Understanding Intel’s Parallel Programming Models

Michael McCool, Arch Robison, and James Reinders
SC11, Seattle, November 13, 2011
Course Outline

• Introduction
  – Motivation, goals, patterns

• Background
  – Machine model
  – Complexity, work-span

• Cilk™ Plus and Threading Building Blocks
  – Programming model
  – Examples

• Array Building Blocks
  – Programming model
  – Examples
Introduction

James Reinders
SC11, Seattle, November 13, 2011
Introduction: Outline

• Evolution of Hardware to Parallelism
• Software Engineering Considerations
• Structured Programming with Patterns
• Intel’s Parallel Programming Models
• Simple Example: Dot Product
Hardware Evolution: Transistors on a Chip

Processer Transistor Counts

source: James Reinders, Intel
Hardware Evolution: Clock Rate

Processor Clock Rates

source: James Reinders, Intel
There are limits to “automatic” improvement of scalar performance:

1. **The Power Wall:** Clock frequency cannot be increased without exceeding air cooling.

2. **The Memory Wall:** Access to data is a limiting factor.

3. **The ILP Wall:** All the existing instruction-level parallelism (ILP) is already being used.

→ **Conclusion:** Explicit parallel mechanisms and explicit parallel programming are required for performance scaling.
Hardware Parallelism: Cores and Threads

- Processor Core and Thread Counts

- Threads
- Cores

Source: James Reinders, Intel
Hardware Parallelism: Register Width

- Parallelism growth:
- Width of registers has also been growing dramatically over time.

Source: James Reinders, Intel
Parallel SW Engineering Considerations

• **Problem:** Amdahl’s Law* notes that scaling will be limited by the serial fraction of your program.

• **Solution:** scale the parallel part of your program faster than the serial part using data parallelism.

• **Problem:** Locking, access to data (memory and communication), and overhead will strangle scaling.

• **Solution:** use programming approaches with good data locality and low overhead, and avoid locks.

• **Problem:** Parallelism introduces new debugging challenges: deadlocks and race conditions.

• **Solution:** use structured programming strategies to avoid these by design, improving maintainability.

*Except Amdahl was an optimist, as we will discuss.*
Patterns

Michael McCool
SC11, Seattle, November 13, 2011
Structured Programming with Patterns

• Patterns are “best practices” for solving specific problems.
• Patterns can be used to organize your code, leading to algorithms that are more scalable and maintainable.
• A pattern supports a particular “algorithmic structure” with an efficient implementation.
• Intel’s programming models support a set of useful parallel patterns with low-overhead implementations.
Structured Serial Patterns

The following patterns are the basis of "structured programming" for serial computation:

- Sequence
- Selection
- Iteration
- Nesting
- Functions
- Recursion

- Random read
- Random write
- Stack allocation
- Heap allocation
- Objects
- Closures

Using these patterns, "goto" can (mostly) be eliminated and the maintainability of software improved.
Structured Parallel Patterns

The following additional parallel patterns can be used for “structured parallel programming”:

- Superscalar sequence
- Speculative selection
- Map
- Recurrence
- Scan
- Reduce
- Pack/expand
- Fork/join
- Pipeline
- Partition
- Segmentation
- Stencil
- Search/match
- Gather
- Merge scatter
- Priority scatter
- *Permutation scatter
- !Atomic scatter

Using these patterns, threads and vector intrinsics can (mostly) be eliminated and the maintainability of software improved.
Parallel Patterns: Overview
A serial sequence is executed in the exact order given:

\[
F = f(A) ; \\
G = g(F) ; \\
B = h(G) ;
\]
Superscalar Sequence

F = f(A);
G = g(F);
H = h(B,G);
R = r(G);
P = p(F);
Q = q(F);
S = s(H,R);
C = t(S,P,Q);

- Tasks ordered only by data dependencies
- Tasks can run whenever input data is ready

Developer writes “serial” code:
• **Map** replicates a function over every element of an index set.

• The index set may be abstract or associated with the elements of an array.

\[ A = \text{map}(f)(B); \]

• Map replaces *one specific* usage of iteration in serial programs: independent operations.

**Examples:** gamma correction and thresholding in images; color space conversions; Monte Carlo sampling; ray tracing.
**Reduction**

- **Reduce** combines every element in a collection into one element using an associative operator.

\[ b = \text{reduce}(f)(B) \]

- For example: reduce can be used to find the sum or maximum of an array.

**Examples:** averaging of Monte Carlo samples; convergence testing; image comparison metrics; matrix operations.
• **Scan** computes all partial reductions of a collection

\[ A = \text{scan}(f)(B); \]

• Operator must be (at least) associative.

• Diagram shows one possible parallel implementation using three-phase strategy

**Examples:** random number generation, pack, tabulated integration, time series analysis
Semantics and Implementation

**Semantics: What**
- The intended meaning as seen from the “outside”
- For example, for scan: compute all partial reductions given an associative operator

**Implementation: How**
- How it executes in practice, as seen from the “inside”
- For example, for scan: partition, serial reduction in each partition, scan of reductions, serial scan in each partition.
- Many implementations may be possible
- Parallelization may require reordering of operations
- Patterns should not over-constrain the ordering; only the important ordering constraints are specified in the semantics
- Patterns may also specify additional constraints, i.e. associativity of operators
Geometric Decomposition/Partition

- **Geometric decomposition** breaks an input collection into sub-collections
- **Partition** is a special case where sub-collections do not overlap
- Does not move data, it just provides an alternative “view” of its organization

**Examples:** JPG and other macroblock compression; divide-and-conquer matrix multiplication; coherency optimization for cone-beam recon.
• *Stencil* applies a function to neighbourhoods of an array
• Neighbourhoods are given by set of relative offsets
• Boundary conditions need to be considered

**Examples:** image filtering including convolution, median, anisotropic diffusion; simulation including fluid flow, electromagnetic, and financial PDE solvers, lattice QCD
Nesting: Recursive Composition
Fork-Join: Efficient Nesting

- Fork-join can be nested
- Spreads cost of work distribution and synchronization.
- This is how `cilk_for`, `tbb::parallel_for` and `arbb::map` are implemented.

Recursive fork-join enables high parallelism.
# Intel’s Parallel Programming Models

<table>
<thead>
<tr>
<th>Intel® Cilk™ Plus</th>
<th>Intel® Threading Building Blocks</th>
<th>Domain Specific Libraries</th>
<th>Established Standards</th>
<th>Research and Development</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/C++ language extensions to simplify parallelism</td>
<td>Widely used C++ template library for parallelism</td>
<td>Intel® Integrated Performance Primitives Intel® Math Kernel Library</td>
<td>Message Passing Interface (MPI) OpenMP* Coarray Fortran OpenCL*</td>
<td>Intel® Concurrent Collections Offload Extensions Intel® Array Building Blocks River Trail: parallel javascript Intel® SPMD Parallel Compiler</td>
</tr>
</tbody>
</table>

Open sourced Also an Intel product
Open sourced Also an Intel product

**Choice of high-performance parallel programming models**

- Libraries for pre-optimized and parallelized functionality
- Intel® Cilk™ Plus and Intel® Threading Building Blocks supports composable parallelization of a wide variety of applications.
- OpenCL* addresses the needs of customers in specific segments, and provides developers an additional choice to maximize their app performance
- MPI supports distributed computation, combines with other models on nodes
Intel’s Parallel Programming Models

- **Intel® Cilk™ Plus: Compiler extension**
  - Fork-join parallel programming model
  - Serial semantics if keywords ignored (serial elision)
  - Efficient work-stealing load balancing, hyperobjects
  - Supports vector parallelism via array slices and elemental functions

- **Intel® Threading Building Blocks (TBB): Library**
  - Template library for parallelism
  - Efficient work-stealing load balancing
  - Efficient low-level primitives (atomics, memory allocation).

- **Intel® Array Building Blocks (ArBB): Library and code generator**
  - A high-level data parallel programming model
  - Supports dynamic compilation and co-processor offload
  - Deterministic by default, safe by design
"Hello World": Dot Product; SSE Intrinsics

Plain C/C++

```c
float sprod(float *a,
            float *b,
            int size) {
    float sum = 0.0;
    for (int i=0; i < size; i++)
        sum += a[i] * b[i];
    return sum;
}
```

SSE

```c
float sprod(float *a,
            float *b,
            int size){
    _declspec(align(16))
    __m128 sum, prd, ma, mb;
    float tmp = 0.0;
    sum = _mm_setzero_ps();
    for(int i=0; i<size; i+=4){
        ma = _mm_load_ps(&a[i]);
        mb = _mm_load_ps(&b[i]);
        prd = _mm_mul_ps(ma,mb);
        sum = _mm_add_ps(prd,sum);
    }
    prd = _mm_setzero_ps();
    sum = _mm_hadd_ps(sum, prd);
    sum = _mm_hadd_ps(sum, prd);
    _mm_store_ss(&tmp, sum);
    return tmp;
}
```
Plain C/C++

```c
float sprod(float *a, float *b, int size) {
    float sum = 0.0;
    for (int i=0; i < size; i++)
        sum += a[i] * b[i];
    return sum;
}
```

Problems with SSE code:
- Machine dependent
  - Assumes vector length 4
- Verbose
- Hard to maintain
- Only vectorizes
  - SIMD instructions, no threads
- Example not even complete:
  - Array must be multiple of vector length

SSE

```c
float sprod(float *a, float *b, int size) {
    __declspec(align(16))
    __m128 sum, prd, ma, mb;
    float tmp = 0.0;
    sum = _mm_setzero_ps();
    for(int i=0; i<size; i+=4){
        ma = _mm_load_ps(&a[i]);
        mb = _mm_load_ps(&b[i]);
        prd = _mm_mul_ps(ma,mb);
        sum = _mm_add_ps(prd,sum);
    }
    prd = _mm_setzero_ps();
    sum = _mm_hadd_ps(sum, prd);
    sum = _mm_hadd_ps(sum, prd);
    _mm_store_ss(&tmp, sum);
    return tmp;
}
```
“Hello World”: in ArBB

Plain C/C++

```c
float sprod(float *a, 
            float *b, 
            int size) {
    float sum = 0;
    for (int i=0; i < size; i++)
        sum += a[i] * b[i];
    return sum;
}
```

ArBB

```c
void sprod(dense<f32> a, 
           dense<f32> b, 
           f32 &result) {
    result = sum(a * b);
}
```
“Hello World”: in ArBB + Glue Code

Plain C/C++

```c
float sprod(float *a,
            float *b,
            int size) {
    float sum = 0;
    for (int i=0; i < size; i++)
        sum += a[i] * b[i];
    return sum;
}
```

ArBB

```c
void sprod(dense<f32> a,
           dense<f32> b,
           f32 &result) {
    result = sum(a * b);
}
```

```c
// plus some glue code to get data in and out
float sprod(float *a,
            float *b,
            int size) {
    dense<f32> va(size), vb(size);
    memcpy(&va.write_only_range()[0],
           a, sizeof(float)*size);
    memcpy(&vb.write_only_range()[0],
           b, sizeof(float)*size);
    f32 res;
    call(sprod)(va, vb, res);
    return value(res);
}
**"Hello World": in Cilk™ Plus**

**Plain C/C++**

```c
float sprod(float *a,
            float *b,
            int size) {
    float sum = 0;
    for (int i=0; i < size; i++)
        sum += a[i] * b[i];
    return sum;
}
```

**Cilk™ Plus**

```c
float sprod(float a*,
            float b*,
            int size) {
    return __sec_reduce_add(
                 a[0:size] * b[0:size] );
}
```
“Hello World”: in Cilk™ Plus + Partitioning

Plain C/C++

```c
float sprod(float *a,  
    float *b,  
    int size) {
    float sum = 0;
    for (int i=0; i < size; i++)
        sum += a[i] * b[i];
    return sum;
}
```

Cilk™ Plus

```c
float sprod(float* a,  
    float* b,  
    int size) {
    int s = 4096;
    cilk::reducer_opadd<float> sum(0);
    cilk_for (int i=0; i<size; i+=s) {
        int m = std::min(s,size-i);
        sum += __sec_reduce_add(
            a[i:m] * b[i:m] );
    }
    return sum.get_value();
}
```
"Hello World": in TBB

Plain C/C++

```c
float sprod(float *a, 
            float *b, 
            int size) {
    float sum = 0;
    for (int i=0; i < size; i++)
        sum += a[i] * b[i];
    return sum;
}
```

TBB

```c
float sprod(const float a[],
            const float b[],
            size_t n ) {
    return tbb::parallel_reduce(
            tbb::blocked_range<size_t>(0,n),
            0.0f,
            [=](
                tbb::blocked_range<size_t>& r, 
                float in
            ) {
                return std::inner_product( 
                    a+r.begin(), a+r.end(),
                    b+r.begin(), in );
            },
            std::plus<float>()
    );
}
```
Patterns in Intel’s Parallel Programming Models

**Intel® Cilk™ Plus**

- `cilk_spawn`: nesting, fork-join
- Hyperobjects: reduce
- `cilk_for`, elemental functions: map
- Array notation: scatter, gather

**Intel® Threading Building Blocks**

- `parallel_invoke`, task-graph: nesting, fork-join
- `parallel_for`, `parallel_foreach`: map
- `parallel_do`: workpile (map + incr. task addition)
- `parallel_reduce`, `parallel_scan`: reduce, scan
- `parallel_pipeline`: pipeline

**Intel® Array Building Blocks**

- Elemental functions: map
- Collective operations: reduce, scan (predefined ops only)
- Permutation operations: pack, scatter, gather (incl shift), merge scatter
Conclusions

• **Explicit parallelism is a requirement for scaling**
  • Moore’s Law is still in force.
  • However, it is about transistor density, not scalar performance.

• **Patterns are a structured way to think about applications and programming models**
  • Useful for communicating and understanding structure
  • Useful for achieving a scalable implementation

• **Intel’s parallel programming models support scalable parallel patterns**
  • Parallelism, data locality, determinism
  • Low-overhead implementations
Machine Model

Michael McCool
SC11, Seattle, November 13, 2011
Course Outline

• Introduction
  – Motivation, goals, patterns

• **Background**
  – Machine model, complexity, work-span

• Cilk™ Plus and Threading Building Blocks
  – Programming model
  – Examples

• Array Building Blocks
  – Programming model
  – Examples

• Practical matters
  – Debugging and profiling
Background: Outline

• Machine model
  • Parallel hardware mechanisms
  • Memory architecture and hierarchy

• Speedup and efficiency

• DAG model of computation
  • Greedy scheduling

• Work-span parallel complexity model
  • Brent’s Lemma and Amdahl’s Law
  • Amdahl was an optimist: better bounds with work-span
  • Parallel slack
  • Potential vs. actual parallelism
What you (probably) want

Performance
- Compute results efficiently
- Improve absolute computation times over serial implementations

Portability
- Code that work well on a variety of machines without significant changes
- Scalable: make efficient use of more and more cores

Productivity
- Write code in a short period of time
- Debug, validate, and maintain it efficiently
Typical System Architecture
Cache Hierarchy
MIC Architecture
Nehalem

Integrated Memory Controller - 3 Ch DDR3

Core  Core  Core  Core

Quadrant  Quadrant  Quadrant  Quadrant

Shared L3 Cache
Westmere

Memory Controller

- Core
- Core
- Core
- Core
- Core
- Core

Shared L3 Cache**

Shared L3 Cache**
Key Factors

Compute: Parallelism
What mechanisms do processors provide for using parallelism?
- Implicit: instruction pipelines, superscalar issues
- Explicit: cores, hyperthreads, vector units

• How to map potential parallelism to actual parallelism?

Data: Locality
How is data managed and accessed, and what are the performance implications?
- Cache behavior, conflicts, sharing, coherency, (false) sharing; alignments with cache lines, pages, vector lanes

• How to design algorithms that have good data locality?
Pitfalls

Load imbalance
  – Too much work on some processors, too little on others

Overhead
  – Too little real work getting done, too much time spent managing the work

Deadlocks
  – Resource allocation loops causing lockup

Race conditions
  – Incorrect interleavings permitted, resulting in incorrect and non-deterministic results

Strangled scaling
  – Contended locks causing serialization
Array of structures (AoS) tends to cause cache alignment problems, and is hard to vectorize.

Structure of arrays (SoA) can be easily aligned to cache boundaries and is vectorizable.
Data Layout: Alignment

Array of Structures (AoS), padding at end.

Array of Structures (AoS), padding after each structure.

Structure of Arrays (SoA), padding at end.

Structure of Arrays (SoA), padding after each component.
Complexity Measures

Arch Robison
SC11, Seattle, November 13, 2011
**Speedup and Efficiency**

- $T_1 =$ time to run with 1 worker
- $T_P =$ time to run with $P$ workers
- $T_1/T_P = speedup$
  - The relative reduction in time to complete the same task
  - Ideal case is linear in $P$
    - i.e. 4 workers gives a best-case speedup of 4.
  - In real cases, speedup often significantly less
  - In rare cases, such as search, *can* be superlinear
- $T_1/(PT_P) = efficiency$
  - 1 is perfect efficiency
  - Like linear speedup, perfect efficiency is hard to achieve
  - Note that this is not the same as “utilization”
DAG Model of Computation

• Program is a directed acyclic graph (DAG) of tasks
• The hardware consists of workers
• Scheduling is *greedy*
  – No worker idles while there is a task available.
Departures from Greedy Scheduling

• Contended mutexes.
  – Blocked worker could be doing another task
    Avoid mutexes, use wait-free atomics instead.

• One linear stack per worker
  – Caller blocked until callee completes
    Intel® Cilk™ Plus has cactus stack.
    Intel® TBB uses continuation-passing style inside algorithm templates.
Work-Span Model

- $T_P = \text{time to run with } P \text{ workers}$
- $T_1 = \text{work}$
  - time for serial execution
  - sum of all work
- $T_\infty = \text{span}$
  - time for critical path
Work-Span Example

\[ T_1 = \text{work} = 7 \]
\[ T_\infty = \text{span} = 5 \]
Burdened Span

• Includes extra cost for synchronization
• Often dominated by cache line transfers.
Lower Bound on Greedy Scheduling

Work-Span Limit

$$\max\left(\frac{T_1}{P}, T_\infty\right) \leq T_P$$
Upper Bound on Greedy Scheduling

Brent’s Lemma

\[ T_P \leq \frac{(T_1 - T_\infty)}{P} + T_\infty \]
Applying Brent’s Lemma to 2 Processors

\[ T_1 = 7 \]

\[ T_\infty = 5 \]

\[ T_2 \leq \frac{(T_1 - T_\infty)}{P} + T_\infty \]

\[ \leq \frac{(7 - 5)}{2} + 5 \]

\[ \leq 6 \]
Amdahl Was An Optimist

Amdahl’s Law

\[ \frac{T_{\text{serial}} + T_{\text{parallel}}}{P} \leq T_P \]

- Speedup vs. Number of Threads (P)
  - Blue line: Amdahl's Law
  - Red line: Work-Span Bound
  - Orange line: Brent's Lemma

*Other brands and names are the property of their respective owners.*
Estimating Running Time

• Scalability requires that $T_\infty$ be dominated by $T_1$.

$$T_P \approx \frac{T_1}{P} + T_\infty \quad \text{if } T_\infty \ll T_1$$

• Increasing work hurts parallel execution proportionately.
• The span impacts scalability, even for finite $P$. 
Parallel Slack

- Sufficient parallelism implies linear speedup.

\[ T_P \approx \frac{T_1}{T_P} \text{ if } \frac{T_1}{T_\infty} \gg P \]

Linear speedup \quad Parallel slack
Definitions for Asymptotic Notation

- \( T(N) = O(f(N)) \iff T(N) \leq c \cdot f(N) \) for some constant \( c \).
- \( T(N) = \Omega(f(N)) \iff T(N) \geq c \cdot f(N) \) for some constant \( c \).
- \( T(N) = \Theta(f(N)) \iff c_1 \cdot f(N) \leq T(N) \leq c_2 \cdot f(N) \) for some constants \( c_1 \) and \( c_2 \).

**Quiz:** If \( T_1(N) = O(N^2) \) and \( T_\infty(N) = O(N) \), then \( T_1/T_\infty = ? \)

a. \( O(N) \)
b. \( O(1) \)
c. \( O(1/N) \)
d. need more information
Amdahl vs. Gustafson-Baris

Amdahl

serial work

parallelizable work

Time

P=1
P=2
P=4
P=8
Amdahl vs. Gustafson-Baris

Gustafson-Baris

serial work

parallelizable work

Time

P=1

P=2

P=4

P=8
Optional Versus Mandatory Parallelism

• Task constructs in Intel® TBB and Cilk™ Plus grant permission for parallel execution, but do not mandate it.
  – Exception: TBB’s std::thread (a.k.a. tbb::tbb_thread)

• Optional parallelism is key to efficiency
  – You provide parallel slack (over decomposition).
  – Potential parallelism should be greater than physical parallelism.
  – PBB converts potential parallelism to actual parallelism as appropriate.

A task is an opportunity for parallelism
Reminder of Some Assumptions

• Memory bandwidth is not a limiting resource.
• There is no speculative work.
• The scheduler is greedy.
Intel® Cilk™ Plus and Intel® Threading Building Blocks (TBB)

Arch Robison
SC11, Seattle, November 13, 2011
Course Outline

• Introduction
  – Motivation, goals, patterns

• Background
  – Machine model, complexity, work-span

• Cilk™ Plus and Threading Building Blocks
  – Programming model
  – Examples

• Array Building Blocks
  – Programming model
  – Examples

• Practical matters
  – Debugging and profiling
Cilk™ Plus and TBB: Outline

• Feature summaries
• C++ review
• Map pattern
• Reduction pattern
• Fork-join pattern
• Example: polynomial multiplication
• Complexity analysis
• Pipeline pattern
Summary of Cilk™ Plus

**Thread Parallelism**
cilk_spawn
cilk_sync
cilk_for

**Reducers**
reducer
reducer_op{add, and, or, xor}
reducer_{{min, max}}{_index}
reducer_list
reducer_ostream
reducer_string

**Vector Parallelism**
array notation
#pragma simd
elemental functions
TBB 4.0 Components

Parallel Algorithms
- parallel_for
- parallel_for_each
- parallel_invoke
- parallel_do
- parallel_scan
- parallel_sort
- parallel_[deterministic]_reduce

Macro Dataflow
- parallel_pipeline
- tbb::flow::...

Concurrent Containers
- concurrent_hash_map
- concurrent_unordered_{map,set}
- concurrent_[bounded_]queue
- concurrent_priority_queue
- concurrent_vector

Task scheduler
- task_group, structured_task_group
- task
- task_scheduler_init
- task_scheduler_observer

Synchronization Primitives
- atomic, condition_variable
- [recursive_]mutex
- {spin,queuing,null} [rw]_mutex
- critical_section, reader_writer_lock

Threads
- std::thread

Macro Dataflow
- parallel_pipeline
- tbb::flow::...

Concurrent Containers
- concurrent_hash_map
- concurrent_unordered_{map,set}
- concurrent_[bounded_]queue
- concurrent_priority_queue
- concurrent_vector

Thread Local Storage
- combinable
- enumerable_thread_specific

Memory Allocation
- tbb_allocator
- zero_allocator
- cache_aligned_allocator
- scalable_allocator
C++ Review

“Give me six hours to chop down a tree and I will spend the first four sharpening the axe.”

- Abraham Lincoln
C++ Review: Half Open Interval

• STL specifies a sequence as a half-open interval \([first, last)\)
  - \(last - first\) == size of interval
  - \(first == last\) \(\iff\) empty interval

• If object \(x\) contains a sequence
  - \(x\).begin() points to first element.
  - \(x\).end() points to “one past last” element.

```cpp
void PrintContainerOfTypeX( const X& x ) {
    for( X::iterator i=x.begin(); i!=x.end(); ++i )
        cout << *i << endl;
}
```
C++ Review: Function Template

• Type-parameterized function.
  – Strongly typed.
  – Obey scope rules.
  – Actual arguments evaluated exactly once.
  – Not redundantly instantiated.

```cpp
template<typename T>
void swap( T& x, T& y ) {
    T z = x;
    x = y;
    y = z;
}
```

```cpp
void reverse( float* first, float* last ) {
    while( first<last-1 )
        swap( *first++, *--last );
}
```

Compiler instantiates template `swap` with `T=float`.

[first,last) define half-open interval

void reverse( float* first, float* last ) {
    while( first<last-1 )
        swap( *first++, *--last );
    }


Genericity of swap

```cpp
template<typename T>
void swap(T& x, T& y) {
    T z = x;   // Construct a copy
    x = y;
    y = z;    // Destroy z
}
```

Requirements for T

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>T(const T&amp;)</code></td>
<td>Copy constructor</td>
</tr>
<tr>
<td><code>void T::operator=(const T&amp;)</code></td>
<td>Assignment</td>
</tr>
<tr>
<td><code>~T()</code></td>
<td>Destructor</td>
</tr>
</tbody>
</table>
C++ Review: Template Class

• Type-parameterized class

```cpp
template<typename T, typename U>
class pair {
public:
    T first;
    U second;
    pair( const T& x, const U& y ) : first(x), second(y) {}
};

pair<string,int> x;
x.first = "abc";
x.second = 42;
```

Compiler instantiates template pair with T=string and U=int.
C++ Function Object

• Also called a “functor”
• Is object with member operator().

```cpp
class LinearOp {
    float a, b;
public:
    float operator() ( float x ) const {return a*x+b;}
    Linear( float a_, float b_ ) : a(a_), b(b_) {}
};

LinearOp f(2,5);
y = f(3);
```

Could write as
```
y = f.operator()(3);
```
Template Function + Functor = Flow Control

template<typename I, typename Func>
void ForEach( I lower, I upper, const Func& f ) {
    for( I i=lower; i<upper; ++i )
        f(i);
}

class Accumulate {
    float& acc;
    float* src;
public:
    Accumulate( float& acc_, float* src_ ) : acc(acc_), src(src_) {}  
    void operator()( int i ) const { acc += src[i];}
};

float Example() {
    float a[4] = {1,3,9,27};
    float sum = 0;
    ForEach( 0, 4, Accumulate(sum,a) );
    return sum;
}

Pass functor to template function. Functo becomes “body” of control flow “statement”.

Template function for iteration

Functor

Pass functor to template function. Functo becomes “body” of control flow “statement”.

Other brands and names are the property of their respective owners.

So Far

• Abstract control structure as a template function.
• Encapsulate block of code as a functor.
• Template function and functor can be arbitrarily complex.
Recap: Capturing Local Variables

• Local variables were captured via fields in the functor

```
class Accumulate {
    float& acc;
    float* src;
public:
    Accumulate( float& acc_, float* src_ ) : acc(acc_), src(src_) {}
    void operator()( int i ) const { acc += src[i]; }
};

float Example() {
    float a[4] = {1,3,9,27};
    float sum = 0;
    ForEach( 0, 4, Accumulate(sum,a) );
    return sum;
}
```

Field holds reference to `sum`.
Capture reference to `sum` in `acc`.
Use reference to `sum`.
Formal parameter `acc_` bound to local variable `sum`.

Array Can Be Captured as Pointer Value

Field for capturing `a` declared as a pointer.

```cpp
class Accumulate {
  float& acc;
  float* src;
public:
  Accumulate( float& acc_, float* src_ ) : acc(acc_), src(src_) {}
  void operator()( int i ) const { acc += src[i]; }
};

float Example() {
  float a[4] = {1,3,9,27};
  float sum = 0;
  ForEach( 0, 4, Accumulate(sum,a) );
  return sum;
}
```

`a` implicitly converts to pointer
An Easier Naming Scheme

• Name each field and parameter after the local variable that it captures.

```cpp
class Accumulate {
    float& sum;
    float* a;
public:
    Accumulate( float& sum_, float* a_ ) : sum(sum_), a(a_) {}
    void operator()( int i ) const { sum += a[i]; }
};

float Example() {
    float a[4] = {1,3,9,27};
    float sum = 0;
    ForEach( 0, 4, Accumulate(sum,a) );
    return sum;
}
```

This is tedious mechanical work. Can we make the compiler do it?
C++11 Lambda Expression

• Part of C++11
• Concise notation for functor-with-capture.
• Available in recent Intel, Microsoft, and GNU C++ compilers.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Intel Compiler Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux* OS</td>
<td>-std=c++0x</td>
</tr>
<tr>
<td>Mac* OS</td>
<td></td>
</tr>
<tr>
<td>Windows* OS</td>
<td>/Qstd:c++0x on by default</td>
</tr>
</tbody>
</table>
With Lambda Expression

```cpp
float Example() {
    float a[4] = {1,3,9,27};
    float sum = 0;
    ForEach( 0, 4, [&]( int i ) { sum += a[i]; } );
    return sum;
}
```

`[&]` introduces lambda expression that constructs instance of `functor`.

Parameter list and body for `functor::operator()`.

Compiler automatically defines custom `functor` type tailored to capture `sum` and `a`.

```cpp
class Accumulate {
    float& acc;
    float* src;
public:
    Accumulate( float& acc_, float* src_ ) : acc(acc_), src(src_) {}
    void operator()( int i ) const { acc += src[i]; }
};
```

`[&]` introduces lambda expression that constructs instance of `functor`.

`functor` type tailored to capture `sum` and `a`.

Compiler automatically defines custom `functor` type tailored to capture `sum` and `a`.
Lambda Syntax

\[
\text{[capture\_mode]} \ (\text{formal\_parameters}) \rightarrow \text{return\_type} \ \{\text{body}\}
\]

- `&` ⇒ by-reference
- `=` ⇒ by-value
- `[]` ⇒ no capture

Can omit if there are no parameters \textit{and} return type is implicit.

Can omit if return type is \texttt{void} or \texttt{code} is “return \texttt{expr};”

Examples

- `[](\text{float} \ x) \ \{ \text{sum}+=x; \}`
- `[]\{\text{return rand();}\}
- `[](\text{float} \ x) \ \{ \text{return } \*p++; \}`
- `[](\text{float} \ x, \text{float} \ y) -> \text{float} \ \{ \text{if}(x<y) \ \text{return} \ x; \ \text{else return} \ y; \}`

Not covered here: how to specify capture mode on a per-variable basis.
Note About Anonymous Types

- Lambda expression returns a functor with *anonymous type*.
  - Thus lambda expression is typically used only as argument to template function or with C++11 auto keyword.
  - Later we’ll see two other uses unique to Cilk Plus.

```cpp
template<typename F>
void Eval( const F& f ) {
    f();
}

void Example1() {
    Eval( []{printf("Hello, world\n");} );
}
```

Template deduces functor’s type instead of specifying it.

```cpp
void Example2() {
    auto f = []{printf("Hello, world\n");};
    f();
}
```

Expression `[]{...}` has anonymous type.

Compiler deduces type of `f` from right side expression.
Note on Cilk™ Plus Keywords

• Include `<cilk/cilk.h>` to get nice spellings

```
#include <cilk/cilk.h>

int main() {
    cilk_for( int i=0; i<10; ++i ) {
        cilk_spawn f();
        g();
        cilk_sync;
    }
}
```
Cilk™ Plus Elision Property

• Cilk program has corresponding serialization
  – Equivalent to executing program with single worker.
• Different ways to force serialization:
  – #include `<cilk/cilk_stub.h>` at top of source file
  
  In `<cilk/cilk_stub.h>`

  #define _Cilk_sync
  #define _Cilk_spawn
  #define _Cilk_for for

  – Command-line option
    – icc: `-cilk-serialize`
    – icl: `/Qcilk-serialize`

  – Visual Studio:
    – Properties → C/C++ → Language [Intel C++] → Replace Intel Cilk Plus Keywords with Serial Equivalent
Note on TBB Names

- Most public TBB names reside in namespace `tbb`

```cpp
#include "tbb/tbb.h"
using namespace tbb;
```

- C++11 names are in namespace `std`.

```cpp
#include "tbb/compat/condition_variable"
#include "tbb/compat/thread"
#include "tbb/compat/tuple"
```

- Microsoft PPL names can be injected from namespace `tbb` into namespace `Concurrency`.

```cpp
#include "tbb/compat/ppl.h"
```
Map Pattern

Intel® Cilk™ Plus

\[ a[0:n] = f(b[0:n]); \]

\texttt{#pragma simd}
\begin{verbatim}
for( int i=0; i<n; ++i )
a[i] = f(b[i]);
\end{verbatim}

\texttt{cilk_for( int i=0; i<n; ++i )}
\begin{verbatim}
a[i] = f(b[i]);
\end{verbatim}

Intel® TBB

\begin{verbatim}
parallel_for( 0, n, [&]( int i ) {
    a[i] = f(b[i]);
});
\end{verbatim}

\begin{verbatim}
parallel_for(
    blocked_range<int>(0,n),
    [&]( blocked_range<int> r ) {
        for( int i=r.begin(); i!=r.end(); ++i )
            a[i] = f(b[i]);
    });
\end{verbatim}
Map in Array Notation

- Lets you specify parallel intent
  - Give license to the compiler to vectorize

```c
// Set y[i] ← y[i] + a · x[i] for i ∈ [0..n)
void saxpy(float a, float x[], float y[], size_t n) {
    y[0:n] += a*x[0:n];
}
```
Array Section Notation

- Pointer or array
- `base[first:length:stride]`
- First index
- Number of elements (different than F90!)
- Optional stride

**Rules for section$_1$ op section$_2$**
- Elementwise application of $op$
- Also works for $func(section$_1$, section$_2$)$
- Sections must be the same length
- Scalar arguments implicitly extended
More Examples

- Rank 2 Example – Update $m \times n$ tile with corner $[i][j]$.

\[ Vx[i:m][j:n] += a \times (U[i:m][j+1:n] - U[i:m][j:n]); \]

- Function call

\[ \text{theta}[0:n] = \text{atan2}(y[0:n],1.0); \]

- Gather/scatter

\[ w[0:n] = x[i[0:n]]; \]
\[ y[i[0:n]] = z[0:n]; \]
Improvement on Fortran 90

- Compiler does *not* generate temporary arrays.
  - Would cause unpredictable space demands and performance issues.
  - Want abstraction with minimal penalty.
  - Partial overlap of left and right sides is undefined.

- Exact overlap still allowed for updates.
  - Just like for structures in C/C++.

x[0:n] = 2*x[1:n];  // Undefined – partial overlap*
x[0:n] = 2*x[0:n];  // Okay – exact overlap
x[0:n:2] = 2*x[1:n:2];  // Okay – interleaved

*unless n≤1.
template<typename T>
T* destructive_move( T* first, T* last, T* output ) {
    size_t n = last-first;
    []( T& in, T& out ){
        out = std::move(in);
        in.~T();
    } ( first[0:n], output[0:n] );
    return output+n;
}
#pragma simd

- Another way to specify vectorization
  - Ignorable by compilers that do not understand it.
  - Similar in style to OpenMP “#pragma parallel for”

```c
void saxpy( float a, float x[], float y[], size_t n ) {
  #pragma simd
  for( size_t i=0; i<n; ++i )
    y[i] += a*x[i];
}
```
Clauses for Trickier Cases

• `linear` clause for induction variables
• `private`, `firstprivate`, `lastprivate` à la OpenMP

```c
void zip( float *x, float *y, float *z, size_t n ) {
    #pragma simd linear(x,y,z:2)
    for( size_t i=0; i<n; ++i ) {
        *z++ = *x++;
        *z++ = *y++;
    }
}
```

z has step of 2 per iteration.
Elemental Functions

• Enables vectorization of separately compiled scalar callee.

**In file with definition.**

```c
__declspec(vector)
float add(float x, float y) {
    return x + y;
}
```

**In file with call site.**

```c
__declspec(vector) float add(float x, float y);

void saxpy( float a, float x[], float y[], size_t n ) {
    #pragma simd
    for( size_t i=0; i<n; ++i )
        y[i] = add(y[i], a*x[i]);
}
```
Final Comment on Array Notation and 
#pragma simd

• No magic – just does tedious bookkeeping.
• Use “structure of array” (SoA) instead of “array of structure” (AoS) to get SIMD benefit.
cilk_for

• A way to specify thread parallelism.

```c
void saxpy( float a, float x[], float y[], size_t n ) {
    cilk_for( size_t i=0; i<n; ++i )
        y[i] += a*x[i];
}
```
Syntax for `cilk_for`

- Has restrictions so that iteration space can be computed before executing loop.

\[
\text{cilk\_for( type index = expr; condition; incr ) body;}
\]

- Must be integral type or random access iterator

- `index relop limit` and `limit relop index` might be evaluated only once.

- `index += stride` or `index -= stride` or `++index` or `index++` or `--index` or `index--`

- Iterations must be okay to execute in parallel.
Controlling grainsize

• By default, cilk_for tiles the iteration space.
  – Thread executes entire tile
  – Avoids excessively fine-grained synchronization

• For severely unbalanced iterations, this might be suboptimal.
  – Use **pragma cilk grainsize** to specify size of a tile

```c
#pragma cilk grainsize = 1

cilk_for( int i=0; i<n; ++i )
a[i] = f(b[i]);
```
tbb::parallel_for

- Has several forms.

  Execute `functor(i)` for all \( i \in [\text{lower}, \text{upper}) \)

  \[
  \text{parallel}\_\text{for}( \text{lower}, \text{upper}, \text{functor} );
  \]

  Execute `functor(i)` for all \( i \in \{\text{lower}, \text{lower+stride}, \text{lower+2*stride}, ...\} \)

  \[
  \text{parallel}\_\text{for}( \text{lower}, \text{upper}, \text{stride}, \text{functor} );
  \]

  Execute `functor(subrange)` for all `subrange` in `range`

  \[
  \text{parallel}\_\text{for}( \text{range}, \text{functor} );
  \]
Range Form

template <typename Range, typename Body>
void parallel_for(const Range& r, const Body& b);

• Requirements for a Range type R:

  R(const R&)
  R::~R()  
  bool R::empty() const
  bool R::is_divisible() const
  R::R (R& r, split)

  Copy a range
  Destroy a range
  Is range empty?
  Can range be split?
  Split r into two subranges

• Enables parallel loop over any recursively divisible range. Library provides blocked_range, blocked_range2d, blocked_range3d

• Programmer can define new kinds of ranges

• Does not have to be dimensional!
2D Example

```c++
// serial
for( int i=0; i<m; ++i )
    for( int j=0; j<n; ++j )
        a[i][j] = f(b[i][j]);

parallel_for(
    blocked_range2d<int>(0,m,0,n),
    [&] blocked_range2d<int> r ) {
    for( int i=r.rows().begin(); i!=r.rows().end(); ++i )
        for( int j=r.rows().begin(); j!=r.cols().end(); ++j )
            a[i][j] = f(b[i][j]);
});
```

Does 2D tiling, hence better cache usage in some cases than nesting 1D parallel_for.
Optional *partitioner* Argument

Recurse all the way down *range*.

```cpp
tbb::parallel_for( range, functor, tbb::simple_partitioner() );
```

Choose recursion depth heuristically.

```cpp
tbb::parallel_for( range, functor, tbb::auto_partitioner() );
```

Replay with cache optimization.

```cpp
tbb::parallel_for( range, functor, affinity_partitioner );
```
Iteration ↔ Thread Affinity

- Big win for serial repetition of a parallel loop.
  - Numerical relaxation methods
  - Time-stepping marches

```c
affinity_partitioner ap;
...
for( t=0; ...; t++ )
  parallel_for(range, body, ap);
```
Map Recap

Intel® Cilk™ Plus

cilk_for( int i=0; i<n; ++i )
  a[i] = f(b[i]);

Thread parallelism

a[0:n] = f(b[0:n]);

#pragma simd
for( int i=0; i<n; ++i )
a[i] = f(b[i]);

Vector parallelism

Intel® TBB

parallel_for( 0, n, [&]( int i ) {
  a[i] = f(b[i]);
});

parallel_for(
  blocked_range<int>(0,n),
  [&]( blocked_range<int> r ) {
    for( int i=r.begin(); i!=r.end(); ++i )
      a[i] = f(b[i]);
  });
Reduction Pattern

Intel® Cilk™ Plus

```c
float sum = __sec_reduce_add(a[i:n]);

#pragma simd reduction(+:sum)
float sum=0;
for( int i=0; i<n; ++i )
    sum += a[i];

cilk::reducer_opadd<float> sum = 0;
cilk_for( int i=0; i<n; ++i )
    sum += a[i];
... = sum.get_value();
```

Intel® TBB

```c
enumerable_thread_specific<float> sum;
parallel_for( 0, n, [&]( int i ) {
    sum.local() += a[i];
});
... = sum.combine(std::plus<float>());

sum = parallel_reduce(
    blocked_range<int>(0,n),
    0.f,
    [](blocked_range<int> r, float s) -> float {
        for( int i=r.begin(); i!=r.end(); ++i )
            s += a[i];
        return s;
    },
    std::plus<float>()
);
```
Reduction in Array Notation

• Build-in reduction operation for common cases +, *, min, index of min, etc.
• User-defined reductions allowed too.

```c
float dot( float x[], float y[], size_t n ) {
    return __sec_reduce_add( x[0:n]*y[0:n] );
}
```

sum reduction

elementwise multiplication
Reduction with `#pragma simd`

- **reduction clause for reduction variable**

```c
float dot( float x[], float y[], size_t n ) {
    float sum = 0;
    #pragma simd reduction(+:sum)
    for( size_t i=0; i<n; ++i )
        sum += x[i]*y[i];
    return sum;
}
```

Indicates that loop performs + reduction with `sum`. 
Reducers in Cilk Plus

• Enable lock-free use of shared reduction locations
  – Work for any associative reduction operation.
  – Reducer does *not* have to be local variable, or even a variable!

```c
    cilk::reducer_opadd<float> sum = 0;
    ...
    cilk_for( size_t i=1; i<n; ++i )
        sum += f(i);
    ...
    = sum.get_value();
```

*Not lexically bound to a particular loop.*

Updates local view of `sum`.

Get global view of `sum`.
Reduction with `enumerable_thread_specific`

- Good when:
  - Operation is commutative
  - Operand type is big

```cpp
enumerable_thread_specific<BigMatrix> sum;
...
parallel_for( 0, n, [&]( int i ) {
    sum.local() += a[i];
});
...
= sum.combine(std::plus<BigMatrix>())
```

Get thread-local element of `sum`.

Container of thread-local elements.

Return reduction over thread-local elements.
Reduction with `parallel_reduce`

- **Good when**
  - Operation is non-commutative
  - Tiling is important for performance

```cpp
sum = parallel_reduce(
    blocked_range<int>(0,n),
    0.f,
    [&] (blocked_range<int> r, float s) -> float {
        for (int i=r.begin(); i!=r.end(); ++i )
            s += a[i];
        return s;
    },
    std::plus<float>()
);
```

- Identity value.
- Reduce subrange.
- Recursive range.
- Functor for combining subrange results.
- Initial value for reducing subrange r. Must be included!
How `parallel_reduce` works

Reduce subranges

Reduction tree for subrange results.

Chaining of subrange reductions.
Notes on `parallel_reduce`

- Optional `partitioner` argument can be used, same as for `parallel_for`
- There is another tiled form that avoids almost all copying overhead.
  - C++11 “move semantics” offer another way to avoid the overhead.
- `parallel_deterministic_reduce` always does tree reduction.
  - Generates deterministic reduction even for floating-point.
  - Requires specification of grainsize.
  - No partitioner allowed.
Using *parallel_deterministic_reduce*

```cpp
sum = parallel_deterministic_reduce (blocked_range<int>(0,n,10000),
  0.f,
  [&](blocked_range<int> r, T s) -> float {
    for( int i=r.begin(); i!=r.end(); ++i )
      s += a[i];
    return s;
  },
  std::plus<T>())
```

- **Changed name**
- **Added grainsize parameter.** (Default is 1)
Reduction Pattern

**Intel® Cilk™ Plus**

```cpp
cilk::reducer_opadd<float> sum = 0;
cilk_for( int i=0; i<n; ++i )
    sum += a[i];
... = sum.get_value();
```

**Intel® TBB**

```cpp
enumerable_thread_specific<float> sum;
parallel_for( 0, n, [&]( int i ) {
    sum.local() += a[i];
});
... = sum.combine(std::plus<float>());
```

**Thread parallelism**

```cpp
float sum = __sec_reduce_add(a[i:n]);
```

**Vector parallelism**

```cpp
#pragma simd reduction(+:sum)
float sum=0;
for( int i=0; i<n; ++i )
    sum += a[i];
```
Fork-Join Pattern

Intel® Cilk™ Plus

cilk_spawn a();
cilk_spawn b();
c();
cilk_sync();

Intel® TBB

parallel_invoke(a, b, c);

task_group g;
g.run(a);
g.run(b);
g.run_and_wait(c);
Fork-Join in Cilk Plus

- `spawn` = asynchronous function call

```
x = cilk_spawn f(*p++);
y = g(*p--);
cilk_sync;
z = x+y;
```

Arguments to spawned function evaluated before fork.

```
tmp=*p++;
x=f(tmp);
y = g(*p--);
z = x+y;
```
cilk_sync Has Function Scope

- Scope of **cilk_sync** is entire function.
- There is implicit **cilk_sync** at end of a function.

```c
void bar() {
    for (int i=0; i<3; ++i) {
        cilk_spawn f(i);
        if (i&1) cilk_sync;
    }
    // implicit cilk_sync
}
```

**Serial call/return property:**
All Cilk Plus parallelism created by a function completes before it returns.
Style Issue

// Bad Style
cilk_spawn f();
cilk_spawn g();
// nop
cilk_sync;

// Preferred style
cilk_spawn f();
g();
// nop
cilk_sync;
Spawning a Statement in Cilk Plus

• Just spawn a lambda expression.

```
cilk_spawn [&]{
    for( int i=0; i<n; ++i )
        a[i] = 0;
}
(...)

cilk_sync;
```
Fork-Join in TBB

Useful for $n$-way fork when $n$ is small constant.

\[
\text{parallelInvoke}( \text{functor}_1, \text{functor}_2, \ldots );
\]

\[
\text{taskGroup} \ g;
\]
\[
\ldots
\]
\[
\text{g.run}( \text{functor}_1 );
\]
\[
\ldots
\]
\[
\text{g.run}( \text{functor}_2 );
\]
\[
\ldots
\]
\[
\text{g.wait}();
\]

Useful for $n$-way fork when $n$ is large or run-time value.
Fine Point About Cancellation/Exceptions

```c
#include <cppmicroservices.h>

int main() {
    task_group g;
    g.run( functor_1 );
    g.run( functor_2 );
    functor_3();
    g.wait();
    return 0;
}
```

Even if `g.cancel()` is called, `functor_3` still always runs.

```c
#include <cppmicroservices.h>

int main() {
    task_group g;
    g.run( functor_1 );
    g.run( functor_2 );
    g.run_and_wait( functor_3 );
    return 0;
}
```

Optimized `run + wait`. 

*Other brands and names are the property of their respective owners.
Steal Child or Continuation?

```c
task_group g;
for( int i=0; i<n; ++i )
  g.run( f(i) );
g.wait();
```

```
for( int i=0; i<n; ++i )
  cilk_spawn f(i);
cilk_sync;
```

[Diagram showing the execution flow of task_group and for-loops with task spawning and synchronization.]
What Thread Continues After a Wait?

```c
a();
task_group g;
g.run( b() );
c();
g. wait();
d();
a();
cilk_spawn b();
c();
cilk_sync;
d();
```

Whichever thread arrives last continues execution.
Implications of Mechanisms for Stealing/Waiting

- **Overhead**
  - Cilk Plus makes unstolen tasks cheap.
  - But Cilk Plus requires compiler support.

- **Space**
  - Cilk Plus has strong space guarantee: $S_p \leq P \cdot S_1 + P \cdot K$
  - TBB relies on heuristic.

- **Time**
  - Cilk Plus uses greedy scheduling.
  - TBB is only approximately greedy.

- **Thread local storage**
  - Function containing cilk_spawn can return on a *different* thread than it was called on.
  - Use reducers, not thread local storage.
    - However, Cilk Plus run-time does guarantee that “top level” call returns on same thread.
Fork-Join: Nesting

- Fork-join can be nested
- Spreads cost of work distribution and synchronization.
- This is how `cilk_for` and `tbb::parallel_for` are implemented.

Recursive fork-join enables high parallelism.
Faking Fork-Join in Vector Parallelism

- Using array section to control “if”:

  ```c
  if( a[0:n] < b[0:n] )
      c[0:n] += 1;
  else
      c[0:n] -= 1;
  ```

  Can be implemented by executing both arms of if-else with masking.

- Using `#pragma simd` on loop with “if”:

  ```c
  #pragma simd
  for( int i=0; i<n; ++i )
      if( a[i]<b[i] )
          c[i] += 1;
      else
          c[i] -= 1;
  ```

  Each fork dilutes gains from vectorization.
Fork-Join Pattern

**Intel® Cilk™ Plus**

```c
cilk_spawn a();
cilk_spawn b();
c();
cilk_sync();
```

**Intel® TBB**

```c
parallel_invoke( a, b, c );
```

**Thread parallelism**

```c
task_group g;
g.run( a );
g.run( b );
g.run( c );
g.wait();
```

**Fake Fork-Join for Vector parallelism**

```c
if( x[0:n] < 0 )
    x[0:n] = -x[0:n];

#pragma simd
for( int i=0; i<n; ++i )
    if( x[i] < 0 )
        x[i] = -x[i];
```
Polynomial Multiplication

Example: $c = a \cdot b$

$$
\begin{array}{c}
\begin{array}{c}
\text{x}^2 + 2x + 3 \\
\text{x}^2 + 4x + 5 \\
\hline
\text{5x}^2 + 10x + 15 \\
\text{4x}^3 + 8x^2 + 12x \\
\text{4x}^4 + 2x^3 + 3x^2 \\
\hline
\text{4x}^4 + 6x^3 + 16x^2 + 22x + 15
\end{array}
\end{array}
$$

b

a

5x^2 + 10x + 15

4x^3 + 8x^2 + 12x

4x^4 + 2x^3 + 3x^2

4x^4 + 6x^3 + 16x^2 + 22x + 15

c
Storage Scheme for Coefficients

\[ b[2] \quad b[1] \quad b[0] \]
\[ a[2] \quad a[1] \quad a[0] \]
Vector Parallelism with Cilk Plus Array Notation

• More concise than serial version
• Highly parallel: $T_1/T_\infty = n^2/n = \Theta(n)$
• What’s not to like about it?

```c
void simple_mul( T c[], const T a[], const T b[], size_t n ) {
    c[0:2*n-1] = 0;
    for (size_t i=0; i<n; ++i)
        c[i:n] += a[i]*b[0:n];
}
```
Too Much Work!

<table>
<thead>
<tr>
<th>Method</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grade school method</td>
<td>$\Theta(n^2)$</td>
</tr>
<tr>
<td>Karatsuba</td>
<td>$\Theta(n^{1.5})$</td>
</tr>
<tr>
<td>FFT method</td>
<td>$\Theta(n \log n)$</td>
</tr>
</tbody>
</table>

However, the FFT approach has high constant factor. For $n$ about 32-1024, Karatsuba is a good choice.
Karatsuba Trick: Divide and Conquer

• Suppose polynomials a and b have degree n
  - let $K = x^{\lfloor n/2 \rfloor}$
    
    $a = a_1 K + a_0$
    $b = b_1 K + b_0$

• Compute:
  
  $t_0 = a_0 \cdot b_0$
  $t_1 = (a_0 + a_1) \cdot (b_0 + b_1)$
  $t_2 = a_1 \cdot b_1$

• Then
  
  $a \cdot b \equiv t_2 K^2 + (t_1 - t_0 - t_2) K + t_0$

Partition coefficients.

3 half-sized multiplications. Do these recursively.

Sum products, shifted by multiples of K.
void karatsuba( T c[], const T a[], const T b[], size_t n ) {
    if( n<=CutOff ) {
        simple_mul( c, a, b, n );
    } else {
        size_t m = n/2;
        karatsuba( c, a, b, m );               // \( t_0 = a_0 \times b_0 \)
        karatsuba( c+2*m, a+m, b+m, n-m );    // \( t_2 = a_1 \times b_1 \)
        temp_space<T> s(4*(n-m));
        T *a_=s.data(), *b_=a_+(n-m), *t=b_+(n-m);
        a_[0:m] = a[0:m]+a[m:m];               // \( a_\_ = (a_0+a_1) \)
        b_[0:m] = b[0:m]+b[m:m];               // \( b_\_ = (b_0+b_1) \)
        karatsuba( t, a_, b_, n-m );           // \( t_1 = (a_0+a_1) \times (b_0+b_1) \)
        t[0:2*m-1] -= c[0:2*m-1] + c[2*m:2*m-1]; // \( t = t_1-t_0-t_2 \)
        c[2*m-1] = 0;
        c[m:2*m-1] += t[0:2*m-1];              // \( c = t_2K^2+(t_1-t_0-t_2)K+t_0 \)
    }
}
Sub-Products Can Be Computed in Parallel

\[ t_0 = a_0 \times b_0 \]
\[ t_2 = a_1 \times b_1 \]
\[ t_1 = (a_0 + a_1) \cdot (b_0 + b_1) \]

\[ \mathbf{a} \cdot \mathbf{b} \equiv t_2K^2 + (t_1 - t_0 - t_2)K + t_0 \]
Multithreaded Karatsuba in Cilk Plus

```c
void karatsuba( T c[], const T a[], const T b[], size_t n ) {  
    if( n<=CutOff ) {  
        simple_mul( c, a, b, n );
    } else {  
        size_t m = n/2;  
        cilk_spawn karatsuba( c, a, b, m );  
        cilk_spawn karatsuba( c+2*m, a+m, b+m, n-m );
        temp_space<T> s(4*(n-m));  
        T *a_=s.data(), *b_=a_+(n-m), *t=b_+(n-m);  
        a_[0:m] = a[0:m]+a[m:m];  // a_ = (a_0+a_1)  
        b_[0:m] = b[0:m]+b[m:m];  // b_ = (b_0+b_1)  
        karatsuba( t, a_, b_, n-m );  // t_1 = (a_0+a_1) * (b_0+b_1)  
        cilk_sync;  
        t[0:2*m-1] -= c[0:2*m-1] + c[2*m:2*m-1];  // t = t_1-t_0-t_2  
        c[2*m-1] = 0;
        c[m:2*m-1] += t[0:2*m-1];  // c = t_2K^2+(t_1-t_0-t_2)K+t_0  
    }
}
```

Only change is insertion of Cilk Plus keywords.
Multithreaded Karatsuba in TBB (1 of 2)

```c
void karatsuba(T c[], const T a[], const T b[], size_t n) {
    if (n <= CutOff) {
        simple_mul(c, a, b, n);
    } else {
        size_t m = n / 2;
        temp_space<T> s(4*(n-m));
        T* t = s.data();
        tbb::parallel_invoke(
            ..., // t0 = a0 \times b0
            ..., // t2 = a1 \times b1
            ... // t1 = (a0+a1) \times (b0+b1)
        );
        // t = t1-t0-t2
        for (size_t j=0; j<2*m-1; ++j )
            t[j] += c[j] + c[2*m+j];
        // c = t2K^2+(t1-t0-t2)K+t0
        c[2*m-1] = 0;
        for (size_t j=0; j<2*m-1; ++j )
            c[m+j] += t[j];
    }
}
```

Declared `temp_space` where it can be used after `parallel_invoke`.

Three-way fork-join specified with `parallel_invoke`.

Explicit loops replace Array Notation.
void karatsuba(T c[], const T a[], const T b[], size_t n) {
    ...
    tbb::parallel_invoke(
        [&] {
            karatsuba(c, a, b, m);
        },
        [&] {
            karatsuba(c+2*m, a+m, b+m, n-m);  // t2 = a1 × b1
        },
        [&] {
            T *a_=t+2*(n-m), *b_=a_+(n-m);
            for(size_t j=0; j<m; ++j) {
                a_[j] = a[j]+a[m+j];            // a_ = (a0+a1)
                b_[j] = b[j]+b[m+j];            // b_ = (b0+b1)
            }
            karatsuba(t, a_, b_, n-m);        // t1 = (a0+a1) × (b0+b1)
        }
    );
    ...
}
Work-Span Analysis for Fork-Join

- Let $B \| C$ denote the fork-join composition of $A$ and $B$

\[
T_1(A \| B) = T_1(A) + T_1(B)
\]

\[
T_\infty(A \| B) = \max(T_\infty(A), T_\infty(B))
\]
Master Method

If equations have this form:
\[ T(N) = aT(N/b) + cN^d \]
\[ T(1) = e \]

Then solution is:
\[ T(N) = \Theta(N^{\log_b a}) \quad \text{if } \log_b a > d \]
\[ T(N) = \Theta(N^d \log N) \quad \text{if } \log_b a = d \]
\[ T(N) = \Theta(N^d) \quad \text{if } \log_b a < d \]
Work-Span Analysis for Karatsuba Routine

• Let N be number of coefficients

Equations for work
\[ T_1(N) = 3T_1(N/2) + cN \]
\[ T_1(1) = \Theta(1) \]

Equations for span
\[ T_\infty(N) = T_\infty(N/2) + cN \]
\[ T_\infty(1) = \Theta(1) \]

Equations almost identical, except for one coefficient.

Solutions
\[ T_1(N) = \Theta(N^{\log_2 3}) \]
\[ T_\infty(N) = \Theta(N) \]

\[ \text{speedup} = \frac{T_1(N)}{T_\infty(N)} = \Theta(N^{0.58 \ldots}) \]
Space Analysis for Karatsuba Routine

• Let N be number of coefficients

Equations for serial space
\[ S_1(N) = S_1(N/2) + cN \]
\[ S_1(1) = \Theta(1) \]

Equations for parallel space?
\[ S_\infty(N) \leq 3S_\infty(N/2) + cN \]
\[ S_\infty(1) = \Theta(1) \]

Solutions
\[ S_1(N) = \Theta(N) \]
\[ S_\infty(N) = O(N^{\log_2^3}) \]

But what about the space \( S_p \)?
Cilk Plus Bound for Karatsuba Routine

- Cilk Plus guarantees $S_p \leq P \cdot S_1 + P \cdot K$

\[
S_p(N) = P \cdot O(N) + P \cdot K
= O(P \cdot N)
\]

For small $P$, a big improvement on the bound $S_\infty(N) = \Theta(N^{\log_2 3})$
Reducers Revisited

Global variable

cilk::reducer_opadd<float> sum;

void f( int m ) {
    sum += m;
}

float g() {
    cilk_spawn f(1);
    f(2);
    cilk_sync;
    return sum.get_value();
}

Reducers enable safe use of global variables without locks.

New view!

Merge views

sum₁ += 1
sum₂ = 0
sum₂ += 2
sum₁ += sum₂
sum₁ += 1
sum₁ += 2
Pipeline Pattern

Intel® Cilk™ Plus
(special case)

S s;
reducer_consume<S,U> sink ( 
    &s, h
);
...
void Stage2( T x ) {
    sink.consume(g(x));
}
...
while( T x = f() )
    cilk_spawn Stage2(x);
cilk_sync;

Intel® TBB

parallel_pipeline ( 
    ntoken,
    make_filter<void,T>(
        filter::serial_in_order,
        [&]( flow_control & fc ) -> T{
            T item = f();
            if( !item ) fc.stop();
            return item;
        }
    ) &
    make_filter<T,U>(
        filter::parallel,
        g
    ) &
    make_filter<U,void>(
        filter:: serial_in_order,
        h
    )
);
Serial vs. Parallel Stages

Parallel stage is functional transform.

Serial stage has associated state.

```
make_filter<X,Y>(
    filter::parallel,
    []( X x ) -> Y {
        Y y = foo(x);
        return y;
    }
)
```

You must ensure that parallel invocations of `foo` are safe.

```
make_filter<X,Y>(
    filter::serial_in_order,
    []( X x ) -> Y {
        extern int count;
        ++count;
        Y y = bar(x);
        return y;
    }
)
```
In-Order vs. Out-of-Order Serial Stages

- Each in-order stage receives values in the order the previous in-order stage returns them.
Special Rule for Last Stage

```
make_filter<X,void>(
    filter::serial_in_order,
    [&]( X x ) {
        cout << x;
    }
)
```

“to” type is **void**
Special Rules for First Stage

```
make_filter<void,Y>(
    filter::serial_in_order,
    [&]( flow_control& fc ) -> Y {
        Y y;
        cin >> y;
        if( cin.fail() ) fc.stop();
        return y;
    }
)
```

“from” type is **void**

**serial_out_of_order** or **parallel** allowed too.

First stage receives special **flow_control** argument.
Composing Stages

• Compose stages with operator &

\[
\text{make\_filter}\langle X,Y\rangle(\text{...}) \end{aligned} 
\begin{aligned}
\text{& make\_filter}\langle Y,Z\rangle(\text{...})
\end{aligned}
\]

Type Algebra

\[
\text{make\_filter}\langle T,U\rangle(mode,functor) \rightarrow \text{filter\_t}\langle T,U\rangle
\]
\[
\text{filter\_t}\langle T,U\rangle \& \text{filter\_t}\langle U,V\rangle \rightarrow \text{filter\_t}\langle T,V\rangle
\]
Running the Pipeline

```c
parallel_pipeline(
    size_t ntoken, const filter_t<void,void>& filter );
```

- **Token limit.**
- **filter** must map `void`→`void`. 
Bzip2 with `parallel_pipeline`

- **Input stream**
- **Read block**
  - Run-length encoding
- **Burrows-Wheeler Transform**
  - Move To Front
  - Run-length Encoding
  - Huffman Coding
- **Checksum**
  - Bit-align
- **Write block**
- **Output stream**

- **input file ptr**
- **checksum bit position output file ptr**
Pipeline Hack in Cilk Plus

- General TBB-style pipeline not possible (yet)
- But 3-stage serial-parallel-serial is.

```
serial loop

y = cilk_spawn g(x)

s = h(s, y)
```

Problem: \( h \) is not always associative
Monoid via Deferral

- Element is *state* or *list*
  - *list* = \{y_0, y_1, ..., y_n\}
- Identity = \{
- Operation = \( \otimes \)
  - \( \text{list}_1 \otimes \text{list}_2 \rightarrow \text{list}_1 \text{ concat list}_2 \)
  - \( \text{state} \otimes \text{list} \rightarrow \text{state}' \)
  - ... \( \otimes \text{state}_2 \) disallowed

\[
\text{h} = h(s, y)
\]

- Declare the stage
  - `reducer_consume sink<S,Y>( &s, h );`
- Feed one item to it.
  - `sink.consume(y);`
All Three Stages

while (T x = f())
    
    cilkSpawn Stage2(x);
    cilkSync;

void Stage2(T x) {
    sink.consume(g(x));
}

S s;
reducerConsume<S,U> sink ( 
    &s, h
);
#include `<cilk/reducer.h>`
#include `<list>`

template<typename State, typename Item>
class reducer_consume {
   public:
      typedef void (*consumer_func)(State*, Item);
   private:
      struct View {...};
      struct Monoid: cilk::monoid_base<View> {...};
      cilk::reducer<Monoid> impl;
   public:
      reducer_consume( State* s, consumer_func f );
};
struct View {
    std::list<Item> items;
    bool is_leftmost;
    View( bool is_leftmost_=false ) : is_leftmost(is_leftmost_) {}
    ~View() {}
};
**reducer_consumer::Monoid**

```cpp
struct Monoid: cilk::monoid_base<View> {
    State* state;
    consumer_func func;
    void munch(const Item& item) const {
        func(state, item);
    }
    void reduce(View* left, View* right) const {
        assert(!right->is_leftmost);
        if(left->is_leftmost)
            while(!right->items.empty()) {
                munch(right->items.front());
                right->items.pop_front();
            }
        else
            left->items.splice(left->items.end(), right->items);
    }
    Monoid(State* s, consumer_func f) : state(s), func(f) {});
};
```

**default implementation for a monoid**

**define “left = left ⊗ right”**
template<typename State, typename Item>
class reducer_consume {
public:
    typedef void (*consumer_func)(State*,Item);
private:
    struct View;
    struct Monoid;
    cilk::reducer<Monoid> impl;
public:
    reducer_consume( State* s, consumer_func f) : impl(Monoid(s,f), /*is_leftmost=*/true) {} 
    void consume( const Item& item ) {
        View& v = impl.view();
        if( v.is_leftmost )
            impl.monoid().munch( item );
        else
            v.items.push_back(item);
    }
};
Pipeline Pattern

Intel® Cilk™ Plus (special case)

S s;
reducer_consume<S,U> sink ( &s, h );
...
void Stage2( T x ) {
    sink.consume( g(x) );
}
...
while( T x = f() )
cilk_spawn Stage2(x);
cilk_sync;

Thread parallelism

Intel® TBB

parallel_pipeline ( ntoken,
    make_filter<void,T>(
        filter::serial_in_order,
        [&]( flow_control & fc ) -> T{
            T item = f();
            if( !item ) fc.stop();
            return item;
        }
    ) &
    make_filter<T,U>(
        filter::parallel,
        g
    ) &
    make_filter<U,void>(
        filter:: serial_in_order,
        h
    )
);
Summary(1)

• Cilk Plus and TBB are similar at an abstract level.
  – Both enable parallel pattern work-horses
    – Map, reduce, fork-join

• Details differ because of design assumptions
  – Cilk Plus is a language extension with compiler support.
  – TBB is a pure library approach.
  – Different syntaxes
Summary (2)

• Vector parallelism
  – Cilk Plus has two syntaxes for vector parallelism
    – Array Notation
    – \#pragma simd
  – TBB does not support 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.”
Intel® Array Building Blocks (ArBB)

Michael McCool
SC11, Seattle, November 13, 2011
Course Outline

• Introduction
  – Motivation, goals, patterns

• Background
  – Machine model, complexity, work-span

• Cilk™ Plus and Threading Building Blocks
  – Programming model
  – Examples

• **Array Building Blocks**
  – Programming model
  – Examples

• Practical matters
  – Debugging and profiling
Array Building Blocks: Outline

- Array Building Blocks
  - What it is
  - Architecture
  - Feature summary
- High-level ISO C++ Interface
- Examples:
  - (Introduction: Dot product)
  - Mandelbrot set
  - Matrix multiplication
  - Heat diffusion stencil
- Virtual Machine (VM) Interface
  - Example: Dot product
Intel® Array Building Blocks (ArBB)

Dynamic data parallel programming and a code generation infrastructure

• **A high-level parallel programming model**
  – Translates declarative, structured specifications of data-parallel computations into efficient parallel implementations
  – Takes advantage of both vector and thread parallel mechanisms

• **Based on staged dynamic compilation**
  – Embedded language in API can remove modularity overhead
  – Powerful support for runtime generic programming

• **Improves portability, productivity, and maintainability**
  – Avoids dependencies on particular hardware architectures
  – Integrates with existing C++ development tools
  – Race conditions and deadlocks are avoided by construction
ArBB Architecture

- Example C++ front-end available using only ISO standard C++ features
  - Other language interfaces possible
- ArBB VM has a standard compliant C89 interface
- VM supports portable
  - Thread/task management
  - Dynamically generation of optimized vector code
  - Shared and remote memory
- Code is portable across different SIMD widths and different core counts, even in binary form
Summary of ArBB

**Elemental Functions**
- `map(f)`
- Control flow
- Varying and uniform arguments

**Vector Parallelism**
- `call(f)`
- Arithmetic on containers
- Select
- Section
- Replace
- Repeat (row, column, ...)
- Control flow

**Data Management**
- Scalars
- Containers (dense collections)
- Homogeneous tuples (array)
- Heterogeneous structures (struct)

**Collective Operations**
- Reduce
- Scan

**Data Reorganization**
- Pack
- Unpack
- Shift
- Rotate
- Gather
- Scatter

**Code Management**
- `closure<func_type>`
- `auto_closure`
Containers, Tuples, and Structs

- Arrays of Structures (AoS) represented internally as Structures of Arrays (SoA) in order to support vectorization
Managing Data

• Data must be in an ArBB container for ArBB to operate on it

• Storage location managed by the runtime
  • may be remote, i.e. in local memory of a PCIe-based coprocessor

• Assignment always “as-if” done by value
  • Always read-before-write; shadow copies made if needed

• Elements of collections can be
  • ArBB types
  • User-defined structs or tuples of ArBB types

• Data layout optimized for vectorization
  • Arrays of Structures (AoS) represented internally as Structures of Arrays (SoA)
Expressing Computation

Vector Processing

Elemental Functions

dense<f32> A, B, C, D;
A += (B/C) * D;

void kernel(f32& a, f32 b, f32 c, f32 d) {
  a += (b/c)*d;
}
...
dense<f32> A, B, C, D;
map(kernel) (A, B, C, D);
Composing and Fusing Operations

Vector Processing

```c
void f(dense<f32>& e, dense<f32> a, dense<f32> b)
{
    dense<f32> c, d;
    c = a * b;
    map(k)(d, a, b);
    e = c + d;
}

dense<f32> E, A, B;
call(f)(E, A, B);
```

Elemental Functions

```c
void k(f32& c, f32 a, f32 b) {
    c = a / b;
}
```
Example: Mandelbrot Set

```c
int max_count = ...;
void mandel(i32& d, std::complex<f32> c) {
    i32 i;
    std::complex<f32> z = 0.0f;
    _for (i = 0, i < max_count, i++) {
        _if (abs(z) >= 2.0f) {
            _break;
        } _end_if;
        z = z*z + c;
    } _end_for;
    d = i;
}
void doit(dense<i32,2>& D, dense<std::complex<f32>,2> C) {
    map(mandel)(D,C);
} call(doit)(dest, pos);
```
Example: Matrix Multiplication – C

```c
for (int j = 0; j < n; ++j) {
    result[j] = matrix[j * n] * vector[0];
    for (int i = 1; i < m; ++i) {
        result[j] +=
            matrix[j * n + i] * vector[i];
    }
}
```
Example: Matrix Multiplication – ArBB

• Can eliminate loops completely in ArBB

    usize nrows = matrix.num_rows();
    result = add_reduce(
        matrix * repeat_row(vector, nrows)
    );
Example: Heat Dissipation Stencil

Heating

Cooling
Heat dissipation example (algorithm)

- **Data structure:**
  - 2D grid (N x M cells)
  - Boundary cells

- **Algorithm:**
  - Sweep over the grid
  - Update non-boundary cells
    - Read cells N, S, E, and W of the current cell
    - Take the average of the value
Heat dissipation example (C/C++)

```c
void run(double** grid1, double** grid2) {
    for (int iter = 0; iter < ITERATIONS; iter++) {
        step(grid1, grid2);
        tmp = grid1;
        grid1 = grid2;
        grid2 = tmp;
    }
}

void step(double** src, double** dst) {
    for (int i = 1; i < SIZE-1; i++) {
        for (int j = 1; j < SIZE-1; j++) {
            dst[i][j] = 0.25*(src[i+1][j] + src[i-1][j] +
                              src[i][j+1] + src[i][j-1]);
        }
    }
}
```

Run ITERATIONS sweeps over the 2D grid.

After each sweep, swap source and destination grid.

For each grid cell...

... apply stencil.
Heat dissipation example (ArBB) - I

```c
void stencil(f32& value, usize width, f32 boundary_value) {
    array<usize,2> i = position<2>();
    _if (all(1 <= i && i < width - 1)) {
        f32 north = neighbor(value, 0, -1);
        f32 south = neighbor(value, 0, +1);
        f32 west  = neighbor(value, -1, 0);
        f32 east  = neighbor(value, +1, 0);
        value = 0.25f * (north + south + west + east);
    }
}
```

- The stencil averages north, south, east, and west neighbors.
- Note:
  - The stencil uses a single parameter for both input and output
  - The ArBB runtime and memory manager take care of the shadow copy
void apply_stencil(
    usize niter, dense<f32,2>& grid, f32 bval
) {
    _for (i32 i = 0, i < niters, ++i) {
        map(stencil)(grid, grid.num_cols(), bval);
    }_end_for;
}

• An ArBB _for implements the iterative sweeps on the grid
• The map() operator applies the stencil for each solution cell
• Again:
  – The stencil uses a single parameter for both input and output
  – The ArBB runtime and memory manager take care of the shadow copy
ArBB Architecture

- Example C++ front-end available using *only* ISO standard C++ features
  - Other language interfaces possible
- ArBB VM has a standard compliant C89 interface
- VM supports portable
  - Thread/task management
  - Dynamically generation of optimized vector code
  - Shared and remote memory
- Code is **portable** across different SIMD widths and different core counts, *even in binary form*
Capture and staged compilation

**ArBB C++ headers**

**C++ source program**

**ISO C++ compiler**

**C++ object file**

**Capture/call**

**ArBB dynamic library**

**ArBB code generator**

**Optimized binary code for target (generated at run time)**

**Execution on host**

**Dynamic link**

**Captured ArBB closure (trace of ArBB operations)**

**Execution on target**

- Capture/call
- Captured ArBB closure (trace of ArBB operations)
- Optimized binary code for target (generated at run time)
- Execution on host
- Execution on target
Virtual Machine Functionality

• Data management
  – Declare new types
  – Allocate containers
  – Move/bind data between the application and the VM

• Function definition
  – Generate functions at runtime
  – Using sequences of both scalar and collective operations

• Execution management
  – Execute functions
  – Including remote and parallel execution
Virtual Machine Interface

• The VM interface is a C-only interface
  – C calling convention widespread
  – Almost all modern programming languages can call into C functions
  – Very generic applicability of the VM interface

• Requirements
  – No assumptions about host language
  – No assumptions about memory management
  – Easily bind as DLL or dynamic library
  – Fixed ABI, i.e., simple to update without recompilation

• API
  – All internal types are opaque, no data is exposed publicly
  – Consistent error handling through opaque error structure
    – No use of exceptions
    – Suitable for any language frontend
Example: Dot Product

```cpp
void arbb_sprod(dense<f32> a,
    dense<f32> b,
    f32& result) {
    result = add_reduce(a * b);
}
```

Structure of the code
- Get a handle to the ArBB VM
- Create the input/output types needed (scalars, dense container)
- Create a function called “dot”
  - Define the input and output parameters of “dot”
  - Create local variables
- Create a function called “dot”
  - Do a element-wise multiplication between the input containers
  - Do an add_reduce on the result of the multiplication
  - ... store the result in the output argument

• The next slides will show how this can be done with the VM API
Here’s the whole sequence. We’ll show it step by step.
ArBB VM code for dot product, I

```c
arbb_function_t generate_dot() {
    arbb_context_t context;
    arbb_get_default_context(&context, NULL);

    arbb_type_t base_type;
    arbb_get_scalar_type(context, &base_type, arbb_f32, NULL);

    arbb_type_t dense_1d_f32;
    arbb_get_dense_type(context, &dense_1d_f32, base_type, 1, NULL);

    arbb_type_t inputs[] = { dense_1d_f32, dense_1d_f32 };
    arbb_type_t outputs[] = { base_type };

    arbb_type_t fn_type;
    arbb_get_function_type(context, &fn_type, 1, outputs, 2, inputs, NULL);

    // continued on the next slide
}
```
ArBB VM code for dot product, II

```
arbb_function_t generate_dot() {
    // continued from previous slide

    arbb_function_t function;
    arbb_begin_function(context, &function, fn_type, "dot", 0, NULL);

    arbb_variable_t a, b, c;
    enum { is_input, is_output };
    arbb_get_parameter(function, &a, is_input, 0, NULL);
    arbb_get_parameter(function, &b, is_input, 1, NULL);
    arbb_get_parameter(function, &c, is_output, 0, NULL);

    arbb_variable_t tmp[1];
    arbb_create_local(function, tmp, dense_1d_f32, 0, NULL);

    // continued on the next slide
}
```
ArBB VM code for dot product, III

```c
arbb_function_t generate_dot() {
    // continued from previous slide

    // fixed number of arguments, use arbb_op
    arbb_variable_t in[] = { a, b };
    arbb_op(function, arbb_op_mul, tmp, in, 0, NULL);

    // variable number of arguments, use arbb_op_dynamic
    arbb_variable_t result[] = { c };
    arbb_op_dynamic(function, arbb_op_add_reduce,
        1, result, 1, tmp, 0, NULL);

    arbb_end_function(function, NULL);

    arbb_compile(function, NULL);

    return function;
}
```
Inspecting the generated “dot” function

Can output a representation of the generated code

```c
void print_generated_code() {
    arbb_function_t function = generate_dot();
    arbb_string_t serialized;
    arbb_serialize_function(function, &serialized, NULL);
    const char *cstring = arbb_get_c_string(serialized);
    printf("%s", cstring);
    arbb_free_string(serialized);
}
```

**Outputs:**

```c
function _dot(out $f32 _0, in dense<$f32> _1,
    in dense<$f32> _2) {
    _3 = mul<dense<$f32>>(_1, _2);
    _0 = add_reduce<$f32>(_3);
}
```
Summary

• Intel® Array Building Blocks is...
  – A powerful data-parallel code generator
  – A library, like TBB, but also a compiler, like Cilk™ Plus, that can generate vectorized machine language
  – Designed around a set of high-level deterministic patterns that avoid data races and deadlocks by construction
  – Parallelization and also eliminates modularity overhead of C++
  – Interesting new capabilities enabled by dynamic code gen

• Now available on WhatIf!
Conclusion

James Reinders
SC11, Seattle, November 13, 2011
## Intel’s Parallel Programming Models

<table>
<thead>
<tr>
<th>Intel® Cilk™ Plus</th>
<th>Intel® Threading Building Blocks</th>
<th>Domain Specific Libraries</th>
<th>Established Standards</th>
<th>Research and Development</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/C++ language extensions to simplify parallelism</td>
<td>Widely used C++ template library for parallelism</td>
<td>Intel® Integrated Performance Primitives Intel® Math Kernel Library</td>
<td>Message Passing Interface (MPI) OpenMP* Coarray Fortran OpenCL*</td>
<td>Intel® Concurrent Collections Offload Extensions Intel® Array Building Blocks River Trail: parallel javascript Intel® SPMD Parallel Compiler</td>
</tr>
</tbody>
</table>

### Choice of high-performance parallel programming models

- Libraries for pre-optimized and parallelized functionality
- Intel® Cilk™ Plus and Intel® Threading Building Blocks supports composable parallelization of a wide variety of applications.
- OpenCL* addresses the needs of customers in specific segments, and provides developers an additional choice to maximize their app performance
- MPI supports distributed computation, combines with other models on nodes
Other Parallel Programming Models

OpenMP
- Syntax based on pragmas and an API
- Works with C, C++, and Fortran

MPI
- Distributed computation, API based

Co-array Fortran
- Distributed computation, language extension

OpenCL
- Uses separate kernel language plus a control API
- Two-level memory and task decomposition hierarchy

CnC
- Coordination language based on a graph model
- Actual computation must be written in C or C++

ISPC
- Based on elemental functions, type system for uniform computations

River Trail
- Data-parallel extension to Javascript
Course Summary

• Have presented a subset of the parallel programming models available from Intel
  – Useful for writing efficient and scalable parallel programs
  – Presented Cilk Plus, Threading Building Blocks (TBB), and Array Building Blocks (ArBB)
  – There will be a workshop on Friday if you would like some hands-on experience with Cilk Plus and TBB

• Also presented structured approach to parallel programming based on patterns
  – With examples for some of the most important patterns

• A book, *Structured Parallel Programming: Patterns for Efficient Computation*, will soon be available that builds on the material in this course:
  http://parallelbook.com
# 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.
Legal Disclaimer

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.


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

Copyright © 2011. Intel Corporation.

http://intel.com/software/products
Details on ArBB
ArBB Data Types and Code Constructs

• Scalar types
  – Equivalent to primitive C++ types

• Vector types
  – Parallel collections of (scalar) data

• Operators
  – Scalar operators
  – Vector operators

• Functions
  – User-defined parameterizable code sequences
  – Closures (dynamically generated compiled code objects)

• Control flow
  – Conditionals and iteration (serial control flow)
  – Vector operations and “map” used for expressing parallelism
Scalar types

Scalar types provide equivalent functionality to the scalar types built into C/C++

<table>
<thead>
<tr>
<th>Types</th>
<th>Description</th>
<th>C++ equivalents</th>
</tr>
</thead>
<tbody>
<tr>
<td>f32, f64</td>
<td>32/64 bit floating point number</td>
<td>float, double</td>
</tr>
<tr>
<td>i8, i16, i32, i64</td>
<td>8/16/32 bit signed integers</td>
<td>char, short, int</td>
</tr>
<tr>
<td>u8, u16, u32, u64</td>
<td>8/16/32 bit unsigned integers</td>
<td>unsigned char/short/int</td>
</tr>
<tr>
<td>boolean</td>
<td>Boolean value (true or false)</td>
<td>bool</td>
</tr>
<tr>
<td>usize, isize</td>
<td>Signed/unsigned integers sufficiently large to store addresses</td>
<td>size_t (equiv. usize) ssize_t (equiv. isize)</td>
</tr>
</tbody>
</table>
Containers

template<typename T, std::size_t D = 1>
class dense;

• This is the equivalent to std::vector or C arrays
• Dimensionality is optional, defaults to 1

<table>
<thead>
<tr>
<th>Property</th>
<th>Restrictions</th>
<th>Can be set at</th>
</tr>
</thead>
<tbody>
<tr>
<td>Element type</td>
<td>Must be an ArBB scalar or user-defined type</td>
<td>Compile time</td>
</tr>
<tr>
<td>Dimensionality</td>
<td>1, 2, or 3</td>
<td>Compile time</td>
</tr>
<tr>
<td>Size</td>
<td>Only restricted by free memory</td>
<td>Runtime</td>
</tr>
</tbody>
</table>
### Declaration and Construction

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Element type</th>
<th>Dimensionality</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dense&lt;f32&gt;</code> a1;</td>
<td>f32</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td><code>dense&lt;f32, 1&gt;</code> a2;</td>
<td>f32</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td><code>dense&lt;i32, 2&gt;</code> b;</td>
<td>i32</td>
<td>2</td>
<td>0, 0</td>
</tr>
<tr>
<td><code>dense&lt;f32&gt;</code> c(1000);</td>
<td>f32</td>
<td>1</td>
<td>1000</td>
</tr>
<tr>
<td><code>dense&lt;f32&gt;</code> d(c);</td>
<td>f32</td>
<td>1</td>
<td>1000</td>
</tr>
<tr>
<td><code>dense&lt;i8, 3&gt;</code> e(5, 3, 2);</td>
<td>i8</td>
<td>3</td>
<td>5, 3, 2</td>
</tr>
</tbody>
</table>
Element-wise & Vector-scalar Operators

- **Arithmetic operators:**
  
  +, +=, ++ (prefix and postfix), addition, increment  
  -, -=, -- (prefix and postfix), subtraction, decrement  
  *, *=, multiplication  
  /, /=, division  
  %, %=, modulo

- **Bitwise operators:**
  
  &, &=, bitwise AND  
  |, |=, bitwise OR  
  ^, ^=, bitwise XOR  
  ~, ~=, bitwise NOT  
  <<, <<=, shift left  
  >>, >>=, shift right

- **Logical / comparison operators:**
  
  ==, !=, equals  
  >, >=, greater than  
  <, <=, less than  
  &&, ||, !, logical AND/OR/NOT
## Element-wise & Vector-scalar Operators

- **Unary operators:**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs</td>
<td>absolute value</td>
</tr>
<tr>
<td>acos</td>
<td>arccosine</td>
</tr>
<tr>
<td>asin</td>
<td>arcsine</td>
</tr>
<tr>
<td>atan</td>
<td>arctangent</td>
</tr>
<tr>
<td>ceil</td>
<td>round towards infinity</td>
</tr>
<tr>
<td>cos</td>
<td>cosine</td>
</tr>
<tr>
<td>cosh</td>
<td>hyperbolic cosine</td>
</tr>
<tr>
<td>exp</td>
<td>exponent</td>
</tr>
<tr>
<td>floor</td>
<td>round towards neg. infinity</td>
</tr>
<tr>
<td>log10</td>
<td>common logarithm</td>
</tr>
<tr>
<td>log</td>
<td>natural logarithm</td>
</tr>
<tr>
<td>rcp</td>
<td>reciprocal</td>
</tr>
<tr>
<td>round</td>
<td>round to nearest integer</td>
</tr>
<tr>
<td>rsqrt</td>
<td>reciprocal square root</td>
</tr>
<tr>
<td>sin</td>
<td>sine</td>
</tr>
<tr>
<td>sinh</td>
<td>hyperbolic sine</td>
</tr>
<tr>
<td>sqrt</td>
<td>square root</td>
</tr>
<tr>
<td>tan</td>
<td>tangent</td>
</tr>
<tr>
<td>tanh</td>
<td>hyperbolic tangent</td>
</tr>
</tbody>
</table>
Collective Operators

• Computations over entire vectors.  
  - The output(s) can in theory depend on all the inputs

• Reductions apply an operator over an entire vector to compute a distilled value (or values depending on the type of vector):

  \[
  \text{add\_reduce}([1 \ 0 \ 2 \ -1 \ 4]) \ \text{yields} \\
  1+0+2-1+4=6
  \]

• Scans compute reductions on all prefixes of a collection, either \textit{inclusively} or \textit{exclusively}:

  \[
  \begin{align*}
  \text{add\_iscan}([1 \ 0 \ 2 \ -1 \ 4]) & \ \text{yields} \\
  [1 \ (1+0) \ (1+0+2) \ (1+0+2+(-1)) \ (1+0+2+(-1)+4)] \\
  [1 \ 1 \ 3 \ 2 \ 6] \\
  \text{add\_scan}([1 \ 0 \ 2 \ -1 \ 4]) & \ \text{yields} \\
  [0 \ 1 \ (1+0) \ (1+0+2) \ (1+0+2+(-1))] \\
  [0 \ 1 \ 1 \ 3 \ 2]
  \end{align*}
  \]
Collective Operators

\textbf{op\_reduce}

\textbf{op\_iscan}

\textbf{op\_scan}

\textbf{Default}
Collective Operators

- **add_reduce**
  - Input: 4, 2, -5, 8
  - Output: 9

- **add_iscan**
  - Input: 4, 2, -5, 8
  - Output: 10, 6, 1, 9

- **add_scan**
  - Input: 10, 4, 2, -5, 8
  - Output: 10, 14, 16, 11
Collective Operators

### Reductions

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add_reduce</td>
<td>add all elements</td>
</tr>
<tr>
<td>sum</td>
<td>add over all dimensions</td>
</tr>
<tr>
<td>and_reduce</td>
<td>logical AND all elements</td>
</tr>
<tr>
<td>all</td>
<td>AND over all dimensions</td>
</tr>
<tr>
<td>mul_reduce</td>
<td>multiply all elements</td>
</tr>
<tr>
<td>ior_reduce</td>
<td>logical OR on all elements</td>
</tr>
<tr>
<td>any</td>
<td>OR over all dimensions</td>
</tr>
<tr>
<td>max_reduce</td>
<td>maximum of all elements</td>
</tr>
<tr>
<td>min_reduce</td>
<td>minimum of all elements</td>
</tr>
<tr>
<td>xor_reduce</td>
<td>XOR on all elements</td>
</tr>
</tbody>
</table>

**NOTE:** “*_reduce*” operations on multidimensional collections operate a dimension at a time.

### Scans

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add_scan</td>
<td>prefix sum</td>
</tr>
<tr>
<td>add_iscan</td>
<td>inclusive prefix sum</td>
</tr>
<tr>
<td>and_scan</td>
<td>prefix logical and</td>
</tr>
<tr>
<td>and_iscan</td>
<td>inclusive prefix logical and</td>
</tr>
<tr>
<td>ior_scan</td>
<td>prefix logical or</td>
</tr>
<tr>
<td>ior_iscan</td>
<td>inclusive prefix logical or</td>
</tr>
<tr>
<td>max_scan</td>
<td>prefix maximum</td>
</tr>
<tr>
<td>max_iscan</td>
<td>inclusive prefix maximum</td>
</tr>
<tr>
<td>min_scan</td>
<td>prefix minimum</td>
</tr>
<tr>
<td>min_iscan</td>
<td>inclusive prefix minimum</td>
</tr>
<tr>
<td>mul_scan</td>
<td>prefix multiply</td>
</tr>
<tr>
<td>mul_iscan</td>
<td>inclusive prefix multiply</td>
</tr>
<tr>
<td>xor_scan</td>
<td>prefix exclusive-or</td>
</tr>
<tr>
<td>xor_iscan</td>
<td>inclusive prefix exclusive-or</td>
</tr>
</tbody>
</table>
Data Reorganization Operators

\[ a = b(\{1,2,1,0\}); \]
\[ a = \text{gather}(b, \{1,2,1,0\}) \]
\[ x = b[2]; \]
\[ x = \text{gather}(b, 2); \]

\[ a = \text{scatter}(b, \{3,0,1,4\}, 5, 42); \]
Data Reorganization Operators

\[ a = \text{pack}(b, \{0, 1, 1, 0, 1\}); \]

\[ a = \text{unpack}(b, \{0, 1, 1, 0, 1\}, 42); \]
Data Reorganization Operators

\[ a = \text{shift}(b, 1, \text{default}); \quad \text{// left if positive; right if negative} \]

\[ a = \text{shift\_sticky}(b, -1); \quad \text{// shift with duplicated boundary value} \]

\[ a = \text{rotate}(b, 1); \quad \text{// shift with rotated boundary values} \]
Functions

• C++ Space
  • No call operator
    – Standard C/C++ function call
    – Compiler decides on inlining, or can use inline keyword
  • call operator
    – call() to invoke an C++ function w/ ArBB code
    – Triggers on-demand code compilation

• Intel® ArBB Space
  • No operator
    – Standard C/C++ function call
    – Full inlining of function body
      – Including virtual functions and callbacks!
  • call operator
    – call() to invoke an C++ function w/ ArBB code
    – True function call involving a branch
  • map operator
    – Replicates function over index space of array
Calls

call(function_ptr)(arg1, arg2, ..., argn);

- Call used from native C/C++ code or ArBB code
  - Manages and caches dynamic compilations

function_ptr(arg1, arg2, ..., argn);

- Direct invocation from within ArBB function
  - Direct invocation of ArBB operations outside of ArBB functions is not recommended
Map

```c
map(function_ptr)(arg1, arg2, ..., argn);
```

- **Invokes elemental function**
  - Operations are on scalar elements, which simplifies porting
  - Elemental function can also be used as a scalar function

- **Actual arguments**
  - Should match formal argument type exactly OR
  - Be a collection with an element type that matches the formal argument type exactly

- **Only used from within ArBB “call” function**
Stencils in “Extended” Elemental Functions

• Use `neighbor()` for stencil codes:

```c
void fun3x3<f32 a,
    f32 w0, f32 w1, f32 w2, f32 w3, f32 w4,
    f32& r) {
    r = w0 * a +
        w1 * neighbor(a, -2) + w2 * neighbor(a, -1) +
        w3 * neighbor(a, 1) + w4 * neighbor(a, 2);
}
```

-2 -1 0 1 2
Closures

closure<ret_t(arg1_t, arg2_t, ...9)>
  - A code sequence that has been captured and can be invoked
  - Has a related non-template type, auto_closure, that performs run-time type checking instead of static type checking

call() actually returns a closure<...>
  • call() of same function pointer does capture only the first time
  • after that, always returns same closure from cache
  • Provides simplified usage for new users

capture() just returns closure, does not call or cache

```cpp
void my_function(f32& out, f32 in);
f32 f1, f2;

auto_closure c1 = call(my_function);
auto_closure c2 = call(my_function);
assert(c1 == c2);
call(my_function)(f1, f2);  // works as expected.
c1(f1, f2);                 // equivalent to previous line
```
C++ control flow

• Regular C++ control flow works *during capture*
  – When a function is captured into a closure, the closure contains the *effect* of the control flow *at the time it was captured*

```cpp
bool positive = true;
void my_function(f32& result, f32 input) {
    if (positive) {
        result = input + 1.0f;
    } else {
        result = input - 1.0f;
    }
}

int main() {
    closure<void (f32&, f32)> closure_pos = capture(my_function);
    positive = false;
    closure<void (f32&, f32)> closure_neg = capture(my_function);
}
```
**ArBB control flow**

- ArBB provides its own control flow constructs
  - They can only be used with ArBB types (e.g. boolean)
  - The *control flow* will be captured, not just its effects

```c
boolean positive = true;
void my_function(f32& result, f32 input) {
  _if (positive) {
    result = input + 1.0f;
  } _else {
    result = input - 1.0f;
  } _end_if;
}

int main() {
  closure<void (f32&, f32)> closure = capture(my_function);
  positive = false;
  // No need to re-capture
}
```