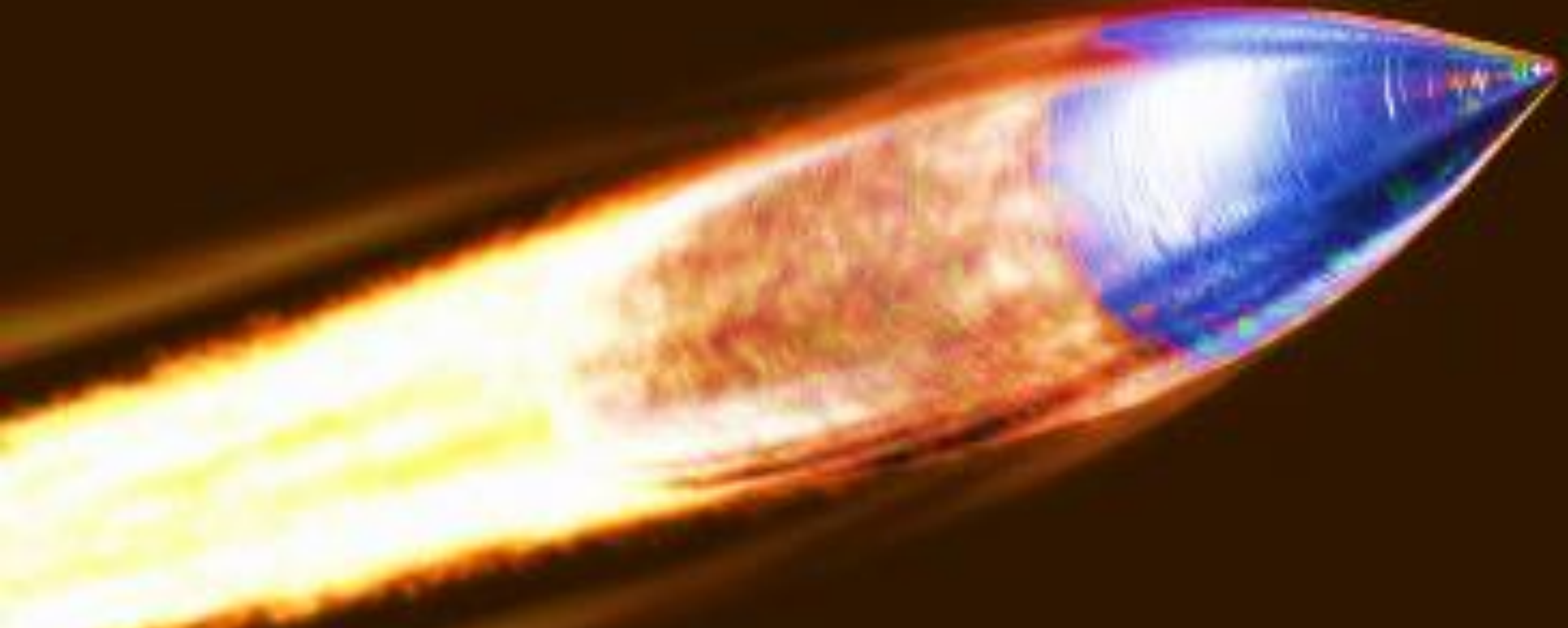
Fastware for C++



Scott Meyers, Ph.D.
Software Development Consultant
http://aristeia.com
smeyers@aristeia.com

Last Revised: 6/9/15

Achieving *Fastware* in C++

Requires knowledge and effective use of

- **Software engineering**
- **Computer science**
- **C++**

We'll address issues in each area.

Overview

Day 1 (Approximate):

- Speed as a correctness criterion.
- Optimizing systems rather than programs.
- CPU caches and why you care.
- Making effective use of C++.
  - Move semantics.
  - Avoiding unnecessary object creation.
  - Custom heap management.

Overview

Day 2 (Approximate):

- Making effective use of the STL.
  - reserve and shrink_to_fit.
  - Range member functions.
  - Function objects.
  - Sorted vectors.
  - Sorting algorithms.
- Concurrent data structures.
- Parallel algorithms.
- Exploiting “free” concurrency.
- PGO and WPO.
- Further information.

The Big Picture

Architecture & Design	Coding	Tuning
▪ Speed as a correctness criterion ▪ Optimizing systems rather than programs ▪ CPU caches ▪ Concurrent data structures ▪ Parallel algorithms ▪ Exploiting "free" concurrency ▪ Sorted vectors	▪ Move semantics ▪ Avoiding unnecessary object creation ▪ `reserve` and `shrink_to_fit` ▪ Range member functions ▪ Function objects ▪ Sorting algorithms	▪ CPU caches ▪ Concurrent data structures ▪ Parallel algorithms ▪ Exploiting "free" concurrency ▪ Custom heap management ▪ Sorted vectors ▪ Sorting algorithms ▪ PGO and WPO

Some C++ Vocabulary

- **C++98: Standard C++ prior to 2011.**
  - Minor 2003 revision known as C++03.
- **TR1: Augmented C++98/03 standard library functionality**
  - Approved in 2005.
  - Common compilers ship with most of TR1.
- **C++11: Standard C++ between 2011 and 2014.**
  - Library additions largely based on TR1.
  - Current compilers generally support most or all of C++11.
- **C++14: Current standard C++.**
  - Largely bug-fixes for C++11, but adds some new features.
- **Boost: Important repository for open-source C++ libraries.**
  - Basis for most of TR1.
    - Offers free cross-platform implementations (most parts).
  - boost.org.

Treat Speed as a Correctness Criterion

Performance concerns early in development often met with quotes:

- Donald Knuth:

  Premature optimization is the root of all evil.

- Michael A Jackson:

  The First Rule of Program Optimization: Don't do it.

- W.A. Wulf:

  More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason – including blind stupidity."

Treat Speed as a Correctness Criterion

Common advice: “First make it right, then make it fast.”

- Michael A Jackson:

  The Second Rule of Program Optimization (for experts only!):
  Don't do it yet.

Treat Speed as a Correctness Criterion

Problems:

- **Fast is a component of right.**
  - Right ≡ Acceptable.
  - Every system can be unacceptably slow.
    - Too slow ≡ Wrong.
- **Adding speed may call for fundamental redesigns:**
  - Different algorithms, data structures, control flow.
    - E.g., ST ⇒ MT.
    - E.g., Undistributed/nonscalable ⇒ distributed/scalable.
  - Donald Knuth again:

    Premature optimization is the root of all evil.

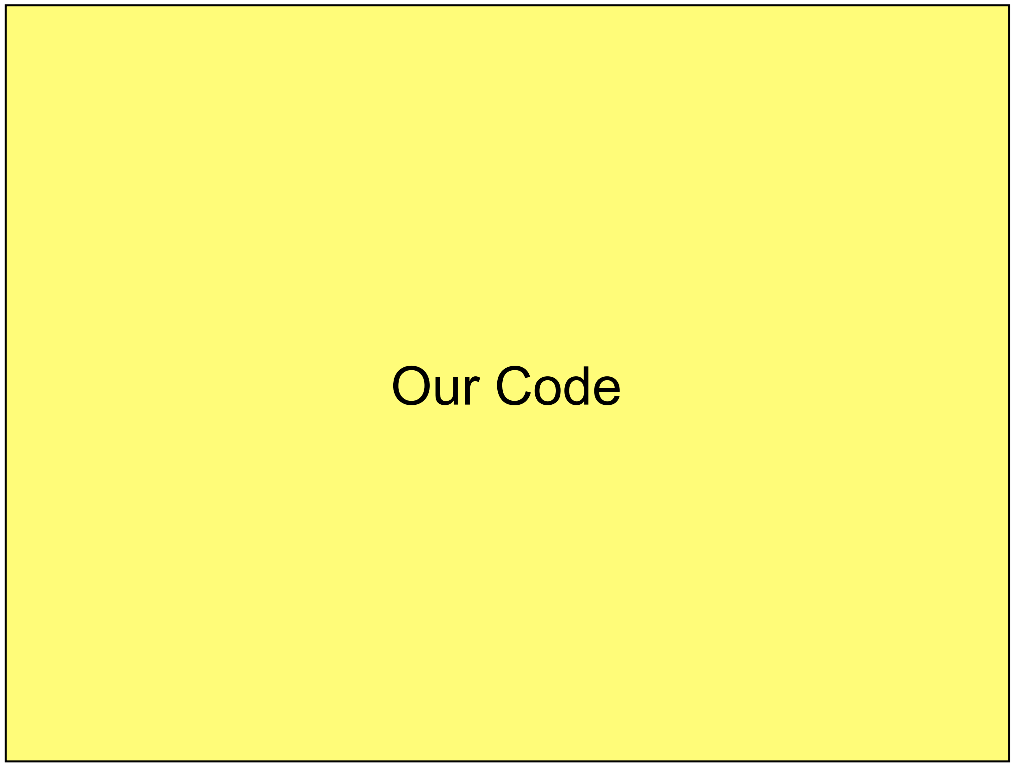
Guideline

Treat speed as a correctness criterion.

- Recognize its importance.
- Define it.
- Design for it.
- Verify it.

Optimize the System, not the Program

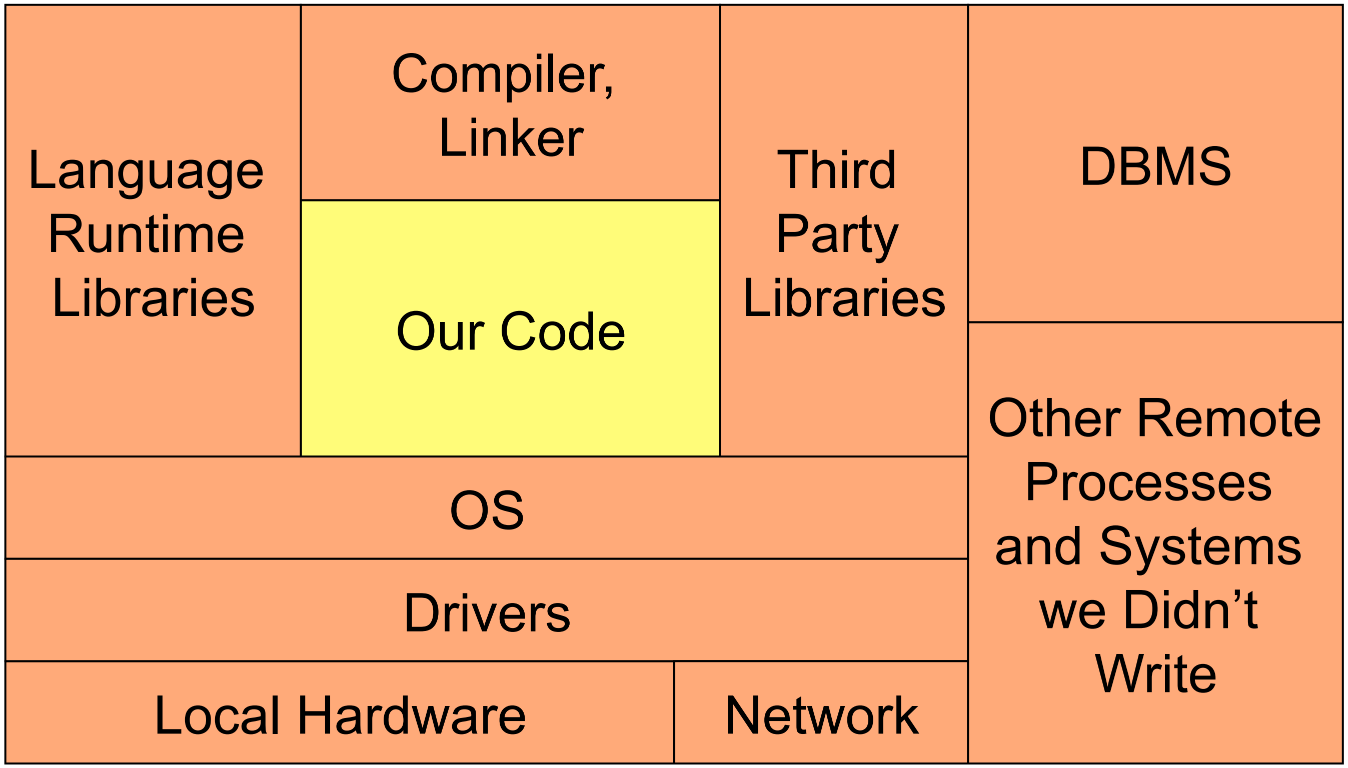
Naïve developer world view:



The System

Optimize the System, not the Program

More likely world:



The System

Optimize the System, not the Program

Implications:

- Before optimizing Our Code, ensure it's a bottleneck.
  - ➡ To reduce likelihood, use C++ appropriately.
- Optimizing rest of system requires indirect means:
  - ➡ Appropriate hardware usage (e.g., CPU caches).
  - ➡ Approprite API usage (e.g., STL, other libraries).
  - ➡ Appropriate tool usage (e.g., compiler).

Guideline

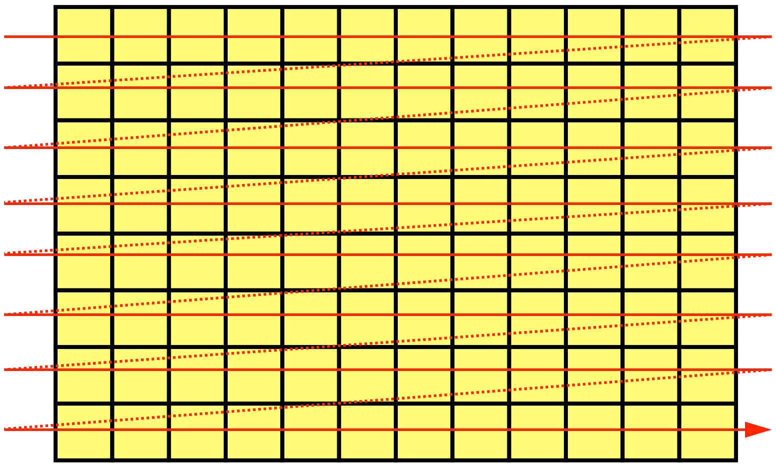
Optimize the system, not the program.

- Control "foreign" components via effective API use.
  - Requires deep understanding of components' APIs/behaviors.
- Minimize data-transfer latency.

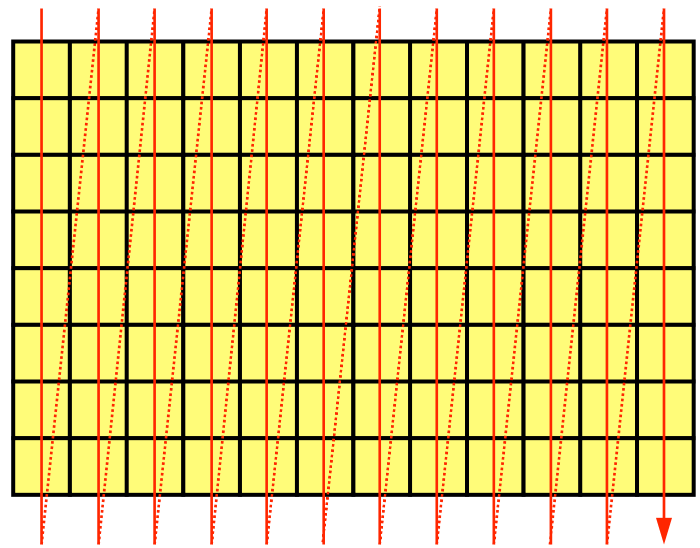
Understand the Importance of CPU Caches

Two ways to traverse a matrix:

- Each touches exactly the same memory.



Row Major



Column Major

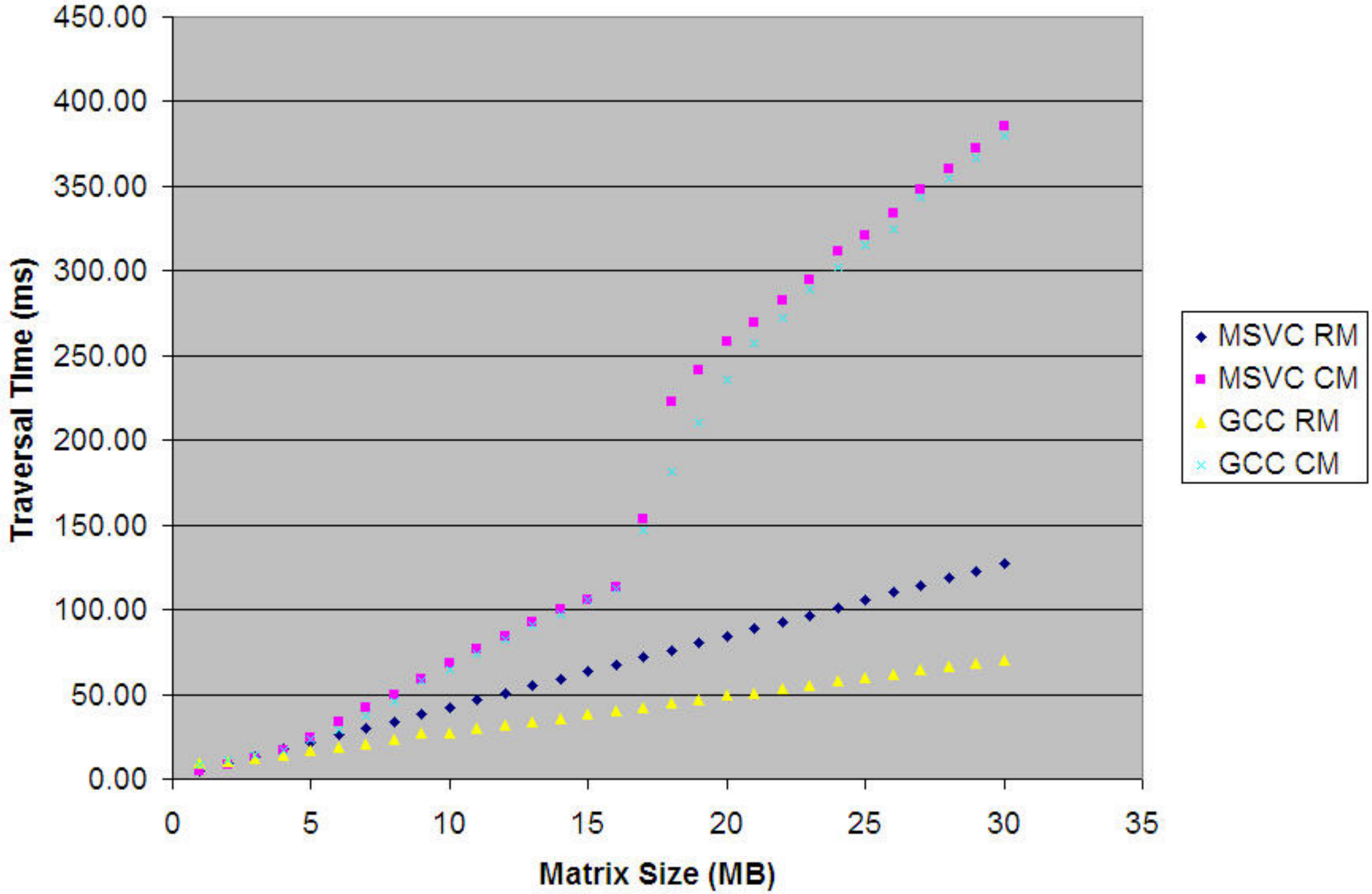
CPU Caches

Code very similar:

```
void sumMatrix(const Matrix<int>& m,
               long long& sum, TraversalOrder order)
{
  sum = 0;

  if (order == RowMajor) {
    for (unsigned r = 0; r < m.rows(); ++r) {
      for (unsigned c = 0; c < m.columns(); ++c) {
        sum += m[r][c];
      }
    }
  } else {
    for (unsigned c = 0; c < m.columns(); ++c) {
      for (unsigned r = 0; r < m.rows(); ++r) {
        sum += m[r][c];
      }
    }
  }
}
```

CPU Caches

Performance isn't:



CPU Caches

Traversal order matters.

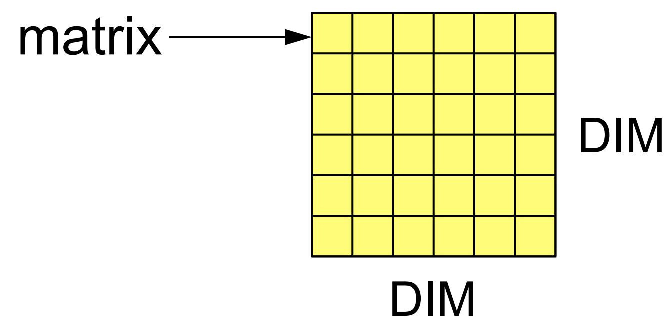
Why?

CPU Caches

Herb Sutter's scalability issue in counting odd matrix elements.

- Square matrix of side DIM with memory in array matrix.
- Sequential pseudocode:

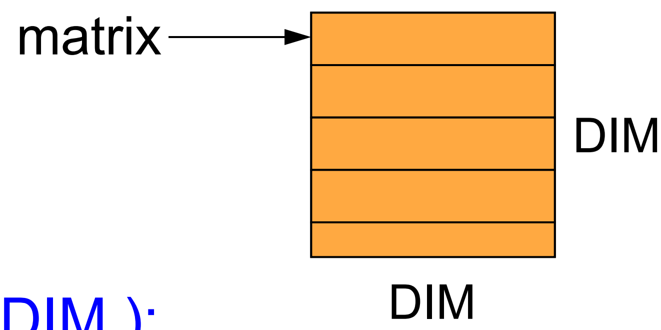
```
int odds = 0;
for( int i = 0; i < DIM; ++i )
   for( int j = 0; j < DIM; ++j )
      if( matrix[i*DIM + j] % 2 != 0 )
         ++odds;
```



CPU Caches

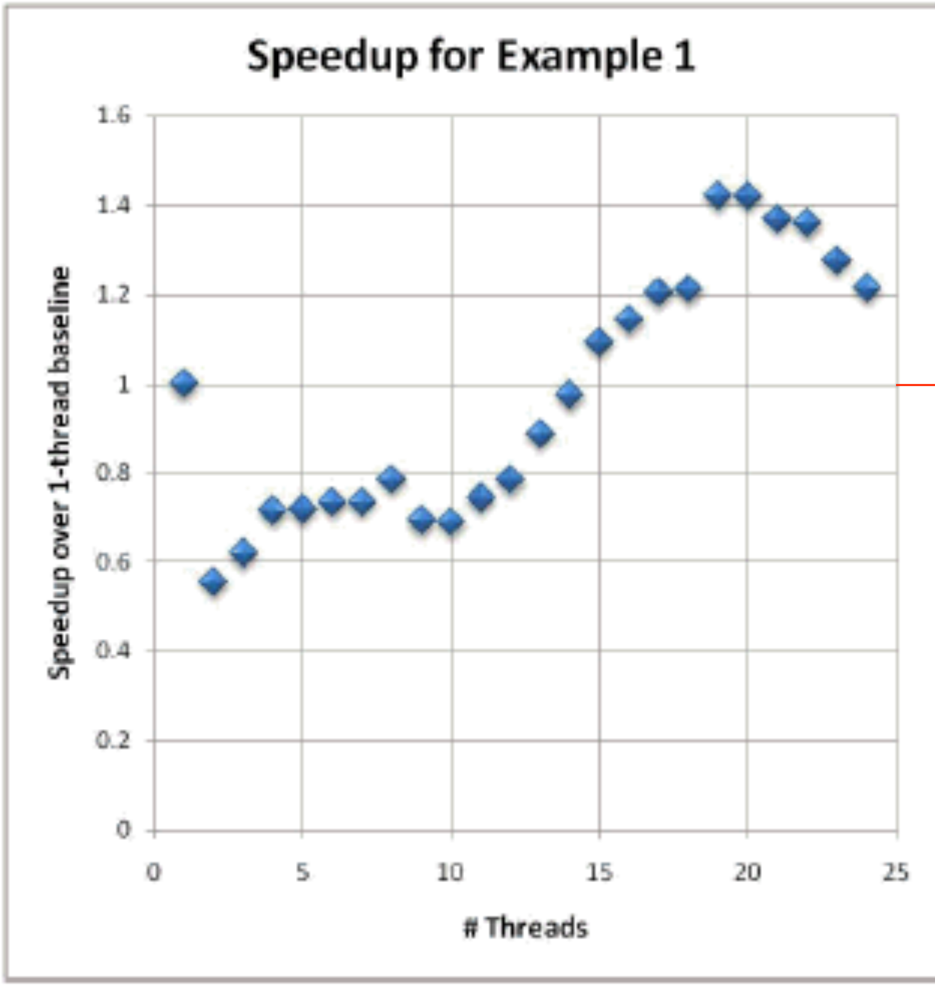
- Parallel pseudocode, take 1:

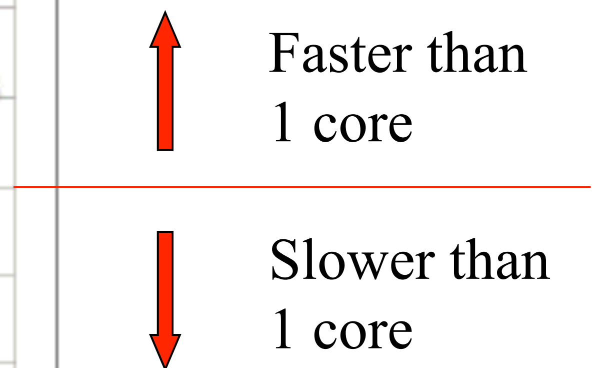
```
int result[P];

// Each of P parallel workers processes 1/P-th of the data;
// the p-th worker records its partial count in result[p]
for (int p = 0; p < P; ++p )
  pool.run( [&,p] {
    result[p] = 0;
    int chunkSize = DIM/P + 1;
    int myStart = p * chunkSize;
    int myEnd = min( myStart+chunkSize, DIM );
    for( int i = myStart; i < myEnd; ++i )
      for( int j = 0; j < DIM; ++j )
        if( matrix[i*DIM + j] % 2 != 0 )
          ++result[p]; } );

pool.join();                      // Wait for all tasks to complete

odds = 0;                         // combine the results
for( int p = 0; p < P; ++p )
  odds += result[p];
```



CPU Caches

Scalability unimpressive:





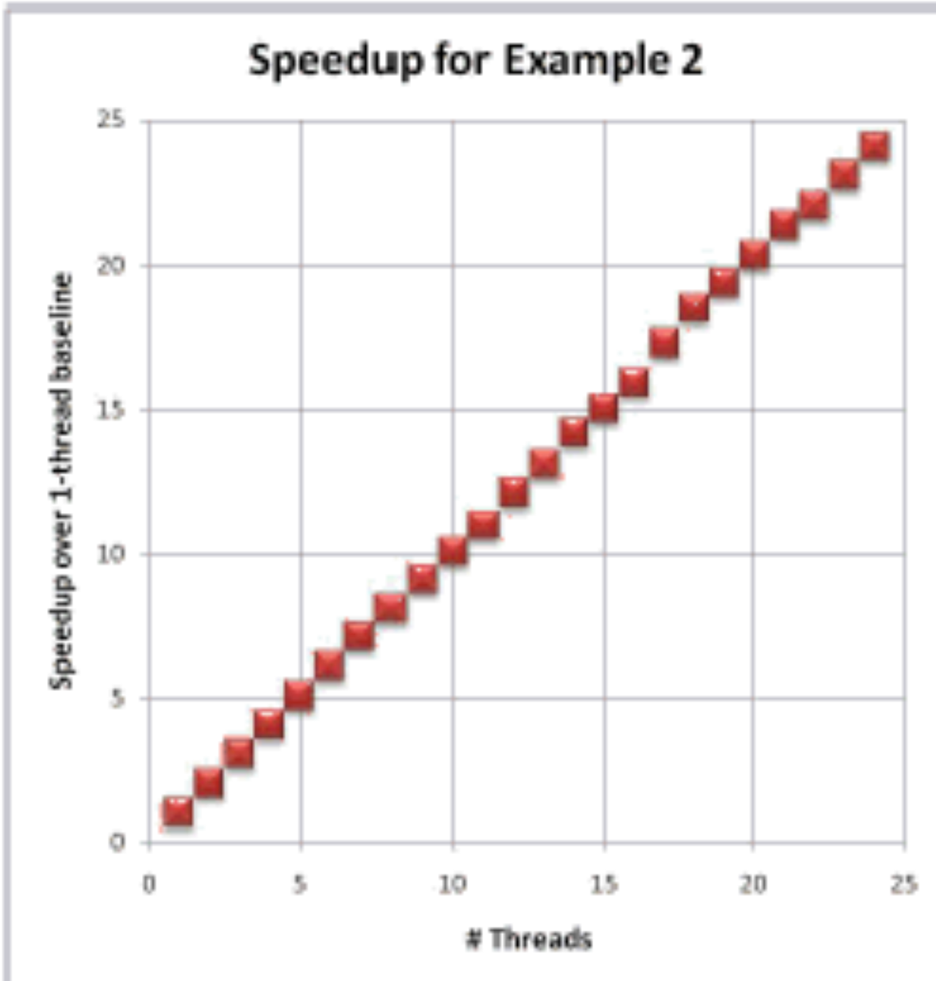
CPU Caches

- Parallel pseudocode, take 2:

```
int result[P];

for (int p = 0; p < P; ++p )
  pool.run( [&,p] {
    int count = 0;                          // instead of result[p]
    int chunkSize = DIM/P + 1;
    int myStart = p * chunkSize;
    int myEnd = min( myStart+chunkSize, DIM );
    for( int i = myStart; i < myEnd; ++i )
      for( int j = 0; j < DIM; ++j )
        if( matrix[i*DIM + j] % 2 != 0 )
          ++count;                          // instead of result[p]
    result[p] = count; } );                 // new statement

...          // nothing else changes
```

CPU Caches

Scalability now perfect!



CPU Caches

Thread memory access matters.

Why?

CPU Caches

Small amounts of unusually fast memory.

- Generally hold contents of recently accessed memory locations.
- Access latency much smaller than for main memory.

CPU Caches

Three common types:

- Data (D-cache, D$)
- Instruction (I-cache, I$)
- Translation lookaside buffer (TLB)
  - ➡ Caches virtual→real address translations

Voices of Experience

Sergey Solyanik (from Microsoft):

> Linux was routing packets at ~30Mbps [wired], and wireless at ~20. Windows CE was crawling at barely 12Mbps wired and 6Mbps wireless. ...
>
> We found out Windows CE had a LOT more instruction cache misses than Linux. ...
>
> After we changed the routing algorithm to be more cache-local, we started doing 35MBps [wired], and 25MBps wireless - 20% better than Linux.

Voices of Experience

Jan Gray (from the MS CLR Performance Team):

> If you are passionate about the speed of your code, it is imperative that you consider ... the cache/memory hierarchy as you design and implement your algorithms and data structures.

Dmitriy Vyukov (developer of Relacy Race Detector):

> Cache-lines are the key! Undoubtedly! If you will make even single error in data layout, you will get 100x slower solution! No jokes!

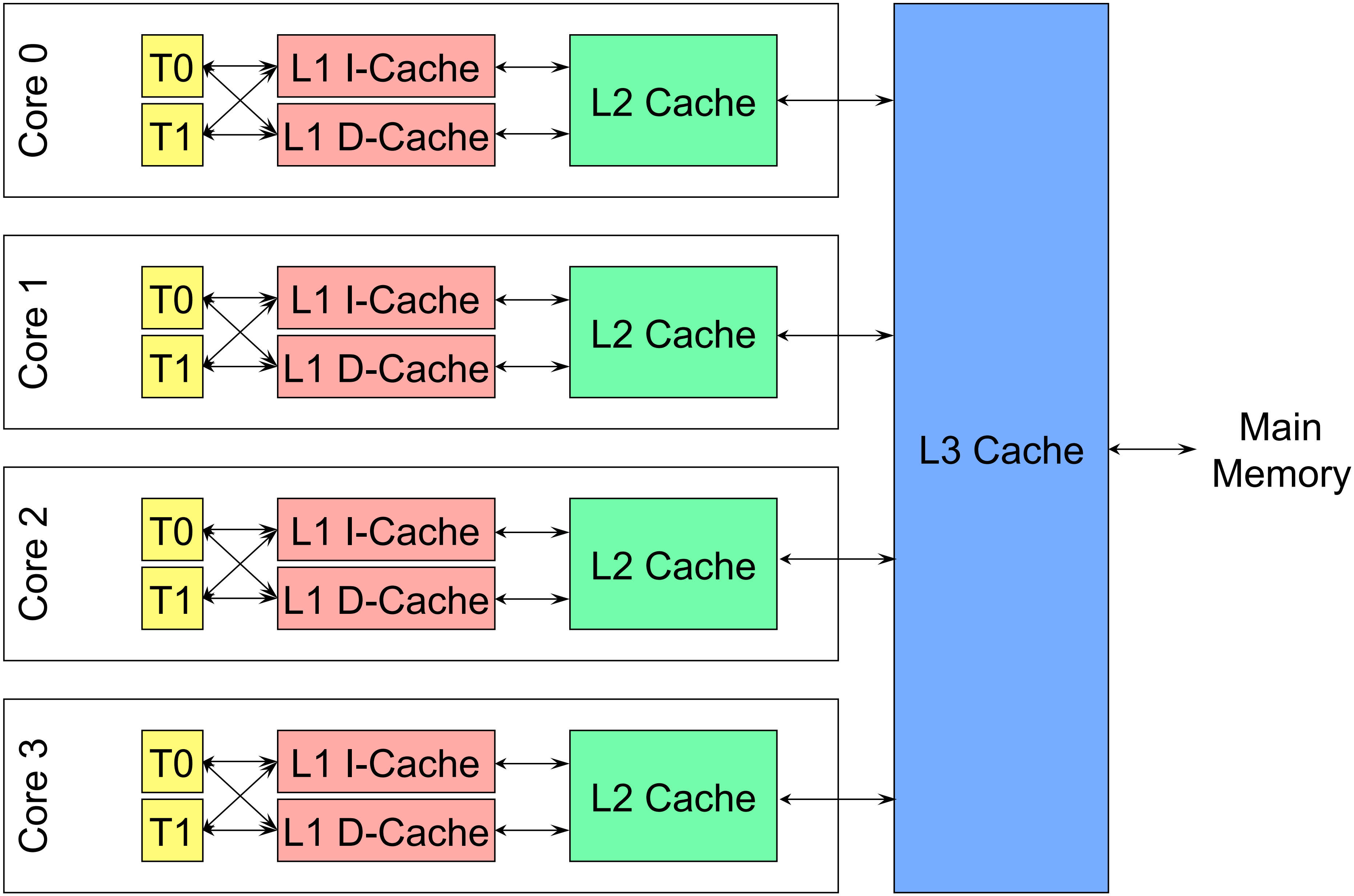
Cache Hierarchies

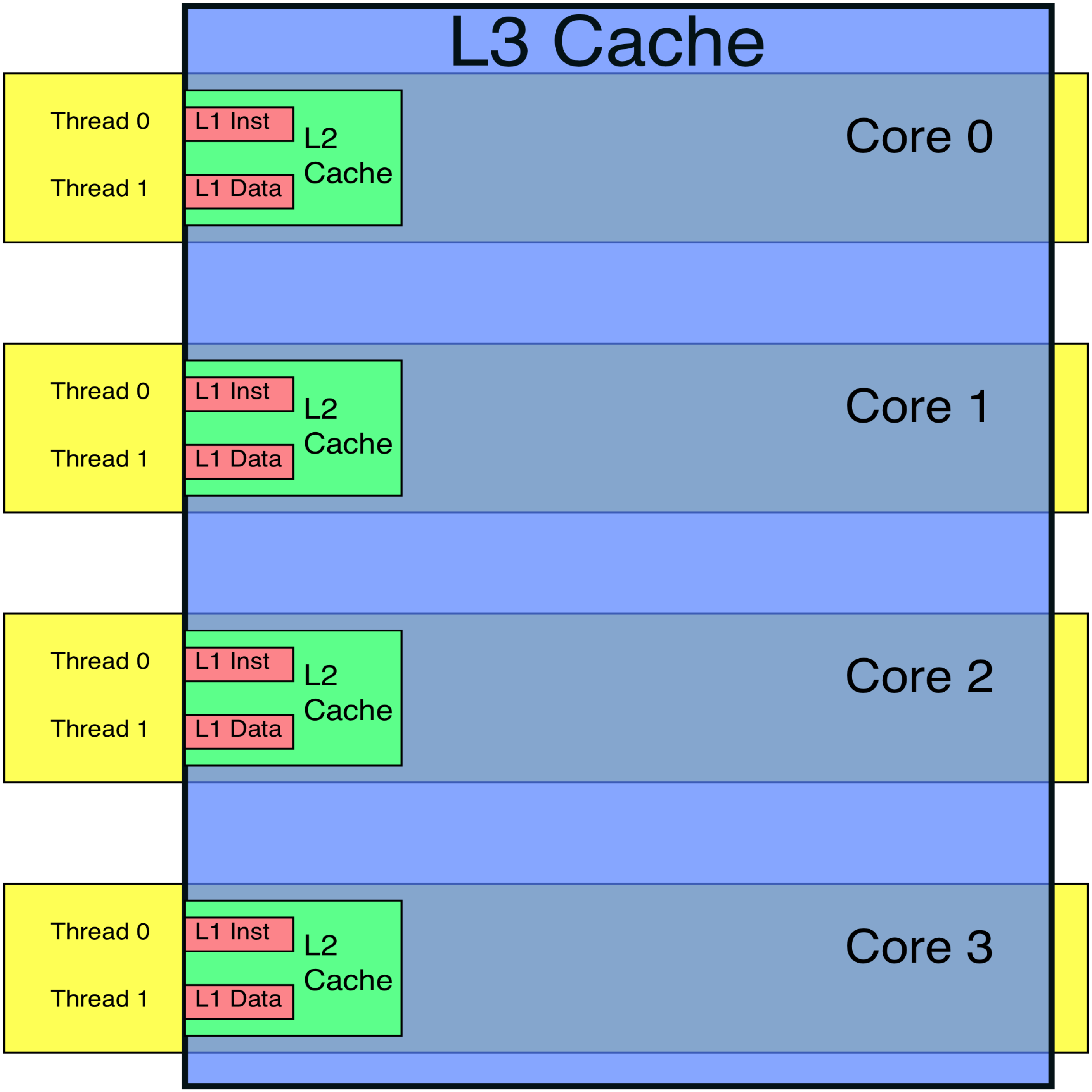
Cache hierarchies (*multi-level caches*) are common.

E.g., Intel Core i7-9xx processor:

- 32KB L1 I-cache, 32KB L1 D-cache per core
  - Shared by 2 HW threads
- 256 KB L2 cache per core
  - Holds both instructions and data
  - Shared by 2 HW threads
- 8MB L3 cache
  - Holds both instructions and data
  - Shared by 4 cores (8 HW threads)

Core i7-9xx Cache Hierarchy



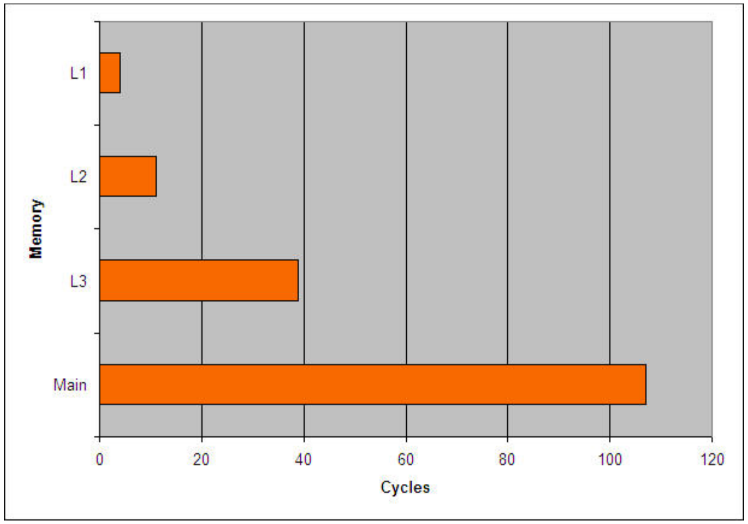

L3 Cache
Thread 0
Thread 1
L1 Inst
L1 Data
L2
Cache
Core 0
Thread 0
Thread 1
L1 Inst
L1 Data
L2
Cache
Core 1
Thread 0
Thread 1
L1 Inst
L1 Data
L2
Cache
Core 2
Thread 0
Thread 1
L1 Inst
L1 Data
L2
Cache
Core 3

CPU Cache Characteristics

Caches are small.

- Assume 100MB program at runtime (code + data).
  - ➡ 8% fits in core-i79xx's L3 cache.
    - ◆ L3 cache shared by *every running process* (incl. OS).
  - ➡ 0.25% fits in each L2 cache.
  - ➡ 0.03% fits in each L1 cache.

Caches much faster than main memory.



- For Core i7-9xx:
  - ➡ L1 latency is 4 cycles.
  - ➡ L2 latency is 11 cycles.
  - ➡ L3 latency is 39 cycles.
  - ➡ Main memory latency is 107 cycles.
    - ◆ 27 times slower than L1!
    - ◆ 100% CPU utilization ⇒ >99% CPU idle time!

Effective Memory = CPU Cache Memory

From speed perspective, total memory = total cache.

- Core i7-9xx has 8MB fast memory for *everything*.
  - ➡ Everything in L1 and L2 caches also in L3 cache.
- Non-cache access can slow things by orders of magnitude.

Small ≡ fast.

- No time/space tradeoff at hardware level.
- Compact, well-localized code that fits in cache is fastest.
- Compact data structures that fit in cache are fastest.
- Data structure traversals touching only cached data are fastest.

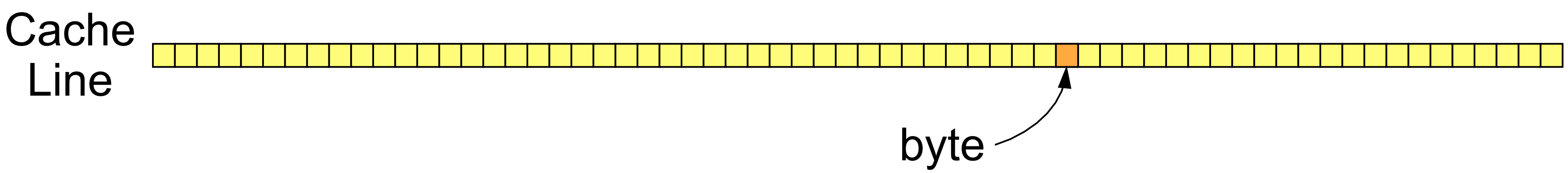
Cache Lines

Caches consist of *lines*, each holding multiple adjacent words.

- On Core i7, cache lines hold 64 bytes.
  - ➨ 64-byte lines common for Intel/AMD processors.
  - ➨ 64 bytes = 16 32-bit values, 8 64-bit values, etc.
    - ◆ E.g., 16 32-bit array elements.

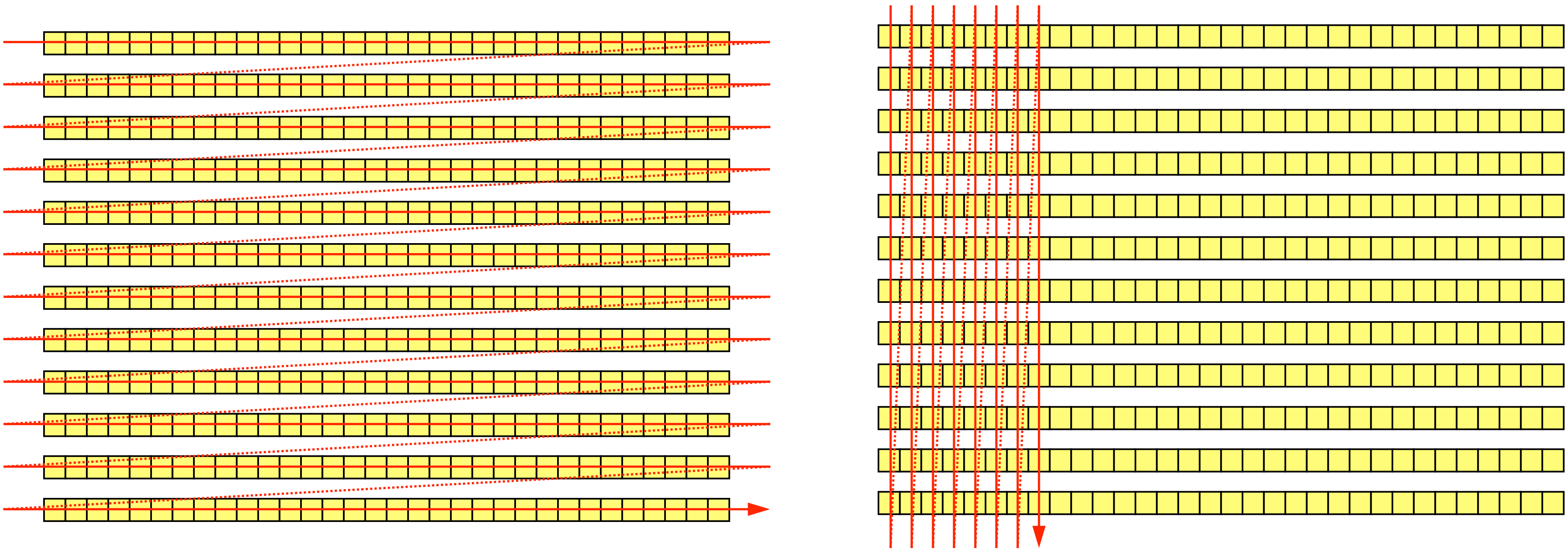
Main memory read/written in terms of cache lines.

- Read byte not in cache ⇒ read full cache line from main memory.
- Write byte ⇒ write full cache line to main memory (eventually).



Cache Lines

Explains why row-major matrix traversal better than column-major:



Cache Line Prefetching

Hardware speculatively prefetches cache lines:

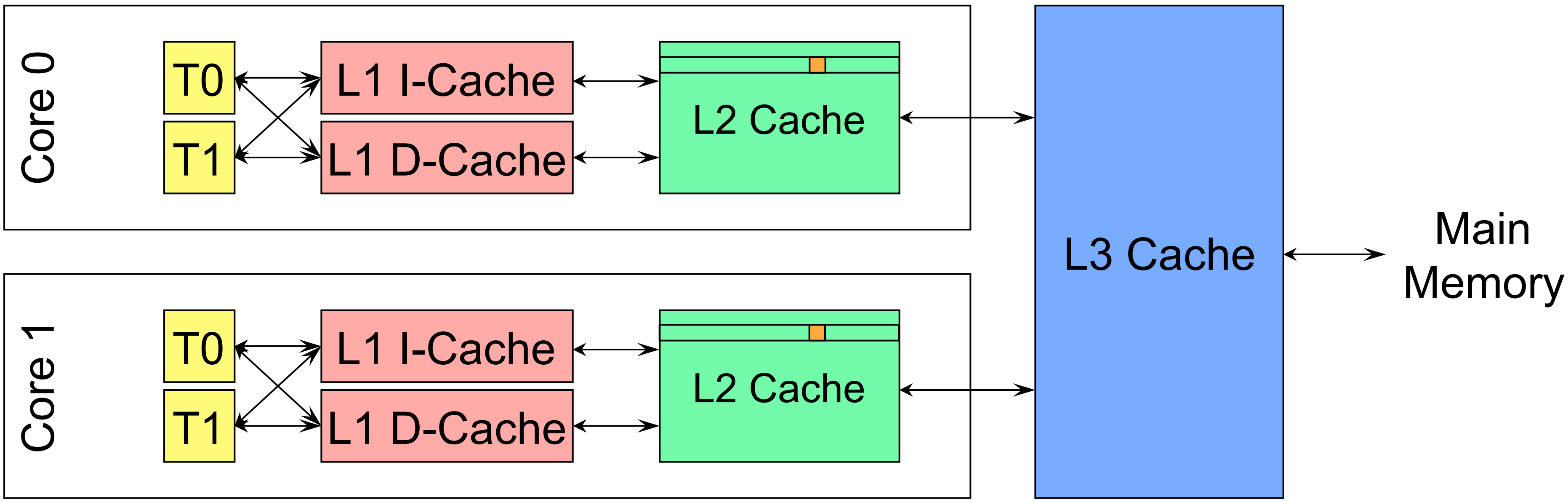
- Forward traversal through cache line *n* ⇒ prefetch line *n+1*
- Reverse traversal through cache line *n* ⇒ prefetch line *n-1*

Implications

- Locality counts.
  - Reads/writes at address $A \Rightarrow$ contents near A already cached.
    - E.g., on the same cache line.
    - E.g., on nearby cache line that was prefetched.
- Predictable access patterns count.
  - "Predictable" $\cong$ forward or backwards traversals.
- Linear array traversals *very* cache-friendly.
  - Excellent locality, predictable traversal pattern.
  - Linear array search can beat $log_2\ n$ searches of heap-based BSTs.
  - $log_2\ n$ binary search of sorted array can beat $O(1)$ searches of heap-based hash tables.
  - Big-Oh wins for large n, but hardware caching takes early lead.

Cache Coherency

From core i7's architecture:



Assume both cores have cached the value at (virtual) address *A*.

- Whether in L1 or L2 makes no difference.

Consider:

- Core 0 writes to *A*.
- Core 1 reads *A*.

What value does Core 1 read?

Cache Coherency

Caches a latency-reducing optimization:

- There's only one virtual memory location with address *A*.
- It has only one value.

Hardware invalidates Core 1's cached value when Core 0 writes to *A*.

- It then puts the new value in Core 1's cache(s).

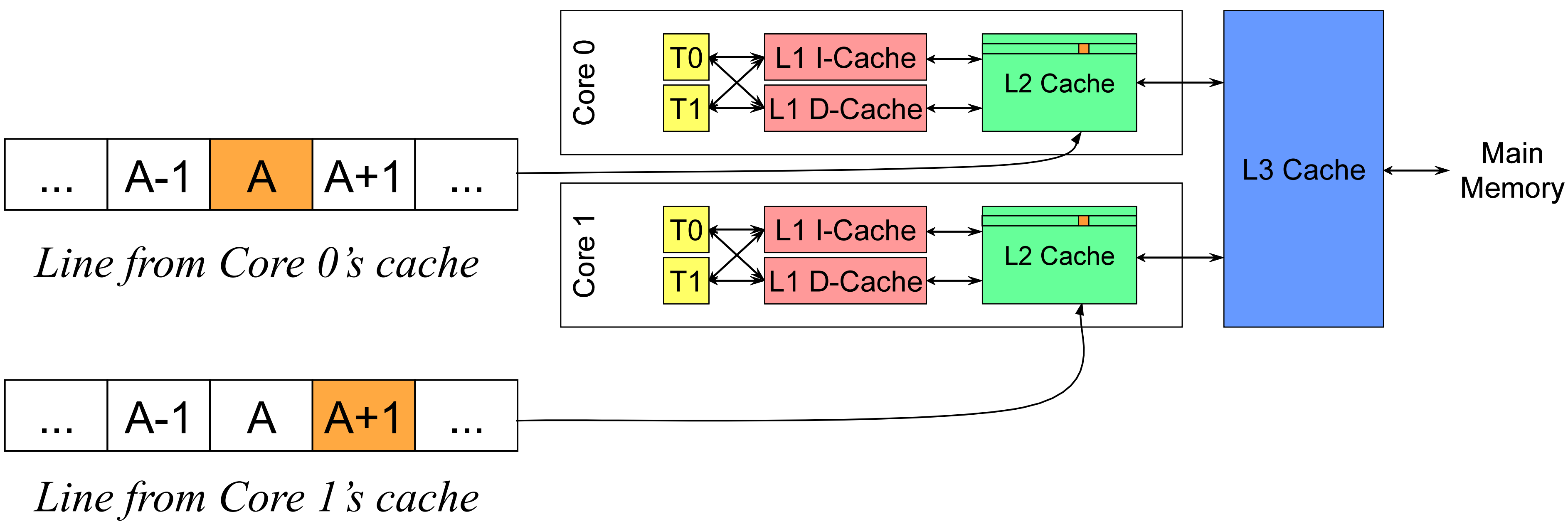
Happens automatically.

- You need not worry about it.
  - Provided you synchronize access to shared data...
- But it takes time.

False Sharing

Suppose Core 0 accesses *A* and Core 1 accesses *A+1*.

- *Independent* pieces of memory; concurrent access is safe.
- But *A* and *A+1* probably map to the same cache line.
  - ➨ If so, Core 0's writes to *A* invalidates *A+1*'s cache line in Core 1.
    - ◆ And vice versa.
    - ◆ This is *false sharing*.



False Sharing

It explains Herb Sutter's issue:

```
int result[P];                        // many elements on 1 cache line

for (int p = 0; p < P; ++p )
  pool.run( [&,p] {                   // run P threads concurrently
    result[p] = 0;
    int chunkSize = DIM/P + 1;
    int myStart = p * chunkSize;
    int myEnd = min( myStart+chunkSize, DIM );
    for( int i = myStart; i < myEnd; ++i )
      for( int j = 0; j < DIM; ++j )
        if( matrix[i*DIM + j] % 2 != 0 )
          ++result[p]; } );           // each repeatedly accesses the
                                      // same array (albeit different
                                      // elements)
```

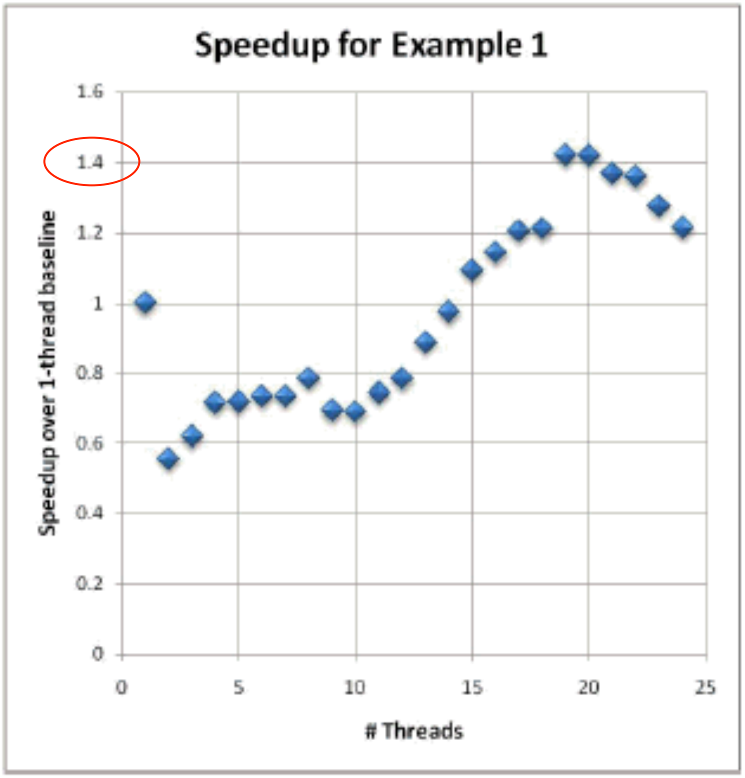
False Sharing

And his solution:

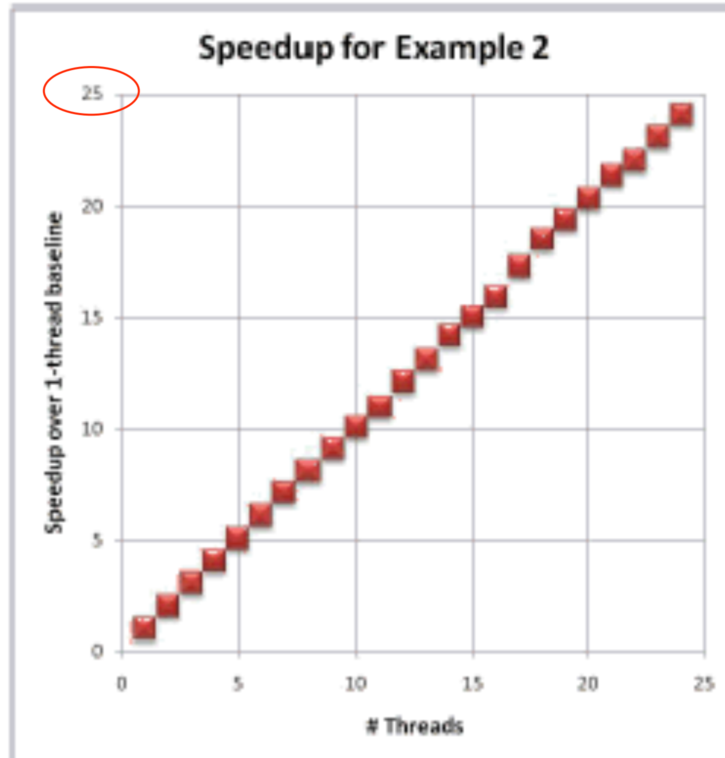
```
int result[P];                          // still multiple elements per
                                        // cache line

for (int p = 0; p < P; ++p )
  pool.run( [&,p] {
    int count = 0;                      // use local var for counting
      int chunkSize = DIM/P + 1;
    int myStart = p * chunkSize;
    int myEnd = min( myStart+chunkSize, DIM );
    for( int i = myStart; i < myEnd; ++i )
      for( int j = 0; j < DIM; ++j )
        if( matrix[i*DIM + j] % 2 != 0 )
          ++count;                      // update local var
    result[p] = count; } );             // access shared cache line
                                        // only once
```

False Sharing

His scalability results are worth repeating:



With False Sharing



Without False Sharing

False Sharing

Problems arise only when **all** are true:

- Independent values/variables fall on one cache line.
- Different cores concurrently access that line.
- Frequently.
- At least one is a writer.

All types of data are susceptible:

- Statically allocated (e.g., globals, statics).
- Heap allocated.
- Automatics and thread-locals (if pointers/references handed out).

Voice of Experience

Joe Duffy at Microsoft:

> During our Beta1 performance milestone in Parallel Extensions, most of our performance problems came down to stamping out false sharing in numerous places.

Summary

- **Small ≡ fast.**
  - ➡ No time/space tradeoff in the hardware.
- **Locality counts.**
  - ➡ Stay in the cache.
- **Predictable access patterns count.**
  - ➡ Be prefetch-friendly.

Guidance

For data:

- **Where practical, employ linear array traversals.**
  - ➡ "I don't know [data structure], but I know an array will beat it."
- **Use as much of a cache line as possible.**
  - ➡ Bruce Dawson's antipattern (from reviews of video games):

```
struct Object {                         // assume sizeof(Object) ≥ 64
  bool isLive;                          // possibly a bit field
  ...
};

std::vector<Object> objects;                            // or an array

for (std::size_t i = 0; i < objects.size(); ++i) {      // pathological if
  if (objects[i].isLive)                                // most objects
    doSomething();                                      // not alive
}
```

- **Be alert for false sharing in MT systems.**

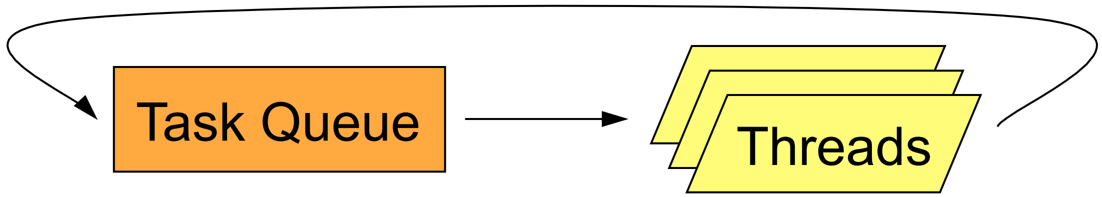
Guidance

For code:

- **Fit working set in cache.**
  - Avoid iteration over heterogeneous sequences with virtual calls.
    - E.g., sort sequences by type.
- **Make “fast paths” branch-free sequences.**
  - Use up-front conditionals to screen out “slow” cases.
- **Inline cautiously:**
  - The good:
    - Reduces branching.
    - Facilitates code-reducing optimizations.
  - The bad:
    - Code duplication reduces effective cache size.
- **Take advantage of PGO and WPO.**
  - Can automate some of above.

Example: Cache-Aware Design

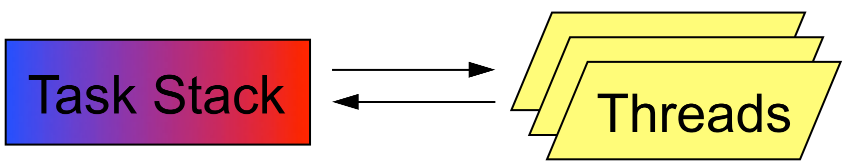
First cut at a thread scheduler:



Child tasks often exhibit locality wrt their parents.

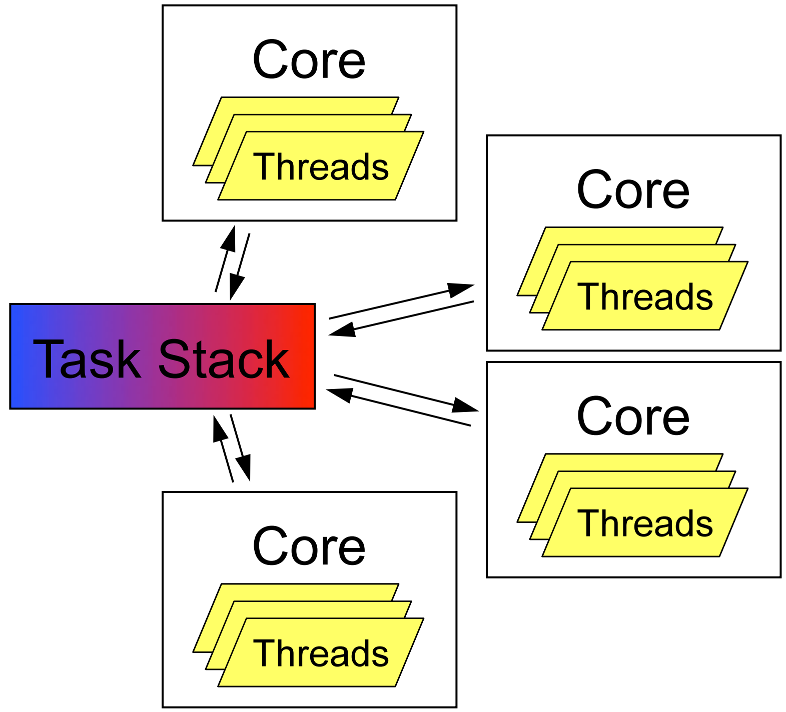
- Divide-and-conquer algs run same code on data subset.
  - Same code ⇒ I$ locality.
  - Data subset ⇒ D$ locality.

Task *stack* often cache-friendlier:



Example: Cache-Aware Design

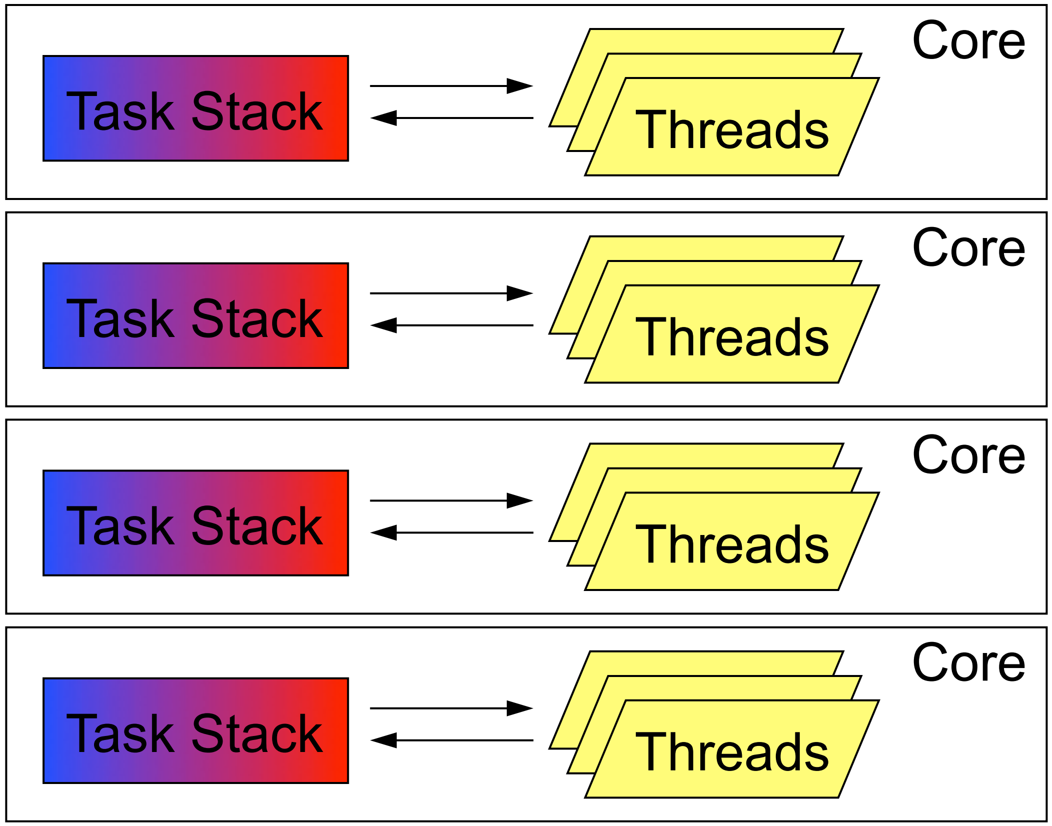
Global task stack likely scalability bottleneck with multiple cores.



- Each core reads/writes stack.
  - ➡ Shared data ⇒ Cost of cache coherency.
- Non-interfering readers/writers could cause false sharing.

Example: Cache-Aware Design

Per-core stacks avoid real and false sharing:



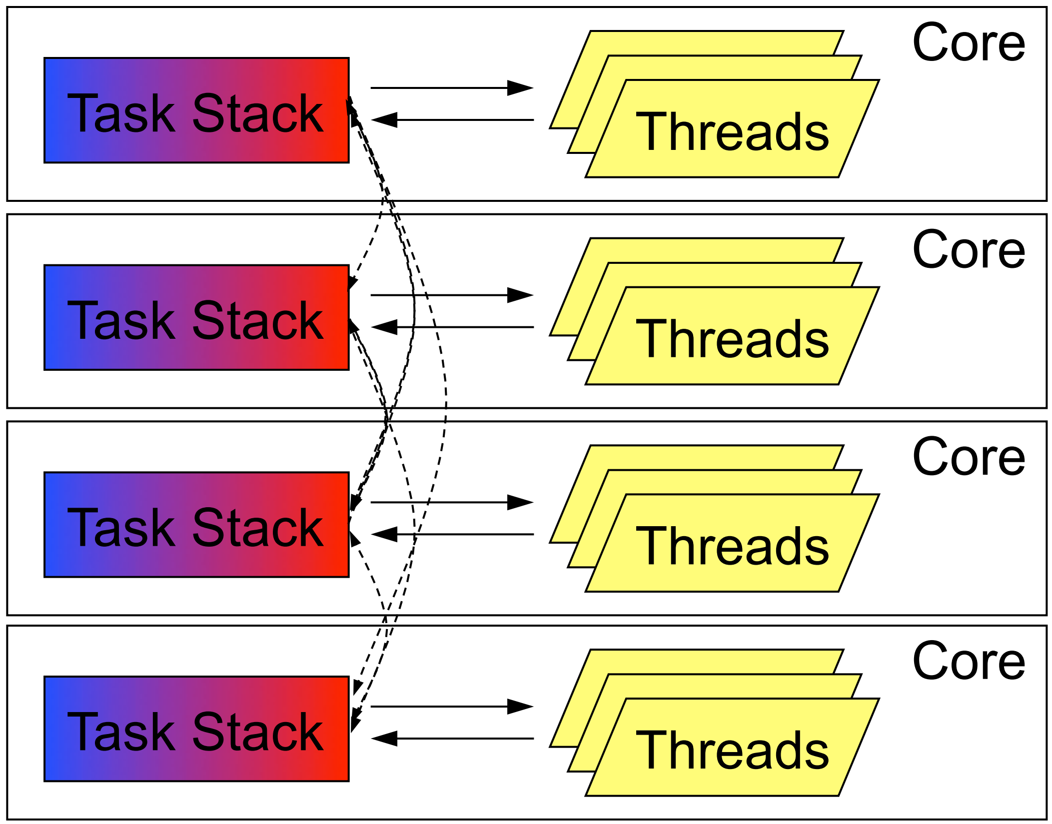
Load-balancing now a problem.

- Core’s stack empty ⇒ core sits idle.

Example: Cache-Aware Design

Work-stealing addresses that problem:

- Empty stack steals a task from a randomly-chosen stack:

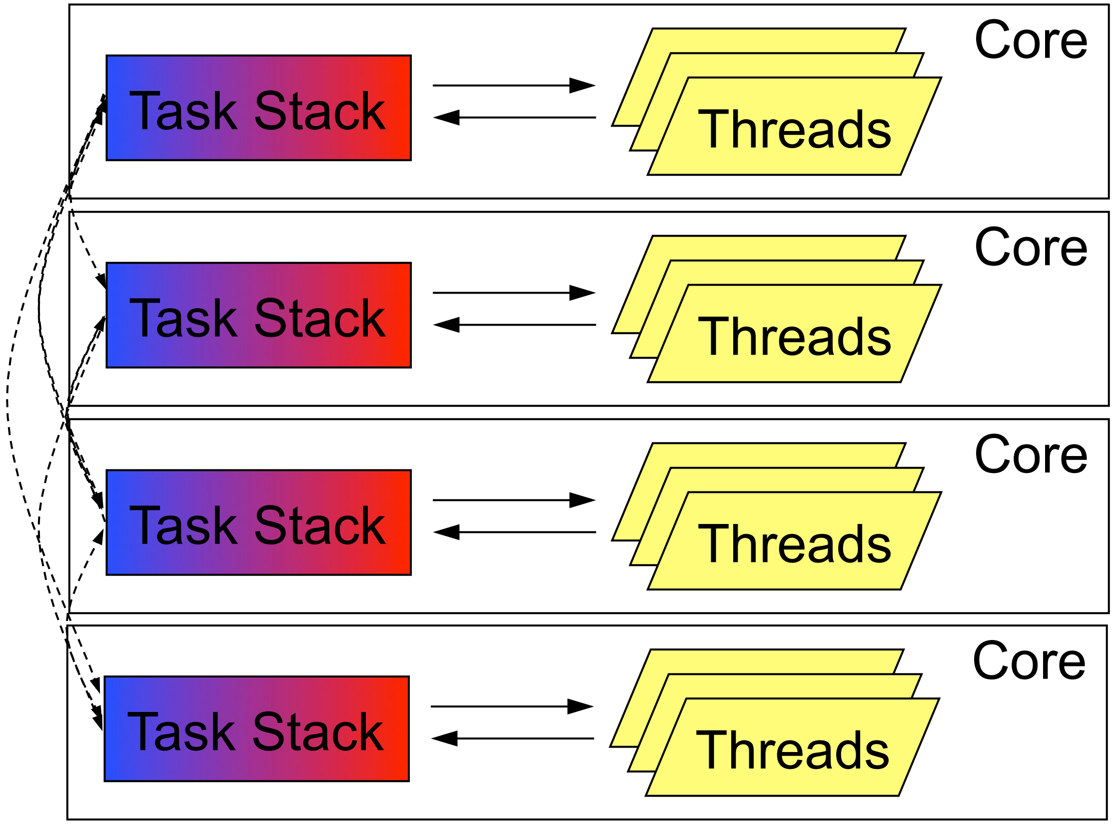


But stealing from top of stack cache-hostile:

- Code/data for task there probably warmest in victim's caches.

Example: Cache-Aware Design

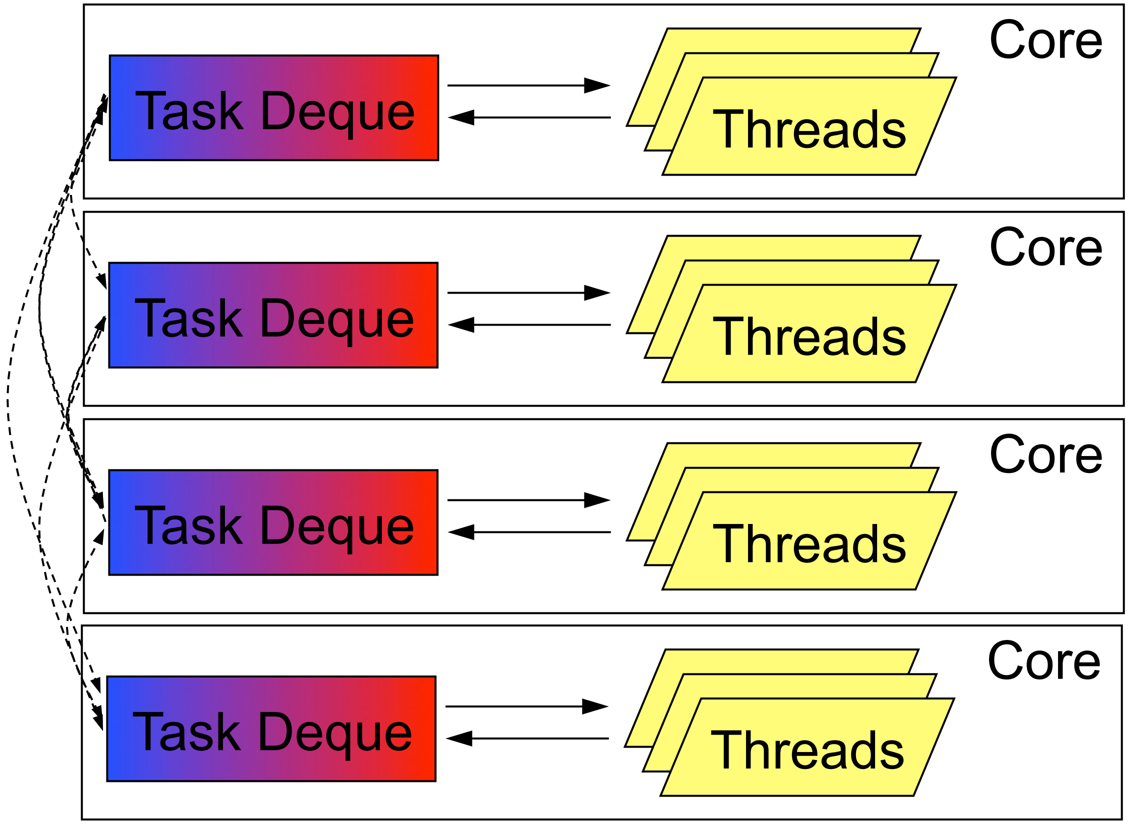
Better to steal from bottom of stack:



But stacks don't support "pop-off-bottom" functionality.

Example: Cache-Aware Design

Deques do:

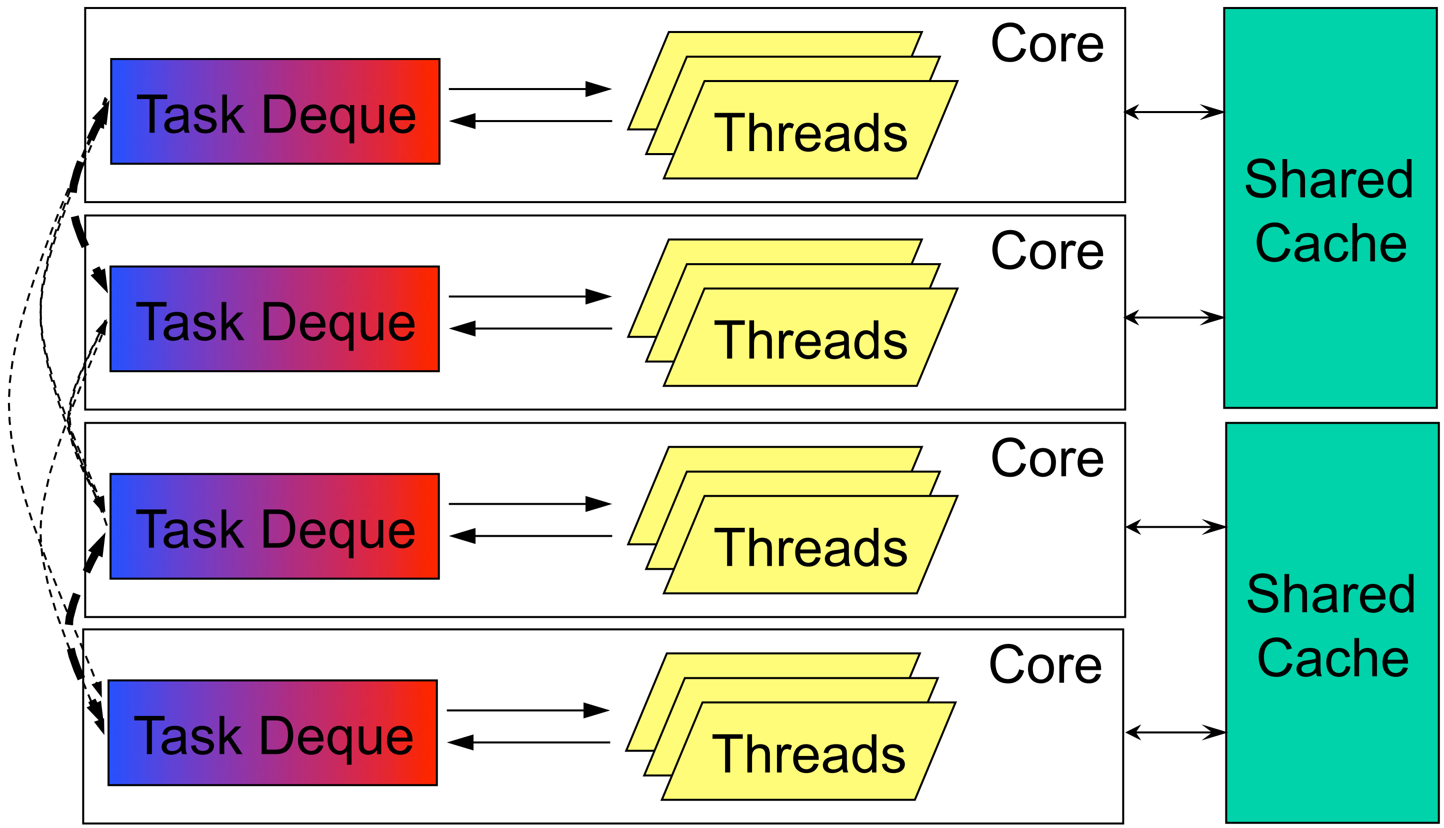


But random stealing ignores cache topologies.

- Different cores may share different higher-level caches.
  - E.g., each core pair in Intel Core 2 Quad shares an L2 cache.
    - Unlike Intel Core i7-9xx, where caches are per-core or global.
  - Multiprocessors.

Example: Cache-Aware Design

Theft from a core with a lower-level shared cache preferable.



Summary: Cache-Aware Design Example

Cache issues affect data structures and algorithms:

- Task stack better than task queue.
- Per-core stack better than global stack.
- Task deque better than task stack for work-stealing.
- Preferable to steal from deques sharing same cache.

Cache considerations not the *only* considerations.

- In this example, others include load balancing and contention minimization.

Beyond Surface-Scratching

Cache-related topics not really addressed:

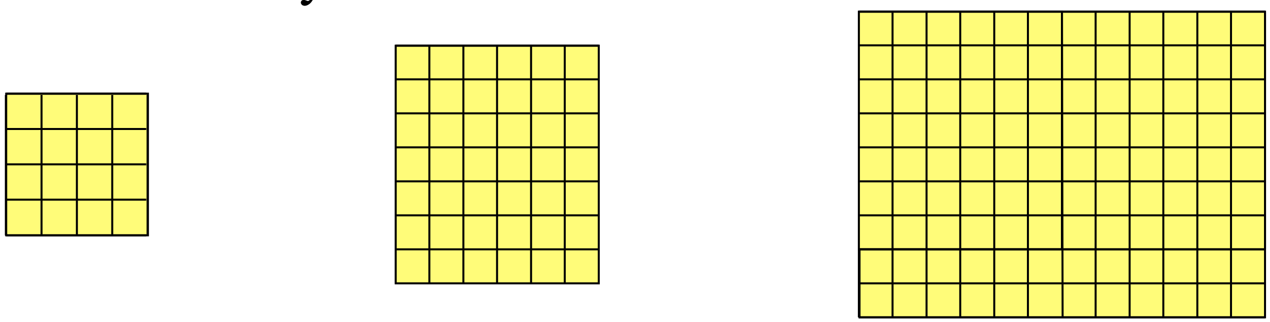
- Other cache technology issues:
  - Memory banks.
  - Associativity (but wait...).
  - Inclusive vs. exclusive content.
- Latency-hiding techniques.
  - Hyperthreading.
- Cache performance evaluation:
  - Why it's critical.
  - Why it's hard.
  - Tools that can help.
- Cache-oblivious algorithm design.

Beyond Surface-Scratching

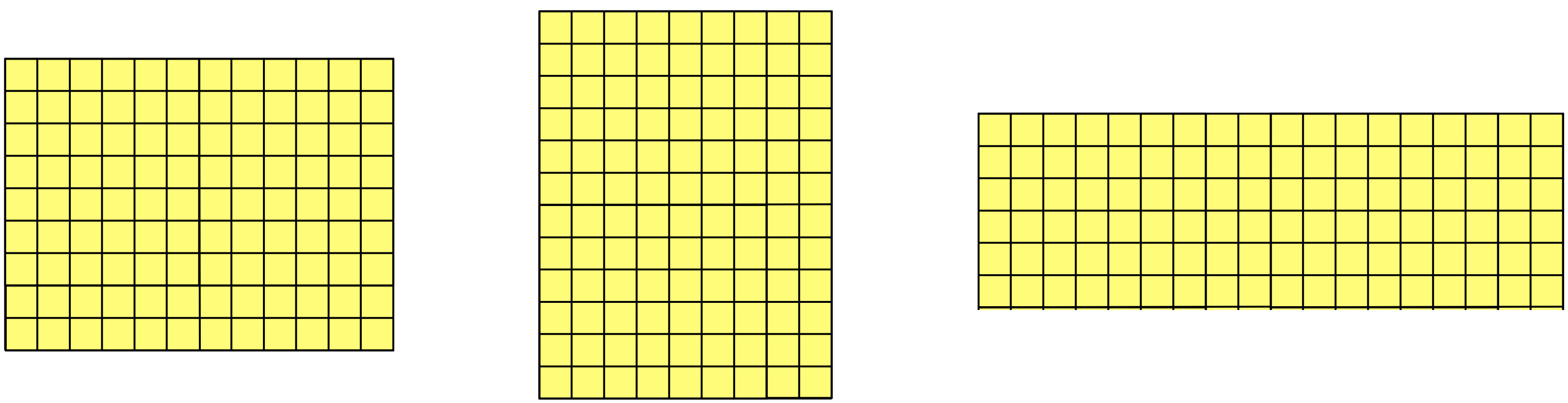
Overall cache behavior can be counterintuitive.

Matrix traversal redux:

- Matrix size can vary.

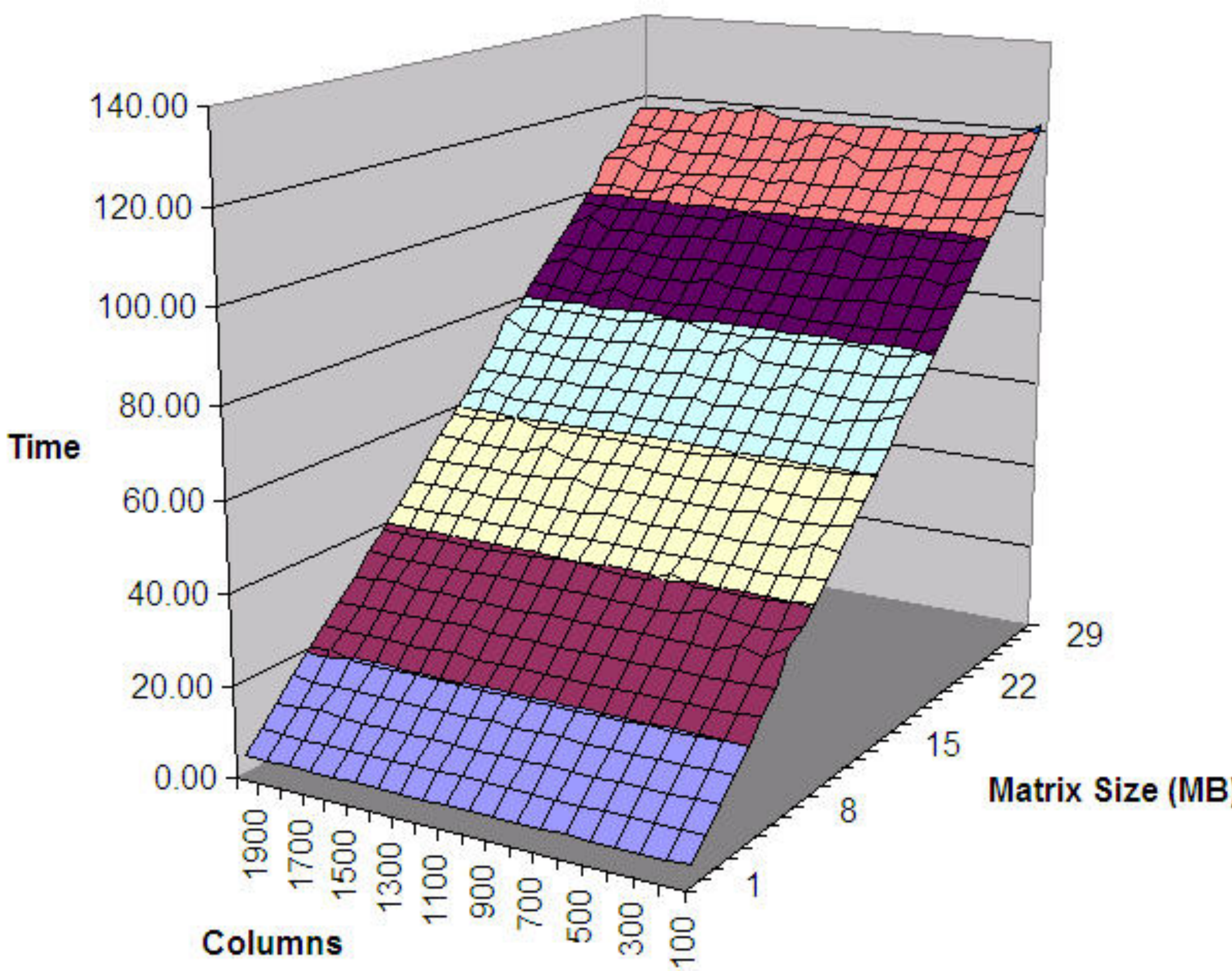


- For given size, shape can vary:



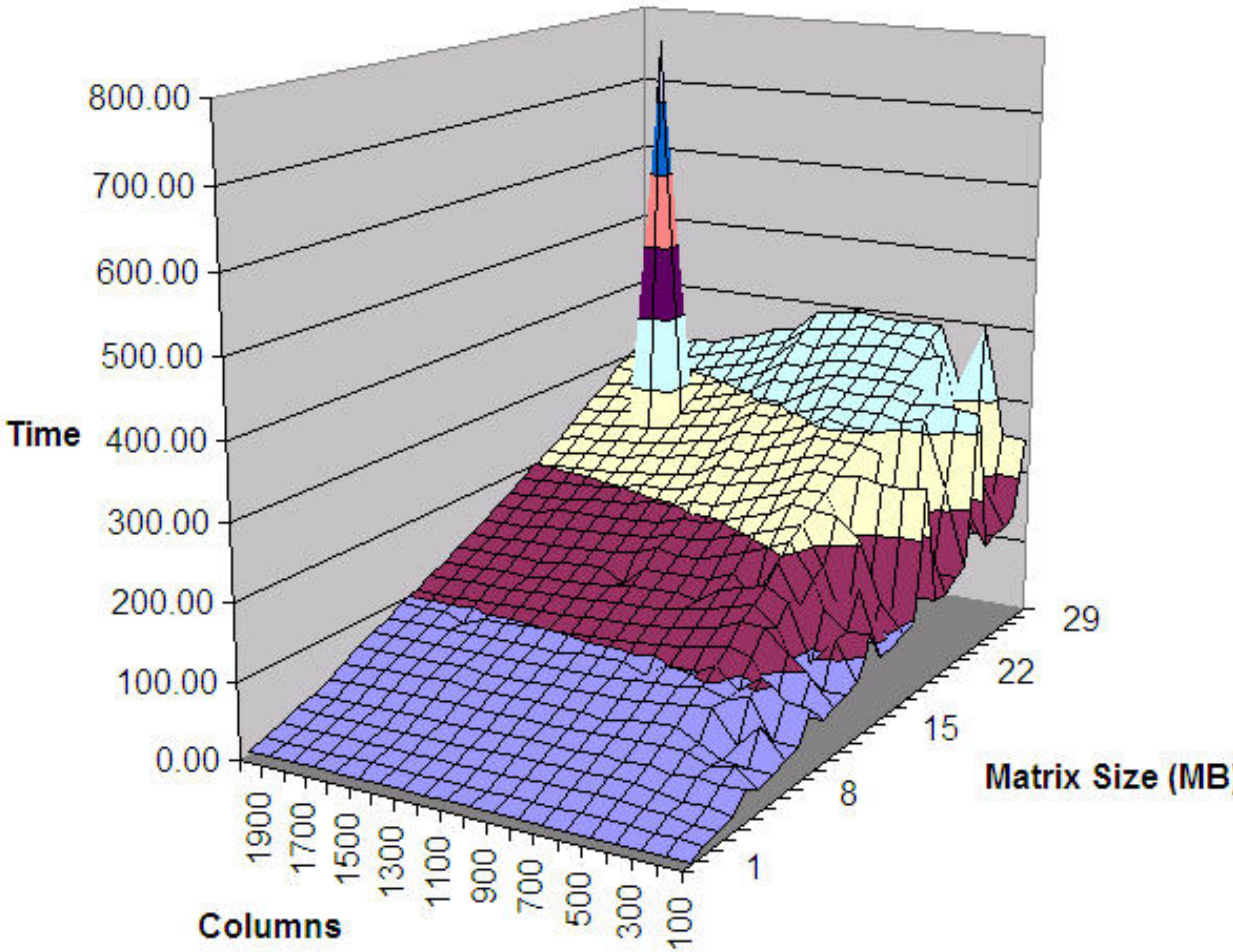
Beyond Surface-Scratching

Row major traversal performance unsurprising:



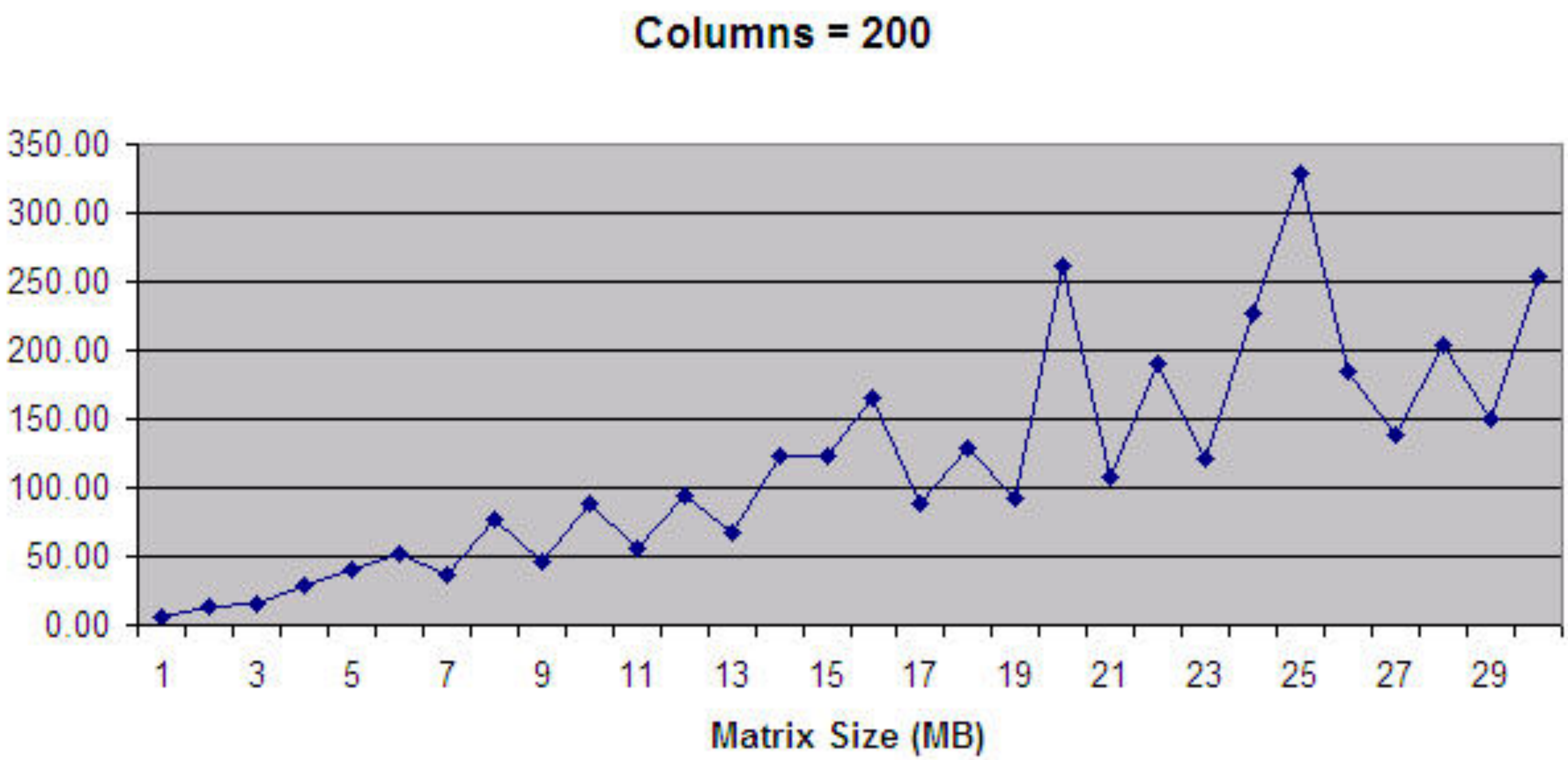
Beyond Surface-Scratching

Column major a different story:



Beyond Surface-Scratching

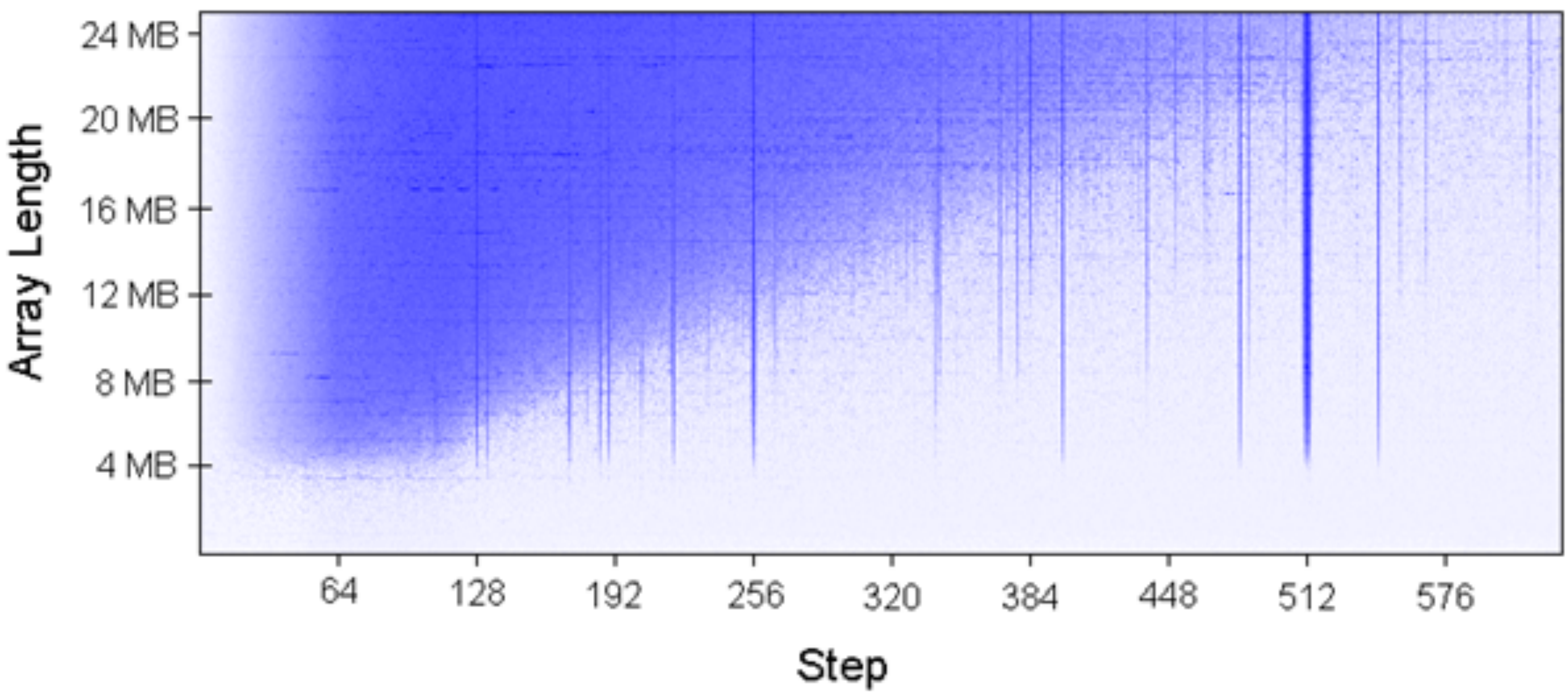
A slice through the data:



Beyond Surface-Scratching

Igor Ostrovsky's demonstration of cache-associativity effects.

- White ⇒ fast.
- Blue ⇒ slow.



Guideline

Understand the importance of CPU caches.

Writing Fast C++: The Language

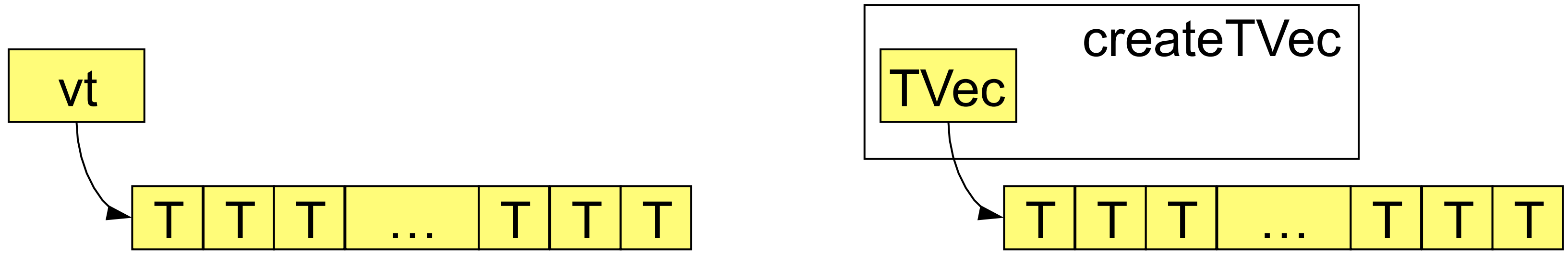
- Move Semantics
- Avoiding Unnecessary Object Creation
- Custom Heap Management

Take Advantage of Move Semantics

The most important speed-related feature in C++11.

C++ sometimes performs unnecessary copying:

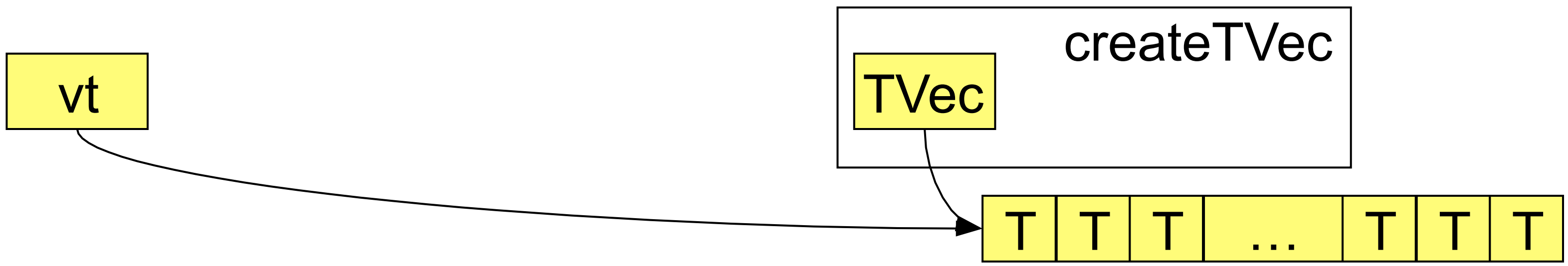
```
typedef std::vector<T> TVec;
TVec createTVec();                 // factory function
TVec vt;
…
vt = createTVec();                 // copy return value object to vt,
                                   // then destroy return value object
```



Move Support

Moving values would be cheaper:

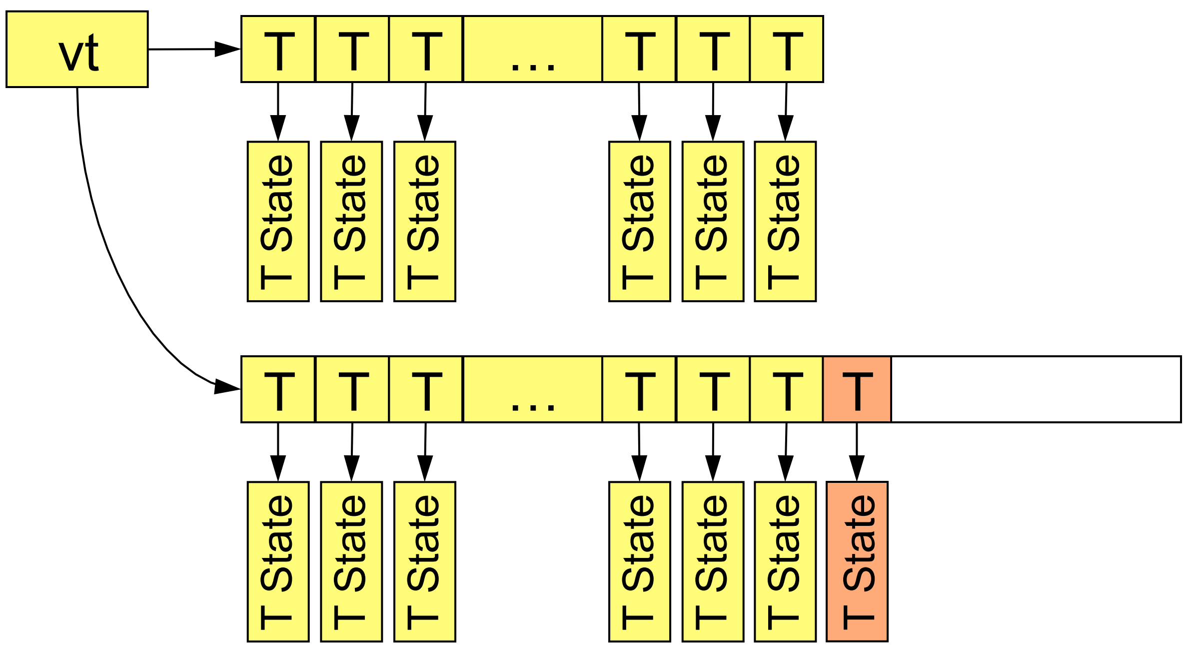
```
TVec vt;
…
vt = createTVec();                // move data in return value object
                                  // to vt, then destroy return value
                                  // object
```



Move Support

Appending to a full vector causes much copying before the append:

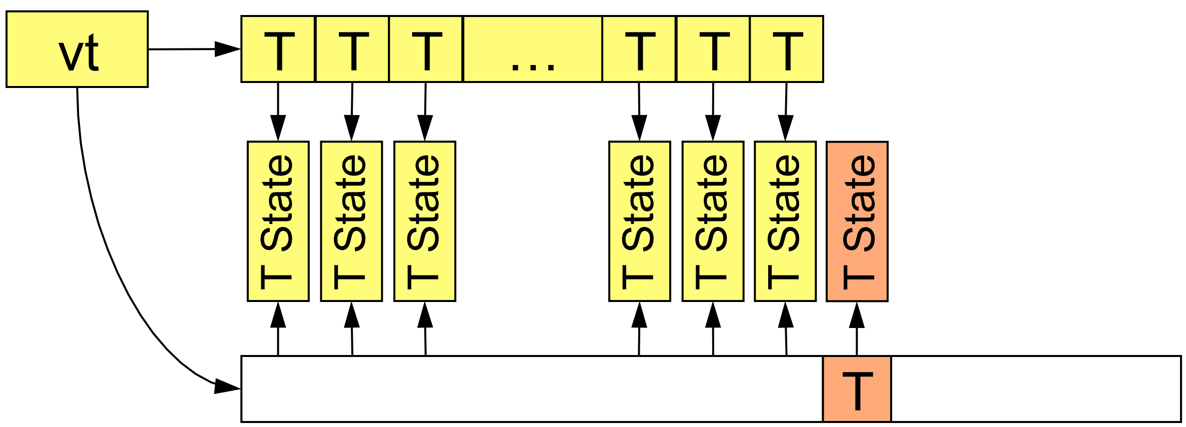
```
std::vector<T> vt;
...
vt.push_back(T object);                 // assume vt lacks
                                        // unused capacity
```



Move Support

Again, moving would be more efficient:

```
std::vector<T> vt;
...
vt.push_back(T object);                    // assume vt lacks
                                           // unused capacity
```



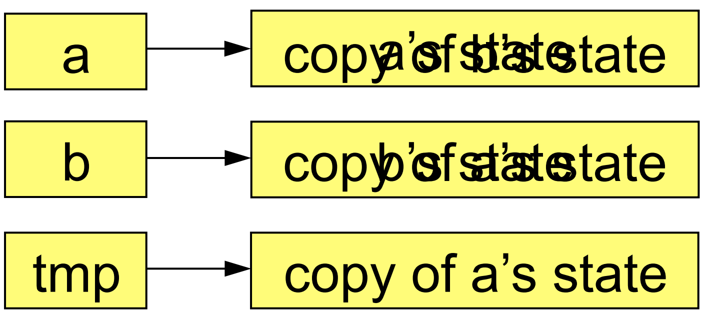
Other vector and deque operations could similarly benefit.

- insert, emplace, resize, erase, etc.

Move Support

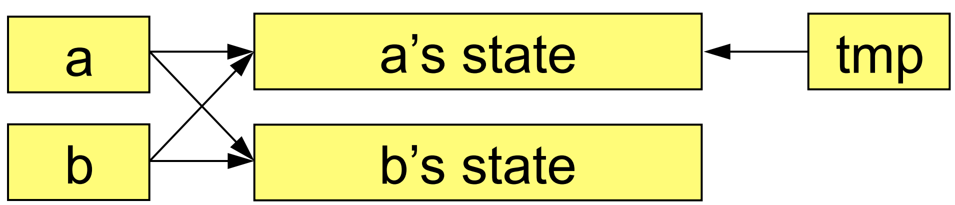
Still another example:

```
template<typename T>            // straightforward std::swap impl.
void swap(T& a, T& b)
{
  T tmp(a);                     // copy a to tmp (⇒ 2 copies of a)
  a = b;                        // copy b to a (⇒ 2 copies of b)
  b = tmp;                      // copy tmp to b (⇒ 2 copies of tmp)
}                               // destroy tmp
```



Move Support

```
template<typename T>                    // straightforward std::swap impl.
void swap(T& a, T& b)
{
  T tmp(std::move(a));                  // move a's data to tmp
  a = std::move(b);                     // move b's data to a
  b = std::move(tmp);                   // move tmp's data to b
}                                       // destroy (eviscerated) tmp
```



Move Support

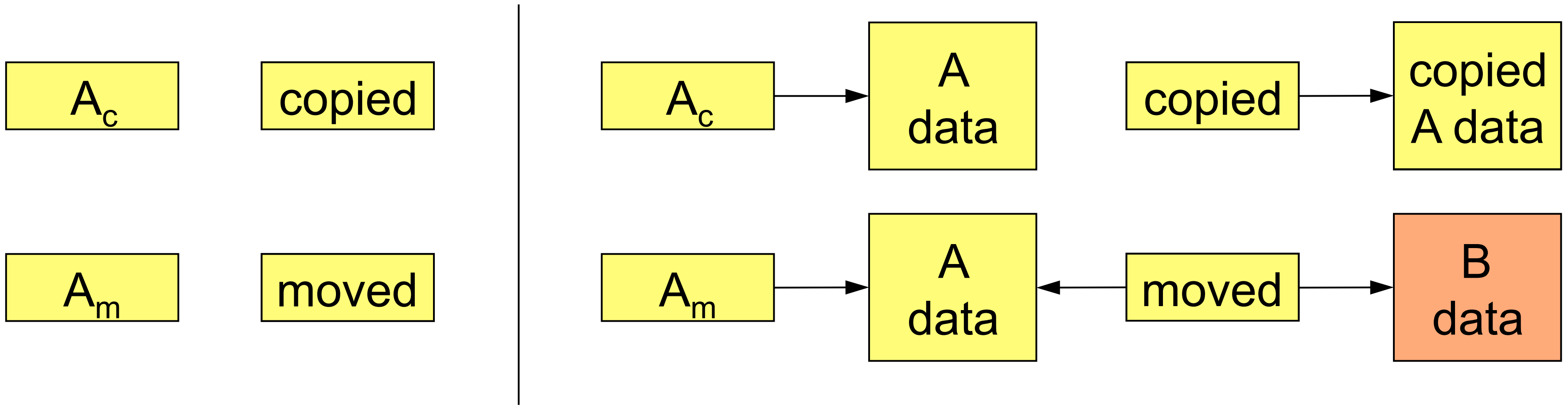
Moving most important when:

- Object has data in separate memory (e.g., on heap).
- Copying is deep.

Moving copies only object memory.

- Copying copies object memory + **separate memory**.

Consider copying/moving A to B:



Moving never slower than copying, and often faster.

Simple Performance Test

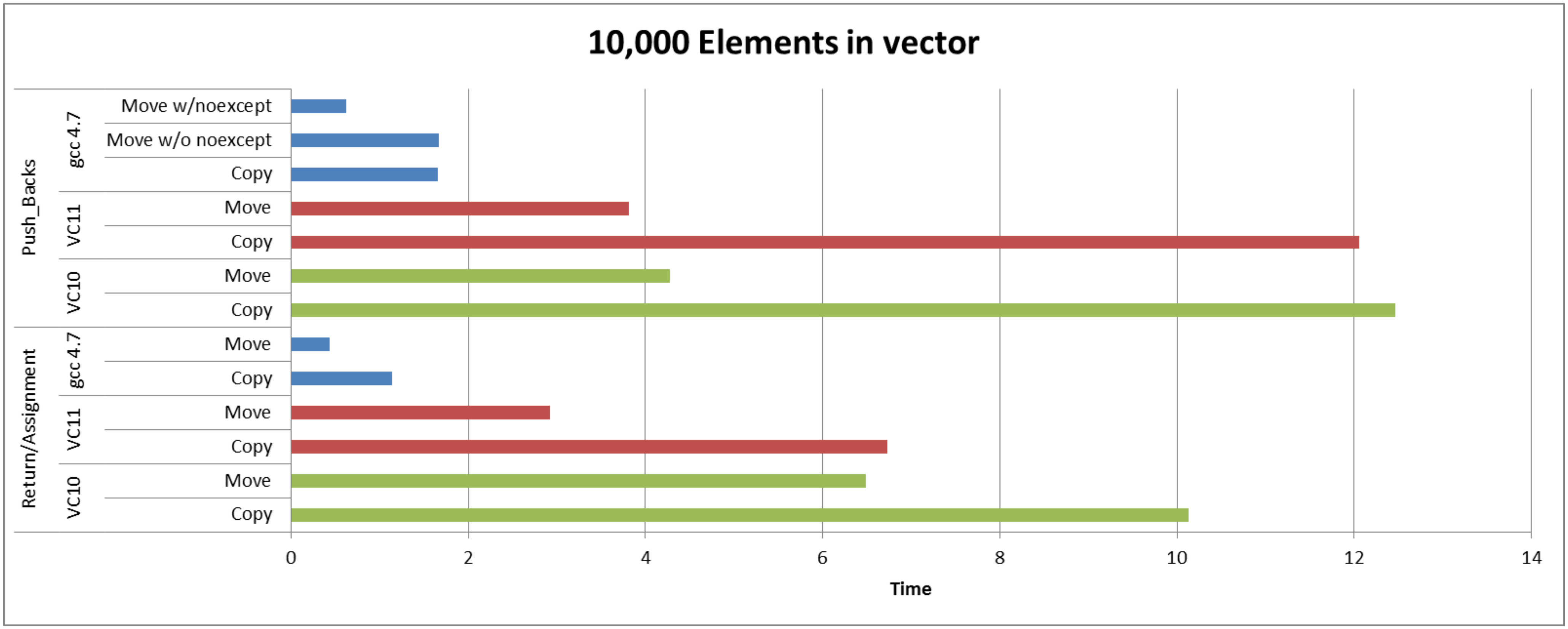
Given

```
const std::string stringValue("This string has 29 characters");

class Widget {
private:
  std::string s;

public:
  Widget(): s(stringValue) {}
  ...                                   // copy and move operations
};

typedef std::vector<Widget> TVec;
```

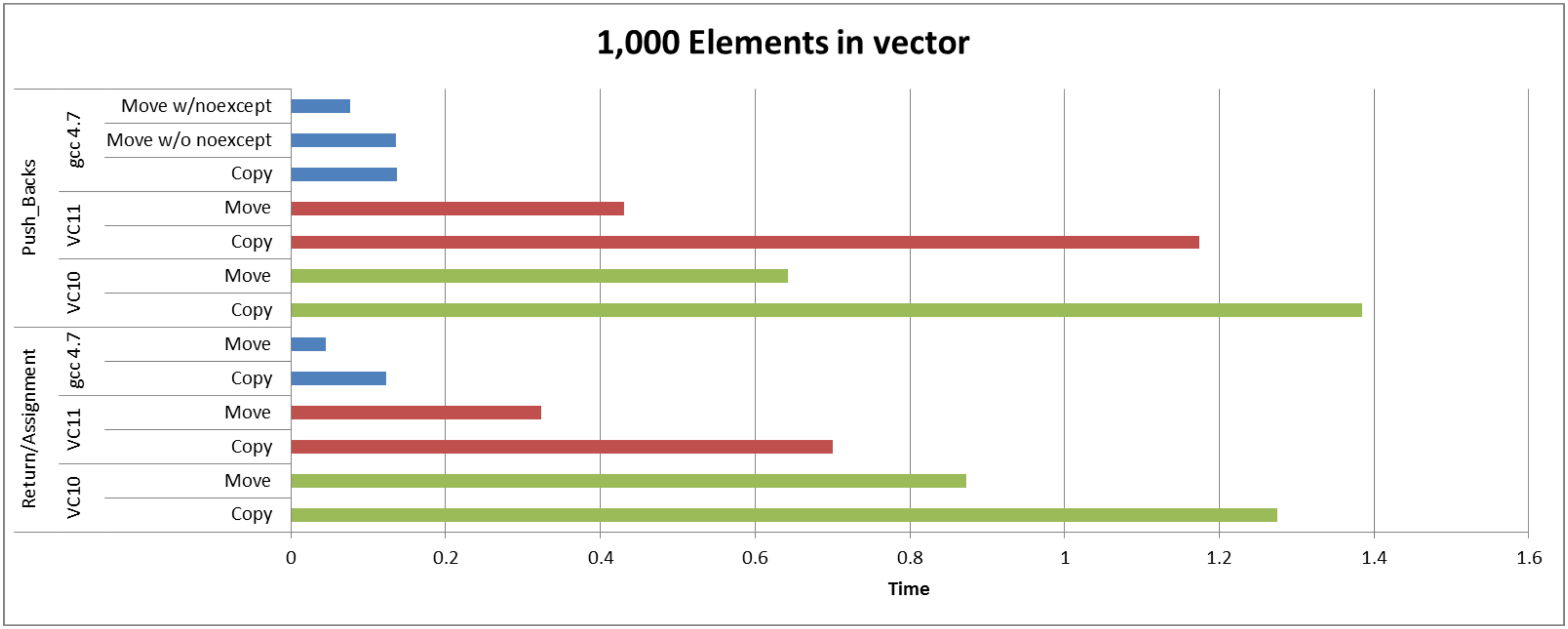
consider these use cases again:

```
vt = createTVec();                      // return/assignment of TVec

vt.push_back(T object);                 // push_back onto full TVec
```

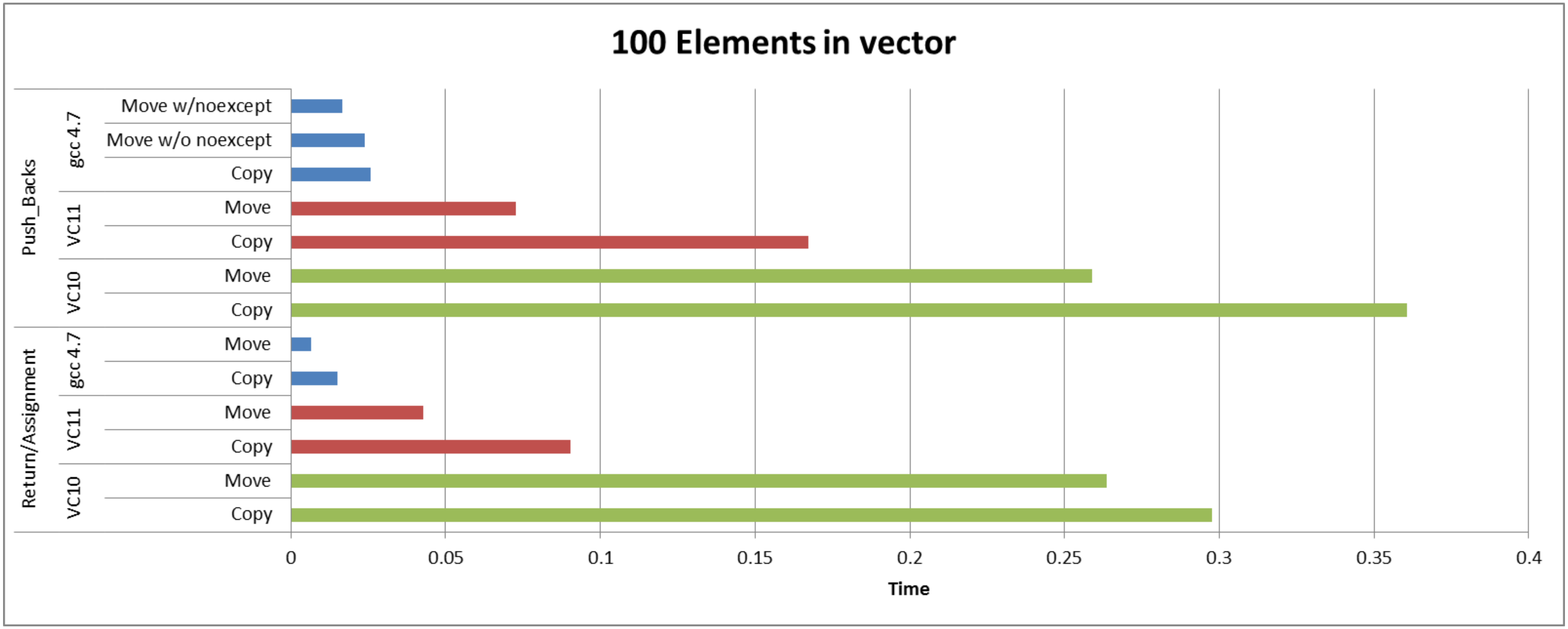
Performance Data



Performance Data



Performance Data



Move Support

Lets C++ recognize move opportunities and take advantage of them.

- How recognize them?
- How take advantage of them?

Lvalues and Rvalues

Lvalues are generally things you can take the address of:

- Named objects.
- Lvalue references.
  - ➡ More on this term in a moment.

Rvalues are generally things you can't take the address of.

- Typically unnamed temporary objects.

Examples:

```
int x, *pInt;                    // x, pInt, *pInt are lvalues
std::size_t f(std::string str);  // f and str are lvalues,
                                 // f's return is rvalue
f("Hello");                      // temp string created for call
                                 // is rvalue
std::vector<int> vi;             // vi is lvalue
…
vi[5] = 0;                       // vi[5] is lvalue
```

- ➡ Recall that `vector<T>::operator[]` returns `T&`.

Moving and Lvalues

Value movement generally not safe when the source is an lvalue.

- The lvalue object continues to exist, may be referred to later:

```
TVec vt1;
…
TVec vt2(vt1);                  // author expects vt1 to be
                                // copied to vt2, not moved!

…use vt1…                       // value of vt1 here should be
                                // same as above
```

Moving and Rvalues

Value movement is safe when the source is an rvalue.

- Temporaries go away at statement's end.
  - ➡ No way to tell if their value has been modified.

```
TVec createTVec();              // as before
TVec vt1;
vt1 = createTVec();             // rvalue source: move okay
TVec vt2(createTVec());         // rvalue source: move okay
vt1 = vt2;                      // lvalue source: copy needed
TVec vt3(vt2);                  // lvalue source: copy needed

std::size_t f(std::string str); // as before
f("Hello");                     // rvalue (temp) source: move okay
std::string s("C++11");
f(s);                           // lvalue source: copy needed
```

Rvalue References

C++11 introduces **rvalue references**.

- Syntax: `T&&`
- "Normal" references now known as **lvalue references**.

Rvalue references behave similarly to lvalue references.

- Must be initialized, can't be rebound, etc.

Rvalue references identify objects that may be moved from.

Reference Binding Rules

Important for overloading resolution.

As always:

- Lvalues may bind to lvalue references.
- Rvalues may bind to lvalue references to `const`.

In addition:

- Rvalues may bind to rvalue references to non-`const`.
- Lvalues may *not* bind to rvalue references.
  - ➡ Otherwise lvalues could be accidentally modified.

Rvalue References

Examples:

```
void f1(const TVec&);            // takes const lvalue ref
TVec vt;
f1(vt);                          // fine (as always)
f1(createTVec());                // fine (as always)

void f2(const TVec&);            // #1: takes const lvalue ref
void f2(TVec&&);                 // #2: takes non-const rvalue ref
f2(vt);                          // lvalue ⇒ #1
f2(createTVec());                // both viable, non-const rvalue ⇒ #2

void f3(const TVec&&);           // #1: takes const rvalue ref
void f3(TVec&&);                 // #2: takes non-const rvalue ref
f3(vt);                          // error!  lvalue
f3(createTVec());                // both viable, non-const rvalue ⇒ #2
```

Distinguishing Copying from Moving

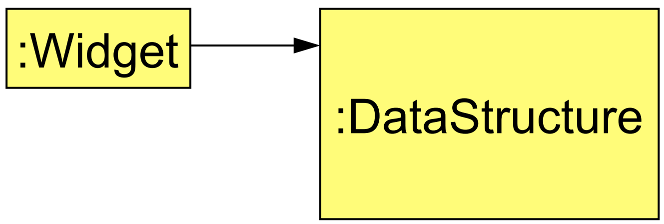
Overloading exposes move-instead-of-copy opportunities:

```
class Widget {
public:
  Widget(const Widget&);                  // copy constructor
  Widget(Widget&&);                       // move constuctor

  Widget& operator=(const Widget&);       // copy assignment op
  Widget& operator=(Widget&&);            // move assignment op
  …
};

Widget createWidget();                    // factory function

Widget w1;

Widget w2 = w1;                           // lvalue src ⇒ copy req’d

w2 = createWidget();                      // rvalue src ⇒ move okay

w1 = w2;                                  // lvalue src ⇒ copy req’d
```

Implementing Move Semantics

Move operations take source's value, but leave source in valid state:

```
class Widget {
public:
  Widget(Widget&& rhs)
  : pds(rhs.pds)                        // take source's value
{ rhs.pds = nullptr; }                  // leave source in valid state

  Widget& operator=(Widget&& rhs)
  {
    delete pds;                         // get rid of current value
    pds = rhs.pds;                      // take source's value
      rhs.pds = nullptr;                // leave source in valid state
    return *this;
   }
  …
private:
  struct DataStructure;
  DataStructure *pds;
};
```



Easy for built-in types (e.g., pointers). Trickier for UDTs…

Implementing Move Semantics

Part of C++11's string type:

```
string::string(const string&);        // copy constructor
string::string(string&&);             // move constructor
```

An incorrect move constructor:

```
class Widget {
private:
  std::string s;

public:
  Widget(Widget&& rhs)                // move constructor
  : s(rhs.s)                          // compiles, but copies!
  { … }
  …
};
```

- rhs.s an **lvalue**, because it has a name.
  - ➡ Lvalueness/rvalueness orthogonal to type!
    - ◆ ints can be lvalues or rvalues, and rvalue references can, too.
  - ➡ s initialized by string's *copy* constructor.

Implementing Move Semantics

Another example:

```
class WidgetBase {
public:
  WidgetBase(const WidgetBase&);          // copy ctor
  WidgetBase(WidgetBase&&);               // move ctor
  …
};

class Widget: public WidgetBase {
public:
  Widget(Widget&& rhs)                    // move ctor
  : WidgetBase(rhs)                       // copies!
  { … }
  …
};
```

- rhs is an **lvalue**, because it has a name.
  - ➡ Its declaration as Widget&& is not relevant!

Explicit Move Requests

To request a move on an lvalue, use std::move:

```
class WidgetBase { … };
class Widget: public WidgetBase {
public:
  Widget(Widget&& rhs)                          // move constructor
  : WidgetBase(std::move(rhs)),                 // request move
    s(std::move(rhs.s))                         // request move
  { … }
  Widget& operator=(Widget&& rhs)               // move assignment
  {
    WidgetBase::operator=(std::move(rhs));      // request move
    s = std::move(rhs.s);                       // request move
    return *this;
  }
  …
};
```

std::move turns lvalues into rvalues.

- The overloading rules do the rest.

Implementing std::move

std::move is simple – in concept:

```
template<typename T>
T&&                                // return as an rvalue whatever
move(MagicReferenceType obj)       // is passed in; must work with
{                                  // both lvalue/rvalues
  return obj;
}
```

Arcane language rules require an implementation like this:

```
template<typename T>
typename std::remove_reference<T>::type&&
move(T&& obj)
{
  return
    static_cast<typename std::remove_reference<T>::type&&>(obj);
}
```

- It's just a cast.

"T&&" Parameters

Compare conceptual and actual declarations for std::move:

```
template<typename T>
T&& move(MagicReferenceType obj);                          // conceptual

template<typename T>
typename std::remove_reference<T>::type&& move(T&& obj);  // actual
```

In a **function template**, a T&& parameter "takes anything:"

- Binds to lvalue or rvalue, const or non-const.
  - ➡ For lvalue arguments, it becomes T&, for rvalue args, it's T&&.
    - ◆ It really is a magic reference type!

```
template<typename T>
void f1(T&& param);                // takes anything
```

In a **non-template function**, a T&& parameter is an rvalue reference.

- It binds only to non-const rvalues.

```
void f2(Widget&& param);           // takes only non-const rvalues
```

Move is an Optimization of Copy

Move requests for copyable types w/o move support yield copies:

```
class Widget {                          // class w/o move support
public:
  Widget(const Widget&);                // copy ctor
};

class Gadget {                          // class with move support
public:
  Gadget(Gadget&& rhs)                  // move ctor
  : w(std::move(rhs.w))                 // request to move w's value
  { … }

private:
  Widget w;                             // lacks move support
};
```

rhs.w is *copied* to w:

- std::move(rhs.w) returns an rvalue of type Widget.
- That rvalue is passed to Widget's copy constructor.

Move is an Optimization of Copy

If Widget adds move support:

```
class Widget {
public:
  Widget(const Widget&);              // copy ctor
  Widget(Widget&&);                   // move ctor
};

class Gadget {                        // as before
public:
  Gadget(Gadget&& rhs)
  : w(std::move(rhs.w)) { … }         // as before

private:
  Widget w;
};
```

rhs.w is now *moved* to w:

- std::move(rhs.w) still returns an rvalue of type Widget.
- That rvalue now passed to Widget's move constructor.
  - Via normal overloading resolution.

Move is an Optimization of Copy

Implications:

- Giving classes move support can improve performance even for move-unaware code.
  - Copy requests for rvalues may silently become moves.
- Move requests safe for types w/o explicit move support.
  - Such types perform copies instead.
    - E.g., all built-in types.

In short:

- **Give classes move support when moving faster than copying.**
- **Use `std::move` for lvalues that may safely be moved from.**

Beyond Move Construction/Assignment

Move support useful for other functions, e.g., setters:

```
class Widget {
public:
  ...
  void setName(const std::string& newName)
  { name = newName; }                          // copy param

  void setName(std::string&& newName)
  { name = std::move(newName); }               // move param

  void setCoords(const std::vector<int>& newCoords)
  { coordinates = newCoords; }                 // copy param

  void setCoords(std::vector<int>&& newCoords)
  { coordinates = std::move(newCoords); }      // move param
  ...

private:
  std::string name;
  std::vector<int> coordinates;
};
```

Construction and Perfect Forwarding

Constructors often copy parameters to data members:

```
class Widget {
public:
  Widget(const std::string& n, const std::vector<int>& c)
  :  name(n),                                // copy n to name
     coordinates(c)                          // copy c to coordinates
  {}
  ...
private:
  std::string name;
  std::vector<int> coordinates;
};
```

Construction and Perfect Forwarding

Moves for rvalue arguments would be preferable:

```
std::string lookupName(int id);
int widgetID;
...
std::vector<int> tempVec;                  // used only for Widget ctor
...
Widget w(lookupName(widgetID),             // rvalues args, but Widget
         std::move(tempVec));              // ctor copies to members
```

Overloading Widget ctor for lvalue/rvalue combos ⇒ 4 functions.

- Generally, n parameters requires 2^n overloads.
  - Impractical for large n.
  - Boring/repetitive/error-prone for smaller n.

Construction and Perfect Forwarding

Goal: one function that copies lvalue args, but moves rvalue args.

Solution is a **perfect forwarding** ctor:

- A "takes anything" ctor forwarding T&& params to members:

```
class Widget {
public:
  template<typename T1, typename T2>
  Widget(T1&& n, T2&& c)                    // n and c bind everything
  : name(std::forward<T1>(n)),              // forward n to string ctor
    coordinates(std::forward<T2>(c))        // forward c to vector ctor
  {}
  ...
private:
  std::string name;
  std::vector<int> coordinates;
};
```

- Lvalue arg passed to n ⇒ std::string ctor receives lvalue.
- Rvalue arg passed to n ⇒ std::string ctor receives rvalue.
- Similarly for for c and std::vector ctor.

Perfect Forwarding

- Applicable only to function templates.
  - ➡ *Any* function template.
    - ◆ Not just constructors, not just member functions, e.g.,

```
class Widget {                        // as before
public:
   ...
   template<typename T>
   void setName(T&& newName)
   { name = std::forward<T>(newName); }

   template<typename T>
   void setCoords(T&& newCoords)
   { coordinates = std::forward<T>(newCoords); }

   ...
};
```

- Preserves arguments' lvalueness/rvalueness/`const`ness when forwarding them to other functions.
- Implemented via `std::forward`.
- Consult Further Information for details.

Guideline

Take advantage of move semantics.

Avoid Unnecessary Object Creation

Constructors called:

- When object defined (stack, heap, or static)
- When array of objects defined (stack, heap, or static)
- When function parameter passed by value
- When function returns an object

Applies even to compiler-generated temporary objects.

Destructors called:

- When named stack object, array, or parameter goes out of scope
- When heap object or array is deleted
- For static objects, at end of the program
- For temporary objects, at end of "full expression" in which they are created

Object Creation and Destruction

```
#include <string>
  // class string {                     // string acts as if it were
  // public:                            // defined like this
  //    string();
  //    string(const char*);
  //    string(const string& rhs);      // copy ctor
  //    string(string&& rhs);           // move ctor
  //    ...
  // };

std::string s1("Hello");                // 1 ctor call

std::string s2(s1);                     // 1 ctor call
std::string s3 = "Hello";               // 1 or 2 ctor calls

std::string sa1[10];                    // 10 ctor calls
std::string sa2[] =                     // 3 or 6 ctor calls
  { std::string("One"), std::string("Two"), std::string("Three") };
```

Destructors called when objects go out of scope.

Object Creation and Destruction

```
std::string interleave(std::string str1, std::string str2);
std::cout << interleave(s1, "Hello");     // at least 3 ctor calls
```

Destructors called when these objects go away.

Objects, Inheritance, and Containment

Inheritance results in implicit calls:

- Base class ctors/dtors called for derived class objects

So does containment:

- Data members initialized via ctors and destroyed via dtors

Objects, Inheritance, and Containment

```
class Person {
public:
  Person(const std::string& who, const std::string& where);
  ...
private:
  std::string name, address;
};
class Student: public Person {
public:
  Student(const std::string& who, const std::string& where);
  ...
private:
  std::string idNumber;
};
std::string name("Chris");
std::string location("Bermuda");
int main() {
  Student s(name, location);          // 5 ctors called
}                                     // 5 dtors called
```

Construction/destruction of objects can be expensive!

Avoiding Unnecessary Objects

1. Pass read-only parameters by ref-to-const instead of by value:

```
bool operator==( Widget lhs, Widget rhs);          // bad

bool operator==( const Widget& lhs,                // good
                 const Widget& rhs);
```

- Requires the existence of const member functions!
- Especially important when writing templates:

```
template<typename T>
bool operator!=(const T& lhs, const T& rhs) { return !(lhs == rhs); }
```

Avoiding Unnecessary Objects

- Built-in types an exception; pass-by-value okay for them:

```
std::vector<Widget>
makeClones(const Widget& w,              // pass by ref
           int numClones);               // pass by value
```

- Another exception: STL iterators and function objects:

```
template<typename It, typename Func>              // from C++
Func for_each(It begin, It end, Func f);          // std lib
```

Avoiding Unnecessary Objects

2. Defer object definitions as long as possible:
   - Ideally until initialization arguments can be provided

```
std::string getUserName();

void f()                                // bad
{
  std::string name;
  ...
  name = getUserName();
  ...
}

void f()                                // good
{
  ...
  std::string name(getUserName());
  ...
}
```

Avoiding Unnecessary Objects

3. Prefer initialization to assignment in constructors:

```
class NamedData {
public:
  NamedData(const std::string& initName, void *dataPtr);
  ...
private:
  std::string name;
  void *data;
};

NamedData::NamedData(const std::string& initName, void *dataPtr)
{
  name = initName;                         // bad
  data = dataPtr;
}

NamedData::NamedData(const std::string& initName, void *dataPtr)
: name(initName), data(dataPtr)            // good
{}
```

Avoiding Unnecessary Objects

4. Consider overloading to avoid implicit type conversions:

```
bool operator==(const std::string& lhs,        // declared in
                const std::string& rhs);       // <string>

std::string s;

if (s == "Hello") ...                  // converts "Hello" to string
if ("Hello" == s) ...                  // via a temporary object
```

Overloads avoid need to generate temporaries:

```
bool operator==(const std::string& lhs, const char *rhs);

bool operator==(const char *lhs, const std::string& rhs);
```

Standard library includes all these functions for std::string.

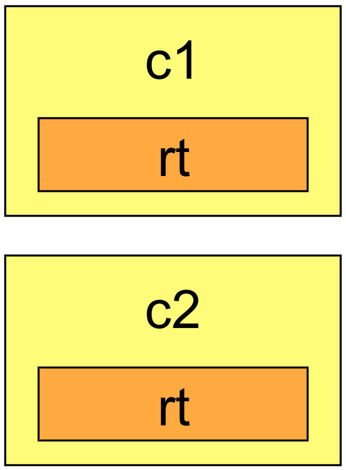
Avoiding Unnecessary Objects

A class for mobile phone contacts supporting custom ringtones:

```
class Ringtone { ... };                    // audio info for ringtone

class Contact {
  Ringtone rt;                             // ringtone for this contact
  ...
};
```

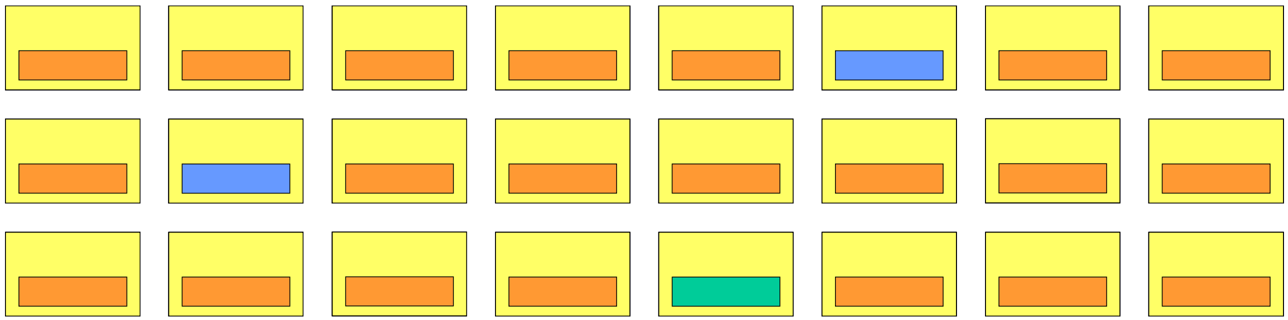
Copying a Contact copies its Ringtone:

```
Contact c1;
...

Contact c2(c1);
```



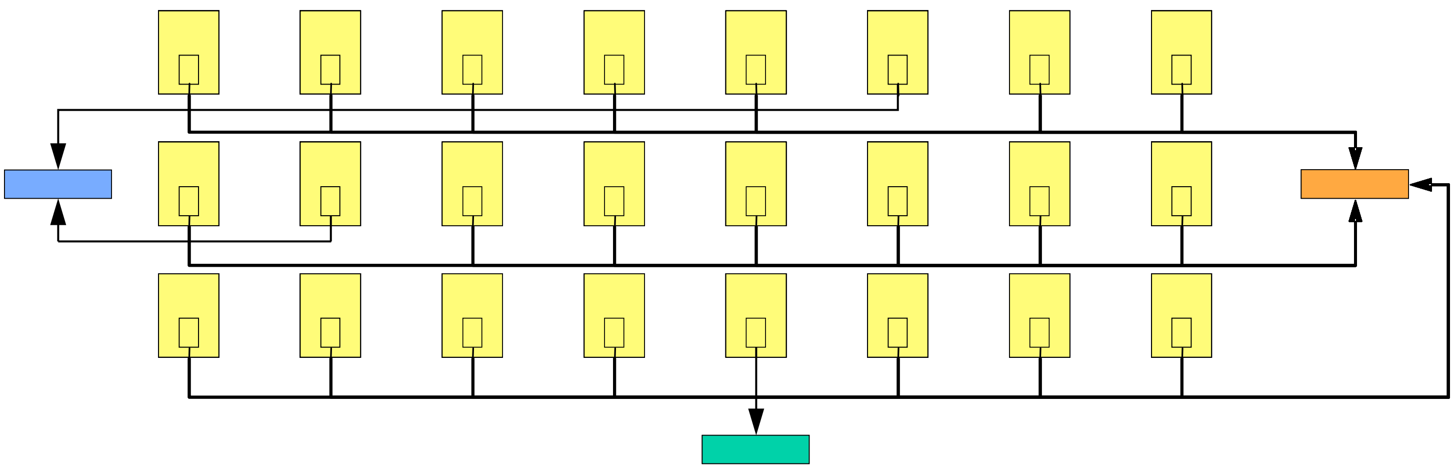
Avoiding Unnecessary Objects

Over time, many copies of a single Ringtone could arise:



Pointers to shared Ringtones would reduce ctor/dtor calls:

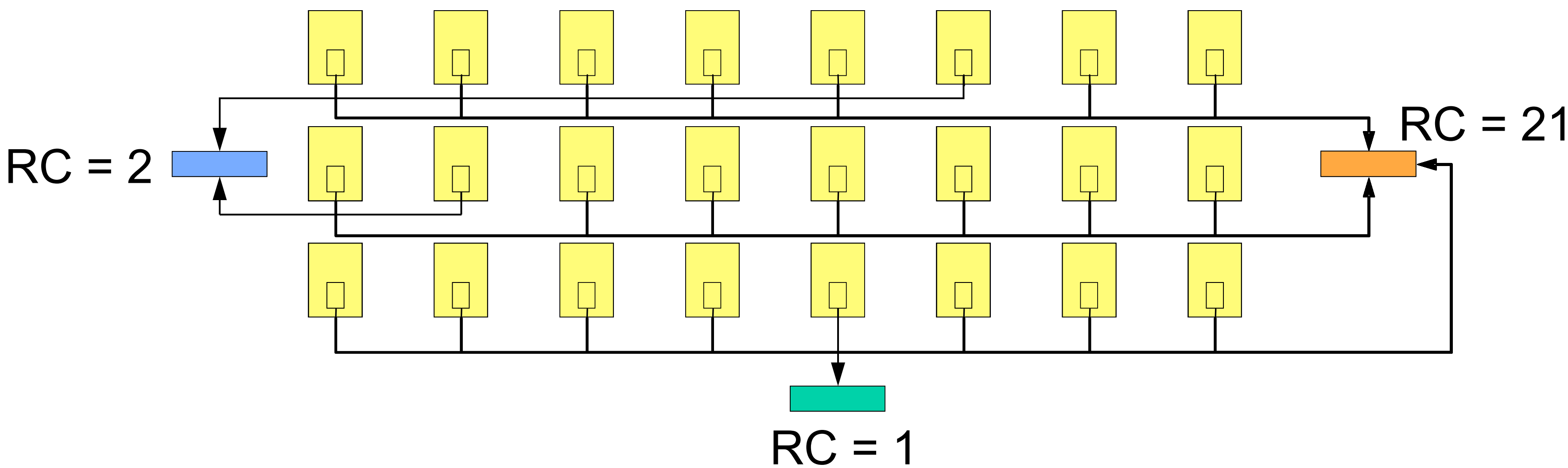
- Also save memory.



Avoiding Unnecessary Objects

When destroy shared values?

- When they're no longer needed.
  - ➡ I.e., when no other objects refer to them.
    - ◆ I.e., when their *reference count* (RC) → 0.



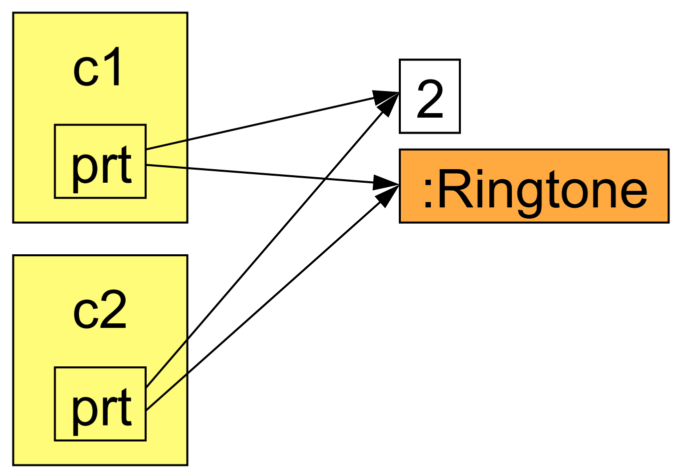
Avoiding Unnecessary Objects

std::shared_ptr automates RC manipulations:

- RC increased when shared_ptr created/copied.
- RC decreased when shared_ptr assigned/destroyed.

Makes employing RC easy:

```
class Ringtone { ... };                    // as before

class Contact {
  std::shared_ptr<Ringtone> prt;     // RC ptr to contact's ringtone
  ...
};

Contact c1;

...

Contact c2(c1);
```



Avoiding Unnecessary Objects

5. Consider using reference counting.
   - Typically via `std::shared_ptr`.

Allocations and std::shared_ptr

Given

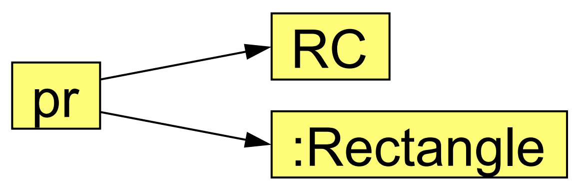
```
class Rectangle {
public:
  Rectangle(int lx, int ly, int rx, int ry);
  ...
};
```

typical shared_ptr initialization takes this form:

```
std::shared_ptr<Rectangle> pr(new Rectangle(0, 0, 10, 20));
```

Result is 2 allocations:

- 1 for Rectangle.
- 1 for implicit reference count.

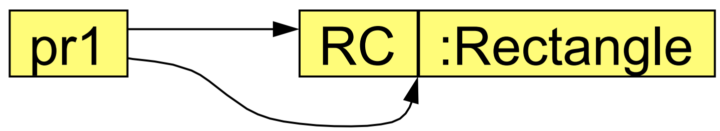


Avoiding Unnecessary Objects

6. Use std::make_shared and std::allocate_shared.

   ➡ Performs only 1 allocation:

```
std::shared_ptr<Rectangle>
  pr1(std::make_shared<Rectangle>(0, 0, 10, 20));
```



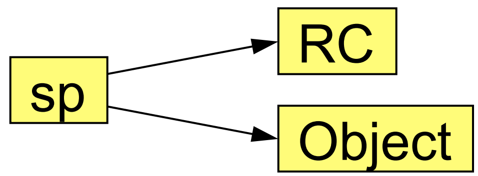
   ➡ allocate_shared allows use of custom allocator:

```
CustomAllocator ca;
...
std::shared_ptr<Rectangle>
  pr2(std::allocate_shared<Rectangle>(ca, 0, 0, 10, 20));
```

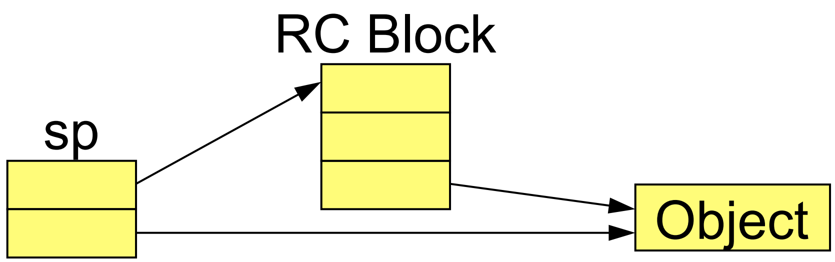
   ➡ These functions part of C++11 and Boost.

Memory Use and std::shared_ptr

Conceptual view of std::shared_ptr sp when std::make_shared and std::allocate_shared not used:

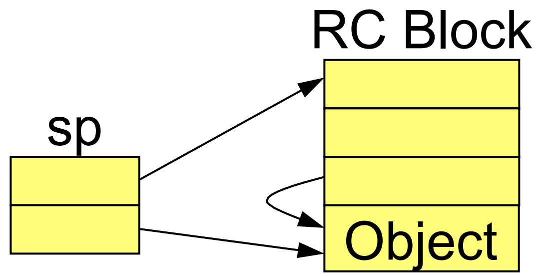


Common in-memory layout:

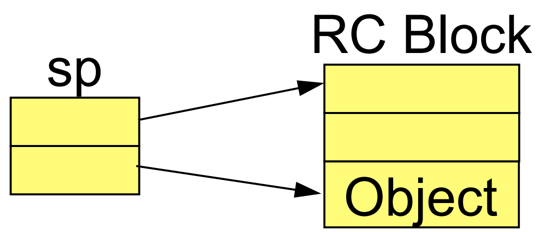


Memory Use and std::shared_ptr

Common layout when std::make_shared is used:



The "We know where you live" optimization eliminates the pointer from the RC block:



Hence:

```
std::shared_ptr<Widget>                     // 2 allocations; sp uses
  sp(new Widget);                           // 2 ptrs to Widget

std::shared_ptr<Widget>                     // 1 allocation; sp uses
  sp(std::make_shared<Widget>());           // 1 ptr to Widget (if
                                            // WKWYL implemented)
```

Same applies to std::make_allocate, though layout is a bit different.

Temporary Objects for Container Insertion

Not uncommon to create objects only to put into STL containers:

```
class Rectangle {                                    // as before
public:
  Rectangle(int lx, int ly, int rx, int ry);
  ...
};

std::deque<Rectangle> dr;
...
int leftx, lefty, rightx, righty;
...
dr.push_back(Rectangle(leftx, lefty, rightx, righty));
...
Rectangle temp(leftx, lefty, rightx, righty);   // used only to add to dr
dr.push_front(temp);
...
dr.insert(someIterator, Rectangle(leftx, lefty, rightx, righty));
```

Avoiding Unnecessary Objects

7. Use emplacment functions.

   ➡ Constructs new objects directly in container.

```
dr.emplace_back(leftx, lefty, rightx, righty);
...
dr.emplace_front(leftx, lefty, rightx, righty);
...
dr.emplace(someIterator, leftx, lefty, rightx, righty);
```

   ➡ These functions part of C++11.

Guideline

Avoid unnecessary object creation.

- Pass read-only parameters by ref-to-const instead of by value.
- Defer object definitions as long as possible.
- Prefer initialization to assignment in constructors.
- Consider overloading to avoid implicit type conversions.
- Consider using reference counting.
- Use std::make_shared and std::allocate_shared.
- Use emplacement functions.

Consider Custom Heap Management

Heap memory managed by

- operator new and operator delete (for single objects)
- operator new[] and operator delete[] (for arrays)

Custom versions may be defined at global or class scope.

Non-Performance Motivations

Customization can be useful even if performance not an issue:

- Detecting memory leaks.
- Detecting multiple deallocations.
- Detecting underwrites/overwrites.

Performance Motivations

Vendors' new/new[] and delete/delete[] are general-purpose.

Must handle variation in:

- How long a program runs.
- Sizes of memory allocation requests.
- Lifetimes of dynamically allocated objects.
- Thread-safety requirements.

Few applications require such generality.

Performance Motivations

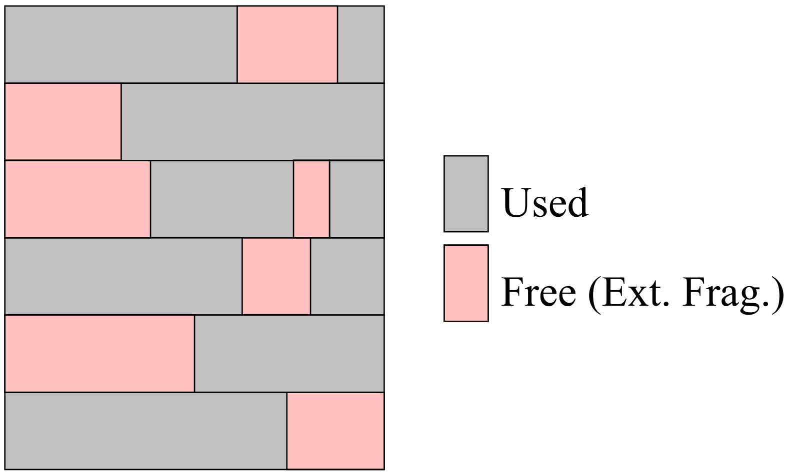
Custom heap management can improve performance when:

- The default heap managers are a bottleneck.
- You can develop better-performing implementations.
  - Typically by eliminating generality.
    - E.g. a thread-unsafe allocator for requests of exactly 64 bytes.

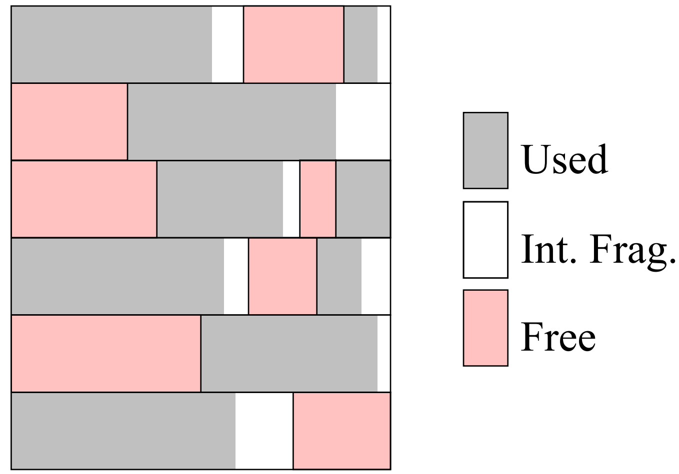
Fragmentation

General-purpose allocators must prevent excessive fragmentation.

External: free blocks too small to satisfy memory requests:



Internal: inaccessible memory within allocated blocks:



Custom allocators may be able to reduce fragmentation risk.

- Less risk ⇒ less code to deal with fragmentation.

When Custom Heap Management May Help

- Many objects' lifetimes end simultaneously:
  - ➡ All can be put in a heap ("arena") that's freed in one operation.
    - ◆ Deallocation cost for individual objects avoided.
- Objects are naturally used together:
  - ➡ All can be put in a single heap ("clustered").
    - ◆ Page faults and cache misses are reduced.
- Code is single-threaded, but default allocators are thread-safe:
  - ➡ ST programs or thread-specific allocators in MT programs.
- High allocator contention in MT software.
  - ➡ Try scalable allocators such as Hoard, mtmalloc, TCMalloc, etc.
- Very few allocation sizes vastly dominate:
  - ➡ Size-specific allocators ("pools") eliminate most fragmentation.
    - ◆ Heap manager overhead (time + size) declines.
- Default allocators offer suboptimal alignment:
  - ➡ On i86, access to doubles fastest when 8-byte aligned, but some default allocators may 4-byte align them.

Verifying Performance Improvements

Whether custom heap management helps depends on:

- Whether dynamic allocation/deallocation is a bottleneck.
- Behavior of custom allocator/deallocator.
- Behavior of default `new`/`new[]` and `delete`/`delete[]`:
  - ➡ Varies across compilers, compiler releases, OSes, etc.
    - ◆ What helps in one environment may hurt in another.

Real-world results vary and can be counterintuitive:

- **Empirically verify expected/alleged performance benefits.**
  - ➡ (True for all code changes to improve performance.)

Instrumented Heap Managers

They collect information about memory use.

- Typically use default allocators for actual allocation/deallocation.

Can yield insight into application behavior:

- Distribution of requested allocation sizes?
- Distribution of allocation lifetimes?
- "High water mark" for dynamically allocated memory?
- Size or temporal allocation patterns that tend to recur?
  - ➡ Is dynamic memory mostly LIFO? Mostly FIFO?
    - ◆ DB query evaluation is largely recursive ⇒ LIFO allocation.
  - ➡ Do patterns change across program phases?
    - ◆ Eg.., compiler front-end vs. back-end.

Insights can facilitate development of faster/smaller heap managers.

Guideline

Consider custom heap management.

C++ Language Summary

- Take advantage of move semantics.
- Avoid unnecessary object creation.
- Consider custom heap management.

Writing Fast C++: The Standard Library

- reserve and shrink_to_fit
- Range member functions
- Function objects
- Sorted vectors
- Sorting algorithms

Use reserve to Reduce Reallocations in vector and string

vectors and strings automatically grow to accommodate new insertions:

```
std::vector<int> v;

for (int i = 1; i <= 1000; ++i)          // v automatically grows to
  v.push_back(i);                        // make room for the insertions


std::string alphabet("abcdefghijklmnopqrstuvwxyz");
std::string s;

for (int j = 1; j <=1000; ++j)
  s += alphabet;                         // ditto for s
```

This is very convenient.

Reducing vector/string Reallocations

Convenience is not free.

- Multiple realloc-like operations typically take place in the previous examples:
  - On the platforms I've tested, v was reallocated 2-18 times, and s was reallocated 9-999 times!
  - Each involves memory allocation (via the container's allocator)
  - Each often involves
    - Copying or moving container elements from old memory to new memory and
    - Destructing elements in the old memory
- Each realloc-like operation invalidates all pointers, references, and iterators into the container:
  - Other data structures dependent on such pointers/references/ iterators may have to take the time to update themselves

Reducing vector/string Reallocations

Reallocations can be avoided or reduced via reserve:

```
std::vector<int> v;
v.reserve(1000);

for (int i = 1; i <= 1000; ++i)          // v's capacity never changes;
  v.push_back(i);                        // we know it's big enough

std::string alphabet("abcdefghijklmnopqrstuvwxyz");
std::string s;
s.reserve(26000);

for (int j = 1; j <=1000; ++j)
  s += alphabet;                         // ditto for s
```

Benefit:

- No time spent reallocating memory or copying/moving/ destroying elements

Excess Capacity

Reserving too much space can lead to excess capacity.

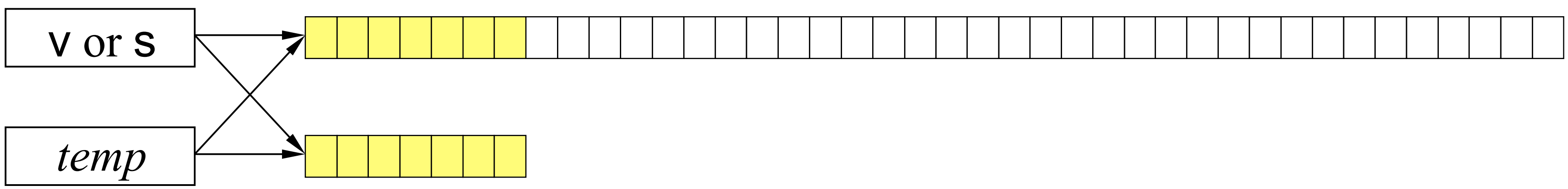
```
std::vector<int> v;
v.reserve(maxDataValues);        // reserve maximum needed space
...                              // put data into v. When done,
                                 // v.size() may be much less than
                                 // v.capacity()
std::string s;
s.reserve(maxNumChars);          // reserve maximum needed space
...          // s.size() could be much less
                                 // than s.capacity()
```

The swap Trick

Use "the swap trick" to eliminate excess capacity:

```
std::vector<int>(v.begin(), v.end()).swap(v);   // copy v's contents
                                                // into an unnamed
                                                // temporary vector,
                                                // then swap their
                                                // internal pointers

std::string(s.begin(), s.end()).swap(s);        // same thing
```

It works like this:



The swap Trick

Trick isn't guaranteed to work perfectly:

- Implementations may establish minimum capacities.

Advisable to hide details behind a nice interface:

```
template<typename T>
void shrink_to_fit(T& container)
{
  T(container.begin(), container.end()).swap(container);
}

std::vector<int> v;
std::string s;

…

shrink_to_fit(v);
shrink_to_fit(s);
```

shrink_to_fit in C++11

C++11 offers this functionality directly:

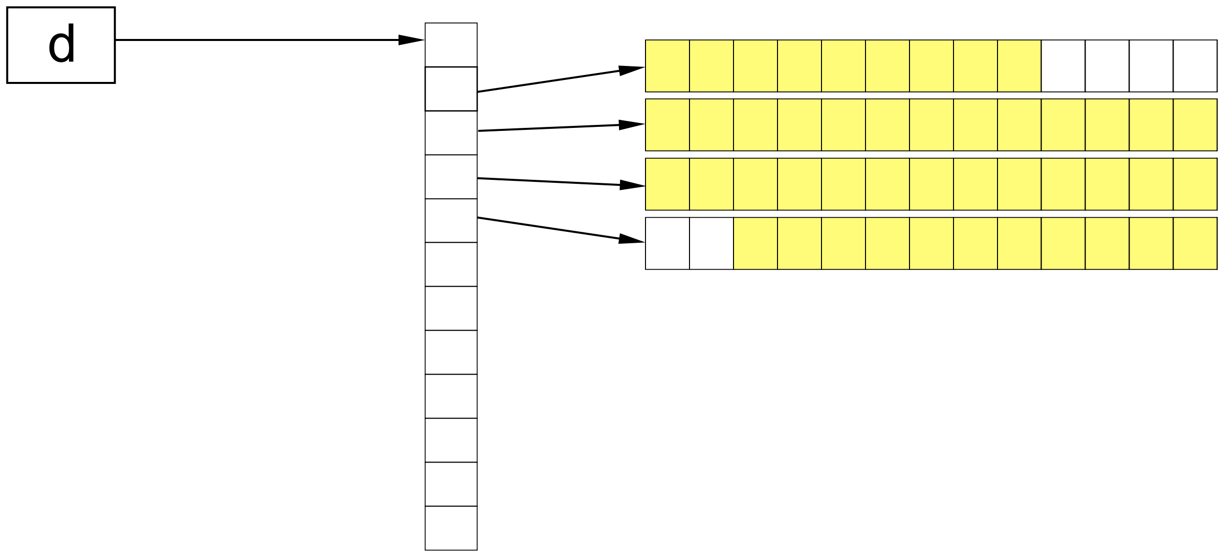
```
std::vector<int> v;
v.reserve(maxDataValues);          // as before
...                                // put data into v. As before,
                                              // excess capacity
may result
v.shrink_to_fit();                 // request elimination of excess
                                   // capacity
std::string s;
s.reserve(maxNumChars);            // as before
...        // again, s.size() could be much
                                   // less than s.capacity()
s.shrink_to_fit();                 // request elimination of excess
                                   // capacity
```

shrink_to_fit in C++11

Also available for deque, although deque lacks reserve.

```
std::deque<std::shared_ptr<Widget>> d;
...                                        // add many elements
...                                        // erase many elements
d.shrink_to_fit();                         // request elimination of
                                           // excess memory
```

- Applies to capacity of internal array:



shrink_to_fit in C++11

Writing shrink_to_fit may still be worthwhile for vector.

- Both VC10 and gcc 4.5 use copy-and-swap for vector, e.g.:

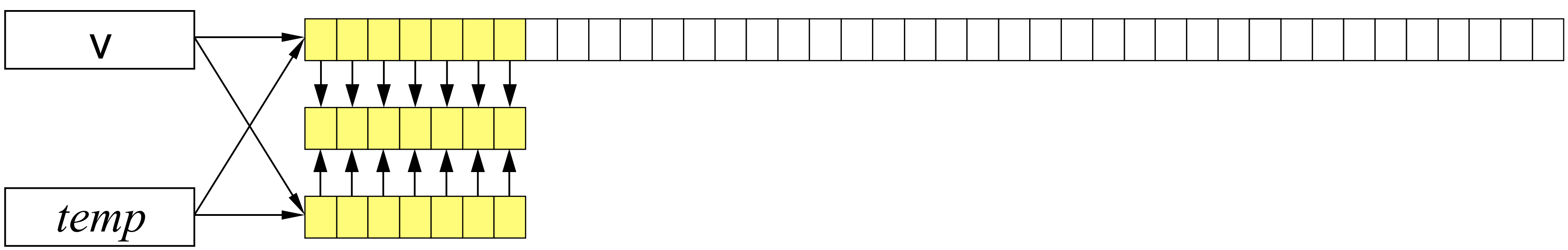
```
void shrink_to_fit()                  // VC10's vector::shrink_to_fit
{
  if (size() < capacity()) {
    _Myt _Tmp(*this);                 // copy *this to _Tmp (*this is lvalue)
    swap(_Tmp);                       // swap contents of *this and _Tmp
  }
}
```

- This *copies* elements from old to new storage.
- *Moving* would be preferable.

shrink_to_fit in C++11

Here's a move-based version:

```
template<typename T>
void shrink_to_fit(std::vector<T>& v)
{
  std::vector<T>(std::make_move_iterator(v.begin()),
                 std::make_move_iterator(v.end())).swap(v);
}
```



shrink_to_fit in C++11

Worth considering only for vector:

- string elements must be PODs.
  - ➡ Moving no faster than copying.
- deque::shrink_to_fit doesn't copy or move deque elements.

Test your compiler first:

- Already implemented in VC11 and gcc 4.7.

Guidelines

Use reserve to reduce reallocations in vector and string.

Use the swap trick or shrink_to_fit to reduce excess capacity in vectors, strings, and deques.

Prefer Range Member Functions to Single-Element Versions for Sequence Containers

Repeated single-element calls can be expensive. Example:

```
void addToMiddle( const std::vector<int>& src,     // copy elements of src
                  std::vector<int>& dest)          // into middle of dest
{
  auto destLoc = dest.cbegin() + dest.size() / 2;

  for (auto it = src.cbegin(); it != src.cend(); ++it) {
    destLoc = dest.insert(destLoc, *it);
  }
}
```

Before considering efficiency, this code has a subtle error. What is it?

- Hint: consider the final ordering of the elements.

Range vs. Single-Element Member Functions

But back to efficiency...

Each call to insert might:

- Exhaust dest's capacity, in which case it must:
  - Allocate more memory
  - Copy or move existing data to new memory
  - Destroy data in old memory
  - Deallocate old memory

There's more:

- Each element beyond the insertion point must be moved up one position to make room for the new element
  - This linear-looking algorithm is really quadratic!

Range vs. Single-Element Member Functions

The range version:

```
void addToMiddle(const std::vector<int>& src, std::vector<int>& dest)
{
  dest.insert(dest.cbegin()+dest.size()/2, src.cbegin(), src.cend());
}
```

- Performs at most one reallocation:
  - ➡ Needed space determined via `std::distance(src.cbegin(), src.cend())`
- Puts each value into final position via a single copy/move.
  - ➡ The operation is truly linear.
- Code is easier to write.

Fine print:

- For input iterators, complexity is typically quadratic.

Range vs. Single-Element Member Functions

Similarly:

- Range erasures better than repeated single-element erasures.
  - Avoids shifting values down one position at a time.
  - Implication: erase(remove(...), ...) idiom better than hand- written loop calling erase:

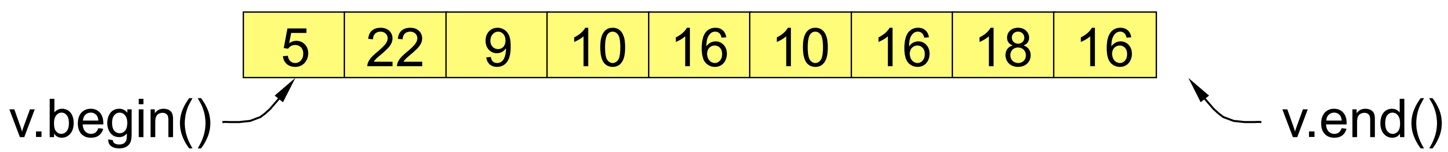
```
std::vector<int> v;
...
int val;
...
auto it = v.cbegin();
while (it != v.cend()) {
   if (*it == val) it = v.erase(it);                    // bad
   else ++it;
}

v.erase(std::remove(v.begin(), v.end(), val),
         v.cend());                                      // good
```

Aside: The Behavior of std::remove

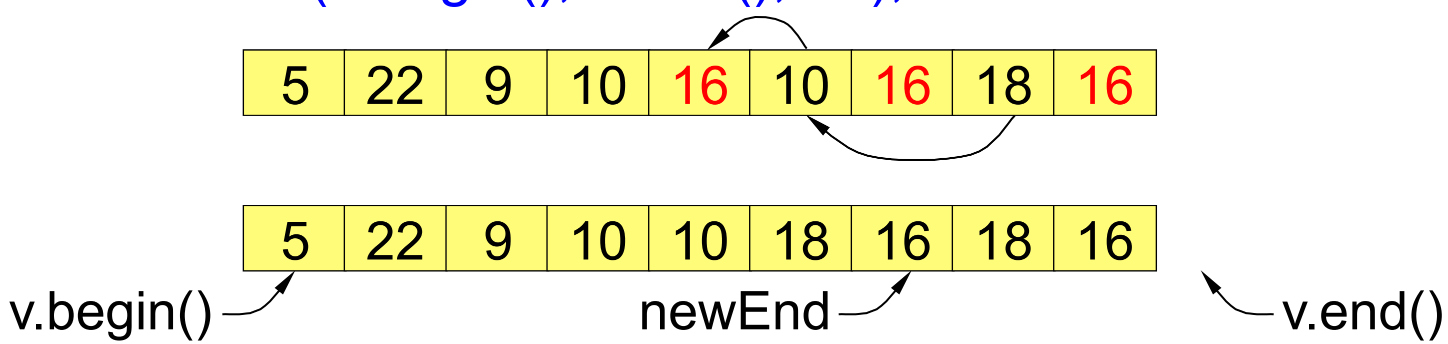
remove's behavior counterintuitive:

- **Doesn't physically get rid of anything.**
  - ➡ Doesn't know what container it's operating on.
- Values to be removed are holes. V**alues shift down** to fill them:

```
std::vector<int> v;
...                                          // put values in v
```




```
auto newEnd =                                // "remove" all elements
  std::remove(v.begin(), v.end(), 16);       // with value 16
```

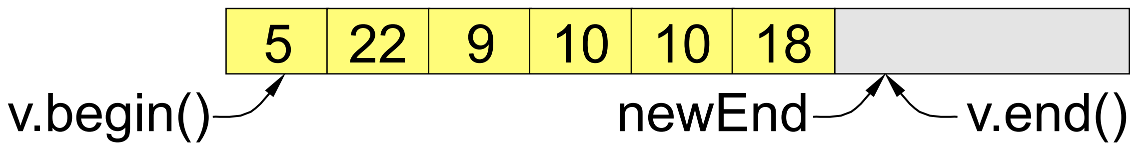


Aside: The Behavior of std::remove

`remove` often followed by `erase`.

- *This* changes size of the container.

```
v.erase(newEnd, v.cend());     // erase elements from newEnd on
```



Idiomatic to combine these calls — the *erase-remove idiom*:

```
v.erase(std::remove(v.begin(), v.end(), 16), v.cend());
```

Range vs. Single-Element Member Functions

- Range construction can be more efficient than default construction plus insertions:

```
std::deque<int> d;
...
std::vector<long> v1;
for (auto it = d.cbegin(); it != d.cend(); ++it) {
  v1.push_back(*it);                                    // bad
}
std::vector<long> v2;
v2.insert(v2.cend(), d.cbegin(), d.cend());             // better
std::vector<long> v3(d.cbegin(), d.cend());             // best
```

Sequence Containers Beyond vector

Analysis for string and deque essentially the same as for vector.

- deque never needs to do a realloc-like operation, but insertions/erasures may require moving container values up/down.

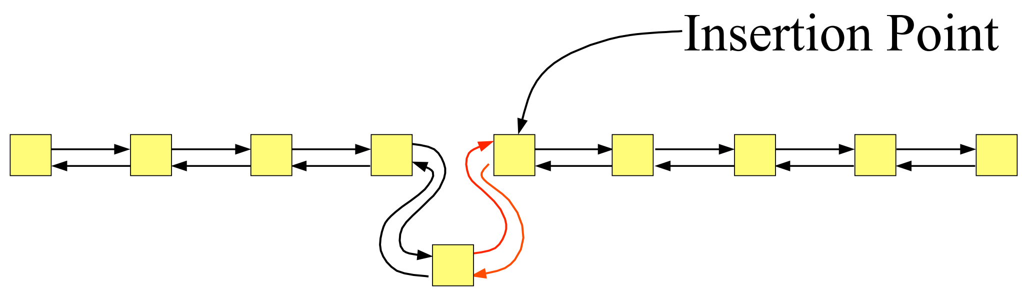
list and forward_list are different. Some problems don't arise:

- realloc-like operations don't take place, so data copying/moving not an issue.
- Insertions/erasures affect only links, so data movement not an issue.

The case for std::list

However:

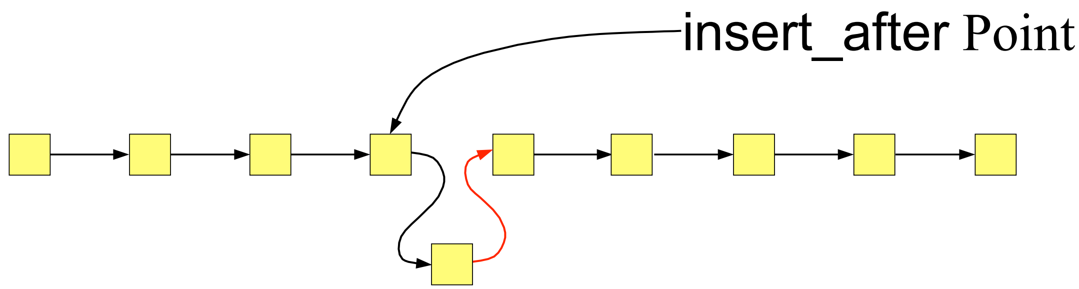
- Iterative single-element insertions require multiple adjustments of two links:
  - ➡ next pointer on new node, prev pointer on existing node.
  - ➡ *n* single-element insertions requires 2(*n*-1) superfluous pointer adjustments.
  - ➡ Range insertion of *n* elements requires none.



- Analysis is similar for erasure.

The case for std::forward_list

Essentially the same as for std::list, but with only one pointer:



Guideline

Prefer range member functions to single-element versions for sequence containers.

Prefer Function Objects to Functions

Every widget has a weight:

```
class Widget {
public:
  ...
  int weight() const;
  ...
};
```

How sort vector of Widgets by weight?

```
std::vector<Widget> vw;

...

std::sort(vw.begin(), vw.end(), ???);
```

Two ways to specify sorting criterion:

- Function
- Function object

Prefer Function Objects to Functions

Using a function:

```
inline
bool widgetLess(const Widget& lhs, const Widget& rhs)
{ return lhs.weight() < rhs.weight(); }

std::sort(vw.begin(), vw.end(), widgetLess);
```

Using a function object:

```
struct WidgetLess {
  bool operator()(const Widget& lhs, const Widget& rhs) const
  { return lhs.weight() < rhs.weight(); }
};

std::sort(vw.begin(), vw.end(), WidgetLess());
```

Differences:

- Use of the function is simpler
- Use of the function object is probably more efficient

Prefer Function Objects to Functions

The reason is inlining. Look at the declaration for sort:

```
template<class RandomAccessIterator, class Compare>
void sort( RandomAccessIterator first, RandomAccessIterator last,
           Compare comp);
```

When we pass sort a *function*, the type of comp is bool (*)(const Widget&, const Widget&):

- A function *pointer*
- Compilers rarely inline calls through function pointers
  - ➡ Even if the function is inline and visible during compilation

When we pass sort a *function object*, the type of comp is WidgetLess:

- A *class* with an (implicitly) inline operator() function
- Compilers routinely inline calls to such functions

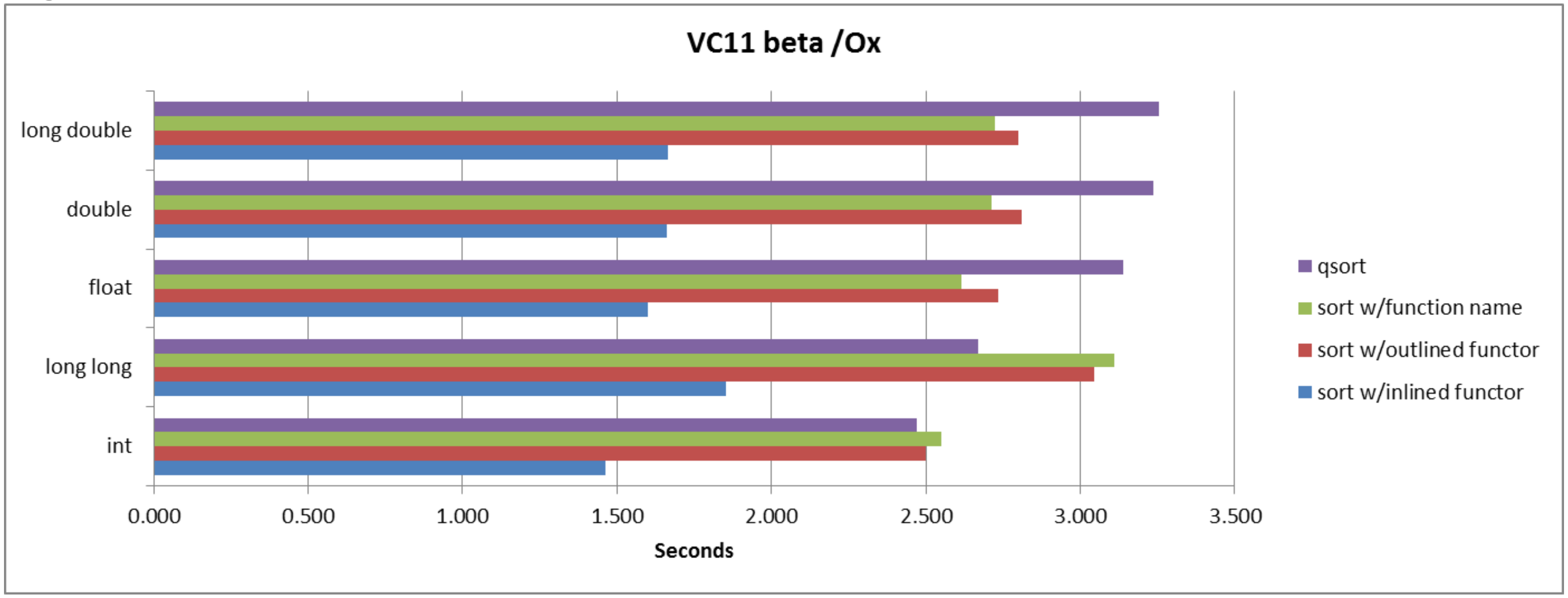
Example: std::sort vs. std::qsort

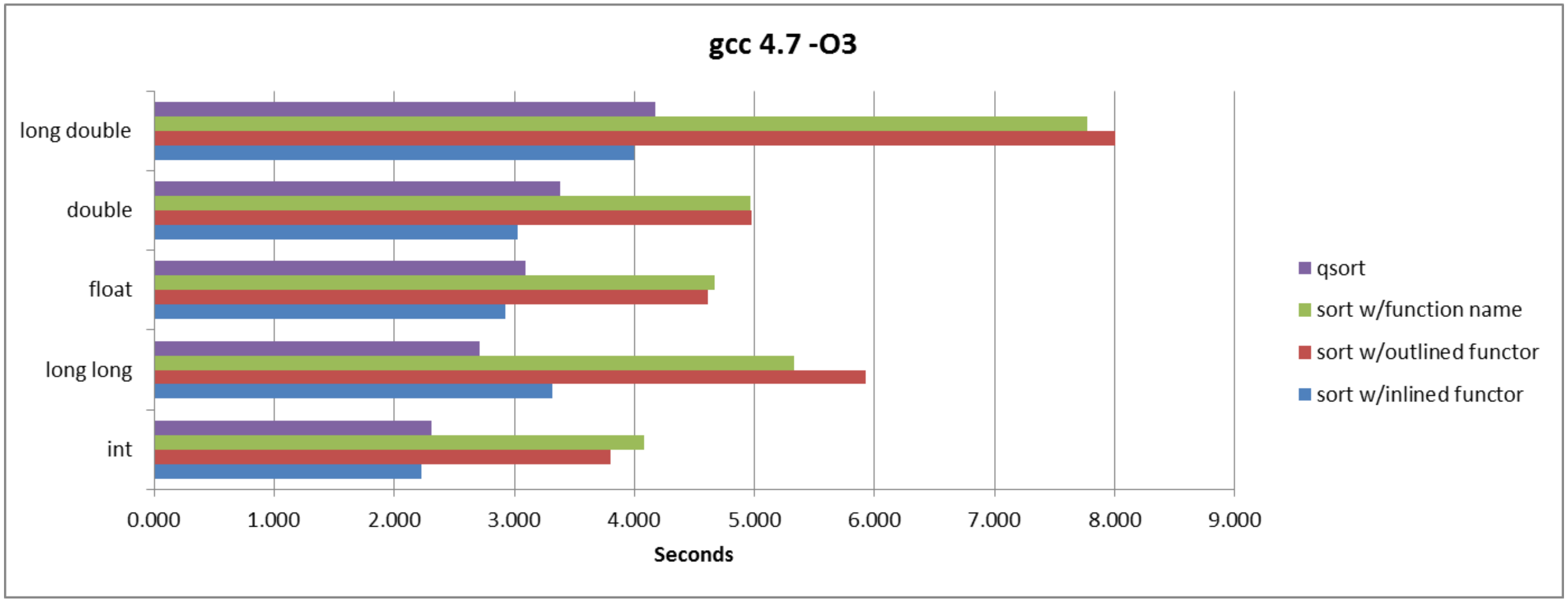
Explains why STL's sort is faster than qsort.

- sort plus a function object allows for maximal inlining
  - Important, because comparison functions are typically small, called frequently
- qsort uses function pointers and is compiled in advance.
  - Compilers can't inline, only linkers can
  - In practice, it just doesn't happen

Example: sort vs. qsort

Sorting a vector of 10,000,000 elements:





Using Lambdas

C++11's lambda expressions make function object creation easy:

```
struct WidgetLess {                                              // as before
  bool operator()(const Widget& lhs, const Widget& rhs) const
  { return lhs.weight() < rhs.weight(); }
};

std::sort(vw.begin(), vw.end(), WidgetLess());                   // C++98

std::sort(vw.begin(), vw.end(),                                  // C++11
          [](const Widget& lhs, const Widget& rhs)
          { return lhs.weight() < rhs.weight(); });
```

C++14 makes it even easier:

```
std::sort(vw.begin(), vw.end(),                                  // C++14
          [](const auto& lhs, const auto& rhs)
          { return lhs.weight() < rhs.weight(); });
```

Guideline

Prefer function objects to functions.

Consider sorted vectors for Fast Lookups

Standard associative containers have different lookup complexities:

- set/multiset/map/multimap offer $O(\log_2 n)$.
- TR1/C++11 hashed containers offer O(1).

Sorted vectors also offer $O(\log_2 n)$ lookup complexity:

- Via std::binary_search, std::lower_bound, etc.

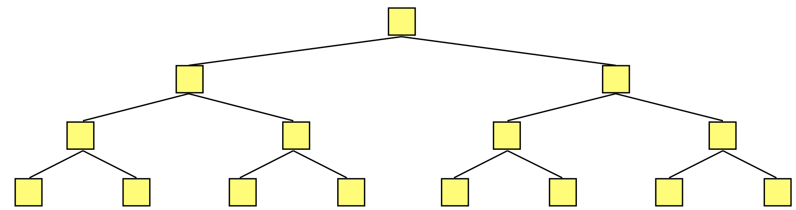
Sorted vectors often the best choice:

- Faster (e.g., 35-50% faster)
- Use less memory

vectors vs. Associative Containers

Compared to set/multiset/map/multimap:

- Binary trees.
  - ➡ Nodes add overhead to payload:
    - Pointers to children
    - Pointer to parent
    - Possibly other data (e.g., node color in red-black trees)

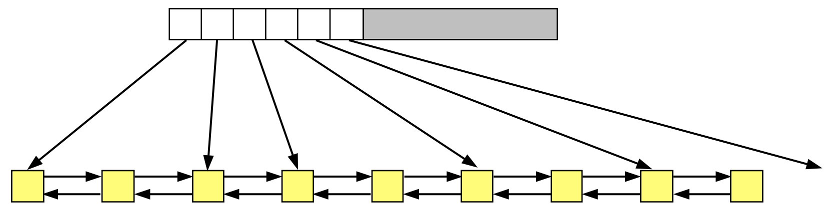


- Nodes may be scattered across memory pages.
  - ➡ Can lead to cache misses, page faults.

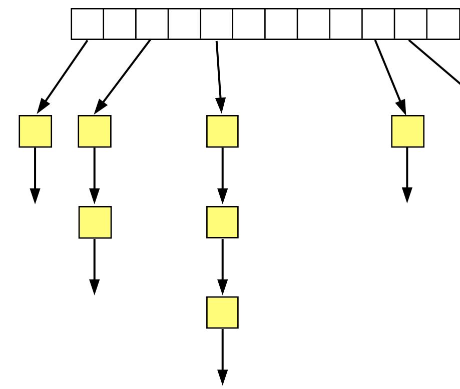
vectors vs. Associative Containers

Compared to hashed containers:

- Varying implementations, but nodes always have overhead:
  - ➡ Pointer to next bucket element.
  - ➡ Possibly pointer to previous element.



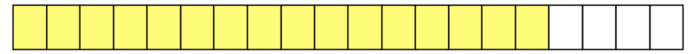
- Nodes may be scattered across memory pages.
  - ➡ Can lead to cache misses, page faults.



vectors vs. Associative Containers

Sorted vectors don't have nodes:

- No per-element overhead:



  - ➡ Maybe unused capacity, but can be ignored for lookup purposes
    - ◆ Or "shrink to fit" the vector to get rid of it.
- Elements stored in order in contiguous memory.
  - ➡ Values near one another near each other in memory.

vectors vs. Associative Containers

Sorted vectors have a different problem:

- Insertions and erasures are O(n) instead of O($\log_2$ n)!

Associative containers designed for a mixture of:

- Insertions
- Erasures
- Lookups

In software that mixes them, associative containers a good choice.

vectors vs. Associative Containers

Some software divided into phases:

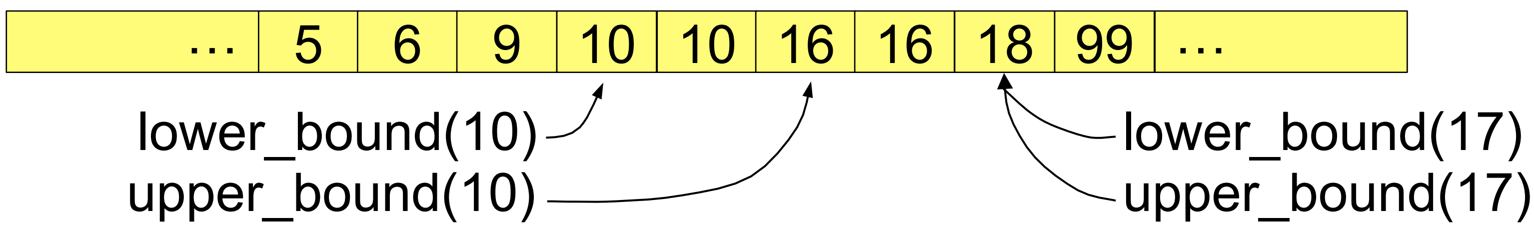
1. Setup: many insertions, maybe some erasures, few lookups.
2. Lookup: many lookups, very few insertions or erasures.
   - Cost of this phase vastly dominates.

Then sorted vectors a viable candidate:

- In phase 1, insert data into vector and sort it.
  - Various approaches are possible, e.g.:
    - Insert data, then invoke sort.
    - Insert data into set or multiset, then copy/move into vector.
  - If duplicates prohibited, maintain constraint manually.
    - E.g., invoke unique after sorting.
- In phase 2, search vector via binary_search or lower_bound.
  - The next page summarizes these algorithms.
  - Do insertions in proper location, i.e., maintain sortedness.

Binary Searches on Sorted vectors

- binary_search: returns bool, hence of limited utility.
- lower_bound or upper_bound:
  - lower_bound(v) returns where the first v is or v's proper insertion location.
  - upper_bound(v) returns one past the last v or v's proper insertion location.



Sorted vector as a set

```
std::vector<int> v;                    // emulates a set<int>
...          // phase 1: produce sorted
                                       // vector w/o duplicates
int key;                               // holds value to look up in v
...
// lookup via binary_search
if (std::binary_search(v.begin(), v.end(), key)) ...
// lookup via lower_bound
std::vector<int>::iterator it;
if ((it = std:: lower_bound(v.begin(), v.end(), key)) != v.end() && !(key < *it))
...
```

Sorted vector as a map

Design issues:

- vector holds pair objects (just like map).
- Predicate used for lookups takes two different parameter types:
  - One for the pair's first type (the key being searched for)
  - One for a pair (the "map" element being compared against)
  - Parameters may come in either order; both must be handled.
    - Two operator() functions in a single functor class.

Sorted **vector** as a **map**

For our example:

```
typedef std::vector<std::pair<int, std::string> > MapVec;

struct MapCmpFuncs {
  // comparison function for doing lookups — variation 1
  bool operator()(const std::pair<int, std::string>& p,
                  int key) const
  { return p.first < key; }

  // comparison function for doing lookups — variation 2
  bool operator()(int key,
                  const std::pair<int, std::string>& p) const
  { return key < p.first; }
};
```

Sorted vector as a map

```
MapVec m;                            // emulates a map<int, string>
...                          // phase 1: produce sorted vector
                                     // w/o duplicates
int key;                             // holds value to look up in m
...
// lookup via binary_search
if (std::binary_search(m.begin(), m.end(), key, MapCmpFuncs())) ...
// lookup via lower_bound
MapVec::iterator it;
if ((it = std:: lower_bound( m.begin(), m.end(),
                             key, MapCmpFuncs())) != m.end()
  && !MapCmpFuncs()(key, *it)) ...
```

Keeping Comparison Functions Consistent

Move the real work to one function:

```
class MapCmpFuncs {
public:
  bool operator()(const std::pair<int, std::string>& p,          // for lookups,
                  int key) const                                  // variation 1
  { return keyLess(p.first, key); }

  bool operator()(int key,                                        // for lookups,
                  const std::pair<int, std::string>& p) const    // variation 2
  { return keyLess(key, p.first); }

private:
  bool keyLess(int key1, int key2) const                          // the real work
  { return key1 < key2; }                                         // is done here
};
```

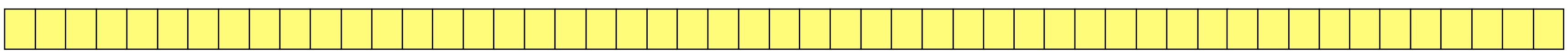
Consider Sorted vectors

Recall lookup complexities:

- set/multiset/map/multimap offer $O(\log_2 n)$.
- So does sorted vector.
- TR1/C++11 hashed containers offer O(1).

For large enough n, O(1) wins:

- Several probes in sorted vector could lead to page faults!



- At some point, hashed containers run faster.

Boost's flat_ Containers

Off-the-shelf set/map/multiset/multimap implementations that:

- Use a sorted vector as underlying storage.
- Adhere to C++11 interfaces (as much as possible).
  - E.g., support move operations, emplacement, stateful allocators, etc.

Names are:

- flat_set
- flat_map
- flat_multiset
- flat_multimap

Guideline

Consider sorted vectors for fast lookups.

Know Your Sorting Options

Everybody knows sort, but sometimes that's overkill.

Suppose you need the 20 best Widgets in a container:

- You don't need a full sort.
- You need only a partial_sort:

```
class Widget { ... };

bool qualityCompare(const Widget& lhs, const Widget& rhs)
{
  // return whether lhs's quality is better than rhs's quality
}

std::vector<Widget> widgets;
...
std::partial_sort(widgets.begin(),         // put the best 20 elements
                  widgets.begin() + 20,    // (in order) at the front of
                  widgets.end(),           // widgets
                  qualityCompare);
...
```

Know Your Sorting Options

Passing function object preferable:

- For C++11, lambda expression most natural:

```
std::partial_sort( widgets.begin(),
                   widgets.begin() + 20,
                   widgets.end(),
                   [](const Widget& lhs, const Widget& rhs)
                   { return qualityCompare(lhs, rhs); });
```

- For C++98, use hand-written functor class.
  - std::tr1::bind unlikely to inline call to comparison function:

```
std::partial_sort(
    widgets.begin(),
    widgets.begin() + 20,
    widgets.end(),
    std::tr1::bind(qualityCompare, _1, _2)    // bind obj holds pointer
);                                            // to qualityCompare
```

Know Your Sorting Options

If order unimportant, partial_sort too much work.

- nth_element puts an element in the correct place, with all other elements correctly preceding or following that element.

Values not following v in full sort	V	Values not preceding v in full sort

```
std::nth_element(widgets.begin(),          // put best 20 elements
                 widgets.begin() + 19,     // at the front of widgets;
                 widgets.end(),            // their order unimportant
                 qualityCompare);
```

With function object (C++11):

```
std::nth_element(widgets.begin(),          // same as above (C++11)
                 widgets.begin() + 19,
                 widgets.end(),
                 [](const Widget& lhs, const Widget& rhs)
                 { return qualityCompare(lhs, rhs); });
```

std::nth_element

nth_element remarkably useful:

- Find the Widget with the median level of quality.
  - I.e., the one in the middle if the vector were sorted by quality:

```
std::nth_element(widgets.begin(),                          // C++11
                 widgets.begin() + widgets.size() / 2,
                 widgets.end(),
                 [](const Widget& lhs, const Widget& rhs)
                 { return qualityCompare(lhs, rhs); });
```

std::nth_element

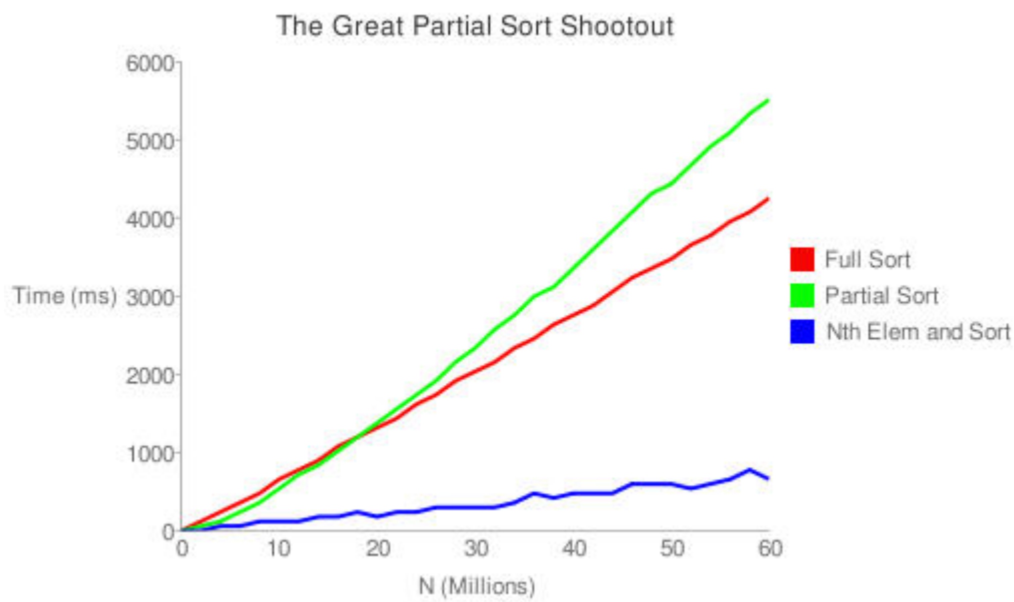
- Find Widget with a level of quality at the 75th percentile:

```
std::nth_element(widgets.begin(),                          // C++11
                 widgets.begin() + 0.25 * widgets.size(),
                 widgets.end(),
                 [](const Widget& lhs, const Widget& rhs)
                 { return qualityCompare(lhs, rhs); });
```

nth_element + sort vs. partial_sort

nth_element also useful for finding endpoint for "first n" sort.

- nth_element + sort can be much faster than partial_sort.
  - ➡ Thomas Guest's data on finding first 10% of N 32-bit integers:



- General approach (first 10%, default comparison function):

```
container::iterator begin = container.begin();
container::size_type offset = 0.10 * container.size();
std::nth_element(begin, begin + offset, container.end());
std::sort(begin, begin + offset);
```

Know Your Sorting Options

To find all Widgets with a quality of 2 or better.

- sort and partial_sort do more work than you need.
- nth_element doesn't really do what you want.
- partition does:

```
bool hasOKQuality(const Widget& w)
{
  // return whether w has a quality rating of 2 or better;
}

std::vector<Widget>::iterator goodEnd =   // move Widgets satisfying
  std::partition( widgets.begin(),        // hasOKQuality to front
                  widgets.end(),          // front of widgets; return
                  hasOKQuality);          // iterator to first Widget
                                          // that isn't satisfactory
```

Desired Widgets now between widgets.begin() and goodEnd.

Know Your Sorting Options

As usual, passing function object preferable:

```
std::vector<Widget>::iterator goodEnd =                      // C++11
  std::partition( widgets.begin(),
                  widgets.end(),
                  [](const Widget& w){ return hasOKQuality(w); });
```

Stability

When Widgets have equivalent quality ratings, how do sort, partial_sort, nth_element, and partition order them?

- Any way they want to.
  - You can't change this.

Sometimes you want *stability*:

- Relative positions of equivalent elements guaranteed to be preserved during reordering.

sort not stable, but stable_sort is.

partition not stable, but stable_partition is.

partial_sort and nth_element not stable.

- The STL offers no stable versions of these algorithms.
- For stability, typically use stable_sort.

Iterator Requirements

sort, stable_sort, partial_sort, nth_element require random access iterators:

- Applicable to vectors, strings, deques, valarrays, and arrays.
- Also TR1/C++11 std::arrays.

partition/stable_partition require only bidirectional iterators.

- Applicable to all standard sequence containers except std::forward_list.

Makes no sense to try to sort the standard associative containers.

Sorting std::lists

list has sort member function:

- Performs a stable sort.
- Options for using other sorting algorithms on data in a list:
  - Copy/move list data into container with random access iterators, sort that, copy/move data back.
  - Create container of list::iterators, sort that, access list elements via the iterators.
  - Create container of list::iterators, sort that, use it to iteratively splice list elements into desired order.

Summary of Sorting Options

For sequence containers other than list:

- For full sort, use sort or stable_sort.
- For first *n* elements, use partial_sort or nth_element + sort.
- For element at position *n* or first *n* elements (ignoring order), use nth_element.
- For elements that do and don't satisfy a criterion, use partition or stable_partition.

For list:

- Use list::sort in place of sort and stable_sort.
- Use partition and stable_partition directly.

In addition:

- set/multiset/map/multimap keep things sorted all the time.
- So does priority_queue.

Relative Efficiency of the Sorting Algorithms

In general,

- More work takes longer.
- Stability costs.

From fewest resource demands to most:

1. partition
2. stable_partition
3. nth_element
4. partial_sort
5. sort
6. stable_sort

Sorting Under the Hood

Standard doesn't dictate implementations, but typically:

- sort implemented via *introsort*.
  - Usually quicksorts, but on pathological inputs, switches to heapsort to maintain $O(n \log_2 n)$ worst case performance.
- partial_sort implemented via heapsort.
  - Usually slower than sort (for full sort).
- stable_sort implemented via mergesort.
  - Also slower than sort (for full sort).
- nth_element implemented via modified quicksort (possibly *introselect*).
  - Runs in linear time on average.
- list::sort implemented via mergesort.

Guideline

Know your sorting options.

C++ Library Summary

- Use reserve to reduce reallocations in vector and string.
- Use the swap trick or shrink_to_fit to reduce excess capacity in vectors, strings, and deques.
- Prefer range member functions to single-element versions for sequence containers.
- Prefer function objects to functions.
- Consider sorted vectors for fast lookups.
- Know your sorting options.

Consider use of Concurrent Data Structures

Concurrent use of STL data structures (DSes) limited:

- Safe to concurrently read/modify separate DSes.
- Safe to concurrently read a single DS.
- Safe to concurrently read/write independent elements in a DS.
  - DS *structure* unaffected.
- **Modification of an STL DS never concurrency-safe.**
  - E.g., concurrent insert/erase, insert/traversals, erase/size, etc.

All above formalized in C++11.

- De facto guaranteed in C++98 by library implementers.

Consider use of Concurrent Data Structures

Concurrent data structures:

- Permit some concurrent operations during some modifications.
  - E.g., concurrent inserts, concurrent insert/traversal, etc.
- Some ≠ all!
  - Details dependent on DS API, e.g., for TBB/PPL:
    - concurrent_unordered_map supports insert + traversal, not insert + erase.
    - concurrent_hash_map supports insert + erase, not insert + traversal.

Terminology

"Concurrent data structure" an informal term.

Standard terms per other threads' behavior if a thread's "delayed:"

- Wait-free: *All* continue to *progress*.
  - ➡ No starvation, deadlock, livelock, priority inversion.
- Lock-free: *At least one* continues to *progress*.
  - ➡ No deadlock, livelock, priority inversion.
- Obstruction-free: *A single thread will progress if all other threads are suspended*.
  - ➡ Requires ability to abort/rollback other threads' actions.
  - ➡ Livelock is possible.
- Non-blocking: All continue to *run*.
  - ➡ Starvation, deadlock, livelock, priority inversion all possible.
    - ◆ E.g., spinlocks.

Wait-free ⊂ lock-free ⊂ obstruction-free ⊂ non-blocking.

Terminology

Related terms:

- Thread-safe: Safe for use in an MT environment.
  - All concurrent invocations yield valid results.
    - Invariants maintained, postconditions satisfied.
  - May or may not use locks.
  - Applicable to both functions and DSes.
  - Applies only to *individual* operations.

```
if (threadSafeDS.size() > 1)        // size is thread-safe
  threadSafeDS[0] = 0;              // unsafe! size may now be 0!
```

- Synchronized: Usually a thread-safe DS.
  - Typically uses an internal lock per DS object.
  - E.g., Java synchronized collections.

Consider use of Concurrent Data Structures

"Concurrent data structures" generally:

- **Lock-free** or
- **Based on invisible locking**.
  - E.g., TBB concurrent_unordered_map documentation:

    The interface has no visible locking. It may hold locks internally, but never while calling user defined code.

Implications:

- Should be no deadlock, livelock, or (for lock-free) priority inversion.

Consider use of Concurrent Data Structures

Mutex + "normal" DS ≠ concurrent DS.

- "Concurrent" access serialized.
  - No reader-writer mutex in standard C++ until C++14.
  - Boost has `shared_mutex`, but writing still exclusive.
    - Ditto for `std::shared_timed_mutex` in C++14.

Concurrent DSes permit (some) *concurrent* operations.

- At least one of which modifies the DS.

Consider use of Concurrent Data Structures

Scalability is primary motivation for concurrent DSes.

- Use of per-DS mutex can limit scalability.
  - ➡ E.g., for large number of threads.
  - ➡ E.g., for few threads with very frequent DS access.
- With concurrent DSes, no mutex needed for concurrent ops.

From *Java Concurrency in Practice*†:

> The principal threat to scalability in concurrent applications is the exclusive resource lock.

† *Java Concurrency in Practice*, Brian Goetz, Addison-Wesley, 2006, ISBN 0-321-34960-1, section 11.4.

Consider use of Concurrent Data Structures

Sample potential use cases:

- Queues in producer-consumer systems.
  - SPSC in realtime logging software.
  - SPSC and MPSC queues in financial trading software.
- Hash tables in MT servers:
  - Cache commonly-requested items in proxy servers.
- Hash tables in web administration software:
  - Eliminate duplicates from proxies' web logs of clients.
- Hash tables in parallel dynamic programming algorithms:
  - Threads share cached ("memoized") results.

TBB rule of thumb:

- Concurrent DSes good when access patterns can be bursty.

Consider use of Concurrent Data Structures

More scalable ≠ better.

- When DS not a bottleneck, scalability unimportant.
- For serial use, concurrent DSes usually slower.
- In largely serial use, “normal” DS + mutex may be faster.
- Tipping point for concurrent DSes affected by:
  - ➨ Mix of DS operations.
  - ➨ Library implementations (e.g., DSes, mutexes, etc.)
  - ➨ Build system (e.g., compiler, linker, etc.)
  - ➨ Hardware platform.

Surest approach:

- Compare serial and concurrent DSes empirically.

Consider use of Concurrent Data Structures

No "standard" concurrent DSes:

- None in C++98, TR1, C++11, or C++14.

However:

- Intel's TBB (Threading Building Blocks) library has some.
  - ➨ Available in open-source and commercial versions.
- Microsoft's PPL (Parallel Patterns Library) has some.
  - ➨ Ships with Visual C++ 2010 and later.
- Boost.Lockfree has some.
  - ➨ As of Boost 1.53.

Commonly Available Concurrent DSes

TBB and PPL have some DSes in common.

- Intel/Microsoft pledge to make them "identical."
  - concurrent_queue (MPMC)
  - concurrent_vector
  - concurrent_unordered_map

Boost.Lockfree offers:

- queue (MPMC, lock-free)
- spsc_queue (SPSC, wait-free).
  - Ringbuffer.
- stack (MPMC, lock-free)

Following overview focuses on TBB/PPL offerings.

- Boost queues offer similar API to TBB/PPL concurrent_queue.

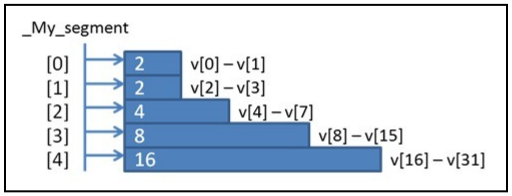
TBB/PPL Concurrent Data Structures

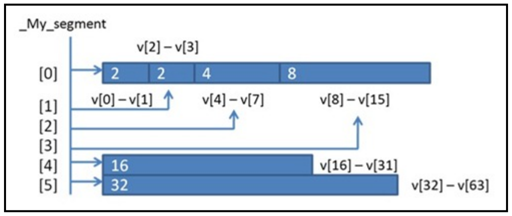
concurrent_queue: lock-free unbounded queue.

- Supports *n* concurrent producers + *m* concurrent consumers.
- Any mix of adding/erasing elements concurrency-safe.
- Traversal, clear, unsafe_size, etc. not concurrency-safe.
- No blocking on reads from empty queue.
  - ➡ Reads return success flag; reads on empty queues get false.
- Useful for producer/consumer when polling readers okay.

TBB/PPL Concurrent Data Structures

concurrent_vector: lock-free extendable array.





- Using and appending elements concurrency-safe.
  - ➡ Elements don't move when concurrent_vector grows.
    - ◆ Layout not a contiguous block of memory.
- reserve, clear, shrink_to_fit, etc. not concurrency-safe.
- API notably different from std::vector:
  - ➡ No insert or erase.
  - ➡ Elements undergoing construction may be visible:
    - ◆ size may include them.
    - ◆ Traversals may visit them.
    - ◆ operator[]/at may return references to them.
- Useful for never-shrinking sequence supporting random access and concurrent growth + element use.

TBB/PPL Concurrent Data Structures

concurrent_unordered_map: MT-friendly hash table.

- Based on "invisible" locking.
  - ➡ Not lock-free, but still no deadlock or livelock.
- Insertion, lookup, traversal concurrency-safe.
- Erasure and bucket methods not concurrency-safe.
- Useful for concurrently created/searched hash table.

TBB/PPL Concurrent Data Structures

APIs differ from STL counterparts.

- concurrent_*container* not just std::*container* + concurrency.
  - Some STL functionality missing, e.g.,
    - No pop in concurrent_queue, no erase in concurrent_vector or concurrent_unordered_map.
  - Some STL operations have different semantics, e.g.,
    - concurrent_vector::size counts not-yet-constructed elements.
  - New functionality present, e.g.,
    - concurrent_vector has grow_by and grow_to_at_least.

concurrent_*container* not a drop-in replacement for std::*container*!

TBB/PPL Concurrent Data Structures

Sample API changes:

- std::queue has pop, concurrent_queue has try_pop:

```
std::queue<int> q;                // from STL

if (!q.empty()) {
  int val = q.front();            // not MT safe: q may be empty
  q.pop();                        // ditto

  use val...
}

concurrent_queue<int> cq;         // from TBB/PPL

int val;
if (cq.try_pop(val)) {            // test/pop in atomic operation
  use val...
}
```

- Boost.Lockfree's queues retain pop name, employ try_pop's API.

TBB/PPL Concurrent Data Structures

- C++11/TR1's unordered_map has erase, concurrent_unordered_map has unsafe_erase:

```
std::unordered_map<int, std::string> um;   // from C++11 STL
...
um.erase(10);                               // MT-unsafe, but not
                                            // surprising


concurrent_unordered_map<int, std::string> cm;   // from TBB/PPL
...
cm.unsafe_erase(10);                        // MT-unsafe, but
                                            // could surprise
```

➡ Not all TBB/PPL concurrency-unsafe functions start with unsafe_.

TBB/PPL Concurrent Data Structures

Removing elements:

- Not possible in concurrent_vector.
- Not concurrency-safe in concurrent_unordered_set.

Implications:

- Consider other DSes when element removal is needed.
  - Some DSes are insert-only (e.g., log merges).

Workarounds:

- concurrent_vector: put special value into "erased" elements.
  - E.g., 0/nullptr for pointers, -1 for only-positive ints, etc.
- concurrent_unordered_set: distinct program phases:
  - Phase A (MT): insertions, lookups, traversals. No removals.
  - Phase B (ST): Removals + other MT-unsafe operations.
    - E.g., MT cache insertions + ST cache trimming.

Additional TBB Concurrent Data Structures

In TBB but not PPL:

- concurrent_bounded_queue:
  - Optionally bounded version of concurrent_queue.
  - Writers block on full queues, readers may on empty queues.
- concurrent_hash_map:
  - Element removal thread-safe.
  - Traversal while inserting/removing not thread-safe.
  - Locking during access visible to clients.
    - Deadlock, livelock, priority inversion possible.

Consider use of Concurrent Data Structures

Concurrent DSes ≠ concurrency silver bullet.

- Thread safety analysis still crucial.
  - ➡ Not all operations on concurrent DSes MT-safe.
  - ➡ Transactional sets of operations still problematic:

```
concurrent_vector<int> cv;
...
auto it = std::find_if(cv.begin(), cv.end(),                  // find 1st even
                       [](int x) { return x%2 == 0; });       // value in cv
if (it != cv.end()) {
  process *it as even number...                               // *it's value may
}                                                             // now be odd
```

Concurrent DSes simply one tool in the concurrency toolbox.

Custom Concurrent Data Structures

Creation of custom concurrent DS worth considering if:

- Been shown that concurrent DS is needed.
  - E.g., scalability of non-concurrent DSes unacceptable.
- Library DSes (e.g., from TBB/PPL/Boost) don't suffice:
  - Inadequate performance (e.g., speed or scalability).
  - Unsuitable APIs (e.g., "wrong" set of concurrent operations).
  - Unavailable on platform.

Should be a last resort.

- Concurrent DS design/implementation *hard*.
  - General correctness (including, e.g., exception-safety).
  - Concurrency correctness.
    - E.g., dealing with relaxed memory visibility.
    - E.g., avoiding ABA problem.
  - Performance (e.g., latency/scalability of operations).

Concurrent DS creation a job for experts.

CAS

Compare-and-swap (CAS) central to many lock-free algorithms.

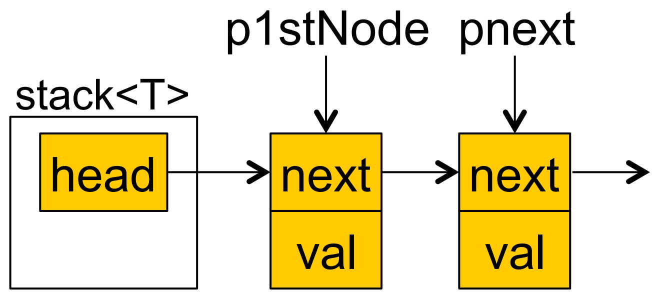
```
template<typename T>
bool CAS( T* pCurrent, T* pExpected, T desired)
{
  if (*pCurrent == *pExpected) {      // if *pCurrent has expected
    *pCurrent = desired;              //  value, replace it with desired
    return true;
  }

  return false;
}
```

- **Function executes atomically.**
- Actually an *assignment*, not a swap.
- Signatures and detailed semantics vary.
  - C++11 has compare_exchange_weak and compare_exchange_strong.
    - Semantics slightly different from above.

The ABA Problem

CAS in loops common.

- Consider stack<T> implemented as linked list.

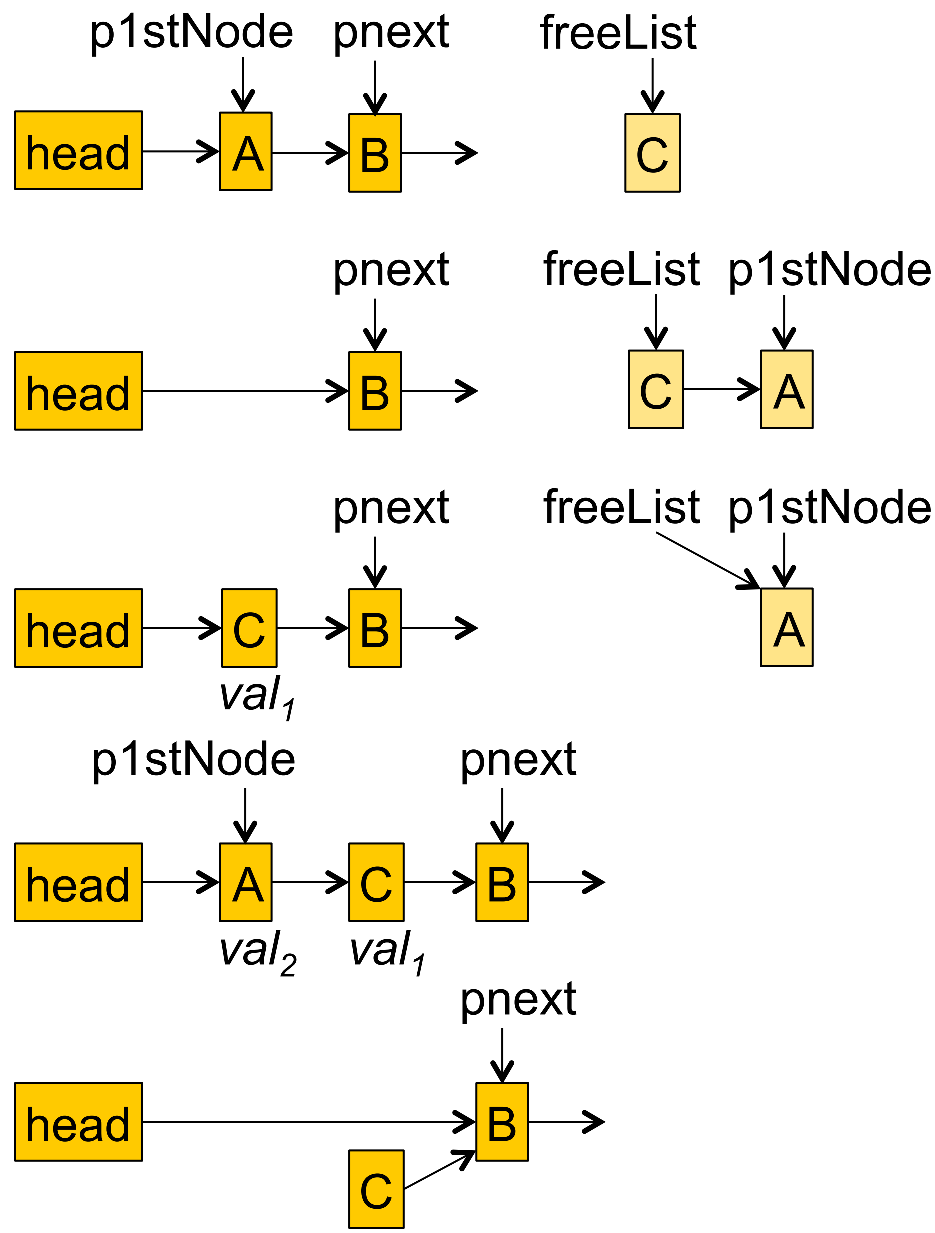



```
template<typename T>                    // if stack not empty, pop top
bool stack<T>::pop(T& value)            // element into val; return
{                                       // whether element popped
  Node *p1stNode;

  do {
    p1stNode = head;
    if (!p1stNode) return false;
  } while !CAS(&head, &p1stNode, p1stNode->next);   "pnext"

  value = std::move(p1stNode->val);

  release *p1stNode;

  return true;
}
```

The ABA Problem

```
stack<int> s;
…
```

```
// Thread 1          // Thread 2
int x;               int y;
s.pop(x);            s.pop(y);
                     s.push(val1);
                     s.push(val2);
```



Guideline

Consider use of concurrent data structures.

Consider use of Parallel Algorithms

"Algorithms" in STL sense (not CS sense).

- Function templates, typically operating on ranges.

Approach:

```
parallelAlgorithm(begin, end);      // do something with elements
                                    // in [begin, end)
```

- Break [begin, end) into subranges.
- Process subranges in parallel.

Parallel approach often faster than serial when:

- Thread management cheaper than work per subrange.
- Platform has available resources.
  - ➡ Subranges actually processed concurrently.

For correctness, subrange processing must be independent.

- No order dependencies.

Consider use of Parallel Algorithms

Some design/implementation issues nontrivial:

- **Thread management** (e.g., avoiding oversubscription).
- **Dealing with exceptions**.

```
parallel_sort(container.begin(), container.end());
```

  - ➡ What's propagated if concurrent subrange sorts throw?
  - ➡ How avoid superfluous work if exception thrown?
    - ◆ Unsorted subranges need not be processed.

Library implementations typically a better choice.

- As with STL.

Consider use of Parallel Algorithms

No "standard" parallel algorithms:

- None in C++98, TR1, C++11, or C++14.
- None in Boost.

TBB and PPL each have some.

- Both have parallel_for_each, parallel_sort, parallel_invoke, parallel_for.
- Each have additional algorithms.

TBB/PPL Parallel Algorithms and Exceptions

Consider this call:

```
parallel_algorithm(range specification);     // process subranges
                                             // in parallel
```

If exceptions arise during processing, TBB/PPL:

- Allow only currently running subrange processing to continue.
  - E.g., in divide-and-conquer algorithms, no further recursion.
  - Subranges not currently being processed won't be processed.
- Propagate "first" exception to parallel_algorithm's caller.
  - Other "concurrent" exceptions swallowed.

TBB/PPL Parallel Algorithms

parallel_for_each: parallel version of std::for_each:

```
void process(int x);                                   // process x in
                                                       // some way

std::vector<int> v;
...
std::for_each(v.cbegin(), v.cend(), process);          // serial version
parallel_for_each(v.cbegin(), v.cend(), process);      // parallel version
```

- Implementation decides whether to actually use parallelism.
- As usual, function objects preferable to functions, e.g.:

```
parallel_for_each(v.cbegin(), v.cend(), [](int x) { process(x); });
```

TBB/PPL Parallel Algorithms

parallel_sort: parallel version of std::sort:

```
std::vector<int> v;
...
std::sort(v.begin(), v.end());                 // serial version
parallel_sort(v.begin(), v.end());             // parallel version
```

- Implementation decides whether to actually use parallelism.

TBB/PPL Parallel Algorithms

parallel_invoke: runs given functions/functors in parallel.

```
void initSubsystemA();
void initSubsystemB();
void initSubsystemC();

initSubsystemA();                               // serial version
initSubsystemB();
initSubsystemC();

parallel_invoke(initSubsystemA,                 // parallel version
                initSubsystemB,
                initSubsystemC);
```

- Implementation decides whether to actually use parallelism.
- Functions' return values (if any) ignored.
- Exception from init. function ⇒ no further parallel invocations.
- Passing lambdas instead of function pointers a potential performance improvement.

TBB/PPL Parallel Algorithms

parallel_invoke takes only argumentless functions:

```
void doThis(int, const char*);
void doThat(double);

parallel_invoke(doThis, doThat);                  // error!

parallel_invoke(doThis(10, "Hello"),              // error!
                doThat(4.5));
```

Lambdas are a C++11 workaround

```
parallel_invoke([] { doThis(10, "Hello"); },      // fine
                [] { doThat(4.5); } );
```

bind from TR1, Boost, or C++11 works, too:

```
parallel_invoke(bind(doThis, 10, "Hello"),        // also fine
                bind(doThat, 4.5));
```

TBB/PPL Parallel Algorithms

parallel_for: parallel version of for loop:

```
void process(int x);                          // process x in
                                              // some way

int lowBound, highBound;
...                                     // give lowBound and
                                              // highBound values

for (int i = lowbound; i < highBound; ++i) {  // serial version
  process(i);
}

parallel_for(lowBound, highBound, process);   // parallel version
```

- Implementation decides whether to actually use parallelism.
- Note use of indices, not iterators.

OpenMP

An extralinguistic alternative:

- Based on #pragmas.
  - Supported by many C++ compilers.
  - Ignored by non-OpenMP compilers.
    - Code ports, parallel directives ignored (⇒ serial execution).
- Simple loop optimization straightforward:

```
#pragma omp parallel for                  // if OpenMP supported,
for (std::size_t i = 0; i < v.size(); ++i) {  // run loop iterations
  process(i);                             // in parallel (otherwise
}                                         // serially as usual)
```

  - Often lower overhead than TBB/PPL approach.
- Offers more than simple loop parallelization:
  - Data isolation across loop iterations (private variables).
  - Reductions.
  - Support for divide and conquer algorithms.
  - Various thread scheduling policies.

OpenMP

No exception support:

```
#pragma omp parallel for
for (std::size_t i = 0; i < v.size(); ++i) {
  process(i);                              // UB if process throws
}                                          // (typically crashes)
```

Possible solution: swallow exceptions, set failure flag:

```
std::atomic<bool> exInLoop(false);         // std::atomic from C++11
#pragma omp parallel for
for (std::size_t i = 0; i < v.size(); ++i) {
  try { process(i); }
  catch (...) { exInLoop = true; }
}
if (exInLoop) ...                          // error handling here
```

- All iterations run, even if one/some throw.
  - ➡ Minor code change reduces cost of unnecessary iterations:

```
try { if (!exInLoop) process(i); }         // do nothing if an except.
                                           // has been thrown
```

OpenMP

Alternative: propagate one of the exceptions:

```
std::exception_ptr xp;                   // from C++11; xp is "null"
std::mutex xpm;                          // mutex protecting xp

#pragma omp parallel for
for (std::size_t i = 0; i < v.size(); ++i) {
  try { process(i); }
  catch (...) {
    std::lock_guard<std::mutex> g(xpm);      // lock xpm
    xp = std::current_exception();
  }                                          // unlock xpm
}
if (xp) std::rethrow_exception(xp);          // propagate one of
                                             // thrown exceptions
```

- All iterations run, even if one/some throw.

Additional TBB Parallel Algorithms

For ranges defined by iterators:

- parallel_do: like parallel_for_each, but range can be augmented during processing.
  - E.g., recursively walk tree using only children at each level.

For ranges defined by range object:

- parallel_reduce: akin to std::accumulate (but different API).
- parallel_scan: akin to std::partial_sum (but different API).

Additional PPL Parallel Algorithms

All over ranges defined by iterators:

- parallel_transform
- parallel_reduce: akin to std::accumulate.
- parallel_buffered_sort
- parallel_radixsort

Guidance for choosing among parallel_sort, parallel_buffered_sort, and parallel_radixsort in Further Information.

Guideline

Consider use of parallel algorithms.

Exploit "Free" Concurrency

Processors increasingly offer multiple cores:

- Hyperthreading, when available, has multiplicative effect.
- Multicore now common, manycore possibly coming.

Standard question:

- How harness cores to do what needs to be done?

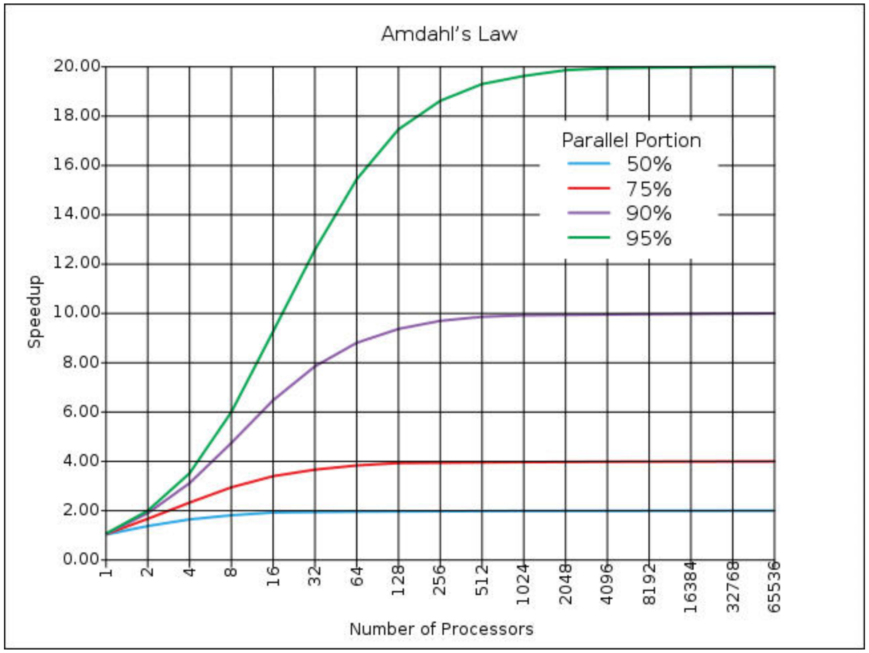
Problems:

- May not be enough work for available cores.
- Amdahl's law may limit what can be done concurrently.

Amdahl's Law

Parallelization speedup limited by sequential bottlenecks:

$$\text{Speedup} = \frac{1}{(1-P) + \frac{P}{S}}$$



Russell Williams on Photoshop's experience with parallelization:

> The scaling limitations imposed by Amdahl's law have become all too familiar....Between each of the steps that process the image data in parallelizable chunks, there are sequential bookkeeping steps. ... Amdahl's law quickly transforms into Amdahl's wall.

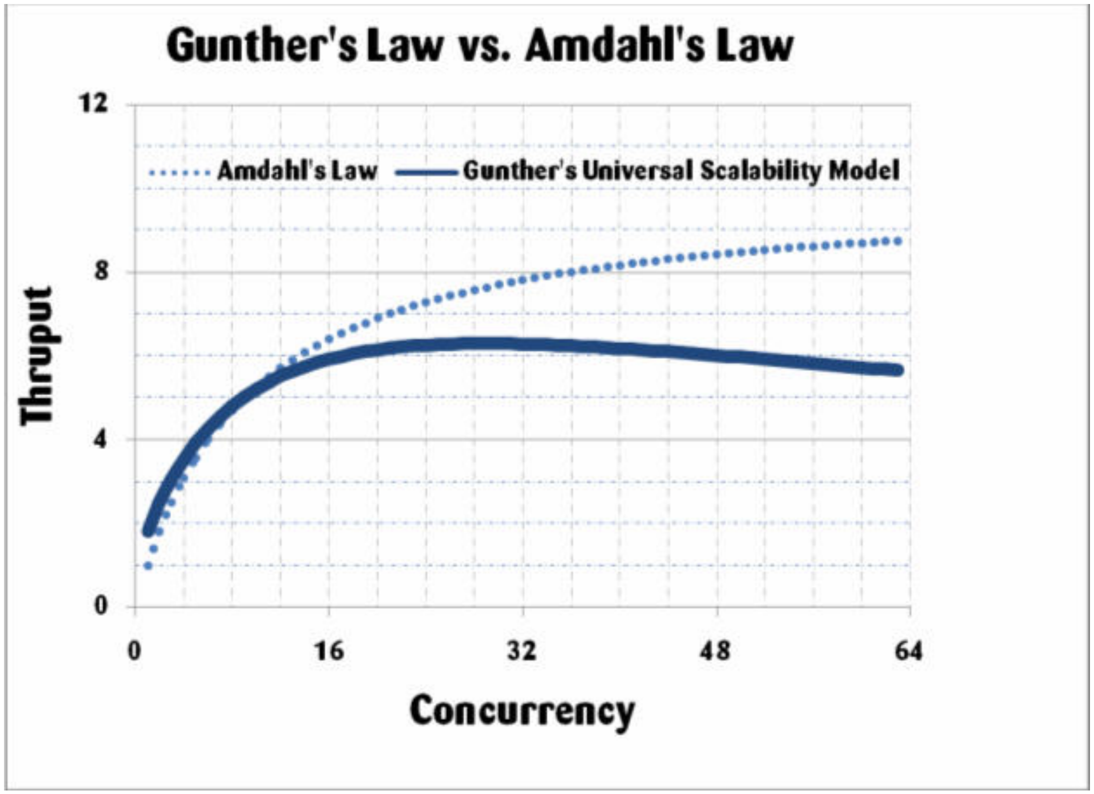
Gunther’s Universal Scalability Law

Neil Gunther’s law adds a factor for *coherence*:

- Overhead for keeping shared mutable data coherent.
  - E.g., for mutexes, atomic instructions, cache coherency, etc.

Resulting law predicts *negative* scalability at high processor counts:

- E.g., graph for when P = .9 (program that’s 90% parallelizable):



Exploit "Free" Concurrency

New question:

- How harness cores to do what *doesn't* need to be done?
  - ➡ Unnecessary work can be useful!
    - "Free" concurrency ⇒ such work is "free."

Only CPU Concurrency is "Free"

Even when cores and L1 caches free, most system resources shared:

- L2 and higher caches.
- Memory bus.
- Main memory.
- Network.
- Disks and other peripherals.

"Free" work on "extra"cores can slow essential work elsewhere.

- May also use more power, generate more heat, etc.
  - ➡ Which may cause system clock to slow down!

"Free" only free for small values of free.

Exploiting “Free” Concurrency

Still a worthwhile question:

- How take advantage of extra processing power?

We’ll consider two ideas:

- Multiple-approach problem solving.
- Speculative execution.

Multiple-Approach Problem Solving

Some problems solvable in multiple ways, e.g.,

- Graph search: DFS, BFS, random walk, etc.
- Sorting: quicksort, mergesort, heapsort, etc.
- Lossless compression: dictionary coding, entropy encoding, etc.
- Edge detection: search-based, zero-crossing based, etc.
- Audio transcription: speech recognition, Mechanical Turk, etc.

Typically, no approach best for all inputs.

Multiple-Approach Problem Solving

Traditional (do-only-what's-necessary) solutions:

- Apply approach that's often best (and rarely bad).
- Examine data to determine which approach to use, e.g.,
  - Data looks like text ⇒ text compression algorithm.
  - Data looks like audio ⇒ audio compression algorithm.

"Concurrency is free" solution:

- Run multiple approaches simultaneously.
- Use "best" result, e.g.
  - Arrives soonest (lowest latency).
  - Most compact output.
  - Etc.

Multiple-Approach Problem Solving

Example: Search graph g for a node satisfying predicate p.

- E.g., a node with value between 9 and 27.

Assume we have:

```
Node* dfs(const Graph& g, const Predicate& p,    // depth-first
          std::atomic<bool>& doneFlag);          // search

Node* bfs(const Graph& g, const Predicate& p,    // breadth-first
          std::atomic<bool>& doneFlag);          // search

Node* rws(const Graph& g, const Predicate& p,    // random walk
          std::atomic<bool>& doneFlag);          // search
```

- doneFlag:
  - Set to true when suitable node found.
  - Polled periodically. If true, search aborted.
- Functions return null if no node found (including aborted runs).

Multiple-Approach Problem Solving

Simple approach (C++11 concurrency API):

```
Node* searchGraph(const Graph& g, const Predicate& p)
{
  std::atomic<bool> doneFlag(false);
  std::vector<std::future<Node*>> futures;

  futures.push_back(std::async([&] { return dfs(g, p, doneFlag); }));    // start
  futures.push_back(std::async([&] { return bfs(g, p, doneFlag); }));    // all
  futures.push_back(std::async([&] { return rws(g, p, doneFlag); }));    // searches

  Node* result = nullptr;                                                // wait for
  for (std::size_t i = 0; i < futures.size(); ++i) {                     // each to
    Node* thisResult = futures[i].get();                                 // return
    if (thisResult != nullptr) result = thisResult;
  };

  return result;                                   // return any non-null result
}                                                  // (if there is one)
```

Multiple-Approach Problem Solving

Real-world considerations:

- **Exceptions**: one or more searches might throw.
  - Shown code propagates exception even if a search succeeds.
    - Might want to propagate only if *all* searches fail.
    - Putting gets in try blocks would allow this.
- **Latency**: searchGraph returns only after all searches finish.
  - Might want to use result as soon as *any* search succeeds.
  - searchGraph could wait on a condvar or std::future<void>.
    - Search threads would notify or set_value on success.
    - When waked, searchGraph would poll for ready future.
    - searchGraph works with result rather than returning it.

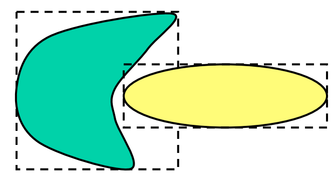
Multiple-Approach Problem Solving

Facilitates use of heuristic algorithms that may yield false negatives.

- Run concurrently with slower, more accurate algorithms.

Examples:

- **Find two non-intersecting shapes.**
  - Fast heuristic: no shape intersection ⇒ bounding boxes don't overlap.
  - False negative on non-intersecting shapes w/overlapping bounding boxes.



- **Find an audio file.**
  - Fast heuristic: audio file ⇒ file extension is mp3.
  - False negative on non-mp3 audio files.

Speculative Execution

Performing operations that *might* be requested later.

- Prefetching a common example:
  - Web pages via links in current page.
  - Images via directory traversal.

Idea more general than prefetching and is common in hardware:

- CPU branch prediction and multipath execution.

Speculative Execution

With spare cores, do what *might* be requested. Ideas:

- **Documents:**
  - ➡ Format for other devices (e.g., A4 PDF, Kindle, iPod, iPad); translate to other languages; generate speech (TTS); etc.
- **Images:**
  - ➡ Resize; convert to greyscale; compress; edge detection; etc.
- **Video:**
  - ➡ Encode for iOS, Android; extract audio; object detection; etc.
- **Financial data:**
  - ➡ Orders that anticipate market moves.
- **Source code, bytecode:**
  - ➡ Refactorings; compile/optimize (pre-JIT); etc.
- **Libraries:**
  - ➡ Compute results based on call history.

Speculative Execution

Possible bases for speculation:

- Expected use patterns.
- Community use history (per collected usage stats).
- Per-user use history (per collected usage stats).

Speculative Execution

Minimizing cost of "free" multicore concurrency:

- Reduce likelihood of wasted speculation.
  - E.g., monitor and adapt to use history.
- Minimize impact on shared resources.
  - Prefer CPU-intensive work.
  - Run with reduced priority.
  - Abort ASAP.
    - Alas, no C++11 support for interruptible threads.
    - See *C++ Concurrency in Action* for manual implemenation.

And of course:

- Avoid side effects!
  - One essentially unavoidable: variability of latency increases.

Extra Nodes ≅ Extra Cores

"Extra" nodes in clusters, server farms, grids, etc. also exploitable:

- Also good for multiple approaches, speculative execution.
- Also only partly free.
  - ➡ Network still shared.

New type of speculative execution: redundant computation:

- Run same job on multiple nodes.
  - ➡ Compensates for node crashes, slow disk controllers, etc.
  - ➡ E.g., Hadoop MapReduce's speculative execution.
    - ◆ Slow-running `map` jobs restarted on "extra" nodes.
    - ◆ Result used from whichever job finishes first.

Guideline

Exploit "free" concurrency.

Consider PGO and WPO

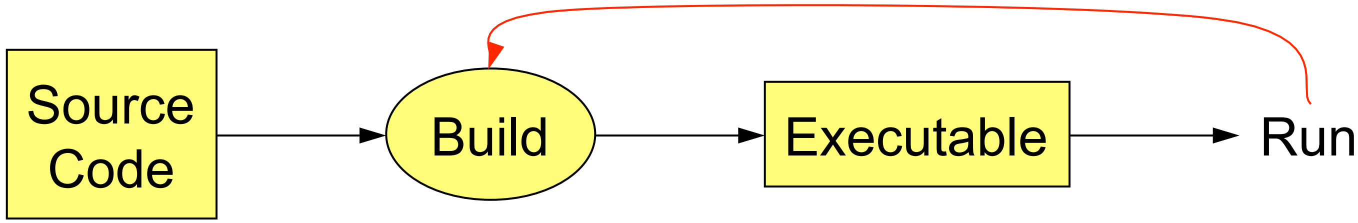
C++ build options:

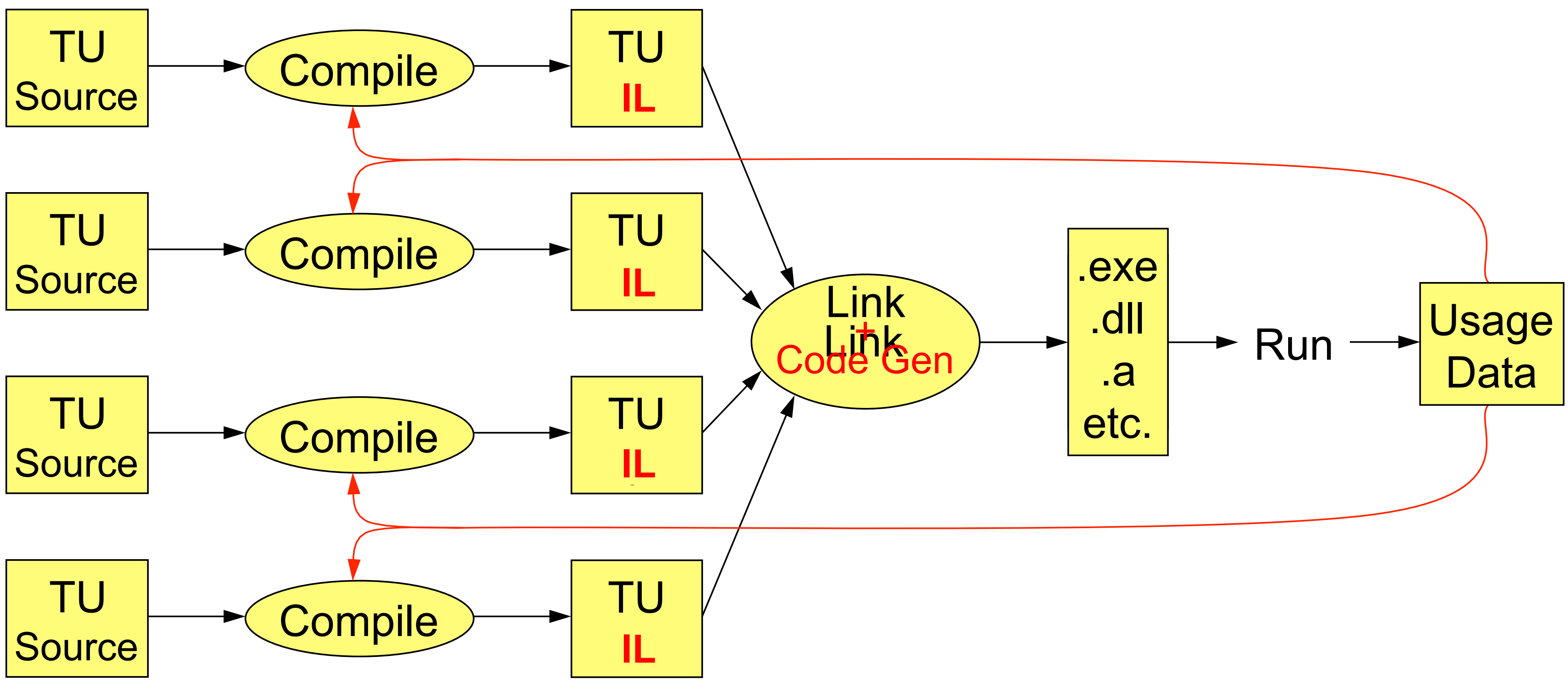
- PGO: Profile-Guided Optimization.
- WPO: Whole Program Optimization.

Commonly supported, e.g.:

- MSVC 2005+:
  - PGO: Compile with /GL, link with /LTCG:PGI or /LTCG:PGO.
  - WPO: Compile with /GL, link with /LTCG.
- gcc 4.1+:
  - PGO: Compile/link with -fprofile-generate or –fprofile-use.
- gcc 4.5+:
  - WPO: Compile/link with –flto.
    - –fwhole-program is orthogonal; it's for single-TU programs.
- Other vendors (e.g., Intel, HP, Oracle)

Build/Run Process without/with PGO





PGO

Approach:

1. Build for instrumentation.
   - E.g., /GL + /LTCG:PGI for MSVC, -fprofile-generate for gcc.
   - Generates instrumented .exe/.dll/.so (or parts thereof).
     - Result bigger/slower than from production build.
2. Run software on representative important use cases.
   - Usage data automatically recorded.
   - Poorly chosen use cases can "pessimize" later builds.
     - Exercise only key paths.
3. Build with PGO.
   - E.g., /GL + /LTCG:PGO for MSVC, -fprofile-use for gcc.
   - Generates optimized .exe/.dll/.so.
4. Compare performance to non-PGO build.

PGO-Enabled Optimizations

Improved basic block ordering:

- Reduce need for branching.
  - ➡ Fall through to most common cases in conditionals.
    - ◆ E.g. `if`/`else`s, `switch`es, loops.
  - ➡ Make better use of hardware branch prediction.
- Improve code locality ⇒ smaller working set.
  - ➡ Improved I-Cache usage ⇒ fewer misses.
  - ➡ Improved page usage ⇒ fewer page faults.

Better register allocation:

- Data on future variable usage ⇒ fewer or less costly spills.

PGO-Enabled Optimizations

Improved function layout:

- Like basic blocks, but for functions:
  - Put functions used together near one another.
    - Reduced I-Cache misses, page faults.
    - Reduced TLB misses (Carlton's *DDJ* article: up to 80%).

Function splitting:

- Move functions' infrequently-executed code to cold pages.
  - E.g., exception/error handlers.
- Improves code locality.
  - Reduces working set, I-Cache misses, page faults.
- Also known as *cold code separation*.

PGO-Enabled Optimizations

More effective inlining:

- Inline only along hot paths.
  - ➡ Excessive inlining bloats executables.
    - ◆ Can decrease I-Cache, TLB, and paging effectiveness.
- Partial inlining.
  - ➡ Inline only hot paths of a function.
- Speculatively inline virtual calls.
  - ➡ Inline most commonly called virtual at call site.
    - ◆ Runtime check ensures call is valid.

Function cloning:

- Context-dependent variations of a single uninlined function.
  - ➡ Different variations called in different contexts.
- Middle ground between inlined and non-inlined functions.

PGO-Enabled Optimizations

Context-dependent optimization strategies:

- Optimize hot paths for speed, cold paths for size.

Structure splitting and field reordering:

- Change object layouts for better D-Cache performance.
  - Violates C++ object model, so valid only when provable that program semantics not affected.

PGO in the Real World

Arjan van Leeuwen on Opera team's experience with PGO and gcc:

- "12.5% improvement on Futuremark Peacekeeper benchmark."

Online forum comments about PGO and gcc:

- "23.7% improvement over baseline. PGO is amazing!"
- "Sped up my program by almost 18%-20%. "
- "Giving ~20% improvement to recent builds."
- "Speed up was about the 20% mark. Executable is also smaller - interestingly by about 20%."

From Kang Su Gatlin's article about MSVC's PGO:

- "30%+ improvement on applications such as SQL Server."

From Lin Xu's blog entry about MSVC's PGO:

- "For part of the C++ intellisense engine in Visual Studio 2010, we saw ~25% better performance on some scenarios. For the compiler, we measured ~10% speedup in throughput."

PGO Caveats

Requires important representative use cases.

Tends to be most helpful for large, non-loop-bound applications.

- Hundreds to thousands of functions.
- Most time spent in branches, calls/returns.

Designed for use after source code freeze.

- By default, source code changes invalidate instrumentation data.

Resource-intensive during builds and instrumented runs.

- Instrumentation insertion/execution/analysis not cheap.

PGO Caveats

Changes meaning of "object file."

- Contents are IL, not native code.
  - Less portable/stable; IL format more volatile than assembly.
  - More difficult to distribute (e.g., as libraries).

Larger object files.

- IL bigger than native code.
- gcc 4.5 emits both IL *and* native code (in one file).
  - Allows IL-unaware tools to work with IL-based object files.

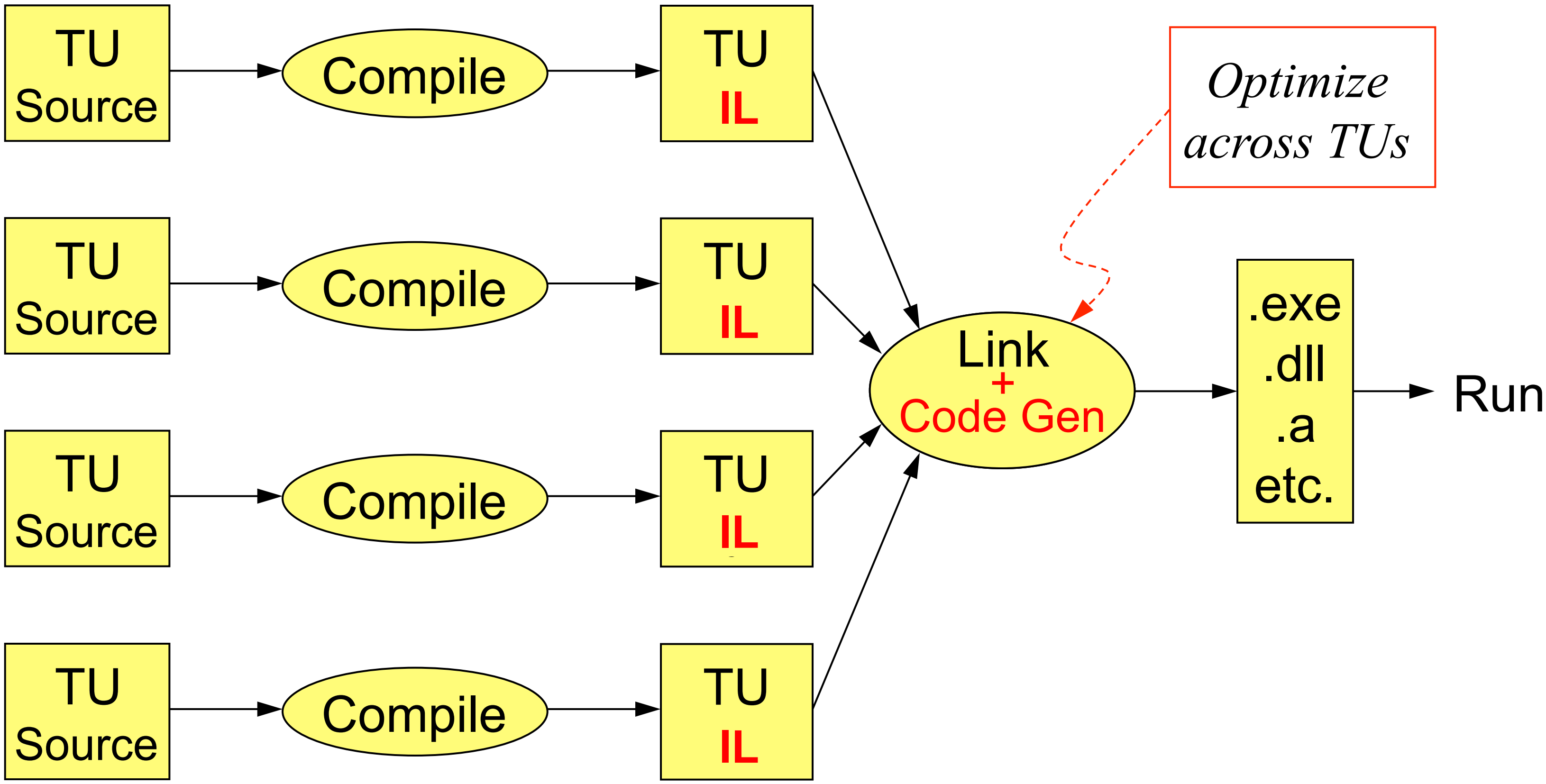
Complicates build configuration.

- Debug, production, and PGO builds.

May be incompatible with other compiler features.

- MSVC rejects attempt to use PGO with OpenMP.

Build/Run Process without/with WPO



WPO-Enabled Optimizations

Cross-module inlining:

- Functions defined in one TU can be inlined into other TUs.
  - C++ `inline` becomes less important.

Cross-module dataflow optimizations:

- Optimize use of registers across function boundaries.
- Comprehensive variable use information:
  - Eliminate fully unused globals.
  - Reduce/eliminate unnecessary stores/loads for globals.
  - Restrict memory potentially addressed by pointers.
    - Reduce unnecessary stores/loads.

Improved TLS handling:

- Use small offsets for most frequently used TLS data.
  - Shrinks code size, increases runtime speed.

WPO-Enabled Optimizations

Improved stack layout for doubles:

- Replace dynamic alignment with static alignment.
- x86 double access fastest when 8-byte aligned.
  - Win32 default alignment is 4-byte.

Custom function calling conventions:

- Rules about passing parameters/return values, stack cleanup, etc.
  - Push/pop order, register usage, etc.
  - Standard Windows options are cdecl, stdcall, fastcall.
- WPO sees all calls, hence can optimize with custom conventions.
  - Produces smaller/faster code.

WPO in the Real World

From online forum comment about MSVC WPO:

- "Reduces the size of the resulting binaries about 15%. "

From Matt Pietrek's article about MSVC WPO:

- "A boost of 3 to 5 percent is common for x86 programs."

From Jerry Goodwin's blog entry about MSVC's WPO:

- "You can typically speed up your code by about 3-4%."

From Lin Xu's blog entry about MSVC's PGO (but about WPO):

- "You might see (on x64) 10% faster code, and on x86, 7%."

MSVC WPO results not additive to PGO results.

- MSVC PGO builds on WPO.

WPO Caveats

Greater build-time demands:

- Compilation may be faster, but linking/codegen is slower.
- Memory use increases: all TUs analyzed simultaneously.

Similar object file implications as for PGO:

- Contents are IL, not native code.

May complicate debugging.

- From gcc 4.7 manual:
  - “[WPO] does not work well with generation of debugging information. Combining `-flto` with `-g` is currently experimental and expected to produce wrong results.”

Guideline

Consider PGO and WPO.

Overall Summary

- Treat speed as a correctness criterion.
- Optimize the system, not the program.
- Understand the importance of CPU caches.
- Use C++ effectively:
  - Take advantage of move semantics.
  - Avoid unnecessary object creation.
  - Consider custom heap management.

Overall Summary

- Use the STL effectively:
  - Use reserve to reduce reallocations in vector and string.
  - Use the swap trick or shrink_to_fit to reduce excess capacity in vectors, strings, and deques.
  - Prefer range member functions to single-element versions for sequence containers.
  - Prefer function objects to functions.
  - Consider sorted vectors for fast lookups.
  - Know your sorting options.
- Consider use of concurrent data structures.
- Consider use of parallel algorithms.
- Exploit “free” concurrency.
- Consider PGO and WPO.

The Big Picture (Reprise)

Architecture & Design	Coding	Tuning
▪ Speed as a correctness criterion ▪ Optimizing systems rather than programs ▪ CPU caches ▪ Concurrent data structures ▪ Parallel algorithms ▪ Exploiting "free" concurrency ▪ Sorted vectors	▪ Move semantics ▪ Avoiding unnecessary object creation ▪ `reserve` and `shrink_to_fit` ▪ Range member functions ▪ Function objects ▪ Sorting algorithms	▪ CPU caches ▪ Concurrent data structures ▪ Parallel algorithms ▪ Exploiting "free" concurrency ▪ Custom heap management ▪ Sorted vectors ▪ Sorting algorithms ▪ PGO and WPO

Further Information

My C++ stuff:

- *Effective C++, Third Edition: 55 Specific Ways to Improve Your Programs and Designs*, Scott Meyers, Addison-Wesley, 2005, ISBN 0-321-33487-6.
- *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*, Scott Meyers, Addison-Wesley, 2001, ISBN 0-201-74962-9.
- *More Effective C++: 35 New Ways to Improve Your Programs and Designs*, Scott Meyers, Addison-Wesley, 1996, ISBN 0-201-63371-X.

Tables of contents for these books are attached.

- DRM-free PDFs available at http://scottmeyers-ebooks.com/.

Further Information

TR1:

- *Scott Meyers' TR1 Information Page*, http://www.aristeia.com/EC3E/TR1_info.html.
  - ➡ Includes links to proposal documents.
- *The C++ Standard Library Extensions*, Pete Becker, Addison-Wesley, 2007, ISBN 0-321-41299-0.
  - ➡ A comprehensive reference for TR1.
- "The Technical Report on C++ Library Extensions," Matthew H. Austern, *Dr. Dobb's Journal*, June 2005.
- "The New C++ Not-So-Standard Library," Pete Becker, *C/C++ Users Journal*, June 2005.

Boost:

- *Boost C++ Libraries*, http://www.boost.org/.

Further Information

C++11:

- "Overview of the New C++ (C++11)," Scott Meyers, Artima Publishing, http://www.artima.com/shop/overview_of_the_new_cpp.
- "C++11," *Wikipedia*.
- *C++11 - the recently approved new ISO C++ standard*, Bjarne Stroustrup, http://www.research.att.com/~bs/C++0xFAQ.html.
- *The C++ Standard Library, Second Edition*, Nicolai M. Josuttis, Addison-Wesley, 2012.
- *Summary of C++11 Feature Availability in gcc and MSVC*, Scott Meyers, http://www.aristeia.com/C++11/C++11FeatureAvailability.htm.
  - ➡ Includes links to summaries for other compilers.
- "C++0x: Ausblick auf den neuen C++-Standard," Bernhard Merkle, *heise Developer*, 11 November 2008.
- *C++11: Der Leitfaden für Programmierer zum neuen Standard,* Rainer Grimm, Addison-Wesley, 2012.

Further Information

The importance of speed:

- "Performance is a Feature," Jeff Atwood, *Coding Horrors* (Blog), 20 June 2011.
- "Response Times: The 3 Important Limits," Jakob Nielsen, http://www.useit.com/papers/responsetime.html.
  - Excerpt from Nielsen's *Usability Engineering*, Morgan Kaufmann, 1993.
- "Scaling Google for Every User," Marissa Mayer, *Google Videos*, 23 June 2007.
- "New Google study on speed in search results," Greg Linden, *Geeking with Greg* (Blog), 29 June 2009.
  - Comments raise valid questions about data consistency across studies.
- "How Fast is Your Web Site?," Patrick Meenan, *ACM Queue*, February 2013.

Further Information

Building fast systems:

- "LPC: Booting Linux in five seconds," Don Marti, *LWN.net*, 22 September 2008.
- "Telemetry and What It Is Good for: Part 1: Nuts and Bolts," Taras Glek, *Taras' Blog*, 25 July 2012.
- "Telemetry and What It Is Good for: Part 2: Telemetry Achievements," Taras Glek, *Taras' Blog*, 25 July 2012.
- "Thinking Clearly about Performance," Cary Millsap, *ACM Queue*, September 2010.
  - Focuses on database-based systems.
- *ACM Queue*, February 2006.
  - Enhancing performance in large, network- based systems.
- *Let's make the web faster*, Google code, http://code.google.com/speed/.
  - Portal for information on building fast web-based applications.

Further Information

More on building fast systems:

- "Parsing XML at the Speed of Light," Arseny Kapoulkine, in *The Performance of Open Source Applications*, 2013. Available at http://tinyurl.com/mv3rpck.
  - ➡ One chapter of a 12-chapter book.

Buffers and Latency:

- "Bufferbloat: Dark Buffers in the Internet," Jim Gettys and Kathleen Nichols, *Communications of the ACM*, January 2012.
  - ➡ How excessive buffering increases TCP latency.

Further Information

Steve Souders' work:

- *High Performance Web Sites*, Steve Souders, O'Reilly, 2007, ISBN 0-596-5230-9.
- "High Performance Web Sites and YSlow," Steve Souders, *Google Tech Talks*, 13 November 2007.
- *Even Faster Web Sites*, Steve Souders, O'Reilly, 2009, ISBN 978-0-596-52230-8.
- "High Performance Web Sites," Steve Souders, *Communications of the ACM*, December 2008.
  - ➡ Summarizes *High Performance Web Sites* and parts of *Even Faster Web Sites*.

Further Information

CPU caches:

- *What Every Programmer Should Know About Memory*, Ulrich Drepper, 21 November 2007, http://people.redhat.com/drepper/cpumemory.pdf.
- "CPU cache," *Wikipedia*.
- "Gallery of Processor Cache Effects," Igor Ostrovsky, *Igor Ostrovsky Blogging* (Blog), 19 January 2010.
- "Writing Faster Managed Code: Know What Things Cost," Jan Gray, *MSDN*, June 2003.
  - Relevant section title is "Of Cache Misses, Page Faults, and Computer Architecture"
- "Optimizing for instruction caches," Amir Kleen et al., *EE Times*, 29 Oct. 2007 (part 1), 5 Nov. 2007 (part 2), 12 Nov. 2007 (part 3).
- "Memory is not free (more on Vista performance)," Sergey Solyanik, *1-800-Magic* (Blog), 9 December 2007.
  - Experience report about optimizing use of I-cache.

Further Information

CPU caches:

- "Martin Thompson on Mechanical Sympathy," *Software Engineering Radio*, 19 February 2014.
- "Eliminate False Sharing," Herb Sutter, *DrDobbs.com*, 14 May 2009.
- "False Sharing is no fun," Joe Duffy, *Generalities & Details: Adventures in the High-tech Underbelly* (Blog), 19 October 2009.
- "Real-World Concurrency," Bryan Cantrill and Jeff Bonwick, *ACM Queue*, September 2008.
  - Discusses false sharing.
- "Native Code Performance and Memory: The Elephant in the CPU," Eric Brumer, *Channel 9*, 28 June 2013.
  - Video of a *Build 2013* presentation.
- "07-26-10 – Virtual Functions," Charles Bloom, *cbloom rants* (Blog), 26 July 2010.
  - Note ryg's comment about per-type operation batching.

Further Information

CPU caches and data structures (e.g., work-stealing runtimes):

- "Data Structures in the Multicore Age," Nir Shavit, *Communications of the ACM*, March 2011.
- "Multicore Desktop Programming with Intel Threading Building Blocks," Wooyoung Kim and Michael Voss, *IEEE Software*, January/February 2011.
  - ➨ Discusses TBB's work-stealing runtime.
- "Efficient Workstealing for Multicore Event-Driven Systems," Fabien Gaud *et al.*, International Conference on Distributed Computing Systems 2010.
  - ➨ Discusses refinements to cache analysis for work-stealing.

Further Information

Rvalue references, move semantics, perfect forwarding:

- *C++ Rvalue References Explained*, Thomas Becker, June 2009, http://thbecker.net/articles/rvalue_references/section_01.html.
  - ➡ Good explanations of std::move/std::forward implementations.
- "Howard's STL / Move Semantics Benchmark," Howard Hinnant, *C++Next*, 13 October 2010.
  - ➡ Move-based speedup for std::vector<std::set<int>>.
- "Making Your Next Move," Dave Abrahams, *C++Next*, 17 September 2009.
- "Your Next Assignment…," Dave Abrahams, *C++Next*, 28 September 2009.
  - ➡ Correctness and performance issues for move operator=s.
- "Onward, Forward!," Dave Abrahams, *C++Next*, 7 Dec. 2009.
  - ➡ Discusses perfect forwarding.
- *Boost.Move*, Ion Gaztanaga, Boost Documentation, boost.org.

Further Information

Optimizations related to std::shared_ptr:

- "STL11: Magic && Secrets," Stephan T. Lavavej, Presentation at *GoingNative 2012*, 2 February 2012, http://tinyurl.com/7dmqfn2.
  - Describes the "We know where you live" optimization.

Further Information

Custom memory management:

- "Memory Management & Embedded Databases," Andrei Gorine and Konstantin Knizhnik, *Dr. Dobbs Journal*, December 2005.
- "Reconsidering Custom Memory Allocation," Emery Berger *et al.*, *Proceedings of OOPSLA 2002*.
- "A Memory Allocator," Doug Lea, http://gee.cs.oswego.edu/dl/html/malloc.html.
- *Modern C++ Design*, Andrei Alexandrescu, Addison-Wesley, 2001, ISBN 0-201-70431-5, Chapter 4.
- "Boost Pool Library," Stephen Cleary, http://www.boost.org/libs/pool/doc/index.html.
- "Policy-Based Memory Allocation," Andrei Alexandrescu and Emery Berger, *C/C++ Users Journal*, December 2005.
- "Improving Performance with Custom Pool Allocators for STL," Anthony Aue, *Dr. Dobbs Journal*, September 2005.

Further Information

Sorted vectors:

- "Why You Shouldn't Use set (and What You Should Use Instead)," Matt Austern, *C++ Report*, April 2000.
- Boost.Container's flat_ containers.
  - Wrappers for a sorted vectors with interfaces modeled on C++11's set/map/multiset/multimap.

Further Information

Sorting:

- "Sorting in the Standard Library," Matt Austern, August 2001, *C/C++ User's Journal*'s Experts Forum.
- "Top Ten Percent," Thomas Guest, *Word Aligned* (Blog), February 2008.
  - Compares partial_sort and nth_element + sort.
- "Why Stable Sorting is Important," Andrew Koenig, *Dr. Dobbs*, 15 July 2011.
- "Another Use For Stable Sorting," Andrew Koenig, *Dr. Dobbs*, 20 July 2011.
  - Explains relationship between sorting and partitioning.

Further Information

Concurrent data structures:

- "Definitions of Non-blocking, Lock-free, and Wait-free," Anthony Williams, *Just Software Solutions*, 7 September 2010.
- "Use Cases for Concurrent Data Structures," Dmitriy Vyukov, comp.programming.threads, 8 September 2010, http://tinyurl.com/27xjoxq.
  - ➡ Very nice overview of data structure option for MT systems.
- "Single-Producer/Single-Consumer Queue," Dmitriy Vyukov, *Intel Knowledge Base*, 27 February 2009, http://tinyurl.com/2ewfdjn.
  - ➡ Non-TBB/PPL DS optimized for 1 producer + 1 consumer.
- "Need help with lock free memory allocator with lock free data structure," thread in comp.programming.threads, 6-8 July 2010, http://tinyurl.com/23zeh6q.
  - ➡ Much discussion is about use of concurrent queues.
- "Common Pitfalls in Writing Lock-Free Algorithms," David Stolp, *memsql developer blog*, 27 March 2013.

Further Information

TBB/PPL concurrent data structures:

- *Threading Building Blocks Home Page*, http://www.threadingbuildingblocks.org/.
- "Parallel Containers and Objects," *msdn*, http://msdn.microsoft.com/en-us/library/dd504906.aspx.
- "TBB containers vs. STL. Performance in the multi-core age.," Andrey Marochko, *Intel Software Blogs*, 20 October 2008.
- "The concurrent_queue Container in VS2010," dmccrady, *Parallel Programming in Native Code*, 23 November 2009.
- "Traversing concurrent_hash_map concurrently," Andrey Marochko, *Intel Software Blogs*, 14 May 2010.
- "Testing simple concurrent containers," Valery Grebnev, *The Code Project*, 25 October 2009.
  - ➡ Times STL containers + various locking schemes vs. TBB's concurrent_hash_map.

Further Information

TBB/PPL data structure implementations:

- "concurrent_vector and concurrent_queue explained," Raghu Simha, *Parallel Programming in Native Code*, 15 January 2010.
- "tbb::concurrent_vector: secrets of memory organization," Andrey Marochko, *Intel Software Blogs*, 24 July 2008.

Further Information

Concurrent data structures beyond TBB/PPL:

- "Boost.Lockfree," Tim Blechmann.
- Liblfds, http://www.liblfds.org/.
  - ➡ Offers C API, stores void* pointers.
- "Some notes on lock-free and wait-free algorithms," Ross Bencina, 13 December 2009, http://tinyurl.com/29qg43p.
  - ➡ Many links in section "Existing source code and libraries"
- "Data Structures in the Multicore Age," Nir Shavit, *Communications of the ACM*, March 2011.
- "Introduction to Lock-free Programming with C++ and Qt," Olivier Goffart, *woboq*, 13 December 2011.
  - ➡ Includes good explanation of the ABA problem.

Further Information

Parallel algorithms:

- "Concurency::parallel_for and Concurrency::parallel_for_each," Marko Radmilac, *Parallel Programming in Native Code*, 18 November 2009.
  - Nice overview, including comparison to OpenMP.
- "Visual C++ 2010 and the Parallel Patterns Library," Kenny Kerr, *MSDN Magazine*, February 2009.
  - Includes information on PPL vs. OpenMP.
- "Parallel Algorithms," *msdn*, http://msdn.microsoft.com/en-us/library/dd470426.aspx.
- "Exception handling in TBB 2.2 - Getting ready for C++0x," Andrey Marochko, *Intel Software Blogs*, 18 August 2009.
- "Concurrency and exceptions," Joe Duffy, *Generalities & Details: Adventures in the High-tech Underbelly*, 23 June 2009.

Further Information

More on parallel algorithms:

- "How to pick your parallel sort," Vinod Subramanian, *Parallel Programming in Native Code* (Blog), 25 January 2011.
  - Compares PPL's parallel_sort, parallel_buffered_sort, parallel_radixsort.
- "Sorting in PPL," Vinod Subramanian, *Parallel Programming in Native Code* (Blog), 14 January 2011.
  - More on choosing among PPL's sorting algorithms.

Further Information

OpenMP:

- *OpenMP Home Page*, http://openmp.org/wp/.
- "Reap the Benefits of Multithreading without All the Work," Kang Su Gatlin and Pete Isensee, *MSDN Magazine*, October 2005.
  - ➡ Very nice introductory/overview article.
- "C++ Parallelization Libraries: OpenMP vs. Thread Building Blocks," *stackoverflow*, initiated 5 March 2009, http://tinyurl.com/2ezlfcs.
- "32 OpenMP Traps for C++ Developers," Alexey Kolosov *et al.*, *Viva64.com*, 20 November 2009.
  - ➡ Nice compendium of common usage errors.

Further Information

Concurrency and scalability:

- "Amdahl's Law," *Wikipedia*.
- "Photoshop Scalability: Keeping It Simple," Clem Cole and Russell Williams, *acm queue*, 9 September 2010.
  - ➡ Contains quote about "Amdahl's wall."
- "Parallel Scalability Isn't Child's Play, Part 2: Amdahl's Law vs. Gunther's Law," Mark B. Friedman, *MSDN Developer Division Performance Engineering blog*, 29 Apr 2009.
  - ➡ Discusses why Amdahl's law is optimistic in practice.
- "Neil J. Gunther," *Wikipedia*.
  - ➡ Article looks like Gunther (or a Gunther fan) wrote it.
  - ➡ Overview of Universal Scalability Law (within *Wikipedia* article) is at http://tinyurl.com/3chlr3n.
- "Real-World Concurrency," Bryan Cantrill and Jeff Bonwick, *ACM Queue*, September 2008.

Further Information

C++11 concurrency:

- *C++ Concurrency in Action*, Anthony Williams, Manning, 2012, ISBN 1-933988-77-0, http://www.manning.com/williams/.
  - Chapter 9 shows an interruptible_thread implementation.
- "MT Design Question," comp.lang.c++, initiated 24 August 2010, http://tinyurl.com/28ns2nw.

Hadoop:

- *Hadoop Home Page*, http://hadoop.apache.org/.
- *Yahoo! Hadoop Tutorial*, "Module 4: MapReduce," http://developer.yahoo.com/hadoop/tutorial/module4.html.

Mechanical Turk:

- "Amazon Mechanical Turk," *Wikipedia,* http://en.wikipedia.org/wiki/Amazon_Mechanical_Turk.

Further Information

Profile-guided optimization (PGO):

- "Profile-Guided Optimizations," Gary Carleton, Knud Kirkegaard, and David Sehr, *Dr. Dobb's Journal*, May 1998.
  - ➨ Still a very nice overview.
- "The Future of Code Coverage Tools," Mohammad Haghighat and David Sehr, *StickyMinds.com*.
- "Build faster and high performing native applications using PGO," Ankit Asthana, *Visual C++ Team Blog*, 4 April 2013.
- "/GL and PGO,"Lin Xu, *Visual C++ Team Blog*, 1 December 2009.
- "POGO," Lawrence Joel, *Visual C++ Team Blog*, 12 November 2008.
- "Profile Guided Optimizations," Shachar Shemesh, *Scribd*, Uploaded 3 June 2009, http://tinyurl.com/2u6lvln.
  - ➨ Much code optimization info, including PGO for gcc.

Further Information

More on PGO:

- "Cache Aware Data Layout Reorganization Optimization in GCC," Mostafa Hagog and Caroline Tice, *Proceedings of the GCC Developers' Summit*, June 2005.
  - ➡ Using PGO to change DS layouts to improve D$ performance.
- "Profile driven optimisations in GCC," Jan Hubička, *Proceedings of the GCC Developers' Summit*, June 2005.
- "GoingNative 12: C++ at Build 2012, Inside Profile Guided Optimization," Channel 9, 28 November 2012.
  - ➡ Video interview about PGO support in MS Visual C++.
- "Profile Guided Optimization (PGO)—Under the Hood," Ankit Asthana, *Visual C++ Team Blog*, 27 May 2013.
- "The *New Performance Optimization Tool* for Visual C++ applications," Ankit Asthana, *Visual C++ Team Blog*, 22 October 2013.

Further Information

Whole-program optimization (WPO):

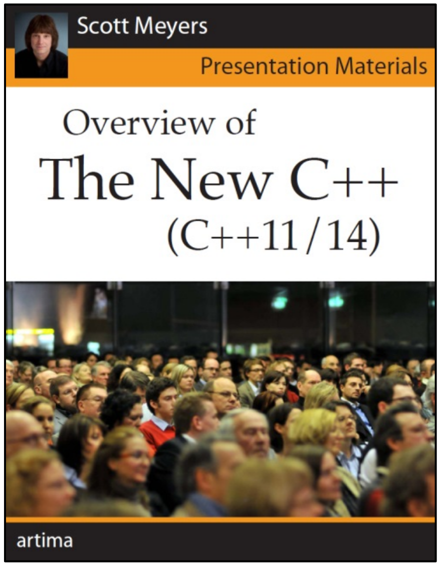
- "Optimizing real-world applications with GCC Link Time Optimization," Taras Glek and Jan Hubička, *Proceedings of the GCC Developers' Summit*, October 2010.
- "Whole Program Optimization with Visual C++ .NET," Brandon Bray, *The Code Project*, 10 December 2001 .
- "Link-Time Code Generation," Matt Pietrek, *MSDN Magazine*, May 2002.
- "Quick Tips On Using Whole Program Optimization," Jerry Goodwin, *Visual C++ Team Blog*, 24 February 2009.
- "Introducing '/Gw' Compiler Switch," Ankit Asthana, *Visual C++ Team Blog*, 11 September 2013.
  - ➡ Enables elimination of unused global data.

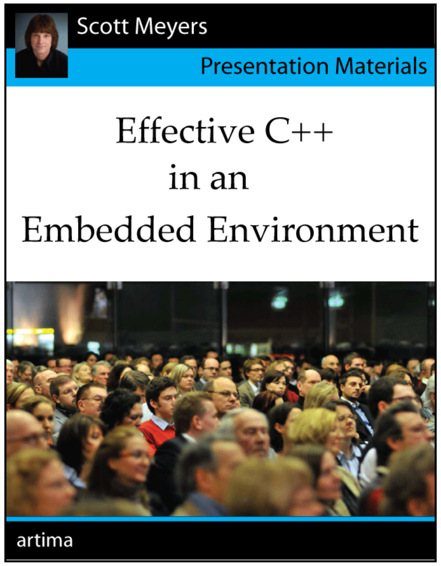
Licensing Information

Scott Meyers licenses materials for this and other training courses for commercial or personal use. Details:

- Commercial use: http://aristeia.com/Licensing/licensing.html
- Personal use: http://aristeia.com/Licensing/personalUse.html

Courses currently available for personal use include:





About Scott Meyers

Scott is a trainer and consultant on the design and implementation of C++ software systems. His web site,

http://www.aristeia.com/

provides information on:

- Training and consulting services
- Books, articles, other publications
- Upcoming presentations
- Professional activities blog

Contents

Reprinted from *Effective C++, Third Edition: 55 Specific Ways to Improve Your Programs and Designs*, by Scott Meyers, Addison-Wesley Publishing Company, 2005.

Reprinted from *Effective C++, Third Edition: 55 Specific Ways to Improve Your Programs and Designs*, by Scott Meyers, Addison-Wesley Publishing Company, 2005.

Contents

Reprinted from *More Effective C++: 35 New Ways to Improve Your Programs and Designs*, by Scott Meyers, Addison-Wesley Publishing Company, 1996.

Contents

Reprinted from *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*, by Scott Meyers, Addison-Wesley Publishing Company, 2001.