Parallel computation has recently become necessary to take full advantage of the gains allowed by Moore’s law. For years, processor makers consistently delivered increases in clock rates and instruction-level parallelism, so that single-threaded code executed faster on newer processors with no modification. Now, to manage CPU power dissipation, processor makers favor multi-core chip designs, and software has to be written in a multi-threaded or multi-process manner to take full advantage of the hardware. Many multi-threaded development paradigms introduce overhead, and will not see a linear increase in speed vs number of processors. This is particularly true while accessing shared or dependent resources, due to lock contention. This effect becomes more noticeable as the number of processors increases.[1]

Today’s systems have to deal with many simultaneous tasks and leverage the power of multi-core CPUs. Modern smartphones have 4 core processors, modern desktop computer has 8 – 16 cores per processor. IAS services like AWS or Azure are acting like catalyst for distributed computing growth.[2] This process is going for at least 5 – 8 years now, but concurrent software is still difficult to write, debug and support. With all this explosive hardware growth, dealing with multiple processors on software level is still really hard.

Clojure is modern Lisp dialect, which specifically focuses on great out of the box concurrency support, writing concurrent application in Clojure is as easy as it should be nowadays. It achieves it using different sophisticated technics, which will be described in detail in further sections.

Language origin and features

Clojure is modern Lisp dialect. Lisp is the second-oldest high-level programming language in widespread use today; only Fortran is older (by one year). Like Fortran, Lisp has changed a great deal since its early days, and a number of dialects have existed over its history, Clojure is one of them. Rich Hickey originally created it in 2007. Its main runtime is JVM, but it has been also ported on CLR (ClojureCLR) and JavaScript runtime (ClojureScript). Hickey developed Clojure because he wanted a modern Lisp for functional programming, symbiotic with the established Java platform, and designed for concurrency.[3]

Like other Lisps, Clojure treats code as data and has a sophisticated macro system. Clojure’s syntax is built on S-expressions that are first parsed into data structures by a reader before being compiled. Clojure’s reader supports literal syntax for maps, sets and vectors in addition to lists, and these are given to the compiler as they are. In other words, the Clojure compiler does not compile only list data structures, but supports all of the mentioned types directly. Clojure is a Lisp-1, and is not intended to be code-compatible with other dialects of Lisp.

Key language features:

- Tight Java integration: By compiling into JVM Byte code, Clojure applications can be easily packaged and deployed to JVMs and application servers without added complexity. The language also provides macros which make it simple to use existing Java APIs. Clojure’s data structures all implement standard Java Interfaces, making it easy to run code implemented in Clojure from Java;
- dynamic development with a read-eval-print loop;
- functions as first-class objects;
- emphasis on recursion and higher-order functions instead of side-effect-based looping;
- lazy sequences;
- provides a rich set of immutable, persistent data structures (including hashmaps, sets and lists);
- concurrent programming through software transactional memory, an agent system, and a dynamic var system;
- multimethods to allow dynamic dispatch on the types and values of any set of arguments (cf. the usual object-oriented polymorphism which dispatches on the type of what is effectively the first method argument).

**Persistent data structures**

Clojure provides developer with prebuild set of collections – lists, maps and sets. This is regular set that any mature development platform must provide, what is special about Clojure’s versions is that all collections are immutable and persistent. Persistent data structure is a data structure that always preserves the previous version of itself when it is modified. There is no way to mutate existing Clojure collection, every operation like adding or removing element will create new copy of collection. This might sound like a huge resource waist, but in fact, it is not. It’s efficient thanks to some sophisticated algorithms that allow to achieve so called structural sharing. This is how it’s implemented for clojure’s PersistentHashMap (figure 1).

PersistentHashMap is a persistent version of the classical hash table data structure. Persistent means that the data structure is immutable, yet has efficient non-destructive operations that correspond to the operations on the classical hash table. E.g., put(K, V) in hash table corresponds to a side-effect free function assoc(P, K, V) which computes from P a new PersistentHashMap P’ which is like P except that it maps key K to value V. The word “efficient” means “on par” with their mutating counterparts. For Clojure data structures, Rich tries to make them within 1-4 of the Java data structure operations; and read-only operations can even be faster than Java’s.

![Figure 1 – Structure sharing](image)

Having immutable data significantly simplifies concurrent development, since there is no state, no race conditions, no locks and deadlocks. But in order for program to do something useful apart from scientific calculations it actually needs to have some sort of state. Clojure allows create statefull data structures through the number of prebuild mechanisms.

**Reference types**
Usually in multithreaded environments different processes tends to coordinated their access to shared resources by lock strategies, this is how it’s done is C++, Java and .Net. When one process needs to work with resource, it locks it for all other processes to ensure consistency of its state. Maintaining this lock strategy is very complicated, deadlocks (when two processes wait for each other) can occur. Clojure’s way to handle this is to provide another abstraction layer on top of locks, developer does not work directly with data he works with reference type instead. Clojure provides developer with a “box” with a value inside, every time developer tries to access value inside the box, platform will take care of everything and will ensure exclusive, consistent access to the value, just like in single threaded environment.

Clojure has different types of these boxes, which one to use depends on programmer needs (table 1).

Table 1 – (Basic) Reference Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Sharing</th>
<th>Synchronicity</th>
<th>Coordination</th>
</tr>
</thead>
<tbody>
<tr>
<td>Var</td>
<td>Isolated</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Atom</td>
<td>Shared</td>
<td>synchronous</td>
<td>autonomous</td>
</tr>
<tr>
<td>Agent</td>
<td>Shared</td>
<td>asynchronous</td>
<td>autonomous</td>
</tr>
<tr>
<td>Ref</td>
<td>Shared</td>
<td>synchronous</td>
<td>coordinated</td>
</tr>
</tbody>
</table>

Sharing answers question - can changes be seen by more than one thread? Synchronicity: Does it happen now or some time later? Coordination: Do multiply changes happen atomically?

**Vars**

Vars provide a mechanism to refer to a mutable storage location that can be dynamically rebound (to a new storage location) on a per-thread basis. Every Var can (but needn’t) have a root binding, which is a binding that is shared by all threads that do not have a per-thread binding. Thus, the value of a Var is the value of its per-thread binding, or, if it is not bound in the thread requesting the value, the value of the root binding, if any. In other words var in a single threaded mutable reference.

**Atoms**

The concept of atomicity might not be new. Atomicity implies that even in an environment with multiple actors (threads) working on the same piece of data, Clojure can guarantee that updates to that data either happen entirely, or don’t happen at all. In practice, this is used when multiple threads are reading and writing to one data structure to share its contents.

The swap! function takes the atom as its first argument, and a function to apply to the value in the atom.

Under the covers, the swap! function is doing a compare-and-set! operation, which means that if two threads are in a race to apply their function to the atom, the loser will retry their swap if the value has changed since they began the swap. Therefore, it’s very important that ‘swap!’ procedures be completely free of side-effects. No log writing, database updating or anything else - otherwise you’ll get unpredictable behavior in I/O layers.

**Refs**

Because Clojure has immutable data structures, Refs provide a mechanism to track the same fact idea (identity) over a period of time.
If atoms are for managing the fact changes of a single data structure, then Refs are for managing changes of multiple data structures together, atomically. Put simply, Refs give in memory transactions using a software transactional memory system.

Refs give the amazing power to coordinate several data structures, forcing their changes to occur together as one atomic group. Similar to atoms, if two threads attempt to change the value of a Ref at the same time, one of them will succeed and the other will fail. When the second one fails, it will be automatically retried. So, just like with Atoms, Ref transactions must be completely free of side effects.

More data structures are grouped together into a transaction, the more transaction retries will system experience under heavy write load, and the more application performance will degrade. The less data structures have to coordinate, the better.

Agents, Promises and Futures

There are a lot of other useful mechanism Clojure provides to handle concurrency efficiently. Agents allows access shared data asynchronously (but still on same machine, not like Actors in Haskell). Futures offer a simple and flexible way to dispatch operations into another thread and then retrieve the results of that computation from the existing thread. Promise is a contract between one thread and another to deliver a calculation at some point in the future. Promises are only reference types in Clojure that can create deadlocks, but these are not results of race conditions but incorrect program logic, therefore can be found very easily.

Conclusion

In the age of fixed processor’s clock speed and multi-core growth, concurrent programing still remains as one of the hardest areas of software engineering. With all evolution of developer tools, platforms and eventually skills multi-threaded applications are still very hard to write, support and maintain. Clojure is a modern approach to solve these problems. Clojure is coming from well-known Lisp family, therefor it inherits all-powerful features of the “language of languages” like macros or lambdas, it also brings new fresh ideas like Software Transaction Memory, Agents model, Persistent data structures. Being general-purpose language Clojure mostly concentrates on concurrency. Traditional Lock based approach of writing concurrent software is too complicated and error prone, Clojure provides developer with layer of abstraction on top of JVM (CLR) concurrency handling mechanisms that makes multithreaded development easy, as it should be. Software engineers can finally concentrate on program logic instead of maintaining their concurrency framework. Clojure’s way to write software is not ultimate solution to all problems around writing concurrent applications, but it is a big step in the right direction.

List of references


Bialykh Yury Yurievich, student of applied mathematics, Yanka Kupala State University of Grodno, notanone@yandex.ru.