CHAPTER 3

Flow Graphs

In Chapter 2, we introduced a set of algorithms that match patterns we often come across in applications. Those are great! And we should use those whenever we can. Unfortunately, not all applications fit nicely into one of those boxes; they can be messy. When things start to get messy, we can become control freaks and try to micromanage everything or just decide to "go with the flow" and react to things as they come along. TBB lets us choose either path.

In Chapter 10, we discuss how to use tasks directly to create our own algorithms. There are both high-level and low-level interfaces to tasks, so if we use tasks directly, we can choose to become control freaks if we really want to.

In this chapter, however, we look at the Threading Building Blocks Flow Graph interface. Most of the algorithms in Chapter 2 are geared toward applications where we have a big chunk of data up front and need to create tasks to divide up and process that data in parallel. The Flow Graph is geared toward applications that react as data becomes available, or toward applications that have dependences that are more complicated than can be expressed by a simple structure. The Flow Graph interfaces have been successfully used in a wide range of domains including in image processing, artificial intelligence, financial services, healthcare, and games.

The Flow Graph interfaces let us express programs that contain parallelism that can be expressed as graphs. In many cases, these applications stream data through a set of filters or computations. We call these *data flow graphs*. Graphs can also express before-after relationships between operations, allowing us to express dependency structures that cannot be easily expressed with a parallel loop or pipeline. Some linear algebra computations, for example, Cholesky decomposition, have efficient parallel implementations that avoid heavyweight synchronization points by tracking dependencies on smaller operations instead. We call graphs that express these before-after relationships *dependency graphs*.

M. Voss, R. Asenjo, J. Reinders, *Pro TBB*, https://doi.org/10.1007/978-1-4842-4398-5_3

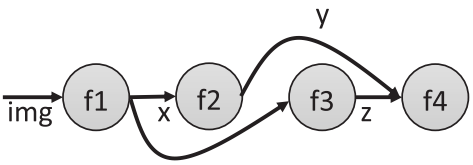
In Chapter 2, we were introduced to two generic parallel algorithms that, like a Flow Graph, do not require all of the data to be known ahead of time, `parallel_do` and `parallel_pipeline`. These algorithms are very effective when they apply; however, both of these algorithms have limitations that a Flow Graph does not have. A `parallel_do` has only a single body function that is applied to each input item as it becomes available. A `parallel_pipeline` applies a linear series of filters to input items as they flow through a pipeline. At the end of Chapter 2, we looked at a 3D stereoscopic example that had more parallelism than could be expressed by a linear series of filters. The Flow Graph APIs let us express more complicated structures than either `parallel_do` or `parallel_pipeline`.

In this chapter, we start with a discussion about why graph-based parallelism is important and then discuss the basics of the TBB Flow Graph API. After that, we explore an example of each of the two major types of flow graphs: a data flow graph and a dependency graph.

Why Use Graphs to Express Parallelism?

An application that is expressed as a graph of computations exposes information that can be effectively used at runtime to schedule its computations in parallel. We can look at the code in Figure 3-1(a) as an example.

```
while (img = getImage()) {
  x = f1(img);
  y = f2(x);
  z = f3(x);
  f4(y,z);
}
```



(a) The source code (b) A data flow graph

Figure 3-1. *An application that can be expressed as a data flow graph*

In each iteration of the while loop in Figure 3-1(a), an image is read and then passed through a series of filters: f1, f2, f3, and f4. We can draw the flow of data between these filters as shown in Figure 3-1(b). In this figure, the variables that were used to pass the data returned from each function are replaced by edges from the node that generates the value to the node(s) that consume the values.

For now, let's assume that the graph in Figure 3-1(b) captures all of the data that is shared between these functions. If so, we (and in turn a library like TBB) can infer a lot about what is legal to execute in parallel as shown in Figure 3-2.

Figure 3-2 shows the types of parallelism that can be inferred from the data flow graph representation of our small example. In the figure, we stream four images through the graph. Since there are no edges between nodes f2 and f3, they can be executed in parallel. Executing two different functions in parallel on the same data is an example of *functional parallelism* (task parallelism). If we assume that the functions are side-effect-free, that is, they do not update global states and only read from their incoming message and write to their outgoing message, then we can also overlap the processing of different messages in the graph, exploiting *pipeline parallelism*. And finally, if the functions are *thread-safe*, that is, we can execute each function in parallel with itself on different inputs, then we can also choose to overlap the execution of two different images in the same node to exploit *data parallelism*.

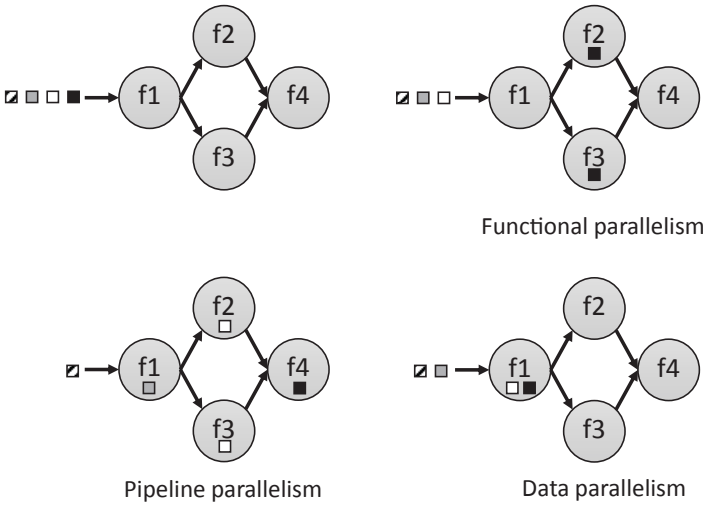


Figure 3-2. *The kinds of parallelism that can be inferred from the graph*

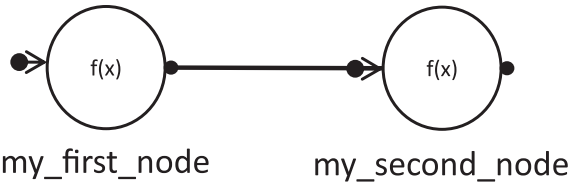
When we express our application as a graph using the TBB flow graph interface, we provide the library with the information it needs to take advantage of these different kinds of parallelism so it can map our computation to the platform hardware to improve performance.

The Basics of the TBB Flow Graph Interface

The TBB flow graph classes and functions are defined in `flow_graph.h` and are contained within the `tbb::flow` namespace. The all-encompassing `tbb.h` also includes `flow_graph.h`, so if we use that header, we do not need to include anything else.

To use a flow graph, we first create a *graph* object. We then create *nodes* to perform operations on messages that flow through the graph, such as applying user computations, joining, splitting, buffering, or reordering messages. We use *edges* to express the message channels or dependencies between these nodes. Finally, after we have assembled a graph from the graph object, node objects, and edges, we feed *messages* into the graph. Messages can be primitive types, objects, or pointers to objects. If we want to wait for processing to complete, we can use the graph object as a handle for that purpose.

Figure 3-3 shows a small example that performs the five steps needed to use a TBB Flow Graph. In this section, we will discuss each of these steps in more detail.



(a) A two node graph

```
#include <iostream>
#include <tbb/tbb.h>

void fig_3_3() {
  // step 1: construct the graph
  tbb::flow::graph g;

  // step 2: make the nodes
  tbb::flow::function_node<int, std::string> my_first_node(g,
    tbb::flow::unlimited,
    []( const int &in ) -> std::string {
      std::cout << "first node received: " << in << std::endl;
      return std::to_string(in);
    }
  );

  tbb::flow::function_node<std::string> my_second_node(g,
    tbb::flow::unlimited,
    []( const std::string &in ) {
      std::cout << "second node received: " << in << std::endl;
    }
  );

  // step 3: add edges
  tbb::flow::make_edge(my_first_node, my_second_node);

  // step 4: send messages
  my_first_node.try_put(10);

  // step 5: wait for graph to complete
  g.wait_for_all();
}
```

(b) The source-code for a two-node graph

Figure 3-3. *An example flow graph with two nodes*

Step 1: Create the Graph Object

The first step to create a flow graph is to construct a graph object. In the flow graph interface, a graph object is used for invoking whole graph operations such as waiting for all tasks related to the graph's execution to complete, resetting the state of all nodes in the graph, and canceling the execution of all nodes in the graph. When building a graph, each node belongs to exactly one graph, and edges are made between nodes in the same graph. Once we have constructed the graph, then we need to construct the nodes that implement the computations of the graph.

Step 2: Make the Nodes

The TBB flow graph interface defines a rich set of node types (Figure 3-4) that can roughly be broken into three groups: functional node types, control flow node types (includes join node types), and buffering node types. A detailed review of the interfaces provided by the graph class and the interfaces provided by all node types can be found in the "Flow Graph: nodes" section of Appendix B. It is not expected that you read these tables in detail now, but instead, that you know that you can reference them as node types are used in this and subsequent chapters.

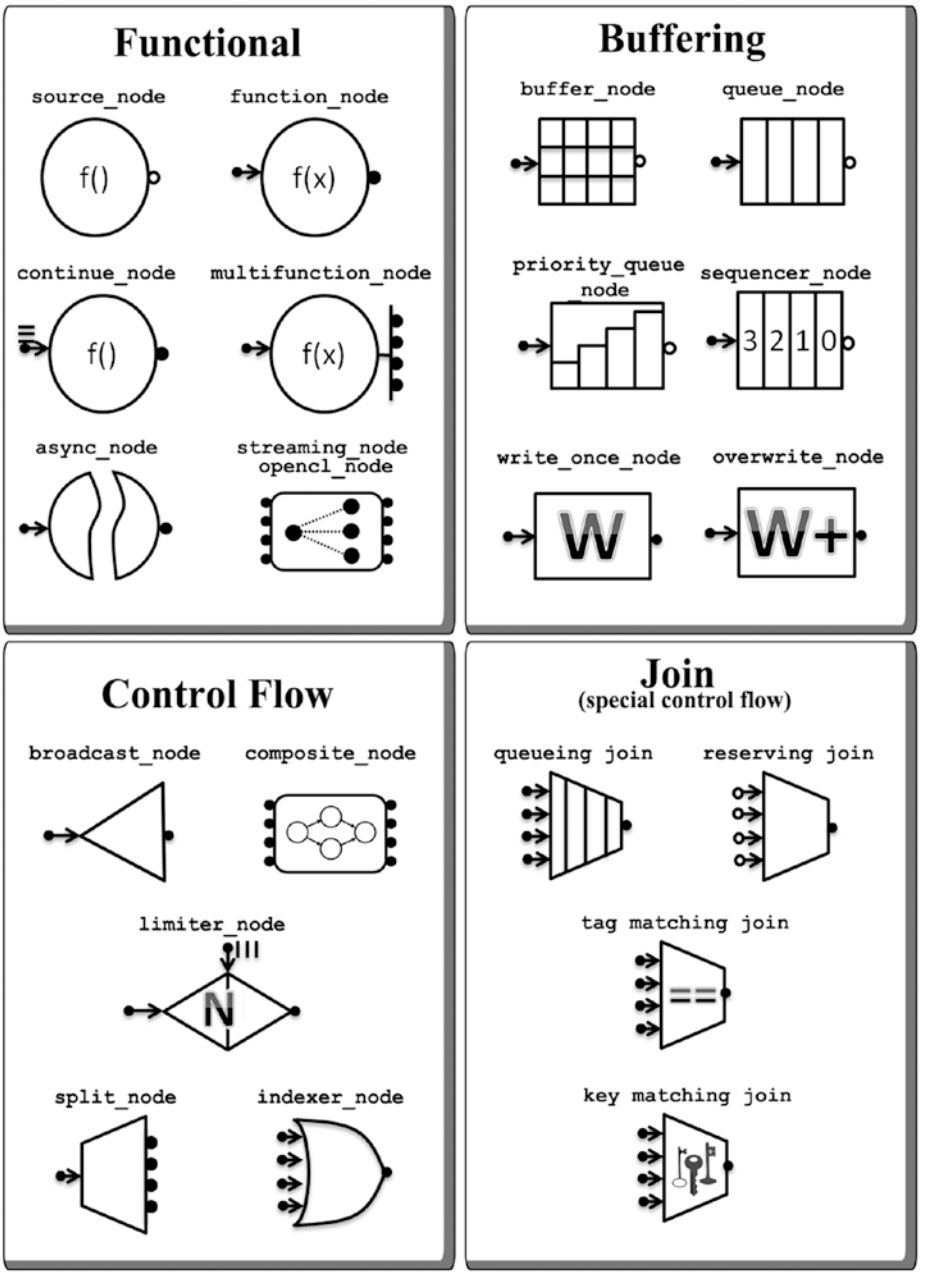


Figure 3-4. *Flow graph node types (see Chapters 3, 17, 18, 19; interface details in Appendix B)*

Like all of the functional nodes, a function_node takes a lambda expression as one of its arguments. We use these body arguments in functional nodes to provide the code we want to apply to incoming messages. In Figure 3-3, we defined the first node to receive an int value, print the value, and then convert it to a std::string, returning the converted value. This node is reproduced as follows:

```
tbb::flow::function_node<int, std::string> my_first_node(g,
  tbb::flow::unlimited,
  [](const int &in) -> std::string {
     std::cout << "first node received: " << in << std::endl;
    return std::to_string(in);
  }
);
```

Nodes are typically connected to each other by edges, but we can also explicitly send a message to a node. For example, we can send a message to my_first_node by calling try_put on it:

```
my_first_node.try_put(10);
```

This causes the TBB library to spawn a task to execute the body of my_first_node on the int message 10, resulting in output such as

```
first node received: 10
```

Unlike the functional nodes, where we provide a body argument, the control flow node types perform predefined operations that join, split, or direct messages as they flow through a graph. For example, we can create a join_node that joins together inputs from multiple input ports to create an output of type std::tuple<int, std::string, double> by providing a tuple type, the join policy, and a reference to the graph object:

```
join_node<tuple<int, std::string, double>,
          queueing> j(g);
```

This join_node, j, has three input ports and one output port. Input port 0 will accept messages of type int. Input port 1 will accept messages of type std::string. Input port 2 will accept messages of type double. There will be a single output port that broadcasts messages of type std::tuple<int, std::string, double>.

A join_node can have one of four join policies: queueing, reserving, key_matching, and tag_matching. For the queueing, key_matching, and tag_matching

policies, the `join_node` buffers incoming messages as they arrive at each of its input ports. The `queueing` policy stores incoming messages in per-port queues, joining the messages into a tuple using a first-in-first-out approach. The `key_matching` and `tag_matching` policies store the incoming messages in per-port maps and join messages based on matching keys or tags.

A reserving `join_node` does not buffer the incoming messages at all. Instead, it tracks the state of the preceding buffers – when it believes that there are messages available for each of its input ports, it tries to reserve an item for each input port. A reservation prevents any other node from consuming the item while the reservation is held. Only if the `join_node` can successfully acquire a reservation on an element for each input port does it then consume these messages; otherwise, it releases all of the reservations and leaves the messages in the preceding buffers. If a reserving `join_node` fails to reserve all of the inputs, it tries again later. We will see use cases of this reserving policy in Chapter 17.

The buffering node types buffer messages. Since the functional nodes, `function_node` and `multifunction_node`, contain buffers at their inputs and `source_node` contains a buffer at its output, buffering nodes are used in limited circumstances – typically in conjunction with a reserving `join_node` (see Chapter 17).

Step 3: Add Edges

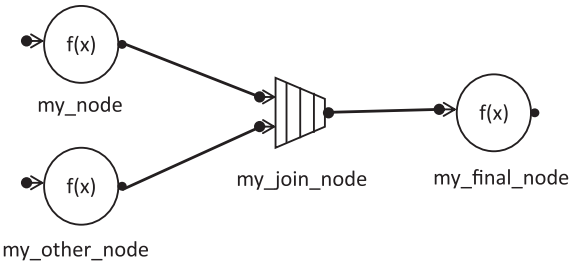
After we construct a graph object and nodes, we use `make_edge` calls to set up the message channels or dependencies:

```
make_edge(predecessor_node, successor_node);
```

If a node has more than one input port or output port, we use the `input_port` and `output_port` function templates to select the ports:

```
make_edge(output_port<0>(predecessor_node),
          input_port<1>(successor_node));
```

In Figure 3-3, we made an edge between `my_first_node` and `my_second_node` in our simple two-node graph. Figure 3-5 shows a slightly more complicated flow graph that has four nodes.



(a) A four node graph

```
void fig_3_5() {
  // step 1: construct the graph
  tbb::flow::graph g;
  // step 2: make the nodes
  tbb::flow::function_node<int, std::string> my_node(g,
    tbb::flow::unlimited,
    []( const int &in ) -> std::string {
      std::cout << "received: " << in << std::endl;
      return std::to_string(in);
    }
  );
  tbb::flow::function_node<int, double> my_other_node(g,
    tbb::flow::unlimited,
    [](const int &in) -> double {
      std::cout << "other received: " << in << std::endl;
      return double(in);
    }
  );
  tbb::flow::join_node<std::tuple<std::string, double>,
                       tbb::flow::queueing> my_join_node(g);
  tbb::flow::function_node<std::tuple<std::string, double>,
                           int> my_final_node(g,
    tbb::flow::unlimited,
    [](const std::tuple<std::string, double> &in) -> int {
      std::cout << "final: " << std::get<0>(in)
                << " and " << std::get<1>(in) << std::endl;
      return 0;
    }
  );
  // step 3: add the edges
  make_edge(my_node, tbb::flow::input_port<0>(my_join_node));
  make_edge(my_other_node, tbb::flow::input_port<1>(my_join_node));
  make_edge(my_join_node, my_final_node);
  // step 4: send messages
  my_node.try_put(1);
  my_other_node.try_put(2);
  // step 5: wait for the graph to complete
  g.wait_for_all();
}
```

(b) The source-code for a four node graph with a `join_node`.

Figure 3-5. *An example flow graph with four nodes*

The first two nodes in Figure 3-5 generate results that are joined together into a tuple by a queueing `join_node`, `my_join_node`. When the edges are made to the input ports of the `join_node`, we need to specify the port number:

```
make_edge(my_node, tbb::flow::input_port<0>(my_join_node));
make_edge(my_other_node, tbb::flow::input_port<1>(my_join_node));
```

The output of the `join_node`, a `std::tuple<std::string, double>`, is sent to `my_final_node`. We do not need to specify a port number when there is only a single port:

```
make_edge(my_join_node, my_final_node);
```

Step 4: Start the Graph

The fourth step in creating and using a TBB flow graph is to start the graph execution. There are two main ways that messages enter a graph either (1) through an explicit `try_put` to a node or (2) as the output of a `source_node`. In both Figure 3-3 and Figure 3-5, we call `try_put` on nodes to start messages flowing into the graph.

A `source_node` is constructed by default in the active state. Whenever an outgoing edge is made, it immediately starts sending messages across the edge. Unfortunately, we believe this is error prone, and so we always construct our source nodes in the inactive state, that is, pass false as the `is_active` argument. To get messages flowing after our graph is completely constructed, we call the `activate()` function on all of our inactive nodes

Figure 3-6 demonstrates how a `source_node` can be used as a replacement for a serial loop to feed messages to a graph. In Figure 3-6(a), a loop repeatedly calls `try_put` on a node `my_node`, sending messages to it. In Figure 3-6(b), a `source_node` is used for the same purpose.

The return value of a `source_node` is used like the boolean condition in a serial loop – if true, another execution of the loop body is performed; otherwise, the loop halts. Since a `source_node`'s return value is used to signal the boolean condition, it returns its output value by updating the argument provided to its body. In Figure 3-6(b), the `source_node` replaces the count loop in Figure 3-6(a).

```
void loop_with_try_put() {
  const int limit = 3;
  tbb::flow::graph g;
  tbb::flow::function_node<int> my_node(g, tbb::flow::unlimited,
    [](int i) {
      std::cout << i << std::endl;
    }
  );
  for (int count = 0; count < limit; ++count) {
    int value = count;
    my_node.try_put(value);
  }
  g.wait_for_all();
}
```

(a) using a for-loop and explicit calls to `try_put`

```
void fig_3_6() {
  tbb::flow::graph g;
  int count = 0;
  tbb::flow::source_node<int> my_src(g,
    [&count](int &i) -> bool {
      const int limit = 3;
      if (count < limit) {
        i = count++;
        return true;
      } else {
        return false;
      }
    },
    false);
  tbb::flow::function_node<int> my_node(g,
    tbb::flow::unlimited,
    [](int i) {
      std::cout << i << std::endl;
    }
  );
  tbb::flow::make_edge(my_src, my_node);
  my_src.activate();
  g.wait_for_all();
}
```

(b) using a `source_node` instead of a loop

Figure 3-6. *In (a), a loop sends the* `int` *values* `0`, `1`, *and* `2` *to a node* `my_node`. *In (b), a* `source_node` *sends the* `int` *values* `0`, `1`, *and* `2` *to the node* `my_node`.

The main advantage of using a `source_node`, instead of loop, is that it responds to other nodes in the graph. In Chapter 17, we discuss how a `source_node` can be used in conjunction with a reserving `join_node` or a `limiter_node` to control how many

messages are allowed to enter a graph. If we use a simple loop, we can flood our graph with inputs, forcing nodes to buffer many messages if they cannot keep up.

Step 5: Wait for the Graph to Complete Executing

Once we have sent messages into a graph either using `try_put` or a `source_node`, we wait for the execution of the graph to complete by calling `wait_for_all()` on the graph object. We can see these calls in Figure 3-3, Figure 3-5, and Figure 3-6.

If we build and execute the graph in Figure 3-3, we see an output like

```
first node received: 10
second node received: 10
```

If we build and execute the graph in Figure 3-5, we see an output like

```
other received: received: 21

final: 1 and 2
```

The output from Figure 3-5 looks a little jumbled, and it is. The first two function nodes execute in parallel, and both are streaming to `std::cout`. In our output, we see a combination of the two outputs jumbled together because we broke the assumption we made earlier in this chapter when we discussed graph-based parallelism – our nodes are not side-effect-free! These two nodes execute in parallel, and both affect the state of the global `std::cout` object. In this example, that's ok since this output is printed just to show the progress of the messages through the graph. But it is an important point to remember.

The final `function_node` in Figure 3-5 only executes when both values from the preceding function nodes are joined together by the `join_node` and are passed to it. This final node therefore executes by itself, and so it streams the expected final output to `std::cout`: "final: 1 and 2".

A More Complicated Example of a Data Flow Graph

In Chapter 2, we introduced an example that applied a red-cyan 3D stereoscopic effect to pairs of left and right images. In Chapter 2, we parallelized this example with a TBB `parallel_pipeline`, but in doing so admitted that we left some parallelism on the table by linearizing the pipeline stages. An example output is shown in Figure 3-7.

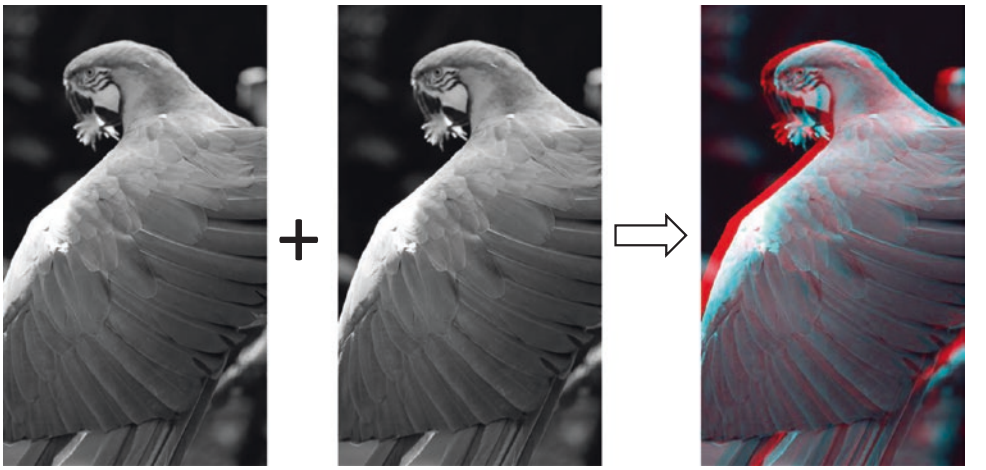


Figure 3-7. *A left and right image are used to generate a red-cyan stereoscopic image. The original photograph was taken by Elena Adams.*

Figure 3-8 shows the data and control dependencies in the serial code that was shown in Figure 2-28. The data dependencies are shown as solid lines and the control dependencies as dotted lines. From this diagram, we can see that the calls to `getLeftImage` followed by `increasePNGChannel` do not depend on the calls to `getRightImage` followed by `increasePNGChannel`. Consequently, these two series of calls can be made in parallel with each other. We can also see that `mergePNGImages` cannot proceed until `increasePNGChannel` has completed on both the left and right images. And finally, `write` must wait until the call to `mergePNGImages` is finished.

Unlike in Chapter 2, where we used a linear pipeline, using a TBB flow graph we can now more accurately express the dependencies. To do so, we need to first understand the constraints in our application that preserve correct execution. For example, each iteration of the while loop does not start until the previous iteration is complete, but this may be just a side effect of using a serial while loop. We need to determine which constraints are truly necessary.

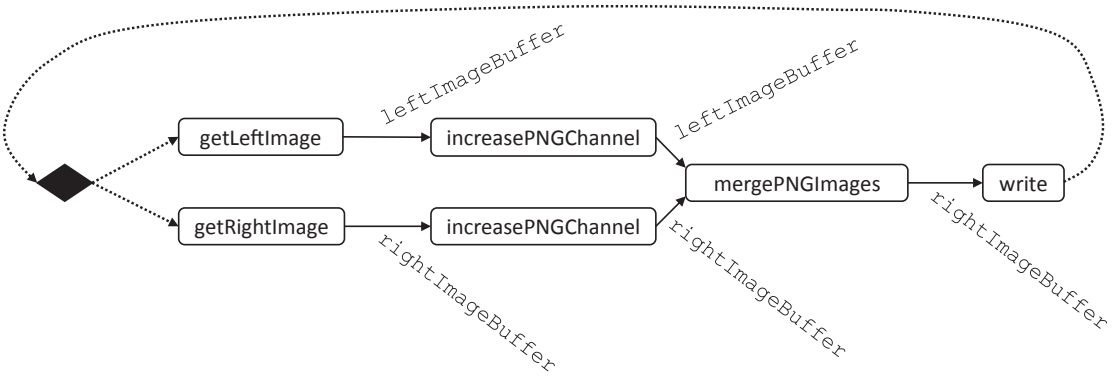


Figure 3-8. *The control and data dependencies from the code sample in Figure 2-28, where the solid lines represent data dependencies and the dotted lines represent control dependencies*

In this example, let us assume that the images represent frames that are read in order, either from a file or from a camera. Since the images must be read in order, we cannot make multiple calls to `getLeftImage` or multiple calls to `getRightImage` in parallel; these are serial operations. We can, however, overlap a call to `getLeftImage` with a call to `getRightImage` because these functions do not interfere with each other. Beyond these constraints though, we will assume that `increasePNGChannel`, `mergePNGImages`, and `write` are safe to execute on different inputs in parallel (they are both side-effect-free and thread-safe). Therefore, the iterations of the while loop cannot be executed completely in parallel, but there is some parallelism that we can exploit both within and across iterations as long as the constraints we have identified here are preserved.

Implementing the Example as a TBB Flow Graph

Now, let's step through the construction of a TBB flow graph that implements our stereoscopic 3D sample. The structure of the flow graph we will create is shown in Figure 3-9. This diagram looks different than Figure 3-8, because now the nodes represent TBB flow graph node objects and the edges represent TBB flow graph edges.

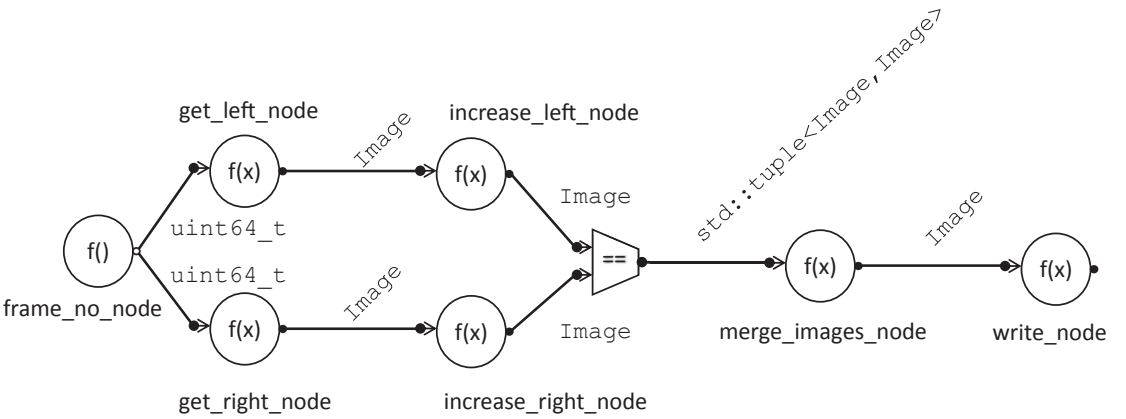


Figure 3-9. *A graph that represents the calls in Figure 2-28. The circles encapsulate the functions from Figure 2-28. The edges represent intermediate values. The trapezoid represents a node that joins messages into a two-tuple.*

Figure 3-10 shows the stereoscopic 3D example implemented using the TBB flow graph interfaces. The five basic steps are outlined in boxes. First, we create a graph object. Next, we create the eight nodes, including a `source_node`, several `function_node` instances, and a `join_node`. We then connect the nodes using calls to `make_edge`. After making the edges, we activate the source node. Finally, we wait for the graph to complete.

In the diagram in Figure 3-9, we see that `frame_no_node` is the source of inputs for the graph, and in Figure 3-10, this node is implemented using a `source_node`. As long as the body of a `source_node` continues to return `true`, the runtime library will continue to spawn new tasks to execute its body, which in turn calls `getNextFrameNumber()`.

As we noted earlier, the `getLeftImage` and `getRightImage` functions must execute serially. In the code in Figure 3-10, we communicate this constraint to the runtime library by setting the concurrency constraint for these nodes to `flow::serial`. For these nodes, we use class `function_node`. You can see more details about `function_node` in Appendix B. If a node is declared with `flow::serial`, the runtime library will not spawn the next task to execute its body until any outstanding body task is finished.

```
tbb::flow::graph g;
```

```
tbb::flow::source_node<uint64_t> frame_no_node(g,
  [] (uint64_t &frame_number) -> bool {
    if (frame_number = getNextFrameNumber())
      return true;
    else
      return false;}, false);
tbb::flow::function_node<uint64_t, Image> get_left_node(g, tbb::flow::serial,
  [] (uint64_t frame_number) -> Image {
    return getLeftImage(frame_number);});
tbb::flow::function_node<uint64_t, Image> get_right_node(g, tbb::flow::serial,
  [] (uint64_t frame_number) -> Image {
    return getRightImage(frame_number);});
tbb::flow::function_node<Image, Image> increase_left_node(g,
  tbb::flow::unlimited,
  [] (Image left) -> Image {
      increasePNGChannel(left, Image::redOffset, 10);
      return left;});
tbb::flow::function_node<Image, Image> increase_right_node(g,
  tbb::flow::unlimited,
  [] (Image right) -> Image {
      increasePNGChannel(right, Image::blueOffset, 10);
      return right;});
tbb::flow::join_node<std::tuple<Image, Image>, tbb::flow::tag_matching >
  join_images_node(g, [] (Image left) { return left.frameNumber; },
                      [] (Image right) { return right.frameNumber; } );
tbb::flow::function_node<std::tuple<Image, Image>, Image>
merge_images_node(g, tbb::flow::unlimited,
  [] (std::tuple<Image, Image> t) -> Image {
    auto &l = std::get<0>(t);
    auto &r = std::get<1>(t);
    mergePNGImages(r, l);
    return r;});
tbb::flow::function_node<Image> write_node(g, tbb::flow::unlimited,
  [] (Image img) {
    img.write();});
```

```
tbb::flow::make_edge(frame_no_node, get_left_node);
tbb::flow::make_edge(frame_no_node, get_right_node);
tbb::flow::make_edge(get_left_node, increase_left_node);
tbb::flow::make_edge(get_right_node, increase_right_node);
tbb::flow::make_edge(increase_left_node,
                     tbb::flow::input_port<0>(join_images_node));
tbb::flow::make_edge(increase_right_node,
                     tbb::flow::input_port<1>(join_images_node));
tbb::flow::make_edge(join_images_node, merge_images_node);
tbb::flow::make_edge(merge_images_node, write_node);
```

```
frame_no_node.activate();
```

```
g.wait_for_all();
```

Figure 3-10. *The stereoscopic 3D example as a TBB flow graph*

In contrast, the `increase_left_node` and the `increase_rigt_node` objects are constructed with a concurrency constraint of `flow::unlimited`. The runtime library will immediately spawn a task to execute the body of these nodes whenever an incoming message arrives.

In Figure 3-9, we see that the `merge_images_node` function needs both a right and left image. In the original serial code, we were ensured that the images would be from the same frame, because the while loop only operated on one frame at a time. In our flow graph version, however, multiple frames may be pipelined through the flow graph and therefore may be in progress at the same time. We therefore need to ensure that we only merge left and right images that correspond to the same frame.

To provide our `merge_images_node` with a pair of matching left and right images, we create the `join_images_node` with a `tag_matching` policy. You can read about `join_node` and its different policies in Appendix B. In Figure 3-10, `join_images_node` is constructed to have two input ports and to create a tuple of `Image` objects based on matching their `frameNumber` member variables. The call to the constructor now includes two lambda expressions that are used to obtain the tag values from the incoming messages on the two input ports. The `merge_images_node` accepts a tuple and generates a single merged image.

The last node created in Figure 3-10 is `write_node`. It is a `flow::unlimited` `function_node` that receives `Image` objects and calls `write` to store each incoming buffer to an output file.

Once constructed, the nodes are connected to each other using calls to `make_edge` to create the topology shown in Figure 3-9. We should note that nodes that have only a single input or output do not require a port to be specified. However, for nodes such as `join_images_node` that have multiple input ports, port accessor functions are used to pass specific ports to the `make_edge` call.

Finally, in Figure 3-10, the `frame_no_node` is activated and a call to `wait_for_all` is used to wait for the graph to complete executing.

Understanding the Performance of a Data Flow Graph

It is important to note that, unlike in some other data flow frameworks, the nodes in a TBB flow graph are not implemented as threads. Instead, TBB tasks are spawned reactively as messages arrive at nodes and concurrency limits allow. Once tasks are spawned, they are then scheduled across the TBB worker threads using the same work-stealing approach used by the TBB generic algorithms (see Chapter 9 for details about work-stealing schedulers).

There are three main factors that can limit the performance of a TBB flow graph: (1) the serial nodes, (2) the number of worker threads, and (3) the overhead from the parallel execution of TBB tasks.

Let's consider how our 3D stereoscopic graph might be mapped to TBB tasks and how these tasks might perform. Nodes `frame_no_node`, `get_left_node`, and `get_right_node` are `flow::serial` nodes. The remaining nodes are `flow::unlimited`.

Serial nodes can cause worker threads to become idle, because they limit the availability of tasks. In our stereoscopic 3D example, the images are read in order. Once each image has been read, the processing of the image can begin immediately and can be overlapped with any other work in the system. Therefore, these three serial nodes are the ones limiting task availability in our graph. If the time to read these images dominates the rest of the processing, we will see very little speedup. If, however, the processing time is much larger than the time to read the images, we may see a significant speedup.

If the image reads are not our limiting factor, the performance is then limited by the number of worker threads and the overhead of parallel execution. When we use a flow graph, we pass data between nodes that may execute on different worker threads and, likewise, processor cores. We also overlap the execution of different functions. Both the passing of data across threads and the execution of functions simultaneously on different threads can affect memory and cache behavior. We will discuss locality and overhead optimizations in more detail in Part 2 of this book.

The Special Case of Dependency Graphs

The TBB flow graph interfaces support both data flow and dependency graphs. Edges in a data flow graph are channels over which data passes between nodes. The stereoscopic 3D example that we constructed earlier in this chapter is an example of a data flow graph – `Image` objects pass over the edges from node to node in the graph.

Edges in a dependency graph represent before-after relationships that must be satisfied for a correct execution. In a dependency graph, data is passed from node to node through shared memory and is not communicated directly by messages that travel over the edges. Figure 3-11 shows a dependency graph for making a peanut butter and jelly sandwich; the edges communicate that a node cannot begin until ***all*** of its predecessors have completed.

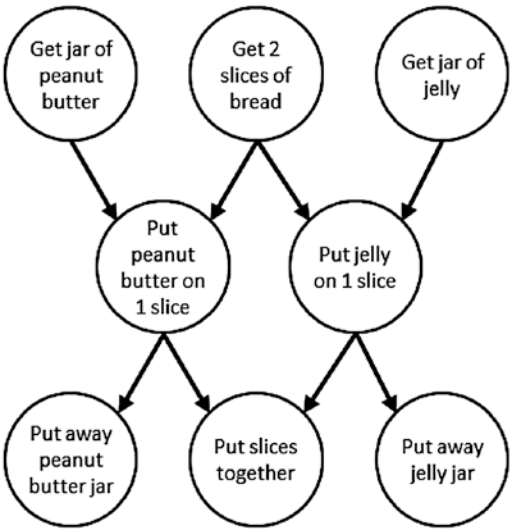


Figure 3-11. *A dependency graph for making a peanut butter and jelly sandwich. The edges here represent before-after relationships.*

To express dependency graphs using the TBB flow graph classes, we use class `continue_node` for the nodes and pass messages of type `continue_msg`. The primary difference between a `function_node` and `continue_node` is how they react to messages. You can see the details of `continue_node` in Appendix B.

When a `function_node` receives a message, it applies its body to that message – either by spawning a task immediately or by buffering the message until it is legal to spawn a task to apply the body. In contrast, a `continue_node` counts the number of messages it receives. When the number of messages it has received is equal to the number of predecessors it has, it spawns a task to execute its body and then resets its messages-received count. For example, if we were to implement Figure 3-11 using `continue_nodes`, the "Put slices together" node would execute each time it received two `continue_msg` objects, since it has two predecessors in the graph.

`continue_node` objects count messages and do not track that each individual predecessor has sent a message. For example, if a node has two predecessors, it will execute after it receives two messages, regardless of where the messages originated. This makes the overhead of these nodes much lower but also requires that dependency graphs are acyclic. Also, while a dependency graph can be executed repeatedly to completion, it is not safe to stream `continue_msg` objects into it. In both cases, when there is a cycle or if we stream items into a dependency graph, the simple counting

mechanism means that the node might mistakenly trigger because it counts messages received from the same successor when it really needs to wait for inputs from different successors.

Implementing a Dependency Graph

The steps for using a dependency graph are the same as for a data flow graph; we create a graph object, make nodes, add edges, and feed messages into the graph. The main differences are that only `continue_node` and `broadcast_node` classes are used, the graph must be acyclic, and we must wait for the graph to execute to completion each time we feed a message into the graph.

Now, let us build an example dependency graph. For our example, let's implement the same forward substitution example that we implemented in Chapter 2 using a TBB `parallel_do`. You can refer to the detailed description of the serial example in that chapter.

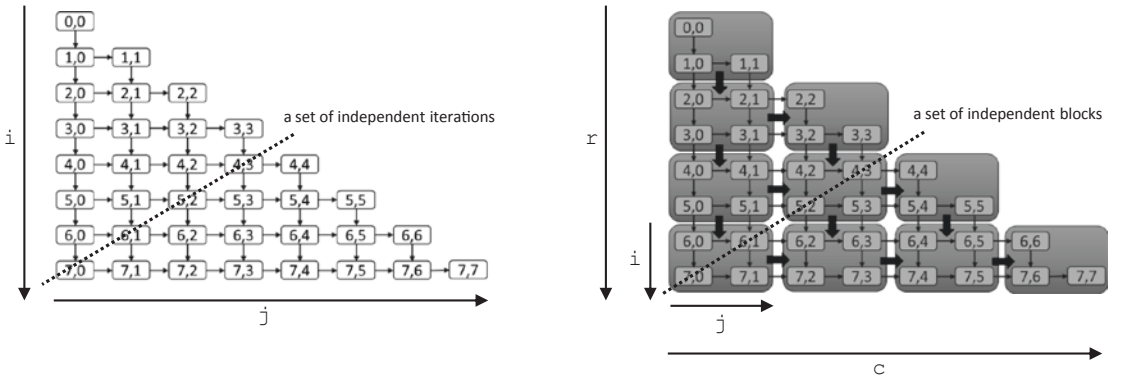
The serial tiled implementation of this example is reproduced in Figure 3-12.

```
void fig_3_12(std::vector<double> &x, const std::vector<double> &a,
              std::vector<double> &b) {
  const int N = x.size();
  const int block_size = 512;
  const int num_blocks = N / block_size;

  for ( int r = 0; r < num_blocks; ++r ) {
    for ( int c = 0; c <= r; ++c ) {
      int i_start = r*block_size, i_end = i_start + block_size;
      int j_start = c*block_size, j_max = j_start + block_size - 1;
      for (int i = i_start; i < i_end; ++i) {
        int j_end = (i <= j_max) ? i : j_max + 1;
        for (int j = j_start; j < j_end; ++j) {
          b[i] -= a[j + i*N] * x[j];
        }
        if (j_end == i) {
          x[i] = b[i] / a[i + i*N];
        }
      }
    }
  }
}
```

Figure 3-12. *The serial blocked code for a direct implementation of forward substitution. This implementation is written to make the algorithm clear – not for best performance.*

In Chapter 2, we discussed the dependencies between the operations in this example and noted, as shown again in Figure 3-13, that there is a wavefront of parallelism that can be seen diagonally across the computation. When using a `parallel_do`, we created a 2D array of atomic counters and had to manually track when each block could be safely fed to the `parallel_do` algorithm for execution. While effective, this was cumbersome and is error-prone.



(a) dependencies between iterations (b) dependencies between blocks

Figure 3-13. *The dependencies in forward substitution for a small 8 × 8 matrix. In (a), the dependencies between iterations are shown. In (b), the iterations are grouped into blocks to reduce scheduling overheads. In both (a) and (b), each node must wait for its neighbor above and its neighbor to its left to complete before it can execute.*

In Chapter 2, we noted that we might also use a `parallel_reduce` to express parallelism in this example. We can see such an implementation in Figure 3-14.

```
void fig_3_14(std::vector<double> &x, const std::vector<double> &a,
              std::vector<double> &b) {
  const int N = x.size();
  for (int i = 0; i < N; ++i) {
    b[i] -= tbb::parallel_reduce(tbb::blocked_range<int>(0,i), 0.0,
      [&a, &x, i, N] (const tbb::blocked_range<int> &b, double init)
        -> double {
        for (int j = b.begin(); j < b.end(); ++j) {
          init += a[j + i*N] * x[j];
        }
        return init;
      },
      std::plus<double>()
    );
    x[i] = b[i] / a[i + i*N];
  }
}
```

Figure 3-14. *Using a* `parallel_reduce` *to make forward substitution parallel*

However, as we can see in Figure 3-15, the main thread must wait for each `parallel_reduce` to complete before it can move on to the next one. This synchronization between the rows adds unnecessary synchronization points. For example, once block 1,0 is done, it is safe to immediately start working on 2,0, but we must wait until the fork-join `parallel_reduce` algorithm is done until we move on to that row.

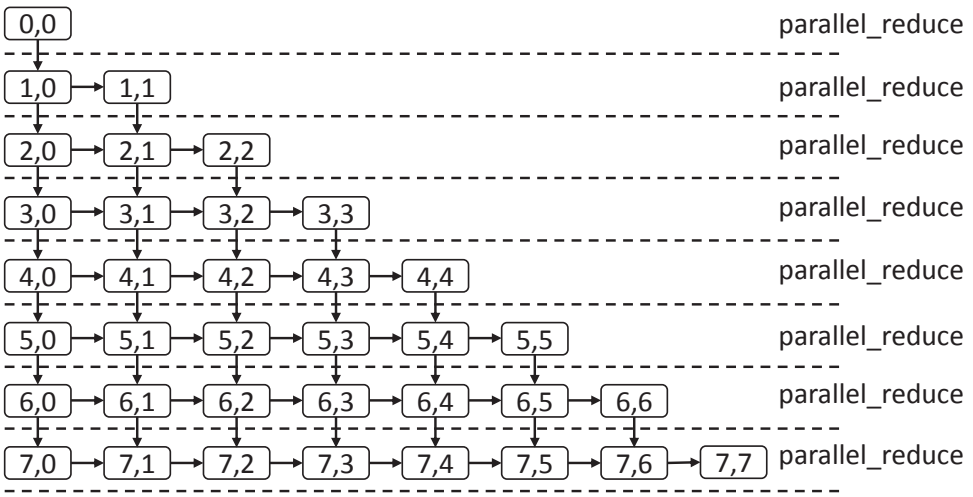


Figure 3-15. *The main thread must wait for each* `parallel_reduce` *to complete before it can move to the next* `parallel_reduce`*, introducing synchronization points*

Using a dependency graph, we simply express the dependencies directly and allow the TBB library to discover and exploit the available parallelism in the graph. We do not have to maintain counts or track completions explicitly like in the `parallel_do` version in Chapter 2, and we do not introduce unneeded synchronization points like in Figure 3-14.

Figure 3-16 shows a dependency graph version of this example. We use a `std::vector nodes` to hold a set of `continue_node` objects, each node representing a block of iterations. To create the graph, we follow the common pattern: (1) create a graph object, (2) create nodes, (3) add edges and (4) feed a message into the graph, and (5) wait for the graph to complete. However, we now create the graph structure using a loop nest as shown in Figure 3-16. The function `createNode` creates a new `continue_node` object for each block, and the function `addEdges` connects the node to the neighbors that must wait for its completion.

```
#include <iostream>
#include <memory>
#include <vector>
#include <tbb/tbb.h>

using Node = tbb::flow::continue_node<tbb::flow::continue_msg>;
using NodePtr = std::shared_ptr<Node>;

NodePtr createNode(tbb::flow::graph &g, int r, int c, int block_size,
                   std::vector<double> &x,
                   const std::vector<double> &a,
                   std::vector<double> &b);

void addEdges(std::vector<NodePtr> &nodes, int r, int c,
              int block_size, int num_blocks);

void fig_3_16(std::vector<double> &x, const std::vector<double> &a,
              std::vector<double> &b) {
  const int N = x.size();
  const int block_size = 1024;
  const int num_blocks = N / block_size;

  std::vector<NodePtr> nodes(num_blocks*num_blocks);
  tbb::flow::graph g;
  for (int r = num_blocks - 1; r >= 0; --r) {
    for (int c = r; c >= 0; --c) {
      nodes[r*num_blocks + c] =
        createNode(g, r, c, block_size, x, a, b);

      addEdges(nodes, r, c, block_size, num_blocks);
    }
  }
  nodes[0]->try_put(tbb::flow::continue_msg());

  g.wait_for_all();
}
```

Figure 3-16. *A dependency graph implementation of the forward substitution example*

In Figure 3-17, we show the implementation of the `createNode`. In Figure 3-18, we show the implementation of the `addEdges` functions.

```
NodePtr createNode(tbb::flow::graph &g, int r, int c, int block_size,
                   std::vector<double> &x,
                   const std::vector<double> &a,
                   std::vector<double> &b) {
  const int N = x.size();
  return std::make_shared<Node>(g,
    [r, c, block_size, N, &x, &a, &b]
    (const tbb::flow::continue_msg &msg) {
      int i_start = r*block_size, i_end = i_start + block_size;
      int j_start = c*block_size, j_max = j_start + block_size - 1;
      for (int i = i_start; i < i_end; ++i) {
        int j_end = (i <= j_max) ? i : j_max + 1;
        for (int j = j_start; j < j_end; ++j) {
          b[i] -= a[j + i*N] * x[j];
        }
        if (j_end == i) {
          x[i] = b[i] / a[i + i*N];
        }
      }
      return msg;
    }
  );
}
```

Figure 3-17. *The createNode function implementation*

The continue_node objects created in createNode use a lambda expression that encapsulates the inner two loops from the blocked version of forward substitution shown in Figure 3-12. Since no data is passed across the edges in a dependency graph, the data each node needs is accessed via shared memory using the pointers that are captured by the lambda expression. In Figure 3-17, the node captures by value the integers r, c, N, and block_size as well as references to the vectors x, a and b.

In Figure 3-18, the function addEdges uses make_edge calls to connect each node to its right and lower neighbors, since they must wait for the new node to complete before they can execute. When the loop nest in Figure 3-16 is finished, a dependency graph similar to the one in Figure 3-13 has been constructed.

```
void addEdges(std::vector<NodePtr> &nodes, int r, int c,
              int block_size, int num_blocks) {
  NodePtr np = nodes[r*num_blocks + c];
  if (c + 1 < num_blocks && r != c)
    tbb::flow::make_edge(*np, *nodes[r*num_blocks + c + 1]);
  if (r + 1 < num_blocks)
    tbb::flow::make_edge(*np, *nodes[(r + 1)*num_blocks + c]);
}
```

Figure 3-18. *The addEdges function implementation*

As shown in Figure 3-16, once the complete graph is constructed, we start it by sending a single `continue_msg` to the upper left node. Any `continue_node` that has no predecessors will execute whenever it receives a message. Sending a message to the top left node starts the dependency graph. Again, we use `g.wait_for_all()` to wait until the graph is finished executing.

Estimating the Scalability of a Dependency Graph

The same performance limitations that apply to data flow graphs also apply to dependency graphs. However, because dependency graphs must be acyclic, it is easier to estimate an upper bound on scalability for them. In this discussion, we use notation introduced by the Cilk project at MIT (see, e.g., *Blumofe, Joerg, Kuszmaul, Leiserson, Randall and Zhou, "Cilk: An Efficient Multithreaded Runtime System," In the Proceedings of the Principles and Practice of Parallel Programming, 1995*).

We denote the sum of the times to execute all nodes in a graph as T_1; the 1 means that this is the time it takes to execute the graph if we have only one thread of execution. And we denote the time to execute the nodes along the critical (longest) path as T_∞ since this is the minimum possible execution time even if we had an infinite number of threads available. The maximum speedup achievable through parallelism in a dependency graph is then T_1/T_∞. When executing on a platform with P processors, the execution time can never be smaller than the largest of T_1/P and T_∞.

For example, let us assume for simplicity that every node in Figure 3-13(a) takes the same amount of time to execute. We will call this time t_n. There are 36 nodes (the number of rows * the number of columns) in the graph, and so $T_1 = 36t_n$. The longest path from `0,0` to `7,7` contains 15 nodes (the number of rows + the number of columns – 1), and so for this graph $T_\infty = 15t_n$. Even if we had an infinite number of processors, the nodes along the critical path must be executed in order and cannot be overlapped. Therefore, our maximum speedup for this small 8×8 graph is $36t_n/15t_n = 2.4$. However, if we have a larger set of equations to solve, let's assume a `512×512` matrix, there would be `512×512=131,328` nodes and `512+512-1=1023` nodes along the critical path, for a maximum speedup of `131,328/1023` ≈ `128`.

When possible, if you are considering implementing a dependency graph version of a serial application, it is good practice to profile your serial code, collect the time for each would-be node, and estimate the critical path length. You can then use the simple calculation described previously to estimate the upper bound on the achievable speedup.

Advanced Topics in TBB Flow Graphs

The TBB flow graph has a rich set of nodes and interfaces, and we have really only begun to scratch this surface with this chapter. In Chapter 17, we delve deeper into the API to answer some important questions, including

- How do we control resource usage in a flow graph?
- When do we need to use buffering?
- Are there antipatterns to avoid?
- Are there effective patterns to mimic?

Also, flow graph enables asynchronous, and heterogeneous, capabilities that we will explore in Chapters 18 and 19.

Summary

In this chapter, we learned about the classes and functions in the `tbb::flow namespace` that let us develop data flow and dependency graphs. We first discussed why expressing parallelism using graphs is useful. We then learned the basics of the TBB flow graph interface, including a brief overview of the different categories of nodes that are available in the interface. Next, we built, step by step, a small data flow graph that applies a 3D stereoscopic effect to sets of left and right images. Afterward, we discussed how these nodes are mapped to TBB tasks and what the limitations are on the performance of flow graphs. Next, we looked at dependency graphs, a special case of data flow graphs, where edges communicate dependency messages instead of data messages. We also built a forward substitution example as a dependency graph and discussed how to estimate its maximum speedup. Finally, we noted some of the important advanced topics that will be covered later in this book.

The photograph used in Figures 2-28a, 2-29, and 3-7, was taken by Elena Adams, and is used with permission from the Halide project's tutorials at `http://halide-lang.org`.