Agenda

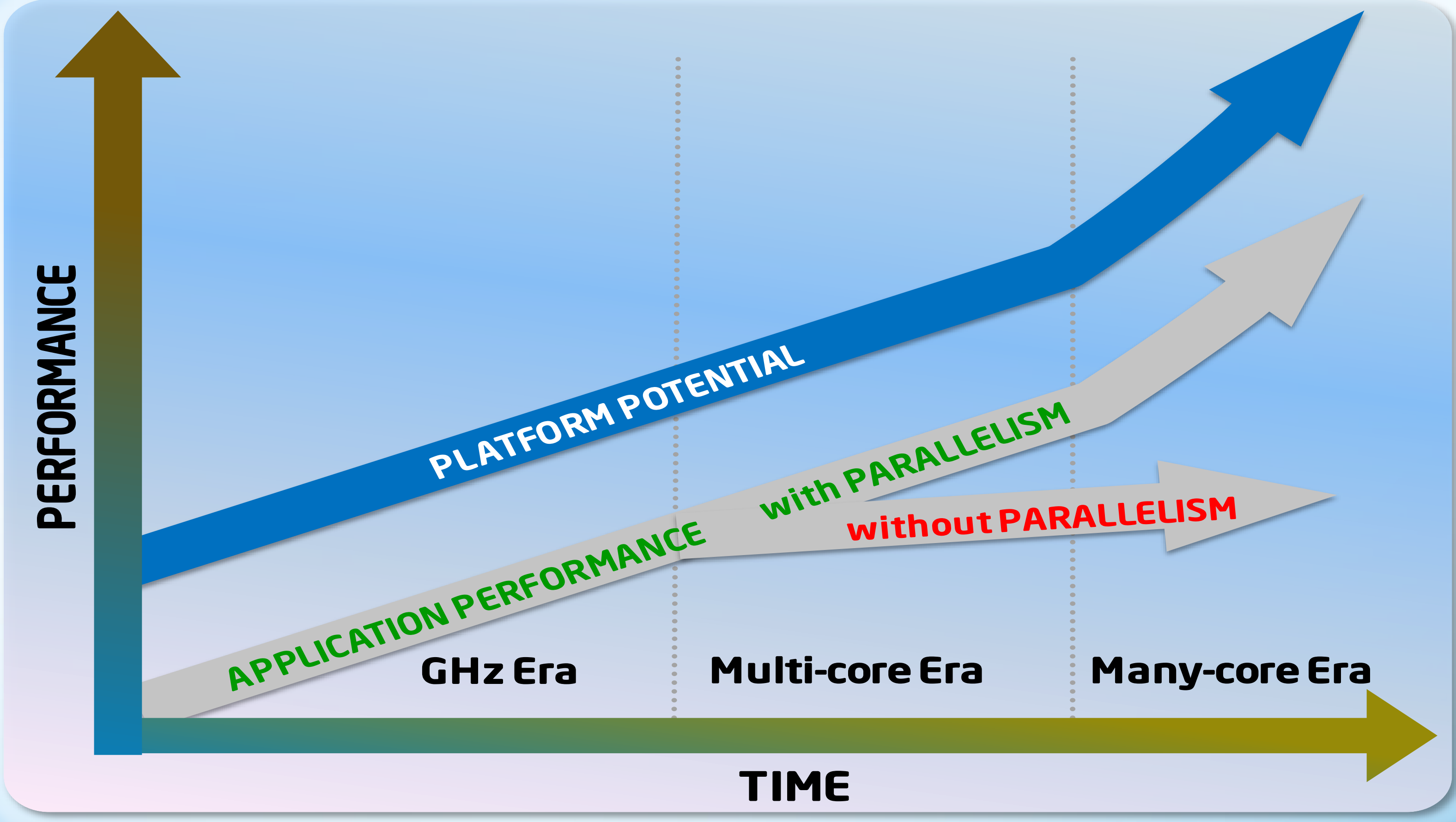
Time	Title
09:00-09:15	Welcome, Introduction
09:15-09:45	Overview – Intel Processor Architecture Evolution
09:45-10:15	Intel's Software Development offerings at a glance
10:15-11:15	**Strategies for Parallelism with Intel® Threading Methodologies**
11:15-12:30	Intel® Composer XE – the powerful compiler and performance libraries collection
12:30-13:30	*Break*
13:30-14:00	Intel® Inspector XE - Detect memory and threading errors
14:00-14:30	Intel® Amplifier XE - Understand performance issues
14:30-15:00	Intel® Advisor XE – Get started with parallelization
15:00-16:00	Intel Development Tools for Multi-threading in a typical Development Cycle – Life-Demo – Wrap-up
16:00- …	Q&A, opens, discussion

Intel® Multithreading Methodologies

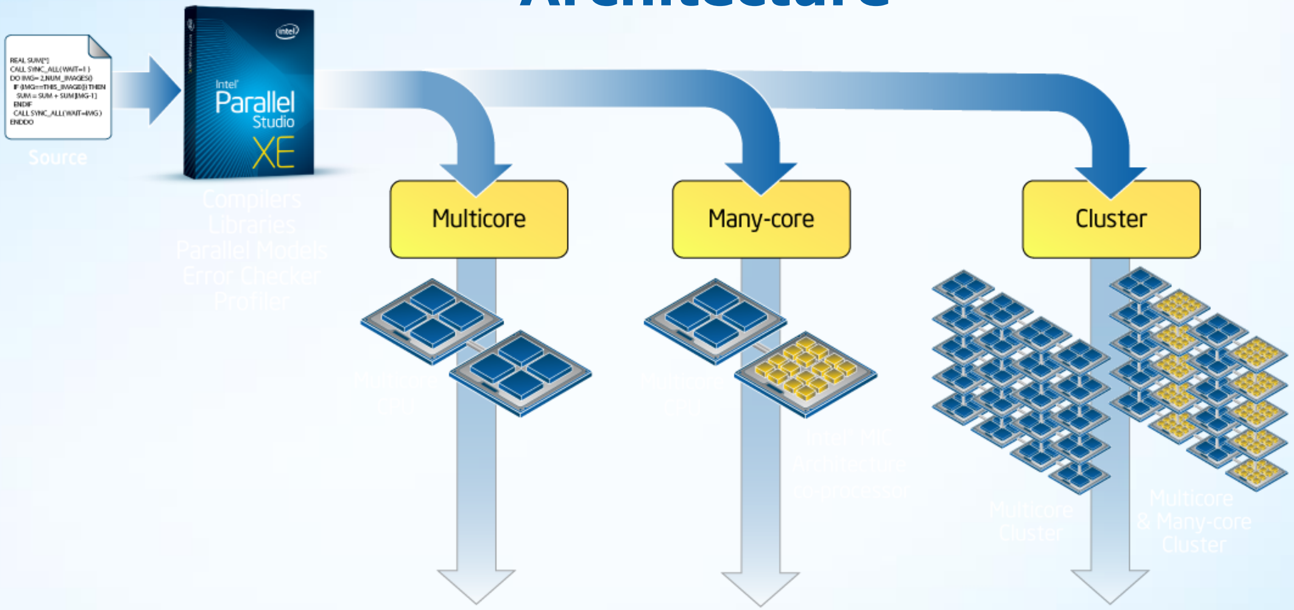
One day seminar at TU Dresden, Oct 17, 2013

Hubert Haberstock
Technical Consulting Engineer
Developer Products Division
Intel Corp.

System/Processor Performance Increase



Advanced Tools to Develop Code for Intel® Xeon® Processors Today, Easily Extends to Intel® MIC Architecture



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

KISTI www.kisti.re.kr

"By just utilizing standard programming on both Intel® Xeon processor and Intel® MIC architecture based platforms, the performance met multi-threading scalability expectations and we observed near-theoretical linear performance scaling with the number of threads." – *Hongsuk Yi, Heterogeneous Computing Team Leader, KISTI Supercomputing Center*

sgi

"SGI understands the significance of inter-processor communications, power, density and usability when architecting for exascale. Intel has made the leap towards exaflop computing with the introduction of Intel® Many Integrated Core (MIC) architecture. Future Intel® MIC products will satisfy all four of these priorities, especially with their expected ten times increase in compute density coupled with their familiar X86 programming environment." – *Dr. Eng Lim Goh, SGI CTO*

Win32 API Native Threads

„Explicit Threads"

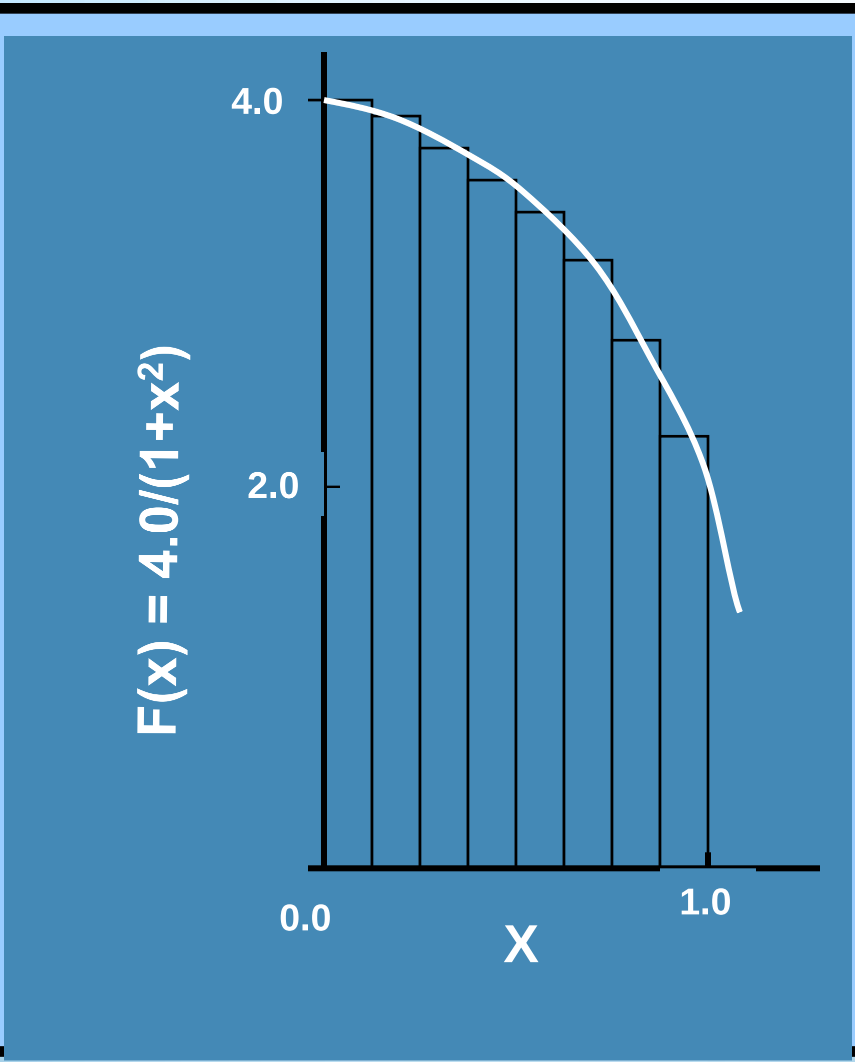
- The OS provides a library implementing an API for creating, managing and destroying threads
- Advantages of thread libraries:
  - The thread library gives you detailed control over the threads
- Disadvantage of thread libraries:
  - The thread library REQUIRES that you take detailed control over the threads
- Full control over all aspects of threading
- Requires thorough understanding of threading mechanisms
- Significant code bloat
  - "re-invent the wheel" with each application
  - hard to maintain
- Not portable among OSs

POSIX* Threads (Pthreads*)

- Create threads to execute work encapsulated within functions
- Typical to wait for threads to terminate
- Coordinate shared access between threads to avoid race conditions
  - Local storage to avoid conflicts
  - Synchronization objects to organize use
- Full control over all aspects of threading
- Requires thorough understanding of threading mechanisms
- Significant code bloat
  - “re-invent the wheel” with each application
  - hard to maintain
- Not portable among OSs

Our running Example: The PI program

Numerical Integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\, dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

PI Program:

The sequential program

```
static long num_steps = 100000;
double step;
void main ()
{
    int i;
    double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;

    for (i=1;i<= num_steps; i++){
          x = (i-0.5)*step;
          sum = sum + 4.0/(1.0+x*x);
    }

   pi = step * sum;
}
```

Win32 API - Complex and Error Prone

```
#include <windows.h>
#define NUM_THREADS 2
HANDLE thread_handles[NUM_THREADS];
CRITICAL_SECTION hUpdateMutex;
static long num_steps = 100000;
double step;
double global_sum = 0.0;

void Pi (void *arg)
{
    int i, start;
   double x, sum = 0.0;


   start = *(int *) arg;
   step = 1.0/(double) num_steps;

   for (i=start;i<= num_steps;
                              i=i+NUM_THREADS){
        x = (i-0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
   }
   EnterCriticalSection(&hUpdateMutex);
   global_sum += sum;
   LeaveCriticalSection(&hUpdateMutex);
}
```

```
void main ()
{
   double pi; int i;
   DWORD threadID;
   int threadArg[NUM_THREADS];

   for(i=0; i<NUM_THREADS; i++)
              threadArg[i] = i+1;

   InitializeCriticalSection(&hUpdateMutex);

   for (i=0; i<NUM_THREADS; i++){
             thread_handles[i] = CreateThread(0, 0,
              (LPTHREAD_START_ROUTINE) Pi,
                             &threadArg[i], 0, &threadID);
   }

   WaitForMultipleObjects(NUM_THREADS,
                             thread_handles, TRUE,INFINITE);

   pi = global_sum * step;

   printf(" pi is %f \n",pi);
}
```

Example - Pi-computation with reduction

Focus on application Logic, not implementation

```
#include <omp.h>
static long num_steps = 100000;
double step;
#define NUM_THREADS 2
void main ()
{   int i;
    double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel for reduction(+:sum) private(x)
    for (i=1;i<= num_steps; i++){
          x = (i-0.5)*step;
          sum = sum + 4.0/(1.0+x*x);
  }
  pi = step * sum;
}
```

OpenMP* – A Standard

- OS independent specification, standardized by www.openmp.org
- Thread Control via compiler pragmas, library routines and environment variables
- Intel® C++ Compiler :
  - */Qopenmp, -openmp* to enable recognition of directives
  - /Qopenmp-report:0|1|2, -openmp-report0|1|2

    ‚0‘ is report disabled, ‚2‘ maximum diagnostics level

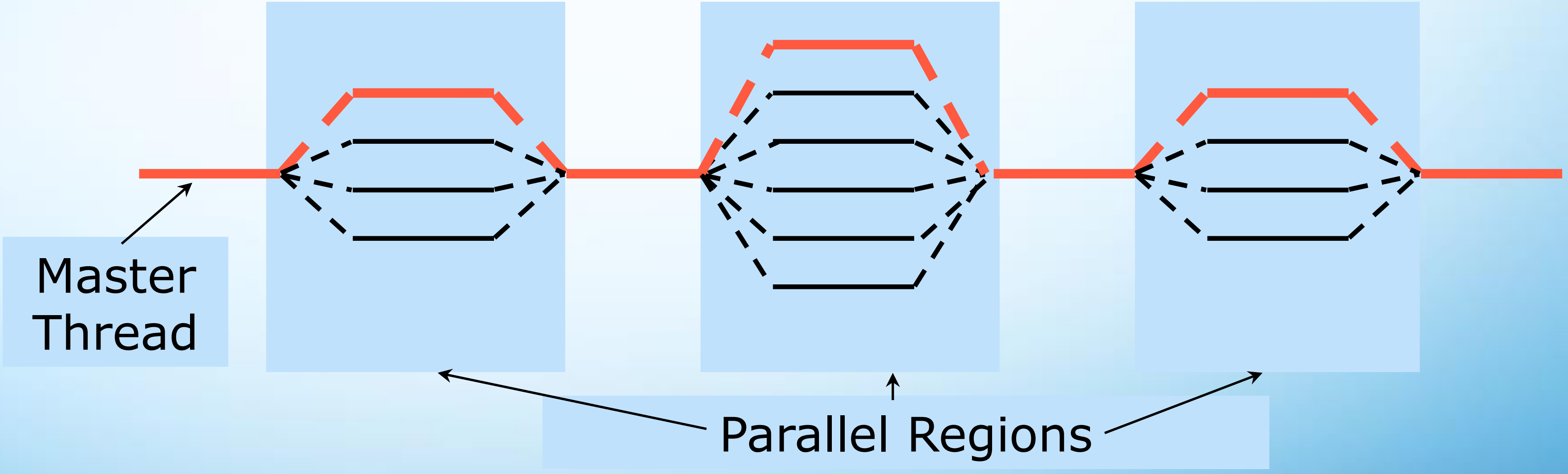
    to provide information about program sections that could be parallelized.
  - Header file or Fortran module

    ```
    #include “omp.h”
    use omp_lib
    ```
  - OpenMP 4.0 RC2 support
- Advantage over native threads:
  - Incremental approach: insert pragmas (use */Qopenmp)*
  - Easily switch back to original code (use*/Qopenmp-stubs*)
  - Threading internals hidden from the user
- Disadvantage over native threads:
  - Less detailed control of threading internals

Programming Model

Fork-join parallelism:

- Master thread spawns a team of threads
- Parallelism is added incrementally: the sequential program evolves into a parallel program



OpenMP* with Intel Compilers

- Intel® Composer XE 2013 compilers support OpenMP* 3.1
  Plus some extensions (**KMP_***) like:
  - **KMP_DETERMINISTIC_REDUCTIONS** (new!)
  - **KMP_AFFINITY**
  - **KMP_PLACE_THREADS** (new!)
  - …
- Intel® Composer XE 2013 Update 2 OpenMP* 4.0 (RC1!) support started:
  - Vectorization
  - Execution on coprocessors
- Both have already been there as Intel only extensions (different syntax, though)

OpenMP* with Intel Compilers (2)

- Intel® Composer XE 2013 SP1 (rel. 4th Sep 2013)
  - OpenMP* 4.0 (RC2!) support

  - User-defined reductions (UDRs)
  - SIMD support
  - Task extensions
  - Accelerator support
  - Cancellation support
  - Extensions
    - Affinity
    - Atomic

OpenMP* 4.0 status

- C/C++:
  http://software.intel.com/en-us/articles/openmp-40-features-in-intel-c-composer-xe-2013
- Fortran:
  http://software.intel.com/en-us/articles/openmp-40-features-in-intel-fortran-composer-xe-2013

User-defined reductions: Motivation

- Allows to extend what types and operations are allowed in an OpenMP reduction clause:

```
struct point {
    int x;
    int y;
};

struct point points[N];
struct point min = { MAX_INT, MAX_INT }, max = {0,0};

#pragma omp parallel for reduction(….)
for ( int i = 0; i < N; i++ )
{
          if ( point[i].x < min.x ) min.x = point[i].x;
          if ( point[i].y < min.y )  min.y = point[i].y;
          if ( point[i].x > max.x ) max.x = point[i].x;
          if ( point[i].y > max.y ) max.y = point[i].y;
}
```

Not possible before 4.0

User-defined reductions: declaration

- New declarative directive

#pragma omp declare reduction(*reduction-identifier* **:** *typename-list* **:** *combiner* **)** [*initializer-clause*]

!$omp declare reduction(*reduction-identifier* **:** *type-list* **:** *combiner* **)** [*initializer-clause*]

- *reduction-identifier* is the “operator” name given to this reduction
- *combiner* specifies how to combine two elements of one the specified types
  - Only two special variables can be used:
    - omp_in
    - omp_out
  - In C/C++, an expression
  - In Fortran, either an assignment statement or subroutine name with its arguments
    - no CALL keyword

User-defined reductions: example

```
struct point {
    int x;
    int y;
};
#pragma omp declare reduction(min : struct point : \
           omp_out.x = omp_in.x > omp_out.x  ? omp_out.x : omp_in.x, \
           omp_out.y = omp_in.y > omp_out.y  ? omp_out.y : omp_in.y ) \
           initializer( omp_priv = { MAX_INT, MAX_INT } )
#pragma omp declare reduction(max : struct point : \
           omp_out.x = omp_in.x < omp_out.x  ? omp_out.x : omp_in.x, \
           omp_out.y = omp_in.y < omp_out.y  ? omp_out.y : omp_in.y ) \
           initializer( omp_priv = { 0,0 } )

struct point points[N];
struct point minp = { MAX_INT, MAX_INT }, maxp = {0,0};

#pragma omp parallel for reduction(min:minp) reduction(max:maxp)
for ( int i = 0; i < N; i++ )
{
           if ( point[i].x < minp.x ) minp.x = point[i].x;
           if ( point[i].y < minp.y )  minp.y = point[i].y;
           if ( point[i].x > maxp.x ) maxp.x = point[i].x;
           if ( point[i].y > maxp.y ) maxp.y = point[i].y;
}
```

Not really necessary

Used here as a regular **reduction**

Optimization Notice

SIMD Support: motivation

- Provides a portable high-level mechanism to specify SIMD parallelism
  - Heavily based on Intel’s SIMD directive
- Two main new directives
  - To SIMDize loops
  - To create SIMD functions

SIMD loops: syntax

#pragma omp simd [*clauses*]
for-loop

!$omp simd [*clauses*]
do-loops
*[***!$omp end simd***]*

- Loop has to be in “Canonical loop form”
  - as do/for worksharing

SIMD loop clauses

- **safelen** (length)
  - Maximum number of iterations that can run concurrently without breaking a dependence
    - in practice, maximum vector length
- **linear** (list[:linear-step])
  - The variable value is in relationship with the iteration number
    - $x_i = x_{orig} + i * \text{linear-step}$
- **aligned** (list[:alignment])
  - Specifies that the list items have a given alignment
  - Default is alignment for the architecture
- **private** (list)
- **lastprivate** (list)
- **reduction** (operator:list)
- **collapse** (n)

Same as existing clauses

SIMD loop example

```
double pi()
{
    double pi = 0.0;
    double t;
#pragma omp simd private(t) reduction(+:pi)
    for (i=0; i<count; i++) {
        t = (double)((i+0.5)/count);
        pi += 4.0/(1.0+t*t);
    }
    pi /= count
    return pi;
}
```

SIMD functions: Syntax

#pragma omp declare simd *[clauses]*

[#pragma omp declare simd *[clauses]]*

function definition or declaration

!$omp declare simd (*function-or-procedure-name***)** *[clauses]*

- Instructs the compiler to
  - generate a SIMD-enabled version(s) of a given function
  - that a SIMD-enabled version of the function is available to use from a SIMD loop

SIMD functions: clauses

- **simdlen(*length*)**
  - generate function to support a given vector length
- **uniform(*argument-list*)**
  - argument has a constant value between the iterations of a given loop
- **inbranch**
  - function always called from inside an if statement
- **notinbranch**
  - function never called from inside an if statement

- **linear(*argument-list[:linear-step]*)**
- **aligned(*argument-list[:alignment]*)**
- **reduction(*operator:list*)**

Same as before

SIMD combined constructs

- Worksharing + SIMD

#pragma omp for simd [clauses]
!$omp do simd [clauses]
[!$omp end do simd]

  – First vectorize the loop, then distribute the resulting iterations among threads

- Parallel + worksharing + SIMD

#pragma omp parallel for simd *[clause[[,] clause] ...]*

!$omp parallel do simd *[clause[[,] clause] ...]*
!$omp end parallel do simd

SIMD functions example

```
#pragma omp simd notinbranch
float min(float a, float b) {
    return a < b ? a : b;
}

#pragma omp simd notinbrach
float distsq(float x, float y) {
    return (x - y) * (x - y);
}


#pragma omp parallel for simd
    for (i=0; i<N; i++)
        d[i] = min(distsq(a[i], b[i]), c[i]);
```

Cancellation Constructs

- Parallel execution cannot be aborted in OpenMP 3.1
  - Code regions must always run to completion
  - (or not start at all)

- Cancellation in OpenMP 4.0 provides a best-effort approach to terminate OpenMP regions
  - Best-effort: not guaranteed to trigger termination immediately
  - Triggered "as soon as" possible

- Two constructs:
  - Cancellation request: `#pragma omp cancel`
  - Cancellation points: `#pragma omp cancellation point`

`cancel` Construct

- Syntax:
  `#pragma omp cancel` *`construct-type-clause [ [, ]if-clause]`*
  `!$omp cancel` *`construct-type-clause [ [, ]if-clause]`*
- Clauses:

  ```
  parallel
  sections
  for  (C/C++)
  do   (Fortran)
  taskgroup
  if (scalar-expression)
  ```

- Semantics
  - Requests cancellation of the inner-most OpenMP region of the type specified
  - Lets the encountering thread/task proceed to the region

`cancellation point` Construct

- Syntax:
  ```
  #pragma omp cancellation point construct-type-clause
  !$omp omp cancellation point construct-type-clause
  ```
- Clauses:
  ```
  parallel
  sections
  for  (C/C++)
  do   (Fortran)
  taskgroup
  ```
- Semantics
  - Introduces a user-defined cancellation point
  - Pre-defined cancellation points:
    - implicit/explicit barriers regions
    - `cancel` regions

OpenMP* - Sudoku Tasking Sample

```
void main()
{
/* Sudoku Bord aufsetzen */
#pragma omp parallel
{
#pragma omp single
{
#pragma omp taskgroup
          {
          solve_parallel(0, 0, sudoku);
          }
}
}
}

void solve_parallel(int x, int y, CSudokuBoard* sudoku, CSudokuBoard* & solution)
{
  if (x == sudoku->getFieldSize()) {      // end of line
    y++;   x = 0;
    if(y == sudoku->getFieldSize())     // end
      return true;
  }

  if (sudoku->get(y, x) > 0) {            // field already set
    return solve_parallel(x+1, y, sudoku); // tackle next field
  }

  for (int i = 1; i <= sudoku->getFieldSize(); i++) { // try all numbers
    if (!sudoku->check(x, y, i)) {
#pragma omp task firstprivate(i,x,y,sudoku)
      {
        CSudokuBoard* new_sudoku = new CSudokuBoard(*sudoku);
        new_sudoku->set(y, x, i); // if number fits, set it
        if (solve_parallel(x+1, y, new_sudoku)) { // tackle next field
          // remember Sudoku board and stop parallel execution
#pragma omp critical
          if (!solution) {
            solution = new_sudoku;
#pragma omp cancel taskgroup
          }
        }
        delete new_sudoku;
      }
    }
  }
#pragma omp taskwait

  sudoku->set(y, x, 0);               // no solution found, reset field
}
```

- Parallel Region
  - Single Task
  - Taskgroup
    - solve_parallel
      - Check for new field
      - for //try all members
        - Create Tasks to find solutions
        - Critical Section to check for solution
        - Cancel Taskgroup if solution found (clear memory)

        Close Tasks
      - Reset field if no solution

Atomic Extensions

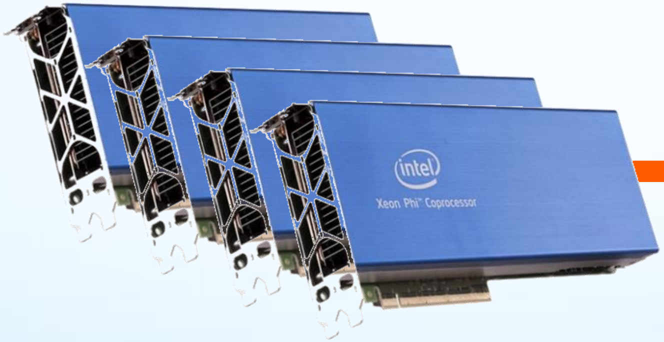
- OpenMP 3.1 introduced “atomic capture”
  - Allow to capture the old/new value of an atomic update
  - Need to implement scheduling-type algorithms
    - Update to next element
    - Process current element
  - New clauses
    - read: atomically read a variable
    - write: atomically write a variable
    - update: “old behavior” of atomic
    - capture: capture old value before update
- OpenMP 4.0 introduces sequentially consistent atomics
  - New clause: seq_cst
  - Includes a flush without a list

Atomic Extensions: Examples

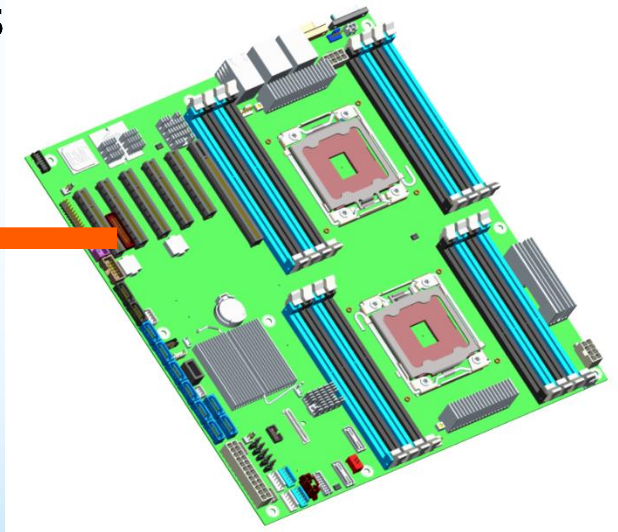
```
#pragma omp atomic read
v = x                 // atomically read x and store in v

#pragma omp atomic write
x = y + z;            // atomically update x

#pragma omp atomic update
x++;                  // atomically increment x

#pragma omp atomic capture
{v = x; x += 1;}      // capture value of x BEFORE update

#pragma omp atomic capture
{x += 1; v = x}        // capture value of x AFTER update
```

Accelerator Support

- OpenMP 4 will support accelerators and coprocessors
- Device model:
  - One host
  - Multiple accelerators/coprocess



Coprocessors



Host

`target declare` Clauses

- **C/C++**

  #pragma omp declare target *new-line*

  [function-definition-or-declaration]

  #pragma omp end declare target *new-line*

- **Fortran**

  !$omp declare target *[(proc-name-list | list)] new-line*

`target` Clauses

#pragma omp target *[clause[[,] clause],...] new-line*
structured-block

Clauses: **device(***scalar-integer-expression***)**
map(alloc | to | from | tofrom: *list***)**
if(*scalar-expr***)**

#pragma omp target data *[clause[[,] clause],...] new-line*
structured-block

Clauses: **device(***scalar-integer-expression***)**
map(alloc | to | from | tofrom: *list***)**
if(*scalar-expr***)**

#pragma omp target update *[clause[[,] clause],...] new-line*

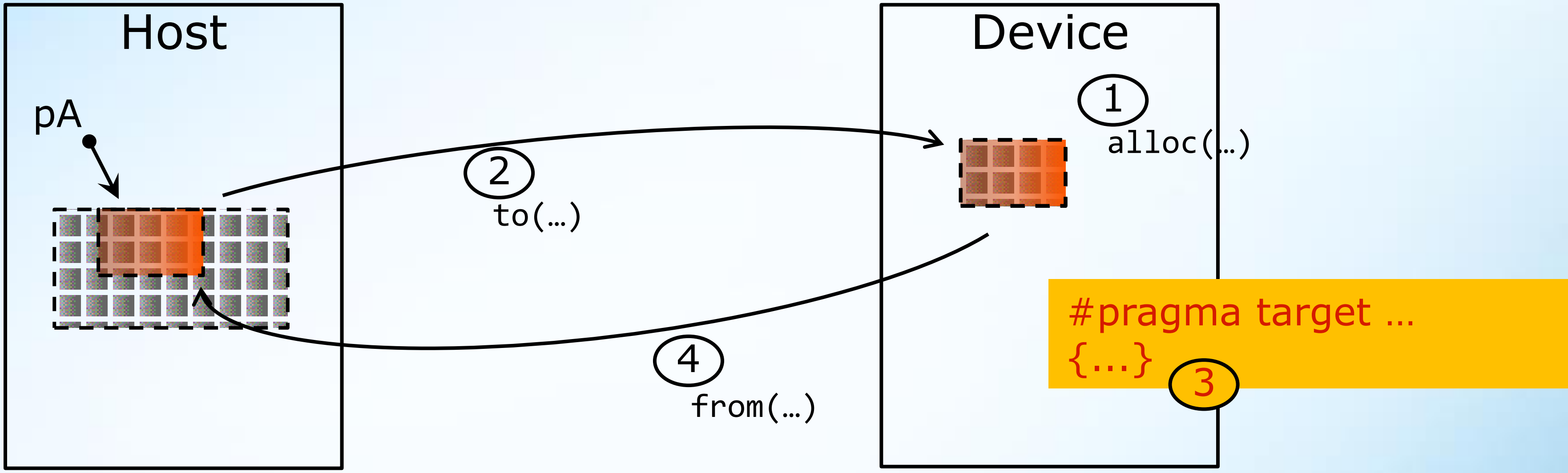
Clauses: **to(** *list* **)**
from(*list* **)**
device(*integer-expression* **)**
if(*scalar-expression* **)**

Examples

```
#pragma omp target map(to(b:count)) map(to(c,d)) map(from(a:count))
  {
#pragma omp parallel for
    for (i=0; i<count; i++) {
      a[i] = b[i] * c + d;
    }
  }
```

```
#pragma omp target data map(alloc(tmp:N)) map(in(input:N)) map(from(result))
  {
#pragma omp parallel
    {
#pragma omp for
      for (i=0; i<N; i++)
        tmp[i] = some_computation(input[i], i);
#pragma omp single
      do_some_other_stuff(tmp);
#pragma omp for reduction(+:result)
      for (i=0; i<N; i++)
        result += final_computation(tmp[i], i)
  }
```

Execution Model



- The `target` construct transfers the control flow to the target device
  - The transfer clauses control direction of data flow
  - Array notation is used to describe array length
- The `target data` construct creates a *scoped* device data environment
  - The transfer clauses control direction of data flow
  - Device data environment is valid through the lifetime of the `target data` region
- Use `target update` to request data transfers from within a `target data` region

OpenMP Affinity

- OpenMP 3.1 introduced a thread binding interface
- OpenMP 4 extends the interface to allow for full thread affinity
  - Vendor-agnostic affinity settings (e.g. no more KMP_AFFINITY)
  - Easier interaction between environment and OpenMP runtime
- Terminology:
  - Place: unordered set of processors
  - Place list: ordered list of available places for execution
  - Place partition: Contiguous interval in the place list
- Policies / affinity types:
  - Master: keep worker threads in the same place partition as the master thread
  - Close: keep worker threads “close” to the master thread in contiguous place partitions
  - Spread: create a sparse distribution of worker threads across the place partitions

OpenMP Affinity

- Additional clause for parallel regions:
  proc_bind(*affinity-type*)

- Environment variables control the affinity settings:
  - OMP_PROC_BIND
    e.g., export OMP_PROC_BIND=“spread,spread,close”
  - OMP_PLACES
    e.g., export
    OMP_PLACES=“{0,1,2,3},{4,5,6,7},{8:4},{12:4}”

- Places are system-specific and are not defined by OpenMP

OpenMP 4.0* (RC2) – Target

- Ideal for coprocessors like Intel® Xeon Phi™ (Intel® MIC):
- C/C++:

```
#pragma omp declare target
    [function-definition-or-declaration]
#pragma omp end declare target
```

- Fortran:

```
!$omp declare target [(proc-name-list|list)]
```

- Similar to existing explicit offload constructs from Intel.
- Existing code just requires minimal changes!

OpenMP 4.0* (RC2) – Target Clauses

- **#pragma omp target [clause[[,] clause],...]**
  structured-block
  Clauses:
  - device(scalar-integer-expression)
  - map(alloc | to | from | tofrom: list)
  - if(scalar-expr)
- **#pragma omp target data [clause[[,] clause],...]**
  structured-block
  Clauses:
  - device(scalar-integer-expression)
  - map(alloc | to | from | tofrom: list)
  - if(scalar-expr)
- **#pragma omp target update [clause[[,] clause],...]**
  Clauses:
  - to(list)
  - from(list)
  - device(integer-expression)
  - if(scalar-expression)

OpenMP 4.0* (RC2) – Target Example

```
#pragma omp target data device(0)
            map(alloc:tmp[0:N]) map(to:input[:N)) map(from:result)
{
#pragma omp target device(0)
#pragma omp parallel for
  for (i=0; i<N; i++)
    tmp[i] = some_computation(input[i], i);

  do_some_other_stuff_on_host();

#pragma omp target device(0)
#pragma omp parallel for reduction(+:result)
  for (i=0; i<N; i++)
    result += final_computation(tmp[i], i)
}
```

OpenMP 4.0* (RC2) – Other features

```
#pragma omp target data device(0)
            map(alloc:tmp[0:N]) map(to:input[:N)) map(from:result)
{
#pragma omp target device(0)
#pragma omp parallel for
  for (i=0; i<N; i++)
    tmp[i] = some_computation(input[i], i);

  do_some_other_stuff_on_host();

#pragma omp target device(0)
#pragma omp parallel for reduction(+:result)
  for (i=0; i<N; i++)
    result += final_computation(tmp[i], i)
}
```

Summary

- OpenMP 4.0 is the next major release of the OpenMP API
- Biggest additions
  - Tasking extensions
  - SIMD constructs
  - Accelerator/coprocessor support
- Lots of minor improvements
- OpenMP 4.0 will put OpenMP a big step forward

Simplify Parallelism

Intel® Cilk™ Plus, Intel® Threading Building Blocks

	Intel® Cilk™ Plus	Intel® Threading Building Blocks
What	Language extensions to simplify task/data parallelism	Widely used C++ template library for task parallelism
Features	• #pragma SIMD and array notation: easy-to-use, powerful vectorization • 3 simple keywords for parallelism • Support for task & data parallelism • Semantics similar to serial code	• Parallel algorithms and data structures • Scalable memory allocation and task scheduling • Synchronization primitives
Why	• Simple way to vectorize, parallelize your code • Sequentially consistent, low overhead, powerful solution • Supports C, C++, Windows and Linux • Get more from your IA hardware	• Rich feature set for general purpose parallelism • Available as open source or commercial license • Supports C++, Windows, Linux, Mac OS X, other OSs

Vectorization and Parallelism Made Easier

Intel® Cilk™ Plus
History

- **MIT Cilk**
  - Over 15 years
  - C only
- **Cilk Arts, Inc. (acquired by Intel in 2009)**
  - Ported MIT Cilk to C++ (Intel® Cilk++)
- **Intel® Cilk™ Plus**
  - Integrated in Intel® Compiler + extensions (C & C++)
- **Open sourced Intel® Cilk™ Plus**
  - Supported by GCC (branch)

Intel® Cilk™ Plus

Task parallelism

Keywords

Set of keywords, for expression of task parallelism:

```
cilk_spawn
cilk_sync
cilk_for
```

Reducers

Reliable access to nonlocal variables without races

```
cilk::reducer_opadd<int> sum(3);
```

Data parallelism

CEAN

Provide data parallelism for arrays

```
mask[:] = a[:] < b[:] ?
-1 : 1;
```

Elementary functions

Define actions that can be applied to whole or parts of arrays or scalars

SIMD Pragma

Directive to extend vectorization

Execution parameters

Runtime system APIs, Environment variables, pragmas

Optimization Notice

Ways to Write Vector Code

Data Level Parallelism with Intel® Cilk™ Plus

(Auto-)Vectorization

```
for(i = 0; i < N; i++){
  A[i] = B[i] + C[i];
}
```

Array Notation for C/C++

```
A[:] = B[:] + C[:];
```

SIMD Pragma/Directive

```
#pragma simd
for(i = 0; i < N; i++) {
  A[i] = B[i] + C[i];
}
```

Elemental Function

```
__declspec(vector)
float foo(float B, float C, int
   i)
{
  return B + C;
}
…
for(i = 0; i < N; i++) {
  A[i] = foo(B[i], C[i], i);   }
```

Broader adoption of technology

- Intel Cilk Plus is available in the "cilkplus" branch of the GCC 4.8
- Explicit vectorization adopted by the OpenMP 4.0 specification.
  - **#pragma omp simd** *[clause[[,] clause] ...] new-line*
    - applied to a loop; indicates that the loop can be transformed into a SIMD loop
  - **#pragma omp declare simd**
    - applied to a function to enable the creation of a version that can process arguments using SIMD instructions from a single invocation from a SIMD loop.

New Parallelism Method: Intel® Cilk™ Plus

- An extension to C and C++ for expressing fine-grained task parallelism
  - Shared-memory multiprocessing (like OpenMP)
- Very simple syntax of 3 keywords only: ***_cilk_spawn*** and ***_cilk_sync*, *_cilk_for***
  - #include <cilk/cilk.h>
- Every Cilk program preserves the ***serial semantic***
- Cilk provides ***performance guarantees*** since it is based on theoretically efficient ***work-stealing*** scheduler
- Preventing races using ***reducer hyperobjects***
- ***Array Notations (CEAN)*** to provide data parallelism for sections of arrays or whole arrays
- ***Elemental Functions*** to enable data parallelism of whole functions or operations
- ***#pragma SIMD*** to express vector parallelism using SIMD hardware registers

What's New: Intel® Cilk™ Plus Increased Performance and Scalability

- Improved SIMD pragma loops and elemental functions vectorization support provides enhanced scalability and performance
- Enhanced performance and productivity with new Holder Hyperobjects for per-thread temporary storage feature
- SIMD pragma loops and elemental functions support for nested loops, array notation, switch statements, and break/continue statements
- More architectural and scalable way to define vector lengths with new SIMD pragma clause "vectorlengthfor" support of vectorization of loops and elemental functions
- Expanded Mac OS support

Intel® Cilk™ Plus keywords

- Cilk Plus adds three keywords to C and C++:

```
_Cilk_spawn
_Cilk_sync
_Cilk_for
```

- If you `#include <cilk/cilk.h>`, you can write the keywords as `cilk_spawn`, `cilk_sync`, and `cilk_for`.
- Cilk Plus runtime controls thread creation and scheduling. A thread pool is created prior to use of Cilk Plus keywords.
- The number of threads matches the number of cores by default, but can be controlled by the user.

Programmers View of Cilk

- Cilk key words and convenient aliases

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

`cilk_spawn`	alias	`_Cilk_spawn`
`cilk_sync`	alias	`_Cilk_sync`
`cilk_for`	alias	`_Cilk_for`

- API

```
__cilkrts_set_param(“nworkers”, “4”)
__cilkrts_get_nworkers()
__cilkrts_get_total_workers()
__cilkrts_get_worker_number()
```

- *Environment variable*

```
CILK_NWORKERS
```

cilk_spawn and cilk_sync

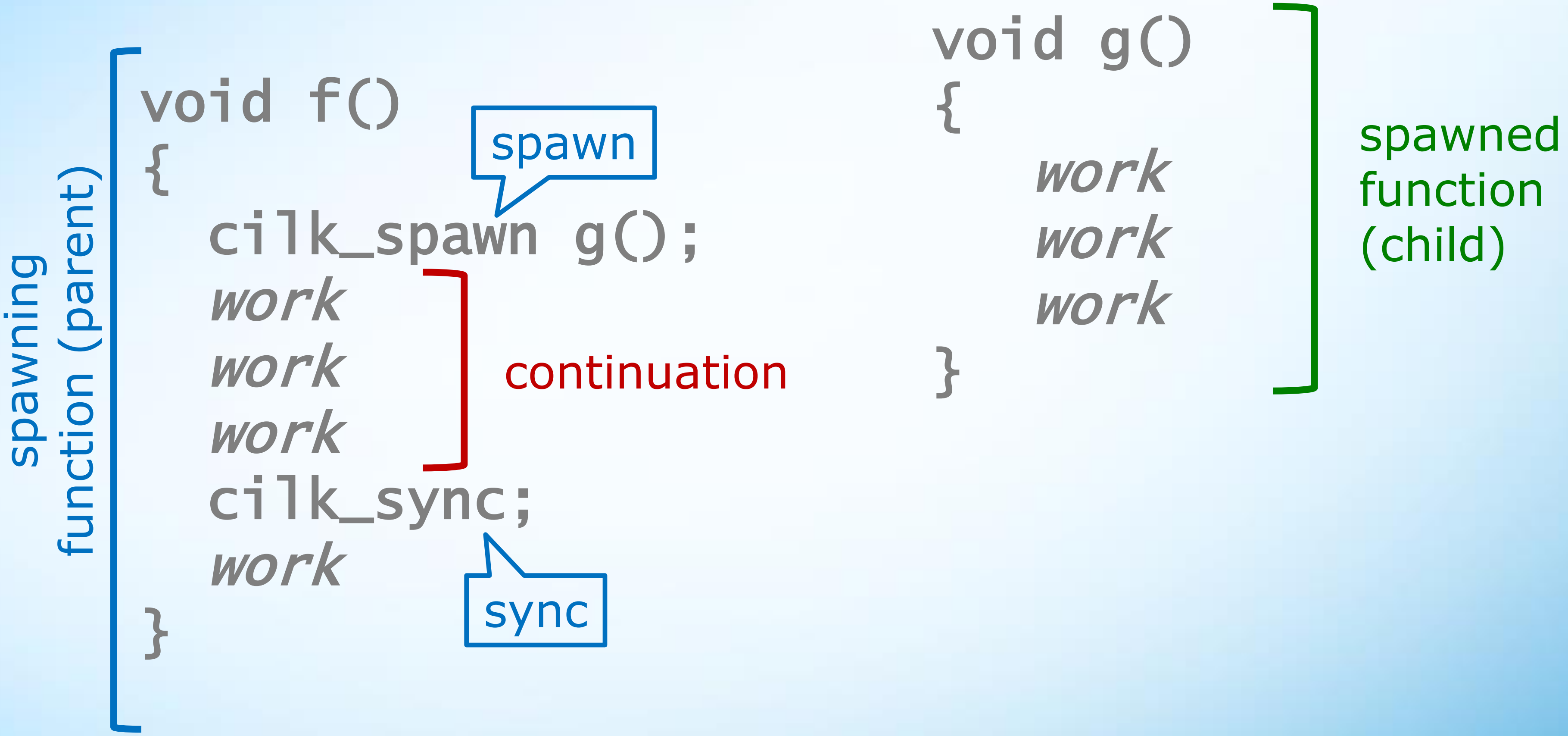
- cilk_spawn gives the runtime *permission* to run a child function asynchronously.
  - No 2nd thread is created or required!
  - If there are no available workers, then the child will execute as a serial function call.
  - The scheduler may *steal* the parent and run it in parallel with the child function.
  - The parent is not *guaranteed* to run in parallel with the child.
- cilk_sync waits for all children to complete before execution proceeds from that point.
  - There are implicit cilk_sync points – will discuss later

Serial Execution

```
void f()
{
   g();
   work
   work
   work

   work
}
```

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

Anatomy of a spawn



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

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

spawning function (parent)

spawn

continuation

sync

spawned function (child)

Work stealing

when no other worker is available

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

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

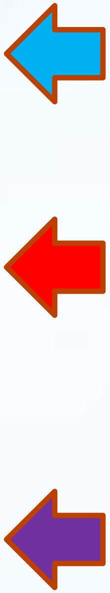
Worker A

Same behavior as serial execution!

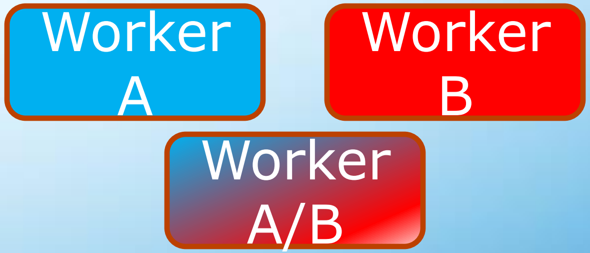
Work stealing

when other worker is available

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



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



A Note on Load Balancing

- The Cilk scheduler does automatic load balancing
- An idle worker will find work to do !
- If the program has enough parallelism, then all workers will stay busy

Work-stealing Overheads

- Spawning is cheap (3-5 times the cost of a function call)
- Stealing is much more expensive (requires locks and memory barriers)
- **Most spawns do not result in steals.**
- Balanced work loads → less stealing →less overhead.

cilk_for loop

- Looks like a normal for loop.

```
cilk_for (int x = 0; x < 1000000; ++x) { … }
```

- Any or all iterations may execute in parallel with one another.
- All iterations complete before program continues.
- Constraints:
  - Limited to a single control variable.
  - Must be able to jump to the start of any iteration at random.
  - Iterations should be independent of one another.

cilk_for vs. serial for with spawn

- Compare the following loops:

```
for (int x = 0; x < n; ++x) { cilk_spawn f(x); }

cilk_for (int x = 0; x < n; ++x) { f(x); }
```

- The above two loops have similar semantics, but…
- they have very different performance characteristics.

cilk_for examples

```
cilk_for (int x; x < 1000000; x += 2) { … }
```

✓

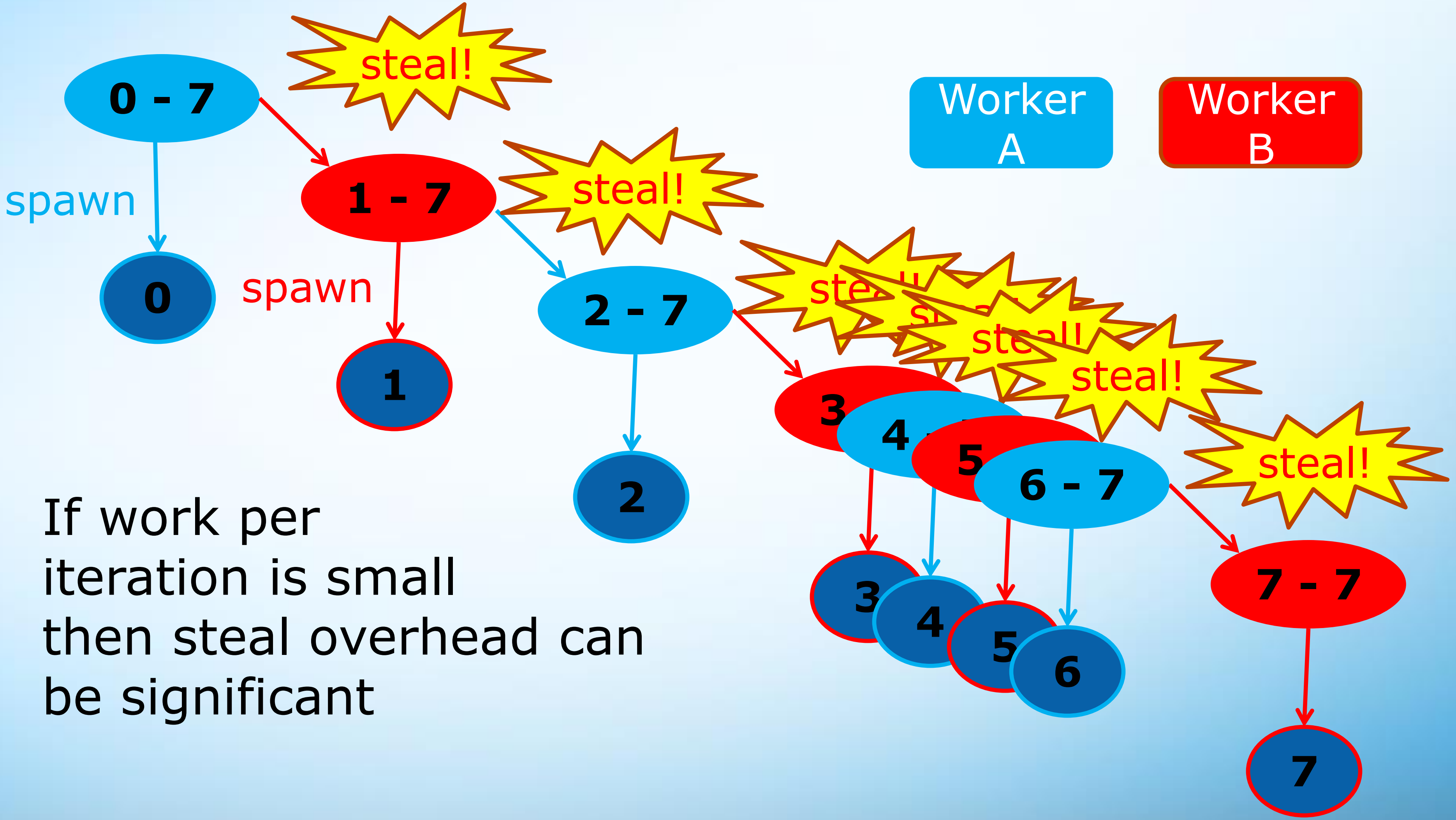
```
cilk_for (vector<int>::iterator x = y.begin();
               x != y.end(); ++x) { … }
```

✓

```
cilk_for (list<int>::iterator x = y.begin();
               x != y.end(); ++x) { … }
```

✗

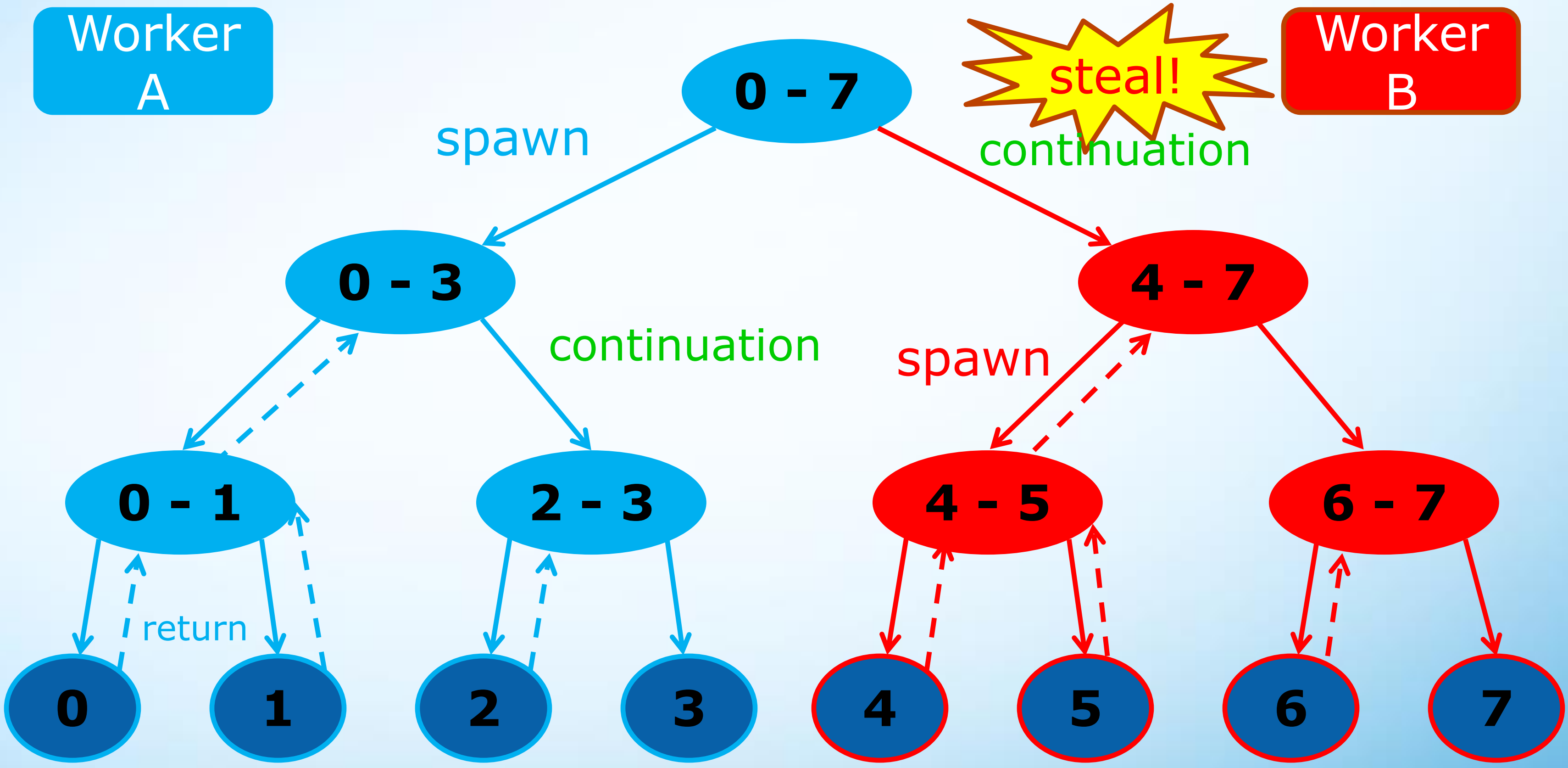
Serial for with spawn: unbalanced



If work per iteration is small then steal overhead can be significant

cilk_for: Divide and Conquer

```
cilk_for (int i=0; i< 8; ++i)
    f(i);
```



Divide and conquer results in fewer steals and less overhead.

Serialization

- Every Cilk Plus program has an equivalent serial program called the *serialization*
- The serialization is obtained by removing **cilk_spawn** and **cilk_sync** keywords and replacing **cilk_for** with **for**
  - The compiler will produce the serialization for you if you compile with `/Qcilk-serialize` (Windows)
- Running with only one worker is equivalent to running the serialization.

Serial Semantics

- A *deterministic* Cilk Plus program will have the same semantics as its serialization.
  - Easier regression testing
  - Easier to debug:
    - Run with one core
    - Run serialized
  - Strong analysis tools (Cilk Plus-specific versions will be posted on WhatIf)
    - race detector
    - parallelism analyzer

Implicit syncs

```
void f() {
  cilk_spawn g();
  cilk_for (int x = 0; x < lots; ++x) {
     ...
  }
  try {
    cilk_spaw      ();
  }
  catch (...) {
    ...
  }
}
```

At end of a `cilk_for` body (does not sync `g()`)

Before entering a `try` block containing a sync

At end of a `try` block containing a spawn

At end of a spawning function

A Summing Example in Serial Code

```
int compute(const X& v);
int main()
{
    const std::size_t n = 1000000;
    extern X myArray[n];
    // ...

    int result = 0;
    for (std::size_t i = 0; i < n; ++i)
    {
        result += compute(myArray[i]);
    }
    std::cout << "The result is: "
              << result
              << std::endl;
    return 0;
}
```

Summing Example in Intel® Cilk™ Plus

```
int compute(const X& v);
int main()
{
    const std::size_t n = 1000000;
    extern X myArray[n];
    // ...

    int result = 0;
    cilk_for (std::size_t i = 0; i < n; ++i)
    {
        result += compute(myArray[i]);
    }
    std::cout << "The result is: "
              << result
              << std::endl;
    return 0;
}
```

Race!

Locking Solution

```
int compute(const X& v);
int main()
{
    const std::size_t n = 1000000;
    extern X myArray[n];
    // ...

    mutex L;
    int result = 0;
    cilk_for (std::size_t i = 0; i < n; ++i)
    {
        int temp = compute(myArray[i]);
        L.lock();
        result += temp;
        L.unlock();
    }
    std::cout << "The result is: "
              << result
              << std::endl;
    return 0;
}
```

Problems

Lock overhead & lock contention.

Intel® Cilk™ Plus Reducer Solution

```
int co
int ma
{
    co                      AY_SI
    extern X myArr      RRAY_SIZ
    // ...

    cilk::reducer_opadd<int> result;
    cilk_for (std::size_t i = 0; i < ARRAY_SIZE; ++i)
    {
        result += compute(myArray[
    }
    std::cout << "The result is: "
              << result.get_value()
              << std::endl;
    return 0;
}
```

Declare result to be a summing reducer over int.

Updates are resolved automatically without races or contention.

At the end, the underlying int value can be extracted.

Reducer Library

Intel® Cilk™ Plus's hyperobject library contains many commonly used reducers:

- `reducer_list_append`
- `reducer_list_prepend`
- `reducer_max`
- `reducer_max_index`
- `reducer_min`
- `reducer_min_index`
- `reducer_opadd`
- `reducer_ostream`
- `reducer_basic_string`
- …

You can also write your own using `cilk::monoid_base` and `cilk::reducer`.

Reducer Benefits/Limitations

- Benefits
  - Reducers do not suffer from lock contention
  - Reducers retain serial semantics
    - For example, a list or ostream reducer will retain the same order you would get from serial execution
    - Even a correct locking solution cannot guarantee this
- Limitations
  - Operations on a reducer must be associative to behave deterministically
    - Refer to the operators supported by a particular reducer class for safe operations to use
    - Floating point types may get different results run-to-run
  - If using custom data types for reducers, refer to the header for the specific reducer for requirements

Intel® Cilk™ Plus Array Notations

- Array Notations provide a syntax to specify sections of arrays on which to perform operations
- syntax :: [<lower bound> : <length> : <stride>]
- Simple example
  - a[0:N] = b[0:N] * c[0:N];
  - a[:] = b[:] * c[:] // if a, b, c are declared with size N
- The Intel® C++ Compiler's automatic vectorization can use this information to apply single operations to multiple elements of the array using Intel® Streaming SIMD Extensions (Intel® SSE) and Intel® Advanced Vector Extensions (Intel® AVX)
  - Default is SSE2. Use compiler options (/Qx, /arch, /Qax) to change the target.
- More advanced example
  - x[0:10:10] = sin(y[20:10:2]);

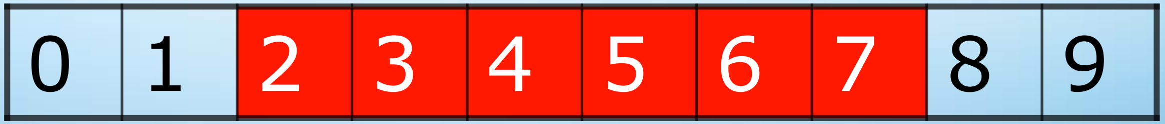
Array Section Notation

- Array Section Notation

  <array base> [<lower bound> : <length> [: <stride>]]
  [<lower bound> : <length> [: <stride>]].....

- Note that length is chosen.
  - Not upper bound as in Fortran [lower bound : upper bound]

```
A[:]              // All elements of vector A
B[2:6]            // Elements 2 to 7 of vector B
D[0:3:2]          // Elements 0,2,4 of vector D
E[0:3][0:4]       // 12 elements from E[0][0] to E[2][3]
```



0	1	2	3	4	5	6	7	8	9

```
float B[10];
B[2:6] = …
```

Operator Maps – Sections in Expressions

- Most arithmetic and logic operators for C/C++ basic data types are available for array sections:

  ```
  +, -, *, /, %, <,==,!=,>,|,&,^,&&,||,!,-(unary),
     +(unary),++,--, +=, -=, *=, /=, *(p)
  ```

- An operator is implicitly mapped to all the elements of the array section operands:

  ```
  a[0:s]+b[2:s] => {a[i]+b[i+2], forall (i=0;i<s;i++)}
  ```

  - Operations are parallel among all the elements
  - Array operands must have the same *rank*
  - Scalar operand is automatically expanded to fill the whole section

    ```
    a[0:s]*c => {a[i]*c, forall (i=0;i<s;i++)}
    ```

Simple Example: Dot product

Serial version

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

Array Notation version

```
float dot_product(unsigned int size, float A[size], float B[size])
{
    return __sec_reduce_add(A[:] * B[:]);
        // A[:] can also be written as A[0:size]
}
```

Intel® Cilk™ Plus Array Notations Example

```
void foo(double * a, double * b, double * c, double * d, double * e, int n) {
    for(int i = 0; i < n; i++)
                a[i] *= (b[i] - d[i]) * (c[i] + e[i]);
}

void goo(double * a, double * b, double * c, double * d, double * e, int n) {
    a[0:n] *= (b[0:n] - d[0:n]) * (c[0:n] + e[0:n]);
}
```

```
icl -Qvec-report3  -c test-array-notations.cpp
test-array-notations.cpp(2) (col. 2): remark: loop was not vectorized: existence of vector dependence.
test-array-notations.cpp(3) (col. 3): remark: vector dependence: assumed FLOW dependence between a line 3 and e line 3.
<…>
Test-array-notations.cpp(7) (col. 6): remark: LOOP WAS VECTORIZED.
```

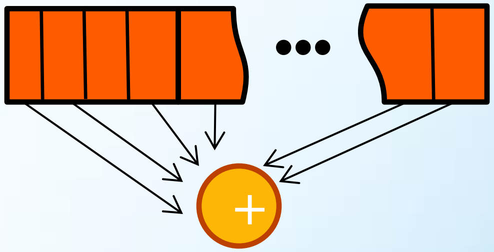
Why Data Parallel Extension to C/C++

- Current parallel hardware offers
  - Data Level Parallelism, e.g. SIMD
  - Thread Level Parallelism, e.g. TBB/Cilk/OpenMP
- Need for natural expression of DLP:
  - Language must make it easy for developer to exploit maximum hardware performance potential
  - Need for data parallel extensions for familiar C/C++
  - Natural mapping between data parallel constructs and underlying parallel hardware to achieve high performance and utilization.
  - Let compiler do the heavy lifting!

Reductions

- Reduction combines array section elements to generate a scalar result

```
int a[] = {1,2,3,4};
sum = __sec_reduce_add(a[:]); // sum
                              // is 10
```



- Some 10 built-in reduction functions supporting basic C data-types:
  - add, mul, max, max_ind, min, min_ind, all_zero, all_non_zero, any_nonzero

- Supports user-defined reduction functions:

```
    type fn(type in1, type in2); // scalar reduction
    function
    out = __sec_reduce(fn, identity_value, in[x:y:z]);
```

Function Maps

- A scalar function call is mapped to the elements of array section parameters by default:

```
a[:] = sin(b[:]);
a[:] = pow(b[:], c); // b[:]**c
a[:] = pow(c, b[:]); // c**b[:]
a[:] = foo(b[:])     // user defined foo()
```

  - Functions are mapped in parallel
  - No specific order on side effects
  - Compiler generates calls to vectorized library functions
  - Compiler may generate vectorized function body for function declared as __declspec(vector)

Function Maps with Array Sections
"Elemental Functions"

- Compiler can convert a user-supplied scalar function to vector function, when called with array notation arguments

Serial code

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

```
// Plain C scalar function declared with __declspec(vector)
__declspec(vector) float saxpy (float a, float x, float y)
{
            return (a * x + y);
}

Z[:] = saxpy(A, X[:], Y[:]); // Call scalar function with
                             // array notation parameters
```

- Compiler automatically maps the function across multiple array elements (in example, the function becomes "a * x[:] + y[:]")

Reducer Views

when there is no steal

```
cilk::reducer_opadd<int> sum(3);
```

3

initial view

```
void f()
{
   cilk_spawn g();
   work
   sum += 2;
   work
   cilk_sync;
   work
}
```

```
void g()
{
   work
   sum++;
   work
}
```

4

6

Worker A

Same behavior as serial execution!

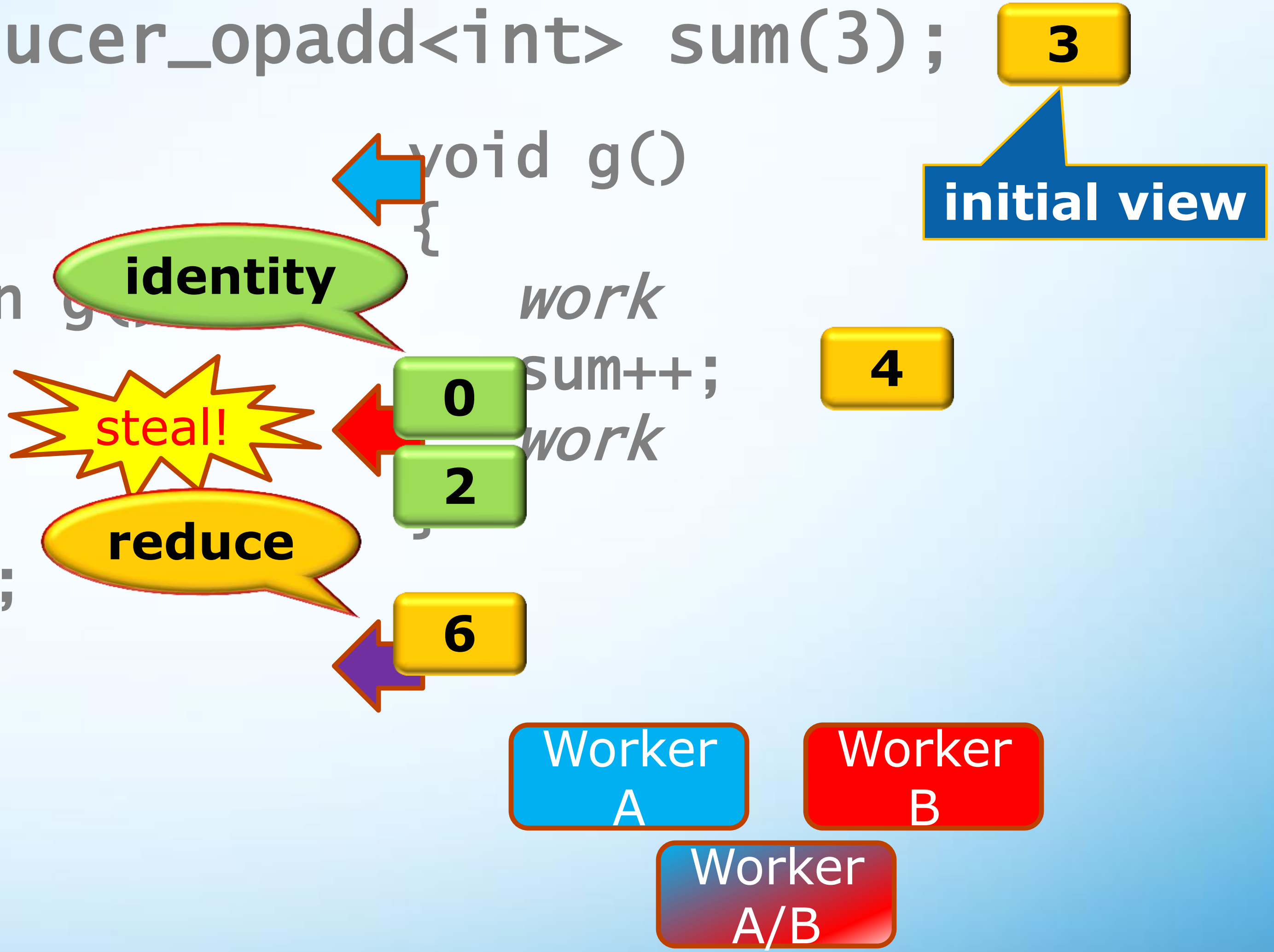
Reducer Views

when continuation is stolen

```
cilk::reducer_opadd<int> sum(3);
```

```
void f()
{
   cilk_spawn g(
   work
   sum += 2;
   work
   cilk_sync;
   work
}
```

```
void g()
{
   work
   sum++;
   work
}
```



Reducer Library

Cilk’s hyperobject library contains many commonly used reducers:

- `reducer_list_append`
- `reducer_list_prepend`
- `reducer_max`
- `reducer_max_index`
- `reducer_min`
- `reducer_min_index`
- `reducer_opadd`
- `reducer_ostream`
- `reducer_basic_string`
- …

You can also write your own using `cilk::monoid_base` and `cilk::reducer`.

Creating Your Own Reducer

1. Choose a type, a reduction operation, and an identity value.
   - This combination is known as a *monoid*
   - The reduction operation must be associative, but need not be commutative.
2. Declare a monoid class derived from `cilk::monoid_base`.
3. Instantiate `cilk::reducer` with your monoid.
4. Optionally create a wrapper class for type safety.

Algebraic Monoids

Definition. A *monoid* is a triple (T, ⊗, e), where

- T is a set,
- ⊗ is an associative binary operator,
- e is an identity for ⊗.

Associative

$a \otimes (b \otimes c) = (a \otimes b) \otimes c$

Identity

$a \otimes e = e \otimes a = a$

Examples:

- (int, +, 0)
- (double, *, 1)
- (bool, &&, true)
- (std::list, concat, empty)
- (int, max, INT_MIN)

Representing Monoids

In Cilk, we represent a monoid over `T` by a C++ class `M` that inherits from `cilk::monoid_base<T>` and defines

- a static or const member function `reduce()` that implements the binary operator ⊗ and
- a static or const member function `identity()` that constructs a fresh identity, e.

Example:

```
struct sum_monoid : cilk::monoid_base<int> {
  static void reduce(int* left, int* right) {
    *left += *right; // store result into *left
  }
  static void identity(int* p) {
    new (p) int(0); // placement construction
  }
};
```

Defining a Reducer

A reducer over `sum_monoid` may now be defined as follows:

```
cilk::reducer<sum_monoid> x;
```

The local view of `x` can be accessed as `x()`.

```
x() += value;
```

Reducer Wrappers

Issues

- It is generally inconvenient to replace every access to `x` in a legacy code base with `x()`.
- Accesses to `x` are not safe. Nothing prevents a programmer from writing "`x() *= 2`", even though the reducer is defined over `+`.

➡ A wrapper class solves these problems.

```
class sum_reducer {
  cilk::reducer<sum_monoid> r;
public:
  sum_reducer(int v = 0) : r(v) { }
  sum_reducer& operator+=(int v) { r() += v; }
  sum_reducer& operator++() { r()++; }
  int get_value() const { return r(); }
  ...
};
```

A Taste of Reducers in C

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

long sum_array(long data[], size_t n)
{
   CILK_C_REDUCER_OPADD(r, int, 0);
   CILK_C_REGISTER_REDUCER(r);

   cilk_for (int j = 0; j < n; ++j)
      REDUCER_VIEW(*r) += data[j];

   CILK_C_UNREGISTER_REDUCER(r);
   return r.value;
}
```

declare opadd reducer over int with initial value 0

register reducer with runtime system

look up reducer view

unregister from runtime

In serial code, reducer's final value is accessible, even after reducer has been unregistered.

Macros and idioms are needed to make up for C's lack of templates and overloading

Intel® Cilk™ Plus Elemental Functions

- The compiler can't assume that user-defined functions are safe for vectorization
- Now you can make your function an elemental function which indicates to the compiler that such a function can be applied to multiple elements of an array in parallel safely.
- Specify __declspec(vector) on both function declarations and definitions as this will affect name-mangling.

Intel® Cilk™ Plus Elemental Functions Example

```
double user_function(double x);
__declspec(vector) double elemental_function(double x);

void foo(double *a, double *b, int n) {
   a[0:n] = user_function(b[0:n]);
}

void goo(double *a, double *b, int n) {
   a[0:n] = elemental_function(b[0:n]);
}
```

```
icl /Qvec-report3 /c test-elemental-functions.cpp
test-elemental-functions.cpp(4) (col. 39): remark: routine skipped: no vectorization
   candidates.
test-elemental-functions.cpp(9) (col. 2): remark: LOOP WAS VECTORIZED.
```

Pragma SIMD

- Write a C/C++/FTN standard loop, add a pragma to get the compiler to vectorize it

- The compiler does not prove equivalence to sequential loop, no performance heuristics

- The programmer may need to provide additional clauses for correct code generation
  - Private, reduction, scalar

- Elemental functions can be called from the loop

```
#pragma simd
for (j = 0; j < N; j++) {
   a[j] = my_ef(b[j]);
}
```

Invoking Elemental Functions

Construct	Example	Semantics

Standard for loop	for (j = 0; j < N; j++) {
	a[j] = my_ef(b[j]);
	}
		Single thread, auto vectorization

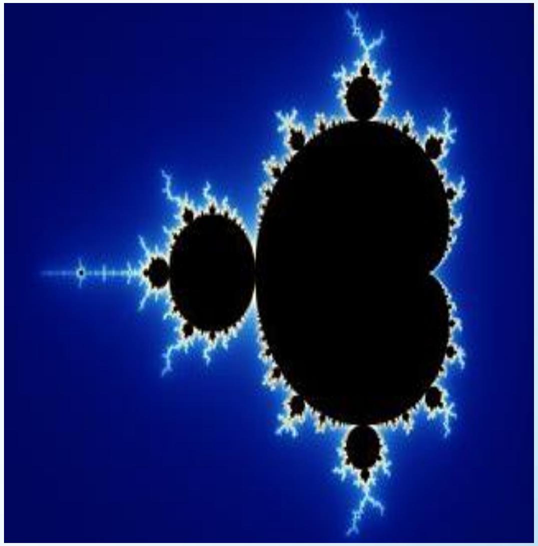
#pragma simd	#pragma simd
	for (j = 0; j < N; j++) {
	a[j] = my_ef(b[j]);
	}
		Single thread, Guaranteed to use the vector version

cilk for loop	cilk_for (j = 0; j < N; j++) {
	a[j] = my_ef(b[j]);
	}
		Both vectorization and concurrent execution

Array notation	a[:] = my_ef(b[:]);	Vectorization. *[Concurrency not yet implemented in compiler]*

Mandelbrot in Cilk

```
int mandel(complex c, int max_count) {
    int count = 0; complex z = 0;
    for (int i = 0; i < max_count; i++) {
        if (abs(z) >= 2.0) break;
        z = z*z + c; count++;
    }
    return count;
}
```



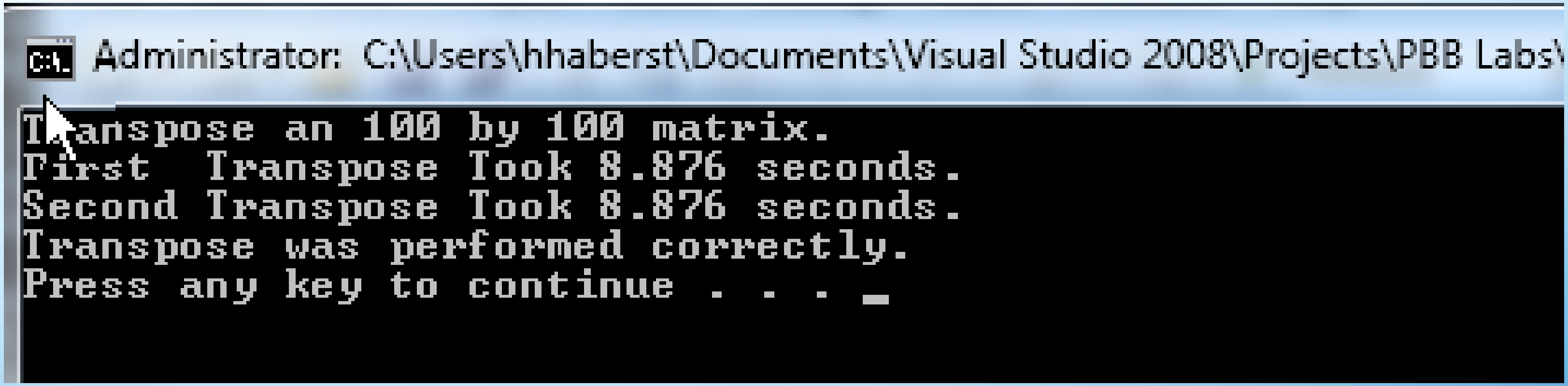
```
cilk_for (int i = 0; i < max_row; i++) {
    for (int j = 0; j < max_col; j++ ) {
        p[i][j] = mandel( complex(scale(i), scale(j)), depth);
}}
```

One line change to sequential version
Compiler support hides complexity

Cilk Plus – Sample

cilk_for, CEAN, Elemental Funtion

```
main()
   matrix_transpose()
      for()
         for() sum_squarroot()
               for()
                  for() sqrt()
               for()
                  for() subtract_squareroot()
```

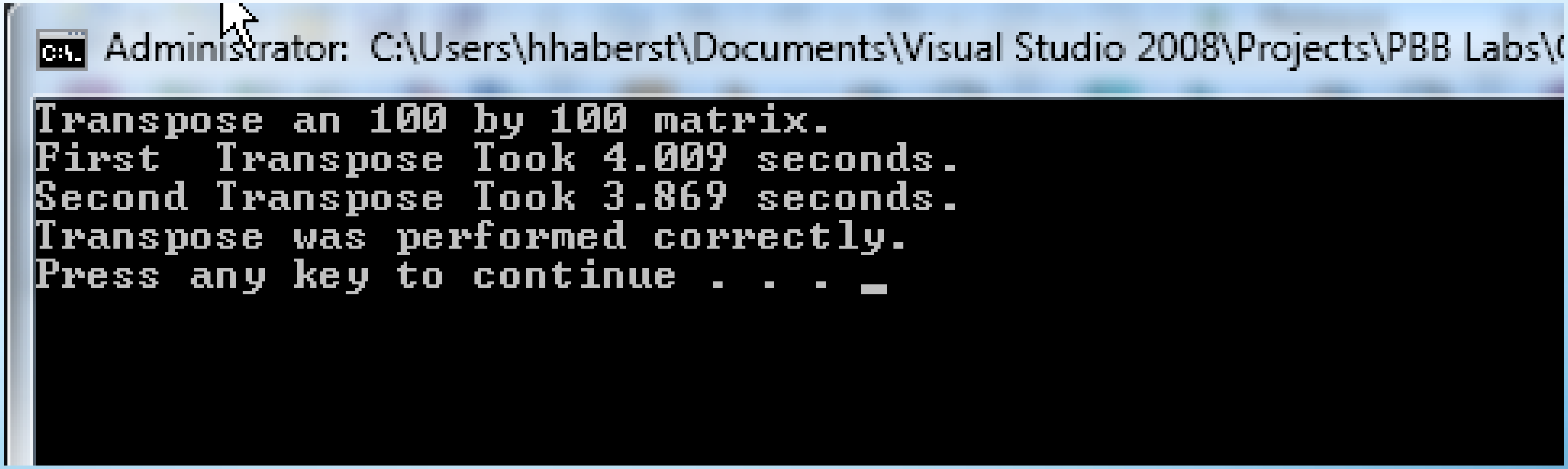


Administrator: C:\Users\hhaberst\Documents\Visual Studio 2008\Projects\PBB Labs\

```
Transpose an 100 by 100 matrix.
First  Transpose Took 8.876 seconds.
Second Transpose Took 8.876 seconds.
Transpose was performed correctly.
Press any key to continue . . . _
```

Cilk Plus – Sample

```
matrix_transpose()
    cilk_for()                      // outer loop data independent
      for() sum_squarroot()         // identified as hotspot
          for()
            for() sqrt()
          for()
            for() subtract_squareroot()
```

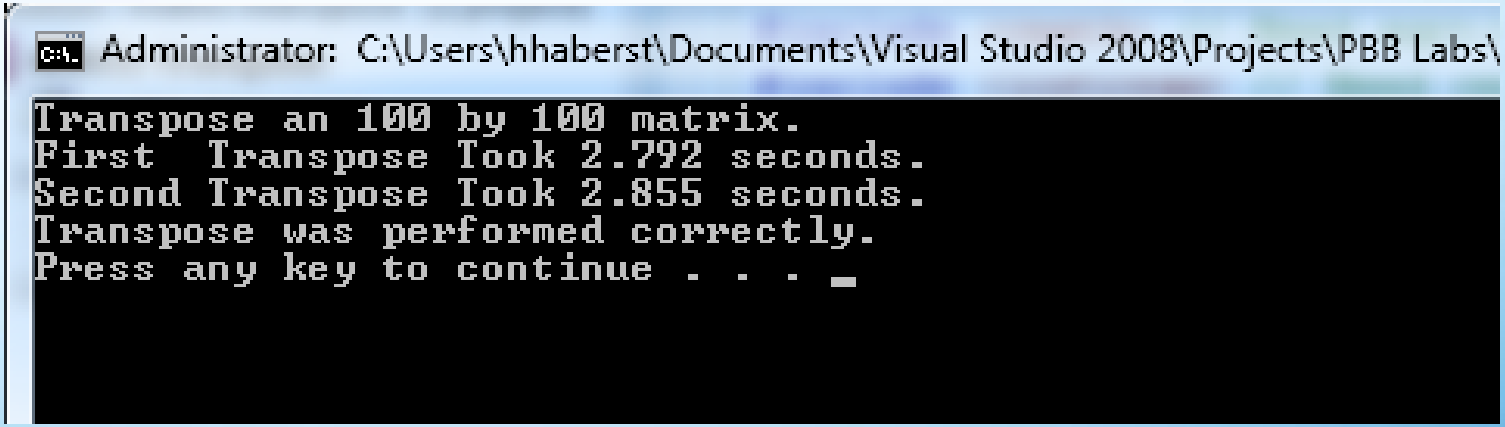


Administrator: C:\Users\hhaberst\Documents\Visual Studio 2008\Projects\PBB Labs\(

```
Transpose an 100 by 100 matrix.
First  Transpose Took 4.009 seconds.
Second Transpose Took 3.869 seconds.
Transpose was performed correctly.
Press any key to continue . . . _
```

Cilk Plus – Sample

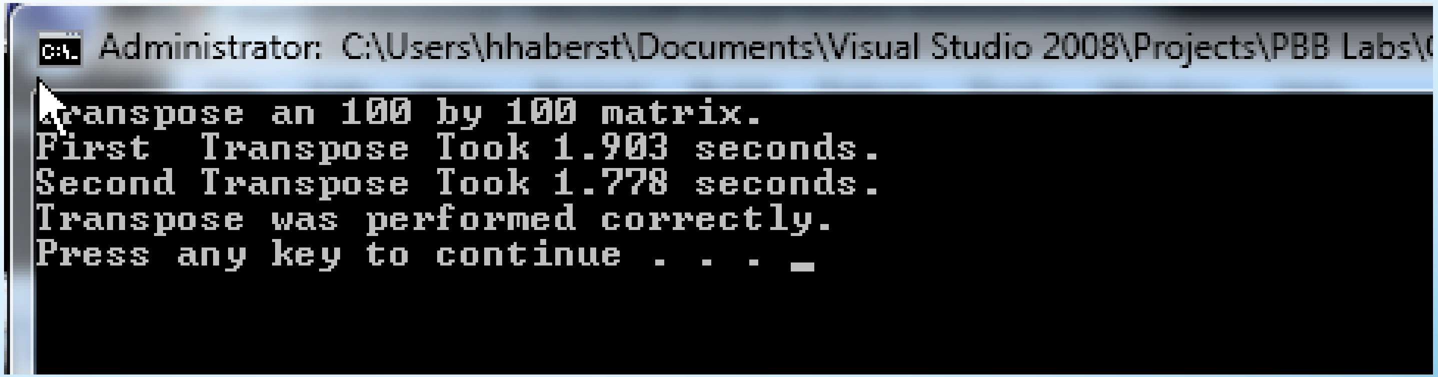
```
matrix_transpose()
    cilk_for()                                  // outer loop data independent
      for() sum_squarroot()                     // identified as hotspot
          for()
              x[0:size] += sqrt ()              // replaced with array notation
          for()
            for() subtract_squareroot()
```


Administrator: C:\Users\hhaberst\Documents\Visual Studio 2008\Projects\PBB Labs\

```
Transpose an 100 by 100 matrix.
First  Transpose Took 2.792 seconds.
Second Transpose Took 2.855 seconds.
Transpose was performed correctly.
Press any key to continue . . . _
```

Cilk Plus – Sample

```
matrix_transpose()
        cilk_for()                              // replaced with cilk_for
            for() sum_squarroot()
                for()
                    x[0:size] += sqrt ()        // replaced with array notation
                for()
                  for() __declspec(vector)subtract_squareroot()
                                                // added elemental function
```



Administrator: C:\Users\hhaberst\Documents\Visual Studio 2008\Projects\PBB Labs\(

```
Transpose an 100 by 100 matrix.
First  Transpose Took 1.903 seconds.
Second Transpose Took 1.778 seconds.
Transpose was performed correctly.
Press any key to continue . . . _
```

Cilk Plus – Sample

cilk_spawn

```
int  fib(int n)
{
    int x, y;

    if (n < 2) return n;

    x = cilk_spawn fib(n–1);
    y = fib(n–2);          // Execution can continue while
                           // fib(n–1) is running)
    cilk_sync;             // Asynchronous call must
                           //complete before using x.
    return x+y;
}
```

Cilk Plus – Sample

Hyperobjekt

```
int result = 0;
    for (std::size_t i = 0; i < n; ++i)
    {
        result += compute(myArray[i]);
    }
    std::cout << "The result is: " << result << std::endl;
```

Cilk Plus – Sample

Hyperobjekt

```
int result = 0;
    cilk_for (std::size_t i = 0; i < n; ++i)
    {
        result += compute(myArray[i]); // data race!!!
    }
    std::cout << "The result is: " << result << std::endl;
```

Cilk Plus – Sample

Hyperobjekt

```
cilk::reducer_opadd<int> result;
    cilk_for (std::size_t i = 0; i < ARRAY_SIZE; ++i)
    {
        result += compute(myArray[i]); // reducer hyperobject
                                       // avoids race
    }
    std::cout << "The result is: "
              << result.get_value()    //value extracted
              << std::endl;
```

References and Contact Information

- Intel® Cilk™ Plus
  - www.cilk.com
  - http://software.intel.com/en-us/intel-cilk-plus
  - http://software.intel.com/en-us/forums/intel-cilk-plus/
- Intel® Threading Building Blocks
  - https://www.threadingbuildingblocks.org/
  - http://software.intel.com/en-us/intel-tbb
  - http://software.intel.com/en-us/forums/intel-threading-building-blocks
- Parallel Programming Basics
  - http://software.intel.com/en-us/courseware-parallel-programming-basics

Intel® Threading Building Blocks (TBB)

Extens C++ for Parallelism

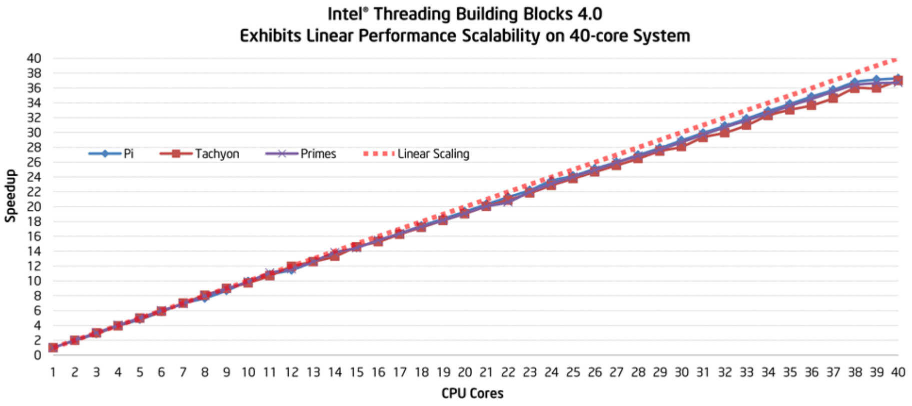
- A kind of „STL for Parallel C++ Programming“
- You specify *tasks* (that can run concurrently) instead of threads
  - Library maps user-defined logical tasks onto physical threads, efficiently using cache and balancing load
  - Full support for *nested parallelism*
- Targets threading for *scalable performance*
  - Portable across Linux*, Mac OS*, Windows*, and Solaris*
- *Compatible* with other threading packages
  - Can be used in concert with native threads and OpenMP*

➔ Flexible, scalable solution with high amount of control at minimum overhead

What’s New: Intel® Threading Building Blocks 4.0

Commercially Supported Code using Intel® TBB Scales Exceptionally Well

- **Flow Graph**
  - API Extends applicability of Intel® TBB to event-driven/reactive programming models
- **Concurrent Unordered Set**
  - Thread-safe container to store and access user objects
- **Memory Pools**
  - Enables greater flexibility and performance by getting thread-safe and scalable object allocation
- **Generic GCC* Atomics Support**
  - Library portability enables development of Intel® TBB-based solutions on a broader range of platforms
- **Improved Interoperability**
  - Arbitrary nesting in Intel® PBB components enabling Intel® Cilk™ Plus users to take advantage of composability



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

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

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

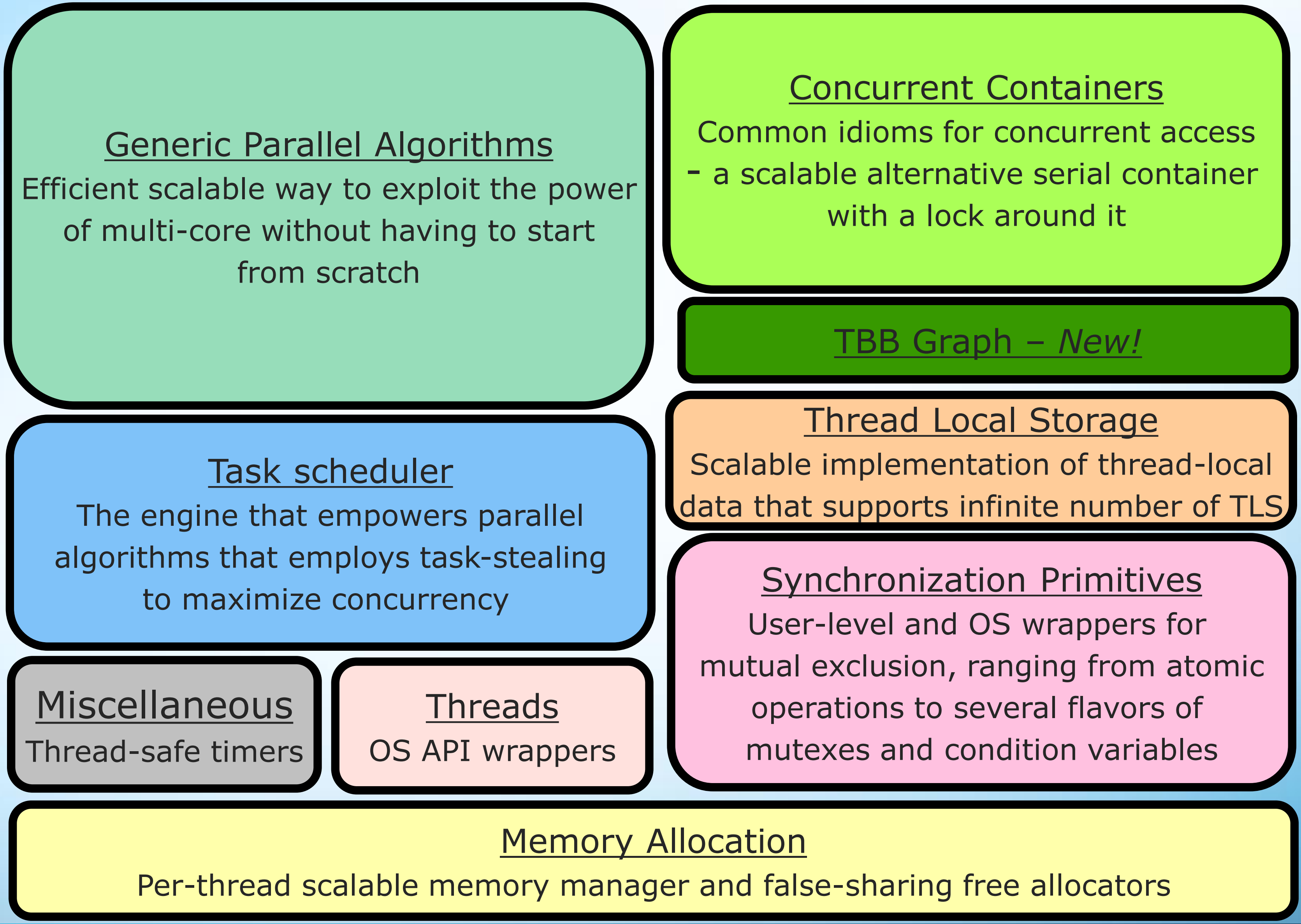
Optimization Notice: Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability,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

What is TBB product wise ?

- Intel® Threading Building Blocks (Intel® TBB) is a production C++ library that simplifies threading for performance.
- Not a new language or extension; works with off-the-shelf C++ compilers.
- Proven to be portable to new compilers, operating systems, and architectures.
- Available as commercial and an open-source version - GPLv2 (plus runtime exception)

http://threadingbuildingblocks.org

Intel® Threading Building Blocks



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

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

TBB Graph – *New!*

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

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

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

Miscellaneous
Thread-safe timers

Threads
OS API wrappers

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

Generic Parallel Algorithms

Loop parallelization

parallel_for

parallel_reduce

- load balanced parallel execution
- fixed number of independent iterations

parallel_scan

- computes parallel prefix

  y[i] = y[i-1] op x[i]

Parallel sorting

parallel_sort

Parallel function invocation

parallel_invoke

- Parallel execution of a number of user-specified functions

Parallel Algorithms for Streams

parallel_do

- Use for unstructured stream or pile of work
- Can add additional work to pile while running

parallel_for_each

- parallel_do without an additional work feeder

pipeline / parallel_pipeline

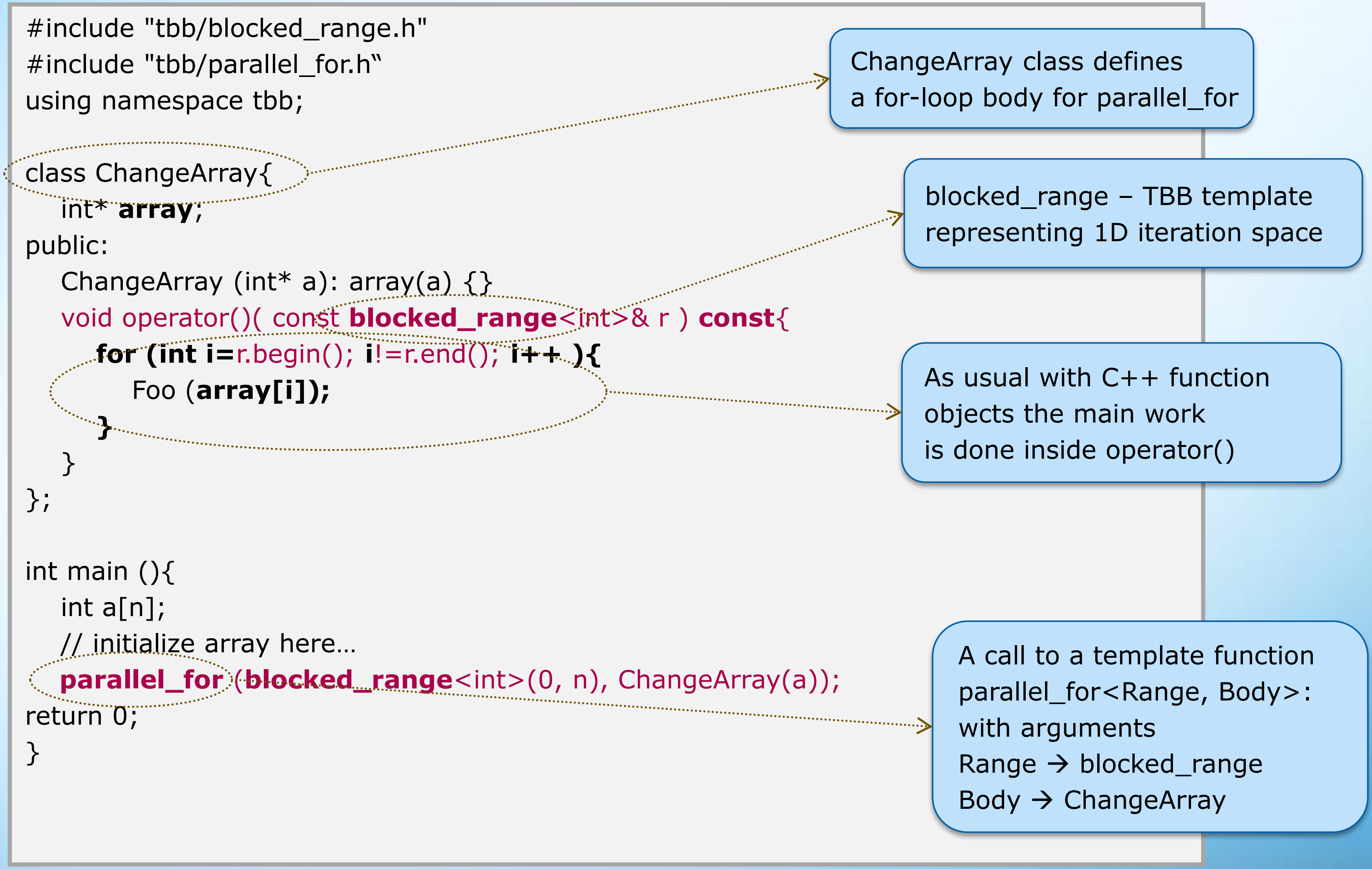
- Linear pipeline of stages
- Each stage can be parallel or serial in-order or serial out-of-order.
- Uses cache efficiently

Computational graph

flow::graph

- Implements dependencies between tasks
- Pass messages between tasks

parallel_for usage example



```
#include "tbb/blocked_range.h"
#include "tbb/parallel_for.h‟
using namespace tbb;

class ChangeArray{
    int* array;
public:
    ChangeArray (int* a): array(a) {}
    void operator()( const blocked_range<int>& r ) const{
        for (int i=r.begin(); i!=r.end(); i++ ){
            Foo (array[i]);
        }
    }
};

int main (){
    int a[n];
    // initialize array here…
    parallel_for (blocked_range<int>(0, n), ChangeArray(a));
return 0;
}
```

ChangeArray class defines a for-loop body for parallel_for

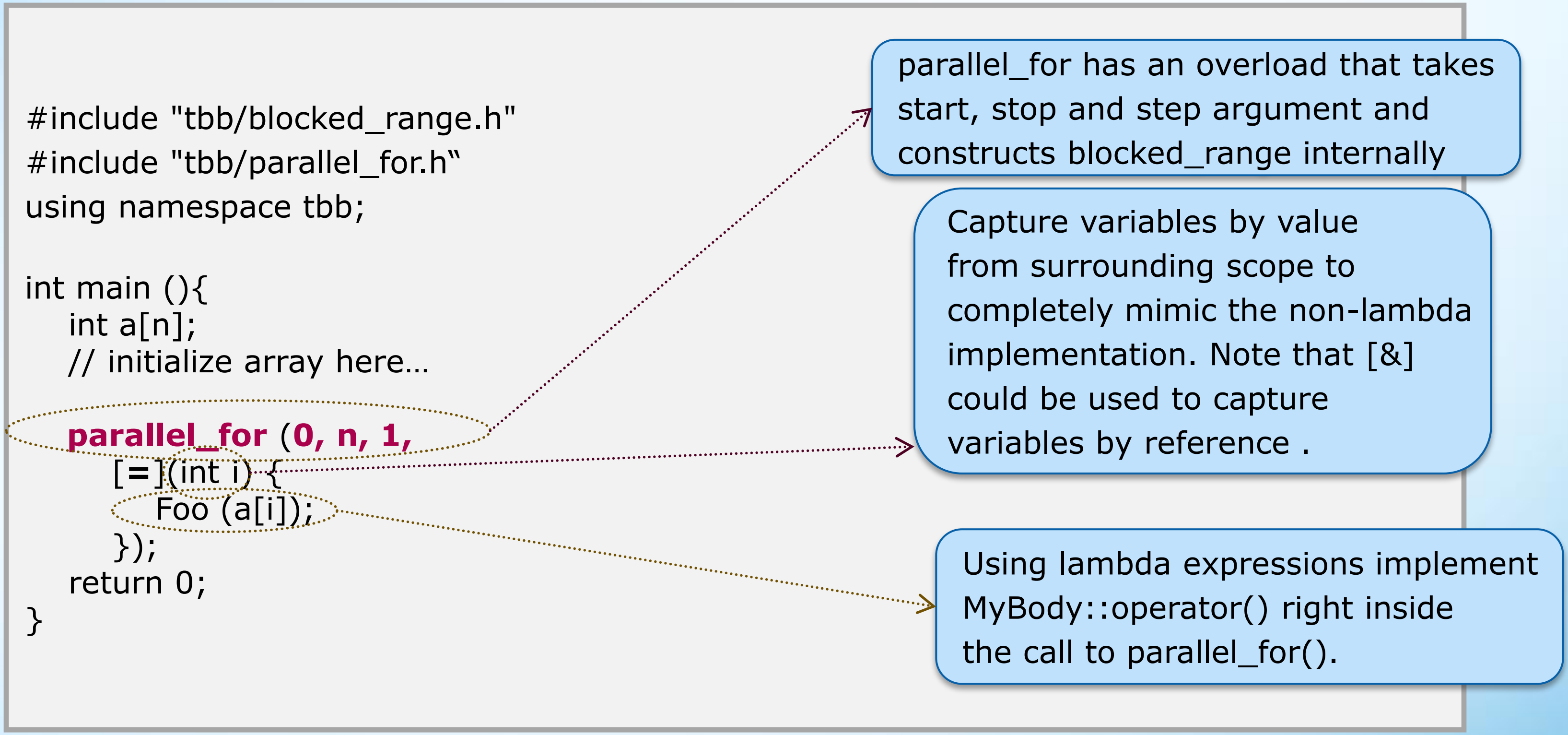
blocked_range – TBB template representing 1D iteration space

As usual with C++ function objects the main work is done inside operator()

A call to a template function parallel_for<Range, Body>: with arguments
Range → blocked_range
Body → ChangeArray

C++ 11 Lambda Expression Support

parallel_for example will transform into:



```
#include "tbb/blocked_range.h"
#include "tbb/parallel_for.h“
using namespace tbb;

int main (){
    int a[n];
    // initialize array here…

    parallel_for (0, n, 1,
        [=](int i) {
            Foo (a[i]);
        });
    return 0;
}
```

parallel_for has an overload that takes start, stop and step argument and constructs blocked_range internally

Capture variables by value from surrounding scope to completely mimic the non-lambda implementation. Note that [&] could be used to capture variables by reference .

Using lambda expressions implement MyBody::operator() right inside the call to parallel_for().

- *auto_partitioner* is used by default for all parallel algorithms (need to really focus on simple_partitioner if required
- explicit *task_scheduler_init* optional

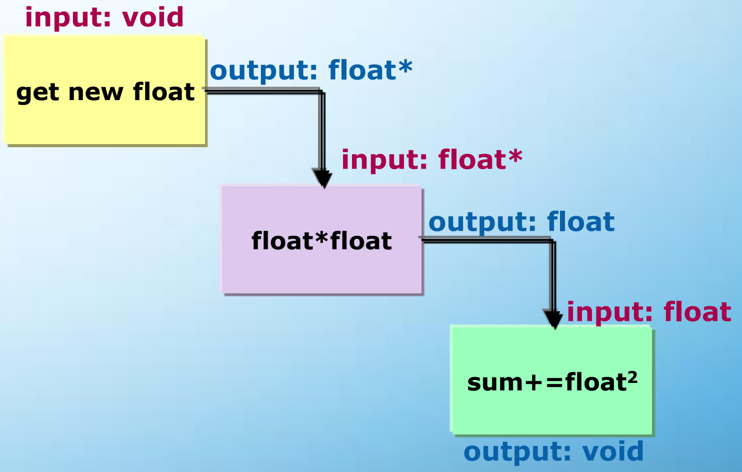
parallel_pipeline

```
float RootMeanSquare( float* first, float* last ) {
    float sum=0;
    parallel_pipeline( /*max_number_of_tokens=*/16,
      make_filter<void,float*>(
        filter::serial,
        [&](flow_control& fc)->float*{
            if( first<last ) {
                return first++;
            } else {
                fc.stop(); // stop processing
                return NULL;
            }
        }
      ) &
      make_filter<float*,float>(
        filter::parallel,
        [](float* p) ->float* {return (*p)*(*p);}
      ) &
      make_filter<float,void>(
        filter::serial,
        [&sum](float x) {sum+=x;}
      )
    );
    // sum=first²+(first+1)² + … +(last-1)² computed in parallel
    return sqrt(sum);
}
```

Call function *tbb::parallel_pipeline* to run pipeline stages (filters)

Create pipeline stage object *tbb::make_filter*<InputDataType, OutputDataType>(mode, body)

Pipeline stage mode can be *serial*, *parallel*, *serial_in_order*, or *serial_out_of_order*



Concurrent Containers

- Intel® TBB provides highly concurrent containers
  - STL containers are not concurrency-friendly: attempt to modify them concurrently can corrupt container
  - Wrapping a lock around an STL container turns it into a serial bottleneck and still does not always guarantee thread safety
    - STL containers are inherently not thread-safe
- Intel TBB provides fine-grained locking or lockless implementations
  - Worse single-thread performance, but better scalability.
  - Can be used with the library, OpenMP*, or native threads.

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

Concurrent Containers Key Features

- **concurrent_hash_map <Key, T, HashCompare, Allocator>**
  - Models hash table of std::pair <const *Key*, *T*> elements
  - Maps *Key* to element of type *T*
  - User defines *HashCompare* to specify how keys are hashed and compared
  - tbb_allocator is a default allocator
- **concurrent_vector <T, Allocator>**
  - Dynamically growable array of *T*: grow_by and grow_to_atleast
  - cache_aligned_allocator is a default allocator
- **concurrent_queue <T, Allocator>**
  - For single threaded run concurrent_queue supports regular "first-in-first-out" ordering
  - If one thread pushes two values and the other thread pops those two values they will come out in the order as they were pushed
  - cache_aligned_allocator is a default allocator
- **concurrent_bounded_queue <T, Allocator>**
  - Similar to concurrent_queue with a difference that it allows specifying capacity. Once the capacity is reached 'push' will wait until other elements will be popped before it can continue.

Example: concurrent_hash_map

```
struct wordsCompare {
    bool equal(const string& w1, const string& w2) const {
        return w1 == w2;
    }
    size_t hash(const string& w) const {
        size_t h = 0; for( const char* s = w.c_str(); *s; s++ ) h = (h*16777179)^*s;
        return h;
    }
};

void ParallelWordsCounting(const text_t& text) {

    parallel_for( blocked_range<size_t>( 0, text.size() ),
        [&text]( const blocked_range<int> &r ) {
            for(int i = r.begin(); i < r.end(); ++i) {
                concurrent_hash_map<string, int, wordsCompare>::accessor acc;
                wordCounters.insert(acc, text[i]);
                acc->second++;
            }
        });
}
```

User-defined "HashCompare" class needs to implement functions for comparing two keys and a hashing function

an element of a concurrent_hash_map can be accessed by creating an "accessor" object, which is somewhat a smart pointer implementing the necessary data access synchronization

Synchronization Primitives Features

- Atomic Operations.
  - High-level abstractions
- Exception-safe Locks
  - **spin_mutex** is VERY FAST in lightly contended situations; use it if you need to protect very few instructions
  - Use **queuing_rw_mutex** when scalability and fairness are important
  - Use **recursive_mutex** when your threading model requires that one thread can re-acquire a lock. All locks should be released by one thread for another one to get a lock.
  - Use reader-writer mutex to allow non-blocking read for multiple threads

Example: spin_rw_mutex

```
#include “tbb/spin_rw_mutex.h”
using namespace tbb;

spin_rw_mutex MyMutex;

int foo (){
    // Construction of ‘lock’ acquires ‘MyMutex’
    spin_rw_mutex::scoped_lock lock (MyMutex, /*is_writer*/ false);
    …
    if (!lock.upgrade_to_writer ()) { … }
    else { … }
    return 0;
    // Destructor of ‘lock’ releases ‘MyMutex’
}
```

- If exception occurs within the protected code block destructor will automatically release the lock if it’s acquired avoiding a dead-lock
- Any reader lock may be upgraded to writer lock; upgrade_to_writer indicates whether the lock had to be released before it was upgraded

Example: atomic operation

```
#include "tbb/blocked_range.h"
#include "tbb/parallel_for.h"
using namespace tbb;

atomic<int> sum;

int main (){
    int a[n];
    // initialize array here…

    parallel_for (0, n, 1,
        [=](int i) {
            Foo (a[i]);
            sum += a[i];
        });
return 0;
}
```

This operation is performed atomically

Optimization Notice

Scalable Memory Allocation

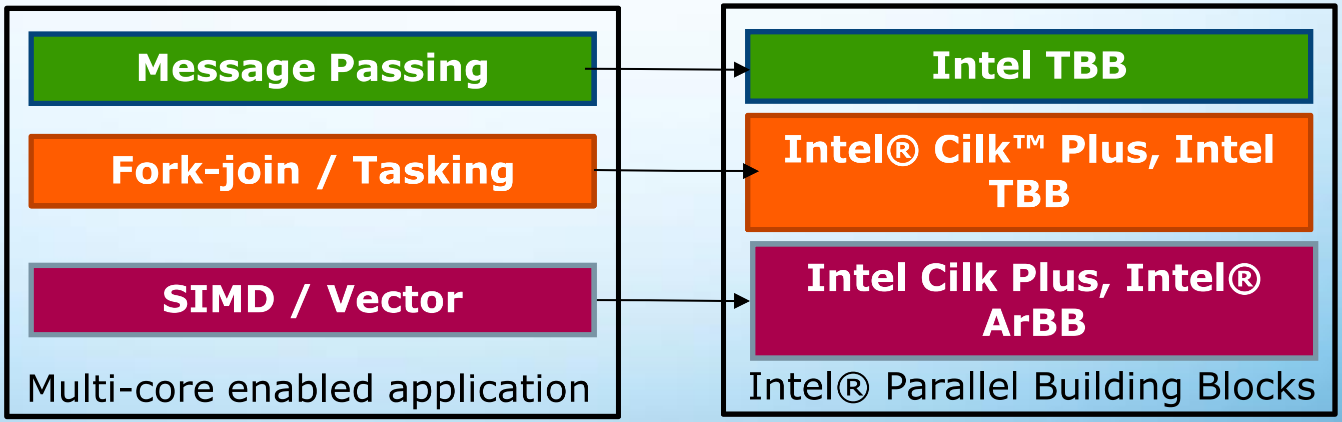
- Problem
  - Memory allocation is a bottle-neck in concurrent environment
    - Threads have to acquire a global lock to allocate/deallocate memory from the global heap
- Solution
  - Intel® Threading Building Blocks provides tested, tuned, and scalable memory allocator based on per-thread memory management algorithm
  - Scalable memory allocator has simple interface that can be used:
    - As an *allocator* argument to STL template classes
    - As a replacement for malloc/realloc/free calls (C programs)
    - As a replacement for global *new* and *delete* operators (C++ programs)
  - tbb_allocator automatically searches for scalable_allocator and uses it, if found. Otherwise it uses plain allocator.

Automatic malloc replacement

- On Windows and Linux operating systems it is possible to automatically replace calls to memory allocating functionality with corresponding Intel® TBB function calls;
- This is done by the use of malloc_proxy libraries:
  - Linux: libtbbmalloc_proxy.so.2 and libtbbmalloc_proxy_debug.so.2
  - Windows:tbbmalloc_proxy.dll and tbbmalloc_debug_proxy.dll

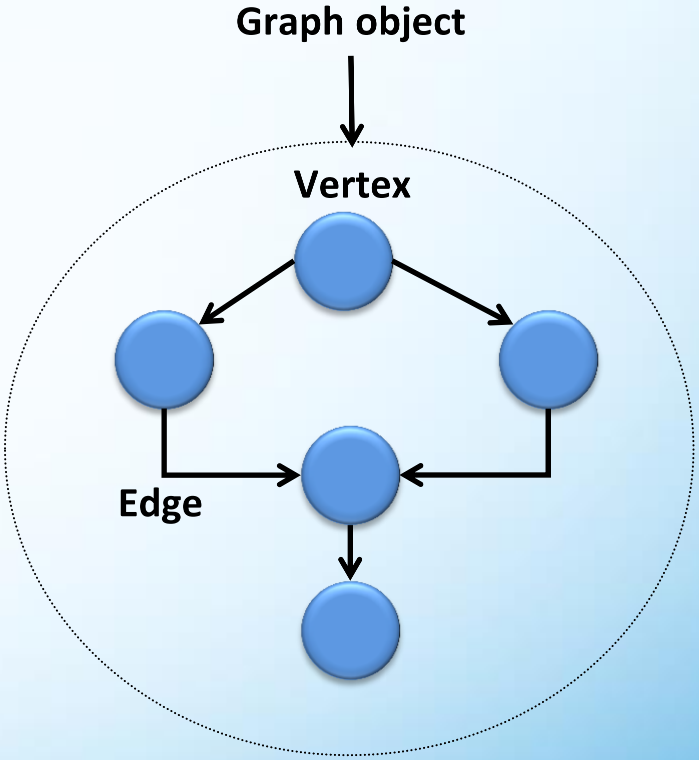
Intel® TBB Class Graph: Motivation

- Many applications can be naturally implemented as computational graphs
- Intel® TBB class graph allows to:
  - Implement dependencies between tasks
  - Pass messages between tasks
- Intel TBB class graph extends applicability of Intel® Parallel Building Blocks to reactive/event-driven parallel models


Message Passing
Fork-join / Tasking
SIMD / Vector
Multi-core enabled application
Intel TBB
Intel® Cilk™ Plus, Intel TBB
Intel Cilk Plus, Intel® ArBB
Intel® Parallel Building Blocks

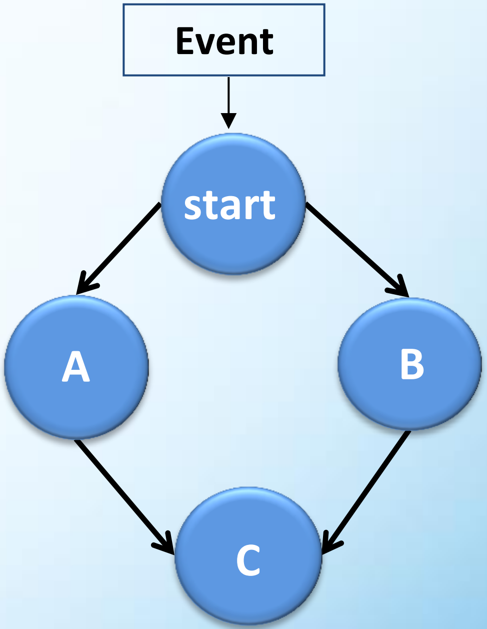
Intel® TBB class graph: Components

- Graph object
  - Contains a pointer to the root task
  - Owns tasks created on behalf of the graph
  - Users can wait for the completion of all tasks of the graph
- Graph nodes
  - Implement *sender* and/or *receiver* interfaces
  - Nodes manage messages and/or execute function objects
- Edges
  - Connect predecessors to successors



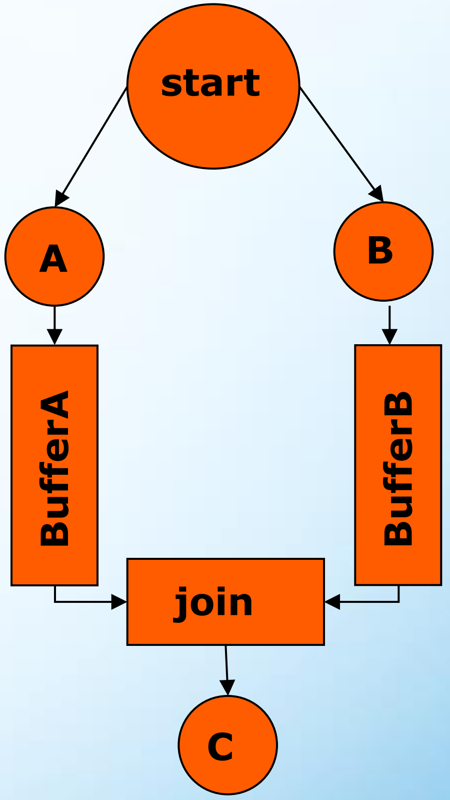
Example: simple dependency graph

```
#include <cstdio>
#define TBB_PREVIEW_GRAPH 1
#include "tbb/flow_graph.h"
using namespace tbb::flow;
struct body {
        std::string my_name;
        body( const char *name ) : my_name(name) {}
        void operator()( continue_msg ) const {
             printf("%s\n", my_name.c_str());
      }
};
int main() {
        graph g;      // graph object
        broadcast_node< continue_msg > start;       // create vertexes
        continue_node<continue_msg> a( g, body("A"));
        continue_node<continue_msg> b( g, body("B"));
        continue_node<continue_msg> c( g, body("C"));
        make_edge( start, a ); // connect vertexes
        make_edge( start, b );
        make_edge( a, c );
        make_edge( b, c );

        start.try_put( continue_msg() );     // start graph execution
        g.wait_for_all();         // wait for all task completion

        return 0;
}
```



Building A Simple Message Graph

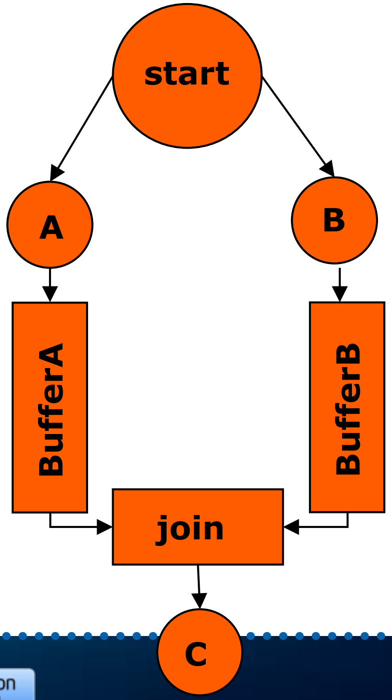
- Model: Tasks A and B start simultaneously and process input item T, task C executes after tasks A and B have finished and processes outputs of A and B
- Tools:
  - "start" node: tbb::broadcast_node<T>
    - Accepts messages of type T and sends them to all accepting successors
  - A, B, C nodes: tbb::function_node<T,U>
    - Accepts messages of type T
    - Executes function object
    - Messages of type U returned by a function object are sent to all accepting successors
  - bufferA, bufferB: tbb::buffer_node<T>
    - Buffers all incoming messages of type T and sends to 1 successor if possible
  - "join" node: tbb::join_node<T0, T1,…>
    - Attempts to reserve predecessors; if successful outputs tuple, else releases



Optimization Notice

Example: Simple Message Graph

```
tbb::graph g;

tbb::broadcast_node<int> start;
tbb::function_node<int, int> A( g, tbb::graph::serial, []( int x )->int {
          std::cout << "A" << std::endl;
          return x*x;
} );
tbb::function_node<int, int> B( g, tbb::graph::serial, []( int x )->int {
          std::cout << "B" << std::endl;
          return x*x*x;
} );
tbb::function_node<std::tuple<int,int>, int>  C( g, tbb::graph::serial, []( std::tuple<int,int> t )->int {
          std::cout << "C" << std::endl;
          return t.get<0>()+t.get<1>();
} );

tbb::buffer_node<int> bufferA( g );
tbb::buffer_node<int> bufferB( g );
tbb::join_node<int, int> join( g );

tbb::make_edge( start, A );
tbb::make_edge( start, B );
tbb::make_edge( A, bufferA );
tbb::make_edge( B, bufferB );
tbb::make_edge( bufferA, join.inputs(). get<0>() );
tbb::make_edge( bufferB, join.inputs(). get<1>() );
tbb::make_edge( join, C );

start.try_put( 12 );
g.wait_for_all();
```



Summary

- Class graph enables new usage models in Intel® Threading Building Blocks
  - Task dependencies and message passing
  - Event-driven programming
- Available now as a community preview feature
  - We are accepting feedback from users

TBB 4.0 recent features

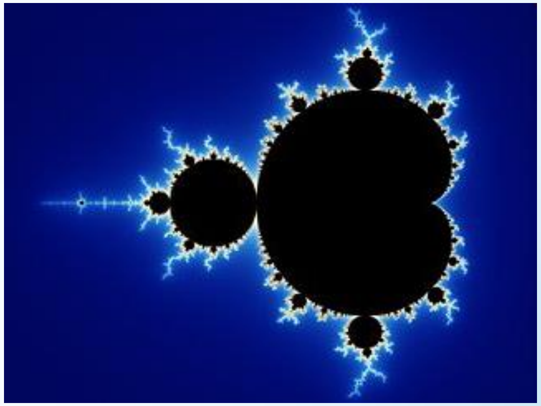
Feature name	Community preview feature	Official feature
Flow graph		●
Scalable Memory Pools	●	
Serial parallel_for	●	
concurrent_priority_queue		●
TBB runtime loader	●	
parallel_deterministic_reduce	●	
Task priorities		●

Recommendations

- Use parallel_pipeline when possible
  - Pipelines can be implemented via Intel® TBB graph but parallel_pipeline does it naturally and might work slightly faster
- Use Intel TBB graph for DAGs
  - The class task has API that supports DAGs, but with graph the implementation is more intuitive and elegant

Mandelbrot in Intel® TBB

```
int mandel(Complex c, int max_count) {
  int count = 0; Complex z = 0;
  for (int i = 0; i < max_count; i++) {
    if (abs(z) >= 2.0) break;
    z = z*z + c; count++;
  }
  return count;
```



Parallel algorithm

Task is a function object.

```
parallel_for( blocked_range<int>(0, max_row),
  [&](blocked_range<int> r> {
   for (size_t i=r.begin(); i!= r.end();++i)
     for (int j = 0; j < max_col; j++)
       p[i][j]=mandel(Complex(scale(i),scale(j)),depth);
});
```

Use new C++ lambda functions to define function object in-line

Intel® Threading Building Block uses C++ generics to provide parallel algorithms to the programmer

TBB – Sample

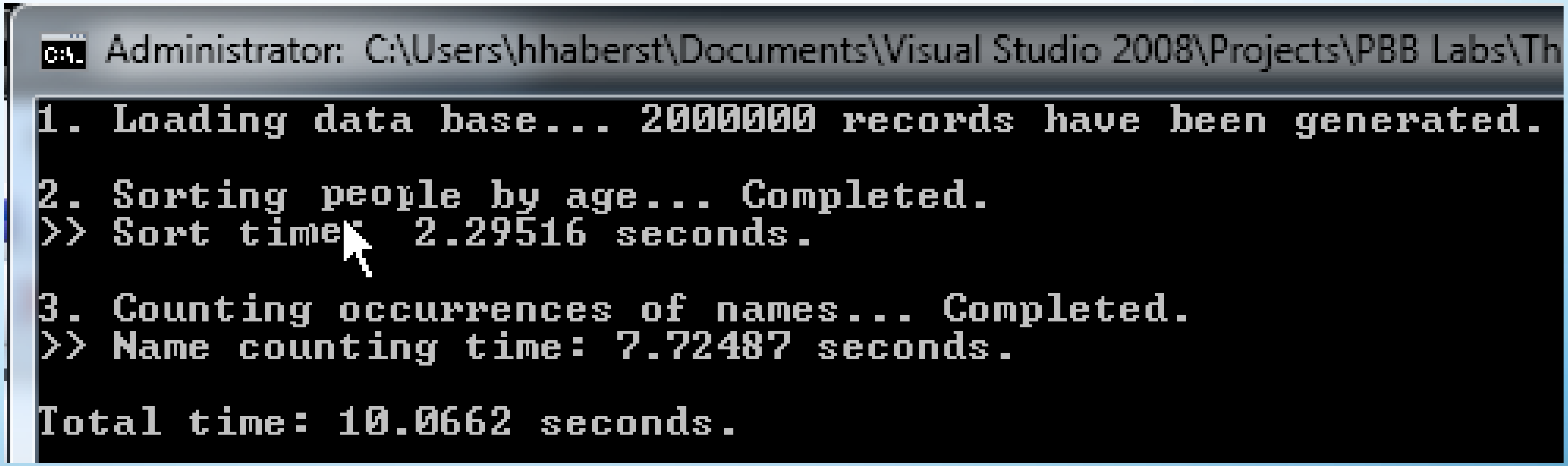
Parallel construct

```
main()
  GenerateData()
  sort()
    ComparePeopleByAges()
  CountNames()
    for()
  ValidateCounts()
    for()
```

TBB – Sample

Parallel construct

```
main()
   GenerateData()
   sort()
      ComparePeopleByAges()
   CountNames()
      CriticalSection
          parallel_for()   // parallel_for TBB template using lambda
   ValidateCounts()
      parallel_for()       // parallel_for TBB template using lambda
```

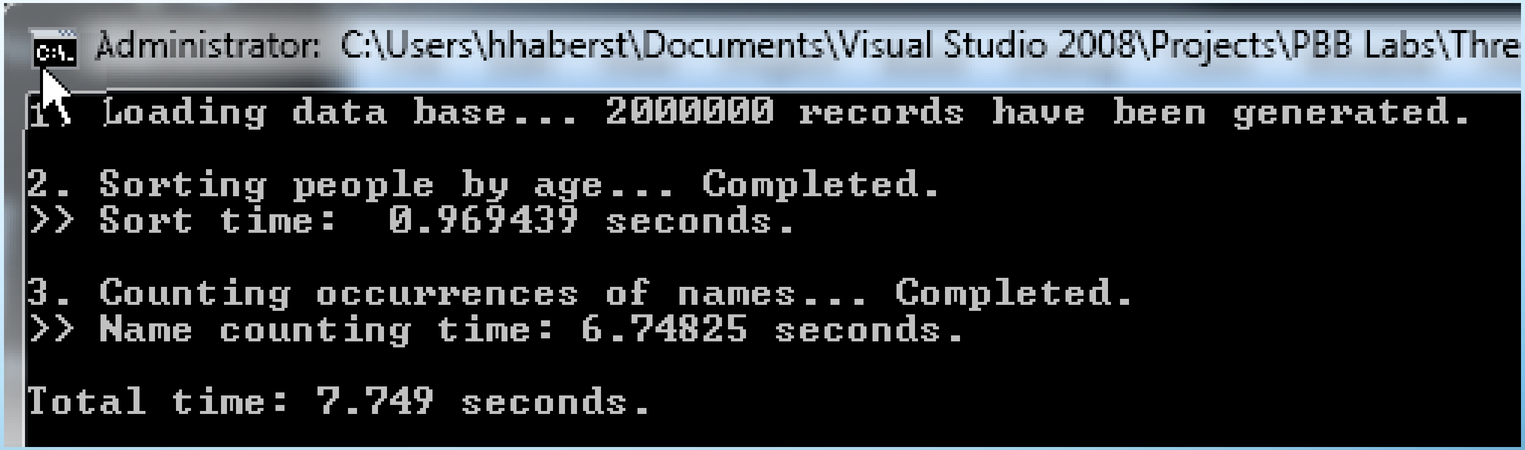

Administrator: C:\Users\hhaberst\Documents\Visual Studio 2008\Projects\PBB Labs\Th

```
1. Loading data base... 2000000 records have been generated.

2. Sorting people by age... Completed.
>> Sort time: 2.29516 seconds.

3. Counting occurrences of names... Completed.
>> Name counting time: 7.72487 seconds.

Total time: 10.0662 seconds.
```

TBB – Sample

STL replacment

```
main()
   GenerateData()
   parallel_sort()         // STL replacment with threaded TBB template
      ComparePeopleByAges()
   CountNames()
      CriticalSection
          parallel_for()   // parallel_for TBB template using lambda
   ValidateCounts()
      parallel_for()       // parallel_for TBB template using lambda
```



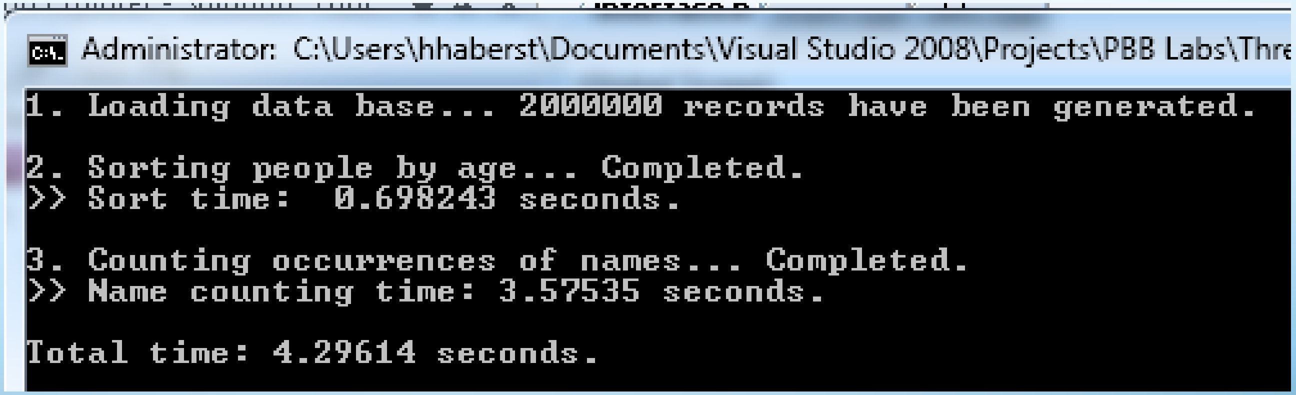
Administrator: C:\Users\hhaberst\Documents\Visual Studio 2008\Projects\PBB Labs\Thre

```
1. Loading data base... 2000000 records have been generated.

2. Sorting people by age... Completed.
>> Sort time:  0.969439 seconds.

3. Counting occurrences of names... Completed.
>> Name counting time: 6.74825 seconds.

Total time: 7.749 seconds.
```

TBB – Sample

Scalable Memory Allocator

```
main()
tbb_allocator()            //  scalable memory allocator replacing
                           // std::allocator()
  GenerateData()
  parallel_sort()          // STL replacment with threaded TBB template
    ComparePeopleByAges()
  CountNames()
     CriticalSection
       parallel_for()      // parallel_for TBB template using lambda
  ValidateCounts()
    parallel_for()         // parallel_for TBB template using lambda
```



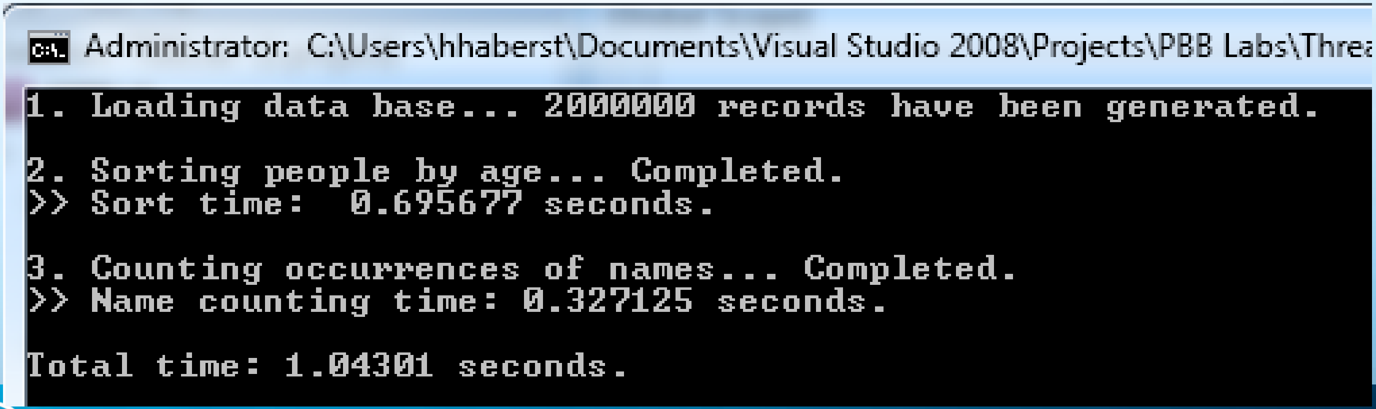
Administrator: C:\Users\hhaberst\Documents\Visual Studio 2008\Projects\PBB Labs\Thre

```
1. Loading data base... 2000000 records have been generated.

2. Sorting people by age... Completed.
>> Sort time:  0.698243 seconds.

3. Counting occurrences of names... Completed.
>> Name counting time: 3.57535 seconds.

Total time: 4.29614 seconds.
```

TBB – Sample

Concurrent Container

```
main()
tbb_allocator()
concurrent_unordered_map<MyString, atomic<int>>
                                   //  concurrent container, atomic operation on var
   GenerateData()
   parallel_sort()       // STL replacment with threaded TBB template
      ComparePeopleByAges()
   CountNames()
         parallel_for()   // parallel_for TBB template using lambda
   ValidateCounts()
      parallel_for()      // parallel_for TBB template using lambda
```

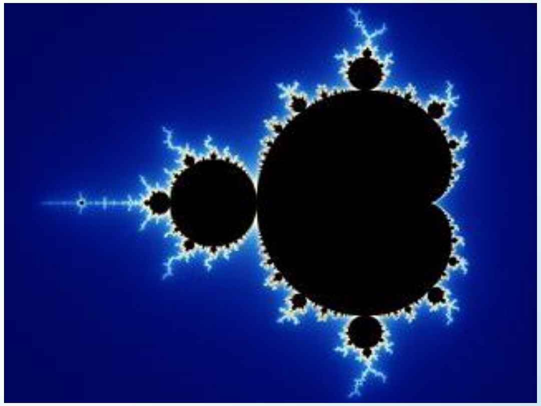


Administrator: C:\Users\hhaberst\Documents\Visual Studio 2008\Projects\PBB Labs\Threa

```
1. Loading data base... 2000000 records have been generated.

2. Sorting people by age... Completed.
>> Sort time:  0.695677 seconds.

3. Counting occurrences of names... Completed.
>> Name counting time: 0.327125 seconds.

Total time: 1.04301 seconds.
```

Mandelbrot in Intel® TBB

```
int mandel(Complex c, int max_count) {
  int count = 0; Complex z = 0;
  for (int i = 0; i < max_count; i++) {
    if (abs(z) >= 2.0) break;
    z = z*z + c; count++;
  }
  return count;
```



Parallel algorithm

Task is a function object.

```
parallel_for( blocked_range<int>(0, max_row),
  [&](blocked_range<int> r> {
    for (size_t i=r.begin(); i!= r.end();++i)
      for (int j = 0; j < max_col; j++)
        p[i][j]=mandel(Complex(scale(i),scale(j)),depth);
});
```

Use new C++ lambda functions to define function object in-line

Intel® Threading Building Block uses C++ generics to provide parallel algorithms to the programmer

What is TBB product wise ?

- Intel® Threading Building Blocks (Intel® TBB) is a production C++ library that simplifies threading for performance.
- Not a new language or extension; works with off-the-shelf C++ compilers.
- Proven to be portable to new compilers, operating systems, and architectures.
- Available as commercial and an open-source version - GPLv2 (plus runtime exception)

  http://threadingbuildingblocks.org

TBB Resources
New Features

Task priorities

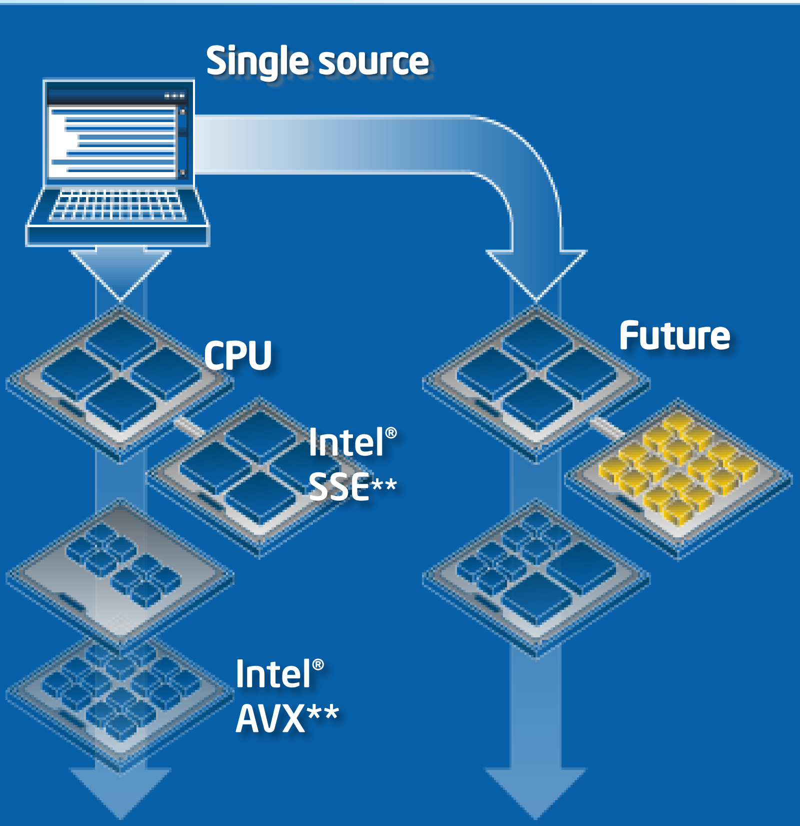
- http://software.intel.com/en-us/blogs/2011/04/01/task-and-task-group-priorities-in-tbb/

Graph (now called tbb::flow_graph)

- http://software.intel.com/en-us/blogs/2011/03/31/the-join-node-in-the-intel-threading-building-blocks-graph-community-preview-feature-2/
- http://software.intel.com/en-us/blogs/2011/01/10/using-the-intel-threading-building-blocks-graph-community-preview-feature-an-implementation-of-dining-philosophers/
- http://software.intel.com/en-us/blogs/2010/12/27/using-the-intel-threading-building-blocks-graph-community-preview-feature-creating-a-simple-dependency-graph/
- http://software.intel.com/en-us/blogs/2011/01/03/using-the-intel-threading-building-blocks-graph-community-preview-feature-creating-a-simple-message-graph/
- http://software.intel.com/en-us/blogs/2010/12/23/intel-threading-building-blocks-version-30-update-5-introduces-graph-as-a-community-preview-feature-2/

Intel® Array Building Blocks

A Research Project (whatif.intel.com)



Productivity

- Integrates with existing tools
- Applicable to many problem domains
- Safe by default: maintainable

Performance

- Efficient and scalable
- Harnesses both vectors and threads
- Eliminates modularity overhead of C++

Portability

- High-level abstraction
- Hardware independent
- Forward scaling

** Intel® Streaming SIMD Extensions
Intel® Advanced Vector Extensions

Outlook on Future Models to Express Parallelism

- What if Software were like this?
- Implementation of these models partially are available as preview features from http://whatif.intel.com
- Some of the models are pure research projects at Intel, e.g.
  - Intel® Array Building Blocks (Intel® ArBB)
    C++ library for parallel array operations
    Dynamic compilation allows architectural customization
    Array notation enabling parallel array operations
- These models might or might not make it into any of Intel's developer products

No decisions should be made regarding these feature being available in future products from Intel

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