ChatGPT

Enabling Parallel Common Test Suites in Rebar3

To integrate parallel suite execution into Rebar3's Common Test (`rebar3 ct`), you will need to modify the Common Test provider to spawn multiple Erlang nodes and run each test suite on its own node concurrently. Below are the required changes and considerations:

1. Cover Compilation (Instrument Once, Reuse on All Nodes)

Before launching tests, ensure code coverage is handled by instrumenting all modules **once** on the master node and sharing that instrumentation with all slave nodes. Rebar3 already supports cover compilation via `cover:compile_beam_directory/1` for each ebin directory (when `--cover` is enabled). You should invoke this once **before** starting any slave nodes, then use distributed cover to propagate the instrumented code to all nodes:

- Start the cover server on the master node (Rebar3 does this in `maybe_cover_compile` by calling `cover:start()` internally).
- After instrumenting modules on the master, add each slave node to the cover server using `cover:start(Node)` **after** the node is spawned. According to the Erlang tools documentation, *"Use cover:start/1 ... to add nodes. The same Cover compiled code will be loaded on each node, and analysis will collect and sum up coverage data results from all nodes."* [1] . This ensures each node runs with the already instrumented code (so you **don't** recompile with cover on every node).

In code, after compiling tests and before running them:

```
ok = rebar_prv_common_test:maybe_cover_compile(State, RawOpts),
% (This runs cover:compile_beam_directory on all relevant dirs on the master)
...
% For each spawned Node:
cover:start(Node) -> ok.
```

This uses cover's distributed mode so that all nodes contribute to a single coverage data set on the master node [1] . Rebar3's `maybe_write_coverdata` (in `rebar_prv_cover`) will then export a combined `.coverdata` file at the end of the run as usual.

Note: Make sure the master node is running in distributed mode (has a node name and cookie) **before** spawning slaves, otherwise `cover:start(Node)` will throw `not_main_node` errors. You may need to start the master node's networking by calling `net_kernel:start([Name, shortnames])` if not already a distributed node. Use `erlang:set_cookie/2` to set a cookie for authentication if needed.

2. Launch a Separate Node for Each Test Suite

Instead of running all suites in one VM or grouping them by directory, modify the test runner to spawn a **slave Erlang node** for each suite. This ensures each suite runs in parallel and in isolation:

- **Discover all suites:** After compiling tests, gather the list of all test suite modules (`*_SUITE`). You can reuse Rebar3's logic that finds suite files. For example, scan each test directory for `*_SUITE.beam` files (under `_build/test`) or use the resolved suite paths from `CTOpts`. This gives a list of suite names or beam file paths.
- **Spawn nodes:** For each suite, start a new Erlang node using the `slave` module. Use the local host and a unique node name per suite. For example, for a suite `my_app_SUITE`, you could call:

```
{ok, Node} = slave:start(Host, SuiteNodeName, Args)
```

where `Host` is the local host atom and `SuiteNodeName` is a unique atom (e.g. `suite1`, `suite2`, ...). Ensure the master (Rebar3) node is alive (named) – *"Common Test Master cannot start test nodes automatically. The nodes must be started in advance..."* [2] – so spawn these before invoking Common Test. Include `-pa` paths for all necessary ebin directories in `Args` so the slave has the same code path (apps and deps). You can retrieve the code paths from the Rebar state (e.g. `rebar_state:code_paths(...)`) and format them as `-pa ...` arguments. Also pass the same cookie (`-setcookie`) as the master.

- **Run the suite on the node:** There are two ways to execute the test on the slave node:

(a) Using `ct_master`: You could utilize `ct_master:run_test/2` to start the suite remotely. The function `ct_master:run_test(Node, Opts)` will spawn the suite on the given node using `ct:run_test/1` under the hood [3]. For example:

```
ct_master:run_test(Node, [{suite, [SuiteModule]}, {logdir, LogDir}])
```

You would build `Opts` for each suite (specifying that suite and a unique log directory). This call returns immediately (`ok`), launching the test asynchronously on the slave. You can call it for all suites in turn.

(b) Using RPC (simpler approach): Alternatively, you can directly RPC call the Common Test runner on the slave. For example:

```
RPCFun = fun(Node, Suite, LogDir) ->
           rpc:call(Node, ct, run_test, [[{suite, [Suite]}, {logdir, LogDir}]])
         end.
```

If you spawn an Erlang process or use `erlang:spawn_monitor` for each suite, each can call the above RPC in parallel. The RPC will block until that suite finishes on its node, then return the result (a summary tuple). This approach gives you direct access to each suite's result without extra parsing.

- **Unique log directories:** When running suites in parallel, give each suite its own `logdir` to avoid file conflicts. For example, under Rebar3's base log directory (`_build/logs`), create a subdirectory per suite or per node: e.g., `_build/logs/my_app_SUITE_node1`, `_build/logs/another_SUITE_node2`, etc. Pass `{logdir, "path/to/unique_logs"}` in the CT options for each run. This ensures each node writes its Common Test HTML logs separately.

Implementation in `rebar_prv_common_test.erl`:
Inside the `do/1` function (after `compile_tests`), intercept the normal execution flow. If no specific `--suite` option is given (i.e., user wants to run all), replace the single `ct:run_test(CTOpts)` call with a parallel execution routine. Pseudocode outline:

```
Suites = find_all_suites(TestApps, InDirs),
ok = maybe_cover_compile(State, RawOpts),  % instrument once on master
% Spawn and run each suite on a separate node:
Results = [ begin
              {ok, Node} = slave:start(Host, SuiteNameAtom, SlaveArgs),
              cover:start(Node),  % attach node to cover
              %% Run test suite (using ct_master or rpc call):
              Monitor = erlang:spawn_monitor(fun() ->
                          Res = rpc:call(Node, ct, run_test, [[{suite,
[SuiteMod]}, {logdir, SuiteLogDir}]]),
                          ParentPid ! {self(), SuiteMod, Res}
                        end),
              Monitor
            end || SuiteMod <- Suites ],
% Wait for all results
FailedSuites = collect_failed_suite_names(Results),
...
rebar_prv_cover:maybe_write_coverdata(State, ?PROVIDER),
... handle failures ...
```

In the above, `Suites` is a list of suite module atoms or file paths, and `FailedSuites` would be a list you build by inspecting each result tuple for failures.

3. Collecting Results and Identifying Failed Suites

After initiating all suite runs, you need to gather their outcomes and report any failures:

- **Synchronize on test completion:** If using `ct_master:run_test/2` (async), you can poll `ct_master:progress()` periodically until all nodes report `finished`. Alternatively, monitor the slave nodes' `ct_run` processes. If using the RPC approach, as shown, you can `receive` the

`{SuiteMod, Res}` messages from each spawned process (or simply collect the returned values if you spawned and waited in that local process).

- **Detect failures:** Each result `Res` from `ct:run_test` will be either `{Passed, Failed, {Skipped, AutoSkipped}}` or a list of such tuples. A non-zero `Failed` count or any `AutoSkipped` indicates that suite had failing tests. For example, a result `{10,2,{0,0}}` means 2 tests failed. Record the suite name in a list of failed suites if so.
- **Output failed suite names:** Modify the console output to list which suites failed. You can extend Rebar3's result handling to print the suite names that had failures. For instance, after gathering all results, if `FailedSuites = [suiteA_SUITE, suiteB_SUITE]`, you might log a message like: `"Test failures in suites: suiteA_SUITE, suiteB_SUITE."`

In code, you could implement a helper like:

```
collect_failed_suite_names(ResultsList) ->
    [ Suite
      || {Suite, {_, Failed, {_, AutoSkipped}}} <- ResultsList,
         Failed > 0 orelse AutoSkipped > 0 ].
```

And then report these suites. This ensures the user knows exactly which suites did not pass, addressing the question *"how would I know which all suites are failed?"*. You may integrate this into `format_result/1` or after the tests in `handle_quiet_results` for final output. (Ensure to only print when verbose is off, or integrate with the existing quiet result handling.)

4. Clean Up and Final Steps

After tests complete and results are processed:

- **Stop slave nodes:** It's good practice to shut down the slave nodes once their work is done. You can use `slave:stop(Node)` for each or rely on them terminating when the master exits (though explicitly stopping is cleaner). Ensure this happens **after** coverage data is collected/exported. If using cover's distributed mode, consider calling `cover:stop(Node)` for each to gracefully detach them from cover (or at least `cover:flush(Node)` to fetch any remaining counters) [1]. Then terminate the node. This prevents lingering beam processes.
- **Maintain existing behaviors:** Keep other CT options working. If the user passes `--suite=X` or other filters, you might bypass the parallel mechanism (since it's specifically for running *all* suites in parallel). Also preserve the non-parallel behavior as a fallback or behind a config flag if needed.

5. Summary of Rebar3 Code Changes

In summary, the key modifications in the Rebar3 repository would be:

1. **Identify suites and override default dir-based run:** In `rebar_prv_common_test:do/1`, if no specific suite is provided, gather all suite names (e.g., by scanning test dirs). Do not pass `{dir, Dirs}` to CT; instead prepare to run each `{suite, Suite}` separately in parallel.

2. **Initialize distribution:** Ensure the main process is a distributed node (call `net_kernel:start/1` if needed) and set a cookie for authentication.
3. **Spawn nodes per suite:** Use `slave:start/3` to start an Erlang node for each test suite. Include `-pa` for ebin paths and `-setcookie` in the args. After spawning, attach it to the cover server (`cover:start(Node)`).
4. **Run tests on slaves:** Invoke the test on each node (via `ct_master:run_test(Node, Opts)` or `rpc:call(Node, ct, run_test, [Opts])`). Construct `Opts` to include the single suite and a unique `logdir` (as well as any other CT options from `RawOpts` that are relevant, e.g. config files or timeouts). Launch all suites concurrently (e.g., spawn a process per suite to call RPC).
5. **Aggregate results:** Collect the results from each suite execution. Determine which suites had failures. Then use Rebar3's output mechanisms to report failures. For example, you might accumulate a count of total failures (to set the exit code as Rebar3 normally does) and also list the failed suite names in the console output for clarity.
6. **Finalize coverage and cleanup:** Call `rebar_prv_cover:maybe_write_coverdata(State, Name)` once after all suites finish, so that one combined coverage report is produced (e.g., a single `ct.coverdata` file covering all nodes). Finally, stop the slave nodes (and possibly call `cover:stop(Node)` on each before stopping if you want to ensure cover flushes data, though if `maybe_write_coverdata` has run, the data should have been collected).

By implementing the above changes, Rebar3 will run each Common Test suite in parallel on its own Erlang node, instrument code only once for coverage, and provide clear feedback on which suites failed. This aligns with the user's requirements: each suite runs in parallel (no multiple suites per node), coverage is handled efficiently one-time upfront, and the output will indicate any suite failures.

References: Rebar3 uses Common Test's distributed features and Erlang's slave nodes to achieve this. The Common Test Master documentation confirms that we must start and manage slave nodes ourselves [2], and the cover tool's docs confirm that cover can aggregate data from multiple nodes when set up as described [1]. Additionally, the `ct_master:run_test/2` API is used to spawn tests on specific nodes [3], which is leveraged in our implementation. These changes involve modifying the `rebar_prv_common_test` module (and possibly adding helper functions) in the Rebar3 codebase as outlined above.

---

[1] Erlang -- cover
https://www.erlang.org/docs/19/man/cover

[2] Using Common Test for Large-Scale Testing — common_test v1.27.5
https://www.erlang.org/doc/apps/common_test/ct_master_chapter.html

[3] Erlang -- ct_master
https://www.erlang.org/docs/21/man/ct_master