John Gross and Jeremy Orme
john@connectivelogic.co.uk
jeremy@connectivelogic.co.uk

Feature article (Part 1)

Multi-Core OO – An Introduction to the Blueprint Toolset

The Problems Introduced by Concurrency

The era of multi-core programming has arrived. Many influential commentators have pointed out that concurrent programming requires developers to learn new techniques, different from those adopted for standard sequential programs. In their paper, Software and the concurrency revolution, Microsoft's Herb Sutter and James Larus describe how:

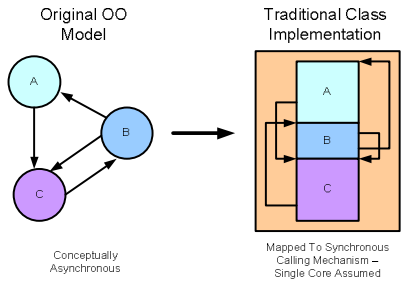
> *"humans are quickly overwhelmed by concurrency and find it much more difficult to reason about concurrent than sequential code".*

However, when humans perform everyday tasks like playing sport or driving cars, they demonstrate an innate ability to deal with complex concurrent situations. Even more ironically, standard Object-Oriented models, fundamental to modern software practice, provide an inherently concurrent view of the world that is divorced from its eventual mapping to processes and hence CPU cores.

Conceptually at least, OO allows asynchronously executing objects to commune with adjacent objects by sending messages to their member functions; and so at its highest level of abstraction object behavior is intrinsically parallel. So why isn't multi-core seen as an enabling technology that facilitates the physical implementation of such a familiar and intuitive logical abstraction?

The answer to this draws attention to another even more fundamental problem; the discontinuity that exists between highly abstracted software designs such as those provided by the UML; and the software implementations that manifest programmers' interpretations of these designs. When modeling systems (formally or informally), designers are able to think at a highly intuitive level of abstraction, and at this level most developers are able to visualize concurrency with very little effort. However, implementing these models is another issue altogether.

Pre multi-core/many-core, OO has thrived in a single core, single machine environment. In this special case, synchronous function calling can replace the asynchronous invocation idealized in the original OO model and the 'stack' can take care of data lifetimes in a simple and intuitive manner:



In the sequential world, mapping an OO model (e.g. parts of a UML design) to a single threaded executable is a largely automatable task and is therefore an ideal candidate for code generation. However mapping a model to a multi-threaded implementation requires additional synchronization and scheduling logic that is assumed, but not prescribed, by typical models; mapping to multiple process implementations, which involves the generation of complex messaging logic, only adds to the problem.

The end result of this is that programmers are required to interpret concurrent OO models by hand, make assumptions about the designer's intentions, and then implement significant amounts of difficult code, versions of which may or may not be in step with corresponding design versions.

The industry's response to this has been to provide engineers with new languages, and/or language extensions/libraries that aim to provide engineers with the highest levels of concurrency abstraction possible. Not surprisingly this works fairly well for the special case of 'regular' concurrency (e.g. the parallelization of for-loops), but fares less well in the more general irregular cases that appear in day to day programming.

The reason that most 'concurrency specialists' would agree with the sentiments expressed in Herb Sutter and James Larus' article, referenced above, becomes apparent with a first foray into the implementation of a physically concurrent object model (required to exploit multi-core).

The principal goal of the Blueprint development environment is to present the developer with OO's highly intuitive view of concurrently executing objects, to explicitly allow them to express their synchronization and scheduling logic with a similarly high level of abstraction, and to do so in a manner that makes no assumptions about the target platform's architecture and/or memory topology. Optimally accreting the application's functionality to one or more multi-core machines is a separate (and orthogonal) activity, which means that unless the application is intrinsically platform-locked the 'same' application code will execute across any platform without modification.

The 'Divide and Conquer' Approach

There's 'more than one way to skin a cat', and developing new languages that explicitly address concurrency is one way forward, but is anathema for developers that have significant investment in legacy (C++, C#, Java) code. The extension of existing languages and/or provision of libraries are alternative approaches, but since they are essentially retrospective, are likely to involve compromise somewhere along the line.

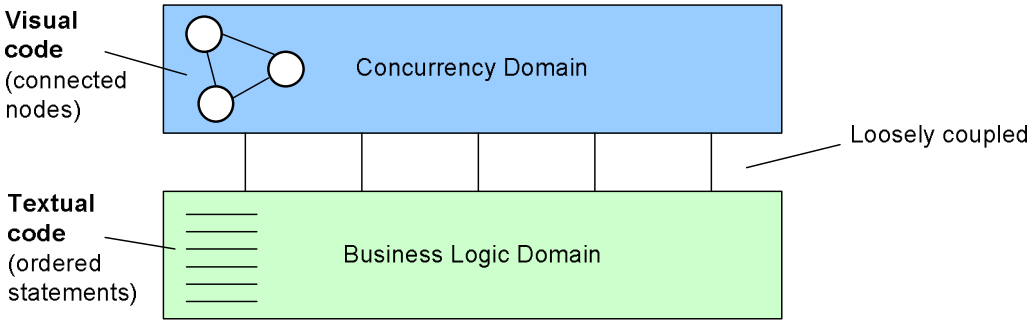
Most would probably agree that algorithmic logic is normally best considered as a set of sequential steps involving conditional logic (e.g. if-then-else). Scheduling logic on the other hand is inherently parallel and is more naturally considered in terms of branching and merging metaphors.

Scheduling and processing are clearly different, and separable concerns; and so an alternative way forward is to allow developers to specifically describe their application's concurrency in terms of it's connectivity and dependency, but leave algorithmic and business logic in it's current sequential form. This means that existing applications can be largely unaffected by migration to multi-core and most developers can continue to work in a familiar sequential environment using familiar tools and languages.

Equally importantly, concurrency needs to be expressed in a manner that does not make any assumptions about the target platform; number of cores, number of machines, memory distribution and so on. This means that programmers need to be presented with a simple and intuitive 'idealized platform'. Mapping functionality to target hardware therefore needs to be another separate stage that should not involve or concern application developers.

Visual Concurrency

Blueprint uses a specialized Visual Programming paradigm to deal with concurrency aspects. This allows descriptions to include branching and merging information in a way that textual equivalents alone do not readily support. In the concurrency domain, statement 'order' is replaced by 'connectivity', but the algorithmic/business domain remains sequential and is decoupled from its scheduling.



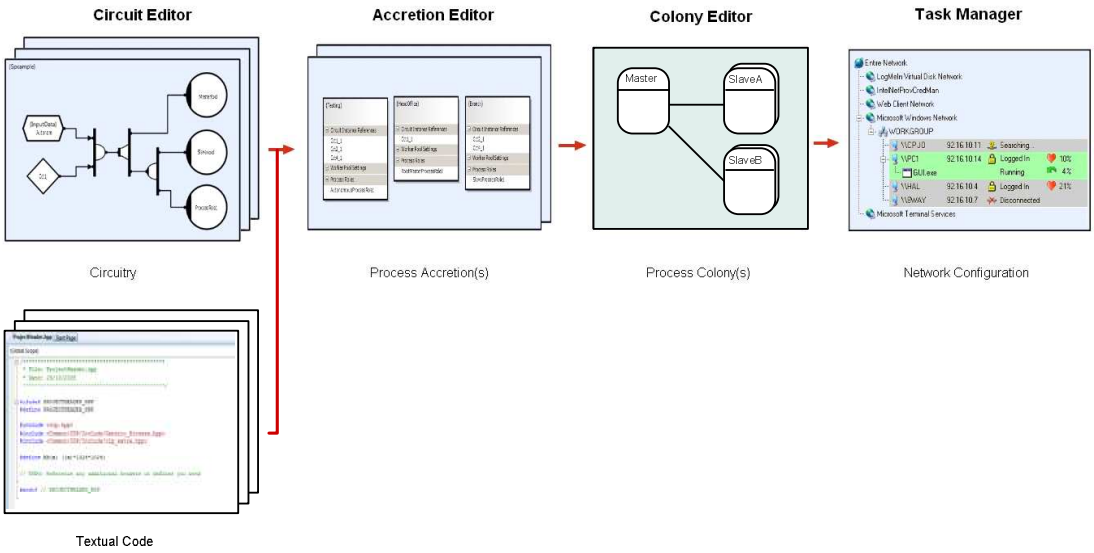
Conventional text programs derive much of their 'meaning' through precise statement ordering, whereas an electronic circuit diagram derives equivalent meaning through its connectivity; this means that the eye can scan circuits in many different orders and still derive exactly the same meaning.

The obvious point here is that connectivity can branch and merge and is therefore an ideal medium for describing concurrency. It is no coincidence therefore that electronic circuitry is usually presented visually, whilst ASIC algorithmic programming (implicitly parallel) is more likely to involve textual descriptions (e.g. VHDL). So arguably, it is the nature of the logic, rather than the nature of the physical hardware, that determines the most intuitive programming approach; and the arrival of multi-core should not be allowed to drastically change the way that developers think.

It is therefore necessary to find a way to map the traditional OO model to code; and to do this it is also necessary to abstract the platform, capture the application's scheduling constraints, and use a new generation of translators to perform the 'heavy lifting' required to take OO's high level concepts, and generate the low level synchronization code that implements it.

The Blueprint Tool-Chain

The first step to providing developers with OO's intuitive and widely accepted concurrent programming abstraction is to create an 'idealized' environment for concurrent applications to execute within; and the second is to provide a 'series' of independent (orthogonal) descriptions that take the high level OO platform independent abstraction, all the way through to the deployment of an arbitrarily runtime-scalable set of executables. The articles that follow will describe each of these steps, and where relevant, reference early adopter projects like the UK's Surface Ship Torpedo Defense (SSTD) system as proof of concept.



Blueprint separates the mapping of high level program logic to physical executables into four independent stages;

The first stage is to develop an application for the idealized Single Virtual Process platform. In most cases it is possible to develop and debug this as a single process on a standard laptop or desktop (specialized I/O devices can be modeled using Blueprint devices). This involves two distinct components; a textual algorithmic/business logic description, and a visual concurrency constraint description.

The second (independent) stage is to use the accretion editor to map program logic to one or more distinct 'processes'.

The third stage is to use the colony editor to identify those processes that are to be 'slaved' (see scale-on-demand). The translator can then build each required process type.

Finally the task manager is used to allocate instances of each process type to appropriate machines in the available network.

The latter three stages are relatively lightweight and do not involve modifying the application itself. There is no limit to the number of accretions, colonies or network configurations that can be applied to a given logical Blueprint application. If the application itself is correctly written (no undetected race conditions) then each mapping will usually execute repeatably (albeit at different speeds), allowing most debugging to be undertaken with a simple single process (and often single threaded) build.

John Gross and Jeremy Orme
john@connectivelogic.co.uk
jeremy@connectivelogic.co.uk

Feature article (Part 2)

Multi-Core OO - A Diagram Is Worth a Thousand Lines of Code

Abstract

The era of multi-core programming has arrived. Many commentators have pointed out that concurrent programming requires us to learn new techniques, different from those adopted for standard sequential programs. In their paper, Software and the concurrency revolution, Microsoft's Herb Sutter and James Larus describe how:

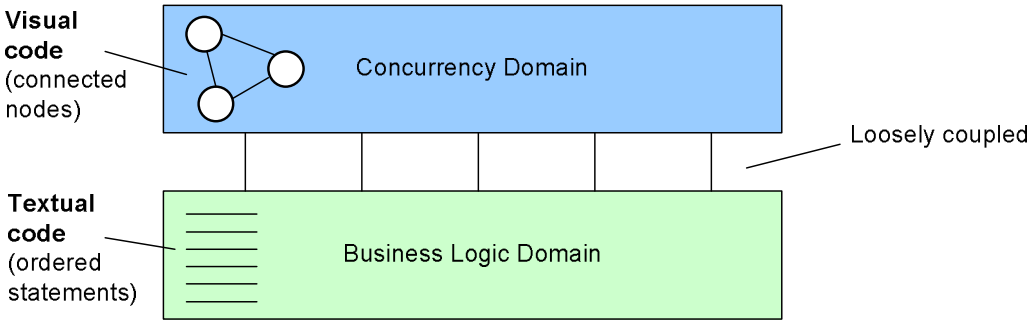
> *"humans are quickly overwhelmed by concurrency and find it much more difficult to reason about concurrent than sequential code".*

However, we know that when playing sports or driving a car, humans demonstrate an innate ability to deal with complex concurrent activities. Why are we so good at managing our concurrent environment but so poor at describing it programmatically?

The Blueprint toolset is built on the premise that we actually have no problem understanding complex concurrency – until we project it into the one-dimensional world of text. For this reason, a visual representation is much easier to work with than its textual equivalent. There are some cases where text is obviously more intuitive – such as parallelizing data-parallel loops – and here technologies such as Microsoft's 'Task Parallel Library' (TPL) and/or Intel's 'Threading Building Blocks' (TBB) can be intuitive and productive.

A Language Just For Concurrency

Blueprint provides a means of separating an application's concurrency logic from its algorithmic/business logic, and uses a specialized Visual Programming paradigm to deal with the concurrency aspects. This allows descriptions to include branching and merging information in a way that textual equivalents alone do not support. In the concurrency domain, statement 'order' is replaced by 'connectivity', but the algorithmic/business domain remains sequential and is decoupled from its scheduling.

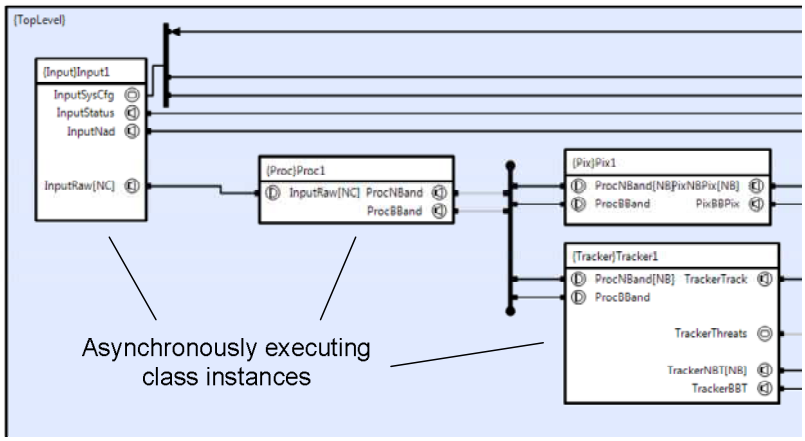


Blueprint presents the application developer with a high level Object Oriented (OO) view of the world meaning it:

- is implicitly concurrent;
- makes no assumptions about target hardware.

This could therefore be a single machine, and/or a heterogeneous network of machines. This means that applications can be built for any process configuration without the need to modify program logic (see Late Accretion below).

Asynchronously executing class instances can be joined together in any way, provided that the prototypes at each end are compatible. Adjacent classes synchronize through their connections (see 'Circuits' below). They can be archived and re-used through a visual drag-drop-and-connect metaphor.



Blueprint has the concept of 'colonies' – a network of compute nodes in which any node can process any task – and this allows additional slave machines to be recruited and/or retired at any stage of application execution without loss of data (Scale-on-Demand).

In addition, the runtime scheduler supports task prioritization, and Blueprint applications will execute preemptively at network scope. This gives the resulting applications an implicit distributed real-time capability.

This article examines the rationale for choosing visual over textual in the concurrency domain and looks at how Blueprint uses its diagrammatic approach to bring additional capabilities to developer's existing multi-core toolboxes.

The Case for Visual Concurrency

The Football Analogy

Anybody who has played football (American, Australian, Rugby or Soccer) knows that you need to try and keep track of each other player on the field, as well as the ball, the referee and yourself; each musician in an orchestra must synchronize with the conductor and each other musician. It would therefore seem that many people are actually very good at dealing with concurrency.

In the frame, there are five players in white and six players in blue. The ball is roughly in the middle of the field. The player closest to the ball is in blue. He has an opportunity to play the ball up-field to one man or back to one of three defenders. He has two opponents closing on him from his left and one in front of him.

Complete Intuitive Picture Subset of Information as Text

However, although most of us can 'see' any number of written sentences at the same time (see above), few of us could 'read' more than one sentence at the same time. Even the most experienced stock market traders would probably be challenged by three or more simultaneous telephone calls; our linguistic skills (textual and verbal) appear to be more or less sequential.

This should not be surprising because linguistic meaning is heavily dependent on word order, and so 'watching' a football match is a very different experience from listening to it on the radio, or reading about it in a paper. Films can usually tell stories in less time than books because apart from anything else, the information bandwidth is higher (but of course that does not necessarily make it a more satisfying aesthetic experience!).

When is Visual Programming Applicable?

Few would disagree that the best way to express an FFT or Matrix inversion algorithm is almost certainly textual; VHDL is now used by electronics engineers for the domains in which it is appropriate. In the same way that the electronics industry now adopts a hybrid visual/textual approach, software engineering can also benefit from the same specializations.

Few people would disagree that GUIs are best developed using graphical drag-and-drop metaphors, class designers are also making increased use of visual semantics, but equally few people would try to use pictures to describe a recursive quick-sort algorithm. TPL, TBB and Blueprint are clearly not mutually exclusive.

Conventional text programs derive much of their 'meaning' through their statement ordering, whereas an electronic circuit diagram derives equivalent meaning through its connectivity (which can branch and merge). In the specific cases where statement order doesn't matter, then there is typically potential for concurrent execution.

Most text languages are primarily concerned with algorithmic logic (usually branching on particular data values) and don't have an intuitive way of expressing concurrency in general, particularly the more complex irregular dependencies like "f3 can be executed when f1 and f2 complete, but f1 can't be executed while f0 is executing".

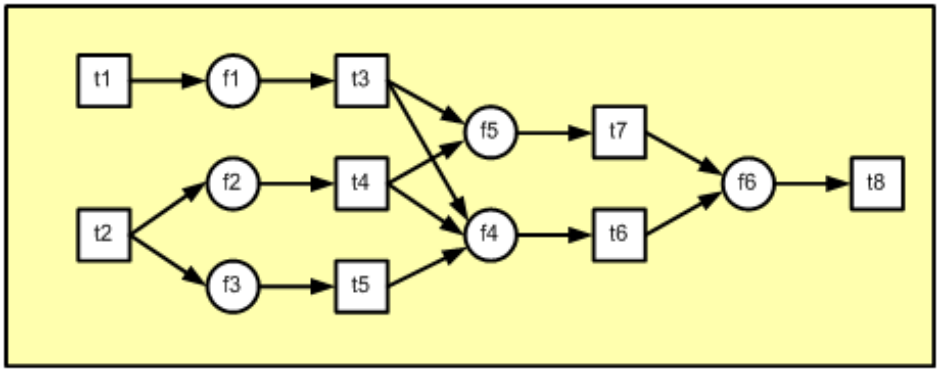
Whilst Parallel-For, Map-Reduce and other existing mechanisms can address 'regular' parallelism in an effective 'bottom-up' manner, the 'futures' approach to irregular problems that involve more than half a dozen concurrent threads of execution can be difficult to conjure with.

A Simple Example

Consider the pseudo-C example function below;

```
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;
}
```

If we analyze the code in the function above and assume for now that these functions have no relevant side-effects, then we can produce an informal dependency diagram such as the one shown below.



In the diagram above

- 'boxes' represent data objects
- circles represent sequential user functions
- arrows represent data flow

In this informal example,

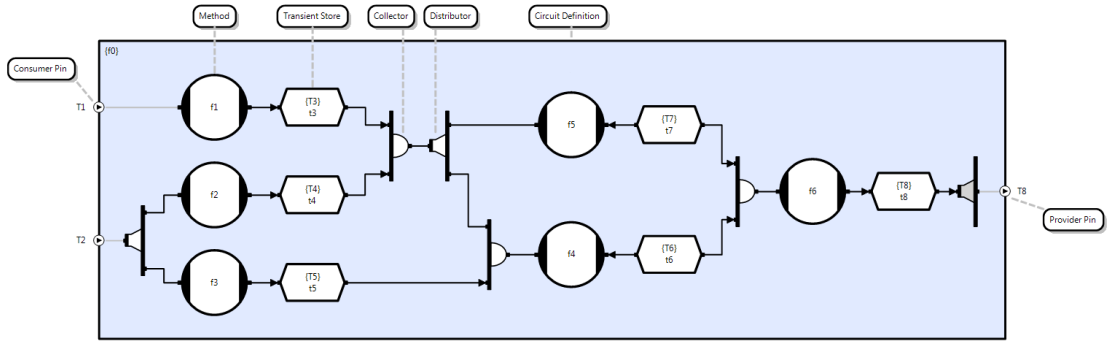
- diverging arrows represent 'distribution'
- converging arrows represent 'collection'

Functions become executable when all of their inputs are collected. So in this example, the f4 function cannot execute until f1, f2 and f3 have all executed, but the order in which they complete doesn't concern f4. Anecdotally at least, most people would appear to be able to appreciate the logical dependency, and hence logical concurrency from this diagram, more rapidly and readily than they could from the sequential text that it represents (see above).

Separation of Concerns

More importantly, information concerning the data objects, and the operations performed by the functions f1…f6, is not required in order to do this; likewise, the data objects and functions that operate on them are not concerned with scheduling. This highlights the fact that at a high enough level of abstraction algorithmic logic and scheduling logic are separable; and in this instance, one is textual and the other visual.

The concurrency description is localized rather than being dispersed across the application, and is uncluttered by algorithmic information that is not relevant to concurrency. When we combine the two descriptions we have a concurrent description that makes no assumptions about its target hardware (this will be addressed later). The figure below shows a formal, machine translatable, Blueprint implementation of the informal dependency diagram above.



Amongst other things, the symbolism disambiguates branching and merging. Branching typically infers ‘sharing’ but could equally well mean ‘competing’; merging may infer ‘collection’ (wait for all), but could also mean ‘multiplexing’ (wait for any). It also addresses issues like destructive/non-destructive reads and so on. The diagram above illustrates how the symbolism uses ‘event operators’ (e.g. collectors and distributors) to precisely describe the required execution constraints.

The translator compiles the diagrams into runtime calls, in the same way that high level language compilers generate assembler/object code, but since developer code and generated code are separated, the details of the generated code are not relevant to the developer. Because locks and other synchronization mechanisms are machine managed, their granularity is not limited by what humans can conjure with; the generated code is therefore able to lock at a very fine granularity and minimize effects due to Amdahl’s law.

Conclusion

In summary, what Blueprint’s visual approach aims to do, is to separate an application’s scheduling logic from its algorithmic/business logic. It replaces 'order' with 'connectivity' and thus exposes concurrency in a way that avoids 'reading'. In particular, it allows for branching and merging operations to be unambiguously specified, which in turn allows the scheduling logic to be decomposed into simultaneously visible, but independently analyzable, strands of execution. The order with which your eyes scan a Blueprint circuit is only important if you are trying to extract some particular information such as data-flow or event-flow; the parallelism and dependencies can be seen at a glance (the branching and merging is obvious).

John Gross and Jeremy Orme
john@connectivelogic.co.uk
jeremy@connectivelogic.co.uk

Feature article (Part 3)

Multi-Core OO – Gaining Freedom from the Platform

Abstract

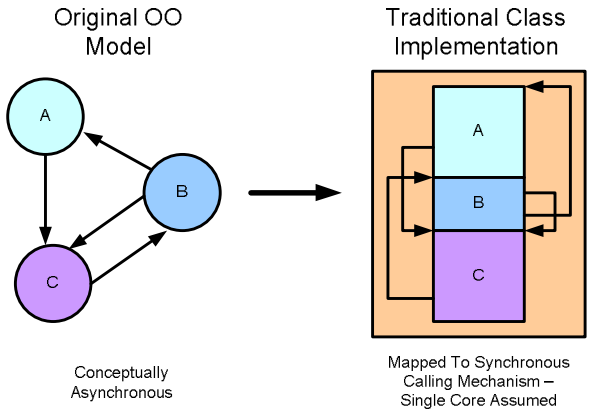
Conceptually at least, standard Object-Oriented models allow objects to invoke adjacent objects by sending messages to their member functions; and so at its highest level of abstraction OO has an inherently concurrent view of the world that is divorced from its eventual mapping to processes and hence CPU cores.

Unfortunately the potential parallelism of OO's model is difficult to realize from this high level view alone because there is insufficient information to precisely describe the communication and synchronization which is required to ensure correct scheduling. Applications that do assume particular memory models (distributed or shared), network topologies, core counts, operating systems, and other such details will almost inevitably be tied to them and so all of these issues have to be abstracted out.

This part of the article considers an 'infrastructural' class equivalent that rectifies these problems; transparently distributing its processing load across the available hardware and communicating asynchronously with adjacent infrastructural classes through strongly prototyped connections.

Freedom from the Platform

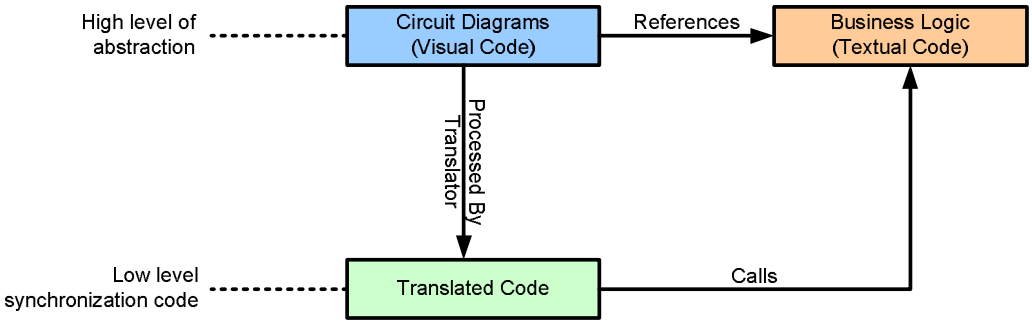
Traditionally, OO has thrived where it has been possible to assume a single core, single machine environment. In such an environment, synchronous function calling can replace the asynchronous invocation idealized in the original OO model and the stack takes care of data lifetimes:



However, the advent of multi-core has led programmers to demand OO to work in an environment where synchronous invocation is no longer viable because it would stall threads and waste time and/or resources.

Blueprint was designed to allow developers to work at the highest levels of OO abstraction and be able to make optimal use of multi-core hardware without needing to consider low level details like threading, message-passing and arbitration (semaphores and locks).

In order to achieve this, it has to enable developers to describe synchronization and communication at a very high level of abstraction using a visual metaphor. The 'heavy lifting' required to implement the lower level synchronization etc, is performed by the diagram translator, which can generate C++ and/or C# as required.

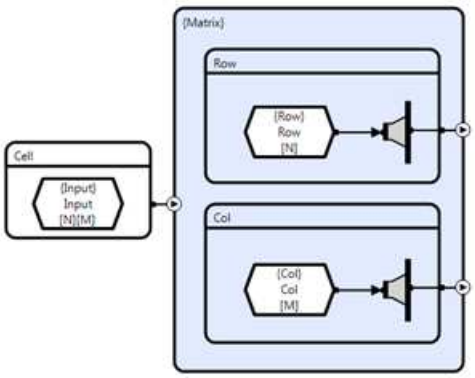


This means that the notion of a 'class' needs to be extended to include the additional information required by the translator. Blueprint's concurrent equivalents of classes are referred to as circuits.

What Is A Circuit?

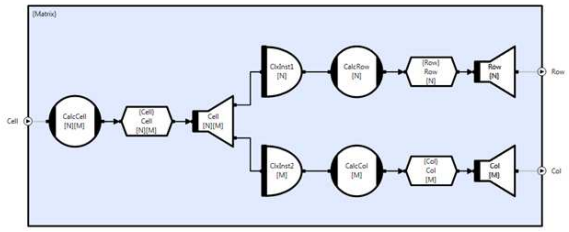
Most developers will be familiar with the concept of a 'class' in an OO sense, and the intention of Blueprint circuitry is to provide a concurrent (infrastructural) equivalent. Their purpose is therefore to localize, encapsulate, archive and re-use program logic (algorithmic and infrastructural).

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.




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

   Row[] outputRow();

   Col[] outputCol();

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

Circuit Prototype (Declaration)



Circuit Body (Definition)

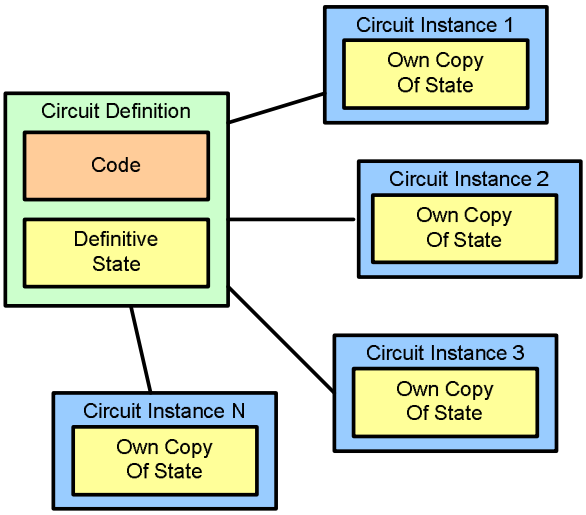
```
// Matrix class definition
void Matrix::inputCell(
   Input in[N][M]) {
   // Read cell data and begin
   // processing
}

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 other circuit definitions.

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.



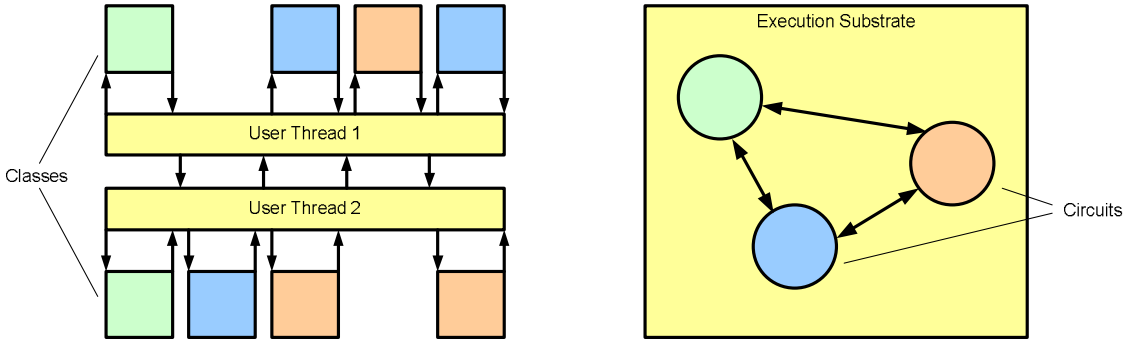
Circuit instances can be joined together in any way, provided that the prototypes at each end are compatible. This means that the complex array output of an FFT for example, can be directly connected to the input of a complex array filter, an RSS feed adaptor component could be connected to any number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

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

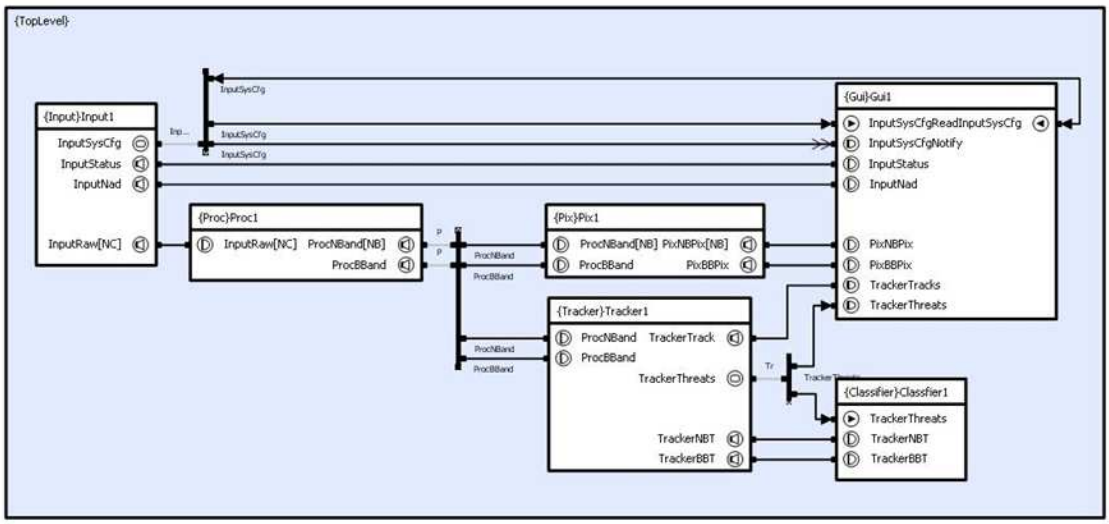
Circuit member functions (methods) can be multi-dimensional, and each element can own its own individual state; this provides implicit concurrency and in many cases replaces constructs like Parallel-For (see example above). It is also possible to specify the number of times that member function elements 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).

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).



Conclusion

The Object-Oriented model is conceptually asynchronous in that its actors can invoke each other's methods concurrently. In single core systems, the OO model can be mapped to classes and synchronous invocation, but this approach does not work for multi-core and distributed systems because of the limitations of the stack based call-and-return mechanism. Applications using synchronous invocation are doomed to spend the rest of their life executing in a single thread or suffer huge upheaval on each platform change as the code is modified to break the application into chunks appropriate for that platform.

Blueprint provides an alternative to a regular class, called a circuit. Circuits have many of the desirable properties of classes that allow code to be encapsulated and re-used but they are self-propelled rather than relying on thread(s) to drive their execution and they use connections instead of calling to invoke each other's operations. This means that complex applications can be built from composing any number of circuit instances, as required, and the computation will be automatically distributed across the available hardware with an even load-balance.

John Gross and Jeremy Orme
john@connectivelogic.co.uk
jeremy@connectivelogic.co.uk

Feature article (Part 4)

Multi-Core OO – Beyond the Single Process

Abstract

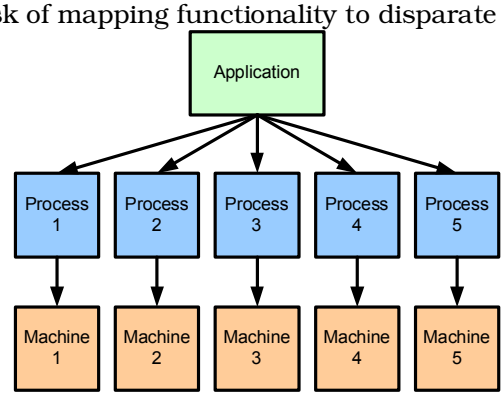
The parts two and three of this series proposed a means of maintaining an OO paradigm in a multi-core environment. This addressed the problem of mapping concurrently executing components to an arbitrary number of CPU cores in a symmetric shared memory environment. However, it did not deal with the more difficult issue of mapping concurrent functionality to multiple processes – each with their own disparate memory space and each running their own operating system instance.

This article will explain why next generation "many-core" processors are likely to make this problem relevant to mainstream developers, what challenges it raises and how it can be addressed with the OO model still in-tact.

In addition, it will show how multiple mappings of functionality to processes can be achieved without the need to modify the application itself (as described in parts two and three of this series). Accretion of functionality to processes is a lightweight task, and this means that applications can be developed, debugged and maintained in a convenient 'single process' form, but deployed in the field with the benefit of scalable processing resources.

Why is Multiple-Process Programming Important?

Multiple-Process programming has always been relevant for distributed applications because a cluster of machines are not able to execute a single process so developers have always had the task of mapping functionality to disparate processes.



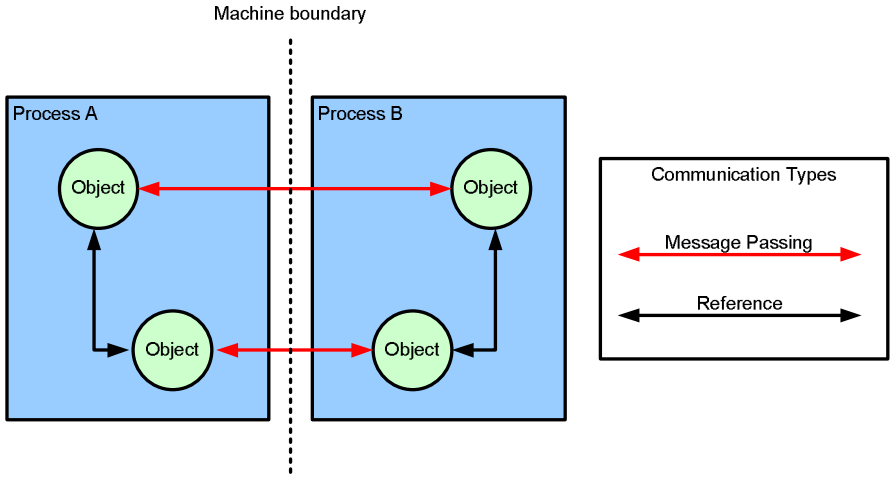
Most engineers would agree that because of this, distributed applications are more difficult to implement than their single process equivalents but up until now (and despite the arrival of multi-core) mainstream developers haven't needed to consider this particular problem.

This situation is almost certain to change in the very near future because the prevailing view is that symmetric memory architectures are unlikely to scale past 8 CPUs. Amongst others, video games developers migrating to the IBM CELL BE have already taken the plunge, and the Tilera64 software environment also predicts 'message-passing' as a 'next-generation' programming model.

What Challenges Does this Raise?

As discussed earlier, at its highest level of abstraction the OO paradigm does not make any assumptions about the target platform. Whilst it is relatively straight forward to abstract operating system calls and even core-counts, it is more difficult to abstract process topology. Efficiently re-mapping an application in this way typically involves considerable low-level re-writes. Why is this?

Process topology is implicitly assumed by the choice of each component's communication paradigm. If two objects find themselves co-located in the same process, then calling each other's methods directly, or using shared memory arbitration and reference is the simplest and most efficient means of exchanging information. If however, objects find themselves in adjacent processes that do not share memory, then data needs to be moved between them using a message-passing paradigm; typically using TCP/IP sockets or some other equivalent.



To avoid becoming locked into a particular topology, objects must use the same API regardless of whether they are co-located in the same process or separated across a process boundary.

There are many parallel APIs that assume shared memory, including:

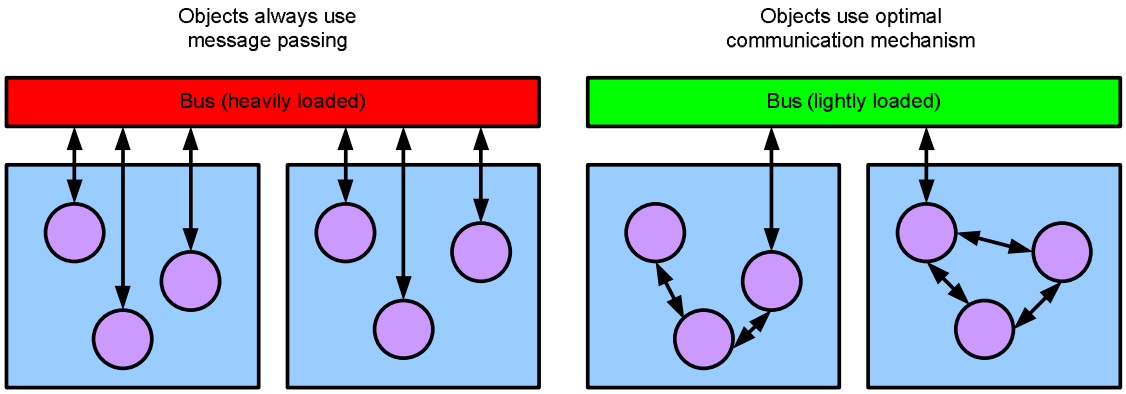
- Intel's Threading Building Blocks;
- Microsoft's Task Parallel Library;
- Intel's OpenMP

There are also many parallel APIs that explicitly assume distributed memory and therefore move data, including:

- Parallel Virtual Machine (PVM);
- Intel's Message Passing Interface (MPI)

The fact that parallel application code usually assumes particular memory architecture arguably limits component re-use and portability in an even more fundamental way than choice of operating system.

A compromise solution is to 'always' use a message passing API (even if the components are in the same memory space). This means that topology doesn't matter quite so much because it will work in all cases. Unfortunately this imposes a performance penalty as co-located objects unnecessarily trawl large volumes of data across the bus and this can outweigh any gains made from using multi-core technology.



Another problem is that load balance can be difficult to calculate statically and is prone to change with the modification and/or addition of functionality. The fact that core counts are expected to double every 18 months for some time to come means that different customers are likely to have different platforms, comprised of machines with differing core counts.

In the worst case, 'turn-key' applications that assume a particular functional accretion may require different versions for different platforms. This problem is compounded by the fact that load-balance frequently depends on the data-set that is being processed and so cannot necessarily be discovered until runtime (see part 5; 'Scale-on-Demand').

Perhaps the most difficult problem however, is providing a means of 'describing' the application's accretion(s); and to do so in a way that doesn't impact on the application source as outlined in parts 2 and 3 of this article.

- Accretions must be independent of each other;
- Translation of the accretion description(s) along with the application source to produce a final executable for each identified process must be automatable;
- Accretion must be a 'cheap' operation, so that it is feasible to 'experiment';
- Accretion must not require any knowledge of the application other than its approximate CPU and bandwidth budgets.

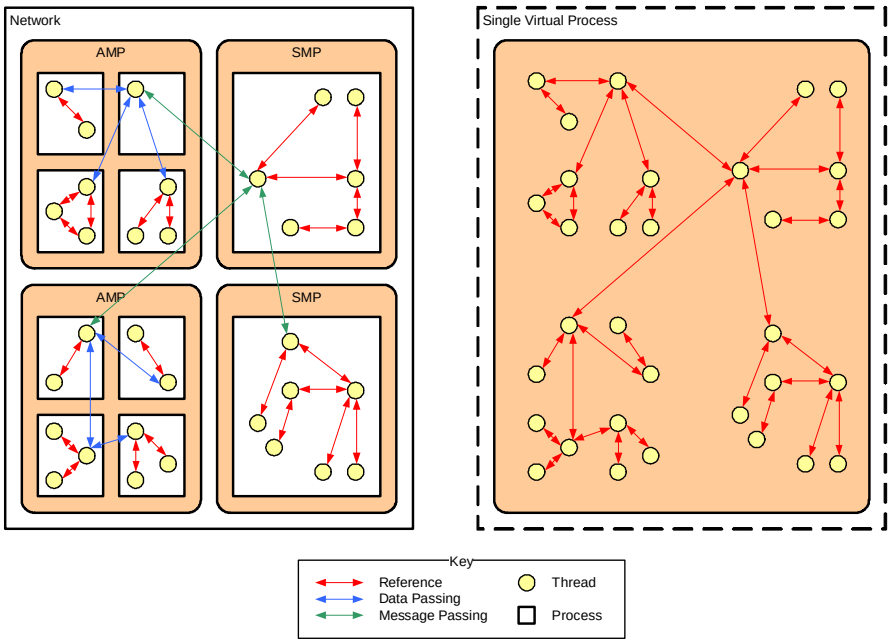
How are these Issues Addressed?

In order to provide an efficient solution that doesn't gratuitously move data, the Blueprint programmer is presented with a beefed-up 'reference' view of the world. This is referred to as the 'Single Virtual Process' (SVP). The developer sees all data by reference, but if two components actually find themselves located in adjacent processes at execution time, then the runtime will:

- Transparently move the referenced data

- Cache it locally
- Garbage-collect it when it is no longer referenced.

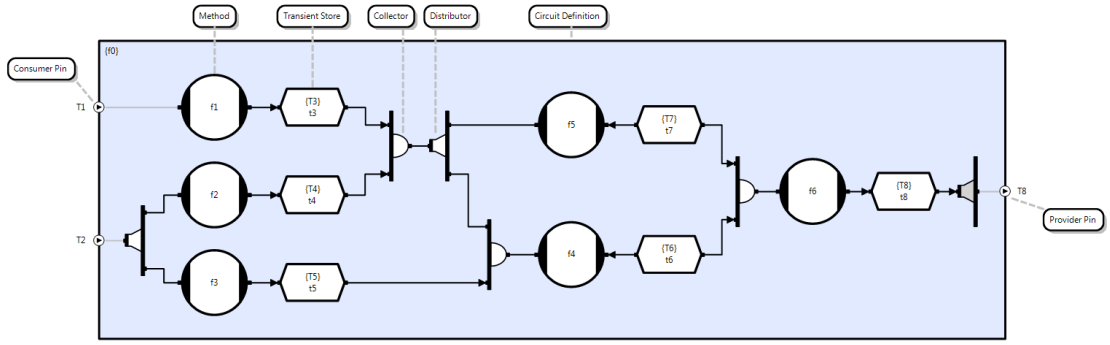
The developer also assumes an unbounded number of 'logical' preemptive threads, but this is actually implemented using a minimal number of system managed worker threads to minimize resource requirements.



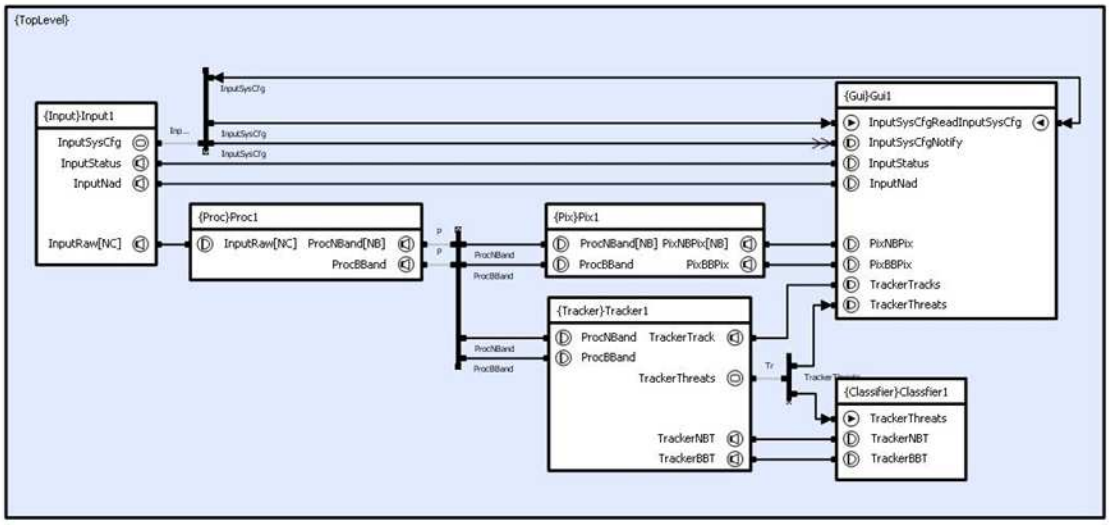
To the programmer, the SVP model appears like an SMP programming model operating at network scope as the data always appears to be accessible by reference. The runtime must keep track of the size of each referenced object so that it can transparently move data between disparate memory spaces and sustain this 'illusion' of direct reference. This topic is beyond the scope of this introductory article but is achieved through the use of 'records' which are data wrappers that allow for variable sized data.

At the top level, the application needs the concept of a set of asynchronously executing autonomous components. The asynchronous nature of these components is crucial because unless they are all co-located to the same process, they will need to execute asynchronously in the true sense (probably on different machines). In order for these components to execute asynchronously, global (inter-process) synchronization is required, but it has to be provided at a high enough level of abstraction to avoid making any assumptions about process and/or thread proximity.

Blueprint addresses the first issue through the concept of 'circuits' (concurrently executing classes), and the second through its use of high level 'event operators' (collectors, distributors, multiplexers etc); these mechanisms are described in earlier parts of this article.



The diagram above shows a typical 'circuit' definition. In this example, collectors and distributors are used to coordinate and synchronize concurrent method execution within the circuit, and the public objects (exposed by their consumer and provider pins) provide synchronization between adjacent circuit instances.



The circuit above shows the top level of a simplified military sonar system. Each 'sub-circuit' executes asynchronously and synchronizes with each other 'sub-circuit' through its public event operators.

Objects ('connectors') must be able to locate and connect-to other objects (their 'connectees') and this is achieved using the runtime's registration service. In order for the translator to be able to generate the necessary code to do this, objects (and their exposed 'pins') need to be uniquely named so that the connectivity information provided by the circuitry can be used.

Conclusion

The days of the physical SMP architecture are numbered due to memory access becoming a major bottleneck as core counts rise. Heterogeneous architectures demand different methods of communication between objects depending on whether they share a common address space (where data can be referenced), or are in separate address spaces (where data must be transferred between them).

To avoid becoming locked into a particular topology it is essential that the same API is used for all communication between objects. This allows the underlying runtime to

select the best location to execute the object and the most efficient means of communication to use.

This also means that because the application code makes no assumptions about topology, functionality can be partitioned between processes easily using a lightweight accretion operation. The application code doesn't change and the accretion description simply specifies which objects map to a particular process.

John Gross and Jeremy Orme
john@connectivelogic.co.uk
jeremy@connectivelogic.co.uk

Feature article (Part 5)

Multi-Core OO – Scale on Demand

Abstract

In this series we have considered how to develop an application to make best use of parallel hardware consisting of perhaps an unknown number of processors in an unknown topology (this information will be 'automatically' discovered at runtime). We have already moved into a world where we can't assume the processor count and it is also likely that 'unknown topology' will also soon become an issue. The process for insulating applications from these uncertainties while retaining the OO abstraction is as follows:

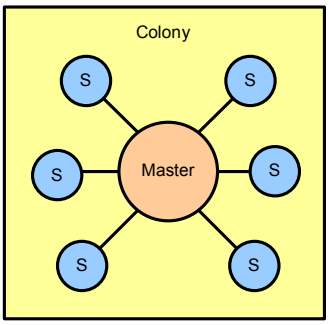
- Separate infrastructure from processing and describe it using an intuitive visual language
- Use an asynchronous class equivalent to encapsulate and re-use concurrent infrastructural logic
- Use the lightweight process of accretion to map the application to multiple processes for deployment

This article considers the next challenge – dynamic reallocation of hardware at runtime. It will expand on the SVP abstraction and introduce the concept of 'process colonies' which provide applications with a real-time scale-on–demand capability. It also provides a high-level overview of the distributed scheduler which is responsible for preemptively balancing the load across participating machines and managing their dynamic recruitment and retirement.

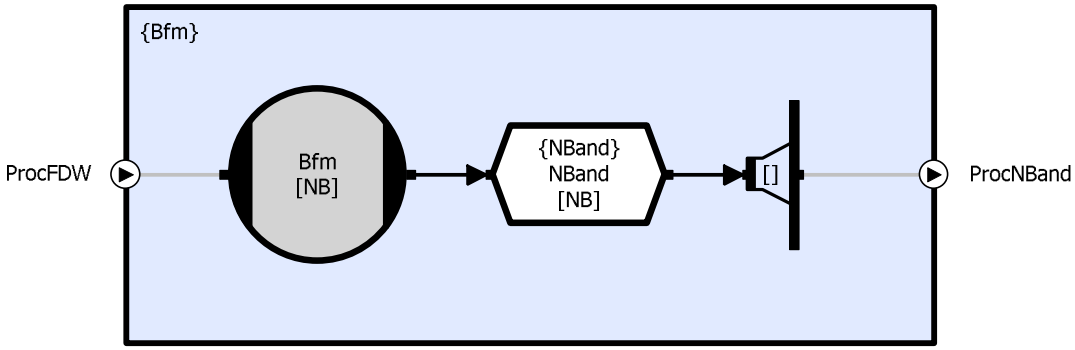
What is Scale-on-Demand Processing?

This abstraction augments the concept of a simple 'process' with an equivalent 'colony' of collaborating processes. Colonization, like Accretion, is consistent with the SVP model of the world, and so application code is unaffected by this later-stage mapping.

Process Colonies



The figure above shows a simple colony containing a single master process and an arbitrary number of slave processes (the general case can support applications with any number of 'slave-able' processes). The translator uses this information to create two versions of the specified process (see Accretion in part 4). The 'master' version will instantiate all of the specified definitive circuitry, whilst its 'slave' equivalent will only instantiate the functionality required to execute the circuitry that has been designated as 'slave-able' (able to be offloaded to slave processes).



Designating a method as slave-able only involves setting one attribute; the method is subsequently displayed with a shaded interior, and non-colonized accretions and builds will ignore the setting. Colonization is completely transparent to the application developer; some of its functionality is provided by generated code, and the rest is provided by the runtime scheduler.

At runtime, each master process in the colony registers with the 'registrar process', which can be hosted by the application instance's 'root-master' process, or a dedicated network server process (with a fixed address). Any number of uniquely named application instances can execute on the same network (the instance name is passed to each process at runtime), but in practice, turn-key systems typically need predictable performance and so the multiple instance capability is primarily provided to allow engineers to share resources for development purposes. This means that slave processes can be deployed on any available machine in the network and will automatically locate and connect-to their designated master instance.

The colony editor allows developers to specify each master/slave colony's resource requirements. In some cases a given accreted process may wish to wait for a specific number, or minimum number of slaves to connect (typical for real-time applications), or in other cases, the application may wish to start execution with whatever is currently available (typical for off-line applications).

Colony Scheduling

The runtime's distributed scheduler transparently manages a number of aspects of application execution;

Slaves can be recruited and/or retired at any stage of execution, and the runtime will automatically ensure that slaves do not 'exit' until all of their allocated tasks are complete, and any definitive data has been redistributed to adjacent slaves (or the master itself).

The scheduler will dynamically distribute the load across each participating slave. In order to support distributed real-time applications, the scheduler addresses prioritization and preemption first, load balance second, and bandwidth optimization third.

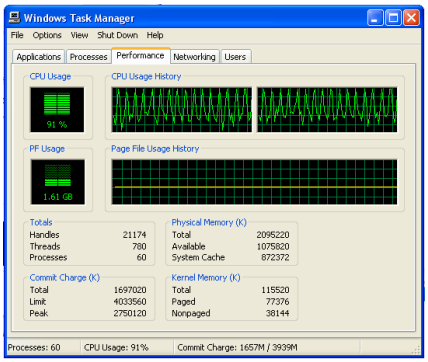
Network scope preemption is very difficult to implement in a topology independent manner, but is actually essential for applications like military sonar, and/or oil and gas survey systems, financial modeling can also be extremely time critical, because the ability to prioritize tasks at network scope can save vital seconds.

Consider a signal processing system that needs to perform a determined number of FFTs at 1Hz, and assume that the network turn around time for this is 0.5 of a second. Now assume that in parallel with this, it also performs another set of smaller FFTs at 20 Hz, and that the turnaround time for this is .02 of a second.

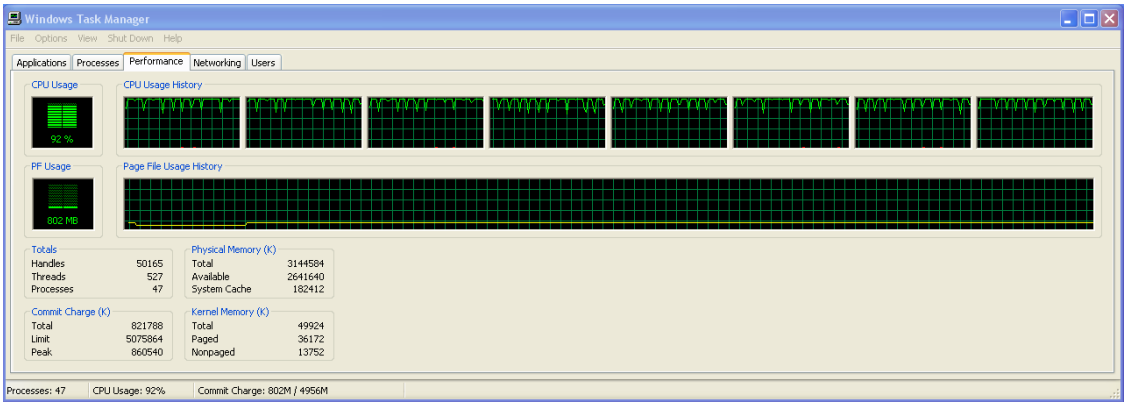
Most standard mechanisms like scatter/gather do not execute preemptively, and so for at least 0.5 of a second, the higher frequency calculations would be stalled. This could be addressed by executing two processes on the same machine, but context switching a large process at 20Hz would not have been practical for most of the Blueprint real-time systems undertaken to date. This then leaves little choice but to map particular processes to particular machines/cores and thus lose the obvious benefit of optimal turnaround time for high priority processes (where every CPU/core in the system is recruited).

In Blueprint applications network scope preemption is transparently achieved by keeping track of each slave's prioritized job list. The master scheduler will never send a priority 'N' job to a target that is fully loaded at priority 'N' (or above), if there is another target that has spare capacity at that priority. This is essentially a distributed equivalent of SMP thread scheduling, and each slave scheduler is responsible for ensuring that it always has enough worker threads to provide preemption for each of its CPU cores.

The task manager screenshots below show a sonar demonstrator with its master process running on a dual core laptop, and a single slave process executing on an 8-way desktop. There is no limit to the number of slave processes that could be recruited if required.



Master

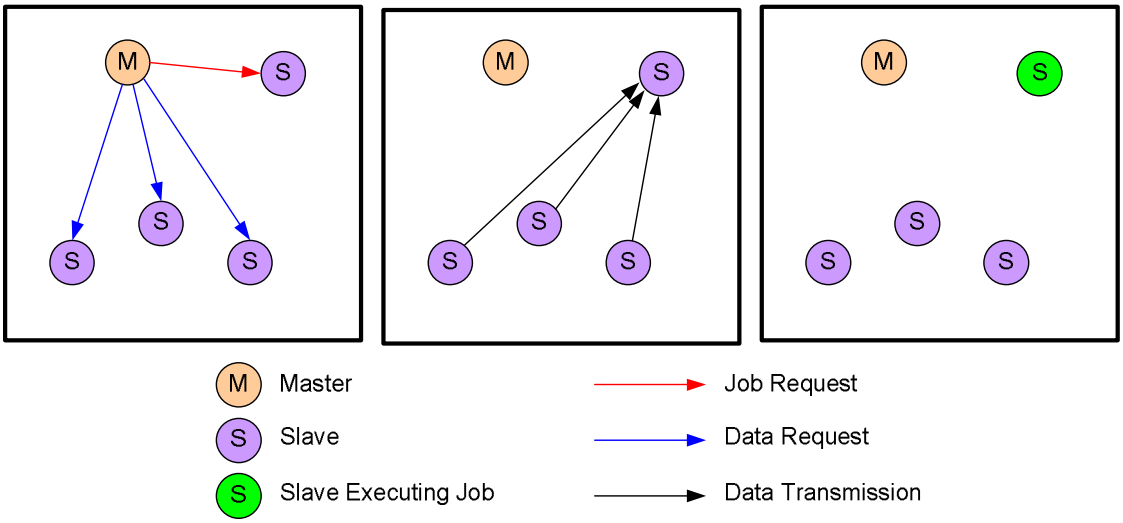


Slave

The runtime will automatically balance the load (within the constraints imposed by preemption), and the more and/or faster cores that a machine provides, the more work it will be given; this allows applications to utilize resources in a whatever/whenever (scale-on-demand) sense, rather than depending on dedicated clusters. If a slave machine becomes busy/not-so-busy at any point (e.g. its virus checker is running/finished), the master scheduler will detect this and automatically adjust the slave machine's workload.

In order to minimize communication bandwidth (within the above constraints), the scheduler adopts a 'jobs-to-data', rather than 'data-to-jobs' strategy. For a job to be executed by a given slave process, the latter must have a copy of each of the job's input arguments, as well as an up to date definitive copy of the job's state (if required). All else being equal, the scheduler will therefore allocate the job to whichever slave owns (or has cached) the most job-specific input/state data.

This is a three stage process. If required, the first stage involves simultaneously selecting and instructing adjacent slaves to send copies of any outstanding data that are required by the target slave. The second stage, involves the nominated slaves sending their data to the target slave. Finally, when all data (and state) has arrived, the target slave will schedule the job for execution and cache/flush its outputs appropriately. Cached data that is no longer referenced is automatically garbage collected.



Scheduling latency is minimized by 'job-overlapping'; the scheduler looks ahead, predicts load, and allocates a specified number of 'buffer-able' jobs to each slave; this allows processing and communication to be fully overlapped and in typical cases, hides the latency associated with the movement of data.

What is its Application?

The scale-on-demand capability provides a generic mechanism for migrating applications between platforms; the core-count, clock-speed, machine-count, communication protocol, and heterogeneity of target networks are all abstracted from the developer. This means that resource calculations are less critical because 'getting-it-wrong' doesn't require a re-work; it just means recruiting more or fewer resources.

As an illustration of this point; the UK Navy has had so many problems with 'so-called' scalable applications over the years, that they typically insist that the CPU load of a delivered system must not exceed 50% (to allow for expansion etc); this doubles hardware costs, heat emissions, footprint, power consumption, weight, single points of failure, spares support and so on. Even with this precaution in place, a 50% load is just as likely to indicate poor load balance as available capacity, and this may not always be apparent until the contingency is actually required. Because scale-on-demand was demonstrable, this requirement was waived for their Surface Ship Torpedo Defense system (see CLiP case studies), and as a result, the 50% requirement was replaced by the less onerous requirement that the delivered rack must have the appropriate spare expansion slots (should they be required).

In most cases scale-on-demand technology also means that applications can be developed, debugged, and maintained in a more convenient single-process configuration and then deployed across an appropriate distributed resource. Choice and acquisition of target hardware can also be deferred until a later stage in the project life-cycle.

The technology also has specific application in a number of sectors;

Distributed Real-Time Applications (Military, Oil and Gas, Financial Modeling)

Many of the early adopter projects to date have been distributed real-time applications (see case studies); in these cases the translator and runtime settings can be optimized for timeliness. These have included surveillance systems, distributed interactive

simulations, and a number of military and commercial sonar with both acoustic and seismic processing capability.

In the real-time case, scale-on-demand capability can be used to optimize for fidelity (simulation and military sonar); adding processing power doesn't necessarily 'speed-up' execution, instead it is utilized to improve the definition of the calculation.

In day-to-day operation, military sonar systems can be required to replay and analyze previously logged data, occasionally run crew training sessions, and at the same time monitor for potential threats. If a threat is detected, processing resources can be automatically withdrawn from the replay and training tasks and then rapidly re-allocated to the more urgent task of tracking and classifying the identified threat(s). This is analogous to the biological response to 'extreme cold' where an individual's resources are necessarily focused on the essential processes that are required for survival, at the expense of protecting an individual's extremities. When a successful Antarctic explorer returns to a sufficiently benign environment, resources are automatically returned to normal without conscious effort.

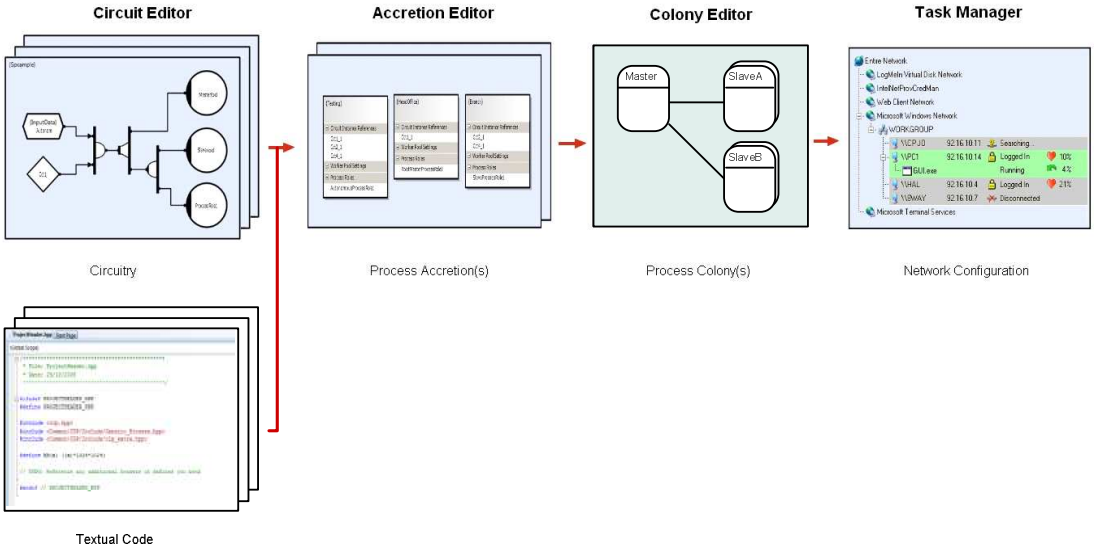
Distributed Offline Applications (Military, Oil and Gas, Financial, Scientific)

In most cases however, additional CPU is used to reduce task turn around time. This would be the case for some surveillance processing, financial modeling, and post-processing of survey data. Because resources can be dynamically recruited, equipment that is only in use during the day can be pressed into service out of office hours.

Scale-on-demand is also useful for systems that have a periodic (typically 24-hour) demand cycle; the processing that a bank requires for example. CPU resources can be added, removed, or just re-focused, in response to dynamically changing requirements.

Deployment

Having implemented, accreted, colonized and built an application; the end result is a set of process executables. The final tool in the Blueprint chain (the Task Manager) provides a convenient means of deploying executables across the available network. Although primarily designed to simplify the management and administration of Blueprint applications, the task manager can be used to configure, deploy and monitor any parallel application.



The tool provides a number of facilities that enable administrators and developers to describe the physical network, create and configure logical networks from the physical hardware, and then deploy application instances across one or more of the identified logical networks.

The Parallel Deployment Language (PDL) will allow the task manager to take programmatic control of the application's deployment. As well as making sure that particular processes are 'pinned' to particular machines when necessary (usually due to some device or peripheral dependency), PDL will also allocate 'floating' processes to available machines, and do so in a manner that ensures that real-time processes do not come into conflict. PDL can also be used to take appropriate action in the event of unscheduled process exits; typical action would be to kill and redeploy particular associated processes.

Conclusion

The process of taking a set of executable processes and describing how they should execute across a network is referred to as colonization. Colonies consist of master processes and slave processes whereby the master orchestrates the behavior of the colony and the slaves perform the bulk of the processing work.

The runtime scheduler employed by Blueprint makes decisions about the most efficient place to execute given tasks within a colony depending on the distribution of data and whether timeliness is most important (real-time systems) or the highest throughput is desired (off-line systems).

In order to control the starting and stopping of processes across the network, a task manager application is employed. This provides a means, through the Parallel Deployment Language (PDL) for the network administrator to ensure that appropriate actions are taken to ensure sufficient processing is allocated each application and to generate events in case of under-performance or failure.

Further Reading

For more detailed information see;
http://www.connectivelogic.co.uk/devzone.asp

For a list of some of the applications to date see;
http://www.connectivelogic.co.uk/solutions.asp

For a comparison between object-oriented and component-oriented paradigms see;
http://en.wikipedia.org/wiki/Component_(software)#Software_component