Concurrent Evaluation of a Directed Acyclic Graph

Rakesh Joshi, EigenSystems LLC
rakesh@eigensystems.com
(917) 312-4188

Abstract

Many performance critical applications in capital markets and computational finance contain significant latent parallelism.

Securities valuation models employ numerical methods that can benefit greatly from concurrency. In practice their migration to parallel code faces the inertia of two decades of investment in refinement, testing and validation.

A more easily exploitable opportunity exists in high-level program structure, using a refactoring approach. A useful class of problems can be expressed as dependency graphs that are continuously evaluated while "immersed" in a real time data stream that updates values of input nodes. Risk computation, market making and electronic trading applications are common examples of this structure.

In the course of migrating risk and trading applications to multi-core, we have encountered this recurring pattern. Our aim is to segregate synchronization and contention concerns into pattern elements to ease the refactoring process.

Intent

Many applications can be formulated as dependency graphs in which successor nodes are evaluated in response to changes in predecessor node states. The analogy of a spreadsheet containing cells with values and formulae can be applied to a useful class of problems.

A shared memory data flow representation of such problems makes inherent concurrency explicit. With increasing numbers of processor cores becoming available in shared memory architectures, significant efficiencies can be achieved over distributed approaches.

Our goal is to encapsulate the scheduling and synchronization mechanisms associated with graph evaluation, to ease migration of serial code to multi-core parallel.

Context

Many applications in high frequency finance can be formulated as a graph, where dependant nodes, (derivative prices, trading signals, risk measures), are evaluated in real time response to changes in input nodes such as stock, commodity prices, interest rates, credit spreads etc. In general these graphs are large; the table below provides indicative measures of scale.

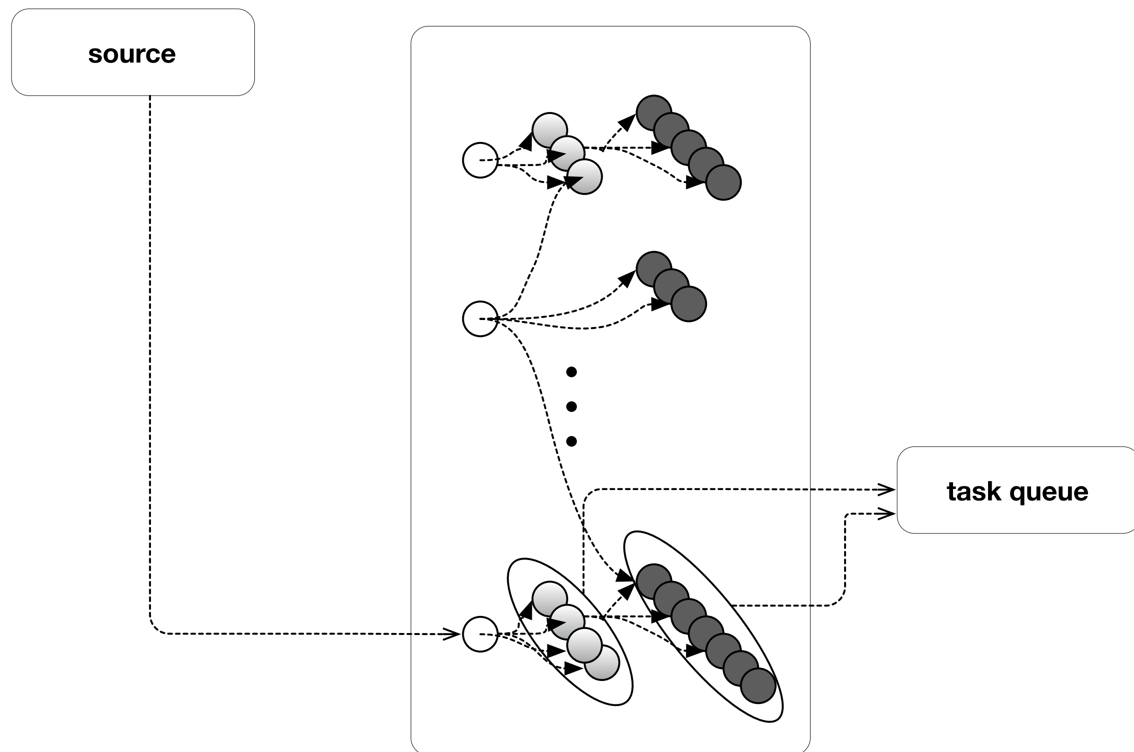
	input nodes	in-degree	out-degree	total outputs
equity options	10^2 - 10^4	2-5	10^1-10^2	10^4
fixed income	10^1	2-3	10^4	10^5
credit derivatives	10^3	2-3	10^2	10^5

Complexity arises due to asynchronous, randomly timed updates of the input nodes and uneven computation workload associated with different dependent nodes.

In a typical application, a stream of data is delivered into the system – input nodes of a graph are identified and updated, and computations are scheduled to update the state of the dependent nodes.

Fig 1. shows a typical computation arranged as a graph. Update methods associated with terminal nodes dispatch computed results to a sink, usually a database or a communication endpoint.

Fig. 1



Indicative computation costs for various kinds of nodes in the graph are shown in the table below; based on common option pricing models.

	typical duration
taylor approximation	~10 μs
closed form models	~10-50 μs
curve / surface fitting	~1-5 ms
lattice methods	~1 ms
PDE solution	~1-20 ms
monte-carlo methods	> 50 ms

Domain specific optimizations may be used, e.g. an approximation look-aside cache. Such optimizations[1] greatly increases load variance. In extreme cases, (e.g. deep out of the money options), a change in inputs will cause no change of output.

Streaming rates on individual inputs can range from 100-500 ms per update on active securities; in volatile markets large number of securities (a few hundred) will be subject to update, resulting in sub-millisecond intervals between updates.

FORCES

Optimization for throughput or latency:

a. A breadth-first traversal favors throughput; because successors of any given node reference a small set of antecedent nodes, cache misses are less likely. This is particularly true of derivative securities – valuation inputs to a derivative (e.g. volatility, yield curves, dividends etc.) are closely related to the underlying security. This may not be true for other application domains and other features may have to be exploited to improve locality.
b. Latency is the elapsed time between input arrival and completion of terminal node update. Minimizing latency requires a depth-first traversal of the dependency graph. Hard limits on the ratio of average out-degree to the number of available processor cores are needed if latency is to be bounded. For shallow graphs, the simplicity of breadth-first traversal is preferred.

Real-time or conflated processing of input:

In market data parlance, the term "conflation" implies compression of successive messages to yield a single message. At times, streaming data rates exceed the

[1] Many computations involve the evaluation of differentiable functions and the computation of partial derivatives is almost universally required. For a small expenditure of memory, previously computed partial derivatives can be exploited to approximate the computation when the change in an input is sufficiently small.

capacity of the system to process all dependant updates without falling behind. To ensure graceful degradation when processing headroom is depleted, a strategy must be chosen. Typical choices are:

a. Retaining unprocessed input in a queue and processing it when peak loads subside
b. Conflation – allow unprocessed input to be overwritten if it is superseded by new data, resulting in skipped input.

These two forces are not orthogonal – when latency is the prime concern, conflation must be chosen. If every input must be processed, simple queuing is the only option.

APPLICABILITY

Fig 2. illustrates a typical computation where nodes correspond to functions of inputs that update in real time.

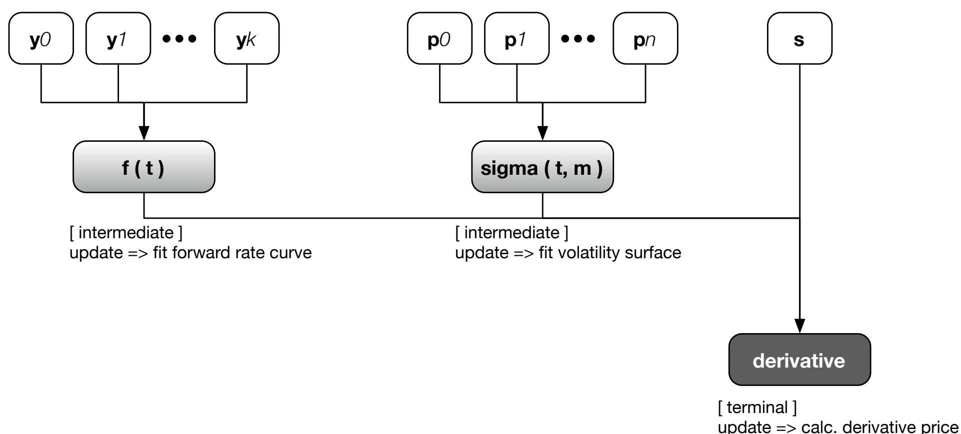
Fig. 2

given:

1) *yield curve* : $f(t)$
2) *volatility surface* : $\sigma(t,m)$
3) *derivative* : $price(s, \sigma, f)$

where:

s : spot price
K : option strike
$f(\bullet)$: fitted using vector $[y_0, y_1, ..., y_k]$ of debt securities
$\sigma(\bullet, \bullet)$: fitted using vector $[p_0, p_1, ..., p_n]$ of quoted option prices
m : "moneyness" measure $\left(\log\left(\frac{s}{K}\right)\right)$



If computations can be represented as a dependency graph whose inputs are linked to streaming data sources; this pattern can be applied.

As input updates are received, update tasks are added to the task queue to be processed by worker threads.

Other criteria for suitability of use are:

- Graphs are shallow - input-to-output path lengths are short
- Asynchronous, streaming inputs
- Dependant computational costs are highly variable, (i.e. fork-join evaluation will have poorly balanced branches and consequently waste processor resources)

Note: It is possible that nodes may be added or removed from the graph as a result of update processing, (e.g. option expiration, barrier events and conversion of a security from one type to another).

SOLUTION

A task graph is setup to reflect the dependency structure of a computation. Task nodes are augmented with state (in the example above, fitted curves and surfaces, derivative prices etc. would constitute this state).

Each node provides a method responsible for updating its state in response to a change in the state of any of its predecessors.

The graph exposes a method `updateNode` using which updates are injected into the system; this injects update tasks for all dependants of the referenced input node into the task queue. Worker threads execute these tasks and inject the next group of update tasks to transitively update successors of updated nodes.

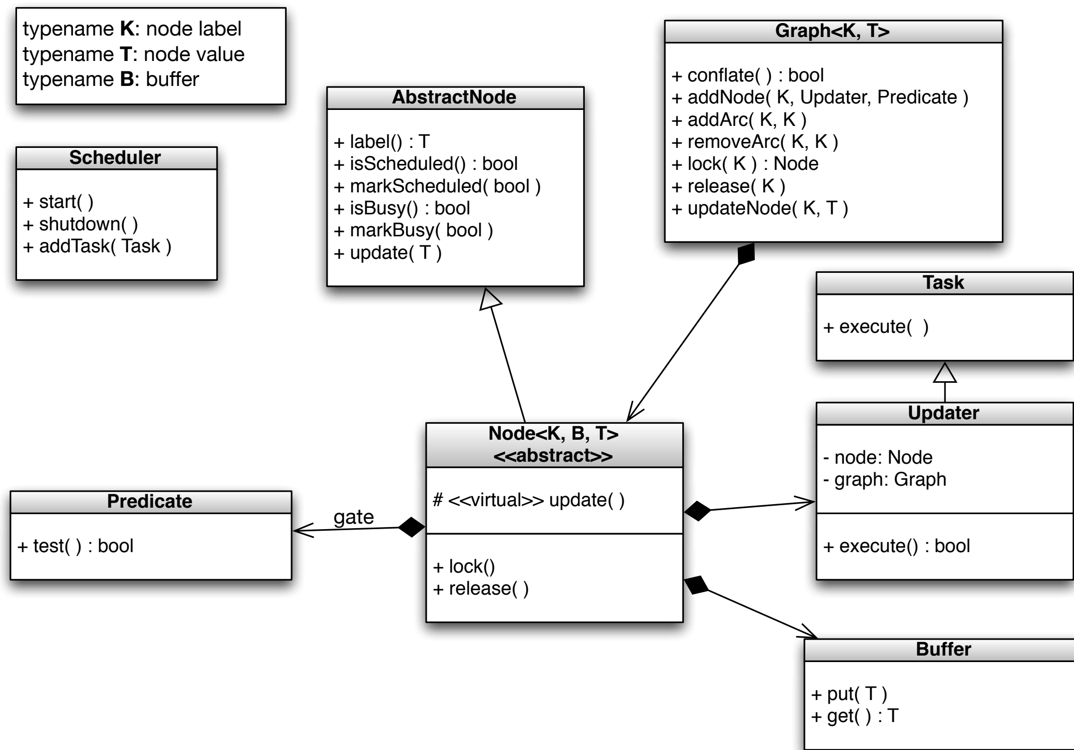
If conflation is specified, the scheduler tags nodes for which update tasks are added – a subsequent task addition request for a node that already has an update task enqueued is ignored. Once an enqueued task is completed, other tasks for its referenced node may be added.

At the time the computation is started, it is initiated using the most recent known values of all its antecedent nodes, so that the results of the computation are based upon current input state.

Participants

Component elements required to implement the graph evaluator pattern are described below.

Fig. 3:



Graph & Nodes

A Graph is a collection of labeled Nodes. Thread safety and low contention are achieved by implementing the graph using concurrent hash map (Intel TBB).

During a node update, the state of its antecedents must be protected from change. Abstract class Node implements explicit lock/release methods, used by the implementation of the `Graph::updateNode`.

Application specialize the Node template using the following template parameters:

- K: node label type
- T: node data content type
- B: class that implements buffering policy. When node data is updated, its dependent nodes are scheduled for update computation. When a scheduled computation begins execution, it must (at least until a local copy is made) place a read lock on predecessor nodes. Node buffering allows new updates to be received while dependant computations have active read locks in place.

Scheduler

The scheduler is based on a simple task queue / worker thread pattern, with the following specializations:

a. For a conflation-enabled graph, the method `Scheduler::addTask` tests if the node associated with a task already has an update task pending in the queue. If that is true, the argument task is ignored; otherwise, it is marked and enqueued. If conflation is not specified, the task is added unconditionally.

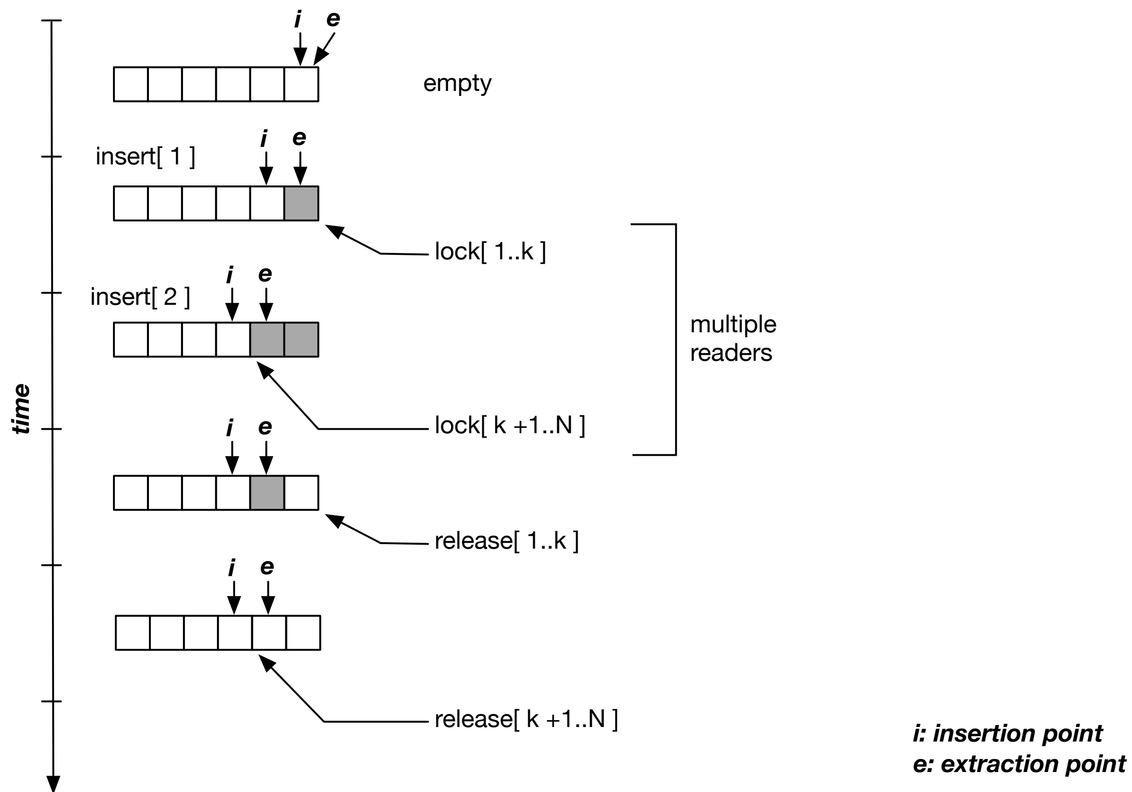
b. After completing the execution of a task drawn from the queue, a worker thread removes the queuing tag and invokes `Graph::updateNode` to cause the successors of the newly updated task to be transitively scheduled for update. This amounts to a breadth-first traversal of the dependency graph.

Buffer

A buffered node allows new input to be received without blocking while its content is in use by tasks updating the state of "downstream" nodes.

Multiple reader semantics are supported to permit concurrently executing tasks to access state / data content of predecessor nodes. A modified ring buffer is used in the current implementation as shown in Fig. 4; the modification allows multiple readers to place read locks on the head entry. Active locks are reference counted to determine if a slot can be reclaimed to receive new input.

Fig. 4



Buffering extracts a cost in memory requirements. It is possible to specialize the node template using a "NoBuffer" class that provides read/write locking on a single instance of the node data type.

Using a buffer improves concurrency, but does not entirely eliminate blocking. Update requests on a full buffer will block. Guideline heuristics for buffer sizing:

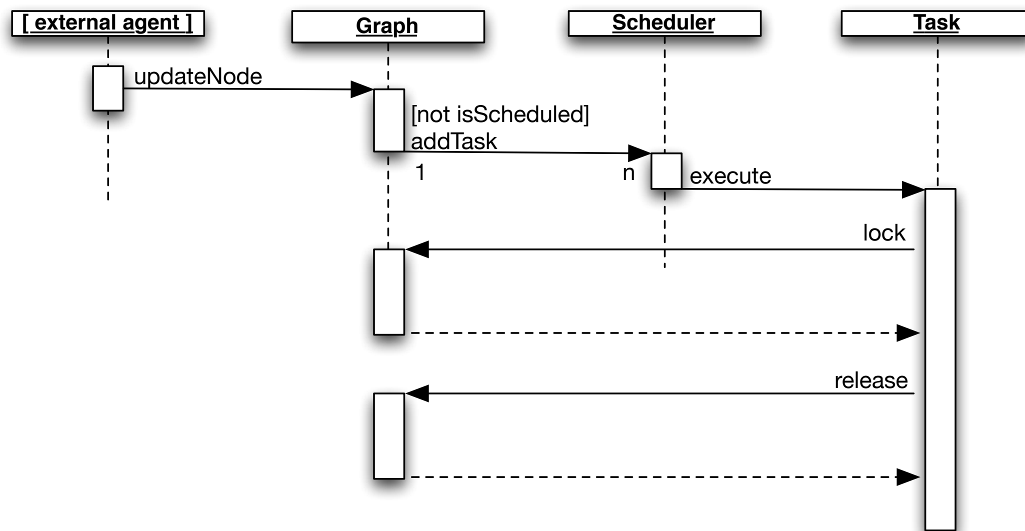
normal	worst case
$round\left(\frac{\text{mean}\ (time\ between\ updates)}{\text{mean}\ (lock\ duration)}\right)$	$round\left(\frac{\max\ (time\ between\ updates)}{\min\ (lock\ duration)}\right)$

A further limitation is illustrated in the figure above; if release,[k + 1..N], occurs before release,[1..k], tasks initiated earlier finish later and the insertion slot remains unavailable until all readers have released their locks. In this instance concurrency degrades to equal a read/write lock.

Collaborations

Fig 5. shows a typical interaction sequence initiated in response to an input received from an external agent such as an asynchronous I/O multiplexor.

Fig. 5



The received updated is processed using a call to `Graph::updateNode`, which:

- enqueues the received value into the node's input buffer (in a conflating graph `Node::setValue` overwrites previous unprocessed input, otherwise new input is queued).

- for every successor of the updated node, enqueue an update task (in a conflating graph, this may be ignored if there is an update task already in the queue)

```
Graph::updateNode( label, value, scheduler ) {
    accessor a;
    if ( _this->find( a, key ) ) {
        Node & node = a->second;
        node.setValue( value );
        iterator s0 = node.successors().begin();
        iterator sN = node.successors().end();
        for( ; s0 != sN; s0++ )
            if ( ! s0->testAndSetScheduled() )
                scheduler.addTask( Updater( this, node, *s0 );
    }
}
```

A worker thread will eventually dequeue the task and invoke Updater::execute; After calling Node::update, the successor nodes of the target are transitively added to the task queue.

```
Updater::execute( graph, scheduler ) {
    Node & node = this->target;
    graph.updateNode(
        node.label(), node.update( graph ), scheduler
    );
}
```

Finally, the implementation of `Node::update` must explicitly acquire read locks on all antecendent nodes that it must access during its computation. Scoped locks can be used for some simplicity and exception safety, though the implementation of the update computation is responsible for ensuring thread safety.

Implementation

Given a problem that can be expressed in a graph (as in Fig 2):

- Derive input Node template specializations for input, intermediate and terminal data types
- Provide implementations of `Node::update` for each type – a skeletal implementation might take the form:

```
MyNode::update( graph ) {
    accessor ul( graph, this->underlyingId ) );
    accessor yc( graph, this->yieldCurveId ) );
    accessor vs( graph, this->volSurfaceId ) );
    setValue( calcPrice( *ul, *yc, *vs );
}
```

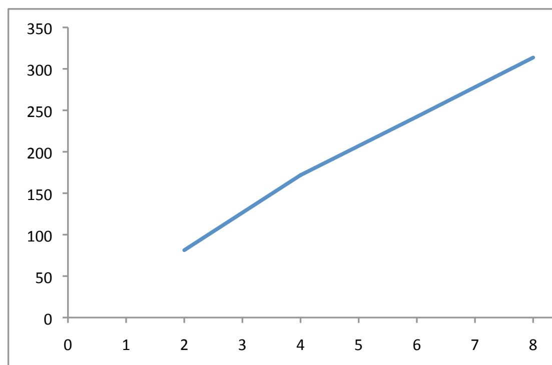
- Instantiate a Graph and Scheduler
- Instantiate every node (e.g. by reading reference data from a database) and add it to the graph
- Setup dependency arcs between nodes – if node instantiation and dependency are interleaved, this process will be sequence sensitive as all nodes have not been created first

Aspects of tuning need to be refined in order to maximize performance, e.g. buffer sizing, data dependent optimizations

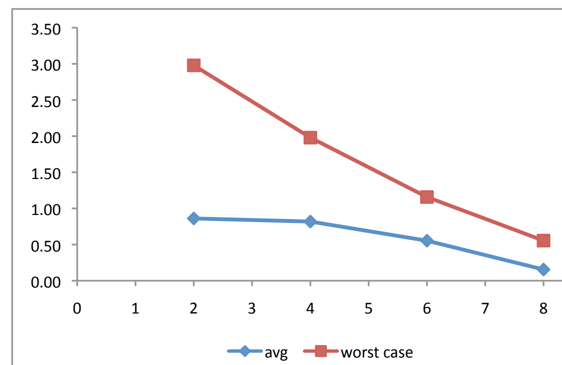
Indicative measurements of scaling behavior are shown below. Throughput scales linearly as cores are increased. Worst-case latency declines asymptotically, leveling off at the amount of time needed for a unit computation. Average latency, however is highly sensitive to the order of node execution; scheduling smaller computations first reduces average latency. Ordering the dependants by computation cost minimizes average latency without affecting worst case behavior.

Fig. 6: throughput and latency scaling

throughput (x1000 nodes / sec)



latency (ms)



A graph with the following properties was setup to test performance:

- 75 input Nodes
- 2000 output Nodes
- Minimum out-degree: 10
- Mean out-degree: 100
- Maximum out-degree: 200
- 3 Node types with computational costs of 100 μs, 1ms and 5ms respectively

Randomly chosen input Node updates were injected at 1-millisecond intervals. The mix of computation costs has an observed mean of 7ms.

Consequences

By encapsulating synchronization & scheduling complexity, the Graph Evaluator pattern reduces the complexity of exploiting hardware parallelism.

An existing computation can be incorporated into the evaluator by implementing a specialization of the Node template whose `update` method invokes the desired computation.

By remaining peripheral to application semantics, existing code can be incorporated without modifications to existing classes; legacy C and Fortran code can also be invoked.

While such code can be incorporated, it is not without the effort of implementing adaptors. At minimum, the adaptation of the `update` method must explicitly instantiate scoped locks on referenced nodes.

In addition, the setup of the graph and dependency arcs must be explicitly done, and deciding whether or not to use node buffering, buffer depth etc. is not easy and requires experimentation.

Related Patterns

Nodes are similar to Adaptors helping to incorporate existing functions or classes into a graph, and to Observers in respect of expressing dependency.

The graph pattern resembles a network of pipeline filters, notably the recently available FastFlow framework (which relies on assemblies of SPSC queues, copy semantics and allocates dedicated copy threads (Emitters & Collectors) to realize lock-free/CAS-free operation.

(http://calvados.di.unipi.it/dokuwiki/doku.php?id=ffnamespace:about).

Conclusions

Directed graphs provide a mechanism for exploiting parallelism in a commonly encountered class of computational problems. High aggregate throughput and core utilization can be achieved with parallel scheduling in the presence of high frequency input data streams.

This work was undertaken in an effort to simplify multi-core migration of existing code by segregating scheduling, locking and concurrency concerns away from application functionality, and to structure the migration as a refactoring problem.

Thread safety does remain a requirement that application code must satisfy; however it has been our experience that this turns out to be less of a concern than it might seem at first sight. State is rarely maintained in static variables; singletons are

straightforward to identify and remove or adapt. As tools for validating thread safety and performance are adopted we expect a continuing migration of legacy code to benefit from the increasing number of available cores.

Acknowledgements

Many people have helped to make this work possible. Sameer Tipnis has been an early collaborator in implementation and testing. Many thanks are due to Arun Lakhotia, Dr. Richard Appleton and Nick Albicelli for thoughtful and in-depth review, and to Sarah Richards for her keen editorial eye.