Abstract—Distributed systems became an architectural standard for modern computer systems. An actor model is one of the possible solutions for process management in such an environment.

This project report discusses about actor model in general, Elixir implementation in particular with examples of applications and abstractions based on the model and interaction in a pseudo-distributed environment. Moreover, it provides comparison of parallel algorithm implementation using operating systems threads with an actor-based solution. The experiment results demonstrates a performance advantage over the former approach.

I. OVERVIEW

A. Introduction

From the classical perspective, the unit of parallel execution is a thread, which exists inside an operating system process structure and can be executed on some processor (physical or logical). Each thread has its attributes related to execution, such as context, stack and shared process heap. Imagine it exists a server, which is needed to handle one million requests per second. It is really hard to process such load using native OS threads and there are several reasons why:

1) Each thread must share dynamic memory (heap) with others. This feature can produce deadlocks, race conditions and starvation (when one thread has to wait too long to get a resource access).

2) The number of threads in each OS is bounded[1]. There is not possible to create one million threads within one machine.

3) The thread switch on a processor (context switching) is an expensive operation and it is better to avoid frequent switching.

4) A thread is a more lightweight structure than a process, but still takes time and memory to create and also time for release allocated resources.

The thing is the management of large thread numbers is hard for any processor. Is it any alternatives for a native thread? One needs something more lightweight, but still effective in terms of execution. This is where an actor model comes to play.

B. Actor model explanation

An actor model is a mathematical model of concurrent computation, proposed by Carl Hewitt in 1973. It is based on the term "actor" - an entity, which can:

1) Send a message to another actor (asynchronously).

2) Receive a message from another actor.

3) Create new actor(s) (actor-children).

The actor itself has several components:
1) Mailbox - the limited queue of messages (shown in figure 2). It allows actors to process messages from other actors sequentially.

2) State - internal actor data, which can be accessed directly only from an actor-owner (shown in figure 3). When one actor wants to read the data of another actor, it has to send a message and receive back the state.

3) Execution function (similar to the thread function). It can contain problem-oriented logic as well as a behaviour related to message processing. After function finish, actor terminates.

Nowadays, there exists several languages and libraries implementing the actor model: Erlang, Elixir, Akka (Scala language library) and others.

C. Comparison with native threads

What are the actors in terms of OS? Each actor is a lightweight thread running on some tiny hosted virtual machine (e.g. Beam VM or Java VM). The VM is responsible for actor scheduling (with low scheduling overhead) and memory management[2]. This approach is better than thread-based in terms of the initial task because:

1) Actors do not share memory. Each actor can use its own allocated space, so one can forget about race conditions and unexpected data modifications.
2) The actors usually run within a virtual machine, so their number is not limited by thread count. But still, one million of actors can not be created at a single moment.
3) There is no need for frequent thread context switches because a VM uses the number of threads equal to the number of logical processors in an OS.
4) The actor can be created and terminated fast. That means, during one second, it is possible to create 1 million actors and handle all the requests.

Actor and thread comparison represented in the table below.

<table>
<thead>
<tr>
<th>Property</th>
<th>Thread (Linux)</th>
<th>Actor (BeamVM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runs on</td>
<td>OS</td>
<td>VM</td>
</tr>
<tr>
<td>Memory model</td>
<td>Shared memory</td>
<td>Message passing</td>
</tr>
<tr>
<td>Creation and termination time cost</td>
<td>Normal</td>
<td>Low</td>
</tr>
<tr>
<td>Thread context switching</td>
<td>Frequent</td>
<td>Not really frequent</td>
</tr>
</tbody>
</table>

II. THREADS AND ACTORS EXPERIMENTS

Consider a simple exercise: it needs to sort an array in parallel. One of the best algorithm candidates for this is merge sort. It can be simple paralleled using a data-parallel approach shown in figure 4 below.

As a native-thread language there is Rust and Elixir as an actor-based one. Although, Elixir uses list structure for sequential data representation, it
should not affect the whole performance test, because merge sort does not use item access by index.

For parallelization in Elixir, it uses Task abstraction over actor parallel computation and in Rust there are native OS threads from limited pool (the source code is provided in Links section).

Experiments environment:
- Intel Core i7-6700HQ 2.6 GHz (4 physical cores, 8 virtual cores)
- Ubuntu 18.04
- rustc version: 1.42.0
- elixir version: 1.10.3

It was implemented both sequential and parallel merge-sort using mentioned languages. Parallel versions have an option to limit a number of executors (for Elixir - an actor, for Rust - a thread). First experiment compares execution time with fixed array size (100000 elements) and different executors number. The result is shown in figure 5 (y-axis has a logarithmic scale). It is clear, that Elixir’s actors outperform native Linux threads in terms of high-granularity tasks. Moreover, the actor-parallel program runs quicker than Elixir’s sequential one with increasing of executors number, while thread-parallel process slows down more and more during the experiment. Even the sequential Rust program runs better than parallel one after 50 executors, but parallel Elixir program outperforms sequential version after 100 executors. This can be caused by OS thread issues like context switching and creation overhead.

In case, when one doesn’t need so high parallelization level, it is better to fix number of executors. In second experiment, merge sort uses only 8 executors (equal to the number of logical processors) and processes different array sizes. Result is shown in figure 6. Actors again perform better than threads. This proves that the actor-based approach is more efficient in terms of execution time.

Third experiment checks than speedup of these two approaches (shown in figure 7). Here one can notice, that thread-based program has much better speedup. That means, Elixir’s actor system is not an ideal solution, particularly in tasks, where one needs high speedup improvement in comparison with a sequential solution.
III. ACTOR MODEL ABSTRACTIONS

A. Introduction to actor communication

Now one can consider each actor as an isolated process with internal state and mailbox (hereinafter as an actor-based application is Elixir application). Also, an actor in Elixir has its own PID (process id), which is a unique actor identifier, used for message sender or receiver specification. In BeamVM every execution unit is an actor including the shell used for REPL. It simply reads data from standard input, executes given instructions with BeamVM and returns a result to the standard output. As a normal actor, it can use message-passing primitives and create child processes (actors).

B. Message passing

1) Message send: The example of message send is shown in listing 1.

```
1  # Actor A function code
2  send(other_actor_pid, message)
```

Listing 1. Message send example

For direct message send it uses send function, which takes two arguments: PID of actor-receiver and the message itself. It runs asynchronously, so after this function invocation, actor A will continue its execution.

2) Message receive: The example of message receive is shown in listing 2.

```
1  # Actor A function code
2  receive do
3    <msg pattern 1> ->
4      # Actions definition
5      ...
6  after
7    1_000 -> "nothing after 1s"
8  end
```

Listing 2. Message receive example

For message receive it uses receive construction, which includes:

1) Pairs `<msg pattern I> -> <related actions>`. If next message from mailbox matches to a given pattern, related actions is performed. Otherwise, actor waits for a new message matching to any other pattern.

2) Statement `after`, which specifies timeout for message delivery. If no new messages in mailbox in this time, the actor stops waiting and continues execution.

C. Actor creation

As it mentioned before, an actor can create a child actor. There are two different types of relation between parent and child:

1) Unlinked, which means next behaviour: if actor-child terminates with an error, actor-parent is not terminated too, it continues its normal execution. For example, actor creation represented in listing 3.

```
1  # Actor A function code
2  child_pid = spawn(fn ->
3    ...
4  end) # Actor B function definition
```

Listing 3. Simple actor creation example

2) Linked, which means next behaviour: if actor-child terminates with an error, actor-parent terminates too. This case is shown in listing 4.

```
1  # Actor A function code
2  child_pid = spawn_link(fn ->
3    ...
4  end) # Actor B function definition
```

Listing 4. Linked actor creation example

Actually, the actor is a low-level computation mechanism and it is better to use something more
complex and functional. This section will cover several abstractions based on actor system.

D. Task: asynchronous computation primitive

As it mentioned in previous section, in merge-sort Elixir implementation it uses Task for parallel computation. What is Task structure? First of all, let consider simple actor interaction:

1) Actor \( A \) is computing its function, when it needs to run some computation \( F \) in parallel. For this it can be used actor child, then \( A \) creates a new actor \( B \) with execution function \( F \) (finished with message sending to \( A \)). After this, \( A \) continues its own execution and then starts waiting for message from \( B \).

2) Actor \( B \) computes \( F \) in parallel with \( A \) and when computation is finished \( B \) sends result \( R \) in a message to actor-parent (i.e. actor \( A \)) and terminates with successful code.

3) Meanwhile, \( A \) receives this message with \( R \) and uses it after.

The whole process is shown in figure 8.

![Fig. 8. Asynchronous execution using actors](image)

The interaction looks like typical fork-join pattern and uses quite commonly in Elixir. That’s why it exists Task abstraction for this entire process. One can just write next code lines and get the same execution, shown in listing 5.

```
1  # Actor A function code
2  f = ... # some complex function
3  b_task = Task.async(f) # Step 1 (fork)
4  # ...
5  # Other computation
6  # ...
7  Task.await(b_task) # Step 3 (message obtaining)
```

Listing 5. Task template

This is much shorter than implement such functionality with bare actors.

E. Generic server

1) \texttt{GenServer}: Asynchronous computation can be performed not only with peer-to-peer semantic, but one can also use a server for this. In Elixir, it exits a generic server(GenServer) pattern. This server has all of the actors’ features and some more:

- Internal functions (user can override):
  - start empty server with an initial state (\texttt{init}) and return PID
  - handle some request synchronously (\texttt{handle_call}) and reply
  - handle some request asynchronously (\texttt{handle_call}) with no reply

- External API (user can invoke):
  - perform a synchronous request (\texttt{call})
  - perform a asynchronous request (\texttt{cast})

Each handle method is implemented on top of message passing and the key feature of that approach is after another request handling a server won’t terminate like a simple actor. For these, it uses a continuous loop, shown in figure 9. One only needs to implement request handling and processing logic (orange blocks in algorithm).

In Elixir, there is GenServer module(namespace-like structure), which is like abstract class from OOP, because one can “inherit” its behaviour. It only needs to implement the handlers and logical functionality.
How can one use it? For example, one can create own module (set of functions) and implement basic key-value storage functions as it shown in figure 10.

```elixir
lex(1)> GSExample.start_link() 
{ok, #PID<0.142.0>}
lex(2)> GSExample.add(a, 1) 
{ok}
lex(3)> GSExample.all() 
%{a: 1}
lex(4)> GSExample.add_async(b, 2) 
{ok}
lex(5)> GSExample.all() 
%{a: 1, b: 2}
lex(6)> GSExample.get(a) 
1
lex(7)> GSExample.all() 
%{a: 1, b: 2}
lex(8)> GSExample.pop(a) 
1
lex(9)> GSExample.all() 
%{a: 1}
lex(10)> GSExample.stop() 
{ok}
```

Fig. 10. GenServer application example

After `start_link()` call, one got a reply tuple `:ok, #PID<0.142.0>`, which means server is created and its PID (process id) is set. Here one can see, GenServer is an actor with several unique functions. The storage itself is a dictionary and at the same time the state of `GSExample` server. Using this, one can perform common queries to key-value storage like `pop, get, add and clean`.

**The source code of GSExample server**

2) **Agent**: GenServer is useful abstraction, but still is low-level because of need to implement request handling logic. What if one wants to focus only on application logic (request processing)? Then Agent abstraction can help us. Agent is Elixir vision of generic server, when GenServer is module from Erlang. It is implemented on top of GenServer providing simple and understandable API:

- **update** method for state changing
- **get** method for state reading
- **get_and_update** for reading and changing state
- **call** for custom synchronous request
- **cast** for custom asynchronous request

Usage example for Agent is represented in figure 11.

```elixir
lex(1)> AgentExample.start_link() 
{ok, #PID<0.142.0>}
lex(2)> AgentExample.all() 
%{}
lex(3)> AgentExample.add(a, 1) 
{ok}
lex(4)> AgentExample.all() 
%{a: 1}
lex(5)> AgentExample.add_async(b, 2) 
{ok}
lex(6)> AgentExample.all() 
%{a: 1, b: 2}
lex(7)> AgentExample.get(a) 
1
lex(8)> AgentExample.pop(a) 
1
lex(9)> AgentExample.all() 
%{a: 1}
lex(10)> AgentExample.stop() 
{ok}
```

Fig. 11. GenServer application example

This approach requires much less code lines and provides the same API, so Agent is more likely to use when user wants to simply implement generic server functionality. But if one prefer to have more control on execution, GenServer is better choice.

**The source code of AgentExample server**

F. **Supervisor**

Everything is still fine: one can create actors and even servers. Let’s imagine the next situation then:
simple data storage is created and a user tries to get a key, which is not present in the storage. What will happen then? The consequences are shown in figure 12 (\texttt{.get!} method raises an exception when queried key is not found).

\begin{verbatim}
18:34:57.012 [error] GenServer GSEExample terminating
** (KeyError) key :b not found in: %w{a: 1}
** (EXIT) no process: the process is not alive or there's no process current (elixir 1.14.3) lib/gen_server.ex:1593: GenServer.call/3

18:34:57.012 [error] GenServer GSEExample terminating
** (KeyError) key :b not found in: %w{a: 1}
** (EXIT) no process: the process is not alive or there's no process current (elixir 1.14.3) lib/gen_server.ex:1593: GenServer.call/3
\end{verbatim}

Fig. 12. GenServer fail example (stack trace is reduced)

In this case, the server falls and the user can not get an access to it anymore. How can one handle such situation? There is a simple approach using classical \texttt{try ... catch} construction, but actor model provides another way. One of the main principles for actor-based systems is "Let it crash", which means the model allows arbitrary parts of the system to fail sometimes. But what will it do after a failure? One can restart the dead unit using Supervisor.

Supervisor is another module supported by Elixir, which do not perform any computations, but controls execution of another unit. From actors perspective, Supervisor is an actor, which creates actor-children and controls their lifetime. In the case it will work like in figure 13.

\begin{verbatim}
18:34:57.012 [error] GenServer GSEExample terminating
** (KeyError) key :b not found in: %w{a: 1}
** (EXIT) no process: the process is not alive or there's no process current (elixir 1.14.3) lib/gen_server.ex:1593: GenServer.call/3

18:34:57.012 [error] GenServer GSEExample terminating
** (KeyError) key :b not found in: %w{a: 1}
** (EXIT) no process: the process is not alive or there's no process current (elixir 1.14.3) lib/gen_server.ex:1593: GenServer.call/3
\end{verbatim}

Fig. 13. Supervisor example (stack trace is reduced)

As it can be noticed, this approach works: if the child (GSEExample in this case) raises an exception and dies, the Supervisor (SupExample) captures it and restarts the server with initial state.

In general, Supervisor has two main options[4]:

1) Restart option (:restart) controls what supervisor consider as a successful child process (actor-child) termination. One specifies it to define WHEN the system will restart the child. Possible values:
   a) :permanent (used in the example) means child actor is always restarted by supervisor, even if it finished execution with normal reason. Every termination is considered as unsuccessful.
   b) :temporary means child actor is never restarted by supervisor. Every termination is considered as successful.
   c) :transient means child actor is restarted by supervisor only if execution is finished with abnormal reason (other than :normal, :shutdown, or \{:shutdown, term\}).

2) Supervision :strategy option defines HOW supervisor will restart child actor. Possible values:
   a) :one_for_one (used in the example) means if an actor-child terminates, only it is restarted
   b) :one_for_all means if an actor-child terminates, all other children are terminated and then all child processes (including the terminated one) are restarted
   c) :rest_for_one means if an actor-child terminates, it and the rest of the children started after it, will be terminated and restarted

The Supervisor behaviour is important to use in huge systems for different units control and hierarchy definition.

G. Working with different nodes

Still, it is examples of actors communication within single BeamVM. How about multiple VM? This is also possible and allows us to use the actor model in a distributed environment. Communication of two separate VM represented in figure 14.
Now let’s try to establish a connection between 2 VMs, but within one node (figure 15).

Function `Node.ping(node_id)` allows us to connect to other VM with known name (combination of a name itself and hostname). As it can be noticed, both nodes have each other in list of remote VMs.

BeamVM supports many complex distributed functions, but let’s focus on these three (hereinafter node means BeamVM instance):

1) Create an actor in the connected VM. The experiment example is shown in figure 16.

It looks similar to common actor creation, but it do it within remote VM. Note: function `node()` returns the node name.

2) Send a message to an actor on other VM. The experiment example is shown in figure 17.

Firstly, one creates a function for child-actor, secondly, spawns this actor on other machine and then sends a message to it. From the figure that is clear, send is successful, and also, as it mentioned in function, the actor respond to the shell with another message.

3) Receive message from an actor on other VM. To receive a message one typically needs to define receive template (figure 18).

From the figure it is apparent that the spawn-send-receive mechanism works in a distributed environment.

IV. CONCLUSION

The actor model is not a new concept but is still out of mainstream. However, there are several famous companies using this approach[5]. It indicates existing demand on reliable distributed software, for what actor model was created.

During this project, it is implemented parallel merge-sort algorithm with Rust and Elixir, examples of abstractions based on the actor model and interaction in pseudo-distributed environment. As a result, the actor model has better performance over classical threads in highly granular problems and can be a robust foundation for different parallel systems.
V. Links

Elixir code repository

Rust code repository

References


