N3361=12-0051
Presented to EWG 8 Feb 2012

C++ Language Constructs for Parallel Programming

Pablo Halpern pablo.g.halpern@intel.com
Stefanus Du Toit stefanus.du.toit@intel.com
Clark Nelson clark.nelson@intel.com
Robert Geva robert.geva@intel.com

Goals

- To solicit interest in adding language constructs for parallelism to C++.
- To present the concepts in Intel® Cilk™ Plus as a basis for potential changes to C++.
- To create a common understanding of the factors that should influence the design of such constructs.

- **This presentation is not a proposal.**
  - Today we are presenting concepts
  - Next meeting we will bring proposals

Agenda

- About Parallelism
- Existing approaches
- An overview of Intel® Cilk™ Plus
- Desirable qualities of a parallel extension
- Implementation
- Conclusion

Agenda

- **About Parallelism**
  - Why parallelism?
  - Why parallelism constructs?
  - Task and data parallelism
- Existing approaches
- An overview of Intel® Cilk™ Plus
- Desirable qualities of a parallel extension
- Implementation
- Conclusion

Why Parallelism?

- Virtually all computers today contain multiple cores and vector instruction sets, and mobile devices are rapidly catching up.
- Many-core architectures such as Intel's MIC and modern GPUs are being tapped for computation.
- It is more power efficient to use multiple compute elements than to increase the clock rate of a single element.
- These developments will continue/accelerate
  - Transistor densities continue to increase
  - Mobile and data center both demand more speed with less power consumption
  - Expect 1000s of cores to become commonplace

Why Add Parallelism Constructs to C++?

- Parallel programming is **Hard!**
- Without standard support, concurrent programming often falls back on error-prone, ad-hoc protocols.
  - Similar to parameter-passing protocols before Fortran
- Programming directly with threads often leads to undesirable non-determinism[1]
- Threads and locks are not composable: Combining components introduces errors (e.g., deadlocks) or performance problems (e.g., resource contention).

Multicore and vector parallelism technologies have matured. It is time that we give C++ programmers access to them.

[1]Bocchino et. al., *Parallel Programming Must Be Deterministic by Default*

Task and data parallelism

- Task parallelism:
  - Performs separate operations in parallel
  - Takes advantage of multiple CPUs and hardware threads.
- Data parallelism:
  - Performs essentially the same operation on multiple data elements in parallel
  - Takes advantage of all HW resources: vector units and GP GPUs as well as multiple CPUs.
- Other parallelism (not explored in this presentation)
  - Coordination languages
  - Distributed parallelism
  - Parallel workloads

Threads and Tasks

- **Threads** are used for *coarse-grained* ***concurrency***
  - Concurrency is *mandated* – forward-progress is expected on all (non-blocked) threads.
  - Expensive to create – usually long-lived
  - Best for user interfaces, independent workloads (e.g., web server sessions), client-server workloads, etc.
- **Tasks** are used for *fine-grained* ***parallelism***
  - Concurrency is *allowed* but not mandated – only one task at a time is typically required to make forward progress
  - Inexpensive to create – can be very short-lived
  - Best for parallelizing an algorithm to take advantage of available parallel hardware resources.

Agenda

- About Parallelism
- **Existing approaches**
- An overview of Intel® Cilk™ Plus
- Desirable qualities of a parallel extension
- Implementation
- Conclusion

Existing approaches to parallelism in C++

- OpenMP*
- Intel® Threading Building Blocks™ (Intel® TBB) and Microsoft Parallel Patterns Library (PPL)
- std::thread, std::async, std::future
- Auto parallelization
- CUDA* and OpenCL*

- **Intel® Cilk™ Plus**
  - Intel believes that Cilk Plus can be the basis for a set of standard language features for parallelism in C++
  - Based on 15+ years of research (MIT, CMU, GA Tech)
  - Has both task and data parallel constructs
  - Well structured, composable, and has serial semantics

Agenda

- About Parallelism
- Existing approaches
- **An overview of Intel® Cilk™ Plus**
  - Task-parallel features
  - Data-parallel features
- Desirable qualities of a parallel extension
- Implementation
- Conclusion

Intel® Cilk Plus

Parallel tasks	• Easy to learn: 3 keywords (C & C++) • Tasks, not threads • Load balancing
Hyper Objects	• Mitigate data races on non-local variables
Array notations	• Data-parallel array operations • Targets SIMD, GPU
Elemental Functions	• Data-parallel function mapping
SIMD Annotation	• Vectorization annotation for loops • Currently expressed as a pragma

AO bench: Ambient Occlusion Renderer

- Small program for benchmarking real-world floating point performance.
- Case study for combining task and data parallelism in Cilk Plus.
- Parallelized in 1 day using Intel® Cilk™ Plus

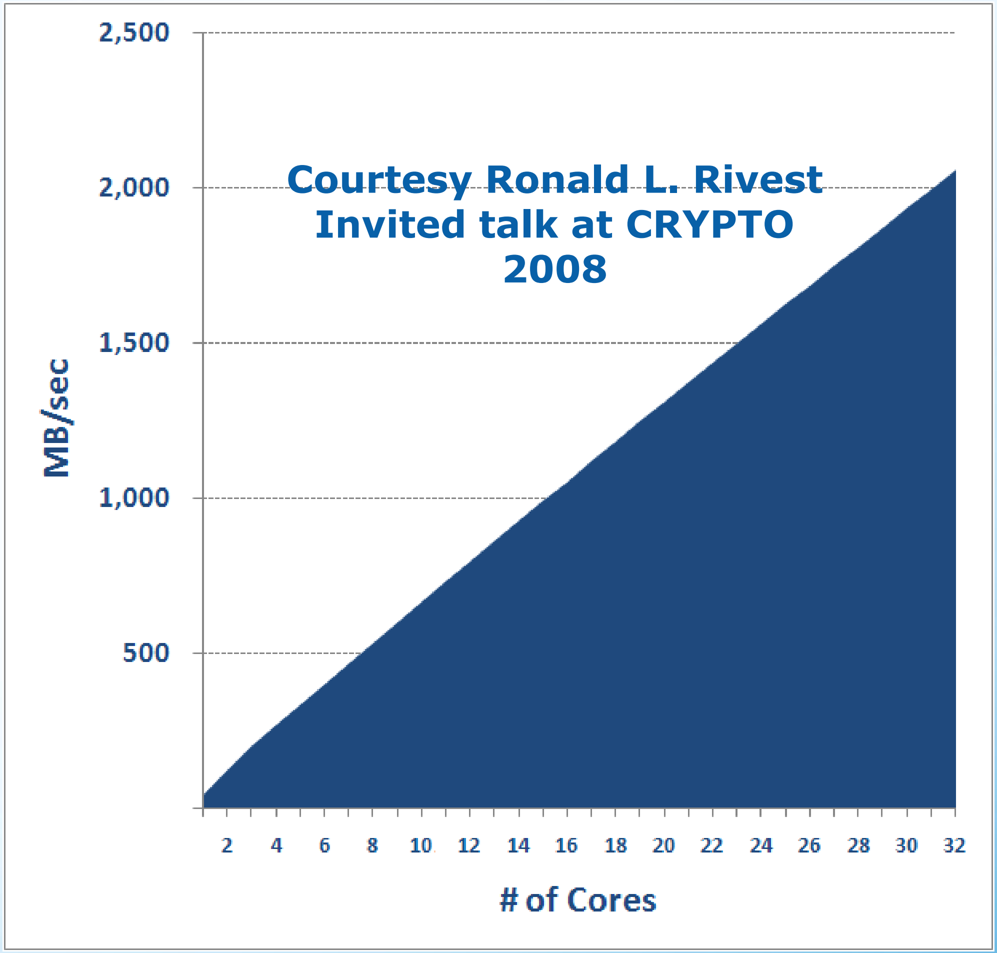
AObench Video	
Serial 0.67 FPS 1.00 X	Array Notation 1.42 FPS 2.10X
Intel Cilk tasks 3.30 FPS 4.90 X	Intel Cilk tasks + Array Notation 6.89 FPS 10.25 X

3.2GHz Intel® architecture code-name Nehalem (8 hyperthreads)
32-bit Win 7, 3GB RAM

http://software.intel.com/en-us/articles/data-and-thread-parallelism/

MD6: Cryptographic Hash Function

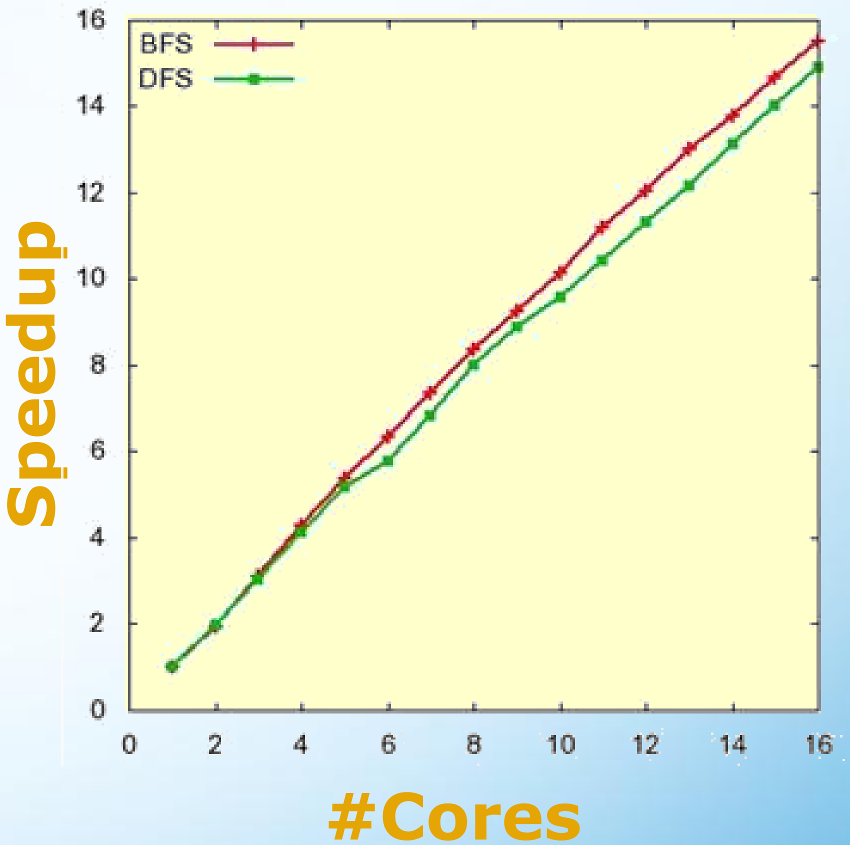
- Cryptographic hash function submitted to NIST competition
- Leverages large memory and multicore CPU’s
- Multicore-enabled in 1 day (Cilk Arts Cilk++ SDK)
- HP DL785
  - 8 Quad-Core Opterons
  - 1.1 GHz
  - 16 GB RAM



http://software.intel.com/en-us/articles/intel-cilk-sdk-resource-library/

Murphi: Model Checker

- Finite-state-machine verification tool developed at Stanford University.
- 3 programmer-months to ship a production-quality multicore-enabled version with Cilk Arts Cilk++ SDK.
- Employs parallel breadth-first and depth-first search algorithms on a sparse graph.



http://software.intel.com/en-us/articles/intel-cilk-sdk-resource-library/

Agenda

- About Parallelism
- Existing approaches
- **An overview of Intel® Cilk™ Plus**
  - **Task-parallel features**
    - `cilk_spawn`, `cilk_sync` and `cilk_for` keywords
    - hyperobjects
  - Data-parallel features
    - array notation
    - elemental functions
    - pragma simd
- Desirable qualities of a parallel extension
- Implementation
- Conclusion

Serial Tree Walk

```
#include <cilk/cilk.h>
int tree_walk(node *nodep)
{
    int a = 0, b = 0;
    if (nodep->left)
        a =            tree_walk(nodep->left);
    if (nodep->right)
        b =            tree_walk(nodep->right);
    int c = f(nodep->value);

    return a + b + c;
}
```

cilk_spawn and cilk_sync Keywords

```
#include <cilk/cilk.h>
int tree_walk(node *nodep)
{
    int a = 0, b = 0;
    if (nodep->left)
        a = cilk_spawn tree_walk(nodep->left);
    if (nodep->right)
        b = cilk_spawn tree_walk(nodep->right);
    int c = f(nodep->value);
    cilk_sync;
    return a + b + c;
}
```

Asynchronous recursive call to tree_walk

Call to f() can run in parallel with recursive tree walks

Implicit sync at the end of every function keeps code well structured

cilk_for Loop

```
cilk_for (int i = start; i < finish; i += stride)
    { /* Body of loop uses i */ }
f();
```

Iterations can execute in parallel.

All iterations complete before f() executes

```
cilk_for (iterator x = c.begin(); x != c.end(); ++x)
    { /* Body of loop uses *x */ }
f();
```

Random-access iterator

- A high-quality implementation will use dynamic load-balancing for unbalanced iterations.
- Iterations are independent -- compiler can apply data-parallel optimizations such as vectorization.

Spawning is not Thread Creation

- `cilk_spawn` gives the runtime *permission* to continue before the called function (child) returns.
  - Low cost (5x to 10x cost of a function call) No new threads
  - Code is *processor oblivious*: the number of cores is not specified.
  - If no available resources, then child executes serially.
  - A work-stealing scheduler (described later) may *steal* the parent and run it asynchronously.
- `cilk_for` gives the runtime *permission* to run iterations in parallel
- `cilk_sync` does not cause any thread to stall
  - A worker thread just finds other work to steal.
  - No global barrier is implied

Reducer Hyperobjects

- "Traditional" reduction on a parallel for loop:

```
long a[sz];
cilk::reducer_opadd<long> sum(0);
cilk_for (std::size_t i = 0; i < sz; ++i)
    sum += a[i];
```

Warning: reducer_opadd<float> would not be fully deterministic!

Parallel accesses each get their own "view" of sum

- Generalized reduction for any code executing in parallel:

```
cilk::reducer_list_append<int> lst(0);
void tree_walk2(node* nodep) {
  if (nodep->left) cilk_spawn tree_walk2(nodep->left);
  if (nodep->right) cilk_spawn tree_walk2(nodep->right);
  lst.push_back(f(nodep->value));
}
```

Final list has same order as for serial execution!

- You can define your own reducer types.

Agenda

- About Parallelism
- Existing approaches
- **An overview of Intel® Cilk™ Plus**
  - Task-parallel features
    - cilk_spawn, cilk_sync and cilk_for keywords
    - hyperobjects
  - **Data-parallel features**
    - array notation
    - elemental functions
    - pragma simd
- Desirable qualities of a parallel extension
- Implementation
- Conclusion

Array Notations

- Concise data-parallel notation encourages effective exploitation of SIMD, multi-core, and/or GPU
- The [:] operator delineates an *array section*:

  array-expression[*lower-bound* : *length* : *stride*]
- Each argument to [:] may be omitted:
  - Default *lower-bound* is 0
  - Default *length* is the length of the array (if known)
  - Default *stride* is 1 (second colon may be omitted)
- Array sections can be used with unary and binary operators for element-by-element computation:

```
a[10:count] = b[0:count] + c[0:count:2];
```

- Intrinsic functions operate on entire array sections

Array Notation Example

- Serial Example

```
float dot_product(unsigned int sz,
                  float A[], float B[]) {
    float dp=0.0f;
    for (int i=0; i<sz; i++)
         dp += A[i] * B[i];
    return dp;
}
```

- Array Notation Version

```
float dot_product(unsigned int sz,
                  float A[], float B[]) {
    return __sec_reduce_add(A[0:sz] * B[0:sz]);
}
```

Intrinsic reduction

Array Section

Element-wise multiplication

Rank and Shape

- No new type(s) for array sections
  - Type of an array section is just the element type
  - Additionally, an array section has rank and shape
- Rank: number of array-section operators on a single array
- Shape: vector of lengths of array sections
  - Conceptual, not concrete
  - Rank is the length of the shape vector

Expression	Rank	Shape
`a[0]`	0	
`a[0:n]`	1	n
`a[0][i:10]`	1	10
`a[i:n][j:m]`	2	n × m

Masked vector operations

- Array notation can be used within conditionals.
- A vectorizing compiler can generate a mask that allows vector computations based on the condition.

```
if (a[:] > b[:]) {        // Create a (logical) bit-mask, M
    c[:] = d[:] * e[:];   // For indexes where M contains 1
} else {
    c[:] = d[:] * 2;      // For indexes where M contains 0
}
```

Elemental Functions

- A general construct to express data parallelism:
  - Write a function to describe the operation on a single element
  - Invoke the function across a parallel data structure (arrays) or from within a vectorizable loop.
  - Implementation: A high-quality compiler vectorizes across consecutive invocations of the function
- Polymorphic: a vectorizing compiler may create both array and scalar versions of the function.
- Function parameters can be `varying`, `uniform`, `linear`
  - Allows mapping to the most efficient load/store available.
  - Allows optimization of address computations.
- Authoring the function is independent of its invocation
  - The function can invoked on scalars, within serial for or cilk_for loops, using array notation, etc..

Elemental Functions - Example

- Defining an elemental function:

```
__declspec (vector) double option_price_call_black_scholes(
    double S, double K, double r, double sigma, double time)
{
    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);
}
```

Compiler breaks data into SIMD vectors and calls function on each vector

- Invoking the elemental function:

```
// The following loop can also use cilk_for
call[0:N] = option_price_call_black_scholes(S[0:N], K[0:N], r,
                                            sigma, time[0:N]);
```

#pragma SIMD

- Loop annotation informs the compiler that vectorized loop will have same semantics as serial loop:

```
void f(float *a, const float *b, const int *e, int n)
{
#pragma simd
    for (int i = 0; i < n; ++i)
        a[i] = 2 * b[e[i]];
}
```

Potential aliasing and loop-carried dependencies would thwart auto-vectorization

- Currently implemented as a pragma, but other methods of annotating the loop can be considered.
- Additional clauses for reductions and other vectorization guidance (borrowed from OpenMP*)

Agenda

- About Parallelism
- Existing approaches
- An overview of Intel® Cilk™ Plus
- **Desirable qualities of a parallel extension**
  - Structured Parallelism, Determinism & Serial Semantics
  - Why a language extension instead of a library-only approach?
- Implementation
- Conclusion

Desirable qualities of a parallel extension

- Minimal changes to the existing language
- Efficient exploitation of all forms of mainstream hardware parallelism
- Hardware independent and scalable to future hardware (e.g. more cores & wider vector units).
- Composable (parallelism in library components does not introduce errors or performance issues)
- **Support for building programs that are easy to reason about**
  - Clean, expressive, syntax
  - Serial semantics
  - Deterministic results

Structured Parallelism and Determinism

- The biggest challenge in writing correct programs is making it possible to reason about your program.
- Experience has shown that *structure* is an important quality for parallel constructs that can be reasoned about. (Compare loops vs. goto.)
- Ideally, a parallel program would be as easy to reason about as a serial program.
- A good parallelization model abstracts away as much non-determinism as possible.
  - Language constructs should favor determinism
  - Non-determinism should be available for when the need arises.

Benefits of Structured Parallelism

- Composability
  - Given a call to f(), the caller doesn't need to know whether f() uses cilk_spawn or array operations.
  - Scalable to large codebases
- Tools
  - The **Cilkscreen** race detector is guaranteed to find races in any ostensibly deterministic Cilk Plus program.
  - The **Cilkview** scalability analyzer can determine how your program will scale to more cores.
- Optimization
  - The compiler can determine invariants only when the parallel constructs are fully structured

Serial Semantics

- Definition: a deterministic (race-free) program has the same semantics when running on one HW thread or many HW threads
- Dramatically simplify the task of reasoning about the program logic.
- Simplify the task of converting a serial program to a parallel program
- Allow serial debugging separate from parallel debugging.
- Allow tools and debuggers to discover many properties of a program – even parallel properties – by running the program serially (i.e., on one core)

Why a language extension instead of a library-only approach?

- Serial semantics: Structure is enforced by the compiler.
- Simpler syntax: A few keywords and operators can take the place of a large number of templates
- C compatibility: we want to propose similar extensions to C and C++.
- Implementation options: Difficult to implement lazy task creation using a library approach.
- Optimization: Huge opportunities for the compiler to apply algorithms and heuristics to the parallel code, e.g., much more effective vectorization.

Agenda

- About Parallelism
- Existing approaches
- An overview of Intel® Cilk™ Plus
- Desirable qualities of a parallel extension
- **Implementation**
  - Load balancing and work-stealing schedulers
  - Implementation experience
- Conclusion

Work Stealing

This Slide is not meaningful without animation

```
void f()
{
   cilk_spawn g();
   work
   work
   work
   cilk_sync;
   work
}
```

steal!

```
void g()
{
   work
   work
   work
}
```

Worker A

Worker B

Worker ?

Why Work Stealing?

- A work-stealing scheduler can be shown mathematically to be nearly optimal for a program with sufficient parallelism.
  - Gracefully handles control-flow and data divergence.
  - Used by most modern parallel programming systems
- Intel® Cilk™ Plus implements *lazy task creation*
  - Scheduler performs parent stealing, not child stealing
  - Serial semantics, even when using futures or the like.
  - Deterministic memory use
- Any C++ parallel extension should support (though not necessarily require) a work stealing scheduler that uses lazy task creation.

Intel® Cilk™ Plus Implementation Experience

- Current features available in Intel® compilers
  - For CPU, Many integrated cores (MIC), and integrated GPU
  - Run-time library is open source
- Partial implementation in GCC – ongoing
- At least three approaches have been used successfully for the work-stealing cactus stack
  - Heap-based (Cilk-5 from MIT, Cilk++ from Cilk Arts)
  - Multiple stacks (Intel® Cilk™ Plus) **Link Compatible!**
  - Per-core memory-mapped stacks (Cilk-M from MIT)
- Specification for Intel® Cilk™ Plus is available at: http://software.intel.com/en-us/articles/intel-cilk-plus-specification/

Agenda

- About Parallelism
- Existing approaches
- An overview of Intel® Cilk™ Plus
- Desirable qualities of a parallel extension
- Implementation
- **Conclusion**
  - Known challenges
  - Next Steps
  - References
  - Acknowledgements

Conclusions

“The Free Lunch is Over”

– Herb Sutter

“Parallel programming need not be just a collection of *ad hoc odd hacks*”

– Charles Leiserson

Conclusions

- Programmers will write parallel programs in C++, whether with hacks or non-standard extensions.
- The technologies for parallelization are much more mature than most people realize
  - Vectorization and work-stealing schedulers have been around for decades.
  - This used to be the province of super computers, but today a super computer fits in your pocket!
- Intel® Cilk™ Plus provides a fully-implemented starting point for standardization.

Next Steps

- Intel® Cilk™ Plus will become the basis for a set of proposals for C++ language constructs.
  - We are still evolving Intel Cilk Plus and are open to feedback
- We will not necessarily be proposing the entire Intel® Cilk™ Plus specification as a single proposal
  - Individual features may be proposed separately
  - Some features may evolve before being proposed
  - Some features may not be proposed
- Expect the first proposals at the October C++ Standards Meeting.

References

- Intel® Cilk™ Plus Specification
  http://software.intel.com/en-us/articles/intel-cilk-plus-specification/
- Edward Lee, *The Problem with Threads*
  http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.pdf
- Bocchino et. al., *Parallel Programming Must Be Deterministic by Default*
  http://www.usenix.org/event/hotpar09/tech/full_papers/bocchino/bocchino.pdf
- Mohr et. al., *Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs*
  http://dl.acm.org/citation.cfm?id=629042

Acknowledgements

- Thanks to the following people and working groups for help in drafting and reviewing this presentation:
  - Charles Leiserson
  - Arch Robison
  - The Cilk Plus runtime team at Intel
  - The Parallel Programming Models Working Group at Intel

Backup Slides

User-defined reducers

```
template <class T> struct multiply_monoid
  : public cilk::monoid_base<T>
{
    void identity(T* p) { ::new(p) T(1); }
    void reduce(T* left, T* right) {
        *left *= *right;
    }
};
...
cilk::reducer<multiply_monoid<double>> product(1.0);
cilk_for (std::size_t i = 0; i < sz; ++i)
    product() *= a[i];
```

Other Types of Hyperobjects

- Holders
  - Work like thread-local storage, but are tied to logical structure of parallel program, not to thread.
  - Can also be used to optimize logically-parallel computations that actually occur serially.
- Splitters
  - Allows parallel strands to read the same value, but keeps modifications separate
  - Implementation techniques are still an area of research

Array Notations → Vector Operations

- Selection of array elements
  - "vector" refers to a 1D array. Current implementation is does not allow [:] to be overloaded, e.g., for std::vector.

```
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
```

Known Challenges & Limitations

- Functions that spawn may return on a different thread than they were called on, causing problems with TLS.
- Array notation is currently limited to built-in arrays
  - No overloading for vector<>, array<>, etc.
- Array sections do not participate in the C++ type system; you cannot declare a variable of array-section type.
- #pragma is an ugly syntax for SIMD annotation
- Overlapping (i.e., partially redundant) functionality among array notation, cilk_for, and #pragma SIMD.

Optimization Notice

Optimization Notice

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

Legal Disclaimer

http://intel.com/software/products