

# Creating safe multi-threaded applications in C++11

Jos van Eijndhoven  
jos@vectorfabrics.com

**ACCU conference**

Bristol, UK  
April 2014



# Vector Fabrics' activities

## PRODUCTS

Pareon makes your C/C++ code run faster



[PAREON OVERVIEW >](#)

Tool development and licensing

## CONSULTANCY

Software optimization



[CHECK OUT OUR EXPERTISE >](#)

Consultancy services

## TRAINING

Multicore programming training

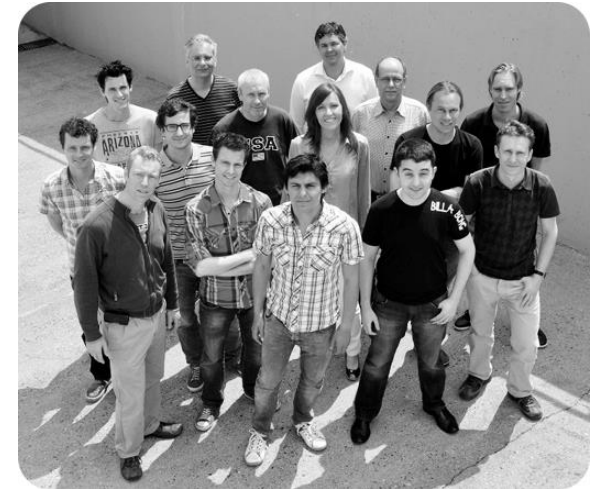


[SEE OUR TRAINING PROGRAM >](#)

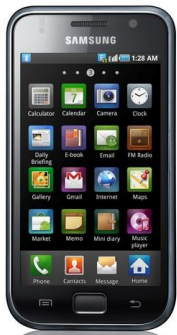
Training in-house and on-site

# Vector Fabrics – the company

- Founded February 2007 in Eindhoven, the Netherlands
- Currently 15 FTE: 6 PhD, 7 MSc
- Recognition
  - “Hot Startup” in EE Times Silicon 60 list, since 2011
  - Selected by Gartner as “Cool vendor in Embedded Systems & Software” 2013
  - Global Semiconductors Alliance award, March 2013



# You all see the proliferation of multi-core



Galaxy S (2010)  
1 processor



Galaxy S2 (2011)  
2 cores



Galaxy S3 (2012)  
4 cores



Galaxy S4 (2013)  
8 cores

# Multi-core systems drive programmer awareness

## **Homogeneous multi-core, hardware cache-coherency, one shared OS kernel:**

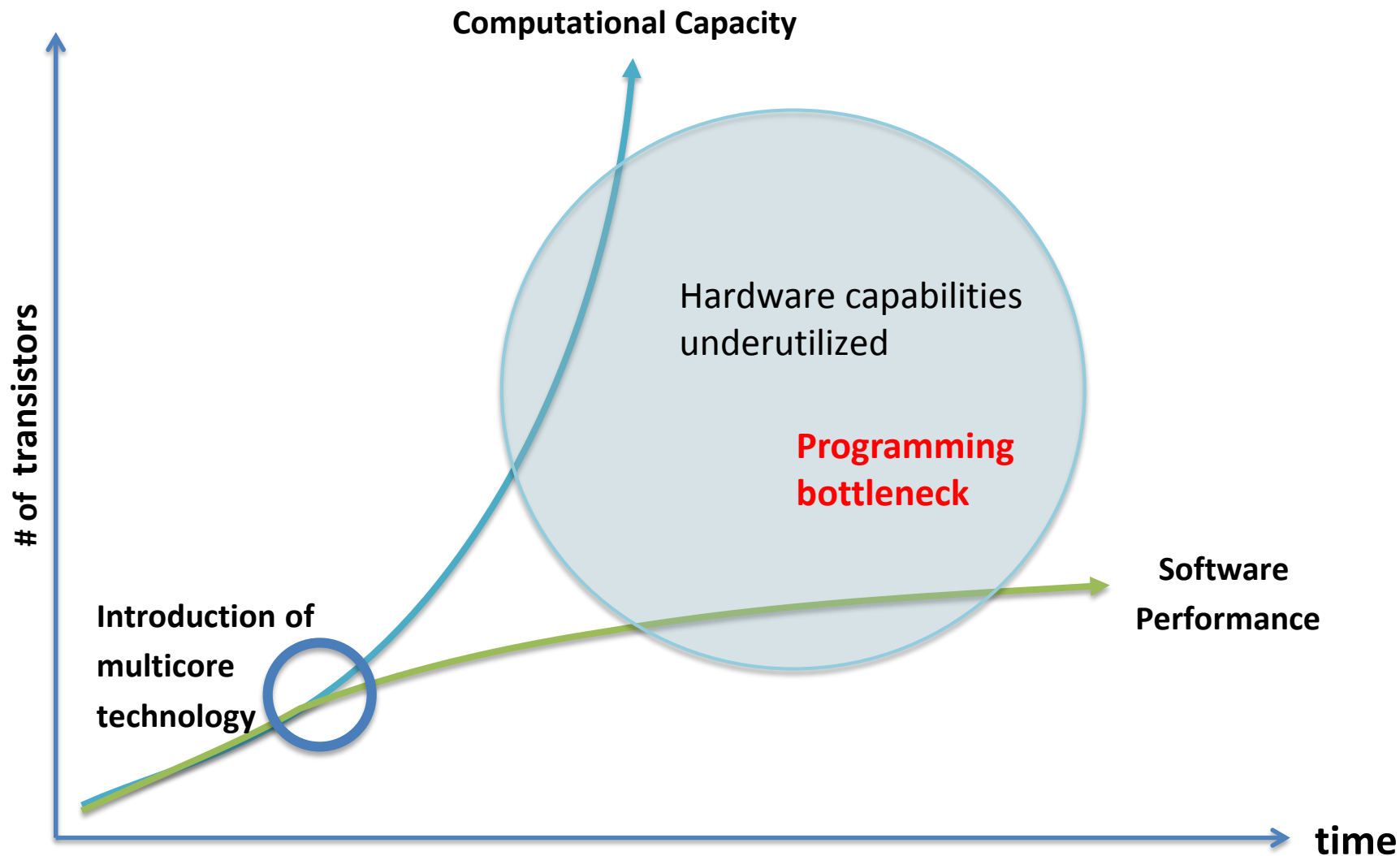
Industry proven successful combination, long history

- IBM 3084: 4-cpu mainframe (1982)
- Silicon Graphics 'Origin': 1024-cpu supercomputer (2000)
- Intel Pentium D: dual core single chip (2005)
- Sun Niagara: 8-core single chip (2005)
- ARM Cortex-A9 dual-core on on Nvidia tegra-2 chip (2011)

And more recent on the server side:

- Intel Xeon Phi: 60-core single chip (2012)
- IBM Blue Gene/Q: 1.6M cores, 1.6PB memory (2012)

# Moore's law versus Amdahl's law





# Creating parallel programs is hard...



Herb Sutter, ISO C++ standards committee, Microsoft:

“Everybody who learns concurrency thinks they understand it, ends up finding mysterious races they thought weren't possible, and discovers that they didn't actually understand it yet after all”

Edward A. Lee, EECS professor at U.C. Berkeley:

“Although threads seem to be a small step from sequential computation, in fact, they represent a huge step. They discard the most essential and appealing properties of sequential computation: understandability, predictability, and determinism.”



# Problems, anyone?

Nissan recalls some Infiniti Q50 sedans with steer-by-wire software glitch

<http://www.autonews.com/article/20131216/RETAIL05/131219890/nissan-recalls-some-infiniti-q50-sedans-with-steer-by-wire-software#>

2013 Ram 1500 Recalled For Stability Control Software Glitch

[http://www.thecarconnection.com/news/1085613\\_2013-ram-1500-recalled-for-stability-control-software-glitch](http://www.thecarconnection.com/news/1085613_2013-ram-1500-recalled-for-stability-control-software-glitch)

Delhi-bound AI Dreamliner lands in Kuala Lumpur due to software glitch

<http://ibnlive.in.com/news/delhibound-ai-dreamliner-lands-in-kuala-lumpur-due-to-software-glitch/450184-2.html>

Toyota is recalling 1.9 million of its top-selling Prius hybrid cars because of a software fault that may cause the vehicle to slow down suddenly

<http://www.bbc.com/news/business-26148711>

Recall Roundup: Software Glitches Force Several Recalls

<http://autos.jdpower.com/content/blog-post/AuY6uUi/recall-roundup-software-glitches-force-several-recalls.htm>

Bug Sends Space Probe 'Spinning Out of Control,' NASA Says

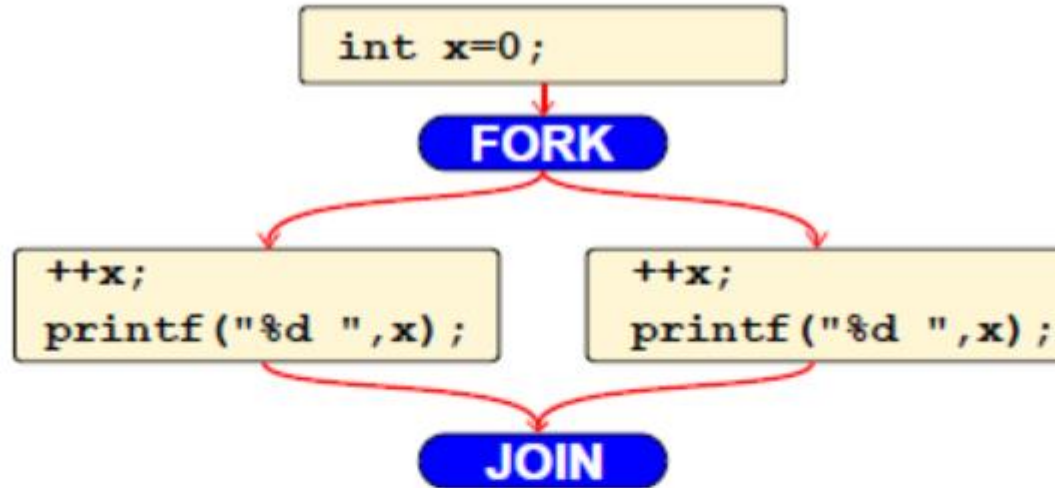
<http://www.weather.com/news/science/space/deep-impact-spacecraft-20130910>

Volvo recalls 2014 models to correct software glitch

<http://uk.reuters.com/article/2013/09/05/uk-autos-volvo-recall-idUKBRE9840U320130905>



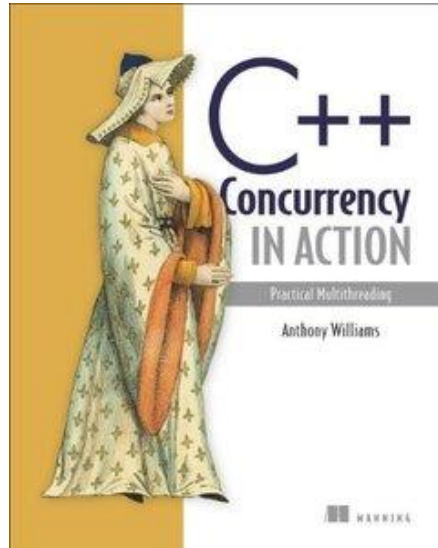
# Multi-threading: non-deterministic behavior



**Quiz:** Without further synchronization, which are valid print-outs according to C (and Java) language semantics?

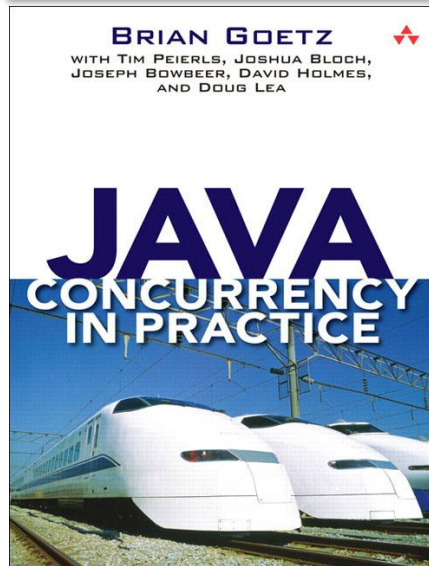
- 1 1
- 1 2
- 2 1
- 2 2

# Learning raises the awareness of complexity

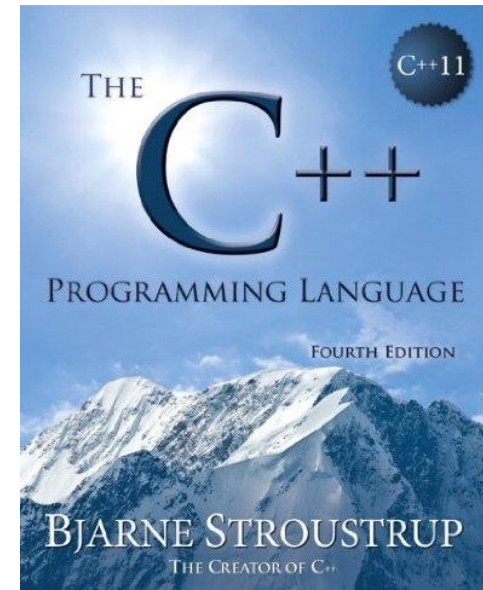


- Provides good insight in C++ concurrency
- C++11 standardizes concurrency primitives
- Warns for *many many* subtle problems

- The authoritative description (4th edition)
- Apparently requires 1300+ pages...



- Safe concurrency by defensive design
- Shows that Java shares many concurrency issues with C++



# HOWTO: Parallelization of sequential C/C++

- Analyze behavior of sequential program:  
Establishes **functional reference**, **deterministic** behavior
- Look for loops that provide good opportunity:
  - Contain a significant amount of all work
  - Loop-carried dependencies seem manageable...
- Make an inventory of loop-carried dependencies (group by object, or by class type)
- Do a 'what if resolved' performance estimate...
- ...maybe for different target architectures
- Verify the correctness of your concurrent implementation

# PAREON: performance analysis

**View on call tree with relative workload**

Name	Coverage	Delay
sgtf2_	72	71.93
xerbla_		0.00
slamch_	100	0.00
<b>Loop_32</b>	<b>81</b>	<b>71.92</b>
sgetrs_	57	3.67
printf		0.10
Loop_238	100	0.02
printf		0.05
printf		0.03

**Loop\_32 total loop\_carried transfer rate: 83.7 Mi transfers/s**  
0 streaming pattern clusters (0.0 transfers/s); 1 data dependency clusters (83.7 Mi transfers/s);  
2 compute dependencies (50.0 Ki transfers/s); 6 anti- and output dependency clusters

Property	Value
Loop	Loop_32 (sgtf2_)
#invocations	1
#iterations / invocation	100 (iteration histogram)
Invocation time	3.9 ms (39.0 us / iteration) (time hist)
Memory penalty	94.2 us (2.4 %)
Data cache penalties	92.5 us (2.4 %)
Data cache misses	
Level 1	11733 (1.1 %)
Level 2	8 (0.0 %)
DRAM traffic	512 B (128 KiB/s)
Load count	717340
Store count	352894
Mapped to Instance	Cpu0.[0]
Source location	sgtf2.c:141-185
Line coverage	80.8 %
Uncovered lines	

**Loop-carried dependencies hinder parallel execution of loop iterations**

**Other performance statistics: Iteration counts, cache penalties**

**Dynamic Help**  
**2D-Profile**  
The 2D-Profile presents a birds-eye view on the function and loop invocations of your

# PAREON: data dependency analysis

The image displays the PAREON software interface, which is used for data dependency analysis. It is divided into several panels:

- Profile / Partitions:** A table showing the coverage and delay for various code partitions. 

Name	Coverage	Delay
sscal_	79	0.88
Loop_56	0	0.00
sger_	61	66.43
xerbla_		0.00
<b>Loop_51</b>	<b>92</b>	<b>66.33</b>
Loop_52	100	63.79
Loop_53	100	0.80
Loop_54		0.00
sgetrs_	57	3.67
printf		0.10
- Properties / My changes:** A table showing the configuration for the selected partition (Loop\_51). 

Property	Value
Compute dependency 51.78	
Resolve for data partitioning	Resolve as induction
Producers	
Operation (pointer arithmetic)	Loop_51 (sger_)
Location	sger_custom.c:128
Consumers	
Operation (pointer arithmetic)	Loop_51 (sger_)
Location	sger_custom.c:128
Operation (pointer arithmetic)	Loop_52 (sger_)
Location	sger_custom.c:137
Operation (pointer arithmetic)	Loop_52 (sger_)
Location	sger_custom.c:138
Operation (pointer arithmetic)	Loop_53 (sger_)
Location	sger_custom.c:143
#Loop-carried transfers	1.3 Mi transfers/s
Symbols	a
Loop carried	yes
- 2D-Profile:** A visualization of the code's execution flow, showing nested loops and their dependencies. A callout box provides detailed information for Loop\_51: 

Loop\_51 total loop\_carried transfer rate: 2.6 Mi transfers/s  
0 streaming pattern clusters (0.0 transfers/s); 0 data dependency clusters (0.0 transfers/s);  
2 compute dependencies (2.6 Mi transfers/s); 0 anti- and output dependency clusters

Detailed info on loop-carried dependencies:  
producer & consumer source locations,  
allocation location, symbol name



# PAREON: Schedule data dependencies

The screenshot displays the PAREON interface with several panels:

- Partitioning candidates - Loop\_38**: Shows CPU data partitioning for 4 threads. Global speedup is 2.3, global overhead is 6%. A table lists the partitioning candidate:

Invocation	Speedup	Overhead	Streams
Loop_38	3.9	1 %	1

- Properties**: Shows details for Loop\_38 (sgtf2\_). Iteration count is 150. Iteration statistics include: Computation time (85.3 us, 92.7%), Memory penalty (6.8 us, 7.3%), Load count (15770), Store count (7802), and Instruction count (104658).
- 2D-Profile**: Shows a schedule overview with a yellow box highlighting a region. Below it, a detailed schedule execution graph shows iterations #68 through #83. The graph illustrates data dependencies between iterations across different threads, with a blue callout pointing to the graph.

Estimate multi-thread fork/join overhead

Obtain a *preview* on a potential parallelization assume synchronization on complex dependencies

# PAREON: Loop statistics

The screenshot displays the PAREON software interface. On the left, the 'Partitions' tab shows 'Partitioning candidates - Loop\_51' with a 'Number of threads' set to 2. Below this, a table lists partitioning candidates:

Candidate	Speedup	Overhead
Loop_51	1.2	40%

The 'Properties' tab for 'Loop\_51 (sger\_)' shows the following details:

- Loop: Loop\_51 (sger\_)
- #invocations: 99
- #iterations / invocation: 50 (iteration histogram)
- Average invocation time: 36.4 us (727 ns / iteration) (time histogram)
- Memory penalty: 819 ns (2.3 %)
- Mapped to Instance: Cpu0.[0]
- Source location: sger\_custom.c:128-144

At the bottom left, a histogram shows the distribution of invocation times, with the x-axis labeled 'Invocation time' and the y-axis labeled 'Occurrence (count)'. The x-axis values range from 1.2 us to 141.2 us.

The main window, titled '2D-Profile', shows a 2D execution profile for 'sgetf2.c' and 'sger\_custom.c'. A blue callout box points to the profile with the text: 'Histogram on execution time per iteration: wide variation is not nice...'. The profile shows a series of blue bars representing execution time, with a 'Loop\_51' section highlighted. A 'SPEED-UP' indicator in the top right corner shows a value of 1.0.



# WHAT IF application is partially parallelized?

- Some parallelization was done before using Pareon
- Or some parallelization was done on Pareon's advice, but we want to look for more opportunities...

Tracing load-store dependencies becomes harder!

- Obtaining the inter-thread load-store dependencies is OK, but:
- Actual load-store interleaving over time (mutual ordering) is schedule-dependent (is non-deterministic)
- How to decide whether observed inter-thread data exchange is good or wrong?

**C++11 comes to rescue! 😊**

# Pre-11 C/C++ constructs for threading

Three basic primitives, and some OS-level functionality

- Volatile variable declarations:  
force compiler load/store generation, limit compiler re-orderings
- Memory fence operations:  
force load/store ordering at runtime in the memory system
- Atomic operations:  
indivisible read-modify-write (increment, test-and-set)
- Higher-level abstractions (semaphores, condition variables) that include OS and kernel support → thread sleep and wakeup

Only 'volatile' is standardized in C/C++. Originally designed for I/O to hardware.

Posix thread library in 1995, fences/atomics are compiler specific intrinsics

# Pre-11 C/C++ constructs for threading

Creation of multi-threaded programs:

- The C/C++ compiler performs strong optimizations that are only valid in single-threaded execution mode
- ‘volatiles’ and ‘fences’ are **required**, often **forgotten**, **clutter** your program, **degrade** performance beyond need.

This **forgotten** leads to rarely occurring bugs, which are not reproducible.

And: programs that seemed correct on X86, appear buggy on ARM

# 1995-2011 C/C++ constructs for threading

Three basic primitives, and some OS-level functionality

- Volatile variable declarations:  
force compiler load/store generation, limit compiler re-orderings
- Memory fence operations:  
force load/store ordering at runtime in the memory system
- Atomic operations:  
indivisible read-modify-write (increment, test-and-set)
- Higher-level abstractions (semaphores, condition variables) that  
include OS and kernel support → threads sleep and wakeup

Get rid of all of this 15 years of programming practice

**bold** move by the C++11 committee!

# C11/C++11 parallel programming

Creation of multi-threaded programs:

- The C/C++ compiler will **always** assume multi-threaded access to variables with global scope. This inhibits some optimizations. (C++11 has no 'volatile' to denote inter-thread data exchange)
- **Atomic** operations are overloaded with **memory fence** behaviors. These are the basic building blocks for inter-thread synchronization.

If the programmer creates **SC-DRF** programs, then the system ensures correct (deterministic) behavior!

Sequentially Consistent Data Race Free

**Finally:** multi-threaded behavior is properly specified for C/C++ !!

# Satisfy *Data Race Free*

**Sufficient** condition to satisfy 'Data Race Free':

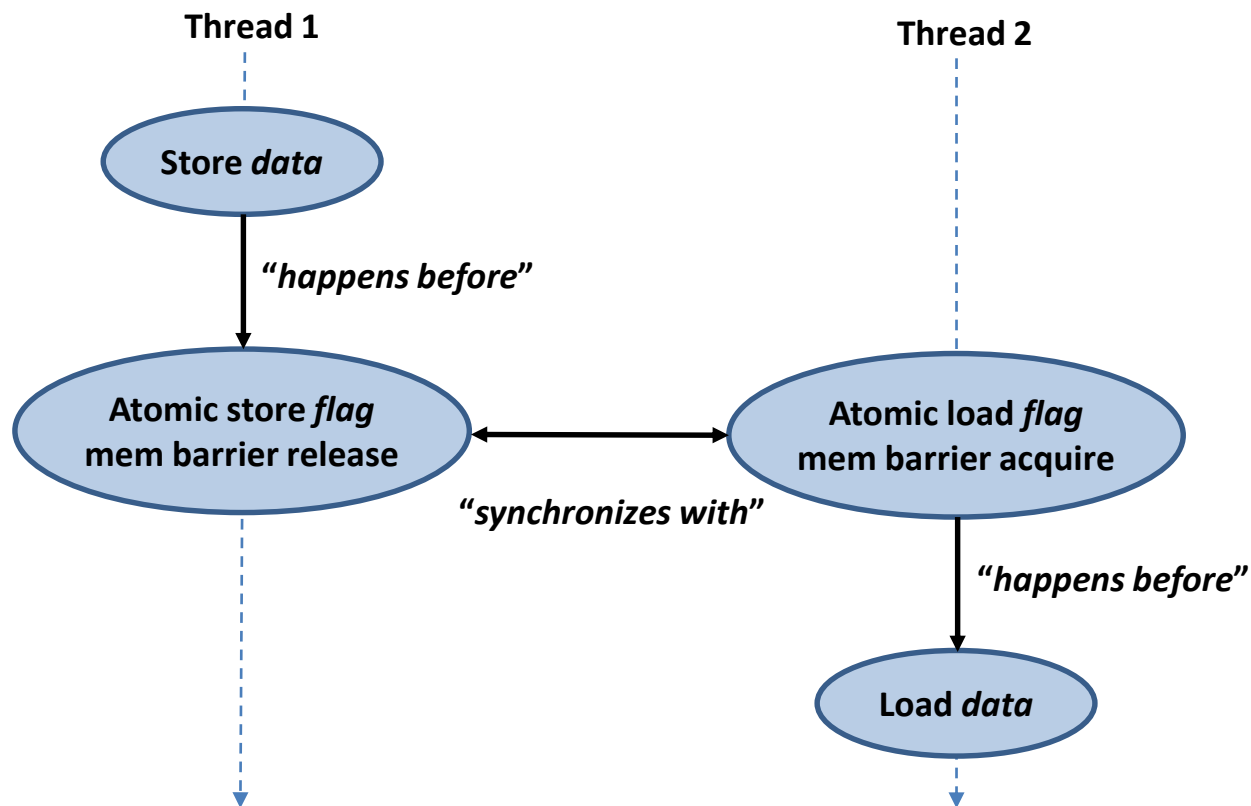
Whenever a variable is accessed by operations from two threads:

- Both operations are loads -or-
- Both are executed in a well-defined order

Inter-thread order requires explicit memory barriers.

**Carefully chosen barrier semantics** should limit performance penalties: impose weak ordering constraints

# Building ordering relations



Local order relations allow to extract global ordering (transitive closures)



If you want to learn more...



# atomic<> Weapons

The C++11 Memory Model  
and Modern Hardware

Herb Sutter

3hr presentation at "C++ and Beyond", Aug. 2012

# Example: ping-pong buffer

```
std::atomic<int> flag;
int bucket;

void consume() { // thread A
    while (true) {
        while (!flag.load(std::memory_order_acquire))
            ; // busy wait
        int my_work = bucket;
        flag.store(0, std::memory_order_release);
        consume_stuff( my_work );
    }
}

void produce() { // thread B
    while (true) {
        int my_stuff = produce_stuff();
        while (flag.load(std::memory_order_acquire))
            ;// busy wait
        bucket = mystuff;
        flag.store( 1, std::memory_order_release);
    }
}
```

# Example: ping-pong buffer

```
std::atomic<int> flag;  
int bucket;
```

```
void consume() { // thread A  
    while (true) {  
        while (!flag.load(std::memory_order_acquire))  
            ; // busy wait  
        int my_work = bucket;  
        flag.store(0, std::memory_order_release);  
        consume_stuff( my_work );  
    }  
}
```

```
void produce() { // thread B  
    while (true) {  
        int my_stuff = produce_stuff();  
        while (flag.load(std::memory_order_acquire))  
            ;// busy wait  
        bucket = mystuff;  
        flag.store( 1, std::memory_order_release);  
    }  
}
```

Ordered data dependency (RaW)

# Example: ping-pong buffer

```
std::atomic<int> flag;  
int bucket;
```

```
void consume() { // thread A  
    while (true) {  
        while (!flag.load(std::memory_order_acquire))  
            ; // busy wait  
        int my_work = bucket;  
        flag.store(0, std::memory_order_release);  
        consume_stuff( my_work );  
    }  
}
```

```
void produce() { // thread B  
    while (true) {  
        int my_stuff = produce_stuff();  
        while (flag.load(std::memory_order_acquire))  
            ; // busy wait  
        bucket = mystuff;  
        flag.store( 1, std::memory_order_release);  
    }  
}
```

Ordered anti dependency (WaR)

(?)

Ordered data dependency (RaW)

# Learn from this simple example

- Such low-level synchronization is still **hard and error-prone**.
  - You should **re-use higher-level functionality** offered through libraries.
  - Have clear semantics through well-known **design patterns**
- Checking for SC-DRF should be a tool responsibility. But, we are **not there yet...**

# Example with datarace (BAD!)

```
int main()
{
    // create an empty bucket
    std::set<int> bucket;

    // Use a background task to insert value '5' in the bucket
    std::thread t([&]() { bucket.insert(5); });

    // Check if value '3' is in the bucket (not expected :-))
    bool contains3 = bucket.find(3) != bucket.cend();
    std::cout << "Foreground find: " << contains3 << std::endl;

    // Wait for the background thread to finish
    t.join();

    // verify that value '5' did arrive in the bucket
    bool contains5 = bucket.find(5) != bucket.cend();
    std::cout << "Background: " << contains5 << std::endl;

    return 0;
}
```



# Issues with faulty std::set example

- C++ STL containers are **not thread-safe** for write access! Programmers would know to not create such code **if they read** their documentation
- If your job is to create concurrency in an existing large code base (>100K lines), code inspection would easily overlook this (the read and write could be far apart, in different files)
- The program seems to run fine: the bug reveals itself rarely
- Today's data-race checking **tools seem to miss this one**



# Conclusion

- C++11 obtained a properly defined memory model and threading primitives, finally allowing to create portable programs!
- Bold change: Atomics and volatile became totally different. Some compiler optimizations are now illegal.
- Creating deterministic (SC-DRF) programs remains challenging.
- The programmer community **needs more and better tools** to **improve productivity** and bridge the gap with multi-core hardware

# Thank you

Check [www.vectorfabrics.com](http://www.vectorfabrics.com) for a free demo on concurrency analysis

