

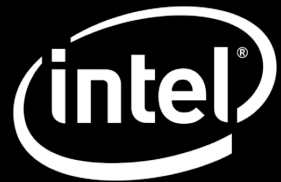
# Parallel Programming for C and C++ done right

(a work in progress)

James Reinders, Intel Corp.



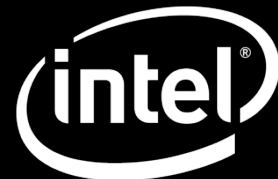
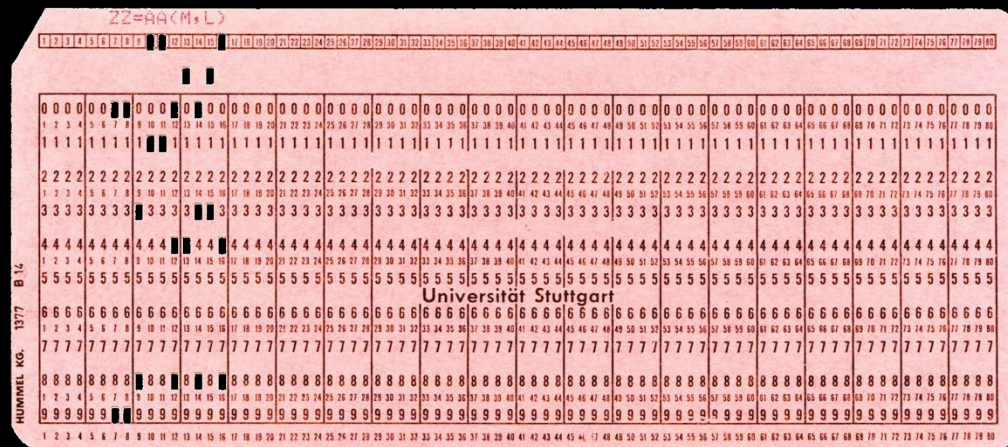
I  Fortran



# 1957 FORTRAN

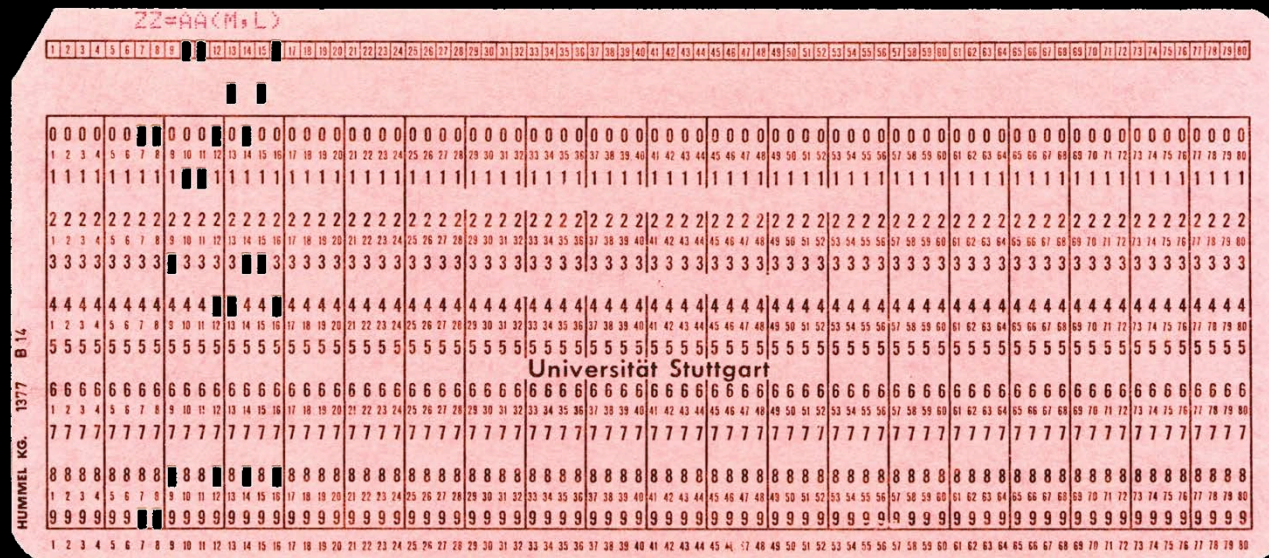
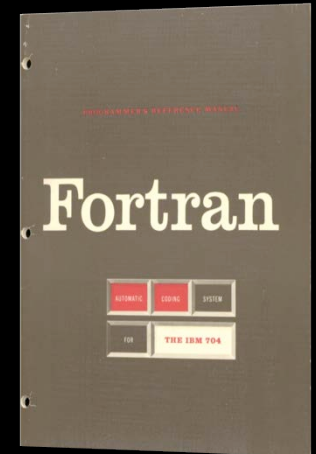
- Before microprocessors

```
WRITE (6,7)  
7 FORMAT(13H HELLO, WORLD)  
STOP  
END
```



# Fortran

- First Compiler: 1957
  - Some of the influences on design:
    - Punch-cards
    - Optimization  
(*users were reluctant to switch from assembly language*)
    - Manipulation of sense switches and sense lights
    - Mathematical exceptions (overflow, divide check)
    - Tape operations (read, write, rewind, backspace)



Photos: Wikimedia Commons (<http://commons.wikimedia.org>)



# Fortran

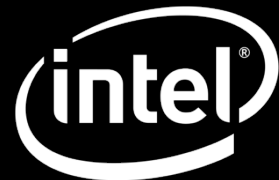
- Key adaptations that came later:
  - Subroutines and Functions (Fortran II, 1958)
  - File I/O, characters, strings (Fortran 77, 1978)
  - Recursion (Fortran 90, 1991)  
*[common non-standard extension available in many Fortran-77 compilers]*
  - Free-form input, not based on 80 column Punched Card (Fortran 90, 1991)
  - Variable names up to 31 characters instead of 6 (Fortran 90, 1991)
  - Inline comments (Fortran 90, 1991)
  - **Array Notations** (Fortran 90, 1991)
  - Operator overloading (Fortran 90, 1991)
  - Dynamic Memory Allocation (Fortran 90, 1991)
  - **FORALL** (Fortran 95, 1995)
  - OOP (Fortran 2003, 2003)
  - **DO CONCURRENT** (Fortran 2008, 2010)
  - **Co-Array Fortran** (Fortran 2008, 2010)



# Array Notation (Fortran90 [1991])

```
print *, a(:, 3) ! thirdcolumn  
print *, a(n, :) ! last row  
print *, a(:3, :3) ! Leading 3-by-3 submatrix
```

This is so important, I'll come back to it later.



# Coarray Fortran (Fortran 2008 [2010])

Sum in Fortran, using co-array feature:

```
REAL SUM[*]  
CALL SYNC_ALL( WAIT=1 )  
DO IMG= 2, NUM_IMAGES()  
    IF (IMG==THIS_IMAGE()) THEN  
        SUM = SUM + SUM[IMG-1]  
    ENDIF  
    CALL SYNC_ALL( WAIT=IMG )  
ENDDO
```

- **A standard, explicit notation for data decomposition**
- **Shared memory and distributed memory systems**



# OpenMP\* (Open Multi-Processing)

```
!$omp parallel do  
  do i=1,10  
    A(i) = B(i) * C(i)  
  enddo  
!$omp end parallel
```

- **Standard used by many parallel applications**
  - Supported by every major compiler for Fortran and C
- **OpenMP 4.0 in the works**





# DO CONCURRENT (Fortran 2008 [2010])

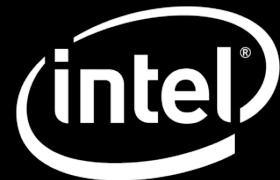
```
do concurrent (i=1:m)
  a(k+i) = a(k+i) + factor*a(l+i)
end do
```

- **OpenMP\* is a standard used by many parallel applications**
  - Supported by every major compiler for Fortran, C, and C++
- **OpenMP 4.0 in the works**



# Fortran got “right”

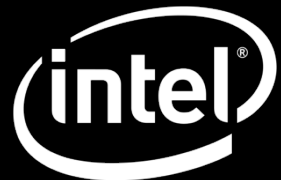
- avoids evil pointers
  - helps optimizations
- supports arrays directly
  - helps vectorization
- straight forward usage (no templates, etc.)
  - helps mask composability issues with OpenMP
- Still: C and C++ needed in the universe and they need help (more)



I  Fortran

but...

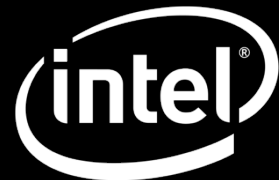
I promised to talk about C and C++



# 1972 C

- Before microprocessors

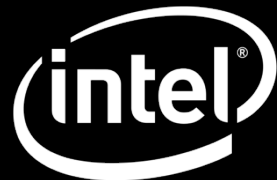
```
#include <stdio.h>
int main(int argc, char *argv[])
{
    printf("Hello, World!\n");
    return 0;
}
```



# 1983 C++

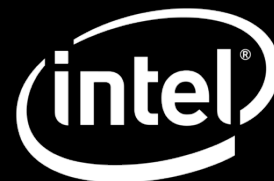
- 80286 (16 bits)

```
#include<iostream>
using namespace std;
int main(int argc, const char *argv[])
{
    cout << "Hello, World!" << endl;
    return 0;
}
```



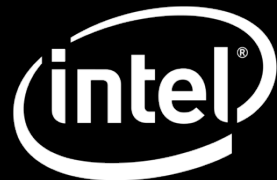
# in order to fix

- We need to jointly understand the “problem”

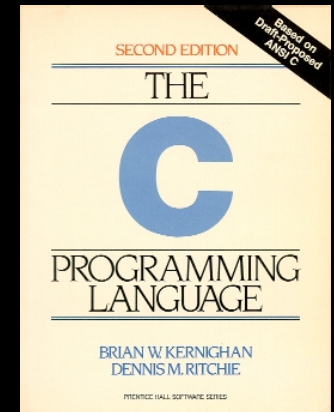


# I will motivate more than show the solution

- I'm learning that SHOWING SOLUTIONS is useless if the PROBLEM is not FELT
- Our solutions are heavily adopted by those who were in pain already!



- C
  - early key features “register” keyword out of use
  - “volatile” fading in usage
  - added: stronger typing (ANSI C, 1989)
  - C11
  - OpenMP\* (1996)
  - Cilk™ Plus (2010)



- C++
  - Objected oriented
  - Intel® Threading Building Blocks (2006)
  - C++11
  - Cilk™ Plus (2010)

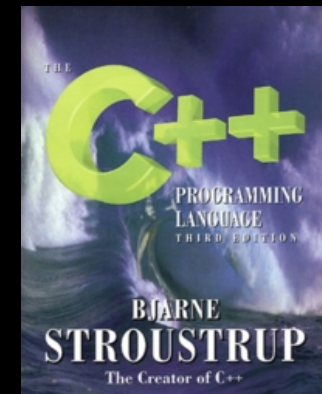
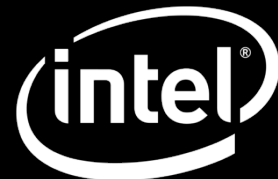


Photo: Wikimedia Commons (<http://commons.wikimedia.org>)





# C++11 (some applies to C11 also)

## Core language runtime performance enhancements

- Rvalue references and move constructors
- Generalized constant expressions
- Modification to the definition of plain old data

## Core language build time performance enhancements

- Extern template

## Core language usability enhancements

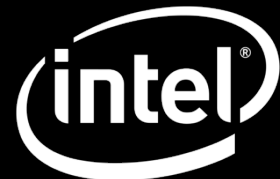
- Initializer lists
  - Uniform initialization
  - Type inference
  - ~~Range based for-loop~~
  - **Lambda functions and expressions**
  - Alternative function syntax
  - Object construction improvement
  - Explicit overrides and final
  - Null pointer constant
  - Strongly typed enumerations
  - Right angle bracket
  - Explicit conversion operators
  - Alias templates
  - Unrestricted unions
- anonymous functions**

## Core language functionality improvements

- Variadic templates
- New string literals
- User-defined literals
- **Multithreading memory model**
- ~~Thread-local storage~~
- Explicitly defaulted and deleted special member functions
- Type long long int
- Static assertions
- Allow sizeof to work on members of classes without an explicit object
- Control and query object alignment
- Allow garbage collected implementations

## C++ standard library changes

- Upgrades to standard library components
  - **Threading facilities**
  - Tuple types
  - Hash tables
  - Regular expressions
  - General-purpose smart pointers
  - Extensible random number facility
  - Wrapper reference
  - Polymorphic wrappers for function objects
  - Type traits for metaprogramming
  - Uniform method for computing the return type of function objects
- futures & promises, async**



# What about futures & promises?

future : *think of as a* consumer end of a 1-element produce/consumer queue

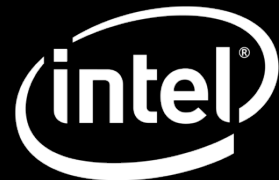
- A future can be created only from an existing promise object.
- Producer computes the value: calls `set_value()` on the promise.
- Consumer needs the future value: it calls `get()` on the future.
- Consumer blocks waiting on the producer if producer has not yet `set_value()`.
- Futures can be used via the `async()` member function.

```
double foo(double arg); // consider normal function
```

```
// You can execute foo(x) asynchronously by calling  
std::future<double> result = std::async(foo, x);
```

```
...
```

```
double val = result.get();
```



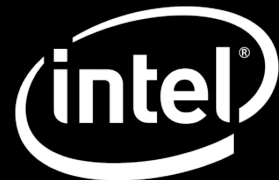
# What about futures & promises?

The problems with the future/async model are both linguistic and performance-related.

The key flaw is that the whole notion of scalability with using futures was soundly refuted in the seminal 1993 paper:

*Space-efficient scheduling of multithreaded computations* by Blumofe and Leiserson.

This is the paper that motivated the development of Cilk in the first place.



# What about futures & promises?

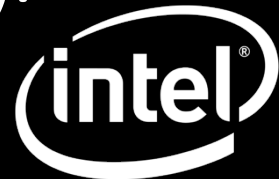
The linguistic problems are more subtle.

The following two statements that do roughly the same thing:

```
std::future<double> result = std::async(foo, x);  
double result = cilk_spawn foo(x);
```

The first statement looks like a call to `async()`.

The second statement looks like a call to `foo()`.



# What about futures & promises?

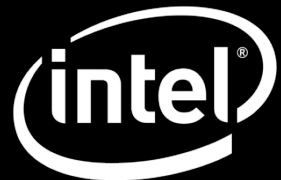
Semantically, consider the following:

```
std::string s("hello");
```

```
int bar(const std::string& s);
```

```
std::future<int> result = std::async(bar, s + " world");
```

The above statement is intended to pass "hello world" to bar and run it asynchronously.



# What about futures & promises?

Semantically, consider the following:

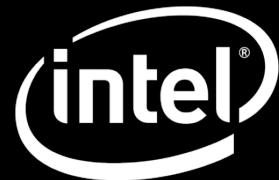
```
std::string s("hello");
```

```
int bar(const std::string& s);
```

```
std::future<int> result = std::async(bar, s + " world");
```

The above statement is intended to pass "hello world" to bar and run it asynchronously.

The problem is that `s + " world"` is a *temporary* object that gets destroyed as soon as the statement completes.



# What about futures & promises?

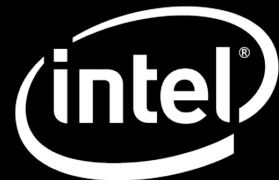
```
std::string s("hello");  
int bar(const std::string& s);
```

```
std::future<int> result = std::async(bar, s + " world");
```

Boosters of `std::async` will counter that all you need is to add a lambda:

```
std::future<int> result = std::async([&]{ bar(s + " world"); });
```

Without the lambda - it is a *race condition* that should **not** exist in a linguistically sound parallel construct, but it is pretty much unavoidable in a library-only specification.

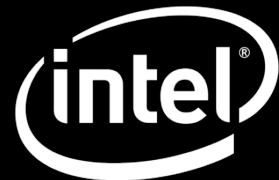


# Let's fix C and C++

and the solution is NOT OpenMP

and the solution is NOT CUDA

and the solution is NOT OpenCL





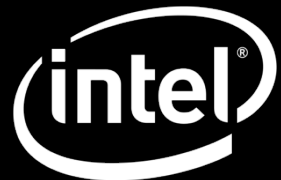
# Let's fix C and C++

and the solution is NOT OpenMP

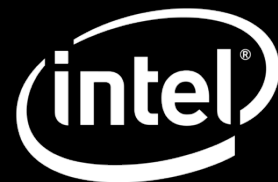
and the solution is NOT CUDA

and the solution is NOT OpenCL

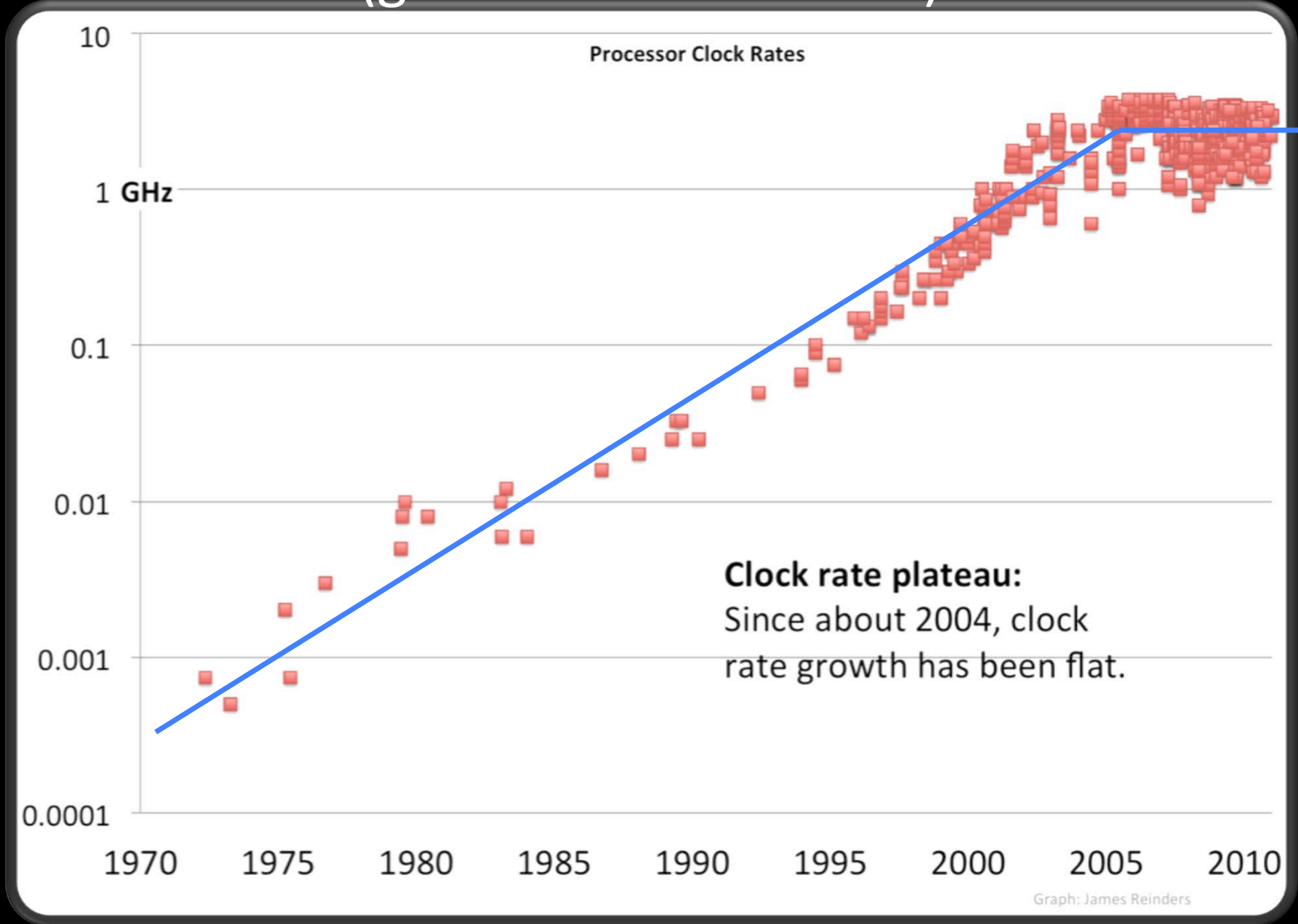
for starters, none of them are “composable”



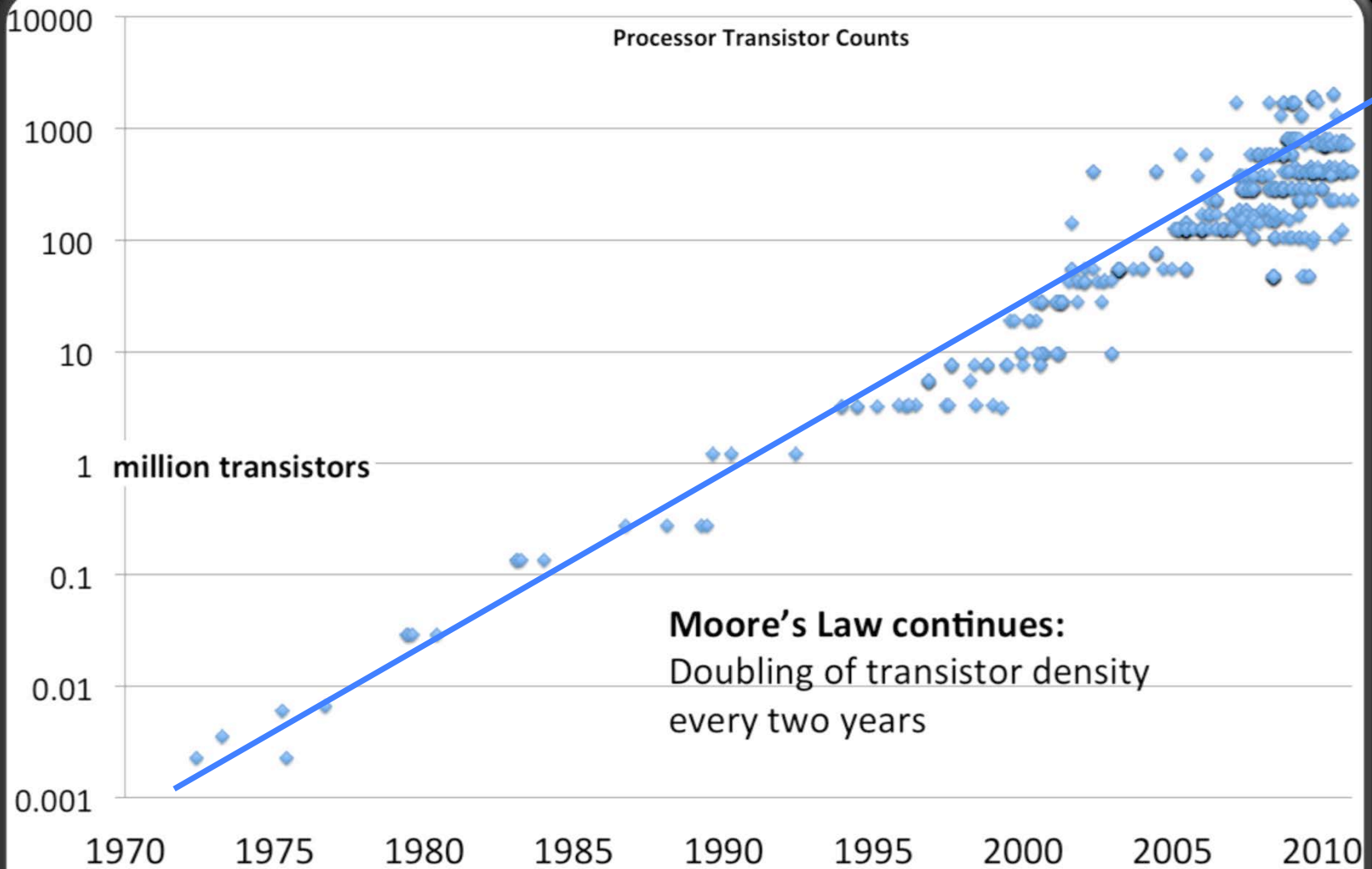
# Problem Statement?



# Processor Clock Rate (growth halted ~2005)



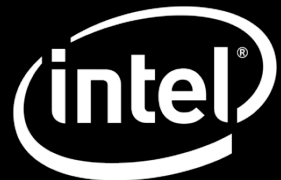
# Transistors per Processor Continuing to grow (Moore's Law)



Graph: James Reinders

# Problem Statement

- Parallel Hardware
  - Scale
  - Vectorize
  - Specialization



# Problem Statement

- Parallel Hardware
  - Scale
  - Vectorize
  - Specialization

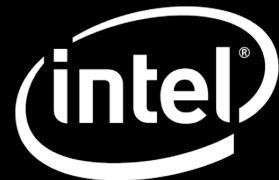
Let's think about **HARDWARE TRENDS**.



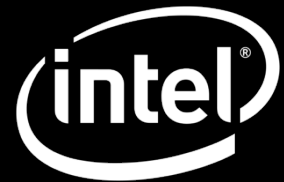
# Problem Statement

- Parallel Hardware
  - Scale
  - Vectorize
  - Specialization

Scale: cores, execution units

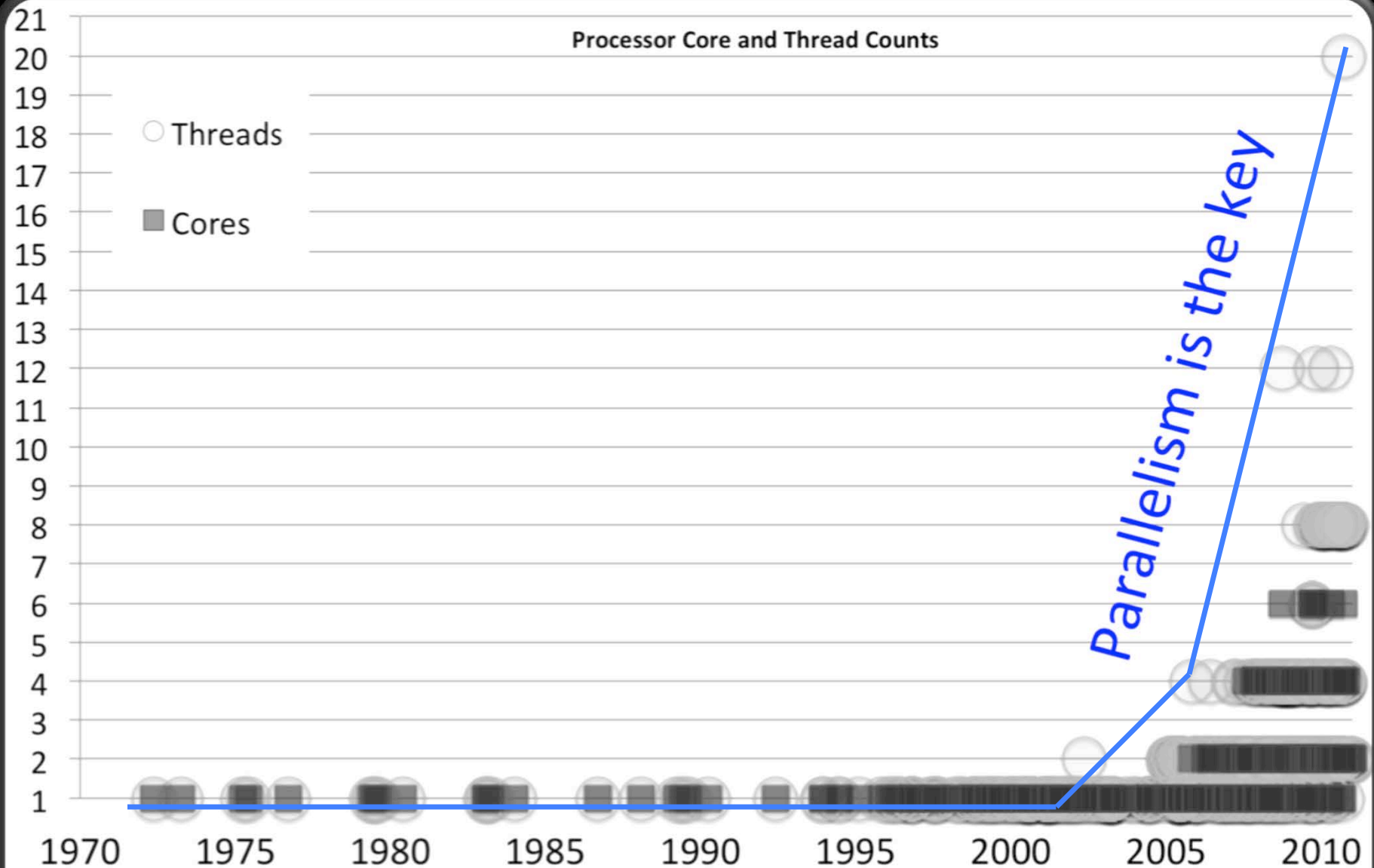


# Locks Kill Scaling



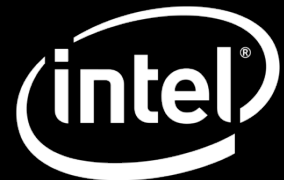


# Hardware Threads ○ & Cores ■



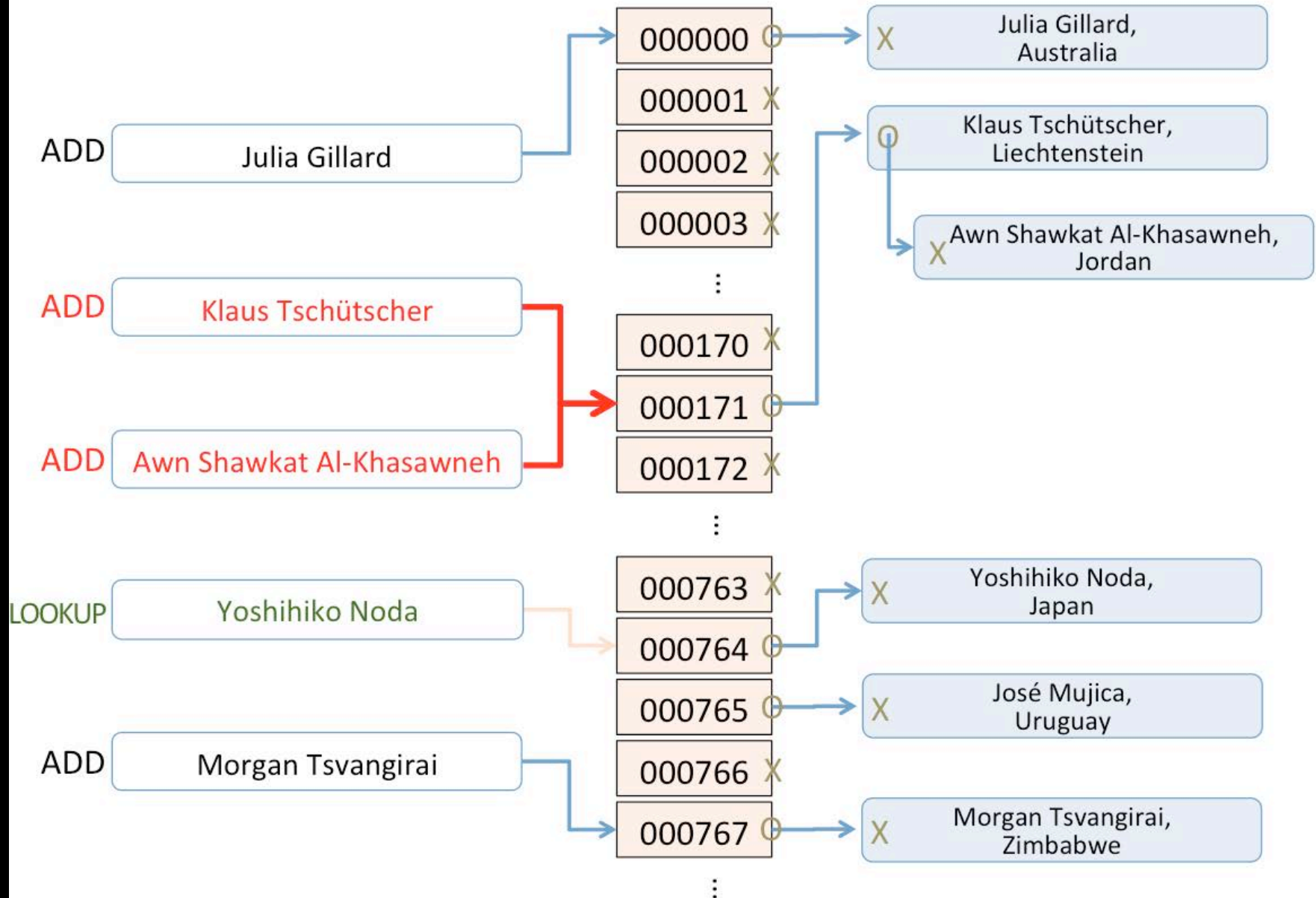
Graph: James Reinders

Locks Kill Scaling  
often



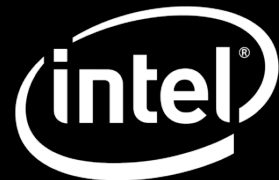
## The FIVE operations

## The HASH table and contents (after the operations)

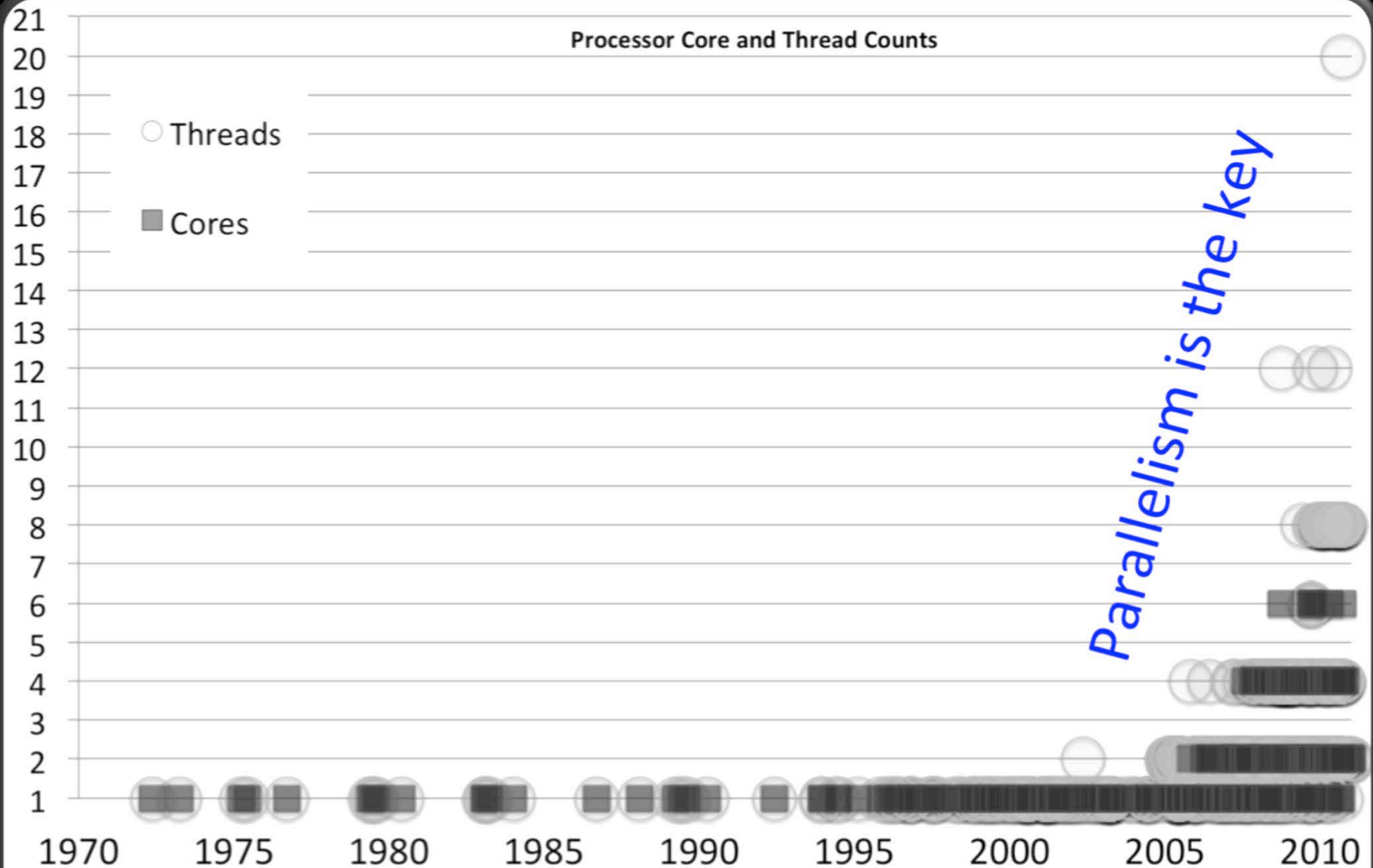


# Transactional Synchronization Extensions

- a beautiful example of **HARDWARE** making life “simple again” helping **CONCURRENCY / PARALLEL PROGRAMMING**
- **HLE** is a hint inserted in front of a LOCK operation to indicate a region is a candidate for lock elision
  - XACQUIRE (0xF2) and XRELEASE (0xF3) prefixes
  - Don’t actually acquire lock, but execute region speculatively
  - Hardware buffers loads and stores, checkpoints registers
  - Hardware attempts to commit atomically without locks
  - If cannot do without locks, restart, execute non-speculatively
- **RTM** is three new instructions (XBEGIN, XEND, XABORT)
  - Similar operation as HLE (except no locks, new ISA)
  - If cannot commit atomically, go to handler indicated by XBEGIN
  - Provides software additional capabilities over HLE



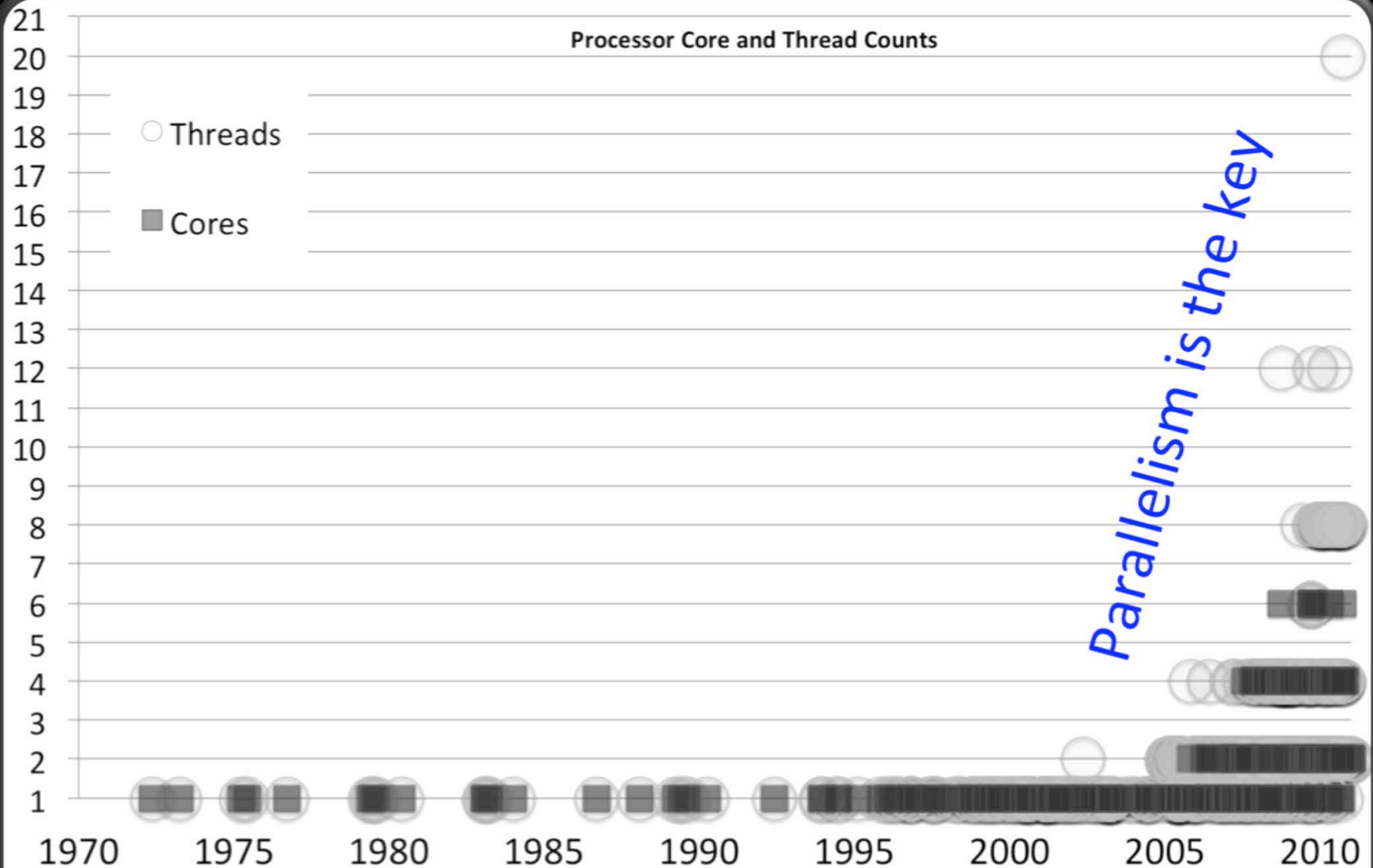
# Hardware Threads ○ & Cores ■



Parallelism is the key

Graph: James Reinders

# Hardware Threads ○ & Cores ■



Graph: James Reinders

# Hardware Threads ○ & Cores ■



Over 200 hardware threads

Over 50 cores



Corner

...the *first*  
using Intel's  
MIC architecture

# 1996



ASCI Red: 1 TeraFlop/sec  
December 1996

1996 First System 1 TF/s  
Sustained

(with 2/3<sup>rd</sup> of the system built...  
7264 Intel® Pentium Pro processors)

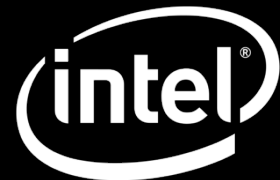
OS: Cougar

72 Cabinets

\* Full system 1.3 TeraFlop/sec, later upgraded  
to 3.1 TeraFlops/sec with  
9298 Intel® Pentium II Xeon processors  
Source and Photo: [http://en.wikipedia.org/wiki/ASCI\\_Red](http://en.wikipedia.org/wiki/ASCI_Red)

## Intel: Shattering Barriers

More than one sustained  
TeraFlop/sec





# 2011



ASCI Red: 1 TeraFlop/sec  
December 1996

1996 First System 1 TF/s  
Sustained

(with 2/3<sup>rd</sup> of the system built...  
7264 Intel® Pentium Pro processors)

OS: Cougar

72 Cabinets

\* Full system 1.3 TeraFlop/sec, later upgraded  
to 3.1 TeraFlops/sec with  
9298 Intel® Pentium II Xeon processors  
Source and Photo: [http://en.wikipedia.org/wiki/ASCI\\_Red](http://en.wikipedia.org/wiki/ASCI_Red)



Knights Corner: 1 TeraFlop/sec  
November 2011

2011 First Chip 1 TF/s Sustained

One 22nm Chip

OS: Linux\*

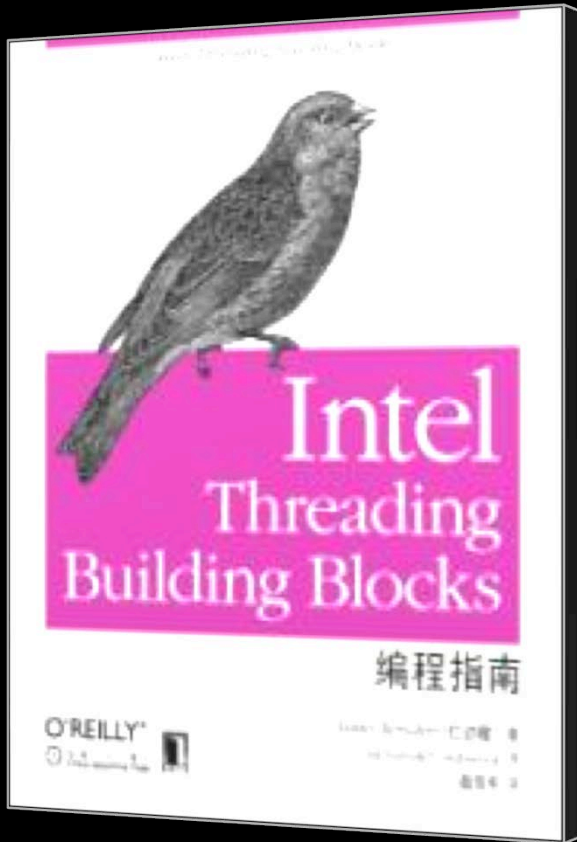
One PCI express slot

\* Other names and brands may be claimed as the property of others.

© Intel 2012, All Rights Reserved



# www.threadingbuildingblocks.org



- ✓ Most popular C++ abstraction
- ✓ Windows\*
- ✓ Linux\*
- ✓ Mac OS\* X
- ✓ Xbox 360
- ✓ Solaris\*
- ✓ FreeBSD\*
- ✓ Intel processors
- ✓ AMD processors
- ✓ SPARC processors
- ✓ IBM processors
- ✓ open source
- ✓ standard committee submissions

The most used method to parallelize C++ programs

\* Other names and brands may be claimed as the property of others.

# Intel® Threading Building Blocks

## Generic Parallel Algorithms

Efficient scalable way to exploit the power of multi-core without having to start from scratch

## Concurrent Containers

Common idioms for concurrent access  
- a scalable alternative serial container with a lock around it

## Thread Local Storage

Scalable implementation of thread-local data that supports infinite number of TLS

## Task scheduler

The engine that empowers parallel algorithms that employs task-stealing to maximize concurrency

## Synchronization Primitives

User-level and OS wrappers for mutual exclusion, ranging from atomic operations to several flavors of mutexes and condition variables

## Miscellaneous

Thread-safe timers

## Threads

OS API wrappers

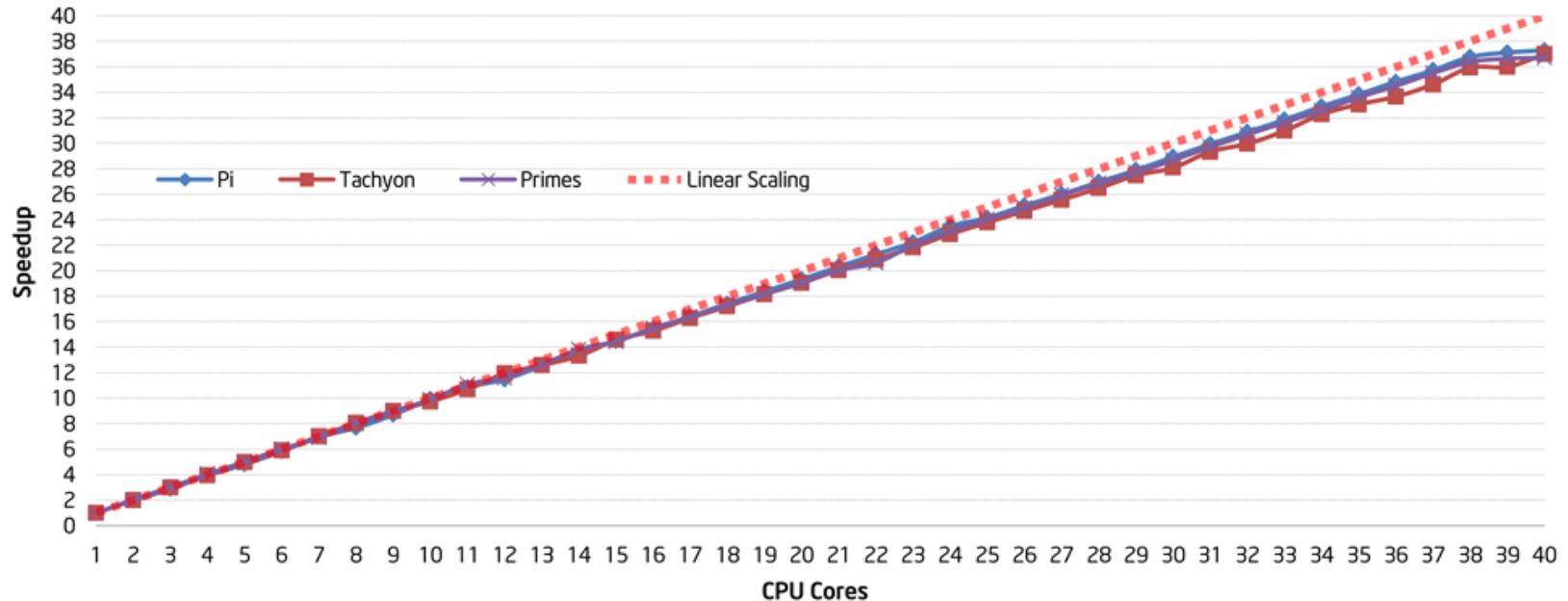
## Memory Allocation

Per-thread scalable memory manager and false-sharing free allocators

# Scale Forward

Intel® Threading Building Blocks 4.0 scales exceptionally well

Intel® Threading Building Blocks 4.0  
Exhibits Linear Performance Scalability on 40-core System



Configuration Info - SW Versions: Intel® C++ Intel® 64 Compiler, Version 12.1, Intel® Threading Building Blocks 4.0; Hardware: 4 \* Intel® Xeon® CPU E7- 4860 @ 2.27GHz (40 cores), 256GB Main Memory; Operating System: Linux, Red Hat® Enterprise Server® release 5.4, kernel 2.6.18-194.11.4.el5; Benchmark Source: Intel Corp.

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, refer to [www.intel.com/performance/resources/benchmark\\_limitations.htm](http://www.intel.com/performance/resources/benchmark_limitations.htm). \* Other brands and names are the property of their respective owners

**Optimization Notice:** Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804

# Intel® TBB Class Graph: Components

New Feature as of TBB 4.0 Release (2011)

- Graph object

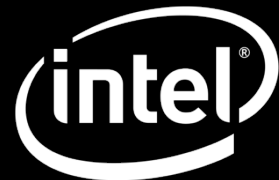
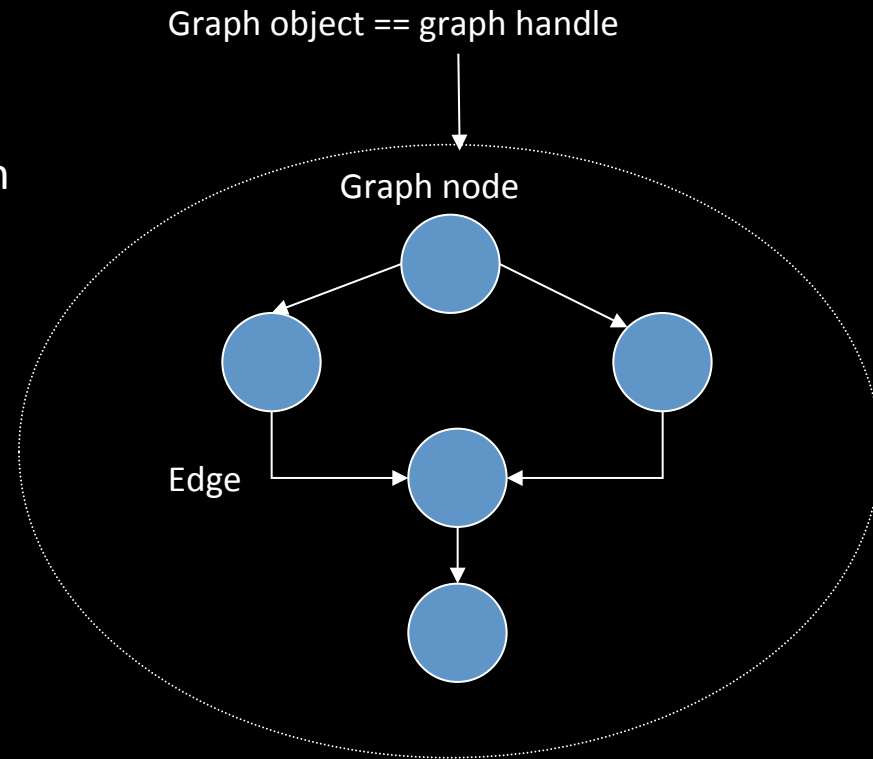
- Contains a pointer to the root task
- Owns tasks created on behalf of the graph
- Users can wait for the completion of all tasks of the graph

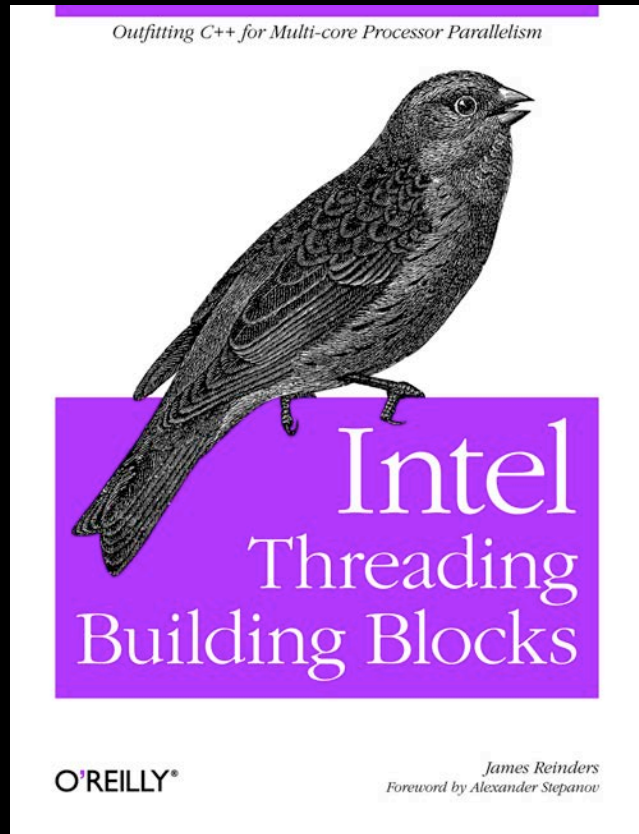
- Graph nodes

- Implement *sender* and/or *receiver* interfaces
- Nodes manage messages and/or execute function objects

- Edges

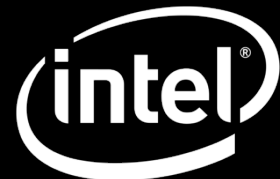
- Connect predecessors to successors





## TBB for C++ scaling

Most popular solution for  
C++ parallel programming





Outfitting C++ for Multi-core Processor Parallelism



O'REILLY®

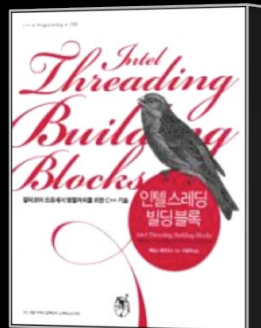
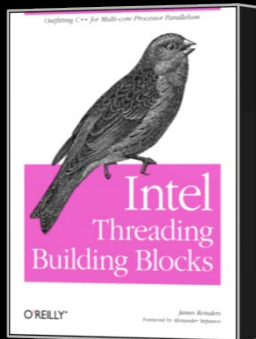
James Reinders  
Foreword by Alexander Stepanov

threadingbuildingblocks.org  
cilkplus.org

TBB has a “sister”

Cilk™ Plus:

- Help for C programmers
- Involve compiler
- Vectorization support



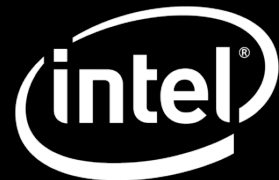
# Scale Efficiently

## Intel® Cilk™ Plus, three keywords to go parallel

```
cilk_for (int i=0; i<n; ++i)  
    Foo(a[i]);  
}
```

Parallel loops made easy

Open specification at [cilkplus.org](http://cilkplus.org)





# Scale Efficiently

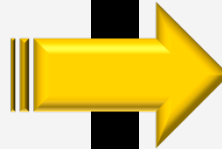
## Intel® Cilk™ Plus, three keywords to go parallel

```
cilk_for (int i=0; i<n; ++i) {  
    Foo(a[i]);  
}
```

Parallel loops made easy

```
int fib(int n)  
{  
    if (n <= 2)  
        return n;  
    else {  
        int x,y;  
        x = fib(n-1);  
        y = fib(n-2);  
        return x+y;  
    }  
}
```

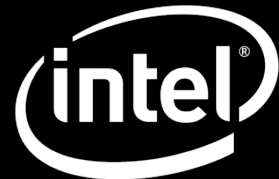
Turn serial code



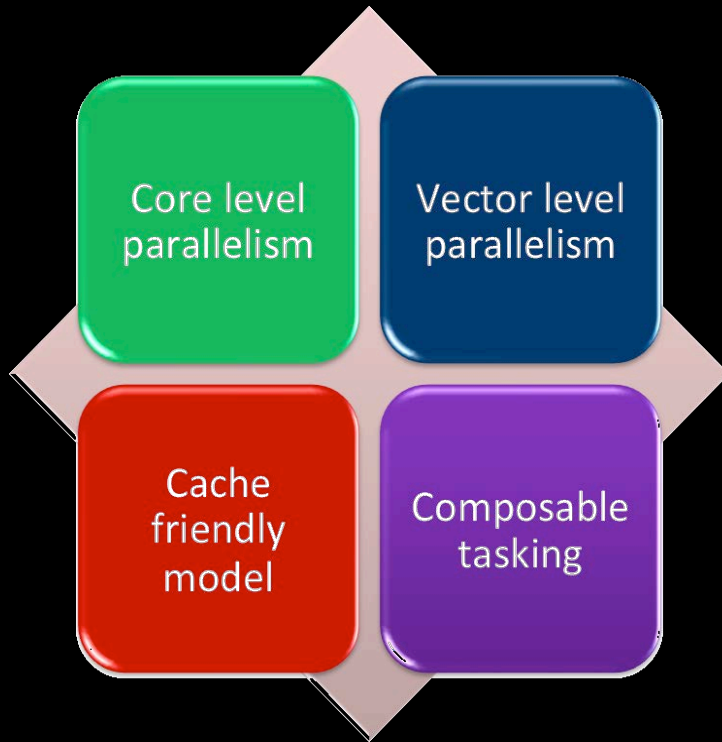
```
int fib(int n)  
{  
    if (n <= 2)  
        return n;  
    else {  
        int x,y;  
        x = cilk_spawn fib(n-1);  
        y = fib(n-2);  
        cilk_sync;  
        return x+y;  
    }  
}
```

Into parallel code

Open specification at [cilkplus.org](http://cilkplus.org)

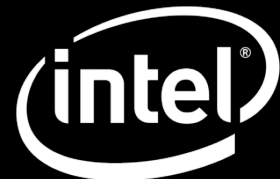


# www.cilkplus.org



- ✓ Windows\*
- ✓ Linux\*
- ✓ Mac OS\* X
- ✓ gcc: experimental branch
- ✓ open specification
- ✓ other compiler vendors reviewing
- ✓ standard committee submissions

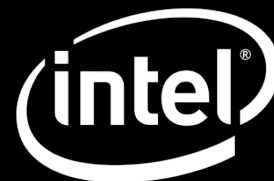
\* Other names and brands may be claimed as the property of others.



# Problem Statement

- Parallel Hardware
  - Scale
  - **Vectorize**
  - Tap specialization

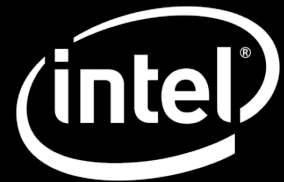
**Scale: wider vectors instructions, warps...**



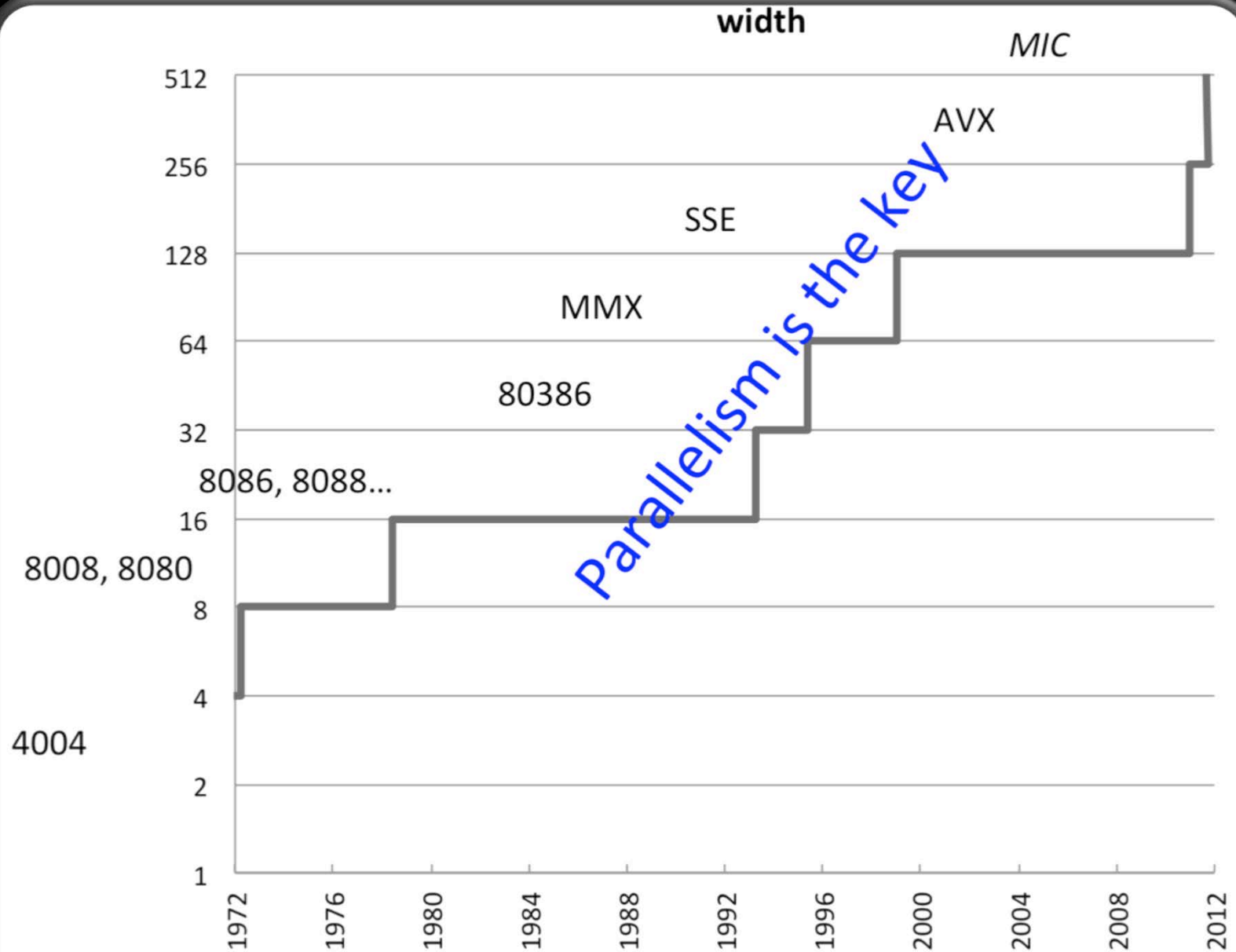
# Cray Supercomputers



Photo: Wikimedia Commons (<http://commons.wikimedia.org>)



# Vector Width



Graph: James Reinders

# Auto Vectorization: Useful, but limited by language

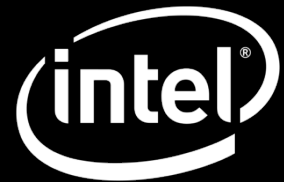
```
void v_add (float *c, float *a, float *b)
{
    for (int i=0; i<= MAX; i++)
        c[i]=a[i]+b[i];
}
```

- C/C++ language implies that vectorizing this loop is “illegal”
- Some code can be re-written in a way that the compiler can vectorize
- Hard to learn
- Impossible to completely automate

Consider a Solution:  
Allow the programmer to express  
operations without unintended serial  
execution, using a new syntax.

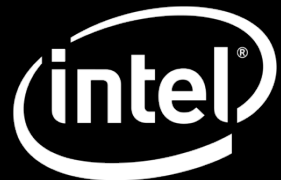


# What went wrong?



# What went wrong?

- Arrays not really in the language
- Pointers are, evil pointers!





## 1966: Fortran 66

The Fortran 66 programmers were only able to take advantage of slicing matrices by row, and then only when passing that row to a [subroutine](#):

```
SUBROUTINE PRINT V(VEC, LEN)
  REAL VEC(*)
  PRINT *, (VEC(I), I = 1, LEN)
END

PROGRAM MAIN
  PARAMETER(LEN = 3)
  REAL MATRIX(LEN, LEN)
  DATA MATRIX/1, 1, 1, 2, 4, 8, 3, 9, 27/
  CALL PRINT V(MATRIX(1, 2), LEN)
END
```

Result:

```
2. 4. 8.
```

Note that there is no [dope vector](#) in FORTRAN 66 hence the length of the slice must also be passed as an argument - or some other means - to the [SUBROUTINE](#). 1970s [Pascal](#) and [C](#) had similar restrictions.

## 1968: Algol 68

Algol68 final report contains an early example of slicing, slices are specified in the form:

```
[lower bound:upper bound] ¢ for computers with extended character sets ¢
```

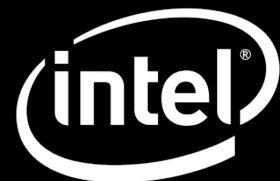
or

```
(LOWER BOUND..UPPER BOUND) # FOR COMPUTERS WITH ONLY 6 BIT CHARACTERS. #
```

Both bounds are inclusive and can be omitted, in which case they default to the declared array bounds. Neither the stride facility, nor diagonal slice aliases are part of the revised report.

Source: [http://en.wikipedia.org/wiki/Array\\_slicing](http://en.wikipedia.org/wiki/Array_slicing)

© Intel 2012, All Rights Reserved



## 1970s: MATLAB/GNU Octave/Scilab

```
> A = round(rand(3, 4, 5)*10) # 3x4x5 three-dimensional or cubic array
> A(:, :, 3) # 3x4 two-dimensional array along first and second dimensions
ans =

     8     3     5     7
     8     9     1     4
     4     4     2     5

> A(:, 2:3, 3) # 3x2 two-dimensional array along first and second dimensions
ans =

     3     5
     9     1
     4     2

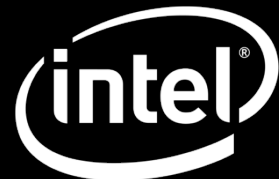
> A(1, :, 3) # single-dimension array along second dimension
ans =

     8     3     5     7

> A(1, 2, 3) # single value
ans = 3
```

Source: [http://en.wikipedia.org/wiki/Array\\_slicing](http://en.wikipedia.org/wiki/Array_slicing)

© Intel 2012, All Rights Reserved



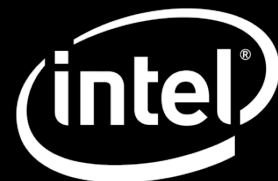
## 1976: S/R

Arrays in **S** and **GNU R** are always one-based, thus the indices of a new slice will begin with *one* for each dimension, regardless of the previous indices. Dimensions with length of *one* will be dropped (unless `drop = FALSE`). Dimension names (where present) will be preserved.

```
> A <- array(1:60, dim = c(3, 4, 5)) # 3x4x5 three-dimensional or cubic array
> A[, , 3] # 3x4 two-dimensional array along first and second dimensions
      [, 1] [, 2] [, 3] [, 4]
[1,]    25    28    31    34
[2,]    26    29    32    35
[3,]    27    30    33    36
> A[, 2:3, 3, drop = FALSE] # 3x2x1 cubic array subset (preserved dimensions)
      [, 1] [, 2]
[1,]    28    31
[2,]    29    32
[3,]    30    33
> A[, 2, 3] # single-dimension array along first dimension
[1] 28 29 30
> A[1, 2, 3] # single value
[1] 28
```

Source: [http://en.wikipedia.org/wiki/Array\\_slicing](http://en.wikipedia.org/wiki/Array_slicing)

© Intel 2012, All Rights Reserved



# 1977: Fortran 77

The Fortran 77 standard introduced the ability to slice and concatenate strings:

```
PROGRAM MAIN  
  PRINT *, 'ABCDE' (2:4)  
END
```

Produces:

```
BCD
```

Source: [http://en.wikipedia.org/wiki/Array\\_slicing](http://en.wikipedia.org/wiki/Array_slicing)

© Intel 2012, All Rights Reserved



## 1983: Ada 83 and above

Ada 83 supports slices for all array types. Like Fortran 77 such arrays could be passed by reference to another subroutine, the length would also be passed transparently to the subroutine as a kind of **short** dope vector.

```
with Text_IO;  
  
procedure Main is  
    Text : String := "ABCDE";  
begin  
    Text_IO.Put_Line (Text (2 .. 4));  
end Main;
```

Produces:

BCD

Source: [http://en.wikipedia.org/wiki/Array\\_slicing](http://en.wikipedia.org/wiki/Array_slicing)

© Intel 2012, All Rights Reserved





# 1987: Perl

If we have

```
@a = (2, 5, 7, 3, 8, 6, 4)
```

as above, then the first 3 elements, middle 3 elements and last 3 elements would be:

```
@a[0..2];    # (2, 5, 7)
@a[3..5];    # (7, 3, 8);    #should be (3,8,6) perl -e '@a = (2, 5, 7, 3, 8, 6, 4); print @a
@a[-3..-1];  # (8, 6, 4);
```

Perl supports negative list indices. The -1 index is the last element, -2 the penultimate element, etc. In addition Perl supports slicing based on expressions, for example:

```
@a[ 3.. $#a ];    # 4th element until the end (3, 8, 6, 4)
@a[ grep { !($_ % 3) } (0...$#a) ]    # 1st, 4th and 7th element (2,3,4)
@a[ grep { !(($_+1) % 3) } (0..$#a) ] # every 3rd element (7,6)
```

Source: [http://en.wikipedia.org/wiki/Array\\_slicing](http://en.wikipedia.org/wiki/Array_slicing)

© Intel 2012, All Rights Reserved



# 1991: Python

If you have a list

```
nums = [1, 3, 5, 7, 8, 13, 20]
```

, then it is possible to slice by using a notation similar to element retrieval:

```
nums[3]    #equals 7, no slicing  
nums[:3]   #equals [1, 3, 5], from index 0 (inclusive) until index 3 (exclusive)  
nums[1:5]  #equals [3, 5, 7, 8]  
nums[-3:]  #equals [8, 13, 20]
```

Note that Python allows negative list indices. The index -1 represents the last element, -2 the penultimate element, etc. Python also allows a step property by append an extra colon and a value. For example:

```
nums[3::]  #equals [7, 8, 13, 20], same as nums[3:]  
nums[::3]  #equals [1, 7, 20] (starting at index 0 and getting every third element)  
nums[1:5:2] #equals [3, 7] (from index 1 until index 5 and getting every second element)
```

Source: [http://en.wikipedia.org/wiki/Array\\_slicing](http://en.wikipedia.org/wiki/Array_slicing)

© Intel 2012, All Rights Reserved





# 1992: Fortran 90 and above

In Fortran 90, slices are specified in the form

```
lower_bound:upper_bound[:stride]
```

Both bounds are inclusive and can be omitted, in which case they default to the declared array bounds. Stride defaults to 1. Example:

```
real, dimension(m, n):: a  ! declaration of a matrix

print *, a(:, 2)  ! second column
print *, a(m, :)  ! last row
print *, a(:10, :10)  ! leading 10-by-10 submatrix
```

Source: [http://en.wikipedia.org/wiki/Array\\_slicing](http://en.wikipedia.org/wiki/Array_slicing)

© Intel 2012, All Rights Reserved



## 1998: S-Lang

Array slicing was introduced in version 1.0. Earlier versions did not support this feature.

Suppose that A is a 1-d array such as

```
A = [1:50];           % A = [1, 2, 3, ..., 49, 50]
```

Then an array B of first 5 elements of A may be created using

```
B = A[[:4]];
```

Similarly, B may be assigned to an array of the last 5 elements of A via:

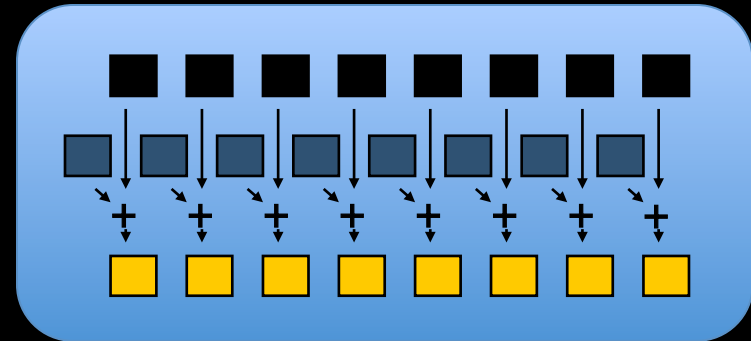
```
B = A[[-5:]];
```

# Cilk™ Plus solution: Array Notations → Vector Operations

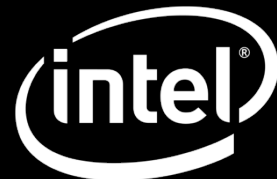
`<array base> [<lower bound>:<length>[:<stride>]]+`

`A[:]` // All of vector A  
`B[2:6]` // Elements 2 to 7 of vector B  
`C[:,5]` // Column 5 of matrix C  
`D[0:3:2]` // Elements 0,2,4 of vector D

```
if (a[:] > b[:]) {  
    c[:] = d[:] * e[:];  
} else {  
    c[:] = d[:] * 2;  
}
```



*A simple and elegant solution:  
a language construct for vector level  
parallelism.*



## 2010: Cilk Plus

Cilk Plus supports syntax for array slicing as an extension to C and C++.

```
array_base [lower_bound:length[:stride]]*
```

Cilk Plus slicing looks as follows:

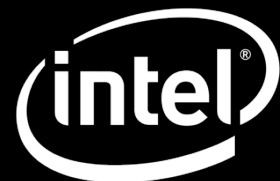
```
A[:]      // All of vector A  
B[2:6]    // Elements 2 to 7 of vector B  
C[:,5]    // Column 5 of matrix C  
D[0:3:2]  // Elements 0,2,4 of vector D
```

Differs from Fortran array slicing syntax by using length as the second parameter instead of the upper bound, in order to be consistent with standard C libraries.

Differs from Fortran array assignment semantics in that assignments are required to be either non-overlapping or perfectly overlapping, otherwise the result is undefined. This means temporaries are never required by the semantics.

Source: [http://en.wikipedia.org/wiki/Array\\_slicing](http://en.wikipedia.org/wiki/Array_slicing)

© Intel 2012, All Rights Reserved



# Better than Intrinsics Code

Intrinsics

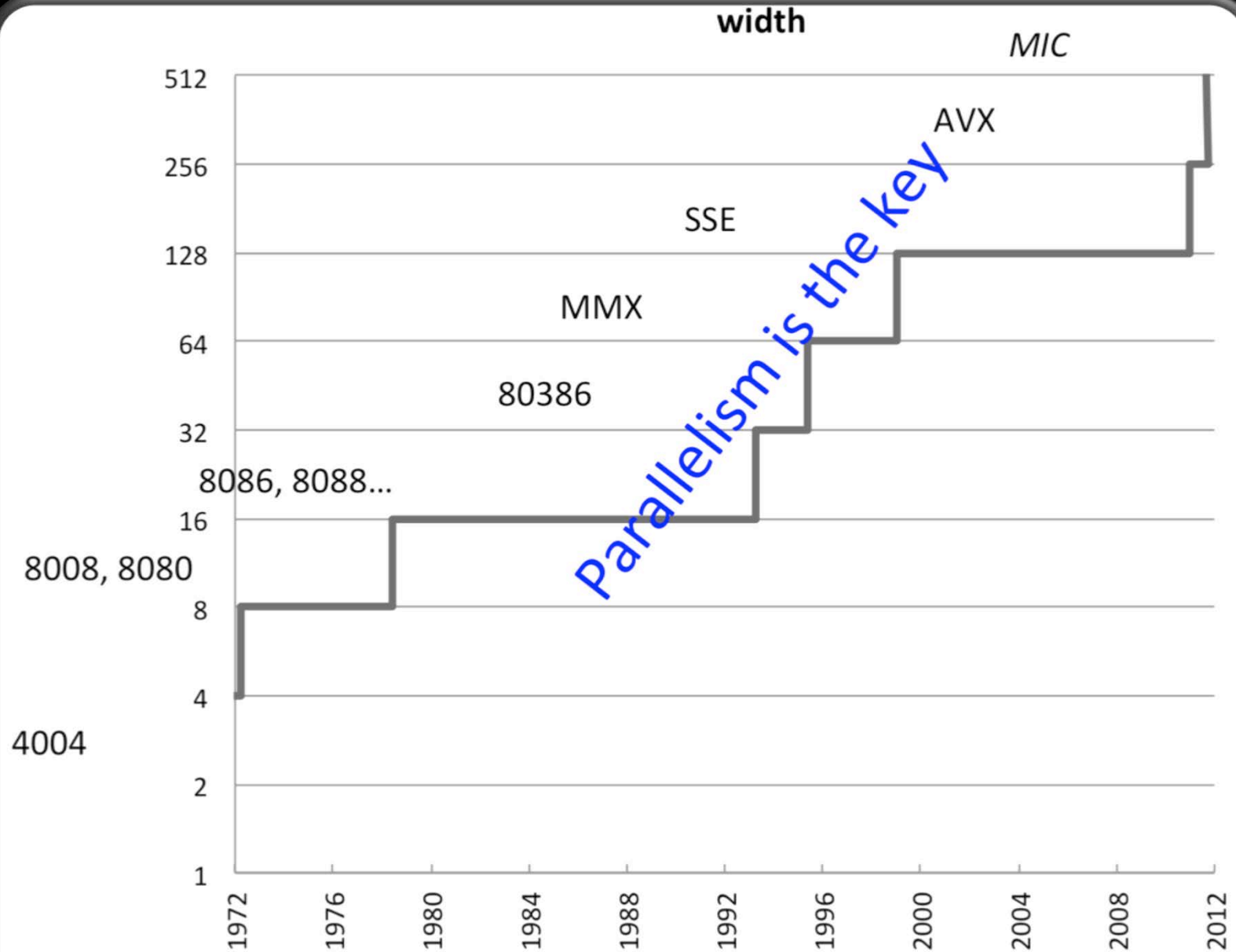
```
for(j = 0; j < num_pnt-3; j+=4) {  
    v_specularN = _mm_mul_ps(v_specularN, v_cosalpha);  
    cmp = _mm_cmpgt_ps(v_cosalpha, v_zero);  
    v_specularN = _mm_and_ps(v_specularN, cmp);  
    v_intensityR = _mm_add_ps(v_intensityR, v_specularN);  
    v_intensityG = _mm_add_ps(v_intensityG, v_specularN);  
    v_intensityB = _mm_add_ps(v_intensityB, v_specularN);  
} // compute the leftovers (use scalar C code)  
    for(; j < num_pnt; j++){  
        if (cosalpha > 0.0){ ... 7 more lines !!! ...
```

```
v_specularN[0:num_pnt] *= pow(cosalpha[0:num_pnt], phongconst);  
if (cosalpha[0:num_pnt] > 0.0){  
    specularN[0:num_pnt] = specular;  
    intensityR[0:num_pnt] += specularN[0:num_pnt];  
    intensityG[0:num_pnt] += specularN[0:num_pnt];  
    intensityB[0:num_pnt] += specularN[0:num_pnt];  
}
```

Cilk™ Plus



# Vector Width



Graph: James Reinders

# Additional Cilk Plus helpful feature: Elemental Functions

Use a function to describe the operation on a single element

`__declspec (vector)`

```
__declspec (vector) double option_price_call_black_scholes(
    double S,          // spot (underlying) price
    double K,          // strike (exercise) price,
    double r,          // interest rate
    double sigma,      // volatility
    double time)       // time to maturity
{
    double time_sqrt = sqrt(time);
    double d1 = (log(S/K)+r*time)/(sigma*time_sqrt)+0.5*sigma*time_sqrt;
    double d2 = d1-(sigma*time_sqrt);
    return S*N(d1) - K*exp(-r*time)*N(d2);
}
```

Invoke the function in a data parallel context

```
// invoke calculations for call-options
cilk_for (int i=0; i<NUM_OPTIONS; i++) {
    call[i] = option_price_call_black_scholes(S[i], K[i], r, sigma, time[i]);
}
```

The compiler generates vector version(s) of the function:  
Can yield a vector of results as fast as a single result.

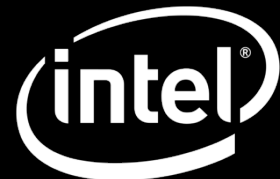
The secret sauce



# Options for using Elemental Functions

Construct	Example	Semantics
Standard for loop	<pre>for (j = 0; j &lt; N; j++) {     a[j] = my_ef(b[j]); }</pre>	Single thread, auto vectorization
#pragma simd	<pre>#pragma simd for (j = 0; j &lt; N; j++) {     a[j] = my_ef(b[j]); }</pre>	Single thread, Guaranteed to use the vector version
cilk for loop	<pre>cilk_for (j = 0; j &lt; N; j++) {     a[j] = my_ef(b[j]); }</pre>	Both vectorization and concurrent execution
Array notation	<pre>a[:] = my_ef(b[:]);</pre>	Vectorization. Concurrency allowed (but not yet implemented in compilers)

The execution of the elemental functions is serial with respect to the code that follows the invocation.





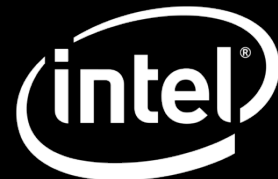
# #pragma SIMD: forcing “auto” vectorization

```
// vectorizable outer loop
```

```
#pragma simd
```

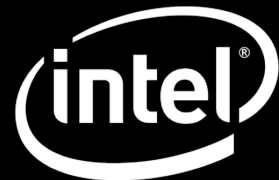
```
for (i=0; i<n; i++) {  
    complex<float> c = a[i];  
    complex<float> z = c;  
    int j = 0;  
    while ((j < 255)  
           && (abs(z)< limit))  
    {  
        z = z*z + c;  
        j++;  
    };  
    color[i] = j;  
}
```

- Combine standard C/C++ syntax with vector semantics.
- This program results in good utilization of vector level parallelism and provides measureable speedups.
- Arguably out of reach of auto vectorizers
- Outlining the loop body can be written as an elemental function. Yet, inline code is normally more efficient.



# TBB and Cilk Plus make a great combination

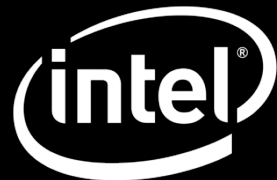
- Vector parallelism
  - Cilk Plus has two syntaxes for vector parallelism
    - Array Notation
    - `#pragma simd`
  - TBB relies on things outside TBB for vector parallelism.
    - TBB + `#pragma simd` is an attractive combination
- Thread parallelism
  - Cilk Plus is a strict fork-join language
    - Straitjacket enables strong guarantees about space.
  - TBB permits arbitrary task graphs
    - “Flexibility provides hanging rope.”



# Problem Statement

- Parallel Hardware
  - Scale
  - Vectorize
  - Specialization

Scale: GPUs, A/D, cameras, co-processors...

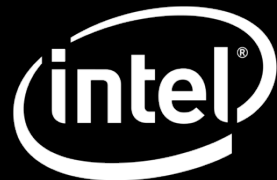


# Problem Statement

- Parallel Hardware
  - Scale
  - Vectorize
  - Specialization

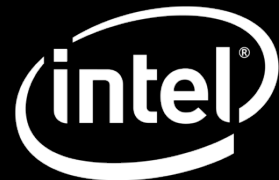
We know how to do “scale” and “vectorize” so let’s do that.

Tapping “specialization” is new, unproven and needs years of pain before we standardize.



# Moral of the story

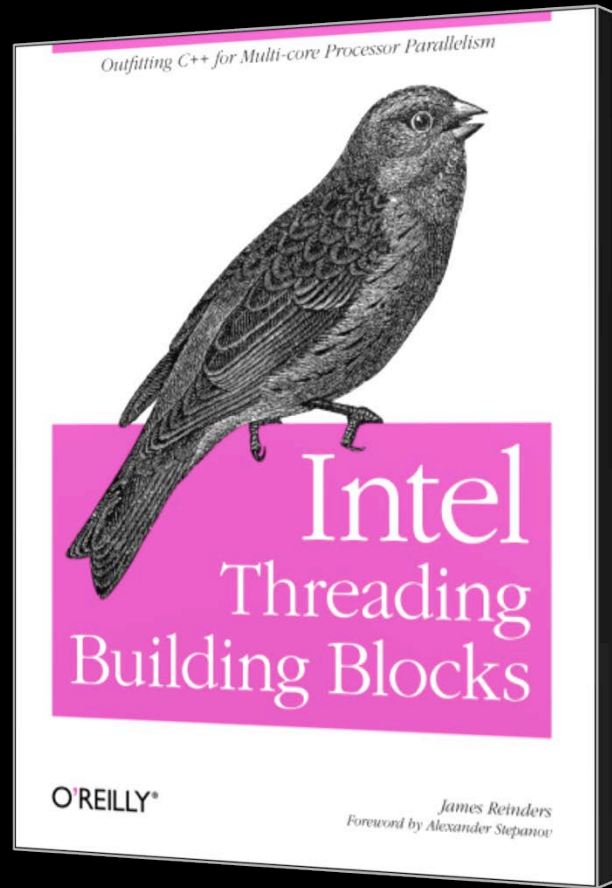
- Composability needs to be KING
- Simplicity is ESSENTIAL
- Harness parallelism in hardware by:
  - Scaling
  - Vectorization
  - Specialization



# Resources to help in quest to “Think Parallel”



# Structured Parallel Programming using TBB and Cilk™ Plus

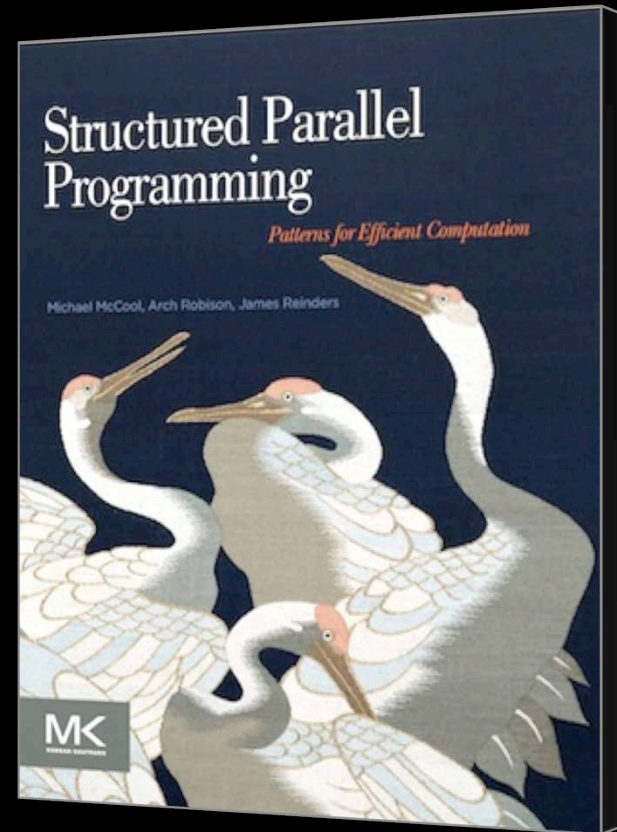


- Intel Threading Building Blocks (TBB)
- Most popular C++ parallel programming abstraction
- Book available in American English



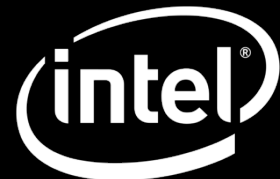
# Structured Parallel Programming using TBB and Cilk™ Plus

- Teaching *structured* parallel programming
- Designed for *programmers* not computer architects
- Teach *best methods* (known as patterns)



Coming:  
July 2012

[www.parallelbook.com](http://www.parallelbook.com)





# http://software.intel.com

http://software.intel.com/en-us/articles/vectorization-toolkit/



Intel® Software Network

[Communities](#)

[Partners](#)

[Tools & Downloads](#)

[Forums & Support](#)

[Blog](#)

[Resources](#)

[Home](#) > [Articles](#)

## Vectorization Toolkit

[Submit](#)

February 7, 2012 12:00 AM PST

Options:

### Future-Proof Your Application's Performance With Vectorization

The following toolkit gives you 6 Steps to Increase Performance Through Vectorization in Your Application. Click on the links to get more details on how to perform each step.

Want more context?

[View the webinar](#)

[Read a white paper](#)

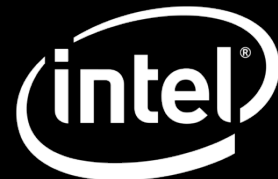
[Try some Sample code](#)

[Get some help](#)

#### The 6-Step Process for Vectorizing your Application

1. Measure Baseline [Release Build Performance](#)
2. Determine [Hotspots](#) Using Intel® VTune™ Amplifier XE
3. Determine Loop Candidates Using Intel Compiler [Vectorization Report](#)
4. Get Advice Using the Intel Compiler [GAP Report](#) and Toolkit Resources
5. Implement GAP Advice, Other Suggestions
6. Repeat!

(As needed until performance achieved or no good candidates left)



# intel.com/thinkparallel

Home > Intel® Academic Community Showcase

Welcome James Reinders (Intel) | Admin | Logout

## Intel® Academic Community Showcase

1 2 3 < >



### Meet the Academic Community at Supercomputing!

We will be holding an OpenMP workshop, educational discussions, and more!

[Check out our schedule >](#)

### ACTIVITY SPOTLIGHT

Learn how Academic Community members are advancing parallel programming education for undergraduates



#### Professor Pacheco University of San Francisco

Shares some best practices and materials to teach parallelism to lower division undergrads

[Learn More](#)

### NEWS FEATURES

#### Keys to Architecting Parallel Software with Patterns



UC Berkeley Professor Kurt Keutzer discusses the keys to architecting parallel software with design patterns in this modular video--complete with an introduction, structural patterns, computational patterns, and examples!

#### IDF 2011: Share the Excitement with your Students



It isn't too late to give your students a glimpse into tomorrow's technologies! See the presentations and videos that recap Intel's biggest tech event this year.

#### Microgrant Winners Present an Array of Teaching Techniques



Congratulations to the first winners of Intel Microgrants! Learn how Intel is enabling these professors to advance parallelism in undergraduate education. See how you can get involved!

#### Cluster Build-outs at Supercomputing 2011

Come to the LittleFe build-outs at Supercomputing 2011! LittleFe is a mini-cluster for teaching parallel programming.



#### Get Intel Academic Community Benefits

- Courseware
- Software Tools
- Hardware Access
- Intel Grants

[Learn How >](#)



### TECH BRIEFS

Updates from the Intel® Manycore Testing Lab

#### Using OpenMP to Scale Algorithms



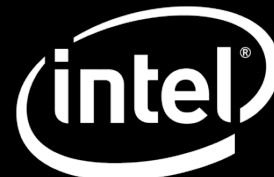
Yekaterina Zhmud and Evgeny Gorkunov (Novosibirsk State University) used the Intel Manycore Testing Lab test the scalability of an algorithm that finds automorphism

### LATEST TWITTER POSTS

It's Film Friday! Intel's Ron Green gives a demonstration of Intel Parallel Studio XE at SC11. [intel.ly/tHPjEf](#)  
9 Dec 11 | reply | retweet | favorite

New Intel Academic Showcase-- see how professors can work with Intel on parallel programming! [software.intel.com/en-us/articles/i...](#)  
12 Oct 11 | reply | retweet | favorite

Which #Intel parallel programming tool is best for



- Use more Fortran

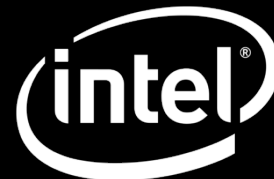
and

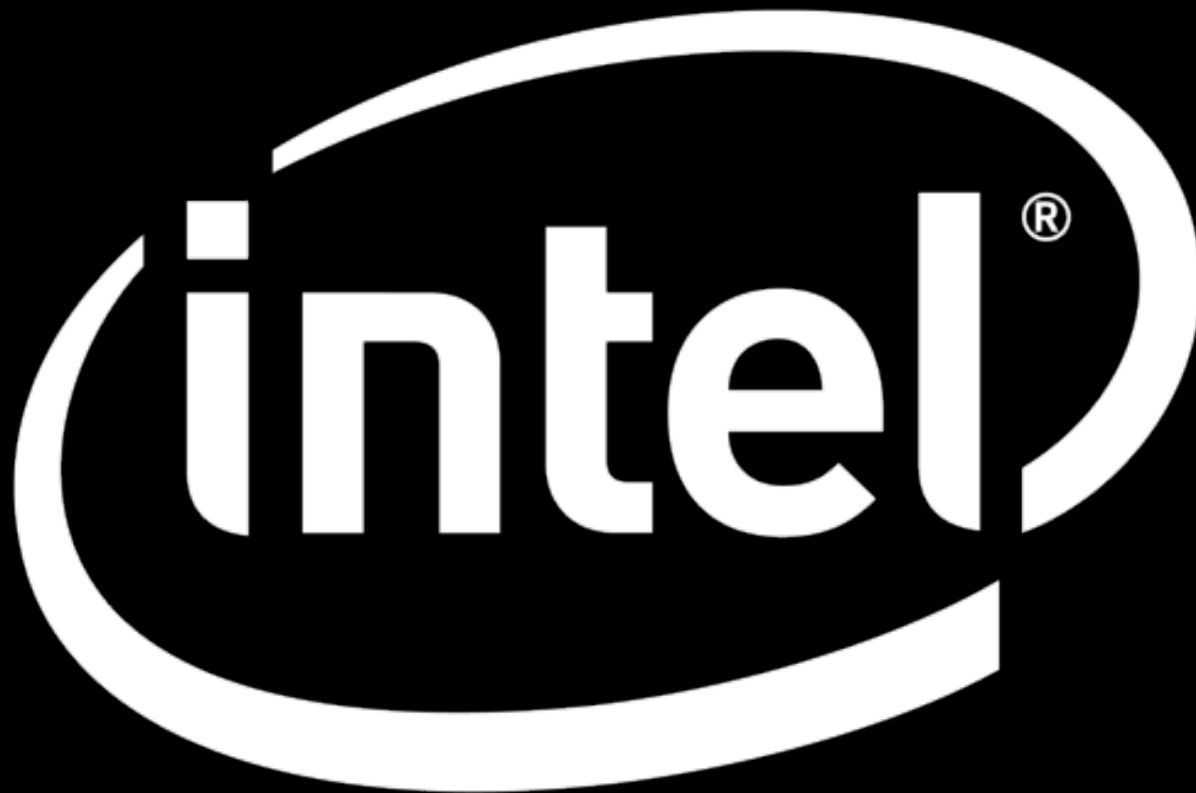
- Let's work together and fix up C and C++ for the 21<sup>st</sup> Century

## Moral of the story

- Composability needs to be KING
- Simplicity is ESSENTIAL
- Harness parallelism in hardware by:
  - Scaling
  - Vectorization
  - Specialization

[intel.com/go/parallel](http://intel.com/go/parallel)





# Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, reference [www.intel.com/software/products](http://www.intel.com/software/products).

Copyright ©, Intel Corporation. All rights reserved. Intel, the Intel logo, Xeon, Core, VTune, and Cilk are trademarks of Intel Corporation in the U.S. and other countries. \*Other names and brands may be claimed as the property of others.

## Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

