

INTEL® ADVISOR FLOW GRAPH ANALYZER

User Guide



Revision History

Revision Number	Description	Revision Date
001	Initial release.	March 2014
002	Described simplified collector setup and trace conversion. Added Linux* OS support for collector. Described sample traces included in distribution.	May 2014
003	Described additional samples.	August 2014
004	Described GUI and workflow changes. Added section on GUI-based data collection.	August 2015
005	Updated to match name change. Updated copyright and legal information.	April 2016
006	Described new layouts and features.	April 2017
007	Updated to reflect new icons and features introduced for the Technology Preview.	July 2017
008	Changed title and modified fonts. Included new Release Notes.	October 2017
009	Added initial support for nested parallelism: Threading Building Blocks (TBB) parallel_for algorithms and OpenMP* parallel regions. Extended the Scalability Projection to include data flow graphs. Added nested parallelism to samples.	March 2018
010	Fixed bug related to loading of TraceML* file, where graph contains async node, nested within a composite node.	May 2018
011	Described GUI and workflow changes. Added section to Analyzer Workflow describing how to Reduce Scheduler Overhead using Lightweight Policy and how to Identify Task with Common Inputs.	July 2018
012	Added a new section on the experimental support for OpenMP* applications that details the data collection, modeling and visualization aspects of OpenMP* task dependencies.	October 2018
013	Added scaled projections and support for lightweight policy in scalability analysis. Improved performance of Treemap view and added different layouts for analysis view in preferences.	December 2018



Revision Number	Description	Revision Date
014	Added preferences settings to enable inter-graph edges. Updated collector scripts to support spaces in paths.	April 2019
015	Added preferences settings to select different hierarchical sort algorithm to use and to enable semi accurate edge placement and node placement for new hierarchical layout. Added new feature, Symbol resolution, that enables user to view nodes to source code mapping on FGA.	July 2019



Contents

About This Document	7
Setting up Flow Graph Analyzer	7
System Requirements	7
Software Supported for Analyzer	7
Software Supported for Collector	7
Installing Flow Graph Analyzer	8
Launching Flow Graph Analyzer	9
Using Flow Graph Collector	9
Flow Graph Analyzer GUI Overview	10
Menus	11
Toolbars	13
Tabs	15
Main Canvas	18
Charts	19
Designer Workflow	21
Adding Nodes and Edges	22
Modifying Node Properties	23
Viewing Edge Properties	25
Validating a Graph	25
Saving a Graph to a File	26
Generating C++ Stubs	27
Preferences	31
Scalability Analysis	33
Running an Analysis	33
Concurrency Specification	33
Data Count	35
Weight	35
Activating the Graph	36
Exploring the Parallelism in a Concurrent Node	36
Showing Non-Parallel Nature of a Serial Node	37
Exploring the Parallelism Provided by the Topology of a Graph	37
Understanding Analysis Color Codes	38
Collecting Traces from Applications	39
Prerequisites	39
Simple Sample Application	39



Building the Application.....	40
Building on a Windows* Operating System	40
Using a Visual Studio* Command Prompt	40
Building from a Visual Studio* IDE	41
Building on a Linux* Operating System.....	41
Building on a Mac* Operating System	42
Collecting the Trace Files	42
Collecting Traces Inside the Flow Graph Analyzer GUI	43
Collecting Traces Outside the Flow Graph Analyzer GUI	44
Collecting Trace Files with an fgtrun Script.....	45
Collecting Trace Files without an fgtrun Script	46
Nested Parallelism in Flow Graph Analyzer	47
Tracing Nested TBB Algorithms.....	48
Tracing Nested OpenMP* Algorithms.....	48
Analyzer Workflow	48
Finding Time Regions of Low Concurrency and Their Cause	49
Finding the Critical Path.....	50
Finding Tasks with Very Small Durations.....	50
Reducing Scheduler Overhead using Lightweight Policy	52
Identifying Tasks that Operate on Common Input	54
Experimental Support for OpenMP* Applications.....	56
Collecting Traces for OpenMP* Applications	58
OpenMP* Constructs in the Per-Thread Task View.....	59
OpenMP* Constructs in the Graph Canvas	60
Parallel Regions	60
OpenMP* Tasks.....	60
OpenMP* Task Dependencies.....	61
OpenMP* Nodes to Source Code Mapping:	63
Sample Trace Files	65
code_generation Samples	66
Dining Philosophers.....	66
Feature Detection	67
performance_analysis Samples	68
Forward Substitution with Trace.....	68
Feature Detection with Trace.....	69
Computer Vision with Trace.....	70
Release Notes and Known Issues	72
August, 2019	72
October, 2018	72



August, 2018	72
May, 2018	72
October, 2017	73
September, 2017	73
Additional Resources	73
<i>Legal Information</i>	74



About This Document

This document explains how to use Intel® Advisor – Flow Graph Analyzer (FGA), a graphical tool for the construction, analysis, and visualization of applications that use the Threading Building Blocks (TBB) flow graph interfaces. Through a graphical interface, the Flow Graph Analyzer lets you start with a blank canvas and construct a flow graph application by interactively adding nodes and edges. It also lets you collect events during the execution of an existing application; these events allow you to explore the topology and performance of the flow graphs used by that application.

Setting up Flow Graph Analyzer

System Requirements

For general operations with the user interface and all data collection:

- A system based on IA-32 or Intel® 64 architecture supporting Intel® SIMD Streaming Extensions 2 (Intel® SSE2) instructions and Intel® HD Graphics 4000 or higher
- At least 4 GB of RAM with 8 GB recommended
- 200 MB free disk space to support data collection from flow graph applications

Software Supported for Analyzer

Tested operating systems:

- Microsoft Windows* 8.1 and 10
- Microsoft Windows Server* 2012 and 2016
- Debian 9
- Ubuntu 16.04 LTS and 18.04 LTS
- SUSE* Linux Enterprise Server* 12 SP3 and 15
- MacOS 10.12, 10.13 and 10.14
- Red Hat Enterprise Linux 7
- Centos 7
- Fedora 28, 29

NOTE: On Windows* operating systems, the Flow Graph Analyzer requires Microsoft Visual Studio* 2015 runtime components to operate correctly. You can download it from this URL:

<https://www.microsoft.com/en-us/download/details.aspx?id=48145>

Software Supported for Collector

Tested operating systems:

- Microsoft Windows* 8.1 and 10
- Microsoft Windows Server* 2012 and 2016
- Ubuntu* 14.04 LTS and Ubuntu* 16.04 LTS



- Red Hat* Enterprise Linux* 7
- Centos* 7
- Fedora* 28, 29
- SUSE* Linux Enterprise Server* 12
- MacOS 10.12, 10.13 and 10.14

Supported IDEs and compilers:

- Microsoft Visual Studio* 2015 and 2017
- GNU gcc* Compiler
- Clang compiler
- Intel® C++ Compiler (ICC) 15.0, 16.0, 17.0, 18.0 and 19.0

NOTE: On Windows* operating systems, the Flow Graph Collector requires at least Visual Studio* 2015 runtime components to operate correctly. You can download it from these URLs:

- <https://www.microsoft.com/en-us/download/details.aspx?id=48145> (Visual Studio* 2015)
- <https://visualstudio.microsoft.com/downloads/> (Visual Studio* 2017)

NOTE: On Linux* OS-based operating systems, the Flow Graph Collector requires support for GLIBCXX_3.4.19.

Installing Flow Graph Analyzer

The Flow Graph Analyzer package consist of a Flow Graph Analyzer tool and an associated collector to capture traces from Threading Building Blocks (TBB) flow graph and OpenMP* applications.

This package is part of Intel® Advisor and is installed in the following directory when the Intel® Advisor is installed: <advisor-install-dir>/fga

Under this directory, you can find the Flow Graph Analyzer tool, the collector, and supporting documentation.

- Flow Graph Analyzer:
`<advisor-install-dir>/fga/fga`
- Flow Graph Collector:
`<advisor-install-dir>/fga/fgt`
- Documentation:
`<advisor-install-dir>/fga/doc`
- Samples:
`<advisor-install-dir>/fga/samples`
- License:
`<advisor-install-dir>/fga/EULA.pdf`

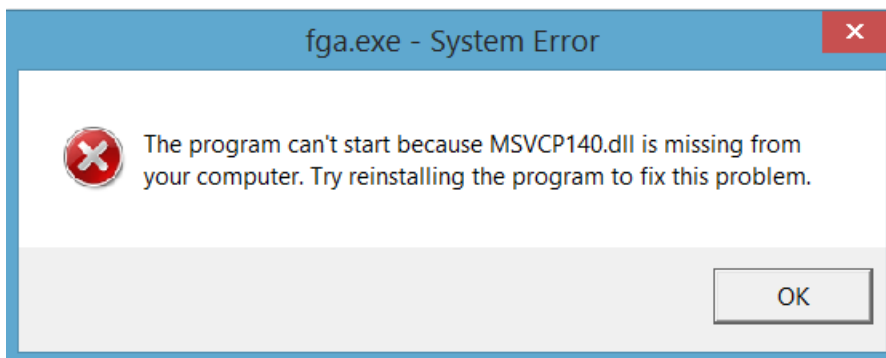


Launching Flow Graph Analyzer

The Flow Graph Analyzer executable and all its supporting files are located in the directory `<advisor-install-dir>/fga/fga` and you can launch it by double-clicking `fga.exe` on Windows* platforms, or `fga` on Linux* platforms. Moving the executable to a different location may cause the application to fail. For Linux* platforms, your `LD_LIBRARY_PATH` must contain `“.”`. Alternatively, on Linux* platforms you can invoke `run_fga.sh` from the command line, which sets the `LD_LIBRARY_PATH` and then starts the executable.

On a MacOS operating system, you can also invoke `run_fga.sh` from the command line to start FGA. Alternatively, you can add `“.”` to `DYLD_LIBRARY_PATH` and also `“./Frameworks”` to `DYLD_FRAMEWORK_PATH`, then invoke the `fga` executable directly.

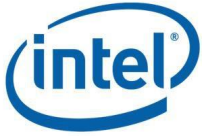
On Windows* systems without the Visual Studio* 2015 runtime components, you encounter an error while launching the Flow Graph Analyzer application.



Solution: download the Visual Studio* 2015 runtime components and install them before running the application. See the [Software Requirements for Analyzer](#) section for the location to download and install the runtime components.

Using Flow Graph Collector

The Flow Graph Collector allows you to capture the topology and the execution trace information of a running flow graph application. The Flow Graph Analyzer can then load this captured data for closer inspection of the graph described by the application and the performance data for the graph. The [Scalability Analysis](#) section provides the steps to build your application so traces are enabled and to capture the trace information. The [Collecting Traces from Applications](#) section discusses how to use the Flow Graph Analyzer tool for analyzing the performance of flow graph applications after a trace has been collected.

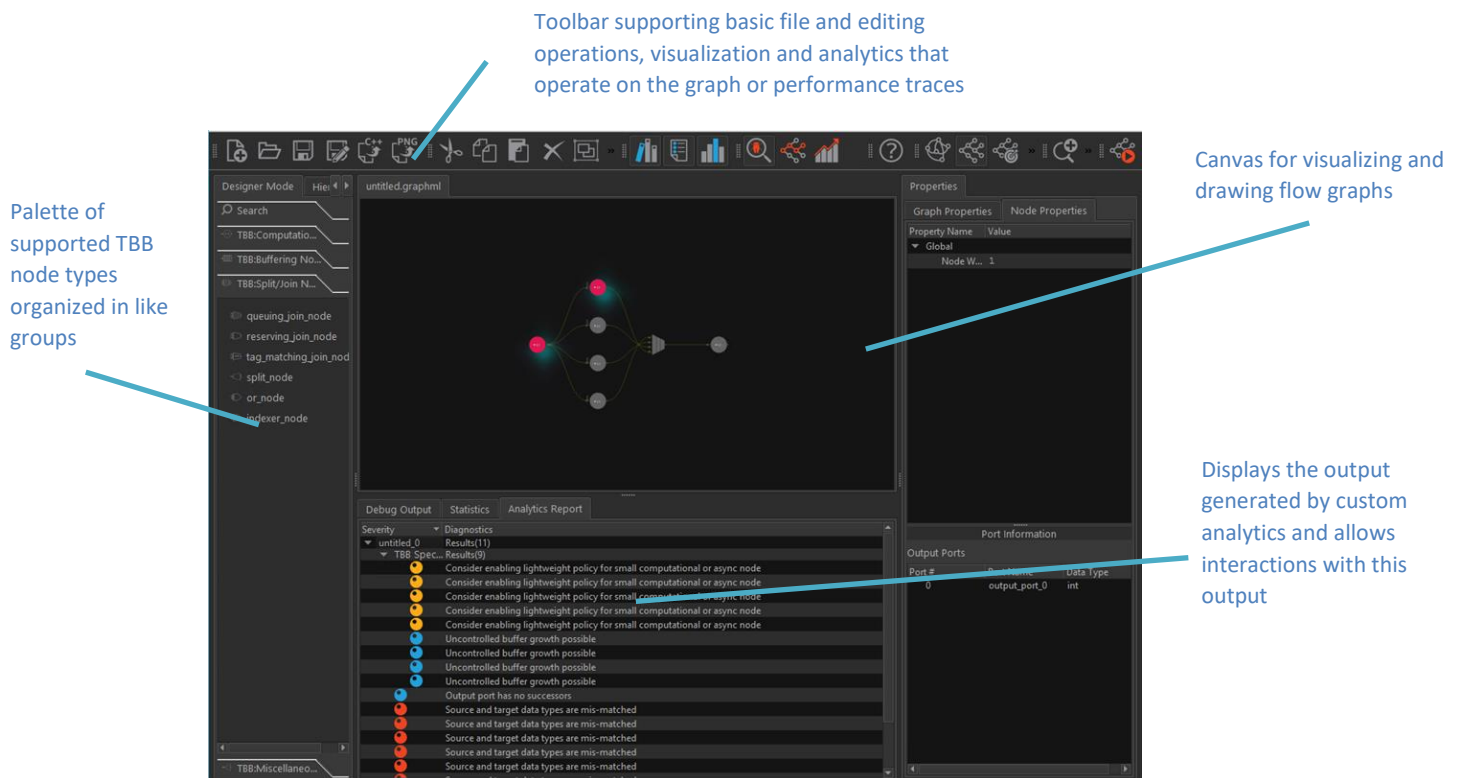


Flow Graph Analyzer GUI Overview

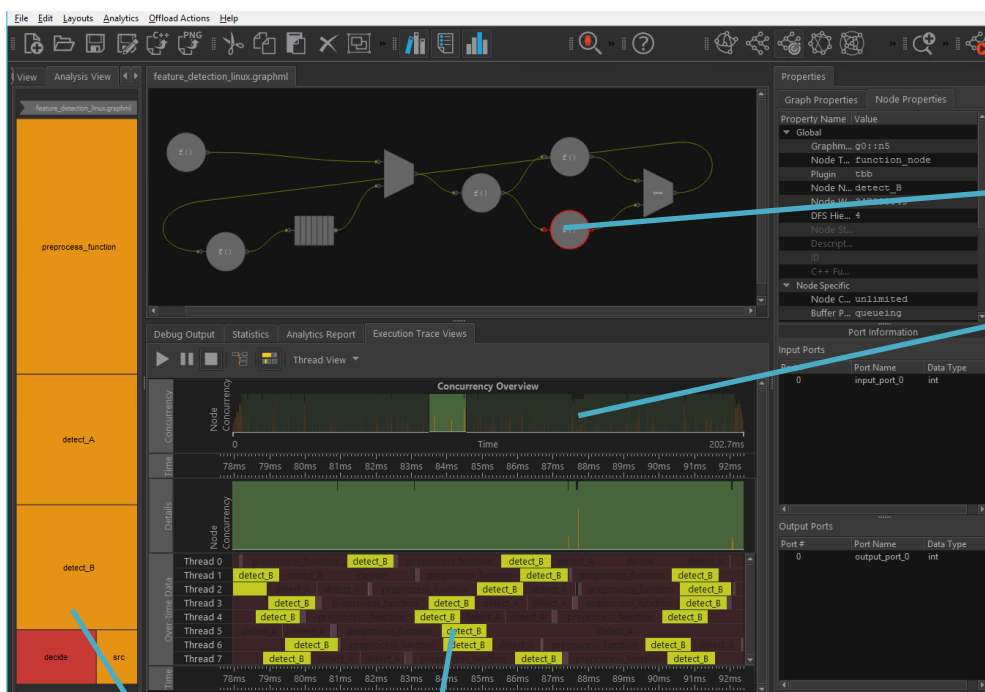
This section describes the Flow Graph Analyzer tool and the various features it offers to speed up the development of new flow graph applications. There is a component that enhances productivity during development and another component that supports performance-tuning tasks, and these are represented by two workflows – the designer and the analyzer. A brief introduction to the GUI layout is provided here and the two workflows enabled by the tool are described in subsequent chapters.

The Flow Graph Analyzer GUI provides the means to create new graphs visually and to load previously created or application-generated graphs.

The following figure describes the basic GUI layout and the key elements necessary for constructing graphs visually. These graphs can be saved for later use and, as described in the [Generating C++ Stubs](#) section, used to generate C++ framework code.



When analyzing an existing application's performance, the graph topology and performance data are captured from a running application and saved for post-mortem analysis. If performance traces are available when the graph is loaded, they are displayed in a timeline window below the canvas area. You can interact with the trace data in many ways, from cursory inspection of the trace data to detailed inspection of specific tasks and the nodes they map to. The following figure shows the timeline charts created when trace data is available.



Selection on the timeline highlights the nodes 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.

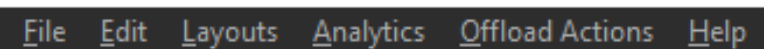
Treemap view provides the general health of the graph's performance, along with the ability to dive to the node level.

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

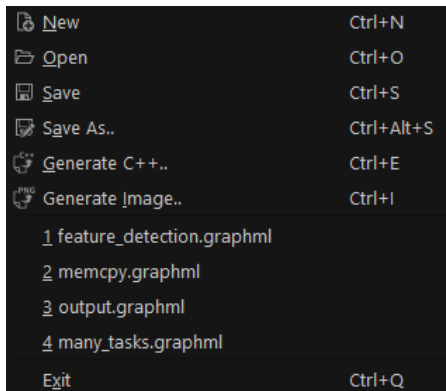
The various components referenced above are listed below with a brief description of what they help you accomplish:

Menus

The menus in the menu bar have some fixed components such as *File*, *Edit* and *Help*, and dynamic components such as *Layouts*, *Analytics*, and *Trace Data Collection*. The fixed components are always available and the dynamic components may change depending on the plugins registered with the tool.

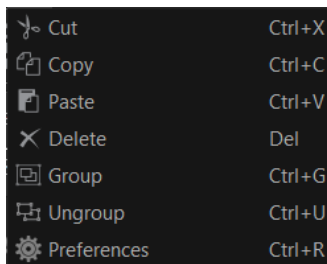


- **File Menu:** Allows you to create a new graph, load an existing graph, or save the current graph on the canvas to a GraphML* file or export it as C++ source files.

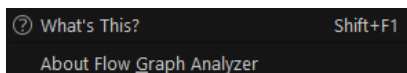


In addition to these options, the menu also maintains a list of recently used files for quick access. Print support is currently unavailable; the *Generate Image* option enables printing the graph displayed in the canvas as a PNG file.

- **Edit Menu:** Allows you to edit the graph displayed on the canvas and supports common edit actions, such as *Cut*, *Copy*, *Paste*, *Delete*, *Group*, *Ungroup* and *Preferences*. These actions support the common keyboard shortcuts.



- **Help Menu:** Help information for various GUI elements is provided through the *What's This?* paradigm, and this menu allows you to get into the *What's This?* mode. In this mode, you can click any GUI element that has supporting help information to view more information about the element and what it helps you accomplish.



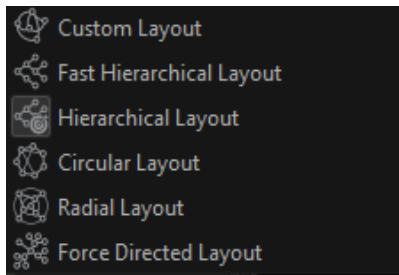
You can also get into the *What's This?* mode using the keyboard shortcut Shift+F1.

- **Layout Menu:** This menu helps you visualize the graph on the canvas in many different ways. Currently supported layout types are *Hierarchical*, *Radial*, *Force-Directed*, *New Hierarchical*, and *Circular*. For most graphs, the *Hierarchical Layout* is sufficient and is set as the default layout when any graph is loaded for display. If the *Hierarchical Layout* does not work properly for a graph model, the *New Hierarchical Layout* can be used.

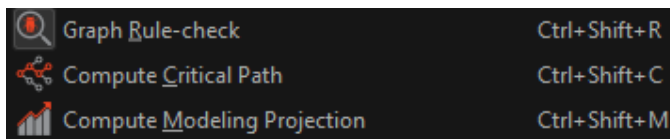
NOTE: The *New Hierarchical Layout* is 3x slower than the default *Hierarchical Layout*.



If you cannot get a visually pleasing layout using the hierarchical layouts, use the *Radial*, *Circular*, or *Force-Directed* layouts. The *Circular Layout* and *Force-Directed Layout* use the Boost* Graph library. The cost of the running the *Force-Directed Layout* is high compared to other layouts, but it provides better graph layout visuals.



- **Analytics Menu:** This menu is populated by algorithms that are present in available plugins, and this menu changes as new plugins are added.



These analytical algorithms are available for Threading Building Blocks (TBB) flow graphs and more algorithms may be added in the future.

Compute Critical Path computes one or more critical paths for a graph and lists them in the *Analytics Report* tab. You can interact with these critical paths to see which nodes are part of them.

Graph Rule-check performs some basic rule checks on the graph that highlight potential performance and correctness problems.

Compute Modeling Projection projects the speedup of a graph with varying numbers of threads. The speedup with the corresponding number of threads is shown in the *Analytics Reports* window, while a chart showing the ideal versus actual speedup is shown in the chart area. You can set the thread counts to be used for the projection in the *Concurrency list* entry of the *Graph Properties*. The format is *a-b:n*, where *a* is the start thread count, *b* is the end thread count, and *n* is the step size. Other accepted formats are *a,b,c ...* where *a*, *b*, and *c* are the defined thread counts. A single thread count is also acceptable.

Toolbars

The menu items are also exposed in the toolbar area as shortcuts to frequently used operations. Hovering the mouse over an icon on a toolbar provides a tooltip describing functionality.

- **File Toolbar:** Provides access to the functionality exposed by the *File* menu.



- **Edit Toolbar:** Provides access to the edit operations exposed by the *Edit* menu.



- **Window Toolbar:** Many GUI display elements are configurable and this toolbar exposes the show/hide toggle capability for a few tabs. From left to right, these icons represent the *Toolbox*, *Reports*, and *Charts* tabs.



The *Toolbox* tab, which contains the *Designer Mode*, *Analysis Mode*, and *Hierarchical View* tabs, can be hidden to increase available screen real estate for large graphs. The *Reports* tab, where information such as node properties or output of analytics algorithms is displayed, can also be hidden to get more screen real estate for visualizing large graphs. The *Charts* tab is always hidden when the graph does not have associated execution trace data. However, when the trace data is available, the *Charts* tab is displayed by default and this toggle switch allows you to hide or restore it.

- **Analytics Toolbar:** The actions exposed in this toolbar are a subset of the *Analytics* menu. The plugin has a choice to register the analytics algorithm with the toolbar or not, but is always be registered with the *Analytics* menu.



The analytics supported for TBB flow graphs are, from left to right in the toolbar, *Compute Critical Path*, *Graph Rule-Check*, and *Compute Modeling Projection*. These algorithms may be used for designing new graphs and tuning existing graphs.

- **Layout Toolbar:** This toolbar exposes the layouts that are most frequently used and may be a subset of the layouts present in the *Layout* menu.



- **Zoom Toolbar:** The canvas area that displays the graph allows you to zoom in or out, and this toolbar supports the zoom capabilities. You can also zoom in or out using the mouse-wheel when the mouse is in the canvas area.



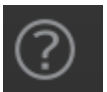
The reset-zoom button resets the zoom factor so the entire graph is visible in the canvas area.



- **Trace Data Collector Toolbar:** This invokes a dialog box that helps you configure a data collection run on a TBB flow graph application. This dialog box also allows you to configure the environment before launching the application.

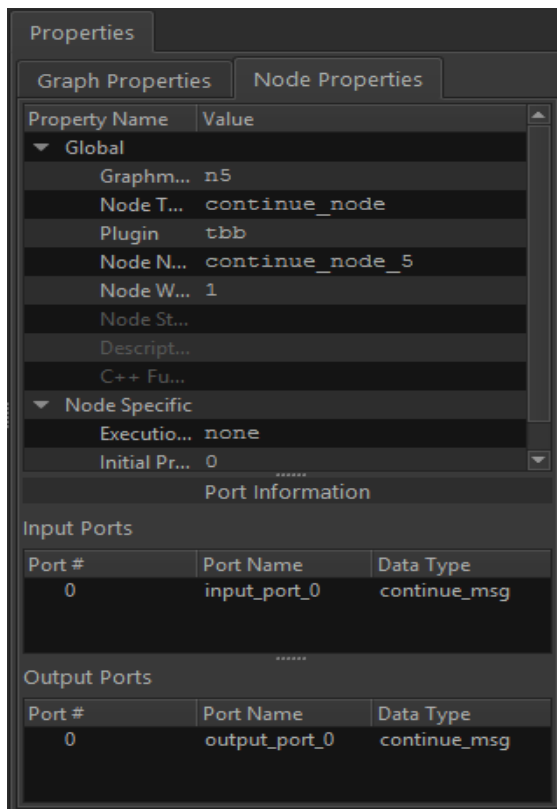


- **What's This? Toolbar:** This is a single button that puts you in the *What's This?* mode. You can also invoke this option by using the Shift+F1 keyboard shortcut.



Tabs

- **Properties Tab:** This area is primarily used to display information about a selected graph in the *Graph Properties* tab or a node in the *Node Properties* tab.

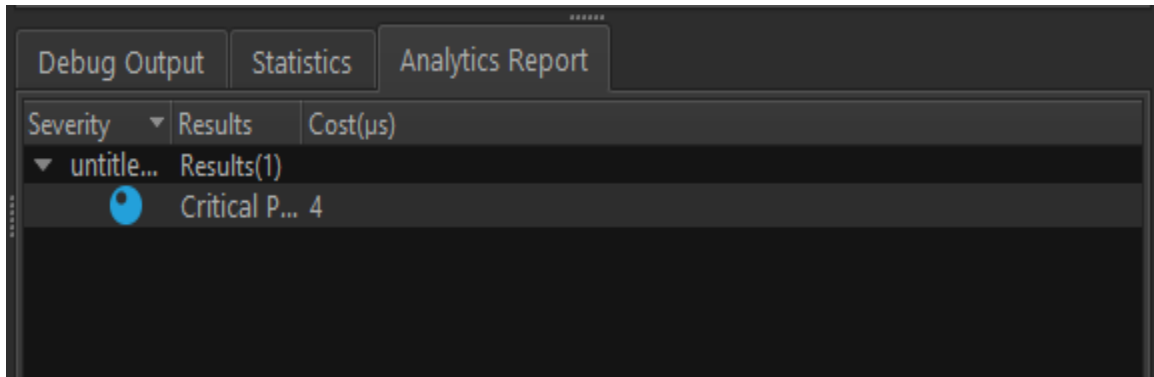


The *Node Properties* tab displays all of the properties supported for a given node type and allows interactive editing of these properties. However, some of these properties may be



ted to the node type and are marked as read-only. See the [Designer Workflow](#) section for instructions on editing the properties of a node.

- **Analytics Report tab** displays the output generated by any invoked analytic algorithms. Because the output generated by each analytic algorithm could be different, this view might differ when different algorithms are run on the graph. The screenshot below shows a sample output for *Compute Critical Path*. The columns in this tab are sortable and enable efficient data manipulation during the performance tuning workflow.

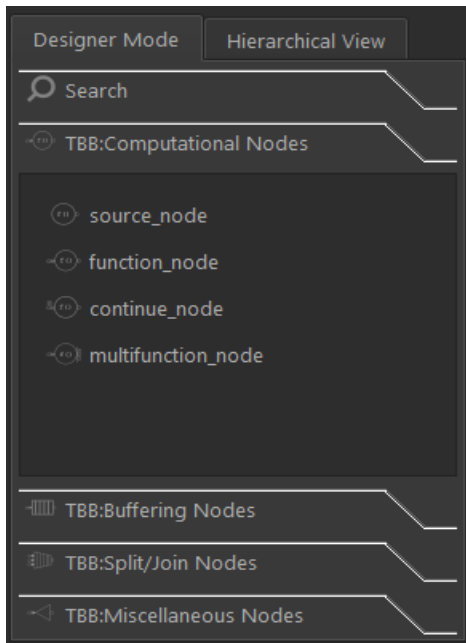


Severity	Results	Cost(μs)
▼ untitled...	Results(1)	
●	Critical P...	4

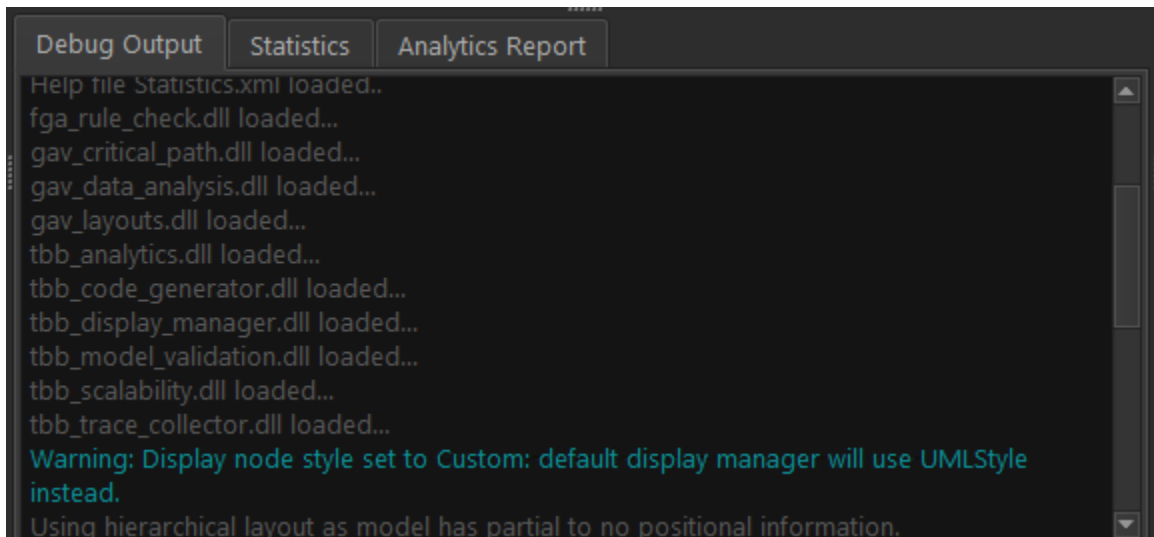
- **Designer Mode Tab:** Flow Graph Analyzer exposes the available node types for use during the construction of a graph in a menu area on the left side of the tool. This tab may change when new nodes are added to the TBB flow graph interface. To get more information about each node type, use the *What's This?* functionality on the node-type buttons.

After a node type is selected for insertion into the graph, the mouse cursor assumes the shape of the selected node type. You may add as many nodes of the same type as you want to by placing the cursor at a location on the canvas and clicking the mouse. To switch to a different node type, select the node type of interest in the tab.

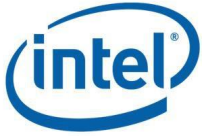
When you are done adding all nodes, you can exit the *InsertNode* mode by pressing the ESC key if the mouse is placed in the canvas area, or by selecting the *InsertEdge* or *MoveNode* modes from the *Toolbar* menu. To return to the *InsertNode* mode, just select a node type from the *Designer Mode* tab.



- **Debug Output Tab:** This tab displays messages output by the tool. Most of the messages are informational, but warning or error messages may appear. Warning messages are usually in green and Error messages are usually in yellow.

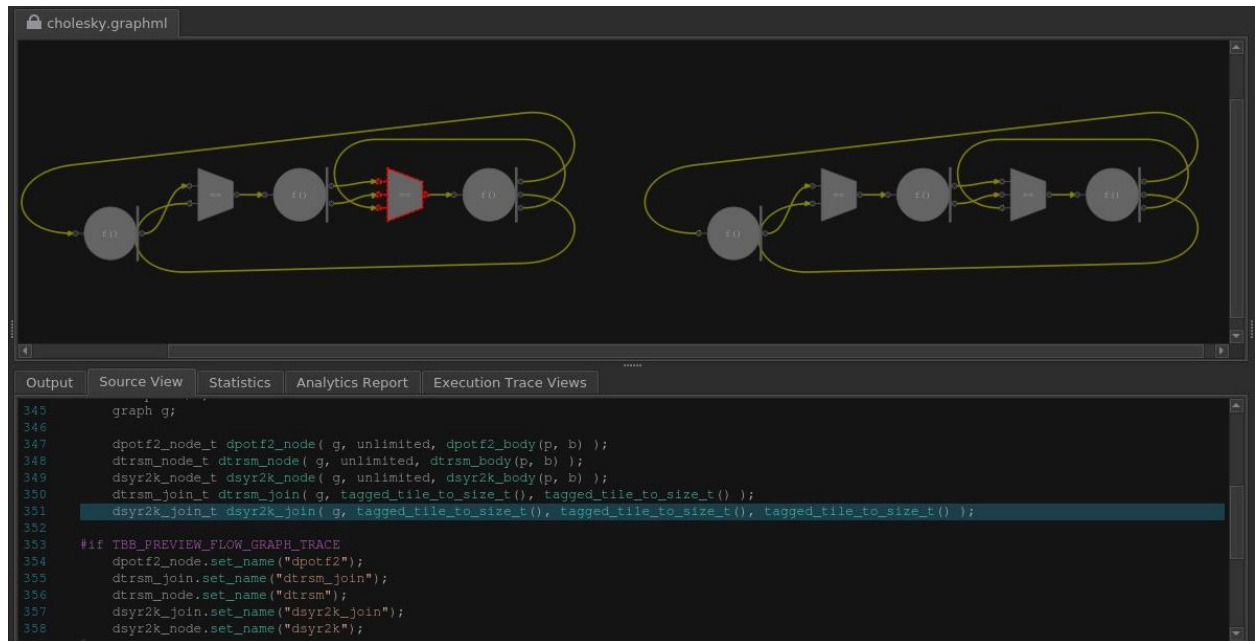


The preceding screenshot shows a partial capture of sample debug output.



- **Source View Tab:** This tab displays the source mapping of a selected node if symbol resolution information has been captured during the collection.

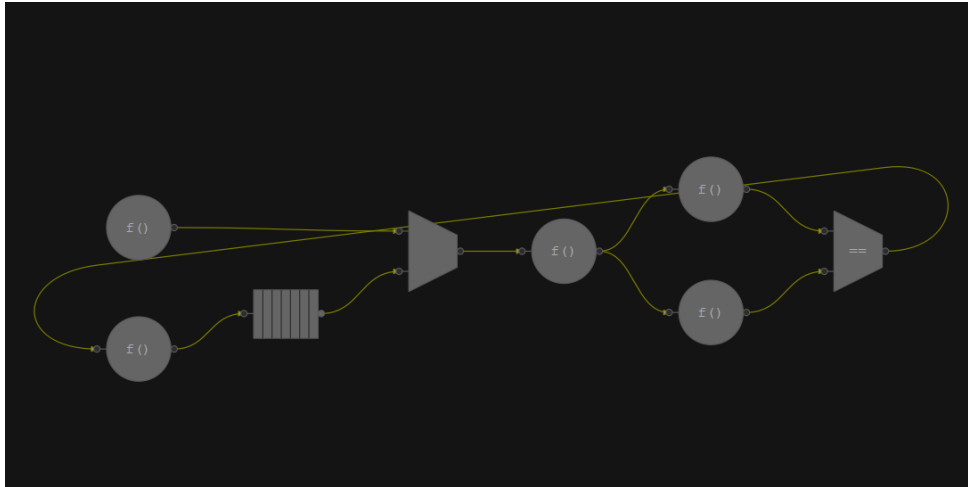
NOTE: Currently this feature is only supported on Linux. In order to enable data collection that includes symbol resolution information, please look at [Building on a Linux* Operating System](#) and [Collecting Trace Files with an fgtrun.sh Script](#).



The preceding screenshot shows source code mapping with nodes in cholesky example that ships with Threading Building Blocks samples.

Main Canvas

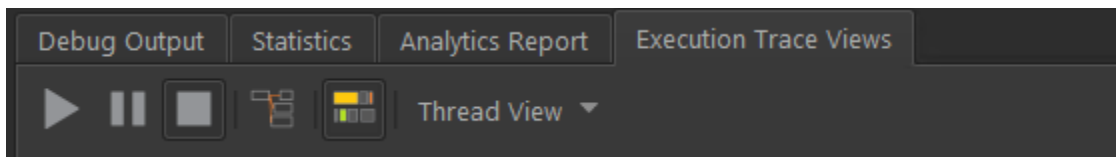
This is where the graphs that are created or loaded are displayed. This area supports zoom capabilities through the use of the mouse-wheel. All node objects displayed in this area enable the context menu for node-specific menu options.



Nodes in this area can be selected and moved when in the *MoveNode* mode and new edges can be inserted between two nodes when in the *InsertEdge* mode.

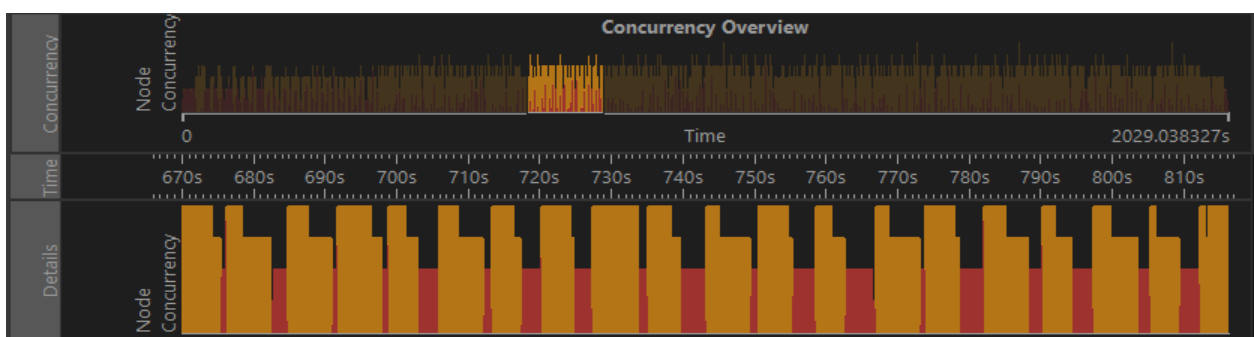
Charts

This area displays the task execution trace data available for a graph. All charts in this area can be inspected at various resolutions with the help of zoom functionality. This is enabled via the mouse-wheel actions over the chart or through the use of zoom-related buttons in the toolbar above the charts.

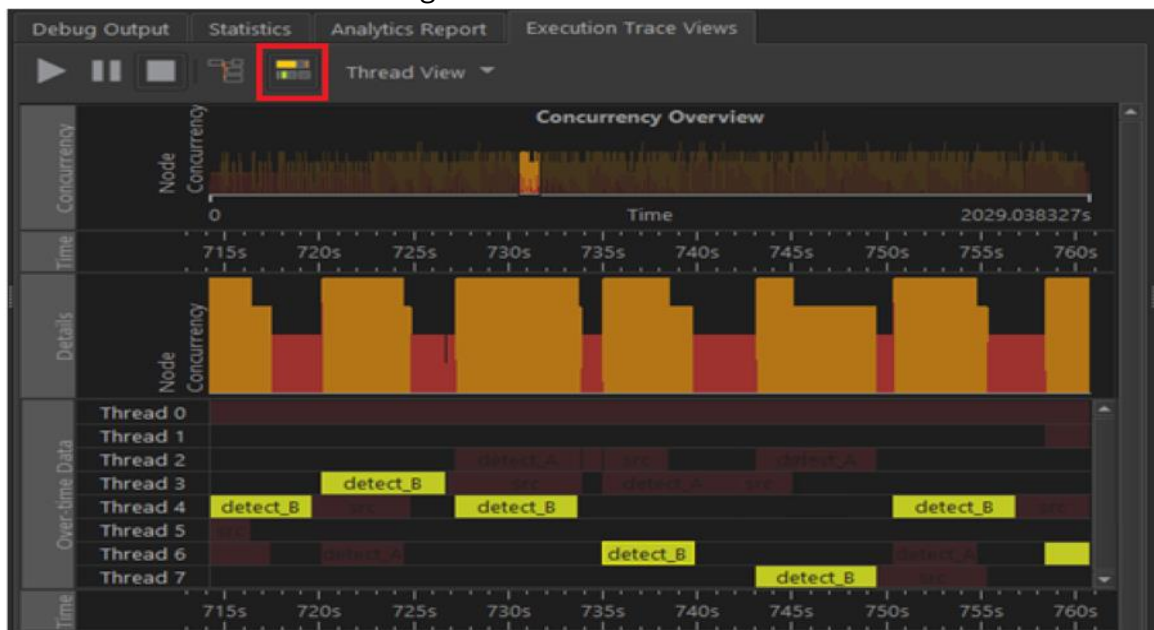


The execution traces are displayed in two different forms:

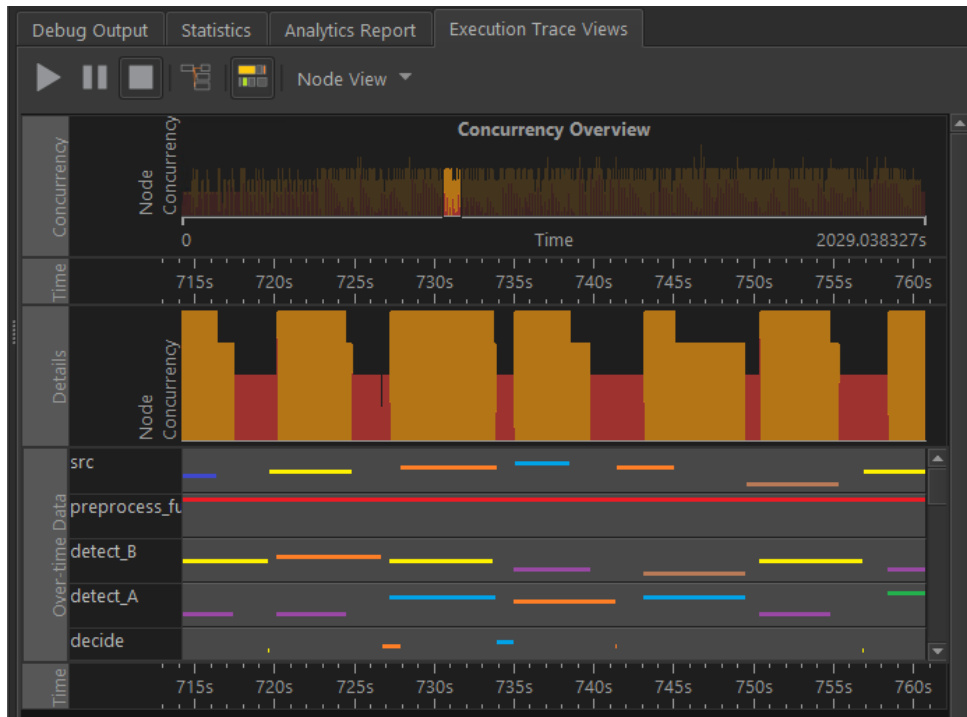
- Node concurrency display:** In this form, there are two charts. The top chart shows the concurrency display for the entire length of the application run, and the bottom chart that shows the details of the zoomed region in the upper chart. In both charts, the node concurrency is plotted over time. The maximum node concurrency is limited to the maximum number of threads in the TBB thread pool.



- **Per-thread task display:** In this chart, you can see the tasks executed by each thread and their duration. You can also see tasks associated with particular node, by enabling “Show/Hide Selected tasks” button, highlighted in red in the figure below. The drop-down box in the toolbar of the *Execution Trace Views* tab, the chart display can be toggled between a *Thread View* and a *Node View*. You can zoom in or out of the data in both views with the help of the specific buttons in the chart toolbar or through the use of a mouse-wheel. In some cases, the trace data can contain additional information about the logical core on which a task executes and the data ID it is processing with the help of user-APIs support by TBB and the Flow Graph Analyzer. When this information is available, you can visualize the *Thread View* data and color them by core information or by the data it is processing.
- In the *Thread View*, the y-axis is the set of threads that participated in executing the flow graph and the x-axis is time. Tasks with short durations are displayed with a lighter color than those with a longer duration. The lighter color highlights tasks that are small relative to the cost of scheduling the task.



- In the *Node View*, a set of thread timelines is created for each node in the graph. In each set, the y-axis is the set of threads that participated in executing the flow graph and the x-axis is time. In the *Node View*, a node's set of timelines only displays tasks related to that node, while in the *Thread View*, a single set of timelines shows the tasks related to all nodes.



Designer Workflow

Designing flow graph applications using the Threading Building Blocks (TBB) library involves understanding the various node types supported, how to map them to end-user concepts or computational entities, and linking them all together to form the flow graph. However, the human brain has trouble visualizing and mapping such computational blocks when the count of such blocks goes beyond a handful.

To help solve this visualization problem, the Flow Graph Analyzer tool supports two workflows:

- **Designer workflow** – Enables expressing these relationships using TBB flow graph node types.
- **Analyzer workflow** – Enables capturing and viewing graph topologies and related performance data captured from executed applications.

The Flow Graph Analyzer designer workflow allows you to describe your graphs visually, set some meaningful properties, and let the tool generate the flow graph framework code in C++. Before generating the framework code, you can also perform rule-checks to make sure the described graph does not have any potential issues that could lead to incorrect execution or a poorly performing graph. The generated code can be compiled and run without modification in many cases. Sometimes, the generated code may have to be modified to provide meaningful inputs or outputs.

Specifically, the tool supports the following capabilities necessary for visual design of graphs:

- Choose from a variety of available node types to build the graph.
- Express the explicit relationships with edges.

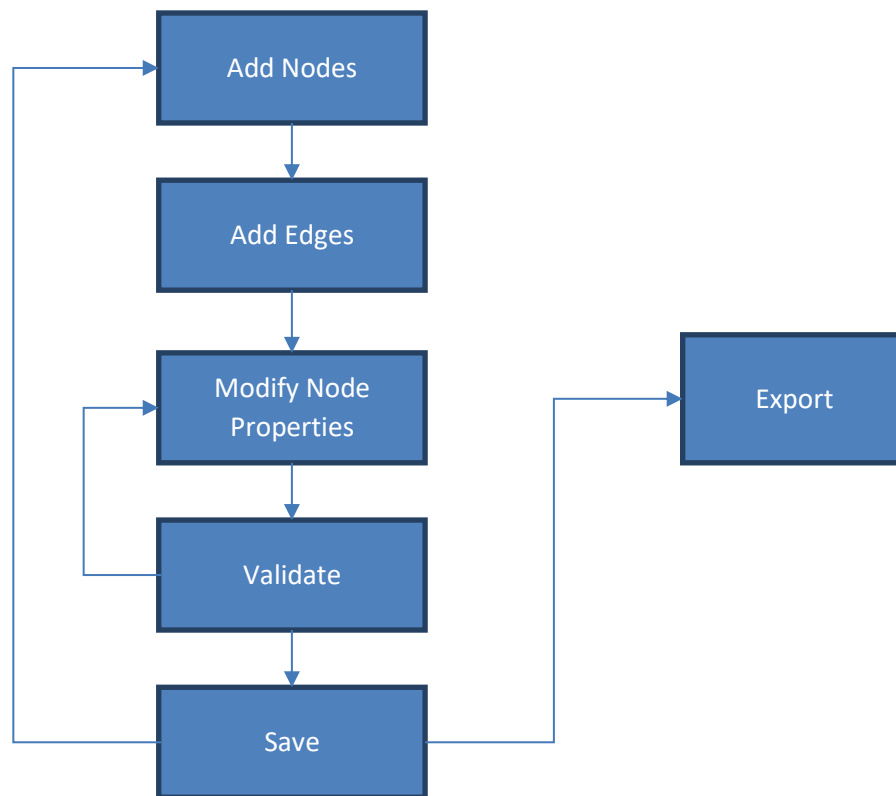


- Edit the properties of these nodes.
- Perform common editing operations such as Cut, Copy, Paste, Delete, Undo, Redo, Group, and Ungroup.
- Save the described graph and reload it at a later time.

In addition to these basic capabilities, the tool provides the means to:

- Validate each node to ensure that flow graph rules are not broken.
- Perform basic rule-checks on the graph to highlight potential performance problems.
- Export the graph as C++ framework code that uses the TBB flow graph API.
- Export the graph as PNG image.

This section steps through the design workflow and the capabilities that support it. The following figure shows the simple flow of the design process using the tool as a designer.



Adding Nodes and Edges

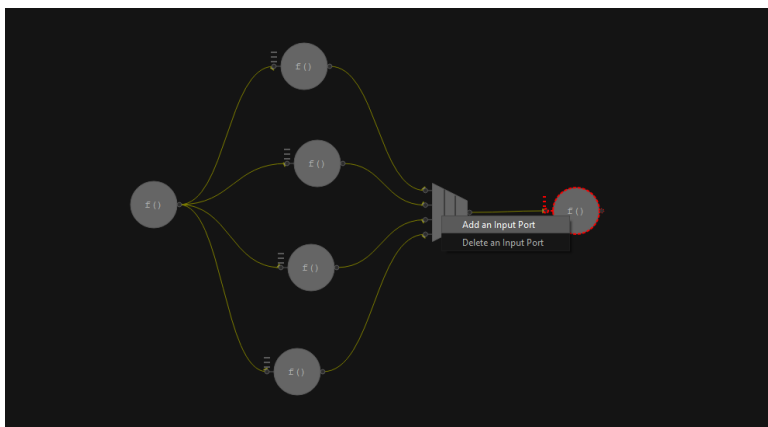
When the tool starts up, you are presented with a blank canvas to which you can add nodes from the list of nodes under Designer Mode pane on the left side of the tool. Drag the required nodes to the canvas. Then add the dependencies (*edges*) between them by clicking the output port of a node and dragging to the input port of another node.

Modifying Node Properties

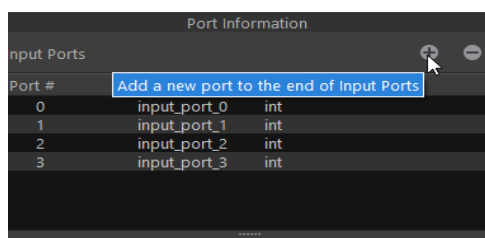
After you add the nodes to the graph and connect them with edges, inspect the nodes to ensure the data flowing through the graph is correctly typed. Some data types are dictated by the Threading Building Blocks (TBB) flow graph node types themselves or by the logic the graph represents. Because the data flows through nodes and edges are connected to ports, the data types are managed at the port level. The default node data types are of type `int` for most ports and `continue_msg` for nodes that expect this type of data.

Edit the node properties using the steps outlined below:

1. Select the node. The figure below shows the node menu items for a `join_node`, which can take multiple inputs, right-click the node to display a context menu that allows you to add new ports or delete existing ports. The addition and deletion of ports happens in order. When a new port is added, it is the highest-numbered port for the node. When a port is deleted, the highest-numbered port is removed.



You also can highlight a node and add or remove ports from the *Node Properties* tab in the Reports area:



2. In the *Port Information* section of the *Node Properties* tab, change the *Data Type* of a port by selecting its type and editing the field.



Input Ports		
Port #	Port Name	Data Type
0	input_port_0	int
1	input_port_1	int
2	input_port_2	int
3	input_port_3	float

For certain nodes, such as a join_node, only the input port data types can be modified and the output data type is automatically generated when the input port data types are updated.

Port Information		
Input Ports		
Port #	Port Name	Data Type
0	input_port_0	int
1	input_port_1	int
2	input_port_2	int
3	input_port_3	float
Output Ports		
Port #	Port Name	Data Type
0	output_port_0	tbb::flow::tuple<int,int,int,float>

3. Selecting a node on the canvas updates the *Node Properties* tab. The node properties for the join-node discussed in the previous bullet are shown below. This property pane displays all the properties that can be edited for a given node type. The properties that are not currently set for a given node type are shown in a different color. In the figure below, you can see that the *Description* property is not set for the join-node.

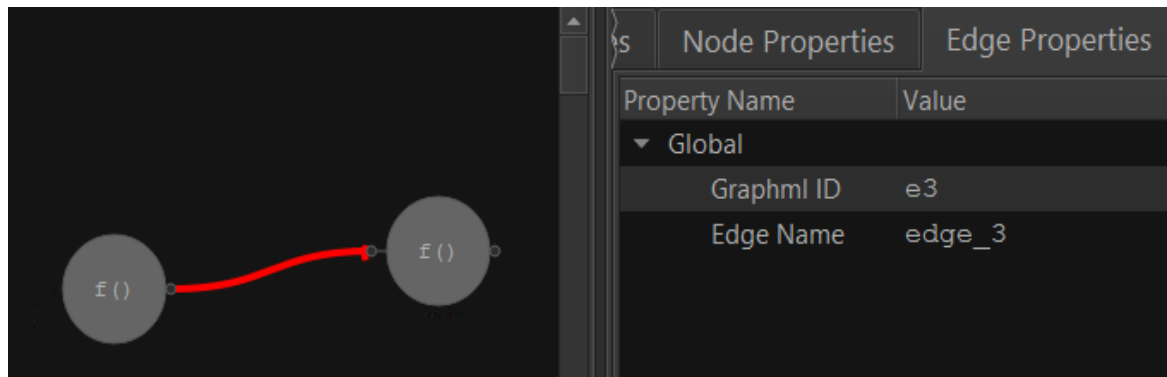
Properties	
Graph Properties Node Properties	
Property Name	Value
▼ Global	
Graphml ID	n13
Node Type	join_node
Plugin	tbb
Node Name	join_node_13
Node Weight (µs)	0
Node Status	
Description	
C++ Function Object	
▼ Node Specific	
Buffer Policy	queueing

Some of the properties for the nodes are automatically set and are tied to the node type. Such properties are not available for editing and the *Node Properties* tab enforces these rules. For example, you cannot change the *Node Type* property for a node, but you can edit the *Node Weight* or the *Node Name*.

- The *Node Weight* is provided as a placeholder to indicate the computational complexity of the node. The larger the number, the more computationally intensive the node is with respect to the other nodes in the graph. This number is also used by the C++ code generator to create a busy loop in the empty body that is created for each node. See the [Generating C++ Stubs](#) section for more details.
- The *Node Name* initially assigned to a node is a unique name you can change to something meaningful. This name is the variable name of the object generated for the node by the C++ code generator.


Viewing Edge Properties

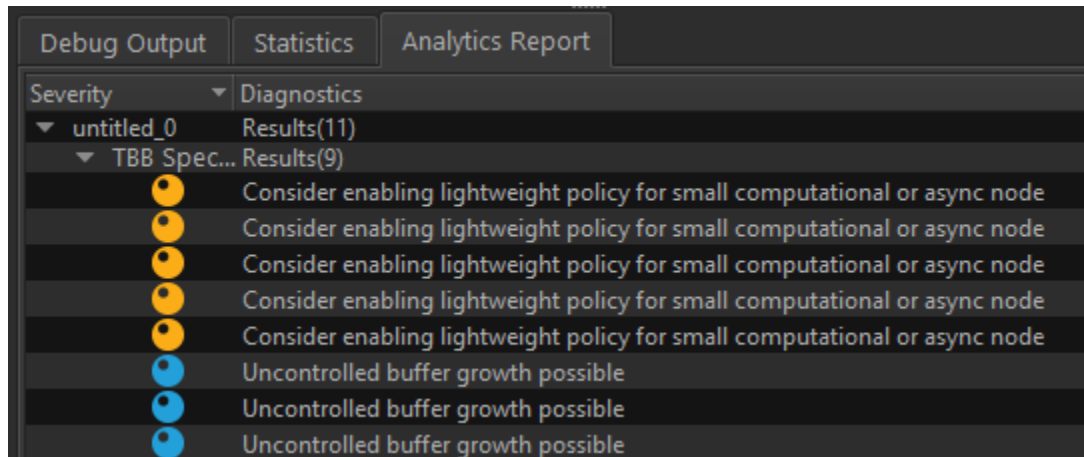
The properties of an edge can be viewed by selecting the desired edge on the canvas. This populates the properties of the edge in the *Edge Properties* tab as shown below.



Validating a Graph

After all nodes and edges are created and their properties are set, perform a validation step of the entire graph to point out data type mismatches between source and target nodes and highlight other possible issues that may manifest within the described graph topology.

To invoke a sequence of rule-checks on the graph that tests various aspects of the graph construction, click the *Graph Rule-Check Analytics* icon () on the toolbar. The results are reported in the *Analytics Report* tab in the *Reports* tab. The following figure shows sample output for a graph.



Clicking a reported diagnostic highlights the node that needs to be inspected again. In the case of data mismatches, both the source and target nodes for a given edge are highlighted. Review the *Node Properties* tab to address any changes that are needed.

Saving a Graph to a File

After the graph is in a consistent state and all of the major issues are addressed, save it to a GraphML* file. GraphML* format is an open-standard file format for representing graphs and is the chosen format for representing graphs in the tool.

You can return to a graph at a later time to modify its topology and data types.

See the [Generating C++ Stubs](#) section for details on how to save the graph to C++ code form and what is actually generated by the code generator.

You can also save a graph in the canvas to a PNG image by clicking the *Generate Image* icon on the toolbar. The *Generate Image* icon exports the image to the same directory where the corresponding GraphML* file is saved.

When you load a graphml file for the first time, the tool computes many pieces of information such as derived performance metrics and graph layout positions. This derived information computation can be expensive and it is recommended that you save the graph after this first load. This derived information is saved in a METAXML file and provides caching benefits that enable the tool to exhibit better performance on subsequent loads of the same graphml file.

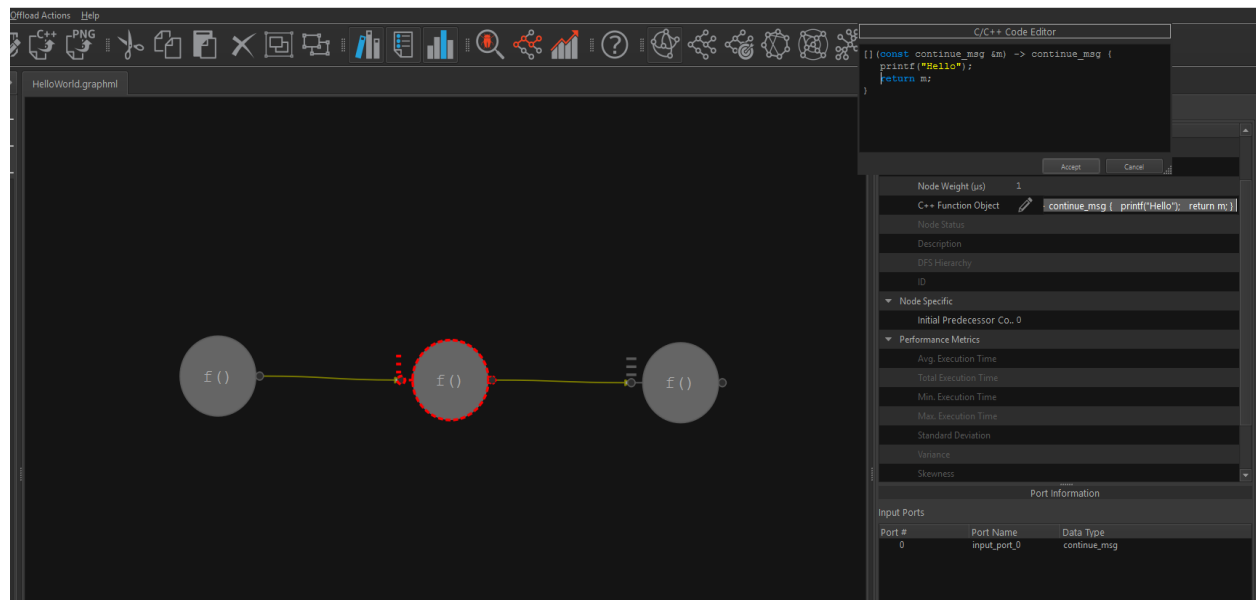
NOTE: You may be asked, if you wish to save the graphml files after load even if no interaction or modification took place. This is due to the tool running the layout algorithm on the graph to display it in an intuitive manner.

Generating C++ Stubs

To generate stubs for a working C++ application:

1. Use the Flow Graph Analyzer to draw the graph in the canvas and save the graph as described in the [Designer Workflow](#) section.
2. Click the *Generate C++* icon on the toolbar to create C++ files.

For example, below is a three-node graph you can use to create a Hello World sample. This graph consists of a source_node followed by two continue_node objects. The first node is named s0 and the next two nodes are named c0 and c1. All nodes have continue_msg objects as their input and/or output types. In addition to drawing the graph, the body of each node is defined by its *C++ Function Object* field, as shown below. The important properties set for each node in this sample are summarized in the following table.

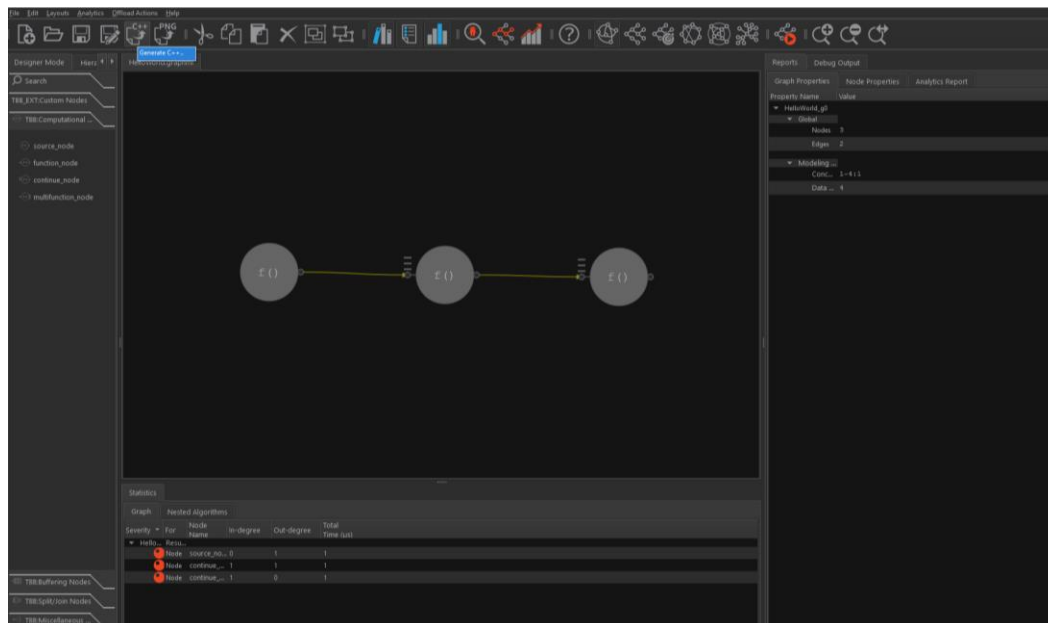


Node Name	Node Type	Input Type	Port	Output Type	Port	C++ Function Object
s0	source_node	None		continue_msg		<pre> [] (continue_msg &c) -> bool { static bool done = false; if (!done) { done = true; return true; } else { return false; } } </pre>

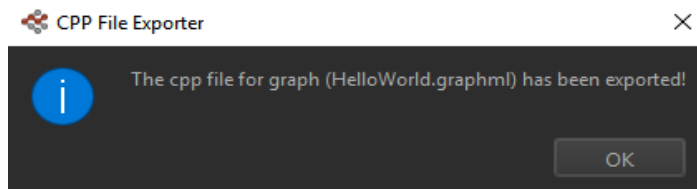


Node Name	Node Type	Input Port Type	Port	Output Port Type	Port	C++ Function Object
c0	continue_node	continue_msg		continue_msg		<pre>[(const continue_msg &m) -> continue_msg { printf("Hello"); return m; }]</pre>
c1	continue_node	continue_msg		continue_msg		<pre>[(const continue_msg &m) -> continue_msg { printf(" World!\n"); return m; }]</pre>

After drawing the graph, setting the node properties, and saving this graph as HelloWorld.graphml, the C++ stubs are generated by clicking the *Generate C++* icon on the toolbar, as shown below.



The generation of the stub files should be reported as successful:



The result of C++ code generation is one file, located in the same directory where the GraphML* file was last saved. The file name generated is HelloWorld_stubs.cpp. Below are the contents.



```
//
// Automatically generated by Flow Graph Analyzer:
// C++ Code Generator Plugin version XYZ
//

#define TBB_PREVIEW_FLOW_GRAPH_NODES 1
#include "tbb/flow_graph.h"
#include "tbb/tick_count.h"
#include "tbb/atomic.h"
#include <cstdlib>

using namespace std;
using namespace tbb;
using namespace tbb::flow;

size_t key_from_message(char *k) {
    return reinterpret_cast<size_t>(k);
}

template<typename T>
size_t key_from_message(const T &k) {
    return static_cast<size_t>(k);
}

static void spin_for( double weight = 0.0 ) {
    if ( weight > 0.0 ) {
        tick_count t1, t0 = tick_count::now();
        const double weight_multiple = 1e-6;
        const double end_time = weight_multiple * weight;
        do {
            t1 = tick_count::now();
        } while ( (t1-t0).seconds() < end_time );
    }
}

int build_and_run>HelloWorld_g0() {
    graph>HelloWorld_g0;

    source_node< continue_msg > s0(>HelloWorld_g0,
    [](>continue_msg &c) -> bool {
        static bool done = false;
        if (!done) {
            done = true;

```



```
        return true;
    } else {
        return false;
    }
}, false);

continue_node< continue_msg > c0( HelloWorld_g0, 0,
[] (const continue_msg &m) -> continue_msg {
    printf("Hello");
    return m;
});

continue_node< continue_msg > c1( HelloWorld_g0, 0,
[] (const continue_msg &m) -> continue_msg {
    printf(" World!\n");
    return m;
});

#ifdef TBB_PREVIEW_FLOW_GRAPH_TRACE
HelloWorld_g0.set_name("HelloWorld_g0");
s0.set_name("s0");
c0.set_name("c0");
c1.set_name("c1");
#endif

make_edge( s0, c0 );
make_edge( c0, c1 );

s0.activate();
HelloWorld_g0.wait_for_all();
return 0;
}

int main(int argc, char *argv[]) {
    return build_and_run_HelloWorld_g0();
}
```

In the code above, note the s0, c0, and c1 nodes reflect the properties described in the previous table.

If you have the paths to the Threading Building Blocks (TBB) library set up in your environment on a Windows* system, you can build this application using a Visual Studio* command prompt using the following command:

```
cl /EHsc HelloWorld_stubs.cpp tbb.lib
```



If you have the paths to the TBB library set up in your environment on a Linux* system, you can build this application using the following command:

```
g++ -std=c++11 HelloWorld_stubs.cpp -ltbb
```


In addition, Flow Graph Analyzer allows you to control execution policies for nodes, such as setting lightweight for computational nodes and asynchronous nodes. If you set lightweight policies for any node, the current code generator generates stubs for Threading Building Blocks(TBB) 2019.

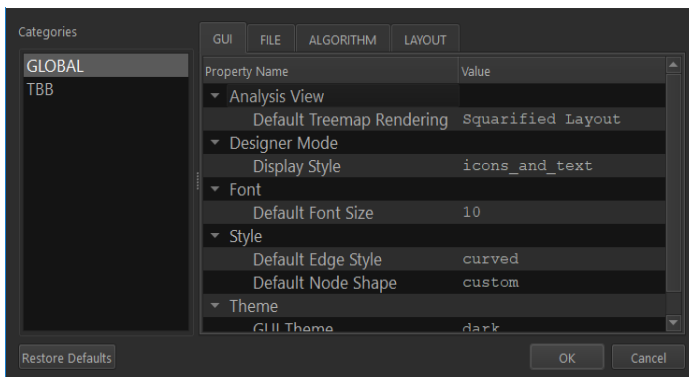
Lightweight policy is not supported with TBB 2018 U3 or lower. Hence, the nodes should not have lightweight policy enabled if you use TBB version that does not support lightweight policy. Lightweight policy is supported as preview feature with TBB 2018 U4 or U5. However, the generated code has to be compiled with TBB_PREVIEW_FLOW_GRAPH_LIGHTWEIGHT macro and linked against the tbb_preview library. See Reducing Scheduler Overhead using Lightweight Policy to learn more about how to set lightweight policy.

See more samples demonstrating this feature in the `samples/code_generation` directory.

Preferences

Use the *Preferences* dialog box to set your preferred global values for certain properties.

You can invoke the *Preferences* dialog box from the *Edit* menu or by clicking  on the toolbar.



To set a preference, simply change the property value. The Flow Graph Analyzer applies your preferences for the entire session and restores the preferences after shutdown and restart.

An example of a global preference is the node shape. Possible shape property values include box, circle, basic, uml, and custom. The default is the custom node shape.

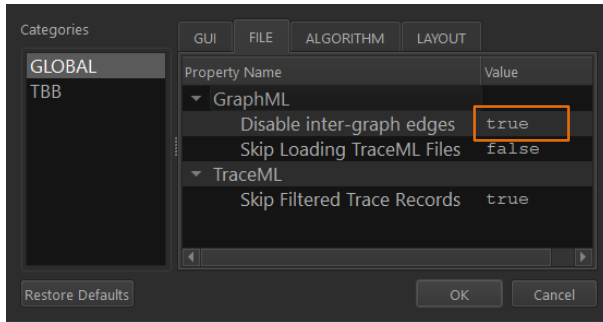
Another example of a global preference is the application theme. Possible themes include dark and light.

NOTE: The Flow Graph Analyzer applies a theme change only after restart.

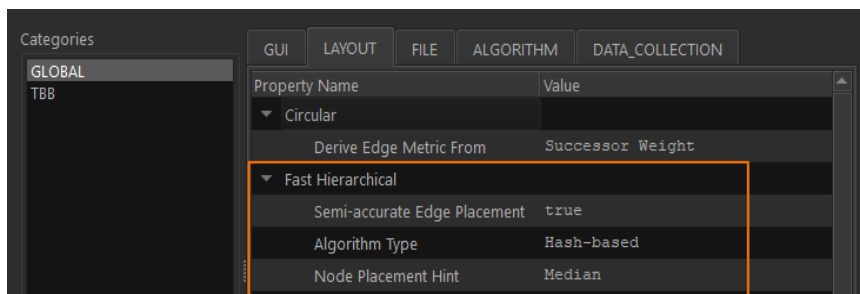


The Flow Graph Analyzer in Intel® Parallel Studio XE 2019 Update 4 or later supports viewing GraphML*files with inter-graph edges. This capability is disabled by default (set to true), as shown in the preferences dialog box below.

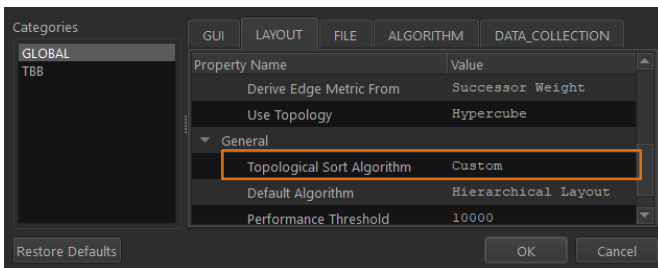
NOTE: Even though cross-graph edges are strongly discouraged in Threading Building Blocks documentation, some use-cases may require them.



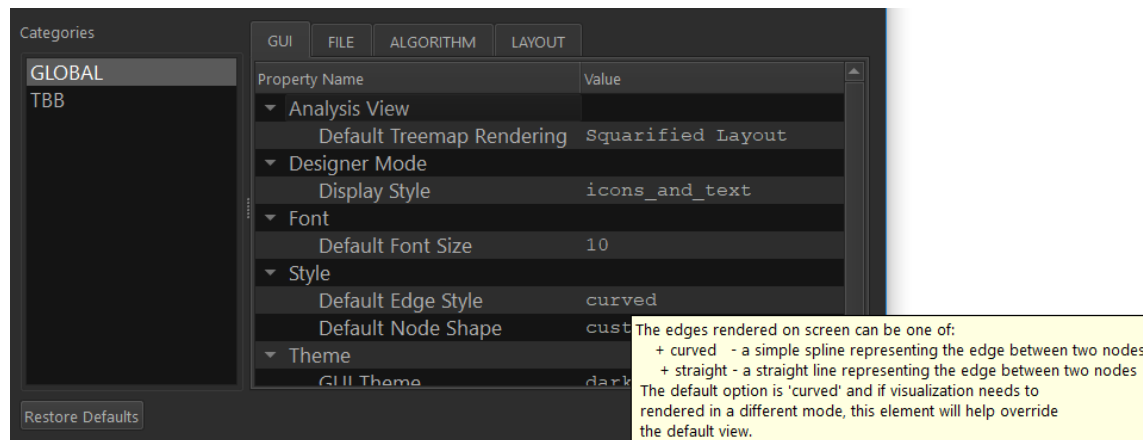
New fast hierarchical layout algorithm has been introduced in Flow Graph Analyzer in Intel® Parallel Studio XE 2019 Update 5. This allows user to render graphs with semi accurate edge placement and different node placement with respect to its children such as Median, Average or Minimum distance. There are three different algorithms for this layout that can be applied and each has its own performance characteristics. Simple algorithm type sacrifices accuracy for speed, Hash-based type honors node placement and hash based dfs helps with better visual quality. The preference settings for this new layout are highlighted in below image.



User can also select between two different depth-first search (DFS) algorithms for performing a topological sort of the graph: (1) custom sort algorithm is optimized for topologies that are frequently encountered in TBB and openMP and (2) Boost library implementation of DFS. These settings can be changed in the preferences as shown in below image.



To get more information about a preference property, hover your mouse over the property to enable a tooltip that contains more information on the property. See screenshot of tooltip for node shape below:



Scalability Analysis

Typically, during the design of a parallel application, it is essential to have an idea of how the application scales with the addition of multiple threads. It is useful to know if application performance continues to increase with the addition of more threads or tapers off at some point.

The parallelism in a graph is usually provided by the topology of the graph as well as the inherent parallelism provided within nodes that have unlimited concurrency. In a complex application, it is not obvious what the overall parallelism of the application is. Even when this information is available, it is not obvious what the contributing factors are. To gain insight into these issues, use the scalability analysis plugin.

The scalability analysis plugin allows you to run a graph with a varying number of threads, and provides speedup information of the graph with respect to running the graph serially. It allows for exploration of the parallelism provided by the topology of the graph as well as the parallelism provided by nodes with unlimited concurrency. Any desired configuration of the graph can be made and run to aid in the design and analysis of the graph.

Running an Analysis



The scalability analysis is triggered by clicking the *Scalability* icon on the toolbar:

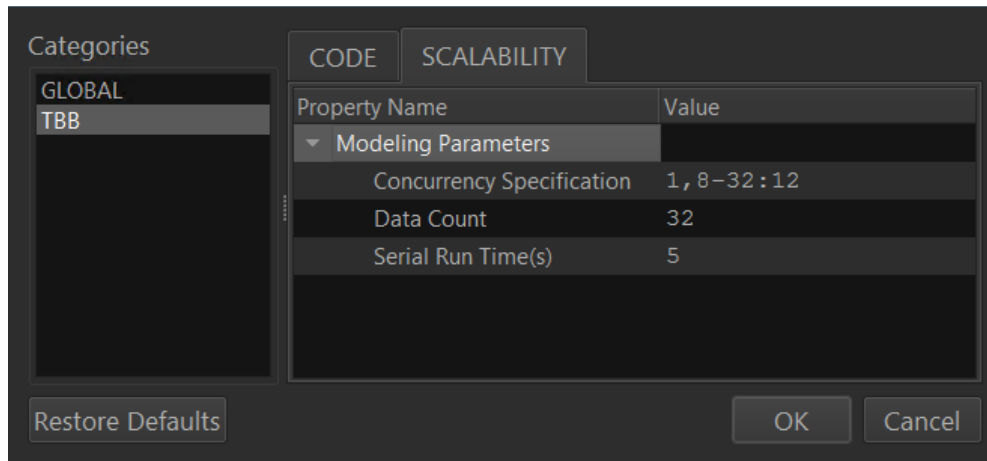
To run the analysis on a graph, the following are required: A concurrency specification, a data count value and node weights. These are described next.

Concurrency Specification

The concurrency specification is a list of the number of threads on which the graph is run. Supply this list in the *Concurrency Specification* field as shown below in the *Preferences* dialog box. Because the speedup



is with respect to a serial run of the graph, by default, all graphs are first run serially before run on multiple threads.



The default format is $1, a-b:n$, where

- 1 signifies the serial run. Because the graph is always run serially, the addition of 1 is merely symbolic and informational, as its omission has no effect on the list.
- a is the starting number of multiple threads
- b is the ending number of multiple threads
- n is the step size

The default end value is the number of cores on the system and the default step size is a quarter of this value. The end value is always included in the list, even if it is not a direct multiple of the step size.

As an example, on a 32-core system, the default is $1, 8-32:8$. This results in the list 1, 8, 16, 24, 32, as shown below.

Graph Name	Graph	Threads
untitled...	Results(5)	
	Scalability projection	1
	Scalability projection	8
	Scalability projection	16
	Scalability projection	24
	Scalability projection	32

Other supported formats are a,b,c and also simply a if you are interested in discovering the speedup for only one set of threads.

You can also combine formats. The following shows some valid concurrency specifications:

- Range without a step size: 16-64 yields 1, 16, 32, 48, 64
- Range with a step size: 16-64:8 yields 1, 16, 24, 32, 40, 48, 56, 64
- Single value: 16 yields 1, 16



- List of values: 16, 32, 64 yields 1, 16, 32, 64
- Mix of range and list: 16-64, 40, 48, 56 yields 1, 16, 32, 40, 48, 56, 64

Data Count

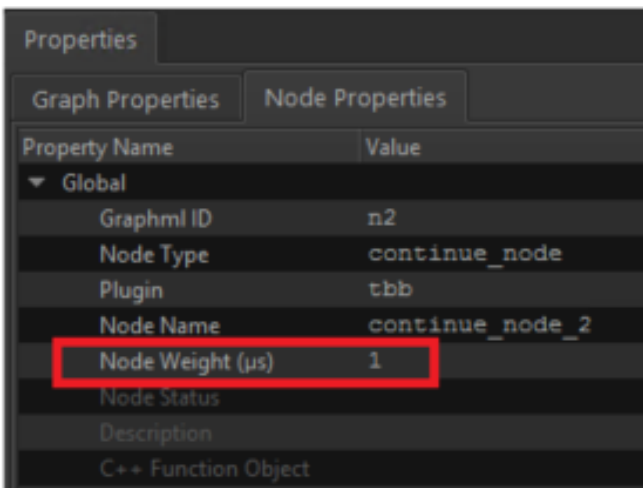
Data count is the number of data items generated and pushed through the graph.

For graphs where potential parallelism is apparent from topology, pushing a single item through the graph suffices to explore the parallelism of the graph.

However, for cases where the parallelism is not readily apparent, as with a node with an unlimited concurrency, pushing a single item through the node is not enough to explore parallelism. Therefore, to ensure the graph is saturated, set the default number of data items to the number of cores available on the system in the *Data Count* field, based on the topology of the graph and the type of nodes in the graph.

Weight

The weight determines the simulated amount of time the node spins or does active work per data item passed to the node. The default unit for the weight is microseconds. For example, per the default unit of microseconds, a weight value of 1000 makes a node spin for half a second. Note therefore that if the specified weight is high relative to the unit, the computation could run for a relatively longer time. If the graph has an associated trace, the unit of the weight is overwritten by the unit specified in the trace. Edit the weight in the *Node Weight* field, as shown below.



Only nodes with a body take into account the value of the weight. Examples of such nodes are `source_node`, `continue_node`, `function_node`, `multifunction_node`, `async_node`, and `tag_matching join_node`. Other nodes that simply assist in the topology of the graph do not use the weight value. For instance, entering a weight value for a `broadcast_node` has no effect on the node.

To prevent long runtimes, the scalability analysis scales all runs with total serial times beyond a certain threshold. The current default threshold is 5 seconds. You can modify this value in the *Serial Run Time(s)* field in the *Preferences* dialog box. (Edit -> Preferences -> TBB -> Serial Run Time(s)).



The total serial runtime also takes into account the number of data items passed through the graph. For example, a graph with a serial runtime of 10 seconds and 4 data items has a total serial runtime of 40 seconds.

To ensure node weights are not scaled into regions that make the overhead dominant, the analysis uses original weights and does not implement scaling if no node has a weight of 100 microseconds after scaling.

If a graph contains nodes with performance that could benefit from the use of the TBB lightweight policy, the analysis activates the lightweight policy for the recommended nodes and lists the possible improvement in the results.

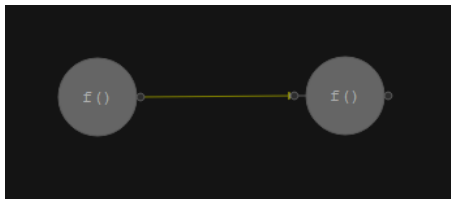
Activating the Graph

To run the analysis, the graph must contain at least one source_node for activation. This source_node(s) pushes the data items through the graph.

The plugin internally ensures that valid data types flow through the graph, so even if a rule check on the data types fail, the scalability analysis still runs. Therefore selecting and connecting nodes on the canvas should suffice; there's no further need to edit the input and output data types to ensure they match. Note however this might be required for other plugins, like the code generator, to ensure accurate code generation.

Exploring the Parallelism in a Concurrent Node

In this example, we explore the parallelism inherent in a node with unlimited concurrency. The node used is a function_node. A source_node is connected to the function_node, as shown below, and 8 items are pushed through to the function_node from the source_node. The function_node has a weight of 1s (weight = 1e6). To ensure we only have timing results from the function_node, the source_node has a comparatively negligible weight of 1e-6s (weight=1). The concurrency specification used is 1, 2, 4, 8. The results are shown below.



Debug Output	Statistics	Analytics Report	
Graph Narr ▾	Graph	Threads	Time(s) Speedup
▼ untitled...	Results(4)		
●	Scalabili... 1	8.00019	1
●	Scalabili... 2	4.0001	2
●	Scalabili... 4	2.00012	3.99986
●	Scalabili... 8	1.00014	7.9991



As you can see, for a serial execution with only 1 thread, the total time is 8s as the same thread evaluates the tasks one after the other. With 2 threads and 2 overlapping tasks, the total time is 4s. Similarly, with 4 threads and 8 tasks, the total time is 2s. With 8 threads and 8 tasks, all tasks overlap, giving a total time of 1s.

Showing Non-Parallel Nature of a Serial Node

Conversely, you can also use the plugin to ascertain if some nodes have no parallelism or to tell when an unlimited node is running serially.

For this example, a function_node is once again used, but this time the concurrency in the function_node is reduced to 1. That is, it runs serially. This can be done by entering either 1 or `serial` in the *Node Concurrency* field of the function_node. The same concurrency specification of 1, 2, 4, 8 is used. As with the previous example, the weight of the function_node is 1s and the weight of the source_node is 1e-6s. The results are shown below.

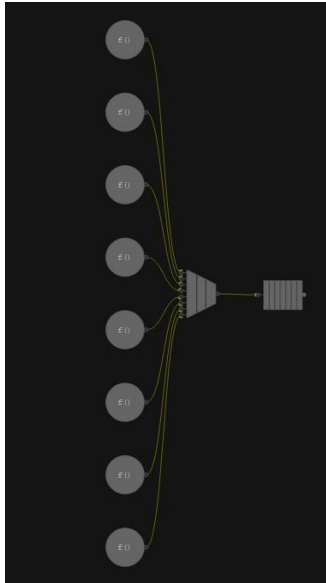
Debug Output	Statistics	Analytics Report			
Graph Name	Graph	Threads	Time(s)	Speedup	
▼ untitled_0	Results(4)				
	Scalability projection	1	8.00017	1	
	Scalability projection	2	8.00009	1.00001	
	Scalability projection	4	8.00007	1.00001	
	Scalability projection	8	8.00007	1.00001	

As expected, time is 8s, regardless of the number of threads used.

The same results are obtained using a source_node with a weight of 1s and a function_node with a relatively negligible weight. This is because a source_node executes its body serially.

Exploring the Parallelism Provided by the Topology of a Graph

In this example, we explore the parallelism provided by the topology of a graph. To make the results as predictable as possible, we used a graph that is almost embarrassingly parallel, as shown below.



Because the source_node is serial, there is no parallelism provided from within the node. This ensures all parallelism observed is provided by the topology of the graph. Eight source_nodes are connected to a join_node and then to a queue_node. In this graph, only the source_nodes do useful work. Because the parallelism is solely from the topology of the graph, it suffices to only pass 1 item per source_node through the graph. Each source_node has a weight of 1s(1e6). The results are shown below.

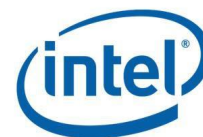
Graph Name	Graph	Threads	Time(s)	Speedup
▼ untitled_0	Results(4)			
	Scalability projection	1	8.00177	1
	Scalability projection	2	4.00031	2.00029
	Scalability projection	4	2.00035	4.00017
	Scalability projection	8	1.0005	7.99775

Once again, the speedup is directly proportional to the number of threads.

Understanding Analysis Color Codes

The results from the scalability analysis runs are color coded. Green dots denote results obtained from runs where the number of threads is either equal to or less than the number of cores on the system. If the number of threads is greater than the number of cores, the results are coded in blue.

For example, the following image shows the result for a concurrency specification of 1, 4, 8, 32, 48, 64 on a 32-core system.



Graph Name	Graph	Threads	Time(s)	Speedup
▼ untitled_0	Results(6)			
	Scalability projection	1	8.00019	1
	Scalability projection	4	2.00016	3.99977
	Scalability projection	8	1.00011	7.99933
	Scalability projection	32	1.0001	7.99942
	Scalability projection	48	1.0001	7.99941
	Scalability projection	64	1.0001	7.99936

Results for runs with less than 32 threads are coded in green, while those with more than 32 threads are coded in blue.

Collecting Traces from Applications

This section explains how to collect traces from an application that uses the Threading Building Blocks (TBB) flow graph interfaces.

Prerequisites

You need the following to collect traces from an application:

- An application that uses the TBB library flow graph interface
- TBB 4.3 or later installed on your system
- The Flow Graph Collector library installed on your system

Check the links in the [Additional Resources](#) section if you are missing any of these prerequisites.

Simple Sample Application

This section uses the sample code below as a running example. Assume this code is contained in a file `example.cpp`. You can use your own application or sample in place of this simple example if desired.

```
#include "tbb/flow_graph.h"
#include <iostream>
using namespace std;
using namespace tbb::flow;
int main() {
    graph g;
    continue_node< continue_msg> hello( g,
        [] ( const continue_msg &) {
            cout << "Hello";
        }
    );
    continue_node< continue_msg> world( g,
        [] ( const continue_msg &) {
            cout << " World\n";
        }
    );
}
```



```
);  
make_edge(hello, world);  
hello.try_put(continue_msg());  
g.wait_for_all();  
return 0;  
}
```

Building the Application

To build an application enabled for trace collection:

1. Use an environment that meets the system requirements described in the [Software Requirements for Collector](#) section.
2. For TBB 2019 or above version, define TBB_USE_THREADING_TOOLS macro and link against tbb library.
3. For TBB versions prior to TBB 2019, define TBB_PREVIEW_FLOW_GRAPH_TRACE macro and link against tbb_preview or tbb_preview_debug library.
4. Compile using Threading Building Blocks (TBB) 4.3 or later.

For TBB 2019 and later versions, the TBB_USE_THREADING_TOOLS macro activates the needed instrumentations in the flow_graph.h header and the tbb library supports flow graph and algorithm profiling. All features other than set_name extensions are available as non-preview features.

For versions prior to TBB 2019, The TBB_PREVIEW_FLOW_GRAPH_TRACE macro activates the needed instrumentation in the flow_graph.h header. The tbb_preview and tbb_preview_debug libraries offer support for Preview features not yet supported in the main library. The instrumentation needed by the Flow Graph Analyzer is supported as a Preview feature in TBB 4.3 and newer versions of the libraries.

Building on a Windows Operating System*

Using a Visual Studio Command Prompt*

After you open a Visual Studio* command prompt and set up the proper paths for using the Threading Building Blocks (TBB) library, use the following command line to build a Release executable for the running example:

TBB version 2019 and later,

```
cl /EHsc /DTBB_USE_THREADING_TOOLS example.cpp tbb.lib
```

Prior to TBB version 2019,

```
cl /EHsc /DTBB_PREVIEW_FLOW_GRAPH_TRACE example.cpp tbb_preview.lib
```

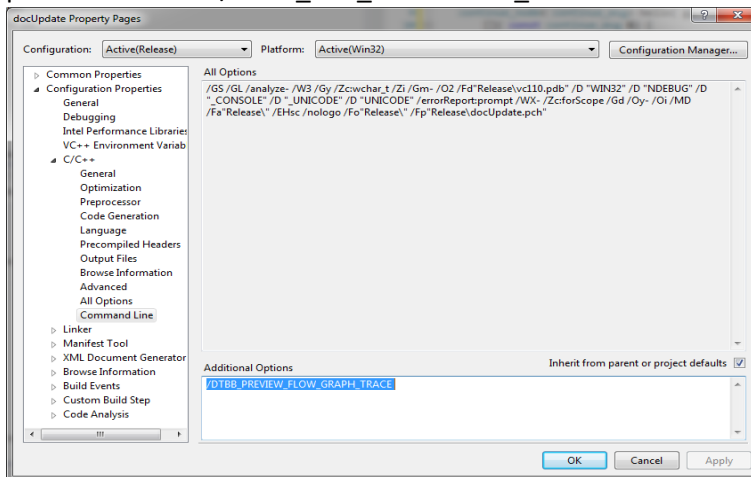
As explained above, this command line defines the required macro and also links the application against appropriate tbb library based on the version you use.



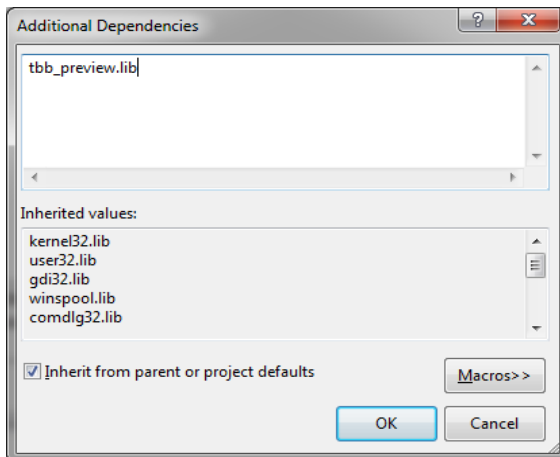
Building from a Visual Studio* IDE

To build a Release configuration within a Visual Studio* IDE, you must change your project to define the `TBB_PREVIEW_FLOW_GRAPH_TRACE/TBB_USE_THREADING_TOOLS` macro and link against the `tbb_preview.lib/tbb.lib`, as shown below for the Visual Studio* 2015 IDE based on the TBB version you use.

1. Open the *Project Properties* dialog box and select **Configuration Properties > C/C++ > Command Line**. In the *Additional Options* textbox, enter: `/DTBB_PREVIEW_FLOW_GRAPH_TRACE` for TBB versions prior to 2019 and `/DTBB_USE_THREADING_TOOLS` for TBB version 2019 and later.



2. Select **Configuration Properties > Linker > Input**. In the *Additional Dependencies* field, enter: `tbb_preview.lib` for TBB versions prior to 2019 and `tbb.lib` for TBB version 2019 and later.



For a Debug build, enter: `tbb_preview_debug.lib` for TBB versions prior to 2019 and `tbb_debug.lib` for TBB version 2019 and later.

Building on a Linux* Operating System

Use the following command line to build an executable for the running example:

TBB version 2019 and later,



```
icpc -std=c++11 -DTBB_USE_THREADING_TOOLS example.cpp -ltbb
```

Prior to TBB version 2019,

```
icpc -std=c++11 -DTBB_PREVIEW_FLOW_GRAPH_TRACE example.cpp -ltbb_preview
```

As explained above, this command line defines the `TBB_USE_THREADING_TOOLS/`
`TBB_PREVIEW_FLOW_GRAPH_TRACE` macro and also links the application against `tbb.lib/`
`tbb_preview.lib` based on the TBB version you use. `-std=c++11` is present because the running
example uses lambda expressions, a C++11 feature.

In order to [map nodes to source code](#), use `-g` and `-DTBB_PREVIEW_FLOW_GRAPH_TRACE` flag to
build the application along with `-DTBB_USE_THREADING_TOOLS`.

```
icpc -g -std=c++11 -DTBB_USE_THREADING_TOOLS -DTBB_PREVIEW_FLOW_GRAPH_TRACE  
example.cpp -ltbb_preview
```

Building on a Mac Operating System*

Use the following command line to build an executable for the running example:

TBB version 2019 and later,

```
clang++ -std=c++11 -DTBB_USE_THREADING_TOOLS example.cpp -ltbb
```

Prior to TBB version 2019,

```
clang++ -std=c++11 -DTBB_PREVIEW_FLOW_GRAPH_TRACE example.cpp -ltbb_preview
```

As explained above, this command line defines the `TBB_USE_THREADING_TOOLS/`
`TBB_PREVIEW_FLOW_GRAPH_TRACE` macro and also links the application against `tbb.lib/`
`tbb_preview.lib` based on the TBB version you use. `-std=c++11` is present because the running
example uses lambda expressions, a C++11 feature.

Collecting the Trace Files


While executing, your application only collects trace files if there is an appropriate collector library at the
locations specified by the `INTEL_LIBITNOTIFY32` or `INTEL_LIBITNOTIFY64` environment variables. The
`INTEL_LIBITNOTIFY32` path is searched by 32-bit executables and the `INTEL_LIBITNOTIFY64` path is
searched by 64-bit executables.

There are two options for collecting and converting trace files for use with Flow Graph Analyzer:

- Collect traces by starting your application from the Flow Graph Analyzer GUI.
- Set up your environment so a trace is collected when the application is started outside of the GUI.

Both approaches require the application to be traced is built using the previously described steps
described.

Collecting Traces Inside the Flow Graph Analyzer GUI

To run an existing TBB application and collection execution trace information for analysis, the tool allows you to launch the TBB trace collector feature by clicking on the *Run and collect traces* option under the *Offload Actions* menu or by clicking the  icon in the toolbar. Assuming the paths for the Threading Building Blocks (TBB) libraries are set up (*see Release Notes and Known Issues for limitations*),

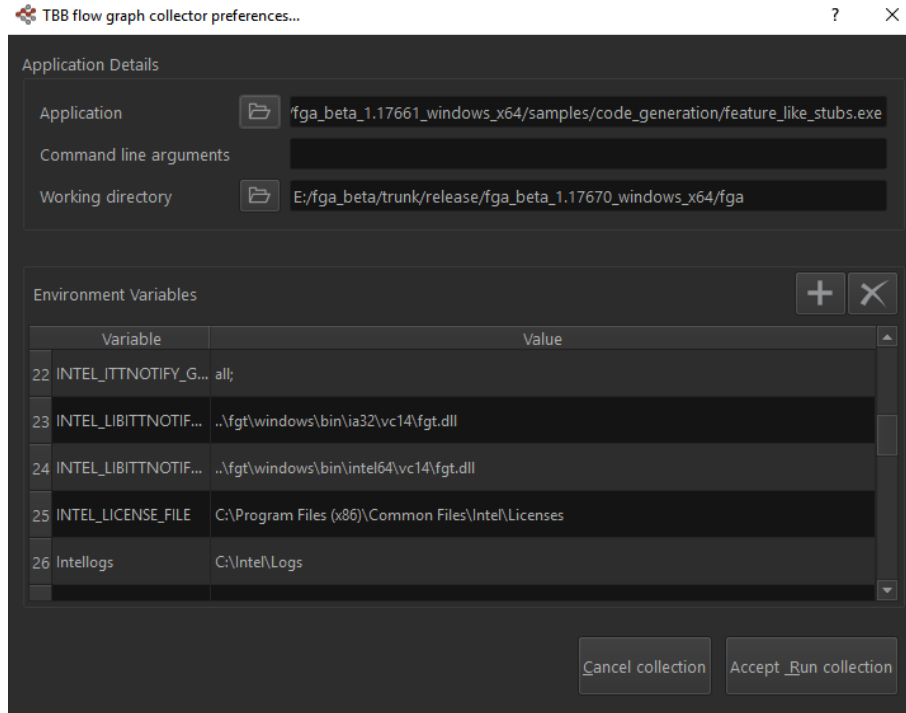
1. Run the Flow Graph Analyzer from the installation directory.
2. Click the *Run and Collect Traces* icon on the toolbar:



The environment variables for running trace collection have default settings if they are not set in the environment from which the Flow Graph Analyzer is launched. The inherited values are used if the environment variables are set in the environment.

OS	Environment Variable	Default Value	Description
Windows*	INTEL_LIBITNOTIFY32	..\fgt\windows\bin\ia32\vc14.1\fgt.dll	32-bit collector library
	INTEL_LIBITNOTIFY64	..\fgt\windows\bin\intel64\vc14.1\fgt.dll	64-bit collector library
	INTEL_ITTNOTIFY_GROUPS	all;	Trace events from all groups
Linux*	INTEL_LIBITNOTIFY32	../fgt/linux/lib/ia32/cc4.8_libc2.19_kernel3.13.0/libfgt.so	32-bit collector library
	INTEL_LIBITNOTIFY64	../fgt/linux/lib/intel64/cc4.8_libc2.19_kernel3.13.0/libfgt.so	64-bit collector library
	INTEL_ITTNOTIFY_GROUPS	all;	Trace events from all groups
macOS*	INTEL_LIBITNOTIFY32	../fgt/macos/lib/ia32/osx10.12.6_kernel16.7.0/libfgt.dylib	32-bit collector library
	INTEL_LIBITNOTIFY64	../fgt/macos/lib/intel64/osx10.12.6_kernel16.7.0/libfgt.dylib	64-bit collector library
	INTEL_ITTNOTIFY_GROUPS	all;	Trace events from all groups

The collector *Preferences* window lets you specify the application to run:



The *Working Directory* defaults to the Flow Graph Analyzer directory. The *Debug Output* pane displays any error messages and the output of the application. You can view and set other environment variables, including INTEL_LIBITNOTIFY32 and INTEL_LIBITNOTIFY64, using the *Environment Variables* section of the dialog box.

After you click the **Accept Run collection** button, the application executes and, if it completed normally, the trace files are converted to GraphML* format. The output file is stored in the working directory, with a name based on the executable name and the time of the trace collection run. You must manually load the GraphML* file into the GUI to examine it.

Collecting Traces Outside the Flow Graph Analyzer GUI

If you can launch your application from a Windows* command prompt or Linux* terminal, the simplest approach is to use an `fgtrun` script to run your application. If you cannot launch your application from a prompt or terminal, or you want to launch it from within an IDE, you must manually perform the steps performed by the `fgtrun` script.

In either case, you must update your PATH environment variable to add the paths to essential tools, as described below. You must also set the FGT_ROOT variable.

The directions in this section assume the full path to your Flow Graph Analyzer installation is `<advisor-install-dir>\fga<version>`. The version of your Visual Studio* compiler is `<vc version>`, with possible values of `vc12`, `vc14` and `vc14.1`.



OS	Environment Variable	Value	Description
Windows*	FGT_ROOT	<advisor-install-dir>\fga\fgt	The path to the Flow Graph Collector installation
	PATH	%FGT_ROOT%\windows\bin;%FGT_ROOT%\windows\bin\ia32\<vc version>;%FGT_ROOT%\windows\bin\intel64\<vc version>;%PATH%	The path should include paths to fgtrun.bat and fgt2xml.exe
Linux*	FGT_ROOT	<advisor-install-dir>/fga/fgt	The path to the Flow Graph Collector installation
	PATH	\${FGT_ROOT}/linux/bin:\${FGT_ROOT}/linux/bin/ia32/cc4.8_libc2.19_kernel3.13.0:\${FGT_ROOT}/linux/bin/intel64/cc4.8_libc2.19_kernel3.13.0:\${PATH}	The path should include paths to fgtrun.sh, fgtrun.csh, and fgt2xml
macOS*	FGT_ROOT	<advisor-install-dir>/fga/fgt	The path to the Flow Graph Collector installation
	PATH	\${FGT_ROOT}/macos/bin:\${FGT_ROOT}/macos/bin/ia32/osx10.12.6_kernel16.7.0:\${FGT_ROOT}/macos/bin/intel64/osx10.12.6_kernel16.7.0:\${PATH}	The path should include paths to fgtrun.sh, fgtrun.csh, and fgt2xml

Collecting Trace Files with an fgtrun Script

Use the `fgtrun` script to collect the trace information from your application. The script sets the paths necessary to execute your application and generate the GraphML* and TraceML* files that can be loaded into the Flow Graph Analyzer for visualization. The script requires the variable `FGT_ROOT` be set before it is invoked. The following table lists the directories in which the scripts are located on a given system.

OS	Version	Location	Example Use
Windows*	fgtrun.bat	%FGT_ROOT%\windows\bin	fgtrun.bat app-binary-name [binary-args] [--ia32/ --intel64] [--vc12/ --vc14/ --vc14.1] [--xml]
Linux*	fgtrun.sh	\${FGT_ROOT}/linux/bin	fgtrun.sh app-binary-name [binary-args] [--ia32/ --intel64] [--omp] [--xml]
macOS*	fgtrun.sh	\${FGT_ROOT}/macos/bin	fgtrun.sh app-binary-name [binary-args] [--ia32/ --intel64] [--omp] [--xml]

The `fgtrun` script attempts to automatically detect the architecture and C/C++ runtime version (Windows* OS only) of the executable used to collect the traces and requires the presence of helper tools. If the helper tools are not available or fail to identify the required information, `fgtrun` scripts sets default values and runs the collection. Optionally you can override this default values by setting architecture and C/C++ runtime version information (Windows* OS only) using command line arguments when the script is invoked.



Passing `--omp` as argument to the `fgtrun` script enables omp trace for applications linked with `qopenmp`. The OpenMP* runtime environment must be set correctly before the script is launched.

NOTE: This OpenMP* trace collection capability is currently not supported on the Windows* OS.

The default behavior of the collector is to generate binary traces. If trace collection fails, you can switch to XML trace generation mode to debug the cause of the failure. To collect traces in XML format, pass `--xml` as argument to the `fgtrun` script.

To get mapping between [nodes and source code](#) `--sym` argument should be passed to `fgtrun` script.

NOTE: Symbol resolution feature is currently only supported on Linux.

You can optionally run each of the steps performed by the `fgtrun` script manually. The next section describes these steps in more detail.

Collecting Trace Files without an fgtrun Script

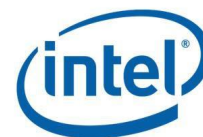
You may choose to collect traces without using the `fgtrun` script if you do not want to launch your application from a Visual Studio* command prompt or Linux* terminal. If this is the case, follow the steps below to collect and convert trace files manually.

NOTE: These steps do not outline steps required to capture symbol resolution information.

1. Set the paths to the collector libraries.

Set up or update the environment variables shown below. For Windows* systems, specify the proper version of the Visual Studio* compiler (vc12 or vc14 or vc14.1).

OS	Environment Variable	Default Value	Description
Windows*	INTEL_LIBITNOTIFY32	..\fgt\windows\bin\ia32\<vc version>\fgt.dll	32-bit collector library
	INTEL_LIBITNOTIFY64	..\fgt\windows\bin\intel64\<vc version>\fgt.dll	64-bit collector library
	INTEL_ITTNOTIFY_GROUPS	all;	Trace events from all groups
Linux*	INTEL_LIBITNOTIFY32	../fgt/linux/lib/ia32/cc4.8_libc2.19_kernel3.13.0/libfgt.so	32-bit collector library
	INTEL_LIBITNOTIFY64	../fgt/linux/lib/intel64/cc4.8_libc2.19_kernel3.13.0/libfgt.so	64-bit collector library
	INTEL_ITTNOTIFY_GROUPS	all;	Trace events from all groups
macOS*	INTEL_LIBITNOTIFY32	../fgt/macos/lib/ia32/osx10.12.6_kernel16.7.0/libfgt.dylib	32-bit collector library



OS	Environment Variable	Default Value	Description
	INTEL_LIBITNOTIFY64	../fgt/macos/lib/intel64/osx10.12.6_kerne l16.7.0/libfgt.dylib	64-bit collector library
	INTEL_ITTNOTIFY_GROUPS	all;	Trace events from all groups

If you want the environment variables set in a Visual Studio* command prompt to be used by the Visual Studio* IDE, you can launch the Visual Studio* IDE from the command prompt using the following command:

```
devenv /useenv
```

2. Run the application.

If your paths are set up correctly, the application generates one or more files that start with `_fgt`. There is one file per thread that participates in executing the parallelism in the application. So, for example, if two threads participate in the execution of the flow graph, your run generates two files: `_fgt.0` and `_fgt.1`, with an autogenerated folder in the format `_fga_YYYYMMDD_HHMMSS` according to its creation (for example, `20191111_1111`).

3. Convert the trace files to GraphML* and TraceML* format.

Convert the `_fgt` binary files to the XML format understood by the Flow Graph Analyzer tool using the `fgt2xml.exe` converter in the directory containing the folder with the trace files:

```
fgt2xml.exe desired_name
```

This converter scans the current directory for all `_fgt` files within the most recent folder according to its name, and generates two output files: `desired_name.graphml` and `desired_name.traceml`.

Nested Parallelism in Flow Graph Analyzer

Flow Graph Analyzer supports visualization of applications that contain multiple levels of parallelism, such as nested TBB algorithms and OpenMP* parallel regions. This feature requires additional support from the parallel runtime libraries and can be used in combination with TBB flow graph.

The sample code below represents an example of nested parallelism that combines a TBB flow graph, a TBB `parallel_for` algorithm, and an OpenMP* parallel region.

```
#include "tbb/tbb.h"
#include "tbb/flow_graph.h"
#include <omp.h>
#include <iostream>
```

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



```
graph g;
const int size = 20;
continue_node< continue_msg> hello( g,
    []( const continue_msg &) {
        std::cout << "Hello\n";
        tbb::parallel_for(0, size, 1, [=](int k) {
            std::cout << k << "\n"; });
    });
continue_node< continue_msg> world( g,
    []( const continue_msg &)
        std::cout << " World\n";
    #pragma omp parallel for
        for (int i=0; i<20; i++) {
            std::cout << i << "\n"; } }
);
make_edge(hello, world);
hello.try_put(continue_msg());
g.wait_for_all();
return 0;
}
```

Tracing Nested TBB Algorithms

Threading Building Blocks (TBB) 2019 enables general tracing of parallel algorithms, which is enabled by default and activated by the Flow Graph Analyzer trace collector.

As a result, Flow Graph Analyzer can display TBB library activity in nested and non-nested algorithms. Therefore, task context switches are captured and can be visualized in the Flow Graph Analyzer GUI. This work appears similar to tasks in the timeline and is named according to its algorithm (for example `parallel_for`). This information might not be available for user-defined task groups.

Tracing Nested OpenMP Algorithms*

For detailed information on Flow Graph Analyzer support for OpenMP* technology, see [Experimental Support for OpenMP* Applications](#).

Analyzer Workflow

NOTE: This section describes a recommended workflow to identify performance issues in the executed graph. This workflow may change as more analytics plugins are added; however, the fundamental principal should not change, as the goal is to maximize the throughput of the graph in a streaming case and provide the best scaling performance with respect to the serial run.

The Flow Graph Analyzer provides the following capabilities for analyzing flow graph performance:

Display the graph for which the execution trace is captured. See

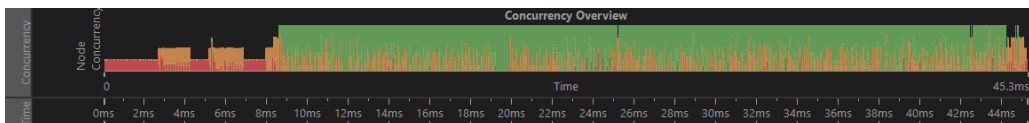


- Preferences section for details on how to enable loading graphml files that contain graphs with cross-graph edges.
- Display the trace information in a manner that highlights parallel performance issues.
- Map poorly scaling time regions to nodes executing at that time.
- Compute the critical path of the graph.
- View compute statistics for the computational nodes based on the execution traces.
- View prioritized diagnostics.

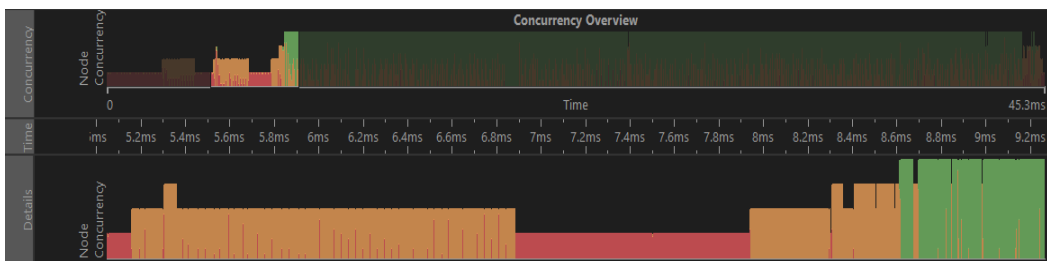
We recommend performing the following steps to analyze performance.

Finding Time Regions of Low Concurrency and Their Cause

1. Inspect the node concurrency histogram for regions in red, which indicate low concurrency.



2. Zoom in to a red region to inspect the data at a higher resolution and provide a better idea as to how concurrency varies over time.



3. Select a point in the chart where the concurrency is low to highlight the relevant node(s). Hover the mouse over the highlighted node to identify the node name.

Because the analysis tool does not have built-in symbol resolution, the determination of the C++ class of the body executed by a node must be explicitly encoded into the application. Explicit encoding affects the *Name* and/or *object_name* fields in the *Node Properties* tab. For example:

```
tbb::flow::graph g;
...
tbb::flow::source_node<int> s_node (g, source_node_body(),
                                     false);

#if TBB_PREVIEW_FLOW_GRAPH_TRACE
s_node.set_name("My Source Node");
#endif
```

This coding enables node annotation with the specified string during trace collection, and the annotation appears when you hover the mouse over the node. **NOTE:** The `set_name` functions are only available when the `TBB_PREVIEW_FLOW_GRAPH_TRACE` macro is defined at compile-time.

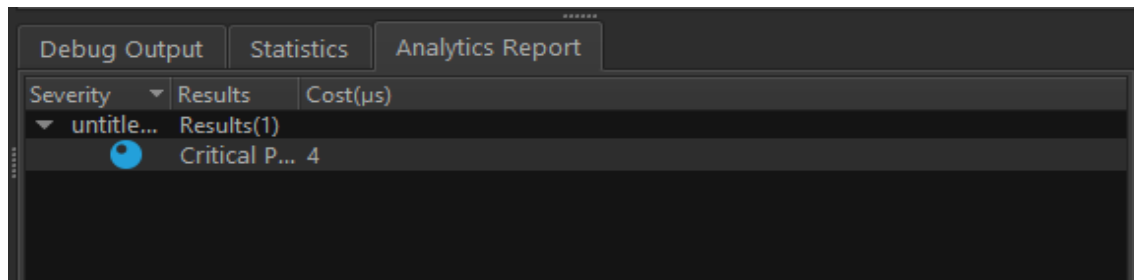


Finding the Critical Path

The critical path from a node N to a node M is the longest (in time) path from N to M and its length is a lower-bound on the execution time of a computation that begins at N and ends at M.

The Critical Path Analytic in the Flow Graph Analyzer computes a critical path for each source node/sink node pair. Therefore a given graph has as many critical paths as the product of the number of source nodes and the number of sink nodes. Source nodes are the nodes in the graph without any predecessors, or nodes with an in-degree of zero. Sink nodes are the nodes in the graph without any successors, or nodes with an out-degree of zero.

Click the *Compute Critical Path* icon on the toolbar to calculate the critical paths in the graph. These critical paths are displayed in descending order by cost. Inspect the topmost critical path first because, as the longest critical path, it sets the lower bound on the execution time for the whole graph. The screenshot below shows a sample critical path report.



Selecting the critical path in the *Analytics Report* window highlights all the nodes on the critical path.

Finding Tasks with Very Small Durations

Tasks executed by a flow graph are spawned as Threading Building Blocks (TBB) tasks, and therefore the duration of a task must be large enough to amortize the cost of a task spawn.

1. Click the *Graph* tab under the *Statistics* tab in the bottom pane to calculate execution time metrics by computing the mean and standard deviation for each node based on the execution traces.
2. Sort the resulting data based on the *Avg Task Duration* column to identify the nodes with the smallest average durations.

Any node that exhibits an average duration in the range of a few microseconds warrants additional inspection, because any concurrency gained by its parallel execution may be overwhelmed by its scheduling costs.



Debug Output Statistics Analytics Report Execution Trace Views									
Graph		Data Analysis							
Severity	For	Node Name	Instance Count	In-degree	Out-degree	Total Time (μs)	Avg. Task Duration (μs)	Std. Dev.	Average Concurrency
▼ g0	Resu...								
	Node	src	251	0	1	1.04226e+09	4.15242e+06	1.2806e+06	4
	Node	buffers	unset	1	1	unset	unset	unset	unset
	Node	resource_join	unset	2	1	unset	unset	unset	unset
	Node	preprocess_fun...	250	1	2	2.00888e+09	8.03554e+06	1.79901e+06	2.44
	Node	detect_A	250	1	1	1.55358e+09	6.2143e+06	745388	1.65
	Node	detect_B	250	1	1	1.57988e+09	6.31952e+06	635088	1.99
	Node	detection_join	unset	2	1	unset	unset	unset	unset
	Node	decide	250	1	1	4.46198e+07	178479	410715	1

The screenshot above shows a sample graph that executed 250 times. Notice the *Count* column, which depicts the number of times a node executes, is 250 for all functional nodes. Inspect the *Avg Task Duration* column for nodes that execute in a few microseconds. **NOTE:** Times are provided in milliseconds.

An alternate way to visualize task durations and the average concurrencies of each node is the Treemap view in the *Analysis Mode* tab. If multiple graphs are present in the application run, you see a high-level treemap showing the health of all the graphs in the run.

The Treemap view organizes the nodes in the graph based on node durations. The larger the area of the square, the more time the node spent on the CPU. The node color is determined by the average observed concurrency of the graph while the node was running, and the colors use the same scale as in the active thread chart.



The sample Treemap view above shows the transitions that occur when you hover the mouse over or double-click a treemap entry.



A red node likely indicates poor concurrency when the node executed on the system.

When a graph has nested subgraphs, the treemap presents this information by embedding the nested subgraph in the node which spawned the subgraph. This is visually represented by increasing the width of the border surrounding each node that contain subgraphs. Hovering the mouse over such nodes that have embedded subgraphs will cause the display to show the hidden subgraph. Double-clicking the node zooms to child-node level and shows the contents of the embedded subgraph as a treemap.

Clicking any node in the Treemap view highlights that node in the graph view on the canvas. If you select the default zoom factor or reset the zoom factor, clicking a node in the Treemap view zooms in and centers on the node in the graph view. The smaller the node size, the smaller the tasks executed by that node.

The Treemap view supports three different layouts for visualizing the treemap: squarified layout, alternating layout, and snake layout. All three layouts use:

- The node CPU time to determine the size of each node in the treemap
- The average concurrency observed in the graph while a node was active to determine the color

You can switch between layouts by changing the *Default Treemap Rendering* option in the *Preferences* dialog box under Preferences->GUI->Analysis View->Default Treemap Rendering Option.

Reducing Scheduler Overhead using Lightweight Policy

The Flow Graph API provides a way to apply lightweight policy for computational nodes such as function node, multifunction node, continue node and async node. Enabling lightweight policy helps reduce scheduling overhead. However it can limit parallel execution of tasks, so apply this policy on a per-node basis after careful evaluation.

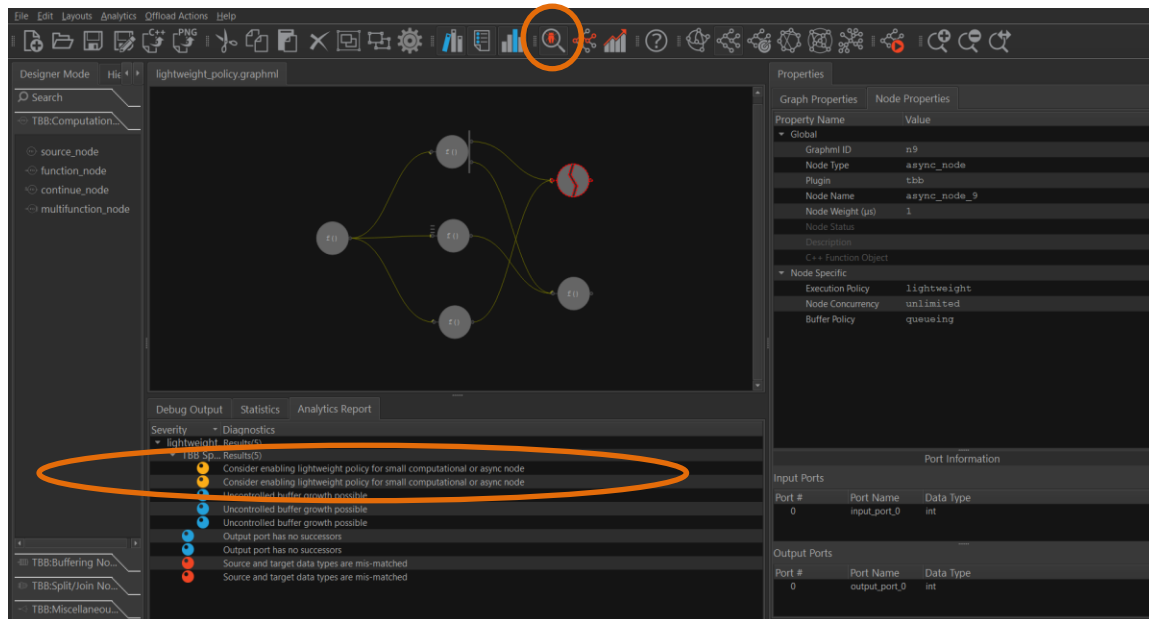
The lightweight policy indicates that the body of the node contains a small amount of work and should, if possible, be executed without the overhead of scheduling a task. By default, async node has lightweight policy enabled because they have small computation weight. By default, all other computational nodes do not have lightweight policy enabled when they are dragged and dropped into the canvas.

As a rule of thumb, the lightweight policy is applicable under the following conditions:

- Node weight is less than 1 micro-second when no trace information is available or,
- Node average time is less than 1 micro-second if the graph is loaded into context with trace data

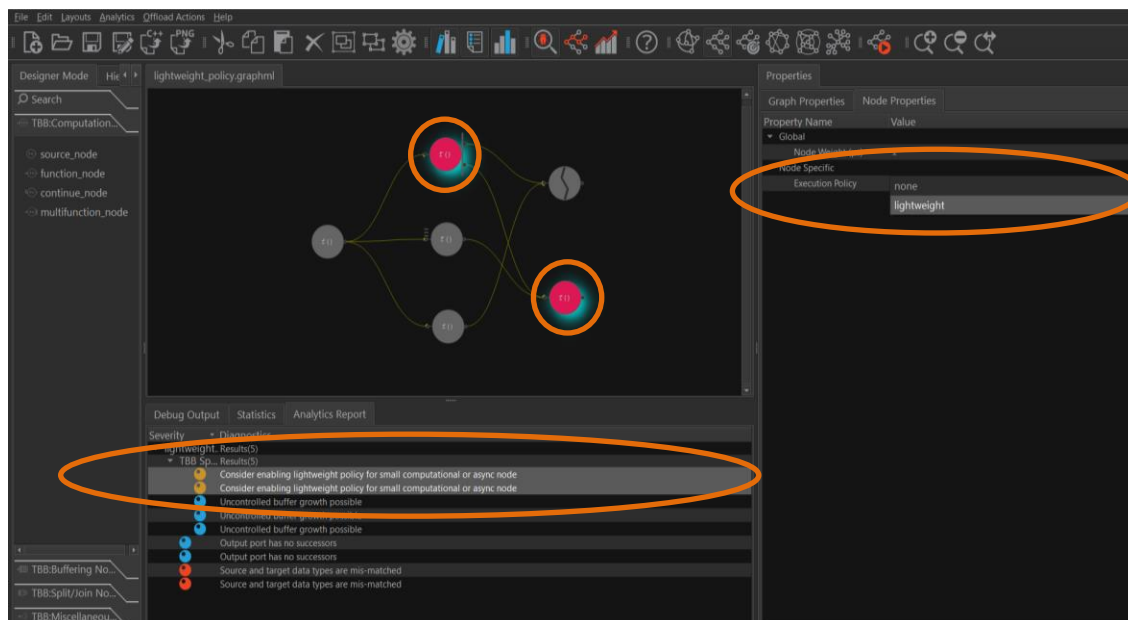
Analyzing which nodes are applicable for lightweight policy is embedded into the graph rule check to validate a graph. If the above conditions are not met but the lightweight policy is set, the graph rule check recommends removal of lightweight policy for the corresponding node.

To display recommendations for applying the lightweight policy, click the graph rule check icon on the toolbar and the *Analytics Report* tab provides you with the results, as shown in the figure below.



To set an execution policy for a single node, click the node to display the node properties on the right pane. Then set the *Execution policy* property to *none* to disable lightweight policy or to *lightweight* to enable lightweight policy.

To set a lightweight execution policy for all the nodes listed by graph rule check, multi-select the report that says *Consider enabling lightweight policy for small computational or async node* to highlight all nodes in the canvas and display the common properties for all the selected nodes. Then set the *Execution policy* property to *lightweight*.





Follow the same steps to disable lightweight policy for a node or nodes listed by graph rule check with the message *Consider disabling lightweight policy for small computational or async node* in the *Analytics Report* pane.

Another important attribute related to node execution policy is buffer policy. If you set buffer policy to *queueing* when lightweight policy is enabled, the Flow Graph Analyzer adds *queueing_lightweight* to the policy parameter of the node declaration during C++ code generation. If you set buffer policy to *rejecting*, the Flow Graph Analyzer adds *rejecting_lightweight* to the policy parameter.

By default, async node is set to *queueing_lightweight*, therefore the Flow Graph Analyzer does not add any policy during code generation for async node. Setting buffer policy to *rejecting* for async node adds *rejecting_lightweight* to the policy parameter of async node declaration during code generations.

Identifying Tasks that Operate on Common Input

Threading Building Blocks (TBB) 2018 introduced a new preview feature in U5 that adds additional meta information to trace data, such as a frame number a current task is working on. You can take advantage of this metadata to track different pipeline stages in execution, for example to identify an unbalanced pipeline. The following demonstrates how to use the user event tracing interface to enable the Flow Graph Analyzer *color-by-data* feature.

Highlighted in red is code that enables the Flow Graph Analyzer to add a unique id (frame id) to group tasks.

```
#include "tbb/flow_graph.h"
#include "tbb/tbb_profiling.h"
#include <string>
#include <vector>

int main() {
    tbb::flow::graph g;
    const int max_frames = 20;
    std::vector<tbb::profiling::event*> e;
    for(int i=0; i<nbr_of_frames;++i)
        e.push_back(new tbb::profiling::event(std::to_string(i)));
    tbb::flow::source_node<int> source( g,
        [&]( int &v) -> bool {
            static int i = 0;
            if( i < max_frames ) {
                e[i]->emit();
            }
        }
    );
}
```



```

        v = i++;

        return true;
    }

    return false;
}, false);

tbb::flow::function_node<int> foo( g, tbb::flow::unlimited,
    []( const int &input1) -> int {

        tbb::profiling::event::emit(std::to_string(input1));

        return input1;
    });

tbb::flow::function_node<int> bar( g, tbb::flow::unlimited, TBB
    []( const int &input1) -> int {

        tbb::profiling::event::emit(std::to_string(input1));

        return input1;
    });

make_edge(source, foo);
make_edge(source, bar);
source.activate();
g.wait_for_all();

return 0;
}

```

Compilation of above code differs based on the TBB version you use.

For TBB version 2019 and later, compile the code *TBB_USE_THREADING_TOOLS* macro and link against *tbb* library.

For TBB version 2018 Update 5, compile the code with *TBB_PREVIEW_FLOW_GRAPH_TRACE* and *TBB_USE_THREADING_TOOLS* macros and link against *tbb_preview* library.

Then you have two options:

- Create an event object or a collection of events upfront, where the only argument is a string (data-id) that identifies the event. Later, call the emit function of the object to tag a task with a data id
- Call a static function inline (inside a task body).



Result: The Flow Graph Analyzer groups tasks, that share the same data id and displays them in a common color:



Experimental Support for OpenMP* Applications

You can now trace, visualize, and analyze OpenMP* parallel regions, tasks, and task dependencies in your application with the Flow Graph Analyzer.

Flow Graph Analyzer support for OpenMP technology is experimental and currently covers two basic scenarios:

- OpenMP parallel regions nested inside a Threading Building Blocks (TBB) flow graph
- OpenMP tasks that use `depends` clauses

The sample code below, `omp_nested.cpp`, is an example of an OpenMP construct nested inside a TBB flow graph.

```
#include "tbb/tbb.h"
#include "tbb/flow_graph.h"
#include <omp.h>
#include <iostream>
using namespace tbb;
using namespace tbb::flow;
int main() {
    graph g;
    const int size = 20;
```




```

continue_node< continue_msg> hello( g,
    [] ( const continue_msg &) {
        std::cout << "Hello\n";
        tbb::parallel_for(0, size, 1, [=](int k) {
            std::cout << k << "\n"; });
    });
continue_node< continue_msg> world( g,
    [] ( const continue_msg &)
        std::cout << " World\n";
        #pragma omp parallel for
        for (int i=0; i<20; i++) {
            std::cout << i << "\n";
        }
    )
);
make_edge(hello, world);
hello.try_put(continue_msg());
g.wait_for_all();
return 0;
}

```

The sample code below, `omp_depend.cpp`, is a hello-world example of OpenMP task dependencies:

```

#include <omp.h>
#include<iostream>

int main() {
    #pragma omp parallel
    {
        std::string s = "";
        #prgma omp single
        {
            #pragma omp task depend( out: i)
            {
                s = "hello";
                printf("%s", s);
            }
            #pragma omp task depend( out: s )
            {
                s = "world";
                printf("%s",s);
            }
        }
    }
    return 0;
}

```

In the case of OpenMP parallel regions nested inside TBB flow graphs, the Flow Graph Analyzer shows the execution of the parallel regions in the per-thread task execution timelines. In the case of OpenMP* tasks



with `depend` clauses, the Flow Graph Analyzer not only shows task execution in the timelines, but also provides experimental support that lets you see the dependency relationships between OpenMP tasks as a graph in the graph canvas.

Collecting Traces for OpenMP* Applications

OpenMP* trace collection is currently supported only on Linux* and macOS* operating systems (see the [Software Requirements for Collector](#) section). Support for Windows* OS machines may be added in a future release.

To collect OpenMP traces for an application, you need an OMPT-enabled OpenMP* version 5.0 library, such as [LLVM-OpenMP](#).

To build the sample code shown above on a Linux* OS with an Intel compiler, use the following command:

```
icpc -std=c++11 -qopenmp example.cpp -o example
```

NOTE: Please use `-g` compiler flag to enable [symbol resolution](#) information.

To create XML files, follow these steps:

1. Update your PATH environment variable to add the paths to essential tools, as described below. Also set the FGT_ROOT variable.

OS	Environment Variable	Value	Description
Linux*	FGT_ROOT	<advisor-install-dir>/fga<version>/fgt	The path to the Flow Graph Collector installation
	PATH	<code>\${FGT_ROOT}/linux/bin:\${FGT_ROOT}/linux/bin/ia32/cc4.8_libc2.19_kernel3.13.0:\${FGT_ROOT}/linux/bin/intel64/cc4.8_libc2.19_kernel3.13.0:\${PATH}</code>	The path should include paths to <code>fgtrun.sh</code> , <code>fgtrun.csh</code> , and <code>fgt2xml.exe</code>

2. To enable tracing from OMPT, set the following:

OS	Environment Variable	Value	Description
Linux*	OMP_TOOL	Enabled	OMPT tool support enabler
	OMP_TOOL_LIBRARIES	<code>\${FGT_ROOT}/linux/lib/intel64/cc4.8_libc2.19_kernel3.13.0/libfgt.so</code>	Path to OpenMP collector library

3. Run the application.

If your paths are set up correctly, the application generates one or more files that start with `_fgt`. There is one file per thread that participates in executing the parallelism in the application. So, for



example, if two threads participate in the execution of the flow graph, running the application generates two files, `_fgt.0` and `_fgt.1`, with an autogenerated folder in the format `_fga_YYYYMMDD_HHMMSS` according to its creation (for example, 2019

1111_1111).

4. Convert the trace files to GraphML* and TraceML* format.

Convert the `_fgt` binary files to the XML format understood by the Flow Graph Analyzer using the `fgt2xml.exe` converter in the directory containing the folder with the trace files:

```
fgt2xml.exe desired_name --omp_experimental
```

or

```
fgt2xml.exe --omp_experimental desired_name
```

or

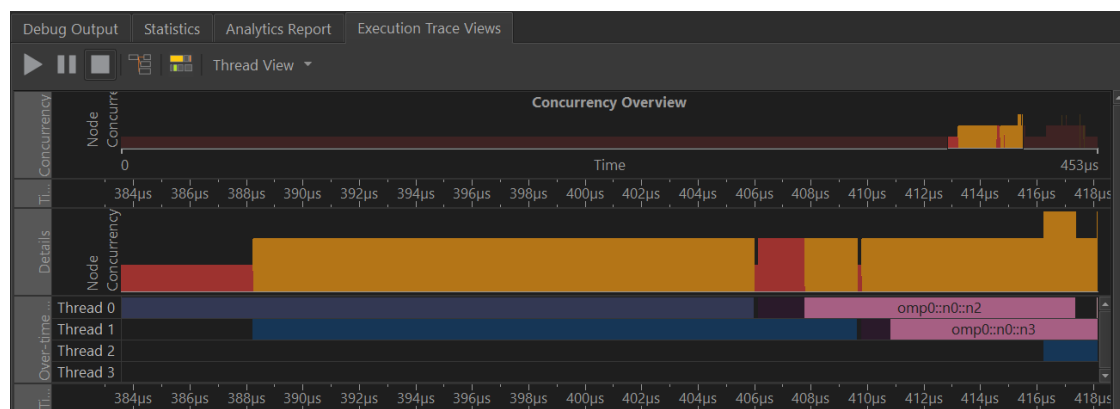
```
fgt2xml.exe --omp_experimental
```

This converter scans the current directory for all `_fgt` files within the most recent folder according to its name and generates two output files: `desired_name.graphml` and `desired_name.traceml`. If you do not provide a `desired_name`, the converter creates `unknown.graphml` and `unknown.traceml`.

The `omp_experimental` flag enables display of a subgraph and tasks dependence graph. By default, display support is disabled and you see only information related to OpenMP constructs in the per-thread traces.

OpenMP* Constructs in the Per-Thread Task View

When the steps outlined above are followed for the `omp_depend.cpp` sample described above, the execution of the OpenMP* tasks are visible in the per-thread timelines. A display example is shown below. The OpenMP region names are prefixed with *omp*.





OpenMP* Constructs in the Graph Canvas

When activated using the `--omp_experimental` flag, the `fgt2xml` converter maps OpenMP* parallel region and task constructs to a graph in which nodes represent parallel regions and tasks, and edges represent task dependencies.

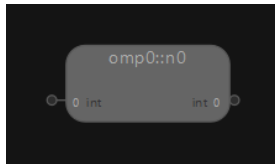
Parallel Regions

All OpenMP-related parallelism is contained within OpenMP* parallel regions. In the Flow Graph Analyzer, a parallel region is mapped to a subgraph node in the graph canvas. Inside the subgraph node are at least two nodes:

- A node that represents the start of the parallel region
- A node that represents the implicit barrier at the end of the region

For example, given an empty parallel region like the following, the Flow Graph Analyzer creates a subgraph node, such as `omp0::n0`, in the graph canvas.

```
#pragma omp parallel
{
}
```



When you double-click the subgraph node, you see the following, where `omp0::n0::n1` is the start of the parallel region and `omp0::n0::n2` is the implicit barrier at the end of the node.



OpenMP* Tasks

An OpenMP* task is a block of code contained in a parallel region that can be executed simultaneously with other tasks in the same region. In the Flow Graph Analyzer, an OpenMP task is mapped to a generic node. For example, in the code below, there are two tasks: one prints “hello” and the other prints “world”. The order in which these tasks execute is not specified, so they might execute in any order; however, the two tasks always start after the enclosing parallel region begins, and they complete before the enclosing parallel region ends.

```
#pragma omp parallel
{
    #pragma omp task
```

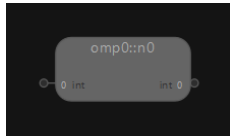


```

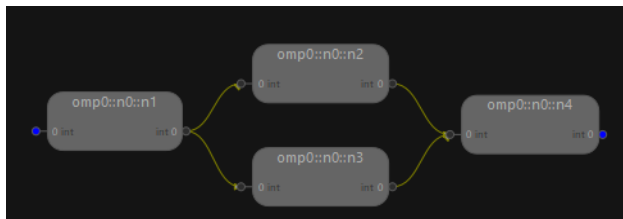
    { printf("hello "); }
#pragma omp task
    { printf("world "); }
}

```

When you visualize this program in the Flow Graph Analyzer, it looks like this:



When you double-click this subgraph, you see the following, where `omp0::n0::n1` is the start of the parallel region, `omp0::n0::n4` is the implicit barrier at the end of the region, `omp0::n0::n2` is the “hello” task and `omp0::n0::n3` is the “world” task.



OpenMP* Task Dependencies

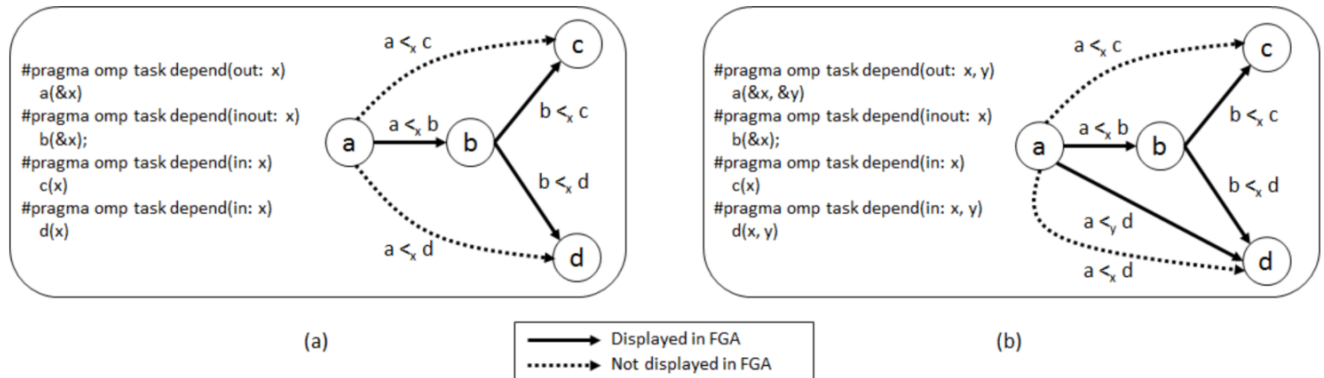
In the OpenMP* specification, a partial ordering of tasks can be expressed with `depend` clauses. The task dependence is fulfilled when the predecessor task completes. There are three dependence types support by the OpenMP API: *in*, *out*, and *in-out*:

- **in** dependency-type: The generated task is a dependent task of all previously generated sibling tasks that reference at least one of the list items in an *out* or *in-out* clause.
- **out** and **in-out** dependency-type: The generated task is a dependent task of all previously generated sibling tasks that reference at least one of the list items in an *in*, *out*, or *in-out* clause.

In the Flow Graph Analyzer, task dependencies are represented by edges between the nodes that represent OpenMP tasks.

It is important to understand what dependencies are visualized in the Flow Graph Analyzer. The task dependency graph represents the partial order imposed by the `depend` clauses for the set of OpenMP tasks executed by the application, with the nodes representing OpenMP tasks and the edges representing the partial order.

To reduce the complexity of the graph, the Flow Graph Analyzer omits some **transitive** dependencies for a specific variable as shown in the figure below. A transitive dependence is a dependence between three tasks, such that if it holds between the first and second tasks, and it also holds between the second and third tasks, it must necessarily hold between the first and third tasks. In the figure below, we denote that a node **a** must execute before a node **b** in the partial order due to a dependence on location *x* as $a <_x b$.



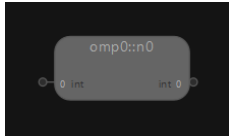
In part (a) of the figure above, we can see an example that only includes dependencies due to a single location x . Because $a \leq_x b$ and $b \leq_x d$, the Flow Graph Analyzer does not show the **transitive** edge $a \leq_x d$. In part (b) of the figure, there are two locations involved in determining the partial order, x and y . In this case there are two potential dependence edges from a to d : $a \leq_x d$ and $a \leq_y d$. The Flow Graph Analyzer includes an edge from a to d because a is the direct source of y for d ; however, it excludes $a \leq_x d$.

NOTE: If there are parallel edges between two nodes and at least one of them can be omitted due to transitivity, they all can be omitted without changing the partial order. Even so, the Flow Graph Analyzer includes edges like $a \leq_y d$ in the graph-topology because including edges to satisfy all required data dependencies is the most natural representation.

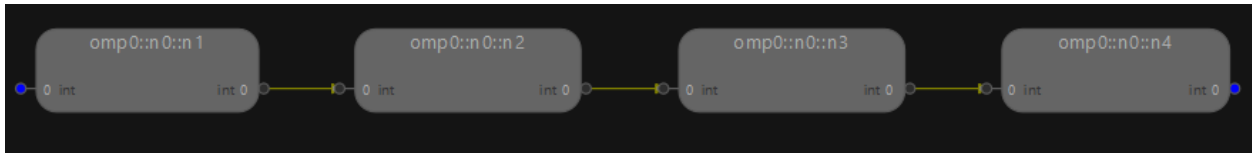
For example:

```
#pragma omp parallel
{
    std::string s = "";
    #pragma omp single
    {
        #pragma omp task depend( out: s )
        {
            s = "hello";
            printf("%s", s);
        }
        #pragma omp task depend( out: s )
        {
            s = "world";
            printf("%s",s);
        }
    }
}
```

This application, when visualized with the Flow Graph Analyzer, has a single top-level subgraph node representing the OpenMP parallel region.



When you double-click this subgraph, you see the following:



The edge between `omp0::n0::n2` and `omp0::n0::n3` represents task dependency due to variable `s`.

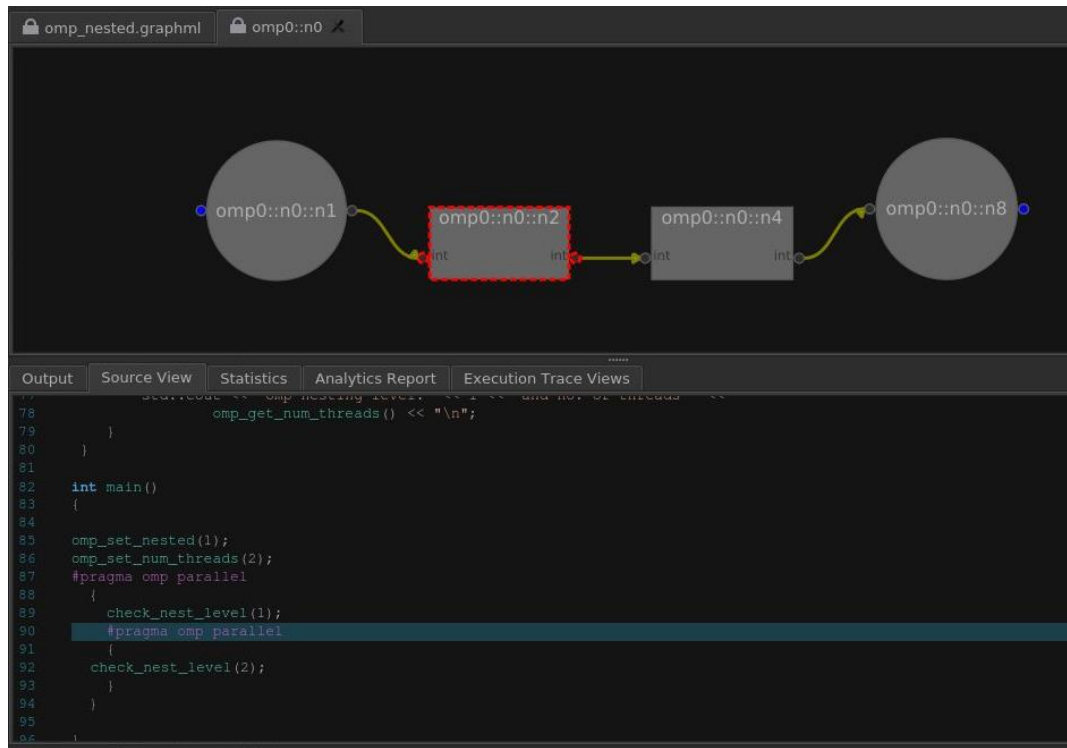
The main components of the Flow Graph Analyzer include the tree-map view, the graph-topology canvas, the timeline and concurrency histogram view, and the critical-path report. OpenMP task traces map naturally to these views:

- The tree-map view shows the time spent in each OpenMP parallel region, colored according to the average application concurrency during the time it was executing.
- The graph topology canvas shows the partial ordering of the tasks.
- The timeline and concurrency histogram view show the execution of each task on the OpenMP runtime threads and the application concurrency over time.
- The critical report shows the most time-consuming path from each source to each sink in the graph, sorted with the longest critical path at the top.

For more examples, see https://link.springer.com/chapter/10.1007/978-3-319-98521-3_12.

OpenMP Nodes to Source Code Mapping:*

Not only FGA provides graphical view of OpenMP Task dependency graphs, but it also provides, [nodes mapping to corresponding source code](#). In order to get this information, user must be build openmp application with `-g` flag.



The preceding screenshot shows source code mapping with subgraph nodes(parallel region)



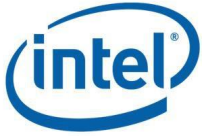
Sample Trace Files

Included with the Flow Graph Analyzer are five sets of sample traces you can explore to become familiar with the features of the tool. These traces are in the `samples` directory. Whenever you launch the Flow Graph Analyzer, the *File* dialog box defaults to the `samples` directory.

The `samples` subdirectories contain samples you can load. The samples are described in the table below and explained in more detail in the sections that follow.

Location	XML files	Notes
samples/code_generation	dining_like.graphml	Generate a Dining Philosophers sample.
	feature_like.graphml	Generate a Features Detection sample.
samples/performance_analysis	feature_detection.graphml feature_detection.traceml	Provides a runtime trace of a feature detection sample.
	forward_substitution.graphml forward_substitution.traceml	Provides a runtime trace of a forward substitution sample.
	computer_vision.graphml computer_vision.traceml	Provides a runtime trace of a computer vision sample.

NOTE: The `performance_analysis` samples were captured by runtime tracing. Because runtime tracing cannot infer all types and cannot capture the user bodies of nodes, these samples do not contain complete descriptions of applications and therefore cannot be used to regenerate the applications. While C++ code can be generated from these samples, the resulting code will be incomplete and will need modification before compilation and execution. In contrast, the `code_generation` samples were completely described from within Flow Graph Analyzer. When the `code_generation` samples are used to generate C++ code, no modifications are necessary before compilation and execution.



code_generation Samples

Dining Philosophers

The `dining_like.graphml` sample provides a complete description of a Threading Building Blocks (TBB) flow graph application that implements a version of the dining philosophers problem. You can generate a complete TBB flow graph by loading this file in to Flow Graph Analyzer and then following the instructions provided in the [Generating C++ Stubs](#) section.

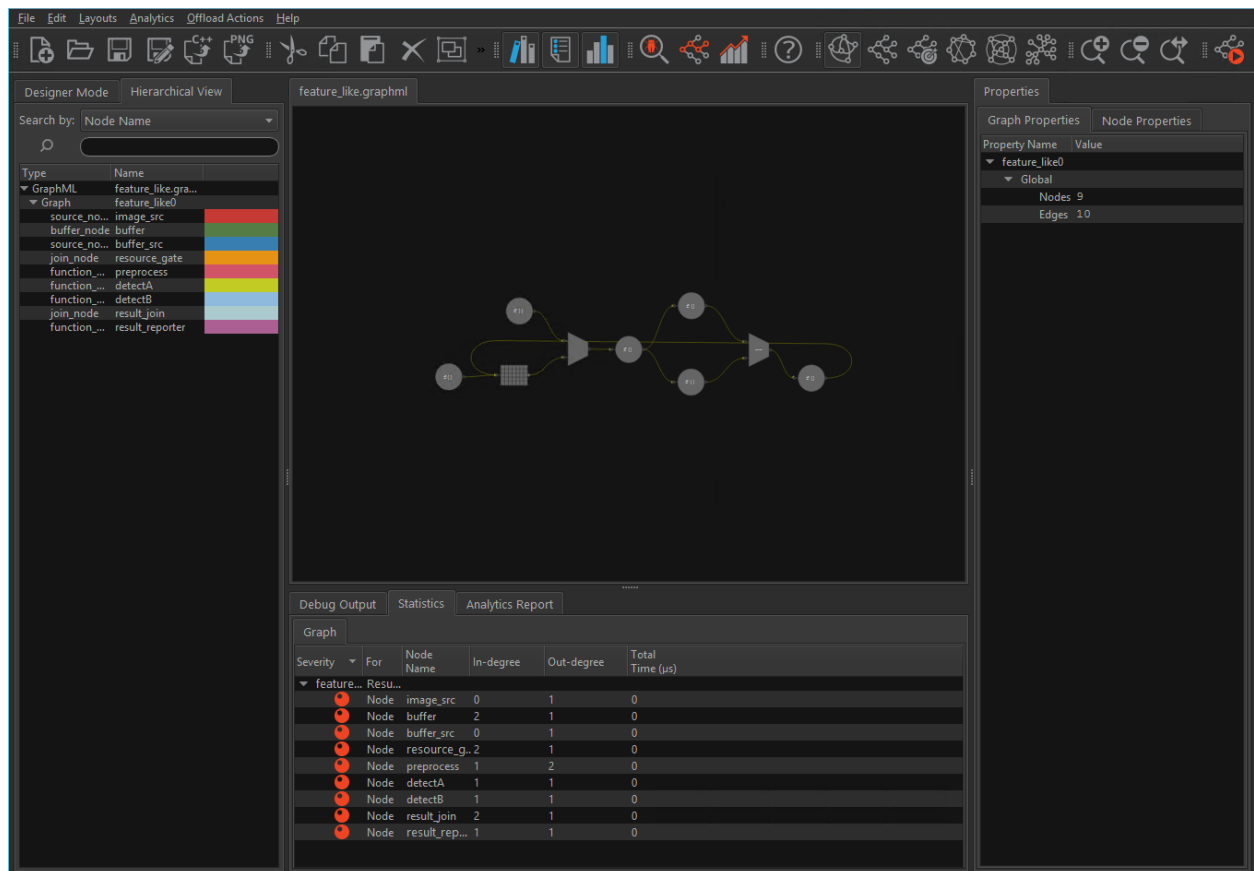
The screenshot displays the Flow Graph Analyzer Gold 1.1.17775 interface. The main window shows a graph titled `dining_like.graphml`. The left sidebar contains a search bar and a list of nodes categorized by type (GraphML, Graph, function, buffer, multifunc, join, source). The central area shows a complex flow graph with nodes and edges. The right sidebar shows the Properties panel, which is divided into Graph Properties and Node Properties. The Node Properties section shows details for the selected node, including its name, type, and various properties.

Severity	For	Node Name	In-degree	Out-degree	Total Time (µs)
Warning	dining...	Node t0	2	1	0
Warning	dining...	Node d0	1	1	0
Warning	dining...	Node c4	3	2	0
Warning	dining...	Node e0	1	3	0
Warning	dining...	Node d1	1	1	0
Warning	dining...	Node t1	2	1	0
Warning	dining...	Node e1	1	3	0
Warning	dining...	Node c0	3	2	0



Feature Detection

The `feature_like.graphml` sample provides a complete description of a feature detection application based on the example described in the blog posting at <https://software.intel.com/en-us/blogs/2011/09/09/a-feature-detection-example-using-the-intel-threading-building-blocks-flow-graph>. You can generate a complete Threading Building Blocks (TBB) flow graph by loading this file in to Flow Graph Analyzer and then following the instructions provided in the [Generating C++ Stubs](#) section.

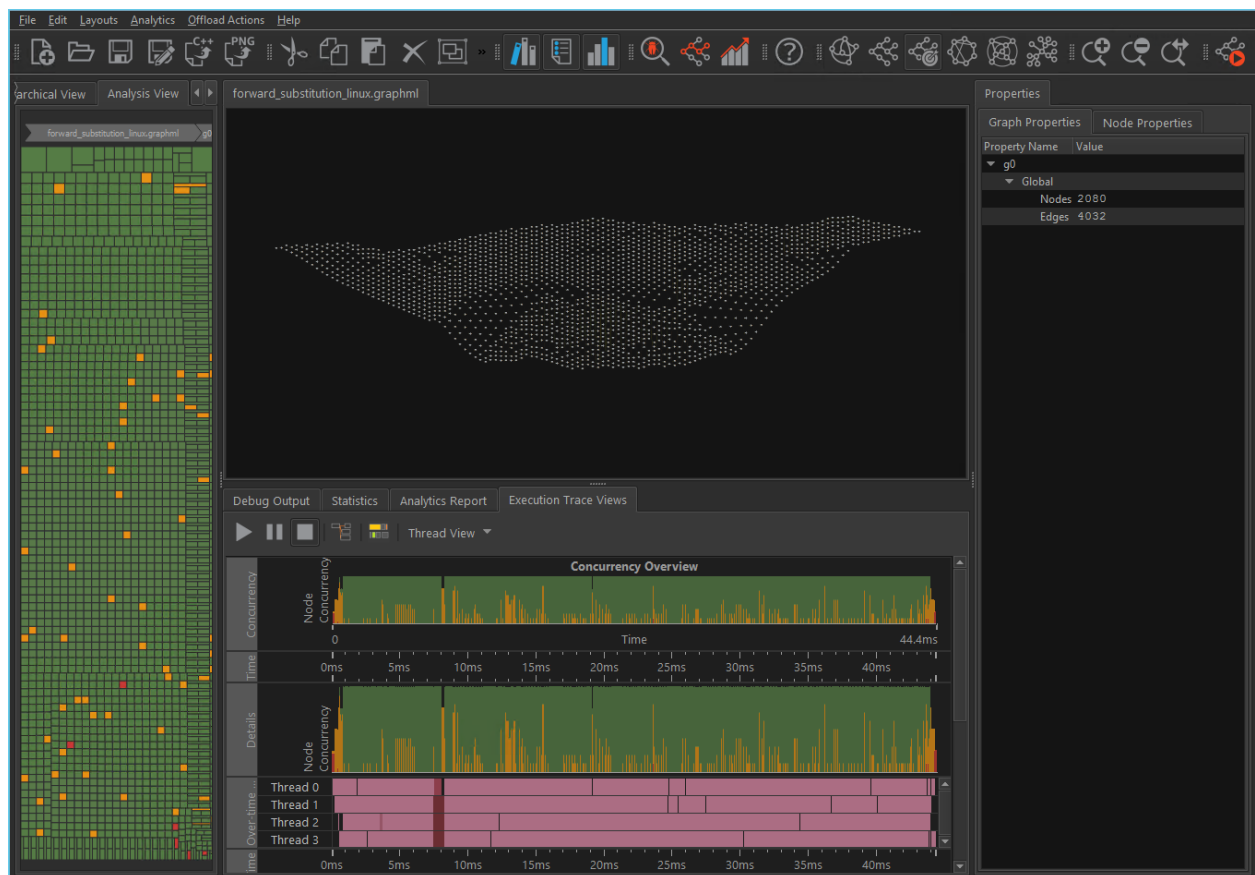




performance_analysis Samples

Forward Substitution with Trace

The `forward_substitution.graphml` sample shows the topology and behavior of a Threading Building Block (TBB) flow graph application that provides an implementation of forward substitution on a lower-triangular matrix. The trace is for a single execution of the graph, using 4 threads for a 8192x8192 matrix with a block size of 128. The runtime trace of the application is contained in the matching `forward_substitution.traceml` file. This matching file is loaded automatically by Flow Graph Analyzer.

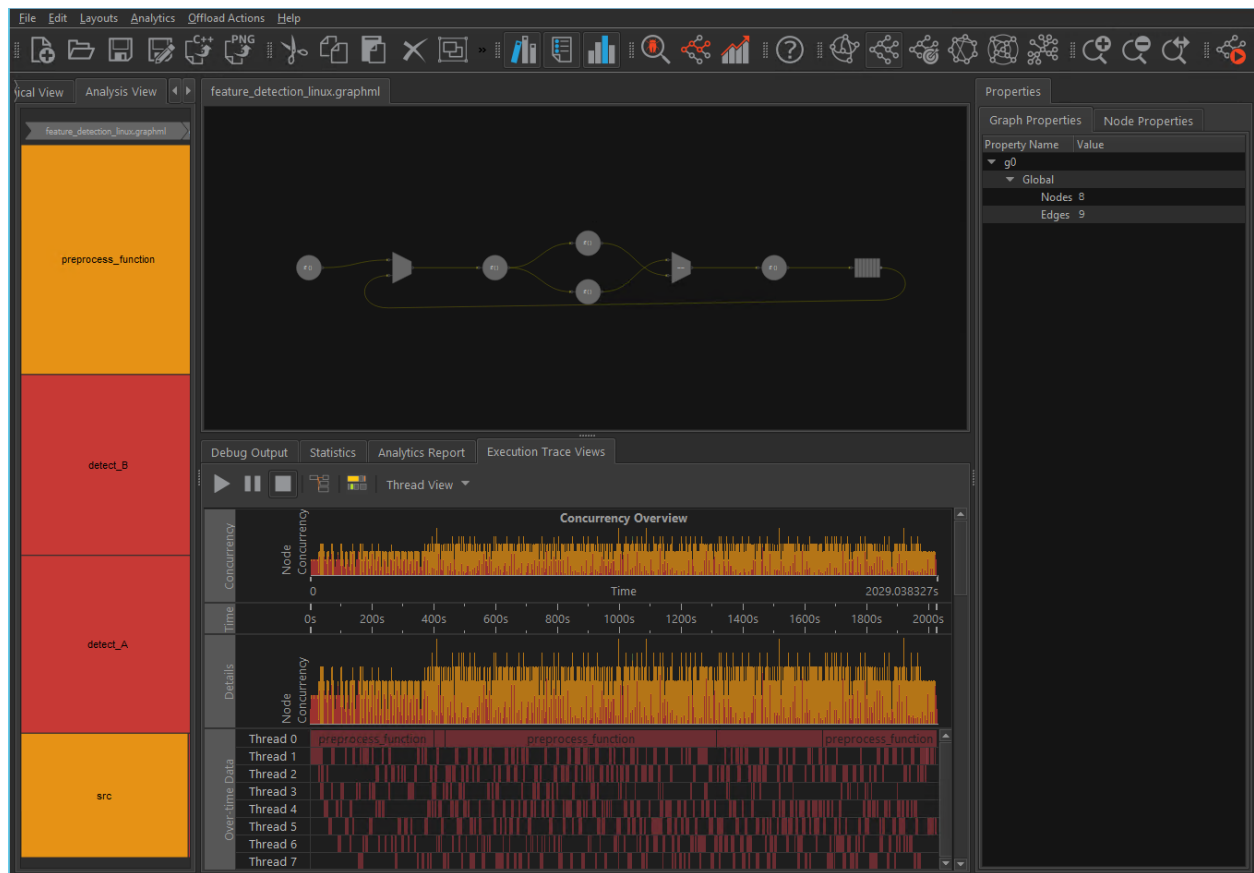




Feature Detection with Trace

The `feature_detection.graphml` sample shows the topology and behavior of a Threading Building Blocks (TBB) flow graph application based on the example described in the blog posting at <https://software.intel.com/en-us/blogs/2011/09/09/a-feature-detection-example-using-the-intel-threading-building-blocks-flow-graph>.

This trace was collected using 8 threads and 32 buffers provided to the buffer queue. The concurrency varies over time, but is limited to at most 8 threads.





Computer Vision with Trace

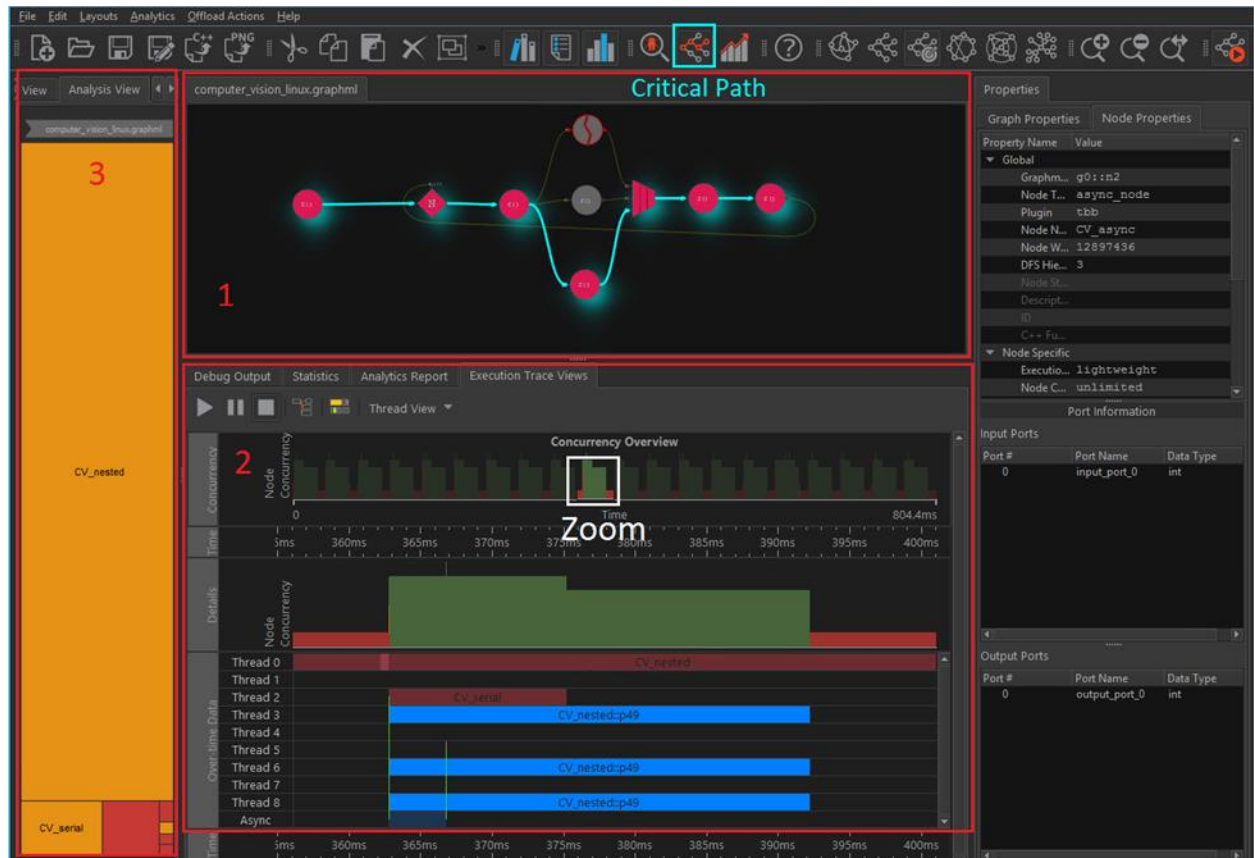
The `computer_vision.graphml` sample shows the topology and behavior of a Threading Building Blocks (TBB) flow graph application that represents a classic example of expressing data flow parallelism. It is composed of three different computer vision (CV) algorithms that process the same input data. The data is a video input stream, and you can observe a resulting regular pattern in the timeline chart (the trace contains around 20 frames).

You can analyze a single frame execution in detail by zooming in the timeline (white box within red-outlined area #2 below). Each frame processing begins with a task spawned by the source node, which is the first stage of the image processing pipeline. Next, a limiter node is used to balance the pipeline; it forwards the frame only if the number of frames that are currently executed is below a user-specified threshold. The next stage is the algorithm part, where three different algorithms are executed in parallel; concurrency changes during the algorithm stage because less work is available. In the timeline, a high concurrency is colored in green.

For a TBB flow graph, an external activity can be encapsulated in a predefined `async` node. This activity represents offloading work to an Accelerator (FPGA, GPU, ...). The beginning and end of this activity are displayed as green vertical lines in the timeline (lower part of red-outlined area #2 below). You can find a single execution within a single frame for each CV algorithm (represented by the nodes `CV serial`, `CV nested`, `CV async`). `CV nested` represents a node with a nested TBB parallel for algorithm that consumes most of the CPU time on average.

The Treemap (red-outlined area #3) shows the average node weight. Here, `CV nested` includes a TBB parallel for algorithm and consumes most of the CPU time.

You can use the critical path calculation functionality (turquoise box) to identify bottlenecks in the data flow. As a result of this feature, all nodes on the critical path are highlighted (red-outlined area #1)





Release Notes and Known Issues

August, 2019

- Triggering the Fruchterman-Reingold layout on macOS* will not launch the progress bar. The computation however proceeds and FGA will refresh and display the layout once the computation finishes.
- Symbol resolution is only supported at the graph level in TBB and with at least version 19 of the Intel® C/C++ Compiler.

October, 2018

- Flow Graph Analyzer 2019 Update 1 includes a new experimental feature that supports OpenMP* task dependencies using the OMPT interface support offered by the Intel® compiler OpenMP* implementation. The OMPT interface support is currently available in the Linux* and macOS* versions of the compiler runtimes only, thereby limiting the support for data collection of OpenMP* applications to these OSes. The ability to visualize task dependency graphs in Flow Graph Analyzer is disabled by default and will have to be explicitly enable using the `--omp-experimental` flag as described in the Collecting Traces for OpenMP* Applications section.
- When graphs are nested, whether they are TBB flow graphs or OpenMP* dependency graphs, the nested graphs are shown as peer graphs in the graph canvas.

August, 2018

- Flow Graph Analyzer 2019 includes a TBB 2019 shared library in the root directory. To ensure the correct execution of Flow Graph Analyzer, the path of this shared library appears first in the shared library search order. Trace collection performed from within the Flow Graph Analyzer GUI may load this DLL even if a user's environment is set up to use a different version of TBB. If an application is compiled against a TBB library version that is newer than TBB 2019, this can lead to failures in running that application during trace collection. This issue does not affect applications that are linked against a preview version of the TBB library, since Flow Graph Analyzer does not include a preview version of the TBB shared library.

May, 2018

- Large graphs display canvas that might appear blank when initially loaded on Flow Graph Analyzer.

If you load a GraphML* file with large graphs (graphs with more than 10,000 nodes and edges), the initial display showing the entire graph might appear blank.

To fix: Click one of the nodes displayed in *Hierarchical View* or *Statistics* or *Treemap View* tab.

- Code generation creates invalid names if file names, node names, etc., invalidate C++ naming conventions.



If you create a GraphML* file with a numeric name, such as 123.graphml, or give a node a numeric name and then generate a C++ file using Flow Graph Analyzer code generation functionality, the file will not compile

To fix: Follow C++ naming conventions.

October, 2017

- Composite_node generated code fails to compile.

If you generate code that contains a subgraph, that is a `tbb::flow::composite_node`, and then compile it with `TBB_PREVIEW_FLOW_GRAPH_TRACE` defined, the compilation fails. The `set_name` calls in the generated code have an incorrect argument.

There is a plan to fix this error in a future release. To work around: Remove the offending calls to `set_name` or fix the arguments.

- Composite_nodes that contain async_nodes result in corrupted trace files.

This issue was fixed and released in May, 2018. ~~If an application contains a composite_node that encapsulates an async_node, the trace file that is generated during execution will fail to load in Flow Graph Analyzer.~~

September, 2017

- End of support for Visual Studio* 2012 in the Threading Building Blocks (TBB) library

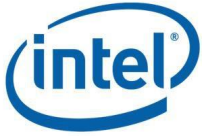
Some of the examples used in this document refer to Visual Studio* 2012. The latest releases of Threading Building Blocks (TBB) no longer support Visual Studio* 2012. If you are using a recent release of TBB, you need a later version of Visual Studio* software to work through these examples.

- Nested opencl_node port issues

There is a known issue in Threading Building Blocks (TBB) that results in compilation failures for `opencl_nodes` that are directly nested in a `composite_node`. If you group nodes into a subgraph in Flow Graph Analyzer and expose an input or output port of an `opencl_node` as an input or output port of the `composite_node`, the C++ code generated by Flow Graph Analyzer may fail to compile when using current versions of TBB.

Additional Resources

- Flow Graph Analyzer is released as a feature of Intel® Advisor. You can find out more information about Intel® Advisor and instructions on how to obtain it and its components at <https://software.intel.com/en-us/intel-advisor-xe>
- You can download or find out more about the Threading Building Blocks (TBB) library by visiting <http://software.intel.com/en-us/intel-tbb> or <https://www.threadingbuildingblocks.org/>
- You can download or find out more about Intel® Parallel Studio XE at <https://software.intel.com/en-us/intel-parallel-studio-xe>



Legal Information

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

The products and services described may contain defects or errors known as errata which may cause deviations from published specifications. Current characterized errata are available on request.

Intel and the Intel logo are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries.

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

Microsoft, Windows, and the Windows logo are trademarks, or registered trademarks of Microsoft Corporation in the United States and/or other countries.

Copyright © 2014-2019 Intel Corporation.

This software and the related documents are Intel copyrighted materials, and your use of them is governed by the express license under which they were provided to you (**License**). Unless the License provides otherwise, you may not use, modify, copy, publish, distribute, disclose or transmit this software or the related documents without Intel's prior written permission.

This software and the related documents are provided as is, with no express or implied warranties, other than those that are expressly stated in the License.

Flow Graph Analyzer ships Qt*(version 5.12.2) libraries licensed under the GNU Lesser Public License (LGPL) or Runtime General Public License. Their source code can be downloaded from <https://download.01.org/qt/source/>

Flow Graph Analyzer ships Boost* C++ libraries licensed under the Boost Software License. Their source code can be downloaded from http://www.boost.org/users/history/version_1_62_0.html