OpenMP* and Threading Building Blocks Task Graphs: unraveling the spaghetti with Intel® Advisor – Flow Graph Analyzer

Vasanth Tovinkere | Architect, Flow Graph Analyzer

Intel® Corporation

What will be covered today

Task-based parallelism and task graphs

- Challenges

Overview of Intel® Advisor - Flow Graph Analyzer (FGA)

Walking through a sample

Summary

Task-based parallelism and task graphs

Task-based parallelism

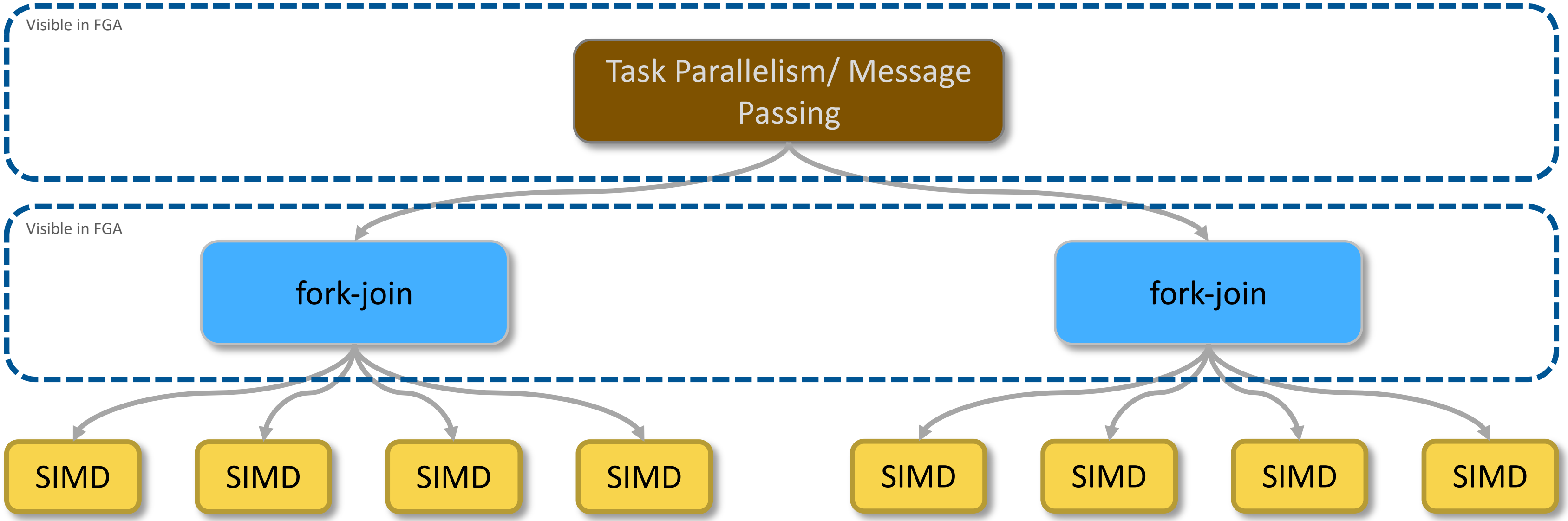
Advantages of task-based parallelism

- Makes parallelization efficient for irregular and runtime dependent execution
- Promotes higher level thinking
- Improves load balancing

Tasks with dependencies

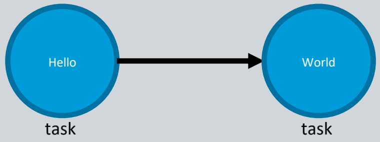
- Fall into two categories: explicit and implicit
- Extends the expressiveness of task-based parallel programming
- Reduces need for global synchronization mechanism such as task barriers

Applications often contain multiple levels of parallelism



Asynchronous task graphs (implicit vs. explicit)

OpenMP*

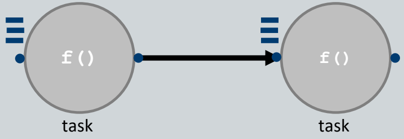



```
#pragma omp parallel
{
   #pragma omp single
    {
       std::string s;
       {
          #pragma omp task depend(out: s)
          {
               s = “Hello ”;
               cout << s;
          }
          #pragma omp task depend(out: s)
          {
               s = “World!\n”;
               cout << s;
          }
       }
    }
}
```

Implicit

Implicit dependency derived from the depend clause, in this case the variable **‘s’**

Threading Building Blocks (TBB)




```
graph g;
continue_node<continue_msg> h( g,
        []( continue_msg & ) {
             cout << “Hello “;
        } );

continue_node<continue_msg> w( g,
        []( continue_msg & ) {
             cout << “World!\n“;
        } );

make_edge(h, w);
h.try_put(continue_msg());
g.wait_for_all();
```

Explicit

Explicit dependency expressed through the `make_edge()` call

Challenges with asynchronous task graphs

Creating implicit or explicit task graphs programmatically is easy

- Determining what was created is hard in many cases

New programming paradigm

Allows you to stream data through the graph, which makes debugging challenging

Graph algorithms can be latency-bound or throughput-bound

Parallelism is unstructured in certain types of graphs, so performance analysis can be challenging

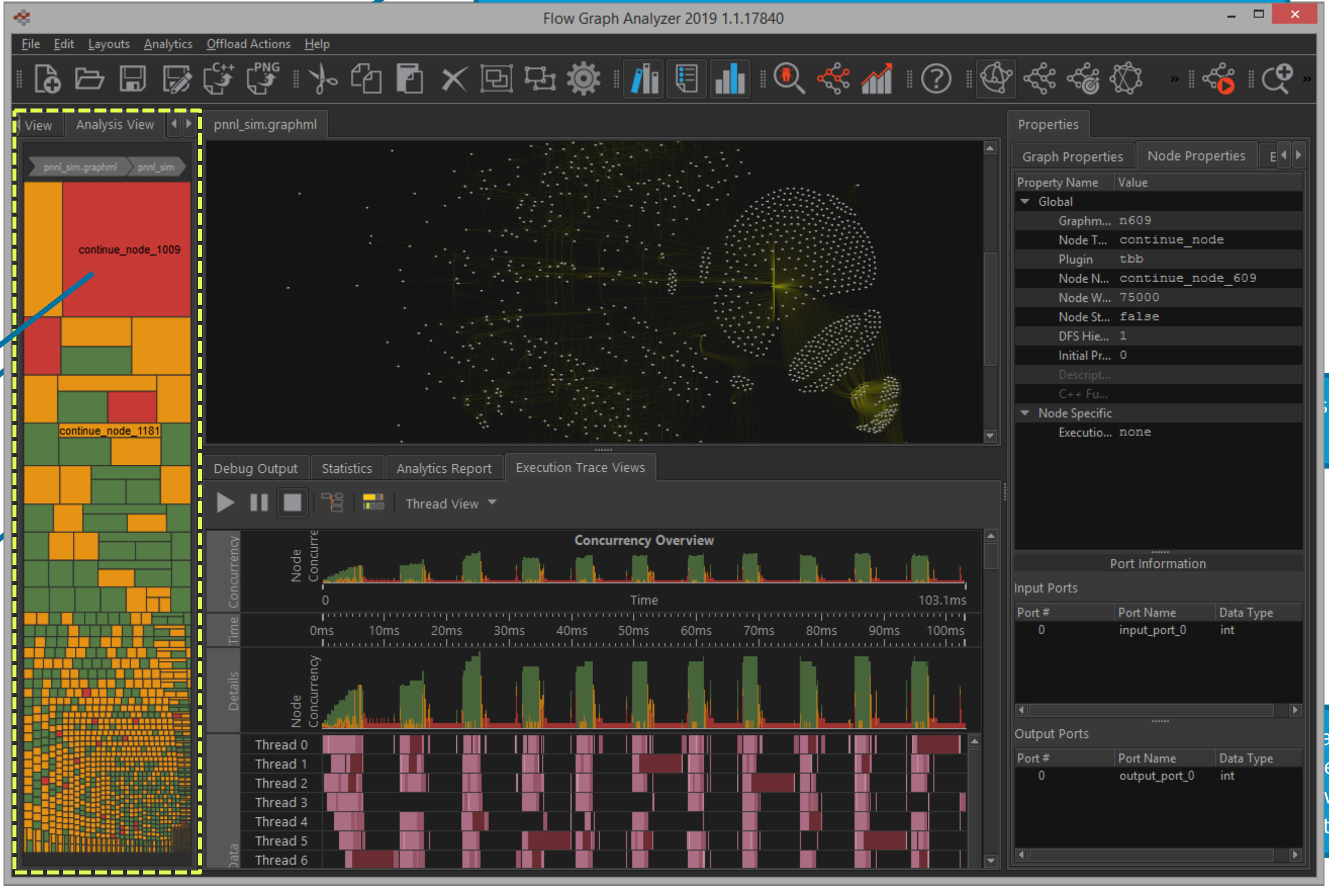
Overview of intel® Advisor – flow graph analyzer (FGA)

Intel® Advisor – Flow Graph Analyzer

Toolbar supporting basic file and edition operations, visualization and analytics

General health of the graph displayed as a tree-map

The area of the squares represent the CPU time taken by a node as a percentage of the application run and the color indicates the concurrency observed when that node was active



s for visualizing graphs

ace data, graph enerated by ws interactions ta

Overview of intel® Advisor – flow graph analyzer

Workflows and UI features

Workflows: Create, Debug, Visualize and Analyze

Design mode

- Allows you to create a graph topology interactively
- Validate the graph and explore what-if scenarios
- Add C/C++ code to the node body
- Export C++ code using Threading Building Blocks (TBB) flow graph API

Analysis mode

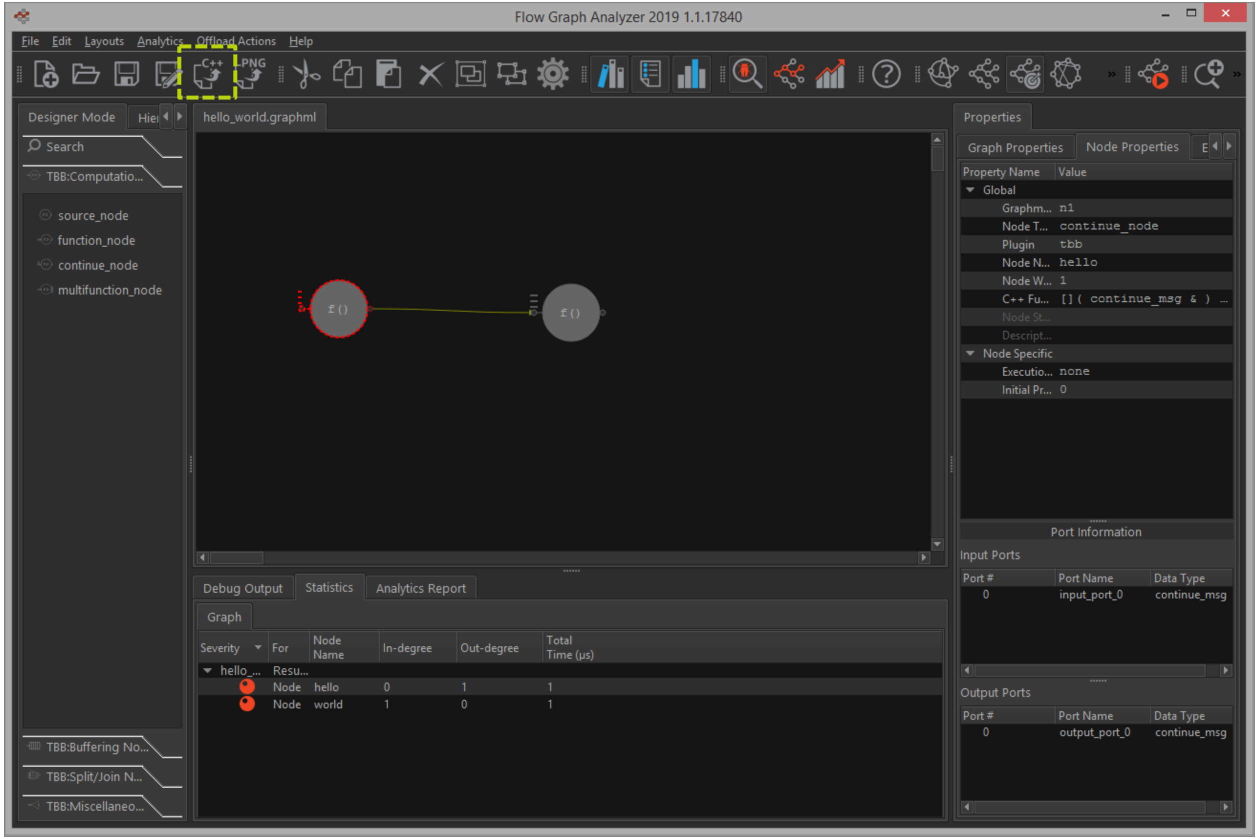
- Compile your application (with tracing enabled)
- Capture execution traces during the application run
- Visualize/analyze in Flow Graph Analyzer

Overview of intel® Advisor – flow graph analyzer

Creating Asynchronous Task-graphs

Intel® Advisor – Flow Graph Analyzer (Design mode)

Graph Creation



Drag and Drop Support

Interactive Canvas

Analytics and Modeling

Validation

Code Generation

Intel® Advisor – Flow Graph Analyzer (Design mode)

Serialization

GraphML* file format – uses extensions

hello_world.graphml

```
  <key id="d31" namespace="all::node::tbb::streaming_node" for="node" attr.name="opencl_ndrang
  <key id="d32" namespace="all::node::fga::subgraph_node" for="node" attr.name="subgraph_refer
  <key id="d33" namespace="all::edge" for="edge" attr.name="graphml_id" attr.type="string"/>
  <key id="d34" namespace="all::edge" for="edge" attr.name="edge_name" attr.type="string"/>
  <graph id="hello_world_g0" edgedefault="directed">
    <data key="d1">2</data>
    <data key="d2">1</data>
    <node id="n1">
      <data key="d3">continue_node</data>
      <data key="d16">none</data>
      <data key="d5">tbb</data>
      <data key="d7">1</data>
      <data key="d19">0</data>
      <data key="d6">hello</data>
      <data key="d12">[]( continue_msg &amp; ) {     cout &lt;&lt; “Hello “; }</data>
      <port data_type="continue_msg" name="input_port_0" type="input" offset="0"/>
      <port data_type="continue_msg" name="output_port_0" type="output" offset="0"/>
    </node>
    <node id="n2">
      <data key="d3">continue_node</data>
      <data key="d16">none</data>
      <data key="d5">tbb</data>
      <data key="d7">1</data>
      <data key="d19">0</data>
      <data key="d6">world</data>
      <data key="d12">[]( continue_msg &amp; ) {     cout &lt;&lt; “World!\n“; } </data>
      <port data_type="continue_msg" name="input_port_0" type="input" offset="0"/>
      <port data_type="continue_msg" name="output_port_0" type="output" offset="0"/>
    </node>
    <edge id="e3" target="n2" source="n1" targetoffset="0" targetport="input_port_0" source
      <data key="d34">edge_3</data>
    </edge>
  </graph>
</graphml>
```

C/C++ code generated from the graph

hello_world.graphml | hello_world_stubs.cpp

```cpp
int build_and_run_hello_world_g0() {
    graph hello_world_g0;
    continue_node< continue_msg > hello( hello_world_g0, 0,[]( continue_msg & ) { cout << "Hello ";
    continue_node< continue_msg > world( hello_world_g0, 0,[]( continue_msg & ) { cout << "World!\r

    #if TBB_PREVIEW_FLOW_GRAPH_TRACE
    hello_world_g0.set_name("hello_world_g0");
    hello.set_name("hello");
    world.set_name("world");
    #endif

    make_edge( hello, world );
    hello_world_g0.wait_for_all();
    return 0;
}

int main(int argc, char *argv[]) {
    return build_and_run_hello_world_g0();
}
```

Challenges With asynchronous task graphs

Creating implicit or explicit task-graphs programmatically is easy

- Determining what was created is hard in many cases

✓ New programming paradigm

Allows you to stream data through the graph, which makes debugging challenging

Graph algorithms can be latency-bound or throughput-bound

Parallelism is unstructured in certain types of graphs, so performance analysis can be challenging

Intel® Advisor – Flow Graph Analyzer (Design mode)

Compiling and collecting traces

Path must be updated so fgtrun.bat and fgt2xml.exe can be run from the command line

```
>cl hello_world.cpp /O2 /DTBB_USE_THREADING_TOOLS ... /link tbb.lib /OUT:hello_world.exe
```

```
>set FGT_ROOT=<installation-directory>\fga\fgt
```

```
>set INTEL_LIBITTNOTIFY64=<installation-directory>\fga\fgt\windows\bin\intel64\<vc-version>\fgt.dll
```

```
>hello_world.exe
```

Traces are saved to a unique directory _fgt_<date>_<time>

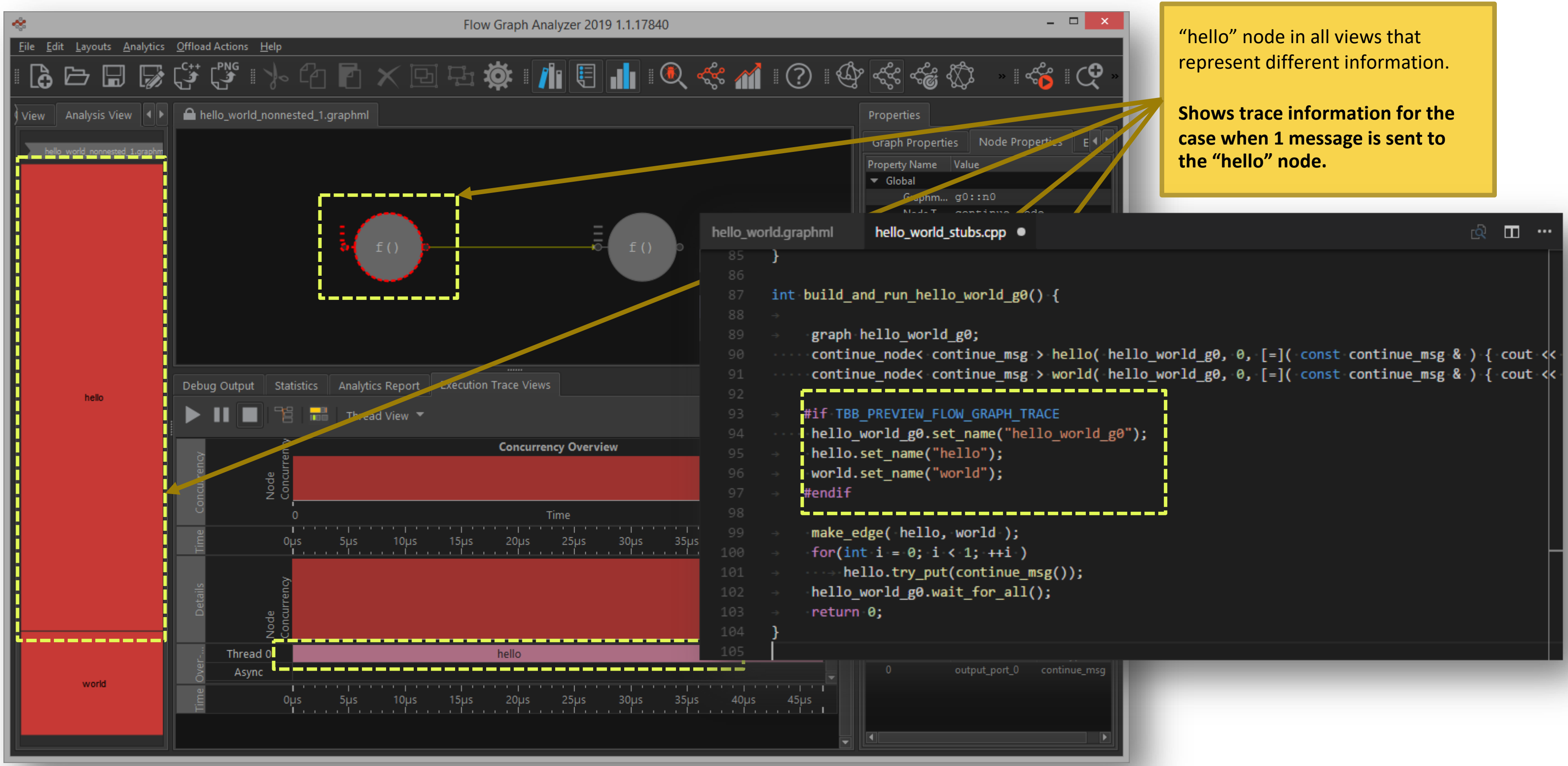
```
>fgt2xml.exe <name-for-the-trace-data-file>
```

Automatically converts the latest timestamped directory

Overview of intel® Advisor – flow graph analyzer

Understanding Graph Execution

Examining the trace data: what's possible?

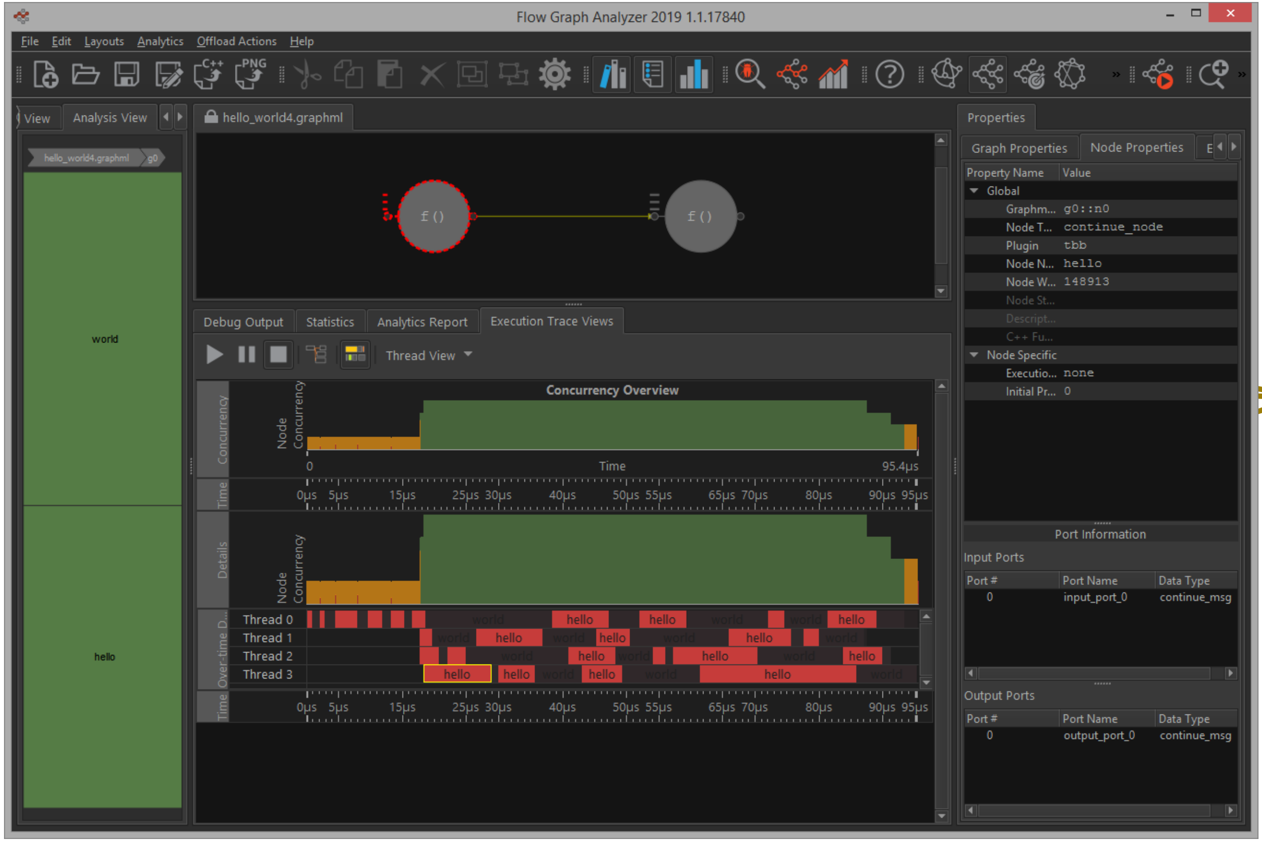


"hello" node in all views that represent different information.

Shows trace information for the case when 1 message is sent to the "hello" node.

Optimization Notice

Examining the trace data: correlation



"hello" node in all views that represent different information.

Shows trace information for the case when 25 messages are sent to the "hello" node.

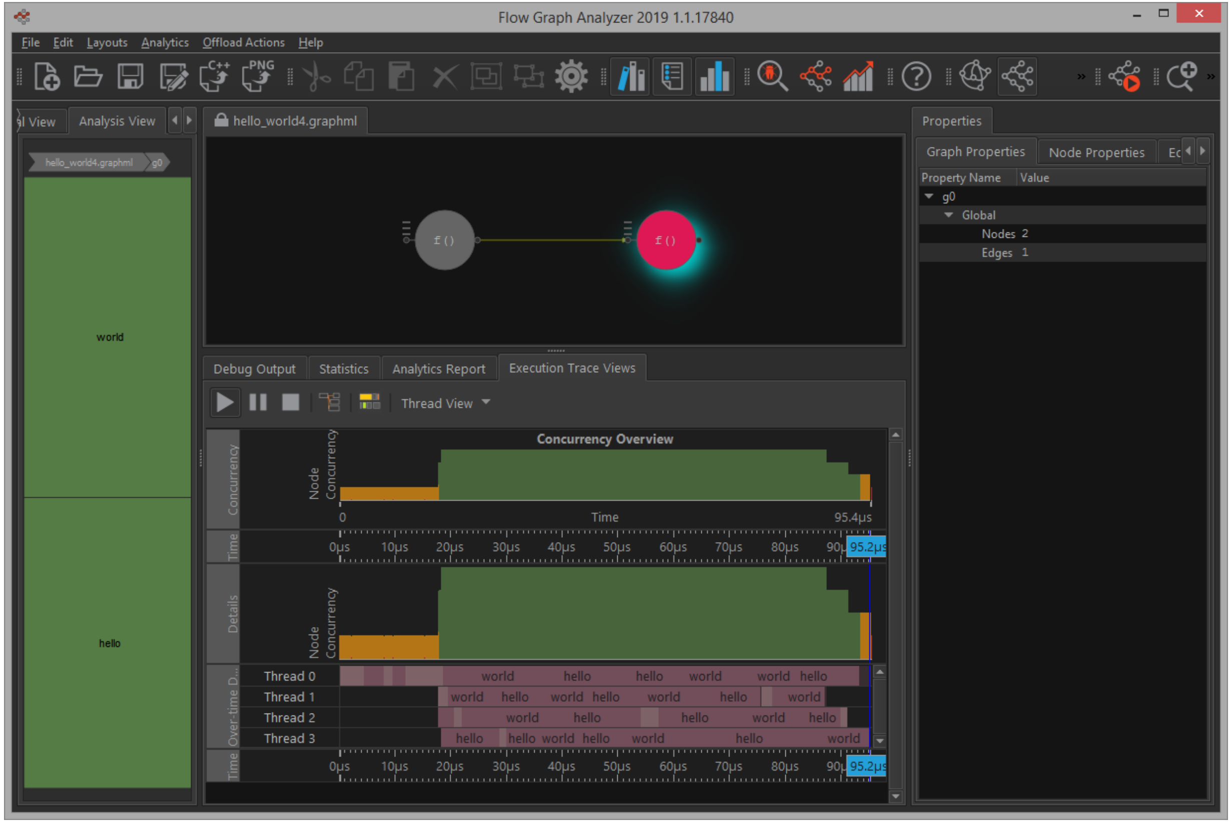
Interacting with the canvas

Clicking on a node on the canvas can highlight the corresponding node's tasks in the timeline. This is turned OFF by default.

Clicking on a section with low concurrency will highlight the nodes that are active at that time.

These nodes would be the starting point of a cause-and-effect analysis to see if they were responsible for the lower concurrency

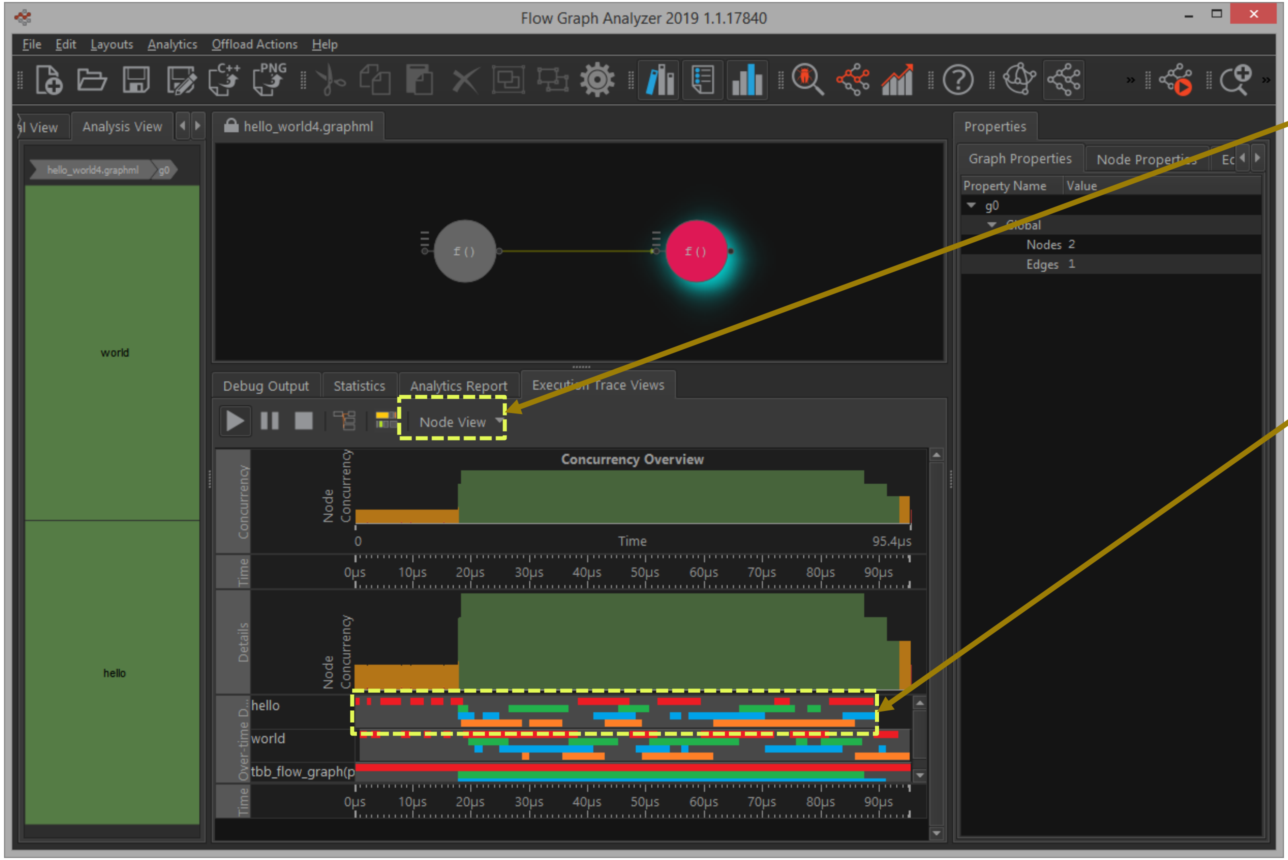
Examining the trace data through Trace Playback



Playback of execution traces to see how data is flowing through the graph.

Allows you to see how the data flows through the graph and what sections of the graph result in good or poor scaling

Examining the trace data: node view



Node view captures all execution traces for a given node and presents it in a single swim-lane for the node

Each node swimlane is comprised of multiple swimlanes representing the threads which executed an instance of the node.

Provides a compact representation of a node's execution

Challenges With asynchronous task graphs

Creating implicit or explicit task-graphs programmatically is easy

- Determining what was created is hard in many cases

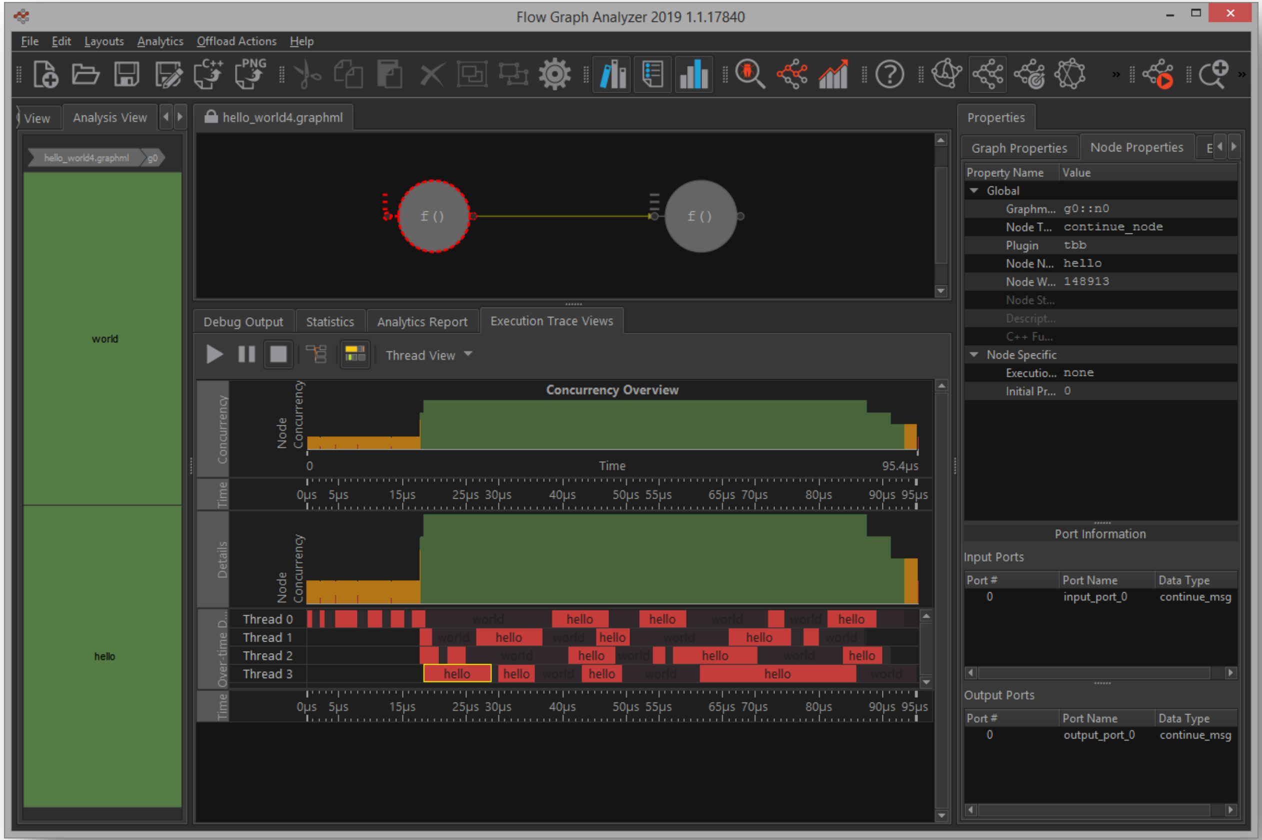
✓ New programming paradigm

✓ Allows you to stream data through the graph, which makes debugging challenging

Graph algorithms can be latency-bound or throughput-bound

Parallelism is unstructured in certain types of graphs, so performance analysis can be challenging

Examining the trace data with data analysis



How do we know which instance of the **Hello** task is in response to which input message?

Helps answer the following questions:

Are the tasks operating on data retiring in order?

Are they out of order?

We need to track the data flowing through the graph

Examining the trace data with data analysis, cont.

Harder to track the data in dependency graphs as the Data ID cannot be propagated from one node to the next

- **continue_node** requires an input of type **continue_msg**

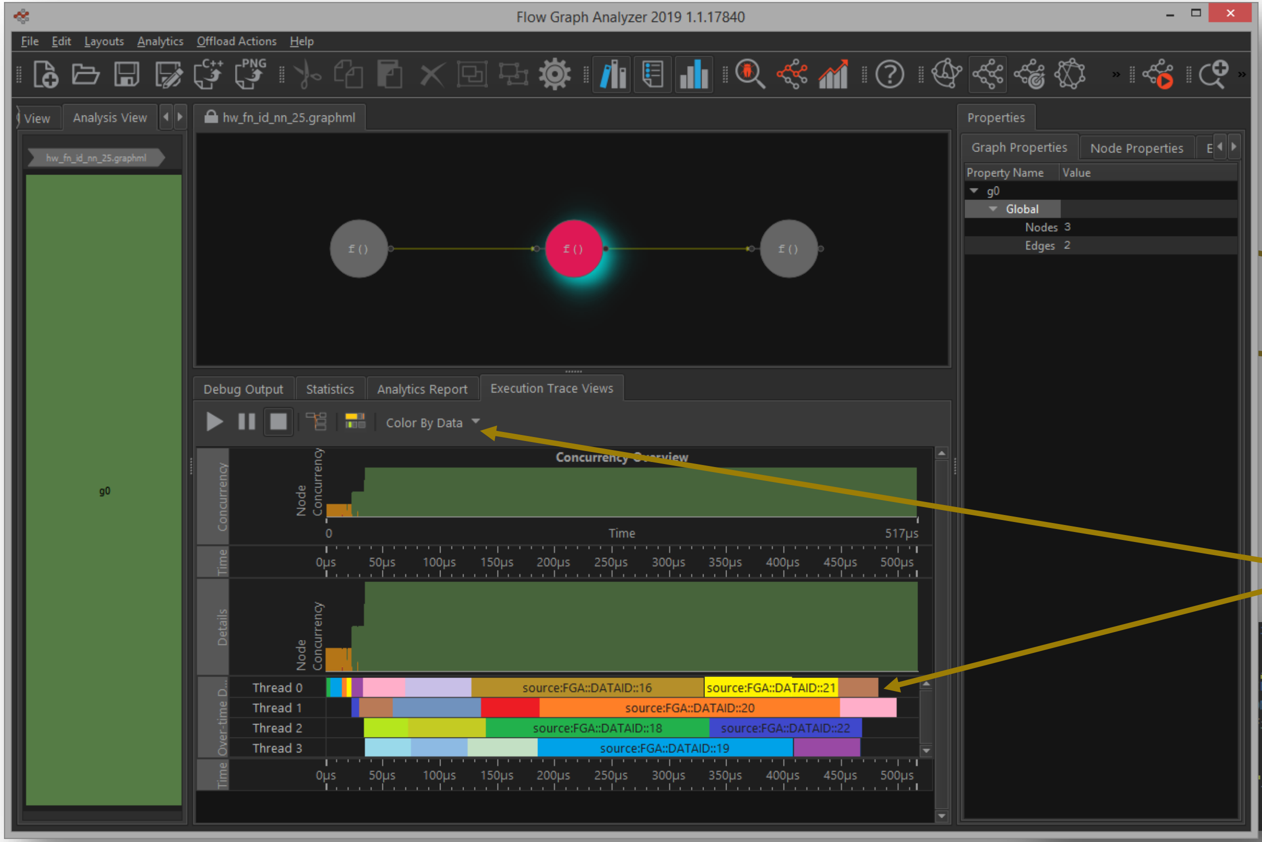
```
continue_node<continue_msg> hello( hello_world_g0, []( continue_msg & ) {
           cout << “Hello “;
       } );

continue_node<continue_msg> world(( hello_world_g0,[]( continue_msg & ) {
           cout << “World!\n“;
       } );
```

We are going to convert the Hello World example to use **function_node** instead so we can send the ID from one node to the next

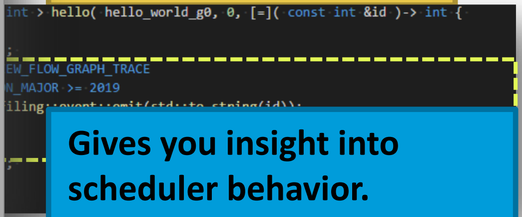
```
function_node< int, int > hello( hello_world_g0, 0, [=]( const int &id )-> int { ...
});
function_node< int, int > world( hello_world_g0, 0, [=]( const int &id ) -> int {...
} );
```

Examining the trace data with data analysis, cont.



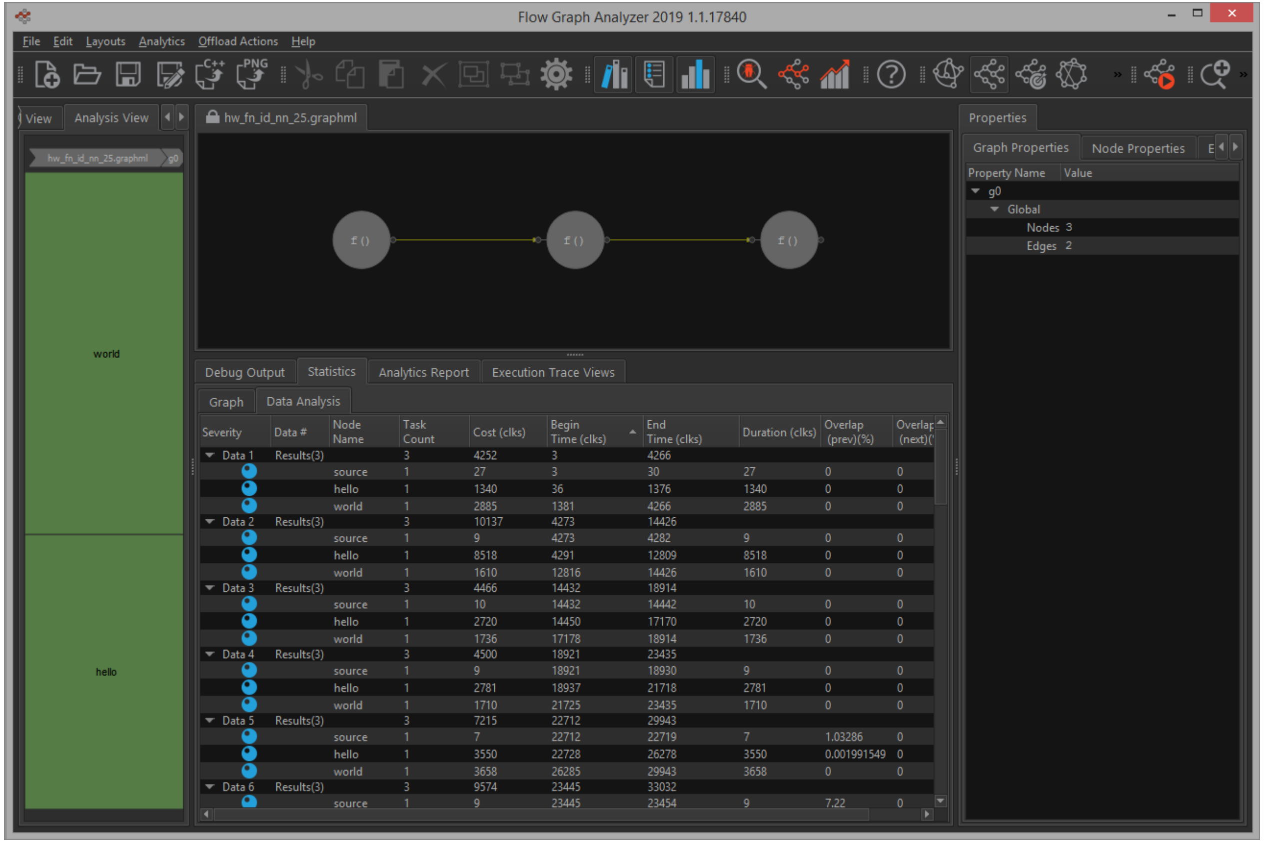
Data tracking using an experimental feature will allow you to track which task instance is for which inputs.

1. We changed our graph to use a **function_node** instead of a **continue_node**
2. We have a **source_node** that streams 25 messages/data through the graph
3. We modified the graph to emit the data id from the node **source** to **hello** and **hello** to **world**.
4. We add an user event API to tell the tool which data we are processing in each node.



Gives you insight into scheduler behavior.

Examining the trace data with data analysis, cont.



Severity	Data #	Node Name	Task Count	Cost (clks)	Begin Time (clks)	End Time (clks)	Duration (clks)	Overlap (prev)(%)	Overlap (next)('
Data 1	Results(3)		3	4252	3	4266			
		source	1	27	3	30	27	0	0
		hello	1	1340	36	1376	1340	0	0
		world	1	2885	1381	4266	2885	0	0
Data 2	Results(3)		3	10137	4273	14426			
		source	1	9	4273	4282	9	0	0
		hello	1	8518	4291	12809	8518	0	0
		world	1	1610	12816	14426	1610	0	0
Data 3	Results(3)		3	4466	14432	18914			
		source	1	10	14432	14442	10	0	0
		hello	1	2720	14450	17170	2720	0	0
		world	1	1736	17178	18914	1736	0	0
Data 4	Results(3)		3	4500	18921	23435			
		source	1	9	18921	18930	9	0	0
		hello	1	2781	18937	21718	2781	0	0
		world	1	1710	21725	23435	1710	0	0
Data 5	Results(3)		3	7215	22712	29943			
		source	1	7	22712	22719	7	1.03286	0
		hello	1	3550	22728	26278	3550	0.001991549	0
		world	1	3658	26285	29943	3658	0	0
Data 6	Results(3)		3	9574	23445	33032			
		source	1	9	23445	23454	9	7.22	0

Data tracking using an experimental feature will allow you to track which task instance is for which inputs

Statistics for the graph is organized by data operated on and can be seen in Data Analysis tab under Statistics

Using data analysis, the questions posed earlier can be answered.

You can examine the trace data to see if the data is retiring in-order or out-of-order.

If the algorithm is meant to be latency bound, then order is important. If it is throughput bound, data can retire out-of-order

Challenges with asynchronous task graphs

Creating implicit or explicit task-graphs programmatically is easy

- Determining what was created is hard in many cases

✓ New programming paradigm

✓ Allows you to stream data through the graph, which makes debugging challenging

✓ Graph algorithms can be latency-bound or throughput-bound

Parallelism is unstructured in certain types of graphs, so performance analysis can be challenging

Walking through an Example

Understanding the performance

A simulation example

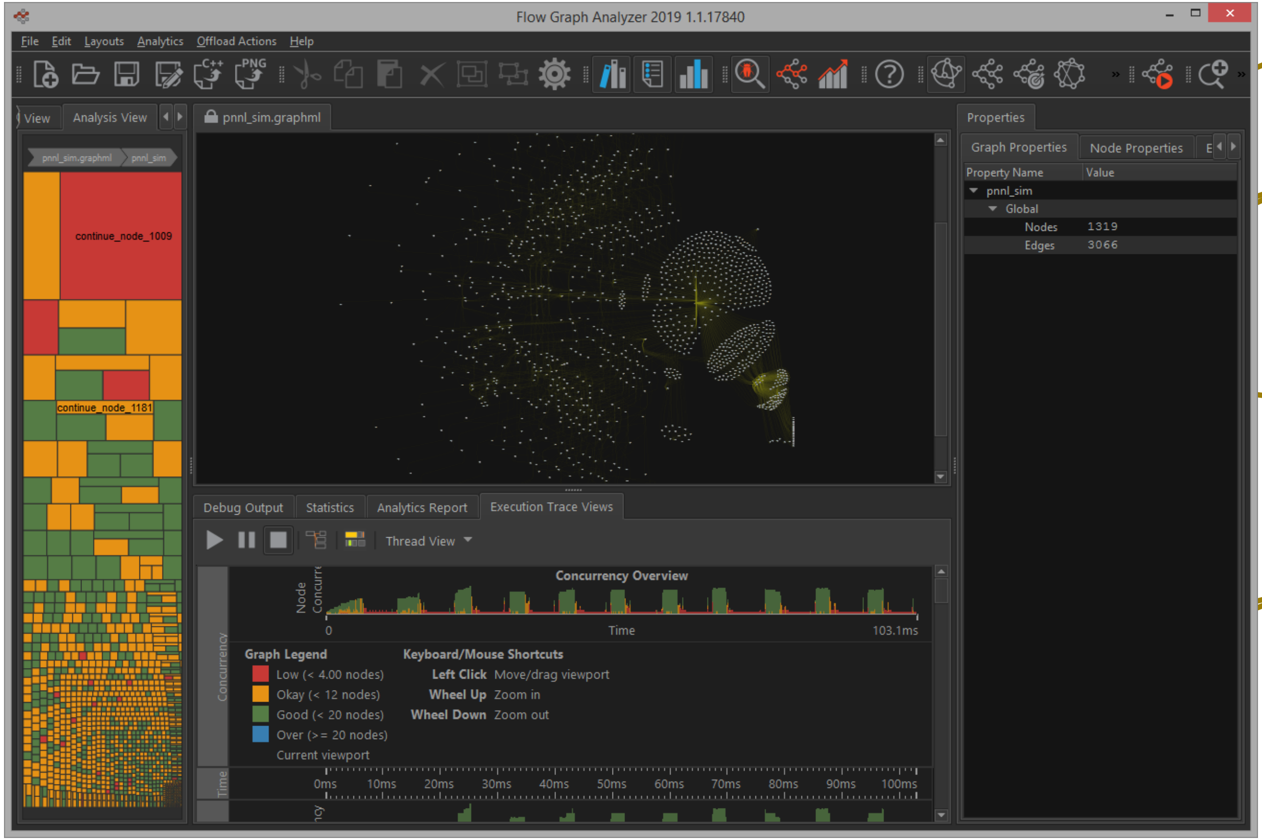
Goes through multiple time steps

Graph is created once programmatically and executed for each time step

- A message is sent to the graph to trigger each time step
- Wait for the graph to process the message (current time step) before the next time step is triggered
- Implemented as a dependency graph using TBB continue_node

Measured performance shows some performance scaling w.r.t serial implementation

Example: performance analysis



A complex graph was created programmatically.

Graph has 1319 nodes and 3066 edges.

General health of the graph with a mix of red, yellow and green

Concurrency observed over time ranges from good concurrency where all cores are kept busy to very few kept busy

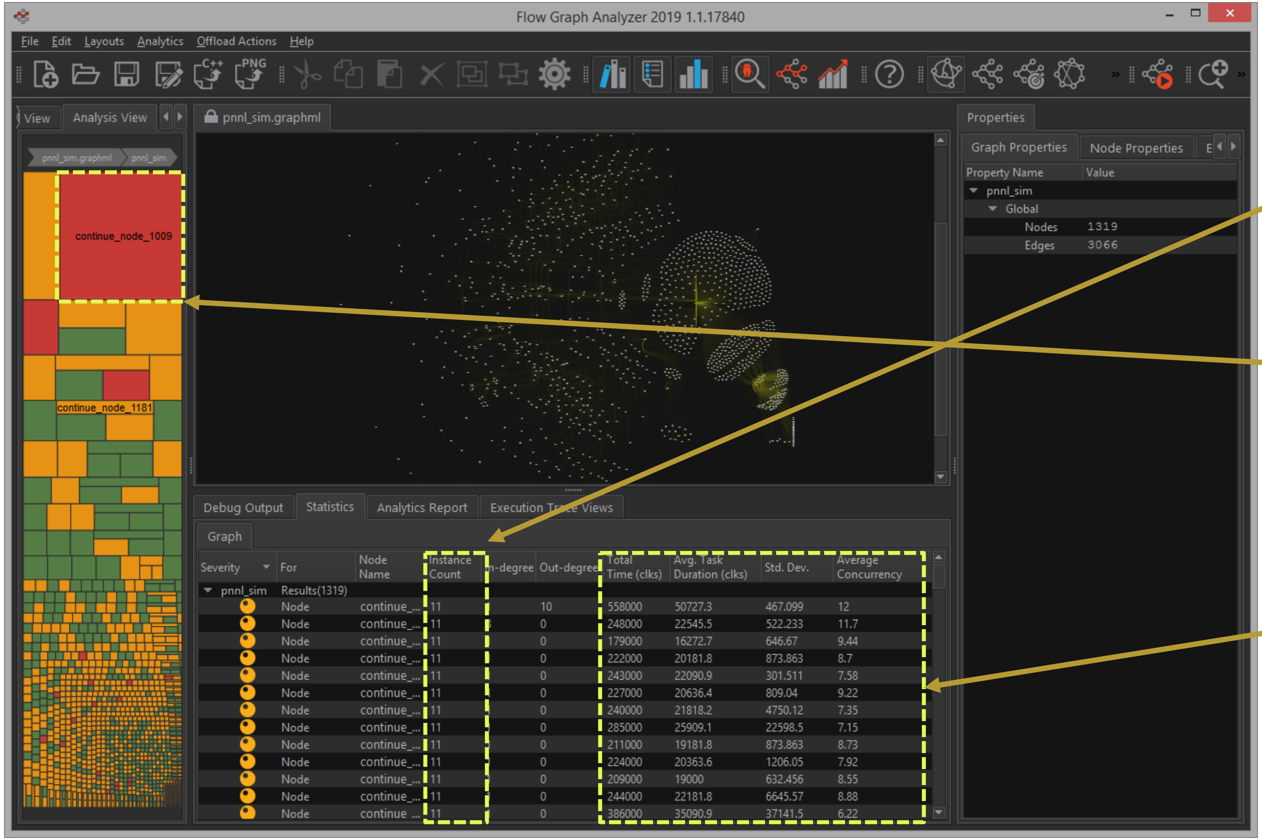
What do the colors mean?

Challenges with asynchronous task graphs

- ✓ Creating implicit or explicit task-graphs programmatically is easy
  - ✓ Determining what was created is hard in many cases
- ✓ New programming paradigm
- ✓ Allows you to stream data through the graph, which makes debugging challenging
- ✓ Graph algorithms can be latency-bound or throughput-bound

Parallelism is unstructured in certain types of graphs, so performance analysis can be challenging

Example: identifying problem areas



What was run and how much was run?

Run captures 11 time steps

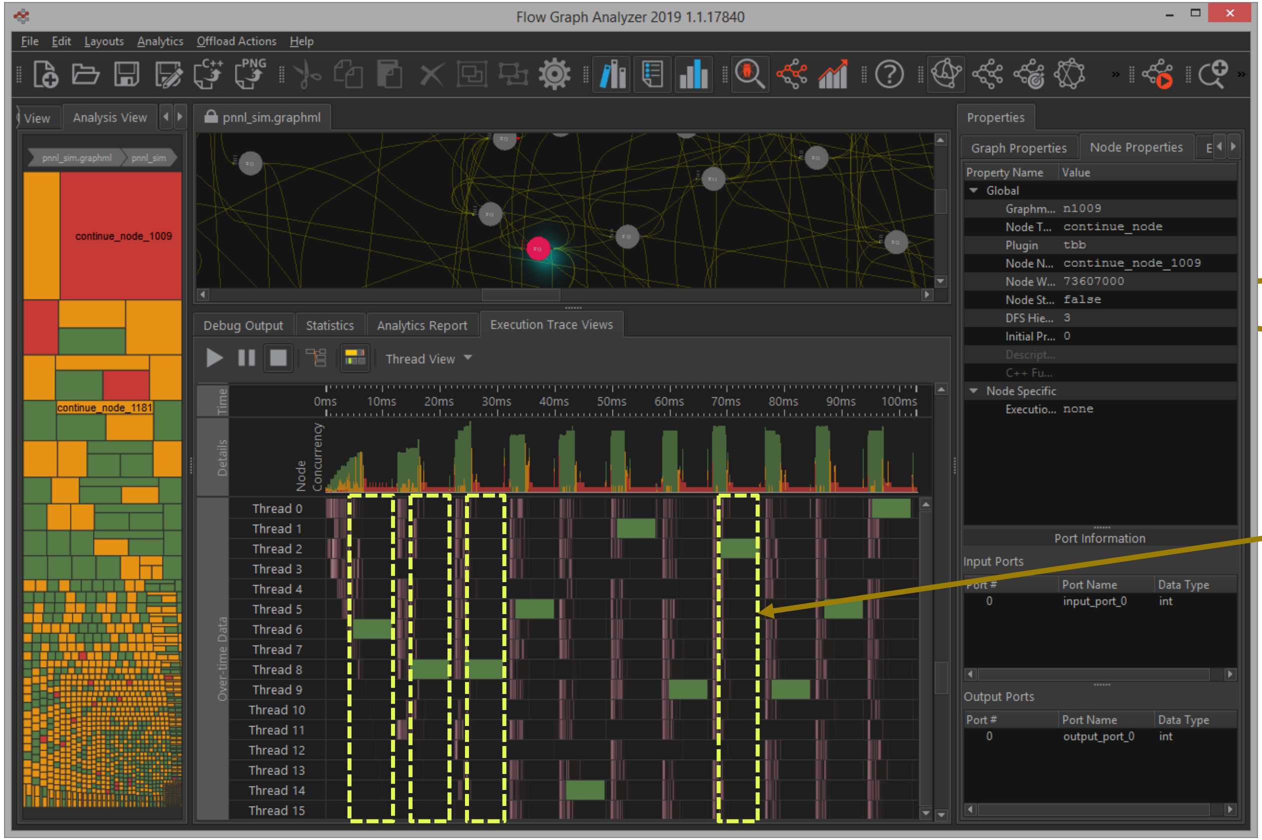
Appears to have one node that consumes a lot of CPU time.

This node also has an observed concurrency that is poor when it executes

Clicking on the node takes you to the node in the graph visualization

You can also sort on the appropriate column in the statistics table.

Example: identifying problem areas, cont.



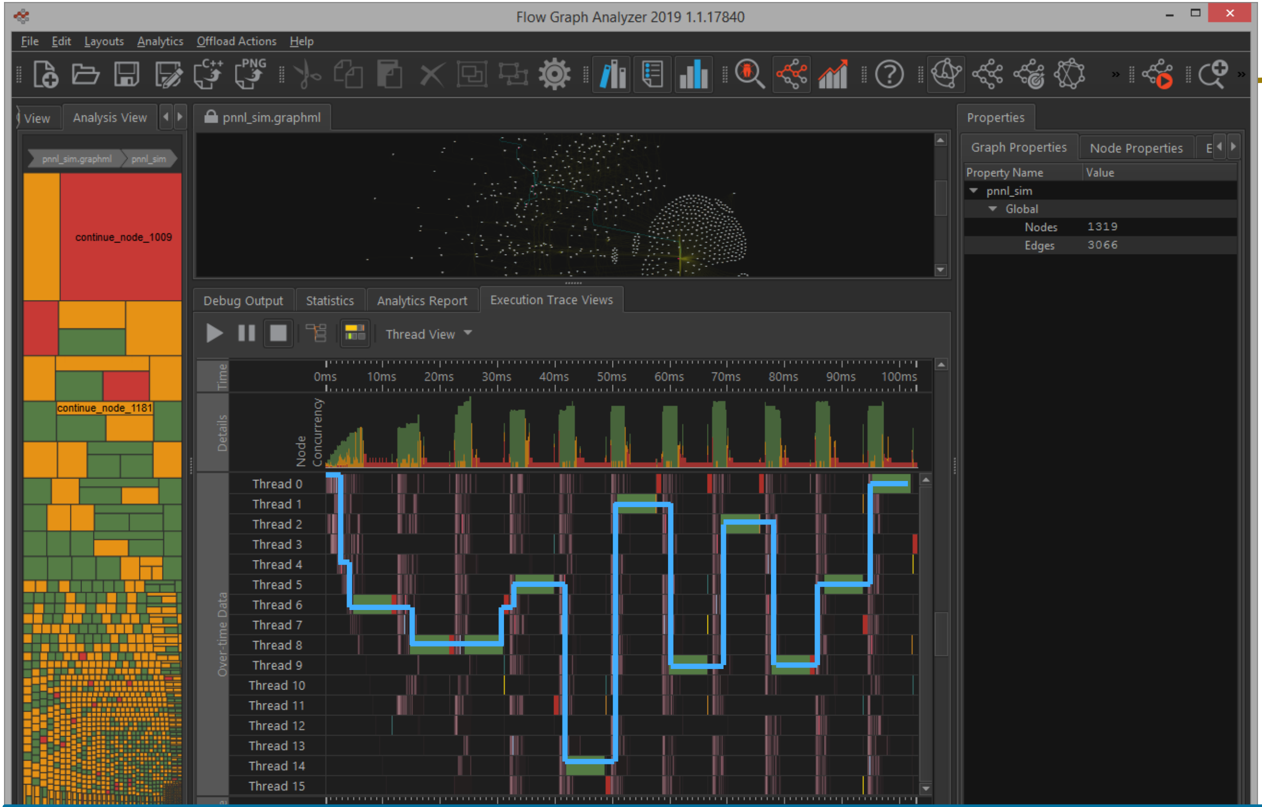
Clicking on the node takes you to the node in the graph visualization

1. **To see all tasks belonging to this node in the execution trace, you will have to enable this interaction.**
2. **Click on the Show/Hide tasks button**
3. **Now select the node in the canvas**

When this node is executing, the resource utilization is very poor.

1. **Improving the performance of this one node will substantially improve the performance of the graph.**

Example: critical path



Analysis features

1. **Critical Path**
2. **Rule-check**

Critical Path

Computes the Critical Path(s) for the graph using the execution trace information

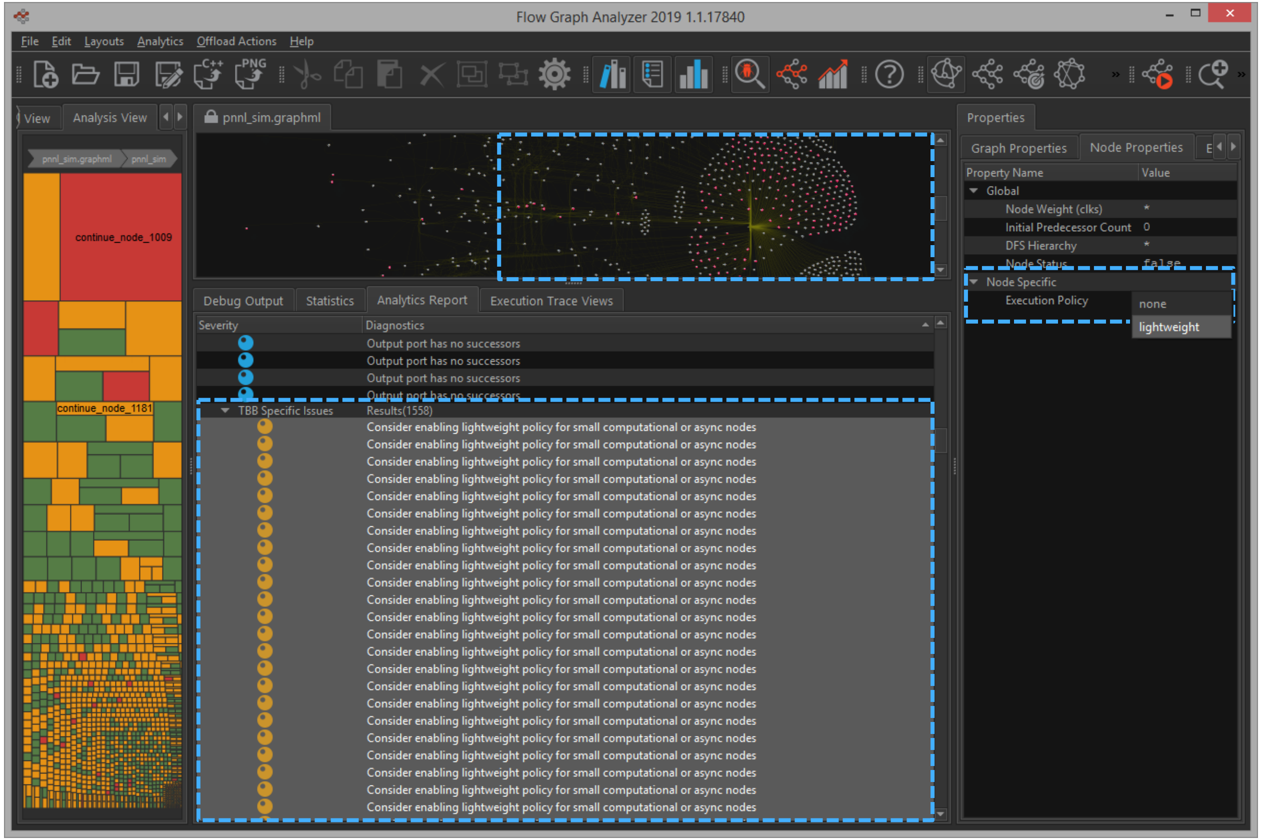
The most dominant task that had the maximum CPU Time and a corresponding low concurrency (continue_node_1009) is on this critical path

Critical path reduces the complexity in large graphs by isolating a small set of nodes for analysis and tuning for performance improvements

We found the problem Node: Fix it!

What else can we look at?

Example: performance analysis



Analysis features

1. **Critical Path**
2. **Rule-check**

Rule check

Rule-check runs registered rules that may include validation and performance rules

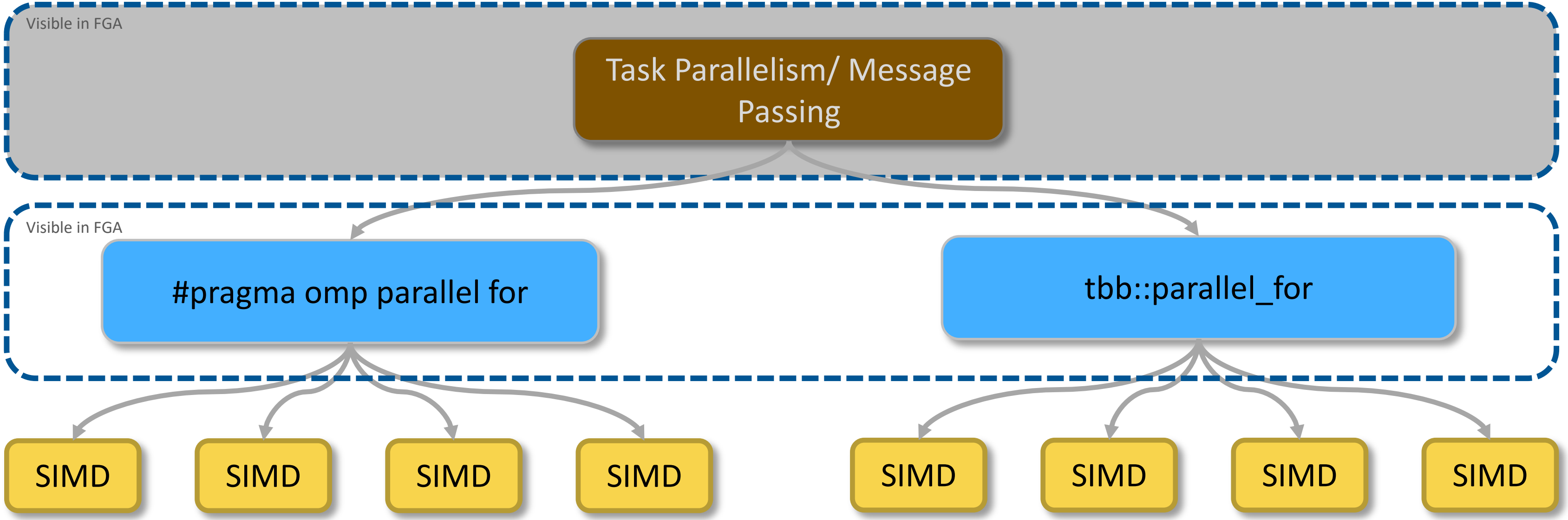
Challenges with asynchronous task graphs

- ✓ Creating implicit or explicit task-graphs programmatically is easy
  - ✓ Determining what was created is hard in many cases
- ✓ New programming paradigm
- ✓ Allows you to stream data through the graph, which makes debugging challenging
- ✓ Graph algorithms can be latency-bound or throughput-bound
- ✓ Parallelism is unstructured in certain types of graphs, so performance analysis can be challenging

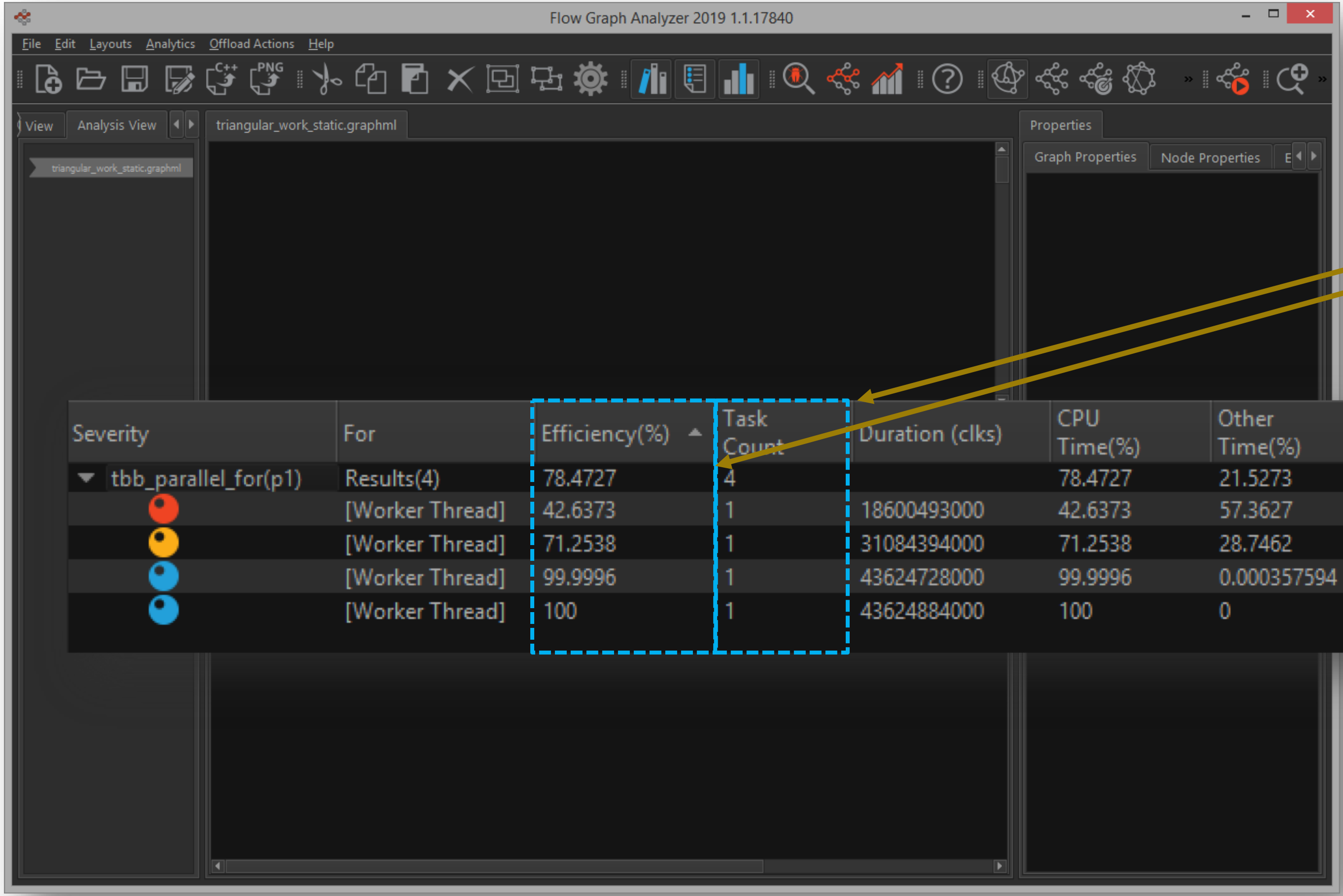
Nested and multi-level parallelism

What does it look like in FGA?

Applications often contain multiple levels of parallelism



Fork-join parallelism: tbb::parallel_for

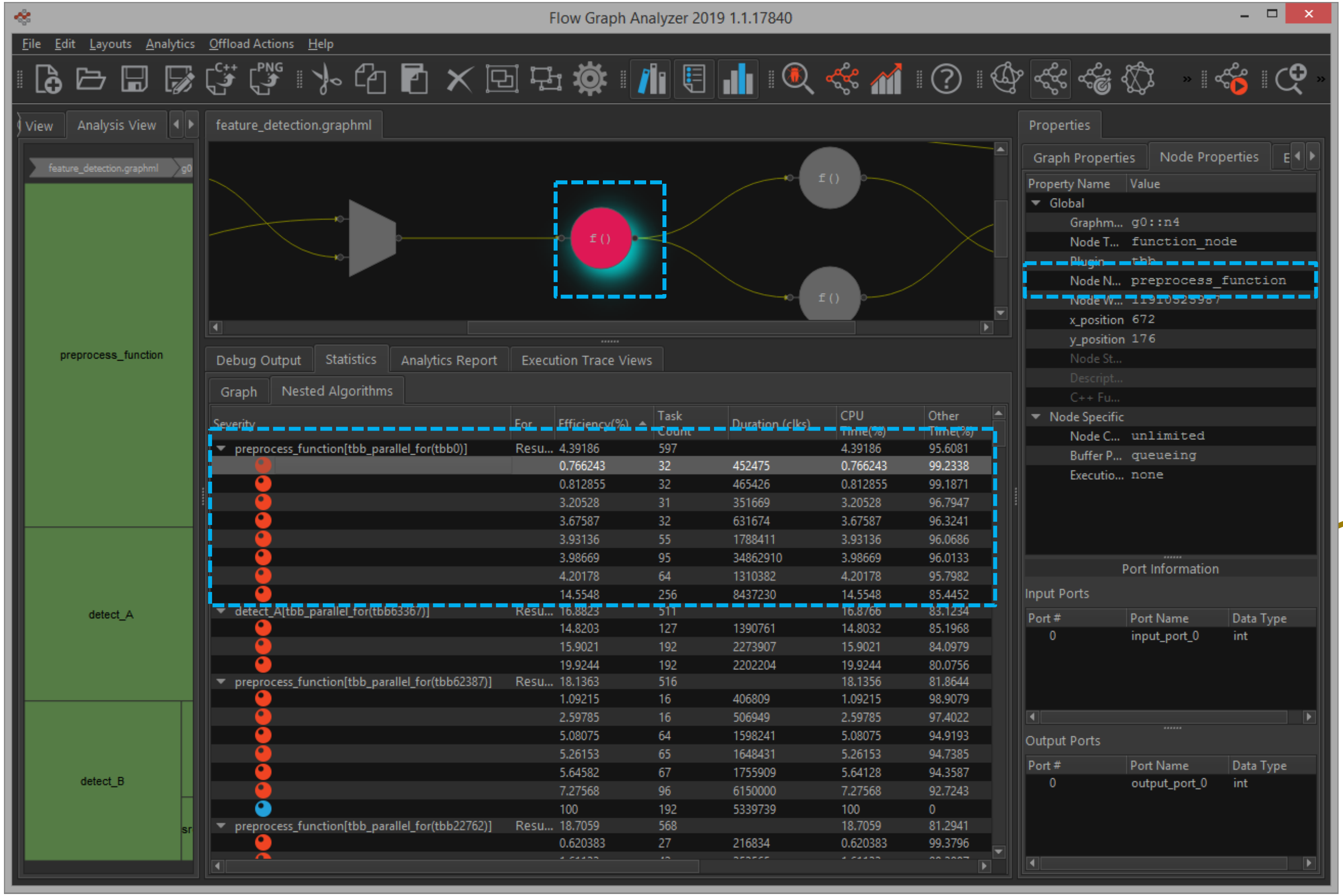


Severity	For	Efficiency(%)	Task Count	Duration (clks)	CPU Time(%)	Other Time(%)	Scheduler Time(%)
tbb_parallel_for(p1)	Results(4)	78.4727	4		78.4727	21.5273	0
	[Worker Thread]	42.6373	1	18600493000	42.6373	57.3627	0
	[Worker Thread]	71.2538	1	31084394000	71.2538	28.7462	0
	[Worker Thread]	99.9996	1	43624728000	99.9996	0.000357594	0
	[Worker Thread]	100	1	43624884000	100	0	0

Captures the execution task-graph for a fork-join construct and provides additional analytics that present information about the construct

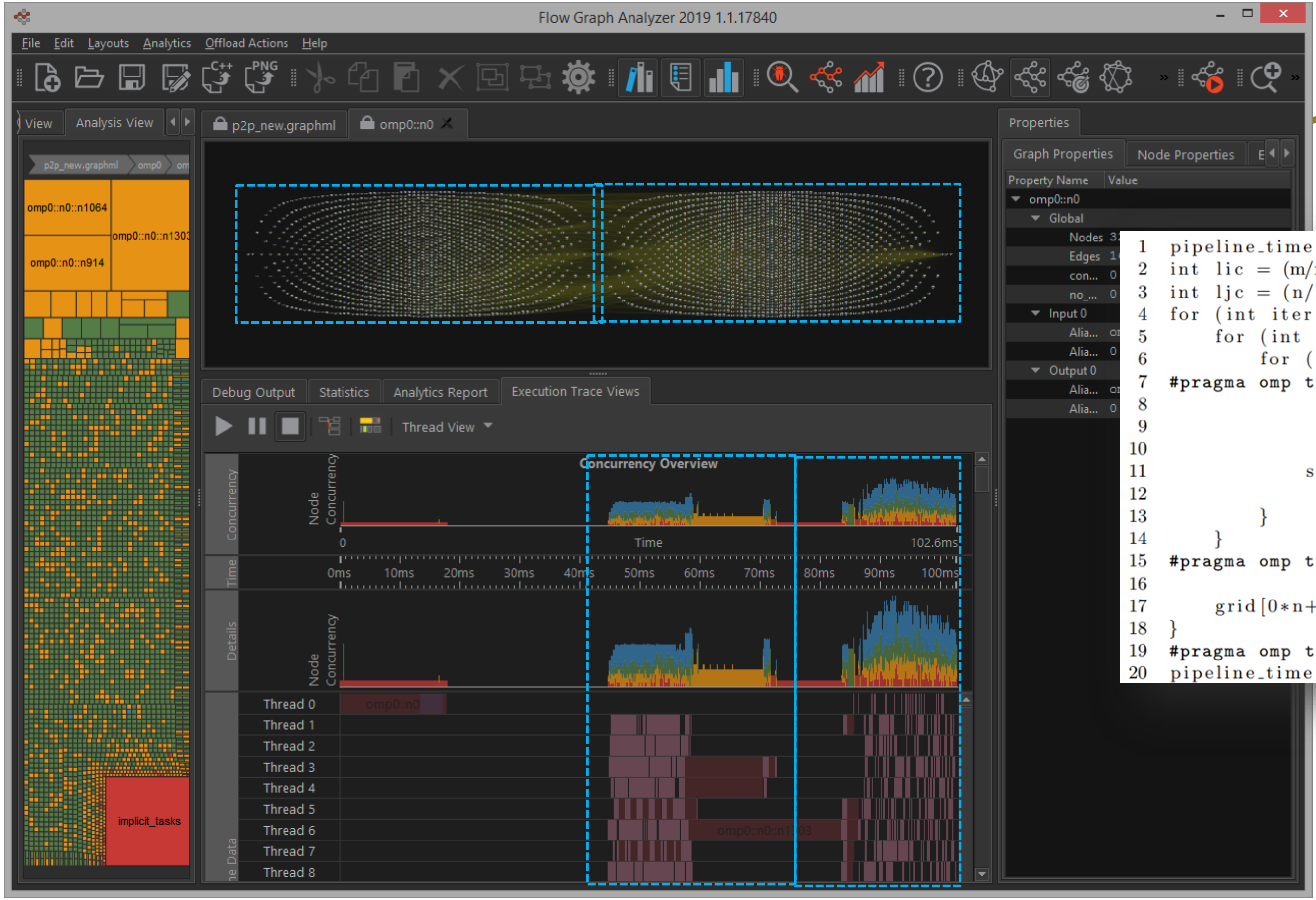
1. **Imbalance**
2. **Efficiency**

Multi-level parallelism: graph level + fork-join



Timeline shows trace information for the graph and any nested parallelism that is present

Multi-level parallelism in OpenMP*



Double-click on the parallel region node to see the activity within the region

```
pipeline_time = prk_wtime();
int lic = (m/mc-1) * mc + 1;
int ljc = (n/nc-1) * nc + 1;
for (int iter = 0; iter<=iterations; iter++) {
    for (int i=1; i<m; i+=mc) {
        for (int j=1; j<n; j+=nc) {
#pragma omp task depend(in:grid[0],grid[(i-mc)*n+j],\
                            grid[i*n+(j-nc)],\
                            grid[(i-mc)*n+(j-nc)]) \
                 depend(out:grid[i*n+j])
            sweep_tile(i, MIN(m,i+mc), j, \
                       MIN(n,j+nc), n, grid);
        }
    }
#pragma omp task depend(in:grid[(lic-1)*n+(ljc)])\
                 depend(out:grid[0])
    grid[0*n+0] = -grid[(m-1)*n+(n-1)];
}
#pragma omp taskwait
pipeline_time = prk_wtime() - pipeline_time;
```

Intel® Advisor – Flow graph analyzer

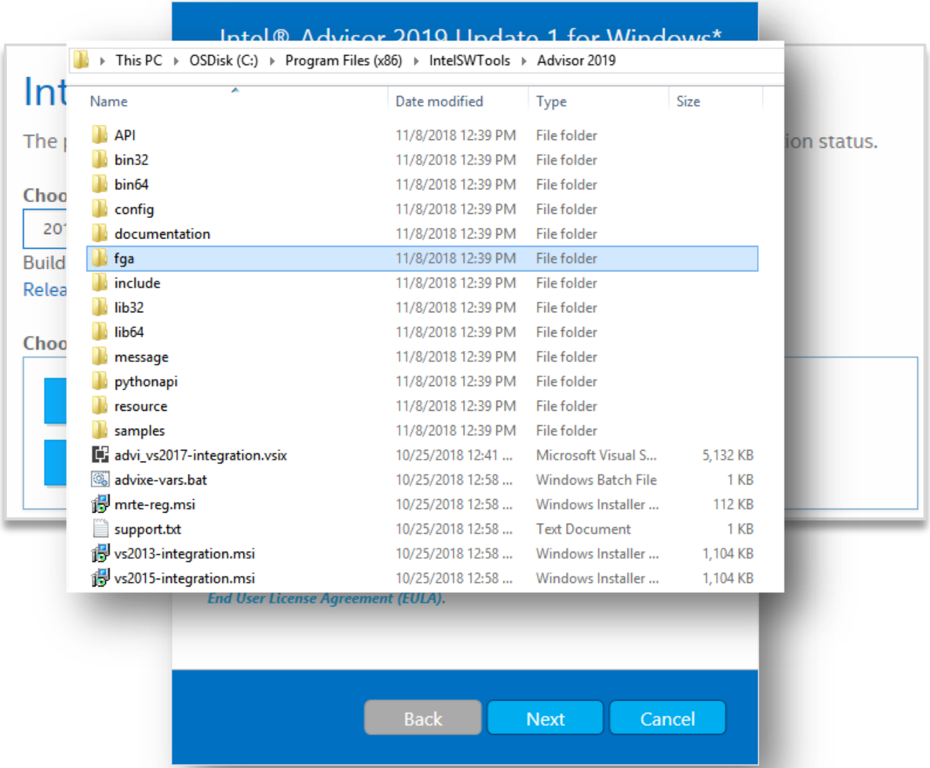
Download through Intel® Advisor package

Intel® Advisor – Flow Graph Analyzer

Product feature in Intel® Parallel Studio XE 2019

Tool supports analysis and design of parallel applications using OpenMP* and Threading Building Blocks

Available for Windows*, Linux* and MacOS*



https://software.intel.com/en-us/articles/getting-started-with-flow-graph-analyzer

Summary

Asynchronous task-graphs improves the efficiency of irregular and runtime dependent execution

- TBB and OpenMP* provide mechanisms to program in this manner

Flow Graph Analyzer helps you create, debug, visualize and analyze such graphs

- Critical path analysis is crucial in reducing the complexity of the analysis problem to a handful of nodes
- Runtime specific analyses, such as the lightweight policy analysis for TBB, target additional performance improvements

Resources

Getting started with FGA

https://software.intel.com/en-us/articles/getting-started-with-flow-graph-analyzer

Driving Code Performance with Intel® Advisor's Flow Graph Analyzer

https://software.intel.com/en-us/download/parallel-universe-magazine-issue-30-october-2017

IWOMP 2018: Visualization of OpenMP* Task Dependencies Using Intel® Advisor – Flow Graph Analyzer

https://link.springer.com/chapter/10.1007%2F978-3-319-98521-3_12

CPUs, GPUs, FPGAs: Managing the alphabet soup with Intel Threading Building Blocks

https://software.intel.com/en-us/videos/cpus-gpus-fpgas-managing-the-alphabet-soup-with-intel-threading-building-blocks

Legal Disclaimer & Optimization Notice

TECH.
DECODED

Software