Wavenets: A new (?) model for parallel servers

John Chapin 3/19/97

Motivation

Existing sequential and parallel programming languages are based on a turing machine model: input, processing, output, halt. However, many important programs (operating systems, databases, Emacs, web servers, ...) do not fit this model. These programs are *servers*: they run forever and must satisfy multiple input requests in parallel. Although turing machine-based languages can be (and today are) used to implement these systems, they suffer from a major disadvantage: the abstraction mechanisms present in these languages work poorly for these problems.

In particular, if you attempt to implement a server on a parallel machine, you quickly find that any procedure or function that contains a synchronization action (lock grab, semaphore release, etc.) cannot be treated as a black box. Fixing synchronization problems in today's large complex systems requires studying the internals of all modules in the system at the finest possible level of detail.

Similarly, the mechanisms used to model the meaning of programs in these languages break down in parallel servers. Each function or line of code is considered as a state transformation: input, processing, output, halt. However, in a parallel implementation of a server the concept of "state" is poorly defined: many different actions are proceeding in parallel, all modifying the state of the system. Misunderstanding what is "state" and what is not from the perspective of any given function or line of code leads to the classic race condition bug, which in my experience is the most pernicious bug in large production parallel systems.

One solution to the problem is to implement servers only on non-parallel systems. In fact the early versions of UNIX that ran only on uniprocessors were notably simpler and easier to understand than today's multiprocessor implementations. This option is unattractive today because multiprocessors are widely available and becoming cheap. I predict most desktop PCs will be multiprocessor (or at least multithreaded) within 5 years, and application vendors will want to take advantage of the parallelism for performance.

Another solution is to rigorously follow a monitor-based approach or use an object-oriented programming style in which each object has a monitor lock. System builders who have tried this approach uniformly report that it runs out of gas at even medium levels of complexity due to the nested monitor problem.

There are various "concurrent object-oriented programming" methodologies designed for implementing parallel servers, and corresponding theoretical models such as CSP. These approaches essentially change a centralized server with shared state into a distributed system without shared state. The programmer then has to cope with all of the distributed system issues (except failure), such as resource reservations and their corresponding deadlock problems, update ordering, and so on. More fundamentally, distributed systems like this are hard to understand and modify because most important algorithms are spread across multiple objects.

My proposal is that we reexamine the fundamentals of the problem. If a turing machine is not a useful model for a parallel server, then what is? Once we have a model attuned to the problem at

hand, we can begin considering abstraction mechanisms that can deal with the complexity of a parallel server without partitioning it into a distributed system. The model need not worry explicitly about the parallel implementation at first, but should allow representation of the system at a high enough abstraction level that parallelism can be extracted from it.

The wavenet idea presented here is my first attempt at such a model. The intuition is that a network of springs can compute: you wiggle a spring at the edge, the wave goes into the network, interacts based on the way the network is wired, and eventually a response comes wiggling back out at you. Multiple requests and responses can be active at the same time. If such a model could be made turing-complete, it promises an interesting new way to represent what a parallel server is doing.

Given that intuition, I threw into the mix ideas from lazy languages, circuit design, and the copenhagen interpretation of quantum mechanics. The result may not be efficiently computable but it is turing complete. It also has one major property that I was hunting for: wavenets are *composable*. If you have one implementing one subsystem and another implementing a different subsystem, you can plug them together and their constraints join together to form a correct union that expresses the maximum available flexibility of the system. If this property can be built into an efficient programming language, we will have made a major step forward in our ability to build and reason about complex parallel servers.

Structure

A wavenet is a 4-tuple (V, E, L, S) of nodes V, edges E, latches L, and initial state S. The edges E are directional arcs (n_1,n_2) where $n \in (V \cup L)$. Each node $v \in V$ has at least one input edge (n,v) and at least one output edge (v,n). Each latch $l \in L$ has at most one input edge (n,l) and at most one output edge (l,n). Latches with an output edge only are called *input latches*, latches with an input edge only are called *output latches*, and those with both input and output edges are called *storage latches*.

Each node v contains a nonempty set P of *production rules*. Each rule $p \in P$ has the form

$$i\ (+\ i)^* \rightarrow_{[\varepsilon]} o\ (+\ o)^*$$

where each i is an *input term* and each o is an *output term*. Terms have the form e:s where e identifies an edge of the node and s is a symbol. A production rule using $\rightarrow$ is a *standard production* and a production rule using $\rightarrow_\varepsilon$ is a *default production*.

The initial state S is an assignment of a symbol or $\perp$ (*empty*) to each storage latch. All input and output latches initially contain $\perp$.

Semantics

Computation proceeds by alternating *propagation* and *latching*. In the propagation phase, all production rules in the wavenet are activated until values on the input edges of all latches stabilize. In the latching phase, each latch grabs a value from its input edge.

Think of the wavenet as a quantum-mechanical system. Each latch outputs a wave of information (a *meme*) carrying its output symbol. The meme can contribute to only one production rule in each node it passes through, but the decision as to which rule activates is delayed until the meme collapses in the latching phase. In order to delay the decision, the meme divides at each node where a decision must be made, combines with other memes propagating through the wavenet,

and eventually contributes some component to the probability that a latch will observe some symbol on its input.

Propagation: Each latch outputs a meme carrying the symbol it stored in the previous latching phase, unless it stored ⊥ in which case it generates no output. A production rule *activates* if all its input terms match memes that are present on the corresponding input edges. For example, a rule e_1:a → ... activates if a meme carrying the symbol a appears on input edge 1 of that node, while e_1:a + e_2:b → ... activates only if there is a meme on input edge 1 carrying the symbol a and a meme on input edge 2 carrying the symbol b.

When a production rule activates it places memes on the node's output edges for each of the output terms of the rule. For example, a rule ... → e_1:c places a meme carrying the symbol c on output edge 1. The output memes are *children* of the *parent* memes that contributed input terms to them. One parent meme can contribute to multiple children at the same node if there are multiple active production rules.

There are two types of memes. Those output by latches or by standard production rules are called *standard memes*. Those output by default production rules are called *default memes*. All descendants of a default meme are default memes.

Multiple memes may be present on an edge at the same time. In fact, the same symbol may be on an edge multiple times, carried by different memes. The memes do not combine in this case; each meme is independent and is considered separately in further nodes or latches. Similarly, a production rule can activate multiple times, with each activation being independent.

A meme that passes through a node and splits into two child memes can take only one of those paths when a decision is finally made. Therefore the propagation phase follows the *no-diamonds* rule: a production rule does not activate if it would combine descendants of the same meme.

Latching: After propagation is complete, each latch may observe memes on its input edge. In the latching phase, each latch *grabs* one meme in order to store the symbol it carries, or grabs no memes and stores ⊥. A meme *collapses* when one of its descendants is grabbed.

Which memes are grabbed by which latches is random but satisfies the following constraints:

- A meme collapses only once (i.e. only one of its descendants is grabbed).
- All memes collapse.
- As few default memes as possible are grabbed.

The wavenet *jams* if uncollapsed memes remain after latching. Jamming is an error condition.

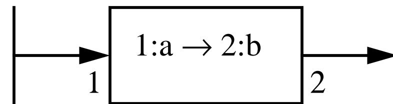
Input/Output: The wavenet receives input through its input latches and produces output through its output latches. Note that a symbol that appears in an input latch must be consumed immediately or the wavenet will jam. Similarly, a symbol presented to an output latch in a single latching phase is assumed to have been consumed by the environment. These semantics are chosen to support composition of wavenets (see below).

Output latches can be used to sink discarded memes that would otherwise cause the wavenet to jam. Such a latch is called a *ground latch* by analogy to electric circuits.

Examples

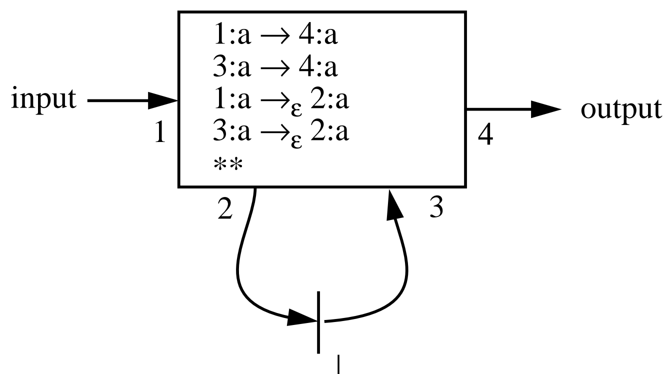
In the following diagrams, a vertical bar across an edge is a latch. Edges have arrows indicating the direction of information flow. A symbol next to a latch is an initial value.

Turing machine: One valid model (in the algebraic sense) for a wavenet makes a “symbol” any arbitrarily large or complex data structure and a “production rule” any arbitrarily large or complex function. Turing-completeness follows immediately:



Wavenets are turing-complete even without this sleight-of-hand. Proof is by straightforward construction. [Notes: big array of overwrite cells for the tape, one latch containing current position, node that combines that latch with inputs from all the cells to output current symbol, big array of nodes on the other side, one per state, latch containing current state id, node that uses current state id to forward current symbol to correct state node, take output from that node to update tape, state, and position latches].

Storage cell: How to store something for a while. Use of default productions ensures that the storage latch only grabs a meme if whatever is downstream cannot consume it.

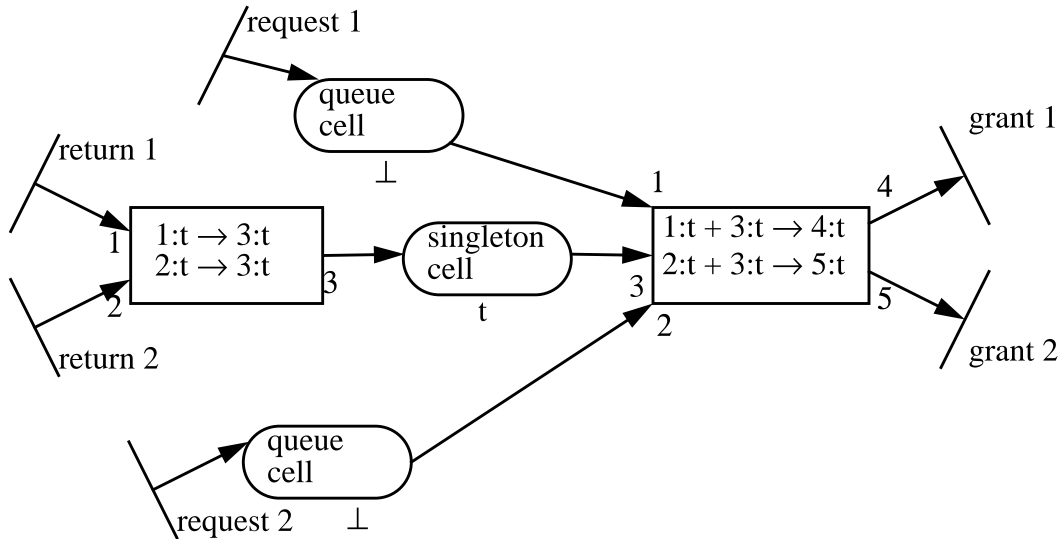


**Different variants of the production rules modify storage cell behavior in the case that a new meme arrives at the input when the output has not yet consumed the symbol stored in the latch. Storage cell types include:

- Overwrite cell: new input always goes into storage.
  1:a + 3:a → 4:a + 2:a
  1:a + 3:a $\to_\varepsilon$ 2:a
- Queue cell: new input ignored if storage latch full.
  1:a + 3:a → 4:a
  1:a + 3:a $\to_\varepsilon$ 2:a

- Singleton cell: jam if new input while symbol stored.
  No two-input-term rule.

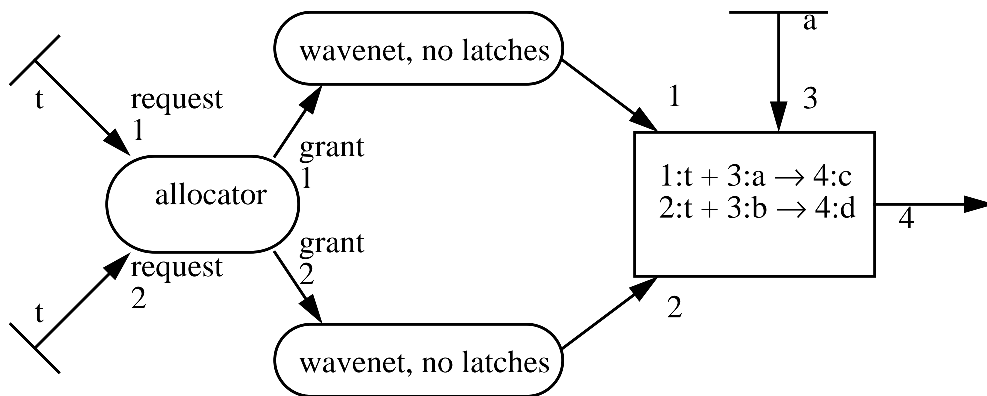
Arbitrator: A server that grants exclusive access to some shared resource. Clients request a token and then return it. This implementation relies on the randomization of grab order to avoid starvation. Note that if either client has a bug and returns the token when it doesn't have it the system will jam, which is good. The arbitrator grants a request in a single propagation phase if the token is present.



Composability

To compose wavenets, you plug the output latch of one into the input latch of another and *remove the pair of latches*. This has a powerful effect: memes propagate from one wavenet into the other before collapsing, so constraints in the downstream wavenet modify the behavior of the upstream wavenet. If the two wavenets contact at multiple points, the new constraints can flow in both directions without the programmer having to think about it.

Consider the allocator shown above. It randomly grants the resource token to either client 1 or client 2. But what if it were composed into the following larger system.



In this case the input symbol a to the node at the right forces the latch at the far right to collapse the 1:t production rule rather than the 2:t production rule. This constraint reaches back to the allocator and guarantees that the token is allocated to request 1 rather than to request 2.

This type of lookahead is useful for composing complex subsystems. The parallelism and behavioral constraints of the composition is directly derived from the constraints in each subsystem and so will not violate them.

Building systems based on the model

There are two ways to build systems based on the model: either compile to a more standard threads and locks implementation, or interpret the wavenet itself (actually alternate propagation and latching at run time). Compilation is worth investigating, but only the interpretation version is discussed here.

The problem with interpretation is that computing the latching is a global optimization problem. It appears NP-complete to choose an assignment of memes to latches such that the wavenet does not jam. This leads to several questions:

- Is there a way to constrain the wavenet so that there is an efficient algorithm to find a non-jamming grab assignment?
- Is there a heuristic algorithm that does well in the cases likely to occur in practice, and a companion algorithm that can determine at compile-time whether a given wavenet matches the requirements of the heuristic?
- Can we specify some constraints on the wavenet that, if followed, render its externally-visible behavior insensitive to the grab assignment?

We obviously want to parallelize both propagation and grabbing if implementing a wavenet on a multiprocessor. Propagation is embarassingly parallel, although some creativity is required to handle the no-diamonds rule efficiently.

- Is it more efficient to propagate backwards from latch input edges?
- Forward propagation allows pattern matching by input terms. Does adding pattern matching change the expressiveness of the model?
- How much of the propagation can we precompute? Computing a random number and a few boolean predicates at the start of the propagation phase may allow bypassing both propagation and latching. If so, the output latch values could be computed entirely in parallel after dissemination of the initial predicates. For example, the arbitrator could be compiled this way efficiently.
- It is straightforward to transform a wavenet in which some production rules could activate multiple times in the same propagation phase into one in which each rule activates at most once. This requires replicating nodes. Does the simplification of the propagation algorithm provide sufficient performance benefits to outweigh the space cost of replication?

Parallelizing the grabbing phase is a much more interesting problem. Two latches are *collapse-independent* if the memes on their input edges cannot share common ancestors. Two latches are *grab-independent* if there is no combination of grabs they can make that will make it impossible to satisfy the global properties of the grab assignment.

- Can two latches be collapse-independent but not grab-independent?
- Is there an algorithm that can compute in advance which latches (or which latches under which conditions) are grab-independent?

Optimizing a wavenet for efficiency consists of transforming the graph to minimize the maximum size of a *combinational subnet* and maximize the fraction of subnets that are *simple*. A combinational subnet is a collection of nodes and edges that must be considered in a single propagation phase. A simple subnet is a combinational subnet where the grab assignment can be computed offline.

Possible transforms include inserting latches, collapsing adjacent nodes, replicating nodes, and reordering nodes and latches.

- Which transforms are valid (ie. preserve the semantics of the wavenet)?
- Which transforms do not preserve full look-ahead but are otherwise valid?
- Which subnet patterns are irreducible?
- How can we give feedback to the programmer to assist them in modifying their system to eliminate the largest irreducible subnet?

An *asynchronous wavenet* is one where propagation and latching phases can alternate as fast as possible in each subnet. A *synchronous wavenet* is one where all combinational subnets must latch at the same time. If the original wavenet (generated by the programming language before optimization) is asynchronous, which valid transforms preserve this property?

Objections to the model

Q: I introduced wavenets by arguing that the concurrent object-oriented programming methodologies intended for parallel servers are not attractive because they force the programmer to program to a distributed-system model. Wavenets also delete the concept of shared state from the system, so don't they have the same problem?

A: The fundamental difference is that a distributed system forces the programmer to think hard about the order of operations, and the programmer must make decisions at point A before getting over to point B and discovering that the state of the world is not consistent with that decision. Wavenets free the programmer from those concerns by transparently propagating information about the constraint at point B to the decision at point A.

Q: A wavenet's behavior changes when it is plugged into a larger system, so therefore a wavenet tested and validated in isolation can no longer be trusted in the larger context.

A: The behavioral change is of a limited, tractable type: a wavenet can be prevented from taking some action but not caused to take an unexpected action. In other words composition preserves safety but not liveness properties. It may be possible to create tools that validate the preservation of liveness, or at least can answer for any given execution of the system what constraints prevented a certain state change from occurring.

Q: Wavenets are essentially circuits. Anyone who has both programmed and designed circuits knows that it is much harder to implement complex interesting behaviors in a circuit than in a program which has conditional branches and data structures. Won't wavenets have the same problem?

A: The turing machine sleight-of-hand that was the first example above points the way to the solution. A wavenet-based language could be used to express the glue level of a system at which the subsystems interact. Once parallelism and other constraints have been (dynamically) resolved at the glue level, lower-level code in a traditional language can implement the productions. Each symbol can be a full data structure. This is related to my *programming by modeling* approach which is discussed elsewhere.

One advantage of organizing a system this way, with a circuit-like model at the top, is that some of the approaches that have been developed in the VLSI community to validate circuits and measure test coverage may be applicable to parallel implementations of parallel servers. This would be a significant improvement over the state of the art in validating these systems.