Juli 02, 2007

The Pillars of Concurrency

Building a consistent mental model for reasoning about concurrency

(Page 1 of 5)

Email Print
Discuss Reprint
add to:
Del.icio.us Slashdot
Digg Y! MyWeb
Google Blink
Spurl Furl

TABLE OF CONTENTS

Herb Sutter

In his inaugural column, Herb makes the case that we must build a consistent mental model before talking about concurrency.

Herb is a software architect at Microsoft and chair of the ISO C++ Standards committee. He can be contacted at www.gotw.ca.

Quick: What does "concurrency" mean to you? To your colleagues? To your project?

You probably know the fable of the blind men and the elephant: Several blind men explore an elephant by feel, and each one draws his own conclusion about what the elephant is. The man at the tail concludes that the elephant is like a rope; the one at the tusk asserts that, no, the elephant is like a spear; the one at the leg thinks it's like a tree; the one at the ear that it's like a fan; and so on. The blind men argue, each one knowing he is "right" from his own experience and point of view. In time, several give up hope of ever agreeing on what the creature is, and some resign themselves to accepting that the best they can do is deeply analyze each disconnected part in isolation as its own separate self. Meanwhile, the poor elephant waits patiently, faintly puzzled, because of course, it is not like any of those things alone and it doesn't see itself as that hard to understand as a whole creature.

Have you ever talked with another developer about concurrency, and felt as though you were somehow speaking completely different languages? If so, you're not alone. You can see the confusion in our vocabulary (and this is not an exhaustive list):

acquire, and-parallel, associative, atomic, background, cancel, consistent, data-driven, dialogue, dismiss, fairness, fine-grained, fork-join, hierarchical, interactive, invariant, isolation, message, nested, overhead, performance, priority, protocol, read, reduction, release, structured, repeatable, responsiveness, scalable, schedule, serializable update, side effect, systolic, timeout, transaction, throughput, virtual, wait, write,...

Some of the words have many meanings (performance, for instance), while others are unrelated (responsiveness and throughput, for example). Much of the confusion we may encounter arises when people inadvertently talk past each other by using incompatible words.

A basic problem is that this shouldn't be a single list. But how can we group these terms sensibly, when experienced parallel programmers know scores or hundreds of different concurrency requirements and techniques that seem to defy grouping?

1 The Pillars of Concurrency | 2 Callahan's Pillars: Pillar 1 | 3 Pillar 2: Throughput and Scalability Via Concurrent Collections | 4 Pillar 3: Consistency Via Safely Shared Resources | 5 Composability: More Than The Sum of the Parts Next Page ▶

Juli 02, 2007

The Pillars of Concurrency

(Page 2 of 5)

Email Print
Discuss Reprint
add to:
Del.icio.us Slashdot
Digg Y! MyWeb
Google Blink
Spurl Furl

TABLE OF CONTENTS

Callahan's Pillars

My colleague David Callahan leads a team within Visual Studio that is working on programming models for concurrency. He has pointed out that fundamental concurrency requirements and techniques fall into three basic categories, or pillars. They are summarized in Table 1 [1]. Understanding these pillars gives us a framework for reasoning clearly about all aspects of concurrency—from the concurrency requirements and tradeoffs that matter to our current project, to why specific design patterns and implementation techniques are applicable to getting specific results and how they are liable to interact, and even to evaluating how future tools and technologies will fit with our needs.

Let's consider an overview of each pillar in turn, note why techniques in different pillars compose well, and see how this framework helps to clarify our vocabulary.

Table 1: The Pillars of Concurrency

Pillar	**1. Responsiveness and Isolation Via Asynchronous Agents**	**2. Throughput and ScalabilityVia Concurrent Collections**	**3. Consistency Via Safely Shared Resources**
Tagline	Less coupling/blocking	Re-enable the free lunch	Don't corrupt shared state
Summary	Stay responsive by running tasks independently and tasks asynchronously, communicating via messages	Use more cores to get theanswer faster by running operations on groupsof things; exploit parallelismin data and algorithmstructures	Avoid races by synchronizing access to shared resources,especially mutable objectsin shared memory
Examples	GUIs, web services, background print/compile	Trees, quicksort,compilation	Mutable shared objects in memory, database tables
Key metrics	Responsiveness	Throughput, scalability	Race-free, deadlock-free
Requirements	Isolated, independent	Low overhead	Composable, serializable
Today's Mainstream Tools	Threads, thread pools, message queues, futures	Thread pools, futures, OpenMP	Explicit locks, lock-free libraries,transactions
Tomorrow's Tools?	Active objects/services, channels, contracts	Chores, work stealing,parallel STL & LINQ	Transactional memory,improved locking
Sample Vocabulary	background, cancel, dismiss, dialogue, fairness, fork-join, interactive, isolation, message, overhead, performance, priority,protocol, responsiveness, schedule, timeout, wait	and-parallel, associative,data-driven, fine-grained,fork-join, hierarchical, nested,overhead, performance, reduction,repeatable, scalable, schedule,serializable, side effect, structured, systolic, throughput	acquire, atomic, consistent, fairness, invariant, isolation,nested, overhead,performance, priority, read, release, serializable, transactionupdate, virtual, write

Pillar 1: Responsiveness and Isolation Via Asynchronous Agents

Pillar 1 is all about running separate tasks, or agents, independently and letting them communicate via asynchronous messages. We particularly want to avoid blocking, especially on user-facing and other time-sensitive threads, by running expensive work asynchronously. Also, isolating separable tasks makes them easier to test separately and then deploy into various parallel contexts with confidence. Here we use key terms like "interactive" and "responsive" and "background"; "message" and "dialogue"; and "timeout" and "cancel."

A typical Pillar 1 technique is to move expensive work off an interactive application's main GUI pump thread. We never want to freeze our display for seconds or longer; users should still be able to keep clicking away and interacting with a responsive GUI while the hard work churns away in the background. It's okay for users to experience a change in the application while the work is being performed (for example, some buttons or menu items might be disabled, or an animated icon or progress bar might indicate status of the background work), but they should never experience a "white screen of death"—a GUI thread that stops responding to basic messages like "repaint" for a while because the new messages pile up behind one that's taking a long time to process synchronously. Typically, users keep trying to click on things anyway to wake the application up, and finally either give up and kill the application because they think it crashed (possibly losing or corrupting work, even though the program would eventually have started responding again!) or else wait it out only to experience a flood where all the clicks they tried to perform finally get processed, usually with unsatisfactory or wrong results. Don't let this happen to your application, even if big companies let it happen to theirs.

So what kind of work do we want to ship out of responsiveness-sensitive threads? It can be work that performs an expensive or high-latency computation (background compilation or print rendering, for instance) or actual blocking (idle waiting for a lock, a database result, or a web service reply). Some of these tasks merely want to return a value; others will interact more to provide intermediate results or accept additional input as they make progress on their work.

Finally, how should the independent tasks communicate? A key is to have the communication itself be asynchronous, preferably using asynchronous messages where possible because messages are nearly always preferable to sharing objects in memory (which is Pillar 3's territory). In the case of a GUI thread, this is an easy fit because GUIs already use message-based event-driven models.

Today, we typically express Pillar 1 by running the background work on its own thread or as a work item on a thread pool; the foreground task that wants to stay responsive is typically long-running and is usually a thread; and communication happens through message queues and message-like abstractions like futures (Java *Future*, .NET *IAsyncResult*). In coming years, we'll get new tools and abstractions in this pillar, where potential candidates include active objects/services (objects that conceptually run on their own thread, and calling a method is an asynchronous message); channels of communication between two or more tasks; and contracts that let us explicitly express, enforce, and validate the expected order of messages.

This pillar is not about keeping hundreds of cores busy; that job belongs to Pillar 2. Pillar 1 is all about responsiveness, asynchrony, and independence; but it may keep some number of cores busy purely as a side effect, because it still expresses work that can be done independently, and therefore, in parallel.

◀ Previous Page | 1 The Pillars of Concurrency | 2 Callahan's Pillars: Pillar 1 | 3 Pillar 2: Throughput and Scalability Via Concurrent Collections | 4 Pillar 3: Consistency Via Safely Shared Resources | 5 Composability: More Than The Sum of the Parts Next Page ▶

Architecture & Design

Juli 02, 2007

Email Print
Discuss Reprint
add to:
Del.icio.us Slashdot
Digg Y! MyWeb
Google Blink
Spurl Furl

The Pillars of Concurrency

(Page 3 of 5)

TABLE OF CONTENTS

Pillar 2: Throughput and Scalability Via Concurrent Collections

Pillar 2, on the other hand, is about keeping hundreds of cores busy to compute results faster, thereby re-enabling the "free lunch"[2]. We particularly want to target operations performed on collections (any group of things, not just containers) and exploit parallelism in data and algorithm structures. Here, we use key terms like "scalability" and "throughput"; "data-driven" and "fine-grained" and "schedule"; and "side effect" and "reduction."

New hardware no longer delivers the "free lunch" of automatically running single-threaded code faster to the degree it did historically. Instead, it provides increasing capacity to run more tasks concurrently on more CPU cores and hardware threads. How can we write applications that will regain the free lunch, that we can ship today and know they will naturally execute faster on future machines having ever greater parallelism.

The key to scalability is not to divide the computation-intensive work across some fixed number of explicit threads hard-coded into the structure of the application (for instance, when a game might try to divide its computation work among a physics thread, a rendering thread, and an everything-else thread). As we'll see next month, that path leads to an application that prefers to run on some constant number *K* of cores, which can penalize a system with fewer than *K* cores and doesn't scale on a system with more than *K* cores. That's fine if you're targeting a known fixed hardware configuration, like a particular game console whose architecture isn't going to change until the next console generation, but it isn't scalable to hardware that supports greater parallelism.

Rather, the key to scalability is to express lots of latent concurrency in the program that scales to match its inputs (number of messages, size of data). We do this in two main ways.

The first way is to use libraries and abstractions that let you say what you want to do rather than specifically how to do it. Today, we may use tools like OpenMP to ask to execute a loop's iterations in parallel and let the runtime system decide how finely to subdivide the work to fit the number of cores available. Tomorrow, tools like parallel STL and parallel LINQ [5] will let us express queries like "select the names of all undergraduate students sorted by grade" that can be executed in parallel against an in-memory container as easily as they are routinely executed in parallel by a SQL database server.

The second way, is to explicitly express work that can be done in parallel. Today, we can do this by explicitly running work items on a thread pool (for instance, using Java *ThreadPoolExecutor* or .NET *BackgroundWorker*). Just remember that there is overhead to moving the work over to a pool, so the onus is on us to make sure the work is big enough to make that worthwhile. For example, we might implement a recursive algorithm like quicksort to at each step sort the left and right subranges in parallel if the subranges are large enough, or serially if they are small. Future runtime systems based on work stealing will make this style even easier by letting us simply express all possible parallelism without worrying if it's big enough, and rely on the runtime system to dynamically decide not to actually perform the work in parallel if it isn't worth it on a given user's machine (or with the load of other work in the system at a given time), with an acceptably low cost for the unrealized parallelism (for example, if the system decides to run it serially, we would want the performance penalty compared to if we had just written the recursive call purely serially in the first place to be similar to the overhead of calling an empty function).

Previous Page | 1 The Pillars of Concurrency | 2 Callahan's Pillars: Pillar 1 | 3 Pillar 2: Throughput and Scalability Via Concurrent Collections | 4 Pillar 3: Consistency Via Safely Shared Resources | 5 Composability: More Than The Sum of the Parts Next Page

Architecture & Design

Juli 02, 2007

Email Print
Discuss Reprint
add to:
Del.icio.us Slashdot
Digg Y! MyWeb
Google Blink
Spurl Furl

The Pillars of Concurrency

(Page 4 of 5)

TABLE OF CONTENTS

Pillar 3: Consistency Via Safely Shared Resources

Pillar 3 is all about dealing with shared resources, especially shared memory state, without either corruption or deadlock. Here we use key terms like acquire and release; read and write; and atomic and consistent and transaction. In these columns, I'll mostly focus on dealing with mutable objects in shared memory.

Today's status quo for synchronizing access to mutable shared objects is locks. Locks are known to be inadequate (see [3] and [4]), but they are nevertheless the best general-purpose tools we have. Some frameworks provide selected lock-free data structures (hash tables) that are internally synchronized using atomic variables so that they can be used safely without taking locks either internally inside the data structure implementation or externally in your calling code; these are useful, but they are not a way to avoid locking in general because they are few and many common data structures have no known lock-free implementations at all.

In the future, we can look forward to improved support for locks (for example, being able to express lock levels/hierarchies in a portable way, and what data is protected by what lock) and probably transactional memory (where the idea is to automatically version memory, so that the programmer can just write "begin transaction; do work; end transaction" like we do with databases and let the system handle synchronization and contention automatically). Until we have those, though, learn to love locks.

Previous Page | 1 The Pillars of Concurrency | 2 Callahan's Pillars: Pillar 1 | 3 Pillar 2: Throughput and Scalability Via Concurrent Collections | 4 Pillar 3: Consistency Via Safely Shared Resources | 5 Composability: More Than The Sum of the Parts Next Page

Juli 02, 2007

The Pillars of Concurrency

(Page 5 of 5)

Email Print
Discuss Reprint
add to:
Del.icio.us Slashdot
Digg Y! MyWeb
Google Blink
Spurl Furl

TABLE OF CONTENTS

Composability: More Than The Sum of the Parts

Because the pillars address independent issues, they also compose well, so that a given technique or pattern can apply elements from more than one category.

For example, an application can move an expensive tree traversal from the main GUI thread to run in the background to keep the GUI free to pump new messages (responsiveness, Pillar 1), while the tree traversal task itself can internally exploit the parallelism in the tree to traverse it in parallel and compute the result faster (throughput, Pillar 2). The two techniques are independent of each other and target different goals using different patterns and techniques, but can be used effectively together: The user has an application that is responsive no matter how long the computation takes on a less-powerful machine; he also has a scalable application that runs faster on more powerful hardware.

Conversely, you can use this framework as a tool to decompose concurrency tools, requirements, and techniques into their fundamental parts. By better understanding the parts and how they relate, we can get a more accurate understanding of exactly what the whole is trying to achieve and evaluate whether it makes sense, whether it's a good approach, or how it can be improved by changing one of the fundamental pieces while leaving the others intact.

Summary

Have a consistent mental model for reasoning about concurrency—including requirements, tradeoffs, patterns, techniques, and technologies both current and future. Distinguish among the goals of responsiveness (by doing work asynchronously), throughput (by minimizing time to solution), and consistency (by avoiding corruption due to races and deadlocks).

In future columns, I'll dig into various specific aspects of these three pillars. Next month, we'll answer the question, "how much concurrency does your application have or need?" and distinguish between *O(1)*, *O(K)*, and *O(N)* concurrency. Stay tuned.

Notes

[1] The elephant analogy and the pillar segmentation were created by David Callahan (www.microsoft.com/presspass/exec/de/Callahan/default.mspx) in an unpublished work.

[2] H. Sutter. "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software" (www.ddj.com/dept/architect/184405990).

[3] H. Sutter. "The Trouble With Locks" (www.ddj.com/dept/cpp/184401930)

[4] H. Sutter and J. Larus. "Software and the Concurrency Revolution" (*ACM Queue*, September 2005). (gotw.ca/publications/concurrency-acm.htm).

[5] J. Duffy, http://www.bluebytesoftware.com/ blog/PermaLink,guid,81ca9c00-b43e-4860-b96b-4fd2bd735c9f.aspx.

Previous Page | 1 The Pillars of Concurrency | 2 Callahan's Pillars: Pillar 1 | 3 Pillar 2: Throughput and Scalability Via Concurrent Collections | 4 Pillar 3: Consistency Via Safely Shared Resources | 5 Composability: More Than The Sum of the Parts