Concurrency and Message Passing in Erlang

Developers use the open source Erlang programming language in domains such as telecommunications, database systems, and the Web due to its superior support for concurrency and reliability. Erlang applications comprise numerous processes—lightweight user-space threads—that communicate via message passing. This article focuses on Erlang's concurrency support and details an example 1D Poisson solver program.

Erlang is a concurrent and distributed functional programming language for building practical systems requiring high levels of distribution, concurrency, fault tolerance, availability, and uptime. Developers use it in various domains—including finance, database systems, social networking systems, and the Web—but it was originally conceived for use in telecommunications applications. In the world of telecom, equipment uptime and reliability has always been a significant concern. It's not unusual for production telecom equipment to have allowable downtime of only 3 to 4 minutes per year, with governmental fines levied for any outages exceeding those limits.

In the 1980s, the Ericsson Computer Science Laboratory searched for ways to make the process of developing software for telecom equipment result in more reliable systems that were easier and less expensive to build and maintain. Under the direction of Bjarne Däcker, Joe Armstrong started experimenting in the mid-1980s with a wide variety of programming languages, including Smalltalk, ML, Ada, Prolog, and Ericsson's proprietary Programming Language for Exchanges (PLEX), leading him to build a series of prototypes of what would eventually become Erlang. Mike Williams and Robert Virding later joined Armstrong in helping to further develop the prototypes and evolve the language. Armstrong wrote his initial prototypes of the Erlang virtual machine (VM) in Prolog, but in the early 1990s, Williams rewrote the VM in C.

Erlang began to see commercial use in the early 1990s, and by the mid-1990s, Ericsson had begun to use it on several large-scale commercial projects, such as the successful AXD301 ATM switch. By 1996, Ericsson had also built the Open Telecom Platform (OTP) framework,[1] a layer of libraries and frameworks that enhanced Erlang's capabilities for use in soft real-time, highly concurrent, highly reliable systems. Ericsson released Erlang/OTP as open source in 1998, and the Erlang community has been growing ever since.

Today, the Ericsson Erlang/OTP development team, led by Kenneth Lundin, regularly produces high-quality releases of the Erlang system with the help of the open source community. You can download their releases of Erlang from www.erlang.org. Members of the Ericsson team also serve as the keepers of the language, deciding which features and extensions are added or deprecated. They develop Erlang using GitHub (https://github.com/erlang/otp), accepting modifications and patches from the Erlang community through GitHub's source code management facilities.

Here, I explain Erlang's history and features, focusing on its concurrency support, and detail an

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

Steve Vinoski
Basho Technologies

example concurrent 1D Poisson solver program written in Erlang.

A Tour of Erlang

Erlang is a straightforward language with comparatively few syntactical elements. Let's review those elements.

Syntax

Because early Erlang VM prototypes were built using Prolog, Erlang's syntax (as I'll show in later examples) bears a strong resemblance to it. Some developers whose programming experience is limited to Java, C++, or C initially have difficulty with Erlang's syntax because it differs considerably from what they're used to, but most become comfortable with the syntax after just the first few days of learning to read and write it.

Types

Erlang has just a few types as follows. When bound to a value, Erlang variables gain the type of the value being bound; the language doesn't require explicit type declarations for variables.

Integers and floating-point numbers. Erlang integers are of arbitrary size and can represent any whole number. Floating-point numbers are represented using the IEEE 754-1985 double-precision standard.

***Atoms, known as* non-numeric constants.** An atom usually starts with a lowercase letter, but can also be specified by putting single quotes around the atom name. So, for example, both `exit` and `'EXIT'` are atoms.

Lists and tuples are both collections of Erlang terms, where a term is an instance of any Erlang data type. Lists are delimited by square brackets, whereas tuples are delimited by curly braces. Each type can be heterogeneous, storing elements of differing types within the same collection. Tuples are typically used for fixed-size collections, while lists are better for variable-sized collections because—as I'll show in some example code later—Erlang provides special syntax and functions for easily creating lists of arbitrary length, for adding to or removing from the head of a list, and for accessing the heads and tails of lists.

Erlang doesn't have a specific string type; a string is just a list of characters. Strings can be defined using either double quotes or square brackets. The constructs "`string`" and `[$s,$t,$r,$i,$n,$g]`—where the "`$c`" syntax represents the ASCII value of the character `c` following the dollar sign—are equivalent.

Records are collections with named fields. Records are implemented under the covers as tuples. However, unlike tuples, record field values can be accessed and set by name.

Binaries are basically memory buffers. Fundamentally, they're contiguous blocks of raw bytes, but they allow memory to be set and accessed as integers, strings, floating-point numbers, characters, and arbitrary-length bit fields. Binaries are useful for representing network protocol headers, for example, because header structures tend to contain fields of less than a byte in size. Erlang's syntax for handling binaries is something to behold, requiring just a single line to extract numerous fields of a binary, where other languages might require several dozen lines to retrieve the value of each field separately. For example, the various fields comprising a TCP segment's header as well as its accompanying payload can be obtained from a binary variable in a single operation[2] as follows:

```
<<SourcePort:16, DestinationPort:16,
     SequenceNumber:32, AckNumber:32,
     DataOffset:4, _Reserved:4,
       Flags:8, WindowSize:16,
     Checksum:16, UrgentPointer:16,
       Payload/binary>> = BinaryVar.
```

Here, we assume `BinaryVar` stores a TCP segment. The `<<` and `>>` tokens delimit the definition of a binary on the left-hand side of the equal sign. Within the binary, each header field is declared with its bit length in standard order of appearance within a TCP segment. The assignment operator, as explained in more detail later, asserts a match of its left- and right-hand sides, and at runtime such matching causes the value on the right-hand side to be deconstructed into all the variables on the left-hand side. Following the header fields is the `Payload` variable, which is itself a binary, allowing it to capture all remaining bytes of the segment following the header fields, regardless of total segment length.

Anonymous functions, or* funs*, are functions without names. Anonymous functions can be passed as arguments to other functions, returned from functions, and stored in collections, as can references to named functions.

Process identifiers, or* pids*, serve as handles to Erlang's lightweight processes. A process holding a pid of another process can use it to send messages to the other process, monitor it, control it, or request runtime information about it.

Sequential Flow Control

Sequential flow control constructs in Erlang are similarly sparse. For example, unlike most languages, there are no "`for`" or "`while`" loops. Instead, looping is achieved through recursive function invocation. Erlang supports tail calls, where recursion occurs not by pushing new frames onto a stack—which could cause the application to run out of stack space—but rather by jumping from the function tail back to its start.

In addition to recursion, the use of pattern matching for sequential flow control is prevalent in Erlang programming. Erlang's pattern matching allows for the deconstruction of data structures, the assignment of multiple variables, and—when combined with Erlang's "`case`," "`if`," and "`receive`" statements and multiclause functions—the choice of code paths based on values. Treating assignment as matching in programming languages is unusual, given that most languages treat assignment as a way to replace a variable's current value. However, doing so in Erlang is impossible, because Erlang variables are single-assignment—once assigned, they can't be modified. Here's how assignment works:

```
Variable = some_value(),
Variable = foo.
```

On the first line, we call the function `some_value()`. Let's assume it returns the atom `foo`. In this assignment, `Variable` has no value yet—it's unbound—so the Erlang runtime binds the atom value `foo` to it. The second line, rather than reassigning `Variable`, asserts its equality to the atom `foo`, similar to an algebraic equation. If `Variable` had a value other than `foo`, the second assignment statement would throw a `badmatch` exception at runtime.

Compared to mainstream languages such as Java, single-assignment variables are unusual, but they help make reasoning about program flow easier, especially in highly concurrent systems. One thing to consider is that Erlang doesn't support global variables; all variables are defined within functions. In other programming languages, global variables are often sources of problems in multithreaded programs.

Here's a brief example showing some of Erlang's sequential programming features:

```
-module(list_ops).
-export([add/1]).

add(List) when is_list(List) ->
    add(List, 0).
add([Head|Tail], TotalSoFar) ->
    NewTotal = TotalSoFar + Head,
    add(Tail, NewTotal);
add([], Total) ->
    Total.
```

The first line defines the module's name, `list_ops`. On the second line, we export the module's functions that we want accessible outside the module—in this case, a single function, `add/1`, where the integer 1 refers to the function's *arity* or the number of arguments it expects. The other seven lines of code define three functions: the exported `add/1` function and two `add/2` functions that are local to the module. On the first of these lines, we define the `add/1` function head, including its `List` argument between the parentheses.

All variables in Erlang begin with an uppercase letter. A guard, which is an optional clause that, in this case, constrains the `List` argument to having the list type, follows the argument list. The arrow "`->`" separates the head of the function from its body. In the function body, we call the function `add/2`. Calling a function in a different module requires prefixing the function name with its module name, separated by a colon character, whereas calling a local function requires no prefix. The return value of the `add/1` function is the value of its final statement, so whatever `add/2` returns will be the return value of `add/1` as well.

The first clause of `add/2` uses pattern matching in its argument list. The first argument is specified as a list of at least one element named `Head`, followed by a possibly empty list named `Tail`. The vertical bar character separates the list's head from its tail. The `add/2` function adds `Head` to `TotalSoFar` to create `NewTotal`, then invokes itself recursively, passing `Tail` as the first argument and `NewTotal` as the second argument.

The second clause of `add/2`, which also uses pattern matching in its argument list, differs from the first clause in an important way: because its first argument is specified as the empty list, it provides a way for the recursion to end. When the list passed to `add/2` is empty, the function's second clause is chosen via pattern matching, and it simply returns its `Total` argument.

Developers with functional programming experience will recognize our example `list_ops:add/1` and `add/2` functions as an instance of a *fold*, in which a single value is computed by applying a function to all elements of a list. Erlang provides fold functionality in the form of the `foldl/3` and `foldr/3` functions, both found in the standard Erlang `lists` module. The three function calls shown here produce the same result:

```
List = [1,2,3,4,5],
15 = list_ops:add(List),
15 = lists:foldl(fun erlang:'+'/2,
     0, List),
15 = lists:sum(List).
```

The `lists:foldl/3` function takes three arguments: an arity 2 function to apply to each list element and accumulate the result, the accumulator's initial value, and the list on which to operate. This example shows one way of passing a function as an argument to another function: the function we pass as the first argument to `lists:foldl/3` is Erlang's built-in addition operator. In Erlang, all function names are atoms. The construct `'+'` therefore represents the atom that names the addition function; as with all of Erlang's built-in functions, it lives within the `erlang` module. The `fun` keyword marks the construct as a function reference.

On the final line of the example, we call `lists:sum/1`—another standard function that's less general than a fold, but equivalent in behavior to the `list_ops:add/1` function. All three statements use pattern-matching assignment to assert that each function call returns the expected value 15.

Erlang Concurrency

Other differences aside, what sets Erlang apart from most other languages is its support for concurrency. Erlang's concurrency model is built around the idea of lightweight processes available as part of the language, rather than being added on through extra optional libraries. Erlang's processes are lightweight enough that a single VM instance can execute millions of them concurrently.

Erlang processes communicate via message passing, rather than through shared memory. Just as with immutable variables, message passing lets developers more easily write, read, and reason about concurrent programs. Each process has a mailbox, or message queue, through which it receives messages. Erlang guarantees that messages sent from one process to another arrive in the order sent.

When it comes to executing processes, the Erlang VM's implementation is similar to an operating system (OS) in that it contains schedulers that map Erlang's processes onto underlying OS threads. By default, the Erlang VM starts one scheduler per CPU core, making Erlang an excellent fit for today's multicore systems. A scheduler tracks both the runtime execution and I/O of each Erlang process, allowing it to run for either a maximum number of operations or until it encounters blocking I/O, at which time it schedules that process out and runs another one that's ready. Should a particular scheduler run out of work, it can steal work from other schedulers.

Consider an example consisting of two Erlang processes. One process sends the atom `ping` to the other process, which in turn sends the atom `pong` back to the first process. First, the code for the `ping` process:

```
ping(0, _PongPid) ->
        ok;
ping(N, PongPid) ->
         PongPid ! ping,
         receive
                pong ->
                    ping(N-1, PongPid)
         end.
```

The `ping/2` function takes two arguments: a counter `N` and a process ID `PongPid` identifying the `pong` process. In the second clause, the function first uses the "`!`" operator to send the message `ping` to `PongPid`. The function then calls `receive`, a built-in function that blocks waiting for one or more specified messages. The only message `receive` waits for here is the atom `pong`; when that arrives, `ping` calls itself recursively, decrementing its `N` counter. Once `N` reaches zero, the first clause of the `ping` function matches, and the recursion ends, returning the atom `ok`.

The code for the `pong` process is similar:

```
pong(0, _PingPid) ->
          ok;
pong(N, PingPid) ->
          receive
                ping ->
                    PingPid ! pong,
                    pong(N-1, PingPid)
        end.
```

The primary difference here is that the `pong` process acts as a server, first receiving and then sending. It goes directly into its `receive` call, waiting for a `ping` message. Once that arrives, it sends the message `pong` to `PingPid`, and then decrements its counter and invokes itself recursively. The recursion ends when the counter hits zero.

Starting the ping and pong processes is straightforward:

```
start(N) ->
      PingPid = self(),
      PongPid = spawn(fun() ->
                      pong(N, PingPid)
                end),
      ping(N, PongPid).
```

The `start/1` function takes a single argument, `N`, the recursion counter. It sets the `PingPid` variable to `self()`, a built-in function that returns the process ID of the calling process. Thus, the process executing `start/1` is also the `ping` process. To start the `pong` process, the `start/1` function calls the `spawn` built-in function, telling it what to execute by passing it an anonymous function that invokes the `pong/2` function. Finally, `start/1` calls `ping/2` to kick off the ping-pong message exchange. Once `N` instances of both the `ping` and `pong` messages are exchanged, the `start/1` function ends, as does the `PingPid` process. The `PongPid` process exits once the `pong/2` function completes its recursions.

This example shows the simplicity with which developers can write concurrent applications using Erlang. Idiomatic Erlang code makes use of many processes; applications are typically composed of hundreds or even many thousands of communicating processes. The ease with which Erlang allows new processes to be started and coordinated is notable, especially given the lack of mutexes, locks, condition variables, and other special constructs and keywords that tend to litter multithreaded code written in other languages.

Thinking of applications in terms of concurrently active processes passing messages to each other can take some getting used to, because it's more akin to writing a distributed system than a sequential single-threaded program. Interestingly, Erlang also provides exceptional support for distributed programming: the code to pass messages between processes is the same regardless of the location of the processes. When processes are distributed, the Erlang runtime transparently sends any messages passed between them over the network.

Erlang's support for reliable systems is a by-product of its strong concurrency and distribution capabilities. Distribution is important for reliable systems, which employ multiple boxes that must communicate with and monitor each other so that an operational box can pick up the work of any that become unavailable due to hardware or software faults. Concurrency is also important for reliable systems; because Erlang can create and destroy processes so cheaply, it encourages programmers to follow a "let it crash" philosophy with their code—whereby if a process executing a function encounters an unhandled error, it crashes. This unique approach to error handling lets programmers write only the code they expect to work for the non-error case and leave any cleanup for error cases to the application's supervision scaffolding.

In OTP applications, processes are divided into supervisors and workers, with supervisors monitoring the workers. If a worker dies, a supervisor can either start a replacement worker or relay the problem to another higher-level supervisor to let it take corrective action. This approach saves Erlang developers from having to write the numerous lines of exception-handling code typically found in other languages, and less code means smaller applications with fewer bugs.

The Erlang runtime also uses processes to allow for live system upgrades. The runtime lets two versions of a module be present in memory simultaneously. Processes currently executing functions within the old version are allowed to finish, while all new calls to the module are directed to the new version, including fully qualified recursive calls. The runtime also executes any custom upgrade functions that an upgrade package specifies, allowing for the modification or update of any state variables passing from a function in the old module to the new one via a recursive call.

A 1D Poisson Solver

The design of my Erlang solver for the 1D Poisson equation is similar to that of the original Python solver discussed in the guest editors' introduction. As Figure 1 shows, my solution employs several concurrent Erlang processes acting as solvers for individual Poisson lattice sites. These processes cooperate by exchanging numeric values with adjacent processes as needed to calculate a solution for each lattice site.

First, the Erlang solver declares the module name along with the functions it exports:

```
-module(poisson).

%% API
-export([poisson1d/0]).

%% Internal exports
-export([phi/4]).
```

The first `-export` directive defines the module's API, which is the list of functions it exports for other modules to use. Here, the module exports just a single function, `poisson1d/0`. The second `-export` line semantically behaves just like the first, but by convention I separate it from the first to indicate that the function it exports isn't intended to be part of the module API. Rather, the function it specifies, `phi/4`, is exported for use in an Erlang `spawn` call, where it's referenced only as an atom and not as a function. Ordinarily, the Erlang compiler optimizes away any local function that appears unused, but exporting the function prevents that.

The first part of the `poisson1d` API function defines several constant values:

```
poisson1d() ->
    NSites = 16,
    H = 0.1,
    NIters = 10,
    Collector = self(),
```

As noted earlier, in Erlang all variables begin with an uppercase letter. The `NSites`, `H`, and `NIters` variables correspond directly to their counterparts in the original Python solver. Just as in the Python code, `NSites` is the number of sites on the Poisson lattice, `H` is the lattice spacing, and `NIters` is the number of solver iterations. The `Collector` variable is initialized from the Erlang `self()` function, which returns the current process's identifier. The job of the `Collector` is to eventually collect all the solution results that the concurrent solver processes produce.

Next, we create a list of concurrent solvers:

```
Sentinel = spawn(fun sentinel/0),
Pids = lists:foldl(fun(I,
  [Prev|_]=Acc) ->
    Rho = case NSites div 2 of
                        I -> 1;
                       _ -> 0
          end,
```

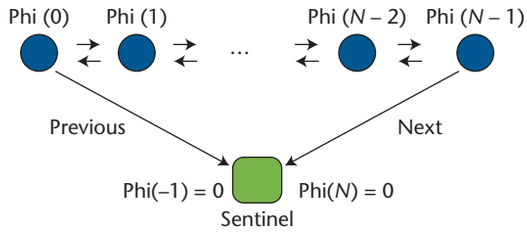

Figure 1. Each process in the lattice is aware of its previous and next processes. The processes at the ends of the lattice refer to the `Sentinel` process, which polymorphically handles lattice boundary conditions.

```
    Args = [Collector, NIters, Prev,
                {I, H, Rho}],
  Pid = spawn(?MODULE, phi, Args),
  Prev ! {set_next, Pid},
  [Pid | Acc]
end, [Sentinel], lists:seq(0,
  NSites-1)),
```

Here, we first spawn a process to execute the `sentinel/0` function, then store its process ID in the `Sentinel` variable. I'll explain the implementation of the `sentinel/0` function later.

The rest of this part of the code creates one solver process for each lattice site. It does this by folding a function over a list of integers ranging from zero to one less than the `NSites` constant. Each integer serves as an identifier for a lattice site. The `lists:foldl/3` function operates by passing each integer in turn as the first argument to the fold function. The second argument to the fold function is the fold accumulator, which essentially stores the fold's state as it advances through the list from one integer to the next.

The fold function performs several activities for each lattice site. First, it trivially calculates `Rho` for each site. For the site whose integer identifier corresponds to half the number of total sites, `Rho` is 1; otherwise it's 0. The `case` statement handles the calculation of `Rho`. Next, the fold function spawns a new solver process, to which it passes the following arguments:

- the `Collector` variable, indicating the process ID of the solution-collector process;
- the `NIters` variable, indicating the number of iterations each solver must perform;
- the `Prev` variable, indicating the process ID of the solver process created in the previous fold iteration; and
- a tuple consisting of the integer `I` for this solver, the constant value `H`, and the `Rho` value for this solver.

Notice the declaration of the `Prev` variable: it's the head of the accumulator list, which is the second argument to the fold function. The Erlang syntax `[Prev|_]` represents a list where `Prev` is the list's head and the "don't care" variable, represented as an underscore, is its tail. The full syntax of the second argument, `[Prev|_]=Acc`, is a way of asserting that the `Acc` argument is a list. The equal sign here indicates equivalence, not assignment. The construct asserts that `Acc` is a list comprising at least one element, `Prev`, while also naming the list's first element for convenient use within the function body. This argument-matching construct involving the definition of multiple variables is quite common in idiomatic Erlang code.

As in the original Python solution, my Erlang approach requires each solver to know about its adjacent solvers. Within the fold function, we inform each `Prev` solver process—the one created in the fold function's previous iteration—of the process ID of its next solver process by sending the `Prev` process a message:

```
Prev ! {set_next, Pid},
```

The message this statement sends is a tuple composed of the atom `set_next` and the `Pid` variable, where `Pid` is the pid for the newly created solver. Later, I'll explain the details of how the receiving `Prev` process acts on this message.

The final line of the fold function builds a new value for the fold accumulator by prepending the `Pid` process ID to the current accumulator's head:

```
[Pid | Acc].
```

This is Erlang's special syntax for list construction, and it's highly efficient, because the runtime just stores the new value in the list's head and links the tail to it. This new list serves as the return value of the fold function, which `lists:fold/3` carries forward as the accumulator's new value for the next iteration of the fold. When the fold completes, `lists:fold/3` returns the accumulator's final value as its return value. Interestingly, prepending new pids to the accumulator means the accumulator is built in reverse order, with lattice site 0 toward the right end of the list and lattice site `NSites-1` at the left end. I address the list order later in the code.

The second argument to `lists:fold/3` is the accumulator's initial value, which for this application is a single-element list consisting of the `Sentinel`'s process ID. The `Sentinel` is a special process that handles the lattice boundaries. It responds to the same messages as regular solvers, but responds with special replies suited for a solver at either end of the lattice. Regular solvers therefore don't need special logic to deal with missing neighbors if they're at either end of the lattice. Instead, they just send the same messages to their neighbors, the same as any other solver, and the `Sentinel` responds appropriately.

The final argument to `lists:fold/3` is a sequence of integers ranging from 0 to `NSites-1`, generated by the `lists:seq/2` function. These integers identify each solver process's lattice site, and are used primarily when the application's final results are printed to the screen.

Upon completion, the `lists:fold/3` function returns a list of all the pids of the solvers it created, with each solver except the leftmost one knowing the process ID of its next neighbor. The next line of code in our solution adds the final process link, informing the list's leftmost process that its left neighbor process is the `Sentinel`:

```
hd(Pids) ! {set_next, Sentinel},
```

The `hd/1` function returns the head of a list; here, it lets us isolate the leftmost process in the `Pids` list so that we can send it a message.

Before explaining the rest of the `poisson1d/0` function, which is concerned with collecting the results of the solver processes, it's instructive to examine the code's details that each concurrent solver executes. Each solver process executes the `phi/4` function shown next:

```
phi(Collector, NIters, Prev, Consts) ->
  receive
    {set_next, Next} ->
       phi(NIters, Collector, Prev,
         Next, Consts, 0.0)
 end.
```

Here, the `receive` statement declares that the solver expects to get a message of the form `{set_next, Next}`, where `Next` is a variable bound to the second element of the received two-tuple. When it receives such a message, the function executes the body associated with that message, which in this case simply invokes a separate `phi/6` function. Because the `Prev` solver is passed as a function argument to `phi/4` and the `Next` solver is received in the `set_next` message tuple, each solver is aware of both its right and left neighbors by the time it enters the `phi/6` function.

The `phi/6` function consists of two separate clauses. The first clause handles the special case

of the solver iteration count dropping to 0, while the second clause is a recursive function that calculates the Poisson solution:

```
phi(0, Collector, _, _, {I, _, _},
   Phi) ->
       Collector ! {self(), I, Phi};
phi(NIters, Collector, Prev, Next,
   {_, H, Rho}=Consts, Phi) ->
       PhiPrev = adjacent_phi(Prev, Phi),
       PhiNext = adjacent_phi(Next, Phi),
       NewPhi = (PhiPrev + PhiNext)/2
          + H/2 * Rho,
       phi(NIters - 1, Collector, Prev,
          Next, Consts, NewPhi).
```

The first clause simply sends the calculation result `Phi` together with the lattice site number `I` and the solver's pid (retrieved via the `self()` call) back to the `Collector` process. The second clause, though, is more involved. It invokes the `adjacent_phi/2` function twice, first to send its current `Phi` value to the `Prev` solver, which is its neighbor to the right, and request that solver's current `Phi` value, and then to do the same with the `Next` solver, its neighbor to the left. Once it obtains new values from each of its neighbors, it uses them together with `H` and `Rho` to calculate the `NewPhi` value for itself. Finally, it invokes itself recursively, passing its decremented iteration counter and its `NewPhi` value. Only when the counter reaches zero will the first clause of `phi/6` match, thus ending the recursion.

The `adjacent_phi/2` function is responsible for communicating with a solver's neighbors. It takes an adjacent solver's pid and the calling solver's current `Phi` value as arguments.

```
adjacent_phi(Adjacent, Phi) ->
    Adjacent ! {put, Phi, self()},
    receive
      {put, AdjacentPhi, Adjacent} ->
            AdjacentPhi
    end.
```

It first sends a message to the `Adjacent` solver in a three-tuple consisting of the atom `put`, the value of `Phi`, and its own pid obtained via the `self()` call. The receiving solver uses the included pid to know where to send its reply, which is a message of the same three-tuple form. Upon receiving the reply, the `adjacent_phi/2` function returns the `AdjacentPhi` value sent by its neighbor.

Because the `Sentinel` process described earlier is a special form of a solver, it handles the same messages as other solver processes, but does so differently.

```
sentinel() ->
   receive
         {set_next, _} ->
               ok;
         {put, _, Pid} ->
             Pid ! {put, 0.0, self()};
         stop ->
             exit(normal)
  end,
  sentinel().
```

The `Sentinel` handles the `set_next` message by simply ignoring the pid included in the message (via the special underscore "don't care" variable). This is because, unlike a regular solver, the `Sentinel` has no need to initiate communication with any neighbors, so it need not store any pids for previous- or next-solver processes. To handle a `put` message, the `Sentinel` throws away the included phi value and always sends back 0.0 for its own phi value. Finally, the `Sentinel` also handles a `stop` message, which allows the `poisson1d/0` function to tell the `Sentinel` to exit once all the solvers have completed their calculations. Figure 1 shows the arrangement of the solver processes and the sentinel process.

The `sentinel/0` function effectively lets the `Sentinel` process implement polymorphism, thereby simplifying all solver processes. By handling the same messages as regular solver processes, it lets solvers be unconcerned with the boundary conditions existing at either end of the Poisson lattice. Whether communicating with a regular solver process or the `Sentinel` process as its neighbor, a solver sends the same form of messages and receives the same form of replies.

The final portion of my Erlang 1D Poisson program, which is at the end of the `poisson1d/0` function, is responsible for collecting solutions from each solver.

```
lists:foreach(
   fun(Pid) ->
      receive
        {Pid, I, Phi} ->
           io:format("~2w ~.10f~n",
              [I, Phi])
      end
   end, tl(lists:reverse(Pids))),
Sentinel ! stop,
ok.
```

To collect each solver's solutions, we walk the list of solver processes using `lists:foreach/2`. It takes two arguments—a function and a list—and simply calls its function argument for each list element from left to right. The list we pass to `lists:foreach/2` is a modified copy of the list of the solver pids we created earlier via `lists:foldl/3`. Recall that the original process list contains the `Sentinel` process, which we don't want to include in our collection process. Also recall that the process list is in reverse order because of how we built the accumulator during the fold. To fix these issues, we reverse the list's order by passing it to `lists:reverse/1`, which results in the `Sentinel` process being at the front of the list. To skip it, we use the `tl/1` function to obtain just the list's tail. The result of these two function calls is a list of solver pids ordered from lattice points 0 to `NSites-1`.

For each solver process, our solution collector waits to receive a message from that specific process indicating its lattice point number `I` and its calculated value `Phi`. The presence of the `Pid` variable within the three-tuple message that the `receive` call waits for means that the collector won't proceed until it receives a matching message from that specific process. Selective receive calls of this nature are extremely useful within Erlang applications, allowing different parts of the code to be concerned with receiving only the messages it knows about and understands, even if messages intended for some other parts of the code are already sitting unreceived in the process message queue. The selective receive scans the message queue to find a type matching the pattern or patterns specified in the `receive` call, skipping any messages that don't match. If it scans the whole queue without finding a matching message, `receive` will then block awaiting the arrival of a matching message. By default, `receive` blocks forever, but it can take an optional `after` clause to specify a maximum number of milliseconds to wait.

Using the selective receive capability also helps order our Erlang solution's printed output. If you run the original Python solution, you're likely to see it print garbled output to the screen due to thread preemption. But because the Erlang solution collector waits in order for the result message from each solver process, its printed output is never garbled as a result of overlapped process execution.

The collector effectively serializes the results of all the solvers by taking advantage of Erlang's selective receive capabilities. Even if the collector receives a message from a solver out of order, that message will sit in the collector's message queue until the collector executes a `receive` specifically matching it, and only then will the Erlang runtime system extract that message from the queue and deliver it to the collector process. To implement something similar, the Python program would have to join each solver thread in order of lattice site identifier and then print the solution within the main thread instead of within each solver thread. Erlang's selective receive makes our collector approach simple and elegant.

Interestingly, our Erlang program contains no condition variables, mutex locks, synchronization blocks, or other thread-related constructs required in mainstream languages such as Java and C++.

Performance

Writing high-performing systems in Erlang is possible, but it depends on the application type. This article shows an Erlang Poisson 1D solver, but that choice isn't intended to imply that Erlang is an excellent programming language for number crunching; the requirement to show a Poisson 1D solver solution was simply part of the call for articles for this special issue. Solving such numerical problems would be better done in a general-purpose language such as Fortran or C, or perhaps in a math-oriented domain-specific language.

Erlang is well suited for applications requiring coordination, integration, and control, especially in systems that perform a significant amount of I/O such as Web and database servers. Using the Poisson 1D problem as a way to compare Erlang's performance to C or Fortran would be fruitless.

Rather than examining raw performance figures, it's more instructive to examine how the Erlang lightweight process model fares on multicore systems. As explained earlier, by default the Erlang runtime executes one scheduler per CPU core. However, it also provides command-line options to enable only a particular number of schedulers, as well as to turn off its virtual machine's symmetric multiprocessing (SMP) capabilities altogether.

To illustrate Erlang's ability to make use of multicore systems, I made the following changes to the Poisson 1D program to make it easier to use as a test program:

- allowed the number of lattice sites and number of iterations to be specified from the command line;

- disabled all program output, to avoid measuring the time required to print results for numerous lattice sites to the screen; and
- avoided the use of the selective receive capability.

The reason I avoided selective receive for these experiments is worth noting. First, the results collector in the program originally used selective receive not only to order the solution results, but also to pedagogically explain a useful feature of Erlang messaging. The original program also processed only 16 lattice sites; however, for a much greater number of sites, selective receive can be expensive because the Erlang runtime must scan through a potentially large number of messages in the collector's message queue to find a match. The collector is essentially a contended resource to which all site processes must send their results, so to better handle larger numbers of lattice sites, I modified the collector to avoid using selective receive and to instead receive all lattice site result messages in whatever order they arrive and sort the entire results set once all site messages have arrived.

I then ran the program with four different numbers of lattice sites—8,000, 16,000, 32,000, and 64,000—and ran each for 10,000 iterations. Lattice spacing for each run was 0.1, just as in the original program. I measured elapsed time and CPU usage for each different lattice size, first with Erlang's SMP capabilities turned off, and then with varying numbers of schedulers. I tested these combinations using Erlang/OTP release R15B01 on an otherwise idle Ubuntu 12.04 system with 16 Gbytes of RAM and a 3.4-GHz Intel i7-2600K processor, which is a quad-core CPU but due to hyperthreading appears to Erlang as an eight-core system. I thus capped my testing at a maximum of eight Erlang schedulers.

Figures 2 and 3 show the results of this testing. Figure 2 shows the elapsed time of each program run, and Figure 3 focuses on the CPU usage for each run. Not surprisingly, running the solver with SMP disabled or with just a single scheduler thread uses only one CPU core and takes longer than any of the other trials where multiple schedulers were used. Interestingly, running the solver in a single scheduler thread takes longer than running the same program with Erlang SMP disabled; this is because the SMP version of the Erlang runtime is larger and more complicated because it must handle multiple OS threads with appropriate locking, condition variables, and other typical, low-level concurrency constructs. For two or more schedulers, elapsed execution time generally decreased as the number of schedulers increased, regardless of the number of lattice sites.

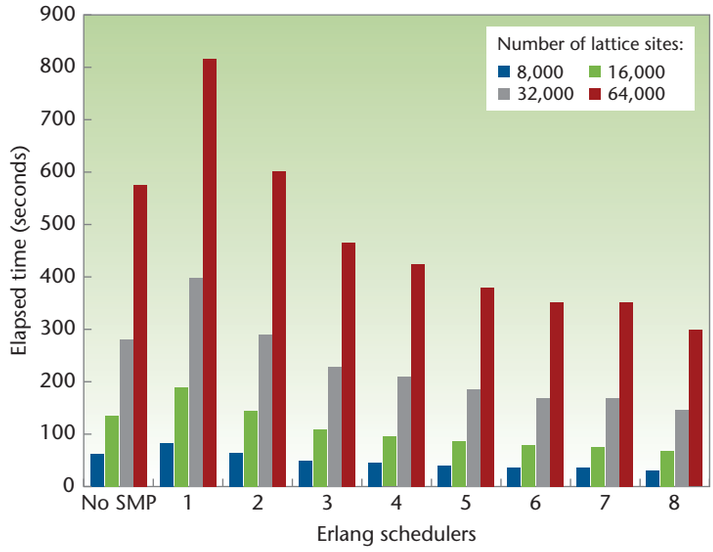

Figure 2. Program execution time as a function of the number of Erlang schedulers. As the number of Erlang schedulers increases, more CPU cores are used, thus reducing elapsed time for the revised Poisson 1D solver program independent of the number of Poisson lattice sites. (SMP stands for symmetric multiprocessing.)

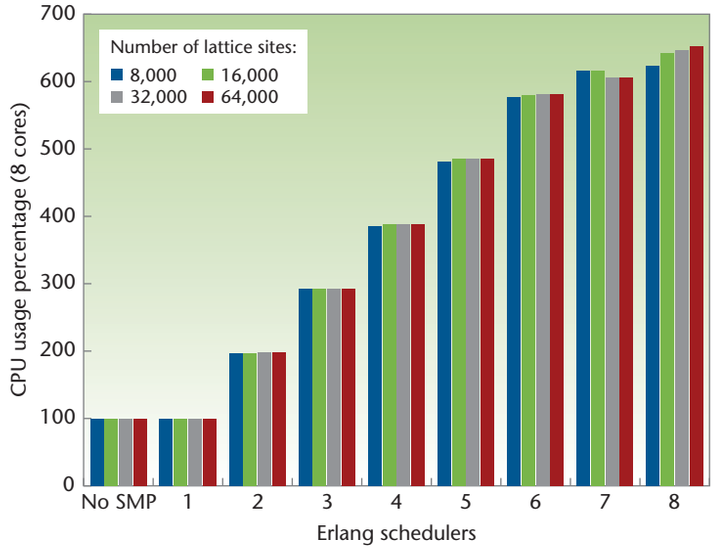

Figure 3. System CPU usage as a function of the number of Erlang schedulers. As the number of Erlang schedulers increases, the CPU usage of the revised Poisson 1D solver program increases fairly linearly, independent of the number of Erlang processes serving as Poisson lattice sites.

Figure 3 shows that CPU usage increased linearly for smaller numbers of schedulers, but increased somewhat more shallowly for higher numbers of schedulers. The graph also shows CPU usage to be independent of the number of lattice sites measured.

Overall, these trials show that Erlang lets applications easily make excellent use of multicore systems. Notably, the code used for all test runs was identical; no code modifications or tuning were required to work with the varying numbers of Erlang schedulers.

The Erlang development team at Ericsson works constantly to ensure that each new release of Erlang takes as much advantage of the underlying OS and hardware concurrency capabilities as possible. Team members often have access to advanced multicore machines that they use to measure Erlang, reduce or eliminate any concurrency-related bottlenecks, and tune their code so that it continues to perform well as CPU core counts continue to increase.

The 1D Poisson solver shows the beauty, power, and simplicity of the Erlang concurrency model, but there's much more to Erlang/OTP than what this article has explored. Erlang/OTP comes with an extensive set of standard libraries and frameworks, and you can find many additional open source libraries and tools written in Erlang on GitHub and other sites. It also comes with several databases, including *mnesia*, its distributed transactional database. The runtime includes an extensive tracing system that lets developers peer into their running applications to see what messages its processes are sending and receiving, and what functions are being called along with the arguments and return values of those functions. The tracing feature is an invaluable debugging aid, and its overhead is low enough that it can be used on live production systems.

The source code for Erlang/OTP is freely available at www.erlang.org, and you can find documentation and learning materials there as well. Several excellent Erlang books are available,[2–5] and the *Learn You Some Erlang for Great Good*! website (http://learnyousomeerlang.com) provides extensive tutorials for the language and the standard OTP frameworks.

References

1. *OTP Design Principles User's Guide*, version 5.9.1, Ericsson AB, 1 Apr. 2012; www.erlang.org/doc/design_principles/users_guide.html.
2. F. Cesarini and S. Thompson, *Erlang Programming,* O'Reilly Media, 2009.
3. J. Armstrong, *Programming Erlang: Software for a Concurrent World,* Pragmatic Bookshelf, 2007.
4. M. Logan, E. Merritt, and R. Carlsson, *Erlang and OTP in Action,* Manning Publications, 2010.
5. S. St. Laurent, *Introducing Erlang*, O'Reilly Media, 2012.

Steve Vinoski *is an architect at Basho Technologies. His work has focused on distributed systems and middleware systems for more than 20 years, including distributed object systems, service-oriented systems, and RESTful Web services. Vinoski has a BS in electrical engineering from Christian Brothers University in Memphis, Tennessee. He writes "The Functional Web" column for* IEEE Internet Computing*, in which he explores the use of functional programming languages for Web development. He's a senior member of IEEE and a member of the ACM. Contact him at vinoski@ieee.org, read his blog at http://steve.vinoski.net, and find him on Twitter at @stevevinoski.*