DaLi - Concurrency Model

1 Introduction

In this document, I describe the concurrency model of dali, and ways to realize it on single and distributed machines.

2 The Branch Concurrency Model

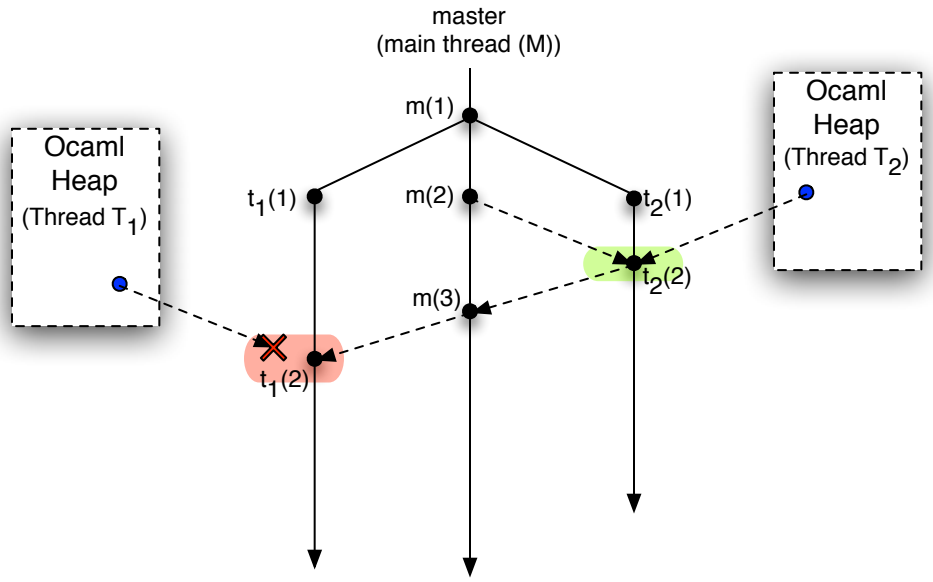

Figure 1: The branch concurrency model with a simple two-level hierarchy

A DaLi program consists of multiple concurrent threads operating on a persistent shared state. The OCaml type of the shared state distinguishes it from the thread-local state. Unlike imperative programs, where the there exists a single copy of the shared state that is *updated* by concurrent threads, DaLi programs manage shared state in a fashion similar to the version control software, i.e., by *forking* off new (concurrent) versions and *merging* them back. We refer to a totally-ordered sequence of versions as a *branch*. The program starts execution in its main thread with an initial version of the shared state, which is the first version of the *master* branch. Certain operations performed by the program (described later) result in the creation of later versions on the master branch. Forking off new threads by the program results in forking off new versions of the state along the new branches; each thread is associated a new branch. In Fig. 1 for example, main thread (M) operating on the master branch (m) creates two new threads - T_1 and T_2, which fork off two new branches - t_1 and t_2, respectively. The first versions on both the branches, i.e., $t_1(1)$ and $t_2(1)$, denote the

same state (same as the forked version $m(1)$ on the master), yet the versions themselves are deemed distinct and concurrent. Formally, $t_1(1) \neq t_2(1) \neq m(1)$, but $[\![t_1(1)]\!] = [\![t_2(1)]\!] = [\![m(1)]\!]$.

The concurrency model allows threads to periodically synchronize their branches with the master branch via the `sync_next_version` operation. The operation is so named because it synchronizes the immediate next version on the caller thread's branch with a version on the master that is later than (or same as) the last master version known to the branch. Concurrent threads that only serve read requests are expected to periodically sync with the master to contain the staleness of the data served. However, threads may also wish to write new data and make it available globally. For this purpose, `sync_next_version` allows the caller to (optionally) pass an argument proposing the next version of the state. If there haven't been any later versions on the master since the last known version, the proposal is accepted and published as the next version on the branch, and also the master. If there have been later master versions, then latest such version is merged with the proposed version and published. For this reason, applications are expected to implement a merge function for the type of the state. In Fig. 1, thread T_2 proposes a new version (the blue dot), which is merged with the version $m(2)$ of the master (merge highlighted in the green background), and the result is published as the version $t_2(2)$ on the t_2 branch and $m(3)$ on the master branch. It may sometimes transpire that the version proposed by the thread may be in conflict with the latest version on the master, in which case the merge function is allowed to return an error value and fail. This scenario is captured in Fig. 1 in the red background. The failure of merge results in the thread's proposal not being accepted, and master's version being published as the next version on the branch. In Fig. 1, the failure of merge in T_1 results in master's $m(3)$ getting published as $t_1(2)$ on t_1. Note that this version denotes the same state as present on the t_2 branch, which incorporates T_2's proposal, thus demonstrating the information flow among the branches.

The threads can access the lastest version on their branches via the `get_latest_version` operation. If the last merge operation is a success, then this operation returns the merged version, allowing the thread to continue its computation. If the last merge operation failed, then it allows the thread to read the latest version of the master and revise its proposal for the next `sync` operation.

The behavior of `sync_next_version` and `get_latest_version` is more-or-less same in the main thread. While `sync` without a propoisal is effectively a `nop` in the main thread, a `sync` with a proposal tries to merge the proposal with the lastest master version, which happens to be the local version for the main thread. As with other threads, `sync` fails if the proposed version is incompatible with the later version. There is however a slight difference in behavior of the `get_latest_version` in the main thread; unlike in other threads, successive calls to `get_latest_version` in the main thread may return different results even if they are not interleaved by a `sync` operation. This happens if one of the child threads syncs its branch with the master inbetween the two calls.

2.1 Generalizing to branch hierarchies of any depth

Observe that the branch concurrency model desribed above confines branching to a 2-level hierarchy with the master branch at the top and forked branches at the bottom. The model does not define the behavior of the version/thread fork operation originating from one of the child threads. We now generalize the model for branch hierarchies of any depth, while also eliminating the slight asymmetry between the main thread and the child threads described above. We achieve this generalization by redefining the `sync_next_version` operation as a two-step process that first merges the proposed version with the latest version on the current branch, and then merges the result with the latest version on the parent branch (note: not necessarily the master branch). The additional first step is now necessary because the latest version on the current branch may not be same as the one that was last read through the `get_latest_version`; a new version may have since been committed by one of the child threads of the current thread. The first step merges the proposed version with this version. The second stepe merges the result of the first step with the lastest version on the master, and publishes the merge result on both the branches.

In Fig. 2, thread T_1 forks a child T_2, which forks T_3. The initial versions on the corresponding branches are $t_1(1)$, $t_2(1)$, and $t_3(1)$, respectively. Thread T_2 starts a local computation on its initial version copied into its heap (the red dot). At some point, T_3 publishes a later version ($t_2(2)$) to its parent. A little while

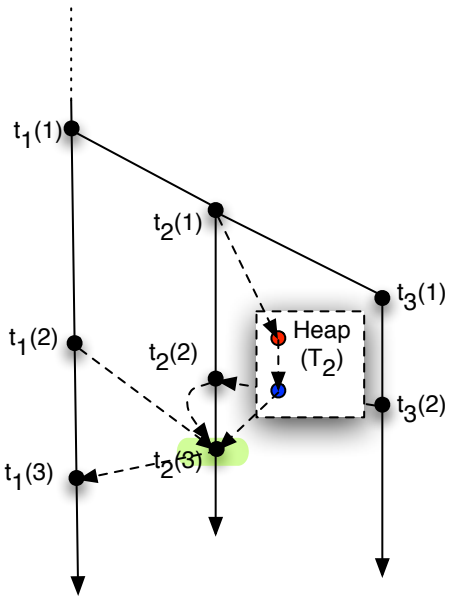

Figure 2: The branch concurrency model generalized to hierarchies of any depth

later, T_2 concludes its local computation and proposes a candidate for the new version on t_2 (the blue dot; the dotted line connecting the red and blue dots denotes data dependency). This candidate is merged with the latest local version ($t_2(2)$), and also the latest parent version ($t_1(2)$), and the merge result[1] is published as $t_2(3)$ on the local branch and $t_1(3)$ on the parent branch. Subsequence `get_latest_version` returns these versions on both the branches.

Observe that the behavior of `sync_next_version` and `get_latest_version` is now uniform across all the threads. The first and second steps (resp.) of `sync_next_version` are effectively nop for the root and leaves of the thread hierarchy, because the master branch doesn't have a parent and the bottom-most branch doesn't have a child. the main thread doesn't have a parent,

2.2 Blocking sync only when necessary

One problem with the previous model is that the call to `sync_later_version` always blocks until the merge operation succeeds and the result is published on both the branches. This is unnecessary if the merge operation is always guaranteed to succeed (for e.g., in the case of a mergeable queue). A non-blocking `sync` is more appropriate for such cases. Selective non-blocking behavior for can be accommodated in the semantics of `sync` as following: Along with the proposal for the later version, `sync` also accepts the current continuation of the caller. If there is a possiblity that any of the two merge operations might fail, then `sync` waits until it obtains the result of merges before calling the continuation with an appropriate status. Otherwise, it calls the current continuation immediately, reporting success status. Some subsequent (but not necessarily immediate) call to `get_latest_version` is guaranteed to return the merged version[2].

But how does the `sync` know if the merge might return conflict? By observing the diff (`git diff`) of the proposed and the latest local versions w.r.t to the last known version of the parent. For example, let us consider an application whose shared state contains three data items - `x`, `y`, and `z`, which are indepedently mergeable. Suppose that the type of the merge function for `z` indicates a possibility of conflict (`type 'a result = Ok of 'a | Conflict`), whereas the merge functions for `x` and `y` do not admit that possibility.

[1]Convergence requires merge to be commutative, so the order of merging doesn't matter.

[2]This guarantee is conditional on the EC guarantee of the underlying machine and the commutativity property of the merge function.

Syntax

$$
\begin{array}{rcll}
 & & t \in \texttt{Thread Ids} & v, e \in \texttt{ML Expressions} \\
s & \in & \texttt{Expressions} & ::= \quad e \mid s \gg\!= s \mid \texttt{return}\ s \mid \texttt{run}\ s\ s \\
 & & & \quad \mid \texttt{fork}\ s \mid \texttt{pull} \mid \texttt{push}\ s \\
b & \in & \texttt{Branches} & ::= \quad [\,] \mid v :: b \\
p & \in & \texttt{Programs} & ::= \quad (s, b)_t \mid p \,||\, p
\end{array}
$$

Evaluation Contexts

$$
\begin{array}{rcl}
E & ::= & \bullet \mid \bullet \gg\!= s \mid \texttt{return}\ \bullet \mid \texttt{run}\ \bullet\ e \\
P & ::= & E_t \mid \bullet \,||\, p \mid p \,||\, \bullet
\end{array}
$$

Reduction Relation $\boxed{p;\ M \longrightarrow p';\ M'}$

$$
\begin{array}{lcll}
(\texttt{run}\ v\ s, b)_t; M & \longrightarrow & (s, [v])_t; M[t \mapsto [v]] & \\
(\texttt{fork}\ s, v :: b)_t; M & \longrightarrow & ((), v :: b)_t \,||\, (s, [v])_{t'}; M[t' \mapsto M(t)] & \texttt{where}\ t' \notin dom(M) \\
(\texttt{pull}\,, b)_t; M[t \mapsto v :: m] & \longrightarrow & (v, v :: b)_t; M[t \mapsto v :: m] & \\
(\texttt{push}\ v_1, v :: b)_t; M[t \mapsto v_2 :: m] & \longrightarrow & ((), v_1 :: v :: b)_t; M'[t \mapsto v' :: v_2 :: m] & \texttt{where}\ v' = \texttt{merge}\ v\, v_1\, v_2\ \texttt{and} \\
 & & & \forall (t' \in dom(M)).\ M'(t') = v' :: M(t')
\end{array}
$$

Figure 3: DaLi: Syntax and High-level Operational Semantics

Let the last know parent version be {`x=a; y=b; z=c`}, the latest local version be {`x=a'; y=b'; z=c`}, and the proposed version be {`x=a''; y=b; z=c`}. It is clear that the merge function on `z` will not be called while merging the versions, hence there is not possibility that the merge fails. Consequencely, `sync` can be non-blocking in this case. On the other hand, if `z` is assigned `c'` in the proposed version, then the merge on `z` will be called, which might fail. Therefore, `sync` needs to block.

The banality of selective blocking optimization belies its true importance, for the selectively blocking `sync` is what lets DaLi implement a highly available replicated data store with on-demand strong consistency, without requiring programmers to explicitly reason about the consistency.

3 Operational Semantics

$$
\begin{array}{lcll}
(\texttt{push}\ v_1, v :: b)_t; M[t \mapsto v_2 :: m] & \longrightarrow & ((), v_1 :: v :: b)_t; M[t \mapsto v' :: v_2 :: m] & \texttt{where}\ t \neq t'\ \texttt{and} \\
 & & & v = \texttt{merge}\ v\, v_1\, v_2\ \texttt{and}
\end{array}
$$

$$
\frac{t \neq t' \quad v_1 \not\preceq v_2 \wedge v_2 \not\preceq v_1 \quad v = \texttt{merge}\,(\texttt{lca}\,(M(t), M(t')), v_1, v_2)}{(s, b)_t; M[t \mapsto v_1 :: m_1][t' \mapsto v_2 :: m_2] \ \longrightarrow\ (s, b)_t; M[t \mapsto v :: v_1 :: m_1][t' \mapsto v_2 :: m_2]}
$$

Figure 4: DaLi: Low-level Operational Semantics

4 Instantiations

DaLi's concurrency model is general enough that it can be instantiated in system-specific ways.

Syntax

$$
\begin{array}{rcll}
 & & t \in \texttt{Thread Ids} & v, e \in \texttt{ML Expressions} \\
s & \in & \texttt{Expressions} & ::= \quad e \mid s \gg\!= s \mid \texttt{return}\ s \mid \texttt{run}\ s\ s \\
 & & & \quad \mid \texttt{fork}\ s \mid \texttt{pull} \mid \texttt{push}\ s \\
f & \in & \texttt{Operations} & ::= \quad \texttt{INIT} \mid \texttt{FORK}\,(b) \mid \texttt{PUSH} \mid \texttt{MERGE}\,(b) \\
b & \in & \texttt{Branches} & ::= \quad [(v, f)] \mid (v, f) :: b \\
p & \in & \texttt{Programs} & ::= \quad (s, b)_t \mid p \,||\, p
\end{array}
$$

Evaluation Contexts

$$
\begin{array}{rcl}
E & ::= & \bullet \mid \bullet \gg\!= s \mid \texttt{return}\ \bullet \mid \texttt{run}\ \bullet\ e \\
P & ::= & E_t \mid \bullet \,||\, p \mid p \,||\, \bullet
\end{array}
$$

Reduction Relation $\boxed{p;\ H \longrightarrow p';\ H'}$

$$
\begin{array}{lcll}
(\texttt{run}\ v\ s, \_)_t; H & \longrightarrow & (s, [(v, \texttt{FORK}\,[(v, \texttt{INIT})])])_t; H[t \mapsto [(v, \texttt{INIT})]] & \\
(\texttt{fork}\ s, b)_t; H & \longrightarrow & ((), b)_t \,||\, (s, [(v, \texttt{FORK}\, m_{t'})])_{t'}; H[t' \mapsto m_{t'}] & \texttt{where}\ t' \notin dom(H) \\
 & & & \texttt{and}\ m_{t'} = [(v, \texttt{FORK}\, H(t))] \\
(\texttt{push}\ v, b)_t; H & \longrightarrow & ((), (v, \texttt{PUSH}) :: b)_t; H & \\
(\texttt{pull}, \_)_t; H(t \mapsto (v, \_) :: m) & \longrightarrow & (v, [(v, \texttt{FORK}\, H(t))])_t; H &
\end{array}
$$

$$
\frac{t_1 \neq t_2 \quad v_1 \not\succeq v_2 \quad \texttt{world}\,(H, t_2) \succeq_\circ \texttt{world}\,(H, t_1) \quad v = \texttt{merge}\,(\texttt{lca}\,(H, t_1, t_2), v_1, v_2)}{(s, v_0)_t; H(t_1 \mapsto (v_1, f_1) :: m_1)(t_2 \mapsto (v_2, f_2) :: m_2) \ \longrightarrow\ (s, v_0)_t; H[t_1 \mapsto (v, \texttt{MERGE}\, H(t_2)) :: (v_1, f_1) :: m_1]}
$$

$$
\frac{\texttt{world}\,(b) \succeq_\circ \texttt{world}\,(H, t_1) \quad v = \texttt{merge}\,(\texttt{lca}\,(H, t_1, t_2), v_1, v_2)}{(s, (v, \texttt{PUSH}) :: b)_t; H \ \longrightarrow\ (s, [(v', \texttt{FORK}\ m)])_t; H[t \mapsto m]}
$$

Figure 5: DaLi: Syntax and High-level Operational Semantics

4.1 Single-node database

ToDo.

4.2 Replicated data store

ToDo.