees
Integral bitset
Bit mani pulation
Bits

They soon realized that to make the most efficient trees ever they would need very efficient ways to manipulate bits.

And this is how this whole story started...

They soon realized that to make the most efficient trees ever they would need very efficient ways to manipulate bits.

And this is how this whole story started...

Thanksfully, the old magician of C++ lands came to them to help them to start their quest

They soon realized that to make the most efficient trees ever they would need very efficient ways to manipulate bits.

And this is how this whole story started...

Thanksfully, the old magician of C++ lands came to them to help them to start their quest

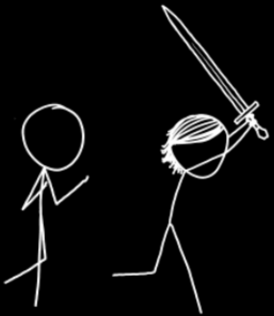
He gave them a very precious (std::)map and disappeared...


TERRA INCOGNITA
programmers eat each other here
Dreadful Random Memory Corruption Dragon of Doom
Obelisk of Constexpr
C++11 border
λ Notorious Gates of Anonymity
Nest of the Cute long long
auto
C++98 border
Data stronghold
private
protected
public
final
friend
override
Diamond mine
Virtual inheritance mountains
Ancient structure with class circle
Waterfall of destruction
Multiple inheritance forest
Exceptional swamps
try
catch
throw
unexpected
terminate
Templates maze
Alexandrescu castle
Curiously recurring template Ouroboros
Bjarne Stroustrup tower
Type casting ghosts dance on the meadow
Little Lost Goto
Inheritance forest
Sutter's Mill
THE HOLY STANDARD TERRITORY
ISO/IEC 14882:1998
ISO/IEC 14882:2003
ISO/IEC 14882:2011
Pure virtual river
Bugs live here there and everywhere
Bitwise canyon
Memory managment desert
Castle of a Dynamic Memory
The Epic Heap Memory Allocation Siege
New
Heap
Delete
Quicksands. Beware!
not enough memory
access violation
memory leaks
malloc Oasis
free Oasis
realloc
#include fort
STL
BOOST
std::rope
std::thread
std::unordered_set
std::unordered_map
The Great New Standard War
std::pair
std::map
Ruins of std::auto_ptr
std::multimap
std::list
std::vector
ios lake
istream
iostream
ostream
boost::graph
boost::gregorian
boost::bimap
boost::asio
boost::shared_ptr
boost::tuple
boost::regex
Concepts Shipwreck
WTL
external libraries seas
Loki
Qt
C++/Tk
ACE
The C++ Lands
Its amazing creatures and weird beasts
2012 edition
explorer
Alena [http://alenacpp.blogspot.com]
cartographer
Jim [http://jimblog.me]

TERRA INCOGNITA
programmers eat each other here
Dreadful Random Memory Corruption Dragon of Doom
Obelisk of Constexpr
C++11 border
Notorious Gates of Anonymity
STL
std::rope
std::thread
std::unordered_set
std::power
std::unordered_map
The Great New Standard War
std::pair
std::map
std::multimap
std::list
std::vector
ios lake
istream
iostream
ostream
BOOST
boost::graph
boost::gregorian
boost::bimap
boost::asio
boost::shared_ptr
boost::tuple
boost::regex
Ruins of std::auto_ptr
Concepts Shipwreck
WTL
external libraries seas
try
catch
throw
Exceptional swamps
unexpected
terminate
Templates maze
Alexandrescu castle
Curiously recurring template Ouroboros
Loki
Qt
ISO/I
Bitwise canyon
Stroustrup tower
THE HOLY STANDARD TERRITORY
Sutter's Mill
ISO/IEC 14882:2003
ISO/IEC 14882:1998
ISO/IEC 14882:2011
Pure virtual river
#include fort
Bitwise canyon
Bugs live here there and everywhere
Memory managment desert
The Epic Heap Memory Allocation Siege
Quicksands. Beware!
not enough memory
access violation
memory leaks
malloc Oasis
free Oasis
realloc
New
Delete
Heap
ACE
The C++ Lands
Its amazing creatures and weird beasts
2012 edition
explorer
Alena [http://alenacpp.blogspot.com]
cartographer
Jim [http://jimblog.me]


TERRA INCOGNITA
programmers eat each other here
Dreadful Random Memory Corruption Dragon of Doom
Obelisk of Constexpr
Notorious Gates of Anonymity
std::vector<bool>
std::bitset
Here be dragons
Bitwise canyon
ISO/IEC 14882:2011
Alexandrescu castle
Loki
Curiously recurring template Ouroboros
Stroustrup tower
THE HOLY STANDARD TERRITORY
Sutter's Mill
ISO/IEC 14882:1998
#include fort
Memory managment desert
The Epic Heap Memory Allocation Siege
New
Delete
Heap
malloc Oasis
free Oasis
realloc
ACE
Qt
The C++ Lands
Its amazing creatures and weird beasts
2012 edition
explorer
Alena [http://alenacpp.blogspot.com]
cartographer
Jim [http://jimblog.me]

So they decided to go there
with the hope to leverage bit manipulation
to speed up their trees and their simulations...



Chapter II: Manipulating bits with the current ISO C++ Standard

Constant-size bit container: std::bitset

template <std::size_t N> class bitset

The class template bitset represents a fixed-size sequence of N bits. Bitsets can be manipulated by standard logic operators and converted to and from strings and integers. (*source: cppreference.com*)

```
// Example from cppreference.com
#include <iostream>
#include <bitset>

int main() {
  // Initialization
  std::bitset<8> b("00010010");
  std::cout<<"initial value: "<<b<<'\n';
  // Find the first unset bit
  size_t idx = 0;
  while (idx < b.size() && b[idx]) ++idx;
  // Continue setting bits until half the bitset is filled
  while (idx < b.size() && b.count() < b.size()/2) {
    b.set(idx);
    std::cout<<"setting bit "<<idx<<": "<<b<<'\n';
    while (idx < b.size() && b.test(idx)) ++idx;
  }
}

// Output
// initial value: 00010010
// setting bit 0: 00010011
// setting bit 2: 00010111
```

Constant size bit container: `std::bitset`

What is good with bitsets?

- Very simple and easy to use
- Set of optimized members like `test`, `all`, `any`, `none`, `count`, `set`, `reset`, `flip`

Limitations

- Very limited functionality
- No `begin` and `end` iterators: not compatible with algorithms and standard containers
- No control on the underlying representation

Underlying representation

In terms of implementation, bits are likely to be stored in a contiguous array of unsigned integers. However, there is no way to access this underlying representation.

What does std::bitset::operator[] return?

```
std::bitset<8> b(42);
bool boolean = b[0];
auto something = b[0];
boolean += 1;
something += 1; // Failure
```

Limitations

- std::bitset::operator[] returns a proxy of type std::bitset::reference
- *Almost* like a bool...
- But only *almost*: different promotion rules, different arithmetic, a member flip and a different behavior for operator∼
- Very confusing and error-prone

Very confusing and error-prone?

```cpp
// Initialization
std::bitset<8> b(0);
bool boolean = b[0];
auto something = b[0];

// Operations
boolean = ~(~boolean);
something = ~(~something);

// Display
std::cout<<boolean<<something<<std::endl;
```

Question

What does that print? 00, 11, 10 or 01?

Very confusing and error-prone?

```cpp
// Initialization
std::bitset<8> b(0);
bool boolean = b[0];
auto something = b[0];

// Operations
boolean = ~(~boolean);
something = ~(~something);

// Display
std::cout<<boolean<<something<<std::endl;
```

Question

What does that print? 00, 11, 10 or 01?

Answer

10

Why?

- `boolean` → `~false` → `~(-1)` → `boolean = 0` → `boolean == 0`
- `something` → `~reference(0)` → `~(true)` → `something = -2` → `something == 1`

Dynamic size bit container: std::vector<bool>

What is std::vector<bool>?

A mistake

template <class Allocator> class vector<bool, Allocator>

std::vector<bool> is a space-efficient specialization of std::vector for the type bool. The manner in which std::vector<bool> is made space efficient (as well as whether it is optimized at all) is implementation defined. One potential optimization involves coalescing vector elements such that each element occupies a single bit instead of sizeof(bool) bytes. std::vector<bool> behaves similarly to std::vector, but in order to be space efficient, it:

- Does not necessarily store its elements as a contiguous array (so &v[0] + n != &v[n])
- Exposes class std::vector<bool>::reference as a method of accessing individual bits. In particular, objects of this class are returned by operator[] by value.
- Does not use std::allocator_traits::construct to construct bit values.
- Does not guarantee that different elements in the same container can be modified concurrently by different threads.

(*source: cppreference.com*)

Dynamic size bit container: std::vector<bool>

What is std::vector<bool>?

It's not a container

What's wrong with you std::vector<bool>?

- "std::vector<bool> is nonconforming, and forces optimization choice", H. Sutter (N1185) [1999]
- "std::vector<bool> : More problems, better solutions", H. Sutter (N1847) [2005]
- "Library issue 96: Fixing std::vector<bool>", B. Dawes (N2160) [2007]
- "A specification to deprecate std::vector<bool>", A. Meredith (N2204) [2007]

$\Rightarrow$ 2016: std::vector<bool> is still alive

Dynamic size bit container: `std::vector<bool>`

What is good with vector bool?

- Compact storage in memory
- `begin` and `end` iterators: compatible with standard algorithms

Problems

- Poor performances
- No access to the underlying representation
- No thread safety
- Breaks the normal behavior of containers
- Error-prone behavior of `std::vector<bool>::reference` (*almost* a `bool`).

On dynamic bitsets

The functionality is ok, but specializing `std::vector` for it was not the best idea ever. A `std::dynamic_bitset` (as in Boost) would have been a far better option. But even with that, most of the problems remain.

Bit manipulation

Example of use of bit instructions (Hilewitz, 2008)

Application \ Instruction		bfly ibfly	pex	pdep	setib	setb	pex.v	pdep.v	grp	bmm
Binary Compression			×							(×)
Binary Decompression				×						(×)
LSB Steganography	Encoding			×		×				(×)
	Decoding		×		×					(×)
Transfer Coding	Encoding			×						(×)
	Decoding		×							(×)
Integer Compression	Compression			×						(×)
	Decompression		×							(×)
Binary Image Morphology			×				(×)			(×)
Random Number Generation	Von Neumann						×			
	Toeplitz									×
Bioinformatics	Compression		×							(×)
	BLASTZ Alignment		×							(×)
	BLASTX Translation			×						(×)
	Reversal	×								(×)
Cryptography	Block Ciphers	×							×	×
	Stream Ciphers						×			×
	Public Key									×
	Future	?	?	?	?	?	?	?	?	?
Cryptanalysis	Linear									×*
	Algebraic									×
	Future/Proprietary	?	?	?	?	?	?	?	?	?
DARPA HPCS Discrete Math Benchmarks	Matrix Transpose									×*
	Equation Solving									×
Linear Feedback Shift Registers										×
Error Correction	Block Codes									×
	Convolutional Codes									×
	Puncturing		×	×						(×)

The current state of bit manipulation

Instructions

CPUs include more and more very efficient bit manipulation instructions but they are often left unused because of the lack of standard utilities to access them.

Bit Manipulation Instruction Sets

- ABM: Advanced Bit Manipulation, `POPCNT`, `LZCNT` (Intel SSE 4.2, AMD ABM)
- BMI1: Bit Manipulation Instruction Set 1 ($\geq$ Intel Haswell, AMD Piledriver)
- BMI2: Bit Manipulation Instruction Set 2 ($\geq$ Intel Haswell, AMD Excavator)
- TBM: Trailing Bit Manipulation ($\geq$ AMD Piledriver)
- BME: Bit Manipulation Engine (ARM Cortex)

Compiler intrinsic examples

- `_bzhi_u32`, `_bzhi_u64`, `_pdep_u32`, `_pdep_u64`, `_pext_u32`, `_pext_u64`
- `__builtin_clz`, `__builtin_ctz`, `__builtin_clrsb`, `__builtin_popcount`
- `__builtin_ia32_bextr_u32`, `__builtin_ia32_bextr_u64`
- `__builtin_ia32_lzcnt_u32`, `__builtin_ia32_lzcnt_u64`

The current state of bit manipulation

Past proposal

"A constexpr bitwise operations library for C++" (Fioravante, 2014)

Problems

- More than 60 bit-specialized functions: too many functions
- Too low level and domain specific
- Limited to integral arguments
- Does not provide tools to manipulate arbitrary long sequence of bits

Genericity

Need of something far more generic

Chapter III: Introducing The Bit Library

Summary of The Bit Library and P0237

What?

Designing tools to provide a generic way to operate on bits and sequence of bits.

Why?

- To be able to use unsigned integers as sets of bits
- To make the most of bit manipulation instruction sets
- To provide users with utilities to build fast bit manipulation algorithms
- To provide users with utilities to build efficient bit-based data structures

Application areas

- Hashing
- Video games
- Image processing
- Cryptography
- Random number generation
- Binary compression
- Error correction
- High-performance computing
- Arbitrary-precision arithmetic

Concrete examples of use

- User-defined bit sets and bit arrays, iteration over bits
- Access to the underlying bits of bounded and unbounded integers
- Allowing `std::count` to call a `POPCNT` instruction when executed on bits

Motivation

Bjarne Stroustrup - The C++ Programming Language (2013)

"unsigned integer types are ideal for uses that treat storage as a bit array."

Except that...

...there is no standard way to access and manipulate bits in C++.

Main motivations

- Ease the use of unsigned integers as bit arrays
- Provide a standard way to access and manipulate unique bits (set, reset, flip...)
- Provide an abstraction to leverage bit manipulation instruction sets
- Provide efficient versions of standard algorithms on bits
- Facilitate the design of fast bit manipulation algorithms on sequences of bits
- Facilitate the implementation of data structures based on bit sequences
- Provide tools to access the underlying representation of such data structures
- Provide tools to easily build alternatives to `std::vector<bool>`

Objectives

Simplicity, genericity and efficiency.

Solution summary

Key idea: bit_iterator

All of that can be achieved through a carefully designed `bit_iterator` acting as an iterator adaptor (like `std::reverse_iterator`).

template <class Iterator> class bit_iterator

A `bit_iterator` is a tool that allows to reinterpret a sequence of unsigned integers as a sequence of bits. It provides an API that is both:

- high-level: easy to use from the general user point of view
- low-level: gives access to the underlying representation for optimization purposes

High-level point of view: counting bits (assuming std:: prefix)

```
// Initialization
using uint_t = unsigned int;
using container_t = std::list<uint_t>;
using iterator_t = typename container_t::iterator;
container_t container = {0, 1, 2, 3, 4};

// From the bit number 5 of container[0], to the end of the container
std::bit_iterator<iterator_t> first(std::begin(container), 5);
std::bit_iterator<iterator_t> last(std::end(container));

// Counts the bits set to 1
std::cout<<std::count(first, last, std::one_bit)<<std::endl;
```

Solution summary

```
unsigned int i = 1751901291;
```

Object representation in memory (little endian)

01101011 11100100 01101011 01101000

01101000011010111110010001101011

Binary value

```
std::list<unsigned int> uintlist =
{1751901291, 1751901292, 1751901293, 1751901294};
```

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
00	0	1	1	0	1	0	0	0	0	1	1	0	1	0	1	1	1	1	1	0	0	1	0	0	0	1	1	0	1	0	1	1
01	0	1	1	0	1	0	0	0	0	1	1	0	1	0	1	1	1	1	1	0	0	1	0	0	0	1	1	0	1	1	0	0
02	0	1	1	0	1	0	0	0	0	1	1	0	1	0	1	1	1	1	1	0	0	1	0	0	0	1	1	0	1	1	0	1
03	0	1	1	0	1	0	0	0	0	1	1	0	1	0	1	1	1	1	1	0	0	1	0	0	0	1	1	0	1	1	1	0

[red] - 23 = [green] [red] + 55 = [yellow]

Solution summary

3 key functions

- it.base() (bit_iterator::base): returns the underlying iterator, inspired from std::reverse_iterator::base
- it.position() and (*it).position() (bit_iterator::position and bit_reference::position): returns the current bit position in the underlying value, starting from 0 (LSB) to binary_digits<T>::value − 1 (MSB)
- (*it).address() (bit_reference::address): returns a pointer to the underlying value

Low-level point of view: counting bits (1/2)

```
template <class InputIt>
typename bit_iterator<InputIt>::difference_type
count(
    bit_iterator<InputIt> first,
    bit_iterator<InputIt> last,
    bit_value value
)
{
    // Assertions
    _assert_range_viability(first, last);

    // Initialization
    using underlying_type = typename bit_iterator<InputIt>::underlying_type;
    using difference_type = typename bit_iterator<InputIt>::difference_type;
    constexpr difference_type digits = binary_digits<underlying_type>::value;
```

Solution summary

Low-level point of view: counting bits (2/2)

```
    difference_type result = 0;
    auto it = first.base();

    // Computation when bits belong to several underlying values
    if (first.base() != last.base()) {
        if (first.position() != 0) {
            result = _popcnt(*first.base() >> first.position());
            ++it;
        }
        for (; it != last.base(); ++it) {
            result += _popcnt(*it);
        }
        if (last.position() != 0) {
            result += _popcnt(*last.base() << (digits - last.position()));
        }
    // Computation when bits belong to the same underlying value
    } else {
        result = _popcnt(_bextr<underlying_type>(
            *first.base(),
            first.position(),
            last.position() - first.position()
        ));
    }

    // Negates when the number of zero bits is requested
    if (!static_cast<bool>(value)) {
        result = std::distance(first, last) - result;
    }

    // Finalization
    return result;
}
```

Design questions

Key design questions

- What is `bit_iterator<Iterator>::difference_type`?
- What is `bit_iterator<Iterator>::iterator_category`?
- What is `bit_iterator<Iterator>::value_type`?
- What is `bit_iterator<Iterator>::reference`?
- What is `bit_iterator<Iterator>::pointer`?

Design questions

Key design questions

- What is `bit_iterator<Iterator>::difference_type`?
- What is `bit_iterator<Iterator>::iterator_category`?
- What is `bit_iterator<Iterator>::value_type`?
- What is `bit_iterator<Iterator>::reference`?
- What is `bit_iterator<Iterator>::pointer`?

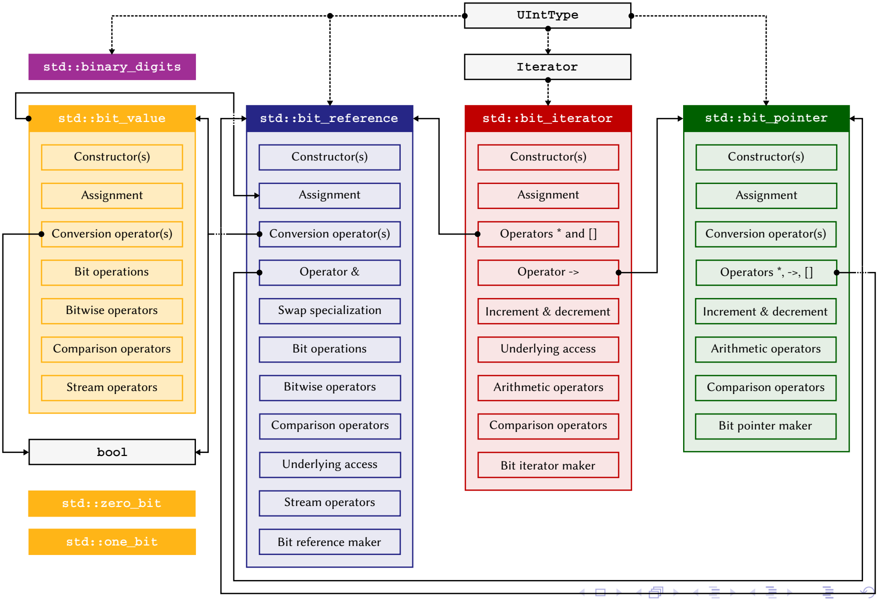
Answer: the easy ones

- `bit_iterator<Iterator>::difference_type`
  ⇒ Implementation defined, but at least as large as `std::ptrdiff_t`
- `bit_iterator<Iterator>::iterator_category`
  ⇒ Same as `std::iterator_traits<Iterator>::iterator_category`

Answer: the difficult ones

- `bit_iterator<Iterator>::value_type` ⇒ ?
- `bit_iterator<Iterator>::reference` ⇒ ?
- `bit_iterator<Iterator>::pointer` ⇒ ?

Global architecture



Chapter IV: On some tricky details

What is bit_iterator<Iterator>::value_type?

What is a bit?

The innocent question, that is actually very complex to answer...

Related questions

- What functionalities a bit should provide?
- What should the arithmetic behaviour of a bit be?

Existing problem in std::bitset and std::vector<bool>

Their boolean value_type has a different behaviour than their reference type leading to potential errors: as an example, the behaviour of the operator∼ is different, and operator+= exists for their value type, but not for their reference type.

What is a bit?

```cpp
struct field {unsigned int b : 1;};

bool b0 = false; b0 = ~b0; b0 = ~b0;                      // 1
auto x0 = std::bitset<1>{}[0]; x0 = ~x0; x0 = ~x0;        // 0
auto f0 = field{}; f0.b = ~f0.b; f0.b = ~f0.b;            // 0

bool b1 = false; b1 = ~~b1;                               // 0
auto x1 = std::bitset<1>{}[0]; x1 = ~~x1;                 // 1
auto f1 = field{}; f1.b = ~~f1.b;                         // 0

bool b2 = false; b2 += 1; b2 += 1;                        // 1
auto x2 = std::bitset<1>{}[0]; x2 += 1; x2 += 1;          // X
auto f2 = field{}; f2.b += 1; f2.b += 1;                  // 0

bool b3 = false; b2 = b3 + 1; b3 = b3 + 1;                // 1
auto x3 = std::bitset<1>{}[0]; x3 = x3 + 1; x3 = x3 + 1;  // 1
auto f3 = field{}; f3.b = f3.b + 1; f3.b = f3.b + 1;      // 0

bool b4 = false; b4 += 3;                                 // 1
auto x4 = std::bitset<1>{}[0]; x4 += 3;                   // X
auto f4 = field{}; f4.b += 3;                             // 1
```

What is a bit?

According to the C standard (section 3.5)

A *bit* is a unit of data storage in the execution environment large enough to hold an object that may have one of two values.

According to the C++ standard ([intro.memory])

A *bit* is an element of a contiguous sequence forming a byte, a *byte* being the fundamental storage unit in the C++ memory model and being at least 8-bit long.

Tentative of mathematical definition (Wikipedia)

The word *bit* stands for *binary digit*, a digit being a numeric symbol used in combinations to represent numbers in positional numeral systems.

What is a boolean data type? (Wikipedia)

A boolean data type is a data type, having two values (usually denoted true and false), intended to represent the truth values of logic and boolean algebra.

One of the most important slide of this talk

Summary

- A bit is a binary digit
- A boolean is a logical data type

A bit is not a bool

Bits and booleans are often identified, but they are two very different concepts. Both just happen to have two values.

On `std::vector<bool>`

If bits and booleans were the same thing, `std::vector<bool>` would not raise any design issue and standardization papers about it would not exist. But they do exist...

On `std::array<bool, N>` and `std::bitset<N>`

If bits and booleans were the same thing, `std::array<bool, N>` and `std::bitset<N>` would be equivalent. But they are not...

What is `bit_iterator<Iterator>::value_type`? ⇒ a `bit_value`

`class bit_value`

Represents an independent individual bit.

Naming: why `bit_value` and not `bit`?

Because a `bit_value` mimics a bit, but is not actually a bit since bits cannot be stored individually in memory.

Main functionalities

- Take an unsigned integer and a bit position for construction
- No arithmetic behavior to avoid confusion (same approach as `std::byte`)
- Bitwise operators
- Flip, set and reset members
- Explicit conversion to `bool`

```
bit_value bval(3U, 1); // Get the bit at position 1 of 3
bval.flip();           // Flips the bit
bval = bit_value(1U);  // Same as bit_value(1U, 0)
bval.set();            // Sets the bit to one
std::cout<<bval<<'\n'; // Prints the value of the bit (1)
```

What is bit_iterator<Iterator>::reference? ⇒ a bit_reference

template <class UInt> class bit_reference<UInt>

Represents a bit reference to a bit of an unsigned integer. Can be implemented as a reference (or a pointer) to an unsigned integer and a position.

Main functionalities

- Same behavior as bit_value
- Take a reference to an unsigned integer and a bit position for construction
- Overloaded operator& to return a bit_pointer
- address and position members to get the address of the referenced unsigned integer and the position of the bit within bit

```
using uint_t = unsigned int;             // Sets the type of unsigned integer
uint_t ui = 4;                           // Creates an unsigned integer
bit_reference<uint_t> bref(ui, 3);       // Creates a ref to the 3rd bit of ui
bref.flip();                             // Flips the 3rd bit of ui
std::cout<<bref<<'\n';                   // Prints the 3rd bit of ui (1)
std::cout<<bref.position()<<'\n';        // Prints the position of the bit (3)
std::cout<<*(bref.address())<<'\n';      // Prints ui (12)
```

What is bit_iterator<Iterator>::pointer? ⇒ a bit_pointer

template <class UInt> class bit_pointer<UInt>

Represents a bit pointer to a bit of an unsigned integer. Can be implemented as a pointer to an unsigned integer and a position.

Main functionalities

- Complementary of bit_reference, mimicking a pointer to a bit
- Take a pointer to an unsigned integer and a bit position for construction
- Overloaded operator* to return a bit_reference

```
using uint_t = unsigned int;                   // Sets the type of unsigned integer
uint_t ui[2] = {4, 10};                        // Creates an array of unsigned integer
bit_pointer<uint_t> bptr0(&ui[0], 3); // Creates a pointer to the 3rd bit of ui[0]
bit_pointer<uint_t> bptr1(&ui[1], 8); // Creates a pointer to the 8th bit of ui[1]
bptr0->flip();                                 // Flips the 3rd bit of ui[0]
std::cout<<*bptr0<<'\n';                       // Prints the 3rd bit of ui (1)
std::cout<<bptr0->position()<<'\n';            // Prints the position of the bit (3)
std::cout<<*(bptr0->address())<<'\n'; // Prints ui (12)
std::cout<<bptr1 - bptr0<<std::endl;           // Prints the distance in bits (37)
```

On binary_digits and bit_iterator

template <class UInt> struct binary_digits

Helper struct inheriting from std::integral_constant<std::size_t, N> and giving the number of bits of unsigned integral types. Bit values, references, pointers and iterators rely on this information. Can be specialized for user types to adapt the bit library.

template <class Iterator> class bit_iterator

Combine all the preceding and provides a generic tool to manipulate bit sequences. Can used to design bit manipulation algorithms and bit oriented data structures such as multiprecision integers.

Main functionalities

- Based on bit_value, bit_reference and bit_pointer
- Take an iterator on a sequence of unsigned integers and a position for construction
- Overloaded operator* to return a bit_reference
- base and position members to get the underlying iterator and the current bit position within the current underlying unsigned integer

Last words on bit_iterator

```
// Initialization
using uint_t = unsigned int;
using container_t = std::list<uint_t>;
using iterator_t = typename container_t::iterator;
container_t container = {0, 1, 2, 3, 4};

// From the bit number 5 of container[0], to the end of the container
bit_iterator<iterator_t> first(std::begin(container), 5);
bit_iterator<iterator_t> last(std::end(container));

// Counts the bits set to 1
std::cout<<count(first, last, one_bit)<<std::endl;
```

Advantages: easy to use, generic, efficient

- Very generic: can be used to reinterpret any kind of sequence of unsigned integers as a sequence of bits
- Zero overhead: most compilers can optimize the abstraction
- Acts as a standard API between users (high level) and implementers of bit manipulation algorithms or bit oriented data structures (low level)
- Good integration with the standard library: standard algorithms can be specialized to use intrinsics on bit iterators

Chapter V: Defeating vector bool

Preliminary words

Previous work

Investigations have been done in the past to iterate of bit sequences efficiently. See in particular the excellent blog post by Howard Hinnant "On vector bool" (2012).

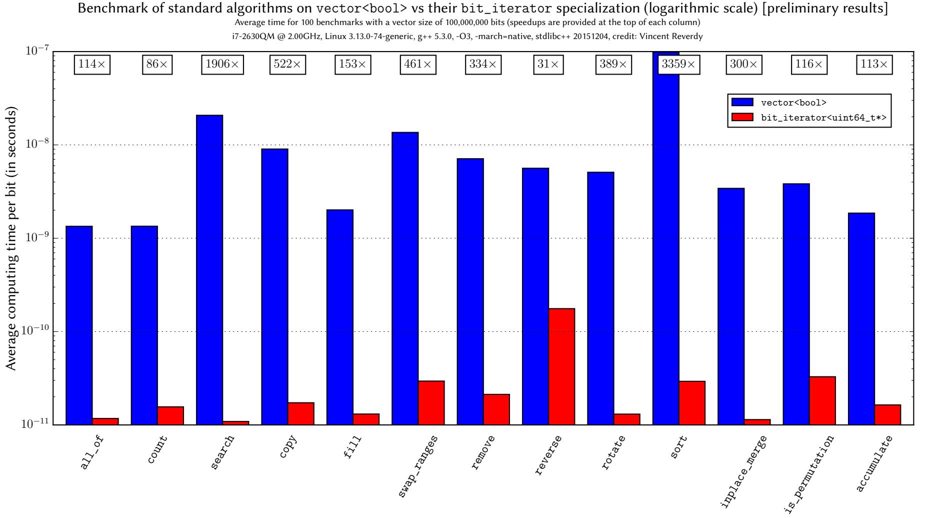
Bit Twiddling Hacks

The webpage "Bit Twiddling Hacks" by Sean Eron Anderson has been a great source of inspiration to implement some of the bit manipulation algorithms.

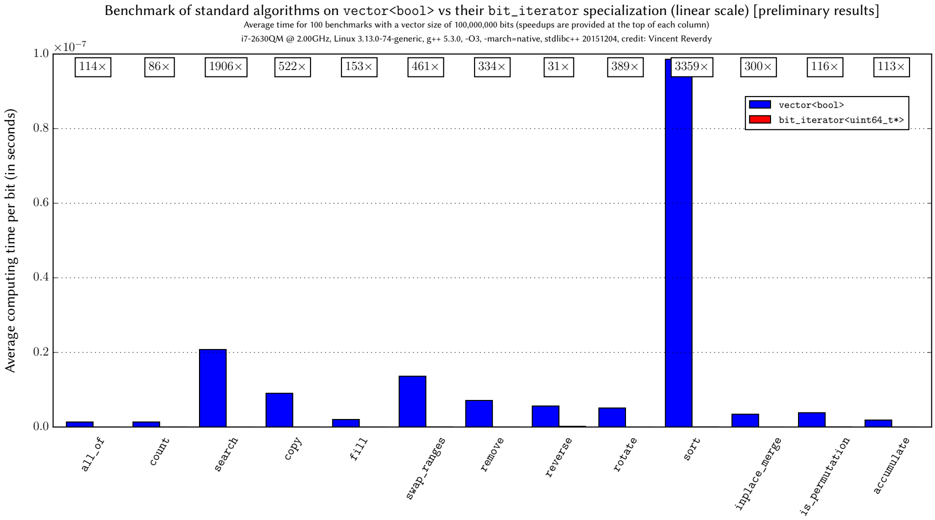
Acknowledgments

The implementation of `reverse` was done with Maghav Kumar (mkumar10@illinois.edu) at the University of Illinois at Urbana-Champaign.

Performances: early benchmark



Performances: early benchmark



Implementing a new version of std::reverse for bit iterators (1/3)

```
// Reverses the order of the bits in the provided range
template <class BidirIt>
void reverse(bit_iterator<BidirIt> first, bit_iterator<BidirIt> last)
{
    // Assertions
    _assert_range_viability(first, last);

    // Initialization
    using underlying_type = typename bit_iterator<BidirIt>::underlying_type;
    using size_type = typename bit_iterator<BidirIt>::size_type;
    constexpr size_type digits = binary_digits<underlying_type>::value;
    const bool is_last_null = last.position() == 0;
    size_type diff = (digits - last.position()) * !is_last_null;
    auto it = first.base();
    underlying_type first_value = {};
    underlying_type last_value = {};

    // Reverse when bit iterators are aligned
    if (first.position() == 0 && last.position() == 0) {
        std::reverse(first.base(), last.base());
        for (; it != last.base(); ++it) {
            *it = _bitswap(*it);
        }
```

Implementing a new version of std::reverse for bit iterators (2/3)

```
    // Reverse when bit iterators do not belong to the same underlying value
    } else if (first.base() != last.base()) {
        // Save first and last element
        first_value = *first.base();
        last_value = *std::prev(last.base(), is_last_null);
        // Reverse the underlying sequence
        std::reverse(first.base(), std::next(last.base(), !is_last_null));
        // Shift the underlying sequence to the left
        if (first.position() < diff) {
            it = first.base();
            diff = diff - first.position();
            for (; it != last.base(); ++it) {
                *it = _shld<underlying_type>(*it, *std::next(it), diff);
            }
            *it <<= diff;
            it = first.base();
        // Shift the underlying sequence to the right
        } else if (first.position() > diff) {
            it = std::prev(last.base(), is_last_null);
            diff = first.position() - diff;
            for (; it != first.base(); --it) {
                *it = _shrd<underlying_type>(*it, *std::prev(it), diff);
            }
            *it >>= diff;
            it = first.base();
        }
        // Bitswap every element of the underlying sequence
        for (; it != std::next(last.base(), !is_last_null); ++it) {
            *it = _bitswap(*it);
        }
```

Implementing a new version of std::reverse for bit iterators (3/3)

```
            // Blend bits of the first element
            if (first.position() != 0) {
                *first.base() = _bitblend<underlying_type>(
                    first_value,
                    *first.base(),
                    first.position(),
                    digits - first.position()
                );
            }
            // Blend bits of the last element
            if (last.position() != 0) {
                *last.base() = _bitblend<underlying_type>(
                    *last.base(),
                    last_value,
                    last.position(),
                    digits - last.position()
                );
            }
        // Reverse when bit iterators belong to the same underlying value
        } else {
            *it = _bitblend<underlying_type>(
                *it,
                _bitswap(*it >> first.position()) >> diff,
                first.position(),
                last.position() - first.position()
            );
        }
}
```

Enhanced version of std::reverse

Difference between the two versions

- The new version works for all cases, not only aligned bit sequences
- The fundamental low level functions like `_bitswap` have been improved (`bit_details` file for details), combining compiler intrinsics, "Bit Twiddling Hacks" and template metaprogramming

Results

- Speed-up of the old version: 31×
- Speed-up of the new version: 86×

Epilogue

Summary

Summary: The Bit Library

- `std::vector<bool>` is broken and `std::bitset` is very limited
- `bit_iterator` is a good way to combine ease of use, genericity and performance (orders of magnitude better than `std::vector<bool>`) for bit manipulation
- Abstracting bits is not an easy task
- The Bit Library is still work in progress: specialization of most of the standard algorithms need to be done, `bit_value` and `bit_reference` functionalities are still likely to evolve
- The library is available online for everyone to test, benchmark, share and participate

Summary: standardization

- Proposal and wording `P0237` already submitted several times
- Positive feedback from LEWG
- Some issues need to be solved particularly regarding to cv-qualifiers
- We are aiming for C++Next

Open questions and future directions

Open questions

- Should `bit_value` be a template class to allow `bit_value&` to be implicitly convertible to `bit_reference`?
- Template and cv-qualifiers do not combine well for proxy classes such as bit abstractions: for example what should happen for a `const bit_value<volatile T>` or a `const bit_reference<T>`?
  $\Rightarrow$ If anyone has a clear view on that problem, please come or contact me!

Future directions (collaborations welcome!)

- Specialization of all relevant standard algorithms for bit iterators (for high performance and low latency computing)
- Implementation of bit ranges (Range proposal)
- Implementation of container adapters to reinterpret containers as static or dynamic bitsets
- Work on multiprecision integer arithmetic

Final words

Credits

The drawings are coming from the webcomic `xkcd` by Randall Munroe

Acknowledgments to the C++ community and in particular to

- Nathan Myers, Tomasz Kaminski, Lawrence Crowl, Howard Hinnant, Jens Maurer, Tony Van Eerd, Klemens Morgenstern, Vicente Botet Escriba, Odin Holmes and the other contributors of the ISO C++ Standard - Discussion and of the ISO C++ Standard - Future Proposals groups for their initial reviews and comments.
- The C++ Standards Committee
- The organizers of CPPCON 2016

Research acknowledgments

- Vincent Reverdy and Robert J. Brunner have been supported by the National Science Foundation Grant AST-1313415. Robert J. Brunner has been supported in part by the Center for Advanced Studies at the University of Illinois.
- Vincent Reverdy is thankfull to the Laboratoire Univers et Théories (LUTH), in France, where he started his early investigations on high performance trees.

Final words

This research has been done in/at

The Tree Building Blocks library team

↑

The Data Science Group (LCDM), under the supervision of Prof. Robert Brunner

↑

Department of Astronomy

↑

University of Illinois at Urbana-Champaign (UIUC)

Contact and links

- GitHub: `https://github.com/vreverdy/bit`
- Contact: `vince.rev@gmail.com`
- ISO C++ proposal: `P0237`
- Collaborations are very welcome! Same for comments! Same for benchmarks!