"Simultaneous" Considered Harmful: Modular Parallelism (deprecating "locks", "semaphores", serializability, and other sequential thinking)

David P. Reed
SAP Research

Abstract

The dominant view of computing is based on sequential processing as the "normal" case, with parallelism considered to be optional "acceleration." In this position paper, we argue that parallel is becoming the norm, so we must re-examine the primacy of serialized computations in time, simultaneous action-at-a-distance, and total ordering in our thinking and our engineering practices.

We focus in this paper on some of these problematic ideas, design, and implementation structures that must be deprecated, and cleaner modularity concepts supported, if we in the systems part of the computing community are to unshackle parallel computing from its misplaced entanglement to a sequential model of computing. In the end, ours is a call to action, not revolutionary action that discards the past, but rapid evolutionary change, abandoning obsolete ideas.

Overview

"We live in a parallel world," in two different ways – one ancient, one nascent. The world of human experience and our understanding of the physical world has always included many actions happening at once.[1] But we've inherited a view of computing, shaped by mathematical models such as stepwise lambda-reduction, Turing machines, von Neumann architectures [Minsky67], and procedural languages, that is strongly based on *sequential* processing as the "normal" case, with *parallelism* considered to be optional "acceleration" – so complex and dangerous that only computing's high priests could understand it and practice it, and only in small doses.

Were computing to have begun differently, perhaps evolving from analog computers and programmable logic specified in RTL, parallel computing would be the norm in computing today, while sequential computation would be an unusual case. Much of the complexity related to parallel computing today arises from some weak early concepts that we re-examine in this paper.

The fundamental source of complexity in parallel programming is that parallelism disrupts *good* modularity. It does so because we build parallel systems based on a notion of time that is thoroughly tangled with sequential processing concepts – involving absolute simultaneity and total ordering of events, and a co-routine notion of multiplexing a sequential machine.

We begin where Einstein began in formulating his theories of relativity – by making the notion of "simultaneity" unspeakable (in both the philosophical and "polite society" senses), hence the title. Remarkably, we lose nothing by abandoning absolute simultaneity[2] and gain much, especially in building parallel systems that can scale in power, with reliability, and simple design.

Second, to show the importance of defining proper abstractions for effective modular parallelism, we revisit the abstraction that is conventionally called an "atomic action", to derive a more natural meaning in a parallel world, with semantics derived from Parnas' *information hiding criterion* for good modularization [Parnas72]. This alternative, natural derivation was first made explicit in the present author's 1978 Ph.D. thesis[Reed78] as the basis for unifying the mechanisms of recovery and synchronization in physically decentralized data management so that a) modules were composable, and b) modules were timing- and fault-tolerant. We suggest another new modular abstraction – the *idempotent action*, used to compose replicated computation for latency reduction or repeatable execution for fault-tolerance. These clean abstractions support modular, parallel composition of independently implemented, decentralized subsystems.

Third, we focus on the need to virtualize parallel computing to hide implementation that embeds sequentiality and simultaneity. The concept of virtualization is a

1 We are now learning from neuroscience that the illusion of "single-threaded" consciousness is a mental fabrication to "explain" the body's actions and perceptions – *ex post facto*. [Gazzaniga11]

2 Absolute simultaneity, abandoned in special relativity theory, presumes one universal total ordering of all events. Relativity replaces absolute simultaneity with "causal ordering" - events cause others, *only* causally related events have meaningful ordering.

key part of our systems engineering repertoire.[3] VMWare is one current example. The benefits of virtualization go beyond its origin: multiplexing underlying physical resources for cost economies. In particular, virtualization is key to security, fault-tolerance, capacity management, and now power management of systems by dynamic, late and reversible binding of programs and data to resources. Yet many of the virtual abstractions in use today fall apart when confronted with today's parallel systems. However, by virtualizing entire systems rather than components, virtualization delivers its full set of benefits to parallel systems – if the virtualized abstractions are defined without basis in "simultaneous" events and total orderings.

Our focus must be on systems' evolution and adaptation. Parallelism should not be a brittle patch on a fixed global design – system designs must be locally extensible and always evolving in place. The most compelling new parallel computing chalenge has evolved in the context of mobile computing and the Internet – modern "e-commerce" is a vast, open-ended, evolving naturally parallel system at all levels of abstraction.

Internet services are built by composing many component services provided from many sources, while mobile users expect that requested actions complete reliably, consistently, and irrevocably throughout all elements of the system. Many separately managed systems must combine dynamically, operating at the same time in order to achieve the latency and reliability expectations of all the users, and the system must continue to function through partial outages with little or no disruption of services. Treating this parallelism differently than designed-in parallelism would make such systems intractable, perhaps even deadlocking in hidden shared services. Good modularity is a must.

Simultaneous action: unneeded and costly

Leslie Lamport first used the metaphor of special relativity to suggest why controlling causal ordering of events was necessary and sufficient to build correct decentralized systems. [Lamport78] This author, and many others since, have devised computing mechanisms based on time-like orderings (virtual clocks) that are compatible with causal ordering, allowing us to say they are equivalent to *some* "sequential" execution of steps - "as-if serialized". In defining causal ordering, both special relativity and Lamport-time replace the notion of absolute simultaneity of events in the system with a much weaker definition that two events are *simultaneous* if the pair has no causal ordering, but there is *no event* that precedes one, but follows the other. In other words, they *may* happened "in parallel" (or not).

3 Virtual processors running on virtual memories is a concept that goes back to computer systems way before VM/370 – operating systems early adopted the "process abstraction" [Dijkstra68b]

Whether they do happen in parallel does not matter, and cannot be known inside the system. In fact, even defining this "weak" *simultaneous* implies a closed, bounded system – because the term "no event" requires that all events be knowable at design/proof time. But we cannot know what will change as the system evolves... in open, unbounded, evolving systems, i.e. any useful computer system, there is no utility to speaking of "simultaneous" at all; a more economic way of saying this is that simultaneity is an "unspeakable" - a verboten concept that should not be used in constructing or showing correctness of any part of the system.

In other words, the concept of "simultaneous" should not be necessary to define or prove the behavior of a system, just as the "GOTO" is not needed to write any sequential program.[4] But many of our programming abstractions are defined in terms of absolute simultaneity. This leads us down a difficult path, requiring global synchrony in the systems implementation. Yet the goal of parallelism is usually something simpler – for example: minimizing end-to-end latency in a system that need only be causally ordered.

Other powerful modular abstractions besides atomic actions are important in system design, and should be defined in a naturally parallel ways, so they can be implemented without using "simultaneous action". We focus on clean *atomic action*s in the next section, and there are others. For example, *idempotent actions* defined below are useful for fault-tolerance and latency reduction.

Many of our primitive "parallel" programming techniques promote poor modularity – where local changes in one part of the system have global effects, such as wrong answers or deadlocks. Common to these abstractions is the use of "simultaneous" to define their *effect* or meaning ("as-if simultaneous"). Mutual exclusion and arbitration are the concepts that lead us into trouble, and mutual exclusion and arbitration embedded in low-level programs or hardware are the worst. For example, in the MESI cache coherence protocol family, writing a new value for a cached memory cell requires that the write happen *only* under the condition that all cache managers in the system agree *simultaneously* that the caches in all other processors in the system will stall access rather than let them execute a read or write to that address. Ultimately, this *definition* of cache coherence does not scale, because it may require a global system "stall" across all computing elements.

Worse, "atomic compare and swap" operations, etc. also are defined as "simultaneous" among active computing elements (perhaps via the MESI protocol's exclusion of simultaneous writes). Such facilities may be useful in very "local" contexts, isolated within a module, but because the memory hardware is not aware of

4 Hence the title, referring to Dijkstra.[Dijkstra68a]

the meaning of these operations in a system design, it must implement the most global meaning of "atomic".

Another primitive defined in most operating systems such as Linux and Windows, is the *mprotect* syscall that changes access rights to shared memory. The semantics of *mprotect* require enforcing simultaneous action across the entire system – every event throughout the system can be only "before" or "after" the *mprotect*. Thus *mprotect* does not scale, nor does it support good modularity in a parallel system. Its cost is linear in system diameter. the cost *slope* can be tuned [Multik09], but not zeroed. Limiting *mprotect*'s scope would create a more modular and more scalable OS primitive.

Locks on records and tables in DBMS's also break good modularity principles, requiring simultaneous action, and having correctness un-connected to data semantics.

And at the largest scale where humans, the natural world, and systems interact, "correct input-output behavior" need never include specifications depending on a total ordering of all events in the physical world. Causal ordering suffices, even at this scale. Invoking "simultaneous" or "absolute ordering" adds unnecessary complexity, implying a global "arbiter" to assign a total ordering to independent I/O events, though a causal ordering matches physical reality. (Approximate timestamps or tickets can model input, and output deferral can control *necessary* orderings of output.)

Our point is that from the external point of view, programs, systems, and so forth can be fully and completely *specified and implemented* without using the concept "simultaneous" in its absolute, temporal sense. Causal ordering is sufficient, and causal ordering does not contain absolute simultaneity as a concept. Absolute simultaneity and total orderings are *too powerful – so powerful that they are difficult or impossible to implement well.* Avoiding all abstractions that imply absolute simultaneity, systems will scale better. Researchers in theory of distributed systems have explored this; it's time we systems engineers applied it.

Modularity, not Locks and Logs

When the System R team at IBM pioneered relational DBMS implementation techniques as the Winchester disk moved data from tape to disk, they confronted the problem of implementing the relational algebra on an inherently asynchronous collection of sequential elements – a sequential processor and memory, disks that saw one record at a time pass under its read/write head, users entering queries and data from multiple terminals, etc. Clever optimization of the asynchronous disks' timing, while multiplexing the underlying processor and memory resources to match, could vastly improve performance. Designers needed to have a clear definition of "correct operation" that did not depend on their internal implementation. The team chose to define correct behavior of the system using *serializability* – that observable behavior would be correct iff consistent with some "serial schedule" of all queries and updates [SRLock76]. Correct behavior of the implementation-could be shown by correspondence to a serial schedule. The implementation thus used various forms of mutual exclusion to enforce orderings that were demonstrably "equivalent to" some serial schedule, even though some parts of each operation were executed "out of order." (note implicit time-based reasoning)

This work set a precedent that "transaction atomicity" in its "ACID" formulation *must* be based on sequential execution "with small safe permutations", because of this choice: correspondence with totally ordered execution in both definition and implementation.

When this author designed a framework for modular composition and coordination among relatively autonomous, decentralized systems a few years later[Reed78], he had to define an abstraction that allowed for the modular composition of parallel activities. He (and others) began using a new term - "atomic action." However, the term was difficult to define in an "open-ended" way – the System R team could presume that all transactions would be known to the system at all times, but in a looser system designed by autonomous actors, one did not want to build global monitor that must know all actions that were occurring at all times; communications outages and system failures made such knowledge impossible to gather. Here's the author's definition:

We define an atomic action as a program-specified computation that, although composed of primitive computational steps executed at different times and in different places, cannot be decomposed from the point of view of computations outside the atomic action. During the execution of atomic actions, intermediate states of data objects that arise will never be observed by computations outside the atomic action. During the execution of an atomic action, objects whose values are read by steps of the atomic action can be modified only by other steps belonging to the atomic action during the execution of the atomic action. [Reed83]

Replacing the words "during" with "within", no reference to timing need appear in the definition. This definition of "atomic action" fits Parnas' *information hiding criterion* for modularity: the specific information hidden is the locus of computational steps of a module. Step order must not be exposed, *no matter how other computations might try to access the functional units, memory stores, and data flows of the atomic action, whether or not the atomic action completes or encounters an obstacle, no matter what other activities in the system are executing.* In this framing, an atomic action is isolated in a *causal* sense – it does not cause, nay cannot cause, external events except those exposed in its specification.

Our point here is to define atomic actions as a tool for obtaining good system modularity, with its engineering benefits, not "as-if-serialized" internal execution.

Easy composition of modules to create larger modules is one of the benefits of modularity. This definition of atomic actions provides that benefit. The difference between the "serializability" criterion and the "causal isolation" definition above makes a huge difference. In a sequential system, composition is serialization: A composed with B means "execute A, then B". In a parallel system, one can compose two computations either serially, or in parallel - "execute A then apply B in the resulting state" or "execute A and B on the same state, combining the results into a new state". Of course, both are useful modular compositions – but the parallel one is "ruled out" by reducing correctness to serializability – one has to "add back in" the notion of parallel execution to get the benefits of using resources at the same time.

This is exactly where the "serializable" definition of atomic action causes trouble: systems based on "locking" or other forms of global exclusion in time make it very hard to do parallel composition of actions. For example, a single global lock can be used to implement all atomic actions in a system, if all we care about is correct serializability – the art of designing fine-grained locking protocols with various distinctions about reading, writing, etc. is often added later to enable more parallelism constrained precisely by the roles of data in each "atomic action module", but fine-grained locking may not provide causal isolation, which is why changes to the system's design that are "outside" the atomic action may require redesigning the locking inside what should have been an independent module. Parnas points out that when design changes outside a module require redesign of the module, one has chosen a poor modularity.

Idempotent actions are composable modules whose effect is independent of how many times or places the idempotent action is started. An example would be the "thunk" that implements a Haskell expression. Idempotent actions can safely be executed in parallel, can fail to complete, and prevent "stalls" in temporarily disconnected subsystems. In most mission-critical systems, it is essential that preventive maintenance and repair of the faulty portion of the system proceed "in parallel" with continued operation of the functional portion of the system. What we really mean is that the functioning part cannot be affected by the maintenance and repair processes, and vice versa – we must design the system with modularity of a type that allows disconnection and replacement or re-connection of new modules. Idempotent actions are a clean way to deal with maintenance and recovery; *undo/redo logs* are often used for this purpose, but like locking, log writing breaks good modular boundaries in a parallel system.

An example of this is "replicated computing". Rather than carry out the evaluation of a functional module in exactly one place, one can evaluate it in many places, at many different times giving the same result. Such redundant evaluation can enhance fault-tolerance and minimize the effects of communications delays or failures. Of course, minimizing redundant evaluation often reduces cost. In Haskell, a "thunk" implements a functional module in such a way that it need be evaluated exactly once, but the concept is more powerful than that – because executing it more than once is OK, too, because of idempotency. This idempotency allows multiple evaluations to occur (or not) as needed [MarJonSin09], eliminating the need for mutual exclusion at all, potentially overcoming communications' latency, and providing a simple tool for fault tolerance.

Effective exploitation of natural parallelism requires us to design systems at all levels using modular abstractions that support parallel composition of modules, without respect to modules' implementation.

Virtualization of parallel computing

One might fear that any new programming paradigm that abandons simultaneous action might require rebuilding all of our current system's designs and investment in software applications, at great cost. We would argue otherwise. Most current software systems are designed in a serialized model, and few attempt to use parallel computing, while even fewer gain very much from parallel architectures. This is due to Amdahl's Law, of course – as long as parallel computing is confined to a niche used as a "last resort", most programs are written in a sequential manner, using sequential modular abstractions, with parallel aspects forced into a Procrustean Bed of sequential execution. Automatic parallelization of such programs is quite hard, because the specification becomes "overconstrained".

Besides deprecating the primitives mentioned earlier, we need to evolve the "system" abstraction itself, from sequential processor, sequential I/O, and sequential memory to a naturally parallel model.

Though some have proposed that message-passing seems more parallel than shared memory as a "causal ordering" concept, the issue is neither with using virtual memory for holding intermediate results or a repository for state, nor a shared virtual address space. Two examples suggest the problem here: 1) vector convolution can be done in parallel by taking the FFT of the two vectors followed by pairwise multiplication, and an inverse FFT; no compiler can take the C-code of a highly optimized sequential composition algorithm to construct the FFT-based algorithm and schedule it on parallel underlying resources. If we know that cores are cheap and plentiful, it would be better to allow the operating system and hardware to map a parallelized, causally ordered implementation onto available re-

sources. 2) The DBMS language SQL's semantics is based on a small set of highly parallel operations – selection, joins, union, sorting/merging, etc. of tables and columns, and extensions to SQL include additional functional operators such as mapping and reduction. Yet *all* commercial SQL *implementations* execute such operations in an "as-if-sequential" manner, probably because they all inherit the "serializability" correctness criterion derived from the System R *implementation techniques*, but not from SQL's semantic model. Worse, the modular interfaces to SQL implementations embed sequential constructs like cursors that destroy parallelism.

Since today's hardware support extremely efficient reversible bindings of virtual memory to physical memory (L1-L3 cache and RAM), and execution to cores, the execution of these naturally parallel systems or modules can be collectively bound to memory and processing elements that are "wired together" by an OS or hypervisor into a virtual parallel system under control of a high level scheduler.

Rather than virtualize individual memories or CPUs or I/O devices, we virtualize entire parallel "systems", either in hypervisor or OS, in a Virtual System that can span many system elements. This system can be quite heterogeneous, but the key thing is that *every* system resource is a virtual abstraction, temporarily bound to a set of resources. Of course, today's "legacy" VM concept can be a restricted kind of the Virtual System, encapsulating "legacy code" that won't scale well anyway.

This is a small, evolutionary change, but it can have a huge impact, because today's virtualization boundaries are too fine-grained, exposing quirks of the virtualization techniques themselves, especially quirks where too-tight bindings to specific hardware create unmanageable interactions with independent activities sharing the same platform resources.

I/O at the boundary of an entire Virtual System (not between modules, but at the physical/human interface of the computing system) has to map the virtualized system behavior to "reality". This requires another specific approach to isolation of the sort that enables parallel composition in Haskell, which exploits non-strict ordering of evaluation (including lazy evaluation). Haskell's "IO monad" provides a clear separation between the highly parallelizable "functional" parts of the system, and the causally ordered external world, in an elegant way. One need not adopt Haskell as the interface – but the separation of external causal ordering from internal execution timing is very cleanly done. A similar decoupling is included in the Remus extension to Xen, and in the event-driven programming style used to construct GUI applications, (and also used in many "device driver models" in operating systems) suggests how one might start to organize a virtual machine for parallel computation, if it were extended to include inexpensive "fork" and "join" operations.

Shared address spaces (as a form of naming and inter-thread communication) are extremely useful tools that match well with the way we organize intermodule communications (stack frames, object "instance variables"). The issue with sharing memory is that the notion of *current* contents of a memory cell is defined using "simultaneous" and "mutual exclusion" concepts. This is easy to fix, without modifying today's hardware architectures, because the hardware architectures support local contexts, local contexts can support distinct content *versions* in caches and "copy-on-write" techniques. The problems arise from attempting to achieve semantics implicitly based on simultaneous global changes, such as one global page-table hierarchy across the whole system. This simulation requires enforcement of absolute simultaneity, leading to the problem of unscalable TLB shootdowns, etc. One need not abandon shared address spaces, if one provides the right isolation between independent concurrent activities, by using distinct page-table maps for isolated computations. The underlying memory can be shared, even allowing virtual memory in isolated modules to have the same global address. Conflict resolution can be enhanced by library subroutines knowledgeable of hardware (so, for example, false sharing can be resolved by page-grained copy-on-write followed by "merging" when parallel isolated execution completes, if modules provide the libraries with the bounds on their concurrent accesses). Modern memory mapping hardware can also allows versions to be constructed, and exposed only as a an isolated module completes its execution.

Conclusions

We must re-engineer hardware and software systems from the ground up with a much better set of tools for "designing in a parallel environment" – in particular, with tools that do not try to force the system to appear "sequential" in order to apply poorly modular design concepts such as "cooperating sequential processes", "log servers", "locks", "synchronization primitives", "database cursors" etc. Ideally, we would also begin revising Programming 101 around naturally parallel algorithms on virtual parallel machines. This evolutionary step exposes opportunities for substantial system performance improvements as well, because we can focus on scaling parallel applications.

Bibliography

[Minsky67] M.L. Minsky. Computation: Finite and Infinite Machines. Prentice-Hall, 1967.

[Gazzaniga11] M. Gazzaniga. Who's in charge? Free Will and the Science of the Brain. Harper Collins, 2011.

[Dijkstra68a] E. Dijkstra. Go to statement considered harmful. CACM 11, 3 (March 1968) 147-148.

[Dijkstra68b] E.Dijkstra. The structure of the “THE” Multiprogramming system. *CACM* 11,5 (May , 1968), 341-346.

[Parnas72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *CACM* 15, 12 (December 1972), 1053-1058.

[Lamport78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM* 21, 7 (July 1978), 558-565.

[SRLock76] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. 1976. The notions of consistency and predicate locks in a database system. *CACM* 19, 11 (November 1976), 624-633.

[Reed78] D.P. Reed. Naming and Synchronization in a Decentralized Computer System. MIT LCS TR-205, 1978.

[Reed83] David P. Reed. Implementing atomic actions on decentralized data. *ACM TOCS* 1, 1 (February 1983), 3-23.

[Multik09] A.Baumann, et al. The multikernel: a new OS architecture for scalable multicore systems. *ACM SOSP '09*

[MarJonSin09] S.Marlow, S.Jones, S.Singh. Runtime support for multicore Haskell. *SIGPLAN Not.* 44, 9 (August 2009), 65-78