PROGRAMMING WITH TBB FLOW GRAPH

Aleksei Fedotov

November 29, 2017

Agenda

- An overview of the Intel® Threading Building Blocks (Intel® TBB) library
  - It's three high-level execution interfaces and how they map to the common three layers of parallelism in applications
- The heterogeneous programming extensions in Intel TBB
  - async_node, streaming_node, opencl_node, etc…

Intel® Threading Building Blocks (Intel® TBB)

Celebrated it's 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

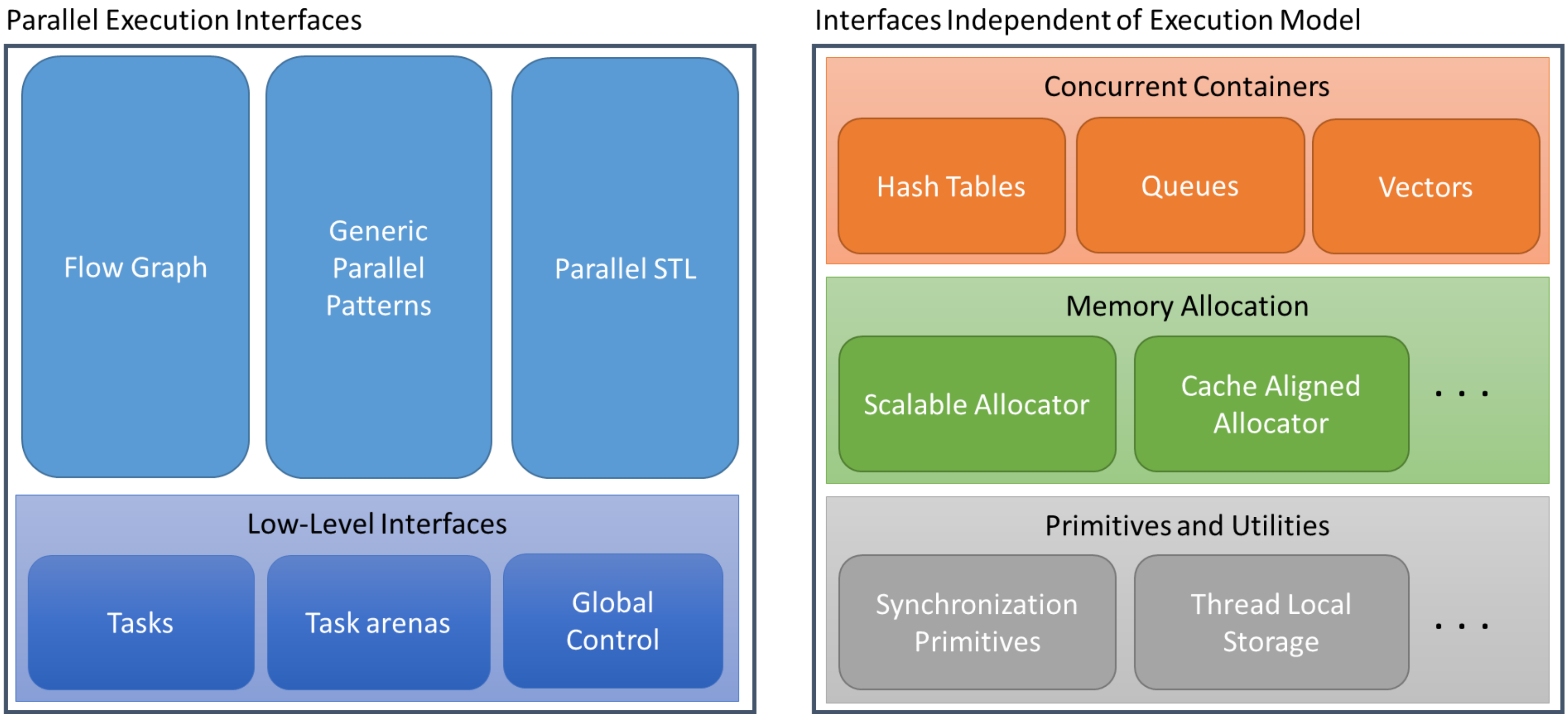
Benefits

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

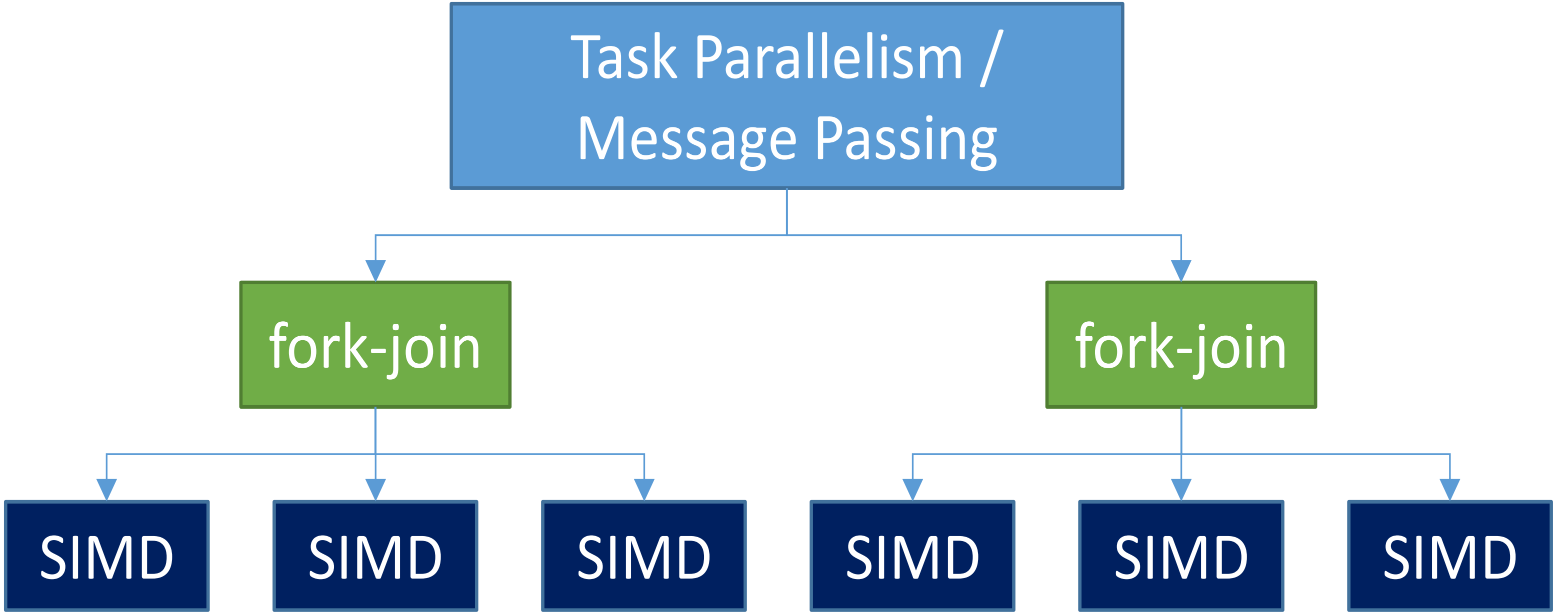
http://threadingbuildingblocks.org http://software.intel.com/intel-tbb

Intel® Threading Building Blocks

threadingbuildingblocks.org

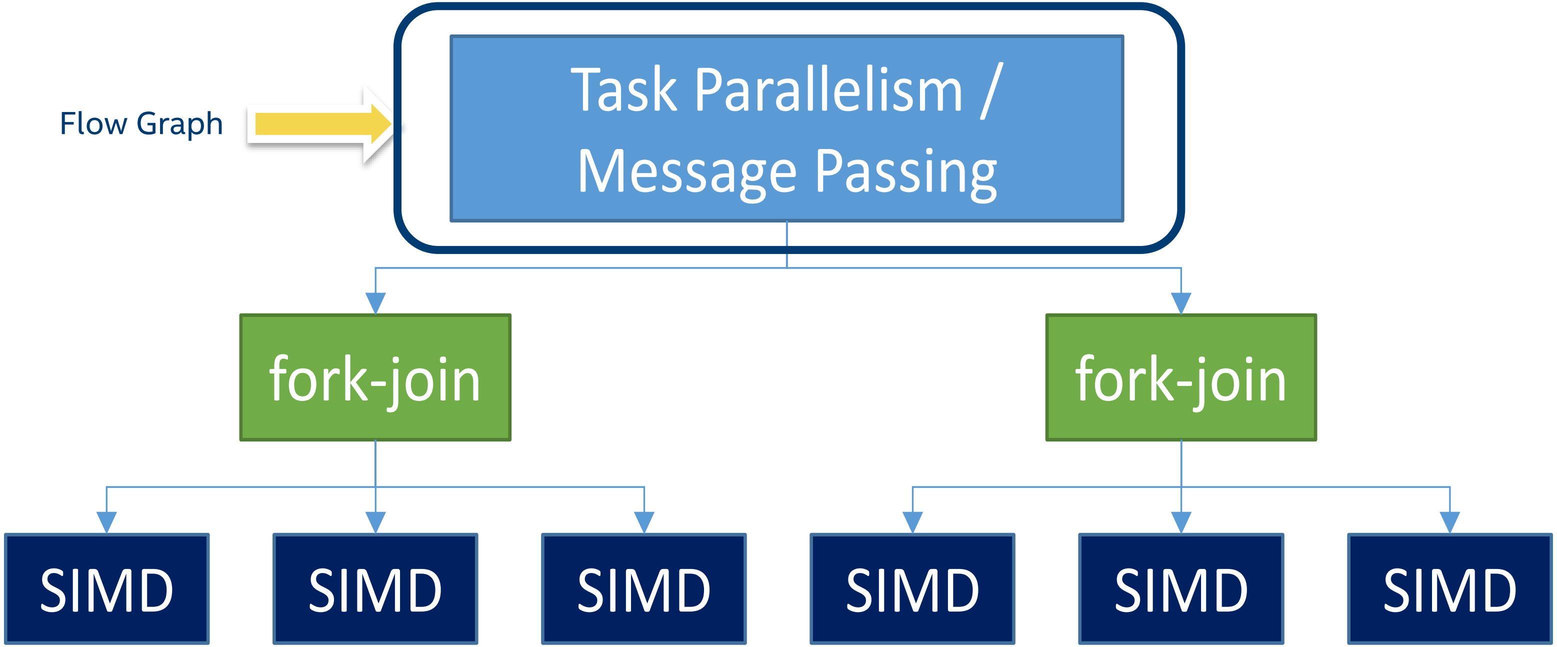


Applications often contain multiple levels of parallelism



Intel TBB helps to develop composable levels

Applications often contain multiple levels of parallelism



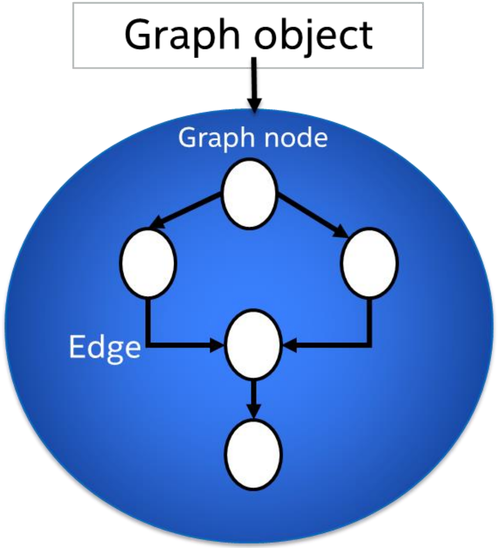
Intel TBB helps to develop composable levels

Intel Threading Building Blocks flow graph

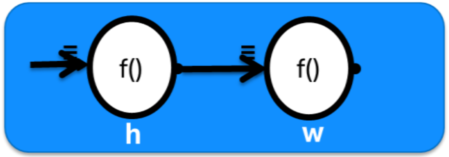
Efficient implementation of dependency graph and data flow algorithms

Initially designed for shared memory applications

Enables developers to exploit parallelism at higher levels

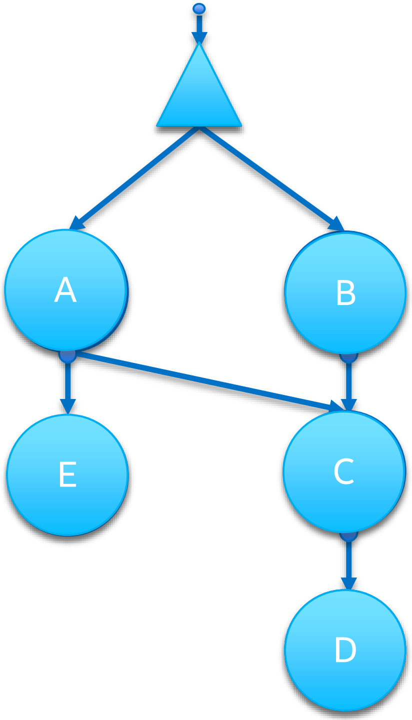


Hello World



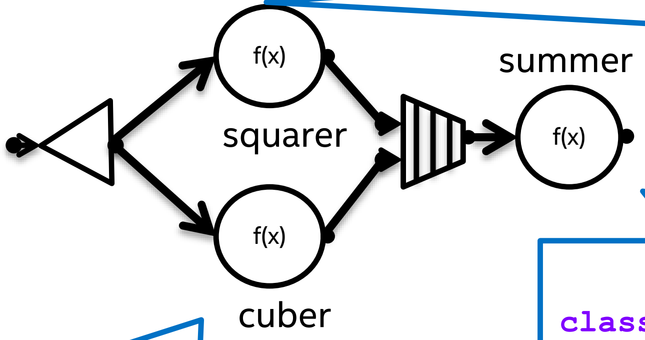

```
graph g;
continue_node< continue_msg > h( g,
    []( const continue_msg & ) {
        cout << "Hello ";
    } );
continue_node< continue_msg > w( g,
    []( const continue_msg & ) {
        cout << "World\n";
    } );
make_edge( h, w );
h.try_put(continue_msg());
g.wait_for_all();
```

Example with nonlinear dependencies




```
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;
  broadcast_node< continue_msg > start(g);
  continue_node< continue_msg > a( g, body("A") );
  continue_node< continue_msg > b( g, body("B") );
  continue_node< continue_msg > c( g, body("C") );
  continue_node< continue_msg > d( g, body("D") );
  continue_node< continue_msg > e( g, body("E") );
  make_edge( start, a );  make_edge( start, b );
  make_edge( a, c );      make_edge( b, c );
  make_edge( c, d );      make_edge( a, e );
  for (int i = 0; i < 3; ++i ) {
    start.try_put( continue_msg() );
    g.wait_for_all();
  }
  return 0;
}
```

Data flow graph example




```
struct square {
  int operator()(int v) {
    return v * v;
  }
};
```

```
struct cube {
  int operator()(int v) {
    return v * v * v;
  }
};
```

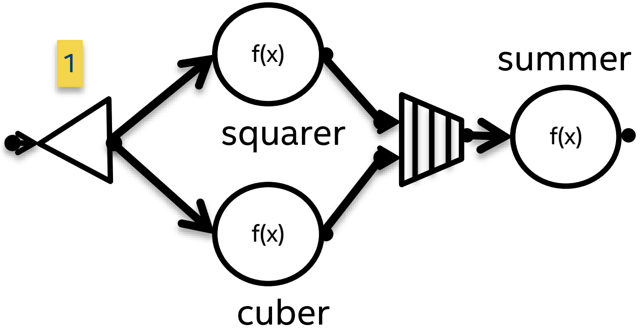
```
class sum {
  int &my_sum;
public:
  sum( int &s ) : my_sum(s) {}
  int operator()( tuple<int, int> v ) {
    my_sum += get<0>(v) + get<1>(v);
    return my_sum;
  }
};
```

Optimization Notice

Data flow graph example

```
int main() {
  int result = 0;
  graph g;
  broadcast_node<int> input(g);
  function_node<int, int> squarer( g, unlimited, square() );
  function_node<int, int> cuber( g, unlimited, cube() );
  join_node< tuple<int, int>, queueing > j( g );
  function_node< tuple<int, int>, int > summer( g, serial, sum(result) );

  make_edge( input, squarer );
  make_edge( input, cuber );
  make_edge( squarer, input_port<0>(j) );
  make_edge( cuber, input_port<1>(j) );
  make_edge( j, summer );

  for (int i = 1; i <= 3; ++i)
      input.try_put(i);
  g.wait_for_all();
  printf("Final result is %d\n", result);
  return 0;
}
```

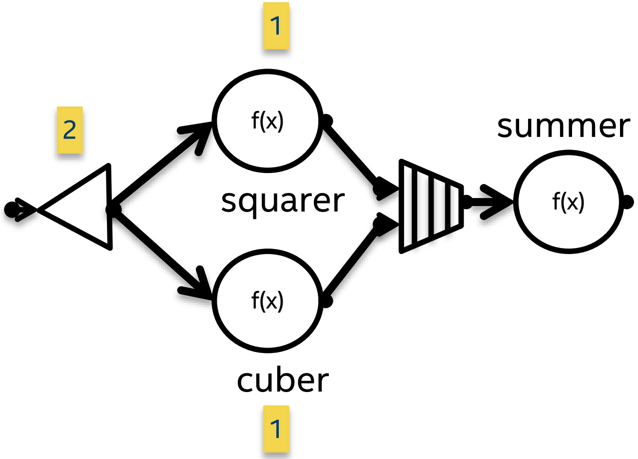
Data flow graph example




```
broadcast_node<int> input(g);
input.try_put(1);
```

Max concurrency = 1

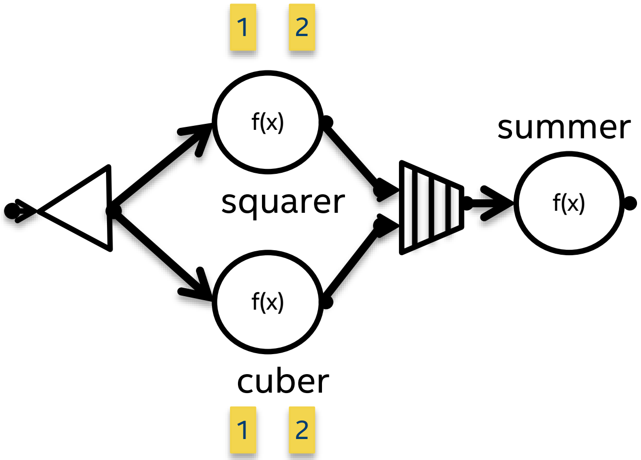
Data flow graph example




```
broadcast_node<int> input(g);
input.try_put(2);
```

Max concurrency = 3

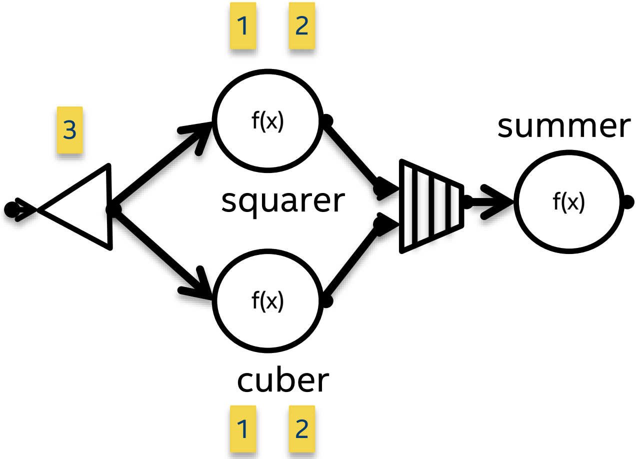
Data flow graph example




```
function_node<int, int> squarer( g, unlimited, square() );
function_node<int, int> cuber( g, unlimited, cube() );
```

Max concurrency = 5

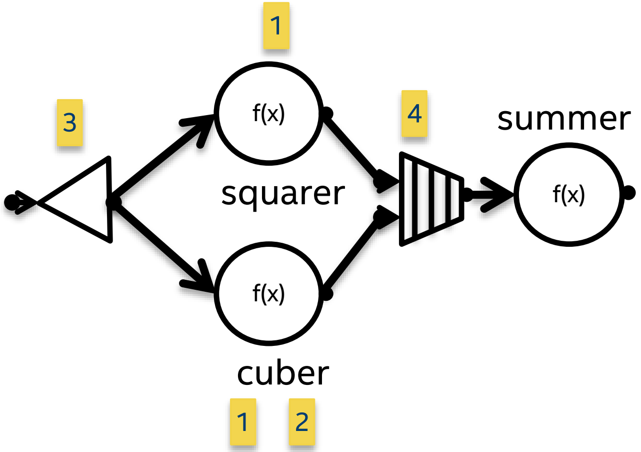
Data flow graph example




```
broadcast_node<int> input(g);
input.try_put(3);
```

Max concurrency = 5

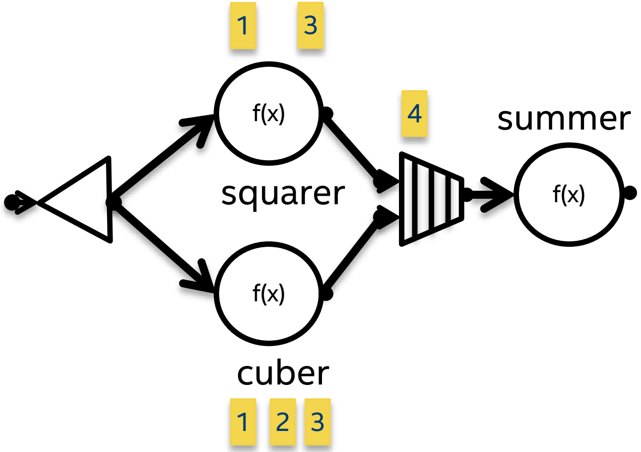
Data flow graph example




```
join_node< tuple<int, int>, queueing > j( g );
```

Max concurrency = 4

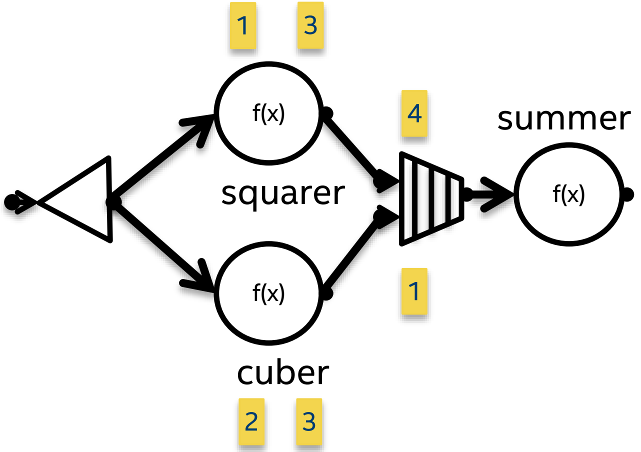
Data flow graph example




```
function_node<int, int> squarer( g, unlimited, square() );
function_node<int, int> cuber( g, unlimited, cube() );
```

Max concurrency = 6

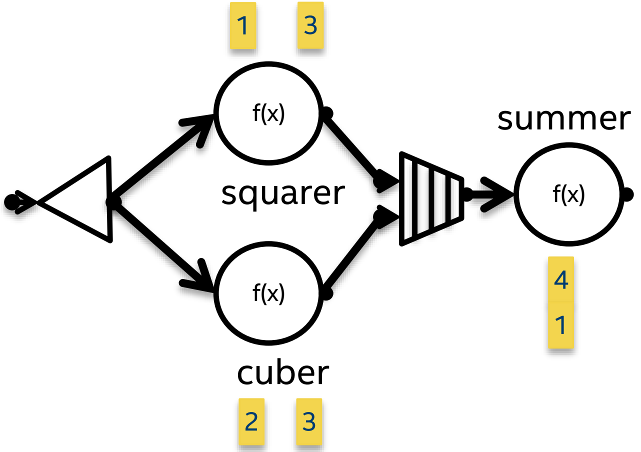
Data flow graph example




```
join_node< tuple<int, int>, queueing > j( g );
```

Max concurrency = 5

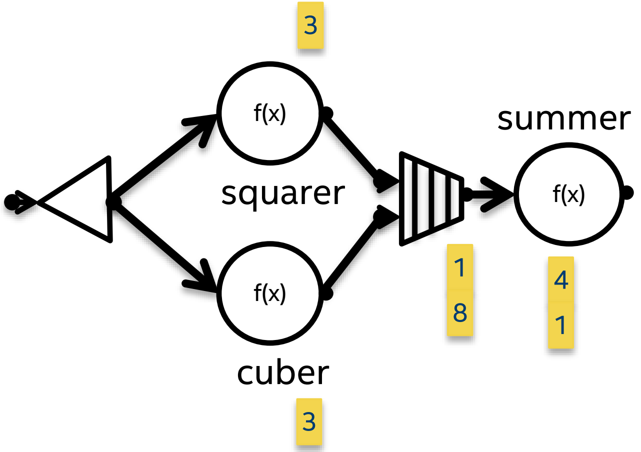
Data flow graph example




```
int result = 0;
function_node< tuple<int, int>, int >
      summer( g, serial, sum(result) );
```

Result = 0
Max concurrency = 6

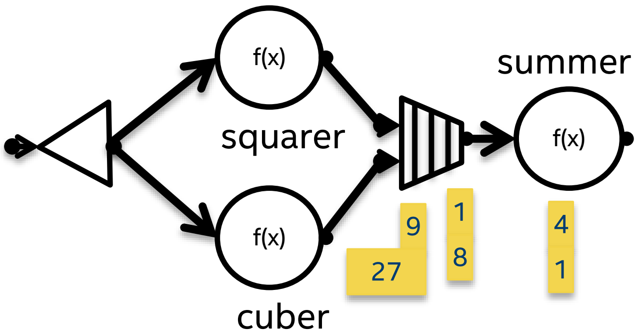
Data flow graph example




```
join_node< tuple<int, int>, queueing > j( g );
function_node< tuple<int, int>, int >
      summer( g, serial, sum(result) );
```

Result = 0
Max concurrency = 4

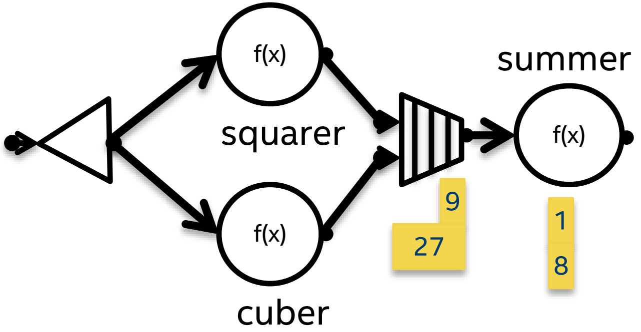
Data flow graph example




```
join_node< tuple<int, int>, queueing > j( g );
function_node< tuple<int, int>, int >
      summer( g, serial, sum(result) );
```

Result = 0
Max concurrency = 2

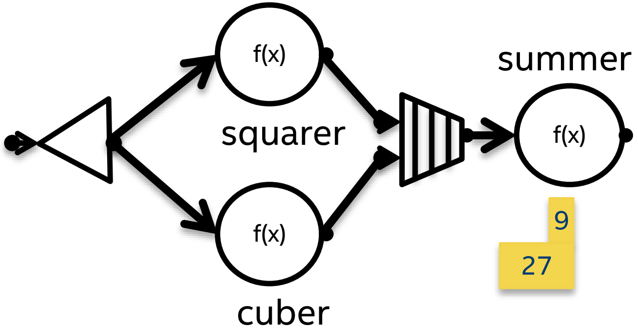
Data flow graph example




```
join_node< tuple<int, int>, queueing > j( g );
function_node< tuple<int, int>, int >
      summer( g, serial, sum(result) );
```

Result = 5
Max concurrency = 2

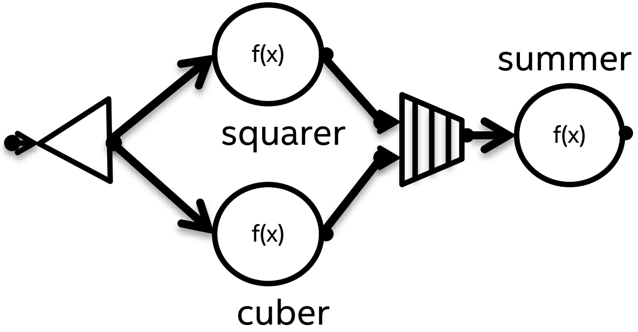
Data flow graph example




```
join_node< tuple<int, int>, queueing > j( g );
function_node< tuple<int, int>, int >
      summer( g, serial, sum(result) );
```

Result = 14
Max concurrency = 2

Data flow graph example

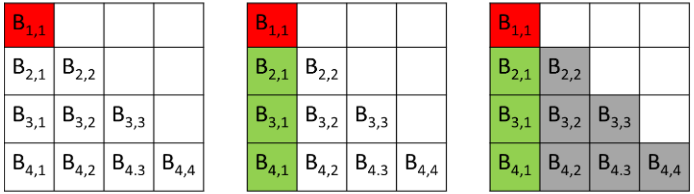



```
g.wait_for_all();
printf("Final result is %d\n", result);
```

Result = 50
Max concurrency = 1

Cholesky decomposition ($A = LL^T$)

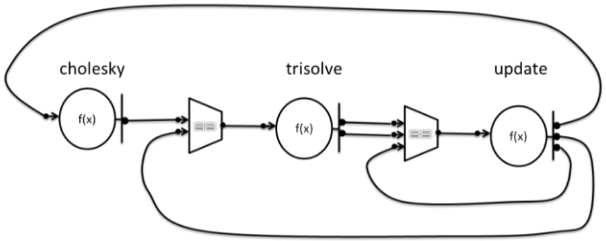
Aparna Chandramowlishwaran, Kathleen Knobe, and Richard Vuduc, “Performance Evaluation of Concurrent Collections on High-Performance Multicore Computing Systems”, 2010 Symposium on Parallel & Distributed Processing (IPDPS), April 2010.



(a) serial Cholesky Factorization

(b) Triangular solve

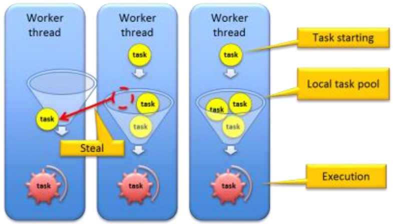
(c) Symmetric rank-k update



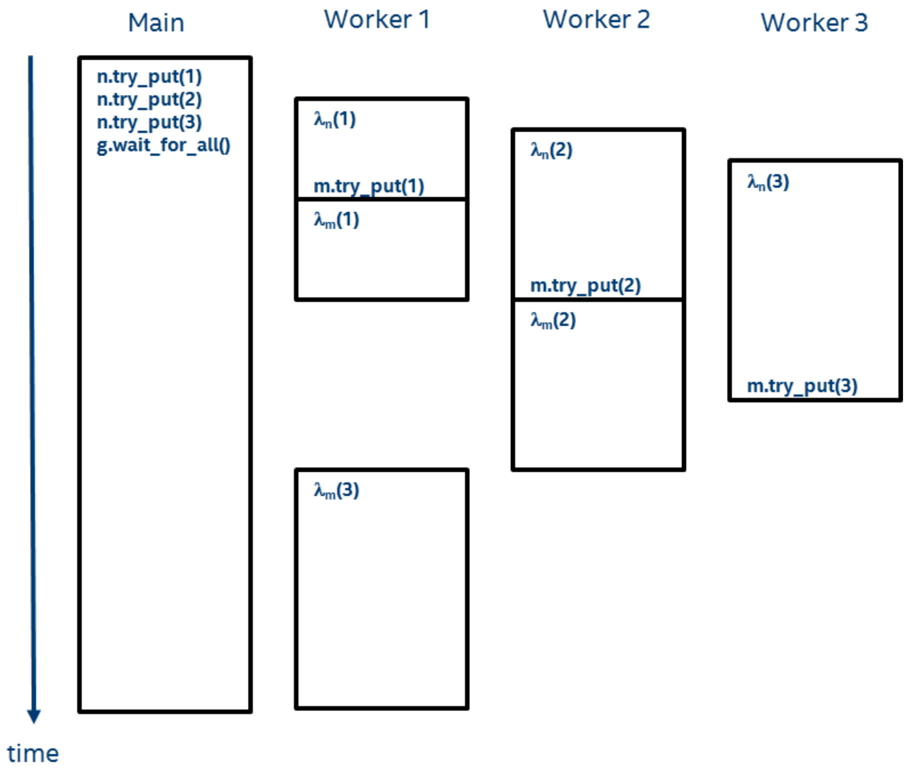
(a) flow based implementation

(b) dependence based implementation

How nodes map to Intel TBB tasks

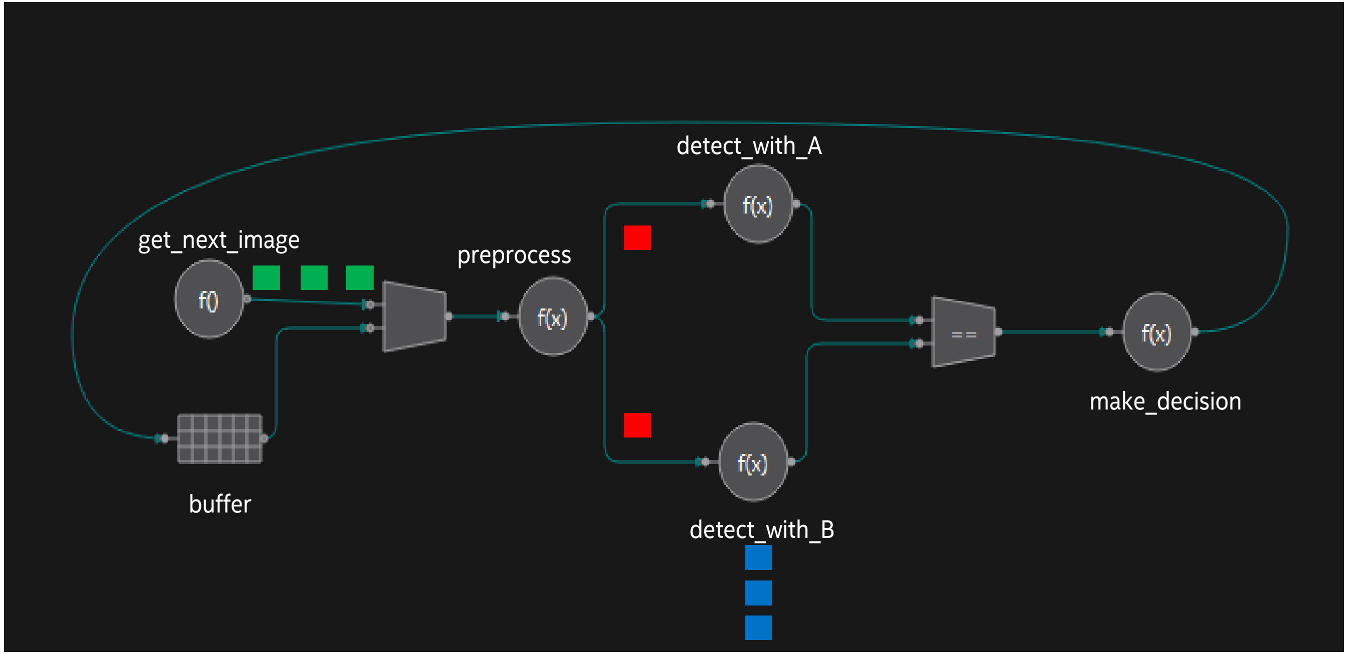



```
graph g;
function_node< int, int > n( g, unlimited,
    []( int v ) -> int {
        cout << v;
        spin_for( v );
        cout << v;
        return v;
    }
);
function_node< int, int > m( g, serial,
    []( int v ) -> int {
        v *= v;
        cout << v;
        spin_for( v );
        cout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



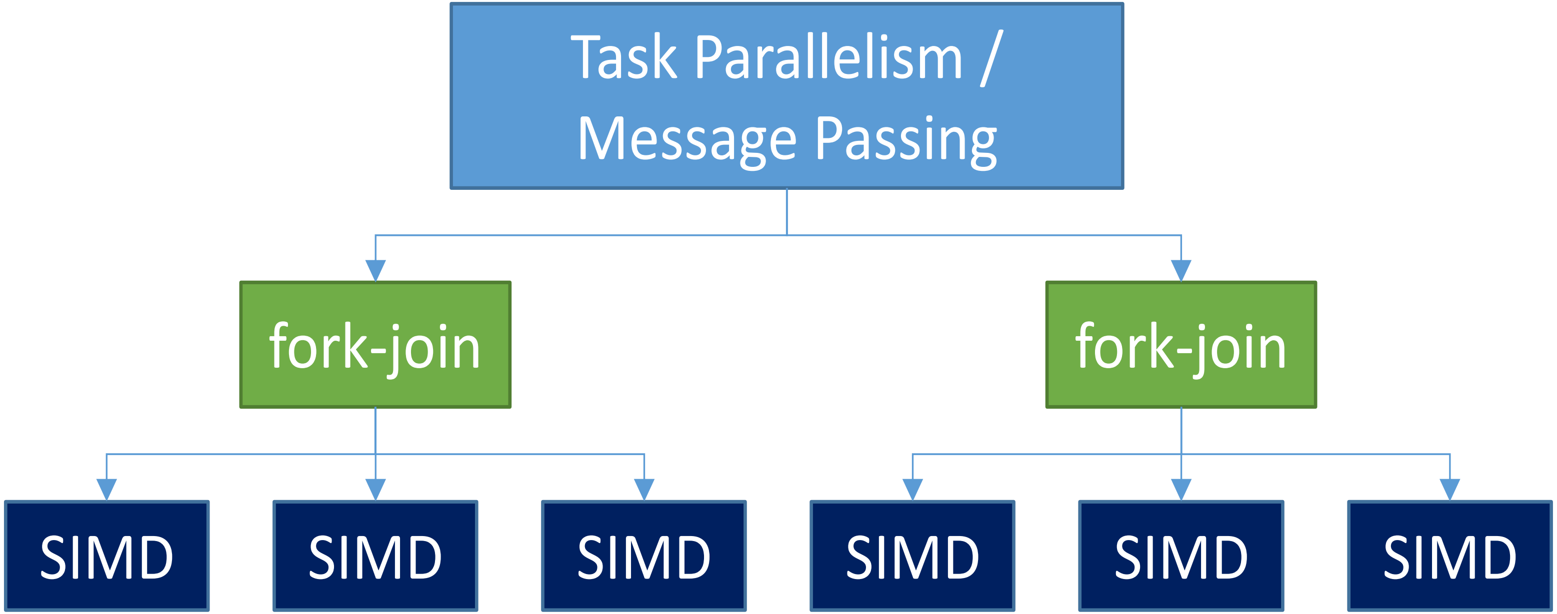
One possible execution – stealing is random

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

Possible Problems with Parallelism

Applying parallelism only for the innermost loop can be inefficient: scalability

- Over-synchronized: overheads become visible if there is not enough work inside
- Over-utilization: distribution to the whole machine can be inefficient
- Amdahl law: serial regions limit scalability of the whole program

Applying parallelism on the outermost level only:

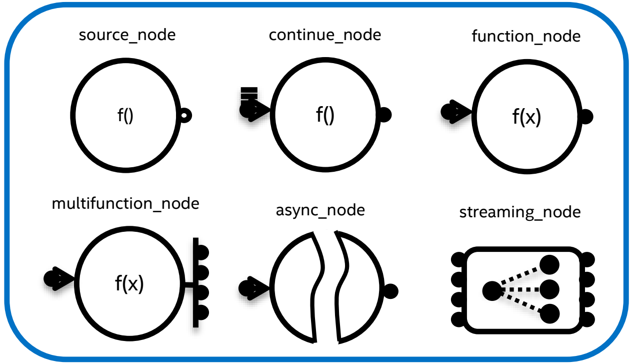
- Under-utilization: it does not scale if there is not enough tasks or/and load imbalance
- Provokes oversubscription if nested level is threaded independently&unconditionally

Frameworks can be used from both levels

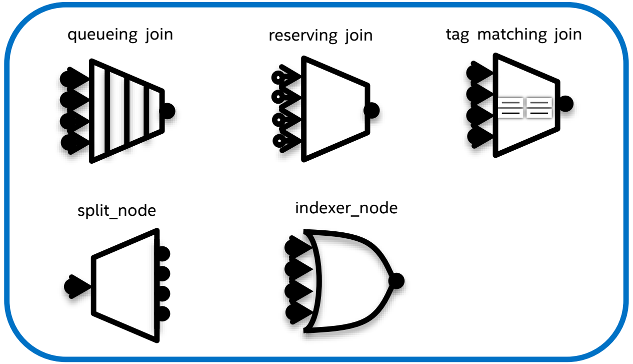
- To parallel or not to parallel? That is the question

Intel TBB Flow Graph node types:

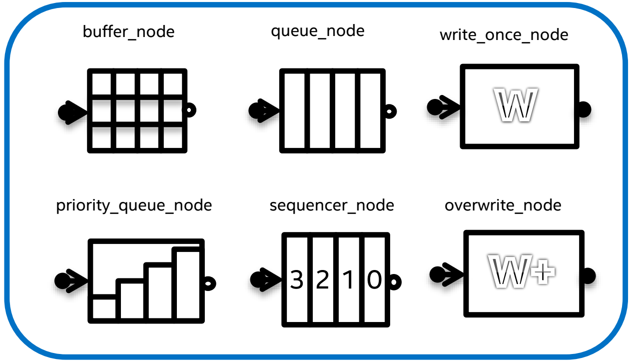
Functional



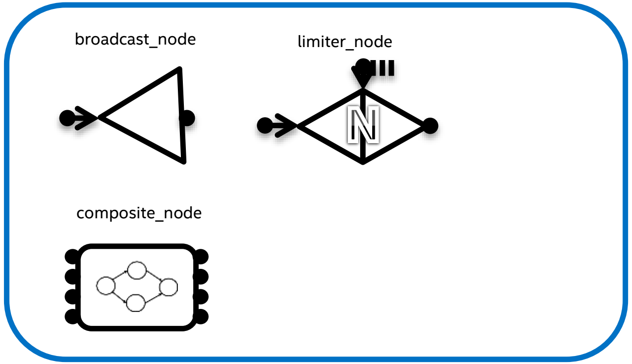
Split / Join



Buffering

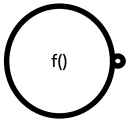


Other



Node types used in examples

source_node

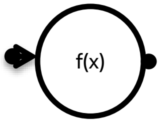


template < typename OutputType > class source_node;

template < typename Body > source_node::source_node(graph &g, Body body, bool is_active=true);

The Body is repeatedly invoked until it returns false. The Body updates one of its arguments.

function_node

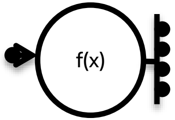


template < typename InputType, typename OutputType, graph_buffer_policy = queueing,
typename Allocator = cache_aligned_allocator<InputType> > class function_node;

template < typename Body > function_node::function_node(graph &g, size_t concurrency, Body body);

For each input message, outputs a single output message. Users can set concurrency limit and buffer policy.

multifunction_node



template <typename Input, typename Output, graph_buffer_policy = queueing,
typename Allocator=cache_aligned_allocator<Input> > class multifunction_node;

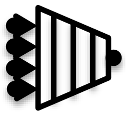
template < typename Body > multifunction_node::multifunction_node(graph &g, size_t concurrency, Body body);

For each input message, zero or more outputs can be explicitly put to the output ports from within the body. Users can set concurrency limit and buffer policy.

Node types used in examples

```
template < typename OutputTuple, class JoinPolicy = queueing > class  join_node;
```

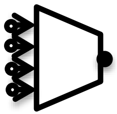
queueing join



```
join_node<OutputTuple, queueing>::join_node(graph &g);
```

Creates a tuple<T0, T1, ...> from the set of messages received at its input ports. Messages are joined in to the output tuple using a first-in-first-out policy at the input ports.

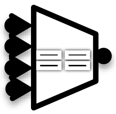
reserving join



```
join_node<OutputTuple, reserving>::join_node(graph &g);
```

Creates a tuple<T0, T1, ...> from the set of messages received at its input ports. The tuple is only created when a message can be reserved at a successor for each input port. A reservation holds the value in the predecessor without consuming it. If a reservation cannot be made at each input port all reservations are released. ***Useful when using a join to control resource usage – e.g. pairing with a limited number of buffers or tokens.***

tag matching join

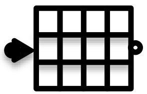


```
template < typename B0, typename B1, …. >
join_node<OutputTuple, tag_matching>::join_node(graph &g, B0 b0, B1 b1, …);
```

Creates a tuple<T0, T1, ...> from the set of messages received at its input ports. Tuples are created for messages with matching tags. The tags are calculated for each input message type by applying the corresponding user-provided functor. ***Useful when streaming in a graph, and related messages must be joined together.***

Node types used in examples

buffer_node

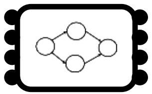


template < typename T, typename A = cache_aligned_allocator<InputType> > class buffer_node;

template < typename Body > buffer_node::buffer_node(graph &g);

An unbounded buffer of messages of type T.

composite_node



template<typename InputTuple, typename OutputTuple> class composite_node;

composite_node::composite_node(graph &g);

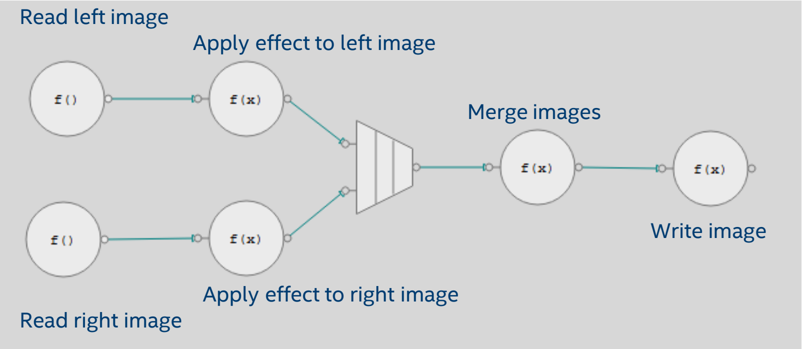
A node that encapsulates a sub-graph of other nodes, exposing input and output ports that are aliased to input and output ports of contained nodes.

Hands-on exercises

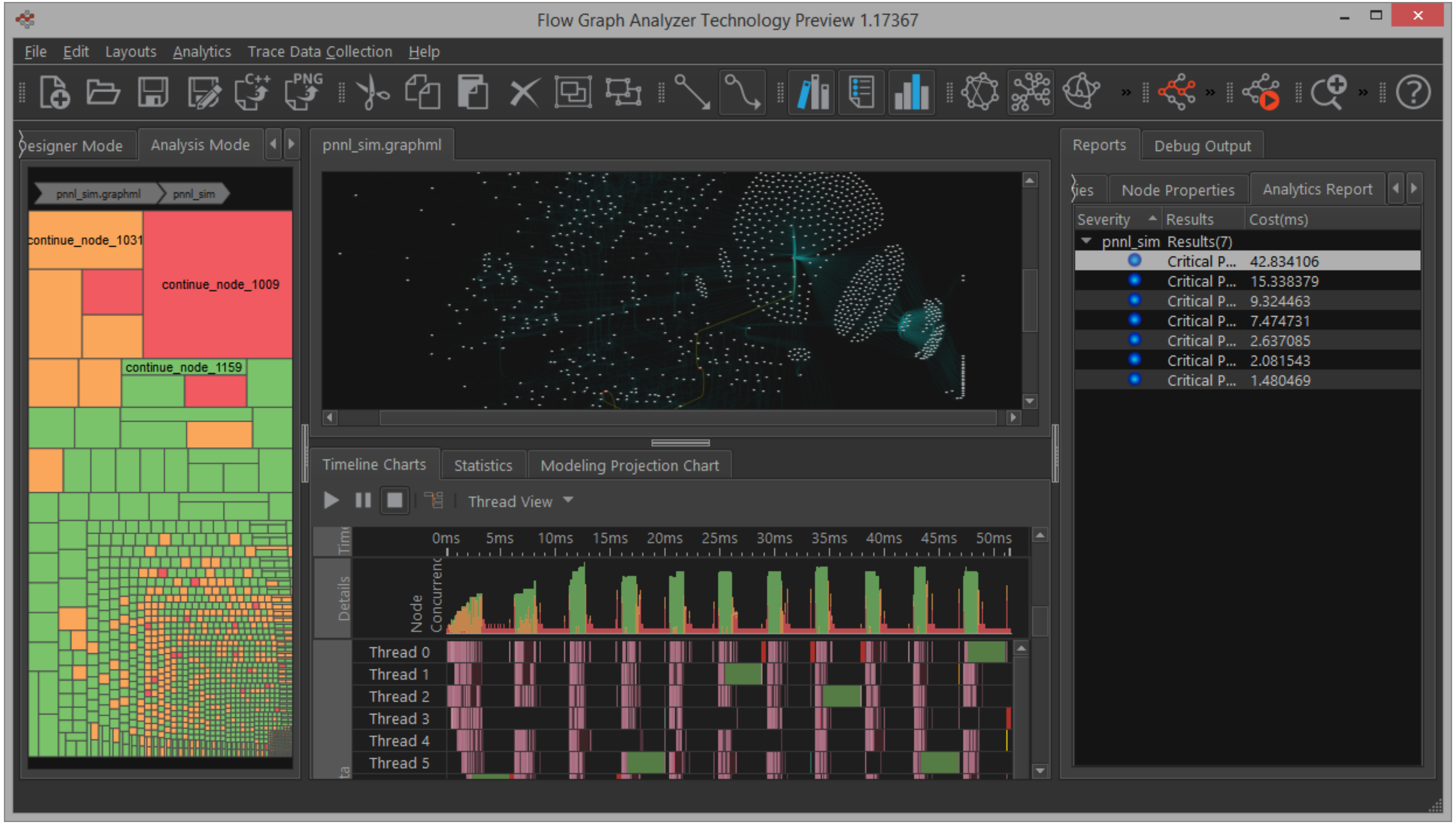
ex0: Inspect and execute the serial stereoscopic 3D example

1. Read left image
2. Read right image
3. Apply effect to left image
4. Apply effect to right image
5. Merge left and right images
6. Write out resulting image

ex1: Convert stereo example in to a TBB flow graph



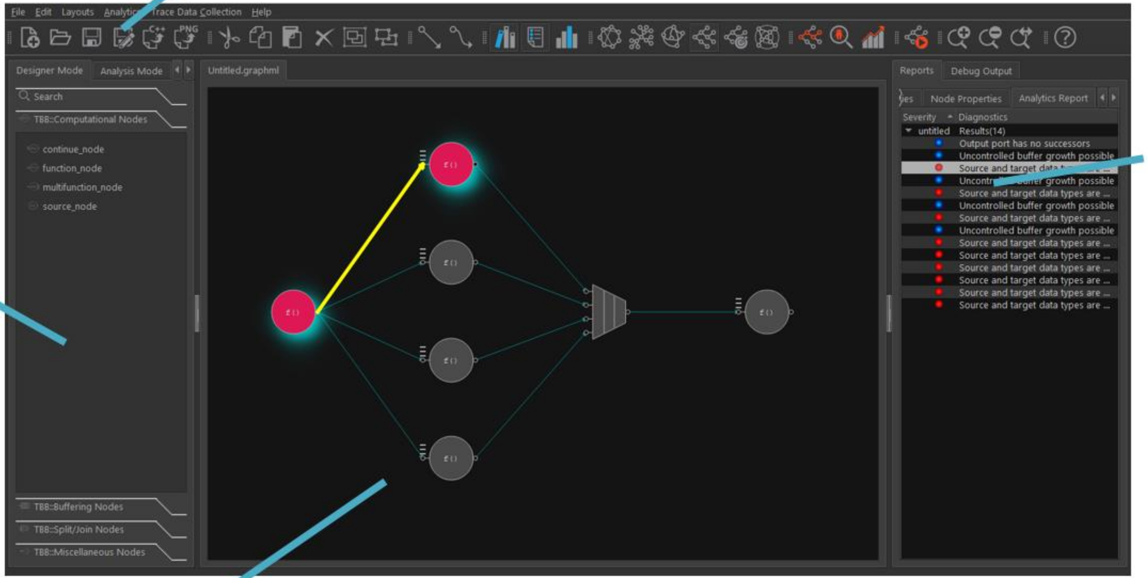
Flow Graph Analyzer for Intel® Threading Building Blocks



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

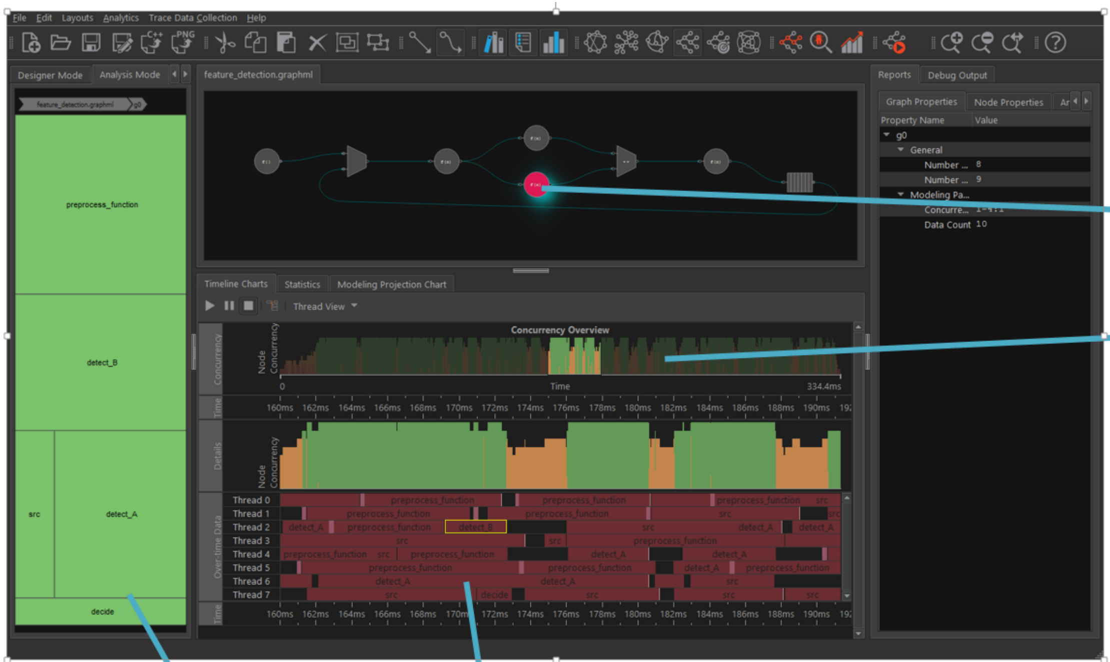
Palette of supported Intel® TBB node types organized in like groups



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.

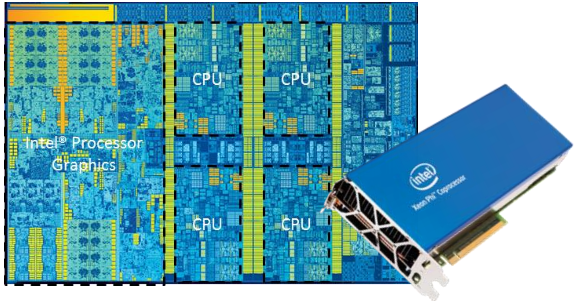
Treemap view gives you the general health of the graph's performance along with the ability to dive to the node level.

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

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…

\+

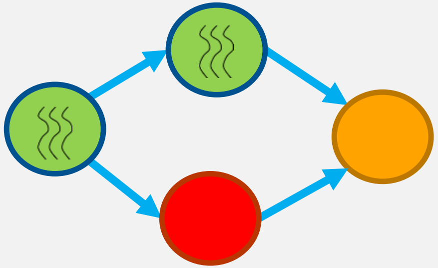
Intel® Threading Building Blocks
OpenVX*
OpenCL*
COI/SCIF
….

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

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

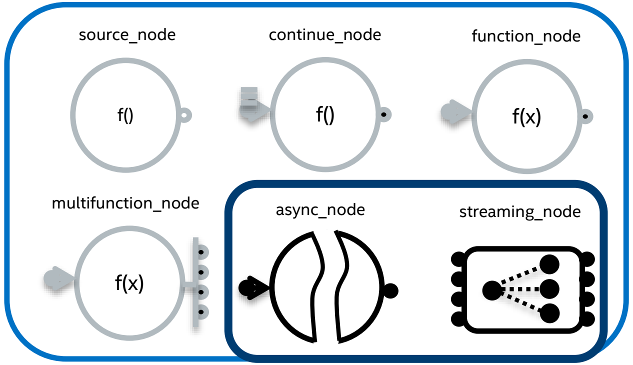
Intel TBB flow graph as a **coordination layer**

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

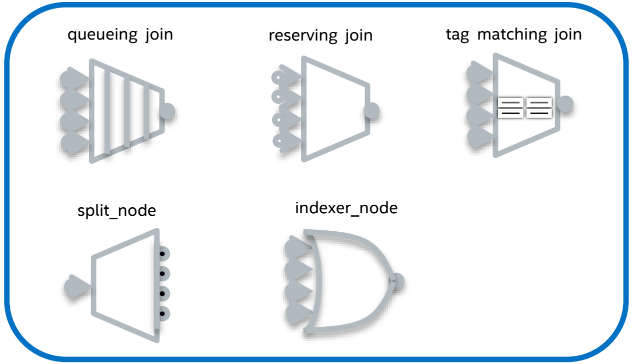


Intel TBB Flow Graph node types (for Heterogeneous Computing):

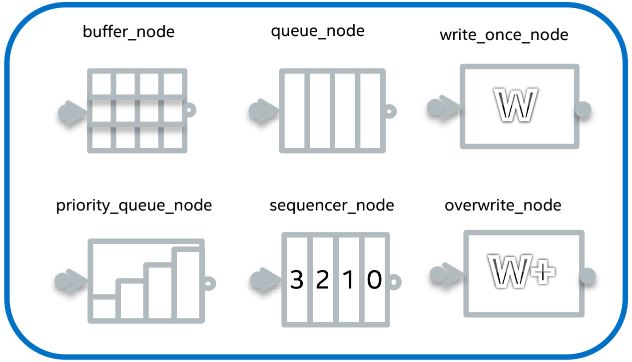
Functional



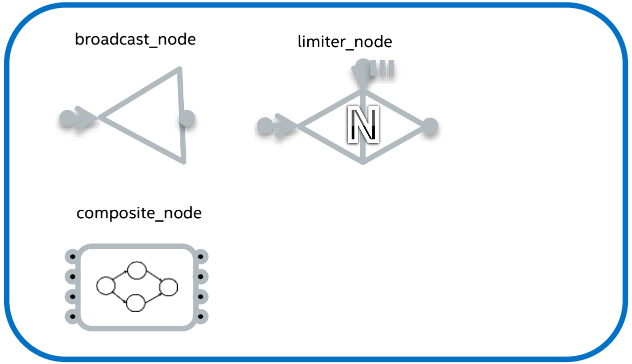
Split / Join



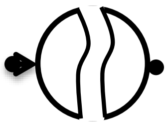
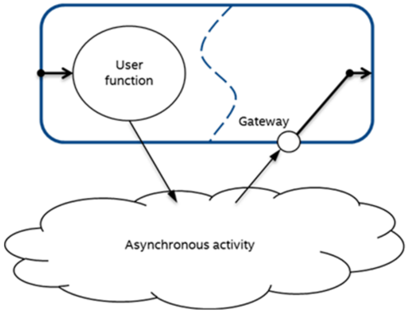
Buffering



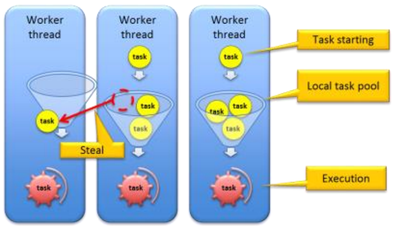
Other



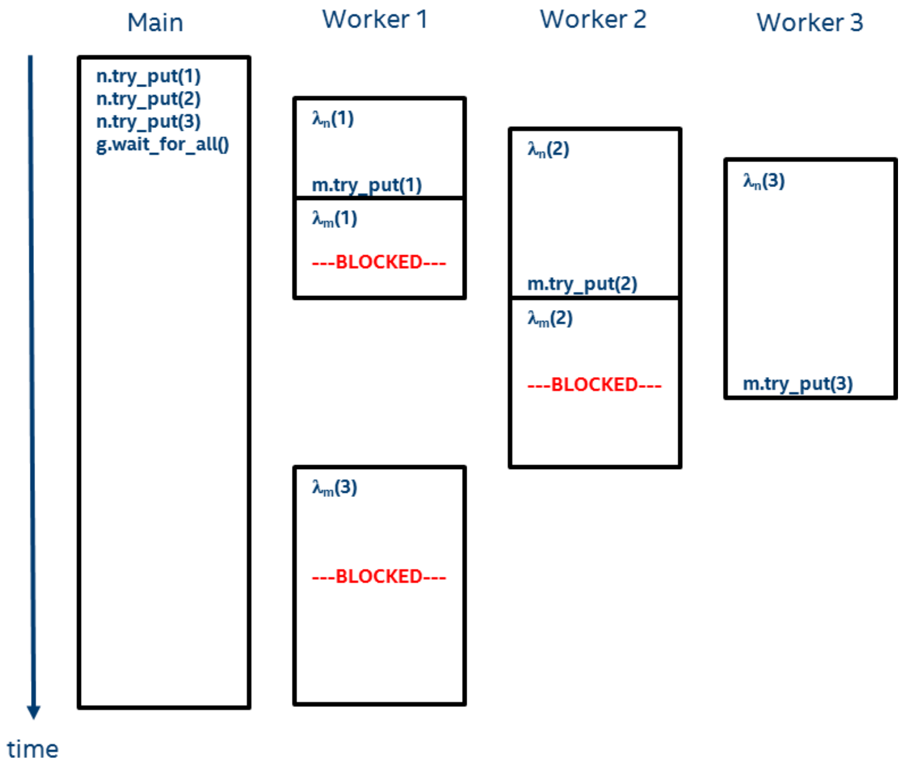
Heterogeneous support in the Intel TBB flow graph (1 of 3)

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.	

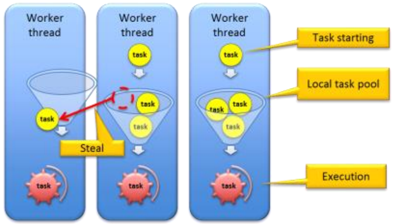
Why is extra support needed?




```
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();
```

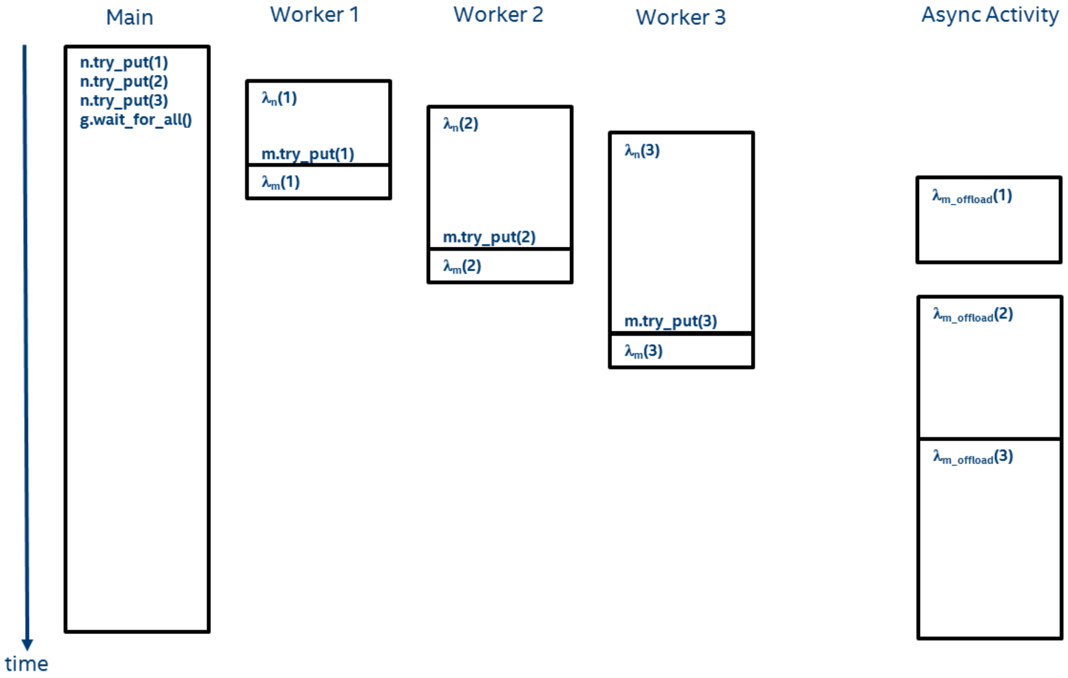


One possible execution – stealing is random



With async_node

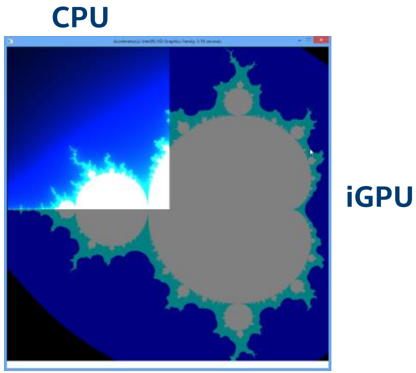
```
graph g;
my_async_activity_type my_async_activity;
function_node< int, int > n( g, unlimited, []( int v ) -> int {
   cout << v;
   spin_for( v );
   cout << v;
   return v;
} );
typedef typename async_node<int, int>::gateway_type gw_t;
async_node< int, int > m( g, serial, []( int v, gw_t &gw ) {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

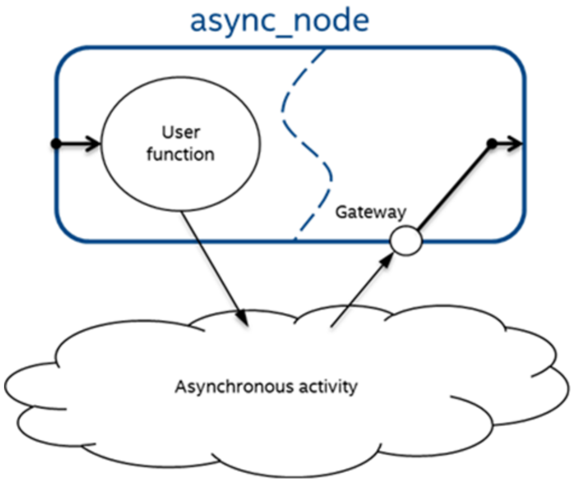


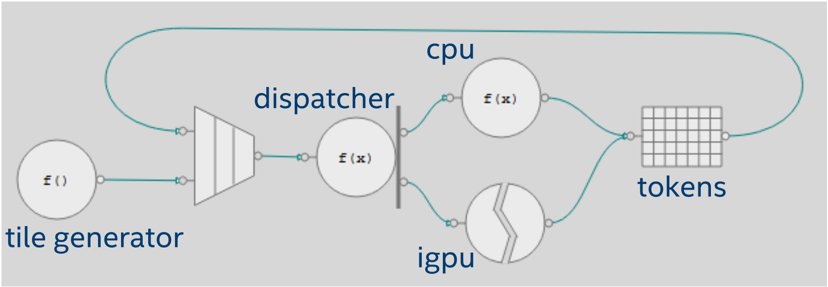
One possible execution – stealing is random

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

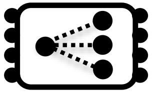
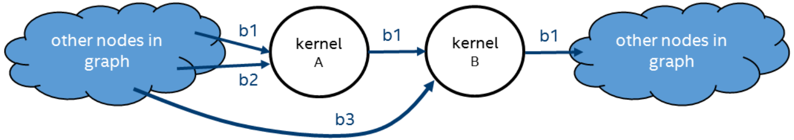
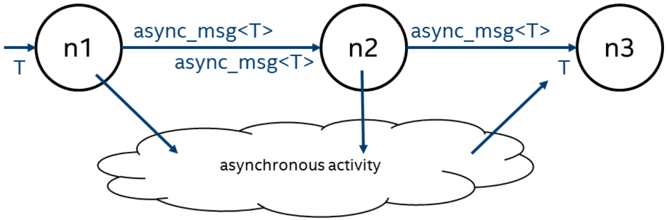




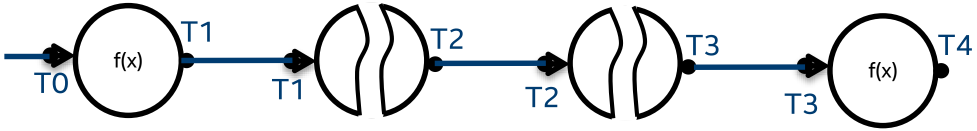


async_node makes coordinating with any model easier and efficient

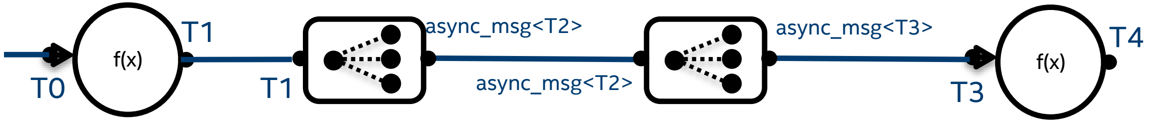
Heterogeneous support in the Intel TBB flow graph (2 of 3)

Feature	Description	Diagram
streaming_node *Available as preview feature* 	Higher level abstraction for streaming models; e.g. OpenCL*, Direct X Compute*, Vulkan*, 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

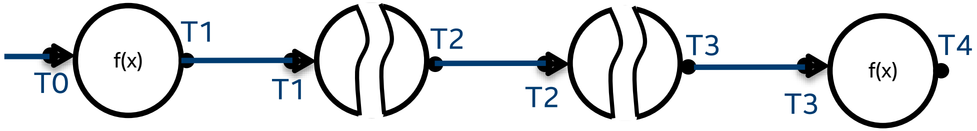


- async_node receives and sends unwrapped message types
- output message is sent after computation is done by asynchronous activity
- simpler to use when offloading a single computation and chaining is not needed

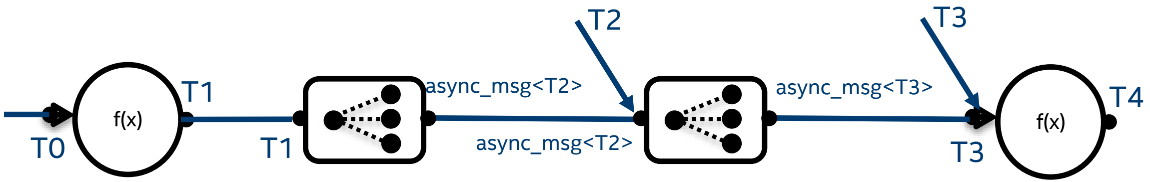


- streaming_node receives unwrapped message types or async_msg types
- sends async_msg types after enqueueing kernel, but (likely) before computation is done by asynchronous activity
- handles connections to non-streaming_nodes by deferring receive until value is set
- simple to use with pre-defined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

async_node vs streaming_node

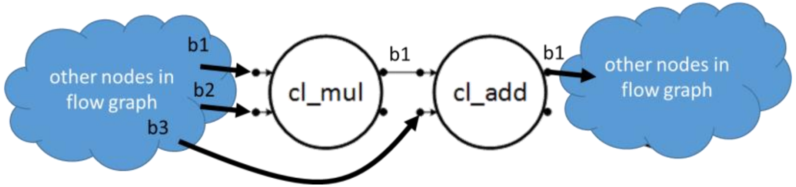


- async_node receives and sends unwrapped message types
- output message is sent after computation is done by asynchronous activity
- simpler to use when offloading a single computation and chaining is not needed

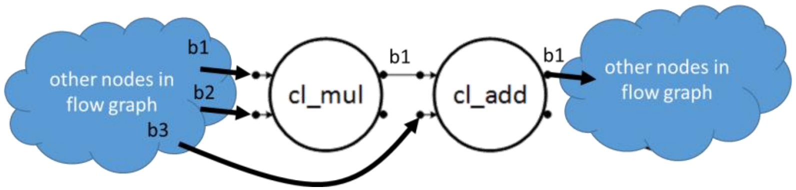


- streaming_node receives unwrapped message types or async_msg types
- sends async_msg types after enqueueing kernel, but (likely) before computation is done by asynchronous activity
- handles connections to non-streaming_nodes by deferring receive until value is set
- simple to use with pre-defined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



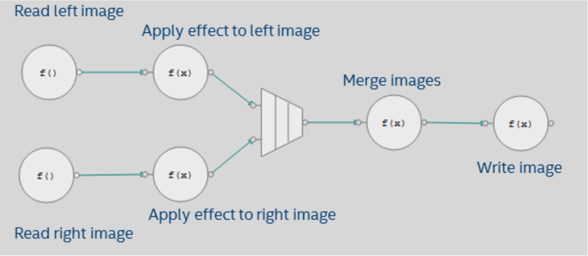
- Provides a first order node type that takes in OpenCL* programs or SPIR* binaries that can be executed on any supported OpenCL device
- Is a streaming_node with opencl_factory
- https://software.intel.com/en-us/blogs/2015/12/09/opencl-node-overview

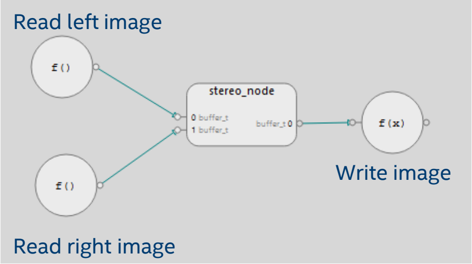
```
#define TBB_PREVIEW_FLOW_GRAPH_NODES 1
#define TBB_PREVIEW_FLOW_GRAPH_FEATURES 1

#include "tbb/flow_graph.h“
#include "tbb/flow_graph_opencl_node.h“

#include <algorithm>

int main() {
    using namespace tbb::flow;
    const char str[] = "Hello from ";
    opencl_buffer<cl_char> b( sizeof(str) );
    std::copy_n( str, sizeof(str), b.begin() );

    graph g;
    opencl_program<> program("hello_world.cl");
    opencl_node<tuple<opencl_buffer<cl_char>>>
        gpu_node(g, program.get_kernel("print"));

    std::array<unsigned int, 1> range{1};
    gpu_node.set_range(range);
    input_port<0>(gpu_node).try_put(b);

    g.wait_for_all();

    return 0;
}
__kernel void print( global char *str ) {
    for( ; *str; ++str ) printf( "%c", *str );
    printf( "GPU!\n" );
}
```

Optimization Notice

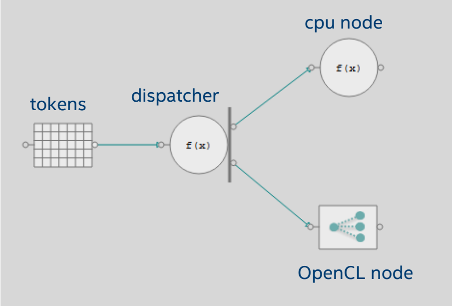
Hands-on exercises

ex2: Encapsulate stereo in a composite_node



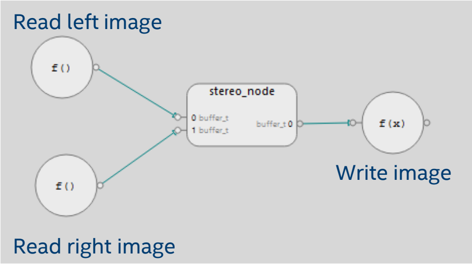


ex3: Hello OpenCL*

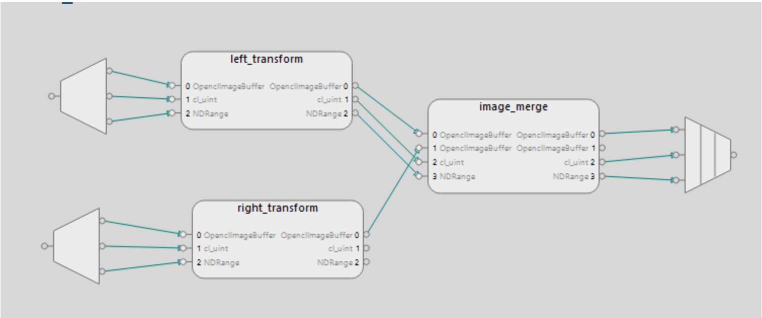


Hands-on exercises

ex4: Run OpenCL* Stereo

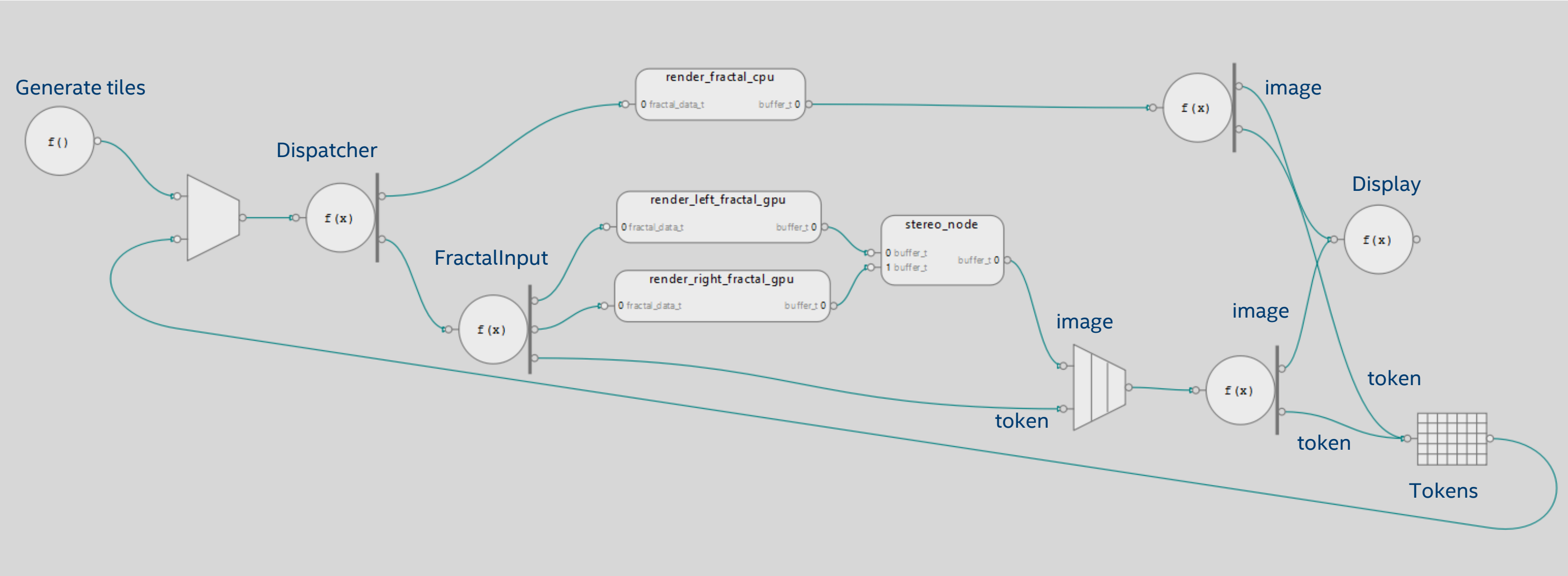


stereo_node



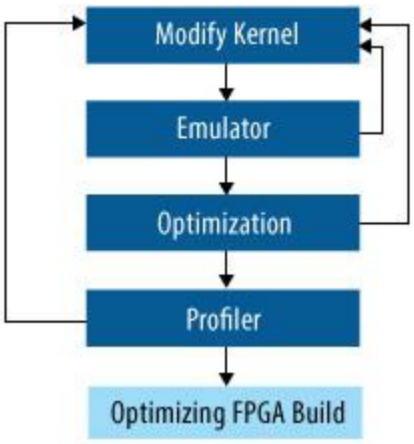
Hands-on exercises

ex5: Run a Stereoscopic 3D Fractal Generator that uses Tokens



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

Notes on opencl_node for FPGAs

- Current opencl_node executes a single kernel
- Communication is optimized between consecutive kernels through chaining
  - But this does not map well to FPGAs
  - Typically, FPGA kernels will communicate via channels or pipes
- Future work on OpenCL support for FPGAs
  - Define an API more appropriate for FPGAs
  - Multiple kernels in a single node
  - Kernels directly communicating through channels instead of async_msg through host
- async_node can be used for communication with FPGAs

Using other GPGPU models with Intel TBB

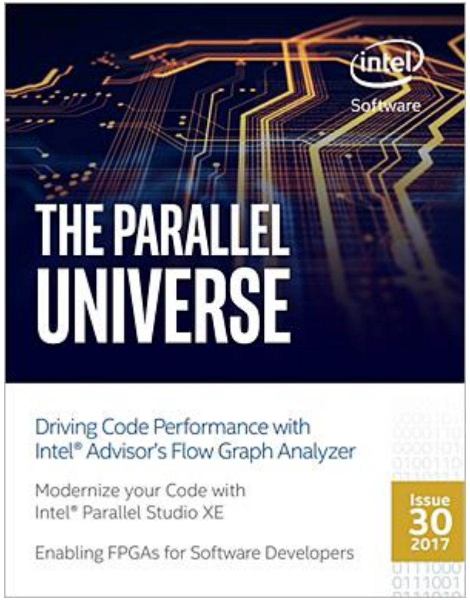
- CUDA*, Vulkan*, Direct Compute*, etc…
- Two approaches

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

To Learn More:

See Intel's The Parallel Universe Magazine

https://software.intel.com/en-us/intel-parallel-universe-magazine



http://threadingbuildingblocks.org

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

Contacts

- Ask questions:
  - By email: inteltbbdevelopers@intel.com
  - Use forum: https://software.intel.com/en-us/forums/intel-threading-building-blocks
- Create pull requests:
  - https://github.com/01org/tbb

Legal Disclaimer & 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

intel®

experience what's inside™

BACKUP

A simple asynchronous activity with async_node

1. We need an asynchronous activity
   - Can receive an incoming message without blocking
   - Executes work outside of the context of the task that sent the message
   - Can send result back through a call to async_node gateway.
   - Graph lifetime must be managed
2. We need to implement an async_node body
   - Passes incoming message and gateway to asynchronous activity
   - Does not block waiting for message to be processed
3. We need to build and execute the flow graph

1. We need an asynchronous activity

```
template <typename MessageType>
class user_async_activity {
public:
   static user_async_activity* instance();
   static void destroy();
   void addWork( const MessageType& msg );

private:
   user_async_activity();
   struct my_task { .... };
   static void threadFunc(user_async_activity<MessageType>* activity) :
      myThread(&user_async_activity::threadFunc, this) {
   tbb::concurrent_bounded_queue<my_task>  myQueue;
   std::thread                              myThread;
   static user_async_activity*          s_Activity;
};
```

```
template<typename AsyncNodeType>
class user_async_msg {
public:
   typedef typename AsyncNodeType::input_type input_type;
   typedef typename AsyncNodeType::gateway_type gateway_type;
   user_async_msg() : mGateway(NULL) {}
   user_async_msg(const input_type& input, gateway_type &gw) :
      mInputData(input), mGateway(&gw) {}
   const input_type& getInput() const { return mInputData; }
   gateway_type& getGateway() const { return *mGateway; }

private:
   input_type mInputData;
   gateway_type *mGateway;
};
```

1. We need an asynchronous activity

```
template< typename MessageType >
void user_async_activity<MessageType>::addWork(const MessageType& msg) {
   msg.getGateway().reserve_wait();
   myQueue.push(my_task(msg));
}

template< typename MessageType >
void user_async_activity<MessageType>::threadFunc(user_async_activity<MessageType>* activity) {
   my_task work;

   for(;;) {
      activity->myQueue.pop(work);
      if (work.myFinishFlag) {
         std::cout << "async activity is done." << std::endl;
         break;
      } else {
         std::cout << work.myMsg.getInput() << ' ' << std::flush;
         typename MessageType::gateway_type &gw = work.myMsg.getGateway();
         gw.try_put(std::string("Processed: ") + work.myMsg.getInput());
         gw.release_wait();
      }
   }
}
```

2. We need to implement an async_node body

```
int main() {
   typedef async_node<std::string, std::string> node_t;
   typedef user_async_msg< node_t > msg_t;
   typedef user_async_activity<msg_t> activity_t;

   graph g;
   node_t node(g, unlimited, [](const node_t::input_type &s, node_t::gateway_type &gw) {
      activity_t::instance()->addWork(msg_t(s, gw));
   });

   std::string final;
   function_node< std::string > destination(g, serial, [&final](const std::string& result) {  final += result + "; ";  });

   make_edge(node, destination);
   node.try_put("hello");
   node.try_put("world");

   g.wait_for_all();
   activity_t::destroy();
   std::cout << std::endl << "done" << std::endl << final << std::endl;
   return 0;
}
```

Optimization Notice

3. We need to build and execute the flow graph

```
int main() {
   typedef async_node<std::string, std::string> node_t;
   typedef user_async_msg< node_t > msg_t;
   typedef user_async_activity<msg_t> activity_t;

   graph g;
   node_t node(g, unlimited, [](const node_t::input_type &s, node_t::gateway_type &gw) {
      activity_t::instance()->addWork(msg_t(s, gw));
   });

   std::string final;
   function_node< std::string > destination(g, serial, [&final](const std::string& result) {  final += result + "; ";  });

   make_edge(node, destination);
   node.try_put("hello");
   node.try_put("world");

   g.wait_for_all();
   activity_t::destroy();
   std::cout << std::endl << "done" << std::endl << final << std::endl;
   return 0;
}
```

3. We need to build and execute the flow graph

```
int main() {
    typedef async_node<std::string, std::string> node_t;
    typedef user_async_msg< node_t > msg_t;
    typedef user_async_activity<msg_t> activity_t;

    graph g;
    node_t node(g, unlimited, [](const node_t::input_type &s, node_t::gateway_type &gw) {
        activity_t::instance()->addWork(msg_t(s, gw));
    });

    std::string final;
    function_node< std::string > destination(g, serial, [&final](const std::string& result) {  final += result + "; ";  });

    make_edge(node, destination);
    node.try_put("hello");
    node.try_put("world");

    g.wait_for_all();
    activity_t::destroy();
    std::cout << std::endl << "done" << std::endl << final << std::endl;
    return 0;
}
```

```
Tutorial> ./async_node.exe
hello world async activity is done.

done
Processed: hello; Processed: world;
Tutorial>
```

A simple asynchronous activity with streaming_node

1. We need an asynchronous activity
   - Can receive an incoming async_msg message without blocking
   - Executes work outside of the context of the task that sent the message
   - Sets result in the async_msg
   - Graph lifetime must be managed
2. We need to implement device_factory and device_selector
   - Passes incoming message and gateway to asynchronous activity
   - Does not block waiting for message to be processed
3. We need to build and execute the flow graph

1. We need an asynchronous activity

```
template <typename MessageType>
class user_async_activity {
public:
   static user_async_activity* instance();
   static void destroy();
   void addWork( const MessageType& msg );

private:
   user_async_activity();
   struct my_task { .... };
   static void threadFunc(user_async_activity<MessageType>* activity) :
      myThread(&user_async_activity::threadFunc, this) {
   tbb::concurrent_bounded_queue<my_task>  myQueue;
   std::thread                             myThread;
   static user_async_activity*             s_Activity;
};
```

```
template<typename T>
class user_async_msg : public tbb::flow::async_msg<T>
{
public:
   typedef tbb::flow::async_msg<T> base;
   user_async_msg() : base() {}
   user_async_msg(const T& input) : base(), mInputData(input) {}
   const T& getInput() const { return mInputData; }

private:
   T mInputData;
};
```

Inherits a set(const T& v) function from async_msg

1. We need an asynchronous activity

```
template< typename MessageType >
void user_async_activity<MessageType>::addWork(const MessageType& msg) {
   myQueue.push(my_task(msg));
}

template< typename MessageType >
void user_async_activity<MessageType>::threadFunc(user_async_activity<MessageType>* activity) {
   my_task work;
   for(;;) {
      activity->myQueue.pop(work);
      if (work.myFinishFlag)
         break;
      else {
         std::cout << work.myMsg.getInput() << ' ';
         work.myMsg.set("Processed: " + work.myMsg.getInput());
      }
   }
}
```

Note: Unlike with async_node example, the graph lifetime is not managed by activity

2. We need to implement device_factory and device_selector

```
class device_factory {
public:
   typedef int device_type;
   typedef int kernel_type;

   device_factory(graph &g) : mGraph(g) {}

   /* ... some empty definitions ... */

   void send_kernel( device_type /*device*/, const kernel_type& /*kernel*/, user_async_msg<std::string>& msg) {
      mGraph.increment_wait_count();
      activity_t::instance()->addWork(msg);
   }
private:
   graph &mGraph;
};

template<typename Factory>
class device_selector {
public:
   typename Factory::device_type operator()(Factory&) { return 0; }
};
```

3. We need build and execute the flow graph

```
int main() {
     typedef streaming_node< tuple<std::string>, queueing, device_factory > streaming_node_type;

    graph g;
    device_factory factory(g);
    device_selector<device_factory> device_selector;

    streaming_node_type node(g, 0, device_selector, factory);

    std::string final;
    function_node< std::string > destination(g, serial, [&g,&final](const std::string& result) {
        final += result + "; ";
        g.decrement_wait_count();
    });

    make_edge(node, destination);
    input_port<0>(node).try_put("hello");
    input_port<0>(node).try_put("world");

    g.wait_for_all();
    activity_t::destroy();

    std::cout << std::endl << "done" << std::endl << final << std::endl;
    return 0;
}
```

3. We need build and execute the flow graph

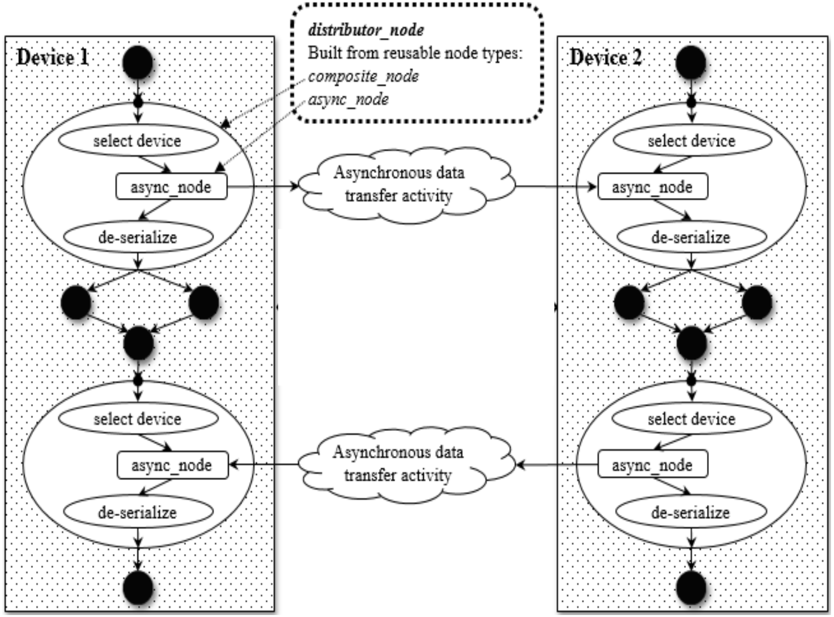
```
int main() {
    typedef streaming_node< tuple<std::string>, queueing, device_factory > streaming_node_type;

   graph g;
   device_factory factory(g);
   device_selector<device_factory> device_selector;

   streaming_node_type node(g, 0, device_selector, factory);

   std::string final;
   function_node< std::string > destination(g, serial, [&g,&final](const std::string& result) {
      final += result + "; ";
      g.decrement_wait_count();
   });

   make_edge(node, destination);
   input_port<0>(node).try_put("hello");
   input_port<0>(node).try_put("world");

   g.wait_for_all();
   activity_t::destroy();

   std::cout << std::endl << "done" << std::endl << final << std::endl;
   return 0;
}
```

3. We need build and execute the flow graph

```
int main() {
    typedef streaming_node< tuple<std::string>, queueing, device_factory > streaming_node_type;

    graph g;
    device_factory factory(g);
    device_selector<device_factory> device_selector;

    streaming_node_type node(g, 0, device_selector, factory);

    std::string final;
    function_node< std::string > destination(g, serial, [&g,&final](const std::string& result) {
        final += result + "; ";
        g.decrement_wait_count();
    });

    make_edge(node, destination);
    input_port<0>(node).try_put("hello");
    input_port<0>(node).try_put("world");

    g.wait_for_all();
    activity_t::destroy();

    std::cout << std::endl << "done" << std::endl << final << std::endl;
    return 0;
}
```

```
Tutorial> ./streaming_node.exe
hello world
done
Processed: hello; Processed: world;
Tutorial>
```

Support for Distributed Programming

Feature	Description	Diagram
distributor_node *Proof of concept*	Enables communication between different memory domains. Each device is capable of running a graph; e.g. hosts, Xeon Phi cards, etc… Graphs runs on all devices. Communication can be initiated from any device to any device. Whole sub-graphs may execute on a device between communication points.	

Streamed FFT example

The host generates 4000 arrays of floating point numbers

On each array, FFT is performed (serially)

Execution of FFT is offloaded to KNC

Parallelism comes from multiple arrays processed at the same time