Intel Thread Building Blocks, Part V

SPD course 2017-18
Massimo Coppola
23/04/2018

UNIVERSITÀ DI PISA
ISTITUTO DI SCIENZA E TECNOLOGIE DELL'INFORMAZIONE "A. FAEDO"

The Flow Graph

- Allow fast & efficient implementation of dataflow, dependency graph algorithms
  - Introduced in TBB 4
  - Evolution of the pipeline idea
- Computation represented as
  - A **`graph`** object
  - A set of *nodes* : computation units
    - one or more inputs and outputs
  - A set of *edges* : comm. channel AND dependencies

- loops? Yes, but examples are mostly DAGS
- Node execution = TBB task instantiation
- Namespace **`tbb::flow`**

The graph object

- Dynamically created by program code, using node and edge constructor methods
  - Can be run multiple times
  - Owns all tasks created during the flow graph execution
  - Executes its tasks either in a specified **`task_group_context`**, or in a newly created context
- Feeding the graph is done via enqueueing data
  - However, a less than trivial protocol is used to let the node communicate with each other with low overhead
- Interactions
  - Waiting for the graph to finish its computation
  - Registering interactions with the graph
    - Will actually cause tasks to be run within the graph
- Most examples are DAGS, but this is not mandatory
  - Generic looping graphs are much harder to design & debug

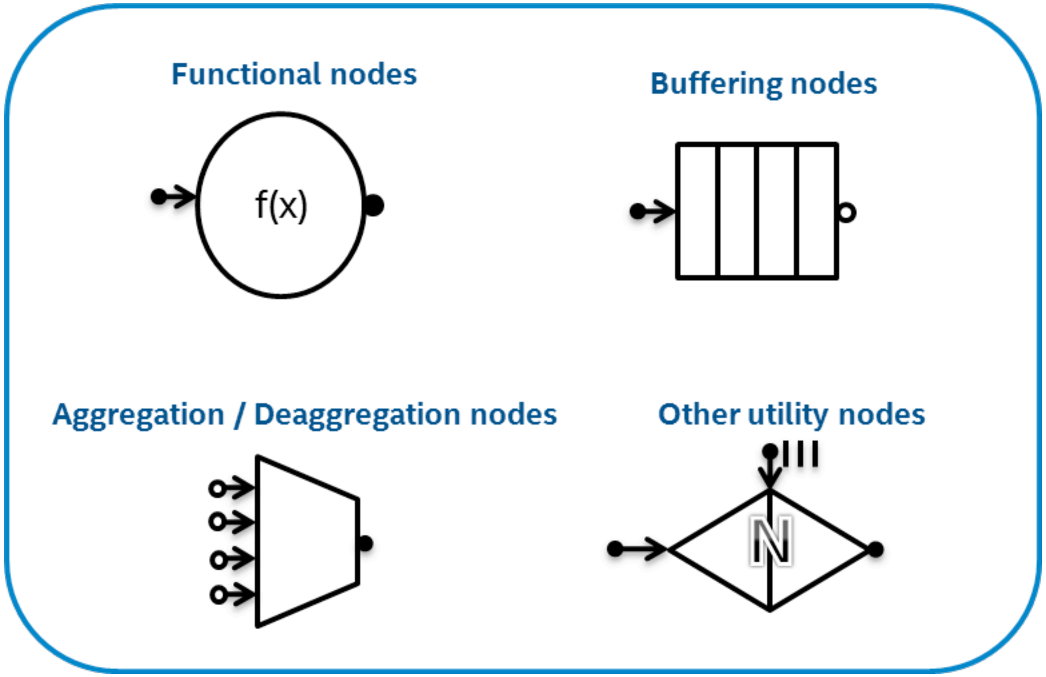
Types of messages

- continue_msg
  - Empty class used for dependency messages
- flow::tuple
  - Used to manage messages built of many parts
  - Supports a subset of the methods of the std::tuple
- class tagged_msg
  - Template to add a tag to a multipart message
  - A specified TagType is used to inform the receiver on the content of the message, which may be only known at runtime

  - ```
    template<typename TagType, typename T0,
    typename T1...typename TN>
    class tagged_msg;
    ```
    ```

TBB Flow Graph Nodes

- Several types of nodes
  - Functional
  - Buffering, filtering of messages
  - Aggregation/deaggregation (broadcast, order)
  - Utility



TBB Flow Graph Nodes

- Several types of nodes
  - Functional
  - Buffering, filtering of messages
  - Aggregation/deaggregation (broadcast, order)
  - Utility
- Node input and output types are defined at creation via template parameters
  - Multiple inputs are managed via tuples an read with get<0>, get<1> …
- Node invoke user-provided functions
  - Executed as tasks, so choose wisely their grain
- Can also be created in *inactive* state and be *activated* later on
  - Pay attention to the creation order (e.g. use reverse dependency order), or risk losing messages

Class node abstract templates

- templates helping define different types of nodes
  - abstract classes with default implementation of some methods
  - you may have to redefine some virtual methods
  - register and remove methods are for TBB internal use!
- **`Graph_node`** base template class
- **`Sender`** template class
  - Nodes that act as data/message sender
- **`Receiver`** template class
- **`Continue_receiver`**
  - Receives multiple continue_msg, computes when the number of messages hits the set threshold
- **`execute()`** is Triggered by predecessors' calling **`try_put()`**

“Functional” nodes

- These nodes compute a function
  - of the predecessor(s) input(s) if any are connected
  - send the results (data or empty message) to the successor(s)
- **Continue_node**
  - Awaits one or mode dependency messages in input
  - Performs a computation and brodcasts a data/dep message to its successors
- **Function_node**
- **Source_node**
  - Strictly **serial** node, no predecessors, user function will generate messages that are broadcast to successors
- **Multifunction_node**
  - One input, multiple output broadcast to successors
  - can be assigned a concurrency limit
- **Asnc_node**
  - One input, one output, obeys concurrency limit
  - Forward messages ***outside*** TBB for external processing
  - Provides a gateway tpe to return back results

Buffering nodes

- **Overwrite_node**
  - Single item buffer, can overwrite
- **Write_once**
  - Single item buffer, no overwrite unless clear() is called
- **Buffer_node**
  - Unbounded buffer (arbitrary order) toward a single successor *
  - Accepts a reservation
- **Queue_node**
  - Unbounded FIFO queue toward a single successor *
  - Accepts a reservation, will stall the queue
- **Priority_queue_node**
  - Uses a priority queue to a successor *, reservation will stall queue
- **Sequencer_node**
  - Unbounded buffer toward a successor *
  - Sends message in strict 0..N sequence order
  - Will reject duplicate sequence numbers

- A single successor:*
  - Sends messages to 1st registered successor, when one msg is refused, ignore that successor, try next one (if any)

Service nodes

- **Join** nodes
  - Create a tuple <T0 .. Tn> from messages received at its inputs, broadcast the tuple to all its successors
- Multifunction_node
  - Has input and a tuple of outputs
  - May spawn a new task at each input received
  - Up to a degree of concurrency **if predefined**
- Split_node
  - Input is a tuple, and has a tuple of outputs
  - Each component of the input tuple is sent to teh corresponding output
- Indexer_node
  - Broadcast to all output each message received on any input
  - Message is tagged with the input index
- Composite_node
  - Encapsulates a collection of (any nuber of) other nodes
  - Requires C++11
  - A tuple of inputs and a tuple of outputs forward messages in and out
  - Can also be specialized to only inputs or only outputs

TBB flow graph edges

- Created with the method **`make_edge( srcnode, destnode)`**
- Encode node dependencies
  - Use class **`continue_msg`** to activate successor nodes
- Express communications
  - A data message to a successor node activates it
  - Data sent is copied, so send *references* to large data items whenever it is possible
- dataflow-style activation, i.e.
  - when all inputs are present
  - independent nodes can run concurrently

Message communication protocol

- Issue with push/pull protocol
  – Nodes will switch between push message forwarding and pull forwarding to avoid the need of retries

- Potential message discard if no receiver accepts
  – Some of the nodes do not buffer the message, so if no successor accepts the message can be lost

Node push/pull/buffer policies

- Two policies for forwarding message
  - **broadcast-push**
    - Push to all successors that accept
  - **single-push**
    - Push to the 1st successors that accept
- Two policies when no successors accept
  - **Buffering**
  - **Discarding**
- Two policies for accepting messages
  - Accept
    - Accept all pushed messages
  - Switch
    - Do not accept, and switch to pull mechanisms

Node	Reception Policy	try_get()	try_reserve()	Forwarding
Functional Nodes				
source_node	—	yes	yes	broadcast-push
function_node<rejecting>	accept/switch	no	no	broadcast-push
function_node<queueing>	accept	no	no	broadcast-push
continue_node	accept	no	no	broadcast-push
multifunction_node<rejecting>	accept/switch	no	no	broadcast-push
multifunction_node<queueing>	accept	no	no	broadcast-push
Buffering Nodes				
buffer_node	accept	yes	yes	single-push
priority_queue_node	accept	yes	yes	single-push
queue_node	accept	yes	yes	single-push
sequencer_node	accept	yes	yes	single-push
overwrite_node	accept	yes	no	broadcast-push
write_once_node	accept once	yes	no	broadcast-push
Split/Join Nodes				
join_node<queueing>	accept	yes	no	broadcast-push
join_node<reserving>	switch	yes	no	broadcast-push
join_node<tag_matching>	accept	yes	no	broadcast-push
split_node	accept	no	no	broadcast-push
indexer_node	accept	no	no	broadcast-push
Other Nodes				
broadcast_node	accept	no	no	broadcast-push
limiter_node	accept/switch	no	no	broadcast-push

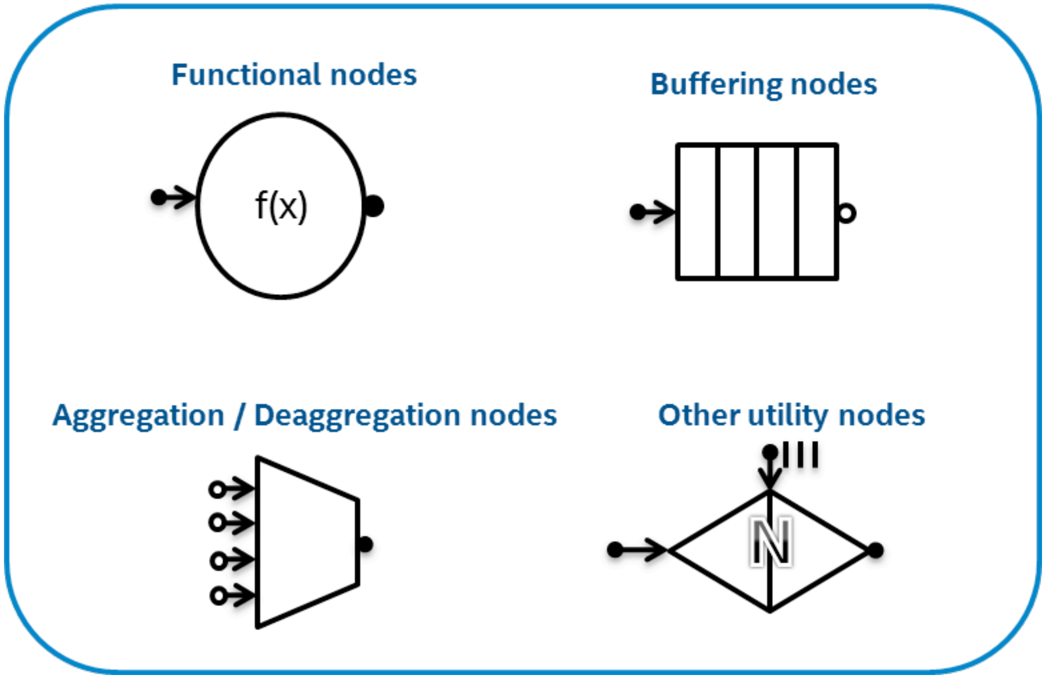
Examples of edge creation

Examples of node creation

- Creation of dependence nodes
- Creation of typed input/output nodes
- Examples of the different types of nodes

- Che differenza tra nodi broadcast e nodi con più dipendenze in uscita?
- L'esistenza della coda in input è implicita?

- aaaa



Functional nodes
f(x)
Buffering nodes
Aggregation / Deaggregation nodes
Other utility nodes
N

OpenCL nodes

Scheduler initialization

- Task_scheduler_init provides means for the user to customize the scheduler
  - When the scheduler is constructed/destroyed
  - How many worker threads the scheduler uses
  - The stack size of worker threads
- Either activated immediately on construction, or subsequently
  - Via ::deferred and and initialize()
- A task scheduler init affects all subsequently created schedulers
  - Also wrt floating point settings