2014 43rd International Conference on Parallel Processing Workshops

Flow Graph Designer

A Tool for Designing and Analyzing Intel® Threading Building Blocks flow graphs

Vasanth Tovinkere and Michael Voss
Software and Services Group
Intel Corporation
vasanth.tovinkere@intel.com, michaelj.voss@intel.com

Abstract*—Flow Graph Designer is a visualization tool that supports the analysis and design of parallel applications that use the Intel® Threading Building Blocks (Intel® TBB) flow graph interface. The flow graph interface allows developers to express the dependency, streaming and data flow graphs present in many domains such as media, gaming, finance, high performance computing and healthcare. Because the flow graph interface introduced a style of programming that is different than loop- and task-parallel approaches, there is a need for new tool support. This paper presents Flow Graph Designer, an experimental graphical tool developed to make the flow graph interface easier to adopt, use, debug and tune. This paper provides a brief introduction to the Intel TBB flow graph interface and an overview of the key features provided by the Flow Graph Designer. Three examples are then presented that demonstrate the utility of this tool for important usage scenarios encountered while designing and analyzing flow graph applications.

Keywords*—data flow graphs, dependency graphs, performance analysis, code construction

I. Introduction

The Intel® Threading Building Blocks (Intel® TBB) library is a widely used C++ template library that provides generic parallel algorithms, concurrent containers, a scalable memory allocator and other features that help developers easily create threaded programs that take advantage of the performance potential of multicore architectures [1]. The flow graph interface [2] introduced in Intel TBB 4.0 extended its capabilities to allow fast, efficient implementations of dependency graph and data flow algorithms, enabling developers to exploit parallelism at higher levels in their applications.

Because the flow graph interface introduced a style of programming that is different than the loop- and task-parallel approaches previously supported by Intel TBB, the OpenMP API [3] and Intel® Cilk™ Plus [4], there is a need for tool support that is specific to this style of parallel programming.

This paper presents Flow Graph Designer (FGD), available at [22], an experimental tool that supports Intel TBB flow graph technology and allows developers to visually design, model, validate, implement and analyze the performance of complex applications that use that interface. While this tool is specialized to a specific interface in the Intel TBB library, the data-flow, reactive and streaming styles of programming enabled by the flow graph interface are not unique to this library. Therefore we expect that insights gained from this tool may be applied to tools designed for other similar languages.

This paper makes the following contributions:

- We present an experimental tool, Flow Graph Designer (FGD), which contains features unique to the analysis and design of data flow and dependency graph applications.
- We provide three example usage scenarios that demonstrate the important features of FGD:

1. A design scenario, where the high-level structure of an application is drawn in the tool, this structure is validated using checks built in to the tool, a C++ scalability model is generated and evaluated, and a final implementation is created by filling in the C++ stubs generated by the tool.
2. A performance analysis scenario, where an existing flow graph application is executed and a trace is collected for analysis in the tool. The charts and analytics engine capabilities of FGD are used to explore task granularities, the concurrency profile and the application's critical path to determine if any scalability issues exist.
3. A model generation scenario, where a trace of an existing flow graph application is used to create a C++ model that mimics important program characteristics without including application code that might contain sensitive intellectual property.

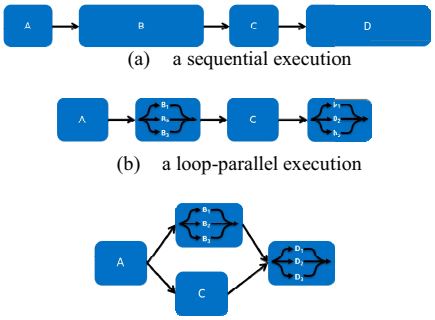
The rest of the paper is organized as follows: In Section II, we present a short introduction to the Intel TBB flow graph interface and how it is used to build parallel data flow or dependency graph applications. In Section III, we describe the features and capabilities of Flow Graph Designer that are used to design and analyze these parallel applications. In Section IV, we provide three example scenarios that highlight important use cases for the tool's features. Related work is provided in Section V and we present our conclusions in Section VI.

II. The Flow Graph Interface

The Intel® Threading Building Blocks (Intel® TBB) flow graph interface allows developers to express parallelism at levels in their applications that are typically not amenable to loop-level parallelism. It provides features to support the dependency, streaming and data flow graphs that can be found in many domains such as media, gaming, finance, high performance computing and healthcare.

1530-2016/14 $31.00 © 2014 IEEE
DOI 10.1109/ICPPW.2014.31

Consider Fig 1(a), where a simple application invokes four functions sequentially. A loop-parallel approach, Fig 1(b), exploits concurrency in each function with algorithms such as an Intel TBB `parallel_for` or `parallel_reduce`, reducing the execution time of functions where it applies.



(a) a sequential execution

(b) a loop-parallel execution

(c) a graph- and loop-parallel execution

Fig 1. A simple application with different forms of parallelization

However, Fig 1(b) may still be overly constrained. For example, functions B and C may both require the output generated by A, but C may not depend on the output of B. Fig 1(c) shows the graph- and loop-parallel implementation of this example. In this implementation, the loop-level parallelism is exposed and the overly constrained total ordering is replaced with a partial ordering that would allow B and C to execute concurrently.

When using an Intel TBB flow graph, computations are represented by nodes and the communication channels between these computations are represented by edges. The user is responsible for using edges to express all dependencies that must be respected when nodes are scheduled for execution, giving the Intel TBB scheduler the flexibility to exploit the parallelism that is explicit in the graph topology. When a node in the graph receives a message, an Intel TBB task is spawned to execute its body object on the incoming message.

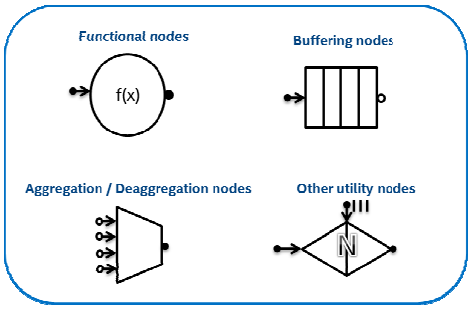


Fig 2. The types of nodes supported by the flow graph interface

The flow graph interface supports several different types of nodes, as shown in Fig 2, including functional nodes that execute user-provided body objects, buffering nodes that can be used to order and buffer messages as they flow through the graph, aggregation and deaggregation nodes that join and split messages, and other special purpose nodes. Users programmatically connect instances of these node types together with edges to specify the dependencies between them and provide body objects to perform their work.

Below is the source code for a simple "Hello World" flow graph application. This example does not contain any parallelism, but demonstrates the syntax of the interface. In this code, two nodes are created, `hello` and `world`, which are constructed with C++ lambda expressions that print "Hello" and "World" respectively. Each node is of type `continue_node`, a functional node type provided by the interface. The `make_edge` call creates an edge between the `hello` node and the `world` node.

```
#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;
}
```

In the code above, the call to `hello.try_put(continue_msg())` sends a message to the `hello` node, causing it to spawn an Intel TBB task to execute its body object. When that task completes, it sends a message to the `world` node, which will spawn a task to execute its body object. Only when all of the tasks spawned by the nodes are complete, does the call to `g.wait_for_all()` return.

The Intel TBB flow graph interface enables the expression of very complex graphs that may include thousands of nodes and edges, cycles, buffering and more. The flow graph interface is a set of C++ classes and functions that can be used to express these applications – it is not a visual language. However, the Flow Graph Designer tool that is described in this paper enables the visual construction and analysis of applications that use this interface.

More information about the flow graph interface can be found at the Intel TBB web site [1].

III. Overview of Flow Graph Designer

The Flow Graph Designer supports two workflows: (1) the design of flow graph applications and (2) the analysis of the runtime performance of flow graph applications.

A. The design of flow graph applications

The graphical user interface of the Flow Graph Designer allows you to create new graphs visually or load graphs that were previously created by the designer. Fig 3 shows a

screenshot of the Flow Graph Designer displaying a sample graph.

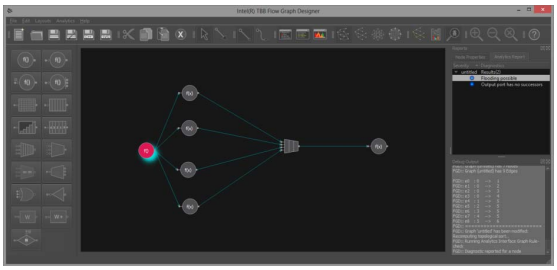


Fig 3. Flow Graph Designer

A palette of available Intel® TBB flow graph node types is available for developers to drag and drop nodes on to the canvas. These nodes are then connected together by drawing edges between them. A context menu on each node can be used to modify the data types for the messages that will flow through the input and output ports.

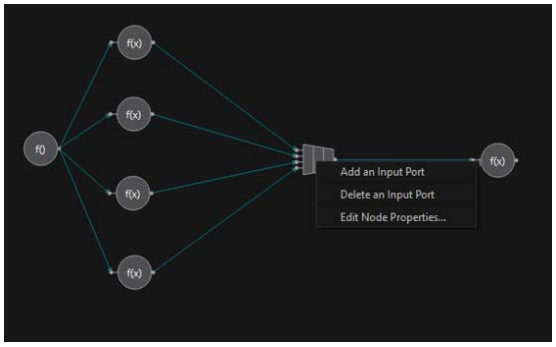


Fig 4. Context menu available to set the port types for a node.

Fig 4 shows a sample screenshot of the context menu where one can select the "Edit node properties" option to override the default port data types.

Some complex nodes, such as a `split_node` and `join_node`, require that their input or output data type are a tuple of all of their output types and input types respectively. Fig 5 shows the Node Properties dialog box for a `join_node`, which is an aggregation node that combines the inputs to form a tuple data type for the output port. In the dialog box, the input type for a given port is set by selecting the "Input port" for which the data type needs to be modified. After the types are set for all of the input ports, clicking the validate button will automatically generate the data type for the output port. This allows the node's input and output data types to be in sync at all times.

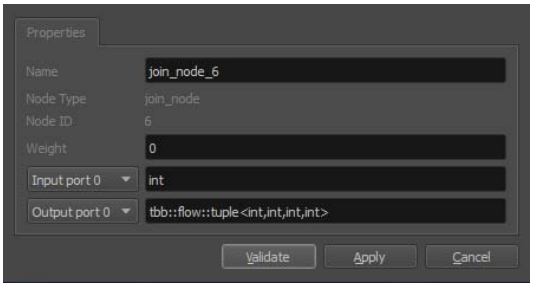


Fig 5. A Node Properties dialog box

In addition to validating a single node, the Flow Graph Designer tool also provides a "rule-check" algorithm that looks at all of the edges to determine if there is a mismatch of any kind. In addition to detecting data type inconsistencies, the rule-check algorithm also checks for cycles and other potentially problematic patterns. Once these identified issues have been addressed by the user, the tool allows the user to save the graph in GraphML file format [5]. One can also generate the framework C++ code for the graph, which uses the Intel® TBB flow graph interface.

The Node Properties dialog box also allows a "Weight" to be set for each node to indicate the node's computational complexity. When C++ code is generated from the graph, the body of each node is set to a busy spin loop and the duration of this loop is proportional to the "Weight" provided for the node. This C++ code can be compiled and run without adding application-specific code. If executed using various numbers of threads, these models can be used to estimate the scalability limits of the final implementation.

B. The performance analysis of graph applications

In addition to designing complex data flow and dependency graphs, the tool also allows developers to capture the graph topology and task execution traces from a running Intel® TBB flow graph application. The data from the running flow graph application is captured by the Flow Graph Collector, a companion component of Flow Graph Designer.

The Flow Graph Collector captures traces of important flow graph events, such as node and edge construction, and node body executions. To collect these events, it relies on instrumentation activated at compile-time and available in Intel TBB 4.2 update 3 and later. The Flow Graph Collector generates two output files: (1) a GraphML file that contains the graph topology and (2) a TraceML file that contains the task execution trace.

When a GraphML file is loaded into the Flow Graph Designer, the tool displays the topology described in the file. It also loads any associated TraceML file and displays the task execution trace data in two forms: per-thread task data over time and task concurrency over time.

1) Per-thread task data over time

Flow Graph Designer displays the tasks executed by each thread and the duration of each task is shown as a colored box on the horizontal band that represents the timeline of that thread. This box has a begin time equal to the begin time of the

task and an end time that reflects the end time of the task. The color of the box depends on the duration of the task.

The rule-of-thumb recommendation for an Intel TBB task is that it needs to be at least 10,000 instructions to amortize the cost of a task spawn and achieve good scaling. If at least 2 instructions are dispatched/retired every clock tick, then this would result in approximately 5,000 clock ticks worth of work per task. This is on the order of 2 µsecs per task. FGD colors tasks red if they are at or below a threshold of 3 µsecs, which indicates they are approaching this lower limit. All tasks less than or equal to 3 µsecs will be shown in red and all tasks over 1 msec will be shown in green. Tasks that fall between 3 µsecs and 1 msec are colored light yellow through dark orange.

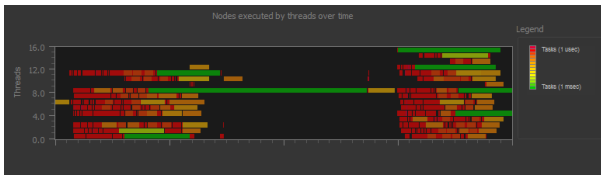


Fig 6. Per-thread task execute trace data over time.

Fig 6 provides a sample chart that shows the tasks executed by each thread. As described above, the colors of the tasks vary from red to green. All tasks that are yellow or green have enough work in them to amortize the cost of a task spawn. If tasks are too small, an option to improve performance is for the developer to group several nodes with small task execution times into one larger task to better amortize overheads.

In addition to visualizing the task execution times, "mouse-tracking" mode can be used with charts in FGD to see which node each task maps to. With this mode active, hovering the mouse over a task in the chart highlights the corresponding node in the topology canvas. This feature is extremely useful when the performance of a graph has to be debugged and allows one to quickly identify the node or nodes that affect the performance of a graph.

2) Task concurrency over time

FGD also provides a chart that shows the task concurrency over time. Again, color coding is used to highlight important values. Fig 7 shows a sample task concurrency histogram for an example application. This chart provides two key pieces of information: (1) the number of unique threads there were active during each region of time and (2) the concurrency break-down during that region. Since displays have limited resolution, each pixel-wide section of the chart represents not a single point in time but a region of time. The extent of this region shrinks as the user zooms in and grows as the user zooms out of the view. The height of the region at each pixel shows the total number of unique threads that were active during that region. The area under the pixel is a stacked bar that shows the break-down of task/node concurrency over that region.

The colors assigned in the concurrency histogram are set according to the default ranges defined by the lower and upper bounds as shown in TABLE 1. The lower and upper bounds indicate the number of active threads for the concurrency levels defined by the colors red, yellow, green and blue. Again, red indicates a region that warrants further investigation as this concurrency level usually indicates poor utilization of the system. Ideal concurrency is indicated by green color and the concurrency here will be close to system maximum of 100% at the mid-point of the range.

TABLE 1: Histogram Coloring Defaults Based on Number of Hardware Threads Available on the System

	Lower Bound	Upper Bound
red	0	0.25 x num hardware threads
yellow	0.25 x num hardware threads	0.75 x num hardware threads
green	0.75 x num hardware threads	1.25 x num hardware threads
blue	1.25 x num hardware threads	none

For example if on a system with 16 hardware threads, there is a region in which 16 unique threads executed tasks, the height of that region will be 16. If during half of that region, all 16 threads executed nodes concurrently, then half the bar will be green. If during the other 50% of the region only 1 node was executing concurrently (and it could be a different node or thread at each point of time), then the other half of the bar will be red. This display therefore shows the number of threads that are participating in the flow graph as well as how much true parallelism is achieved. There could be, for example, a region with a height of 16, showing that all threads participated, but is colored completely red, indicating that no node executions are overlapped during that time region.

Returning to Fig 7, on the left side of this bi-modal distribution, the concurrency of the graph is less than twelve for this application running on a 16-core machine. On the right hand side of the distribution, the concurrency is closer to sixteen. This chart provides a quick overview of the health of the flow graph. In the chart, there are periods of serial behavior between two concurrent phases. Identifying the serial node or nodes and improving their performance is a path to increase the scalability of this application.

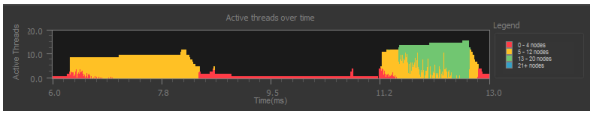


Fig 7. Task concurrency histogram over time.

The tool also allows one to interact with the data by zooming into the data, selecting a time on the chart to see which nodes are active at that time, and so on. Since the tool can map bad performing time regions to nodes, it is easier for developers to understand which portions of the graph are slowing the graph execution times.

3) The statistics engine

The Flow Graph Designer also computes additional statistics for each node, such as average task duration and standard deviation. This information is made available by the "graph statistics" analytics engine. This engine also displays the node in-degree and out-degree, and can compute the critical paths through the graph.

The chart area supports playback of the execution traces. One can playback the entire trace or zoom into a specific section and play just that region back. During playback, the

tasks that are executed over time are shown on the graph by highlighting the node that corresponds to the task. Also, the graph statistics are cleared and new statistics are computed for only the playback region.

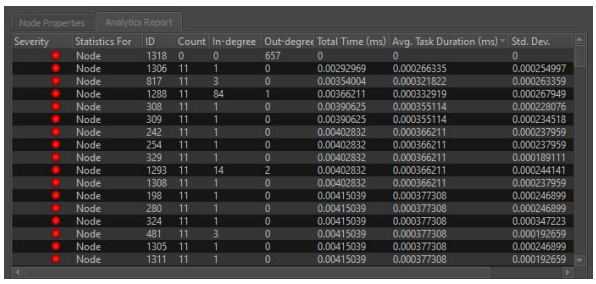


Fig 8. Sample graph statistics outputs

Fig 8 shows the sample output of the graph statistics analytics engine with the data sorted by average task duration. Each column in this table is sortable and provides users with the flexibility to look at problematic nodes based on in-degree, out-degree or task duration. The "Severity" column uses the same coloring scheme employed for task durations in Fig 6.

In addition to the analytics and charts described in this section, FGD also supports many graph layout schemes to visualize the graphs, such as hierarchical, circular and radial layouts.

IV. EXAMPLE USAGE SCENARIOS

A. System Configuration

In this section, experiments are run to demonstrate the effectiveness of Flow Graph Designer during the design and analysis of Intel TBB flow graph applications. All measurements were obtained on a system with two six core Intel® Xeon® E5-2640 Processors running at 2.5 GHz with a 15 MB Intel® Smart Cache and 48 GB of RAM. The system ran Microsoft* Windows Server 2012*. All applications were compiled in release mode with optimization on using Microsoft Visual Studio Ultimate 2012*, Intel® C++ Composer XE 2013 and Intel® Threading Building Blocks 4.2 update 3.

Refer to http://software.intel.com/en-us/articles/optimization-notice for more information regarding performance and optimization choices in Intel software products.

B. Generate code for a simple flow graph

As described in Section III, the Flow Graph Designer tool supports a designer workflow that lets developers describe their graphs visually, run rule checks on the description and then generate C++ stubs. In this section, we demonstrate this workflow by creating a flow graph application for the feature detection example shown in Fig 9.

In this example, a series of images are received. After each image is preprocessed, it is run through two different detection algorithms, A and B, and a decision is made about the image based on the results of both algorithms.

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

Before the graph can be drawn in the Flow Graph Designer, the dependencies in the application must be analyzed. This is a manual process and is not done automatically by the tool.

```
image buffer;
while ( image *i = get_next_image() ) {
    preprocess_image( i, &buffer );
    bool a = detect_with_A( &buffer );
    bool b = detect_with_B( &buffer );
    make_decision( &buffer, a, b );
    delete i;
}
```

Fig 9.Serial implementation of the feature detection algorithm

In the example from Fig 9, each image must be read by `get_next_image` sequentially. Afterwards, the function `preprocess_image` can run independently on each image. The functions `detect_with_A` and `detect_with_B` depend on the result of `preprocess_image`, but do not depend on each other. Both detection functions are also side-effect free, so they can run safely on more than one image concurrently. Finally, the function `make_decision` depends on the results of `detect_with_A` and `detect_with_B` and must be executed sequentially since it performs I/O operations.

1) Describe the graph visually

Once the dependencies are known, Flow Graph Designer makes the layout of the graph simple. Using the features described in Section III, the graph shown in Fig 10 was drawn in the canvas. A more detailed description behind the architecture of this graph and how it satisfies the dependencies in the feature detection application can be found in [6].

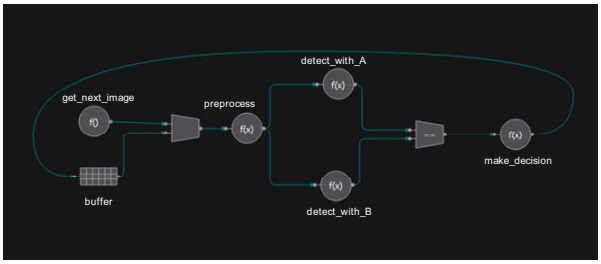
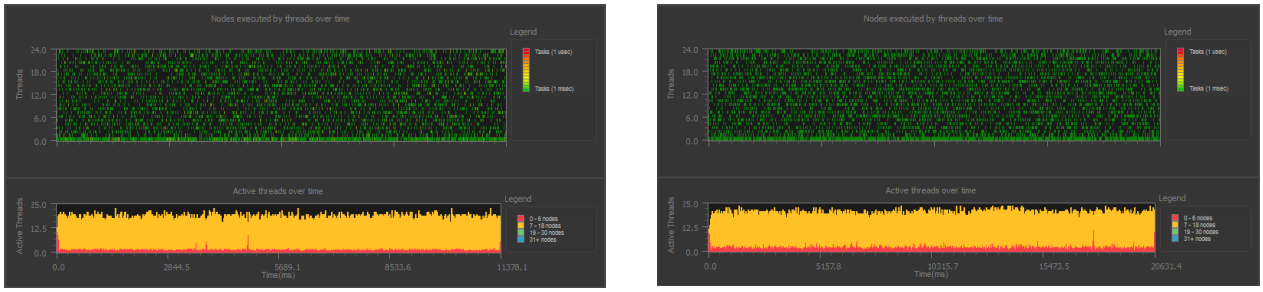


Fig 10. The flow graph as drawn in the Flow Graph Designer.

2) Run rule checks and node validation

As the graph is drawn in the tool, nodes are named, input and output types are set, and nodes are connected by edges. Flow Graph Designer assists in preventing and finding errors at this stage through node validation and rules checks. For some nodes, such an `join_nodes`, the output type is a tuple of the input types. The tool only requires that the user sets the input types and then it automatically generates the output type, preventing simple copying errors.

When the graph was fully drawn and all of the types specified, a Rule Check Analysis was run on the whole graph. Fig 11 shows an example output that has detected a type mismatch between two nodes. Once all errors were addressed, and the Rule Check Analysis returned no more errors, the C++ code for the graph was generated.



(a) Feature detection "Spin" implementation

(b) Feature detection "Compute" implementation

Fig 12. The traces of examples as displayed by Flow Graph Designer: (a) the trace of Feature Detection with the default bodies that perform a simple spin and (b) the trace of Feature Detection after the bodies are replaced with calls to the real user code.

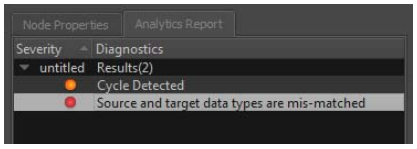


Fig 11. An example output from a Rule Check Analytic.

3) Generate C++ code and measure scalability

To confirm that the graph structure specified in the tool has the potential for reasonable scalability, it is useful to compile and run the stub code generated by Flow Graph Designer before adding the real application code to the node bodies.

One option when generating C++ stubs, is to provide *weights* for each node. If a non-zero weight is provided, the stub body that is generated for a node will spin for at least 10*weight microseconds. The node body repeatedly reads the timestamp counter and returns when the time elapsed is greater than or equal to its spin value.

To set the weights for this example appropriately, the code shown in Fig 9 was profiled by streaming 10,000 2MB images through the original serial algorithm. For each node, if the node accounted for T seconds in the profile, its weight in Flow Graph Designer was set so that its spin time multiplied by 10,000 invocations would equal T seconds.

After the weights were set for each node in the tool, the C++ code was generated and the code was modified to stream 10,000 images. To further simplify this initial "Spin" version, integers were streamed through the graph instead of real images. The application was then run with the Flow Graph Collector activated to collect a trace for 10,000 integers. Fig 12(a) shows the per-thread timelines and concurrency profile for this "Spin" version of the code.

TABLE 2 Ratio of the total time spent in each function in the two flow graph implementations of Feature Detection to the time spent in those functions in the original serial code. Ideally this ratio should be 1; a value larger than 1 indicates that the function consumes more time on average per invocation.

	Spin/Serial	Compute/Serial
get_next_image	1.03	1.91
preprocess	1.02	2.64
detect_with_A	1.02	2.06
detect_with_B	1.02	2.05
decide	1.03	1.73

Flow Graph Designer provides analytics to calculate per-node statistics from a collected trace. Column 2 of TABLE 2 shows the ratios of the times for each node as calculated by the Flow Graph Designer node statistics to the times from the original serial run. These ratios are calculated from the total time spent in the node across all threads. TABLE 2 shows that time spent in each function in the "Spin" version is within 3% of the accumulated time spent in the serial code. The total work done by each node is therefore similar to the serial version, although this work is now distributed across threads.

The total application run time of the serial version (averaged over 10 runs) was 81.7 seconds. The "Spin" implementation (averaged over 10 runs) was 11.4 seconds on 24 threads; yielding a speedup of 7.1.

A speedup of 7.1 is reasonable for this application, since it is structured as a pipeline and therefore its performance is limited by its slowest serial stage. In the serial code, `get_next_image` accounts for 13% of the total time. Since this function remains serial in the flow graph implementation, at least 13% of the original time remains serial and the maximum speedup is limited to 1/0.13 = 7.7. The speedup of the "Spin" version is therefore within 8% of this maximum.

4) Fill in the bodies with the actual function calls

Fig 12(b) shows the per-thread timelines and concurrency profile after the spin bodies are replaced with calls to the same functions used in the serial code and 2 MB images are now used in place of the integers. We refer to this version as the "Compute" implementation. While Fig 12(a) and Fig 12(b) are visually similar, the total run time and time for each node invocation is significantly inflated in Fig 12(b).

Column 3 of TABLE 2 shows the ratios of the times for each node in the "Compute" version to the times from the original serial run. Looking at the per-node statistics in Flow Graph Designer, each node consumes roughly 2x the time spent in the serial code. The total work being done by each node therefore appears to increase as we distribute the real computation across the threads. This inflation in execution time is likely due to the added communication and cache effects when 24 threads are used and the 2MB images are passed between threads and across cores and caches.

The total time for the "Compute" flow graph implementation is 20.6 seconds on 24 threads, yielding a speedup of 4. This first cut at a realistic implementation is therefore yielding only 56% of the speedup predicted by the "Spin" version. This difference warrants further investigation and it is likely that further optimization of this example to address these locality issues is possible; but this investigation and optimization is beyond the scope of this paper.

This example does, however, demonstrate that Flow Graph Designer can be used to quickly create a parallel flow graph application and get an initial estimate of its scalability limit before committing to additional work. While we did not achieve the speedup of 7.1 provided by the "Spin" version in our first full implementation; we do know that we can at most expect a speedup of 7.1 given the graph topology that we expressed.

C. *Finding performance problems in an existing application*

In the previous section, we created a new flow graph application using Flow Graph Designer (FGD). In this section, we present a scenario where FGD is used to analyze the performance of an existing flow graph application.

The application we focus on is one of the two asynchronous parallel flow graph implementations of Cholesky Factorization presented by Voss [8]. These implementations use an algorithm similar to that presented by Chandramowlishwaran et al [7]. In this section, we look at the performance of the implementation that builds a directed-acyclic graph of `continue_node` objects, where each node in the graph updates a single tile during a specific step in the algorithm. Each node invokes an Intel® Math Kernel Library (Intel® MKL) function such as `dpotf2`, `dtrsm` and `dysyr2k` to compute an update to a single tile of the matrix. Since there are many calls to these functions in this algorithm, there are many nodes in the graph. For example, to perform the factorization of a 4000x4000 matrix using 100x100 tiles, over 11,000 nodes and nearly 32,000 edges are created in the graph.

A smaller flow based implementation, which we do not use in this study, is also presented in [8]. That smaller implementation contains only one node for each type of Intel MKL call and has only five nodes in the graph. We choose to use the larger flow graph implementation to demonstrate the capability of the FGD to handle such large graphs.

In this section, we analyze the performance of Cholesky Factorization applied to a 4000x4000 matrix using a tile size of 100x100 elements.

Flow Graph Designer helps quickly identify common performance issues in a given graph. These issues may affect the overall scalability of the solution and can be categorized into three problem areas: (1) task granularities that are too small to amortize scheduling overheads, (2) time regions with low concurrency and (3) a long critical path that inhibits concurrency. We will discuss each of these issues below in the context of the Cholesky example.

1) *Task granularity*

Fig 13 shows the execution traces and the concurrency histogram for Cholesky using a tile size of 100x100. We can see in the execution trace chart, there are few tasks that are colored red and most tasks are in the range of orange to yellow. This indicates that the average task duration is much larger than the 2-3 µsecs minimum tasks size recommended by Intel® TBB. It can be safely assumed that this particular implementation of Cholesky does not suffer from performance issues that are caused by extremely fine-grained tasks.

Fig 13. Execution traces and concurrency histogram for Cholesky using tile size 100x100.

2) *Low concurrency time regions*

The task concurrency histogram for Cholesky is shown at the bottom of Fig 13. This chart shows how many tasks are being executed simultaneously at any given point in time. Looking for regions of time where the concurrency is very low and interacting with this chart by clicking on these time regions highlights the nodes that are active when the concurrencies are low. At the beginning of the histogram, we can see the concurrency gradually increase as the Cholesky algorithm hits steady state and the same can be observed at the very end where the concurrency drops off.

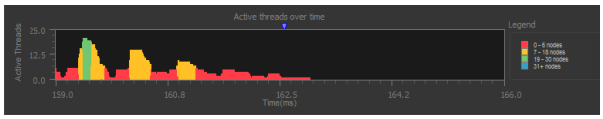
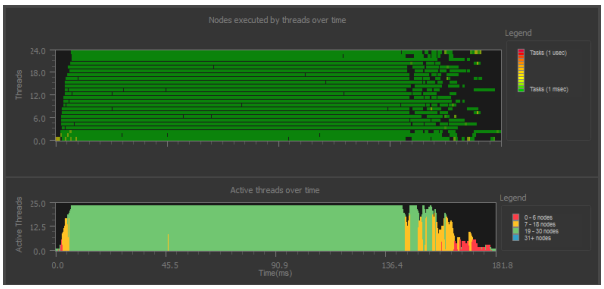


Fig 14. Concurrency histogram at the tail end of the Cholesky execution trace

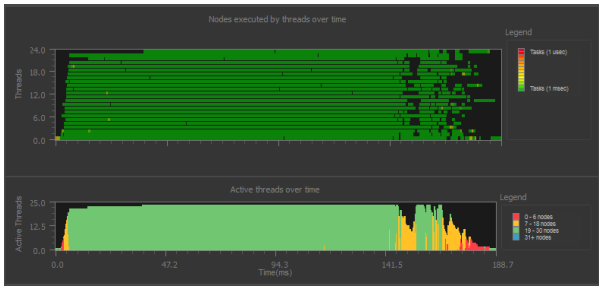
Fig 14 shows a zoomed region at the tail end of the histogram in Fig 13. In this region, the concurrency varies from 1 to 16 and the serial portions of the code seem to be dominating. By interacting with the serial or near serial portions of the histogram, FGD shows that the last few nodes in the graph were connected in a serial manner. The point in time selected is indicated by the blue marker in Fig 14.

Fig 15 shows the screenshot of the result of the interaction described above. A node is highlighted in a chain of nodes at the very end of a mostly serial sub section of the flow graph.

In this example, the serial region is a natural consequence of the wave-front algorithm used to perform the factorization and therefore cannot be easily removed.



(a) Cholesky "Actual" implementation using 24 threads



(b) Cholesky "M24" implementation using 24 threads

Fig 16. The traces of examples as displayed by Flow Graph Designer: (a) the trace of the real flow graph implementation of Cholesky run on 24 threads and (b) the trace of the Cholesky Model generated from the 24-thread trace when run on 24 threads.

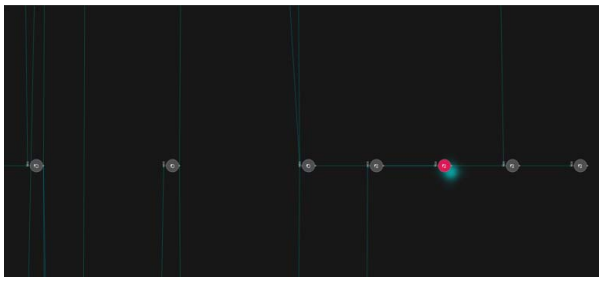


Fig 15. Sample screenshot of the Cholesky graph showing the actives nodes at the time of a selection.

3) Optimizing along the critical path

Lastly, we ran the "Critical path" analysis on the Cholesky graph to identify the critical paths in the graph and determine their costs. In this case, the tool found a single critical path that runs from the first function applied to the top-left-most tile to the final function applied to the bottom-right-most tile. The critical path length was equal to less than 12% of the total run time, indicating that the program runtime is not dominated by execution along its critical path.

The FGD tool showed for this benchmark that the task granularities were large enough to amortize overheads, that in the steady state the application shows excellent concurrency with lower concurrency during start-up and shutdown, and that the critical path does not dominate the runtime.

D. Building a C++ model from an existing application

In this final example, we show that Flow Graph Designer can also generate a C++ model from an existing flow graph application. This can be useful, for example, if one wants to debug the performance of a customer's run of an application without obtaining their data set, or if one wants to share a model of an application without exposing sensitive intellectual property (IP).

To demonstrate this, we again use the dependency graph asynchronous parallel flow graph implementation of Cholesky Factorization, which was used in Section IV.C.

To create a C++ model, the Cholesky Factorization flow graph example was compiled, run and traced using the Flow Graph Collector for both a 1-thread and 24-thread run. In each run, we factored a 4000x4000 matrix using a tile size of 200x200. Fig 16(a) shows the trace of the 24-thread run of the original application as displayed by the tool. For each trace, we loaded the trace in to Flow Graph Designer and clicked on the "Generate C++" button.

When a trace of an application's execution is available to Flow Graph Designer, the tool reconstructs the topology of the graph in the canvas and populates the "weight" for each node with the average execution time of that node from the trace. The generated C++ code therefore mimics the original application, containing the same number and types of nodes and edges. It also uses stub bodies that spin-wait for the average time measured for that node in the traced run.

We refer to the C++ model generated from the 1-thread traced run as M1 and the model generated from the 24-thread traced run as M24. After generating the C++ models, we compiled, ran and traced each model using 1 and 24 threads on the same system that was used to collect the original application traces. The traces of the modeled and original application were then loaded in to Flow Graph Designer and inspected.

Fig 16(a) shows the trace of the 24-thread run of the original application as displayed by the tool. Fig 16(b) shows the trace of the 24-thread run of the M24 model as displayed by the tool. While the traces do not match exactly, it is clear that there is a qualitative similarity between the displays.

The traces show that the M1 model matches the 1-thread run of the original application well. Likewise, the M24 model matches the 24-thread run of the original application well. However, there were significant differences when the M1 model's trace was compared against the 24-thread run of the original application, and when the M24 model's trace was compared against the performance of the 1-thread run of the original application.

TABLE 3 shows the ratio of the execution times of each model to the original application when run on 1 and 24 threads. Flow Graph Designer was also used to find and calculate the length of the critical path of each run. Rows 3 and 5 of TABLE 3 show the ratios of the critical path lengths of each

model to the critical path lengths of the original application when executed on 1 and 24 threads.

As may be expected, and in line with the results from Section IV.B, the spin-wait models do not reflect the changes in communication and memory access costs as the number of threads change. Therefore, as shown by TABLE 3, the M1 model is best able to approximate the 1-thread run and the M24 model is best able to approximate the 24-thread run. But if used to predict the execution time on 24-threads, the M1 model underestimates the total time. Likewise, if the M24 model is used to predict the execution time on 1 thread, it overestimates the total time.

TABLE 3 Ratio of the execution times of the models to the execution time of the original flow graph code. The M1 model is most accurate when run on a single thread, while the M24 model is most accurate when run on 24 threads.

	M1 / Original	M24 / Original
Times measured with 1 thread	1.05	3.45
Critical path lengths on 1 thread	1.03	5.39
Times measured with 24 threads	0.33	1.04
Critical path lengths on 24 threads	0.20	1.02

From this example one can conclude that while the C++ models generated by Flow Graph Designer from an existing application can be useful, care must be taken not to make predictions about systems or configurations that do not closely resemble the system and configurations on which the traces were collected. Still, these models can be useful to communicate performance issues and provide reproducers without the need to share IP. A C++ model of an existing application can be generated and shared, and if executed on a similar system can provide useful insight in to the topology, execution profile and scalability of the graph.

V. RELATED WORK

There are many other software tools for viewing and editing graph diagrams such as GraphViz [9], yFiles [10] and Gephi [11]. These tools support various file formats for storing the graph topology information such as the DOT language [12], Graph Modeling Language (GML) [13] and GraphML [5]. Flow Graph Designer provides graph layout algorithms that are similar to some of those used by these tools and uses GraphML to store its graph data. Unlike these tools however, FGD is specific to the Intel TBB flow graph interface and therefore is able to provide many specific features beyond just graph visualization.

There are a number of well know algorithms for calculating and displaying layouts for graphs [14][15]. Flow Graph Designer uses some of these well-known algorithms including forms of clustering and hierarchical, radial and circular graph layouts.

Flow Graph Designer provides performance analysis capabilities. There are many existing profiling and analysis tools such as Intel® VTune™ Amplifier XE [16], the Microsoft* Visual Studio* Profiling Tools [17], gprof [18] and the Tau Performance System [19]. These tools are in general statement and function based, and would not project performance events back on to the high-level parallel structure that is explicit in a flow graph application. One of the key features of FGD is that it captures performance and then relates the task profiles and concurrency histograms back to the topology of the graph that was specified using the Intel TBB interfaces.

FGD also provides some features that fall in to the domain of computer-aided software engineering. In particular, it provides features to validate individual nodes and the overall graph structure before generating C++ stubs. Many interactive development environments such as Eclipse [20] and Microsoft Visual Studio [21] provide tools or plug-ins that generate and validate code. FGD is unique in that its support is directly tailored to the Intel TBB flow graph interface and therefore provides lightweight and tightly focused features.

VI. CONCLUSION

This paper presented Flow Graph Designer (FGD), an experimental tool that helps developers visually design, model, validate, implement and analyze the performance of parallel applications that use the Intel® Threading Building Blocks (Intel® TBB) flow graph interface. Because the flow graph interface introduces a style of programming that is different than traditional loop- and task-parallel approaches, the features provided by FGD add support that is specific to this style of programming and are lacking in other tools.

In Section II, a brief overview of the Intel TBB flow graph interface was provided. In Section III, the features and workflows of Flow Graph Designer that assist in the development and analysis of flow graph applications were described.

In Section IV, three usage scenarios were presented highlighting the utility of the tool. In the first scenario, a feature detection application was drawn and validated using features built in to the tool. After the application was designed, a C++ scalability model was generated and evaluated, and a final implementation was created by filling in the C++ stubs generated by the tool.

In a second scenario, a Cholesky Factorization application was analyzed for potential scalability issues. The granularities of its tasks were analyzed, it's concurrency histogram explored and the length of its critical path was computed and compared against the application runtime.

In a final scenario, a C++ model was generated from a runtime trace of the Cholesky example. This scenario demonstrated FGD's ability to create C++ models that mimic important program characteristics without including application code that might contain sensitive IP.

Flow Graph Designer is freely available for download and evaluation at [22]. We encourage readers to download and evaluate the tool and provide feedback through the web site.

REFERENCES

[1] "Intel® Threading Building Blocks (Intel® TBB)," https://www.threadingbuildingblocks.org/, accessed April 2014.

[2] M. Voss, "The Intel Threading Building Blocks Flow Graph", Dr. Dobb's, October 2011, http://www.drdobbs.com/tools/the-intel-threading-building-blocks-flow/231900177.

[3] The OpenMP Architecture Review Board, "OpenMP Application Program Interface Version 4.0," http://openmp.org/wp/openmp-specifications/, 2013.

[4] Intel Corporation, "Intel® Cilk™ Plus," https://software.intel.com/en-us/intel-cilk-plus, 2014.

[5] U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, and M.S. Marshall, "GraphML progress report: structural layer proposal," *Proc. 9th International Symposium on Graph Drawing (GD'01)*, LNCS 2265, pp 501-512, Springer-Verlag, 2002.

[6] M. Voss, "A feature-detection example using the Intel® Threading Building Blocks flow graph", http://software.intel.com/en-us/blogs/2011/09/09/a-feature-detection-example-using-the-intel-threading-building-blocks-flow-graph, Sepetmber 2011.

[7] A. Chandramowlishwaran, K. Knobe and R. Vuduc, "Performance Evaluation of Concurrent Collections on High-Performance Multicore Computing Systems", 2010 Symposium on Parallel & Distributed Processing (IPDPS), April 2010.

[8] M. Voss, "Thoughts on the Intel® Threading Building Blocks Flow Graph and Intel® Concurrent Collections", The Fourth Annual Concurrent Collections Workshop (CnC-2012), December 2012.

[9] J. Ellson, E. Ganser, E. Koutsofios, S. North and G. Woodhull, "Graphviz and Dynagraph – static and dynanmic graph drawing tools," *Graph Drawing Software*, Springer-Verlag, pp. 127-148, 2004.

[10] "yWorks: the diagramming company," http://www.yworks.com/en/index.html , accessed April 2014.

[11] M. Bastian, S. Heymann and M. Jacomy, "Gephi: an open source software for exploring and manipulating networks," International AAAI Conference on Weblogs and Social Media, 2009.

[12] "The DOT language," http://www.graphviz.org/content/dot-language, Accessed April 2014.

[13] M. Himsolt, "GML: A portable graph file format," Tehnical Report, Universitat Passau, Germany.

[14] G. Battista, P. Eades, R. Tamassia and I. Tollis, *Graph drawing: algorithms for the visualization of graphs*, Prentice Hall, ISBN 0-13-301615-3, 1999.

[15] I. Herman, G. Melancon and M. S. Marshall, "Graph visualization and navigation in information visualization: a survey," IEEE Transactions on Visualization and Computer Graphics, Vol 6, No. 1, January - March 2000, pp 24-43.

[16] Intel Corporation, "Intel® VTune™ Amplifier XE 2013," https://software.intel.com/en-us/intel-vtune-amplifier-xe, access 2014.

[17] Microsoft, "Analyzing Application Performance by Using Profiling Tools," http://msdn.microsoft.com/en-us/library/z9z62c29.aspx, accessed 2014.

[18] S. Graham, P. Kessler and M. K. McKusick, "gprof: A Call Graph Execution Profiler," Proceedings of the ACM SIGPLAN 1982 Symposium on Compiler Construction, SIGPLAN Notices 17, 6, Boston, MA. June 1982.

[19] S. Shende and A. D. Malony, "The TAU Parallel Performance System," International Journal of High Performance Computing Applications, Volume 20 Number 2 Summer 2006. Pages 287-311.

[20] The Eclipse Foundation, "eclipse", https://www.eclipse.org/, accessed 2014

[21] Microsoft, "Visual Studio," http://msdn.microsoft.com/en-US/vstudio//, accessed 2014.

[22] M. Voss, "Flow Graph Designer," https://software.intel.com/en-us/articles/flow-graph-designer, accessed 2014.