Multi-Threaded Graph Framework Exploration for DUNE

Clifton Tyler Terwilliger

Department of Energy

Fermi National Accelerator Laboratory

**Abstract**

The Deep Underground Neutrino Experiment, DUNE for short, is Fermilab’s leading-edge neutrino experiment. This experiment will produce a tremendous amount of data that will need to be processed. Multi-threaded code will most certainly be required; however, poorly written multi-threaded code can be slow or worse, thread-unsafe. To explore a solution to this problem, a prototype framework has been developed based on directed graph processing. This prototype makes it easier to create fast and correct multi-threaded code, thus improving resource usage in a straightforward manner. Such an approach is a potential solution for meeting DUNE’s framework needs.

1. **Introduction**

The Deep Underground Neutrino Experiment, DUNE for short, is Fermilab’s leading-edge neutrino experiment. The experiment will regularly create gigabytes of data each second. In the event of a supernova burst, terabytes of data each second will be collected. This data will need to be handled in real-time, and it will require further processing to make conclusions about neutrino physics. Such a vast amount of data will require an enormous amount of processing, which must be streamlined to achieve results in a practical timeframe.

One approach to streamlining data-processing is to use multi-threading, which uses threads to allow multiple computer processors to work together. The cooperation of processors can result in a reduction of the total processing time of a computer program. However, developing multi-threaded code has significant challenges: it is often a time-consuming process that does not always yield good-quality code. Poor-quality code can be inefficient in terms of its resource utilization, such as memory usage and processing speed. In addition, code that does not properly protect its data during multi-threaded execution will result in corrupt data, which is unusable for physics analysis. On top of this, multi-threaded code is notoriously hard to debug due to needing to keep track of what the computer is simultaneously doing among its threads.

This paper outlines a potential solution for addressing these complications in the context of a data-processing framework, which handles much of the bookkeeping required for an in-depth physics analysis.

1. **Multi-Threaded Framework – A Potential Solution**

 If not addressed, the complications of writing multi-threaded code mentioned above could cause unnecessary costs to the DUNE experiment in terms of time, money, and more importantly, lost or incorrect data. The first priority is thus to develop a system that prevents lost or incorrect data. One way to do this is to provide a framework that simplifies the writing of multi-threaded code, making it easier to develop correct (and thread-safe) code. Simpler code is easier to understand, review, debug, and maintain.

 As a result of simplifying the coding process for users, the framework would reduce the code-development time required of DUNE users. Coding for the framework should be significantly easier than writing traditional code and require less time and effort from developers. The framework would also automatically take full advantage of the available hardware (e.g. CPU cores), removing that responsibility from the users, who are not experienced in such details. Ideally, the framework would be very lightweight in terms of memory and processing overhead. Even if this is not possible, the overhead incurred using a framework is preferable to poorly written multi-threaded code. This framework would enable a reduction of total processing time and processing resources for DUNE, and consequently a reduction in cost.

 These potential benefits should not be ignored. Therefore, to explore the viability of developing and subsequently using such a framework for DUNE, an exploratory framework has been developed. This framework was designed with all the above potential benefits in mind, focusing on usability, readability, simplicity, and performance.

1. **Graph Framework**

 This framework prototype was created in C++ to take advantage of the performance benefits of a compiled framework. However, despite its processing speed, C++ code can be very complicated to read and write compared to many other modern languages. Thus, one goal of the framework is to improve the readability by taking advantage of modern C++ features, such as automatic type deduction for user-defined functions (using C++17 and C++20). This allowed for simplifications of writing code intended for the framework and within the framework itself.

 To emphasize usability and simplicity, it was decided that the framework should work on a graph logic system (Example in Figure 1). The framework graph contains *nodes*, each of which are responsible for executing a user-provided function. Each node is connected via *edges* to any amount of succeeding nodes and preceding nodes. These nodes take input from the preceding nodes, process the input, and send data to every succeeding node. Fortunately, a public multi-threaded graph library does exist, known as OneTBB (see https://github.com/oneapi-src/oneTBB). This library supports directed graph processing and appears to be lightweight and highly performant. Despite some shortcomings, it was decided that the framework would be developed around OneTBB and adapted to our use case.



Figure 1

 The framework supports two main node types and an input node. The first main node type, the producer node or process node, takes an input, performs a function on that input, and returns the function’s to the succeeding nodes. The function called by the node can be any user-defined function. The second node type is the filter node. This node takes an input, then determines if the data is to be flagged as “terminated”. Flagged data is then ignored by all succeeding nodes, and just passed down the graph, without running the user-defined functions. Essentially, this procedure allows for early termination of the graph, thus reducing wasted processing on terminated data and should ultimately save on processing time. These two main nodes are automatically created based on what user-defined function is provided to the framework. Lastly, the input node takes input data from the user and sends it through the graph. However, there exist two node types that the user is unaware of: the join node, and the end-of-graph node. These two nodes are essential to the framework; however, the user is unaware of those two nodes’ existence.

1. **Framework Generated Nodes**

 First is the join node, or the dynamic join node. Due to the limitations of OneTBB, a “join node” is required anytime that a node has more than one preceding node. A join node waits until it has received an input from every input port, before passing the data on. It also repackages the data together and ensures each input has the same key. OneTBB does not support more than 10 inputs to a join node. However, DUNE is expected to need an arbitrary number of inputs to some nodes. To solve this limitation, the framework quadratically links multiple join nodes together to allow for any number of inputs. In this case, quadratically means each base join node connects to up to 10 other base join nodes that can connect to up to 10 joins nodes and so on. Additionally, the framework requires that join nodes check and see if any of the inputs have been flagged as “Terminated”; if so, the output will be flagged as such. Examples can be found in Appendix A.2 and A.3

 The other invisible node, the end-of-graph node, is much simpler: it exists simply to allow access to the data outside of the framework. The framework automatically places end-of-graph nodes after the last succeeding nodes. Whenever these nodes receive data, the data is added to a container that is accessible outside of the framework graph. This data is unsorted due to the nondeterministic execution of functions in multi-threaded code. Whilst quite basic, this node could easily be adapted to output data to file, or any other means of accessing such data.

1. **Data Access**

 DUNE requires the ability to flag data as terminated and the ability to have any input to each node. The framework requires a data identifier to be passed with the data. This identifier ensures that all inputs and outputs correspond to the same initial input throughout the execution of the graph, which can process multiple input data objects simultaneously. OneTBB requires rigid data types for each node’s inputs and outputs. To enable more flexibility, we use a universal type (called the *user data store*) for each input and output so that the rigid C++ type can be specified by the user and not required by the framework. The data passed among nodes is thus the identifier and a pointer to a user-data store object. This data storage method was chosen for two reasons. First, it’s a small data package to send around; only 32 bytes, this is for performance purposes. More importantly, it compactly specifies the identifier and whether the associated data should be terminated. The user data store contains all of the data that will be sent through the graph. It is a map of pointers to any data type, that the framework uses keys to access data. Since every node takes this as an input, we can universally use the key system for variable inputs to the user functions. More importantly, this allows the framework to add data to the user store in each graph, allowing the output of the producer nodes.

1. **Graph Construction and Execution**

 The framework needs to support the ability for any input, and any output for the user-defined functions. However, requiring the user to also specify the types for the input and output would complicate the development process. Thus, the framework automatically deduces what the inputs and outputs of each function are based solely on the function itself. The framework uses deduction guides (provided by C++17) to automatically determine the types and passes them to the constructor of the node. This is also when the node is determined to be a producer node, or filter node. A node with no output keys, but with a Boolean return type, is determined to be a filter node, otherwise, it is a producer node.

The user provides a name for each node, keys for each input and output, and the function called by the node. Then, the user specifies the connections between each node. Lastly, the user provides the input data used to launch each execution of the graph. The user may run the graph and collect their data from the end-of-graph node as needed. The framework handles the rest. An example of code for the framework can be found in appendix A.1.

Finally, once the framework has enough information from the user, the user may construct the graph. The framework connects every node to its respective previous node, and join nodes as required. Then the user may add data to be sent through the graph and call for the graph to run.

1. **Results and Future Work**

The prototype framework is found at <https://github.com/ctterwilliger/graph_framework>. Given the many possibilities created by the framework, a set of tests were used to make certain that the framework can properly works. These tests make sure every element of the framework performs as intended and are crucial to ensuring that the framework could even be a potential solution for DUNE data processing. Whilst the framework seems to be easier to code for than traditional multi-threaded code, it is more critical to ensure that framework is correct.

 Overall, the framework seems like it could be a potential solution to reduce resource usage and time for DUNE data processing. The overhead for the framework was designed to be as light as possible. Additionally, it appears to be far easier to write code for the framework than writing traditional multi-threaded code. Far more importantly, it seems to be easier to make correct multi-threaded code. While this appears to be a possible solution, this framework was more an exploration of a possible solution. The current framework would require more features to fully support DUNE data processing.

**APPENDIX**

**A.1 – Example Code**

Below is an example of code the user would give the framework. Whilst quite simple, it shows the key elements. The graph itself is a C++ class, so first is the constructor. After that, nodes can be added to the graph through calling the “add\_proccess\_node” function. Each node, requires a unique identifier (a C++ std::string), and then (optionally) keys for the inputs and outputs, and then the function itself. An input vector of user-data stores is added, the graph is built, and finally the graph can be run.

****

**A.2 – Basic Example of Framework Internal Nodes**

Below are two graphs. The top one is the graph that the user specifies, and the bottom one is the graph the framework creates given the top graph. As shown, the graph creates three node types that the user did not provide. First is the start node, which simply acts as the input to the graph. The second is the dynamic join node (represented by a j*nodename*). This is added any time a node has more than one preceding node. This is to ensure that the graph runs as the user intends. The last added node, is the end-of-graph node (EoG), which outputs the resulting data to a C++ std::vector so a user can access the output of the graph.

**** ****

**A.3 – Example of 27 nodes being joined**

 The graph below is an example of a node that has 27 prior nodes, thus requiring a join node of size 27. The graph shows the creation of 3 nodes to create a composite join node.

****