Dr. Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

Parallel Microsoft-Style

The actor model of concurrency is gaining favor in Java but remains largely ignored by Microsoft

July 20, 2011
URL:http://www.drdobbs.com/architecture-and-design/parallel-microsoft-style/231500268

One area in which Microsoft operating systems have long been on the vanguard is multithreading APIs. Even before multicore processors were the norm, Microsoft offered expansive APIs for managing threads and thread-like execution streams called fibers. The Win32 platform had a wide selection of thread-management functions that could be used with considerable ease. While I can't speak for every operating system, that collection of APIs was considerably greater and more sophisticated than UNIX or Linux counterparts. Pthreads, the now-standard threading implementation on Linux, is a much smaller solution. However, Pthreads does provide a benefit singularly lacking in the Windows implementation: portability. Pthreads originated on UNIX where it still runs on the various variants. A good port on Win32 exists, in addition to the multiple Linux distributions.

The Microsoft APIs, however, are very much drawn from the traditional approach to threading, which is a complex model of creating threads, managing them individually, and using mutual exclusion to prevent them from interfering with each other. Even though Microsoft's Visual Studio IDE has decent support for working with threads, developers can expect to spend a lot of time in the debugger. In this shared-memory model, the two great debugging challenges are replicating the problem, and once it's replicated, diagnosing it.

The issue of replication has driven more than one developer to the brink of insanity, if not pushed him over the edge entirely. Some errors show up only when certain threads interact in a rare way, with each one handling a specific piece of data. This might show up as a defect report indicating that, occasionally, transactions aren't logged correctly to the database. Trying to recreate what conditions might lead to an event transiently not occurring — when dozens of threads are in various states of operation — can be exceedingly difficult, if not impossible. But even if the situation can be faithfully reproduced, trapping the condition and determining how the threads are crossing each other incorrectly is arduous. In sum, the shared-memory model is no one's idea of programming fun. And as a result, much software that should have been parallelized long ago runs serially because the job is constantly deferred until parallelization is the *only* way to gain the needed performance.

Various attempts have been made to remove this complexity and thereby facilitate parallel development. One of the strongest pushes has come from Intel, which spearheaded OpenMP a decade ago. OpenMP sought to add keywords to the C (via pragmas) and Fortran, so that portions of existing programs could be parallelized. This approach, while limited to only sections of the program, worked well. It was easy to grasp and it cleverly handled the messy thread-management issues behind the scenes. If the code ran correctly on a single thread, chances were excellent that it would still run correctly when parallelized with OpenMP keywords. Microsoft's support of OpenMP made it a natural choice for many applications.

More recently, Intel has come out with a more elaborate solution, Cilk Plus, which follows the OpenMP model and leverages Intel's deep knowledge of threading (acquired from a decade of investment in threading tools). James Reinders of Intel provided an excellent technical overview of the problem and solution as viewed from within the industry in a recent blog post.

Java and the Actor Alternative

Interestingly, the languages on the JVM are heading towards a different kind of solution, namely actors. If you're not familiar with actors, it's easiest to think of them as functions to which discrete tasks can be sent for execution. The most common operational metaphor is a thread that has an incoming mailbox to which messages are sent by other actors. The actor processes these messages (which can include tasks to execute or data to operate on) and the output is sent to another actor for downstream work.

Actors avoid the dangers of the shared-memory model because they touch only the data that's sent to them in messages. If they need some external data, they request it from other actors (by sending the request in the form of a message). In addition, the data actors receive is immutable. They don't change the data, rather they copy it and transform the copy. In this way, no two threads are ever contending for access to the same data item, nor are their internal operations interfering with one another. As a result, the nightmares I described earlier disappear almost completely.

The actor model has been used in applications that naturally align with message passing: telephone switches, Web servers, and the like. Increasingly, however, JVM languages are using them for run-of-the-mill applications. Obviously, actors require a different kind of architecture (in most actor applications, everything is done by actors — mixed-model applications are generally not favored.). And in support of this, they are finding support in several recent languages. For example, Scala and Fantom have built in-actor libraries (Scala's enjoys native syntactic support). Groovy has a widely used actor library. And actor frameworks, such as Akka and Killim, that can be used by any JVM language have recently come to market.

For all the extensive threading APIs Microsoft offers, the company has not substantially embraced actors. A good start was made in 2010 when the Redmond giant released Asynchronous Agents Library, but this is only a partial actor implementation and it runs only in C++. (Note: A highly portable, widely used C++ library with actor building blocks — but no actor framework — is the ACE library.)

Microsoft has released a pair of packages that contain many actor primitives. These packages, named the Concurrency and Coordination Runtime (CCR) and the Decentralized Software Services (DSS), were released in 2008 as part of the Microsoft Robotics toolkit, curiously enough. The packages have languished there ever since and are not part of the .NET framework. However, some posts on Microsoft's website suggest that, at some future point, the packages will be migrated to .NET. The timing is left ambiguous and it's not clear whether the APIs will be changed in the migration, nor whether a full actor framework will be constructed from them.

Another concern about CCR and DSS is that they have not been updated for more than a year, although in the forums, a project lead from Microsoft refers at several points to an "upcoming" announcement.

In either case, there is no long history of working with this technology — nor any evidence that I can find of any other support for actors in the .NET languages. Microsoft is still primarily oriented towards the traditional models of parallel programming. The benefit of this approach is raw performance. For the same reasons, Intel's libraries target only C, C++, and Fortran — three languages that are associated with high-speed computation and frequently run as native code.

Application developers who are willing to trade a little performance for ease in writing parallel applications are likely to find Java's embrace of actors an appealing solution (although Java certainly has the traditional primitives for developers who want to follow that path.)

I expect that over time, Microsoft will increase its support for the actor programming model. But if it comes, the support will likely not be fully embraced by Redmond until pressure for large-scale actor models is applied either by its customers or competition in the market place. In the latter case, the pressure will be exerted primarily by JVM-based solutions.

— Andrew Binstock
Editor in Chief
alb@drdobbs.com
Twitter: platypusguy