PART 2 AND 3: INTEL® THREADING BUILDING BLOCKS AND ITS HETEROGENEOUS PROGRAMMING FEATURES

Pablo Reble

November 13, 2017

Agenda

Intel® Threading Building Blocks (Intel® TBB) and the flow graph interfaces

- What is Intel TBB and why is it relevant to HPC
- A brief introduction to Intel TBB and the flow graph

Using heterogeneous flow graph nodes to coordinate accelerators

- A deep dive in to the flow graph heterogeneous programming extensions
- Using Intel TBB beyond the CPU and integrated GPU

An Overview of Intel® Threading Building Blocks and its flow graph interfaces

Intel® Threading Building Blocks (Intel® TBB)

Celebrated its 10 year anniversary in 2016!

A widely used C++ template library for shared-memory parallel programming

What

Parallel algorithms and data structures
Threads and synchronization primitives
Scalable memory allocation and task scheduling

Benefits

Is a library-only solution that does not depend on special compiler support
Is both a commercial product and an open-source project
Supports C++, Windows*, Linux*, OS X*, Android* and other OSes
Commercial support for Intel® Atom™, Core™, Xeon® processors and for Intel® Xeon Phi™ coprocessors

http://threadingbuildingblocks.org http://software.intel.com/intel-tbb

Didn’t we solve the Threading problem in the 1990s?

Pthreads standard:	IEEE 1003.1c-1995
OpenMP* 1.0 standard:	1997

Yes, **but…**

- How to split up work?
- How to keep caches hot?
- How to balance load between threads?
- What about nested parallelism (call chain)?

Programming with threads is HARD

- Atomicity, ordering, and/vs. scalability
- Data races, dead locks, etc.

Threads are too low level a model.

What Do We Mean by “Task” ?

A piece of work represented by a (lambda) function and its captured arguments that we can run in parallel with other tasks

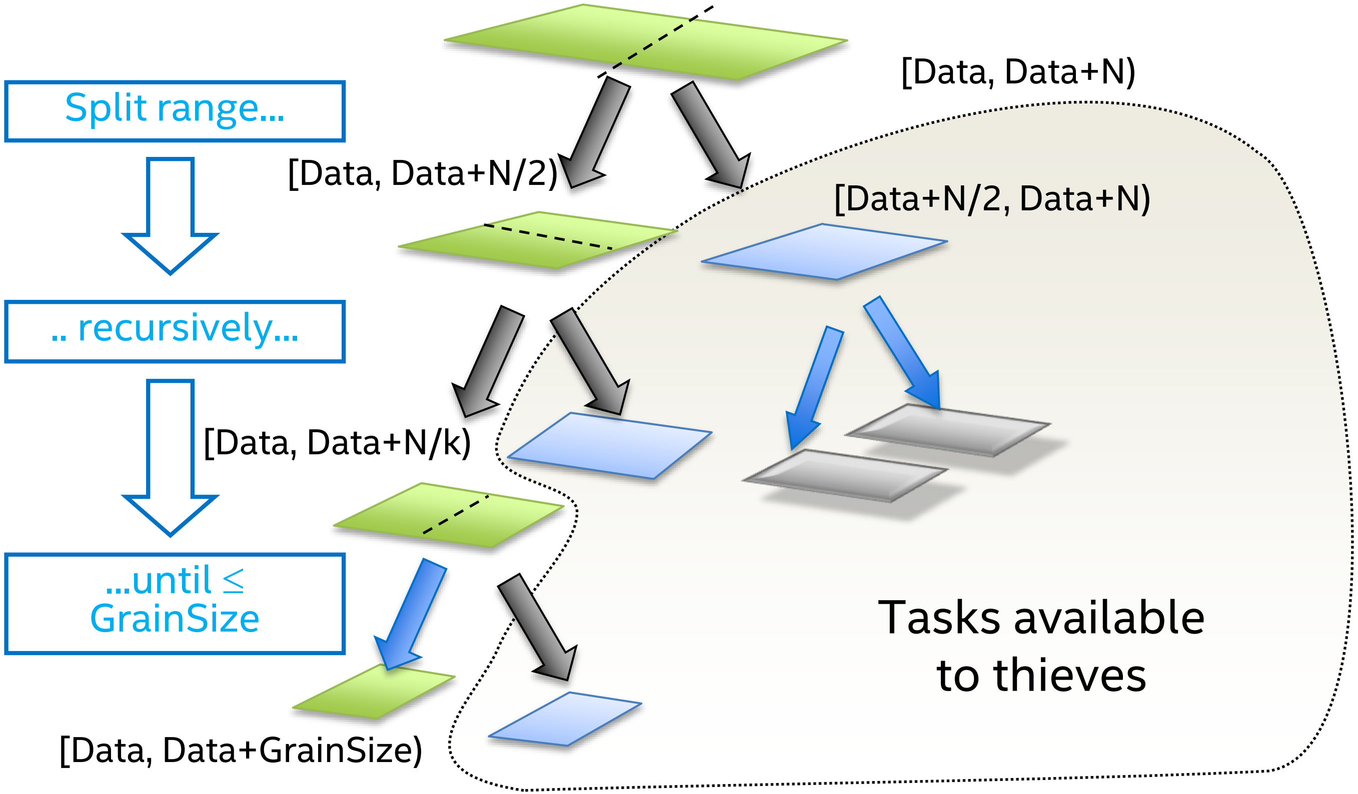
Modern C++ uses lambda functions in the STL, e.g. `std::for_each`

```
std::vector<float> array;
// Replace each element in an array with its square root
std::for_each (array.begin(), array.end(),
   [=](float & elem) { elem = sqrt(elem);});
```

Intel® TBB also exploits them, e.g. parallelize the code above

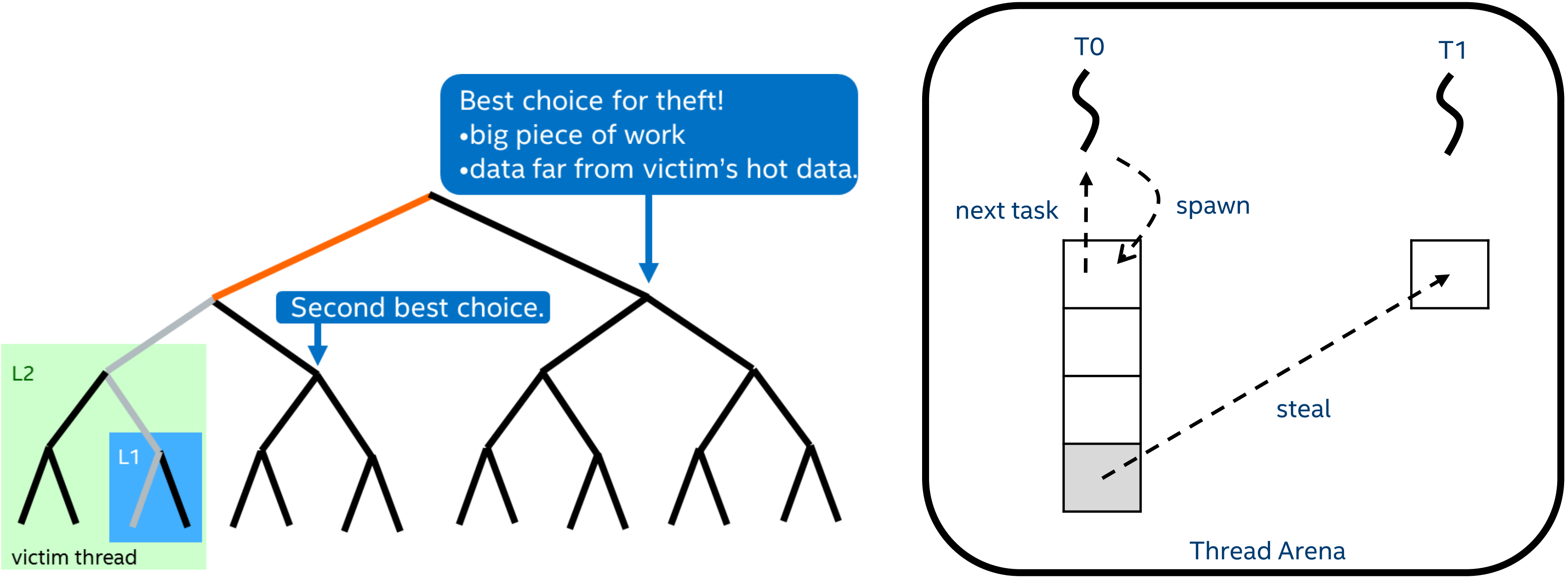
```
std::vector<float> array;
// Replace each element in an array with its square root
tbb::parallel_for_each (array.begin(), array.end(),
   [=](float & elem) { elem = sqrt(elem);});
```

Recursive parallelism



Work Depth First; Steal Breadth First

For Cache Oblivious Algorithms



Critical Points

You express parallelism in your algorithm

Intel TBB works out how to exploit that

- Runtime load-balancing

You don't (and shouldn't want to) know about threads

Even with the pipeline, and flow::graph which *look* as if you're connecting threads with FIFOs, you're not!

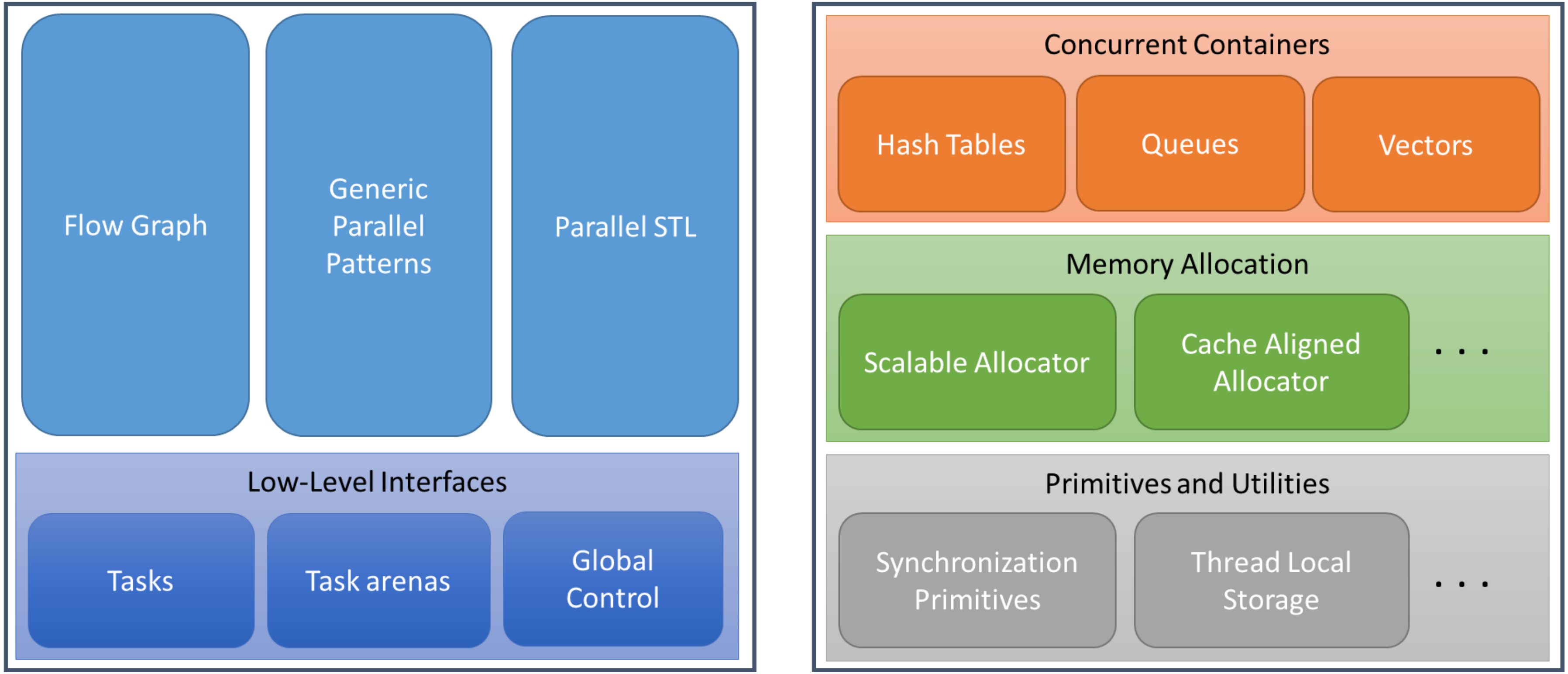
Because you're not scheduling or worrying about threads, all levels of parallelism can co-operate

With flow graph you can support multiple CPUs and other computational hardware in the same framework

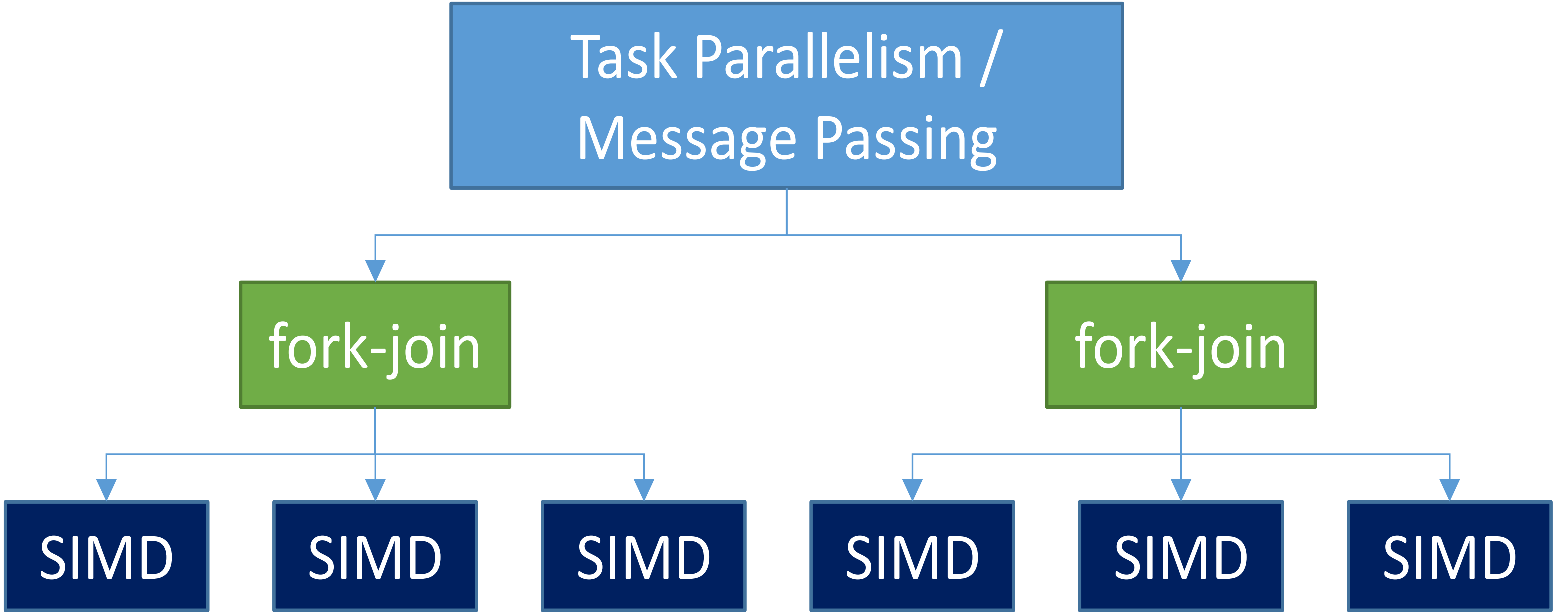
Intel® Threading Building Blocks

threadingbuildingblocks.org

Parallel Execution Interfaces — Interfaces Independent of Execution Model

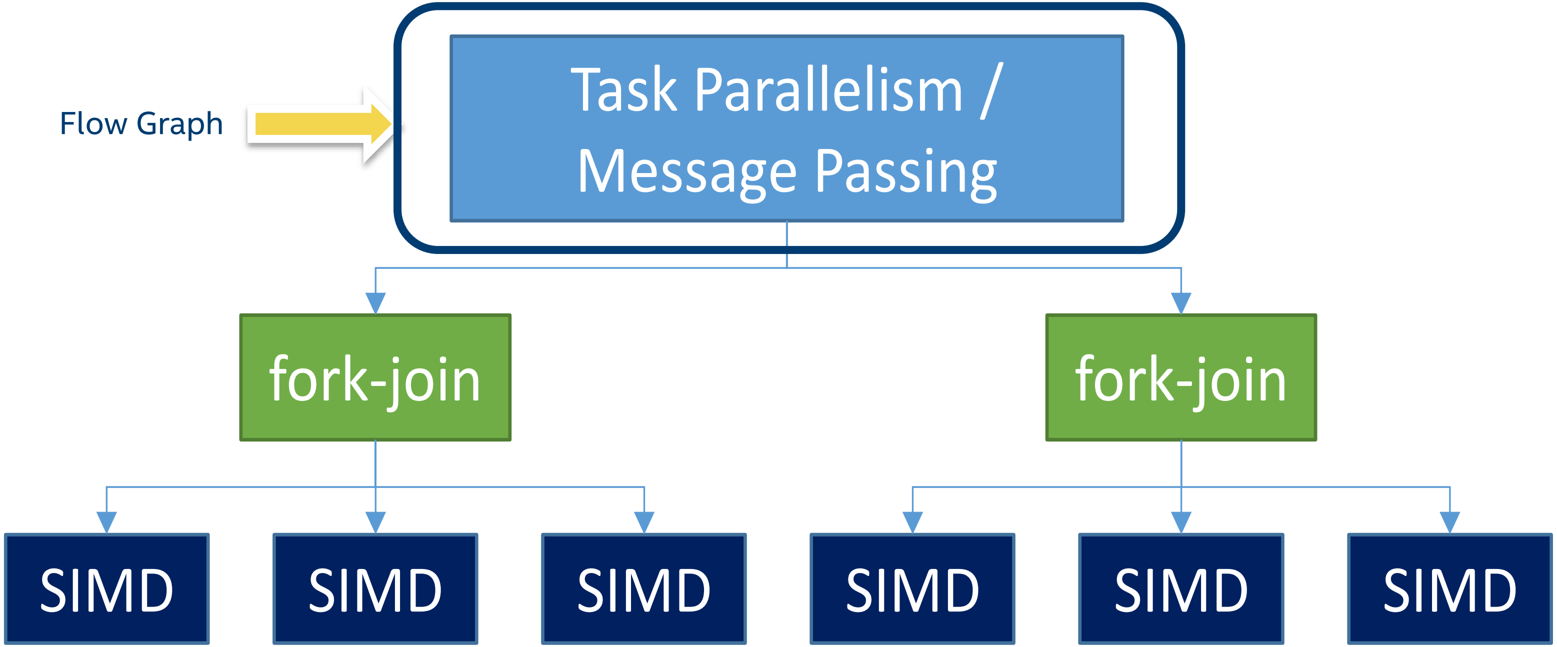


Applications often contain multiple levels of parallelism



Intel TBB helps to develop composable levels

Applications often contain multiple levels of parallelism



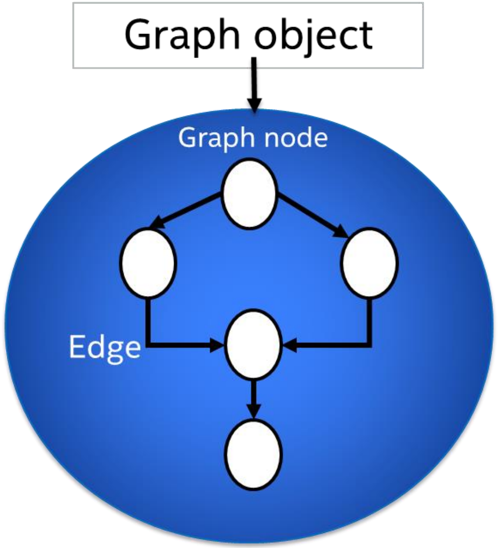
Intel TBB helps to develop composable levels

Intel Threading Building Blocks flow graph

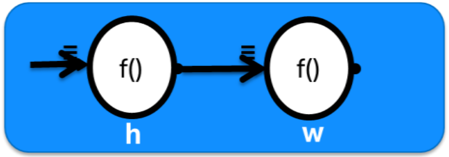
Efficient implementation of dependency graph and data flow algorithms

Initially designed for shared memory applications

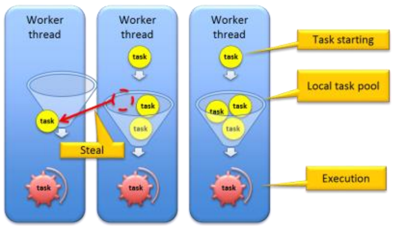
Enables developers to exploit parallelism at higher levels



Hello World

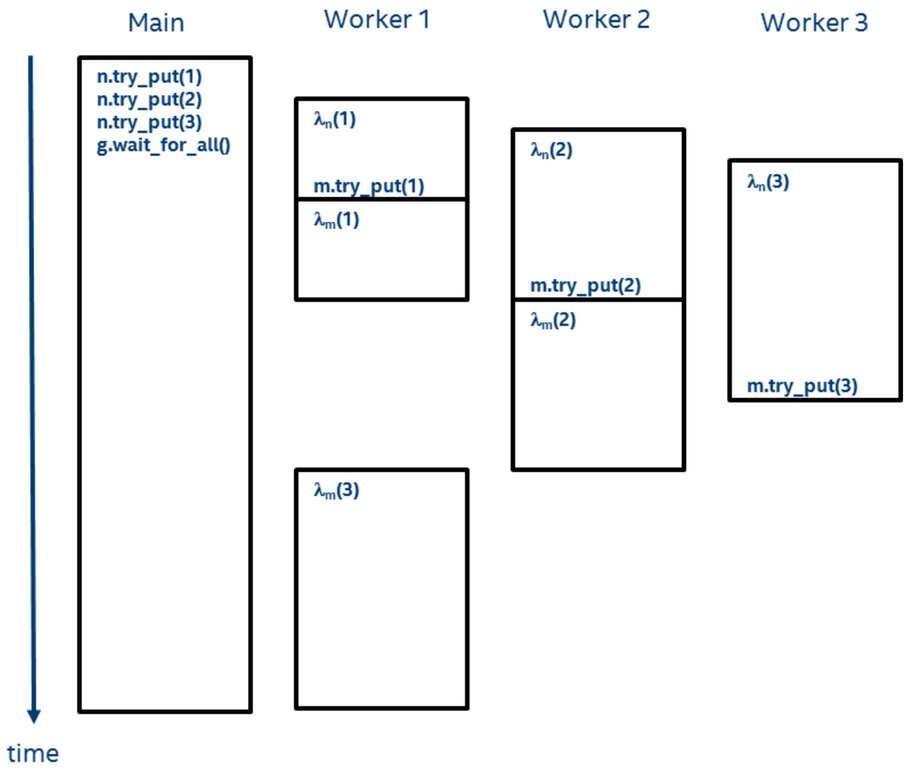



```
graph g;
continue_node< continue_msg > h( g,
    []( const continue_msg & ) {
        cout << “Hello “;
    } );
continue_node< continue_msg > w( g,
    []( const continue_msg & ) {
        cout << “World\n“;
    } );
make_edge( h, w );
h.try_put(continue_msg());
g.wait_for_all();
```



How nodes map to Intel TBB tasks

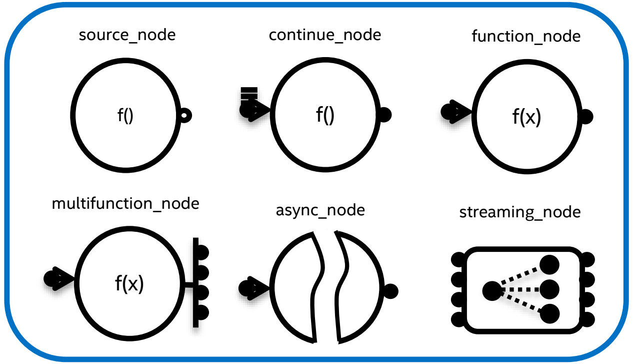
```
    graph g;
    function_node< int, int > n( g, unlimited,
[]( int v ) -> int {
        cout << v;
        spin_for( v );
        cout << v;
        return v;
    } );
    function_node< int, int > m( g, serial, [](
int v ) -> int {
        v *= v;
        cout << v;
        spin_for( v );
        cout << v;
        return v;
    } );
    make_edge( n, m );
    n.try_put( 1 );
    n.try_put( 2 );
    n.try_put( 3 );
    g.wait_for_all();
```



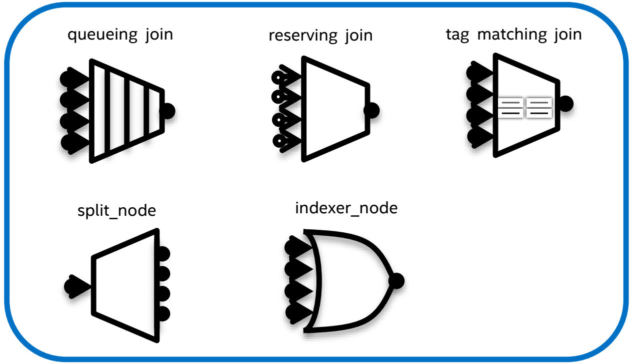
One possible execution – stealing is random

Intel TBB Flow Graph node types:

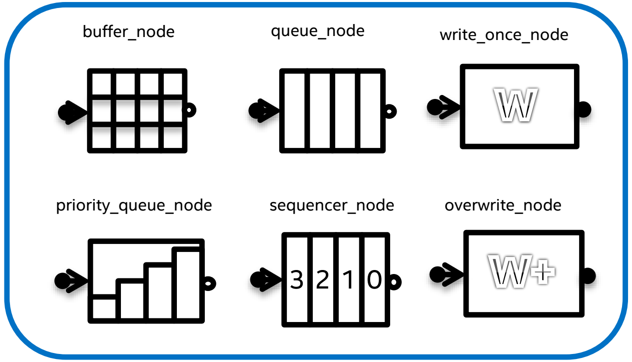
Functional



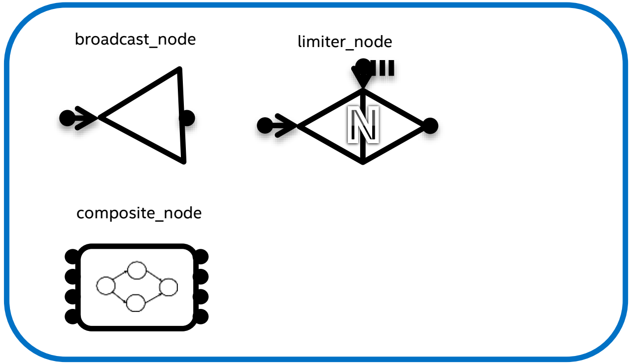
Split / Join



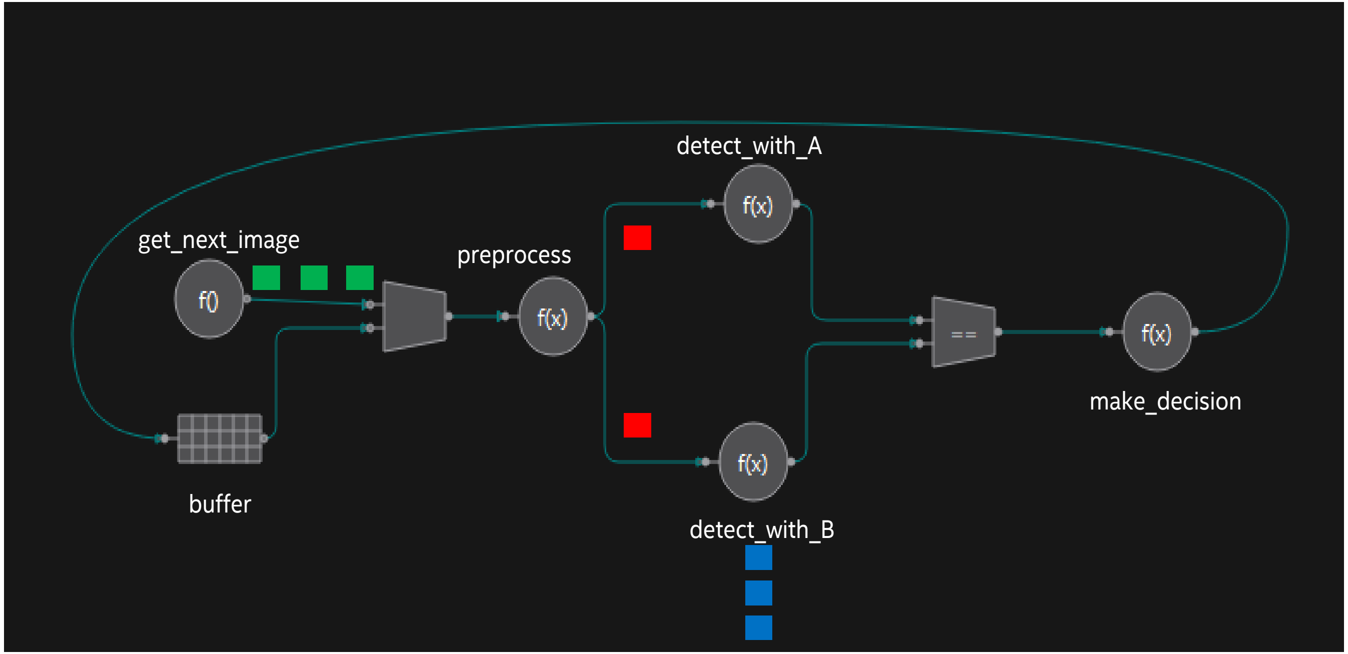
Buffering



Other

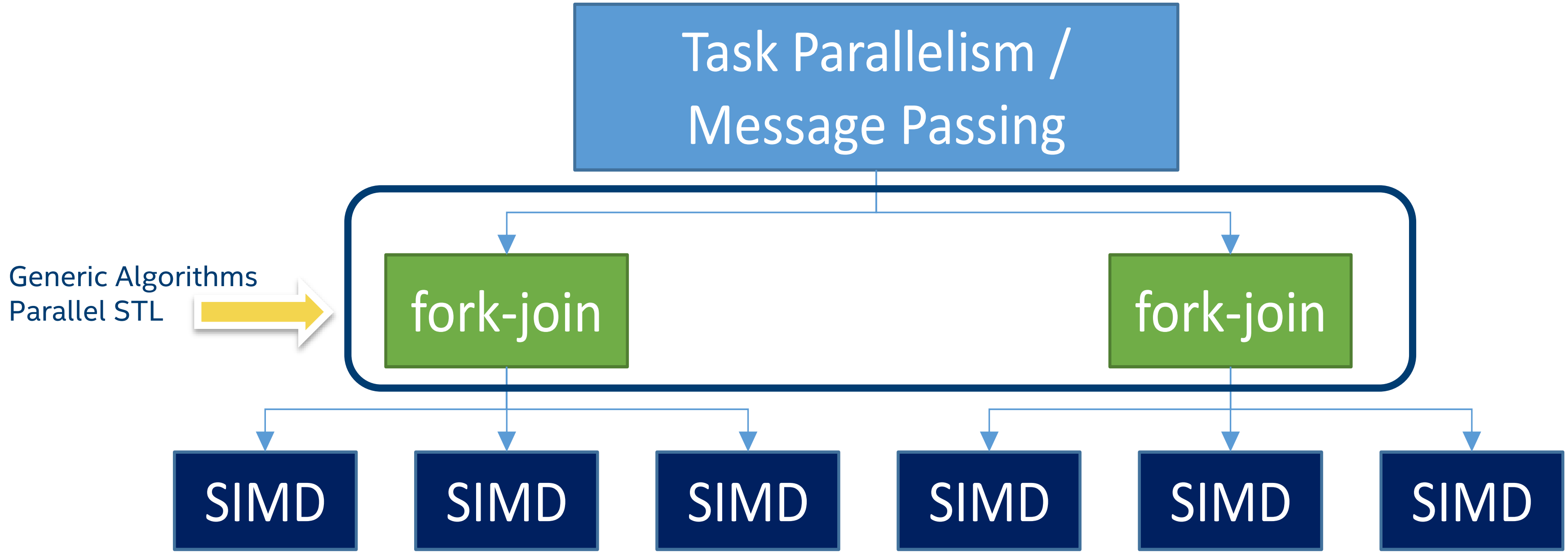


An example feature detection algorithm



Can express pipelining, task parallelism and data parallelism

Applications often contain multiple levels of parallelism



Intel TBB helps to develop composable levels

Intel® TBB Generic Parallel Algorithms

Loop parallelization

parallel_for

parallel_reduce

parallel_scan

Parallel sorting

parallel_sort

Parallel function invocation

parallel_invoke

Streaming

parallel_do

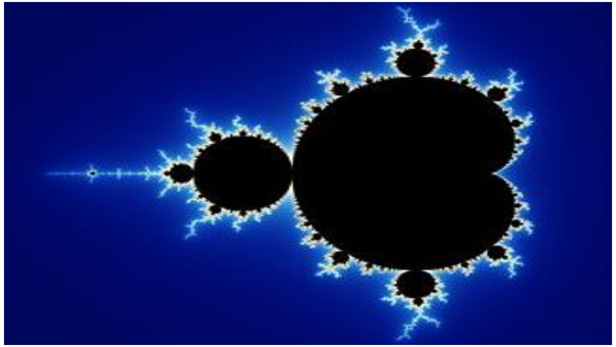
parallel_for_each

pipeline / parallel_pipeline

flow graph

Mandelbrot Example

Intel® Threading Building Blocks (Intel® TBB)



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

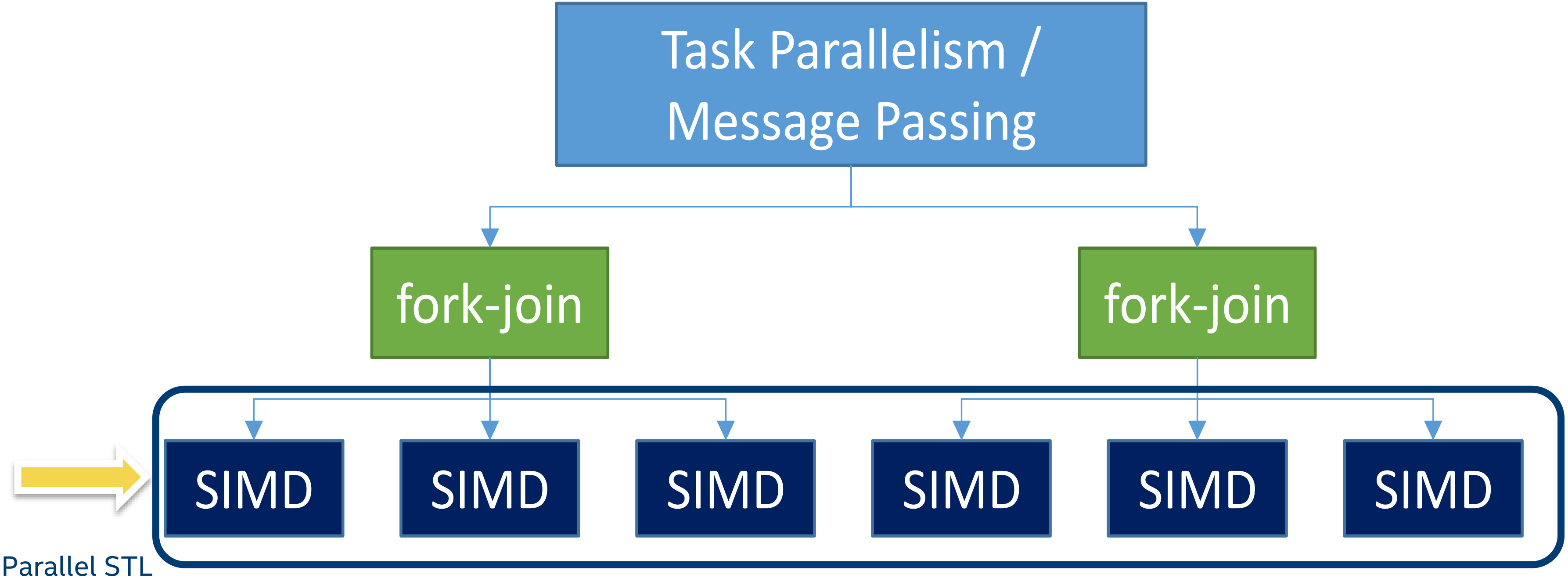
Task is a function object

Parallel algorithm

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

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

Applications often contain multiple levels of parallelism



Parallel STL
#pragma omp simd

Intel TBB helps to develop composable levels

Standard Template Library

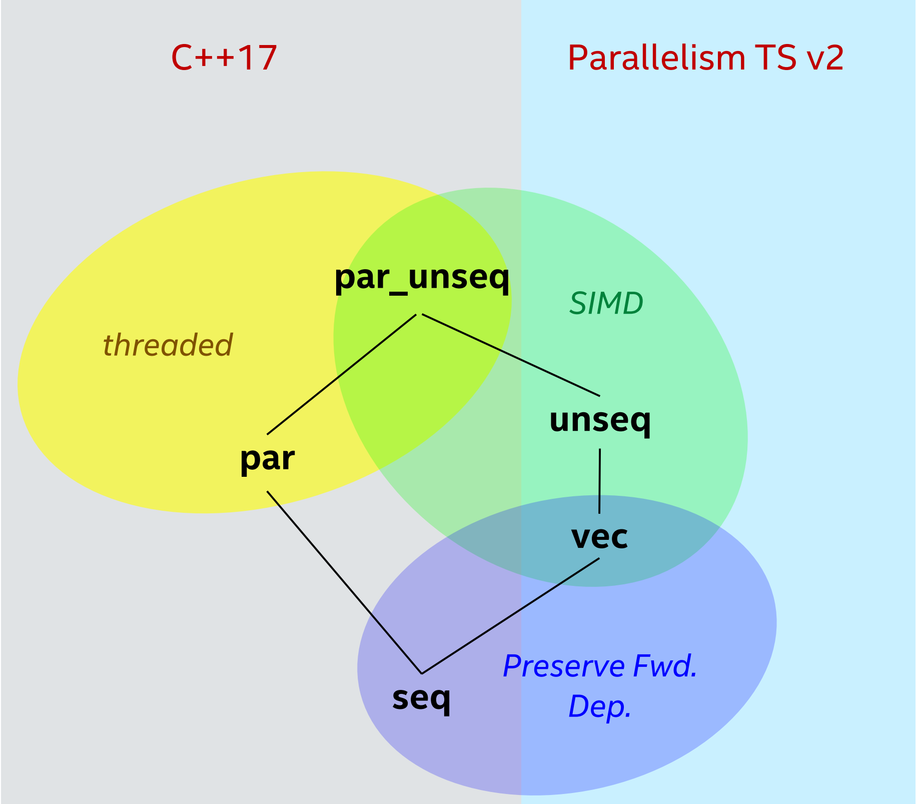
Container ← *Iterator* → *Algorithm*

std::vector<float>   float*   transform

```
#include <algorithm>

void increment( float *in, float *out, int N ) {
    using namespace std;
    transform( in, in + N, out, []( float f ) {
        return f+1;
    });
}
```

Enter Parallel STL

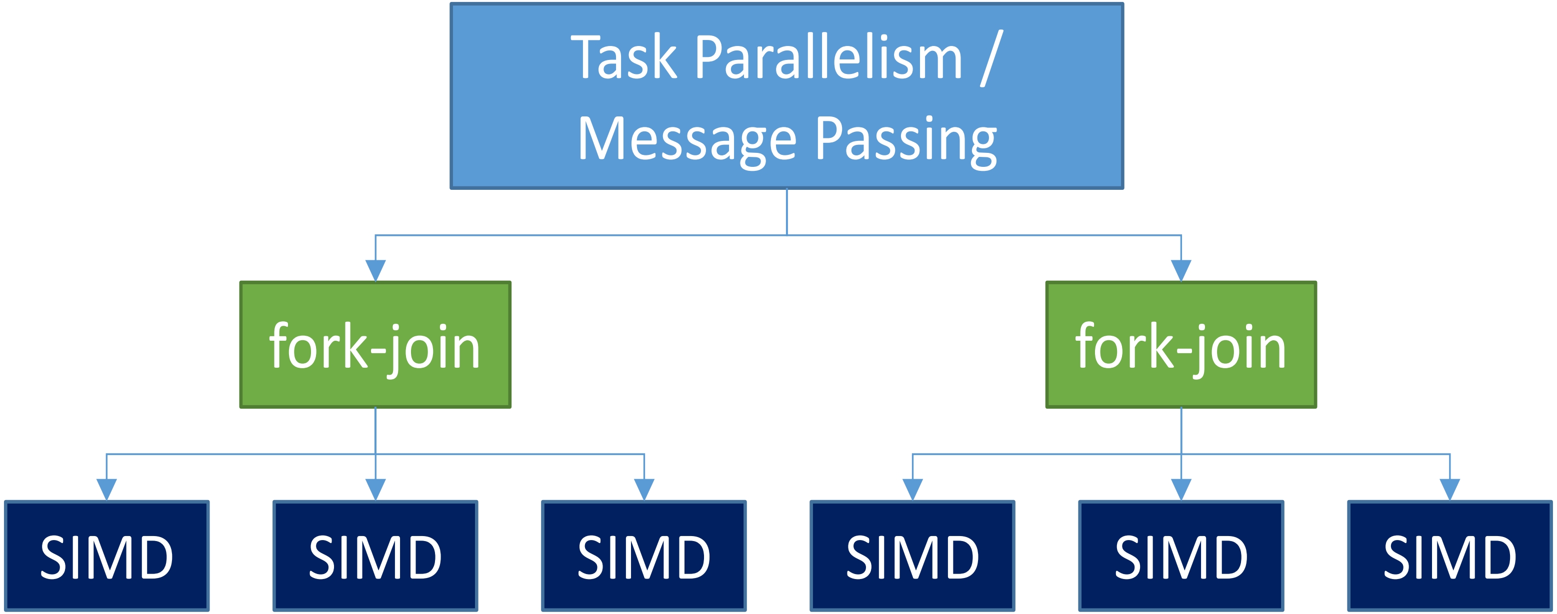
- Extension of C++ Standard Template Library algorithms with the “execution policy” argument
- Support for parallel execution policies is approved for C++17
- Support for vectorization policies is being developed in Parallelism Technical Specification (TS) v2



Parallel STL Examples

```
// standard sequential sort
sort(v.begin(), v.end());

// explicitly sequential sort
sort(execution::seq,v.begin(), v.end());

// permitting parallel execution
sort(execution::par,v.begin(), v.end());

// permitting vectorization as well
sort(execution::par_unseq,v.begin(), v.end());

// Parallelism TS v2
// permitting vectorization only (no parallel execution)
sort(execution::unseq,v.begin(), v.end());
```

Applications often contain multiple levels of parallelism

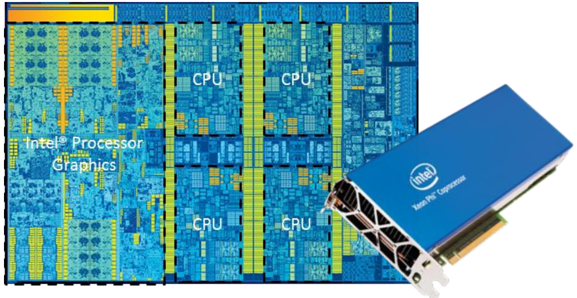


Intel TBB helps to develop composable levels

The heterogeneous programming features of Intel® Threading Building Blocks

Heterogeneous support in Intel® TBB

Intel TBB flow graph as a coordination layer for heterogeneity that retains optimization opportunities and composes with existing models



FPGAs, integrated and discrete GPUs, co-processors, etc...

\+

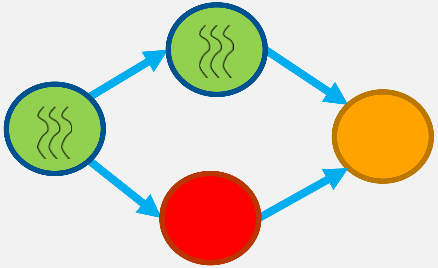
Intel® Threading Building Blocks
OpenVX*
OpenCL*
COI/SCIF
....

Intel TBB as a **composability layer** for library implementations

- One threading engine ***underneath*** all CPU-side work

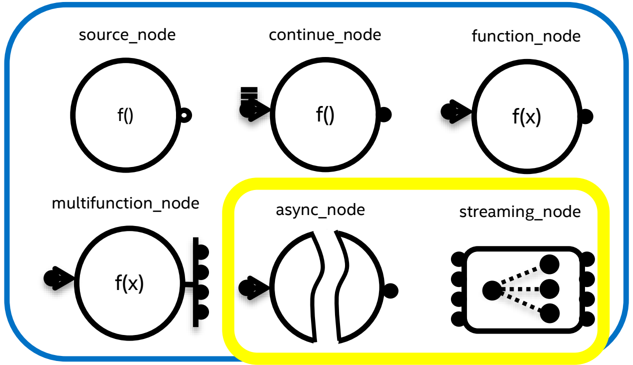
Intel TBB flow graph as a **coordination layer**

- Be the glue that connects hetero HW and SW together
- Expose parallelism between blocks; simplify integration

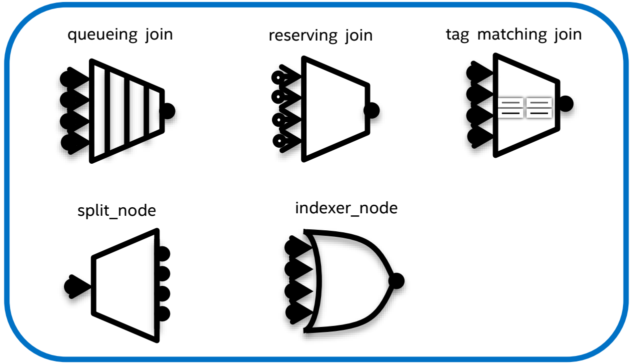


Intel TBB Flow Graph node types:

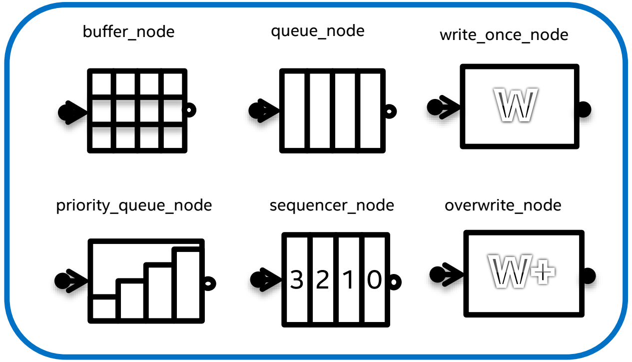
Functional



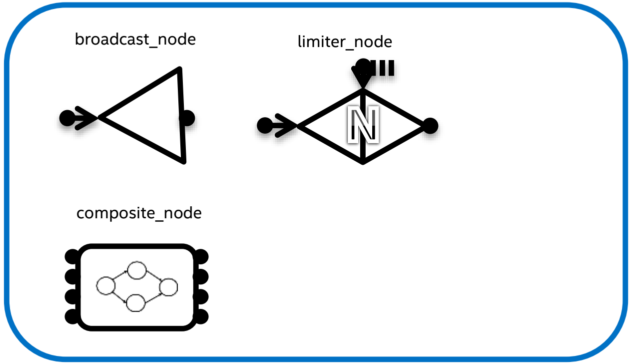
Split / Join

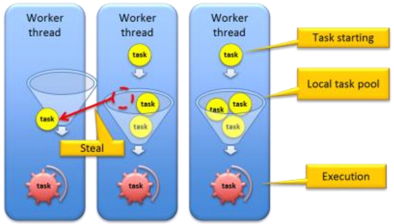


Buffering



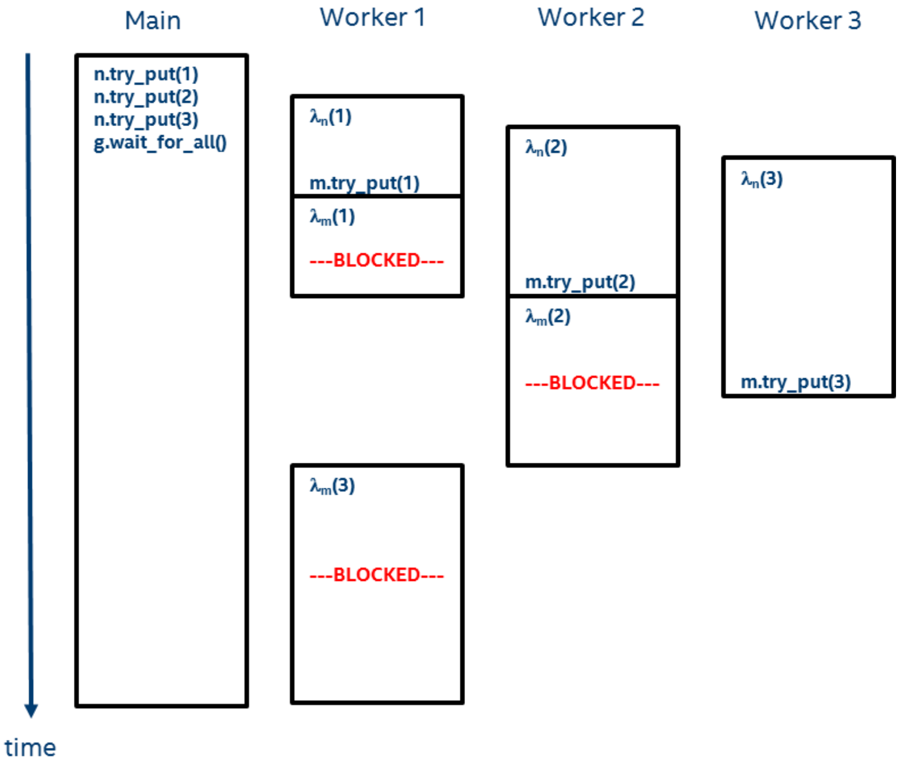
Other





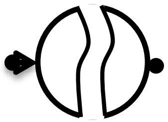
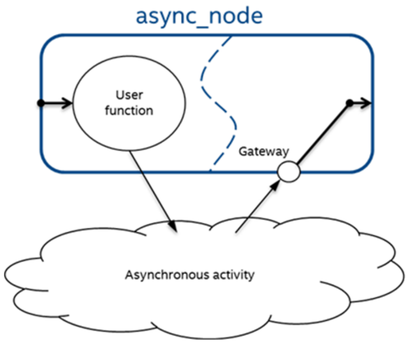
Why is extra support needed?

```
    graph g;
    function_node< int, int > n( g, unlimited,
[]( int v ) -> int {
        cout << v;
        spin_for( v );
        cout << v;
        return v;
    } );
    function_node< int, int > m( g, serial, [](
int v ) -> int {
        BLOCKING_OFFLOAD_TO_ACCELERATOR();
    } );
    make_edge( n, m );
    n.try_put( 1 );
    n.try_put( 2 );
    n.try_put( 3 );
    g.wait_for_all();
```



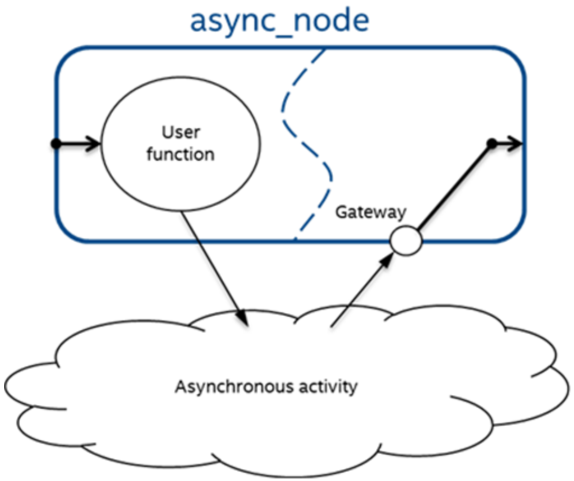
One possible execution – stealing is random

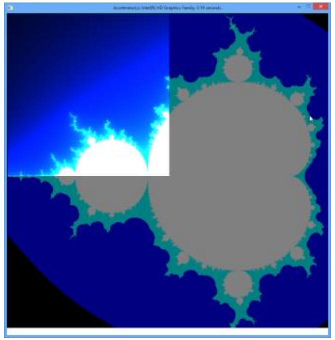
Heterogeneous support in the Intel TBB flow graph (1 of 3)

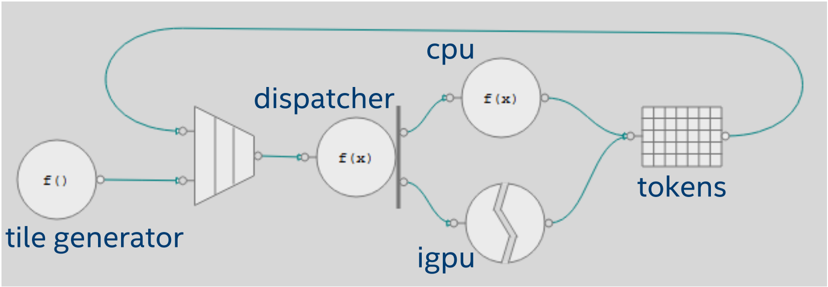
Feature	Description	Diagram
async_node<Input,Output> 	Basic building block. Enables asynchronous communication from a single/isolated node to an asynchronous activity. User is responsible for managing communication. Graph runs on host.	

async_node example

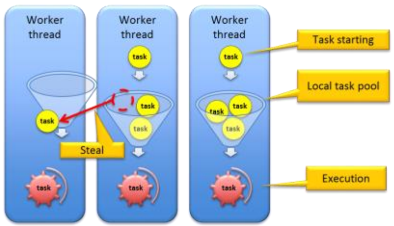
- Allows the data flow graph to offload data to any asynchronous activity and receive the data back to continue execution on the CPU
- Avoids blocking a worker thread





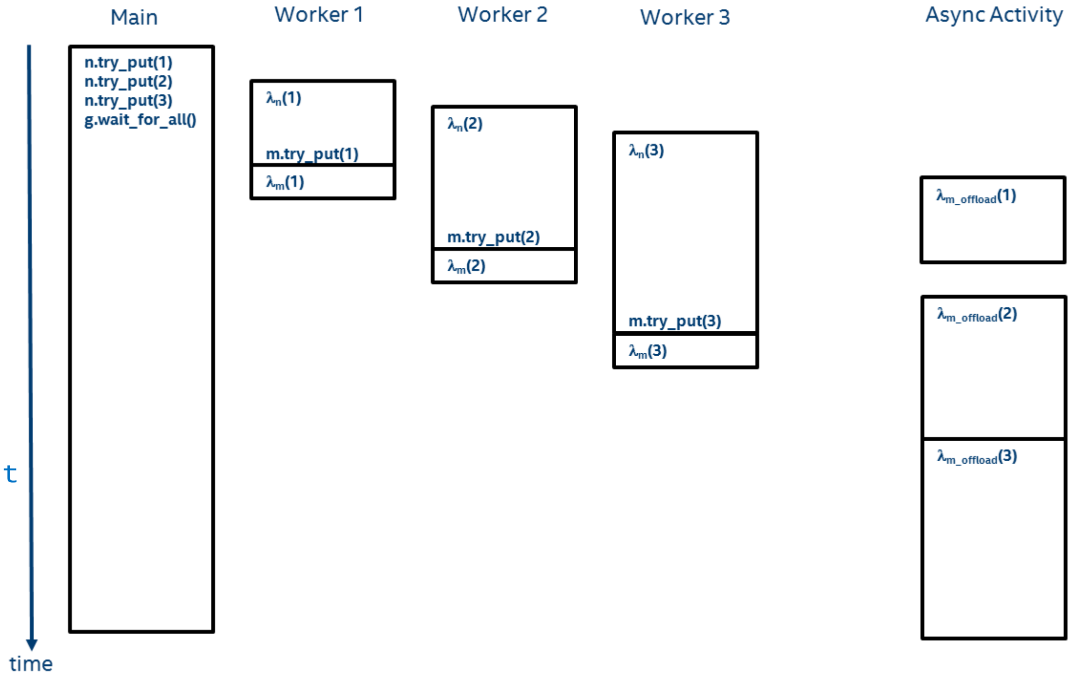


async_node makes coordinating with any model easier and efficient



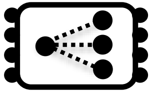
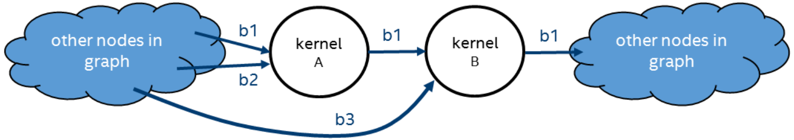
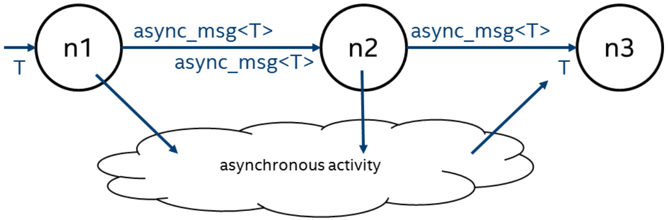
With async_node

```
    graph g;
    my_async_activity_type my_async_activity;
    function_node< int, int > n( g, unlimited,
[]( int v ) -> int {
        cout << v;
        spin_for( v );
        cout << v;
        return v;
    } );
    typedef typename async_node<int,
int>::gateway_type gw_t;
    async_node< int, int > m( g, serial, []( int
v, gw_t &gw ) {
        my_async_activity.push(v, gw);
    } );
    make_edge( n, m );
    n.try_put( 1 );
    n.try_put( 2 );
    n.try_put( 3 );
    g.wait_for_all();
```

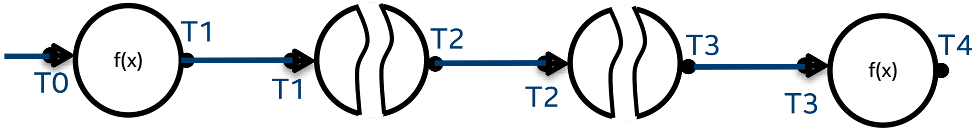


One possible execution – stealing is random

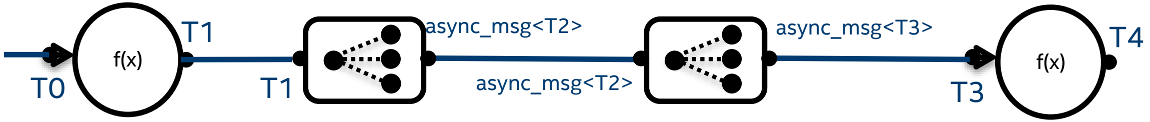
Heterogeneous support in the Intel TBB flow graph (2 of 3)

Feature	Description	Diagram
streaming_node *Available as preview feature* 	Higher level abstraction for streaming models; e.g. OpenCL*, Direct X Compute*, GFX, etc.... Users provide Factory that describes buffers, kernels, ranges, device selection, etc... Uses async_msg so supports chaining. Graph runs on the host.	
async_msg<T> *Available as preview feature*	Basic building block. Enables async communication with chaining across graph nodes. User responsible for managing communication. Graph runs on the host.	

async_node vs streaming_node

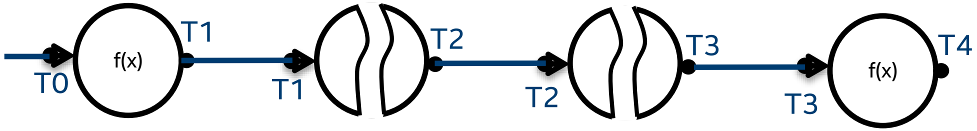


- async_node receives and sends unwrapped message types
- output message is sent after computation is done by asynchronous activity
- simpler to use when offloading a single computation and chaining is not needed

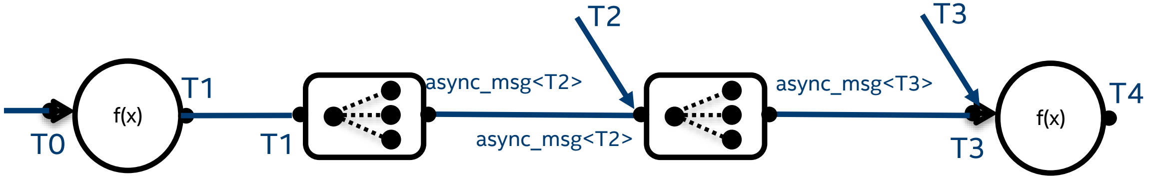


- streaming_node receives unwrapped message types or async_msg types
- sends async_msg types after enqueueing kernel, but (likely) before computation is done by asynchronous activity
- handles connections to non-streaming_nodes by deferring receive until value is set
- simple to use with pre-defined factories (like OpenCL* factory)
- non-trivial to implement factories

async_node vs streaming_node

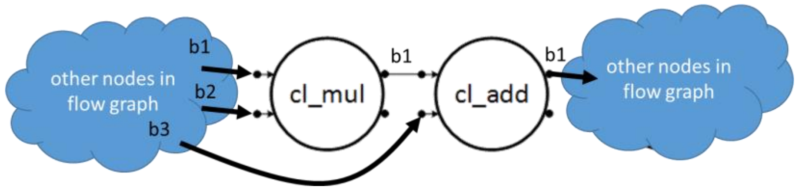


- async_node receives and sends unwrapped message types
- output message is sent after computation is done by asynchronous activity
- simpler to use when offloading a single computation and chaining is not needed

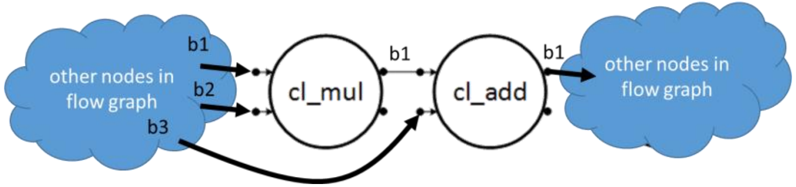


- streaming_node receives unwrapped message types or async_msg types
- sends async_msg types after enqueueing kernel, but (likely) before computation is done by asynchronous activity
- handles connections to non-streaming_nodes by deferring receive until value is set
- simple to use with pre-defined factories (like OpenCL* factory)
- non-trivial to implement factories

Heterogeneous support in the Intel TBB flow graph (3 of 3)

Feature	Description	Diagram
opencl_node *Available as preview feature*	A factory provided for streaming_node that supports OpenCL*. User provides OpenCL* program and kernel and the runtime handles the initialization, buffer management, communications, etc.. Graph runs on host.	

opencl_node example



- Provides a first order node type that takes in OpenCL* programs or SPIR* binaries that can be executed on any supported OpenCL device
- Is a streaming_node with opencl_factory
- https://software.intel.com/en-us/blogs/2015/12/09/opencl-node-overview

```cpp
#define TBB_PREVIEW_FLOW_GRAPH_NODES 1
#include "tbb/flow_graph_opencl_node.h"

#include <algorithm>

int main() {
    using namespace tbb::flow;

    opencl_graph g;
    opencl_node<tuple<opencl_buffer<cl_char>>>
        clPrint( g, "hello_world.cl", "print" );

    const char str[] = "Hello, World!";
    opencl_buffer<cl_char> b( g, sizeof(str) );
    std::copy_n( str, sizeof(str), b.begin() );

    clPrint.set_ndranges( { 1 } );
    input_port<0>(clPrint).try_put( b );

    g.wait_for_all();

    return 0;
}
```

Using Intel® TBB beyond the CPU and integrated GPU

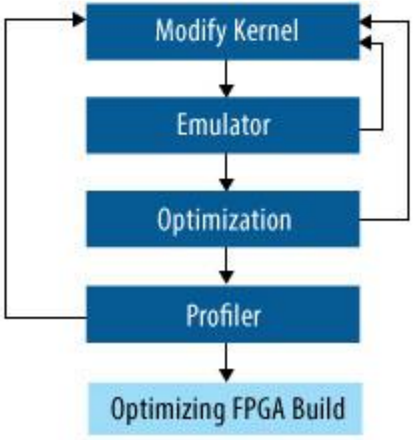
Using other GPGPU models with Intel TBB

- CUDA*, Vulkan*, Direct Compute*, etc…
- Two approaches

1. Use an async_node to avoid blocking a worker thread
2. Create (or advocate for) a streaming_node factory
   - Intel TBB accepts contributions!

FPGAs and other non-GPU devices

- OpenCL* supports more than CPU and GPU
- The Intel® FPGA SDK for Open Computing Language (OpenCL)



https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html

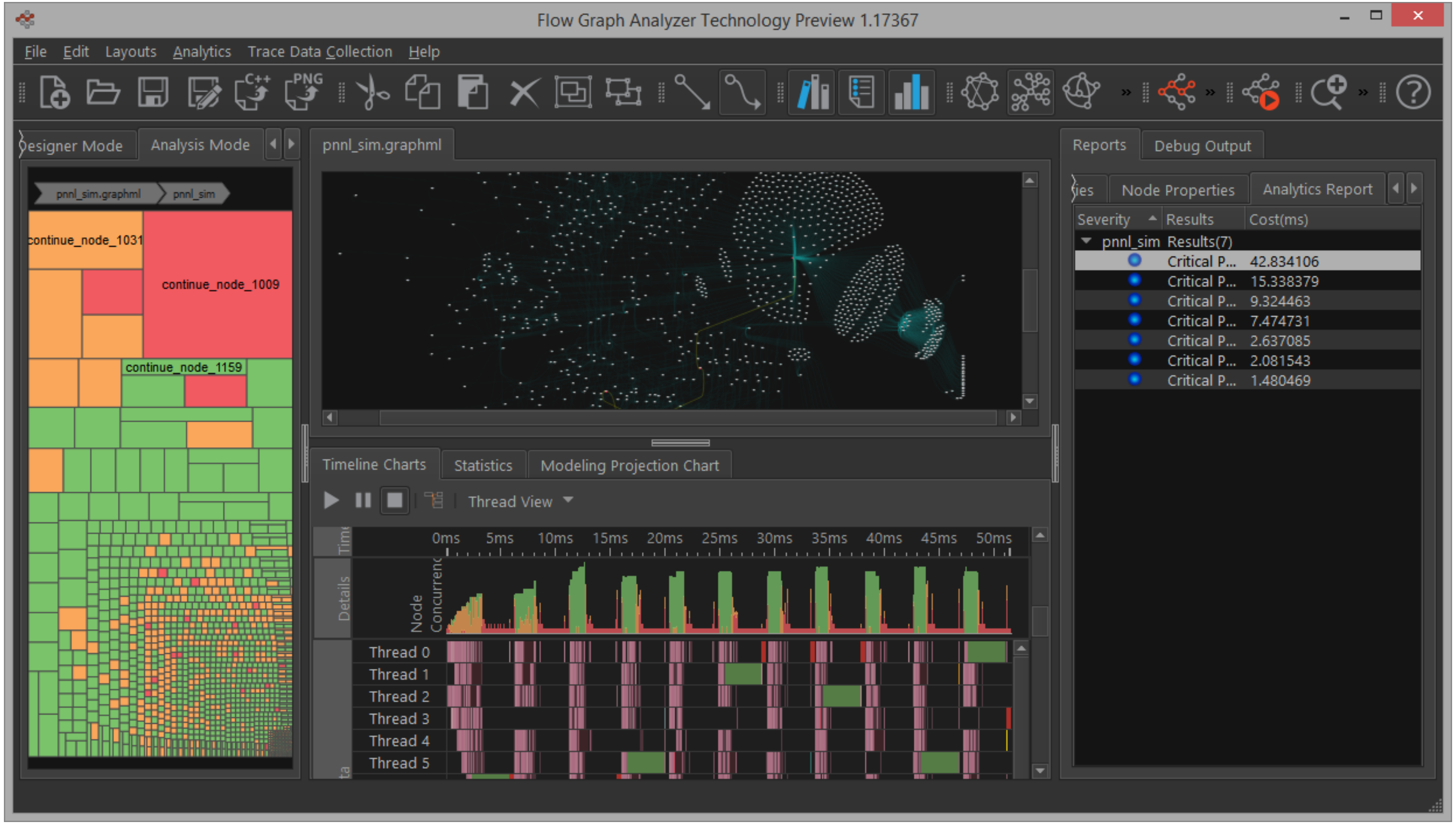
- Working on improved support from within Intel TBB

Developing kernels for FPGAs and other non-GPU devices

- There are projects such as http://halide-lang.org/ that assist in writing more easily retargeted kernels.
  - Halide is a domain-specific C++ library that targets image processing.
  - Halide supports OpenCL*, CUDA* and others as a back-ends.
  - Tuned kernels can be dropped in to streaming_nodes or invoked from async_node.
- Make use nested calls to optimized libraries such as Intel® Math Kernel Library (MKL)

Flow Graph Analyzer

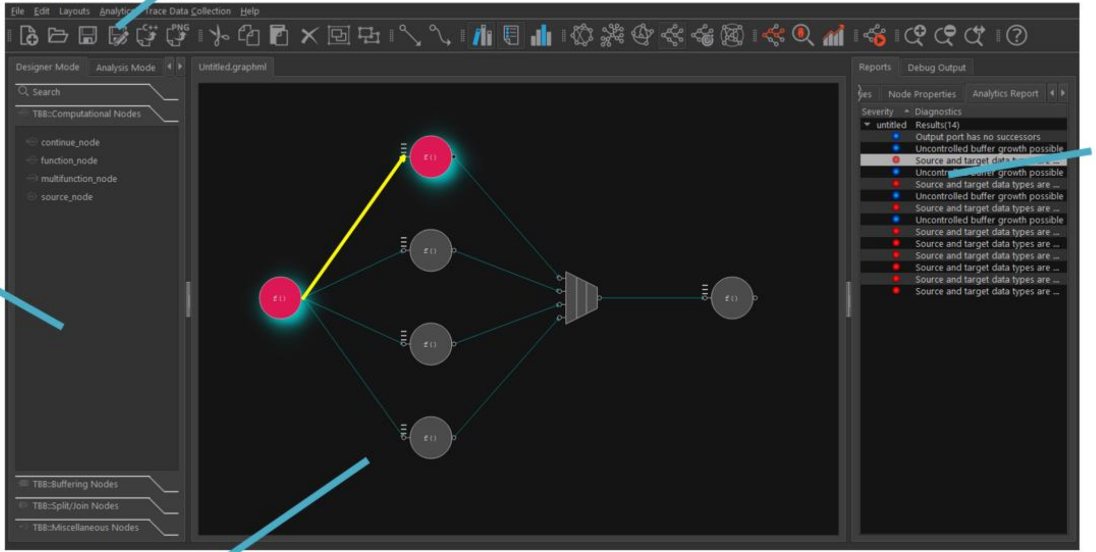
Flow Graph Analyzer for Intel® Threading Building Blocks



Flow Graph Analyzer for Intel® TBB (Designer Workflow)

Toolbar supporting basic file and editing operations, visualization and analytics that operate on the graph or performance traces

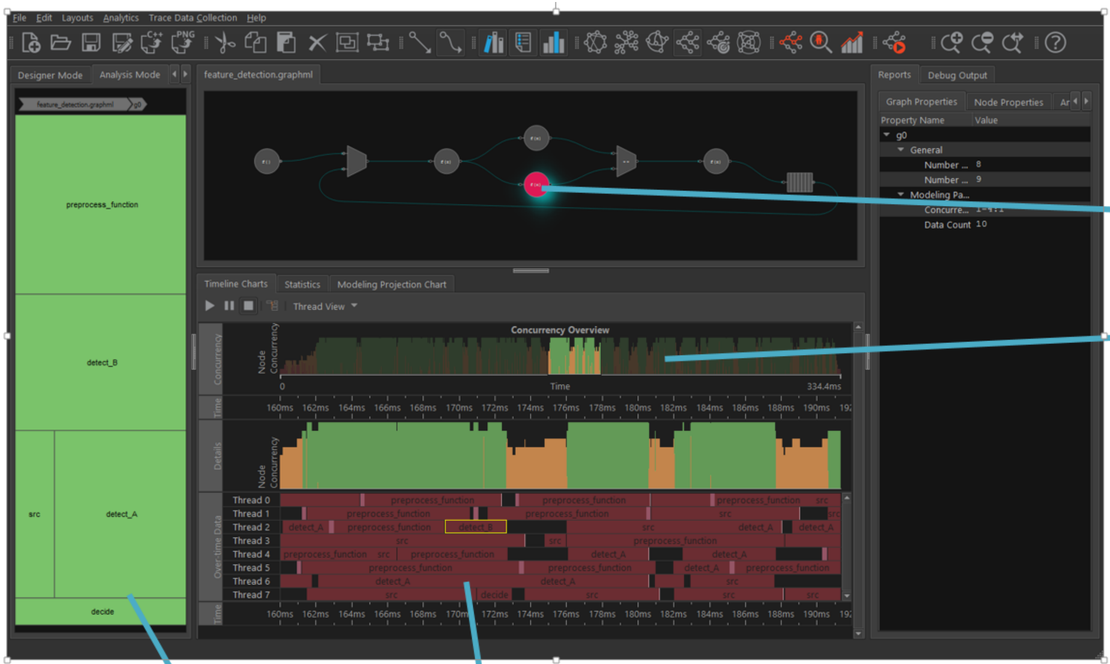
Palette of supported Intel® TBB node types organized in like groups



Displays the output generated by custom analytics and allows interactions with this output

Canvas for visualizing and drawing flow graphs

Flow Graph Analyzer for Intel® TBB (Analyzer Workflow)



Selection on the timeline highlights the nodes that were executing at that point in time.

The concurrency histogram shows the parallelism achieved by the graph over time. You can interact with this chart by zooming in to a region of time, for example during low concurrency.

The concurrency histogram remains at the initial zoom level, and the zoomed in region is displayed below it.

Treemap view gives you the general health of the graph's performance along with the ability to dive to the node level.

The per-thread task view shows the tasks executed by each thread along with the task durations.

Additional resources

The Intel® Threading Building Blocks open-source community:
https://www.threadingbuildingblocks.org/

The Intel® Threading Building Blocks GitHub repository:
https://github.com/01org/tbb/

Special Issue of Parallel Universe Magazine on Intel TBB:
https://software.intel.com/sites/default/files/managed/4f/e5/ParallelUniverseMagazine_Special_Edition_v2.pdf

Flow Graph Analyzer:
https://software.intel.com/en-us/articles/flow-graph-designer

Intel FPGA SDK for Open Computing Language (OpenCL)
https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html

The Khronos Group consortium (API standards including OpenCL, OpenVX and Vulkan)
https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html

Legal Disclaimer & Optimization Notice

experience what's inside™