Concurrency in Modern Programming Languages

In the last 10 years, CPU clock speeds haven't increased significantly, but the computing power of CPUs has continued to grow according to Moore's law. Researchers achieved this by increasing the number of computing cores per chip. This process has made parallel computing relevant to a much broader community.

Writing computer programs that take advantage of multicores is a challenge because of the diversity of architectures and because of the inherent difficulty of writing parallel programs. In today's multicore world, the means to express parallelism and the effectiveness in achieving parallel performance are increasingly important.

Traditionally, there have been two approaches to parallel programming: threads that communicate via shared memory and processes that

1521-9615/12/$31.00 © 2012 IEEE
Copublished by the IEEE CS and the AIP

Massimo Di Pierro
DePaul University
David Skinner
Lawrence Berkeley Laboratory

communicate via message passing. Modern programming languages introduce different approaches, some of which are discussed in this special issue of *CiSE*.

About the Articles

In "Clojure for Number Crunching on Multicore Machines," Martin Kalin and David Miller describe the Clojure language, which was first released to the public by Rich Hickey in 2007. Clojure—like Lisp, Erlang, and Haskell—is a functional language. In functional languages, the output of any function depends only on the input, and there's no global state. This approach alone should make functional languages appealing to scientists. The input and output consist of immutable data structures that can safely be accessed concurrently without the need for locking.

If a problem is decomposable, the functions that constitute the intermediate steps to the solution can be executed in parallel, and the underlying virtual machine (VM) can automatically allocate them to different cores. Yet not everything is always so simple and some objects must be mutable. Different languages deal with mutability in different ways.

Clojure uses transactional memory. With some limitations, if two functions reference the same mutable object and try to modify it in parallel (think about incrementing a counter), the conflict is detected, and effectively the access to the object is serialized without need for explicit locking. Clojure runs on the Java VM and can access all of the Java APIs.

Steve Vinoski's article "Concurrency and Message Passing in Erlang" describes Erlang and the Open Telecom Platform (OTP) released by Erlang in 1996. Erlang is a functional language originally developed with the goal of making software development more reliable, easier, and less expensive.

In Erlang, applications consist of many lightweight processes that are managed by a scheduler in the VM and dispatched to the available cores automatically. These lightweight processes have no overhead, and one VM can run millions of them. They can only communicate via message passing, thus avoiding the need for explicit locking.

Duncan Coutts and Andres Löh's article "Deterministic Parallel Programming with Haskell" describes the Haskell language. Haskell is a modern functional programming language that, instead of using concurrent threads, provides a library (Repa) to allow writing concise high-level parallel programs that are guaranteed to be deterministic.

Haskell doesn't use a VM, but it compiles to native code. Like Erlang, it supports lightweight threads, but it doesn't use explicit message passing. Instead, it determines data dependencies from the data structures. For example, a Haskell library called Repa defines the notion of a parallel array. Parallel array operations are automatically parallelized into as many chunks as there are CPU cores.

Coutts and Löh also mention ongoing development efforts to let Haskell support grids, GPUs, and CPU single-instruction, multiple-data (SIMD) instructions. The goal is to allow different parallel programming models within a single programming language.

A Parallel Assignment

To better compare Clojure, Erlang, and Haskell, we assigned the authors the same problem: parallelize a naive solver for a 1D Poisson equation:

$$\nabla^2\phi(x) = \rho(x).$$

Once discretized ($x \to x_i = x_0 + h_i$, $\phi(x) \to \phi_i$, $\rho(x) \to \rho_i$) this equation turns into an iterative expression:

$$\forall i \quad \phi_i \leftarrow \frac{\phi_{i-1} + \phi_{i+1}}{2} - \frac{h^2}{2}\rho_i,$$

where ρ and ϕ can be thought of as input and output, respectively. Here, h is the discretization step. The expression must be iterated until convergence of the array ϕ. Each iteration can be parallelized in the integer variable i.

There are three issues of importance here: ease of implementation of the parallel approach; parallel scaling with the number of cores; and overall language speed. We leave the discussion of the first two issues to the authors.

A truly unbiased speed comparison is impossible, and real-life performance is problem-specific. Still yet, we refer the reader to *The Computer Language Benchmarks Game* (http://shootout.alioth.debian.org), from which we report benchmarks (see Table 1).

The first column of Table 1 shows the average running time of a battery of benchmarks on a single core of a Q6600 Intel CPU normalized to C++ running time. This measures the language speed but not its scaling. The second column shows the results of a specific benchmark, called *thread-ring*, on four cores. Thread-ring creates 503 threads and passes a token around them in a ring, while performing no computation. This measures the language overhead in dealing with concurrency. In both cases, smaller numbers indicate better performance.

Table 1. Computer language benchmarks.

Language	One core (average)	Four cores (thread-ring)
C++ (GNU C++)	1.00	1.00
Java	1.49	1.86
Haskell (Glasgow Haskell Compiler)	1.58	0.05
Clojure	3.31	0.51
Erlang (High-Performance Erlang)	7.83	0.19

These languages aren't for everyone, and we don't expect the readers to immediately switch to using them. Yet their expressiveness and their ability to handle concurrency in a simpler way will be appealing to some. Moreover, they might provide an indication as to the future of programming languages.

In fact, some of the ideas presented in these articles aren't unique to functional languages. An example is the Go language (http://golang.org) proposed by Google as a possible C replacement. As with Erlang and Haskell, Go implements lightweight "goroutines" that communicate via buffered channels, thus providing a low-level and efficient message passing interface to concurrent tasks. Another example is the D language (http://dlang.org), which is a systems programming language similar to C++. D's approach to concurrency is also to use isolated threads or processes that communicate via messages, although it also supports old-style synchronization of critical sections using explicit mutex locks.

The proliferation of new languages and paradigms can at first appear confusing, but it shows a clear trend toward increasing the expressiveness and readability of languages while decoupling the coding of the algorithm from parallelization and concurrency optimizations.

Massimo Di Pierro *is an associate professor at the School of Computing of Digital Media at DePaul University. His main research interests are in lattice quantum chromodynamics, quantitative risk management, and Web development. Massimo has a PhD in physics from the University of Southampton, United Kingdom. Contact him at mdipierro@cs.depaul.edu.*

David Skinner *currently leads the Scientific Discovery through Advanced Computing (SciDAC) Outreach Center at Lawrence Berkeley Laboratory, which provides information and services that support SciDAC's outreach, training, and research objectives. His research focuses on the performance analysis of computing architectures and HPC applications. Skinner has a PhD in theoretical chemistry from the University of California, Berkeley. Contact him at deskinner@lbl.gov.*