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

Vasanth Tovinkere  |  Architect, Flow Graph Analyzer
Intel® Corporation

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

Threading Building Blocks (TBB)

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

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

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

Canvas for visualizing graphs

Displays the execution trace data, graph statistics and output generated by custom analytics and allows interactions with this data

Hierarchical view of the graph displayed shown as a tree

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
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

C/C++ code generated from the graph
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
Compiling and collecting traces

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

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

>set FGT_ROOT=<installation-directory>ga\fgt

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

>hello_world.exe
```

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

```shell
>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.
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**

```cpp
continue_node<continue_msg> hello( hello_world_g0, []( continue_msg & ) {
    cout << "Hello \n; 
};

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

```cpp
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.

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

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

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.
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

Optimization Notice
Copyright © 2018, Intel Corporation. All rights reserved.
Other names and brands may be claimed as the property of others.
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

Task Parallelism/ Message Passing

#pragma omp parallel for

tbb::parallel_for

SIMD  SIMD  SIMD  SIMD  SIMD  SIMD  SIMD  SIMD
Fork-join parallelism: tbb::parallel_for

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

1. Imbalance
2. Efficiency

<table>
<thead>
<tr>
<th>Sev.</th>
<th>For</th>
<th>Efficiency</th>
<th>Task Count</th>
<th>Duration (cls)</th>
<th>CPU Time(%</th>
<th>Other Time(%)</th>
<th>Scheduler Time(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>tbb_parallel_for(p1)</td>
<td></td>
<td>4</td>
<td>1</td>
<td>18600493000</td>
<td>42.6373</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>[Worker Thread]</td>
<td>78.4727</td>
<td>1</td>
<td>1</td>
<td>31084394000</td>
<td>71.2538</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>[Worker Thread]</td>
<td>99.9996</td>
<td>1</td>
<td>1</td>
<td>43624728000</td>
<td>99.9996</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>[Worker Thread]</td>
<td>100</td>
<td>1</td>
<td>1</td>
<td>43624884000</td>
<td>100</td>
<td>0</td>
</tr>
</tbody>
</table>
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*

Optimization Notice

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

```
1  pipeline.time = prk.wtime();
2  int lic = (m/mc-1) * mc + 1;
3  int ljc = (n/nc-1) * nc + 1;
4  for (int iter = 0; iter <= iterations; iter++) {
5      for (int i=1; i<mc; i++) {
6          for (int j=1; j<nc; j++) {
7              #pragma omp task depend(in:grid[0], grid[(i-mc)*n+j],
8                  grid[i+n+(j-nc)],
9                  grid[(i-mc)*n+(j-nc)])
10                 depend(out:grid[i*n+j])
11                  sweep.tile(i, MIN(m, i+mc), j, 
12                      MIN(n, j+nc), n, grid);
13          }
14      }
15      #pragma omp task depend(in:grid[((lic-1)*n+(ljc))])
16         depend(out:grid[0])
17      grid[0*n+0] = -grid[(m-1)*n+(n-1)];
18  }
19  #pragma omp taskwait
20  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*

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


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


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

Legal Disclaimer & Optimization Notice

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit www.intel.com/benchmarks.

INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Copyright © 2018, Intel Corporation. All rights reserved. Intel, the Intel logo, Pentium, Xeon, Core, VTune, OpenVINO, Cilk, are trademarks of Intel Corporation or its subsidiaries in the U.S. and other countries.

Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804