Message Passing: An Alternative to Shared State

SparkNotes™

- Message passing is an alternative to the shared state model where threads send duplicates of their data to one another via messages, thus eliminating the need for locks et al.
- Erlang is a soft real-time, dynamically typed, functional programming language that uses the message passing metaphor.
- Erlang allows threads to use pattern matching (think Regular Expressions) to filter, order and ignore messages, however pattern matching can be used in other scenarios, like if statements.
- A tuple space is a data structure that can be built with message passing and allows for thread-safe sharing of memory; it can be used to implement more complex patterns like producer/consumer.
- Erlang, and most message passing implementations, are well suited for highly parallelizable problems where a lot of data doesn’t need to be shared, but they are no panacea for all concurrency issues.

Abstract

Over the past few weeks, we have approached the subject of concurrency from the perspective of the shared state model. That is, building threads that operate on a common location in memory, using constructs such as locks and atomic primitives to ensure a valid execution. One of the strongest alternatives to this model is that of message passing, where individual threads can’t access a common state. Our project examines this alternative via a programming language called Erlang, which we used to build and test a data structure called a tuple space to demonstrate how message passing can be used in a ‘real world’ scenario.

A Quick Review of Message Passing

As discussed in the Andrews and Schneider article, message passing, is architecture wherein threads share state not by working on the same memory locations, but instead by sending duplicate copies of data to other threads via ‘messages’. Thus the shared state model would be similar to workers sitting at the same table discussing a problem they were trying to solve, while trying to co-ordinate such that they didn’t talk over one another. Whereas in message passing, workers sit in separate rooms sending emails of their progress to specific workers or broadcasting them to many.

Introducing Erlang
Erlang is a *soft real-time*, functional programming language created by Ericsson. It is a dynamically typed, single assignment language that allows its runtime to manage memory, using a garbage collector (that maintains the soft real-time property) to clean up after processes.

As opposed to more prominent language like C++ and Java, Erlang doesn’t expose the concept of kernel threads to programs in the runtime. Instead Erlang uses a lightweight-threading model wherein threads are managed by the runtime, and it is not implausible to have millions running at the same time. While Erlang calls these threads processes, we will continue to refer to Erlang’s processes as threads so as to avoid confusion with the traditional definition of processes.

One of the subtle differences between Erlang and traditional object oriented languages is the concept of pattern matching. In languages like Java where a function call must match the exact signature of the callee, Erlang allows for more flexibility. Specifically the use of wildcards is not only permitted, but a standard practice in the Erlang community. Wildcards can be used to ignore specific parameters or all parameters, acting as a catchall.

Pattern matching becomes an important part of the Erlang toolkit when it comes to the case of inter-thread communication. When a thread wishes to communicate with one or multiple threads, it does so by sending messages to a receiving thread’s mailbox. By using pattern matching against the incoming message, the receiving thread can read, ignore or redirect the messages sent to it. Thus in the classic publisher/subscriber model, a publisher could send out all messages as a broadcast and allow the consumer threads to filter out messages that aren’t relevant to that thread.

Pattern matching is implemented using fairly standard techniques in functional programming languages. At its heart, pattern matching uses decision trees generated by the compiler to determine which patterns, if any, match a given input. While decision trees themselves are not necessary for pattern matching, they help optimize the match process by ensuring a given condition at a specific position is checked once. Additionally conditions can be checked in parallel for a given depth in the decision tree. When compiling the pattern, Erlang builds a tree such that terms are matched in a left to right order. The resulting tree is translated to Erlang’s bytecode where it can then be executed by the runtime. The matching process itself doesn’t actually complete until the resulting function called by a match finishes executing.

**Erlang and Concurrency**

It is this combination of message passing, pattern matching and single assignment that makes Erlang such a powerful tool in addressing concurrency. Switching to memory passing has allowed Erlang to eliminate the traditional synchronization primitives. Instead of locking on an object to update a value, one can simply send messages to the actor that represents the state of that ‘object’.
Additionally, sending a message is asynchronous, meaning a thread can fire off a message and immediately continue along if it wants to. So if one needed to add an element to a shared list, the processes doesn’t have to wait around for the insertion to complete if it doesn’t need to.

In Erlang actors are represented by a thread. Each thread has a relatively low overhead of just 309 words of memory. This size includes 233 words for the heap and stack. The garbage collector can grow and shrink the heap area as needed. The nature of actors, being closer to threads than a function, means they don’t have a return value. In Erlang almost all data is shared via message passing.

Erlang’s scheduler is somewhat simplistic. Each thread is assigned a number of ‘reductions’. Each reduction represents a single instruction in Erlang. When the number of reductions assigned to a process reaches zero, or if the thread enters a receive block when there are no waiting messages the thread can be preempted. In previous versions of the Erlang runtime, threads were allocated 2,000 reductions by default, however in later versions (13 and later) the number of reductions assigned is decreased as more threads are created. This system ensures each process gets a relatively fair shot at processor time, but doesn’t guarantee they will receive equal time.

```erlang
  go()->
  Pid = spawn(?MODULE, reciever, []),  % make receive function a thread
  Pid ! {msg, "Hello world!"},        % send the function a message

  reciever()->
  receive
    {msg, Text} -> io:format("\nMessage Received: ~s \n", [Text])
  end.
  The hello world of message passing.
```

Ordering of events also proves to be a particularly troublesome issue in shared state languages. A disordering of instructions can lead to race conditions in code. Erlang allows for the selective reception of messages; thus one can write a function that requires a message matching a specific pattern to be received before continuing to wait on a different message. More clearly, if a function requires messages A and B before it can run successfully, you can easily write code so that if message B arrives before message A, the function will block until A is received.

```erlang
  receiver() ->
  receive
    { first_message } ->
      io:format("\nGot First")
    end,
  receive
    { second_message } ->
      io:format("\nGot Second\n")
  end.
```

```erlang
...```
This function will not continue until both first_message and second_message are received.

Erlang also benefits from a runtime where support for concurrency isn’t tacked on. In most languages starting a process may be simple, but telling if it still running or has encountered an error is much more complicated. Using a specialized version of the spawn function – Erlang’s parlance for creating a thread – the creator of a thread can ask the runtime to send it a message should the new thread error out. The Erlang runtime also maintains a thread registry, making it easy to find and send messages to actors that should be shared across the application, such as caches or global queues.

Tuple Spaces
If one wanted to recreate a data structure that was similar to the thread pool created in class, how would one go about that? Erlang makes spawning threads incredibly cheap, so one needn’t be as concerned about reusing threads so much as sharing tasks concurrently. More generically, can we create an actor that makes it easy to add, update, delete and search variables shared with other threads without having to communicate with said threads directly, while at the same time maintaining mutual exclusion?

One of the simplest and cleanest solutions to this is the tuple space. The idea is that actors are allowed to add and remove elements in an atomic fashion. Thus if one wanted to implement the producer/consumer pattern, the tuple space would allow for safely adding work items while items are being requested to removed or found at the same time. Additionally the structure would offer functionality that would guarantee that the same element wouldn’t be returned to two different consumers.

Our implementation of the tuple space internally is based off of Erlang’s list structures. When an item needs to be added or removed from the tuple space, the tuple space retrieves the message from its mailbox and then recursively redefines itself if any changes need to be made to the internal structure. Tuples are grouped by atoms (which are akin to enums in Java), thus it is possible to store any data type, as well as mixing data types in our tuple space.

Testing and Benchmarking
For our benchmarking we decided to create a scenario that is common to Facebook, Twitter and systems that use a publisher/subscriber model. Essentially we spawn a bunch of threads where users send multiple messages to other users via the tuple space, while ensuring no messages were lost and the data structure maintained consistency. Due to Erlang’s message passing architecture we sacrificed a precise ordering of messages (that is if A sends a message to C before B, the message from B might be received first) but were rewarded with a concise data structure that took more time to correct syntax errors than logic errors.
Additionally we created a similar test using a built in Erlang data structure called ETS or Erlang Term Storage. Like our tuple space it allows for tuples to be added and removed in an atomic manner. However, unlike our structure ETS doesn’t guarantee that tuples will be added, the insertion may fail and used a more complex internal data structure – a hashtable.

Prior to running performance tests on our ETS and list-based tuple space implementations, we assumed that ETS would be faster than our implementation. As we started running the experiments, however, we began to realize that our initial hypothesis was far from the truth. Running on the CIMS machines (Energon1, 8 cores), we found that the list-based tuple space we implemented was orders of magnitude faster than ETS. The ETS implementation did better on "edge cases" where we have a very large number of clients sending fewer messages and when we had fewer clients sending many messages, but this just means that instead of processing the same messages ~700x slower, it was only ~300x slower. This raises the question, why? As it turns out, this is directly related to how the two data structures deal with memory.

ETS tables are included within the Erlang runtime, but more specifically, a portion of the machine where there is no garbage-collection, or rather, you can think of adding/removing to an ETS table as malloc/free commands. The big appeal of this is that you do not have to deal with the performance hit of garbage collector invocation, while heap-based data (i.e. lists) will have a hidden cost of garbage collection that will increase proportionally with the amount of data in your data set. Unfortunately, ETS tables suffer even further slowdown from the fact that any time you wish to access data in the ETS table, it must be copied from the table into the process heap before it can be used. Using ets:match and ets:select allow you test potential data prior to copying over, but the problem still remains. Further, unlike our list-based structure that resizes by copying itself, ETS takes a hit when it has to resize and rehash the internal structure.

Given this new knowledge, our results now seem to bear more meaning. As the size of the data set increases, so does the overhead experienced on the heap. Thus, when we ran our simulations with large number of clients or large number of messages, we began to see better times for ETS relative to the list implementation because ETS’ memory model was compensating for some of the time.

Conclusion
Erlang, or any other message passing system, are not the panacea for all concurrency issues. It doesn’t eliminate all problems that crop up with parallel code, like deadlocks and resource starvation, and its single assignment type system will cause bottlenecks not found in shared space languages when large blocks of data need to be manipulated by multiple workers, as each worker requires a new copy of the data set.

That being said, the message passing architecture is an extremely powerful abstraction when used correctly. For scenarios where smaller tasks and blocks of data need to be manipulated by multiple workers Erlang not only performs at its best, but it also makes coding a joy. Our implementation would have taken orders of magnitude more time to debug in languages that used standard locking primitives
and the resulting code would not have been nearly as concise. Erlang is a powerful tool in the realm of concurrent programs that compliments traditional shared state models. If one has a problem that fits well into this model, the reward will be cleaner and easier to implement code.

Appendix

Actor Model – a programming model wherein ‘actors’ represent the state of the program. In this model the actors maintain mutable data while communicating via immutable messages. An actor is the only one who can access its state directly. While actors don’t control what messages they receive they do determine how to handle (including ignoring) messages. Actors are also able to spawn more actors.

Decision Tree – a graph like structure where each node represents a certain choice or match, leaves represent a complete match for that pattern path. Wildcards are allowed. Erlang uses decision trees to implement pattern matching.

Single Assignment – in type systems that use single assignment, a ‘variable’ may only be set once per its scope. Thus languages that use single assignment, the following would not be allowed:

\[
\begin{align*}
X &= 5 \\
X &= X + 1
\end{align*}
\]

To add one to X, you must assign that value to a new variable. However, if a variable falls out of scope, it may be declared again.

Soft Real-Time - a system is said to be soft real time when it is generally expected to finish a given task in a predictable amount of time. If the system fails to meet this deadline the experience is degraded, this contrasts with hard real-time systems where a missed deadline is considered to be a system failure. A DVD player is a good example of soft-real time. If the system isn’t able to render a frame in a certain amount of time, the picture experience is degraded. Anti-Lock brakes are an example of hard-real time, where an arbitrary delay in responding to stepping on your breaks is unacceptable.

References

