CHAPTER 8

Mapping Parallel Patterns to TBB

It has been said that history does not repeat, it rhymes.

It could be said that software rhymes as well. While we may not write the same code over and over, there are patterns that emerge in the problems we solve and the code we write. We can learn from similar solutions.

This chapter takes a look at patterns that have proven to be effective in solving problems in a scalable manner, and we connect them with how to implement them using TBB (Figure 8-1). In order to achieve scalable parallelization, we should focus on data parallelism; data parallelism is the best overall strategy for scalable parallelism. Our coding needs to encourage the subdivision of any task into multiple tasks, with the number of tasks able to grow with the overall problem size; an abundance of tasks enables better scaling. Assisted best by the patterns we promote in this chapter, coding to provide an abundance of tasks helps us achieve scalability in our algorithms.

We can learn to "Think Parallel" by seeing how others have done it effectively already. Of course, we can stand on the shoulders of giants and reach ever further.

This chapter is about learning from prior experiences of parallel programmers, and in the course of doing that, learning better how to use TBB. We talk in terms of patterns as inspiration and useful tools for "Thinking Parallel." We do not describe patterns to form a perfect taxonomy of programming.

Parallel Patterns vs. Parallel Algorithms

As we mentioned in Chapter 2, it has been suggested to us by reviewers of this book that "TBB parallel algorithms" should be referred to as *patterns* instead of algorithms. That may be true, but in order to align with the terminology that the TBB library has been using for many years, we refer to these features as generic parallel *algorithms* throughout this book and in the TBB documentation. The effect is the same – they offer the opportunity for us to benefit from the experience of those who have explored optimal solutions to these patterns before us – not only to *use* them, but to be encouraged to prefer using these particular patterns (algorithms) over other possible approaches because they tend work best (achieve better scaling).

Pattern Concept	Pattern Name	TBB Template
Powerful boost to all patterns: The ability to nest without restriction patterns within patterns.	**nesting**	*all of TBB*
Best pattern to use: Division of work into uniform independent tasks.	**map**	`parallel_for, parallel_invoke`
Generalization of map with incremental task addition.	**workpile** *(generalized map with incremental task addition)*	`parallel_do`
Common operation: Division of work into independent tasks to compute partials results that are reduced (combined) into a final result.	**reduce**	`parallel_reduce`
Specialized reduction capability to do a computation of a prefix computation (also known as a scan) in parallel: `y[i]=y[i-1] op x[i]`	**scan** *sometimes called "prefix"*	`parallel_scan`
Classic and powerful: Control flow splits into two flows (tasks) which eventually rejoin.	**fork-join**	`parallel_invoke, task_group, flow_graph`
Specialization of fork-join: divide work into subtasks recursively.	**divide-and-conquer**	`parallel_for, parallel_reduce, parallel_invoke, task_group, flow_graph, parallel_sort`
Specialization of fork-join: divide work into subtask recursively with pruning to reduce the need for exhaustive search. Cancellation support (Chapter 15) can very valuable for implementing.	**branch-and-bound**	`parallel_for, parallel_reduce, parallel_invoke, task_group, flow_graph` *plus controls for cancellation (Chapter 15)*
Deceivingly powerful: Tasks connected in a producer-consumer relationship in a regular, nonchanging data flow.	**pipeline**	`parallel_pipeline, flow_graph`
Handles real world messy flows well: Tasks connected in a producer-consumer relationship with an irregular, and possibly changing, interaction between tasks.	**Event Based Coordination**	`flow_graph`

Figure 8-1. *TBB templates that express important "Patterns that work"*

Patterns Categorize Algorithms, Designs, etc.

The value of object-oriented programming was described by the Gang of Four (Gamma, Helm, Johnson, and Vlissides) and their landmark work *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley). Many credit that book with bringing more order to the world of object-oriented programming. Their book gathered the collective wisdom of the community and boiled it down into simple "patterns" with names, so people could talk about them.

Patterns for Parallel Programming by Mattson, Sanders, and Massingill (Addison-Wesley) has similarly collected wisdom from the parallel programming community. Experts use common tricks and have their own language to talk about techniques. With parallel patterns in mind, programmers can quickly come up to speed in parallel programming just as object-oriented programmers have done with the famous Gang-of-Four book.

Patterns for Parallel Programming is longer than this book, and very dense reading, but with some help from author Tim Mattson, we can summarize how the patterns relate to TBB.

Tim et al. propose that programmers need to work through four design spaces to develop a parallel program:

1. Finding concurrency.

   For this design space, we work within our problem domain to identify available concurrency and expose it for use in the algorithm design. TBB simplifies this effort by encouraging us to find as many tasks as we can without having to worry about how to map them to hardware threads. We also provide information on how to best make the tasks split in half when the task is considered large enough. Using this information, TBB then automatically divides large tasks repeatedly to help spread work evenly among processor cores. An abundance of tasks leads to scalability for our algorithms.

2. Algorithm structures.

   This design space embodies our high-level strategy for organizing a parallel algorithm. We need to figure out how we want to organize our workflow. Figure 8-1 lists important patterns that we can consult to guide our selection toward a pattern that best suits our needs. These "patterns that work" are the focus of *Structured Parallel Programming* by McCool, Robison, and Reinders (Elsevier).

3. Supporting structures.

   This step involves the details for turning algorithm strategy into actual code. We consider how the parallel program will be organized and the techniques used to manage shared (especially mutable) data. These considerations are critical and have an impact that reaches across the entire parallel programming process. TBB is well designed to encourage the right level of abstraction, so this design space is satisfied by using TBB well (something we hope we teach in this book).

4. Implementation mechanisms.

   This design space includes thread management and synchronization. Threading Building Blocks handles all the thread management, leaving us free to worry only about tasks at a higher level of design. When using TBB, most programmers code to avoid explicit synchronization coding and debugging. TBB algorithms (Chapter 2) and flow graph (Chapter 3) aim to minimize explicit synchronization. Chapter 5 discusses synchronization mechanisms for when we do need them, and Chapter 6 offers containers and thread local storage to help limit the need for explicit synchronization.

   Using a pattern language can guide the creation of better parallel programming environments and help us make the best use of TBB to write parallel software.

Patterns That Work

Armed with the language of patterns, we should regard them as tools. We emphasize patterns that have proven useful for developing the most scalable algorithms. We know that two prerequisites for achieving parallel scalability are good data locality and avoidance of overhead. Fortunately, many good strategies have been developed for achieving these objectives and are accessible using TBB (see table in Figure 8-1). Consideration of the need to be well tuned, to real machines, details are already provided for within TBB including issues related to the implementation of patterns such as granularity control and good use of cache.

In these terms, TBB handles the details of implementation, so that we can program at a higher level. This is what lets code written using TBB be portable, leaving machine-specific tuning inside of TBB. TBB in turn, by virtue of algorithms such as task-stealing, helps minimize the tuning needed to port TBB. The abstraction of the algorithm strategy into semantics and implementation has proven to work extremely well in practice. The separation makes it possible to reason about the high-level algorithm design and the low-level (and often machine-specific) details separately.

Patterns provide a common vocabulary for discussing approaches to problem solving and allow reuse of best practices. Patterns transcend languages, programming models, and even computer architectures, and we can use patterns whether or not the programming system we are using explicitly supports a given pattern with a specific feature. Fortunately, TBB was designed to emphasize proven patterns that lead to well-structured, maintainable, and efficient programs. Many of these patterns are in fact also deterministic (or can be run in a deterministic mode – see Chapter 16), which means they give the same result every time they are executed. Determinism is a useful property since it leads to programs that are easier to understand, debug, test, and maintain.

Data Parallelism Wins

The best overall strategy for scalable parallelism is data parallelism. Definitions of data parallelism vary. We take a wide view and define data parallelism as any kind of parallelism that grows as the data set grows or, more generally, as the problem size grows. Typically, the data is split into chunks and each chunk processed with a separate task. Sometimes, the splitting is flat; other times, it is recursive. What matters is that bigger data sets generate more tasks.

Whether similar or different operations are applied to the chunks is irrelevant to our definition. In general, data parallelism can be applied whether a problem is regular or irregular. Because data parallelism is the best strategy for scalable parallelism, hardware support for data parallelism is commonly found in all types of hardware – CPUs, GPUs, ASIC designs, and FPGA designs. Chapter 4 discussed support for SIMD precisely to connect with such hardware support.

The opposite of data parallelism is functional decomposition (also called task parallelism), an approach that runs different program functions in parallel. At best, functional decomposition improves performance by a constant factor. For example, if a program has functions `f`, `g`, `and h`, running them in parallel at best triples performance, and in practice less. Sometimes functional decomposition can deliver an additional bit of parallelism required to meet a performance target, but it should not be our primary strategy, because it does not scale.

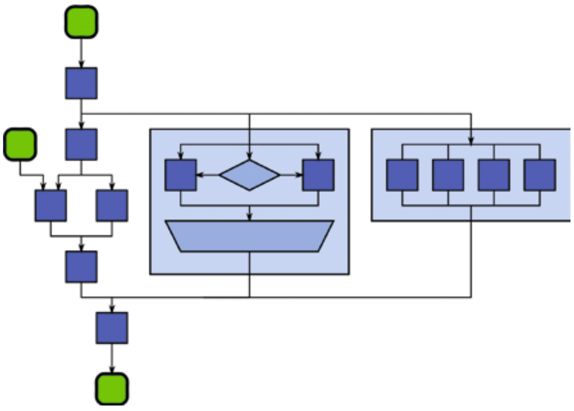

Figure 8-2. *Nesting pattern: a compositional pattern that allows other patterns to be composed in a hierarchy. Nesting is that any task block in a pattern can be replaced with a pattern with the same input and output configuration and dependencies.*

Nesting Pattern

Nesting (Figure 8-2) may seem obvious and normal, but in the parallel programming world it is not. TBB makes life simple – nesting just works, without severe oversubscription issues that other models such as OpenMP can have.

Two implications to emphasize what we get because of nesting support:

- We do not need to know if we are in a "parallel region" or a "serial region" when choosing if we should invoke a TBB template. Since using TBB just creates tasks, we do not have to worry about oversubscription of threads.

- We do not need to worry about calling a library, which was written with TBB, and controlling if it might use parallelism.

Nesting can be thought of as a meta-pattern because it means that patterns can be hierarchically composed. This is important for modular programming. Nesting is extensively used in serial programming for composability and information hiding but can be a challenge to carry over into parallel programming. The key to implementing nested parallelism is to specify optional, not mandatory, parallelism. This is one area that TBB excels compared to other models.

The importance of nesting was well understood when TBB was introduced in 2006, and it has always been well supported in all of TBB. In contrast, the OpenMP API was introduced in 1997 when we did not adequately foresee the critical importance of the nesting pattern for future machines. As a result, the nesting pattern is not supported throughout OpenMP. This can make OpenMP much more difficult to use for anything outside the world of applications which focus almost all work inside computationally intensive loop nests. These are the application types that dominated our thinking when creating OpenMP and its predecessors in the 1980s and 1990s. The nesting pattern, with modularity and composability, was key in our thinking when TBB was created (we credit the Cilk research work at MIT for the pioneering work that influenced our thinking heavily – see Appendix A for many more comments on influences, including Cilk).

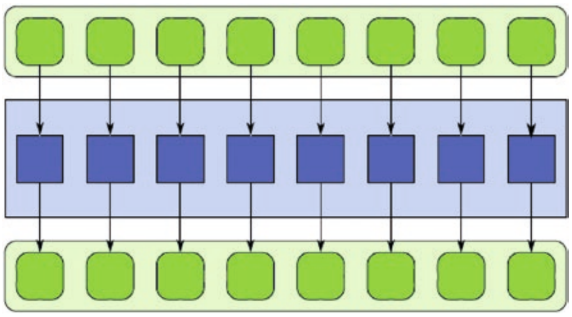

Figure 8-3. *Map pattern: a function is applied to all elements of a collection, usually producing a new collection with the same shape as the input.*

Map Pattern

The **map** pattern (Figure 8-3) is the most optimal pattern for parallel programming possible: dividing work into uniform independent parts that run in parallel with no dependencies. This represents a regular parallelization that is referred to as *embarrassing* parallelism. That is to say, the parallelism seems most obvious in cases where there is independent parallel work to be done. There is *nothing* embarrassing about getting great performance when an algorithm scales well! This quality makes the map pattern worth using whenever possible since it allows for both efficient parallelization and efficient vectorization.

A map pattern involves *no* shared mutable state between the parts; a map function (the independent work parts) must be "pure" in the sense that it must not modify shared state. Modifying shared (mutable) state would break perfect independence. This can result in nondeterminism from data races and result in undefined behavior including possible application failure. Hidden shared data can be present when using complex data structures, for instance `std::share_ptr`, which may have sharing implications.

Usages for map patterns include gamma correction and thresholding in images, color space conversions, Monte Carlo sampling, and ray tracing. Use `parallel_for` to implement map efficiently with TBB (example in Figure 8-4). Additionally, `parallel_invoke` can be used for a small amount of map type parallelism, but the limited amount will not provide much scalability unless parallelism also exists at other levels (e.g., inside the invoked functions).

```
tbb::parallel_for( size_t(0), n, [&](size_t i) {
  Foo(i);
} );
```

Figure 8-4. *Map pattern realized in parallel with `parallel_for`*

Workpile Pattern

The **workpile** pattern is a generalized map pattern where each instance (map function) can generate more instances. In other words, work can be added to the "pile" of things to do. This can be used, for example, in the recursive search of a tree, where we might want to generate instances to process each of the children of each node of the tree. Unlike the case with the map pattern, with the workpile pattern, the total number of instances of the map function is not known in advance nor is the structure of the work regular. This makes the workpile pattern harder to vectorize (Chapter 4) than the map pattern. Use `parallel_do` (Chapter 2) to implement workpile efficiently with TBB.

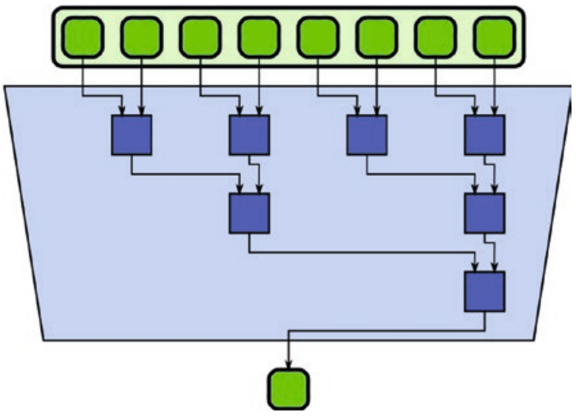

Figure 8-5. *Reduction pattern: subtasks produce subresults that are combined to form a final single answer.*

Reduction Patterns (Reduce and Scan)

The **reduce** pattern (Figure 8-5) can be thought of as a map operation where each subtask produces a subresult that we need to combine to form a final single answer. A reduce pattern combines the multiple subresults using an associative "combiner function." Because of the associativity of the combiner function, different orderings of the combining are possible run-to-run which is both a curse and a blessing. The blessing is that an implementation is free to maximize performance by combining in any order that is most efficient. The curse is that this offers a nondeterminism in the output if there are variations run-to-run in results due to rounding or saturation. Combining to find the maximum number or to find the boolean AND of all subresults does not suffer from these issues. However, a global addition using floating-point numbers will be nondeterministic due to rounding variations.

TBB offers both nondeterministic (highest performance) and deterministic (typically only a slight performance penalty) for reduction operations. The term deterministic refers only to the deterministic order of reduction run-to-run. If the combining function is deterministic, such as boolean AND, then the nondeterministic order of `parallel_reduce` will yield a deterministic result.

Typical combiner functions include addition, multiplication, maximum, minimum, and boolean operations AND, OR, and XOR. We can use `parallel_reduce` (Chapter 2) to implement nondeterministic reduction. We can use `parallel_deterministic_reduce` (Chapter 16) to implement deterministic reduction. Both allow us the ability to define our own combiner functions.

The **scan** pattern (Figure 8-6) computes a prefix computation (also known as a scan) in parallel (`y[i]=y[i-1] op x[i]`). As with other reductions, this can be done in parallel if `op` is associative. This can be useful in scenarios that appear to have inherently serial dependencies. Many people are surprised that there is a scalable way to do this at all. A sample of what serial code may look like is shown in Figure 8-7. A parallel version requires more operations than a serial version, but it offers scaling. TBB `parallel_scan` (Chapter 2) is used to implement scan operations.

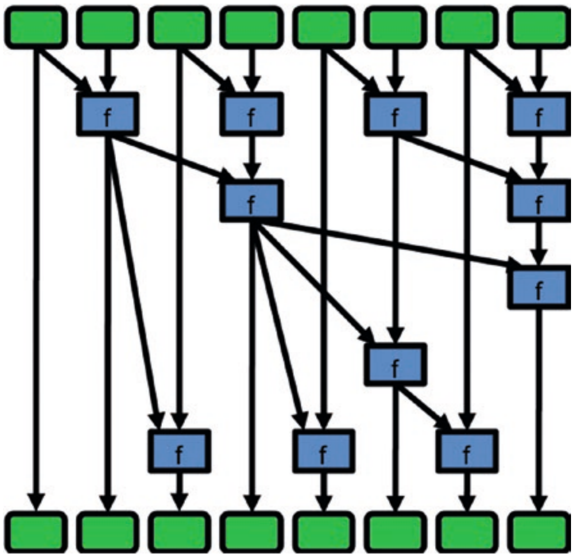

Figure 8-6. *Scan pattern: the complexity gives a visualization of the extra operations needed to offering scaling.*

```
void my_add_iscan(
  const float a[],     // input array
  float b[],           // output array
  size_t n ) {         // number of elements
  if (n>0) b[0]=a[0]; // equiv. to assuming b[i-1] is zero
  for (int i=1; i < n; ++i)
    b[i] = b[i-1] + a[i]; // iterations depends prior
}
```

Figure 8-7. *Serial code doing a scan operation*

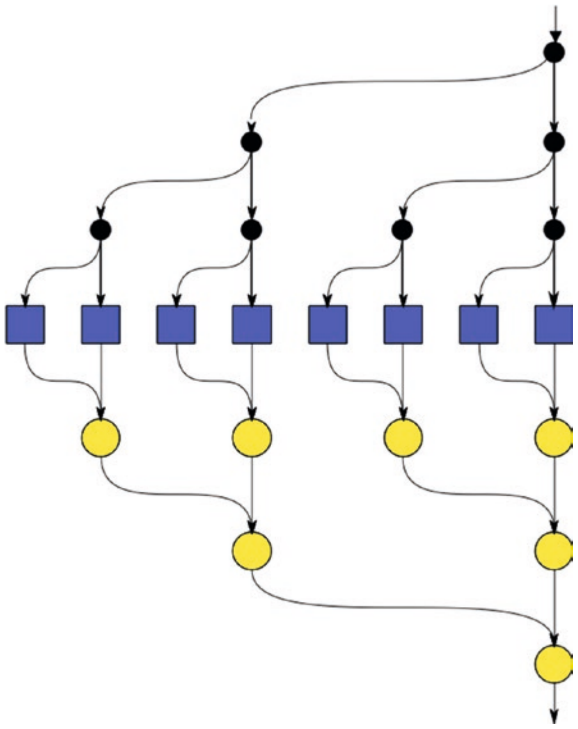

***Figure 8-8.** Fork-join pattern: allows control flow fork into multiple parallel flows that rejoin later*

Fork-Join Pattern

The **fork-join** pattern (Figure 8-8) recursively subdivides a problem into subparts and can be used for both regular and irregular parallelization. It is useful for implementing a **divide-and-conquer** strategy (sometimes called a pattern itself) or a **branch-and-bound** strategy (also, sometimes called a pattern itself). A fork-join should not be confused with barriers. A barrier is a synchronization construct across multiple threads. In a barrier, each thread must wait for all other threads to reach the barrier before any of them leaves. A join also waits for all threads to reach a common point, but the difference is that after a barrier, all threads continue, but after a join, only one does. Work that runs independently for a while, then uses barriers to synchronize, and then proceeds independently again is effectively the same as using the map pattern repeatedly with barriers in between. Such programs are subject to Amdahl's Law penalties (see more in the Preface) because time is spent waiting instead of working (serialization).

We should consider `parallel_for` and `parallel_reduce` since they automatically implement capabilities that may do what we need if our needs are not too irregular. TBB templates `parallel_invoke` (Chapter 2), `task_group` (Chapter 10), and `flow_graph` (Chapter 3) are ways to implement the fork-join pattern. Aside from these direct coding methods, it is worth noting that fork-join usage and nesting support within the implementation of TBB makes it possible to get the benefits of fork-join and nesting without explicitly coding either. A `parallel_for` will automatically use an optimized fork-join implementation to help span the available parallelism while remaining composable so that nesting (including nested `parallel_for` loops) and other forms of parallelism can be active at the same time.

Divide-and-Conquer Pattern

The **fork-join** pattern can be considered the basic pattern, with **divide-and-conquer** being a strategy in how we fork and join. Whether this is a distinct pattern is a matter of semantics, and is not important for our purposes here.

A divide-and-conquer pattern applies if a problem can be divided into smaller subproblems recursively until a base case is reached that can be solved serially. Divide-and-conquer can be described as dividing (partitioning) a problem and then using the map pattern to compute solutions to each subproblem in the partition. The resulting solutions to subproblems are combined to give a solution to the original problem. Divide-and-conquer lends itself to parallel implementation because of ease of which work can be subdivided whenever more workers (tasks) would be advantageous.

The `parallel_for` and `parallel_reduce` implement capabilities that should be considered first when divide-and-conquer is desired. Also, divide-and-conquer can be implemented with the same templates which can serve as methods to implement the fork-join pattern (`parallel_invoke`, `task_group`, and `flow_graph`).

Branch-and-Bound Pattern

The **fork-join** pattern can be considered the basic pattern, with **branch-and-bound** being a strategy in how we fork and join. Whether this is a distinct pattern is a matter of semantics and is not important for our purposes here.

Branch-and-bound is a *nondeterministic* search method to find one satisfactory answer when many may be possible. Branch refers to using concurrency, and bound refers to limiting the computation in some manner – for example, by using an upper bound (such as the best result found so far). The name "branch and bound" comes from the fact that we recursively divide the problem into parts, then bound the solution in each part. Related techniques, such as alpha-beta pruning, are also used in state-space search in artificial intelligence including move evaluations for Chess and other games.

Branch-and-bound can lead to superlinear speedups, unlike many other parallel algorithms. However, whenever there are multiple possible matches, this pattern is nondeterministic because which match is returned depends on the timing of the searches over each subset. To get a superlinear speedup, the cancellation of in-progress tasks needs to be implemented in an efficient manner (see Chapter 15).

Search problems do lend themselves to parallel implementation, since there are many points to search. However, because enumeration is far too expensive computationally, the searches should be coordinated in some way. A good solution is to use a branch-and-bound strategy. Instead of exploring all possible points in the search space, we choose to repetitively divide the original problem into smaller subproblems, evaluate specific characteristics of the subproblems so far, set up constraints (bounds) according to the information at hand, and eliminate subproblems that do not satisfy the constraints. This elimination is often referred to as "pruning." The bounds are used to "prune" the search space, eliminating candidate solutions that can be proven will not contain an optimal solution. By this strategy, the size of the feasible solution space can be reduced gradually. Therefore, we will need to explore only a small part of the possible input combinations to find the optimal solution.

Branch-and-bound is a nondeterministic method and a good example of when nondeterminism can be useful. To do a parallel search, the simplest approach is to partition the set and search each subset in parallel. Consider the case where we only need one result, and any data that satisfies the search criteria is acceptable. In that case, once an item matching the search criteria is found, in any one of the parallel subset searches, the searches in the other subsets can be canceled.

Branch-and-bound can also be used for mathematical optimization, with some additional features. In mathematical optimization, we are given an objective function, some constraint equations, and a domain. The function depends on certain parameters. The domain and the constraint equations define legal values for the parameters. Within the given domain, the goal of optimization is to find values of the parameters that maximize (or minimize) the objective function.

The parallel_for and parallel_reduce implement capabilities that should be considered first when branch-and-bound is desired. Also, divide-and-conquer can be implemented with the same templates which can serve as methods to implement the fork-join pattern (parallel_invoke, task_group and flow_graph). Understanding TBB support for cancellation (see Chapter 15) may be particularly useful when implementing branch-and-bound.

Pipeline Pattern

The **pipeline** pattern (Figure 8-9) can be easily underestimated. The opportunities for parallelism through nesting and pipelining are enormous. A pipeline pattern connects tasks in a producer-consumer relationship in a regular, nonchanging data flow.

Conceptually, all stages of the pipeline are active at once, and each stage can maintain state which can be updated as data flows through them. This offers parallelism through pipelining. Additionally, each stage can have parallelism within itself thanks to nesting support in TBB. TBB parallel_pipeline (Chapter 2) supports basic pipelines. More generally, a set of stages could be assembled in a directed acyclic graph (a network). TBB flow_graph (Chapter 3) supports both pipelines and generalized pipelines.

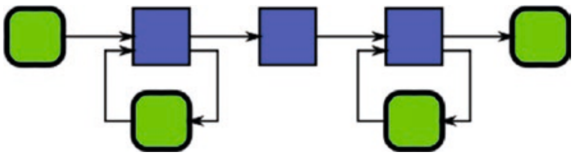

Figure 8-9. *Pipeline pattern: tasks connected in a regular nonchanging producer-consumer relationship*

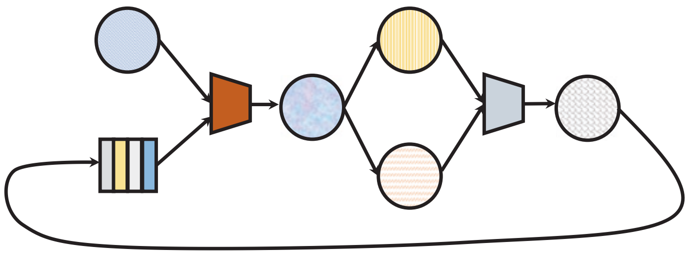

Figure 8-10. *Event-based coordination pattern: tasks connected in a producer-consumer relationship with an irregular, and possibly changing, interaction between tasks*

Event-Based Coordination Pattern (Reactive Streams)

The **event-based coordination** pattern (Figure 8-10) connects tasks in a producer-consumer relationship with an irregular, and possibly changing, interaction between tasks. Dealing with asynchronous activities is a common programming challenge.

This pattern can be easily underestimated for the same reasons many underestimate the scalability of a pipeline. The opportunities for parallelism through nesting and pipelining are enormous.

We are using the term "event-based coordination," but we are not trying to differentiate it from "actors," "reactive streams," "asynchronous data streams," or "event-based asynchronous."

The unique control flow aspects needed for this pattern led to the development of the `flow_graph` (Chapter 3) capabilities in TBB.

Examples of asynchronous events include interrupts from multiple real-time data feed sources such as image feeds or Twitter feeds, or user interface activities such as mouse events. Chapter 3 offers much more detail on `flow_graph`.

Summary

TBB encourages us to think about patterns that exist in our algorithmic thinking, and in our applications, and to map those patterns only onto capabilities that TBB offers. TBB offers support for patterns that can be effective for scalable applications, while proving an abstraction dealing with implementation details to keep everything modular and fully composable. The "super pattern" of nesting is very well supported in TBB, and therefore TBB offers composability not associated with many parallel programming models.

For More Information

TBB can be used to implement additional patterns which we did not discuss. We highlighted what we have found to be key patterns and their support in TBB, but one chapter can hardly compete with entire books on patterns.

Structured Parallel Programming by McCool, Robison, and Reinders (Elsevier, 2012) offers a hands-on coverage of "patterns that work." This is a book for programmers looking to have a more in-depth look at patterns with hands-on examples.

Patterns for Parallel Programming by Mattson, Sanders, and Massingill (Addison-Wesley, 2004) offers a much deeper, and more academic, look at patterns and their taxonomy and components.