CHAPTER 17

Flow Graphs: Beyond the Basics

This chapter contains some key tips on getting top performance from flow graphs in TBB. The less structured nature of the TBB flow graph APIs offers an expressiveness that requires some thinking to get the best scalable performance – we dive into details in this chapter that let us tune flow graphs to their full potential.

In Chapter 3, we introduced the classes and functions in the `tbb::flow` namespace and how they can be used to express simple data flow and dependency graphs. In this chapter, we discuss some of the more advanced questions and issues that arise when using TBB flow graphs. As in Chapter 16, much of our discussion will revolve around granularity, effective memory use, and creating sufficient parallelism. But because the flow graph APIs let us express parallelism that is less structured than the parallel algorithms described in Chapter 16, we will also discuss some dos and don'ts to be aware of when architecting a flow graph.

The section "Key FG Advice: Dos and Don'ts," starting on page 480, gives very specific rules of thumb that are invaluable when using flow graphs with TBB.

We conclude this chapter with a brief overview of the Flow Graph Analyzer (FGA), a tool available within Intel Parallel Studio XE. It has strong support for the graphical design and analysis of TBB flow graphs. While using FGA is not required when working with flow graphs, visualizing graphs during design and analysis can be very helpful. The tool is freely available to everyone, and we highly recommend it for anyone doing serious TBB flow graph work.

M. Voss, R. Asenjo, J. Reinders, *Pro TBB*, https://doi.org/10.1007/978-1-4842-4398-5_17

Optimizing for Granularity, Locality, and Parallelism

In this section, we focus on the same three concerns that drove our discussions in Chapter 16. We first look at the impact of node granularity on performance. Because flow graphs are used for less structured algorithms, we need to consider how parallelism is introduced as we discuss granularity – does the structure require a significant amount of stealing or is the generation of tasks spread well across the threads? Also, we may want to use some very small nodes in a flow graph simply because they make the design clearer – in such cases, we describe how a node with a `lightweight` execution policy can be used to limit overheads. The second issue we will address is data locality. Unlike the TBB parallel algorithms, the flow graph API does not provide abstractions like Ranges and Partitioners; instead, it is designed to enhance locality naturally. We will discuss how threads follow data to exploit locality. Our third issue is creating sufficient parallelism. Just as in Chapter 16, optimizing for granularity and locality sometimes comes at the cost of restricted parallelism – we need to be sure we walk this tightrope carefully.

Node Granularity: How Big Is Big Enough?

In Chapter 16, we discussed Ranges and Partitioners and how these can be used to ensure that the tasks created by the TBB generic algorithms are large enough to amortize scheduling overheads while still being small enough to provide enough independent work items for scalability. The TBB flow graph does not have support for Ranges and Partitioners, but we still need to be concerned about task granularity.

To see if our rule of thumb for 1 microsecond tasks that we introduced in Chapter 16 applies as well to flow graph nodes as it does to parallel algorithm bodies, we will explore a few simple microbenchmarks that capture the extremes that can exist in flow graphs. We will compare the execution times of four functions and use different amounts of work per node execution. We will refer to these functions as **Serial**, **FG loop**, **Master loop**, and **FG loop per worker**.

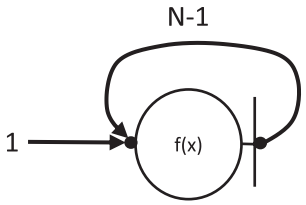
It is our belief that studying these examples (Figures 17-1 to 17-4) is critical to having an intuitive grasp of some key issues that differentiate highly scalable flow graph usage and disappointing uses of flow graph. The APIs themselves, fully documented in Appendix B, do not provide this education – we hope you will study these examples enough to grasp the concepts as we believe this will make you much better at getting the most out of using TBB flow graphs (peek at Figure 17-5 to see a quantification of the benefits on performance of understanding these!).

The **Serial** loop is our baseline and contains a for-loop that calls an active spin-wait function N times, as shown in Figure 17-1.

```
double fig_17_1(int num_trials, int N, double per_node_time) {
  tbb::tick_count t0, t1;
  for (int t = -1; t < num_trials; ++t) {
    if (!t) t0 = tbb::tick_count::now();
    for (int i = 0; i < N; ++i) {
      spinWaitForAtLeast(per_node_time);
    }
  }
  t1 = tbb::tick_count::now();
  return (t1-t0).seconds()/num_trials;
}
```

Figure 17-1. *Serial: A function that times the baseline serial loop*

The **FG loop** function is shown in Figure 17-2. This function builds a flow graph that has a single `multifunction_node` with an edge from its output to its input. A single message starts the cycle and the node then spin-waits and sends a message back to its input. The cycle repeats N-1 times. Because the node spins before sending the message back to its input, this graph is still a mostly serial loop – the bulk of the work in the body tasks will not overlap. However, because the message is sent before the body returns, there is still a small-time gap during which another thread can steal the task that the `try_put` generates. We can use this graph to see the basic overhead of the flow graph infrastructure.



(a) diagram of the serial flow graph

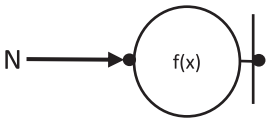
```
double fig_17_2(int num_trials, int N, double per_node_time) {
  tbb::tick_count t0, t1;
  using node_t = tbb::flow::multifunction_node<int, std::tuple<int>>;
  tbb::flow::graph g;
  node_t n{g, tbb::flow::unlimited,
    [N, per_node_time](int i, node_t::output_ports_type &p) {
      spinWaitForAtLeast(per_node_time);
      if (i+1 < N) {
        std::get<0>(p).try_put(i+1);
      }
    }
  };
  tbb::flow::make_edge(tbb::flow::output_port<0>(n), n);

  for (int t = -1; t < num_trials; ++t) {
    if (!t) t0 = tbb::tick_count::now();
    n.try_put(0);
    g.wait_for_all();
  }
  t1 = tbb::tick_count::now();
  return (t1-t0).seconds()/num_trials;
}
```

(b) source for the implementation

Figure 17-2. *FG loop: A function that times a serial flow graph*

Our next microbenchmarking function, **Master loop** shown in Figure 17-3, does not create a cycle. It instead sends all N messages to the `multifunction_node` directly from the master thread in a serial loop. Since the `multifunction_node` has unlimited parallelism and the serial for-loop will send messages very quickly, there are a lot of parallel tasks created. However, because the master thread is the only thread that calls the `try_put` method on node `n`, all body tasks are spawned into the master thread's local deque. Worker threads that participate in executing this graph will be forced to steal each

task they execute – and only after they have randomly selected the master as their victim. We can use this graph to see the behavior of a flow graph with sufficient parallelism but that requires an extreme amount of work-stealing.



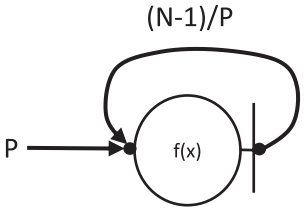
(a) diagram of a parallel flow graph with N initial messages

```
double fig_17_3(int num_trials, int N, double per_node_time) {
  tbb::tick_count t0, t1;
  using node_t = tbb::flow::multifunction_node<int, std::tuple<int>>;
  tbb::flow::graph g;
  node_t n(g, tbb::flow::unlimited,
    [N, per_node_time](int i, node_t::output_ports_type &p) {
      spinWaitForAtLeast(per_node_time);
    }
  );

  for (int t = -1; t < num_trials; ++t) {
    if (!t) t0 = tbb::tick_count::now();
    for (int i = 0; i < N; ++i) {
      n.try_put(0);
    }
    g.wait_for_all();
  }
  t1 = tbb::tick_count::now();
  return (t1-t0).seconds();
}
```

(b) source for the implementation

Figure 17-3. *Master loop: A function that submits messages only from the master thread; workers must steal every task they execute*

Finally, Figure 17-4 shows the **FG loop per worker** function. This function spreads the tasks across the master and worker threads' local deques, since once a thread has stolen its initial task, it will then spawn tasks into its own local deque. We can use this graph to see the behavior of a flow graph with a very small amount of stealing.



(a) diagram of a parallel flow graph with P initial messages

```
double fig_17_4(int num_trials, int P,
                int N_per_P, double per_node_time) {
  tbb::tick_count t0, t1;
  using node_t = tbb::flow::multifunction_node<int, std::tuple<int>>;
  tbb::flow::graph g;
  node_t n(g, tbb::flow::unlimited,
    [N_per_P, per_node_time](int i,
                             node_t::output_ports_type &p) {
      spinWaitForAtLeast(per_node_time);
      if (i+1 < N_per_P) {
        std::get<0>(p).try_put(i+1);
      }
    }
  );
  tbb::flow::make_edge(tbb::flow::output_port<0>(n), n);

  for (int t = -1; t < num_trials; ++t) {
    if (!t) t0 = tbb::tick_count::now();
    for (int p = 0; p < P; ++p) {
      n.try_put(0);
    }
    g.wait_for_all();
  }
  t1 = tbb::tick_count::now();
  return (t1-t0).seconds()/num_trials;
}
```

(b) source for the implementation

Figure 17-4. *FG loop per worker: A function that creates just enough parallelism to satisfy the number of workers. Once a worker has stolen its initial task, it will execute the remainder of its tasks from its local deque.*

Unless otherwise noted, all performance results presented in this chapter were collected on a single socket server with an Intel Xeon Processor E3-1230 with four cores supporting two hardware threads per core; the processor has a base frequency of 3.4 GHz,

a shared 8 MB L3 cache, and per-core 256 KB L2 caches. The system was running SUSE Linux Enterprise Server 12. All samples were compiled using the Intel C++ Compiler 19.0 with Threading Building Blocks 2019, using the compiler flags "`-std=c++11 -O2 -tbb`".

We ran these microbenchmarks using `N`=65,536 and spin-wait times of 100 ns, 1 us, 10 us, and 100 us. We collected their average execution times over 10 trials and present the results in Figure 17-5. From these results, we can see that when the task sizes are very small, 100 nanoseconds for example, the overhead of the flow graph infrastructure leads to degraded performance in all cases. With task sizes of at least 1 microsecond, we begin to profit from parallel execution. And by the time we reach a task size of 100 microseconds, we are able to reach close to perfect linear speedups.

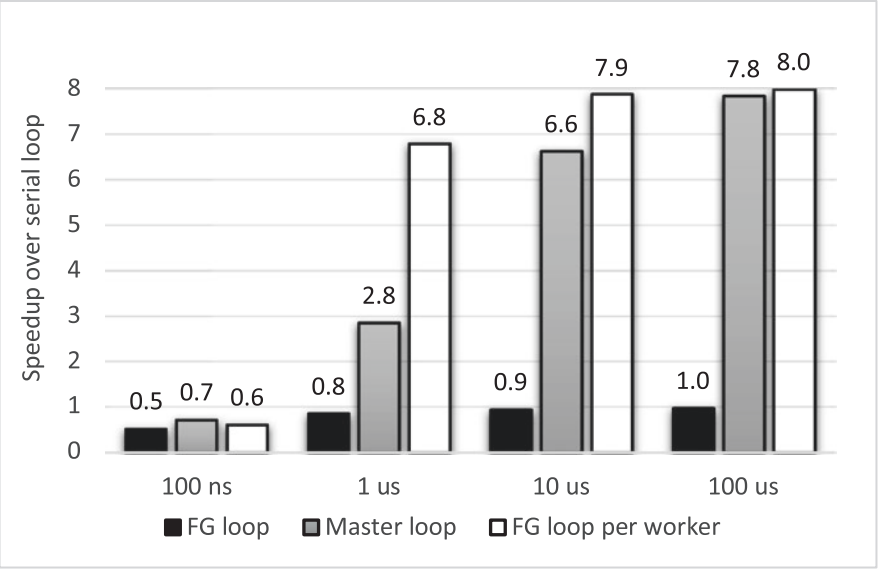


Figure 17-5. *The speedups $T_{serial}/T_{benchmark}$ for different spin wait times*

We can further understand the performance of our microbenchmarks by collecting a trace and viewing the results in Flow Graph Analyzer (FGA) – FGA is described in more detail at the end of this chapter. Figure 17-6 shows per-thread timelines for the different functions when using a spin-wait time of 1 microsecond. These timelines, which are all of the same length, show the work done by each thread over time. The gaps (in gray) in the timelines indicate when a thread is not actively executing a node's body. In Figure 17-6(a), we see the behavior of **FG loop**, which acts like a serial loop. But we can see that the small gap between the `try_put` in the body and the exit from the task allows the tasks to ping-pong between the threads since they are able to steal each task as it

is spawned. This partially explains the fairly large overheads for this microbenchmark shown in Figure 17-5. As we explain later in this chapter, most functional nodes use scheduler bypass to follow their data to the next node when possible (see the discussion on Pipelines and data locality and thread affinity in Chapter 16 for a more detailed discussion of why scheduler bypass improves cache performance). Since a `multifunction_node` puts output messages to its output ports directly inside of the body implementation, it cannot immediately follow the data to the next node using scheduler bypass – it has to finish its own body first! A `multifunction_node` therefore does not use scheduler bypass to optimize for locality. In any case, this makes the performance in Figure 17-6(a) a worst-case overhead, since scheduler bypass is not used.

In Figure 17-6(b), we see the case where the master thread is generating all of the tasks and the workers must steal each task, but tasks can be executed in parallel once they are stolen. Because the worker threads must steal each task, they are much slower at finding tasks than the master thread. The master thread is continually busy in Figure 17-6(b) – it can quickly pop a next task from its local deque – while the worker threads' timelines show gaps during which they are fighting with each other to steal their next task from the master's local deque.

Figure 17-6(c) shows the good behavior of **FG loop per worker**, where each thread is able to quickly pop its next task from its local deque. Now we see very few gaps in the timelines.

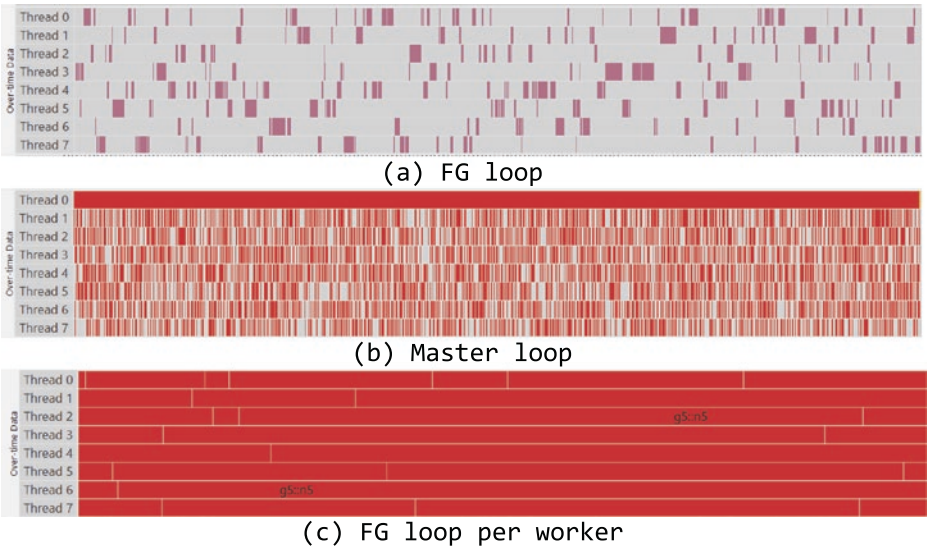


Figure 17-6. *Two millisecond regions of the timelines for each microbenchmark when using a spin wait of 1 microsecond*

Looking at these extremes of behavior and noting the performance in Figure 17-5, we feel comfortable recommending a similar rule of thumb for flow graph nodes. While a pathological case, like **Master loop**, shows a limited speedup of 2.8 with a 1 microsecond body, it still shows a speedup. If the work is more balanced, such as with **FG loop per worker,** a 1 microsecond body provides a good speedup. With these caveats in mind, we again recommend a 1 microsecond execution time as a crude guideline:

RULE OF THUMB Flow graph nodes should be at least 1 microsecond in execution time in order to profit from parallel execution. This translates to several thousand CPU cycles – if you prefer using cycles, we suggest a `10,000 cycle` rule of thumb.

Just like with the TBB algorithms, this rule *does not* mean that we must avoid nodes smaller than 1 microsecond at all costs. Only if our flow graph's execution time is dominated by small nodes do we really have a problem. If we have a mix of nodes with different execution times, the overhead introduced by the small nodes may be negligible compared to the execution time of the larger nodes.

What to Do If Nodes Are Too Small

If some of the nodes in a flow graph are smaller than the recommended 1 microsecond threshold, there are three options: (1) do nothing at all if the node does not have significant impact on the total execution time of the application, (2) merge the node with other surrounding nodes to increase granularity, or (3) use the `lightweight` execution policy.

If the node's granularity is small, but its contribution to total execution time is also small, then the node can be safely ignored; just leave it as it is. In these cases, clarity of design may trump any inconsequential efficiency gained.

If the node's granularity has to be addressed, one option is to merge it with surrounding nodes. Does the node really need to be encapsulated separately from its predecessors and successors? If the node has a single predecessor or a single successor and the same concurrency level, it might be easily combined with those nodes. If it has multiple predecessors or successors, then perhaps the operations that are performed by the node can be copied into each of the nodes. In any case, merging the nodes together can be an option if the merging does not change the semantics of the graph.

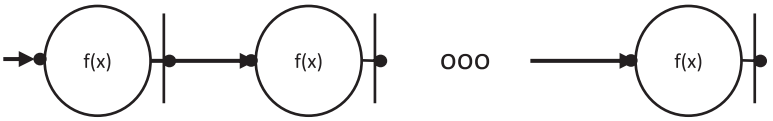
Finally, the node can be changed to use a lightweight execution policy via a template argument when the node is constructed. For example:

```
tbb::flow::function_node<int, int, lightweight> n(...);
```

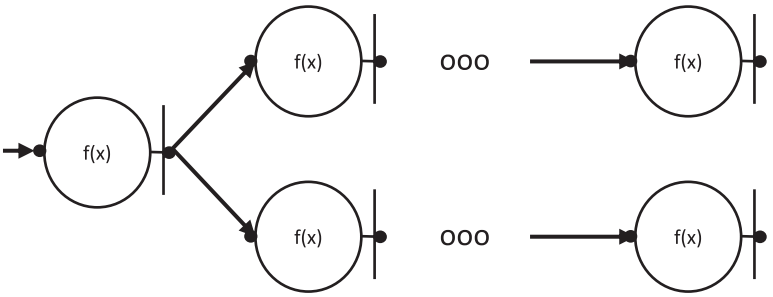
This policy indicates that the body of the node contains a small amount of work and should, if possible, be executed without the overhead of scheduling a task.

There are three lightweight policies to choose from: `queueing_lightweight`, `rejecting_lightweight`, and `lightweight`. These policies are described in detail in Appendix B. All of the functional nodes, except `source_node`, support lightweight policies. A lightweight node may not spawn a task to execute the body, but instead execute the body immediately inside of the `try_put` within the context of the calling thread. This means that the overheads of spawning are removed – but there is no opportunity for other threads to steal the task, so parallelism is restricted!

Figure 17-7 shows two simple graphs that we can use to demonstrate the benefits and risks of the lightweight policies: the first is a chain `multifunction_node` objects and the second is a `multifunction_node` object that is connected to two chains of `multifunction_node` objects.



(a) A single chain of multifunction nodes



(b) Two chains of multifunction nodes

Figure 17-7. *Flow graph used to examine the impacts of the* `lightweight` *policies*

Figure 17-8 shows the impact of using the `lightweight` policy on the graphs shown in Figure 17-7 using chains of 1000 nodes, all using the same execution policy (`lightweight` or not). We send a single message through each graph and vary the time each node spins from 0 to 1 millisecond. We should note that the single chain does not allow for any parallelism when only one message is sent, while with two chains we can achieve a maximum speedup of 2.

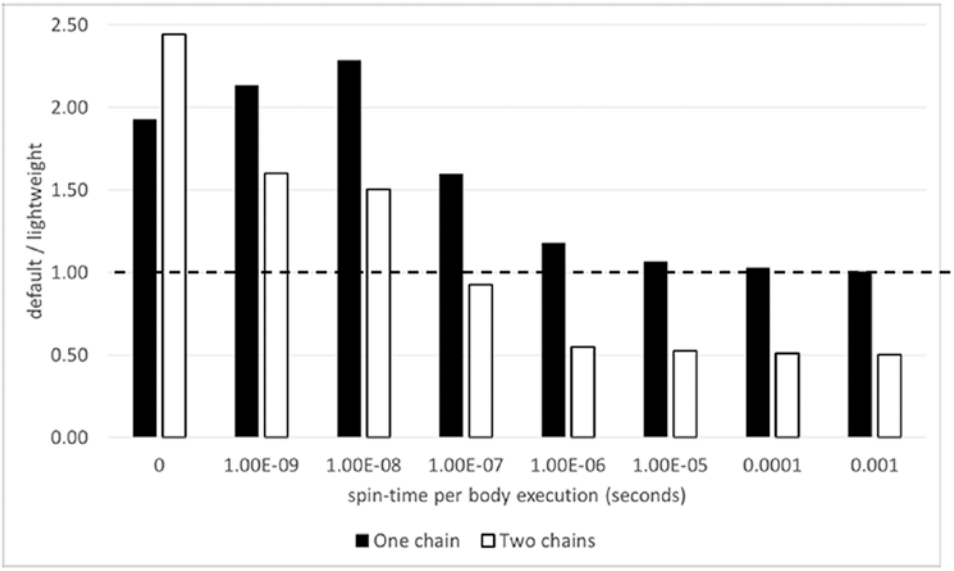


Figure 17-8. *The impact of using a lightweight policy for the one chain and two chains samples. A value greater than 1 means that the lightweight policy improved performance.*

The `lightweight` policy cannot limit parallelism for the one chain case, since there is no parallelism in this graph to begin with. We therefore see in Figure 17-8 that it improves performance for all cases, although its impact becomes less significant as the node granularity increases. For the one chain case, the ratio approaches 1.0 as the overhead of spawning tasks becomes negligible compared to the body's spin time. The two-chain case does have potential parallelism. However, if all of the nodes use a `lightweight` policy, both chains will be executed by the thread that executes the first `multifunction_node` and the potential parallelism will be eliminated. As we might expect then, as we approach our rule of thumb execution time of 1 microsecond, the benefits of the `lightweight` policy are overshadowed by the restricted parallelism. Even if the nodes spin for 0.1 microsecond, the ratio drops below 1. The ratio approaches 0.5

as the serialization of the graph results in the complete loss of our expected speedup of 2 when using two chains.

Addressing granularity issues through merging of nodes, or by using the `lightweight` policy, can decrease overheads, but as we see, they can also limit scalability. These "optimizations" can result in significant improvements, but must be applied judiciously or else they may do more harm than good.

Memory Usage and Data Locality

Unlike the TBB parallel algorithms that iterate over data structures, a flow graph passes data structures from node to node. The messages can be primitive types, objects, pointers or, in the case of a dependence graph, `tbb::flow::continue_msg` objects. For best performance, we need to consider both data locality and memory consumption. We will discuss both of these issues in this section.

Data Locality in Flow Graphs

Data passes between nodes, and when a node receives a message, it executes its body on the message as a TBB task. The task is scheduled using the same work-stealing dispatchers used by all TBB tasks. In Figure 17-6(a) when a serial loop was executed as a flow graph, we saw that a task spawned by one thread may be executed by another. We noted however that this was due in part to the microbenchmark using `multifunction_node` objects, which do not use scheduler bypass to optimize for performance.

In general, the other functional nodes, including `source_node`, `function_node`, and `continue_node`, use scheduler bypass if one of the successors can be immediately run. If the data accessed by one of these nodes fits into a data cache, then it can be reused by the same thread when it executes the successor.

Since we can benefit from locality in a flow graph, it is worth considering data size and even breaking the data into smaller pieces that can benefit from locality through scheduler bypass. For example, we can revisit the matrix transposition kernel that we used in Chapter 16 as an example to demonstrate this effect. We will now pass three pairs of `a`, `b` matrices using the `FGMsg` structure shown in Figure 17-9. You can see the serial, cache oblivious and parallel_for implementations of the matrix transposition kernel in Chapter 16 in Figure 16-6 through Figure 16-13.

Our first implementation that does not break the arrays into small pieces is also shown in Figure 17-9. The `source_node, initialize`, sends three messages, each being one of three matrix pairs. This node is connected to a single `function_node, transpose`, that has an unlimited concurrency. The `transpose` node invokes the simple, serial matrix transposition function from Chapter 16. A final node, `check`, confirms that the transposition is done correctly.

```
struct FGMsg {
  int N;
  double *a;
  double *b;
  FGMsg() : N(0), a(0), b(0) {}
  FGMsg(int _N, double *_a, double *_b) : N(_N), a(_a), b(_b) {}
};

double fig_17_9(int N, double *a[3], double *b[3]) {
  tbb::tick_count t0 = tbb::tick_count::now();
  tbb::flow::graph g;
  int i = 0;
  tbb::flow::source_node<FGMsg> initialize{g,
  [&](FGMsg &msg) -> bool {
    if (i < 3) {
      msg = {N, setArray(N, a[i]), setArray(N, b[i])};
      ++i;
      return true;
    } else {
      return false;
    }
  }, false};
  tbb::flow::function_node<FGMsg, FGMsg> transpose{g,
    tbb::flow::unlimited,
    [](const FGMsg &msg) -> FGMsg {
      serialTranspose(msg.N, msg.a, msg.b);
      return msg;
  }};
  tbb::flow::function_node<FGMsg> check{g, tbb::flow::unlimited,
    [](const FGMsg &msg) -> FGMsg {
    checkArray(msg.N, msg.b);
  }};
  tbb::flow::make_edge(initialize, transpose);
  tbb::flow::make_edge(transpose, check);
  initialize.activate();
  g.wait_for_all();
}
```

Figure 17-9. *A graph that sends a series of matrices to transpose, each of which is transposed using the simple serial matrix transposition from Chapter 16*

Our simple implementation sends the full matrices, and these are processed, in a non-cache-oblivious fashion, by `transpose`. As we might expect, this does not perform well. On our test machine, it was only 8% faster than executing the non-cache-oblivious serial implementation of our matrix transposition from Chapter 16 three times in a row, once on each pair of matrices. This isn't very surprising since the benchmark is memory bound – trying to execute multiple transpositions in parallel doesn't help much when we can't feed one transposition with the data it needs from memory. If we compare our simple flow graph to the serial cache-oblivious transposition from Chapter 16, it looks even worse, taking 2.5 times *longer* to process the three pairs of matrices when executed on our test machine. Luckily, there are many options for improving the performance of this flow graph. For example, we can use a serial cache-oblivious implementation in the `transpose` node. Or, we can use the `parallel_for` implementation from Chapter 16 that uses a `blocked_range2d` and `simple_partitioner` in the `transpose` node. We will see shortly that each of these will greatly improve our base case speedup of 1.08.

However, we might also send blocks of the matrices as messages instead of sending each pair of `a` and `b` matrices as a single big message. To do so, we extend our message structure to include a `blocked_range2d`:

```
using RType = tbb::blocked_range2d<int, int>;
struct FGTiledMsg {
  int N;
  double *a;
  double *b;
  RType r;
  FGTiledMsg() : N(0), a(0), b(0), r(0, 0, 0, 0, 0, 0) {}
  FGTiledMsg(int _N, double *_a, double *_b, const RType &_r)
    : N(_N), a(_a), b(_b), r(_r) {}
};
```

We can then construct an implementation in which the `initialize` node sends blocks of the `a` and `b` matrices as messages; sending all of the blocks from one pair of matrices before moving on to the next. Figure 17-10 shows one possible implementation. In this implementation, a stack is maintained by the `source_node` to mimic the depth-first subdivision and execution of the blocks that would come about through the recursive subdivision of ranges performed by a TBB `parallel_for`. We will not describe the implementation in Figure 17-10 in depth. Instead, we will simply note that it sends blocks instead of full matrices.

```
double fig_17_10(int N, double *a[3], double *b[3], int gs) {
  tbb::tick_count t0 = tbb::tick_count::now();
  tbb::flow::graph g;
  int i = 0;
  std::vector<RType> stack;
  stack.push_back(RType(0, N, gs, 0, N, gs));
  tbb::flow::source_node<FGTiledMsg> initialize{g,
  [&](FGTiledMsg &msg) -> bool {
    if (i < 3) {
      if (stack.empty()) {
        if (++i == 3) return false;
        stack.push_back(RType(0, N, gs, 0, N, gs));
      }
      RType r = stack.back();
      stack.pop_back();
      while (r.is_divisible()) {
        RType rhs(r, tbb::split());
        stack.push_back(rhs);
      }
      msg = {N, setBlock(r, a[i]), setTransposedBlock(r, b[i]), r};
      return true;
    } else {
      return false;
    }
  }, false};
  tbb::flow::function_node<FGTiledMsg, FGTiledMsg>
  transpose{g, tbb::flow::unlimited,
    [](const FGTiledMsg &msg) {
    double *a = msg.a, *b = msg.b;
    int N = msg.N, ie = msg.r.rows().end(), je = msg.r.cols().end();
    for (int i = msg.r.rows().begin(); i < ie; ++i) {
      for (int j = msg.r.cols().begin(); j < je; ++j) {
        b[j*N + i] = a[i*N + j];
      }
    }
    return msg;
  }};
  tbb::flow::function_node<FGTiledMsg> check{g, tbb::flow::unlimited,
    [](const FGTiledMsg &msg) {
    checkTransposedBlock(msg.r, msg.b);
  }};
  tbb::flow::make_edge(initialize, transpose);
  tbb::flow::make_edge(transpose, check);
  initialize.activate();
  g.wait_for_all();
  return total_time;
}
```

Figure 17-10. *A graph that sends a series of tiles of matrices to transpose, leveraging the blocked_range2d described in Chapter 16 (Advanced Algorithms)*

Figure 17-11 shows the speedup of several variants of matrix transposition when executed on our test machine. We can see that our first implementation, labeled "flow graph," shows the small 8% improvement. The pfor-br2d implementation is the `parallel_for` based implementation from Figure 16-11, with `blocked_range2d` and `simple_partitioner`, executed three times, once on each pair of matrices. The remaining bars all correspond to optimized flow graph versions: "flow graph + oblivious" is similar to Figure 17-9 but calls the serial cache-oblivious implementation of matrix transposition from within the body of the `transpose` node; "flow graph + pfor-br2d" uses a `parallel_for` in the `transpose` body; "tiled flow graph" is our implementation from Figure 17-10; and "tiled flow graph + pfor2d" is similar to Figure 17-10 but uses a `parallel_for` to process its tiles. The tiled flow graph from Figure 17-10 performed the best.

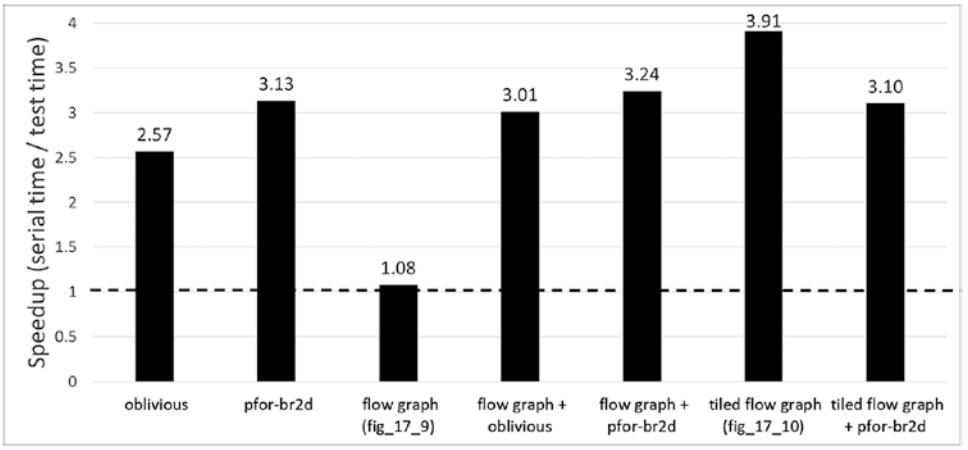


Figure 17-11. *The speedup of the different variants of matrix transposition. We use 32×32 tiles since this performed best on our test system.*

It might be surprising that the tiled flow graph version with nested `parallel_fors` did not perform as well as the tiled flow graph without nested parallelism. In Chapter 9, we claimed that we can use nested parallelism with impunity in TBB – so what went wrong? The harsh reality is that once we start tuning the performance of our TBB applications – we often need to trade away full composability for performance (see the Aspects of Composability Sidebar). In this case, the nested parallelism interfered with the cache optimizations we were carefully trying to implement. Each node was sent a tile to process that was a good fit for its data cache – with nested parallelism, we then undid this perfect fit by sharing the tile with other threads.

ASPECTS OF COMPOSABILITY

We can break down composability into three desires:

(1) Correctness (as an absolute)

(2) Ability to use (as a practical matter)

(3) Performance (as an aspiration)

In the first, we hope we can mix and match code without concerns that it will suddenly malfunction (get the wrong answer). TBB gives us this ability, and it is largely a solved problem – the one wrinkle being that nondeterministic order-of-execution will make answers vary when using finite precision math such as native floating-point arithmetic. We discuss that in Chapter 16 offering approaches to maintain the "correctness" aspects of composability in this light.

In the second, we hope that the program will not crash. This is a practical matter in many cases, because the most common problem (unbounded memory usage) could be theoretically solved with infinite sized memories. ☺ TBB largely solves this aspect of composability, giving it an advantage of programming models that do not (such as OpenMP). TBB does need more help here for the less structured flow graphs, so we discuss using `limiter_nodes` with flow graphs to keep memory usage in check – especially important in large flow graphs.

Finally, for optimal performance, we know of no general solution to full performance composability. The reality is that highly optimized code competing with other code running on the same hardware will interfere with the optimal performance of either code. This means we can benefit from manually tuning the code. Fortunately, TBB gives us control to tune, and tools like Flow Graph Analyzer help give us insights to guide our tuning. Once tuned, it is our experience that code can work well and feel composable – but the technology to blindly use code and get top performance does not exist. "Good enough" performance may happen often, but "great" requires work.

We shouldn't get too focused on the specifics of the results in Figure 17-11 – this is, after all, a single memory-bound microbenchmark. But it does make clear that we can benefit by considering the size of our nodes, not only from a granularity perspective, but also from a data locality perspective. When we moved from a naïve implementation that sent whole arrays and did not implement tuned kernels in the nodes to our more cache-aware tiled flow graph version, we saw a significant performance improvement.

Picking the Best Message Type and Limiting the Number of Messages in Flight

As we allow messages into a graph, or make copies as we split them along multiple paths through a flow graph, we consume more memory. In addition to worrying about locality, we may also need to limit memory growth.

When a message is passed to a node in a data flow graph, it may be copied into the internal buffers in that node. For example, if a serial node needs to defer the spawning of task, it holds incoming messages in a queue until it is legal to spawn a task to process them. If we pass very large objects around in our flow graph, this copying can be expensive! Therefore, when possible, it is better to pass around pointers to large objects instead of the objects themselves.

The C++11 standard introduced classes (in `namespace std`) `unique_ptr` and `shared_ptr`, which are very useful for simplifying memory management of objects passed by pointer in a flow graph. For example, in Figure 17-12, let us assume that a `BigObject` is large and slow to construct. By passing the object using a `shared_ptr`, only the `shared_ptr` is copied into the serial node `n`'s input buffer not the entire `BigObject`. Also, since a `shared_ptr` is used, each `BigObject` is automatically destroyed once it reaches the end of the graph and its reference count reaches zero. How convenient!

```
class BigObject {
  const int id;  // plus some big data not shown
public:
  BigObject() : id(-1) { }
  BigObject(int i) : id(i) { spinWaitForAtLeast(0.001); }
  BigObject(const BigObject &b) : id(b.id) {
    spinWaitForAtLeast(0.001); // simulates copy time
  }
  int get_id() const {return id;}
};

void fig_17_12() {
  tbb::flow::graph g;

  tbb::flow::function_node<std::shared_ptr<BigObject>, int>
  n(g, tbb::flow::serial,
    [](std::shared_ptr<BigObject> b) -> int {
      int id = b->get_id();
      spinWaitForAtLeast(0.01);
      return id;
    }
  );

  for (int i = 0; i < 100; ++i) {
    n.try_put(std::make_shared<BigObject>(i));
  }
  g.wait_for_all();
}
```

Figure 17-12. *Using a* `std::shared_ptr` *to avoid slow copies while simplifying memory management*

Of course, we need to be careful when we use pointers to objects. By passing pointers and not objects, multiple nodes may have access to the same object at the same time through the `shared_ptr`. This is especially true if your graph relies on functional parallelism, where the same message is broadcast to multiple nodes. The `shared_ptr` will correctly handle the increments and decrements of the reference counts, but we need to be sure that we are properly using edges to prevent any potential race conditions when accessing the object that is pointed to.

As we saw in our discussion of how nodes map to tasks, when messages arrive at functional nodes, tasks may be spawned or messages may be buffered. When designing a data flow graph, we should not forget about these buffers and tasks, and their memory footprint.

For example, let's consider Figure 17-13. There are two nodes, `serial_node` and `unlimited_node`; both contain a long spin loop. The `for` loop quickly allocates a large number of inputs for both nodes. Node `serial_node` is serial and so its internal buffer will grow quickly as it receives messages faster than its tasks complete. In contrast, node `unlimited_node` will immediately spawn tasks as each message arrives – quickly flooding the system with a very large number of tasks – many more than the number of worker threads. These spawned tasks will be buffered in the internal worker thread queues. In both cases, our graph might quickly consume a large amount of memory because they allow BigObject messages to enter the graph more quickly than they can be processed.

Our example uses an atomic counter, `bigObjectCount`, to track how many `ObjectCount` objects are currently allocated at any given time. At the end of the execution, the example prints the maximum value. When we ran the code in Figure 17-13 with `A_VERY_LARGE_NUMBER=4096`, we saw a `"maxCount == 8094"`. Both the `serial_node` and the `unlimited_node` quickly accumulate BigObject objects!

```
tbb::atomic<int> bigObjectCount;
int maxCount = 0;

class BigObject {
   const int id;
   /* And a big amount of other data */
public:
   BigObject() : id(-1) { }
   BigObject(int i) : id(i) {
     int cnt = bigObjectCount.fetch_and_increment() + 1;
     if (cnt > maxCount)
       maxCount = cnt;
   }
   BigObject(const BigObject &b) : id(b.id) { }
   virtual ~BigObject() {
     bigObjectCount.fetch_and_decrement();
   }
   int get_id() const {return id;}
};

using BigObjectPtr = std::shared_ptr<BigObject>;

void fig_17_13() {
  tbb::flow::graph g;
  tbb::flow::function_node<BigObjectPtr, BigObjectPtr>
  serial_node{g, tbb::flow::serial,
    [] (BigObjectPtr m) {
      spinWaitForAtLeast(0.0001);
      return m;
    }};
  tbb::flow::function_node<BigObjectPtr, BigObjectPtr>
  unlimited_node{g, tbb::flow::unlimited,
    [] (BigObjectPtr m) {
      spinWaitForAtLeast(0.0001);
      return m;
    }};
  bigObjectCount = 0;
  for (int i = 0; i < A_VERY_LARGE_NUMBER; ++i) {
    serial_node.try_put(std::make_shared<BigObject>(i));
    unlimited_node.try_put(std::make_shared<BigObject>(i));
  }
  g.wait_for_all();
  std::cout << "maxCount == " << maxCount << std::endl;
}
```

Figure 17-13. *An example with a serial* `function_node`*,* `serial_node`*, and an unlimited* `function_node`*,* `unlimited_node`

There are three common approaches to managing resource consumption in a flow graph: (1) use a `limiter_node`, (2) use concurrency limits, and/or (3) use a token-passing pattern.

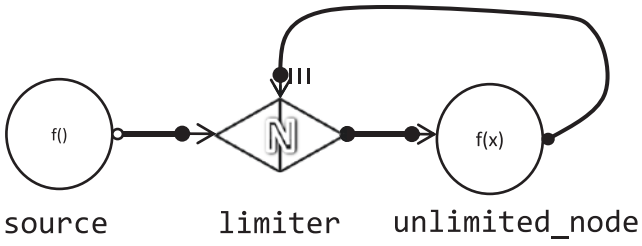
We use a `limiter_node` to set a limit on the number of messages that can flow through a given point in a graph. A subset of the interface of `limiter_node` is shown in Figure 17-14.

```
class limiter_node : public graph_node,
  public receiver<T>, public sender<T> {
public:
    limiter_node( graph &g, size_t threshold,
                  int number_of_decrement_predecessors = 0 );
    limiter_node( const limiter_node &src );

    // a continue_receiver
    implementation-dependent-type decrement;

    // receiver<T>
    typedef T input_type;
    bool try_put( const input_type &v );

    // sender<T>
    typedef T output_type;
};
```

Figure 17-14. *The subset of `limiter_node` interface used by the examples*

A `limiter_node` maintains an internal count of the messages that pass through it. A message sent to the `decrement` port on a `limiter_node` decrements the count, allowing additional messages to pass through. If the count is equal to the node's `threshold`, any new messages that arrive at its input port are rejected.

In Figure 17-15, a `source_node source` generates a large number of `BigObjects`. A `source_node` only spawns a new task to generate a message once its previously generated message is consumed. We insert a `limiter_node limiter`, constructed with a limit of 3, between `source` and `unlimited_node` to limit the number of messages that are sent to `unlimited_node`. We also add an edge from `unlimited_node` back to the `limiter_node`'s decrement port. The number of messages sent through `limiter` will now at most be 3 more than the number of messages sent back through `limiter`'s decrement port.



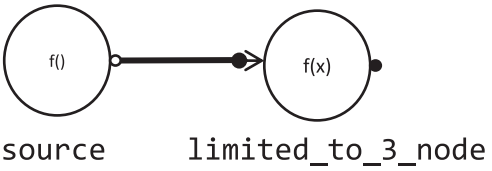
(a) diagram of the graph

```
void fig_17_15() {
  tbb::flow::graph g;

  int src_count = 0;
  tbb::flow::source_node<BigObjectPtr>
  source{g, [&src_count] (BigObjectPtr &m) -> bool {
      if (src_count < A_VERY_LARGE_NUMBER) {
        m = std::make_shared<BigObject>(src_count++);
        return true;
      }
      return false;
    }, false
  };
  tbb::flow::limiter_node<BigObjectPtr> limiter{g, 3};
  tbb::flow::function_node<BigObjectPtr,
tbb::flow::continue_msg>
  unlimited_node{g, tbb::flow::unlimited,
    [] (BigObjectPtr m) {
      spinWaitForAtLeast(0.0001);
      return tbb::flow::continue_msg();
    }
  };
  tbb::flow::make_edge(source, limiter);
  tbb::flow::make_edge(limiter, unlimited_node);
  tbb::flow::make_edge(unlimited_node, limiter.decrement);

  bigObjectCount = 0;
  maxCount = 0;
  source.activate();
  g.wait_for_all();
  std::cout << "maxCount == " << maxCount << std::endl;
}
```

(b) implementation of the graph

Figure 17-15. *Using a* `limiter_node` *to allow only three* `BigObjects` *to reach* `unlimited_node` *at a time*

We can also use the concurrency limits on nodes to limit resource consumption as shown in Figure 17-16. In the code, we have a node that can safely execute with an unlimited concurrency, but we choose a smaller number to limit the number of tasks that will be spawned concurrently.



(a) diagram of the graph

```
void fig_17_16() {
  tbb::flow::graph g;

  int src_count = 0;
  tbb::flow::source_node< BigObjectPtr > source{g,
    [&] (BigObjectPtr &m) -> bool {
      if (src_count < A_VERY_LARGE_NUMBER) {
        m = std::make_shared<BigObject>(src_count++);
        return true;
      }
      return false;
    }, false};
  tbb::flow::function_node<BigObjectPtr, BigObjectPtr,
                           tbb::flow::rejecting>
  limited_to_3_node{g, 3, [] (BigObjectPtr m) {
      spinWaitForAtLeast(0.0001);
      return m;
    }};
  tbb::flow::make_edge(source, limited_to_3_node);

  bigObjectCount = 0;
  maxCount = 0;
  source.activate();
  g.wait_for_all();
  std::cout << "maxCount == " << maxCount << std::endl;
}
```

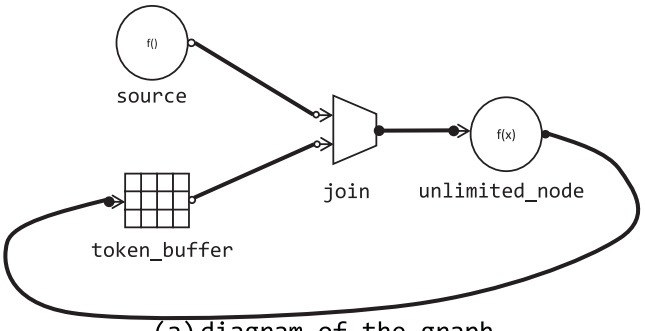
(b) implementation of the graph

Figure 17-16. *Using a* `tbb::flow::rejecting` *policy and a* `concurrency_limit` *to allow only three* `BigObjects` *to reach the* `limited_to_3_node` *at a time*

We can turn off the internal buffering for a `function_node` by constructing it with an execution policy, `flow::rejecting` or `flow::rejecting_lightweight`. The `source_node` in Figure 17-16 continues to generate new outputs only if they are being consumed.

The final common approach for limiting resource consumption in a data flow graph is to use a token-based system. As described in Chapter 2, `tbb::parallel_pipeline` algorithm uses tokens to limit the maximum number of items that will be in flight in a pipeline. We can create a similar system using tokens and a reserving `join_node` as shown in Figure 17-17. In this example, we create a `source_node source` and `buffer_node token_buffer`. These two nodes are connected to the inputs of a reserving `join_node join`. A reserving `join_node, join_node< tuple< BigObjectPtr, token_t >, flow::reserving >`, only consumes items when it can first reserve inputs at each of its ports. Since a `source_node` stops generating new messages when its previous message has not been consumed, the availability of tokens in the `token_buffer` limits the number of items that can be generated by the `source_node`. As tokens are returned to the `token_buffer` by node `unlimited_node`, they can be paired with additional messages generated by the `source`, allowing new `source` tasks to be spawned.

Figure 17-18 shows the speedup of each approach over a serial execution of the node bodies. In this figure, the spin time is 100 microseconds, and we can see that the token passing approach has a slightly higher overhead, although all three approaches show speedups close to 3, as we would expect.



(a) diagram of the graph

```
void fig_17_17() {
  using token_t = int;
  tbb::flow::graph g;
  int src_count = 0;
  tbb::flow::source_node<BigObjectPtr> source{g,
  [&] (BigObjectPtr &m) -> bool {
    if (src_count < A_VERY_LARGE_NUMBER) {
      m = std::make_shared<BigObject>(src_count++);
      return true;
    }
    return false;}, false};
  tbb::flow::buffer_node<token_t> token_buffer{g};
  tbb::flow::join_node<std::tuple<BigObjectPtr, token_t>,
                       tbb::flow::reserving> join{g};
  tbb::flow::function_node<std::tuple<BigObjectPtr,token_t>, token_t>
  unlimited_node{g, tbb::flow::unlimited,
  [] (const  std::tuple<BigObjectPtr, token_t> &m) {
    spinWaitForAtLeast(0.0001);
    return std::get<1>(m);}};
  tbb::flow::make_edge(source, tbb::flow::input_port<0>(join));
  tbb::flow::make_edge(token_buffer, tbb::flow::input_port<1>(join));
  tbb::flow::make_edge(join, unlimited_node);
  tbb::flow::make_edge(unlimited_node, token_buffer);
  bigObjectCount = 0; maxCount = 0;
  for (token_t i = 0; i < 3; ++i) token_buffer.try_put(i); // fill it
  source.activate();
  g.wait_for_all();
  std::cout << "maxCount == " << maxCount << std::endl;
}
```

(b) implementation of the graph

Figure 17-17. *A token passing pattern uses tokens and a* `tbb::flow::reserving` *join_node to limit the items that can reach node* `unlimited_node`

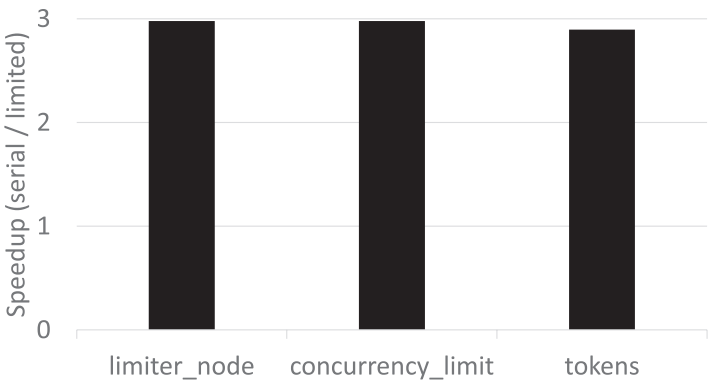


Figure 17-18. *All three approaches limit the speedup since only three items are allowed into node n at a time*

In Figure 17-18, we use `int` as the token type. In general, we can use any type as a token, even large objects or pointers. For example, we could use `BigObjectPtr` objects as the tokens if we want to recycle `BigObject` objects instead of allocating them for each new input.

Task Arenas and Flow Graph

Both implicit and explicit task arenas impact the behavior of TBB tasks and the TBB generic parallel algorithms. The arena in which tasks are spawned controls which threads can participate in executing the tasks. In Chapter 11, we saw how we can use implicit and explicit arenas to control the number of threads that participate in executing parallel work. In Chapters 12–14, we saw that explicit task arenas can be used with `task_sheduler_observer` objects to set the properties of threads as they join arenas. Because of the impact of task arenas on available parallelism and data locality, in this section, we take a closer look at how task arenas mix with flow graphs.

The Default Arena Used by a Flow Graph

When we construct a `tbb::flow::graph` object, the graph object captures a reference to the arena of the thread that constructed the object. Whenever a task is spawned to execute work in the graph, the tasks are spawned in this arena, not in the arena of the thread that caused the task to be spawned.

Why?

Well, TBB flow graphs are less structured than TBB parallel algorithms. TBB algorithms use fork-join parallelism and the behavior of TBB task arenas matches this pattern well – each master thread has its own default arena and so if different master threads execute algorithms concurrently, their tasks are isolated from each other in different task arenas. But with a TBB flow graph, there may be one or more master threads explicitly putting messages into the same graph. If the tasks related to these interactions are spawned in each master thread's arena, some tasks from a graph would be isolated from other tasks from the same graph. This is very likely not the behavior we would like.

So instead, all tasks are spawned into a single arena, the arena of the thread that constructed the graph object.

Changing the Task Arena Used by a Flow Graph

We can change the task arena used by a graph by calling the graph's `reset()` function. This reinitializes the graph, including recapturing the task arena. We demonstrate this in Figure 17-19 by constructing a simple graph with one `function_node` that prints the number of slots in the arena in which its body task executes. Since the main thread constructs the graph object, the graph will use the default arena, which we initialize with eight slots.

```
void fig_17_19() {
  tbb::task_scheduler_init init{8};
  tbb::task_arena a2{2};
  tbb::task_arena a4{4};

  tbb::flow::graph g;
  tbb::flow::function_node<std::string> f{g, tbb::flow::unlimited,
    [](const std::string &str) {
      int P = tbb::this_task_arena::max_concurrency();
      std::cout << str << " : " << P << std::endl;
    }
  };

  std::cout << "Without reset:" << std::endl;
  f.try_put("default");
  g.wait_for_all();
  a2.execute( [&]() {
    f.try_put("a2");
    g.wait_for_all();
  } );
  a4.execute( [&]() {
    f.try_put("a4");
    g.wait_for_all();
  } );

  std::cout << "With reset:" << std::endl;
  f.try_put("default");
  g.wait_for_all();
  a2.execute( [&]() {
    g.reset();
    f.try_put("a2");
    g.wait_for_all();
  } );
  a4.execute( [&]() {
    g.reset();
    f.try_put("a4");
    g.wait_for_all();
  } );
}
```

Figure 17-19. *Using* `graph::reset` *to change the task arena used by a graph*

In the first three calls to `n.try_put` in Figure 17-19, we do not reset that graph g, and we can see that the tasks execute in the default arena with eight slots.

```
Without reset:
default : 8
a2 : 8
a4 : 8
```

But in the second set of calls, we call reset to reinitialize the graph, and the node executes first in the default arena, then in `arena a2`, and finally in `arena a4`.

```
With reset:
default : 8
a2 : 2
a4 : 4
```

Setting the Number of Threads, Thread-to-Core Affinities, etc.

Now that we know how to associate task arenas with flow graphs, we can use all of the performance tuning optimizations described in Chapters 11–14 that rely on task arenas. For example, we can use task arenas to isolate one flow graph from another. Or, we can pin threads to cores for a particular task arena using a `task_scheduler_observer` object and then associate that arena with a flow graph.

Key FG Advice: Dos and Don'ts

The flow graph API is flexible – maybe too flexible. When first working with flow graph, the interface can be daunting since there are so many options. In this section, we provide several dos and don'ts that capture some of our experience when using this high-level interface. However, just like with our rule of thumb for node execution time, these are just suggestions. There are many valid patterns of usage that are not captured here, and we're sure that some of the patterns we say to avoid may have valid use cases. We present these best-known methods, but your mileage may vary.

Do: Use Nested Parallelism

Just like with a pipeline, a flow graph can have great scalability if it uses parallel (`flow::unlimited`) nodes but can have limited scalability if it has serial nodes. One way to increase scaling is to use nested parallel algorithms inside of TBB flow graph nodes. TBB is all about composability, so we should use nested parallelism when possible.

Don't: Use Multifunction Nodes in Place of Nested Parallelism

As we have seen throughout this book, the TBB parallel algorithms such as `parallel_for` and `parallel_reduce` are highly optimized and include features like Ranges and Partitioners that let us optimize performance even more. We have also seen that the flow graph interface is very expressive – we can express graphs that include loops and use nodes like `multifunction_node` to output many messages from each invocation. We should therefore be on the lookout for cases where we create patterns in our graphs that are better expressed using nested parallelism. One simple example is shown in Figure 17-20.

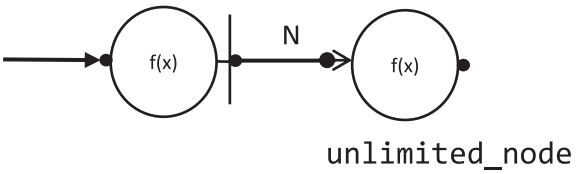


Figure 17-20. *A `multifunction_node` that sends many messages for each message it receives. This pattern may be better expressed as a nested `parallel_for` loop.*

In Figure 17-20, for each message that the `multifunction_node` receives, it generates many output messages that flow into a `function_node` with unlimited concurrency. This graph will act a lot like a parallel loop, with the `multifunction_node` acting as the control loop and the `function_node` as the body. But it will require a lot of stealing to distribute the work like the Master loop from Figures 17-3 and 17–5. While there may be valid uses of this pattern, it is likely more efficient to use a highly optimized parallel loop algorithm instead. This entire graph might be collapsed into a single node that contains a nested `parallel_for`, for example. Of course, whether or not this replacement is possible or desirable depends on the application.

Do: Use `join_node`, `sequencer_node`, or `multifunction_node` to Reestablish Order in a Flow Graph When Needed

Because a flow graph is less structured than a simple pipeline, we may sometimes need to establish an ordering of messages at points in the graph. There are three common approaches for establishing order in a data flow graph: use a key-matching `join_node`, use a `sequencer_node`, or use a `multifunction_node`.

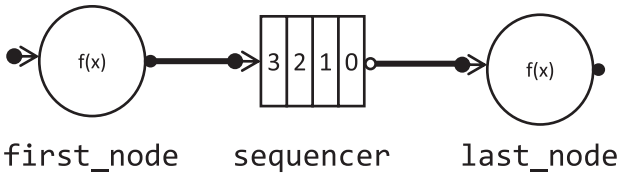
For example, in Chapter 3, the parallelism in our stereoscopic 3D flow graph allowed the left and right images to arrive out of order at the `mergeImageBuffersNode`. In that example, we ensured that the correct two images were paired together as inputs to the `mergeImageBuffersNode` by using a tag-matching `join_node`. A tag-matching `join_node` is a type of key-matching `join_node`. By using this `join_node` type, inputs can arrive in different orders at the two input ports but will still be properly matched based on their tag or key. You can find more information on the different join policies in Appendix B.

Another way to establish order is to use a `sequencer_node`. A `sequencer_node` is a buffer that outputs messages in sequence order, using a user-provided body object to obtain the sequence number from the incoming message.

In Figure 17-21, we can see a three-node graph, with nodes `first_node`, `sequencer`, and `last_node`. We use a `sequencer_node` to reestablish the input order of the messages before the final serial output node `last_node`. Because `function_node first_node` is unlimited, its tasks can finish out of order and send their output as they complete. The `sequencer_node` reestablishes the input order by using the sequence number assigned when each message was originally constructed.

If we execute a similar example without a sequencer node and `N=10`, the output is scrambled as the messages pass each other on their way to `last_node`:

```
9 no sequencer
8 no sequencer
7 no sequencer
0 no sequencer
1 no sequencer
2 no sequencer
6 no sequencer
5 no sequencer
4 no sequencer
3 no sequencer
```



(a) diagram of the graph

```
struct Message {
  size_t my_seq_no;
  std::string my_string;
  Message(int i) : my_seq_no(i), my_string(std::to_string(i)) { }
};

using MessagePtr = std::shared_ptr<Message>;

void fig_17_21() {
  const int N = 10;
  tbb::flow::graph g;
  tbb::flow::function_node<MessagePtr, MessagePtr>
  first_node{g, tbb::flow::unlimited, [] (MessagePtr m) {
    m->my_string += " with sequencer";
    return m;
  }};
  tbb::flow::sequencer_node<MessagePtr>
  sequencer(g, [] (MessagePtr m) {
    return m->my_seq_no;
  });
  tbb::flow::function_node<MessagePtr, int, tbb::flow::rejecting>
  last_node{g, tbb::flow::serial, [] (MessagePtr m) {
      std::cout << m->my_string << std::endl;
      return 0;
  }};
  tbb::flow::make_edge(first_node, sequencer);
  tbb::flow::make_edge(sequencer, last_node);

  for (int i = 0; i < N; ++i)
    first_node.try_put(std::make_shared<Message>(i));
  g.wait_for_all();
}
```

(b) implementation of the graph

Figure 17-21. *A* `sequencer_node` *is used to ensure that the messages print in the order dictated by their* `my_seq_no` *member variables*

When we execute the code in Figure 17-21, we see the output:

```
0 with sequencer
1 with sequencer
2 with sequencer
3 with sequencer
4 with sequencer
5 with sequencer
6 with sequencer
7 with sequencer
8 with sequencer
9 with sequencer
```

As we can see, a `sequencer_node` can reestablish the order of the messages, but it does require us to assign the sequence number and also to provide a body to the `sequencer_node` that can obtain that number from an incoming message.

A final approach to establishing order is to use a serial `multifunction_node`. A `multifunction_node` can output zero or more messages on any of its output ports for a given input message. Since it is not forced to output a message for each incoming message, it can buffer incoming messages and hold them until some user-defined ordering constraint is met.

For example, Figure 17-22 shows how we can implement a `sequencer_node` using a `multifunction_node` by buffering incoming messages until the next message in sequencer order has arrived. This example assumes that at most `N` messages are sent to a node `sequencer` and that the sequence numbers start at `0` and are contiguous up to `N-1`. Vector `v` is created with `N` elements initialized as empty `shared_ptr` objects. When a message arrives at `sequencer`, it is assigned to the corresponding element of `v.` Then starting at the last sent sequence number, each element of `v` that has a valid message is sent and the sequence number is incremented. For some incoming messages, no output message will be sent; for others, one or more messages may be sent.

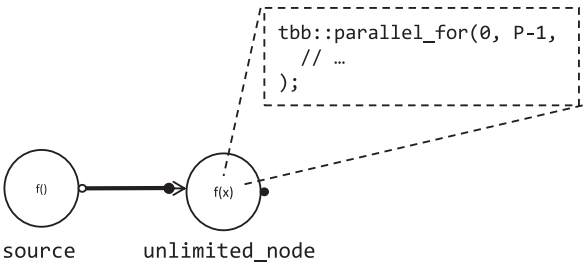
```
using MFNSequencer =
  tbb::flow::multifunction_node<MessagePtr,
std::tuple<MessagePtr>>;
using MFNPorts = typename MFNSequencer::output_ports_type;
int seq_i = 0;
std::vector<MessagePtr> v{N, MessagePtr{}};
MFNSequencer sequencer{g, tbb::flow::serial,
[&seq_i, &v](MessagePtr m, MFNPorts &p) {
  v[m->my_seq_no] = m;
  while (seq_i < N && v[seq_i].use_count()) {
    std::get<0>(p).try_put(v[seq_i++]);
  }
}};
```

Figure 17-22. *A `multifunction_node` is used to implement a `sequencer_node`*

While Figure 17-22 shows how a `multifunction_node` can be used to reorder messages by sequence order, in general, any user-defined ordering or bundling of messages can be used.

Do: Use the `Isolate` Function for Nested Parallelism

In Chapter 12, we talked about how we may sometimes need to create isolation for performance or correctness reasons when using TBB algorithms. The same is true for flow graphs, and as with the generic algorithms, this can be especially true with nested parallelism. The implementation of the graph in Figure 17-23 shows a simple graph with nodes `source` and `unlimited_node`, and nested parallelism inside node `unlimited_node`. A thread may moonlight (see Chapter 12) while waiting for the nested `parallel_for` loop in node `unlimited_node` to complete, and pick up another instance of node `unlimited_node`. The node `unlimited_node` prints "`X started by Y`", where `X` is the node instance number and `Y` is the thread id.



(a) diagram of the graph

```
void fig_17_23() {
  int P = tbb::task_scheduler_init::default_num_threads();
  tbb::concurrent_vector<std::string> trace;
  double spin_time = 1e-3;
  tbb::flow::graph g;

  int src_cnt = 0;
  tbb::flow::source_node<int> source{g,
    [&src_cnt, P, spin_time](int &i) -> bool {
       if (src_cnt < P) {
         i = src_cnt++;
         spinWaitForAtLeast(spin_time);
         return true;
       }
       return false;
    }, false};
  tbb::flow::function_node<int>
  unlimited_node(g, tbb::flow::unlimited,
    [&trace, P, spin_time](int i) {
      int tid = tbb::this_task_arena::current_thread_index();
      trace.push_back(std::to_string(i) + " started by "
                      + std::to_string(tid));
      tbb::parallel_for(0, P-1, [spin_time](int i) {
        spinWaitForAtLeast((i+1)*spin_time);
      });
      trace.push_back(std::to_string(i) + " completed by "
                      + std::to_string(tid));
    });

  tbb::flow::make_edge(source, unlimited_node);
  source.activate();
  g.wait_for_all();

  for (auto s : trace) std::cout << s << std::endl;
}
```

(b) implementation of the graph

Figure 17-23. *A graph with nested parallelism*

On our test system with eight logical cores, one output showed that our thread 0 was so bored it pick up not just one, but three different instances of `unlimited_node`, while waiting for its first `parallel_for` algorithm to finish as shown in Figure 17-24.

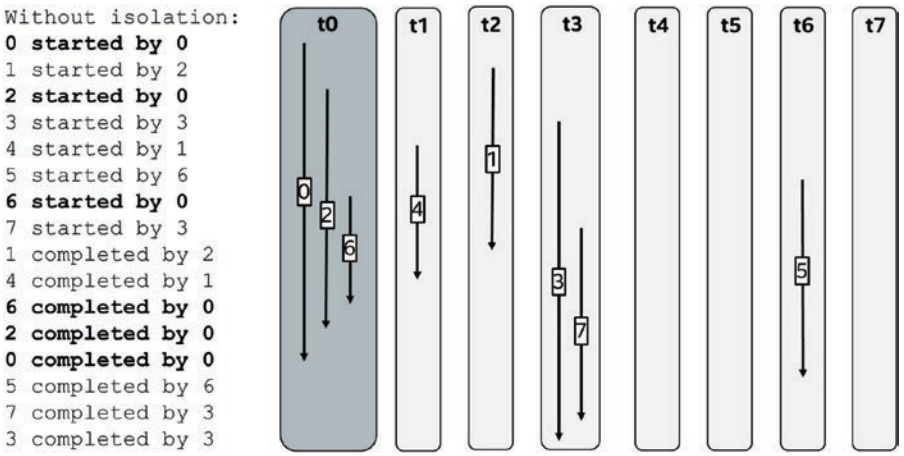


Figure 17-24. *An output from the example in Figure 17-23 is shown on the left, with a diagram showing the overlapped executions on the right. Thread 0 participates in the execution of three different node invocations concurrently.*

As we discussed in Chapter 12, moonlighting is typically benign, which is the case here since we're not computing anything real. But as we highlighted in our previous discussions about isolation, this behavior is not always benign and can lead to correctness issues, or decreased performance, in some cases.

We can address moonlighting in a flow graph just as we did with general tasks in Chapter 12, with the `this_task_arena::isolate` function or with explicit task arenas. For example, instead of calling the `parallel_for` directly in the node body, we can invoke it inside of an isolate call:

```
tbb::this_task_arena::isolate([P,spin_time]() {
  tbb::parallel_for(0, P-1, [spin_time](int i) {
    spinWaitForAtLeast((i+1)*spin_time);
  });
});
```

After changing our code to use this function, we see that the threads no longer moonlight and each thread stays focused on a single node until that node is complete as shown in Figure 17-25.

```
Without isolation:
0 started by 0
1 started by 6
2 started by 1
3 started by 5
0 completed by 0
4 started by 7
5 started by 4
1 completed by 6
6 started by 6
2 completed by 1
3 completed by 5
7 started by 5
5 completed by 4
4 completed by 7
6 completed by 6
7 completed by 5
```

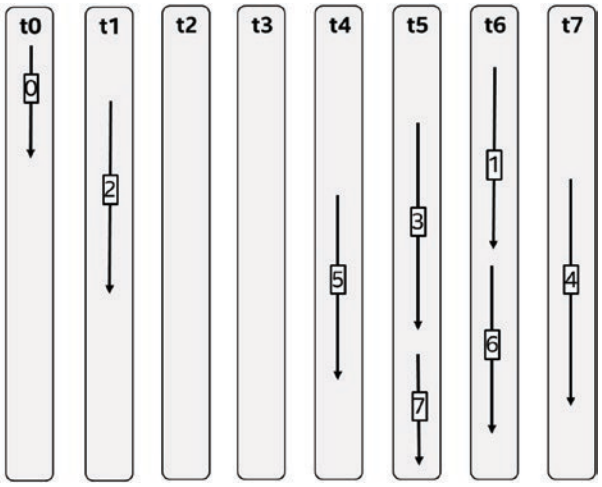


Figure 17-25. *None of the nodes execute different node invocations concurrently*

Do: Use Cancellation and Exception Handling in Flow Graphs

In Chapter 15, we discussed task cancellation and exception handling when using TBB tasks in general. Since we are already familiar with this topic, we will only highlight the flow graph related aspects in this section.

Each Flow Graph Uses a Single `task_group_context`

A flow graph instance spawns all of its tasks into a single task arena, and it also uses a single `task_group_context` object for all of these tasks. When we instantiate a graph object, we can pass in an explicit `task_group_context` to the constructor:

```
tbb::task_group_context tgc;
tbb::flow::graph g{tgc};
```

If we don't pass one to the constructor, a default object will be created for us.

Canceling a Flow Graph

If we want to cancel a flow graph, we cancel it using the `task_group_context`, just as we would with the TBB generic algorithms.

```
tgc.cancel_group_excution();
```

And just as with TBB algorithms, the tasks that have already started will complete but no new tasks related to the graph will start. As described in Appendix B, there is also a helper function in the graph class that lets us check the status of a graph directly:

```
if (g.is_cancelled()) {
  std::cout << "My graph was cancelled!" << std::endl;
}
```

If we need to cancel a graph, but do not have a reference to its `task_group_context`, we can get one from within the task:

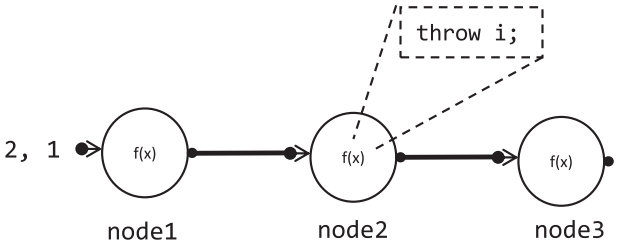
```
tbb::task::self().cancel_group_execution();
```

Resetting a Flow Graph After Cancellation

If a graph is canceled, whether directly or due to an exception, we need to reset the graph, `g.reset()`, before we can use it again. This resets the state of the graph – clearing internal buffers, putting the edges back into their initial states, and so on. See Appendix B for more details.

Exception Handling Examples

To learn about how exceptions work with a flow graph, let's look at the implementation of the graph in Figure 17-26. This figure provides a small, three-node graph that throws an exception in its second node, `node2`.



(a) diagram of the graph

```
void fig_17_26() {
  tbb::flow::graph g;

  tbb::flow::function_node<int, int> node1(g,
tbb::flow::serial,
    [](int i) { return i; }
  );
  tbb::flow::function_node<int, int> node2(g,
tbb::flow::serial,
    [](int i) {
      throw i;
      return i;
    }
  );
  tbb::flow::function_node<int, int> node3(g,
tbb::flow::serial,
    [](int i) { return i; }
  );
  tbb::flow::make_edge(node1,node2);
  tbb::flow::make_edge(node2,node3);
  node1.try_put(1);
  node2.try_put(2);
  g.wait_for_all();
}
```

(b) implementation of the graph

Figure 17-26. *A flow graph that throws an exception in one of its nodes*

If we execute this example, we get an exception (hopefully this did not come as a surprise):

```
terminate called after throwing an instance of 'int'
```

Since we didn't handle the exception, it propagates to the outer scope and our program terminates. We can, of course, modify the implementation of our node `node2`, so that it catches the exception within its own body, as shown in Figure 17-27.

```
tbb::flow::function_node<int, int> node2(g, tbb::flow::serial,
  [](int i) {
    try {
      throw i;
    } catch (int j) {
      std::cout << "Caught " << j << std::endl;
    }
    return i;
  }
);
```

Figure 17-27. *A flow graph that throws an exception in one of its nodes*

If we make this change, our example will run to completion, printing out the "Caught" messages, in no particular order:

```
Caught 2
Caught 1
```

So far, none of this is very exceptional (pun intended); it's just how exceptions should work.

The unique part of exception handling in a flow graph is that we can catch exceptions at the call to the graph's `wait_for_all` function, as shown in Figure 17-28.

```
try {
  g.wait_for_all();
} catch (int j) {
  std::cout << "Caught " << j << std::endl;
}
```

Figure 17-28. *A flow graph that throws an exception in one of its nodes*

If we re-run our original example from Figure 17-26 but use a try-catch block around the call to `wait_for_all`, we will see only one "Catch" message (either for 1 or 2):

```
Caught 2
```

The exception thrown in node `node2` is not caught in the node's body, so it will propagate to the thread that waits at the call to `wait_for_all`. If a node's body throws an exception, the graph it belongs to is canceled. In this case, we see that there is no second "Caught" message, since `node2` will only execute once.

And of course, if we want to re-execute the graph after we deal with the exception that we catch at the `wait_for_all`, we need to call `g.reset()` since the graph has been canceled.

Do: Set a Priority for a Graph Using `task_group_context`

We can set priorities for all of the tasks spawned by a graph by using the graph's `task_group_context`, for example:

```
if (auto t = g.root_task()) {
  t->group()->set_priority(tbb::priority_high);
}
```

Or we can pass in a `task_group_context` object with a preset priority to the graph's constructor. In either case though, this sets the priorities for all of the tasks related to the graph. We can create one graph with a high priority and another graph with a low priority.

Shortly before the publication of this book, support for relative priorities for functional nodes was added to TBB as a preview feature. Using this feature, we can pass a parameter to a node's constructor to give it a priority relative to other functional nodes. This interface was first provided in TBB 2019 Update 3. Interested readers can learn more details about this new functionality in the online TBB release notes and documentation.

Don't: Make an Edge Between Nodes in Different Graphs

All graph nodes require a reference to a graph object as one of the arguments to their constructor. In general, it is only safe to construct edges between nodes that are part of the same graph. Connecting two nodes in different graphs can make it difficult to reason about graph behaviors, such as what task arenas will be used, if our calls to `wait_for_all` will properly detect graph termination, and so on. To optimize performance, the TBB library takes advantage of its knowledge about edges. If we connect two graphs by an edge, the TBB library will freely reach across this edge for optimization purposes.

We may believe that we have created two distinct graphs, but if there are shared edges, TBB can start mixing their executions together in unexpected ways.

To demonstrate how we can get unexpected behavior, we implemented the class `WhereAmIRunningBody` shown in Figure 17-29. It prints `max_concurrency` and priority settings, which we will use to infer what task arena and `task_group_context` this body's task is using when it executes.

```
struct WhereAmIRunningBody {
  std::string node_name;
  WhereAmIRunningBody(const char *name) : node_name(name) {}

  int operator()(int i) {
    int P = tbb::this_task_arena::max_concurrency();
    std::string priority = "normal";

    if (tbb::task::self().group()->priority() == tbb::priority_high)
      priority = "high";

    std::cout << i << ":" << node_name
              << " executing in arena " << P
              << " with priority " << priority << std::endl;
    spinWaitForAtLeast(0.1);
    return i;
  }
};
```

Figure 17-29. *A body class that lets us infer what task arena and `task_group_context` are used by a node execution*

Figure 17-30 provides an example that uses the `WhereAmIRunningBody` to demonstrate an unexpected behavior. In this example, we create two nodes: `g2_node` and `g4_node`. The node `g2_node` is constructed with a reference to `g2`. The graph `g2` is passed a reference to a `task_group_context` that has `priority_normal` and `g2` is `reset()` in a `task_arena` with a concurrency of 2. We should therefore expect `g2_node` to execute with normal priority in an arena with 2 threads, right? The node `g4_node` is constructed such that we should expect it to execute with high priority in an arena with four threads.

The first group of calls that include `g2_node.try_put(0)` and `g4_node.try_put(1)` match these expectations:

```
0:g2_node executing in arena 2 with priority normal
1:g4_node executing in arena 4 with priority high

void fig_17_30() {
  tbb::task_arena a2{2};
  tbb::task_group_context tcg2;
  tcg2.set_priority(tbb::priority_normal);
  tbb::flow::graph g2{tcg2};
  a2.execute([&]() {
    g2.reset();
  });

  tbb::task_arena a4{4};
  tbb::task_group_context tcg4;
  tcg4.set_priority(tbb::priority_high);
  tbb::flow::graph g4{tcg4};
  a4.execute([&]() {
    g4.reset();
  });

  tbb::flow::function_node<int, int>
  g2_node{g2, tbb::flow::serial, WhereAmIRunningBody("g2_node")};

  tbb::flow::function_node<int, int>
  g4_node{g4, tbb::flow::serial, WhereAmIRunningBody("g4_node")};

  g2_node.try_put(0);
  g2.wait_for_all();

  g4_node.try_put(1);
  g4.wait_for_all();

  tbb::flow::make_edge(g2_node,g4_node);
  g2_node.try_put(2);
  g2.wait_for_all();
  g4.wait_for_all();
}
```

Figure 17-30. *An example that has unexpected behavior because of cross-graph communication*

But, when we make an edge from `g2_node` to `g4_node`, we make a connection between nodes that exist in two different graphs. Our second set of calls that include `g2_node.try_put(2)` again cause the body of `g2_node` to execute with normal priority in `arena a2`. But TBB, trying to reduce scheduling overheads, uses scheduler bypass (see Scheduler Bypass in Chapter 10) when it invokes `g4_node` due to the edge from `g2_node` to `g4_node`. The result is that `g4_node` executes in the same thread as `g2_node`, but this

thread belongs to `arena a2` not `a4`. It still uses the correct `task_group_context` when the task is constructed, but it winds up being scheduled in an unexpected arena.

```
2:g2_node executing in arena 2 with priority normal
2:g4_node executing in arena 2 with priority high
```

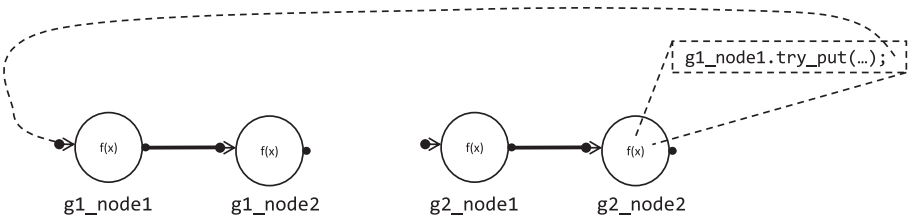
From this simple example, we can see that this edge breaks the separation between the graphs. If we were using arenas `a2` and `a4` to control the number of threads, for work isolation or for thread affinity purposes, this edge will undo our efforts. We ***should not*** make edges between graphs.

Do: Use `try_put` to Communicate Across Graphs

In the previous "Don't," we decided that we should not make edges between graphs. But what if we really need to communicate across graphs? The least dangerous option is to explicitly call `try_put` to send a message from a node in one graph to a node in another graph. We don't introduce an edge, so the TBB library won't do anything sneaky to optimize the communication between the two nodes. Even in this case though, we still need to be careful as our example in Figure 17-31 demonstrates.

Here, we create a graph `g2` that sends a message to graph `g1` and then waits for both graph `g1` and `g2`. But, the waiting is done in the wrong order!

Since node `g2_node2` sends a message to `g1_node1`, the call to `g1.wait_for_all()` will likely return immediately since nothing is going on in `g1` at the time of the call. We then call `g2.wait_for_all()`, which returns after `g2_node2` is done. After this call returns, `g2` is finished but `g1` has just received a message from `g2_node2` and its node `g1_node1` has just started to execute!



(a) diagram of the graph

```
void fig_17_31() {
  tbb::flow::graph g1;
  tbb::flow::function_node<int, int> g1_node1{g1, tbb::flow::serial,
    [](int i) {
      std::cout << "g1_node1\n";
      spinWaitForAtLeast(i*0.1);
      return i;}};
  tbb::flow::function_node<int, int> g1_node2{g1, tbb::flow::serial,
    [](int i) {
      std::cout << "g1_node2\n";
      spinWaitForAtLeast(i*0.1);
      return i;}};
  tbb::flow::make_edge(g1_node1,g1_node2);

  tbb::flow::graph g2;
  tbb::flow::function_node<int, int> g2_node1{g2, tbb::flow::serial,
    [](int i) {
      std::cout << "g2_node1\n";
      spinWaitForAtLeast(i*0.1);
      return i;}};
  tbb::flow::function_node<int, int> g2_node2{g2, tbb::flow::serial,
    [&g1_node1](int i) {
      std::cout << "g2_node2\n";
      spinWaitForAtLeast(i*0.1);
      g1_node1.try_put(i);
      return i;}};
  tbb::flow::make_edge(g2_node1,g2_node2);

  g2_node1.try_put(1);
  g1.wait_for_all(); // returns immediately
  g2.wait_for_all(); // returns after g2_node1 and g2_node2
  std::cout << "At the end of the function\n";
  // we reach here before g1 (started by g2) is done
}
```

(b) implementation of the graph

Figure 17-31. *A flow graph that sends a message to another flow graph*

Luckily, if we call the waits in the reverse order, things will work as expected:

```
g2.wait_for_all();
g1.wait_for_all();
```

But still, we can see that using explicit `try_puts` is not without dangers. We need to be very careful when graphs communicate with each other!

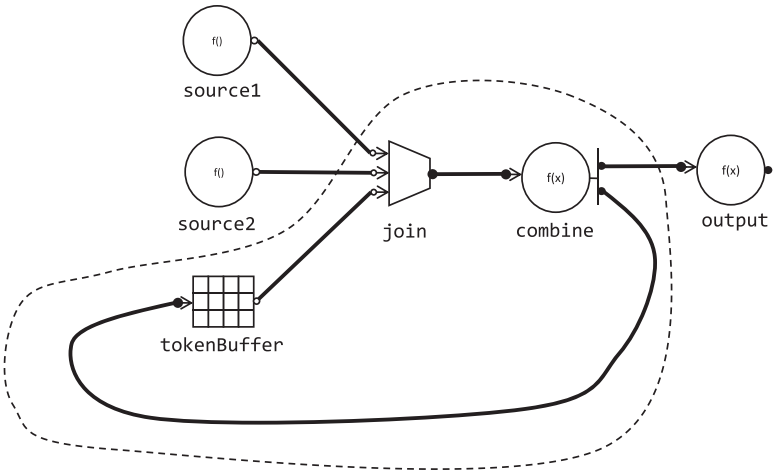
Do: Use `composite_node` to Encapsulate Groups of Nodes

In the previous two sections, we warned that communication between graphs can lead to errors. Often developers use more than one graph because they want to logically separate some nodes from others. Encapsulating a group of nodes is convenient if there is a common pattern that needs to be created many times or if there is too much detail in one large flat graph.

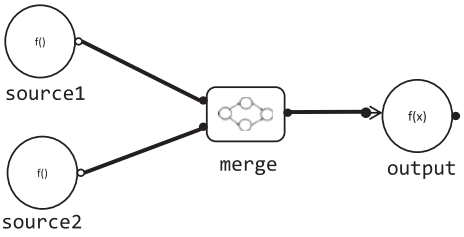
In both of these cases, we can use a `tbb::flow::composite_node`. A `composite_node` is used to encapsulate a collection of other nodes so they can be used like a first-class graph node. Its interface follows:

```
template< typename... InputTypes, typename... OutputTypes>
class composite_node <tbb::flow::tuple<InputTypes...>,
                      tbb::flow::tuple<OutputTypes...> > {
public:
  /* implementation defined */ input_ports_type;
  /* implementation defined */ output_ports_type;

  composite_node( graph &g );
  virtual ~composite_node();

  void set_external_ports(input_ports_type&& input_ports_tuple,
                          output_ports_type&& output_ports_tuple);
  input_ports_type& input_ports();
  output_ports_type& output_ports();
};
```

Unlike the other node types that we have discussed in this chapter and in Chapter 3, we need to create a new class that inherits from `tbb::flow::composite_node` to make use of its functionality. For example, let's consider the flow graph in Figure 17-32(a). This graph combines two inputs from `source1` and `source2`, and uses a token passing scheme to limit memory consumption.



(a) a graph with unneeded detail at the top level



(b) a simplified graph using a `composite_node`

Figure 17-32. *An example that benefits from a* `composite_node`

If this token passing pattern is commonly used in our application, or by members of our development team, it might make sense to encapsulate it into its own node type, as shown in Figure 17-32(b). It also cleans up the high-level view of our application by hiding the details.

Figure 17-33 shows what a flow graph implementation looks like if we have a node that implements the dotted parts of Figure 17-32(a), replacing it with a single `merge` node. In Figure 17-33, we use the `merge` node object like any other flow graph node, making edges to its input and output ports. Figure 17-34 shows how we use `tbb::flow::composite_node` to implement our `MergeNode` class.

```
void fig_17_33() {
  tbb::flow::graph g;

  int src1_count = 0;
  tbb::flow::source_node<BigObjectPtr> source1{g,
  [&] (BigObjectPtr &m) -> bool {
    if (src1_count < A_LARGE_NUMBER) {
      m = std::make_shared<BigObject>(src1_count++);
      return true;
    }
    return false;
  }, false};

  int src2_count = 0;
  tbb::flow::source_node<BigObjectPtr> source2{g,
  [&] (BigObjectPtr &m) -> bool {
    if (src2_count < A_LARGE_NUMBER) {
      m = std::make_shared<BigObject>(src2_count++);
      return true;
    }
    return false;
  }, false};

  MergeNode merge{g};

  tbb::flow::function_node<BigObjectPtr> output{g,
    tbb::flow::serial,
    [] (BigObjectPtr b) {
      std::cout << "Received id == " << b->getId()
                << " in final node" << std::endl;
    }
  };

  tbb::flow::make_edge(source1, tbb::flow::input_port<0>(merge));
  tbb::flow::make_edge(source2, tbb::flow::input_port<1>(merge));
  tbb::flow::make_edge(merge, output);

  bigObjectCount = 0;
  maxCount = 0;
  source1.activate();
  source2.activate();
  g.wait_for_all();
  std::cout << "maxCount == " << maxCount << std::endl;
}
```

Figure 17-33. *Creating a flow graph that uses a class* `MergeNode` *that inherits from* `tbb::flow::composite_node`

```
using BigObjectPtr = std::shared_ptr<BigObject>;
using CompositeType =
  tbb::flow::composite_node<std::tuple<BigObjectPtr, BigObjectPtr>,
                            std::tuple<BigObjectPtr>>;
using MFNode =
  tbb::flow::multifunction_node<std::tuple<BigObjectPtr,
                                           BigObjectPtr, token_t>,
                                std::tuple<BigObjectPtr, token_t>>;


class MergeNode : public CompositeType {
  tbb::flow::buffer_node<token_t> tokenBuffer;
  tbb::flow::join_node<std::tuple<BigObjectPtr, BigObjectPtr, token_t>,
                       tbb::flow::reserving> join;
  MFNode combine;

public:
  MergeNode(tbb::flow::graph &g) :
    CompositeType{g},
    tokenBuffer{g},
    join{g},
    combine{g, tbb::flow::unlimited,
      [] (const MFNode::input_type &in, MFNode::output_ports_type &p) {
        BigObjectPtr b0 = std::get<0>(in);
        BigObjectPtr b1 = std::get<1>(in);
        token_t t = std::get<2>(in);
        spinWaitForAtLeast(0.0001);
        b0->mergeIds(b0->getId(), b1->getId());
        std::get<0>(p).try_put(b0);
        std::get<1>(p).try_put(t);
      }}
  {
    tbb::flow::make_edge(tokenBuffer, tbb::flow::input_port<2>(join));
    tbb::flow::make_edge(join, combine);
    tbb::flow::make_edge(tbb::flow::output_port<1>(combine),
                         tokenBuffer);

    CompositeType::set_external_ports(
      CompositeType::input_ports_type(
        tbb::flow::input_port<0>(join),
        tbb::flow::input_port<1>(join)),
      CompositeType::output_ports_type(
        tbb::flow::output_port<0>(combine))
    );

    for (token_t i = 0; i < 3; ++i) tokenBuffer.try_put(i);
  }
};
```

Figure 17-34. *The implementation of* `MergeNode`

In Figure 17-34, MergeNode inherits from CompositeType, which is an alias for

```
tbb::flow::composite_node<std::tuple<BigObjectPtr, BigObjectPtr>,
                          std::tuple<BigObjectPtr>>;
```

The two template arguments indicate that a MergeNode will have two input ports, both that receive BigObjectPtr messages, and a single output port that sends BigObjectPtr messages. The class MergeNode has a member variable for each node it encapsulates: a tokenBuffer, a join, and a combine node. And these member variables are initialized in the member initializer list of the MergeNode constructor. In the constructor body, calls to tbb::flow::make_edge set up all of the internal edges. A call to set_external_ports is used to assign the ports from the member nodes to the external ports of the MergeNode. In this case, the first two input ports of join become the inputs of the MergeNode and the output of combine becomes the output the MergeNode. Finally, because the node is implementing a token passing scheme, the tokenBuffer is filled with tokens.

While creating a new type that inherits from tbb::flow::composite_node may appear daunting at first, using this interface can lead to more readable and reusable code, especially as your flow graphs become larger and more complicated.

Introducing Intel Advisor: Flow Graph Analyzer

The Flow Graph Analyzer (FGA) tool is available in Intel Parallel Studio XE 2019 and later. It is provided as a feature of the Intel Advisor tool. Instructions for getting the tool can be found at https://software.intel.com/en-us/articles/intel-advisor-xe-release-notes.

FGA was developed to support the design, debugging, visualization, and analysis of graphs built using the TBB flow graph API. That said, many of the capabilities of FGA are generically useful for analyzing computational graphs, regardless of their origin. Currently, the tool has limited support for other parallel programming models including the OpenMP API.

For our purposes in this book, we will focus only on how the design and analysis workflows in the tool apply to TBB. We also use FGA to analyze some of the samples in this chapter. However, all of the optimizations presented in this chapter can be done with or without FGA. So, if you have no interest in using FGA, you can skip this section. But again, we believe there is significant value in this tool, so skipping it would be a mistake.

The FGA Design Workflow

The design workflow in FGA lets us graphically design TBB flow graphs, validate that they are correct, estimate their scalability, and, after we are satisfied with our design, generate a C++ implementation that uses the TBB flow graph classes and functions. FGA is not a full Integrated Development Environment (IDE) like Microsoft Visual Studio, Eclipse or Xcode. Instead, it gets us started with our flow graph design, but then we need to step outside of the tool to complete the development. However, if we use the design workflow in a constrained way, as we will describe later, iterative development in the designer is possible.

Figure 17-35 shows the FGA GUI used during the design workflow. We will only briefly describe the components of the tool here as we describe the typical workflow; the Flow Graph Analyzer documentation provides a more complete description.

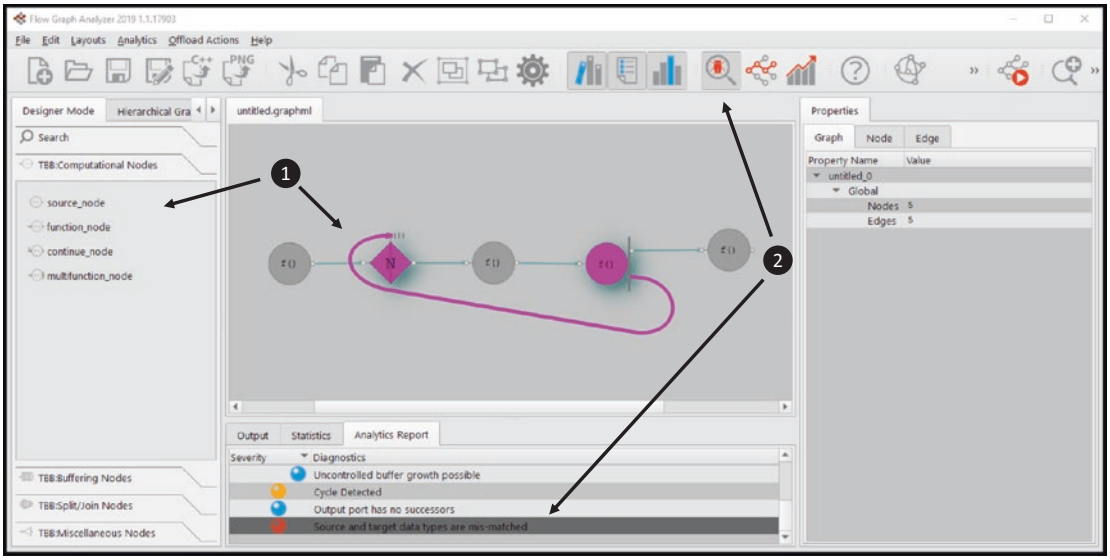


Figure 17-35. *Using the FGA design workflow*

The typical design workflow starts with a blank canvas and project. As highlighted by the black circle numbered 1 in Figure 17-35, we select nodes in the node palette and place them on the canvas, connecting them together by drawing edges between their ports. The node palette contains all of the node types available in the TBB flow graph interface and provides tooltips that remind us about the functionality of each type. For each node on the canvas, we can modify its type-specific properties; for a `function_node`

for example, we can provide the C++ code for the body, set a concurrency limit, and so on. We can also provide an estimated "weight" that represents the computational complexity of the node so that later we can run a Scalability Analysis to see if our graph will perform well.

Once we have drawn our graph on the canvas, we run a Rule Check that analyzes the graph looking for common mistakes and anti-patterns. The Rule Check results, highlighted by the black circle numbered 2 in Figure 17-35, show issues such as unnecessary buffering, type mismatches, suspicious cycles in the graph, and so on. In Figure 17-35, the Rule Check has discovered that there is a type mismatch between the input of our `limiter_node` and the output of our `multifunction_node`. In response, we can then, for example, modify the port output type of our `multifunction_node` to fix this issue.

When we have fixed all correctness issues uncovered by the Rule Check, we can then run a Scalability Analysis. The Scalability Analysis constructs a TBB flow graph in memory, replacing the computational node bodies with dummy bodies that actively spin for a time proportional to their "weight" property. FGA runs this model of our graph on various numbers of threads and provides a table of the speedups, for example:

Graph Name	Graph	Threads	Time(s)	Speedup
simple_g0	Results(4)			
	Scalability projection	1	4.00006	1
	Scalability projection	2	2.00003	2
	Scalability projection	3	2.00004	1.99999
	Scalability projection	4	1.01446	3.94305

Using these features, we can iteratively refine our graph design. Along the way, we can save our graph design in GraphML format (a common standard for representing graphs). When we are satisfied with our design we can generate C++ code that uses the TBB flow graph interface to express our design. This code generator is more accurately viewed as a code wizard than an IDE since it does not directly support an iterative code development model. If we change the generated code, there is no way to reimport our changes into the tool.

Tips for Iterative Development with FGA

If we want to create a design that we can continue to tune from within FGA, we can use a constrained approach, where we specify node bodies that redirect to implementations that are maintained outside of FGA. This is necessary because there is no way to reimport modified C++ code back into FGA.

For example, if we want to make iterative development easier, we should not specify a `function_node` that exposes its implementation directly in the body code:

```
tbb::flow::function_node<int, std::string>
comp2str(g, tbb::flow::unlimited,
  [](int i) -> std::string {
    // we specified a complete implementation
    // using the node's body property in FGA
    int output_value = i*i;
    return std::to_string(output_value);
  }
);
```

Instead, we should specify only the interface and redirect to an implementation that can be maintained separately:

```
tbb::flow::function_node<int, std::string>
comp2str(g, tbb::flow::unlimited,
  [](int i) -> std::string {
    // we specified a redirected implementation
    // using the node's body property in FGA
    return comp2str_impl(i);
  }
);
```

If we take this constrained approach, we can often maintain the graph design in FGA and its `GraphML` representation, iteratively tuning the topology and node properties without losing any node body implementation changes we make outside of the tool. Whenever we generate new C++ code from FGA, we simply include the most up-to-date implementation header and the node bodies use these implementations that are maintained outside of the tool.

Flow Graph Analyzer does not require us to use this approach of course, but it is good practice if we want to use the code generation features of FGA as more than a simple code wizard.

The FGA Analysis Workflow

The analysis workflow in FGA is independent of the design workflow. While we can surely analyze a flow graph that was designed in FGA, we can just as easily analyze a TBB flow graph that is designed and implemented outside of the tool. This is possible because the TBB library is instrumented to provide runtime events to the FGA trace collector. A trace collected from a TBB application lets FGA reconstruct the graph structure and the timeline of the node body executions – it does ***not*** depend on the GraphML files developed during the design workflow.

If we want to use FGA to analyze a TBB application that uses a flow graph, the first step is to collect an FGA trace. By default, TBB does not generate traces, so we need to activate trace collection. The FGA instrumentation in TBB was a preview feature prior to TBB 2019. We need to take extra steps if we are using an older version of TBB. We refer readers to the FGA documentation for instructions on how to collect traces for the version of TBB and FGA that they are using.

Once we have a trace of our application, the analysis workflow in FGA uses the activities highlighted by the numbered circles in Figure 17-36: (1) inspect the tree-map view for an overview of the graph performance and use this as an index into the graph topology display, (2) run the critical path algorithm to determine the critical paths through the computation, and (3) examine the timeline and concurrency data for insight into performance over time. Analysis is most commonly an interactive process that moves between these different activities as the performance of the application is explored.

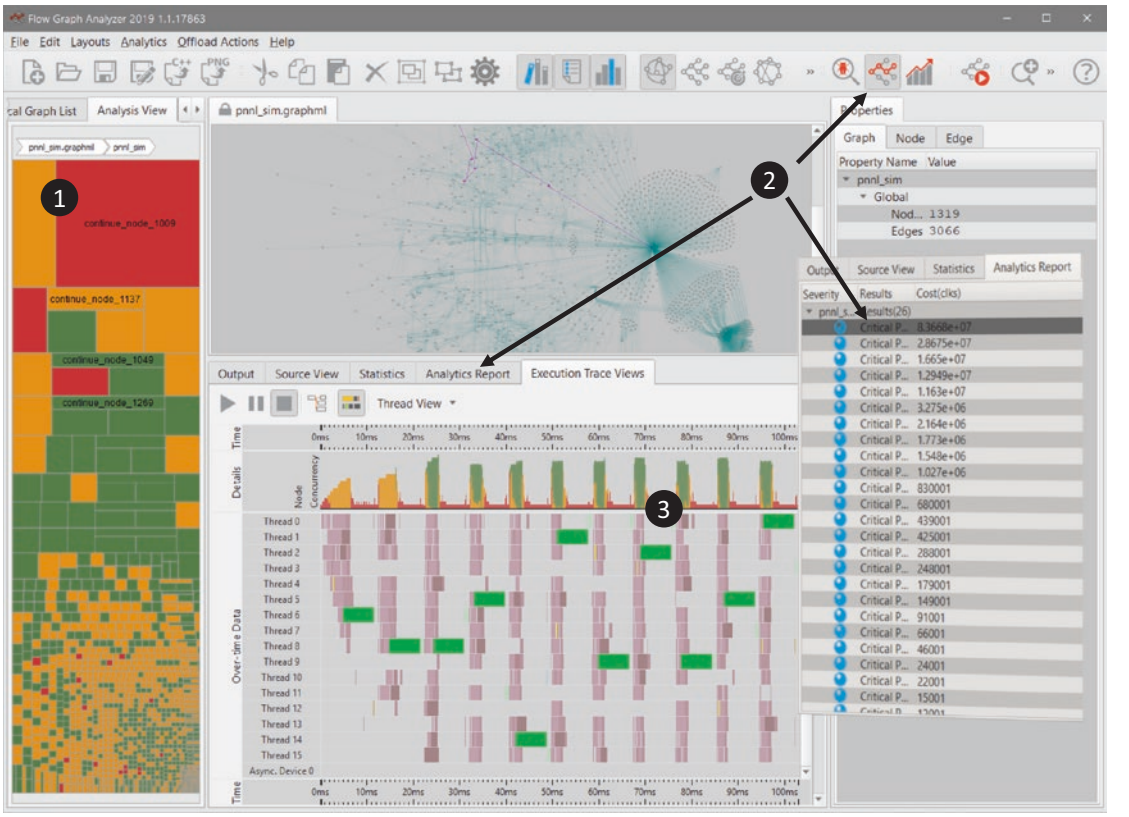


Figure 17-36. *Using the FGA analysis workflow. These results were collected on a system with 16 cores.*

The tree-map view labeled as (1) in Figure 17-36 provides an overview of the overall health of a graph. In the tree map, the area of each rectangle represents the total aggregated CPU time of the node and the color of each square indicates the concurrency observed during the execution of the node. The concurrency information is categorized as poor (red), ok (orange), good (green), and oversubscribed (blue).

Nodes with a large area that are marked as "poor" are hotspots and have an average concurrency between 0% and 25% of the hardware concurrency. These are therefore good candidates for optimization. The tree-map view also serves as an index into a large graph; clicking on a square will highlight the node in the graph and selecting this highlighted node will in turn mark tasks from all instances of this node in the timeline trace view.

The graph topology canvas is synchronized with other views in the tool. Selecting a node in the tree-map view, the timeline, or in a data analytics report will highlight the node in the canvas. This lets users quickly relate performance data to the graph structure.

One of the most important analytic reports provided by FGA is the list of critical paths in a graph. This feature is particularly useful when one has to analyze a large and complex graph. Computing the critical paths results in a list of nodes that form the critical paths as shown in the region labeled (2) in Figure 17-36. As we discussed in Chapter 3, an upper bound on speedup of dependency graphs can be quickly computed by dividing the aggregate total time spent by all nodes in a graph by the time spent on the longest critical path, T_1/T_∞. This upper bound can be used to set expectations on the potential speedup for an application expressed as a graph.

The timeline and concurrency view labeled as (3) in Figure 17-36 displays the raw traces in swim lanes mapped to software threads. Using this trace information, FGA computes additional derived data such as the average concurrency of each node and the concurrency histogram over time for the graph execution. Above the per-thread swim lanes, a histogram shows how many nodes are active at that point in time. This view lets users quickly identify time regions with low concurrency. Clicking on nodes in the timelines during these regions of low concurrency lets developers find the structures in their graph that lead to these bottlenecks.

Diagnosing Performance Issues with FGA

In this chapter, we discussed a number of potential performance issues that can arise when using a flow graph. In this section, we briefly discuss how FGA can be used to explore these issues in a TBB-based application.

Diagnosing Granularity Issues with FGA

Just like with our TBB generic loop algorithms, we need to be concerned about tasks that are too small to profit from parallelization. But we need to balance this concern with the need to create enough tasks to allow our workload to scale. In particular, as we discussed in Chapter 3, scalability can be limited by serial nodes if they become a bottleneck in the computation.

In an example timeline from FGA shown in Figure 17-37, we can see that there is a dark serial task, named `m`, which causes regions of low concurrency. The color indicates that this task is about 1 millisecond in length – this is above the threshold for efficient scheduling but, from the timeline, it appears to be a serializing bottleneck. If possible, we should break this task up into tasks that can be scheduled in parallel – either by breaking it into multiple independent nodes or through nested parallelism.

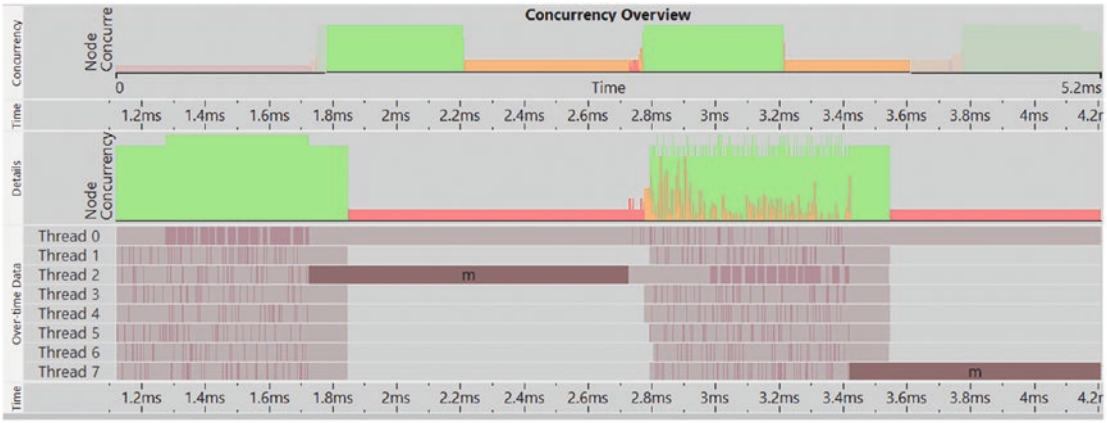


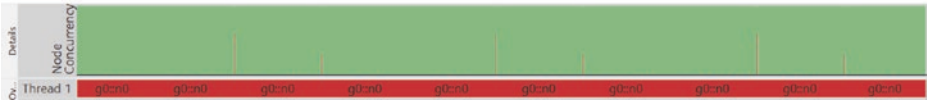
Figure 17-37. *The FGA timeline colors tasks according to their execution times. Lighter tasks are smaller.*

In contrast, there are regions in Figure 17-37 where smaller tasks, named `n`, are executed in parallel. By their coloring, it appears these are close to the 1 microsecond threshold, and consequently we can see gaps in the timelines during this region, indicating that there may be some non-negligible scheduling overheads involved. In this case, it may benefit us to merge nodes or to use a `lightweight` policy if possible to decrease overheads.

Recognizing Slow Copies in FGA

Figure 17-38 shows how we might recognize slow copies in FGA. In the figure, we see 100 millisecond segments from the timelines of runs of graphs similar to Figure 17-12, but that pass `BigObject` messages directly (Figure 17-38(a)) and `shared_ptr<BigObject>` messages (Figure 17-38(b)). To make the construction appear expensive, we inserted a spin-wait in the `BigObject` constructor so that it takes 10 milliseconds to construct each object – making the construction time of a `BigObject` and our `function_node` body's execution times equal. In Figure 17-38(a), we can see the time it takes to copy the message between nodes appears as gaps in the timeline. In Figure 17-38(b), where we pass by pointer, the message passing time is negligible, so no gaps are seen.

(a) using `BigObject` messages



(b) using `shared_ptr<BigObject>` messages

Figure 17-38. *In FGA, the long copies appear as gaps between the node body executions. Each timeline segment shown is approximately 100 milliseconds long.*

When using FGA to analyze our flow graph applications, gaps in the timeline indicate inefficiencies that need to be further investigated. In this section, they indicated costly copies between nodes and in the previous section they indicated that the overhead of scheduling was large compared to the task sizes. In both cases, these gaps should prompt us to look for ways to improve performance.

Diagnosing Moonlighting using FGA

Earlier in this chapter, we discussed the execution of the moonlighting graph in Figure 17-23 that generated the output in Figure 17-24. FGA provides a Stacked View in its execution timeline that lets us easily detect moonlighting as shown in Figure 17-39.

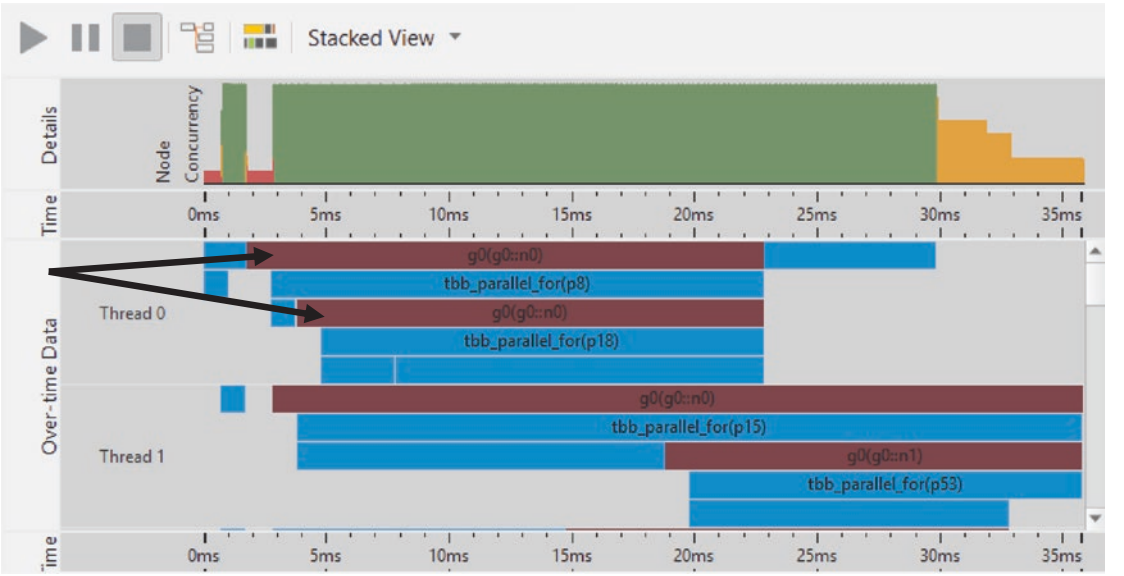


Figure 17-39. *FGA timelines grouped by Node/Region. We can see that thread 0 is moonlighting since it shown as concurrently executing more than one parallel region.*

In a Stacked View, we see all of the nested tasks that a thread is executing, including those that come from flow graph nodes and those that come from TBB Generic Parallel Algorithms. If we see that a thread executes two nodes concurrently, it is moonlighting. In Figure 17-39, for example, we see that Thread 0 starts executing another instance of node `n0` inside of an existing instance of `n0`. In our previous discussions about moonlighting, we know this can happen if a thread steals work while it is waiting for a nested parallel algorithm to complete. The Stacked View in Figure 17-39, lets us easily see that a nested `parallel_for`, labeled `p8`, is the culprit in this case.

Using the timeline views from FGA, we can identify when threads are moonlighting simply by noticing a thread's overlapped participation in more than one region or node. As developers, and possibly through other interactions with FGA, we then need to determine if the moonlighting is benign or needs to be addressed by TBB's isolation features.

Summary

The flow graph API is a flexible and powerful interface for creating dependency and data flow graphs. In this chapter, we discussed some of the more advanced considerations in using the TBB flow graph high-level execution interface. Because it is implemented on

top of TBB tasks, it shares the composability and optimization features supported by TBB tasks. We discussed how these can be used to optimize for granularity, effective cache, and memory use and create sufficient parallelism. We then listed some dos and don'ts that can be helpful when first exploring the flow graph interfaces. Finally, we provided a brief overview of the Flow Graph Analyzer (FGA), a tool available in Intel Parallel Studio XE that has support for the graphical design and analysis of TBB flow graphs.

For More Information

Michael Voss, "The Intel Threading Building Blocks Flow Graph," Dr. Dobb's, October 5, 2011. www.drdobbs.com/tools/the-intel-threading-building-blocks-flow/231900177.

Vasanth Tovinkere, Pablo Reble, Farshad Akhbari and Palanivel Guruvareddiar, "Driving Code Performance with Intel Advisor's Flow Graph Analyzer," Parallel Universe Magazine, https://software.seek.intel.com/driving-code-performance.

Richard Friedman, "Intel Advisor's TBB Flow Graph Analyzer: Making Complex Layers of Parallelism More Manageable," Inside HPC, December 14, 2017, https://insidehpc.com/2017/12/intel-flow-graph-analyzer/.