Whitepaper

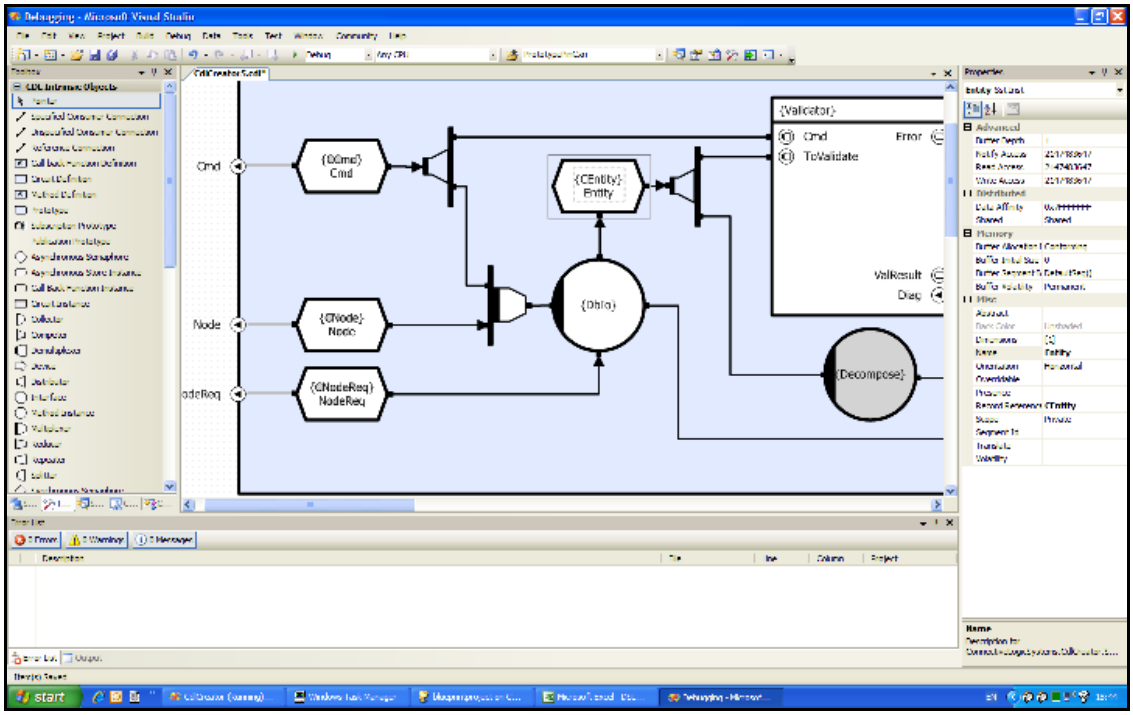
Connective Logic Infrastructure Programming (CLIP)

A Technology Overview

http://www.connectivelogic.co.uk

1 What Is Connective Logic?

Connective logic is a high level object oriented programming paradigm that provides an intuitive event based approach to concurrency. It underpins the Concurrent Description Language (CDL) which utilizes a dozen or so object primitives. Most engineers that have used CLIP have found it considerably easier than having to become familiar with threads, locks, semaphores and sockets. In the latter case, there are a lot of other lessons that also have to be learned in order to avoid the more common pitfalls of threading and messaging. These include dealing with tortuous problems like races, deadlocks and log-jams which are amongst the most difficult bugs to repeat and locate. Because CLIP's symbolism is 100% translatable, CLIP allows Domain Experts, with little or no programming experience to 'fix' the application 'design' and produce significant quantities of highly optimized executable code. Most engineers can continue to work with sequential code using familiar languages and tools.



In practice, most programs only require minimal use of CDL in order to achieve large scalable concurrency and most existing code remains identical. This would also be true of conventional threading, but the benefit of CDL is that it operates at a much more intuitive level, and for those that find visualization useful, it can also be presented as diagrams that can then be translated into conventional source code (this significantly reduces the time taken to code parallel applications).

CDL should not be confused with UML; the latter is a completely general purpose modeling/visualization aid that addresses all areas of program (and other system) design; CLIP is a programming language (100% compile-able) with far less symbolism, and only addresses the parallel execution of concurrent software applications.

So in a sense CLIP is analogous with the form designers that GUI developers use; it only does a specific part of the job, but it does it in a way that reduces and simplifies the task. It does not require learning a complex new language, and because it localizes all concurrency issues to a few diagrams rather than being dispersed throughout the whole application, in practice only a few engineers need to become involved with it.

In order to optimize what can be done with multi-core technology, it is important to make sure that as execution progresses, any function anywhere in the system that is ready for execution, is scheduled for execution. This turns out to be a non-trivial problem for data-parallel functions with data dependent execution time and even more difficult for the general (and extremely common) case of irregular concurrency. Along with Amdahls law, these phenomena contribute to Intel's rough estimate that suggests that quad core CPUs for example, are doing well if they can execute your parallelized

application 3.4 times faster than its sequential equivalent (see Optimizing Software for Multicore Processors).

There are at least four considerations facing any engineer that is developing parallel code, regardless of methodology; these are granularity, exclusion, dependency and data-lifecycle. Granularity determines the size of concurrently executable functions and impacts scalability and complexity, exclusion provides techniques that prevent two or more threads of execution corrupting shared data, dependency ensures that functions cannot execute until their inputs are ready (and there is space available for their outputs), and finally, data-lifecycle logic is required to 'free' data that is no longer referenced. This article will show how CDL deals with each of these and other issues.

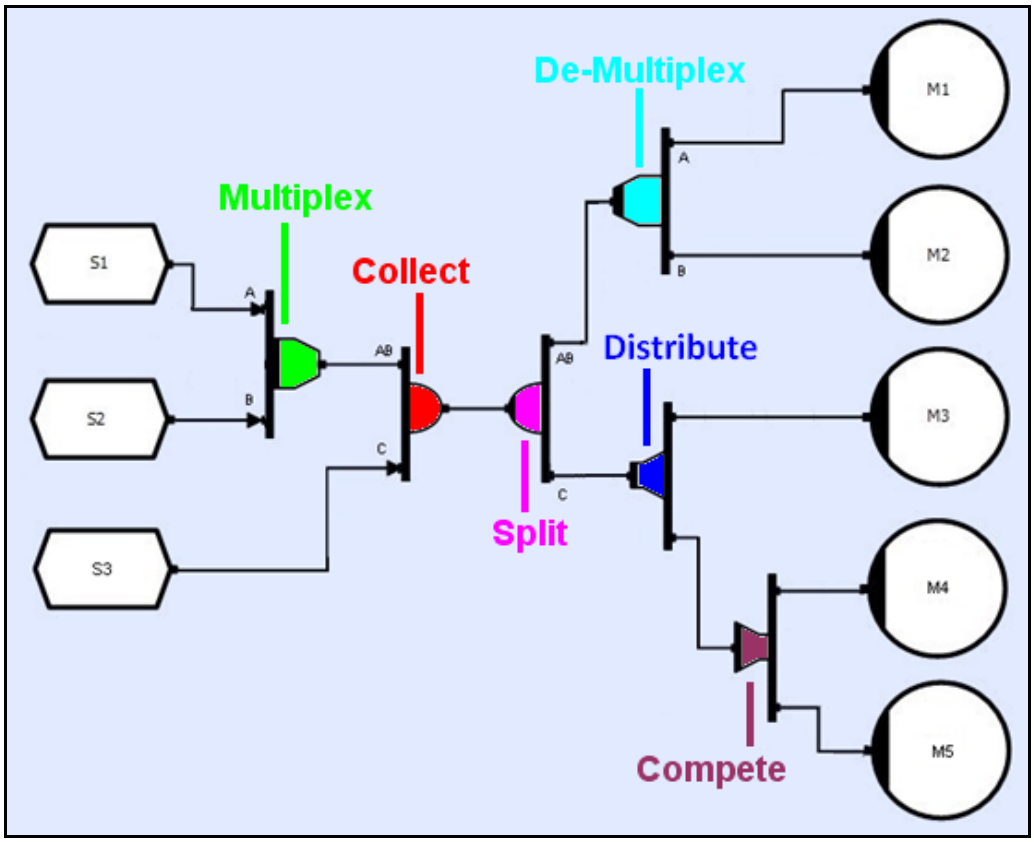
2 CDL Primitives

This section introduces some of the most commonly used CDL primitives. Events are produced by 'leaf' objects (stores and semaphores), propagate through event operators (colored objects below), and are eventually consumed by 'root objects (threads, methods and call-backs). Root objects 'contain' developer code in a similar manner to GUI call-backs. By default, references to the 'leaf' events are passed to the developer code which is then executed. The code which manages this is referred to as 'harness' code and is generated from the circuitry by the translator.

At runtime, methods and call-backs (see below) automatically generate requests to their providing objects which set off a chain of requests that propagate outward to the circuitry's leaf objects. This 'request' cycle will continue until blocked. As later events become available, the requests will progress until the whole 'tree' is available, at which point the root requester(s) become executable and are scheduled.

When each root object returns from execution, the generated harness code executes a 'close' on its providing object which then sets off a chain of 'closes', all the way back to the leaf objects. Closing leaf objects typically generates further events (e.g. closing a store for 'write' will typically unblock a consumer blocked on an awaited 'read'); and in this way, the calculation progresses through the designated circuitry. When a method or call-back closes, the scheduler will automatically generate another request and so the cycle repeats. In most cases, the latency of this cycle is hidden by the asynchronous nature of the connective logic engine. Although beyond the scope of this introduction, all objects have a notion of reentrancy which means that while a root object is executing with its current event(s), the next 'request' cycle can already be asynchronously executing. This considerably reduces latencies associated with 'network' scope applications.

The most important property of the event operator objects is that they are arbitrarily compose-able and so there is no limit to the number of patterns that can be produced. Circuits (see next section) are self-distributing concurrent equivalents of conventional 'classes' and so 'circuitry' (such as that below) can be encapsulated, archived into libraries and re-used. Because CDL operators are compose-able, circuits are also (usually) compose-able which leads to extremely rapid 'drag-drop-connect' application development.



2.1 Store Objects

CDL supports two types of 'store' objects, and these are used as repositories for share-able data.

2.1.1 Transient Stores (TST)

Transient stores (see S1, S2, and S3 in the figure above) provide a share-able alternative to conventional stack data. Reads are destructive, and when combined with 'distributors', have a reference counting capability. Data is 'only freed' when the last reader 'closes' (when the data is no longer referenced). Writes are also share-able, and in this case, the write is considered 'complete' when the last writer 'closes' (no longer 'write' referenced). Writers block when the store is full, and readers block when the stores is empty.

Data is propagated by reference between objects in the same memory space, but transparently moved and cached between disparate memory spaces. When data is no longer referenced all copies (including remotely cached copies) are automatically 'freed' by the runtime. The developer always sees data by reference (see the Single Virtual Process (SVP) model in the holographic processing paper).

2.1.2 Arbitrated Stores (AST)

Arbitrated stores provide a means of sharing persistent data (reads are non-destructive). Read, write and update access is arbitrated according to a number of configurable schemes. Typically, a store can be accessed by a single writer, a single updater, or any number of sharing readers. Deadly embraces are mitigated through the use of sequential collectors (see below), and these provide a means of ensuring that ASTs are accessed in a consistent order. If required, distributors can be used to ensure that sharer groups access consistent instances of store data. The use of collectors and distributors means that library circuits are usually compose-able.

As with ASTs, data is always accessed by reference and in cases where sharers are in disparate memory spaces, the runtime transparently moves data to maintain consistency.

2.2 Event Operators

CDL supports a number of event operators (colored objects in the illustration above) and these have evolved through use with a wide range of applications that have been developed since 1995. Experience suggests that they represent a necessary and sufficient set of operations for general case concurrency. They operate asynchronously in all cases, but have 'passive' implementations (there are no hidden threads created). Event operators are arbitrarily nestable and compose-able, so collection can be collected and so on.

2.2.1 Multiplexors (MPX)

The multiplexor object passes 'any' event that is provided to any of its inputs; to whichever object is consuming from it. In the example therefore; all writes to either S1 or S2 will be passed to the collector along the AB connection. This provides analogous functionality to Windows' WaitForMultipleObjects, or UNIX 'Select' calls.

2.2.2 Demultiplexors (DMX)

The demultiplexor object provides reciprocal functionality to the multiplexor and allows events to be routed according to the particular connection that they arrived on. In the example, the multiplexed event could arrive from either the S1 or S2 store, and following demultiplexing, would be routed to method M1 or M2 respectively. Collector/splitter and multiplexor/demultiplexor operations are arbitrarily nestable, and as in the example, can be separated by any composition of intermediate operations (in this case collection and splitting).

2.2.3 Collectors (CLX)

The collector object waits until 'all' of its inputs are provided, and then passes this 'compound' event (containing references to all of the inputs) to whichever object is consuming from it. There are two modes of operation; in 'sequential' mode the collector will collect in link-order regardless of the order

that provided events become available (avoiding deadly embraces), and in 'random' mode it will collect in the order that events arrive.

Collectors (like all event operators) operate asynchronously and so as well as addressing non-determinism, they also significantly reduce latency. In the example, the collector collects two events (in any order); the AB connection consumes a 'multiplexed' event containing either S1 or S2 store events; and the C connection consumes an S3 store event. In this example the resulting compound event is passed directly to a splitter object.

2.2.4 Splitters (SPX)

The splitter object provides reciprocal functionality to the collector and allows compound events to be broken back out into their component parts (which could themselves be further split-able compound events). In typical cases compound events pass through intermediate operators (e.g. distributors, competers etc) before being split, but in this simple example the collector event (containing S1 or S2; and S3) is split back to the multiplexed S1 or S2; and the S3 events respectively. As discussed above, collector/splitter and multiplexor/demultiplexor operations are arbitrarily nestable.

2.2.5 Distributors (DBX)

The distributor object propagates a reference to its consumed event, to each of its consumers (analogous to a multi-cast). When all of the consumers have 'closed' (and the distributed event is no longer referenced) the distributor will 'close' (setting off a chain of 'closes' back to the leaf object(s) that provided the event(s)).

2.2.6 Competers (CPX)

The competer object propagates its consumed events to its consumers in the order that they issue requests; so in a sense, it provides the 'opposite' functionality to a distributor which 'shares' its inputs with each of its consumers.

2.3 Root (Active) Objects

Active objects are repositories for processing logic (also called algorithmic and/or business logic). These objects execute the application's sequential code (grains), and so most engineers are therefore involved with these objects, and as such, do not need to become embroiled in the distraction of parallel execution, and can continue to work with familiar languages and tools.

2.3.1 Methods (MTHD)

Methods are analogous in many ways to GUI call-backs; that is to say that events are passed to them for non-blocking (or transient blocking) execution. In the example above, methods M1 to M5 will execute on arrival of their appropriate 'trigger' events which typically contain a number of references to input and output buffers. Method code is transparently scheduled and executed by runtime managed worker threads and so the underlying core-count is abstracted from the application.

When the method's processing code returns, the runtime will automatically 'close' the triggering event tree, tidy up any unreferenced memory, and schedule tasks that become executable as a result. The developer just needs to provide an 'entry point' for execution.

Methods can also own 'manual' connections which allow the sequential code to explicitly access CDL objects, devices etc, and this is provided as a means of dealing with conditional (data dependent) execution. Methods are permitted to block, and if required, the runtime can automatically spawn additional worker threads to maintain parallelism.

2.3.2 Call-Backs (CBF)

Call-Backs are similar to methods but instead of being executed by the worker pool, they are executed by the application's message pump (typically the GUI thread). This provides a high level portable interface between an application's GUI, and its 'back-end' processing. This simplifies the development of GUI oriented applications like Word Processors and Integrated Development Environments.

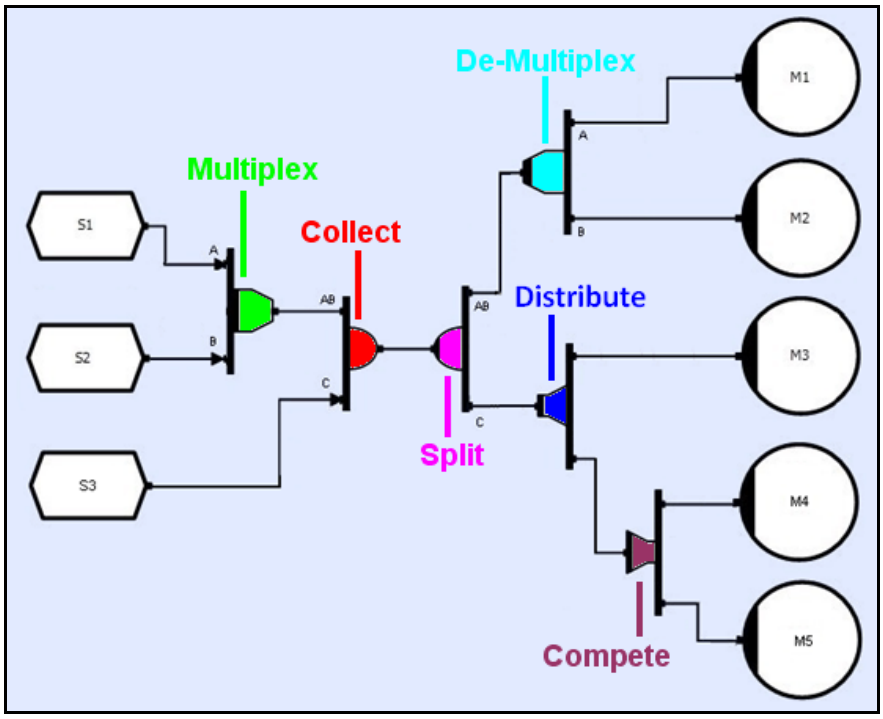
2.3.3 Threads (THRD)

CDL threads provide a portable interface to operating system threads. They are seldom required, but are provided as a means of dealing with some low level I/O operations that involve blocking an application outside of the runtime.

For a definitive list of CDL primitives and more detailed discussion see CLIP Documentation.

2.4 The Example Circuitry

The circuitry below would therefore provide the following (slightly pointless, but hopefully illustrative) functionality;



1. When S1 or S2 are written-to, the multiplexor will pass 'ready-for-read' events, to S1 or S2 respectively, along the AB connection to the collector
2. The collector will block until there is at least one event on both the AB and C connections. So this could either be S3 and S1, or, S2 and S1
3. The collector then passes the compound event (containing 2 store data events) to the splitter. Note that if all of the circuitry is in a shared memory space, that only references are actually passed
4. The splitter then separates the compound event back into its 2 components (rather pointless in this example)
    a. The multiplexed event (containing either S1 or S2) is passed along the AB connection to a de-multiplexor. If the event originated from S1 then a reference to the S1 data will be passed to the M1 method which will then execute its allocated code and return. When the method code returns, the de-multiplexor will automatically be closed, which will automatically close the AB splitter link. Closing the splitter automatically closes the multiplexor, which then closes the store, which then frees up the referenced buffer for write access (and unblocks any pending writers). If the event originated in the S2 store, then the functionality would be identical, except that the M2 method would be executed (as opposed to the M1 method)
    b. The S3 store event is passed along the C splitter link to the distributor. The distributor passes the event to 2 consumers; the first is M3 method (which executes and

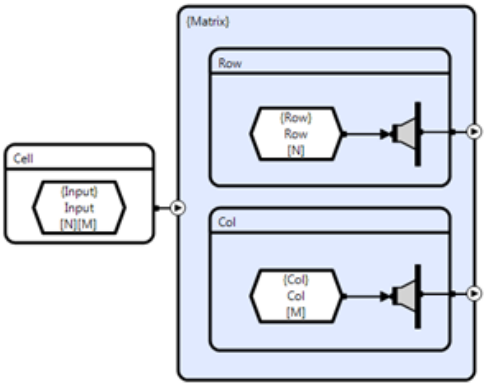
returns), and the second is a competer. The competer will then pass the S3 event to either the M4 or M5 method, depending upon which one queued its request first. So in a sense, competers provide the 'opposite' functionality to distributors. When M3 has executed and either M4 or M5 (whichever was the first to request), then the distributor will automatically close (both of its links have been closed). This will then cause S3 to close, and the referenced buffer made available for write (unblocking any pending writers).

Note that the order of write events is largely immaterial to the developer, as are the execution times of the methods. Managing non-determinism in this way considerably simplifies the developer's task. This topic is dealt with at more length in Edward Lee's highly recommended technical report; The Problem with Threads.

3 CDL Circuits

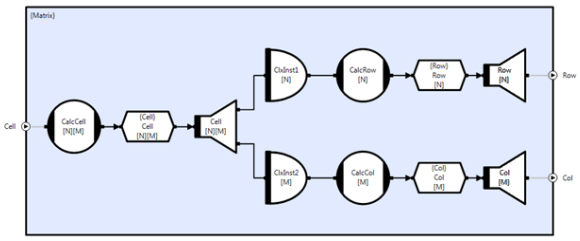
Most developers will be familiar with the concept of a 'class' in an OO sense, and the intention of Blueprint circuitry is to provide a concurrent (infrastructural) equivalent. Their purpose is therefore to localize, encapsulate, archive and re-use program logic (algorithmic and infrastructural).

A C++ class consists of a declaration where its members are specified and a definition where the code within those class members resides. Similarly, a circuit has a prototype that describes its public interface and it has a definition that contains the executable concurrent code.



Circuit Prototype (Declaration)

```
// Matrix class declaration
class Matrix
{
public:
   // Public interface
   void inputCell(
      Input in[N][M]);
      Row[] outputRow();
      Col[] outputCol();

private:
   // Internal state
   // ...
};
```



Circuit Body (Definition)

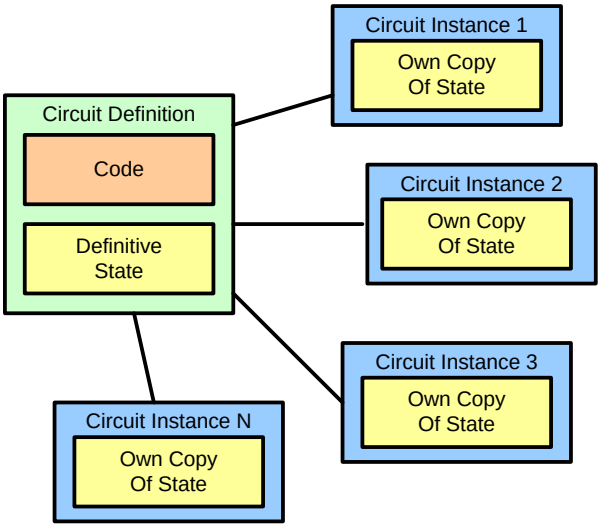
```
// Matrix class definition
void Matrix::inputCell(
   Input in[N][M]) {
   // Read cell data and begin
   // processing
}

Row[] Matrix::outputRow() {
   // Output row data
}

Col[] Matrix::outputCol() {
   // Output col data
}
```

Circuits may contain any of the following items:

- Blueprint's intrinsic primitives
- Instantiations of other circuit definitions
- References to external circuit instances.

The primitive symbols provide the fundamental language for defining the concurrent behavior of the circuitry. Instantiations and references allow encapsulated circuitry to be re-used by creating a new instance of the circuitry or connecting to an instance that was created elsewhere.

When a new instance of a class is created, it is allocated memory for its data members. This means that any number of classes can be instantiated and they all work on their own area of memory. Similarly, circuits have a concept of definitive state so there is no limit to the number of named circuit instances that can be created.

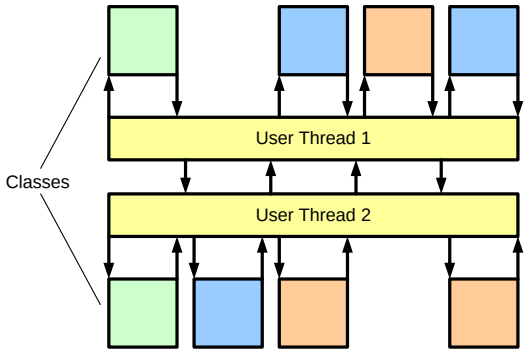


Circuit instances can be joined together in any way, provided that the prototypes at each end are compatible. This means that the complex array output of an FFT for example, can be directly connected to the input of a complex array filter, an RSS feed adaptor component could be connected to any number of business processes, and so on.

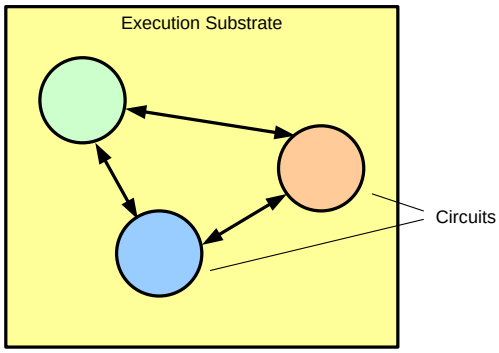
Classes are used for both modeling data (e.g. data collected from a sensor) and for defining active behavioral components (e.g. a sonar beam-former). Circuits can be freely interchanged with conventional classes but data classes are typically left as classes and exchanged between circuits that implement the active behavior. Due to this interchangeability, migration from legacy application classes to their circuit-equivalents can be performed incrementally. Conventional classes can invoke circuits; and vice-versa.

3.1 The Differences between Circuits and Classes

Whilst class instances are passive entities that are 'executed' by one or more explicitly created threads, circuit instances can be thought of as active entities that execute asynchronously and communicate through their connections. Synchronization is mostly achieved through the use of high level operations like collection, multiplexing, distribution, repetition and their reciprocal operations; splitting, de-multiplexing, competing and reduction. Explicit locking is provided but is seldom required.



Programmer uses threads to drive execution and explicitly invokes class methods



Circuits are self-propelled and methods are called automatically as the circuitry dictates

Circuit member functions (methods) can be multi-dimensional, and each element can own its own individual state; this provides implicit concurrency and in many cases replaces constructs like Parallel-For (see example above). It is also possible to specify the number of times that member function elements can be re-entered simultaneously without blocking (reentrancy).

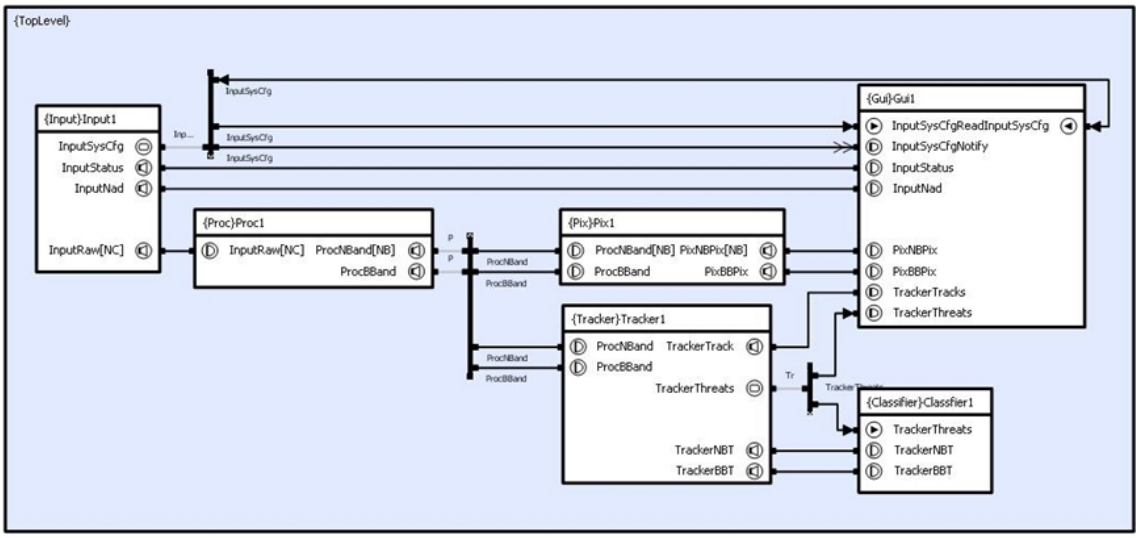
There is no limit to the concurrency that a circuit component or circuit instance can have. If a multi-dimensional component has multi-dimensional member components then the resulting concurrency will be the product of the two component concurrencies. If the application's synchronization logic permits two or more components to execute simultaneously then the resulting concurrency will be the

sum of the two or more component concurrencies, and so-on. This means that concurrency can be incrementally realized and accrued in a top-down manner, rather than starting with low level loops; but both approaches will work interoperably as appropriate (methods can contain TPL, TBB etc).

3.1.1 Building Applications from Components

Circuit components are intrinsically 'self distributing' and so developer code doesn't typically need to know anything about their internals in order to re-use them; it's usually just a case of connecting the appropriate input and output pins (a prototyped operation). Since most user defined circuits are arbitrarily compose-able, they can be archived into 'topic' libraries and re-used across projects. This means that applications can be created by domain experts using a drag-drop-and-connect metaphor.

The example below shows a simplified military sonar system constructed from re-usable components (circuit instances).



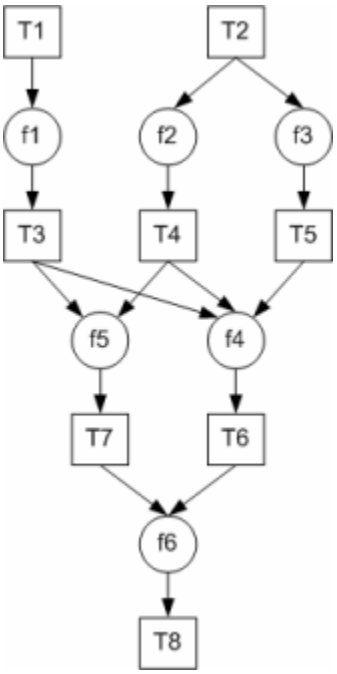
4 Example 1 - Irregular dependency

In order to introduce CDL it is probably best to start with the half a dozen or so most commonly used CDL objects, and show how these can be used to parallelize most existing sequential computations. We can start by considering the parallelization of a very simple sequential function (see below).

Note that the order in which the sequential code executes is to some extent arbitrary. We could for example calculate t5 before t4 without affecting the outcome. Experience suggests that a domain expert, given a whiteboard and a marker pen, is just as likely to start with a sketch such as the one on the right, as they are to write pseudo code such as that on the left. Crucially, the description on the right contains considerably more information than the code on the left (it tells us how we can distribute the calculation).

If we assume for this example that all of the subroutine calls made by f0 are of sufficient weight to justify farming out as separate tasks (granularity), and if we also assume that there are no secret data sharing issues (requiring exclusion), then the remaining considerations are dependency and data life-cycles.

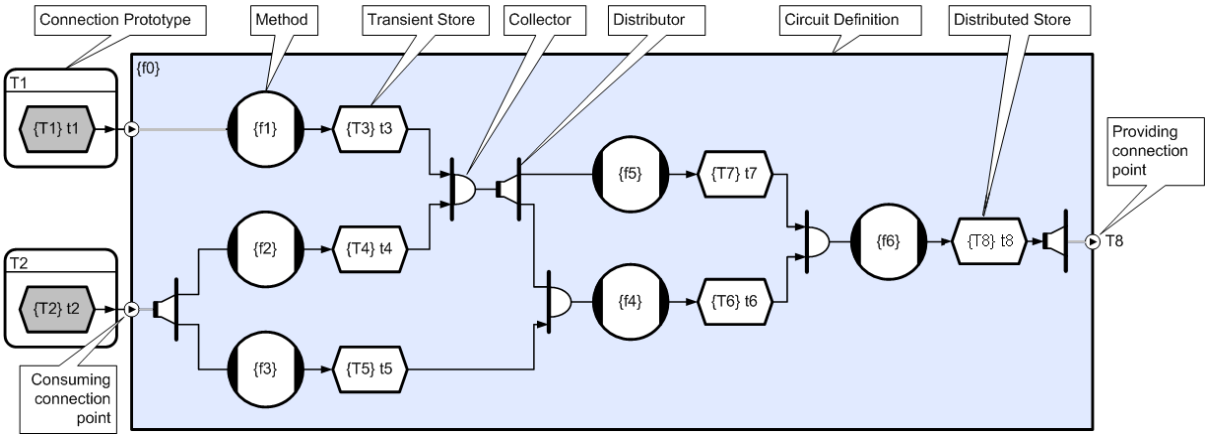
```
T8 f0( T1 t1, T2 t2 )
{
   T3 t3;
   T4 t4;
   T5 t5;
   T6 t6;
   T7 t7;
   T8 t8;

   t3 = f1( t1 );
   t4 = f2( t2 );
   t5 = f3( t2 );
   t6 = f4( t3, t4, t5 );
   t7 = f5( t3, t4 );
   t8 = f6( t6, t7 );

   return t8;
}
```



We start with a very simple function f0 (above) which does not contain any looping or branching. We will assume that we have no way of knowing how long any of its subroutine calls are going to take because their execution times may vary according to their input data (the general case). The diagram on the right shows the dependencies for each call that the function on the left makes. The rule is simply that each subroutine call can only happen when all of its input arguments have been calculated.

If we assume that when the f0 function is executed, the two input arguments t1 and t2 are populated, this would mean that functions f1, f2 and f3 would be immediately executable (giving an initial concurrency of 3). When f1 and f2 have both executed (populating data objects t3 and t4), f5 can execute, and when f3 completes f4 can also execute. The first thing to note is that even in this simple case, without knowing execution times we cannot predict the actual runtime concurrency which will depend on whether f5 completes before f3 and so on.

In fact the diagram on the right is really just another way of expressing the sequential description on the left but in this case it makes the execution dependencies explicit, telling a scheduler what can and cannot be executed in parallel. What CDL does is to formalize this information into diagrams that can be translated into code that then ensures that at runtime f0's subroutines will execute correctly and with as much concurrency as the calculation allows.



In this particular case, most of the symbolism simply reflects the information in the dependency diagram above (zoom to read the text). It is worth mentioning at this point that most of the time parallelization of sequential computations (such as this example) only requires 'collection' and 'distribution' operations; but general case programming requires a wider range of operations in order to manage GUI interaction, I/O, real-time execution and so on.

Note that the inwardly pointing arrow on the connection points implies that the connection point is an event consumer whilst the outwardly pointing arrow implies that the connection point is an event provider. This should not be confused with read/write access.

We can now see what the f0 circuit is going to do when its inputs are provided;

1. f1 will execute as soon as its input is ready (a new value in t1) and there is space in t3 for its output.
2. f2 and f3 will both begin execution as soon as their shared input is ready (new value in t2), and space exists for their outputs (t4 and t5 respectively).
3. We cannot say whether all three will actually execute simultaneously because it depends on the arrival order of their input and of course the availability of space for their outputs. But the really important point is that we don't actually care because the collectors and distributors will ensure that they will execute as soon as they can, but will never execute until they are ready. As we shall see in our next example, it is the management of this indeterminism that is the key to creating massive concurrencies without the need to become buried in complexity (see Professor Lee's paper The Problem with Threads).
4. Once f1 and f2 have executed f5 becomes executable and once f3 completes f4 is similarly ready.
5. Finally, once f4 and f5 complete then f6 can execute and so the function completes by producing a result in f6 which will typically trigger execution of another sub-circuit which will also probably have multiple concurrency. In fact in the most general case, completion of f0 will trigger execution of more than one sub-circuit, each of which may contain their own sub-circuits, and so on. It is this compose-ability that enables CLIP applications to very rapidly accumulate very large concurrencies even in the non-trivial irregular case.
6. It is also worth pointing out here that most sub-circuits can actually be re-entered and if the data stores are multiply buffered, the f0 sub-circuit can actually achieve a concurrency of six under optimal conditions. If the methods f1 to f6 are stateless (do not have any instance specific data) then they can themselves be re-entered and in this case the only limit to concurrency is the number of available CPUs.

There are a couple more points to make before moving to the next example. Firstly, as mentioned above, CDL objects are arbitrarily compose-able. Collectors can feed into distributors that than can then be fed into collectors or distributors (as well as another half dozen or so event operators) and in this way any sequential case can have an optimally parallel equivalent. Because CDL operators are compose-able, CDL circuits are compose-able and this considerably reduces the chance that applications will be afflicted by deadlocks or races and perhaps most usefully of all, because CDL

primitives have known properties, and object connectivity is explicit, almost all potential deadlocks and/or races can be discovered by analysis of application circuitry (an automatable process).

As we can now see from the dependency diagram and the CDL equivalent, developing concurrent execution from sequential code can be a pretty methodical process and in most cases only requires knowledge of a few objects and some fairly simple rules. Most code remains unchanged and at no stage does the application programmer need to assume the core-count.

5 Example 2 Scheduling Latency

This example is similar to the first but for purposes of illustration considers how a problem might be addressed by the proposed 'futures' extensions to C++ (asynchronous function calls). In this case we consider the parallelization of a class (rather than a function) and in order to simplify the description we will use pseudo C++ syntax and imagine the existence of three functions that would make the task a bit easier. We will also assume the existence of a transparent worker thread pool that executes the asynchronously launched function calls. As in the first example, we will also assume that member functions do not share state. Consider the following very simple class which has one public member function and seven private member functions.

```
class FCalc
{
public:
   int F( A a, B b )
   {
      // Transient workspace objects
      C c; D d; E e; F f; G g; H h;
      CalcC( a, c );       // Calculate c from a
      CalcD( a, d );       // Calculate d from a
      CalcE( a, e );       // Calculate e from a
      CalcF( c, d, f );    // Calculate f from c and d
      CalcG( c, e, g );    // Calculate g from c and e
      CalcH( d, e, h );    // Calculate h from d and e
      CalcB( f, g, h, b ); // Calculate b from f, g, and h
   }
};
```

Again, what we are primarily interested in here are the issues arising from dependency considerations (making sure that functions aren't invoked until their inputs are ready), but we will also use this example to highlight some of the reasons that irregular concurrency is considered to be such a difficult problem, and in particular, why many multi-core applications fail to scale as predicted. In the two code examples that follow we have introduced 3 functions (purely for read-ability);

```
start( f0(a,..), f1(a,..),  );
```

This function is assumed to asynchronously launch each function in its argument list and automatically pass each of their respective arguments.

```
waitForAny( a, b,  );
```

This function is assumed to block until at least one of the data objects in its list is updated by an asynchronously executing function, and then return an identifier indicating (the first) value to be updated. This is therefore analogous to the Windows waitForMultipleObjects function call.

```
waitForAll( a, b,  );
```

This function is similar to the previous one but in this case it is assumed to block until all of the values have been updated.

First consider a very simple solution;

```
int F( A &a, B &b )
{
   // Transient workspace objects
   C c; D d; E e; F f; G g; H h;

   // Launch calcC, calcD and calcE which use a to calculate
```

```
   // c, d and e repectively
   start( CalcC( a, c ), CalcD( a, d), CalcE( a, e ) );

   // Now block and wait for c, d and e to be calculated
   waitForAll( c, d, e );

   // Now launch calcF, calcG and calcH which use c, d and e
   // to calculate f, g and h
   start( CalcF( c, d, f ), CalcG( c, e, g ), CalcH( d, e, h ) );

   // Now block and wait for f, g and h to be calculated
   waitForAll( f, g, h );

   return CalcB( f, g, h, b );
}
```

This solution is simple to create and understand but suffers from a limitation which is termed 'scheduling latency', and refers to the fact that because runnable tasks whose inputs are actually ready, cannot be executed immediately (the launching thread is blocked on the third result). This solution is unlikely to scale well under anything but perfect conditions. The problem stems from the fact that the first waitForAll call is waiting for 3 results when in practice, only two are required for the calculation to progress (the application is therefore unnecessarily blocked).

This is not the same thing as Amdahl's law which stems from fundamental limitations at the algorithmic level, but is important to raise, because although it looks a lot like Amdahl's effect, in most cases 'scheduling latency' can actually be addressed.

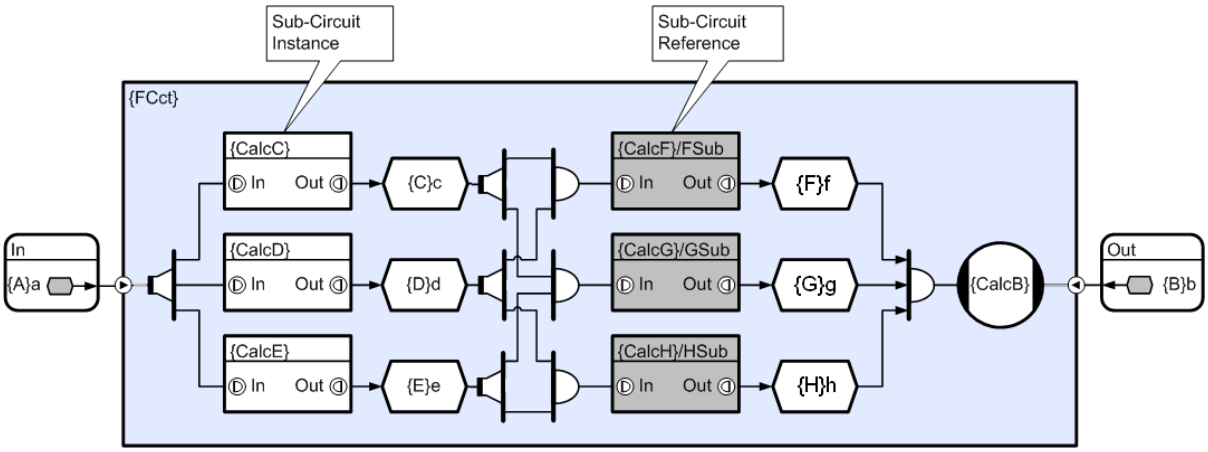
In order to illustrate an optimal solution we need to use the waitForAny call and then use the result of each call to decide what to do next. So if for example calcC and calcD complete first we could launch calcF but if calcC and calcE finish first we could launch calcG. It might sound easy, but the following code shows just how many combinations need to be considered for the general case where all execution times are data dependent. In fact, this solution still blocks the launching thread gratuitously but this final optimization is going to be ignored here.

The solution below is 'optimal' in a latency sense but is certainly not the most elegant solution. Whilst there are better solutions, in this case the number of lines of code has a factorial relationship with concurrency.

```
int F( A a, B b )
{
   // Transient workspace objects
   C c; D d; E e; F f; G g; H h;

   start( CalcC( a, c), CalcD( a, d), CalcE( a, e ) );
   objId = waitForAny( c, d, e );

   if ( objId == c )
   {
      objId = waitForAny( d, e );
      if ( objId == d )
      {
         start( CalcF( c, d, f ) );
         objId = waitForAny( e );
         start( CalcG( c, e, g ) );
      }
      else
      {
         start( CalcG( c, e, g ) );
```

```
         objId = waitForAny( d );
         start( CalcF( c, d, f ) );
      }
      start( CalcH( d, e, f ) );
   }
   else if ( objId == d )
   {
      objId = waitForAny( c, e );
      if ( objId == c )
      {
         start( CalcF( c, d, f ) );
         objId = waitForAny( e );
         start( CalcH( d, e, h ) );
      }
      else
      {
         start( CalcH( d, e, h ) );
         objId = waitForAny( c );
         start( CalcF( c, d, f ) );
      }
      start( CalcG( c, e, g ) );
   }
   else
   {
      objId = waitForAny( c, d );
      if ( objId == c )
      {
         start( CalcG( c, e, g ) );
         objId = waitForAny( d );
         start( CalcH( d, e, h ) );
      }
      else
      {
         start( CalcH( d, e, h ) );
         objId = waitForAny( c );
         start( CalcG( c, e, g ) );
      }
      start( CalcF( c, d, f ) );
   }

   waitForAll( f, g, h );

   return CalcB( f, g, h, b );
}
```

The circuit below illustrates a CDL solution and introduces a few more properties of the technique;



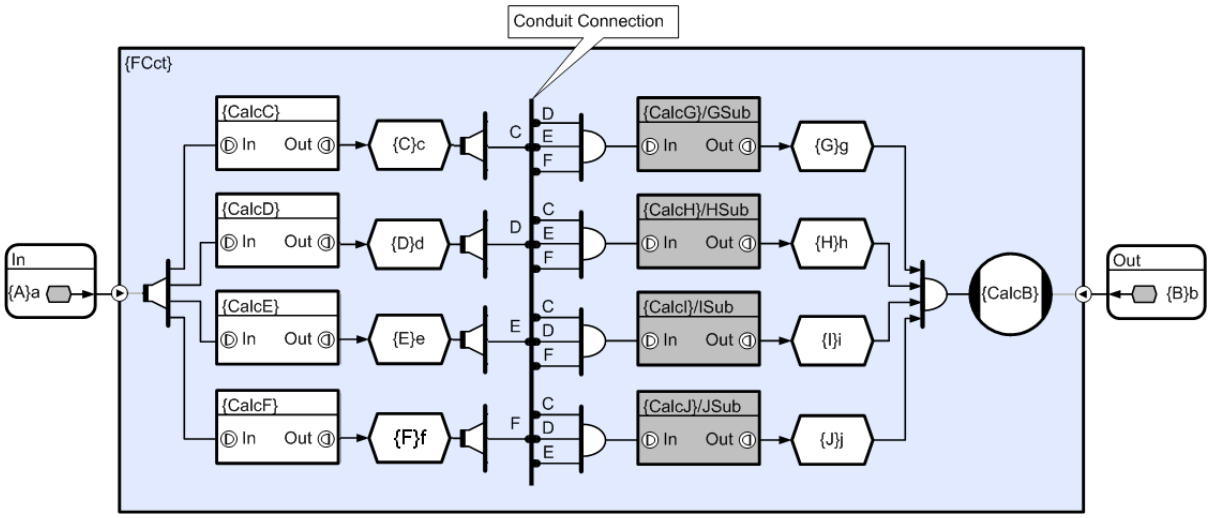
In the first example, the subroutine calls (f1 through f6) used CLIP methods which means that although these can execute in parallel with each other, each actual method invocation executes sequentially (so the potential concurrency is six). In many cases, the translation to CDL can continue to finer granularity and in the above diagram we have assumed that this is the case. This would obviously increase concurrency but would also increase scheduling overhead and deciding how far down to take CDL is an important design decision.

Although the sub-circuits in this example all happen to have one input and one output, this is not a restriction imposed by CDL, and they can in fact have an arbitrary number of entry points (equivalent to public member functions), and each of these can have any number of inputs and/or outputs.

The CalcC, CalcD and CalcE sub-circuits are white and the CalcF, CalcG and CalcH sub-circuits are colored gray. This is because in this example, the first 3 are members of the parent circuit and exist as instances with their own state, whilst the gray sub-circuits are actually calls to remote circuits that are instantiated somewhere else in the application. In the latter case these are typically references to sub-circuits on remote machines and the technique is provided for cases where circuits need access to particular devices (disks, sensors etc), or equally often because of specialist processing issues like those arising when high performance applications need to offload to specialist processing devices like an IBM Cell BE, ClearSpeed card or GPU. This would be analogous with a Service Oriented Architecture (SOA) view of the world.

Before moving onto the next example, it is worth considering how the complexity of the two approaches can scale. In the optimal (but naive) asynchronous solution above, 30 to 40 lines of code were required for an irregular problem with a concurrency of three. If the concurrency is increased to four, about 4 x 30 lines are needed, if concurrency is increased to 5 then about 5 x 4 x 30 (600 lines) are required and complexity continues to grow with factorial scaling. Clearly there are better solutions but they are non-trivial and this is left as an exercise for readers.

The circuit below addresses the case with concurrency four, but in this case, complexity scales linearly (in terms of the required number of objects and connections). In this particular example however, the number of crossed lines means that the diagram can become confusing and this example therefore introduces the notion of a conduit connection. In this case all connections to the conduit sharing common labels (in this case C, D, E, or F) are made. So in this case for example, GSUB's collector will collect inputs from the d, e, and f transient stores and when all three are ready, GSUB will be triggered.



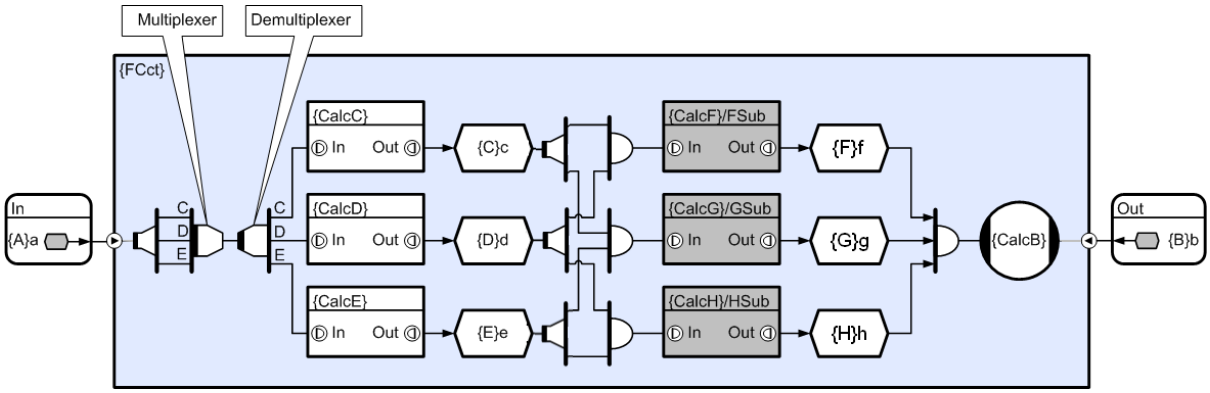
This illustrates a useful property of connective logic which is that timing indeterminism is addressed by the scheduler and not by the engineer. This simplifies the parallelization of the general irregular concurrency case considerably, and when this is combined with its other significant property (circuitry is compose-able) the task of achieving highly scalable applications under typical conditions becomes attainable for most applications.

6 Example 3 Exclusion through Scheduling

Broadly speaking there are two ways of addressing the problem of data races (where two threads of execution are able to access the same data inconsistently) and these are; scheduling exclusion, and data arbitration. In the first case, it is necessary to ensure that two pieces of code that access the same data are never scheduled simultaneously, and in the second, some kind of arbitration scheme that is associated with particular groups of data is required. In the arbitration case, a blocking scheme (usually called a lock or mutex) could be used, or alternatively one of the more recent techniques based on the concept of STM (software transactional memory) could be used. CDL supports both approaches but at the time of writing has yet to adopt STM for the latter.

In order to address this topic it is necessary to introduce two new CDL objects referred to as a multiplexor and a de-multiplexor. Like all CDL objects they have a concept of reentrancy which determines the number of times that they can be simultaneously opened by provider/consumer pairs. Reentrancy is an advanced topic that will not be discussed here, but in the special case where the de-multiplexor has a concurrency of one, they can be used as means of serializing pieces of code that we do not want to execute in parallel.

We can illustrate this by making a small modification to the previous example and thereby ensure that CalcC, CalcD and CalcE will never execute in parallel (although they can still execute serially in any order).



The explanation is as follows. When a new value is delivered to the circuit, it is replicated three times by the distributor and then serialized as three events by the multiplexor. The de-multiplexor then passes each event to the appropriate sub-circuit which then executes sequentially with respect to the two adjacent sub-circuits (but in parallel with all non-excluded circuitry). Note the use of labels on the links (C, D and E) which are used to map multiplexor inputs to de-multiplexor outputs.

Although not relevant to this example, if we raised the de-multiplexor concurrency to two (a creation attribute) then the sub-circuits would be able to execute two at a time, and if we raised it to three, we would be back to full parallel execution (therefore a bit pointless!).

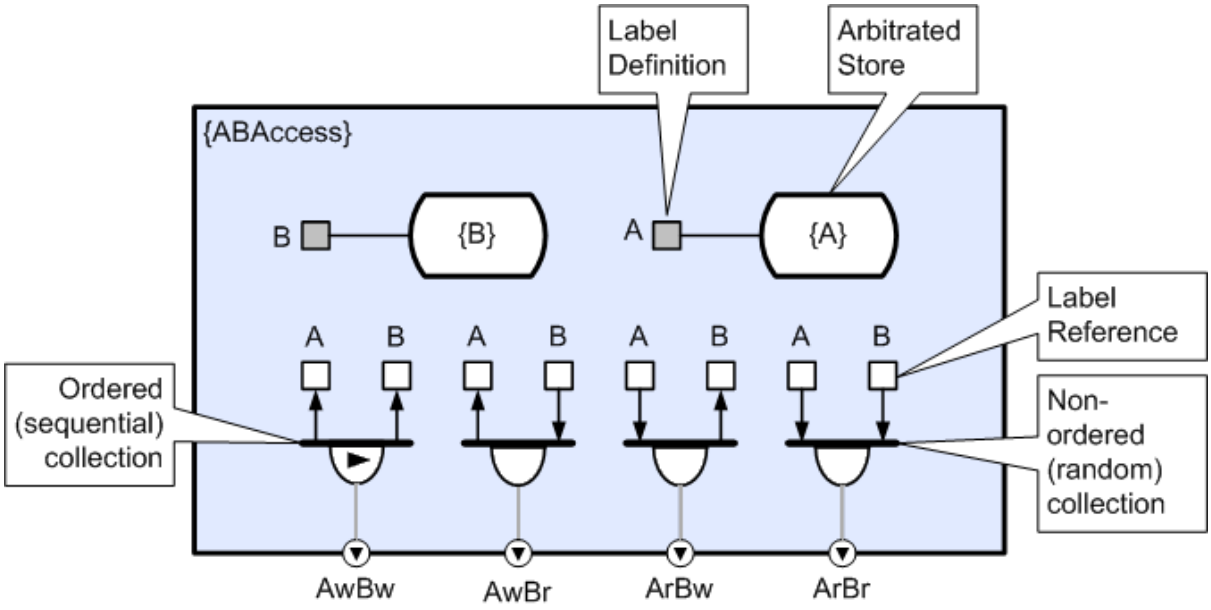
7 Example 4 Exclusion through Arbitration

The previous example provides a safe compose-able technique for avoiding data-races but unless the application's concurrency exceeds the core count of the target hardware, it could reduce overall performance by introducing a convoy effect. Under these circumstances a finer grained exclusion scheme is required and this introduces another CDL object referred to as an arbitrated store (AST).

To continue the class/circuit analogy, ASTs are used as repositories for circuit state (analogous to object state). There is no limit to the size of data object that they arbitrate and again the decomposition of state into smaller grain ASTs is another important design consideration. Coarse grain store schemes are easier to manage, but finer grain schemes are less likely to introduce unnecessary scheduling latencies through convoy effects.

ASTs differ from their transient equivalent used in earlier examples in that their data persists (is not invalidated when no longer referenced) and access to their data is restricted by one of a number of configurable exclusion schemes. Typically, stores allow any number of readers, or one writer, but if the store is double buffered then readers can access the current value whilst one writer is updating the other which then becomes the current value for subsequent read accesses. This provides safe arbitration and greatly reduces blocking; which can only occur if more than one simultaneous write/update is attempted.

However, because multiple writers could still block, this could lead to deadly embraces in exactly the same way that conventional locks do (the store has no implicit advantage over other problematic schemes). But in practice they seldom occur because most AST access sequences are achieved through the use of CDL's ordered collectors (which ensure that stores are only accessed using particular sequences). Because all CDL objects have a notion of scope, data stores can be made inaccessible through anything other than published schemes.



This example illustrates how such schemes can be created and implemented. Note that in the example above, it is assumed that the store has been configured so that reads will never block, and writes/updates will only block if more than one simultaneous write/update is attempted (requires multiple buffers).

This example introduces the arbitrated store object and also the concept of labels and references. The shaded boxes define named labels, and the un-shaded (white) boxes represent references to them. So in this case, each collector is actually connected to both stores.

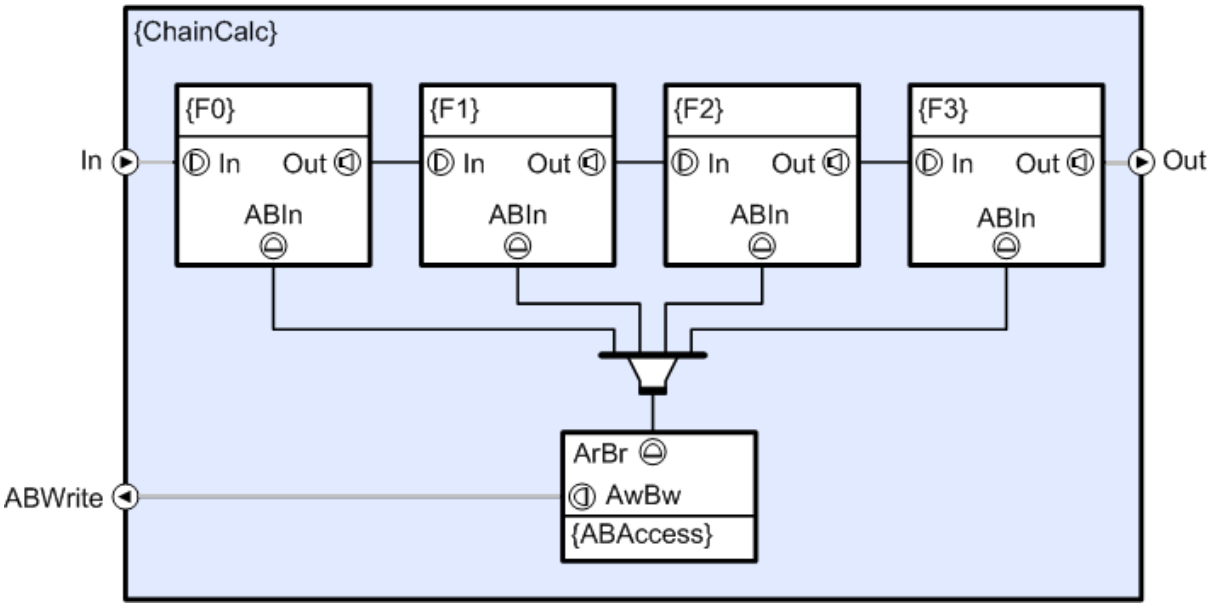
Also note, that because of the particular arbitration scheme used by this example circuit, only when A and B are simultaneously opened for write, could a deadlock occur and that is why only the AwBw collector needs to be ordered. Provided the stores are only accessed through these four entry points, deadlock cannot occur. The same approach applies in the general case where collections of collections of stores are required (collectors are nestable).

In the case where stores are publicly exposed (and sometimes there are good reasons for wanting to do this), then the CDL translator will spot most deadly embraces through analysis of connectivity. The only cases that the analyzer cannot detect are cases where methods access arbitrated stores for write during their execution and whilst this is permitted it is seldom required.

Another issue that often arises with persistent data is consistency. It could be that we have a long processing chain, and in order for program logic to work correctly, all stages must use a consistent latest value. Although we want readers to see a consistent value, we don't want to block updaters, and we may need the value to stay consistent for relatively long periods.

Under these circumstances it is quite common to use a distributor to ensure that each stage of the calculation uses the same AST data instance, and in practice this does not prevent the store being written to, and indeed it can do so any number of times without the need for data to be locked throughout a complete iteration. This avoids the convoy effect that can occur if a data object needs to be protected from change for extended periods (coarse locking). It also saves bus bandwidth by avoiding the need to make multiple gratuitous copies of the data object between stages in the processing chain.

When the last stage of the chain (F3) completes, the distributor will close, and this will then allow any writes/updates that occurred during the processing of the chain to become the current value.



One final point is that collectors do not behave in the same way that a sub-routine requesting a series of locks would behave because incomplete collections requested by CLIP methods do not block the execution of any particular worker thread whereas conventional threading models will block if any of the locks in the sequence are unavailable (although this may well turn out to be avoidable by a successful conclusion to current research with STM).

In fact CLIP method scheduling uses a technique that means that worker threads only ever block if the number of executable tasks dips below the number of available CPUs, or if the method explicitly accesses block-able objects (unnecessary but sometimes useful). This reduces context switching overhead as well as stack usage and is described in the holographic processing paper that accompanies this.

8 Example 5 Data Parallel Execution

This last example addresses the more familiar territory of data-parallel applications. This case is considered to be more familiar than those dealt with earlier because historically, high performance computing has tended to involve this type of technique (although this will arguably change with the arrival of multi-core). This section looks at how CLIP deals with the data-parallel sections of general concurrent applications.

Before moving to an actual example it is probably useful to differentiate various levels of data-parallelism. Firstly there is the most specialized case of all; Single Instruction Multiple Data (SIMD) which will execute efficiently on GPUs and other accelerator devices. Common applications are pixel rendering, Fast Fourier Transforms (FFTs) and numerical filters.

If a function branches on data (includes statements like if, switch, while, or data dependent length for-loops) then it is more likely to be a Multiple Instruction Multiple Data (MIMD) or some other more general class of problem and what this means is that it is going to be a lot more difficult to scale efficiently.

Since this latter case is considerably more typical of day to day programming requirements, the chosen example falls into this more general category and highlights some of the reasons that multi-core applications are not expected to scale linearly even in this idealized case. The example highlights how a form of scheduling latency can often be a major cause and explains a CDL technique for avoiding it. However, it is important to point out that this is not the only obstacle to perfect linear scalability and there are others relating to sheer physical constraints like bus bandwidth which CLIP will have no special means of avoiding.

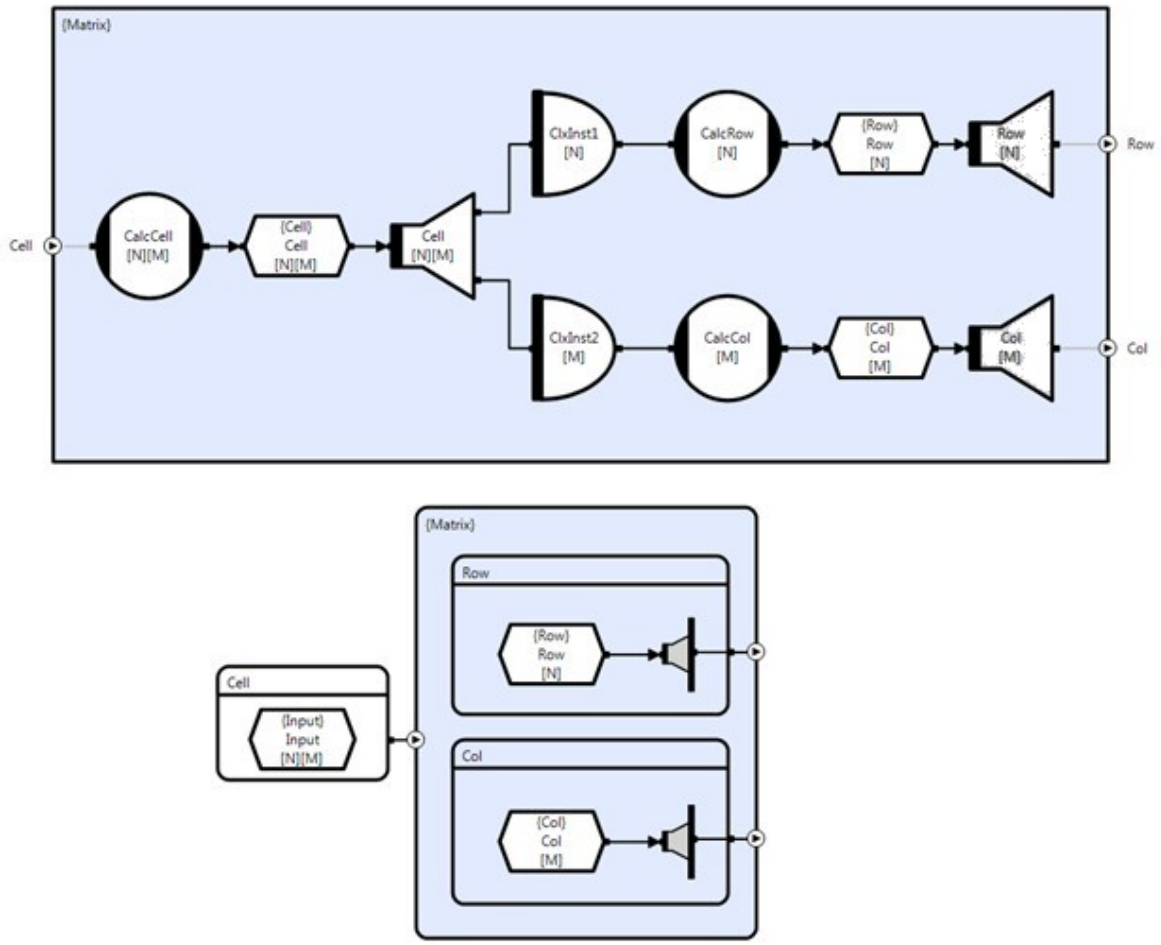
This example is specifically picked to highlight something that it can address which is a problem that arises from the use of processing barriers such as those employed by scatter/gather type mechanisms and others provided by OpenMP and MPI type approaches. As with the asynchronous programming approach discussed earlier we are not necessarily saying that scheduling latency cannot be minimized using these technologies, but we are pointing out that to achieve it may require an unexpected amount of effort.

```
void MatToColRow::Calc( A Input[N][M], B Rows[N], C Cols[M] )
{
   D Grid[N][M];
   for ( i = 0; i < N; i ++ )
   {
      for ( j = 0; j < M; j ++ )
      {
         CalcGrid( Input, i, j, Grid );
      }
   }
   for ( i = 0; i < N; i ++ )
   {
      CalcRow( Grid, i, Rows );
   }
   for ( j = 0; j < M; j ++ )
   {
      CalcCol( Grid, j, Cols );
   }
}
```

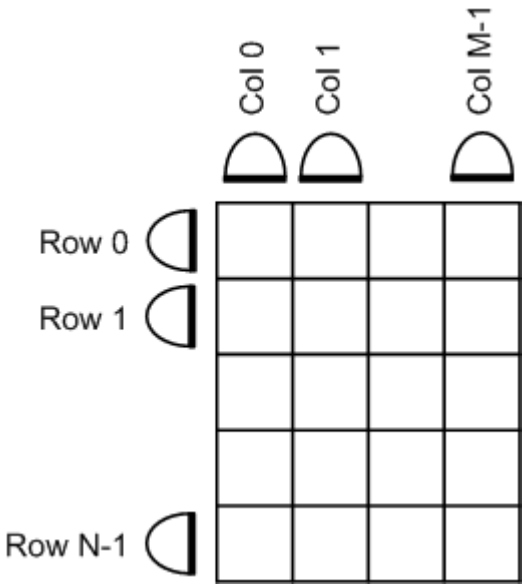
The example above is probably fairly typical of the sort of code that will need to be migrated to multi-core over the coming years. But even if we assume that it is embarrassingly parallel (no dependency between iterations) there are a number of issues that could prevent it from scaling in the general MIMD case where the execution time of each function call is dependent on its data. And it is generally true to say that the more exaggerated the variability, the less well it will perform.

Firstly there is the obvious point that if the loop count is not divisible by the number of cores then different CPUs will end up performing different numbers of tasks, and secondly, if each task takes a

different number of cycles then we have no way of knowing how each loop will run down as it synchronizes each of its CPUs. The point is that as each thread synchronizes at the end of the loop, parallelism will drop from N (typically the number of cores) all the way down to one (there has to be a last thread to complete) and while this is happening the scaling has to be less than N to 1 (not all CPUs are executing). So how can we avoid processing barriers and keep the concurrency above the core count?



The circuit above has the same functionality as the three loop example above that, but in order to explain how it works it is necessary to briefly introduce CDL's tensor style notation (connections between multi-dimensional objects). Without going into too much detail, the default rule is that repeated dimensions connect one to one (those shared by provider and consumer), whilst residual dimensions imply many-to-one operations like collection, multiplexing, distribution etc (the precise operation depends on the particular provider/consumer pair). It is worth mentioning that defaults can be over-ridden and irregular cases are supported, but this is only seldom required in practice for typical data-parallel applications.



In the example circuit above, each element of the [N] dimensioned collector therefore collects [M] elements from the grid (corresponding to a complete row). Conversely, each element of the [M] dimensioned collector therefore collects [N] elements from the grid (corresponding to a complete column).

So the example circuit has the property that as the two dimensional grid becomes populated by the calcGrid method, each row calculation eventually collects a complete row's worth of input, and each column calculation eventually collects a complete column's worth of data. There is no concept of a processing barrier and so the row and column calculations can proceed in whatever order their non-deterministic completion order dictates. This is important because in the worst case a scheme that chooses to calculate the complete grid first, then all rows, then all columns; could in the very worst case have 'all columns except one' available to execute (i.e. all inputs ready) but be completely stalled because it chose to execute rows first (and there are none actually ready). The worst case is unlikely to occur often, but then so is the best case, the CDL solution exhibits optimal scheduling latency in all cases.

With the CDL solution, row and column calculations become executable even before the complete two dimensional grid has been populated, and their completion order does not limit progress. Concurrency is further liberated by the likelihood that in this particular example, the grid calculation can also be re-entered whilst the row and column calculation (and even residual grid calculation) are still proceeding for the previous iteration.

This concludes the overview of the connective logic programming paradigm, and interested readers are invited to read the accompanying paper that explains how CDL is executed by its associated runtime (holographic processing).